# Oracle® AI Database

## Data Warehousing Guide

26ai
G43670-01
October 2025

Oracle AI Database Data Warehousing Guide, 26ai

G43670-01

Copyright © 2001, 2025, Oracle and/or its affiliates.

Primary Author: Frederick Kush

Contributors: Hermann Baer, Mark Bauer, Subhransu Basu, Nigel Bayliss, Donna Carver, Maria Colgan, Benoit Dageville, Luping Ding, Bud Endress, Bruce Golbus, John Haydu, Keith Laker, Paul Lane, Chun-Chieh Lin, William Lee, George Lumpkin, David McDermid, Alex Melidis, Valarie Moore, Ananth Raghavan, Jack Raitto, Andy Rivenes, Lei Sheng, Wayne Smith, Sankar Subramanian, Margaret Taft, Murali Thiyagarajan, Jean-Francois Verrier, Andreas Walter, Andy Witkowski, Min Xiao, Tsae-Feng Yu, Fred Zemke, Mohamed Ziauddin, Frederick Kush, Rachel Voirin

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

# Contents


## Preface

Audience i
Documentation Accessibility i
Related Documents i
Conventions i

## Part I Data Warehouse - Fundamentals

## 1 Introduction to Data Warehousing Concepts

1.1 What Is a Data Warehouse? 1
1.1.1 Key Characteristics of a Data Warehouse 3
1.2 Contrasting OLTP and Data Warehousing Environments 3
1.3 Common Data Warehouse Tasks 4
1.4 Data Warehouse Architectures 5
1.4.1 Data Warehouse Architecture: Basic 5
1.4.2 Data Warehouse Architecture: with a Staging Area 6
1.4.3 Data Warehouse Architecture: with a Staging Area and Data Marts 6

## 2 Data Warehousing Logical Design

2.1 Logical Versus Physical Design in Data Warehouses 1
2.2 Creating a Logical Design 1
2.2.1 What is a Schema? 2
2.3 About Third Normal Form Schemas 2
2.3.1 About Normalization 3
2.3.2 Design Concepts for 3NF Schemas 4
2.3.2.1 Identifying Candidate Primary Keys 4
2.3.2.2 Foreign Key Relationships and Referential Integrity Constraints 4
2.3.2.3 Denormalization 5
2.4 About Star Schemas 5
2.4.1 About Facts and Dimensions in Star Schemas 6
2.4.1.1 About Fact Tables in Data Warehouses 6




Copyright © 2001, 2025, Oracle and/or its affiliates.

2.4.1.2 About Dimension Tables in Data Warehouses 7
2.4.2 Design Concepts in Star Schemas 8
2.4.3 About Snowflake Schemas 10
2.5 Improved Analytics Using the In-Memory Column Store 10
2.5.1 About Improving Query Performance Using In-Memory Expressions 11
2.5.2 About Using In-Memory Virtual Columns to Improve Query Performance 12
2.5.3 About In-Memory Column Store and Automatic Data Optimization 12
2.6 Automatic Big Table Caching to Improve the Performance of In-Memory Parallel Queries 13

## 3 Data Warehousing Physical Design

3.1 Moving from Logical to Physical Design 1
3.2 About Physical Design 1
3.2.1 Physical Design Structures 2
3.2.1.1 About Tablespaces in Data Warehouses 2
3.2.1.2 About Partitioning in Data Warehouses 3
3.2.1.3 Index Partitioning in Data Warehouses 4
3.2.1.4 About Partitioning for Manageability 5
3.2.1.5 About Partitioning for Performance 5
3.2.1.6 About Partitioning for Availability 5
3.2.2 About Views in Data Warehouses 6
3.2.3 About Integrity Constraints in Data Warehouses 6
3.2.4 About Indexes and Partitioned Indexes in Data Warehouses 6
3.2.5 About Materialized Views in Data Warehouses 7
3.2.6 About Dimensions in Data Warehouses 7
3.2.6.1 About Dimension Hierarchies 7
3.2.6.2 Typical Dimension Hierarchy 8

## 4 Data Warehousing Optimizations and Techniques

4.1 Using Indexes in Data Warehouses 1
4.1.1 About Using Bitmap Indexes in Data Warehouses 1
4.1.1.1 About Bitmap Indexes and Nulls 2
4.1.1.2 About Bitmap Indexes on Partitioned Tables 2
4.1.2 Benefits of Indexes for Data Warehousing Applications 3
4.1.3 About Cardinality and Bitmap Indexes 3
4.1.4 How to Determine Candidates for Using a Bitmap Index 5
4.1.5 Using Bitmap Join Indexes in Data Warehouses 5
4.1.5.1 Four Join Models for Bitmap Join Indexes in Data Warehouses 6
4.1.5.2 Bitmap Join Index Restrictions and Requirements 8
4.1.6 Using B-Tree Indexes in Data Warehouses 8


Copyright © 2001, 2025, Oracle and/or its affiliates.

4.1.7 Using Index Compression 9
4.1.8 Choosing Between Local Indexes and Global Indexes 9
4.2 Using Integrity Constraints in a Data Warehouse 10
4.2.1 Overview of Constraint States 10
4.2.2 Typical Data Warehouse Integrity Constraints 11
4.2.2.1 UNIQUE Constraints in a Data Warehouse 11
4.2.2.2 FOREIGN KEY Constraints in a Data Warehouse 12
4.2.2.3 RELY Constraints in a Data Warehouse 13
4.2.2.4 NOT NULL Constraints in a Data Warehouse 13
4.2.2.5 Integrity Constraints and Parallelism in a Data Warehouse 13
4.2.2.6 Integrity Constraints and Partitioning in a Data Warehouse 14
4.2.2.7 View Constraints in a Data Warehouse 14
4.3 About Parallel Execution in Data Warehouses 14
4.3.1 Why Use Parallel Execution? 15
4.3.1.1 When to Implement Parallel Execution 15
4.3.1.2 When Not to Implement Parallel Execution 16
4.3.2 Automatic Degree of Parallelism and Statement Queuing 16
4.3.3 About In-Memory Parallel Execution in Data Warehouses 17
4.4 About Optimizing Storage Requirements in Data Warehouses 18
4.4.1 Using Data Compression to Improve Storage in Data Warehouses 18
4.5 Optimizing Star Queries and 3NF Schemas 19
4.5.1 Optimizing Star Queries 19
4.5.1.1 Tuning Star Queries 19
4.5.2 Using Star Transformation 19
4.5.2.1 Star Transformation with a Bitmap Index 20
4.5.2.2 Execution Plan for a Star Transformation with a Bitmap Index 21
4.5.2.3 Star Transformation with a Bitmap Join Index 22
4.5.2.4 Execution Plan for a Star Transformation with a Bitmap Join Index 22
4.5.2.5 How Oracle Chooses to Use Star Transformation 23
4.5.2.6 Star Transformation Restrictions 23
4.5.3 Optimizing Third Normal Form Schemas 24
4.5.3.1 3NF Schemas: Partitioning 24
4.5.3.2 3NF Schemas: Parallel Query Execution 27
4.5.4 Optimizing Star Queries Using VECTOR GROUP BY Aggregation 28
4.6 About Approximate Query Processing 29
4.6.1 Running Queries Containing Exact Functions Using SQL Functions that Return Approximate Values 30
4.7 About Approximate Top-N Query Processing 31

Part II Optimizing Data Warehouses


Copyright © 2001, 2025, Oracle and/or its affiliates.

# 5 Basic Materialized Views


5.1 Overview of Data Warehousing with Materialized Views 1
5.1.1 About Materialized Views for Data Warehouses 2
5.1.2 About Materialized Views for Distributed Computing 2
5.1.3 About Materialized Views for Mobile Computing 2
5.1.4 The Need for Materialized Views 2
5.1.5 Components of Summary Management 4
5.1.6 Data Warehousing Terminology 5
5.1.7 About Materialized View Schema Design 6
5.1.7.1 Schemas and Dimension Tables 6
5.1.7.2 Guidelines for Materialized View Schema Design 7
5.1.8 About Loading Data into Data Warehouses 8
5.1.9 Overview of Materialized View Management Tasks 9
5.2 Types of Materialized Views 10
5.2.1 About Materialized Views with Aggregates 10
5.2.1.1 Requirements for Using Materialized Views with Aggregates 12
5.2.2 About Materialized Views Containing Only Joins 13
5.2.2.1 Materialized Join Views FROM Clause Considerations 14
5.2.3 About Nested Materialized Views 15
5.2.3.1 Why Use Nested Materialized Views? 15
5.2.3.2 About Nesting Materialized Views with Joins and Aggregates 16
5.2.3.3 Nested Materialized View Usage Guidelines 16
5.2.3.4 Restrictions When Using Nested Materialized Views 17
5.3 Creating Materialized Views 17
5.3.1 Creating Materialized Views with Column Alias Lists 19
5.3.2 Creating Materialized Views Based on Hybird Partitioned Tables 20
5.3.3 About Materialized Views Names 20
5.3.4 About Storage And Table Compression for Materialized Views 21
5.3.5 About Build Methods for Materialized Views 21
5.3.6 About Enabling Query Rewrite for Materialized Views 22
5.3.7 About Query Rewrite Restrictions 22
5.3.7.1 About Materialized View Restrictions for Query Rewrite 22
5.3.7.2 General Query Rewrite Restrictions 23
5.3.8 About Refresh Options for Materialized Views 23
5.3.8.1 About Refresh Modes for Materialized Views 24
5.3.8.2 About Types of Materialized View Refresh 24
5.3.8.3 About Using Trusted Constraints and Materialized View Refresh 25
5.3.8.4 General Restrictions on Fast Refresh 26
5.3.8.5 Restrictions on Fast Refresh on Materialized Views with Joins Only 26
5.3.8.6 Restrictions on Fast Refresh on Materialized Views with Aggregates 27
5.3.8.7 Restrictions on Fast Refresh on Materialized Views with UNION ALL 28




Copyright © 2001, 2025, Oracle and/or its affiliates.

5.3.8.8 About Achieving Refresh Goals 29
5.3.8.9 Refreshing Nested Materialized Views 29
5.3.9 ORDER BY Clause in Materialized Views 30
5.3.10 Using Oracle Enterprise Manager to Create Materialized Views 30
5.3.11 Using Materialized Views with NLS Parameters 30
5.3.12 Adding Comments to Materialized Views 31
5.4 Creating Materialized View Logs 31
5.4.1 Using the FORCE Option With Materialized View Logs 32
5.4.2 Purging Materialized View Logs 32
5.5 Creating Materialized Views Based on Approximate Queries 33
5.6 Creating a Materialized View Containing Bitmap-based COUNT(DISTINCT) Functions 34
5.7 Registering Existing Materialized Views 34
5.8 Choosing Indexes for Materialized Views 36
5.9 Dropping Materialized Views 36
5.10 Analyzing Materialized View Capabilities 37
5.10.1 Using the DBMS_MVIEW.EXPLAIN_MVIEW Procedure 37
5.10.1.1 DBMS_MVIEW.EXPLAIN_MVIEW Declarations 38
5.10.1.2 Using MV_CAPABILITIES_TABLE 38
5.10.1.3 MV_CAPABILITIES_TABLE.CAPABILITY_NAME Details 40
5.10.1.4 MV_CAPABILITIES_TABLE Column Details 42

## 6 Advanced Materialized Views

6.1 About Partitioning and Materialized Views 1
6.1.1 About Partition Change Tracking 1
6.1.1.1 About Partition Key and Partition Change Tracking 3
6.1.1.2 About Join Dependent Expression and Partition Change Tracking 3
6.1.1.3 About Partition Markers and Partition Change Tracking 4
6.1.1.4 About Partial Rewrite in Partition Change Tracking 5
6.1.2 Partitioning a Materialized View 5
6.1.3 Partitioning a Prebuilt Table 6
6.1.3.1 Benefits of Partitioning a Materialized View 6
6.1.4 Rolling Materialized Views 7
6.1.5 About Automatic Partitioning of Materialized Views 7
6.2 About Materialized Views in Analytic Processing Environments 7
6.2.1 About Materialized Views and Analytic Views 8
6.2.2 About Materialized Views and Hierarchical Cubes 8
6.2.3 Benefits of Partitioning Materialized Views 9
6.2.4 About Compressing Materialized Views 9
6.2.5 About Materialized Views with Set Operators 9
6.2.5.1 Examples of Materialized Views Using UNION ALL 10
6.3 About Materialized Views and Models 11


Copyright © 2001, 2025, Oracle and/or its affiliates.

6.4 About Security Issues with Materialized Views 11
6.4.1 Querying Materialized Views with Virtual Private Database (VPD) 12
6.4.1.1 Using Query Rewrite with Virtual Private Database 12
6.4.1.2 Restrictions with Materialized Views and Virtual Private Database 13
6.5 Invalidating Materialized Views 13
6.6 Altering Materialized Views 14
6.7 Using Real-time Materialized Views 14
6.7.1 Overview of Real-time Materialized Views 15
6.7.1.1 Restrictions on Using Real-time Materialized Views 16
6.7.1.2 About Accessing Real-time Materialized Views 16
6.7.2 Creating Real-time Materialized Views 16
6.7.3 Converting an Existing Materialized View into a Real-time Materialized View 18
6.7.4 Enabling Query Rewrite to Use Real-time Materialized Views 19
6.7.5 Using Real-time Materialized Views During Query Rewrite 19
6.7.6 Using Real-time Materialized Views for Direct Query Access 21
6.7.7 Listing Real-time Materialized Views 23
6.7.8 Improving Real-time Materialized Views Performance 23

## 7 Refreshing Materialized Views

7.1 About Refreshing Materialized Views 1
7.1.1 About Complete Refresh for Materialized Views 3
7.1.2 About Fast Refresh for Materialized Views 4
7.1.3 About Partition Change Tracking (PCT) Refresh for Materialized Views 4
7.1.4 About Logical Partition Change Tracking (LPCT) Refresh for Materialized Views 4
7.1.5 About the Out-of-Place Refresh Option 6
7.1.5.1 Types of Out-of-Place Refresh 6
7.1.5.2 Restrictions and Considerations with Out-of-Place Refresh 7
7.1.6 About ON COMMIT Refresh for Materialized Views 7
7.1.7 About ON STATEMENT Refresh for Materialized Views 7
7.1.8 About Manual Refresh Using the DBMS_MVIEW Package 8
7.1.9 Refreshing Specific Materialized Views with REFRESH 9
7.1.10 Refreshing All Materialized Views with REFRESH_ALL_MVIEWS 10
7.1.11 Refreshing Dependent Materialized Views with REFRESH_DEPENDENT 11
7.1.12 About Using Job Queues for Refresh 12
7.1.13 When Fast Refresh is Possible 12
7.1.14 Refreshing Materialized Views Based on Approximate Queries 12
7.1.15 About Concurrent Refresh of On-Commit Materialized Views 13
7.1.16 About Refreshing Dependent Materialized Views During Online Table Redefinition 14
7.1.17 Recommended Initialization Parameters for Parallelism 15
7.1.18 Monitoring a Refresh 15


Copyright © 2001, 2025, Oracle and/or its affiliates.

7.1.19 Checking the Status of a Materialized View 16
7.1.19.1 Examples of Using Views to Determine Freshness 17
7.1.20 Scheduling Refresh of Materialized Views 18
7.2 Tips for Refreshing Materialized Views 19
7.2.1 Tips for Refreshing Materialized Views with Aggregates 19
7.2.2 Tips for Refreshing Materialized Views Without Aggregates 21
7.2.3 Tips for Refreshing Nested Materialized Views 22
7.2.4 Tips for Fast Refresh with UNION ALL 23
7.2.5 Tips for Fast Refresh with Commit SCN-Based Materialized View Logs 23
7.2.6 Tips After Refreshing Materialized Views 24
7.3 Using Materialized Views with Partitioned Tables 24
7.3.1 Materialized View Fast Refresh with Partition Change Tracking 24
7.3.1.1 PCT Fast Refresh for Materialized Views: Scenario 1 24
7.3.1.2 PCT Fast Refresh for Materialized Views: Scenario 2 26
7.3.1.3 PCT Fast Refresh for Materialized Views: Scenario 3 26
7.4 Refreshing Materialized Views Based on Hybrid Partitioned Tables 27
7.5 Using Partitioning to Improve Data Warehouse Refresh 28
7.5.1 Data Warehouse Refresh Scenarios 31
7.5.2 Scenarios for Using Partitioning for Refreshing Data Warehouses 32
7.5.2.1 Partitioning for Refreshing Data Warehouses: Scenario 1 32
7.5.2.2 Partitioning for Refreshing Data Warehouses: Scenario 2 33
7.6 Optimizing DML Operations During Refresh 33
7.6.1 Implementing an Efficient MERGE Operation 33
7.6.2 Maintaining Referential Integrity in Data Warehouses 36
7.6.3 Purging Data from Data Warehouses 36

## 8 Synchronous Refresh

8.1 About Synchronous Refresh for Materialized Views 1
8.1.1 What Is Synchronous Refresh? 1
8.1.2 Why Use Synchronous Refresh? 2
8.1.3 Registering Tables and Materialized Views for Synchronous Refresh 3
8.1.4 Specifying Change Data for Refresh 3
8.1.5 Synchronous Refresh Preparation and Execution 4
8.1.6 Materialized View Eligibility Rules and Restrictions for Synchronous Refresh 4
8.1.6.1 Synchronous Refresh Restrictions: Partitioning 5
8.1.6.2 Synchronous Refresh Restrictions: Refresh Options 5
8.1.6.3 Synchronous Refresh Restrictions: Constraints 5
8.1.6.4 Synchronous Refresh Restrictions: Tables 5
8.1.6.5 Synchronous Refresh Restrictions: Materialized Views 6
8.1.6.6 Synchronous Refresh Restrictions: Materialized Views with Aggregates 6
8.2 Using Synchronous Refresh for Materialized Views 6


Copyright © 2001, 2025, Oracle and/or its affiliates.

8.2.1 Synchronous Refresh Step 1: Registration Phase 7
8.2.2 Synchronous Refresh Step 2: Synchronous Refresh Phase 8
8.2.3 Synchronous Refresh Step 3: The Unregistration Phase 8
8.3 Using Synchronous Refresh Groups 9
8.3.1 Examples of Common Actions with Synchronous Refresh Groups 10
8.3.2 Examples of Working with Multiple Synchronous Refresh Groups 11
8.4 Specifying and Preparing Change Data for Synchronous Refresh 12
8.4.1 Working with Partition Operations While Capturing Change Data for Synchronous Refresh 12
8.4.2 Working with Staging Logs While Capturing Change Data for Synchronous Refresh 14
8.4.2.1 About the Staging Log Key 15
8.4.2.2 About Staging Log Rules 15
8.4.2.3 About Columns Being Updated to NULL 16
8.4.2.4 Examples of Working with Staging Logs 16
8.4.2.5 Error Handling in Preparing Staging Logs 18
8.5 Troubleshooting Synchronous Refresh Operations 18
8.5.1 Overview of the Status of Refresh Operations 19
8.5.2 How PREPARE_REFRESH Sets the STATUS Fields 19
8.5.3 Examples of Preparing for Synchronous Refresh Using PREPARE_REFRESH 20
8.5.4 How EXECUTE_REFRESH Sets the Status Fields During Synchronous Refresh 21
8.5.5 Examples of Executing Synchronous Refresh Using EXECUTE_REFRESH 23
8.5.6 Example of EXECUTE_REFRESH with Constraint Violations 26
8.6 Performing Synchronous Refresh Eligibility Analysis 27
8.6.1 Using SYNCREF_TABLE to Store the Results of Synchronous Refresh Eligibility Analysis 28
8.6.2 Using a VARRAY to Store the Results of Synchronous Refresh Eligibility Analysis 28
8.6.3 Demo Scripts 29
8.7 Overview of Synchronous Refresh Security Considerations 29

## 9 Monitoring Materialized View Refresh Operations

9.1 About Materialized View Refresh Statistics 1
9.2 Overview of Managing Materialized View Refresh Statistics 2
9.3 About Data Dictionary Views that Store Materialized View Refresh Statistics 2
9.4 Collecting Materialized View Refresh Statistics 4
9.4.1 About Collecting Materialized View Refresh Statistics 4
9.4.2 Specifying Default Settings for Collecting Materialized View Refresh Statistics 5
9.4.3 Modifying the Collection Level for Materialized View Refresh Statistics 5
9.5 Retaining Materialized View Refresh Statistics 6
9.5.1 About Retaining Materialized View Refresh Statistics 6
9.5.2 Specifying the Default Retention Period for Materialized View Refresh Statistics 7


Copyright © 2001, 2025, Oracle and/or its affiliates.

9.5.3 Modifying the Retention Period for Materialized View Refresh Statistics 7
9.6 Viewing Materialized View Refresh Statistics Settings 8
9.7 Purging Materialized View Refresh Statistics 9
9.8 Viewing Materialized View Refresh Statistics 10
9.8.1 Viewing Basic Refresh Statistics for a Materialized View 10
9.8.2 Viewing Detailed Statistics for Each Materialized View Refresh Operation 11
9.8.3 Viewing Change Data Statistics During Materialized View Refresh Operations 12
9.8.4 Viewing the SQL Statements Associated with A Materialized View Refresh Operation 13
9.9 Analyzing Materialized View Refresh Performance Using Refresh Statistics 14

## 10 Dimensions

10.1 What are Dimensions? 1
10.1.1 Requirements for Dimensions in Data Warehouses 3
10.2 Creating Dimensions 3
10.2.1 Dropping and Creating Attributes with Columns 6
10.2.2 Using Multiple Hierarchies While Creating Joins 6
10.2.3 Using Normalized Dimension Tables to Create Dimensions 7
10.3 Viewing Dimensions 8
10.3.1 Viewing Dimensions With Oracle Enterprise Manager 8
10.3.2 Viewing Dimensions With the DESCRIBE_DIMENSION Procedure 8
10.4 Using Dimensions with Constraints 9
10.5 Validating Dimensions 9
10.6 Altering Dimensions 10
10.7 Deleting Dimensions 11

## 11 Basic Query Rewrite for Materialized Views

11.1 Overview of Query Rewrite 1
11.1.1 About Query Rewrite and the Optimizer 1
11.1.2 When Does Oracle Rewrite a Query? 2
11.2 Ensuring that Query Rewrite Takes Effect 2
11.2.1 Enabling Query Rewrite for Materialized Views 2
11.2.2 About Initialization Parameters for Query Rewrite 3
11.2.3 Controlling Query Rewrite 4
11.2.4 About the Accuracy of Query Rewrite 4
11.2.5 About Privileges for Enabling Query Rewrite 5
11.2.6 Sample Schema and Materialized Views 5
11.2.7 How to Verify if Query Rewrite Occurred 6
11.3 Example of Query Rewrite 7


Copyright © 2001, 2025, Oracle and/or its affiliates.

# 12 Advanced Query Rewrite for Materialized Views


12.1 How Oracle Rewrites Queries 1
12.1.1 About Cost-Based Optimization and Query Rewrite 2
12.1.2 General Query Rewrite Methods 3
12.1.2.1 When are Constraints and Dimensions Needed for Query Rewrite? 4
12.1.3 About Checks Made by Query Rewrite 4
12.1.3.1 Join Compatibility Check for Query Rewrite 4
12.1.3.2 Data Sufficiency Check for Query Rewrite 9
12.1.3.3 Grouping Compatibility Check for Query Rewrite 9
12.1.3.4 Aggregate Computability Check for Query Rewrite 10
12.1.4 About Query Rewrite Using Dimensions 10
12.1.4.1 Benefits of Using Dimensions in a Query Rewrite Environment 10
12.1.4.2 How to Define Dimensions for Query Rewrite 10
12.2 Types of Query Rewrite 12
12.2.1 Query Rewrite Method 1: Text Match Rewrite 12
12.2.2 Query Rewrite Method 2: Join Back 13
12.2.3 Query Rewrite Method 3: Aggregate Computability 15
12.2.4 Query Rewrite Method 4: Aggregate Rollup 16
12.2.5 Query Rewrite Method 5: Rollup Using a Dimension 16
12.2.6 Query Rewrite Method 6: When Materialized Views Have Only a Subset of Data 17
12.2.6.1 Query Rewrite Definitions When Materialized Views Have Only a Subset of Data 17
12.2.6.2 Selection Categories When Materialized Views Have Only a Subset of Data 18
12.2.6.3 Examples of Query Rewrite Selection 19
12.2.6.4 About Handling of the HAVING Clause in Query Rewrite 21
12.2.6.5 About Query Rewrite When the Materialized View has an IN-List 21
12.2.7 Partition Change Tracking (PCT) Rewrite 22
12.2.7.1 PCT Rewrite Based on Range Partitioned Tables 22
12.2.7.2 PCT Rewrite Based on Range-List Partitioned Tables 24
12.2.7.3 PCT Rewrite Based on List Partitioned Tables 26
12.2.7.4 PCT Rewrite and PMARKER 28
12.2.7.5 PCT Rewrite Using Rowid as PMARKER 29
12.2.8 About Query Rewrite Using Multiple Materialized Views 30
12.3 Other Query Rewrite Considerations 38
12.3.1 About Query Rewrite Using Nested Materialized Views 38
12.3.2 About Query Rewrite in the Presence of Inline Views 39
12.3.3 About Query Rewrite Using Remote Tables 40
12.3.4 About Query Rewrite in the Presence of Duplicate Tables 40
12.3.5 About Query Rewrite Using Date Folding 42
12.3.6 About Query Rewrite Using View Constraints 43




Copyright © 2001, 2025, Oracle and/or its affiliates.

12.3.6.1 Abut View Constraints Restrictions 45
12.3.7 About Query Rewrite in the Presence of Hybrid Partitioned Tables 45
12.3.8 Query Rewrite Using Set Operator Materialized Views 46
12.3.8.1 UNION ALL Marker and Query Rewrite 47
12.3.9 About Query Rewrite in the Presence of Grouping Sets 49
12.3.9.1 About Query Rewrite When Using GROUP BY Extensions 49
12.3.9.2 Hint for Rewriting Queries with Extended GROUP BY 52
12.3.10 Query Rewrite in the Presence of Window Functions 52
12.3.11 Query Rewrite and Expression Matching 53
12.3.11.1 Query Rewrite Using Partially Stale Materialized Views 53
12.3.12 Bind Variables During Query Rewrite 56
12.3.13 Handling Expressions in Query Rewrite 57
12.4 Advanced Query Rewrite Using Equivalences 57
12.5 Creating Result Cache Materialized Views with Equivalences 60
12.6 Query Rewrite and Materialized Views Based on Approximate Queries 62
12.7 Query Rewrite and Materialized Views Based on Bitmap-based COUNT(DISTINCT) Functions 65
12.8 Verifying that Query Rewrite has Occurred 66
12.8.1 Using EXPLAIN PLAN with Query Rewrite 67
12.8.2 Using the EXPLAIN_REWRITE Procedure with Query Rewrite 67
12.8.2.1 DBMS_MVIEW.EXPLAIN_REWRITE Syntax 68
12.8.2.2 Using REWRITE_TABLE to View EXPLAIN_REWRITE Output 68
12.8.2.3 Using a Varray to View EXPLAIN_REWRITE Output 69
12.8.2.4 EXPLAIN_REWRITE Benefit Statistics 71
12.8.2.5 Support for Query Text Larger than 32KB in EXPLAIN_REWRITE 71
12.8.2.6 About EXPLAIN_REWRITE and Multiple Materialized Views 72
12.8.2.7 About EXPLAIN_REWRITE Output 72
12.9 Design Considerations for Improving Query Rewrite Capabilities 73
12.9.1 Query Rewrite Considerations: Constraints 74
12.9.2 Query Rewrite Considerations: Dimensions 74
12.9.3 Query Rewrite Considerations: Outer Joins 74
12.9.4 Query Rewrite Considerations: Text Match 74
12.9.5 Query Rewrite Considerations: Aggregates 74
12.9.6 Query Rewrite Considerations: Grouping Conditions 74
12.9.7 Query Rewrite Considerations: Expression Matching 75
12.9.8 Query Rewrite Considerations: Date Folding 75
12.9.9 Query Rewrite Considerations: Statistics 75
12.9.10 Query Rewrite Considerations: Hints 75
12.9.10.1 Query Rewrite: REWRITE and NOREWRITE Hints 75
12.9.10.2 Query Rewrite: REWRITE_OR_ERROR Hint 76
12.9.10.3 Query Rewrite: Multiple Materialized View Rewrite Hints 76
12.9.10.4 Query Rewrite: EXPAND_GSET_TO_UNION Hint 76


Copyright © 2001, 2025, Oracle and/or its affiliates.

12.10 About Semi-Join Materialized View Rewrite 76

13 Working With Automatic Materialized Views

13.1 Overview of Automatic Materialized Views 1
13.2 Workload Information Provided by the Object Activity Tracking System 2
13.3 Data Dictionary Views That Provide Information About Automatic Materialized Views and OATS 2
13.4 The DBMS_AUTO_MV Package 4
13.5 The DBMS_ACTIVITY Package 12

14 Attribute Clustering

14.1 About Attribute Clustering 1
14.1.1 Methods of Clustering Data 1
14.1.2 Types of Attribute Clustering 2
14.1.2.1 Attribute Clustering with Linear Ordering 2
14.1.2.2 Attribute Clustering with Interleaved Ordering 3
14.1.3 Example: Attribute Clustered Table 3
14.1.4 Guidelines for Using Attribute Clustering 4
14.1.5 Advantages of Attribute-Clustered Tables 4
14.1.6 About Defining Attribute Clustering for Tables 5
14.1.7 About Specifying When Attribute Clustering Must be Performed 6
14.2 Attribute Clustering Operations 6
14.2.1 Privileges for Attribute-Clustered Tables 7
14.2.2 Creating Attribute-Clustered Tables with Linear Ordering 7
14.2.2.1 Examples of Attribute Clustering with Linear Ordering 7
14.2.3 Creating Attribute-Clustered Tables with Interleaved Ordering 8
14.2.3.1 Examples of Attribute Clustering with Interleaved Ordering 8
14.2.4 Maintaining Attribute Clustering 10
14.2.4.1 Adding Attribute Clustering to an Existing Table 10
14.2.4.2 Modifying Attribute Clustering Definitions 11
14.2.4.3 Dropping Attribute Clustering for an Existing Table 11
14.2.4.4 Using Hints to Control Attribute Clustering for DML Operations 11
14.2.4.5 Overriding Table-level Settings for Attribute Clustering During DDL Operations 11
14.2.4.6 Clustering Table Data During Online Table Redefinition 12
14.3 Viewing Attribute Clustering Information 13
14.3.1 Determining if Attribute Clustering is Defined for Tables 13
14.3.2 Viewing Attribute-Clustering Information for Tables 14
14.3.3 Viewing Information About the Columns on Which Attribute Clustering is Performed 14


Copyright © 2001, 2025, Oracle and/or its affiliates.

14.3.4 Viewing Information About Dimensions and Joins on Which Attribute Clustering is Performed 15
14.4 About Automatic Data Clustering 15
14.4.1 User Controls for Automatic Data Clustering 16

## 15 Using Zone Maps

15.1 About Zone Maps 1
15.1.1 Difference Between Zone Maps and Indexes 2
15.1.2 Zone Maps and Attribute Clustering 2
15.1.3 Types of Zone Maps 2
15.1.4 Benefits of Zone Maps 2
15.1.5 Scenarios Which Benefit from Zone Maps 3
15.1.6 About Maintaining Zone Maps 3
15.1.6.1 Operations that Require Zone Map Maintenance 4
15.1.6.2 Scenarios in Which Zone Maps are Automatically Refreshed 4
15.2 Zone Map Operations 5
15.2.1 Privileges Required for Zone Maps 5
15.2.2 Creating Zone Maps 6
15.2.2.1 Creating Zone Maps with Attribute Clustering 6
15.2.2.2 Creating Zone Maps Independent of Attribute Clustering 8
15.2.3 About Automatic Zone Maps 9
15.2.4 About the DBMS_AUTO_ZONEMAP Package 9
15.2.4.1 CONFIGURE Procedure 10
15.2.4.2 ACTIVITY_REPORT Function 11
15.2.4.3 Viewing Information About Automatic Zone Maps 13
15.2.5 Modifying Zone Maps 13
15.2.6 Dropping Zone Maps 14
15.2.7 Compiling Zone Maps 14
15.2.8 Controlling the Use of Zone Maps 15
15.2.8.1 Controlling Zone Map Usage for Entire SQL Workloads 15
15.2.8.2 Controlling Zone Map Usage for Specific SQL Statements 15
15.2.9 Maintaining Zone Maps 16
15.2.9.1 Zone Map Maintenance Considerations 17
15.3 Refresh and Staleness of Zone Maps 18
15.3.1 About Staleness of Zone Maps 18
15.3.2 About Refreshing Zone Maps 20
15.3.3 Refreshing Zone Maps 21
15.3.3.1 Refreshing Zone Maps Using the ALTER MATERIALIZED ZONEMAP Command 21
15.3.3.2 Refreshing Zone Maps Using the DBMS_MVIEW Package 22
15.4 Performing Pruning Using Zone Maps 22
15.4.1 How Oracle AI Database Performs Pruning Using Zone Maps 22


Copyright © 2001, 2025, Oracle and/or its affiliates.

15.4.1.1 Pruning Tables Using Zone Maps 22
15.4.1.2 Pruning Partitioned Tables Using Zone Maps and Attribute Clustering 23
15.4.2 Examples: Performing Pruning with Zone Maps and Attribute Clustering 24
15.4.2.1 Example: Partitions and Table Scan Pruning 25
15.4.2.2 Example: Zone Map Join Pruning 26
15.5 Viewing Zone Map Information 27
15.5.1 Viewing Details of Zone Maps in the Database 27
15.5.2 Viewing the Measures of a Zone Map 27

Part III Data Movement/ETL

16 Data Movement/ETL Overview

16.1 Overview of ETL in Data Warehouses 1
16.1.1 ETL Basics in Data Warehousing 1
16.1.1.1 Extraction of Data in Data Warehouses 1
16.1.1.2 Transportation of Data in Data Warehouses 2
16.2 ETL Tools for Data Warehouses 2
16.2.1 Daily Operations in Data Warehouses 2
16.2.2 Evolution of the Data Warehouse 2

17 Extraction in Data Warehouses

17.1 Overview of Extraction in Data Warehouses 1
17.2 Introduction to Extraction Methods in Data Warehouses 1
17.2.1 Logical Extraction Methods 2
17.2.2 Physical Extraction Methods 2
17.2.3 Change Tracking Methods 3
17.3 Data Warehousing Extraction Examples 4
17.3.1 Extraction Using Data Files 5
17.3.1.1 Extracting into Flat Files Using SQL*Plus 5
17.3.1.2 Extracting into Flat Files Using OCI or Pro*C Programs 6
17.3.1.3 Exporting into Export Files Using the Export Utility 6
17.3.1.4 Extracting into Export Files Using External Tables 7
17.3.2 Extraction Through Distributed Operations 7

18 Transportation in Data Warehouses

18.1 Overview of Transportation in Data Warehouses 1
18.2 Introduction to Transportation Mechanisms in Data Warehouses 1
18.2.1 Transportation Using Flat Files 1
18.2.2 Transportation Through Distributed Operations 2


Copyright © 2001, 2025, Oracle and/or its affiliates.

18.2.3 Transportation Using Transportable Tablespaces 2
18.2.3.1 Using Transportable Tablespaces to Transport Data into Data Warehouses: Example 2
18.2.3.2 Other Uses of Transportable Tablespaces 5

## 19 Loading and Transformation in Data Warehouses

19.1 Overview of Loading and Transformation in Data Warehouses 1
19.1.1 Data Warehouses: Transformation Flow 1
19.1.1.1 Multistage Data Transformation in Data Warehouses 1
19.1.1.2 Pipelined Data Transformation in Data Warehouses 2
19.1.1.3 Staging Area in Data Warehouses 3
19.1.2 About Batch Updates and Online Table Redefinition 3
19.1.3 Overview of Monitoring ETL Operations 3
19.2 Loading Mechanisms for Data Warehouses 4
19.2.1 Loading a Data Warehouse with SQL*Loader 4
19.2.1.1 Using SQL*Loader to Load From an Object Store 5
19.2.2 Loading a Data Warehouse with External Tables 6
19.2.2.1 Using DBMS_CLOUD to Create External Tables for Object Store Data 8
19.2.3 Loading a Data Warehouse with OCI and Direct-Path APIs 8
19.2.4 Loading a Data Warehouse with Export/Import 8
19.3 Transformation Mechanisms in Data Warehouses 8
19.3.1 Transforming Data Using SQL 9
19.3.1.1 CREATE TABLE ... AS SELECT And INSERT /*+APPEND*/ AS SELECT 9
19.3.1.2 Transforming Data Using UPDATE 10
19.3.1.3 Transforming Data Using MERGE 10
19.3.1.4 Transforming Data Using Multitable INSERT 10
19.3.2 Transforming Data Using PL/SQL 12
19.3.3 Transforming Data Using Table Functions 12
19.3.3.1 What is a Table Function? 12
19.4 Error Logging and Handling Mechanisms 19
19.4.1 Business Rule Violations 19
19.4.2 Data Rule Violations (Data Errors) 19
19.4.2.1 Handling Data Errors with SQL 20
19.4.2.2 Handling Data Errors in PL/SQL 21
19.4.2.3 Handling Data Errors with an Error Logging Table 21
19.5 Loading and Transformation Scenarios 23
19.5.1 Key Lookup Scenario 23
19.5.2 Business Rule Violation Scenario 23
19.5.3 Data Error Scenarios 24
19.5.4 Pivoting Scenarios 26


Copyright © 2001, 2025, Oracle and/or its affiliates.

# Part IV Relational Analytics

## 20 SQL for Analysis and Reporting


20.1 Overview of SQL for Analysis and Reporting 1
20.2 Ranking, Windowing, and Reporting Functions 3
20.2.1 Ranking Functions 4
20.2.1.1 RANK and DENSE_RANK Functions 4
20.2.1.2 APPROX_RANK Function 9
20.2.1.3 Bottom N Ranking Functions 11
20.2.1.4 CUME_DIST Function 11
20.2.1.5 PERCENT_RANK Function 12
20.2.1.6 NTILE Function 12
20.2.1.7 ROW_NUMBER Function 13
20.2.2 Windowing Functions 13
20.2.2.1 Examples of Window Clauses 15
20.2.2.2 Examples of Windowing Clause Extensions 16
20.2.2.3 About Treatment of NULLs as Input to Window Functions 17
20.2.2.4 Windowing Functions with Logical Offset 17
20.2.2.5 Centered Aggregate Function 18
20.2.2.6 Windowing Aggregate Functions in the Presence of Duplicates 19
20.2.2.7 Varying Window Size for Each Row 20
20.2.2.8 Windowing Aggregate Functions with Physical Offsets 20
20.2.2.9 Parallel Partition-Wise Operations with Windowing Functions 21
20.2.2.10 QUALIFY Clause 21
20.2.3 Reporting Functions 22
20.2.3.1 RATIO_TO_REPORT Function 24
20.2.4 LAG/LEAD Functions 24
20.2.4.1 LAG/LEAD Syntax 25
20.2.5 FIRST_VALUE, LAST_VALUE, and NTH_VALUE Functions 26
20.2.5.1 FIRST_VALUE and LAST_VALUE Functions 26
20.2.5.2 NTH_VALUE Function 27
20.3 Advanced Aggregates for Analysis 28
20.3.1 About Approximate Aggregates 28
20.3.2 LISTAGG Function 30
20.3.2.1 LISTAGG as Aggregate 31
20.3.2.2 LISTAGG as Reporting Aggregate 34
20.3.3 FIRST/LAST Functions 34
20.3.3.1 FIRST/LAST As Regular Aggregates 35
20.3.3.2 FIRST/LAST As Reporting Aggregates 35
20.3.4 Inverse Percentile Functions 36




Copyright © 2001, 2025, Oracle and/or its affiliates.

20.3.4.1 Normal Aggregate Syntax 36
20.3.4.2 Inverse Percentile Example Basis 37
20.3.4.3 As Reporting Aggregates 38
20.3.4.4 Restrictions on Inverse Percentile Functions 39
20.3.4.5 Using Percentile Functions that Return Approximate Results 39
20.3.5 Hypothetical Rank Functions 41
20.3.6 Linear Regression Functions 42
20.3.6.1 REGR_COUNT Function 42
20.3.6.2 REGR_AVGY and REGR_AVGX Functions 42
20.3.6.3 REGR_SLOPE and REGR_INTERCEPT Functions 43
20.3.6.4 REGR_R2 Function 43
20.3.6.5 REGR_SXX, REGR_SYY, and REGR_SXY Functions 43
20.3.6.6 Linear Regression Statistics Examples 43
20.3.6.7 Sample Linear Regression Calculation 43
20.3.7 About Statistical Aggregates 44
20.3.7.1 Descriptive Statistics 44
20.3.7.2 Hypothesis Testing - Parametric Tests 45
20.3.7.3 Crosstab Statistics 45
20.3.7.4 Hypothesis Testing - Non-Parametric Tests 45
20.3.7.5 Non-Parametric Correlation 46
20.3.8 About User-Defined Aggregates 46
20.4 Pivoting Operations 47
20.4.1 Creating the View Used for Pivoting Examples 47
20.4.2 Pivoting Example 48
20.4.3 Pivoting on Multiple Columns 48
20.4.4 Pivoting: Multiple Aggregates 49
20.4.5 Distinguishing PIVOT-Generated Nulls from Nulls in Source Data 50
20.4.6 Wildcard and Subquery Pivoting with XML Operations 51
20.5 Unpivoting Operations 52
20.6 Data Densification for Reporting 53
20.6.1 About Partition Join Syntax 54
20.6.2 Sample of Sparse Data 54
20.6.3 Filling Gaps in Data 55
20.6.4 Filling Gaps in Two Dimensions 56
20.6.5 Filling Gaps in an Inventory Table 57
20.6.6 Computing Data Values to Fill Gaps 58
20.7 Time Series Calculations on Densified Data 59
20.7.1 Period-to-Period Comparison for One Time Level: Example 60
20.7.2 Period-to-Period Comparison for Multiple Time Levels: Example 61
20.7.3 Creating a Custom Member in a Dimension: Example 65
20.8 Miscellaneous Analysis and Reporting Capabilities 66
20.8.1 TIME_BUCKET Function 67


Copyright © 2001, 2025, Oracle and/or its affiliates.

20.8.1.1 TIME_BUCKET Syntax 67
20.8.1.2 TIME_BUCKET Examples 68
20.8.2 WIDTH_BUCKET Function 71
20.8.2.1 WIDTH_BUCKET Syntax 71
20.8.3 Linear Algebra 73
20.8.4 CASE Expressions 75
20.8.4.1 Creating Histograms Using CASE Statement 76
20.8.5 Frequent Itemsets in SQL Analytics 77
20.9 Limiting SQL Rows 77
20.9.1 SQL Row Limiting Restrictions and Considerations 80

## 21 SQL for Aggregation in Data Warehouses

21.1 Overview of SQL for Aggregation in Data Warehouses 1
21.1.1 About Analyzing Across Multiple Dimensions 2
21.1.2 About Optimized Aggregation Performance 3
21.1.3 Data Warehousing: An Aggregate Scenario 3
21.2 ROLLUP Extension to GROUP BY 4
21.2.1 When to Use ROLLUP 4
21.2.2 ROLLUP Syntax 5
21.2.3 Partial Rollup 6
21.3 CUBE Extension to GROUP BY 7
21.3.1 When to Use CUBE 7
21.3.2 CUBE Syntax 7
21.3.3 Partial CUBE 8
21.3.4 Calculating Subtotals Without CUBE 9
21.4 GROUPING Functions 9
21.4.1 GROUPING Function 10
21.4.2 When to Use GROUPING 12
21.4.3 GROUPING_ID Function 12
21.4.4 GROUP_ID Function 13
21.5 GROUPING SETS Expression 14
21.5.1 GROUPING SETS Syntax 15
21.6 About Composite Columns and Grouping 15
21.7 Concatenated Groupings and Data Aggregation 17
21.7.1 Concatenated Groupings and Hierarchical Data Cubes 19
21.8 Considerations when Using Aggregation in Data Warehouses 20
21.8.1 Hierarchy Handling in ROLLUP and CUBE 20
21.8.2 Column Capacity in ROLLUP and CUBE 21
21.8.3 HAVING Clause Used with GROUP BY Extensions 21
21.8.4 ORDER BY Clause Used with GROUP BY Extensions 21
21.8.5 Using Other Aggregate Functions with ROLLUP and CUBE 22


Copyright © 2001, 2025, Oracle and/or its affiliates.

21.8.6 Using In-Memory Aggregation 22
21.9 Computation Using the WITH Clause 23
21.10 Working with Hierarchical Cubes in SQL 24
21.10.1 Specifying Hierarchical Cubes in SQL 24
21.10.2 Querying Hierarchical Cubes in SQL 24
21.10.2.1 SQL for Creating Materialized Views to Store Hierarchical Cubes 26
21.10.2.2 Examples of Hierarchical Cube Materialized Views 26

## 22 SQL for Pattern Matching

22.1 Overview of Pattern Matching in Data Warehouses 1
22.1.1 Why Use Pattern Matching? 2
22.1.2 How Data is Processed in Pattern Matching 5
22.1.3 About Pattern Matching Special Capabilities 6
22.2 Basic Topics in Pattern Matching 6
22.2.1 Basic Examples of Pattern Matching 6
22.2.2 Tasks and Keywords in Pattern Matching 10
22.2.3 Pattern Matching Syntax 11
22.3 Pattern Matching Details 13
22.3.1 PARTITION BY: Logically Dividing the Rows into Groups 14
22.3.2 ORDER BY: Logically Ordering the Rows in a Partition 14
22.3.3 [ONE ROW | ALL ROWS] PER MATCH: Choosing Summaries or Details for Each Match 14
22.3.4 MEASURES: Defining Calculations for Use in the Query 15
22.3.5 PATTERN: Defining the Row Pattern to Be Matched 15
22.3.5.1 Reluctant Versus Greedy Quantifier 16
22.3.5.2 Operator Precedence 17
22.3.6 SUBSET: Defining Union Row Pattern Variables 17
22.3.7 DEFINE: Defining Primary Pattern Variables 18
22.3.8 AFTER MATCH SKIP: Defining Where to Restart the Matching Process After a Match Is Found 20
22.3.9 Expressions in MEASURES and DEFINE 21
22.3.9.1 MATCH_NUMBER: Finding Which Rows Are in Which Match 21
22.3.9.2 CLASSIFIER: Finding Which Pattern Variable Applies to Which Rows 22
22.3.9.3 Row Pattern Column References 22
22.3.9.4 Aggregates 23
22.3.9.5 Row Pattern Navigation Operations 24
22.3.9.6 Running Versus Final Semantics and Keywords 26
22.3.10 Row Pattern Output 31
22.3.10.1 Correlation Name and Row Pattern Output 31
22.4 Advanced Topics in Pattern Matching 31
22.4.1 Nesting FIRST and LAST Within PREV and NEXT in Pattern Matching 32
22.4.2 Handling Empty Matches or Unmatched Rows in Pattern Matching 33


Copyright © 2001, 2025, Oracle and/or its affiliates.

22.4.2.1 Handling Empty Matches in Pattern Matching 33
22.4.2.2 Handling Unmatched Rows in Pattern Matching 34
22.4.3 How to Exclude Portions of the Pattern from the Output 34
22.4.4 How to Express All Permutations 35
22.5 Rules and Restrictions in Pattern Matching 36
22.5.1 Input Table Requirements in Pattern Matching 36
22.5.2 Prohibited Nesting in the MATCH_RECOGNIZE Clause 37
22.5.3 Concatenated MATCH_RECOGNIZE Clause 37
22.5.4 Aggregate Restrictions 38
22.6 Examples of Pattern Matching 38
22.6.1 Pattern Matching Examples: Stock Market 38
22.6.2 Pattern Matching Examples: Security Log Analysis 46
22.6.3 Pattern Matching Examples: Sessionization 50
22.6.4 Pattern Matching Example: Financial Tracking 55
22.7 Fuzzy String Matching 56

## 23 SQL for Modeling

23.1 Overview of SQL Modeling in Data Warehouses 1
23.1.1 How Data is Processed in a SQL Model 3
23.1.2 Why Use SQL Modeling in Data Warehouses? 3
23.1.3 About SQL Modeling Capabilities 5
23.2 Basic Topics in SQL Modeling 7
23.2.1 Base Schema for SQL Modeling Examples 8
23.2.2 MODEL Clause Syntax 8
23.2.3 Keywords in SQL Modeling 10
23.2.3.1 Assigning Values and Null Handling 10
23.2.3.2 Calculation Definition 11
23.2.4 About Cell Referencing in SQL Modeling 11
23.2.4.1 Symbolic Dimension References 12
23.2.4.2 Positional Dimension References 12
23.2.5 About Rules for SQL Modeling 12
23.2.6 Order of Evaluation of SQL Modeling Rules 15
23.2.7 Global and Local Keywords for SQL Modeling Rules 16
23.2.8 UPDATE, UPSERT, and UPSERT ALL Behavior 16
23.2.8.1 UPDATE Behavior 16
23.2.8.2 UPSERT Behavior 17
23.2.8.3 UPSERT ALL Behavior 17
23.2.9 Treatment of NULLs and Missing Cells in SQL Modeling 18
23.2.9.1 Distinguishing Missing Cells from NULLs 19
23.2.9.2 Use Defaults for Missing Cells and NULLs 20
23.2.9.3 Using NULLs in a Cell Reference 20


Copyright © 2001, 2025, Oracle and/or its affiliates.

23.2.10 About Reference Models in SQL Modeling 21
23.3 Advanced Topics in SQL Modeling 23
23.3.1 FOR Loops in SQL Modeling 23
23.3.1.1 Evaluation of Formulas with FOR Loops 26
23.3.2 Iterative Models in SQL Modeling 28
23.3.3 Rule Dependency in AUTOMATIC ORDER Models 29
23.3.4 Ordered Rules in SQL Modeling 30
23.3.5 Analytic Functions in SQL Modeling 31
23.3.6 Unique Dimensions Versus Unique Single References in SQL Modeling 32
23.3.7 Rules and Restrictions when Using SQL for Modeling 34
23.4 Performance Considerations with SQL Modeling 36
23.4.1 Parallel Execution and SQL Modeling 36
23.4.2 Aggregate Computation and SQL Modeling 36
23.4.3 Using EXPLAIN PLAN to Understand Model Queries 37
23.5 Examples of SQL Modeling 39
23.5.1 SQL Modeling Example 1: Calculating Sales Differences 40
23.5.2 SQL Modeling Example 2: Calculating Percentage Change 40
23.5.3 SQL Modeling Example 3: Calculating Net Present Value 40
23.5.4 SQL Modeling Example 4: Calculating Using Simultaneous Equations 41
23.5.5 SQL Modeling Example 5: Calculating Using Regression 42
23.5.6 SQL Modeling Example 6: Calculating Mortgage Amortization 42

24 Advanced Analytical SQL

24.1 Examples of Business Intelligence Queries 1
24.1.1 Business Intelligence Query Example 1: Percent Change in Market Share of Products in a Calculated Set 2
24.1.2 Business Intelligence Query Example 2: Sales Projection that Fills in Missing Data 4
24.1.3 Business Intelligence Query Example 3: Customer Analysis by Grouping Customers into Buckets 6
24.1.4 Business Intelligence Query Example 4: Frequent Itemsets 8

Part V Analytic Views

25 Overview of Analytic Views

25.1 What Are Analytic Views? 1
25.2 New Features for Analytic Views 3
25.3 Privileges for Analytic Views 4
25.4 Application Programming Interfaces for Analytic Views 5
25.5 Compilation States of Analytic Views 8


Copyright © 2001, 2025, Oracle and/or its affiliates.

25.6 Validation of Data 8
25.7 Classifications for Analytic Views 9
25.8 Share Analytic Views with Application Containers 9
25.9 Alter or Drop an Analytic View Object 10
25.10 Data and Scripts for Examples 10
25.10.1 About the Data and Scripts for Examples 10
25.10.2 Create Attribute Dimension Statements 12
25.10.3 Create Hierarchy Statements 15
25.10.4 Create Analytic View Statements 16

26 Attribute Dimension and Hierarchy Objects

26.1 About Attribute Dimensions and Hierarchies 1
26.2 Attributes and Hierarchical Attributes 4
26.3 Order Levels 7
26.4 Level Keys 8
26.5 Determine Attribute Relationships 9

27 Analytic View Objects

27.1 About Analytic Views 1
27.2 Measures of Analytic Views 2
27.3 Create Analytic Views 3
27.4 Examples of Calculated Measures 6
27.5 Attribute Reporting 11
27.6 Analytic View Queries with Filtered Facts and Added Measures 15
27.6.1 Analytic View Query with Filtered Facts 15
27.6.2 Analytic View Query with Added Measures 16
27.6.3 Analytic View Query with Filtered Facts and Multiple Added Measures 17

Glossary

Index


Copyright © 2001, 2025, Oracle and/or its affiliates.

# Preface

This preface contains these topics:


- Audience
- Related Documents
- Documentation Accessibility
- Conventions


## Audience

This guide is intended for database administrators, system administrators, and database application developers who design, maintain, and use data warehouses.

To use this document, you need to be familiar with relational database concepts, basic Oracle server concepts, and the operating system environment under which you are running Oracle.

## Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

**Access to Oracle Support**

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

## Related Documents

Many of the examples in this book use the sample schemas of the seed database, which is installed by default when you install Oracle. Refer to Oracle AI Database Sample Schemas for information on how these schemas were created and how you can use them yourself.

Note that this book is meant as a supplement to standard texts about data warehousing. This book focuses on Oracle-specific material and does not reproduce in detail material of a general nature. For additional information, see:

- *The Data Warehouse Toolkit* by Ralph Kimball (John Wiley and Sons, 1996)
- *Building the Data Warehouse* by William Inmon (John Wiley and Sons, 1996)

## Conventions

The following text conventions are used in this document:



Copyright © 2001, 2025, Oracle and/or its affiliates.

| Convention | Meaning |
| --- | --- |
| **boldface** | Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary. |
| *italic* | Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values. |
| `monospace` | Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter. |


Copyright © 2001, 2025, Oracle and/or its affiliates.

# Part I
# Data Warehouse - Fundamentals

This section introduces basic data warehousing concepts.

It contains the following chapters:


- Introduction to Data Warehousing Concepts
- Data Warehousing Logical Design
- Data Warehousing Physical Design
- Data Warehousing Optimizations and Techniques




Copyright © 2001, 2025, Oracle and/or its affiliates.

1

# Introduction to Data Warehousing Concepts

This chapter provides an overview of the Oracle data warehousing implementation. It contains:


- What Is a Data Warehouse?
- Contrasting OLTP and Data Warehousing Environments
- Common Data Warehouse Tasks
- Data Warehouse Architectures


## 1.1 What Is a Data Warehouse?

A **data warehouse** is a database designed to enable business intelligence activities: it exists to help users understand and enhance their organization's performance. It is designed for query and analysis rather than for transaction processing, and usually contains historical data derived from transaction data, but can include data from other sources. Data warehouses separate analysis workload from transaction workload and enable an organization to consolidate data from several sources. This helps in:

- Maintaining historical records
- Analyzing the data to gain a better understanding of the business and to improve the business

In addition to a relational database, a data warehouse environment can include an extraction, transportation, transformation, and loading (ETL) solution, statistical analysis, reporting, data mining capabilities, client analysis tools, and other applications that manage the process of gathering data, transforming it into useful, actionable information, and delivering it to business users.

To achieve the goal of enhanced business intelligence, the data warehouse works with data collected from multiple sources. The source data may come from internally developed systems, purchased applications, third-party data syndicators and other sources. It may involve transactions, production, marketing, human resources and more. In today's world of big data, the data may be many billions of individual clicks on web sites or the massive data streams from sensors built into complex machinery.

Data warehouses are distinct from online transaction processing (OLTP) systems. With a data warehouse you separate analysis workload from transaction workload. Thus data warehouses are very much read-oriented systems. They have a far higher amount of data reading versus writing and updating. This enables far better analytical performance and avoids impacting your transaction systems. A data warehouse system can be optimized to consolidate data from many sources to achieve a key goal: it becomes your organization's "single source of truth". There is great value in having a consistent source of data that all users can look to; it prevents many disputes and enhances decision-making efficiency.

A data warehouse usually stores many months or years of data to support historical analysis. The data in a data warehouse is typically loaded through an extraction, transformation, and loading (ETL) process from multiple data sources. Modern data warehouses are moving toward an extract, load, transformation (ELT) architecture in which all or most data transformation is performed on the database that hosts the data warehouse. It is important to note that defining the ETL process is a very large part of the design effort of a data warehouse.



Copyright © 2001, 2025, Oracle and/or its affiliates.

Similarly, the speed and reliability of ETL operations are the foundation of the data warehouse once it is up and running.

Users of the data warehouse perform data analyses that are often time-related. Examples include consolidation of last year's sales figures, inventory analysis, and profit by product and by customer. But time-focused or not, users want to "slice and dice" their data however they see fit and a well-designed data warehouse will be flexible enough to meet those demands. Users will sometimes need highly aggregated data, and other times they will need to drill down to details. More sophisticated analyses include trend analyses and data mining, which use existing data to forecast trends or predict futures. The data warehouse acts as the underlying engine used by middleware business intelligence environments that serve reports, dashboards and other interfaces to end users.

Although the discussion above has focused on the term "data warehouse", there are two other important terms that need to be mentioned. These are the data mart and the operation data store (ODS).

A data mart serves the same role as a data warehouse, but it is intentionally limited in scope. It may serve one particular department or line of business. The advantage of a data mart versus a data warehouse is that it can be created much faster due to its limited coverage. However, data marts also create problems with inconsistency. It takes tight discipline to keep data and calculation definitions consistent across data marts. This problem has been widely recognized, so data marts exist in two styles. Independent data marts are those which are fed directly from source data. They can turn into islands of inconsistent information. Dependent data marts are fed from an existing data warehouse. Dependent data marts can avoid the problems of inconsistency, but they require that an enterprise-level data warehouse already exist.

Operational data stores exist to support daily operations. The ODS data is cleaned and validated, but it is not historically deep: it may be just the data for the current day. Rather than support the historically rich queries that a data warehouse can handle, the ODS gives data warehouses a place to get access to the most current data, which has not yet been loaded into the data warehouse. The ODS may also be used as a source to load the data warehouse. As data warehousing loading techniques have become more advanced, data warehouses may have less need for ODS as a source for loading data. Instead, constant trickle-feed systems can load the data warehouse in near real time.

A common way of introducing data warehousing is to refer to the characteristics of a data warehouse as set forth by William Inmon:

- Subject Oriented
- Integrated
- Nonvolatile
- Time Varient

**Subject Oriented**

Data warehouses are designed to help you analyze data. For example, to learn more about your company's sales data, you can build a data warehouse that concentrates on sales. Using this data warehouse, you can answer questions such as "Who was our best customer for this item last year?" or "Who is likely to be our best customer next year?" This ability to define a data warehouse by subject matter, sales in this case, makes the data warehouse subject oriented.

**Integrated**

Integration is closely related to subject orientation. Data warehouses must put data from disparate sources into a consistent format. They must resolve such problems as naming


Copyright © 2001, 2025, Oracle and/or its affiliates.

conflicts and inconsistencies among units of measure. When they achieve this, they are said to be integrated.

**Nonvolatile**

Nonvolatile means that, once entered into the data warehouse, data should not change. This is logical because the purpose of a data warehouse is to enable you to analyze what has occurred.

**Time Varient**

A data warehouse's focus on change over time is what is meant by the term time variant. In order to discover trends and identify hidden patterns and relationships in business, analysts need large amounts of data. This is very much in contrast to online transaction processing (OLTP) systems, where performance requirements demand that historical data be moved to an archive.

### 1.1.1 Key Characteristics of a Data Warehouse

The key characteristics of a data warehouse are as follows:

- Data is structured for simplicity of access and high-speed query performance.
- End users are time-sensitive and desire speed-of-thought response times.
- Large amounts of historical data are used.
- Queries often retrieve large amounts of data, perhaps many thousands of rows.
- Both predefined and ad hoc queries are common.
- The data load involves multiple sources and transformations.

In general, fast query performance with high data throughput is the key to a successful data warehouse.

## 1.2 Contrasting OLTP and Data Warehousing Environments

There are important differences between an OLTP system and a data warehouse. One major difference between the types of system is that data warehouses are not exclusively in third normal form (3NF), a type of data normalization common in OLTP environments.

Data warehouses and OLTP systems have very different requirements. Here are some examples of differences between typical data warehouses and OLTP systems:

- Workload

  Data warehouses are designed to accommodate *ad hoc* queries and data analysis. You might not know the workload of your data warehouse in advance, so a data warehouse should be optimized to perform well for a wide variety of possible query and analytical operations.

  OLTP systems support only predefined operations. Your applications might be specifically tuned or designed to support only these operations.

- Data modifications

  A data warehouse is updated on a regular basis by the ETL process (run nightly or weekly) using bulk data modification techniques. The end users of a data warehouse do not directly update the data warehouse except when using analytical tools, such as data mining, to make predictions with associated probabilities, assign customers to market segments, and develop customer profiles.


Copyright © 2001, 2025, Oracle and/or its affiliates.

In OLTP systems, end users routinely issue individual data modification statements to the database. The OLTP database is always up to date, and reflects the current state of each business transaction.

- Schema design

  Data warehouses often use partially denormalized schemas to optimize query and analytical performance.

  OLTP systems often use fully normalized schemas to optimize update/insert/delete performance, and to guarantee data consistency.

- Typical operations

  A typical data warehouse query scans thousands or millions of rows. For example, "Find the total sales for all customers last month."

  A typical OLTP operation accesses only a handful of records. For example, "Retrieve the current order for this customer."

- Historical data

  Data warehouses usually store many months or years of data. This is to support historical analysis and reporting.

  OLTP systems usually store data from only a few weeks or months. The OLTP system stores only historical data as needed to successfully meet the requirements of the current transaction.

## 1.3 Common Data Warehouse Tasks

As an Oracle data warehousing administrator or designer, you can expect to be involved in the following tasks:

- Configuring an Oracle database for use as a data warehouse
- Designing data warehouses
- Performing upgrades of the database and data warehousing software to new releases
- Managing schema objects, such as tables, indexes, and materialized views
- Managing users and security
- Developing routines used for the extraction, transformation, and loading (ETL) processes
- Creating reports based on the data in the data warehouse
- Backing up the data warehouse and performing recovery when necessary
- Monitoring the data warehouse's performance and taking preventive or corrective action as required

In a small-to-midsize data warehouse environment, you might be the sole person performing these tasks. In large, enterprise environments, the job is often divided among several DBAs and designers, each with their own specialty, such as database security or database tuning.

These tasks are illustrated in the following:

- For more information regarding partitioning, see *Oracle AI Database VLDB and Partitioning Guide*.
- For more information regarding database security, see *Oracle AI Database Security Guide*.
- For more information regarding database performance, see *Oracle AI Database Performance Tuning Guide* and *Oracle AI Database SQL Tuning Guide*.


Copyright © 2001, 2025, Oracle and/or its affiliates.

- For more information regarding backup and recovery, see *Oracle AI Database Backup and Recovery User's Guide*.
- For more information regarding ODI, see *Oracle Fusion Middleware Developer's Guide for Oracle Data Integrator*.

# 1.4 Data Warehouse Architectures

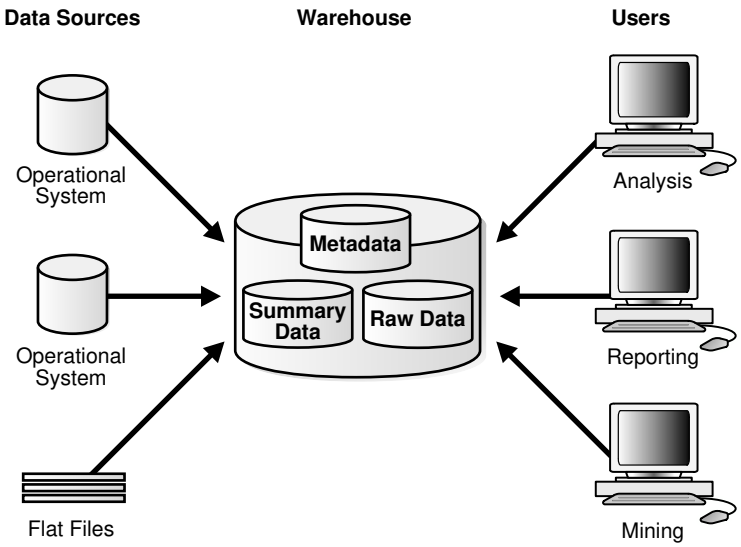
Data warehouses and their architectures vary depending upon the specifics of an organization's situation. Three common architectures are:

- Data Warehouse Architecture: Basic
- Data Warehouse Architecture: with a Staging Area
- Data Warehouse Architecture: with a Staging Area and Data Marts

## 1.4.1 Data Warehouse Architecture: Basic

Figure 1-1 shows a simple architecture for a data warehouse. End users directly access data derived from several source systems through the data warehouse.

**Figure 1-1 Architecture of a Data Warehouse**




In Figure 1-1, the metadata and raw data of a traditional OLTP system is present, as is an additional type of data, summary data. Summaries are a mechanism to pre-compute common expensive, long-running operations for sub-second data retrieval. For example, a typical data warehouse query is to retrieve something such as August sales. A summary in an Oracle database is called a materialized view.

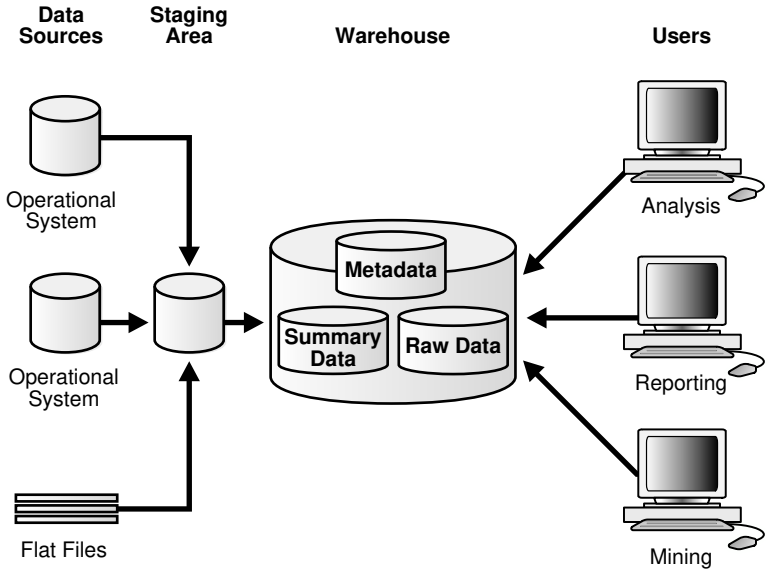
The consolidated storage of the raw data as the center of your data warehousing architecture is often referred to as an Enterprise Data Warehouse (EDW). An EDW provides a 360-degree view into the business of an organization by holding all relevant business information in the most detailed format.


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 1.4.2 Data Warehouse Architecture: with a Staging Area

You must clean and process your operational data before putting it into the warehouse, as shown in Figure 1-2. You can do this programmatically, although most data warehouses use a staging area instead. A staging area simplifies data cleansing and consolidation for operational data coming from multiple source systems, especially for enterprise data warehouses where all relevant information of an enterprise is consolidated. Figure 1-2 illustrates this typical architecture.

**Figure 1-2 Architecture of a Data Warehouse with a Staging Area**

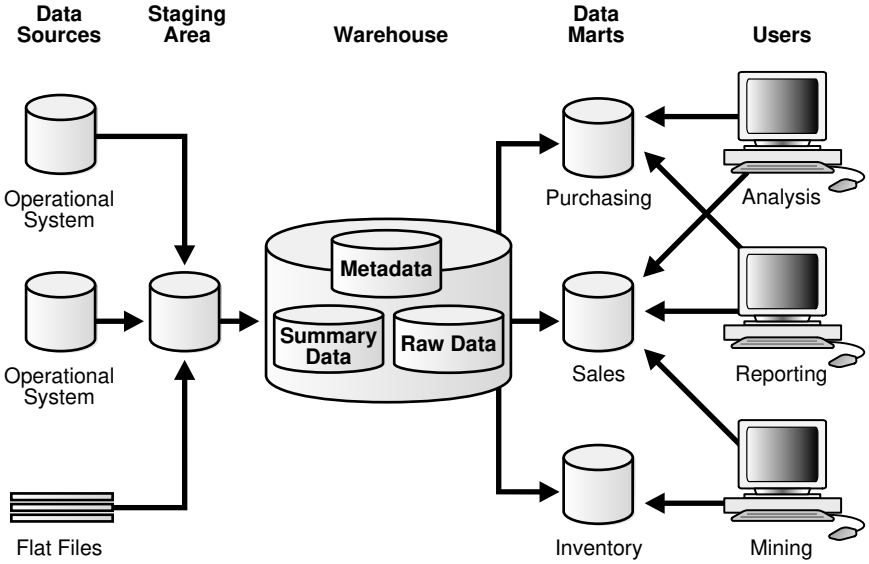



## 1.4.3 Data Warehouse Architecture: with a Staging Area and Data Marts

Although the architecture in Figure 1-2 is quite common, you may want to customize your warehouse's architecture for different groups within your organization. You can do this by adding **data marts**, which are systems designed for a particular line of business. Figure 1-3 illustrates an example where purchasing, sales, and inventories are separated. In this example, a financial analyst might want to analyze historical data for purchases and sales or mine historical data to make predictions about customer behavior.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Figure 1-3 Architecture of a Data Warehouse with a Staging Area and Data Marts**




> **Note**
>
> Data marts can be physically instantiated or implemented purely logically though views. Furthermore, data marts can be co-located with the enterprise data warehouse or built as separate systems. Building an end-to-end data warehousing architecture with an enterprise data warehouse and surrounding data marts is not the focus of this book.


Copyright © 2001, 2025, Oracle and/or its affiliates.

2

# Data Warehousing Logical Design

This chapter explains how to create a logical design for a data warehousing environment and includes the following topics:


- Logical Versus Physical Design in Data Warehouses
- Creating a Logical Design
- About Third Normal Form Schemas
- About Star Schemas
- Improved Analytics Using the In-Memory Column Store
- Automatic Big Table Caching to Improve the Performance of In-Memory Parallel Queries


## 2.1 Logical Versus Physical Design in Data Warehouses

Your organization has decided to build an enterprise data warehouse. You have defined the business requirements and agreed upon the scope of your business goals, and created a conceptual design. Now you need to translate your requirements into a system deliverable. To do so, you create the logical and physical design for the data warehouse. You then define:

- The specific data content
- Relationships within and between groups of data
- The system environment supporting your data warehouse
- The data transformations required
- The frequency with which data is refreshed

The logical design is more conceptual and abstract than the physical design. In the logical design, you look at the logical relationships among the objects. In the physical design, you look at the most effective way of storing and retrieving the objects as well as handling them from a transportation and backup/recovery perspective.

Orient your design toward the needs of the end users. End users typically want to perform analysis and look at aggregated data, rather than at individual transactions. However, end users might not know what they need until they see it. In addition, a well-planned design allows for growth and changes as the needs of users change and evolve.

By beginning with the logical design, you focus on the information requirements and save the implementation details for later.

## 2.2 Creating a Logical Design

A logical design is conceptual and abstract. You do not deal with the physical implementation details yet. You deal only with defining the types of information that you need.

One technique you can use to model your organization's logical information requirements is entity-relationship modeling. Entity-relationship modeling involves identifying the things of importance (entities), the properties of these things (attributes), and how they are related to one another (relationships).



Copyright © 2001, 2025, Oracle and/or its affiliates.

The process of logical design involves arranging data into a series of logical relationships called entities and attributes. An entity represents a chunk of information. In relational databases, an entity often maps to a table. An attribute is a component of an entity that helps define the uniqueness of the entity. In relational databases, an attribute maps to a column.

To ensure that your data is consistent, you must use unique identifiers. A unique identifier is something you add to tables so that you can differentiate between the same item when it appears in different places. In a physical design, this is usually a primary key.

Entity-relationship modeling is purely logical and applies to both OLTP and data warehousing systems. It is also applicable to the various common physical schema modeling techniques found in data warehousing environments, namely normalized (3NF) schemas in Enterprise Data Warehousing environments, star or snowflake schemas in data marts, or hybrid schemas with components of both of these classical modeling techniques.

**See Also**

- *Oracle Fusion Middleware Developing Integration Projects with Oracle Data Integrator* for more details regarding ODI

### 2.2.1 What is a Schema?

A schema is a collection of database objects, including tables, views, indexes, and synonyms. You can arrange schema objects in the schema models designed for data warehousing in a variety of ways. Most data warehouses use a dimensional model.

The model of your source data and the requirements of your users help you design the data warehouse schema. You can sometimes get the source model from your company's enterprise data model and reverse-engineer the logical data model for the data warehouse from this. The physical implementation of the logical data warehouse model may require some changes to adapt it to your system parameters—size of computer, number of users, storage capacity, type of network, and software. A key part of designing the schema is whether to use a third normal form, star, or snowflake schema, and these are discussed later.

## 2.3 About Third Normal Form Schemas

Third Normal Form design seeks to minimize data redundancy and avoid anomalies in data insertion, updates and deletion. 3NF design has a long heritage in online transaction processing (OLTP) systems. OLTP systems must maximize performance and accuracy when inserting, updating and deleting data. Transactions must be handled as quickly as possible or the business may be unable to handle the flow of events, perhaps losing sales or incurring other costs. Therefore, 3NF designs avoid redundant data manipulation and minimize table locks, both of which can slow inserts, updates and deletes. 3NF designs also works well to abstract the data from specific application needs. If new types of data are added to the environment, you can extend the data model with relative ease and minimal impact to existing applications. Likewise, if you have completely new types of analyses to perform in your data warehouse, a well-designed 3NF schema will be able to handle them without requiring redesigned data structures.3NF designs have great flexibility, but it comes at a cost. 3NF databases use very many tables and this requires complex queries with many joins. For full scale enterprise models built in 3NF form, over one thousand tables are commonly encountered in the schema. With the kinds of queries involved in data warehousing, which will often need access to many rows from many tables, this design imposes understanding and performance penalties. It can be complex for query builders, whether they are humans or business intelligence tools and applications, to choose and join the tables needed for a given

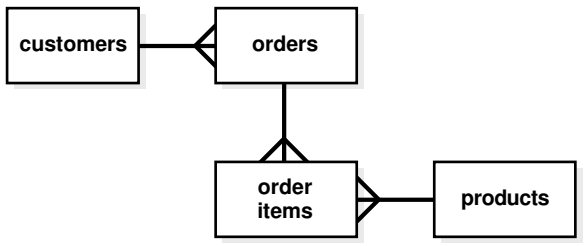

Copyright © 2001, 2025, Oracle and/or its affiliates.

piece of data when there are very large numbers of tables available. Even when the tables are readily chosen by the query generator, the 3NF schema often requires that a large number of tables be used in a single query. More tables in a query mean more potential data access paths, which makes the database query optimizer's job harder. The end result can be slow query performance.

The issue of slow query performance in a 3NF system is not necessarily limited to the core queries used to create reports and analyses. It can also show up in the simpler task of users browsing subsets of data to understand the contents. Similarly, the complexity of a 3NF schema may impact generating the pick-lists of data used to constrain queries and reports. Although these may seem relatively minor issues, speedy response time for such processes makes a big impact on user satisfaction.

Figure 2-1 presents a tiny fragment of a 3NF Schema. Note how order information is broken into order and order items to avoid redundant data storage. The "crow's feet" markings on the relationship between tables indicate one-to-many relationships among the entities. Thus, one order may have multiple order items, a single customer may have many orders, and a single product may be found in many order items. Although this diagram shows a very small case, you can see that minimizing data redundancy can lead to many tables in the schema.

**Figure 2-1 Fragment of a Third Normal Form Schema**




**See Also**

Design Concepts for 3NF Schemas

## 2.3.1 About Normalization

Normalization is a data design process that has a high level goal of keeping each fact in just one place to avoid data redundancy and insert, update, and delete anomalies. There are multiple levels of normalization, and this section describes the first three of them. Considering how fundamental the term third normal form (3NF) term is, it only makes sense to see how 3NF is reached.

Consider a situation where you are tracking sales. The core entity you track is sales orders, where each sales order contains details about each item purchased (referred to as a line item): its name, price, quantity, and so on. The order also holds the name and address of the customer and more. Some orders have many different line items, and some orders have just one.

In first normal form (1NF), there are no repeating groups of data and no duplicate rows. Every intersection of a row and column (a field) contains just one value, and there are no groups of columns that contain the same facts. To avoid duplicate rows, there is a primary key. For sales orders, in first normal form, multiple line items of each sales order in a single field of the table are not displayed. Also, there will not be multiple columns showing line items.


Copyright © 2001, 2025, Oracle and/or its affiliates.

Then comes second normal form (2NF), where the design is in first normal form and every non-key column is dependent on the complete primary key. Thus, the line items are broken out into a table of sales order line items where each row represents one line item of one order. You can look at the line item table and see that the names of the items sold are not dependent on the primary key of the line items table: the sales item is its own entity. Therefore, you move the sales item to its own table showing the item name. Prices charged for each item can vary by order (for instance, due to discounts) so these remain in the line items table. In the case of sales order, the name and address of the customer is not dependent on the primary key of the sales order: customer is its own entity. Thus, you move the customer name and address columns out into their own table of customer information.

Next is third normal form, where the goal is to ensure that there are no dependencies on non-key attributes. So the goal is to take columns that do not directly relate to the subject of the row (the primary key), and put them in their own table. So details about customers, such as customer name or customer city, should be put in a separate table, and then a customer foreign key added into the orders table.

Another example of how a 2NF table differs from a 3NF table would be a table of the winners of tennis tournaments that contained columns of tournament, year, winner, and winner's date of birth. In this case, the winner's date of birth is vulnerable to inconsistencies, as the same person could be shown with different dates of birth in different records. The way to avoid this potential problem is to break the table into one for tournament winners, and another for the player dates of birth.

## 2.3.2 Design Concepts for 3NF Schemas

The following section discusses some basic concepts when modeling for a data warehousing environment using a 3NF schema approach. The intent is not to discuss the theoretical foundation for 3NF modeling (or even higher levels of normalization), but to highlight some key components relevant for data warehousing.

Some key 3NF schema design concepts that are relevant to data warehousing are as follows:

- Identifying Candidate Primary Keys
- Foreign Key Relationships and Referential Integrity Constraints
- Denormalization

### 2.3.2.1 Identifying Candidate Primary Keys

A primary key is an attribute that uniquely identifies a specific record in a table. Primary keys can be identified through single or multiple columns. It is normally preferred to achieve unique identification through as little columns as possible - ideally one or two - and to either use a column that is most likely not going to be updated or even changed in bulk. If your data model does not lead to a simple unique identification through its attributes, you would require too many attributes to uniquely identify a single records, or the data is prone to changes, the usage of a surrogate key is highly recommended.

Specifically, 3NF schemas rely on proper and simple unique identification since queries tend to have many table joins and all columns necessary to uniquely identify a record are needed as join condition to avoid row duplication through the join.

### 2.3.2.2 Foreign Key Relationships and Referential Integrity Constraints

3NF schemas in data warehousing environments often resemble the data model of its OLTP source systems, in which the logical consistency between data entities is expressed and enforced through primary key - foreign key relationships, also known as parent-child


Copyright © 2001, 2025, Oracle and/or its affiliates.

relationship. A foreign key resolves a 1-to-many relationship in relational system and ensures logical consistency: for example, you cannot have an order line item without an order header, or an employee working for a non-existent department.

While such referential are always enforced in OLTP system, data warehousing systems often implement them as declarative, non-enforced conditions, relying on the ETL process to ensure data consistency. Whenever possible, foreign keys and referential integrity constraints should be defined as non-enforced conditions, since it enables better query optimization and cardinality estimates.

### 2.3.2.3 Denormalization

Proper normalized modelling tends to decompose logical entities - such as a customer. a product, or an order - into many physical tables, making even the retrieval of perceived simple information requiring to join many tables. While this is not a problem from a query processing perspective, it can put some unnecessary burden on both the application developer (for writing code) as well as the database (for joining information that is always used together). It is not uncommon to see some sensible level of denormalization in 3NF data warehousing models, in a logical form as views or in a physical form through slightly denormalized tables.

Care has to be taken with the physical denormalization to preserve the subject-neutral shape and therefore the flexibility of the physical implementation of the 3NF schema.

## 2.4 About Star Schemas

Star schemas are often found in data warehousing systems with embedded logical or physical data marts. The term star schema is another way of referring to a "dimensional modeling" approach to defining your data model. Most descriptions of dimensional modeling use terminology drawn from the work of Ralph Kimball, the pioneering consultant and writer in this field. Dimensional modeling creates multiple star schemas, each based on a business process such as sales tracking or shipments. Each star schema can be considered a data mart, and perhaps as few as 20 data marts can cover the business intelligence needs of an enterprise. Compared to 3NF designs, the number of tables involved in dimensional modeling is a tiny fraction. Many star schemas will have under a dozen tables. The star schemas are knit together through conformed dimensions and conformed facts. Thus, users are able to get data from multiple star schemas with minimal effort.

The goal for star schemas is structural simplicity and high performance data retrieval. Because most queries in the modern era are generated by reporting tools and applications, it's vital to make the query generation convenient and reliable for the tools and application. In fact, many business intelligence tools and applications are designed with the expectation that a star schema representation will be available to them.

Discussions of star schemas are less abstracted from the physical database than 3NF descriptions. This is due to the pragmatic emphasis of dimensional modeling on the needs of business intelligence users.

Note how different the dimensional modeling style is from the 3NF approach that minimizes data redundancy and the risks of update/inset/delete anomalies. The star schema accepts data redundancy (denormalization) in its dimension tables for the sake of easy user understanding and better data retrieval performance. A common criticism of star schemas is that they limit analysis flexibility compared to 3NF designs. However, a well designed dimensional model can be extended to enable new types of analysis, and star schemas have been successful for many years at the largest enterprises.

As noted earlier, the modern approach to data warehousing does not pit star schemas and 3NF against each other. Rather, both techniques are used, with a foundation layer of 3NF - the


Copyright © 2001, 2025, Oracle and/or its affiliates.

Enterprise Data Warehouse of 3NF, acting as the bedrock data, and star schemas as a central part of an access and performance optimization layer.

**See Also**

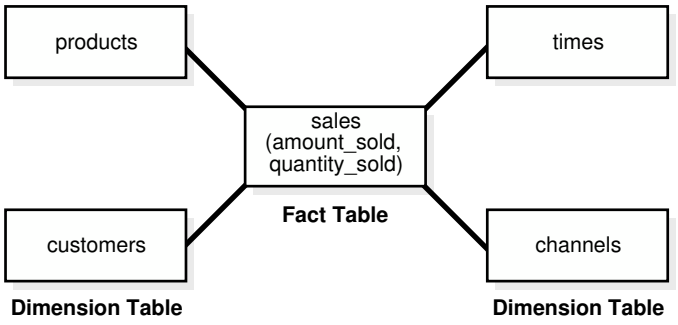
- About Facts and Dimensions in Star Schemas
- Design Concepts in Star Schemas

## 2.4.1 About Facts and Dimensions in Star Schemas

Star schemas divide data into facts and dimensions. Facts are the measurements of some event such as a sale and are typically numbers. Dimensions are the categories you use to identify facts, such as date, location, and product.

The name "star schema" comes from the fact that the diagrams of the schemas typically show a central fact table with lines joining it to the dimension tables, so the graphic impression is similar to a star. Figure 2-2 is a simple example with sales as the fact table and products, times, customers, and channels as the dimension table.

**Figure 2-2 Star Schema**




**See Also**

- About Fact Tables in Data Warehouses
- About Dimension Tables in Data Warehouses

### 2.4.1.1 About Fact Tables in Data Warehouses

Fact tables have measurement data. They have many rows but typically not many columns. Fact tables for a large enterprise can easily hold billions of rows. For many star schemas, the fact table will represent well over 90 percent of the total storage space. A fact table has a composite key made up of the primary keys of the dimension tables of the schema.

A fact table contains either detail-level facts or facts that have been aggregated. Fact tables that contain aggregated facts are often called summary tables. A fact table usually contains facts with the same level of aggregation. Though most facts are additive, they can also be semi-additive or non-additive. Additive facts can be aggregated by simple arithmetical addition. A common example of this is sales. Non-additive facts cannot be added at all. An example of


Copyright © 2001, 2025, Oracle and/or its affiliates.

this is averages. Semi-additive facts can be aggregated along some of the dimensions and not along others. An example of this is inventory levels stored in physical warehouses, where you may be able to add across a dimension of warehouse sites, but you cannot aggregate across time.

In terms of adding rows to data in a fact table, there are three main approaches:

- Transaction-based

  Shows a row for the finest level detail in a transaction. A row is entered only if a transaction has occurred for a given combination of dimension values. This is the most common type of fact table.

- Periodic Snapshot

  Shows data as of the end of a regular time interval, such as daily or weekly. If a row for the snapshot exists in a prior period, a row is entered for it in the new period even if no activity related to it has occurred in the latest interval. This type of fact table is useful in complex business processes where it is difficult to compute snapshot values from individual transaction rows.

- Accumulating Snapshot

  Shows one row for each occurrence of a short-lived process. The rows contain multiple dates tracking major milestones of a short-lived process. Unlike the other two types of fact tables, rows in an accumulating snapshot are updated multiple times as the tracked process moves forward.

## 2.4.1.2 About Dimension Tables in Data Warehouses

Dimension tables provide category data to give context to the fact data. For instance, a star schema for sales data will have dimension tables for product, date, sales location, promotion and more. Dimension tables act as lookup or reference tables because their information lets you choose the values used to constrain your queries. The values in many dimension tables may change infrequently. As an example, a dimension of geographies showing cities may be fairly static. But when dimension values do change, it is vital to update them fast and reliably. Of course, there are situations where data warehouse dimension values change frequently. The customer dimension for an enterprise will certainly be subject to a frequent stream of updates and deletions.

A key aspect of dimension tables is the hierarchy information they provide. Dimension data typically has rows for the lowest level of detail plus rows for aggregated dimension values. These natural rollups or aggregations within a dimension table are called hierarchies and add great value for analyses. For instance, if you want to calculate the share of sales that a specific product represents within its specific product category, it is far easier and more reliable to have a predefined hierarchy for product aggregation than to specify all the elements of the product category in each query. Because hierarchy information is so valuable, it is common to find multiple hierarchies reflected in a dimension table.

Dimension tables are usually textual and descriptive, and you will use their values as the row headers, column headers and page headers of the reports generated by your queries. While dimension tables have far fewer rows than fact tables, they can be quite wide, with dozens of columns. A location dimension table might have columns indicating every level of its rollup hierarchy, and may show multiple hierarchies reflected in the table. The location dimension table could have columns for its geographic rollup, such as street address, postal code, city, state/province, and country. The same table could include a rollup hierarchy set up for the sales organization, with columns for sales district, sales territory, sales region, and characteristics.


Copyright © 2001, 2025, Oracle and/or its affiliates.

> **See Also**
>
> Dimensions for further information regarding dimensions

## 2.4.2 Design Concepts in Star Schemas

Here we touch on some of the key terms used in star schemas. This is by no means a full set, but is intended to highlight some of the areas worth your consideration.

### Data Grain

One of the most important tasks when designing your model is to consider the level of detail it will provide, referred to as the grain of the data. Consider a sales schema: will the grain be very fine, storing every single item purchased by each customer? Or will it be a coarse grain, storing only the daily totals of sales for each product at each store? In modern data warehousing there is a strong emphasis on providing the finest grain data possible, because this allows for maximum analytic power. Dimensional modeling experts generally recommend that each fact table store just one grain level. Presenting fact data in single-grain tables supports more reliable querying and table maintenance, because there is no ambiguity about the scope of any row in a fact table.

### Working with Multiple Star Schemas

Because the star schema design approach is intended to chunk data into distinct processes, you need reliable and performant ways to traverse the schemas when queries span multiple schemas. One term for this ability is a data warehouse bus architecture. A data warehouse bus architecture can be achieved with conformed dimensions and conformed facts.

### Conformed Dimensions

Conformed dimensions means that dimensions are designed identically across the various star schemas. Conformed dimensions use the same values, column names and data types consistently across multiple stars. The conformed dimensions do not have to contain the same number of rows in each schema's copy of the dimension table, as long as the rows in the shorter tables are a true subset of the larger tables.

### Conformed Facts

If the fact columns in multiple fact tables have exactly the same meaning, then they are considered conformed facts. Such facts can be used together reliably in calculations even though they are from different tables. Conformed facts should have the same column names to indicate their conformed status. Facts that are not conformed should always have different names to highlight their different meanings.

### Surrogate Keys

Surrogate or artificial keys, usually sequential integers, are recommended for dimension tables. By using surrogate keys, the data is insulated from operational changes. Also, compact integer keys may allow for better performance than large and complex alphanumeric keys.

### Degenerate Dimensions

Degenerate dimensions are dimension columns in fact tables that do not join to a dimension table. They are typically items such as order numbers and invoice numbers. You will see them when the grain of a fact table is at the level of an order line-item or a single transaction.


Copyright © 2001, 2025, Oracle and/or its affiliates.

### Junk Dimensions

Junk dimensions are abstract dimension tables used to hold text lookup values for flags and codes in fact tables. These dimensions are referred to as junk, not because they have low value, but because they hold an assortment of columns for convenience, analogous to the idea of a "junk drawer" in your home. The number of distinct values (cardinality) of each column in a junk dimension table is typically small.

### Embedded Hierarchy

Classic dimensional modeling with star schemas advocates that each table contain data at a single grain. However, there are situations where designers choose to have multiple grains in a table, and these commonly represent a rollup hierarchy. A single sales fact table, for instance, might contain both transaction-level data, then a day-level rollup by product, then a month-level rollup by product. In such cases, the fact table will need to contain a level column indicating the hierarchy level applying to each row, and queries against the table will need to include a level predicate.

### Factless Fact Tables

Factless fact tables do not contain measures such as sales price or quantity sold. Instead, the rows of a factless fact table are used to show events not represented by other fact tables. Another use for factless tables is as a "coverage table" which holds all the possible events that could have occurred in a given situation, such as all the products that were part of a sales promotion and might have been sold at the promotional price.

### Slowly Changing Dimensions

One of the certainties of data warehousing is that the way data is categorized will change. Product names and category names will change. Characteristics of a store will change. The areas included in sales territories will change. The timing and extent of these changes will not always be predictable. How can these slowly changing dimensions be handled? Star schemas treat these in three main ways:

- Type 1

  The dimension values that change are simply overwritten, with no history kept. This creates a problem for time-based analyses. Also, it invalidates any existing aggregates that depended on the old value of the dimension.

- Type 2

  When a dimension value changes, a new dimension row showing the new value and having a new surrogate key is created. You may choose to include date columns in our dimension showing when the new row is valid and when it is expired. No changes need be made to the fact table.

- Type 3

  When a dimension value is changed, the prior value is stored in a different column of the same row. This enables easy query generation if you want to compare results using the current and prior value of the column.

In practice, Type 2 is the most common treatment for slowly changing dimensions.

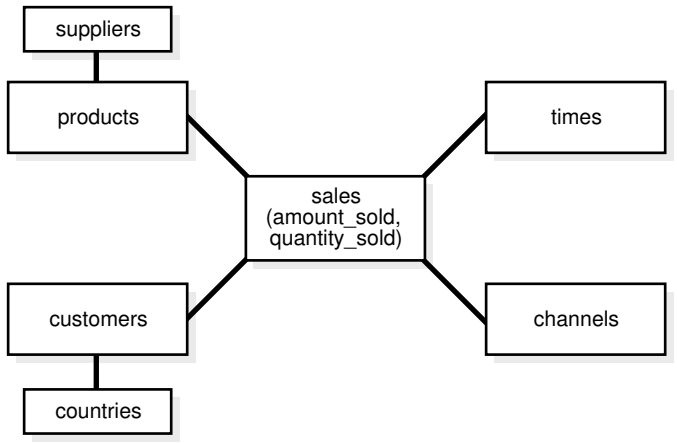

Copyright © 2001, 2025, Oracle and/or its affiliates.

## 2.4.3 About Snowflake Schemas

The snowflake schema is a more complex data warehouse model than a star schema, and is a type of star schema. It is called a snowflake schema because the diagram of the schema resembles a snowflake.

Snowflake schemas normalize dimensions to eliminate redundancy. That is, the dimension data has been grouped into multiple tables instead of one large table. For example, a product dimension table in a star schema might be normalized into a `products` table, a `product_category` table, and a `product_manufacturer` table in a snowflake schema. While this saves space, it increases the number of dimension tables and requires more foreign key joins. The result is more complex queries and reduced query performance. Figure 2-3 presents a graphical representation of a snowflake schema.

**Figure 2-3 Snowflake Schema**




# 2.5 Improved Analytics Using the In-Memory Column Store

The In-Memory column store (IM column store) is an optional portion of the system global area (SGA) that stores copies of tables, table partitions, and other database objects in a compressed columnar format that is optimized for rapid scans.

Columnar format lends itself easily to vector processing thus making aggregations, joins, and certain types of data retrieval faster than the traditional on-disk formats. The columnar format exists only in memory and does not replace the on-disk or buffer cache format. Instead, it supplements the buffer cache and provides an additional, transaction-consistent, copy of the table that is independent of the disk format.

Traditional analytics have certain limitations or requirements that need to be managed to obtain good performance for analytic queries. You need to know user access patterns and then customize your data structures to provide optimal performance for these access patterns. Existing indexes, materialized views need to be tuned. Certain data marts and reporting databases have complex ETL and thus need specialized tuning. Additionally, you need to strike a balance between performing analytics on stale data and slowing down OLTP operations on the production databases.

The Oracle In-Memory Column Store (IM column store) within the Oracle AI Database provides improved performance for both ad-hoc queries and analytics on live data. The live


Copyright © 2001, 2025, Oracle and/or its affiliates.

transactional database is used to provide instant answers to queries, thus enabling you to seamlessly use the same database for OLTP transactions and data warehouse analytics.

The IM column store integrates seamlessly with the Oracle AI Database and provides the following benefits in data warehousing environments:

- Improved query performance
  - Processing of ad-hoc queries with unanticipated access patterns is faster

    IM column store provides fast throughput for analyzing large amounts of data. Querying a subset of columns in a table provides quick results because only the columns necessary for the specific data analysis task are scanned.
  - Scanning of large number of rows and the application of filters that use operators such as =,<,>, and IN are faster with the use of SIMD vector processing
  - Storing frequently evaluated expressions using IM expressions reduces repeated computations of the same expressions
  - Using IM virtual columns and populating specified virtual columns into the IM column store avoids repeated evaluation of virtual columns
- Enhanced join performance using bloom filters

  Certain types of joins run faster when the tables being joined are stored in the IM column store. IM column store takes advantage of bloom filters with hash joins that speed up joins by converting predicates on small dimension tables to filters on a large fact table.
- Efficient aggregation using `VECTOR GROUP BY` transformation and vector array processing

  Queries that aggregate data and join one or more relatively small tables to a larger table, as often occurs in a star query, run faster. VECTOR GROUP BY will be chosen by the optimizer based on cost estimates.
- Reduced storage space and significantly less processing overhead because fewer indexes, materialized views are required when IM column store is used.

> **See Also**
>
> *Oracle AI Database In-Memory Guide* for detailed information about using the IM column store

## 2.5.1 About Improving Query Performance Using In-Memory Expressions

When you use the In-Memory Column Store (IM column store), query performance can be further enhanced by using In-Memory Expressions (IM expressions) for frequently evaluated expressions.

Most queries in a data warehousing environment involve querying large data sets and are computationally intensive as they contain complex expressions or calculations. IM expressions provide enhanced performance for queries that contain frequently evaluated expressions. The optimizer automatically identifies and records repeatedly used expressions in the Expression Statistics Store (ESS). Expressions captured in the ESS are candidates for IM expressions. To facilitate reuse, IM expressions are materialized and populated into In-Memory Expression Units (IMEUs) within the IM column store. The database then maintains IM expressions and ensures that they are consistent with any modifications made to the source columns on which


Copyright © 2001, 2025, Oracle and/or its affiliates.

these expressions are based. Populating IM expressions into the IM column store reduces repeated computations of the same expressions.

For example, total cost, which is a product of the price and number of units sold, is a candidate for an IM expression. Without IM expressions, the value of total cost needs to be recomputed for every query and for every row returned by the query. With IM expressions, this frequently evaluated expression can be materialized and stored in the IM column store. This eliminates the need to repeatedly recompute the expression used in the query. Oracle AI Database rewrites the queries at runtime to use expression results stored in the IM column store thereby improving query performance.

The initialization parameter `INMEMORY_EXPRESSIONS_USAGE` controls which IM expressions must be populated into the IM column store. Procedures in the `DBMS_INMEMORY_ADMIN` package specify when IM expressions are identified, populated, and used.

**Related Topics**

- *Oracle AI Database In-Memory Guide*

## 2.5.2 About Using In-Memory Virtual Columns to Improve Query Performance

When you use the In-Memory Column Store (IM column store), In-Memory virtual columns (IM virtual columns) enable you to avoid repeated evaluations of virtual columns by populating specified virtual columns into the IM column store.

Virtual columns are user-created, named expressions that Oracle treats like regular columns. For example, if the `SALARY` table contains the column `monthly_salary`, you can define a virtual column called `annual_salary` as `monthly_salary * 12`. IM virtual columns are virtual columns that can be populated into the IM column store. You can populate all or a subset of the virtual columns defined in a table into the IM column store. Storing precomputed virtual columns in the IM column store improves query performance by avoiding repeated evaluations. Virtual column values can also be scanned and filtered using in-memory techniques such as SIMD vector processing.

The initialization parameter `INMEMORY_VIRTUAL_COLUMNS` determines if IM virtual columns must be created for tables enabled for IM column store.

**Related Topics**

- *Oracle AI Database In-Memory Guide*

## 2.5.3 About In-Memory Column Store and Automatic Data Optimization

Automatic Data Optimization (ADO) can be used to manage the contents of the In-Memory Column Store (IM column store).

The performance benefits provided by the IM column store can be optimized by effectively managing the contents of the IM column store. Objects that benefit most from being stored in the IM column store must be retained. This requires a constant monitoring of the IM column store to determine which objects must be retained and which objects must be removed from the IM column store.

Automatic Data Optimization (ADO) automates the management of the IM column store contents. Heat map statistics are gathered for objects in the IM column store and these statistics are used to determine the least active and the most active objects. You can define ADO policies to specify when objects are eligible to be moved out of the IM column store.


Copyright © 2001, 2025, Oracle and/or its affiliates.

In data warehousing applications, the frequency with which objects are accessed typically decreases over time. Therefore, objects are accessed most frequently when they are first loaded in to the data warehouse and the activity levels decrease subsequently. Data warehouse performance can be enhanced by defining ADO policies that move objects that are accessed the least out of the IM column store.

**Related Topics**

- *Oracle AI Database In-Memory Guide*

# 2.6 Automatic Big Table Caching to Improve the Performance of In-Memory Parallel Queries

Automatic big table caching enhances the in-memory query capabilities of Oracle AI Database. When a table does not fit in memory, the database decides which buffers to cache based on access patterns. This provides efficient caching for large tables, even if they do not fully fit in the buffer cache.

An optional section of the buffer cache, called the big table cache, is used to store data for table scans. The big table cache is integrated with the buffer cache and uses a temperature-based, object-level replacement algorithm to manage the big table cache contents. This is different from the access-based, block level LRU algorithm used by the buffer cache.

> **Note**
>
> The automatic big table caching feature is available starting with Oracle Database 12*c* Release 1 (12.1.0.2).

Typical data warehousing workloads scan multiple tables. Performance may be impacted if the combined size of these tables is greater than the combined size of the buffer cache. With automatic big table caching, the scanned tables are stored in the big table cache instead of the buffer cache. The temperature-based, object-level replacement algorithm used by the big table cache can provide enhanced performance for data warehousing workloads by:

- Selectively caching the "hot" objects

  Each time an object is accessed, Oracle AI Database increments the temperature of that object. An object in the big table cache can be replaced only by another object whose temperature is higher than its own temperature.

- Avoiding thrashing

  Partial objects are cached when objects cannot be fully cached.

In Oracle Real Application Clusters (Oracle RAC) environments, automatic big table caching is supported only for parallel queries. In single instance environments, this functionality is supported for both serial and parallel queries.

To use automatic big table caching, you must enable the big table cache. To use automatic big table caching for serial queries, you must set the `DB_BIG_TABLE_CACHE_PERCENT_TARGET` initialization parameter to a nonzero value. To use automatic big table caching for parallel queries, you must set `PARALLEL_DEGREE_POLICY` to `AUTO` or `ADAPTIVE` and `DB_BIG_TABLE_CACHE_PERCENT_TARGET` to a nonzero value.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**See Also**

*Oracle AI Database VLDB and Partitioning Guide* for more information about the big table cache and how it can be used


Copyright © 2001, 2025, Oracle and/or its affiliates.

3

# Data Warehousing Physical Design

This chapter describes the physical design of a data warehousing environment, and includes the following topics:


- Moving from Logical to Physical Design
- About Physical Design


## 3.1 Moving from Logical to Physical Design

Logical design is what you draw with a pen and paper or design with a tool such as Oracle Designer before building your data warehouse. Physical design is the creation of the database with SQL statements.

During the physical design process, you convert the data gathered during the logical design phase into a description of the physical database structure. Physical design decisions are mainly driven by query performance and database maintenance aspects. For example, choosing a partitioning strategy that meets common query requirements enables Oracle AI Database to take advantage of partition pruning, a way of narrowing a search before performing it.

**See Also**

- *Oracle AI Database VLDB and Partitioning Guide* for further information regarding partitioning
- *Oracle AI Database Concepts* for further conceptual material regarding design matters.

## 3.2 About Physical Design

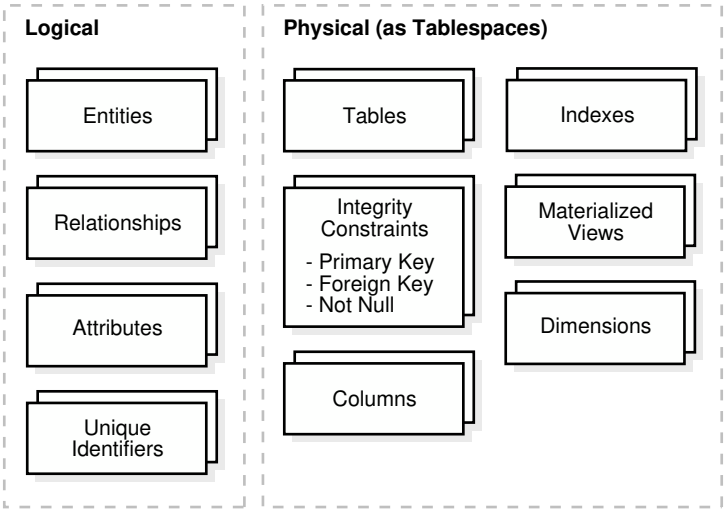
During the logical design phase, you defined a model for your data warehouse consisting of entities, attributes, and relationships. The entities are linked together using relationships. Attributes are used to describe the entities. The unique identifier (UID) distinguishes between one instance of an entity and another.

Figure 3-1 illustrates a graphical way of distinguishing between logical and physical designs.



Copyright © 2001, 2025, Oracle and/or its affiliates.

**Figure 3-1 Logical Design Compared with Physical Design**




During the physical design process, you translate the expected schemas into actual database structures. At this time, you must map:

- Entities to tables
- Relationships to foreign key constraints
- Attributes to columns
- Primary unique identifiers to primary key constraints
- Unique identifiers to unique key constraints

## 3.2.1 Physical Design Structures

To convert your logical design into a physical design, you must create some or all of the following structures: tablespaces, tables, partitions on tables or index-organized tables, indexes including partitioned indexes, views, integrity constraints, materialized views, and dimensions.

### 3.2.1.1 About Tablespaces in Data Warehouses

A tablespace consists of one or more datafiles, which are physical structures within the operating system you are using. A datafile is associated with only one tablespace. From a design perspective, tablespaces are containers for physical design structures.

Tablespaces need to be separated by differences. For example, tables should be separated from their indexes and small tables should be separated from large tables. Tablespaces should also represent logical business units if possible. Because a tablespace is the coarsest granularity for backup and recovery or the transportable tablespaces mechanism, the logical business design affects availability and maintenance operations.

You can now use ultralarge data files, a significant improvement in very large databases.


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 3.2.1.2 About Partitioning in Data Warehouses

Oracle partitioning is an extremely important functionality for data warehousing, improving manageability, performance and availability. This section presents the key concepts and benefits of partitioning noting special value for data warehousing.

Partitioning allows tables, indexes or index-organized tables to be subdivided into smaller pieces. Each piece of the database object is called a partition. Each partition has its own name, and may optionally have its own storage characteristics. From the perspective of a database administrator, a partitioned object has multiple pieces that can be managed either collectively or individually. This gives the administrator considerable flexibility in managing a partitioned object. However, from the perspective of the user, a partitioned table is identical to a non-partitioned table; no modifications are necessary when accessing a partitioned table using SQL DML commands.

Database objects - tables, indexes, and index-organized tables - are partitioned using a partitioning key, a set of columns that determine in which partition a given row will reside. For example a sales table partitioned on sales date, using a monthly partitioning strategy; the table appears to any application as a single, normal table. However, the DBA can manage and store each monthly partition individually, potentially using different storage tiers, applying table compression to the older data, or store complete ranges of older data in read only tablespaces.

### 3.2.1.2.1 Basic Partitioning Strategies Used in Data Warehouses

Oracle partitioning offers three fundamental data distribution methods that control how the data is actually placed into the various individual partitions, namely:

- Range

  The data is distributed based on a range of values of the partitioning key (for a date column as the partitioning key, the 'January-2012' partition contains rows with the partitioning key values between '01-JAN-2012' and '31-JAN-2012'). The data distribution is a continuum without any holes and the lower boundary of a range is automatically defined by the upper boundary of the preceding range.

- List

  The data distribution is defined by a list of values of the partitioning key (for a region column as the partitioning key, the `North_America` partition may contain values `Canada`, `USA`, and `Mexico`). A special `DEFAULT` partition can be defined to catch all values for a partition key that are not explicitly defined by any of the lists.

- Hash

  A hash algorithm is applied to the partitioning key to determine the partition for a given row. Unlike the other two data distribution methods, hash does not provide any logical mapping between the data and any partition.

Along with these fundamental approaches Oracle AI Database provides several more:

- Interval Partitioning

  An extension to range partitioning that enhances manageability. Partitions are defined by an interval, providing equi-width ranges. With the exception of the first partition all partitions are automatically created on-demand when matching data arrives.

- Partitioning by Reference

  Partitioning for a child table is inherited from the parent table through a primary key - foreign key relationship. Partition maintenance is simplified and partition-wise joins enabled.


Copyright © 2001, 2025, Oracle and/or its affiliates.

- Virtual column based Partitioning

  Defined by one of the above mentioned partition techniques and the partitioning key is based on a virtual column. Virtual columns are not stored on disk and only exist as metadata. This approach enables a more flexible and comprehensive match of the business requirements.

Using the above-mentioned data distribution methods, a table can be partitioned either as single or composite partitioned table:

- Single (one-level) Partitioning

  A table is defined by specifying one of the data distribution methodologies, using one or more columns as the partitioning key. For example consider a table with a number column as the partitioning key and two partitions `less_than_five_hundred` and `less_than_thousand`, the `less_than_thousand` partition contains rows where the following condition is true: `500 <= Partitioning key <1000`.

  You can specify range, list, and hash partitioned tables.

- Composite Partitioning
- Combinations of two data distribution methods are used to define a composite partitioned table. First, the table is partitioned by data distribution method one and then each partition is further subdivided into subpartitions using a second data distribution method. All sub-partitions for a given partition together represent a logical subset of the data. For example, a range-hash composite partitioned table is first range-partitioned, and then each individual range-partition is further subpartitioned using the hash partitioning technique.

**See Also**

- *Oracle AI Database VLDB and Partitioning Guide*
- *Oracle AI Database Concepts* for more information about Hybrid Columnar Compression

## 3.2.1.3 Index Partitioning in Data Warehouses

Irrespective of the chosen index partitioning strategy, an index is either coupled or uncoupled with the partitioning strategy of the underlying table. The appropriate index partitioning strategy is chosen based on the business requirements, making partitioning well suited to support any kind of application. Oracle Database 12*c* differentiates between three types of partitioned indexes.

- Local Indexes

  A local index is an index on a partitioned table that is coupled with the underlying partitioned table, 'inheriting' the partitioning strategy from the table. Consequently, each partition of a local index corresponds to one - and only one - partition of the underlying table. The coupling enables optimized partition maintenance; for example, when a table partition is dropped, Oracle AI Database simply has to drop the corresponding index partition as well. No costly index maintenance is required. Local indexes are most common in data warehousing environments.

- Global Partitioned Indexes

  A global partitioned index is an index on a partitioned or nonpartitioned table that is partitioned using a different partitioning-key or partitioning strategy than the table. Global-partitioned indexes can be partitioned using range or hash partitioning and are uncoupled


Copyright © 2001, 2025, Oracle and/or its affiliates.

from the underlying table. For example, a table could be range-partitioned by month and have twelve partitions, while an index on that table could be hash-partitioned using a different partitioning key and have a different number of partitions. Global partitioned indexes are more common for OLTP than for data warehousing environments.

- Global Non-Partitioned Indexes

  A global non-partitioned index is essentially identical to an index on a non-partitioned table. The index structure is not partitioned and uncoupled from the underlying table. In data warehousing environments, the most common usage of global non-partitioned indexes is to enforce primary key constraints.

### 3.2.1.4 About Partitioning for Manageability

A typical usage of partitioning for manageability is to support a 'rolling window' load process in a data warehouse. Suppose that a DBA loads new data into a table on a daily basis. That table could be range partitioned so that each partition contains one day of data. The load process is simply the addition of a new partition. Adding a single partition is much more efficient than modifying the entire table, because the DBA does not need to modify any other partitions. Another advantage of using partitioning is when it is time to remove data. In this situation, an entire partition can be dropped, which is very efficient and fast, compared to deleting each row individually.

### 3.2.1.5 About Partitioning for Performance

By limiting the amount of data to be examined or operated on, partitioning provides a number of performance benefits. Two features specially worth noting are:

- Partitioning pruning: Partitioning pruning is the simplest and also the most substantial means to improve performance using partitioning. Partition pruning can often improve query performance by several orders of magnitude. For example, suppose an application contains an `ORDERS` table containing an historical record of orders, and that this table has been partitioned by day. A query requesting orders for a single week would only access seven partitions of the `ORDERS` table. If the table had two years of historical data, this query would access seven partitions instead of 730 partitions. This query could potentially execute 100x faster simply because of partition pruning. Partition pruning works with all of Oracle's other performance features. Oracle AI Database will utilize partition pruning in conjunction with any indexing technique, join technique, or parallel access method.
- Partition-wise joins: Partitioning can also improve the performance of multi-table joins, by using a technique known as partition-wise joins. Partition-wise joins can be applied when two tables are being joined together, and at least one of these tables is partitioned on the join key. Partition-wise joins break a large join into smaller joins of 'identical' data sets for the joined tables. 'Identical' here is defined as covering exactly the same set of partitioning key values on both sides of the join, thus ensuring that only a join of these 'identical' data sets will produce a result and that other data sets do not have to be considered. Oracle AI Database is using either the fact of already (physical) equi-partitioned tables for the join or is transparently redistributing ("repartitioning") one table at runtime to create equipartitioned data sets matching the partitioning of the other table, completing the overall join in less time. This offers significant performance benefits both for serial and parallel execution.

### 3.2.1.6 About Partitioning for Availability

Partitioned database objects provide partition independence. This characteristic of partition independence can be an important part of a high-availability strategy. For example, if one partition of a partitioned table is unavailable, all of the other partitions of the table remain online


Copyright © 2001, 2025, Oracle and/or its affiliates.

and available. The application can continue to execute queries and transactions against this partitioned table, and these database operations will run successfully if they do not need to access the unavailable partition. The database administrator can specify that each partition be stored in a separate tablespace; this would allow the administrator to do backup and recovery operations on an individual partition or sets of partitions (by virtue of the partition-to-tablespace mapping), independent of the other partitions in the table. Therefore in the event of a disaster, the database could be recovered with just the partitions comprising the active data, and then the inactive data in the other partitions could be recovered at a convenient time, thus decreasing the system down-time.In light of the manageability, performance and availability benefits, it should be part of every data warehouse.

**See Also**

*Oracle AI Database VLDB and Partitioning Guide*

## 3.2.2 About Views in Data Warehouses

A view is a tailored presentation of the data contained in one or more tables or other views. A view takes the output of a query and treats it as a table. Views do not require any space in the database.

**See Also**

*Oracle AI Database Concepts*

## 3.2.3 About Integrity Constraints in Data Warehouses

Integrity constraints are used to enforce business rules associated with your database and to prevent having invalid information in the tables. Integrity constraints in data warehousing differ from constraints in OLTP environments. In OLTP environments, they primarily prevent the insertion of invalid data into a record, which is not a big problem in data warehousing environments because accuracy has already been guaranteed. In data warehousing environments, constraints are only used for query rewrite. `NOT NULL` constraints are particularly common in data warehouses. Under some specific circumstances, constraints need space in the database. These constraints are in the form of the underlying unique index.

**See Also**

*Oracle AI Database Concepts*

## 3.2.4 About Indexes and Partitioned Indexes in Data Warehouses

Indexes are optional structures associated with tables or clusters. In addition to the classical B-tree indexes, bitmap indexes are very common in data warehousing environments. Bitmap indexes are optimized index structures for set-oriented operations. Additionally, they are necessary for some optimized data access methods such as star transformations.


Copyright © 2001, 2025, Oracle and/or its affiliates.

Indexes are just like tables in that you can partition them, although the partitioning strategy is not dependent upon the table structure. Partitioning indexes makes it easier to manage the data warehouse during refresh and improves query performance.

**See Also**

- Index Partitioning in Data Warehouses
- *Oracle AI Database Concepts*

## 3.2.5 About Materialized Views in Data Warehouses

Materialized views are query results that have been stored in advance so long-running calculations are not necessary when you actually execute your SQL statements. From a physical design point of view, materialized views resemble tables or partitioned tables and behave like indexes in that they are used transparently and improve performance.

**See Also**

Basic Materialized Views

## 3.2.6 About Dimensions in Data Warehouses

A dimension is a structure, often composed of one or more hierarchies, that categorizes data. Dimensional attributes help to describe the dimensional value. They are normally descriptive, textual values. Several distinct dimensions, combined with facts, enable you to answer business questions. Commonly used dimensions are customers, products, and time.

A dimension schema object defines hierarchical relationships between columns or column sets. A hierarchical relationship is a functional dependency from one level of a hierarchy to the next one. A dimension object is a container of logical relationships and does not require any space in the database. A typical dimension is city, state (or province), region, and country.

Dimension data is typically collected at the lowest level of detail and then aggregated into higher level totals that are more useful for analysis. These natural rollups or aggregations within a dimension table are called hierarchies.

This section contains the following topics:

- About Dimension Hierarchies
- Typical Dimension Hierarchy

### 3.2.6.1 About Dimension Hierarchies

Hierarchies are logical structures that use ordered levels to organize data. A hierarchy can be used to define data aggregation. For example, in a time dimension, a hierarchy might aggregate data from the month level to the quarter level to the year level. A hierarchy can also be used to define a navigational drill path and to establish a family structure.

Within a hierarchy, each level is logically connected to the levels above and below it. Data values at lower levels aggregate into the data values at higher levels. A dimension can be


Copyright © 2001, 2025, Oracle and/or its affiliates.

composed of more than one hierarchy. For example, in the product dimension, there might be two hierarchies—one for product categories and one for product suppliers.

Dimension hierarchies also group levels from general to granular. Query tools use hierarchies to enable you to drill down into your data to view different levels of granularity. This is one of the key benefits of a data warehouse.

When designing hierarchies, you must consider the relationships in business structures. For example, a divisional multilevel sales organization can have complicated structures.

Hierarchies impose a family structure on dimension values. For a particular level value, a value at the next higher level is its parent, and values at the next lower level are its children. These familial relationships enable analysts to access data quickly.

**See Also**

- About Levels
- About Level Relationships

#### 3.2.6.1.1 About Levels

A level represents a position in a hierarchy. For example, a time dimension might have a hierarchy that represents data at the month, quarter, and year levels. Levels range from general to specific, with the root level as the highest or most general level. The levels in a dimension are organized into one or more hierarchies.

#### 3.2.6.1.2 About Level Relationships

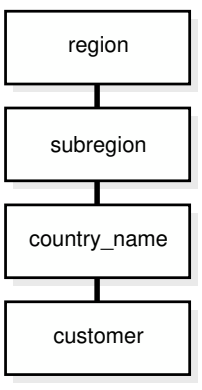
Level relationships specify top-to-bottom ordering of levels from most general (the root) to most specific information. They define the parent-child relationship between the levels in a hierarchy.

Hierarchies are also essential components in enabling more complex rewrites. For example, the database can aggregate an existing sales revenue on a quarterly base to a yearly aggregation when the dimensional dependencies between quarter and year are known.

### 3.2.6.2 Typical Dimension Hierarchy

Figure 3-2 illustrates a dimension hierarchy based on `customers`.

**Figure 3-2 Typical Levels in a Dimension Hierarchy**





Copyright © 2001, 2025, Oracle and/or its affiliates.

Copyright © 2001, 2025, Oracle and/or its affiliates.

4

# Data Warehousing Optimizations and Techniques

The following topics provide information about schemas in a data warehouse:

- Using Indexes in Data Warehouses
- Using Integrity Constraints in a Data Warehouse
- About Parallel Execution in Data Warehouses
- About Optimizing Storage Requirements in Data Warehouses
- Optimizing Star Queries and 3NF Schemas
- About Approximate Query Processing
- About Approximate Top-N Query Processing

## 4.1 Using Indexes in Data Warehouses

Indexes enable faster retrieval of data stored in data warehouses. This section discusses the following aspects of using indexes in data warehouses:

- About Using Bitmap Indexes in Data Warehouses
- Benefits of Indexes for Data Warehousing Applications
- About Cardinality and Bitmap Indexes
- How to Determine Candidates for Using a Bitmap Index
- Using Bitmap Join Indexes in Data Warehouses
- Using B-Tree Indexes in Data Warehouses
- Using Index Compression
- Choosing Between Local Indexes and Global Indexes

### 4.1.1 About Using Bitmap Indexes in Data Warehouses

Bitmap indexes are widely used in data warehousing environments. The environments typically have large amounts of data and ad hoc queries, but a low level of concurrent DML transactions. For such applications, bitmap indexing provides:

- Reduced response time for large classes of ad hoc queries.
- Reduced storage requirements compared to other indexing techniques.
- Dramatic performance gains even on hardware with a relatively small number of CPUs or a small amount of memory.

Fully indexing a large table with a traditional B-tree index can be prohibitively expensive in terms of disk space because the indexes can be several times larger than the data in the table. Bitmap indexes are typically only a fraction of the size of the indexed data in the table.


Copyright © 2001, 2025, Oracle and/or its affiliates.

An index provides pointers to the rows in a table that contain a given key value. A regular index stores a list of rowids for each key corresponding to the rows with that key value. In a bitmap index, a bitmap for each key value replaces a list of rowids.

Each bit in the bitmap corresponds to a possible rowid, and if the bit is set, it means that the row with the corresponding rowid contains the key value. A mapping function converts the bit position to an actual rowid, so that the bitmap index provides the same functionality as a regular index. Bitmap indexes store the bitmaps in a compressed way. If the number of distinct key values is small, bitmap indexes compress better and the space saving benefit compared to a B-tree index becomes even better.

Bitmap indexes are most effective for queries that contain multiple conditions in the `WHERE` clause. Rows that satisfy some, but not all, conditions are filtered out before the table itself is accessed. This improves response time, often dramatically. If you are unsure of which indexes to create, the SQL Access Advisor can generate recommendations on what to create. As the bitmaps from bitmap indexes can be combined quickly, it is usually best to use single-column bitmap indexes.

In addition, you should keep in mind that bitmap indexes are usually easier to destroy and re-create than to maintain.

### 4.1.1.1 About Bitmap Indexes and Nulls

Unlike most other types of indexes, bitmap indexes include rows that have `NULL` values. Indexing of nulls can be useful for some types of SQL statements, such as queries with the aggregate function `COUNT`.

**Example 4-1 Bitmap Index**

```
SELECT COUNT(*) FROM customers WHERE cust_marital_status IS NULL;
```

This query uses a bitmap index on `cust_marital_status`. Note that this query would not be able to use a B-tree index, because B-tree indexes do not store the `NULL` values.

```
SELECT COUNT(*) FROM customers;
```

Any bitmap index can be used for this query because all table rows are indexed, including those that have `NULL` data. If nulls were not indexed, the optimizer would be able to use indexes only on columns with `NOT NULL` constraints.

### 4.1.1.2 About Bitmap Indexes on Partitioned Tables

You can create bitmap indexes on partitioned tables but they must be local to the partitioned table—they cannot be global indexes. A partitioned table can only have global B-tree indexes, partitioned or nonpartitioned.

**See Also**

- *Oracle AI Database SQL Language Reference*
- *Oracle AI Database VLDB and Partitioning Guide*


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 4.1.2 Benefits of Indexes for Data Warehousing Applications

Bitmap indexes are primarily intended for data warehousing applications where users query the data rather than update it. They are not suitable for OLTP applications with large numbers of concurrent transactions modifying the data.

Indexes are more beneficial for high cardinality columns.

> **See Also**
>
> About Cardinality and Bitmap Indexes

Parallel query and parallel DML work with bitmap indexes. Bitmap indexing also supports parallel create indexes and concatenated indexes.

## 4.1.3 About Cardinality and Bitmap Indexes

The advantages of using bitmap indexes are greatest for columns in which the ratio of the number of distinct values to the number of rows in the table is small. This ratio is referred to as the degree of cardinality. A gender column, which has only two distinct values (male and female), is optimal for a bitmap index. However, data warehouse administrators also build bitmap indexes on columns with higher cardinalities.

For example, on a table with one million rows, a column with 10,000 distinct values is a candidate for a bitmap index. A bitmap index on this column can outperform a B-tree index, particularly when this column is often queried in conjunction with other indexed columns. In fact, in a typical data warehouse environments, a bitmap index can be considered for any non-unique column.

B-tree indexes are most effective for high-cardinality data: that is, for data with many possible values, such as `customer_name` or `phone_number`. In a data warehouse, B-tree indexes should be used only for unique columns or other columns with very high cardinalities (that is, columns that are almost unique). The majority of indexes in a data warehouse should be bitmap indexes.

In ad hoc queries and similar situations, bitmap indexes can dramatically improve query performance. `AND` and `OR` conditions in the `WHERE` clause of a query can be resolved quickly by performing the corresponding Boolean operations directly on the bitmaps before converting the resulting bitmap to rowids. If the resulting number of rows is small, the query can be answered quickly without resorting to a full table scan.

The following query output shows a portion of a company's `customers` table.

```
SELECT cust_id, cust_gender, cust_marital_status, cust_income_level
FROM customers;

CUST_ID    C CUST_MARITAL_STATUS  CUST_INCOME_LEVEL
---------- - -------------------- --------------------
...
        70 F                      D: 70,000 - 89,999
        80 F married              H: 150,000 - 169,999
        90 M single               H: 150,000 - 169,999
       100 F                      I: 170,000 - 189,999
       110 F married              C: 50,000 - 69,999
       120 M single               F: 110,000 - 129,999
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
       130 M                    J: 190,000 - 249,999
       140 M married            G: 130,000 - 149,999
...
```

Because `cust_gender`, `cust_marital_status`, and `cust_income_level` are all low-cardinality columns (there are only three possible values for marital status, two possible values for gender, and 12 for income level), bitmap indexes are ideal for these columns. Do not create a bitmap index on `cust_id` because this is a unique column. Instead, a unique B-tree index on this column provides the most efficient representation and retrieval.

Table 4-1 illustrates the bitmap index for the `cust_gender` column in this example. It consists of two separate bitmaps, one for gender.

**Table 4-1 Sample Bitmap Index**

| cust_id | gender='M' | gender='F' |
|---|---|---|
| cust_id 70 | 0 | 1 |
| cust_id 80 | 0 | 1 |
| cust_id 90 | 1 | 0 |
| cust_id 100 | 0 | 1 |
| cust_id 110 | 0 | 1 |
| cust_id 120 | 1 | 0 |
| cust_id 130 | 1 | 0 |
| cust_id 140 | 1 | 0 |

Each entry (or bit) in the bitmap corresponds to a single row of the `customers` table. The value of each bit depends upon the values of the corresponding row in the table. For example, the bitmap `cust_gender='F'` contains a one as its first bit because the gender is `F` in the first row of the `customers` table. The bitmap `cust_gender='F'` has a zero for its third bit because the gender of the third row is not `F`.

An analyst investigating demographic trends of the company's customers might ask, "How many of our married customers have an income level of G or H?" This corresponds to the following query:

```
SELECT COUNT(*) FROM customers
WHERE cust_marital_status = 'married'
AND cust_income_level IN ('H: 150,000 - 169,999', 'G: 130,000 - 149,999');
```

Bitmap indexes can efficiently process this query by merely counting the number of ones in the bitmap illustrated in Figure 4-1. The result set will be found by using bitmap `OR` merge operations without the necessity of a conversion to rowids. To identify additional specific customer attributes that satisfy the criteria, use the resulting bitmap to access the table after a bitmap to rowid conversion.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Figure 4-1 Executing a Query Using Bitmap Indexes**

## 4.1.4 How to Determine Candidates for Using a Bitmap Index

Bitmap indexes should help when either the fact table is queried alone, and there are predicates on the indexed column, or when the fact table is joined with two or more dimension tables, and there are indexes on foreign key columns in the fact table, and predicates on dimension table columns.

A fact table column is a candidate for a bitmap index when the following conditions are met:

- There are 100 or more rows for each distinct value in the indexed column. When this limit is met, the bitmap index will be much smaller than a regular index, and you will be able to create the index much faster than a regular index. An example would be one million distinct values in a multi-billion row table.

And either of the following are true:

- The indexed column will be restricted in queries (referenced in the `WHERE` clause).

or

- The indexed column is a foreign key for a dimension table. In this case, such an index will make star transformation more likely.

## 4.1.5 Using Bitmap Join Indexes in Data Warehouses

In addition to a bitmap index on a single table, you can create a bitmap join index, which is a bitmap index for the join of two or more tables. In a bitmap join index, the bitmap for the table to be indexed is built for values coming from the joined tables. In a data warehousing environment, the join condition is an equi-inner join between the primary key column or columns of the dimension tables and the foreign key column or columns in the fact table.

A bitmap join index can improve the performance by an order of magnitude. By storing the result of a join, the join can be avoided completely for SQL statements using a bitmap join index. Furthermore, because it is most likely to have a much smaller number of distinct values for a bitmap join index compared to a regular bitmap index on the join column, the bitmaps compress better, yielding to less space consumption than a regular bitmap index on the join column.

Bitmap join indexes are much more efficient in storage than materialized join views, an alternative for materializing joins in advance. This is because the materialized join views do not compress the rowids of the fact tables.

B-tree and bitmap indexes have different maximum column limitations.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**See Also**

- Four Join Models for Bitmap Join Indexes in Data Warehouses
- Bitmap Join Index Restrictions and Requirements
- *Oracle AI Database SQL Language Reference* for details regarding these limitations

## 4.1.5.1 Four Join Models for Bitmap Join Indexes in Data Warehouses

The most common usage of a bitmap join index is in star model environments, where a large table is indexed on columns joined by one or several smaller tables. The large table is referred to as the fact table and the smaller tables as dimension tables. The following examples describe the four different join models supported by bitmap join indexes, using the Sales History (sh) sample schema.

The following exampleshows a bitmap join index where one dimension table column joins one fact table. Unlike the example in About Cardinality and Bitmap Indexes, where a bitmap index on the cust_gender column on the customers table was built, you now create a bitmap join index on the fact table sales for the joined column customers(cust_gender). Table sales stores cust_id values only:

```
SELECT time_id, cust_id, amount_sold FROM sales WHERE time_id > '24-DEC-22';

TIME_ID         CUST_ID    AMOUNT_SOLD
_____________ ___________ _____________
25-DEC-22          3060          22.37
25-DEC-22          3130          22.37
25-DEC-22          3474          22.37
25-DEC-22          4093          22.37
25-DEC-22          7526          22.37
25-DEC-22          8102          22.37
25-DEC-22         10584          22.37
25-DEC-22         13187          22.37
...
```

To create such a bitmap join index, column customers(cust_gender) has to be joined with table sales. The join condition is specified as part of the CREATE statement for the bitmap join index as follows:

```
CREATE BITMAP INDEX sales_cust_gender_bjix
ON sales(customers.cust_gender)
FROM sales, customers
WHERE sales.cust_id = customers.cust_id
LOCAL NOLOGGING COMPUTE STATISTICS;
```

The following query shows the join result that is used to create the bitmaps that are stored in the bitmap join index:

```
SELECT * FROM (
    SELECT sales.time_id, customers.cust_gender, sales.amount_sold
    FROM sales, customers
    WHERE sales.cust_id = customers.cust_id
    ORDER BY sales.time_id DESC
)
WHERE ROWNUM <= 8;

...
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
TIME_ID      CUST_GENDER       AMOUNT_SOLD
___________  ________________  _____________
31-DEC-22    M                        68.57
31-DEC-22    M                        44.57
31-DEC-22    F                        44.57
31-DEC-22    F                        44.57
31-DEC-22    M                        44.57
31-DEC-22    M                        68.16
31-DEC-22    F                        68.16
31-DEC-22    M                        68.16
```

Table 4-2 illustrates the bitmap representation for the bitmap join index in this example.

**Table 4-2 Sample Bitmap Join Index**

| sales record | cust_gender='M' | cust_gender='F' |
|---|---|---|
| sales record 1 | 1 | 0 |
| sales record 2 | 0 | 1 |
| sales record 3 | 1 | 0 |
| sales record 4 | 1 | 0 |
| sales record 5 | 1 | 0 |
| sales record 6 | 1 | 0 |
| sales record 7 | 1 | 0 |

You can create other bitmap join indexes using more than one column or more than one table, as shown in these examples.

### Example 4-2 Bitmap Join Index: Multiple Dimension Columns Join One Fact Table

You can create a bitmap join index on more than one column from a single dimension table, as in the following example, which uses `customers(cust_gender, cust_marital_status)` from the `sh` schema:

```
CREATE BITMAP INDEX sales_cust_gender_ms_bjix
ON sales(customers.cust_gender, customers.cust_marital_status)
FROM sales, customers
WHERE sales.cust_id = customers.cust_id
LOCAL NOLOGGING COMPUTE STATISTICS;
```

### Example 4-3 Bitmap Join Index: Multiple Dimension Tables Join One Fact Table

You can create a bitmap join index on multiple dimension tables, as in the following, which uses `customers(gender)` and `products(category)`:

```
CREATE BITMAP INDEX sales_c_gender_p_cat_bjix
ON sales(customers.cust_gender, products.prod_category)
FROM sales, customers, products
WHERE sales.cust_id = customers.cust_id
AND sales.prod_id = products.prod_id
LOCAL NOLOGGING COMPUTE STATISTICS;
```

### Example 4-4 Bitmap Join Index: Snowflake Schema

You can create a bitmap join index on more than one table, in which the indexed column is joined to the indexed table by using another table. For example, you can build an index on `countries.country_name`, even though the `countries` table is not joined directly to the `sales`


Copyright © 2001, 2025, Oracle and/or its affiliates.

table. Instead, the `countries` table is joined to the `customers` table, which is joined to the `sales` table. This type of schema is commonly called a snowflake schema.

```
CREATE BITMAP INDEX sales_co_country_name_bjix
ON sales(countries.country_name)
FROM sales, customers, countries
WHERE sales.cust_id = customers.cust_id
  AND customers.country_id = countries.country_id
LOCAL NOLOGGING COMPUTE STATISTICS;
```

### 4.1.5.2 Bitmap Join Index Restrictions and Requirements

Join results must be stored, therefore, bitmap join indexes have the following restrictions:

- Parallel DML is only supported on the fact table. Parallel DML on one of the participating dimension tables will mark the index as unusable.
- Only one table can be updated concurrently by different transactions when using the bitmap join index.
- No table can appear twice in the join.
- You cannot create a bitmap join index on a temporary table.
- The columns in the index must all be columns of the dimension tables.
- The dimension table join columns must be either primary key columns or have unique constraints.
- The dimension table column(s) participating in the join with the fact table must be either the primary key column(s) or the unique constraint.
- If a dimension table has composite primary key, each column in the primary key must be part of the join.
- The restrictions for creating a regular bitmap index also apply to a bitmap join index. For example, you cannot create a bitmap index with the `UNIQUE` attribute. See *Oracle AI Database SQL Language Reference* for other restrictions.

## 4.1.6 Using B-Tree Indexes in Data Warehouses

A B-tree index is organized like an upside-down tree. The bottom level of the index holds the actual data values and pointers to the corresponding rows, much as the index in a book has a page number associated with each index entry.

In general, use B-tree indexes when you know that your typical query refers to the indexed column and retrieves a few rows. In these queries, it is faster to find the rows by looking at the index. However, using the book index analogy, if you plan to look at every single topic in a book, you might not want to look in the index for the topic and then look up the page. It might be faster to read through every chapter in the book. Similarly, if you are retrieving most of the rows in a table, it might not make sense to look up the index to find the table rows. Instead, you might want to read or scan the table.

B-tree indexes are most commonly used in a data warehouse to enforce unique keys. In many cases, it may not even be necessary to index these columns in a data warehouse, because the uniqueness was enforced as part of the preceding ETL processing, and because typical data warehouse queries may not work better with such indexes. B-tree indexes are more common in environments using third normal form schemas. In general, bitmap indexes should be more common than B-tree indexes in most data warehouse environments.

B-tree and bitmap indexes have different maximum column limitations. See *Oracle AI Database SQL Language Reference* for these limitations.


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 4.1.7 Using Index Compression

Bitmap indexes are always stored in a patented, compressed manner without the need of any user intervention. B-tree indexes, however, can be stored specifically in a compressed manner to enable huge space savings, storing more keys in each index block, which also leads to less I/O and better performance.

Key compression lets you compress a B-tree index, which reduces the storage overhead of repeated values. In the case of a nonunique index, all index columns can be stored in a compressed format, whereas in the case of a unique index, at least one index column has to be stored uncompressed. In addition to key compression, OLTP index compression may provide a higher degree of compression, but is more appropriate for OLTP applications than data warehousing environments.

Generally, keys in an index have two pieces, a grouping piece and a unique piece. If the key is not defined to have a unique piece, Oracle AI Database provides one in the form of a rowid appended to the grouping piece. Key compression is a method of breaking off the grouping piece and storing it so it can be shared by multiple unique pieces. The cardinality of the chosen columns to be compressed determines the compression ratio that can be achieved. So, for example, if a unique index that consists of five columns provides the uniqueness mostly by the last two columns, it is most optimal to choose the three leading columns to be stored compressed. If you choose to compress four columns, the repetitiveness will be almost gone, and the compression ratio will be worse.

Although key compression reduces the storage requirements of an index, it can increase the CPU time required to reconstruct the key column values during an index scan. It also incurs some additional storage overhead, because every prefix entry has an overhead of four bytes associated with it.

**See Also**

- *Oracle AI Database Administrator's Guide* for more information regarding key compression
- *Oracle AI Database Administrator's Guide* for more information regarding OLTP index compression

## 4.1.8 Choosing Between Local Indexes and Global Indexes

B-tree indexes on partitioned tables can be global or local. With Oracle8*i* and earlier releases, Oracle recommended that global indexes not be used in data warehouse environments because a partition DDL statement (for example, `ALTER TABLE ... DROP PARTITION`) would invalidate the entire index, and rebuilding the index is expensive. Global indexes can be maintained without Oracle marking them as unusable after DDL, which makes global indexes effective for data warehouse environments.

However, local indexes will be more common than global indexes. Global indexes should be used when there is a specific requirement which cannot be met by local indexes (for example, a unique index on a non-partitioning key, or a performance requirement).

Bitmap indexes on partitioned tables are always local.


Copyright © 2001, 2025, Oracle and/or its affiliates.

# 4.2 Using Integrity Constraints in a Data Warehouse

Integrity constraints provide a mechanism for ensuring that data conforms to guidelines specified by the database administrator.

The most common types of constraints include:

- `UNIQUE` constraints

  To ensure that a given column is unique
- `NOT NULL` constraints

  To ensure that no null values are allowed
- `FOREIGN KEY` constraints

  To ensure that two keys share a primary key to foreign key relationship

Constraints can be used for these purposes in a data warehouse:

- Data cleanliness

  Constraints verify that the data in the data warehouse conforms to a basic level of data consistency and correctness, preventing the introduction of dirty data.
- Query optimization

  The Oracle AI Database utilizes constraints when optimizing SQL queries. Although constraints can be useful in many aspects of query optimization, constraints are particularly important for query rewrite of materialized views.

Unlike data in many relational database environments, data in a data warehouse is typically added or modified under controlled circumstances during the extraction, transformation, and loading (ETL) process. Multiple users normally do not update the data warehouse directly, as they do in an OLTP system.

**See Also**

- Data Movement/ETL Overview

This section contains the following topics:

- Overview of Constraint States
- Typical Data Warehouse Integrity Constraints

## 4.2.1 Overview of Constraint States

To understand how best to use constraints in a data warehouse, you should first understand the basic purposes of constraints.

Some of these purposes are:

- Enforcement

  In order to use a constraint for enforcement, the constraint must be in the `ENABLE` state. An enabled constraint ensures that all data modifications upon a given table (or tables) satisfy the conditions of the constraints. Data modification operations which produce data that violates the constraint fail with a constraint violation error.


Copyright © 2001, 2025, Oracle and/or its affiliates.

- Validation

  To use a constraint for validation, the constraint must be in the `VALIDATE` state. If the constraint is validated, then all data that currently resides in the table satisfies the constraint.

  Note that validation is independent of enforcement. Although the typical constraint in an operational system is both enabled and validated, any constraint could be validated but not enabled or vice versa (enabled but not validated). These latter two cases are useful for data warehouses.

- Belief

  In some cases, you will know that the conditions for a given constraint are true, so you do not need to validate or enforce the constraint. However, you may wish for the constraint to be present anyway to improve query optimization and performance. When you use a constraint in this way, it is called a belief or `RELY` constraint, and the constraint must be in the `RELY` state. The `RELY` state provides you with a mechanism for telling Oracle that a given constraint is believed to be true.

  Note that the `RELY` state only affects constraints that have not been validated.

## 4.2.2 Typical Data Warehouse Integrity Constraints

This section assumes that you are familiar with the typical use of constraints. That is, constraints that are both enabled and validated. For data warehousing, many users have discovered that such constraints may be prohibitively costly to build and maintain. The topics discussed are:

- UNIQUE Constraints in a Data Warehouse
- FOREIGN KEY Constraints in a Data Warehouse
- RELY Constraints in a Data Warehouse
- NOT NULL Constraints in a Data Warehouse
- Integrity Constraints and Parallelism in a Data Warehouse
- Integrity Constraints and Partitioning in a Data Warehouse
- View Constraints in a Data Warehouse

### 4.2.2.1 UNIQUE Constraints in a Data Warehouse

A `UNIQUE` constraint is typically enforced using a `UNIQUE` index. However, in a data warehouse whose tables can be extremely large, creating a unique index can be costly both in processing time and in disk space.

Suppose that a data warehouse contains a table `sales`, which includes a column `sales_id`. `sales_id` uniquely identifies a single sales transaction, and the data warehouse administrator must ensure that this column is unique within the data warehouse.

One way to create the constraint is as follows:

```
ALTER TABLE sales ADD CONSTRAINT sales_uk
UNIQUE (prod_id, cust_id, promo_id, channel_id, time_id);
```

By default, this constraint is both enabled and validated. Oracle implicitly creates a unique index on `sales_id` to support this constraint. However, this index can be problematic in a data warehouse for three reasons:


Copyright © 2001, 2025, Oracle and/or its affiliates.

- The unique index can be very large, because the `sales` table can easily have millions or even billions of rows.
- The unique index is rarely used for query execution. Most data warehousing queries do not have predicates on unique keys, so creating this index will probably not improve performance.
- If `sales` is partitioned along a column other than `sales_id`, the unique index must be global. This can detrimentally affect all maintenance operations on the `sales` table.

A unique index is required for unique constraints to ensure that each individual row modified in the `sales` table satisfies the `UNIQUE` constraint.

For data warehousing tables, an alternative mechanism for unique constraints is illustrated in the following statement:

```
ALTER TABLE sales ADD CONSTRAINT sales_uk
UNIQUE (prod_id, cust_id, promo_id, channel_id, time_id) DISABLE VALIDATE;
```

This statement creates a unique constraint, but, because the constraint is disabled, a unique index is not required. This approach can be advantageous for many data warehousing environments because the constraint now ensures uniqueness without the cost of a unique index.

However, there are trade-offs for the data warehouse administrator to consider with `DISABLE VALIDATE` constraints. Because this constraint is disabled, no DML statements that modify the unique column are permitted against the `sales` table. You can use one of two strategies for modifying this table in the presence of a constraint:

- Use DDL to add data to this table (such as exchanging partitions). See the example in Refreshing Materialized Views.
- Before modifying this table, drop the constraint. Then, make all necessary data modifications. Finally, re-create the disabled constraint. Re-creating the constraint is more efficient than re-creating an enabled constraint. However, this approach does not guarantee that data added to the `sales` table while the constraint has been dropped is unique.

## 4.2.2.2 FOREIGN KEY Constraints in a Data Warehouse

In a star schema data warehouse, `FOREIGN KEY` constraints validate the relationship between the fact table and the dimension tables. A sample constraint might be:

```
ALTER TABLE sales ADD CONSTRAINT sales_time_fk
FOREIGN KEY (time_id) REFERENCES times (time_id)
ENABLE VALIDATE;
```

However, in some situations, you may choose to use a different state for the `FOREIGN KEY` constraints, in particular, the `ENABLE NOVALIDATE` state. A data warehouse administrator might use an `ENABLE NOVALIDATE` constraint when either:

- The tables contain data that currently disobeys the constraint, but the data warehouse administrator wishes to create a constraint for future enforcement.
- An enforced constraint is required immediately.

Suppose that the data warehouse loaded new data into the fact tables every day, but refreshed the dimension tables only on the weekend. During the week, the dimension tables and fact tables may in fact disobey the `FOREIGN KEY` constraints. Nevertheless, the data warehouse administrator might wish to maintain the enforcement of this constraint to prevent any changes that might affect the `FOREIGN KEY` constraint outside of the ETL process. Thus, you can create


Copyright © 2001, 2025, Oracle and/or its affiliates.

the FOREIGN KEY constraints every night, after performing the ETL process, as shown in the following:

```
ALTER TABLE sales ADD CONSTRAINT sales_time_fk
FOREIGN KEY (time_id) REFERENCES times (time_id)
ENABLE NOVALIDATE;
```

ENABLE NOVALIDATE can quickly create an enforced constraint, even when the constraint is believed to be true. Suppose that the ETL process verifies that a FOREIGN KEY constraint is true. Rather than have the database re-verify this FOREIGN KEY constraint, which would require time and database resources, the data warehouse administrator could instead create a FOREIGN KEY constraint using ENABLE NOVALIDATE.

### 4.2.2.3 RELY Constraints in a Data Warehouse

The ETL process commonly verifies that certain constraints are true. For example, it can validate all of the foreign keys in the data coming into the fact table. This means that you can trust it to provide clean data, instead of implementing constraints in the data warehouse. You create a RELY constraint as follows:

```
ALTER TABLE sales ADD CONSTRAINT sales_time_fk
FOREIGN KEY (time_id) REFERENCES times (time_id)
RELY DISABLE NOVALIDATE;
```

This statement assumes that the primary key is in the RELY state. RELY constraints, even though they are not used for data validation, can:

- Enable more sophisticated query rewrites for materialized views. See Basic Query Rewrite for Materialized Views for further details.
- Enable other data warehousing tools to retrieve information regarding constraints directly from the Oracle data dictionary.

Creating a RELY constraint is inexpensive and does not impose any overhead during DML or load. Because the constraint is not being validated, no data processing is necessary to create it.

### 4.2.2.4 NOT NULL Constraints in a Data Warehouse

When using query rewrite, you should consider whether NOT NULL constraints are required. The primary situation where you will need to use them is for join back query rewrite.

**See Also**

- Advanced Query Rewrite for Materialized Views for further information regarding NOT NULL constraints when using query rewrite

### 4.2.2.5 Integrity Constraints and Parallelism in a Data Warehouse

All constraints can be validated in parallel. When validating constraints on very large tables, parallelism is often necessary to meet performance goals. The degree of parallelism for a given constraint operation is determined by the default degree of parallelism of the underlying table.


Copyright © 2001, 2025, Oracle and/or its affiliates.

### 4.2.2.6 Integrity Constraints and Partitioning in a Data Warehouse

You can create and maintain constraints before you partition the data. Later chapters discuss the significance of partitioning for data warehousing. Partitioning can improve constraint management just as it does to management of many other operations. For example, Refreshing Materialized Views provides a scenario creating `UNIQUE` and `FOREIGN KEY` constraints on a separate staging table, and these constraints are maintained during the `EXCHANGE PARTITION` statement.

For external tables, you can only define `RELY` constraints in `DISABLE` mode. This is applicable to primary key, unique key, and foreign key constraints.

### 4.2.2.7 View Constraints in a Data Warehouse

You can create constraints on views. The only type of constraint supported on a view is a `RELY` constraint.

This type of constraint is useful when queries typically access views instead of base tables, and the database administrator thus needs to define the data relationships between views rather than tables.

**See Also**

- Basic Materialized Views
- Basic Query Rewrite for Materialized Views

## 4.3 About Parallel Execution in Data Warehouses

Databases today, irrespective of whether they are data warehouses, operational data stores, or OLTP systems, contain a large amount of information. However, finding and presenting the right information in a timely fashion can be a challenge because of the vast quantity of data involved.

Parallel execution is the capability that addresses this challenge. Using parallel execution (also called parallelism), terabytes of data can be processed in minutes, not hours or days, simply by using multiple processes to accomplish a single task. This dramatically reduces response time for data-intensive operations on large databases typically associated with decision support systems (DSS) and data warehouses. You can also implement parallel execution on OLTP system for batch processing or schema maintenance operations such as index creation. Parallelism is the idea of breaking down a task so that, instead of one process doing all of the work in a query, many processes do part of the work at the same time. An example of this is when four processes combine to calculate the total sales for a year, each process handles one quarter of the year instead of a single processing handling all four quarters by itself. The improvement in performance can be quite significant.

Parallel execution improves processing for:

- Queries requiring large table scans, joins, or partitioned index scans
- Creations of large indexes
- Creation of large tables (including materialized views)
- Bulk inserts, updates, merges, and deletes


Copyright © 2001, 2025, Oracle and/or its affiliates.

You can also use parallel execution to access object types within an Oracle database. For example, you can use parallel execution to access large objects (LOBs).

Large data warehouses should always use parallel execution to achieve good performance. Specific operations in OLTP applications, such as batch operations, can also significantly benefit from parallel execution.

This section contains the following topics:

- Why Use Parallel Execution?
- Automatic Degree of Parallelism and Statement Queuing
- About In-Memory Parallel Execution in Data Warehouses

## 4.3.1 Why Use Parallel Execution?

Imagine that your task is to count the number of cars in a street. There are two ways to do this. One, you can go through the street by yourself and count the number of cars or you can enlist a friend and then the two of you can start on opposite ends of the street, count cars until you meet each other and add the results of both counts to complete the task.

Assuming your friend counts equally fast as you do, you expect to complete the task of counting all cars in a street in roughly half the time compared to when you perform the job all by yourself. If this is the case, then your operations scales linearly. That is, twice the number of resources halves the total processing time.

A database is not very different from the counting cars example. If you allocate twice the number of resources and achieve a processing time that is half of what it was with the original amount of resources, then the operation scales linearly. Scaling linearly is the ultimate goal of parallel processing, both in counting cars as well as in delivering answers from a database query.

**See Also**

- *Oracle AI Database VLDB and Partitioning Guide* for more information about using parallel execution

This following topics provide guidance on the scenarios in which parallel execution is useful:

- When to Implement Parallel Execution
- When Not to Implement Parallel Execution

### 4.3.1.1 When to Implement Parallel Execution

Parallel execution benefits systems with all of the following characteristics:

- Symmetric multiprocessors (SMPs), clusters, or massively parallel systems
- Sufficient I/O bandwidth
- Underutilized or intermittently used CPUs (for example, systems where CPU usage is typically less than 30%)
- Sufficient memory to support additional memory-intensive processes, such as sorts, hashing, and I/O buffers


Copyright © 2001, 2025, Oracle and/or its affiliates.

If your system lacks any of these characteristics, parallel execution might not significantly improve performance. In fact, parallel execution may reduce system performance on overutilized systems or systems with small I/O bandwidth.

The benefits of parallel execution can be seen in DSS and data warehousing environments. OLTP systems can also benefit from parallel execution during batch processing and during schema maintenance operations such as creation of indexes. The average simple DML or `SELECT` statements, accessing or manipulating small sets of records or even single records, that characterize OLTP applications would not see any benefit from being executed in parallel.

### 4.3.1.2 When Not to Implement Parallel Execution

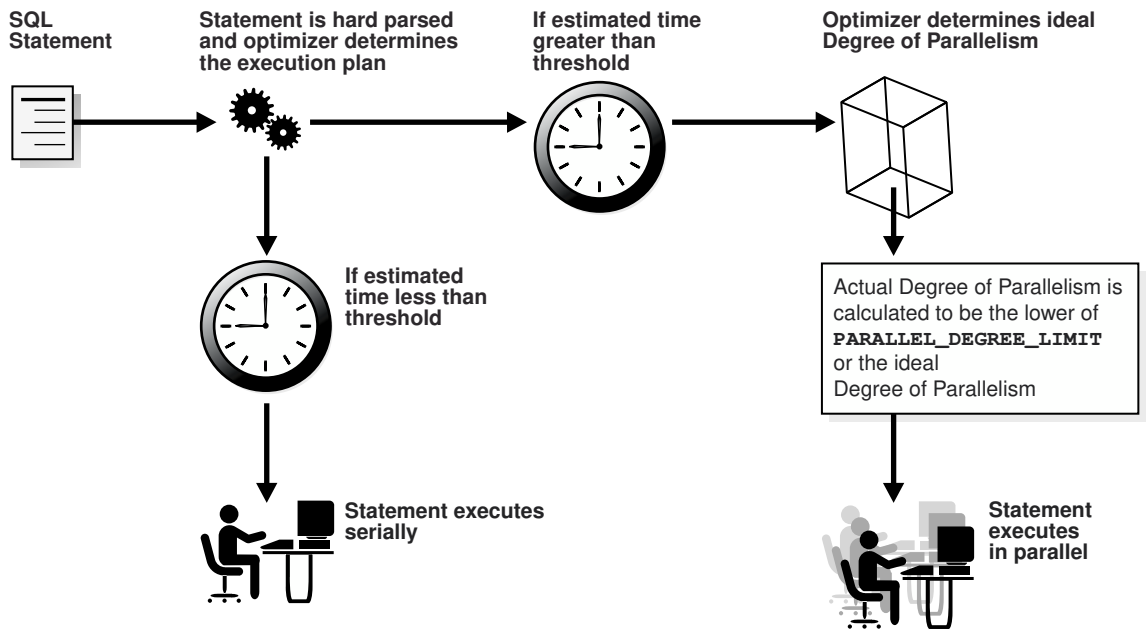
Parallel execution is not normally useful for:

- Environments in which the typical query or transaction is very short (a few seconds or less). This includes most online transaction systems. Parallel execution is not useful in these environments because there is a cost associated with coordinating the parallel execution servers; for short transactions, the cost of this coordination may outweigh the benefits of parallelism.
- Environments in which the CPU, memory, or I/O resources are heavily utilized, even with parallel execution. Parallel execution is designed to exploit additional available hardware resources; if no such resources are available, then parallel execution does not yield any benefits and indeed may be detrimental to performance.

## 4.3.2 Automatic Degree of Parallelism and Statement Queuing

As the name implies, automatic degree of parallelism is where Oracle AI Database determines the degree of parallelism (DOP) with which to run a statement (DML, DDL, and queries) based on the execution cost - the resource consumption of CPU, I/O, and memory - as determined by the Optimizer. That means that the database parses a query, calculates the cost and then determines a DOP to run with. The cheapest plan may be to run serially, which is also an option. Figure 4-2 illustrates this decision making process.

**Figure 4-2 Optimizer Calculation: Serial or Parallel?**





Copyright © 2001, 2025, Oracle and/or its affiliates.

Should you choose to use automatic DOP, you may potentially see many more statements running in parallel, especially if the threshold is relatively low, where low is relative to the system and not an absolute quantifier.

Because of this expected behavior of more statements running in parallel with automatic DOP, it becomes more important to manage the utilization of the parallel processes available. That means that the system must be intelligent about when to run a statement and verify whether the requested numbers of parallel processes are available. The requested number of processes in this is the DOP for that statement.

The answer to this workload management question is parallel statement queuing with the Database Resource Manager. Parallel statement queuing runs a statement when its requested DOP is available. For example, when a statement requests a DOP of 64, it will not run if there are only 32 processes currently free to assist this customer, so the statement will be placed into a queue.

With Database Resource Manager, you can classify statements into workloads through consumer groups. Each consumer group can then be given the appropriate priority and the appropriate levels of parallel processes. Each consumer group also has its own queue to queue parallel statements based on the system load.

**See Also**

- *Oracle AI Database VLDB and Partitioning Guide* for more information about using automatic DOP with parallel execution
- *Oracle AI Database Administrator's Guide* for more information about using the Database Resource Manager

## 4.3.3 About In-Memory Parallel Execution in Data Warehouses

Traditionally, parallel processing by-passed the database buffer cache for most operations, reading data directly from disk (through direct path I/O) into the parallel execution server's private working space. Only objects smaller than about 2% of `DB_CACHE_SIZE` would be cached in the database buffer cache of an instance, and most objects accessed in parallel are larger than this limit. This behavior meant that parallel processing rarely took advantage of the available memory other than for its private processing. However, over the last decade, hardware systems have evolved quite dramatically; the memory capacity on a typical database server is now in the double or triple digit gigabyte range. This, together with Oracle's compression technologies and the capability of Oracle AI Database to exploit the aggregated database buffer cache of an Oracle Real Application Clusters environment, enables caching of objects in the terabyte range.

In-memory parallel execution takes advantage of this large aggregated database buffer cache. Having parallel execution servers accessing objects using the buffer cache enables full parallel in-memory processing of large volumes of data, leading to performance improvements in orders of magnitudes.

With in-memory parallel execution, when a SQL statement is issued in parallel, a check is conducted to determine if the objects accessed by the statement should be cached in the aggregated buffer cache of the system. In this context, an object can either be a table, index, or, in the case of partitioned objects, one or multiple partitions.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**See Also**

- *Oracle AI Database VLDB and Partitioning Guide* for more information about using in-memory parallel execution

# 4.4 About Optimizing Storage Requirements in Data Warehouses

You can reduce your storage requirements by compressing data, which is achieved by eliminating duplicate values in a database block. "Using Data Compression to Improve Storage in Data Warehouses" describes how you can use compress data.

Database objects that can be compressed include tables and materialized views. For partitioned tables, you can compress some or all partitions. Compression attributes can be declared for a tablespace, a table, or a partition of a table. If declared at the tablespace level, then all tables created in that tablespace are compressed by default. You can alter the compression attribute for a table (or a partition or tablespace), and the change applies only to new data going into that table. As a result, a single table or partition may contain some compressed blocks and some regular blocks. This guarantees that data size will not increase as a result of compression. In cases where compression could increase the size of a block, it is not applied to that block.

## 4.4.1 Using Data Compression to Improve Storage in Data Warehouses

You can compress several partitions or a complete partitioned heap-organized table. You do this either by defining a complete partitioned table as being compressed, or by defining it on a per-partition level. Partitions without a specific declaration inherit the attribute from the table definition or, if nothing is specified on the table level, from the tablespace definition.

The decision about whether or not a partition should be compressed is based on the same rules as a nonpartitioned table. Because of the ability of range and composite partitioning to separate data logically into distinct partitions, a partitioned table is an ideal candidate for compressing parts of the data (partitions) that are mainly read-only. It is, for example, beneficial in all rolling window operations as a kind of intermediate stage before aging out old data. With data compression, you can keep more old data online, minimizing the burden of additional storage use.

You can also change any existing uncompressed table partition later, add new compressed and uncompressed partitions, or change the compression attribute as part of any partition maintenance operation that requires data movement, such as `MERGE PARTITION`, `SPLIT PARTITION`, or `MOVE PARTITION`. The partitions can contain data, or they can be empty.

The access and maintenance of a partially or fully compressed partitioned table are the same as for a fully uncompressed partitioned table. All rules that apply to fully uncompressed partitioned tables are also valid for partially or fully compressed partitioned tables.

**To use data compression:**

The following example creates a range-partitioned table with one compressed partition `costs_old`. The compression attribute for the table and all other partitions is inherited from the tablespace level.

```
CREATE TABLE costs_demo (
   prod_id     NUMBER(6),    time_id     DATE,
   unit_cost   NUMBER(10,2), unit_price  NUMBER(10,2))
PARTITION BY RANGE (time_id)
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
(PARTITION costs_old
   VALUES LESS THAN (TO_DATE('01-JAN-2003', 'DD-MON-YYYY')) COMPRESS,
 PARTITION costs_q1_2003
   VALUES LESS THAN (TO_DATE('01-APR-2003', 'DD-MON-YYYY')),
 PARTITION costs_q2_2003
   VALUES LESS THAN (TO_DATE('01-JUN-2003', 'DD-MON-YYYY')),
 PARTITION costs_recent VALUES LESS THAN (MAXVALUE));
```

# 4.5 Optimizing Star Queries and 3NF Schemas

Oracle data warehouses can work well with star schemas and third normal form schemas. This section presents important techniques for optimizing performance in both types of schema. For conceptual background on star and 3NF schemas, see "About Third Normal Form Schemas". and "About Star Schemas".

You should consider the following when using star queries:

- Optimizing Star Queries
- Using Star Transformation
- Optimizing Third Normal Form Schemas
- Optimizing Star Queries Using VECTOR GROUP BY Aggregation

## 4.5.1 Optimizing Star Queries

A star query is a join between a fact table and a number of dimension tables. Each dimension table is joined to the fact table using a primary key to foreign key join, but the dimension tables are not joined to each other. The optimizer recognizes star queries and generates efficient execution plans for them. "Tuning Star Queries" describes how to improve the performance of star queries.

### 4.5.1.1 Tuning Star Queries

To get the best possible performance for star queries, it is important to follow some basic guidelines:

- A bitmap index should be built on each of the foreign key columns of the fact table or tables.
- The initialization parameter `STAR_TRANSFORMATION_ENABLED` should be set to `TRUE`. This enables an important optimizer feature for star-queries. It is set to `FALSE` by default for backward-compatibility.

When a data warehouse satisfies these conditions, the majority of the star queries running in the data warehouse uses a query execution strategy known as the star transformation. The star transformation provides very efficient query performance for star queries.

## 4.5.2 Using Star Transformation

The star transformation is a powerful optimization technique that relies upon implicitly rewriting (or transforming) the SQL of the original star query. The end user never needs to know any of the details about the star transformation. Oracle AI Database includes a query optimizer to automatically chooses the star transformation where appropriate.

The star transformation is a query transformation aimed at executing star queries efficiently. Oracle AI Database processes a star query using two basic phases. The first phase retrieves exactly the necessary rows from the fact table (the result set). Because this retrieval utilizes


Copyright © 2001, 2025, Oracle and/or its affiliates.

bitmap indexes, it is very efficient. The second phase joins this result set to the dimension tables. An example of an end user query is: "What were the sales and profits for the grocery department of stores in the west and southwest sales districts over the last three quarters?" This is a simple star query.

This section contains the following topics:


- Star Transformation with a Bitmap Index
- Execution Plan for a Star Transformation with a Bitmap Index
- Star Transformation with a Bitmap Join Index
- Execution Plan for a Star Transformation with a Bitmap Join Index
- How Oracle Chooses to Use Star Transformation
- Star Transformation Restrictions


## 4.5.2.1 Star Transformation with a Bitmap Index

A prerequisite of the star transformation is that there be a single-column bitmap index on every join column of the fact table. These join columns include all foreign key columns.

For example, the `sales` table of the `sh` sample schema has bitmap indexes on the `time_id`, `channel_id`, `cust_id`, `prod_id`, and `promo_id` columns.

Consider the following star query:

```
SELECT ch.channel_class, c.cust_city, t.calendar_quarter_desc,
   SUM(s.amount_sold) sales_amount
FROM sales s, times t, customers c, channels ch
WHERE s.time_id = t.time_id
AND   s.cust_id = c.cust_id
AND   s.channel_id = ch.channel_id
AND   c.cust_state_province = 'CA'
AND   ch.channel_desc in ('Internet','Catalog')
AND   t.calendar_quarter_desc IN ('1999-Q1','1999-Q2')
GROUP BY ch.channel_class, c.cust_city, t.calendar_quarter_desc;
```

This query is processed in two phases. In the first phase, Oracle AI Database uses the bitmap indexes on the foreign key columns of the fact table to identify and retrieve only the necessary rows from the fact table. That is,Oracle AI Database retrieves the result set from the fact table using essentially the following query:

```
SELECT ... FROM sales
WHERE time_id IN
  (SELECT time_id FROM times
   WHERE calendar_quarter_desc IN('1999-Q1','1999-Q2'))
   AND cust_id IN
  (SELECT cust_id FROM customers WHERE cust_state_province='CA')
   AND channel_id IN
  (SELECT channel_id FROM channels WHERE channel_desc IN('Internet','Catalog'));
```

This is the transformation step of the algorithm, because the original star query has been transformed into this subquery representation. This method of accessing the fact table leverages the strengths of bitmap indexes. Intuitively, bitmap indexes provide a set-based processing scheme within a relational database. Oracle has implemented very fast methods for doing set operations such as `AND` (an intersection in standard set-based terminology), `OR` (a set-based union), `MINUS`, and `COUNT`.


Copyright © 2001, 2025, Oracle and/or its affiliates.

In this star query, a bitmap index on `time_id` is used to identify the set of all rows in the fact table corresponding to `sales` in `1999-Q1`. This set is represented as a bitmap (a string of 1's and 0's that indicates which rows of the fact table are members of the set).

A similar bitmap is retrieved for the fact table rows corresponding to the sale from `1999-Q2`. The bitmap `OR` operation is used to combine this set of `Q1` sales with the set of `Q2` sales.

Additional set operations will be done for the `customer` dimension and the `product` dimension. At this point in the star query processing, there are three bitmaps. Each bitmap corresponds to a separate dimension table, and each bitmap represents the set of rows of the fact table that satisfy that individual dimension's constraints.

These three bitmaps are combined into a single bitmap using the bitmap `AND` operation. This final bitmap represents the set of rows in the fact table that satisfy all of the constraints on the dimension table. This is the result set, the exact set of rows from the fact table needed to evaluate the query. Note that none of the actual data in the fact table has been accessed. All of these operations rely solely on the bitmap indexes and the dimension tables. Because of the bitmap indexes' compressed data representations, the bitmap set-based operations are extremely efficient.

Once the result set is identified, the bitmap is used to access the actual data from the sales table. Only those rows that are required for the end user's query are retrieved from the fact table. At this point, Oracle AI Database has effectively joined all of the dimension tables to the fact table using bitmap indexes. This technique provides excellent performance because Oracle AI Database is joining all of the dimension tables to the fact table with one logical join operation, rather than joining each dimension table to the fact table independently.

The second phase of this query is to join these rows from the fact table (the result set) to the dimension tables. Oracle uses the most efficient method for accessing and joining the dimension tables. Many dimension are very small, and table scans are typically the most efficient access method for these dimension tables. For large dimension tables, table scans may not be the most efficient access method. In the previous example, a bitmap index on `product.department` can be used to quickly identify all of those products in the grocery department. Oracle AI Database includes an optimizer that automatically determines which access method is most appropriate for a given dimension table, based upon the optimizer's knowledge about the sizes and data distributions of each dimension table.

The specific join method (as well as indexing method) for each dimension table will likewise be intelligently determined by the optimizer. A hash join is often the most efficient algorithm for joining the dimension tables. The final answer is returned to the user once all of the dimension tables have been joined. The query technique of retrieving only the matching rows from one table and then joining to another table is commonly known as a semijoin.

### 4.5.2.2 Execution Plan for a Star Transformation with a Bitmap Index

The following typical execution plan might result from "Star Transformation with a Bitmap Index":

```
SELECT STATEMENT
 SORT GROUP BY
  HASH JOIN
   TABLE ACCESS FULL                          CHANNELS
   HASH JOIN
    TABLE ACCESS FULL                         CUSTOMERS
    HASH JOIN
     TABLE ACCESS FULL                        TIMES
     PARTITION RANGE ITERATOR
      TABLE ACCESS BY LOCAL INDEX ROWID       SALES
       BITMAP CONVERSION TO ROWIDS
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
BITMAP AND
 BITMAP MERGE
  BITMAP KEY ITERATION
   BUFFER SORT
    TABLE ACCESS FULL                 CUSTOMERS
   BITMAP INDEX RANGE SCAN            SALES_CUST_BIX
 BITMAP MERGE
  BITMAP KEY ITERATION
   BUFFER SORT
    TABLE ACCESS FULL                 CHANNELS
   BITMAP INDEX RANGE SCAN            SALES_CHANNEL_BIX
 BITMAP MERGE
  BITMAP KEY ITERATION
   BUFFER SORT
    TABLE ACCESS FULL                 TIMES
   BITMAP INDEX RANGE SCAN            SALES_TIME_BIX
```

In this plan, the fact table is accessed through a bitmap access path based on a bitmap `AND`, of three merged bitmaps. The three bitmaps are generated by the `BITMAP MERGE` row source being fed bitmaps from row source trees underneath it. Each such row source tree consists of a `BITMAP KEY ITERATION` row source which fetches values from the subquery row source tree, which in this example is a full table access. For each such value, the `BITMAP KEY ITERATION` row source retrieves the bitmap from the bitmap index. After the relevant fact table rows have been retrieved using this access path, they are joined with the dimension tables and temporary tables to produce the answer to the query.

## 4.5.2.3 Star Transformation with a Bitmap Join Index

In addition to bitmap indexes, you can use a bitmap join index during star transformations. Assume you have the following additional index structure:

```
CREATE BITMAP INDEX sales_c_state_bjix
ON sales(customers.cust_state_province)
FROM sales, customers
WHERE sales.cust_id = customers.cust_id
LOCAL NOLOGGING COMPUTE STATISTICS;
```

The processing of the same star query using the bitmap join index is similar to the previous example. The only difference is that Oracle utilizes the join index, instead of a single-table bitmap index, to access the customer data in the first phase of the star query.

## 4.5.2.4 Execution Plan for a Star Transformation with a Bitmap Join Index

The following typical execution plan might result from "Execution Plan for a Star Transformation with a Bitmap Join Index":

```
SELECT STATEMENT
 SORT GROUP BY
  HASH JOIN
   TABLE ACCESS FULL                  CHANNELS
   HASH JOIN
    TABLE ACCESS FULL                 CUSTOMERS
    HASH JOIN
     TABLE ACCESS FULL                TIMES
     PARTITION RANGE ALL
      TABLE ACCESS BY LOCAL INDEX ROWID SALES
       BITMAP CONVERSION TO ROWIDS
        BITMAP AND
         BITMAP INDEX SINGLE VALUE    SALES_C_STATE_BJIX
         BITMAP MERGE
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
  BITMAP KEY ITERATION
   BUFFER SORT
    TABLE ACCESS FULL                CHANNELS
   BITMAP INDEX RANGE SCAN           SALES_CHANNEL_BIX
 BITMAP MERGE
  BITMAP KEY ITERATION
   BUFFER SORT
    TABLE ACCESS FULL                TIMES
   BITMAP INDEX RANGE SCAN           SALES_TIME_BIX
```

The difference between this plan as compared to the previous one is that the inner part of the bitmap index scan for the `customer` dimension has no subselect. This is because the join predicate information on `customer.cust_state_province` can be satisfied with the bitmap join index `sales_c_state_bjix`.

## 4.5.2.5 How Oracle Chooses to Use Star Transformation

The optimizer generates and saves the best plan it can produce without the transformation. If the transformation is enabled, the optimizer then tries to apply it to the query and, if applicable, generates the best plan using the transformed query. Based on a comparison of the cost estimates between the best plans for the two versions of the query, the optimizer then decides whether to use the best plan for the transformed or untransformed version.

If the query requires accessing a large percentage of the rows in the fact table, it might be better to use a full table scan and not use the transformations. However, if the constraining predicates on the dimension tables are sufficiently selective that only a small portion of the fact table must be retrieved, the plan based on the transformation will probably be superior.

Note that the optimizer generates a subquery for a dimension table only if it decides that it is reasonable to do so based on a number of criteria. There is no guarantee that subqueries will be generated for all dimension tables. The optimizer may also decide, based on the properties of the tables and the query, that the transformation does not merit being applied to a particular query. In this case, the best regular plan will be used.

## 4.5.2.6 Star Transformation Restrictions

Star transformation is not supported for tables with any of the following characteristics:

- Queries with a table hint that is incompatible with a bitmap access path
- Tables with too few bitmap indexes. There must be a bitmap index on a fact table column for the optimizer to generate a subquery for it.
- Remote fact tables. However, remote dimension tables are allowed in the subqueries that are generated.
- Anti-joined tables
- Tables that are already used as a dimension table in a subquery
- Tables that are really unmerged views, which are not view partitions
- Tables where the fact table is an unmerged view
- Tables where a partitioned view is used as a fact table

The star transformation may not be chosen by the optimizer for the following cases:

- Tables that have a good single-table access path
- Tables that are too small for the transformation to be worthwhile


Copyright © 2001, 2025, Oracle and/or its affiliates.

In addition, temporary tables will not be used by star transformation under the following conditions:

- The database is in read-only mode
- The star query is part of a transaction that is in serializable mode

## 4.5.3 Optimizing Third Normal Form Schemas

Optimizing a third normal form (3NF) schema requires the following:

- Power

  Power means that the hardware configuration must be balanced. Many data warehouse operations are based upon large table scans and other IO-intensive operations, which perform vast quantities of random IOs. In order to achieve optimal performance the hardware configuration must be sized end to end to sustain this level of throughput. This type of hardware configuration is called a balanced system. In a balanced system, all components - from the CPU to the disks - are orchestrated to work together to guarantee the maximum possible IO throughput.

- Partitioning

  The larger tables should be partitioned using composite partitioning (range-hash or list-hash). There are three reasons for this:

  – Easier manageability of terabytes of data
  – Faster accessibility to the necessary data
  – Efficient and performant table joins

  See 3NF Schemas: Partitioning.

- Parallel Execution

  Parallel Execution enables a database task to be parallelized or divided into smaller units of work, thus allowing multiple processes to work concurrently. By using parallelism, a terabyte of data can be scanned and processed in minutes or less, not hours or days.

  See 3NF Schemas: Parallel Query Execution.

### 4.5.3.1 3NF Schemas: Partitioning

Partitioning allows a table, index or index-organized table to be subdivided into smaller pieces. Each piece of the database object is called a partition. Each partition has its own name, and may optionally have its own storage characteristics. From the perspective of a database administrator, a partitioned object has multiple pieces that can be managed either collectively or individually.

This gives the administrator considerable flexibility in managing partitioned objects. However, from the perspective of the application, a partitioned table is identical to a non-partitioned table; no modifications are necessary when accessing a partitioned table using SQL DML commands. Partitioning can provide tremendous benefits to a wide variety of applications by improving manageability, availability, and performance.

#### 4.5.3.1.1 Partitioning for Manageability

Range partitioning will help improve the manageability and availability of large volumes of data. Consider the case where two year's worth of sales data or 100 terabytes (TB) is stored in a table. At the end of each day a new batch of data needs to be to loaded into the table and the oldest days worth of data needs to be removed. If the Sales table is ranged partitioned by day


Copyright © 2001, 2025, Oracle and/or its affiliates.

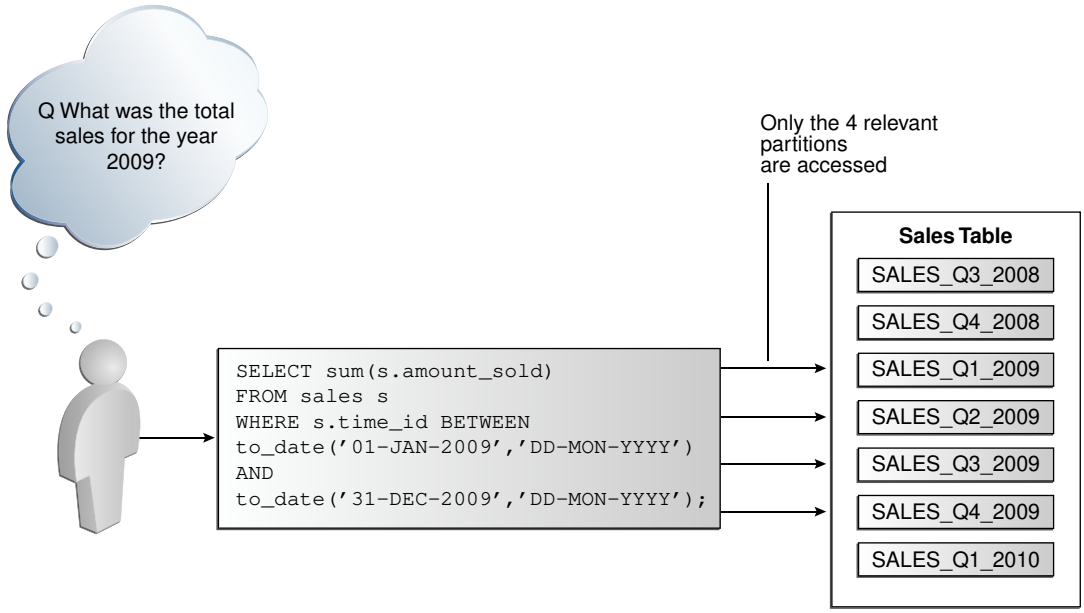
the new data can be loaded using a partition exchange load. This is a sub-second operation and should have little or no impact to end user queries. In order to remove the oldest day of data simply issue the following command:

```
SH@DBM1 > ALTER TABLE SALES DROP PARTITION Sales_Q4_2009;
```

### 4.5.3.1.2 Partitioning for Easier Data Access

Range partitioning will also help ensure only the necessary data to answer a query will be scanned. Let's assume that the business users predominately access the sales data on a weekly basis, to check total sales per week. Then range partitioning this table by day will ensure that the data is accessed in the most efficient manner, as only four partitions need to be scanned to answer the business users query instead of the entire table. The ability to avoid scanning irrelevant partitions is known as partition pruning.

**Figure 4-3 Partition Pruning**




You can define partitions for external tables. External tables are tables that do not reside in the database and can be in any format for which an access driver is provided. The files for partitioned external tables can be stored in a file system, in Apache Hive storage, or in a Hadoop Distributed File System (HDFS).

Partitioning for external tables improves query performance and enables easier data maintenance. It also enables external tables to take advantage of performance optimizations, such as partition pruning and partition-wise joins, that are available to partitioned tables stored in the database. Most partitioning techniques supported for tables in the database, except hash partitioning, are supported for partitioned external tables. However, Oracle AI Database cannot guarantee that the external storage files for partitions contain data that satisfies the partitioning conditions.

Note that as of Oracle AI Database 26ai, you can use external table partitioning with folder names as part of the file paths. External table columns also can return the filename of the source file for each row.


Copyright © 2001, 2025, Oracle and/or its affiliates.

> **See Also**
>
> *Oracle AI Database Administrator's Guide* for detailed information about partitioned external tables
>
> *Using SQL*Loader for External Tables with Partition Values in File Paths* in the *Oracle AI Database Utilities Guide*.

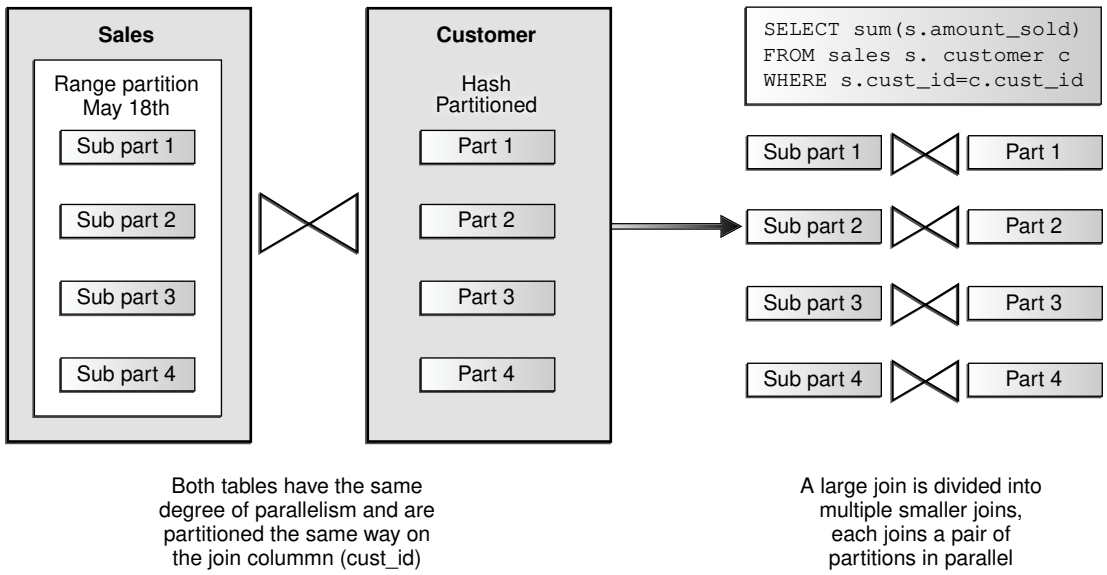
### 4.5.3.1.3 Partitioning for Join Performance

Sub-partitioning by hash is used predominately for performance reasons. Oracle uses a linear hashing algorithm to create sub-partitions. In order to ensure that the data gets evenly distributed among the hash partitions, it is highly recommended that the number of hash partitions is a power of 2 (for example, 2, 4, 8, and so on). Each hash partition should be at least 16MB in size. Any smaller and they will not have efficient scan rates with parallel query.

One of the main performance benefits of hash partitioning is partition-wise joins. Partition-wise joins reduce query response time by minimizing the amount of data exchanged among parallel execution servers when joins execute in parallel. This significantly reduces response time and improves both CPU and memory resource usage. In a clustered data warehouse, this significantly reduces response times by limiting the data traffic over the interconnect (IPC), which is the key to achieving good scalability for massive join operations. Partition-wise joins can be full or partial, depending on the partitioning scheme of the tables to be joined.

A full partition-wise join divides a join between two large tables into multiple smaller joins. Each smaller join performs a joins on a pair of partitions, one for each of the tables being joined. For the optimizer to choose the full partition-wise join method, both tables must be equi-partitioned on their join keys. That is, they have to be partitioned on the same column with the same partitioning method. Parallel execution of a full partition-wise join is similar to its serial execution, except that instead of joining one partition pair at a time, multiple partition pairs are joined in parallel by multiple parallel query servers. The number of partitions joined in parallel is determined by the Degree of Parallelism (DOP).

**Figure 4-4 Full Partition-Wise Join**





Copyright © 2001, 2025, Oracle and/or its affiliates.

Figure 4-4 illustrates the parallel execution of a full partition-wise join between two tables, `Sales` and `Customers`. Both tables have the same degree of parallelism and the same number of partitions. They are range partitioned on a date field and sub-partitioned by hash on the `cust_id` field. As illustrated in the picture, each partition pair is read from the database and joined directly. There is no data redistribution necessary, thus minimizing IPC communication, especially across nodes. Figure 4-5 below shows the execution plan you would see for this join.

To ensure that you get optimal performance when executing a partition-wise join in parallel, the number of partitions in each of the tables should be larger than the degree of parallelism used for the join. If there are more partitions than parallel servers, each parallel server will be given one pair of partitions to join, when the parallel server completes that join, it will requests another pair of partitions to join. This process repeats until all pairs have been processed. This method enables the load to be balanced dynamically (for example, 128 partitions with a degree of parallelism of 32).

What happens if only one of the tables you are joining is partitioned? In this case the optimizer could pick a partial partition-wise join. Unlike full partition-wise joins, partial partition-wise joins can be applied if only one table is partitioned on the join key. Hence, partial partition-wise joins are more common than full partition-wise joins. To execute a partial partition-wise join, Oracle dynamically repartitions the other table based on the partitioning strategy of the partitioned table. Once the other table is repartitioned, the execution is similar to a full partition-wise join. The redistribution operation involves exchanging rows between parallel execution servers. This operation leads to interconnect traffic in Oracle RAC environments, because data needs to be repartitioned across node boundaries.

**Figure 4-5 Partial Partition-Wise Join**

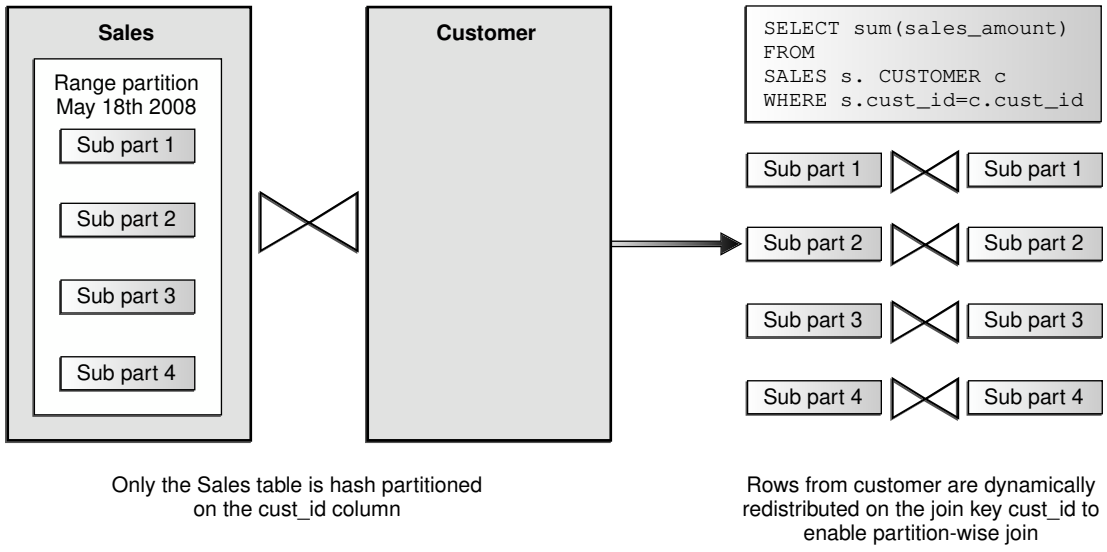



Figure 4-5 illustrates a partial partition-wise join. It uses the same example as in Figure 4-4, except that the customer table is not partitioned. Before the join operation is executed, the rows from the customers table are dynamically redistributed on the join key.

## 4.5.3.2 3NF Schemas: Parallel Query Execution

3NF schemas can leverage parallelism in multiple ways, but here the focus is on one facet of parallelism that is specially significant to 3NF: SQL parallel execution for large queries. SQL parallel execution in the Oracle AI Database is based on the principles of a coordinator (often


Copyright © 2001, 2025, Oracle and/or its affiliates.

called the Query Coordinator or QC) and parallel servers. The QC is the session that initiates the parallel SQL statement and the parallel servers are the individual sessions that perform work in parallel. The QC distributes the work to the parallel servers and may have to perform a minimal mostly logistical - portion of the work that cannot be executed in parallel. For example a parallel query with a `SUM()` operation requires adding the individual sub-totals calculated by each parallel server.

The QC is easily identified in the parallel execution in Figure 4-5 as PX COORDINATOR. The process acting as the QC of a parallel SQL operation is the actual user session process itself. The parallel servers are taken from a pool of globally available parallel server processes and assigned to a given operation. The parallel servers do all the work shown in a parallel plan BELOW the QC.

By default, the Oracle AI Database is configured to support parallel execution out-of-the-box and is controlled by two initialization parameters `parallel_max_servers` and `parallel_min_servers`. While parallel execution provides a very powerful and scalable framework to speed up SQL operations, you should not forget to use some common sense rules; while parallel execution might buy you an additional incremental performance boost, it requires more resources and might also have side effects on other users or operations on the same system. Small tables/indexes (up to thousands of records; up to 10s of data blocks) should never be enabled for parallel execution. Operations that only hit small tables will not benefit much from executing in parallel, but they will use parallel servers that you will want to be available for operations accessing large tables. Remember also that once an operation starts at a certain degree of parallelism (DOP), there is no way to reduce its DOP during the execution.

The general rules of thumb for determining the appropriate DOP for an object are:

- Objects smaller than 200 MB should not use any parallelism
- Objects between 200 MB and 5GB should use a DOP of 4
- Objects beyond 5GB use a DOP of 32

Needless to say the optimal settings may vary on your system - either in size range or DOP - and highly depend on your target workload, the business requirements, and your hardware configuration. Whether or Not to Use Cross Instance Parallel Execution in Oracle RAC describes parallel execution in Oracle RAC environments.

#### 4.5.3.2.1 Whether or Not to Use Cross Instance Parallel Execution in Oracle RAC

By default, Oracle AI Database enables inter-node parallel execution (parallel execution of a single statement involving more than one node). As mentioned earlier, the interconnect in an Oracle RAC environment must be sized appropriately as inter-node parallel execution may result in a lot of interconnect traffic. If you are using a relatively weak interconnect in comparison to the I/O bandwidth from the server to the storage subsystem, you may be better off restricting parallel execution to a single node or to a limited number of nodes. Inter-node parallel execution will not scale with an undersized interconnect. From Oracle Database 11*g* onwards, it is recommended to use Oracle RAC services to control parallel execution on a cluster.

### 4.5.4 Optimizing Star Queries Using VECTOR GROUP BY Aggregation

`VECTOR GROUP BY` aggregation optimizes queries that aggregate data and join one or more relatively small tables to a larger table. This transformation can be chosen by the SQL optimizer based on cost estimates. In the context of data warehousing, `VECTOR GROUP BY` will often be chosen for star queries that select data from in-memory columnar tables.


Copyright © 2001, 2025, Oracle and/or its affiliates.

`VECTOR GROUP BY` aggregation is similar to a bloom filter in that it transforms the join condition between a small table and a large table into a filter on the larger table. `VECTOR GROUP BY` aggregation further enhances query performance by aggregating data during the scan of the fact table rather than as a separate step following the scan.

**See Also**

- Using In-Memory Aggregation
- *Oracle AI Database In-Memory Guide* for a detailed `VECTOR GROUP BY` scenario

# 4.6 About Approximate Query Processing

Approximate query processing uses SQL functions to provide real-time responses to explorative queries where approximations are acceptable. A query containing SQL functions that return approximate results is referred to as an approximate query.

Business intelligence (BI) applications extensively use aggregate functions, including analytic functions, to provide answers to common business queries. For some types of queries, when the data set is extremely large, providing exact answers can be resource intensive. For example, counting the number of unique customer sessions on a website or establishing the median house price within each zip code across a state. In certain scenarios, these types of queries may not require exact answers because you are more interested in approximate trends or patterns, which can then be used to drive further analysis. Approximate query processing is primarily used in data discovery applications to return quick answers to explorative queries. Users typically want to locate interesting data points within large amounts of data and then drill down to uncover further levels of detail. For explorative queries, quick responses are more important than exact values.

Oracle provides a set of SQL functions that enable you to obtain approximate results with negligible deviation from the exact result. There are additional approximate functions that support materialized view based summary aggregation strategies. The functions that provide approximate results are as follows:

- `APPROX_COUNT_DISTINCT`
- `APPROX_COUNT_DISTINCT_DETAIL`
- `APPROX_COUNT_DISTINCT_AGG`
- `TO_APPROX_COUNT_DISTINCT`
- `APPROX_MEDIAN`
- `APPROX_PERCENTILE`
- `APPROX_PERCENTILE_DETAIL`
- `APPROX_PERCENTILE_AGG`
- `TO_APPROX_PERCENTILE`

Approximate query processing can be used without any changes to your existing code. When you set the appropriate initialization parameters, Oracle AI Database replaces exact functions in queries with the corresponding SQL functions that return approximate results.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**See Also**

- Running Queries Containing Exact Functions Using SQL Functions that Return Approximate Values
- Creating Materialized Views Based on Approximate Queries
- Query Rewrite and Materialized Views Based on Approximate Queries
- Refreshing Materialized Views Based on Approximate Queries
- About Approximate Aggregates
- Using Percentile Functions that Return Approximate Results
- *Oracle AI Database SQL Language Reference* for information about the SQL functions

## 4.6.1 Running Queries Containing Exact Functions Using SQL Functions that Return Approximate Values

Queries containing exact functions can be run by using the corresponding SQL functions that return approximate results, without modifying the queries. This enables you to run existing applications, without modifications, by using the corresponding SQL functions that return approximate results.

Oracle AI Database provides the following initialization parameters to indicate that exact functions must be replaced with the corresponding SQL functions that return approximate results at runtime: `approx_for_aggregation`, `approx_for_count_distinct`, and `approx_for_percentile`. You can replace all exact functions at runtime with the corresponding functions that return approximate results. If you need more fine-grained control over the list of functions that must be replaced with their corresponding approximate versions, then you can specify the type of functions that must be replaced at runtime. For example, if a query contains `COUNT(DISTINCT)`, then setting `approx_for_aggregation` to `TRUE` results in this query being run using `APPROX_COUNT_DISTINCT` instead of `COUNT(DISTINCT)`.

- To run all queries using the corresponding SQL functions that return approximate results instead of the specified SQL functions:

  Set the `approx_for_aggregation` initialization parameter to `TRUE` for the current session or for the entire database. This parameter acts as an umbrella parameter for enabling the use of functions that return approximate results. Setting this is equivalent to setting the `APPROX_COUNT_DISTINCT` and `APPROX_FOR_PERCENTILE` parameters.

  The following command sets `approx_for_aggregation` to true for the current session:

  ```
  alter session set approx_for_aggregation = TRUE;
  ```

- To replace only the `COUNT(DISTINCT)` function in queries with the `APPROX_COUNT_DISTINCT` function:

  Set the `approx_for_count_distinct` initialization parameter to `TRUE` for the current session or for the entire database.

- To replace percentile functions with the corresponding functions that return approximate results:


Copyright © 2001, 2025, Oracle and/or its affiliates.

Set `approx_for_percentile` to `PERCENTILE_CONT`, `PERCENTILE_DISC`, or `ALL` (replaces all percentile functions) for the current session or for the entire database. The default value of this parameter is `NONE`.

> **See Also**
>
> – `APROX_FOR_AGGREGATION` in *Oracle AI Database Reference*
> – `APPROX_FOR_COUNT_DISTINCT` in *Oracle AI Database Reference*
> – `APPROX_FOR_PERCENTILE` in *Oracle AI Database Reference*

# 4.7 About Approximate Top-N Query Processing

Starting with Oracle Database Release 18c, to obtain top N query results much faster than traditional queries, the `APPROX_COUNT` and `APPROX_SUM` SQL functions can be used with `APPROX_RANK`.

### APPROX_COUNT

`APPROX_COUNT` returns the approximate count of an expression. If `MAX_ERROR` is supplied as the second argument, then the function returns the maximum error between the actual and approximate count.

This function must be used with a corresponding `APPROX_RANK` function in the `HAVING` clause. If a query uses `APPROX_COUNT`, `APPROX_SUM`, or `APPROX_RANK`, then the query must not use any other aggregation functions.

> **See Also**
>
> - *Oracle AI Database SQL Language Reference*
> - APPROX_RANK Function

### APPROX_SUM

`APPROX_SUM` returns the approximate sum of an expression. If `MAX_ERROR` is supplied as the second argument, then the function returns the maximum error between the actual and approximate sum.

This function must be used with a corresponding `APPROX_RANK` function in the `HAVING` clause. If a query uses `APPROX_COUNT`, `APPROX_SUM`, or `APPROX_RANK`, then the query must not use any other aggregation functions.

> **Note**
>
> `APPROX_SUM` returns an error when the input is a negative number.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**See Also**

- *Oracle AI Database SQL Language Reference*
- APPROX_RANK Function


Copyright © 2001, 2025, Oracle and/or its affiliates.

# Part II
# Optimizing Data Warehouses

This section deals with the physical design of a data warehouse.

It contains the following chapters:


- Basic Materialized Views
- Advanced Materialized Views
- Refreshing Materialized Views
- Synchronous Refresh
- Monitoring Materialized View Refresh Operations
- Dimensions
- Basic Query Rewrite for Materialized Views
- Advanced Query Rewrite for Materialized Views
- Attribute Clustering
- Using Zone Maps




Copyright © 2001, 2025, Oracle and/or its affiliates.

5

# Basic Materialized Views

This chapter describes the use of materialized views. It contains the following topics:


- Overview of Data Warehousing with Materialized Views
- Types of Materialized Views
- Creating Materialized Views
- Creating Materialized View Logs
- Creating Materialized Views Based on Approximate Queries
- Registering Existing Materialized Views
- Choosing Indexes for Materialized Views
- Dropping Materialized Views
- Analyzing Materialized View Capabilities


## 5.1 Overview of Data Warehousing with Materialized Views

Typically, data flows from one or more online transaction processing (OLTP) database into a data warehouse on a monthly, weekly, or daily basis. The data is normally processed in a staging file before being added to the data warehouse. Data warehouses commonly range in size from hundreds of gigabytes to petabytes. Usually, the vast majority of the data is stored in a few very large fact tables.

One technique employed in data warehouses to improve performance is the creation of summaries. Summaries are special types of aggregate views that improve query execution times by precalculating expensive joins and aggregation operations prior to execution and storing the results in a table in the database. For example, you can create a summary table to contain the sums of sales by region and by product.

The summaries or aggregates that are referred to in this book and in literature on data warehousing are created in Oracle AI Database using a schema object called a materialized view. Materialized views can perform a number of roles, such as improving query performance or providing replicated data.

The database administrator creates one or more materialized views, which are the equivalent of a summary. The end user queries the tables and views at the detail data level. The query rewrite mechanism in the Oracle server automatically rewrites the SQL query to use the summary tables. This mechanism reduces response time for returning results from the query. Materialized views within the data warehouse are transparent to the end user or to the database application.

Although materialized views are usually accessed through the query rewrite mechanism, an end user or database application can construct queries that directly access the materialized views. However, serious consideration should be given to whether users should be allowed to do this because any change to the materialized views affects the queries that reference them.

This section contains the following topics:


- About Materialized Views for Data Warehouses




Copyright © 2001, 2025, Oracle and/or its affiliates.

- About Materialized Views for Distributed Computing
- About Materialized Views for Mobile Computing
- The Need for Materialized Views
- Components of Summary Management
- Data Warehousing Terminology
- About Materialized View Schema Design
- About Loading Data into Data Warehouses
- Overview of Materialized View Management Tasks


## 5.1.1 About Materialized Views for Data Warehouses

In data warehouses, you can use materialized views to precompute and store aggregated data such as the sum of sales. Materialized views in these environments are often referred to as summaries, because they store summarized data. They can also be used to precompute joins with or without aggregations. A materialized view eliminates the overhead associated with expensive joins and aggregations for a large or important class of queries.

## 5.1.2 About Materialized Views for Distributed Computing

In distributed environments, you can use materialized views to replicate data at distributed sites and to synchronize updates done at those sites with conflict resolution methods. These replica materialized views provide local access to data that otherwise would have to be accessed from remote sites. Materialized views are also useful in remote data marts.

> **See Also**
>
> *Oracle AI Database Heterogeneous Connectivity User's Guide*

## 5.1.3 About Materialized Views for Mobile Computing

You can also use materialized views to download a subset of data from central servers to mobile clients, with periodic refreshes and updates between clients and the central servers. This chapter focuses on the use of materialized views in data warehouses.

> **See Also**
>
> *Oracle AI Database Heterogeneous Connectivity User's Guide*

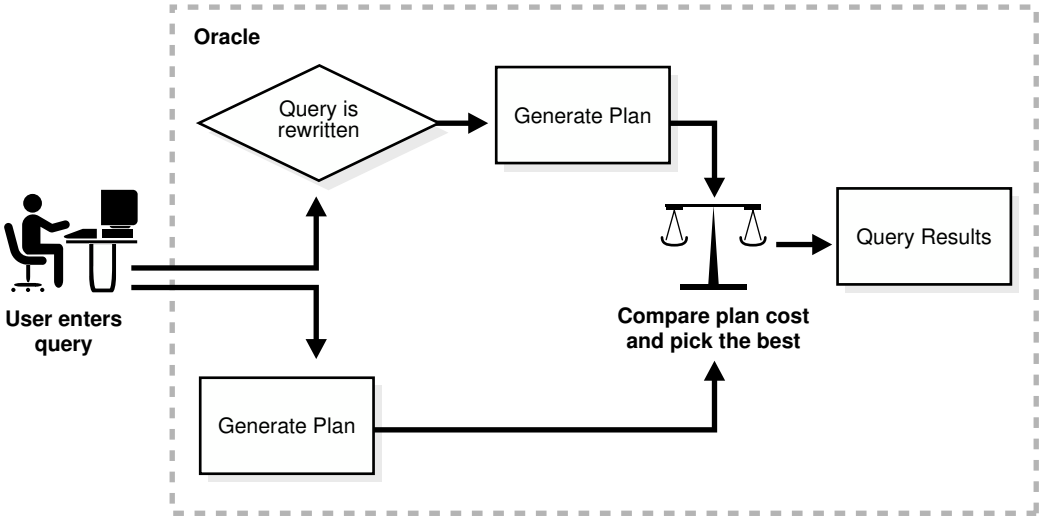
## 5.1.4 The Need for Materialized Views

You can use materialized views to increase the speed of queries on very large databases. Queries to large databases often involve joins between tables, aggregations such as `SUM`, or both. These operations are expensive in terms of time and processing power. The type of materialized view you create determines how the materialized view is refreshed and used by query rewrite.


Copyright © 2001, 2025, Oracle and/or its affiliates.

Materialized views improve query performance by precalculating expensive join and aggregation operations on the database prior to execution and storing the results in the database. The query optimizer automatically recognizes when an existing materialized view can and should be used to satisfy a request. It then transparently rewrites the request to use the materialized view. Queries go directly to the materialized view and not to the underlying detail tables. In general, rewriting queries to use materialized views rather than detail tables improves response time. Figure 5-1 illustrates how query rewrite works.

**Figure 5-1 Transparent Query Rewrite**




When using query rewrite, create materialized views that satisfy the largest number of queries. For example, if you identify 20 queries that are commonly applied to the detail or fact tables, then you might be able to satisfy them with five or six well-written materialized views. A materialized view definition can include any number of aggregations (`AVG`, `BIT_AND_AGG`, `BIT_OR_AGG`, `BIT_XOR_AGG`, `COUNT(x)`, `COUNT(*)`, `COUNT(DISTINCT x)`, `KURTOSIS_POP`, `KURTOSIS_SAMP`, `MAX`, `MIN`, `SKEWNESS_POP`, `SKEWNESS_SAMP`, `STDDEV`, `SUM`, and `VARIANCE`). It can also include any number of joins. If you are unsure of which materialized views to create, Oracle AI Database provides the SQL Access Advisor, which is a set of advisory procedures in the `DBMS_ADVISOR` package to help in designing and evaluating materialized views for query rewrite.

If a materialized view is to be used by query rewrite, it must be stored in the same database as the detail tables on which it depends. A materialized view can be partitioned, and you can define a materialized view on a partitioned table. You can also define one or more indexes on the materialized view.

Unlike indexes, materialized views can be accessed directly using a `SELECT` statement. However, it is recommended that you try to avoid writing SQL statements that directly reference the materialized view, because then it is difficult to change them without affecting the application. Instead, let query rewrite transparently rewrite your query to use the materialized view.

Note that the techniques shown in this chapter illustrate how to use materialized views in data warehouses. Materialized views can also be used by Oracle Replication.


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 5.1.5 Components of Summary Management

Summary management consists of:

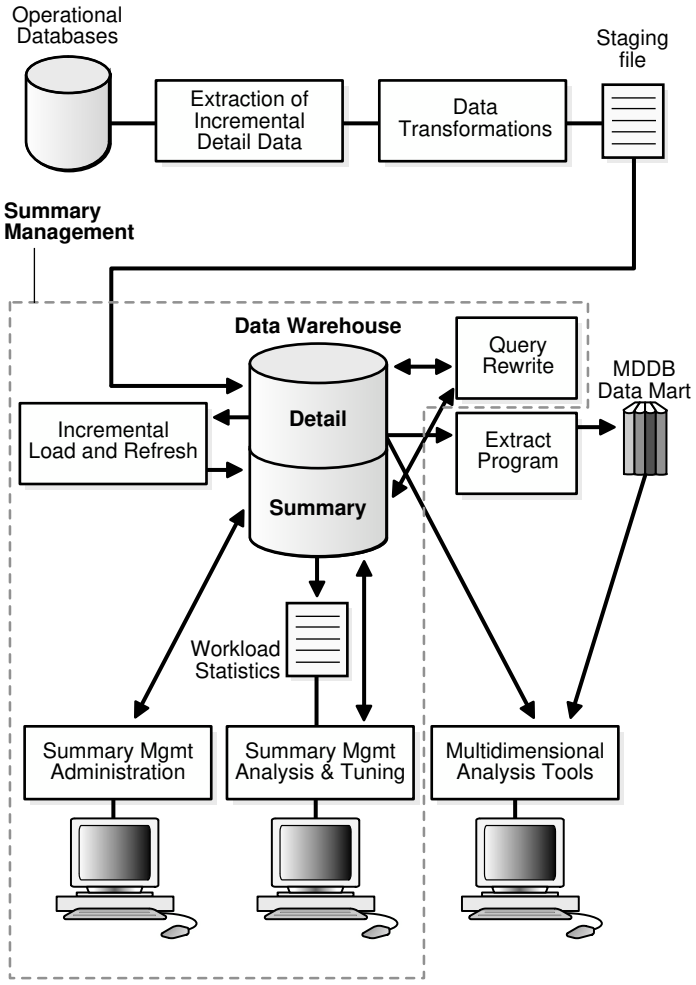
- Mechanisms to define materialized views and dimensions.
- A refresh mechanism to ensure that all materialized views contain the latest data.
- A query rewrite capability to transparently rewrite a query to use a materialized view.
- The SQL Access Advisor, which recommends materialized views, partitions, and indexes to create.
- The `TUNE_MVIEW` package, which shows you how to make your materialized view fast refreshable and use general query rewrite.

The use of summary management features imposes no schema restrictions, and can enable some existing DSS database applications to improve performance without the need to redesign the database or the application.

Figure 5-2 illustrates the use of summary management in the warehousing cycle. After the data has been transformed, staged, and loaded into the detail data in the warehouse, you can invoke the summary management process. First, use the SQL Access Advisor to plan how you will use materialized views. Then, create materialized views and design how queries will be rewritten. If you are having problems trying to get your materialized views to work then use `TUNE_MVIEW` to obtain an optimized materialized view.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Figure 5-2 Overview of Summary Management**




Understanding the summary management process during the earliest stages of data warehouse design can yield large dividends later in the form of higher performance, lower summary administration costs, and reduced storage requirements.

## 5.1.6 Data Warehousing Terminology

Some basic data warehousing terms are defined as follows:

- **Dimension tables** describe the business entities of an enterprise, represented as hierarchical, categorical information such as time, departments, locations, and products. Dimension tables are sometimes called lookup or reference tables.

  Dimension tables usually change slowly over time and are not modified on a periodic schedule. They are used in long-running decision support queries to aggregate the data returned from the query into appropriate levels of the dimension hierarchy.
- **Hierarchies** describe the business relationships and common access patterns in the database. An analysis of the dimensions, combined with an understanding of the typical work load, can be used to create materialized views. See Dimensions for more information.
- **Fact tables** describe the business transactions of an enterprise.


Copyright © 2001, 2025, Oracle and/or its affiliates.

The vast majority of data in a data warehouse is stored in a few very large fact tables that are updated periodically with data from one or more operational OLTP databases.

Fact tables include facts (also called measures) such as sales, units, and inventory.

- A simple measure is a numeric or character column of one table such as `fact.sales`.
- A computed measure is an expression involving measures of one table, for example, `fact.revenues - fact.expenses`.
- A multitable measure is a computed measure defined on multiple tables, for example, `fact_a.revenues - fact_b.expenses`.

Fact tables also contain one or more foreign keys that organize the business transactions by the relevant business entities such as time, product, and market. In most cases, these foreign keys are non-null, form a unique compound key of the fact table, and each foreign key joins with exactly one row of a dimension table.

- A materialized view is a precomputed table comprising aggregated and joined data from fact and possibly from dimension tables.

## 5.1.7 About Materialized View Schema Design

Summary management can perform many useful functions, including query rewrite and materialized view refresh, even if your data warehouse design does not follow these guidelines. However, you realize significantly greater query execution performance and materialized view refresh performance benefits and you require fewer materialized views if your schema design complies with these guidelines.

A materialized view definition includes any number of aggregates, as well as any number of joins. In several ways, a materialized view behaves like an index:

- The purpose of a materialized view is to increase query execution performance.
- The existence of a materialized view is transparent to SQL applications, so that a database administrator can create or drop materialized views at any time without affecting the validity of SQL applications.
- A materialized view consumes storage space.
- The contents of the materialized view must be updated when the underlying detail tables are modified.

This section contains the following topics:

- Schemas and Dimension Tables
- Guidelines for Materialized View Schema Design

### 5.1.7.1 Schemas and Dimension Tables

In the case of normalized or partially normalized dimension tables (a dimension that is stored in multiple tables), identify how these tables are joined. Note whether the joins between the dimension tables can guarantee that each child-side row joins with one and only one parent-side row. In the case of denormalized dimensions, determine whether the child-side columns uniquely determine the parent-side (or attribute) columns. These relationships can be enabled with constraints, using the `NOVALIDATE` and `RELY` options if the relationships represented by the constraints are guaranteed by other means. Note that if the joins between fact and dimension tables do not support the parent-child relationship described previously, you still gain significant performance advantages from defining the dimension with the `CREATE DIMENSION` statement. Another alternative, subject to some restrictions, is to use outer joins in the materialized view definition (that is, in the `CREATE MATERIALIZED VIEW` statement).


Copyright © 2001, 2025, Oracle and/or its affiliates.

You must not create dimensions in any schema that does not satisfy these relationships. Incorrect results can be returned from queries otherwise.

## 5.1.7.2 Guidelines for Materialized View Schema Design

Before starting to define and use the various components of summary management, you should review your schema design to abide by the following guidelines wherever possible. Guidelines 1 and 2 are more important than guideline 3. If your schema design does not follow guidelines 1 and 2, it does not then matter whether it follows guideline 3. Guidelines 1, 2, and 3 affect both query rewrite performance and materialized view refresh performance.

**Dimensions Guideline 1**

Dimensions should either be denormalized (each dimension contained in one table) or the joins between tables in a normalized or partially normalized dimension should guarantee that each child-side row joins with exactly one parent-side row.

You can enforce this condition by adding `FOREIGN KEY` and `NOT NULL` constraints on the child-side join keys and `PRIMARY KEY` constraints on the parent-side join keys.

**Dimensions Guideline 2**

If dimensions are denormalized or partially denormalized, hierarchical integrity must be maintained between the key columns of the dimension table. Each child key value must uniquely identify its parent key value, even if the dimension table is denormalized. Hierarchical integrity in a denormalized dimension can be verified by calling the `VALIDATE_DIMENSION` procedure of the `DBMS_DIMENSION` package.

**Dimensions Guideline 3**

Fact and dimension tables should similarly guarantee that each fact table row joins with exactly one dimension table row. This condition must be declared, and optionally enforced, by adding `FOREIGN KEY` and `NOT NULL` constraints on the fact key column(s) and `PRIMARY KEY` constraints on the dimension key column(s), or by using outer joins. In a data warehouse, constraints are typically enabled with the `NOVALIDATE` and `RELY` clauses to avoid constraint enforcement performance overhead.

**Dimensions Guideline 4**

After each load and before refreshing your materialized view, use the `VALIDATE_DIMENSION` procedure of the `DBMS_DIMENSION` package to incrementally verify dimensional integrity.

**Incremental Loads Guideline**

Incremental loads of your detail data should be done using the SQL*Loader direct-path option, or any bulk loader utility that uses Oracle's direct-path interface. This includes `INSERT ... AS SELECT` with the `APPEND` or `PARALLEL` hints, where the hints cause the direct loader log to be used during the insert.

**Partitions Guideline**

Range/composite partition your tables by a monotonically increasing time column if possible (preferably of type `DATE`).

**Time Dimensions Guideline**

If a time dimension appears in the materialized view as a time column, partition and index the materialized view in the same manner as you have the fact tables.


Copyright © 2001, 2025, Oracle and/or its affiliates.

If you are concerned with the time required to enable constraints and whether any constraints might be violated, then use the `ENABLE NOVALIDATE` with the `RELY` clause to turn on constraint checking without validating any of the existing constraints. The risk with this approach is that incorrect query results could occur if any constraints are broken. Therefore, as the designer, you must determine how clean the data is and whether the risk of incorrect results is too great.

**See Also**

- "Types of Materialized Views"
- "Creating Dimensions" for details on the benefits of maintaining a child-side row join with a parent-side row
- *Oracle AI Database SQL Language Reference*

## 5.1.8 About Loading Data into Data Warehouses

A popular and efficient way to load data into a data warehouse or data mart is to use a `CREATE TABLE AS SELECT` or `INSERT AS SELECT` statement, loading external data using external tables. This allows massively parallel, complex data loading using the power of SQL inside the database, avoiding any unnecessary staging.

Alternative ways are to load data using SQL*Loader with the `DIRECT` or `PARALLEL` option, Data Pump, or to use another loader tool that uses the Oracle direct-path API.

Loading strategies can be classified as one-phase or two-phase. In one-phase loading, data is loaded directly into the target table, quality assurance tests are performed, and errors are resolved by performing DML operations prior to refreshing materialized views. If a large number of deletions are possible, then storage utilization can be adversely affected, but temporary space requirements and load time are minimized.

In a two-phase loading process:

- Data is first loaded into a temporary table in the warehouse.
- Quality assurance procedures are applied to the data.
- Referential integrity constraints on the target table are disabled, and the local index in the target partition is marked unusable.
- The data is copied from the temporary area into the appropriate partition of the target table using `INSERT AS SELECT` with the `PARALLEL` or `APPEND` hint. The temporary table is then dropped. Alternatively, if the target table is partitioned, you can create a new (empty) partition in the target table and use `ALTER TABLE ... EXCHANGE PARTITION` to incorporate the temporary table into the target table. See *Oracle AI Database SQL Language Reference* for more information.
- The constraints are enabled, usually with the `NOVALIDATE` option.

Immediately after loading the detail data and updating the indexes on the detail data, the database can be opened for operation, if desired. You can disable query rewrite at the system level by issuing an `ALTER SYSTEM SET QUERY_REWRITE_ENABLED = FALSE` statement until all the materialized views are refreshed.

If `QUERY_REWRITE_INTEGRITY` is set to `STALE_TOLERATED`, access to the materialized view can be allowed at the session level to any users who do not require the materialized views to reflect the data from the latest load by issuing an `ALTER SESSION SET QUERY_REWRITE_ENABLED = TRUE` statement. This scenario does not apply when `QUERY_REWRITE_INTEGRITY` is either


Copyright © 2001, 2025, Oracle and/or its affiliates.

ENFORCED or TRUSTED because the system ensures in these modes that only materialized views with updated data participate in a query rewrite.

> **See Also**
>
> *Oracle AI Database Utilities* for the restrictions and considerations when using SQL*Loader with the DIRECT or PARALLEL keywords

## 5.1.9 Overview of Materialized View Management Tasks

The motivation for using materialized views is to improve performance, but the overhead associated with materialized view management can become a significant system management problem. When reviewing or evaluating some of the necessary materialized view management activities, consider some of the following:

- Identifying what materialized views to create initially.
- Indexing the materialized views.
- Ensuring that all materialized views and materialized view indexes are refreshed properly each time the database is updated.
- Checking which materialized views have been used.
- Determining how effective each materialized view has been on workload performance.
- Measuring the space being used by materialized views.
- Determining which new materialized views should be created.
- Determining which existing materialized views should be dropped.
- Archiving old detail and materialized view data that is no longer useful.

After the initial effort of creating and populating the data warehouse or data mart, the major administration overhead is the update process, which involves:

- Periodic extraction of incremental changes from the operational systems.
- Transforming the data.
- Verifying that the incremental changes are correct, consistent, and complete.
- Bulk-loading the data into the warehouse.
- Refreshing indexes and materialized views so that they are consistent with the detail data.

The update process must generally be performed within a limited period of time known as the update window. The update window depends on the update frequency (such as daily or weekly) and the nature of the business. For a daily update frequency, an update window of two to six hours might be typical.

You need to know your update window for the following activities:

- Loading the detail data
- Updating or rebuilding the indexes on the detail data
- Performing quality assurance tests on the data
- Refreshing the materialized views
- Updating the indexes on the materialized views


Copyright © 2001, 2025, Oracle and/or its affiliates.

# 5.2 Types of Materialized Views

The SELECT clause in the materialized view creation statement defines the data that the materialized view is to contain. Only a few restrictions limit what can be specified. Any number of tables can be joined together. Besides tables, other elements such as views, inline views (subqueries in the FROM clause of a SELECT statement), subqueries, and materialized views can all be joined or referenced in the SELECT clause. You cannot, however, define a materialized view with a subquery in the SELECT list of the defining query. You can, however, include subqueries elsewhere in the defining query, such as in the WHERE clause.

The types of materialized views are:

- About Materialized Views with Aggregates
- About Materialized Views Containing Only Joins
- About Nested Materialized Views

## 5.2.1 About Materialized Views with Aggregates

In data warehouses, materialized views normally contain aggregates as shown in Example 5-1. For fast refresh to be possible, the SELECT list must contain all of the GROUP BY columns (if present), and there must be a COUNT(*) and a COUNT(column) on any aggregated columns. Also, materialized view logs must be present on all tables referenced in the query that defines the materialized view. The valid aggregate functions are: AVG, BIT_AND_AGG, BIT_OR_AGG, BIT_XOR_AGG, COUNT(x), COUNT(*), COUNT(x), KURTOSIS_POP, KURTOSIS_SAMP, MAX, MIN, SKEWNESS_POP, SKEWNESS_SAMP, STDDEV, SUM, and VARIANCE, and the expression to be aggregated can be any SQL value expression. See "Restrictions on Fast Refresh on Materialized Views with Aggregates".

> **See Also**
>
> "Requirements for Using Materialized Views with Aggregates"

Fast refresh for a materialized view containing joins and aggregates is possible after any type of DML to the base tables (direct load or conventional INSERT, UPDATE, or DELETE). It can be defined to be refreshed ON COMMIT or ON DEMAND. A REFRESH ON COMMIT materialized view is refreshed automatically when a transaction that does DML to one of the materialized view's detail tables commits. The time taken to complete the commit may be slightly longer than usual when this method is chosen. This is because the refresh operation is performed as part of the commit process. Therefore, this method may not be suitable if many users are concurrently changing the tables upon which the materialized view is based.

Here are some examples of materialized views with aggregates. Note that materialized view logs are only created because this materialized view is fast refreshed.

**Example 5-1 Creating a Materialized View (Total Number and Value of Sales)**

```
CREATE MATERIALIZED VIEW LOG ON products
WITH SEQUENCE, ROWID
(prod_id, prod_name, prod_desc, prod_subcategory, prod_subcat_desc,
prod_category, prod_cat_desc, prod_weight_class, prod_unit_of_measure,
 prod_pack_size, supplier_id, prod_status, prod_list_price, prod_min_price)
INCLUDING NEW VALUES;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
CREATE MATERIALIZED VIEW LOG ON sales
WITH SEQUENCE, ROWID
(prod_id, cust_id, time_id, channel_id, promo_id, quantity_sold, amount_sold)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW product_sales_mv
BUILD IMMEDIATE
REFRESH FAST
ENABLE QUERY REWRITE
AS SELECT p.prod_name, SUM(s.amount_sold) AS dollar_sales,
COUNT(*) AS cnt, COUNT(s.amount_sold) AS cnt_amt
FROM sales s join products p
ON (s.prod_id = p.prod_id) GROUP BY p.prod_name;
```

**Note**

As of Oracle AI Database 26ai, materialized views support full rewrite capabilities for SQL statements using ANSI join syntax and for materialized view definitions using ANSI join syntax. You can use either ANSI or Oracle syntax in your MV definition and your SQL statements. Any SQL statement will be rewritten as much as possible, independent of the chosen join syntax in either the MV or the SQL statement. In the above example, the join in the `CREATE MATERIALIZED VIEW` statement is expressed in ANSI syntax:

```
FROM sales s join products p
ON (s.prod_id = p.prod_id) GROUP BY p.prod_name
```

Oracle syntax is also valid:

```
FROM sales s, products p
WHERE s.prod_id = p.prod_id GROUP BY p.prod_name
```

This example creates a materialized view `product_sales_mv` that computes total number and value of sales for a product. It is derived by joining the tables `sales` and `products` on the column `prod_id`. The materialized view is populated with data immediately because the build method is immediate and it is available for use by query rewrite. In this example, the default refresh method is `FAST`, which is allowed because the appropriate materialized view logs have been created on tables `products` and `sales`.

You can achieve better fast refresh performance for local materialized views if you use a materialized view log that contains a `WITH COMMIT SCN` clause. An example is the following:

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID(prod_id, cust_id, time_id),
  COMMIT SCN INCLUDING NEW VALUES;
```

**Example 5-2 Creating a Materialized View (Computed Sum of Sales)**

```
CREATE MATERIALIZED VIEW product_sales_mv_agg
BUILD DEFERRED
REFRESH COMPLETE ON DEMAND
ENABLE QUERY REWRITE AS
SELECT p.prod_name, SUM(s.amount_sold) AS dollar_sales
FROM sales s, products p WHERE s.prod_id = p.prod_id
GROUP BY p.prod_name;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

Using ANSI join syntax to illustrate the same example, the equivalent materialized view would look like this:

```
CREATE MATERIALIZED VIEW product_sales_mv_agg
BUILD DEFERRED
REFRESH COMPLETE ON DEMAND
ENABLE QUERY REWRITE AS
SELECT p.prod_name, SUM(s.amount_sold) AS dollar_sales
FROM sales s join products p on (s.prod_id = p.prod_id)
GROUP BY p.prod_name;
```

This example creates a materialized view `product_sales_mv` that computes the sum of sales by `prod_name`. It is derived by joining the tables `sales` and `products` on the column `prod_id`. The materialized view does not initially contain any data, because the build method is `DEFERRED`. A complete refresh is required for the first refresh of a build deferred materialized view. When it is refreshed and once populated, this materialized view can be used by query rewrite.

**Example 5-3 Creating a Materialized View (Aggregates on a Single Table)**

```
CREATE MATERIALIZED VIEW LOG ON sales WITH SEQUENCE, ROWID
(prod_id, cust_id, time_id, channel_id, promo_id, quantity_sold, amount_sold)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW sum_sales
PARALLEL
BUILD IMMEDIATE
REFRESH FAST ON COMMIT AS
SELECT s.prod_id, s.time_id, COUNT(*) AS count_grp,
   SUM(s.amount_sold) AS sum_dollar_sales,
   COUNT(s.amount_sold) AS count_dollar_sales,
   SUM(s.quantity_sold) AS sum_quantity_sales,
   COUNT(s.quantity_sold) AS count_quantity_sales
FROM sales s
GROUP BY s.prod_id, s.time_id;
```

This example creates a materialized view that contains aggregates on a single table. Because the materialized view log has been created with all referenced columns in the materialized view's defining query, the materialized view is fast refreshable. If DML is applied against the `sales` table, then the changes are reflected in the materialized view when the commit is issued.

> **See Also**
>
> *Oracle AI Database SQL Language Reference* for syntax of the `CREATE MATERIALIZED VIEW` and `CREATE MATERIALIZED VIEW LOG` statements

## 5.2.1.1 Requirements for Using Materialized Views with Aggregates

Table 5-1 illustrates the aggregate requirements for materialized views. If aggregate `X` is present, aggregate `Y` is required and aggregate `Z` is optional.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Table 5-1 Requirements for Materialized Views with Aggregates**

| X | Y | Z |
|---|---|---|
| BIT_AND_AGG | - | - |
| BIT_OR_AGG | - | - |
| BIT_XOR_AGG | - | - |
| COUNT(expr) | - | - |
| MIN(expr) | - | - |
| MAX(expr) | - | - |
| SUM(expr) | COUNT(expr) | - |
| SUM(col), col has NOT NULL constraint | - | - |
| AVG(expr) | COUNT(expr) | SUM(expr) |
| STDDEV(expr) | COUNT(expr) SUM(expr) | SUM(expr * expr) |
| VARIANCE(expr) | COUNT(expr) SUM(expr) | SUM(expr * expr) |
| KURTOSIS_POP(expr) KURTOSIS_SAMP(expr) | COUNT(expr) SUM(expr) | SUM(expr^2) COUNT(expr^2) SUM(expr^3) COUNT(expr^3) |
| SKEWNESS_POP(expr) SKEWNESS_SAMP(expr) | COUNT(expr) SUM(expr) VARIANCE(expr) COUNT(*) | SUM(expr^2) COUNT(expr^2) |

Note that COUNT(*) must always be present to guarantee all types of fast refresh. Otherwise, you may be limited to fast refresh after inserts only. Oracle recommends that you include the optional aggregates in column Z in the materialized view in order to obtain the most efficient and accurate fast refresh of the aggregates.

## 5.2.2 About Materialized Views Containing Only Joins

Some materialized views contain only joins and no aggregates , such as in Materialized Join Views FROM Clause Considerations, where a materialized view is created that joins the sales table to the times and customers tables. The advantage of creating this type of materialized view is that expensive joins are precalculated.

> **Note**
>
> As of Oracle AI Database 26ai, ANSI and Oracle join syntax are interchangeable. Both are fully supported.

> **See Also**
>
> "Materialized Join Views FROM Clause Considerations"

Fast refresh for a materialized view containing only joins is possible after any type of DML to the base tables (direct-path or conventional INSERT, UPDATE, or DELETE).


Copyright © 2001, 2025, Oracle and/or its affiliates.

A materialized view containing only joins can be defined to be refreshed `ON COMMIT` or `ON DEMAND`. If it is `ON COMMIT`, the refresh is performed at commit time of the transaction that does DML on the materialized view's detail table.

If you specify `REFRESH FAST`, Oracle AI Database performs further verification of the query definition to ensure that fast refresh can be performed if any of the detail tables change. These additional checks are:

- A materialized view log must be present for each detail table unless the table supports partition change tracking (PCT). Also, when a materialized view log is required, the `ROWID` column must be present in each materialized view log.
- The rowids of all the detail tables must appear in the `SELECT` list of the materialized view query definition.

If some of these restrictions are not met, you can create the materialized view as `REFRESH FORCE` to take advantage of fast refresh when it is possible. If one of the tables did not meet all of the criteria, but the other tables did, the materialized view would still be fast refreshable with respect to the other tables for which all the criteria are met.

To achieve an optimally efficient refresh, you should ensure that the defining query does not use an outer join that behaves like an inner join. If the defining query contains such a join, consider rewriting the defining query to contain an inner join.

**See Also**

- "Restrictions on Fast Refresh on Materialized Views with Joins Only" for more information regarding the conditions that cause refresh performance to degrade.
- "About Partition Change Tracking (PCT) Refresh for Materialized Views"

## 5.2.2.1 Materialized Join Views FROM Clause Considerations

If the materialized view contains only joins, the `ROWID` columns for each table (and each instance of a table that occurs multiple times in the `FROM` list) must be present in the `SELECT` list of the materialized view.

If the materialized view has remote tables in the `FROM` clause, all tables in the `FROM` clause must be located on that same site in order to perform incremental (fast) refresh for the materialized view. Further, `ON COMMIT` refresh is not supported for materialized view with remote tables. Except for SCN-based materialized view logs, materialized view logs must be present on the remote site for each detail table of the materialized view and `ROWID` columns must be present in the `SELECT` list of the materialized view, as shown in the following example.

**Example 5-4 Materialized View Containing Only Joins**

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID;
CREATE MATERIALIZED VIEW LOG ON times WITH ROWID;
CREATE MATERIALIZED VIEW LOG ON customers WITH ROWID;
CREATE MATERIALIZED VIEW detail_sales_mv
PARALLEL BUILD IMMEDIATE
REFRESH FAST AS
SELECT s.rowid "sales_rid", t.rowid "times_rid", c.rowid "customers_rid",
       c.cust_id, c.cust_last_name, s.amount_sold, s.quantity_sold, s.time_id
FROM sales s, times t, customers c
WHERE  s.cust_id = c.cust_id(+) AND s.time_id = t.time_id(+);
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

Alternatively, if the previous example did not include the columns `times_rid` and `customers_rid`, and if the refresh method was `REFRESH FORCE`, then this materialized view would be fast refreshable only if the sales table was updated but not if the tables `times` or `customers` were updated. Also note that this version uses ANSI join syntax. Both ANSI join and Oracle join syntax are equally supported.

```
CREATE MATERIALIZED VIEW detail_sales_mv2
PARALLEL
BUILD IMMEDIATE
REFRESH FORCE AS
SELECT s.rowid "sales_rid", c.cust_id, c.cust_last_name, s.amount_sold, s.quantity_sold,
s.time_id
FROM sales s
 RIGHT OUTER JOIN times t ON t.time_id = s.time_id
 RIGHT OUTER JOIN customers c ON c.cust_id = s.cust_id;
```

## 5.2.3 About Nested Materialized Views

A nested materialized view is a materialized view whose definition is based on another materialized view. A nested materialized view can reference other relations in the database in addition to referencing materialized views.

This section contains the following topics:

- Why Use Nested Materialized Views?
- About Nesting Materialized Views with Joins and Aggregates
- Nested Materialized View Usage Guidelines
- Restrictions When Using Nested Materialized Views

### 5.2.3.1 Why Use Nested Materialized Views?

In a data warehouse, you typically create many aggregate views on a single join (for example, rollups along different dimensions). Incrementally maintaining these distinct materialized aggregate views can take a long time, because the underlying join has to be performed many times.

Using nested materialized views, you can create multiple single-table materialized views based on a joins-only materialized view and the join is performed just once. In addition, optimizations can be performed for this class of single-table aggregate materialized view and thus refresh is very efficient.

**Example 5-5 Nested Materialized View**

You can create a nested materialized view on materialized views, but all parent and base materialized views must contain joins or aggregates. If the defining queries for a materialized view do not contain joins or aggregates, it cannot be nested. All the underlying objects (materialized views or tables) on which the materialized view is defined must have a materialized view log. All the underlying objects are treated as if they were tables. In addition, you can use all the existing options for materialized views.

Using the tables and their columns from the `sh` sample schema, the following materialized views illustrate how nested materialized views can be created.

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID;
CREATE MATERIALIZED VIEW LOG ON customers WITH ROWID;
CREATE MATERIALIZED VIEW LOG ON times WITH ROWID;

/*create materialized view join_sales_cust_time as fast refreshable at
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
   COMMIT time */
CREATE MATERIALIZED VIEW join_sales_cust_time
REFRESH FAST ON COMMIT AS
SELECT c.cust_id, c.cust_last_name, s.amount_sold, t.time_id,
       t.day_number_in_week, s.rowid srid, t.rowid trid, c.rowid crid
FROM sales s, customers c, times t
WHERE s.time_id = t.time_id AND s.cust_id = c.cust_id;
```

To create a nested materialized view on the table `join_sales_cust_time`, you would have to create a materialized view log on the table. Because this will be a single-table aggregate materialized view on `join_sales_cust_time`, you must log all the necessary columns and use the `INCLUDING NEW VALUES` clause.

```
/* create materialized view log on join_sales_cust_time */
CREATE MATERIALIZED VIEW LOG ON join_sales_cust_time
WITH ROWID (cust_last_name, day_number_in_week, amount_sold)
INCLUDING NEW VALUES;

/* create the single-table aggregate materialized view sum_sales_cust_time
on join_sales_cust_time as fast refreshable at COMMIT time */
CREATE MATERIALIZED VIEW sum_sales_cust_time
REFRESH FAST ON COMMIT AS
SELECT COUNT(*) cnt_all, SUM(amount_sold) sum_sales, COUNT(amount_sold)
       cnt_sales, cust_last_name, day_number_in_week
FROM join_sales_cust_time
GROUP BY cust_last_name, day_number_in_week;
```

## 5.2.3.2 About Nesting Materialized Views with Joins and Aggregates

Some types of nested materialized views cannot be fast refreshed. Use `EXPLAIN_MVIEW` to identify those types of materialized views. You can refresh a tree of nested materialized views in the appropriate dependency order by specifying the `nested = TRUE` parameter with the `DBMS_MVIEW.REFRESH` parameter. For example, if you call `DBMS_MVIEW.REFRESH ('SUM_SALES_CUST_TIME', nested => TRUE)`, the `REFRESH` procedure will first refresh the `join_sales_cust_time` materialized view, and then refresh the `sum_sales_cust_time` materialized view.

## 5.2.3.3 Nested Materialized View Usage Guidelines

You should keep the following in mind when deciding whether to use nested materialized views:

- If you want to use fast refresh, you should fast refresh all the materialized views along any chain.
- If you want the highest level materialized view to be fresh with respect to the detail tables, you must ensure that all materialized views in a tree are refreshed in the correct dependency order before refreshing the highest-level. You can automatically refresh intermediate materialized views in a nested hierarchy using the `nested = TRUE` parameter, as described in "About Nesting Materialized Views with Joins and Aggregates". If you do not specify `nested = TRUE` and the materialized views under the highest-level materialized view are stale, refreshing only the highest-level will succeed, but makes it fresh only with respect to its underlying materialized view, not the detail tables at the base of the tree.
- When refreshing materialized views, you must ensure that all materialized views in a tree are refreshed. If you only refresh the highest-level materialized view, the materialized views under it will be stale and you must explicitly refresh them. If you use the `REFRESH` procedure with the `nested` parameter value set to `TRUE`, only specified materialized views and their child materialized views in the tree are refreshed, and not their top-level


Copyright © 2001, 2025, Oracle and/or its affiliates.

materialized views. Use the REFRESH_DEPENDENT procedure with the nested parameter value set to TRUE if you want to ensure that all materialized views in a tree are refreshed.

- If complete refresh is the only refresh option supported for a particular nested materialized view, then a complete refresh is performed even when a fast refresh is specified.
- Freshness of a materialized view is calculated relative to the objects directly referenced by the materialized view. When a materialized view references another materialized view, the freshness of the topmost materialized view is calculated relative to changes in the materialized view it directly references, not relative to changes in the tables referenced by the materialized view it references.

### 5.2.3.4 Restrictions When Using Nested Materialized Views

You cannot create both a materialized view and a prebuilt materialized view on the same table. For example, If you have a table costs with a materialized view cost_mv based on it, you cannot then create a prebuilt materialized view on table costs. The result would make cost_mv a nested materialized view and this method of conversion is not supported.

## 5.3 Creating Materialized Views

A materialized view can be created with the CREATE MATERIALIZED VIEW statement.

It is not uncommon in a data warehouse to have already created summary or aggregation tables, and you might not wish to repeat this work by building a new materialized view. In this case, the table that already exists in the database can be registered as a prebuilt materialized view. This technique is described in "Registering Existing Materialized Views".

Once you have selected the materialized views you want to create, follow these steps for each materialized view.

1. Design the materialized view. Existing user-defined materialized views do not require this step.

   If the materialized view contains many rows, then, if appropriate, the materialized view should be partitioned (if possible) and should match the partitioning of the largest or most frequently updated detail or fact table (if possible). Refresh performance benefits from partitioning, because it can take advantage of parallel DML capabilities and possible PCT-based refresh.

2. Use the CREATE MATERIALIZED VIEW statement to create and, optionally, populate the materialized view.

   If a user-defined materialized view already exists, then use the ON PREBUILT TABLE clause in the CREATE MATERIALIZED VIEW statement. Otherwise, use the BUILD IMMEDIATE clause to populate the materialized view immediately, or the BUILD DEFERRED clause to populate the materialized view later. A BUILD DEFERRED materialized view is disabled for use by query rewrite until the first COMPLETE REFRESH, after which it is automatically enabled, provided the ENABLE QUERY REWRITE clause has been specified.

**Example 5-6 Creating a Materialized View**

This example illustrates creating a materialized view called cust_sales_mv.

```
CREATE MATERIALIZED VIEW cust_sales_mv
PARALLEL
BUILD IMMEDIATE
REFRESH COMPLETE
ENABLE QUERY REWRITE AS
SELECT  c.cust_last_name, SUM(amount_sold) AS sum_amount_sold
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
FROM customers c, sales s WHERE s.cust_id = c.cust_id
GROUP BY c.cust_last_name;
```

> **Note**
>
> Both Oracle join syntax (shown above) and ANSI join syntax are supported. Below is the same example rewritten to use ANSI join syntax.

```
CREATE MATERIALIZED VIEW cust_sales_mv
PARALLEL
BUILD IMMEDIATE
REFRESH COMPLETE
ENABLE QUERY REWRITE AS
SELECT  c.cust_last_name, SUM(amount_sold) AS sum_amount_sold
FROM customers c
JOIN sales s ON s.cust_id = c.cust_id
GROUP BY c.cust_last_name;
```

**Example 5-7 Creating a Materialized View with JSON Columns**

This example creates a materialized view based on a table `purchase_order` that contains a column of data type `JSON`.

```
CREATE MATERIALIZED VIEW po_mv
BUILD IMMEDIATE
REFRESH FAST ON STATEMENT WITH ROWID
AS
SELECT o.rowid AS id, v.*
FROM purchase_order o,
    JSON_TABLE(o.c FORMAT json, '$' error on error null on empty
           COLUMNS
           (
             poNum varchar2(10) PATH '$.poNum',
             poDate varchar2(12) PATH '$.poDate',
           NESTED PATH '$.items[*]'
            COLUMNS
            (
             item_seq for ordinality,
             itemName varchar2(10) PATH '$.itemName',
             itemPrice number PATH '$.itemPrice',
             itemQuantity varchar2(10) PATH '$.itemQuantity'
            )
       )
) v;
```

> **See Also**
>
> *Oracle AI Database SQL Language Reference*for descriptions of the SQL statements `CREATE MATERIALIZED VIEW`, `ALTER MATERIALIZED VIEW`, and `DROP MATERIALIZED VIEW`


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 5.3.1 Creating Materialized Views with Column Alias Lists

Currently, when a materialized view is created, if its defining query contains same-name columns in the SELECT list, the name conflicts need to be resolved by specifying unique aliases for those columns. Otherwise, the CREATE MATERIALIZED VIEW statement fails with the error messages of columns ambiguously defined. However, the standard method of attaching aliases in the SELECT clause for name resolution restricts the use of the full text match query rewrite and it will occur only when the text of the materialized view's defining query and the text of user input query are identical. Thus, if the user specifies select aliases in the materialized view's defining query while there is no alias in the query, the full text match comparison fails. This is particularly a problem for queries from Discoverer, which makes extensive use of column aliases.

The following is an example of the problem. sales_mv is created with column aliases in the SELECT clause but the input query Q1 does not have the aliases. The full text match rewrite fails. The materialized view is as follows:

```
CREATE MATERIALIZED VIEW sales_mv
ENABLE QUERY REWRITE AS
SELECT s.time_id sales_tid, c.time_id costs_tid
FROM sales s, products p, costs c
WHERE s.prod_id = p.prod_id AND c.prod_id = p.prod_id AND
      p.prod_name IN (SELECT prod_name FROM products);
```

Input query statement Q1 is as follows:

```
SELECT s.time_id, c1.time_id
FROM sales s, products p, costs c1
WHERE s.prod_id = p.prod_id AND c1.prod_id = p.prod_id AND
      p.prod_name IN (SELECT prod_name FROM products);
```

Even though the materialized view's defining query is almost identical and logically equivalent to the user's input query, query rewrite does not happen because of the failure of full text match that is the only rewrite possibility for some queries (for example, a subquery in the WHERE clause).

You can add a column alias list to a CREATE MATERIALIZED VIEW statement. The column alias list explicitly resolves any column name conflict without attaching aliases in the SELECT clause of the materialized view. The syntax of the materialized view column alias list is illustrated in the following example:

```
CREATE MATERIALIZED VIEW sales_mv (sales_tid, costs_tid)
ENABLE QUERY REWRITE AS
SELECT s.time_id, c.time_id
FROM sales s, products p, costs c
WHERE s.prod_id = p.prod_id AND c.prod_id = p.prod_id AND
      p.prod_name IN (SELECT prod_name FROM products);
```

In this example, the defining query of sales_mv now matches exactly with the user query Q1, so full text match rewrite takes place.

Note that when aliases are specified in both the SELECT clause and the new alias list clause, the alias list clause supersedes the ones in the SELECT clause.


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 5.3.2 Creating Materialized Views Based on Hybird Partitioned Tables

Use the `CREATE MATERIALIZED VIEW` statement to create a materialized view that is based on a hybrid partitioned table.

In a hybrid partitioned table, some partitions are stored in database segments, whereas other partitions are stored externally. If a materialized view that is based on a hybrid partitioned table includes the partition key or partition marker in its `SELECT` statement, it meets the requirements for PCT refresh.

To create a materialized view based on a hybrid partitioned table:

1. Create a hybrid partitioned table.

   The following command creates a hybrid partitioned table named `hybrid_sales`.

```
CREATE TABLE hybrid_sales(time_id date, customer number, price number, …)
…
PARTITION BY RANGE (time_id)
(
   PARTITION century_19 VALUES LESS THAN (TO_DATE('01-01-1900', 'DD-MM-
YYYY'))
       EXTERNAL LOCATION (data_dir1:'sales_1.csv'),
   PARTITION century_20 VALUES LESS THAN (TO_DATE('01-01-2000', 'DD-MM-
YYYY'))
       EXTERNAL DEFAULT DIRECTORY data_dir2 LOCATION ('sales_2.csv'),
   PARTITION year_2000 VALUES LESS THAN (TO_DATE('01-01-2001', 'DD-MM-
YYYY')),
   PARTITION year_2001 VALUES LESS THAN (TO_DATE('01-01-2002', 'DD-MM-
YYYY'))
);
```

2. Create a materialized view that is based on the hybrid partitioned table.

   The following command creates a materialized view named `hypt_mv` that is based on the hybrid partitioned table `hybrid_sales`:

```
CREATE MATERIALIZED VIEW HyPT_MV
REFRESH FAST ON DEMAND AS
SELECT time_id, customer_no, sum(price) as total_price
FROM hybrid_sales
GROUP BY time_id, customer_no;
```

   Assume that there is a corresponding materialized view log on the table `hybrid_sales`.

## 5.3.3 About Materialized Views Names

The name of a materialized view must conform to standard Oracle naming conventions. However, if the materialized view is based on a user-defined prebuilt table, then the name of the materialized view must exactly match that table name.

If you already have a naming convention for tables and indexes, you might consider extending this naming scheme to the materialized views so that they are easily identifiable. For example, instead of naming the materialized view `sum_of_sales`, it could be called `sum_of_sales_mv` to denote that this is a materialized view and not a table or view.


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 5.3.4 About Storage And Table Compression for Materialized Views

Unless the materialized view is based on a user-defined prebuilt table, it requires and occupies storage space inside the database. Therefore, the storage needs for the materialized view should be specified in terms of the tablespace where it is to reside and the size of the extents.

If you do not know how much space the materialized view requires, then the `DBMS_MVIEW.ESTIMATE_MVIEW_SIZE` package can estimate the number of bytes required to store this uncompressed materialized view. This information can then assist the design team in determining the tablespace in which the materialized view should reside.

You should use table compression with highly redundant data, such as tables with many foreign keys. This is particularly useful for materialized views created with the `ROLLUP` clause. Table compression reduces disk use and memory use (specifically, the buffer cache), often leading to a better scaleup for read-only operations. Table compression can also speed up query execution at the expense of update cost.

**See Also**

- *Oracle AI Database VLDB and Partitioning Guide* for more information about table compression
- *Oracle AI Database Administrator's Guide* for more information about table compression
- *Oracle AI Database SQL Language Reference* for a complete description of `STORAGE` semantics

## 5.3.5 About Build Methods for Materialized Views

Two build methods are available for creating the materialized view, as shown in Table 5-2. If you select `BUILD IMMEDIATE`, the materialized view definition is added to the schema objects in the data dictionary, and then the fact or detail tables are scanned according to the `SELECT` expression and the results are stored in the materialized view. Depending on the size of the tables to be scanned, this build process can take a considerable amount of time.

An alternative approach is to use the `BUILD DEFERRED` clause, which creates the materialized view without data, thereby enabling it to be populated at a later date using the `DBMS_MVIEW.REFRESH` package.

**See Also**

Refreshing Materialized Views

**Table 5-2 Build Methods**

| Build Method | Description |
|---|---|
| `BUILD IMMEDIATE` | Create the materialized view and then populate it with data. |
| `BUILD DEFERRED` | Create the materialized view definition but do not populate it with data. |


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 5.3.6 About Enabling Query Rewrite for Materialized Views

Before creating a materialized view, you can verify what types of query rewrite are possible by calling the procedure DBMS_MVIEW.EXPLAIN_MVIEW, or use DBMS_ADVISOR.TUNE_MVIEW to optimize the materialized view so that many types of query rewrite are possible. Once the materialized view has been created, you can use DBMS_MVIEW.EXPLAIN_REWRITE to find out if (or why not) it will rewrite a specific query.

Even though a materialized view is defined, it will not automatically be used by the query rewrite facility. Even though query rewrite is enabled by default, you also must specify the ENABLE QUERY REWRITE clause if the materialized view is to be considered available for rewriting queries.

If this clause is omitted or specified as DISABLE QUERY REWRITE when the materialized view is created, the materialized view can subsequently be enabled for query rewrite with the ALTER MATERIALIZED VIEW statement.

If you define a materialized view as BUILD DEFERRED, it is not eligible for query rewrite until it is populated with data through a complete refresh.

## 5.3.7 About Query Rewrite Restrictions

Query rewrite is not possible with all materialized views. If query rewrite is not occurring when expected, DBMS_MVIEW.EXPLAIN_REWRITE can help provide reasons why a specific query is not eligible for rewrite. If this shows that not all types of query rewrite are possible, use the procedure DBMS_ADVISOR.TUNE_MVIEW to see if the materialized view can be defined differently so that query rewrite is possible. Also, check to see if your materialized view satisfies all of the following conditions:

- About Materialized View Restrictions for Query Rewrite
- General Query Rewrite Restrictions

### 5.3.7.1 About Materialized View Restrictions for Query Rewrite

You should keep in mind the following restrictions:

- The defining query of the materialized view cannot contain any non-repeatable expressions (ROWNUM, SYSDATE, non-repeatable PL/SQL functions, and so on).
- The query cannot contain any references to LONG or LONG RAW data types or object REFs.
- If the materialized view was registered as PREBUILT, the precision of the columns must agree with the precision of the corresponding SELECT expressions unless overridden by the WITH REDUCED PRECISION clause.
- The defining query cannot contain any references to objects or XMLTYPEs.
- A materialized view is a noneditioned object and cannot depend on editioned objects unless it mentions an evaluation edition in which names of editioned objects are to be resolved.
- A materialized view may only be eligible for query rewrite in a specific range of editions. The *query_rewrite_clause* in the CREATE or ALTER MATERIALIZED VIEW statement lets you specify the range of editions in which a materialized view is eligible for query rewrite.


Copyright © 2001, 2025, Oracle and/or its affiliates.

> **See Also**
>
> - Advanced Query Rewrite for Materialized Views
> - *Oracle AI Database SQL Language Reference*

### 5.3.7.2 General Query Rewrite Restrictions

You should keep in mind the following restrictions:

- A query can reference both local and remote tables. Such a query can be rewritten as long as an eligible materialized view referencing the same tables is available locally.
- Neither the detail tables nor the materialized view can be owned by `SYS`.
- If a column or expression is present in the `GROUP BY` clause of the materialized view, it must also be present in the `SELECT` list.
- Aggregate functions must occur only as the outermost part of the expression. That is, aggregates such as `AVG(AVG(x))` or `AVG(x)+ AVG(x)` are not allowed.
- `CONNECT BY` clauses are not allowed.

> **See Also**
>
> - Advanced Query Rewrite for Materialized Views
> - *Oracle AI Database SQL Language Reference*

## 5.3.8 About Refresh Options for Materialized Views

When you define a materialized view, you can specify three refresh options: how to refresh, what type of refresh, and can trusted constraints be used. If unspecified, the defaults are assumed as `ON DEMAND`, `FORCE`, and `ENFORCED` constraints respectively.

> **Note**
>
> As of Oracle AI Database 26ai, refresh support for JSON table materialized views includes the ability to fast refresh more types of materialized views of JSON tables, as well as Query Rewrite support for these materialized views. Performance for JSON table materialized views is improved through fast refresh of both single-table and multi-table MJVs and MAVs (Materialized Aggregate Views), as well as fast refresh of sub-query materialized views that generates JSON data. In previous releases, materialized view support on JSON data is limited to MJVs (Materialized View Join Views) on a single table only. In addition, Query Rewrite support for JSON table materialized views as of Oracle AI Database 26ai provides query performance that is generally an order of magnitude faster than in previous releases.


Copyright © 2001, 2025, Oracle and/or its affiliates.

### 5.3.8.1 About Refresh Modes for Materialized Views

The refresh execution modes are `ON COMMIT` , `ON DEMAND`, and `ON STATEMENT`. Depending on the materialized view you create, some options may not be available. Table 5-3 describes the refresh modes.

**Table 5-3 Refresh Modes**

| Refresh Mode | Description |
|---|---|
| `ON COMMIT` | Refresh occurs automatically when a transaction that modified one of the materialized view's detail tables commits. This can be specified as long as the materialized view is fast refreshable (in other words, not complex). The `ON COMMIT` privilege is necessary to use this mode. |
| `ON DEMAND` | Refresh occurs when a user manually executes one of the available refresh procedures contained in the `DBMS_MVIEW` package (`REFRESH`, `REFRESH_ALL_MVIEWS`, `REFRESH_DEPENDENT`). |
| `ON STATEMENT` | Refresh occurs automatically, without the need to commit the transaction, when a DML operation is performed on any of the materialized view's base tables. This method does not require the creation of materialized view logs on materialized view's base tables. This mode can be used as long as the materialized view is fast refreshable. |

When using the `ON STATEMENT` or `ON COMMIT` method, the time to complete a DML or commit may be slightly longer than usual. This is because the refresh operation is performed as part of the DML (for `ON STATEMENT` refresh) or as part of the commit (for `ON COMMIT` refresh). Therefore, these methods may not be suitable if many users are concurrently changing the tables upon which the materialized view is based.

If you anticipate performing insert, update or delete operations on tables referenced by a materialized view concurrently with the refresh of that materialized view, and that materialized view includes joins and aggregation, Oracle recommends you use `ON COMMIT` fast refresh rather than `ON DEMAND` fast refresh.

If you think the materialized view did not refresh, check the alert log or trace file.

If a materialized view fails during refresh at DML or commit time, you must explicitly invoke the refresh procedure using the `DBMS_MVIEW` package after addressing the errors specified in the trace files. Until this is done, the materialized view will no longer be refreshed automatically at commit time.

### 5.3.8.2 About Types of Materialized View Refresh

You can specify how you want your materialized views to be refreshed from the detail tables by selecting one of four options: `COMPLETE`, `FAST`, `FORCE`, and `NEVER`. Table 5-4 describes the refresh options.

**Table 5-4 Refresh Options**

| Refresh Option | Description |
|---|---|
| `COMPLETE` | Refreshes by recalculating the materialized view's defining query. |


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Table 5-4 (Cont.) Refresh Options**

| Refresh Option | Description |
|---|---|
| FAST | Applies incremental changes to refresh the materialized view using the information logged in the materialized view logs, or from a SQL*Loader direct-path or a partition maintenance operation. |
| FORCE | Applies FAST refresh if possible; otherwise, it applies COMPLETE refresh. |
| NEVER | Indicates that the materialized view will not be refreshed with refresh mechanisms. |

Whether the fast refresh option is available depends upon the type of materialized view. You can call the procedure DBMS_MVIEW.EXPLAIN_MVIEW to determine whether fast refresh is possible.

## 5.3.8.3 About Using Trusted Constraints and Materialized View Refresh

You can also specify if it is acceptable to use trusted constraints and QUERY_REWRITE_INTEGRITY = TRUSTED during refresh. Any nonvalidated RELY constraint is a trusted constraint. For example, nonvalidated foreign key/primary key relationships, functional dependencies defined in dimensions or a materialized view in the UNKNOWN state. If query rewrite is enabled during refresh, these can improve the performance of refresh by enabling more performant query rewrites. Any materialized view that can use TRUSTED constraints for refresh is left in a state of trusted freshness (the UNKNOWN state) after refresh.

This is reflected in the column STALENESS in the view USER_MVIEWS. The column UNKNOWN_TRUSTED_FD in the same view is also set to Y, which means yes.

You can define this property of the materialized view either during create time by specifying REFRESH USING TRUSTED [ENFORCED] CONSTRAINTS or by using ALTER MATERIALIZED VIEW DDL.

**Table 5-5 Constraints**

| Constraints to Use | Description |
|---|---|
| TRUSTED CONSTRAINTS | Refresh can use trusted constraints and QUERY_REWRITE_INTEGRITY = TRUSTED during refresh.This allows use of non-validated RELY constraints and rewrite against materialized views in UNKNOWN or FRESH state during refresh.<br>The USING TRUSTED CONSTRAINTS clause enables you to create a materialized view on top of a table that has a non-NULL Virtual Private Database (VPD) policy on it. In this case, ensure that the materialized view behaves correctly. Materialized view results are computed based on the rows and columns filtered by VPD policy. Therefore, you must coordinate the materialized view definition with the VPD policy to ensure the correct results. Without the USING TRUSTED CONSTRAINTS clause, any VPD policy on a base table will prevent a materialized view from being created. |
| ENFORCED CONSTRAINTS | Refresh can use validated constraints and QUERY_REWRITE_INTEGRITY = ENFORCED during refresh. This allows use of only validated, enforced constraints and rewrite against materialized views in FRESH state during refresh. |

The fast refresh of a materialized view is optimized using the available primary and foreign key constraints on the join columns. This foreign key/primary key optimization can significantly improve refresh performance. For example, for a materialized view that contains a join between a fact table and a dimension table, if only new rows were inserted into the dimension table with no change to the fact table since the last refresh, then there will be nothing to refresh


Copyright © 2001, 2025, Oracle and/or its affiliates.

for this materialized view. The reason is that, because of the primary key constraint on the join column(s) of the dimension table and foreign key constraint on the join column(s) of the fact table, the new rows inserted into the dimension table will not join with any fact table rows, thus there is nothing to refresh. Another example of this refresh optimization is when both the fact and dimension tables have inserts since the last refresh. In this case, Oracle AI Database will only perform a join of delta fact table with the dimension table. Without the foreign key/primary key optimization, two joins during the refresh would be required, a join of delta fact with the dimension table, plus a join of delta dimension with an image of the fact table from before the inserts.

Note that this optimized fast refresh using primary and foreign key constraints on the join columns is available with and without constraint enforcement. In the first case, primary and foreign key constraints are enforced by the Oracle AI Database. This, however, incurs the cost of constraint maintenance. In the second case, the application guarantees primary and foreign key relationships so the constraints are declared `RELY NOVALIDATE` and the materialized view is defined with the `REFRESH FAST USING TRUSTED CONSTRAINTS` option.

### 5.3.8.4 General Restrictions on Fast Refresh

The defining query of the materialized view is restricted as follows:

- The materialized view must not contain references to non-repeating expressions like `SYSDATE` and `ROWNUM`.
- The materialized view must not contain references to `RAW` or `LONG RAW` data types.
- It cannot contain a `SELECT` list subquery.
- It cannot contain analytic functions (for example, `RANK`) in the `SELECT` clause.
- It cannot reference a table on which an `XMLIndex` index is defined.
- It cannot contain a `MODEL` clause.
- It cannot contain a `HAVING` clause with a subquery.
- It cannot contain nested queries that have `ANY`, `ALL`, or `NOT EXISTS`.
- It cannot contain a `[START WITH …] CONNECT BY` clause.
- It cannot contain multiple detail tables at different sites.
- `ON COMMIT` materialized views cannot have remote detail tables.
- Nested materialized views must have a join or aggregate.
- Materialized join views and materialized aggregate views with a `GROUP BY` clause cannot select from an index-organized table.
- It cannot be based on a remote view. Only complete refresh and force refresh is supported for materialized views based on remote views.

  If fast refresh is required, then create the materialized view based on the remote table on which the remote view is based.

### 5.3.8.5 Restrictions on Fast Refresh on Materialized Views with Joins Only

Defining queries for materialized views with joins only and no aggregates have the following restrictions on fast refresh:

- All restrictions from "General Restrictions on Fast Refresh".
- They cannot have `GROUP BY` clauses or aggregates.


Copyright © 2001, 2025, Oracle and/or its affiliates.

- Rowids of all the tables in the FROM list must appear in the SELECT list of the query.
- Materialized view logs must exist with rowids for all the base tables in the FROM list of the query.
- You cannot create a fast refreshable materialized view from multiple tables with simple joins that include an object type column in the SELECT statement.

Also, the refresh method you choose will not be optimally efficient if:

- The defining query uses an outer join that behaves like an inner join. If the defining query contains such a join, consider rewriting the defining query to contain an inner join.
- The SELECT list of the materialized view contains expressions on columns from multiple tables.

## 5.3.8.6 Restrictions on Fast Refresh on Materialized Views with Aggregates

Defining queries for materialized views with aggregates or joins have the following restrictions on fast refresh:

- All restrictions from "General Restrictions on Fast Refresh".

Fast refresh is supported for both ON COMMIT and ON DEMAND materialized views, however the following restrictions apply:

- All tables in the materialized view must have materialized view logs, and the materialized view logs must:
  - Contain all columns from the table referenced in the materialized view.
  - Specify with ROWID and INCLUDING NEW VALUES.
  - Specify the SEQUENCE clause if the table is expected to have a mix of inserts/direct-loads, deletes, and updates.
- Only AVG, BIT_AND_AGG, BIT_OR_AGG, BIT_XOR_AGG, COUNT, KURTOSIS_POP, KURTOSIS_SAMP, MIN, MAX, SKEWNESS_POP, SKEWNESS_SAMP, STDDEV, SUM, and VARIANCE are supported for fast refresh.
- You must specify COUNT(*).
- Aggregate functions must occur only as the outermost part of the expression. That is, aggregates such as AVG(AVG(x)) or AVG(x)+ AVG(x) are not allowed.
- For each aggregate such as AVG(expr), the corresponding COUNT(expr) must be present. Oracle recommends that you specify SUM(expr).
- If you specify VARIANCE(expr) or STDDEV(expr), you must also specify COUNT(expr) and SUM(expr). Oracle recommends that you specify SUM(expr *expr).
- If you specify KURTOSIS_POP, KURTOSIS_SAMP, SKEWNESS_POP, or SKEWNESS_SAMP, you must also specify COUNT(expr) and SUM(expr). For SKEWNESS_POP, and SKEWNESS_SAMP, you must also specify VARIANCE(expr) and COUNT(*).
- The SELECT column in the defining query cannot be a complex expression with columns from multiple base tables. A possible workaround to this is to use a nested materialized view.
- The SELECT list must contain all GROUP BY columns.
- The materialized view is not based on one or more remote tables.
- If you use a CHAR data type in the filter columns of a materialized view log, the character sets of the primary site and the materialized view must be the same.


Copyright © 2001, 2025, Oracle and/or its affiliates.

- If the materialized view has one of the following, then fast refresh is supported only on conventional DML inserts and direct loads.
  - Materialized views with `MIN` or `MAX` aggregates
  - Materialized views which have `SUM(expr)` but no `COUNT(expr)`
  - Materialized views without `COUNT(*)`

  Such a materialized view is called an insert-only materialized view.
- A materialized view with `MAX` or `MIN` is fast refreshable after delete or mixed DML statements if it does not have a `WHERE` clause.

  The max/min fast refresh after delete or mixed DML does not have the same behavior as the insert-only case. It deletes and recomputes the max/min values for the affected groups. You need to be aware of its performance impact.
- Materialized views with named views or subqueries in the `FROM` clause can be fast refreshed provided the views can be completely merged. For information on which views will merge, see *Oracle AI Database SQL Language Reference*.
- If there are no outer joins, you may have arbitrary selections and joins in the `WHERE` clause.
- Materialized aggregate views with outer joins are fast refreshable after conventional DML and direct loads, provided only the outer table has been modified. Also, unique constraints must exist on the join columns of the inner join table. If there are outer joins, all the joins must be connected by `ANDs` and must use the equality (=) operator.
- For materialized views with `CUBE`, `ROLLUP`, grouping sets, or concatenation of them, the following restrictions apply:
  - The `SELECT` list should contain grouping distinguisher that can either be a `GROUPING_ID` function on all `GROUP BY` expressions or `GROUPING` functions one for each `GROUP BY` expression. For example, if the `GROUP BY` clause of the materialized view is "`GROUP BY CUBE(a, b)`", then the `SELECT` list should contain either "`GROUPING_ID(a, b)`" or "`GROUPING(a) AND GROUPING(b)`" for the materialized view to be fast refreshable.
  - `GROUP BY` should not result in any duplicate groupings. For example, "`GROUP BY a, ROLLUP(a, b)`" is not fast refreshable because it results in duplicate groupings "`(a), (a, b), AND (a)`".

> **See Also**
>
> Requirements for Using Materialized Views with Aggregates

## 5.3.8.7 Restrictions on Fast Refresh on Materialized Views with UNION ALL

Materialized views with the `UNION ALL` set operator support the `REFRESH FAST` option if the following conditions are satisfied:

- The defining query must have the `UNION ALL` operator at the top level.

  The `UNION ALL` operator cannot be embedded inside a subquery, with one exception: The `UNION ALL` can be in a subquery in the `FROM` clause provided the defining query is of the form `SELECT * FROM` (view or subquery with `UNION ALL`) as in the following example:

```
CREATE VIEW view_with_unionall AS
(SELECT c.rowid crid, c.cust_id, 2 umarker
 FROM customers c WHERE c.cust_last_name = 'Smith'
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
 UNION ALL
 SELECT c.rowid crid, c.cust_id, 3 umarker
 FROM customers c WHERE c.cust_last_name = 'Jones');

CREATE MATERIALIZED VIEW unionall_inside_view_mv
REFRESH FAST ON DEMAND AS
SELECT * FROM view_with_unionall;
```

Note that the view `view_with_unionall` satisfies the requirements for fast refresh.

- Each query block in the `UNION ALL` query must satisfy the requirements of a fast refreshable materialized view with aggregates or a fast refreshable materialized view with joins.

  The appropriate materialized view logs must be created on the tables as required for the corresponding type of fast refreshable materialized view.

  Note that the Oracle AI Database also allows the special case of a single table materialized view with joins only provided the `ROWID` column has been included in the `SELECT` list and in the materialized view log. This is shown in the defining query of the view `view_with_unionall`.

- The `SELECT` list of each query must include a `UNION ALL` marker, and the `UNION ALL` column must have a distinct constant numeric or string value in each `UNION ALL` branch. Further, the marker column must appear in the same ordinal position in the `SELECT` list of each query block. See "UNION ALL Marker and Query Rewrite" for more information regarding `UNION ALL` markers.

- Some features such as outer joins, insert-only aggregate materialized view queries and remote tables are not supported for materialized views with `UNION ALL`. Note, however, that materialized views used in replication, which do not contain joins or aggregates, can be fast refreshed when `UNION ALL` or remote tables are used.

- The compatibility initialization parameter must be set to 9.2.0 or higher to create a fast refreshable materialized view with `UNION ALL`.

## 5.3.8.8 About Achieving Refresh Goals

In addition to the `EXPLAIN_MVIEW` procedure, which is discussed throughout this chapter, you can use the `DBMS_ADVISOR.TUNE_MVIEW` procedure to optimize a `CREATE MATERIALIZED VIEW` statement to achieve `REFRESH FAST` and `ENABLE QUERY REWRITE` goals.

See Refreshing Materialized Views on Prebuilt Tables.

### 5.3.8.8.1 Refreshing Materialized Views on Prebuilt Tables

For materialized views created with the prebuilt option, the index `I_snap$` is not created by default. This index helps fast refresh performance, and a description of how to create this index is illustrated in "Choosing Indexes for Materialized Views".

## 5.3.8.9 Refreshing Nested Materialized Views

A nested materialized view is considered to be fresh as long as its data is synchronized with the data in its detail tables, even if some of its detail tables could be stale materialized views.

You can refresh nested materialized views in two ways: `DBMS_MVIEW.REFRESH` with the `nested` flag set to `TRUE` and `REFRESH_DEPENDENT` with the `nested` flag set to `TRUE` on the base tables. If you use `DBMS_MVIEW.REFRESH`, the entire materialized view chain is refreshed and the coverage starting from the specified materialized view in top-down fashion. That is, the specified


Copyright © 2001, 2025, Oracle and/or its affiliates.

materialized view and all its child materialized views in the dependency hierarchy are refreshed in order. With DBMS_MVIEW.REFRESH_DEPENDENT, the entire chain is refreshed from the bottom up. That is, all the parent materialized views in the dependency hierarchy starting from the specified table are refreshed in order.

**Example 5-8 Example of Refreshing a Nested Materialized View**

The following statement shows an example of refreshing a nested materialized view:

```
DBMS_MVIEW.REFRESH('SALES_MV,COST_MV', nested => TRUE);
```

This statement will first refresh all child materialized views of sales_mv and cost_mv based on the dependency analysis and then refresh the two specified materialized views.

You can query the STALE_SINCE column in the *_MVIEWS views to find out when a materialized view became stale.

## 5.3.9 ORDER BY Clause in Materialized Views

An ORDER BY clause is allowed in the CREATE MATERIALIZED VIEW statement. It is used only during the initial creation of the materialized view. It is not used during a full refresh or a fast refresh.

To improve the performance of queries against large materialized views, store the rows in the materialized view in the order specified in the ORDER BY clause. This initial ordering provides physical clustering of the data. If indexes are built on the columns by which the materialized view is ordered, accessing the rows of the materialized view using the index often reduces the time for disk I/O due to the physical clustering.

The ORDER BY clause is not considered part of the materialized view definition. As a result, there is no difference in the manner in which Oracle AI Database detects the various types of materialized views (for example, materialized join views with no aggregates). For the same reason, query rewrite is not affected by the ORDER BY clause. This feature is similar to the CREATE TABLE ... ORDER BY capability.

## 5.3.10 Using Oracle Enterprise Manager to Create Materialized Views

A materialized view can also be created using Enterprise Manager by selecting the materialized view object type. There is no difference in the information required if this approach is used.

## 5.3.11 Using Materialized Views with NLS Parameters

When using certain materialized views, you must ensure that your NLS parameters are the same as when you created the materialized view. Materialized views with this restriction are as follows:

- Expressions that may return different values, depending on NLS parameter settings. For example, (date > "01/02/03") or (rate <= "2.150") are NLS parameter dependent expressions.
- Equijoins where one side of the join is character data. The result of this equijoin depends on collation and this can change on a session basis, giving an incorrect result in the case of query rewrite or an inconsistent materialized view after a refresh operation.
- Expressions that generate internal conversion to character data in the SELECT list of a materialized view, or inside an aggregate of a materialized aggregate view. This restriction


Copyright © 2001, 2025, Oracle and/or its affiliates.

does not apply to expressions that involve only numeric data, for example, `a+b` where `a` and `b` are numeric fields.

## 5.3.12 Adding Comments to Materialized Views

You can add comments to materialized views.

**Example: Adding Comments to a Materialized View**

The following statement adds a comment to data dictionary views for an existing materialized view:

```
COMMENT ON MATERIALIZED VIEW sales_mv IS 'sales materialized view';
```

To view the comment after the preceding statement execution, you can query the catalog views, {USER, DBA} ALL_MVIEW_COMMENTS. For example, consider the following example:

```
SELECT MVIEW_NAME, COMMENTS
FROM USER_MVIEW_COMMENTS WHERE MVIEW_NAME = 'SALES_MV';
```

The output will resemble the following:

```
MVIEW_NAME                     COMMENTS
-----------      ----------------------
SALES_MV         sales materialized view
```

Note: If the compatibility is set to 10.0.1 or higher, `COMMENT ON TABLE` will not be allowed for the materialized view container table. The following error message will be thrown if it is issued.

```
ORA-12098: cannot comment on the materialized view.
```

In the case of a prebuilt table, if it has an existing comment, the comment will be inherited by the materialized view after it has been created. The existing comment will be prefixed with `'(from table)'`. For example, table `sales_summary` was created to contain sales summary information. An existing comment `'Sales summary data'` was associated with the table. A materialized view of the same name is created to use the prebuilt table as its container table. After the materialized view creation, the comment becomes `'(from table) Sales summary data'`.

However, if the prebuilt table, `sales_summary`, does not have any comment, the following comment is added: `'Sales summary data'`. Then, if you drop the materialized view, the comment will be passed to the prebuilt table with the comment: `'(from materialized view) Sales summary data'`.

# 5.4 Creating Materialized View Logs

Materialized view logs are required if you want to use fast refresh, with the exception of partition change tracking refresh. That is, if a detail table supports partition change tracking for a materialized view, the materialized view log on that detail table is not required in order to do fast refresh on that materialized view. As a general rule, though, you should create materialized view logs if you want to use fast refresh. Materialized view logs are defined using a `CREATE MATERIALIZED VIEW LOG` statement on the base table that is to be changed. They are not created on the materialized view unless there is another materialized view on top of that materialized view, which is the case with nested materialized views. For fast refresh of materialized views, the definition of the materialized view logs must normally specify the `ROWID` clause. In addition, for aggregate materialized views, it must also contain every column in the table referenced in the materialized view, the `INCLUDING NEW VALUES` clause and the `SEQUENCE`


Copyright © 2001, 2025, Oracle and/or its affiliates.

clause. You can typically achieve better fast refresh performance of local materialized views containing aggregates or joins by using a WITH COMMIT SCN clause.

An example of a materialized view log is shown as follows where one is created on the table sales:

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID
(prod_id, cust_id, time_id, channel_id, promo_id, quantity_sold, amount_sold)
INCLUDING NEW VALUES;
```

Alternatively, you could create a commit SCN-based materialized view log as follows:

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID
 (prod_id, cust_id, time_id, channel_id, promo_id, quantity_sold, amount_sold),
COMMIT SCN INCLUDING NEW VALUES;
```

Oracle recommends that the keyword SEQUENCE be included in your materialized view log statement unless you are sure that you will never perform a mixed DML operation (a combination of INSERT, UPDATE, or DELETE operations on multiple tables). The SEQUENCE column is required in the materialized view log to support fast refresh with a combination of INSERT, UPDATE, or DELETE statements on multiple tables. You can, however, add the SEQUENCE number to the materialized view log after it has been created.

The boundary of a mixed DML operation is determined by whether the materialized view is ON COMMIT or ON DEMAND.

- For ON COMMIT, the mixed DML statements occur within the same transaction because the refresh of the materialized view will occur upon commit of this transaction.
- For ON DEMAND, the mixed DML statements occur between refreshes. The following example of a materialized view log illustrates where one is created on the table sales that includes the SEQUENCE keyword:

  ```
  CREATE MATERIALIZED VIEW LOG ON sales WITH SEQUENCE, ROWID
  (prod_id, cust_id, time_id, channel_id, promo_id,
   quantity_sold, amount_sold) INCLUDING NEW VALUES;
  ```

This section contains the following topics:

- Using the FORCE Option With Materialized View Logs
- Purging Materialized View Logs

## 5.4.1 Using the FORCE Option With Materialized View Logs

If you specify FORCE and any items specified with the ADD clause have already been specified for the materialized view log, Oracle does not return an error, but silently ignores the existing elements and adds to the materialized view log any items that do not already exist in the log. For example, if you used a filter column such as cust_id and this column already existed, Oracle AI Database ignores the redundancy and does not return an error.

## 5.4.2 Purging Materialized View Logs

Purging materialized view logs can be done during the materialized view refresh process or deferred until later, thus improving refresh performance time. You can choose different options for when the purge will occur, using a PURGE clause, as in the following:

```
CREATE MATERIALIZED VIEW LOG ON sales
PURGE START WITH sysdate NEXT sysdate+1
WITH ROWID
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
 (prod_id, cust_id, time_id, channel_id, promo_id, quantity_sold, amount_sold)
INCLUDING NEW VALUES;
```

You can also query `USER_MVIEW_LOGS` for purge information, as in the following:

```
SELECT PURGE_DEFERRED, PURGE_INTERVAL, LAST_PURGE_DATE, LAST_PURGE_STATUS
FROM USER_MVIEW_LOGS
WHERE LOG_OWNER "SH" AND PRIMARY = 'SALES';
```

In addition to setting the purge when creating a materialized view log, you can also modify an existing materialized view log by issuing a statement resembling the following:

```
ALTER MATERIALIZED VIEW LOG ON sales PURGE IMMEDIATE;
```

> **See Also**
>
> *Oracle AI Database SQL Language Reference* for more information regarding materialized view log syntax

# 5.5 Creating Materialized Views Based on Approximate Queries

A materialized view based on approximate queries uses SQL functions that return approximate functions in its defining query.

You can compute summary and aggregate approximations and store these results in materialized views for further analysis or querying. The summary approximation, which computes approximate aggregates for all dimensions within a group of rows, can be used to perform detailed aggregation. You can further aggregate the summary data to obtain aggregate approximations that can be used for high-level analysis so that the Oracle AI Database does not scan the base tables again to compute higher-level aggregates. Oracle AI Database does not scan the base tables again to compute higher-level aggregates. It just uses the existing aggregated results to compute the higher-level aggregates. For example, you can create a summary approximation that stores the approximate number of products sold within each state and within each country. This aggregate approximation is then used to return the approximate distinct number of products within each country.

To create a materialized view containing SQL functions that return approximate results:

- Run the `CREATE MATERIALIZED VIEW` statement, with the defining query containing the appropriate functions

  For example, use the `APPROX_PERCENTILE` function in the defining query of the materialized view.

**Example 5-9 Creating a Materialized View Based on Approximate Queries**

The following example creates a materialized view that stores the approximate number of distinct products that are sold on each day.

```
CREATE MATERIALIZED VIEW approx_count_distinct_pdt_mv
ENABLE QUERY REWRITE AS
SELECT t.calendar_year, t.calendar_month_number, t.day_number_in_month,
approx_count_distinct(prod_id) daily_detail
FROM sales s, times t
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
WHERE s.time_id = t.time_id
GROUP BY t.calendar_year, t.calendar_month_number, t.day_number_in_month;
```

**See Also**

- Refreshing Materialized Views Based on Approximate Queries
- Using Percentile Functions that Return Approximate Results

# 5.6 Creating a Materialized View Containing Bitmap-based COUNT(DISTINCT) Functions

Materialized views based on `COUNT(DISTINCT)` functions can provide enhanced performance by using bitmap-based operations on integer columns.

Starting with Oracle Database Release 19c, you can create materialized views based on SQL aggregate functions that use bitmap representation to express the computation of `COUNT(DISTINCT)` operations. These functions include `BITMAP_BUCKET_NUMBER`, `BITMAP_BIT_POSITION` and `BITMAP_CONSTRUCT_AGG`.

To create a materialized view based on bitmaps:

1. Ensure that materialized view logs exist for the tables on which the materialized view will be based.
2. Use the `CREATE MATERIALIZED VIEW` command to create the materialized view.

   The following example creates a materialized view based on the `SH.SALES` table and containing non-additive facts.

```
SQL> create materialized view mv_sales as
  2  select PROMO_ID,
  3  BITMAP_BUCKET_NUMBER(PROD_ID) bm_bktno,
  4  BITMAP_CONSTRUCT_AGG(BITMAP_BIT_POSITION(PROD_ID),'RAW') bm_details
  5  from sales
  6  group by PROMO_ID,BITMAP_BUCKET_NUMBER(PROD_ID);

Materialized view created.
```

**Related Topics**

- Query Rewrite and Materialized Views Based on Bitmap-based COUNT(DISTINCT) Functions
  Queries that contain `COUNT(DISTINCT)` operations on integer columns can be rewritten to use materialized views that contain bitmap-based functions.

# 5.7 Registering Existing Materialized Views

Some data warehouses have implemented materialized views in ordinary user tables. Although this solution provides the performance benefits of materialized views, it does not:

- Provide query rewrite to all SQL applications.


Copyright © 2001, 2025, Oracle and/or its affiliates.

- Enable materialized views defined in one application to be transparently accessed in another application.
- Generally support fast parallel or fast materialized view refresh.

Because of these limitations, and because existing materialized views can be extremely large and expensive to rebuild, you should register your existing materialized view tables whenever possible. You can register a user-defined materialized view with the `CREATE MATERIALIZED VIEW ... ON PREBUILT TABLE` statement. Once registered, the materialized view can be used for query rewrites or maintained by one of the refresh methods, or both.

The contents of the table must reflect the materialization of the defining query at the time you register it as a materialized view, and each column in the defining query must correspond to a column in the table that has a matching data type. However, you can specify `WITH REDUCED PRECISION` to allow the precision of columns in the defining query to be different from that of the table columns.

The table and the materialized view must have the same name, but the table retains its identity as a table and can contain columns that are not referenced in the defining query of the materialized view. These extra columns are known as unmanaged columns. If rows are inserted during a refresh operation, each unmanaged column of the row is set to its default value. Therefore, the unmanaged columns cannot have `NOT NULL` constraints unless they also have default values.

Materialized views based on prebuilt tables are eligible for selection by query rewrite provided the parameter `QUERY_REWRITE_INTEGRITY` is set to `STALE_TOLERATED` or `TRUSTED`.

> **See Also**
>
> Basic Query Rewrite for Materialized Views for details about integrity levels

When you drop a materialized view that was created on a prebuilt table, the table still exists—only the materialized view is dropped.

The following example illustrates the two steps required to register a user-defined table. First, the table is created, then the materialized view is defined using exactly the same name as the table. This materialized view `sum_sales_tab_mv` is eligible for use in query rewrite.

```
CREATE TABLE sum_sales_tab
PCTFREE 0  TABLESPACE demo
STORAGE (INITIAL 8M) AS
SELECT s.prod_id, SUM(amount_sold) AS dollar_sales,
       SUM(quantity_sold) AS unit_sales
FROM sales s GROUP BY s.prod_id;

CREATE MATERIALIZED VIEW sum_sales_tab_mv
ON PREBUILT TABLE WITHOUT REDUCED PRECISION
ENABLE QUERY REWRITE AS
SELECT s.prod_id, SUM(amount_sold) AS dollar_sales,
    SUM(quantity_sold) AS unit_sales
FROM sales s GROUP BY s.prod_id;
```

You could have compressed this table to save space.

In some cases, user-defined materialized views are refreshed on a schedule that is longer than the update cycle. For example, a monthly materialized view might be updated only at the end of each month, and the materialized view values always refer to complete time periods. Reports written directly against these materialized views implicitly select only data that is not in


Copyright © 2001, 2025, Oracle and/or its affiliates.

the current (incomplete) time period. If a user-defined materialized view already contains a time dimension:

- It should be registered and then fast refreshed each update cycle.
- You can create a view that selects the complete time period of interest.
- The reports should be modified to refer to the view instead of referring directly to the user-defined materialized view.

If the user-defined materialized view does not contain a time dimension, then you should create a new materialized view that does include the time dimension (if possible). Also, in this case, the view should aggregate over the time column in the new materialized view.

## 5.8 Choosing Indexes for Materialized Views

The two most common operations on a materialized view are query execution and fast refresh, and each operation has different performance requirements. Query execution might need to access any subset of the materialized view key columns, and might need to join and aggregate over a subset of those columns. Consequently, query execution usually performs best if a single-column bitmap index is defined on each materialized view key column.

In the case of materialized views containing only joins using fast refresh, Oracle recommends that indexes be created on the columns that contain the rowids to improve the performance of the refresh operation.

If a materialized view using aggregates is fast refreshable, then an index appropriate for the fast refresh procedure is created unless `USING NO INDEX` is specified in the `CREATE MATERIALIZED VIEW` statement.

If the materialized view is partitioned, then, after doing a partition maintenance operation on the materialized view, the indexes become unusable, and they need to be rebuilt for fast refresh to work.

If you create a materialized view with the prebuilt option, the `I_snap$` index is not automatically created. This index significantly improves fast refresh performance, and you can create it manually by issuing a statement such as the following:

```
CREATE UNIQUE INDEX <OWNER>."I_SNAP$_<MVIEW_NAME>" ON <OWNER>.<MVIEW_NAME>
 (SYS_OP_MAP_NONNULL("LOG_DATE"))
  PCTFREE 10 INITRANS 2 MAXTRANS 255 COMPUTE STATISTICS
  STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
  PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT FLASH_CACHE DE
FAULT CELL_FLASH_CACHE DEFAULT)
  TABLESPACE <TABLESPACE_NAME>;
```

> **See Also**
>
> *Oracle AI Database SQL Tuning Guide* for information on using the SQL Access Advisor to determine what indexes are appropriate for your materialized view

## 5.9 Dropping Materialized Views

Use the `DROP MATERIALIZED VIEW` statement to drop a materialized view. For example, consider the following statement:

```
DROP MATERIALIZED VIEW sales_sum_mv;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

This statement drops the materialized view `sales_sum_mv`. If the materialized view was prebuilt on a table, then the table is not dropped, but it can no longer be maintained with the refresh mechanism or used by query rewrite. Alternatively, you can drop a materialized view using Oracle Enterprise Manager.

# 5.10 Analyzing Materialized View Capabilities

You can use the `DBMS_MVIEW.EXPLAIN_MVIEW` procedure to learn what is possible with a materialized view or potential materialized view. In particular, this procedure enables you to determine:

- If a materialized view is fast refreshable
- What types of query rewrite you can perform with this materialized view
- Whether partition change tracking refresh is possible

Using this procedure is straightforward and described in "Using the DBMS_MVIEW.EXPLAIN_MVIEW Procedure". You simply call `DBMS_MVIEW.EXPLAIN_MVIEW`, passing in as a single parameter the schema and materialized view name for an existing materialized view. Alternatively, you can specify the `SELECT` string for a potential materialized view or the complete `CREATE MATERIALIZED VIEW` statement. The materialized view or potential materialized view is then analyzed and the results are written into either a table called `MV_CAPABILITIES_TABLE`, which is the default, or to an array called `MSG_ARRAY`.

Note that you must run the `utlxmv.sql` script prior to calling `EXPLAIN_MVIEW` except when you are placing the results in `MSG_ARRAY`. The script is found in the `admin` directory. It is to create the `MV_CAPABILITIES_TABLE` in the current schema. An explanation of the various capabilities is in Table 5-6, and all the possible messages are listed in Table 5-7.

## 5.10.1 Using the DBMS_MVIEW.EXPLAIN_MVIEW Procedure

The `EXPLAIN_MVIEW` procedure has the following parameters:

- `stmt_id`

  An optional parameter. A client-supplied unique identifier to associate output rows with specific invocations of `EXPLAIN_MVIEW`.

- `mv`

  The name of an existing materialized view or the query definition or the entire `CREATE MATERIALIZED VIEW` statement of a potential materialized view you want to analyze.

- `msg-array`

  The PL/SQL `VARRAY` that receives the output.

`EXPLAIN_MVIEW` analyzes the specified materialized view in terms of its refresh and rewrite capabilities and inserts its results (in the form of multiple rows) into `MV_CAPABILITIES_TABLE` or `MSG_ARRAY`.

> **See Also**
>
> *Oracle AI Database PL/SQL Packages and Types Reference* for further information about the `DBMS_MVIEW` package


Copyright © 2001, 2025, Oracle and/or its affiliates.

This section contains the following topics:


- DBMS_MVIEW.EXPLAIN_MVIEW Declarations
- Using MV_CAPABILITIES_TABLE
- MV_CAPABILITIES_TABLE.CAPABILITY_NAME Details
- MV_CAPABILITIES_TABLE Column Details


## 5.10.1.1 DBMS_MVIEW.EXPLAIN_MVIEW Declarations

The following PL/SQL declarations that are made for you in the `DBMS_MVIEW` package show the order and data types of these parameters for explaining an existing materialized view and a potential materialized view with output to a table and to a `VARRAY`.

Explain an existing or potential materialized view with output to `MV_CAPABILITIES_TABLE`:

```
DBMS_MVIEW.EXPLAIN_MVIEW (mv           IN VARCHAR2,
                          stmt_id IN VARCHAR2:= NULL);
```

Explain an existing or potential materialized view with output to a `VARRAY`:

```
DBMS_MVIEW.EXPLAIN_MVIEW (mv          IN VARCHAR2,
                          msg_array   OUT SYS.ExplainMVArrayType);
```

## 5.10.1.2 Using MV_CAPABILITIES_TABLE

One of the simplest ways to use `DBMS_MVIEW.EXPLAIN_MVIEW` is with the `MV_CAPABILITIES_TABLE`, which has the following structure:

```
CREATE TABLE MV_CAPABILITIES_TABLE
(STATEMENT_ID      VARCHAR(30),   -- Client-supplied unique statement identifier
 MVOWNER           VARCHAR(30),   -- NULL for SELECT based EXPLAIN_MVIEW
 MVNAME            VARCHAR(30),   -- NULL for SELECT based EXPLAIN_MVIEW
 CAPABILITY_NAME   VARCHAR(30),   -- A descriptive name of the particular
                                  -- capability:
                                  -- REWRITE
                                  --   Can do at least full text match
                                  --   rewrite
                                  -- REWRITE_PARTIAL_TEXT_MATCH
                                  --   Can do at least full and partial
                                  --   text match rewrite
                                  -- REWRITE_GENERAL
                                  --   Can do all forms of rewrite
                                  -- REFRESH
                                  --   Can do at least complete refresh
                                  -- REFRESH_FROM_LOG_AFTER_INSERT
                                  --   Can do fast refresh from an mv log
                                  --   or change capture table at least
                                  --   when update operations are
                                  --   restricted to INSERT
                                  -- REFRESH_FROM_LOG_AFTER_ANY
                                  --   can do fast refresh from an mv log
                                  --   or change capture table after any
                                  --   combination of updates
                                  -- PCT
                                  --   Can do Enhanced Update Tracking on
                                  --   the table named in the RELATED_NAME
                                  --   column.  EUT is needed for fast
                                  --   refresh after partitioned
                                  --   maintenance operations on the table
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
                                -- named in the RELATED_NAME column
                                -- and to do non-stale tolerated
                                -- rewrite when the mv is partially
                                -- stale with respect to the table
                                -- named in the RELATED_NAME column.
                                -- EUT can also sometimes enable fast
                                -- refresh of updates to the table
                                -- named in the RELATED_NAME column
                                -- when fast refresh from an mv log
                                -- or change capture table is not
                                -- possible.
                                -- See Table 5-6
POSSIBLE          CHARACTER(1), -- T = capability is possible
                                -- F = capability is not possible
RELATED_TEXT      VARCHAR(2000), -- Owner.table.column, alias name, and so on
                                -- related to this message. The specific
                                -- meaning of this column depends on the
                                -- NSGNO column. See the documentation for
                                -- DBMS_MVIEW.EXPLAIN_MVIEW() for details.
RELATED_NUM       NUMBER,       -- When there is a numeric value
                                -- associated with a row, it goes here.
MSGNO             INTEGER,      -- When available, QSM message # explaining
                                -- why disabled or more details when
                                -- enabled.
MSGTXT            VARCHAR(2000), -- Text associated with MSGNO.
SEQ               NUMBER);      -- Useful in ORDER BY clause when
                                -- selecting from this table.
```

You can use the `utlxmv.sql` script found in the `admin` directory to create `MV_CAPABILITIES_TABLE`.

> **See Also**
>
> - Refreshing Materialized Views for further details about partition change tracking
> - Advanced Query Rewrite for Materialized Views for further details about partition change tracking

### Example 5-10 DBMS_MVIEW.EXPLAIN_MVIEW

First, create the materialized view. Alternatively, you can use `EXPLAIN_MVIEW` on a potential materialized view using its `SELECT` statement or the complete `CREATE MATERIALIZED VIEW` statement.

```
CREATE MATERIALIZED VIEW cal_month_sales_mv
BUILD IMMEDIATE
REFRESH FORCE
ENABLE QUERY REWRITE AS
SELECT t.calendar_month_desc,  SUM(s.amount_sold) AS dollars
FROM sales s,  times t WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

Then, you invoke `EXPLAIN_MVIEW` with the materialized view to explain. You need to use the `SEQ` column in an `ORDER BY` clause so the rows will display in a logical order. If a capability is not possible, `N` will appear in the `P` column and an explanation in the `MSGTXT` column. If a capability is not possible for multiple reasons, a row is displayed for each reason.

```
EXECUTE DBMS_MVIEW.EXPLAIN_MVIEW ('SH.CAL_MONTH_SALES_MV');
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
SELECT capability_name,  possible, SUBSTR(related_text,1,8)
  AS rel_text, SUBSTR(msgtxt,1,60) AS msgtxt
FROM MV_CAPABILITIES_TABLE
ORDER BY seq;


CAPABILITY_NAME                P    REL_TEXT     MSGTXT
---------------                -    --------     ------
PCT                            N
REFRESH_COMPLETE               Y
REFRESH_FAST                   N
REWRITE                        Y
PCT_TABLE                      N    SALES        no partition key or PMARKER in select list
PCT_TABLE                      N    TIMES        relation is not a partitioned table
REFRESH_FAST_AFTER_INSERT      N    SH.TIMES     mv log must have new values
REFRESH_FAST_AFTER_INSERT      N    SH.TIMES     mv log must have ROWID
REFRESH_FAST_AFTER_INSERT      N    SH.TIMES     mv log does not have all necessary columns
REFRESH_FAST_AFTER_INSERT      N    SH.SALES     mv log must have new values
REFRESH_FAST_AFTER_INSERT      N    SH.SALES     mv log must have ROWID
REFRESH_FAST_AFTER_INSERT      N    SH.SALES     mv log does not have all necessary columns
REFRESH_FAST_AFTER_ONETAB_DML  N    DOLLARS      SUM(expr) without COUNT(expr)
REFRESH_FAST_AFTER_ONETAB_DML  N                 see the reason why
                                                 REFRESH_FAST_AFTER_INSERT is disabled
REFRESH_FAST_AFTER_ONETAB_DML  N                 COUNT(*) is not present in the select list
REFRESH_FAST_AFTER_ONETAB_DML  N                 SUM(expr) without COUNT(expr)
REFRESH_FAST_AFTER_ANY_DML     N                 see the reason why
                                                 REFRESH_FAST_AFTER_ONETAB_DML is disabled
REFRESH_FAST_AFTER_ANY_DML     N    SH.TIMES     mv log must have sequence
REFRESH_FAST_AFTER_ANY_DML     N    SH.SALES     mv log must have sequence
REFRESH_PCT                    N                 PCT is not possible on any of the detail
                                                 tables in the materialized view
REWRITE_FULL_TEXT_MATCH        Y
REWRITE_PARTIAL_TEXT_MATCH     Y
REWRITE_GENERAL                Y
REWRITE_PCT                    N                 PCT is not possible on any detail tables
```

## 5.10.1.3 MV_CAPABILITIES_TABLE.CAPABILITY_NAME Details

Table 5-6 lists explanations for values in the CAPABILITY_NAME column.

**Table 5-6 CAPABILITY_NAME Column Details**

| CAPABILITY_NAME | Description |
|---|---|
| PCT | If this capability is possible, partition change tracking is possible on at least one detail relation. If this capability is not possible, partition change tracking is not possible with any detail relation referenced by the materialized view. |
| REFRESH_COMPLETE | If this capability is possible, complete refresh of the materialized view is possible. |
| REFRESH_FAST | If this capability is possible, fast refresh is possible at least under certain circumstances. |
| REWRITE | If this capability is possible, at least full text match query rewrite is possible. If this capability is not possible, no form of query rewrite is possible. |


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Table 5-6 (Cont.) CAPABILITY_NAME Column Details**

| CAPABILITY_NAME | Description |
|---|---|
| PCT_TABLE | If this capability is possible, it is possible with respect to a particular partitioned table in the top level FROM list. When possible, partition change tracking (PCT) applies to the partitioned table named in the RELATED_TEXT column.<br>PCT is needed to support fast refresh after partition maintenance operations on the table named in the RELATED_TEXT column.<br>PCT may also support fast refresh with regard to updates to the table named in the RELATED_TEXT column when fast refresh from a materialized view log is not possible.<br>PCT is also needed to support query rewrite in the presence of partial staleness of the materialized view with regard to the table named in the RELATED_TEXT column.<br>When disabled, PCT does not apply to the table named in the RELATED_TEXT column. In this case, fast refresh is not possible after partition maintenance operations on the table named in the RELATED_TEXT column. In addition, PCT-based refresh of updates to the table named in the RELATED_TEXT column is not possible. Finally, query rewrite cannot be supported in the presence of partial staleness of the materialized view with regard to the table named in the RELATED_TEXT column. |
| PCT_TABLE_REWRITE | If this capability is possible, it is possible with respect to a particular partitioned table in the top level FROM list. When possible, PCT applies to the partitioned table named in the RELATED_TEXT column.<br>This capability is needed to support query rewrite against this materialized view in partial stale state with regard to the table named in the RELATED_TEXT column.<br>When disabled, query rewrite cannot be supported if this materialized view is in partial stale state with regard to the table named in the RELATED_TEXT column. |
| REFRESH_FAST_AFTER_INSERT | If this capability is possible, fast refresh from a materialized view log is possible at least in the case where the updates are restricted to INSERT operations; complete refresh is also possible. If this capability is not possible, no form of fast refresh from a materialized view log is possible. |
| REFRESH_FAST_AFTER_ONETAB_DML | If this capability is possible, fast refresh from a materialized view log is possible regardless of the type of update operation, provided all update operations are performed on a single table. If this capability is not possible, fast refresh from a materialized view log may not be possible when the update operations are performed on multiple tables. |
| REFRESH_FAST_AFTER_ANY_DML | If this capability is possible, fast refresh from a materialized view log is possible regardless of the type of update operation or the number of tables updated. If this capability is not possible, fast refresh from a materialized view log may not be possible when the update operations (other than INSERT) affect multiple tables. |
| REFRESH_FAST_PCT | If this capability is possible, fast refresh using PCT is possible. Generally, this means that refresh is possible after partition maintenance operations on those detail tables where PCT is indicated as possible. |
| REWRITE_FULL_TEXT_MATCH | If this capability is possible, full text match query rewrite is possible. If this capability is not possible, full text match query rewrite is not possible. |
| REWRITE_PARTIAL_TEXT_MATCH | If this capability is possible, at least full and partial text match query rewrite are possible. If this capability is not possible, at least partial text match query rewrite and general query rewrite are not possible. |
| REWRITE_GENERAL | If this capability is possible, all query rewrite capabilities are possible, including general query rewrite and full and partial text match query rewrite. If this capability is not possible, at least general query rewrite is not possible. |
| REWRITE_PCT | If this capability is possible, query rewrite can use a partially stale materialized view even in QUERY_REWRITE_INTEGRITY = ENFORCED or TRUSTED modes. When this capability is not possible, query rewrite can use a partially stale materialized view only in QUERY_REWRITE_INTEGRITY = STALE_TOLERATED mode. |


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 5.10.1.4 MV_CAPABILITIES_TABLE Column Details

Table 5-7 lists the semantics for RELATED_TEXT and RELATED_NUM columns.

**Table 5-7 MV_CAPABILITIES_TABLE Column Details**

| MSGNO | MSGTXT | RELATED_NUM | RELATED_TEXT |
|---|---|---|---|
| NULL | NULL | | For PCT capability only: [*owner.*]*name* of the table upon which PCT is enabled |
| 2066 | This statement resulted in an Oracle error | Oracle error number that occurred | |
| 2067 | No partition key or PMARKER or join dependent expression in SELECT list | | [*owner.*]*name* of relation for which PCT is not supported |
| 2068 | Relation is not partitioned | | [*owner.*]*name* of relation for which PCT is not supported |
| 2069 | PCT not supported with multicolumn partition key | | [*owner.*]*name* of relation for which PCT is not supported |
| 2070 | PCT not supported with this type of partitioning | | [*owner.*]*name* of relation for which PCT is not supported |
| 2071 | Internal error: undefined PCT failure code | The unrecognized numeric PCT failure code | [*owner.*]*name* of relation for which PCT is not supported |
| 2072 | Requirements not satisfied for fast refresh of nested materialized view | | |
| 2077 | Materialized view log is newer than last full refresh | | [*owner.*]*table_name* of table upon which the materialized view log is needed |
| 2078 | Materialized view log must have new values | | [*owner.*]*table_name* of table upon which the materialized view log is needed |
| 2079 | Materialized view log must have ROWID | | [*owner.*]*table_name* of table upon which the materialized view log is needed |
| 2080 | Materialized view log must have primary key | | [*owner.*]*table_name* of table upon which the materialized view log is needed |
| 2081 | Materialized view log does not have all necessary columns | | [*owner.*]*table_name* of table upon which the materialized view log is needed |
| 2082 | Problem with materialized view log | | [*owner.*]*table_name* of table upon which the materialized view log is needed |
| 2099 | Materialized view references a remote table or view in the FROM list | Offset from the SELECT keyword to the table or view in question | [*owner.*]*name* of the table or view in question |
| 2126 | Multiple primary sites | | Name of the first different node, or NULL if the first different node is local |
| 2129 | Join or filter condition(s) are complex | | [*owner.*]*name* of the table involved with the join or filter condition (or NULL when not available) |
| 2130 | Expression not supported for fast refresh | Offset from the SELECT keyword to the expression in question | The alias name in the SELECT list of the expression in question |


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Table 5-7 (Cont.) MV_CAPABILITIES_TABLE Column Details**

| MSGNO | MSGTXT | RELATED_NUM | RELATED_TEXT |
|---|---|---|---|
| 2150 | SELECT lists must be identical across the UNION operator | Offset from the SELECT keyword to the first different select item in the SELECT list | The alias name of the first different select item in the SELECT list |
| 2182 | PCT is enabled through a join dependency | | [*owner*.]*name* of relation for which PCT_TABLE_REWRITE is not enabled |
| 2183 | Expression to enable PCT not in PARTITION BY of analytic function or model | The unrecognized numeric PCT failure code | [*owner*.]*name* of relation for which PCT is not enabled |
| 2184 | Expression to enable PCT cannot be rolled up | | [*owner*.]*name* of relation for which PCT is not enabled |
| 2185 | No partition key or PMARKER in the SELECT list | | [*owner*.]*name* of relation for which PCT_TABLE_REWRITE is not enabled |
| 2186 | GROUP OUTER JOIN is present | | |
| 2187 | Materialized view on external table | | |


Copyright © 2001, 2025, Oracle and/or its affiliates.

# 6
# Advanced Materialized Views

This chapter discusses advanced topics in using materialized views. It contains the following topics:


- About Partitioning and Materialized Views
- About Materialized Views in Analytic Processing Environments
- About Materialized Views and Models
- About Security Issues with Materialized Views
- Invalidating Materialized Views
- Altering Materialized Views
- Using Real-time Materialized Views


## 6.1 About Partitioning and Materialized Views

Because of the large volume of data held in a data warehouse, partitioning is an extremely useful option when designing a database. Partitioning the fact tables improves scalability, simplifies system administration, and makes it possible to define local indexes that can be efficiently rebuilt. Partitioning the fact tables also improves the opportunity of fast refreshing the materialized view because this may enable partition change tracking (PCT) refresh on the materialized view. Partitioning a materialized view also has benefits for refresh, because the refresh procedure can then use parallel DML in more scenarios and PCT-based refresh can use truncate partition to efficiently maintain the materialized view.

> **See Also**
>
> *Oracle AI Database VLDB and Partitioning Guide* for further details about partitioning

This section contains the following topics:


- About Partition Change Tracking
- Partitioning a Materialized View
- Partitioning a Prebuilt Table
- Rolling Materialized Views


### 6.1.1 About Partition Change Tracking

It is possible and advantageous to track freshness to a finer grain than the entire materialized view. You can achieve this through partition change tracking (PCT), which is a method to identify which rows in a materialized view are affected by a certain detail table partition. When one or more of the detail tables are partitioned, it may be possible to identify the specific rows in the materialized view that correspond to a modified detail partition(s); those rows become stale when a partition is modified while all other rows remain fresh.



Copyright © 2001, 2025, Oracle and/or its affiliates.

You can use PCT to identify which materialized view rows correspond to a particular partition. PCT is also used to support fast refresh after partition maintenance operations on detail tables. For instance, if a detail table partition is truncated or dropped, the affected rows in the materialized view are identified and deleted.

Identifying which materialized view rows are fresh or stale, rather than considering the entire materialized view as stale, allows query rewrite to use those rows that are fresh while in `QUERY_REWRITE_INTEGRITY = ENFORCED` or `TRUSTED` modes. Several views, such as `DBA_MVIEW_DETAIL_PARTITION`, detail which partitions are stale or fresh. Oracle does not rewrite against partial stale materialized views if partition change tracking on the changed table is enabled by the presence of join dependent expressions in the materialized view.

**See Also**

"About Join Dependent Expression and Partition Change Tracking" for more information

Note that, while partition change tracking tracks the staleness on a partition and subpartition level (for composite partitioned tables), the level of granularity for PCT refresh is only the top-level partitioning strategy. Consequently, any change to data in one of the subpartitions of a composite partitioned-table will only mark the single impacted subpartition as stale and have the rest of the table available for rewrite, but the PCT refresh will refresh the whole partition that contains the impacted subpartition.

To support PCT, a materialized view must satisfy the following requirements:

- At least one of the detail tables referenced by the materialized view must be partitioned.
- Partitioned tables must use either range, list or composite partitioning with range or list as the top-level partitioning strategy.
- The top level partition key must consist of only a single column.
- The materialized view must contain either the partition key column or a partition marker or `ROWID` or join dependent expression of the detail table.
- If you use a `GROUP BY` clause, the partition key column or the partition marker or `ROWID` or join dependent expression must be present in the `GROUP BY` clause.
- If you use an analytic window function or the `MODEL` clause, the partition key column or the partition marker or `ROWID` or join dependent expression must be present in their respective `PARTITION BY` subclauses.
- Data modifications can only occur on the partitioned table. If PCT refresh is being done for a table which has join dependent expression in the materialized view, then data modifications should not have occurred in any of the join dependent tables.
- The `COMPATIBILITY` initialization parameter must be a minimum of 9.0.0.0.0.

PCT is not supported for a materialized view that refers to views, remote tables, or outer joins.

**See Also**

*Oracle AI Database PL/SQL Packages and Types Reference* for details regarding the `DBMS_MVIEW.PMARKER` function and partition markers


Copyright © 2001, 2025, Oracle and/or its affiliates.

This section contains the following topics:

- About Partition Key and Partition Change Tracking
- About Join Dependent Expression and Partition Change Tracking
- About Partition Markers and Partition Change Tracking
- About Partial Rewrite in Partition Change Tracking

## 6.1.1.1 About Partition Key and Partition Change Tracking

Partition change tracking requires sufficient information in the materialized view to be able to correlate a detail row in the source partitioned detail table to the corresponding materialized view row. This can be accomplished by including the detail table partition key columns in the `SELECT` list and, if `GROUP BY` is used, in the `GROUP BY` list.

Consider an example of a materialized view storing daily customer sales. The following example uses the `sh` sample schema and the three detail tables `sales`, `products`, and `times` to create the materialized view. `sales` table is partitioned by `time_id` column and `products` is partitioned by the `prod_id` column. `times` is not a partitioned table.

**Example 6-1 Materialized View with Partition Key**

```
CREATE MATERIALIZED VIEW LOG ON SALES WITH ROWID
   (prod_id, time_id, quantity_sold, amount_sold) INCLUDING NEW VALUES;
CREATE MATERIALIZED VIEW LOG ON PRODUCTS WITH ROWID
   (prod_id, prod_name, prod_desc) INCLUDING NEW VALUES;
CREATE MATERIALIZED VIEW LOG ON TIMES WITH ROWID
   (time_id, calendar_month_name, calendar_year) INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW cust_dly_sales_mv
BUILD DEFERRED REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT s.time_id, p.prod_id, p.prod_name, COUNT(*),
         SUM(s.quantity_sold), SUM(s.amount_sold),
         COUNT(s.quantity_sold), COUNT(s.amount_sold)
FROM sales s, products p, times t
WHERE  s.time_id = t.time_id AND s.prod_id = p.prod_id
GROUP BY s.time_id, p.prod_id, p.prod_name;
```

For `cust_dly_sales_mv`, PCT is enabled on the `sales` table because the partitioning key column `time_id` is in the materialized view.

## 6.1.1.2 About Join Dependent Expression and Partition Change Tracking

An expression consisting of columns from tables directly or indirectly joined through equijoins to the partitioned detail table on the partitioning key and which is either a dimensional attribute or a dimension hierarchical parent of the joining key is called a join dependent expression. The set of tables in the path to detail table are called join dependent tables. Consider the following:

```
SELECT s.time_id, t.calendar_month_name
FROM sales s, times t WHERE s.time_id = t.time_id;
```

In this query, `times` table is a join dependent table because it is joined to `sales` table on the partitioning key column `time_id`. Moreover, `calendar_month_name` is a dimension hierarchical attribute of `times.time_id`, because `calendar_month_name` is an attribute of `times.mon_id` and `times.mon_id` is a dimension hierarchical parent of `times.time_id`. Hence, the expression `calendar_month_name` from `times` tables is a join dependent expression. Let's consider another example:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
SELECT s.time_id, y.calendar_year_name
FROM sales s, times_d d, times_m m, times_y y
WHERE s.time_id = d.time_id AND d.day_id = m.day_id AND m.mon_id = y.mon_id;
```

Here, `times` table is denormalized into `times_d`, `times_m` and `times_y` tables. The expression `calendar_year_name` from `times_y` table is a join dependent expression and the tables `times_d`, `times_m` and `times_y` are join dependent tables. This is because `times_y` table is joined indirectly through `times_m` and `times_d` tables to sales table on its partitioning key column `time_id`.

This lets users create materialized views containing aggregates on some level higher than the partitioning key of the detail table. Consider the following example of materialized view storing monthly customer sales.

**Example 6-2 Creating a Materialized View: Join Dependent Expression**

Assuming the presence of materialized view logs defined earlier, the materialized view can be created using the following DDL:

```
CREATE MATERIALIZED VIEW cust_mth_sales_mv
BUILD DEFERRED REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT t.calendar_month_name, p.prod_id, p.prod_name, COUNT(*),
    SUM(s.quantity_sold), SUM(s.amount_sold),
    COUNT(s.quantity_sold), COUNT(s.amount_sold)
FROM sales s, products p, times t
WHERE  s.time_id = t.time_id AND s.prod_id = p.prod_id
GROUP BY t.calendar_month_name, p.prod_id, p.prod_name;
```

Here, you can correlate a detail table row to its corresponding materialized view row using the join dependent table `times` and the relationship that `times.calendar_month_name` is a dimensional attribute determined by `times.time_id`. This enables partition change tracking on `sales` table. In addition to this, PCT is enabled on products table because of presence of its partitioning key column `prod_id` in the materialized view.

## 6.1.1.3 About Partition Markers and Partition Change Tracking

The `DBMS_MVIEW.PMARKER` function is designed to significantly reduce the cardinality (the ratio of distinct values to the number of table rows) of the materialized view (see Example 6-3 for an example). The function returns a partition identifier that uniquely identifies the partition or subpartition for a specified row within a specified partitioned table. Therefore, the `DBMS_MVIEW.PMARKER` function is used instead of the partition key column in the `SELECT` and `GROUP BY` clauses.

Unlike the general case of a PL/SQL function in a materialized view, use of the `DBMS_MVIEW.PMARKER` does not prevent rewrite with that materialized view even when the rewrite mode is `QUERY_REWRITE_INTEGRITY = ENFORCED`.

As an example of using the `PMARKER` function, consider calculating a typical number, such as revenue generated by a product category during a given year. If there were 1000 different products sold each month, it would result in 12,000 rows in the materialized view.

**Example 6-3 Using Partition Markers in a Materialized View**

Consider an example of a materialized view storing the yearly sales revenue for each product category. With approximately hundreds of different products in each product category, including the partitioning key column `prod_id` of the `products` table in the materialized view would substantially increase the cardinality. Instead, this materialized view uses the


Copyright © 2001, 2025, Oracle and/or its affiliates.

DBMS_MVIEW.PMARKER function, which increases the cardinality of materialized view by a factor of the number of partitions in the products table.

```
CREATE MATERIALIZED VIEW prod_yr_sales_mv
BUILD DEFERRED
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT DBMS_MVIEW.PMARKER(p.rowid), p.prod_category, t.calendar_year, COUNT(*),
       SUM(s.amount_sold), SUM(s.quantity_sold),
       COUNT(s.amount_sold), COUNT(s.quantity_sold)
FROM   sales s, products p, times t
WHERE  s.time_id = t.time_id AND s.prod_id = p.prod_id
GROUP BY  DBMS_MVIEW.PMARKER (p.rowid), p.prod_category, t.calendar_year;
```

prod_yr_sales_mv includes the DBMS_MVIEW.PMARKER function on the products table in its SELECT list. This enables partition change tracking on products table with significantly less cardinality impact than grouping by the partition key column prod_id. In this example, the desired level of aggregation for the prod_yr_sales_mv is to group by products.prod_category. Using the DBMS_MVIEW.PMARKER function, the materialized view cardinality is increased only by a factor of the number of partitions in the products table. This would generally be significantly less than the cardinality impact of including the partition key columns.

Note that partition change tracking is enabled on sales table because of presence of join dependent expression calendar_year in the SELECT list.

### 6.1.1.4 About Partial Rewrite in Partition Change Tracking

A subsequent INSERT statement adds a new row to the sales_part3 partition of table sales. At this point, because cust_dly_sales_mv has PCT available on table sales using a partition key, Oracle can identify the stale rows in the materialized view cust_dly_sales_mv corresponding to sales_part3 partition (The other rows are unchanged in their freshness state). Query rewrite cannot identify the fresh portion of materialized views cust_mth_sales_mv and prod_yr_sales_mv because PCT is available on table sales using join dependent expressions. Query rewrite can determine the fresh portion of a materialized view on changes to a detail table only if PCT is available on the detail table using a partition key or partition marker.

## 6.1.2 Partitioning a Materialized View

Partitioning a materialized view involves defining the materialized view with the standard Oracle partitioning clauses, as illustrated in the following example. This statement creates a materialized view called part_sales_mv, which uses three partitions, can be fast refreshed, and is eligible for query rewrite:

```
CREATE MATERIALIZED VIEW part_sales_mv
PARALLEL PARTITION BY RANGE (time_id)
(PARTITION month1
      VALUES LESS THAN (TO_DATE('31-12-1998', 'DD-MM-YYYY'))
      PCTFREE 0
      STORAGE (INITIAL 8M)
      TABLESPACE sf1,
 PARTITION month2
      VALUES LESS THAN (TO_DATE('31-12-1999', 'DD-MM-YYYY'))
      PCTFREE 0
      STORAGE (INITIAL 8M)
      TABLESPACE sf2,
 PARTITION month3
      VALUES LESS THAN (TO_DATE('31-12-2000', 'DD-MM-YYYY'))
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
      PCTFREE 0
      STORAGE (INITIAL 8M)
      TABLESPACE sf3)
BUILD DEFERRED
REFRESH FAST
ENABLE QUERY REWRITE AS
SELECT s.cust_id, s.time_id,
  SUM(s.amount_sold) AS sum_dol_sales, SUM(s.quantity_sold) AS sum_unit_sales
  FROM sales s GROUP BY s.time_id, s.cust_id;
```

## 6.1.3 Partitioning a Prebuilt Table

Alternatively, a materialized view can be registered to a partitioned prebuilt table. "Benefits of Partitioning a Materialized View" describes the benefits of partitioning a prebuilt table. The following example illustrates this:

```
CREATE TABLE part_sales_tab_mv(time_id, cust_id, sum_dollar_sales, sum_unit_sale)
PARALLEL PARTITION BY RANGE (time_id)
(PARTITION month1
      VALUES LESS THAN (TO_DATE('31-12-1998', 'DD-MM-YYYY'))
      PCTFREE 0
      STORAGE (INITIAL 8M)
      TABLESPACE sf1,
 PARTITION month2
      VALUES LESS THAN (TO_DATE('31-12-1999', 'DD-MM-YYYY'))
      PCTFREE 0
      STORAGE (INITIAL 8M)
      TABLESPACE sf2,
PARTITION month3
      VALUES LESS THAN (TO_DATE('31-12-2000', 'DD-MM-YYYY'))
      PCTFREE 0
      STORAGE (INITIAL 8M)
      TABLESPACE sf3)  AS
SELECT s.time_id, s.cust_id, SUM(s.amount_sold) AS sum_dollar_sales,
  SUM(s.quantity_sold) AS sum_unit_sales
FROM sales s GROUP BY s.time_id, s.cust_id;

CREATE MATERIALIZED VIEW part_sales_tab_mv
ON PREBUILT TABLE
ENABLE QUERY REWRITE AS
SELECT s.time_id,  s.cust_id, SUM(s.amount_sold) AS sum_dollar_sales,
     SUM(s.quantity_sold) AS sum_unit_sales
FROM sales s GROUP BY s.time_id, s.cust_id;
```

In this example, the table `part_sales_tab_mv` has been partitioned over three months and then the materialized view was registered to use the prebuilt table. This materialized view is eligible for query rewrite because the `ENABLE QUERY REWRITE` clause has been included.

### 6.1.3.1 Benefits of Partitioning a Materialized View

When a materialized view is partitioned on the partitioning key column or join dependent expressions of the detail table, it is more efficient to use a `TRUNCATE PARTITION` statement to remove one or more partitions of the materialized view during refresh and then repopulate the partition with new data. Oracle AI Database uses this variant of fast refresh (called PCT refresh) with partition truncation if the following conditions are satisfied in addition to other conditions described in "About Partition Change Tracking".

- The materialized view is partitioned on the partitioning key column or join dependent expressions of the detail table.


Copyright © 2001, 2025, Oracle and/or its affiliates.

- If PCT is enabled using either the partitioning key column or join expressions, the materialized view should be range or list partitioned.
- PCT refresh is nonatomic.

### 6.1.4 Rolling Materialized Views

When a data warehouse or data mart contains a time dimension, it is often desirable to archive the oldest information and then reuse the storage for new information. This is called the rolling window scenario. If the fact tables or materialized views include a time dimension and are horizontally partitioned by the time attribute, then management of rolling materialized views can be reduced to a few fast partition maintenance operations provided the unit of data that is rolled out equals, or is at least aligned with, the range partitions.

If you plan to have rolling materialized views in your data warehouse, you should determine how frequently you plan to perform partition maintenance operations, and you should plan to partition fact tables and materialized views to reduce the amount of system administration overhead required when old data is aged out. An additional consideration is that you might want to use data compression on your infrequently updated partitions.

You are not restricted to using range partitions. For example, a composite partition using both a time value and a key value could result in a good partition solution for your data.

> **See Also**
>
> Refreshing Materialized Views for further details regarding `CONSIDER FRESH` and for details regarding compression

### 6.1.5 About Automatic Partitioning of Materialized Views

Automatic materialized views support automatic partitioning.

Note the following:

- An automatic materialize view is automatically partitioned only if the underlying fact table is partitioned.
- Only only list partitioning is used in automatic partitioning.

> **See Also**
>
> The Database Licensing Information User Manual for availability details.

## 6.2 About Materialized Views in Analytic Processing Environments

This section discusses the concepts used by analytic SQL and how relational databases can handle these types of queries. It also illustrates the best approach for creating materialized views using a common scenario.


Copyright © 2001, 2025, Oracle and/or its affiliates.

The following topics contain additional information about materialized views in different environments:

- About Materialized Views and Analytic Views
- About Materialized Views and Hierarchical Cubes
- Benefits of Partitioning Materialized Views
- About Compressing Materialized Views
- About Materialized Views with Set Operators

## 6.2.1 About Materialized Views and Analytic Views

Creating a materialized view over queries of an analytic view or a hierarchy is not supported.

## 6.2.2 About Materialized Views and Hierarchical Cubes

While data warehouse environments typically view data in the form of a star schema, for analytical SQL queries, data is held in the form of a hierarchical cube. A hierarchical cube includes the data aggregated along the rollup hierarchy of each of its dimensions and these aggregations are combined across dimensions. It includes the typical set of aggregations needed for business intelligence queries.

**Example 6-4 Hierarchical Cube**

Consider a sales data set with two dimensions, each of which has a four-level hierarchy:

- Time, which contains (all times), year, quarter, and month.
- Product, which contains (all products), division, brand, and item.

This means there are 16 aggregate groups in the hierarchical cube. This is because the four levels of time are multiplied by four levels of product to produce the cube. Table 6-1 shows the four levels of each dimension.

**Table 6-1 ROLLUP By Time and Product**

| ROLLUP By Time | ROLLUP By Product |
|---|---|
| year, quarter, month | division, brand, item |
| year, quarter | division, brand |
| year | division |
| all times | all products |

Note that as you increase the number of dimensions and levels, the number of groups to calculate increases dramatically. This example involves 16 groups, but if you were to add just two more dimensions with the same number of levels, you would have 4 x 4 x 4 x 4 = 256 different groups. Also, consider that a similar increase in groups occurs if you have multiple hierarchies in your dimensions. For example, the time dimension might have an additional hierarchy of fiscal month rolling up to fiscal quarter and then fiscal year. Handling the explosion of groups has historically been the major challenge in data storage for online analytical processing systems.

Typical online analytical queries slice and dice different parts of the cube comparing aggregations from one level to aggregation from another level. For instance, a query might find sales of the grocery division for the month of January, 2002 and compare them with total sales of the grocery division for all of 2001.


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 6.2.3 Benefits of Partitioning Materialized Views

Materialized views with multiple aggregate groups give their best performance for refresh and query rewrite when partitioned appropriately.

PCT refresh in a rolling window scenario requires partitioning at the top level on some level from the time dimension. And, partition pruning for queries rewritten against this materialized view requires partitioning on `GROUPING_ID` column. Hence, the most effective partitioning scheme for these materialized views is to use composite partitioning (range-list on (`time`, `GROUPING_ID`) columns). By partitioning the materialized views this way, you enable:

- PCT refresh, thereby improving refresh performance.
- Partition pruning: only relevant aggregate groups are accessed, thereby greatly reducing the query processing cost.

If you do not want to use PCT refresh, you can just partition by list on `GROUPING_ID` column.

## 6.2.4 About Compressing Materialized Views

You should consider data compression when using highly redundant data, such as tables with many foreign keys. In particular, materialized views created with the `ROLLUP` clause are likely candidates.

**See Also**

- *Oracle AI Database SQL Language Reference* for data compression syntax and restrictions
- "About Storage And Table Compression for Materialized Views" for details regarding compression

## 6.2.5 About Materialized Views with Set Operators

Oracle AI Database provides support for materialized views whose defining query involves set operators. Materialized views with set operators can now be created enabled for query rewrite. You can refresh the materialized view using either `ON COMMIT` or `ON DEMAND` refresh.

Fast refresh is supported if the defining query has the `UNION ALL` operator at the top level and each query block in the `UNION ALL`, meets the requirements of a materialized view with aggregates or materialized view with joins only. Further, the materialized view must include a constant column (known as a `UNION ALL` marker) that has a distinct value in each query block, which, in the following example, is columns `1 marker` and `2 marker`.

**See Also**

"Restrictions on Fast Refresh on Materialized Views with UNION ALL" for detailed restrictions on fast refresh for materialized views with `UNION ALL`.


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 6.2.5.1 Examples of Materialized Views Using UNION ALL

The following examples illustrate creation of fast refreshable materialized views involving UNION ALL.

**Example 6-5 Materialized View Using UNION ALL with Two Join Views**

To create a UNION ALL materialized view with two join views, the materialized view logs must have the rowid column and, in the following example, the UNION ALL marker is the columns, 1 marker and 2 marker.

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID;
CREATE MATERIALIZED VIEW LOG ON customers WITH ROWID;

CREATE MATERIALIZED VIEW unionall_sales_cust_joins_mv
REFRESH FAST ON COMMIT
ENABLE QUERY REWRITE AS
(SELECT c.rowid crid, s.rowid srid, c.cust_id, s.amount_sold, 1 marker
FROM sales s, customers c
WHERE s.cust_id = c.cust_id AND c.cust_last_name = 'Smith')
UNION ALL
(SELECT c.rowid crid, s.rowid srid, c.cust_id, s.amount_sold, 2 marker
FROM sales s, customers c
WHERE s.cust_id = c.cust_id AND c.cust_last_name = 'Brown');
```

**Example 6-6 Materialized View Using UNION ALL with Joins and Aggregates**

The following example shows a UNION ALL of a materialized view with joins and a materialized view with aggregates. A couple of things can be noted in this example. Nulls or constants can be used to ensure that the data types of the corresponding SELECT list columns match. Also, the UNION ALL marker column can be a string literal, which is 'Year' umarker, 'Quarter' umarker, or 'Daily' umarker in the following example:

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID, SEQUENCE
(amount_sold, time_id)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW LOG ON times WITH ROWID, SEQUENCE
  (time_id, fiscal_year, fiscal_quarter_number, day_number_in_week)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW unionall_sales_mix_mv
REFRESH FAST ON DEMAND AS
(SELECT 'Year' umarker, NULL, NULL, t.fiscal_year,
        SUM(s.amount_sold) amt, COUNT(s.amount_sold), COUNT(*)
 FROM sales s, times t
 WHERE s.time_id = t.time_id
 GROUP BY t.fiscal_year)
UNION ALL
(SELECT 'Quarter' umarker, NULL, NULL, t.fiscal_quarter_number,
        SUM(s.amount_sold) amt, COUNT(s.amount_sold), COUNT(*)
FROM sales s, times t
WHERE s.time_id = t.time_id and t.fiscal_year = 2001
GROUP BY t.fiscal_quarter_number)
UNION ALL
(SELECT 'Daily' umarker, s.rowid rid, t.rowid rid2, t.day_number_in_week,
        s.amount_sold amt, 1, 1
FROM sales s, times t
WHERE s.time_id = t.time_id
AND t.time_id between '01-Jan-01' AND '01-Dec-31');
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

# 6.3 About Materialized Views and Models

Models, which provide array-based computations in SQL, can be used in materialized views. Because the `MODEL` clause calculations can be expensive, you may want to use two separate materialized views: one for the model calculations and one for the `SELECT ... GROUP BY` query. For example, instead of using one, long materialized view, you could create the following materialized views:

```
CREATE MATERIALIZED VIEW my_groupby_mv
REFRESH FAST
ENABLE QUERY REWRITE AS
SELECT country_name country, prod_name prod, calendar_year year,
  SUM(amount_sold) sale, COUNT(amount_sold) cnt, COUNT(*) cntstr
FROM sales, times, customers, countries, products
WHERE sales.time_id = times.time_id AND
      sales.prod_id = products.prod_id AND
      sales.cust_id = customers.cust_id AND
      customers.country_id = countries.country_id
GROUP BY country_name, prod_name, calendar_year;

CREATE MATERIALIZED VIEW my_model_mv
ENABLE QUERY REWRITE AS
SELECT country, prod, year, sale, cnt
FROM my_groupby_mv
MODEL PARTITION BY(country) DIMENSION BY(prod, year)
   MEASURES(sale s) IGNORE NAV
(s['Shorts', 2000] =  0.2 * AVG(s)[CV(), year BETWEEN 1996 AND 1999],
s['Kids Pajama', 2000] = 0.5 * AVG(s)[CV(), year BETWEEN 1995 AND 1999],
s['Boys Pajama', 2000] = 0.6 * AVG(s)[CV(), year BETWEEN 1994 AND 1999],
...
<hundreds of other update rules>);
```

By using two materialized views, you can incrementally maintain the materialized view `my_groupby_mv`. The materialized view `my_model_mv` is on a much smaller data set because it is built on `my_groupby_mv` and can be maintained by a complete refresh.

Materialized views with models can use complete refresh or PCT refresh only, and are available for partial text query rewrite only.

> **See Also**
>
> SQL for Modeling for further details about model calculations

# 6.4 About Security Issues with Materialized Views

To create a materialized view in your own schema, you must have the `CREATE MATERIALIZED VIEW` privilege and the `SELECT` or `READ` privilege to any tables referenced that are in another schema. To create a materialized view in another schema, you must have the `CREATE ANY MATERIALIZED VIEW` privilege and the owner of the materialized view needs `SELECT` or `READ` privileges to the tables referenced if they are from another schema. Moreover, if you enable query rewrite on a materialized view that references tables outside your schema, you must have the `GLOBAL QUERY REWRITE` privilege or the `QUERY REWRITE` object privilege on each table outside your schema.


Copyright © 2001, 2025, Oracle and/or its affiliates.

If the materialized view is on a prebuilt container, the creator, if different from the owner, must have the `READ WITH GRANT` or `SELECT WITH GRANT` privilege on the container table.

If you continue to get a privilege error while trying to create a materialized view and you believe that all the required privileges have been granted, then the problem is most likely due to a privilege not being granted explicitly and trying to inherit the privilege from a role instead. The owner of the materialized view must have explicitly been granted `SELECT` or `READ` access to the referenced tables if the tables are in a different schema.

If the materialized view is being created with `ON COMMIT REFRESH` specified, then the owner of the materialized view requires an additional privilege if any of the tables in the defining query are outside the owner's schema. In that case, the owner requires the `ON COMMIT REFRESH` system privilege or the `ON COMMIT REFRESH` object privilege on each table outside the owner's schema.

**See Also**

Querying Materialized Views with Virtual Private Database (VPD)

## 6.4.1 Querying Materialized Views with Virtual Private Database (VPD)

For all security concerns, a materialized view serves as a view that happens to be materialized when you are directly querying the materialized view. When creating a view or materialized view, the owner must have the necessary permissions to access the underlying base relations of the view or materialized view that they are creating. With these permissions, the owner can publish a view or materialized view that other users can access, assuming they have been granted access to the view or materialized view.

Using materialized views with Virtual Private Database is similar. When you create a materialized view, there must not be any VPD policies in effect against the base relations of the materialized view for the owner of the materialized view. If any VPD policies exist, then you must use the `USING TRUSTED CONSTRAINTS` clause when creating the materialized view. The owner of the materialized view may establish a VPD policy on the new materialized view. Users who access the materialized view are subject to the VPD policy on the materialized view. However, they are not additionally subject to the VPD policies of the underlying base relations of the materialized view, because security processing of the underlying base relations is performed against the owner of the materialized view.

This section contains the following topics:

- Using Query Rewrite with Virtual Private Database
- Restrictions with Materialized Views and Virtual Private Database

### 6.4.1.1 Using Query Rewrite with Virtual Private Database

When you access a materialized view using query rewrite, the materialized view serves as an access structure much like an index. As such, the security implications for materialized views accessed in this way are much the same as for indexes: all security checks are performed against the relations specified in the request query. The index or materialized view is used to speed the performance of accessing the data, not provide any additional security checks. Thus, the presence of the index or materialized view presents no additional security checking.

This holds true when you are accessing a materialized view using query rewrite in the presence of VPD. The request query is subject to any VPD policies that are present against


Copyright © 2001, 2025, Oracle and/or its affiliates.

the relations specified in the query. Query rewrite may rewrite the query to use a materialize view instead of accessing the detail relations, but only if it can guarantee to deliver exactly the same rows as if the rewrite had not occurred. Specifically, query rewrite must retain and respect any VPD policies against the relations specified in the request query. However, any VPD policies against the materialized view itself do not have effect when the materialized view is accessed using query rewrite. This is because the data is already protected by the VPD policies against the relations in the request query.

### 6.4.1.2 Restrictions with Materialized Views and Virtual Private Database

Query rewrite does not use its full and partial text match modes with request queries that include relations with active VPD policies, but it does use general rewrite methods. This is because VPD transparently transforms the request query to affect the VPD policy. If query rewrite were to perform a text match transformation against a request query with a VPD policy, the effect would be to negate the VPD policy.

In addition, when you create or refresh a materialized view, the owner of the materialized view must not have any active VPD policies in effect against the base relations of the materialized view, or an error is returned. The materialized view owner must either have no such VPD policies, or any such policy must return `NULL`. This is because VPD would transparently modify the defining query of the materialized view such that the set of rows contained by the materialized view would not match the set of rows indicated by the materialized view definition.

One way to work around this restriction yet still create a materialized view containing the desired VPD-specified subset of rows is to create the materialized view in a user account that has no active VPD policies against the detail relations of the materialized view. In addition, you can include a predicate in the `WHERE` clause of the materialized view that embodies the effect of the VPD policy. When query rewrite attempts to rewrite a request query that has that VPD policy, it matches up the VPD-generated predicate on the request query with the predicate you directly specify when you create the materialized view.

## 6.5 Invalidating Materialized Views

Dependencies related to materialized views are automatically maintained to ensure correct operation. When a materialized view is created, the materialized view depends on the detail tables referenced in its definition. Any DML operation, such as an `INSERT`, or `DELETE`, `UPDATE`, or DDL operation on any dependency in the materialized view will cause it to become invalid. To revalidate a materialized view, use the `ALTER MATERIALIZED VIEW COMPILE` statement.

A materialized view is automatically revalidated when it is referenced. In many cases, the materialized view will be successfully and transparently revalidated. However, if a column has been dropped in a table referenced by a materialized view or the owner of the materialized view did not have one of the query rewrite privileges and that privilege has now been granted to the owner, you should use the following statement to revalidate the materialized view:

```
ALTER MATERIALIZED VIEW mview_name COMPILE;
```

The state of a materialized view can be checked by querying the data dictionary views `USER_MVIEWS` or `ALL_MVIEWS`. The column `STALENESS` will show one of the values `FRESH`, `STALE`, `UNUSABLE`, `UNKNOWN`, `UNDEFINED`, or `NEEDS_COMPILE` to indicate whether the materialized view can be used. The state is maintained automatically. However, if the staleness of a materialized view is marked as `NEEDS_COMPILE`, you could issue an `ALTER MATERIALIZED VIEW ... COMPILE` statement to validate the materialized view and get the correct staleness state. If the state of a materialized view is `UNUSABLE`, you must perform a complete refresh to bring the materialized view back to the `FRESH` state. If the materialized view is based on a prebuilt table that you never refresh, you must drop and re-create the materialized view. The staleness of remote


Copyright © 2001, 2025, Oracle and/or its affiliates.

materialized views is not tracked. Thus, if you use remote materialized views for rewrite, they are considered to be trusted.

## 6.6 Altering Materialized Views

The following modifications can be made to a materialized view:

- Change its refresh option (`FAST/FORCE/COMPLETE/NEVER`).
- Change its refresh mode (`ON COMMIT/ON DEMAND`).
- Recompile it.
- Enable or disable its use for query rewrite.
- Consider it fresh.
- Partition maintenance operations.
- Enable on-query computation

All other changes are achieved by dropping and then re-creating the materialized view. The success of a modification operation depends on whether the requirement for the change is satisfied. For example, a fast refresh succeeds if materialized view logs exist on all the base tables.

The `COMPILE` clause of the `ALTER MATERIALIZED VIEW` statement can be used when the materialized view has been invalidated. This compile process is quick, and allows the materialized view to be used by query rewrite again.

**See Also**

- *Oracle AI Database SQL Language Reference* for further information about the `ALTER MATERIALIZED VIEW` statement
- "Invalidating Materialized Views"

## 6.7 Using Real-time Materialized Views

Real-time materialized views provide fresh data to user queries even when the materialized view is marked as stale.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**See Also**

- Overview of Real-time Materialized Views
- Creating Real-time Materialized Views
- Converting an Existing Materialized View into a Real-time Materialized View
- Enabling Query Rewrite to Use Real-time Materialized Views
- Using Real-time Materialized Views During Query Rewrite
- Using Real-time Materialized Views for Direct Query Access
- Listing Real-time Materialized Views
- Improving Real-time Materialized Views Performance

## 6.7.1 Overview of Real-time Materialized Views

A real-time materialized view is a type of materialized view that provides fresh data to user queries even when the materialized view is not in sync with its base tables because of data changes.

Unless a SQL session is set to stale tolerated mode, a materialized view that is marked stale cannot be used for query rewrite. Organizations that require real-time data typically use the `ON COMMIT` refresh mode to ensure that the materialized view is updated with changes made to the base tables. However, when DML changes to the base tables are huge and very frequent, this mode may result in resource contention and reduced refresh performance. Real-time materialized views provide a lightweight solution for obtaining fresh data from stale materialized views by recomputing the data on the fly.

Real-time materialized views can use any available out-of-place refresh method including log-based or PCT based refresh. They can be used either with on demand or scheduled automatic refresh, but not with automatic refresh specified using the `ON COMMIT` clause.

**Advantages of Real-time Materialized Views**

- Provides improved availability for materialized views
- Provides fresh data for user queries that access a materialized view that may be stale

**How Do Real-time Materialized Views Work?**

Real-time materialized views use a technique called *on-query computation* to provide fresh data with stale materialized views. When a query accesses a real-time materialized view, Oracle AI Database first checks if the real-time materialized view is marked as stale. If it is not stale, then the required data is provided using the real-time materialized view as it is. If the real-time materialized view is marked as stale, then the on-query computation technique is used to generate the fresh data and return the correct query result.

Real-time materialized views use a technique that is similar log-based refresh to provide fresh data with stale materialized view. They combine the existing data with the changes that are recorded in change logs to obtain the latest data. However, unlike log-based refresh, real-time materialized views do not use the materialized view logs to update the data in the real-time materialized view. Instead, when a query accesses a stale real-time materialized view, the data that is recomputed using on-query computation is used directly to answer the query.

A real-time materialized view is created by using the `ON QUERY COMPUTATION` clause in the materialized view definition.


Copyright © 2001, 2025, Oracle and/or its affiliates.

### 6.7.1.1 Restrictions on Using Real-time Materialized Views

Using real-time materialized views is subject to certain restrictions.

- Real-time materialized views cannot be used when:
  - one or more materialized view logs created on the base tables are either unusable or nonexistent.
  - out-of-place, log-based or PCT refresh is not feasible for the change scenarios.
  - automatic refresh is specified using the `ON COMMIT` clause.
- If a real-time materialized view is a nested materialized view that is defined on top of one or more base materialized views, then query rewrite occurs only if all the base materialized views are fresh. If one or more base materialized views are stale, then query rewrite is not performed using this real-time materialized view.

The cursors of queries that directly access real-time materialized views are not shared.

### 6.7.1.2 About Accessing Real-time Materialized Views

As with materialized views, multiple methods exist to access data stored in real-time materialized views.

Data stored in real-time materialized views can be accessed in one of the following ways:

- Query rewrite

  A user query that is similar to the real-time materialized view definition is rewritten to use the real-time materialized view.
- Direct access of real-time materialized views

  A user query directly references the real-time materialized view by using its name.

In both scenarios, the content of a real-time materialized view can be accessed as stale data or can trigger an on-query computation of the correct result. Whether or not on-query computation is triggered depends on the environment and the actual SQL statement.

The output of the `EXPLAIN PLAN` statement contains messages indicating if on-query computation was used for a particular user query.

> **See Also**
>
> - Using Real-time Materialized Views for Direct Query Access
> - Using Real-time Materialized Views During Query Rewrite

## 6.7.2 Creating Real-time Materialized Views

To create a real-time materialized view, use the `ON QUERY COMPUTATION` clause in the `CREATE MATERIALIZED VIEW` statement.

You can create real-time materialized views even if they are not applicable for on-query computation for all change scenarios. The minimum requirement to create a real-time materialized view is that it supports out-of-place refresh for `INSERT` operations. If other change


Copyright © 2001, 2025, Oracle and/or its affiliates.

scenarios, such as mixed DML operations, are encountered, then on-query computation may not be feasible for all types of real-time materialized views.

Real-time materialized view must use an out-of-place log-based refresh mechanism (including PCT refresh). The `ON COMMIT` refresh mode cannot be used for real-time materialized views.

**To create a real-time materialized view:**

1. Ensure that materialized view logs exist on all the base tables of the real-time materialized view.
2. Create materialized view logs for all the tables on which the real-time materialized view is based.
3. Create the real-time materialized view by including the `ENABLE ON QUERY COMPUTATION` clause in the `CREATE MATERIALIZED VIEW` statement.

**Example 6-7 Creating a Real-time Materialized View**

This example creates a real-time materialized view called `SUM_SALES_RTMV` which is based on data aggregated from the `SALES` and `PRODUCTS` tables in the `SH` schema. Before you create the real-time materialized view ensure that the required prerequisites are met.

1. Create materialized view logs on the base tables `SALES` and `PRODUCTS`.

   The following command creates a materialized view log on the `SALES` table:

   ```
   CREATE MATERIALIZED VIEW LOG ON sales
   WITH SEQUENCE, ROWID
   (prod_id, quantity_sold, amount_sold)
   INCLUDING NEW VALUES;
   ```

   The following command creates a materialized view log on the `PRODUCTS` table.

   ```
   CREATE MATERIALIZED VIEW LOG ON products
   WITH ROWID
   (prod_id, prod_name, prod_category, prod_subcategory)
   INCLUDING NEW VALUES;
   ```

2. Create a real-time materialized view by including the `ON QUERY COMPUTATION` clause in the `CREATE MATERIALIZED VIEW` statement. The fast refresh method is used for this real-time materialized view and the `ENABLE QUERY REWRITE` clause indicates that query rewrite must be enabled.

   ```
   CREATE MATERIALIZED VIEW sum_sales_rtmv
   REFRESH FAST ON DEMAND
   ENABLE QUERY REWRITE
   ENABLE ON QUERY COMPUTATION
   AS
   SELECT prod_name, SUM(quantity_sold) AS sum_qty, COUNT(quantity_sold) AS
   cnt_qty, SUM(amount_sold) AS sum_amt,
        COUNT(amount_sold) AS cnt_amt, COUNT(*) AS cnt_star
   FROM sales, products
   WHERE sales.prod_id = products.prod_id
   GROUP BY prod_name;
   ```


Copyright © 2001, 2025, Oracle and/or its affiliates.

After the SUM_SALES_RTMV real-time materialized view is created, assume that the following query is run.

```
SELECT prod_name, SUM(quantity_sold), SUM(amount_sold)
FROM sales, products
WHERE sales.prod_id = products.prod_id
GROUP BY prod_name;
```

If SUM_SALES_RTMV is not stale, then the query result is returned using the data stored in this real-time materialized view. However, if SUM_SALES_RTMV is stale and the cost of rewriting the query using the materialized view with on-query computation is lower than the base table access, then the query is answered by combining the delta changes in the materialized view logs on the SALES and PRODUCTS tables with the data in the real-time materialized view SUM_SALES_RTMV.

## 6.7.3 Converting an Existing Materialized View into a Real-time Materialized View

If the prerequisites for a real-time materialized view are met, then an existing materialized view can be converted into a real-time materialized view by altering its definition and enabling on-query computation.

**To convert a materialized view into a real-time materialized view:**

- Modify the materialized view definition and enable on-query computation by using the ON QUERY COMPUTATION clause in the ALTER MATERIALIZED VIEW statement.

You can convert a real-time materialized view into a regular materialized view by disabling on-query computation using the DISABLE ON QUERY COMPUTATION clause in the ALTER MATERIALIZED VIEW statement.

**Example 6-8 Converting a Materialized View into a Real-time Materialized View**

The materialized view SALES_RTMV is based on the SALES, TIMES, and PRODUCTS tables and uses fast refresh. Materialized view logs exist on all three base tables. You want to modify this materialized view and convert it into a real-time materialized view.

1. Modify the materialized view definition and include the ON QUERY COMPUTATION clause to change it into a real-time materialized view.

   ```
   ALTER MATERIALIZED VIEW sales_rtmv ENABLE ON QUERY COMPUTATION;
   ```

2. Query the DBA_MVIEWS view to determine if on-query computation is enabled for SALES_RTMV.

   ```
   SELECT mview_name, on_query_computation
   FROM dba_mviews
   WHERE mview_name = 'SALES_RTMV';
   ```


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 6.7.4 Enabling Query Rewrite to Use Real-time Materialized Views

For the query rewrite mechanism to rewrite a user query to use real-time materialized views, query rewrite must be enabled for the real-time materialized view.

You can enable query rewrite for a real-time materialized view either at creation time or subsequently, by modifying the definition of the real-time materialized view. The `ENABLE QUERY REWRITE` clause is used to enable query rewrite.

To enable query rewrite for an existing real-time materialized view:

- Run the `ALTER MATERIALIZED VIEW` command and include the `ENABLE QUERY REWRITE` clause.

**Example 6-9 Enabling Query Rewrite for Real-time Materialized Views**

The real-time materialized view `my_rtmv` uses the fast refresh mechanism. You want to modify the definition of this real-time materialized view and specify that the query rewrite mechanism must consider this real-time materialized view while rewriting queries.

The following command enables query rewrite for `my_rtmv`:

```
ALTER MATERIALIZED VIEW my_rtmv ENABLE QUERY REWRITE;
```

## 6.7.5 Using Real-time Materialized Views During Query Rewrite

Query rewrite can use a real-time materialized view to provide results to user queries, even if the real-time materialized view is stale, if query rewrite is enabled for the real-time materialized view. A nested real-time materialized view is eligible for query rewrite only if all its base real-time materialized views are fresh.

When a user query is run, query rewrite first checks if a fresh materialized view is available to provide the required data. If a suitable materialized view does not exist, then query rewrite looks for a real-time materialized view that can be used to rewrite the user query. A fresh materialized view is preferred over a real-time materialized view because some overhead is incurred in computing fresh data for real-time materialized view. Next, the cost based optimizer determines the cost of the SQL query with on-query computation and then decides if the real-time materialized view will be used to answer this user query.

If the `QUERY_REWRITE_INTEGRITY` mode of the current SQL session is set to `STALE_TOLERATED`, then on-query computation will not be used during query rewrite. The `STALE_TOLERATED` rewrite mode indicates that fresh results are not required to satisfy a query, so on-query computation is not necessary.

**For query rewrite to use a real-time materialized view:**

1. Ensure that `QUERY_REWRITE_INTEGRITY` is set to either `ENFORCED` or `TRUSTED` mode. `QUERY_REWRITE_INTEGRITY` mode should not be set to `STALE_TOLERATED` mode.
2. Run a user query that matches the SQL query that was used to define the real-time materialized view.

   Any query that can be rewritten to take advantage of a real-time materialized view will use the real-time materialized view with on-query computation.

   Use `EXPLAIN PLAN` to verify that the query was rewritten using the real-time materialized view.


Copyright © 2001, 2025, Oracle and/or its affiliates.

### Example 6-10 Using Real-time Materialized Views During Query Rewrite

This example creates a real-time materialized view with query rewrite enabled and then demonstrates that it was used by query rewrite to provide data for a user query.

1. Create a materialized view log on the SALES table, which is the base table for the real-time materialized view being created.
2. Create a real-time materialized view mav_sum_sales with query rewrite enabled.

```
CREATE MATERIALIZED VIEW mav_sum_sales
REFRESH FAST ON DEMAND
ENABLE ON QUERY COMPUTATION
ENABLE QUERY REWRITE
AS
SELECT prod_id, sum(quantity_sold) as sum_qty, count(quantity_sold) as
cnt_qty,
       sum(amount_sold) sum_amt, count(amount_sold) cnt_amt, count(*) as
cnt_star
FROM sales
GROUP BY prod_id;
```

3. Run the following query:

```
SELECT prod_id, sum(quantity_sold), sum(amount_sold)
FROM  sales
WHERE prod_id < 1000
GROUP BY prod_id;
```

   Observe that the query is similar to the one used to define the real-time materialized view mav_sum_sales. Because no other materialized view with a definition that is similar to the query exists, query rewrite can use the mav_sum_sales real-time materialized view to determine the query result. You can verify that query rewrite has taken place by checking the SQL cursor cache (for example, with DBMS_XPLAN), using SQL Monitor, or using EXPLAIN PLAN.

   The internally rewritten query that uses mav_sum_sales is analogous to the following statement:

```
SELECT prod_id, sum_qty, sum_amt
FROM  mav_sum_sales
WHERE prod_id < 1000;
```

4. Verify that the real-time materialized view was used to provide the query result. Use the EXPLAIN PLAN statement to view the execution plan for the query.

   The following execution plan shows direct access to the real-time materialized view. If the materialized view is stale, then the execution plan will become more complex and include access to other objects (for example, the materialized view logs), depending on the outstanding DML operations.

```
EXPLAIN PLAN for SELECT prod_id, sum(quantity_sold), sum(amount_sold) FROM
sales WHERE prod_id < 1000 GROUP BY prod_id;
SELECT plan_table_output FROM
table(dbms_xplan.display('plan_table',null,'serial'));

PLAN_TABLE_OUTPUT
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
-------------------------------------------------------------------------------
--------------------
Plan hash value: 13616844
-------------------------------------------------------------------------------
--------------------
| Id  | Operation             | Name          | Rows  | Bytes | Cost
(%CPU) | Time      |
-------------------------------------------------------------------------------
--------------------
| 0   | SELECT STATEMENT      |               | 92    | 3588  | 3
(0)      | 00:00:01  |
| *1  | MAT_VIEW ACCESS FULL  | MAV_SUM_SALES | 92    | 3588  | 3
(0)      | 00:00:01  |
-------------------------------------------------------------------------------
--------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("PROD_ID"<1000)

Note
-----
   - dynamic statistics used: dynamic sampling (level=2)

17 rows selected.
```

## 6.7.6 Using Real-time Materialized Views for Direct Query Access

You can access a real-time materialized view directly by referencing the name of the real-time materialized view in a query.

If the real-time materialized view specified in a user query is fresh, then the required data is directly fetched from the real-time materialized view. If the real-time materialized view is stale, then you must use the `FRESH_MV` hint to perform on-query computation and obtain fresh data. Oracle AI Database does not automatically perform on-query computation for a real-time materialized view that is accessed directly in a user query.

To obtain fresh data from a stale real-time materialized view when directly accessing the real-time materialized view:

- Use the `FRESH_MV` hint in the user query to indicate that on-query computation must be performed.

**Example 6-11 Creating a Real-Time Materialized View and Using it in Queries**

This example creates a real-time materialized view `MY_RTMV` that is based on the `SALES_NEW` table. The `SALES_NEW` table is created as a copy of the `SH.SALES` table. A row is inserted into the base table after the real-time materialized view is created. Next the `fresh_mv` hint is used to access fresh data from the real-time materialized view by using the materialized view name in a user query.

1. Create a materialized view log on the base table `sales_new`.

   Materialized view logs on the base table are mandatory for creating real-time materialized views.

   ```
   CREATE MATERIALIZED VIEW LOG on sales_new
   WITH sequence, ROWID (prod_id, cust_id, time_id, channel_id, promo_id,
   ```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
quantity_sold, amount_sold)
INCLUDING NEW VALUES;
```

2. Create a real-time materialized view called `my_rtmv` with `sales_new` as the base table.

   The `ON QUERY COMPUTATION` clause indicates that a real-time materialized view is created. The refresh mode specified is log-based fast refresh. Query rewrite is enabled for the real-time materialized view.

```
CREATE MATERIALIZED VIEW my_rtmv
REFRESH FAST
ENABLE ON QUERY COMPUTATION
ENABLE QUERY REWRITE
AS
SELECT prod_id, cust_id, channel_id, sum(quantity_sold) sum_q,
count(quantity_sold) cnt_q, avg(quantity_sold) avg_q,
        sum(amount_sold) sum_a, count(amount_sold) cnt_a, avg(amount_sold)
avg_a
FROM sales_new
GROUP BY prod_id, cust_id, channel_id;
```

3. Insert a row into `sales_new`, the base table of the real-time materialized view and commit this change.

```
INSERT INTO sales_new (prod_id, cust_id, time_id, channel_id, promo_id,
quantity_sold, amount_sold)
    VALUES (116,100450, sysdate,9,9999,10,350);

COMMIT;
```

4. Query the real-time materialized view directly to display data for the row that was added to the real-time materialized view's base table in the previous step.

```
SELECT * from my_rtmv
WHERE prod_id = 116 AND cust_id=100450 AND channel_id = 9;

PROD_ID   CUST_ID   CHANNEL_ID   SUM_Q   CNT_Q    AVG_Q    SUM_A
CNT_A    AVG_A
-------   -------   ----------   -----   -----    -----    -----
-----    ------
116       100450    9            1       1        1        11.99
1        11.99
```

   Note that the query result does not display the updated value for this data. This is because the real-time materialized view has not yet been refreshed with the changes made to its base table.

5. Include the `FRESH_MV` hint while querying the real-time materialized view to display the row updated in the base table.

```
SELECT /*+ fresh_mv */ * FROM my_rtmv
WHERE prod_id = 116 AND cust_id=100450 AND channel_id = 9;

PROD_ID   CUST_ID   CHANNEL_ID  SUM_Q   CNT_Q   AVG_Q    SUM_A    CNT_A
AVG_A
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
-------   -------   ----------  -----   -----   -----    -----    -----
------
116       100450    9           11      2       5.5      361.99   2
180.995
```

Notice that this time the updated row is displayed. This is because the FRESH_MV hint triggers on-query computation for the real-time materialized view and recomputed the fresh data.

## 6.7.7 Listing Real-time Materialized Views

The ON_QUERY_COMPUTATION column in the data dictionary views ALL_MVIEWS, DBA_MVIEWS, and USER_MVIEWS indicates if a materialized view is a real-time materialized view.

A value of Y in the ON_QUERY_COMPUTATION column indicates a real-time materialized view.

To list all real-time materialized views in your user schema:

- Query the USER_MVIEWS view and display details of the materialized view with the ON_QUERY_COMPUTATION column set to Y.

**Example 6-12 Listing Real-time Materialized Views in the Current User's Schema**

```
SELECT owner, mview_name, rewrite_enabled, staleness
FROM user_mviews
WHERE on_query_computation = 'Y';

OWNER   MVIEW_NAME        REWRITE_ENABLED   STALENESS
------  ------------    ------------------ ------------
SH      SALES_RTMV          N                FRESH
SH      MAV_SUM_SALES       Y                FRESH
SH      MY_SUM_SALES_RTMV   Y                FRESH
SH      NEW_SALES_RTMV      Y                STALE
```

## 6.7.8 Improving Real-time Materialized Views Performance

To obtain better performance for user queries that use a real-time materialized view, you can follow certain guidelines.

Use the following guidelines with real-time materialized views:

- Frequently refresh real-time materialized views to enhance the performance of queries that may use these real-time materialized views.

  Since real-time materialized views work by combining the delta changes to the base tables with the existing materialized view data, query response time is enhanced when the delta changes to be computed are small. With more outstanding DML operations, on-query computation can become more complex (and expensive), up to the point where direct base table access can become more efficient (in case of query rewrite).

- Collect statistics for the base tables, the real-time materialized view, and the materialized view logs to enable the optimizer to accurately determine the cost of a query.

  For query rewrite, the cost-based rewrite mechanism uses the optimizer to determine whether the rewritten query should be used. The optimizer uses statistics to determine the cost.


Copyright © 2001, 2025, Oracle and/or its affiliates.

7

# Refreshing Materialized Views

This chapter discusses how to refresh materialized views, which is a key element in maintaining good performance and consistent data when working with materialized views in a data warehousing environment.

This chapter includes the following sections:


- About Refreshing Materialized Views
- Tips for Refreshing Materialized Views
- Using Materialized Views with Partitioned Tables
- Using Partitioning to Improve Data Warehouse Refresh
- Optimizing DML Operations During Refresh


## 7.1 About Refreshing Materialized Views

The database maintains data in materialized views by refreshing them after changes to the base tables.

Performing a refresh operation requires temporary space to rebuild the indexes and can require additional space for performing the refresh operation itself. Some sites might prefer not to refresh all of their materialized views at the same time: as soon as some underlying detail data has been updated, all materialized views using this data become stale. Therefore, if you defer refreshing your materialized views, you can either rely on your chosen rewrite integrity level to determine whether or not a stale materialized view can be used for query rewrite, or you can temporarily disable query rewrite with an `ALTER SYSTEM SET QUERY_REWRITE_ENABLED = FALSE` statement. After refreshing the materialized views, you can re-enable query rewrite as the default for all sessions in the current database instance by specifying `ALTER SYSTEM SET QUERY_REWRITE_ENABLED` as `TRUE`. Refreshing a materialized view automatically updates all of its indexes. In the case of full refresh, this requires temporary sort space to rebuild all indexes during refresh. This is because the full refresh truncates or deletes the table before inserting the new full data volume. If insufficient temporary space is available to rebuild the indexes, then you must explicitly drop each index or mark it `UNUSABLE` prior to performing the refresh operation.

**About Types of Refresh for Materialized Views**

There are three incremental refresh methods:

- log-based refresh
- partition change tracking (PCT) refresh
- logical partition change tracking (LPCT) refresh

When there have been Partition Maintenance Operations (PMOPS) on the base tables, PCT is the only incremental refresh method that can be used.

The incremental refresh is commonly called `FAST` refresh because it usually performs faster than the complete refresh.



Copyright © 2001, 2025, Oracle and/or its affiliates.

A complete refresh occurs when the materialized view is initially created when it is defined as `BUILD IMMEDIATE`, unless the materialized view references a prebuilt table or is defined as `BUILD DEFERRED`. Users can perform a complete refresh at any time after the materialized view is created. The complete refresh involves executing the query that defines the materialized view. This process can be slow, especially if the database must read and process huge amounts of data.

An incremental refresh eliminates the need to rebuild materialized views from scratch. Thus, processing only the changes can result in a very fast refresh time. Materialized views can be refreshed either on demand or at regular time intervals. Alternatively, materialized views in the same database as their base tables can be refreshed whenever a transaction commits its changes to the base tables.

For materialized views that use the log-based fast refresh method, a materialized view log and/or a direct loader log keep a record of changes to the base tables. A materialized view log is a schema object that records changes to a base table so that a materialized view defined on the base table can be refreshed incrementally. Each materialized view log is associated with a single base table. The materialized view log resides in the same database and schema as its base table.

LPCT is similar to PCT, although LPCT requires a logical partitioning scheme rather than a physical partitioning on the base table. As in the case of a PCT enabled materialized view, an LPCT enabled materialized view does not require a materialized view log for refresh operations. A base table on which a materialized view is defined is logically partitioned using key ranges. Because there is no physical partitioning on the table using the LPCT partitioning key, the table rows belonging to an LPCT key range are not segregated into separate physical partitions. The base table can be physically non-partitioned, or physically partitioned on a key that is different from the logical partition key.

The PCT refresh method can be used if the modified base tables are partitioned and the modified base table partitions can be used to identify the affected partitions or portions of data in the materialized view. This method removes all data in the affected materialized view partitions or affected portions of data and recompute them from scratch.

Note that if a table is already physically partitioned, LPCT cannot be defined on the same physical partitioning key. Furthermore, any PMOPS on a table requires full refresh before LPCT refresh can be used again.

### About Refresh Modes for Materialized Views

When creating a materialized view, you have the option of specifying whether the refresh occurs `ON DEMAND` or `ON COMMIT`.

If you anticipate performing insert, update or delete operations on tables referenced by a materialized view concurrently with the refresh of that materialized view, and that materialized view includes joins and aggregation, Oracle recommends you use `ON COMMIT` fast refresh rather than `ON DEMAND` fast refresh.

In the case of `ON COMMIT`, the materialized view is changed every time a transaction commits, thus ensuring that the materialized view always contains the latest data. Alternatively, you can control the time when refresh of the materialized views occurs by specifying `ON DEMAND`. In the case of `ON DEMAND` materialized views, the refresh can be performed with refresh methods provided in either the `DBMS_SYNC_REFRESH` or the `DBMS_MVIEW` packages:

- The `DBMS_SYNC_REFRESH` package contains the APIs for synchronous refresh. For details, see Synchronous Refresh.


Copyright © 2001, 2025, Oracle and/or its affiliates.

- The DBMS_MVIEW package contains the APIs whose usage is described in this chapter. There are three basic types of refresh operations: complete refresh, fast refresh, and partition change tracking (PCT) refresh.

The DBMS_MVIEW package contains three APIs for performing refresh operations:

- DBMS_MVIEW.REFRESH

  Refresh one or more materialized views.

- DBMS_MVIEW.REFRESH_ALL_MVIEWS

  Refresh all materialized views.

- DBMS_MVIEW.REFRESH_DEPENDENT

  Refresh all materialized views that depend on a specified primary table or materialized view or list of primary tables or materialized views.

**How to Refresh Materialized Views?**

For each of these refresh options, you have two techniques for how the refresh is performed, namely in-place refresh and out-of-place refresh. The in-place refresh executes the refresh statements directly on the materialized view. The out-of-place refresh creates one or more outside tables and executes the refresh statements on the outside tables and then switches the materialized view or affected materialized view partitions with the outside tables. Both in-place refresh and out-of-place refresh achieve good performance in certain refresh scenarios. However, the out-of-place refresh enables high materialized view availability during refresh, especially when refresh statements take a long time to finish.

The out-of-place mechanism, called synchronous refresh, targets the common usage scenario in the data warehouse where both fact tables and their materialized views are partitioned in the same way or their partitions are related by a functional dependency.

The refresh approach enables you to keep a set of tables and the materialized views defined on them to be always in sync. In this refresh method, the user does not directly modify the contents of the base tables but must use the APIs provided by the synchronous refresh package that will apply these changes to the base tables and materialized views at the same time to ensure their consistency. The synchronous refresh method is well-suited for data warehouses, where the loading of incremental data is tightly controlled and occurs at periodic intervals.

**See Also**

- About the Out-of-Place Refresh Option

## 7.1.1 About Complete Refresh for Materialized Views

A complete refresh occurs when the materialized view is initially defined as BUILD IMMEDIATE, unless the materialized view references a prebuilt table. For materialized views using BUILD DEFERRED, a complete refresh must be requested before it can be used for the first time. A complete refresh may be requested at any time during the life of any materialized view. The refresh involves reading the detail tables to compute the results for the materialized view. This can be a very time-consuming process, especially if there are huge amounts of data to be read and processed. Therefore, you should always consider the time required to process a complete refresh before requesting it.


Copyright © 2001, 2025, Oracle and/or its affiliates.

There are, however, cases when the only refresh method available for an already built materialized view is complete refresh because the materialized view does not satisfy the conditions specified in the following section for a fast refresh.

## 7.1.2 About Fast Refresh for Materialized Views

Most data warehouses have periodic incremental updates to their detail data. As described in "About Materialized View Schema Design", you can use the SQL*Loader or any bulk load utility to perform incremental loads of detail data. Fast refresh of your materialized views is usually efficient, because instead of having to recompute the entire materialized view, the changes are applied to the existing data. Thus, processing only the changes can result in a very fast refresh time.

## 7.1.3 About Partition Change Tracking (PCT) Refresh for Materialized Views

Partition Change Tracking is the capability to leverage the knowledge about changes within individual partitions on a table contained in a materialized view for a potentially more efficient materialized refresh of the materialized view.

When there have been some partition maintenance operations on the detail tables, Partition Change Tracking is the only method of fast refresh that can be used. PCT-based refresh on a materialized view is enabled only if all the conditions described in "About Partition Change Tracking" are satisfied.

In the absence of partition maintenance operations on detail tables, when you request a `FAST` method (`method => 'F'`) of refresh through procedures in `DBMS_MVIEW` package, Oracle uses a heuristic rule to try log-based rule fast refresh before choosing PCT refresh. Similarly, when you request a `FORCE` method (`method => '?'`), Oracle chooses the refresh method based on the following order: log-based fast refresh, PCT refresh, LPCT refresh, and complete refresh. Alternatively, you can request the PCT method (`method => 'P'`), and Oracle uses the PCT method provided all PCT requirements are satisfied.

Oracle can use `TRUNCATE PARTITION` on a materialized view if it satisfies the conditions in "Benefits of Partitioning a Materialized View" and hence, make the PCT refresh process more efficient.

**See Also**

- "About Partition Change Tracking" for more information regarding partition change tracking

## 7.1.4 About Logical Partition Change Tracking (LPCT) Refresh for Materialized Views

Logical Partition Change Tracking is the capability to leverage the knowledge about changes within individual logical partitions on a table contained in a materialized view for a potentially more efficient materialized refresh of the materialized view. Unlike Partition Change Tracking which relies on the physical partitioning of tables, you define the logical partitions of your tables independently of any existing or non-existing partitioning schema of a table.

With LPCT, materialized view staleness can be tracked at the granularity of the logical partitions, and consequently the Query Rewrite engine can use the data in fresh logical partitions of the materialized view, even if some parts of the materialized may be stale. As a


Copyright © 2001, 2025, Oracle and/or its affiliates.

result, the materialized views becomes more usable. In many real-world applications, this results in significant improvement to query performance due to the fine-grained query rewrite. LPCT can perform refresh operations targeted at stale logical partitions only, which avoids complete re-loading the data.

Without LPCT, a materialized view on a non-partitioned table is either completely stale or completely fresh. When a materialized view is determined to be stale, it cannot be used for query rewrites even though the data needed by the query may be fresh.

The LPCT tracking mechanism records and consolidates the change statistics based on given LPCT key. Adjacent change data is grouped into a logical partition. During refresh, instead of using materialized view log, LPCT looks at the changes in the logical partitions. These are limited to a single column and RANGE or INTERVAL logical partitions. LPCT does not enforce partitioning of the base table. It tracks changes within defined RANGE or INTERVAL partitions in new dictionary tables. The syntax for refreshing a materialized view using LPCT refresh is as follows.

```
execute DBMS_MVIEW.REFRESH(<mview_name>,'L');
```

Unlike PCT, which must be specified at table creation, LPCT can be created, modified, or dropped on the base table at any time, independent of table creation. The base table can be partitioned or non-partitioned, which makes it more adaptable than PCT. Because the LPCT framework requires metadata only and no changes to the base table, it costs less physical overhead than PCT. LPCT can be combined with PCT to identify more fine-grained stale ranges and allow for more query rewrites and faster materialized view refresh. If a base table is both physically partitioned and also has an LPCT defined on a different partitioning key, then any dependent materialized views can be refreshed using a combined LPCT and PCT methods by specifying the `'L'` option in the call to `DBMS_MVIEW.REFRESH()`.

In tracking, LPCT is more lightweight than a materialized view log because it does not log each modified row on base table. Since it does not need to scan the entire materialized view log and join with base table for fresh data, an LPCT refresh outperforms log-based fast refresh, especially if modified rows are relatively large.

Logical Partition Change Tracking (LPCT) outperforms log-based refresh when modified rows are relatively large. An LPCT refresh can be combined with log-based refresh to improve the efficiency even more by targeting at more precise rows.

### See Also

- "About Partition Change Tracking" for more information regarding partition change tracking
- `USER_MVIEW_DETAIL_LOGICAL_PARTITION` in the *Oracle AI Database Reference* for views to identify staleness corresponding to the logical partitions of base tables. Also see the corresponding descriptions of `ALL_MVIEW_DETAIL_LOGICAL_PARTITION` and `DBA_MVIEW_DETAIL_LOGICAL_PARTITION`.
- See `DBMS_MVIEW` in the *PL/SQL Packaging and Types Reference* for details on how to use the DBMS_MVIEW package with logical partitions.


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 7.1.5 About the Out-of-Place Refresh Option

Beginning with Oracle Database 12*c* Release 1, a new refresh option is available to improve materialized view refresh performance and availability. This refresh option is called out-of-place refresh because it uses outside tables during refresh as opposed to the existing "in-place" refresh that directly applies changes to the materialized view container table. The out-of-place refresh option works with all existing refresh methods, such as `FAST` (`'F'`), `COMPLETE` (`'C'`), `PCT` (`'P'`), and `FORCE` (`'?'`). Out-of-place refresh is particularly effective when handling situations with large amounts of data changes, where conventional DML statements do not scale well. It also enables you to achieve a very high degree of availability because the materialized views that are being refreshed can be used for direct access and query rewrite during the execution of refresh statements. In addition, it helps to avoid potential problems such as materialized view container tables becoming fragmented over time or intermediate refresh results being seen.

In out-of-place refresh, the entire or affected portions of a materialized view are computed into one or more outside tables. For partitioned materialized views, if partition level change tracking is possible, and there are local indexes defined on the materialized view, the out-of-place method also builds the same local indexes on the outside tables. This refresh process is completed by either switching between the materialized view and the outside table or partition exchange between the affected partitions and the outside tables. Note that query rewrite is not supported during the switching or partition exchange operation. During refresh, the outside table is populated by direct load, which is efficient.

This section contains the following topics:

- Types of Out-of-Place Refresh
- Restrictions and Considerations with Out-of-Place Refresh

### 7.1.5.1 Types of Out-of-Place Refresh

There are three types of out-of-place refresh:

- out-of-place fast refresh

  This offers better availability than in-place fast refresh. It also offers better performance when changes affect a large part of the materialized view.

- out-of-place PCT refresh

  This offers better availability than in-place PCT refresh. There are two different approaches for partitioned and non-partitioned materialized views. If truncation and direct load are not feasible, you should use out-of-place refresh when the changes are relatively large. If truncation and direct load are feasible, in-place refresh is preferable in terms of performance. In terms of availability, out-of-place refresh is always preferable.

- out-of-place complete refresh

  This offers better availability than in-place complete refresh.

Using the refresh interface in the `DBMS_MVIEW` package, with `method = ?` and `out_of_place = true`, out-of-place fast refresh are attempted first, then out-of-place PCT refresh, and finally out-of-place complete refresh. An example is the following:

```
DBMS_MVIEW.REFRESH('CAL_MONTH_SALES_MV', method => '?',
   atomic_refresh => FALSE, out_of_place => TRUE);
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

### 7.1.5.2 Restrictions and Considerations with Out-of-Place Refresh

Out-of-place refresh has all the restrictions that apply when using the corresponding in-place refresh. In addition, it has the following restrictions:

- Only materialized join views and materialized aggregate views are allowed
- No `ON COMMIT` refresh is permitted
- No remote materialized views, cube materialized views, object materialized views are permitted
- No `LOB` columns are permitted
- Not permitted if materialized view logs, triggers, or constraints (except `NOT NULL`) are defined on the materialized view
- Not permitted if the materialized view contains the `CLUSTERING` clause
- Not permitted if the materialized view has security policy defined on it.
- Not applied to complete refresh within a `CREATE` or `ALTER MATERIALIZED VIEW` session or an `ALTER TABLE` session
- Atomic mode is not permitted. If you specify `atomic_refresh` as `TRUE` and `out_of_place` as `TRUE`, an error is displayed

For out-of-place PCT refresh, there is the following restriction:

- No `UNION ALL` or grouping sets are permitted

For out-of-place fast refresh, there are the following restrictions:

- No `UNION ALL`, grouping sets or outer joins are permitted
- Not allowed for materialized join views when more than one base table is modified with mixed DML statements

Out-of-place refresh requires additional storage for the outside table and the indexes for the duration of the refresh. Thus, you must have enough available tablespace or auto extend turned on.

The partition exchange in out-of-place PCT refresh impacts the global index on the materialized view. Therefore, if there are global indexes defined on the materialized view container table, Oracle disables the global indexes before doing the partition exchange and rebuild the global indexes after the partition exchange. This rebuilding is additional overhead.

## 7.1.6 About ON COMMIT Refresh for Materialized Views

A materialized view can be refreshed automatically using the `ON COMMIT` method. Therefore, whenever a transaction commits which has updated the tables on which a materialized view is defined, those changes are automatically reflected in the materialized view. The advantage of using this approach is you never have to remember to refresh the materialized view. The only disadvantage is the time required to complete the commit will be slightly longer because of the extra processing involved. However, in a data warehouse, this should not be an issue because there is unlikely to be concurrent processes trying to update the same table.

## 7.1.7 About ON STATEMENT Refresh for Materialized Views

A materialized view that uses the `ON STATEMENT` refresh mode is automatically refreshed every time a DML operation is performed on any of the materialized view's base tables.


Copyright © 2001, 2025, Oracle and/or its affiliates.

With the ON STATEMENT refresh mode, any changes to the base tables are immediately reflected in the materialized view. There is no need to commit the transaction or maintain materialized view logs on the base tables. If the DML statements are subsequently rolled back, then the corresponding changes made to the materialized view are also rolled back.

To use the ON STATEMENT refresh mode, a materialized view must be fast refreshable. An index is automatically created on ROWID column of the fact table to improve fast refresh performance.

The advantage of the ON STATEMENT refresh mode is that the materialized view is always synchronized with the data in the base tables, without the overhead of maintaining materialized view logs. However, this mode may increase the time taken to perform a DML operation because the materialized view is being refreshed as part of the DML operation.

> **See Also**
>
> *Oracle AI Database SQL Language Reference* for the ON STATEMENT clause restrictions

**Example 7-1 Creating a Materialized View with ON STATEMENT Refresh**

This example creates a materialized view sales_mv_onstat that uses the ON STATEMENT refresh mode and is based on the sh.sales, sh.customers, and sh.products tables. The materialized view is automatically refreshed when a DML operation is performed on any of the base tables. No commit is required after the DML operation to refresh the materialized view.

```
CREATE MATERIALIZED VIEW sales_mv_onstat
REFRESH FAST ON STATEMENT USING TRUSTED CONSTRAINT
AS
SELECT s.rowid sales_rid, c.cust_first_name first_name, c.cust_last_name
last_name,
       p.prod_name prod_name,
    s.quantity_sold quantity_sold, s.amount_sold amount_sold
FROM sh.sales s, sh.customers c, sh.products p
WHERE s.cust_id = c.cust_id and s.prod_id = p.prod_id;
```

## 7.1.8 About Manual Refresh Using the DBMS_MVIEW Package

When a materialized view is refreshed ON DEMAND, one of four refresh methods can be specified as shown in the following table. You can define a default option during the creation of the materialized view. Table 7-1 details the refresh options.

**Table 7-1 ON DEMAND Refresh Methods**

| Refresh Option | Parameter | Description |
|---|---|---|
| COMPLETE | C | Refreshes by recalculating the defining query of the materialized view. |


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Table 7-1 (Cont.) ON DEMAND Refresh Methods**

| Refresh Option | Parameter | Description |
|---|---|---|
| FAST | F | Refreshes by incrementally applying changes to the materialized view.<br>For local materialized views, it chooses the refresh method which is estimated by optimizer to be most efficient. The refresh methods considered are log-based FAST and FAST_PCT. |
| FAST_PCT | P | Refreshes by recomputing the rows in the materialized view affected by changed physical partitions in the detail tables. |
| LPCT | L | Refreshes by recomputing the rows in the materialized view affected by changed logical partitions in the detail tables. |
| FORCE | ? | Attempts a fast refresh. If that is not possible, it does a complete refresh.<br>For local materialized views, it chooses the refresh method which is estimated by optimizer to be most efficient. The refresh methods considered are log based FAST, FAST_PCT, FAST_LPCT, and COMPLETE. |

Three refresh procedures are available in the DBMS_MVIEW package for performing ON DEMAND refresh. Each has its own unique set of parameters.

> **See Also**
>
> *Oracle AI Database PL/SQL Packages and Types Reference* for detailed information about the DBMS_MVIEW package

## 7.1.9 Refreshing Specific Materialized Views with REFRESH

Use the DBMS_MVIEW.REFRESH procedure to refresh one or more materialized views. Some parameters are used only for replication, so they are not mentioned here. The required parameters to use this procedure are:

- The comma-delimited list of materialized views to refresh
- The refresh method: F-Fast, P-Fast_PCT, L-FAST_LPCT, ?-Force, C-Complete
- The rollback segment to use
- Refresh after errors (TRUE or FALSE)

  A Boolean parameter. If set to TRUE, the number_of_failures output parameter is set to the number of refreshes that failed, and a generic error message indicates that failures occurred. The alert log for the instance gives details of refresh errors. If set to FALSE, which is the default, then refresh stops after it encounters the first error, and any remaining materialized views in the list are not refreshed.
- The following four parameters are used by the replication process. For warehouse refresh, set them to FALSE, 0,0,0.
- Atomic refresh (TRUE or FALSE)


Copyright © 2001, 2025, Oracle and/or its affiliates.

If set to TRUE, then all refreshes are done in one transaction. If set to FALSE, then each of the materialized views is refreshed non-atomically in separate transactions. If set to FALSE, Oracle can optimize refresh by using parallel DML and truncate DDL on a materialized views. When a materialized view is refreshed in atomic mode, it is eligible for query rewrite if the rewrite integrity mode is set to stale_tolerated. Atomic refresh cannot be guaranteed when refresh is performed on nested views.

- Whether to use out-of-place refresh

  This parameter works with all existing refresh methods (F, P, C, ?). So, for example, if you specify F and out_of_place = true, then an out-of-place fast refresh is attempted. Similarly, if you specify P and out_of_place = true, then out-of-place PCT refresh is attempted.

For example, to perform a fast refresh on the materialized view cal_month_sales_mv, the DBMS_MVIEW package would be called as follows:

```
DBMS_MVIEW.REFRESH('CAL_MONTH_SALES_MV', 'F', '', TRUE, FALSE, 0,0,0,
   FALSE, FALSE);
```

Multiple materialized views can be refreshed at the same time, and they do not all have to use the same refresh method. To give them different refresh methods, specify multiple method codes in the same order as the list of materialized views (without commas). For example, the following specifies that cal_month_sales_mv be completely refreshed and fweek_pscat_sales_mv receive a fast refresh:

```
DBMS_MVIEW.REFRESH('CAL_MONTH_SALES_MV, FWEEK_PSCAT_SALES_MV', 'CF', '',
  TRUE, FALSE, 0,0,0, FALSE, FALSE);
```

If the refresh method is not specified, the default refresh method as specified in the materialized view definition is used.

## 7.1.10 Refreshing All Materialized Views with REFRESH_ALL_MVIEWS

An alternative to specifying the materialized views to refresh is to use the procedure DBMS_MVIEW.REFRESH_ALL_MVIEWS. This procedure refreshes all materialized views. If any of the materialized views fails to refresh, then the number of failures is reported.

The parameters for this procedure are:

- The number of failures (this is an OUT variable)
- The refresh method: F-Fast, P-Fast_PCT, L-FAST_ LPCT, ?-Force, C-Complete
- Refresh after errors (TRUE or FALSE)

  A Boolean parameter. If set to TRUE, the number_of_failures output parameter is set to the number of refreshes that failed, and a generic error message indicates that failures occurred. The alert log for the instance gives details of refresh errors. If set to FALSE, the default, then refresh stops after it encounters the first error, and any remaining materialized views in the list is not refreshed.

- Atomic refresh (TRUE or FALSE)

  If set to TRUE, then all refreshes are done in one transaction. If set to FALSE, then each of the materialized views is refreshed non-atomically in separate transactions. If set to FALSE, Oracle can optimize refresh by using parallel DML and truncate DDL on a materialized views. When a materialized view is refreshed in atomic mode, it is eligible for query rewrite if the rewrite integrity mode is set to stale_tolerated. Atomic refresh cannot be guaranteed when refresh is performed on nested views.


Copyright © 2001, 2025, Oracle and/or its affiliates.

- Whether to use out-of-place refresh

  This parameter works with all existing refresh method (F, P, C, ?). So, for example, if you specify F and out_of_place = true, then an out-of-place fast refresh is attempted. Similarly, if you specify P and out_of_place = true, then out-of-place PCT refresh is attempted.

An example of refreshing all materialized views is the following:

```
DBMS_MVIEW.REFRESH_ALL_MVIEWS(failures,'C','', TRUE, FALSE, FALSE);
```

## 7.1.11 Refreshing Dependent Materialized Views with REFRESH_DEPENDENT

The third procedure, DBMS_MVIEW.REFRESH_DEPENDENT, refreshes only those materialized views that depend on a specific table or list of tables. For example, suppose the changes have been received for the orders table but not for customer payments. The refresh dependent procedure can be called to refresh only those materialized views that reference the orders table.

The parameters for this procedure are:

- The number of failures (this is an OUT variable)
- The dependent table
- The refresh method: F-Fast, P-Fast_PCT, L-FAST_LPCT, ?-Force, C-Complete
- The rollback segment to use
- Refresh after errors (TRUE or FALSE)

  A Boolean parameter. If set to TRUE, the number_of_failures output parameter is set to the number of refreshes that failed, and a generic error message indicates that failures occurred. The alert log for the instance gives details of refresh errors. If set to FALSE, the default, then refresh stops after it encounters the first error, and any remaining materialized views in the list are not refreshed.

- Atomic refresh (TRUE or FALSE)

  If set to TRUE, then all refreshes are done in one transaction. If set to FALSE, then each of the materialized views is refreshed non-atomically in separate transactions. If set to FALSE, Oracle can optimize refresh by using parallel DML and truncate DDL on a materialized views. When a materialized view is refreshed in atomic mode, it is eligible for query rewrite if the rewrite integrity mode is set to stale_tolerated. Atomic refresh cannot be guaranteed when refresh is performed on nested views.

- Whether it is nested or not

  If set to TRUE, refresh all the dependent materialized views of the specified set of tables based on a dependency order to ensure the materialized views are truly fresh with respect to the underlying base tables.

- Whether to use out-of-place refresh

  This parameter works with all existing refresh methods (F, P, C, ?). So, for example, if you specify F and out_of_place = true, then an out-of-place fast refresh is attempted. Similarly, if you specify P and out_of_place = true, then out-of-place PCT refresh is attempted.

To perform a full refresh on all materialized views that reference the customers table, specify:

```
DBMS_MVIEW.REFRESH_DEPENDENT(failures, 'CUSTOMERS', 'C', '', FALSE, FALSE, FALSE);
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 7.1.12 About Using Job Queues for Refresh

Job queues can be used to refresh multiple materialized views in parallel. If queues are not available, fast refresh sequentially refreshes each view in the foreground process. To make queues available, you must set the `JOB_QUEUE_PROCESSES` parameter. This parameter defines the number of background job queue processes and determines how many materialized views can be refreshed concurrently. Oracle tries to balance the number of concurrent refreshes with the degree of parallelism of each refresh. The order in which the materialized views are refreshed is determined by dependencies imposed by nested materialized views and potential for efficient refresh by using query rewrite against other materialized views (See "Scheduling Refresh of Materialized Views" for details). This parameter is only effective when `atomic_refresh` is set to `FALSE`.

If the process that is executing `DBMS_MVIEW.REFRESH` is interrupted or the instance is shut down, any refresh jobs that were executing in job queue processes are requeued and continue running. To remove these jobs, use the `DBMS_JOB.REMOVE` procedure.

**See Also**

- *Oracle AI Database PL/SQL Packages and Types Reference* for detailed information about the `DBMS_JOB` package

## 7.1.13 When Fast Refresh is Possible

Not all materialized views may be fast refreshable. Therefore, use the package `DBMS_MVIEW.EXPLAIN_MVIEW` to determine what refresh methods are available for a materialized view.

If you are not sure how to make a materialized view fast refreshable, you can use the `DBMS_ADVISOR.TUNE_MVIEW` procedure, which provides a script containing the statements required to create a fast refreshable materialized view.

**See Also**

- *Oracle AI Database SQL Tuning Guide*
- Basic Materialized Views for further information about the `DBMS_MVIEW` package

## 7.1.14 Refreshing Materialized Views Based on Approximate Queries

Oracle AI Database performs fast refresh for materialized views that are defined using approximate queries.

Approximate queries contain SQL functions that return approximate results. Refreshing materialized views containing approximate queries depends on the DML operation that is performed on the base tables of the materialized view.

- For insert operations, fast refresh is used for materialized views containing detailed percentiles.


Copyright © 2001, 2025, Oracle and/or its affiliates.

- For delete operations or any DML operation that leads to deletion (such as `UPDATE` or `MERGE`), fast refresh is used for materialized views containing approximate aggregations only if the materialized view does not contain a `WHERE` clause.

Materialized view logs must exist on all base tables of a materialized view that needs to be fast refreshed.

- To refresh a materialized view that is based on an approximate query:

  Run the `DBMS_REFRESH.REFRESH` procedure to perform a fast refresh of the materialized view

**Example 7-2 Refreshing Materialized Views Based on Approximate Queries**

The following example performs a fast refresh of the materialized view `percentile_per_pdt` that is based on an approximate query.

```
exec DBMS_MVIEW.REFRESH('percentile_per_pdt', method => 'F');
```

**See Also**

- About Approximate Query Processing
- Creating Materialized Views Based on Approximate Queries
- Query Rewrite and Materialized Views Based on Approximate Queries

## 7.1.15 About Concurrent Refresh of On-Commit Materialized Views

As of Oracle AI Database 26ai, on-commit materialized views can be refreshed concurrently.

Concurrent materialized view refresh means that multiple sessions are able to refresh the same on-commit atomic materialized view at the same time.

When concurrent refresh is enabled, multiple sessions which perform DML on a base table can refresh the materialized view concurrently. There is no limit on the number of concurrent sessions.

When concurrent refresh is disabled, materialized view refreshes are serialized. In this case, multiple sessions cannot concurrently refresh a materialized view. Only one refresh session at a time is able to update the materialized view. Other refreshes are blocked until the current session is completed. Then the next session is allowed to continue.

**Best Use Cases for Concurrent On-Commit Materialized View Refresh**

This capability is useful when you perform on-commit materialized view refreshes in OLTP and also in cases which many concurrent DML transactions update the fact table only.

**Enabling Concurrent On-Commit Materialized View Refresh**

Concurrent materialized view refresh is disabled by default. You can enable or disable concurrent refresh in a `CREATE MATERIALIZED VIEW` or `ALTER MATERIALIZED VIEW` statement:

```
{ ENABLE | DISABLE } CONCURRENT REFRESH
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

For example:

```
CREATE MATERIALIZED VIEW "T1"."MV1" ("C1", "C2")
  SEGMENT CREATION DEFERRED
  REFRESH FAST ON COMMIT
  WITH PRIMARY KEY USING DEFAULT LOCAL ROLLBACK SEGMENT
  USING ENFORCED CONSTRAINTS DISABLE ON QUERY COMPUTATION DISABLE QUERY
REWRITE
ENABLE CONCURRENT REFRESH
  AS SELECT "TB1"."C1" "C1","TB1"."C2" "C2" FROM "TB1" "TB1
```

#### Conditions that Allow Concurrent On-Commit Materialized View Refresh

The following conditions determine if concurrent refresh can proceed.

- Concurrent refresh is enabled. (Note only on-commit refresh materialized views can be enabled.)
- All concurrent DML sessions update the same base table.
- The materialized view rows updated in different refresh sessions do not overlap.

#### Limitations

You cannot enable concurrent refresh for on-commit fast refresh of materialized view DDLs. Only on-commit materialized views can enable concurrent refresh.

#### How to Determine if an On-Commit Materialized View has Concurrent Refresh Enabled

You can check the view ALL_MVIEWS to see all of the properties of a materialzed view, including whether nor not concurrent refresh is enabled.

> **See Also**
>
> The CREATE MATERIALIZED VIEW statement description in the *Oracle AI Database SQL Language Reference* shows { ENABLE | DISABLE } CONCURRENT REFRESH in its full context.

See DBMS_MVIEW_REFRESH and other APIs related to refresh of materialized views in the *Database PL/SQL Packages and Types Reference* .

## 7.1.16 About Refreshing Dependent Materialized Views During Online Table Redefinition

While redefining a table online using the `DBMS_REDEFINITION` package, you can perform incremental refresh of fast refreshable materialized views that are dependent on the table being redefined.

Prior to Oracle Database 12*c* Release 2 (12.2), to refresh dependent materialized views on tables undergoing redefinition, you must execute complete refresh manually after the redefinition process completes.

To incrementally refresh dependent materialized views during online table redefinition, set the `refresh_dep_mviews` parameter in the `DBMS_REDEFINITON.REDEF_TABLE` procedure to Y . Dependent materialized views can be refreshed during online table redefinition only if the


Copyright © 2001, 2025, Oracle and/or its affiliates.

materialized view is fast refreshable and is not a ROWID-based materialized view or materialized join view. Materialized views that do not follow these restrictions are not refreshed.

Consider the table my_sales that has the following dependent materialized views:

- my_sales_pk_mv: fast refreshable primary key-based materialized view
- my_sales_rid_mv: fast refreshable ROWID-based materialized view
- my_sales_mjv: fast refreshable materialized join view
- my_sales_mav: fast refreshable materialized aggregate view
- my_sales_rmv: only fully-refreshable materialized view

When you run the following command, fast refresh is performed only for the my_sales_pk_mv and my_sales_mav materialized views:

```
DBMS_REDEFINITION.REDEF_TABLE(
uname => 'SH',
tname => 'MY_SALES',
table_compression_type => 'ROW STORE COMPRESS ADVANCED',
refresh_dep_mviews => 'Y');
```

**See Also**

*Oracle AI Database Administrator's Guide*

## 7.1.17 Recommended Initialization Parameters for Parallelism

The following initialization parameters need to be set properly for parallelism to be effective:

- PARALLEL_MAX_SERVERS should be set high enough to take care of parallelism. You must consider the number of child processes needed for the refresh statement. For example, with a degree of parallelism of eight, you need 16 child processes.
- PGA_AGGREGATE_TARGET should be set for the instance to manage the memory usage for sorts and joins automatically. If the memory parameters are set manually, SORT_AREA_SIZE should be less than HASH_AREA_SIZE.
- OPTIMIZER_MODE should equal all_rows.

Remember to analyze all tables and indexes for better optimization.

**See Also**

*Oracle AI Database VLDB and Partitioning Guide*

## 7.1.18 Monitoring a Refresh

While a job is running, you can query the V$SESSION_LONGOPS view to tell you the progress of each materialized view being refreshed.

```
SELECT * FROM V$SESSION_LONGOPS;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

To look at the progress of which jobs are on which queue, use:

```
SELECT * FROM DBA_JOBS_RUNNING;
```

## 7.1.19 Checking the Status of a Materialized View

These are views that enable you to verify the status of table partitions and determine which ranges of materialized view data are fresh or stale:

- `*_MVIEWS`

  To determine partition change tracking (PCT) information for the materialized view.

- `*_MVIEW_DETAIL_RELATIONS`

  To display partition information for the detail table a materialized view is based on.

- `*_MVIEW_DETAIL_PARTITION`

  To determine which partitions are fresh. (Physical partitions only.)

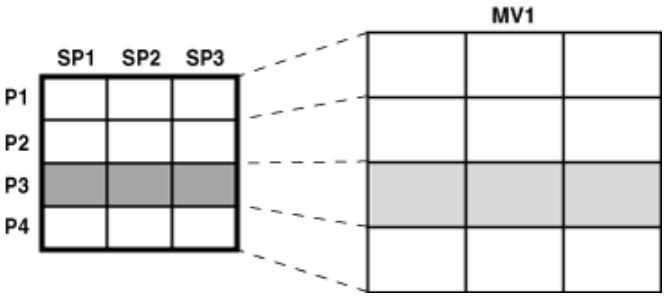
- `*_MVIEW_DETAIL_SUBPARTITION`

  To determine which subpartitions are fresh. (Physical partitions only.)

- `*_MVIEW_DETAIL_LOGICAL_PARTITIONS`
  To determine logical partition change tracking (LPCT) information for the materialized view.

### Determining the Freshness of Physical Partitions

You can check freshness of physical partitions using Partition Change Tracking (PCT). Here's an instance where there is a stale partition.




Query `USER_MVIEW_DETAIL_PARTITION` to access PCT freshness information for partitions, as shown in the following:

```
SELECT MVIEW_NAME,DETAILOBJ_NAME,DETAIL_PARTITION_NAME,
   DETAIL_PARTITION_POSITION,FRESHNESS
FROM USER_MVIEW_DETAIL_PARTITION
WHERE MVIEW_NAME = MV1;
```

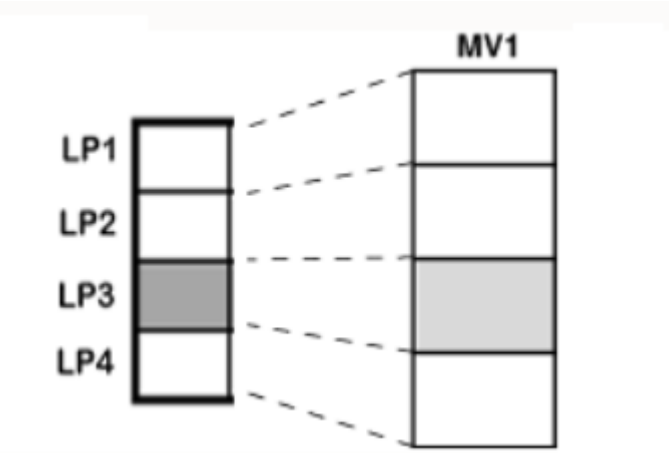
```
MVIEW_NAME  DETAILOBJ_NAME  DETAIL_PARTITION_NAME  DETAIL_PARTITION_POSITION  FRESHNESS
----------  --------------  ---------------------  -------------------------  ---------
       MV1              T1                     P1                          1      FRESH
       MV1              T1                     P2                          2      FRESH
       MV1              T1                     P3                          3      STALE
       MV1              T1                     P4                          4      FRESH
```

### Determining the Freshness of Logical Partitions

Use Logical Partition Change Tracking (LPCT) to determine the staleness/freshness of logical partitions.


Copyright © 2001, 2025, Oracle and/or its affiliates.

In this example, the logical partitions are LP1, LP2, LP3, and LP4. Assume that LP3 is stale.




> **Note**
>
> Logical partitioning does not support subpartitions.

You can query `DBA_MVIEW_DETAIL_LOGICAL_PARTITION` to determine freshness of logical partitions.

```
SQL> SELECT MVIEW_NAME, DETAILOBJ_NAME, LPARTNAME, LPART#, FRESHNESS \
FROM DBA_MVIEW_DETAIL_LOGICAL_PARTITION WHERE MVIEW_NAME = 'MV1' ORDER BY
1,2,3,4;

MVIEW_NAME            DETAILOBJ_NAME       LPARTNAME       LPART#     FRESHNESS
-------------------------------------------------------------------------------
       MV1                    SALES             LP1            1         FRESH
       MV1                    SALES             LP2            2         FRESH
       MV1                    SALES             LP3            3         STALE
       MV1                    SALES             LP4            4         FRESH
```

> **See Also**
>
> Examples of Using Views to Determine Freshness for more details and examples.

## 7.1.19.1 Examples of Using Views to Determine Freshness

Below are some examples that show how to view partition freshness information for materialized views and their detail tables.

**Example 7-3 Verifying the PCT Status of a Materialized View**

Query `USER_MVIEWS` to access PCT information about the materialized view, as shown in the following:

```
SELECT MVIEW_NAME, NUM_PCT_TABLES, NUM_FRESH_PCT_REGIONS,
   NUM_STALE_PCT_REGIONS
FROM USER_MVIEWS
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
WHERE MVIEW_NAME = MV1;

MVIEW_NAME NUM_PCT_TABLES NUM_FRESH_PCT_REGIONS NUM_STALE_PCT_REGIONS
---------- -------------- --------------------- --------------------
       MV1              1                     9                    3
```

**Example 7-4 Verifying the PCT Status in a Materialized View's Detail Table**

Query USER_MVIEW_DETAIL_RELATIONS to access PCT detail table information, as shown in the following:

```
SELECT MVIEW_NAME, DETAILOBJ_NAME, DETAILOBJ_PCT,
   NUM_FRESH_PCT_PARTITIONS, NUM_STALE_PCT_PARTITIONS
FROM USER_MVIEW_DETAIL_RELATIONS
WHERE MVIEW_NAME = MV1;


MVIEW_NAME  DETAILOBJ_NAME  DETAIL_OBJ_PCT  NUM_FRESH_PCT_PARTITIONS  NUM_STALE_PCT_PARTITIONS
----------  --------------  --------------  ------------------------  ------------------------
        MV1             T1               Y                         3                         1
```

**Example 7-5 Verifying Which Subpartitions are Fresh**

Query USER_MVIEW_DETAIL_SUBPARTITION to access PCT freshness information for subpartitions, as shown in the following:

```
SELECT MVIEW_NAME,DETAILOBJ_NAME,DETAIL_PARTITION_NAME, DETAIL_SUBPARTITION_NAME,
    DETAIL_SUBPARTITION_POSITION,FRESHNESS
FROM USER_MVIEW_DETAIL_SUBPARTITION
WHERE MVIEW_NAME = MV1;


MVIEW_NAME DETAILOBJ DETAIL_PARTITION DETAIL_SUBPARTITION_NAME DETAIL_SUBPARTITION_POS FRESHNESS
---------- --------- ---------------- ------------------------ ------------------------ ---------
       MV1        T1               P1                       SP1                        1     FRESH
       MV1        T1               P1                       SP2                        1     FRESH
       MV1        T1               P1                       SP3                        1     FRESH
       MV1        T1               P2                       SP1                        1     FRESH
       MV1        T1               P2                       SP2                        1     FRESH
       MV1        T1               P2                       SP3                        1     FRESH
       MV1        T1               P3                       SP1                        1     STALE
       MV1        T1               P3                       SP2                        1     STALE
       MV1        T1               P3                       SP3                        1     STALE
       MV1        T1               P4                       SP1                        1     FRESH
       MV1        T1               P4                       SP2                        1     FRESH
       MV1        T1               P4                       SP3                        1     FRESH
```

## 7.1.20 Scheduling Refresh of Materialized Views

Very often you have multiple materialized views in the database. Some of these can be computed by rewriting against others. This is very common in data warehousing environment where you may have nested materialized views or materialized views at different levels of some hierarchy.

In such cases, you should create the materialized views as BUILD DEFERRED, and then issue one of the refresh procedures in DBMS_MVIEW package to refresh all the materialized views. Oracle AI Database computes the dependencies and refreshes the materialized views in the right order. Consider the example of a complete hierarchical cube described in "Examples of Hierarchical Cube Materialized Views". Suppose all the materialized views have been created as BUILD DEFERRED. Creating the materialized views as BUILD DEFERRED only creates the


Copyright © 2001, 2025, Oracle and/or its affiliates.

metadata for all the materialized views. And, then, you can just call one of the refresh procedures in DBMS_MVIEW package to refresh all the materialized views in the right order:

```
DECLARE numerrs PLS_INTEGER;
BEGIN DBMS_MVIEW.REFRESH_DEPENDENT (
   number_of_failures => numerrs, list=>'SALES', method => 'C');
DBMS_OUTPUT.PUT_LINE('There were ' || numerrs || ' errors during refresh');
END;
/
```

The procedure refreshes the materialized views in the order of their dependencies (first sales_hierarchical_mon_cube_mv, followed by sales_hierarchical_qtr_cube_mv, then, sales_hierarchical_yr_cube_mv and finally, sales_hierarchical_all_cube_mv). Each of these materialized views gets rewritten against the one prior to it in the list).

The same kind of rewrite can also be used while doing PCT refresh. PCT refresh recomputes rows in a materialized view corresponding to changed rows in the detail tables. And, if there are other fresh materialized views available at the time of refresh, it can go directly against them as opposed to going against the detail tables.

Hence, it is always beneficial to pass a list of materialized views to any of the refresh procedures in DBMS_MVIEW package (irrespective of the method specified) and let the procedure figure out the order of doing refresh on materialized views.

# 7.2 Tips for Refreshing Materialized Views

This section contains the following topics with tips on refreshing materialized views:

- Tips for Refreshing Materialized Views with Aggregates
- Tips for Refreshing Materialized Views Without Aggregates
- Tips for Refreshing Nested Materialized Views
- Tips for Fast Refresh with UNION ALL
- Tips for Fast Refresh with Commit SCN-Based Materialized View Logs
- Tips After Refreshing Materialized Views

## 7.2.1 Tips for Refreshing Materialized Views with Aggregates

Following are some guidelines for using the refresh mechanism for materialized views with aggregates.

- For fast refresh, create materialized view logs on all detail tables involved in a materialized view with the ROWID, SEQUENCE and INCLUDING NEW VALUES clauses.

  Include all columns from the table likely to be used in materialized views in the materialized view logs.

  Fast refresh may be possible even if the SEQUENCE option is omitted from the materialized view log. If it can be determined that only inserts or deletes will occur on all the detail tables, then the materialized view log does not require the SEQUENCE clause. However, if updates to multiple tables are likely or required or if the specific update scenarios are unknown, make sure the SEQUENCE clause is included.

- Use Oracle's bulk loader utility or direct-path INSERT (INSERT with the APPEND hint for loads). Starting in Oracle Database 12*c*, the database automatically gathers table statistics as part of a bulk-load operation (CTAS and IAS) similar to how statistics are gathered


Copyright © 2001, 2025, Oracle and/or its affiliates.

when an index is created. By gathering statistics during the data load, you avoid additional scan operations and provide the necessary statistics as soon as the data becomes available to the users.

This is a lot more efficient than conventional insert. During loading, disable all constraints and re-enable when finished loading. Note that materialized view logs are required regardless of whether you use direct load or conventional DML.

Try to optimize the sequence of conventional mixed DML operations, direct-path `INSERT` and the fast refresh of materialized views. You can use fast refresh with a mixture of conventional DML and direct loads. Fast refresh can perform significant optimizations if it finds that only direct loads have occurred, as illustrated in the following:

1. Direct-path `INSERT` (SQL*Loader or `INSERT /*+ APPEND */`) into the detail table
2. Refresh materialized view
3. Conventional mixed DML
4. Refresh materialized view

You can use fast refresh with conventional mixed DML (`INSERT`, `UPDATE`, and `DELETE`) to the detail tables. However, fast refresh is able to perform significant optimizations in its processing if it detects that only inserts or deletes have been done to the tables, such as:

- DML `INSERT` or `DELETE` to the detail table
- Refresh materialized views
- DML update to the detail table
- Refresh materialized view

Even more optimal is the separation of `INSERT` and `DELETE`.

If possible, refresh should be performed after each type of data change (as shown earlier) rather than issuing only one refresh at the end. If that is not possible, restrict the conventional DML to the table to inserts only, to get much better refresh performance. Avoid mixing deletes and direct loads.

Furthermore, for refresh `ON COMMIT`, Oracle keeps track of the type of DML done in the committed transaction. Therefore, do not perform direct-path `INSERT` and DML to other tables in the same transaction, as Oracle may not be able to optimize the refresh phase.

For `ON COMMIT` materialized views, where refreshes automatically occur at the end of each transaction, it may not be possible to isolate the DML statements, in which case keeping the transactions short will help. However, if you plan to make numerous modifications to the detail table, it may be better to perform them in one transaction, so that refresh of the materialized view is performed just once at commit time rather than after each update.

- Oracle recommends partitioning the tables because it enables you to use:
  - Parallel DML

    For large loads or refresh, enabling parallel DML helps shorten the length of time for the operation.
  - Partition change tracking (PCT) fast refresh

    You can refresh your materialized views fast after partition maintenance operations on the detail tables. "About Partition Change Tracking" for details on enabling PCT for materialized views.
- Partitioning the materialized view also helps refresh performance as refresh can update the materialized view using parallel DML. For example, assume that the detail tables and


Copyright © 2001, 2025, Oracle and/or its affiliates.

materialized view are partitioned and have a parallel clause. The following sequence would enable Oracle to parallelize the refresh of the materialized view.

1. Bulk load into the detail table.
2. Enable parallel DML with an `ALTER SESSION ENABLE PARALLEL DML` statement.
3. Refresh the materialized view.

- For refresh using `DBMS_MVIEW.REFRESH`, set the parameter `atomic_refresh` to `FALSE`.
  - For `COMPLETE` refresh, this causes a `TRUNCATE` to delete existing rows in the materialized view, which is faster than a delete.
  - For `PCT` refresh, if the materialized view is partitioned appropriately, this uses `TRUNCATE PARTITION` to delete rows in the affected partitions of the materialized view, which is faster than a delete.
  - For `FAST` or `FORCE` refresh, if `COMPLETE` or PCT refresh is chosen, this is able to use the `TRUNCATE` optimizations described earlier.
- When using `DBMS_MVIEW.REFRESH` with `JOB_QUEUES`, remember to set `atomic` to `FALSE`. Otherwise, `JOB_QUEUES` is not used. Set the number of job queue processes greater than the number of processors.

  If job queues are enabled and there are many materialized views to refresh, it is faster to refresh all of them in a single command than to call them individually.
- Use `REFRESH FORCE` to ensure refreshing a materialized view so that it can definitely be used for query rewrite. The best refresh method is chosen. If a fast refresh cannot be done, a complete refresh is performed.
- Refresh all the materialized views in a single procedure call. This gives Oracle an opportunity to schedule refresh of all the materialized views in the right order taking into account dependencies imposed by nested materialized views and potential for efficient refresh by using query rewrite against other materialized views.

## 7.2.2 Tips for Refreshing Materialized Views Without Aggregates

If a materialized view contains joins but no aggregates, then having an index on each of the join column rowids in the detail table enhances refresh performance greatly, because this type of materialized view tends to be much larger than materialized views containing aggregates. For example, consider the following materialized view:

```
CREATE MATERIALIZED VIEW detail_fact_mv BUILD IMMEDIATE AS
SELECT s.rowid "sales_rid", t.rowid "times_rid", c.rowid "cust_rid",
   c.cust_state_province, t.week_ending_day, s.amount_sold
FROM sales s, times t, customers c
WHERE s.time_id = t.time_id AND s.cust_id = c.cust_id;
```

Indexes should be created on columns `sales_rid`, `times_rid` and `cust_rid`. Partitioning is highly recommended, as is enabling parallel DML in the session before invoking refresh, because it greatly enhances refresh performance.

This type of materialized view can also be fast refreshed if DML is performed on the detail table. It is recommended that the same procedure be applied to this type of materialized view as for a single table aggregate. That is, perform one type of change (direct-path `INSERT` or DML) and then refresh the materialized view. This is because Oracle AI Database can perform significant optimizations if it detects that only one type of change has been done.


Copyright © 2001, 2025, Oracle and/or its affiliates.

Also, Oracle recommends that the refresh be invoked after each table is loaded, rather than load all the tables and then perform the refresh.

For refresh `ON COMMIT`, Oracle keeps track of the type of DML done in the committed transaction. Oracle therefore recommends that you do not perform direct-path and conventional DML to other tables in the same transaction because Oracle may not be able to optimize the refresh phase. For example, the following is not recommended:

1. Direct load new data into the fact table
2. DML into the store table
3. Commit

Also, try not to mix different types of conventional DML statements if possible. This would again prevent using various optimizations during fast refresh. For example, try to avoid the following:

1. Insert into the fact table
2. Delete from the fact table
3. Commit

If many updates are needed, try to group them all into one transaction because refresh is performed just once at commit time, rather than after each update.

In a data warehousing environment, assuming that the materialized view has a parallel clause, the following sequence of steps is recommended:

1. Bulk load into the fact table
2. Enable parallel DML
3. An `ALTER SESSION ENABLE PARALLEL DML` statement
4. Refresh the materialized view

## 7.2.3 Tips for Refreshing Nested Materialized Views

All underlying objects are treated as ordinary tables when refreshing materialized views. If the `ON COMMIT` refresh option is specified, then all the materialized views are refreshed in the appropriate order at commit time. In other words, Oracle builds a partially ordered set of materialized views and refreshes them such that, after the successful completion of the refresh, all the materialized views are fresh. The status of the materialized views can be checked by querying the appropriate `USER_`, `DBA_`, or `ALL_MVIEWS` view.

If any of the materialized views are defined as `ON DEMAND` refresh (irrespective of whether the refresh method is `FAST`, `FORCE`, or `COMPLETE`), you must refresh them in the correct order (taking into account the dependencies between the materialized views) because the nested materialized view are refreshed with respect to the current contents of the other materialized views (whether fresh or not). This can be achieved by invoking the refresh procedure against the materialized view at the top of the nested hierarchy and specifying the `nested` parameter as `TRUE`.

If a refresh fails during commit time, the list of materialized views that has not been refreshed is written to the alert log, and you must manually refresh them along with all their dependent materialized views.

Use the same `DBMS_MVIEW` procedures on nested materialized views that you use on regular materialized views.

These procedures have the following behavior when used with nested materialized views:


Copyright © 2001, 2025, Oracle and/or its affiliates.

- If REFRESH is applied to a materialized view my_mv that is built on other materialized views, then my_mv is refreshed with respect to the current contents of the other materialized views (that is, the other materialized views are not made fresh first) unless you specify nested => TRUE.
- If REFRESH_DEPENDENT is applied to materialized view my_mv, then only materialized views that directly depend on my_mv are refreshed (that is, a materialized view that depends on a materialized view that depends on my_mv will not be refreshed) unless you specify nested => TRUE.
- If REFRESH_ALL_MVIEWS is used, the order in which the materialized views are refreshed is guaranteed to respect the dependencies between nested materialized views.
- GET_MV_DEPENDENCIES provides a list of the immediate (or direct) materialized view dependencies for an object.

## 7.2.4 Tips for Fast Refresh with UNION ALL

You can use fast refresh for materialized views that use the UNION ALL operator by providing a maintenance column in the definition of the materialized view. For example, a materialized view with a UNION ALL operator can be made fast refreshable as follows:

```
CREATE MATERIALIZED VIEW fast_rf_union_all_mv AS
SELECT x.rowid AS r1, y.rowid AS r2, a, b, c, 1 AS marker
FROM x, y WHERE x.a = y.b
UNION ALL
SELECT p.rowid, r.rowid, a, c, d, 2 AS marker
FROM p, r WHERE p.a = r.y;
```

The form of a maintenance marker column, column MARKER in the example, must be numeric_or_string_literal AS column_alias, where each UNION ALL member has a distinct value for numeric_or_string_literal.

## 7.2.5 Tips for Fast Refresh with Commit SCN-Based Materialized View Logs

You can often improve fast refresh performance by ensuring that your materialized view logs on the base table contain a WITH COMMIT SCN clause, often significantly. By optimizing materialized view log processing WITH COMMIT SCN, the fast refresh process can save time. The following example illustrates how to use this clause:

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID
 (prod_id, cust_id, time_id, channel_id, promo_id, quantity_sold, amount_sold),
COMMIT SCN INCLUDING NEW VALUES;
```

The materialized view refresh automatically uses the commit SCN-based materialized view log to save refresh time.

Note that only new materialized view logs can take advantage of COMMIT SCN. Existing materialized view logs cannot be altered to add COMMIT SCN unless they are dropped and recreated.

When a materialized view is created on both base tables with timestamp-based materialized view logs and base tables with commit SCN-based materialized view logs, an error (ORA-32414) is raised stating that materialized view logs are not compatible with each other for fast refresh.


Copyright © 2001, 2025, Oracle and/or its affiliates.

### 7.2.6 Tips After Refreshing Materialized Views

After you have performed a load or incremental load and rebuilt the detail table indexes, you must re-enable integrity constraints (if any) and refresh the materialized views and materialized view indexes that are derived from that detail data. In a data warehouse environment, referential integrity constraints are normally enabled with the `NOVALIDATE` or `RELY` options. An important decision to make before performing a refresh operation is whether the refresh needs to be recoverable. Because materialized view data is redundant and can always be reconstructed from the detail tables, it might be preferable to disable logging on the materialized view. To disable logging and run incremental refresh non-recoverably, use the `ALTER MATERIALIZED VIEW ... NOLOGGING` statement prior to refreshing.

If the materialized view is being refreshed using the `ON COMMIT` method, then, following refresh operations, consult the alert log `alert_SID.log` and the trace file `ora_SID_number.trc` to check that no errors have occurred.

## 7.3 Using Materialized Views with Partitioned Tables

A major maintenance component of a data warehouse is synchronizing (refreshing) the materialized views when the detail data changes. Partitioning the underlying detail tables can reduce the amount of time taken to perform the refresh task. This is possible because partitioning enables refresh to use parallel DML to update the materialized view. Also, it enables the use of partition change tracking.

"Materialized View Fast Refresh with Partition Change Tracking" provides additional information about PCT refresh.

### 7.3.1 Materialized View Fast Refresh with Partition Change Tracking

In a data warehouse, changes to the detail tables can often entail partition maintenance operations, such as `DROP`, `EXCHANGE`, `MERGE`, and `ADD PARTITION`. To maintain the materialized view after such operations used to require manual maintenance (see also `CONSIDER FRESH`) or complete refresh. You now have the option of using an addition to fast refresh known as partition change tracking (PCT) refresh.

For PCT to be available, the detail tables must be partitioned. The partitioning of the materialized view itself has no bearing on this feature. If PCT refresh is possible, it occurs automatically and no user intervention is required in order for it to occur. See "About Partition Change Tracking" for PCT requirements.

The following examples illustrate the use of this feature:

- PCT Fast Refresh for Materialized Views: Scenario 1
- PCT Fast Refresh for Materialized Views: Scenario 2
- PCT Fast Refresh for Materialized Views: Scenario 3

#### 7.3.1.1 PCT Fast Refresh for Materialized Views: Scenario 1

In this scenario, assume `sales` is a partitioned table using the `time_id` column and `products` is partitioned by the `prod_category` column. The table `times` is not a partitioned table.

1. Create the materialized view. The following materialized view satisfies requirements for PCT.


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
CREATE MATERIALIZED VIEW cust_mth_sales_mv
BUILD IMMEDIATE
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT s.time_id, s.prod_id, SUM(s.quantity_sold), SUM(s.amount_sold),
       p.prod_name, t.calendar_month_name, COUNT(*),
       COUNT(s.quantity_sold), COUNT(s.amount_sold)
FROM sales s, products p, times t
WHERE  s.time_id = t.time_id AND s.prod_id = p.prod_id
GROUP BY t.calendar_month_name, s.prod_id, p.prod_name, s.time_id;
```

2. Run the DBMS_MVIEW.EXPLAIN_MVIEW procedure to determine which tables allow PCT refresh.

```
MVNAME              CAPABILITY_NAME   POSSIBLE  RELATED_TEXT  MSGTXT
-----------------   ---------------   --------  ------------  ----------------
CUST_MTH_SALES_MV   PCT               Y         SALES
CUST_MTH_SALES_MV   PCT_TABLE         Y         SALES
CUST_MTH_SALES_MV   PCT_TABLE         N         PRODUCTS      no partition key
                                                              or PMARKER
                                                              in SELECT list
CUST_MTH_SALES_MV   PCT_TABLE         N         TIMES         relation is not
                                                              partitionedtable
```

   As can be seen from the partial sample output from EXPLAIN_MVIEW, any partition maintenance operation performed on the sales table allows PCT fast refresh. However, PCT is not possible after partition maintenance operations or updates to the products table as there is insufficient information contained in cust_mth_sales_mv for PCT refresh to be possible. Note that the times table is not partitioned and hence can never allow for PCT refresh. Oracle AI Database applies PCT refresh if it can determine that the materialized view has sufficient information to support PCT for all the updated tables. You can verify which partitions are fresh and stale with views such as DBA_MVIEWS and DBA_MVIEW_DETAIL_PARTITION.

   See "Analyzing Materialized View Capabilities" for information on how to use this procedure and also some details regarding PCT-related views.

3. Suppose at some later point, a SPLIT operation of one partition in the sales table becomes necessary.

   ```
   ALTER TABLE SALES
   SPLIT PARTITION month3 AT (TO_DATE('05-02-1998', 'DD-MM-YYYY'))
   INTO (PARTITION month3_1 TABLESPACE summ,
         PARTITION month3 TABLESPACE summ);
   ```

4. Insert some data into the sales table.

5. Fast refresh cust_mth_sales_mv using the DBMS_MVIEW.REFRESH procedure.

   ```
   EXECUTE DBMS_MVIEW.REFRESH('CUST_MTH_SALES_MV', 'F',
      '',TRUE,FALSE,0,0,0,FALSE);
   ```

Fast refresh automatically performs a PCT refresh as it is the only fast refresh possible in this scenario. However, fast refresh will not occur if a partition maintenance operation occurs when any update has taken place to a table on which PCT is not enabled. This is shown in "PCT Fast Refresh for Materialized Views: Scenario 2".

"PCT Fast Refresh for Materialized Views: Scenario 1" would also be appropriate if the materialized view was created using the PMARKER clause as illustrated in the following:

```
CREATE MATERIALIZED VIEW cust_sales_marker_mv
BUILD IMMEDIATE
REFRESH FAST ON DEMAND
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
ENABLE QUERY REWRITE AS
SELECT DBMS_MVIEW.PMARKER(s.rowid) s_marker, SUM(s.quantity_sold),
  SUM(s.amount_sold), p.prod_name, t.calendar_month_name, COUNT(*),
  COUNT(s.quantity_sold), COUNT(s.amount_sold)
FROM sales s, products p, times t
WHERE  s.time_id = t.time_id AND s.prod_id = p.prod_id
GROUP BY DBMS_MVIEW.PMARKER(s.rowid),
         p.prod_name, t.calendar_month_name;
```

### 7.3.1.2 PCT Fast Refresh for Materialized Views: Scenario 2

In this scenario, the first three steps are the same as in "PCT Fast Refresh for Materialized Views: Scenario 1". Then, the `SPLIT` partition operation to the `sales` table is performed, but before the materialized view refresh occurs, records are inserted into the `times` table.

1. The same as in "PCT Fast Refresh for Materialized Views: Scenario 1".
2. The same as in "PCT Fast Refresh for Materialized Views: Scenario 1".
3. The same as in "PCT Fast Refresh for Materialized Views: Scenario 1".
4. After issuing the same `SPLIT` operation, as shown in "PCT Fast Refresh for Materialized Views: Scenario 1", some data is inserted into the `times` table.

   ```
   ALTER TABLE SALES
   SPLIT PARTITION month3 AT (TO_DATE('05-02-1998', 'DD-MM-YYYY'))
   INTO (PARTIITION month3_1 TABLESPACE summ,
         PARTITION month3 TABLESPACE summ);
   ```

5. Refresh `cust_mth_sales_mv`.

   ```
   EXECUTE DBMS_MVIEW.REFRESH('CUST_MTH_SALES_MV', 'F',
       '', TRUE, FALSE, 0, 0, 0, FALSE, FALSE);
   ORA-12052: cannot fast refresh materialized view SH.CUST_MTH_SALES_MV
   ```

The materialized view is not fast refreshable because DML has occurred to a table on which PCT fast refresh is not possible. To avoid this occurring, Oracle recommends performing a fast refresh immediately after any partition maintenance operation on detail tables for which partition tracking fast refresh is available.

If the situation in "PCT Fast Refresh for Materialized Views: Scenario 2" occurs, there are two possibilities; perform a complete refresh or switch to the `CONSIDER FRESH` option outlined in the following, if suitable. However, it should be noted that `CONSIDER FRESH` and partition change tracking fast refresh are not compatible. Once the `ALTER MATERIALIZED VIEW cust_mth_sales_mv CONSIDER FRESH` statement has been issued, PCT refresh is no longer be applied to this materialized view, until a complete refresh is done. Moreover, you should not use `CONSIDER FRESH` unless you have taken manual action to ensure that the materialized view is indeed fresh.

A common situation in a data warehouse is the use of rolling windows of data. In this case, the detail table and the materialized view may contain say the last 12 months of data. Every month, new data for a month is added to the table and the oldest month is deleted (or maybe archived). PCT refresh provides a very efficient mechanism to maintain the materialized view in this case.

### 7.3.1.3 PCT Fast Refresh for Materialized Views: Scenario 3

1. The new data is usually added to the detail table by adding a new partition and exchanging it with a table containing the new data.


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
ALTER TABLE sales ADD PARTITION month_new ...
ALTER TABLE sales EXCHANGE PARTITION month_new month_new_table
```

2. Next, the oldest partition is dropped or truncated.

```
ALTER TABLE sales DROP PARTITION month_oldest;
```

3. Now, if the materialized view satisfies all conditions for PCT refresh.

```
EXECUTE DBMS_MVIEW.REFRESH('CUST_MTH_SALES_MV', 'F', '', TRUE, FALSE, 0, 0, 0,
FALSE, FALSE);
```

Fast refresh will automatically detect that PCT is available and perform a PCT refresh.

# 7.4 Refreshing Materialized Views Based on Hybrid Partitioned Tables

You can use the complete, fast, or PCT refresh methods to refresh a materialized view that is based on a hybrid partitioned table.

Because Oracle AI Database has no control over how data is maintained in the external source, data in the external partitions is not guaranteed to be fresh and its freshness is marked as UNKNOWN. Data from external partitions can be used only in trusted integrity mode or stale-tolerated mode.

Refreshing data that originates from external partitions can be an expensive and often unnecessary (when source data is unchanged) operation. You can skip refreshing materialized view data that corresponds to external partitions by using the `skip_ext_data` attribute in the `DBMS_MVIEW.REFRESH` procedure. When you set this attribute to TRUE, the materialized view data corresponding to external partitions is not recomputed and remains in trusted mode with the state UNKNOWN. By default, `skip_ext_data` is FALSE.

> **Note**
>
> If the hybrid partitioned table on which a materialized view is based is not PCT-enabled, then `COMPLETE` and `FORCE` are the only refresh methods supported. `FAST` refresh is not supported.

**Example 7-6 Refreshing a Materialized View that is Based on a Hybrid Partitioned Table**

Assume that the internal partition, `year_2020`, in the materialized view named `hypt_mv` is stale. This materialized view is based on a hybrid partitioned table. Querying the catalog view `USER_MVIEW_DETAIL_PARTITION` displays the following:

```
SELECT mview_name, detail_partition_name, freshness, last_refresh_time
    FROM USER_MVIEW_DETAIL_PARTITION;

MVIEW_NAME    DETAIL_PARTITION_NAME    FRESHNESS    LAST_REFRESH_TIME
----------       --------------------    ---------    -----------------
HyPT_MV          century_19            UNKNOWN          2018-10-31 20:48:00.20
HyPT_MV          century_20                  UNKNOWN          2018-10-31
20:48:00.20
HyPT_MV          year_2020             STALE         2018-10-31 20:48:00.20
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
HyPT_MV          year_2021                  FRESH            2021-10-31
20:48:00.20
```

Use the following command to perform a fast refresh of the materialized view:

```
DBMS_MVIEW.REFERSH('HyPT_MV', 'F', skip_ext_data => false);
```

Querying the catalog view `USER_MVIEW_DETAIL_PARTITION` after the refresh, displays the following:

```
SELECT mview_name, detail_partition_name, freshness, last_refresh_time
    FROM USER_MVIEW_DETAIL_PARTITION;

MVIEW_NAME     DETAIL_PARTITION_NAME    FRESHNESS      LAST_REFRESH_TIME
----------     ---------------------    ---------      -----------------
HyPT_MV        century_19               UNKNOWN        2018-10-31 20:48:00.20
HyPT_MV        century_20               UNKNOWN        2018-10-31 20:48:00.20
HyPT_MV        year_2020                FRESH          2021-10-31 21:32:17.00
HyPT_MV        year_2021                FRESH          2021-10-31 20:48:00.20
```

Note that only the internal partition, `year_2020`, was refreshed. The partition, `year_2021`, was not refreshed as it was already fresh. When `skip_ext_data` is set to FALSE, a full refresh of the external partitions and a fast refresh of the internal partitions is performed.

# 7.5 Using Partitioning to Improve Data Warehouse Refresh

ETL (Extraction, Transformation and Loading) is done on a scheduled basis to reflect changes made to the original source system. During this step, you physically insert the new, clean data into the production data warehouse schema, and take all of the other steps necessary (such as building indexes, validating constraints, taking backups) to make this new data available to the end users. Once all of this data has been loaded into the data warehouse, the materialized views have to be updated to reflect the latest data.

The partitioning scheme of the data warehouse is often crucial in determining the efficiency of refresh operations in the data warehouse load process. In fact, the load process is often the primary consideration in choosing the partitioning scheme of data warehouse tables and indexes.

The partitioning scheme of the largest data warehouse tables (for example, the fact table in a star schema) should be based upon the loading paradigm of the data warehouse.

Most data warehouses are loaded with new data on a regular schedule. For example, every night, week, or month, new data is brought into the data warehouse. The data being loaded at the end of the week or month typically corresponds to the transactions for the week or month. In this very common scenario, the data warehouse is being loaded by time. This suggests that the data warehouse tables should be partitioned on a date column. In our data warehouse example, suppose the new data is loaded into the `sales` table every month. Furthermore, the `sales` table has been partitioned by month. These steps show how the load process proceeds to add the data for a new month (January 2001) to the table `sales`.

1. Place the new data into a separate table, `sales_01_2001`. This data can be directly loaded into `sales_01_2001` from outside the data warehouse, or this data can be the result of previous data transformation operations that have already occurred in the data warehouse.


Copyright © 2001, 2025, Oracle and/or its affiliates.

sales_01_2001 has the exact same columns, data types, and so forth, as the sales table. Gather statistics on the sales_01_2001 table.

2. Create indexes and add constraints on sales_01_2001. Again, the indexes and constraints on sales_01_2001 should be identical to the indexes and constraints on sales. Indexes can be built in parallel and should use the NOLOGGING and the COMPUTE STATISTICS options. For example:

```
CREATE BITMAP INDEX sales_01_2001_customer_id_bix
  ON sales_01_2001(customer_id)
      TABLESPACE sales_idx NOLOGGING PARALLEL 8 COMPUTE STATISTICS;
```

   Apply all constraints to the sales_01_2001 table that are present on the sales table. This includes referential integrity constraints. A typical constraint would be:

```
ALTER TABLE sales_01_2001 ADD CONSTRAINT sales_customer_id
      REFERENCES customer(customer_id) ENABLE NOVALIDATE;
```

   If the partitioned table sales has a primary or unique key that is enforced with a global index structure, ensure that the constraint on sales_pk_jan01 is validated without the creation of an index structure, as in the following:

```
ALTER TABLE sales_01_2001 ADD CONSTRAINT sales_pk_jan01
PRIMARY KEY (sales_transaction_id) DISABLE VALIDATE;
```

   The creation of the constraint with ENABLE clause would cause the creation of a unique index, which does not match a local index structure of the partitioned table. You must not have any index structure built on the nonpartitioned table to be exchanged for existing global indexes of the partitioned table. The exchange command would fail.

3. Add the sales_01_2001 table to the sales table.

   In order to add this new data to the sales table, you must do two things. First, you must add a new partition to the sales table. You use an ALTER TABLE ... ADD PARTITION statement. This adds an empty partition to the sales table:

```
ALTER TABLE sales ADD PARTITION sales_01_2001
VALUES LESS THAN (TO_DATE('01-FEB-2001', 'DD-MON-YYYY'));
```

   Then, you can add our newly created table to this partition using the EXCHANGE PARTITION operation. This exchanges the new, empty partition with the newly loaded table.

```
ALTER TABLE sales EXCHANGE PARTITION sales_01_2001 WITH TABLE sales_01_2001
INCLUDING INDEXES WITHOUT VALIDATION UPDATE GLOBAL INDEXES;
```

   The EXCHANGE operation preserves the indexes and constraints that were already present on the sales_01_2001 table. For unique constraints (such as the unique constraint on sales_transaction_id), you can use the UPDATE GLOBAL INDEXES clause, as shown previously. This automatically maintains your global index structures as part of the partition maintenance operation and keep them accessible throughout the whole process. If there were only foreign-key constraints, the exchange operation would be instantaneous.

   Note that, if you use synchronous refresh, instead of performing Step 3, you must register the sales_01_2001 table using the DBMS_SYNC_REFRESH.REGISTER_PARTITION_OPERATION package. See Synchronous Refresh for more information.

The benefits of this partitioning technique are significant. First, the new data is loaded with minimal resource utilization. The new data is loaded into an entirely separate table, and the index processing and constraint processing are applied only to the new partition. If the sales table was 50 GB and had 12 partitions, then a new month's worth of data contains approximately four GB. Only the new month's worth of data must be indexed. None of the


Copyright © 2001, 2025, Oracle and/or its affiliates.

indexes on the remaining 46 GB of data must be modified at all. This partitioning scheme additionally ensures that the load processing time is directly proportional to the amount of new data being loaded, not to the total size of the `sales` table.

Second, the new data is loaded with minimal impact on concurrent queries. All of the operations associated with data loading are occurring on a separate `sales_01_2001` table. Therefore, none of the existing data or indexes of the `sales` table is affected during this data refresh process. The `sales` table and its indexes remain entirely untouched throughout this refresh process.

Third, in case of the existence of any global indexes, those are incrementally maintained as part of the exchange command. This maintenance does not affect the availability of the existing global index structures.

The exchange operation can be viewed as a publishing mechanism. Until the data warehouse administrator exchanges the `sales_01_2001` table into the `sales` table, end users cannot see the new data. Once the exchange has occurred, then any end user query accessing the `sales` table is immediately able to see the `sales_01_2001` data.

Partitioning is useful not only for adding new data but also for removing and archiving data. Many data warehouses maintain a rolling window of data. For example, the data warehouse stores the most recent 36 months of `sales` data. Just as a new partition can be added to the `sales` table (as described earlier), an old partition can be quickly (and independently) removed from the `sales` table. These two benefits (reduced resources utilization and minimal end-user impact) are just as pertinent to removing a partition as they are to adding a partition.

Removing data from a partitioned table does not necessarily mean that the old data is physically deleted from the database. There are two alternatives for removing old data from a partitioned table. First, you can physically delete all data from the database by dropping the partition containing the old data, thus freeing the allocated space:

```
ALTER TABLE sales DROP PARTITION sales_01_1998;
```

Also, you can exchange the old partition with an empty table of the same structure; this empty table is created equivalent to steps 1 and 2 described in the load process. Assuming the new empty table stub is named `sales_archive_01_1998`, the following SQL statement empties partition `sales_01_1998`:

```
ALTER TABLE sales EXCHANGE PARTITION sales_01_1998
WITH TABLE sales_archive_01_1998 INCLUDING INDEXES WITHOUT VALIDATION
UPDATE GLOBAL INDEXES;
```

Note that the old data is still existent as the exchanged, nonpartitioned table `sales_archive_01_1998`.

If the partitioned table was setup in a way that every partition is stored in a separate tablespace, you can archive (or transport) this table using Oracle AI Database's transportable tablespace framework before dropping the actual data (the tablespace).

In some situations, you might not want to drop the old data immediately, but keep it as part of the partitioned table; although the data is no longer of main interest, there are still potential queries accessing this old, read-only data. You can use Oracle's data compression to minimize the space usage of the old data. You also assume that at least one compressed partition is already part of the partitioned table.


Copyright © 2001, 2025, Oracle and/or its affiliates.

> **See Also**
>
> - "Transportation Using Transportable Tablespaces" for further details regarding transportable tablespaces
> - *Oracle AI Database Administrator's Guide* for more information regarding table compression
> - *Oracle AI Database VLDB and Partitioning Guide* for more information regarding partitioning and table compression

## 7.5.1 Data Warehouse Refresh Scenarios

A typical scenario might not only need to compress old data, but also to merge several old partitions to reflect the granularity for a later backup of several merged partitions. Let us assume that a backup (partition) granularity is on a quarterly base for any quarter, where the oldest month is more than 36 months behind the most recent month. In this case, you are therefore compressing and merging `sales_01_1998`, `sales_02_1998`, and `sales_03_1998` into a new, compressed partition `sales_q1_1998`.

1. Create the new merged partition in parallel in another tablespace. The partition is compressed as part of the `MERGE` operation:

```
ALTER TABLE sales MERGE PARTITIONS sales_01_1998, sales_02_1998, sales_03_1998
 INTO PARTITION sales_q1_1998 TABLESPACE archive_q1_1998
COMPRESS UPDATE GLOBAL INDEXES PARALLEL 4;
```

2. The partition `MERGE` operation invalidates the local indexes for the new merged partition. You therefore have to rebuild them:

```
ALTER TABLE sales MODIFY PARTITION sales_q1_1998
REBUILD UNUSABLE LOCAL INDEXES;
```

Alternatively, you can choose to create the new compressed table outside the partitioned table and exchange it back. The performance and the temporary space consumption is identical for both methods:

1. Create an intermediate table to hold the new merged information. The following statement inherits all `NOT NULL` constraints from the original table by default:

```
CREATE TABLE sales_q1_1998_out TABLESPACE archive_q1_1998
NOLOGGING COMPRESS PARALLEL 4 AS SELECT * FROM sales
WHERE time_id >=  TO_DATE('01-JAN-1998','dd-mon-yyyy')
  AND time_id < TO_DATE('01-APR-1998','dd-mon-yyyy');
```

2. Create the equivalent index structure for table `sales_q1_1998_out` than for the existing table `sales`.
3. Prepare the existing table sales for the exchange with the new compressed table `sales_q1_1998_out`. Because the table to be exchanged contains data actually covered in three partitions, you have to create one matching partition, having the range boundaries you are looking for. You simply have to drop two of the existing partitions. Note that you have to drop the lower two partitions `sales_01_1998` and `sales_02_1998`; the lower boundary of a range partition is always defined by the upper (exclusive) boundary of the previous partition:

```
ALTER TABLE sales DROP PARTITION sales_01_1998;
ALTER TABLE sales DROP PARTITION sales_02_1998;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

4. You can now exchange table `sales_q1_1998_out` with partition `sales_03_1998`. Unlike what the name of the partition suggests, its boundaries cover Q1-1998.

```
ALTER TABLE sales EXCHANGE PARTITION sales_03_1998
WITH TABLE sales_q1_1998_out INCLUDING INDEXES WITHOUT VALIDATION
UPDATE GLOBAL INDEXES;
```

Both methods apply to slightly different business scenarios: Using the `MERGE PARTITION` approach invalidates the local index structures for the affected partition, but it keeps all data accessible all the time. Any attempt to access the affected partition through one of the unusable index structures raises an error. The limited availability time is approximately the time for re-creating the local bitmap index structures. In most cases, this can be neglected, because this part of the partitioned table should not be accessed too often.

The CTAS approach, however, minimizes unavailability of any index structures close to zero, but there is a specific time window, where the partitioned table does not have all the data, because you dropped two partitions. The limited availability time is approximately the time for exchanging the table. Depending on the existence and number of global indexes, this time window varies. Without any existing global indexes, this time window is a matter of a fraction to few seconds.

These examples are a simplification of the data warehouse rolling window load scenario. Real-world data warehouse refresh characteristics are always more complex. However, the advantages of this rolling window approach are not diminished in more complex scenarios.

Note that before you add single or multiple compressed partitions to a partitioned table for the first time, all local bitmap indexes must be either dropped or marked unusable. After the first compressed partition is added, no additional actions are necessary for all subsequent operations involving compressed partitions. It is irrelevant how the compressed partitions are added to the partitioned table.

**See Also**

- *Oracle Database VLDB and Partitioning Guide* for more information regarding partitioning and table compression
- *Oracle Database Administrator's Guide* for further details about partitioning and table compression.

## 7.5.2 Scenarios for Using Partitioning for Refreshing Data Warehouses

This section describes the following two typical scenarios where partitioning is used with refresh:

- Partitioning for Refreshing Data Warehouses: Scenario 1
- Partitioning for Refreshing Data Warehouses: Scenario 2

### 7.5.2.1 Partitioning for Refreshing Data Warehouses: Scenario 1

Data is loaded daily. However, the data warehouse contains two years of data, so that partitioning by day might not be desired.

The solution is to partition by week or month (as appropriate). Use `INSERT` to add the new data to an existing partition. The `INSERT` operation only affects a single partition, so the benefits described previously remain intact. The `INSERT` operation could occur while the partition remains a part of the table. Inserts into a single partition can be parallelized:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
INSERT /*+ APPEND*/ INTO sales PARTITION (sales_01_2001)
SELECT * FROM new_sales;
```

The indexes of this `sales` partition is maintained in parallel as well. An alternative is to use the `EXCHANGE` operation. You can do this by exchanging the `sales_01_2001` partition of the `sales` table and then using an `INSERT` operation. You might prefer this technique when dropping and rebuilding indexes is more efficient than maintaining them.

### 7.5.2.2 Partitioning for Refreshing Data Warehouses: Scenario 2

New data feeds, although consisting primarily of data for the most recent day, week, and month, also contain some data from previous time periods.

**Solution 1**

Use parallel SQL operations (such as `CREATE TABLE ... AS SELECT`) to separate the new data from the data in previous time periods. Process the old data separately using other techniques.

New data feeds are not solely time based. You can also feed new data into a data warehouse with data from multiple operational systems on a business need basis. For example, the sales data from direct channels may come into the data warehouse separately from the data from indirect channels. For business reasons, it may furthermore make sense to keep the direct and indirect data in separate partitions.

**Solution 2**

Oracle supports composite range-list partitioning. The primary partitioning strategy of the sales table could be range partitioning based on `time_id` as shown in the example. However, the subpartitioning is a list based on the channel attribute. Each subpartition can now be loaded independently of each other (for each distinct channel) and added in a rolling window operation as discussed before. The partitioning strategy addresses the business needs in the most optimal manner.

## 7.6 Optimizing DML Operations During Refresh

You can optimize DML performance through the following techniques:

- Implementing an Efficient MERGE Operation
- Maintaining Referential Integrity in Data Warehouses
- Purging Data from Data Warehouses

### 7.6.1 Implementing an Efficient MERGE Operation

Commonly, the data that is extracted from a source system is not simply a list of new records that needs to be inserted into the data warehouse. Instead, this new data set is a combination of new records as well as modified records. For example, suppose that most of data extracted from the OLTP systems will be new sales transactions. These records are inserted into the warehouse's `sales` table, but some records may reflect modifications of previous transactions, such as returned merchandise or transactions that were incomplete or incorrect when initially loaded into the data warehouse. These records require updates to the `sales` table.

As a typical scenario, suppose that there is a table called `new_sales` that contains both inserts and updates that are applied to the `sales` table. When designing the entire data warehouse load process, it was determined that the `new_sales` table would contain records with the following semantics:


Copyright © 2001, 2025, Oracle and/or its affiliates.

- If a given sales_transaction_id of a record in new_sales already exists in sales, then update the sales table by adding the sales_dollar_amount and sales_quantity_sold values from the new_sales table to the existing row in the sales table.
- Otherwise, insert the entire new record from the new_sales table into the sales table.

This UPDATE-ELSE-INSERT operation is often called a merge. A merge can be executed using one SQL statement.

**Example 7-7 MERGE Operation**

```
MERGE INTO sales s USING new_sales n
ON (s.sales_transaction_id = n.sales_transaction_id)
WHEN MATCHED THEN
UPDATE SET s.sales_quantity_sold = s.sales_quantity_sold + n.sales_quantity_sold,
 s.sales_dollar_amount = s.sales_dollar_amount + n.sales_dollar_amount
WHEN NOT MATCHED THEN INSERT (sales_transaction_id, sales_quantity_sold,
sales_dollar_amount)
VALUES (n.sales_transcation_id, n.sales_quantity_sold, n.sales_dollar_amount);
```

In addition to using the MERGE statement for unconditional UPDATE ELSE INSERT functionality into a target table, you can also use it to:

- Perform an UPDATE only or INSERT only statement.
- Apply additional WHERE conditions for the UPDATE or INSERT portion of the MERGE statement.
- The UPDATE operation can even delete rows if a specific condition yields true.

**Example 7-8 Omitting the INSERT Clause**

In some data warehouse applications, it is not allowed to add new rows to historical information, but only to update them. It may also happen that you do not want to update but only insert new information. The following example demonstrates INSERT-only with UPDATE-only functionality:

```
MERGE USING Product_Changes S     -- Source/Delta table
INTO Products D1                  -- Destination table 1
ON (D1.PROD_ID = S.PROD_ID)       -- Search/Join condition
WHEN MATCHED THEN UPDATE          -- update if join
SET D1.PROD_STATUS = S.PROD_NEW_STATUS
```

**Example 7-9 Omitting the UPDATE Clause**

The following statement illustrates an example of omitting an UPDATE:

```
MERGE USING New_Product S           -- Source/Delta table
INTO Products D2                    -- Destination table 2
ON (D2.PROD_ID = S.PROD_ID)         -- Search/Join condition
WHEN NOT MATCHED THEN               -- insert if no join
INSERT (PROD_ID, PROD_STATUS) VALUES (S.PROD_ID, S.PROD_NEW_STATUS)
```

When the INSERT clause is omitted, Oracle AI Database performs a regular join of the source and the target tables. When the UPDATE clause is omitted, Oracle AI Database performs an antijoin of the source and the target tables. This makes the join between the source and target table more efficient.

**Example 7-10 Skipping the UPDATE Clause**

In some situations, you may want to skip the UPDATE operation when merging a given row into the table. In this case, you can use an optional WHERE clause in the UPDATE clause of the MERGE. As a result, the UPDATE operation only executes when a given condition is true. The following statement illustrates an example of skipping the UPDATE operation:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
MERGE
USING Product_Changes S                        -- Source/Delta table
INTO Products P                                -- Destination table 1
ON (P.PROD_ID = S.PROD_ID)                     -- Search/Join condition
WHEN MATCHED THEN
UPDATE                                         -- update if join
SET P.PROD_LIST_PRICE = S.PROD_NEW_PRICE
WHERE P.PROD_STATUS <> "OBSOLETE"              -- Conditional UPDATE
```

This shows how the UPDATE operation would be skipped if the condition P.PROD_STATUS <> "OBSOLETE" is not true. The condition predicate can refer to both the target and the source table.

**Example 7-11 Conditional Inserts with MERGE Statements**

You may want to skip the INSERT operation when merging a given row into the table. So an optional WHERE clause is added to the INSERT clause of the MERGE. As a result, the INSERT operation only executes when a given condition is true. The following statement offers an example:

```
MERGE USING Product_Changes S                      -- Source/Delta table
INTO Products P                                    -- Destination table 1
ON (P.PROD_ID = S.PROD_ID)                         -- Search/Join condition
WHEN MATCHED THEN UPDATE                           -- update if join
SET P.PROD_LIST_PRICE = S.PROD_NEW_PRICE
WHERE P.PROD_STATUS <> "OBSOLETE"                  -- Conditional
WHEN NOT MATCHED THEN
INSERT (PROD_ID, PROD_STATUS, PROD_LIST_PRICE)     -- insert if not join
VALUES (S.PROD_ID, S.PROD_NEW_STATUS, S.PROD_NEW_PRICE)
WHERE S.PROD_STATUS <> "OBSOLETE";                 -- Conditional INSERT
```

This example shows that the INSERT operation would be skipped if the condition S.PROD_STATUS <> "OBSOLETE" is not true, and INSERT only occurs if the condition is true. The condition predicate can refer to the source table only. The condition predicate can only refer to the source table.

**Example 7-12 Using the DELETE Clause with MERGE Statements**

You may want to cleanse tables while populating or updating them. To do this, you may want to consider using the DELETE clause in a MERGE statement, as in the following example:

```
MERGE USING Product_Changes S
INTO Products D ON (D.PROD_ID = S.PROD_ID)
WHEN MATCHED THEN
UPDATE SET D.PROD_LIST_PRICE =S.PROD_NEW_PRICE, D.PROD_STATUS = S.PROD_NEWSTATUS
DELETE WHERE (D.PROD_STATUS = "OBSOLETE")
WHEN NOT MATCHED THEN
INSERT (PROD_ID, PROD_LIST_PRICE, PROD_STATUS)
VALUES (S.PROD_ID, S.PROD_NEW_PRICE, S.PROD_NEW_STATUS);
```

Thus when a row is updated in products, Oracle checks the delete condition D.PROD_STATUS = "OBSOLETE", and deletes the row if the condition yields true.

The DELETE operation is not as same as that of a complete DELETE statement. Only the rows from the destination of the MERGE can be deleted. The only rows that are affected by the DELETE are the ones that are updated by this MERGE statement. Thus, although a given row of the destination table meets the delete condition, if it does not join under the ON clause condition, it is not deleted.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Example 7-13 Unconditional Inserts with MERGE Statements**

You may want to insert all of the source rows into a table. In this case, the join between the source and target table can be avoided. By identifying special constant join conditions that always result to FALSE, for example, 1=0, such MERGE statements are optimized and the join condition are suppressed.

```
MERGE USING New_Product S       -- Source/Delta table
INTO Products P                 -- Destination table 1
ON (1 = 0)                      -- Search/Join condition
WHEN NOT MATCHED THEN           -- insert if no join
INSERT (PROD_ID, PROD_STATUS) VALUES (S.PROD_ID, S.PROD_NEW_STATUS)
```

## 7.6.2 Maintaining Referential Integrity in Data Warehouses

In some data warehousing environments, you might want to insert new data into tables in order to guarantee referential integrity. For example, a data warehouse may derive sales from an operational system that retrieves data directly from cash registers. sales is refreshed nightly. However, the data for the product dimension table may be derived from a separate operational system. The product dimension table may only be refreshed once for each week, because the product table changes relatively slowly. If a new product was introduced on Monday, then it is possible for that product's product_id to appear in the sales data of the data warehouse before that product_id has been inserted into the data warehouses product table.

Although the sales transactions of the new product may be valid, this sales data do not satisfy the referential integrity constraint between the product dimension table and the sales fact table. Rather than disallow the new sales transactions, you might choose to insert the sales transactions into the sales table. However, you might also wish to maintain the referential integrity relationship between the sales and product tables. This can be accomplished by inserting new rows into the product table as placeholders for the unknown products.

As in previous examples, assume that the new data for the sales table is staged in a separate table, new_sales. Using a single INSERT statement (which can be parallelized), the product table can be altered to reflect the new products:

```
INSERT INTO product
  (SELECT sales_product_id, 'Unknown Product Name', NULL, NULL ...
   FROM new_sales WHERE sales_product_id NOT IN
  (SELECT product_id FROM product));
```

## 7.6.3 Purging Data from Data Warehouses

Occasionally, it is necessary to remove large amounts of data from a data warehouse. A very common scenario is the rolling window discussed previously, in which older data is rolled out of the data warehouse to make room for new data.

However, sometimes other data might need to be removed from a data warehouse. Suppose that a retail company has previously sold products from XYZ Software, and that XYZ Software has subsequently gone out of business. The business users of the warehouse may decide that they are no longer interested in seeing any data related to XYZ Software, so this data should be deleted.

One approach to removing a large volume of data is to use parallel delete as shown in the following statement:

```
DELETE FROM sales WHERE sales_product_id IN (SELECT product_id
   FROM product WHERE product_category = 'XYZ Software');
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

This SQL statement spawns one parallel process for each partition. This approach is much more efficient than a series of DELETE statements, and none of the data in the sales table needs to be moved. However, this approach also has some disadvantages. When removing a large percentage of rows, the DELETE statement leaves many empty row-slots in the existing partitions. If new data is being loaded using a rolling window technique (or is being loaded using direct-path INSERT or load), then this storage space is not reclaimed. Moreover, even though the DELETE statement is parallelized, there might be more efficient methods. An alternative method is to re-create the entire sales table, keeping the data for all product categories except XYZ Software.

```
CREATE TABLE sales2 AS SELECT * FROM sales, product
WHERE sales.sales_product_id = product.product_id
AND product_category <> 'XYZ Software'
NOLOGGING PARALLEL (DEGREE 8)
#PARTITION ... ; #create indexes, constraints, and so on
DROP TABLE SALES;
RENAME SALES2 TO SALES;
```

This approach may be more efficient than a parallel delete. However, it is also costly in terms of the amount of disk space, because the sales table must effectively be instantiated twice.

An alternative method to utilize less space is to re-create the sales table one partition at a time:

```
CREATE TABLE sales_temp AS SELECT * FROM sales WHERE 1=0;
INSERT INTO sales_temp
SELECT * FROM sales PARTITION (sales_99jan), product
WHERE sales.sales_product_id = product.product_id
AND product_category <> 'XYZ Software';
<create appropriate indexes and constraints on sales_temp>
ALTER TABLE sales EXCHANGE PARTITION sales_99jan WITH TABLE sales_temp;
```

Continue this process for each partition in the sales table.


Copyright © 2001, 2025, Oracle and/or its affiliates.

# 8
# Synchronous Refresh

This chapter describes a method to synchronize changes to the tables and materialized views in a data warehouse. This method is based on synchronizing updates to tables and materialized views, and is called synchronous refresh.

This chapter includes the following sections:


- About Synchronous Refresh for Materialized Views
- Using Synchronous Refresh for Materialized Views
- Using Synchronous Refresh Groups
- Specifying and Preparing Change Data for Synchronous Refresh
- Troubleshooting Synchronous Refresh Operations
- Performing Synchronous Refresh Eligibility Analysis
- Overview of Synchronous Refresh Security Considerations


## 8.1 About Synchronous Refresh for Materialized Views

Synchronous refresh is a refresh method introduced in Oracle Database 12*c* Release 1 that enables you to keep a set of tables and the materialized views defined on them to be always in sync. It is well-suited for data warehouses, where the loading of incremental data is tightly controlled and occurs at periodic intervals.

In most data warehouses, the fact tables are partitioned along the time dimension and, very often, the incremental data load consists mainly of changes to recent time periods. Synchronous refresh exploits these characteristics to greatly improve refresh performance and throughput. This results in fast query performance for both planned and ad hoc queries, which is key to a successful data warehouse.

This section describes the main requirements and basic concepts of synchronous refresh, and includes the following:


- What Is Synchronous Refresh?
- Why Use Synchronous Refresh?
- Registering Tables and Materialized Views for Synchronous Refresh
- Specifying Change Data for Refresh
- Synchronous Refresh Preparation and Execution
- Materialized View Eligibility Rules and Restrictions for Synchronous Refresh


### 8.1.1 What Is Synchronous Refresh?

Synchronous refresh is a new approach for maintaining tables and materialized views in a data warehouse where tables and materialized views are refreshed at the same time. In traditional refresh methods, the changes are applied to the base tables and the materialized views are refreshed separately with one of the following refresh methods:



Copyright © 2001, 2025, Oracle and/or its affiliates.

- Log-based incremental (fast) refresh using materialized view logs if such logs are available
- PCT refresh if it is applicable
- Complete refresh

Synchronous refresh combines some elements of log-based incremental (fast) refresh and PCT refresh methods, but it is applicable only to `ON DEMAND` materialized views, unlike the other two methods. There are three major differences between it and the other refresh methods:

- Synchronous refresh requires you to register the tables and materialized views.
- Synchronous refresh requires you to specify changes to the data according to some formally specified rules.
- Synchronous refresh works by dividing the refresh operation into two steps: preparation and execution. This approach provides some important advantages over the other methods, such as better performance and more control.

Synchronous refresh APIs are defined in a new package called `DBMS_SYNC_REFRESH`. For more information about this package, see *Oracle AI Database PL/SQL Packages and Types Reference*.

## 8.1.2 Why Use Synchronous Refresh?

Synchronous refresh offers the following advantages over traditional types of methods used to refresh materialized views in a data warehouse:

- It coordinates the loading of the changes into the base tables with the extremely efficient refresh of the dependent materialized views themselves.
- It decreases the time materialized views are not available to be used by the Optimizer to rewrite queries.
- It is well-suited for a wide class of materialized views (materialized aggregate views and materialized join views) commonly used in data warehouses. It does require the materialized views be partitioned as well as the fact tables, and if materialized views are not currently partitioned, they can be efficiently partitioned to take advantage of synchronous refresh.
- It fully exploits partitioning and the nature of the data warehouse load cycle to guarantee synchronization between the materialized view and the base table throughout the refresh procedure.
- In a typical data warehouse, data preparation consists of extracting the data from one or more sources, cleansing, and formatting it for consistency, and transforming into the data warehouse schema. The data preparation area is called the staging area and the base tables in a data warehouse are loaded from the tables in the staging area. The synchronous refresh method fits into this model because it allows you to load change data into the staging logs.
- The staging logs play the same role as materialized view logs in the conventional fast refresh method. There is, however, an important difference. In the conventional fast refresh method, the base table is first updated and the changes are then applied from the materialized view log to the materialized views. But in the synchronous refresh method, the changes from the staging log are applied to refresh the materialized views while also being applied to the base tables.
- Most materialized views in a data warehouse typically employ a star or snowflake schema with fact and dimension tables joined in a foreign key to primary key relationship. The synchronous refresh method can handle both schemas in all possible change data load


Copyright © 2001, 2025, Oracle and/or its affiliates.

scenarios, ranging from rows being added to only the fact table, to arbitrary changes to the fact and dimension tables.

- Instead of providing the change load data in the staging logs, you have a choice of directly providing the change data in the form of outside tables containing the data to be exchanged with the affected partition in the base table. This capability is provided by the `REGISTER_PARTITION_OPERATION` procedure in the `DBMS_SYNC_REFRESH` package.

## 8.1.3 Registering Tables and Materialized Views for Synchronous Refresh

Before actually performing synchronous refresh, you must register the appropriate tables and materialized views. Synchronous refresh provides these methods to register tables and materialized views:

- Tables are registered with synchronous refresh by creating a **staging log** on them. A staging log is created with the `CREATE MATERIALIZED VIEW LOG` statement whose syntax has been extended in this release to create staging logs as well as the familiar materialized view logs used for the traditional incremental refresh. After you create a staging log on a table, it is deemed to be registered with synchronous refresh and can be modified only by using the synchronous refresh procedures. In other words, a table with a staging log defined on it is registered with synchronous refresh and cannot be modified directly by the user.
- Materialized views are registered with synchronous refresh using the `REGISTER_MVIEWS` procedure in the `DBMS_SYNC_REFRESH` package. The `REGISTER_MVIEWS` procedure implicitly creates groups of related objects called sync refresh groups. A **sync refresh group** consists of all related materialized views and tables that must be refreshed together as a single entity because they are dependent on one another.

> **See Also**
>
> - *Oracle AI Database SQL Language Reference* for more information about the `CREATE MATERIALIZED VIEW LOG` statement
> - *Oracle AI Database PL/SQL Packages and Types Reference* for more information about the `DBMS_SYNC_REFRESH` package

## 8.1.4 Specifying Change Data for Refresh

In the other refresh methods, you can directly modify the base tables of the materialized view, and the issue of specifying change data does not arise. But with synchronous refresh, you are required to specify and prepare the change data according to certain formally specified rules and using APIs provided by the `DBMS_SYNC_REFRESH` package.

There are two ways to specify the change data:

- Provide the change data in an outside table and register it with the `REGISTER_PARTITION_OPERATION` procedure.

  See "Working with Partition Operations While Capturing Change Data for Synchronous Refresh" for more details.
- Provide the change data by in staging logs and process them with the `PREPARE_STAGING_LOG` procedure. The format of the staging logs and rules for populating are described in "Working with Staging Logs While Capturing Change Data for


Copyright © 2001, 2025, Oracle and/or its affiliates.

Synchronous Refresh". You are required to run the PREPARE_STAGING_LOG procedure for every table before performing the refresh operation on that table.

## 8.1.5 Synchronous Refresh Preparation and Execution

After preparing the change data, you can perform the actual refresh operation. Synchronous refresh takes a new approach to refresh execution. It works by dividing the refresh operation into two steps: preparation and execution. This is one of the main differences between it and the other refresh methods and provides some important benefits.

The preparation step determines the mapping between the fact table partitions and the materialized view partitions. This step computes the new tables corresponding only to the partitions of the fact table that have been changed by the incremental change data load. After these tables, called **outside tables**, have been computed, the actual execution of the refresh operation takes place in the execution step, which consists of just exchanging the outside tables with the corresponding partitions in the fact table or materialized view.

By dividing the refresh execution step into two phases and providing separate procedures for them, synchronous refresh not only provides you control over the refresh execution process, but also improves overall system performance. It does this by minimizing the time the materialized views are not available for use by direct access or the Optimizer because they are modified by the refresh process. During the preparation phase, the materialized view and its tables are not modified because at this time all the refresh changes are recorded in the outside table. Consequently, the materialized view is available to any query that needs to read them. It is only during execution that the tables and materialized views are modified. Execution performance is mainly affected by the number of changes to the dimension tables; if this number is small, then the performance should be very good because the exchange partition operations are themselves very fast.

The DBMS_SYNC_REFRESH package provides the PREPARE_REFRESH and EXECUTE_REFRESH procedures to perform these two steps.

**See Also**

- *Oracle AI Database PL/SQL Packages and Types Reference*

## 8.1.6 Materialized View Eligibility Rules and Restrictions for Synchronous Refresh

The primary requirement for a materialized view to be eligible for synchronous refresh is that the materialized view must be partitioned with a key that can be derived from the partition key of its fact table. The following sections describe the other requirements for eligibility for synchronous refresh.

This section contains the following topics:

- Synchronous Refresh Restrictions: Partitioning
- Synchronous Refresh Restrictions: Refresh Options
- Synchronous Refresh Restrictions: Constraints
- Synchronous Refresh Restrictions: Tables
- Synchronous Refresh Restrictions: Materialized Views


Copyright © 2001, 2025, Oracle and/or its affiliates.

- Synchronous Refresh Restrictions: Materialized Views with Aggregates

## 8.1.6.1 Synchronous Refresh Restrictions: Partitioning

There are two key requirements to use synchronous refresh:

- The materialized view must be partitioned along the same dimension as the fact table.
- The partition key of the fact table should functionally determine the partition key of the materialized view.

The term *functionally determine* means the partition key of the materialized view can be derived from the partition key of the fact table based on a foreign key constraint relationship. This condition is satisfied if the partition key of the materialized view is the same as that for the fact table or related by joins from the fact table to the dimension table as in a star or snowflake schema. For example, if the fact table is partitioned by a date column, such as `TIME_KEY`, the materialized view can be partitioned by `TIME_KEY`, `MONTH`, or `YEAR`.

Synchronous refresh supports two types of partitioning on fact tables and materialized views: range partitioning and composite partitioning, when the top-level partitioning type is range.

## 8.1.6.2 Synchronous Refresh Restrictions: Refresh Options

When you define a materialized view, you can specify three refresh options: how to refresh; whether trusted constraints can be used; and what type of refresh is to be performed. If unspecified, the defaults are assumed to be `ON DEMAND`, `ENFORCED` constraints, and `FORCE` respectively. Synchronous refresh requires that the first two of these options must have the values `ON DEMAND` and `TRUSTED` constraints respectively. Synchronous refresh does not require the type of refresh to have any specific value, so it can be `FAST`, `FORCE`, or `COMPLETE`.

## 8.1.6.3 Synchronous Refresh Restrictions: Constraints

The relationships between the fact and dimension tables are declared by foreign and primary key constraints on the tables. Synchronous refresh trusts these constraints to perform the refresh, and requires that `USING TRUSTED CONSTRAINTS` must be specified in the materialized view definition. This allows using nonvalidated `RELY` constraints and rewriting against materialized views in an `UNKNOWN` or `FRESH` state during refresh.

When a table is registered for synchronous refresh, its constraints might be in a `VALIDATE` or `NOVALIDATE` state. If the table is a dimension table, synchronous refresh will retain this state during the refresh execution process.

However, if the table is a fact table, synchronous refresh marks the constraints `NOVALIDATE` state during refresh execution. This avoids the need for validating the constraint on existing data during a partition exchange that is the basis of the synchronous refresh method, and improves the performance of refresh execution.

Because the constraints on the fact table are not enforced by synchronous refresh, it is you who must verify the integrity and consistently of the data provided.

## 8.1.6.4 Synchronous Refresh Restrictions: Tables

To be eligible for synchronous refresh, a table must satisfy the following conditions:

- The table cannot have VPD or triggers defined on it.
- The table cannot have any `RAW` type.
- The table cannot be remote.


Copyright © 2001, 2025, Oracle and/or its affiliates.

- The staging log key of each table registered for synchronous refresh should satisfy the requirements described in "About the Staging Log Key".

### 8.1.6.5 Synchronous Refresh Restrictions: Materialized Views

There are some other restrictions that are specific to materialized views registered for synchronous refresh:

- The `ROWID` column cannot be used to define the query. It is not relevant because it uses partition exchange, which replaces the original partition with the outside table. Hence, the defining query should not include any `ROWID` columns.
- Synchronous refresh does not support nested materialized views, `UNION ALL` materialized views, subqueries, or complex queries in the materialized view definition. The defining query must conform to the star or snowflake schema.
- These SQL constructs are also not supported: analytic window functions (such as `RANK`), the `MODEL` clause, and the `CONNECT BY` clause.
- Synchronous refresh is not supported for a materialized view that refers to views, remote tables, or outer joins.
- The materialized view must not contain references to nonrepeating expressions like `SYSDATE` and `ROWNUM`.

In general, most restrictions that apply to PCT-refresh, fast refresh, and general query rewrite also apply to synchronous refresh. Those restrictions are available at:

- "About Materialized View Restrictions for Query Rewrite"
- "General Query Rewrite Restrictions"
- "General Restrictions on Fast Refresh"

### 8.1.6.6 Synchronous Refresh Restrictions: Materialized Views with Aggregates

For materialized views with aggregates, synchronous refresh shares these restrictions with fast refresh:

- Only `AVG`, `BIT_AND_AGG`, `BIT_OR_AGG`, `BIT_XOR_AGG`, `COUNT`, `KURTOSIS_POP`, `KURTOSIS_SAMP`, `MIN`, `MAX`, `STDDEV`, `SUM`, `SKEWNESS_POP`, `SKEWNESS_SAMP`, and `VARIANCE` are supported.
- You must specify `COUNT(*)`.
- Aggregate functions must occur only as the outermost part of the expression. That is, aggregates such as `AVG(AVG(x))` or `AVG(x)+ AVG(x)` are not allowed.
- For each aggregate, such as `AVG(expr)`, the corresponding `COUNT(expr)` must be present. Oracle recommends that you specify `SUM(expr)`.
- If you specify `VARIANCE(expr)` or `STDDEV(expr)`, you must also specify `COUNT(expr)` and `SUM(expr)`. Oracle recommends that you specify `SUM(expr *expr)`.
- If you specify `KURTOSIS_POP`, `KURTOSIS_SAMP`, `SKEWNESS_POP`, or `SKEWNESS_SAMP`, you must also specify `COUNT(expr)` and `SUM(expr)`. For `SKEWNESS_POP` and `SKEWNESS_SAMP`, you must also specify `VARIANCE(expr)` and `COUNT(*)`.

## 8.2 Using Synchronous Refresh for Materialized Views

Synchronous refresh differs from the other refresh methods in a number of ways. One is that the API for synchronous refresh is contained in a new package called `DBMS_SYNC_REFRESH`,


Copyright © 2001, 2025, Oracle and/or its affiliates.

whereas other refresh methods are declared in the DBMS_MVIEW package. Another difference is that after objects are registered with synchronous refresh, and, once registered, the other refresh methods cannot be used with them.

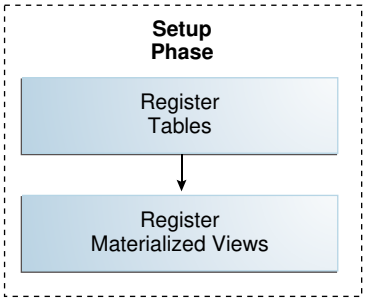
The operations associated with synchronous refresh can be divided into the following three broad phases:

- Synchronous Refresh Step 1: Registration Phase
- Synchronous Refresh Step 2: Synchronous Refresh Phase
- Synchronous Refresh Step 3: The Unregistration Phase

## 8.2.1 Synchronous Refresh Step 1: Registration Phase

In this phase (Figure 8-1), you register the objects for use with synchronous refresh. The two steps in this phase are registration of tables first and then materialized views. You register the tables (by creating staging logs) and materialized views (with the REGISTER_MVIEWS procedure). The staging logs are created with the CREATE MATERIALIZED LOG … FOR SYNCHRONOUS REFRESH statement. If a table already has a regular materialized view log, the ALTER MATERIALIZED LOG … FOR SYNCHRONOUS REFRESH statement can be used to convert it to a staging log.

**Figure 8-1 Registration Phase**




You can create a staging log with a statement, as show in Example 8-1.

**Example 8-1 Registering Tables**

```
CREATE MATERIALIZED VIEW LOG ON fact
FOR SYNCHRONOUS REFRESH USING st_fact;
```

If a table has a materialized view log, you can alter it to a staging log with a statement, such as the following:

```
ALTER MATERIALIZED VIEW LOG ON fact
FOR SYNCHRONOUS REFRESH USING st_fact;
```

You can register a materialized view with a statement, as shown in Example 8-2.

**Example 8-2 Registering Materialized Views**

```
EXECUTE DBMS_SYNC_REFRESH.REGISTER_MVIEWS('MV1');
```

You can register multiple materialized views at one time:

```
EXECUTE DBMS_SYNC_REFRESH.REGISTER_MVIEWS('mv2, mv2_year, mv1_halfmonth');
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 8.2.2 Synchronous Refresh Step 2: Synchronous Refresh Phase

Figure 8-2 shows the synchronous refresh phase. This phase can be used repeatedly to perform synchronous refresh. The three main steps in this phase are:

1. Prepare the change data for the refresh operation. You can provide the change data in a table and register it with the `REGISTER_PARTITION_OPERATION` procedure or provide the data by populating the staging logs. The staging logs must be processed with the `PREPARE_STAGING_LOG` procedure before proceeding to the next step.

   An example is Example 8-12.

2. Perform the first step of the refresh operation (`PREPARE_REFRESH`). This can potentially be a long-running operation because it prepares and loads the outside tables.

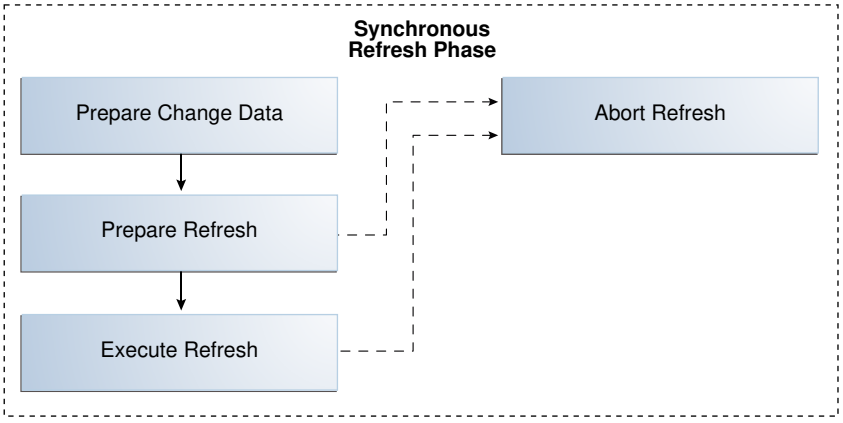
   An example is Example 8-16.

3. Perform the second and last step of the refresh operation (`EXECUTE_REFRESH`). This usually runs very fast because it usually consists of a series of partition-exchange operations.

   An example is Example 8-20.

In Figure 8-2, solid arrows show the standard control flow and dashed arrows are used for error-handling cases. If either of the refresh operations (`PREPARE_REFRESH` or `EXECUTE_REFRESH`) raises user errors, you use an `ABORT_REFRESH` procedure to restore tables and materialized views to the state that existed before the refresh operation, fix the problem, and retry the refresh operation starting from the beginning.

**Figure 8-2 Refresh Phase**

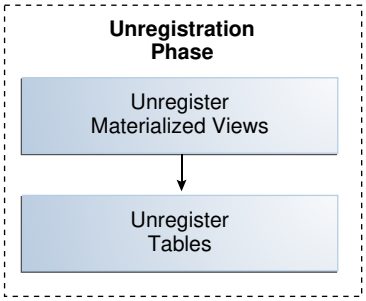



## 8.2.3 Synchronous Refresh Step 3: The Unregistration Phase

If you choose to stop using synchronous refresh, then you must unregister the materialized views as shown in Figure 8-3. The materialized views are first unregistered with the `UNREGISTER_MVIEWS` procedure. The tables are then unregistered by either dropping their staging logs or altering the staging logs to ordinary logs. Note that if the staging logs are converted to be ordinary materialized view logs with an `ALTER MATERIALIZED LOG … FOR FAST REFRESH` statement, then the materialized views can be maintained with standard fast-refresh methods.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Figure 8-3 Unregistration Phase**




Example 8-3 illustrates how to unregister the single materialized view MV1.

**Example 8-3 Unregister Materialized Views**

```
EXECUTE DBMS_SYNC_REFRESH.UNREGISTER_MVIEWS('MV1');
```

You can unregister multiple materialized views at one time:

```
EXECUTE DBMS_SYNC_REFRESH.UNREGISTER_MVIEWS('mv2, mv2_year, mv1_halfmonth');
```

You can verify to see that a materialized view has been unregistered by querying the DBA_SR_OBJ_ALL view.

Example 8-4 illustrates how to drop the staging log.

**Example 8-4 Unregister Tables**

```
DROP MATERIALIZED VIEW LOG ON fact;
```

Or you can alter the table to a materialized view log:

```
ALTER MATERIALIZED VIEW LOG ON fact
FOR FAST REFRESH;
```

You can verify to see that a table has been unregistered by querying the DBA_SR_OBJ_ALL view.

# 8.3 Using Synchronous Refresh Groups

The distinguishing feature of synchronous refresh is that changes to a table and its materialized views are loaded and refreshed together, hence the name synchronous refresh. For tables and materialized views to be maintained by synchronous refresh, the objects must be registered. Tables are registered for synchronous refresh when staging logs are created on them, and materialized views are registered using the REGISTER_MVIEWS procedure.

Synchronous refresh supports the refresh of materialized views built on multiple tables, with changes in one or more of them. Tables that are related by constraints must all necessarily be refreshed together to ensure data integrity. Furthermore, it is possible that some of the tables registered for synchronous refresh have several materialized views built on top of them, in which case, all those materialized views must also be refreshed together.

Instead of having you keep track of these dependencies, and issue the refresh commands on the right set of tables, Oracle AI Database automatically generates the minimal sets of tables and materialized views that must necessarily be refreshed together. These sets are termed synchronous refresh groups or just sync refresh groups. Each sync refresh group is identified by a GROUP_ID.value.


Copyright © 2001, 2025, Oracle and/or its affiliates.

The three procedures related to performing synchronous refresh (PREPARE_REFRESH, EXECUTE_REFRESH and ABORT_REFRESH) take as input either a single group ID or a list of group IDs identifying the sync refresh groups.

Each table or materialized view registered for synchronous refresh is assigned a GROUP_ID value, which may change over time, if the dependencies among them change. This happens when you issue the REGISTER_MVIEWS and UNREGISTER_MVIEWS procedures. The examples that follow show the sync refresh groups in a number of scenarios.

Because the GROUP_ID value can change with time, Oracle recommends the actual GROUP_ID value not be used when invoking the synchronous refresh procedures, but that the function DBMS_SYNC_REFRESH.GET_GROUP_ID be used instead. This function takes a materialized view name as input and returns the materialized view's GROUP_ID value.

> **See Also**
>
> *Oracle AI Database PL/SQL Packages and Types Reference* for information about how to use the DBMS_SYNC_REFRESH.REGISTER_MVIEWS procedure

This section contains the following topics:

- Examples of Common Actions with Synchronous Refresh Groups
- Examples of Working with Multiple Synchronous Refresh Groups

## 8.3.1 Examples of Common Actions with Synchronous Refresh Groups

The synchronous refresh demo scripts in the rdbms/demo directory enable you to view typical operations that you are likely to perform. The main script is syncref_run.sql, and its log is syncref_run.log. Example 8-5, Example 8-6, and Example 8-7 below illustrate the different contexts in which the GET_GROUP_ID function can be used.

**Example 8-5 Display the Objects Registered in a Group**

This example illustrates how to display the objects registered in a group after registering them.

```
EXECUTE DBMS_SYNC_REFRESH.REGISTER_MVIEWS('MV1');
SELECT NAME, TYPE, STAGING_LOG_NAME FROM USER_SR_OBJ
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1')
ORDER BY TYPE, NAME;

NAME       TYPE       STAGING_LOG_NAME
---------- ---------- ----------------
MV1        MVIEW
FACT       TABLE      ST_FACT
STORE      TABLE      ST_STORE
TIME       TABLE      ST_TIME
```

**Example 8-6 Invoke Refresh Operations**

This example illustrates how to invoke refresh operations.

```
EXECUTE DBMS_SYNC_REFRESH.PREPARE_REFRESH( -
        DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));
EXECUTE DBMS_SYNC_REFRESH.EXECUTE_REFRESH( -
        DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));
SELECT NAME, TYPE, STATUS FROM USER_SR_OBJ_STATUS
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1')
ORDER BY TYPE, NAME;
```

**Example 8-7 Verify the Status of Objects Registered in a Group**

This example illustrates how to verify the status of objects registered in a group after an EXECUTE_REFRESH operation.

```
SELECT NAME, TYPE, STATUS FROM USER_SR_OBJ_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1')
ORDER BY TYPE, NAME;

NAME             TYPE       STATUS
---------------- ---------- ----------------
MV1              MVIEW      COMPLETE
FACT             TABLE      COMPLETE
STORE            TABLE      COMPLETE
TIME             TABLE      COMPLETE
```

## 8.3.2 Examples of Working with Multiple Synchronous Refresh Groups

You can work with multiple refresh groups at one time with the following APIs:

- GET_GROUP_ID_LIST

  Takes a list of materialized views as input and returns their group IDs in a list.

- GET_ALL_GROUP_IDS

  Returns the group IDs of all groups in the system in a list.

- The prepare refresh procedures (PREPARE_REFRESH, EXECUTE_REFRESH, and ABORT_REFRESH) can work multiple groups. Their overloaded versions accept lists of group IDs at a time.

**Example 8-8 Prepare Sync Refresh Groups**

This example illustrates how to prepare the sync refresh groups of MV1, MV2, and MV3.

```
EXECUTE DBMS_SYNC_REFRESH.PREPARE_REFRESH(
        DBMS_SYNC_REFRESH.GET_GROUP_ID_LIST('MV1, MV2, MV3'));
```

Note that it is not necessary that these three materialized views be all in different groups. It is possible that two of the materialized views are in one group, and third in another; or even that all three materialized views are in the same group. Because PREPARE_REFRESH is overloaded to accept either a group ID or a list of group IDs, the above call will work in all cases.

**Example 8-9 Execute Sync Refresh Groups**

This example illustrates how to prepare and execute the refresh of all sync refresh groups in the system.

```
EXECUTE DBMS_SYNC_REFRESH.PREPARE_REFRESH(
        DBMS_SYNC_REFRESH.GET_ALL_GROUP_IDS);

EXECUTE DBMS_SYNC_REFRESH.EXECUTE_REFRESH(
        DBMS_SYNC_REFRESH.GET_ALL_GROUP_IDS);
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

# 8.4 Specifying and Preparing Change Data for Synchronous Refresh

Synchronous refresh requires you to specify and prepare the change data that serves as the input to the PREPARE_REFRESH and EXECUTE_REFRESH procedures. There are two methods for specifying the change data:

- Provide the change data in an outside table and register it with the REGISTER_PARTITION_OPERATION procedure as described in Working with Partition Operations While Capturing Change Data for Synchronous Refresh.
- Provide the change data by in staging logs and process them with the PREPARE_STAGING_LOG procedure as described in Working with Staging Logs While Capturing Change Data for Synchronous Refresh.

Some important points about change data are:

- The two methods are not mutually exclusive and can be employed at the same time, even on the same table, but there cannot be any conflicts in the changes specified. For instance, you can use the staging log to specify the change in a partition with a small number of changes, but if another partition has extensive changes, you can provide the changes for that partition in an outside table.
- For dimension tables, you can use only the staging logs to provide changes.
- Synchronous refresh can handle arbitrary combinations of changes in fact and dimension tables, but it is optimized for the most common data warehouse usage scenarios, where the bulk of the changes are made to only a few partitions of the fact table.
- Synchronous refresh places no restrictions on the use of nondestructive partition maintenance operations (PMOPS), such as add partition, used commonly in data warehouses. The use of such PMOPS is not directly related to the method used to specify change data.
- Synchronous refresh requires that all staging logs in the group must be prepared, even if the staging log has no changes registered in it.

## 8.4.1 Working with Partition Operations While Capturing Change Data for Synchronous Refresh

Using the REGISTER_PARTITION_OPERATION procedure, you can provide the change data directly. This method is applicable only to fact tables. For each fact table partition that is changed, you must provide an outside table containing the data for that partition. The synchronous refresh demo (syncref_run.sql and syncref_run.log) contains an example. The steps are:

1. Create an outside table for the partition that it is intended to replace. It must have the same constraints as the fact table, and can be created in any desired tablespace.

```
CREATE TABLE      fact_ot_fp3(
   time_key       DATE NOT NULL REFERENCES time(time_key),
   store_key      INTEGER NOT NULL REFERENCES store(store_key),
   dollar_sales   NUMBER (6,2),
   unit_sales     INTEGER)
   tablespace     syncref_fp3_tbs;
```

2. Insert the data for this partition into the outside table.


Copyright © 2001, 2025, Oracle and/or its affiliates.

3. Register this table for partition exchange.

```
begin
   DBMS_SYNC_REFRESH.REGISTER_PARTITION_OPERATION(
       partition_op                => 'EXCHANGE',
       schema_name                 => 'SYNCREF_USER',
       base_table_name             => 'FACT',
       partition_name              => 'FP3',
       outside_partn_table_schema  => 'SYNCREF_USER',
       outside_partn_table_name    => 'FACT_OT_FP3');
   end;
  /
/
```

When you register the outside table and execute the refresh, Oracle AI Database performs the following operation at `EXECUTE_REFRESH` time:

```
ALTER TABLE FACT EXCHANGE PARTITION fp3 WITH TABLE fact_ot_fp3
INCLUDING INDEXES WITHOUT VALIDATION;
```

However, you are not allowed to issue the above statement directly on your own. If you do, Oracle AI Database will give this error:

```
ORA-31908: Cannot modify the contents of a table with a staging log.
```

Besides the `EXCHANGE` operation, the two other partition operations that can be registered with the `REGISTER_PARTITION_OPERATION` procedure are `DROP` and `TRUNCATE`.

**Example 8-10 Registering a DROP Operation**

This example illustrates how to specify the drop of the first partition (`FP1`), by using the following statement.

```
begin
 DBMS_SYNC_REFRESH.REGISTER_PARTITION_OPERATION(
              partition_op                => 'DROP',
              schema_name                 => 'SYNCREF_USER',
              base_table_name             => 'FACT',
              partition_name              => 'FP1');
end;
/
```

If you wanted to truncate the partition instead, you could specify `TRUNCATE` instead of `DROP` for the `partition_op` parameter.

The three partition operations (`EXCHANGE`, `DROP`, and `TRUNCATE`) are called destructive PMOPS because they modify the contents of the table. The following partition operations are not destructive, and can be performed directly on a table registered with synchronous refresh:

- `ADD PARTITION`
- `SPLIT PARTITION`
- `MERGE PARTITIONS`
- `MOVE PARTITION`
- `RENAME PARTITION`

In data warehouses, these partition operations are commonly used to manage the large volumes of data, and synchronous refresh places no restrictions on their usage. Oracle AI Database requires only that these operations be performed before the `PREPARE_REFRESH` command is issued. This is because the `PREPARE_REFRESH` procedure computes the mapping between the fact table partitions and the materialized view partitions, and if any partition-


Copyright © 2001, 2025, Oracle and/or its affiliates.

maintenance is done between the PREPARE_REFRESH and EXECUTE_REFRESH procedures, Oracle AI Database will detect this at EXECUTE_REFRESH and show an error.

You can use the USER_SR_PARTN_OPS catalog view to display the registered partition operations.

```
SELECT   TABLE_NAME, PARTITION_OP, PARTITION_NAME,
         OUTSIDE_TABLE_SCHEMA ot_schema, OUTSIDE_TABLE_NAME ot_name
FROM     USER_SR_PARTN_OPS
ORDER BY TABLE_NAME;

TABLE_NAME PARTITION_ PARTITION_NAME  OT_SCHEMA       OT_NAME
---------- ---------- --------------- --------------- --------------------
FACT       EXCHANGE   FP3             SYNCREF_USER    FACT_OT_FP3

1 row selected.
```

These partition operations are consumed by the synchronous refresh operation and are automatically unregistered by the EXECUTE_REFRESH procedure. So if you query USER_SR_PARTN_OPS after EXECUTE_REFRESH, it will show no rows.

After registering a partition, if you find you made a mistake or change your mind, you can undo it with the UNREGISTER_PARTITION_OPERATION command:

```
begin
 DBMS_SYNC_REFRESH.UNREGISTER_PARTITION_OPERATION(
              partition_op                  => 'EXCHANGE',
              schema_name                   => 'SYNCREF_USER',
              base_table_name               => 'FACT',
              partition_name                => 'FP3');
end;
/
```

## 8.4.2 Working with Staging Logs While Capturing Change Data for Synchronous Refresh

In synchronous refresh, staging logs play a role similar to materialized view logs in incremental refresh. They are created with a DDL statement and can be altered to a materialized view log. Unlike materialized view logs, however, you are responsible for loading changes into the staging logs in a specified format. Each row in the staging log must have a key to identify it uniquely; this key is called the **staging log key**, and is defined in "About the Staging Log Key".

You are responsible for populating the staging log, which will consist of all the columns in the base table and an additional control column DMLTYPE$$ of type CHAR(2). This must have the value 'I' to denote the row is being inserted, 'D' for delete, and 'UN' and 'UO' for the new and old values of the row being updated, respectively. The last two must occur in pairs.

The staging log is validated by the PREPARE_STAGING_LOG procedure and consumed by the synchronous refresh operations (PREPARE_REFRESH and EXECUTE_REFRESH). During validation by PREPARE_STAGING_LOG, if errors are detected, they will be captured in an exceptions table. You can query the view USER_SR_STLOG_EXCEPTIONS to get details on the exceptions.

Synchronous refresh requires that, before calling PREPARE_REFRESH for sync refresh groups, the staging logs of all tables in the group must be processed with PREPARE_STAGING_LOG. This is necessary even if a table has no change data and its staging log is empty.

This section contains the following topics:

- About the Staging Log Key
- About Staging Log Rules


Copyright © 2001, 2025, Oracle and/or its affiliates.

- About Columns Being Updated to NULL
- Examples of Working with Staging Logs
- Error Handling in Preparing Staging Logs

## 8.4.2.1 About the Staging Log Key

In order to create a staging log on a base table, the base table must have a key. If the table has a primary key, the primary key is deemed to be staging log key on the table's staging log. Note that every dimension table has a primary key.

With fact tables, it is less common for them to have a primary key. If a table does not have a primary key, the columns that are the foreign keys of its dimension tables constitute its staging log key.

The key of a staging log can be described as:

- The primary key of the base table. If a fact table has a primary key, it is sometimes called a surrogate key.
- The set of foreign keys for a fact table. This applies if the fact table does not have a primary key. This assumption is common in data warehouses, though it is not enforced.

The rules for loading staging logs are described in "About Staging Log Rules".

The `PREPARE_STAGING_LOG` procedure verifies that each key value is specified at most once. When populating the staging log, it is your responsibility to consolidate the changes if a row with the same key value is changed more than once. This process is known as **change consolidation**. When doing the change consolidation, you must:

- Consolidate a delete-insert of the same row into an update operation with rows `'UO'` and `'UN'`.
- Consolidate multiple updates into a single update.
- Prevent null changes such as an insert-update-delete of the same row from appearing in the staging log.
- Consolidate an insert followed by multiple updates into a single insert.

## 8.4.2.2 About Staging Log Rules

Every row should contain non-null values for all the columns comprising the primary key. You are required to consolidate all the changes so that each key in the staging log can be specified only for one type of operation.

For the rows being inserted (`DMLTYPE$$` is `'I'`), all columns in the staging log must be supplied with valid values, conforming to any constraint on the corresponding columns in the base table. Keys of rows being inserted must not exist in the base table.

For the rows being deleted (`DMLTYPE$$` is `'D'`), the non-key column values are optional. Similarly, for the rows specifying the old values of the columns being updated (`DMLTYPE$$` is `'UO'`), the non-key column values are optional; an important exception is the column whose values are being updated to `NULL`, as explained subsequently.

For the rows specifying the new values of the columns being updated (`DMLTYPE$$` is `'UN'`), the non-key column values are optional except for the values of the columns that were changed.


Copyright © 2001, 2025, Oracle and/or its affiliates.

### 8.4.2.3 About Columns Being Updated to NULL

If a column is being updated to `NULL`, its old value must be specified. Otherwise, Oracle AI Database may not be able to distinguish this from a column whose value is being left unchanged in the update.

For example, let table `T1` have three columns `c1`, `c2`, and `c3`. Let there be a row with `(c1, c2, c3) = (1, 5, 10)`, and you supply the following information in the staging log:

| DMLTYPE$$ | C1 | C2 | C3 |
|---|---|---|---|
| UO | 1 | NULL | NULL |
| UN | 1 | NULL | 11 |

The result would be that the new row could be `(1, 5, 11)` or `(1, NULL, 11)` without having specified the old value. However, with that specification, it is clear the new row is `(1, 5, 11)`. If you want to specify `NULL` for `c2`, you should specify the old value in the `UO` row as follows:

| DMLTYPE$$ | C1 | C2 | C3 |
|---|---|---|---|
| UO | 1 | 5 | NULL |
| UN | 1 | NULL | 11 |

Because the old value of `c2` is `5`, (the correct previously updated value for the column), its new value, will be `NULL` and the new row is `(1, NULL, 11)`.

### 8.4.2.4 Examples of Working with Staging Logs

This section illustrates examples of working with staging logs.

The `PREPARE_STAGING_LOG` procedure has an optional third parameter called `PSL_MODE`. This allows you to specify whether any or all of the three types of DML statements specified in the staging log can be treated as trusted, and not be subject to verification by the `PREPARE_STAGING_LOG` procedure, as shown in Example 8-11.

**Example 8-11 Specifying Trusted DML Statements**

```
EXECUTE DBMS_SYNC_REFRESH.PREPARE_STAGING_LOG('syncref_user', 'store',
                                 DBMS_SYNC_REFRESH.INSERT_TRUSTED +
                                 DBMS_SYNC_REFRESH.DELETE_TRUSTED);
```

This call will skip verification of `INSERT` and `DELETE` DML statements in the staging log of `STORE` but will verify `UPDATE` DML statements.

**Example 8-12 Preparing Staging Logs**

This example is taken from the demo `syncref_run.sql`. It shows that the user has provided values for all columns for the delete and update operations. This is recommended if these values are available.

```
INSERT INTO st_store (dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE)
VALUES ('I',   5, 5, 'Store 5', '03060');

INSERT INTO st_store (dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE)
VALUES ('I',   6, 6, 'Store 6', '03062');

INSERT INTO st_store (dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE)
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
VALUES ('UO',  4, 4, 'Store 4', '03062');

INSERT INTO st_store (dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE)
VALUES ('UN',  4, 4, 'Stor4NewNam', '03062');

INSERT INTO st_store (dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE)
VALUES ('D',   3, 3, 'Store 3', '03060');

EXECUTE DBMS_SYNC_REFRESH.PREPARE_STAGING_LOG('syncref_user', 'store');

-- display initial contents of st_store

SELECT dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE
FROM st_store
ORDER BY STORE_KEY ASC, dmltype$$ DESC;

DM  STORE_KEY  STORE_NUMBER  STORE_NAME    ZIPCODE
--  ---------  ------------  ----------    -------
D            3             3  Store 3         03060
UO           4             4  Store 4         03062
UN           4             4  Stor4NewNam     03062
I            5             5  Store 5         03060
I            5             5  Store 6         03062

5 rows selected.
```

### Example 8-13 Filling in Missing Values for Deleting and Updating Records

This example shows that if you do not supply all the values for the delete and update operations, then when you run the `PREPARE_STAGING_LOG` procedure, Oracle AI Database will fill in missing values.

```
INSERT INTO st_store (dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE)
VALUES ('D',  3, NULL, NULL, NULL);

INSERT INTO st_store (dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE)
VALUES ('UO',  4, NULL, NULL, NULL);

INSERT INTO st_store (dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE)
VALUES ('UN',  4, NULL, NULL, '03063');

EXECUTE DBMS_SYNC_REFRESH.PREPARE_STAGING_LOG('syncref_user', 'store');

SELECT dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE
FROM ST_STORE ORDER BY STORE_KEY ASC, dmltype$$ DESC;

DM STORE_KEY STORE_NUMBER STORE_NAME  ZIPCODE
-- --------- ------------ ----------- ---------
D          3            3 Store 3     03060
UO         4            4 Store 4     03062
UN         4            4 Store 4     03063
```

### Example 8-14 Updating a Column to NULL

This example illustrates how to update a column to `NULL`. If you want to update a column value to `NULL`, then you must provide its old value in the `UO` record.

In this example, your goal is to change the zipcode of store 4 to 03063 and its name to `NULL`. You can supply the old zipcode value, but you must supply the old value of `store_name` in the `'UO'` row, or else `store_name` will be unchanged.


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
INSERT INTO st_store (dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE)
VALUES ('UO',  4, NULL, 'Store 4', NULL);

INSERT INTO st_store (dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE)
VALUES ('UN',  4, NULL, NULL, '03063');

EXECUTE DBMS_SYNC_REFRESH.PREPARE_STAGING_LOG('syncref_user', 'store');

SELECT dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE
FROM st_store ORDER BY STORE_KEY ASC, dmltype$$ DESC;

DM STORE_KEY STORE_NUMBER STORE_NAME  ZIPCODE
-- --------- ------------ ----------- --------
UO         4            4 Store 4     03062
UN         4            4             03063
```

**Example 8-15 Displaying Staging Log Statistics**

This example illustrates how to use the USER_SR_STLOG_STATS catalog view to display the staging log statistics.

```
SELECT   TABLE_NAME, STAGING_LOG_NAME, NUM_INSERTS, NUM_DELETE, NUM_UPDATES
FROM     USER_SR_STLOG_STATS
ORDER BY TABLE_NAME;

TABLE_NAME STAGING_LOG_NAME NUM_INSERTS NUM_DELETES NUM_UPDATES
---------- ---------------- ----------- ----------- -----------
FACT       ST_FACT                    4           1           1
STORE      ST_STORE                   2           1           1
TIME       ST_TIME                    1           0           0

3 rows selected.
```

If you use the same query at the end of the EXECUTE_REFRESH procedure, then you will get no rows, indicating the change data has all been consumed by synchronous refresh.

### 8.4.2.5 Error Handling in Preparing Staging Logs

When a table is processed by the PREPARE_STAGING_LOG procedure, it will detect and report errors in the specification of change data that relates only to that table. For example, it will verify that keys of rows being inserted do not already exist in the base table and that keys of rows being deleted or updated do exist. However, the PREPARE_STAGING_LOG procedure cannot detect errors related to the referential integrity constraints on the table; that is, it cannot detect errors if there are inconsistencies in the specification of change data that involves more than one table. Such errors will be detected at the time of the EXECUTE_REFRESH procedure.

## 8.5 Troubleshooting Synchronous Refresh Operations

This section describes how to monitor the status of the two synchronous refresh procedures, PREPARE_REFRESH and EXECUTE_REFRESH and how to troubleshoot errors that may occur. To be successful in using synchronous refresh, you should be aware of the different types of errors that can arise and how to deal with them.

One of the most likely sources of errors is from incorrect preparation of the change data. These errors will present themselves as referential constraint violations when the EXECUTE_REFRESH procedure is run. In such cases, the status of the group is set to ABORT. It is important to learn to recognize these errors and address them.

The topics covered in this section are:


Copyright © 2001, 2025, Oracle and/or its affiliates.

- Overview of the Status of Refresh Operations
- How PREPARE_REFRESH Sets the STATUS Fields
- Examples of Preparing for Synchronous Refresh Using PREPARE_REFRESH
- How EXECUTE_REFRESH Sets the Status Fields During Synchronous Refresh
- Examples of Executing Synchronous Refresh Using EXECUTE_REFRESH
- Example of EXECUTE_REFRESH with Constraint Violations


## 8.5.1 Overview of the Status of Refresh Operations

The DBMS_SYNC_REFRESH package provides three procedures to control the refresh execution process. You initiate synchronous refresh with the PREPARE_REFRESH procedure, which plans the entire refresh operation and does the bulk of the computational work for refresh, followed by the EXECUTE_REFRESH procedure, which carries out the refresh. The third procedure provided is ABORT_REFRESH, which is used to recover from errors if either of these procedures fails.

The USER_SR_GRP_STATUS and USER_SR_OBJ_STATUS catalog views contain all the information on the status of these refresh operations for current groups:

- The USER_SR_GRP_STATUS view shows the status of the group as a whole.
  - The OPERATION field indicates the current refresh procedure run on the group: PREPARE or EXECUTE.
  - The STATUS field indicates the status of the operation - RUNNING, COMPLETE, ERROR-SOFT, ERROR-HARD, ABORT, PARTIAL. These are explained in detail later.
  - The group is identified by its group ID.
- The USER_SR_OBJ_STATUS view shows the status of each individual object.
  - The object is identified by its owner, name, and type (TABLE or MVIEW) and group ID.
  - The STATUS field, which may be NOT PROCESSED, ABORT, or COMPLETE. These are explained in detail later.

## 8.5.2 How PREPARE_REFRESH Sets the STATUS Fields

When you launch a new PREPARE_REFRESH job, the group's STATUS is set to RUNNING and the STATUS of the objects in the group is set to NOT PROCESSED. When the PREPARE_REFRESH job finishes, the status of the objects remains unchanged, but the group's status is changed to one of following three values:

- COMPLETE if the job completed successfully.
- ERROR_SOFT if the job encountered the ORA-01536: space quota exceeded for tablespace '%s' error.
- ERROR_HARD otherwise (that is, if the job encountered any error other than ORA-01536).

Some points to keep in mind when using the PREPARE_REFRESH procedure:

- The NOT PROCESSED status of the objects in the group signifies that the data of the objects has not been modified by the PREPARE_REFRESH job. The data modification will occur only in the EXECUTE_REFRESH step, at which time the status will be changed as appropriate. This is described later.


Copyright © 2001, 2025, Oracle and/or its affiliates.

- If the STATUS is ERROR_SOFT, you can fix the ORA-01536 error by increasing the space quota for the specified tablespace, and resume PREPARE_REFRESH. Alternatively, you can choose to terminate the refresh with ABORT_REFRESH.
- If the STATUS value is ERROR_HARD, then your only option is to terminate the refresh with ABORT_REFRESH.
- If the STATUS value after the PREPARE_REFRESH procedure finishes is RUNNING, then an error has occurred. Contact Oracle Support Services for assistance.
- A STATUS value of ERROR_HARD might be related to running out of resources because the PREPARE_REFRESH procedure can be resource-intensive. If you are not able to identify the problem, then contact Oracle Support Services for assistance. But if you can identify the problem and fix it, then you might be able to continue using synchronous refresh, by first running ABORT_REFRESH and then the PREPARE_REFRESH procedure.
- Remember that you can launch a new PREPARE_REFRESH job only when the previous refresh operation on the group (if any) has either completed execution successfully or has terminated.
- If the STATUS value of the PREPARE_REFRESH procedure at the end is not COMPLETE, you cannot proceed to the EXECUTE_REFRESH step. If you are unable to get PREPARE_REFRESH to work correctly, then you can proceed to the unregistration phase, and maintain the objects in the groups with other refresh methods.

## 8.5.3 Examples of Preparing for Synchronous Refresh Using PREPARE_REFRESH

This section offers examples of common cases when preparing a refresh.

**Example 8-16 PREPARE_REFRESH Succeeds with Status COMPLETE**

This example shows a PREPARE_REFRESH procedure completing successfully.

```
EXECUTE DBMS_SYNC_REFRESH.PREPARE_REFRESH( DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));

PL/SQL procedure successfully completed.

SELECT OPERATION, STATUS
FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

OPERATION  STATUS
---------  ------
PREPARE    COMPLETE
```

**Example 8-17 PREPARE_REFRESH Fails with Status ERROR_SOFT**

This example shows a PREPARE_REFRESH procedure encountering ORA-01536.

```
EXECUTE DBMS_SYNC_REFRESH.PREPARE_REFRESH( DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));
BEGIN DBMS_SYNC_REFRESH.PREPARE_REFRESH(DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1')); END;

*
ERROR at line 1:
ORA-01536: space quota exceeded for tablespace 'DUMMY_TS'
ORA-06512: at "SYS.DBMS_SYNC_REFRESH", line 63
ORA-06512: at "SYS.DBMS_SYNC_REFRESH", line 411
ORA-06512: at "SYS.DBMS_SYNC_REFRESH", line 429
ORA-06512: at line 1PL/SQL procedure successfully completed.
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
SELECT OPERATION, STATUS
FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

OPERATION  STATUS
---------  ------
PREPARE    ERROR_SOFT
```

**Example 8-18 Resume of PREPARE_REFRESH Succeeds**

This example is a continuation of Example 8-17. After the ORA-01536 error is raised, increase the tablespace for DUMMY_TS and rerun the PREPARE_REFRESH procedure, which now completes successfully. Note that the PREPARE_REFRESH procedure will resume processing from the place where it stopped. Also note the usage of the PREPARE_REFRESH procedure is no different from normal, and does not require any parameters or settings to indicate the procedure is being resumed.

```
EXECUTE DBMS_SYNC_REFRESH.PREPARE_REFRESH(DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));

PL/SQL procedure successfully completed.

SELECT OPERATION, STATUS
FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

OPERATION  STATUS
---------  ------
PREPARE    COMPLETE
```

**Example 8-19 Terminate PREPARE_REFRESH**

This example assumes the PREPARE_REFRESH procedure has failed and the STATUS value is ERROR_HARD. You then run the ABORT_REFRESH procedure to terminate the prepare job. Note that the STATUS value has changed from ERROR_HARD to ABORT at the end.

```
SELECT OPERATION, STATUS
FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

OPERATION  STATUS
---------  ------
PREPARE    ERROR_HARD

EXECUTE DBMS_SYNC_REFRESH.ABORT_REFRESH( DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));

PL/SQL procedure successfully completed.

SELECT OPERATION, STATUS
FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

OPERATION  STATUS
---------  ------
PREPARE    ABORT
```

## 8.5.4 How EXECUTE_REFRESH Sets the Status Fields During Synchronous Refresh

The EXECUTE_REFRESH procedure divides the group of objects in the sync refresh group into subgroups, each of which is refreshed atomically. The first subgroup consists of the base


Copyright © 2001, 2025, Oracle and/or its affiliates.

tables. Each materialized view in the sync refresh group is placed in a separate subgroup and refreshed atomically.

In the case of the EXECUTE_REFRESH procedure, the possible end states of the STATUS field are: COMPLETE, PARTIAL, and ABORT:

- STATUS = COMPLETE

  This state is reached if the base tables and all the materialized views refresh successfully.

- STATUS = ABORT

  This state indicates the refresh of the base tables subgroup has failed; the data in the tables and materialized views is consistent but unchanged. If this happens, then there should be an error associated with the failure. If it is a user error, such as a constraint violation, then you can fix the problem and retry the synchronous refresh operation from the beginning (that is, PREPARE_STAGING_LOG for each table in the group PREPARE_REFRESH and EXECUTE_REFRESH.). If it is not a user error, then you should contact Oracle Support Services.

- STATUS = PARTIAL

  If all the base tables refresh successfully and some, but not all, materialized views refresh successfully, then this state is reached. The data in the tables and materialized views that have refreshed successfully are consistent with one another; the other materialized views are stale and need complete refresh. If this happens, there should be an error associated with the failure. Most likely this is not a user error, but an Oracle error that you should report to Oracle Support Services. You have two choices in this state:

  – Retry execution of the EXECUTE_REFRESH procedure. In such a case, EXECUTE_REFRESH will retry the refresh of the failed materialized views with another refresh method like PCT-refresh or COMPLETE refresh. If all materialized views succeed, then the status will be set to COMPLETE. Otherwise, the status will remain at PARTIAL.

  – Invoke the ABORT_REFRESH procedure to terminate the materialized views. This will roll back changes to all materialized views and base tables. They will all have the same data as in the original state before any of the changes in the staging logs or registered partition operations has been applied to them.

In the case of errors in the EXECUTE_REFRESH procedure, the following fields in the USER_SR_GRP_STATUS view are also useful:

- NUM_MVS_COMPLETED, which contains the number of materialized views that completed the refresh operation successfully.
- NUM_MVS_ABORTED, which contains the number of materialized views that terminated.
- ERROR and ERROR_MESSAGE, which records the error encountered in the operation.

At the end of the EXECUTE_REFRESH, procedure, the statuses of the objects in the group are marked as follows in the USER_SR_OBJ_STATUS view:

- The status of an object is set to COMPLETE if the changes were applied to it successfully.
- The status of an object is set to ABORT if the changes were not applied successfully. In this case, the object will be in the same state as it was before the refresh operation. The ERROR and ERROR_MESSAGE fields record the error encountered in the operation.
- The status of an object remains NOT PROCESSED if no changes were applied to it.


Copyright © 2001, 2025, Oracle and/or its affiliates.

# 8.5.5 Examples of Executing Synchronous Refresh Using EXECUTE_REFRESH

This section provides examples of common cases when executing a refresh.

**Example 8-20 EXECUTE_REFRESH Completes Successfully**

Example 8-20 shows an EXECUTE_REFRESH procedure completing successfully.

```
EXECUTE DBMS_SYNC_REFRESH.EXECUTE_REFRESH( DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));

PL/SQL procedure successfully completed.

SELECT OPERATION, STATUS
FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

OPERATION  STATUS
---------  ------
EXECUTE    COMPLETE
```

**Example 8-21 EXECUTE_REFRESH Succeeds Partially**

Example 8-21 shows an EXECUTE_REFRESH procedure succeeding partially. In this example, the EXECUTE_REFRESH procedure fails after refreshing the base tables but before completing the refresh of all the materialized views. The resulting status of the group is PARTIAL and the QSM-03280 error message is thrown.

```
EXECUTE DBMS_SYNC_REFRESH.EXECUTE_REFRESH(DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));
BEGIN DBMS_SYNC_REFRESH.EXECUTE_REFRESH(DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1')); END;

*
ERROR at line 1:
ORA-31928: Synchronous refresh error
QSM-03280: One or more materialized views failed to refresh successfully.
ORA-06512: at "SYS.DBMS_SYNC_REFRESH", line 63
ORA-06512: at "SYS.DBMS_SYNC_REFRESH", line 411
ORA-06512: at "SYS.DBMS_SYNC_REFRESH", line 446
ORA-06512: at line 1
```

Check the status of the group itself after the EXECUTE_REFRESH.procedure. Note that the operation field is set to EXECUTE and the status is PARTIAL.

```
SELECT OPERATION, STATUS FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

OPERATION  STATUS
---------  -------------
EXECUTE    PARTIAL
```

By querying the USER_SR_GRP_STATUS view, you find the number of materialized views that have terminated is 1 and the failed materialized view is MV1.

If you examine the status of objects in the group, because STORE and TIME are unchanged, then their status is NOT PROCESSED.

```
SELECT NAME, TYPE, STATUS FROM USER_SR_OBJ_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1')
ORDER BY TYPE, NAME;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
NAME             TYPE       STATUS
---------------- ---------- ----------------
MV1              MVIEW      ABORT
MV1_HALFMONTH    MVIEW      COMPLETE
MV2              MVIEW      COMPLETE
MV2_YEAR         MVIEW      COMPLETE
FACT             TABLE      COMPLETE
STORE            TABLE      NOT PROCESSED
TIME             TABLE      NOT PROCESSED

7 rows selected.

SELECT NUM_TBLS, NUM_MVS, NUM_MVS_COMPLETED, NUM_MVS_ABORTED
FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

NUM_TBLS NUM_MVS NUM_MVS_COMPLETED NUM_MVS_ABORTED
-------- ------- ----------------- ---------------
       3       4                 3               1
```

At this point, you can attempt to run the `EXECUTE_REFRESH` procedure once more. If the retry succeeds and the failed materialized views succeed, then the group status will be set to `COMPLETE`. Otherwise, the status will remain at `PARTIAL`. This is shown in Example 8-22. You can also terminate the refresh procedure and return to the original state. This is shown in Example 8-23.

**Example 8-22 Retrying a Refresh After a PARTIAL Status**

Example 8-22 illustrates a continuation of Example 8-21. You retry the `EXECUTE_REFRESH` procedure and it succeeds:

```
EXECUTE DBMS_SYNC_REFRESH.EXECUTE_REFRESH(DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));

PL/SQL procedure successfully completed.

--Check the status of the group itself after the EXECUTE_REFRESH operation;
--note that the operation field is set to EXECUTE and status is COMPLETE.

SELECT OPERATION, STATUS
FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

OPERATION  STATUS
---------  ---------
EXECUTE    COMPLETE
```

By querying the `USER_SR_GRP_STATUS` view, you find the number of materialized views that have terminated is 0 and the status of `MV1` is `COMPLETE`. If you examine the status of objects in the group, because `STORE` and `TIME` are unchanged, then their status is `NOT PROCESSED`.

```
SELECT NAME, TYPE, STATUS FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1')
ORDER BY TYPE, NAME;

NAME             TYPE       STATUS
---------------- ---------- ----------------
MV1              MVIEW      COMPLETE
MV1_HALFMONTH    MVIEW      COMPLETE
MV2              MVIEW      COMPLETE
MV2_YEAR         MVIEW      COMPLETE
FACT             TABLE      COMPLETE
STORE            TABLE      NOT PROCESSED
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
TIME             TABLE      NOT PROCESSED

7 rows selected.

SELECT NUM_TBLS, NUM_MVS, NUM_MVS_COMPLETED, NUM_MVS_ABORTED
FROM  USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

NUM_TBLS NUM_MVS NUM_MVS_COMPLETED NUM_MVS_ABORTED
-------- ------- ----------------- ---------------
       3       4                 4               0
```

You can examine the tables and materialized views to verify that the changes in the change data have been applied to them correctly, and the materialized views and tables are consistent with one another.

**Example 8-23 Terminating a Refresh with a PARTIAL Status**

Example 8-23 illustrates terminating a refresh procedure that is in a `PARTIAL` state.

```
EXECUTE DBMS_SYNC_REFRESH.ABORT_REFRESH(DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));

PL/SQL procedure successfully completed.
```

Check the status of the group itself after the `ABORT_REFRESH` procedure; note that the operation field is set to `EXECUTE` and status is `ABORT`.

```
SELECT OPERATION, STATUS FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

OPERATION  STATUS
---------- -------
EXECUTE    ABORT
```

By querying the `USER_SR_GRP_STATUS` view, you see that all the materialized views have terminated, and the fact table as well. Check the status of objects in the group; because `STORE` and `TIME` are unchanged, their status is `NOT PROCESSED`.

```
SELECT NAME, TYPE, STATUS FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1')
ORDER BY TYPE, NAME;

NAME             TYPE       STATUS
---------------- ---------- ----------------
MV1              MVIEW      ABORT
MV1_HALFMONTH    MVIEW      ABORT
MV2              MVIEW      ABORT
MV2_YEAR         MVIEW      ABORT
FACT             TABLE      ABORT
STORE            TABLE      NOT PROCESSED
TIME             TABLE      NOT PROCESSED

7 rows selected.

SELECT NUM_TBLS, NUM_MVS, NUM_MVS_COMPLETED, NUM_MVS_ABORTED
FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

NUM_TBLS NUM_MVS NUM_MVS_COMPLETED NUM_MVS_ABORTED
-------- ------- ----------------- ---------------
       3       4                 0               4
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

You can examine the tables and materialized views to verify that they are all in the original state and no changes from the change data have been applied to them.

## 8.5.6 Example of EXECUTE_REFRESH with Constraint Violations

In the synchronous refresh method, change data is loaded into the tables and materialized views at the same time to keep them synchronized. In the other refresh methods, change data is loaded into tables first, and any constraints that are enabled are checked at that time. In the synchronous refresh method, the outside table is prepared using trusted data from the user, and constraint validation is turned off to save execution time. The following example shows a constraint violation that is caught by the EXECUTE_REFRESH procedure. In such cases, the final status of the EXECUTE_REFRESH procedure will be ABORT. You will have to identify and fix the problem in the change data and begin the synchronous refresh phase all over.

**Example 8-24 Child Key Constraint Violation**

In Example 8-24, assume the same tables as in the file syncref_run.sql in the rdbms/demo directory are used and populated with the same data. In particular, the table STORE has four rows with the primary key STORE_KEY having the values 1 through 4, and the FACT table has rows referencing all four stores, including store 3.

To demonstrate a parent-key constraint violation, populate the staging log of STORE with the delete of the row having the STORE_KEY of 3. There are no other changes to the other tables. When the EXECUTE_REFRESH procedure runs, it fails with the ORA-02292 error as shown.

```
INSERT INTO st_store (dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE)
VALUES ('D',   3, 3, 'Store 3', '03060');

-- Prepare the staging logs
EXECUTE DBMS_SYNC_REFRESH.PREPARE_STAGING_LOG('syncref_user', 'fact');
EXECUTE DBMS_SYNC_REFRESH.PREPARE_STAGING_LOG('syncref_user', 'time');
EXECUTE DBMS_SYNC_REFRESH.PREPARE_STAGING_LOG('syncref_user', 'store');

-- Prepare the refresh
EXECUTE DBMS_SYNC_REFRESH.PREPARE_REFRESH(DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));

-- Execute the refresh
EXECUTE DBMS_SYNC_REFRESH.EXECUTE_REFRESH( -
           DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));
BEGIN DBMS_SYNC_REFRESH.EXECUTE_REFRESH(DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1')); END;

*
ERROR at line 1:
ORA-02292: integrity constraint (SYNCREF_USER.SYS_C0031765) violated - child record found
ORA-06512: at line 1
ORA-06512: at "SYS.DBMS_SYNC_REFRESH", line 63
ORA-06512: at "SYS.DBMS_SYNC_REFRESH", line 411
ORA-06512: at "SYS.DBMS_SYNC_REFRESH", line 446
ORA-06512: at line 1
```

Examine the status of the group itself after the EXECUTE_REFRESH procedure. Note that the operation field is set to EXECUTE and the status is ABORT.

```
SELECT OPERATION, STATUS
FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

OPERATION  STATUS
---------- --------------
EXECUTE    ABORT
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

If you check the contents of the base tables and of MV1, then you will find there is no change, and they all have the original values.

# 8.6 Performing Synchronous Refresh Eligibility Analysis

The CAN_SYNCREF_TABLE function tells you whether a table and its dependent materialized views are eligible for synchronous refresh. It provides an explanation of its analysis. If the table and views are not eligible, you can examine the reasons and take appropriate action if possible. To be eligible for synchronous refresh, a table must satisfy the various criteria described earlier.

You can invoke CAN_SYNCREF_TABLE function in two ways:

- Use a table to store the output of the CAN_SYNCREF_TABLE function

  The following shows the basic syntax for using an output table:

```
can_syncref_table(schema_name   IN VARCHAR2,
                  table_name    IN VARCHAR2,
                  statement_id  IN VARCHAR2)
```

- Use a VARRAY to store the output of the CAN_SYNCREF_TABLE function

  To direct the output of the CAN_SYNCREF_TABLE function to a VARRAY instead of a table, call the procedure as follows:

```
can_syncref_table(schema_name   IN VARCHAR2,
                  table_name    IN VARCHAR2,
                  output_array  IN OUT Sys.CanSyncRefTypeArray)
```

You can create an output table called SYNCREF_TABLE by executing the utlcsrt.sql script.

**Table 8-1 CAN_SYNCREF_TABLE**

| Parameter | Description |
|---|---|
| schema_name | Name of the schema of the base table. |
| base_table_name | Name of the base table. |
| statement_id | A string (VARCHAR2(30) to identify the rows pertaining to a call of the CAN_SYNCREF_TABLE function when the output is directed to a table named SYNCREF_TABLE in the user's schema. |
| output_array | The output array into which CAN_SYNCREF_TABLE records the information on the eligibility of the base table and its dependent materialized views for synchronous refresh. |

> **Note**
>
> Only one statement_id or output_array parameter need be provided to the CAN_SYNCREF_TABLE function.


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 8.6.1 Using SYNCREF_TABLE to Store the Results of Synchronous Refresh Eligibility Analysis

The output of the CAN_SYNCREF_TABLE function can be directed to a table named SYNCREF_TABLE. You are responsible for creating SYNCREF_TABLE; it can be dropped when it is no longer needed. The format of SYNCREF_TABLE is as follows:

```
CREATE TABLE SYNCREF_TABLE(
          statement_id             VARCHAR2(30),
          schema_name              VARCHAR2(30),
          table_name               VARCHAR2(30),
          mv_schema_name           VARCHAR2(30),
          mv_name                  VARCHAR2(30),
          eligible                 VARCHAR2(1),   -- 'Y' , 'N'
          seq_num                  NUMBER,
          msg_number               NUMBER,
          message                  VARCHAR2(4000)
);
```

You must provide a different statement_id parameter for each invocation of this procedure on the same table. If not, an error will be thrown. The statement_id, schema_name, and table_name fields identify the results for a given table and statement_id.

Each row contains information on the eligibility of either the table or its dependent materialized view. The CAN_SYNCREF_TABLE function guarantees that each row has values for both mv_schema_name and mv_name that are either NULL or non-NULL. These rows have the following semantics:

- If the mv_schema_name value is NULL and mv_name is NULL, then the ELIGIBLE field describes whether the table is eligible for synchronous refresh; if the table is not eligible, the MSG_NUMBER and MESSAGE fields provide the reason for this.
- If the mv_schema_name value is NOT NULL and mv_name is NOT NULL, then the ELIGIBLE field describes whether the materialized view is eligible for synchronous refresh; if the materialized view is not eligible, the MSG_NUMBER and MESSAGE fields provide the reason for this.

You must provide a different statement_id parameter for each invocation of this procedure on the same table, or else an error will be thrown. The statement_id, schema_name, and table_name fields identify the results for a given table and statement_id.

## 8.6.2 Using a VARRAY to Store the Results of Synchronous Refresh Eligibility Analysis

You can save the output of the CAN_SYNCREF_TABLE function in a PL/SQL VARRAY. The elements of this array are of type CanSyncRefMessage, which is predefined in the SYS schema as shown in the following example:

```
TYPE CanSyncRefMessage IS OBJECT (
          schema_name         VARCHAR2(30),
          table_name          VARCHAR2(30),
          mv_schema_name      VARCHAR2(30),
          mv_name             VARCHAR2(30),
          eligible            VARCHAR2(1),    -- 'Y' , 'N'
          seq_num             NUMBER,
          msg_number          NUMBER,
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
        message             VARCHAR2(4000)
);
```

The array type, CanSyncRefArrayType, which is a VARRAY of CanSyncRefMessage objects, is predefined in the SYS schema as follows:

```
TYPE  CanSyncRefArrayType AS VARRAY(256) OF CanSyncRefMessage;
```

Each CanSyncRefMessage record provides a message concerning the eligibility of the base table or a dependent materialized view for synchronous refresh. The semantics of the fields is the same as that of the corresponding fields in SYNCREF_TABLE. However, SYNCREF_TABLE has a statement_id field that is absent in CanSyncRefMessage because no statement_id is supplied (because it is not required) when the CAN_SYNCREF_TABLE procedure is called with a VARRAY parameter.

The default size limit for CanSyncRefArrayType is 256 elements. If you need more than 256 elements, then connect as SYS and redefine CanSyncRefArray. The following commands, when connected as the SYS user, redefine CanSyncRefArray and set the limit to 2048 elements:

```
CREATE OR REPLACE TYPE CanSyncRefArrayType AS VARRAY(2048) OF SYS.CanSyncRefMessage;
/
GRANT EXECUTE ON SYS.CanSyncRefMessage TO PUBLIC;

CREATE OR REPLACE PUBLIC SYNONYM CanSyncRefMessage FOR SYS.CanSyncRefMessage;
/
GRANT EXECUTE ON SYS.CanSyncRefArrayType TO PUBLIC;

CREATE OR REPLACE PUBLIC SYNONYM CanSyncRefArrayType FOR SYS.CanSyncRefArrayType;
/
```

### 8.6.3 Demo Scripts

The synchronous refresh demo scripts in the rdbms/demo directory contain examples of the most common scenarios of the various synchronous refresh operations, including CAN_SYNCREF_API. The main script is syncref_run.sql and its log is syncref_run.log. The file syncref_cst.sql defines two procedures DO_CST and DO_CST_ARR, which simplify the usage of the CAN_SYNCREF_TABLE function and display the information on the screen in a convenient format. This format is documented in the syncref_cst.sql file.

## 8.7 Overview of Synchronous Refresh Security Considerations

The execute privilege on the DBMS_SYNC_REFRESH package is granted to PUBLIC, so all users can execute the procedures in that package to perform synchronous refresh on objects owned by them. The database administrator can perform synchronous refresh operation on all tables and materialized views in the database.

In general, if a user without the DBA privilege wants to use synchronous refresh on another user's table, they must have complete privileges to read from and write to that table; that is, the user must have the SELECT, INSERT, UPDATE, and DELETE privileges on that table or materialized view. The user can have the READ privilege instead of the SELECT privilege. A couple of exceptions occur in the following:

- PURGE_REFRESH_STATS and ALTER_REFRESH_STATS_RETENTION functions

  These two functions implement the purge policy and can be used to change the default retention period. These functions can be executed only by the database administrator.

- The CAN_SYNCREF_TABLE function


Copyright © 2001, 2025, Oracle and/or its affiliates.

This is an advisory function that examines the eligibility for synchronous refresh of all the materialized views associated with a specified table. Hence, this function requires the `READ` or `SELECT` privilege on all materialized views associated with the specified table.


Copyright © 2001, 2025, Oracle and/or its affiliates.

9

# Monitoring Materialized View Refresh Operations

This chapter describes how to use refresh statistics to monitor the performance of materialized view refresh operations.

This chapter contains the following topics:


- About Materialized View Refresh Statistics
- Overview of Managing Materialized View Refresh Statistics
- About Data Dictionary Views that Store Materialized View Refresh Statistics
- Collecting Materialized View Refresh Statistics
- Retaining Materialized View Refresh Statistics
- Viewing Materialized View Refresh Statistics Settings
- Purging Materialized View Refresh Statistics
- Viewing Materialized View Refresh Statistics
- Analyzing Materialized View Refresh Performance Using Refresh Statistics


## 9.1 About Materialized View Refresh Statistics

Oracle AI Database collects and stores statistics about materialized view refresh operations. These statistics are accessible using data dictionary views.

Statistics for both current and historical materialized view refresh operations are stored in the database. Historical materialized view refresh statistics enable you to understand and analyze materialized view refresh performance over time in your database. Refresh statistics can be collected at varying levels of granularity.

Maintaining materialized view refresh statistics provides the following:

- Reporting capabilities for materialized view refresh operations
  - Display both current and historical statistics for materialized view refresh operations
  - Display statistics on actual refresh execution times
  - Track the performance of materialized view refresh over time using statistics on actual refresh execution times
- Diagnostic capabilities for materialized view refresh performance

  Detailed current and historical statistics can be used to quickly analyze the performance of materialized view refresh operations. For example, if a materialized view takes a long time to refresh, you can use refresh statistics to determine if the slowdown is due to increased system load or vastly varying change data.



Copyright © 2001, 2025, Oracle and/or its affiliates.

# 9.2 Overview of Managing Materialized View Refresh Statistics

Oracle AI Database manages the collection and retention of materialized view refresh statistics based on the defined database settings. By default, the database collects and stores basic statistics about materialized view refresh operations for the entire database.

Managing materialized view refresh statistics comprises of the defining policies that control the following:

- Level of details for materialized view refresh statistics
- Retention period of materialized view refresh statistics

Use the following techniques to define policies that manage materialized view refresh statistics:

- Define default settings that are applicable to the entire database

  The `DBMS_MVIEW_STATS.SET_SYSTEM_DEFAULT` procedure defines default settings that manage the collection and retention of materialized view refresh statistics for the entire database.
- Define collection and retention policies for individual materialized views

> **Note**
>
> The `DBMS_MVIEW_STATS.SET_MVREF_STATS_PARAMS` procedure provides more fine-grained control over materialized view refresh statistics by managing the collection and retention of statistics at the level in individual materialized views. Settings made at the materialized view level override the database-level settings.
>
> However, the view `DBA_MVREF_RUN_STAT` provides a record for each refresh group and does not provide details for individual materialized views.

> **See Also**
>
> - Collecting Materialized View Refresh Statistics
> - Retaining Materialized View Refresh Statistics

# 9.3 About Data Dictionary Views that Store Materialized View Refresh Statistics

Oracle AI Database stores materialized view refresh statistics in the data dictionary. Setting the collection level for materialized view refresh controls the detail level of refresh statistics collected.

Each materialized view refresh operation is identified using a unique refresh ID. A single refresh operation could refresh multiple materialized views. For example, when the `REFRESH_DEPENDENT` procedure is used to refresh a single materialized view, all materialized views that are dependent on the specified materialized view are also refreshed as part of the same refresh operation. Thus, all the materialized views refreshed as part of this operation will have the same refresh ID.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Table 9-1 Data Dictionary Views that Store Materialized View Refresh Statistics**

| View Name | Description |
|---|---|
| DBA_MVREF_STATS | Stores basic statistics for a materialized view refresh such as the refresh ID and basic timing statistics for the refresh operation.<br>This view contains the following information about each materialized view for which refresh statistics are collected:<br>• name of the materialized view<br>• refresh method used<br>• number of rows in the materialized view at the beginning and end of the refresh operation<br>• number of steps used to refresh the materialized view<br>**Note**<br>This is view populated for fast refresh of materialized views with aggregates or joins only. It is not populated for other types of materialized view refreshes. |
| DBA_MVREF_RUN_STATS | Stores detailed information about each materialized view refresh operation including the following:<br>• parameters specified when running the refresh operation such as list of materialized views, refresh method, purge option, and so on.<br>• number of materialized views refreshed in the refresh operation.<br>• detailed timing statistics for the refresh operation including start time, end time, and elapsed time. |
| DBA_MVREF_CHANGE_STATS | Contains change data load information for the base tables associated with a materialized view refresh operation.<br>The details include base table names, materialized view names, number of rows inserted, number of rows updated, number of rows deleted, number of direct-load inserts, PMOPs details, and number of rows at the beginning of the refresh operation. |
| DBA_MVREF_STMT_STATS | Contains information related to each refresh statement that is part of a single materialized view refresh operation.<br>This includes information such as materialized view name, refresh ID, the refresh statement, SQLID of the refresh statement, and execution plan of the statement. |

**See Also**

*Oracle AI Database Reference*


Copyright © 2001, 2025, Oracle and/or its affiliates.

# 9.4 Collecting Materialized View Refresh Statistics

Oracle AI Database collects basic statistics about materialized view refresh operations. These statistics are stored in the data dictionary and can be used to analyze the performance of materialized view refresh operations.

**See Also**

- About Collecting Materialized View Refresh Statistics
- Specifying Default Settings for Collecting Materialized View Refresh Statistics
- Modifying the Collection Level for Materialized View Refresh Statistics

## 9.4.1 About Collecting Materialized View Refresh Statistics

By default, Oracle AI Database collects basic refresh statistics for all materialized views refresh operations.

Oracle AI Database enables you to control the granularity and level at which materialized view refresh statistics are collected. Statistics can be collected for all materialized views in the database or for a specific set of materialized views. If you are interested in monitoring only some materialized views in the database, then you can collect statistics at the materialized view level. Collecting refresh statistics for a selected set of materialized views is useful because refresh patterns of materialized views can vary widely.

The collection level defines the amount of statistics that the database collects for materialized view refresh operations. You can either collect basic statistics or more detailed information such as the parameters used and the SQL statements run during the materialized view refresh operation.

Use the procedures in the `DBMS_MVIEW_STATS` package to set the `COLLECTION_LEVEL` parameter, which specifies the collection level for materialized view refresh statistics. The values that can be set for the `COLLECTION_LEVEL` parameter are:

- NONE

  No statistics are collected for materialized view refresh operations.

- TYPICAL

  Only basic refresh statistics are collected for materialized view refresh operations. This is the default setting.

- ADVANCED

  Detailed statistics, including the parameters used in the refresh operation and the SQL statements that are run, are collected for materialized view refresh operations.


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 9.4.2 Specifying Default Settings for Collecting Materialized View Refresh Statistics

The `DBMS_MVIEW_STATS.SET_SYSTEM_DEFAULT` procedure enables you to set defaults for managing the collection of materialized view refresh statistics at the database level.

You can override the system defaults by specifying different settings at the individual materialized view level. Materialized views for which the default settings are not overridden will use the system default settings.

By default, Oracle AI Database does not collect basic statistics about materialized view refresh operations. You can change the default setting or disable statistics collection by modifying the statistics collection level.

To set the default collection level for materialized view refresh statistics at the database level:

- Run the `DBMS_MVIEW_STATS.SET_SYSTEM_DEFAULT` procedure and set the `COLLECTION_LEVEL` parameter.

**Example 9-1 Setting Materialized View Refresh Statistics Collection Level for the Database**

This example sets the default collection level for materialized view refresh statistics to ADVANCED indicating that detailed statistics about materialized view refresh operations will be collected and stored.

```
DBMS_MVIEW_STATS.SET_SYSTEM_DEFAULT ('COLLECTION_LEVEL','ADVANCED');
```

**Example 9-2 Disabling Statistics Collection for Materialized View Refresh**

This example sets the default collection level for materialized view refresh statistics to NONE thereby disabling statistics collection.

```
DBMS_MVIEW_STATS.SET_SYSTEM_DEFAULT ('COLLECTION_LEVEL','NONE');
```

> **See Also**
>
> *Oracle AI Database PL/SQL Packages and Types Reference*

## 9.4.3 Modifying the Collection Level for Materialized View Refresh Statistics

You can modify the settings that manage the collection of materialized view refresh statistics by using the `DBMS_MVIEW_STATS.SET_MVREF_STATS_PARAMS` procedure.

You can modify the statistics collection behavior either for the entire database or for one or more materialized views. The new collection settings override the default settings made at the database level or previous settings made for the specified materialized views. For example, the system default for `COLLECTION_LEVEL` is set to TYPICAL for the database. You then use the `DBMS_MVIEW_STATS.SET_MVREF_STATS_PARAMS` procedure to modify the collection level for the materialized views `MV1` and `MV2` to ADVANCED. The remaining materialized views in the database will continue to use the TYPICAL collection level.


Copyright © 2001, 2025, Oracle and/or its affiliates.

To modify the collection level for materialized view refresh statistics, either at the database level or materialized view level:

- Run the `DBMS_MVIEW_STATS.SET_MVREF_STATS_PARAMS` procedure and set the `COLLECTION_LEVEL` parameter to the required value

**Example 9-3 Setting the Materialized View Statistics Collection Level for the Entire Database**

The following example modifies the collection level for materialized view refresh statistics at the database level to TYPICAL. Specifying `NULL` instead of one or more materialized view names indicates that this setting is for the entire database.

```
DBMS_MVIEW_STATS.SET_MVREF_STATS_PARAMS (NULL, 'TYPICAL');
```

**Example 9-4 Setting the Materialized View Statistics Collection Level for Multiple Materialized Views**

This example sets the collection level for the materialized views `SALES_2013_MV` and `SALES_2014_MV` in the `SH` schema to ADVANCED. The retention period is set to 60 days. This setting overrides any default settings that may have been specified at the database level.

```
DBMS_MVIEW_STATS.SET_MVREF_STATS_PARAMS ('SH.SALES_2013_MV,
SH.SALES_2014_MV','ADVANCED',60);
```

> **See Also**
>
> *Oracle AI Database PL/SQL Packages and Types Reference*

# 9.5 Retaining Materialized View Refresh Statistics

Oracle AI Database stores the collected materialized view refresh statistics for a period of time specified by the retention period.

> **See Also**
>
> - About Retaining Materialized View Refresh Statistics
> - Specifying the Default Retention Period for Materialized View Refresh Statistics
> - Modifying the Retention Period for Materialized View Refresh Statistics

## 9.5.1 About Retaining Materialized View Refresh Statistics

The retention period defines the duration, in days, for which materialized view refresh statistics are stored in the data dictionary. Collected statistics are automatically purged after the retention period is reached.

The retention period for materialized view refresh statistics can be set either at the database level or the materialized view level. The `RETENTION_PERIOD` parameter in `DBMS_MVIEW_STATS.SET_SYSTEM_DEFAULT` or `DBMS_MVIEW_STATS.SET_MVREF_STATS_PARAMS`


Copyright © 2001, 2025, Oracle and/or its affiliates.

enables you to specify the duration for which materialized view refresh statistics must be retained in the data dictionary.

## 9.5.2 Specifying the Default Retention Period for Materialized View Refresh Statistics

The `DBMS_MVIEW_STATS.SET_SYSTEM_DEFAULT` procedure sets defaults for managing the retention of materialized view refresh statistics at the database level.

By default, Oracle AI Database retains materialized view refresh statistics for 365 days from the date of collection. After the retention period is reached, the statistics are purged from the data dictionary. You can override the system default setting by specifying different settings at the individual materialized view level. Materialized views for which the default settings are not overridden will continue to use the system default settings.

You can specify that refresh statistics must never be purged from the database by setting the retention period to -1.

To specify a new default retention period for the entire database:

- Set the `RETENTION_PERIOD` parameter of the `DBMS_MVIEW_STATS.SET_SYSTEM_DEFAULT` procedure to the required number of days

**Example 9-5 Setting the Retention Period for Materialized View Refresh Statistics**

This example sets the default retention period for materialized view refresh statistics for the entire database to 60 days.

```
DBMS_MVIEW_STATS.SET_SYSTEM_DEFAULT ('RETENTION_PERIOD',60);
```

**Example 9-6 Preventing the Purging of Materialized View Refresh Statistics**

This example sets the retention period for materialized view refresh statistics to -1 thereby ensuring that refresh statistics are not automatically purged when the default retention period is reached. When you use this setting, refresh statistics will need to be explicitly purged from the data dictionary using the `DBMS_MVIEW_STATS.PURGE_REFRESH_STATS` procedure.

```
DBMS_MVIEW_STATS.SET_SYSTEM_DEFAULT ('RETENTION_PERIOD',–1);
```

> **See Also**
>
> *Oracle AI Database PL/SQL Packages and Types Reference*

## 9.5.3 Modifying the Retention Period for Materialized View Refresh Statistics

The `DBMS_MVIEW_STATS.SET_MVREF_STATS_PARAMS` procedure enables you to modify the retention period set for materialized view refresh statistics.

You can modify the retention period either for the entire database or for one or more materialized views. When you modify the retention period only for specific materialized views, the remaining materialized views in the database continue to use their existing retention period.


Copyright © 2001, 2025, Oracle and/or its affiliates.

Suppose that your system default setting is to collect basic materialized view refresh statistics and retain them for 60 days. However, for a particular set of materialized views, you want to collect detailed statistics and retain these statistics for 45 days. In this case, for the specific set of materialized views, you set `COLLECTION_LEVEL` to ADVANCED and `RETENTION_PERIOD` to 45.

To modify the retention period for materialized view refresh statistics either at the database level to materialized view level:

- Run the `DBMS_MVIEW_STATS.SET_MVREF_STATS_PARAMS` procedure and set the `RETENTION_PERIOD` parameter to the required value

**Example 9-7 Using Default Materialized View Refresh Statistics Settings for Retention Period**

This example sets the collection level for the materialized view `SALES_MV` in the `SH` schema to TYPICAL. Since NULL is used for the retention period, the system-wide default setting for retention period is used for this materialized view.

```
DBMS_MVIEW_STATS.SET_MVREF_STATS_PARAMS ('SH.SALES_MV','TYPICAL',NULL);
```

**Example 9-8 Setting the Retention Period for a Materialized View**

This example sets the collection level for the `SH.SALES_MV` to ADVANCED and the retention period to 45 days. This overrides the existing retention period set for this materialized view.

```
DBMS_MVIEW_STATS.SET_MVREF_STATS_PARAMS ('SH.SALES_MV','ADVANCED',45);
```

> **See Also**
>
> *Oracle AI Database PL/SQL Packages and Types Reference*

# 9.6 Viewing Materialized View Refresh Statistics Settings

Data dictionary views store both the default settings and materialized view-specific settings that manage materialized view refresh statistics.

To view the database-level default settings for collecting and retaining materialized view refresh statistics:

- Query the `parameter_name` and `value` columns in the `DBA_MVREF_STATS_SYS_DEFAULTS` view

To view the collection and retention settings for refresh statistics of one or more materialized views:

- Query the `parameter_name` and `value` columns in the `DBA_MVREF_STATS_PARAMS` view by filtering data using the `mv_owner` and `mv_name` columns


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Example 9-9 Displaying the Database-level Default Settings for Managing Materialized View Refresh Statistics**

The following query displays the database level default settings for managing materialized view refresh statistics:

```
SELECT parameter_name, value from DBA_MVREF_STATS_SYS_DEFAULTS;

PARAMETER_NAME      VALUE
____________________ ________
COLLECTION_LEVEL    NONE
RETENTION_PERIOD    31
```

**Example 9-10 Displaying the Refresh Statistics Settings for a Set of Materialized Views**

The following query displays the refresh statistics settings for all the materialized view owned by the SH schema:

```
SELECT mv_name,collection_level,retention_period
FROM DBA_MVREF_STATS_PARAMS
WHERE mv_owner = 'SH';
MV_NAME                          COLLECTION_LEVEL       RETENTION_PERIOD
_________________________________ ____________________ ____________________
SUM_SALES_PSCAT_WEEK_MV          NONE                                   31
SUM_SALES_PROD_WEEK_MV           NONE                                   31
SUM_SALES_PSCAT_MONTH_CITY_MV    NONE                                   31
CAL_MONTH_SALES_MV               NONE                                   31
JOIN_SALES_TIME_PRODUCT_MV       NONE                                   31
FWEEK_PSCAT_SALES_MV             NONE                                   31
JOIN_SALES_TIME_PRODUCT_OJ_MV    NONE                                   31
```

# 9.7 Purging Materialized View Refresh Statistics

The DBMS_MVIEW_STATS.PURGE_REFRESH_STATS procedure enables you to explicitly purge materialized view refresh statistics that are older than a specified period from the data dictionary.

By default, materialized view refresh statistics are removed from the data dictionary after the specified retention period. Depending on your settings, the purging may be performed for the entire database or for a set of specified materialized views. You can use the DBMS_MVIEW_STATS.PURGE_REFRESH_STATS procedure to explicitly purge refresh statistics that are older than a specified time without altering the set retention period. Explicit purging of refresh statistics overrides the current setting for retention period but does not alter the setting.

To purge materialized view refresh statistics stored in the database:

- Run the DBMS_MVIEW_STATS.PURGE_REFRESH_STATS procedure.

  Specify the materialized views for which statistics must be purged and the duration beyond which statistics must be purged.

**Example 9-11 Purging Refresh Statistics for a Materialized View**

Assume that the retention period for refresh statistics of the materialized view SALES_MV is 60 days. At any given time, the refresh statistics for the previous 60 days are available. However, because of space constraints, you want to purge the statistics for the last 30 days. Use the DBMS_MVIEW_STATS.PURGE_REFRESH_STATS procedure to do this.


Copyright © 2001, 2025, Oracle and/or its affiliates.

Note that the retention period set for SALES_MV remains unaltered. The purge is a one-time operation.

```
DBMS_MVIEW_STATS.PURGE_REFRESH_STATS ('SH.SALES_MV',30);
```

**Example 9-12 Purging Refresh Statistics for All Materialized Views**

This example purges materialized view refresh statistics that are older than 20 days for all materialized views in the database. Specifying NULL instead of one or more materialized views indicates that this setting is for the entire database.

```
DBMS_MVIEW_STATS.PURGE_REFRESH_STATS (NULL,20);
```

**See Also**

*Oracle AI Database PL/SQL Packages and Types Reference*

# 9.8 Viewing Materialized View Refresh Statistics

You can view both current and historical statistics for materialized view refresh operations by querying the data dictionary views that store refresh statistics.

Depending on the collection level setting, materialized view refresh statistics are stored in one or more of the following views: DBA_MVREFS_STATS, DBA_MVREF_RUN_STATS, DBA_MVREF_CHANGE_STATS, and DBA_MVREF_STMT_STATS. There are corresponding USER_ versions for all these views. The views contain a REFRESH_ID column that can be used to join one or more views, when required.

**See Also**

- Viewing Basic Refresh Statistics for a Materialized View
- Viewing Detailed Statistics for Each Materialized View Refresh Operation
- Viewing Change Data Statistics During Materialized View Refresh Operations
- Viewing the SQL Statements Associated with A Materialized View Refresh Operation

## 9.8.1 Viewing Basic Refresh Statistics for a Materialized View

Use the DBA_MVREF_STATS view to display basic statistics about materialized view refresh operations.

Each materialized view refresh operation is identified using a unique refresh ID. The DBA_MVREF_STATS view stores the refresh ID, refresh method, names of materialized views refreshed, basic execution times, and the number of steps in the refresh operation.

To view basic refresh statistics for materialized view refresh operations:

- Query the DBA_MVREF_STATS view with list of required columns and use conditions to filter the required data


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Example 9-13 Displaying Basic Statistics for a Materialized View Refresh Operation**

The following query displays some refresh statistics for refresh operations on the SH.NEW_SALES_RTMV materialized view. Information includes the refresh method, refresh time, number of rows in the materialized view at the start of the refresh operation, and number of rows at the end of the refresh operation.

```
SELECT refresh_id, refresh_method, elapsed_time, initial_num_rows,
final_num_rows
FROM dba_mvref_stats
WHERE mv_name = 'NEW_SALES_RTMV' and mv_owner = 'SH';

REFRESH_ID    REFRESH_METHOD    ELAPSED_TIME    INITIAL_NUM_ROWS
FINAL_NUM_ROWS
----------    --------------    -------------   ----------------
----------------
49            FAST              0               766                788
61            FAST              1               788                788
81            FAST              1               788                798

3 rows selected.
```

**Example 9-14 Displaying Materialized Views Based on their Refresh Times**

The following example displays the names of materialized views whose refresh operations took more than 10 minutes. Since elapsed_time is specified in seconds, we use 600 in the query.

```
SELECT mv_owner, mv_name, refresh_method
FROM dba_mvref_stats
WHERE elapsed_time > 600;
```

## 9.8.2 Viewing Detailed Statistics for Each Materialized View Refresh Operation

The DBA_MVREF_RUN_STATS view stores detailed statistics about materialized view refresh operation. When a refresh operation affects multiple materialized views, detailed statistics are available for all affected materialized views.

Materialized views can be refreshed using one of the following procedures in the DBMS_MVIEW package: REFRESH, REFRESH_DEPENDENT, or REFRESH_ALL. Each procedure contains different parameters that specify how the refresh must be performed. The DBA_MVREF_RUN_STATS view contains information about the parameters specified for the refresh operation, the number of materialized views refreshed, execution times, and log purge time.

To view detailed refresh statistics for materialized view refresh operations:

- Query the DBA_MVREF_RUN_STATS view with the list of required columns and use conditions to filter the required data


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Example 9-15 Listing All Materialized Views Refreshed in a Single Refresh Operation**

The following example displays the materialized views and refresh times for materialized views that were refreshed as part of the specified refresh ID.

```
SELECT mviews, elapsed_time, complete_stats_available
FROM dba_mvref_run_stats
WHERE refresh_id = 100;

MVIEWS              ELAPSED_TIME      COMPLETE_STATS_AVAIALBE
--------            ------------      -------------------------
"SH"."SALES_RTMV"   1                 Y
```

**Example 9-16 Viewing the Parameters Specified During a Materialized View Refresh Operation**

The following example displays the list of refreshed materialized views and some of the parameters specified during the refresh operation for refresh ID 81.

```
SELECT mviews, refresh_after_errors, purge_option, parallelism, nested
FROM dba_mvref_run_stats
WHERE run_owner = 'SH' and refresh_id=81;

MVIEWS                  R     PURGE_OPTION    PARALLELISM   NESTED
------                  -     ------------    ------------  -------
"SH"."SALES_RTMV"    N    1        0    N
```

**Example 9-17 Displaying Detailed Statistics for a Materialized View Refresh Operation**

The following example displays detailed statistics for the refresh operation with refresh ID 156. The details include the number of materialized views refreshed, the owner and names of materialized views, and the time taken for the refresh.

```
SELECT num_mvs, mv_owner, mv_name, r.elapsed_time
FROM dba_mvref_stats s, dba_mvref_run_stats r
WHERE s.refresh_id = r.refresh_id and refresh_id = 156;

NUM_MVS     MV_OWNER    MV_NAME            ELAPSED_TIME
--------    --------    --------        -----------
1           SH          SALES_RTMV        5
```

> **See Also**
>
> *Oracle AI Database Reference*

## 9.8.3 Viewing Change Data Statistics During Materialized View Refresh Operations

The `DBA_MVREF_CHANGE_STATS` view stores detailed change data statistics for materialized view refresh operations. This includes the base tables that were refreshed, the number of rows


Copyright © 2001, 2025, Oracle and/or its affiliates.

inserted, number of rows updated, number of rows deleted, and partition maintenance operations (PMOPs) details.

You can join the DBA_MVREF_CHANGE_STATS view with other views that contain materialized view refresh statistics to provide more complete statistics.

To view detailed change data statistics for materialized view refresh operations:

- Query the DBA_MVREF_CHANGE_STATS view with the list of required columns and use conditions to filter the required data

**Example 9-18 Determining if a Refresh Operation Resulted in PMOPs**

The following example displays the base table names and PMOP details for the refresh operation with refresh ID 1876. The query output contains one record for each base table of the materialized view.

```
SELECT tbl_name, mv_name, pmops_occurred, pmop_details
FROM dba_mvref_change_stats
WHERE refresh_id =1876;

TBL_NAME      MV_NAME     PMOPS_OCCURRED    PMOP_DETAILS
---------     --------    --------------    ------------
MY_SALES      SALES_RTMV    N
```

**Example 9-19 Displaying the Number of Rows Modified During a Refresh Operation**

This example displays the following details about each base table in a refresh operation on the SH.MY_SALES materialized view: number of rows in the tables, number of rows inserted, number of rows updates, number of rows deleted, number of direct load inserts, and details of PMOP operations.

```
SELECT tbl_name, num_rows, num_rows_ins, num_rows_upd, num_rows_del,
num_rows_dl_ins, pmops_occurred, pmop_details
FROM dba_mvref_change_stats
WHERE mv_name = 'MY_SALES' and mv_owner = 'SH';
```

> **See Also**
>
> *Oracle AI Database Reference*

## 9.8.4 Viewing the SQL Statements Associated with A Materialized View Refresh Operation

Query the DBA_MVREF_STMT_STATS view to display information about all the SQL statements used in a materialized view refresh operation.

Each refresh operation can consist of multiple steps, each of which is performed using a SQL statement. For each step in a refresh operation, you can view the step number and the SQL statement.

To view the SQL statements associated with materialized view refresh operations:


Copyright © 2001, 2025, Oracle and/or its affiliates.

- Query the DBA_MVREF_STMT_STATS view with the list of required columns and use conditions to filter the required data

**Example 9-20 Displaying SQL Statements for Each Step in a Refresh Operation**

The following example displays the materialized view names, SQL statements used to refresh the materialized view, and execution time for the materialized view refresh operation with refresh ID is 1278.

```
SELECT mv_name, step, stmt, execution_time
FROM dba_mvref_stmt_stats
WHERE refresh_id = 1278;
```

**Example 9-21 Displaying Refresh Statements Used in the Current Refresh of an Materialized View**

This example displays the individual SQL statements that are used to the refresh the MY_SALES materialized view. A single refresh operation may consist of multiple steps, each of which executes a SQL statement. The details displayed in this example include the step number, SQL ID of the SQL statement, the SQL statement that is executed, and the execution time for the SQL statement.

```
SELECT step, sqlid, stmt, execution_time
FROM DBA_MVREF_STATS M, DBA_MVREF_STMT_STATS S
WHERE M.refresh_id = S.refresh_id and M.mv_name = 'MY_SALES'
ORDER BY step;
```

**See Also**

*Oracle AI Database Reference*

# 9.9 Analyzing Materialized View Refresh Performance Using Refresh Statistics

Materialized view refresh statistics that are stored in data dictionary views can be used to analyze the refresh performance of materialized views.

Refresh statistics provide detailed information that enables you to understand and analyze materialized view refresh operations and their performance. Typically, you analyze refresh statistics for critical or long running materialized view refresh operations. If a materialized view takes longer to refresh than it does normally, then you can analyze its past refresh times and change data to identify any differences that may account for the increased time (for example, 5 times more data that needs to be refreshed this time).

**To analyze materialized view refresh performance:**

1. Set the collection level and retention period for the materialized view to collect refresh statistics over a period of time.

   You can set these at the database level or at the materialized view level.

2. Identify the materialized views whose refresh performance needs to be analyzed.


Copyright © 2001, 2025, Oracle and/or its affiliates.

Typically, you would be interested in analyzing the refresh performance of a specific set of materialized views in the database. In this case, you can modify the refresh statistics settings for these materialized views as per your requirement.

3. Where multiple refresh operations take place over a period of time (for the materialized views you want to analyze), Oracle AI Database collects the desired refresh statistics.
4. Query the data dictionary views that store refresh statistics and analyze the refresh behavior of materialized views of interest over time to understand refresh behavior.

   The database stores both historical and current statistics which can be analyzed to understand refresh behavior.


Copyright © 2001, 2025, Oracle and/or its affiliates.

# 10 Dimensions

This chapter discusses using dimensions in a data warehouse: It contains the following topics:


- What are Dimensions?
- Creating Dimensions
- Viewing Dimensions
- Using Dimensions with Constraints
- Validating Dimensions
- Altering Dimensions
- Deleting Dimensions


## 10.1 What are Dimensions?

A dimension is a structure that categorizes data in order to enable users to answer business questions. Commonly used dimensions are customers, products, and time. For example, each sales channel of a clothing retailer might gather and store data regarding sales and reclamations of their Cloth assortment. The retail chain management can build a data warehouse to analyze the sales of its products across all stores over time and help answer questions such as:

- What is the effect of promoting one product on the sale of a related product that is not promoted?
- What are the sales of a product before and after a promotion?
- How does a promotion affect the various distribution channels?

The data in the retailer's data warehouse system has two important components: dimensions and facts. The dimensions are products, customers, promotions, channels, and time. One approach for identifying your dimensions is to review your reference tables, such as a product table that contains everything about a product, or a promotion table containing all information about promotions. The facts are sales (units sold) and profits. A data warehouse contains facts about the sales of each product at on a daily basis.

A typical relational implementation for such a data warehouse is a star schema. The fact information is stored in what is called a fact table, whereas the dimensional information is stored in dimension tables. In our example, each sales transaction record is uniquely defined as for each customer, for each product, for each sales channel, for each promotion, and for each day (time).

In Oracle AI Database, the dimensional information itself is stored in a dimension table. In addition, the database object dimension helps to organize and group dimensional information into hierarchies. This represents natural `1:n` relationships between columns or column groups (the levels of a hierarchy) that cannot be represented with constraint conditions. Going up a level in the hierarchy is called rolling up the data and going down a level in the hierarchy is called drilling down the data. In the retailer example:

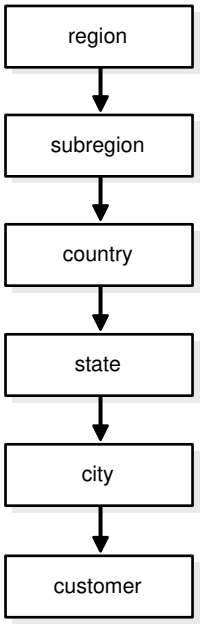
- Within the `time` dimension, months roll up to quarters, quarters roll up to years, and years roll up to all years.



Copyright © 2001, 2025, Oracle and/or its affiliates.

- Within the `product` dimension, products roll up to subcategories, subcategories roll up to categories, and categories roll up to all products.
- Within the `customer` dimension, customers roll up to `city`. Then cities roll up to `state`. Then states roll up to `country`. Then countries roll up to `subregion`. Finally, subregions roll up to `region`, as shown in Figure 10-1.

**Figure 10-1 Sample Rollup for a Customer Dimension**




Data analysis typically starts at higher levels in the dimensional hierarchy and gradually drills down if the situation warrants such analysis.

Dimension schema objects (dimensions) do not have to be defined. However, if your application uses dimensional modeling, it is worth spending time creating them as it can yield significant benefits, because they help query rewrite perform more complex types of rewrite. Dimensions are also beneficial to certain types of materialized view refresh operations and with the SQL Access Advisor. They are only mandatory if you use the SQL Access Advisor (a GUI tool for materialized view and index management) without a workload to recommend which materialized views and indexes to create, drop, or retain.

In spite of the benefits of dimensions, you must not create dimensions in any schema that does not fully satisfy the dimensional relationships described in this chapter. Incorrect results can be returned from queries otherwise.

**See Also**

- Data Warehousing Optimizations and Techniques for further details about schemas
- Basic Query Rewrite for Materialized Views for further details regarding query rewrite
- *Oracle AI Database SQL Tuning Guide* for further details regarding the SQL Access Advisor


Copyright © 2001, 2025, Oracle and/or its affiliates.

### 10.1.1 Requirements for Dimensions in Data Warehouses

- There must be a `1:n` relationship between a parent and children. A parent can have one or more children, but a child can have only one parent.
- There must be a `1:1` attribute relationship between hierarchy levels and their dependent dimension attributes. For example, if there is a column `fiscal_month_desc`, then a possible attribute relationship would be `fiscal_month_desc` to `fiscal_month_name`. For skip `NULL` levels, if a row of the relation of a skip level has a `NULL` value for the level column, then that row must have a `NULL` value for the attribute-relationship column, too.
- If the columns of a parent level and child level are in different relations, then the connection between them also requires a `1:n` join relationship. Each row of the child table must join with one and only one row of the parent table unless you use the `SKIP WHEN NULL` clause. This relationship is stronger than referential integrity alone, because it requires that the child join key must be non-null, that referential integrity must be maintained from the child join key to the parent join key, and that the parent join key must be unique.
- You must ensure (using database constraints if necessary) that the columns of each hierarchy level are non-null unless you use the `SKIP WHEN NULL` clause and that hierarchical integrity is maintained.
- An optional join key is a join key that connects the immediate non-skip child (if such a level exists), `CHILDLEV`, of a skip level to the nearest non-skip ancestor (again, if such a level exists), `ANCLEV`, of the skip level in the hierarchy. Also, this joinkey is allowed only when `CHILDLEV` and `ANCLEV` are defined over different relations.
- The hierarchies of a dimension can overlap or be disconnected from each other. However, the columns of a hierarchy level cannot be associated with more than one dimension.
- Join relationships that form cycles in the dimension graph are not supported. For example, a hierarchy level cannot be joined to itself either directly or indirectly.

> **Note**
>
> The information stored with a dimension objects is only declarative. The previously discussed relationships are not enforced with the creation of a dimension object. You should validate any dimension definition with the `DBMS_DIMENSION.VALIDATE_DIMENSION` procedure, as discussed in "Validating Dimensions".

## 10.2 Creating Dimensions

Before you can create a dimension object, the dimension tables must exist in the database possibly containing the dimension data. For example, if you create a customer dimension, one or more tables must exist that contain the city, state, and country information. In a star schema data warehouse, these dimension tables already exist. It is therefore a simple task to identify which ones will be used.

Now you can draw the hierarchies of a dimension as shown in Figure 10-1. For example, `city` is a child of `state` (because you can aggregate city-level data up to state), and `country`. This hierarchical information will be stored in the database object dimension.

In the case of normalized or partially normalized dimension representation (a dimension that is stored in more than one table), identify how these tables are joined. Note whether the joins


Copyright © 2001, 2025, Oracle and/or its affiliates.

between the dimension tables can guarantee that each child-side row joins with one and only one parent-side row. In the case of denormalized dimensions, determine whether the child-side columns uniquely determine the parent-side (or attribute) columns. If you use constraints to represent these relationships, they can be enabled with the `NOVALIDATE` and `RELY` clauses if the relationships represented by the constraints are guaranteed by other means.

You may want the capability to skip `NULL` levels in a dimension. An example of this is with Puerto Rico. You may want Puerto Rico to be included within a region of North America, but not include it within the state category. If you want this capability, use the `SKIP WHEN NULL` clause. See the sample dimension later in this section for more information and *Oracle AI Database SQL Language Reference* for syntax and restrictions.

You create a dimension using either the `CREATE DIMENSION` statement or the Dimension Wizard in Oracle Enterprise Manager. Within the `CREATE DIMENSION` statement, use the `LEVEL` clause to identify the names of the dimension levels.

This customer dimension contains a single hierarchy with a geographical rollup, with arrows drawn from the child level to the parent level, as shown in Figure 10-1.

Each arrow in this graph indicates that for any child there is one and only one parent. For example, each city must be contained in exactly one state and each state must be contained in exactly one country. States that belong to more than one country violate hierarchical integrity. Also, you must use the `SKIP WHEN NULL` clause if you want to include cities that do not belong to a state, such as Washington D.C. Hierarchical integrity is necessary for the correct operation of management functions for materialized views that include aggregates.

For example, you can declare a dimension `products_dim`, which contains levels `product`, `subcategory`, and `category`:

```
CREATE DIMENSION products_dim
       LEVEL product           IS (products.prod_id)
       LEVEL subcategory       IS (products.prod_subcategory)
       LEVEL category          IS (products.prod_category) ...
```

Each level in the dimension must correspond to one or more columns in a table in the database. Thus, level `product` is identified by the column `prod_id` in the products table and level `subcategory` is identified by a column called `prod_subcategory` in the same table.

In this example, the database tables are denormalized and all the columns exist in the same table. However, this is not a prerequisite for creating dimensions. "Using Normalized Dimension Tables to Create Dimensions" shows how to create a dimension `customers_dim` that has a normalized schema design using the `JOIN KEY` clause.

The next step is to declare the relationship between the levels with the `HIERARCHY` statement and give that hierarchy a name. A hierarchical relationship is a functional dependency from one level of a hierarchy to the next level in the hierarchy. Using the level names defined previously, the `CHILD OF` relationship denotes that each child's level value is associated with one and only one parent level value. The following statement declares a hierarchy `prod_rollup` and defines the relationship between products, subcategory, and category:

```
HIERARCHY prod_rollup
 (product         CHILD OF
  subcategory     CHILD OF
  category)
```

In addition to the `1:n` hierarchical relationships, dimensions also include `1:1` attribute relationships between the hierarchy levels and their dependent, determined dimension attributes. For example, the dimension `times_dim`, as defined in *Oracle AI Database Sample*


Copyright © 2001, 2025, Oracle and/or its affiliates.

*Schemas*, has columns `fiscal_month_desc`, `fiscal_month_name`, and `days_in_fiscal_month`. Their relationship is defined as follows:

```
LEVEL fis_month IS TIMES.FISCAL_MONTH_DESC
...
ATTRIBUTE fis_month DETERMINES
      (fiscal_month_name, days_in_fiscal_month)
```

The `ATTRIBUTE ... DETERMINES` clause relates `fis_month` to `fiscal_month_name and days_in_fiscal_month`. Note that this is a unidirectional determination. It is only guaranteed, that for a specific `fiscal_month`, for example, `1999-11`, you will find exactly one matching values for `fiscal_month_name`, for example, `November` and `days_in_fiscal_month`, for example, 28. You cannot determine a specific `fiscal_month_desc` based on the `fiscal_month_name`, which is `November` for every fiscal year.

In this example, suppose a query were issued that queried by `fiscal_month_name` instead of `fiscal_month_desc`. Because this `1:1` relationship exists between the attribute and the level, an already aggregated materialized view containing `fiscal_month_desc` can be joined back to the dimension information and used to identify the data.

A sample dimension definition follows:

```
CREATE DIMENSION products_dim
    LEVEL product          IS (products.prod_id)
    LEVEL subcategory      IS (products.prod_subcategory) [SKIP WHEN NULL]
    LEVEL category         IS (products.prod_category)
    HIERARCHY prod_rollup (
         product         CHILD OF
         subcategory     CHILD OF
         category)
    ATTRIBUTE product DETERMINES
     (products.prod_name, products.prod_desc,
      prod_weight_class, prod_unit_of_measure,
      prod_pack_size, prod_status, prod_list_price, prod_min_price)
    ATTRIBUTE subcategory DETERMINES
     (prod_subcategory, prod_subcategory_desc)
    ATTRIBUTE category DETERMINES
     (prod_category, prod_category_desc);
```

Alternatively, the *`extended_attribute_clause`* could have been used instead of the *`attribute_clause`*, as shown in the following example:

```
CREATE DIMENSION products_dim
    LEVEL product          IS (products.prod_id)
    LEVEL subcategory      IS (products.prod_subcategory)
    LEVEL category         IS (products.prod_category)
    HIERARCHY prod_rollup (
         product         CHILD OF
         subcategory     CHILD OF
         category
        )
    ATTRIBUTE product_info LEVEL product DETERMINES
     (products.prod_name, products.prod_desc,
      prod_weight_class, prod_unit_of_measure,
      prod_pack_size, prod_status, prod_list_price, prod_min_price)
    ATTRIBUTE subcategory DETERMINES
     (prod_subcategory, prod_subcategory_desc)
    ATTRIBUTE category DETERMINES
     (prod_category, prod_category_desc);
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

The design, creation, and maintenance of dimensions is part of the design, creation, and maintenance of your data warehouse schema. Once the dimension has been created, verify that it meets the requirements described in Requirements for Dimensions in Data Warehouses.

> **See Also**
>
> - Basic Query Rewrite for Materialized Views for further details of using dimensional information
> - *Oracle AI Database SQL Language Reference* for a complete description of the `CREATE DIMENSION` statement

## 10.2.1 Dropping and Creating Attributes with Columns

You can use the attribute clause in a `CREATE DIMENSION` statement to specify one or multiple columns that are uniquely determined by a hierarchy level.

If you use the *`extended_attribute_clause`* to create multiple columns determined by a hierarchy level, you can drop one attribute column without dropping them all. Alternatively, you can specify an attribute name for each attribute clause `CREATE` or `ALTER DIMENSION` statement so that an attribute name is specified for each attribute clause where multiple level-to-column relationships can be individually specified.

The following statement illustrates how you can drop a single column without dropping all columns:

```
CREATE DIMENSION products_dim
LEVEL product         IS (products.prod_id)
LEVEL subcategory     IS (products.prod_subcategory)
LEVEL category        IS (products.prod_category)
HIERARCHY prod_rollup (
          product        CHILD OF
          subcategory    CHILD OF category)
ATTRIBUTE product DETERMINES
         (products.prod_name, products.prod_desc,
          prod_weight_class, prod_unit_of_measure,
          prod_pack_size,prod_status, prod_list_price, prod_min_price)
ATTRIBUTE subcategory_att DETERMINES
         (prod_subcategory, prod_subcategory_desc)
ATTRIBUTE category DETERMINES
         (prod_category, prod_category_desc);

ALTER DIMENSION products_dim
DROP ATTRIBUTE subcategory_att LEVEL subcategory COLUMN prod_subcategory;
```

> **See Also**
>
> *Oracle AI Database SQL Language Reference* for a complete description of the `CREATE DIMENSION` statement

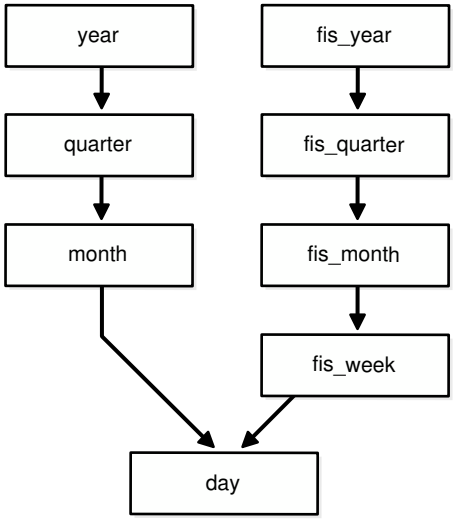
## 10.2.2 Using Multiple Hierarchies While Creating Joins

A single dimension definition can contain multiple hierarchies. Suppose our retailer wants to track the sales of certain items over time. The first step is to define the time dimension over


Copyright © 2001, 2025, Oracle and/or its affiliates.

which sales will be tracked. Figure 10-2 illustrates a dimension `times_dim` with two time hierarchies.

**Figure 10-2 times_dim Dimension with Two Time Hierarchies**




From the illustration, you can construct the hierarchy of the denormalized `time_dim` dimension's `CREATE DIMENSION` statement as follows.

```
CREATE DIMENSION times_dim
   LEVEL day         IS times.time_id
   LEVEL month       IS times.calendar_month_desc
   LEVEL quarter     IS times.calendar_quarter_desc
   LEVEL year        IS times.calendar_year
   LEVEL fis_week    IS times.week_ending_day
   LEVEL fis_month   IS times.fiscal_month_desc
   LEVEL fis_quarter IS times.fiscal_quarter_desc
   LEVEL fis_year    IS times.fiscal_year
   HIERARCHY cal_rollup    (
             day     CHILD OF
             month   CHILD OF
             quarter CHILD OF
             year
   )
   HIERARCHY fis_rollup    (
             day         CHILD OF
             fis_week    CHILD OF
             fis_month   CHILD OF
             fis_quarter CHILD OF
             fis_year
   ) <attribute determination clauses>;
```

## 10.2.3 Using Normalized Dimension Tables to Create Dimensions

The tables used to define a dimension may be normalized or denormalized and the individual hierarchies can be normalized or denormalized. If the levels of a hierarchy come from the same table, it is called a fully denormalized hierarchy. For example, `cal_rollup` in the `times_dim` dimension is a denormalized hierarchy. If levels of a hierarchy come from different tables, such a hierarchy is either a fully or partially normalized hierarchy. This section shows how to define a normalized hierarchy.


Copyright © 2001, 2025, Oracle and/or its affiliates.

Suppose the tracking of a customer's location is done by city, state, and country. This data is stored in the tables customers and countries. The customer dimension `customers_dim` is partially normalized because the data entities `cust_id` and `country_id` are taken from different tables. The clause `JOIN KEY` within the dimension definition specifies how to join together the levels in the hierarchy. The dimension statement is partially shown in the following. This example does not use the Oracle-supplied sample schemas.

```
CREATE DIMENSION customers_dim
   LEVEL customer  IS (customers.cust_id)
   LEVEL city      IS (customers.cust_city)
   LEVEL state     IS (customers.cust_state_province)
   LEVEL country   IS (countries.country_id)
   LEVEL subregion IS (countries.country_subregion)
   LEVEL region IS (countries.country_region)
   HIERARCHY geog_rollup (
      customer        CHILD OF
      city            CHILD OF
      state           CHILD OF
      country         CHILD OF
      subregion       CHILD OF
      region
   JOIN KEY (customers.country_id) REFERENCES country);
```

If you use the `SKIP WHEN NULL` clause, you can use the `JOIN KEY` clause to link levels that have a missing level in their hierarchy. For example, the following statement enables a state level that has been declared as `SKIP WHEN NULL` to join city and country:

```
JOIN KEY (city.country_id) REFERENCES country;
```

This ensures that the rows at customer and city levels can still be associated with the rows of country, subregion, and region levels.

# 10.3 Viewing Dimensions

Dimensions can be viewed through one of two methods:

- Viewing Dimensions With Oracle Enterprise Manager
- Viewing Dimensions With the DESCRIBE_DIMENSION Procedure

## 10.3.1 Viewing Dimensions With Oracle Enterprise Manager

All of the dimensions that exist in the data warehouse can be viewed using Oracle Enterprise Manager. Select the **Dimension** object from within the **Schema** icon to display all of the dimensions. Select a specific dimension to graphically display its hierarchy, levels, and any attributes that have been defined.

## 10.3.2 Viewing Dimensions With the DESCRIBE_DIMENSION Procedure

To view the definition of a dimension, use the `DESCRIBE_DIMENSION` procedure in the `DBMS_DIMENSION` package. For example, if a dimension is created in the `sh` sample schema with the following statements:

```
CREATE DIMENSION channels_dim
        LEVEL channel       IS (channels.channel_id)
        LEVEL channel_class IS (channels.channel_class)
        HIERARCHY channel_rollup (
                channel CHILD OF channel_class)
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
        ATTRIBUTE channel DETERMINES (channel_desc)
        ATTRIBUTE channel_class DETERMINES (channel_class);
```

Execute the DESCRIBE_DIMENSION procedure as follows:

```
SET SERVEROUTPUT ON FORMAT WRAPPED;  --to improve the display of info
EXECUTE DBMS_DIMENSION.DESCRIBE_DIMENSION('SH.CHANNELS_DIM');
```

You then see the following output results:

```
EXECUTE DBMS_DIMENSION.DESCRIBE_DIMENSION('SH.CHANNELS_DIM');
  DIMENSION SH.CHANNELS_DIM
    LEVEL CHANNEL IS SH.CHANNELS.CHANNEL_ID
    LEVEL CHANNEL_CLASS IS SH.CHANNELS.CHANNEL_CLASS

    HIERARCHY CHANNEL_ROLLUP (
             CHANNEL CHILD OF
             CHANNEL_CLASS)

    ATTRIBUTE CHANNEL LEVEL CHANNEL DETERMINES
SH.CHANNELS.CHANNEL_DESC
    ATTRIBUTE CHANNEL_CLASS LEVEL CHANNEL_CLASS DETERMINES
SH.CHANNELS.CHANNEL_CLASS
```

# 10.4 Using Dimensions with Constraints

Constraints play an important role with dimensions. Full referential integrity is sometimes enabled in data warehouses, but not always. This is because operational databases normally have full referential integrity and you can ensure that the data flowing into your data warehouse never violates the already established integrity rules.

It is recommended that constraints be enabled and, if validation time is a concern, then the NOVALIDATE clause should be used as follows:

```
ENABLE NOVALIDATE CONSTRAINT pk_time;
```

Primary and foreign keys should be implemented also. Referential integrity constraints and NOT NULL constraints on the fact tables provide information that query rewrite can use to extend the usefulness of materialized views.

In addition, you should use the RELY clause to inform query rewrite that it can rely upon the constraints being correct as follows:

```
ALTER TABLE time MODIFY CONSTRAINT pk_time RELY;
```

This information is also used for query rewrite. See Basic Query Rewrite for Materialized Views for more information.

If you use the SKIP WHEN NULL clause, at least one of the referenced level columns should not have NOT NULL constraints.

# 10.5 Validating Dimensions

The information of a dimension object is declarative only and not enforced by the database. If the relationships described by the dimensions are incorrect, incorrect results could occur. Therefore, you should verify the relationships specified by CREATE DIMENSION using the DBMS_DIMENSION.VALIDATE_DIMENSION procedure periodically.

This procedure is easy to use and has only four parameters:


Copyright © 2001, 2025, Oracle and/or its affiliates.

- `dimension`: the owner and name.
- `incremental`: set to `TRUE` to check only the new rows for tables of this dimension.
- `check_nulls`: set to `TRUE` to verify that all columns that are not in the levels containing a `SKIP WHEN NULL` clause are not null.
- `statement_id`: a user-supplied unique identifier to identify the result of each run of the procedure.

The following example validates the dimension `TIME_FN` in the `sh` schema:

```
@utldim.sql
EXECUTE DBMS_DIMENSION.VALIDATE_DIMENSION ('SH.TIME_FN', FALSE, TRUE,
  'my first example');
```

Before running the `VALIDATE_DIMENSION` procedure, you need to create a local table, `DIMENSION_EXCEPTIONS`, by running the provided script `utldim.sql`. If the `VALIDATE_DIMENSION` procedure encounters any errors, they are placed in this table. Querying this table will identify the exceptions that were found. The following illustrates a sample:

```
SELECT * FROM dimension_exceptions
WHERE statement_id = 'my first example';

STATEMENT_ID       OWNER  TABLE_NAME  DIMENSION_NAME   RELATIONSHIP   BAD_ROWID
------------       -----  ----------  --------------   ------------   ---------
my first example   SH     MONTH       TIME_FN          FOREIGN KEY    AAAAuwAAJAAAARwAAA
```

However, rather than query this table, it may be better to query the rowid of the invalid row to retrieve the actual row that has violated the constraint. In this example, the dimension `TIME_FN` is checking a table called `month`. It has found a row that violates the constraints. Using the rowid, you can see exactly which row in the month table is causing the problem, as in the following:

```
SELECT * FROM month
WHERE rowid IN (SELECT bad_rowid
                FROM dimension_exceptions
                WHERE statement_id = 'my first example');

MONTH    QUARTER   FISCAL_QTR    YEAR    FULL_MONTH_NAME    MONTH_NUMB
------   -------   ----------    ----    ---------------    ----------
199903     19981        19981    1998              March             3
```

# 10.6 Altering Dimensions

You can modify a dimension using the `ALTER DIMENSION` statement. You can add or drop a level, hierarchy, or attribute from the dimension using this command.

Referring to the time dimension in Figure 10-2, you can remove the attribute `fis_year`, drop the hierarchy `fis_rollup`, or remove the level `fiscal_year`. In addition, you can add a new level called `f_year` as in the following:

```
ALTER DIMENSION times_dim DROP ATTRIBUTE fis_year;
ALTER DIMENSION times_dim DROP HIERARCHY fis_rollup;
ALTER DIMENSION times_dim DROP LEVEL fis_year;
ALTER DIMENSION times_dim ADD LEVEL f_year IS times.fiscal_year;
```

If you used the *`extended_attribute_clause`* when creating the dimension, you can drop one attribute column without dropping all attribute columns. This is illustrated in Dropping and Creating Attributes with Columns, which shows the following statement:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
ALTER DIMENSION product_dim
DROP ATTRIBUTE size LEVEL prod_type COLUMN Prod_TypeSize;
```

If you try to remove anything with further dependencies inside the dimension, Oracle AI Database rejects the altering of the dimension. A dimension becomes invalid if you change any schema object that the dimension is referencing. For example, if the table on which the dimension is defined is altered, the dimension becomes invalid.

You can modify a dimension by adding a level containing a `SKIP WHEN NULL` clause, as in the following statement:

```
ALTER DIMENSION times_dim
ADD LEVEL f_year IS times.fiscal_year SKIP WHEN NULL;
```

You cannot, however, modify a level that contains a `SKIP WHEN NULL` clause. Instead, you need to drop the level and re-create it.

To check the status of a dimension, view the contents of the column `invalid` in the `ALL_DIMENSIONS` data dictionary view. To revalidate the dimension, use the `COMPILE` option as follows:

```
ALTER DIMENSION times_dim COMPILE;
```

Dimensions can also be modified or deleted using Oracle Enterprise Manager.

## 10.7 Deleting Dimensions

A dimension is removed using the `DROP DIMENSION` statement. For example, you could issue the following statement:

```
DROP DIMENSION times_dim;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

11

# Basic Query Rewrite for Materialized Views

This chapter discusses query rewrite in Oracle, and contains:


- Overview of Query Rewrite
- Ensuring that Query Rewrite Takes Effect
- Example of Query Rewrite


## 11.1 Overview of Query Rewrite

When base tables contain large amount of data, it is expensive and time-consuming to compute the required aggregates or to compute joins between these tables. In such cases, queries can take minutes or even hours. Because materialized views contain already precomputed aggregates and joins, Oracle AI Database employs an extremely powerful process called query rewrite to quickly answer the query using materialized views.

One of the major benefits of creating and maintaining materialized views is the ability to take advantage of query rewrite, which transforms a SQL statement expressed in terms of tables or views into a statement accessing one or more materialized views that are defined on the detail tables. The transformation is transparent to the end user or application, requiring no intervention and no reference to the materialized view in the SQL statement. Because query rewrite is transparent, materialized views can be added or dropped just like indexes without invalidating the SQL in the application code. "When Does Oracle Rewrite a Query?" describes the conditions that must be met for a query to be rewritten.

### 11.1.1 About Query Rewrite and the Optimizer

A query undergoes several checks to determine whether it is a candidate for query rewrite.

If the query fails any check, then the query is applied to the detail tables rather than the materialized view. The inability to rewrite can be costly in terms of response time and processing power.

The optimizer uses two different methods to determine when to rewrite a query in terms of a materialized view. The first method matches the SQL text of the query to the SQL text of the materialized view definition. If the first method fails, then the optimizer uses the more general method in which it compares joins, selections, data columns, grouping columns, and aggregate functions between the query and materialized views.

Query rewrite operates on queries and subqueries in the following types of SQL statements:

- `SELECT`
- `CREATE TABLE … AS SELECT`
- `INSERT INTO … SELECT`

It also operates on subqueries in the set operators `UNION`, `UNION ALL`, `INTERSECT`, `INTERSECT ALL`, `EXCEPT`, `EXCEPT ALL`, `MINUS`, and `MINUS ALL`, and subqueries in DML statements such as `INSERT`, `DELETE`, and `UPDATE`.



Copyright © 2001, 2025, Oracle and/or its affiliates.

Dimensions, constraints, and rewrite integrity levels affect whether a query is rewritten to use materialized views. Additionally, query rewrite can be enabled or disabled by `REWRITE` and `NOREWRITE` hints and the `QUERY_REWRITE_ENABLED` session parameter.

The `DBMS_MVIEW.EXPLAIN_REWRITE` procedure advises whether query rewrite is possible on a query and, if so, which materialized views are used. It also explains why a query cannot be rewritten.

### 11.1.2 When Does Oracle Rewrite a Query?

A query is rewritten only when a certain number of conditions are met:

- Query rewrite must be enabled for the session.
- A materialized view must be enabled for query rewrite.
- The rewrite integrity level should allow the use of the materialized view. For example, if a materialized view is not fresh and query rewrite integrity is set to `ENFORCED`, then the materialized view is not used.
- Either all or part of the results requested by the query must be obtainable from the precomputed result stored in the materialized view or views.

To test these conditions, the optimizer may depend on some of the data relationships declared by the user using constraints and dimensions, among others, hierarchies, referential integrity, and uniqueness of key data, and so on.

## 11.2 Ensuring that Query Rewrite Takes Effect

You must follow several conditions to enable query rewrite:

1. Individual materialized views must have the `ENABLE QUERY REWRITE` clause.

   If this step is not completed, as described in Enabling Query Rewrite for Materialized Views, then a materialized view is never eligible for query rewrite.
2. The session parameter `QUERY_REWRITE_ENABLED` must be set to `TRUE` (the default) or `FORCE`.
3. Cost-based optimization must be used by setting the initialization parameter `OPTIMIZER_MODE` to `ALL_ROWS`, `FIRST_ROWS`, or `FIRST_ROWS_`*n*.

You can use the `DBMS_ADVISOR.TUNE_MVIEW` procedure to optimize a `CREATE MATERIALIZED VIEW` statement to enable general `QUERY REWRITE`.

> **See Also**
>
> Initialization Parameters for Query Rewrite

### 11.2.1 Enabling Query Rewrite for Materialized Views

You can specify `ENABLE QUERY REWRITE` either with the `ALTER MATERIALIZED VIEW` statement or when the materialized view is created, as illustrated in the following:

```
CREATE MATERIALIZED VIEW join_sales_time_product_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_id, p.prod_name, t.time_id, t.week_ending_day,
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
      s.channel_id, s.promo_id, s.cust_id, s.amount_sold
FROM  sales s, products p, times t
WHERE s.time_id=t.time_id  AND s.prod_id = p.prod_id;
```

> **Note**
>
> Both Oracle join syntax (shown above) and ANSI join syntax are supported.

The `NOREWRITE` hint disables query rewrite in a SQL statement, overriding the `QUERY_REWRITE_ENABLED` parameter, and the `REWRITE` hint (when used with `mv_name`) restricts the eligible materialized views to those named in the hint.

## 11.2.2 About Initialization Parameters for Query Rewrite

Query rewrite behavior is controlled by certain database initialization parameters.

**Table 11-1 Initialization Parameters that Control Query Rewrite Behavior**

| Initialization Parameter Name | Initialization Parameter Value | Behavior of Query Rewrite |
|---|---|---|
| `OPTIMIZER_MODE` | `ALL_ROWS` (default), `FIRST_ROWS`, or `FIRST_ROWS_`*n* | With `OPTIMIZER_MODE` set to `FIRST_ROWS`, the optimizer uses a mix of costs and heuristics to find a best plan for fast delivery of the first few rows. When set to `FIRST_ROWS_`*n*, the optimizer uses a cost-based approach and optimizes with a goal of best response time to return the first *n* rows (where n = 1, 10, 100, 1000). |
| `QUERY_REWRITE_ENABLED` | `TRUE` (default), `FALSE`, or `FORCE` | This option enables the query rewrite feature of the optimizer, enabling the optimizer to utilize materialized views to enhance performance. If set to `FALSE`, this option disables the query rewrite feature of the optimizer and directs the optimizer not to rewrite queries using materialized views even when the estimated query cost of the unrewritten query is lower.<br>If set to `FORCE`, this option enables the query rewrite feature of the optimizer and directs the optimizer to rewrite queries using materialized views even when the estimated query cost of the unrewritten query is lower. |
| `QUERY_REWRITE_INTEGRITY` | `STALE_TOLERATED`, `TRUSTED`, or `ENFORCED` (the default) | This parameter is optional. However, if it is set, the value must be one of these specified in the Initialization Parameter Value column.<br>By default, the integrity level is set to `ENFORCED`. In this mode, all constraints must be validated. Therefore, if you use `ENABLE NOVALIDATE RELY` , certain types of query rewrite might not work. To enable query rewrite in this environment (where constraints have not been validated), you should set the integrity level to a lower level of granularity such as `TRUSTED` or `STALE_TOLERATED`. |

**Related Topics**

- About the Accuracy of Query Rewrite
  Query rewrite offers three levels of rewrite integrity that are controlled by the initialization parameter `QUERY_REWRITE_INTEGRITY`.


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 11.2.3 Controlling Query Rewrite

A materialized view is only eligible for query rewrite if the `ENABLE QUERY REWRITE` clause has been specified, either initially when the materialized view was first created or subsequently with an `ALTER MATERIALIZED VIEW` statement.

You can set the session parameters described previously for all sessions using the `ALTER SYSTEM SET` statement or in the initialization file. For a given user's session, `ALTER SESSION` can be used to disable or enable query rewrite for that session only. An example is the following:

```
ALTER SESSION SET QUERY_REWRITE_ENABLED = TRUE;
```

You can set the level of query rewrite for a session, thus allowing different users to work at different integrity levels. The possible statements are:

```
ALTER SESSION SET QUERY_REWRITE_INTEGRITY = STALE_TOLERATED;
ALTER SESSION SET QUERY_REWRITE_INTEGRITY = TRUSTED;
ALTER SESSION SET QUERY_REWRITE_INTEGRITY = ENFORCED;
```

## 11.2.4 About the Accuracy of Query Rewrite

Query rewrite offers three levels of rewrite integrity that are controlled by the initialization parameter `QUERY_REWRITE_INTEGRITY`.

The values that you can set for the `QUERY_REWRITE_INTEGRITY` parameter are as follows:

- `ENFORCED`

  This is the default mode. The optimizer only uses fresh data from the materialized views and only use those relationships that are based on `ENABLED VALIDATED` primary, unique, or foreign key constraints.

- `TRUSTED`

  In `TRUSTED` mode, the optimizer trusts that the relationships declared in dimensions and `RELY` constraints are correct. In this mode, the optimizer also uses prebuilt materialized views or materialized views based on views, and it uses relationships that are not enforced as well as those that are enforced. It also trusts declared but not `ENABLED VALIDATED` primary or unique key constraints and data relationships specified using dimensions. This mode offers greater query rewrite capabilities but also creates the risk of incorrect results if any of the trusted relationships you have declared are incorrect.

- `STALE_TOLERATED`

  In `STALE_TOLERATED` mode, the optimizer uses materialized views that are valid but contain stale data as well as those that contain fresh data. This mode offers the maximum rewrite capability but creates the risk of generating inaccurate results.

If rewrite integrity is set to the safest level, `ENFORCED`, the optimizer uses only enforced primary key constraints and referential integrity constraints to ensure that the results of the query are the same as the results when accessing the detail tables directly.

If the rewrite integrity is set to levels other than `ENFORCED`, there are several situations where the output with rewrite can be different from that without it:

- A materialized view can be out of synchronization with the primary copy of the data. This generally happens because the materialized view refresh procedure is pending following bulk load or DML operations to one or more detail tables of a materialized view. At some


Copyright © 2001, 2025, Oracle and/or its affiliates.

data warehouse sites, this situation is desirable because it is not uncommon for some materialized views to be refreshed at certain time intervals.

- The relationships implied by the dimension objects are invalid. For example, values at a certain level in a hierarchy do not roll up to exactly one parent value.
- The values stored in a prebuilt materialized view table might be incorrect.
- A wrong answer can occur because of bad data relationships defined by unenforced table or view constraints.

You can set `QUERY_REWRITE_INTEGRITY` either in your initialization parameter file or using an `ALTER SYSTEM` or `ALTER SESSION` statement.

## 11.2.5 About Privileges for Enabling Query Rewrite

Use of a materialized view is based not on privileges the user has on that materialized view, but on the privileges the user has on detail tables or views in the query.

The system privilege `GRANT QUERY REWRITE` lets you enable materialized views in your own schema for query rewrite only if all tables directly referenced by the materialized view are in that schema. The `GRANT GLOBAL QUERY REWRITE` privilege enables you to enable materialized views for query rewrite even if the materialized view references objects in other schemas. Alternatively, you can use the `QUERY REWRITE` object privilege on tables and views outside your schema.

The privileges for using materialized views for query rewrite are similar to those for definer's rights procedures.

## 11.2.6 Sample Schema and Materialized Views

The following sections use the `sh` sample schema and a few materialized views to illustrate how the optimizer uses data relationships to rewrite queries.

The query rewrite examples in this chapter mainly refer to the following materialized views. These materialized views do not necessarily represent the most efficient implementation for the `sh` schema. Instead, they are a base for demonstrating rewrite capabilities. Further examples demonstrating specific functionality can be found throughout this chapter.

The following materialized views contain joins and aggregates:

```
CREATE MATERIALIZED VIEW sum_sales_pscat_week_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_subcategory, t.week_ending_day,
       SUM(s.amount_sold) AS sum_amount_sold
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id  AND s.prod_id=p.prod_id
GROUP BY p.prod_subcategory, t.week_ending_day;

CREATE MATERIALIZED VIEW sum_sales_prod_week_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_id, t.week_ending_day, s.cust_id,
       SUM(s.amount_sold) AS sum_amount_sold
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id AND s.prod_id=p.prod_id
GROUP BY p.prod_id, t.week_ending_day, s.cust_id;

CREATE MATERIALIZED VIEW sum_sales_pscat_month_city_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold,
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
       COUNT(s.amount_sold) AS count_amount_sold
FROM   sales s, products p, times t, customers c
WHERE  s.time_id=t.time_id AND s.prod_id=p.prod_id AND s.cust_id=c.cust_id
GROUP BY p.prod_subcategory, t.calendar_month_desc, c.cust_city;
```

The following materialized views contain joins only:

```
CREATE MATERIALIZED VIEW join_sales_time_product_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_id, p.prod_name, t.time_id, t.week_ending_day,
       s.channel_id, s.promo_id, s.cust_id, s.amount_sold
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id AND s.prod_id = p.prod_id;

CREATE MATERIALIZED VIEW join_sales_time_product_oj_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_id, p.prod_name, t.time_id, t.week_ending_day,
       s.channel_id, s.promo_id, s.cust_id, s.amount_sold
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id AND s.prod_id=p.prod_id(+);
```

> **Note**
>
> Both Oracle join syntax (shown above) and ANSI join syntax are supported.

Although it is not a strict requirement, it is highly recommended that you collect statistics on the materialized views so that the optimizer can determine whether to rewrite the queries. You can do this either on a per-object base or for all newly created objects without statistics. The following is an example of a per-object base, shown for `join_sales_time_product_mv`:

```
EXECUTE DBMS_STATS.GATHER_TABLE_STATS ( -
   'SH','JOIN_SALES_TIME_PRODUCT_MV', estimate_percent   => 20, -
    block_sample   => TRUE, cascade   => TRUE);
```

The following illustrates a statistics collection for all newly created objects without statistics:

```
EXECUTE DBMS_STATS.GATHER_SCHEMA_STATS ( 'SH', -
   options            => 'GATHER EMPTY', -
   estimate_percent   =>  20, block_sample  =>  TRUE, -
   cascade            =>  TRUE);
```

## 11.2.7 How to Verify if Query Rewrite Occurred

Because query rewrite occurs transparently, special steps have to be taken to verify that a query has been rewritten. Of course, if the query runs faster, this should indicate that rewrite has occurred, but that is not proof. Therefore, to confirm that query rewrite does occur, use the `EXPLAIN PLAN` statement or the `DBMS_MVIEW.EXPLAIN_REWRITE` procedure. See "Verifying that Query Rewrite has Occurred" for further information.


Copyright © 2001, 2025, Oracle and/or its affiliates.

# 11.3 Example of Query Rewrite

This example illustrates the power of query rewrite with materialized views.

Consider the following materialized view, `cal_month_sales_mv`, which provides an aggregation of the dollar amount sold in every month:

```
CREATE MATERIALIZED VIEW cal_month_sales_mv
ENABLE QUERY REWRITE AS
SELECT t.calendar_month_desc, SUM(s.amount_sold) AS dollars
FROM sales s, times t WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

Let us assume that, in a typical month, the number of sales in the store is around one million. So this materialized aggregate view has the precomputed aggregates for the dollar amount sold for each month.

Consider the following query, which asks for the sum of the amount sold at the store for each calendar month:

```
SELECT t.calendar_month_desc, SUM(s.amount_sold)
FROM sales s, times t WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

> **Note**
>
> Both Oracle join syntax and ANSI join syntax now supported. For example, the previous query could be written using the ANSI syntax as follows.
>
> ```
> SELECT t.calendar_month_desc, SUM(s.amount_sold)
> FROM sales s JOIN times t ON s.time_id = t.time_id
> GROUP BY t.calendar_month_desc;
> ```

In the absence of the previous materialized view and query rewrite feature, Oracle AI Database must access the `sales` table directly and compute the sum of the amount sold to return the results. This involves reading many million rows from the `sales` table, which will invariably increase the query response time due to the disk access. The join in the query will also further slow down the query response as the join needs to be computed on many million rows.

In the presence of the materialized view `cal_month_sales_mv`, query rewrite will transparently rewrite the previous query into the following query:

```
SELECT calendar_month, dollars
FROM cal_month_sales_mv;
```

Because there are only a few dozen rows in the materialized view `cal_month_sales_mv` and no joins, Oracle AI Database returns the results instantly.


Copyright © 2001, 2025, Oracle and/or its affiliates.

12

# Advanced Query Rewrite for Materialized Views

This chapter discusses advanced query rewrite topics in Oracle, and contains:


- How Oracle Rewrites Queries
- Types of Query Rewrite
- Other Query Rewrite Considerations
- Advanced Query Rewrite Using Equivalences
- Creating Result Cache Materialized Views with Equivalences
- Query Rewrite and Materialized Views Based on Approximate Queries
- Verifying that Query Rewrite has Occurred
- Design Considerations for Improving Query Rewrite Capabilities


## 12.1 How Oracle Rewrites Queries

The optimizer uses a number of different methods to rewrite a query. The first step in determining whether query rewrite is possible is to see if the query satisfies the following prerequisites:

- Joins present in the materialized view are present in the SQL.
- There is sufficient data in the materialized view(s) to answer the query.

After that, it must determine how it will rewrite the query. The simplest case occurs when the result stored in a materialized view exactly matches what is requested by a query. The optimizer makes this type of determination by comparing the text of the query with the text of the materialized view definition. This text match method is most straightforward but the number of queries eligible for this type of query rewrite is minimal.

When the text comparison test fails, the optimizer performs a series of generalized checks based on the joins, selections, grouping, aggregates, and column data fetched. This is accomplished by individually comparing various clauses (`SELECT`, `FROM`, `WHERE`, `HAVING`, or `GROUP BY`) of a query with those of a materialized view.

You can use the following types of query rewrite: Query Rewrite Method 1: Text Match Rewrite or General Query Rewrite Methods.

This following topics discuss the optimizer in more detail:


- About Cost-Based Optimization and Query Rewrite
- General Query Rewrite Methods
- About Checks Made by Query Rewrite
- About Query Rewrite Using Dimensions




Copyright © 2001, 2025, Oracle and/or its affiliates.

## 12.1.1 About Cost-Based Optimization and Query Rewrite

When a query is rewritten, Oracle's cost-based optimizer compares the cost of the rewritten query and original query and chooses the cheaper execution plan.

Query rewrite is available with cost-based optimization. Oracle AI Database optimizes the input query with and without rewrite and selects the least costly alternative. The optimizer rewrites a query by rewriting one or more query blocks, one at a time.

If query rewrite has a choice between several materialized views to rewrite a query block, it selects the ones which can result in reading in the least amount of data. After a materialized view has been selected for a rewrite, the optimizer then tests whether the rewritten query can be rewritten further with other materialized views. This process continues until no further rewrites are possible. Then the rewritten query is optimized and the original query is optimized. The optimizer compares these two optimizations and selects the least costly alternative.

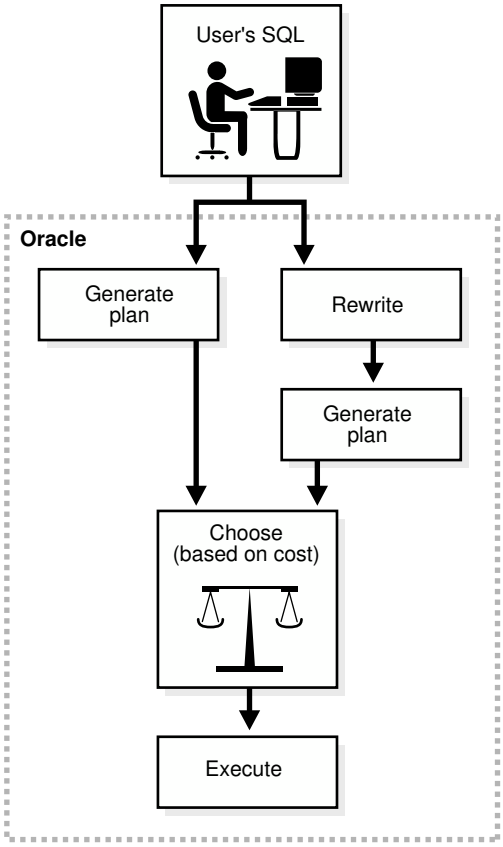
Because optimization is based on cost, it is important to collect statistics both on tables involved in the query and on the tables representing materialized views. Statistics are fundamental measures, such as the number of rows in a table, that are used to calculate the cost of a rewritten query. They are created by using the `DBMS_STATS` package.

Queries that contain inline or named views are also candidates for query rewrite. When a query contains a named view, the view name is used to do the matching between a materialized view and the query. When a query contains an inline view, the inline view can be merged into the query before matching between a materialized view and the query occurs.

Figure 12-1 presents a graphical view of the cost-based approach used during the rewrite process.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Figure 12-1 The Query Rewrite Process**




## 12.1.2 General Query Rewrite Methods

The optimizer has a number of different types of query rewrite methods that it can choose from to answer a query. When text match rewrite is not possible, this group of rewrite methods is known as general query rewrite. The advantage of using these more advanced techniques is that one or more materialized views can be used to answer a number of different queries and the query does not always have to match the materialized view exactly for query rewrite to occur.

When using general query rewrite methods, the optimizer uses data relationships on which it can depend, such as primary and foreign key constraints and dimension objects. For example, primary key and foreign key relationships tell the optimizer that each row in the foreign key table joins with at most one row in the primary key table. Furthermore, if there is a `NOT NULL` constraint on the foreign key, it indicates that each row in the foreign key table must join to exactly one row in the primary key table. A dimension object describes the relationship between, say, day, months, and year, which can be used to roll up data from the day to the month level.

Data relationships such as these are very important for query rewrite because they tell what type of result is produced by joins, grouping, or aggregation of data. Therefore, to maximize the rewritability of a large set of queries when such data relationships exist in a database, you should declare constraints and dimensions.


Copyright © 2001, 2025, Oracle and/or its affiliates.

> **See Also**
>
> When are Constraints and Dimensions Needed for Query Rewrite?

### 12.1.2.1 When are Constraints and Dimensions Needed for Query Rewrite?

Table 12-1 illustrates when dimensions and constraints are required for different types of query rewrite. These types of query rewrite are described throughout this chapter.

**Table 12-1 Dimension and Constraint Requirements for Query Rewrite**

| Query Rewrite Types | Dimensions | Primary Key/Foreign Key/Not Null Constraints |
|---|---|---|
| Matching SQL Text | Not Required | Not Required |
| Join Back | Required OR | Required |
| Aggregate Computability | Not Required | Not Required |
| Aggregate Rollup | Not Required | Not Required |
| Rollup Using a Dimension | Required | Not Required |
| Filtering the Data | Not Required | Not Required |
| PCT Rewrite | Not Required | Not Required |
| Multiple Materialized Views | Not Required | Not Required |

## 12.1.3 About Checks Made by Query Rewrite

For query rewrite to occur, there are a number of checks that the data must pass. These checks are:

- Join Compatibility Check for Query Rewrite
- Data Sufficiency Check for Query Rewrite
- Grouping Compatibility Check for Query Rewrite
- Aggregate Computability Check for Query Rewrite

### 12.1.3.1 Join Compatibility Check for Query Rewrite

In this check, the joins in a query are compared against the joins in a materialized view. In general, this comparison results in the classification of joins into three categories:

- Common joins that occur in both the query and the materialized view. These joins form the common subgraph.

  See Common Joins.

- Delta joins that occur in the query but not in the materialized view. These joins form the query delta subgraph.

  See Query Delta Joins.

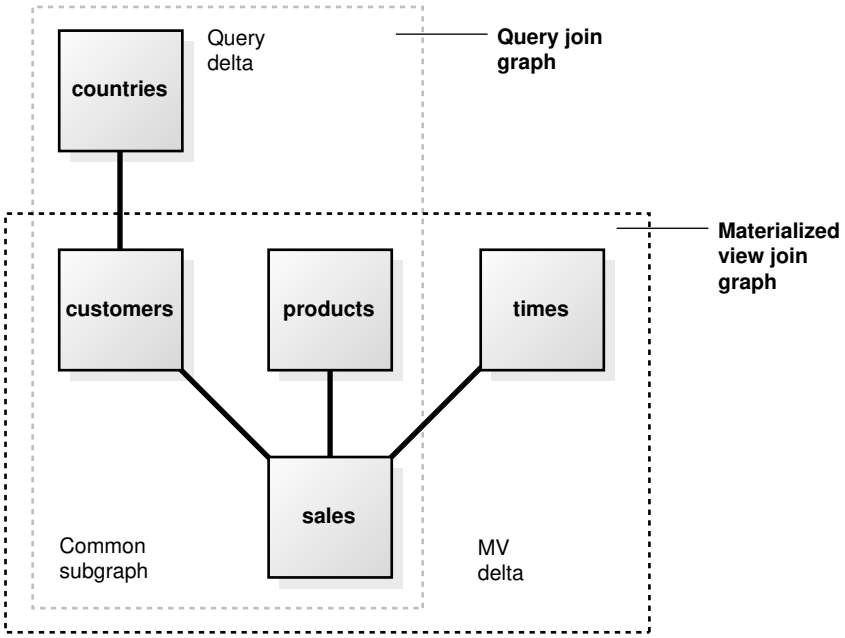
- Delta joins that occur in the materialized view but not in the query. These joins form the materialized view delta subgraph.

  See Materialized View Delta Joins.


Copyright © 2001, 2025, Oracle and/or its affiliates.

These can be visualized as shown in Figure 12-2.

**Figure 12-2 Query Rewrite Subgraphs**




> **See Also**
>
> About Checks Made by Query Rewrite

### 12.1.3.1.1 Common Joins

The common join pairs between the two must be of the same type, or the join in the query must be derivable from the join in the materialized view. For example, if a materialized view contains an outer join of table `A` with table `B`, and a query contains an inner join of table `A` with table `B`, the result of the inner join can be derived by filtering the antijoin rows from the result of the outer join. For example, consider the following query:

```
SELECT p.prod_name, t.week_ending_day, SUM(s.amount_sold)
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id AND s.prod_id = p.prod_id
AND mv.week_ending_day BETWEEN TO_DATE('01-AUG-1999', 'DD-MON-YYYY')
                           AND TO_DATE('10-AUG-1999', 'DD-MON-YYYY')
GROUP BY p.prod_name, mv.week_ending_day;
```

The common joins between this query and the materialized view `join_sales_time_product_mv` are:

```
s.time_id = t.time_id AND s.prod_id = p.prod_id
```

They match exactly and the query can be rewritten as follows:

```
SELECT p.prod_name, mv.week_ending_day, SUM(s.amount_sold)
FROM   join_sales_time_product_mv
WHERE  mv.week_ending_day BETWEEN TO_DATE('01-AUG-1999','DD-MON-YYYY')
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
                          AND   TO_DATE('10-AUG-1999','DD-MON-YYYY')
GROUP BY mv.prod_name, mv.week_ending_day;
```

The query could also be answered using the `join_sales_time_product_oj_mv` materialized view where inner joins in the query can be derived from outer joins in the materialized view. The rewritten version (transparently to the user) filters out the antijoin rows. The rewritten query has the following structure:

```
SELECT mv.prod_name, mv.week_ending_day, SUM(mv.amount_sold)
FROM   join_sales_time_product_oj_mv mv
WHERE  mv.week_ending_day BETWEEN TO_DATE('01-AUG-1999','DD-MON-YYYY')
  AND  TO_DATE('10-AUG-1999','DD-MON-YYYY') AND mv.prod_id IS NOT NULL
GROUP BY mv.prod_name, mv.week_ending_day;
```

In general, if you use an outer join in a materialized view containing only joins, you should put in the materialized view either the primary key or the rowid on the right side of the outer join. For example, in the previous example, `join_sales_time_product_oj_mv`, there is a primary key on both `sales` and `products`.

Another example of when a materialized view containing only joins is used is the case of a semijoin rewrites. That is, a query contains either an `EXISTS` or an `IN` subquery with a single table. Consider the following query, which reports the products that had sales greater than $1,000:

```
SELECT DISTINCT p.prod_name
FROM products p
WHERE EXISTS (SELECT p.prod_id, SUM(s.amount_sold) FROM sales s
              WHERE p.prod_id=s.prod_id HAVING SUM(s.amount_sold) > 1000)
              GROUP BY p.prod_id);
```

This query could also be represented as:

```
SELECT DISTINCT p.prod_name
FROM products p WHERE p.prod_id IN (SELECT s.prod_id FROM sales s
                   WHERE s.amount_sold > 1000);
```

This query contains a semijoin (`s.prod_id = p.prod_id`) between the `products` and the `sales` table.

This query can be rewritten to use either the `join_sales_time_product_mv` materialized view, if foreign key constraints are active or `join_sales_time_product_oj_mv` materialized view, if primary keys are active. Observe that both materialized views contain `s.prod_id=p.prod_id`, which can be used to derive the semijoin in the query. The query is rewritten with `join_sales_time_product_mv` as follows:

```
SELECT mv.prod_name
FROM (SELECT DISTINCT mv.prod_name FROM  join_sales_time_product_mv mv
      WHERE mv.amount_sold > 1000);
```

If the materialized view `join_sales_time_product_mv` is partitioned by `time_id`, then this query is likely to be more efficient than the original query because the original join between `sales` and `products` has been avoided. The query could be rewritten using `join_sales_time_product_oj_mv` as follows.

```
SELECT mv.prod_name
FROM (SELECT DISTINCT mv.prod_name FROM join_sales_time_product_oj_mv mv
      WHERE mv.amount_sold > 1000 AND mv.prod_id IS NOT NULL);
```

Rewrites with semi-joins are restricted to materialized views with joins only and are not possible for materialized views with joins and aggregates.


Copyright © 2001, 2025, Oracle and/or its affiliates.

> **See Also**
>
> About Checks Made by Query Rewrite

### 12.1.3.1.2 Query Delta Joins

A **query delta join** is a join that appears in the query but not in the materialized view. Any number and type of delta joins in a query are allowed and they are simply retained when the query is rewritten with a materialized view. In order for the retained join to work, the materialized view must contain the joining key. Upon rewrite, the materialized view is joined to the appropriate tables in the query delta. For example, consider the following query against the `sh` sample schema.

```
SELECT p.prod_name, t.week_ending_day, c.cust_city, SUM(s.amount_sold)
FROM   sh.sales s, sh.products p, sh.times t, sh.customers c
WHERE  s.time_id=t.time_id AND s.prod_id = p.prod_id
AND    s.cust_id = c.cust_id
GROUP BY p.prod_name, t.week_ending_day, c.cust_city;
```

Using the materialized view `join_sales_time_product_mv`, common joins are: `s.time_id=t.time_id` and `s.prod_id=p.prod_id`. The delta join in the query is `s.cust_id=c.cust_id`. The rewritten form then joins the `join_sales_time_product_mv` materialized view with the `customers` table as follows:

```
SELECT mv.prod_name, mv.week_ending_day, c.cust_city, SUM(mv.amount_sold)
FROM   join_sales_time_product_mv mv, customers c
WHERE  mv.cust_id = c.cust_id
GROUP BY mv.prod_name, mv.week_ending_day, c.cust_city;
```

> **See Also**
>
> About Checks Made by Query Rewrite

### 12.1.3.1.3 Materialized View Delta Joins

A **materialized view delta join** is a join that appears in the materialized view but not the query. All delta joins in a materialized view are required to be lossless with respect to the result of common joins. A lossless join guarantees that the result of common joins is not restricted. A **lossless** join is one where, if two tables called `A` and `B` are joined together, rows in table `A` will always match with rows in table `B` and no data will be lost, hence the term lossless join. For example, every row with the foreign key matches a row with a primary key provided no nulls are allowed in the foreign key. Therefore, to guarantee a lossless join, it is necessary to have `FOREIGN KEY`, `PRIMARY KEY`, and `NOT NULL` constraints on appropriate join keys. Alternatively, if the join between tables `A` and `B` is an outer join (`A` being the outer table), it is lossless as it preserves all rows of table `A`.

All delta joins in a materialized view are required to be non-duplicating with respect to the result of common joins. A non-duplicating join guarantees that the result of common joins is not duplicated. For example, a non-duplicating join is one where, if table `A` and table `B` are joined together, rows in table `A` will match with at most one row in table `B` and no duplication occurs. To guarantee a non-duplicating join, the key in table `B` must be constrained to unique values by using a primary key or unique constraint.


Copyright © 2001, 2025, Oracle and/or its affiliates.

Consider the following query that joins `sales` and `times` in the `sh` sample schema:

```
SELECT t.week_ending_day, SUM(s.amount_sold)
FROM   sh.sales s, sh.times t
WHERE s.time_id = t.time_id AND t.week_ending_day BETWEEN TO_DATE
 ('01-MAR-2022', 'DD-MON-YYYY') AND TO_DATE('10-AUG-2022', 'DD-MON-YYYY')
GROUP BY week_ending_day;
```

The materialized view `join_sales_time_product_mv` has an additional join (`s.prod_id=p.prod_id`) between `sales` and `products`. This is the delta join in `join_sales_time_product_mv`. You can rewrite the query if this join is lossless and non-duplicating. This is the case if `s.prod_id` is a foreign key to `p.prod_id` and is not null. The query is therefore rewritten as:

```
SELECT week_ending_day, SUM(amount_sold)
FROM   join_sales_time_product_mv
WHERE  week_ending_day BETWEEN TO_DATE('01-MAR-2022', 'DD-MON-YYYY')
       AND TO_DATE('10-AUG-2022', 'DD-MON-YYYY')
GROUP BY week_ending_day;
```

The query can also be rewritten with the materialized view `join_sales_time_product_mv_oj` where foreign key constraints are not needed. This view contains an outer join (`s.prod_id=p.prod_id(+)`) between `sales` and `products`. This makes the join lossless. If `p.prod_id` is a primary key, then the non-duplicating condition is satisfied as well and optimizer rewrites the query as follows:

```
SELECT week_ending_day, SUM(amount_sold)
FROM   join_sales_time_product_oj_mv
WHERE  week_ending_day BETWEEN TO_DATE('01-AUG-1999', 'DD-MON-YYYY')
  AND  TO_DATE('10-AUG-1999', 'DD-MON-YYYY')
GROUP BY week_ending_day;
```

The query can also be rewritten with the materialized view `join_sales_time_product_mv_oj` where foreign key constraints are not needed. This view contains an outer join (`s.prod_id=p.prod_id(+)`) between `sales` and `products`. This makes the join lossless. If `p.prod_id` is a primary key, then the non-duplicating condition is satisfied as well and optimizer rewrites the query as follows:

```
SELECT week_ending_day, SUM(amount_sold)
FROM   join_sales_time_product_oj_mv
WHERE  week_ending_day BETWEEN TO_DATE('01-MAR-2022', 'DD-MON-YYYY')
  AND  TO_DATE('10-AUG-2022', 'DD-MON-YYYY')
GROUP BY week_ending_day;
```

Note that the outer join in the definition of `join_sales_time_product_mv_oj` is not necessary because the primary key - foreign key relationship between `sales` and `products` in the `sh` schema is already lossless. It is used for demonstration purposes only, and would be necessary if `sales.prod_id` were nullable, thus violating the losslessness of the join condition `sales.prod_id = products.prod_id`.

Current limitations restrict most rewrites with outer joins to materialized views with joins only. There is limited support for rewrites with materialized aggregate views with outer joins, so those materialized views should rely on foreign key constraints to assure losslessness of materialized view delta joins.


Copyright © 2001, 2025, Oracle and/or its affiliates.

> **See Also**
>
> About Checks Made by Query Rewrite

#### 12.1.3.1.4 Join Equivalence Recognition

Query rewrite is able to make many transformations based upon the recognition of equivalent joins. Query rewrite recognizes the following construct as being equivalent to a join:

```
WHERE table1.column1 = F(args)   /* sub-expression A */
AND table2.column2 = F(args)     /* sub-expression B */
```

If `F(args)` is a PL/SQL function that is declared to be deterministic and the arguments to both invocations of `F` are the same, then the combination of subexpression `A` with subexpression `B` be can be recognized as a join between `table1.column1` and `table2.column2`. That is, the following expression is equivalent to the previous expression:

```
WHERE table1.column1 = F(args)          /* sub-expression A */
AND table2.column2 = F(args)            /* sub-expression B */
AND table1.column1 = table2.column2     /* join-expression J */
```

Because join-expression `J` can be inferred from sub-expression `A` and subexpression `B`, the inferred join can be used to match a corresponding join of `table1.column1 = table2.column2` in a materialized view.

### 12.1.3.2 Data Sufficiency Check for Query Rewrite

In this check, the optimizer determines if the necessary column data requested by a query can be obtained from a materialized view. For this, the equivalence of one column with another is used. For example, if an inner join between table `A` and table `B` is based on a join predicate `A.X = B.X`, then the data in column `A.X` equals the data in column `B.X` in the result of the join. This data property is used to match column `A.X` in a query with column `B.X` in a materialized view or vice versa. For example, consider the following query against the `sh` sample schema:

```
SELECT p.prod_name, s.time_id, t.week_ending_day, SUM(s.amount_sold)
FROM sh.sales s, sh.products p, sh.times t
WHERE s.time_id=t.time_id AND s.prod_id = p.prod_id
GROUP BY p.prod_name, s.time_id, t.week_ending_day;
```

This query can be answered with `join_sales_time_product_mv` even though the materialized view does not have `s.time_id`. Instead, it has `t.time_id`, which, through a join condition `s.time_id=t.time_id`, is equivalent to `s.time_id`. Thus, the optimizer might select the following rewrite:

```
SELECT prod_name, time_id, week_ending_day, SUM(amount_sold)
FROM join_sales_time_product_mv
GROUP BY prod_name, time_id, week_ending_day;
```

### 12.1.3.3 Grouping Compatibility Check for Query Rewrite

This check is required only if both the materialized view and the query contain a `GROUP BY` clause. The optimizer first determines if the grouping of data requested by a query is exactly the same as the grouping of data stored in a materialized view. In other words, the level of grouping is the same in both the query and the materialized view. If the materialized views groups on all the columns and expressions in the query and also groups on additional columns


Copyright © 2001, 2025, Oracle and/or its affiliates.

or expressions, query rewrite can reaggregate the materialized view over the grouping columns and expressions of the query to derive the same result requested by the query.

### 12.1.3.4 Aggregate Computability Check for Query Rewrite

This check is required only if both the query and the materialized view contain aggregates. Here the optimizer determines if the aggregates requested by a query can be derived or computed from one or more aggregates stored in a materialized view. For example, if a query requests `AVG(X)` and a materialized view contains `SUM(X)` and `COUNT(X)`, then `AVG(X)` can be computed as `SUM(X)/COUNT(X)`.

If the grouping compatibility check determined that the rollup of aggregates stored in a materialized view is required, then the aggregate computability check determines if it is possible to roll up each aggregate requested by the query using aggregates in the materialized view.

## 12.1.4 About Query Rewrite Using Dimensions

This section discusses the following aspects of using dimensions in a rewrite environment:

- Benefits of Using Dimensions in a Query Rewrite Environment
- How to Define Dimensions for Query Rewrite

### 12.1.4.1 Benefits of Using Dimensions in a Query Rewrite Environment

A dimension defines a hierarchical (parent/child) relationships between columns, where all the columns do not have to come from the same table.

Dimension definitions increase the possibility of query rewrite because they help to establish functional dependencies between the columns. In addition, dimensions can express intra-table relationships that cannot be expressed by constraints. A dimension definition does not occupy additional storage. Rather, a dimension definition establishes metadata that describes the intra- and inter-dimensional relationships within your schema. Before creating a materialized view, the first step is to review the schema and define the dimensions as this can significantly improve the chances of rewriting a query.

### 12.1.4.2 How to Define Dimensions for Query Rewrite

For any given schema, use the following steps to create dimensions:

1. Identify all dimensions and dimension tables in the schema
2. Identify the hierarchies within each dimension
3. Identify the attribute dependencies within each level of the hierarchy
4. Identify joins from the fact table in a data warehouse to each dimension

Remember to set the parameter `QUERY_REWRITE_INTEGRITY` to `TRUSTED` or `STALE_TOLERATED` for query rewrite to take advantage of the relationships declared in dimensions.

**Identify all dimensions and dimension tables in the schema**

If the dimensions are normalized, that is, stored in multiple tables, then check that a join between the dimension tables guarantees that each child-side row joins with one and only one parent-side row. In the case of denormalized dimensions, check that the child-side columns uniquely determine the parent-side (or attribute) columns. Failure to abide by these rules may result in incorrect results being returned from queries.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Identify the hierarchies within each dimension**

As an example, day is a child of month (you can aggregate day level data up to month), and quarter is a child of year.

**Identify the attribute dependencies within each level of the hierarchy**

As an example, identify that `calendar_month_name` is an attribute of month.

**Identify joins from the fact table in a data warehouse to each dimension**

Then check that each join can guarantee that each fact row joins with one and only one dimension row. This condition must be declared, and optionally enforced, by adding `FOREIGN KEY` and `NOT NULL` constraints on the fact key columns and `PRIMARY KEY` constraints on the parent-side join keys. If these relationships can be guaranteed by other data handling procedures (for example, your load process), these constraints can be enabled using the `NOVALIDATE` option to avoid the time required to validate that every row in the table conforms to the constraints. The `RELY` clause is also required for all nonvalidated constraints to make them eligible for use in query rewrite.

### 12.1.4.2.1 Example SQL Statement to Create Time Dimensions

```
CREATE DIMENSION times_dim
LEVEL day IS TIMES.TIME_ID
LEVEL month IS TIMES.CALENDAR_MONTH_DESC
LEVEL quarter IS TIMES.CALENDAR_QUARTER_DESC
LEVEL year IS TIMES.CALENDAR_YEAR
LEVEL fis_week IS TIMES.WEEK_ENDING_DAY
LEVEL fis_month  IS TIMES.FISCAL_MONTH_DESC
LEVEL fis_quarter IS TIMES.FISCAL_QUARTER_DESC
LEVEL fis_year IS TIMES.FISCAL_YEAR
        HIERARCHY cal_rollup
        (day  CHILD OF month CHILD OF quarter CHILD OF year)
        HIERARCHY fis_rollup
        (day  CHILD OF fis_week CHILD OF fis_month CHILD OF fis_quarter
        CHILD OF fis_year)

        ATTRIBUTE day DETERMINES
        (day_number_in_week, day_name, day_number_in_month,
         calendar_week_number)

        ATTRIBUTE month DETERMINES
        (calendar_month_desc, calendar_month_number, calendar_month_name,
         days_in_cal_month, end_of_cal_month)

        ATTRIBUTE quarter DETERMINES
        (calendar_quarter_desc, calendar_quarter_number,days_in_cal_quarter,
         end_of_cal_quarter)

        ATTRIBUTE year DETERMINES
        (calendar_year,  days_in_cal_year, end_of_cal_year)

        ATTRIBUTE fis_week DETERMINES
        (week_ending_day, fiscal_week_number);
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

# 12.2 Types of Query Rewrite

Queries that have aggregates that require computations over a large number of rows or joins between very large tables can be expensive and thus can take a long time to return the results. Query rewrite transparently rewrites such queries using materialized views that have pre-computed results, so that the queries can be answered almost instantaneously. These materialized views can be broadly categorized into two groups, namely materialized aggregate views and materialized join views. Materialized aggregate views are tables that have pre-computed aggregate values for columns from original tables. Similarly, materialized join views are tables that have pre-computed joins between columns from original tables. Query rewrite transforms an incoming query to fetch the results from materialized view columns. Because these columns contain already pre-computed results, the incoming query can be answered almost instantaneously.

This section discusses the following methods that can be used to rewrite a query:

- Query Rewrite Method 1: Text Match Rewrite
- Query Rewrite Method 2: Join Back
- Query Rewrite Method 3: Aggregate Computability
- Query Rewrite Method 4: Aggregate Rollup
- Query Rewrite Method 5: Rollup Using a Dimension
- Query Rewrite Method 6: When Materialized Views Have Only a Subset of Data
- Partition Change Tracking (PCT) Rewrite
- About Query Rewrite Using Multiple Materialized Views

## 12.2.1 Query Rewrite Method 1: Text Match Rewrite

The query rewrite engine always initially tries to compare the text of incoming query with the text of the definition of any potential materialized views to rewrite the query. This is because the overhead of doing a simple text comparison is usually negligible comparing to the cost of doing a complex analysis required for the general rewrite.

The query rewrite engine uses two text match methods, full text match rewrite and partial text match rewrite. In full text match the entire text of a query is compared against the entire text of a materialized view definition (that is, the entire `SELECT` expression), ignoring the white space during text comparison. For example, assume that you have the following materialized view, `sum_sales_pscat_month_city_mv`:

```
CREATE MATERIALIZED VIEW sum_sales_pscat_month_city_mv
ENABLE QUERY REWRITE AS
   SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
   SUM(s.amount_sold) AS sum_amount_sold,
   COUNT(s.amount_sold) AS count_amount_sold
   FROM sales s, products p, times t, customers c
   WHERE s.time_id=t.time_id
      AND     s.prod_id=p.prod_id
      AND     s.cust_id=c.cust_id
  GROUP BY p.prod_subcategory, t.calendar_month_desc, c.cust_city;
```

Consider the following query:

```
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold,
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
        COUNT(s.amount_sold) AS count_amount_sold
        FROM sales s, products p, times t, customers c
        WHERE s.time_id=t.time_id
          AND     s.prod_id=p.prod_id
          AND     s.cust_id=c.cust_id
        GROUP BY p.prod_subcategory, t.calendar_month_desc, c.cust_city;
```

This query matches `sum_sales_pscat_month_city_mv` (white space excluded) and is rewritten as:

```
SELECT mv.prod_subcategory, mv.calendar_month_desc, mv.cust_city,
       mv.sum_amount_sold, mv.count_amount_sold
FROM   sum_sales_pscat_month_city_mv;
```

When full text match fails, the optimizer then attempts a partial text match. In this method, the text starting from the `FROM` clause of a query is compared against the text starting with the `FROM` clause of a materialized view definition. Therefore, the following query can be rewritten:

```
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
       AVG(s.amount_sold)
FROM   sales s, products p, times t, customers c
WHERE  s.time_id=t.time_id AND s.prod_id=p.prod_id
AND    s.cust_id=c.cust_id
GROUP BY p.prod_subcategory, t.calendar_month_desc, c.cust_city;
```

This query is rewritten as:

```
SELECT mv.prod_subcategory, mv.calendar_month_desc, mv.cust_city,
       mv.sum_amount_sold/mv.count_amount_sold
FROM   sum_sales_pscat_month_city_mv mv;
```

Note that, under the partial text match rewrite method, the average of sales aggregate required by the query is computed using the sum of sales and count of sales aggregates stored in the materialized view.

When neither text match succeeds, the optimizer uses a general query rewrite method.

Text match rewrite can distinguish contexts where the difference between uppercase and lowercase is significant and where it is not. For example, the following statements are equivalent:

```
SELECT X, 'aBc' FROM Y

Select x, 'aBc' From y
```

## 12.2.2 Query Rewrite Method 2: Join Back

If some column data requested by a query cannot be obtained from a materialized view, the optimizer further determines if it can be obtained based on a data relationship called a functional dependency. When the data in a column can determine data in another column, such a relationship is called a functional dependency or functional determinance. For example, if a table contains a primary key column called `prod_id` and another column called `prod_name`, then, given a `prod_id` value, it is possible to look up the corresponding `prod_name`. The opposite is not true, which means a `prod_name` value need not relate to a unique `prod_id`.

When the column data required by a query is not available from a materialized view, such column data can still be obtained by joining the materialized view back to the table that contains required column data provided the materialized view contains a key that functionally determines the required column data. For example, consider the following query:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
SELECT p.prod_category, t.week_ending_day, SUM(s.amount_sold)
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id  AND s.prod_id=p.prod_id AND p.prod_category='Golf'
GROUP BY p.prod_category, t.week_ending_day;
```

The materialized view `sum_sales_prod_week_mv` contains `p.prod_id`, but not `p.prod_category`. However, you can join `sum_sales_prod_week_mv` back to `products` to retrieve `prod_category` because `prod_id` functionally determines `prod_category`. The optimizer rewrites this query using `sum_sales_prod_week_mv` as follows:

```
SELECT p.prod_category, mv.week_ending_day, SUM(mv.sum_amount_sold)
FROM   sum_sales_prod_week_mv mv, products p
WHERE  mv.prod_id=p.prod_id AND p.prod_category='Golf'
GROUP BY p.prod_category, mv.week_ending_day;
```

Here the `products` table is called a joinback table because it was originally joined in the materialized view but joined again in the rewritten query.

You can declare functional dependency in two ways:

- Using the primary key constraint (as shown in the previous example)
- Using the `DETERMINES` clause of a dimension

The `DETERMINES` clause of a dimension definition might be the only way you could declare functional dependency when the column that determines another column cannot be a primary key. For example, the `products` table is a denormalized dimension table that has columns `prod_id`, `prod_name`, and `prod_subcategory` that functionally determines `prod_subcat_desc` and `prod_category` that determines `prod_cat_desc`.

The first functional dependency can be established by declaring `prod_id` as the primary key, but not the second functional dependency because the `prod_subcategory` column contains duplicate values. In this situation, you can use the `DETERMINES` clause of a dimension to declare the second functional dependency.

The following dimension definition illustrates how functional dependencies are declared:

```
CREATE DIMENSION products_dim
        LEVEL product           IS (products.prod_id)
        LEVEL subcategory       IS (products.prod_subcategory)
        LEVEL category          IS (products.prod_category)
        HIERARCHY prod_rollup (
                product         CHILD OF
                subcategory     CHILD OF
                category
        )
        ATTRIBUTE product DETERMINES products.prod_name
        ATTRIBUTE product DETERMINES products.prod_desc
        ATTRIBUTE subcategory DETERMINES products.prod_subcat_desc
        ATTRIBUTE category DETERMINES products.prod_cat_desc;
```

The hierarchy `prod_rollup` declares hierarchical relationships that are also `1:n` functional dependencies. The `1:1` functional dependencies are declared using the `DETERMINES` clause, as seen when `prod_subcategory` functionally determines `prod_subcat_desc`.

If the following materialized view is created:

```
CREATE MATERIALIZED VIEW sum_sales_pscat_week_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_subcategory, t.week_ending_day,
       SUM(s.amount_sold) AS sum_amount_sole
FROM sales s, products p, times t
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
WHERE s.time_id = t.time_id AND s.prod_id = p.prod_id
GROUP BY p.prod_subcategory, t.week_ending_day;
```

Then consider the following query:

```
SELECT p.prod_subcategory_desc, t.week_ending_day, SUM(s.amount_sold)
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id AND s.prod_id=p.prod_id
AND    p.prod_subcat_desc LIKE '%Men'
GROUP BY p.prod_subcat_desc, t.week_ending_day;
```

This can be rewritten by joining `sum_sales_pscat_week_mv` to the `products` table so that `prod_subcat_desc` is available to evaluate the predicate. However, the join is based on the `prod_subcategory` column, which is not a primary key in the `products` table; therefore, it allows duplicates. This is accomplished by using an inline view that selects distinct values and this view is joined to the materialized view as shown in the rewritten query.

```
SELECT iv.prod_subcat_desc, mv.week_ending_day, SUM(mv.sum_amount_sold)
FROM  sum_sales_pscat_week_mv mv,
     (SELECT DISTINCT prod_subcategory, prod_subcat_desc
      FROM products) iv
WHERE  mv.prod_subcategory=iv.prod_subcategory
AND iv.prod_subcat_desc LIKE '%Men'
GROUP BY iv.prod_subcat_desc, mv.week_ending_day;
```

This type of rewrite is possible because `prod_subcategory` functionally determines `prod_subcategory_desc` as declared in the dimension.

## 12.2.3 Query Rewrite Method 3: Aggregate Computability

Query rewrite can also occur when the optimizer determines if the aggregates requested by a query can be derived or computed from one or more aggregates stored in a materialized view. For example, if a query requests `AVG(X)` and a materialized view contains `SUM(X)` and `COUNT(X)`, then `AVG(X)` can be computed as `SUM(X)/COUNT(X)`.

In addition, if it is determined that the rollup of aggregates stored in a materialized view is required, then, if it is possible, query rewrite also rolls up each aggregate requested by the query using aggregates in the materialized view.

For example, `SUM(sales)` at the city level can be rolled up to `SUM(sales)` at the state level by summing all `SUM(sales)` aggregates in a group with the same state value. However, `AVG(sales)` cannot be rolled up to a coarser level unless `COUNT(sales)` or `SUM(sales)` is also available in the materialized view. Similarly, `VARIANCE(sales)` or `STDDEV(sales)` cannot be rolled up unless both `COUNT(sales)` and `SUM(sales)` are also available in the materialized view. For example, consider the following query from the `sh` sample schema:

```
ALTER TABLE times MODIFY CONSTRAINT time_pk RELY;
ALTER TABLE customers MODIFY CONSTRAINT customers_pk RELY;
ALTER TABLE sales MODIFY CONSTRAINT sales_time_pk RELY;
ALTER TABLE sales MODIFY CONSTRAINT sales_customer_fk RELY;
SELECT  p.prod_subcategory, AVG(s.amount_sold) AS avg_sales
FROM  sh.sales s, sh.products p WHERE s.prod_id = p.prod_id
GROUP BY p.prod_subcategory;
```

This statement can be rewritten with materialized view `sum_sales_pscat_month_city_mv` provided the joins between `sales` and `times` and `sales` and `customers` are lossless and non-duplicating. Further, the query groups by `prod_subcategory` whereas the materialized view groups by `prod_subcategory`, `calendar_month_desc` and `cust_city`, which means the


Copyright © 2001, 2025, Oracle and/or its affiliates.

aggregates stored in the materialized view have to be rolled up. The optimizer rewrites the query as the following:

```
SELECT mv.prod_subcategory, SUM(mv.sum_amount_sold)/COUNT(mv.count_amount_sold)
   AS avg_sales
FROM sum_sales_pscat_month_city_mv mv
GROUP BY mv.prod_subcategory;
```

The argument of an aggregate such as `SUM` can be an arithmetic expression such as `A+B`. The optimizer tries to match an aggregate `SUM(A+B)` in a query with an aggregate `SUM(A+B)` or `SUM(B+A)` stored in a materialized view. In other words, expression equivalence is used when matching the argument of an aggregate in a query with the argument of a similar aggregate in a materialized view. To accomplish this, Oracle converts the aggregate argument expression into a canonical form such that two different but equivalent expressions convert into the same canonical form. For example, `A*(B-C)`, `A*B-C*A`, `(B-C)*A`, and `-A*C+A*B` all convert into the same canonical form and, therefore, they are successfully matched.

## 12.2.4 Query Rewrite Method 4: Aggregate Rollup

If the grouping of data requested by a query is at a coarser level than the grouping of data stored in a materialized view, the optimizer can still use the materialized view to rewrite the query. For example, the materialized view `sum_sales_pscat_week_mv` groups by `prod_subcategory` and `week_ending_day`. This query groups by `prod_subcategory`, a coarser grouping granularity:

```
ALTER TABLE times MODIFY CONSTRAINT time_pk RELY;
ALTER TABLE sales MODIFY CONSTRAINT sales_time_fk RELY;
SELECT p.prod_subcategory, SUM(s.amount_sold) AS sum_amount
FROM   sh.sales s, sh.products p WHERE  s.prod_id=p.prod_id
GROUP BY p.prod_subcategory;
```

Therefore, the optimizer rewrites this query as:

```
SELECT mv.prod_subcategory, SUM(mv.sum_amount_sold)
FROM   sum_sales_pscat_week_mv mv
GROUP BY mv.prod_subcategory;
```

## 12.2.5 Query Rewrite Method 5: Rollup Using a Dimension

When reporting is required at different levels in a hierarchy, materialized views do not have to be created at each level in the hierarchy provided dimensions have been defined. This is because query rewrite can use the relationship information in the dimension to roll up the data in the materialized view to the required level in the hierarchy.

In the following example, a query requests data grouped by `prod_category` while a materialized view stores data grouped by `prod_subcategory`. If `prod_subcategory` is a `CHILD OF` `prod_category` (see the dimension example earlier), the grouped data stored in the materialized view can be further grouped by `prod_category` when the query is rewritten. In other words, aggregates at `prod_subcategory` level (finer granularity) stored in a materialized view can be rolled up into aggregates at `prod_category` level (coarser granularity).

For example, consider the following `sh` sample schema query:

```
SELECT p.prod_category, t.week_ending_day, SUM(s.amount_sold) AS sum_amount
FROM   sh.sales s, sh.products p, sh.times t
WHERE  s.time_id=t.time_id AND s.prod_id=p.prod_id
GROUP BY p.prod_category, t.week_ending_day;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

Because prod_subcategory functionally determines prod_category, sum_sales_pscat_week_mv can be used with a joinback to products to retrieve prod_category column data, and then aggregates can be rolled up to prod_category level, as shown in the following:

```
SELECT pv.prod_category, mv.week_ending_day, SUM(mv.sum_amount_sold)
FROM   sum_sales_pscat_week_mv mv,
       (SELECT DISTINCT prod_subcategory, prod_category
        FROM products) pv
WHERE mv.prod_subcategory= pv.prod_subcategory
GROUP BY pv.prod_category, mv.week_ending_day;
```

## 12.2.6 Query Rewrite Method 6: When Materialized Views Have Only a Subset of Data

Oracle supports rewriting of queries so that they will use materialized views in which the HAVING or WHERE clause of the materialized view contains a selection of a subset of the data in a table or tables. For example, only those customers who live in New Hampshire. In other words, the WHERE clause in the materialized view will be WHERE state = 'New Hampshire'.

To perform this type of query rewrite, Oracle must determine if the data requested in the query is contained in, or is a subset of, the data stored in the materialized view. The following sections detail the conditions where Oracle can solve this problem and thus rewrite a query to use a materialized view that contains a filtered portion of the data in the detail table.

To determine if query rewrite can occur on filtered data, a selection computability check is performed when both the query and the materialized view contain selections (non-joins) and the check is done on the WHERE as well as the HAVING clause. If the materialized view contains selections and the query does not, then the selection compatibility check fails because the materialized view is more restrictive than the query. If the query has selections and the materialized view does not, then the selection compatibility check is not needed.

A materialized view's WHERE or HAVING clause can contain a join, a selection, or both, and still be used to rewrite a query. Predicate clauses containing expressions, or selecting rows based on the values of particular columns, are examples of non-join predicates.

This section contains the following topics:

- Query Rewrite Definitions When Materialized Views Have Only a Subset of Data
- Selection Categories When Materialized Views Have Only a Subset of Data
- Examples of Query Rewrite Selection
- About Handling of the HAVING Clause in Query Rewrite
- About Query Rewrite When the Materialized View has an IN-List

### 12.2.6.1 Query Rewrite Definitions When Materialized Views Have Only a Subset of Data

Before describing what is possible when query rewrite works with only a subset of the data, the following definitions are useful:

- *join relop*

  Is one of the following (=, <, <=, >, >=)

- *selection relop*


Copyright © 2001, 2025, Oracle and/or its affiliates.

Is one of the following (=, <, <=, >, >=, !=, [NOT] BETWEEN | IN| LIKE |NULL)

- *join predicate*

  Is of the form (*column1 join relop column2*), where columns are from different tables within the same FROM clause in the current query block. So, for example, an outer reference is not possible.

- *selection predicate*

  Is of the form *left-hand-side-expression relop right-hand-side-expression*. All non-join predicates are selection predicates. The left-hand side usually contains a column and the right-hand side contains the values. For example, color='red' means the left-hand side is color and the right-hand side is 'red' and the relational operator is (=).

## 12.2.6.2 Selection Categories When Materialized Views Have Only a Subset of Data

Selections are categorized into the following cases:

- Simple

  Simple selections are of the form *expression relop constant*.

- Complex

  Complex selections are of the form *expression relop expression*.

- Range

  Range selections are of a form such as WHERE (cust_last_name BETWEEN 'abacrombe' AND 'anakin').

  Note that simple selections with relational operators (<,<=,>,>=)are also considered range selections.

- IN-lists

  Single and multi-column IN-lists such as WHERE(prod_id) IN (102, 233, ....).

  Note that selections of the form (column1='v1' OR column1='v2' OR column1='v3' OR ....) are treated as a group and classified as an IN-list.

- IS [NOT] NULL
- [NOT] LIKE
- Other

  Other selections are when it cannot determine the boundaries for the data. For example, EXISTS.

When comparing a selection from the query with a selection from the materialized view, the left-hand side of both selections are compared.

If the left-hand side selections match, then the right-hand side values are checked for containment. That is, the right-hand side values of the query selection must be contained by right-hand side values of the materialized view selection.

You can also use expressions in selection predicates. This process resembles the following:

```
expression relational operator constant
```

Where *expression* can be any arbitrary arithmetic expression allowed by the Oracle AI Database. The expression in the materialized view and the query must match. Oracle attempts to discern expressions that are logically equivalent, such as A+B and B+A, and always recognizes identical expressions as being equivalent.


Copyright © 2001, 2025, Oracle and/or its affiliates.

You can also use queries with an expression on both sides of the operator or user-defined functions as operators. Query rewrite occurs when the complex predicate in the materialized view and the query are logically equivalent. This means that, unlike exact text match, terms could be in a different order and rewrite can still occur, as long as the expressions are equivalent.

## 12.2.6.3 Examples of Query Rewrite Selection

Here are a number of examples showing how query rewrite can still occur when the data is being filtered.

**Example 12-1 Single Value Selection**

If the query contains the following clause:

```
WHERE prod_id = 102
```

And, if a materialized view contains the following clause:

```
WHERE prod_id BETWEEN 0 AND 200
```

Then, the left-hand side selections match on `prod_id` and the right-hand side value of the query `102` is within the range of the materialized view, so query rewrite is possible.

**Example 12-2 Bounded Range Selection**

A selection can be a bounded range (a range with an upper and lower value). For example, if the query contains the following clause:

```
WHERE prod_id > 10 AND prod_id < 50
```

And if a materialized view contains the following clause:

```
WHERE prod_id BETWEEN 0 AND 200
```

Then, the selections are matched on `prod_id` and the query range is within the materialized view range. In this example, notice that both query selections are based on the same column.

**Example 12-3 Selection With Expression**

If the query contains the following clause:

```
WHERE (sales.amount_sold * .07) BETWEEN 1.00 AND 100.00
```

And if a materialized view contains the following clause:

```
WHERE (sales.amount_sold * .07) BETWEEN 0.0 AND 200.00
```

Then, the selections are matched on `(sales.amount_sold *.07)` and the right-hand side value of the query is within the range of the materialized view, therefore query rewrite is possible. Complex selections such as this require that the left-hand side and the right-hand side be matched within range of the materialized view.

**Example 12-4 Exact Match Selections**

If the query contains the following clause:

```
WHERE (cost.unit_price * 0.95) > (cost_unit_cost  * 1.25)
```

And if a materialized view contains the following:

```
WHERE (cost.unit_price * 0.95) > (cost_unit_cost  * 1.25)
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

If the left-hand side and the right-hand side match the materialized view and the *selection_relop* is the same, then the selection can usually be dropped from the rewritten query. Otherwise, the selection must be kept to filter out extra data from the materialized view.

If query rewrite can drop the selection from the rewritten query, all columns from the selection may not have to be in the materialized view so more rewrites can be done. This ensures that the materialized view data is not more restrictive than the query.

**Example 12-5 More Selection in the Query**

Selections in the query do not have to be matched by any selections in the materialized view but, if they are, then the right-hand side values must be contained by the materialized view. For example, if the query contains the following clause:

```
WHERE prod_name = 'Shorts' AND prod_category = 'Men'
```

And if a materialized view contains the following clause:

```
WHERE prod_category = 'Men'
```

Then, in this example, only selection with `prod_category` is matched. The query has an extra selection that is not matched but this is acceptable because if the materialized view selects `prod_name` or selects a column that can be joined back to the detail table to get `prod_name`, then query rewrite is possible. The only requirement is that query rewrite must have a way of applying the `prod_name` selection to the materialized view.

**Example 12-6 No Rewrite Because of Fewer Selections in the Query**

If the query contains the following clause:

```
WHERE prod_category = 'Men'
```

And if a materialized view contains the following clause:

```
WHERE prod_name = 'Shorts' AND prod_category = 'Men'
```

Then, the materialized view selection with `prod_name` is not matched. The materialized view is more restrictive that the query because it only contains the product Shorts, therefore, query rewrite does not occur.

**Example 12-7 Multi-Column IN-List Selections**

Query rewrite also checks for cases where the query has a multi-column `IN`-list where the columns are fully matched by individual columns from the materialized view single column `IN`-lists. For example, if the query contains the following:

```
WHERE (prod_id, cust_id) IN ((1022, 1000), (1033, 2000))
```

And if a materialized view contains the following:

```
WHERE prod_id IN (1022,1033) AND cust_id IN (1000, 2000)
```

Then, the materialized view `IN`-lists are matched by the columns in the query multi-column `IN`-list. Furthermore, the right-hand side values of the query selection are contained by the materialized view so that rewrite occurs.

**Example 12-8 Selections Using IN-Lists**

Selection compatibility also checks for cases where the materialized view has a multi-column `IN`-list where the columns are fully matched by individual columns or columns from `IN`-lists in the query. For example, if the query contains the following:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
WHERE prod_id = 1022 AND cust_id IN (1000, 2000)
```

And if a materialized view contains the following:

```
WHERE (prod_id, cust_id) IN ((1022, 1000), (1022, 2000))
```

Then, the materialized view `IN`-list columns are fully matched by the columns in the query selections. Furthermore, the right-hand side values of the query selection are contained by the materialized view. So rewrite succeeds.

**Example 12-9 Multiple Selections or Expressions**

If the query contains the following clause:

```
WHERE (city_population > 15000 AND city_population < 25000
   AND state_name = 'New Hampshire')
```

And if a materialized view contains the following clause:

```
WHERE (city_population < 5000 AND state_name = 'New York') OR
   (city_population BETWEEN 10000 AND 50000 AND state_name = 'New Hampshire')
```

Then, the query is said to have a single disjunct (group of selections separated by `AND`) and the materialized view has two disjuncts separated by `OR`. The single query disjunct is contained by the second materialized view disjunct so selection compatibility succeeds. It is clear that the materialized view contains more data than needed by the query so the query can be rewritten.

## 12.2.6.4 About Handling of the HAVING Clause in Query Rewrite

Query rewrite can also occur when the query specifies a range of values for an aggregate in the `HAVING` clause, such as `SUM(s.amount_sold) BETWEEN 10000 AND 20000`, as long as the range specified is within the range specified in the materialized view.

```
CREATE MATERIALIZED VIEW product_sales_mv
BUILD IMMEDIATE
REFRESH FORCE
ENABLE QUERY REWRITE AS
SELECT p.prod_name, SUM(s.amount_sold) AS dollar_sales
FROM sh.products p, sh.sales s
WHERE p.prod_id = s.prod_id
GROUP BY prod_name
HAVING SUM(s.amount_sold) BETWEEN 5000 AND 50000;
```

Then, a query such as the following could be rewritten:

```
SELECT p.prod_name, SUM(s.amount_sold) AS dollar_sales
FROM sh.products p, sh.sales s WHERE p.prod_id = s.prod_id
GROUP BY prod_name
HAVING SUM(s.amount_sold) BETWEEN 5000 AND 50000;
```

This query is rewritten as follows:

```
SELECT mv.prod_name, mv.dollar_sales FROM product_sales_mv mv
WHERE mv.dollar_sales BETWEEN 5000 AND 50000;
```

## 12.2.6.5 About Query Rewrite When the Materialized View has an IN-List

You can use query rewrite when the materialized view contains an `IN`-list. For example, given the following materialized view definition:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
CREATE MATERIALIZED VIEW popular_promo_sales_mv
BUILD IMMEDIATE
REFRESH FORCE
ENABLE QUERY REWRITE AS
SELECT p.promo_name, SUM(s.amount_sold) AS sum_amount_sold
FROM  promotions p, sales s
WHERE s.promo_id = p.promo_id
AND p.promo_name IN ('coupon', 'premium', 'giveaway')
GROUP BY promo_name;
```

The following query can be rewritten:

```
SELECT p.promo_name, SUM(s.amount_sold)
FROM  promotions p, sales s
WHERE s.promo_id = p.promo_id AND p.promo_name IN ('coupon', 'premium')
GROUP BY p.promo_name;
```

This query is rewritten as follows:

```
SELECT * FROM popular_promo_sales_mv mv
WHERE mv.promo_name IN ('coupon', 'premium');
```

## 12.2.7 Partition Change Tracking (PCT) Rewrite

PCT rewrite enables the optimizer to accurately rewrite queries with fresh data using materialized views that are only partially fresh. To do so, Oracle AI Database keeps track of which partitions in the detail tables have been updated. Oracle AI Database then tracks which rows in the materialized view originate from the affected partitions in the detail tables. The optimizer is then able to use those portions of the materialized view that are known to be fresh. You can check details about freshness with the `DBA_MVIEWS`, `DBA_DETAIL_RELATIONS`, and `DBA_MVIEW_DETAIL_PARTITION` views. See "Viewing Partition Freshness" for examples of using these views.

The optimizer uses PCT rewrite in `QUERY_REWRITE_INTEGRITY = ENFORCED` and `TRUSTED` modes. The optimizer does not use PCT rewrite in `STALE_TOLERATED` mode because data freshness is not considered in that mode. Also, for PCT rewrite to occur, a `WHERE` clause is required.

You can use PCT rewrite with partitioning, but hash partitioning is not supported. The following topics discuss aspects of using PCT:

- PCT Rewrite Based on Range Partitioned Tables
- PCT Rewrite Based on Range-List Partitioned Tables
- PCT Rewrite Based on List Partitioned Tables
- PCT Rewrite and PMARKER
- PCT Rewrite Using Rowid as PMARKER

### 12.2.7.1 PCT Rewrite Based on Range Partitioned Tables

The following example illustrates a PCT rewrite example where the materialized view is PCT enabled through partition key and the underlying base table is range partitioned on the time key.

```
CREATE TABLE part_sales_by_time (time_id, prod_id, amount_sold,
       quantity_sold)
  PARTITION BY RANGE (time_id)
  (
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
    PARTITION old_data
      VALUES LESS THAN (TO_DATE('01-01-1999', 'DD-MM-YYYY'))
      PCTFREE 0
      STORAGE (INITIAL 8M),
    PARTITION quarter1
      VALUES LESS THAN (TO_DATE('01-04-1999', 'DD-MM-YYYY'))
      PCTFREE 0
      STORAGE (INITIAL 8M),
    PARTITION quarter2
      VALUES LESS THAN (TO_DATE('01-07-1999', 'DD-MM-YYYY'))
      PCTFREE 0
      STORAGE (INITIAL 8M),
    PARTITION quarter3
      VALUES LESS THAN (TO_DATE('01-10-1999', 'DD-MM-YYYY'))
      PCTFREE 0
      STORAGE (INITIAL 8M),
    PARTITION quarter4
      VALUES LESS THAN (TO_DATE('01-01-2000', 'DD-MM-YYYY'))
      PCTFREE 0
      STORAGE (INITIAL 8M),
    PARTITION max_partition
      VALUES LESS THAN (MAXVALUE)
      PCTFREE 0
      STORAGE (INITIAL 8M)
  )
  AS
  SELECT s.time_id, s.prod_id, s.amount_sold, s.quantity_sold
  FROM sales s;
```

Then create a materialized view that contains the total number of products sold by date.

```
CREATE MATERIALIZED VIEW  sales_in_1999_mv
  BUILD IMMEDIATE
  REFRESH FORCE ON DEMAND
  ENABLE QUERY REWRITE
  AS
  SELECT s.time_id, s.prod_id, p.prod_name, SUM(quantity_sold)
  FROM part_sales_by_time s, products p
  WHERE p.prod_id = s.prod_id
    AND s.time_id BETWEEN TO_DATE('01-01-1999', 'DD-MM-YYYY')
    AND TO_DATE('31-12-1999', 'DD-MM-YYYY')
  GROUP BY s.time_id, s.prod_id, p.prod_name;
```

Note that the following query will be rewritten with materialized view `sales_in_1999_mv`:

```
SELECT s.time_id, p.prod_name, SUM(quantity_sold)
  FROM part_sales_by_time s, products p
  WHERE p.prod_id = s.prod_id
    AND s.time_id < TO_DATE('01-02-1999', 'DD-MM-YYYY')
    AND s.time_id >= TO_DATE('01-01-1999', 'DD-MM-YYYY')
  GROUP BY s.time_id, p.prod_name;
```

If you add a row to `quarter4` in `part_sales_by_time` as:

```
INSERT INTO part_sales_by_time
  VALUES (TO_DATE('26-12-1999', 'DD-MM-YYYY'),38920,2500, 20);

commit;
```

Then the materialized view `sales_in_1999_mv` becomes stale. With PCT rewrite, you can rewrite queries that request data from only the fresh portions of the materialized view. Note that


Copyright © 2001, 2025, Oracle and/or its affiliates.

because the materialized view `sales_in_1999_mv` has the `time_id` in its `SELECT` and `GROUP BY` clause, it is PCT enabled so the following query will be rewritten successfully as no data from `quarter4` is requested.

```
SELECT s.time_id, p.prod_name, SUM(quantity_sold)
  FROM part_sales_by_time s, products p
  WHERE p.prod_id = s.prod_id
  AND s.time_id < TO_DATE('01-07-1999', 'DD-MM-YYYY')
  AND s.time_id >= TO_DATE('01-03-1999', 'DD-MM-YYYY')
  GROUP BY s.time_id, p.prod_name;
```

The following query cannot be rewritten if multiple materialized view rewrite is set to off. Because multiple materialized view rewrite is on by default, the following query is rewritten with materialized view and base tables:

```
SELECT s.time_id, p.prod_name, SUM(quantity_sold)
  FROM part_sales_by_time s, products p
  WHERE p.prod_id = s.prod_id
  AND s.time_id < TO_DATE('31-10-1999', 'DD-MM-YYYY') AND
      s.time_id > TO_DATE('01-07-1999', 'DD-MM-YYYY')
  GROUP BY s.time_id, p.prod_name;
```

## 12.2.7.2 PCT Rewrite Based on Range-List Partitioned Tables

If the detail table is range-list partitioned, a materialized view that depends on this detail table can support PCT at both the partitioning and subpartitioning levels. If both the partition and subpartition keys are present in the materialized view, PCT can be done at a finer granularity; materialized view refreshes can be done to smaller portions of the materialized view and more queries could be rewritten with a stale materialized view. Alternatively, if only the partition key is present in the materialized view, PCT can be done with courser granularity.

Consider the following range-list partitioned table:

```
CREATE TABLE sales_par_range_list
 (calendar_year, calendar_month_number, day_number_in_month,
  country_name, prod_id, prod_name, quantity_sold, amount_sold)
PARTITION BY RANGE (calendar_month_number)
SUBPARTITION BY LIST (country_name)
 (PARTITION q1 VALUES LESS THAN (4)
 (SUBPARTITION q1_America VALUES
 ('United States of America', 'Argentina'),
   SUBPARTITION q1_Asia VALUES ('Japan', 'India'),
   SUBPARTITION q1_Europe VALUES ('France', 'Spain', 'Ireland')),
   PARTITION q2 VALUES LESS THAN (7)
  (SUBPARTITION q2_America VALUES
   ('United States of America', 'Argentina'),
   SUBPARTITION q2_Asia VALUES ('Japan', 'India'),
   SUBPARTITION q2_Europe VALUES ('France', 'Spain', 'Ireland')),
     PARTITION q3 VALUES LESS THAN (10)
  (SUBPARTITION q3_America VALUES
   ('United States of America', 'Argentina'),
   SUBPARTITION q3_Asia VALUES ('Japan', 'India'),
   SUBPARTITION q3_Europe VALUES ('France', 'Spain', 'Ireland')),
     PARTITION q4 VALUES LESS THAN (13)
  (SUBPARTITION q4_America VALUES
   ('United States of America', 'Argentina'),
   SUBPARTITION q4_Asia VALUES ('Japan', 'India'),
   SUBPARTITION q4_Europe VALUES ('France', 'Spain', 'Ireland')))
  AS SELECT t.calendar_year, t.calendar_month_number,
     t.day_number_in_month, c1.country_name, s.prod_id,
     p.prod_name, s.quantity_sold, s.amount_sold
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
  FROM times t, countries c1, products p, sales s, customers c2
  WHERE s.time_id = t.time_id AND s.prod_id = p.prod_id AND
        s.cust_id = c2.cust_id AND c2.country_id = c1.country_id AND
        c1.country_name IN ('United States of America', 'Argentina',
          'Japan', 'India', 'France', 'Spain', 'Ireland');
```

Then consider the following materialized view `sum_sales_per_year_month_mv`, which has the total amount of products sold each month of each year:

```
CREATE MATERIALIZED VIEW  sum_sales_per_year_month_mv
BUILD IMMEDIATE
REFRESH FORCE ON DEMAND
ENABLE QUERY REWRITE AS
SELECT s.calendar_year, s.calendar_month_number,
         SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_range_list s WHERE s.calendar_year > 1990
GROUP BY s.calendar_year, s.calendar_month_number;
```

`sales_per_country_mv` supports PCT against `sales_par_range_list` at the range partitioning level as its range partition key `calendar_month_number` is in its `SELECT` and `GROUP BY` list:

```
INSERT INTO sales_par_range_list
   VALUES (2001, 3, 25, 'Spain', 20, 'PROD20',  300,  20.50);
```

This statement inserts a row with `calendar_month_number = 3` and `country_name = 'Spain'`. This row is inserted into partition `q1` subpartition `Europe`. After this `INSERT` statement, `sum_sales_per_year_month_mv` is stale with respect to partition `q1` of `sales_par_range_list`. So any incoming query that accesses data from this partition in `sales_par_range_list` cannot be rewritten, for example, the following statement:

Note that the following query accesses data from partitions `q1` and `q2`. Because q1 was updated, the materialized view is stale with respect to `q1` so PCT rewrite is unavailable.

```
SELECT s.calendar_year, SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_range_list s
WHERE s.calendar_year = 2000
  AND s.calendar_month_number BETWEEN 2 AND 6
GROUP BY s.calendar_year;
```

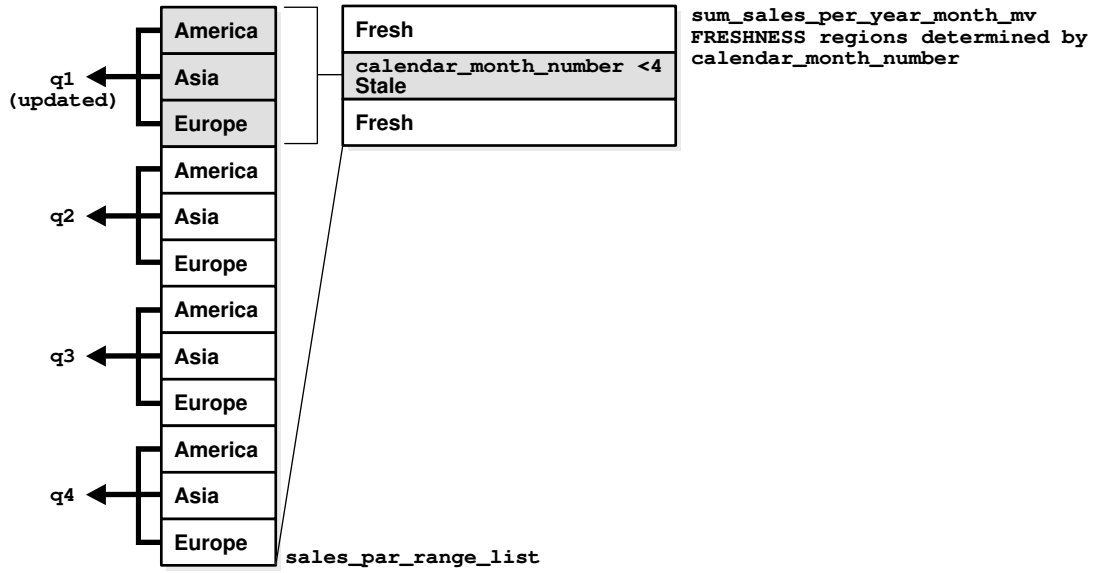
An example of a statement that does rewrite after the `INSERT` statement is the following, because it accesses fresh material:

```
SELECT s.calendar_year, SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_range_list s
WHERE s.calendar_year = 2000 AND s.calendar_month_number BETWEEN 5 AND 9
GROUP BY s.calendar_year;
```

Figure 12-3 offers a graphical illustration of what is stale and what is fresh.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Figure 12-3 PCT Rewrite and Range-List Partitioning**




## 12.2.7.3 PCT Rewrite Based on List Partitioned Tables

If the `LIST` partitioning key is present in the materialized view's `SELECT` and `GROUP BY`, then PCT will be supported by the materialized view. Regardless of the supported partitioning type, if the partition marker or rowid of the detail table is present in the materialized view then PCT is supported by the materialized view on that specific detail table.

```
CREATE TABLE sales_par_list
(calendar_year, calendar_month_number, day_number_in_month,
 country_name, prod_id, quantity_sold, amount_sold)
 PARTITION BY LIST (country_name)
 (PARTITION America
      VALUES ('United States of America', 'Argentina'),
  PARTITION Asia
      VALUES ('Japan', 'India'),
  PARTITION Europe
      VALUES ('France', 'Spain', 'Ireland'))
  AS SELECT t.calendar_year, t.calendar_month_number,
         t.day_number_in_month, c1.country_name, s.prod_id,
         s.quantity_sold, s.amount_sold
  FROM times t, countries c1, sales s, customers c2
  WHERE s.time_id = t.time_id and s.cust_id = c2.cust_id and
        c2.country_id = c1.country_id and
        c1.country_name IN ('United States of America', 'Argentina',
       'Japan', 'India', 'France', 'Spain', 'Ireland');
```

If a materialized view is created on the table `sales_par_list`, which has a list partitioning key, PCT rewrite will use that materialized view for potential rewrites.

To illustrate this feature, the following example creates a materialized view that has the total amounts sold of every product in each country for each year. The view depends on detail tables `sales_par_list` and `products`.

```
CREATE MATERIALIZED VIEW sales_per_country_mv
BUILD IMMEDIATE
REFRESH FORCE ON DEMAND
ENABLE QUERY REWRITE AS
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
SELECT s.calendar_year AS calendar_year, s.country_name AS country_name,
 p.prod_name AS prod_name, SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_list s, products p
WHERE s.prod_id = p.prod_id AND s.calendar_year <= 2000
GROUP BY s.calendar_year, s.country_name, prod_name;
```

sales_per_country_mv supports PCT against sales_par_list as its list partition key country_name is in its SELECT and GROUP BY list. Table products is not partitioned, so sales_per_country_mv does not support PCT against this table.

A query could be rewritten (in ENFORCED or TRUSTED modes) in terms of sales_per_country_mv even if sales_per_country_mv is stale if the incoming query accesses only fresh parts of the materialized view. You can determine which parts of the materialized view are FRESH only if the updated tables are PCT enabled in the materialized view. If non-PCT enabled tables have been updated, then the rewrite is not possible with fresh data from that specific materialized view as you cannot identify the FRESH portions of the materialized view.

sales_per_country_mv supports PCT on sales_par_list and does not support PCT on table product. If table products is updated, then PCT rewrite is not possible with sales_per_country_mv as you cannot tell which portions of the materialized view are FRESH.

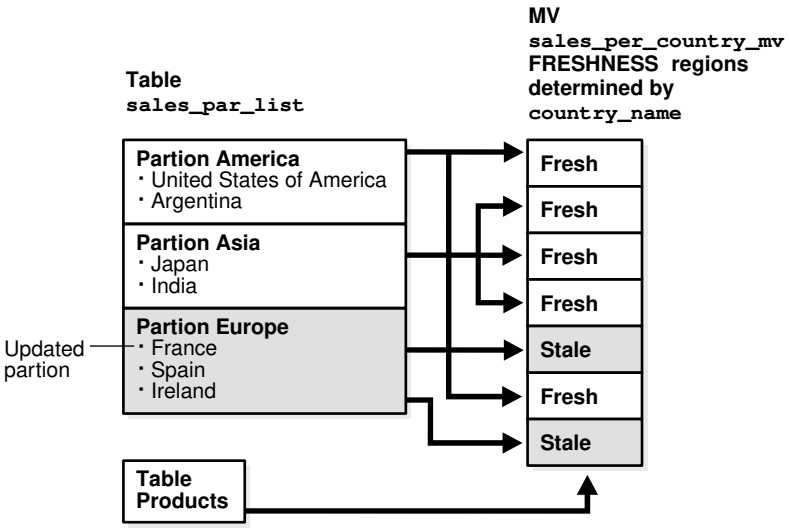
The following updates sales_par_list as follows:

```
INSERT INTO sales_par_list VALUES (2000, 10, 22, 'France', 900, 20, 200.99);
```

This statement inserted a row into partition Europe in table sales_par_list. Now sales_per_country_mv is stale, but PCT rewrite (in ENFORCED and TRUSTED modes) is possible as this materialized view supports PCT against table sales_par_list. The fresh and stale areas of the materialized view are identified based on the partitioned detail table sales_par_list.

Figure 12-4 illustrates what is fresh and what is stale in this example.

**Figure 12-4 PCT Rewrite and List Partitioning**




Consider the following query:

```
SELECT s.country_name, p.prod_name, SUM(s.amount_sold) AS sum_sales,
   COUNT(*) AS cnt
FROM sales_par_list s, products p
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
WHERE s.prod_id = p.prod_id AND s.calendar_year = 2000
  AND s.country_name IN ('United States of America', 'Japan')
GROUP BY s.country_name, p.prod_name;
```

This query accesses partitions America and Asia in sales_par_list; these partition have not been updated so rewrite is possible with stale materialized view sales_per_country_mv as this query will access only FRESH portions of the materialized view.

The query is rewritten in terms of sales_per_country_mv as follows:

```
SELECT country_name, prod_name, SUM(sum_sales) AS sum_sales, SUM(cnt) AS cnt
FROM sales_per_country_mv WHERE calendar_year = 2000
  AND country_name IN ('United States of America', 'Japan')
GROUP BY country_name, prod_name;
```

Now consider the following query:

```
SELECT s.country_name,  p.prod_name,
 SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_list s, products p
WHERE  s.prod_id = p.prod_id AND s.calendar_year = 1999
  AND s.country_name IN ('Japan', 'India', 'Spain')
GROUP BY s.country_name, p.prod_name;
```

This query accesses partitions Europe and Asia in sales_par_list. Partition Europe has been updated, so this query cannot be rewritten in terms of sales_per_country_mv as the required data from the materialized view is stale.

You will be able to rewrite after any kinds of updates to sales_par_list, that is DMLs, direct loads and Partition Maintenance Operations (PMOPs) if the incoming query accesses FRESH parts of the materialized view.

## 12.2.7.4 PCT Rewrite and PMARKER

When a partition marker is provided, the query rewrite capabilities are limited to rewrite queries that access whole detail table partitions as all rows from a specific partition have the same pmarker value. That is, if a query accesses a portion of a detail table partition, it is not rewritten even if that data corresponds to a FRESH portion of the materialized view. Now FRESH portions of the materialized view are determined by the pmarker value. To determine which rows of the materialized view are fresh, you associate freshness with the marker value, so all rows in the materialized view with a specific pmarker value are FRESH or are STALE.

The following creates a materialized view has the total amounts sold of every product in each detail table partition of sales_par_list for each year. This materialized view will also depend on detail table products as shown in the following:

```
CREATE MATERIALIZED VIEW sales_per_dt_partition_mv
BUILD IMMEDIATE
REFRESH FORCE ON DEMAND
ENABLE QUERY REWRITE AS
SELECT s.calendar_year AS calendar_year, p.prod_name AS prod_name,
       DBMS_MVIEW.PMARKER(s.rowid) pmarker,
       SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_list s, products p
WHERE s.prod_id = p.prod_id AND s.calendar_year > 2000
GROUP BY s.calendar_year, DBMS_MVIEW.PMARKER(s.rowid), p.prod_name;
```

The materialized view sales_per_dt_partition_mv provides the sum of sales for each detail table partition. This materialized view supports PCT rewrite against table sales_par_list


Copyright © 2001, 2025, Oracle and/or its affiliates.

because the partition marker is in its SELECT and GROUP BY clauses. Table 12-2 lists the partition names and their pmarkers for this example.

**Table 12-2 Partition Names and Their Pmarkers**

| Partition Name | Pmarker |
|---|---|
| America | 1000 |
| Asia | 1001 |
| Europe | 1002 |

Then update the table sales_par_list as follows:

```
DELETE FROM sales_par_list WHERE country_name = 'India';
```

You have deleted rows from partition Asia in table sales_par_list. Now sales_per_dt_partition_mv is stale, but PCT rewrite (in ENFORCED and TRUSTED modes) is possible as this materialized view supports PCT (pmarker based) against table sales_par_list.

Now consider the following query:

```
SELECT p.prod_name, SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_list s, products p
WHERE  s.prod_id = p.prod_id AND s.calendar_year = 2001 AND
       s.country_name IN ('United States of America', 'Argentina')
GROUP BY p.prod_name;
```

This query can be rewritten in terms of sales_per_dt_partition_mv as all the data corresponding to a detail table partition is accessed, and the materialized view is FRESH with respect to this data. This query accesses all data in partition America, which has not been updated.

The query is rewritten in terms of sales_per_dt_partition_mv as follows:

```
SELECT prod_name, SUM(sum_sales) AS sum_sales, SUM(cnt) AS cnt
FROM sales_per_dt_partition_mv
WHERE calendar_year = 2001 AND pmarker = 1000
GROUP BY prod_name;
```

## 12.2.7.5 PCT Rewrite Using Rowid as PMARKER

A materialized view supports PCT rewrite provided a partition key or a partition marker is provided in its SELECT and GROUP BY clause, if there is a GROUP BY clause. You can use the rowids of the partitioned table instead of the pmarker or the partition key. Note that Oracle converts the rowids into pmarkers internally. Consider the following table:

```
CREATE TABLE product_par_list
(prod_id, prod_name, prod_category,
 prod_subcategory, prod_list_price)
 PARTITION BY LIST (prod_category)
 (PARTITION prod_cat1
      VALUES ('Boys', 'Men'),
  PARTITION prod_cat2
      VALUES ('Girls', 'Women'))
 AS
   SELECT prod_id, prod_name, prod_category,
      prod_subcategory, prod_list_price
   FROM products;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

Let us create the following materialized view on tables, sales_par_list and product_par_list:

```
CREATE MATERIALIZED VIEW  sum_sales_per_category_mv
BUILD IMMEDIATE
REFRESH FORCE ON DEMAND
ENABLE QUERY REWRITE AS
SELECT p.rowid prid, p.prod_category,
       SUM (s.amount_sold) sum_sales, COUNT(*) cnt
FROM sales_par_list s, product_par_list p
WHERE s.prod_id = p.prod_id and s.calendar_year <= 2000
GROUP BY p.rowid, p.prod_category;
```

All the limitations that apply to pmarker rewrite apply here as well. The incoming query should access a whole partition for the query to be rewritten. The following pmarker table is used in this case:

```
product_par_list       pmarker value
---------------        -------------
prod_cat1                     1000
prod_cat2                     1001
prod_cat3                     1002
```

Then update table product_par_list as follows:

```
DELETE FROM product_par_list WHERE prod_name = 'MEN';
```

So sum_sales_per_category_mv is stale with respect to partition prod_list1 from product_par_list.

Now consider the following query:

```
SELECT p.prod_category, SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_list s, product_par_list p
WHERE  s.prod_id = p.prod_id AND p.prod_category IN
       ('Girls', 'Women') AND s.calendar_year <= 2000
GROUP BY p.prod_category;
```

This query can be rewritten in terms of sum_sales_per_category_mv as all the data corresponding to a detail table partition is accessed, and the materialized view is FRESH with respect to this data. This query accesses all data in partition prod_cat2, which has not been updated. Following is the rewritten query in terms of sum_sales_per_category_mv:

```
SELECT prod_category, sum_sales, cnt
FROM sum_sales_per_category_mv WHERE DBMS_MVIEW.PMARKER(srid) IN (1000)
GROUP BY prod_category;
```

## 12.2.8 About Query Rewrite Using Multiple Materialized Views

Query rewrite has been extended to enable the rewrite of a query using multiple materialized views. If query rewrite determines that there is no set of materialized views that returns all of the data, then query rewrite retrieves the remaining data from the base tables.

Query rewrite using multiple materialized views can take advantage of many different types and combinations of rewrite, such as using PCT and IN-lists. The following examples illustrate some of the queries where query rewrite is now possible.

Consider the following two materialized views, cust_avg_credit_mv1 and cust_avg_credit_mv2. cust_avg_credit_mv1 asks for all customers average credit limit for each postal code that were born between the years 1940 and 1950. cust_avg_credit_mv2


Copyright © 2001, 2025, Oracle and/or its affiliates.

asks for customers average credit limit for each postal code that were born after 1950 and before or on 1970.

The materialized views' definitions for this example are as follows:

```
CREATE MATERIALIZED VIEW cust_avg_credit_mv1
ENABLE QUERY REWRITE
AS SELECT cust_postal_code, cust_year_of_birth,
       SUM(cust_credit_limit) AS sum_credit,
       COUNT(cust_credit_limit) AS count_credit
FROM customers
WHERE cust_year_of_birth BETWEEN 1940 AND 1950
GROUP BY cust_postal_code, cust_year_of_birth;

CREATE MATERIALIZED VIEW cust_avg_credit_mv2
ENABLE QUERY REWRITE
AS SELECT cust_postal_code, cust_year_of_birth,
       SUM(cust_credit_limit) AS sum_credit,
       COUNT(cust_credit_limit) AS count_credit
FROM customers
WHERE cust_year_of_birth > 1950 AND cust_year_of_birth <= 1970
GROUP BY cust_postal_code, cust_year_of_birth;
```

### Query 1: One Matched Interval in Materialized View and Query

Consider a query that asks for all customers average credit limit for each postal code who were born between 1940 and 1970. This query is matched by the interval `BETWEEN` on `cust_year_of_birth`.

```
SELECT cust_postal_code, AVG(cust_credit_limit) AS avg_credit
FROM customers c
WHERE cust_year_of_birth BETWEEN 1940 AND 1970
GROUP BY cust_postal_code;
```

The preceding query can be rewritten in terms of these two materialized views to get all the data as follows:

```
SELECT v1.cust_postal_code,
SUM(v1.sum_credit)/SUM(v1.count_credit) AS avg_credit
FROM (SELECT cust_postal_code, sum_credit, count_credit
 FROM cust_avg_credit_mv1
 GROUP BY cust_postal_code
 UNION ALL
 SELECT cust_postal_code, sum_credit, count_credit
 FROM cust_avg_credit_mv2
 GROUP BY cust_postal_code) v1
 GROUP BY v1.cust_postal_code;
```

Note that the `UNION ALL` query is used in an inline view because of the re-aggregation that needs to take place. Note also how query rewrite was the count aggregate to perform this rollup.

### Query 2: Query Outside of Data Contained in Materialized View

When the materialized view goes beyond the range asked by the query, a filter (also called selection) is added to the rewritten query to drop out the unneeded rows returned by the materialized view. This case is illustrated in the following query:

```
SELECT cust_postal_code, SUM(cust_credit_limit) AS sum_credit
FROM customers c
WHERE cust_year_of_birth BETWEEN 1945 AND 1955
GROUP BY cust_postal_code;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

Query 2 is rewritten as:

```
SELECT v1.cust_postal_code, SUM(v1.sum_credit)
FROM
(SELECT cust_postal_code, SUM(sum_credit) AS sum_credit
FROM cust_avg_credit_mv1
WHERE cust_year_of_birth BETWEEN 1945 AND 1950
GROUP BY cust_postal_code
UNION ALL
SELECT cust_postal_code, SUM(sum_credit) AS sum_credit
FROM cust_birth_mv2
WHERE cust_year_of_birth > 1950 AND cust_year_of_birth <= 1955
GROUP BY cust_postal_code) v1
GROUP BY v1.cust_postal_code;
```

### Query 3: Requesting More Data Than is in the Materialized View

What if a query asks for more data than is contained in the two materialized views? It still rewrites using both materialized views and the data in the base table. In the following example, a new set of materialized views without aggregates is defined It will still rewrite using both materialized views and the data in the base table.

```
CREATE MATERIALIZED VIEW cust_birth_mv1
ENABLE QUERY REWRITE
AS SELECT cust_last_name, cust_first_name, cust_year_of_birth
FROM customers WHERE cust_year_of_birth BETWEEN 1940 AND 1950;

CREATE MATERIALIZED VIEW cust_avg_credit_mv2
ENABLE QUERY REWRITE
AS SELECT cust_last_name, cust_first_name, cust_year_of_birth
FROM customers
WHERE cust_year_of_birth > 1950 AND cust_year_of_birth <= 1970;
```

Our queries now require all customers born between 1940 and 1990.

```
SELECT cust_last_name, cust_first_name
FROM customers c WHERE cust_year_of_birth BETWEEN 1940 AND 1990;
```

Query rewrite needs to access the base table to access the customers that were born after 1970 and before or on 1990. Therefore, Query 3 is rewritten as the following:

```
SELECT cust_last_name, cust_first_name
FROM cust_birth_mv1
UNION ALL
SELECT cust_last_name, cust_first_name
FROM cust_birth_mv2
UNION ALL
SELECT cust_last_name, cust_first_name
FROM customers c
WHERE cust_year_of_birth > 1970 AND cust_year_of_birth <= 1990;
```

### Query 4: Requesting Data on Multiple Selection Columns

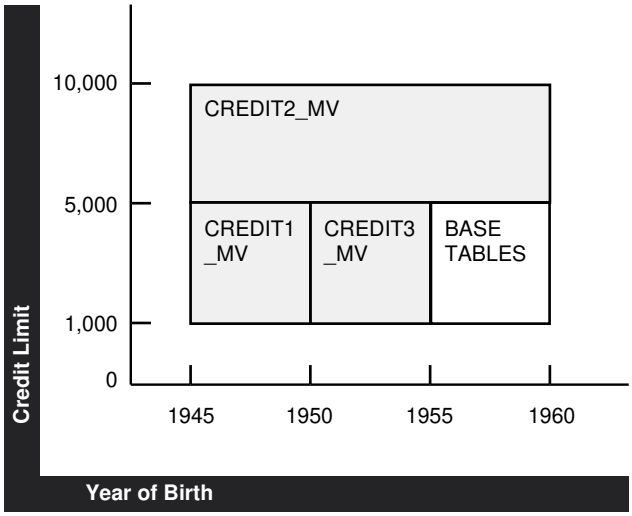
Consider the following query, which asks for all customers who have a credit limit between 1,000 and 10,000 and were born between the years 1945 and 1960. This query is a multi-selection query because it is asking for data on multiple selection columns.

```
SELECT cust_last_name, cust_first_name
FROM customers WHERE cust_year_of_birth BETWEEN 1945 AND 1960 AND
   cust_credit_limit BETWEEN 1000 AND 10000;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

Figure 12-5 shows a two-selection query, which can be rewritten with the two-selection materialized views described in the following section.

**Figure 12-5 Query Rewrite Using Multiple Materialized Views**




The graph in Figure 12-5 illustrates the materialized views that can be used to satisfy this query. `credit_mv1` asks for customers that have credit limits between 1,000 and 5,000 and were born between 1945 and 1950. `credit_mv2` asks for customers that have credit limits > 5,000 and <= 10,000 and were born between 1945 and 1960. `credit_mv3` asks for customers that have credit limits between 1,000 and 5,000 and were born after 1950 and before or on 1955.

The materialized views' definitions for this case are as follows:

```
CREATE MATERIALIZED VIEW credit_mv1
ENABLE QUERY REWRITE
AS SELECT cust_last_name, cust_first_name,
    cust_credit_limit, cust_year_of_birth
FROM customers
WHERE cust_credit_limit BETWEEN 1000 AND 5000
AND cust_year_of_birth BETWEEN 1945 AND 1950;

CREATE MATERIALIZED VIEW credit_mv2
ENABLE QUERY REWRITE
AS SELECT cust_last_name, cust_first_name,
   cust_credit_limit, cust_year_of_birth
FROM customers
WHERE cust_credit_limit > 5000
   AND cust_credit_limit <= 10000 AND cust_year_of_birth
   BETWEEN 1945 AND 1960;

CREATE MATERIALIZED VIEW credit_mv3
ENABLE QUERY REWRITE AS
SELECT cust_last_name, cust_first_name,
   cust_credit_limit, cust_year_of_birth
FROM customers
WHERE cust_credit_limit BETWEEN 1000 AND 5000
  AND cust_year_of_birth > 1950 AND cust_year_of_birth <= 1955;
```

Query 4 can be rewritten by using all three materialized views to access most of the data. However, because not all the data can be obtained from these three materialized views, query


Copyright © 2001, 2025, Oracle and/or its affiliates.

rewrite also accesses the base tables to retrieve the data for customers who have credit limits between 1,000 and 5,000 and were born between 1955 and 1960. It is rewritten as follows:

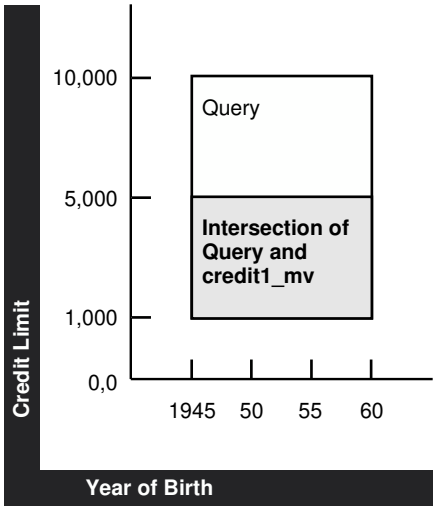
```
SELECT cust_last_name, cust_first_name
FROM credit_mv1
UNION ALL
SELECT cust_last_name, cust_first_name
FROM credit_mv2
UNION ALL
SELECT cust_last_name, cust_first_name
FROM credit_mv3
UNION ALL
SELECT cust_last_name, cust_first_name
FROM customers
WHERE cust_credit_limit BETWEEN 1000 AND 5000
  AND cust_year_of_birth > 1955 AND cust_year_of_birth <= 1960;
```

This example illustrates how a multi-selection query can be rewritten with multiple materialized views. The example was simplified to show no overlapping data among the three materialized views. However, query rewrite can perform similar rewrites.

### Query 5: Intervals and Constrained Intervals

This example illustrates how a multi-selection query can be rewritten using a single selection materialized view. In this example, there are two intervals in the query and one constrained interval in the materialized view. It asks for customers that have credit limits between 1,000 and 10,000 and were born between 1945 and 1960. But suppose that `credit_mv1` asks for just customers that have credit limits between 1,000 and 5,000. `credit_mv1` is not constrained by a selection in `cust_year_of_birth`, therefore covering the entire range of birth year values for the query.

**Figure 12-6 Constrained Materialized View Selections**




The area between the lines in Figure 12-6 represents the data `credit1_mv`.

The new `credit_mv1` is defined as follows:

```
CREATE MATERIALIZED VIEW credit_mv1
ENABLE QUERY REWRITE
AS SELECT cust_last_name, cust_first_name,
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
   cust_credit_limit, cust_year_of_birth
FROM customers WHERE cust_credit_limit BETWEEN 1000 AND 5000;
```

The query is as follows:

```
SELECT cust_last_name, cust_first_name
FROM customers WHERE cust_year_of_birth BETWEEN 1945 AND 1960
  AND cust_credit_limit BETWEEN 1000 AND 10000;
```

And finally the rewritten query is as follows:

```
SELECT cust_last_name, cust_first_name
FROM credit_mv1 WHERE cust_year_of_birth BETWEEN 1945 AND 1960
UNION ALL
SELECT cust_last_name, cust_first_name
FROM customers WHERE cust_year_of_brith BETWEEN 1945 AND 1960
  AND cust_credit_limit > 5000 AND cust_credit_limit <= 10000;
```

### Query 6: Query has Single Column IN-List and Materialized Views have Single Column Intervals

Multiple materialized view query rewrite can process an `IN`-list in the incoming query and rewrite the query in terms of materialized views that have intervals on the same selection column. Given that an `IN`-list represents discrete values in an interval, this rewrite capability is a natural extension to the intervals only scenario described earlier.

The following is an example of a one column `IN`-list selection in the query and one column interval selection in the materialized views. Consider a query that asks for the number of customers for each country who were born in any of the following year: 1945, 1950, 1955, 1960, 1965, 1970 or 1975. This query is constrained by an `IN`-list on `cust_year_of_birth`.

```
SELECT c2.country_name, count(c1.country_id)
FROM customers c1, countries c2
WHERE c1.country_id = c2.country_id AND
   c1.cust_year_of_birth IN (1945, 1950, 1955, 1960, 1965, 1970, 1975)
GROUP BY c2.country_name;
```

Consider the following two materialized views. `cust_country_birth_mv1` asks for the number of customers for each country that were born between the years 1940 and 1950. `cust_country_birth_mv2` asks for the number of customers for each country that were born after 1950 and before or on 1970. The preceding query can be rewritten in terms of these two materialized views to get the total number of customers for each country born in 1945, 1950, 1955, 1960, 1965 and 1970. The base table access is required to obtain the number of customers that were born in 1975.

The materialized views' definitions for this example are as follows:

```
CREATE MATERIALIZED VIEW cust_country_birth_mv1
ENABLE QUERY REWRITE
AS SELECT c2.country_name, c1.cust_year_of_birth,
 COUNT(c1.country_id) AS count_customers
FROM customers c1, countries c2
WHERE c1.country_id = c2.country_id AND
      cust_year_of_birth BETWEEN 1940 AND 1950
GROUP BY c2.country_name, c1.cust_year_of_birth;

CREATE MATERIALIZED VIEW cust_country_birth_mv2
ENABLE QUERY REWRITE
AS SELECT c2.country_name, c1.cust_year_of_birth,
 COUNT(c1.country_id) AS count_customers
FROM customers c1, countries c2
WHERE c1.country_id = c2.country_id AND cust_year_of_birth > 1950
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
AND cust_year_of_birth <= 1970
GROUP BY c2.country_name, c1.cust_year_of_birth;
```

So, Query 6 is rewritten as:

```
SELECT v1.country_name, SUM(v1.count_customers)
FROM (SELECT country_name, SUM(count_customers) AS count_customers
FROM cust_country_birth_mv1
WHERE cust_year_of_birth IN (1945, 1950)
GROUP BY country_name
UNION ALL
SELECT country_name, SUM(count_customers) AS count_customers
FROM cust_country_birth_mv2
WHERE cust_year_of_birth IN (1955, 1960, 1965, 1970)
GROUP BY country_name
UNION ALL
SELECT c2.country_name, COUNT(c1.country_id) AS count_customers
FROM customers c1, countries c2
WHERE c1.country_id = c2.country_id AND cust_year_of_birth IN (1975)
GROUP BY c2.country_name) v1
GROUP BY v1.country_name;
```

### Query 7: PCT Rewrite with Multiple Materialized Views

Rewrite with multiple materialized views can also take advantage of PCT rewrite. PCT rewrite refers to the capability of rewriting a query with only the fresh portions of a materialized view when the materialized view is stale. This feature is used in `ENFORCED` or `TRUSTED` integrity modes, and with multiple materialized view rewrite, it can use the fresh portions of the materialized view to get the fresh data from it, and go to the base table to get the stale data. So the rewritten query will `UNION ALL` only the fresh data from one or more materialized views and obtain the rest of the data from the base tables to answer the query. Therefore, all the PCT rules and conditions apply here as well. The materialized view should be PCT enabled and the changes made to the base table should be such that the fresh and stale portions of the materialized view can be clearly identified.

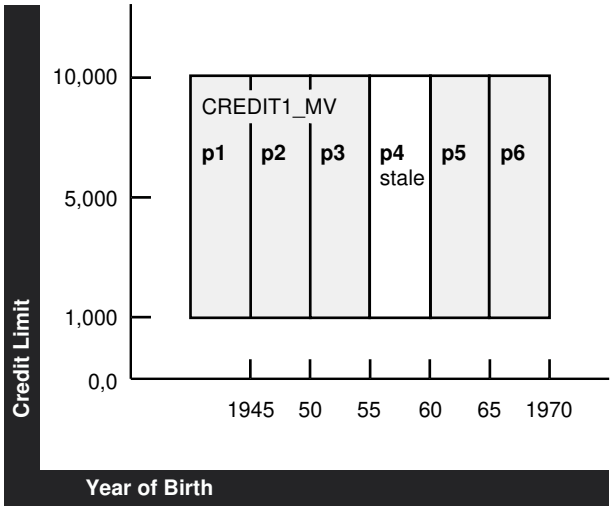
This example assumes you have a query that asks for customers who have credit limits between 1,000 and 10,000 and were born between 1945 and 1964. Also, the customer table is partitioned by `cust_date_of_birth` and there is a PCT-enabled materialized view called `credit_mv1` that also asks for customers who have a credit limit between 1,000 and 10,000 and were born between 1945 and 1964.

```
SELECT cust_last_name, cust_first_name
FROM customers WHERE cust_credit_limit BETWEEN 1000 AND 10000;
```

In Figure 12-7, the diagram illustrates those regions of the materialized view that are fresh (dark) and stale (light) with respect to the base table partitions `p1-p6`.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Figure 12-7 PCT and Multiple Materialized View Rewrite**




Let us say that you are in `ENFORCED` mode and that `p1`, `p2`, `p3`, `p5`, and `p6` of the customer table are fresh and partition `p4` is stale. This means that all partitions of `credit_mv1` cannot be used to answer the query. The rewritten query must get the results for customer partition `p4` from some other materialized view or as shown in this example, from the base table. Below, you can see part of the table definition for the `customers` table showing how the table is partitioned:

```
CREATE TABLE customers
(PARTITION BY RANGE (cust_year_of_birth)
 PARTITION p1 VALUES LESS THAN (1945),
 PARTITION p2 VALUES LESS THAN (1950),
 PARTITION p3 VALUES LESS THAN (1955),
 PARTITION p4 VALUES LESS THAN (1960),
 PARTITION p5 VALUES LESS THAN (1965),
 PARTITION p6 VALUES LESS THAN (1970);
```

The materialized view definition for the preceding example is as follows:

```
CREATE MATERIALIZED VIEW credit_mv1
ENABLE QUERY REWRITE
AS SELECT cust_last_name, cust_first_name,
   cust_credit_limit, cust_year_of_birth
FROM customers
WHERE cust_credit_limit BETWEEN 1000 AND 10000
AND cust_year_of_birth BETWEEN 1945 AND 1964;
```

Note that this materialized view is PCT enabled with respect to table `customers`.

The rewritten query is as follows:

```
SELECT cust_last_name, cust_first_name FROM credit_mv1
WHERE cust_credit_limit BETWEEN 1000 AND 10000 AND
   (cust_year_of_birth >= 1945 AND cust_year_of_birth < 1955 OR
    cust_year_of_birth BETWEEN 1945 AND 1964)
UNION ALL
SELECT cust_last_name, cust_first_name
FROM customers WHERE cust_credit_limit BETWEEN 1000 AND 10000
  AND cust_year_of_birth < 1960 AND cust_year_of_birth >= 1955;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

# 12.3 Other Query Rewrite Considerations

The following topics discusses some of the other cases when query rewrite is possible:


- About Query Rewrite Using Nested Materialized Views
- About Query Rewrite in the Presence of Inline Views
- About Query Rewrite Using Remote Tables
- About Query Rewrite in the Presence of Duplicate Tables
- About Query Rewrite Using Date Folding
- About Query Rewrite Using View Constraints
- Query Rewrite Using Set Operator Materialized Views
- About Query Rewrite in the Presence of Grouping Sets
- Query Rewrite in the Presence of Window Functions
- Query Rewrite and Expression Matching
- Cursor Sharing and Bind Variables During Query Rewrite
- Handling Expressions in Query Rewrite


## 12.3.1 About Query Rewrite Using Nested Materialized Views

Query rewrite attempts to iteratively take advantage of nested materialized views. Oracle AI Database first tries to rewrite a query with materialized views having aggregates and joins, then with a materialized view containing only joins. If any of the rewrites succeeds, Oracle repeats that process again until no rewrites are found. For example, assume that you had created materialized views `join_sales_time_product_mv` and `sum_sales_time_product_mv` as in the following:

```
CREATE MATERIALIZED VIEW join_sales_time_product_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_id, p.prod_name, t.time_id, t.week_ending_day,
       s.channel_id, s.promo_id, s.cust_id, s.amount_sold
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id AND s.prod_id = p.prod_id;

CREATE MATERIALIZED VIEW sum_sales_time_product_mv
ENABLE QUERY REWRITE AS
SELECT mv.prod_name, mv.week_ending_day, COUNT(*) cnt_all,
       SUM(mv.amount_sold) sum_amount_sold,
       COUNT(mv.amount_sold) cnt_amount_sold
FROM join_sales_time_product_mv mv
GROUP BY mv.prod_name, mv.week_ending_day;
```

Then consider the following query:

```
SELECT p.prod_name, t.week_ending_day, SUM(s.amount_sold)
FROM sales s, products p, times t
WHERE s.time_id=t.time_id AND s.prod_id=p.prod_id
GROUP BY p.prod_name, t.week_ending_day;
```

Oracle finds that `join_sales_time_product_mv` is eligible for rewrite. The rewritten query has this form:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
SELECT mv.prod_name, mv.week_ending_day, SUM(mv.amount_sold)
FROM join_sales_time_product_mv mv
GROUP BY mv.prod_name, mv.week_ending_day;
```

Because a rewrite occurred, Oracle tries the process again. This time, the query can be rewritten with single-table aggregate materialized view `sum_sales_store_time` into the following form:

```
SELECT mv.prod_name, mv.week_ending_day, mv.sum_amount_sold
FROM sum_sales_time_product_mv mv;
```

## 12.3.2 About Query Rewrite in the Presence of Inline Views

Oracle AI Database supports query rewrite with inline views in two ways:

- when the text from the inline views in the materialized view exactly matches the text in the request query
- when the request query contains inline views that are equivalent to the inline views in the materialized view

Two inline views are considered equivalent if their `SELECT` lists and `GROUP BY` lists are equivalent, `FROM` clauses contain the same or equivalent objects, their join graphs, including all the selections in the `WHERE` clauses are equivalent and their `HAVING` clauses are equivalent.

The following examples illustrate how a query with an inline view can rewrite with a materialized view using text match and general inline view rewrites. Consider the following materialized view that contains an inline view:

```
CREATE MATERIALIZED VIEW SUM_SALES_MV
ENABLE QUERY REWRITE AS
SELECT mv_iv.prod_id, mv_iv.cust_id,
sum(mv_iv.amount_sold) sum_amount_sold
FROM (SELECT prod_id, cust_id, amount_sold
FROM sales, products
WHERE sales.prod_id = products.prod_id) MV_IV
GROUP BY mv_iv.prod_id, mv_iv.cust_id;
```

The following query has an inline view whose text matches exactly with that of the materialized view's inline view. Hence, the query inline view is internally replaced with the materialized view's inline view so that the query can be rewritten:

```
SELECT iv.prod_id, iv.cust_id,
SUM(iv.amount_sold) sum_amount_sold
FROM (SELECT prod_id, cust_id, amount_sold
FROM sales, products
WHERE sales.prod_id = products.prod_id) IV
GROUP BY iv.prod_id, iv.cust_id;
```

The following query has an inline view that does not have exact text match with the inline view in the preceding materialized view. Note that the join predicate in the query inline view is switched. Even though this query does not textually match with that of the materialized view's inline view, query rewrite identifies the query's inline view as equivalent to the materialized view's inline view. As before, the query inline view will be internally replaced with the materialized view's inline view so that the query can be rewritten.

```
SELECT iv.prod_id, iv.cust_id,
SUM(iv.amount_sold) sum_amount_sold
FROM (SELECT prod_id, cust_id, amount_sold
FROM sales, products
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
WHERE products.prod_id = sales.prod_id) IV
GROUP BY iv.prod_id, iv.cust_id;
```

Both of these queries are rewritten with `SUM_SALES_MV` as follows:

```
SELECT prod_id, cust_id, sum_amount_sold
FROM SUM_SALES_MV;
```

General inline view rewrite is not supported for queries that contain set operators, `GROUPING SET` clauses, nested subqueries, nested inline views, and remote tables.

## 12.3.3 About Query Rewrite Using Remote Tables

Oracle AI Database supports query rewrite with materialized views that reference tables at a single remote database site. Note that the materialized view should be present at the site where the query is being issued. Because any remote table update cannot be propagated to the local site simultaneously, query rewrite only works in the `stale_tolerated` mode. Whenever a query contains columns that are not found in the materialized view, it uses a technique called join back to rewrite the query. However, if the join back table is not found at the local site, query rewrite does not take place. Also, because the constraint information of the remote tables is not available at the remote site, query rewrite does not make use of any constraint information.

The following query contains tables that are found at a single remote site:

```
SELECT p.prod_id, t.week_ending_day, s.cust_id,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales@remotedbl s, products@remotedbl p, times@remotedbl t
WHERE s.time_id=t.time_id AND s.prod_id=p.prod_id
GROUP BY p.prod_id, t.week_ending_day, s.cust_id;
```

The following materialized view is present at the local site, but it references tables that are all found at the remote site:

```
CREATE MATERIALIZED VIEW sum_sales_prod_week_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_id, t.week_ending_day, s.cust_id,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales@remotedbl s, products@remotedbl p, times@remotedbl t
WHERE s.time_id=t.time_id AND s.prod_id=p.prod_id
GROUP BY p.prod_id, t.week_ending_day, s.cust_id;
```

Even though the query references remote tables, it is rewritten using the previous materialized view as follows:

```
SELECT prod_id, week_ending_day, cust_id,  sum_amount_sold
FROM sum_sales_prod_week_mv;
```

## 12.3.4 About Query Rewrite in the Presence of Duplicate Tables

Oracle AI Database accomplishes query rewrite of queries that contain multiple references to the same tables, or self joins by employing two different strategies. Using the first strategy, you need to ensure that the query and the materialized view definitions have the same aliases for the multiple references to a table. If you do not provide a matching alias, Oracle tries the second strategy, where the joins in the query and the materialized view are compared to match the multiple references in the query to the multiple references in the materialized view.


Copyright © 2001, 2025, Oracle and/or its affiliates.

The following is an example of a materialized view and a query. In this example, the query is missing a reference to a column in a table so an exact text match does not work. General query rewrite can occur, however, because the aliases for the table references match.

To demonstrate the self-join rewriting possibility with the sh sample schema, the following addition is assumed to include the actual shipping and payment date in the fact table, referencing the same dimension table times. This is for demonstration purposes only and does not return any results:

```
ALTER TABLE sales ADD (time_id_ship DATE);
ALTER TABLE sales ADD (CONSTRAINT time_id_book_fk FOREIGN key (time_id_ship)
 REFERENCES times(time_id) ENABLE NOVALIDATE);
ALTER TABLE sales MODIFY CONSTRAINT time_id_book_fk RELY;
ALTER TABLE sales ADD (time_id_paid DATE);
ALTER TABLE sales ADD (CONSTRAINT time_id_paid_fk FOREIGN KEY (time_id_paid)
 REFERENCES times(time_id) ENABLE NOVALIDATE);
ALTER TABLE sales MODIFY CONSTRAINT time_id_paid_fk RELY;
```

Now, you can define a materialized view as follows:

```
CREATE MATERIALIZED VIEW sales_shipping_lag_mv
ENABLE QUERY REWRITE AS
SELECT t1.fiscal_week_number, s.prod_id,
         t2.fiscal_week_number - t1.fiscal_week_number AS lag
FROM times t1, sales s, times t2
WHERE t1.time_id = s.time_id AND t2.time_id = s.time_id_ship;
```

The following query fails the exact text match test but is rewritten because the aliases for the table references match:

```
SELECT s.prod_id, t2.fiscal_week_number - t1.fiscal_week_number AS lag
FROM times t1, sales s, times t2
WHERE t1.time_id = s.time_id AND t2.time_id = s.time_id_ship;
```

Note that Oracle AI Database performs other checks to ensure the correct match of an instance of a multiply instanced table in the request query with the corresponding table instance in the materialized view. For instance, in the following example, Oracle correctly determines that the matching alias names used for the multiple instances of table `times` does not establish a match between the multiple instances of table `times` in the materialized view.

The following query cannot be rewritten using `sales_shipping_lag_mv`, even though the alias names of the multiply instanced table `time` match because the joins are not compatible between the instances of `time` aliased by `t2`:

```
SELECT s.prod_id, t2.fiscal_week_number - t1.fiscal_week_number AS lag
FROM times t1, sales s, times t2
WHERE t1.time_id = s.time_id AND t2.time_id = s.time_id_paid;
```

This request query joins the instance of the `time` table aliased by `t2` on the `s.time_id_paid` column, while the materialized views joins the instance of the `times` table aliased by `t2` on the `s.time_id_ship` column. Because the join conditions differ, Oracle correctly determines that rewrite cannot occur.

The following query does not have any matching alias in the materialized view, `sales_shipping_lag_mv`, for the table, times. But query rewrite now compares the joins between the query and the materialized view and correctly match the multiple instances of times.

```
SELECT s.prod_id, x2.fiscal_week_number - x1.fiscal_week_number AS lag
FROM times x1, sales s, times x2
WHERE x1.time_id = s.time_id AND x2.time_id = s.time_id_ship;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 12.3.5 About Query Rewrite Using Date Folding

Date folding rewrite is a specific form of expression matching rewrite. In this type of rewrite, a date range in a query is folded into an equivalent date range representing higher date granules. The resulting expressions representing higher date granules in the folded date range are matched with equivalent expressions in a materialized view. The folding of date range into higher date granules such as months, quarters, or years is done when the underlying data type of the column is an Oracle `DATE`. The expression matching is done based on the use of canonical forms for the expressions.

`DATE` is a built-in data type which represents ordered time units such as seconds, days, and months, and incorporates a time hierarchy (second -> minute -> hour -> day -> month -> quarter -> year). This hard-coded knowledge about `DATE` is used in folding date ranges from lower-date granules to higher-date granules. Specifically, folding a date value to the beginning of a month, quarter, year, or to the end of a month, quarter, year is supported. For example, the date value `1-jan-1999` can be folded into the beginning of either year `1999` or quarter `1999-1` or month `1999-01`. And, the date value `30-sep-1999` can be folded into the end of either quarter `1999-03` or month `1999-09`.

> **Note**
>
> Due to the way date folding works, you should be careful when using `BETWEEN` and date columns. The best way to use `BETWEEN` and date columns is to increment the later date by 1. In other words, instead of using `date_col BETWEEN '1-jan-1999' AND '30-jun-1999'`, you should use `date_col BETWEEN '1-jan-1999' AND '1-jul-1999'`. You could also use the `TRUNC` function to get the equivalent result, as in `TRUNC(date_col) BETWEEN '1-jan-1999' AND '30-jun-1999'`. `TRUNC` will, however, strip time values.

Because date values are ordered, any range predicate specified on date columns can be folded from lower level granules into higher level granules provided the date range represents an integral number of higher level granules. For example, the range predicate `date_col >= '1-jan-1999' AND date_col < '30-jun-1999'` can be folded into either a month range or a quarter range using the `TO_CHAR` function, which extracts specific date components from a date value.

The advantage of aggregating data by folded date values is the compression of data achieved. Without date folding, the data is aggregated at the lowest granularity level, resulting in increased disk space for storage and increased I/O to scan the materialized view.

Consider a query that asks for the sum of sales by product types for the year 1998:

```
SELECT p.prod_category, SUM(s.amount_sold)
FROM sales s, products p
WHERE s.prod_id=p.prod_id AND s.time_id >= TO_DATE('01-jan-1998', 'dd-mon-yyyy')
  AND s.time_id <   TO_DATE('01-jan-1999', 'dd-mon-yyyy')
GROUP BY p.prod_category;

CREATE MATERIALIZED VIEW sum_sales_pcat_monthly_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_category, TO_CHAR(s.time_id,'YYYY-MM') AS month,
       SUM(s.amount_sold) AS sum_amount
FROM sales s, products p
WHERE s.prod_id=p.prod_id
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
GROUP BY p.prod_category, TO_CHAR(s.time_id, 'YYYY-MM');

SELECT p.prod_category, SUM(s.amount_sold)
FROM sales s, products p
WHERE s.prod_id=p.prod_id
AND TO_CHAR(s.time_id, 'YYYY-MM') >= '01-jan-1998'
AND TO_CHAR(s.time_id, 'YYYY-MM') < '01-jan-1999'
GROUP BY p.prod_category;

SELECT mv.prod_category, mv.sum_amount
FROM sum_sales_pcat_monthly_mv mv
WHERE month >= '01-jan-1998' AND month < '01-jan-1999';
```

The range specified in the query represents an integral number of years, quarters, or months. Assume that there is a materialized view `mv3` that contains pre-summarized sales by `prod_type` and is defined as follows:

```
CREATE MATERIALIZED VIEW mv3
ENABLE QUERY REWRITE AS
SELECT prod_name, TO_CHAR(sales.time_id,'yyyy-mm')
   AS month, SUM(amount_sold) AS sum_sales
FROM sales, products WHERE sales.prod_id = products.prod_id
GROUP BY prod_name, TO_CHAR(sales_time_id, 'yyyy-mm');
```

The query can be rewritten by first folding the date range into the month range and then matching the expressions representing the months with the month expression in `mv3`. This rewrite is shown in two steps (first folding the date range followed by the actual rewrite).

```
SELECT prod_name, SUM(amount_sold) AS sum_sales
FROM sales, products
WHERE sales.prod_id = products.prod_id AND TO_CHAR(sales.time_id, 'yyyy-mm') >=
      TO_CHAR('01-jan-1998', 'yyyy-mm') AND TO_CHAR(sales.time_id, '01-jan-1999',
 'yyyy-mm') < TO_CHAR(TO_DATE(''01-jan-1999'', ''dd-mon-yyyy''), ''yyyy-mm'')
GROUP BY prod_name;

SELECT prod_name, sum_sales
FROM mv3 WHERE month >=
      TO_CHAR(TO_DATE('01-jan-1998', 'dd-mon-yyyy'), 'yyyy-mm')
  AND month < TO_CHAR(TO_DATE('01-jan-1999', 'dd-mon-yyyy'), 'yyyy-mm');
```

If `mv3` had pre-summarized sales by `prod_name` and year instead of `prod_name` and month, the query could still be rewritten by folding the date range into year range and then matching the year expressions.

## 12.3.6 About Query Rewrite Using View Constraints

Data warehouse applications recognize multi-dimensional cubes in the database by identifying integrity constraints in the relational schema. Integrity constraints represent primary and foreign key relationships between fact and dimension tables. By querying the data dictionary, applications can recognize integrity constraints and hence the cubes in the database. However, this does not work in an environment where database administrators, for schema complexity or security reasons, define views on fact and dimension tables. In such environments, applications cannot identify the cubes properly. By allowing constraint definitions between views, you can propagate base table constraints to the views, thereby allowing applications to recognize cubes even in a restricted environment.

View constraint definitions are declarative in nature, but operations on views are subject to the integrity constraints defined on the underlying base tables, and constraints on views can be enforced through constraints on base tables. Defining constraints on base tables is necessary,


Copyright © 2001, 2025, Oracle and/or its affiliates.

not only for data correctness and cleanliness, but also for materialized view query rewrite purposes using the original base objects.

> **See Also**
>
> Abut View Constraints Restrictions

Materialized view rewrite extensively uses constraints for query rewrite. They are used for determining lossless joins, which, in turn, determine if joins in the materialized view are compatible with joins in the query and thus if rewrite is possible.

`DISABLE NOVALIDATE` is the only valid state for a view constraint. However, you can choose `RELY` or `NORELY` as the view constraint state to enable more sophisticated query rewrites. For example, a view constraint in the `RELY` state allows query rewrite to occur when the query integrity level is set to `TRUSTED`. Table 12-3 illustrates when view constraints are used for determining lossless joins.

Note that view constraints cannot be used for query rewrite integrity level `ENFORCED`. This level enforces the highest degree of constraint enforcement `ENABLE VALIDATE`.

**Table 12-3 View Constraints and Rewrite Integrity Modes**

| Constraint States | RELY | NORELY |
|---|---|---|
| ENFORCED | No | No |
| TRUSTED | Yes | No |
| STALE_TOLERATED | Yes | No |

**Example 12-10 View Constraints**

To demonstrate the rewrite capabilities on views, you need to extend the `sh` sample schema as follows:

```
CREATE VIEW time_view AS
SELECT time_id, TO_NUMBER(TO_CHAR(time_id, 'ddd')) AS day_in_year FROM times;
```

You can now establish a foreign key/primary key relationship (in `RELY` mode) between the view and the fact table, and thus rewrite takes place as described in Table 12-3, by adding the following constraints. Rewrite will then work for example in `TRUSTED` mode.

```
ALTER VIEW time_view ADD (CONSTRAINT time_view_pk
   PRIMARY KEY (time_id) DISABLE NOVALIDATE);
ALTER VIEW time_view MODIFY CONSTRAINT time_view_pk RELY;
ALTER TABLE sales ADD (CONSTRAINT time_view_fk FOREIGN KEY (time_id)
   REFERENCES time_view(time_id) DISABLE NOVALIDATE);
ALTER TABLE sales MODIFY CONSTRAINT time_view_fk RELY;
```

Consider the following materialized view definition:

```
CREATE MATERIALIZED VIEW sales_pcat_cal_day_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_category, t.day_in_year, SUM(s.amount_sold) AS sum_amount_sold
FROM time_view t, sales s, products p
WHERE t.time_id = s.time_id AND p.prod_id = s.prod_id
GROUP BY p.prod_category, t.day_in_year;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

The following query, omitting the dimension table products, is also rewritten without the primary key/foreign key relationships, because the suppressed join between sales and products is known to be lossless.

```
SELECT t.day_in_year, SUM(s.amount_sold) AS sum_amount_sold
FROM time_view t, sales s WHERE t.time_id = s.time_id
GROUP BY t.day_in_year;
```

However, if the materialized view sales_pcat_cal_day_mv were defined only in terms of the view time_view, then you could not rewrite the following query, suppressing then join between sales and time_view, because there is no basis for losslessness of the delta materialized view join. With the additional constraints as shown previously, this query will also rewrite.

```
SELECT p.prod_category, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p WHERE p.prod_id = s.prod_id
GROUP BY p.prod_category;
```

To undo the changes you have made to the sh schema, issue the following statements:

```
ALTER TABLE sales DROP CONSTRAINT time_view_fk;
DROP VIEW time_view;
```

### 12.3.6.1 Abut View Constraints Restrictions

If the referential constraint definition involves a view, that is, either the foreign key or the referenced key resides in a view, the constraint can only be in DISABLE NOVALIDATE mode.

A RELY constraint on a view is allowed only if the referenced UNIQUE or PRIMARY KEY constraint in DISABLE NOVALIDATE mode is also a RELY constraint.

The specification of ON DELETE actions associated with a referential Integrity constraint, is not allowed (for example, DELETE cascade). However, DELETE, UPDATE, and INSERT operations are allowed on views and their base tables as view constraints are in DISABLE NOVALIDATE mode.

## 12.3.7 About Query Rewrite in the Presence of Hybrid Partitioned Tables

Query rewrite considers external partitions in a hybrid partitioned table to be of UNKNOWN freshness. Therefore, when a query requests data from one or more external partitions, it can only be rewritten under TRUSTED or STALE_TOLERATED integrity mode.

When a materialized view that is based on a hybrid partitioned table includes the partition key or partition marker in its SELECT list, it is eligible for partition tracking. For materialized views based on hybrid partitioned table tables that are not PCT-enabled, STALE_TOLERATED is the only possible integrity mode.

Queries against hybrid partitioned tables can be rewritten using PCT rewrite under ENFORCED and TRUSTED integrity modes only if the hybrid partitioned table is range or list partitioned.

**Example 12-11 Query Rewrite and Materialized Views Based on Hybrid Partitioned Tables**

The hybrid partitioned table named hybrid_sales uses the ENFORCED integrity mode. One of the internal partitions is stale.

The following query is run:

```
SELECT customer_no, sum(price) as sum_price
FROM hybrid_sales WHERE
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
  time_id > TO_DATE('01-01-1950') and time_id < TO_DATE('06-01-2001')
GROUP BY customer_no;
```

This query can be rewritten to use the hybrid partitioned table. PCT rewrite selects the fresh partitions from the materialized view and any stale partitions and external partitions directly from the base table. The rewritten query is as follows:

```
SELECT v1.customer_no, SUM(v1.total_price) sum_price
FROM
  (SELECT customer_no, SUM(total_price) FROM Hybrid_sales WHERE
    time_id > TO_DATE('01-01-1950') and time_id < TO_DATE('01-01-2000')
   GROUP BY customer_no
   UNION ALL
   SELECT customer_no, SUM(total_price) FROM HyPT_MV WHERE
    time_id > TO_DATE('01-01-2000') and time_id < TO_DATE('01-01-2001')
   GROUP BY customer_no
   UNION ALL
   SELECT customer_no, SUM(total_price) FROM Hybrid_sales WHERE
    time_id > TO_DATE('01-01-2001') and time_id < TO_DATE('06-01-2001')
   GROUP BY customer_no
) v1
GROUP BY v1.customer_no;
```

## 12.3.8 Query Rewrite Using Set Operator Materialized Views

You can use query rewrite with materialized views that contain set operators. In this case, the query and materialized view do not have to match textually for rewrite to occur. As an example, consider the following materialized view, which uses the postal codes for male customers from San Francisco or Los Angeles:

```
CREATE MATERIALIZED VIEW cust_male_postal_mv
ENABLE QUERY REWRITE AS
SELECT c.cust_city, c.cust_postal_code
FROM sh.customers c
WHERE c.cust_gender = 'M' AND c.cust_city = 'San Francisco'
UNION ALL
SELECT c.cust_city, c.cust_postal_code
FROM sh.customers c
WHERE c.cust_gender = 'M' AND c.cust_city = 'Los Angeles';
```

If you have the following query, which displays the postal codes for male customers from San Francisco or Los Angeles:

```
SELECT c.cust_city, c.cust_postal_code
FROM sh.customers c
WHERE c.cust_city = 'Los Angeles' AND c.cust_gender = 'M'
UNION ALL
SELECT c.cust_city, c.cust_postal_code
FROM sh.customers c
WHERE c.cust_city = 'San Francisco' AND c.cust_gender = 'M';
```

The rewritten query will be the following:

```
SELECT mv.cust_city, mv.cust_postal_code
FROM cust_male_postal_mv mv;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

The rewritten query has dropped the UNION ALL and replaced it with the materialized view. Normally, query rewrite has to use the existing set of general eligibility rules to determine if the SELECT subselections under the UNION ALL are equivalent in the query and the materialized view.

See UNION ALL Marker and Query Rewrite.

If, for example, you have a query that retrieves the postal codes for male customers from San Francisco, Palmdale, or Los Angeles, the same rewrite can occur as in the previous example but query rewrite must keep the UNION ALL with the base tables, as in the following:

```
SELECT c.cust_city, c.cust_postal_code
FROM sh.customers c
WHERE c.cust_city= 'Palmdale' AND c.cust_gender ='M'
UNION ALL
SELECT c.cust_city, c.cust_postal_code
FROM sh.customers c
WHERE c.cust_city = 'Los Angeles' AND c.cust_gender = 'M'
UNION ALL
SELECT c.cust_city, c.cust_postal_code
FROM sh.customers c
WHERE c.cust_city = 'San Francisco' AND c.cust_gender = 'M';
```

The rewritten query will be:

```
SELECT mv.cust_city, mv.cust_postal_code
FROM cust_male_postal_mv mv
UNION ALL
SELECT c.cust_city, c.cust_postal_code
FROM sh.customers c
WHERE c.cust_city = 'Palmdale' AND c.cust_gender = 'M';
```

So query rewrite detects the case where a subset of the UNION ALL can be rewritten using the materialized view cust_male_postal_mv.

UNION, UNION ALL, and INTERSECT are commutative, so query rewrite can rewrite regardless of the order the subselects are found in the query or materialized view. However, MINUS is not commutative. A MINUS B is not equivalent to B MINUS A. Therefore, the order in which the subselects appear under the MINUS operator in the query and the materialized view must be in the same order for rewrite to happen. As an example, consider the case where there exists an old version of the customer table called customer_old and you want to find the difference between the old one and the current customer table only for male customers who live in London. That is, you want to find those customers in the current one that were not in the old one. The following example shows how this is done using a MINUS operator:

```
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city= 'Los Angeles' AND c.cust_gender = 'M'
MINUS
SELECT c.cust_city, c.cust_postal_code
FROM customers_old c
WHERE c.cust_city = 'Los Angeles' AND c.cust_gender = 'M';
```

Switching the subselects would yield a different answer. This illustrates that MINUS is not commutative.

## 12.3.8.1 UNION ALL Marker and Query Rewrite

If a materialized view contains one or more UNION ALL operators, it can also include a UNION ALL marker. The UNION ALL marker is used to identify from which UNION ALL subselect each row


Copyright © 2001, 2025, Oracle and/or its affiliates.

in the materialized view originates. Query rewrite can use the marker to distinguish what rows coming from the materialized view belong to a certain UNION ALL subselect. This is useful if the query needs only a subset of the data from the materialized view or if the subselects of the query do not textually match with the subselects of the materialized view. As an example, the following query retrieves the postal codes for male customers from San Francisco and female customers from Los Angeles:

```
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_gender = 'M' and c.cust_city = 'San Francisco'
UNION ALL
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_gender = 'F' and c.cust_city = 'Los Angeles';
```

The query can be answered using the following materialized view:

```
CREATE MATERIALIZED VIEW cust_postal_mv
ENABLE QUERY REWRITE AS
SELECT 1 AS marker, c.cust_gender, c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city = 'Los Angeles'
UNION ALL
SELECT 2 AS marker, c.cust_gender, c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city = 'San Francisco';
```

The rewritten query is as follows:

```
SELECT mv.cust_city, mv.cust_postal_code
FROM cust_postal_mv mv
WHERE mv.marker = 2 AND mv.cust_gender = 'M'
UNION ALL
SELECT mv.cust_city, mv.cust_postal_code
FROM cust_postal_mv mv
WHERE mv.marker = 1 AND mv.cust_gender = 'F';
```

The WHERE clause of the first subselect includes mv.marker = 2 and mv.cust_gender = 'M', which selects only the rows that represent male customers in the second subselect of the UNION ALL. The WHERE clause of the second subselect includes mv.marker = 1 and mv.cust_gender = 'F', which selects only those rows that represent female customers in the first subselect of the UNION ALL. Note that query rewrite cannot take advantage of set operators that drop duplicate or distinct rows. For example, UNION drops duplicates so query rewrite cannot tell what rows have been dropped, as in the following:

```
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city= 'Palmdale' AND c.cust_gender ='M'
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_gender = 'M' and c.cust_city = 'San Francisco'
UNION ALL
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_gender = 'F' and c.cust_city = 'Los Angeles';
```

The rewritten query using UNION ALL markers is as follows:

```
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city= 'Palmdale' AND c.cust_gender ='M'
UNION ALL
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
SELECT mv.cust_city, mv.cust_postal_code

FROM cust_postal_mv mv
WHERE mv.marker = 2 AND mv.cust_gender = 'M'
UNION ALL
  SELECT mv.cust_city, mv.cust_postal_code
  FROM cust_postal_mv mv
  WHERE mv.marker = 1 AND mv.cust_gender = 'F';
```

The rules for using a marker are that it must:

- Be a constant number or string and be the same data type for all `UNION ALL` subselects.
- Yield a constant, distinct value for each `UNION ALL` subselect. You cannot reuse the same value in multiple subselects.
- Be in the same ordinal position for all subselects.

## 12.3.9 About Query Rewrite in the Presence of Grouping Sets

This section discusses the following considerations for using query rewrite with grouping sets:

- About Query Rewrite When Using GROUP BY Extensions
- Hint for Rewriting Queries with Extended GROUP BY

### 12.3.9.1 About Query Rewrite When Using GROUP BY Extensions

Several extensions to the `GROUP BY` clause in the form of `GROUPING SETS`, `CUBE, ROLLUP`, and their concatenation are available. These extensions enable you to selectively specify the groupings of interest in the `GROUP BY` clause of the query. For example, the following is a typical query with grouping sets. This query uses the `sh` sample schema.

```
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
  SUM(s.amount_sold) AS sum_amount_sold
FROM sh.sales s, sh.customers c, sh.products p, sh.times t
WHERE s.time_id=t.time_id AND s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY GROUPING SETS ((p.prod_subcategory, t.calendar_month_desc),
   (c.cust_city, p.prod_subcategory));
```

The term **base grouping** for queries with `GROUP BY` extensions denotes all unique expressions present in the `GROUP BY` clause. In the previous query, the following grouping (`p.prod_subcategory, t.calendar_month_desc, c.cust_city`) is a base grouping.

The extensions can be present in user queries and in the queries defining materialized views. In both cases, materialized view rewrite applies and you can distinguish rewrite capabilities into the following scenarios:

- Materialized View has Simple GROUP BY and Query has Extended GROUP BY
- Materialized View has Extended GROUP BY and Query has Simple GROUP BY
- Both Materialized View and Query Have Extended GROUP BY

#### 12.3.9.1.1 Materialized View has Simple GROUP BY and Query has Extended GROUP BY

When a query contains an extended `GROUP BY` clause, it can be rewritten with a materialized view if its base grouping can be rewritten using the materialized view as listed in the rewrite rules explained in "When Does Oracle Rewrite a Query?". For example, in the following query which uses the `sh` sample schema:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
  SUM(s.amount_sold) AS sum_amount_sold
FROM sh.sales s, sh.customers c, sh.products p, sh.times t
WHERE s.time_id=t.time_id AND s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY GROUPING SETS
((p.prod_subcategory, t.calendar_month_desc),
   (c.cust_city, p.prod_subcategory));
```

The base grouping is `(p.prod_subcategory, t.calendar_month_desc, c.cust_city, p.prod_subcategory))` and, consequently, Oracle can rewrite the query using `sum_sales_pscat_month_city_mv` as follows:

```
SELECT mv.prod_subcategory, mv.calendar_month_desc, mv.cust_city,
  SUM(mv.sum_amount_sold) AS sum_amount_sold
FROM sum_sales_pscat_month_city_mv mv
GROUP BY GROUPING SETS
((mv.prod_subcategory, mv.calendar_month_desc),
  (mv.cust_city, mv.prod_subcategory));
```

A special situation arises if the query uses the `EXPAND_GSET_TO_UNION` hint. See "Hint for Rewriting Queries with Extended GROUP BY" for an example of using `EXPAND_GSET_TO_UNION`.

### 12.3.9.1.2 Materialized View has Extended GROUP BY and Query has Simple GROUP BY

In order for a materialized view with an extended `GROUP BY` to be used for rewrite, it must satisfy two additional conditions:

- It must contain a grouping distinguisher, which is the `GROUPING_ID` function on all `GROUP BY` expressions. For example, if the `GROUP BY` clause of the materialized view is `GROUP BY CUBE(a, b)`, then the `SELECT` list should contain `GROUPING_ID(a, b)`.
- The `GROUP BY` clause of the materialized view should not result in any duplicate groupings. For example, `GROUP BY GROUPING SETS((a, b), (a, b))` would disqualify a materialized view from general rewrite.

A materialized view with an extended `GROUP BY` contains multiple groupings. Oracle finds the grouping with the lowest cost from which the query can be computed and uses that for rewrite. For example, consider the following materialized view based on the `sh` sample schema:

```
CREATE MATERIALIZED VIEW sum_grouping_set_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_category, p.prod_subcategory, c.cust_state_province, c.cust_city,
  GROUPING_ID(p.prod_category,p.prod_subcategory,
              c.cust_state_province,c.cust_city) AS gid,
  SUM(s.amount_sold) AS sum_amount_sold
FROM sh.sales s, sh.products p, sh.customers c
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY GROUPING SETS
((p.prod_category, p.prod_subcategory, c.cust_city),
  (p.prod_category, p.prod_subcategory, c.cust_state_province, c.cust_city),
  (p.prod_category, p.prod_subcategory));
```

In this case, the following query is rewritten:

```
SELECT p.prod_subcategory, c.cust_city, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY p.prod_subcategory, c.cust_city;
```

This query is rewritten with the closest matching grouping from the materialized view. That is, the `(prod_category, prod_subcategory, cust_city)` grouping:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
SELECT prod_subcategory, cust_city, SUM(sum_amount_sold) AS sum_amount_sold
FROM sum_grouping_set_mv
WHERE gid = grouping identifier of (prod_category,prod_subcategory, cust_city)
GROUP BY prod_subcategory, cust_city;
```

### 12.3.9.1.3 Both Materialized View and Query Have Extended GROUP BY

When both materialized view and the query contain `GROUP BY` extensions, Oracle uses two strategies for rewrite: grouping match and `UNION ALL` rewrite. First, Oracle tries grouping match. The groupings in the query are matched against groupings in the materialized view and if all are matched with no rollup, Oracle selects them from the materialized view. For example, consider the following query:

```
SELECT p.prod_category, p.prod_subcategory, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY GROUPING SETS
((p.prod_category, p.prod_subcategory, c.cust_city),
  (p.prod_category, p.prod_subcategory));
```

This query matches two groupings from `sum_grouping_set_mv` and Oracle rewrites the query as the following:

```
SELECT prod_subcategory, cust_city, sum_amount_sold
FROM sum_grouping_set_mv
WHERE gid = grouping identifier of (prod_category,prod_subcategory, cust_city)
   OR gid = grouping identifier of (prod_category,prod_subcategory)
```

If grouping match fails, Oracle tries a general rewrite mechanism called `UNION ALL` rewrite. Oracle first represents the query with the extended `GROUP BY` clause as an equivalent `UNION ALL` query. Every grouping of the original query is placed in a separate `UNION ALL` branch. The branch will have a simple `GROUP BY` clause. For example, consider this query:

```
SELECT p.prod_category, p.prod_subcategory, c.cust_state_province,
  t.calendar_month_desc, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY GROUPING SETS
((p.prod_subcategory, t.calendar_month_desc),
  (t.calendar_month_desc),
  (p.prod_category, p.prod_subcategory, c.cust_state_province),
  (p.prod_category, p.prod_subcategory));
```

This is first represented as `UNION ALL` with four branches:

```
SELECT null, p.prod_subcategory, null,
    t.calendar_month_desc, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY p.prod_subcategory, t.calendar_month_desc
UNION ALL
  SELECT null, null, null,
    t.calendar_month_desc, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY t.calendar_month_desc
UNION ALL
SELECT p.prod_category, p.prod_subcategory, c.cust_state_province,
    null, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY p.prod_category, p.prod_subcategory, c.cust_state_province
UNION ALL
  SELECT p.prod_category, p.prod_subcategory, null,
    null, SUM(s.amount_sold) AS sum_amount_sold
  FROM sales s, products p, customers c, times t
  WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
  GROUP BY p.prod_category, p.prod_subcategory;
```

Each branch is then rewritten separately using the rules from "When Does Oracle Rewrite a Query?". Using the materialized view `sum_grouping_set_mv`, Oracle can rewrite only branches three (which requires materialized view rollup) and four (which matches the materialized view exactly). The unrewritten branches will be converted back to the extended `GROUP BY` form. Thus, eventually, the query is rewritten as:

```
SELECT null, p.prod_subcategory, null,
    t.calendar_month_desc, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY GROUPING SETS
  ((p.prod_subcategory, t.calendar_month_desc),
    (t.calendar_month_desc),)
UNION ALL
  SELECT prod_category, prod_subcategory, cust_state_province,
    null, SUM(sum_amount_sold) AS sum_amount_sold
  FROM sum_grouping_set_mv
  WHERE gid = <grouping id of (prod_category,prod_subcategory, cust_city)>
  GROUP BY p.prod_category, p.prod_subcategory, c.cust_state_province
UNION ALL
  SELECT prod_category, prod_subcategory, null,
    null, sum_amount_sold
  FROM sum_grouping_set_mv
  WHERE gid = <grouping id of (prod_category,prod_subcategory)>
```

Note that a query with extended `GROUP BY` is represented as an equivalent `UNION ALL` and recursively submitted for rewrite optimization. The groupings that cannot be rewritten stay in the last branch of `UNION ALL` and access the base data instead.

### 12.3.9.2 Hint for Rewriting Queries with Extended GROUP BY

You can use the `EXPAND_GSET_TO_UNION` hint to force expansion of the query with `GROUP BY` extensions into the equivalent `UNION ALL` query. This hint can be used in an environment where materialized views have simple `GROUP BY` clauses only. In this case, Oracle extends rewrite flexibility as each branch can be independently rewritten by a separate materialized view.

## 12.3.10 Query Rewrite in the Presence of Window Functions

Window functions are used to compute cumulative, moving and centered aggregates. These functions work with the following aggregates: `AVG`, `BIT_AND_AGG`, `BIT_OR_AGG`, `BIT_XOR_AGG`, `CHECKSUM`, `COUNT`, `FIRST_VALUE`, `KURTOSIS_POP`, `KURTOSIS_SAMP`, `LAST_VALUE`, `MAX`, `MIN`, `SKEWNESS_POP`, `SKEWNESS_SAMP`, `SUM`, `STDDEV`, and `VARIANCE`. A query with a window function can be rewritten using exact text match rewrite. This requires that the materialized view definition also matches the query exactly. When there is no window function on the materialized view, then a query with a window function can be rewritten provided the aggregate in the query is found in the materialized view and all other eligibility checks such as the join computability checks are successful. A window function on the query is compared to the window function in the materialized view using its canonical form format. This enables query rewrite to rewrite even complex window functions.


Copyright © 2001, 2025, Oracle and/or its affiliates.

When a query with a window function requires rollup during query rewrite, query rewrite, whenever possible, splits the query into an inner query with the aggregate and an outer query with the windowing function. This permits query rewrite to rewrite the aggregate in the inner query before applying the window function. One exception is that if the query has both a window function and a grouping set, then the presence of the grouping set prevents the splitting of the query, and therefore query rewrite does not take place.

## 12.3.11 Query Rewrite and Expression Matching

An expression that appears in a query can be replaced with a simple column in a materialized view provided the materialized view column represents a precomputed expression that matches with the expression in the query. If a query can be rewritten to use a materialized view, it will be faster. This is because materialized views contain precomputed calculations and do not need to perform expression computation.

The expression matching is done by first converting the expressions into canonical forms and then comparing them for equality. Therefore, two different expressions will generally be matched as long as they are equivalent to each other. Further, if the entire expression in a query fails to match with an expression in a materialized view, then subexpressions of it are tried to find a match. The subexpressions are tried in a top-down order to get maximal expression matching.

Consider a query against the `sh` sample schema that asks for sum of sales by age brackets (1-10, 11-20, 21-30, and so on).

```
CREATE MATERIALIZED VIEW sales_by_age_bracket_mv
ENABLE QUERY REWRITE AS
SELECT TO_CHAR((2000-c.cust_year_of_birth)/10-0.5,999) AS age_bracket,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sh.sales s, sh.customers c WHERE s.cust_id=c.cust_id
GROUP BY TO_CHAR((2000-c.cust_year_of_birth)/10-0.5,999);
```

The following query rewrites, using expression matching:

```
SELECT TO_CHAR(((2000-c.cust_year_of_birth)/10)-0.5,999), SUM(s.amount_sold)
FROM sales s, customers c WHERE s.cust_id=c.cust_id
GROUP BY TO_CHAR((2000-c.cust_year_of_birth)/10-0.5,999);
```

This query is rewritten in terms of `sales_by_age_bracket_mv` based on the matching of the canonical forms of the age bracket expressions (that is, 2000 - `c.cust_year_of_birth`)/ 10-0.5), as follows:

```
SELECT age_bracket, sum_amount_sold FROM sales_by_age_bracket_mv;
```

### 12.3.11.1 Query Rewrite Using Partially Stale Materialized Views

When a partition of the detail table is updated, only specific sections of the materialized view are marked stale. The materialized view must have information that can identify the partition of the table corresponding to a particular row or group of the materialized view. The simplest scenario is when the partitioning key of the table is available in the `SELECT` list of the materialized view because this is the easiest way to map a row to a stale partition. The key points when using partially stale materialized views are:

- Query rewrite can use a materialized view in `ENFORCED` or `TRUSTED` mode if the rows from the materialized view used to answer the query are known to be `FRESH`.
- The fresh rows in the materialized view are identified by adding selection predicates to the materialized view's `WHERE` clause. Oracle rewrites a query with this materialized view if its answer is contained within this (restricted) materialized view.


Copyright © 2001, 2025, Oracle and/or its affiliates.

The fact table sales is partitioned based on ranges of time_id as follows:

```
PARTITION BY RANGE (time_id)
(PARTITION SALES_Q1_1998
           VALUES LESS THAN (TO_DATE('01-APR-1998', 'DD-MON-YYYY')),
 PARTITION SALES_Q2_1998
           VALUES LESS THAN (TO_DATE('01-JUL-1998', 'DD-MON-YYYY')),
 PARTITION SALES_Q3_1998
           VALUES LESS THAN (TO_DATE('01-OCT-1998', 'DD-MON-YYYY')),
...
```

Suppose you have a materialized view grouping by time_id as follows:

```
CREATE MATERIALIZED VIEW sum_sales_per_city_mv
ENABLE QUERY REWRITE AS
SELECT s.time_id, p.prod_subcategory, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id
GROUP BY time_id, prod_subcategory, cust_city;
```

Also suppose new data will be inserted for December 2000, which will be assigned to partition sales_q4_2000. For testing purposes, you can apply an arbitrary DML operation on sales, changing a different partition than sales_q1_2000 as the following query requests data in this partition when this materialized view is fresh. For example, the following:

```
INSERT INTO SALES VALUES(17, 10, '01-DEC-2000', 4, 380, 123.45, 54321);
```

Until a refresh is done, the materialized view is generically stale and cannot be used for unlimited rewrite in enforced mode. However, because the table sales is partitioned and not all partitions have been modified, Oracle can identify all partitions that have not been touched. The optimizer can identify the fresh rows in the materialized view (the data which is unaffected by updates since the last refresh operation) by implicitly adding selection predicates to the materialized view defining query as follows:

```
SELECT s.time_id, p.prod_subcategory, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id
AND   s.time_id < TO_DATE('01-OCT-2000','DD-MON-YYYY')
OR s.time_id >= TO_DATE('01-OCT-2001','DD-MON-YYYY'))
GROUP BY time_id, prod_subcategory, cust_city;
```

Note that the freshness of partially stale materialized views is tracked on a per-partition base, and not on a logical base. Because the partitioning strategy of the sales fact table is on a quarterly base, changes in December 2000 causes the complete partition sales_q4_2000 to become stale.

Consider the following query, which asks for sales in quarters 1 and 2 of 2000:

```
SELECT s.time_id, p.prod_subcategory, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id
AND   s.time_id BETWEEN TO_DATE('01-JAN-2000', 'DD-MON-YYYY')
AND TO_DATE('01-JUL-2000', 'DD-MON-YYYY')
GROUP BY time_id, prod_subcategory, cust_city;
```

Oracle AI Database knows that those ranges of rows in the materialized view are fresh and can therefore rewrite the query with the materialized view. The rewritten query looks as follows:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
SELECT time_id, prod_subcategory, cust_city, sum_amount_sold
FROM sum_sales_per_city_mv
WHERE time_id BETWEEN TO_DATE('01-JAN-2000', 'DD-MON-YYYY')
AND TO_DATE('01-JUL-2000', 'DD-MON-YYYY');
```

Instead of the partitioning key, a partition marker (a function that identifies the partition given a rowid) can be present in the SELECT (and GROUP BY list) of the materialized view. You can use the materialized view to rewrite queries that require data from only certain partitions (identifiable by the partition-marker), for instance, queries that have a predicate specifying ranges of the partitioning keys containing entire partitions. See Advanced Materialized Views for details regarding the supplied partition marker function DBMS_MVIEW.PMARKER.

The following example illustrates the use of a partition marker in the materialized view instead of directly using the partition key column:

```
CREATE MATERIALIZED VIEW sum_sales_per_city_2_mv
ENABLE QUERY REWRITE AS
SELECT DBMS_MVIEW.PMARKER(s.rowid) AS pmarker,
       t.fiscal_quarter_desc, p.prod_subcategory, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id
AND   s.time_id = t.time_id
GROUP BY DBMS_MVIEW.PMARKER(s.rowid),
    p.prod_subcategory, c.cust_city, t.fiscal_quarter_desc;
```

Suppose you know that the partition sales_q1_2000 is fresh and DML changes have taken place for other partitions of the sales table. For testing purposes, you can apply an arbitrary DML operation on sales, changing a different partition than sales_q1_2000 when the materialized view is fresh. An example is the following:

```
INSERT INTO SALES VALUES(17, 10, '01-DEC-2000', 4, 380, 123.45, 54321);
```

Although the materialized view sum_sales_per_city_2_mv is now considered generically stale, Oracle AI Database can rewrite the following query using this materialized view. This query restricts the data to the partition sales_q1_2000, and selects only certain values of cust_city, as shown in the following:

```
SELECT p.prod_subcategory, c.cust_city, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id AND s.time_id = t.time_id
AND c.cust_city= 'Nuernberg'
AND s.time_id >=TO_DATE('01-JAN-2000','dd-mon-yyyy')
AND s.time_id <  TO_DATE('01-APR-2000','dd-mon-yyyy')
GROUP BY prod_subcategory, cust_city;
```

Note that rewrite with a partially stale materialized view that contains a PMARKER function can only take place when the complete data content of one or more partitions is accessed and the predicate condition is on the partitioned fact table itself, as shown in the earlier example.

The DBMS_MVIEW.PMARKER function gives you exactly one distinct value for each partition. This dramatically reduces the number of rows in a potential materialized view compared to the partitioning key itself, but you are also giving up any detailed information about this key. The only information you know is the partition number and, therefore, the lower and upper boundary values. This is the trade-off for reducing the cardinality of the range partitioning column and thus the number of rows.

Assuming the value of p_marker for partition sales_q1_2000 is 31070, the previously shown queries can be rewritten against the materialized view as follows:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
SELECT mv.prod_subcategory, mv.cust_city, SUM(mv.sum_amount_sold)
FROM sum_sales_per_city_2_mv mv
WHERE mv.pmarker = 31070 AND mv.cust_city= 'Nuernberg'
GROUP BY prod_subcategory, cust_city;
```

So the query can be rewritten against the materialized view without accessing stale data.

## 12.3.12 Bind Variables During Query Rewrite

Query rewrite is supported when the query contains user bind variables.

**Predicate-Based Rewrite with Bind Variables**

Consider the following materialized view, `customer_mv`, which has a predicate, (`cust_id > 1000`), in the `WHERE` clause:

```
CREATE MATERIALIZED VIEW customer_mv
ENABLE QUERY REWRITE AS
SELECT cust_id, prod_id,  SUM(amount_sold) AS total_amount
FROM sales
WHERE cust_id > 1000
GROUP BY cust_id, prod_id;
```

Previously, the following query, which uses a user bind variable, `:user_id`, in its `WHERE` clause, could not be rewritten:

```
SELECT cust_id, prod_id, SUM(amount_sold) AS sum_amount
FROM sales
WHERE cust_id > :user_id
GROUP BY cust_id, prod_id;
```

Because `customer_mv` includes a filter on `cust_id`, query rewrite required knowing the actual value of `:user_id` to determine containment. Since bind variables were not available at query rewrite time, this query was not eligible for rewrite.

**Bind Peeking in Query Rewrite**

Starting in Oracle AI Database 26ai, query rewrite supports evaluating user bind variables by determining whether their values fall within the boundary range defined by the materialized view. If the bind variable falls within this range, the query can be rewritten, allowing transformations to proceed. If the bind variable falls outside this range, then query rewrite is not performed.

**Example:**

If `:user_id > 1000`, query rewrite can occur, and the rewritten query is:

```
SELECT cust_id, prod_id, total_amount
FROM customer_mv
WHERE customer_mv.cust_id > :user_id;
```

If `:user_id = 1000`, the predicate matches the MV exactly, and the rewritten query is:

```
SELECT cust_id, prod_id, total_amount
FROM customer_mv;
```

If `:user_id < 1000`, query rewrite cannot be performed because the bind value falls outside the range supported by `customer_mv`.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Rewrite with Bind Variables Outside the WHERE or HAVING Clause**

Query rewrite is also possible when a query includes user bind variables in the SELECT list, because these bind variables do not affect containment checks. For example, consider a materialized view that includes an expression in the SELECT list:

```
SELECT MATERIALIZED VIEW customer_mv
ENABLE QUERY REWRITE AS
SELECT cust_id + 5 AS c1, prod_id, SUM(amount_sold) AS total_amount
FROM sales
GROUP BY cust_id + 5, prod_id;
```

Now consider a query that includes a user bind variable, :b1, in the corresponding expression:

```
-- Assume :b1 = 5
SELECT cust_id + :b1 AS c1, prod_id, SUM(amount_sold) AS total_amount
FROM sales s
GROUP BY cust_id + 5, prod_id;
```

Because :b1 = 5, the select list expression matches the materialized view exactly, and the rewritten query becomes:

```
SELECT mv.c1, mv.prod_id, mv.total_amount
FROM customer_mv mv;
```

## 12.3.13 Handling Expressions in Query Rewrite

Rewrite with some expressions is also supported when the expression evaluates to a constant, such as TO_DATE('12-SEP-1999','DD-Mon-YYYY'). For example, if an existing materialized view is defined as:

```
CREATE MATERIALIZED VIEW sales_on_valentines_day_99_mv
BUILD IMMEDIATE
REFRESH FORCE
ENABLE QUERY REWRITE AS
SELECT s.prod_id, s.cust_id, s.amount_sold
FROM times t, sales s WHERE s.time_id = t.time_id
AND t.time_id = TO_DATE('14-FEB-1999', 'DD-MON-YYYY');
```

Then the following query can be rewritten:

```
SELECT s.prod_id, s.cust_id, s.amount_sold
FROM sales s, times t WHERE s.time_id = t.time_id
AND t.time_id = TO_DATE('14-FEB-1999', 'DD-MON-YYYY');
```

This query would be rewritten as follows:

```
SELECT * FROM sales_on_valentines_day_99_mv;
```

Whenever TO_DATE is used, query rewrite only occurs if the date mask supplied is the same as the one specified by the NLS_DATE_FORMAT.

# 12.4 Advanced Query Rewrite Using Equivalences

There is a special type of query rewrite that is possible where a declaration is made that two SQL statements are functionally equivalent. This capability enables you to place inside application knowledge into the database so the database can exploit this knowledge for improved query performance. You do this by declaring two SELECT statements to be


Copyright © 2001, 2025, Oracle and/or its affiliates.

functionally equivalent (returning the same rows and columns) and indicating that one of the SELECT statements is more favorable for performance.

This advanced rewrite capability can generally be applied to a variety of query performance problems and opportunities. Any application can use this capability to affect rewrites against complex user queries that can be answered with much simpler and more performant queries that have been specifically created, usually by someone with inside application knowledge.

There are many scenarios where you can have inside application knowledge that would allow SQL statement transformation and tuning for significantly improved performance. The types of optimizations you may wish to affect can be very simple or as sophisticated as significant restructuring of the query. However, the incoming SQL queries are often generated by applications and you have no control over the form and structure of the application-generated queries.

To gain access to this capability, you need to connect as SYSDBA and explicitly grant execute access to the desired database administrators who will be declaring rewrite equivalences. See *Oracle Database PL/SQL Packages and Types Reference* for more information.

To illustrate this type of advanced rewrite, some examples using multidimensional data are provided. To optimize resource usage, an application may employ complicated SQL, custom C code or table functions to retrieve the data from the database. This complexity is irrelevant as far as end users are concerned. Users would still want to obtain their answers using typical queries with SELECT ... GROUP BY.

**Example 12-12 Rewrite Using Equivalence**

This example declares to Oracle that a given user query must be executed using a specified alternative query. Oracle would recognize this relationship and every time the user asked the query, it would transparently rewrite it using the alternative. Thus, the user is saved from the trouble of understanding and writing SQL for complicated aggregate computations.

There are two base tables sales_fact and geog_dim. You can compute the total sales for each city, state and region with a rollup, by issuing the following statement:

```
SELECT g.region, g.state, g.city,
GROUPING_ID(g.city, g.state, g.region), SUM(sales)
FROM sales_fact f, geog_dim g WHERE f.geog_key = g.geog_key
GROUP BY ROLLUP(g.region, g.state, g.city);
```

An application may want to materialize this query for quick results. Unfortunately, the resulting materialized view occupies too much disk space. However, if you have a dimension rolling up city to state to region, you can easily compress the three grouping columns into one column using a decode statement. (This is also known as an embedded total):

```
DECODE (gid, 0, city, 1, state, 3, region, 7, "grand_total")
```

What this does is use the lowest level of the hierarchy to represent the entire information. For example, saying Boston means Boston, MA, New England Region and saying CA means CA, Western Region. An application can store these embedded total results into a table, say, embedded_total_sales.

However, when returning the result back to the user, you would want to have all the data columns (city, state, region). In order to return the results efficiently and quickly, an application may use a custom table function (et_function) to retrieve the data back from the embedded_total_sales table in the expanded form as follows:

```
SELECT * FROM TABLE (et_function);
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

In other words, this feature allows an application to declare the equivalence of the user's preceding query to the alternative query, as in the following:

```
DBMS_ADVANCED_REWRITE.DECLARE_REWRITE_EQUIVALENCE (
   'EMBEDDED_TOTAL',
   'SELECT g.region, g.state, g.city,
    GROUPING_ID(g.city, g.state, g.region), SUM(sales)
    FROM sales_fact f, geog_dim g
    WHERE f.geog_key = g.geog_key
    GROUP BY ROLLUP(g.region, g.state, g.city)',
    'SELECT * FROM TABLE(et_function)');
```

> **Note**
>
> Both Oracle join syntax (shown above) and ANSI join syntax are supported.

This invocation of `DECLARE_REWRITE_EQUIVALENCE` creates an equivalence declaration named `EMBEDDED_TOTAL` stating that the specified `SOURCE_STMT` and the specified `DESTINATION_STMT` are functionally equivalent, and that the specified `DESTINATION_STMT` is preferable for performance. After the DBA creates such a declaration, the user need have no knowledge of the space optimization being performed underneath the covers.

This capability also allows an application to perform specialized partial materializations of a SQL query. For instance, it could perform a rollup using a `UNION ALL` of three relations as shown in Example 12-13.

**Example 12-13 Rewrite Using Equivalence (UNION ALL)**

```
CREATE MATERIALIZED VIEW T1
AS SELECT g.region, g.state, g.city, 0 AS gid, SUM(sales) AS sales
FROM sales_fact f, geog_dim g WHERE f.geog_key = g.geog_key
GROUP BY g.region, g.state, g.city;

CREATE MATERIALIZED VIEW T2 AS
SELECT t.region, t.state, SUM(t.sales) AS sales
FROM T1 GROUP BY t.region, t.state;

CREATE VIEW T3 AS
SELECT t.region, SUM(t.sales) AS sales
FROM T2 GROUP BY t.region;
```

The `ROLLUP(region, state, city)` query is then equivalent to:

```
SELECT * FROM T1 UNION ALL
SELECT region, state, NULL, 1 AS gid, sales FROM T2 UNION ALL
SELECT region, NULL, NULL, 3 AS gid, sales FROM T3 UNION ALL
SELECT NULL, NULL, NULL, 7 AS gid, SUM(sales) FROM T3;
```

By specifying this equivalence, Oracle AI Database would use the more efficient second form of the query to compute the `ROLLUP` query asked by the user.

```
DBMS_ADVANCED_REWRITE.DECLARE_REWRITE_EQUIVALENCE (
   'CUSTOM_ROLLUP',
   'SELECT g.region, g.state, g.city,
   GROUPING_ID(g.city, g.state, g.region), SUM(sales)
   FROM sales_fact f, geog_dim g
   WHERE f.geog_key = g.geog_key
   GROUP BY ROLLUP(g.region, g.state, g.city ',
   ' SELECT * FROM T1
   UNION ALL
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
   SELECT region, state, NULL, 1 as gid, sales FROM T2
   UNION ALL
   SELECT region, NULL, NULL, 3 as gid, sales FROM T3
   UNION ALL
   SELECT NULL, NULL, NULL, 7 as gid, SUM(sales) FROM T3');
```

Another application of this feature is to provide users special aggregate computations that may be conceptually simple but extremely complex to express in SQL. In this case, the application asks the user to use a specified custom aggregate function and internally compute it using complex SQL.

**Example 12-14 Rewrite Using Equivalence (Using a Custom Aggregate)**

Suppose the application users want to see the sales for each city, state and region and also additional sales information for specific seasons. For example, the New England user wants additional sales information for cities in New England for the winter months. The application would provide you a special aggregate `Seasonal_Agg` that computes the earlier aggregate. You would ask a classic summary query but use `Seasonal_Agg(sales, region)` rather than `SUM(sales)`.

```
SELECT g.region, t.calendar_month_name, Seasonal_Agg(f.sales, g.region) AS sales
FROM sales_fact f, geog_dim g, times t
WHERE f.geog_key = g.geog_key AND f.time_id = t.time_id
GROUP BY g.region, t.calendar_month_name;
```

Instead of asking the user to write SQL that does the extra computation, the application can do it automatically for them by using this feature. In this example, `Seasonal_Agg` is computed using the spreadsheet functionality (see SQL for Modeling). Note that even though `Seasonal_Agg` is a user-defined aggregate, the required behavior is to add extra rows to the query's answer, which cannot be easily done with simple PL/SQL functions.

```
DBMS_ADVANCED_REWRITE.DECLARE_REWRITE_EQUIVALENCE (
   'CUSTOM_SEASONAL_AGG',
   SELECT g.region, t.calendar_month_name, Seasonal_Agg(sales, region) AS sales
   FROM sales_fact f, geog_dim g, times t
   WHERE f.geog_key = g.geog_key AND f.time_id = t.time_id
   GROUP BY g.region, t.calendar_month_name',
   'SELECT g,region, t.calendar_month_name, SUM(sales) AS sales
   FROM sales_fact f, geog_dim g
   WHERE f.geog_key = g.geog_key AND t.time_id = f.time_id
   GROUP BY g.region, g.state, g.city, t.calendar_month_name
   DIMENSION BY g.region, t.calendar_month_name
   (sales ['New England', 'Winter'] = AVG(sales) OVER calendar_month_name IN
    ('Dec', 'Jan', 'Feb', 'Mar'),
   sales ['Western', 'Summer' ] = AVG(sales) OVER calendar_month_name IN
   ('May', 'Jun', 'July', 'Aug'), .);
```

# 12.5 Creating Result Cache Materialized Views with Equivalences

A special type of materialized view, called a result cache materialized view (RCMV), enables you to use a result cache when running query rewrite. These result cache materialized views offer the main advantages of the result cache, faster access with less space required, without the normal drawback of being unable to run query rewrite against them.

An example of using this type of materialized view is the following.


Copyright © 2001, 2025, Oracle and/or its affiliates.

### Example 12-15 Result Cache Materialized View

First, grant the requisite permissions:

```
CONNECT / AS SYSDBA
GRANT CREATE MATERIALIZED VIEW TO sh;
GRANT EXECUTE ON DBMS_ADVANCED_REWRITE TO sh;
```

Next, create the result cache materialized view:

```
CONNECT sh/sh
begin
   sys.DBMS_ADVANCED_REWRITE.Declare_Rewrite_Equivalence
    (
     Name               => 'RCMV_SALES',
     Source_Stmt        =>
      'select channel_id, prod_id, sum(amount_sold), count(amount_sold)
       from sales
       group by prod_id, channel_id',
     Destination_Stmt   =>
      'select * from
      (select /*+ RESULT_CACHE(name=RCMV_SALES) */
               channel_id, prod_id, sum(amount_sold), count(amount_sold)
         from sales
         group by prod_id, channel_id)',
       Validate         => FALSE,
       Rewrite_Mode     => 'GENERAL'
       );
end;
/

ALTER SESSION SET query_rewrite_integrity = stale_tolerated;
```

Then, verify that different queries all rewrite to RCMV_SALES by looking at the explain plan:

```
EXPLAIN PLAN FOR
  SELECT channel_id, SUM(amount_sold) FROM sales GROUP BY channel_id;
@?/rdbms/admin/utlxpls

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------
Plan hash value: 3903632134
------------------------------------------------------------------------------------
|Id |         Operation       | Name  |Rows|Bytes|Cost(%CPU)| Time |Pstart|Pstop|
------------------------------------------------------------------------------------
| 0 | SELECT STATEMENT        |       |  4 |   64| 1340 (68)|00:00:17|    |    |
| 1 |  HASH GROUP BY          |       |  4 |   64| 1340 (68)|00:00:17|    |    |
| 2 |   VIEW                  |       | 204| 3264| 1340 (68)|00:00:17|    |    |
| 3 |    RESULT CACHE         |3gps5zr86gyb53y36js9zuay2s| | | |     |    |    |
| 4 |     HASH GROUP BY       |       | 204| 2448| 1340 (68)|00:00:17|    |    |
| 5 |      PARTITION RANGE ALL|       |918K|  10M|  655 (33)|00:00:08|  1 | 28 |
| 6 |       TABLE ACCESS FULL | SALES |918K|  10M|  655 (33)|00:00:08|  1 | 28 |
------------------------------------------------------------------------------------

Result Cache Information (identified by operation id):
------------------------------------------------------
   3 - column-count=4; dependencies=(SH.SALES); name="RCMV_SALES"

18 rows selected.
```

Then, execute the query that creates the cached result:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
SELECT channel_id, SUM(amount_sold)
FROM sales
GROUP BY channel_id;

CHANNEL_ID  SUM(AMOUNT_SOLD)
----------  ----------------
         2        26346342.3
         4          13706802
         3        57875260.6
         9         277426.26
```

Next, verify that the materialized view was materialized in the result cache:

```
CONNECT / AS SYSDBA

SELECT name, scan_count hits, block_count blocks, depend_count dependencies
FROM V$RESULT_CACHE_OBJECTS
WHERE name = 'RCMV_SALES';

NAME         HITS    BLOCKS   DEPENDENCIES
----------   ----    ------   ------------
RCMV_SALES      0         5              1
```

Finally, drop the RCMV query equivalence:

```
begin
  sys.DBMS_ADVANCED_REWRITE.Drop_Rewrite_equivalence('RCMV_SALES');
end;
/
```

For more information regarding result caches, see *Oracle AI Database SQL Tuning Guide*.

# 12.6 Query Rewrite and Materialized Views Based on Approximate Queries

Queries containing SQL functions that return approximate results are automatically rewritten to use a matching materialized view, if these queries can be answered using the materialized view.

For a query containing SQL functions that return approximate results to be rewritten using a materialized view that is based on an approximate query, ensure that query rewrite is enabled for the materialized view. Query rewrite must also be enabled either at the database level or for the current session.

Consider the materialized view `approx_count_distinct_pdt_mv` that was defined as follows:

```
CREATE MATERIALIZED VIEW approx_count_distinct_pdt_mv
ENABLE QUERY REWRITE AS
SELECT t.calendar_year, t.calendar_month_number, t.day_number_in_month,
approx_count_distinct_detail(prod_id) daily_detail
FROM sales s, times t
WHERE s.time_id = t.time_id
GROUP BY t.calendar_year, t.calendar_month_number, t.day_number_in_month;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

> **Note**
>
> Both Oracle join syntax (shown above) and ANSI join syntax are supported.

When a query that matches the defining query of `approx_count_distinct_pdt_mv` is run, and the prerequisites described in this section are met, the query is automatically rewritten to use this materialized view. The following query is rewritten to use `approx_count_distinct_pdt_mv`, as indicated by the execution plan generated for the query.

```
SELECT t.calendar_year, t.calendar_month_number, t.day_number_in_month,
approx_count_distinct(prod_id)
FROM sales s, times t
WHERE s.time_id = t.time_id
GROUP BY t.calendar_year, t.calendar_month_number, t.day_number_in_month;

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
------------------------------
Plan hash value: 2307354865
------------------------------------------------------------------------------
------------------------------
| Id  | Operation                    | Name                         | Rows  |
Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
------------------------------
|   0 | SELECT STATEMENT             |                              |  1460 |
74460 |   205   (0)| 00:00:01 |
|   1 |  MAT_VIEW REWRITE ACCESS FULL| APPROX_COUNT_DISTINCT_PDT_MV |  1460 |
74460 |   205   (0)| 00:00:01 |
------------------------------------------------------------------------------
------------------------------

8 rows selected.
```

The following query is also rewritten to use `approx_count_distinct_pdt_mv` as indicated by the execution plan. Note that this query aggregates data to a higher level than that defined by the defining query of `approx_count_distinct_pdt_mv`.

```
SELECT t.calendar_year, t.calendar_month_number,
approx_count_distinct(prod_id)
FROM sales s, times t
WHERE s.time_id = t.time_id
GROUP BY t.calendar_year, t.calendar_month_number;

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
------------------------------
Plan hash value: 827336432
------------------------------------------------------------------------------
------------------------------
| Id  | Operation                     | Name                          | Rows
| Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
------------------------------
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
|   0 | SELECT STATEMENT             |                                |    34
|  1632 |   206   (1)| 00:00:01 |
|   1 |  HASH GROUP BY APPROX        |                                |    34
|  1632 |   206   (1)| 00:00:01 |
|   2 |   MAT_VIEW REWRITE ACCESS FULL| APPROX_COUNT_DISTINCT_PDT_MV |  1460
| 70080 |   205   (0)| 00:00:01 |
--------------------------------------------------------------------------------
--------------------------------

9 rows selected.
```

### Rewriting Queries with Exact Functions to Use Materialized Views that Contain Approximate Functions

If you set database initialization parameters that substitute exact functions with the corresponding SQL functions that return approximate values, then the optimizer can rewrite queries containing exact functions to use materialized views that are defined using the approximate versions of the same functions. You need not rewrite the query to use the corresponding approximate functions.

For example, if the `approx_for_count_distinct` parameter is set to `TRUE`, then the optimizer rewrites the following query to use the materialized view `approx_count_distinct_pdt_mv`:

```
ALTER SESSION SET approx_for_count_distinct = TRUE;

SELECT t.calendar_year, t.calendar_month_number, COUNT (DISTINCT prod_id)
FROM sales s, times t
WHERE s.time_id = t.time_id
GROUP BY t.calendar_year, t.calendar_month_number;

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
--------------------------------
Plan hash value: 827336432
--------------------------------------------------------------------------------
--------------------------------
| Id  | Operation              | Name                 | Rows  | Bytes | Cost
(%CPU)| Time     |
--------------------------------------------------------------------------------
--------------------------------
|   0 | SELECT STATEMENT          |                  |      34 |    1632
|     206   (1)| 00:00:01 |
|   1 |  HASH GROUP BY APPROX          |                  |      34 |    1632
|     206   (1)| 00:00:01 |
|   2 |   MAT_VIEW REWRITE ACCESS FULL| APPROX_COUNT_DISTINCT_PDT_MV |
1460 | 70080 |     205   (0)| 00:00:01 |
--------------------------------------------------------------------------------
--------------------------------

9 rows selected.
```

Observe that the above execution plan is the same as the execution plan that was generated when the query uses the `approx_count_distinct` in the previous example.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**See Also**

- About Approximate Query Processing
- About Approximate Aggregates
- Creating Materialized Views Based on Approximate Queries

# 12.7 Query Rewrite and Materialized Views Based on Bitmap-based COUNT(DISTINCT) Functions

Queries that contain `COUNT(DISTINCT)` operations on integer columns can be rewritten to use materialized views that contain bitmap-based functions.

Enable query rewrite for the materialized view so that SQL queries can be rewritten using this materialized views.

**Example 12-16 Query Rewrite Using Materialized Views Containing COUNT(DISTINCT)**

The materialized view `mv_sales` was created using the following command:

```
create materialized view mv_sales as
  select PROMO_ID, BITMAP_BUCKET_NUMBER(PROD_ID) bm_bktno,
  BITMAP_CONSTRUCT_AGG(BITMAP_BIT_POSITION(PROD_ID),'RAW') bm_details
  from sales
  group by PROMO_ID,BITMAP_BUCKET_NUMBER(PROD_ID);
```

Query rewrite has been enabled for the materialized view `mv_sales`.

When a SQL query performs a `COUNT(DISTINCT)` operation on a numeric column that is included in the `mv_sales` materialized view definition, the query is rewritten to use the materialized view. The execution plan below shows that the materialized view was used.

```
SQL> EXPLAIN PLAN FOR select PROMO_ID,count(distinct PROD_ID) from sales
group by PROMO_ID order by PROMO_ID;

Explained.

SQL> SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
------------------------------
Plan hash value: 2440767223

-----------------------------------------------
| Id  | Operation               | Name       |
-----------------------------------------------
|   0 | SELECT STATEMENT        |            |
|   1 |  SORT GROUP BY          |            |
|   2 |   VIEW                  |            |
|   3 |    HASH GROUP BY        |            |
|   4 |     MAT_VIEW ACCESS FULL| MV_SALES   |
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
-----------------------------------------------

Note
-----
   - dynamic statistics used: dynamic sampling (level=2)

15 rows selected.
```

**Example 12-17 Query Rewrite with Bitmap-based COUNT(DISTINCT) and Rollup**

The following command was used to create the materialized view `mv_sales_amount`:

```
create materialized view mv_sales_amount AS
SELECT PROMO_ID, CHANNEL_ID,
       BITMAP_BUCKET_NUMBER(PROD_ID) as bm_bktno,
       BITMAP_CONSTRUCT_AGG(BITMAP_BIT_POSITION(PROD_ID)) as bm_details,
       SUM(AMOUNT_SOLD) as amount_sold
FROM sales
GROUP BY PROMO_ID, CHANNEL_ID, BITMAP_BUCKET_NUMBER(PROD_ID);
```

Query rewrite has been enabled for the materialized view `mv_sales_amount`.

The execution plan for the SQL command shown below demonstrates query rewrite to satisfy queries containing a `COUNT(DISTINCT)` function. Query rewrite is performed at different levels of aggregation and in the presence of other aggregates.

```
EXPLAIN PLAN FOR
SELECT PROMO_ID, COUNT(DISTINCT PROD_ID), SUM(AMOUNT_SOLD)
FROM sales
GROUP BY PROMO_ID;
-------------------------------------------------------------------------------
-------------
| Id | Operation          | Name            | Rows| Bytes| Cost (%CPU) |
Time     |
-------------------------------------------------------------------------------
-------------
| 0 | SELECT STATEMENT    |                 | 163 | 6357 | 8 (13)      |
00:00:01 |
| 1 | HASH GROUP BY       |                 | 163 | 6357 | 8 (13)      |
00:00:01 |
| 2 | VIEW                |                 | 163 | 6357 | 8 (13)      |
00:00:01 |
| 3 | HASH GROUP BY       |                 | 163 | 324K | 8 (13)      |
00:00:01 |
| 4 | MAT_VIEW ACCESS FULL| MV_SALES_AMOUNT | 163 | 324K | 7 (0)       |
00:00:01 |
-------------------------------------------------------------------------------
-------------
```

# 12.8 Verifying that Query Rewrite has Occurred

Because query rewrite occurs transparently, special steps have to be taken to verify that a query has been rewritten. Of course, if the query runs faster, this should indicate that rewrite


Copyright © 2001, 2025, Oracle and/or its affiliates.

has occurred, but that is not proof. Therefore, to confirm that query rewrite does occur, use the EXPLAIN PLAN statement or the DBMS_MVIEW.EXPLAIN_REWRITE procedure.

This section contains the following topics:

- Using EXPLAIN PLAN with Query Rewrite
- Using the EXPLAIN_REWRITE Procedure with Query Rewrite

## 12.8.1 Using EXPLAIN PLAN with Query Rewrite

The EXPLAIN PLAN facility is used as described in *Oracle Database SQL Language Reference*. For query rewrite, all you need to check is that the operation shows MAT_VIEW REWRITE ACCESS. If it does, then query rewrite has occurred. An example is the following, which creates the materialized view cal_month_sales_mv:

```
CREATE MATERIALIZED VIEW cal_month_sales_mv
ENABLE QUERY REWRITE AS
SELECT  t.calendar_month_desc, SUM(s.amount_sold) AS dollars
FROM sales s, times t WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

If EXPLAIN PLAN is used on the following SQL statement, the results are placed in the default table PLAN_TABLE. However, PLAN_TABLE must first be created using the utlxplan.sql script. Note that EXPLAIN PLAN does not actually execute the query.

```
EXPLAIN PLAN FOR
SELECT  t.calendar_month_desc, SUM(s.amount_sold)
FROM  sales s, times t WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

For the purposes of query rewrite, the only information of interest from PLAN_TABLE is the operation OBJECT_NAME, which identifies the method used to execute this query. Therefore, you would expect to see the operation MAT_VIEW REWRITE ACCESS in the output as illustrated in the following:

```
SELECT OPERATION, OBJECT_NAME FROM PLAN_TABLE;

OPERATION                 OBJECT_NAME
--------------------      ----------------------
SELECT STATEMENT
MAT_VIEW REWRITE ACCESS    CALENDAR_MONTH_SALES_MV
```

## 12.8.2 Using the EXPLAIN_REWRITE Procedure with Query Rewrite

It can be difficult to understand why a query did not rewrite. The rules governing query rewrite eligibility are quite complex, involving various factors such as constraints, dimensions, query rewrite integrity modes, freshness of the materialized views, and the types of queries themselves. In addition, you may want to know why query rewrite chose a particular materialized view instead of another. To help with this matter, Oracle AI Database provides the DBMS_MVIEW.EXPLAIN_REWRITE procedure to advise you when a query can be rewritten and, if not, why not. Using the results from DBMS_MVIEW.EXPLAIN_REWRITE, you can take the appropriate action needed to make a query rewrite if at all possible.

Note that the query specified in the EXPLAIN_REWRITE statement does not actually execute.

This section contains the following topics:

- DBMS_MVIEW.EXPLAIN_REWRITE Syntax


Copyright © 2001, 2025, Oracle and/or its affiliates.

- Using REWRITE_TABLE to View EXPLAIN_REWRITE Output
- Using a Varray to View EXPLAIN_REWRITE Output
- EXPLAIN_REWRITE Benefit Statistics
- Support for Query Text Larger than 32KB in EXPLAIN_REWRITE
- About EXPLAIN_REWRITE and Multiple Materialized Views
- About EXPLAIN_REWRITE Output

## 12.8.2.1 DBMS_MVIEW.EXPLAIN_REWRITE Syntax

You can obtain the output from `DBMS_MVIEW.EXPLAIN_REWRITE` in two ways. The first is to use a table, while the second is to create a `VARRAY`. The following shows the basic syntax for using an output table:

```
DBMS_MVIEW.EXPLAIN_REWRITE (
    query           VARCHAR2,
    mv              VARCHAR2(30),
    statement_id    VARCHAR2(30));
```

You can create an output table called `REWRITE_TABLE` by executing the `utlxrw.sql` script.

The `query` parameter is a text string representing the SQL query. The parameter, `mv`, is a fully-qualified materialized view name in the form of `schema.mv`. This is an optional parameter. When it is not specified, `EXPLAIN_REWRITE` returns any relevant messages regarding all the materialized views considered for rewriting the given query. When `schema` is omitted and only `mv` is specified, `EXPLAIN_REWRITE` looks for the materialized view in the current schema.

If you want to direct the output of `EXPLAIN_REWRITE` to a varray instead of a table, you should call the procedure as follows:

```
DBMS_MVIEW.EXPLAIN_REWRITE (
    query           [VARCHAR2 | CLOB],
    mv               VARCHAR2(30),
    output_array     SYS.RewriteArrayType);
```

Note that if the query is less than 256 characters long, `EXPLAIN_REWRITE` can be easily invoked with the `EXECUTE` command from SQL*Plus. Otherwise, the recommended method is to use a PL/SQL `BEGIN... END` block, as shown in the examples in `/rdbms/demo/smxrw*`.

## 12.8.2.2 Using REWRITE_TABLE to View EXPLAIN_REWRITE Output

The output of `EXPLAIN_REWRITE` can be directed to a table named `REWRITE_TABLE`. You can create this output table by running the `utlxrw.sql` script. This script can be found in the `admin` directory. The format of `REWRITE_TABLE` is as follows:

```
CREATE TABLE REWRITE_TABLE(
  statement_id          VARCHAR2(30),   -- id for the query
  mv_owner              VARCHAR2(30),   -- owner of the MV
  mv_name               VARCHAR2(30),   -- name of the MV
  sequence              INTEGER,        -- sequence no of the msg
  query                 VARCHAR2(2000), -- user query
  query_block_no        INTEGER,        -- block no of the current subquery
  rewritten_txt         VARCHAR2(2000), -- rewritten query
  message               VARCHAR2(512),  -- EXPLAIN_REWRITE msg
  pass                  VARCHAR2(3),    -- rewrite pass no
  mv_in_msg             VARCHAR2(30),   -- MV in current message
  measure_in_msg        VARCHAR2(30),   -- Measure in current message
  join_back_tbl         VARCHAR2(30),   -- Join back table in message
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
  join_back_col         VARCHAR2(30),   -- Join back column in message
  original_cost         INTEGER,        -- Cost of original query
  rewritten_cost        INTEGER,        -- Cost of rewritten query
  flags                 INTEGER,        -- associated flags
  reserved1             INTEGER,        -- currently not used
  reerved2              VARCHAR2(10))   -- currently not used;
```

**Example 12-18 EXPLAIN_REWRITE Using REWRITE_TABLE**

An example PL/SQL invocation is:

```
EXECUTE DBMS_MVIEW.EXPLAIN_REWRITE -
('SELECT p.prod_name, SUM(amount_sold) ' || -
'FROM sales s, products p ' || -
'WHERE s.prod_id = p.prod_id ' || -
' AND prod_name > ''B%'' ' || -
' AND prod_name < ''C%'' ' || -
'GROUP BY prod_name', -
'TestXRW.PRODUCT_SALES_MV', -
'SH');

SELECT message FROM rewrite_table ORDER BY sequence;
MESSAGE
--------------------------------------------------------------------------------
QSM-01033: query rewritten with materialized view, PRODUCT_SALES_MV
1 row selected.
```

The demo file `xrwutl.sql` contains a procedure that you can call to provide a more detailed output from `EXPLAIN_REWRITE`. See "About EXPLAIN_REWRITE Output" for more information.

The following is an example where you can see a more detailed explanation of why some materialized views were not considered and, eventually, the materialized view `sales_mv` was chosen as the best one.

```
DECLARE
  qrytext VARCHAR2(500)  :='SELECT cust_first_name, cust_last_name,
 SUM(amount_sold) AS dollar_sales FROM sales s, customers c WHERE s.cust_id=
 c.cust_id GROUP BY cust_first_name, cust_last_name';
    idno    VARCHAR2(30) :='ID1';
BEGIN
  DBMS_MVIEW.EXPLAIN_REWRITE(qrytext, '', idno);
END;
/
SELECT message FROM rewrite_table ORDER BY sequence;


SQL> MESSAGE
--------------------------------------------------------------------------------
QSM-01082: Joining materialized view, CAL_MONTH_SALES_MV, with table, SALES, not possible
QSM-01022: a more optimal materialized view than PRODUCT_SALES_MV was used to rewrite
QSM-01022: a more optimal materialized view than FWEEK_PSCAT_SALES_MV was used to rewrite
QSM-01033: query rewritten with materialized view, SALES_MV
```

## 12.8.2.3 Using a Varray to View EXPLAIN_REWRITE Output

You can save the output of `EXPLAIN_REWRITE` in a PL/SQL `VARRAY`. The elements of this array are of the type `RewriteMessage`, which is predefined in the `SYS` schema as shown in the following:

```
TYPE RewriteMessage IS OBJECT(
  mv_owner          VARCHAR2(30),   -- MV's schema
  mv_name           VARCHAR2(30),   -- Name of the MV
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
  sequence        NUMBER(3),      -- sequence no of the msg
  query_text      VARCHAR2(2000), -- User query
  query_block_no  NUMBER(3),      -- block no of the current subquery
  rewritten_text  VARCHAR2(2000), -- rewritten query text
  message         VARCHAR2(512),  -- EXPLAIN_REWRITE error msg
  pass            VARCHAR2(3),    -- Query rewrite pass no
  mv_in_msg       VARCHAR2(30),   -- MV in current message
  measure_in_msg  VARCHAR2(30),   -- Measure in current message
  join_back_tbl   VARCHAR2(30),   -- Join back table in current msg
  join_back_col   VARCHAR2(30),   -- Join back column in current msg
  original_cost   NUMBER(10),     -- Cost of original query
  rewritten_cost  NUMBER(10),     -- Cost rewritten query
  flags           NUMBER,         -- Associated flags
  reserved1       NUMBER,         -- For future use
  reserved2       VARCHAR2(10)    -- For future use
);
```

The array type, `RewriteArrayType`, which is a varray of `RewriteMessage` objects, is predefined in the `SYS` schema as follows:

- `TYPE RewriteArrayType AS VARRAY(256) OF RewriteMessage;`
- Using this array type, now you can declare an array variable and specify it in the `EXPLAIN_REWRITE` statement.
- Each `RewriteMessage` record provides a message concerning rewrite processing.
- The parameters are the same as for `REWRITE_TABLE`, except for `statement_id`, which is not used when using a varray as output.
- The `mv_owner` field defines the owner of materialized view that is relevant to the message.
- The `mv_name` field defines the name of a materialized view that is relevant to the message.
- The `sequence` field defines the sequence in which messages should be ordered.
- The `query_text` field contains the first 2000 characters of the query text under analysis.
- The `message` field contains the text of message relevant to rewrite processing of `query`.
- The `flags`, `reserved1`, and `reserved2` fields are reserved for future use.

**Example 12-19 EXPLAIN_REWRITE Using a VARRAY**

Consider the following materialized view:

```
CREATE MATERIALIZED VIEW avg_sales_city_state_mv
ENABLE QUERY REWRITE AS
SELECT c.cust_city, c.cust_state_province, AVG(s.amount_sold)
FROM sales s, customers c WHERE s.cust_id = c.cust_id
GROUP BY c.cust_city, c.cust_state_province;
```

You might try to use this materialized view with the following query:

```
SELECT c.cust_state_province, AVG(s.amount_sold)
FROM sales s, customers c WHERE s.cust_id = c.cust_id
GROUP BY c.cust_state_province;
```

However, the query does not rewrite with this materialized view. This can be quite confusing to a novice user as it seems like all information required for rewrite is present in the materialized view. You can find out from `DBMS_MVIEW.EXPLAIN_REWRITE` that `AVG` cannot be computed from the given materialized view. The problem is that a `ROLLUP` is required here and `AVG` requires a `COUNT` or a `SUM` to do `ROLLUP`.

An example PL/SQL block for the previous query, using a `VARRAY` as its output, is as follows:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
SET SERVEROUTPUT ON
DECLARE
  Rewrite_Array SYS.RewriteArrayType := SYS.RewriteArrayType();
  querytxt VARCHAR2(1500) := 'SELECT c.cust_state_province,
  AVG(s.amount_sold)
   FROM sales s, customers c WHERE s.cust_id = c.cust_id
   GROUP BY c.cust_state_province';
  i NUMBER;
BEGIN
  DBMS_MVIEW.EXPLAIN_REWRITE(querytxt, 'AVG_SALES_CITY_STATE_MV',
  Rewrite_Array);
  FOR i IN 1..Rewrite_Array.count
  LOOP
    DBMS_OUTPUT.PUT_LINE(Rewrite_Array(i).message);
  END LOOP;
END;
/
```

The following is the output of this `EXPLAIN_REWRITE` statement:

```
QSM-01065: materialized view, AVG_SALES_CITY_STATE_MV, cannot compute
  measure, AVG, in the query
QSM-01101: rollup(s) took place on mv, AVG_SALES_CITY_STATE_MV
QSM-01053: NORELY referential integrity constraint on table, CUSTOMERS,
  in TRUSTED/STALE TOLERATED integrity mode
PL/SQL procedure successfully completed.
```

## 12.8.2.4 EXPLAIN_REWRITE Benefit Statistics

The output of `EXPLAIN_REWRITE` contains two columns, `original_cost` and `rewritten_cost`, that can help you estimate query cost. `original_cost` gives the optimizer's estimation for the query cost when query rewrite was disabled. `rewritten_cost` gives the optimizer's estimation for the query cost when query was rewritten using a materialized view. These cost values can be used to find out what benefit a particular query receives from rewrite.

## 12.8.2.5 Support for Query Text Larger than 32KB in EXPLAIN_REWRITE

In this release, the `EXPLAIN_REWRITE` procedure has been enhanced to support large queries. The input query text can now be defined using a `CLOB` data type instead of a `VARCHAR` data type. This allows `EXPLAIN_REWRITE` to accept queries up to 4 GB.

The syntax for using `EXPLAIN_REWRITE` using `CLOB` to obtain the output into a table is shown as follows:

```
DBMS_MVIEW.EXPLAIN_REWRITE(
   query          IN CLOB,
   mv             IN VARCHAR2,
   statement_id   IN VARCHAR2);
```

The second argument, `mv`, and the third argument, `statement_id`, can be `NULL`. Similarly, the syntax for using `EXPLAIN_REWRITE` using `CLOB` to obtain the output into a varray is shown as follows:

```
DBMS_MVIEW.EXPLAIN_REWRITE(
   query           IN CLOB,
   mv              IN VARCHAR2,
   msg_array       IN OUT SYS.RewriteArrayType);
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

As before, the second argument, mv, can be NULL. Note that long query texts in CLOB can be generated using the procedures provided in the DBMS_LOB package.

## 12.8.2.6 About EXPLAIN_REWRITE and Multiple Materialized Views

The syntax for using EXPLAIN_REWRITE with multiple materialized views is the same as using it with a single materialized view, except that the materialized views are specified by a comma-delimited string. For example, to find out whether a given set of materialized views mv1, mv2, and mv3 could be used to rewrite the query, query_txt, and, if not, why not, use EXPLAIN_REWRITE as follows:

```
DBMS_MVIEW.EXPLAIN_REWRITE(query_txt, 'mv1, mv2, mv3')
```

If the query, query_txt, rewrote with the given set of materialized views, then the following message appears:

```
QSM-01127: query rewritten with materialized view(s), mv1, mv2, and mv3.
```

If the query fails to rewrite with one or more of the given set of materialized views, then the reason for the failure will be output by EXPLAIN_REWRITE for each of the materialized views that did not participate in the rewrite.

## 12.8.2.7 About EXPLAIN_REWRITE Output

Some examples showing how to use EXPLAIN_REWRITE are included in /rdbms/demo/smxrw.sql. There is also a utility called SYS.XRW included in the demo xrw area to help you select the output from the EXPLAIN_REWRITE procedure. When EXPLAIN_REWRITE evaluates a query, its output includes information such as the rewritten query text, query block number, and the cost of the rewritten query. The utility SYS.XRW outputs the user specified fields in a neatly formatted way, so that the output can be easily understood. The syntax is as follows:

```
SYS.XRW(list_of_mvs, list_of_commands, query_text),
```

where list_of_mvs are the materialized views the user would expect the query rewrite to use. If there is more than one materialized view, they must be separated by commas, and list_of_commands is one of the following fields:

```
QUERY_TXT:      User query text
REWRITTEN_TXT:  Rewritten query text
QUERY_BLOCK_NO: Query block number to identify each query blocks in
                case the query has subqueries or inline views
PASS:           Pass indicates whether a given message was generated
                before or after the view merging process of query rewrite.
COSTS:          Costs indicates the estimated execution cost of the
                original query and the rewritten query
```

The following example illustrates the use of this utility:

```
DROP MATERIALIZED VIEW month_sales_mv;

CREATE MATERIALIZED VIEW month_sales_mv
  ENABLE QUERY REWRITE
  AS
  SELECT t.calendar_month_number, SUM(s.amount_sold) AS sum_dollars
  FROM sales s, times t
  WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_number;

SET SERVEROUTPUT ON
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
DECLARE
  querytxt VARCHAR2(1500) := 'SELECT t.calendar_month_number,
         SUM(s.amount_sold) AS sum_dollars FROM sales s, times t
  WHERE s.time_id = t.time_id GROUP BY t.calendar_month_number';
BEGIN
  SYS.XRW('MONTH_SALES_MV', 'COSTS, PASS, REWRITTEN_TXT, QUERY_BLOCK_NO', querytxt);
END;
/
```

Following is the output from SYS.XRW. As can be seen from the output, SYS.XRW outputs both the original query cost, rewritten costs, rewritten query text, query block number and whether the message was generated before or after the view merging process.

```
================================================================================
>> MESSAGE  : QSM-01151: query was rewritten
>> RW QUERY : SELECT MONTH_SALES_MV.CALENDAR_MONTH_NUMBER CALENDAR_MONTH_NUMBER,
MONTH_SALES_MV.SUM_DOLLARS SUM_DOLLARS FROM SH.MONTH_SALES_MV MONTH_SALES_MV
>> ORIG COST: 19.952763130792                  RW COST: 1.80687108
================================================================================
>>
------------------------ ANALYSIS OF QUERY REWRITE ------------------------
>>
>> QRY BLK #: 0
>> MESSAGE  : QSM-01209: query rewritten with materialized view,
   MONTH_SALES_MV, using text match algorithm
>> RW QUERY : SELECT MONTH_SALES_MV.CALENDAR_MONTH_NUMBER CALENDAR_MONTH_NUMBER,
   MONTH_SALES_MV.SUM_DOLLARS SUM_DOLLARS FROM SH.MONTH_SALES_MV MONTH_SALES_MV
>> ORIG COST: 19.952763130792                  RW COST: 1.80687108
>> MESSAGE OUTPUT BEFORE VIEW MERGING...
============================ END OF MESSAGES ============================
PL/SQL procedure successfully completed.
```

# 12.9 Design Considerations for Improving Query Rewrite Capabilities

This section discusses design considerations that will help in obtaining the maximum benefit from query rewrite. They are not mandatory for using query rewrite and rewrite is not guaranteed if you follow them. They are general rules to consider, and are the following:

- Query Rewrite Considerations: Constraints
- Query Rewrite Considerations: Dimensions
- Query Rewrite Considerations: Outer Joins
- Query Rewrite Considerations: Text Match
- Query Rewrite Considerations: Aggregates
- Query Rewrite Considerations: Grouping Conditions
- Query Rewrite Considerations: Expression Matching
- Query Rewrite Considerations: Date Folding
- Query Rewrite Considerations: Statistics
- Query Rewrite Considerations: Hints


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 12.9.1 Query Rewrite Considerations: Constraints

Make sure all inner joins referred to in a materialized view have referential integrity (foreign key/primary key constraints) with additional `NOT NULL` constraints on the foreign key columns. Because constraints tend to impose a large overhead, you could make them `NO VALIDATE` and `RELY` and set the parameter `QUERY_REWRITE_INTEGRITY` to `STALE_TOLERATED` or `TRUSTED`. However, if you set `QUERY_REWRITE_INTEGRITY` to `ENFORCED`, all constraints must be enabled, enforced, and validated to get maximum rewritability.

You should avoid using the `ON DELETE` clause as it can lead to unexpected results.

## 12.9.2 Query Rewrite Considerations: Dimensions

You can express the hierarchical relationships and functional dependencies in normalized or denormalized dimension tables using the `HIERARCHY` and `DETERMINES` clauses of a dimension. Dimensions can express intra-table relationships which cannot be expressed by constraints. Set the parameter `QUERY_REWRITE_INTEGRITY` to `TRUSTED` or `STALE_TOLERATED` for query rewrite to take advantage of the relationships declared in dimensions.

## 12.9.3 Query Rewrite Considerations: Outer Joins

Another way of avoiding constraints is to use outer joins in the materialized view. Query rewrite will be able to derive an inner join in the query, such as `(A.a=B.b)`, from an outer join in the materialized view `(A.a = B.b(+))`, as long as the rowid of `B` or column `B.b` is available in the materialized view. Most of the support for rewrites with outer joins is provided for materialized views with joins only. To exploit it, a materialized view with outer joins should store the rowid or primary key of the inner table of an outer join. For example, the materialized view `join_sales_time_product_mv_oj` stores the primary keys `prod_id` and `time_id` of the inner tables of outer joins.

## 12.9.4 Query Rewrite Considerations: Text Match

If you need to speed up an extremely complex, long-running query, you could create a materialized view with the exact text of the query. Then the materialized view would contain the query results, thus eliminating the time required to perform any complex joins and search through all the data for that which is required.

## 12.9.5 Query Rewrite Considerations: Aggregates

To get the maximum benefit from query rewrite, make sure that all aggregates which are needed to compute ones in the targeted set of queries are present in the materialized view. The conditions on aggregates are quite similar to those for incremental refresh. For instance, if `AVG(x)` is in the query, then you should store `COUNT(x)` and `AVG(x)` or store `SUM(x)` and `COUNT(x)` in the materialized view. See "General Restrictions on Fast Refresh" for fast refresh requirements.

## 12.9.6 Query Rewrite Considerations: Grouping Conditions

Aggregating data at lower levels in the hierarchy is better than aggregating at higher levels because lower levels can be used to rewrite more queries. Note, however, that doing so will also take up more space. For example, instead of grouping on state, group on city (unless space constraints prohibit it).


Copyright © 2001, 2025, Oracle and/or its affiliates.

Instead of creating multiple materialized views with overlapping or hierarchically related GROUP BY columns, create a single materialized view with all those GROUP BY columns. For example, instead of using a materialized view that groups by city and another materialized view that groups by month, use a single materialized view that groups by city and month.

Use GROUP BY on columns that correspond to levels in a dimension but not on columns that are functionally dependent, because query rewrite will be able to use the functional dependencies automatically based on the DETERMINES clause in a dimension. For example, instead of grouping on prod_name, group on prod_id (as long as there is a dimension which indicates that the attribute prod_id determines prod_name, you will enable the rewrite of a query involving prod_name).

## 12.9.7 Query Rewrite Considerations: Expression Matching

If several queries share the same common subselect, it is advantageous to create a materialized view with the common subselect as one of its SELECT columns. This way, the performance benefit due to precomputation of the common subselect can be obtained across several queries.

## 12.9.8 Query Rewrite Considerations: Date Folding

When creating a materialized view that aggregates data by folded date granules such as months or quarters or years, always use the year component as the prefix but not as the suffix. For example, TO_CHAR(date_col, 'yyyy-q') folds the date into quarters, which collate in year order, whereas TO_CHAR(date_col, 'q-yyyy') folds the date into quarters, which collate in quarter order. The former preserves the ordering while the latter does not. For this reason, any materialized view created without a year prefix will not be eligible for date folding rewrite.

## 12.9.9 Query Rewrite Considerations: Statistics

Optimization with materialized views is based on cost and the optimizer needs statistics of both the materialized view and the tables in the query to make a cost-based choice. Materialized views should thus have statistics collected using the DBMS_STATS package.

## 12.9.10 Query Rewrite Considerations: Hints

This section discusses the following considerations:

- Query Rewrite: REWRITE and NOREWRITE Hints
- Query Rewrite: REWRITE_OR_ERROR Hint
- Query Rewrite: Multiple Materialized View Rewrite Hints
- Query Rewrite: EXPAND_GSET_TO_UNION Hint

### 12.9.10.1 Query Rewrite: REWRITE and NOREWRITE Hints

You can include hints in the SELECT blocks of your SQL statements to control whether query rewrite occurs. Using the NOREWRITE hint in a query prevents the optimizer from rewriting it.

The REWRITE hint with no argument in a query forces the optimizer to use a materialized view (if any) to rewrite it regardless of the cost. If you use the REWRITE(mv1,mv2,...) hint with arguments, this forces rewrite to select the most suitable materialized view from the list of names specified.


Copyright © 2001, 2025, Oracle and/or its affiliates.

To prevent a rewrite, you can use the following statement:

```
SELECT /*+ NOREWRITE */ p.prod_subcategory, SUM(s.amount_sold)
FROM   sales s, products p WHERE  s.prod_id = p.prod_id
GROUP BY p.prod_subcategory;
```

To force a rewrite using `sum_sales_pscat_week_mv` (if such a rewrite is possible), use the following statement:

```
SELECT /*+ REWRITE (sum_sales_pscat_week_mv) */
       p.prod_subcategory,  SUM(s.amount_sold)
FROM   sales s, products p WHERE  s.prod_id=p.prod_id
GROUP BY p.prod_subcategory;
```

Note that the scope of a rewrite hint is a query block. If a SQL statement consists of several query blocks (`SELECT` clauses), you must specify a rewrite hint on each query block to control the rewrite for the entire statement.

### 12.9.10.2 Query Rewrite: REWRITE_OR_ERROR Hint

Using the `REWRITE_OR_ERROR` hint in a query causes the following error if the query failed to rewrite:

```
ORA-30393: a query block in the statement did not rewrite
```

For example, the following query issues an ORA-30393 error when there are no suitable materialized views for query rewrite to use:

```
SELECT /*+ REWRITE_OR_ERROR */ p.prod_subcategory, SUM(s.amount_sold)
FROM sales s, products p WHERE s.prod_id = p.prod_id
GROUP BY p.prod_subcategory;
```

### 12.9.10.3 Query Rewrite: Multiple Materialized View Rewrite Hints

There are two hints to control rewrites when using multiple materialized views. The `NO_MULTIMV_REWRITE` hint prevents the query from being rewritten with more than one materialized view and the `NO_BASETABLE_MULTIMV_REWRITE` hint prevents the query from being rewritten with a combination of materialized views and the base tables.

### 12.9.10.4 Query Rewrite: EXPAND_GSET_TO_UNION Hint

You can use the `EXPAND_GSET_TO_UNION` hint to force expansion of the query with `GROUP BY` extensions into the equivalent `UNION ALL` query. See "Hint for Rewriting Queries with Extended GROUP BY" for further information.

## 12.10 About Semi-Join Materialized View Rewrite

Semi-Join Materialized View Rewrite is a unique form of query rewrite introduced in Oracle AI Database 26ai.

In Semi-Join Materialized View Rewrite, a single large unified dimension table in the query is replaced with one or more join-specific materialized views. In a unified dimension data model, where multiple dimension tables are merged into a single large dimension table, Semi-Join Materialized Views materialize one or more of the joins of such a single, large unified dimension table with the fact table.

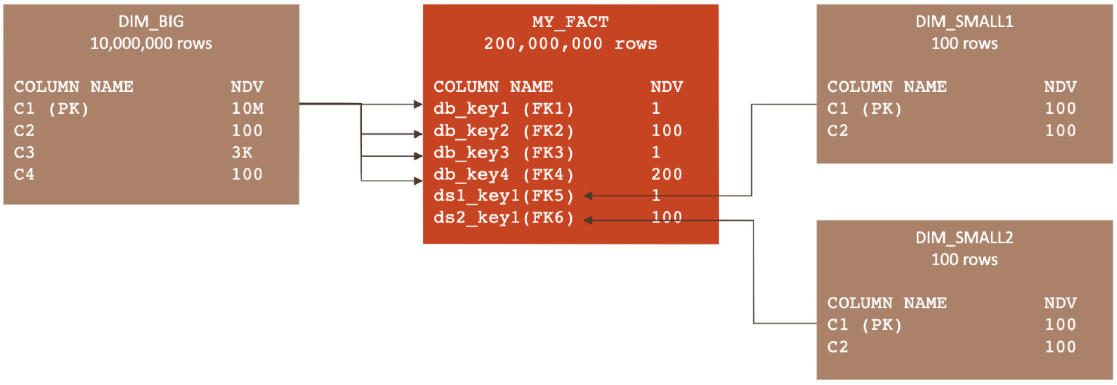
Replacing multiple joins to a large dimension table with joins to much smaller semi-join materialized views significantly improves the run-time and resource consumption for complex


Copyright © 2001, 2025, Oracle and/or its affiliates.

analytical operations. Semi-Join Materialized Views are especially beneficial when the number of applicable dimension keys derived from the large unified dimension table (through semi-join) is small.

Here is a candidate for this optimization:

**Figure 12-8 Candidate Dimension Table for Semi-Join Materialized View Rewrite**




Note that this is an extremely large dimension table and the NDVs of the foreign key join columns are small. A query against the fact table in this case can have a very long execution time. For example, the query below may take a few minutes to execute:

```
SELECT DB1.C4 A, DB2.C4 B, DB3.C4 C, DB4.C4 D, DS1.C2 E, DS2.C2 F,
SUM(M1) H, MAX(M2) I
FROM MY_FACT F
LEFT OUTER JOIN DIM_BIG DB1 ON F.DB_KEY1=DB1.C1
LEFT OUTER JOIN DIM_BIG DB2 ON F.DB_KEY2=DB2.C1
LEFT OUTER JOIN DIM_BIG DB3 ON F.DB_KEY3=DB3.C1
LEFT OUTER JOIN DIM_BIG DB4 ON F.DB_KEY4=DB4.C1
LEFT OUTER JOIN DIM_SMALL1 DS1 ON F.DS1_KEY1=DS1.C1
LEFT OUTER JOIN DIM_SMALL2 DS2 ON F.DS2_KEY1=DS2.C1
GROUP BY DB1.C4, DB2.C4, DB3.C4, DB4.C4, DS1.C2, DS2.C2;
```

Using the original query against the `MY_FACT` table shown in the example above, we use iterative materialized view rewrite to create a materialized view for each join predicate between `MY_FACT` and `DIM_BIG`. For example:

```
CREATE MATERIALIZED VIEW MV1
ENABLE QUERY REWRITE AS
SELECT * FROM DIM_BIG
WHERE C1 IN
(SELECT db_key1 from MY_FACT);
```

A materialized view like this is also be created for `db_key2`, `db_key3`, and `db_key4` in the `MY_FACT` example.. The query is then rewritten as follows:

```
SELECT MV1.C4 A,MV2.C4 B,MV3.C4 C,MV4.C4 D,SUM(F.M1),SUM(F.M2)
FROM MV1, MV2, MV3, MV4, MY_FACT F
WHERE F.DB_KEY1=MV1.C1(+) AND F.DB_KEY2=MV2.C1(+) AND F.DB_KEY3=MV3.C1(+)
AND F.DB_KEY4=MV4.C1(+) GROUP BY MV1.C4,MV2.C4,MV3.C4,MV4.C4
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

The execution time for this rewrite of the original query is reduced to a few seconds.


Copyright © 2001, 2025, Oracle and/or its affiliates.

13

# Working With Automatic Materialized Views

Materialized views can be created and maintained automatically, starting with Oracle Database Release 21c, .

The Oracle AI Database can automatically create and manage materialized views in order to optimize query performance. With very little or no interaction with the DBA, background tasks monitor and analyze workload characteristics and identifies where materialized views will improve SQL performance. The performance benefit of candidate materialized views is measured in the background (using workload queries) before they are made visible to the workload.

> **Note**
>
> Automatic materialized views support partitioned and non-partitioned base tables. Incremental materialized view refresh is supported. In addition, for partitioned tables, there is support for Partition Change Tracking (PCT) view refresh. To be eligible for PCT-based refresh, partitioned base tables must use either range, list, or composite partitioning. If there is performance advantage, the automatic materialized view recommendations will include a partitioned automatic materialized view based on the partitioning of the base table of the materialized view. The partitioning type supported is auto-list partitioning, which will mirror the partitioning of the fact table.
>
> The automatic materialized view maintenance module decides the type of refresh that is the most beneficial at the time of refresh, and will decide during run time whether to switch from incremental refresh to full refresh.

## 13.1 Overview of Automatic Materialized Views

The database automatically collects workload information, workload queries and query execution statistics. It also maintains and purges the history of the workload. This eliminates a time-consuming DBA task.

Although automatic materialized views can run with minimal DBA interaction, their behavior can be easily adjusted.

This is a summary of automatic materialized view functionality:

- Automatically detects and collects workload query execution statistics. These include buffer-gets, database time, estimated cost, and other statistics.
- Creates candidate materialized views hidden from the database workload and verifies that they will deliver the projected performance benefit. It does this by test executing a sample of workload queries in the background.
- Provides reports detailing performance test results and which materialized views have been implemented.
- Provides automatic materialized view refresh.


Copyright © 2001, 2025, Oracle and/or its affiliates.

The database implements only automatic materialized views whose benefits far outweigh the cost of maintaining them. It does not implement those that provide marginal benefit.

# 13.2 Workload Information Provided by the Object Activity Tracking System

Automatic materialized views use workload information provided by the Object Activity Tracking System (OATS) as part of the automated decision-making processes.

The Object Activity Tracking System (OATS) tracks various activities associated with database objects (starting in Oracle Database 21c.) The automatic materialized view feature is one of the clients of this system. In the case of automatic materialized views, the usage data provided by OATS is one of the inputs into the analysis of cost versus benefit for creating or refreshing a materialized view, as well as in determining the best type of refresh and optimal refresh schedule.

OATS takes periodic snapshots of activity within any number of selected tables. The snapshot for each table captures the number of scans, loads, inserts/updates/deletes, truncations, and partition-related activity within the table from the beginning to the end of the snapshot interval. The DBA can use the `DBMS_ACTIVITY` PL/SQL package to set the OATS capture interval, snapshot retention period, and space limits.

For example, the `DBA_ACTIVITY_TABLE` view shows the usage data captured within each snaphot.

# 13.3 Data Dictionary Views That Provide Information About Automatic Materialized Views and OATS

Oracle AI Database includes data dictionary views that display information about automatic materialized views as well as OATS (Object Activity Tracking System) starting in Oracle Database 21c.

**Views for Monitoring Automatic Materialized Views**

Use the following data dictionary views to check the automatic materialized view configuration and to examine various aspects of automatic materialized views activity:

- `DBA_AUTO_MV_ANALYSIS_ACTIONS`

  Displays information about analysis and tuning tasks, including actions, commands, advisor-specific flags, and command parameters.
- `DBA_AUTO_MV_ANALYSIS_EXECUTIONS`
  Displays information about analysis and tuning executions, including concurrency, degree of parallelism (DOP) requested by the user and actual DOP upon execution finish, status, associated advisor, and informational or error message.
- `DBA_AUTO_MV_ANALYSIS_RECOMMENDATIONS`
  Displays recommendations associated with automatic materialized views.
- `DBA_AUTO_MV_ANALYSIS_REPORT`
  Reports on analyses and recommendations, including task and execution names, sequence number of the journal entry, and message entry in the journal.
- `DBA_AUTO_MV_ANALYSIS_TASK`


Copyright © 2001, 2025, Oracle and/or its affiliates.

Displays analysis details associated with automatic materialized views, including task identifiers and task description, creation and last modification dates, execution data, parent task, status, and other information.

- `DBA_AUTO_MV_CONFIG`
  Displays the current automatic materialized view configuration.

> **Note**
>
> The configuration parameters displayed in this view can be updated with `CONFIGURE` procedure of the `DBMS_AUTO_MV` package.

- `DBA_AUTO_MV_MAINT_REPORT`
  Displays the date, time, and message associated with automatic materialized view maintenance actions.
- `DBA_AUTO_MV_REFRESH_HISTORY`
  Displays the owner name, view name, date, start and end time, elapsed time, status, and error number (if an error occurred) for each automatic materialized view refresh.
- `DBA_AUTO_MV_VERIFICATION_REPORT`
  Displays the task name, execution name, and message associated with verifications.
- `DBA_AUTO_MV_VERIFICATION_STATUS`
  Displays the owner, start/end timestamps of verifications, SQL tuning sets used, and SQL Performance Analyzer tasks used in each verification.

**Views for Monitoring OATS**

- `DBA_ACTIVITY_CONFIG`
  Displays the current value of the configuration parameters that control OATS.

> **Note**
>
> The configuration parameters displayed in this view can be updated with CONFIGURE procedure of the `DBMS_ACTIVITY` package.

- `DBMS_ACTIVITY_TABLE`
  Describes table activity snapshots that were recently taken by OATS.
- `DBA_ACTIVITY_SNAPSHOT_META`
  Displays information about activity snapshots taken by OATS.
- `DBA_ACTIVITY_MVIEW`
  Describes materialized view activity snapshots that were recently taken by OATS.

> **See Also**
>
> *Oracle Database Reference*


Copyright © 2001, 2025, Oracle and/or its affiliates.

# 13.4 The DBMS_AUTO_MV Package

This package contains procedures for controlling automatic materialized views.

### DBMS_AUTO_MV.CONFIGURE

The DBA can use the `CONFIGURE` procedure of `DBMS_AUTO_MV` to start, stop, and configure automatic materialized views.

**Table 13-1 Configure Procedure Parameters**

| Parameter | Description and Examples |
|---|---|
| `AUTO_MV_MODE` | `IMPLEMENT`: Implements automatic materialized views.<br>`OFF`: Turns off automatic materialized views.<br>REPORT ONLY: Report-only mode.<br><br>`exec dbms_auto_mv.configure('AUTO_MV_MODE', 'IMPLEMENT');`<br>`exec dbms_auto_mv.configure('AUTO_MV_MODE', 'OFF');`<br>`exec dbms_auto_mv.configure('AUTO_MV_MODE', 'REPORT ONLY');` |
| `AUTO_MV_MAINT_TASK` | `ENABLE`: Activates the task performing the maintenance (refreshes, validations, and cleanup).<br>`DISABLE`: Deactivates the task performing the maintenance.<br>`CLEANUP_AND_DISABLE`: Drops all automatic materialized views and deactivates the task. If automatic materialized views maintenance is in progress, then maintenance is allowed to finish before the task is deactivated.<br><br>`exec dbms_auto_mv.configure('AUTO_MV_MAINT_TASK', 'ENABLE');`<br>`exec dbms_auto_mv.configure('AUTO_MV_MAINT_TASK', 'DISABLE');`<br>`exec dbms_auto_mv.configure('AUTO_MV_MAINT_TASK', 'CLEANUP_AND_DISABLE');` |


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Table 13-1 (Cont.) Configure Procedure Parameters**

| Parameter | Description and Examples |
|---|---|
| AUTO_MV_SPACE_BUDGET | Specifies the percentage of space budgeted for implementing automatic materialized views within the tablespace where those views were created. This is a percentage of the total space used by all automatic materialized views and associated indexes within the tablespace.<br>A condition on the enforcement of AUTO_MV_SPACE_BUDGET is the value of AUTO_MV_DEFAULT_TABLESPACE:<br>• If AUTO_MV_DEFAULT_TABLESPACE is not defined (NULL), then automatic materialized views are created on the tablespace of the view's parent object (which is the largest FACT table in the view's definition). In this case, the budget defined by AUTO_MV_SPACE_BUDGET is enforced within that tablespace.<br>• If AUTO_MV_DEFAULT_TABLESPACE is defined, then automatic materialized views are created in the designated default tablespace. In this case, the budget set by AUTO_MV_SPACE_BUDGET is ignored.<br>If the budget is exceeded (possibly because of the growth of automatic materialized views), then the least-used automatic materialized view is dropped.<br>The value is an integer from 1 to 100. The default is 67 (67% of the total volume of the tablespace).<br>`exec dbms_auto_mv.configure('AUTO_MV_SPACE_BUDGET', 15);` |
| AUTO_MV_DEFAULT_TABLESPACE | Specifies the default tablespace for the creation of automatic materialized views. Possible values are the name of a valid temporary tablespace or NULL (the default). In the case of NULL, new automatic materialized view is created in the default tablespace of the owner of the parent object. If the view has more than one parent object, such as materialized views defined on multiple base tables, then the default tablespace of the owner of largest base table is selected.<br>If the value is changed dynamically, the change takes effect the next time automatic materialized view recommendations are implemented.<br>`exec dbms_auto_mv.configure('AUTO_MV_DEFAULT_TABLESPACE', 'MYTABLESPACE');`<br>`exec dbms_auto_mv.configure('AUTO_MV_DEFAULT_TABLESPACE');` |


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Table 13-1 (Cont.) Configure Procedure Parameters**

| Parameter | Description and Examples |
|---|---|
| AUTO_MV_TEMP_TABLESPACE | Specifies the temporary tablespace used for creation or refresh of automatic materialized views. Possible values are the name of a valid temporary tablespace or NULL. In the case of NULL, the tablepace is assigned to the owner of the largest parent object of the automatic materialized views. The default is NULL.<br><br>exec dbms_auto_mv.configure('AUTO_MV_TEMP_TABLESPACE','TEMP2');<br>exec dbms_auto_mv.configure('AUTO_MV_TEMP_TABLESPACE'); |
| AUTO_MV_RETENTION | Specifies the number of days automatic materialized views can continue to exist without being queried. If an automatic materialized view remains unused beyond this retention time, it is automatically dropped.<br>Possible values are any integer between 1 and 373. The default is 33 days.<br><br>exec dbms_auto_mv.configure('AUTO_MV_RETENTION', 365); |
| AUTO_MV_ANALYZE_REPORT_RETENTION | AUTO_MV_ANALYZE_REPORT_RETENTION Specifies the maximum number of days to retain analysis and recommendation history. Possible values are any integer from 0 to 90. A value of 0 means no history is maintained. The default is 31 days.<br><br>exec dbms_auto_mv.configure('AUTO_MV_ANALYZE_REPORT_RETENTION', 60); |
| AUTO_MV_VERIFY_REPORT_RETENTION | Specifies the maximum number of days to retain verification history. Possible values are any integer from 0 to 90. The value 0 specifies that no verification history will be maintained. The default is 31 days.<br><br>exec dbms_auto_mv.configure('AUTO_MV_VERIFY_REPORT_RETENTION', 7); |
| AUTO_MV_MAINT_REPORT_RETENTION | Specifies the maximum number of days to retain history of automatic materialized view maintenance (refreshes) in the DBA_AUTO_MV_REFRESH_* dictionary tables. Possible values are any integer from 0 to 90. The value 0 specifies that no refresh history will be maintained. The default is 31 days.<br><br>exec dbms_auto_mv.configure('AUTO_MV_MAINT_REPORT_RETENTION', 14); |


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Table 13-1 (Cont.) Configure Procedure Parameters**

| Parameter | Description and Examples |
|---|---|
| AUTO_MV_ANALYZE_WORKLOAD_WINDOW | Specifies the maximum number of hours to investigate queries from the latest snapshots and make recommendations. Possible values are any integer between from 1 to 8760. The default is 24 hours.<br><br>`exec dbms_auto_mv.configure('AUTO_MV_ANALYZE_WORKLOAD_WINDOW', 48);` |
| AUTO_MV_ANALYZE_WORKLOAD_MIN_TIME | Specifies the minimum time in seconds for a query to be considered for automatic materialized views recommendation. Queries below this threshold are not considered for recommendations. Possible values are any integer from 0 to 3600. The default is 120 seconds.<br><br>`exec dbms_auto_mv.configure('AUTO_MV_ANALYZE_WORKLOAD_MIN_TIME', 1800);` |
| AUTO_MV_SCHEMA | Specifies a schema to be either included or excluded during the creation of automatic materialized views. The schema is added to the *inclusion list* or the *exclusion list* in the configuration. Initially, both lists are empty and automatic materialized views can be created in all the schemas in a database where automatic materialized views are enabled. You can build the inclusion and exclusion lists by calling AUTO_MV_SCHEMA multiple times.<br>The boolean ALLOW determines if the schema is added to the inclusion list (TRUE) or to the exclusion list (FALSE). The default is TRUE. During workload processing, any query that does not contain a reference at least one table in a schema on the inclusion list is not analyzed and not auto tuned. It is not factored into recommendations and verifications. Likewise, if a query references a table in a schema on the exclusion list, that query is excluded from processing.<br><br>`exec dbms_auto_mv.configure('AUTO_MV_SCHEMA', 'SCHEMA_A');`<br>`exec dbms_auto_mv.configure('AUTO_MV_SCHEMA', 'SCHEMA_B', FALSE);`<br><br>To enable or disable processing of all schemas, you can specific the schema as NULL. This either enables or disables all of them, depending on the value of ALLOW.<br><br>`exec dbms_auto_mv.configure('AUTO_MV_SCHEMA','', TRUE);` |


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Table 13-1 (Cont.) Configure Procedure Parameters**

| Parameter | Description and Examples |
|---|---|
| AUTO_MV_APP_MODULE | Specifies application modules to include or exclude from the creation of automatic materialized views.<br><br>`exec dbms_auto_mv.configure('AUTO_MV_APP_MODULE', 'MODULE1', TRUE);`<br>`exec dbms_auto_mv.configure('AUTO_MV_APP_MODULE', 'MODULE1', FALSE);`<br>`exec dbms_auto_mv.configure('AUTO_MV_APP_MODULE', 'MODULE1');` |

### DBMS_AUTO_MV.DROP_AUTO_MVS

This procedure drops an automatic materialized view. It can be executed only by users who have the DBA role.

| Parameter | Description |
|---|---|
| OWNER | The name of the owner of the automatic materialized view. |
| MV_NAME | The name of the automatic materialized view. |
| ALLOW_RECREATE | Allow the materialized view to be recreated if necessary. Optional. |

Note that if OWNER is specified and MV_NAME is set to NULL, then all automatic materialized views owned by OWNER are dropped.

```
exec dbms_auto_mv.drop_auto_mvs('SH', 'AUTO_MV$$_G2MKPB9SA1FB7', TRUE);
exec dbms_auto_mv.drop_auto_mvs('SH', 'AUTO_MV$$_G2MKPB9SA1FB7');
exec dbms_auto_mv.drop_auto_mvs('SH', '');
exec dbms_auto_mv.drop_auto_mvs('SH', '', TRUE);
```

### DBMS_AUTO_MV.RECOMMEND

DBMS_AUTO_MV.RECOMMEND generates automatic materialized recommendations based on a given SQL tuning set. This API enables you to manually run automatic materialized view analysis and verification from a command line (instead of through an Automatic SQL Tuning task). You set the workload start and end time and determine whether this execution results in a report only, or an actual implementation. There is no default time limit for the workload window.

Execution of this API requires the DBA role.

> **Note**
>
> Automatic materialized view configuration parameters can influence the results of DBMS_AUTO_MV.RECOMMEND. For example, the analysis and recommendations of this API are restricted to the schemas specified by the configuration parameter AUTO_MV_SCHEMA.


Copyright © 2001, 2025, Oracle and/or its affiliates.

| Parameter | Description |
|---|---|
| STS_OWNER | The name of the owner of the SQL tuning set.<br>Default: SYS. |
| STS_NAME | The name of the SQL tuning set.<br>Default: SYS_AUTO_STS. |
| WORKLOAD_START_TIME | Start time for the workload window.<br>Default: WORKLOAD_END_TIME minus 24 hours. |
| WORKLOAD_END_TIME | End time for the workload window.<br>Default: The current timestamp. |
| AUTO_MV_MODE | REPORT ONLY (recommendations only) or IMPLEMENT.<br>Default: REPORT ONLY |

**Example:**

Generate and report recommendations using SYS_AUTO_STS for the past 24 hours. Note that the default behavior is REPORT_ONLY, which means that no automatic materialized view will be implemented.

```
var exec_name varchar2(200);
begin
    :exec_name := dbms_auto_mv.recommend();
end;
SELECT * FROM DBA_AUTO_MV_ANALYSIS_RECOMMENDATIONS
    WHERE exec_name = :exec_name;
```

### DBMS_AUTO_MV.REFRESH

The DBMS_AUTO_MV.RECOMMEND API enables you force a refresh of all stale automatic materialized views. The stale automatic materialized views are unconditionally refreshed in descending order, based on their verified query rewrite benefit values. There are no parameters. This routine can be executed only by users with the DBA role.

```
exec dbms_auto_mv.dbms_auto_refresh();
```

### DBMS_AUTO_MV.REPORT_ACTIVITY

The DBMS_AUTO_MV.REPORT_ACTIVITY This API generates a report on automatic materialized view activities and usage within a specified time window. The report is returned as a CLOB.

| Parameter | Description |
|---|---|
| ACTIVITY_START | The start of the time window.<br>Default: SYSTIMESTAMP -1. |
| ACTIVITY_END | The end of the time window.<br>Default: SYSTIMESTAMP. |
| TYPE | The format of the report. 'TEXT', 'HTML', and 'XML' are supported.<br>Default: 'TEXT'. |


Copyright © 2001, 2025, Oracle and/or its affiliates.

| Parameter | Description |
|---|---|
| SECTION | The section or sections covered by the report. The value can be any combination of: SUMMARY, MV_DETAILS, QUERY_DETAILS, VERIFICATION_DETAILS or ALL.<br>Default: 'ALL'.<br><br>**Note**<br>Use the “+” or “-“ operator to concatenate a single string that includes or excludes sections of the report. This is shown in one of the examples below. |
| LEVEL | The level of detail in the report: BASIC, TYPICAL or ALL.<br>Default: 'TYPICAL'. |

**Examples:**

Generate a report on all automatic materialized view activities. Output the report in HTML format:

```
select dbms_auto_mv.report_activity(type => ‘HTML') from dual;
```

Generate a report on all automatic materilalized view activities. Exclude the verification details. Output the report in XML format.

```
select dbms_auto_mv.report_activity(type => ‘XML', section => ‘ALL-
VERIFICATION_DETAILS’) from dual;
```

### DBMS_AUTO_MV.REPORT_LAST_ACTIVITY

The DBMS_AUTO_MV.REPORT_LAST_ACTIVITY API generates a report on the most recent automatic materialized view activities and usage.

| Parameter | Description |
|---|---|
| TYPE | The format of the report. 'TEXT', 'HTML', and 'XML' are supported.<br>Default: 'TEXT'. |


Copyright © 2001, 2025, Oracle and/or its affiliates.

| Parameter | Description |
|---|---|
| SECTION | The section or sections covered by the report. The value can be any combination of: SUMMARY, MV_DETAILS, QUERY_DETAILS, VERIFICATION_DETAILS or ALL. Default: 'ALL'. |
| LEVEL | The level of detail in the report: BASIC, TYPICAL or ALL. Default: 'TYPICAL'. |

> **Note**
>
> Use the “+” or “-“ operator to concatenate a single string that includes or excludes sections of the report. See the examples below.

**Examples:**

Generate a comprehensive report of the most recent activity, at the typical level of detail. Output the report in text format (the default). Note that both of these statements return the same results.

```
select dbms_auto_mv.report_last_activity('TEXT', 'ALL', ‘TYPICAL’) from dual;


select dbms_auto_mv.report_last_activity() from dual;
```

Generate a report of the most recent activity that includes only the summary and the details of the materialized view. Report at the maximum level of detail. Output in XML format:

```
select dbms_auto_mv.report_last_activity(‘XML', 'SUMMARY+MV_DETAILS', 'ALL')
from dual;
```

Generate a report of the most recent activity at the basic level of detail. Exclude the verification details. Output in HTML format.

```
select dbms_auto_mv.report_last_activity(‘XML', 'ALL-VERIFICATION_DETAIL',
'BASIC') from dual;
```

**For More Information**

> **See Also**
>
> The *Oracle Database PL/SQL Packages and Types Reference*.


Copyright © 2001, 2025, Oracle and/or its affiliates.

# 13.5 The DBMS_ACTIVITY Package

The `DBMS_ACTIVITY` PL/SQL package contains functions and procedures for configuring Object Activity Tracking System (OATS) information collection and management. Data collected by OATS is used in analyses performed by automatic materialized views.

DBAs can use the `DBMS_ACTIVITY.CONFIGURE` procedure to control three OATS parameters within a specific database.

- `ACTIVITY_INTERVAL`
  The interval between snapshots.

  ```
  exec dbms_activity.configure('ACTIVITY_INTERVAL_MINUTES',30)
  ```

- `ACTIVITY_RETENTION_DAYS`
  How long snapshots are saved.

  ```
  exec dbms_activity.configure('ACTIVITY_RETENTION_DAYS',60)
  ```

- `ACTIVITY_SPACE_PERCENT`
  How much of available space is reserved for snapshots.

  ```
  exec dbms_activity.configure('ACTIVITY_SPACE_PERCENT',10)
  ```

> **Note**
>
> OATS is intended to be self-managing and the default configuration is recommended, particularly if the automatic materialized views feature is used.

> **See Also**
>
> The *PL/SQL Packages and Types Reference*.


Copyright © 2001, 2025, Oracle and/or its affiliates.

14

# Attribute Clustering

Attribute clustering is a table-level directive that clusters data in close physical proximity based on the content of certain columns. Storing data that logically belongs together in close physical proximity can greatly reduce the amount of data to be processed and can lead to better performance of certain queries in the workload.

This chapter includes the following sections:

- About Attribute Clustering
- Attribute Clustering Operations
- Viewing Attribute Clustering Information

## 14.1 About Attribute Clustering

An attribute-clustered table stores data in close proximity on disk in an ordered way based on the values of a certain set of columns in the table or a set of columns in the other tables.

You can cluster according to the linear order of specified columns or by using a function that permits multi-dimensional clustering (also known as interleaved clustering). Attribute clustering improves the effectiveness of zone maps, Exadata Storage Indexes, and In-memory min/max pruning. Queries that qualify clustered columns will access only the clustered regions. When attribute clustering is defined on a partitioned table, the clustering applies to all partitions.

Attribute clustering is a directive property of a table. It is not enforced for every DML operation, but only affects direct-path insert operations, data movement, or table creation. Conventional DML operations on the table are not affected by attribute clustering. This means that whatever is done to cluster the data is an operation that is only done on the current working data set. This is in contrast to a manually-applied `ORDER BY` command, such as what occurs as part of a CTAS operation.

This section contains the following topics:

- Methods of Clustering Data
- Types of Attribute Clustering
- Example: Attribute Clustered Table
- Guidelines for Using Attribute Clustering
- Advantages of Attribute-Clustered Tables
- About Defining Attribute Clustering for Tables
- About Specifying When Attribute Clustering Must be Performed

### 14.1.1 Methods of Clustering Data

You can cluster data using the following methods:

- Clustering based on one or more columns of the table on which attribute clustering is defined.


Copyright © 2001, 2025, Oracle and/or its affiliates.

- Clustering based on one or more columns that are joined with the table on which attribute clustering is defined. Clustering based on joined columns is called **join attribute clustering**. The tables should be connected through a primary key-foreign key relationship but foreign keys do not have to be enforced.

  Because star queries typically qualify dimension hierarchies, it can be beneficial if fact tables are clustered based on columns (attributes) of one or more dimension tables. With join attribute clustering, you can join one or more dimension tables with a fact table and then cluster the fact table data by dimension hierarchy columns. To cluster a fact table on columns from one or more dimension tables, the join to the dimension tables must be on a primary or unique key of the dimension tables. Join attribute clustering in the context of star queries is also known as *hierarchical clustering* because the table data is clustered by dimension hierarchies, each made up of an ordered list of hierarchical columns (for example, the `nation`, `state`, and `city` columns forming a `location` hierarchy).

  **Note:** In contrast with Oracle Table Clusters, join attribute clustered tables do not store data from a group of tables in the same database blocks. For example, consider an attribute clustered table `sales` joined with a dimension table `products`. The `sales` table will only contain rows from the `sales` table, but the ordering of the rows will be based on the values of columns joined from `products` table. The appropriate join will be executed during data movement, direct path insert and CTAS operations.

## 14.1.2 Types of Attribute Clustering

Attribute clustering is a user-defined table directive that provides data clustering on one or more columns in a table. The directives can be specified when the table is created or modified.

Oracle AI Database provides the following types of attribute clustering:

- Attribute Clustering with Linear Ordering
- Attribute Clustering with Interleaved Ordering

Regardless of the type of attribute clustering used, you can either cluster data based on a single table or by joining multiple tables (join attribute clustering).

### 14.1.2.1 Attribute Clustering with Linear Ordering

Linear ordering stores the data according to the order of specified columns. This is the default type of clustering. For example, linear ordering on the `(prod_id, channel_id)` columns of the table `SALES` sorts the data by `prod_id` first and then by `channel_id`. The sorted data is stored on disk with the data for clustered columns being in close proximity.

Linear ordering can be defined on single tables or multiple tables that are connected through a primary key-foreign key relationship.

Use the `CLUSTERING ... BY LINEAR ORDER` directive to perform attribute clustering based on the order of specified columns.

Attribute clustering based on linear ordering of columns is best used in the following scenarios:

- Queries specify the prefix of columns included in the `CLUSTERING` clause in a single table

  For example, if queries on `sales` often specify either a customer ID or a combination of customer ID and product ID, then you could cluster data in the table using the column order `cust_id`, `prod_id`.

- Columns used in the `CLUSTERING` clause have an acceptable level of cardinality


Copyright © 2001, 2025, Oracle and/or its affiliates.

The potential data reduction that can be obtained in the scenarios described in "Advantages of Attribute-Clustered Tables" increases in direct proportion to the data reduction obtained from a predicate on a column.

Linear clustering combined with zone maps is very effective in I/O reduction.

### 14.1.2.2 Attribute Clustering with Interleaved Ordering

Interleaved ordering uses a special multidimensional clustering technique based on Z-order curve fitting. It maps multiple column attribute values (multidimensional data points) to a single one-dimensional value while preserving the multidimensional locality of column values (data points). Interleaved ordering is supported on single tables or multiple tables. Unlike linear ordering, this method does not require the leading columns of the clustering definition to be present to achieve I/O pruning benefits for the scenarios described in "Advantages of Attribute-Clustered Tables".

Columns can be used individually or grouped together into column groups. Each individual column or column group will be used to constitute one of the multidimensional data points in the cluster. Grouped columns are bracketed by '(..)', and must follow the dimensional hierarchy from the coarsest to the finest level of granularity. For example, (`product_category`, `product_subcategory`).

Use the `CLUSTERING ... BY INTERLEAVED ORDER` directive to perform clustering by interleaved ordering.

Interleaved clustering is most beneficial for SQL operations with varying predicates on multiple columns. This is often the case for star queries against a dimensional model, where the query predicates are on dimension tables and the number of predicates vary. Using interleaved join attribute clustering is most common in environments where the fact table is clustered based on columns from the dimension tables. The columns from a dimension table will likely contain a hierarchy, for example, the hierarchy of a product category and sub-category. In this case, clustering of the fact table would occur on dimension columns forming a hierarchy. This is the reason join attribute clustering for star schemas is sometimes referred to as hierarchical clustering. For example, if queries on `sales` specify columns from different dimensions, then you could cluster data in the `sales` table according to columns in these dimensions.

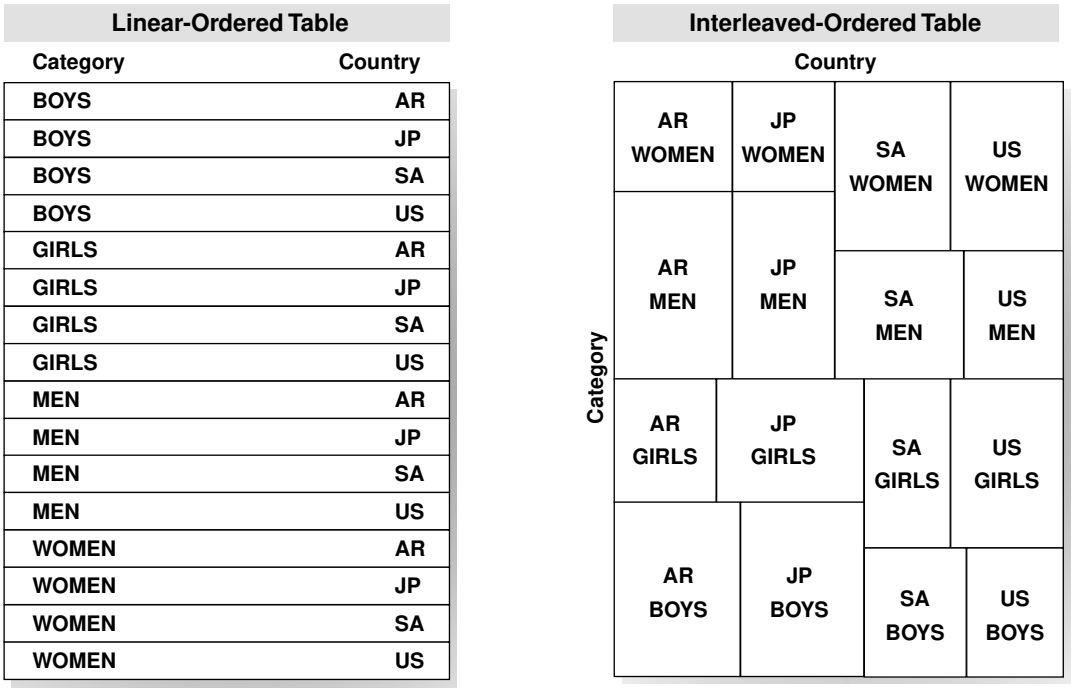
Interleaved clustering combined with zone maps is very effective in I/O pruning for star schema queries. In addition, it enables you to provide a very efficient I/O pruning for queries using zone maps, and enhances compression because the same column values are close to each other and can be easily compressed.

## 14.1.3 Example: Attribute Clustered Table

An example of how a clustered table looks is illustrated in Figure 14-1. Assume you have a table `sales` with columns `(category, country)`. The table on the left is clustered using linear ordering, and the table on the right is clustered using interleaved ordering. Observe that, in the interleaved-ordered table, there are contiguous regions on disk that contain data with a given category and country.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Figure 14-1 Attribute-Clustered Tables**




## 14.1.4 Guidelines for Using Attribute Clustering

The following are some considerations when defining an attribute clustered table:

- Use attribute clustering in combination with zone maps to facilitate zone pruning and its associated I/O reduction.
- Consider large tables that are frequently queried with predicates on medium to low cardinality columns.
- Consider fact tables that are frequently queried by dimensional hierarchies.
- For a partitioned table, consider including columns that correlate with partition keys (to facilitate zone map partition pruning).
- For linear ordering, list columns in prefix-to-suffix order.
- Group together columns that form a dimensional hierarchy. This constitutes a column group. Within each column group, list columns in order of coarsest to finest granularity.
- If there are more than four dimension tables, include the dimensions that are most commonly specified with filters. Limit the number of dimensions to two or three for better clustering effect.
- Consider using attribute clustering instead of indexes on low to medium cardinality columns.
- If the primary key of a dimension table is composed of dimension hierarchy values (for example, the primary key is made up of year, quarter, month, day values), make the corresponding foreign key as clustering column instead of dimension hierarchy.

## 14.1.5 Advantages of Attribute-Clustered Tables

- Eliminates storage costs associated with using indexes


Copyright © 2001, 2025, Oracle and/or its affiliates.

- Enables the accessing of clustered regions rather than performing random I/O or full table scans when used in conjunction with zone maps
- Provides I/O reduction when used in conjunction with any of the following:
  - Oracle Exadata Storage Indexes
  - Oracle In-memory min/max pruning
  - Zone maps

  Attribute clustering provides data clustering based on the attributes that are used as filter predicates. Because both Exadata Storage Indexes and Oracle In-memory min/max pruning track the minimum and maximum values of columns stored in each physical region, clustering reduces the I/O required to access data.

  I/O pruning using zone maps can significantly reduce I/O costs and CPU cost of table scans and index scans.
- Enables clustering of fact tables based on dimension columns in star schemas

  Techniques such as traditional table clusters do not provide for ordering by columns of other tables. In star schemas, most queries qualify dimension tables and not fact tables, so clustering by fact table columns is not effective. Oracle AI Database supports clustering on columns in dimension tables.
- Improves data compression ratios and in this way indirectly improves table scan costs

  Compression can be improved because, with clustering, there is a high probability that clustered columns with the same values are close to each other on disk, hence the database can more easily compress them.
- Minimizes table lookup and single block I/O operations for index range scan operations when the attribute clustering is on the index selection criteria.
- Enables I/O reduction in OLTP applications for queries that qualify a prefix in and use attribute clustering with linear order
- Enables I/O reduction on a subset of the clustering columns for attribute clustering with interleaved ordering

  If table data is ordered on multiple columns, as in an index-organized table, then a query must specify a prefix of the columns to gain I/O savings. In contrast, a `BY INTERLEAVED` table permits queries to benefit from I/O pruning when they specify columns from multiple tables in a non-prefix order.

## 14.1.6 About Defining Attribute Clustering for Tables

Attribute clustering information is part of the table metadata. You can define attribute clustering for a table either when table is first created or subsequently, by altering the table definition.

Use the `CLUSTERING` clause of the `CREATE TABLE` statement to define attribute clustering for a table. The type of attribute clustering is specified by including `BY LINEAR ORDER` or `BY INTERLEAVED ORDER`.

**See Also**

- "Creating Attribute-Clustered Tables with Linear Ordering"
- "Creating Attribute-Clustered Tables with Interleaved Ordering"


Copyright © 2001, 2025, Oracle and/or its affiliates.

If attribute clustering was not defined when the table was created, you can modify the table definition and add clustering. Use the `ALTER TABLE ... ADD CLUSTERING` statement to define attribute clustering for an existing table.

> **See Also**
>
> "Adding Attribute Clustering to an Existing Table"

### 14.1.7 About Specifying When Attribute Clustering Must be Performed

Performing clustering may be expensive because it involves reorganization of the table and clustering data during DML operations. Oracle AI Database does not enforce the clustering of data on conventional DML, conventional insert, update, and merge.

Clustering can be performed in two ways. The first is to automatically perform clustering for certain DML operations on the table. This is done by defining, as part of the table metadata, the operations for which clustering is triggered. The second is to explicitly specify that clustering must be performed as described in "Using Hints to Control Attribute Clustering for DML Operations" and "Overriding Table-level Settings for Attribute Clustering During DDL Operations". In this case, you can perform clustering for a table even if its metadata definition does not include clustering.

As part of the table definition, you can specify that attribute clustering must be performed when the following operations are triggered:

- Direct-path insert operations

  Set the `ON LOAD` option to `YES` to specify that attribute clustering must be performed during direct-path insert operations.

- Data movement operations

  Set the `ON DATA MOVEMENT` option to `YES` to specify clustering must be performed during data movement operations. This includes online table redefinition and the following partition operations: `MOVE`, `MERGE`, `SPLIT`, and `COALESCE`.

The `ON LOAD` and `ON DATA MOVEMENT` options can be included in a `CREATE TABLE` or `ALTER TABLE` statement. If neither `YES ON LOAD` nor `YES ON DATA MOVEMENT` is specified, then clustering is not enforced automatically.

It will serve only as metadata defining natural clustering of the table that may be used later for zone map creation. In this case, it is up to the user to enforce clustering during loads.

> **See Also**
>
> "Adding Attribute Clustering to an Existing Table" for an example on using the `ON LOAD` and `ON DATA MOVEMENT` options

## 14.2 Attribute Clustering Operations

This section describes common tasks involving attribute clustering and includes:

- Privileges for Attribute-Clustered Tables


Copyright © 2001, 2025, Oracle and/or its affiliates.

- Creating Attribute-Clustered Tables with Linear Ordering
- Creating Attribute-Clustered Tables with Interleaved Ordering
- Maintaining Attribute Clustering

## 14.2.1 Privileges for Attribute-Clustered Tables

To define attribute clustering for a table, you must have the `CREATE` or `ALTER` privilege on the table. Additionally, for join attribute clustering, you must also have the `SELECT` or `READ` privilege on the joined table or tables.

> **See Also**
>
> *Oracle Database SQL Language Reference* for syntax and semantics of the `CLUSTERING` clause of `CREATE TABLE`

## 14.2.2 Creating Attribute-Clustered Tables with Linear Ordering

Linear ordering stores the data according to the order of specified columns, equivalent to an `ORDER BY` clause. Linear ordering is supported on columns of a single table or multiple tables in a star schema. Examples of Attribute Clustering with Linear Ordering contains examples of attribute-clustered tables with linear ordering.

> **See Also**
>
> *Oracle Database SQL Language Reference* for information about attribute clustering restrictions

### 14.2.2.1 Examples of Attribute Clustering with Linear Ordering

Example 14-1 and Example 14-2 illustrate linear ordering.

**Example 14-1 Creating a Table with Linear Ordering**

Assume that queries on `sales` often specify either a customer ID or a combination of a customer ID and product ID. You can create an attribute-clustered table so that such queries benefit from I/O reduction for the scenarios described in "Advantages of Attribute-Clustered Tables".

The following statement creates the `sales` table with linear ordering:

```
CREATE TABLE sales (
  prod_id        NUMBER(6) NOT NULL,
  cust_id        NUMBER NOT NULL,
  time_id        DATE NOT NULL,
  channel_id     CHAR(1) NOT NULL,
  promo_id       NUMBER(6) NOT NULL,
  quantity_sold  NUMBER(3) NOT NULL,
  amount_sold    NUMBER(10,2) NOT NULL
)
CLUSTERING
  BY LINEAR ORDER (cust_id, prod_id);
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

This clustered table is useful for queries containing a predicate on `cust_id` or predicates on both `cust_id` and `prod_id`.

**Example 14-2 Creating a Table with Linear Ordering and a Join**

Assume that the `products` dimension table has a unique key or primary key on the `prod_id` column. Other columns in this table include, but are not limited to, `prod_name`, `prod_desc`, `prod_category`, `prod_subcategory`, and `prod_status`. Queries on the `my_sales` fact table often contain one of the following:

- a predicate on `cust_id`
- predicates on `cust_id` and `prod_category`
- predicates on `cust_id`, `prod_category`, and `prod_subcategory`

Defining attribute clustering for the `my_sales` table is useful for queries that contain the predicates included in the `CLUSTERING` clause.

```
CREATE TABLE my_sales (
  prod_id        NUMBER(6) NOT NULL,
  cust_id        NUMBER NOT NULL,
  time_id        DATE NOT NULL,
  channel_id     CHAR(1) NOT NULL,
  promo_id       NUMBER(6) NOT NULL,
  quantity_sold  NUMBER(3) NOT NULL,
  amount_sold    NUMBER(10,2) NOT NULL
)
CLUSTERING
  my_sales JOIN products ON (my_sales.prod_id = products.prod_id)
  BY LINEAR ORDER (cust_id, prod_category, prod_subcategory);
```

> **See Also**
>
> *Oracle Database SQL Language Reference* for syntax and semantics of the `BY LINEAR ORDER` clause

## 14.2.3 Creating Attribute-Clustered Tables with Interleaved Ordering

Interleaved ordering uses a special multidimensional clustering technique similar to a Z-order sort. It is especially beneficial when you have a specific set of predicates that are commonly used most of the time, but do not always use all of them. Interleaved ordering is useful for dimensional hierarchies of star schemas in a data warehouse. "Examples of Attribute Clustering with Interleaved Ordering" contains examples of attribute-clustered tables with interleaved ordering.

> **See Also**
>
> *Oracle Database SQL Language Reference* for information about attribute clustering restrictions

### 14.2.3.1 Examples of Attribute Clustering with Interleaved Ordering

Example 14-3 and Example 14-4 illustrate interleaved ordering.


Copyright © 2001, 2025, Oracle and/or its affiliates.

You can also create an attribute clustered table so that queries benefit from pruning with zone maps. "Creating Zone Maps with Attribute Clustering" contains examples of defining zone maps with attribute clustering.

**Example 14-3 Creating a Table with Interleaved Ordering**

Assume that queries on `sales` often specify either a time ID or a combination of time ID and product ID. You can create `sales` with interleaved attribute clustering using the following command:

```
CREATE TABLE sales (
  prod_id        NUMBER(6) NOT NULL,
  cust_id        NUMBER NOT NULL,
  time_id        DATE NOT NULL,
  channel_id     CHAR(1) NOT NULL,
  promo_id       NUMBER(6) NOT NULL,
  quantity_sold  NUMBER(3) NOT NULL,
  amount_sold    NUMBER(10,2) NOT NULL
)
CLUSTERING
  BY INTERLEAVED ORDER (time_id, prod_id);
```

This clustered table is useful for queries containing one of the following:

- a predicate on `time_id`
- a predicate on `prod_id`
- predicates on `time_id` and `prod_id`

**Example 14-4 Creating a Table with Interleaved Ordering and a Join**

Large data warehouses frequently organize data in star schemas. A dimension table uses a parent-child hierarchy and is connected to a fact table by a foreign key. Clustering a fact table with interleaved ordering enables the database to use a special function to skip values in dimension columns during table scans. Note that clustering does not require an enforced foreign key relationship. However, Oracle AI Database does require primary or unique keys on the dimension tables.

The following command defines attribute clustering using interleaved ordering for the `sales` fact table:

```
CREATE TABLE sales (
  prod_id        NUMBER(6) NOT NULL,
  cust_id        NUMBER NOT NULL,
  time_id        DATE NOT NULL,
  channel_id     CHAR(1) NOT NULL,
  promo_id       NUMBER(6) NOT NULL,
  quantity_sold  NUMBER(3) NOT NULL,
  amount_sold    NUMBER(10,2) NOT NULL
)
CLUSTERING
  sales JOIN products ON (sales.prod_id = products.prod_id)
  BY INTERLEAVED ORDER ((time_id), (prod_category, prod_subcategory));
```

This clustered table is useful for queries containing one of the following:

- a predicate on `time_id`
- a predicate on `prod_category`
- predicates on `prod_category` and `prod_subcategory`
- predicates on `time_id` and `prod_category`


Copyright © 2001, 2025, Oracle and/or its affiliates.

- predicates on `time_id`, `prod_category`, and `prod_subcategory`

> **See Also**
>
> *Oracle Database SQL Language Reference* for information on the `CREATE TABLE` statement and `CLUSTERING` clause

## 14.2.4 Maintaining Attribute Clustering

You can add, drop, and update the attribute clustering definition of a table at any point in time. The modified definition does not affect existing table data, but can only be used as directive for future operations.

The following maintenance operations modify table metadata:

- Adding Attribute Clustering to an Existing Table
- Modifying Attribute Clustering Definitions
- Dropping Attribute Clustering for an Existing Table

You can also override the attribute clustering definitions on a table at runtime. The maintenance operations that influence attribute clustering behavior at runtime are:

- Using Hints to Control Attribute Clustering for DML Operations
- Overriding Table-level Settings for Attribute Clustering During DDL Operations
- Clustering Table Data During Online Table Redefinition

### 14.2.4.1 Adding Attribute Clustering to an Existing Table

When you create a table with clustering, it is created with a zone map by default. You can, however, explicitly prevent this by using `WITHOUT ZONEMAP`. This could be done for several reasons, such as wanting to create a zone map on clustering columns plus additional columns that correlate to clustering columns, or to use specific zone map storage options instead of the defaults.

Use the `ALTER TABLE ... ADD CLUSTERING` command to add attribute clustering to an existing table that does not currently use attribute clustering.

The following command adds attribute clustering to the `SALES` fact table. The modified table will use interleaved clustering that is based on the joined dimension tables `CUSTOMERS` and `PRODUCTS`.

```
ALTER TABLE sales
ADD CLUSTERING sales JOIN customers ON (sales.cust_id = customers.cust_id)
                     JOIN products ON (sales.prod_id = products.prod_id)
    BY INTERLEAVED ORDER ((prod_category, prod_subcategory),
                          (country_id, cust_state_province, cust_city))
    YES ON LOAD YES ON DATA MOVEMENT
    WITHOUT MATERLALIZED ZONEMAP;
```

When you add clustering to a table, the existing data is not clustered. To force the existing data to be clustered, you need to move the content of the table using an `ALTER TABLE...MOVE` statement. You can do this partition by partition.

The following command clusters data in the `sales` table:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
ALTER TABLE sales MOVE PARTITION sales_1995 UPDATE INDEXES ALLOW CLUSTERING;
```

For more information about zone maps, see "About Zone Maps".

### 14.2.4.2 Modifying Attribute Clustering Definitions

Use the `ALTER TABLE ... MODIFY CLUSTERING` statement to modify when attribute clustering is triggered for a table. Modifying clustering definitions does not affect the existing table data. The modified definitions are applicable only to future data movement or direct-path insert operations.

The following command modifies the clustering definition of the `SALES` table and enables clustering during data movement.

```
ALTER TABLE sales MODIFY CLUSTERING YES ON DATA MOVEMENT;
```

You can also modify a table definition and create or drop a zone map that is based on the attribute clustering. The following statement modifies the definition of the `SALES` table and adds a zone map:

```
ALTER TABLE sales MODIFY CLUSTERING WITH MATERIALIZED ZONEMAP;
```

Use the following statement to modify the definition of the attribute-clustered table `SALES` and remove zone maps.

```
ALTER TABLE sales MODIFY CLUSTERING WITHOUT MATERIALIZED ZONEMAP;
```

### 14.2.4.3 Dropping Attribute Clustering for an Existing Table

If attribute clustering is defined for an existing table, use the `ALTER TABLE ... DROP CLUSTERING` statement to remove attribute clustering. Dropping a clustering definition does not have any impact on the existing table data.

The following command removes attribute clustering for the `SALES` table:

```
ALTER TABLE sales DROP CLUSTERING;
```

### 14.2.4.4 Using Hints to Control Attribute Clustering for DML Operations

You can use hints to enforce the use of clustering or to prevent its use during direct-path insert operations. Use the `CLUSTERING` hint to enforce clustering for a table and `NO_CLUSTERING` hint to prevent the use of clustering.

The following command disables attribute clustering while inserting data into the `SALES` table. This table was created with the `YES ON LOAD` option.

```
INSERT /*+ APPEND NO_CLUSTERING */ INTO sales SELECT * FROM external_sales;
```

See "Controlling the Use of Zone Maps" for more information about hints.

### 14.2.4.5 Overriding Table-level Settings for Attribute Clustering During DDL Operations

You can override the attribute clustering definition during data movement DDL operations such as partition maintenance that creates new data segments (split or merge operations) or moving a table, partition, or subpartition. For example, if a table was defined using the `NO ON DATA MOVEMENT` option, then you can cluster data for this table during a data movement operation by using the `ALTER TABLE ... ALLOW CLUSTERING` statement.


Copyright © 2001, 2025, Oracle and/or its affiliates.

The following command allows clustering during data movement for the sales_2010 partition of the SALES tables that was defined using the NO ON DATA MOVEMENT option:

```
ALTER TABLE sales MOVE PARTITION sales_2010 UPDATE INDEXES ALLOW CLUSTERING;
```

Similarly, you can disable clustering during data movement for a table that was defined using the YES ON DATA MOVEMENT option by including the DISALLOW CLUSTERING clause in the ALTER TABLE command that is used to move data.

## 14.2.4.6 Clustering Table Data During Online Table Redefinition

Online table redefinition enables you to modify the logical or physical structure of a table without significantly affecting its availability. The table is accessible to both queries and DML during much of the redefinition process.

You can redefine a table online and add attribute clustering to a table that did not previously use attribute clustering. The DBMS_REDEFINITION package enables you redefine tables online and add attribute clustering to them.

> **See Also**
>
> *Oracle AI Database PL/SQL Packages and Types Reference* for more information about the DBMS_REDEFINITION package

**Example 14-5 Redefining an Attribute-Clustered Table Online**

Assume that you want to redefine the sales table to change the data type of amount_sold from a number to a float, add attribute clustering to the table, and cluster the data during online redefinition.

Use the following steps to redefine the sales table in the SH schema and cluster its data during online table redefinition:

1. Verify that the table can be redefined online by invoking the CAN_REDEF_TABLE procedure of the DBMS_REDEFINITION package.

   The following command verifies that the sales table can be redefined online:

   ```
   exec DBMS_REDEFINITION.CAN_REDEF_TABLE('SH','SALES');
   ```

2. Create the interim table in the SH schema with the desired physical and logical attributes that you want to use for the redefined table.

   The following command creates the interim table sales_interim. The data type of the amount_sold column is binary_double and the CLUSTERING clause specifies how attribute clustering must be performed.

   ```
   CREATE TABLE sales_interim
   (
    PROD_ID        NUMBER(6) PRIMARY KEY,
    CUST_ID        NUMBER NOT NULL,
    TIME_ID        DATE NOT NULL,
    CHANNEL_ID     CHAR(1) NOT NULL,
    PROMO_ID       NUMBER(6),
    QUANTITY_SOLD  NUMBER(3) NOT NULL,
    AMOUNT_SOLD    binary_double
   )
   CLUSTERING sales_interim JOIN customers ON
             (sales_interim.cust_id = customers.cust_id)
   ```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
                    JOIN products ON (sales_interim.prod_id = products.prod_id)
    BY INTERLEAVED ORDER ( (prod_category, prod_subcategory),
                           (country_id, cust_state_province, cust_city));
```

3. Start the online table redefinition process using the `DBMS_REDEFINITON.START_REDEF_TABLE` procedure. The `sales` table is available for queries and DML during this process.

   The following command starts the redefinition process for the `sales` table:

   ```
   exec DBMS_REDEFINITION.START_REDEF_TABLE(uname => 'SH',orig_table  => 'SALES',
   int_table => 'SALES_INTERIM', options_flag => DBMS_REDEFINITION.CONS_USE_ROWID);
   ```

4. Optionally synchronize the interim table with the original table.

   Synchronization is recommended if a large number of DML statements may have been executed on the original table after the redefinition was started. This step reduces the time taken to finish the redefinition process.

   The following command synchronizes the `sales_interim` table with the original `sales` table:

   ```
   exec DBMS_REDEFINITION.SYNC_INTERIM_TABLE('SH', 'SALES', 'SALES_INTERIM');
   ```

5. Complete the online table redefinition using the `DBMS_REDEFINITION.FINISH_REDEF_TABLE` procedure.

   The following command completes the online redefinition of the `sales` table:

   ```
   exec DBMS_REDEFINITION.FINISH_REDEF_TABLE('SH', 'SALES', 'SALES_INTERIM');
   ```

# 14.3 Viewing Attribute Clustering Information

Oracle AI Database provides a set of data dictionary views that contain information about attribute clustering. This section describes how you can use these views to obtain information about attribute clustering.

This section contains the following topics:

- Determining if Attribute Clustering is Defined for Tables
- Viewing Attribute-Clustering Information for Tables
- Viewing Information About the Columns on Which Attribute Clustering is Performed
- Viewing Information About Dimensions and Joins on Which Attribute Clustering is Performed

## 14.3.1 Determining if Attribute Clustering is Defined for Tables

The `CLUSTERING` column in the views `DBA_TABLES`, `USER_TABLES`, and `ALL_TABLES` specifies if attribute clustering is defined for the tables. The `CLUSTERING` column displays `YES` if attribute clustering is defined for the table and `NO` otherwise.

The following query displays the names of tables in the `SH` schema and indicates if they use attribute clustering.

```
SELECT TABLE_NAME, CLUSTERING FROM DBA_TABLES WHERE OWNER='SH';

TABLE_NAME      CLUSTERING
-----------    ------------
SALES             YES
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
PRODUCTS          NO
MY_SALES          YES
```

## 14.3.2 Viewing Attribute-Clustering Information for Tables

Use one of the following data dictionary views to obtain details about attribute clustering for tables:

- DBA_CLUSTERING_TABLES to describe all attribute-clustered tables in the database
- ALL_CLUSTERING_TABLES to describe attribute-clustered table accessible to the user
- USER_CLUSTERING_TABLES to describe attribute-clustered tables owned by the user

The following query displays details about the attribute clustering for the SALES table. The details include the type of attribute clustering and the operations for which clustering is enabled for the table. The output has been formatted to fit on the page.

```
SELECT owner, table_name, clustering_type, on_load, on_datamovement, with_zonemap
FROM DBA_CLUSTERING_TABLES WHERE table_name='SALES';

OWNER    TABLE_NAME CLUSTERING_TYPE  ON_LOAD  ON_DATAMOVEMENT  WITH_ZONEMAP
------   ---------- ---------------  -------- ---------------  -------------
SH       SALES       LINEAR          YES      YES              YES

SELECT owner, table_name, clustering_type, on_load, on_datamovement
FROM DBA_CLUSTERING_TABLES WHERE table_name='SALES';

OWNER    TABLE_NAME CLUSTERING_TYPE  ON_LOAD  ON_DATAMOVEMENT
------   ---------- ---------------  -------- ---------------
SH       SALES       LINEAR          YES      YES
```

## 14.3.3 Viewing Information About the Columns on Which Attribute Clustering is Performed

Use one of the following data dictionary views to obtain information about the columns on which attribute clustering is defined for tables:

- DBA_CLUSTERING_KEYS
- ALL_CLUSTERING_KEYS
- USER_CLUSTERING_KEYS

For example, the data in the table SALES is clustered using linear ordering. Use the following command to display the columns on which table is clustered. The output has been formatted to fit in the page.

```
SELECT detail_owner, detail_name, detail_column, position
FROM DBA_CLUSTERING_KEYS
WHERE table_name='SALES';

DETAIL_OWNER    DETAIL_NAME       DETAIL_COLUMN      POSITION
------------   --------------   -----------------   ---------
SH             SALES             PROD_ID               2
SH             SALES             TIME_ID               1
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

### 14.3.4 Viewing Information About Dimensions and Joins on Which Attribute Clustering is Performed

To view information about the dimension tables by which a fact table is clustered, query the `DBA_CLUSTERING_DIMENSIONS`, `ALL_CLUSTERING_DIMENSIONS`, or `USER_CLUSTERING_DIMENSIONS` data dictionary views.

To view details about the joins of the fact table and dimension tables, query the `DBA_CLUSTERING_JOINS`, `ALL_CLUSTERING_JOINS`, or `USER_CLUSTERING_JOINS` views. The output has been formatted to fit in the page.

The following query displays the dimension tables by which the fact table `SALES` is attribute-clustered.

```
SELECT * FROM DBA_CLUSTERING_DIMENSIONS WHERE table_name='MY_SALES';

OWNER      TABLE_NAME      DIMENSION_OWNER        DIMENSION_NAME
------   --------------   ------------------  --------------------
SH         MY_SALES           SH                   PRODUCTS
```

The following query displays the columns used to join the fact table `my_sales` with dimension table `products`. The output has been formatted to fit in the page.

```
SELECT tab1_owner, tab1_name,tab1_column
FROM DBA_CLUSTERING_JOINS
WHERE table_name='MY_SALES';

TAB1_OWNER       TAB1_NAME     TAB1_COLUMN
-----------   -------------   --------------------
SH               MY_SALES        PROD_ID
```

## 14.4 About Automatic Data Clustering

Oracle AI Database automatically and transparently clusters storage-based data in response to the type of queries used by the application workload. You can make configuration changes to this process.

Automatic data clustering allows the workload to make more efficient use of data access optimizations, such as storage indexes, zone maps, and join zone maps. With little user interaction, the Automatic Data Clustering feature monitors and analyzes workload query execution, recommends the tables to cluster, how to cluster, and what zone maps to create. The impact of clustering is verified before it is implemented.

The steps are as follows:

- Workload capture.
- Workload analysis.
- You generate a clustering and zone map recommendation.
- You verify the recommendation.
- You apply the recommendation.

#### Workload Capture and Analysis

Workload SQL statements can be captured using the automatic SQL tuning set. Over time, workload queries are captured without manual intervention. By default, automatic clustering


Copyright © 2001, 2025, Oracle and/or its affiliates.

analyzes a workload captured in this system-maintained SQL tuning set. Alternatively, automatic clustering can be directed to base and analysis and recommendation on a workload captured in any manually populated SQL tuning set.

**Clustering and Zone Map Recommendation**

Automatic clustering identifies fact and dimension tables and analyzes joins between them using workload SQL statements captured in the automatic SQL tuning set or in a user-specified SQL tuning set. Certain criteria must be met before clustering is considered. For example, fact tables must be a minimum of 64GB, optimizer statistics must be up-to-date, and there should be no existing manual clustering clause or zone map.

The following may be returned to the fact table:

- a single table or join clustering with dimension tables to join
- a clustering method (linear or interleaved clustering) with clustering columns and their ordering
- zone map columns

## 14.4.1 User Controls for Automatic Data Clustering

The workflow for Automatic Data Clustering is to reconfigure the process, generate a recommendation, verify the recommendation, and apply the recommendation.

**Reconfigure the Process**

```
DBMS_AUTO_CLUSTERING.CONFIGURE (
     parameter_name      IN VARCHAR2,
     parameter_value     IN VARCHAR2,
     allow               IN BOOLEAN DEFAULT TRUE);
```

**Example:**

```
exec dbms_auto_clustering.configure('AUTO_CLUSTERING_SCHEMA', 'DW_SCHEMA',
allow=> TRUE);

select parameter_name,parameter_value
from   DBA_AUTO_CLUSTERING_CONFIG
where  parameter_name = 'AUTO_CLUSTERING_SCHEMA_INCLUDE';

PARAMETER_NAME                                                  PARAMETER_VALUE
--------------------------------------------------------------------------------
AUTO_CLUSTERING_SCHEMA_INCLUDE                                  DW_SCHEMA
```

**Generate Recommendations**

```
DBMS_AUTO_CLUSTERING.RECOMMEND_CLUSTERING_METHOD (
     sqlset_owner      IN  VARCHAR2 DEFAULT ‘SYS’,
     sqlset_name       IN  VARCHAR2 DEFAULT ‘SYS_AUTO_STS’,
     table_owner       IN  VARCHAR2 DEFAULT  NULL,
     table_name        IN  VARCHAR2 DEFAULT  NULL,
     report_type       IN  VARCHAR2 DEFAULT ‘TEXT’,
     report_section    IN  VARCHAR2 DEFAULT ‘SUMMARY’,
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
     report_level      IN  VARCHAR2 DEFAULT 'TYPICAL',
     verification      IN  BOOLEAN  DEFAULT  TRUE)
RETURN RAW;
```

**Example:**

```
set serveroutput on
declare
   rec_id raw(16);
begin
     rec_id := dbms_auto_clustering.recommend_clustering_method(table_owner
=> user,
                table_name => 'MYFACT',
                verification=>false,
                report_level=>'ALL', report_section=>'ALL');
     dbms_output.put_line('Rec ID: '||rec_id);
end;
/
Rec ID: F76780956B8AACA1E053624F46641292

set linesize 200
set pagesize 1000
column table_name format a20
select table_name, status
from  dba_auto_clustering_recommendations
where recommendation_id = 'F767C7BD8F92BF4AE053624F4664E9A2';

TABLE_NAME   STATUS
------------ ----------
MYFACT       UNVERIFIED
```

### Verify Recommendations

The verification step measures compares the performance of the application workload with and without the recommended clustering method and zone maps. If performance is improved, the recommendations are marked as `VERIFIED` and they can be accepted. If performance is not improved significantly, the recommendation is marked `REJECTED`.

```
DBMS_AUTO_CLUSTERING.VERIFY_RECOMMENDATION (
     recommendation_id,
     table_owner,
     table_name );
```

**Example:**

```
exec dbms_auto_clustering.verify_recommendation('
F76780956B8AACA1E053624F46641292')

set long 10000
column report format a100
var report clob
begin
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
   :report := dbms_auto_clustering.report_last_activity(
                type=>'TEXT',
                section=>'ALL',
                level=>'ALL');
end;
/

select :report report
from dual;

REPORT
------------------------------------------------------------------------------
---------------------
GENERAL INFORMATION
------------------------------------------------------------------------------
-
 Activity start             : <Date>
 Activity end               : <Date>
 Recommendations generated  : 1
 Recommendation verified    : 1
 Recommendation applied     : 0
------------------------------------------------------------------------------
-

CLUSTERING DETAILS
------------------------------------------------------------------------------
-
1. The following clusterings were created:
------------------------------------------------------------------------------
-
-------------------------------------------------------------------------
| Owner     | Table      | Recommendation ID                | Status   |
Method |
-------------------------------------------------------------------------
| DW_SCHEMA | MYFACT     | F76780956B8AACA1E053624F46641292 | ACCEPTED |
SINGLE |
------------------------------------------------------------------------------
-
* Clustering DDLs

ALTER TABLE "ADHOC"."F" ADD CLUSTERING BY ORDER ( "DW_SCHEMA"."MYFACT"."Y")

------------------------------------------------------------------------------
-

VERIFICATION DETAILS
```

### Apply Recommendations

```
DBMS_AUTO_CLUSTERING.APPLY_RECOMMENDATION (
     table_owner,
     table_name,
     recommendation_id,
     apply_mode );
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Example:**

```
declare
  rec_id raw(16) := null;
  begin
    rec_id := dbms_auto_clustering.get_recommendation('DW_SCHEMA', 'MYFACT');
    dbms_auto_clustering.apply_recommendation(rec_id, 'DW_SCHEMA', 'MYFACT');
  end;
/

select table_name,
       clustering_type,
       on_load,
       on_datamovement,
       valid,
       with_zonemap,
       automatic
from user_clustering_tables;

TABLE_NAME           CLUSTERING_ ON_ ON_ VAL WIT AUT
-------------------- ----------- --- --- --- --- ---
MYFACT               LINEAR      YES YES YES NO  YES

select zonemap_name,
       query,
       automatic
from user_zonemaps;

ZONEMAP_NAME                   QUERY
AUTOMATIC
------------------------------ ------------------------------------------
---------
ACZMAP$_MYFACT                 SELECT SYS_OP_ZONE_ID("MYFACT"."ROWID") …   YES
```

**See Also**

- In the *Oracle Database Reference*, the view DBA_AUTO_CLUSTERING_CONFIG displays the current configuration parameter settings for automatic clustering.. DBA_AUTO_CLUSTERING_RECOMMENDATIONS displays recommendations associated with automatic clustering.
- The *Oracle Database PL/SQL Packages and Types Reference* describes the DBMS_AUTO_CLUSTERING package, which includes the APIs for configuring automatic data clustering as well as generating, verifying, and applying recommendations.


Copyright © 2001, 2025, Oracle and/or its affiliates.

15

# Using Zone Maps

A zone map is a independent access structure that can be built for a table. During table and index scans, zone maps enable you to prune disk blocks of a table and potentially full partitions of a partitioned table based on predicates on the table columns. Zone maps can be used with or without attribute clustering.

This chapter includes the following sections:


- About Zone Maps
- Zone Map Operations
- Refresh and Staleness of Zone Maps
- Performing Pruning Using Zone Maps
- Viewing Zone Map Information


## 15.1 About Zone Maps

A **zone map** is an independent access structure built for a table that stores information about zones of a table. Zone maps enable the database to prune data blocks that cannot satisfy predicates on table columns. A **zone** is a set of a contiguous data blocks on disk.

Traditional zone maps store the minimum and maximum values of a column in a table per disk unit, set of blocks, or extents. If queries qualify on clustering columns, then I/O pruning takes place. Zone maps in an Oracle AI Database store minimum and maximum values of columns for a range of blocks (known as a zone). In addition to performing I/O pruning based on predicates of clustered fact tables, zone maps prune on predicates of dimension tables provided the fact tables are attribute-clustered by the dimension attributes through outer joins with the dimension tables.

You can define at most one zone map on a table. In the case of a partitioned table, there is one zone map for all partitions (and subpartitions). A zone map of a partitioned table also keeps track of the minimum and maximum values per zone, per partition, and per subpartition. Zone map definitions can include minimum and maximum values of dimension columns provided the table has an outer join with the dimension tables.

This section contains the following topics:


- Difference Between Zone Maps and Indexes
- Zone Maps and Attribute Clustering
- Types of Zone Maps
- Benefits of Zone Maps
- Scenarios Which Benefit from Zone Maps
- About Maintaining Zone Maps




Copyright © 2001, 2025, Oracle and/or its affiliates.

## 15.1.1 Difference Between Zone Maps and Indexes

A zone map is analogous to a coarse index structure. However, there are fundamental differences to an index:

- A zone map stores information per zone instead of per row. Thus, it is much more compact than an index.
- A zone map is not actively managed the way an index is kept in sync with DML actions. Thus, even if a zone map has the `REFRESH ON COMMIT` option, it can still be stale within a transaction until commit or rollback occurs.
- A zone map can contain stale information for some zones and fresh information for the rest of the zones, and Oracle AI Database will still use the zone map to perform I/O pruning during the scan of the fact table.

## 15.1.2 Zone Maps and Attribute Clustering

Attribute clustering is not a mandatory pre-requirement for zone maps. Zone maps can be used with or without attribute clustering. Therefore, you can specify attribute clustering without zone maps and build zone maps without clustering on the table.

It is common for data warehousing environments to have reasonably clustered data due to ETL processing, for example, clustering by time columns or by geographical regions. Due to clustering, minimum and maximum values of the columns are more likely to be correlated with consecutive data blocks in the attribute-clustered table, which allows for more efficient pruning using zone maps. Zone maps enable more efficient pruning by taking advantage of data ordering performed by attribute clustering. During table scans and index scans (for example, fetch by rowid), zone maps allow pruning of data blocks that do not satisfy predicates on table columns.

> **See Also**
>
> "About Attribute Clustering" for information about attribute clustering

## 15.1.3 Types of Zone Maps

There are two types of zone maps:

- A basic zone map is defined on a single table and maintains the minimum and maximum values of some columns of this table.
- A join zone map is defined on a table that has an outer join to one or more other tables and maintains the minimum and maximum values of some columns in the other tables; these join conditions are common in primary-detail relationships as well as in star schemas between fact and dimension tables.

  For star queries, multiple dimension tables are joined through PK-FK relationships with a fact table. Here a join zone map maintains the minimum and maximum values of columns from the dimension tables for zones of the fact table.

## 15.1.4 Benefits of Zone Maps

- Enables I/O reduction during sequential or index scans of tables or table partitions


Copyright © 2001, 2025, Oracle and/or its affiliates.

- Enables partition pruning based on non-key columns for partitioned and composite-partitioned tables when zone map columns correlate with the partitioning key
- Enables I/O reduction on a subset of the clustering columns for attribute clustering with interleaved ordering
- Eliminate storage costs associated with using indexes

> **See Also**
>
> *Oracle Database SQL Language Reference*

## 15.1.5 Scenarios Which Benefit from Zone Maps

Using zone maps can be beneficial in the following scenarios:

- Table scans are performed with frequently-used predicates

  Zone maps enable Oracle AI Database to avoid scanning zones that are excluded by column predicates.
- Joins are defined between a fact table and dimension tables with frequently-used predicates on the dimension hierarchy columns

  Fact table rows can be ordered by dimension attribute values, pruning zones that are excluded by predicates on attribute values.
- Columns in partitioned tables contain values that correlate with the partition key

  This will facilitate partition pruning based on “non-key" columns. For example, a table partitioned by date will often have other date columns that correlate well with the partition key or columns that contain sequenced values that change or cycle over time.
- Data clustering is performed on the zone map column values

  Attribute clustering is designed specifically for this purpose. Alternatively, it is appropriate to make use of ordering inherent in the data (for example, time-based column values loaded sequentially or data that is sorted on load).
- Frequent and low cardinality index range scans are performed on tables

  Attribute clustering can be used alone to improve compression factors. Zone maps can be used to improve the efficiency of the index scans by pruning lookups from excluded zones. Alternatively, zone maps can be used to replace indexes.

## 15.1.6 About Maintaining Zone Maps

Zone maps are based on tables and, therefore, any changes to the underlying tables impacts the state of the zone map. Depending on the operation performed on the table, some or all zones of a zone map are impacted. Zone maps affected by changes to the underlying tables require maintenance.

Zone map maintenance consists of one or more of the tasks:

- Checking the validity of affected zone maps
- Tracking the staleness of the affected zone maps
- Refreshing the affected zone maps that have become stale (depending on the refresh mode set for the zone map)


Copyright © 2001, 2025, Oracle and/or its affiliates.

When there is a change in the structure of base tables on which a zone map is based, for example, dropping a table column whose minimum and maximum values are maintained by the zone map, then the zone map becomes **invalid**. Invalid zone maps are not used by queries and Oracle AI Database does not maintain these zone maps. However, when there is a change in the structure of the base tables that does not relate to the zone map, for example, adding a new column to the table, the zone map stays valid but it needs to be compiled. Oracle AI Database automatically compiles the zone map on a subsequent operation such as the use of zone map by a query. Or, you can compile the zone map using the `COMPILE` clause of the `ALTER MATERIALIZED ZONEMAP` command.

> **See Also**
>
> "Compiling Zone Maps"

When there is a change in the data of the underlying tables, the zones that are impacted by these changes are marked **stale**. Only the data in a stale zone map is not current or fresh but its definition is still valid. Oracle AI Database automatically tracks the staleness of zone maps due to different types of operations on the underlying tables. Depending on the type of operation performed on the underlying table the Oracle AI Database will either mark the entire zone map as stale, or some zones in the zone map as stale.

This section contains the following topics:

- Operations that Require Zone Map Maintenance
- Scenarios in Which Zone Maps are Automatically Refreshed

## 15.1.6.1 Operations that Require Zone Map Maintenance

Zone map maintenance is required when the following operations are performed on one or more of the underlying tables:

- DML (insert, delete, update, conventional load).
- Direct-path insert and load.
- Partition Maintenance Operations (`MOVE`, `SPLIT`, `MERGE`, `DROP`, `TRUNCATE`, and `EXCHANGE`), moving table data, and online redefinition of table.

## 15.1.6.2 Scenarios in Which Zone Maps are Automatically Refreshed

The zone map refresh mode determines if Oracle AI Database will automatically refresh the zone maps affected by above operations.

Oracle AI Database performs automatic refresh for zone maps affected by the following:

- DML operations if the refresh mode is `REFRESH ON COMMIT`. Zone maps with `REFRESH ON COMMIT` mode stay transactionally fresh. The refresh is performed when the transaction is committed.
- Direct-path insert or load if the refresh mode is `REFRESH ON LOAD`.

  Zone maps with `REFRESH ON LOAD` can become stale after DML or PMOP operation on underlying table.
- PMOPs (`MOVE`, `SPLIT`, `MERGE`, `DROP`) or table move if the refresh mode is `REFRESH ON DATA MOVEMENT`.


Copyright © 2001, 2025, Oracle and/or its affiliates.

Zone maps with `REFRESH ON DATA MOVEMENT` can become stale after DML, direct-path insert or load, PMOP (`TRUNCATE`, `EXCHANGE`), or online redefinition of underlying table

- Direct-path insert or load, PMOP (`MOVE`, `SPLIT`, `MERGE`, `DROP`) or table move if the refresh mode is `REFRESH ON LOAD DATA MOVEMENT`.

  Zone maps with `REFRESH ON LOAD DATA MOVEMENT` can become stale after DML, PMOP (`TRUNCATE`, `EXCHANGE`), or online redefinition of underlying table.

Oracle AI Database does not perform automatic refresh of zone maps affected by any operation on underlying table if their refresh mode is `REFRESH ON DEMAND`. Zone maps with `REFRESH ON DEMAND` have to be manually refreshed

**See Also**

- "Refresh and Staleness of Zone Maps"
- "Maintaining Zone Maps"

# 15.2 Zone Map Operations

This section describes common tasks involving zone maps, and includes:

- Privileges Required for Zone Maps
- Creating Zone Maps
- Modifying Zone Maps
- Dropping Zone Maps
- Compiling Zone Maps
- Controlling the Use of Zone Maps
- Maintaining Zone Maps

## 15.2.1 Privileges Required for Zone Maps

- To create, alter, or drop zone maps in your own schema, you must have the `CREATE MATERIALIZED ZONEMAP` privilege
- To create zone maps in other schemas, you must have the `CREATE ANY MATERIALIZED ZONEMAP` privilege.
- To create zone maps in your own schema but on tables from other schemas, you must have the `SELECT ANY TABLE` or `READ ANY TABLE` privilege.
- To create zone maps in other schemas using tables from other schemas, you must have both the `SELECT ANY TABLE` and `CREATE ANY MATERIALIZED ZONEMAP` privileges. You can have the `READ ANY TABLE` privilege instead of the `SELECT ANY TABLE` privilege.
- To alter zone maps in other schemas, you must have the `ALTER ANY MATERIALIZED ZONEMAP` privilege.
- To drop zone maps in other schemas, you must have the `DROP ANY MATERIALIZED ZONEMAP` privilege.


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 15.2.2 Creating Zone Maps

While zone maps can be created along with attribute clustering on a table, zone maps are independent of attribute clustering. Zone maps can be independently created, irrespective of attribute clustering.

Storage structures used by zone maps are created in the default tablespace of the tables on which they are defined.

> **See Also**
>
> - *Oracle Database SQL Language Reference* for zone map creation syntax
> - *Oracle Database SQL Language Reference* for information about zone map restrictions

This section contains the following topics:

- Creating Zone Maps with Attribute Clustering
- Creating Zone Maps Independent of Attribute Clustering

### 15.2.2.1 Creating Zone Maps with Attribute Clustering

You can create a zone map by using `WITH MATERIALIZED ZONEMAP` subclause. You can use this subclause when you define attribute clustering for a table or later when you modify the clustering definition.

Use the steps described in any of the following topics to create a zone map with attribute clustering:

- Creating a Basic Zone Map with Linear Attribute Clustering
- Creating a Join Zone Map with Interleaved Attribute Clustering
- Creating a Zone Map After Attribute Clustering

> **See Also**
>
> Attribute Clustering for information about attribute clustering

#### 15.2.2.1.1 Creating a Basic Zone Map with Linear Attribute Clustering

Assume that queries of sales often specify either a customer ID or a combination of a customer ID and product ID. You can create an attribute-clustered table so that queries benefit from pruning with zone maps. You create a table as follows:

```
CREATE TABLE sales (
   prod_id NUMBER NOT NULL,
   cust_id NUMBER NOT NULL,
   time_id        DATE NOT NULL,
   channel_id     NUMBER NOT NULL,
   promo_id       NUMBER NOT NULL,
   quantity_sold  NUMBER(10,2),
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
   amount_sold NUMBER(10,2)
)
CLUSTERING
BY LINEAR ORDER (cust_id, prod_id)
YES ON LOAD YES ON DATA MOVEMENT
WITH MATERIALIZED ZONEMAP;
```

Zone map ZMAP$_SALES on columns (cust_id, prod_id) is created. Here, ZMAP$_SALES is the name automatically generated by Oracle AI Database for the zone map. However, you can specify a name for the zone map by enclosing it in parentheses following the WITH MATERIALIZED ZONEMAP as described in "Creating a Join Zone Map with Interleaved Attribute Clustering".

Queries that qualify both columns cust_id and prod_id or the prefix cust_id experience natural pruning. The following examples show how the database can prune during table scans.

An application issues the following query:

```
SELECT * FROM sales WHERE cust_id = 100;
```

Because the table is a BY LINEAR ORDER clustered, the database must only read the zones that include the cust_id value of 100.

An application issues the following query:

```
SELECT * FROM sales WHERE cust_id = 100 AND prod_id = 2300;
```

Because the table is a BY LINEAR ORDER clustered, the database must only read the zones that include the cust_id value of 100 and prod_id of 2300.

### 15.2.2.1.2 Creating a Join Zone Map with Interleaved Attribute Clustering

Consider a data warehouse that contains a sales fact table and its two dimension tables: customers and products. Most queries have predicates on the customers table hierarchy (country_id, cust_state_province, cust_city) and the products hierarchy (prod_category, prod_subcategory). You can use interleaved ordering for the sales table as shown in the following partial statement:

```
CREATE TABLE sales (
   prod_id NUMBER NOT NULL,
   cust_id NUMBER NOT NULL,
   amount_sold NUMBER(10,2))
CLUSTERING
sales JOIN products ON (sales.prod_id = products.prod_id)
JOIN customers ON (sales.cust_id = customers.cust_id)
BY INTERLEAVED ORDER
(
(products.prod_category, products.prod_subcategory),
(customers.country_id, customers.cust_state_province, customers.cust_city)
)
YES ON LOAD YES ON DATA MOVEMENT
WITH MATERIALIZED ZONEMAP (sales_zmap);
```

A zone map called sales_zmap is created for the attribute clustered table. Note that, in this clustering clause, the join columns of the dimension table must have primary or unique key constraints. Note that for interleaved order columns from a single dimension should appear in the clustering clause a separate group bracketed by '('..')' for example (prod_category, prod_subcategory). Furthermore, the columns should follow the hierarchy in the dimension (such as the natural hierarchy of prod_category, prod_subcategory), and the order of the


Copyright © 2001, 2025, Oracle and/or its affiliates.

columns in the group should follow that of the hierarchy. This allows Oracle AI Database to effectively cluster the data according to the hierarchies present in the dimension tables.

#### 15.2.2.1.3 Creating a Zone Map After Attribute Clustering

Assume a table called `sales` exists in the database. You can define attribute clustering for the `sales` table using the following command:

```
ALTER TABLE sales ADD CLUSTERING BY INTERLEAVED ORDER (cust_id, prod_id)
      YES ON LOAD YES ON DATA MOVEMENT;
```

Although this command adds attribute clustering to the table definition, it does not cluster the existing data in the `sales` table. When you perform a data movement operation on the `sales` table, its data will be clustered because of the `YES ON DATA MOVEMENT` option.

The following command clusters the data in the `sales` table:

```
ALTER TABLES sales MOVE;
```

After the data in `sales` table is clustered, you can define a zone map on the `sales` table by modifying the clustering using the following command:

```
ALTER TABLE sales MODIFY CLUSTERING WITH MATERIALIZED ZONEMAP (sales_zmap);
```

Subsequently, if necessary, you can drop the zone map by modifying the clustering using the following command:

```
ALTER TABLE sales MODIFY CLUSTERING WITHOUT MATERIALIZED ZONEMAP;
```

### 15.2.2.2 Creating Zone Maps Independent of Attribute Clustering

Use the `CREATE MATERIALIZED ZONEMAP` command to create a zone map on a table. This zone map is independent of attribute clustering, which means it can be created on a clustered or non-clustered table. Also, the set of columns used for the zone map can be same or different from the set of columns used for attribute clustering.

When you create a zone map, you must specify the table columns on which the zone map is based.

Use the steps described in any of the following topics to create a zone map independent of attribute clustering:

- Creating a Basic Zone Map Independent of Attribute Clustering
- Creating a Join Zone Map Independent of Attribute Clustering

#### 15.2.2.2.1 Creating a Basic Zone Map Independent of Attribute Clustering

Assume that queries on the `sales` table frequently specify a customer ID, product ID, or a combination of the two columns. You can create a zone map on the customer ID and product ID columns of the `sales` table so that queries benefit from pruning as shown in Example 15-1.

**Example 15-1 Creating a Basic Zone Map Independent of Attribute Clustering**

You can create a zone map `sales_zmap` on the `sales` table using the following statement:

```
CREATE MATERIALIZED ZONEMAP sales_zmap ON sales (cust_id, prod_id);
```

This statement is equivalent to the following `CREATE ... AS` statement:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
CREATE MATERIALIZED ZONEMAP sales_zmap
REFRESH ON LOAD DATA MOVEMENT
AS
SELECT SYS_OP_ZONE_ID(rowid),MIN(cust_id),MAX(cust_id),MIN(prod_id),MAX(prod_id)
FROM sales
GROUP BY SYS_OP_ZONE_ID(rowid);
```

In this statement, the `SYS_OP_ZONE_ID(rowid)` function is used to work with zone maps. The `SYS_OP_ZONE_ID` function identifies a particular range of contiguous disk blocks (zone) given the rowid of fact table row. This function helps to maintain minimum and maximum ranges at a partition level, performing partition pruning and fast refresh of zone maps. When used with zone maps, it helps to map all rows from a set of contiguous data blocks to a single zone.

#### 15.2.2.2.2 Creating a Join Zone Map Independent of Attribute Clustering

Consider a data warehouse that contains the `sales` fact and multiple dimensions. Most queries have predicates on the `customers` table hierarchy (`cust_state_province, cust_city`). You can use interleaved ordering for the sales table as shown in Example 15-2.

**Example 15-2 Creating a Join Zone Map Independent of Attribute Clustering**

A join zone map involves outer joins from the table on which the zone map is created to one or more other tables. Most commonly used in star schema setups, a join zone map tracks the minimum and maximum of columns from dimension tables rather than columns from the fact table, as is illustrated in the following statement:

```
CREATE MATERIALIZED ZONEMAP sales_zmap
REFRESH ON LOAD DATA MOVEMENT
AS
SELECT SYS_OP_ZONE_ID(s.rowid), MIN(cust_state_province),
       MAX(cust_state_province), MIN(cust_city), MAX(cust_city)
FROM sales s, customers c
WHERE s.cust_id = c.cust_id(+)
GROUP BY SYS_OP_ZONE_ID(s.rowid);
```

### 15.2.3 About Automatic Zone Maps

You can enable automatic creation and maintenance of basic zone maps for both partitioned and non-partitioned tables.

This functionality is not available for join zone maps, IOTs (Oracle Index-organized Tables ), external tables, or temporary tables.

Automatic zone map creation is turned off by default.

> **Note**
>
> See *Oracle Database Licensing Information User Manual* for details on which features are supported for different editions and services.

### 15.2.4 About the `DBMS_AUTO_ZONEMAP` Package

The `DBMS_AUTO_ZONEMAP` package provides controls for turning automatic zone map creation and maintenance on or off, and for generating activity reports.

Execution of members within `DBMS_AUTO_ZONEMAP` requires DBA privileges.


Copyright © 2001, 2025, Oracle and/or its affiliates.

Use the `configure` procedure to turn automatic zone map creation on or off. The procedure also lets you push all automatic zone map creation and maintenance into the background only, foreground only, or allow it in both. When foreground processing is enabled, automatic zone map maintenance is done by the user process accessing the table for direct path and data movement operations. Likewise, when background processing is enabled, automatic zone map creation and maintenance is done by an auto task running in a background process.

The package also includes the `activity_report` function, which displays data about automatic zone map activity within a specified time window and at a configurable level of detail.

### 15.2.4.1 `CONFIGURE` Procedure

The `DBMS_AUTO_ZONEMAP` procedure sets the configuration options for automatic zone maps.

**Syntax**

The procedure accepts two parameters – the parameter name and the parameter value.

For example: `exec dbms_auto_zonemap.configure ('AUTO_ZONEMAP_MODE','ON');`

**Table 15-1 AUTO_ZONEMAP_MODE Parameter Values**

| Parameter | Data Type | Description |
|---|---|---|
| `parameter_name` | `VARCHAR2` | `AUTO_ZONEMAP_MODE` is the only configure parameter name that is currently allowed. If you specify any other name, an invalid argument error message is displayed. |
| `parameter_value` | `VARCHAR2` | This parameter can be assigned one the following values. Each of these values represents an alternative automatic zone map processing mode.<br>• `ON`<br>Turns on automatic zone maps creation and maintenance and enables both foreground and background processing.<br>Fresh zone maps offer the best query performance. The `ON` mode keeps zone maps as fresh as possible. However, foreground tasks that move and bulk-load data may take longer to complete in this mode, because the zone maps are maintained immediately and this adds overhead to the task..<br>• `OFF`<br>Turns off automatic zone map creation and maintenance in both the foreground and background. This is the default.<br>• `FOREGROUND`<br>Turns on automatic zone map creation and maintenance. All processing is done in the foreground only.<br>This mode is not commonly used. It may be appropriate for closely-managed environments where control over the timing of refresh operations is required.<br>• `BACKGROUND`<br>Turns on automatic zone map creation and maintenance. All processing is done in the background only.<br>For bulk load and data movement foreground processes, this mode avoids adding the immediate overhead of zone map maintenance to these processes. However, it takes longer for the relevant zone maps to be refreshed in this mode, because the refresh is done asynchronously in the background. |


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 15.2.4.2 ACTIVITY_REPORT Function

This DBMS_AUTO_ZONEMAP function reports all automatic zone maps activity within a given time window.

The background job that performs automatic zone map processing starts once per hour (and each run may last up to three hours). The report shows activity for all instances of the job running within the specified time window. The report is returned as a CLOB.

### Syntax

```
DBMS_AUTO_ZONEMAP.ACTIVITY_REPORT ( START_TIME, END_TIME, TYPE, SECTION,
LEVEL );
```

The returned CLOB can contain a report formated as TEXT, HTML, or XML. The format is set by the type parameter described in the table below.

**Table 15-2 ACTIVITY_REPORT Parameters**

| Parameter | Data Type | Description |
|---|---|---|
| start_time | TIMESTAMP | Start of the time window from which automatic zone map executions are observed for the report. The default value is NULL.<br>Possible values:<br>• time-value<br>Report all activity proceding from this start time<br>• NULL<br>Report recorded zone map maintenance activity from the earliest data available. Note that this history may be subject to purging. |
| end_time | TIMESTAMP | End of the time window for the report.<br>Possible values:<br>• time-value<br>Report all activity up to this end time.<br>• NULL<br>Report all activity up to the end time of recorded activity.<br><br>**Note**<br>If both start_time and end_time are NULL, then report returned shows the activity from the most recent run of the job only. |
| type | VARCHAR2 | The output type of the report. Possible values are: TEXT, XML, and HTML. The default value is TEXT. The report formatted as any of these types is stored as a CLOB. |


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Table 15-2 (Cont.) ACTIVITY_REPORT Parameters**

| Parameter | Data Type | Description |
|---|---|---|
| section | VARCHAR2 | Sections that you want to include in the report.<br>Possible values are:<br>• SUMMARY<br>A high-level summary including the counts of new zone maps created and zone maps maintained for the given time window.<br>• DETAILS<br>A more detailed expansion of the summary, that includes names and other data about new zone maps created and zone maps maintained for the given time window. It also includes findings details.<br>• ALL<br>Includes the summary, details, as well as the content from time series-based execution and action logs.<br>The default value is DETAILS. |
| level | VARCHAR2 | Sets the level of detail within each section of the report. Possible values are:<br>• BASIC<br>Shows only important messages from the action logs as well as a summary of up to two lines on zone maps created or maintained in the time window. Details such as column cluster ratios, process of compilation, and rebuilding of zone maps are not reported at this level.<br>• TYPICAL<br>In addition to the same information reported at BASIC level, this level reports column clustering ratio computation information, table eligibility criteria checks, and some other details.<br>• ALL<br>Returns all of the details provided by BASIC and TYPICAL and also shows the detailed time series logs on all actions performed during zone map maintenance. This level is most useful for debugging and also for in-depth analysis of automatic zone map activity.<br>The default value is TYPICAL. |

### Usage Examples

- dbms_auto_zonemap.activity_report()
  Report on the last job execution only. Format the report as TEXT. Include the TYPICAL level of detail (the default level).

```
SET LONG 100000
SELECT dbms_auto_zonemap.activity_report()report FROM dual;
```

- dbms_auto_zonemap.activity_report(systimestamp-2)
  Report on all execution history for last two days. Format the report as TEXT. Include the TYPICAL level of detail.

```
SELECT dbms_auto_zonemap.activity_report(systimestamp-2) report FROM dual;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

- `dbms_auto_zonemap.activity_report(systimestamp-2, systimestamp, 'XML', 'ALL', 'ALL')`
  Return a report for last 48 hours in `XML` format. Include `ALL` sections of the reported and include `ALL` details.

  ```
  SELECT dbms_auto_zonemap.activity_report(systimestamp-2, systimestamp,
  'XML', 'ALL', 'ALL') report FROM dual;
  ```

### 15.2.4.3 Viewing Information About Automatic Zone Maps

Use the DBA_AUTO_ZONEMAP_CONFIG data dictionary view to display information about automatic zone maps in the database. For example:

```
SELECT parameter_name, parameter_value FROM dba_auto_zonemap_config WHERE
parameter_name = 'AUTO_ZMAP_MODE';


PARAMETER_NAME                    PARAMETER_VALUE
----------------------------------- --------------
AUTO_ZMAP_MODE                                OFF
```

> **See Also**
>
> DBA_AUTO_ZONEMAP_CONFIG in the *Oracle Database Reference*.

## 15.2.5 Modifying Zone Maps

You can alter a zone map with an `ALTER MATERIALIZED ZONEMAP` statement.

**Example 15-3 Making a Zone Map Unusable**

The following statement makes a zone map unusable, which means that queries no longer use this zone map, and Oracle AI Database no longer maintains the zone map.

```
ALTER MATERIALIZED ZONEMAP sales_zmap UNUSABLE;
```

**Example 15-4 Performing Complete Refresh for a Zone Map**

The following statement performs a complete refresh of the zone map:

```
ALTER MATERIALIZED ZONEMAP sales_zmap REBUILD COMPLETE;
```

As part of the rebuild, the zone map is also made usable, if it was earlier marked as unusable.

**Example 15-5 Refreshing Zone Maps**

The following statement performs a fast refresh, if possible. Else, a complete refresh is performed.

```
ALTER MATERIALIZED ZONEMAP sales_zmap REBUILD;
```

**Example 15-6 Disabling Pruning for Zone Maps**

The following statement disables pruning, which you might want to do for performance measurement:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
ALTER MATERIALIZED ZONEMAP sales_zmap DISABLE PRUNING;
```

**Example 15-7 Enabling Pruning for Zone Maps**

The following statement enables pruning, which may have been disabled earlier, for the zone map:

```
ALTER MATERIALIZED ZONEMAP sales_zmap ENABLE PRUNING;
```

**Example 15-8 Disabling Refresh for Zone Maps**

The following statement turns off refresh on load and data movement, which offers you control over how and when zone maps are refreshed:

```
ALTER MATERIALIZED ZONEMAP sales_zmap REFRESH ON DEMAND;
```

**Example 15-9 Enabling Refresh on Commit for Zone Maps**

The following statement turns on the refresh of the zone map on each transaction commit:

```
ALTER MATERIALIZED ZONEMAP sales_zmap REFRESH ON COMMIT;
```

**See Also**

- *Oracle Database SQL Language Reference* for the syntax to alter a zone map

## 15.2.6 Dropping Zone Maps

You can drop zone maps by issuing a `DROP MATERIALIZED ZONEMAP` statement, such as the following:

```
DROP MATERIALIZED ZONEMAP sales_zmap;
```

**See Also**

*Oracle Database SQL Language Reference* for the syntax to drop a zone map

## 15.2.7 Compiling Zone Maps

Any DDL operation on the base table on which a zone map is based will affect the compile state of the zone map. This means that the query that defines the zone map must be compiled to check if the zone map remains valid or not. This behavior is similar to materialized views, which are also affected by DDL performed on the base table. Oracle AI Database will compile the zone map the first time it tries to use it following a DDL operation. You can, however, explicitly compile a zone map using an alter DDL statement such as the following:

```
ALTER MATERIALIZED ZONEMAP sales_zmap COMPILE;
```

The result of compiling a zone map will either be valid or invalid depending on the specific action performed by the DDL. For example, if DDL was done to add a column to the fact table, then the zone map will be valid after compilation. But if the DDL was done to drop a column that is referenced in the definition query, then the zone map will be invalid after compilation.

Some of the points to keep in mind are:


Copyright © 2001, 2025, Oracle and/or its affiliates.

- If a column that appears in the clustering clause is dropped, then clustering is dropped. In addition, if there was a zone map created as part of clustering, then the zone map will be dropped as well.
- If a dimension table from a star schema is dropped, and it is involved in clustering on a fact table, then the clustering on the fact table is dropped. In addition, if there was a zone map created as part of the clustering, then the zone map will be dropped.
- If a user drops a required primary key or unique key on the dimension table involved in a clustering clause, then clustering is invalidated (data will not be clustered on subsequent loads or data movement operations performed by certain types of PMOPs).

## 15.2.8 Controlling the Use of Zone Maps

You can control the use of zone maps for the entire SQL workload or for specific SQL statements.

This section contains the following topics:

- Controlling Zone Map Usage for Entire SQL Workloads
- Controlling Zone Map Usage for Specific SQL Statements

### 15.2.8.1 Controlling Zone Map Usage for Entire SQL Workloads

You can control the use of zone maps at the object level. Object-level changes apply to all statements in the SQL workload. When you create a zone map, it is available for pruning unless you override the default by specifying `DISABLE PRUNING`. For example, the following statement creates a zone map with pruning disabled:

```
CREATE MATERIALIZED ZONEMAP sales_zmap
    DISABLE PRUNING ON sales(cust_id, prod_id);
```

This zone map is created and maintained by Oracle AI Database, but is not used for any SQL in the workload. You can make it available for pruning by using the following `ALTER MATERIALIZED ZONEMAP` statement:

```
ALTER MATERIALIZED ZONEMAP sales_zmap ENABLE PRUNING;
```

Similarly, you can use the following statement to make a zone map unavailable for pruning:

```
ALTER MATERIALIZED ZONEMAP sales_zmap DISABLE PRUNING;
```

### 15.2.8.2 Controlling Zone Map Usage for Specific SQL Statements

You can use hints to control the use of zone maps at the individual SQL statement level. Note that hints cannot be used to control zone map usage if pruning is disabled for the zone map. You can achieve a finer control through hints by leaving pruning enabled and specifying negative hints in individual SQL statements.

Use the `NO_ZONEMAP` hint to disable the usage of a zone map for pruning. The following examples disable the usage of zone maps while pruning data.

**Example 15-10 Scan Pruning: Disabling Zone Maps with the NO_ZONEMAP Hint**

```
SELECT /*+ NO_ZONEMAP (S SCAN) */* FROM sales S
WHERE s.time_id BETWEEN '1-15-2008' AND '1-31-2008';
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Example 15-11 Join Pruning: Disabling Zone Maps with the NO_ZONEMAP Hint**

```
SELECT /*+ NO_ZONEMAP (S JOIN) */* FROM sales s
WHERE s.time_id BETWEEN '1-15-2008' AND '1-31-2008';
```

**Example 15-12 Partition Pruning: Disabling Zone Maps with the NO_ZONEMAP Hint**

```
SELECT /*+ NO_ZONEMAP (S PARTITION) */* FROM sales S
WHERE s.time_id BETWEEN '1-15-2008' AND '1-31-2008';
```

## 15.2.9 Maintaining Zone Maps

You can specify how zone maps must be maintained either at the time of creating the zone map or, later, by altering the zone map definition. Refer to "Zone Map Maintenance Considerations".

> **See Also**
>
> "About Maintaining Zone Maps"

Use the `REFRESH` clause in the `CREATE MATERIALIZED ZONEMAP` or `ALTER MATERIALIZED ZONEMAP` statement to specify how zone maps must be maintained. If you omit the `REFRESH` clause in the `CREATE MATERIALIZED ZONEMAP` statement, the default used is `REFRESH ON LOAD DATA MOVEMENT`, which enables the maintenance of the zone map by Oracle AI Database upon direct path load and certain data movement operations.

The following statement creates a zone map whose maintenance is managed manually by the user:

```
CREATE MATERIALIZED ZONEMAP sales_zmap
REFRESH ON DEMAND
ON sales (cust_id, prod_id);
```

The following statement creates a zone map whose maintenance is managed by Oracle AI Database at the end of each transaction commit:

```
CREATE MATERIALIZED ZONEMAP sales_zmap
REFRESH ON COMMIT
ON sales (cust_id, prod_id);
```

Because it is refreshed on commit, the above zone map never becomes stale.

Use the `ALTER MATERIALIZED ZONEMAP` statement to change the maintenance of existing zone maps.

**Example 15-13 Enabling Zone Map Maintenance on Data Movement**

The following statement enables zone map maintenance by Oracle AI Database on data movement operations such as `MOVE`, `SPLIT`, `MERGE`, and `DROP`:

```
ALTER MATERIALIZED ZONEMAP sales_zmap REFRESH ON DATA MOVEMENT;
```

**Example 15-14 Enabling Zone Map Maintenance on Direct Path Load**

The following statement enables zone map maintenance by Oracle AI Database on direct path load operations, such as `INSERT /*+ APPEND */` statements:

```
ALTER MATERIALIZED ZONEMAP sales_zmap REFRESH ON LOAD;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Example 15-15 Enabling Zone Map Maintenance on both Data Movement and Load**

The following statement enables zone map maintenance by Oracle AI Database on data movement and load operations:

```
ALTER MATERIALIZED ZONEMAP sales_zmap REFRESH ON LOAD DATA MOVEMENT;
```

Note that `REFRESH ON LOAD DATA MOVEMENT` is the default option.

## 15.2.9.1 Zone Map Maintenance Considerations

The following are some of the issues to keep in mind when maintaining zone maps or tracking their staleness:

- DML/Parallel DML operations to the fact table

  When a zone map is created, an internal trigger is created by Oracle AI Database to track the row changes made by conventional DML operations. For example, if a new row is inserted into the `sales` table, this trigger will compute `zone_id` from `rowid` and mark the corresponding aggregate row in the zone map as stale. So the staleness of a zone map is tracked zone by zone, which means even after DML has been done to the fact table the zone map can still be used for pruning using the `MIN`/`MAX` aggregates in the fresh zones.

  Upon fact table update, if the columns being updated are not referenced by the zone map, then zone map staleness is not affected. Otherwise, zones corresponding to updated rows are marked as stale by the internal trigger.

- Direct loads (that is, `INSERT /*+ APPEND */`) operations to fact table

  Even though direct loads insert data above the high water mark, newly added rows can belong to zones already computed for the zone map. Therefore, Oracle AI Database will identify existing zones whose `MIN`/`MAX` aggregates are potentially affected by newly added data and mark such zones as stale. Again, Oracle AI Database can continue to use the zone map for pruning in spite of direct loads to the zone map by utilizing `MIN`/`MAX` aggregates of zones that still remain fresh. If the zone maps has the `REFRESH ON LOAD` option, then Oracle AI Database will perform zone map refresh at the end of the load.

- Data movement (for example, partition maintenance operations) on the fact table

  Data movement operations include partition maintenance operations and online partition/table redefinition. However, data movement (for example, move partition) will make the existing zones belonging to the old partition obsolete in the zone map while zones belonging to the new partition are not computed until the zone map is refreshed. Oracle AI Database will continue to use the zone map for pruning following data movement operations regardless of whether the zone map was refreshed or not. If the zone map has the `REFRESH ON DATA MOVEMENT` option, Oracle AI Database will perform refresh at the end of the data movement operation.

- Data movements on the dimension table

  This operation does not affect the zone map.

- Any DML to the dimension table

  This operation makes the entire zone map stale, so it requires a full refresh. However, there is one exception. If it is an update operation and the set of updated columns are not referenced by the zone map, then it remains unaffected.

- Direct loads to dimension table

  This operation makes the entire zone map stale. If the `REFRESH ON LOAD` option is specified for the zone map, then Oracle AI Database will perform zone map refresh immediately following the load operation.


Copyright © 2001, 2025, Oracle and/or its affiliates.

- DDL to the fact or dimension table

  Upon DDL operation the zone map is marked with unknown staleness (that is, stale set to `'unknown'`) and requiring compilation (that is, `compile_state` set to `'needs_compile'`). Under this state, Oracle AI Database will not use the zone map for pruning. However, upon the first use of a zone map following the DDL operation Oracle AI Database will compile the zone map and based on its outcome appropriately set the invalid and stale states. For example, if the DDL operation dropped a column whose `MIN`/`MAX` aggregates are stored in the zone map, then zone map compilation will fail so zone map `compile_state` is set to `'compilation error'`, stale remains as `'unknown'`, and invalid is set to `'yes'`.

# 15.3 Refresh and Staleness of Zone Maps

Oracle AI Database marks either the zone maps as stale or individual zones within zone maps as stale when the data in their base tables changes. Stale zone maps are not used for pruning, but zone maps with stale zones are still used for pruning. You must refresh the zone maps to update the zones and make them usable for pruning.

This section contains the following topics:

- About Staleness of Zone Maps
- About Refreshing Zone Maps
- Refreshing Zone Maps

## 15.3.1 About Staleness of Zone Maps

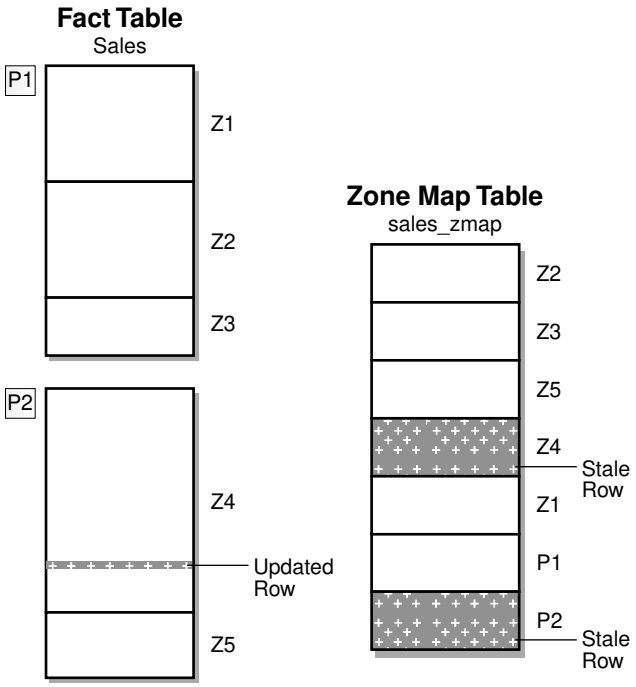
When the data in the tables on which a zone map is based changes, the zones corresponding to the changed rows are marked as stale. You need to refresh the zone map to make the zones current.

When a row in a partition of the fact table is updated, the row corresponding to the zone in the partitioned table is marked as stale because of the update. This automatically invalidates the aggregated partition-level information, and pruning can only happen on a zone level. The row in the zone map corresponding to this particular partition is also marked as stale because of the update.

In Figure 15-1, this is illustrated with an update in `Z4` of `P2`, and the corresponding `Z4` is marked as stale. Note that the zone map is still usable, however. Table data corresponding to `Z4` will always be read (no pruning is performed on `Z4`) as long as the zone map is partially stale.

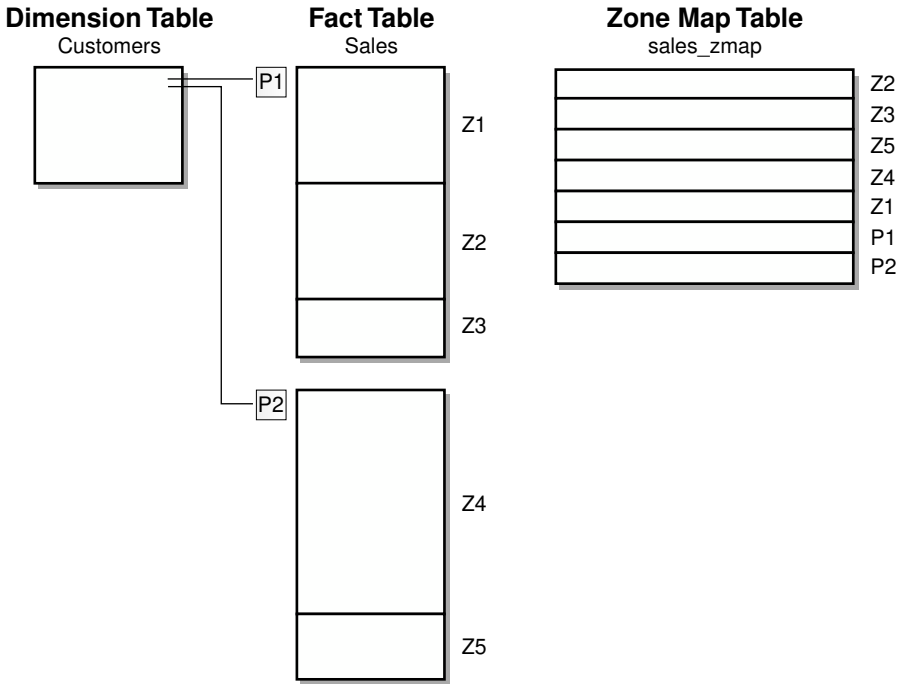

Copyright © 2001, 2025, Oracle and/or its affiliates.

**Figure 15-1 Partially Stale Zone Map**




If a dimension table is added to the fact table, then the status resembles that in Figure 15-2.

**Figure 15-2 Zone Map with Dimension Table**

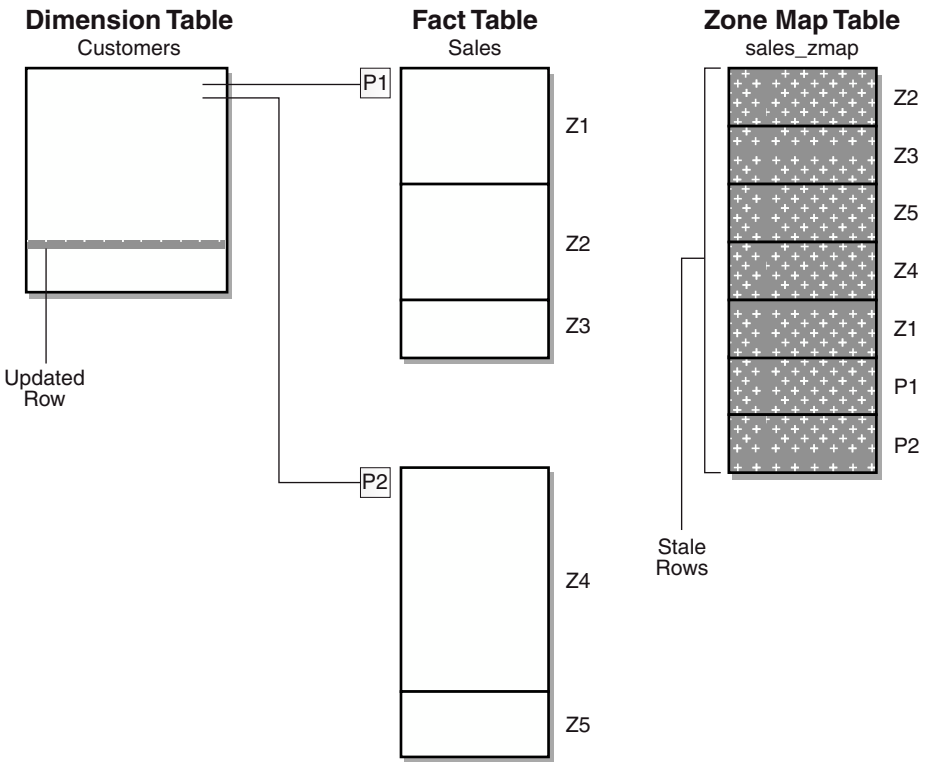



If any DML is made to the dimension table, the zone map becomes fully stale, as is illustrated in Figure 15-3. Because the zone map becomes fully state, it is not available for pruning until it


Copyright © 2001, 2025, Oracle and/or its affiliates.

is completely refreshed. Use the `REBUILD` option of `ALTER MATERIALIZED ZONEMAP` statement to refresh the zone map.

**Figure 15-3 Zone Map with Dimension Table and Staleness**




## 15.3.2 About Refreshing Zone Maps

Oracle AI Database needs to maintain zone maps by refreshing them after changes to their underlying tables. The refresh method used for zone maps can be a complete refresh or an incremental refresh. A complete refresh, specified using the `REFRESH COMPLETE` clause, involves rebuilding all the zones in the zone map. A complete refresh is slow when large amounts of data need to be processed. An incremental refresh, specified using the `REFRESH FAST` clause, processes only the changes that have occurred since the last refresh. This method enables you to refresh the zone map without rebuilding them from scratch. Although zone maps are internally implemented using materialized view, materialized view logs on base tables are not required to perform a fast refresh of a zone map

The refresh mode specifies the operations that trigger zone map refresh. Use one of the following refresh modes:

- `ON COMMIT`

  Zone maps are refreshed when changes to the base tables are committed.

- `ON DEMAND`

  Zone maps must be refreshed manually after DML or partition maintenance operations.

- `ON DATA MOVEMENT`

  Zone maps are refreshed when data movement operations are performed on the base tables.


Copyright © 2001, 2025, Oracle and/or its affiliates.

- `ON LOAD`

  Zone maps are refreshed when direct-path insert operations are performed on the base tables.

- `ON LOAD DATA MOVEMENT`

  Zone maps are refreshed when direct-path insert or certain data movement operations are performed on the base tables. This is the default.

By default, zone maps are refreshed on load and on data movement. To override this default, specify one of the following refresh modes when creating or modifying the zone map: `ON COMMIT`, `ON LOAD`, `ON DATA MOVEMENT`, or `ON LOAD`.

## 15.3.3 Refreshing Zone Maps

When you create a zone map without specifying the `REFRESH` option, Oracle AI Database by default performs zone map maintenance after direct load and certain data movement operations. The exception is the DML operations such as delete, insert, and update. For these operations, Oracle AI Database will appropriately mark the zone map or some zones in the zone map as stale. To manually control the refresh maintenance of zone maps, you must specify the `REFRESH ON DEMAND` option.

The following command creates a zone map whose refresh maintenance is disabled which means that you must manually refresh the zone map after changes are made to the underlying tables.

```
CREATE MATERIALIZED ZONEMAP sales_zmap
      ON sales (time_id, cust_id)
      REFRESH ON DEMAND;
```

Oracle AI Database provides the following two methods of refreshing zone maps:

- Refreshing Zone Maps Using the ALTER MATERIALIZED ZONEMAP Command
- Refreshing Zone Maps Using the DBMS_MVIEW Package

### 15.3.3.1 Refreshing Zone Maps Using the ALTER MATERIALIZED ZONEMAP Command

Use the `REBUILD` option of `ALTER MATERIALIZED ZONEMAP` command to refresh zone maps.

The following command performs a complete refresh of the zone map:

```
ALTER MATERIALIZED ZONEMAP sales_zmap REBUILD COMPLETE;
```

The following command performs a complete refresh if the zone map is fully stale or marked as unusable. Otherwise, an incremental (fast) refresh is performed.

```
ALTER MATERIALIZED ZONEMAP sales_zmap REBUILD;
```

> **See Also**
>
> *Oracle Database SQL Language Reference* for the syntax to refresh a zone map


Copyright © 2001, 2025, Oracle and/or its affiliates.

### 15.3.3.2 Refreshing Zone Maps Using the DBMS_MVIEW Package

You can use the `REFRESH` procedure of `DBMS_MVIEW` package to refresh zone maps.

When `DBMS_MVIEW.REFRESH` procedure is used, Oracle AI Database will refresh the zone map according to the value specified for its `refresh_method` parameter as follows:

- C: Performs a complete refresh.
- F - Performs a fast refresh. If a fast refresh is not possible, then an error is issued.
- ? - Performs a fast refresh if possible. else a complete refresh is performed.

  This is the default used if no value is specified.

An example of using the `REFRESH` procedure is the following:

```
EXECUTE DBMS_MVIEW.REFRESH('sales_zmap','C');
```

## 15.4 Performing Pruning Using Zone Maps

The primary benefit of zone maps is I/O reduction for table scans. Pruning leverages information about the natural locality of records to avoid unnecessary I/O. When a SQL statement contains predicates on columns tracked in the zone map, the database compares the predicate values to the minimum and maximum for each zone to determine which zones of blocks to read or skip during the table scan.

Candidates for zone map pruning include the following predicates:

- Relational predicates =, <=, <, >, >=
- (of the form `column_name relational_predicate constant`, for example, `WHERE country_name='US'` or `WHERE country_name=:name`)
- `IN` lists (for example, `WHERE product_name IN ('a','b')`)
- `LIKE` predicates suffixed with `%` (for example, `company_name LIKE 'ORA%'`)

This section contains the following topics:

- How Oracle Database Performs Pruning Using Zone Maps
- Examples: Performing Pruning with Zone Maps and Attribute Clustering

## 15.4.1 How Oracle AI Database Performs Pruning Using Zone Maps

This section uses the following examples to illustrate how pruning is performed with zone maps and attribute clustering:

- Pruning Tables Using Zone Maps
- Pruning Partitioned Tables Using Zone Maps and Attribute Clustering

### 15.4.1.1 Pruning Tables Using Zone Maps

This example illustrates creating a zone map that can prune data in a query whose predicate contains a constant. The `lineitem` table, illustrated in Table 15-3, is created using the following statement:

```
CREATE TABLE lineitem
  ( orderkey      NUMBER
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
  , shipdate      DATE
  , receiptdate   DATE
  , destination   VARCHAR2(50)
  , quantity      NUMBER);
```

Assume that this table contains four data blocks with two rows per block. Table 15-3 shows the eight rows of the table.

**Table 15-3 Data Blocks for lineitem Table**

| Block | orderkey | shipdate | receiptdate | destination | quantity |
|---|---|---|---|---|---|
| 1 | 1 | 1-1-2011 | 1-10-2011 | San_Fran | 100 |
| 1 | 2 | 1-2-2011 | 1-10-2011 | San_Fran | 200 |
| 2 | 3 | 1-3-2011 | 1-5-2011 | San_Fran | 100 |
| 2 | 4 | 1-5-2011 | 1-10-2011 | San_Diego | 100 |
| 3 | 5 | 1-10-2011 | 1-15-2011 | San_Fran | 100 |
| 3 | 6 | 1-12-2011 | 1-16-2011 | San_Fran | 200 |
| 4 | 7 | 1-13-2011 | 1-20-2011 | San_Fran | 100 |
| 4 | 8 | 1-15-2011 | 1-30-2011 | San_Jose | 100 |

Next, you use the `CREATE MATERIALED ZONEMAP` statement to create a zone map on the `lineitem` table.

```
CREATE MATERIALIZED ZONEMAP lineitem_zmap
     ON lineitem (orderkey, shipdate, receiptdate);
```

Each zone contains two blocks and stores the minimum and maximum of the `orderkey`, `shipdate`, and `receiptdate` columns. Table 15-4 represents the zone map.

**Table 15-4 Zone Map for lineitem Table**

| Block Range | min orderkey | max orderkey | min shipdate | max shipdate | min receiptdate | max receiptdate |
|---|---|---|---|---|---|---|
| 1-2 | 1 | 4 | 1-1-2011 | 1-5-2011 | 1-5-2011 | 1-10-2011 |
| 3-4 | 5 | 8 | 1-10-2011 | 1-15-2011 | 1-15-2011 | 1-30-2011 |

When you execute the following query, the database can read the zone map and then scan only blocks 1 and 2 because the date 1-3-2011 falls between the minimum and maximum dates:

```
SELECT * FROM lineitem WHERE shipdate = '1-3-2011';
```

## 15.4.1.2 Pruning Partitioned Tables Using Zone Maps and Attribute Clustering

This following statement creates a zone map, with attribute clustering, on a partitioned table:

```
CREATE TABLE sales
(
   prod_id        NUMBER NOT NULL,
   cust_id        NUMBER NOT NULL,
   time_id        DATE NOT NULL,
   channel_id     NUMBER NOT NULL,
   promo_id       NUMBER NOT NULL,
   quantity_sold  NUMBER(10,2) NOT NULL,
```

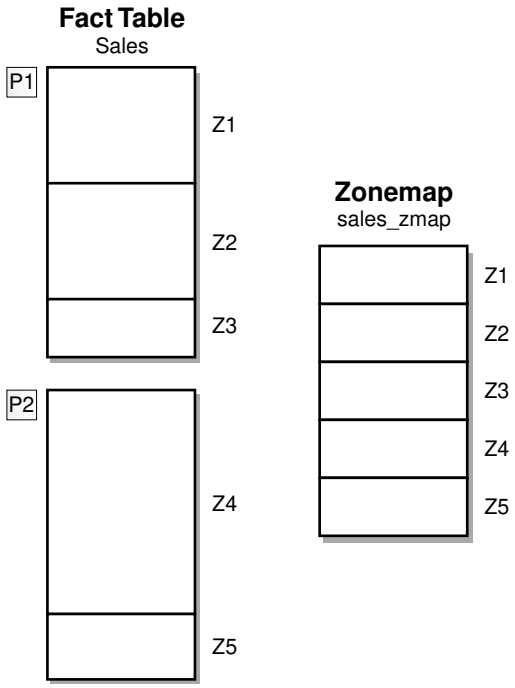

Copyright © 2001, 2025, Oracle and/or its affiliates.

```
   amount_sold    NUMBER(10,2)
)
CLUSTERING sales JOIN products ON (sales.prod_id = products.prod_id)
   BY LINEAR ORDER (products.prod_id)
   WITH MATERIALIZED ZONEMAP (sales_zmap)
PARTITION BY HASH (amount_sold)
     ( PARTITION p1, PARTITION p2);
```

Figure 15-4 illustrates creating zone maps for the partitioned table `sales`. For each of the five zones, the zone map will store the minimum and maximum of the columns tracked in the zone map. If a predicate is outside the minimum and maximum for a stored column of a given zone, then this zone does not have to be read. As an example, if zone `Z4` tracks a minimum of 10 and a maximum of 100 for a column `prod_id`, then a predicate `prod_id = 200` will never have any matching records in this zone, so zone `Z4` will not be read.

For partitioned tables, pruning can happen both on a partition as well as zone level. If the aggregated partition-level information in the zone maps rules out any matching data for a given set of predicates, then the whole partition will be pruned; otherwise, pruning for the partition will happen on a per zone level.

**Figure 15-4 Zone Map for a Partitioned Fact Table**




## 15.4.2 Examples: Performing Pruning with Zone Maps and Attribute Clustering

This section provides examples on performing pruning using zone maps and attribute clustering. The examples are based on the `my_sales` table that is created as shown in Example 15-16.

**Example 15-16 Creating the my_sales Table**

The `my_sales` table is a join attribute clustered table that contains a zone map. It is based on the `sales` tables in the `SH` schema and is created using the following statement:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
CREATE TABLE my_sales
PARTITION BY LIST (channel_id)
 (PARTITION mysales_chan_c VALUES ('C'),
  PARTITION mysales_chan_i VALUES ('I'),
  PARTITION mysales_chan_p VALUES ('P'),
  PARTITION mysales_chan_s VALUES ('S'),
  PARTITION mysales_chan_t VALUES ('T'))
CLUSTERING
  my_sales JOIN customers ON (my_sales.cust_id = customers.cust_id)
  BY INTERLEAVED ORDER ((my_sales.time_id),
                        (customers.country_id,
                         customers.cust_state_province,
                         customers.cust_city))
  WITH MATERIALIZED ZONEMAP (mysales_zmap)
AS SELECT * FROM sales;
```

This section contains the following topics:

- Example: Partitions and Table Scan Pruning
- Example: Zone Map Join Pruning

## 15.4.2.1 Example: Partitions and Table Scan Pruning

This example illustrates how zone maps can prune zones and partitions (or sub-partitions in a composite-partitioned table).

1. Create the `my_sales` table. Example 15-16 contains the syntax used to create this table.
2. Use the following statement to query the `my_sales` table joined with the `customers` dimension:

```
SELECT c.cust_city, SUM(quantity_sold)
   FROM   my_sales s, customers c
   WHERE  s.cust_id = c.cust_id
      AND    c.country_id = 'US'
      AND    c.cust_state_province = 'CA'
      AND    s.promo_id < 50
   GROUP BY c.cust_city;
```

3. Display the plan using the following statement:

```
SELECT *
FROM TABLE(dbms_xplan.display_cursor(FORMAT => 'BASIC PREDICATE PARTITION'));

-------------------------------------------------------------------------------
| Id  | Operation                           | Name      | Pstart| Pstop |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |           |       |       |
|   1 |  HASH GROUP BY                      |           |       |       |
|*  2 |   HASH JOIN                         |           |       |       |
|   3 |    JOIN FILTER CREATE               | :BF0000   |       |       |
|*  4 |     TABLE ACCESS FULL               | CUSTOMERS |       |       |
|   5 |    JOIN FILTER USE                  | :BF0000   |       |       |
|   6 |     PARTITION LIST ITERATOR         |           |KEY(ZM)|KEY(ZM)|
|*  7 |      TABLE ACCESS FULL WITH ZONEMAP | MY_SALES  |KEY(ZM)|KEY(ZM)|
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------
   2 - access("S"."CUST_ID"="C"."CUST_ID")
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
   4 - filter(("C"."CUST_STATE_PROVINCE"='CA' AND
      "C"."COUNTRY_ID"='US'))
   7 - filter((SYS_ZMAP_FILTER('/* ZM_PRUNING */ SELECT "ZONE_ID$",
      CASE WHEN BITAND(zm."ZONE_STATE$",1)=1 THEN 1 ELSE CASE WHEN
      (zm."MIN_2_COUNTRY_ID" > :1 OR zm."MAX_2_COUNTRY_ID" < :2 OR
      zm."MIN_3_CUST_STATE_PROVINCE" > :3 OR zm."MAX_3_CUST_STATE_PROVINCE" <
      :4) THEN 3 ELSE 2 END END FROM "SH"."MYSALES_ZMAP" zm WHERE
      zm."ZONE_LEVEL$"=0 ORDER BY zm."ZONE_ID$"',SYS_OP_ZONE_ID(ROWID),'US','U
      S','CA','CA')<3 AND "S"."PROMO_ID"<50 AND
      SYS_OP_BLOOM_FILTER(:BF0000,"S"."CUST_ID")))
```

Line 7 illustrates that a zone map is being used. Note the zone map partition list iterator “KEY(ZM)".

## 15.4.2.2 Example: Zone Map Join Pruning

This example illustrates join pruning using zone maps and attribute clustering. If the primary key of a dimension comprises of dimension hierarchy values, it is sufficient to cluster the fact table by the corresponding foreign key. In this example, `times.time_id` comprises of (`calendar_year`, `calendar_month_number`, `day_number_in_month`). Thus, `time_id` translates to the calendar time hierarchy as well as the fiscal time hierarchy. You can prune the join between `times` and `my_sales` when there are predicates for either the fiscal or calendar hierarchies.

1. Create the `my_sales` table. Example 15-16 contains the syntax used to create this table.
2. Query the `my_sales` table joined with `times` using the following statement:

```
SELECT SUM(quantity_sold)
FROM   my_sales s, times tWHERE  s.time_id = t.time_id AND t.calendar_year = '1999';
```

3. Display the plan using the following statement:

```
SELECT *
FROM TABLE(dbms_xplan.display_cursor(FORMAT => 'BASIC PREDICATE PARTITION'));

-------------------------------------------------------------------------------
| Id  | Operation                               | Name    | Pstart| Pstop |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                        |         |       |       |
|   1 |  SORT AGGREGATE                         |         |       |       |
|*  2 |   HASH JOIN                             |         |       |       |
|   3 |    JOIN FILTER CREATE                   | :BF0000 |       |       |
|*  4 |     TABLE ACCESS FULL                   | TIMES   |       |       |
|   5 |    JOIN FILTER USE                      | :BF0000 |       |       |
|   6 |     PARTITION LIST ALL                  |         |     1 |     5 |
|*  7 |      TABLE ACCESS FULL WITH ZONEMAP     | MY_SALES|     1 |     5 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

PLAN_TABLE_OUTPUT
-----------------------------------------------------
   2 - access("S"."TIME_ID"="T"."TIME_ID")
   4 - filter("T"."CALENDAR_YEAR"=1999)
   7 - filter((SYS_ZMAP_FILTER('/* ZM_PRUNING */ SELECT "ZONE_ID$",
      CASE WHEN BITAND(zm."ZONE_STATE$",1)=1 THEN 1 ELSE CASE WHEN
      ((ORA_RAWCOMPARE(zm."MIN_1_TIME_ID",:1,8)>0 OR
      ORA_RAWCOMPARE(zm."MAX_1_TIME_ID",:2,8)<0)) THEN 3 ELSE 2 END END FROM
      "SH"."MYSALES_ZMAP" zm WHERE zm."ZONE_LEVEL$"=0 ORDER BY
      zm."ZONE_ID$"',SYS_OP_ZONE_ID(ROWID),SYSVARCOL,SYSVARCOL)<3 AND
      SYS_OP_BLOOM_FILTER(:BF0000,"S"."TIME_ID")))
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

Line 7 illustrates that a zone map is being used, joining on matching `time_id` zones.

# 15.5 Viewing Zone Map Information

Information about zone maps and their measures is stored in data dictionary views.

This section contains the following topics:

- Viewing Details of Zone Maps in the Database
- Viewing the Measures of a Zone Map

## 15.5.1 Viewing Details of Zone Maps in the Database

Use one of the following data dictionary views to display information about the zone maps in the database:

- `DBA_ZONEMAPS` to display all zone maps in the database
- `ALL_ZONEMAPS` to display zone maps that are accessible to the user
- `USER_ZONEMAPS` to display zone maps that are owned by the user

The following query displays the name, base table, type, refresh mode, and staleness of the zone maps owned by the current user and indicates if zone maps were created with attribute clustering:

```
SELECT zonemap_name,fact_table,hierarchical,with_clustering,refresh_mode,stale
       FROM USER_ZONEMAPS;

ZONEMAP_NAME    FACT_TABLE  HIERARCHICAL  WITH_CLUSTERING REFRESH_MODE      STALE
------------    ----------  ------------  --------------- ------------      -----
ZMAP$_MY_SALES  MY_SALES      NO          YES             LOAD DATAMOVEMENT   NO
```

The following query displays the status of all zone maps that are accessible to the user. Zone maps with `PRUNING` disabled are not used for I/O pruning. Zone maps marked invalid need to be recompiled because the structure of the underlying base tables has changed.

```
SQL> SELECT zonemap_name,pruning,refresh_method,invalid,complie_state
    FROM all_zonemaps;

ZONEMAP_NAME     PRUNING    REFRESH_METHOD INVALID  UNUSABLE   COMPILE_STATE
------------    ---------   -------------- -------  ------     -------------
SALES_ZMAP       ENABLED    FORCE           NO      NO         VALID
ZMAP$_MY_SALES   DISABLED   FORCE           NO      NO         VALID
```

## 15.5.2 Viewing the Measures of a Zone Map

Use one of the following views to display information about the measures in a zone map:

- `DBA_ZONEMAP_MEASURES` to display the measures for all zone maps in the database
- `ALL_ZONEMAP_MEASURES` to display the measures for zone maps that are accessible to the user
- `USER_ZONEMAP_MEASURES` to display the measures for zone maps that are owned by the user

The following query displays the zone map, measure, and column whose MIN/MAX values are maintained for each zone that are accessible to the current user:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
SELECT zonemap_name, measure, agg_function
FROM ALL_ZONEMAP_MEASURES;

ZONEMAP_NAME      MEASURE                        AGG_FUNCTION
---------------   -------------------------      -------------
ZMAP$_MY_SALES   "SH"."MY_SALES"."PROD_ID"        MAX
ZMAP$_MY_SALES   "SH"."MY_SALES"."PROD_ID"        MIN
ZMAP$_MY_SALES   "SH"."MY_SALES"."CUST_ID"        MAX
ZMAP$_MY_SALES   "SH"."MY_SALES"."CUST_ID"        MIN
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

# Part III
# Data Movement/ETL

This section discusses the tasks necessary for managing a data warehouse.

It contains the following chapters:


- Data Movement/ETL Overview
- Extraction in Data Warehouses
- Transportation in Data Warehouses
- Loading and Transformation in Data Warehouses




Copyright © 2001, 2025, Oracle and/or its affiliates.

16

# Data Movement/ETL Overview

This chapter discusses the process of extracting, transporting, transforming, and loading data in a data warehousing environment. It includes the following topics:

- Overview of ETL in Data Warehouses
- ETL Tools for Data Warehouses

## 16.1 Overview of ETL in Data Warehouses

You must load your data warehouse regularly so that it can serve its purpose of facilitating business analysis. To do this, data from one or more operational systems must be extracted and copied into the data warehouse. The challenge in data warehouse environments is to integrate, rearrange and consolidate large volumes of data over many systems, thereby providing a new unified information base for business intelligence.

The process of extracting data from source systems and bringing it into the data warehouse is commonly called ETL, which stands for extraction, transformation, and loading. Note that ETL refers to a broad process, and not three well-defined steps. The acronym ETL is perhaps too simplistic, because it omits the transportation phase and implies that each of the other phases of the process is distinct. Nevertheless, the entire process is known as ETL.

The methodology and tasks of ETL have been well known for many years, and are not necessarily unique to data warehouse environments: a wide variety of proprietary applications and database systems are the IT backbone of any enterprise. Data has to be shared between applications or systems, trying to integrate them, giving at least two applications the same picture of the world. This data sharing was mostly addressed by mechanisms similar to what is now called ETL

### 16.1.1 ETL Basics in Data Warehousing

What happens during the ETL process? The following tasks are the main actions in the process:

- Extraction of Data in Data Warehouses
- Transportation of Data in Data Warehouses

#### 16.1.1.1 Extraction of Data in Data Warehouses

During extraction, the desired data is identified and extracted from many different sources, including database systems and applications. Very often, it is not possible to identify the specific subset of interest, therefore more data than necessary has to be extracted, so the identification of the relevant data will be done at a later point in time. Depending on the source system's capabilities (for example, operating system resources), some transformations may take place during this extraction process. The size of the extracted data varies from hundreds of kilobytes up to gigabytes, depending on the source system and the business situation. The same is true for the time delta between two (logically) identical extractions: the time span may vary between days/hours and minutes to near real-time. Web server log files, for example, can easily grow to hundreds of megabytes in a very short period.


Copyright © 2001, 2025, Oracle and/or its affiliates.

### 16.1.1.2 Transportation of Data in Data Warehouses

After data is extracted, it has to be physically transported to the target system or to an intermediate system for further processing. Depending on the chosen way of transportation, some transformations can be done during this process, too. For example, a SQL statement which directly accesses a remote target through a gateway can concatenate two columns as part of the `SELECT` statement.

The emphasis in many of the examples in this section is scalability. Many long-time users of Oracle AI Database are experts in programming complex data transformation logic using PL/SQL. These chapters suggest alternatives for many such data manipulation operations, with a particular emphasis on implementations that take advantage of Oracle's new SQL functionality, especially for ETL and the parallel query infrastructure.

## 16.2 ETL Tools for Data Warehouses

Designing and maintaining the ETL process is often considered one of the most difficult and resource-intensive portions of a data warehouse project. Many data warehousing projects use ETL tools to manage this process. Oracle Data Integrator (ODI), for example, provides ETL capabilities and takes advantage of inherent database abilities. Other data warehouse builders create their own ETL tools and processes, either inside or outside the database.

Besides the support of extraction, transformation, and loading, there are some other tasks that are important for a successful ETL implementation as part of the daily operations of the data warehouse and its support for further enhancements. Besides the support for designing a data warehouse and the data flow, these tasks are typically addressed by ETL tools such as ODI.

Oracle is not an ETL tool and does not provide a complete solution for ETL. However, Oracle does provide a rich set of capabilities that can be used by both ETL tools and customized ETL solutions. Oracle offers techniques for transporting data between Oracle databases, for transforming large volumes of data, and for quickly loading new data into a data warehouse.

### 16.2.1 Daily Operations in Data Warehouses

The successive loads and transformations must be scheduled and processed in a specific order. Depending on the success or failure of the operation or parts of it, the result must be tracked and subsequent, alternative processes might be started. The control of the progress as well as the definition of a business workflow of the operations are typically addressed by ETL tools such as Oracle Data Integrator (ODI).

### 16.2.2 Evolution of the Data Warehouse

As the data warehouse is a living IT system, sources and targets might change. Those changes must be maintained and tracked through the lifespan of the system without overwriting or deleting the old ETL process flow information. To build and keep a level of trust about the information in the warehouse, the process flow of each individual record in the warehouse can be reconstructed at any point in time in the future in an ideal case.


Copyright © 2001, 2025, Oracle and/or its affiliates.

17

# Extraction in Data Warehouses

This chapter discusses extraction, which is the process of taking data from an operational system and moving it to your data warehouse or staging system. The chapter discusses:

- Overview of Extraction in Data Warehouses
- Introduction to Extraction Methods in Data Warehouses
- Data Warehousing Extraction Examples

## 17.1 Overview of Extraction in Data Warehouses

Extraction is the operation of extracting data from a source system for further use in a data warehouse environment. This is the first step of the ETL process. After the extraction, this data can be transformed and loaded into the data warehouse.

The source systems for a data warehouse are typically transaction processing applications. For example, one of the source systems for a sales analysis data warehouse might be an order entry system that records all of the current order activities.

Designing and creating the extraction process is often one of the most time-consuming tasks in the ETL process and, indeed, in the entire data warehousing process. The source systems might be very complex and poorly documented, and thus determining which data needs to be extracted can be difficult. The data has to be extracted normally not only once, but several times in a periodic manner to supply all changed data to the data warehouse and keep it up-to-date. Moreover, the source system typically cannot be modified, nor can its performance or availability be adjusted, to accommodate the needs of the data warehouse extraction process.

These are important considerations for extraction and ETL in general. This chapter, however, focuses on the technical considerations of having different kinds of sources and extraction methods. It assumes that the data warehouse team has already identified the data that will be extracted, and discusses common techniques used for extracting data from source databases.

Designing this process means making decisions about the following two main aspects:

- Which extraction method do I choose?

  This influences the source system, the transportation process, and the time needed for refreshing the warehouse.

- How do I provide the extracted data for further processing?

  This influences the transportation method, and the need for cleaning and transforming the data.

## 17.2 Introduction to Extraction Methods in Data Warehouses

The extraction method you should choose is highly dependent on the source system and also from the business needs in the target data warehouse environment. Very often, there is no possibility to add additional logic to the source systems to enhance an incremental extraction of data due to the performance or the increased workload of these systems. Sometimes even the customer is not allowed to add anything to an out-of-the-box application system.


Copyright © 2001, 2025, Oracle and/or its affiliates.

This section contains the following topics:

- Logical Extraction Methods
- Physical Extraction Methods
- Change Tracking Methods

## 17.2.1 Logical Extraction Methods

There are two types of logical extraction:

- Full Extraction
- Incremental Extraction

**Full Extraction**

The data is extracted completely from the source system. Because this extraction reflects all the data currently available on the source system, there's no need to keep track of changes to the data source since the last successful extraction. The source data will be provided as-is and no additional logical information (for example, timestamps) is necessary on the source site. An example for a full extraction may be an export file of a distinct table or a remote SQL statement scanning the complete source table.

**Incremental Extraction**

At a specific point in time, only t the data that has changed since a well-defined event back in history is extracted. This event may be the last time of extraction or a more complex business event like the last booking day of a fiscal period. To identify this delta change there must be a possibility to identify all the changed information since this specific time event. This information can be either provided by the source data itself such as an application column, reflecting the last-changed timestamp or a change table where an appropriate additional mechanism keeps track of the changes besides the originating transactions. In most cases, using the latter method means adding extraction logic to the source system.

Many data warehouses do not use any change-capture techniques as part of the extraction process. Instead, entire tables from the source systems are extracted to the data warehouse or staging area, and these tables are compared with a previous extract from the source system to identify the changed data. This approach may not have significant impact on the source systems, but it clearly can place a considerable burden on the data warehouse processes, particularly if the data volumes are large.

## 17.2.2 Physical Extraction Methods

Depending on the chosen logical extraction method and the capabilities and restrictions on the source side, the extracted data can be physically extracted by two mechanisms. The data can either be extracted online from the source system or from an offline structure. Such an offline structure might already exist or it might be generated by an extraction routine.

There are the following methods of physical extraction:

- Online Extraction
- Offline Extraction

**Online Extraction**

The data is extracted directly from the source system itself. The extraction process can connect directly to the source system to access the source tables themselves or to an


Copyright © 2001, 2025, Oracle and/or its affiliates.

intermediate system that stores the data in a preconfigured manner (for example, snapshot logs or change tables). Note that the intermediate system is not necessarily physically different from the source system.

With online extractions, you must consider whether the distributed transactions are using original source objects or prepared source objects.

**Offline Extraction**

The data is not extracted directly from the source system but is staged explicitly outside the original source system. The data already has an existing structure (for example, redo logs, archive logs or transportable tablespaces) or was created by an extraction routine.

You should consider the following structures:

- Flat files

  Data in a defined, generic format. Additional information about the source object is necessary for further processing.

- Dump files

  Oracle-specific format. Information about the containing objects may or may not be included, depending on the chosen utility.

- Redo and archive logs

  Information is in a special, additional dump file.

- Transportable tablespaces

  A powerful way to extract and move large volumes of data between Oracle databases. A more detailed example of using this feature to extract and transport data is provided in Transportation in Data Warehouses. Oracle recommends that you use transportable tablespaces whenever possible, because they can provide considerable advantages in performance and manageability over other extraction techniques.

  See *Oracle AI Database Utilities* for more information on using export/import.

## 17.2.3 Change Tracking Methods

An important consideration for extraction is incremental extraction, also called change tracking. If a data warehouse extracts data from an operational system on a nightly basis, then the data warehouse requires only the data that has changed since the last extraction (that is, the data that has been modified in the past 24 hours). Change tracking is also the key-enabling technology for providing near real-time, or on-time, data warehousing.

When it is possible to efficiently identify and extract only the most recently changed data, the extraction process (and all downstream operations in the ETL process) can be much more efficient, because it must extract a much smaller volume of data. Unfortunately, for many source systems, identifying the recently modified data may be difficult or intrusive to the operation of the system. Change tracking is typically the most challenging technical issue in data extraction.

Because change tracking is often desirable as part of the extraction process, this section describes several techniques for implementing a self-developed change capture on Oracle AI Database source systems:

- Timestamps
- Partitioning
- Triggers


Copyright © 2001, 2025, Oracle and/or its affiliates.

These techniques are based upon the characteristics of the source systems, or may require modifications to the source systems. Thus, each of these techniques must be carefully evaluated by the owners of the source system prior to implementation.

Each of these techniques can work in conjunction with the data extraction technique discussed previously. For example, timestamps can be used whether the data is being unloaded to a file or accessed through a distributed query.

### Timestamps

The tables in some operational systems have timestamp columns. The timestamp specifies the time and date that a given row was last modified. If the tables in an operational system have columns containing timestamps, then the latest data can easily be identified using the timestamp columns. For example, the following query might be useful for extracting today's data from an `orders` table:

```
SELECT * FROM orders
WHERE TRUNC(CAST(order_date AS date),'dd') =
  TO_DATE(SYSDATE,'dd-mon-yyyy');
```

If the timestamp information is not available in an operational source system, you are not always able to modify the system to include timestamps. Such modification would require, first, modifying the operational system's tables to include a new timestamp column and then creating a trigger to update the timestamp column following every operation that modifies a given row.

### Partitioning

Some source systems might use range partitioning, such that the source tables are partitioned along a date key, which allows for easy identification of new data. For example, if you are extracting from an `orders` table, and the `orders` table is partitioned by week, then it is easy to identify the current week's data.

### Triggers

Triggers can be created in operational systems to keep track of recently updated records. They can then be used in conjunction with timestamp columns to identify the exact time and date when a given row was last modified. You do this by creating a trigger on each source table that requires change data capture. Following each DML statement that is executed on the source table, this trigger updates the timestamp column with the current time. Thus, the timestamp column provides the exact time and date when a given row was last modified.

A similar internalized trigger-based technique is used for Oracle materialized view logs. These logs are used by materialized views to identify changed data, and these logs are accessible to end users. However, the format of the materialized view logs is not documented and might change over time.

Materialized view logs rely on triggers, but they provide an advantage in that the creation and maintenance of this change-data system is largely managed by the database.

Trigger-based techniques might affect performance on the source systems, and this impact should be carefully considered prior to implementation on a production source system.

## 17.3 Data Warehousing Extraction Examples

You can extract data in two ways:

- Extraction Using Data Files


Copyright © 2001, 2025, Oracle and/or its affiliates.

- Extraction Through Distributed Operations

## 17.3.1 Extraction Using Data Files

Most database systems provide mechanisms for exporting or unloading data from the internal database format into flat files. Extracts from mainframe systems often use COBOL programs, but many databases, and third-party software vendors, provide export or unload utilities.

Data extraction does not necessarily mean that entire database structures are unloaded in flat files. In many cases, it may be appropriate to unload entire database tables or objects. In other cases, it may be more appropriate to unload only a subset of a given table such as the changes on the source system since the last extraction or the results of joining multiple tables together. Different extraction techniques vary in their capabilities to support these two scenarios.

When the source system is an Oracle database, several alternatives are available for extracting data into files:

- Extracting into Flat Files Using SQL*Plus
- Extracting into Flat Files Using OCI or Pro*C Programs
- Exporting into Export Files Using the Export Utility
- Extracting into Export Files Using External Tables

### 17.3.1.1 Extracting into Flat Files Using SQL*Plus

The most basic technique for extracting data is to execute a SQL query in SQL*Plus and direct the output of the query to a file. For example, to extract a flat file, `country_city.log`, with the pipe sign as delimiter between column values, containing a list of the cities in the US in the tables `countries` and `customers`, the following SQL script could be run:

```
SET echo off SET pagesize 0 SPOOL country_city.log
SELECT distinct t1.country_name ||'|'|| t2.cust_city
FROM countries t1, customers t2 WHERE t1.country_id = t2.country_id
AND t1.country_name= 'United States of America';
SPOOL off
```

The exact format of the output file can be specified using SQL*Plus system variables.

This extraction technique offers the advantage of storing the result in a customized format. Note that, using the external table data pump unload facility, you can also extract the result of an arbitrary SQL operation. The example previously extracts the results of a join.

This extraction technique can be parallelized by initiating multiple, concurrent SQL*Plus sessions, each session running a separate query representing a different portion of the data to be extracted. For example, suppose that you wish to extract data from an `orders` table, and that the `orders` table has been range partitioned by month, with partitions `orders_jan1998`, `orders_feb1998`, and so on. To extract a single year of data from the `orders` table, you could initiate 12 concurrent SQL*Plus sessions, each extracting a single partition. The SQL script for one such session could be:

```
SPOOL order_jan.dat
SELECT * FROM orders PARTITION (orders_jan1998);
SPOOL OFF
```

These 12 SQL*Plus processes would concurrently spool data to 12 separate files. You can then concatenate them if necessary (using operating system utilities) following the extraction. If you are planning to use SQL*Loader for loading into the target, these 12 files can be used as is


Copyright © 2001, 2025, Oracle and/or its affiliates.

for a parallel load with 12 SQL*Loader sessions. See Transportation in Data Warehouses for an example.

Even if the `orders` table is not partitioned, it is still possible to parallelize the extraction either based on logical or physical criteria. The logical method is based on logical ranges of column values, for example:

```
SELECT ... WHERE order_date
BETWEEN TO_DATE('01-JAN-99') AND TO_DATE('31-JAN-99');
```

The physical method is based on a range of values. By viewing the data dictionary, it is possible to identify the Oracle AI Database data blocks that make up the `orders` table. Using this information, you could then derive a set of rowid-range queries for extracting data from the `orders` table:

```
SELECT * FROM orders WHERE rowid BETWEEN value1 and value2;
```

Parallelizing the extraction of complex SQL queries is sometimes possible, although the process of breaking a single complex query into multiple components can be challenging. In particular, the coordination of independent processes to guarantee a globally consistent view can be difficult. Unlike the SQL*Plus approach, using the external table data pump unload functionality provides transparent parallel capabilities.

Note that all parallel techniques can use considerably more CPU and I/O resources on the source system, and the impact on the source system should be evaluated before parallelizing any extraction technique.

### 17.3.1.2 Extracting into Flat Files Using OCI or Pro*C Programs

OCI programs (or other programs using Oracle call interfaces, such as Pro*C programs), can also be used to extract data. These techniques typically provide improved performance over the SQL*Plus approach, although they also require additional programming. Like the SQL*Plus approach, an OCI program can extract the results of any SQL query. Furthermore, the parallelization techniques described for the SQL*Plus approach can be readily applied to OCI programs as well.

When using OCI or SQL*Plus for extraction, you need additional information besides the data itself. At minimum, you need information about the extracted columns. It is also helpful to know the extraction format, which might be the separator between distinct columns.

### 17.3.1.3 Exporting into Export Files Using the Export Utility

The Export utility allows tables (including data) to be exported into Oracle AI Database export files. Unlike the SQL*Plus and OCI approaches, which describe the extraction of the results of a SQL statement, Export provides a mechanism for extracting database objects. Thus, Export differs from the previous approaches in several important ways:

- The export files contain metadata as well as data. An export file contains not only the raw data of a table, but also information on how to re-create the table, potentially including any indexes, constraints, grants, and other attributes associated with that table.
- A single export file may contain a subset of a single object, many database objects, or even an entire schema.
- Export cannot be directly used to export the results of a complex SQL query. Export can be used only to extract subsets of distinct database objects.
- The Fxport utility can create Data Pump files locally. Also, in cases where the data is being copied to an object store, the Export utility can copy the files directly into the object store.


Copyright © 2001, 2025, Oracle and/or its affiliates.

- When importing into Oracle AI Database, the output of the Export utility must be processed using the Import utility.

Oracle AI Database provides the original Export and Import utilities for backward compatibility and the data pump export/import infrastructure for high-performant, scalable and parallel extraction. See *Oracle AI Database Utilities* for further details.

### 17.3.1.4 Extracting into Export Files Using External Tables

In addition to the Export Utility, you can use external tables to extract the results from any `SELECT` operation. The data is stored in the platform independent, Oracle-internal data pump format and can be processed as regular external table on the target system.

The following example extracts the result of a join operation in parallel into the four specified files. The only allowed external table type for extracting data is the Oracle-internal format `ORACLE_DATAPUMP`.

```
CREATE DIRECTORY def_dir AS '/net/private/jdoe/WORK/FEATURES/et';
DROP TABLE extract_cust;
CREATE TABLE extract_cust
ORGANIZATION EXTERNAL
(TYPE ORACLE_DATAPUMP DEFAULT DIRECTORY def_dir ACCESS PARAMETERS
(NOBADFILE NOLOGFILE)
LOCATION ('extract_cust1.exp', 'extract_cust2.exp', 'extract_cust3.exp',
          'extract_cust4.exp'))
PARALLEL 4 REJECT LIMIT UNLIMITED AS
SELECT  c.*, co.country_name, co.country_subregion, co.country_region
FROM customers c, countries co where co.country_id=c.country_id;
```

The total number of extraction files specified limits the maximum degree of parallelism for the write operation. Note that the parallelizing of the extraction does not automatically parallelize the `SELECT` portion of the statement.

Unlike using any kind of export/import, the metadata for the external table is not part of the created files when using the external table data pump unload. To extract the appropriate metadata for the external table, use the `DBMS_METADATA` package, as illustrated in the following statement:

```
SET LONG 2000
SELECT DBMS_METADATA.GET_DDL('TABLE','EXTRACT_CUST') FROM DUAL;
```

## 17.3.2 Extraction Through Distributed Operations

Using distributed-query technology, one Oracle database can directly query tables located in various different source systems, such as another Oracle database or a legacy system connected with the Oracle gateway technology. Specifically, a data warehouse or staging database can directly access tables and data located in a connected source system. Gateways are another form of distributed-query technology. Gateways allow an Oracle database (such as a data warehouse) to access database tables stored in remote, non-Oracle databases. This is the simplest method for moving data between two Oracle databases because it combines the extraction and transformation into a single step, and requires minimal programming. However, this is not always feasible.

Suppose that you wanted to extract a list of employee names with department names from a source database and store this data into the data warehouse. Using an Oracle Net connection and distributed-query technology, this can be achieved using a single SQL statement:

```
CREATE TABLE country_city AS SELECT distinct t1.country_name, t2.cust_city
FROM countries@source_db t1, customers@source_db t2
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
WHERE t1.country_id = t2.country_id
AND t1.country_name='United States of America';
```

This statement creates a local table in a data mart, `country_city`, and populates it with data from the `countries` and `customers` tables on the source system.

This technique is ideal for moving small volumes of data. However, the data is transported from the source system to the data warehouse through a single Oracle Net connection. Thus, the scalability of this technique is limited. For larger data volumes, file-based data extraction and transportation techniques are often more scalable and thus more appropriate.

### See Also

- *Oracle Database Heterogeneous Connectivity User's Guide* for more information regarding distributed queries
- *Oracle Database Concepts* for more information regarding distributed queries


Copyright © 2001, 2025, Oracle and/or its affiliates.

# 18
# Transportation in Data Warehouses

The following topics provide information about transporting data into a data warehouse:

- Overview of Transportation in Data Warehouses
- Introduction to Transportation Mechanisms in Data Warehouses

## 18.1 Overview of Transportation in Data Warehouses

Transportation is the operation of moving data from one system to another system. In a data warehouse environment, the most common requirements for transportation are in moving data from:

- A source system to a staging database or a data warehouse database
- A staging database to a data warehouse
- A data warehouse to a data mart

Transportation is often one of the simpler portions of the ETL process, and can be integrated with other portions of the process. For example, as shown in Extraction in Data Warehouses, distributed query technology provides a mechanism for both extracting and transporting data.

## 18.2 Introduction to Transportation Mechanisms in Data Warehouses

You have three basic choices for transporting data in warehouses:

- Transportation Using Flat Files
- Transportation Through Distributed Operations
- Transportation Using Transportable Tablespaces

### 18.2.1 Transportation Using Flat Files

The most common method for transporting data is by the transfer of flat files, using mechanisms such as FTP or other remote file system access protocols. Data is unloaded or exported from the source system into flat files using techniques discussed in Extraction in Data Warehouses, and is then transported to the target platform using FTP or similar mechanisms.

Because source systems and data warehouses often use different operating systems and database systems, using flat files is often the simplest way to exchange data between heterogeneous systems with minimal transformations. However, even when transporting data between homogeneous systems, flat files are often the most efficient and most easy-to-manage mechanism for data transfer.


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 18.2.2 Transportation Through Distributed Operations

Distributed queries, either with or without gateways, can be an effective mechanism for extracting data. These mechanisms also transport the data directly to the target systems, thus providing both extraction and transformation in a single step. Depending on the tolerable impact on time and system resources, these mechanisms can be well suited for both extraction and transformation.

As opposed to flat file transportation, the success or failure of the transportation is recognized immediately with the result of the distributed query or transaction.

**See Also**

- Extraction in Data Warehouses for further information

## 18.2.3 Transportation Using Transportable Tablespaces

Oracle transportable tablespaces are the fastest way for moving large volumes of data between two Oracle databases. Previous to the introduction of transportable tablespaces, the most scalable data transportation mechanisms relied on moving flat files containing raw data. These mechanisms required that data be unloaded or exported into files from the source database, Then, after transportation, these files were loaded or imported into the target database. Transportable tablespaces entirely bypass the unload and reload steps.

Using transportable tablespaces, Oracle data files (containing table data, indexes, and almost every other Oracle database object) can be directly transported from one database to another. Furthermore, like import and export, transportable tablespaces provide a mechanism for transporting metadata in addition to transporting data.

Transportable tablespaces have some limitations: source and target systems must be running Oracle8*i* (or higher), must use compatible character sets, and, before Oracle Database 10*g*, must run on the same operating system. For details on how to transport tablespace between operating systems, see *Oracle Database Administrator's Guide*.

The most common applications of transportable tablespaces in data warehouses are in moving data from a staging database to a data warehouse, or in moving data from a data warehouse to a data mart.

This section contains the following topics:

- Using Transportable Tablespaces to Transport Data into Data Warehouses: Example
- Other Uses of Transportable Tablespaces

### 18.2.3.1 Using Transportable Tablespaces to Transport Data into Data Warehouses: Example

Suppose that you have a data warehouse containing sales data, and several data marts that are refreshed monthly. Also suppose that you are going to move one month of sales data from the data warehouse to the data mart.

Use the following steps to create a transportable tablespace:

1. Place the Data to be Transported into its own Tablespace


Copyright © 2001, 2025, Oracle and/or its affiliates.

2. Export the Metadata
3. Copy the Datafiles and Export File to the Target System
4. Import the Metadata

**Place the Data to be Transported into its own Tablespace**

The current month's data must be placed into a separate tablespace in order to be transported. In this example, you have a tablespace `ts_temp_sales`, which holds a copy of the current month's data. Using the `CREATE TABLE ... AS SELECT` statement, the current month's data can be efficiently copied to this tablespace:

```
CREATE TABLE temp_jan_sales NOLOGGING TABLESPACE ts_temp_sales
AS SELECT * FROM sales
WHERE time_id BETWEEN '31-DEC-1999' AND '01-FEB-2000';
```

Following this operation, the tablespace `ts_temp_sales` is set to read-only:

```
ALTER TABLESPACE ts_temp_sales READ ONLY;
```

A tablespace cannot be transported unless there are no active transactions modifying the tablespace. Setting the tablespace to read-only enforces this.

The tablespace `ts_temp_sales` may be a tablespace that has been especially created to temporarily store data for use by the transportable tablespace features. Following "Copy the Datafiles and Export File to the Target System", this tablespace can be set to read/write, and, if desired, the table `temp_jan_sales` can be dropped, or the tablespace can be re-used for other transportations or for other purposes.

In a given transportable tablespace operation, all of the objects in a given tablespace are transported. Although only one table is being transported in this example, the tablespace `ts_temp_sales` could contain multiple tables. For example, perhaps the data mart is refreshed not only with the new month's worth of sales transactions, but also with a new copy of the customer table. Both of these tables could be transported in the same tablespace. Moreover, this tablespace could also contain other database objects such as indexes, which would also be transported.

Additionally, in a given transportable-tablespace operation, multiple tablespaces can be transported at the same time. This makes it easier to move very large volumes of data between databases. Note, however, that the transportable tablespace feature can only transport a set of tablespaces which contain a complete set of database objects without dependencies on other tablespaces. For example, an index cannot be transported without its table, nor can a partition be transported without the rest of the table. You can use the `DBMS_TTS` package to check that a tablespace is transportable.

> **See Also**
>
> *Oracle Database PL/SQL Packages and Types Reference* for detailed information about the `DBMS_TTS` package

In this step, you have copied the January sales data into a separate tablespace; however, in some cases, it may be possible to leverage the transportable tablespace feature without even moving data to a separate tablespace. If the sales table has been partitioned by month in the data warehouse and if each partition is in its own tablespace, then it may be possible to directly transport the tablespace containing the January data. Suppose the January partition, `sales_jan2000`, is located in the tablespace `ts_sales_jan2000`. Then the tablespace


Copyright © 2001, 2025, Oracle and/or its affiliates.

ts_sales_jan2000 could potentially be transported, rather than creating a temporary copy of the January sales data in the ts_temp_sales.

However, the same conditions must be satisfied in order to transport the tablespace ts_sales_jan2000 as are required for the specially created tablespace. First, this tablespace must be set to READ ONLY. Second, because a single partition of a partitioned table cannot be transported without the remainder of the partitioned table also being transported, it is necessary to exchange the January partition into a separate table (using the ALTER TABLE statement) to transport the January data. The EXCHANGE operation is very quick, but the January data will no longer be a part of the underlying sales table, and thus may be unavailable to users until this data is exchanged back into the sales table after the export of the metadata. The January data can be exchanged back into the sales table after you complete the step "Copy the Datafiles and Export File to the Target System".

### Export the Metadata

The Export utility is used to export the metadata describing the objects contained in the transported tablespace. For our example scenario, the Export command could be:

```
EXP TRANSPORT_TABLESPACE=y TABLESPACES=ts_temp_sales FILE=jan_sales.dmp
```

This operation generates an export file, jan_sales.dmp. The export file is small, because it contains only metadata. In this case, the export file contains information describing the table temp_jan_sales, such as the column names, column data type, and all other information that the target Oracle database needs in order to access the objects in ts_temp_sales.

### Copy the Datafiles and Export File to the Target System

Copy the data files that make up ts_temp_sales, as well as the export file jan_sales.dmp to the data mart platform, using any transportation mechanism for flat files. Once the datafiles have been copied, the tablespace ts_temp_sales can be set to READ WRITE mode if desired.

### Import the Metadata

Once the files have been copied to the data mart, the metadata should be imported into the data mart:

```
IMP TRANSPORT_TABLESPACE=y DATAFILES='/db/tempjan.f'
    TABLESPACES=ts_temp_sales FILE=jan_sales.dmp
```

At this point, the tablespace ts_temp_sales and the table temp_sales_jan are accessible in the data mart. You can incorporate this new data into the data mart's tables.

You can insert the data from the temp_sales_jan table into the data mart's sales table in one of two ways:

```
INSERT /*+ APPEND */ INTO sales SELECT * FROM temp_sales_jan;
```

Following this operation, you can delete the temp_sales_jan table (and even the entire ts_temp_sales tablespace).

Alternatively, if the data mart's sales table is partitioned by month, then the new transported tablespace and the temp_sales_jan table can become a permanent part of the data mart. The temp_sales_jan table can become a partition of the data mart's sales table:

```
ALTER TABLE sales ADD PARTITION sales_00jan VALUES
  LESS THAN (TO_DATE('01-feb-2000','dd-mon-yyyy'));
ALTER TABLE sales EXCHANGE PARTITION sales_00jan
  WITH TABLE temp_sales_jan INCLUDING INDEXES WITH VALIDATION;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

### 18.2.3.2 Other Uses of Transportable Tablespaces

The previous example illustrates a typical scenario for transporting data in a data warehouse. However, transportable tablespaces can be used for many other purposes. In a data warehousing environment, transportable tablespaces should be viewed as a utility (much like Import/Export or SQL*Loader), whose purpose is to move large volumes of data between Oracle databases. When used in conjunction with parallel data movement operations such as the `CREATE TABLE ... AS SELECT` and `INSERT ... AS SELECT` statements, transportable tablespaces provide an important mechanism for quickly transporting data for many purposes.


Copyright © 2001, 2025, Oracle and/or its affiliates.

19

# Loading and Transformation in Data Warehouses

This chapter helps you create and manage a data warehouse, and discusses:

- Overview of Loading and Transformation in Data Warehouses
- Loading Mechanisms for Data Warehouses
- Transformation Mechanisms in Data Warehouses
- Error Logging and Handling Mechanisms
- Loading and Transformation Scenarios

## 19.1 Overview of Loading and Transformation in Data Warehouses

Data transformations are often the most complex and, in terms of processing time, the most costly part of the extraction, transformation, and loading (ETL) process. They can range from simple data conversions to extremely complex data scrubbing techniques. Many, if not all, data transformations can occur within an Oracle database, although transformations are often implemented outside of the database (for example, on flat files) as well.

This chapter introduces techniques for implementing scalable and efficient data transformations within the Oracle AI Database. The examples in this chapter are relatively simple. Real-world data transformations are often considerably more complex. However, the transformation techniques introduced in this chapter meet the majority of real-world data transformation requirements, often with more scalability and less programming than alternative approaches.

This chapter does not seek to illustrate all of the typical transformations that would be encountered in a data warehouse, but to demonstrate the types of fundamental technology that can be applied to implement these transformations and to provide guidance in how to choose the best techniques.

### 19.1.1 Data Warehouses: Transformation Flow

From an architectural perspective, you can transform your data in the following ways:

- Multistage Data Transformation in Data Warehouses
- Pipelined Data Transformation in Data Warehouses
- Staging Area in Data Warehouses

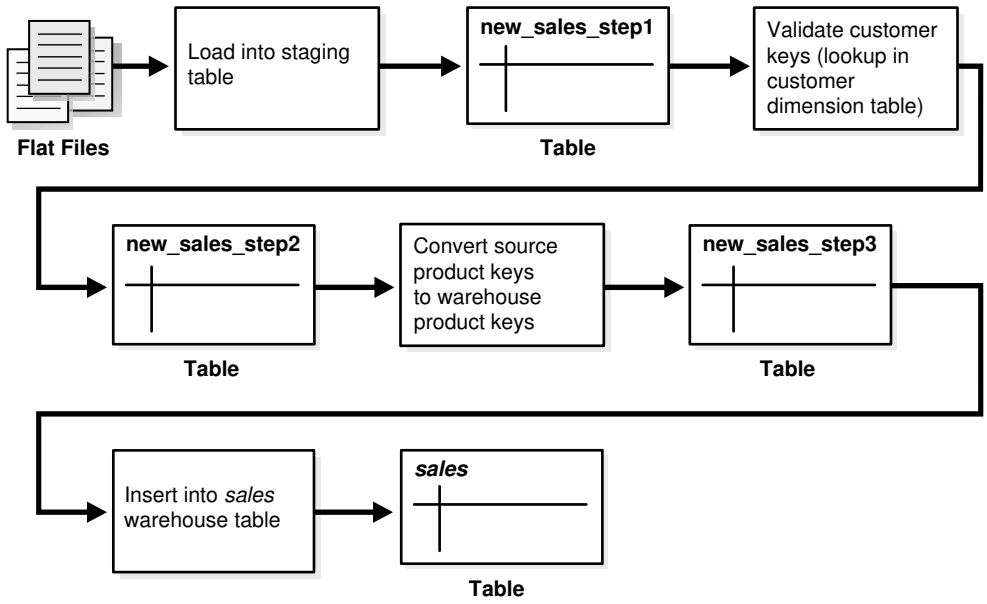
#### 19.1.1.1 Multistage Data Transformation in Data Warehouses

The data transformation logic for most data warehouses consists of multiple steps. For example, in transforming new records to be inserted into a sales table, there may be separate logical transformation steps to validate each dimension key.


Copyright © 2001, 2025, Oracle and/or its affiliates.

Figure 19-1 offers a graphical way of looking at the transformation logic.

**Figure 19-1 Multistage Data Transformation**




When using Oracle AI Database as a transformation engine, a common strategy is to implement each transformation as a separate SQL operation and to create a separate, temporary staging table (such as the tables `new_sales_step1` and `new_sales_step2` in Figure 19-1) to store the incremental results for each step. This load-then-transform strategy also provides a natural checkpointing scheme to the entire transformation process, which enables the process to be more easily monitored and restarted. However, a disadvantage to multistaging is that the space and time requirements increase.

It may also be possible to combine many simple logical transformations into a single SQL statement or single PL/SQL procedure. Doing so may provide better performance than performing each step independently, but it may also introduce difficulties in modifying, adding, or dropping individual transformations, as well as recovering from failed transformations.

## 19.1.1.2 Pipelined Data Transformation in Data Warehouses

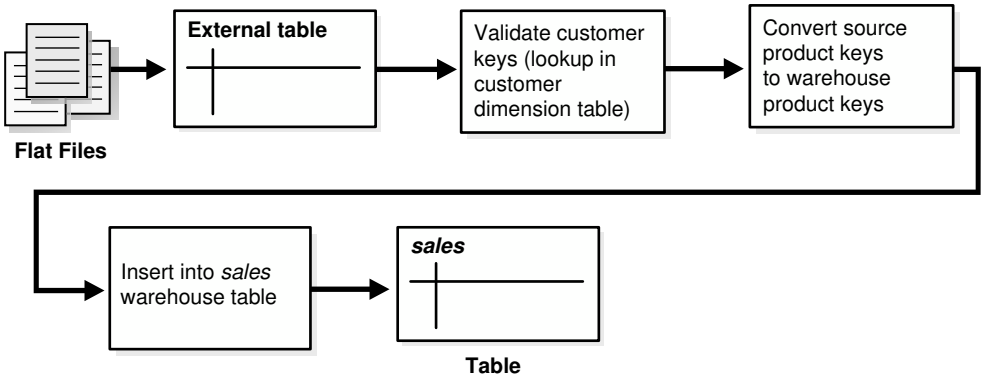
The ETL process flow can be changed dramatically and the database becomes an integral part of the ETL solution.

The new functionality renders some of the former necessary process steps obsolete while some others can be remodeled to enhance the data flow and the data transformation to become more scalable and non-interruptive. The task shifts from serial transform-then-load process (with most of the tasks done outside the database) or load-then-transform process, to an enhanced transform-while-loading.

Oracle offers a wide variety of new capabilities to address all the issues and tasks relevant in an ETL scenario. It is important to understand that the database offers toolkit functionality rather than trying to address a one-size-fits-all solution. The underlying database has to enable the most appropriate ETL process flow for a specific customer need, and not dictate or constrain it from a technical perspective. Figure 19-2 illustrates the new functionality, which is discussed throughout later sections.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Figure 19-2 Pipelined Data Transformation**




### 19.1.1.3 Staging Area in Data Warehouses

The overall speed of your load is determined by how quickly the raw data can be read from the staging area and written to the target table in the database. It is highly recommended that you stage your raw data across as many physical disks as possible to ensure the reading of the raw data is not a bottleneck during the load.

An excellent place to stage the data is in a Database File System (DBFS). DBFS creates a mountable file system which can be used to access files stored in the database as SecureFiles LOBs. DBFS is similar to NFS in that it provides a shared network file system that looks like a local file system. Oracle recommends that you create the DBFS in a separate database from the data warehouse, and that the file system be mounted using the `DIRECT_IO` option to avoid contention on the system page cache while moving the raw data files in and out of the file system. More information on setting up DBFS can be found in *Oracle AI Database SecureFiles and Large Objects Developer's Guide*.

## 19.1.2 About Batch Updates and Online Table Redefinition

You can optimize bulk updates to the table by using the `EXECUTE_UPDATE` procedure. Because the updates are not logged in the redo log, performance is optimized.

The `DBMS_REDEFINITION.EXECUTE_UPDATE` procedure allows you to run `UPDATE` statements in direct insert mode. Because redo is not logged during this operation, you cannot recover the redefinition and data updates using media recovery. To maintain recoverability, it is recommended that a database or tablespace backup be performed before the redefinition begins.

> **See Also**
>
> *Oracle AI Database Administrator's Guide*

## 19.1.3 Overview of Monitoring ETL Operations

Because ETL can become complex and suffer from poor performance, Oracle AI Database provides a user interface that enables you to monitor and report on database operations that are part of an ETL plan.


Copyright © 2001, 2025, Oracle and/or its affiliates.

A database operation is a user-defined logical object that contains a set of related database tasks, for example an ETL processing job, defined by end users or application code. Each database operation is uniquely identified by its name and execution ID and can be executed multiple times.

Database operation monitoring is extremely useful for troubleshooting a suboptimally performing job and helps to identify where and how much resources are being consumed at any given step. It enables you to track related information, identify performance bottlenecks, and reduce the time to tune database performance problems.You can begin a database operation on an arbitrary session by specifying its session ID and serial number in the `DBMS_SQL_MONITOR.BEGIN_OPERATION` function.

> **See Also**
>
> *Oracle AI Database SQL Tuning Guide*

# 19.2 Loading Mechanisms for Data Warehouses

You can use the following mechanisms for loading a data warehouse:

- Loading a Data Warehouse with SQL*Loader
- Loading a Data Warehouse with External Tables
- Loading a Data Warehouse with OCI and Direct-Path APIs
- Loading a Data Warehouse with Export/Import

## 19.2.1 Loading a Data Warehouse with SQL*Loader

Before any data transformations can occur within the database, the raw data must become accessible for the database. One approach is to load it into the database. Transportation in Data Warehouses, discusses several techniques for transporting data to an Oracle data warehouse. Perhaps the most common technique for transporting data is by way of flat files.

SQL*Loader is used to move data from flat files into an Oracle data warehouse. During this data load, SQL*Loader can also be used to implement basic data transformations. When using direct-path SQL*Loader, basic data manipulation, such as data type conversion and simple `NULL` handling, can be automatically resolved during the data load. Most data warehouses use direct-path loading for performance reasons.

The conventional-path loader provides broader capabilities for data transformation than a direct-path loader: SQL functions can be applied to any column as those values are being loaded. This provides a rich capability for transformations during the data load. However, the conventional-path loader is slower than direct-path loader. For these reasons, the conventional-path loader should be considered primarily for loading and transforming smaller amounts of data.

Data warehouses can use direct path mode to run batch updates to avoid the overhead of maintaining redo data. You can run batch updates on a table during online table redefinition.

The following is a simple example of a SQL*Loader control file to load data into the `sales` table of the `sh` sample schema from an external file `sh_sales.dat`. The external flat file `sh_sales.dat` consists of sales transaction data, aggregated on a daily level. Not all columns of this external file are loaded into `sales`. This external file is also used as a source for loading the second fact table of the `sh` sample schema, which is done using an external table:


Copyright © 2001, 2025, Oracle and/or its affiliates.

The following shows the control file (`sh_sales.ctl`) loading the `sales` table:

```
LOAD DATA INFILE sh_sales.dat APPEND INTO TABLE sales
FIELDS TERMINATED BY "|"
(PROD_ID, CUST_ID, TIME_ID, CHANNEL_ID, PROMO_ID, QUANTITY_SOLD, AMOUNT_SOLD)
```

It can be loaded with the following command:

```
$  sqlldr control=sh_sales.ctl direct=true
Username:
Password:
```

In the case of SQL*Loader Express mode, you do not use a control file. Instead, it uses table column definitions to determine input data types.

**See Also**

- *Oracle AI Database Utilities* for more information.

  Note that as described in Automatic Parallel Load of Table Data with SQL*Loader in the *Oracle Database Utilities* guide, the SQL*Loader client can automatically start a parallel direct path load for data without dividing the data into separate files and starting multiple SQL*Loader clients. The data does not need to be on the database server. Cloud users can employ this feature to load data in parallel without having to move the data onto the cloud system (if sufficient network bandwidth is available).
- *Oracle AI Database Administrator's Guide* for information about bulk updates using the `DBMS_REDEFINITION` package

### 19.2.1.1 Using SQL*Loader to Load From an Object Store

SQL*Loader can load data from files in an object store into Oracle AI Database tables.

**Note**

To manage wallets, Oracle recommends that you use the `orapki` command line tool.

The loader must pass a `CREDENTIAL` parameter for authentication against the object store.

Before you start, use the `orapki` utility to create an Oracle Wallet if you do not already have one that you want to use. You can specify any wallet path.

```
orapki wallet create -wallet /u01/app/oracle/product/wallet/ -pwd password -
auto_login
```

1. Create the CREDENTIAL.
   Use the mkstore utility to create the `CREDENTIAL` and the object store username and password entries in the wallet.


Copyright © 2001, 2025, Oracle and/or its affiliates.

a. Create the CREDENTIAL (*credential_name*) and at the same time add the username (*object_store_username*) that will be authenticated by the object store:

```
mkstore -wrl wallet_location_directory -createEntry
oracle.sqlldr.credential.credential_name.username object_store_username
```

b. Add the password associated with the username.

```
mkstore -wrl wallet_location_directory -createEntry
oracle.sqlldr.credential.same_credential_name.password
object_store_user_password
```

This example creates CREDENTIAL cred1 for the user djones. In both command responses, mkstore prompts for the wallet password.

```
mkstore -wrl /u01/app/oracle/product/wallet/ -createEntry
oracle.sqlldr.credential.cred1.username djones
Enter wallet password:
mkstore -wrl /u01/app/oracle/product/wallet/ -createEntry
oracle.sqlldr.credential.cred1.password Z!1A4z96
Enter wallet password:
```

2. Create a control file.
The INFILE parameter in the example below points to a CSV file in the object store. In this case, the data from the file is loaded into the table "DEPTOS" in Oracle AI Database.

```
LOAD DATA
INFILE  'https://domain.example.com/v1/pkistore/dept.csv'
truncate
INTO TABLE DEPTOS
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
(DEPTNO, DNAME, LOC)
```

Note that you can either provide the URL in the control file as shown above or set it as the value of the DATA parameter in the sqldir command.

3. Run SQL*Loader.
Include the CREDENTIAL parameter in the sqldir command:

```
sqlldr sqlldr/test@cdb1_pdb6 dept.ctl credential=cred1 log=dept.log
external_table=not_used proxy=https://www.example.com:80
```

## 19.2.2 Loading a Data Warehouse with External Tables

Another approach for handling external data sources is using external tables. Oracle's external table feature enables you to use external data as a virtual table that can be queried and joined directly and in parallel without requiring the external data to be first loaded in the database. You can then use SQL, PL/SQL, and Java to access the external data.

External tables enable the pipelining of the loading phase with the transformation phase. The transformation process can be merged with the loading process without any interruption of the data streaming. It is no longer necessary to stage the data inside the database for further processing inside the database, such as comparison or transformation. For example, the conversion functionality of a conventional load can be used for a direct-path INSERT AS SELECT


Copyright © 2001, 2025, Oracle and/or its affiliates.

statement in conjunction with the SELECT from an external table. Starting in Oracle Database 12*c*, the database automatically gathers table statistics as part of a bulk-load operation (CTAS and IAS) similar to how statistics are gathered when an index is created. By gathering statistics during the data load, you avoid additional scan operations and provide the necessary statistics as soon as the data becomes available to the users.

The main difference between external tables and regular tables is that externally organized tables are read-only. No DML operations (UPDATE/INSERT/DELETE) are possible and no indexes can be created on them.

External tables are mostly compliant with the existing SQL*Loader functionality and provide superior functionality in most cases. External tables are especially useful for environments where the complete external source has to be joined with existing database objects or when the data has to be transformed in a complex manner. For example, unlike SQL*Loader, you can apply any arbitrary SQL transformation and use the direct-path insert method. In addition, you can specify a program to be executed (such as zcat) that processes files (such as compressed data files) and enables Oracle AI Database to use the output (such as uncompressed data files), which means you can load large amounts of compressed data without first uncompressing it on a disk.

You can create an external table named sales_transactions_ext, representing the structure of the complete sales transaction data, represented in the external file sh_sales.gz. The product department is especially interested in a cost analysis on product and time. You thus create a fact table named cost in the sh schema. The operational source data is the same as for the sales fact table. However, because you are not investigating every dimensional information that is provided, the data in the cost fact table has a coarser granularity than in the sales fact table, for example, all different distribution channels are aggregated.

You cannot load the data into the cost fact table without applying the previously mentioned aggregation of the detailed information, due to the suppression of some of the dimensions.

The external table framework offers a solution to solve this. Unlike SQL*Loader, where you would have to load the data before applying the aggregation, you can combine the loading and transformation within a single SQL DML statement, as shown in the following. You do not have to stage the data temporarily before inserting into the target table.

The object directories must already exist, and point to the directory containing the sh_sales.gz file as well as the directory containing the bad and log files.

```
CREATE TABLE sales_transactions_ext
(PROD_ID NUMBER, CUST_ID NUMBER,
 TIME_ID DATE, CHANNEL_ID NUMBER,
 PROMO_ID NUMBER, QUANTITY_SOLD NUMBER,
 AMOUNT_SOLD NUMBER(10,2), UNIT_COST NUMBER(10,2),
 UNIT_PRICE NUMBER(10,2))
ORGANIZATION external (TYPE oracle_loader
  DEFAULT DIRECTORY data_file_dir ACCESS PARAMETERS
  (RECORDS DELIMITED BY NEWLINE CHARACTERSET US7ASCII
    PREPROCESSOR EXECDIR:'zcat'
    BADFILE log_file_dir:'sh_sales.bad_xt'
    LOGFILE log_file_dir:'sh_sales.log_xt'
    FIELDS TERMINATED BY "|" LDRTRIM
    ( PROD_ID, CUST_ID,
      TIME_ID         DATE(10) "YYYY-MM-DD",
      CHANNEL_ID, PROMO_ID, QUANTITY_SOLD, AMOUNT_SOLD,
      UNIT_COST, UNIT_PRICE))
  location ('sh_sales.gz')
)REJECT LIMIT UNLIMITED;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

The external table can now be used from within the database, accessing some columns of the external data only, grouping the data, and inserting it into the `costs` fact table:

```
INSERT /*+ APPEND */ INTO COSTS
(TIME_ID, PROD_ID, UNIT_COST, UNIT_PRICE)
SELECT TIME_ID, PROD_ID, AVG(UNIT_COST), AVG(amount_sold/quantity_sold)
FROM sales_transactions_ext GROUP BY time_id, prod_id;
```

**See Also**

- *Oracle Database SQL Language Reference* for a complete description of external table syntax
- *Oracle Database Utilities* for usage examples

### 19.2.2.1 Using DBMS_CLOUD to Create External Tables for Object Store Data

The DBMS_CLOUD PL/SQL package enables you to connect the data warehouse to object stores in the Cloud.

`DBMS_CLOUD` provides APIs to create external tables and enable access to data from files and objects stored in the Cloud. You can load data from text, Parquet, and Avro files as well as Data Pump files in the Cloud into external tables.

Authentication against the object store is acquired through a separately-created credential object which includes a username and password. The object store administrator must provide these credentials and provision the user with appropriate permissions to access data in the store.

The package supports loading files from Oracle Object Storage, Microsoft Azure Blob Storage, and Amazon S3.

**See Also**

*Database PL/SQL Packages and Types Reference*, which describes the `DBMS_CLOUD` APIs.

## 19.2.3 Loading a Data Warehouse with OCI and Direct-Path APIs

OCI and direct-path APIs are frequently used when the transformation and computation are done outside the database and there is no need for flat file staging.

## 19.2.4 Loading a Data Warehouse with Export/Import

Export and import are used when the data is inserted as is into the target system. No complex extractions are possible. See Extraction in Data Warehouses for further information.

# 19.3 Transformation Mechanisms in Data Warehouses

You have the following choices for transforming data inside the database:

- Transforming Data Using SQL


Copyright © 2001, 2025, Oracle and/or its affiliates.

- Transforming Data Using PL/SQL
- Transforming Data Using Table Functions

## 19.3.1 Transforming Data Using SQL

Once data is loaded into the database, data transformations can be executed using SQL operations. There are four basic techniques for implementing SQL data transformations:

- CREATE TABLE ... AS SELECT And INSERT /*+APPEND*/ AS SELECT
- Transforming Data Using UPDATE
- Transforming Data Using MERGE
- Transforming Data Using Multitable INSERT

### 19.3.1.1 CREATE TABLE ... AS SELECT And INSERT /*+APPEND*/ AS SELECT

The `CREATE TABLE ... AS SELECT` statement (CTAS) is a powerful tool for manipulating large sets of data. As shown in the following example, many data transformations can be expressed in standard SQL, and CTAS provides a mechanism for efficiently executing a SQL query and storing the results of that query in a new database table. The `INSERT /*+APPEND*/ ... AS SELECT` statement offers the same capabilities with existing database tables.

In a data warehouse environment, CTAS is typically run in parallel using `NOLOGGING` mode for best performance.

A simple and common type of data transformation is data substitution. In a data substitution transformation, some or all of the values of a single column are modified. For example, our `sales` table has a `channel_id` column. This column indicates whether a given sales transaction was made by a company's own sales force (a direct sale) or by a distributor (an indirect sale).

You may receive data from multiple source systems for your data warehouse. Suppose that one of those source systems processes only direct sales, and thus the source system does not know indirect sales channels. When the data warehouse initially receives sales data from this system, all sales records have a `NULL` value for the `sales.channel_id` field. These `NULL` values must be set to the proper key value. For example, you can do this efficiently using a SQL function as part of the insertion into the target sales table statement. The structure of source table `sales_activity_direct` is as follows:

```
DESC sales_activity_direct
Name           Null?    Type
------------   -----    ----------------
SALES_DATE              DATE
PRODUCT_ID              NUMBER
CUSTOMER_ID             NUMBER
PROMOTION_ID            NUMBER
AMOUNT                  NUMBER
QUANTITY                NUMBER
```

The following SQL statement inserts data from `sales_activity_direct` into the `sales` table of the sample schema, using a SQL function to truncate the sales date values to the midnight time and assigning a fixed channel ID of 3.

```
INSERT /*+ APPEND NOLOGGING PARALLEL */
INTO sales SELECT product_id, customer_id, TRUNC(sales_date), 3,
       promotion_id, quantity, amount
FROM sales_activity_direct;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

### 19.3.1.2 Transforming Data Using UPDATE

Another technique for implementing a data substitution is to use an UPDATE statement to modify the sales.channel_id column. An UPDATE provides the correct result. However, if the data substitution transformations require that a very large percentage of the rows (or all of the rows) be modified, then, it may be more efficient to use a CTAS statement than an UPDATE.

### 19.3.1.3 Transforming Data Using MERGE

Oracle AI Database's merge functionality extends SQL, by introducing the SQL keyword MERGE, in order to provide the ability to update or insert a row conditionally into a table or out of line single table views. Conditions are specified in the ON clause. This is, besides pure bulk loading, one of the most common operations in data warehouse synchronization.

**Example 19-1 Merge Operation Using SQL**

The following example discusses various implementations of a merge. It assumes that new data for the dimension table products is propagated to the data warehouse and has to be either inserted or updated. The table products_delta has the same structure as products.

```
MERGE INTO products t USING products_delta s
ON (t.prod_id=s.prod_id)
WHEN MATCHED THEN UPDATE SET
  t.prod_list_price=s.prod_list_price, t.prod_min_price=s.prod_min_price
WHEN NOT MATCHED THEN INSERT (prod_id, prod_name, prod_desc, prod_subcategory,
  prod_subcategory_desc, prod_category, prod_category_desc, prod_status,
  prod_list_price, prod_min_price)
VALUES (s.prod_id, s.prod_name, s.prod_desc, s.prod_subcategory,
  s.prod_subcategory_desc, s.prod_category, s.prod_category_desc,
  s.prod_status, s.prod_list_price, s.prod_min_price);
```

### 19.3.1.4 Transforming Data Using Multitable INSERT

Many times, external data sources have to be segregated based on logical attributes for insertion into different target objects. It is also frequent in data warehouse environments to fan out the same source data into several target objects. Multitable inserts provide a new SQL statement for these kinds of transformations, where data can either end up in several or exactly one target, depending on the business transformation rules. This insertion can be done conditionally based on business rules or unconditionally.

It offers the benefits of the INSERT ... SELECT statement when multiple tables are involved as targets. In doing so, it avoids the drawbacks of the two obvious alternatives. You either had to deal with *n* independent INSERT … SELECT statements, thus processing the same source data *n* times and increasing the transformation workload *n* times. Alternatively, you had to choose a procedural approach with a per-row determination how to handle the insertion. This solution lacked direct access to high-speed access paths available in SQL.

As with the existing INSERT ... SELECT statement, the new statement can be parallelized and used with the direct-load mechanism for faster performance.

**Example 19-2 Unconditional Insert**

The following statement aggregates the transactional sales information, stored in sales_activity_direct, on a daily basis and inserts into both the sales and the costs fact table for the current day.

```
INSERT ALL
   INTO sales VALUES (product_id, customer_id, today, 3, promotion_id,
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
                      quantity_per_day, amount_per_day)
   INTO costs VALUES (product_id, today, promotion_id, 3,
                      product_cost, product_price)
SELECT TRUNC(s.sales_date) AS today, s.product_id, s.customer_id,
  s.promotion_id, SUM(s.amount) AS amount_per_day, SUM(s.quantity)
  quantity_per_day, p.prod_min_price*0.8 AS product_cost, p.prod_list_price
  AS product_price
FROM sales_activity_direct s, products p
WHERE s.product_id = p.prod_id AND TRUNC(sales_date) = TRUNC(SYSDATE)
GROUP BY TRUNC(sales_date), s.product_id, s.customer_id, s.promotion_id,
  p.prod_min_price*0.8, p.prod_list_price;
```

### Example 19-3 Conditional ALL Insert

The following statement inserts a row into the `sales` and `costs` tables for all sales transactions with a valid promotion and stores the information about multiple identical orders of a customer in a separate table `cum_sales_activity`. It is possible two rows will be inserted for some sales transactions, and none for others.

```
INSERT ALL
WHEN promotion_id IN (SELECT promo_id FROM promotions) THEN
   INTO sales VALUES (product_id, customer_id, today, 3, promotion_id,
                      quantity_per_day, amount_per_day)
   INTO costs VALUES (product_id, today, promotion_id, 3,
                      product_cost, product_price)
WHEN num_of_orders > 1 THEN
   INTO cum_sales_activity VALUES (today, product_id, customer_id,
     promotion_id, quantity_per_day, amount_per_day, num_of_orders)
SELECT TRUNC(s.sales_date) AS today, s.product_id, s.customer_id,
   s.promotion_id, SUM(s.amount) AS amount_per_day, SUM(s.quantity)
   quantity_per_day, COUNT(*) num_of_orders, p.prod_min_price*0.8
   AS product_cost, p.prod_list_price AS product_price
FROM sales_activity_direct s, products p
WHERE s.product_id = p.prod_id
AND TRUNC(sales_date) = TRUNC(SYSDATE)
GROUP BY TRUNC(sales_date), s.product_id, s.customer_id,
 s.promotion_id, p.prod_min_price*0.8, p.prod_list_price;
```

### Example 19-4 Conditional FIRST Insert

The following statement inserts into an appropriate shipping manifest according to the total quantity and the weight of a product order. An exception is made for high value orders, which are also sent by express, unless their weight classification is too high. All incorrect orders, in this simple example represented as orders without a quantity, are stored in a separate table. It assumes the existence of appropriate tables `large_freight_shipping`, `express_shipping`, `default_shipping`, and `incorrect_sales_order`.

```
INSERT FIRST WHEN (sum_quantity_sold > 10 AND prod_weight_class < 5) AND
sum_quantity_sold >=1) OR (sum_quantity_sold > 5 AND prod_weight_class > 5) THEN
  INTO large_freight_shipping VALUES
      (time_id, cust_id, prod_id, prod_weight_class, sum_quantity_sold)
  WHEN sum_amount_sold > 1000 AND sum_quantity_sold >=1 THEN
  INTO express_shipping VALUES
      (time_id, cust_id, prod_id, prod_weight_class,
       sum_amount_sold, sum_quantity_sold)
WHEN (sum_quantity_sold >=1) THEN INTO default_shipping VALUES
      (time_id, cust_id, prod_id, sum_quantity_sold)
ELSE INTO incorrect_sales_order VALUES (time_id, cust_id, prod_id)
SELECT s.time_id, s.cust_id, s.prod_id, p.prod_weight_class,
       SUM(amount_sold) AS sum_amount_sold,
       SUM(quantity_sold) AS sum_quantity_sold
FROM sales s, products p
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
WHERE s.prod_id = p.prod_id AND s.time_id = TRUNC(SYSDATE)
GROUP BY s.time_id, s.cust_id, s.prod_id, p.prod_weight_class;
```

**Example 19-5 Mixed Conditional and Unconditional Insert**

The following example inserts new customers into the `customers` table and stores all new customers with `cust_credit_limit` higher then 4500 in an additional, separate table for further promotions.

```
INSERT FIRST WHEN cust_credit_limit >= 4500 THEN INTO customers
  INTO customers_special VALUES (cust_id, cust_credit_limit)
  ELSE INTO customers
SELECT * FROM customers_new;
```

> **See Also**
>
> Refreshing Materialized Views for more information regarding `MERGE` operations

## 19.3.2 Transforming Data Using PL/SQL

In a data warehouse environment, you can use procedural languages such as PL/SQL to implement complex transformations in the Oracle AI Database. Whereas CTAS operates on entire tables and emphasizes parallelism, PL/SQL provides a row-based approached and can accommodate very sophisticated transformation rules. For example, a PL/SQL procedure could open multiple cursors and read data from multiple source tables, combine this data using complex business rules, and finally insert the transformed data into one or more target table. It would be difficult or impossible to express the same sequence of operations using standard SQL statements.

Using a procedural language, a specific transformation (or number of transformation steps) within a complex ETL processing can be encapsulated, reading data from an intermediate staging area and generating a new table object as output. A previously generated transformation input table and a subsequent transformation will consume the table generated by this specific transformation. Alternatively, these encapsulated transformation steps within the complete ETL process can be integrated seamlessly, thus streaming sets of rows between each other without the necessity of intermediate staging. You can use table functions to implement such behavior.

## 19.3.3 Transforming Data Using Table Functions

Table functions provide the support for pipelined and parallel execution of transformations implemented in PL/SQL, C, or Java. Scenarios as mentioned earlier can be done without requiring the use of intermediate staging tables, which interrupt the data flow through various transformations steps. Detailed information about table functions is provided in "What is a Table Function?".

### 19.3.3.1 What is a Table Function?

A table function is defined as a function that can produce a set of rows as output. Additionally, table functions can take a set of rows as input. Prior to Oracle9*i*, PL/SQL functions:

- Could not take cursors as input.
- Could not be parallelized or pipelined.


Copyright © 2001, 2025, Oracle and/or its affiliates.

Now, functions are not limited in these ways. Table functions extend database functionality by allowing:

- Multiple rows to be returned from a function.
- Results of SQL subqueries (that select multiple rows) to be passed directly to functions.
- Functions take cursors as input.
- Functions can be parallelized.
- Returning result sets incrementally for further processing as soon as they are created. This is called incremental pipelining

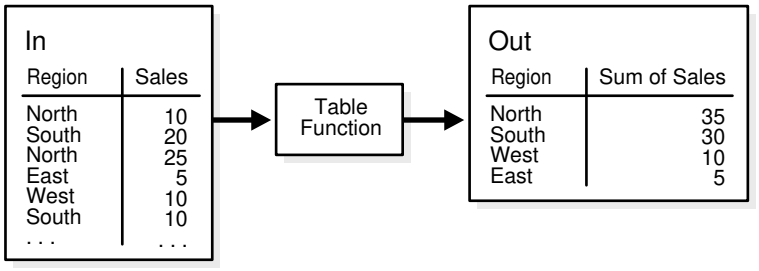
Table functions can be defined in PL/SQL using a native PL/SQL interface, or in Java or C using the Oracle Data Cartridge Interface (ODCI).

**See Also**

- *Oracle Database PL/SQL Language Reference* for further information
- *Oracle Database Data Cartridge Developer's Guide* for further information

Figure 19-3 illustrates a typical aggregation where you input a set of rows and output a set of rows, in that case, after performing a `SUM` operation.

**Figure 19-3 Table Function Example**




The pseudocode for this operation would be similar to:

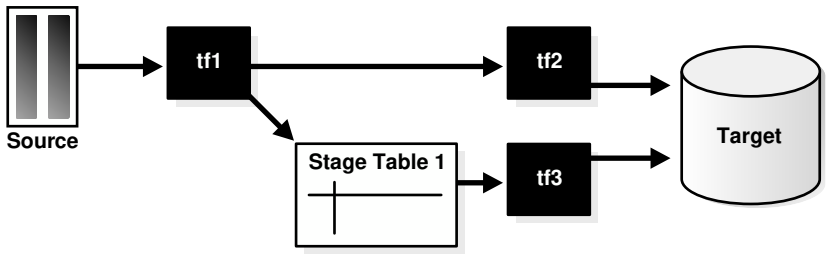
```
INSERT INTO Out SELECT * FROM ("Table Function"(SELECT * FROM In));
```

The table function takes the result of the `SELECT` on `In` as input and delivers a set of records in a different format as output for a direct insertion into `Out`.

Additionally, a table function can fan out data within the scope of an atomic transaction. This can be used for many occasions like an efficient logging mechanism or a fan out for other independent transformations. In such a scenario, a single staging table is needed.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Figure 19-4 Pipelined Parallel Transformation with Fanout**




The pseudocode for this would be similar to:

```
INSERT INTO target SELECT * FROM (tf2(SELECT *
FROM (tf1(SELECT * FROM source))));
```

This inserts into `target` and, as part of `tf1`, into `Stage Table 1` within the scope of an atomic transaction.

```
INSERT INTO target SELECT * FROM tf3(SELT * FROM stage_table1);
```

> **See Also**
>
> - *Oracle Database PL/SQL Language Reference* for details about table functions
> - *Oracle Database Data Cartridge Developer's Guide* for details about tables functions implemented in languages other than PL/SQL

### Objects to Create Before Running Table Function Examples

The following examples demonstrate the fundamentals of table functions, without the usage of complex business rules implemented inside those functions. They are chosen for demonstration purposes only, and are all implemented in PL/SQL.

Table functions return sets of records and can take cursors as input. Besides the `sh` sample schema, you have to set up the following database objects before using the examples:

```
CREATE TYPE product_t AS OBJECT (
      prod_id                    NUMBER(6)
    , prod_name                  VARCHAR2(50)
    , prod_desc                  VARCHAR2(4000)
    , prod_subcategory           VARCHAR2(50)
    , prod_subcategory_desc      VARCHAR2(2000)
    , prod_category              VARCHAR2(50)
    , prod_category_desc         VARCHAR2(2000)
    , prod_weight_class          NUMBER(2)
    , prod_unit_of_measure       VARCHAR2(20)
    , prod_pack_size             VARCHAR2(30)
    , supplier_id                NUMBER(6)
    , prod_status                VARCHAR2(20)
    , prod_list_price            NUMBER(8,2)
    , prod_min_price             NUMBER(8,2)
);
/
CREATE TYPE product_t_table AS TABLE OF product_t;
/
COMMIT;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
CREATE OR REPLACE PACKAGE cursor_PKG AS
  TYPE product_t_rec IS RECORD (
      prod_id                   NUMBER(6)
    , prod_name                 VARCHAR2(50)
    , prod_desc                 VARCHAR2(4000)
    , prod_subcategory          VARCHAR2(50)
    , prod_subcategory_desc     VARCHAR2(2000)
    , prod_category             VARCHAR2(50)
    , prod_category_desc        VARCHAR2(2000)
    , prod_weight_class         NUMBER(2)
    , prod_unit_of_measure      VARCHAR2(20)
    , prod_pack_size            VARCHAR2(30)
    , supplier_id               NUMBER(6)
    , prod_status               VARCHAR2(20)
    , prod_list_price           NUMBER(8,2)
    , prod_min_price            NUMBER(8,2));
  TYPE strong_refcur_t IS REF CURSOR RETURN product_t_rec;
  TYPE refcur_t IS REF CURSOR;
END;
/

REM artificial help table, used later
CREATE TABLE obsolete_products_errors (prod_id NUMBER, msg VARCHAR2(2000));
```

### Example 19-6 Table Functions Example: Basic Example

This example demonstrates a simple filtering; it shows all obsolete products except the `prod_category` Electronics. The table function returns the result set as a set of records and uses a weakly typed `REF CURSOR` as input.

```
CREATE OR REPLACE FUNCTION obsolete_products(cur cursor_pkg.refcur_t)
RETURN product_t_table
IS
    prod_id                   NUMBER(6);
    prod_name                 VARCHAR2(50);
    prod_desc                 VARCHAR2(4000);
    prod_subcategory          VARCHAR2(50);
    prod_subcategory_desc     VARCHAR2(2000);
    prod_category             VARCHAR2(50);
    prod_category_desc        VARCHAR2(2000);
    prod_weight_class         NUMBER(2);
    prod_unit_of_measure      VARCHAR2(20);
    prod_pack_size            VARCHAR2(30);
    supplier_id               NUMBER(6);
    prod_status               VARCHAR2(20);
    prod_list_price           NUMBER(8,2);
    prod_min_price            NUMBER(8,2);
    sales NUMBER:=0;
    objset product_t_table := product_t_table();
    i NUMBER := 0;
BEGIN
   LOOP
     -- Fetch from cursor variable
     FETCH cur INTO prod_id, prod_name, prod_desc, prod_subcategory,
    prod_subcategory_desc, prod_category, prod_category_desc, prod_weight_class,
    prod_unit_of_measure, prod_pack_size, supplier_id, prod_status,
    prod_list_price, prod_min_price;
     EXIT WHEN cur%NOTFOUND; -- exit when last row is fetched
     -- Category Electronics is not meant to be obsolete and will be suppressed
     IF prod_status='obsolete' AND prod_category != 'Electronics' THEN
     -- append to collection
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
     i:=i+1;
     objset.extend;
     objset(i):=product_t( prod_id, prod_name, prod_desc, prod_subcategory,
     prod_subcategory_desc, prod_category, prod_category_desc,
     prod_weight_class, prod_unit_of_measure, prod_pack_size, supplier_id,
     prod_status, prod_list_price, prod_min_price);
     END IF;
   END LOOP;
   CLOSE cur;
   RETURN objset;
END;
/
```

You can use the table function in a SQL statement to show the results. Here you use additional SQL functionality for the output:

```
SELECT DISTINCT UPPER(prod_category), prod_status
FROM TABLE(obsolete_products(
   CURSOR(SELECT prod_id, prod_name, prod_desc, prod_subcategory,
   prod_subcategory_desc, prod_category, prod_category_desc, prod_weight_class,
   prod_unit_of_measure, prod_pack_size,
   supplier_id, prod_status, prod_list_price, prod_min_price
          FROM products)));
```

### Example 19-7 Table Functions Example: Filtering Using REF CURSOR

This example implements the same filtering as Example 19-6. The main differences between the two are:

- This example uses a strong typed `REF CURSOR` as input and can be parallelized based on the objects of the strong typed cursor, as shown in one of the following examples.
- The table function returns the result set incrementally as soon as records are created.

```
CREATE OR REPLACE FUNCTION
  obsolete_products_pipe(cur cursor_pkg.strong_refcur_t) RETURN product_t_table
PIPELINED
PARALLEL_ENABLE (PARTITION cur BY ANY) IS
    prod_id                 NUMBER(6);
    prod_name               VARCHAR2(50);
    prod_desc               VARCHAR2(4000);
    prod_subcategory        VARCHAR2(50);
    prod_subcategory_desc   VARCHAR2(2000);
    prod_category           VARCHAR2(50);
    prod_category_desc      VARCHAR2(2000);
    prod_weight_class       NUMBER(2);
    prod_unit_of_measure   VARCHAR2(20);
    prod_pack_size         VARCHAR2(30);
    supplier_id            NUMBER(6);
    prod_status            VARCHAR2(20);
    prod_list_price        NUMBER(8,2);
    prod_min_price         NUMBER(8,2);
    sales NUMBER:=0;
BEGIN
 LOOP
     -- Fetch from cursor variable
     FETCH cur INTO prod_id, prod_name, prod_desc, prod_subcategory,
     prod_subcategory_desc, prod_category, prod_category_desc,
     prod_weight_class, prod_unit_of_measure, prod_pack_size, supplier_id,
     prod_status, prod_list_price, prod_min_price;
     EXIT WHEN cur%NOTFOUND; -- exit when last row is fetched
     IF prod_status='obsolete' AND prod_category !='Electronics' THEN
       PIPE ROW (product_t( prod_id, prod_name, prod_desc, prod_subcategory,
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
 prod_subcategory_desc, prod_category, prod_category_desc, prod_weight_class,
 prod_unit_of_measure, prod_pack_size, supplier_id, prod_status,
 prod_list_price, prod_min_price));
     END IF;
   END LOOP;
   CLOSE cur;
   RETURN;
END;
/
```

You can use the table function as follows:

```
SELECT DISTINCT prod_category,
               DECODE(prod_status,'obsolete','NO LONGER AVAILABLE','N/A')
FROM TABLE(obsolete_products_pipe(
  CURSOR(SELECT prod_id, prod_name, prod_desc, prod_subcategory,
         prod_subcategory_desc, prod_category, prod_category_desc,
         prod_weight_class, prod_unit_of_measure, prod_pack_size,
         supplier_id, prod_status, prod_list_price, prod_min_price
         FROM products)));
```

You now change the degree of parallelism for the input table products and issue the same statement again:

```
ALTER TABLE products PARALLEL 4;
```

The session statistics show that the statement has been parallelized:

```
SELECT * FROM V$PQ_SESSTAT WHERE statistic='Queries Parallelized';

STATISTIC              LAST_QUERY  SESSION_TOTAL
--------------------   ----------  -------------
Queries Parallelized            1              3

1 row selected.
```

### Example 19-8 Table Functions Example: Fanning Out Results into Persistent Tables

Table functions are also capable to fanout results into persistent table structures. In this example, the function filters all obsolete products except those of a specific `prod_category` (default Electronics), which was set to status `obsolete` by error. The result set of the table function consists of all other obsolete product categories. The detected wrong `prod_id` IDs are stored in a separate table structure `obsolete_products_error`. Note that if a table function is part of an autonomous transaction, it must `COMMIT` or `ROLLBACK` before each `PIPE ROW` statement to avoid an error in the callings subprogram. Its result set consists of all other obsolete product categories. It furthermore demonstrates how normal variables can be used in conjunction with table functions:

```
CREATE OR REPLACE FUNCTION obsolete_products_dml(cur cursor_pkg.strong_refcur_t,
 prod_cat varchar2 DEFAULT 'Electronics') RETURN product_t_table
PIPELINED
PARALLEL_ENABLE (PARTITION cur BY ANY) IS
    PRAGMA AUTONOMOUS_TRANSACTION;
    prod_id                   NUMBER(6);
    prod_name                 VARCHAR2(50);
    prod_desc                 VARCHAR2(4000);
    prod_subcategory          VARCHAR2(50);
    prod_subcategory_desc     VARCHAR2(2000);
    prod_category             VARCHAR2(50);
    prod_category_desc        VARCHAR2(2000);
    prod_weight_class         NUMBER(2);
    prod_unit_of_measure      VARCHAR2(20);
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
    prod_pack_size            VARCHAR2(30);
    supplier_id               NUMBER(6);
    prod_status               VARCHAR2(20);
    prod_list_price           NUMBER(8,2);
    prod_min_price            NUMBER(8,2);
    sales                     NUMBER:=0;
BEGIN
   LOOP
     -- Fetch from cursor variable
     FETCH cur INTO prod_id, prod_name, prod_desc, prod_subcategory,
  prod_subcategory_desc, prod_category, prod_category_desc, prod_weight_class,
  prod_unit_of_measure, prod_pack_size, supplier_id, prod_status,
     prod_list_price, prod_min_price;
     EXIT WHEN cur%NOTFOUND; -- exit when last row is fetched
     IF prod_status='obsolete' THEN
       IF prod_category=prod_cat THEN
          INSERT INTO obsolete_products_errors VALUES
          (prod_id, 'correction: category '||UPPER(prod_cat)||' still
   available');
          COMMIT;
       ELSE
       PIPE ROW (product_t( prod_id, prod_name, prod_desc, prod_subcategory,
 prod_subcategory_desc, prod_category, prod_category_desc, prod_weight_class,
 prod_unit_of_measure, prod_pack_size, supplier_id, prod_status,
 prod_list_price, prod_min_price));
       END IF;
     END IF;
   END LOOP;
   CLOSE cur;
   RETURN;
END;
/
```

The following query shows all obsolete product groups except the `prod_category` Electronics, which was wrongly set to status `obsolete`:

```
SELECT DISTINCT prod_category, prod_status FROM TABLE(obsolete_products_dml(
CURSOR(SELECT prod_id, prod_name, prod_desc, prod_subcategory,
  prod_subcategory_desc, prod_category, prod_category_desc, prod_weight_class,
  prod_unit_of_measure, prod_pack_size, supplier_id, prod_status,
  prod_list_price, prod_min_price
FROM products)));
```

As you can see, there are some products of the `prod_category` Electronics that were obsoleted by accident:

```
SELECT DISTINCT msg FROM obsolete_products_errors;
```

Taking advantage of the second input variable, you can specify a different product group than Electronics to be considered:

```
SELECT DISTINCT prod_category, prod_status
FROM TABLE(obsolete_products_dml(
CURSOR(SELECT prod_id, prod_name, prod_desc, prod_subcategory,
  prod_subcategory_desc, prod_category, prod_category_desc, prod_weight_class,
  prod_unit_of_measure, prod_pack_size, supplier_id, prod_status,
  prod_list_price, prod_min_price
FROM products),'Photo'));
```

Because table functions can be used like a normal table, they can be nested, as shown in the following:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
SELECT DISTINCT prod_category, prod_status
FROM TABLE(obsolete_products_dml(CURSOR(SELECT *
FROM TABLE(obsolete_products_pipe(CURSOR(SELECT prod_id, prod_name, prod_desc,
 prod_subcategory, prod_subcategory_desc, prod_category, prod_category_desc,
 prod_weight_class, prod_unit_of_measure, prod_pack_size, supplier_id,
 prod_status, prod_list_price, prod_min_price
FROM products))))));
```

The biggest advantage of Oracle AI Database's ETL is its toolkit functionality, where you can combine any of the latter discussed functionality to improve and speed up your ETL processing. For example, you can take an external table as input, join it with an existing table and use it as input for a parallelized table function to process complex business logic. This table function can be used as input source for a `MERGE` operation, thus streaming the new information for the data warehouse, provided in a flat file within one single statement through the complete ETL process.

# 19.4 Error Logging and Handling Mechanisms

Having data that is not clean is very common when loading and transforming data, especially when dealing with data coming from a variety of sources, including external ones. If this dirty data causes you to terminate a long-running load or transformation operation, a lot of time and resources is wasted.

The following topics discuss the two main causes of errors and how to address them:

- Business Rule Violations
- Data Rule Violations (Data Errors)

## 19.4.1 Business Rule Violations

Data that is logically not clean violates business rules that are known prior to any data consumption. Most of the time, handling these kind of errors will be incorporated into the loading or transformation process. However, in situations where the error identification for all records would become too expensive and the business rule can be enforced as a data rule violation, for example, testing hundreds of columns to see if they are `NOT NULL`, programmers often choose to handle even known possible logical error cases more generically. An example of this is shown in "Data Error Scenarios".

Incorporating logical rules can be as easy as applying filter conditions on the data input stream or as complex as feeding the dirty data into a different transformation workflow. Some examples are as follows:

- Filtering of logical data errors using SQL. Data that does not adhere to certain conditions is filtered out prior to being processed.
- Identifying and separating logical data errors. In simple cases, this can be accomplished using SQL, as shown in #unique_610/unique_610_Connect_42_i1006454, or in more complex cases in a procedural approach, as shown in Example 19-6.

## 19.4.2 Data Rule Violations (Data Errors)

Unlike logical errors, data rule violations are not usually anticipated by the load or transformation process. Such unexpected data rule violations (also known as data errors) that are not handled from an operation cause the operation to fail. Data rule violations are error conditions that happen inside the database and cause a statement to fail. Examples of this are data type conversion errors or constraint violations.


Copyright © 2001, 2025, Oracle and/or its affiliates.

In the past, SQL did not offer a way to handle data errors on a row level as part of its bulk processing. The only way to handle data errors inside the database was to use PL/SQL. Now, however, you can log data errors into a special error table while the DML operation continues. You can also handle data conversion errors using SQL functions.

The following sections briefly discuss the various exception handling strategies:

- Handling Data Errors with SQL
- Handling Data Errors in PL/SQL
- Handling Data Errors with an Error Logging Table

## 19.4.2.1 Handling Data Errors with SQL

External data that is used during the data transformation process may sometimes be inaccurate thereby causing data conversion errors. Certain SQL functions can be used to handle data conversion errors.

The `COMPATIBLE` parameter must be set to 12.2 to use SQL functions that handle data conversion errors.

The following strategies are available to handle data conversion errors with SQL functions:

- Explicit filtering of either valid or invalid data

  The `VALIDATE_CONVERSION` function identifies problem data that cannot be converted to the required data type. It returns 1 if a given expression can be converted to the specified data type, else it returns 0.

- Error handling within SQL data type conversion functions

  The `CAST`, `TO_NUMBER`, `TO_BINARY_FLOAT`, `TO_BINARY_DOUBLE`, `TO_DATE`, `TO_TIMESTAMP`, `TO_TIMESTAMP_TZ`, `TO_DSINTERVAL`, and `TO_YMINTERVAL` functions can return a user-specified value, instead of an error, when data type conversion errors occur. This reduces failures during an ETL process.

  The user-specified value is returned only if an error occurs while converting the expression, not when evaluating the expression. The `CAST` function also allows format strings and NLS parameter strings as arguments for certain data types.

**Example 19-9 Using VALIDATE_CONVERSION and CAST to Handle Data Conversion Errors**

Assume that data is loaded into the `PRODUCTS` table from the `TMP_PRODUCTS` table. The number and names of columns in both tables are the same, but the data type of the `prod_id` column is different. The `prod_id` column in the `PRODUCTS` table is of data type `NUMBER`. Although the data in the `prod_id` column in the `TMP_PRODUCTS` table is numeric, its data type is `VARCHAR2`. While loading data into the `PRODUCTS` table, you can handle data type conversion errors on the `prod_id` column by either filtering out the rows containing incorrect `prod_id` values or assigning a default value for `prod_id` values that cannot be converted to `NUMBER`.

The following command loads data from the `TMP_PRODUCTS` table into `PRODUCTS` table. Only rows where `tmp_products.prod_id` can be successfully converted into a numeric value are inserted.

```
INSERT INTO PRODUCTS
   (SELECT prod_id, prod_name, prod_desc, prod_category_id,
prod_category_name,
       prod_category_desc, prod_list_price
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
    FROM tmp_products
    WHERE VALIDATE_CONVERSION(prod_id AS NUMBER)=1);
```

You can use the CAST function to handle prod_id values that cause data type conversion errors. The following INSERT command loads data from the TMP_PRODUCTS table into the PRODUCTS table. The CAST function used with prod_id ensures that the default value of 0 is assigned to prod_id when a data type conversion error occurs. This ensures that the load operation does not fail because of data type conversion errors.

```
INSERT INTO PRODUCTS
   (SELECT CAST(prod_id AS NUMBER DEFAULT 0 ON CONVERSION ERROR), prod_name,
       prod_desc, prod_category_id, prod_category_name, prod_category_desc,
       prod_list_price
    FROM tmp_products);
```

**See Also**

*Oracle AI Database SQL Language Reference* for more information about the CAST and VALIDATE_CONVERSION functions and their supported data types

## 19.4.2.2 Handling Data Errors in PL/SQL

The following statement is an example of how error handling can be done using PL/SQL. Note that you have to use procedural record-level processing to catch any errors. This statement is a rough equivalent of the statement discussed in "Handling Data Errors with an Error Logging Table".

```
DECLARE
errm number default 0;
BEGIN
FOR crec IN (SELECT product_id, customer_id, TRUNC(sales_date) sd,
                    promotion_id, quantity, amount
             FROM sales_activity_direct) loop

BEGIN
 INSERT INTO sales VALUES (crec.product_id, crec.customer_id,
                           crec.sd, 3, crec.promotion_id,
                           crec.quantity, crec.amount);
exception
WHEN others then
 errm := sqlerrm;
 INSERT INTO sales_activity_error
        VALUES (errm, crec.product_id, crec.customer_id, crec.sd,
                crec.promotion_id, crec.quantity, crec.amount);
END;
END loop;
END;
/
```

## 19.4.2.3 Handling Data Errors with an Error Logging Table

DML error logging extends existing DML functionality by enabling you to specify the name of an error logging table into which Oracle AI Database should record errors encountered during


Copyright © 2001, 2025, Oracle and/or its affiliates.

DML operations. This enables you to complete the DML operation in spite of any errors, and to take corrective action on the erroneous rows at a later time.

This DML error logging table consists of several mandatory control columns and a set of user-defined columns that represent either all or a subset of the columns of the target table of the DML operation using a data type that is capable of storing potential errors for the target column. For example, you need a `VARCHAR2` data type in the error logging table to store `TO_NUM` data type conversion errors for a `NUMBER` column in the target table. You should use the `DBMS_ERRLOG` package to create the DML error logging tables. See the *Oracle Database PL/SQL Packages and Types Reference* for more information about this package and the structure of the logging table.

The column name mapping between the DML target table and an error logging table determines which columns besides the control columns is logged for a DML operation.

The following statement illustrates how to enhance the example in "Transforming Data Using SQL" with DML error logging:

```
INSERT /*+ APPEND PARALLEL */
INTO sales SELECT product_id, customer_id, TRUNC(sales_date), 3,
   promotion_id, quantity, amount
FROM sales_activity_direct
LOG ERRORS INTO sales_activity_errors('load_20040802')
REJECT LIMIT UNLIMITED
```

All data errors are logged into table `sales_activity_errors`, identified by the optional tag `load_20040802`. The `INSERT` statement succeeds even in the presence of data errors. Note that you have to create the DML error logging table prior to using this statement.

If `REJECT LIMIT X` had been specified, the statement would have failed with the error message of `error X=1`. The error message can be different for different reject limits. In the case of a failing statement, only the DML statement is rolled back, not the insertion into the DML error logging table. The error logging table will contain X+1 rows.

A DML error logging table can be in a different schema than the executing user, but you must fully specify the table name in that case. Optionally, the name of the DML error logging table can be omitted; Oracle then assumes a default name for the table as generated by the `DBMS_ERRLOG` package.

Oracle AI Database logs the following errors during DML operations:

- Column values that are too large.
- Constraint violations (`NOT NULL`, unique, referential, and check constraints).
- Errors raised during trigger execution.
- Errors resulting from type conversion between a column in a subquery and the corresponding column of the table.
- Partition mapping errors.

The following conditions cause the statement to fail and roll back without invoking the error logging capability:

- Violated deferred constraints.
- Out of space errors.
- Any direct-path `INSERT` operation (`INSERT` or `MERGE`) that raises a unique constraint or index violation.
- Any `UPDATE` operation (`UPDATE` or `MERGE`) that raises a unique constraint or index violation.


Copyright © 2001, 2025, Oracle and/or its affiliates.

In addition, you cannot track errors in the error logging table for `LONG`, `LOB`, or object type columns. See *Oracle Database SQL Language Reference* for more information on restrictions when using error logging.

DML error logging can be applied to any kind of DML operation. Several examples are discussed in the following section.

Note that SQL*Loader as an external load utility offers the functionality of logging data errors as well, but lacks the advantage of the integrated ETL processing inside the database.

# 19.5 Loading and Transformation Scenarios

The following sections offer examples of typical loading and transformation tasks:

- Key Lookup Scenario
- Business Rule Violation Scenario
- Data Error Scenarios
- Pivoting Scenarios

## 19.5.1 Key Lookup Scenario

A typical transformation is the key lookup. For example, suppose that sales transaction data has been loaded into a retail data warehouse. Although the data warehouse's `sales` table contains a `product_id` column, the sales transaction data extracted from the source system contains Uniform Price Codes (UPC) instead of product IDs. Therefore, it is necessary to transform the UPC codes into product IDs before the new sales transaction data can be inserted into the `sales` table.

In order to execute this transformation, a lookup table must relate the `product_id` values to the UPC codes. This table might be the `product` dimension table, or perhaps another table in the data warehouse that has been created specifically to support this transformation. For this example, you assume that there is a table named `product`, which has a `product_id` and an `upc_code` column.

This data substitution transformation can be implemented using the following CTAS statement:

```
CREATE TABLE temp_sales_step2 NOLOGGING PARALLEL AS SELECT sales_transaction_id,
  product.product_id sales_product_id, sales_customer_id, sales_time_id,
  sales_channel_id, sales_quantity_sold, sales_dollar_amount
FROM  temp_sales_step1, product
WHERE temp_sales_step1.upc_code = product.upc_code;
```

This CTAS statement converts each valid UPC code to a valid `product_id` value. If the ETL process has guaranteed that each UPC code is valid, then this statement alone may be sufficient to implement the entire transformation.

## 19.5.2 Business Rule Violation Scenario

In the preceding example, if you must also handle new sales data that does not have valid UPC codes (a logical data error), you can use an additional CTAS statement to identify the invalid rows:

```
CREATE TABLE temp_sales_step1_invalid NOLOGGING PARALLEL AS
SELECT * FROM temp_sales_step1 s
WHERE NOT EXISTS (SELECT 1 FROM product p WHERE p.upc_code=s.upc_code);
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

This invalid data is now stored in a separate table, `temp_sales_step1_invalid`, and can be handled separately by the ETL process.

Another way to handle invalid data is to modify the original CTAS to use an outer join, as in the following statement:

```
CREATE TABLE temp_sales_step2 NOLOGGING PARALLEL AS
SELECT sales_transaction_id, product.product_id sales_product_id,
   sales_customer_id, sales_time_id, sales_channel_id, sales_quantity_sold,
   sales_dollar_amount
FROM  temp_sales_step1, product
WHERE temp_sales_step1.upc_code = product.upc_code (+);
```

Using this outer join, the sales transactions that originally contained invalidated UPC codes are assigned a `product_id` of `NULL`. These transactions can be handled later. Alternatively, you could use a multi-table insert, separating the values with a `product_id` of `NULL` into a separate table; this might be a beneficial approach when the expected error count is relatively small compared to the total data volume. You do not have to touch the large target table but only a small one for a subsequent processing.

```
INSERT /*+ APPEND PARALLEL */ FIRST
WHEN sales_product_id IS NOT NULL THEN
   INTO temp_sales_step2
   VALUES (sales_transaction_id, sales_product_id,
   sales_customer_id, sales_time_id, sales_channel_id,
   sales_quantity_sold, sales_dollar_amount)
ELSE
   INTO temp_sales_step1_invalid
   VALUES (sales_transaction_id, sales_product_id,
   sales_customer_id, sales_time_id, sales_channel_id,
   sales_quantity_sold, sales_dollar_amount)
SELECT sales_transaction_id, product.product_id sales_product_id,
   sales_customer_id, sales_time_id, sales_channel_id,
   sales_quantity_sold, sales_dollar_amount
FROM temp_sales_step1, product
WHERE temp_sales_step1.upc_code = product.upc_code (+);
```

Note that for this solution, the empty tables `temp_sales_step2` and `temp_sales_step1_invalid` must already exist.

Additional approaches to handling invalid UPC codes exist. Some data warehouses may choose to insert null-valued `product_id` values into their `sales` table, while others may not allow any new data from the entire batch to be inserted into the `sales` table until all invalid UPC codes have been addressed. The correct approach is determined by the business requirements of the data warehouse. Irrespective of the specific requirements, exception handling can be addressed by the same basic SQL techniques as transformations.

## 19.5.3 Data Error Scenarios

If the quality of the data is unknown, the example discussed in Business Rule Violation Scenario could be enhanced to handle unexpected data errors, for example, data type conversion errors, as shown in the following:

```
INSERT /*+ APPEND PARALLEL */ FIRST
WHEN sales_product_id IS NOT NULL THEN
INTO temp_sales_step2
VALUES (sales_transaction_id, sales_product_id,
   sales_customer_id, sales_time_id, sales_channel_id,
   sales_quantity_sold, sales_dollar_amount)
LOG ERRORS INTO sales_step2_errors('load_20040804')
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
REJECT LIMIT UNLIMITED
ELSE
INTO temp_sales_step1_invalid
VALUES (sales_transaction_id, sales_product_id,
   sales_customer_id, sales_time_id, sales_channel_id,
   sales_quantity_sold, sales_dollar_amount)
LOG ERRORS INTO sales_step2_errors('load_20040804')
REJECT LIMIT UNLIMITED
SELECT sales_transaction_id, product.product_id sales_product_id,
   sales_customer_id, sales_time_id, sales_channel_id,
   sales_quantity_sold, sales_dollar_amount
FROM temp_sales_step1, product
WHERE temp_sales_step1.upc_code = product.upc_code (+);
```

This statement tracks the logical data error of not having a valid product UPC code in table `temp_sales_step1_invalid` and all other possible errors in a DML error logging table called `sales_step2_errors`. Note that an error logging table can be used for several DML operations.

An alternative to this approach would be to enforce the business rule of having a valid UPC code on the database level with a `NOT NULL` constraint. Using an outer join, all orders not having a valid UPC code would be mapped to a `NULL` value and then treated as data errors. This DML error logging capability is used to track these errors in the following statement:

```
INSERT /*+ APPEND PARALLEL */
INTO temp_sales_step2
VALUES (sales_transaction_id, sales_product_id,
   sales_customer_id, sales_time_id, sales_channel_id,
   sales_quantity_sold, sales_dollar_amount)
SELECT sales_transaction_id, product.product_id sales_product_id,
   sales_customer_id, sales_time_id, sales_channel_id,
   sales_quantity_sold, sales_dollar_amount
FROM temp_sales_step1, product
WHERE temp_sales_step1.upc_code = product.upc_code (+)
LOG ERRORS INTO sales_step2_errors('load_20040804')
REJECT LIMIT UNLIMITED;
```

The error logging table contains all records that would have caused the DML operation to fail. You can use its content to analyze and correct any error. The content in the error logging table is preserved for any DML operation, irrespective of the success of the DML operation itself. Let us assume the following SQL statement failed because the reject limit was reached:

```
SQL> INSERT /*+ APPEND NOLOGGING PARALLEL */ INTO sales_overall
2 SELECT * FROM sales_activity_direct
3 LOG ERRORS INTO err$_sales_overall ('load_test2')
4 REJECT LIMIT 10;
SELECT * FROM sales_activity_direct
*
ERROR at line 2:
ORA-01722: invalid number
```

The name of the error logging table, `err$_sales_overall`, is the default derived by using the `DBMS_ERRLOG` package. See *Oracle Database PL/SQL Packages and Types Reference* for more information.

The error message raised by Oracle occurs where the first after the error limit is reached. The next error (number 11) is the one that raised an error. The error message that is displayed is based on the error that exceeded the limit, so, for example, the ninth error could be different from the eleventh error.

The target table `sales_overall` will not show any records being entered (assumed that the table was empty before), but the error logging table will contain 11 rows (`REJECT LIMIT + 1`)


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
SQL> SELECT COUNT(*) FROM sales_overall;
COUNT(*)
----------
0

SQL> SELECT COUNT(*) FROM err$_sales_overall;
COUNT(*)
----------
11
```

A DML error logging table consists of several fixed control columns that are mandatory for every error logging table. Besides the Oracle error number, Oracle enforces storing the error message as well. In many cases, the error message provides additional information to analyze and resolve the root cause for the data error. The following SQL output of a DML error logging table shows this difference. Note that the second output contains the additional information for rows that were rejected due to `NOT NULL` violations.

```
SQL> SELECT DISTINCT ora_err_number$ FROM err$_sales_overall;

ORA_ERR_NUMBER$
---------------
           1400
           1722
           1830
           1847

SQL> SELECT DISTINCT ora_err_number$, ora_err_mesg$ FROM err$_sales_overall;

ORA_ERR_NUMBER$       ORA_ERR_MESG$
           1400       ORA-01400: cannot insert NULL into
                      ("SH"."SALES_OVERALL"."CUST_ID")
           1400       ORA-01400: cannot insert NULL into
                      ("SH"."SALES_OVERALL"."PROD_ID")
           1722       ORA-01722: invalid number
           1830       ORA-01830: date format picture ends before
                      converting entire input string
           1847       ORA-01847: day of month must be between 1 and last
                      day of month
```

> **See Also**
>
> *Oracle Database Administrator's Guide* for a detailed description of control columns.

## 19.5.4 Pivoting Scenarios

A data warehouse can receive data from many different sources. Some of these source systems may not be relational databases and may store data in very different formats from the data warehouse. For example, suppose that you receive a set of sales records from a nonrelational database having the form:

```
product_id, customer_id, weekly_start_date, sales_sun, sales_mon, sales_tue,
  sales_wed, sales_thu, sales_fri, sales_sat
```

The input table looks like the following:

```
SELECT * FROM sales_input_table;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
PRODUCT_ID CUSTOMER_ID WEEKLY_ST  SALES_SUN  SALES_MON  SALES_TUE  SALES_WED SALES_THU  SALES_FRI  SALES_SAT
---------- ----------- --------- ---------- ---------- ---------- ---------- ---------- ---------- ----------
       111         222 01-OCT-00        100        200        300        400        500        600        700
       222         333 08-OCT-00        200        300        400        500        600        700        800
       333         444 15-OCT-00        300        400        500        600        700        800        900
```

In your data warehouse, you would want to store the records in a more typical relational form in a fact table `sales` of the `sh` sample schema:

```
prod_id, cust_id, time_id, amount_sold
```

> **Note**
>
> A number of constraints on the `sales` table have been disabled for purposes of this example, because the example ignores a number of table columns for the sake of brevity.

Thus, you need to build a transformation such that each record in the input stream must be converted into seven records for the data warehouse's `sales` table. This operation is commonly referred to as pivoting, and Oracle AI Database offers several ways to do this.

The result of the previous example will resemble the following:

```
SELECT prod_id, cust_id, time_id, amount_sold FROM sales;

   PROD_ID    CUST_ID   TIME_ID   AMOUNT_SOLD
---------- ----------   --------- -----------
       111        222   01-OCT-00         100
       111        222   02-OCT-00         200
       111        222   03-OCT-00         300
       111        222   04-OCT-00         400
       111        222   05-OCT-00         500
       111        222   06-OCT-00         600
       111        222   07-OCT-00         700
       222        333   08-OCT-00         200
       222        333   09-OCT-00         300
       222        333   10-OCT-00         400
       222        333   11-OCT-00         500
       222        333   12-OCT-00         600
       222        333   13-OCT-00         700
       222        333   14-OCT-00         800
       333        444   15-OCT-00         300
       333        444   16-OCT-00         400
       333        444   17-OCT-00         500
       333        444   18-OCT-00         600
       333        444   19-OCT-00         700
       333        444   20-OCT-00         800
       333        444   21-OCT-00         900
```

**Example 19-10 Pivoting Example**

The following example uses the multitable insert syntax to insert into the demo table `sh.sales` some data from an input table with a different structure. The multitable `INSERT` statement looks like the following:

```
INSERT ALL INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date, sales_sun)
      INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date+1, sales_mon)
      INTO sales (prod_id, cust_id, time_id, amount_sold)
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
      VALUES (product_id, customer_id, weekly_start_date+2, sales_tue)
      INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date+3, sales_wed)
      INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date+4, sales_thu)
      INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date+5, sales_fri)
      INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date+6, sales_sat)
SELECT product_id, customer_id, weekly_start_date, sales_sun,
      sales_mon, sales_tue, sales_wed, sales_thu, sales_fri, sales_sat
FROM sales_input_table;
```

This statement only scans the source table once and then inserts the appropriate data for each day.

### See Also

- "Pivoting Operations" for more information regarding pivoting
- *Oracle Database SQL Language Reference* for `pivot_clause` syntax


Copyright © 2001, 2025, Oracle and/or its affiliates.

# Part IV
# Relational Analytics

This section deals with ways to improve your data warehouse's performance, and contains the following chapters:

- SQL for Analysis and Reporting
- SQL for Aggregation in Data Warehouses
- SQL for Pattern Matching
- SQL for Modeling
- Advanced Analytical SQL


Copyright © 2001, 2025, Oracle and/or its affiliates.

20

# SQL for Analysis and Reporting

The following topics provide information about analytical SQL features and techniques in Oracle. Although these topics are presented in terms of data warehousing, they are applicable to any activity needing analysis and reporting.

- Overview of SQL for Analysis and Reporting
- Ranking, Windowing, and Reporting Functions
- Advanced Aggregates for Analysis
- Pivoting Operations with examples such as Example 20-26
- Data Densification for Reporting
- Time Series Calculations on Densified Data
- Miscellaneous Analysis and Reporting Capabilities
- Limiting SQL Rows

## 20.1 Overview of SQL for Analysis and Reporting

Oracle AI Database provides a large family of analytic SQL functions. These analytic functions enable you to calculate:

- Rankings and percentiles
- Moving window calculations
- Lag/lead analysis
- First/last analysis
- Linear regression statistics

Ranking functions include cumulative distributions, percent rank, and N-tiles. Moving window calculations allow you to find moving and cumulative aggregations, such as sums and averages. Lag/lead analysis enables direct inter-row references so you can calculate period-to-period changes. First/last analysis enables you to find the first or last value in an ordered group.

Other SQL elements valuable for analysis and reporting include the `CASE` expression and partitioned outer join. `CASE` expressions provide if-then logic useful in many situations. Partitioned outer join is a variant of ANSI outer join syntax that allows users to selectively densify certain dimensions while keeping others sparse. This allows reporting tools to selectively densify dimensions, for example, the ones that appear in their cross-tabular reports while keeping others sparse.

To enhance performance, analytic functions can be parallelized: multiple processes can simultaneously execute all of these statements. These capabilities make calculations easier and more efficient, thereby enhancing database performance, scalability, and simplicity.

Analytic functions are classified as described in Table 20-1.

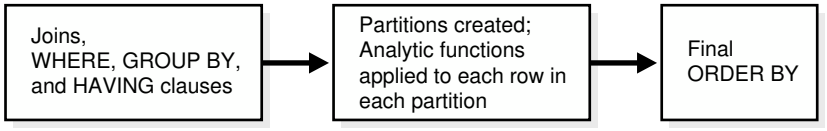

Copyright © 2001, 2025, Oracle and/or its affiliates.

**Table 20-1** ***Analytic Functions and Their Uses***

| Type | Used For |
|---|---|
| Ranking | Calculating ranks, percentiles, and n-tiles of the values in a result set. |
| Windowing | Calculating cumulative and moving aggregates. Works with these functions: `AVG`, `BIT_AND_AGG`, `BIT_OR_AGG`, `BIT_XOR_AGG`, `CHECKSUM`, `COUNT`, `FIRST_VALUE`, `KURTOSIS_POP`, `KURTOSIS_SAMP`, `LAST_VALUE`, `MAX`, `MIN`, `SKEWNESS_POP`, `SKEWNESS_SAMP`, `SUM`, `STDDEV`, and `VARIANCE`, and new statistical functions. Note that the `DISTINCT` keyword is not supported in windowing functions except for `MAX` and `MIN`. |
| Reporting | Calculating shares, for example, market share. Works with these functions: `SUM`, `AVG`, `MIN`, `MAX`, `COUNT` (with/without `DISTINCT`), `VARIANCE`, `STDDEV`, `RATIO_TO_REPORT`, `BIT_AND_AGG`, `BIT_OR_AGG`, `BIT_XOR_AGG`, `KURTOSIS_POP`, `KURTOSIS_SAMP`, `SKEWNESS_POP`, `SKEWNESS_SAMP`, and new statistical functions. Note that the `DISTINCT` keyword may be used in those reporting functions that support `DISTINCT` in aggregate mode. |
| `LAG/LEAD` | Finding a value in a row a specified number of rows from a current row. |
| `FIRST/LAST` | First or last value in an ordered group. |
| Linear Regression | Calculating linear regression and other statistics (slope, intercept, and so on). |
| Inverse Percentile | The value in a data set that corresponds to a specified percentile. |
| Hypothetical Rank and Distribution | The rank or percentile that a row would have if inserted into a specified data set. |

To perform these operations, the analytic functions add several new elements to SQL processing. These elements build on existing SQL to allow flexible and powerful calculation expressions. With just a few exceptions, the analytic functions have these additional elements. The processing flow is represented in Figure 20-1.

**Figure 20-1 Processing Order**




The essential concepts used in analytic functions are:

- Processing order

  Query processing using analytic functions takes place in three stages. First, all joins, `WHERE`, `GROUP BY` and `HAVING` clauses are performed. Second, the result set is made available to the analytic functions, and all their calculations take place. Third, if the query has an `ORDER BY` clause at its end, the `ORDER BY` is processed to allow for precise output ordering. The processing order is shown in Figure 20-1.

- Result set partitions

  The analytic functions allow users to divide query result sets into groups of rows called partitions. Note that the term **partitions** used with analytic functions is unrelated to the table partitions feature. Throughout this chapter, the term partitions refers to only the meaning related to analytic functions. Partitions are created after the groups defined with `GROUP BY` clauses, so they are available to any aggregate results such as sums and


Copyright © 2001, 2025, Oracle and/or its affiliates.

averages. Partition divisions may be based upon any desired columns or expressions. A query result set may be partitioned into just one partition holding all the rows, a few large partitions, or many small partitions holding just a few rows each.

- Window

  For each row in a partition, you can define a sliding window of data. This window determines the range of rows used to perform the calculations for the current row. Window sizes can be based on either a physical number of rows or a logical interval such as time. The window has a starting row and an ending row. Depending on its definition, the window may move at one or both ends. For instance, a window defined for a cumulative sum function would have its starting row fixed at the first row of its partition, and its ending row would slide from the starting point all the way to the last row of the partition. In contrast, a window defined for a moving average would have both its starting and end points slide so that they maintain a constant physical or logical range.

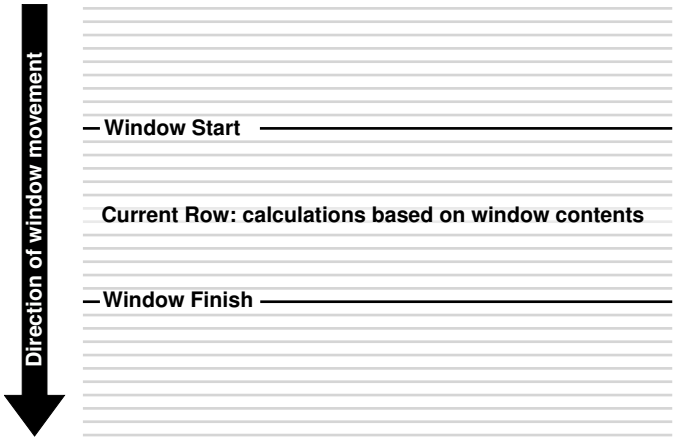
  A window can be set as large as all the rows in a partition or just a sliding window of one row within a partition. When a window is near a border, the function returns results for only the available rows, rather than warning you that the results are not what you want.

  When using window functions, the current row is included during calculations, so you should only specify (*n*-1) when you are dealing with *n* items.

- Current row

  Each calculation performed with an analytic function is based on a current row within a partition. The current row serves as the reference point determining the start and end of the window. For instance, a centered moving average calculation could be defined with a window that holds the current row, the six preceding rows, and the following six rows. This would create a sliding window of 13 rows, as shown in Figure 20-2.

**Figure 20-2 Sliding Window Example**




# 20.2 Ranking, Windowing, and Reporting Functions

This section illustrates the basic analytic functions for ranking, windowing, and reporting. It contains the following topics:

- Ranking Functions
- Windowing Functions
- Reporting Functions


Copyright © 2001, 2025, Oracle and/or its affiliates.

- LAG/LEAD Functions
- FIRST_VALUE_ LAST_VALUE_ and NTH_VALUE Functions

## 20.2.1 Ranking Functions

A ranking function computes the rank of a record compared to other records in the data set based on the values of a set of measures. The types of ranking function are:

- RANK and DENSE_RANK Functions
- Bottom N Ranking Functions
- CUME_DIST Function
- PERCENT_RANK Function
- NTILE Function
- ROW_NUMBER Function

### 20.2.1.1 RANK and DENSE_RANK Functions

The `RANK` and `DENSE_RANK` functions allow you to rank items in a group, for example, finding the top three products sold in California last year. There are two functions that perform ranking, as shown by the following syntax:

```
RANK ( ) OVER ( [query_partition_clause] order_by_clause )
DENSE_RANK ( ) OVER ( [query_partition_clause] order_by_clause )
```

The difference between `RANK` and `DENSE_RANK` is that `DENSE_RANK` leaves no gaps in ranking sequence when there are ties. That is, if you were ranking a competition using `DENSE_RANK` and had three people tie for second place, you would say that all three were in second place and that the next person came in third. The `RANK` function would also give three people in second place, but the next person would be in fifth place.

The following are some relevant points about `RANK`:

- Ascending is the default sort order, which you may want to change to descending.
- The expressions in the optional `PARTITION BY` clause divide the query result set into groups within which the `RANK` function operates. That is, `RANK` gets reset whenever the group changes. In effect, the value expressions of the `PARTITION BY` clause define the reset boundaries.
- If the `PARTITION BY` clause is missing, then ranks are computed over the entire query result set.
- The `ORDER BY` clause specifies the measures (`<value expression>`) on which ranking is done and defines the order in which rows are sorted in each group (or partition). Once the data is sorted within each partition, ranks are given to each row starting from 1.
- The `NULLS FIRST | NULLS LAST` clause indicates the position of `NULLs` in the ordered sequence, either first or last in the sequence. The order of the sequence would make `NULLs` compare either high or low with respect to non-`NULL` values. If the sequence were in ascending order, then `NULLS FIRST` implies that `NULLs` are smaller than all other non-`NULL` values and `NULLS LAST` implies they are larger than non-`NULL` values. It is the opposite for descending order. See the example in "Examples: Treatment of NULLs in Ranking Functions".
- If the `NULLS FIRST | NULLS LAST` clause is omitted, then the ordering of the null values depends on the `ASC` or `DESC` arguments. Null values are considered larger than any other


Copyright © 2001, 2025, Oracle and/or its affiliates.

values. If the ordering sequence is ASC, then nulls will appear last; nulls will appear first otherwise. Nulls are considered equal to other nulls and, therefore, the order in which nulls are presented is non-deterministic.

### 20.2.1.1.1 Ranking Order in RANK and DENSE_RANK Functions

The following example shows how the [ASC | DESC] option of RANK changes the ranking order.

**Example 20-1 Ranking Order**

```
SELECT channel_desc, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$,
   RANK() OVER (ORDER BY SUM(amount_sold)) AS default_rank,
   RANK() OVER (ORDER BY SUM(amount_sold) DESC NULLS LAST) AS custom_rank
FROM sales, products, customers, times, channels, countries
WHERE sales.prod_id=products.prod_id AND sales.cust_id=customers.cust_id
  AND customers.country_id = countries.country_id AND sales.time_id=times.time_id
  AND sales.channel_id=channels.channel_id
  AND times.calendar_month_desc IN ('2000-09', '2000-10')
  AND country_iso_code='US'
GROUP BY channel_desc;

CHANNEL_DESC         SALES$         DEFAULT_RANK CUSTOM_RANK
-------------------- -------------- ------------ -----------
Direct Sales              1,320,497            3           1
Partners                    800,871            2           2
Internet                    261,278            1           3
```

While the data in this result is ordered on the measure SALES$, in general, it is not guaranteed by the RANK function that the data will be sorted on the measures. If you want the data to be sorted on SALES$ in your result, you must specify it explicitly with an ORDER BY clause, at the end of the SELECT statement.

### 20.2.1.1.2 Ranking on Multiple Expressions

Ranking functions must resolve ties between values in the set. If the first expression cannot resolve ties, the second expression is used to resolve ties and so on. For example, here is a query ranking three of the sales channels over two months based on their dollar sales, breaking ties with the unit sales. (Note that the TRUNC function is used here only to create tie values for this query.)

**Example 20-2 Ranking On Multiple Expressions**

```
SELECT channel_desc, calendar_month_desc, TO_CHAR(TRUNC(SUM(amount_sold),-5),
  '9,999,999,999') SALES$, TO_CHAR(SUM(quantity_sold), '9,999,999,999')
  SALES_Count, RANK() OVER (ORDER BY TRUNC(SUM(amount_sold), -5)
  DESC, SUM(quantity_sold) DESC) AS col_rank
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id AND sales.cust_id=customers.cust_id
  AND sales.time_id=times.time_id AND sales.channel_id=channels.channel_id
  AND times.calendar_month_desc IN ('2000-09', '2000-10')
  AND channels.channel_desc<>'Tele Sales'
GROUP BY channel_desc, calendar_month_desc;

CHANNEL_DESC         CALENDAR SALES$         SALES_COUNT     COL_RANK
-------------------- -------- -------------- -------------- ---------
Direct Sales         2000-10       1,200,000         12,584         1
Direct Sales         2000-09       1,200,000         11,995         2
Partners             2000-10         600,000          7,508         3
Partners             2000-09         600,000          6,165         4
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
Internet            2000-09         200,000          1,887          5
Internet            2000-10         200,000          1,450          6
```

The sales_count column breaks the ties for three pairs of values.

If you only want to see the top five results for this query, you can add an ORDER BY COL_RANK FETCH FIRST 5 ROWS ONLY statement. See "Limiting SQL Rows" for further information.

### 20.2.1.1.3 Example: Difference Between RANK and DENSE_RANK

The difference between RANK and DENSE_RANK functions is illustrated in Example 20-3.

**Example 20-3 RANK and DENSE_RANK**

```
SELECT channel_desc, calendar_month_desc,
   TO_CHAR(TRUNC(SUM(amount_sold),-5), '9,999,999,999') SALES$,
      RANK() OVER (ORDER BY TRUNC(SUM(amount_sold),-5) DESC) AS RANK,
DENSE_RANK() OVER (ORDER BY TRUNC(SUM(amount_sold),-5) DESC) AS DENSE_RANK
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id
  AND sales.cust_id=customers.cust_id
  AND sales.time_id=times.time_id AND sales.channel_id=channels.channel_id
  AND times.calendar_month_desc IN ('2000-09', '2000-10')
  AND channels.channel_desc<>'Tele Sales'
GROUP BY channel_desc, calendar_month_desc;

CHANNEL_DESC         CALENDAR SALES$              RANK DENSE_RANK
-------------------- -------- -------------- --------- ----------
Direct Sales         2000-09       1,200,000         1          1
Direct Sales         2000-10       1,200,000         1          1
Partners             2000-09         600,000         3          2
Partners             2000-10         600,000         3          2
Internet             2000-09         200,000         5          3
Internet             2000-10         200,000         5          3
```

Note that, in the case of DENSE_RANK, the largest rank value gives the number of distinct values in the data set.

### 20.2.1.1.4 Ranking Within Groups: Example

The RANK function can be made to operate within groups, that is, the rank gets reset whenever the group changes. This is accomplished with the PARTITION BY clause. The group expressions in the PARTITION BY subclause divide the data set into groups within which RANK operates. For example, to rank products within each channel by their dollar sales, you could issue the following statement.

**Example 20-4 Per Group Ranking Example 1**

```
SELECT channel_desc, calendar_month_desc, TO_CHAR(SUM(amount_sold),
 '9,999,999,999') SALES$, RANK() OVER (PARTITION BY channel_desc
  ORDER BY SUM(amount_sold) DESC) AS RANK_BY_CHANNEL
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id AND sales.cust_id=customers.cust_id
  AND sales.time_id=times.time_id AND sales.channel_id=channels.channel_id
  AND times.calendar_month_desc IN ('2000-08', '2000-09', '2000-10', '2000-11')
  AND channels.channel_desc IN ('Direct Sales', 'Internet')
GROUP BY channel_desc, calendar_month_desc;

CHANNEL_DESC         CALENDAR SALES$         RANK_BY_CHANNEL
-------------------- -------- -------------- ---------------
Direct Sales         2000-08       1,236,104               1
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
Direct Sales         2000-10       1,225,584              2
Direct Sales         2000-09       1,217,808              3
Direct Sales         2000-11       1,115,239              4
Internet             2000-11         284,742              1
Internet             2000-10         239,236              2
Internet             2000-09         228,241              3
Internet             2000-08         215,107              4

8 rows selected.
```

A single query block can contain more than one ranking function, each partitioning the data into different groups (that is, reset on different boundaries). The groups can be mutually exclusive. The following query ranks products based on their dollar sales within each month (`rank_of_product_per_region`) and within each channel (`rank_of_product_total`).

**Example 20-5 Per Group Ranking Example 2**

```
SELECT channel_desc, calendar_month_desc, TO_CHAR(SUM(amount_sold),
  '9,999,999,999') SALES$, RANK() OVER (PARTITION BY calendar_month_desc
  ORDER BY SUM(amount_sold) DESC) AS RANK_WITHIN_MONTH, RANK() OVER (PARTITION
 BY channel_desc ORDER BY SUM(amount_sold) DESC) AS RANK_WITHIN_CHANNEL
FROM sales, products, customers, times, channels, countries
WHERE sales.prod_id=products.prod_id AND sales.cust_id=customers.cust_id
  AND customers.country_id = countries.country_id AND sales.time_id=times.time_id
  AND sales.channel_id=channels.channel_id
  AND times.calendar_month_desc IN ('2000-08', '2000-09', '2000-10', '2000-11')
  AND channels.channel_desc IN ('Direct Sales', 'Internet')
GROUP BY channel_desc, calendar_month_desc;

CHANNEL_DESC  CALENDAR    SALES$     RANK_WITHIN_MONTH  RANK_WITHIN_CHANNEL
------------- --------    ---------  -----------------  -------------------
Direct Sales   2000-08    1,236,104                  1                    1
Internet       2000-08      215,107                  2                    4
Direct Sales   2000-09    1,217,808                  1                    3
Internet       2000-09      228,241                  2                    3
Direct Sales   2000-10    1,225,584                  1                    2
Internet       2000-10      239,236                  2                    2
Direct Sales   2000-11    1,115,239                  1                    4
Internet       2000-11      284,742                  2                    1
```

## 20.2.1.1.5 Example: Per Cube and Rollup Group Ranking

Analytic functions, `RANK` for example, can be reset based on the groupings provided by a `CUBE`, `ROLLUP`, or `GROUPING SETS` operator. It is useful to assign ranks to the groups created by `CUBE`, `ROLLUP`, and `GROUPING SETS` queries. See SQL for Aggregation in Data Warehouses for further information about the `GROUPING` function.

A sample `CUBE` and `ROLLUP` query is the following:

```
SELECT  channel_desc, country_iso_code, SUM(amount_sold) SALES$,
  RANK() OVER (PARTITION BY GROUPING_ID(channel_desc, country_iso_code)
  ORDER BY SUM(amount_sold) DESC) AS RANK_PER_GROUP
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id
  AND countries.country_id = customers.country_id AND sales.channel_id =
channels.channel_id
  AND channels.channel_desc IN ('Direct Sales', 'Internet') AND
times.calendar_month_desc='2000-07'
  AND country_iso_code IN ('GB', 'US', 'JP')
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
GROUP BY cube(channel_desc, country_iso_code);

CHANNEL_DESC    CO  SALES$         RANK_PER_GROUP
--------------  --  -------------  --------------
Direct Sales      US    616539.04           1
Direct Sales      GB    83869.96          2
Internet          US    82595.71          3
Direct Sales      JP    79047.78          4
Internet          JP    7103.39              5
Internet          GB    6477.98              6
Direct Sales            779456.78           1
Internet                96177.08          2
                    US    699134.75           1
                    GB    90347.94          2
                    JP    86151.17          3
                          875633.86           1
```

### 20.2.1.1.6 Examples: Treatment of NULLs in Ranking Functions

NULLs are treated like normal values. Also, for rank computation, a NULL value is assumed to be equal to another NULL value. Depending on the ASC | DESC options provided for measures and the NULLS FIRST | NULLS LAST clause, NULLs will either sort low or high and hence, are given ranks appropriately. The following example shows how NULLs are ranked in different cases:

```
SELECT  times.time_id time,  sold,
  RANK() OVER (ORDER BY (sold) DESC NULLS LAST) AS NLAST_DESC,
  RANK() OVER (ORDER BY (sold) DESC NULLS FIRST) AS NFIRST_DESC,
  RANK() OVER (ORDER BY (sold) ASC NULLS FIRST) AS NFIRST,
  RANK() OVER (ORDER BY (sold) ASC NULLS LAST) AS NLAST
FROM
  (
   SELECT time_id, SUM(sales.amount_sold)  sold
   FROM sales, products, customers, countries
   WHERE sales.prod_id=products.prod_id
    AND customers.country_id = countries.country_id
    AND sales.cust_id=customers.cust_id
    AND prod_name IN ('Envoy Ambassador', 'Mouse Pad') AND country_iso_code ='GB'
   GROUP BY  time_id)
 v, times
WHERE v.time_id (+) = times.time_id
  AND calendar_year=1999
  AND calendar_month_number=1
ORDER BY sold  DESC NULLS LAST;

TIME            SOLD NLAST_DESC NFIRST_DESC     NFIRST      NLAST
--------- ---------- ---------- ----------- ---------- ----------
25-JAN-99    3097.32          1          18         31         14
17-JAN-99    1791.77          2          19         30         13
30-JAN-99     127.69          3          20         29         12
28-JAN-99     120.34          4          21         28         11
23-JAN-99      86.12          5          22         27         10
20-JAN-99      79.07          6          23         26          9
13-JAN-99       56.1          7          24         25          8
07-JAN-99      42.97          8          25         24          7
08-JAN-99      33.81          9          26         23          6
10-JAN-99      22.76         10          27         21          4
02-JAN-99      22.76         10          27         21          4
26-JAN-99      19.84         12          29         20          3
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
16-JAN-99      11.27         13          30         19          2
14-JAN-99       9.52         14          31         18          1
09-JAN-99                    15           1          1         15
12-JAN-99                    15           1          1         15
31-JAN-99                    15           1          1         15
11-JAN-99                    15           1          1         15
19-JAN-99                    15           1          1         15
03-JAN-99                    15           1          1         15
15-JAN-99                    15           1          1         15
21-JAN-99                    15           1          1         15
24-JAN-99                    15           1          1         15
04-JAN-99                    15           1          1         15
06-JAN-99                    15           1          1         15
27-JAN-99                    15           1          1         15
18-JAN-99                    15           1          1         15
01-JAN-99                    15           1          1         15
22-JAN-99                    15           1          1         15
29-JAN-99                    15           1          1         15
05-JAN-99                    15           1          1         15
```

## 20.2.1.2 APPROX_RANK Function

The `APPROX_RANK` function returns the approximate value in a group of values.

This function takes an optional `PARTITION BY` clause followed by a mandatory `ORDER BY ... DESC` clause. The `PARTITION BY` key must be a subset of the `GROUP BY` key. The `ORDER BY` clause must include either `APPROX_COUNT` or `APPROX_SUM`.

The `APPROX_RANK` function has the following syntax:

```
SELECT expr_1[, expr_2, … expr_j], APPROX_*(expr_k) agg_1[, APPROX_*(expr_l)
agg_2…]
FROM table_name
WHERE …
GROUP BY expr_1[, expr_2, …expr_j]
HAVING APPROX_RANK(PARTITION BY partition_by_clause ORDER BY APPROX_*(expr_k)
DESC) <= N1
[AND APPROX_RANK(PARTITION BY partition_by_clause ORDER BY APPROX_*(expr_l)
DESC) <= N2…)];
```

The examples on this page use the `hr` (Human Resources) sample schema provided by Oracle.

In the following example, the query returns the jobs that are among the top 10 total salaries per department. For each job, the total salary and ranking is also given:

```
SELECT department_id, job_id, APPROX_SUM(salary), APPROX_RANK(PARTITION BY
department_id ORDER BY APPROX_SUM(salary) DESC) rk FROM hr.employees GROUP BY
department_id, job_id HAVING APPROX_RANK(PARTITION BY department_id ORDER BY
APPROX_SUM(salary) DESC) <= 10;
  DEPARTMENT_ID    JOB_ID      APPROX_SUM(SALARY)    RK
  _____            _________   __________________    ____
   10              AD_ASST     4400                 1
   20              MK_REP      6000                 2
   20              MK_MAN      13000                 1
   30              PU_MAN      11000                 2
   30              PU_CLERK    13900                 1
   40              HR_REP      6500                 1
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
  50              ST_MAN      36400                3
  50              ST_CLERK    55700                2
  50              SH_CLERK    64300              1
  60              IT_PROG     28800              1
  70              PR_REP      10000              1
  80              SA_MAN      61000              2
  80              SA_REP      243500              1
  90              AD_PRES     24000                2
  90              AD_VP       34000              1
 100              FI_MGR      12008              2
 100              FI_ACCOUNT  39600                1
 110              AC_ACCOUNT  8300               2
 110              AC_MGR      12008              1
                  SA_REP       7000              1
```

In the following example, the query returns the jobs that are among the top 2 in terms of total salary and among the top 3 in terms of number of employees holding the job titles per department:

```
SELECT department_id, job_id , APPROX_SUM(salary), APPROX_COUNT(*) FROM
employees GROUP BY department_id, job_id  HAVING APPROX_RANK(PARTITION BY
department_id ORDER BY APPROX_SUM(salary) DESC) <= 2 AND
APPROX_RANK(PARTITION BY department_id ORDER BY APPROX_COUNT(*) DESC) <= 3;

    DEPTNO JOB       APPROX_SUM(SAL) APPROX_COUNT(*)
---------- --------- --------------- ---------------
        10 MANAGER              2450               1
        10 PRESIDENT            5000               1
        20 MANAGER              2975               1
        20 ANALYST              6000               2
        30 MANAGER              2850               1
        30 SALESPERSON          5600               4
```

The following example reports the accuracy of the approximate aggregate using the `MAX_ERROR` attribute:

```
SELECT department_id, job_id , APPROX_SUM(salary) sum_salary,
APPROX_SUM(salary,'MAX_ERROR') sum_salary_err
FROM employees
GROUP BY department_id, job_id
HAVING APPROX_RANK(PARTITION BY department_id ORDER BY APPROX_SUM(salary)
DESC) <= 2;
 DEPARTMENT_ID JOB_ID           SUM_SALARY    SUM_SALARY_ERR
_____________ _____________ _____________ _________________
             10 AD_ASST              4400                 0
             20 MK_REP               6000                 0
             20 MK_MAN              13000                 0
             30 PU_MAN              11000                 0
             30 PU_CLERK            13900                 0
             40 HR_REP               6500                 0
             50 ST_CLERK            55700                 0
             50 SH_CLERK            64300                 0
             60 IT_PROG             28800                 0
             70 PR_REP              10000                 0
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
 80 SA_MAN                61000                0
 80 SA_REP               243500                0
 90 AD_PRES               24000                0
 90 AD_VP                 34000                0
100 FI_MGR                12008                0
100 FI_ACCOUNT            39600                0
110 AC_ACCOUNT             8300                0
110 AC_MGR                12008                0
    SA_REP                 7000                0
```

> **See Also**
>
> - *Oracle AI Database SQL Language Reference*

### 20.2.1.3 Bottom N Ranking Functions

Bottom N is similar to top N except for the ordering sequence within the rank expression. Using the previous example, you can order `SUM(s_amount)` ascending instead of descending.

### 20.2.1.4 CUME_DIST Function

The `CUME_DIST` function (defined as the inverse of percentile in some statistical books) computes the position of a specified value relative to a set of values. The order can be ascending or descending. Ascending is the default. The range of values for `CUME_DIST` is from greater than 0 to 1. To compute the `CUME_DIST` of a value x in a set S of size N, you use the formula:

```
CUME_DIST(x) =  number of values in S coming before
   and including x in the specified order/ N
```

Its syntax is:

```
CUME_DIST ( ) OVER ( [query_partition_clause] order_by_clause )
```

The semantics of various options in the `CUME_DIST` function are similar to those in the `RANK` function. The default order is ascending, implying that the lowest value gets the lowest `CUME_DIST` (as all other values come later than this value in the order). `NULL`s are treated the same as they are in the `RANK` function. They are counted toward both the numerator and the denominator as they are treated like non-`NULL` values. The following example finds cumulative distribution of sales by channel within each month:

```
SELECT calendar_month_desc AS MONTH, channel_desc,
     TO_CHAR(SUM(amount_sold) , '9,999,999,999') SALES$,
     CUME_DIST() OVER (PARTITION BY calendar_month_desc ORDER BY
         SUM(amount_sold) ) AS CUME_DIST_BY_CHANNEL
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id AND sales.cust_id=customers.cust_id
  AND sales.time_id=times.time_id AND sales.channel_id=channels.channel_id
  AND times.calendar_month_desc IN ('2000-09', '2000-07','2000-08')
GROUP BY calendar_month_desc, channel_desc;

MONTH    CHANNEL_DESC         SALES$         CUME_DIST_BY_CHANNEL
-------- -------------------- -------------- --------------------
2000-07  Internet                   140,423           .333333333
2000-07  Partners                   611,064           .666666667
2000-07  Direct Sales             1,145,275                    1
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
2000-08  Internet                 215,107          .333333333
2000-08  Partners                 661,045          .666666667
2000-08  Direct Sales           1,236,104                   1
2000-09  Internet                 228,241          .333333333
2000-09  Partners                 666,172          .666666667
2000-09  Direct Sales           1,217,808                   1
```

## 20.2.1.5 PERCENT_RANK Function

PERCENT_RANK is similar to CUME_DIST, but it uses rank values rather than row counts in its numerator. Therefore, it returns the percent rank of a value relative to a group of values. The function is available in many popular spreadsheets. PERCENT_RANK of a row is calculated as:

```
(rank of row in its partition - 1) / (number of rows in the partition - 1)
```

PERCENT_RANK returns values in the range zero to one. The row(s) with a rank of 1 will have a PERCENT_RANK of zero. Its syntax is:

```
PERCENT_RANK () OVER ([query_partition_clause] order_by_clause)
```

## 20.2.1.6 NTILE Function

NTILE allows easy calculation of tertiles, quartiles, deciles and other common summary statistics. This function divides an ordered partition into a specified number of groups called **buckets** and assigns a bucket number to each row in the partition. NTILE is a very useful calculation because it lets users divide a data set into fourths, thirds, and other groupings.

The buckets are calculated so that each bucket has exactly the same number of rows assigned to it or at most 1 row more than the others. For instance, if you have 100 rows in a partition and ask for an NTILE function with four buckets, 25 rows will be assigned a value of 1, 25 rows will have value 2, and so on. These buckets are referred to as equiheight buckets.

If the number of rows in the partition does not divide evenly (without a remainder) into the number of buckets, then the number of rows assigned for each bucket will differ by one at most. The extra rows will be distributed one for each bucket starting from the lowest bucket number. For instance, if there are 103 rows in a partition which has an NTILE(5) function, the first 21 rows will be in the first bucket, the next 21 in the second bucket, the next 21 in the third bucket, the next 20 in the fourth bucket and the final 20 in the fifth bucket.

The NTILE function has the following syntax:

```
NTILE (expr) OVER ([query_partition_clause] order_by_clause)
```

In this, the N in NTILE(N) can be a constant (for example, 5) or an expression.

This function, like RANK and CUME_DIST, has a PARTITION BY clause for per group computation, an ORDER BY clause for specifying the measures and their sort order, and NULLS FIRST | NULLS LAST clause for the specific treatment of NULLs. For example, the following is an example assigning each month's sales total into one of four buckets:

```
SELECT calendar_month_desc AS MONTH , TO_CHAR(SUM(amount_sold),
 '9,999,999,999')
 SALES$, NTILE(4) OVER (ORDER BY SUM(amount_sold)) AS TILE4
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id AND sales.cust_id=customers.cust_id
  AND sales.time_id=times.time_id AND sales.channel_id=channels.channel_id
  AND times.calendar_year=2000 AND prod_category= 'Electronics'
GROUP BY calendar_month_desc;

MONTH    SALES$              TILE4
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
-------- -------------- ----------
2000-02         242,416          1
2000-01         257,286          1
2000-03         280,011          1
2000-06         315,951          2
2000-05         316,824          2
2000-04         318,106          2
2000-07         433,824          3
2000-08         477,833          3
2000-12         553,534          3
2000-10         652,225          4
2000-11         661,147          4
2000-09         691,449          4
```

NTILE ORDER BY statements must be fully specified to yield reproducible results. Equal values can get distributed across adjacent buckets. To ensure deterministic results, you must order on a unique key.

### 20.2.1.7 ROW_NUMBER Function

The ROW_NUMBER function assigns a unique number (sequentially, starting from 1, as defined by ORDER BY) to each row within the partition. It has the following syntax:

```
ROW_NUMBER ( ) OVER ( [query_partition_clause] order_by_clause )
```

**Example 20-6 ROW_NUMBER**

```
SELECT channel_desc, calendar_month_desc,
   TO_CHAR(TRUNC(SUM(amount_sold), -5), '9,999,999,999') SALES$,
   ROW_NUMBER() OVER (ORDER BY TRUNC(SUM(amount_sold), -6) DESC) AS ROW_NUMBER
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id AND sales.cust_id=customers.cust_id
  AND sales.time_id=times.time_id AND sales.channel_id=channels.channel_id
  AND times.calendar_month_desc IN ('2021-09', '2021-10')
GROUP BY channel_desc, calendar_month_desc;

CHANNEL_DESC    CALENDAR  SALES$         ROW_NUMBER
--------------  --------  -------------- ----------
Direct Sales    2021-09   1,200,000      1
Direct Sales    2021-10   1,200,000      2
Internet        2021-10   200,000        3
Partners        2021-10   600,000        4
Partners        2021-09   600,000        5
Internet        2021-09   200,000        6
```

Note that there are three pairs of tie values in these results. Like NTILE, ROW_NUMBER is a non-deterministic function, so each tied value could have its row number switched. To ensure deterministic results, you must order on a unique key. In most cases, that will require adding a new tie breaker column to the query and using it in the ORDER BY specification.

## 20.2.2 Windowing Functions

Windowing functions can be used to compute cumulative, moving, and centered aggregates. They return a value for each row in the table, which depends on other rows in the corresponding window. With windowing aggregate functions, you can calculate moving and cumulative versions of SUM, AVERAGE, COUNT, MAX, MIN, and many more functions. They can be used only in the SELECT and ORDER BY clauses of the query. Windowing aggregate functions include the convenient FIRST_VALUE, which returns the first value in the window; and


Copyright © 2001, 2025, Oracle and/or its affiliates.

LAST_VALUE, which returns the last value in the window. These functions provide access to more than one row of a table without a self-join.

The syntax of the windowing function is:

```
analytic_function([ arguments ])
   OVER {window_name | (analytic_clause)}

 where analytic_clause =
     [ window_name | query_partition_clause ]
     [ order_by_clause [ windowing_clause ] ]

and query_partition_clause =
    PARTITION BY
      { value_expr[, value_expr ]...
      }

and windowing_clause =
     { ROWS | RANGE | GROUPS }
     { BETWEEN
       { UNBOUNDED PRECEDING
       | CURRENT ROW
       | value_expr { PRECEDING | FOLLOWING }
       }
       AND
       { UNBOUNDED FOLLOWING
       | CURRENT ROW
       | value_expr { PRECEDING | FOLLOWING }
       }
    | { UNBOUNDED PRECEDING
      | CURRENT ROW
      | value_expr PRECEDING
      }
}
[ EXCLUDE CURRENT ROW
| EXCLUDE GROUP
| EXCLUDE TIES
| EXCLUDE NO OTHERS ]
```

Note the following:

- The DISTINCT keyword is not supported in windowing functions except for MAX and MIN.
- If GROUPS is specified, then as is similar to ROWS, value_expr must be either a constant or an expression and must evaluate to a positive numeric value.

> **See Also**
>
> *Oracle Database SQL Language Reference* for further information regarding syntax and restrictions

This section contains the following topics:

- About Treatment of NULLs as Input to Window Functions
- Windowing Functions with Logical Offset
- Centered Aggregate Function
- Windowing Aggregate Functions in the Presence of Duplicates


Copyright © 2001, 2025, Oracle and/or its affiliates.

- Varying Window Size for Each Row
- Windowing Aggregate Functions with Physical Offsets
- Parallel Partition-Wise Operations with Windowing Functions

## 20.2.2.1 Examples of Window Clauses

A window clause can be implemented within a windowing function as shown in these examples.

In these examples note that instead of repeating the same analytic clause multiple times, we can define a window name for it and refer to the name in multiple windowing functions. The second example also shows how one window name can be built on top of another window name.

```
select ename, deptno, sal,
       sum(sal) over (w1) sum_sal,
       min(sal) over (w1) min_sal,
       avg(sal) over (w1) avg_sal,
       sum(sal) over (w2) cum_sal
from emp
window w1 as (partition by deptno),
       w2 as (partition by deptno order by sal);

select ename, deptno, sal,
       sum(sal) over (w1 order by sal) cum_sal1,
       sum(sal) over (w2) cum_sal2
from emp
window w1 as (partition by deptno),
       w2 as (w1 order by sal);

select ename, deptno, sal,
       min(sal) over w1 min_sal_3,
       max(sal) over w1 max_sal_3
from emp
window w1 as (partition by deptno order by sal
              rows between 1 preceding and 1 following);
```

> **Note**
>
> When the window name is specified with a windowing clause, it can only be referenced directly (without parentheses). The example below demonstrates this restriction. Notice that the window name `w1` is in parentheses in several places.

```
select ename, deptno, sal,
     min(sal) over (w1) min_sal_3,
     max(sal) over w1 max_sal_3
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
  from emp
  window w1 as (partition by deptno order by sal
                rows between 1 preceding and 1 following);
```

This query results in the following error.

```
ERROR at line 2:
ORA-32785: cannot reference a window name defined with WINDOWING clause
```

## 20.2.2.2 Examples of Windowing Clause Extensions

A windowing clause extension can be implemented within a windowing function as shown in these examples.

The following example shows windowing clauses using the ROWS and EXCLUDE clause with various options.

```
select sal,
       sum(sal) over (w rows between 1 preceding and 1 following
                       exclude current row) as exclude_current_row,
       sum(sal) over (w rows between 1 preceding and 1 following
                       exclude group) as exclude_group,
       sum(sal) over (w rows between 1 preceding and 1 following
                       exclude ties) as exclude_ties,
       sum(sal) over (w rows between 1 preceding and 1 following
                       exclude no others) as exclude_no_others
from emp
window w as (order by sal);
```

The following example shows windowing clauses using the RANGE and EXCLUDE clause with various options.

```
select sal,
       sum(sal) over (w range between 100 preceding and 100 following
                       exclude current row) as exclude_current_row,
       sum(sal) over (w range between 100 preceding and 100 following
                       exclude group) as exclude_group,
       sum(sal) over (w range between 100 preceding and 100 following
                      exclude ties) as exclude_ties,
       sum(sal) over (w range between 100 preceding and 100 following
                       exclude no others) as exclude_no_others
from emp
window w as (order by sal);
```

The following example shows windowing clauses using the GROUPS and EXCLUDE clause with various options.

```
select sal,
       sum(sal) over (w groups between 1 preceding and 1 following
                       exclude current row) as exclude_current_row,
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
       sum(sal) over (w groups between 1 preceding and 1 following
                       exclude group) as exclude_group,
       sum(sal) over (w groups between 1 preceding and 1 following
                      exclude ties) as exclude_ties,
       sum(sal) over (w groups between 1 preceding and 1 following
                       exclude no others) as exclude_no_others
from emp
window w as (order by sal);
```

## 20.2.2.3 About Treatment of NULLs as Input to Window Functions

Window functions' `NULL` semantics match the `NULL` semantics for SQL aggregate functions. Other semantics can be obtained by user-defined functions, or by using the `DECODE` or a `CASE` expression within the window function.

## 20.2.2.4 Windowing Functions with Logical Offset

A logical offset can be specified with constants such as `RANGE 10 PRECEDING`, or an expression that evaluates to a constant, or by an interval specification like `RANGE INTERVAL N DAY/MONTH/YEAR PRECEDING` or an expression that evaluates to an interval.

With logical offset, there can only be one expression in the `ORDER BY` expression list in the function, with type compatible to `NUMERIC` if offset is numeric, or `DATE` if an interval is specified.

An analytic function that uses the `RANGE` keyword can use multiple sort keys in its `ORDER BY` clause if it specifies either of these two windows:

- `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`. The short form of this is `RANGE UNBOUNDED PRECEDING`, which can also be used.
- `RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING`.

Window boundaries that do not meet these conditions can have only one sort key in the analytic function's `ORDER BY` clause.

**Example 20-7 Cumulative Aggregate Function**

The following is an example of cumulative `amount_sold` by customer ID by quarter in 2000:

```
SELECT c.cust_id, t.calendar_quarter_desc, TO_CHAR (SUM(amount_sold),
  '9,999,999,999.99') AS Q_SALES, TO_CHAR(SUM(SUM(amount_sold))
OVER (PARTITION BY c.cust_id ORDER BY c.cust_id, t.calendar_quarter_desc
ROWS UNBOUNDED
PRECEDING), '9,999,999,999.99') AS CUM_SALES
  FROM sales s, times t, customers c
  WHERE s.time_id=t.time_id AND s.cust_id=c.cust_id AND t.calendar_year=2022
    AND c.cust_id IN (2595, 9646, 11111)
  GROUP BY c.cust_id, t.calendar_quarter_desc
  ORDER BY c.cust_id, t.calendar_quarter_desc;
```

```
CUST_ID CALENDAR_QUARTER_DESC     Q_SALES       CUM_SALES
_______ ____________________ ____________ _____________
   2595              2022-01        98.16         98.16
   2595              2022-02        95.11        193.27
   2595              2022-04    10,902.18     11,095.45
   9646              2022-01       422.64        422.64
   9646              2022-02     1,220.11      1,642.75
   9646              2022-03     1,121.65      2,764.40
   9646              2022-04     1,381.58      4,145.98
  11111              2022-01     6,080.14      6,080.14
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
     11111              2022-02     3,322.30      9,402.44
     11111              2022-03     1,073.12     10,475.56
     11111              2022-04    1,420.24      11,895.80
```

In this example, the analytic function SUM defines, for each row, a window that starts at the beginning of the partition (UNBOUNDED PRECEDING) and ends, by default, at the current row.

Nested SUMs are needed in this example because you are performing a SUM over a value that is itself a SUM. Nested aggregations are used very often in analytic aggregate functions.

**Example 20-8 Moving Aggregate Function**

This example of a time-based window shows, for one customer, the moving average of sales for the current month and preceding two months:

```
SELECT c.cust_id, t.calendar_month_desc, TO_CHAR (SUM(amount_sold),
      '9,999,999,999') AS SALES, TO_CHAR(AVG(SUM(amount_sold))
OVER (ORDER BY c.cust_id, t.calendar_month_desc ROWS 2 PRECEDING),
      '9,999,999,999') AS MOVING_3_MONTH_AVG
FROM sales s, times t, customers c
WHERE s.time_id=t.time_id AND s.cust_id=c.cust_id
   AND t.calendar_year=1999 AND c.cust_id IN (6510)
GROUP BY c.cust_id, t.calendar_month_desc
ORDER BY c.cust_id, t.calendar_month_desc;

   CUST_ID CALENDAR SALES          MOVING_3_MONTH
---------- -------- -------------- --------------
      6510 1999-04            125            125
      6510 1999-05          3,395          1,760
      6510 1999-06          4,080          2,533
      6510 1999-07          6,435          4,637
      6510 1999-08          5,105          5,207
      6510 1999-09          4,676          5,405
      6510 1999-10          5,109          4,963
      6510 1999-11            802          3,529
```

Note that the first two rows for the three month moving average calculation in the output data are based on a smaller interval size than specified because the window calculation cannot reach past the data retrieved by the query. You must consider the different window sizes found at the borders of result sets. In other words, you may need to modify the query to include exactly what you want.

## 20.2.2.5 Centered Aggregate Function

Calculating windowing aggregate functions centered around the current row is straightforward. This example computes for all customers a centered moving average of sales for one week in late December 1999. It finds an average of the sales total for the one day preceding the current row and one day following the current row including the current row as well.

The example below uses the sh sample schema.

**Example 20-9 Centered Aggregate**

```
SELECT t.time_id, TO_CHAR (SUM(amount_sold), '9,999,999,999')
AS SALES, TO_CHAR(AVG(SUM(amount_sold)) OVER
  (ORDER BY t.time_id
   RANGE BETWEEN INTERVAL '1' DAY PRECEDING AND
   INTERVAL '1' DAY FOLLOWING), '9,999,999,999') AS CENTERED_3_DAY_AVG
   FROM sh.sales s, sh.times t
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
   WHERE s.time_id=t.time_id AND t.calendar_week_number IN (51)
     AND calendar_year=2022
   GROUP BY t.time_id
   ORDER BY t.time_id;

   TIME_ID             SALES  CENTERED_3_DAY_AVG
__________ _________________  __________________
17-DEC-22           281,264           197,907
18-DEC-22           114,550           161,631
19-DEC-22            89,080           103,482
20-DEC-22           106,816           109,094
21-DEC-22           131,385            87,102
22-DEC-22            23,104            85,469
23-DEC-22           101,917            62,511
```

The starting and ending rows for each product's centered moving average calculation in the output data are based on just two days, because the window calculation cannot reach past the data retrieved by the query. As in the prior example, you must consider the different window sizes found at the borders of result sets: the query may need to be adjusted.

## 20.2.2.6 Windowing Aggregate Functions in the Presence of Duplicates

The following example illustrates how window aggregate functions compute values when there are duplicates, that is, when multiple rows are returned for a single ordering value. The query retrieves the quantity sold to several customers during a specified time range. (Although an inline view was used to define the base data set, it has no special significance and can be ignored.) The query defines a moving window that runs from the date of the current row to 10 days earlier.Note that the `RANGE` keyword is used to define the windowing clause of this example. This means that the window can potentially hold many rows for each value in the range. In this case, there are three pairs of rows with duplicate date values.

**Example 20-10 Windowing Aggregate Functions with Logical Offsets**

```
SELECT time_id, daily_sum, SUM(daily_sum) OVER (ORDER BY time_id
RANGE BETWEEN INTERVAL '10' DAY PRECEDING AND CURRENT ROW)
AS current_group_sum
FROM (SELECT time_id, channel_id, SUM(s.quantity_sold)
AS daily_sum
FROM sh.customers c, sh.sales s, sh.countries
WHERE c.cust_id=s.cust_id
  AND c.country_id = countries.country_id
  AND s.cust_id IN (638, 634, 753, 440 ) AND s.time_id BETWEEN '01-MAY-22'
  AND '13-MAY-23' GROUP BY time_id, channel_id);

TIME_ID         DAILY_SUM    CURRENT_GROUP_SUM
___________ _____________ ____________________
06-MAY-22               4                   12
06-MAY-22               8                   12
10-MAY-22               5                   24
10-MAY-22               4                   24
10-MAY-22               3                   24
12-MAY-22               3                   33
12-MAY-22               6                   33
16-MAY-22               2                   45
16-MAY-22               6                   45
16-MAY-22               4                   45
05-JUN-22               4                   16
05-JUN-22              12                   16
.
.
.
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

In the output of this example, all dates except May 6 and May 12 return two rows. Examine the commented numbers to the right of the output to see how the values are calculated. Note that each group in parentheses represents the values returned for a single day.

Note that this example applies only when you use the `RANGE` keyword rather than the `ROWS` keyword. It is also important to remember that with `RANGE`, you can only use 1 `ORDER BY` expression in the analytic function's `ORDER BY` clause. With the `ROWS` keyword, you can use multiple order by expressions in the analytic function's `ORDER BY` clause.

## 20.2.2.7 Varying Window Size for Each Row

There are situations where it is useful to vary the size of a window for each row, based on a specified condition. For instance, you may want to make the window larger for certain dates and smaller for others. Assume that you want to calculate the moving average of stock price over three working days. If you have an equal number of rows for each day for all working days and no non-working days are stored, then you can use a physical window function. However, if the conditions noted are not met, you can still calculate a moving average by using an expression in the window size parameters.

Expressions in a window size specification can be made in several different sources. the expression could be a reference to a column in a table, such as a time table. It could also be a function that returns the appropriate boundary for the window based on values in the current row. The following statement for a hypothetical stock price database uses a user-defined function in its `RANGE` clause to set window size:

```
SELECT t_timekey, AVG(stock_price)
        OVER (ORDER BY t_timekey RANGE fn(t_timekey) PRECEDING) av_price
FROM stock, time WHERE st_timekey = t_timekey
ORDER BY t_timekey;
```

In this statement, `t_timekey` is a date field. Here, *fn* could be a PL/SQL function with the following specification:

`fn(t_timekey)` returns

- 4 if `t_timekey` is Monday, Tuesday
- 2 otherwise
- If any of the previous days are holidays, it adjusts the count appropriately.

Note that, when window is specified using a number in a window function with `ORDER BY` on a date column, then it is converted to mean the number of days. You could have also used the interval literal conversion function, as `NUMTODSINTERVAL(fn(t_timekey), 'DAY')` instead of just `fn(t_timekey)` to mean the same thing. You can also write a PL/SQL function that returns an `INTERVAL` data type value.

## 20.2.2.8 Windowing Aggregate Functions with Physical Offsets

For windows expressed in rows, the ordering expressions should be unique to produce deterministic results. For this example, which uses the Oracle-provided `sh` sample schema, the following query is not deterministic because `time_id` is not unique in this result set.

**Example 20-11 Windowing Aggregate Functions With Physical Offsets**

```
SELECT t.time_id, TO_CHAR(amount_sold, '9,999,999,999') AS INDIV_SALE,
       TO_CHAR(SUM(amount_sold) OVER (PARTITION BY t.time_id ORDER BY t.time_id
ROWS UNBOUNDED PRECEDING), '9,999,999,999') AS CUM_SALES
FROM sh.sales s, sh.times t, sh.customers c
WHERE s.time_id=t.time_id AND s.cust_id=c.cust_id
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
  AND t.time_id IN
   (TO_DATE('11-DEC-2022'), TO_DATE('12-DEC-2022'))
  AND c.cust_id
BETWEEN 6500 AND 6600
ORDER BY t.time_id;

TIME_ID      INDIV_SALE          CUM_SALES
___________ ____________       ___________
11-DEC-22            61                 61
11-DEC-22           152                214
11-DEC-22           209                423
11-DEC-22            21                444
11-DEC-22            62                506
11-DEC-22           208                714
11-DEC-22            22                736
11-DEC-22            25                761
11-DEC-22            22                783
11-DEC-22           115                899
11-DEC-22           115              1,014
11-DEC-22           112              1,126
11-DEC-22           204              1,330
```

One way to handle this problem would be to add the `prod_id` column to the result set and order on both `time_id` and `prod_id`.

## 20.2.2.9 Parallel Partition-Wise Operations with Windowing Functions

SQL windowing functions can have a query partitioning clause that can partition a query result into groups based on expressions used in the clause. For parallel queries on partitioned tables, the partitioning defined by the clause can be used to perform a partition-wise operation if the requirements for such operations are satisfied. This achieves faster SQL windowing queries on partitioned tables.

> **See Also**
>
> *Oracle Database VLDB and Partitioning Guide*

## 20.2.2.10 QUALIFY Clause

Use the `QUALIFY` clause to filter the results of an analytic function in a `SELECT` statement.

The relationship between the `QUALIFY` clause and analytic functions is analogous to the relationship between the `HAVING` clause and the `GROUP BY` clause. While the `HAVING` clause filters grouped data, the `QUALIFY` clause filters rows after analytic functions (also known as window functions) are evaluated.

You can use `QUALIFY` to filter an analytic function that appears in either the `SELECT` list or within the predicate of the `QUALIFY` clause itself.

The `QUALIFY` clause must appear after the `WINDOW` clause and before the `ORDER BY` and `FETCH FIRST` clauses of a `SELECT` statement.


Copyright © 2001, 2025, Oracle and/or its affiliates.

All WHERE, GROUP BY, and HAVING clauses, as well as analytic functions, are processed before the QUALIFY clause. The general order of SQL operations is as follows:

```
FROM → WHERE → GROUP BY → HAVING → WINDOW → QUALIFY → DISTINCT → ORDER BY →
FETCH FIRST
```

**Example: Filtering Analytic Function Results Using QUALIFY**

The following example returns employees who work in locations where the average salary is greater than 2000. The analytic function AVG(SAL) OVER (PARTITION BY LOC) computes the average salary per location, and the QUALIFY clause filters out locations that do not meet the condition.

```
SELECT ename, sal, dname, loc
FROM emp, dept
WHERE emp.deptno = dept.deptno
QUALIFY AVG(sal) OVER (PARTITION BY loc) > 2000
ORDER BY ename;
```

**Output:**

```
ENAME             SAL DNAME          LOC
---------- ---------- -------------- -------------
ADAMS            1100 RESEARCH       DALLAS
CLARK            2450 ACCOUNTING     NEW YORK
FORD             3000 RESEARCH       DALLAS
JONES            2975 RESEARCH       DALLAS
KING             5000 ACCOUNTING     NEW YORK
MILLER           1300 ACCOUNTING     NEW YORK
SCOTT            3000 RESEARCH       DALLAS
SMITH             800 RESEARCH       DALLAS

8 rows selected.
```

In this query, the QUALIFY clause eliminates any rows from locations where the average salary is 2000 or less, simplifying logic that previously required nesting the query.

> **See Also**
>
> Analytic Functions in the *Oracle AI Database SQL Language Reference*

## 20.2.3 Reporting Functions

After a query has been processed, aggregate values like the number of resulting rows or an average value in a column can be easily computed within a partition and made available to other reporting functions. Reporting aggregate functions return the same aggregate value for every row in a partition. Their behavior with respect to NULLs is the same as the SQL aggregate functions. The syntax is:

```
{SUM | AVG | MAX | MIN | COUNT | STDDEV | VARIANCE  ... }
  ([ALL | DISTINCT] {value expression1 [,...] | *})
   OVER ([PARTITION BY value expression2[,...]])
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

In addition, the following conditions apply:

- An asterisk (*) is only allowed in `COUNT(*)`
- `DISTINCT` is supported only if corresponding aggregate functions allow it.
- *`value expression1`* and *`value expression2`* can be any valid expression involving column references or aggregates.
- The `PARTITION BY` clause defines the groups on which the windowing functions would be computed. If the `PARTITION BY` clause is absent, then the function is computed over the whole query result set.

> **See Also**
>
> RATIO_TO_REPORT Function

Reporting functions can appear only in the `SELECT` clause or the `ORDER BY` clause. The major benefit of reporting functions is their ability to do multiple passes of data in a single query block and speed up query performance. Queries such as "Count the number of salesmen with sales more than 10% of city sales" do not require joins between separate query blocks.

For example, consider the question "For each product category, find the region in which it had maximum sales". The equivalent SQL query against the `sh` sample schema using the `MAX` reporting aggregate function would be:

```
SELECT prod_category, country_region, sales
FROM (SELECT SUBSTR(p.prod_category,1,8) AS prod_category, co.country_region,
 SUM(amount_sold) AS sales,
MAX(SUM(amount_sold)) OVER (PARTITION BY prod_category) AS MAX_REG_SALES
FROM sh.sales s, sh.customers c, sh.countries co, sh.products p
WHERE s.cust_id=c.cust_id AND c.country_id=co.country_id
   AND s.prod_id =p.prod_id AND s.time_id = TO_DATE('11-OCT-2021')
GROUP BY prod_category, country_region)
WHERE sales = MAX_REG_SALES;
```

The inner query with the reporting aggregate function `MAX(SUM(amount_sold))` returns:

```
PROD_CATEGORY    COUNTRY_REGION         SALES    MAX_REG_SALES
_______________  _________________  _________  ________________
Baseball         Americas             10208.6          10208.6
Baseball         Asia                  1592.8          10208.6
Baseball         Europe               4174.76          10208.6
Cricket          Americas             4577.73          4577.73
Cricket          Asia                  731.14          4577.73
Cricket          Europe               1898.42          4577.73
```

The full query results are:

```
PROD_CATEGORY    COUNTRY_REGION         SALES
_______________  _________________  _________
Baseball         Americas             10208.6
Cricket          Americas             4577.73
```

**Example 20-12 Reporting Aggregate Example**

Reporting aggregates combined with nested queries enable you to answer complex queries efficiently. For example, what if you want to know the best selling products in your most


Copyright © 2001, 2025, Oracle and/or its affiliates.

significant product subcategories? The following is a query which finds the 5 top-selling products for each product subcategory that contributes more than 20% of the sales within its product category. You can run this against the sh sample schema.

```
SELECT SUBSTR(prod_category,1,8) AS CATEG, prod_subcategory, prod_id, SALES
FROM (SELECT p.prod_category, p.prod_subcategory, p.prod_id,
      SUM(amount_sold) AS SALES,
      SUM(SUM(amount_sold)) OVER (PARTITION BY p.prod_category) AS CAT_SALES,
      SUM(SUM(amount_sold)) OVER
         (PARTITION BY p.prod_subcategory) AS SUBCAT_SALES,
      RANK() OVER  (PARTITION BY p.prod_subcategory
         ORDER BY SUM(amount_sold) ) AS RANK_IN_LINE
     FROM sh.sales s, sh.customers c, sh.countries co, sh.products p
     WHERE s.cust_id=c.cust_id
       AND c.country_id=co.country_id AND  s.prod_id=p.prod_id
       AND s.time_id=to_DATE('11-OCT-2022')
     GROUP BY p.prod_category, p.prod_subcategory, p.prod_id
     ORDER BY prod_category, prod_subcategory)
  WHERE SUBCAT_SALES>0.2*CAT_SALES AND RANK_IN_LINE<=5;
```

### 20.2.3.1 RATIO_TO_REPORT Function

The RATIO_TO_REPORT function computes the ratio of a value to the sum of a set of values. If the expression value expression evaluates to NULL, RATIO_TO_REPORT also evaluates to NULL, but it is treated as zero for computing the sum of values for the denominator. Its syntax is:

```
RATIO_TO_REPORT ( expr ) OVER ( [query_partition_clause] )
```

In this, the following applies:

- expr can be any valid expression involving column references or aggregates.
- The PARTITION BY clause defines the groups on which the RATIO_TO_REPORT function is to be computed. If the PARTITION BY clause is absent, then the function is computed over the whole query result set.

**Example 20-13 RATIO_TO_REPORT**

To calculate RATIO_TO_REPORT of sales for each channel, you might use the following syntax:

```
SELECT ch.channel_desc, TO_CHAR(SUM(amount_sold),'9,999,999') AS SALES,
     TO_CHAR(SUM(SUM(amount_sold)) OVER (), '9,999,999') AS TOTAL_SALES,
     TO_CHAR(RATIO_TO_REPORT(SUM(amount_sold)) OVER (), '9.999')
     AS RATIO_TO_REPORT
FROM sh.sales s, sh.channels ch
WHERE s.channel_id=ch.channel_id  AND s.time_id=to_DATE('11-OCT-2019')
GROUP BY ch.channel_desc;

CHANNEL_DESC          SALES     TOTAL_SALES    RATIO_TO_REPORT
_____________ _____________ _______________ ___________________
Partners            7,764        28,158       .276
Direct Sales       14,798        28,158       .526
Internet            5,596        28,158       .199
```

## 20.2.4 LAG/LEAD Functions

The LAG and LEAD functions are useful for comparing values when the relative positions of rows can be known reliably. They work by specifying the count of rows which separate the target row from the current row. Because the functions provide access to more than one row of a table at the same time without a self-join, they can enhance processing speed. The LAG function


Copyright © 2001, 2025, Oracle and/or its affiliates.

provides access to a row at a given offset prior to the current position, and the LEAD function provides access to a row at a given offset after the current position. "LAG/LEAD Syntax" describes the syntax of these functions.

The LAG and LEAD functions can be thought of as being related to, and a simplification of, the NTH_VALUE function. With LAG and LEAD, you can only retrieve values from a row at the specified physical offset. If this is insufficient, you can use NTH_VALUE, which enables you to retrieve values from a row based on what is called a logical offset or relative position. You can use the IGNORE NULLS option with the NTH_VALUE function to make it more useful, in the sense that you can specify conditions and filter out rows based on certain conditions. See Example 20-17, where rows with quantities less than eight are filtered out. This cannot be done with LAG or LEAD, as you would not know the offset to the row.

See "NTH_VALUE Function" and *Oracle Database SQL Language Reference* for more information.

## 20.2.4.1 LAG/LEAD Syntax

These functions have the following syntax:

```
{LAG | LEAD} ( value_expr [, offset] [, default] ) [RESPECT NULLS|IGNORE NULLS]
   OVER ( [query_partition_clause] order_by_clause )
```

*offset* is an optional parameter and defaults to 1. *default* is an optional parameter and is the value returned if *offset* falls outside the bounds of the table or partition. When IGNORE NULLS is specified, the value returned will be from a row at a specified lag or lead offset after ignoring rows with NULLs.

**Example 20-14 LAG/LEAD**

This example illustrates a typical case of using LAG and LEAD:

```
SELECT time_id, TO_CHAR(SUM(amount_sold),'9,999,999') AS SALES,
  TO_CHAR(LAG(SUM(amount_sold),1) OVER (ORDER BY time_id),'9,999,999') AS LAG1,
  TO_CHAR(LEAD(SUM(amount_sold),1) OVER (ORDER BY time_id),'9,999,999') AS LEAD1
FROM sales
WHERE time_id>=TO_DATE('10-OCT-2019') AND time_id<=TO_DATE('22-OCT-2019')
GROUP BY time_id;

TIME_ID      SALES          LAG1           LEAD1
___________ _____________ _____________ _____________
10-OCT-19       179,364                        28,158
11-OCT-19        28,158       179,364        22,243
12-OCT-19        22,243        28,158        57,482
13-OCT-19        57,482        22,243       110,980
14-OCT-19       110,980        57,482        30,011
15-OCT-19        30,011       110,980        81,224
16-OCT-19        81,224        30,011       158,634
17-OCT-19       158,634        81,224        77,557
18-OCT-19        77,557       158,634        84,686
19-OCT-19        84,686        77,557       190,446
20-OCT-19       190,446        84,686       227,484
21-OCT-19       227,484       190,446        20,439
22-OCT-19        20,439       227,484
```

See "Data Densification for Reporting" for information showing how to use the LAG/LEAD functions for doing period-to-period comparison queries on sparse data.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Example 20-15 LAG/LEAD Using IGNORE NULLS**

This example illustrates a typical case of using LAG and LEAD with the IGNORE NULLS option:

```
SELECT prod_id, channel_id, SUM(quantity_sold) quantity,
 CASE WHEN SUM(quantity_sold) < 5000 THEN SUM(amount_sold) ELSE NULL END amount,
 LAG(CASE WHEN SUM(quantity_sold) < 5000 THEN SUM(amount_sold) ELSE NULL END)
 IGNORE NULLS OVER (PARTITION BY prod_id ORDER BY channel_id) lag
FROM sales
WHERE prod_id IN (18,127,138)
GROUP BY prod_id, channel_id;

 PROD_ID  CHANNEL_ID  QUANTITY   AMOUNT          LAG
--------  ----------  --------   -----------     ----------
      18           2      2888    4420923.94
      18           3      5615                    4420923.94
      18           4      1088    1545729.81      4420923.94
     127           2      4508     274088.08
     127           3      9626                     274088.08
     127           4      1850     173682.67       274088.08
     138           2      1120      127390.3
     138           3      3878     393111.15        127390.3
     138           4       543      71203.21       393111.15

9 rows selected.
```

## 20.2.5 FIRST_VALUE, LAST_VALUE, and NTH_VALUE Functions

This section illustrates the FIRST_VALUE, LAST_VALUE, and NTH_VALUE functions that are described in the following topics:

- FIRST_VALUE and LAST_VALUE Functions
- NTH_VALUE Function

### 20.2.5.1 FIRST_VALUE and LAST_VALUE Functions

The FIRST_VALUE and LAST_VALUE functions allow you to select the first and last rows from a window. These rows are especially valuable because they are often used as the baselines in calculations. For instance, with a partition holding sales data ordered by day, you might ask "How much was each day's sales compared to the first sales day (FIRST_VALUE) of the period?"

If the IGNORE NULLS option is used with FIRST_VALUE, it returns the first non-null value in the set, or NULL if all values are NULL. If IGNORE NULLS is used with LAST_VALUE, it returns the last non-null value in the set, or NULL if all values are NULL. The IGNORE NULLS option is particularly useful in populating an inventory table properly.

These functions have syntax as follows:

```
FIRST_VALUE|LAST_VALUE ( <expr> ) [RESPECT NULLS|IGNORE NULLS] OVER (analytic clause );
```

**Example 20-16 FIRST_VALUE**

This example illustrates using the IGNORE NULLS option with FIRST_VALUE:

```
SELECT prod_id, channel_id, time_id,
  CASE WHEN MIN(amount_sold) > 9.5
    THEN MIN(amount_sold) ELSE NULL END amount_sold,
  FIRST_VALUE(CASE WHEN MIN(amount_sold) > 9.5
    THEN min(amount_sold) ELSE NULL END)
    IGNORE NULLS OVER (PARTITION BY prod_id
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
                    ORDER BY channel_id DESC, time_id
                    ROWS BETWEEN UNBOUNDED PRECEDING
                    AND UNBOUNDED FOLLOWING) nv FROM sales
                    WHERE prod_id = 115 AND time_id BETWEEN '18-DEC-20'
                    AND '22-DEC-20' GROUP BY prod_id, channel_id, time_id
                    ORDER BY prod_id;

  PROD_ID CHANNEL_ID  TIME_ID    AMOUNT_SOLD         NV
---------- ----------  --------- ----------- ----------
       115          2  18-DEC-20                  10.91
       115          2  20-DEC-20                  10.91
       115          2  21-DEC-20                  10.91
       115          2  22-DEC-20                  10.91
       115          3  18-DEC-20        0.91      10.91
       115          3  20-DEC-20        0.91      10.91
       115          3  21-DEC-20       10.91      10.91
       115          3  22-DEC-20        10.8      10.91
       115          4  18-DEC-20       10.91
       115          4  21-DEC-20       10.91
13 rows selected.
```

## 20.2.5.2 NTH_VALUE Function

The NTH_VALUE function enables you to find column values from an arbitrary row in the window. This could be used when, for example, you want to retrieve the 5th highest closing price for a company's shares during a year.

The LAG and LEAD functions can be thought of as being related to, and a simplification of, the NTH_VALUE function. With LAG and LEAD, you can only retrieve values from a row at the specified physical offset. If this is insufficient, you can use NTH_VALUE, which enables you to retrieve values from a row based on what is called a logical offset or relative position. You can use the IGNORE NULLS option with the NTH_VALUE, FIRST_VALUE, and LAST_VALUE functions to make it more powerful, in the sense that you can specify conditions and filter out rows based on certain conditions. See Example 20-17, where rows with quantities less than eight are filtered out. This cannot be done with LAG or LEAD, as you would not know the offset to the row.

See *Oracle Database SQL Language Reference* for more information.

This function has syntax as follows:

```
NTH_VALUE (<expr>, <n expr>) [FROM FIRST | FROM LAST]
[RESPECT NULLS | IGNORE NULLS] OVER (<window specification>)
```

- expr can be a column, constant, bind variable, or an expression involving them.
- n can be a column, constant, bind variable, or an expression involving them.
- RESPECT NULLS is the default NULL handling mechanism. It determines whether null values of expr are included in or eliminated from the calculation. The default is RESPECT NULLS.
- The FROM FIRST and FROM LAST options determine whether the offset n is from the first or last row. The default is FROM FIRST.
- IGNORE NULLS enables you to skip NULLs in measure values.

**Example 20-17 NTH_VALUE**

The following example returns the amount_sold value of the second channel_id in ascending order for each prod_id in the range between 10 and 20:

```
SELECT prod_id, channel_id, MIN(amount_sold),
  NTH_VALUE(MIN(amount_sold), 2) OVER (PARTITION BY prod_id ORDER BY channel_id
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
  ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) NV
FROM sh.sales
WHERE prod_id BETWEEN 10 AND 20 GROUP BY prod_id, channel_id;

   PROD_ID CHANNEL_ID MIN(AMOUNT_SOLD)         NV
---------- ---------- ---------------- ----------
        13          2           907.34      906.2
        13          3            906.2      906.2
        13          4           842.21      906.2
        14          2          1015.94    1036.72
        14          3          1036.72    1036.72
        14          4           935.79    1036.72
        15          2           871.19     871.19
        15          3           871.19     871.19
        15          4           871.19     871.19
        16          2           266.84     266.84
        16          3           266.84     266.84
        16          4           266.84     266.84
        16          9            11.99     266.84
...
```

# 20.3 Advanced Aggregates for Analysis

Oracle AI Database provides multiple SQL functions to perform advanced aggregations. Additionally, for certain exact functions, corresponding functions that return approximate results are provided.

This section illustrates the following advanced analytic aggregate functions:

- About Approximate Aggregates
- LISTAGG Function
- FIRST/LAST Functions
- Inverse Percentile Functions
- Hypothetical Rank Functions
- Linear Regression Functions
- About Statistical Aggregates
- About User-Defined Aggregates

## 20.3.1 About Approximate Aggregates

Approximate aggregates are computed using SQL functions that return approximate results. They are used primarily in data exploration queries where exact values are not required and approximations are acceptable.

The `APPROX_COUNT_DISTINCT` function returns the approximate number of rows containing a distinct value for a specified expression. The `APPROX_COUNT_DISTINCT_DETAIL` and `APPROX_COUNT_DISTINCT_AGG` functions enable you to compute varying aggregated levels of approximate distinct value counts within specified groupings. The result of these aggregations can be stored in tables or materialized views for further analysis or answering user queries.

The `APPROX_COUNT_DISTINCT_DETAIL` function creates a base-level summary, in binary format, containing tuples for all the dimensions listed in the `WHERE` clause. The `APPROX_COUNT_DISTINCT_AGG` function uses the data generated by the


Copyright © 2001, 2025, Oracle and/or its affiliates.

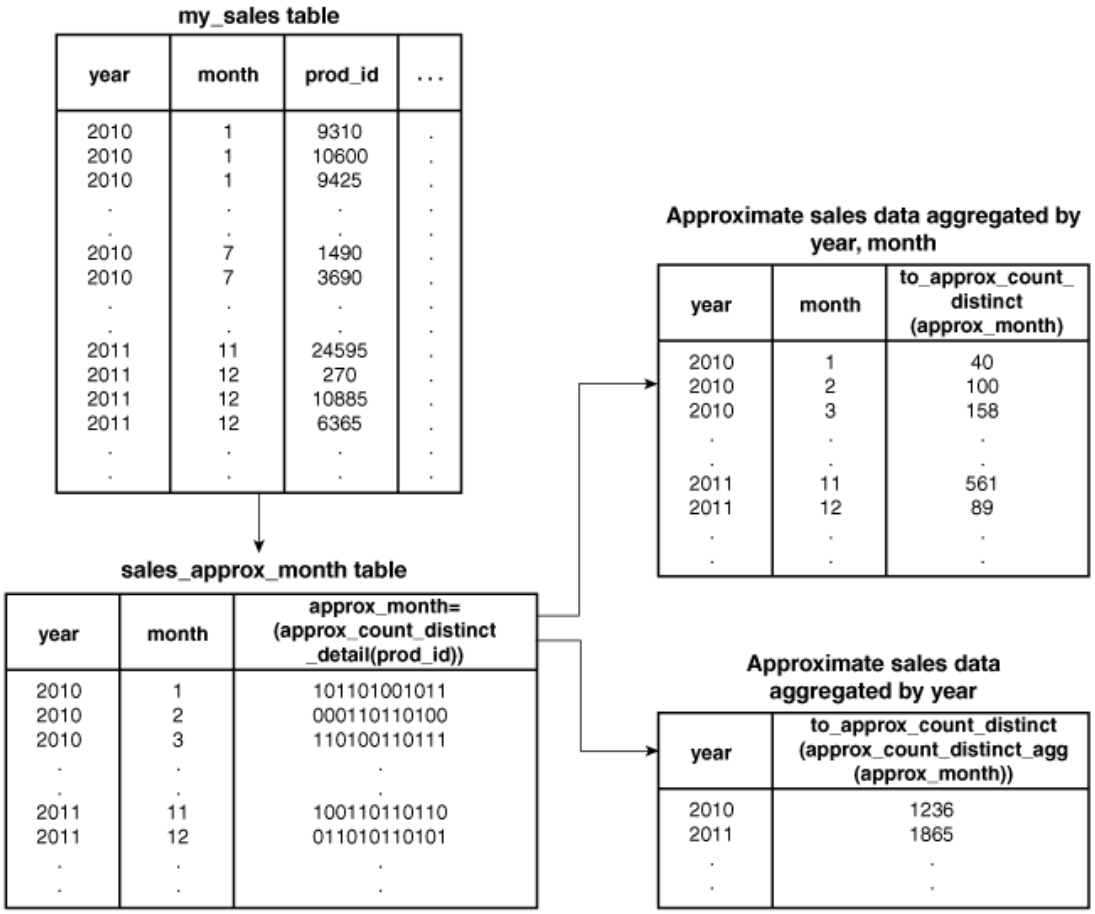
APPROX_COUNT_DISTINCT_DETAIL function to extract the higher level tuples in binary format. This avoids having to rerun the original calculation (in this case, the calculation with APPROX_COUNT_DISTINCT). The aggregate data that uses a binary format is converted into a human-readable format using TO_APPROX_COUNT_DISTINCT.

Figure 20-3 describes an example of using APPROX_COUNT_DISTINCT_DETAIL to obtain the approximate number of distinct products sold each month. The sales data selected from the my_sales table is aggregated by year and month and stored in the SALES_APPROX_MONTH table using a query such as the following:

```
INSERT INTO sales_approx_month
   (SELECT year, month, APPROX_COUNT_DISTINCT_DETAIL(prod_id) approx_month
      FROM my_sales
      GROUP BY year, month);
```

Notice that the values stored in approx_month are binary values. Use the TO_APPROX_COUNT_DISTINCT function to display these binary values in human readable format. To display the distinct number of products sold, aggregated by year and month, use the TO_APPROX_COUNT_DISTINCT function on the approx_month column. To display data aggregated by year, use the TO_APPROX_COUNT_DISTINCT function along with the APPROX_COUNT_DISTINCT_AGG function on the data stored in the approx_month column.

**Figure 20-3 Displaying Approximate Aggregates Using SQL Functions**





Copyright © 2001, 2025, Oracle and/or its affiliates.

Another approach to computing the approximate number of distinct products sold each year could be to use the APPROX_COUNT_DISTINCT_AGG to aggregate the monthly detail stored in the SALES_APPROX_MONTH table and store the results in a table or materialized view.

**Properties of SQL Functions that Return Approximate Percentile Results**

SQL functions that provide approximate percentile results include APPROX_PERCENTILE, APPROX_PERCENTILE_DETAIL, and APPROX_PERCENTILE_AGG. These functions have the following additional properties:

- ERROR_RATE

  Indicates the accuracy of the interpolated percentile values by computing the error rate of the approximate calculation

- CONFIDENCE

  Indicates the confidence in the accuracy of the error rate (when error rate is specified)

- DETERMINISTIC

  Controls the algorithm used to calculate approximations

  If you need consistent and repeatable results, then use DETERMINISTIC. This would typically be the case where results need to be shared with other users

**See Also**

- Using Percentile Functions that Return Approximate Results
- APPROX_COUNT_DISTINCT in *Oracle AI Database SQL Language Reference*
- APPROX_COUNT_DISTINCT_DETAIL in *Oracle AI Database SQL Language Reference* for information about the function and the ERROR_RATE, CONFIDENCE, and DETERMINISTIC properties
- APPROX_COUNT_DISTINCT_AGG in *Oracle AI Database SQL Language Reference*
- TO_APPROX_COUNT_DISTINCT in *Oracle AI Database SQL Language Reference*

## 20.3.2 LISTAGG Function

The LISTAGG function orders data within each group based on the ORDER BY clause and then concatenates the values of the measure column.

In releases prior to Oracle Database 12*c* Release 2 (12.2), if the concatenated value returned by the LISTAGG function exceeds the maximum length supported for the return data type, then the following error is returned:

```
ORA-01489: result of string concatenation is too long
```

Starting with Oracle Database 12*c* Release 2 (12.2), you can truncate the return string to fit within the maximum length supported for the return data type and display a truncation literal to indicate that the return value was truncated. The truncation is performed after the last complete data value thereby ensuring that no incomplete data value is displayed.

The syntax of the LISTAGG function is as follows:

```
LISTAGG ( [ALL] [DISTINCT] <measure_column> [,<delimiter>] [ON OVERFLOW TRUNCATE
[truncate_literal] | ON OVERFLOW ERROR  [WITH | WITHOUT COUNT]])
    WITHIN GROUP (ORDER BY <oby_expression_list>)
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

DISTINCT removes duplicate values from the list.

measure_column can be a column, constant, bind variable, or an expression involving them.

When the return string does not fit within the maximum length supported for the data type, you can either display an error or truncate the return string and display a truncation literal. The default is ON OVERFLOW ERROR, which displays an error when truncation occurs.

truncate_literal can be NULL, a string literal, or constant expression. It is appended to the end of the list of values, after the last delimiter, when LISTAGG returns a value that is larger than the maximum length supported for the return data type. The default value is an ellipsis (...).

WITH COUNT displays the number of data values that were truncated from the LISTAGG output because the maximum length supported for the return data type was exceeded. This is the default option. Use WITHOUT COUNT to omit displaying a count at the end of the LISTAGG function when the string is truncated.

delimiter can be NULL (default value), a string literal, bind variable, or constant expression. This is a mandatory parameter. If no delimiter is specified, then NULL is used as the delimiter.

oby_expression_list can be a list of expressions with optional ordering options to sort in ascending or descending order (ASC or DESC), and to control the sort order of NULLs (NULLS FIRST or NULLS LAST). ASCENDING and NULLS LAST are the defaults.

**See Also**

*Oracle AI Database SQL Language Reference* for information about the maximum length supported for the VARCHAR2 data type

## 20.3.2.1 LISTAGG as Aggregate

You can use the LISTAGG function as an aggregate.

**Example 20-18 LISTAGG as Aggregate**

The following example illustrates using LISTAGG as an aggregate.

```
SELECT prod_id, LISTAGG(cust_first_name||' '||cust_last_name, '; ')
  WITHIN GROUP (ORDER BY amount_sold DESC) cust_list
FROM sales, customers
WHERE sales.cust_id = customers.cust_id AND cust_gender = 'M'
  AND cust_credit_limit = 15000 AND prod_id BETWEEN 15 AND 18
  AND channel_id = 2 AND time_id > '01-JAN-01'
GROUP BY prod_id;

PROD_ID  CUST_LIST
-------  ------------------------------------------------
     15  Hope Haarper; Roxanne Crocker; ... Mason Murray
     16  Manvil Austin; Bud Pinkston; ... Helga Nickols
     17  Opal Aaron; Thacher Rudder; ... Roxanne Crocker
     18  Boyd Lin; Bud Pinkston; ... Erik Ready
```

The output has been modified for readability. In this case, the ellipsis indicate that some values before the last customer name have been omitted from the output.


Copyright © 2001, 2025, Oracle and/or its affiliates.

### Example 20-19 LISTAGG with Return String Exceeding the Maximum Permissible Length

This example orders data within each group specified by the `GROUP BY` clause and concatenates the values in the `cust_first_name` and `cust_last_name` columns. If the list of concatenated names exceeds the maximum length supported for the `VARCHAR2` data type, then the list is truncated to the last complete name. At the end of the list, the overflow literal of '...' is appended followed by the number of values that were truncated.

```
SELECT country_region,
  LISTAGG(s.CUST_FIRST_NAME||' '|| s.CUST_LAST_NAME, ';' ON OVERFLOW TRUNCATE
WITH COUNT) WITHIN GROUP (ORDER BY s.cust_id) AS customer_names
FROM countries c, customers s
WHERE c.country_id = s.country_id
GROUP BY c.country_region
ORDER BY c.country_region;

COUNTRY_REGION
--------------------
CUSTOMER_NAMES
--------------------------------------------------------------------------------
--
Africa
Laurice Lincoln;Kirsten Newkirk;Verna Yarborough;Chloe Dwyer;Betty
Sampler;Terry
 Hole;Waren Parkburg;Uwe Feldman;Douglas Hanson;Woodrow Lazar;Alfred
Doctor;Stac
.
.
Zwolinsky;Buzz Milenova;Abbie Venkayala

COUNTRY_REGION
--------------------
CUSTOMER_NAMES
--------------------------------------------------------------------------------
--
Americas
Linette Ingram;Vida Puleo;Gertrude Atkins;Sibil Haul;Raina Cassidy;Kaula
Daley;G
abriela Sean;Dolores Moore;Erica Vandermark;Madallyn Ladd;Carolyn
Hinkle;Leonora
.
.
emphill;Urban Smyth;Murry Ivy;Steven Lauers;...(21482)

COUNTRY_REGION
--------------------
CUSTOMER_NAMES
--------------------------------------------------------------------------------
--
Asia
Harriett Charles;Willa Fitz;Faith Fischer;Gay Nance;Maggie Cain;Neda
Clatterbuck
;Justa Killman;Penelope Oliver;Mandisa Grandy;Marette Overton;Astrid
Rice;Poppy
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
.
.
ob Gentile;Lynn Hardesty;Mabel Barajas;...(1648)

COUNTRY_REGION
--------------------
CUSTOMER_NAMES
--------------------------------------------------------------------------------
--
Europe
Abigail Kessel;Anne Koch;Buick Emmerson;Frank Hardy;Macklin Gowen;Rosamond
Kride
r;Raina Silverberg;Gloria Saintclair;Macy Littlefield;Yuri Finch;Bertilde
Sexton
.
.
el Floyd;Lincoln Sean;Morel Gregory;Kane Speer;...(30284)

COUNTRY_REGION
--------------------
CUSTOMER_NAMES
--------------------------------------------------------------------------------
--
Middle East
Dalila Rockwell;Alma Elliott;Cara Jeffreys;Joy Sandstrum;Elizabeth
Barone;Whitby
 Burnns;Geoffrey Door;Austin Dutton;Tobin Newcomer;Blake Overton;Lona
Kimball;Lo
.
.
edy;Brandon Moy;Sydney Fenton

COUNTRY_REGION
--------------------
CUSTOMER_NAMES
--------------------------------------------------------------------------------
--
Oceania
Fredericka Umstatt;Viola Nettles;Alyce Reagan;Catherine Odenwalld;Mauritia
Linde
green;Heidi Schmidt;Ray Wade;Cicily Graham;Myrtle Joseph;Joan Morales;Brenda
Obr
.
.
;Fredie Elgin;Gilchrist Lease;Guthrey Cain;...(793)

6 rows selected.
```

### Example 20-20 LISTAGG with Repeating Values Removed Using DISTINCT

This example orders data within each group specified by the `GROUP BY` clause and concatenates the values in the `prod_cateogry` and `prod_desc` columns. If the list of concatenated names exceeds the maximum length supported for the `VARCHAR2` data type, then


Copyright © 2001, 2025, Oracle and/or its affiliates.

the list is truncated to the last complete string. The DISTINCT keyword specifies that duplicate values in the specified measure column must be removed.

```
SELECT cust_id, LISTAGG( DISTINCT prod_category||':'||prod_desc,' ; ' ON
OVERFLOW TRUNCATE WITH COUNT)
WITHIN GROUP (ORDER BY amount_sold)
FROM sh.sales, sh.products
WHERE sales.prod_id=products.prod_id
AND amount_sold > 200 AND products.prod_id BETWEEN 10 and 15
AND time_id > '01-JAN-01'
GROUP BY cust_id;
```

### 20.3.2.2 LISTAGG as Reporting Aggregate

You can use the LISTAGG function as a reporting aggregate.

**Example 20-21 LISTAGG as Reporting Aggregate**

This example illustrates using LISTAGG as a reporting aggregate. It extracts the lowest unit cost for each product within each time period.

```
SELECT time_id, prod_id, LISTAGG(MIN(unit_cost),';')
      WITHIN GROUP (ORDER BY prod_id) OVER (PARTITION BY time_id) lowest_unit_cost
  FROM sh.sales_transactions_ext
  WHERE time_id BETWEEN '20-DEC-01' AND '22-DEC-01' AND prod_id BETWEEN 120 AND 125
  GROUP BY time_id, prod_id;

    TIME_ID      PROD_ID   LOWEST_UNIT_COST
    --------- ----------   ----------------------------------
    20-DEC-01        121    9.11;9.27;15.84;43.95
    20-DEC-01        122    9.11;9.27;15.84;43.95
    20-DEC-01        123    9.11;9.27;15.84;43.95
    21-DEC-01        120    9.11;9.27
    21-DEC-01        121    9.11;9.27
    22-DEC-01        120    9.11;9.27;15.84;43.95;16.06;12.66
    22-DEC-01        121    9.11;9.27;15.84;43.95;16.06;12.66
    22-DEC-01        122    9.11;9.27;15.84;43.95;16.06;12.66
    22-DEC-01        123    9.11;9.27;15.84;43.95;16.06;12.66
    22-DEC-01        124    9.11;9.27;15.84;43.95;16.06;12.66
    22-DEC-01        125    9.11;9.27;15.84;43.95;16.06;12.66
```

## 20.3.3 FIRST/LAST Functions

The FIRST/LAST aggregate functions allow you to rank a data set and work with its top-ranked or bottom-ranked rows. After finding the top or bottom ranked rows, an aggregate function is applied to any desired column. That is, FIRST/LAST lets you rank on column A but return the result of an aggregate applied on the first-ranked or last-ranked rows of column B. This is valuable because it avoids the need for a self-join or subquery, thus improving performance. These functions' syntax begins with a regular aggregate function (AVG, BIT_AND_AGG, BIT_OR_AGG, BIT_XOR_AGG, CHECKSUM, COUNT, KURTOSIS_POP, KURTOSIS_SAMP, MAX, MIN, SKEWNESS_POP, SKEWNESS_SAMP, STDDEV, SUM, and VARIANCE) that produces a single return value per group. To specify the ranking used, the FIRST/LAST functions add a new clause starting with the word KEEP.

These functions have the following syntax:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
aggregate_function KEEP ( DENSE_RANK FIRST | LAST ORDER BY
  expr [ DESC | ASC ] [NULLS { FIRST | LAST }]
  [, expr [ DESC | ASC ] [NULLS { FIRST | LAST }]]...)
[OVER query_partitioning_clause]
```

Note that the `ORDER BY` clause can take multiple expressions.

This section contains the following topics:

- FIRST/LAST As Regular Aggregates
- FIRST/LAST As Reporting Aggregates

## 20.3.3.1 FIRST/LAST As Regular Aggregates

You can use the `FIRST/LAST` family of aggregates as regular aggregate functions.

**Example 20-22 FIRST/LAST Example 1**

The following query lets us compare minimum price and list price of our products. For each product subcategory within the Tennis category, it returns the following:

- List price of the product with the lowest minimum price
- Lowest minimum price
- List price of the product with the highest minimum price
- Highest minimum price

```
SELECT prod_subcategory, MIN(prod_list_price)
  KEEP (DENSE_RANK FIRST ORDER BY (prod_min_price)) AS LP_OF_LO_MINP,
MIN(prod_min_price) AS LO_MINP,
MAX(prod_list_price) KEEP (DENSE_RANK LAST ORDER BY (prod_min_price))
 AS LP_OF_HI_MINP,
MAX(prod_min_price) AS HI_MINP
FROM sh.products WHERE prod_category='Tennis'
GROUP BY prod_subcategory;

PROD_SUBCATEGORY    LP_OF_LO_MINP   LO_MINP  LP_OF_HI_MINP  HI_MINP
----------------    -------------   -------  ------------- ----------
Tennis Balls                11.99     11.99        20.99      20.99
Tennis Racquet             299.99    299.99       599.99     599.99
Tennis Racquet Grip          7.99      7.99         7.99       7.99
Tennis Strings              19.99     19.99        29.99      29.99
```

## 20.3.3.2 FIRST/LAST As Reporting Aggregates

You can also use the `FIRST/LAST` family of aggregates as reporting aggregate functions. An example is calculating which months had the greatest and least increase in head count throughout the year. The syntax for these functions is similar to the syntax for any other reporting aggregate.

Consider the example in Example 20-22 for `FIRST/LAST`. What if you wanted to find the list prices of individual products and compare them to the list prices of the products in their subcategory that had the highest and lowest minimum prices?

The following query lets us find that information for the Cricket subcategory by using `FIRST/LAST` as reporting aggregates.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Example 20-23 FIRST/LAST Example 2**

```
SELECT prod_id, prod_list_price,
    MIN(prod_list_price) KEEP (DENSE_RANK FIRST ORDER BY (prod_min_price))
       OVER(PARTITION BY (prod_subcategory)) AS LP_OF_LO_MINP,
    MAX(prod_list_price) KEEP (DENSE_RANK LAST ORDER BY (prod_min_price))
       OVER(PARTITION BY (prod_subcategory)) AS LP_OF_HI_MINP
FROM sh.products WHERE prod_subcategory = 'Cricket';

   PROD_ID    PROD_LIST_PRICE    LP_OF_LO_MINP    LP_OF_HI_MINP
----------    ---------------    -------------    -------------
       48              11.99             6.99            22.99
      113              22.99             6.99            22.99
      114              18.99             6.99            22.99
      115               8.99             6.99            22.99
      116              11.99             6.99            22.99
      117               8.99             6.99            22.99
      118               7.99             6.99            22.99
      119               6.99             6.99            22.99
```

Using the `FIRST` and `LAST` functions as reporting aggregates makes it easy to include the results in calculations such as "Salary as a percent of the highest salary."

## 20.3.4 Inverse Percentile Functions

Using the `CUME_DIST` function, you can find the cumulative distribution (percentile) of a set of values. However, the inverse operation (finding what value computes to a certain percentile) is neither easy to do nor efficiently computed. To overcome this difficulty, the `PERCENTILE_CONT` and `PERCENTILE_DISC` functions were introduced. These can be used both as window reporting functions as well as normal aggregate functions.

These functions need a sort specification and a parameter that takes a percentile value between 0 and 1. The sort specification is handled by using an `ORDER BY` clause with one expression. When used as a normal aggregate function, it returns a single value for each ordered set.

`PERCENTILE_CONT` is a continuous function computed by interpolation and `PERCENTILE_DISC` is a step function that assumes discrete values. Like other aggregates, `PERCENTILE_CONT` and `PERCENTILE_DISC` operate on a group of rows in a grouped query, but with the following differences:

- They require a parameter between 0 and 1 (inclusive). A parameter specified out of this range results in error. This parameter should be specified as an expression that evaluates to a constant.
- They require a sort specification. This sort specification is an `ORDER BY` clause with a single expression. Multiple expressions are not allowed.

Starting with Oracle Database 12*c* Release 2 (12.2), the approximate inverse distribution function `APPROX_PERCENTILE` returns an approximate interpolated value that would fall into that percentile value with respect to the sort specification.

### 20.3.4.1 Normal Aggregate Syntax

```
[PERCENTILE_CONT | PERCENTILE_DISC]( constant expression )
    WITHIN GROUP ( ORDER BY single order by expression
[ASC|DESC] [NULLS FIRST| NULLS LAST])
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 20.3.4.2 Inverse Percentile Example Basis

Use the following queries to return customer data from the sh sample schema.

```
SELECT cust_id, cust_credit_limit, CUME_DIST()
   OVER (ORDER BY cust_credit_limit) AS CUME_DIST
FROM sh.customers WHERE cust_city='Marshal';

   CUST_ID    CUST_CREDIT_LIMIT     CUME_DIST
----------    -----------------     ----------------------------------------
     32497                 1500     0.17391304347826086956521739130434782608 7
     28344                 1500     0.17391304347826086956521739130434782608 7
     36651                 1500     0.17391304347826086956521739130434782608 7
      8962                 1500     0.17391304347826086956521739130434782608 7
    102343                 3000    0.34782608695652173913043478260869565217 39
    102077                 3000    0.34782608695652173913043478260869565217 39
      8270                 3000    0.34782608695652173913043478260869565217 39
     15192                 3000    0.34782608695652173913043478260869565217 39
    101784                 5000    0.52173913043478260869565217391304347826 09
     13808                 5000    0.52173913043478260869565217391304347826 09
     30420                 5000    0.52173913043478260869565217391304347826 09
     21380                 5000    0.52173913043478260869565217391304347826 09
     35266                 7000    0.65217391304347826086956521739130434782 61
     10346                 7000    0.65217391304347826086956521739130434782 61
     31112                 7000    0.65217391304347826086956521739130434782 61
    100977                 9000    0.73913043478260869565217391304347826086 96
      3424                 9000    0.73913043478260869565217391304347826086 96
    103066                10000    0.78260869565217391304347826086956521739 13
    100421                11000    0.95652173913043478260869565217391304347 83
     14459                11000    0.95652173913043478260869565217391304347 83
     17268                11000    0.95652173913043478260869565217391304347 83
     35225                11000    0.95652173913043478260869565217391304347 83
     41496                15000                                             1
```

PERCENTILE_DISC(x) is computed by scanning up the CUME_DIST values in each group till you find the first one greater than or equal to x, where x is the specified percentile value. For the example query where PERCENTILE_DISC(0.5), the result is 5,000, as the following illustrates:

```
SELECT PERCENTILE_DISC(0.5) WITHIN GROUP
  (ORDER BY cust_credit_limit) AS perc_disc, PERCENTILE_CONT(0.5) WITHIN GROUP
  (ORDER BY cust_credit_limit) AS perc_cont
 FROM sh.customers WHERE cust_city='Marshal';

PERC_DISC   PERC_CONT
---------   ---------
     5000        5000
```

The result of PERCENTILE_CONT is computed by linear interpolation between rows after ordering them. To compute PERCENTILE_CONT(x), you first compute the row number = RN= (1+x*(n-1)), where *n* is the number of rows in the group and *x* is the specified percentile value. The final result of the aggregate function is computed by linear interpolation between the values from rows at row numbers CRN = CEIL(RN) and FRN = FLOOR(RN).

The final result is: PERCENTILE_CONT(X) = if (CRN = FRN = RN), then (value of expression from row at RN) else (CRN - RN) * (value of expression for row at FRN) + (RN -FRN) * (value of expression for row at CRN).


Copyright © 2001, 2025, Oracle and/or its affiliates.

Consider the previous example query, where you compute PERCENTILE_CONT(0.5). Here *n* is 17. The row number RN = (1 + 0.5*(n-1))= 9 for both groups. Putting this into the formula, (FRN=CRN=9), you return the value from row 9 as the result.

Another example is, if you want to compute PERCENTILE_CONT(0.66). The computed row number RN=(1 + 0.66*(n-1))= (1 + 0.66*16)= 11.67. PERCENTILE_CONT(0.66) = (12-11.67)*(value of row 11)+(11.67-11)*(value of row 12). These results are:

```
SELECT PERCENTILE_DISC(0.66) WITHIN GROUP
  (ORDER BY cust_credit_limit) AS perc_disc, PERCENTILE_CONT(0.66) WITHIN GROUP
  (ORDER BY cust_credit_limit) AS perc_cont
FROM sh.customers WHERE cust_city='Marshal';

 PERC_DISC  PERC_CONT
---------- ----------
      9000       8040
```

Inverse percentile aggregate functions can appear in the HAVING clause of a query like other existing aggregate functions.

## 20.3.4.3 As Reporting Aggregates

You can also use the aggregate functions PERCENTILE_CONT, PERCENTILE_DISC as reporting aggregate functions. When used as reporting aggregate functions, the syntax is similar to those of other reporting aggregates.

```
[PERCENTILE_CONT | PERCENTILE_DISC](constant expression)
WITHIN GROUP ( ORDER BY single order by expression
[ASC|DESC] [NULLS FIRST| NULLS LAST])
OVER ( [PARTITION BY value expression [,...]] )
```

This query performs the same computation (median credit limit for customers in this result set), but reports the result for every row in the result set, as shown in the following output:

```
SELECT cust_id, cust_credit_limit, PERCENTILE_DISC(0.5) WITHIN GROUP
     (ORDER BY cust_credit_limit) OVER () AS perc_disc,
  PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY cust_credit_limit)
  OVER () AS perc_cont
FROM sh.customers WHERE cust_city='Marshal';

   CUST_ID   CUST_CREDIT_LIMIT    PERC_DISC     PERC_CONT
----------   -----------------   ----------    ----------
     32497                1500         5000         5000
     28344                1500         5000         5000
     36651                1500         5000         5000
      8962                1500         5000         5000
    102343                3000         5000         5000
    102077                3000         5000         5000
      8270                3000         5000         5000
     15192                3000         5000         5000
    101784                5000         5000         5000
     13808                5000         5000         5000
     30420                5000         5000         5000
     21380                5000         5000         5000
     35266                7000         5000         5000
     10346                7000         5000         5000
     31112                7000         5000         5000
    100977                9000         5000         5000
      3424                9000         5000         5000
    103066               10000         5000         5000
    100421               11000         5000         5000
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
14459          11000          5000          5000
17268          11000          5000          5000
35225          11000          5000          5000
41496          15000          5000          5000
```

## 20.3.4.4 Restrictions on Inverse Percentile Functions

For PERCENTILE_DISC, the expression in the ORDER BY clause can be of any data type that you can sort (numeric, string, date, and so on). However, the expression in the ORDER BY clause must be a numeric or datetime type (including intervals) because linear interpolation is used to evaluate PERCENTILE_CONT. If the expression is of type DATE, the interpolated result is rounded to the smallest unit for the type. For a DATE type, the interpolated value is rounded to the nearest second, for interval types to the nearest second (INTERVAL DAY TO SECOND) or to the month (INTERVAL YEAR TO MONTH).

Like other aggregates, the inverse percentile functions ignore NULLs in evaluating the result. For example, when you want to find the median value in a set, Oracle AI Database ignores the NULLs and finds the median among the non-null values. You can use the NULLS FIRST/NULLS LAST option in the ORDER BY clause, but they will be ignored as NULLs are ignored.

## 20.3.4.5 Using Percentile Functions that Return Approximate Results

Oracle AI Database provides a set of SQL functions that return approximate percentile results. These functions can be used to monitor quality, track social media activity, monitor performance, and search for outliers within a data set.

The following SQL functions compute and display approximate percentile results:

- APPROX_PERCENTILE

  Returns an approximate interpolated value that falls into the percentile value with respect to a sort specification. It can process large amounts of data significantly faster than the PERCENTILE_CONT with negligible deviation from the exact result.

- APPROX_PERCENTILE_DETAIL

  Calculates approximate percentile information, called a detail, within a set of data that is specified using a GROUP BY clause. Detail information created with this function is stored in binary format and is meant to be consumed by both the TO_APPROX_PERCENTILE and APPROX_PERCENT_DETAIL_AGG functions.

- APPROX_PERCENTILE_AGG

  Performs aggregations on the details created using the APPROX_PERCENTILE_DETAIL function.

- TO_APPROX_PECENTILE

  Displays the results of detail or aggregation, which are stored as BLOB values, in human readable format.

The detail and the higher level aggregated data can be stored in tables or materialized views for further analysis.

**Example: Displaying Approximate Percentile Sales Data Within a Country or State**

This example from the sh sample schema uses APPROX_PERCENTILE_DETAIL to perform percentile calculations once, store the results in table, and then perform approximate aggregations based on the stored data. The TO_APPROX_PERCENTILE function is used to display the results of the percentile calculations in human-readable format.


Copyright © 2001, 2025, Oracle and/or its affiliates.

1. Use `APPROX_PERCENTILE_DETAIL` to calculate the approximate percentile of the amount of sales in each state and store the results in a table called `approx_sales_percentile_detail`.

```
CREATE TABLE approx_sales_percentile_detail AS
SELECT c.country_id country, c.cust_state_province state,
approx_percentile_detail(amount_sold) detail
FROM sh.sales s, sh.customers c
WHERE s.cust_id = c.cust_id
GROUP BY c.country_id, c.cust_state_province;
```

2. Use `TO_APPROX_PERCENTILE` to query the detail and aggregate values stored in the table and display these values in human-readable format.

   The following statement uses the `APPROX_PERCENTILE_AGG` function to further aggregate the detail data stored in the `approx_sales_percentile_detail` table. The `TO_APPROX_PERCENTILE` function displays the aggregated results in human-readable format.

```
SELECT country, to_approx_percentile(approx_percentile_agg(detail),0.5)
median_amt_sold
FROM approx_sales_percentile_detail
GROUP BY country
ORDER BY country;

   COUNTRY    MEDIAN_AMT_SOLD
----------    ---------------
     52769              35.43
     52770              33.65
     52771              44.99
     52772              35.48
     52773              29.61
     52774              39.38
     52775              42.09
     52776              33.25
     52777               33.7
     52778              38.99
     52779              33.72
     52782              35.69
     52785              22.99
     52786              44.99
     52787              27.99
     52788              27.13
     52789              37.57
     52790               33.5
```

> **See Also**
>
> `APROX_PERCENTILE`, `APPROX_PERCENTILE_DETAIL`, `APPROX_PERCENTILE_AGG`, and `TO_APPROX_PERCENTILE` in *Oracle AI Database SQL Language Reference*


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 20.3.5 Hypothetical Rank Functions

These functions provide functionality useful for what-if analysis. As an example, what would be the rank of a row, if the row was hypothetically inserted into a set of other rows?

This family of aggregates takes one or more arguments of a hypothetical row and an ordered group of rows, returning the RANK, DENSE_RANK, PERCENT_RANK or CUME_DIST of the row as if it was hypothetically inserted into the group.

```
[RANK | DENSE_RANK | PERCENT_RANK | CUME_DIST]( constant expression [, ...] )
WITHIN GROUP ( ORDER BY order by expression [ASC|DESC] [NULLS FIRST|NULLS LAST][, ...] )
```

Here, *constant expression* refers to an expression that evaluates to a constant, and there may be more than one such expressions that are passed as arguments to the function. The ORDER BY clause can contain one or more expressions that define the sorting order on which the ranking will be based. ASC, DESC, NULLS FIRST, NULLS LAST options will be available for each expression in the ORDER BY.

**Example 20-24 Hypothetical Rank and Distribution Example 1**

Using the list price data from the products table used throughout this section, you can calculate the RANK, PERCENT_RANK and CUME_DIST for a hypothetical sweater with a price of $50 for how it fits within each of the sweater subcategories. The query and results are:

```
SELECT cust_city,
  RANK(6000) WITHIN GROUP (ORDER BY CUST_CREDIT_LIMIT DESC) AS HRANK,
  TO_CHAR(PERCENT_RANK(6000) WITHIN GROUP
  (ORDER BY cust_credit_limit),'9.999') AS HPERC_RANK,
  TO_CHAR(CUME_DIST (6000) WITHIN GROUP
  (ORDER BY cust_credit_limit),'9.999') AS HCUME_DIST
FROM sh.customers
WHERE cust_city LIKE 'Fo%'
GROUP BY cust_city;

CUST_CITY                       HRANK HPERC_ HCUME_
------------------------------ ---------- ------ ------
Fondettes                          13   .455   .478
Fords Prairie                      18   .320   .346
Forest City                        47   .370   .378
Forest Heights                     38   .456   .464
Forestville                        58   .412   .418
Forrestcity                        51   .438   .444
Fort Klamath                       59   .356   .363
Fort William                       30   .500   .508
Foxborough                         52   .414   .420
```

Unlike the inverse percentile aggregates, the ORDER BY clause in the sort specification for hypothetical rank and distribution functions may take multiple expressions. The number of arguments and the expressions in the ORDER BY clause should be the same and the arguments must be constant expressions of the same or compatible type to the corresponding ORDER BY expression. The following is an example using two arguments in several hypothetical ranking functions.

**Example 20-25 Hypothetical Rank and Distribution Example 2**

```
SELECT prod_subcategory,
  RANK(10,8) WITHIN GROUP (ORDER BY prod_list_price DESC,prod_min_price)
  AS HRANK, TO_CHAR(PERCENT_RANK(10,8) WITHIN GROUP
  (ORDER BY prod_list_price, prod_min_price),'9.999') AS HPERC_RANK,
  TO_CHAR(CUME_DIST (10,8) WITHIN GROUP
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
  (ORDER BY prod_list_price, prod_min_price),'9.999') AS HCUME_DIST
FROM sh.products WHERE prod_subcategory LIKE 'Baseball%'
GROUP BY prod_subcategory;

PROD_SUBCATEGORY        HRANK   HPERC_   HCUME_
--------------------    -----   ------   ------
Baseball Clothing           3     .000     .333
Baseballs                   6     .000     .167
```

These functions can appear in the `HAVING` clause of a query just like other aggregate functions. They cannot be used as either reporting aggregate functions or windowing aggregate functions.

## 20.3.6 Linear Regression Functions

The regression functions support the fitting of an ordinary-least-squares regression line to a set of number pairs. You can use them as both aggregate functions or windowing or reporting functions.

The regression functions are as follows:

- REGR_COUNT Function
- REGR_AVGY and REGR_AVGX Functions
- REGR_SLOPE and REGR_INTERCEPT Functions
- REGR_R2 Function
- REGR_SXX, REGR_SYY, and REGR_SXY Functions

Oracle applies the function to the set of (`e1`, `e2`) pairs after eliminating all pairs for which either of `e1` or `e2` is null. `e1` is interpreted as a value of the dependent variable (a "y value"), and `e2` is interpreted as a value of the independent variable (an "x value"). Both expressions must be numbers.

The regression functions are all computed simultaneously during a single pass through the data. They are frequently combined with the `COVAR_POP`, `COVAR_SAMP`, and `CORR` functions.

**See Also**

- Linear Regression Statistics Examples
- Sample Linear Regression Calculation

### 20.3.6.1 REGR_COUNT Function

`REGR_COUNT` returns the number of non-null number pairs used to fit the regression line. If applied to an empty set (or if there are no (`e1`, `e2`) pairs where neither of `e1` or `e2` is null), the function returns 0.

### 20.3.6.2 REGR_AVGY and REGR_AVGX Functions

`REGR_AVGY` and `REGR_AVGX` compute the averages of the dependent variable and the independent variable of the regression line, respectively. `REGR_AVGY` computes the average of its first argument (`e1`) after eliminating (`e1`, `e2`) pairs where either of `e1` or `e2` is null. Similarly,


Copyright © 2001, 2025, Oracle and/or its affiliates.

REGR_AVGX computes the average of its second argument (e2) after null elimination. Both functions return NULL if applied to an empty set.

### 20.3.6.3 REGR_SLOPE and REGR_INTERCEPT Functions

The REGR_SLOPE function computes the slope of the regression line fitted to non-null (e1, e2) pairs.

The REGR_INTERCEPT function computes the y-intercept of the regression line. REGR_INTERCEPT returns NULL whenever slope or the regression averages are NULL.

### 20.3.6.4 REGR_R2 Function

The REGR_R2 function computes the coefficient of determination (usually called "R-squared" or "goodness of fit") for the regression line.

REGR_R2 returns values between 0 and 1 when the regression line is defined (slope of the line is not null), and it returns NULL otherwise. The closer the value is to 1, the better the regression line fits the data.

### 20.3.6.5 REGR_SXX, REGR_SYY, and REGR_SXY Functions

REGR_SXX, REGR_SYY and REGR_SXY functions are used in computing various diagnostic statistics for regression analysis. After eliminating (e1, e2) pairs where either of e1 or e2 is null, these functions make the following computations:

```
REGR_SXX:    REGR_COUNT(e1,e2) * VAR_POP(e2)
REGR_SYY:    REGR_COUNT(e1,e2) * VAR_POP(e1)
REGR_SXY:    REGR_COUNT(e1,e2) * COVAR_POP(e1, e2)
```

### 20.3.6.6 Linear Regression Statistics Examples

Some common diagnostic statistics that accompany linear regression analysis are given in Table 20-2. Note that Oracle enables you to calculate all of these.

**Table 20-2 Common Diagnostic Statistics and Their Expressions**

| Type of Statistic | Expression |
|---|---|
| Adjusted R2 | `1-((1 - REGR_R2)*((REGR_COUNT-1)/(REGR_COUNT-2)))` |
| Standard error | `SQRT((REGR_SYY-(POWER(REGR_SXY,2)/REGR_SXX))/(REGR_COUNT-2))` |
| Total sum of squares | `REGR_SYY` |
| Regression sum of squares | `POWER(REGR_SXY,2) / REGR_SXX` |
| Residual sum of squares | `REGR_SYY` - (`POWER(REGR_SXY,2)/REGR_SXX`) |
| t statistic for slope | `REGR_SLOPE * SQRT(REGR_SXX)` / (Standard error) |
| t statistic for y-intercept | `REGR_INTERCEPT` / ((Standard error) * `SQRT((1/REGR_COUNT)+(POWER(REGR_AVGX,2)/REGR_SXX))` |

### 20.3.6.7 Sample Linear Regression Calculation

In this example from the sh sample schema, you compute an ordinary-least-squares regression line that expresses the quantity sold of a product as a linear function of the product's list price. The calculations are grouped by sales channel. The values SLOPE, INTCPT,


Copyright © 2001, 2025, Oracle and/or its affiliates.

RSQR are slope, intercept, and coefficient of determination of the regression line, respectively. The (integer) value COUNT is the number of products in each channel for whom both quantity sold and list price data are available.

```
SELECT s.channel_id, REGR_SLOPE(s.quantity_sold, p.prod_list_price) SLOPE,
  REGR_INTERCEPT(s.quantity_sold, p.prod_list_price) INTCPT,
  REGR_R2(s.quantity_sold, p.prod_list_price) RSQR,
  REGR_COUNT(s.quantity_sold, p.prod_list_price) COUNT,
  REGR_AVGX(s.quantity_sold, p.prod_list_price) AVGLISTP,
  REGR_AVGY(s.quantity_sold, p.prod_list_price) AVGQSOLD
FROM sh.sales s, sh.products p WHERE s.prod_id=p.prod_id
  AND p.prod_category LIKE 'Baseball%' AND s.time_id=to_DATE('10-OCT-2020')
GROUP BY s.channel_id;
   CHANNEL_ID    SLOPE    INTCPT    RSQR    COUNT
AVGLISTP    AVGQSOLD
_____________ ________ _________ _______ ________
_____________________________________________ ___________
            2        0         1       1      238
103.103445378151260504201680672268907563           1
            3        0         1       1      169
152.179349112426035502958579881656804734           1
            4        0         1       1       75
46.416666666666666666666666666666666666667           1
```

## 20.3.7 About Statistical Aggregates

Oracle AI Database provides a set of SQL statistical functions and a statistics package, DBMS_STAT_FUNCS. This section lists some of the new functions along with basic syntax.

> **See Also**
>
> - *Oracle Database PL/SQL Packages and Types Reference* for detailed information about the DBMS_STAT_FUNCS package
> - *Oracle Database SQL Language Reference* for detailed information about syntax and semantics

This section contains the following topics:

- Descriptive Statistics
- Hypothesis Testing - Parametric Tests
- Crosstab Statistics
- Hypothesis Testing - Non-Parametric Tests
- Non-Parametric Correlation

### 20.3.7.1 Descriptive Statistics

You can calculate the following descriptive statistics:

- Median of a Data Set

  ```
  Median (expr) [OVER (query_partition_clause)]
  ```

- Mode of a Data Set


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
STATS_MODE (expr)
```

Starting with Oracle Database 12*c* Release 2 (12.2), the approximate inverse distribution function `APPROX_MEDIAN` provides an approximate median value of the specified expression.

> **See Also**
>
> *Oracle AI Database SQL Language Reference*

## 20.3.7.2 Hypothesis Testing - Parametric Tests

You can calculate the following descriptive statistics:

- One-Sample T-Test

  ```
  STATS_T_TEST_ONE (expr1, expr2 (a constant) [, return_value])
  ```

- Paired-Samples T-Test

  ```
  STATS_T_TEST_PAIRED (expr1, expr2 [, return_value])
  ```

- Independent-Samples T-Test. Pooled Variances

  ```
  STATS_T_TEST_INDEP (expr1, expr2 [, return_value])
  ```

- Independent-Samples T-Test, Unpooled Variances

  ```
  STATS_T_TEST_INDEPU (expr1, expr2 [, return_value])
  ```

- The F-Test

  ```
  STATS_F_TEST (expr1, expr2 [, return_value])
  ```

- One-Way ANOVA

  ```
  STATS_ONE_WAY_ANOVA (expr1, expr2 [, return_value])
  ```

## 20.3.7.3 Crosstab Statistics

You can calculate crosstab statistics using the following syntax:

```
STATS_CROSSTAB (expr1, expr2 [, return_value])
```

Can return any one of the following:

- Observed value of chi-squared
- Significance of observed chi-squared
- Degree of freedom for chi-squared
- Phi coefficient, Cramer's V statistic
- Contingency coefficient
- Cohen's Kappa

## 20.3.7.4 Hypothesis Testing - Non-Parametric Tests

You can calculate hypothesis statistics using the following syntax:

```
STATS_BINOMIAL_TEST (expr1, expr2, p [, return_value])
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

- Binomial Test/Wilcoxon Signed Ranks Test

  ```
  STATS_WSR_TEST (expr1, expr2 [, return_value])
  ```

- Mann-Whitney Test

  ```
  STATS_MW_TEST (expr1, expr2 [, return_value])
  ```

- Kolmogorov-Smirnov Test

  ```
  STATS_KS_TEST (expr1, expr2 [, return_value])
  ```

### 20.3.7.5 Non-Parametric Correlation

You can calculate the following parametric statistics:

- Spearman's rho Coefficient

  ```
  CORR_S (expr1, expr2 [, return_value])
  ```

- Kendall's tau-b Coefficient

  ```
  CORR_K (expr1, expr2 [, return_value])
  ```

In addition to the functions, this release has a PL/SQL package, `DBMS_STAT_FUNCS`. It contains the descriptive statistical function `SUMMARY` along with functions to support distribution fitting. The `SUMMARY` function summarizes a numerical column of a table with a variety of descriptive statistics. The five distribution fitting functions support normal, uniform, Weibull, Poisson, and exponential distributions.

## 20.3.8 About User-Defined Aggregates

Oracle offers a facility for creating your own functions, called user-defined aggregate functions. These functions are written in programming languages such as PL/SQL, Java, and C, and can be used as analytic functions or aggregates in materialized views. See *Oracle Database Data Cartridge Developer's Guide* for further information regarding syntax and restrictions.

The advantages of these functions are:

- Highly complex functions can be programmed using a fully procedural language.
- Higher scalability than other techniques when user-defined functions are programmed for parallel processing.
- Object data types can be processed.

As a simple example of a user-defined aggregate function, consider the skew statistic. This calculation measures if a data set has a lopsided distribution about its mean. It will tell you if one tail of the distribution is significantly larger than the other. If you created a user-defined aggregate called `udskew` and applied it to the credit limit data in the prior example, the SQL statement and results might look like this:

```
SELECT USERDEF_SKEW(cust_credit_limit) FROM customers
WHERE cust_city='Marshal';

USERDEF_SKEW
============
0.583891
```

Before building user-defined aggregate functions, you should consider if your needs can be met in regular SQL. Many complex calculations are possible directly in SQL, particularly by using the `CASE` expression.


Copyright © 2001, 2025, Oracle and/or its affiliates.

Staying with regular SQL will enable simpler development, and many query operations are already well-parallelized in SQL. Even the earlier example, the skew statistic, can be created using standard, albeit lengthy, SQL.

# 20.4 Pivoting Operations

The data returned by business intelligence queries is often most usable if presented in a crosstabular format. The `pivot_clause` of the `SELECT` statement lets you write crosstabulation queries that rotate rows into columns, aggregating data in the process of the rotation. Pivoting is a key technique in data warehouses. In it, you transform multiple rows of input into fewer and generally wider rows in the data warehouse. When pivoting, an aggregation operator is applied for each item in the pivot column value list. The pivot column cannot contain an arbitrary expression. If you need to pivot on an expression, then you should alias the expression in a view before the `PIVOT` operation. The basic syntax is as follows:

```
SELECT ....
FROM <table-expr>
   PIVOT
     (
      aggregate-function(<column>) AS <alias>
      FOR <pivot-column> IN (<value1>, <value2>,..., <valuen>)
        ) AS <alias>
WHERE .....
```

See *Oracle Database SQL Language Reference* for `pivot_clause` syntax.

This section contains the following topics:

- Creating the View Used for Pivoting Examples
- Pivoting Example
- Pivoting on Multiple Columns
- Pivoting: Multiple Aggregates
- Distinguishing PIVOT-Generated Nulls from Nulls in Source Data
- Wildcard and Subquery Pivoting with XML Operations

## 20.4.1 Creating the View Used for Pivoting Examples

The pivoting and unpivoting examples in this section are based on the `sales_view` view created here.

**Example 20-26 Creating the SALES_VIEW View for Pivoting Examples**

The following example creates the `sales_view` view that is used as the basis to illustrate the use of pivoting.

```
CREATE VIEW sales_view AS
SELECT
 prod_name product, country_name country, channel_id channel,
 SUBSTR(calendar_quarter_desc, 6,2) quarter,
 SUM(amount_sold) amount_sold, SUM(quantity_sold) quantity_sold
FROM sh.sales, sh.times, sh.customers, sh.countries, sh.products
WHERE sales.time_id = times.time_id AND
  sales.prod_id = products.prod_id AND
  sales.cust_id = customers.cust_id AND
  customers.country_id = countries.country_id
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
GROUP BY prod_name, country_name, channel_id,
 SUBSTR(calendar_quarter_desc, 6, 2);
```

## 20.4.2 Pivoting Example

The following statement illustrates a typical pivot on the `channel` column of view `sales_view` created as described in Example 20-26:

```
SELECT * FROM
  (SELECT product, channel, amount_sold
   FROM sales_view
   ) S PIVOT (SUM(amount_sold)
   FOR CHANNEL IN (3 AS DIRECT_SALES, 4 AS INTERNET_SALES,
                   5 AS CATALOG_SALES, 9 AS TELESALES))
ORDER BY product;

PRODUCT                                            DIRECT_SALES    INTERNET_SALES
CATALOG_SALES    TELESALES
_________________________________________________ _______________ _______________
_______________ ____________
11" Youth Field Master Glove                          229512.97
26249.55
11.5" Youth Triple Stripe Series Glove                391230.15
53767.53
12" Premium Series Glove                               495494.7
49750.07
12.75" Premium Series Glove                           577851.94
152500.6
13" Field Master Series Glove                         286291.49
42809.44
2 Competition Grade NFHS Baseballs                    153199.63
28768.04                       5172.75
5 Point Batting Tee                                  1546466.39
196999.42
6 Gallon Empty Ball Bucket                            158223.55
102388.63
...
...
```

Note that the output has created four new aliased columns, `DIRECT_SALES`, `INTERNET_SALES`, `CATALOG_SALES`, and `TELESALES`, one for each of the pivot values. The output is a sum. If no alias is provided, the column heading will be the values of the `IN`-list.

## 20.4.3 Pivoting on Multiple Columns

You can pivot on more than one column. The following statement illustrates a typical multiple column pivot on the view `sales_view` created as described in Example 20-26:

```
SELECT *
FROM
     (SELECT product, channel, quarter, quantity_sold
      FROM sales_view
     ) PIVOT (SUM(quantity_sold)
                FOR (channel, quarter) IN
                  ((5, '02') AS CATALOG_Q2,
                   (4, '01') AS INTERNET_Q1,
                   (4, '04') AS INTERNET_Q4,
                   (2, '02') AS PARTNERS_Q2,
                   (9, '03') AS TELE_Q3
                  )
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
                );

PRODUCT                                  CATALOG_Q2    INTERNET_Q1    INTERNET_Q4
PARTNERS_Q2  TELE_Q3
_________________________________________ _____________ _____________ _____________
_____________  _______
12" Premium Series Glove                                                      197
401          494
12.75" Premium Series Glove                                                   635
898          930
Soccer Ball - Size 5                                                          279
484          872
Pro Style Batting Tee                                                         217
400          865
Match Used Autograph Racquet                                                  195
335          641
Cricket Ball - Training Ball                                                  355
423          774
...
```

Note that this example specifies a multi-column `IN`-list with column headings designed to match the `IN`-list members.

## 20.4.4 Pivoting: Multiple Aggregates

You can pivot with multiple aggregates, as shown in the following example that pivots on multiple aggregates from the `sales_view` created in Example 20-26:

```
SELECT *
FROM
     (SELECT product, channel, amount_sold, quantity_sold
      FROM sales_view
     ) PIVOT (SUM(amount_sold) AS sums,
              SUM(quantity_sold) AS sumq
              FOR channel IN (5, 4, 2, 9)
               )
ORDER BY product;
```

```
PRODUCT                                          5_SUMS     5_SUMQ        4_SUMS     4_SUMQ
2_SUMS     2_SUMQ      9_SUMS     9_SUMQ
_________________________________________ _________ _________ _____________ _________ _____________
_________ ___________ _________
11" Youth Field Master Glove                                        26249.55       861
60554.68      1886
11.5" Youth Triple Stripe Series Glove                              53767.53      1370
132582.67      3103
12" Premium Series Glove                                            49750.07      1170
100341.35      2094
12.75" Premium Series Glove                                         152500.6      2770
316811.53      5361
13" Field Master Series Glove                                       42809.44      1223
183990.15      4881
2 Competition Grade NFHS Baseballs                                  28768.04      1195
62693.53      2596     5172.75       225

...
```

Note that the query creates column headings by concatenating the pivot values, the underscore character (_), and the alias of the aggregate column. If the length of the generated


Copyright © 2001, 2025, Oracle and/or its affiliates.

column heading exceeds the maximum length of a column name, then an ORA-00918 error is returned. To avoid this error, use AS *alias* to specify a shorter column alias for the pivot column heading, the aggregate value column name, or both. "Pivoting on Multiple Columns" demonstrates using an alias for the pivot values.

> **See Also**
>
> *Oracle Database SQL Language Reference* for information about the maximum length of column names

## 20.4.5 Distinguishing PIVOT-Generated Nulls from Nulls in Source Data

**Implicit Group By in PIVOT Operations**

For tables used in PIVOT operations, columns not specified within the PIVOT clause act as hidden group-by keys. This means rows are implicitly grouped by these columns.

For example, in the following query:

```
SELECT *
FROM emp
PIVOT (SUM(sal) FOR deptno IN (10, 20));
```

Columns like ename, hiredate, mgr, and job (not part of the PIVOT clause) are treated as hidden group-by keys. The query groups rows based on these implicit keys.

**Example: Distinguishing Nulls**

You can distinguish between null values that are generated from the use of PIVOT and those that exist in the source data. The following example illustrates nulls that PIVOT generates.

The following query returns rows with five columns, column prod_id, and pivot resulting columns Q1, Q1_COUNT_TOTAL, Q2, Q2_COUNT_TOTAL. For each unique value of prod_id, Q1_COUNT_TOTAL returns the total number of rows whose qtr value is Q1, that is, and Q2_COUNT_TOTAL returns the total number of rows whose qtr value is Q2.

Assume you have a table sales2 of the following structure:

```
PROD_ID   QTR  AMOUNT_SOLD
-------   ---  -----------
100        Q1           10
100        Q1           20
100        Q2         NULL
200        Q1           50
```

Run the following query:

```
SELECT *
FROM sales2
        PIVOT
         ( SUM(amount_sold), COUNT(*) AS count_total
           FOR qtr IN ('Q1', 'Q2')
         );
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

The output is:

```
PROD_ID    "Q1"   "Q1_COUNT_TOTAL"  "Q2"       "Q2_COUNT_TOTAL"
-------    ----   ----------------  ---------  ----------------
    100      30                  2   NULL                     1
    200      50                  1   NULL                     0
```

From the result, you know that for `prod_id` 100:

- There are two sales rows for quarter `Q1`.
- There is one sales row for quarter `Q2`.

For `prod_id` 200:

- There is one sales rows for quarter `Q1`.
- No sales rows exist for quarter `Q2`.

Thus:

- The `NULL` in column `Q2` for `prod_id` 100 originates from a row in the original table where the measure value (`amount_sold`) is `NULL`.
- The `NULL` in column `Q2` for `prod_id` 200 occurs because no rows exist in the original table for this combination.

> **Note**
>
> When working with `PIVOT` operations, it is essential to understand the terms **pivot column** and **pivot value**:
>
> - Pivot Column: The column whose values become headings in the resulting table.
> - Pivot Value: The data points that populate the cells corresponding to the pivot columns in the resulting table.

## 20.4.6 Wildcard and Subquery Pivoting with XML Operations

If you want to use a wildcard argument or subquery in your pivoting columns, you can do so with `PIVOT` XML syntax. With `PIVOT` XML, the output of the operation is properly formatted XML.

The following example illustrates using the wildcard keyword, `ANY`. It outputs XML that includes all channel values in `sales_view`:

```
SELECT *
FROM
   (SELECT product, channel, quantity_sold
    FROM sales_view
    ) PIVOT XML(SUM(quantity_sold)
                FOR channel IN (ANY)
               );
```

See #unique_632/unique_632_Connect_42_CreatingTheSALES_VIEWViewForPivotin-0B828FA2 for the syntax that creates the view `sales_view`.


Copyright © 2001, 2025, Oracle and/or its affiliates.

Note that the keyword ANY is available in PIVOT operations only as part of an XML operation. This output includes data for cases where the channel exists in the data set. Also note that aggregation functions must specify a GROUP BY clause to return multiple values, yet the pivot_clause does not contain an explicit GROUP BY clause. Instead, the pivot_clause performs an implicit GROUP BY.

The following example illustrates using a subquery. It outputs XML that includes all channel values and the sales data corresponding to each channel:

```
SELECT *
FROM
     (SELECT product, channel, quantity_sold
      FROM sales_view
     ) PIVOT XML(SUM(quantity_sold)
                 FOR channel IN (SELECT DISTINCT channel_id FROM CHANNELS)
                );
```

The output densifies the data to include all possible channels for each product.

# 20.5 Unpivoting Operations

An unpivot does not reverse a PIVOT operation. Instead, it rotates data from columns into rows. If you are working with pivoted data, an UNPIVOT operation cannot reverse any aggregations that have been made by PIVOT or any other means.

To illustrate unpivoting, first create a pivoted table that includes four columns, for quarters of the year. The following command creates a table based on the view sales_view created as described in Example 20-26:

```
CREATE TABLE pivotedTable AS
SELECT *
FROM (SELECT product, quarter, quantity_sold, amount_sold
      FROM sales_view)
    PIVOT
    (
       SUM(quantity_sold) AS sumq, SUM(amount_sold) AS suma
       FOR quarter IN ('01' AS Q1, '02' AS Q2, '03' AS Q3, '04' AS Q4));
```

The table's contents resemble the following:

```
SELECT *
FROM pivotedTable
ORDER BY product;

PRODUCT                                            Q1_SUMQ       Q1_SUMA    Q2_SUMQ       Q2_SUMA
Q3_SUMQ       Q3_SUMA    Q4_SUMQ       Q4_SUMA
_____________________________________________ ____________ _____________ __________ _____________
__________ _____________ __________ _____________
11" Youth Field Master Glove                         2750      91653.75       2069      69199.25
2487      82265.49       2217      73198.71
11.5" Youth Triple Stripe Series Glove               3524     157638.52       2898     128662.03
3351     147640.32       3270     143639.48
12" Premium Series Glove                             3196     161666.11       3074     153903.54
3504     172981.83       3234     157034.64
12.75" Premium Series Glove                          4338      261078.2       2992     178748.85
5392     323570.59       4708     283766.43
13" Field Master Series Glove                        2983     114058.14       2074      80042.09
4183     161536.24       4079     157454.61
2 Competition Grade NFHS Baseballs                   3376      83585.33       1699      41743.94
2654      65205.61       2427      59299.07
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

...

The following UNPIVOT operation rotates the quarter columns into rows. For each product, there will be four rows, one for each quarter.

```
SELECT product, DECODE(quarter, 'Q1_SUMQ', 'Q1', 'Q2_SUMQ', 'Q2', 'Q3_SUMQ', 'Q3',
   'Q4_SUMQ', 'Q4') AS quarter, quantity_sold
FROM  pivotedTable
   UNPIVOT INCLUDE NULLS
       (quantity_sold
        FOR quarter IN (Q1_SUMQ, Q2_SUMQ, Q3_SUMQ, Q4_SUMQ))
ORDER BY product, quarter;

PRODUCT                          QU    QUANTITY_SOLD
-------                          --    -------------
1.44MB External 3.5" Diskette    Q1             6098
1.44MB External 3.5" Diskette    Q2             5112
1.44MB External 3.5" Diskette    Q3             6050
1.44MB External 3.5" Diskette    Q4             5848
128MB Memory Card                Q1             1963
128MB Memory Card                Q2             2361
128MB Memory Card                Q3             3069
128MB Memory Card                Q4             2832
...
```

Note the use of INCLUDE NULLS in this example. You can also use EXCLUDE NULLS, which is the default setting.

In addition, you can also unpivot using two columns, as in the following:

```
SELECT product, quarter, quantity_sold, amount_sold
FROM  pivotedTable
   UNPIVOT INCLUDE NULLS
       (
        (quantity_sold, amount_sold)
        FOR quarter IN ((Q1_SUMQ, Q1_SUMA) AS 'Q1', (Q2_SUMQ, Q2_SUMA) AS 'Q2',
(Q3_SUMQ, Q3_SUMA) AS 'Q3', (Q4_SUMQ, Q4_SUMA) AS 'Q4'))
ORDER BY product, quarter;

PRODUCT                          QU   QUANTITY_SOLD   AMOUNT_SOLD
-----------------------------    --   -------------   ------------
1.44MB External 3.5" Diskette    Q1            6098       58301.33
1.44MB External 3.5" Diskette    Q2            5112       49001.56
1.44MB External 3.5" Diskette    Q3            6050       56974.3
1.44MB External 3.5" Diskette    Q4            5848       55341.28
128MB Memory Card                Q1            1963      110763.63
128MB Memory Card                Q2            2361      132123.12
128MB Memory Card                Q3            3069      170710.4
128MB Memory Card                Q4            2832      157736.6
```

# 20.6 Data Densification for Reporting

Data is normally stored in sparse form. That is, if no value exists for a given combination of dimension values, no row exists in the fact table. However, you may want to view the data in dense form, with rows for all combination of dimension values displayed even when no fact data exist for them. For example, if a product did not sell during a particular time period, you may still want to see the product for that time period with zero sales value next to it. Moreover, time series calculations can be performed most easily when data is dense along the time dimension. This is because dense data will fill a consistent number of rows for each period,


Copyright © 2001, 2025, Oracle and/or its affiliates.

which in turn makes it simple to use the analytic windowing functions with physical offsets. Data densification is the process of converting sparse data into dense form.

To overcome the problem of sparsity, you can use a partitioned outer join to fill the gaps in a time series or any other dimension. Such a join extends the conventional outer join syntax by applying the outer join to each logical partition defined in a query. Oracle logically partitions the rows in your query based on the expression you specify in the `PARTITION BY` clause. The result of a partitioned outer join is a `UNION` of the outer joins of each of the partitions in the logically partitioned table with the table on the other side of the join.

Note that you can use this type of join to fill the gaps in any dimension, not just the time dimension. Most of the examples here focus on the time dimension because it is the dimension most frequently used as a basis for comparisons.

This section contains the following topics:

- About Partition Join Syntax
- Sample of Sparse Data
- Filling Gaps in Data
- Filling Gaps in Two Dimensions
- Filling Gaps in an Inventory Table
- Computing Data Values to Fill Gaps

## 20.6.1 About Partition Join Syntax

The syntax for partitioned outer join extends the SQL `JOIN` clause with the phrase `PARTITION BY` followed by an expression list. The expressions in the list specify the group to which the outer join is applied. The following are the two forms of syntax normally used for partitioned outer join:

```
SELECT .....
FROM table_reference
PARTITION BY (expr [, expr ]... )
RIGHT OUTER JOIN table_reference

SELECT .....
FROM table_reference
LEFT OUTER JOIN table_reference
PARTITION BY {expr [,expr ]...)
```

Note that `FULL OUTER JOIN` is not supported with a partitioned outer join.

## 20.6.2 Sample of Sparse Data

A typical situation with a sparse dimension is shown in the following example, which computes the weekly sales and year-to-date sales for the product Bounce for weeks 20-30 in 2000 and 2001:

```
SELECT SUBSTR(p.Prod_Name,1,15) Product_Name, t.Calendar_Year Year,
 t.Calendar_Week_Number Week, SUM(Amount_Sold) Sales
FROM sh.Sales s, sh.Times t, sh.Products p
WHERE s.Time_id = t.Time_id AND s.Prod_id = p.Prod_id AND
 p.Prod_name LIKE ('Tennis%') AND t.Calendar_Year IN (2020,2021) AND
 t.Calendar_Week_Number BETWEEN 20 AND 30
GROUP BY p.Prod_Name, t.Calendar_Year, t.Calendar_Week_Number;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
PRODUCT_NAME          YEAR    WEEK      SALES
____________________ _______ _______ __________
Tennis Balls He       2020      29     2072.4
Tennis Balls He       2020      30    2730.46
Tennis Balls He       2021      30    3854.76
Tennis Strings        2020      25     3051.6
Tennis Strings        2020      23    2555.41
Tennis Strings        2021      21    3776.64
Tennis Strings        2021      24    2875.29
Tennis Strings        2020      26     949.72
Tennis Strings        2020      20     849.84
Tennis Strings        2020      21     379.96
Tennis Strings        2020      22     335.62
Tennis Grip Ove       2020      29     293.64
Tennis Grip Ove       2021      27     265.08
Tennis Racquet        2021      26      63.92
...
```

In this example, you would expect 22 rows of data (11 weeks each from 2 years) if the data were dense. However, you get only 18 rows because weeks 25 and 26 are missing in 2000, and weeks 26 and 28 in 2001.

## 20.6.3 Filling Gaps in Data

You can take the sparse data of the query shown in Sample of Sparse Data and do a partitioned outer join with a dense set of time data. In the following query, you alias the original query as v and you select data from the `times` table, which you alias as `t`. Here you retrieve 22 rows because there are no gaps in the series. The four added rows each have 0 as their Sales value set to 0 by using the `NVL` function.

```
SELECT Product_Name, t.Year, t.Week, NVL(Sales,0) dense_sales
FROM
 (SELECT SUBSTR(p.Prod_Name,1,15) Product_Name,
  t.Calendar_Year Year, t.Calendar_Week_Number Week, SUM(Amount_Sold) Sales
  FROM sh.Sales s, sh.Times t, sh.Products p
  WHERE s.Time_id = t.Time_id AND s.Prod_id = p.Prod_id AND
   p.Prod_name LIKE ('Baseball%') AND t.Calendar_Year IN (2019,2021) AND
   t.Calendar_Week_Number BETWEEN 20 AND 30
  GROUP BY p.Prod_Name, t.Calendar_Year, t.Calendar_Week_Number) v
PARTITION BY (v.Product_Name)
RIGHT OUTER JOIN
 (SELECT DISTINCT Calendar_Week_Number Week, Calendar_Year Year
  FROM Times
  WHERE Calendar_Year IN (2019, 2021)
  AND Calendar_Week_Number BETWEEN 20 AND 30) t
ON (v.week = t.week AND v.Year = t.Year)
ORDER BY t.year, t.week;

PRODUCT_NAME       YEAR    WEEK    DENSE_SALES
____________________ _______ _______ _____________
Baseball Is Lif       2019      20               0
Baseball Is Lif       2019      21               0
Baseball Is Lif       2019      22               0
Baseball Is Lif       2019      23               0
Baseball Is Lif       2019      24               0
Baseball Is Lif       2019      25               0
Baseball Is Lif       2019      26               0
Baseball Is Lif       2019      27         5088.35
Baseball Is Lif       2019      28         2890.79
Baseball Is Lif       2019      29         4993.36
Baseball Is Lif       2019      30         5340.06
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
Baseball Is Lif       2021      20        8686.19
Baseball Is Lif       2021      21         6725.1
Baseball Is Lif       2021      22        7494.93
Baseball Is Lif       2021      23        1010.34
Baseball Is Lif       2021      24        6656.23
Baseball Is Lif       2021      25              0
Baseball Is Lif       2021      26        7925.89
Baseball Is Lif       2021      27        4145.25
Baseball Is Lif       2021      28         3520.6
Baseball Is Lif       2021      29        9874.68
Baseball Is Lif       2021      30        6380.98
```

Note that in this query, a `WHERE` condition was placed for weeks between 20 and 30 in the inline view for the time dimension. This was introduced to keep the result set small.

## 20.6.4 Filling Gaps in Two Dimensions

N-dimensional data is typically displayed as a dense 2-dimensional cross tab of (n - 2) page dimensions. This requires that all dimension values for the two dimensions appearing in the cross tab be filled in. The following is another example where the partitioned outer join capability can be used for filling the gaps on two dimensions:

```
WITH v1 AS
 (SELECT p.prod_id, country_id, calendar_year,
    SUM(quantity_sold) units, SUM(amount_sold) sales
  FROM sh.sales s, sh.products p, sh.customers c, sh.times t
  WHERE s.prod_id in (147, 148) AND t.time_id = s.time_id AND
    c.cust_id = s.cust_id AND p.prod_id = s.prod_id
  GROUP BY p.prod_id, country_id, calendar_year),
v2 AS                                  --countries to use for densifications
  (SELECT DISTINCT country_id
  FROM customers
  WHERE country_id IN (52782, 52785, 52786, 52787, 52788)),
v3 AS                                   --years to use for densifications
  (SELECT DISTINCT calendar_year FROM times)
SELECT v4.prod_id, v4.country_id, v3.calendar_year, units, sales
FROM
  (SELECT prod_id, v2.country_id, calendar_year, units, sales
   FROM v1 PARTITION BY (prod_id)
   RIGHT OUTER JOIN v2                  --densifies on country
   ON (v1.country_id = v2.country_id)) v4
PARTITION BY (prod_id,country_id)
RIGHT OUTER JOIN v3                     --densifies on year
ON (v4.calendar_year = v3.calendar_year)
ORDER BY 1, 2, 3;
```

In this query, the `WITH` subquery factoring clause `v1` summarizes sales data at the product, country, and year level. This result is sparse but users may want to see all the country, year combinations for each product. To achieve this, you take each partition of `v1` based on product values and outer join it on the country dimension first. This will give us all values of country for each product. You then take that result and partition it on product and country values and then outer join it on time dimension. This will give us all time values for each product and country combination.

| PROD_ID | COUNTRY_ID | CALENDAR_YEAR | UNITS | SALES |
|---|---|---|---|---|
| 147 | 52782 | 2019 | | |
| 147 | 52782 | 2020 | 29 | 209.82 |
| 147 | 52782 | 2021 | 71 | 594.36 |


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
       147         52782            2022      345    2754.42
       147         52782            2023
       147         52785            2019        1       7.99
       147         52785            2020
       147         52785            2021
       147         52785            2022
       147         52785            2023
       147         52786            2019        1       7.99
       147         52786            2020
       147         52786            2021        2      15.98
       147         52786            2022
```

## 20.6.5 Filling Gaps in an Inventory Table

An inventory table typically tracks quantity of units available for various products. This table is sparse: it only stores a row for a product when there is an event. For a sales table, the event is a sale, and for the inventory table, the event is a change in quantity available for a product. For example, consider the following inventory table:

```
CREATE TABLE invent_table (
product VARCHAR2(10),
time_id DATE,
quant NUMBER);

INSERT INTO invent_table VALUES
    ('bottle', TO_DATE('01/04/21', 'DD/MM/YY'), 10);
1 row inserted.
SQL> INSERT INTO invent_table VALUES
    ('bottle', TO_DATE('06/04/21', 'DD/MM/YY'), 8);
1 row inserted.
SQL> INSERT INTO invent_table VALUES
    ('can', TO_DATE('01/04/21', 'DD/MM/YY'), 15);
1 row inserted.
SQL> INSERT INTO invent_table VALUES
    ('can', TO_DATE('04/04/21', 'DD/MM/YY'), 11);
1 row inserted.
```

The inventory table now has the following rows:

```
PRODUCT    TIME_ID         QUANT
__________ _____________ _________
bottle     01-APR-21           10
bottle     06-APR-21            8
can        01-APR-21           15
can        04-APR-21           11
```

For reporting purposes, users may want to see this inventory data differently. For example, they may want to see all values of time for each product. This can be accomplished using partitioned outer join. In addition, for the newly inserted rows of missing time periods, users may want to see the values for quantity of units column to be carried over from the most recent existing time period. The latter can be accomplished using analytic window function `LAST_VALUE` value. Here is the query and the desired output:

```
WITH v1 AS
 (SELECT time_id
  FROM sh.times
  WHERE times.time_id BETWEEN
   TO_DATE('01/04/21', 'DD/MM/YY')
   AND TO_DATE('07/04/21', 'DD/MM/YY'))
SELECT product, time_id, quant quantity,
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
  LAST_VALUE(quant IGNORE NULLS)
    OVER (PARTITION BY product ORDER BY time_id)
    repeated_quantity
FROM
 (SELECT product, v1.time_id, quant
  FROM invent_table PARTITION BY (product)
  RIGHT OUTER JOIN v1
  ON (v1.time_id = invent_table.time_id))
ORDER BY 1, 2;
```

The inner query computes a partitioned outer join on time within each product. The inner query densifies the data on the time dimension (meaning the time dimension will now have a row for each day of the week). However, the measure column `quantity` will have nulls for the newly added rows (see the output in the column `quantity` in the following results.

The outer query uses the analytic function `LAST_VALUE`. Applying this function partitions the data by product and orders the data on the time dimension column (`time_id`). For each row, the function finds the last non-null value in the window due to the option `IGNORE NULLS`, which you can use with both `LAST_VALUE` and `FIRST_VALUE`. You see the desired output in the column `repeated_quantity` in the following output:

```
PRODUCT TIME_ID       QUANTITY    REPEATED_QUANTITY
_______ ____________ __________ ___________________
bottle  01-APR-21            10                  10
bottle  02-APR-21                                10
bottle  03-APR-21                                10
bottle  04-APR-21                                10
bottle  05-APR-21                                10
bottle  06-APR-21             8                   8
bottle  07-APR-21                                 8
can     01-APR-21            15                  15
can     02-APR-21                                15
can     03-APR-21                                15
can     04-APR-21            11                  11
can     05-APR-21                                11
can     06-APR-21                                11
can     07-APR-21                                11
```

## 20.6.6 Computing Data Values to Fill Gaps

Examples in sections Filling Gaps in Data, Filling Gaps in Two Dimensions, and Filling Gaps in an Inventory Table illustrate how to use partitioned outer join to fill gaps in one or more dimensions. However, the result sets produced by partitioned outer join have null values for columns that are not included in the `PARTITION BY` list. Typically, these are measure columns. Users can make use of analytic SQL functions to replace those null values with a non-null value.

For example, the following query computes monthly totals for products 64MB Memory card and DVD-R Discs (product IDs 122 and 136) for the year 2000. It uses partitioned outer join to densify data for all months. For the missing months, it then uses the analytic SQL function `AVG` to compute the sales and units to be the average of the months when the product was sold.

If working in SQL*Plus, the following two commands wraps the column headings for greater readability of results:

```
col computed_units  heading  'Computed|_units'
col computed_sales  heading  'Computed|_sales'

WITH V AS
 (SELECT substr(p.prod_name,1,12) prod_name, calendar_month_desc,
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
    SUM(quantity_sold) units, SUM(amount_sold) sales
   FROM sh.sales s, sh.products p, sh.times t
   WHERE s.prod_id IN (122,136) AND calendar_year = 2020
     AND t.time_id = s.time_id
     AND p.prod_id = s.prod_id
   GROUP BY p.prod_name, calendar_month_desc)
SELECT v.prod_name, calendar_month_desc, units, sales,
  NVL(units, AVG(units) OVER (PARTITION BY v.prod_name)) computed_units,
  NVL(sales, AVG(sales) OVER (PARTITION BY v.prod_name)) computed_sales
FROM
  (SELECT DISTINCT calendar_month_desc
   FROM times
   WHERE calendar_year = 2020) t
   LEFT OUTER JOIN V
   PARTITION BY (prod_name)
   USING (calendar_month_desc);
```

| PROD_NAME | CALENDAR_MONTH_DESC | UNITS | SALES | COMPUTED_UNITS | COMPUTED_SALES |
|---|---|---|---|---|---|
| Soccer Jerse | 2020-01 | | | 27 | 940.92 |
| Soccer Jerse | 2020-02 | | | 27 | 940.92 |
| Soccer Jerse | 2020-03 | | | 27 | 940.92 |
| Soccer Jerse | 2020-04 | | | 27 | 940.92 |
| Soccer Jerse | 2020-05 | | | 27 | 940.92 |
| Soccer Jerse | 2020-06 | | | 27 | 940.92 |
| Soccer Jerse | 2020-07 | | | 27 | 940.92 |
| Soccer Jerse | 2020-08 | | | 27 | 940.92 |
| Soccer Jerse | 2020-09 | | | 27 | 940.92 |
| Soccer Jerse | 2020-10 | 34 | 1206.75 | 34 | 1206.75 |
| Soccer Jerse | 2020-11 | 41 | 1425.49 | 41 | 1425.49 |
| Soccer Jerse | 2020-12 | 6 | 190.52 | 6 | 190.52 |
| Wide Brim Ha | 2020-01 | 30 | 631.17 | 30 | 631.17 |
| Wide Brim Ha | 2020-02 | 25 | 484.22 | 25 | 484.22 |

...

# 20.7 Time Series Calculations on Densified Data

Densification is not just for reporting purpose. It also enables certain types of calculations, especially, time series calculations. Time series calculations are easier when data is dense along the time dimension. Dense data has a consistent number of rows for each time periods which in turn make it simple to use analytic window functions with physical offsets.

To illustrate, let us first take the example on "Filling Gaps in Data", and let's add an analytic function to that query. In the following enhanced version, you calculate weekly year-to-date sales alongside the weekly sales. The `NULL` values that the partitioned outer join inserts in making the time series dense are handled in the usual way: the `SUM` function treats them as 0's.

```
SELECT Product_Name, t.Year, t.Week, NVL(Sales,0) Current_sales,
 SUM(Sales)
   OVER (PARTITION BY Product_Name, t.year ORDER BY t.week) Cumulative_sales
FROM
 (SELECT SUBSTR(p.Prod_Name,1,15) Product_Name, t.Calendar_Year Year,
   t.Calendar_Week_Number Week, SUM(Amount_Sold) Sales
  FROM sh.Sales s, sh.Times t, sh.Products p
  WHERE s.Time_id = t.Time_id AND
   s.Prod_id = p.Prod_id AND p.Prod_name IN ('Bounce') AND
   t.Calendar_Year IN (2020,2021) AND
   t.Calendar_Week_Number BETWEEN 20 AND 30
  GROUP BY p.Prod_Name, t.Calendar_Year, t.Calendar_Week_Number) v
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
PARTITION BY (v.Product_Name)
RIGHT OUTER JOIN
(SELECT DISTINCT
 Calendar_Week_Number Week, Calendar_Year Year
 FROM Times
 WHERE Calendar_Year in (2020, 2021)
 AND Calendar_Week_Number BETWEEN 20 AND 30) t
ON (v.week = t.week AND v.Year = t.Year)
ORDER BY t.year, t.week;

PRODUCT_NAME          YEAR    WEEK    CURRENT_SALES    CUMULATIVE_SALES
_____________________ _______ _______ ________________ ________________
Tennis Balls 12       2020      20          2222.52             2222.52
Tennis Balls He       2020      20                0
Tennis Grip Ove       2020      20                0
Tennis Racquet        2020      20                0
Tennis Strings        2020      20           1775.4             6623.44
Tennis Strings        2020      20           985.45             6623.44
Tennis Strings        2020      20          3012.75             6623.44
Tennis Strings        2020      20           849.84             6623.44
Tennis Balls 12       2020      21          1468.75             3691.27
Tennis Balls He       2020      21           248.78              248.78
Tennis Grip Ove       2020      21             20.7                20.7
Tennis Racquet        2020      21           113.76              113.76
Tennis Strings        2020      21          1043.74             10609.1
Tennis Strings        2020      21          2057.76             10609.1
Tennis Strings        2020      21            504.2             10609.1
Tennis Strings        2020      21           379.96             10609.1
Tennis Balls 12       2020      22             1270             4961.27
Tennis Balls He       2020      22                0              248.78
Tennis Grip Ove       2020      22                0                20.7
```

This section contains the following topics:

- Period-to-Period Comparison for One Time Level: Example
- Period-to-Period Comparison for Multiple Time Levels: Example
- Creating a Custom Member in a Dimension: Example

## 20.7.1 Period-to-Period Comparison for One Time Level: Example

How do you use this feature to compare values across time periods? Specifically, how do you calculate a year-over-year sales comparison at the week level? The following query returns on the same row, for each product, the year-to-date sales for each week of 2021 with that of 2020.

Note that in this example you start with a `WITH` clause. This improves readability of the query and lets us focus on the partitioned outer join. If working in SQL*Plus, the following command wraps the column headings for greater readability of results:

```
col Weekly_ytd_sales_prior_year  heading  'Weekly_ytd|_sales_|prior_year'

WITH v AS
  (SELECT SUBSTR(p.Prod_Name,1,6) Prod, t.Calendar_Year Year,
     t.Calendar_Week_Number Week, SUM(Amount_Sold) Sales
   FROM sh.Sales s, sh.Times t, sh.Products p
   WHERE s.Time_id = t.Time_id AND
     s.Prod_id = p.Prod_id AND p.Prod_name in ('Y Box') AND
     t.Calendar_Year in (2020,2021) AND
     t.Calendar_Week_Number BETWEEN 30 AND 40
   GROUP BY p.Prod_Name, t.Calendar_Year, t.Calendar_Week_Number)
SELECT Prod , Year, Week, Sales,
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
  Weekly_ytd_sales, Weekly_ytd_sales_prior_year
FROM
  (SELECT Prod, Year, Week, Sales, Weekly_ytd_sales,
     LAG(Weekly_ytd_sales, 1) OVER
       (PARTITION BY Prod , Week ORDER BY Year) Weekly_ytd_sales_prior_year
   FROM
     (SELECT v.Prod  Prod , t.Year Year, t.Week Week,
        NVL(v.Sales,0) Sales, SUM(NVL(v.Sales,0)) OVER
          (PARTITION BY v.Prod , t.Year ORDER BY t.week) weekly_ytd_sales
      FROM v
      PARTITION BY (v.Prod )
      RIGHT OUTER JOIN
        (SELECT DISTINCT Calendar_Week_Number Week, Calendar_Year Year
         FROM Times
         WHERE Calendar_Year IN (2020, 2021)) t
      ON (v.week = t.week AND v.Year = t.Year)
     ) dense_sales
  ) year_over_year_sales
WHERE Year = 2021 AND Week BETWEEN 30 AND 40
ORDER BY 1, 2, 3;
```

In the `FROM` clause of the inline view `dense_sales`, you use a partitioned outer join of aggregate view `v` and time view `t` to fill gaps in the sales data along the time dimension. The output of the partitioned outer join is then processed by the analytic function `SUM ... OVER` to compute the weekly year-to-date sales (the `weekly_ytd_sales` column). Thus, the view `dense_sales` computes the year-to-date sales data for each week, including those missing in the aggregate view `s`. The inline view `year_over_year_sales` then computes the year ago weekly year-to-date sales using the `LAG` function. The `LAG` function labeled `weekly_ytd_sales_prior_year` specifies a `PARTITION BY` clause that pairs rows for the same week of years 2000 and 2001 into a single partition. You then pass an offset of 1 to the `LAG` function to get the weekly year to date sales for the prior year.The outermost query block selects data from `year_over_year_sales` with the condition `yr = 2001`, and thus the query returns, for each product, its weekly year-to-date sales in the specified weeks of years 2001 and 2000.

## 20.7.2 Period-to-Period Comparison for Multiple Time Levels: Example

While the prior example shows us a way to create comparisons for a single time level, it would be even more useful to handle multiple time levels in a single query. For example, you could compare sales versus the prior period at the year, quarter, month and day levels. How can you create a query which performs a year-over-year comparison of year-to-date sales for all levels of our time hierarchy?

You will take several steps to perform this task. The goal is a single query with comparisons at the day, week, month, quarter, and year level. The steps are as follows:

1. Create a view called `cube_prod_time`, which holds a hierarchical cube of sales aggregated across `times` and `products`.

   See "Create the Hierarchical Cube View".

2. Create a view of the time dimension to use as an edge of the cube. The time edge, which holds a complete set of dates, will be partitioned outer joined to the sparse data in the view `cube_prod_time`.

   See "Create the View edge_time, which is a Complete Set of Date Values".

3. Finally, for maximum performance, create a materialized view, `mv_prod_time`, built using the same definition as `cube_prod_time`.


Copyright © 2001, 2025, Oracle and/or its affiliates.

See "Create the Materialized View mv_prod_time to Support Faster Performance".

4. Create the comparison query.

   See "Create the Comparison Query".

For more information regarding hierarchical cubes, see SQL for Aggregation in Data Warehouses. The materialized view is defined in the following section.

### Create the Hierarchical Cube View

The materialized view shown in the following may already exist in your system; if not, create it now. If you must generate it, note that you limit the query to just two products to keep processing time short:

```
CREATE OR REPLACE VIEW cube_prod_time AS
SELECT
  (CASE
     WHEN ((GROUPING(calendar_year)=0 )
       AND (GROUPING(calendar_quarter_desc)=1 ))
       THEN (TO_CHAR(calendar_year) || '_0')
     WHEN ((GROUPING(calendar_quarter_desc)=0 )
       AND (GROUPING(calendar_month_desc)=1 ))
       THEN (TO_CHAR(calendar_quarter_desc) || '_1')
     WHEN ((GROUPING(calendar_month_desc)=0 )
       AND (GROUPING(t.time_id)=1 ))
       THEN (TO_CHAR(calendar_month_desc) || '_2')
     ELSE (TO_CHAR(t.time_id) || '_3')
  END) Hierarchical_Time,
  calendar_year year, calendar_quarter_desc quarter,
  calendar_month_desc month, t.time_id day,
  prod_category cat, prod_subcategory subcat, p.prod_id prod,
  GROUPING_ID(prod_category, prod_subcategory, p.prod_id,
    calendar_year, calendar_quarter_desc, calendar_month_desc,t.time_id) gid,
  GROUPING_ID(prod_category, prod_subcategory, p.prod_id) gid_p,
  GROUPING_ID(calendar_year, calendar_quarter_desc,
    calendar_month_desc, t.time_id) gid_t,
  SUM(amount_sold) s_sold, COUNT(amount_sold) c_sold, COUNT(*) cnt
FROM SALES s, TIMES t, PRODUCTS p
WHERE s.time_id = t.time_id AND
  p.prod_name IN ('Bounce', 'Y Box') AND s.prod_id = p.prod_id
GROUP BY
  ROLLUP(calendar_year, calendar_quarter_desc, calendar_month_desc, t.time_id),
  ROLLUP(prod_category, prod_subcategory, p.prod_id);
```

Because this view is limited to two products, it returns just over 2200 rows. Note that the column `Hierarchical_Time` contains string representations of time from all levels of the time hierarchy. The `CASE` expression used for the `Hierarchical_Time` column appends a marker (_0, _1, ...) to each date string to denote the time level of the value. A _0 represents the year level, _1 is quarters, _2 is months, and _3 is day. Note that the `GROUP BY` clause is a concatenated `ROLLUP` which specifies the rollup hierarchy for the time and product dimensions. The `GROUP BY` clause is what determines the hierarchical cube contents.

### Create the View edge_time, which is a Complete Set of Date Values

`edge_time` is the source for filling time gaps in the hierarchical cube using a partitioned outer join. The column `Hierarchical_Time` in `edge_time` will be used in a partitioned join with the `Hierarchical_Time` column in the view `cube_prod_time`. The following statement defines `edge_time`:

```
CREATE OR REPLACE VIEW edge_time AS
SELECT
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
  (CASE
     WHEN ((GROUPING(calendar_year)=0 )
       AND (GROUPING(calendar_quarter_desc)=1 ))
       THEN (TO_CHAR(calendar_year) || '_0')
     WHEN ((GROUPING(calendar_quarter_desc)=0 )
       AND (GROUPING(calendar_month_desc)=1 ))
       THEN (TO_CHAR(calendar_quarter_desc) || '_1')
     WHEN ((GROUPING(calendar_month_desc)=0 )
       AND (GROUPING(time_id)=1 ))
       THEN (TO_CHAR(calendar_month_desc) || '_2')
     ELSE (TO_CHAR(time_id) || '_3')
   END) Hierarchical_Time,
   calendar_year yr, calendar_quarter_number qtr_num,
   calendar_quarter_desc qtr, calendar_month_number mon_num,
   calendar_month_desc mon, time_id - TRUNC(time_id, 'YEAR') + 1 day_num,
   time_id day,
GROUPING_ID(calendar_year, calendar_quarter_desc,
  calendar_month_desc, time_id) gid_t
FROM TIMES
GROUP BY ROLLUP
 (calendar_year, (calendar_quarter_desc, calendar_quarter_number),
 (calendar_month_desc, calendar_month_number), time_id);
```

### Create the Materialized View mv_prod_time to Support Faster Performance

The materialized view definition is a duplicate of the view `cube_prod_time` defined earlier. Because it is a duplicate query, references to `cube_prod_time` will be rewritten to use the `mv_prod_time` materialized view. The following materialized may already exist in your system; if not, create it now. If you must generate it, note that you limit the query to just two products to keep processing time short.

```
CREATE MATERIALIZED VIEW mv_prod_time
REFRESH COMPLETE ON DEMAND AS
SELECT
  (CASE
     WHEN ((GROUPING(calendar_year)=0 )
       AND (GROUPING(calendar_quarter_desc)=1 ))
       THEN (TO_CHAR(calendar_year) || '_0')
     WHEN ((GROUPING(calendar_quarter_desc)=0 )
       AND (GROUPING(calendar_month_desc)=1 ))
       THEN (TO_CHAR(calendar_quarter_desc) || '_1')
     WHEN ((GROUPING(calendar_month_desc)=0 )
       AND (GROUPING(t.time_id)=1 ))
       THEN (TO_CHAR(calendar_month_desc) || '_2')
     ELSE (TO_CHAR(t.time_id) || '_3')
  END) Hierarchical_Time,
  calendar_year year, calendar_quarter_desc quarter,
  calendar_month_desc month, t.time_id day,
  prod_category cat, prod_subcategory subcat, p.prod_id prod,
  GROUPING_ID(prod_category, prod_subcategory, p.prod_id,
    calendar_year, calendar_quarter_desc, calendar_month_desc,t.time_id) gid,
  GROUPING_ID(prod_category, prod_subcategory, p.prod_id) gid_p,
  GROUPING_ID(calendar_year, calendar_quarter_desc,
    calendar_month_desc, t.time_id) gid_t,
  SUM(amount_sold) s_sold, COUNT(amount_sold) c_sold, COUNT(*) cnt
FROM SALES s, TIMES t, PRODUCTS p
WHERE s.time_id = t.time_id AND
  p.prod_name IN ('Bounce', 'Y Box') AND s.prod_id = p.prod_id
GROUP BY
  ROLLUP(calendar_year, calendar_quarter_desc, calendar_month_desc, t.time_id),
  ROLLUP(prod_category, prod_subcategory, p.prod_id);
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

### Create the Comparison Query

You have now set the stage for our comparison query. You can obtain period-to-period comparison calculations at all time levels. It requires applying analytic functions to a hierarchical cube with dense data along the time dimension.

Some of the calculations you can achieve for each time level are:

- Sum of sales for prior period at all levels of time.
- Variance in sales over prior period.
- Sum of sales in the same period a year ago at all levels of time.
- Variance in sales over the same period last year.

The following example performs all four of these calculations. It uses a partitioned outer join of the views `cube_prod_time` and `edge_time` to create an inline view of dense data called `dense_cube_prod_time`. The query then uses the `LAG` function in the same way as the prior single-level example. The outer `WHERE` clause specifies time at three levels: the days of August 2001, the entire month, and the entire third quarter of 2001. Note that the last two rows of the results contain the month level and quarter level aggregations.Note that to make the results easier to read if you are using SQL*Plus, the column headings should be adjusted with the following commands. The commands will fold the column headings to reduce line length:

```
col sales_prior_period heading 'sales_prior|_period'
col variance_prior_period heading 'variance|_prior|_period'
col sales_same_period_prior_year heading 'sales_same|_period_prior|_year'
col variance_same_period_p_year heading 'variance|_same_period|_prior_year'
```

Here is the query comparing current sales to prior and year ago sales:

```
SELECT SUBSTR(prod,1,4) prod, SUBSTR(Hierarchical_Time,1,12) ht,
  sales, sales_prior_period,
  sales - sales_prior_period variance_prior_period,
  sales_same_period_prior_year,
  sales - sales_same_period_prior_year variance_same_period_p_year
FROM
 (SELECT cat, subcat, prod, gid_p, gid_t,
    Hierarchical_Time, yr, qtr, mon, day, sales,
    LAG(sales, 1) OVER (PARTITION BY gid_p, cat, subcat, prod,
      gid_t ORDER BY yr, qtr, mon, day)
      sales_prior_period,
    LAG(sales, 1) OVER (PARTITION BY gid_p, cat, subcat, prod,
      gid_t, qtr_num, mon_num, day_num ORDER BY yr)
      sales_same_period_prior_year
  FROM
   (SELECT c.gid, c.cat, c.subcat, c.prod, c.gid_p,
      t.gid_t, t.yr, t.qtr, t.qtr_num, t.mon, t.mon_num,
      t.day, t.day_num, t.Hierarchical_Time, NVL(s_sold,0) sales
    FROM cube_prod_time c
    PARTITION BY (gid_p, cat, subcat, prod)
    RIGHT OUTER JOIN edge_time t
    ON ( c.gid_t = t.gid_t AND
      c.Hierarchical_Time = t.Hierarchical_Time)
   ) dense_cube_prod_time
 )                        --side by side current and prior year sales
WHERE prod IN (139) AND gid_p=0 AND    --1 product and product level data
  ( (mon IN ('2001-08' ) AND gid_t IN (0, 1)) OR --day and month data
  (qtr IN ('2001-03' ) AND gid_t IN (3)))        --quarter level data
ORDER BY day;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

| PROD | HT | SALES | sales_prior _period | variance _prior _period | sales_same _period_prior _year | variance _same_period _prior_year |
|---|---|---|---|---|---|---|
| 139 | 01-AUG-01_3 | 0 | 0 | 0 | 0 | 0 |
| 139 | 02-AUG-01_3 | 1347.53 | 0 | 1347.53 | 0 | 1347.53 |
| 139 | 03-AUG-01_3 | 0 | 1347.53 | -1347.53 | 42.36 | -42.36 |
| 139 | 04-AUG-01_3 | 57.83 | 0 | 57.83 | 995.75 | -937.92 |
| 139 | 05-AUG-01_3 | 0 | 57.83 | -57.83 | 0 | 0 |
| 139 | 06-AUG-01_3 | 0 | 0 | 0 | 0 | 0 |
| 139 | 07-AUG-01_3 | 134.81 | 0 | 134.81 | 880.27 | -745.46 |
| 139 | 08-AUG-01_3 | 1289.89 | 134.81 | 1155.08 | 0 | 1289.89 |
| 139 | 09-AUG-01_3 | 0 | 1289.89 | -1289.89 | 0 | 0 |
| 139 | 10-AUG-01_3 | 0 | 0 | 0 | 0 | 0 |
| 139 | 11-AUG-01_3 | 0 | 0 | 0 | 0 | 0 |
| 139 | 12-AUG-01_3 | 0 | 0 | 0 | 0 | 0 |
| 139 | 13-AUG-01_3 | 0 | 0 | 0 | 0 | 0 |
| 139 | 14-AUG-01_3 | 0 | 0 | 0 | 0 | 0 |
| 139 | 15-AUG-01_3 | 38.49 | 0 | 38.49 | 1104.55 | -1066.06 |
| 139 | 16-AUG-01_3 | 0 | 38.49 | -38.49 | 0 | 0 |
| 139 | 17-AUG-01_3 | 77.17 | 0 | 77.17 | 1052.03 | -974.86 |
| 139 | 18-AUG-01_3 | 2467.54 | 77.17 | 2390.37 | 0 | 2467.54 |
| 139 | 19-AUG-01_3 | 0 | 2467.54 | -2467.54 | 127.08 | -127.08 |
| 139 | 20-AUG-01_3 | 0 | 0 | 0 | 0 | 0 |
| 139 | 21-AUG-01_3 | 0 | 0 | 0 | 0 | 0 |
| 139 | 22-AUG-01_3 | 0 | 0 | 0 | 0 | 0 |
| 139 | 23-AUG-01_3 | 1371.43 | 0 | 1371.43 | 0 | 1371.43 |
| 139 | 24-AUG-01_3 | 153.96 | 1371.43 | -1217.47 | 2091.3 | -1937.34 |
| 139 | 25-AUG-01_3 | 0 | 153.96 | -153.96 | 0 | 0 |
| 139 | 26-AUG-01_3 | 0 | 0 | 0 | 0 | 0 |
| 139 | 27-AUG-01_3 | 1235.48 | 0 | 1235.48 | 0 | 1235.48 |
| 139 | 28-AUG-01_3 | 173.3 | 1235.48 | -1062.18 | 2075.64 | -1902.34 |
| 139 | 29-AUG-01_3 | 0 | 173.3 | -173.3 | 0 | 0 |
| 139 | 30-AUG-01_3 | 0 | 0 | 0 | 0 | 0 |
| 139 | 31-AUG-01_3 | 0 | 0 | 0 | 0 | 0 |
| 139 | 2001-08_2 | 8347.43 | 7213.21 | 1134.22 | 8368.98 | -21.55 |
| 139 | 2001-03_1 | 24356.8 | 28862.14 | -4505.34 | 24168.99 | 187.81 |

The first `LAG` function (`sales_prior_period`) partitions the data on `gid_p`, `cat`, `subcat`, `prod`, `gid_t` and orders the rows on all the time dimension columns. It gets the sales value of the prior period by passing an offset of 1. The second `LAG` function (`sales_same_period_prior_year`) partitions the data on additional columns `qtr_num`, `mon_num`, and `day_num` and orders it on `yr` so that, with an offset of 1, it can compute the year ago sales for the same period. The outermost `SELECT` clause computes the variances.

## 20.7.3 Creating a Custom Member in a Dimension: Example

In many analytical SQL tasks, it is helpful to define custom members in a dimension. For instance, you might define a specialized time period for analyses. You can use a partitioned outer join to temporarily add a member to a dimension. Note that the new SQL `MODEL` clause is suitable for creating more complex scenarios involving new members in dimensions. See SQL for Modeling for more information on this topic.

As an example of a task, what if you want to define a new member for the `time` dimension? You want to create a 13th member of the Month level in the `time` dimension. This 13th month is defined as the summation of the sales for each product in the first month of each quarter of year 2021.

The solution has two steps. Note that you will build this solution using the views and tables created in the prior example. Two steps are required. First, create a view with the new member


Copyright © 2001, 2025, Oracle and/or its affiliates.

added to the appropriate dimension. The view uses a UNION ALL operation to add the new member. To query using the custom member, use a CASE expression and a partitioned outer join.

Our new member for the time dimension is created with the following view:

```
CREATE OR REPLACE VIEW time_c AS
(SELECT * FROM edge_time
UNION ALL
SELECT '2021-13_2', 2021, 5, '2021-05', 13, '2021-13', null, null,
8 -- <gid_of_mon>;
```

In this statement, the view time_c is defined by performing a UNION ALL of the edge_time view (defined in the prior example) and the user-defined 13th month. The gid_t value of 8 was chosen to differentiate the custom member from the standard members. The UNION ALL specifies the attributes for a 13th month member by doing a SELECT from the DUAL table. Note that the grouping id, column gid_t, is set to 8, and the quarter number is set to 5.

Then, the second step is to use an inline view of the query to perform a partitioned outer join of cube_prod_time with time_c. This step creates sales data for the 13th month at each level of product aggregation. In the main query, the analytic function SUM is used with a CASE expression to compute the 13th month, which is defined as the summation of the first month's sales of each quarter.

```
SELECT * FROM  (SELECT SUBSTR(cat,1,12) cat, SUBSTR(subcat,1,12) subcat,
     prod, mon, mon_num,
     SUM(CASE WHEN mon_num IN (1, 4, 7, 10)
           THEN s_sold
           ELSE NULL
         END)
       OVER (PARTITION BY gid_p, prod, subcat, cat, yr) sales_month_13
   FROM
     (SELECT c.gid, c.prod, c.subcat, c.cat, gid_p,
        t.gid_t, t.day, t.mon, t.mon_num,
        t.qtr, t.yr, NVL(s_sold,0) s_sold
      FROM cube_prod_time c
      PARTITION BY (gid_p, prod, subcat, cat)
      RIGHT OUTER JOIN time_c t
      ON (c.gid_t = t.gid_t AND
        c.Hierarchical_Time = t.Hierarchical_Time)
     )
  )
WHERE mon_num=13;
```

The SUM function uses a CASE to limit the data to months 1, 4, 7, and 10 within each year. Due to the tiny data set, with just two products, the rollup values of the results are necessarily repetitions of lower level aggregations.

# 20.8 Miscellaneous Analysis and Reporting Capabilities

This section illustrates the following additional analytic capabilities:

- TIME_BUCKET Function
- WIDTH_BUCKET Function
- Linear Algebra
- CASE Expressions


Copyright © 2001, 2025, Oracle and/or its affiliates.

- Frequent Itemsets in SQL Analytics

## 20.8.1 TIME_BUCKET Function

Time bucketing is a common operation when processing time series or event streaming data where a series of data points within an arbitrarily defined time window need to be mapped to a specific fixed time interval (bucket) for aggregated analysis.

The TIME_BUCKET function supports bucketing the input DATETIMES into the specified interval aligned with a specific origin, and returns the START of the bucket into which time is placed.

TIME_BUCKET is directly supported for the following input DATETIMES:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE
- TIMESTAMP WITH LOCAL TIME ZONE
- NUMBER(EPOCH TIME)

TIME_BUCKET also accepts the following DATETIMES as inputs, but these binary datatypes and the CHAR datatype are implicitly converted as shown in Table 20-3.

**Table 20-3 IMPLICIT TIME_BUCKET CONVERSIONS FOR SOME DATETIMES**

| DATETIME | CONVERTED TO... |
|---|---|
| BINARY_FLOAT | NUMBER |
| BINARY_DOUBLE | NUMBER |
| CHAR | TIMESTAMP |

### 20.8.1.1 TIME_BUCKET Syntax

**Syntax**

```
TIME_BUCKET(DATETIME, STRIDE, ORIGIN[, START_OR_END]
[TIMEBUCKET_OPTIONAL_CLAUSE])
```

**Table 20-4 TIME_BUCKET Inputs**

| Input | Description |
|---|---|
| DATETIME | DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE, EPOCH TIME, BINARY_FLOAT, BINARY_DOUBLE, CHAR, an expression or a bind variable.<br>There are implicit conversions for DATETIME. A BINARY_FLOAT or BINARY_DOUBLE is converted to NUMBER. Note that precision here is the user's responsibility. A CHAR is converted to TIMESTAMP.<br>CHAR should match session nls_timestamp_format. Otherwise, an error is raised. Currently, fractional second is supported only if DATETIME and ORIGIN are EPOCH TIME, BINARY_FLOAT or BINARY_DOUBLE. |


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Table 20-4 (Cont.) TIME_BUCKET Inputs**

| Input | Description |
|---|---|
| STRIDE | The time interval – a positive Oracle INTERVAL, ISO 8601 time interval string, an expression or a bind variable.<br>For Oracle INTERVAL, the two types of valid intervals are INTERVAL YEAR TO MONTH, and INTERVAL DAY TO SECOND. For ISO 8601 time interval string, years, months, days, hours, minutes and seconds are integers between 0 and 999999999.<br>FRAC_SECS is the fractional part of seconds between .0 and .999999999. If year or month is specified, then all other units specified are ignored. Currently, fractional second is supported only if DATETIME and ORIGIN are EPOCH TIME, BINARY_FLOAT or BINARY_DOUBLE.<br>Oracle AI Database provides two kinds of INTERVAL: INTERVAL YEAR TO MONTH and INTERVAL DAY TO SECOND. The ISO 8601 time interval string used should match the definition of Oracle INTERVAL.<br>These are the requirements for the INTERVAL string:<br>• No blanks are allowed in the string.<br>• 'P' is required .<br>• If you specify 'T', then you must specify at least one of the hours, minutes, or seconds values.<br>Note that hours are based on 24-hour time. For example, 'P100DT05H' indicates 100 days and 5 hours. 'P1Y2M' indicates 1 year and 2 months. 'P1M1DT5H30M30S' is equivalent to 'P1M' which indicates 1 month.<br>Only positive values are allowed for STRIDE. |
| ORIGIN | The anchor to which all buckets are aligned. ORIGIN supports the same DATETIMES as the DATETIME argument. Note that the ORIGIN datatype must match the datatype of the datetime.<br>There are implicit conversions for ORIGIN. A BINARY_FLOAT or BINARY_DOUBLE, is converted to NUMBER implicitly.<br>Notice that precision is the user's responsibility. A CHAR is converted to TIMESTAMP implicitly. Note that the CHAR should match session nls_timestamp_format. Otherwise, an error is raised. Currently, fractional second is supported only if DATETIME and ORIGIN are EPOCH TIME, BINARY_FLOAT or BINARY_DOUBLE. |
| START_OR_END | The start or end of the time bucket to be returned. The value is either START or END. The default is START. This argument is optional. |
| TIMEBUCKET_OPTIONAL_CLAUSE | Controls how the function behaves when the return value is an invalid date, or when the ORIGIN is the last day of a month and the STRIDE contains only MONTH and/or YEAR. The possible values are:<br>• ON OVERFLOW ROUND (default): If the return value is an invalid date, then the function returns a value whose date component is the last day of the month.<br>• ON OVERFLOW ERROR: If the return value is an invalid date, then the function returns an error.<br>• LAST DAY OF MONTH: If the ORIGIN is the last day of a month and the STRIDE contains only the MONTH or/and YEAR, then the function always returns a value whose date component is the last day of the month.<br>This clause is optional. |

## 20.8.1.2 TIME_BUCKET Examples

Here are TIME_BUCKET examples using different datetypes.


Copyright © 2001, 2025, Oracle and/or its affiliates.

Assume that all of the examples are run in the same session.

```
SQL> alter session set nls_date_format='dd mon syyyy hh24:mi:ss';

Session altered.
```

The three examples in this set use the datatype `DATE`. We select a five-year window with Oracle `INTERVAL` syntax and then with ISO 8601 syntax. Those first two examples use the `START` argument, which defines the starting point of a window as return value. The third example uses the same statement, but with the `END` argument, which returns the end of a window.

```
SQL> select time_bucket(DATE '2022-06-29', 'P5Y', DATE '2000-01-01', START) ;

TIME_BUCKET(DATE'2022

--------------------

01 jan  2020
00:00:00

SQL> select time_bucket(DATE '2022-06-29', INTERVAL '5' YEAR, DATE
'2000-01-01', START) ;

TIME_BUCKET(DATE'2022

--------------------

01 jan  2020
00:00:00

SQL> select time_bucket(DATE '2022-06-29', INTERVAL '5' YEAR, DATE
'2000-01-01', END) ;

TIME_BUCKET(DATE'2022

--------------------

01 jan  2025
00:00:00
```

The following example uses the `TIMESTAMP` datatype. Note that `DATETIME` and `ORIGIN` must be of the same datatype, as shown here.

```
SQL> select time_bucket(TIMESTAMP '2022-06-29 12:34:56', INTERVAL '5' HOUR,
TIMESTAMP '2022-06-29 00:00:00', END);

TIME_BUCKET(TIMESTAMP'2022-06-2912:34:56',INTERVAL'5'HOUR,TIMESTAMP'2022-06

--------------------------------------------------------------------------

2022-06-29
15:00:00
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

The example below uses Epoch time. The INTERVAL is in fractions of seconds. The alignment is the default – START.

```
SQL> select time_bucket(1234.56, INTERVAL '2.555' SECOND, 1000.6);

TIME_BUCKET(1234.56,INTERVAL'2.555'SECOND,1000.6)

-----------------------------------------------


1233.105
```

The following is a BINARY_FLOAT example, which returns a number.

```
SQL> select time_bucket(TO_BINARY_FLOAT(52.35), 'PT30S', TO_BINARY_FLOAT(10),
START);

TIME_BUCKET(TO_BINARY_FLOAT(52.35),'PT30S',TO_BINARY_FLOAT(10),START)

-------------------------------------------------------------------


40
```

This is a CHAR datatype example. Note that the input is a character in NLS_TIMESTAMP_FORMAT and is automatically converted to TIMESTAMP. This is why the second argument ORIGIN can be a timestamp.

```
SQL> alter session set nls_timestamp_format='yyyy-mm-dd hh24:mi:ss';

Session altered.

SQL> select time_bucket('2022-06-27 12:34:56', 'P5D', TIMESTAMP '2022-06-29
12:14:56');

TIME_BUCKET('2022-06-2712:34:56','P5D',TIMESTAMP'2022-06-2912:14:56')

---------------------------------------------------------------------------

2022-06-24
12:14:56
```

This is a simple example of how to bucket DATE data.

```
SQL> select time_bucket(hiredate, 'P1Y', DATE '1970-01-01') year_of_hire,
count(empno) num_of_hires
from emp
group by time_bucket(hiredate, 'P1Y', DATE '1970-01-01');

YEAR_OF_HIRE
NUM_OF_HIRES
--------------------
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
------------
01 jan  1980 00:00:00
1
01 jan  1981 00:00:00
10
01 jan  1987 00:00:00
2
01 jan  1982 00:00:00
1
```

## 20.8.2 WIDTH_BUCKET Function

For a given expression, the `WIDTH_BUCKET` function returns the bucket number that the result of this expression will be assigned after it is evaluated. "WIDTH_BUCKET Syntax" describes the `WIDTH_BUCKET` syntax.

You can generate equiwidth histograms with this function. Equiwidth histograms divide data sets into buckets whose interval size (highest value to lowest value) is equal. The number of rows held by each bucket will vary. A related function, `NTILE`, creates equiheight buckets.

Equiwidth histograms can be generated only for numeric, date or datetime types. So the first three parameters should be all numeric expressions or all date expressions. Other types of expressions are not allowed. If the first parameter is `NULL`, the result is `NULL`. If the second or the third parameter is `NULL`, an error message is returned, as a `NULL` value cannot denote any end point (or any point) for a range in a date or numeric value dimension. The last parameter (number of buckets) should be a numeric expression that evaluates to a positive integer value; 0, `NULL`, or a negative value will result in an error.

Buckets are numbered from 0 to (`n`+1). Bucket 0 holds the count of values less than the minimum. Bucket(`n`+1) holds the count of values greater than or equal to the maximum specified value.

### 20.8.2.1 WIDTH_BUCKET Syntax

The `WIDTH_BUCKET` takes four expressions as parameters. The first parameter is the expression that the equiwidth histogram is for. The second and third parameters are expressions that denote the end points of the acceptable range for the first parameter. The fourth parameter denotes the number of buckets.

```
WIDTH_BUCKET(expression, minval expression, maxval expression, num buckets)
```

Consider the following data from table `customers`, that shows the credit limits of 17 customers. This data is gathered in the query shown in Example 20-27.

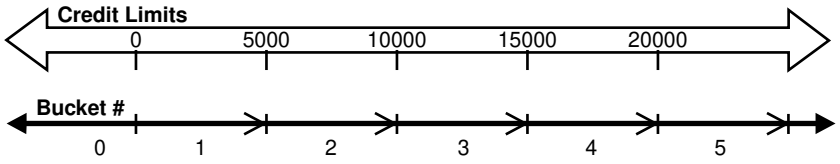
```
CUST_ID    CUST_CREDIT_LIMIT
---------  -----------------
    10346               7000
    35266               7000
    41496              15000
    35225              11000
     3424               9000
    28344               1500
    31112               7000
     8962               1500
    15192               3000
    21380               5000
    36651               1500
    30420               5000
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
  8270              3000
 17268             11000
 14459             11000
 13808              5000
 32497              1500
100977              9000
102077              3000
103066             10000
101784              5000
100421             11000
102343              3000
```

In the table `customers`, the column `cust_credit_limit` contains values between 1500 and 15000, and you can assign the values to four equiwidth buckets, numbered from 1 to 4, by using `WIDTH_BUCKET (cust_credit_limit, 0, 20000, 4)`. Ideally each bucket is a closed-open interval of the real number line, for example, bucket number 2 is assigned to scores between 5000.0000 and 9999.9999..., sometimes denoted [5000, 10000) to indicate that 5,000 is included in the interval and 10,000 is excluded. To accommodate values outside the range [0, 20,000), values less than 0 are assigned to a designated underflow bucket which is numbered 0, and values greater than or equal to 20,000 are assigned to a designated overflow bucket which is numbered 5 (num buckets + 1 in general). See Figure 20-4 for a graphical illustration of how the buckets are assigned.

**Figure 20-4 Bucket Assignments**




You can specify the bounds in the reverse order, for example, `WIDTH_BUCKET (cust_credit_limit, 20000, 0, 4)`. When the bounds are reversed, the buckets will be open-closed intervals. In this example, bucket number 1 is `(15000,20000]`, bucket number 2 is `(10000,15000]`, and bucket number 4, is `(0,5000]`. The overflow bucket will be numbered 0 (`20000, +infinity`), and the underflow bucket will be numbered 5 (`-infinity, 0`].

It is an error if the bucket count parameter is 0 or negative.

**Example 20-27 WIDTH_BUCKET**

The following query shows the bucket numbers for the credit limits in the customers table for both cases where the boundaries are specified in regular or reverse order. You use a range of 0 to 20,000.

```
SELECT cust_id, cust_credit_limit,
 WIDTH_BUCKET(cust_credit_limit,0,20000,4) AS WIDTH_BUCKET_UP,
 WIDTH_BUCKET(cust_credit_limit,20000, 0, 4) AS WIDTH_BUCKET_DOWN
FROM customers WHERE cust_city = 'Marshal';

   CUST_ID CUST_CREDIT_LIMIT WIDTH_BUCKET_UP WIDTH_BUCKET_DOWN
---------- ----------------- --------------- -----------------
     10346              7000               2                 3
     35266              7000               2                 3
     41496             15000               4                 2
     35225             11000               3                 2
      3424              9000               2                 3
     28344              1500               1                 4
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
 31112          7000          2             3
  8962          1500          1             4
 15192          3000          1             4
 21380          5000          2             4
 36651          1500          1             4
 30420          5000          2             4
  8270          3000          1             4
 17268         11000          3             2
 14459         11000          3             2
 13808          5000          2             4
 32497          1500          1             4
100977          9000          2             3
102077          3000          1             4
103066         10000          3             3
101784          5000          2             4
100421         11000          3             2
102343          3000          1             4
```

## 20.8.3 Linear Algebra

Linear algebra is a branch of mathematics with a wide range of practical applications. Many areas have tasks that can be expressed using linear algebra, and here are some examples from several fields: statistics (multiple linear regression and principle components analysis), data mining (clustering and classification), bioinformatics (analysis of microarray data), operations research (supply chain and other optimization problems), econometrics (analysis of consumer demand data), and finance (asset allocation problems). Various libraries for linear algebra are freely available for anyone to use. Oracle's `UTL_NLA` package exposes matrix PL/SQL data types and wrapper PL/SQL subprograms for two of the most popular and robust of these libraries, BLAS and LAPACK.

Linear algebra depends on matrix manipulation. Performing matrix manipulation in PL/SQL in the past required inventing a matrix representation based on PL/SQL's native data types and then writing matrix manipulation routines from scratch. This required substantial programming effort and the performance of the resulting implementation was limited. If developers chose to send data to external packages for processing rather than create their own routines, data transfer back and forth could be time consuming. Using the `UTL_NLA` package lets data stay within Oracle, removes the programming effort, and delivers a fast implementation.

> **See Also**
>
> *Oracle Database PL/SQL Packages and Types Reference* for further information regarding the use of the `UTL_NLA` package and linear algebra

**Example 20-28 Linear Algebra**

Here is an example of how Oracle's linear algebra support could be used for business analysis. It invokes a multiple linear regression application built using the `UTL_NLA` package. The multiple regression application is implemented in an object called `OLS_Regression`. Note that sample files for the OLS Regression object can be found in `$ORACLE_HOME/plsql/demo`.

Consider the scenario of a retailer analyzing the effectiveness of its marketing program. Each of its stores allocates its marketing budget over the following possible programs: media advertisements (`media`), promotions (`promo`), discount coupons (`disct`), and direct mailers (`dmail`). The regression analysis builds a linear relationship between the amount of sales that an average store has in a given year (`sales`) and the spending on the four components of the marketing program. Suppose that the marketing data is stored in the following table:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
sales_marketing_data (
  /* Store information*/
  store_no   NUMBER,
  year       NUMBER,
  /* Sales revenue (in dollars)*/
  sales      NUMBER,   /* sales amount*/
  /* Marketing expenses (in dollars)*/
  media      NUMBER,   /*media advertisements*/
  promo      NUMBER,   /*promotions*/
  disct      NUMBER,   /*discount coupons*/
  dmail      NUMBER,   /*direct mailers*/
```

Then you can build the following sales-marketing linear model using coefficients:

```
Sales Revenue = a  + b Media Advisements
                   + c Promotions
                   + d Discount Coupons
                   + e Direct Mailer
```

This model can be implemented as the following view, which refers to the OLS regression object:

```
CREATE OR REPLACE VIEW sales_marketing_model (year, ols)
   AS SELECT year,
        OLS_Regression(
        /* mean_y => */
        AVG(sales),
        /* variance_y => */
        var_pop(sales),
        /* MV mean vector => */
        UTL_NLA_ARRAY_DBL (AVG(media),AVG(promo),
                           AVG(disct),AVG(dmail)),
        /* VCM variance covariance matrix => */
        UTL_NLA_ARRAY_DBL (var_pop(media),covar_pop(media,promo),
                           covar_pop(media,disct),covar_pop(media,dmail),
                           var_pop(promo),covar_pop(promo,disct),
                           covar_pop(promo,dmail),var_pop(disct),
                           covar_pop(disct,dmail),var_pop(dmail)),
        /* CV covariance vector => */
  UTL_NLA_ARRAY_DBL (covar_pop(sales,media),covar_pop(sales,promo),
                           covar_pop(sales,disct),covar_pop(sales,dmail)))
 FROM sales_marketing_data
 GROUP BY year;
```

Using this view, a marketing program manager can perform an analysis such as "Is this sales-marketing model reasonable for year 2020 data? That is, is the multiple-correlation greater than some acceptable value, say, 0.9?" The SQL for such a query might be as follows:

```
SELECT model.ols.getCorrelation(1)
       AS "Applicability of Linear Model"
FROM sales_marketing_model model
WHERE year = 2020;
```

You could also solve questions such as "What is the expected base-line sales revenue of a store without any marketing programs in 2020?" or "Which component of the marketing program was the most effective in 2021? That is, a dollar increase in which program produced the greatest expected increase in sales?"


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 20.8.4 CASE Expressions

Oracle now supports simple and searched CASE statements. CASE statements are similar in purpose to the DECODE statement, but they offer more flexibility and logical power. They are also easier to read than traditional DECODE statements, and offer better performance as well. They are commonly used when breaking categories into buckets like age (for example, 20-29, 30-39, and so on).

The syntax for simple CASE statements is:

```
CASE expr WHEN comparison_expr THEN return_expr
 [, WHEN comparison_expr THEN return_expr]... [ELSE else_expr] END
```

Simple CASE expressions test if the expr value equals the comparison_expr.

The syntax for searched CASE statements is:

```
CASE WHEN condition THEN return_expr [, WHEN condition THEN return_expr]
   ... ELSE else_expr] END
```

You can use any kind of condition in a searched CASE expression, not just an equality test.

You can specify only 65,535 arguments and each WHEN ... THEN pair counts as two arguments. To avoid exceeding this limit, you can nest CASE expressions so that the return_expr itself is a CASE expression.

> **See Also**
>
> "Creating Histograms Using CASE Statement" for information about using CASE to create histograms

**Example 20-29 CASE**

This statements in this example uses the hr (Human Resources) sample schema.

Suppose you wanted to find the average salary of all employees in the company. If an employee's salary is less than $50000, you want the query to use $50000 instead. Without a CASE statement, you might choose to write this query as follows:

```
SELECT AVG(foo(e.salary)) FROM hr.employees e;
```

In the above example, foo is a function that returns its input if the input is greater than 14000, and returns 14000 otherwise. The query has performance implications because it needs to invoke a function for each row. Writing custom functions can also add to the development load.

Using CASE expressions in the database without PL/SQL, this query can be rewritten as follows.

```
SELECT AVG(CASE when e.salary > 14000 THEN e.salary ELSE 14000 end) AS avg_sal_14k_floor
FROM hr.employees e;
```

Using a CASE expression lets you avoid developing custom functions and can also perform faster.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Example 20-30 CASE for Aggregating Independent Subsets**

Using CASE inside aggregate functions is a convenient way to perform aggregates on multiple subsets of data when a plain GROUP BY will not suffice. For instance, the preceding example could have included multiple AVG columns in its SELECT list, each with its own CASE expression. You might have had a query find the average salary for all employees in the salary ranges 0-14000 and 14001-24000. It would look like:

```
SELECT AVG(CASE WHEN e.salary BETWEEN 0 AND 1400 THEN e.salary ELSE null END) avg14000,
AVG(CASE WHEN e.salary BETWEEN 14001 AND 24000 THEN e.salary ELSE null END) avg24000
FROM employees e;
```

Although this query places the aggregates of independent subsets data into separate columns, by adding a CASE expression to the GROUP BY clause you can display the aggregates as the rows of a single column. The next section shows the flexibility of this approach with two approaches to creating histograms with CASE.

## 20.8.4.1 Creating Histograms Using CASE Statement

You can use the CASE statement when you want to obtain histograms with user-defined buckets (both in number of buckets and width of each bucket). The following are two examples of histograms created with CASE statements. In the first example, the histogram totals are shown in multiple columns and a single row is returned. In the second example, the histogram is shown with a label column and a single column for totals, and multiple rows are returned.

**Example 20-31 Histogram Example 1**

```
SELECT SUM(CASE WHEN cust_credit_limit BETWEEN  0 AND 3999 THEN 1 ELSE 0 END)
  AS "0-3999",
SUM(CASE WHEN cust_credit_limit BETWEEN  4000 AND 7999 THEN 1 ELSE 0 END)
  AS "4000-7999",
SUM(CASE WHEN cust_credit_limit BETWEEN  8000 AND 11999 THEN 1 ELSE 0 END)
  AS "8000-11999",
SUM(CASE WHEN cust_credit_limit BETWEEN  12000 AND 16000 THEN 1 ELSE 0 END)
  AS "12000-16000"
FROM customers WHERE cust_city = 'Marshal';

    0-3999  4000-7999 8000-11999 12000-16000
---------- ---------- ---------- -----------
         8          7          7           1
```

**Example 20-32 Histogram Example 2**

```
SELECT (CASE WHEN cust_credit_limit BETWEEN  0 AND 3999 THEN  ' 0 - 3999'
   WHEN cust_credit_limit BETWEEN  4000 AND 7999 THEN ' 4000 - 7999'
   WHEN cust_credit_limit BETWEEN  8000 AND 11999 THEN  ' 8000 - 11999'
   WHEN cust_credit_limit BETWEEN  12000 AND 16000 THEN '12000 - 16000' END)
  AS BUCKET, COUNT(*) AS Count_in_Group
FROM customers WHERE cust_city = 'Marshal' GROUP BY
 (CASE WHEN cust_credit_limit BETWEEN  0 AND 3999 THEN ' 0 - 3999'
 WHEN cust_credit_limit BETWEEN  4000 AND 7999 THEN ' 4000 - 7999'
 WHEN cust_credit_limit BETWEEN  8000 AND 11999 THEN  ' 8000 - 11999'
 WHEN cust_credit_limit BETWEEN  12000 AND 16000 THEN '12000 - 16000' END);

BUCKET        COUNT_IN_GROUP
------------- --------------
 0 - 3999                  8
 4000 - 7999               7
 8000 - 11999              7
12000 - 16000              1
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

### 20.8.5 Frequent Itemsets in SQL Analytics

Instead of counting how often a given event occurs (for example, how often someone has purchased milk at the grocery), you may find it useful to count how often multiple events occur together (for example, how often someone has purchased both milk and cereal together at the grocery store). You can count these multiple events using what is called a frequent itemset, which is, as the name implies, a set of items. Some examples of itemsets could be all of the products that a given customer purchased in a single trip to the grocery store (commonly called a market basket), the web pages that a user accessed in a single session, or the financial services that a given customer utilizes.

The practical motivation for using a frequent itemset is to find those itemsets that occur most often. If you analyze a grocery store's point-of-sale data, you might, for example, discover that milk and bananas are the most commonly bought pair of items. Frequent itemsets have thus been used in business intelligence environments for many years, with the most common one being for market basket analysis in the retail industry. Frequent itemset calculations are integrated with the database, operating on top of relational tables and accessed through SQL. This integration provides the following key benefits:

- Applications that previously relied on frequent itemset operations now benefit from significantly improved performance as well as simpler implementation.
- SQL-based applications that did not previously use frequent itemsets can now be easily extended to take advantage of this functionality.

Frequent itemsets analysis is performed with the PL/SQL package `DBMS_FREQUENT_ITEMSETS`. See *Oracle Database PL/SQL Packages and Types Reference* for more information. In addition, there is an example of frequent itemset usage in "Business Intelligence Query Example 4: Frequent Itemsets".

## 20.9 Limiting SQL Rows

You can limit the rows returned from SQL queries by either a specific number of rows or a percentage of rows. In some cases, you may need the query results to be ordered before the number of rows returned is limited. A query which first sorts its rows and then limits the number of rows returned is often called a Top-N query, and it offers a straightforward way of creating reports or just a simple view of basic questions, such as "Who are the ten highest-paid employees?" It is also useful for user interfaces that provide the first few rows of a data set for browsing. When you issue a Top-N query, you may also want to specify an offset: the offset excludes the leading rows of the query result set. The query then returns the specified number or percent of rows starting with the first row after the offset. An offset enables you to modify typical questions, so that the question about highest-paid employees might skip the top ten employees and return only those from eleventh to twentieth place in the salary rankings. In a similar manner, you could query the employees by salary, skip the top ten employees and then return the top 10% of the remaining employees.

Queries that limit the rows returned have been possible using the `ROW_NUMBER` window function, the `ROWNUM` pseudocolumn, and other techniques for some time, but can now be written more simply with the ANSI SQL standard syntax of *row_limiting_clause*. When using this clause, you can ensure a deterministic sort order, as needed for Top-N queries, by including an `ORDER BY` clause in the query. The *`row_limiting_clause`* clause appears as the last part of a `SELECT`, after the `ORDER BY` clause, and it starts with either the keyword `FETCH` or `OFFSET`. Its basic syntax is as follows:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
[ OFFSET offset { ROW | ROWS } ]
[ FETCH { FIRST | NEXT } [ { rowcount | percent PERCENT } ]
    { ROW | ROWS } { ONLY | WITH TIES } ]
```

This syntax is illustrated in the following sections.

### OFFSET

This specifies the number of rows to skip before row limiting begins. *offset* must be a number. If you specify a negative number, then offset is treated as 0. If you specify NULL, or a number greater than or equal to the number of rows returned by the query, then 0 rows are returned. If *offset* includes a fraction, then the fractional portion is truncated. If you do not specify this clause, then *offset* is 0 and row limiting begins with the first row. To improve readability, Oracle offers the option of using either ROW or ROWS - both are equivalent.

### FETCH

This specifies the number of rows or percentage of rows to return. If you do not specify this clause, then all rows are returned, beginning at the *offset* + 1 row. If you use the WITH TIES keywords, your query will also include all rows that match the sort key of the last qualified row.

To illustrate how you can limit the number of rows returned in a query, consider the following statements. These statements are run against the hr (Human Resources) schema provided by Oracle.

```
SELECT employee_id, last_name
FROM hr.employees
ORDER BY employee_id
FETCH FIRST 5 ROWS ONLY;

EMPLOYEE_ID LAST_NAME
----------- ----------
        100 King
        101 Yang
        102 Garcia
        103 James
        104 Miller
```

In this statement, the first 5 employees with the lowest employee_id values are returned.

To return the next set of 5 employees, add an OFFSET to the statement:

```
SELECT employee_id, last_name
FROM hr.employees
ORDER BY employee_id
OFFSET 5 ROWS FETCH NEXT 5 ROWS ONLY;

EMPLOYEE_ID LAST_NAME
----------- ----------
        105 Austin
        106 Pataballa
        107 Lorentz
        108 Greenberg
        109 Faviet
```

In this statement, FETCH FIRST and FETCH NEXT are equivalent, but FETCH NEXT is clearer when OFFSET is used.

The offset can be a larger value, such as 10, as in the following statement:

```
SELECT employee_id, last_name
FROM hr.employees
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
ORDER BY employee_id
OFFSET 10 ROWS FETCH NEXT 5 ROWS ONLY;

EMPLOYEE_ID LAST_NAME
----------- ----------
        105 Williams
        106 Jackson
        107 Nguyen
        108 Gruenberg
        109 Faviet
```

You can choose to return values by percentage instead of a fixed number. To illustrate this, the following statement returns the 5 percent of employees with the lowest salaries:

```
SELECT employee_id, last_name, salary
FROM hr.employees
ORDER BY salary
FETCH FIRST 5 PERCENT ROWS ONLY;

EMPLOYEE_ID LAST_NAME             SALARY
----------- -------------------- ------
        132 Olson                  2100
        128 Markle                 2200
        136 Philtanker             2200
        127 Landry                 2400
        135 Gee                    2400
```

In the following result set, 5% is six rows. This is important if you use `OFFSET`, because the percentage calculation is based on the entire result set before the offset is applied. An example of using `OFFSET` is the following statement:

```
SELECT employee_id, last_name, salary
FROM hr.employees
ORDER BY salary, employee_id
OFFSET 6 ROWS FETCH FIRST 5 PERCENT ROWS ONLY;

EMPLOYEE_ID  LAST_NAME           SALARY
----------- -------------------- ------
            132 Olson            2100
           128 Markle            2200
           136 Philtanker        2200
           127 Landry            2400
           135 Gee               2400
           119 Colmenares        2500
```

This statement still returns six rows, but starts with the seventh row of the result set. The additional `employee_id` added to the `ORDER BY` clause was to guarantee a deterministic sort.

You have the option of returning tie values by using `WITH TIES`. This would return the 5 percent with the lowest salaries, plus all additional employees with the same salary as the last row fetched:

```
SELECT employee_id, last_name, salary
FROM hr.employees
ORDER BY salary
FETCH FIRST 5 PERCENT ROWS WITH TIES;

   EMPLOYEE_ID LAST_NAME        SALARY
______________ _____________ __________
           132 Olson              2100
           128 Markle             2200
           136 Philtanker         2200
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
           127 Landry            2400
           135 Gee               2400
           119 Colmenares        2500
           131 Marlow            2500
           140 Patel             2500
           144 Vargas            2500
           182 Sullivan          2500
           191 Perkins           2500
```

You could issue the same query, but skip the first 5 values with the following statement:

```
SELECT employee_id, last_name, salary
FROM hr.employees
ORDER BY salary
OFFSET 5 ROWS FETCH FIRST 5 PERCENT ROWS WITH TIES;

EMPLOYEE_ID LAST_NAME             SALARY
----------- -------------------- ------
        119 Colmenares             2500
        131 Marlow                 2500
        140 Patel                  2500
        144 Vargas                 2500
        182 Sullivan               2500
        191 Perkins                2500
```

## 20.9.1 SQL Row Limiting Restrictions and Considerations

The *row_limiting_clause* clause is subject to the following restrictions:

- You cannot specify this clause with the *for_update_clause*.
- If you specify this clause, then the select list cannot contain the sequence pseudocolumns CURRVAL or NEXTVAL.
- Materialized views are not eligible for an incremental refresh if the defining query contains this clause.

> **See Also**
>
> *Oracle Database SQL Language Reference* for further information regarding syntax and restrictions


Copyright © 2001, 2025, Oracle and/or its affiliates.

21

# SQL for Aggregation in Data Warehouses

This chapter discusses aggregation of SQL, a basic aspect of data warehousing. It contains these topics:


- Overview of SQL for Aggregation in Data Warehouses
- ROLLUP Extension to GROUP BY
- CUBE Extension to GROUP BY
- GROUPING Functions
- GROUPING SETS Expression
- About Composite Columns and Grouping
- Concatenated Groupings and Data Aggregation
- Considerations when Using Aggregation in Data Warehouses
- Computation Using the WITH Clause
- Working with Hierarchical Cubes in SQL


## 21.1 Overview of SQL for Aggregation in Data Warehouses

Aggregation is a fundamental part of data warehousing. To improve aggregation performance in your warehouse, Oracle AI Database provides the following functionality:

- `CUBE` and `ROLLUP` extensions to the `GROUP BY` clause
- Three `GROUPING` functions
- `GROUPING SETS` expression
- Pivoting operations

The `CUBE`, `ROLLUP`, and `GROUPING SETS` extensions to SQL make querying and reporting easier and faster. `CUBE`, `ROLLUP`, and grouping sets produce a single result set that is equivalent to a `UNION ALL` of differently grouped rows. `ROLLUP` calculates aggregations such as `SUM`, `COUNT`, `MAX`, `MIN`, and `AVG` at increasing levels of aggregation, from the most detailed up to a grand total. `CUBE` is an extension similar to `ROLLUP`, enabling a single statement to calculate all possible combinations of aggregations. The `CUBE`, `ROLLUP`, and the `GROUPING SETS` extensions let you specify just the groupings needed in the `GROUP BY` clause. This allows efficient analysis across multiple dimensions without performing a `CUBE` operation. Computing a `CUBE` creates a heavy processing load, so replacing cubes with grouping sets can significantly increase performance.

To enhance performance, `CUBE`, `ROLLUP`, and `GROUPING SETS` can be parallelized: multiple processes can simultaneously execute all of these statements. These capabilities make aggregate calculations more efficient, thereby enhancing database performance, and scalability.

The three `GROUPING` functions help you identify the group each row belongs to and enable sorting subtotal rows and filtering results.

This section contains the following topics:



Copyright © 2001, 2025, Oracle and/or its affiliates.

- About Analyzing Across Multiple Dimensions
- About Optimized Aggregation Performance
- Data Warehousing: An Aggregate Scenario


## 21.1.1 About Analyzing Across Multiple Dimensions

One of the key concepts in decision support systems is multidimensional analysis: examining the enterprise from all necessary combinations of dimensions. The term dimension is used to mean any category used in specifying questions. Among the most commonly specified dimensions are time, geography, product, department, and distribution channel, but the potential dimensions are as endless as the varieties of enterprise activity. The events or entities associated with a particular set of dimension values are usually referred to as facts. The facts might be sales in units or local currency, profits, customer counts, production volumes, or anything else worth tracking.

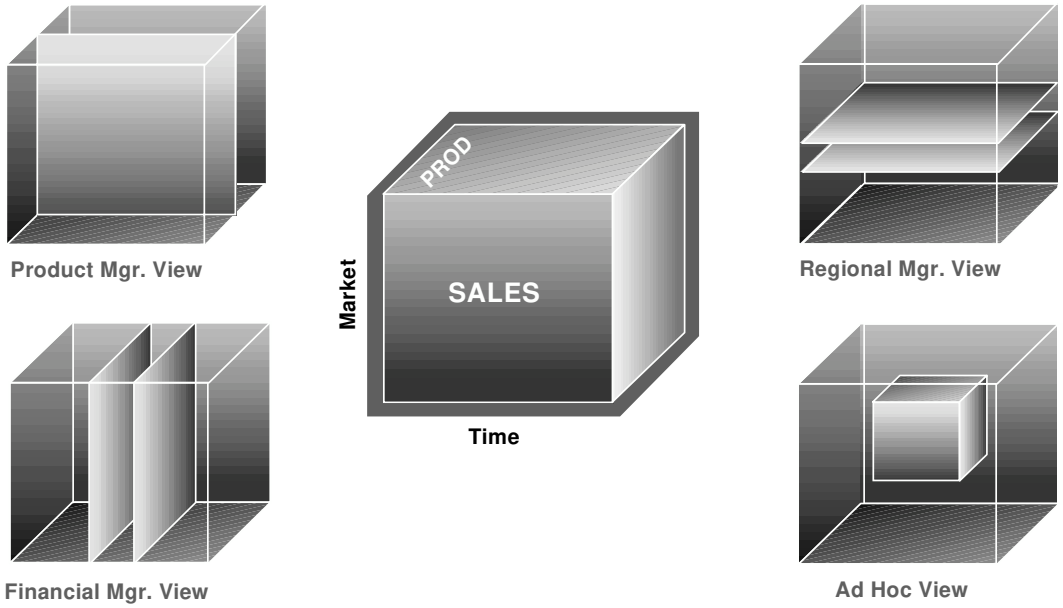
Here are some examples of multidimensional requests:

- Show total sales across all products at increasing aggregation levels for a geography dimension, from state to country to region, for 1999 and 2000.
- Create a cross-tabular analysis of our operations showing expenses by territory in South America for 1999 and 2000. Include all possible subtotals.
- List the top 10 sales representatives in Asia according to 2000 sales revenue for automotive products, and rank their commissions.

All these requests involve multiple dimensions. Many multidimensional questions require aggregated data and comparisons of data sets, often across time, geography or budgets.

To visualize data that has many dimensions, analysts commonly use the analogy of a data cube, that is, a space where facts are stored at the intersection of n dimensions. Figure 21-1 shows a data cube and how it can be used differently by various groups. The cube stores sales data organized by the dimensions of product, market, sales, and time. Note that this is only a metaphor: the actual data is physically stored in normal tables. The cube data consists of both detail and aggregated data.

**Figure 21-1 Logical Cubes and Views by Different Users**





Copyright © 2001, 2025, Oracle and/or its affiliates.

You can retrieve slices of data from the cube. These correspond to cross-tabular reports such as the one shown in #unique_733/unique_733_Connect_42_G1014032. Regional managers might study the data by comparing slices of the cube applicable to different markets. In contrast, product managers might compare slices that apply to different products. An ad hoc user might work with a wide variety of constraints, working in a subset cube.

Answering multidimensional questions often involves accessing and querying huge quantities of data, sometimes in millions of rows. Because the flood of detailed data generated by large organizations cannot be interpreted at the lowest level, aggregated views of the information are essential. Aggregations, such as sums and counts, across many dimensions are vital to multidimensional analyses. Therefore, analytical tasks require convenient and efficient data aggregation.

## 21.1.2 About Optimized Aggregation Performance

Not only multidimensional issues, but all types of processing can benefit from enhanced aggregation facilities. Transaction processing, financial and manufacturing systems—all of these generate large numbers of production reports needing substantial system resources. Improved efficiency when creating these reports will reduce system load. In fact, any computer process that aggregates data from details to higher levels will benefit from optimized aggregation performance.

These extensions provide aggregation features and bring many benefits, including:

- Simplified programming requiring less SQL code for many tasks.
- Quicker and more efficient query processing.
- Reduced client processing loads and network traffic because aggregation work is shifted to servers.
- Opportunities for caching aggregations because similar queries can leverage existing work.

## 21.1.3 Data Warehousing: An Aggregate Scenario

To illustrate the use of the `GROUP BY` extension, this chapter uses the `sh` data of the sample schema. All the examples refer to data from this scenario. The hypothetical company has sales across the world and tracks sales by both dollars and quantities information. Because there are many rows of data, the queries shown here typically have tight constraints on their `WHERE` clauses to limit the results to a small number of rows.

Consider that even a simple report, with just nine values in its grid, generates four subtotals and a grand total. Half of the values needed for this report would not be calculated with a query that requested `SUM(amount_sold)` and did a `GROUP BY(channel_desc, country_id)`. To get the higher-level aggregates would require additional queries. Database commands that offer improved calculation of subtotals bring major benefits to querying, reporting, and analytical operations.

```
SELECT channels.channel_desc, countries.country_iso_code,
  TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sh.sales, sh.customers, sh.times, sh.channels, sh.countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
  sales.channel_id= channels.channel_id AND channels.channel_desc IN
  ('Direct Sales', 'Internet') AND times.calendar_month_desc='2020-09'
  AND customers.country_id=countries.country_id
  AND countries.country_iso_code IN ('US','FR')
GROUP BY CUBE(channels.channel_desc, countries.country_iso_code);
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
CHANNEL_DESC    COUNTRY_ISO_CODE    SALES$
______________  __________________  _______________
                                       884,171
                FR                      64,961
                US                     819,210
Internet                               222,761
Internet        FR                      11,187
Internet        US                     211,574
Direct Sales                           661,411
Direct Sales    FR                      53,774
Direct Sales    US                     607,636
```

**Interpreting NULLs in Aggregation Examples**

NULLs returned by the GROUP BY extensions are not always the traditional null meaning value unknown. Instead, a NULL may indicate that its row is a subtotal. To avoid introducing another non-value in the database system, these subtotal values are not given a special tag.

> **See Also**
>
> GROUPING Functions for details on how the nulls representing subtotals are distinguished from nulls stored in the data

# 21.2 ROLLUP Extension to GROUP BY

ROLLUP enables a SELECT statement to calculate multiple levels of subtotals across a specified group of dimensions. It also calculates a grand total. ROLLUP is a simple extension to the GROUP BY clause, so its syntax is extremely easy to use. The ROLLUP extension is highly efficient, adding minimal overhead to a query.

The action of ROLLUP is straightforward: it creates subtotals that roll up from the most detailed level to a grand total, following a grouping list specified in the ROLLUP clause. ROLLUP takes as its argument an ordered list of grouping columns. First, it calculates the standard aggregate values specified in the GROUP BY clause. Then, it creates progressively higher-level subtotals, moving from right to left through the list of grouping columns. Finally, it creates a grand total.

ROLLUP creates subtotals at n+1 levels, where n is the number of grouping columns. For instance, if a query specifies ROLLUP on grouping columns of time, region, and department(n=3), the result set will include rows at four aggregation levels.

You might want to compress your data when using ROLLUP. This is particularly useful when there are few updates to older partitions.

This section contains the following topics:

- When to Use ROLLUP
- ROLLUP Syntax
- Partial Rollup

## 21.2.1 When to Use ROLLUP

Use the ROLLUP extension in tasks involving subtotals.


Copyright © 2001, 2025, Oracle and/or its affiliates.

- It is very helpful for subtotaling along a hierarchical dimension such as time or geography. For instance, a query could specify a `ROLLUP(y, m, day)` or `ROLLUP(country, state, city)`.
- For data warehouse administrators using summary tables, `ROLLUP` can simplify and speed up the maintenance of summary tables.

## 21.2.2 ROLLUP Syntax

`ROLLUP` appears in the `GROUP BY` clause in a `SELECT` statement. Its form is:

```
SELECT … GROUP BY ROLLUP(grouping_column_reference_list)
```

**Example 21-1 ROLLUP**

This example uses the data in the `sh` sample schema data, the same data as was used in Figure 21-1. The `ROLLUP` is across three dimensions.

```
SELECT channels.channel_desc, calendar_month_desc,
       countries.country_iso_code,
       TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id
  AND sales.cust_id=customers.cust_id
  AND customers.country_id = countries.country_id
  AND sales.channel_id = channels.channel_id
  AND channels.channel_desc IN ('Direct Sales', 'Internet')
  AND times.calendar_month_desc IN ('2019-09', '2019-10')
  AND countries.country_iso_code IN ('GB', 'US')
GROUP BY
  ROLLUP(channels.channel_desc, calendar_month_desc, countries.country_iso_code);

CHANNEL_DESC    CALENDAR_MONTH_DESC    COUNTRY_ISO_CODE    SALES$
_____________ ______________________ ___________________ _____________
Direct Sales    2019-09                GB                     99,275
Direct Sales    2019-10                GB                     93,965
Internet        2019-09                GB                     18,213
Direct Sales    2019-10                US                    656,981
Direct Sales    2019-09                US                    683,977
Internet        2019-10                US                    191,265
Internet        2019-09                US                    144,514
Internet        2019-10                GB                     11,719
Direct Sales    2019-09                                      783,253
Direct Sales    2019-10                                      750,946
Internet        2019-09                                      162,727
Internet        2019-10                                      202,984
Direct Sales                                               1,534,199
Internet                                                     365,711
                                                           1,899,909
```

Note that results do not always add up due to rounding.

This query returns the following sets of rows:

- Regular aggregation rows that would be produced by `GROUP BY` without using `ROLLUP`.
- First-level subtotals aggregating across `country_id` for each combination of `channel_desc` and `calendar_month`.
- Second-level subtotals aggregating across `calendar_month_desc` and `country_id` for each `channel_desc` value.


Copyright © 2001, 2025, Oracle and/or its affiliates.

- A grand total row.

> **Live SQL**
>
> View and run a related example on Oracle Live SQL at Oracle LiveSQL: ROLLUP with GROUP BY

## 21.2.3 Partial Rollup

You can also roll up so that only some of the sub-totals will be included. This partial rollup uses the following syntax:

```
GROUP BY expr1, ROLLUP(expr2, expr3);
```

In this case, the `GROUP BY` clause creates subtotals at (2+1=3) aggregation levels. That is, at level (`expr1`, `expr2`, `expr3`), (`expr1`, `expr2`), and (`expr1`).

**Example 21-2 Partial ROLLUP**

```
SELECT channel_desc, calendar_month_desc, countries.country_iso_code,
   TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sh.sales, sh.customers, sh.times, sh.channels, sh.countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id
  AND customers.country_id = countries.country_id
  AND sales.channel_id= channels.channel_id
  AND channels.channel_desc IN ('Direct Sales', 'Internet')
  AND times.calendar_month_desc IN ('2020-10', '2021-10')
  AND countries.country_iso_code IN ('GB', 'US')
GROUP BY channel_desc, ROLLUP(calendar_month_desc, countries.country_iso_code);
CHANNEL_DESC    CALENDAR_MONTH_DESC    COUNTRY_ISO_CODE    SALES$
_____________ _____________________ ___________________ _____________
Direct Sales    2020-10                GB                     83,685
Direct Sales    2021-10                GB                     91,925
Direct Sales    2021-10                US                    682,297
Direct Sales    2020-10                US                    596,924
Internet        2020-10                US                     32,480
Internet        2021-10                US                    137,054
Internet        2020-10                GB                      2,743
Internet        2021-10                GB                     14,539
Direct Sales    2020-10                                      680,609
Direct Sales    2021-10                                      774,222
Internet        2020-10                                       35,223
Internet        2021-10                                      151,593
Direct Sales                                               1,454,831
Internet                                                     186,816
```

- Regular aggregation rows that would be produced by `GROUP BY` without using `ROLLUP`.
- First-level subtotals aggregating across `country_id` for each combination of `channel_desc` and `calendar_month_desc`.
- Second-level subtotals aggregating across `calendar_month_desc` and `country_id` for each `channel_desc` value.
- It does not produce a grand total row.


Copyright © 2001, 2025, Oracle and/or its affiliates.

# 21.3 CUBE Extension to GROUP BY

`CUBE` takes a specified set of grouping columns and creates subtotals for all of their possible combinations. In terms of multidimensional analysis, `CUBE` generates all the subtotals that could be calculated for a data cube with the specified dimensions. If you have specified `CUBE(time, region, department)`, the result set will include all the values that would be included in an equivalent `ROLLUP` statement plus additional combinations.

For instance, the departmental totals across regions (279,000 and 319,000) would not be calculated by a `ROLLUP(time, region, department)` clause, but they would be calculated by a `CUBE(time, region, department)` clause. If *n* columns are specified for a `CUBE`, there will be 2 to the *n* combinations of subtotals returned.

CUBE Syntax gives an example of a three-dimension cube.

> **See Also**
>
> *Oracle Database SQL Language Reference* for syntax and restrictions

This section contains the following topics:

- When to Use CUBE
- CUBE Syntax
- Partial CUBE
- Calculating Subtotals Without CUBE

## 21.3.1 When to Use CUBE

Consider Using `CUBE` in any situation requiring cross-tabular reports. The data needed for cross-tabular reports can be generated with a single `SELECT` using `CUBE`. Like `ROLLUP`, `CUBE` can be helpful in generating summary tables. Note that population of summary tables is even faster if the `CUBE` query executes in parallel.

`CUBE` is typically most suitable in queries that use columns from multiple dimensions rather than columns representing different levels of a single dimension. For instance, a commonly requested cross-tabulation might need subtotals for all the combinations of month, state, and product. These are three independent dimensions, and analysis of all possible subtotal combinations is commonplace. In contrast, a cross-tabulation showing all possible combinations of year, month, and day would have several values of limited interest, because there is a natural hierarchy in the time dimension. Subtotals such as profit by day of month summed across year would be unnecessary in most analyses. Relatively few users need to ask "What were the total sales for the 16th of each month across the year?" See "Hierarchy Handling in ROLLUP and CUBE" for an example of handling rollup calculations efficiently.

## 21.3.2 CUBE Syntax

`CUBE` appears in the `GROUP BY` clause in a `SELECT` statement. Its form is:

```
SELECT …  GROUP BY CUBE (grouping_column_reference_list)
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Example 21-3 CUBE Keyword in a Query**

```
SELECT channel_desc, calendar_month_desc, countries.country_iso_code,
      TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
  sales.channel_id= channels.channel_id
 AND customers.country_id = countries.country_id
 AND channels.channel_desc IN
  ('Direct Sales', 'Internet') AND times.calendar_month_desc IN
  ('2020-09', '2022-10') AND countries.country_iso_code IN ('GB', 'US')
GROUP BY CUBE(channel_desc, calendar_month_desc, countries.country_iso_code);
CHANNEL_DESC    CALENDAR_MONTH_DESC    COUNTRY_ISO_CODE    SALES$
_______________ ______________________ ___________________ _______________
                                                             1,990,931
                                       GB                      218,674
                                       US                    1,772,256
                2020-09                                        923,577
                2020-09                GB                      104,367
                2020-09                US                      819,210
                2022-10                                      1,067,354
                2022-10                GB                      114,307
                2022-10                US                      953,046
Internet                                                       650,303
Internet                               GB                       52,403
Internet                               US                      597,900
Internet        2020-09                                        224,965
Internet        2020-09                GB                       13,392
Internet        2020-09                US                      211,574
Internet        2022-10                                        425,337
Internet        2022-10                GB                       39,011
Internet        2022-10                US                      386,327
Direct Sales                                                 1,340,628
Direct Sales                           GB                      166,272
Direct Sales                           US                    1,174,356
Direct Sales    2020-09                                        698,612
Direct Sales    2020-09                GB                       90,975
Direct Sales    2020-09                US                      607,636
Direct Sales    2022-10                                        642,016
Direct Sales    2022-10                GB                       75,296
Direct Sales    2022-10                US                      566,720
```

This query illustrates `CUBE` aggregation across three dimensions.

## 21.3.3 Partial CUBE

Partial `CUBE` resembles partial `ROLLUP` in that you can limit it to certain dimensions and precede it with columns outside the `CUBE` operator. In this case, subtotals of all possible combinations are limited to the dimensions within the cube list (in parentheses), and they are combined with the preceding items in the `GROUP BY` list.

The syntax for partial `CUBE` is as follows:

```
GROUP BY expr1, CUBE(expr2, expr3)
```

This syntax example calculates 2*2, or 4, subtotals. That is:

- (expr1, expr2, expr3)
- (expr1, expr2)
- (expr1, expr3)


Copyright © 2001, 2025, Oracle and/or its affiliates.

- (expr1)

**Example 21-4 Partial CUBE in a Query**

Using the sales database, you can issue the following statement:

```
SELECT channel_desc, calendar_month_desc, countries.country_iso_code,
       TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sh.sales, sh.customers, sh.times, sh.channels, sh.countries
WHERE sales.time_id = times.time_id
  AND sales.cust_id = customers.cust_id
  AND customers.country_id=countries.country_id
  AND sales.channel_id = channels.channel_id
  AND channels.channel_desc IN ('Direct Sales', 'Internet')
  AND times.calendar_month_desc IN ('2020-09', '2021-10')
  AND countries.country_iso_code IN ('GB', 'US')
GROUP BY channel_desc, CUBE(calendar_month_desc, countries.country_iso_code);
```

```
CHANNEL_DESC    CALENDAR_MONTH_DESC    COUNTRY_ISO_CODE    SALES$
_____________ ______________________ ___________________ _____________
Internet                                                        376,559
Internet                             GB                          27,931
Internet                             US                         348,628
Internet        2020-09                                         224,965
Internet        2020-09              GB                          13,392
Internet        2020-09              US                         211,574
Internet        2021-10                                         151,593
Internet        2021-10              GB                          14,539
Internet        2021-10              US                         137,054
Direct Sales                                                  1,472,834
Direct Sales                         GB                         182,901
Direct Sales                         US                       1,289,933
Direct Sales    2020-09                                         698,612
Direct Sales    2020-09              GB                          90,975
Direct Sales    2020-09              US                         607,636
Direct Sales    2021-10                                         774,222
Direct Sales    2021-10              GB                          91,925
Direct Sales    2021-10              US                         682,297
```

## 21.3.4 Calculating Subtotals Without CUBE

Just as for ROLLUP, multiple SELECT statements combined with UNION ALL statements could provide the same information gathered through CUBE. However, this might require many SELECT statements. For an n-dimensional cube, 2 to the *n* SELECT statements are needed. In the three-dimension example, this would mean issuing SELECT statements linked with UNION ALL. So many SELECT statements yield inefficient processing and very lengthy SQL.

Consider the impact of adding just one more dimension when calculating all possible combinations: the number of SELECT statements would double to 16. The more columns used in a CUBE clause, the greater the savings compared to the UNION ALL approach.

# 21.4 GROUPING Functions

Two challenges arise with the use of ROLLUP and CUBE. First, how can you programmatically determine which result set rows are subtotals, and how do you find the exact level of aggregation for a given subtotal? You often need to use subtotals in calculations such as percent-of-totals, so you need an easy way to determine which rows are the subtotals. Second, what happens if query results contain both stored NULL values and "NULL" values created by a


Copyright © 2001, 2025, Oracle and/or its affiliates.

ROLLUP or CUBE? How can you differentiate between the two? This section discusses some of these situations.

> **See Also**
>
> *Oracle Database SQL Language Reference* for syntax and restrictions

This section contains the following topics:

- GROUPING Function
- When to Use GROUPING
- GROUPING_ID Function
- GROUP_ID Function

## 21.4.1 GROUPING Function

GROUPING handles these problems. Using a single column as its argument, GROUPING returns 1 when it encounters a NULL value created by a ROLLUP or CUBE operation. That is, if the NULL indicates the row is a subtotal, GROUPING returns a 1. Any other type of value, including a stored NULL, returns a 0.

GROUPING appears in the selection list portion of a SELECT statement. Its form is:

```
SELECT …  [GROUPING(dimension_column)…]  …
  GROUP BY …    {CUBE | ROLLUP| GROUPING SETS}  (dimension_column)
```

**Example 21-5 GROUPING to Mask Columns**

This example uses GROUPING to create a set of mask columns for the result set shown in Example 21-2. The mask columns are easy to analyze programmatically.

```
SELECT channel_desc, calendar_month_desc, country_iso_code,
TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$, GROUPING(channel_desc) AS Ch,
   GROUPING(calendar_month_desc) AS Mo, GROUPING(country_iso_code) AS Co
FROM sh.sales, sh.customers, sh.times, sh.channels, sh.countries
WHERE sales.time_id=times.time_id
  AND sales.cust_id=customers.cust_id
  AND customers.country_id = countries.country_id
  AND sales.channel_id= channels.channel_id
  AND channels.channel_desc IN ('Direct Sales', 'Internet')
  AND times.calendar_month_desc IN ('2019-09', '2022-10')
  AND countries.country_iso_code IN ('GB', 'US')
GROUP BY ROLLUP(channel_desc, calendar_month_desc, countries.country_iso_code);

CHANNEL_DESC    CALENDAR_MONTH_DESC    COUNTRY_ISO_CODE    SALES$               CH
MO    CO
_____
Direct Sales    2019-09                GB                        99,275        0
0     0
Direct Sales    2022-10                GB                        75,296        0
0     0
Internet        2019-09                GB                        18,213        0
0     0
Internet        2022-10                GB                        39,011        0
0     0
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
Internet        2022-10               US                      386,327        0
0     0
Direct Sales    2022-10               US                      566,720        0
0     0
Direct Sales    2019-09               US                      683,977        0
0     0
Internet        2019-09               US                      144,514        0
0     0
Direct Sales    2019-09                                       783,253        0
0     1
Direct Sales    2022-10                                       642,016        0
0     1
Internet        2019-09                                       162,727        0
0     1
Internet        2022-10                                       425,337        0
0     1
Direct Sales                                                1,425,269        0
1     1
Internet                                                      588,064        0
1     1
                                                            2,013,333        1
1     1
```

A program can easily identify the detail rows by a mask of "0 0 0" on the T, R, and D columns. The first level subtotal rows have a mask of "0 0 1", the second level subtotal rows have a mask of "0 1 1", and the overall total row has a mask of "1 1 1".

You can improve the readability of result sets by using the `GROUPING` and `DECODE` functions as shown in Example 21-6.

### Example 21-6 GROUPING For Readability

```
SELECT DECODE(GROUPING(channel_desc), 1, 'Multi-channel sum', channel_desc) AS
 Channel, DECODE (GROUPING (country_iso_code), 1, 'Multi-country sum',
 country_iso_code) AS Country, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sh.sales, sh.customers, sh.times, sh.channels, sh.countries
WHERE sales.time_id=times.time_id
  AND sales.cust_id=customers.cust_id
  AND customers.country_id = countries.country_id
  AND sales.channel_id= channels.channel_id
  AND channels.channel_desc IN ('Direct Sales', 'Internet')
  AND times.calendar_month_desc= '2022-09'
  AND country_iso_code IN ('GB', 'US')
GROUP BY CUBE(channel_desc, country_iso_code);

CHANNEL              COUNTRY              SALES$
____________________ ____________________ _______________
Multi-channel sum    Multi-country sum        1,050,492
Multi-channel sum    GB                         129,672
Multi-channel sum    US                         920,820
Internet             Multi-country sum          336,429
Internet             GB                          36,807
Internet             US                         299,622
Direct Sales         Multi-country sum          714,063
Direct Sales         GB                          92,865
Direct Sales         US                         621,198
```

To understand the previous statement, note its first column specification, which handles the channel_desc column. Consider the first line of the previous statement:

```
SELECT DECODE(GROUPING(channel_desc), 1, 'Multi-Channel sum', channel_desc)AS Channel
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

In this, the channel_desc value is determined with a DECODE function that contains a GROUPING function. The GROUPING function returns a 1 if a row value is an aggregate created by ROLLUP or CUBE, otherwise it returns a 0. The DECODE function then operates on the GROUPING function's results. It returns the text "All Channels" if it receives a 1 and the channel_desc value from the database if it receives a 0. Values from the database will be either a real value such as "Internet" or a stored NULL. The second column specification, displaying country_id, works the same way.

## 21.4.2 When to Use GROUPING

The GROUPING function is not only useful for identifying NULLs, it also enables sorting subtotal rows and filtering results. In Example 21-7, you retrieve a subset of the subtotals created by a CUBE and none of the base-level aggregations. The HAVING clause constrains columns that use GROUPING functions.

**Example 21-7 GROUPING Combined with HAVING**

```
SELECT channel_desc, calendar_month_desc, country_iso_code, TO_CHAR(
SUM(amount_sold), '9,999,999,999') SALES$, GROUPING(channel_desc) CH, GROUPING
  (calendar_month_desc)  MO, GROUPING(country_iso_code) CO
FROM sh.sales, sh.customers, sh.times, sh.channels, sh.countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id
  AND customers.country_id = countries.country_id
  AND sales.channel_id= channels.channel_id
  AND channels.channel_desc IN ('Direct Sales', 'Internet')
  AND times.calendar_month_desc IN ('2020-09', '2022-10')
  AND country_iso_code IN ('GB', 'US')
GROUP BY CUBE(channel_desc, calendar_month_desc, country_iso_code)
HAVING (GROUPING(channel_desc)=1 AND GROUPING(calendar_month_desc)= 1
  AND GROUPING(country_iso_code)=1) OR (GROUPING(channel_desc)=1
  AND GROUPING (calendar_month_desc)= 1) OR (GROUPING(country_iso_code)=1
  AND GROUPING(calendar_month_desc)= 1);

CHANNEL_DESC    CALENDAR_MONTH_DESC    COUNTRY_ISO_CODE    SALES$               CH
MO    CO
_____________ _____________________ ___________________ ________________ _____ _____
_____
Direct Sales                                                  1,340,628        0
1     1
Internet                                                        650,303        0
1     1
                                                              1,990,931        1
1     1
                                      US                      1,772,256        1
1     0
                                      GB                        218,674        1
1     0
```

Compare the result set of Example 21-7 with that in Example 21-2 to see how Example 21-7 is a precisely specified group: it contains only the yearly totals, regional totals aggregated over time and department, and the grand total.

## 21.4.3 GROUPING_ID Function

To find the GROUP BY level of a particular row, a query must return GROUPING function information for each of the GROUP BY columns. If you do this using the GROUPING function, every GROUP BY column requires another column using the GROUPING function. For instance, a four-column GROUP BY clause must be analyzed with four GROUPING functions. This is inconvenient to write in


Copyright © 2001, 2025, Oracle and/or its affiliates.

SQL and increases the number of columns required in the query. When you want to store the query result sets in tables, as with materialized views, the extra columns waste storage space.

To address these problems, you can use the `GROUPING_ID` function. `GROUPING_ID` returns a single number that enables you to determine the exact `GROUP BY` level. For each row, `GROUPING_ID` takes the set of 1's and 0's that would be generated if you used the appropriate `GROUPING` functions and concatenates them, forming a bit vector. The bit vector is treated as a binary number, and the number's base-10 value is returned by the `GROUPING_ID` function. For instance, if you group with the expression `CUBE(a, b)` the possible values are as shown in Table 21-1.

**Table 21-1 GROUPING_ID Example for CUBE(a, b)**

| Aggregation Level | Bit Vector | GROUPING_ID |
|---|---|---|
| a, b | 0 0 | 0 |
| a | 0 1 | 1 |
| b | 1 0 | 2 |
| Grand Total | 1 1 | 3 |

`GROUPING_ID` clearly distinguishes groupings created by grouping set specification, and it is very useful during refresh and rewrite of materialized views.

## 21.4.4 GROUP_ID Function

While the extensions to `GROUP BY` offer power and flexibility, they also allow complex result sets that can include duplicate groupings. The `GROUP_ID` function lets you distinguish among duplicate groupings. If there are multiple sets of rows calculated for a given level, `GROUP_ID` assigns the value of 0 to all the rows in the first set. All other sets of duplicate rows for a particular grouping are assigned higher values, starting with 1. For example, consider the following query, which generates a duplicate grouping:

**Example 21-8 GROUP_ID in a Query**

```
SELECT country_iso_code, SUBSTR(cust_state_province,1,12), SUM(amount_sold),
  GROUPING_ID(country_iso_code, cust_state_province) GROUPING_ID, GROUP_ID()
FROM sh.sales, sh.customers, sh.times, sh.countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id
  AND customers.country_id=countries.country_id AND times.time_id= '30-OCT-22'
  AND country_iso_code IN ('FR')
GROUP BY GROUPING SETS (country_iso_code,
ROLLUP(country_iso_code, cust_state_province));
COUNTRY_ISO_CODE    SUBSTR(CUST_STATE_PROVINCE,1,12)       SUM(AMOUNT_SOLD)
GROUPING_ID    GROUP_ID()
_____________________ _____________________________________ ____________________
_____________ ____________
FR                    Languedoc-Ro                                    722.16
0             0
FR                    Rhtne-Alpes                                    1527.04
0             0
                                                                     2249.2
3             0
FR                                                                   2249.2
1             0
FR                                                                   2249.2
1             1
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

This query generates the following groupings: (country_id, cust_state_province), (country_id), (country_id), and (). Note that the grouping (country_id) is repeated twice. The syntax for GROUPING SETS is explained in "GROUPING SETS Expression".

This function helps you filter out duplicate groupings from the result. For example, you can filter out duplicate (region) groupings from the previous example by adding a HAVING clause condition GROUP_ID()=0 to the query.

# 21.5 GROUPING SETS Expression

You can selectively specify the set of groups that you want to create using a GROUPING SETS expression within a GROUP BY clause. This allows precise specification across multiple dimensions without computing the whole CUBE. "GROUPING SETS Syntax" contains the GROUPING SETS syntax.

For example, you can say:

```
SELECT channel_desc, calendar_month_desc, country_iso_code,
       TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sh.sales, sh.customers, sh.times, sh.channels, sh.countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
  sales.channel_id= channels.channel_id AND channels.channel_desc IN
 ('Direct Sales', 'Internet') AND times.calendar_month_desc IN
 ('2020-09', '2022-10') AND country_iso_code IN ('GB', 'US')
GROUP BY GROUPING SETS((channel_desc, calendar_month_desc, country_iso_code),
    (channel_desc, country_iso_code), (calendar_month_desc, country_iso_code));
```

Note that this statement uses composite columns, described in "About Composite Columns and Grouping". This statement calculates aggregates over three groupings:

- (channel_desc, calendar_month_desc, country_iso_code)
- (channel_desc, country_iso_code)
- (calendar_month_desc, country_iso_code)

Compare the previous statement with the following alternative, which uses the CUBE operation and the GROUPING_ID function to return the desired rows:

```
SELECT channel_desc, calendar_month_desc, country_iso_code,
       TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$,
       GROUPING_ID(channel_desc, calendar_month_desc, country_iso_code) gid
FROM sh.sales, sh.customers, sh.times, sh.channels, sh.countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
  sales.channel_id= channels.channel_id AND channels.channel_desc IN
 ('Direct Sales', 'Internet') AND times.calendar_month_desc IN
 ('2020-09', '2022-10') AND country_iso_code IN ('GB', 'US')
GROUP BY CUBE(channel_desc, calendar_month_desc, country_iso_code)
HAVING GROUPING_ID(channel_desc, calendar_month_desc, country_iso_code)=0
  OR GROUPING_ID(channel_desc, calendar_month_desc, country_iso_code)=2
  OR GROUPING_ID(channel_desc, calendar_month_desc, country_iso_code)=4;
```

This statement computes all the 8 (2 *2 *2) groupings, though only the previous 3 groups are of interest to you.

Another alternative is the following statement, which is lengthy due to several unions. This statement requires three scans of the base table, making it inefficient. CUBE and ROLLUP can be thought of as grouping sets with very specific semantics. For example, consider the following statement:

```
CUBE(a, b, c)
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

This statement is equivalent to:

```
GROUPING SETS ((a, b, c), (a, b), (a, c), (b, c), (a), (b), (c), ())
ROLLUP(a, b, c)
```

And this statement is equivalent to:

```
GROUPING SETS ((a, b, c), (a, b), ())
```

## 21.5.1 GROUPING SETS Syntax

GROUPING SETS syntax lets you define multiple groupings in the same query. GROUP BY computes all the groupings specified and combines them with UNION ALL. For example, consider the following statement:

```
GROUP BY GROUPING sets (channel_desc, calendar_month_desc, country_id )
```

This statement is equivalent to:

```
GROUP BY channel_desc UNION ALL
GROUP BY calendar_month_desc UNION ALL GROUP BY country_id
```

Table 21-2 shows grouping sets specification and equivalent GROUP BY specification. Note that some examples use composite columns.

**Table 21-2 GROUPING SETS Statements and Equivalent GROUP BY**

| GROUPING SETS Statement | Equivalent GROUP BY Statement |
|---|---|
| GROUP BY GROUPING SETS(a, b, c) | GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY c |
| GROUP BY GROUPING SETS(a, b, (b, c)) | GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY b, c |
| GROUP BY GROUPING SETS((a, b, c)) | GROUP BY a, b, c |
| GROUP BY GROUPING SETS(a, (b), ()) | GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY () |
| GROUP BY GROUPING SETS(a, ROLLUP(b, c)) | GROUP BY a UNION ALL GROUP BY ROLLUP(b, c) |

In the absence of an optimizer that looks across query blocks to generate the execution plan, a query based on UNION would need multiple scans of the base table, sales. This could be very inefficient as fact tables will normally be huge. Using GROUPING SETS statements, all the groupings of interest are available in the same query block.

# 21.6 About Composite Columns and Grouping

A composite column is a collection of columns that are treated as a unit during the computation of groupings. You specify the columns in parentheses as in the following statement:

```
ROLLUP (year, (quarter, month), day)
```

In this statement, the data is not rolled up across year and quarter, but is instead equivalent to the following groupings of a UNION ALL:

- (year, quarter, month, day),
- (year, quarter, month),


Copyright © 2001, 2025, Oracle and/or its affiliates.

- (year)
- ()

Here, (quarter, month) form a composite column and are treated as a unit. In general, composite columns are useful in ROLLUP, CUBE, GROUPING SETS, and concatenated groupings. For example, in CUBE or ROLLUP, composite columns would mean skipping aggregation across certain levels. That is, the following statement:

```
GROUP BY ROLLUP(a, (b, c))
```

This is equivalent to:

```
GROUP BY a, b, c UNION ALL
GROUP BY a UNION ALL
GROUP BY ()
```

Here, (b, c) are treated as a unit and rollup will not be applied across (b, c). It is as if you have an alias, for example z, for (b, c) and the GROUP BY expression reduces to GROUP BY ROLLUP(a, z). Compare this with the normal rollup as in the following:

```
GROUP BY ROLLUP(a, b, c)
```

This would be the following:

```
GROUP BY a, b, c UNION ALL
GROUP BY a, b UNION ALL
GROUP BY a UNION ALL
GROUP BY ().
```

Similarly, the following statement is equivalent to the four GROUP BYs:

```
GROUP BY CUBE((a, b), c)

GROUP BY a, b, c UNION ALL
GROUP BY a, b UNION ALL
GROUP BY c UNION ALL
GROUP By ()
```

In GROUPING SETS, a composite column is used to denote a particular level of GROUP BY. See #unique_746/unique_746_Connect_42_G1014117 for more examples of composite columns.

**Example 21-9 Composite Columns**

You do not have full control over what aggregation levels you want with CUBE and ROLLUP. For example, consider the following statement:

```
SELECT channel_desc, calendar_month_desc, country_iso_code,
 TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sh.sales, sh.customers, sh.times, sh.channels, sh.countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id
  AND customers.country_id = countries.country_id
  AND sales.channel_id= channels.channel_id
  AND channels.channel_desc IN ('Direct Sales', 'Internet')
  AND times.calendar_month_desc IN ('2020-09', '2022-10')
  AND country_iso_code IN ('GB', 'US')
GROUP BY ROLLUP(channel_desc, calendar_month_desc, country_iso_code);
```

This statement results in Oracle computing the following groupings:

- (channel_desc, calendar_month_desc, country_iso_code)
- (channel_desc, calendar_month_desc)


Copyright © 2001, 2025, Oracle and/or its affiliates.

- (channel_desc)
- ()

If you are just interested in the first, third, and fourth of these groupings, you cannot limit the calculation to those groupings without using composite columns. With composite columns, this is possible by treating month and country as a single unit while rolling up. Columns enclosed in parentheses are treated as a unit while computing CUBE and ROLLUP. Thus, you would say:

```
SELECT channel_desc, calendar_month_desc, country_iso_code,
    TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sh.sales, sh.customers, sh.times, sh.channels, sh.countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
      sales.channel_id= channels.channel_id  AND channels.channel_desc IN
 ('Direct Sales', 'Internet') AND times.calendar_month_desc IN
 ('2020-09', '2022-10') AND country_iso_code IN ('GB', 'US')
GROUP BY ROLLUP(channel_desc, (calendar_month_desc, country_iso_code));

CHANNEL_DESC    CALENDAR_MONTH_DESC    COUNTRY_ISO_CODE    SALES$
_____________ ______________________ ____________________ _____________
Direct Sales    2020-09                US                     1,189,377
Direct Sales    2022-10                US                     1,088,611
Internet        2020-09                US                       389,343
Internet        2022-10                US                       711,634
Direct Sales    2020-09                GB                     1,189,377
Direct Sales    2022-10                GB                     1,088,611
Internet        2020-09                GB                       389,343
Internet        2022-10                GB                       711,634
Direct Sales                                                  4,555,976
Internet                                                      2,201,954
                                                              6,757,929
```

# 21.7 Concatenated Groupings and Data Aggregation

Concatenated groupings offer a concise way to generate useful combinations of groupings. Groupings specified with concatenated groupings yield the cross-product of groupings from each grouping set. The cross-product operation enables even a small number of concatenated groupings to generate a large number of final groups. The concatenated groupings are specified simply by listing multiple grouping sets, cubes, and rollups, and separating them with commas. Here is an example of concatenated grouping sets:

```
GROUP BY GROUPING SETS(a, b), GROUPING SETS(c, d)
```

This SQL defines the following groupings:

```
(a, c), (a, d), (b, c), (b, d)
```

Concatenation of grouping sets is very helpful for these reasons:

- Ease of query development

  You need not enumerate all groupings manually.

- Use by applications

  SQL generated by analytical applications often involves concatenation of grouping sets, with each grouping set defining groupings needed for a dimension.

**Example 21-10 Concatenated Groupings**

You can also specify more than one grouping in the GROUP BY clause. For example, if you want aggregated sales values for each product rolled up across all levels in the time dimension


Copyright © 2001, 2025, Oracle and/or its affiliates.

(year, month and day), and across all levels in the geography dimension (region), you can issue the following statement:

```
SELECT channel_desc, calendar_year, calendar_quarter_desc, country_iso_code,
  cust_state_province, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sh.sales, sh.customers, sh.times, sh.channels, sh.countries
WHERE sales.time_id = times.time_id AND sales.cust_id = customers.cust_id
 AND sales.channel_id = channels.channel_id AND countries.country_id =
    customers.country_id AND channels.channel_desc IN
   ('Direct Sales', 'Internet') AND times.calendar_month_desc IN ('2020-09',
 '2022-10') AND countries.country_iso_code IN ('GB', 'FR')
GROUP BY channel_desc, GROUPING SETS (ROLLUP(calendar_year,
   calendar_quarter_desc),
ROLLUP(country_iso_code, cust_state_province));
```

This results in the following groupings:

- (channel_desc, calendar_year, calendar_quarter_desc)
- (channel_desc, calendar_year)
- (channel_desc)
- (channel_desc, country_iso_code, cust_state_province)
- (channel_desc, country_iso_code)
- (channel_desc)

This is the cross-product of the following:

- The expression, channel_desc
- ROLLUP(calendar_year, calendar_quarter_desc), which is equivalent to ((calendar_year, calendar_quarter_desc), (calendar_year), ())
- ROLLUP(country_iso_code, cust_state_province), which is equivalent to ((country_iso_code, cust_state_province), (country_iso_code), ())

Note that the output contains two occurrences of (channel_desc) group. To filter out the extra (channel_desc) group, the query could use a GROUP_ID function.

Another concatenated join example is Example 21-11, showing the cross product of two grouping sets.

**Example 21-11 Concatenated Groupings (Cross-Product of Two Grouping Sets)**

```
SELECT country_iso_code, cust_state_province, calendar_year,
calendar_quarter_desc, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sh.sales, sh.customers, sh.times, sh.channels, sh.countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
 countries.country_id=customers.country_id  AND
  sales.channel_id= channels.channel_id AND channels.channel_desc IN
 ('Direct Sales', 'Internet') AND times.calendar_month_desc IN
 ('2020-09', '2022-10') AND country_iso_code IN ('GB', 'FR')
GROUP BY GROUPING SETS (country_iso_code, cust_state_province),
         GROUPING SETS (calendar_year, calendar_quarter_desc);
```

This statement results in the computation of groupings:

- (country_iso_code, year), (country_iso_code, calendar_quarter_desc), (cust_state_province, year) and (cust_state_province, calendar_quarter_desc)


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 21.7.1 Concatenated Groupings and Hierarchical Data Cubes

One of the most important uses for concatenated groupings is to generate the aggregates needed for a hierarchical cube of data. A hierarchical cube is a data set where the data is aggregated along the rollup hierarchy of each of its dimensions and these aggregations are combined across dimensions. It includes the typical set of aggregations needed for business intelligence queries. By using concatenated groupings, you can generate all the aggregations needed by a hierarchical cube with just *n* `ROLLUP`s (where *n* is the number of dimensions), and avoid generating unwanted aggregations.

Consider just three of the dimensions in the `sh` sample schema data set, each of which has a multilevel hierarchy:

- time: `year`, `quarter`, `month`, `day` (`week` is in a separate hierarchy)
- product: `category`, `subcategory`, `prod_name`
- geography: `region`, `subregion`, `country`, `state`, `city`

This data is represented using a column for each level of the hierarchies, creating a total of twelve columns for dimensions, plus the columns holding sales figures.

For your business intelligence needs, you would like to calculate and store certain aggregates of the various combinations of dimensions. In Example 21-12, you create the aggregates for all levels, except for "day", which would create too many rows. In particular, you want to use `ROLLUP` within each dimension to generate useful aggregates. Once you have the `ROLLUP`-based aggregates within each dimension, you want to combine them with the other dimensions. This will generate a hierarchical cube. Note that this is not at all the same as a `CUBE` using all twelve of the dimension columns: that would create 2 to the 12th power (4,096) aggregation groups, of which you need only a small fraction. Concatenated grouping sets make it easy to generate exactly the aggregations you need. Example 21-12 shows where a `GROUP BY` clause is needed.

**Example 21-12 Concatenated Groupings and Hierarchical Cubes**

```
SELECT calendar_year, calendar_quarter_desc, calendar_month_desc,
  country_region, country_subregion, countries.country_iso_code,
 cust_state_province, cust_city, prod_category_desc, prod_subcategory_desc,
 prod_name, TO_CHAR(SUM (amount_sold), '9,999,999,999') SALES$
FROM sh.sales, sh.customers, sh.times, sh.channels, sh.countries, sh.products
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
  sales.channel_id= channels.channel_id AND sales.prod_id=products.prod_id AND
  customers.country_id=countries.country_id AND channels.channel_desc IN
 ('Direct Sales', 'Internet') AND times.calendar_month_desc IN
 ('2020-09', '2022-10') AND prod_name LIKE ('Tennis%') AND countries.country_iso_code IN
('GB', 'US')
GROUP BY ROLLUP(calendar_year, calendar_quarter_desc, calendar_month_desc),
  ROLLUP(country_region, country_subregion, countries.country_iso_code,
         cust_state_province, cust_city),
  ROLLUP(prod_category_desc, prod_subcategory_desc, prod_name);
```

The rollups in the `GROUP BY` specification generate the following groups, four for each dimension.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Table 21-3 Hierarchical CUBE Example**

| ROLLUP By Time | ROLLUP By Product | ROLLUP By Geography |
|---|---|---|
| year, quarter, month | category, subcategory, name | region, subregion, country, state, city |
| | | region, subregion, country, state |
| | | region, subregion, country |
| year, quarter | category, subcategory | region, subregion |
| year | category | region |
| all times | all products | all geographies |

The concatenated grouping sets specified in the previous SQL will take the ROLLUP aggregations listed in the table and perform a cross-product on them. The cross-product will create the 96 (4x4x6) aggregate groups needed for a hierarchical cube of the data. There are major advantages in using three ROLLUP expressions to replace what would otherwise require 96 grouping set expressions: the concise SQL is far less error-prone to develop and far easier to maintain, and it enables much better query optimization. You can picture how a cube with more dimensions and more levels would make the use of concatenated groupings even more advantageous.

See "Working with Hierarchical Cubes in SQL" for more information regarding hierarchical cubes.

# 21.8 Considerations when Using Aggregation in Data Warehouses

This section discusses the following topics.

- Hierarchy Handling in ROLLUP and CUBE
- Column Capacity in ROLLUP and CUBE
- HAVING Clause Used with GROUP BY Extensions
- ORDER BY Clause Used with GROUP BY Extensions
- Using Other Aggregate Functions with ROLLUP and CUBE
- Using In-Memory Aggregation

## 21.8.1 Hierarchy Handling in ROLLUP and CUBE

The ROLLUP and CUBE extensions work independently of any hierarchy metadata in your system. Their calculations are based entirely on the columns specified in the SELECT statement in which they appear. This approach enables CUBE and ROLLUP to be used whether or not hierarchy metadata is available. The simplest way to handle levels in hierarchical dimensions is by using the ROLLUP extension and indicating levels explicitly through separate columns. The following code shows a simple example of this with months rolled up to quarters and quarters rolled up to years.

**Example 21-13 ROLLUP and CUBE Hierarchy Handling**

```
SELECT calendar_year, calendar_quarter_number,
    calendar_month_number, SUM(amount_sold)
FROM sh.sales, sh.times, sh.products, sh.customers, sh.countries
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
WHERE sales.time_id=times.time_id
  AND sales.prod_id=products.prod_id
  AND customers.country_id = countries.country_id
  AND sales.cust_id=customers.cust_id
  AND prod_name LIKE ('%Bat')
  AND country_iso_code = 'GB' AND calendar_year=2020
GROUP BY ROLLUP(calendar_year, calendar_quarter_number, calendar_month_number);
```

| CALENDAR_YEAR | CALENDAR_QUARTER_NUMBER | CALENDAR_MONTH_NUMBER | SUM(AMOUNT_SOLD) |
|---|---|---|---|
| 2020 | 2 | 6 | 13138.29 |
| 2020 | 3 | 7 | 11411.52 |
| 2020 | 3 | 8 | 11869.9 |
| 2020 | 1 | 3 | 10333.08 |
| 2020 | 4 | 10 | 11720.44 |
| 2020 | 2 | 5 | 10056.33 |
| 2020 | 3 | 9 | 12752.25 |
| 2020 | 4 | 11 | 11231.48 |
| 2020 | 1 | 1 | 10652.43 |
| 2020 | 1 | 2 | 16004.55 |
| 2020 | 2 | 4 | 12226.67 |
| 2020 | 4 | 12 | 10500.77 |
| 2020 | 2 | | 35421.29 |
| 2020 | 3 | | 36033.67 |
| 2020 | 1 | | 36990.06 |
| 2020 | 4 | | 33452.69 |
| 2020 | | | 141897.71 |
| | | | 141897.71 |

## 21.8.2 Column Capacity in ROLLUP and CUBE

CUBE, ROLLUP, and GROUPING SETS do not restrict the GROUP BY clause column capacity. The GROUP BY clause, with or without the extensions, can work with up to 255 columns. However, the combinatorial explosion of CUBE makes it unwise to specify a large number of columns with the CUBE extension. Consider that a 20-column list for CUBE would create 2 to the 20 combinations in the result set. A very large CUBE list could strain system resources, so any such query must be tested carefully for performance and the load it places on the system.

## 21.8.3 HAVING Clause Used with GROUP BY Extensions

The HAVING clause of SELECT statements is unaffected by the use of GROUP BY. Note that the conditions specified in the HAVING clause apply to both the subtotal and non-subtotal rows of the result set. In some cases a query may need to exclude the subtotal rows or the non-subtotal rows from the HAVING clause. This can be achieved by using a GROUPING or GROUPING_ID function together with the HAVING clause. See #unique_743/unique_743_Connect_42_i1006706 and its associated SQL statement for an example.

## 21.8.4 ORDER BY Clause Used with GROUP BY Extensions

In many cases, a query must order the rows in a certain way, and this is done with the ORDER BY clause. The ORDER BY clause of a SELECT statement is unaffected by the use of GROUP BY, because the ORDER BY clause is applied after the GROUP BY calculations are complete.

Note that the ORDER BY specification makes no distinction between aggregate and non-aggregate rows of the result set. For instance, you might wish to list sales figures in declining order, but still have the subtotals at the end of each group. Simply ordering sales figures in descending sequence will not be sufficient, because that will place the subtotals (the largest


Copyright © 2001, 2025, Oracle and/or its affiliates.

values) at the start of each group. Therefore, it is essential that the columns in the ORDER BY clause include columns that differentiate aggregate from non-aggregate columns. This requirement means that queries using ORDER BY along with aggregation extensions to GROUP BY will generally need to use one or more of the GROUPING functions.

## 21.8.5 Using Other Aggregate Functions with ROLLUP and CUBE

The examples in this chapter show ROLLUP and CUBE used with the SUM function. While this is the most common type of aggregation, these extensions can also be used with all other functions available to the GROUP BY clause, for example, AVG, BIT_AND_AGG, BIT_OR_AGG, BIT_XOR_AGG, CHECKSUM, COUNT, KURTOSIS_POP, KURTOSIS_SAMP, MAX, MIN, SKEWNESS_POP, SKEWNESS_SAMP, STDDEV, and VARIANCE. COUNT, which is often needed in cross-tabular analyses, is likely to be the second most commonly used function.

## 21.8.6 Using In-Memory Aggregation

Analytic queries typically attempt to find patterns and trends by performing complex aggregations on data. In-memory aggregation uses KEY VECTOR and VECTOR GROUP BY operations to optimize query blocks involving aggregation and joins from a single large table to multiple small tables, such as in a typical star query. These operations use efficient in-memory arrays for joins and aggregation, and are especially effective when the underlying tables are stored in the In-Memory Column Store (IM column store).

The VECTOR GROUP BY transformation is an optimization transformation that enables efficient in-memory array-based aggregation. It accumulates aggregate values into in-memory arrays during table scans. This results in enhanced performance for joins and joins and aggregates.

The VECTOR GROUP BY transformation is a two-part process, similar to that of star transformation, that involves the following steps:

1. The dimension tables are scanned and any WHERE clause predicates are applied. A new data structure called a key vector is created based on the results of these scans.

   The key vector is similar to a bloom filter as it allows the join predicates to be applied as additional filter predicates during the scan of the fact table, but it also enables Oracle AI Database to conduct the GROUP BY or aggregation during the scan of the fact table instead of having to do it afterwards.

2. The results of the fact table scan are joined back to the temporary tables created as part of the key vector creation.

The combination of these two phases dramatically improves the efficiency of a multiple table join with complex aggregations. Both phases are visible in the execution plan of your query.

**Example 21-14 Example: Aggregation Using VECTOR GROUP BY Transformation**

Consider the following query that joins the products, customers, and times dimensions with the sales fact table:

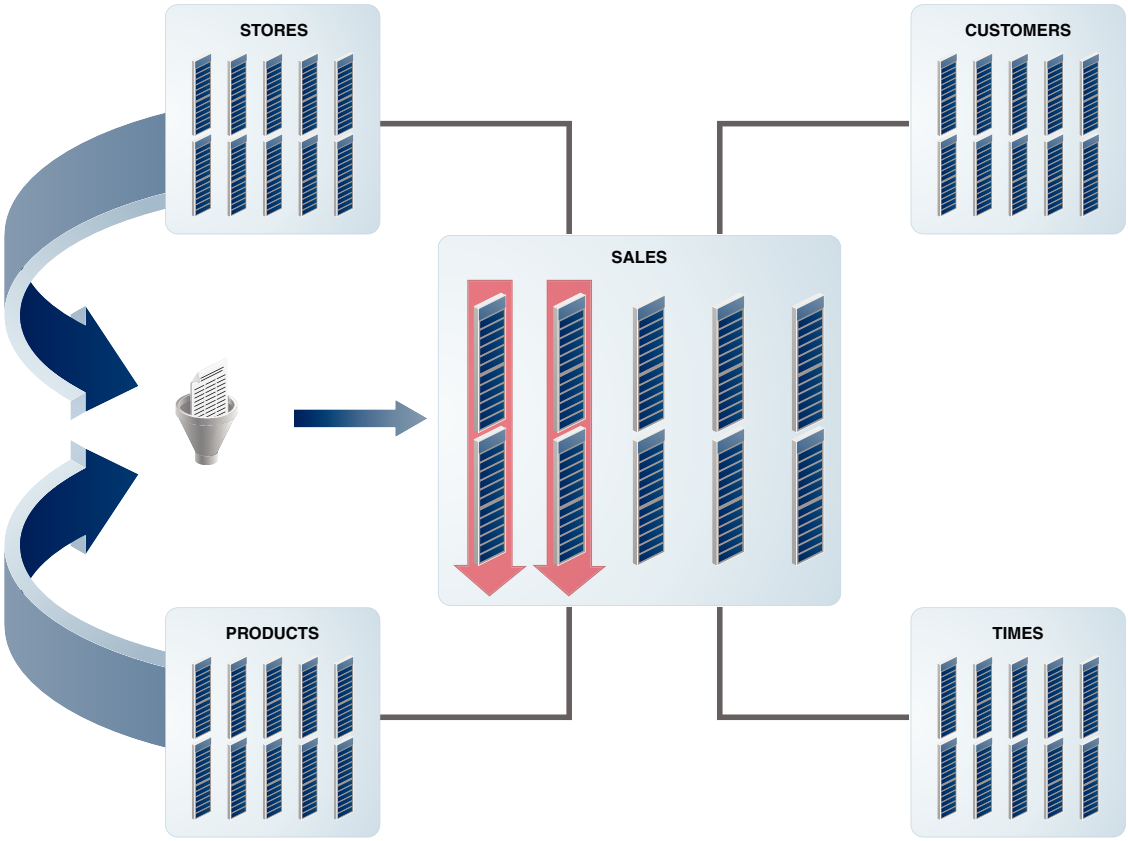
```
SELECT p.department_name, c.customer_id, t.fiscal_year, SUM(sales)
FROM PRODUCTS p, CUSTOMERS c, TIMES t, SALES s
WHERE p.product_id = s.product_id AND c.customer_id = s.customer_id
     AND t.time_id = s.time_id
GROUP BY p.department_name, c.customer_id, t.fiscal_year;
```

When the IM column store is configured, the Optimizer rewrites this query to use vector joins and VECTOR GROUP BY aggregation. Figure 21-2 describes how aggregation is performed using VECTOR GROUP BY. The predicates on the dimension tables PRODUCTS, CUSTOMERS, and TIMES


Copyright © 2001, 2025, Oracle and/or its affiliates.

are converted to filters on the fact table SALES. The GROUP BY is performed simultaneously with the scan of the SALES table by using in-memory arrays.

**Figure 21-2 VECTOR GROUP BY Using Oracle In-Memory Column Store**




# 21.9 Computation Using the WITH Clause

The WITH clause (formally known as subquery_factoring_clause) enables you to reuse the same query block in a SELECT statement when it occurs more than once within a complex query. WITH is a part of the SQL-99 standard. This is particularly useful when a query has multiple references to the same query block and there are joins and aggregations. Using the WITH clause, Oracle retrieves the results of a query block and stores them in the user's temporary tablespace. Depending on how your system is configured, the results may be stored in the shared temporary tablespace or local temporary tablespace. Note that Oracle AI Database does not support recursive use of the WITH clause. Note that Oracle AI Database supports recursive use of the WITH clause that may be used for such queries as are used with a bill of materials or expansion of parent-child hierarchies to parent-descendant hierarchies. See *Oracle Database SQL Language Reference* for more information.

> **Note**
>
> In previous releases, the term *temporary tablespace* referred to what is now called a *shared temporary tablespace*.


Copyright © 2001, 2025, Oracle and/or its affiliates.

The following query is an example of where you can improve performance and write SQL more simply by using the WITH clause. The query calculates the sum of sales for each channel and holds it under the name channel_summary. Then it checks each channel's sales total to see if any channel's sales are greater than one third of the total sales. By using the WITH clause, the channel_summary data is calculated just once, avoiding an extra scan through the large sales table.

**Example 21-15 WITH Clause**

```
WITH channel_summary AS (SELECT channels.channel_desc, SUM(amount_sold)
AS channel_total FROM sh.sales, sh.channels
WHERE sales.channel_id = channels.channel_id GROUP BY channels.channel_desc)
SELECT channel_desc, channel_total
FROM channel_summary WHERE channel_total > (SELECT SUM(channel_total) * 1/3
FROM channel_summary);

CHANNEL_DESC         CHANNEL_TOTAL
-------------------- -------------
Direct Sales            57875260.6
```

Note that this example could also be performed efficiently using the reporting aggregate functions described in SQL for Analysis and Reporting.

# 21.10 Working with Hierarchical Cubes in SQL

This section illustrates examples of working with hierarchical cubes. It contains the following topics:

- Specifying Hierarchical Cubes in SQL
- Querying Hierarchical Cubes in SQL

## 21.10.1 Specifying Hierarchical Cubes in SQL

Oracle AI Database can specify hierarchical cubes in a simple and efficient SQL query. These hierarchical cubes represent the logical cubes referred to in many analytical SQL products. To specify data in the form of hierarchical cubes, you can use one of the extensions to the GROUP BY clause, concatenated grouping sets, to generate the aggregates needed for a hierarchical cube of data. By using concatenated rollup (rolling up along the hierarchy of each dimension and then concatenate them across multiple dimensions), you can generate all the aggregations needed by a hierarchical cube.

**Example 21-16 Concatenated ROLLUP**

The following shows the GROUP BY clause needed to create a hierarchical cube for a 2-dimensional example similar to Example 21-12. The following simple syntax performs a concatenated rollup:

```
GROUP BY ROLLUP(year, quarter, month), ROLLUP(Division, brand, item)
```

This concatenated rollup takes the ROLLUP aggregations similar to those listed in Table 21-3 in the prior section and performs a cross-product on them. The cross-product will create the 16 (4x4) aggregate groups needed for a hierarchical cube of the data.

## 21.10.2 Querying Hierarchical Cubes in SQL

Analytic applications treat data as cubes, but they want only certain slices and regions of the cube. Concatenated rollup (hierarchical cube) enables relational data to be treated as cubes.


Copyright © 2001, 2025, Oracle and/or its affiliates.

To handle complex analytic queries, the fundamental technique is to enclose a hierarchical cube query in an outer query that specifies the exact slice needed from the cube. Oracle AI Database optimizes the processing of hierarchical cubes nested inside slicing queries. By applying many powerful algorithms, these queries can be processed at unprecedented speed and scale. This enables SQL analytical tools and applications to use a consistent style of queries to handle the most complex questions.

**Example 21-17 Hierarchical Cube Query**

Consider the following analytic query. It consists of a hierarchical cube query nested in a slicing query.

```
SELECT month, division, sum_sales FROM
  (SELECT year, quarter, month, division, brand, item, SUM(sales) sum_sales,
      GROUPING_ID(grouping-columns) gid
   FROM sales, products, time
   WHERE join-condition
   GROUP BY ROLLUP(year, quarter, month),
            ROLLUP(division, brand, item))
WHERE division = 25 AND month = 200201 AND gid = gid-for-Division-Month;
```

The inner hierarchical cube specified defines a simple cube, with two dimensions and four levels in each dimension. It would generate 16 groups (4 Time levels * 4 Product levels). The `GROUPING_ID` function in the query identifies the specific group each row belongs to, based on the aggregation level of the *grouping-columns* in its argument.

The outer query applies the constraints needed for our specific query, limiting Division to a value of 25 and Month to a value of 200201 (representing January 2002 in this case). In conceptual terms, it slices a small chunk of data from the cube. The outer query's constraint on the `GID` column, indicated in the query by *gid-for-division-month* would be the value of a key indicating that the data is grouped as a combination of `division` and `month`. The `GID` constraint selects only those rows that are aggregated at the level of a `GROUP BY` month, division clause.

Oracle AI Database removes unneeded aggregation groups from query processing based on the outer query conditions. The outer conditions of the previous query limit the result set to a single group aggregating `division` and `month`. Any other groups involving `year`, `month`, `brand`, and `item` are unnecessary here. The group pruning optimization recognizes this and transforms the query into:

```
SELECT month, division, sum_sales
FROM (SELECT  null, null,  month, division, null, null, SUM(sales) sum_sales,
      GROUPING_ID(grouping-columns) gid
      FROM sales, products, time WHERE join-condition
   GROUP BY month, division)
WHERE division = 25 AND month = 200201 AND gid = gid-for-Division-Month;
```

The bold items highlight the changed SQL. The inner query now has a simple `GROUP BY` clause of `month`, `division`. The columns `year`, `quarter`, `brand`, and `item` have been converted to null to match the simplified `GROUP BY` clause. Because the query now requests just one group, fifteen out of sixteen groups are removed from the processing, greatly reducing the work. For a cube with more dimensions and more levels, the savings possible through group pruning can be far greater. Note that the group pruning transformation works with all the `GROUP BY` extensions: `ROLLUP`, `CUBE`, and `GROUPING SETS`.

While the optimizer has simplified the previous query to a simple `GROUP BY`, faster response times can be achieved if the group is precomputed and stored in a materialized view. Because online analytical queries can ask for any slice of the cube many groups may need to be precomputed and stored in a materialized view. This is discussed in the next section.

This section contains the following topics:


Copyright © 2001, 2025, Oracle and/or its affiliates.

- SQL for Creating Materialized Views to Store Hierarchical Cubes
- Examples of Hierarchical Cube Materialized Views

## 21.10.2.1 SQL for Creating Materialized Views to Store Hierarchical Cubes

Analytical SQL requires fast response times for multiple users, and this in turn demands that significant parts of a cube be precomputed and held in materialized views.

Data warehouse designers can choose exactly how much data to materialize. A data warehouse can have the full hierarchical cube materialized. While this will take the most storage space, it ensures quick response for any query within the cube. Alternatively, a data warehouse could have just partial materialization, saving storage space, but allowing only a subset of possible queries to be answered at highest speed. If the queries cover the full range of aggregate groupings possible in its data set, it may be best to materialize the whole hierarchical cube.

This means that each dimension's aggregation hierarchy is precomputed in combination with each of the other dimensions. Naturally, precomputing a full hierarchical cube requires more disk space and higher creation and refresh times than a small set of aggregate groups. The trade-off in processing time and disk space versus query performance must be considered before deciding to create it. An additional possibility you could consider is to use data compression to lessen your disk space requirements.

**See Also**

- *Oracle Database SQL Language Reference* for table compression syntax and restrictions
- *Oracle Database Administrator's Guide* for further details about table compression
- "About Storage And Table Compression for Materialized Views" for details regarding table compression

## 21.10.2.2 Examples of Hierarchical Cube Materialized Views

This section shows complete and partial hierarchical cube materialized views. Many of the examples are meant to illustrate capabilities, and do not actually run.

In a data warehouse where rolling window scenario is very common, it is recommended that you store the hierarchical cube in multiple materialized views - one for each level of time you are interested in. Hence, a complete hierarchical cube will be stored in four materialized views: `sales_hierarchical_mon_cube_mv`, `sales_hierarchical_qtr_cube_mv`, `sales_hierarchical_yr_cube_mv`, and `sales_hierarchical_all_cube_mv`.

The following statements create a complete hierarchical cube stored in a set of three composite partitioned and one list partitioned materialized view.

**Example 21-18 Complete Hierarchical Cube Materialized View**

```
CREATE MATERIALIZED VIEW sales_hierarchical_mon_cube_mv
PARTITION BY RANGE (mon)
SUBPARTITION BY LIST (gid)
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT calendar_year yr, calendar_quarter_desc qtr, calendar_month_desc mon,
    country_id, cust_state_province, cust_city,
    prod_category, prod_subcategory, prod_name,
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
    GROUPING_ID(calendar_year, calendar_quarter_desc, calendar_month_desc,
                country_id, cust_state_province, cust_city,
                prod_category, prod_subcategory, prod_name) gid,
    SUM(amount_sold) s_sales, COUNT(amount_sold) c_sales,
    COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id AND s.time_id = t.time_id
GROUP BY calendar_year, calendar_quarter_desc, calendar_month_desc,
  ROLLUP(country_id, cust_state_province, cust_city),
  ROLLUP(prod_category, prod_subcategory, prod_name),
...;

CREATE MATERIALIZED VIEW sales_hierarchical_qtr_cube_mv
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT calendar_year yr, calendar_quarter_desc qtr,
    country_id, cust_state_province, cust_city,
    prod_category, prod_subcategory, prod_name,
    GROUPING_ID(calendar_year, calendar_quarter_desc,
                country_id, cust_state_province, cust_city,
                prod_category, prod_subcategory, prod_name) gid,
    SUM(amount_sold) s_sales, COUNT(amount_sold) c_sales,
    COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id
      AND s.time_id = t.time_id
GROUP BY calendar_year, calendar_quarter_desc,
  ROLLUP(country_id, cust_state_province, cust_city),
  ROLLUP(prod_category, prod_subcategory, prod_name),
PARTITION BY RANGE (qtr)
 SUBPARTITION BY LIST (gid)
...;

CREATE MATERIALIZED VIEW sales_hierarchical_yr_cube_mv
PARTITION BY RANGE (year)
SUBPARTITION BY LIST (gid)
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT calendar_year yr, country_id, cust_state_province, cust_city,
    prod_category, prod_subcategory, prod_name,
    GROUPING_ID(calendar_year, country_id, cust_state_province, cust_city,
                prod_category, prod_subcategory, prod_name) gid,
    SUM(amount_sold) s_sales, COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id AND s.time_id = t.time_id
GROUP BY calendar_year,
  ROLLUP(country_id, cust_state_province, cust_city),
  ROLLUP(prod_category, prod_subcategory, prod_name),
...;

CREATE MATERIALIZED VIEW sales_hierarchical_all_cube_mv
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT country_id, cust_state_province, cust_city,
    prod_category, prod_subcategory, prod_name,
    GROUPING_ID(country_id, cust_state_province, cust_city,
                prod_category, prod_subcategory, prod_name) gid,
    SUM(amount_sold) s_sales, COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id AND s.time_id = t.time_id
GROUP BY ROLLUP(country_id, cust_state_province, cust_city),
         ROLLUP(prod_category, prod_subcategory, prod_name),
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
PARTITION BY LIST (gid)
...;
```

This allows use of PCT refresh on the materialized views `sales_hierarchical_mon_cube_mv`, `sales_hierarchical_qtr_cube_mv`, and `sales_hierarchical_yr_cube_mv` on partition maintenance operations to sales table. PCT refresh can also be used when there have been significant changes to the base table and log based fast refresh is estimated to be slower than PCT refresh. You can just specify the method as force (`method => '?'`) in to refresh sub-programs in the `DBMS_MVIEW` package and Oracle AI Database will pick the best method of refresh. See "About Partition Change Tracking (PCT) Refresh for Materialized Views" for more information regarding PCT refresh.

Because `sales_hierarchical_qtr_cube_mv` does not contain any column from `times` table, PCT refresh is not enabled on it. But, you can still call refresh sub-programs in the `DBMS_MVIEW` package with method as force (`method => '?'`) and Oracle AI Database will pick the best method of refresh.

If you are interested in a partial cube (that is, a subset of groupings from the complete cube), then Oracle recommends storing the cube as a "federated cube". A federated cube stores each grouping of interest in a separate materialized view.

```
CREATE MATERIALIZED VIEW sales_mon_city_prod_mv
PARTITION BY RANGE (mon)
...
BUILD DEFERRED
REFRESH FAST ON DEMAND
  USING TRUSTED CONSTRAINTS
ENABLE QUERY REWRITE AS
SELECT calendar_month_desc mon, cust_city, prod_name, SUM(amount_sold) s_sales,
       COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id
AND s.time_id = t.time_id
GROUP BY calendar_month_desc, cust_city, prod_name;

CREATE MATERIALIZED VIEW sales_qtr_city_prod_mv
PARTITION BY RANGE (qtr)
...
BUILD DEFERRED
REFRESH FAST ON DEMAND
  USING TRUSTED CONSTRAINTS
ENABLE QUERY REWRITE AS
SELECT calendar_quarter_desc qtr, cust_city, prod_name,SUM(amount_sold) s_sales,
COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id =p.prod_id AND s.time_id = t.time_id
GROUP BY calendar_quarter_desc, cust_city, prod_name;

CREATE MATERIALIZED VIEW sales_yr_city_prod_mv
PARTITION BY RANGE (yr)
...
BUILD DEFERRED
REFRESH FAST ON DEMAND
USING TRUSTED CONSTRAINTS
ENABLE QUERY REWRITE AS
SELECT calendar_year yr, cust_city, prod_name, SUM(amount_sold) s_sales,
       COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id =p.prod_id AND s.time_id = t.time_id
GROUP BY calendar_year, cust_city, prod_name;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
CREATE MATERIALIZED VIEW sales_mon_city_scat_mv
PARTITION BY RANGE (mon)
...
BUILD DEFERRED
REFRESH FAST ON DEMAND
  USING TRUSTED CONSTRAINTS
ENABLE QUERY REWRITE AS
SELECT calendar_month_desc mon, cust_city, prod_subcategory,
       SUM(amount_sold) s_sales, COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id =p.prod_id AND s.time_id =t.time_id
GROUP BY calendar_month_desc, cust_city, prod_subcategory;

CREATE MATERIALIZED VIEW sales_qtr_city_cat_mv
PARTITION BY RANGE (qtr)
...
BUILD DEFERRED
REFRESH FAST ON DEMAND
  USING TRUSTED CONSTRAINTS
ENABLE QUERY REWRITE AS
SELECT calendar_quarter_desc qtr, cust_city, prod_category cat,
       SUM(amount_sold) s_sales, COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id =p.prod_id AND s.time_id =t.time_id
GROUP BY calendar_quarter_desc, cust_city, prod_category;

CREATE MATERIALIZED VIEW sales_yr_city_all_mv
PARTITION BY RANGE (yr)
...
BUILD DEFERRED
REFRESH FAST ON DEMAND
  USING TRUSTED CONSTRAINTS
ENABLE QUERY REWRITE AS
SELECT calendar_year yr, cust_city, SUM(amount_sold) s_sales,
       COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id AND s.time_id = t.time_id
GROUP BY calendar_year, cust_city;
```

These materialized views can be created as `BUILD DEFERRED` and then, you can execute `DBMS_MVIEW.REFRESH_DEPENDENT(number_of_failures, 'SALES', 'C' ...)` so that the complete refresh of each of the materialized views defined on the detail table `sales` is scheduled in the most efficient order. See "Scheduling Refresh of Materialized Views" for more information.

Because each of these materialized views is partitioned on the time level (month, quarter, or year) present in the `SELECT` list, PCT is enabled on `sales` table for each one of them, thus providing an opportunity to apply PCT refresh method in addition to `FAST` and `COMPLETE` refresh methods.


Copyright © 2001, 2025, Oracle and/or its affiliates.

22

# SQL for Pattern Matching

Recognizing patterns in a sequence of rows has been a capability that was widely desired, but not possible with SQL until now. There were many workarounds, but these were difficult to write, hard to understand, and inefficient to execute. Beginning in Oracle Database 12*c*, you can use the `MATCH_RECOGNIZE` clause to achieve this capability in native SQL that executes efficiently. This chapter discusses how to do this, and includes the following sections:


- Overview of Pattern Matching in Data Warehouses
- Basic Topics in Pattern Matching
- Pattern Matching Details
- Advanced Topics in Pattern Matching
- Rules and Restrictions in Pattern Matching
- Examples of Pattern Matching


## 22.1 Overview of Pattern Matching in Data Warehouses

Pattern matching in SQL is performed using the `MATCH_RECOGNIZE` clause. `MATCH_RECOGNIZE` enables you to do the following tasks:

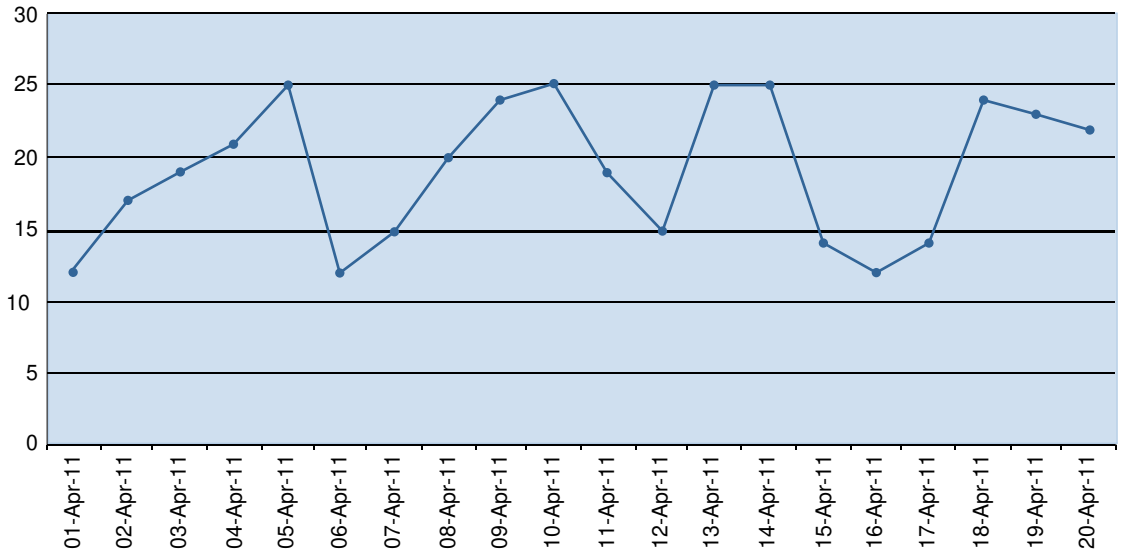
- Logically partition and order the data that is used in the `MATCH_RECOGNIZE` clause with its `PARTITION BY` and `ORDER BY` clauses.
- Define patterns of rows to seek using the `PATTERN` clause of the `MATCH_RECOGNIZE` clause. These patterns use regular expression syntax, a powerful and expressive feature, applied to the pattern variables you define.
- Specify the logical conditions required to map a row to a row pattern variable in the `DEFINE` clause.
- Define measures, which are expressions usable in other parts of the SQL query, in the `MEASURES` clause.

As a simple case of pattern matching, consider the stock price chart illustrated in Figure 22-1.



Copyright © 2001, 2025, Oracle and/or its affiliates.

**Figure 22-1 Stock Chart**




Pattern matching can let you identify price patterns, such as V-shapes and W-shapes illustrated in Figure 22-1, along with performing many types of calculations. For example, your calculations might include the count of observations or the average value on a downward or upward slope.

This section contains the following topics:

- Why Use Pattern Matching?
- How Data is Processed in Pattern Matching
- About Pattern Matching Special Capabilities

## 22.1.1 Why Use Pattern Matching?

The ability to recognize patterns found across multiple rows is important for many kinds of work. Examples include all kinds of business processes driven by sequences of events, such as security applications, where unusual behavior must be detected, and financial applications, where you seek patterns of pricing, trading volume, and other behavior. Other common uses are fraud detection applications and sensor data analysis. One term that describes this general area is complex event processing, and pattern matching is a powerful aid to this activity.

Now consider the query in Example 22-1. It uses the stock price shown in Figure 22-1, which you can load into your database with the `CREATE` and `INSERT` statements that follow. The query finds all cases where stock prices dipped to a bottom price and then rose. This is generally called a V-shape. Before studying the query, look at the output. There are only three rows because the code was written to report just one row per match, and three matches were found. The `MATCH_RECOGNIZE` clause lets you choose between showing one row per match and all rows per match. In this example, the shorter output of one row per match is used.

**Example 22-1 Pattern Match: Simple V-Shape with 1 Row Output per Match**

```
CREATE TABLE Ticker (SYMBOL VARCHAR2(10), tstamp DATE, price NUMBER);

INSERT INTO Ticker VALUES('ACME', '01-Apr-22', 12);
INSERT INTO Ticker VALUES('ACME', '02-Apr-22', 17);
INSERT INTO Ticker VALUES('ACME', '03-Apr-22', 19);
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
INSERT INTO Ticker VALUES('ACME', '04-Apr-22', 21);
INSERT INTO Ticker VALUES('ACME', '05-Apr-22', 25);
INSERT INTO Ticker VALUES('ACME', '06-Apr-22', 12);
INSERT INTO Ticker VALUES('ACME', '07-Apr-22', 15);
INSERT INTO Ticker VALUES('ACME', '08-Apr-22', 20);
INSERT INTO Ticker VALUES('ACME', '09-Apr-22', 24);
INSERT INTO Ticker VALUES('ACME', '10-Apr-22', 25);
INSERT INTO Ticker VALUES('ACME', '11-Apr-22', 19);
INSERT INTO Ticker VALUES('ACME', '12-Apr-22', 15);
INSERT INTO Ticker VALUES('ACME', '13-Apr-22', 25);
INSERT INTO Ticker VALUES('ACME', '14-Apr-22', 25);
INSERT INTO Ticker VALUES('ACME', '15-Apr-22', 14);
INSERT INTO Ticker VALUES('ACME', '16-Apr-22', 12);
INSERT INTO Ticker VALUES('ACME', '17-Apr-22', 14);
INSERT INTO Ticker VALUES('ACME', '18-Apr-22', 24);
INSERT INTO Ticker VALUES('ACME', '19-Apr-22', 23);
INSERT INTO Ticker VALUES('ACME', '20-Apr-22', 22);

SELECT *
FROM Ticker MATCH_RECOGNIZE (
     PARTITION BY symbol
     ORDER BY tstamp
     MEASURES  STRT.tstamp AS start_tstamp,
               LAST(DOWN.tstamp) AS bottom_tstamp,
               LAST(UP.tstamp) AS end_tstamp
     ONE ROW PER MATCH
     AFTER MATCH SKIP TO LAST UP
     PATTERN (STRT DOWN+ UP+)
     DEFINE
        DOWN AS DOWN.price < PREV(DOWN.price),
        UP AS UP.price > PREV(UP.price)
     ) MR
ORDER BY MR.symbol, MR.start_tstamp;

SYMBOL    START_TSTAMP    BOTTOM_TSTAMP    END_TSTAMP
_________ _______________ ________________ _____________
ACME      05-APR-22       06-APR-22        10-APR-22
ACME      10-APR-22       12-APR-22        13-APR-22
ACME      14-APR-22       16-APR-22        18-APR-22
```

What does this query do? The following explains each line in the `MATCH_RECOGNIZE` clause:

- `PARTITION BY` divides the data from the `Ticker` table into logical groups where each group contains one stock symbol.
- `ORDER BY` orders the data within each logical group by `tstamp`.
- `MEASURES` defines three measures: the timestamp at the beginning of a V-shape (`start_tstamp`), the timestamp at the bottom of a V-shape (`bottom_tstamp`), and the timestamp at the end of the a V-shape (`end_tstamp`). The `bottom_tstamp` and `end_tstamp` measures use the `LAST( )` function to ensure that the values retrieved are the final value of the timestamp within each pattern match.
- `ONE ROW PER MATCH` means that for every pattern match found, there will be one row of output.
- `AFTER MATCH SKIP TO LAST UP` means that whenever you find a match you restart your search at the row that is the last row of the `UP` pattern variable. A pattern variable is a variable used in a `MATCH_RECOGNIZE` statement, and is defined in the `DEFINE` clause.
- `PATTERN (STRT DOWN+ UP+)` says that the pattern you are searching for has three pattern variables: `STRT`, `DOWN`, and `UP`. The plus sign (`+`) after `DOWN` and `UP` means that at least one


Copyright © 2001, 2025, Oracle and/or its affiliates.

row must be mapped to each of them. The pattern defines a regular expression, which is a highly expressive way to search for patterns.

- `DEFINE` gives us the conditions that must be met for a row to map to your row pattern variables `STRT`, `DOWN`, and `UP`. Because there is no condition for `STRT`, any row can be mapped to `STRT`. Why have a pattern variable with no condition? You use it as a starting point for testing for matches. Both `DOWN` and `UP` take advantage of the `PREV()` function, which lets them compare the price in the current row to the price in the prior row. `DOWN` is matched when a row has a lower price than the row that preceded it, so it defines the downward (left) leg of our V-shape. A row can be mapped to `UP` if the row has a higher price than the row that preceded it.

The following two figures will help you better understand the results returned by Example 22-1. Figure 22-2 shows the dates mapped to specific pattern variables, as specified in the `PATTERN` clause. After the mappings of pattern variables to dates are available, that information is used by the `MEASURES` clause to calculate the measure values. The measures results are shown in Figure 22-3.

**Figure 22-2 Stock Chart Illustrating Which Dates are Mapped to Which Pattern Variables**

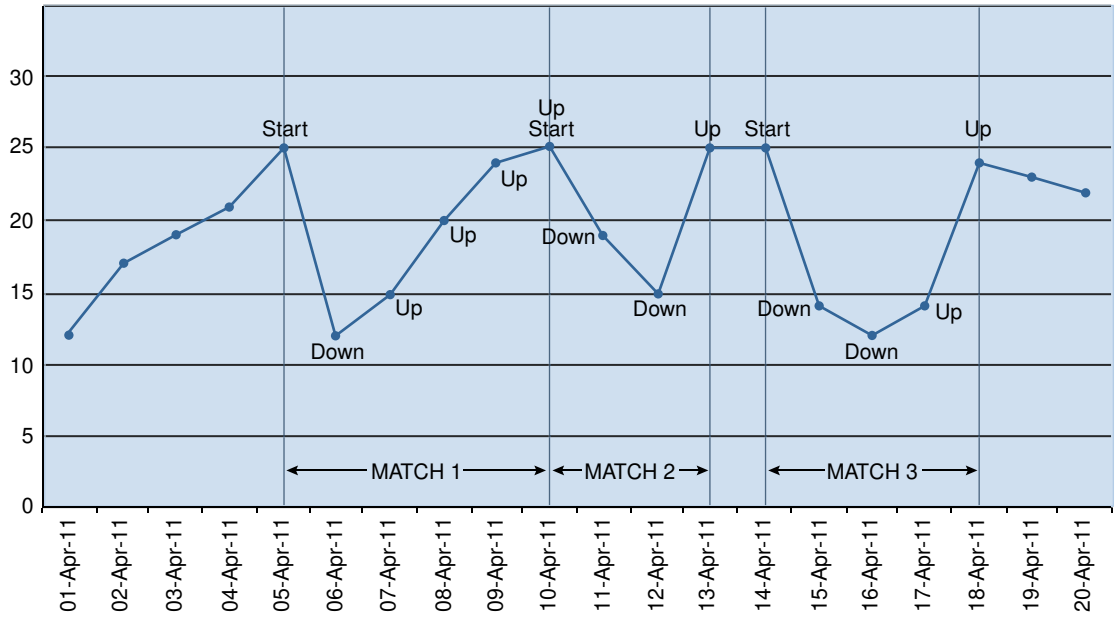
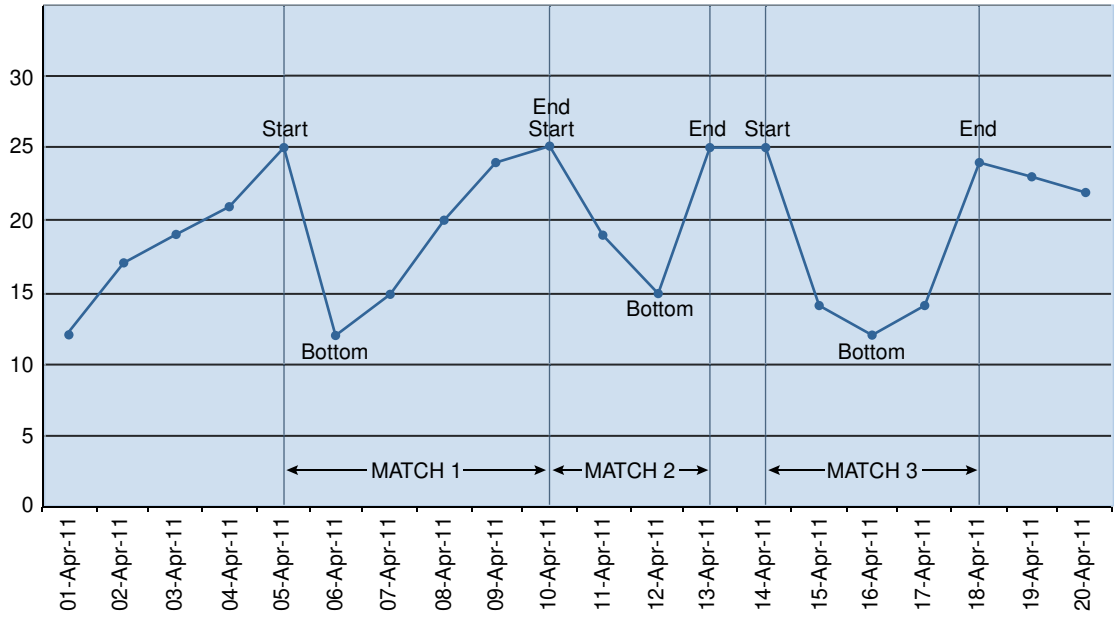



Figure 22-2 labels every date mapped to a pattern variable. The mapping is based on the pattern specified in the `PATTERN` clause and the logical conditions specified in the `DEFINE` clause. The thin vertical lines show the borders of the three matches that were found for the pattern. In each match, the first date has the `STRT` pattern variable mapped to it (labeled as *Start*), followed by one or more dates mapped to the `DOWN` pattern variable, and finally, one or more dates mapped to the `UP` pattern variable.

Because you specified `AFTER MATCH SKIP TO LAST UP` in the query, two adjacent matches can share a row. That means a single date can have two variables mapped to it. For example, 10-April has both the pattern variables `UP` and `STRT` mapped to it: April 10 is the end of Match 1 and the start of Match 2.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Figure 22-3 Stock Chart Showing the Dates to Which the Measures Correspond**




In Figure 22-3, the labels are solely for the measures defined in the `MEASURES` clause of the query: `START` (`start_tstamp` in the query), `BOTTOM` (`bottom_tstamp` in the query), and `END` (`end_tstamp` in the query). As in Figure 22-2, the thin vertical lines show the borders of the three matches found for the pattern. Every match has a `Start` date, a `Bottom` date, and an `End` date. As with Figure 22-2, the date 10-April is found in two matches: it is the `END` measure for Match 1 and the `START` measure for Match 2. The labeled dates of Figure 22-3 show which dates correspond to the measure definitions, which are based on the pattern variable mappings shown in Figure 22-2.

Note that the dates labeled in Figure 22-3 correspond to the nine dates shown earlier in the output of the example. The first row of the output has the dates shown in Match 1, the second row of the output has the dates shown in Match 2, and the third row of the output has the dates shown in Match 3.

## 22.1.2 How Data is Processed in Pattern Matching

The `MATCH_RECOGNIZE` clause performs these steps:

1. The row pattern input table is partitioned according to the `PARTITION BY` clause. Each partition consists of the set of rows of the input table that have the same value on the partitioning columns.
2. Each row pattern partition is ordered according to the `ORDER BY` clause.
3. Each ordered row pattern partition is searched for matches to the `PATTERN`.
4. Pattern matching operates by seeking the match at the earliest row, considering the rows in a row pattern partition in the order specified by the `ORDER BY` clause.

   Pattern matching in a sequence of rows is an incremental process, with one row after another examined to see if it fits the pattern. With this incremental processing model, at any step until the complete pattern is recognized, you only have a partial match, and you do not know what rows might be added in the future, nor to what variables those future rows might be mapped.


Copyright © 2001, 2025, Oracle and/or its affiliates.

If no match is found at the earliest row, the search moves to the next row in the partition, checking if a match can be found starting with that row.

5. After a match is found, row pattern matching calculates the row pattern measure columns, which are expressions defined by the `MEASURES` clause.
6. Using `ONE ROW PER MATCH`, as shown in the first example, pattern matching generates one row for each match that is found. If you use `ALL ROWS PER MATCH`, every row that is matched is included in the pattern match output.
7. The `AFTER MATCH SKIP` clause determines where row pattern matching resumes within a row pattern partition after a non-empty match is found. In the previous example, row pattern matching resumes at the last row of the match found (`AFTER MATCH SKIP TO LAST UP`).

## 22.1.3 About Pattern Matching Special Capabilities

The capabilities are:

- Regular expressions are a robust and long-established way for systems to search for patterns in data. The regular expression features of the language Perl were adopted as the design target for pattern matching rules, and Oracle Database 12*c* Release 1, implements a subset of those rules for pattern matching.
- Oracle's regular expressions differ from typical regular expressions in that the row pattern variables are defined by Boolean conditions rather than characters or sets of characters.
- While pattern matching uses the notation of regular expressions to express patterns, it is actually a richer capability, because the pattern variables may be defined to depend upon the way previous rows were mapped to row pattern variables. The `DEFINE` clause enables pattern variables to be built upon other pattern variables.
- Subqueries are permitted in the definition of row pattern variables and the definition of measures.

# 22.2 Basic Topics in Pattern Matching

This section discusses:

- Basic Examples of Pattern Matching
- Tasks and Keywords in Pattern Matching
- Pattern Matching Syntax

## 22.2.1 Basic Examples of Pattern Matching

This section includes some basic examples for matching patterns.

**Example 22-2 Pattern Match for a Simple V-Shape with All Rows Output per Match**

This example uses the table `Ticker` created in the previous example.

```
SELECT *
FROM Ticker MATCH_RECOGNIZE (
     PARTITION BY symbol
     ORDER BY tstamp
     MEASURES  STRT.tstamp AS start_tstamp,
               FINAL LAST(DOWN.tstamp) AS bottom_tstamp,
               FINAL LAST(UP.tstamp) AS end_tstamp,
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
             MATCH_NUMBER() AS match_num,
             CLASSIFIER() AS var_match
     ALL ROWS PER MATCH
     AFTER MATCH SKIP TO LAST UP
     PATTERN (STRT DOWN+ UP+)
     DEFINE
        DOWN AS DOWN.price < PREV(DOWN.price),
        UP AS UP.price > PREV(UP.price)
     ) MR
ORDER BY MR.symbol, MR.match_num, MR.tstamp;

SYMBOL     TSTAMP       START_TSTAMP    BOTTOM_TSTAMP    END_TSTAMP        MATCH_NUM
VAR_MATCH       PRICE
_________ ___________ _______________ _________________ _______________ ____________
___________ ________
ACME       05-APR-22    05-APR-22       06-APR-22        10-APR-22                 1
STRT               25
ACME       06-APR-22    05-APR-22       06-APR-22        10-APR-22                 1
DOWN               12
ACME       07-APR-22    05-APR-22       06-APR-22        10-APR-22                 1
UP                 15
ACME       08-APR-22    05-APR-22       06-APR-22        10-APR-22                 1
UP                 20
ACME       09-APR-22    05-APR-22       06-APR-22        10-APR-22                 1
UP                 24
ACME       10-APR-22    05-APR-22       06-APR-22        10-APR-22                 1
UP                 25
ACME       10-APR-22    10-APR-22       12-APR-22        13-APR-22                 2
STRT               25
ACME       11-APR-22    10-APR-22       12-APR-22        13-APR-22                 2
DOWN               19
ACME       12-APR-22    10-APR-22       12-APR-22        13-APR-22                 2
DOWN               15
ACME       13-APR-22    10-APR-22       12-APR-22        13-APR-22                 2
UP                 25
ACME       14-APR-22    14-APR-22       16-APR-22        18-APR-22                 3
STRT               25
ACME       15-APR-22    14-APR-22       16-APR-22        18-APR-22                 3
DOWN               14
ACME       16-APR-22    14-APR-22       16-APR-22        18-APR-22                 3
DOWN               12
ACME       17-APR-22    14-APR-22       16-APR-22        18-APR-22                 3
UP                 14
ACME       18-APR-22    14-APR-22       16-APR-22        18-APR-22                 3
UP                 24


15 rows selected.
```

What does this query do? It is similar to the query in Example 22-1 except for items in the `MEASURES` clause, the change to `ALL ROWS PER MATCH`, and a change to the `ORDER BY` at the end of the query. In the `MEASURES` clause, there are these additions:

- `MATCH_NUMBER() AS match_num`

  Because this example gives multiple rows per match, you need to know which rows are members of which match. `MATCH_NUMBER` assigns the same number to each row of a specific match. For instance, all the rows in the first match found in a row pattern partition are assigned the `match_num` value of 1. Note that match numbering starts over again at 1 in each row pattern partition.

- `CLASSIFIER() AS var_match`


Copyright © 2001, 2025, Oracle and/or its affiliates.

To know which rows map to which variable, use the `CLASSIFIER` function. In this example, some rows will map to the `STRT` variable, some rows the `DOWN` variable, and others to the `UP` variable.

- `FINAL LAST()`

  By specifying `FINAL` and using the `LAST()` function for `bottom_tstamp`, every row inside each match shows the same date for the bottom of its V-shape. Likewise, applying `FINAL LAST()` to the `end_tstamp` measure makes every row in each match show the same date for the end of its V-shape. Without this syntax, the dates shown would be the running value for each row.

Changes were made in two other lines:

- `ALL ROWS PER MATCH` - While Example 22-1 gave a summary with just 1 row about each match using the line `ONE ROW PER MATCH`, this example asks to show every row of each match.
- `ORDER BY` on the last line - This was changed to take advantage of the `MATCH_NUM`, so all rows in the same match are together and in chronological order.

Note that the row for April 10 appears twice because it is in two pattern matches: it is the last day of the first match and the first day of the second match.

### Example 22-3 Pattern Match with an Aggregate on a Variable

Example 22-3 highlights the use of aggregate functions in pattern matching queries.

```
SELECT *
FROM Ticker MATCH_RECOGNIZE (
  PARTITION BY symbol
  ORDER BY tstamp
  MEASURES
    MATCH_NUMBER() AS match_num,
    CLASSIFIER() AS var_match,
    FINAL COUNT(UP.tstamp) AS up_days,
    FINAL COUNT(tstamp) AS total_days,
    RUNNING COUNT(tstamp) AS cnt_days,
    price - STRT.price AS price_dif
  ALL ROWS PER MATCH
  AFTER MATCH SKIP TO LAST UP
  PATTERN (STRT DOWN+ UP+)
  DEFINE
    DOWN AS DOWN.price < PREV(DOWN.price),
    UP AS UP.price > PREV(UP.price)
  ) MR
ORDER BY MR.symbol, MR.match_num, MR.tstamp;

SYMBOL TSTAMP    MATCH_NUM VAR_ UP_DAYS TOTAL_DAYS CNT_DAYS PRICE_DIF PRICE
------ --------- --------- ---- ------- ---------- -------- --------- -----
ACME   05-APR-11         1 STRT       4          6        1         0    25
ACME   06-APR-11         1 DOWN       4          6        2       -13    12
ACME   07-APR-11         1 UP         4          6        3       -10    15
ACME   08-APR-11         1 UP         4          6        4        -5    20
ACME   09-APR-11         1 UP         4          6        5        -1    24
ACME   10-APR-11         1 UP         4          6        6         0    25
ACME   10-APR-11         2 STRT       1          4        1         0    25
ACME   11-APR-11         2 DOWN       1          4        2        -6    19
ACME   12-APR-11         2 DOWN       1          4        3       -10    15
ACME   13-APR-11         2 UP         1          4        4         0    25
ACME   14-APR-11         3 STRT       2          5        1         0    25
ACME   15-APR-11         3 DOWN       2          5        2       -11    14
ACME   16-APR-11         3 DOWN       2          5        3       -13    12
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
ACME   17-APR-11         3 UP         2          5        4       -11    14
ACME   18-APR-11         3 UP         2          5        5        -1    24

15 rows selected.
```

What does this query do? It builds on Example 22-2 by adding three measures that use the aggregate function `COUNT()`. It also adds a measure showing how an expression can use a qualified and unqualified column.

- The `up_days` measure (with `FINAL COUNT`) shows the number of days mapped to the `UP` pattern variable within each match. You can verify this by counting the `UP` labels for each match in Figure 22-2.
- The `total_days` measure (also with `FINAL COUNT`) introduces the use of unqualified columns. Because this measure specified the `FINAL count(tstamp)` with no pattern variable to qualify the `tstamp` column, it returns the count of all rows included in a match.
- The `cnt_days` measure introduces the `RUNNING` keyword. This measure gives a running count that helps distinguish among the rows in a match. Note that it also has no pattern variable to qualify the `tstamp` column, so it applies to all rows of a match. You do not need to use the `RUNNING` keyword explicitly in this case because it is the default. See "Running Versus Final Semantics and Keywords" for more information.
- The `price_dif` measure shows us each day's difference in stock price from the price at the first day of a match. In the expression "`price - STRT.price)`," you see a case where an unqualified column, "`price`," is used with a qualified column, "`STRT.price`".

**Example 22-4 Pattern Match for a W-Shape**

This example illustrates a W-Shape.

```
SELECT *
FROM Ticker MATCH_RECOGNIZE (
  PARTITION BY symbol
  ORDER BY tstamp
  MEASURES
    MATCH_NUMBER() AS match_num,
    CLASSIFIER()  AS  var_match,
    STRT.tstamp AS start_tstamp,
    FINAL LAST(UP.tstamp) AS end_tstamp
  ALL ROWS PER MATCH
  AFTER MATCH SKIP TO LAST UP
  PATTERN (STRT DOWN+ UP+ DOWN+ UP+)
  DEFINE
    DOWN AS DOWN.price < PREV(DOWN.price),
    UP AS UP.price > PREV(UP.price)
  ) MR
ORDER BY MR.symbol, MR.match_num, MR.tstamp;

SYMBOL     TSTAMP     MATCH_NUM  VAR_  START_TST  END_TSTAM      PRICE
---------- --------- ----------  ----  ---------  --------- ----------
ACME       05-APR-11          1  STRT  05-APR-11  13-APR-11         25
ACME       06-APR-11          1  DOWN  05-APR-11  13-APR-11         12
ACME       07-APR-11          1  UP    05-APR-11  13-APR-11         15
ACME       08-APR-11          1  UP    05-APR-11  13-APR-11         20
ACME       09-APR-11          1  UP    05-APR-11  13-APR-11         24
ACME       10-APR-11          1  UP    05-APR-11  13-APR-11         25
ACME       11-APR-11          1  DOWN  05-APR-11  13-APR-11         19
ACME       12-APR-11          1  DOWN  05-APR-11  13-APR-11         15
ACME       13-APR-11          1  UP    05-APR-11  13-APR-11         25
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

What does this query do? It builds on the concepts introduced in Example 22-1 and seeks W-shapes in the data rather than V-shapes. The query results show one W-shape. To find the W-shape, the line defining the PATTERN regular expression was modified to seek the pattern DOWN followed by UP two consecutive times: PATTERN (STRT DOWN+ UP+ DOWN+ UP+). This pattern specification means it can only match a W-shape where the two V-shapes have no separation between them. For instance, if there is a flat interval with the price unchanging, and that interval occurs between two V-shapes, the pattern will not match that data. To illustrate the data returned, the output is set to ALL ROWS PER MATCH. Note that FINAL LAST(UP.tstamp) in the MEASURES clause returns the timestamp value for the last row mapped to UP.

## 22.2.2 Tasks and Keywords in Pattern Matching

This section discusses the following tasks and keywords in pattern matching.

**PARTITION BY: Logically Dividing the Rows into Groups**

You will typically want to divide your input data into logical groups for analysis. In the example with stocks, you divide the pattern matching so that it applies to just one stock at a time. You do this with the PARTITION BY keyword. PARTITION BY is used to specify that the rows of the row pattern input table are to be partitioned by one or more columns. Matches are found within partitions and do not cross partition boundaries.

If there is no PARTITION BY, then all rows of the row pattern input table constitute a single row pattern partition.

**ORDER BY: Logically Ordering the Rows in a Partition**

After you divided your input data into logical partitions, you will want to order the data inside each partition. Without row ordering, you cannot have a reliable sequence to check for pattern matches. The ORDER BY keyword is used to specify the order of rows within a row pattern partition.

**[ONE ROW | ALL ROWS] PER MATCH: Choosing Summaries or Details for Each Match**

You will sometimes want summary data about the matches and other times need details. You can do that with the following SQL keywords:

- ONE ROW PER MATCH

  Each match produces one summary row. This is the default.

- ALL ROWS PER MATCH

  A match spanning multiple rows will produce one output row for each row in the match.

The output is explained in "Row Pattern Output".

**MEASURES: Defining Calculations for Export from the Pattern Matching**

The pattern matching clause enables you to create expressions useful in a wide range of analyses. These are presented as columns in the output by using the MEASURES clause. The MEASURES clause defines row pattern measure columns, whose value is computed by evaluating an expression related to a particular match.

**PATTERN: Defining the Row Pattern That Will be Matched**

The PATTERN clause lets you define which pattern variables must be matched, the sequence in which they must be matched, and the quantity of rows which must be matched. The PATTERN clause specifies a regular expression for the match search.


Copyright © 2001, 2025, Oracle and/or its affiliates.

A row pattern match consists of a set of contiguous rows in a row pattern partition. Each row of the match is mapped to a pattern variable. Mapping of rows to pattern variables must conform to the regular expression in the `PATTERN` clause, and all conditions in the `DEFINE` clause must be true.

**DEFINE: Defining Primary Pattern Variables**

Because the `PATTERN` clause depends on pattern variables, you must have a clause to define these variables. They are specified in the `DEFINE` clause.

`DEFINE` is a required clause, used to specify the conditions that a row must meet to be mapped to a specific pattern variable.

A pattern variable does not require a definition. Any row can be mapped to an undefined pattern variable.

**AFTER MATCH SKIP: Restarting the Matching Process After a Match is Found**

After the query finds a match, it must look for the next match at exactly the correct point. Do you want to find matches where the end of the earlier match overlaps the start of the next match? Do you want some other variation? Pattern matching provides great flexibility in specifying the restart point. The `AFTER MATCH SKIP` clause determines the point to resume row pattern matching after a non-empty match was found. The default for the clause is `AFTER MATCH SKIP PAST LAST ROW`: resume pattern matching at the next row after the last row of the current match.

**MATCH_NUMBER: Finding Which Rows are Members of Which Match**

You might have a large number of matches for your pattern inside a given row partition. How do you tell apart all these matches? This is done with the `MATCH_NUMBER` function. Matches within a row pattern partition are numbered sequentially starting with 1 in the order they are found. Note that match numbering starts over again at 1 in each row pattern partition, because there is no inherent ordering between row pattern partitions.

**CLASSIFIER: Finding Which Pattern Variable Applies to Which Rows**

Along with knowing which `MATCH_NUMBER` you are seeing, you may want to know which component of a pattern applies to a specific row. This is done using the `CLASSIFIER` function. The classifier of a row is the pattern variable that the row is mapped to by a row pattern match. The `CLASSIFIER` function returns a character string whose value is the name of the variable the row is mapped to.

## 22.2.3 Pattern Matching Syntax

The pattern matching syntax is as follows:

```
table_reference ::=
  {only (query_table_expression) | query_table_expression }[flashback_query_clause]
   [pivot_clause|unpivot_clause|row_pattern_recognition_clause] [t_alias]

row_pattern_recognition_clause ::=
  MATCH_RECOGNIZE (
   [row_pattern_partition_by ]
   [row_pattern_order_by ]
   [row_pattern_measures ]
   [row_pattern_rows_per_match ]
   [row_pattern_skip_to ]
   PATTERN (row_pattern)
   [ row_pattern_subset_clause]
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
   DEFINE row_pattern_definition_list
   )

row_pattern_partition_by ::=
   PARTITION BY column[, column]...

row_pattern_order_by ::=
   ORDER BY column[, column]...

row_pattern_measures ::=
   MEASURES row_pattern_measure_column[, row_pattern_measure_column]...

row_pattern_measure_column ::=
   expression AS c_alias

row_pattern_rows_per_match ::=
   ONE ROW PER MATCH
  | ALL ROWS PER MATCH

row_pattern_skip_to ::=
   AFTER MATCH {
    SKIP TO NEXT ROW
   | SKIP PAST LAST ROW
   | SKIP TO FIRST variable_name
   | SKIP TO LAST variable_name
   | SKIP TO variable_name}

row_pattern ::=
   row_pattern_term
  | row_pattern "|" row_pattern_term

row_pattern_term ::=
   row_pattern_factor
  | row_pattern_term row_pattern_factor

row_pattern_factor ::=
   row_pattern_primary [row_pattern_quantifier]

row_pattern_quantifier ::=
    *[?]
   |+[?]
   |?[?]
   |"{"[unsigned_integer ],[unsigned_integer]"}"[?]
   |"{"unsigned_integer "}"

row_pattern_primary ::=
   variable_name
   |$
   |^
   |([row_pattern])
   |"{-" row_pattern"-}"
   | row_pattern_permute

row_pattern_permute ::=
   PERMUTE (row_pattern [, row_pattern] ...)

row_pattern_subset_clause ::=
   SUBSET row_pattern_subset_item [, row_pattern_subset_item] ...

row_pattern_subset_item ::=
   variable_name = (variable_name[ , variable_name]...)
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
row_pattern_definition_list ::=
   row_pattern_definition[, row_pattern_definition]...

row_pattern_definition ::=
   variable_name AS condition
```

The syntax for row pattern operations inside pattern matching is:

```
function ::=
 single_row_function
| aggregate_function
| analytic_function
| object_reference_function
| model_function
| user_defined_function
| data_cartridge_function
| row_pattern_recognition_function

row_pattern_recognition_function ::=
 row_pattern_classifier_function
| row_pattern_match_number_function
| row_pattern_navigation_function
| row_pattern_aggregate_function

row_pattern_classifier_function ::=
  CLASSIFIER( )

row_pattern_match_number_function ::=
  MATCH_NUMBER( )

row_pattern_navigation_function ::=
   row_pattern_navigation_logical
  | row_pattern_navigation_physical
  | row_pattern_navigation_compound

row_pattern_navigation_logical ::=
   [RUNNING|FINAL] {FIRST|LAST} (expression[,offset])

row_pattern_navigation_physical ::=
   {PREV|NEXT}(expression[, offset])

row_pattern_navigation_compound ::=
   {PREV | NEXT} (
     [RUNNING| FINAL] {FIRST|LAST} (expression[, offset]) [,offset])
```

The syntax for set function specification inside the pattern matching clause is:

```
row_pattern_aggregate_function ::=
   [RUNNING | FINAL] aggregate_function
```

# 22.3 Pattern Matching Details

This section presents details on the items discussed in Pattern Matching Syntax, plus additional topics. Note that some of the material is unavoidably intricate. Certain aspects of pattern matching require careful attention to subtle details.

- PARTITION BY: Logically Dividing the Rows into Groups
- ORDER BY: Logically Ordering the Rows in a Partition
- [ONE ROW | ALL ROWS] PER MATCH: Choosing Summaries or Details for Each Match


Copyright © 2001, 2025, Oracle and/or its affiliates.

- MEASURES: Defining Calculations for Use in the Query
- PATTERN: Defining the Row Pattern to Be Matched
- SUBSET: Defining Union Row Pattern Variables
- DEFINE: Defining Primary Pattern Variables
- AFTER MATCH SKIP: Defining Where to Restart the Matching Process After a Match Is Found
- Expressions in MEASURES and DEFINE
- Row Pattern Output

## 22.3.1 PARTITION BY: Logically Dividing the Rows into Groups

Typically, you want to divide your input data into logical groups for analysis. In the examples with stocks, the pattern matching is divided so that it applies to just one stock at a time. To do this, use the `PARTITION BY` clause. `PARTITION BY` specifies that the rows of the input table are to be partitioned by one or more columns. Matches are found within partitions and do not cross partition boundaries.

If there is no `PARTITION BY`, then all rows of the row pattern input table constitute a single row pattern partition.

## 22.3.2 ORDER BY: Logically Ordering the Rows in a Partition

The `ORDER BY` clause is used to specify the order of rows within a row pattern partition. If the order of two rows in a row pattern partition is not determined by `ORDER BY`, then the result of the `MATCH_RECOGNIZE` clause is non-deterministic: it may not give consistent results each time the query is run.

## 22.3.3 [ONE ROW | ALL ROWS] PER MATCH: Choosing Summaries or Details for Each Match

You will sometimes want summary data about the matches and other times need details. You can do that with the following SQL:

- `ONE ROW PER MATCH`

  Each match produces one summary row. This is the default.

- `ALL ROWS PER MATCH`

  A match spanning multiple rows will produce one output row for each row in the match.

The output is explained in "Row Pattern Output".

The `MATCH_RECOGNIZE` clause may find a match with zero rows. For an empty match, `ONE ROW PER MATCH` returns a summary row: the `PARTITION BY` columns take the values from the row where the empty match occurs, and the measure columns are evaluated over an empty set of rows.

`ALL ROWS PER MATCH` has three suboptions:

- `ALL ROWS PER MATCH SHOW EMPTY MATCHES`
- `ALL ROWS PER MATCH OMIT EMPTY MATCHES`
- `ALL ROWS PER MATCH WITH UNMATCHED ROWS`


Copyright © 2001, 2025, Oracle and/or its affiliates.

These options are explained in "Advanced Topics in Pattern Matching".

## 22.3.4 MEASURES: Defining Calculations for Use in the Query

The MEASURES clause defines a list of columns for the pattern output table. Each pattern measure column is defined with a column name whose value is specified by a corresponding pattern measure expression.

A value expression is defined with respect to the pattern variables. Value expression can contain set functions, pattern navigation operations, CLASSIFIER(), MATCH_NUMBER(), and column references to any column of the input table. See "Expressions in MEASURES and DEFINE" for more information.

## 22.3.5 PATTERN: Defining the Row Pattern to Be Matched

The PATTERN keyword specifies the pattern to be recognized in the ordered sequence of rows in a partition. Each variable name in a pattern corresponds to a Boolean condition, which is specified later using the DEFINE component of the syntax.

The PATTERN clause is used to specify a regular expression. It is outside the scope of this material to explain regular expression concepts and details. If you are not familiar with regular expressions, you are encouraged to familiarize yourself with the topic using other sources.

The regular expression in a PATTERN clause is enclosed in parentheses. PATTERN may use the following operators:

- Concatenation

  Concatenation is used to list two or more items in a pattern to be matched in that order. Items are concatenated when there is no operator sign between two successive items. For example: PATTERN (A B C).

- Quantifiers

  Quantifiers are POSIX operators that define the number of iterations accepted for a match. The syntax of POSIX extended regular expressions is similar to that of traditional UNIX regular expressions. The following are choices for quantifiers:

  – * — 0 or more iterations
  – + — 1 or more iterations
  – ? — 0 or 1 iterations
  – {n} — n iterations (n > 0)
  – {n,} — n or more iterations (n >= 0)
  – {n,m} — between n and m (inclusive) iterations (0 <= n <= m, 0 < m)
  – {,m} — between 0 and m (inclusive) iterations (m > 0)
  – reluctant quantifiers — indicated by an additional question mark following a quantifier (*?, +?, ??, {n,}?, { n, m }?, {,m}?). See "Reluctant Versus Greedy Quantifier" for the difference between reluctant and non-reluctant quantifiers.

  The following are examples of using quantifier operators:

  – A* matches 0 or more iterations of A
  – A{3,6} matches 3 to 6 iterations of A
  – A{,4} matches 0 to 4 iterations of A


Copyright © 2001, 2025, Oracle and/or its affiliates.

- Alternation

  Alternation matches a single regular expression from a list of several possible regular expressions. The alternation list is created by placing a vertical bar (|) between each regular expression. Alternatives are preferred in the order they are specified. As an example, `PATTERN (A | B | C)` attempts to match `A` first. If `A` is not matched, it attempts to match `B`. If `B` is not matched, it attempts to match `C`.

- Grouping

  Grouping treats a portion of the regular expression as a single unit, enabling you to apply regular expression operators such as quantifiers to that group. Grouping is created with parentheses. As an example, `PATTERN ((A B){3} C)` attempts to match the group `(A B)` three times and then seeks one occurrence of `C`.

- `PERMUTE`

  See "How to Express All Permutations" for more information.

- Exclusion

  Parts of the pattern to be excluded from the output of `ALL ROWS PER MATCH` are enclosed between {- and -}. See "How to Exclude Portions of the Pattern from the Output".

- Anchors

  Anchors work in terms of positions rather than rows. They match a position either at the start or end of a partition.

  – `^` matches the position before the first row in the partition.

  – `$` matches the position after the last row in the partition.

  As an example, `PATTERN (^A+$)` will match only if all rows in a partition satisfy the condition for `A`. The resulting match spans the entire partition.

- Empty pattern (), matches an empty set of rows

This section contains the following topics:

- Reluctant Versus Greedy Quantifier
- Operator Precedence

## 22.3.5.1 Reluctant Versus Greedy Quantifier

Pattern quantifiers are referred to as greedy; they will attempt to match as many instances of the regular expression on which they are applied as possible. The exception is pattern quantifiers that have a question mark `?` as a suffix, and those are referred to as reluctant. They will attempt to match as few instances as possible of the regular expression on which they are applied.

The difference between greedy and reluctant quantifiers appended to a single pattern variable is illustrated as follows: `A*` tries to map as many rows as possible to `A`, whereas `A*?` tries to map as few rows as possible to `A`. For example:

```
PATTERN (X Y* Z)
```

The pattern consists of three variable names, `X`, `Y`, and `Z`, with `Y` quantified with `*`. This means a pattern match will be recognized and reported when the following condition is met by consecutive incoming input rows:


Copyright © 2001, 2025, Oracle and/or its affiliates.

- A row satisfies the condition that defines variable X followed by zero or more rows that satisfy the condition that defines the variable Y followed by a row that satisfies the condition that defines the variable Z.

During the pattern matching process, after a row was mapped to X and 0 or more rows were mapped to Y, if the following row can be mapped to both variables Y and Z (which satisfies the defining condition of both Y and Z), then, because the quantifier * for Y is greedy, the row is preferentially mapped to Y instead of Z. Due to this greedy property, Y gets preference over Z and a greater number of rows to Y are mapped. If the pattern expression was PATTERN (X Y*? Z), which uses a reluctant quantifier *? over Y, then Z gets preference over Y.

### 22.3.5.2 Operator Precedence

The precedence of the elements in a regular expression, in decreasing order, is as follows:

- row_pattern_primary

  These elements include primary pattern variables (pattern variables not created with the SUBSET clause described in "SUBSET: Defining Union Row Pattern Variables"), anchors, PERMUTE, parenthetic expressions, exclusion syntax, and empty pattern

- Quantifier

  A row_pattern_primary may have zero or one quantifier.

- Concatenation
- Alternation

Precedence of alternation is illustrated by PATTERN(A B | C D), which is equivalent to PATTERN ((A B) | (C D)). It is not, however, equivalent to PATTERN (A (B | C) D).

Precedence of quantifiers is illustrated by PATTERN (A B *), which is equivalent to PATTERN (A (B*)). It is not, however, PATTERN ((A B)*).

A quantifier may not immediately follow another quantifier. For example, PATTERN(A**) is prohibited.

It is permitted for a primary pattern variable to occur more than once in a pattern, for example, PATTERN (X Y X).

## 22.3.6 SUBSET: Defining Union Row Pattern Variables

At times, it is helpful to create a grouping of multiple pattern variables that can be referred to with a variable name of its own. These groupings are called union row pattern variables, and you create them with the SUBSET clause. The union row pattern variable created by SUBSET can be used in the MEASURES and DEFINE clauses. The SUBSET clause is optional. It is used to declare union row pattern variables. For example, here is a query using SUBSET to calculate an average based on all rows that are mapped to the union of STRT and DOWN variables, where STRT is the starting point for a pattern, and DOWN is the downward (left) leg of a V shape.

Example 22-5 illustrates creating a union row pattern variable.

**Example 22-5 Defining Union Row Pattern Variables**

```
SELECT *
FROM Ticker MATCH_RECOGNIZE(
     PARTITION BY symbol
     ORDER BY tstamp
     MEASURES FIRST(STRT.tstamp) AS strt_time,
              LAST(DOWN.tstamp) AS bottom,
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
            AVG(STDN.Price) AS stdn_avgprice
     ONE ROW PER MATCH
     AFTER MATCH SKIP TO LAST UP
     PATTERN (STRT DOWN+ UP+)
     SUBSET STDN= (STRT, DOWN)
     DEFINE
        UP AS UP.Price > PREV(UP.Price),
        DOWN AS DOWN.Price < PREV (DOWN.Price)
);

SYMBOL    STRT_TIME    BOTTOM                                          STDN_AVGPRICE
_________ ____________ _____________ ____________________________________________
ACME      05-APR-22    06-APR-22                                                18.5
ACME      10-APR-22    12-APR-22       19.666666666666666666666666666666666666667
ACME      14-APR-22    16-APR-22                                                  17
```

This example declares a single union row pattern variable, `STDN`, and defines it as the union of the rows mapped to `STRT` and the rows mapped to `DOWN`. There can be multiple union row pattern variables in a query. For example:

```
PATTERN (W+ X+ Y+ Z+)
SUBSET XY = (X, Y),
       WZ = (W, Z)
```

The right-hand side of a `SUBSET` item is a comma-separated list of distinct primary row pattern variables within parentheses. This defines the union row pattern variable (on the left-hand side) as the union of the primary row pattern variables (on the right-hand side).

Note that the list of pattern variables on the right-hand side may not include any union row pattern variables (there are no unions of unions).

For every match, there is one implicit union row pattern variable called the universal row pattern variable. The universal row pattern variable is the union of all primary row pattern variables. For instance, if your pattern has primary pattern variable `A`, `B`, and `C`, then the universal row pattern variable is equivalent to a `SUBSET` clause with the argument `(A, B, C)`. Thus, every row of a match is mapped to the universal row pattern variable. Any unqualified column reference within the `MEASURES` or `DEFINE` clauses is implicitly qualified by the universal row pattern variable. Note that there is no keyword to explicitly specify the universal row pattern variable.

## 22.3.7 DEFINE: Defining Primary Pattern Variables

`DEFINE` is a mandatory clause, used to specify the conditions that define primary pattern variables. In the example:

```
DEFINE UP AS UP.Price > PREV(UP.Price),
DOWN AS DOWN.Price < PREV(DOWN.Price)
```

`UP` is defined by the condition `UP.Price > PREV (UP.Price)`, and `DOWN` is defined by the condition `DOWN.Price < PREV (DOWN.Price)`. (`PREV` is a row pattern navigation operation which evaluates an expression in the previous row; see "Row Pattern Navigation Operations" regarding the complete set of row pattern navigation operations.)

A pattern variable does not require a definition; if there is no definition, any row can be mapped to the pattern variable.

A union row pattern variable (see discussion of `SUBSET` in "SUBSET: Defining Union Row Pattern Variables") cannot be defined by `DEFINE`, but can be referenced in the definition of a pattern variable.


Copyright © 2001, 2025, Oracle and/or its affiliates.

The definition of a pattern variable can reference another pattern variable, which is illustrated in Example 22-6.

**Example 22-6 Defining Pattern Variables**

```
SELECT *
FROM Ticker MATCH_RECOGNIZE (
     PARTITION BY Symbol
     FROM Ticker
     MATCH_RECOGNIZE (
     PARTITION BY Symbol
     ORDER BY tstamp
     MEASURES FIRST (A.tstamp) AS A_Firstday,
              LAST (D.tstamp) AS D_Lastday,
              AVG (B.Price) AS B_Avgprice,
              AVG (D.Price) AS D_Avgprice
     PATTERN (A B+ C+ D)
     SUBSET BC = (B,C)
     DEFINE A AS Price > 100,
            B AS B.Price > A.Price,
            C AS C.Price < AVG (B.Price),
            D AS D.Price > MAX (BC.Price)
) M
```

In this example:

- The definition of `A` implicitly references the universal row pattern variable (because of the unqualified column reference `Price`).
- The definition of `B` references the pattern variable `A`.
- The definition of `C` references the pattern variable `B`.
- The definition of `D` references the union row pattern variable `BC`.

The conditions are evaluated on successive rows of a partition in a trial match, with the current row being tentatively mapped to a pattern variable as permitted by the pattern. To be successfully mapped, the condition must evaluate to true.

In the previous example:

```
A AS Price > 100
```

`Price` refers to the `Price` in the current row, because the last row mapped to any primary row pattern variable is the current row, which is tentatively mapped to `A`. Alternatively, in this example, using `A.Price` would have led to the same results.

```
B AS B.Price > A.Price
```

`B.Price` refers to the `Price` in the current row (because `B` is being defined), whereas `A.Price` refers to the last row mapped to `A`. In view of the pattern, the only row mapped to `A` is the first row to be mapped.

```
C AS C.Price < AVG(B.Price)
```

Here `C.Price` refers to the `Price` in the current row, because `C` is being defined. The aggregate `AVG` (that is, insert `Price`) is computed as the average of all rows that are already mapped to `B`.

```
D AS D.Price > MAX(BC.Price)
```

The pattern variable `D` is similar to pattern variable `C`, though it illustrates the use of a union row pattern variable in the Boolean condition. In this case, `MAX(BC.Price)` returns the maximum


Copyright © 2001, 2025, Oracle and/or its affiliates.

price value of the rows matched to variable B or variable C. The semantics of Boolean conditions are discussed in more detail in "Expressions in MEASURES and DEFINE".

## 22.3.8 AFTER MATCH SKIP: Defining Where to Restart the Matching Process After a Match Is Found

The AFTER MATCH SKIP clause determines the point to resume row pattern matching after a non-empty match was found. The default for the clause is AFTER MATCH SKIP PAST LAST ROW. The options are as follows:

- AFTER MATCH SKIP TO NEXT ROW

  Resume pattern matching at the row after the first row of the current match.

- AFTER MATCH SKIP PAST LAST ROW

  Resume pattern matching at the next row after the last row of the current match.

- AFTER MATCH SKIP TO FIRST *pattern_variable*

  Resume pattern matching at the first row that is mapped to the pattern variable.

- AFTER MATCH SKIP TO LAST *pattern_variable*

  Resume pattern matching at the last row that is mapped to the pattern variable.

- AFTER MATCH SKIP TO *pattern_variable*

  The same as AFTER MATCH SKIP TO LAST *pattern_variable*.

When using AFTER MATCH SKIP TO FIRST or AFTER MATCH SKIP TO [LAST], it is possible that no row is mapped to the pattern_variable. For example:

```
AFTER MATCH SKIP TO A
PATTERN (X A* X),
```

The pattern variable A in the example might have no rows mapped to A. If there is no row mapped to A, then there is no row to skip to, so a runtime exception is generated. Another problem condition is that AFTER MATCH SKIP may try to resume pattern matching at the same row that the last match started. For example:

```
AFTER MATCH SKIP TO X
PATTERN (X Y+ Z),
```

In this example, AFTER MATCH SKIP TO X tries to resume pattern matching at the same row where the previous match was found. This would result in an infinite loop, so a runtime exception is generated for this scenario.

Note that the AFTER MATCH SKIP syntax only determines the point to resume scanning for a match after a non-empty match. When an empty match is found, one row is skipped (as if SKIP TO NEXT ROW had been specified). Thus an empty match never causes one of these exceptions. A query that gets one of these exceptions should be rewritten, as, for example, in the following:

```
AFTER MATCH SKIP TO A
PATTERN (X (A | B) Y)
```

This will cause a run-time error when a row is mapped to B, because no row was mapped to A. If the intent is to skip to either A or B, the following will work:

```
AFTER MATCH SKIP TO C
PATTERN (X (A | B) Y)
SUBSET C = (A, B)
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

In the revised example, no runtime error is possible, whether A or B is matched.

As another example:

```
AFTER MATCH SKIP TO FIRST A
PATTERN (A* X)
```

This example gets an exception after the first match, either for skipping to the first row of the match (if A* matches) or for skipping to a nonexistent row (if A* does not match). In this example, SKIP TO NEXT ROW is a better choice.

When using ALL ROWS PER MATCH together with skip options other than AFTER MATCH SKIP PAST LAST ROW, it is possible for consecutive matches to overlap, in which case a row R of the row pattern input table might occur in more than one match. In that case, the row pattern output table will have one row for each match in which the row participates. If a row of the row pattern input table participates in multiple matches, the MATCH_NUMBER function can be used to distinguish among the matches. When a row participates in more than one match, its classifier can be different in each match.

## 22.3.9 Expressions in MEASURES and DEFINE

Pattern matching provides the following scalar expressions that are unique to row pattern matching:

- Row pattern navigation operations, using the functions PREV, NEXT, FIRST and LAST. Row pattern navigation operations are discussed in "Row Pattern Navigation Operations".
- The MATCH_NUMBER function, which returns the sequential number of a row pattern match within its row pattern partition, discussed in "MATCH_NUMBER: Finding Which Rows Are in Which Match".
- The CLASSIFIER function, which returns the name of the primary row pattern variable that a row is mapped to, discussed in "CLASSIFIER: Finding Which Pattern Variable Applies to Which Rows".

Expressions in MEASURES and DEFINE clauses have the same syntax and semantics, with the following exceptions:

- The DEFINE clause only supports running semantics.
- The MEASURES clause defaults to running semantics, but also supports final semantics. This distinction is discussed in "RUNNING Versus FINAL Semantics".

**Working with Expressions**

This section discusses some of the considerations when working with expressions in pattern matching, and includes:

- MATCH_NUMBER: Finding Which Rows Are in Which Match
- CLASSIFIER: Finding Which Pattern Variable Applies to Which Rows
- Row Pattern Column References
- Aggregates
- Row Pattern Navigation Operations

### 22.3.9.1 MATCH_NUMBER: Finding Which Rows Are in Which Match

Matches within a row pattern partition are numbered sequentially starting with 1 in the order they are found. Note that match numbering starts over again at 1 in each row pattern partition,


Copyright © 2001, 2025, Oracle and/or its affiliates.

because there is no inherent ordering between row pattern partitions. `MATCH_NUMBER()` is a function that returns a numeric value with scale 0 (zero) whose value is the sequential number of the match within the row pattern partition.

The previous examples using `MATCH_NUMBER()` have shown it used in the `MEASURES` clause. It is also possible to use `MATCH_NUMBER()` in the `DEFINE` clause, where it can be used to define conditions that depend upon the match number.

### 22.3.9.2 CLASSIFIER: Finding Which Pattern Variable Applies to Which Rows

The `CLASSIFIER` function returns a character string whose value is the name of the pattern variable to which a row is mapped. The `CLASSIFIER` function is allowed in both the `MEASURES` and the `DEFINE` clauses.

In the `DEFINE` clause, the `CLASSIFIER` function returns the name of the primary pattern variable to which the current row is mapped.

In the `MEASURES` clause:

- If `ONE ROW PER MATCH` is specified, the query is using the last row of the match when processing the `MEASURES` clause, so the `CLASSIFIER` function returns the name of the pattern variable to which the last row of the match is mapped.
- If `ALL ROWS PER MATCH` is specified, for each row of the match found, the `CLASSIFIER` function returns the name of the pattern variable to which the row is mapped.

The classifier for the starting row of an empty match is the null value.

### 22.3.9.3 Row Pattern Column References

A row pattern column reference is a column name qualified by an explicit or implicit pattern variable, such as the following:

```
A.Price
```

`A` is the pattern variable and `Price` is a column name. A column name with no qualifier, such as `Price`, is implicitly qualified by the universal row pattern variable, which references the set of all rows in a match. Column references can be nested within other syntactic elements, notably aggregates and navigation operators. (However, nesting in row pattern matching is subject to limitations described in "Prohibited Nesting in the MATCH_RECOGNIZE Clause" for the `FROM` clause.)

Pattern column references are classified as follows:

- Nested within an aggregate, such as `SUM`: an aggregated row pattern column reference.
- Nested within a row pattern navigation operation (`PREV`, `NEXT`, `FIRST`, and `LAST`): a navigated row pattern column reference.
- Otherwise: an ordinary row pattern column reference.

All pattern column references in an aggregate or row pattern navigation operation must be qualified by the same pattern variable. For example:

```
PATTERN (A+ B+)
DEFINE B AS AVG(A.Price + B.Tax) > 100
```

The preceding example is a syntax error, because `A` and `B` are two different pattern variables. Aggregate semantics require a single set of rows; there is no way to form a single set of rows on which to evaluate `A.Price + B.Tax`. However, the following is acceptable:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
DEFINE B AS AVG (B.Price + B.Tax) > 100
```

In the preceding example, all pattern column references in the aggregate are qualified by `B`.

An unqualified column reference is implicitly qualified by the universal row pattern variable, which references the set of all rows in a match. For example:

```
DEFINE B AS AVG(Price + B.Tax) > 1000
```

The preceding example is a syntax error, because the unqualified column reference `Price` is implicitly qualified by the universal row pattern variable, whereas `B.Tax` is explicitly qualified by `B`. However, the following is acceptable:

```
DEFINE B AS AVG (Price + Tax) > 1000
```

In the preceding example, both `Price` and `Tax` are implicitly qualified by the universal row pattern variable.

## 22.3.9.4 Aggregates

The aggregates (`COUNT`, `SUM`, `AVG`, `MAX`, and `MIN`) can be used in both the `MEASURES` and `DEFINE` clauses. Note that the `DISTINCT` keyword is not supported. When used in row pattern matching, aggregates operate on a set of rows that are mapped to a particular pattern variable, using either running or final semantics. For example:

```
MEASURES SUM (A.Price) AS RunningSumOverA,
         FINAL SUM(A.Price) AS FinalSumOverA
ALL ROWS PER MATCH
```

In this example, `A` is a pattern variable. The first pattern measure, `RunningSumOverA`, does not specify either `RUNNING` or `FINAL`, so it defaults to `RUNNING`. This means that it is computed as the sum of `Price` in those rows that are mapped to `A` by the current match, up to and including the current row. The second pattern measure, `FinalSumOverA`, computes the sum of `Price` over all rows that are mapped to `A` by the current match, including rows that may be later than the current row. Final aggregates are only available in the `MEASURES` clause, not in the `DEFINE` clause.

An unqualified column reference contained in an aggregate is implicitly qualified by the universal row pattern variable, which references all rows of the current pattern match. For example:

```
SUM (Price)
```

The running sum of `Price` over all rows of the current row pattern match is computed.

All column references contained in an aggregate must be qualified by the same pattern variable. For example:

```
SUM (Price + A.Tax)
```

Because `Price` is implicitly qualified by the universal row pattern variable, whereas `A.Tax` is explicitly qualified by `A`, you get a syntax error.

The `COUNT` aggregate has special syntax for pattern matching, so that `COUNT(A.*)` can be specified. `COUNT(A.*)` is the number of rows that are mapped to the pattern variable A by the current pattern match. As for `COUNT(*)`, the `*` implicitly covers the rows of the universal row pattern variable, so that `COUNT(*)` is the number of rows in the current pattern match.


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 22.3.9.5 Row Pattern Navigation Operations

There are four functions — PREV, NEXT, FIRST, and LAST — that enable navigation within the row pattern by either physical or logical offsets.

### 22.3.9.5.1 PREV and NEXT

The PREV function can be used to evaluate an expression using a previous row in a partition. It operates in terms of physical rows and is not limited to the rows mapped to a specific variable. If there is no previous row, the null value is returned. For example:

```
DEFINE A AS PREV (A.Price) > 100
```

The preceding example says that the current row can be mapped to A if the row preceding the current row has a price greater than 100. If the preceding row does not exist (that is, the current row is the first row of a row pattern partition), then PREV(A.Price) is null, so the condition is not True, and therefore the first row cannot be mapped to A.

Note that you can use another pattern variable (such as B) in defining the pattern variable A, and have the condition apply a PREV() function to that other pattern variable. That might resemble:

```
DEFINE A AS PREV (B.PRICE) > 100
```

In that case, the starting row used by the PREV() function for its navigation is the last row mapped to pattern variable B.

The PREV function can accept an optional non-negative integer argument indicating the physical offset to the previous rows. Thus:

- PREV (A.Price, 0) is equivalent to A.Price.
- PREV (A.price, 1) is equivalent to PREV (A.Price). Note: 1 is the default offset.
- PREV (A.Price, 2) is the value of Price in the row two rows before to the row denoted by A with running semantics. (If no row is mapped to A, or if there is no row two rows prior, then PREV (A.Price, 2) is null.)

The offset must be a runtime constant (literal, bind variable, and expressions involving them), but not a column or a subquery.

The NEXT function is a forward-looking version of the PREV function. It can be used to reference rows in the forward direction in the row pattern partition using a physical offset. The syntax is the same as for PREV, except for the name of the function. For example:

```
DEFINE A AS NEXT (A.Price) > 100
```

The preceding example looks forward one row in the row pattern partition. Note that pattern matching does not support aggregates that look past the current row during the DEFINE clause, because of the difficulty of predicting what row will be mapped to what pattern variable in the future. The NEXT function does not violate this principle, because it navigates to "future" rows on the basis of a physical offset, which does not require knowing the future mapping of rows.

For example, to find an isolated row that is more than twice the average of the two rows before and two rows after it: using NEXT, this can be expressed:

```
PATTERN ( X )
DEFINE X AS X.Price > 2 * ( PREV (X.Price, 2)
       + PREV (X.Price, 1)
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
        + NEXT (X.Price, 1)
        + NEXT (X.Price, 2) ) / 4
```

Note that the row in which PREV or NEXT is evaluated is not necessarily mapped to the pattern variable in the argument. For example, in this example, PREV (X.Price, 2) is evaluated in a row that is not part of the match. The purpose of the pattern variable is to identify the row from which to offset, not the row that is ultimately reached. (If the definition of pattern variable refers to itself in a PREV() or NEXT(), then it is referring to the current row as the row from which to offset.) This point is discussed further in "Nesting FIRST and LAST Within PREV and NEXT in Pattern Matching".

PREV and NEXT may be used with more than one column reference; for example:

```
DEFINE A AS PREV (A.Price + A.Tax) < 100
```

When using a complex expression as the first argument of PREV or NEXT, all qualifiers must be the same pattern variable (in this example, A).

PREV and NEXT always have running semantics; the keywords RUNNING and FINAL cannot be used with PREV or NEXT. (See the section on "Running Versus Final Semantics and Keywords"). To obtain final semantics, use, for example, PREV (FINAL LAST (A.Price)) as explained in "Nesting FIRST and LAST Within PREV and NEXT in Pattern Matching".

### 22.3.9.5.1.1 FIRST and LAST

In contrast to the PREV and NEXT functions, the FIRST and LAST functions navigate only among the rows mapped to pattern variables: they use logical, not physical, offsets. FIRST returns the value of an expression evaluated in the first row of the group of rows mapped to a pattern variable. For example:

```
FIRST (A.Price)
```

If no row is mapped to A, then the value is null.

Similarly, LAST returns the value of an expression evaluated in the last row of the group of rows mapped to a pattern variable. For example:

```
LAST (A.Price)
```

The preceding example evaluates A.Price in the last row that is mapped to A (null if there is no such row).

The FIRST and LAST operators can accept an optional non-negative integer argument indicating a logical offset within the set of rows mapped to the pattern variable. For example:

```
FIRST (A.Price, 1)
```

The preceding line evaluates Price in the second row that is mapped to A. Consider the following data set and mappings in Table 22-1.

**Table 22-1 Pattern and Row**

| Row | Price | Mapping |
|---|---|---|
| R1 | 10 | A |
| R2 | 20 | B |
| R3 | 30 | A |
| R4 | 40 | C |


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Table 22-1 (Cont.) Pattern and Row**

| Row | Price | Mapping |
|---|---|---|
| R5 | 50 | A |

Then the following:

- `FIRST (A.Price)` = `FIRST (A.Price, 0)` = `LAST (A.Price, 2) = 10`
- `FIRST (A.Price, 1)` = `LAST (A.Price, 1) = 30`
- `FIRST (A.Price, 2)` = `LAST (A.Price, 0) = LAST (A.Price) = 50`
- `FIRST (A.Price, 3)` is null, as is `LAST (A.Price, 3)`

Note that the offset is a logical offset, moving within the set of rows `{R1, R3, R5}` that are mapped to the pattern variable `A`. It is not a physical offset, as with `PREV` or `NEXT`.

The optional integer argument must be a runtime constant (literal, bind variable, and expressions involving them), but not a column or subquery.

The first argument of `FIRST` or `LAST` must have at least one row pattern column reference. Thus, `FIRST(1)` is a syntax error.

The first argument of `FIRST` or `LAST` may have more than one row pattern column reference, in which case all qualifiers must be the same pattern variable. For example, `FIRST (A.Price + B.Tax)` is a syntax error, but `FIRST (A.Price + A.Tax)` is acceptable.

`FIRST` and `LAST` support both running and final semantics. The `RUNNING` keyword is the default, and the only supported option in the `DEFINE` clause. Final semantics can be accessed in the `MEASURES` by using the keyword `FINAL`, as in:

```
MEASURES FINAL LAST (A.Price) AS FinalPrice
ALL ROWS PER MATCH
```

## 22.3.9.6 Running Versus Final Semantics and Keywords

This section discusses some of the considerations to keep in mind when working with `RUNNING` and `FINAL`.

### 22.3.9.6.1 RUNNING Versus FINAL Semantics

Pattern matching in a sequence of rows is usually thought of as an incremental process, with one row after another examined to see if it fits the pattern. With this incremental processing model, at any step until the complete pattern has been recognized, there is only a partial match and it is not known what rows might be added in the future, nor to what variables those future rows might be mapped. Therefore, in pattern matching, a row pattern column reference in the Boolean condition of a `DEFINE` clause has running semantics. This means that a pattern variable represents the set of rows that were already mapped to the pattern variable, up to and including the current row, but not any future rows.

After the complete match is established, it is possible to have final semantics. Final semantics is the same as running semantics on the last row of a successful match. Final semantics is only available in `MEASURES`, because in `DEFINE` there is uncertainty about whether a complete match was achieved.

The keywords `RUNNING` and `FINAL` are used to indicate running or final semantics, respectively; the rules for these keywords are discussed in "RUNNING Versus FINAL Keywords".


Copyright © 2001, 2025, Oracle and/or its affiliates.

The fundamental rule for expression evaluation in `MEASURES` and `DEFINE` is as follows:

- When an expression involving a pattern variable is computed on a group of rows, then the set of rows that is mapped to the pattern variable is used. If the set is empty, then `COUNT` is 0 and any other expression involving the pattern variable is null.
- When an expression requires evaluation in a single row, then the latest row of the set is used. If the set is empty, then the expression is null.

For example, consider the following table and query in Example 22-7.

**Example 22-7 RUNNING Versus FINAL Semantics**

```
SELECT M.Symbol, M.Tstamp, M.Price, M.RunningAvg, M.FinalAvg
FROM TICKER MATCH_RECOGNIZE (
     PARTITION BY Symbol
     ORDER BY tstamp
     MEASURES RUNNING AVG (A.Price) AS RunningAvg,
              FINAL AVG (A.Price) AS FinalAvg
     ALL ROWS PER MATCH
     PATTERN (A+)
     DEFINE A AS A.Price >= AVG (A.Price)
     ) M
;
```

Consider the following ordered row pattern partition of data shown in Table 22-2.

**Table 22-2 Pattern and Partitioned Data**

| Row | Symbol | Timestamp | Price |
|---|---|---|---|
| R1 | XYZ | 09-Jun-09 | 10 |
| R2 | XYZ | 10-Jun-09 | 16 |
| R3 | XYZ | 11-Jun-09 | 13 |
| R4 | XYZ | 12-Jun-09 | 9 |

The following logic can be used to find a match:

- On the first row of the row pattern partition, tentatively map row `R1` to pattern variable `A`. At this point, the set of rows mapped to variable `A` is `{R1}`. To confirm whether this mapping is successful, evaluate the predicate:

  `A.Price >= AVG (A.Price)`

  On the left-hand side, `A.Price` must be evaluated in a single row, which is the last row of the set using running semantics. The last row of the set is `R1`; therefore `A.Price` is 10.

  On the right hand side, `AVG (A.Price)` is an aggregate, which is computed using the rows of the set. This average is 10/1 = 10.

  Thus the predicate asks if `10 >= 10.` The answer is yes, so the mapping is successful. However, the pattern `A+` is greedy, so the query must try to match more rows if possible.

- On the second row of the row pattern partition, tentatively map `R2` to pattern variable `A`. At this point there are two rows mapped to `A`, so the set is `{R1, R2}`. Confirm whether the mapping is successful by evaluating the predicate.

  `A.Price >= AVG (A.Price)`

  On the left hand side, `A.Price` must be evaluated in a single row, which is the last row of the set using running semantics. The last row of the set is `R2`; therefore `A.Price` is 16.On


Copyright © 2001, 2025, Oracle and/or its affiliates.

the right hand side, `AVG (A.Price)` is an aggregate, which is computed using the rows of the set. This average is (10+16)/2 = 13.Thus the predicate asks if 16 >= 13. The answer is yes, so the mapping is successful.

- On the third row of the row pattern partition, tentatively map `R3` to pattern variable `A`. Now there are three rows mapped to `A`, so the set is `{R1, R2, R3}`. Confirm whether the mapping is successful by evaluating the predicate:

  ```
  A.Price >= AVG (A.Price)
  ```

  On the left-hand side, `A.Price` is evaluated in `R3`; therefore, `A.Price` is 13.

  On the right-hand side, `AVG (A.Price)` is an aggregate, which is computed using the rows of the set. This average is (10+16+13)/3 = 13.Thus the predicate asks if 13 >= 13. The answer is yes, so the mapping is successful.

- On the fourth row of the row pattern partition, tentatively map `R4` to pattern variable `A`. At this point, the set is `{R1, R2, R3, R4}`. Confirm whether the mapping is successful by evaluating the predicate:

  ```
  A.Price >= AVG (A.Price)
  ```

  On the left-hand side, `A.Price` is evaluated in `R4`; therefore, `A.Price` is 9.

  On the right-hand side, `AVG (A.Price)` is an aggregate, which is computed using the rows of the set. This average is (10+16+13+9)/4 = 12.Thus the predicate asks if 9 >= 12. The answer is no, so the mapping is not successful.

`R4` did not satisfy the definition of `A`, so the longest match to `A+` is `{R1, R2, R3}`. Because `A+` has a greedy quantifier, this is the preferred match.

The averages computed in the `DEFINE` clause are running averages. In `MEASURES`, especially with `ALL ROWS PER MATCH`, it is possible to distinguish final and running aggregates. Notice the use of the keywords `RUNNING` and `FINAL` in the `MEASURES` clause. The distinction can be observed in the result of the example in Table 22-3.

**Table 22-3 Row Pattern Navigation**

| Symbol | Timestamp | Price | Running Average | Final Average |
|---|---|---|---|---|
| XYZ | 2009-06-09 | 10 | 10 | 13 |
| XYZ | 2009-06-10 | 16 | 13 | 13 |
| XYZ | 2009-06-11 | 13 | 13 | 13 |

It is possible that the set of rows mapped to a pattern variable is empty. When evaluating over an empty set:

- `COUNT` is 0.
- Any other aggregate, row pattern navigation operation, or ordinary pattern column reference is null.

For example:

```
PATTERN ( A? B+ )
DEFINE A AS A.Price > 100,
       B AS B.Price > COUNT (A.*) * 50
```

With the preceding example, consider the following ordered row pattern partition of data in Table 22-4.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Table 22-4 Pattern and Row**

| Row | Price |
|---|---|
| R1 | 60 |
| R2 | 70 |
| R3 | 40 |

A match can be found in this data as follows:

- Tentatively map row `R1` to pattern variable `A`. (The quantifier `?` means to try first for a single match to `A`; if that fails, then an empty match is taken as matching `A?`). To see if the mapping is successful, the predicate `A.Price > 100` is evaluated. `A.Price` is 60; therefore, the predicate is false and the mapping to `A` does not succeed.
- Because the mapping to `A` failed, the empty match is taken as matching `A?`.
- Tentatively map row `R1` to `B`. The predicate to check for this mapping is `B.Price > COUNT (A.*) * 50`

  No rows are mapped to `A`, therefore `COUNT (A.*)` is 0. Because `B.Price = 60` is greater than 0, the mapping is successful.
- Similarly, rows `R2` and `R3` can be successfully mapped to `B`. Because there are no more rows, this is the complete match: no rows mapped `A`, and rows {R1, R2, R3} mapped to `B`.

A pattern variable can make a forward reference, that is, a reference to a pattern variable that was not matched yet. For example:

```
PATTERN (X+ Y+)
DEFINE X AS COUNT (Y.*) > 3,
Y AS Y.Price > 10
```

The previous example is valid syntax. However, this example will never be matched because at the time that a row is mapped to `X`, no row has been mapped to `Y`. Thus `COUNT(Y.*)` is 0 and can never be greater than three. This is true even if there are four future rows that might be successfully mapped to `Y`. Consider this data set in Table 22-5.

**Table 22-5 Pattern and Row**

| Row | Price |
|---|---|
| R1 | 2 |
| R2 | 11 |
| R3 | 12 |
| R4 | 13 |
| R5 | 14 |

Mapping `{R2, R3, R4, R5}` to `Y` would be successful, because all four of these rows satisfy the Boolean condition defined for `Y`. In that case, you might think that you could map `R1` to `X` and have a complete successful match. However, the rules of pattern matching will not find this match, because, according to the pattern `X+ Y+`, at least one row must be mapped to `X` before any rows are mapped to `Y`.


Copyright © 2001, 2025, Oracle and/or its affiliates.

### 22.3.9.6.2 RUNNING Versus FINAL Keywords

`RUNNING` and `FINAL` are keywords used to indicate whether running or final semantics are desired. `RUNNING` and `FINAL` can be used with aggregates and the row pattern navigation operations `FIRST` and `LAST`.

Aggregates, `FIRST` and `LAST` can occur in the following places in a row pattern matching query:

- In the `DEFINE` clause. When processing the `DEFINE` clause, the query is still in the midst of recognizing a match, therefore the only supported semantics is running.
- In the `MEASURES` clause. When processing the `MEASURES` clause, the query has finished recognizing a match; therefore, it becomes possible to consider final semantics. There are two subcases:
  - If `ONE ROW PER MATCH` is specified, then conceptually the query is positioned on the last row of the match, and there is no real difference between running versus final semantics.
  - If `ALL ROWS PER MATCH` is specified, then the row pattern output table will have one row for each row of the match. In this circumstance, the user may wish to see both running and final values, so pattern matching provides the `RUNNING` and `FINAL` keywords to support that distinction.

Based on this analysis, pattern matching specifies the following:

- In `MEASURES`, the keywords `RUNNING` and `FINAL` can be used to indicate the desired semantics for an aggregate, `FIRST` or `LAST`. The keyword is written before the operator, for example, `RUNNING COUNT (A.*)` or `FINAL SUM (B.Price)`.
- In both `MEASURES` and `DEFINE`, the default is `RUNNING`.
- In `DEFINE`, `FINAL` is not permitted; `RUNNING` may be used for added clarity if desired.
- In `MEASURES` with `ONE ROW PER MATCH`, all aggregates, `FIRST`, and `LAST` are computed after the last row of the match is recognized, so that the default `RUNNING` semantics is actually no different from `FINAL` semantics. The user may prefer to think of expressions defaulting to `FINAL` in these cases or the user may choose to write `FINAL` for added clarity.
- Ordinary column references have running semantics. (For `ALL ROWS PER MATCH`, to get final semantics in `MEASURES`, use the `FINAL LAST` row pattern navigation operation instead of an ordinary column reference.)

### 22.3.9.6.3 Ordinary Row Pattern Column References

An ordinary row pattern column reference is one that is neither aggregated nor navigated, for example:

```
A.Price
```

"RUNNING Versus FINAL Keywords" stated that ordinary row pattern column references always have running semantics. This means:

- In `DEFINE`, an ordinary column reference references the last row that is mapped to the pattern variable, up to and including the current row. If there is no such row, then the value is null.
- In `MEASURES`, there are two subcases:
  - If `ALL ROWS PER MATCH` is specified, then there is also a notion of current row, and the semantics are the same as in `DEFINE`.


Copyright © 2001, 2025, Oracle and/or its affiliates.

- If ONE ROW PER MATCH is specified, then conceptually the query is positioned on the last row of the match. An ordinary column reference references the last row that is mapped to the pattern variable. If the variable is not mapped to any row, then the value is null.

These semantics are the same as the LAST operator, with the implicit RUNNING default. Consequently, an ordinary column reference such as X.Price is equivalent to RUNNING LAST (X.Price).

## 22.3.10 Row Pattern Output

The result of MATCH_RECOGNIZE is called the row pattern output table. The shape (row type) of the row pattern output table depends on the choice of ONE ROW PER MATCH or ALL ROWS PER MATCH.

If ONE ROW PER MATCH is specified or implied, then the columns of the row pattern output table are the row pattern partitioning columns in their order of declaration, followed by the row pattern measure columns in their order of declaration. Because a table must have at least one column, this implies that there must be at least one row pattern partitioning column or one row pattern measure column.

If ALL ROWS PER MATCH is specified, then the columns of the row pattern output table are the row pattern partitioning columns in their order of declaration, the ordering columns in their order of declaration, the row pattern measure columns in their order of declaration, and finally any remaining columns of the row pattern input table, in the order they occur in the row pattern input table.

The names and declared types of the pattern measure columns are determined by the MEASURES clause. The names and declared types of the non-measure columns are inherited from the corresponding columns of the pattern input table.

**See Also**

"Correlation Name and Row Pattern Output" for information about assigning a correlation name to row pattern output

### 22.3.10.1 Correlation Name and Row Pattern Output

A correlation name can be assigned to the row pattern output table, similar to the following:

```
SELECT M.Matchno
FROM Ticker MATCH_RECOGNIZE (...
     MEASURE MATCH_NUMBER() AS Matchno
     ...
) M
```

In the preceding example, M is the correlation name assigned to the row pattern output table. The benefit to assigning a correlation name is that the correlation name can be used to qualify the column names of the row pattern output table, as in M.Matchno in the preceding example. This is especially important to resolve ambiguous column names if there are other tables in the FROM clause.

# 22.4 Advanced Topics in Pattern Matching

This section discusses the following advanced topics:


Copyright © 2001, 2025, Oracle and/or its affiliates.

- Nesting FIRST and LAST Within PREV and NEXT in Pattern Matching
- Handling Empty Matches or Unmatched Rows in Pattern Matching
- How to Exclude Portions of the Pattern from the Output
- How to Express All Permutations

## 22.4.1 Nesting FIRST and LAST Within PREV and NEXT in Pattern Matching

`FIRST` and `LAST` provide navigation within the set of rows already mapped to a particular pattern variable; `PREV` and `NEXT` provide navigation using a physical offset from a particular row. These kinds of navigation can be combined by nesting `FIRST` or `LAST` within `PREV` or `NEXT`. This permits expressions such as the following:

```
PREV (LAST (A.Price + A.Tax, 1), 3)
```

In this example, `A` must be a pattern variable. It is required to have a row pattern column reference, and all pattern variables in the compound operator must be equivalent (`A`, in this example).

This compound operator is evaluated as follows:

1. The inner operator, `LAST`, operates solely on the set of rows that are mapped to the pattern variable `A`. In this set, find the row that is *the last minus 1*. (If there is no such row, the result is null.)
2. The outer operator, `PREV`, starts from the row found in Step 1 and backs up three rows in the row pattern partition. (If there is no such row, the result is null.)
3. Let `R` be an implementation-dependent range variable that references the row found by Step 2. In the expression `A.Price + A.Tax`, replace every occurrence of the pattern variable `A` with `R`. The resulting expression `R.Price + R.Tax` is evaluated and determines the value of the compound navigation operation.

For example, consider the data set and mappings in Table 22-6.

**Table 22-6 Data Set and Mappings**

| Row | Price | Tax | Mapping |
|---|---|---|---|
| R1 | 10 | 1 | |
| R2 | 20 | 2 | A |
| R3 | 30 | 3 | B |
| R4 | 40 | 4 | A |
| R5 | 50 | 5 | C |
| R6 | 60 | 6 | A |

To evaluate `PREV (LAST (A.Price + A.Tax, 1), 3)`, the following steps can be used:

- The set of rows mapped to `A` is `{R2, R4, R6}`. `LAST` operates on this set, offsetting from the end to arrive at row `R4`.
- `PREV` performs a physical offset, 3 rows before `R4`, arriving at `R1`.
- Let `R` be a range variable pointing at `R1`. `R.Price + R.Tax` is evaluated, giving 10+1 = 11.


Copyright © 2001, 2025, Oracle and/or its affiliates.

Note that this nesting is not defined as a typical evaluation of nested functions. The inner operator `LAST` does not actually evaluate the expression `A.Price + A.Tax`; it uses this expression to designate a pattern variable (`A`) and then navigate within the rows mapped to that variable. The outer operator `PREV` performs a further physical navigation on rows. The expression `A.Price + A.Tax` is not actually evaluated as such, because the row that is eventually reached is not necessarily mapped to the pattern variable `A`. In this example, `R1` is not mapped to any pattern variable.

## 22.4.2 Handling Empty Matches or Unmatched Rows in Pattern Matching

`ALL ROWS PER MATCH` has three suboptions:

- `ALL ROWS PER MATCH SHOW EMPTY MATCHES`
- `ALL ROWS PER MATCH OMIT EMPTY MATCHES`
- `ALL ROWS PER MATCH WITH UNMATCHED ROWS`

These options are explained in the following topics:

- Handling Empty Matches in Pattern Matching
- Handling Unmatched Rows in Pattern Matching

### 22.4.2.1 Handling Empty Matches in Pattern Matching

Some patterns permit empty matches. For example, `PATTERN (A*)` can be matched by zero or more rows that are mapped to `A`.

An empty match does not map any rows to pattern variables; nevertheless, an empty match has a starting row. For example, there can be an empty match at the first row of a partition, an empty match at the second row of a partition, and so on. An empty match is assigned a sequential match number, based on the ordinal position of its starting row, the same as any other match.

When using `ONE ROW PER MATCH`, an empty match results in one row of the output table. The row pattern measures for an empty match are computed as follows:

- The value of `MATCH_NUMBER()` is the sequential match number of the empty match.
- Any `COUNT` is 0.
- Any other aggregate, row pattern navigation operation, or ordinary row pattern column reference is null.

As for `ALL ROWS PER MATCH`, the question arises, whether to generate a row of output for an empty match, because there are no rows in the empty match. To govern this, there are two options:

- `ALL ROWS PER MATCH SHOW EMPTY MATCHES`: with this option, any empty match generates a single row in the row pattern output table.
- `ALL ROWS PER MATCH OMIT EMPTY MATCHES`: with this option, an empty match is omitted from the row pattern output table. (This may cause gaps in the sequential match numbering.)

`ALL ROWS PER MATCH` defaults to `SHOW EMPTY MATCHES`. Using this option, an empty match generates one row in the row pattern output table. In this row:

- The value of the `CLASSIFIER()` function is null.
- The value of the `MATCH_NUMBER ()` function is the sequential match number of the empty match.


Copyright © 2001, 2025, Oracle and/or its affiliates.

- The value of any ordinary row pattern column reference is null.
- The value of any aggregate or row pattern navigation operation is computed using an empty set of rows (so any COUNT is 0, and all other aggregates and row pattern navigation operations are null).
- The value of any column corresponding to a column of the row pattern input table is the same as the corresponding column in the starting row of the empty match.

### 22.4.2.2 Handling Unmatched Rows in Pattern Matching

Some rows of the row pattern input table may be neither the starting row of an empty match, nor mapped by a non-empty match. Such rows are called unmatched rows.

The option ALL ROWS PER MATCH WITH UNMATCHED ROWS shows both empty matches and unmatched rows. Empty matches are handled the same as with SHOW EMPTY MATCHES. When displaying an unmatched row, all row pattern measures are null, somewhat analogous to the null-extended side of an outer join. Thus, COUNT and MATCH_NUMBER may be used to distinguish an unmatched row from the starting row of an empty match. The exclusion syntax {- -} is prohibited as contrary to the spirit of WITH UNMATCHED ROWS. See "How to Exclude Portions of the Pattern from the Output" for more information.

It is not possible for a pattern to permit empty matches and also have unmatched rows. The reason is that if a row of the row pattern input table cannot be mapped to a primary row pattern variable, then that row can still be the starting row of an empty match, and will not be regarded as unmatched, assuming that the pattern permits empty matches. Thus, if a pattern permits empty matches, then the output using ALL ROWS PER MATCH SHOW EMPTY MATCHES is the same as the output using ALL ROWS PER MATCH WITH UNMATCHED ROWS. Thus WITH UNMATCHED ROWS is primarily intended for use with patterns that do not permit empty matches. However, the user may prefer to specify WITH UNMATCHED ROWS if the user is uncertain whether a pattern may have empty matches or unmatched rows.

Note that if ALL ROWS PER MATCH WITH UNMATCHED ROWS is used with the default skipping behavior (AFTER MATCH SKIP PAST LAST ROW), then there is exactly one row in the output for every row in the input.

Other skipping behaviors are permitted using WITH UNMATCHED ROWS, in which case it becomes possible for a row to be mapped by more than one match and appear in the row pattern output table multiple times. Unmatched rows will appear in the output only once.

## 22.4.3 How to Exclude Portions of the Pattern from the Output

When using ALL ROWS PER MATCH with either the OMIT EMPTY MATCHES or SHOW EMPTY MATCHES suboptions, rows matching a portion of the PATTERN may be excluded from the row pattern output table. The excluded portion is bracketed between {- and -} in the PATTERN clause.

For example, the following example finds the longest periods of increasing prices that start with a price no less than ten.

**Example 22-8 Periods of Increasing Prices**

```
SELECT M.Symbol, M.Tstamp, M.Matchno, M.Classfr, M.Price, M.Avgp
FROM Ticker MATCH_RECOGNIZE (
     PARTITION BY Symbol
     ORDER BY tstamp
     MEASURES FINAL AVG(S.Price) AS Avgp,
              CLASSIFIER() AS Classfr,
              MATCH_NUMBER() AS Matchno
     ALL ROWS PER MATCH
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
     AFTER MATCH SKIP TO LAST B
     PATTERN ( {- A -} B+ {- C+ -} )
     SUBSET S = (A,B)
     DEFINE
        A AS A.Price >= 10,
        B AS B.Price > PREV(B.Price),
        C AS C.Price <= PREV(C.Price)
)  M
ORDER BY symbol, tstamp;

SYMBOL    TSTAMP          MATCHNO CLASSFR
PRICE                                      AVGP
_________ _____________ __________ __________ _________
_____________________________________________
ACME      02-APR-22             1 B
17                                        18.8
ACME      03-APR-22             1 B
19                                        18.8
ACME      04-APR-22             1 B
21                                        18.8
ACME      05-APR-22             1 B
25                                        18.8
ACME      07-APR-22             2 B
15                                        19.2
ACME      08-APR-22             2 B
20                                        19.2
ACME      09-APR-22             2 B
24                                        19.2
ACME      10-APR-22             2 B
25                                        19.2
ACME      13-APR-22             3 B
25                                          20
ACME      17-APR-22             4 B                 14
16.666666666666666666666666666666666666667
ACME      18-APR-22             4 B                 24
16.666666666666666666666666666666666666667
```

The row pattern output table will only have rows that are mapped to `B`, the rows mapped to `A` and `C` will be excluded from the output. Although the excluded rows do not appear in the row pattern output table, they are not excluded from the definitions of union pattern variables, or from the calculation of scalar expressions in the `DEFINE` or `MEASURES`. For example, see the definitions of the primary pattern variables `A` and `C`, the definition of union pattern variable `S`, or the `Avgp` row pattern measure in the previous example.

The exclusion syntax is not permitted with `ALL ROWS PER MATCH WITH UNMATCHED ROWS`.

The exclusion syntax is permitted with `ONE ROW PER MATCH`, though it has no effect because in this case there is only a single summary row per match.

## 22.4.4 How to Express All Permutations

The `PERMUTE` syntax may be used to express a pattern that is a permutation of simpler patterns. For example, `PATTERN (PERMUTE (A, B, C))` is equivalent to an alternation of all permutations of three pattern variables `A`, `B`, and `C`, similar to the following:

```
PATTERN (A B C | A C B | B A C | B C A | C A B | C B A)
```

Note that `PERMUTE` is expanded lexicographically and that each element to permute must be comma-separated from the other elements. (In this example, because the three pattern variables `A`, `B`, and `C` are listed in alphabetic order, it follows from lexicographic expansion that


Copyright © 2001, 2025, Oracle and/or its affiliates.

the expanded possibilities are also listed in alphabetic order.) This is significant because alternatives are attempted in the order written in the expansion. Thus a match to `(A B C)` is attempted before a match to `(A C B)`, and so on; the first attempt that succeeds is what can be called the "winner".

As another example:

```
PATTERN (PERMUTE (X{3}, B C?, D))
```

This is equivalent to the following:

```
PATTERN ((X{3} B C? D)
| (X{3} D B C?)
| (B C? X{3} D)
| (B C? D X{3})
| (D X{3} B C?)
| (D B C? X{3}))
```

Note that the pattern elements "`B C?`" are not comma-separated and so they are treated as a single unit.)

# 22.5 Rules and Restrictions in Pattern Matching

This section discusses the following rules and restrictions:

- Input Table Requirements in Pattern Matching
- Prohibited Nesting in the MATCH_RECOGNIZE Clause
- Concatenated MATCH_RECOGNIZE Clause
- Aggregate Restrictions

## 22.5.1 Input Table Requirements in Pattern Matching

The row pattern input table is the input argument to `MATCH_RECOGNIZE`. You can use a table or view, or a named query (defined in a `WITH` clause). The row pattern input table can also be a derived table (also known as in-line view). For example.

```
FROM (SELECT S.Name, T.Tstamp, T.Price
      FROM Ticker T, SymbolNames S
      WHERE T.Symbol = S.Symbol)
MATCH_RECOGNIZE (...) M
```

The row pattern input table cannot be a joined table. The work-around is to use a derived table, such as the following:

```
FROM (SELECT * FROM A LEFT OUTER JOIN B ON (A.X = B.Y))
MATCH_RECOGNIZE (...) M
```

Column names in the pattern input table must be unambiguous. If the row pattern input table is a base table or a view, this is not a problem, because SQL does not allow ambiguous column names in a base table or view. This is only an issue when the row pattern input table is a derived table. For example, consider a join of two tables, `Emp` and `Dept`, each of which has a column called `Name`. The following is a syntax error:

```
FROM (SELECT D.Name, E.Name, E.Empno, E.Salary
      FROM Dept D, Emp E
      WHERE D.Deptno = E.Deptno)
MATCH_RECOGNIZE (
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
      PARTITION BY D.Name
   ...)
```

The previous example is an error because the variable `D` is not visible within the `MATCH_RECOGNIZE` clause (The scope of `D` is just the derived table). Rewriting similar to the following does not help:

```
FROM (SELECT D.Name, E.Name, E.Empno, E.Salary
      FROM Dept D, Emp E
      WHERE D.Deptno = E.Deptno)
MATCH_RECOGNIZE (
      PARTITION BY Name
   ...)
```

This rewrite eliminates the use of the variable `D` within the `MATCH_RECOGNIZE` clause. However, now the error is that `Name` is ambiguous, because there are two columns of the derived table called `Name`. The way to handle this is to disambiguate the column names within the derived table itself, similar to the following:

```
FROM (SELECT D.Name AS Dname, E.Name AS Ename,
         E.Empno, E.Salary
      FROM Dept D, Emp E
      WHERE D.Deptno = E.Deptno)
MATCH_RECOGNIZE (
      PARTITION BY Dname
   ...)
```

**See Also**

*Oracle Database SQL Language Reference*

## 22.5.2 Prohibited Nesting in the MATCH_RECOGNIZE Clause

The following kinds of nesting are prohibited in the `MATCH_RECOGNIZE` clause:

- Nesting one `MATCH_RECOGNIZE` clause within another.
- Outer references in the `MEASURES` clause or the `DEFINE` subclause. This means that a `MATCH_RECOGNIZE` clause cannot reference any table in an outer query block except the row pattern input table.
- Correlated subqueries cannot be used in `MEASURES` or `DEFINE`. Also, subqueries in `MEASURES` or `DEFINE` cannot reference pattern variables.
- The `MATCH_RECOGNIZE` clause cannot be used in recursive queries.
- The `SELECT FOR UPDATE` statement cannot use the `MATCH_RECOGNIZE` clause.

## 22.5.3 Concatenated MATCH_RECOGNIZE Clause

Note that it is not prohibited to feed the output of one `MATCH_RECOGNIZE` clause into the input of another, as in this example:

```
SELECT ...
FROM ( SELECT *
       FROM Ticker
            MATCH_RECOGNIZE (...) )
       MATCH_RECOGNIZE (...)
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

In this example, the first `MATCH_RECOGNIZE` clause is in a derived table, which then provides the input to the second `MATCH_RECOGNIZE`.

## 22.5.4 Aggregate Restrictions

The aggregate functions `COUNT`, `SUM`, `AVG`, `MAX`, and `MIN` can be used in both the `MEASURES` and `DEFINE` clauses. The `DISTINCT` keyword is not supported.

# 22.6 Examples of Pattern Matching

This section contains the following types of advanced pattern matching examples:

- Pattern Matching Examples: Stock Market
- Pattern Matching Examples: Security Log Analysis
- Pattern Matching Examples: Sessionization
- Pattern Matching Example: Financial Tracking

## 22.6.1 Pattern Matching Examples: Stock Market

This section contains pattern matching examples that are based on common tasks involving share prices and patterns.

**Example 22-9 Price Dips of a Specified Magnitude**

The query in Example 22-9 shows stocks where the current price is more than a specific percentage (in this example 8%) below the prior day's closing price.

```
CREATE TABLE Ticker3Wave (SYMBOL VARCHAR2(10), tstamp DATE, PRICE NUMBER);

INSERT INTO Ticker3Wave VALUES('ACME', '01-Apr-22', 1000);
INSERT INTO Ticker3Wave VALUES('ACME', '02-Apr-22', 775);
INSERT INTO Ticker3Wave VALUES('ACME', '03-Apr-22', 900);
INSERT INTO Ticker3Wave VALUES('ACME', '04-Apr-22', 775);
INSERT INTO Ticker3Wave VALUES('ACME', '05-Apr-22', 900);
INSERT INTO Ticker3Wave VALUES('ACME', '06-Apr-22', 775);
INSERT INTO Ticker3Wave VALUES('ACME', '07-Apr-22', 900);
INSERT INTO Ticker3Wave VALUES('ACME', '08-Apr-22', 775);
INSERT INTO Ticker3Wave VALUES('ACME', '09-Apr-22', 800);
INSERT INTO Ticker3Wave VALUES('ACME', '10-Apr-22', 550);
INSERT INTO Ticker3Wave VALUES('ACME', '11-Apr-22', 900);
INSERT INTO Ticker3Wave VALUES('ACME', '12-Apr-22', 800);
INSERT INTO Ticker3Wave VALUES('ACME', '13-Apr-22', 1100);
INSERT INTO Ticker3Wave VALUES('ACME', '14-Apr-22', 800);
INSERT INTO Ticker3Wave VALUES('ACME', '15-Apr-22', 550);
INSERT INTO Ticker3Wave VALUES('ACME', '16-Apr-22', 800);
INSERT INTO Ticker3Wave VALUES('ACME', '17-Apr-22', 875);
INSERT INTO Ticker3Wave VALUES('ACME', '18-Apr-22', 950);
INSERT INTO Ticker3Wave VALUES('ACME', '19-Apr-22', 600);
INSERT INTO Ticker3Wave VALUES('ACME', '20-Apr-22', 300);

SELECT *
FROM Ticker3Wave MATCH_RECOGNIZE (
     PARTITION BY symbol
     ORDER BY tstamp
     MEASURES B.tstamp AS timestamp,
              A.price AS Aprice,
              B.price AS Bprice,
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
            ((B.price - A.price)*100) / A.price AS PctDrop
    ONE ROW PER MATCH
    AFTER MATCH SKIP TO B
    PATTERN (A B)
    DEFINE
       B AS (B.price - A.price) / A.price  < -0.08
   );

 SYMBOL    TIMESTAMP       APRICE    BPRICE
PCTDROP
_________ ___________ _________ _________ ________________________________________
ACME      02-APR-22         1000       775                                  -22.5
ACME      04-APR-22          900       775   -13.888888888888888888888888888888888889
ACME      06-APR-22          900       775   -13.888888888888888888888888888888888889
ACME      08-APR-22          900       775   -13.888888888888888888888888888888888889
ACME      10-APR-22          800       550                                -31.25
ACME      12-APR-22          900       800   -11.111111111111111111111111111111111111
ACME      14-APR-22         1100       800   -27.272727272727272727272727272727272727
ACME      15-APR-22          800       550                                -31.25
ACME      19-APR-22          950       600   -36.842105263157894736842105263157894736
ACME      20-APR-22          600       300                                  -50
```

### Example 22-10 Prices Dips of Specified Magnitude When They Have Returned to the Original Price

The query in Example 22-10 extends the pattern defined in Example 22-9. It finds a stock with a price drop of more than 8%. It also seeks zero or more additional days when the stock price remains below the original price. Then, it identifies when the stock has risen in price to equal or exceed its initial value. Because it can be useful to know the number of days that the pattern occurs, it is included here. The `start_price` column is the starting price of a match and the `end_price` column is the end price of a match, when the price is equal to or greater than the start price.

```
SELECT *
 FROM Ticker3Wave MATCH_RECOGNIZE (
   PARTITION BY symbol
   ORDER BY tstamp
   MEASURES
      A.tstamp      as start_timestamp,
      A.price       as start_price,
      B.price       as drop_price,
      COUNT(C.*)+1  as cnt_days,
      D.tstamp      as end_timestamp,
      D.price       as end_price
   ONE ROW PER MATCH
   AFTER MATCH SKIP PAST LAST ROW
   PATTERN (A B C* D)
   DEFINE
      B as (B.price - A.price)/A.price < -0.08,
      C as C.price < A.price,
      D as D.price >= A.price
   );

SYMBOL     START_TIM START_PRICE DROP_PRICE  CNT_DAYS  END_TIMES  END_PRICE
---------- --------- ----------- ----------  --------  ---------  ----------
ACME       01-Apr-22        1000        775        11  13-Apr-22        1100
ACME       14-Apr-22         800        550         1  16-Apr-22         800
```

### Example 22-11 Find both V and U Shapes in Trading History

Example 22-11 shows how important it is to take all possible data behavior into account when defining a pattern. The table `TickerVU` is just like the first example's table `Ticker`, except that it


Copyright © 2001, 2025, Oracle and/or its affiliates.

has two equal-price days in a row at the low point of its third bottom, April 16 and 17. This sort of flat bottom price drop is called a U-shape. Can the original example, Example 22-1, recognize that the modified data is a lot like a V-shape, and include the U-shape in its output? No, the query needs to be modified as shown.

```
CREATE TABLE TickerVU (SYMBOL VARCHAR2(10), tstamp DATE, PRICE NUMBER);

INSERT INTO TickerVU values('ACME', '01-Apr-22', 12);
INSERT INTO TickerVU values('ACME', '02-Apr-22', 17);
INSERT INTO TickerVU values('ACME', '03-Apr-22', 19);
INSERT INTO TickerVU values('ACME', '04-Apr-22', 21);
INSERT INTO TickerVU values('ACME', '05-Apr-22', 25);
INSERT INTO TickerVU values('ACME', '06-Apr-22', 12);
INSERT INTO TickerVU values('ACME', '07-Apr-22', 15);
INSERT INTO TickerVU values('ACME', '08-Apr-22', 20);
INSERT INTO TickerVU values('ACME', '09-Apr-22', 24);
INSERT INTO TickerVU values('ACME', '10-Apr-22', 25);
INSERT INTO TickerVU values('ACME', '11-Apr-22', 19);
INSERT INTO TickerVU values('ACME', '12-Apr-22', 15);
INSERT INTO TickerVU values('ACME', '13-Apr-22', 25);
INSERT INTO TickerVU values('ACME', '14-Apr-22', 25);
INSERT INTO TickerVU values('ACME', '15-Apr-22', 14);
INSERT INTO TickerVU values('ACME', '16-Apr-22', 12);
INSERT INTO TickerVU values('ACME', '17-Apr-22', 12);
INSERT INTO TickerVU values('ACME', '18-Apr-22', 24);
INSERT INTO TickerVU values('ACME', '19-Apr-22', 23);
INSERT INTO TickerVU values('ACME', '20-Apr-22', 22);
```

What happens if you run your original query of Example 22-1, modified to use this table name?

```
SELECT *
FROM TickerVU MATCH_RECOGNIZE (
     PARTITION BY symbol
     ORDER BY tstamp
     MEASURES STRT.tstamp AS start_tstamp,
              DOWN.tstamp AS bottom_tstamp,
              UP.tstamp AS end_tstamp
    ONE ROW PER MATCH
    AFTER MATCH SKIP TO LAST UP
    PATTERN (STRT DOWN+ UP+)
    DEFINE DOWN AS DOWN.price < PREV(DOWN.price),
           UP AS UP.price > PREV(UP.price)
) MR
ORDER BY MR.symbol, MR.start_tstamp;

SYMBOL     START_TST BOTTOM_TS END_TSTAM
---------- --------- --------- ---------
ACME       05-Apr-22 06-Apr-22 10-Apr-22
ACME       10-Apr-22 12-Apr-22 13-Apr-22
```

Instead of showing three rows of output (one per price drop), the query shows only two. This happens because no variable was defined to handle a flat stretch of data at the bottom of a price dip. Now, use a modified version of this query, adding a variable for flat data in the `DEFINE` clause and using that variable in the `PATTERN` clause.

```
SELECT *
FROM TickerVU MATCH_RECOGNIZE (
     PARTITION BY symbol
     ORDER BY tstamp
     MEASURES STRT.tstamp AS start_tstamp,
              DOWN.tstamp AS bottom_tstamp,
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
             UP.tstamp AS end_tstamp
     ONE ROW PER MATCH
     AFTER MATCH SKIP TO LAST UP
     PATTERN (STRT DOWN+ FLAT* UP+)
     DEFINE
         DOWN AS DOWN.price < PREV(DOWN.price),
         FLAT AS FLAT.price = PREV(FLAT.price),
         UP AS UP.price > PREV(UP.price)
) MR
ORDER BY MR.symbol, MR.start_tstamp;

SYMBOL     START_TST   BOTTOM_TS   END_TSTAM
---------- ---------   ---------   ---------
ACME       05-Apr-22   06-Apr-22   10-Apr-22
ACME       10-Apr-22   12-Apr-22   13-Apr-22
ACME       14-Apr-22   16-Apr-22   18-Apr-22
```

Now, you get output that includes all three price dips in the data. The lesson here is to consider all possible variations in your data sequence and include those possibilities in your `PATTERN`, `DEFINE`, and `MEASURES` clauses as needed.

### Example 22-12 Finding Elliott Wave Pattern: Multiple Instances of Inverted-V

Example 22-12 shows a simple version of a class of stock price patterns referred to as the Elliott Wave which has multiple consecutive patterns of inverted V-shapes. In this particular case, the pattern expression searches for 1 or more days up followed by 1 or more days down, and this sequence must appear five times consecutively with no gaps. That is, the pattern looks similar to: /\/\/\/\.

```
SELECT MR_ELLIOTT.*
FROM Ticker3Wave MATCH_RECOGNIZE (
     PARTITION BY symbol
     ORDER BY tstamp
     MEASURES
             COUNT(*) as CNT,
             COUNT(P.*) AS CNT_P,
             COUNT(Q.*) AS CNT_Q,
             COUNT(R.*) AS CNT_R,
             COUNT(S.*) AS CNT_S,
             COUNT(T.*) AS CNT_T,
             COUNT(U.*) AS CNT_U,
             COUNT(V.*) AS CNT_V,
             COUNT(W.*) AS CNT_W,
             COUNT(X.*) AS CNT_X,
             COUNT(Y.*) AS CNT_Y,
             COUNT(Z.*) AS CNT_Z,
             CLASSIFIER() AS CLS,
     MATCH_NUMBER() AS MNO
     ALL ROWS PER MATCH
     AFTER MATCH SKIP TO LAST Z
     PATTERN (P Q+ R+ S+ T+ U+ V+ W+ X+ Y+ Z+)
     DEFINE
        Q AS Q.price > PREV(Q.price),
        R AS R.price < PREV(R.price),
        S AS S.price > PREV(S.price),
        T AS T.price < PREV(T.price),
        U AS U.price > PREV(U.price),
        V AS V.price < PREV(V.price),
        W AS W.price > PREV(W.price),
        X AS X.price < PREV(X.price),
        Y AS Y.price > PREV(Y.price),
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
                    Z AS Z.price < PREV(Z.price)
                  ) MR_ELLIOTT
                ORDER BY symbol, tstamp;

SYMB TSTAMP    CNT  CNT_P CNT_Q CNT_R CNT_S CNT_T CNT_U CNT_V CNT_W CNT_X CNT_Y CNT_Z CLS MNO PRICE
---- --------- ---- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- ----- --- --- -----
ACME 02-Apr-22    1     1     0     0     0     0     0     0     0     0     0     0 P     1   775
ACME 03-Apr-22    2     1     1     0     0     0     0     0     0     0     0     0 Q     1   900
ACME 04-Apr-22    3     1     1     1     0     0     0     0     0     0     0     0 R     1   775
ACME 05-Apr-22    4     1     1     1     1     0     0     0     0     0     0     0 S     1   900
ACME 06-Apr-22    5     1     1     1     1     1     0     0     0     0     0     0 T     1   775
ACME 07-Apr-22    6     1     1     1     1     1     1     0     0     0     0     0 U     1   900
ACME 08-Apr-22    7     1     1     1     1     1     1     1     0     0     0     0 V     1   775
ACME 09-Apr-22    8     1     1     1     1     1     1     1     1     0     0     0 W     1   800
ACME 10-Apr-22    9     1     1     1     1     1     1     1     1     1     0     0 X     1   550
ACME 11-Apr-22   10     1     1     1     1     1     1     1     1     1     1     0 Y     1   900
ACME 12-Apr-22   11     1     1     1     1     1     1     1     1     1     1     1 Z     1   800

11 rows selected.
```

### Example 22-13 Finding Elliott Waves and Specifying a Range of Acceptable Row Counts

Similar to Example 22-12, Example 22-13 specifies an Elliott Wave of inverted Vs. However, in this case, regular expressions are used to specify for each pattern variable the number of consecutive rows to match, and it is specified as a range. Set each pattern variable to seek three or four consecutive matches, using the syntax `"{3,4}"`. The output shows all rows for one full match of the pattern and lets you see exactly when each pattern variable has its beginning and end. Note that variables `W` and `X` each have four rows which match, while variables `Y` and `Z` each have only three rows matching.

```
CREATE TABLE tickerwavemulti (symbol VARCHAR2(10), tstamp DATE, price NUMBER);

INSERT INTO tickerwavemulti VALUES('ACME', '01-May-22', 36.25 );
INSERT INTO tickerwavemulti VALUES('BLUE', '01-May-22', 177.85);
INSERT INTO tickerwavemulti VALUES('EDGY', '01-May-22', 27.18);
INSERT INTO tickerwavemulti VALUES('ACME', '02-May-22', 36.47);
INSERT INTO tickerwavemulti VALUES('BLUE', '02-May-22', 177.25);
INSERT INTO tickerwavemulti VALUES('EDGY', '02-May-22', 27.41);
INSERT INTO tickerwavemulti VALUES('ACME', '03-May-22', 36.36);
INSERT INTO tickerwavemulti VALUES('BLUE', '03-May-22', 176.16);
INSERT INTO tickerwavemulti VALUES('EDGY', '03-May-22', 27.43);
INSERT INTO tickerwavemulti VALUES('ACME', '04-May-22', 36.25);
INSERT INTO tickerwavemulti VALUES('BLUE', '04-May-22', 176.28);
INSERT INTO tickerwavemulti VALUES('EDGY', '04-May-22', 27.56);
INSERT INTO tickerwavemulti VALUES('ACME', '05-May-22', 36.36);
INSERT INTO tickerwavemulti VALUES('BLUE', '05-May-22', 177.72);
INSERT INTO tickerwavemulti VALUES('EDGY', '05-May-22', 27.31);
INSERT INTO tickerwavemulti VALUES('ACME', '06-May-22', 36.70);
INSERT INTO tickerwavemulti VALUES('BLUE', '06-May-22', 178.36);
INSERT INTO tickerwavemulti VALUES('EDGY', '06-May-22', 27.23);
INSERT INTO tickerwavemulti VALUES('ACME', '07-May-22', 36.50);
INSERT INTO tickerwavemulti VALUES('BLUE', '07-May-22', 178.93);
INSERT INTO tickerwavemulti VALUES('EDGY', '07-May-22', 27.08);
INSERT INTO tickerwavemulti VALUES('ACME', '08-May-22', 36.66);
INSERT INTO tickerwavemulti VALUES('BLUE', '08-May-22', 178.18);
INSERT INTO tickerwavemulti VALUES('EDGY', '08-May-22', 26.90);
INSERT INTO tickerwavemulti VALUES('ACME', '09-May-22', 36.98);
INSERT INTO tickerwavemulti VALUES('BLUE', '09-May-22', 179.15);
INSERT INTO tickerwavemulti VALUES('EDGY', '09-May-22', 26.73);
INSERT INTO tickerwavemulti VALUES('ACME', '10-May-22', 37.08);
INSERT INTO tickerwavemulti VALUES('BLUE', '10-May-22', 180.39);
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
INSERT INTO tickerwavemulti VALUES('EDGY', '10-May-22', 26.86);
INSERT INTO tickerwavemulti VALUES('ACME', '11-May-22', 37.43);
INSERT INTO tickerwavemulti VALUES('BLUE', '11-May-22', 181.44);
INSERT INTO tickerwavemulti VALUES('EDGY', '11-May-22', 26.78);
INSERT INTO tickerwavemulti VALUES('ACME', '12-May-22', 37.68);
INSERT INTO tickerwavemulti VALUES('BLUE', '12-May-22', 183.11);
INSERT INTO tickerwavemulti VALUES('EDGY', '12-May-22', 26.59);
INSERT INTO tickerwavemulti VALUES('ACME', '13-May-22', 37.66);
INSERT INTO tickerwavemulti VALUES('BLUE', '13-May-22', 181.50);
INSERT INTO tickerwavemulti VALUES('EDGY', '13-May-22', 26.39);
INSERT INTO tickerwavemulti VALUES('ACME', '14-May-22', 37.32);
INSERT INTO tickerwavemulti VALUES('BLUE', '14-May-22', 180.65);
INSERT INTO tickerwavemulti VALUES('EDGY', '14-May-22', 26.31);
INSERT INTO tickerwavemulti VALUES('ACME', '15-May-22', 37.16);
INSERT INTO tickerwavemulti VALUES('BLUE', '15-May-22', 179.51);
INSERT INTO tickerwavemulti VALUES('EDGY', '15-May-22', 26.53);
INSERT INTO tickerwavemulti VALUES('ACME', '16-May-22', 36.98);
INSERT INTO tickerwavemulti VALUES('BLUE', '16-May-22', 180.00);
INSERT INTO tickerwavemulti VALUES('EDGY', '16-May-22', 26.76);
INSERT INTO tickerwavemulti VALUES('ACME', '17-May-22', 37.19);
INSERT INTO tickerwavemulti VALUES('BLUE', '17-May-22', 179.24);
INSERT INTO tickerwavemulti VALUES('EDGY', '17-May-22', 26.63);
INSERT INTO tickerwavemulti VALUES('ACME', '18-May-22', 37.45);
INSERT INTO tickerwavemulti VALUES('BLUE', '18-May-22', 180.48);
INSERT INTO tickerwavemulti VALUES('EDGY', '18-May-22', 26.84);
INSERT INTO tickerwavemulti VALUES('ACME', '19-May-22', 37.79);
INSERT INTO tickerwavemulti VALUES('BLUE', '19-May-22', 181.21);
INSERT INTO tickerwavemulti VALUES('EDGY', '19-May-22', 26.90);
INSERT INTO tickerwavemulti VALUES('ACME', '20-May-22', 37.49);
INSERT INTO tickerwavemulti VALUES('BLUE', '20-May-22', 179.79);
INSERT INTO tickerwavemulti VALUES('EDGY', '20-May-22', 27.06);
INSERT INTO tickerwavemulti VALUES('ACME', '21-May-22', 37.30);
INSERT INTO tickerwavemulti VALUES('BLUE', '21-May-22', 181.19);
INSERT INTO tickerwavemulti VALUES('EDGY', '21-May-22', 27.17);
INSERT INTO tickerwavemulti VALUES('ACME', '22-May-22', 37.08);
INSERT INTO tickerwavemulti VALUES('BLUE', '22-May-22', 179.88);
INSERT INTO tickerwavemulti VALUES('EDGY', '22-May-22', 26.95);
INSERT INTO tickerwavemulti VALUES('ACME', '23-May-22', 37.34);
INSERT INTO tickerwavemulti VALUES('BLUE', '23-May-22', 181.21);
INSERT INTO tickerwavemulti VALUES('EDGY', '23-May-22', 26.71);
INSERT INTO tickerwavemulti VALUES('ACME', '24-May-22', 37.54);
INSERT INTO tickerwavemulti VALUES('BLUE', '24-May-22', 181.94);
INSERT INTO tickerwavemulti VALUES('EDGY', '24-May-22', 26.96);
INSERT INTO tickerwavemulti VALUES('ACME', '25-May-22', 37.69);
INSERT INTO tickerwavemulti VALUES('BLUE', '25-May-22', 180.88);
INSERT INTO tickerwavemulti VALUES('EDGY', '25-May-22', 26.72);
INSERT INTO tickerwavemulti VALUES('ACME', '26-May-22', 37.60);
INSERT INTO tickerwavemulti VALUES('BLUE', '26-May-22', 180.72);
INSERT INTO tickerwavemulti VALUES('EDGY', '26-May-22', 26.47);
INSERT INTO tickerwavemulti VALUES('ACME', '27-May-22', 37.93);
INSERT INTO tickerwavemulti VALUES('BLUE', '27-May-22', 181.54);
INSERT INTO tickerwavemulti VALUES('EDGY', '27-May-22', 26.73);
INSERT INTO tickerwavemulti VALUES('ACME', '28-May-22', 38.17);
INSERT INTO tickerwavemulti VALUES('BLUE', '28-May-22', 182.93);
INSERT INTO tickerwavemulti VALUES('EDGY', '28-May-22', 26.89);

SELECT MR_EW.*
FROM tickerwavemulti MATCH_RECOGNIZE (
     PARTITION by symbol
     ORDER by tstamp
     MEASURES V.tstamp AS START_T,
              Z.tstamp AS END_T,
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
              COUNT(V.price) AS CNT_V,
              COUNT(W.price) AS UP__W,
              COUNT(X.price) AS DWN_X,
              COUNT(Y.price) AS UP__Y,
              COUNT(Z.price) AS DWN_Z,
    MATCH_NUMBER() AS MNO
    ALL ROWS PER MATCH
    AFTER MATCH SKIP TO LAST Z
    PATTERN (V W{3,4} X{3,4} Y{3,4} Z{3,4})
    DEFINE
       W AS W.price > PREV(W.price),
       X AS X.price < PREV(X.price),
       Y AS Y.price > PREV(Y.price),
       Z AS Z.price < PREV(Z.price)
) MR_EW
ORDER BY symbol, tstamp;

SYMBOL    TSTAMP       START_T      END_T           CNT_V    UP__W    DWN_X    UP__Y
DWN_Z    MNO    PRICE
_________ ____________ ____________ ____________ ________ ________ ________ ________
________ ______ ________
ACME      08-MAY-22    08-MAY-22                       1        0        0
0        0      1    36.66
ACME      09-MAY-22    08-MAY-22                       1        1        0
0        0      1    36.98
ACME      10-MAY-22    08-MAY-22                       1        2        0
0        0      1    37.08
ACME      11-MAY-22    08-MAY-22                       1        3        0
0        0      1    37.43
ACME      12-MAY-22    08-MAY-22                       1        4        0
0        0      1    37.68
ACME      13-MAY-22    08-MAY-22                       1        4        1
0        0      1    37.66
ACME      14-MAY-22    08-MAY-22                       1        4        2
0        0      1    37.32
ACME      15-MAY-22    08-MAY-22                       1        4        3
0        0      1    37.16
ACME      16-MAY-22    08-MAY-22                       1        4        4
0        0      1    36.98
ACME      17-MAY-22    08-MAY-22                       1        4        4
1        0      1    37.19
ACME      18-MAY-22    08-MAY-22                       1        4        4
2        0      1    37.45
ACME      19-MAY-22    08-MAY-22                       1        4        4
3        0      1    37.79
ACME      20-MAY-22    08-MAY-22    20-MAY-22          1        4        4
3        1      1    37.49
ACME      21-MAY-22    08-MAY-22    21-MAY-22          1        4        4
3        2      1     37.3
ACME      22-MAY-22    08-MAY-22    22-MAY-22          1        4        4
3        3      1    37.08
```

### Example 22-14 Skipping into the Middle of a Match to Check for Overlapping Matches

Example 22-14 highlights the power of the `AFTER MATCH SKIP TO` clause to find overlapping matches. It has a simple pattern that seeks a W-shape made up of pattern variables `Q`, `R`, `S`, and `T`. For each leg of the W, the number of rows can be one or more. The match also takes advantage of the `AFTER MATCH SKIP TO` clause: when a match is found, it will skip forward only to the last `R` value, which is the midpoint of the W-shape. This enables the query to find matches in the W-shape where the second half of a W-shape is the first half of a following


Copyright © 2001, 2025, Oracle and/or its affiliates.

overlapped W-shape. In the following output, you can see that match one ends on April 5, but match two overlaps and begins on April 3.

```
SELECT MR_W.*
FROM Ticker3Wave MATCH_RECOGNIZE (
     PARTITION BY symbol
     ORDER BY tstamp
     MEASURES
        MATCH_NUMBER() AS MNO,
        P.tstamp AS START_T,
        T.tstamp AS END_T,
        MAX(P.price) AS TOP_L,
        MIN(Q.price) AS BOTT1,
        MAX(R.price) AS TOP_M,
        MIN(S.price) AS BOTT2,
        MAX(T.price) AS TOP_R
     ALL ROWS PER MATCH
     AFTER MATCH SKIP TO LAST R
     PATTERN ( P Q+ R+ S+ T+ )
     DEFINE
        Q AS Q.price < PREV(Q.price),
        R AS R.price > PREV(R.price),
        S AS S.price < PREV(S.price),
        T AS T.price > PREV(T.price)
) MR_W
ORDER BY symbol, mno, tstamp;

SYMB TSTAMP      MNO START_T   END_T     TOP_L BOTT1 TOP_M BOTT2 TOP_R PRICE
---- --------- ----- --------- --------- ----- ----- ----- ----- ----- -----
ACME 01-Apr-22     1 01-Apr-22            1000                       1000
ACME 02-Apr-22     1 01-Apr-22            1000   775                  775
ACME 03-Apr-22     1 01-Apr-22            1000   775   900            900
ACME 04-Apr-22     1 01-Apr-22            1000   775   900   775      775
ACME 05-Apr-22     1 01-Apr-22 05-Apr-22  1000   775   900   775   900   900
ACME 03-Apr-22     2 03-Apr-22             900                       900
ACME 04-Apr-22     2 03-Apr-22             900   775                 775
ACME 05-Apr-22     2 03-Apr-22             900   775   900           900
ACME 06-Apr-22     2 03-Apr-22             900   775   900   775     775
ACME 07-Apr-22     2 03-Apr-22 07-Apr-22   900   775   900   775   900   900
ACME 05-Apr-22     3 05-Apr-22             900                       900
ACME 06-Apr-22     3 05-Apr-22             900   775                 775
ACME 07-Apr-22     3 05-Apr-22             900   775   900           900
ACME 08-Apr-22     3 05-Apr-22             900   775   900   775     775
ACME 09-Apr-22     3 05-Apr-22 09-Apr-22   900   775   900   775   800   800
ACME 07-Apr-22     4 07-Apr-22             900                       900
ACME 08-Apr-22     4 07-Apr-22             900   775                 775
ACME 09-Apr-22     4 07-Apr-22             900   775   800           800
ACME 10-Apr-22     4 07-Apr-22             900   775   800   550     550
ACME 11-Apr-22     4 07-Apr-22 11-Apr-22   900   775   800   550   900   900
ACME 09-Apr-22     5 09-Apr-22             800                       800
ACME 10-Apr-22     5 09-Apr-22             800   550                 550
ACME 11-Apr-22     5 09-Apr-22             800   550   900           900
ACME 12-Apr-22     5 09-Apr-22             800   550   900   800     800
ACME 13-Apr-22     5 09-Apr-22 13-Apr-22   800   550   900   800  1100  1100
ACME 11-Apr-22     6 11-Apr-22             900                       900
ACME 12-Apr-22     6 11-Apr-22             900   800                 800
ACME 13-Apr-22     6 11-Apr-22             900   800  1100           1100
ACME 14-Apr-22     6 11-Apr-22             900   800  1100   800     800
ACME 15-Apr-22     6 11-Apr-22             900   800  1100   550     550
ACME 16-Apr-22     6 11-Apr-22 16-Apr-22   900   800  1100   550   800   800
ACME 17-Apr-22     6 11-Apr-22 17-Apr-22   900   800  1100   550   875   875
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
ACME 18-Apr-22     6 11-Apr-22 18-Apr-22   900   800  1100   550   950   950
```

**Example 22-15 Find Large Transactions Occurring Within a Specified Time Interval**

In Example 22-15, you find stocks which have heavy trading, that is, large transactions in a concentrated period. In this example, heavy trading is defined as three transactions occurring in a single hour where each transaction was for more than 30,000 shares. Note that it is essential to include a pattern variable such as `B`, so the pattern can accept the trades that do not meet the condition. Without the `B` variable, the pattern would only match cases where there were three consecutive transactions meeting the conditions.

The query in this example uses table `stockT04`.

```
CREATE TABLE STOCKT04 (symbol varchar2(10), tstamp TIMESTAMP,
                       price NUMBER, volume NUMBER);

INSERT INTO STOCKT04 VALUES('ACME', '01-May-22 12.00.00.000000 PM', 35, 35000);
INSERT INTO STOCKT04 VALUES('ACME', '01-May-22 12.05.00.000000 PM', 35, 15000);
INSERT INTO STOCKT04 VALUES('ACME', '01-May-22 12.10.00.000000 PM', 35,  5000);
INSERT INTO STOCKT04 VALUES('ACME', '01-May-22 12.11.00.000000 PM', 35, 42000);
INSERT INTO STOCKT04 VALUES('ACME', '01-May-22 12.16.00.000000 PM', 35,  7000);
INSERT INTO STOCKT04 VALUES('ACME', '01-May-22 12.19.00.000000 PM', 35,  5000);
INSERT INTO STOCKT04 VALUES('ACME', '01-May-22 12.20.00.000000 PM', 35,  5000);
INSERT INTO STOCKT04 VALUES('ACME', '01-May-22 12.33.00.000000 PM', 35, 55000);
INSERT INTO STOCKT04 VALUES('ACME', '01-May-22 12.36.00.000000 PM', 35, 15000);
INSERT INTO STOCKT04 VALUES('ACME', '01-May-22 12.48.00.000000 PM', 35, 15000);
INSERT INTO STOCKT04 VALUES('ACME', '01-May-22 12.59.00.000000 PM', 35, 15000);
INSERT INTO STOCKT04 VALUES('ACME', '01-May-22 01.09.00.000000 PM', 35, 55000);
INSERT INTO STOCKT04 VALUES('ACME', '01-May-22 01.19.00.000000 PM', 35, 55000);
INSERT INTO STOCKT04 VALUES('ACME', '01-May-22 01.29.00.000000 PM', 35, 15000);

SELECT *
FROM stockT04 MATCH_RECOGNIZE (
     PARTITION BY symbol
     ORDER BY tstamp
     MEASURES FIRST (A.tstamp) AS in_hour_of_trade,
              SUM (A.volume) AS sum_of_large_volumes
     ONE ROW PER MATCH
     AFTER MATCH SKIP PAST LAST ROW
     PATTERN (A B* A B* A)
     DEFINE
        A AS ((A.volume > 30000) AND
        ((A.tstamp - FIRST (A.tstamp)) < '0    01:00:00.00' )),
        B AS ((B.volume <= 30000) AND ((B.tstamp - FIRST (A.tstamp)) < '0
        01:00:00.00'))
);

SYMBOL IN_HOUR_OF_TRADE               SUM_OF_LARGE_VOLUMES
------ -----------------------------  --------------------
ACME    01-May-22 12.00.00.000000 PM                132000

1 row selected.
```

## 22.6.2 Pattern Matching Examples: Security Log Analysis

The examples in this section deal with a computer system that issues error messages and authentication checks, and stores the events in a system file. To determine if there are security issues and other problems, you want to analyze the system file. This activity is also referred to as log combing because the software combs through the file to find items of concern. Note that


Copyright © 2001, 2025, Oracle and/or its affiliates.

the source data for these examples is not shown because it would use too much space. In these examples, the AUTHENLOG table comes from the log file.

### Example 22-16 Four or More Consecutive Identical Messages

The query in this example seeks occurrences of four or more consecutive identical messages from a set of three possible 'errtype' values: error, notice, and warn.

```
SELECT MR_SEC.ERRTYPE,
       MR_SEC.MNO     AS Pattern,
       MR_SEC.CNT     AS Count,
       SUBSTR(MR_SEC.MSG_W, 1, 30)   AS Message,
       MR_SEC.START_T AS Starting_on,
       MR_SEC.END_T   AS Ending_on
FROM AUTHENLOG
MATCH_RECOGNIZE(
    PARTITION BY errtype
    ORDER BY tstamp
    MEASURES
        S.tstamp         AS START_T,
        W.tstamp         AS END_T,
        W.message        AS MSG_W,
        COUNT(*)         AS CNT,
        MATCH_NUMBER()   AS MNO
    ONE ROW PER MATCH
    AFTER MATCH SKIP PAST LAST ROW
    PATTERN ( S W{3,} )
    DEFINE  W  AS  W.message = PREV (W.message)
    ) MR_SEC
ORDER BY ErrType, Pattern;
```

```
ERRTYP PATTERN COUNT MESSAGE              STARTING_ON                   ENDING_ON
------ ------- ----- -------------------  ----------------------------- -----------------------------
error        1     4 script not found or  09-JAN-10 12.00.06.000006 PM 09-JAN-10 12.00.15.000015 PM
error        2     4 File does not exist  04-FEB-10 12.00.18.000018 PM 04-FEB-10 12.00.23.000023 PM
error        3     4 File does not exist  06-FEB-10 12.00.25.000025 PM 06-FEB-10 12.00.33.000033 PM
error        4     4 File does not exist  13-FEB-10 12.00.19.000019 PM 14-FEB-10 12.00.07.000007 PM
error        5     5 File does not exist  28-FEB-10 12.00.27.000027 PM 28-FEB-10 12.00.34.000034 PM
error        6     4 script not found or  05-APR-10 12.00.19.000019 PM 05-MAR-10 12.00.23.000023 PM
error        7     4 File does not exist  07-MAR-10 12.00.31.000031 PM 08-MAR-10 12.00.02.000002 PM
error        8     4 File does not exist  14-MAR-10 12.00.19.000019 PM 15-MAR-10 12.00.00.000000 PM
error        9     4 File does not exist  20-MAR-10 12.00.02.000002 PM 20-MAR-10 12.00.06.000006 PM
error       10     5 File does not exist  28-APR-10 12.00.24.000024 PM 28-APR-10 12.00.31.000031 PM
error       11     5 script not found or  01-MAY-10 12.00.15.000015 PM 02-MAY-10 12.00.11.000011 PM
error       12     5 user jsmith: authen  02-MAY-10 12.00.54.000054 PM 03-MAY-10 12.00.11.000011 PM
error       13     4 File does not exist  09-MAY-10 12.00.46.000046 PM 10-MAY-10 12.00.01.000001 PM
error       14     4 File does not exist  20-MAY-10 12.00.42.000042 PM 20-MAY-10 12.00.47.000047 PM
error       15     4 user jsmith: authen  21-MAY-10 12.00.08.000008 PM 21-MAY-10 12.00.18.000018 PM
error       16     4 File does not exist  24-MAY-10 12.00.07.000007 PM 25-MAY-10 12.00.01.000001 PM
error       17     4 user jsmith: authen  12-JUN-10 12.00.00.000000 PM 12-JUN-10 12.00.07.000007 PM
error       18     4 script not found or  12-JUN-10 12.00.18.000018 PM 13-JUN-10 12.00.01.000001 PM
error       19     4 File does not exist  17-JUN-10 12.00.23.000023 PM 17-JUN-10 12.00.30.000030 PM
error       20     5 File does not exist  21-JUN-10 12.00.31.000031 PM 22-JUN-10 12.00.01.000001 PM
error       21     4 user jsmith: authen  22-JUN-10 12.00.36.000036 PM 22-JUN-10 12.00.56.000056 PM
error       22     4 File does not exist  08-JUL-10 12.00.29.000029 PM 08-JUL-10 12.00.32.000032 PM
error       23     6 user jsmith: authen  10-JUL-10 12.00.43.000043 PM 11-JUL-10 12.00.06.000006 PM
error       24     4 File does not exist  12-JUL-10 12.00.09.000009 PM 12-JUL-10 12.00.22.000022 PM
error       25     4 File does not exist  26-JUL-10 12.00.18.000018 PM 27-JUL-10 12.00.04.000004 PM
error       26     4 File does not exist  03-AUG-10 12.00.02.000002 PM 03-AUG-10 12.00.11.000011 PM
error       27     4 File does not exist  23-AUG-10 12.00.04.000004 PM 23-AUG-10 12.00.18.000018 PM
error       28     5 File does not exist  24-AUG-10 12.00.09.000009 PM 26-AUG-10 12.00.00.000000 PM
error       29     4 script not found or  09-SEP-10 12.00.03.000003 PM 09-SEP-10 12.00.09.000009 PM
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
error       30     4 script not found or  11-SEP-10 12.00.22.000022 PM 11-SEP-10 12.00.31.000031 PM
error       31     4 script not found or  23-SEP-10 12.00.09.000009 PM 23-SEP-10 12.00.16.000016 PM
error       32     5 script not found or  17-OCT-10 12.00.02.000002 PM 18-OCT-10 12.00.09.000009 PM
error       33     4 File does not exist  20-OCT-10 12.00.35.000035 PM 21-OCT-10 12.00.00.000000 PM
error       34     5 File does not exist  21-OCT-10 12.00.16.000016 PM 21-OCT-10 12.00.35.000035 PM
error       35     4 File does not exist  26-OCT-10 12.00.25.000025 PM 26-OCT-10 12.00.35.000035 PM
error       36     4 user jsmith: authen  26-OCT-10 12.00.43.000043 PM 26-OCT-10 12.00.49.000049 PM
error       37     4 user jsmith: authen  01-NOV-10 12.00.35.000035 PM 01-NOV-10 12.00.39.000039 PM
error       38     4 File does not exist  09-NOV-10 12.00.46.000046 PM 10-NOV-10 12.00.09.000009 PM
error       39     4 user jsmith: authen  11-NOV-10 12.00.14.000014 PM 11-NOV-10 12.00.30.000030 PM
error       40     4 user jsmith: authen  22-NOV-10 12.00.46.000046 PM 23-NOV-10 12.00.07.000007 PM
error       41     4 script not found or  03-DEC-10 12.00.14.000014 PM 03-DEC-10 12.00.27.000027 PM
error       42     5 File does not exist  07-DEC-10 12.00.02.000002 PM 07-DEC-10 12.00.37.000037 PM
error       43     4 user jsmith: authen  11-DEC-10 12.00.06.000006 PM 11-DEC-10 12.00.11.000011 PM
error       44     4 user jsmith: authen  19-DEC-10 12.00.26.000026 PM 20-DEC-10 12.00.04.000004 PM
error       45     4 user jsmith: authen  25-DEC-10 12.00.11.000011 PM 25-DEC-10 12.00.17.000017 PM
error       46     4 File does not exist  04-JAN-11 12.00.09.000009 PM 04-JAN-11 12.00.19.000019 PM
error       47     4 user jsmith: authen  10-JAN-11 12.00.23.000023 PM 11-JAN-11 12.00.03.000003 PM
error       48     4 File does not exist  11-JAN-11 12.00.14.000014 PM 11-JAN-11 12.00.24.000024 PM
notice       1     4 Child 3228: Release  08-JAN-10 12.00.38.000038 PM 09-JAN-10 12.00.02.000002 PM
notice       2     4 Child 3228: Release  16-JAN-10 12.00.10.000010 PM 17-JAN-10 12.00.13.000013 PM
notice       3     4 Child 1740: Startin  28-JAN-10 12.00.17.000017 PM 28-JAN-10 12.00.22.000022 PM
notice       4     4 Child 1740: Child p  08-MAR-10 12.00.37.000037 PM 08-MAR-10 12.00.40.000040 PM
notice       5     4 Child 3228: All wor  19-APR-10 12.00.10.000010 PM 19-APR-10 12.00.15.000015 PM
notice       6     4 Child 1740: Acquire  02-MAY-10 12.00.38.000038 PM 02-MAY-10 12.00.46.000046 PM
notice       7     4 Child 1740: Starting 09-MAY-10 12.00.03.000003 PM 09-MAY-10 12.00.08.000008 PM
notice       8     4 Child 3228: Child pr 18-MAY-10 12.00.38.000038 PM 18-MAY-10 12.00.45.000045 PM
notice       9     4 Child 3228: All work 25-JUL-10 12.00.04.000004 PM 25-JUL-10 12.00.09.000009 PM
notice      10     4 Child 3228: All work 24-AUG-10 12.00.11.000011 PM 24-AUG-10 12.00.18.000018 PM
notice      11     4 Child 1740: Starting 19-SEP-10 12.00.05.000005 PM 19-SEP-10 12.00.15.000015 PM
notice      12     4 Child 1740: Acquired 06-OCT-10 12.00.07.000007 PM 06-OCT-10 12.00.13.000013 PM
notice      13     4 Child 1740: Starting 09-JAN-11 12.00.12.000012 PM 09-JAN-11 12.00.18.000018 PM
warn         1  3448 The ScriptAlias dire 01-JAN-10 12.00.00.000000 PM 17-JAN-11 12.00.18.000018 PM

62 rows selected.
```

### Example 22-17 Four or More Consecutive Authentication Failures

In this example, you are looking for four or more consecutive authentication failures, regardless of IP origination address. The output shows two matches, the first with five rows and the last one with four rows.

```
SELECT MR_SEC2.ERRTYPE  AS Authen,
       MR_SEC2.MNO      AS Pattern,
       MR_SEC2.CNT      AS Count,
       MR_SEC2.IPADDR   AS On_IP,
       MR_SEC2.TSTAMP   AS Occurring_on
FROM AUTHENLOG
MATCH_RECOGNIZE(
    PARTITION BY errtype
    ORDER BY tstamp
    MEASURES
        COUNT(*)          AS CNT,
        MATCH_NUMBER()    AS MNO
    ALL ROWS PER MATCH
    AFTER MATCH SKIP TO LAST W
    PATTERN ( S W{3,} )
    DEFINE S AS S.message LIKE '%authenticat%',
           W AS W.message = PREV (W.message)
    ) MR_SEC2
ORDER BY Authen, Pattern, Count;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
AUTHEN     PATTERN      COUNT  ON_IP           OCCURRING_ON
------     -------  ---------  ------------    ---------------------------
error            1          1  10.111.112.3    02-MAY-10 12.00.54.000054 PM
error            1          2  10.111.112.6    03-MAY-10 12.00.07.000007 PM
error            1          3  10.111.112.6    03-MAY-10 12.00.08.000008 PM
error            1          4  10.111.112.6    03-MAY-10 12.00.09.000009 PM
error            1          5  10.111.112.6    03-MAY-10 12.00.11.000011 PM
error            2          1  10.111.112.5    21-MAY-10 12.00.08.000008 PM
error            2          2  10.111.112.6    21-MAY-10 12.00.16.000016 PM
error            2          3  10.111.112.4    21-MAY-10 12.00.17.000017 PM
error            2          4  10.111.112.6    21-MAY-10 12.00.18.000018 PM
error            3          1  10.111.112.5    12-JUN-10 12.00.00.000000 PM
error            3          2  10.111.112.4    12-JUN-10 12.00.04.000004 PM
error            3          3  10.111.112.3    12-JUN-10 12.00.06.000006 PM
error            3          4  10.111.112.3    12-JUN-10 12.00.07.000007 PM
error            4          1  10.111.112.5    22-JUN-10 12.00.36.000036 PM
error            4          2  10.111.112.5    22-JUN-10 12.00.50.000050 PM
error            4          3  10.111.112.5    22-JUN-10 12.00.53.000053 PM
error            4          4  10.111.112.6    22-JUN-10 12.00.56.000056 PM
error            5          1  10.111.112.4    10-JUL-10 12.00.43.000043 PM
error            5          2  10.111.112.6    10-JUL-10 12.00.48.000048 PM
error            5          3  10.111.112.6    10-JUL-10 12.00.51.000051 PM
error            5          4  10.111.112.3    11-JUL-10 12.00.00.000000 PM
error            5          5  10.111.112.5    11-JUL-10 12.00.04.000004 PM
error            5          6  10.111.112.3    11-JUL-10 12.00.06.000006 PM
error            6          1  10.111.112.4    26-OCT-10 12.00.43.000043 PM
error            6          2  10.111.112.4    26-OCT-10 12.00.47.000047 PM
error            6          3  10.111.112.4    26-OCT-10 12.00.48.000048 PM
error            6          4  10.111.112.5    26-OCT-10 12.00.49.000049 PM
error            7          1  10.111.112.3    01-NOV-10 12.00.35.000035 PM
error            7          2  10.111.112.5    01-NOV-10 12.00.37.000037 PM
error            7          3  10.111.112.5    01-NOV-10 12.00.38.000038 PM
error            7          4  10.111.112.3    01-NOV-10 12.00.39.000039 PM
error            8          1  10.111.112.6    11-NOV-10 12.00.14.000014 PM
error            8          2  10.111.112.5    11-NOV-10 12.00.20.000020 PM
error            8          3  10.111.112.6    11-NOV-10 12.00.24.000024 PM
error            8          4  10.111.112.3    11-NOV-10 12.00.30.000030 PM
error            9          1  10.111.112.5    22-NOV-10 12.00.46.000046 PM
error            9          2  10.111.112.5    22-NOV-10 12.00.51.000051 PM
error            9          3  10.111.112.3    23-NOV-10 12.00.06.000006 PM
error            9          4  10.111.112.3    23-NOV-10 12.00.07.000007 PM
error           10          1  10.111.112.5    11-DEC-10 12.00.06.000006 PM
error           10          2  10.111.112.4    11-DEC-10 12.00.07.000007 PM
error           10          3  10.111.112.5    11-DEC-10 12.00.08.000008 PM
error           10          4  10.111.112.6    11-DEC-10 12.00.11.000011 PM
error           11          1  10.111.112.5    19-DEC-10 12.00.26.000026 PM
error           11          2  10.111.112.5    20-DEC-10 12.00.01.000001 PM
error           11          3  10.111.112.4    20-DEC-10 12.00.03.000003 PM
error           11          4  10.111.112.3    20-DEC-10 12.00.04.000004 PM
error           12          1  10.111.112.4    25-DEC-10 12.00.11.000011 PM
error           12          2  10.111.112.4    25-DEC-10 12.00.12.000012 PM
error           12          3  10.111.112.4    25-DEC-10 12.00.16.000016 PM
error           12          4  10.111.112.3    25-DEC-10 12.00.17.000017 PM
error           13          1  10.111.112.6    10-JAN-11 12.00.23.000023 PM
error           13          2  10.111.112.6    11-JAN-11 12.00.00.000000 PM
error           13          3  10.111.112.3    11-JAN-11 12.00.02.000002 PM
error           13          4  10.111.112.4    11-JAN-11 12.00.03.000003 PM

55 rows selected.
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Example 22-18 Authentication Failures from the Same IP Address**

The query in Example 22-18 is similar to Example 22-17, but it finds authentication failures from the same IP origination address that occurred three or more consecutive times.

```
SELECT MR_S3.MNO AS Pattern, MR_S3.CNT AS Count,
       MR_S3.ERRTYPE AS Type, MR_S3.IPADDR AS On_IP_addr,
       MR_S3.START_T AS Starting_on, MR_S3.END_T AS Ending_on
FROM AUTHENLOG
MATCH_RECOGNIZE(
    PARTITION BY errtype
    ORDER BY tstamp
    MEASURES
        S.tstamp         AS START_T,
        W.tstamp         AS END_T,
        W.ipaddr         AS IPADDR,
        COUNT(*)         AS CNT,
        MATCH_NUMBER()   AS MNO
    ONE ROW  PER MATCH
    AFTER MATCH SKIP TO LAST W
    PATTERN ( S W{2,} )
    DEFINE  S AS S.message LIKE '%authenticat%',
            W  AS  W.message = PREV (W.message)
                   AND W.ipaddr = PREV (W.ipaddr)
    ) MR_S3
ORDER BY Type, Pattern;
```

```
PATTERN COUNT  TYPE    ON_IP_ADDR     STARTING_ON                    ENDING_ON
------- -----  -----   ------------   ----------------------------   ----------------------------
      1     4  error   10.111.112.6   03-MAY-10 12.00.07.000007 PM   03-MAY-10 12.00.11.000011 PM
      2     3  error   10.111.112.5   22-JUN-10 12.00.36.000036 PM   22-JUN-10 12.00.53.000053 PM
      3     3  error   10.111.112.4   27-JUN-10 12.00.03.000003 PM   27-JUN-10 12.00.08.000008 PM
      4     3  error   10.111.112.6   19-JUL-10 12.00.15.000015 PM   19-JUL-10 12.00.17.000017 PM
      5     3  error   10.111.112.4   26-OCT-10 12.00.43.000043 PM   26-OCT-10 12.00.48.000048 PM
      6     3  error   10.111.112.4   25-DEC-10 12.00.11.000011 PM   25-DEC-10 12.00.16.000016 PM
      7     3  error   10.111.112.5   12-JAN-11 12.00.01.000001 PM   12-JAN-11 12.00.08.000008 PM

7 rows selected.
```

## 22.6.3 Pattern Matching Examples: Sessionization

Sessionization is the process of defining distinct sessions of user activity, typically involving multiple events in a single session. Pattern matching makes it easy to express queries for sessionization. For instance, you may want to know how many pages visitors to your website view during a typical session. If you are a communications provider, you may want to know the characteristics of phone sessions between two users where the sessions involve dropped connections and users redialing. Enterprises can derive significant value from understanding their user session behavior, because it can help firms define service offerings and enhancements, pricing, marketing and more.

The following examples include two introductory examples of sessionization related to web site clickstreams followed by an example involving phone calls.

**Example 22-19 Simple Sessionization for Clickstream Data**

Example 22-19 is a simple illustration of sessionization for clickstream data analysis. For a set of rows, the goal is to detect the sessions, assign a session ID to each session, and to display each input row with its session ID. The data below would come from a web server system log that tracks all page requests. You start with a set of rows where each row is the event of a user requesting a page. In this simple example, the data includes a partition key, which is the user ID, and a timestamp indicating when the user requested a page. Web system logs show when


Copyright © 2001, 2025, Oracle and/or its affiliates.

a user requested a given page, but there is no indication of when the user stopped looking at the page.

In Example 22-19, a session is defined as a sequence of one or more time-ordered rows with the same partition key (`User_ID`) where the time gap between timestamps is less than a specified threshold. In this case, the threshold is ten time units. If rows have a timestamp greater than ten units apart, they are considered to be in different sessions. Note that the 10-unit threshold used here is an arbitrary number: each real-world case requires the analyst's judgment to determine the most suitable threshold time gap. Historically, a 30-minute gap has been a commonly used threshold for separating sessions of website visits.

Start by creating a table of clickstream events.

```
CREATE TABLE Events(
   Time_Stamp NUMBER,
   User_ID VARCHAR2(10)
  );
```

Next insert the data. The insert statements below have been ordered and spaced for your reading convenience so that you can see the partitions and the sessions within them. In real life, the events would arrive in timestamp order and the rows for different user sessions would be intermingled.

```
  INSERT INTO Events(Time_Stamp, User_ID) VALUES ( 1, 'Mary');
  INSERT INTO Events(Time_Stamp, User_ID) VALUES (11, 'Mary');

  INSERT INTO Events(Time_Stamp, User_ID) VALUES (23, 'Mary');

  INSERT INTO Events(Time_Stamp, User_ID) VALUES (34, 'Mary');
  INSERT INTO Events(Time_Stamp, User_ID) VALUES (44, 'Mary');
  INSERT INTO Events(Time_Stamp, User_ID) VALUES (53, 'Mary');
  INSERT INTO Events(Time_Stamp, User_ID) VALUES (63, 'Mary');

  INSERT INTO Events(Time_Stamp, User_ID) VALUES ( 3, 'Richard');
  INSERT INTO Events(Time_Stamp, User_ID) VALUES (13, 'Richard');
  INSERT INTO Events(Time_Stamp, User_ID) VALUES (23, 'Richard');
  INSERT INTO Events(Time_Stamp, User_ID) VALUES (33, 'Richard');
  INSERT INTO Events(Time_Stamp, User_ID) VALUES (43, 'Richard');

  INSERT INTO Events(Time_Stamp, User_ID) VALUES (54, 'Richard');
  INSERT INTO Events(Time_Stamp, User_ID) VALUES (63, 'Richard');

  INSERT INTO Events(Time_Stamp, User_ID) VALUES ( 2, 'Sam');
  INSERT INTO Events(Time_Stamp, User_ID) VALUES (12, 'Sam');
  INSERT INTO Events(Time_Stamp, User_ID) VALUES (22, 'Sam');
  INSERT INTO Events(Time_Stamp, User_ID) VALUES (32, 'Sam');

  INSERT INTO Events(Time_Stamp, User_ID) VALUES (43, 'Sam');
  INSERT INTO Events(Time_Stamp, User_ID) VALUES (47, 'Sam');
  INSERT INTO Events(Time_Stamp, User_ID) VALUES (48, 'Sam');

  INSERT INTO Events(Time_Stamp, User_ID) VALUES (59, 'Sam');
  INSERT INTO Events(Time_Stamp, User_ID) VALUES (60, 'Sam');
  INSERT INTO Events(Time_Stamp, User_ID) VALUES (68, 'Sam');
```

The row pattern matching query below will display each input row with its `Session_ID`. As noted above, events are considered to be part of the same session if they are ten or fewer time units apart. That session threshold is expressed in the `DEFINE` clause for pattern variables.

```
SELECT time_stamp, user_id, session_id
FROM Events MATCH_RECOGNIZE
         (PARTITION BY User_ID ORDER BY Time_Stamp
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
          MEASURES match_number() AS session_id
          ALL ROWS PER MATCH
          PATTERN (b s*)
          DEFINE
             s AS (s.Time_Stamp - prev(Time_Stamp) <= 10)
         )
ORDER BY user_id, time_stamp;
```

The output will be:

```
TIME_STAMP USER_ID    SESSION_ID
---------- ---------- ----------
         1 Mary                1
        11 Mary                1
        23 Mary                2
        34 Mary                3
        44 Mary                3
        53 Mary                3
        63 Mary                3
         3 Richard             1
        13 Richard             1
        23 Richard             1
        33 Richard             1
        43 Richard             1
        54 Richard             2
        63 Richard             2
         2 Sam                 1
        12 Sam                 1
        22 Sam                 1
        32 Sam                 1
        43 Sam                 2
        47 Sam                 2
        48 Sam                 2
        59 Sam                 3
        60 Sam                 3
        68 Sam                 3

24 rows selected.
```

### Example 22-20 Simple Sessionization with Aggregation

Assigning session numbers to detail-level rows as in example Example 22-19 just begins the analytic process. The business value of sessionized data emerges only after aggregating by session.

This example aggregates the data to give one row per session with these columns: `Session_ID`, `User_ID`, number of aggregated events per session, and total session duration. This output makes it easy to see how many clicks each user has per session and how long each session lasts. In turn, data from this query could be used to drive many other analyses such as maximum, minimum, and average session duration.

```
SELECT session_id, user_id, start_time, no_of_events, duration
FROM Events MATCH_RECOGNIZE
     (PARTITION BY User_ID
      ORDER BY Time_Stamp
      MEASURES MATCH_NUMBER() session_id,
               COUNT(*) AS no_of_events,
               FIRST(time_stamp) start_time,
               LAST(time_stamp) - FIRST(time_stamp) duration
      PATTERN (b s*)
      DEFINE
         s AS (s.Time_Stamp - PREV(Time_Stamp) <= 10)
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
    )
ORDER BY user_id, session_id;
```

The output will be:

```
SESSION_ID USER_ID    START_TIME NO_OF_EVENTS   DURATION
---------- ---------- ---------- ------------ ----------
         1 Mary                1            2         10
         2 Mary               23            1          0
         3 Mary               34            4         29
         1 Richard             3            5         40
         2 Richard            54            2          9
         1 Sam                 2            4         30
         2 Sam                43            3          5
         3 Sam                59            3          9

8 rows selected.
```

### Example 22-21 Sessionization for Phone Calls with Dropped Connections

In the examples Example 22-19 and Example 22-20 with clickstream data, there was no explicit end point in the source data to indicate the end time for viewing a page. Even if there are clear end points for user activity, an end point may not indicate that a user wanted to end the session. Consider a person using a mobile phone service whose phone connection is dropped: typically, the user will redial and continue the phone call. In that scenario, multiple phone calls involving the same pair of phone numbers should be considered part of a single phone session.

Example 22-21 illustrates phone call sessionization. It uses call detail record data as the base for sessionization, where the call data record rows include `Start_Time`, `End_Time`, `Caller_ID`, `Callee_ID`. The query below does the following:

- Partitions the data by `caller_id` and `callee_id`.
- Finds sessions where calls from a caller to a callee are grouped into a session if the gap between subsequent calls is within a threshold of 60 seconds. That threshold is specified in the `DEFINE` clause for pattern variable `B`.
- Returns for each session (see the `MEASURES` clause):
  - The `session_id`, the caller and callee
  - How many times calls were restarted in a session
  - Total effective call duration (total time during the session when the phones were connected)
  - Total interrupted duration (total time during the session when the phones were disconnected

```
SELECT  Caller, Callee, Start_Time, Effective_Call_Duration,
                (End_Time - Start_Time) - Effective_Call_Duration
                     AS Total_Interruption_Duration, No_Of_Restarts, Session_ID
FROM my_cdr MATCH_RECOGNIZE
        ( PARTITION BY Caller, Callee ORDER BY Start_Time
           MEASURES
                A.Start_Time               AS Start_Time,
                End_Time                   AS End_Time,
                SUM(End_Time - Start_Time) AS Effective_Call_Duration,
                COUNT(B.*)                 AS No_Of_Restarts,
                MATCH_NUMBER()             AS Session_ID
           PATTERN (A B*)
           DEFINE B AS B.Start_Time - PREV(B.end_Time) < 60
         );
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

Because the previous query needs a significant amount of data to be meaningful, and that would consume substantial space, no INSERT statement is included here. However, the following is sample output.

```
SQL> desc my_cdr
Name            Null?        Type
--------------  ----------   ----------
CALLER          NOT NULL     NUMBER(38)
CALLEE          NOT NULL     NUMBER(38)
START_TIME      NOT NULL     NUMBER(38)
END_TIME        NOT NULL     NUMBER(38)

SELECT * FROM my_cdr ORDER BY 1, 2, 3, 4;

CALLER   CALLEE  START_TIME   END_TIME
------   ------  ----------   ---------
     1        7        1354       1575
     1        7        1603       1829
     1        7        1857       2301
     1        7        2320       2819
     1        7        2840       2964
     1        7       64342      64457
     1        7       85753      85790
     1        7       85808      85985
     1        7       86011      86412
     1        7       86437      86546
     1        7      163436     163505
     1        7      163534     163967
     1        7      163982     164454
     1        7      214677     214764
     1        7      214782     215248
     1        7      216056     216271
     1        7      216297     216728
     1        7      216747     216853
     1        7      261138     261463
     1        7      261493     261864
     1        7      261890     262098
     1        7      262115     262655
     1        7      301931     302226
     1        7      302248     302779
     1        7      302804     302992
     1        7      303015     303258
     1        7      303283     303337
     1        7      383019     383378
     1        7      383407     383534
     1        7      424800     425096

30 rows selected.

CALLER CALLEE START_TIME EFFECTIVE_CALL TOTAL_INTERUPTION NO_OF_RE SESSION_ID
------ ------- --------- -------------- ----------------- -------- ----------
    1        7      1354           1514                96        4          1
    1        7     64342            115                 0        0          2
    1        7     85753            724                69        3          3
    1        7    163436            974                44        2          4
    1        7    214677            553                18        1          5
    1        7    216056            752                45        2          6
    1        7    261138           1444                73        3          7
    1        7    301931           1311                95        4          8
    1        7    383019            486                29        1          9
    1        7    424800            296                 0        0         10
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
10 rows selected.
```

## 22.6.4 Pattern Matching Example: Financial Tracking

A common financial application is to search for suspicious financial patterns. Example 22-22 illustrates how to detect money transfers that seem suspicious because certain criteria you have defined as being unusual have been met.

**Example 22-22 Suspicious Money Transfer**

In Example 22-22, we search for a pattern that seems suspicious when transferring funds. In this case, that is defined as three or more small (less than $2000) money transfers within 30 days followed by a large transfer (over $1,000,000) within 10 days of the last small transfer. To simplify things, the table and data are made very basic.

First, we create a table that contains the necessary data:

```
CREATE TABLE event_log
     ( time          DATE,
       userid        VARCHAR2(30),
       amount        NUMBER(10),
       event         VARCHAR2(10),
       transfer_to   VARCHAR2(10));
```

Then we insert data into `event_log`:

```
INSERT INTO event_log VALUES
   (TO_DATE('01-JAN-2021', 'DD-MON-YYYY'), 'john', 1000000, 'deposit', NULL);
INSERT INTO event_log VALUES
   (TO_DATE('05-JAN-2021', 'DD-MON-YYYY'), 'john', 1200000, 'deposit', NULL);
INSERT INTO event_log VALUES
   (TO_DATE('06-JAN-2021', 'DD-MON-YYYY'), 'john', 1000, 'transfer', 'bob');
INSERT INTO event_log VALUES
   (TO_DATE('15-JAN-2021', 'DD-MON-YYYY'), 'john', 1500, 'transfer', 'bob');
INSERT INTO event_log VALUES
   (TO_DATE('20-JAN-2021', 'DD-MON-YYYY'), 'john', 1500, 'transfer', 'allen');
INSERT INTO event_log VALUES
   (TO_DATE('23-JAN-2021', 'DD-MON-YYYY'), 'john', 1000, 'transfer', 'tim');
INSERT INTO event_log VALUES
   (TO_DATE('26-JAN-2021', 'DD-MON-YYYY'), 'john', 1000000, 'transfer', 'tim');
INSERT INTO event_log VALUES
   (TO_DATE('27-JAN-2021', 'DD-MON-YYYY'), 'john', 500000, 'deposit', NULL);
```

Next, we can query this table:

```
SELECT userid, first_t, last_t, amount
FROM (SELECT * FROM event_log WHERE event = 'transfer')
MATCH_RECOGNIZE
   (PARTITION BY userid ORDER BY time
    MEASURES FIRST(x.time) first_t, y.time last_t, y.amount amount
    PATTERN ( x{3,} y )
    DEFINE x AS (event='transfer' AND amount < 2000),
           y AS (event='transfer' AND amount >= 1000000 AND
                 LAST(x.time) - FIRST(x.time) < 30  AND
                 y.time - LAST(x.time) < 10));

USERID        FIRST_T      LAST_T       AMOUNT
----------    ---------    ---------    -------
john          06-JAN-21    26-JAN-21    1000000
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

In this statement, the first text in bold represents the small transfers, the second represents a large transfer, the third that the small transfers occurred within 30 days, and the fourth that the large transfer occurred within 10 days of the last small transfer.

This statement can be further refined to include the recipient of the suspicious transfer, as in the following:

```
SELECT userid, first_t, last_t, amount, transfer_to
FROM (SELECT * FROM event_log WHERE event = 'transfer')
MATCH_RECOGNIZE
   (PARTITION BY userid ORDER BY time
    MEASURES z.time first_t, y.time last_t, y.amount amount,
             y.transfer_to transfer_to
    PATTERN ( z x{2,} y )
    DEFINE z AS (event='transfer' AND amount < 2000),
           x AS (event='transfer' AND amount <= 2000 AND
                 PREV(x.transfer_to) <> x.transfer_to),
           y AS (event='transfer' AND amount >= 1000000 AND
                 LAST(x.time) - z.time < 30 AND
                 y.time - LAST(x.time) < 10 AND
                 SUM(x.amount) + z.amount < 20000));

USERID        FIRST_T      LAST_T       AMOUNT      TRANSFER_TO
----------    ---------    ---------    -------     -----------
john          15-JAN-21    26-JAN-21    1000000     tim
```

In this statement, the first text in bold represents the first small transfer, the next represents two or more small transfers to different accounts, the third represents the sum of all small transfers less than $20,000.

# 22.7 Fuzzy String Matching

Approximate or "fuzzy" string matching methods like PHONIC_ENCODE and FUZZY_MATCH are an important component of data quality capabilities.

Oracle AI Database 26ai introduces two new operators for fuzzy string matching. `PHONIC_ENCODE` converts words or phrases in language-specific codes based on pronunciation. `FUZZY_MATCH`, which is language-neutral, gauges the textual similarity between two strings. By default, the resulting fuzzy match score is usually normalized to be a percentage of the lengths of the strings being compared.

These flexible and intelligent operators can often find valid matches in spite of typographical errors, spelling/phonetic variations, alternative names, initials in place of complete names, and other irregularities that preclude discovery via methods that use exact string matching or regular expressions. Because these operations can be run directly on the database, runtime is much more efficient than in EDO operations where the data must first be moved outside the database for processing.

Both operators are re-implementations of well-known public data quality string match algorithms.

### PHONIC_ENCODE

This operator mainly implements the Double Metaphone algorithm. If the keyword `DOUBLE_METAPHONE` is specified, it returns the main code. If `DOUBLE_METAPHONE_ALT` is specified, it returns the alternative code.

```
 SQL> select phonic_encode(DOUBLE_METAPHONE,     'smith') c1,
  2  phonic_encode(DOUBLE_METAPHONE_ALT, 'smith') c2 from dual;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
C1            C2
------------- -------------
SM0           XMT

1 row selected.
```

### FUZZY_MATCH

This operator supports the following keywords, each representing the corresponding supported algorithm: `LEVENSHTEIN`, `JARO_WINKLER`, `BIGRAM and TRIGRAM`, `WHOLE_WORD_MATCH`, and `LONGEST_COMMON_SUBSTRING`.

```
SQL> select fuzzy_match(LEVENSHTEIN, 'Mohamed Tarik', 'Mo Tariq') from dual;

FUZZY_MATCH(LEVENSHTEIN,'MOHAMEDTARIK','MOTARIQ')
-------------------------------------------------
                                               54
1 row selected.
```

> **See Also**
>
> Data Quality Operators Data Quality Operators in the *SQL Language Reference* provides complete details and more examples for use of `PHONIC_ENCODE` and `FUZZY_MATCH`.


Copyright © 2001, 2025, Oracle and/or its affiliates.

# 23
# SQL for Modeling

This chapter discusses using SQL modeling, and includes:


- Overview of SQL Modeling in Data Warehouses
- Basic Topics in SQL Modeling
- Advanced Topics in SQL Modeling
- Performance Considerations with SQL Modeling
- Examples of SQL Modeling


## 23.1 Overview of SQL Modeling in Data Warehouses

The `MODEL` clause brings a new level of power and flexibility to SQL calculations. With the `MODEL` clause, you can create a multidimensional array from query results and then apply formulas (called rules) to this array to calculate new values. The rules can range from basic arithmetic to simultaneous equations using recursion. For some applications, the `MODEL` clause can replace PC-based spreadsheets. Models in SQL leverage Oracle AI Database's strengths in scalability, manageability, collaboration, and security. The core query engine can work with unlimited quantities of data. By defining and executing models within the database, users avoid transferring large data sets to and from separate modeling environments. Models can be shared easily across workgroups, ensuring that calculations are consistent for all applications. Just as models can be shared, access can also be controlled precisely with Oracle's security features. With its rich functionality, the `MODEL` clause can enhance all types of applications.

The `MODEL` clause enables you to create a multidimensional array by mapping the columns of a query into three groups: partitioning, dimension, and measure columns. These elements perform the following tasks:

- Partition columns define the logical blocks of the result set in a way similar to the partitions of the analytical functions described in SQL for Analysis and Reporting. Rules in the `MODEL` clause are applied to each partition independent of other partitions. Thus, partitions serve as a boundary point for parallelizing the `MODEL` computation.
- Dimension columns define the multi-dimensional array and are used to identify cells within a partition. By default, a full combination of dimensions should identify just one cell in a partition. In default mode, they can be considered analogous to the key of a relational table.
- Measures are equivalent to the measures of a fact table in a star schema. They typically contain numeric values such as sales units or cost. Each cell is accessed by specifying its full combination of dimensions. Note that each partition may have a cell that matches a given combination of dimensions.

The `MODEL` clause enables you to specify rules to manipulate the measure values of the cells in the multi-dimensional array defined by partition and dimension columns. Rules access and update measure column values by directly specifying dimension values. The references used in rules result in a highly readable model. Rules are concise and flexible, and can use wild cards and looping constructs for maximum expressiveness. Oracle AI Database evaluates the rules in an efficient way, parallelizes the model computation whenever possible, and provides a

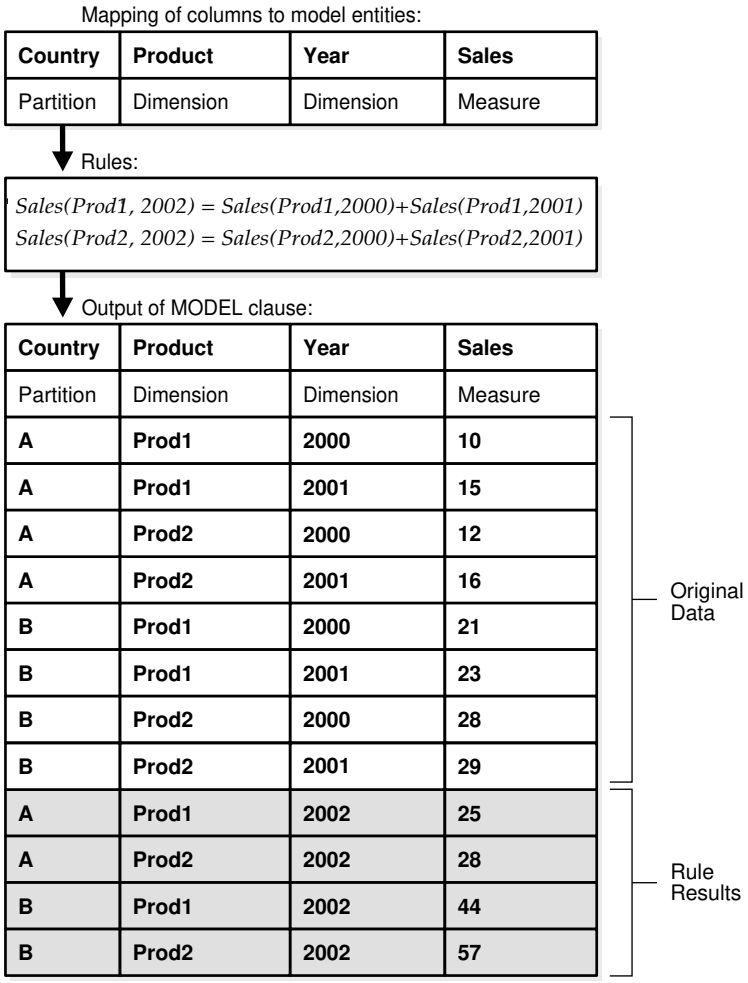


Copyright © 2001, 2025, Oracle and/or its affiliates.

seamless integration of the `MODEL` clause with other SQL clauses. The `MODEL` clause, thus, is a scalable and manageable way of computing business models in the database.

Figure 23-1 offers a conceptual overview of the modeling feature of SQL. The figure has three parts. The top segment shows the concept of dividing a typical table into partition, dimension, and measure columns. The middle segment shows two rules that calculate the value of `Prod1` and `Prod2` for the year 2002. Finally, the third part shows the output of a query that applies the rules to such a table with hypothetical data. The unshaded output is the original data as it is retrieved from the database, while the shaded output shows the rows calculated by the rules. Note that results in partition `A` are calculated independently from results of partition `B`.

**Figure 23-1 Model Elements**




This section contains the following topics:

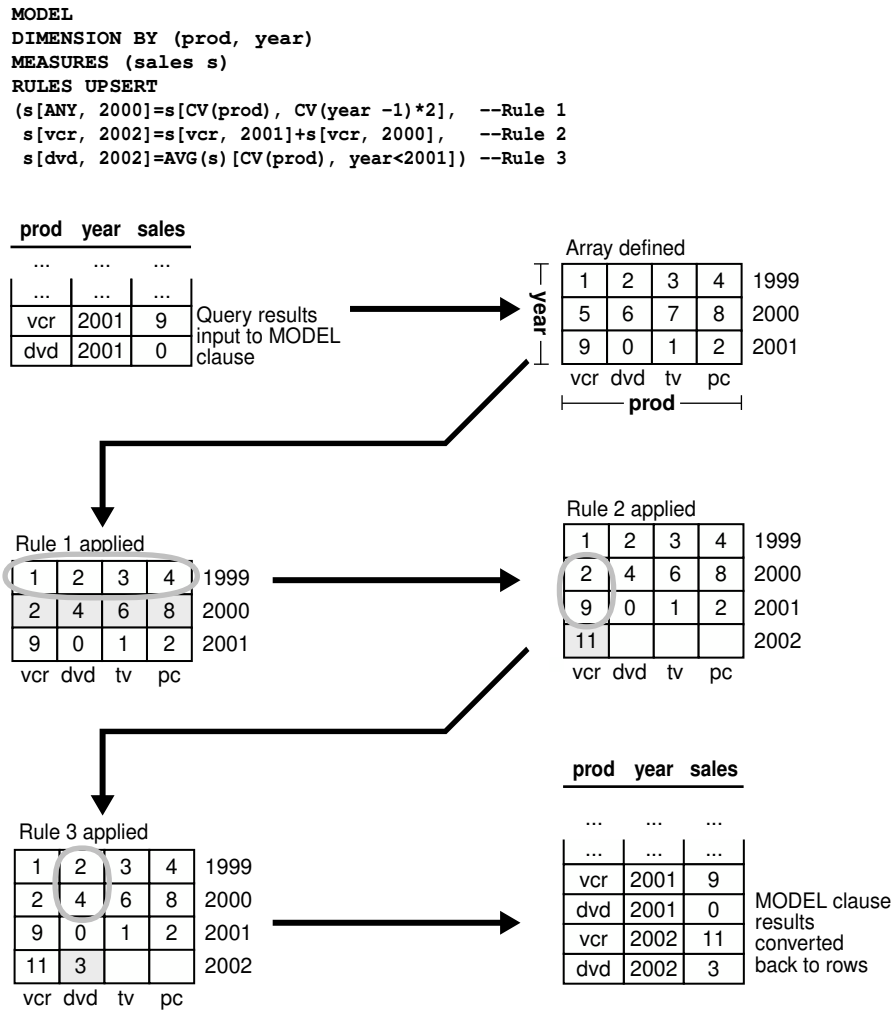
- How Data is Processed in a SQL Model
- Why Use SQL Modeling in Data Warehouses?
- About SQL Modeling Capabilities


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 23.1.1 How Data is Processed in a SQL Model

Figure 23-2 shows the flow of processing within a simple `MODEL` clause. In this case, you will follow data through a `MODEL` clause that includes three rules. One of the rules updates an existing value, while the other two create new values for a forecast. The figure shows that the rows of data retrieved by a query are fed into the `MODEL` clause and rearranged into an array. Once the array is defined, rules are applied one by one to the data. The shaded cells in Figure 23-2 represent new data created by the rules and the cells enclosed by ovals represent the source data for the new values. Finally, the data, including both its updated values and newly created values, is rearranged into row form and presented as the results of the query. Note that no data is inserted into any table by this query.

**Figure 23-2 Model Flow Processing**




## 23.1.2 Why Use SQL Modeling in Data Warehouses?

Oracle modeling enables you to perform sophisticated calculations on your data. A typical case is when you want to apply business rules to data and then generate reports. Because Oracle AI Database integrates modeling calculations into the database, performance and


Copyright © 2001, 2025, Oracle and/or its affiliates.

manageability are enhanced significantly. Consider the following query. The view sales_view has been created in previous sh sample schema examples.

```
SELECT SUBSTR(country, 1, 20) country,
      SUBSTR(product, 1, 15) product, year, sales
FROM sales_view
WHERE country IN ('Italy', 'Japan')
MODEL
  PARTITION BY (country) DIMENSION BY (product, year)
  MEASURES (sales sales)
  RULES
  (sales['Catchers Mitt', 2022] = sales['Catchers Mitt', 2021] + sales['Catchers Mitt',
2020],
   sales['Practice Tennis Balls', 2022] = sales['Practice Tennis Balls', 2021],
   sales['5 Point Batting Tee', 2022] = sales['Catchers Mitt', 2022] + sales['Practice
Tennis Balls', 2022])
ORDER BY country, product, year;
```

This query partitions the data in `sales_view` (which is illustrated in "Base Schema for SQL Modeling Examples") on country so that the model computation, as defined by the three rules, is performed on each country. This model calculates the sales of Bounce in 2022 as the sum of its sales in 2020 and 2021, and sets the sales for Y Box in 2022 to the same value as they were in 2021. Also, it introduces a new product category All_Products (`sales_view` does not have the product All_Products) for year 2022 to be the sum of sales of Bounce and Y Box for that year. The output of this query is as follows, where bold text indicates new values:

```
COUNTRY    PRODUCT               YEAR       SALES
__________ ____________________ _______ ___________
Italy      11" Youth Field       2019     2070.22
Italy      11" Youth Field       2020     4360.09
Italy      11" Youth Field       2021        1276
Italy      11" Youth Field       2022     3186.55
Italy      11.5" Youth Tri       2019     4896.79
...
Japan      6 Gallon Empty        2019     5461.59
Japan      6 Gallon Empty        2020      5897.8
Japan      6 Gallon Empty        2021     2995.93
Japan      6 Gallon Empty        2022     8935.03
Japan      Baseball Is Lif       2019     7510.02
Japan      Baseball Is Lif       2020    19291.97
Japan      Baseball Is Lif       2021    18087.06
Japan      Baseball Is Lif       2022    24092.75
...
```

See that while the sales values for and Y Box exist in the input, the values for All_Products are derived.


Copyright © 2001, 2025, Oracle and/or its affiliates.

> **Note**
>
> If `sales_view` does not exist, you can create it as follows.

```
CREATE VIEW sales_view AS
SELECT
 prod_name product, country_name country, channel_id channel,
 SUBSTR(calendar_quarter_desc, 6,2) quarter,
 SUM(amount_sold) amount_sold, SUM(quantity_sold) quantity_sold
FROM sh.sales, sh.times, sh.customers, sh.countries, sh.products
WHERE sales.time_id = times.time_id AND
  sales.prod_id = products.prod_id AND
  sales.cust_id = customers.cust_id AND
  customers.country_id = countries.country_id
GROUP BY prod_name, country_name, channel_id,
 SUBSTR(calendar_quarter_desc, 6, 2);
```

## 23.1.3 About SQL Modeling Capabilities

Oracle AI Database provides the following capabilities with the `MODEL` clause:

- Cell addressing using dimension values

  Measure columns in individual rows are treated like cells in a multi-dimensional array and can be referenced and updated using dimension values. For example, in a fact table `ft(country, year, sales)`, you can designate `country` and `year` to be dimension columns and `sales` to be the measure and reference sales for a given country and year as `sales[country='Spain', year=2019]`. This gives you the sales value for Spain in 2019. You can also use a shorthand form `sales['Spain', 2019]`, which has the same meaning. There are a few semantic differences between these notations, though. See "About Cell Referencing in SQL Modeling" for further details.

- Symbolic array computation

  You can specify a series of formulas, called rules, to operate on the data. Rules can invoke functions on individual cells or on a set or range of cells. An example involving individual cells is the following:

  ```
  sales[country='Spain',year=2021] = sales['Spain',2020]+ sales['Spain',2019]
  ```

  This sets the sales in Spain for the year 2021 to the sum of sales in Spain for 2019 and 2020. An example involving a range of cells is the following:

  ```
  sales[country='Spain',year=2021] =
     MAX(sales)['Spain',year BETWEEN 1997 AND 2020]
  ```

  This sets the sales in Spain for the year 2021 equal to the maximum sales in Spain between 1997 and 2020.

- `UPSERT`, `UPSERT ALL`, and `UPDATE` options

  Using the `UPSERT` option, which is the default, you can create cell values that do not exist in the input data. If the cell referenced exists in the data, it is updated. If the cell referenced does not exist in the data, and the rule uses appropriate notation, then the cell is inserted. The `UPSERT ALL` option enables you to have `UPSERT` behavior for a wider variety of rules. The `UPDATE` option, on the other hand, would never insert any new cells.


Copyright © 2001, 2025, Oracle and/or its affiliates.

You can specify these options globally, in which case they apply to all rules, or per each rule. If you specify an option at the rule level, it overrides the global option. Consider the following rules:

```
UPDATE sales['Spain', 2019] = 3567.99,
UPSERT sales['Spain', 2021] = sales['Spain', 2020]+ sales['Spain', 2019]
```

The first rule updates the cell for sales in Spain for 2019. The second rule updates the cell for sales in Spain for 2021 if it exists, otherwise, it creates a new cell.

- Wildcard specification of dimensions

  You can use `ANY` and `IS ANY` to specify all values in a dimension. As an example, consider the following statement:

  ```
  sales[ANY, 2021] = sales['Japan', 2020]
  ```

  This rule sets the 2021 sales of all countries equal to the sales value of Japan for the year 2020. All values for the dimension, including nulls, satisfy the `ANY` specification. You can also specify this using an `IS ANY` predicate as in the following:

  ```
  sales[country IS ANY, 2021] = sales['Japan', 2020]
  ```

- Accessing dimension values using the `CV` function

  You can use the `CV` function on the right side of a rule to access the value of a dimension column of the cell referenced on the left side of a rule. It enables you to combine multiple rules performing similar computation into a single rule, thus resulting in concise specification. For example, you can combine the following rules:

  ```
  sales[country='Spain', year=2022] = 1.2 * sales['Spain', 2021],
  sales[country='Italy', year=2022] = 1.2 * sales['Italy', 2021],
  sales[country='Japan', year=2022] = 1.2 * sales['Japan', 2021]
  ```

  They can be combined into one single rule:

  ```
  sales[country IN ('Spain', 'Italy', 'Japan'), year=2022] = 1.2 *
     sales[CV(country), 2021]
  ```

  Observe that the `CV` function passes the value for the `country` dimension from the left to the right side of the rule.

- Ordered computation

  For rules updating a set of cells, the result may depend on the ordering of dimension values. You can force a particular order for the dimension values by specifying an `ORDER BY` in the rule. An example is the following rule:

  ```
  sales[country IS ANY, year BETWEEN 2020 AND 2003] ORDER BY year =
    1.05 * sales[CV(country), CV(year)-1]
  ```

  This ensures that the years are referenced in increasing chronological order.

- Automatic rule ordering

  Rules in the `MODEL` clause can be automatically ordered based on dependencies among the cells using the `AUTOMATIC ORDER` keywords. For example, in the following assignments, the last two rules will be processed before the first rule because the first depends on the second and third:

  ```
  RULES AUTOMATIC ORDER
  {sales[c='Spain', y=2021] = sales[c='Spain', y=2020]
    + sales[c='Spain', y=2019]
  sales[c='Spain', y=2020] = 50000,
  sales[c='Spain', y=2019] = 40000}
  ```


Copyright © 2001, 2025, Oracle and/or its affiliates.

- Iterative rule evaluation

  You can specify iterative rule evaluation, in which case the rules are evaluated iteratively until the termination condition is satisfied. Consider the following specification:

  ```
  MODEL DIMENSION BY (x) MEASURES (s)
    RULES ITERATE (4) (s[x=1] = s[x=1]/2)
  ```

  This statement specifies that the formula `s[x=1] = s[x=1]/2` evaluation be repeated four times. The number of iterations is specified in the `ITERATE` option of the `MODEL` clause. It is also possible to specify a termination condition by using an `UNTIL` clause.

  Iterative rule evaluation is an important tool for modeling recursive relationships between entities in a business application. For example, a loan amount might depend on the interest rate where the interest rate in turn depends on the amount of the loan.

- Reference models

  A model can include multiple ref models, which are read-only arrays. Rules can reference cells from different reference models. Rules can update or insert cells in only one multi-dimensional array, which is called the main model. The use of reference models enables you to relate models with different dimensionality. For example, assume that, in addition to the fact table `ft(country, year, sales)`, you have a table with currency conversion ratios `cr(country, ratio)` with `country` as the dimension column and `ratio` as the measure. Each row in this table gives the conversion ratio of that country's currency to that of US dollar. These two tables could be used in rules such as the following:

  ```
  dollar_sales['Spain',2021] = sales['Spain',2020] * ratio['Spain']
  ```

- Scalable computation

  You can partition data and evaluate rules within each partition independent of other partitions. This enables parallelization of model computation based on partitions. For example, consider the following model:

  ```
  MODEL PARTITION BY (country) DIMENSION BY (year) MEASURES (sales)
    (sales[year=2021] = AVG(sales)[year BETWEEN 1990 AND 2020]
  ```

  The data is partitioned by country and, within each partition, you can compute the sales in 2021 to be the average of sales in the years between 1990 and 2020. Partitions can be processed in parallel and this results in a scalable execution of the model.

# 23.2 Basic Topics in SQL Modeling

This section introduces some of the basic ideas and uses for models, and includes:

- Base Schema for SQL Modeling Examples
- MODEL Clause Syntax
- Keywords in SQL Modeling
- About Cell Referencing in SQL Modeling
- About Rules for SQL Modeling
- Order of Evaluation of SQL Modeling Rules
- Global and Local Keywords for SQL Modeling Rules
- UPDATE, UPSERT, and UPSERT ALL Behavior
- Treatment of NULLs and Missing Cells in SQL Modeling
- About Reference Models in SQL Modeling


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 23.2.1 Base Schema for SQL Modeling Examples

This chapter's examples are based on the following view `sales_view`, which is derived from the `sh` sample schema.

```
CREATE VIEW sales_view AS
SELECT country_name country, prod_name product, calendar_year year,
  SUM(amount_sold) sales, COUNT(amount_sold) cnt,
  MAX(calendar_year) KEEP (DENSE_RANK FIRST ORDER BY SUM(amount_sold) DESC)
  OVER (PARTITION BY country_name, prod_name) best_year,
  MAX(calendar_year) KEEP (DENSE_RANK LAST ORDER BY SUM(amount_sold) DESC)
  OVER (PARTITION BY country_name, prod_name) worst_year
FROM sales, times, customers, countries, products
WHERE sales.time_id = times.time_id AND sales.prod_id = products.prod_id AND
  sales.cust_id =customers.cust_id AND customers.country_id=countries.country_id
GROUP BY country_name, prod_name, calendar_year;
```

This query computes `SUM` and `COUNT` aggregates on the sales data grouped by country, product, and year. It will report for each product sold in a country, the year when the sales were the highest for that product in that country. This is called the `best_year` of the product. Similarly, `worst_year` gives the year when the sales were the lowest.

## 23.2.2 MODEL Clause Syntax

The `MODEL` clause enables you to define multi-dimensional calculations on the data in the SQL query block. In multi-dimensional applications, a fact table consists of columns that uniquely identify a row with the rest serving as dependent measures or attributes. The `MODEL` clause lets you specify the `PARTITION`, `DIMENSION`, and `MEASURE` columns that define the multi-dimensional array, the rules that operate on this multi-dimensional array, and the processing options.

The `MODEL` clause contains a list of updates representing array computation within a partition and is a part of a SQL query block. Its structure is as follows:

```
MODEL
[<global reference options>]
[<reference models>]
[MAIN <main-name>]
  [PARTITION BY (<cols>)]
  DIMENSION BY (<cols>)
  MEASURES (<cols>)
  [<reference options>]
  [RULES]  <rule options>
  (<rule>, <rule>,.., <rule>)
  <global reference options> ::= <reference options> <ret-opt>
   <ret-opt> ::= RETURN {ALL|UPDATED} ROWS
  <reference options> ::=
  [IGNORE NAV | [KEEP NAV]
  [UNIQUE DIMENSION | UNIQUE SINGLE REFERENCE]
  <rule options> ::=
  [UPDATE | UPSERT | UPSERT ALL]
  [AUTOMATIC ORDER | SEQUENTIAL ORDER]
  [ITERATE (<number>)  [UNTIL <condition>]]
  <reference models> ::= REFERENCE ON <ref-name> ON (<query>)
  DIMENSION BY (<cols>) MEASURES (<cols>) <reference options>
```

Each rule represents an assignment. Its left side references a cell or a set of cells and the right side can contain expressions involving constants, host variables, individual cells or aggregates


Copyright © 2001, 2025, Oracle and/or its affiliates.

over ranges of cells. For example, consider the query in Example 23-1, this is based on the view sales_view created as described in Base Schema for SQL Modeling Examples.

**Example 23-1 Simple Query with the MODEL Clause**

```
SELECT SUBSTR(country,1,20) country, SUBSTR(product,1,15) product, year, sales
FROM sales_view
WHERE country in ('Italy', 'Japan')
MODEL
  RETURN UPDATED ROWS
  MAIN simple_model
  PARTITION BY (country)
  DIMENSION BY (product, year)
  MEASURES (sales)
  RULES
   (sales['Bounce', 2021] = 1000,
    sales['Bounce', 2022] = sales['Bounce', 2021] + sales['Bounce', 2020],
    sales['Y Box', 2022] = sales['Y Box', 2021])
ORDER BY country, product, year;
```

This query defines model computation on the rows from sales_view for the countries Italy and Japan. This model has been given the name simple_model. It partitions the data on country and defines, within each partition, a two-dimensional array on product and year. Each cell in this array holds the value of the sales measure. The first rule of this model sets the sales of Bounce in year 2001 to 1000. The next two rules define that the sales of Bounce in 2002 are the sum of its sales in years 2001 and 2000, and the sales of Y Box in 2002 are same as that of the previous year 2001.

Specifying RETURN UPDATED ROWS makes the preceding query return only those rows that are updated or inserted by the model computation. By default or if you use RETURN ALL ROWS, you would get all rows not just the ones updated or inserted by the MODEL clause. The query produces the following output:

```
COUNTRY    PRODUCT       YEAR    SALES
__________ ___________ _______ ________
Italy      Bounce        2021     1000
Italy      Bounce        2022
Italy      Y Box         2022
Japan      Bounce        2021     1000
Japan      Bounce        2022
Japan      Y Box         2022
```

Note that the MODEL clause does not update or insert rows into database tables. The following query illustrates this by showing that sales_view has not been altered:

```
SELECT SUBSTR(country,1,20) country, SUBSTR(product,1,15) product, year, sales
FROM sales_view
WHERE country IN ('Italy', 'Japan');

COUNTRY    PRODUCT               YEAR       SALES
__________ ___________________ _______ ___________
Italy      11" Youth Field       2019     2070.22
Italy      11" Youth Field       2020     4360.09
Italy      11" Youth Field       2021        1276
Italy      11" Youth Field       2022     3186.55
Italy      11.5" Youth Tri       2019     4896.79
Italy      11.5" Youth Tri       2020    11063.49
...
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

Observe that the update of the sales value for Bounce in the 2001 done by this MODEL clause is not reflected in the database. If you want to update or insert rows in the database tables, you should use the INSERT, UPDATE, or MERGE statements.

In the preceding example, columns are specified in the PARTITION BY, DIMENSION BY, and MEASURES list. You can also specify constants, host variables, single-row functions, aggregate functions, analytical functions, or expressions involving them as partition and dimension keys and measures. However, you must alias them in PARTITION BY, DIMENSION BY, and MEASURES lists. You must use aliases to refer these expressions in the rules, SELECT list, and the query ORDER BY. The following example shows how to use expressions and aliases:

```
SELECT country, p product, year, sales, profits
FROM sales_view
WHERE country IN ('Italy', 'Japan')
MODEL
  RETURN UPDATED ROWS
  PARTITION BY (SUBSTR(country,1,20) AS country)
  DIMENSION BY (product AS p, year)
  MEASURES (sales, 0 AS profits)
  RULES
   (profits['11" Youth Field ', 2021] = sales['11" Youth Field ', 2021] * 0.25,
    sales['11" Youth Field e', 2022] = sales['11" Youth Field ', 2021] + sales['11"
Youth Field ', 2000],
    profits['11" Youth Field ', 2022] = sales['11" Youth Field ', 2022] * 0.35)
ORDER BY country, year;

COUNTRY    PRODUCT                 YEAR    SALES    PROFITS
__________ _____________________ _______ ________ __________
Italy      11" Youth Field         2021
Italy      11" Youth Field e       2022
Italy      11" Youth Field         2022
Japan      11" Youth Field         2021
Japan      11" Youth Field         2022
Japan      11" Youth Field e       2022
```

Note that the alias "0 AS profits" initializes all cells of the profits measure to 0. See *Oracle Database SQL Language Reference* for more information regarding MODEL clause syntax.

## 23.2.3 Keywords in SQL Modeling

This section defines keywords used in SQL modeling. It contains the following topics:

- Assigning Values and Null Handling
- Calculation Definition

### 23.2.3.1 Assigning Values and Null Handling

- UPSERT

  This updates the measure values of existing cells. If the cells do not exist, and the rule has appropriate notation, they are inserted. If any of the cell references are symbolic, no cells are inserted.

- UPSERT ALL

  This is similar to UPSERT, except it allows a broader set of rule notation to insert new cells.

- UPDATE

  This updates existing cell values. If the cell values do not exist, no updates are done.


Copyright © 2001, 2025, Oracle and/or its affiliates.

- `IGNORE NAV`

  For numeric cells, this treats values that are not available as 0. This means that a cell not supplied to `MODEL` by the query result set will be treated as a zero for the calculation. This can be used at a global level for all measures in a model.

- `KEEP NAV`

  This keeps cell values that are not available unchanged. It is useful for making exceptions when `IGNORE NAV` is specified at the global level. This is the default, and can be omitted.

### 23.2.3.2 Calculation Definition

- `MEASURES`

  The set of values that are modified or created by the model.

- `RULES`

  The expressions that assign values to measures.

- `AUTOMATIC ORDER`

  This causes all rules to be evaluated in an order based on their logical dependencies.

- `SEQUENTIAL ORDER`

  This causes rules to be evaluated in the order they are written. This is the default.

- `UNIQUE DIMENSION`

  This is the default, and it means that the combination of `PARTITION BY` and `DIMENSION BY` columns in the `MODEL` clause must uniquely identify each and every cell in the model. This uniqueness is explicitly verified at query execution when necessary, in which case it may increase processing time.

- `UNIQUE SINGLE REFERENCE`

  The `PARTITION BY` and `DIMENSION BY` clauses uniquely identify single point references on the right-hand side of the rules. This may reduce processing time by avoiding explicit checks for uniqueness at query execution.

- `RETURN [ALL|UPDATED] ROWS`

  This enables you to specify whether to return all rows selected or only those rows updated by the rules. The default is `ALL`, while the alternative is `UPDATED ROWS`.

## 23.2.4 About Cell Referencing in SQL Modeling

In the `MODEL` clause, a relation is treated as a multi-dimensional array of cells. A cell of this multi-dimensional array contains the measure values and is indexed using `DIMENSION BY` keys, within each partition defined by the `PARTITION BY` keys. For example, consider the following query run on the view `sales_view` created as described in Base Schema for SQL Modeling Examples:

```
SELECT country, product, year, sales, best_year, best_year
FROM sales_view
MODEL
  PARTITION BY (country)
  DIMENSION BY (product, year)
  MEASURES (sales, best_year)
  (<rules> ..)
ORDER BY country, product, year;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

This partitions the data by country and defines within each partition, a two-dimensional array on product and year. The cells of this array contain two measures: `sales` and `best_year`.

Accessing the measure value of a cell by specifying the `DIMENSION BY` keys constitutes a cell reference. An example of a cell reference is as follows:

```
sales[product= 'Bounce', year=2020]
```

Here, you are accessing the sales value of a cell referenced by product Bounce and the year 2000. In a cell reference, you can specify `DIMENSION BY` keys either symbolically as in the preceding cell reference or positionally as in `sales['11" Youth Field ', 2020]`.

This section contains the following topics:

- Symbolic Dimension References
- Positional Dimension References

### 23.2.4.1 Symbolic Dimension References

A symbolic dimension reference (or symbolic reference) is one in which `DIMENSION BY` key values are specified with a boolean expression. For example, the cell reference `sales[year >= 2001]` has a symbolic reference on the `DIMENSION BY` key year and specifies all cells whose year value is greater than or equal to 2001. An example of symbolic references on product and year dimensions is `sales[product = 'Bounce', year >= 2001]`.

### 23.2.4.2 Positional Dimension References

A positional dimension reference (or positional reference, in short) is a constant or a constant expression specified for a dimension. For example, the cell reference `sales['Bounce']` has a positional reference on the product dimension and accesses sales value for the product Bounce. The constants (or constant expressions) in a cell reference are matched to the column order specified for `DIMENSION BY` keys. The following example shows the usage of positional references on dimensions:

```
sales['Bounce', 2001]
```

Assuming `DIMENSION BY` keys to be product and year in that order, it accesses the sales value for Bounce and 2001.

Based on how they are specified, cell references are either single cell or multi-cell reference.

## 23.2.5 About Rules for SQL Modeling

Model computation is expressed in rules that manipulate the cells of the multi-dimensional array defined by `PARTITION BY`, `DIMENSION BY`, and `MEASURES` clauses. A rule is an assignment statement whose left side represents a cell or a range of cells and whose right side is an expression involving constants, bind variables, individual cells or an aggregate function on a range of cells. Rules can use wild cards and looping constructs for maximum expressiveness. An example of a rule is the following:

```
sales['Cricket - Athle', 2020] = 1.2 * sales['Cricket - Athle', 2019]
```

This rule says that, for the product - `Cricket - Athle`, the sales for 2020 are 20% more than that of 2019.


Copyright © 2001, 2025, Oracle and/or its affiliates.

Note that this rule refers to single cells on both the left and right side and is relatively simple. Complex rules can be written with multi-cell references, aggregates, and nested cell references.

### Single Cell References

This type of rule involves single cell reference on the left side with constants and single cell references on the right side. Some examples are the following:

```
sales[product='Wicket Keeper G', year=2021] = 5662
sales['Wicket Keeper G', 2022] = 1.2 * sales['Wicket Keeper G', 2021]
sales[product='Wide Brim Hat', year=2021] = 0.8 * sales['Tennis Strings',
  year=2022] + sales['Wicket Keeper G', 2019]
```

### Multi-Cell References on the Right Side

Multi-cell references can be used on the right side of rules, in which case an aggregate function needs to be applied on them to convert them to a single value. All existing aggregate functions including analytic aggregates (inverse percentile functions, hypothetical rank and distribution functions and so on) and statistical aggregates (correlation, regression slope and so on), and user-defined aggregate functions can be used. Windowing functions such as `RANK` and `MOVING_AVG` can be used as well. For example, the rule to compute the sales of `Team shirt` for 2022 to be 100 more than the maximum sales in the period 2019 to 2021 would be:

```
sales['Team shirt', 2022] = 100 + MAX(sales)['Team shirt', year BETWEEN 2019 AND 2021]
```

The following example illustrates the usage of inverse percentile function `PERCENTILE_DISC`. It projects Finding Fido sales for year 2003 to be 30% more than the median sales for products Finding Fido, Match Used Auto, and Boat for all years prior to 2003.

```
sales[product='Spiked Shoes', year=2019] = 1.5 *
  PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY sales) [product IN ('Spiked Shoes','Tennis
Racquet ','Sonic Core Grap'), year < 2018]
```

Aggregate functions can appear only on the right side of rules. Arguments to the aggregate function can be constants, bind variables, measures of the `MODEL` clause, or expressions involving them. For example, the rule computes the sales of `Soccer Jersey` for 2022 to be the weighted average of its sales for years from 2018 to 2021 would be:

```
sales['Soccer Jersey ', 2022] =
   AVG(sales * weight)['Soccer Jersey', year BETWEEN 2018 AND 2021]
```

### Multi-Cell References on the Left Side

Rules can have multi-cell references on the left side as in the following:

```
sales['Soccer Goal - P ', year > 2021] =
   0.2 * sales['Right-Handed Gr', year=2019]
```

This rule accesses a range of cells on the left side (cells for product `Soccer Goal - P` and year greater than 2021) and assigns sales measure of each such cell to the value computed by the right side expression. Computation by the preceding rule is described as "sales of `Soccer Goal - P` for years after 2021" is 20% of the sales of `Right-Handed Gr` for year 2019 . This computation is simple in that the right side cell references and hence the right side expression are the same for all cells referenced on the left.

### Use of the CV Function

The use of the `CV` function provides the capability of relative indexing where dimension values of the cell referenced on the left side are used on the right side cell references. The `CV` function


Copyright © 2001, 2025, Oracle and/or its affiliates.

takes a dimension key as its argument, so it provides the value of a DIMENSION BY key of the cell currently referenced on the left side. As an example, consider the following:

```
sales[product='Linseed Oil', year>2022] =
  sales[CV(product), CV(year)] + 0.2 * sales['Match Used Auto', 2020]
```

When the left side references the cell Linseed Oil and 2022, the right side expression would be:

```
sales['Match Used Auto', 2020] + 0.2 * sales['Linseed Oil', 2022]
```

Similarly, when the left side references the cell Match Used Auto and 2022, the right side expression you would evaluate is:

```
sales['Match Used Auto', 2022] + 0.2 * sales['Linseed Oil', 2022]
```

It is also possible to use CV without any argument as in CV() and in which case, positional referencing is implied. CV() may be used outside a cell reference, but when used in this way its argument must contain the name of the dimension desired. You can also write the preceding rule as:

```
sales[product='Match Used Auto', year>2020] =
  sales[CV(), CV()] + 0.2 * sales[''Pitching Machin'', 2020]
```

The first CV() reference corresponds to CV(product) and the latter corresponds to CV(year). The CV function can be used only in right side cell references. Another example of the usage of CV function is the following:

```
sales[product IN ('Linseed Oil','Match Used Auto','Plastic - Beach'), year
  BETWEEN 2022 AND 2023] = 2 * sales[CV(product), CV(year)-10]
```

This rule says that, for products Linseed Oil, Match Used Auto, and Plastic - Beach, the sales for years between 2022 and 2023 will be twice of what their sales were 10 years ago.

### Use of the ANY Wildcard

You can use the wild card ANY in cell references to match all dimension values including nulls. ANY may be used on both the left and right side of rules. For example, a rule for the computation "sales of all products for 2023 are 10% more than their sales for 2022" would be the following:

```
sales[product IS ANY, 2023] = 1.1 * sales[CV(product), 2022]
```

Using positional references, it can also be written as:

```
sales[ANY, 2023] = 1.1 * sales[CV(), 2022]
```

Note that ANY is treated as a symbolic reference even if it is specified positionally, because it really means that (dimension IS NOT NULL OR dimension IS NULL).

### Nested Cell References

Cell references can be nested. In other words, cell references providing dimension values can be used within a cell reference. An example, assuming best_year is a measure, for nested cell reference is given as follows:

```
sales[product='Pitching Machin', year = best_year['Pitching Machin', 2019]]
```

Here, the nested cell reference best_year['Pitching Machin', 2019] provides value for the dimension key year and is used in the symbolic reference for year. Measures best_year and worst_year give, for each year (y) and product (p) combination, the year for which sales of


Copyright © 2001, 2025, Oracle and/or its affiliates.

product p were highest or lowest. The following rule computes the sales of Match Used Auto for 2003 to be the average of Match Used Auto sales for the years in which Finding Fido sales were highest and lowest:

```
sales['Match Used Auto', 2023] = (sales[CV(), best_year['Pitching Machin',
  CV(year)]] + sales[CV(), worst_year['Pitching Machin', CV(year)]]) / 2
```

Oracle AI Database allows only one level of nesting, and only single cell references can be used as nested cell references. Aggregates on multi-cell references cannot be used in nested cell references.

## 23.2.6 Order of Evaluation of SQL Modeling Rules

By default, rules are evaluated in the order they appear in the MODEL clause. You can specify an optional keyword SEQUENTIAL ORDER in the MODEL clause to make such an evaluation order explicit. SQL models with sequential rule order of evaluation are called sequential order models. For example, the following RULES specification makes Oracle AI Database evaluate rules in the specified sequence:

```
RULES SEQUENTIAL ORDER
 (sales['Match Used Auto', 2021] =
    sales['Match Used Auto', 2020] + sales['Bounce', 2019],   --Rule R1
  sales['Match Used Auto', 2020] = 50000,                     --Rule R2
  sales['Match Used Auto', 2019] = 40000)                     --Rule R3
```

Alternatively, the option AUTOMATIC ORDER enables Oracle AI Database to determine the order of evaluation of rules automatically. Oracle examines the cell references within rules and finds dependencies among rules. If cells referenced on the left side of rule R1 are referenced on the right side of another rule R2, then R2 is considered to depend on R1. In other words, rule R1 should be evaluated before rule R2. If you specify AUTOMATIC ORDER in the preceding example as in:

```
RULES AUTOMATIC ORDER
 (sales['Match Used Auto', 2021] = sales['Match Used Auto', 2020] + sales['Match Used
Auto', 2019],
  sales['Match Used Auto', 2020] = 50000,
  sales['Match Used Auto', 2019] = 40000)
```

Rules 2 and 3 are evaluated, in some arbitrary order, before rule 1. This is because rule 1 depends on rules 2 and 3 and hence need to be evaluated after rules 2 and 3. The order of evaluation among second and third rules can be arbitrary as they do not depend on one another. The order of evaluation among rules independent of one another can be arbitrary. SQL models with an automatic order of evaluation, as in the preceding fragment, are called automatic order models.

In an automatic order model, multiple assignments to the same cell are not allowed. In other words, measure of a cell can be assigned only once. Oracle AI Database will return an error in such cases as results would be non-deterministic. For example, the following rule specification will generate an error as sales['Match Used Auto', 2021] is assigned more than once:

```
RULES AUTOMATIC ORDER
 (sales['Match Used Auto', 2021] = sales['Match Used Auto', 2020] + sales['Match Used
Auto', 2019],
  sales['Match Used Auto', 2021] = 50000,
  sales['Match Used Auto', 2021] = 40000)
```

The rules assigning the sales of product Bounce for 2001 do not depend on one another and hence, no particular evaluation order can be fixed among them. This leads to non-deterministic results as the evaluation order is arbitrary - sales['Match Used Auto', 2021] can be 40000


Copyright © 2001, 2025, Oracle and/or its affiliates.

or 50000 or sum of Bounce sales for years 1999 and 2000. Oracle AI Database prevents this by disallowing multiple assignments when `AUTOMATIC ORDER` is specified. However, multiple assignments are fine in sequential order models. If `SEQUENTIAL ORDER` was specified instead of `AUTOMATIC ORDER` in the preceding example, the result of `sales['Match Used Auto', 2021]` would be 40000.

## 23.2.7 Global and Local Keywords for SQL Modeling Rules

You can specify an `UPDATE`, `UPSERT`, `UPSERT ALL`, `IGNORE NAV`, and `KEEP NAV` option at the global level in the `RULES` clause in which case all rules operate in the respective mode. These options can be specified at a local level with each rule and in which case, they override the global behavior. For example, in the following specification:

```
RULES UPDATE
(UPDATE s['Match Used Auto',2021] = sales['Match Used Auto',2020] + sales['Match Used
Auto',2019],
 UPSERT s['Pro Maple Bat', 2021] = sales['Pro Maple Bat', 2020] + sales['Pro Maple Bat',
2019],
   sales['Pro Maple Youth', 2021] = sales['Pro Maple Youth', 2020] +
   sales['Pro Maple Youth',2019])
```

The `UPDATE` option is specified at the global level so, the first and third rules operate in update mode. The second rule operates in upsert mode as an `UPSERT` keyword is specified with that rule. Note that no option was specified for the third rule and hence it inherits the update behavior from the global option.

## 23.2.8 UPDATE, UPSERT, and UPSERT ALL Behavior

You can determine how cells in rules behave by choosing whether to have `UPDATE`, `UPSERT`, or `UPSERT ALL` semantics. By default, rules in the `MODEL` clause have `UPSERT` semantics, though you can specify an optional `UPSERT` keyword to make the upsert semantic explicit.

The following sections discuss these three types of behavior:

- UPDATE Behavior
- UPSERT Behavior
- UPSERT ALL Behavior

### 23.2.8.1 UPDATE Behavior

The `UPDATE` option forces strict update mode. In this mode, the rule is ignored if the cell it references on the left side does not exist. If the cell referenced on the left side of a rule exists, then its measure is updated with the value of the right side expression. Otherwise, if a cell reference is positional, a new cell is created (that is, inserted into the multi-dimensional array) with the measure value equal to the value of the right side expression. If a cell reference is not positional, it will not insert cells. Note that if there are any symbolic references in a cell's specification, inserts are not possible in an upsert rule. For example, consider the following rule:

```
sales['Match Used Auto', 2023] = sales['Match Used Auto', 2021] + sales ['Match Used
Auto', 2022]
```

The cell for product `Match Used Auto` and year 2023, if it exists, gets updated with the sum of Bounce sales for years 2021 and 2022, otherwise, it gets created. If you had created the same rule using any symbolic references, no updates would be performed, as in the following:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
sales[prod= 'Match Used Auto', year= 2003] = sales['Match Used Auto', 2021] + sales
['Match Used Auto', 2022]
```

## 23.2.8.2 UPSERT Behavior

Using UPSERT creates a new cell corresponding to the one referenced on the left side of the rule when the cell is missing, and the cell reference contains only positional references qualified by constants. Note that cell references created with FOR loops (described in "Advanced Topics in SQL Modeling") are treated as positional references, so the values FOR loops create will be used to insert new cells. Assuming you do not have cells for years greater than 2020, consider the following rule:

```
UPSERT sales['Match Used Auto', year = 2021] = 1.1 * sales['Match Used Auto', 2020]
```

This would not create any new cell because of the symbolic reference year = 2021. However, consider the following:

```
UPSERT sales['Match Used Auto', 2021] = 1.1 * sales['Match Used Auto', 2020]
```

This would create a new cell for product Match Used Auto for year 2021. On a related note, new cells will not be created if any of the references is ANY. This is because ANY is a predicate that qualifies all dimensional values including NULL. If there is a reference ANY for a dimension d, then it means the same thing as the predicate (d IS NOT NULL OR d IS NULL).

If an UPSERT rule uses FOR loops in its left side cell references, the list of upsert cells is generated by performing a cross product of all the distinct values for each dimension. Although UPSERT with FOR loops can be used to densify dimensions (see "Data Densification for Reporting"), it is generally preferable to densify using the partitioned outer join operation.

## 23.2.8.3 UPSERT ALL Behavior

UPSERT ALL behavior allows model rules with existential predicates (comparisons, IN, ANY, and so on) in their left side to have UPSERT behavior. As an example, the following uses ANY and creates Bay Area as the combination of San Francisco, San Jose, and Oakland:

```
SELECT product, time, city, s sales
FROM cube_subquery
MODEL PARTITION BY (product)
DIMENSION BY (time, city) MEASURES(sales s
RULES UPSERT ALL
(s[ANY, 'Bay Area'] =
   s[CV(), 'San Francisco'] + s[CV(), 'San Jose'] + s[CV(), 'Oakland']
s['2020', ANY] = s['2018', CV()] + s['2019', CV()]);
```

In this example, the first rule simply inserts a Bay Area cell for each distinct time value, and the second rule inserts a 2004 cell for each distinct city value including Bay Area. This example is relatively simple as the existential predicates used on the left side are ANY predicates, but you can also use UPSERT ALL with more complex calculations.

It is important to understand exactly what the UPSERT ALL operation does, especially in cases where there is more than one symbolic dimension reference. Note that the behavior is different than the behavior of an UPSERT rule that uses FOR loops.

When evaluating an UPSERT ALL rule, Oracle AI Database performs the following steps to create a list of cell references to be upserted:

1. Find the existing cells that satisfy all the symbolic predicates of the cell reference.


Copyright © 2001, 2025, Oracle and/or its affiliates.

2. Using just the dimensions that have symbolic references, find the distinct dimension value combinations of these cells.
3. Perform a cross product of these value combinations with the dimension values specified through positional references.
4. The results of Step 3 are then used to upsert new cells into the array.

### 23.2.8.3.1 Example: UPSERT ALL Behavior

To illustrate the four steps described in "UPSERT ALL Behavior", here is a brief example using abstracted data and a model with three dimensions. Consider a model dimensioned by (`product`, `time`, `city`) with a measure called `sales`. You wish to upsert new sales values for the city of `z`, and these sales values are copied from those of the city of `y`.

```
UPSERT ALL sales[ANY, ANY, 'z']= sales[CV(product),CV(time),'y']
```

Our source data set has these four rows:

```
PROD    TIME   CITY   SALES
   1    2022      x      10
   1    2023      x      15
   2    2022      y      21
   2    2023      y      24
```

The following explains the details of the four steps, applied to this data:

1. Because the symbolic predicates of the rule are `ANY`, any of the rows shown in this example is acceptable.
2. The distinct dimension combinations of cells with symbolic predicates that match the condition are: (1, 2022), (1, 2023), (2, 2022), and (2, 2023).
3. You find the cross product of these dimension combinations with the cells specified with positional references. In this case, it is simply a cross product with the value `z`, and the resulting cell references are: (1, 2022, z), (1, 2023, z), (2, 2022, z), and (2, 2023, z).
4. The cells listed in Step 3 will be upserted, with sales calculated based on the city `y`. Because there are no values for product 1 in city `y`, those cells created for product 1 will have `NULL` as their sales value. Of course, a different rule might have generated non-`NULL` results for all the new cells. Our result set includes the four original rows plus four new rows:

```
PROD    TIME   CITY    SALES
   1    2022      x       10
   1    2023      x       15
   2    2022      y       21
   2    2023      y       24
   1    2022      z     NULL
   1    2023      z     NULL
   2    2022      z       21
   2    2023      z       24
```

It is important to note that these results are not a cross product using all values of all dimensions. If that were the case, you would have cells such as (1,2022, y) and (2,2023, x). Instead, the results here are created using dimension combinations found in existing rows.

## 23.2.9 Treatment of NULLs and Missing Cells in SQL Modeling

Applications using models would not only have to deal with non-deterministic values for a cell measure in the form of `NULL`, but also with non-determinism in the form of missing cells. A cell,


Copyright © 2001, 2025, Oracle and/or its affiliates.

referenced by a single cell reference, that is missing in the data is called a missing cell. The MODEL clause provides a default treatment for nulls and missing cells that is consistent with the ANSI SQL standard and also provides options to treat them in other useful ways according to business logic, for example, to treat nulls as zero for arithmetic operations.

By default, NULL cell measure values are treated the same way as nulls are treated elsewhere in SQL. For example, in the following rule:

```
sales['Plastic Cricket', 2021] = sales['Plastic Cricket', 2019] + sales['Plastic
Cricket', 2020]
```

The right side expression would evaluate to NULL if Plastic Cricket sales for one of the years 2019 and 2020 is NULL. Similarly, aggregate functions in rules would treat NULL values in the same way as their regular behavior where NULL values are ignored during aggregation.

Missing cells are treated as cells with NULL measure values. For example, in the preceding rule, if the cell for Plastic Cricket and 2020 is missing, then it is treated as a NULL value and the right side expression would evaluate to NULL.

This section contains the following topics:

- Distinguishing Missing Cells from NULLs
- Use Defaults for Missing Cells and NULLs
- Using NULLs in a Cell Reference

## 23.2.9.1 Distinguishing Missing Cells from NULLs

The functions PRESENTV and PRESENTNNV enable you to identify missing cells and distinguish them from NULL values. These functions take a single cell reference and two expressions as arguments as in PRESENTV(cell, expr1, expr2). PRESENTV returns the first expression expr1 if the cell cell is existent in the data input to the MODEL clause. Otherwise, it returns the second expression expr2. For example, consider the following:

```
PRESENTV(sales['Pro Maple Bat ', 2020], 1.1*sales['Pro Maple Bat ', 2020], 100)
```

If the cell for product Pro Maple Bat and year 2020 exists, it returns the corresponding sales multiplied by 1.1, otherwise, it returns 100. Note that if sales for the product Pro Maple Bat for year 2020 is NULL, the preceding specification would return NULL.

The PRESENTNNV function not only checks for the presence of a cell but also whether it is NULL or not. It returns the first expression expr1 if the cell exists and is not NULL, otherwise, it returns the second expression expr2. For example, consider the following:

```
PRESENTNNV(sales['Pro Maple Bat ', 2020], 1.1*sales['Pro Maple Bat ', 2020], 100)
```

This would return 1.1*sales['Pro Maple Bat ', 2020] if sales['Pro Maple Bat ', 2020] exists and is not NULL. Otherwise, it returns 100.

Applications can use the IS PRESENT predicate in their model to check the presence of a cell in an explicit fashion.This predicate returns TRUE if cell exists and FALSE otherwise. The preceding example using PRESENTNNV can be written using IS PRESENT as:

```
CASE WHEN sales['Pro Maple Bat ', 2020] IS PRESENT AND sales['Pro Maple Bat ', 2020] IS
NOT NULL
THEN 1.1 * sales['Pro Maple Bat ', 2020]
ELSE 100
END
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

The IS PRESENT predicate, like the PRESENTV and PRESENTNNV functions, checks for cell existence in the input data, that is, the data as existed before the execution of the MODEL clause. This enables you to initialize multiple measures of a cell newly inserted by an UPSERT rule. For example, if you want to initialize sales and profit values of a cell, if it does not exist in the data, for product Pro Maple Bat and year 2023 to 1000 and 500 respectively, you can do so by the following:

```
RULES
 (UPSERT sales['Pro Maple Bat ', 2023] =
    PRESENTV(sales['Pro Maple Bat ', 2023], sales['Pro Maple Bat ', 2023], 1000),
  UPSERT profit['Pro Maple Bat ', 2023] =
    PRESENTV(profit['Pro Maple Bat ', 2023], profit['Pro Maple Bat ', 2023], 500))
```

The PRESENTV functions used in this formulation return TRUE or FALSE based on the existence of the cell in the input data. If the cell for Pro Maple Bat and 2023 gets inserted by one of the rules, based on their evaluation order, PRESENTV function in the other rule would still evaluate to FALSE. You can consider this behavior as a preprocessing step to rule evaluation that evaluates and replaces all PRESENTV and PRESENTNNV functions and IS PRESENT predicate by their respective values.

## 23.2.9.2 Use Defaults for Missing Cells and NULLs

The MODEL clause, by default, treats missing cells as cells with NULL measure values. An optional KEEP NAV keyword can be specified in the MODEL clause to get this behavior.If your application wants to default missing cells and nulls to some values, you can do so by using IS PRESENT, IS NULL predicates and PRESENTV, PRESENTNNV functions. But it may become cumbersome if you have lot of single cell references and rules. You can use IGNORE NAV option instead of the default KEEP NAV option to default nulls and missing cells to:

- 0 for numeric data
- Empty string for character/string data
- 01-JAN-2021 for data type data
- NULL for all other data types

Consider the following query:

```
SELECT product, year, sales
FROM sales_view
WHERE country = 'Poland'
MODEL
  DIMENSION BY (product, year) MEASURES (sales sales) IGNORE NAV
  RULES UPSERT
  (sales['Tennis Strings', 2023] = sales['Tennis Strings', 2022] + sales['Tennis
Strings', 2021]);
```

In this, the input to the MODEL clause does not have a cell for product Bounce and year 2022. Because of IGNORE NAV option, sales['Tennis Strings', 2022] value would default to 0 (as sales is of numeric type) instead of NULL. Thus, sales['Tennis Strings', 2023] value would be same as that of sales['Tennis Strings', 2021].

## 23.2.9.3 Using NULLs in a Cell Reference

To use NULL values in a cell reference, you must use one of the following:

- Positional reference using wild card ANY as in sales[ANY].
- Symbolic reference using the IS ANY predicate as in sales[product IS ANY].


Copyright © 2001, 2025, Oracle and/or its affiliates.

- Positional reference of `NULL` as in `sales[NULL]`.
- Symbolic reference using `IS NULL` predicate as in `sales[product IS NULL]`.

Note that symbolic reference `sales[product = NULL]` would not test for nulls in the product dimension. This behavior conforms with the standard handling of nulls by SQL.

## 23.2.10 About Reference Models in SQL Modeling

In addition to the multi-dimensional array on which rules operate, which is called the main model, one or more read-only multi-dimensional arrays, called reference models, can be created and referenced in the `MODEL` clause to act as look-up tables for the main model. Like the main model, a reference model is defined over a query block and has `DIMENSION BY` and `MEASURES` clauses to indicate its dimensions and measures respectively. A reference model is created by the following subclause:

```
REFERENCE model_name ON (query) DIMENSION BY (cols) MEASURES (cols)
   [reference options]
```

Like the main model, a multi-dimensional array for the reference model is built before evaluating the rules. But, unlike the main model, reference models are read-only in that their cells cannot be updated and no new cells can be inserted after they are built. Thus, the rules in the main model can access cells of a reference model, but they cannot update or insert new cells into the reference model. The following is an example using a currency conversion table as a reference model:

```
CREATE TABLE dollar_conv_tbl(country VARCHAR2(30), exchange_rate NUMBER);
INSERT INTO dollar_conv_tbl VALUES('Poland', 0.25);
INSERT INTO dollar_conv_tbl VALUES('France', 0.14);
...
```

Now, to convert the projected sales of Poland and France for 2023 to the US dollar, you can use the dollar conversion table as a reference model as in the following command. The view sales_view was created as described in Base Schema for SQL Modeling Examples.

```
SELECT country, year, sales, dollar_sales
FROM sales_view
GROUP BY country, year
MODEL
  REFERENCE conv_ref ON (SELECT country, exchange_rate FROM dollar_conv_tbl)
    DIMENSION BY (country) MEASURES (exchange_rate) IGNORE NAV
  MAIN conversion
    DIMENSION BY (country, year)
    MEASURES (SUM(sales) sales, SUM(sales) dollar_sales) IGNORE NAV
RULES
(dollar_sales['France', 2023] = sales[CV(country), 2022] * 1.02 *
   conv_ref.exchange_rate['France'],
   dollar_sales['Poland', 2023] =
      sales['Poland', 2022] * 1.05 * exchange_rate['Poland']);
```

Observe in this example that:

- A one dimensional reference model named `conv_ref` is created on rows from the table `dollar_conv_tbl` and that its measure `exchange_rate` has been referenced in the rules of the main model.
- The main model (called `conversion`) has two dimensions, country and year, whereas the reference model `conv_ref` has one dimension, country.
- Different styles of accessing the `exchange_rate` measure of the reference model. For France, it is rather explicit with `model_name.measure_name` notation


Copyright © 2001, 2025, Oracle and/or its affiliates.

conv_ref.exchange_rate, whereas for Poland, it is a simple measure_name reference exchange_rate. The former notation needs to be used to resolve any ambiguities in column names across main and reference models.

Growth rates, in this example, are hard coded in the rules. The growth rate for France is 2% and that of Poland is 5%. But they could come from a separate table and you can have a reference model defined on top of that. Assume that you have a growth_rate(country, year, rate) table defined as the following:

```
CREATE TABLE growth_rate_tbl(country VARCHAR2(30),
   year NUMBER, growth_rate NUMBER);
INSERT INTO growth_rate_tbl VALUES('Poland', 2022, 2.5);
INSERT INTO growth_rate_tbl VALUES('Poland', 2023, 5);
...
INSERT INTO growth_rate_tbl VALUES('France', 2022, 3);
INSERT INTO growth_rate_tbl VALUES('France', 2023, 2.5);
```

Then the following query computes the projected sales in dollars for 2023 for all countries:

```
SELECT country, year, sales, dollar_sales
FROM sales_view
GROUP BY country, year
MODEL
  REFERENCE conv_ref ON
          (SELECT country, exchange_rate FROM dollar_conv_tbl)
           DIMENSION BY (country c) MEASURES (exchange_rate) IGNORE NAV
  REFERENCE growth_ref ON
          (SELECT country, year, growth_rate FROM growth_rate_tbl)
           DIMENSION BY (country c, year y) MEASURES (growth_rate) IGNORE NAV
  MAIN projection
   DIMENSION BY (country, year) MEASURES (SUM(sales) sales, 0 dollar_sales)
  IGNORE NAV
  RULES
  (dollar_sales[ANY, 2023] = sales[CV(country), 2022] *
   growth_rate[CV(country), CV(year)] *
   exchange_rate[CV(country)]);
```

This query shows the capability of the MODEL clause in dealing with and relating objects of different dimensionality. Reference model conv_ref has one dimension while the reference model growth_ref and the main model have two dimensions. Dimensions in the single cell references on reference models are specified using the CV function thus relating the cells in main model with the reference model. This specification, in effect, is performing a relational join between main and reference models.

Reference models also help you convert keys to sequence numbers, perform computations using sequence numbers (for example, where a prior period would be used in a subtraction operation), and then convert sequence numbers back to keys. For example, consider a view that assigns sequence numbers to years:

```
CREATE or REPLACE VIEW year_2_seq (i, year) AS
SELECT ROW_NUMBER() OVER (ORDER BY calendar_year), calendar_year
FROM (SELECT DISTINCT calendar_year FROM TIMES);
```

This view can define two lookup tables: integer-to-year i2y, which maps sequence numbers to integers, and year-to-integer y2i, which performs the reverse mapping. The references y2i.i[year] and y2i.i[year] - 1 return sequence numbers of the current and previous years respectively and the reference i2y.y[y2i.i[year]-1] returns the year key value of the previous year. The following query demonstrates such a usage of reference models:

```
SELECT country, product, year, sales, prior_period
FROM sales_view
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
MODEL
  REFERENCE y2i ON (SELECT year, i FROM year_2_seq) DIMENSION BY (year y)
   MEASURES (i)
  REFERENCE i2y ON (SELECT year, i FROM year_2_seq) DIMENSION BY (i)
   MEASURES (year y)
  MAIN projection2 PARTITION BY (country)
  DIMENSION BY (product, year)
  MEASURES (sales, CAST(NULL AS NUMBER) prior_period)
(prior_period[ANY, ANY] = sales[CV(product), i2y.y[y2i.i[CV(year)]-1]])
ORDER BY country, product, year;
```

Nesting of reference model cell references is evident in the preceding example. Cell reference on the reference model `y2i` is nested inside the cell reference on `i2y` which, in turn, is nested in the cell reference on the main SQL model. There is no limitation on the levels of nesting you can have on reference model cell references. However, you can only have two levels of nesting on the main SQL model cell references.

Finally, the following are restrictions on the specification and usage of reference models:

- Reference models cannot have a `PARTITION BY` clause.
- The query block on which the reference model is defined cannot be correlated to an outer query.
- Reference models must be named and their names should be unique.
- All references to the cells of a reference model should be single cell references.

# 23.3 Advanced Topics in SQL Modeling

This section discusses more advanced topics in SQL modeling, and includes:

- FOR Loops in SQL Modeling
- Iterative Models in SQL Modeling
- Rule Dependency in AUTOMATIC ORDER Models
- Ordered Rules in SQL Modeling
- Analytic Functions in SQL Modeling
- Unique Dimensions Versus Unique Single References in SQL Modeling
- Rules and Restrictions when Using SQL for Modeling

## 23.3.1 FOR Loops in SQL Modeling

The `MODEL` clause provides a `FOR` construct that can be used inside rules to express computations more compactly. It can be used on both the left and right side of a rule. `FOR` loops are treated as positional references when on the left side of a rule. For example, consider the following computation, which estimates the sales of several products for 2020 to be 10% higher than their sales for 2019:

```
RULES UPSERT
(sales['Wide Brim Hat', 2020] = 1.1 * sales['Wide Brim Hat', 2019],
 sales['11" Youth Field', 2020] = 1.1 * sales['11" Youth Field', 2019],
...
 sales['12" Premium Ser', 2020] = 1.1 * sales['12" Premium Ser', 2019])
```

The `UPSERT` option is used in this computation so that cells for these products and 2020 will be inserted if they are not previously present in the multi-dimensional array. This is rather bulky as


Copyright © 2001, 2025, Oracle and/or its affiliates.

you have to have as many rules as there are products. Using the FOR construct, this computation can be represented compactly and with exactly the same semantics as in:

```
RULES UPSERT
(sales[FOR product IN ('Wide Brim Hat', '11" Youth Field', ..., '12" Premium Ser'),
2020] =
   1.1 * sales[CV(product), 2019])
```

If you write a specification similar to this, but without the FOR keyword as in the following:

```
RULES UPSERT
(sales[product IN ('Wide Brim Hat', '11" Youth Field', ..., '12" Premium Ser'), 2020] =
   1.1 * sales[CV(product), 2019])
```

You would get UPDATE semantics even though you have specified UPSERT. In other words, existing cells will be updated but no new cells will be created by this specification. This is because the multi-cell reference on product is a symbolic reference and symbolic references do not permit insertion of new cells. You can view a FOR construct as a macro that generates multiple rules with positional references from a single rule, thus preserving the UPSERT semantics. Conceptually, the following rule:

```
sales[FOR product IN ('Wide Brim Hat', '11" Youth Field', ..., '12" Premium Ser'),
      FOR year IN (2020, 2020)] = 1.1 * sales[CV(product), CV(year)-1]
```

Can be treated as an ordered collection of the following rules:

```
sales['Wide Brim Hat', 2020] = 1.1 * sales[CV(product), CV(year)-1],
sales['Wide Brim Hat', 2020] = 1.1 * sales[CV(product), CV(year)-1],
sales['11" Youth Field', 2020] = 1.1 *
  sales[CV(product), CV(year)-1],
sales['11" Youth Field', 2020] = 1.1 * sales[CV(product),
 CV(year)-1],
...
sales['12" Premium Ser', 2020] = 1.1 * sales[CV(product), CV(year)-1],
sales['12" Premium Ser', 2021] = 1.1 * sales[CV(product), CV(year)-1]
```

The FOR construct in the preceding examples is of type FOR dimension IN *(list of values)*. Values in the list should be single-value expressions such as expressions of constants, single-cell references, and so on. In the last example, there are separate FOR constructs on product and year. It is also possible to specify all dimensions using one FOR construct and specify the values using multi-column IN lists. Consider for example, if you want only to estimate sales for Wide Brim Hat in 2020, 11" Youth Field in 2020 and 12" Premium Ser in 2020 and 2020. This can be formulated as the following:

```
sales[FOR (product, year) IN (('Wide Brim Hat', 2020), ('11" Youth Field', 2020),
  ('12" Premium Ser', 2020), ('12" Premium Ser', 2020))] =
     1.1 * sales[CV(product), CV(year)-1]
```

This FOR construct should be of the form FOR (d1, ..., dn) IN ((d1_val1, ..., dn_val1), ..., (d1_valm, ..., dn_valm)] when there are n dimensions d1, ..., dn and m values in the list.

In some cases, the list of values for a dimension in FOR can be retrieved from a table or a subquery. Oracle AI Database provides a type of FOR construct as in FOR dimension IN *(subquery)* to handle these cases. For example, assume that the products of interest are stored in a table interesting_products, then the following rule estimates their sales in 2020 and 2021:

```
sales[FOR product IN (SELECT product_name FROM interesting_products)
    FOR year IN (2020, 2021)] = 1.1 * sales[CV(product), CV(year)-1]
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

As another example, consider the scenario where you want to introduce a new country, called `new_country`, with sales that mimic those of Poland for all products and years where there are sales in Poland. This is accomplished by issuing the following statement:

```
SELECT country, product, year, s
FROM sales_view
MODEL
DIMENSION BY (country, product, year)
MEASURES (sales s) IGNORE NAV
RULES UPSERT
(s[FOR (country, product, year) IN
       (SELECT DISTINCT 'new_country', product, year
        FROM sales_view
        WHERE country = 'Poland')] = s['Poland',CV(),CV()])
ORDER BY country, year, product;
```

The view sales_view was created as described in Base Schema for SQL Modeling Examples.

Note the multi-column `IN`-list produced by evaluating the subquery in this specification. The subquery used to obtain the `IN`-list cannot be correlated to outer query blocks.

Note that the upsert list created by the rule is a cross-product of the distinct values for each dimension. For example, if there are 10 values for country, 5 values for year, and 3 values for product, you will generate an upsert list containing 150 cells.

If you know that the values of interest come from a discrete domain, you can use `FOR` construct `FOR dimension FROM value1 TO value2 [INCREMENT | DECREMENT] value3`. This specification results in values between `value1` and `value2` by starting from `value1` and incrementing (or decrementing) by `value3`. The values `value1`, `value2`, and `value3` should be single-value expressions. For example, the following rule:

```
sales['Wide Brim Hat', FOR year FROM 2017 TO 2020 INCREMENT 1] =
  sales['Wide Brim Hat', year=CV(year)-1] * 1.2
```

This is semantically equivalent to the following rules in order:

```
sales['Wide Brim Hat', 2017] = sales['Wide Brim Hat', 2016] * 1.2,
sales['Wide Brim Hat', 2018] = sales['Wide Brim Hat', 2017] * 1.2,
...
sales['Wide Brim Hat', 2020] = sales['Wide Brim Hat', 2019] * 1.2
```

This kind of `FOR` construct can be used for dimensions of numeric, date and datetime data types. The type for increment/decrement expression `value3` should be numeric for numeric dimensions and can be numeric or interval for dimensions of date or datetime types. Also, `value3` should be positive. Oracle AI Database returns an error if you use `FOR year FROM 2020 TO 2017 INCREMENT -1`. You should use either `FOR year FROM 2020 TO 2017 DECREMENT 1` or `FOR year FROM 2017 TO 2020 INCREMENT 1`.

To generate string values, you can use the `FOR` construct `FOR dimension LIKE string FROM value1 TO value2 [INCREMENT | DECREMENT] value3`. The string `string` should contain only one `%` character. This specification results in string by replacing `%` with values between `value1` and `value2` with appropriate increment/decrement value `value3`. For example, consider the following rule:

```
sales[FOR product LIKE 'product-%' FROM 1 TO 3 INCREMENT 1, 2019] =
sales[CV(product), 2018] * 1.2
```

This is equivalent to the following:


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
sales['product-1', 2019] = sales['product-1', 2018] * 1.2,
sales['product-2', 2019] = sales['product-2', 2018] * 1.2,
sales['product-3', 2020] = sales['product-3', 2019] * 1.2
```

In `SEQUENTIAL ORDER` models, rules represented by a `FOR` construct are evaluated in the order they are generated. On the contrary, rule evaluation order would be dependency based if `AUTOMATIC ORDER` is specified. For example, the evaluation order for the rules represented by the rule:

```
sales['Wide Brim Hat', FOR year FROM 2020 TO 2017 DECREMENT 1] =
  1.1 * sales['Wide Brim Hat', CV(year)-1]
```

For `SEQUENTIAL ORDER` models, the rules would be generated in this order:

```
sales['Wide Brim Hat', 2020] = 1.1 * sales['Wide Brim Hat', 2019],
sales['Wide Brim Hat', 2019] = 1.1 * sales['Wide Brim Hat', 2018],
sales['Wide Brim Hat', 2018] = 1.1 * sales['Wide Brim Hat', 2017],
sales['Wide Brim Hat', 2017] = 1.1 * sales['Wide Brim Hat', 2016]
```

While for `AUTOMATIC ORDER` models, the order would be equivalent to:

```
sales['Wide Brim Hat', 2017] = 1.1 * sales['Wide Brim Hat', 2016],
sales['Wide Brim Hat', 2018] = 1.1 * sales['Wide Brim Hat', 2017],
sales['Wide Brim Hat', 2019] = 1.1 * sales['Wide Brim Hat', 2018],
sales['Wide Brim Hat', 2020] = 1.1 * sales['Wide Brim Hat', 2019]
```

**See Also**

Evaluation of Formulas with FOR Loops

## 23.3.1.1 Evaluation of Formulas with FOR Loops

The `FOR` loop construct provides an iterative mechanism to generate single-value references for a dimension or for all dimensions (in the case of multi-column for `IN` lists). The evaluation of a formula with `FOR` loops on its left side basically consists of evaluation of the right side of the formula for each single-value reference generated by these `FOR` loops and assigning the result to the specified cell with this single-value reference. The generation of these single reference values is called "unfolding the `FOR` loop". These unfolded cells are evaluated in the order they are generated during the unfolding process.

How unfolding is performed depends on the `UPSERT`, `UPDATE`, and `UPDATE ALL` behavior specified for the rule and the specific characteristics of the rule. To understand this, a discussion of two stages of query processing is needed: query plan creation and query execution. Query plan creation is a stage where certain rule references are resolved in order to create an efficient query execution plan. Query execution is the stage where all remaining unresolved references must be determined. `FOR` loops may be unfolded at either query plan generation or at query execution. Below the details of the unfolding decision are discussed.

**See Also**

- Unfolding For UPDATE and UPSERT Rules
- Unfolding For UPSERT ALL: Rules
- Restrictions on Using FOR Loop Expressions on the Left Side of Formulas


Copyright © 2001, 2025, Oracle and/or its affiliates.

#### 23.3.1.1.1 Unfolding For UPDATE and UPSERT Rules

When using UPDATE or UPSERT rules, if unfolding the left side of a rule is guaranteed to generate single cell references, the unfolding is done at query execution. If the unfolding process cannot generate single cell references, unfolding is performed at query plan creation and a copy of the same formula for each generated reference by the unfolding process is created. For example, the unfolding of the following formula occurs at query execution as unfolding generates single cell references:

```
sales[FOR product IN ('prod1', 'prod2'), 2003] = sales[CV(product), 2002] * 1.2
```

However, consider the following formula, where unfolding reference values do not produce single value references due to the existence of a predicate on another dimension:

```
sales[FOR product in ('prod1', 'prod2'), year >= 2003]
   = sales[CV(product), 2002] * 1.2
```

There is no single-value reference on the year dimension, so even when the FOR loop is unfolded on the product dimension, there will be no single-value references on the left side of this formula. This means that the unfolding occurs at query plan creation and physically replace the original formula with the following formulas:

```
sales['prod1', year >= 2003]  = sales[CV(product), 2002] * 1.2,
sales['prod2', year >= 2003]  = sales[CV(product), 2002] * 1.2
```

The analysis and optimizations performed within the MODEL clause are done after unfolding at query plan creation (if that is what occurs), so, from that point on, everything is as if the multiple rules are specified explicitly in the MODEL clause. By performing unfolding at query plan creation in these cases, more accurate analysis and better optimization of formula evaluation is achieved. One thing to note is that there may be an increase in the number of formulas and, if this increase pushes the total number of formulas beyond the maximum limit, Oracle AI Database signals an error.

#### 23.3.1.1.2 Unfolding For UPSERT ALL: Rules

Rules with UPSERT ALL behavior have a very different approach to unfolding FOR loops. No matter what predicates are used, an UPSERT ALL rule will unfold FOR loops at query execution. This behavior avoids certain FOR loop restrictions discussed in the next section. However, there is a trade-off of fewer restrictions versus more optimized query plans. An UPSERT ALL rule tends toward slower performance than a similar UPSERT or UPDATE rule, and this should be considered when designing models.

#### 23.3.1.1.3 Restrictions on Using FOR Loop Expressions on the Left Side of Formulas

Restrictions on the use of FOR loop constructs are determined based on whether the unfolding takes place at query plan creation or at query execution. If a formula with FOR loops on its left side is unfolded at query plan creation (due to the reasons explained in the previous section), the expressions that need to be evaluated for unfolding must be expressions of constants whose values are available at query plan creation. For example, consider the following statement:

```
sales[For product like 'prod%' from ITERATION_NUMBER
to ITERATION_NUMBER+1, year >= 2003] =  sales[CV(product), 2002]*1.2
```

If this rule does not have UPSERT ALL specified for its behavior, it is unfolded at query plan creation. Because the value of the ITERATION_NUMBER is not known at query plan creation, and the value is needed to evaluate start and end expressions, Oracle AI Database signals an error


Copyright © 2001, 2025, Oracle and/or its affiliates.

unless that rule is unfolded at query execution. However, the following rule would be unfolded at query plan creation without any errors: the value of ITERATION_NUMBER is not needed for unfolding in this case, even though it appears as an expression in the FOR loop:

```
sales[For product in ('prod'||ITERATION_NUMBER, 'prod'||(ITERATION_NUMBER+1)),
 year >= 2003] =  sales[CV(product), 2002]*1.2
```

Expressions that have any of the following conditions cannot be evaluated at query plan creation:

- nested cell references
- reference model look-ups
- ITERATION_NUMBER references

Rules with FOR loops that require the results of such expressions causes an error if unfolded at query plan creation. However, these expressions will not cause any error if unfolding is done at query execution.

If a formula has subqueries in its FOR loop constructs and this formula requires compile-time unfolding, these subqueries are evaluated at query plan creation so that unfolding can happen. Evaluating a subquery at query plan creation can render a cursor non-sharable, which means the same query may need to be recompiled every time it is issued. If unfolding of such a formula is deferred to query execution, no compile-time evaluation is necessary and the formula has no impact on the sharability of the cursor.

Subqueries in the FOR loops of a formula can reference tables in the WITH clause if the formula is to be unfolded at query execution. If the formula has to be unfolded at query plan creation, Oracle AI Database signals an error.

## 23.3.2 Iterative Models in SQL Modeling

Using the ITERATE option of the MODEL clause, you can evaluate rules iteratively for a certain number of times, which you can specify as an argument to the ITERATE clause. ITERATE can be specified only for SEQUENTIAL ORDER models and such models are referred to as iterative models. For example, consider the following:

```
SELECT x, s FROM DUAL
MODEL
  DIMENSION BY (1 AS x) MEASURES (1024 AS s)
  RULES UPDATE ITERATE (4)
(s[1] = s[1]/2);
```

In Oracle, the table DUAL has only one row. Hence this model defines a 1-dimensional array, dimensioned by x with a measure s, with a single element s[1] = 1024. The rule s[1] = s[1]/2 evaluation will be repeated four times. The result of this query is a single row with values 1 and 64 for columns x and s respectively. The number of iterations arguments for the ITERATE clause should be a positive integer constant. Optionally, you can specify an early termination condition to stop rule evaluation before reaching the maximum iteration. This condition is specified in the UNTIL subclause of ITERATE and is checked at the end of an iteration. So, you will have at least one iteration when ITERATE is specified. The syntax of the ITERATE clause is:

```
ITERATE (number_of_iterations) [ UNTIL (condition) ]
```

Iterative evaluation stops either after finishing the specified number of iterations or when the termination condition evaluates to TRUE, whichever comes first.


Copyright © 2001, 2025, Oracle and/or its affiliates.

In some cases, you may want the termination condition to be based on the change, across iterations, in value of a cell. Oracle AI Database provides a mechanism to specify such conditions in that it enables you to access cell values as they existed before and after the current iteration in the `UNTIL` condition. Oracle's `PREVIOUS` function takes a single cell reference as an argument and returns the measure value of the cell as it existed after the previous iteration. You can also access the current iteration number by using the system variable `ITERATION_NUMBER`, which starts at value 0 and is incremented after each iteration. By using `PREVIOUS` and `ITERATION_NUMBER`, you can construct complex termination conditions.

Consider the following iterative model that specifies iteration over rules till the change in the value of `s[1]` across successive iterations falls below 1, up to a maximum of 1000 times:

```
SELECT x, s, iterations FROM DUAL
MODEL
  DIMENSION BY (1 AS x) MEASURES (1024 AS s, 0 AS iterations)
  RULES ITERATE (1000) UNTIL ABS(PREVIOUS(s[1]) - s[1]) < 1
 (s[1] = s[1]/2, iterations[1] = ITERATION_NUMBER);
```

The absolute value function (`ABS`) can be helpful for termination conditions because you may not know if the most recent value is positive or negative. Rules in this model will be iterated over 11 times as after 11th iteration the value of `s[1]` would be 0.5. This query results in a single row with values 1, 0.5, 10 for `x`, `s` and iterations respectively.

You can use the `PREVIOUS` function only in the `UNTIL` condition. However, `ITERATION_NUMBER` can be anywhere in the main model. In the following example, `ITERATION_NUMBER` is used in cell references:

```
SELECT country, product, year, sales
FROM sales_view
MODEL
  PARTITION BY (country) DIMENSION BY (product, year) MEASURES (sales sales)
  IGNORE NAV
  RULES ITERATE(3)
(sales['Tennis Racquet', 2018 + ITERATION_NUMBER] = sales['Tennis Racquet', 2016
  + ITERATION_NUMBER]);
```

This statement achieves an array copy of sales of `Tennis Racquet` from cells in the arrays previously created.

The view `sales_view` was created as described in Base Schema for SQL Modeling Examples.

## 23.3.3 Rule Dependency in AUTOMATIC ORDER Models

Oracle AI Database determines the order of evaluation of rules in an `AUTOMATIC ORDER` model based on their dependencies. A rule is evaluated only after the rules it depends on are evaluated. The algorithm chosen to evaluate the rules is based on the dependency analysis and whether rules in your model have circular (or cyclical) dependencies. A cyclic dependency can be of the form "rule A depends on B and rule B depends on A" or of the self-cyclic "rule depending on itself" form. An example of the former is as follows.

> **Note**
>
> All examples here and in previous sections are operations against the `Products` table in the `sh` sample schema provided by Oracle

.


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
sales['Wide Brim Hat', 2023] = 1.5 * sales['Wicket Keeper G', 2023],
sales['Wicket Keeper G', 2023] = 100000 / sales['Wide Brim Hat', 2023
```

An example of the latter is:

```
sales['Wide Brim Hat', 2023] = 25000 / sales['Wide Brim Hat', 2023]
```

However, there is no self-cycle in the following rule as different measures are being accessed on the left and right side:

```
projected_sales['Wide Brim Hat', 2023] = 25000 / sales['Wide Brim Hat', 2023]
```

When the analysis of an `AUTOMATIC ORDER` model finds that the rules have no circular dependencies, Oracle AI Database evaluates the rules in their dependency order. For example, in the following `AUTOMATIC ORDER` model:

```
MODEL DIMENSION BY (prod, year) MEASURES (sale sales) IGNORE NAV
  RULES AUTOMATIC ORDER
 (sales['SUV', 2021] = 10000,
  sales['Practice Tennis', 2021] = sales['Shin Guards', 2021]
    * 0.10 + sales['Boat', 2021] * 0.50,
  sales['Boat', 2021] = sales['Shin Guards', 2021]
    * 0.25 + sales['SUV', 2021]* 0.75,
  sales['Shin Guards', 2021] = 20000)
```

Rule 2 depends on rules 3 and 4, while rule 3 depends on rules 1 and 4, and rules 1 and 4 do not depend on any rule. Oracle, in this case, will find that the rule dependencies are acyclic and evaluate rules in one of the possible evaluation orders (1, 4, 3, 2) or (4, 1, 3, 2). This type of rule evaluation is called an `ACYCLIC` algorithm.

In some cases, Oracle AI Database may not be able to ascertain that your model is acyclic even though there is no cyclical dependency among the rules. This can happen if you have complex expressions in your cell references. Oracle AI Database assumes that the rules are cyclic and employs a `CYCLIC` algorithm that evaluates the model iteratively based on the rules and data. Iteration stops as soon as convergence is reached and the results are returned. Convergence is defined as the state in which further executions of the model will not change values of any of the cell in the model. Convergence is certain to be reached when there are no cyclical dependencies.

If your `AUTOMATIC ORDER` model has rules with cyclical dependencies, Oracle AI Database employs the earlier mentioned `CYCLIC` algorithm. Results are produced if convergence can be reached within the number of iterations Oracle is going to try the algorithm. Otherwise, Oracle reports a cycle detection error. You can circumvent this problem by manually ordering the rules and specifying `SEQUENTIAL ORDER`.

## 23.3.4 Ordered Rules in SQL Modeling

An ordered rule is one that has `ORDER BY` specified on the left side. It accesses cells in the order prescribed by `ORDER BY` and applies the right side computation. When you have `ANY` or symbolic references on the left side of a rule but without the `ORDER BY` clause, Oracle might return an error saying that the rule's results depend on the order in which cells are accessed and hence are non-deterministic. Consider the following `SEQUENTIAL ORDER` model. The examples here use the `sh` sample schema.

```
SELECT t, s
FROM sh.sales, sh.times
WHERE sales.time_id = times.time_id
GROUP BY calendar_year
MODEL
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
  DIMENSION BY (calendar_year t) MEASURES (SUM(amount_sold) s)
  RULES SEQUENTIAL ORDER
  (s[ANY] = s[CV(t)-1]);
```

This query attempts to set, for all years `t`, sales `s` value for a year to the sales value of the prior year. Unfortunately, the result of this rule depends on the order in which the cells are accessed. If cells are accessed in the ascending order of year, the result would be that of column 3 in Table 23-1. If they are accessed in descending order, the result would be that of column 4.

**Table 23-1 Ordered Rules**

| t | s | If ascending | If descending |
|---|---|---|---|
| 2018 | 1210000982 | null | null |
| 2019 | 1473757581 | null | 1210000982 |
| 2020 | 2376222384 | null | 1473757581 |
| 2021 | 1267107764 | null | 2376222384 |

If you want the cells to be considered in descending order and get the result given in column 4, you should specify:

```
SELECT t, s
FROM sh.sales, sh.times
WHERE sales.time_id = times.time_id
GROUP BY calendar_year
MODEL
  DIMENSION BY (calendar_year t) MEASURES (SUM(amount_sold) s)
  RULES SEQUENTIAL ORDER
  (s[ANY] ORDER BY t DESC = s[CV(t)-1]);
```

In general, you can use any `ORDER BY` specification as long as it produces a unique order among cells that match the left side cell reference. Expressions in the `ORDER BY` of a rule can involve constants, measures and dimension keys and you can specify the ordering options `[ASC | DESC] [NULLS FIRST | NULLS LAST]` to get the order you want.

You can also specify `ORDER BY` for rules in an `AUTOMATIC ORDER` model to make Oracle consider cells in a particular order during rule evaluation. Rules are never considered self-cyclic if they have `ORDER BY`. For example, to make the following `AUTOMATIC ORDER` model with a self-cyclic formula acyclic:

```
MODEL
  DIMENSION BY (calendar_year t) MEASURES (SUM(amount_sold) s)
  RULES AUTOMATIC ORDER
  (s[ANY] = s[CV(t)-1])
```

You must provide the order in which cells need to be accessed for evaluation using `ORDER BY`. For example, you can say:

```
s[ANY] ORDER BY t = s[CV(t) - 1]
```

Then Oracle AI Database picks an `ACYCLIC` algorithm (which is certain to produce the result) for formula evaluation.

## 23.3.5 Analytic Functions in SQL Modeling

Analytic functions (also known as window functions) can be used in the right side of rules. The ability to use analytic functions adds expressive power and flexibility to the `MODEL` clause.The following example combines an analytic function with the `MODEL` clause. First, you create a view


Copyright © 2001, 2025, Oracle and/or its affiliates.

sales_rollup_time that uses the GROUPING_ID function to calculate an identifier for different levels of aggregations. You then use the view in a query that calculates the cumulative sum of sales at both the quarter and year levels.

```
CREATE OR REPLACE VIEW sales_rollup_time
AS
SELECT country_name country, calendar_year year, calendar_quarter_desc quarter,
GROUPING_ID(calendar_year, calendar_quarter_desc) gid, SUM(amount_sold) sale,
COUNT(amount_sold) cnt
FROM sales, times, customers, countries
WHERE sales.time_id = times.time_id AND sales.cust_id = customers.cust_id
  AND customers.country_id = countries.country_id
GROUP BY country_name, calendar_year, ROLLUP(calendar_quarter_desc)
ORDER BY gid, country, year, quarter;

SELECT country, year, quarter, sale, csum
FROM sales_rollup_time
WHERE country IN ('United States of America', 'United Kingdom')
MODEL DIMENSION BY (country, year, quarter)
MEASURES (sale, gid, 0 csum)
(
csum[any, any, any] =
  SUM(sale) OVER (PARTITION BY country, DECODE(gid,0,year,null)
ORDER BY year, quarter
ROWS UNBOUNDED PRECEDING)
)
ORDER BY country, gid, year, quarter;

COUNTRY                             YEAR QUARTER       SALE       CSUM
------------------------------ ---------- ------- ---------- ----------
United Kingdom                      2018 2018-01  484733.96  484733.96
United Kingdom                      2018 2018-02  386899.15  871633.11
United Kingdom                      2018 2018-03  402296.49  1273929.6
United Kingdom                      2018 2018-04  384747.94 1658677.54
United Kingdom                      2019 2019-01  394911.91  394911.91
United Kingdom                      2019 2019-02  331068.38  725980.29
United Kingdom                      2019 2019-03  383982.61  1109962.9
United Kingdom                      2019 2019-04  398147.59 1508110.49
United Kingdom                      2020 2020-01  424771.96  424771.96
United Kingdom                      2020 2020-02  351400.62  776172.58
United Kingdom                      2020 2020-03  385137.68 1161310.26
United Kingdom                      2020 2020-04   390912.8 1552223.06
United Kingdom                      2021 2021-01  343468.77  343468.77
United Kingdom                      2021 2021-02  415168.32  758637.09
United Kingdom                      2021 2021-03  478237.29 1236874.38
United Kingdom                      2021 2021-04  437877.47 1674751.85
United Kingdom                      2018         1658677.54 1658677.54
United Kingdom                      2019         1508110.49 3166788.03
United Kingdom                      2020         1552223.06 4719011.09
United Kingdom                      2021         1674751.85 6393762.94
...  /*and similar output for the US*/
```

There are some specific restrictions when using analytic functions. See "Rules and Restrictions when Using SQL for Modeling" for more information.

## 23.3.6 Unique Dimensions Versus Unique Single References in SQL Modeling

The MODEL clause, in its default behavior, requires the PARTITION BY and DIMENSION BY keys to uniquely identify each row in the input to the model. Oracle verifies that and returns an error if


Copyright © 2001, 2025, Oracle and/or its affiliates.

the data is not unique. Uniqueness of the input rowset on the PARTITION BY and DIMENSION BY keys guarantees that any single cell reference accesses one and only one cell in the model. You can specify an optional UNIQUE DIMENSION keyword in the MODEL clause to make this behavior explicit. For example, the following query run on the view sales_view in the sh sample schema. This view is created as described in Base Schema for SQL Modeling Examples:

```
SELECT country, product, sales
FROM sales_view
WHERE country IN ('France', 'Poland')
MODEL UNIQUE DIMENSION
  PARTITION BY (country) DIMENSION BY (product) MEASURES (sales sales)
  IGNORE NAV RULES UPSERT
(sales['Right-Handed Gr'] = sales['All Products'] * 0.24);
```

This would return a uniqueness violation error as the rowset input to model is not unique on country and product because year is also needed:

```
ERROR at line 2:ORA-32638: Non unique addressing in MODEL dimensions
```

However, the following query does not return such an error:

```
SELECT country, product, year, sales
FROM sales_view
WHERE country IN ('Italy', 'Japan')
MODEL UNIQUE DIMENSION
  PARTITION BY (country) DIMENSION BY (product, year) MEASURES (sales sales)
  RULES UPSERT
(sales['Right-Handed Gr', 2023] = sales['All Products', 2022] * 0.24);
```

Input to the MODEL clause in this case is unique on country, product, and year as shown in:

```
COUNTRY    PRODUCT                                    YEAR       SALES
__________ _________________________________________ _______ ___________
Italy      11" Youth Field Master Glove                2019     2070.22
Italy      11" Youth Field Master Glove                2020     4360.09
Italy      11" Youth Field Master Glove                2021        1276
Italy      11" Youth Field Master Glove                2022     3186.55
Italy      11.5" Youth Triple Stripe Series Glove      2019     4896.79
Italy      11.5" Youth Triple Stripe Series Glove      2020    11063.49
Italy      11.5" Youth Triple Stripe Series Glove      2021     2841.13
Italy      11.5" Youth Triple Stripe Series Glove      2022     5208.63
Italy      12" Premium Series Glove                    2019     5871.16
Italy      12" Premium Series Glove                    2020     12219.3
...
```

If you want to relax this uniqueness checking, you can specify UNIQUE SINGLE REFERENCE keyword. This can save processing time. In this case, the MODEL clause checks the uniqueness of only the single cell references appearing on the right side of rules. So the query that returned the uniqueness violation error would be successful if you specify UNIQUE SINGLE REFERENCE instead of UNIQUE DIMENSION.

Another difference between UNIQUE DIMENSION and UNIQUE SINGLE REFERENCE semantics is the number of cells that can be updated by a rule with a single cell reference on left side. In the case of UNIQUE DIMENSION, such a rule can update at most one row as only one cell would match the single cell reference on the left side. This is because the input rowset would be unique on PARTITION BY and DIMENSION BY keys. With UNIQUE SINGLE REFERENCE, all cells that match the left side single cell reference would be updated by the rule.


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 23.3.7 Rules and Restrictions when Using SQL for Modeling

The following general rules and restrictions apply when using the MODEL clause:

- The only columns that can be updated are the columns specified in the MEASURES subclause of the main SQL model. Measures of reference models cannot be updated.
- The MODEL clause is evaluated after all clauses in the query block except SELECT DISTINCT, and ORDER BY clause are evaluated. These clauses and expressions in the SELECT list are evaluated after the MODEL clause.
- If your query has a MODEL clause, then the query's SELECT and ORDER BY lists cannot contain aggregates or analytic functions. If needed, these can be specified in PARTITION BY, DIMENSION BY, and MEASURES lists and need to be aliased. Aliases can then be used in the SELECT or ORDER BY clauses. In the following example, the analytic function RANK is specified and aliased in the MEASURES list of the MODEL clause, and its alias is used in the SELECT list so that the outer query can order resulting rows based on their ranks.

```
SELECT country, product, year, s, RNK
FROM (SELECT country, product, year, s, rnk
      FROM sales_view
      MODEL
        PARTITION BY (country) DIMENSION BY (product, year)
        MEASURES (sales s, year y, RANK() OVER (ORDER BY sales) rnk)
        RULES UPSERT
          (s['5 Point Batting Tee Increase 90-99', 2021] =
             REGR_SLOPE(s, y) ['5 Point Batting Tee', year BETWEEN 1990 AND 2000],
           s['5 Point Batting Tee', 2021] = s['5 Point Batting Tee', 2000] *
          (1+s['5 Point Batting Tee increase 90-99', 2021])))
WHERE product <> '5 Point Batting Tee Increase 90-99'
ORDER BY country, year, rnk, product;
```

- When there is a multi-cell reference on the right hand side of a rule, you need to apply a function to aggregate the measure values of multiple cells referenced into a single value. You can use any kind of aggregate function for this purpose: regular, analytic aggregate (inverse percentile, hypothetical rank and distribution), or user-defined aggregate.
- Only rules with positional single cell references on the left side have UPSERT semantics. All other rules have UPDATE semantics, even when you specify the UPSERT option for them.
- Negative increments are not allowed in FOR loops. Also, no empty FOR loops are allowed. FOR d FROM 2005 TO 2021 INCREMENT -1 is not permitted. You should use FOR d FROM 2005 TO 2021 DECREMENT 1 instead. FOR d FROM 2005 TO 2021 INCREMENT 1 is not permitted as it designates an empty loop.
- You cannot use nested query expressions (subqueries) in rules except in the FOR construct. For example, it would be not permitted to issue the following:

```
SELECT *
FROM sales_view WHERE country = 'Poland'
MODEL DIMENSION BY (product, year)
  MEASURES (sales sales)
  RULES UPSERT
  (sales['5 Point Batting Tee', 2023] = sales['5 Point Batting Tee', 2022] +
   (SELECT SUM(sales) FROM sales_view));
```

This is because the rule has a subquery on its right side. Instead, you can rewrite the preceding query in the following legal way:

```
SELECT *
FROM sales_view WHERE country = 'Poland'
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
MODEL DIMENSION BY (product, year)
  MEASURES (sales sales, (SELECT SUM(sales) FROM sales_view) AS grand_total)
  RULES UPSERT
  (sales['5 Point Batting Tee', 2023] =sales['5 Point Batting Tee', 2022] +
   grand_total['5 Point Batting Tee', 2022]);
```

- You can also use subqueries in the `FOR` construct specified on the left side of a rule. However, they:
  - Cannot be correlated
  - Must return fewer than 10,000 rows
  - Cannot be a query defined in the `WITH` clause
  - Will make the cursor unsharable

Nested cell references have the following restrictions:

- Nested cell references must be single cell references. Aggregates on nested cell references are not supported. So, it would be not permitted to say `s['5 Point Batting Tee', MAX(best_year)['5 Point Batting Tee', ANY]]`.
- Only one level of nesting is supported for nested cell references on the main model. So, for example, `s['5 Point Batting Tee', best_year['5 Point Batting Tee', 2021]]` is legal, but `s['5 Point Batting Tee', best_year['5 Point Batting Tee', best_year['5 Point Batting Tee', 2021]]]` is not.
- Nested cell references appearing on the left side of rules in an `AUTOMATIC ORDER` model should not be updated in any rule of the model. This restriction ensures that the rule dependency relationships do not arbitrarily change (and hence cause non-deterministic results) due to updates to reference measures.

  There is no such restriction on nested cell references in a `SEQUENTIAL ORDER` model. Also, this restriction is not applicable on nested references appearing on the right side of rules in both `SEQUENTIAL` or `AUTOMATIC ORDER` models.

Reference models have the following restrictions:

- The query defining the reference model cannot be correlated to any outer query. It can, however, be a query with subqueries, views, and so on.
- Reference models cannot have a `PARTITION BY` clause.
- Reference models cannot be updated.

Window functions have the following restrictions:

- The expressions in the `OVER` clause can be expressions of constants, measures, keys from `PARTITION BY` and `DIMENSION BY` of the `MODEL` clause, and single cell expressions. Aggregates are not permitted inside the `OVER` clause. Therefore, the following is okay:

  ```
  rnk[ANY, ANY, ANY] = RANK() OVER (PARTITION BY prod, country ORDER BY sale)
  ```

  While the following is not:

  ```
  rnk[ANY, ANY, ANY] = RANK() OVER (PARTITION BY prod, country ORDER BY SUM(sale))
  ```

- Rules with window functions on their right side cannot have an `ORDER BY` clause on their left side.
- Window functions and aggregate functions cannot both be on the right side of a rule.
- Window functions can only be used on the right side of an `UPDATE` rule.
- If a rule has a `FOR` loop on its left side, a window function cannot be used on the right side of the rule.


Copyright © 2001, 2025, Oracle and/or its affiliates.

# 23.4 Performance Considerations with SQL Modeling

The following sections describe topics that affect performance when using the MODEL clause:

- Parallel Execution and SQL Modeling
- Aggregate Computation and SQL Modeling
- Using EXPLAIN PLAN to Understand Model Queries

## 23.4.1 Parallel Execution and SQL Modeling

MODEL clause computation is scalable in terms of the number of processors you have. Scalability is achieved by performing the MODEL computation in parallel across the partitions defined by the PARTITION BY clause. Data is distributed among processing elements based on the PARTITION BY key values such that all rows with the same values for the PARTITION BY keys will go to the same processing element. Note that the internal processing of partitions will not create a one-to-one match of logical and internally processed partitions. This way, each processing element can finish MODEL clause computation independent of other elements. The data partitioning can be hash based or range based. Consider the following MODEL clause:

```
MODEL
  PARTITION BY (country) DIMENSION BY (product, time) MEASURES (sales)
  RULES UPDATE
  (sales['12.75" Premium Series Glove', 2022] = 1.2 * sales['12.75" Premium Series
Glove', 2001],
   sales['11" Youth Field Master Glove', 2022] = 0.8 * sales['11" Youth Field Master
Glove', 2001])
```

Here input data will be partitioned among processing elements based on the PARTITION BY key country and this partitioning can be hash or range based. Each processing element will evaluate the rules on the data it receives.

Parallelism of the model computation is governed or limited by the way you specify the MODEL clause. If your MODEL clause has no PARTITION BY keys, then the computation cannot be parallelized (with exceptions mentioned in the following). If PARTITION BY keys have very low 11" Youth Field Master Gloved cardinality, then the degree of parallelism will be limited. In such cases, Oracle identifies the DIMENSION BY keys that can used for partitioning. For example, consider a MODEL clause equivalent to the preceding one, but without PARTITION BY keys as in the following:

```
MODEL
  DIMENSION BY (country, product, time) MEASURES (sales)
  RULES UPDATE
  (sales[ANY, '12.75" Premium Series Glove', 2022] = 1.2 * sales[CV(country), '12.75"
Premium Series Glove', 2001],
   sales[ANY, '11" Youth Field Master Glove', 2022] = 0.8 * sales[CV(country), '11"
Youth Field Master Glove', 2001])
```

In this case, Oracle AI Database identifies that it can use the DIMENSION BY key country for partitioning and uses region as the basis of internal partitioning. It partitions the data among processing elements on country and thus effects parallel execution.

## 23.4.2 Aggregate Computation and SQL Modeling

The MODEL clause processes aggregates in two different ways: first, the regular fashion in which data in the partition is scanned and aggregated and second, an efficient window style


Copyright © 2001, 2025, Oracle and/or its affiliates.

aggregation. The first type as illustrated in the following introduces a new dimension member ALL_2022_products and computes its value to be the sum of year 2022 sales for all products:

```
MODEL PARTITION BY (country) DIMENSION BY (product, time) MEASURES (sale sales)
RULES UPSERT
 (sales['ALL_2022_products', 2022] = SUM(sales)[ANY, 2022])
```

To evaluate the aggregate sum in this case, each partition will be scanned to find the cells for 2022 for all products and they will be aggregated. If the left side of the rule were to reference multiple cells, then Oracle will have to compute the right side aggregate by scanning the partition for each cell referenced on the left. For example, consider the following example:

```
MODEL PARTITION BY (country) DIMENSION BY (product, time)
  MEASURES (sale sales, 0 avg_exclusive)
  RULES UPDATE
  (avg_exclusive[ANY, 2022] =  AVG(sales)[product <> CV(product), CV(time)])
```

This rule calculates a measure called `avg_exclusive` for every product in 2022. The measure `avg_exclusive` is defined as the average sales of all products excluding the current product. In this case, Oracle scans the data in a partition for every product in 2022 to calculate the aggregate, and this may be expensive.

Oracle AI Database optimizes the evaluation of such aggregates in some scenarios with window-style computation as used in analytic functions. These scenarios involve rules with multi-cell references on their left side and computing window computations such as moving averages, cumulative sums and so on. Consider the following example:

```
MODEL PARTITION BY (country) DIMENSION BY (product, time)
  MEASURES (sale sales, 0 mavg)
  RULES UPDATE
  (mavg[product IN (5 Point Batting Tee', 'Team shirt', 'Spiked Shoes'), ANY] =
   AVG(sales)[CV(product), time BETWEEN CV(time)
   AND CV(time) - 2])
```

It computes the moving average of sales for products `5 Point Batting Tee`, `Team shirt`, and `Spiked Shoes` over a three year period. It would be very inefficient to evaluate the aggregate by scanning the partition for every cell referenced on the left side. Oracle identifies the computation as being in window-style and evaluates it efficiently. It sorts the input on product, time and then scans the data once to compute the moving average. You can view this rule as an analytic function being applied on the sales data for products `5 Point Batting Tee`, `Team shirt`, and `Spiked Shoes`:

```
AVG(sales) OVER (PARTITION BY product ORDER BY time
RANGE BETWEEN 2 PRECEDING AND CURRENT ROW)
```

This computation style is called `WINDOW (IN MODEL) SORT`. This style of aggregation is applicable when the rule has a multi-cell reference on its left side with no `ORDER BY`, has a simple aggregate (`SUM`, `COUNT`, `MIN`, `MAX`, `STDEV`, and `VAR`) on its right side, only one dimension on the right side has a boolean predicate (<, <=, >, >=, `BETWEEN`), and all other dimensions on the right are qualified with `CV`.

## 23.4.3 Using EXPLAIN PLAN to Understand Model Queries

Oracle's explain plan facility is fully aware of models. You will see a line in your query's main explain plan output showing the model and the algorithm used. Reference models are tagged with the keyword `REFERENCE` in the plan output. Also, Oracle annotates the plan with `WINDOW (IN MODEL) SORT` if any of the rules qualify for window-style aggregate computation.


Copyright © 2001, 2025, Oracle and/or its affiliates.

By examining an explain plan, you can find out the algorithm chosen to evaluate your model. If your model has SEQUENTIAL ORDER semantics, then ORDERED is displayed. For AUTOMATIC ORDER models, Oracle displays ACYCLIC or CYCLIC based on whether it chooses ACYCLIC or CYCLIC algorithm for evaluation. In addition, the plan output will have an annotation FAST in case of ORDERED and ACYCLIC algorithms if all left side cell references are single cell references and aggregates, if any, on the right side of rules are simple arithmetic non-distinct aggregates like SUM, COUNT, AVG, and so on. Rule evaluation in this case would be highly efficient and hence the annotation FAST. Thus, the output you will see in the explain plan would be MODEL {ORDERED [FAST] | ACYCLIC [FAST] | CYCLIC}.

This section contains the following topics:

- Using ORDERED FAST: Example
- Using ORDERED: Example
- Using ACYCLIC FAST: Example
- Using ACYCLIC: Example
- Using CYCLIC: Example

### Using ORDERED FAST: Example

This model has only single cell references on the left side of rules and the aggregate AVG on the right side of first rule is a simple non-distinct aggregate:

```
EXPLAIN PLAN FOR
SELECT country, product, year, sales
FROM sales_view
WHERE country IN ('Italy', 'Japan')
MODEL UNIQUE DIMENSION
  PARTITION BY (country) DIMENSION BY (product, year) MEASURES (sales sales)
  RULES UPSERT
  (sales['Bounce', 2023] = AVG(sales)[ANY, 2022] * 1.24,
   sales['Y Box', 2023] = sales['Bounce', 2023] * 0.25);
```

### Using ORDERED: Example

Because the left side of the second rule is a multi-cell reference, the FAST method will not be chosen in the following:

```
EXPLAIN PLAN FOR
SELECT country, product, year, sales
FROM sales_view
WHERE country IN ('Italy', 'Japan')
MODEL UNIQUE DIMENSION
  PARTITION BY (country) DIMENSION BY (product, year) MEASURES (sales sales)
  RULES UPSERT
  (sales['Bounce', 2023] = AVG(sales)[ANY, 2022] * 1.24,
   sales[prod <> 'Bounce', 2023] = sales['Bounce', 2023] * 0.25);
```

### Using ACYCLIC FAST: Example

Rules in this model are not cyclic and the explain plan will show ACYCLIC. The FAST method is chosen in this case as well.

```
EXPLAIN PLAN FOR
SELECT country, product, year, sales
FROM sales_view
WHERE country IN ('Italy', 'Japan')
MODEL UNIQUE DIMENSION
  PARTITION BY (country) DIMENSION BY (product, year) MEASURES (sales sales)
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
  RULES UPSERT AUTOMATIC ORDER
  (sales['Y Box', 2023] = sales['Bounce', 2023] * 0.25,
   sales['Bounce', 2023] = sales['Bounce', 2022] / SUM(sales)[ANY, 2022] * 2 *
   sales['All Products', 2023],
   sales['All Products', 2023] = 200000);
```

### Using ACYCLIC: Example

Rules in this model are not cyclic. The `PERCENTILE_DISC` aggregate that gives the median sales for year 2022, in the second rule is not a simple aggregate function. Therefore, Oracle will not choose the `FAST` method, and the explain plan will just show `ACYCLIC`.

```
SELECT country, product, year, sales
FROM sales_view
WHERE country IN ('Italy', 'Japan')
MODEL UNIQUE DIMENSION
  PARTITION BY (country) DIMENSION BY (product, year) MEASURES (sales sales)
  RULES UPSERT AUTOMATIC ORDER
  (sales['Y Box', 2023] = sales['Bounce', 2023] * 0.25,
   sales['Bounce',2023] = PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY
   sales)[ANY,2022] / SUM(sales)[ANY, 2022] * 2 * sales['All Products', 2023],
   sales['All Products', 2023] = 200000);
```

### Using CYCLIC: Example

Oracle chooses `CYCLIC` algorithm for this model as there is a cycle among second and third rules.

```
EXPLAIN PLAN FOR
SELECT country, product, year, sales
FROM sales_view
WHERE country IN ('Italy', 'Japan')
MODEL UNIQUE DIMENSION
  PARTITION BY (country) DIMENSION BY (product, year) MEASURES (sales sales)
  IGNORE NAV RULES UPSERT AUTOMATIC ORDER
  (sales['All Products', 2023] = 200000,
   sales['Y Box', 2023] = sales['Bounce', 2023] * 0.25,
   sales['Bounce', 2023] = sales['Y Box', 2023] +
    (sales['Bounce', 2022] / SUM(sales)[ANY, 2022] * 2 *
     sales['All Products', 2023]));
```

## 23.5 Examples of SQL Modeling

The examples in this section assume that in addition to `sales_view` (created in Base Schema for SQL Modeling Examples), you have the following view defined. It finds monthly totals of sales and quantities by product and country.

```
CREATE VIEW sales_view2 AS
SELECT country_name country, prod_name product, calendar_year year,
  calendar_month_name month, SUM(amount_sold) sale, COUNT(amount_sold) cnt
FROM sh.sales, sh.times, sh.customers, sh.countries, sh.products
WHERE sales.time_id = times.time_id AND
      sales.prod_id = products.prod_id AND
      sales.cust_id = customers.cust_id AND
      customers.country_id = countries.country_id
GROUP BY country_name, prod_name, calendar_year, calendar_month_name;
```

This section contains the following examples:

- SQL Modeling Example 1: Calculating Sales Differences


Copyright © 2001, 2025, Oracle and/or its affiliates.

- SQL Modeling Example 2: Calculating Percentage Change
- SQL Modeling Example 3: Calculating Net Present Value
- SQL Modeling Example 4: Calculating Using Simultaneous Equations
- SQL Modeling Example 5: Calculating Using Regression
- SQL Modeling Example 6: Calculating Mortgage Amortization

## 23.5.1 SQL Modeling Example 1: Calculating Sales Differences

Show the sales for Italy and Spain and the difference between the two for each product. The difference should be placed in a new row with `country = 'Diff Italy-Spain'`.

```
SELECT product, country, sales
FROM sales_view
WHERE country IN ('Italy', 'Spain')
GROUP BY product, country
MODEL
  PARTITION BY (product) DIMENSION BY (country) MEASURES (SUM(sales) AS sales)
  RULES UPSERT
  (sales['DIFF ITALY-SPAIN'] = sales['Italy'] - sales['Spain']);
```

See "Examples of SQL Modeling" for information about the views required to run this example.

## 23.5.2 SQL Modeling Example 2: Calculating Percentage Change

If sales for each product in each country grew (or declined) at the same monthly rate from November 2000 to December 2000 as they did from October 2000 to November 2000, what would the fourth quarter's sales be for the whole company and for each country?

```
SELECT country, SUM(sales)
FROM (SELECT product, country, month, sales
      FROM sales_view2
      WHERE year=2020 AND month IN ('October','November')
MODEL
  PARTITION BY (product, country) DIMENSION BY (month) MEASURES (sale sales)
  RULES
   (sales['December']=(sales['November'] /sales['October']) *sales['November']))
GROUP BY GROUPING SETS ((),(country));
```

See "Examples of SQL Modeling" for information about the views required to run this example.

## 23.5.3 SQL Modeling Example 3: Calculating Net Present Value

You want to calculate the net present value (NPV) of a series of periodic cash flows. Your scenario involves two projects, each of which starts with an initial investment at time 0, represented as a negative cash flow. The initial investment is followed by three years of positive cash flow. First, create a table (`cash_flow`) and populate it with some data, as in the following statements:

```
CREATE TABLE cash_flow (year DATE, i INTEGER, prod VARCHAR2(3), amount NUMBER);
INSERT INTO cash_flow VALUES (TO_DATE('2019', 'YYYY'),  0,  'vcr',  -100.00);
INSERT INTO cash_flow VALUES (TO_DATE('2020', 'YYYY'),  1,  'vcr',   12.00);
INSERT INTO cash_flow VALUES (TO_DATE('2021', 'YYYY'),  2,  'vcr',  10.00);
INSERT INTO cash_flow VALUES (TO_DATE('2022', 'YYYY'),  3,  'vcr',  20.00);
INSERT INTO cash_flow VALUES (TO_DATE('2019', 'YYYY'),  0,  'dvd',  -200.00);
INSERT INTO cash_flow VALUES (TO_DATE('2020', 'YYYY'),  1,  'dvd',  22.00);
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
INSERT INTO cash_flow VALUES (TO_DATE('2021', 'YYYY'),  2,  'dvd',  12.00);
INSERT INTO cash_flow VALUES (TO_DATE('2022', 'YYYY'),  3,  'dvd',  14.00);
```

See "Examples of SQL Modeling" for information about the views required to run this example.

To calculate the NPV using a discount rate of 0.14, issue the following statement:

```
SELECT year, i, prod, amount, npv
FROM cash_flow
MODEL PARTITION BY (prod)
  DIMENSION BY (i)
  MEASURES (amount, 0 npv, year)
  RULES
    (npv[0] = amount[0],
     npv[i !=0] ORDER BY i =
       amount[CV()]/ POWER(1.14,CV(i)) + npv[CV(i)-1]);
```

```
YEAR            I PROD       AMOUNT                                                NPV
_____________ ____ _______ _________ ____________________________________________________
01-JAN-19       0 dvd          -200                                                -200
01-JAN-20       1 dvd            22     -180.70175438596491228070175438596491228 1
01-JAN-21       2 dvd            12     -171.46814404432132963988919667590027700 9
01-JAN-22       3 dvd            14      -162.01854281749310179110441539367038711
01-JAN-19       0 vcr          -100                                                -100
01-JAN-20       1 vcr            12    -89.473684210526315789473684210526315789 47
01-JAN-21       2 vcr            10    -81.779008925823330255463219452139119729 14
01-JAN-22       3 vcr            20    -68.279578601783004757199246191810705588 22
```

## 23.5.4 SQL Modeling Example 4: Calculating Using Simultaneous Equations

You want your interest expenses to equal 30% of your net income (net=pay minus tax minus interest). Interest is tax deductible from gross, and taxes are 38% of salary and 28% capital gains. You have salary of $100,000 and capital gains of $15,000. Net income, taxes, and interest expenses are unknown. Observe that this is a simultaneous equation (net depends on interest, which depends on net), thus the `ITERATE` clause is included.

See "Examples of SQL Modeling" for information about the views required to run this example.

First, create a table called `ledger`:

```
CREATE TABLE  ledger  (account  VARCHAR2(20), balance  NUMBER(10,2) );
```

Then, insert the following five rows:

```
INSERT INTO ledger VALUES  ('Salary', 100000);
INSERT INTO ledger VALUES  ('Capital_gains', 15000);
INSERT INTO ledger VALUES  ('Net', 0);
INSERT INTO ledger VALUES  ('Tax', 0);
INSERT INTO ledger VALUES  ('Interest', 0);
```

Next, issue the following statement:

```
SELECT s, account
FROM ledger
MODEL
  DIMENSION  BY (account) MEASURES (balance s)
  RULES  ITERATE (100)
  (s['Net']=s['Salary']-s['Interest']-s['Tax'],
 s['Tax']=(s['Salary']-s['Interest'])*0.38 + s['Capital_gains']*0.28,
  s['Interest']=s['Net']*0.30);
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

The output (with numbers rounded) is:

```
                                        S ACCOUNT
_____________________________________ _________________
                                100000 Salary
                                 15000 Capital_gains
                          48735.24451 Net
                          36644.18212 Tax
                          14620.57335 Interest
```

## 23.5.5 SQL Modeling Example 5: Calculating Using Regression

The sales of the product Helmet in 2021 will increase in comparison to 2020 as they did in the previous three years (between 2017 and 2019). To calculate the increase, use the regression function `REGR_SLOPE` as follows. Because you are calculating the next period's value, it is sufficient to add the slope to the 2020 value.

```
SELECT * FROM
 (SELECT country, product, year, projected_sale, sales
  FROM sales_view
  WHERE country IN ('Italy', 'Japan') AND product IN ('Helmet')
MODEL
  PARTITION BY (country) DIMENSION BY (product, year)
  MEASURES (sales sales, year y, CAST(NULL AS NUMBER) projected_sale) IGNORE NAV
  RULES UPSERT
  (projected_sale[FOR product IN ('Helmet'), 2021] =
             sales[CV(), 2020] +
             REGR_SLOPE(sales, y)[CV(), year BETWEEN 2019 AND 2020]))
ORDER BY country, product, year;
```

See "Examples of SQL Modeling" for information about the views required to run this example.

The output is as follows:

```
COUNTRY    PRODUCT       YEAR    PROJECTED_SALE        SALES
__________ __________ _______ _________________ ____________
Italy      Helmet        2019                        7421.58
Italy      Helmet        2020                        9405.81
Italy      Helmet        2021          11390.04      7804.42
Italy      Helmet        2022                       13115.52
Japan      Helmet        2019                       10226.85
Japan      Helmet        2020                       13827.28
Japan      Helmet        2021          17427.71      11335.5
Japan      Helmet        2022                       18768.84
```

## 23.5.6 SQL Modeling Example 6: Calculating Mortgage Amortization

This example creates mortgage amortization tables for any number of customers, using information about mortgage loans selected from a table of mortgage facts. First, create two tables and insert needed data:

- `mortgage_facts`

  Holds information about individual customer loans, including the name of the customer, the fact about the loan that is stored in that row, and the value of that fact. The facts stored for this example are loan (`Loan`), annual interest rate (`Annual_Interest`), and number of payments (`Payments`) for the loan. Also, the values for two customers, Smith and Jones, are inserted.


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
CREATE TABLE mortgage_facts (customer VARCHAR2(20), fact VARCHAR2(20),
  amount  NUMBER(10,2));
INSERT INTO mortgage_facts  VALUES ('Smith', 'Loan', 100000);
INSERT INTO mortgage_facts  VALUES ('Smith', 'Annual_Interest', 12);
INSERT INTO mortgage_facts  VALUES ('Smith', 'Payments', 360);
INSERT INTO mortgage_facts  VALUES ('Smith', 'Payment', 0);
INSERT INTO mortgage_facts  VALUES ('Jones', 'Loan', 200000);
INSERT INTO mortgage_facts  VALUES ('Jones', 'Annual_Interest', 12);
INSERT INTO mortgage_facts  VALUES ('Jones', 'Payments', 180);
INSERT INTO mortgage_facts  VALUES ('Jones', 'Payment', 0);
```

- `mortgage`

  Holds output information for the calculations. The columns are customer, payment number (`pmt_num`), principal applied in that payment (`principalp`), interest applied in that payment (`interestp`), and remaining loan balance (`mort_balance`). In order to upsert new cells into a partition, you need to have at least one row pre-existing per partition. Therefore, you seed the mortgage table with the values for the two customers before they have made any payments. This seed information could be easily generated using a SQL `INSERT` statement based on the `mortgage_facts` table.

```
CREATE TABLE mortgage_facts (customer VARCHAR2(20), fact VARCHAR2(20),
   amount  NUMBER(10,2));

INSERT INTO mortgage_facts  VALUES ('Smith', 'Loan', 100000);
INSERT INTO mortgage_facts  VALUES ('Smith', 'Annual_Interest', 12);
INSERT INTO mortgage_facts  VALUES ('Smith', 'Payments', 360);
INSERT INTO mortgage_facts  VALUES ('Smith', 'Payment', 0);
INSERT INTO mortgage_facts  VALUES ('Smith', 'PaymentAmt', null);
INSERT INTO mortgage_facts  VALUES ('Jones', 'Loan', 200000);
INSERT INTO mortgage_facts  VALUES ('Jones', 'Annual_Interest', 12);
INSERT INTO mortgage_facts  VALUES ('Jones', 'Payments', 180);
INSERT INTO mortgage_facts  VALUES ('Jones', 'Payment', 0);
INSERT INTO mortgage_facts  VALUES ('Jones', 'PaymentAmt', null);


CREATE TABLE mortgage (customer VARCHAR2(20), pmt_num NUMBER(4),
   principalp NUMBER(10,2), interestp NUMBER(10,2), mort_balance NUMBER(10,2));

INSERT INTO mortgage VALUES ('Jones',0, 0, 0, 200000);
INSERT INTO mortgage VALUES ('Smith',0, 0, 0, 100000);
```

See "Examples of SQL Modeling" for information about the views required to run this example.

The following SQL statement is complex, so individual lines have been annotated as needed. These lines are explained in more detail later.

```
SELECT c, p, m, pp, ip
FROM MORTGAGE
MODEL                                                                --See 1
REFERENCE R ON
   (SELECT customer, fact, amt                                       --See 2
    FROM mortgage_facts
    MODEL DIMENSION BY (customer, fact) MEASURES (amount amt)        --See 3
      RULES
       (amt[any, 'PaymentAmt']= (amt[CV(),'Loan']*
          Power(1+ (amt[CV(),'Annual_Interest']/100/12),
                   amt[CV(),'Payments']) *
         (amt[CV(),'Annual_Interest']/100/12)) /
         (Power(1+(amt[CV(),'Annual_Interest']/100/12),
                amt[CV(),'Payments']) - 1)
       )
    )
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
  DIMENSION BY (customer cust, fact) measures (amt)                    --See 4
MAIN amortization
  PARTITION BY (customer c)                                            --See 5
  DIMENSION BY (0 p)                                                   --See 6
  MEASURES (principalp pp, interestp ip, mort_balance m, customer mc)  --See 7
  RULES
    ITERATE(1000) UNTIL (ITERATION_NUMBER+1 =
r.amt[mc[0],'Payments'])                                               --See 8
    (ip[ITERATION_NUMBER+1] = m[CV()-1] *
       r.amt[mc[0], 'Annual_Interest']/1200,                           --See 9
     pp[ITERATION_NUMBER+1] = r.amt[mc[0], 'PaymentAmt'] - ip[CV()],   --See 10
      m[ITERATION_NUMBER+1] = m[CV()-1] - pp[CV()]                     --See 11
    )
ORDER BY c, p;
```

The following numbers refer to the numbers listed in the example:

1: This is the start of the main model definition.

2 through 4: These lines mark the start and end of the reference model labeled `R`. This model defines a `SELECT` statement that calculates the monthly payment amount for each customer's loan. The `SELECT` statement uses its own `MODEL` clause starting at the line labeled 3 with a single rule that defines the `amt` value based on information from the `mortgage_facts` table. The measure returned by reference model `R` is `amt`, dimensioned by customer name `cust` and fact value `fact` as defined in the line labeled 4.

The reference model is computed once and the values are then used in the main model for computing other calculations. Reference model `R` will return a row for each existing row of `mortgage_facts`, and it will return the newly calculated rows for each customer where the fact type is `Payment` and the `amt` is the monthly payment amount. If you wish to use a specific amount from the `R` output, you address it with the expression `r.amt[<customer_name>,<fact_name>]`.

5: This is the continuation of the main model definition. You will partition the output by customer, aliased as `c`.

6: The main model is dimensioned with a constant value of 0, aliased as `p`. This represents the payment number of a row.

7: Four measures are defined: `principalp (pp)` is the principal amount applied to the loan in the month, `interestp (ip)` is the interest paid that month, `mort_balance (m)` is the remaining mortgage value after the payment of the loan, and `customer (mc)` is used to support the partitioning.

8: This begins the rules block. It will perform the rule calculations up to 1000 times. Because the calculations are performed once for each month for each customer, the maximum number of months that can be specified for a loan is 1000. Iteration is stopped when the `ITERATION_NUMBER+1` equals the amount of payments derived from reference `R`. Note that the value from reference `R` is the `amt` (amount) measure defined in the reference clause. This reference value is addressed as `r.amt[<customer_name>,<fact>]`. The expression used in the iterate line, `"r.amt[mc[0], 'Payments']"` is resolved to be the amount from reference `R`, where the customer name is the value resolved by `mc[0]`. Because each partition contains only one customer, `mc[0]` can have only one value. Thus `"r.amt[mc[0], 'Payments']"` yields the reference clause's value for the number of payments for the current customer. This means that the rules will be performed as many times as there are payments for that customer.

9 through 11: The first two rules in this block use the same type of `r.amt` reference that was explained in 8. The difference is that the ip rule defines the fact value as `Annual_Interest`. Note that each rule refers to the value of one of the other measures. The expression used on


Copyright © 2001, 2025, Oracle and/or its affiliates.

the left side of each rule, `"[ITERATION_NUMBER+1]"` will create a new dimension value, so the measure will be upserted into the result set. Thus the result will include a monthly amortization row for all payments for each customer.

The final line of the example sorts the results by customer and loan payment number.


Copyright © 2001, 2025, Oracle and/or its affiliates.

24

# Advanced Analytical SQL

This chapter illustrates techniques for handling advanced business intelligence queries. We hope to enhance your understanding of how different SQL features can be used together to perform demanding analyses. Although the features shown here have been addressed on an individual basis in SQL for Aggregation in Data Warehouses, SQL for Analysis and Reporting, and SQL for Modeling, seeing features one at a time gives only a limited sense of how they can work together. Here we show the analytic power available when the features are combined.

What makes a business intelligence query "advanced"? The label is best applied to multistep queries, often involving dimension hierarchies. In such queries, the final result depends on several sets of retrieved data, multiple calculation steps, and the data retrieved may involve multiple levels of a dimension hierarchy. Prime examples of advanced queries are market share calculations based on multiple conditions and sales projections that require filling gaps in data.

The examples in this chapter illustrate using nested inline views, `CASE` expressions, partitioned outer join, the `MODEL` and `WITH` clauses, analytic SQL functions, and more. Where relevant to the discussion, query plans will be discussed. This chapter includes:

- Examples of Business Intelligence Queries

## 24.1 Examples of Business Intelligence Queries

The queries in this chapter illustrate various business intelligence tasks. The topics of the queries and the features used in each query are:

- Percent change in market share based on complex multistep conditions. It illustrates nested inline views, `CASE` expression, and analytic SQL functions.

  See "Business Intelligence Query Example 1: Percent Change in Market Share of Products in a Calculated Set"

- Sales projection with gaps in data filled in. It illustrates the `MODEL` clause together with partitioned outer join and the `CASE` expression.

  See "Business Intelligence Query Example 2: Sales Projection that Fills in Missing Data"

- Customer analysis grouping customers into purchase-size buckets. It illustrates the `WITH` clause (query subfactoring) and the analytic SQL functions `percentile_cont` and `width_bucket`.

  See "Business Intelligence Query Example 3: Customer Analysis by Grouping Customers into Buckets"

- Customer item grouping into itemsets. It illustrates calculating frequent itemsets using `DBMS_FREQUENT_ITEMSET.FI_TRANSACTIONAL` as a table function.

  See "Business Intelligence Query Example 4: Frequent Itemsets"


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 24.1.1 Business Intelligence Query Example 1: Percent Change in Market Share of Products in a Calculated Set

What was the percent change in market share for a grouping of my top 20% of products for the current three-month period versus same period year ago for accounts that grew by more than 20 percent in revenue?

We define market share as a product's share of total sales. We do this because there is no data for competitors in the `sh` sample schema, so the typical share calculation of product sales and competitors' sales is not possible. The processing required for our share calculation is logically similar to a competitive market share calculation.

Here are the pieces of information we find in the query, in the order we need to find them:

1. Cities whose purchases grew by more than 20% during the specified 3-month period, versus the same 3-month period last year. Note that cities are limited to one country, and sales involving no promotion.
2. Top 20% of the products for the group of cities found in the prior step. That is, find sales by product summed across this customer group, and then select the 20% of products with the best sales.
3. The share of sales for each product found in the prior step. That is, using the products group found in the prior step, find each product's share of sales of all products. Find the shares for the same period a year ago and then calculate the change in share between the two years.

The techniques used in this example are:

- This query is performed using the `WITH` clause and nested inline views. Each inline view has been given a descriptive alias to show its data element, and comment lines indicate the boundaries of each inline view. Although inline views are powerful, we believe that readability and maintenance are much easier if queries are structured to maximize the use of the `WITH` clause.
- This query does not use the `WITH` clause as extensively as it might: some of the nested inline views could have been expressed as separate subclauses of the `WITH` clause. For instance, in the main query, we use two inline views that return just one value. These are used for the denominator of the share calculations. We could have factored out these items and placed them in the `WITH` clause for greater readability. For a contrast that does use the `WITH` clause to its maximum, see "Business Intelligence Query Example 3: Customer Analysis by Grouping Customers into Buckets" regarding customer purchase analysis.
- Note the use of `CASE` expressions within the arguments to `SUM` functions. The `CASE` expressions simplify the SQL by acting as an extra set of data filters after the `WHERE` clause. They allow us to sum one column of sales for a desired date and another column for a different date.

```
WITH prod_list AS                     --START: Top 20% of products
   ( SELECT prod_id prod_subset, cume_dist_prod
    FROM                              --START: All products Sales for city subset
    ( SELECT s.prod_id, SUM(amount_sold),
         CUME_DIST() OVER (ORDER BY SUM(amount_sold)) cume_dist_prod
      FROM sales s, customers c, channels ch,  products p, times t
      WHERE s.prod_id = p.prod_id AND p.prod_total_id = 1 AND
            s.channel_id = ch.channel_id AND ch.channel_total_id = 1 AND
            s.cust_id = c.cust_id AND
            s.promo_id = 999 AND
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
             s.time_id  = t.time_id AND t.calendar_quarter_id = 1776 AND
             c.cust_city_id IN
          (SELECT cust_city_id            --START: Top 20% of cities
          FROM
            (
              SELECT cust_city_id, ((new_cust_sales - old_cust_sales)
                   / old_cust_sales ) pct_change, old_cust_sales
              FROM
              (
                 SELECT cust_city_id, new_cust_sales, old_cust_sales
                 FROM
                 (          --START: Cities AND sales for 1 country in 2 periods
                   SELECT cust_city_id,
                     SUM(CASE WHEN t.calendar_quarter_id = 1776
                        THEN amount_sold  ELSE  0  END ) new_cust_sales,
                     SUM(CASE WHEN t.calendar_quarter_id = 1772
                        THEN amount_sold ELSE 0 END) old_cust_sales
                   FROM sales s, customers c, channels ch,
                        products p, times t
                   WHERE s.prod_id = p.prod_id AND p.prod_total_id = 1 AND
                         s.channel_id = ch.channel_id AND ch.channel_total_id = 1 AND
                         s.cust_id = c.cust_id AND c.country_id = 52790 AND
                         s.promo_id = 999 AND
                         s.time_id  = t.time_id AND
                        (t.calendar_quarter_id = 1776 OR t.calendar_quarter_id =1772)
                   GROUP BY cust_city_id
                 ) cust_sales_wzeroes
                 WHERE old_cust_sales > 0
              )  cust_sales_woutzeroes
            )         --END: Cities and sales for country in 2 periods
           WHERE old_cust_sales > 0 AND  pct_change >= 0.20)
                                    --END: Top 20% of cities
GROUP BY s.prod_id
)  prod_sales                        --END: All products sales for city subset
    WHERE cume_dist_prod > 0.8      --END: Top 20% products
)
                                    --START: Main query bloc
SELECT  prod_id, ( (new_subset_sales/new_tot_sales)
               -    (old_subset_sales/old_tot_sales)
                  ) *100  share_changes
FROM
(                                   --START: Total sales for country in later period
  SELECT  prod_id,
     SUM(CASE WHEN t.calendar_quarter_id = 1776
                   THEN amount_sold  ELSE  0  END )  new_subset_sales,
        (SELECT SUM(amount_sold) FROM sales s, times t, channels ch,
                customers c, countries co, products p
          WHERE s.time_id  = t.time_id AND t.calendar_quarter_id = 1776 AND
                s.channel_id = ch.channel_id AND ch.channel_total_id = 1 AND
                s.cust_id = c.cust_id AND
                c.country_id = co.country_id AND co.country_total_id = 52806 AND
                s.prod_id = p.prod_id AND p.prod_total_id = 1 AND
                s.promo_id = 999
        )   new_tot_sales,

                                   --END: Total sales for country in later period
                                   --START: Total sales for country in earlier period
     SUM(CASE WHEN t.calendar_quarter_id = 1772
                   THEN amount_sold  ELSE  0  END)  old_subset_sales,
        (SELECT SUM(amount_sold) FROM sales s, times t, channels ch,
                customers c, countries co, products p
          WHERE s.time_id  = t.time_id AND t.calendar_quarter_id = 1772 AND
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
                s.channel_id = ch.channel_id AND ch.channel_total_id = 1 AND
                s.cust_id = c.cust_id AND
                c.country_id = co.country_id AND co.country_total_id = 52806 AND
                       s.prod_id = p.prod_id AND p.prod_total_id = 1 AND
                s.promo_id = 999
        )   old_tot_sales
                                --END: Total sales for country in earlier period
FROM sales s, customers c, countries co, channels ch, times t
WHERE s.channel_id = ch.channel_id AND ch.channel_total_id = 1 AND
      s.cust_id = c.cust_id AND
      c.country_id = co.country_id AND co.country_total_id = 52806 AND
      s.promo_id = 999 AND
      s.time_id  = t.time_id AND
     (t.calendar_quarter_id = 1776 OR t.calendar_quarter_id = 1772)
        AND s.prod_id IN
    (SELECT prod_subset FROM prod_list)
 GROUP BY prod_id);
```

## 24.1.2 Business Intelligence Query Example 2: Sales Projection that Fills in Missing Data

This query projects sales for 2022 based on the sales of 2020 and 2021. It finds the most percentage changes in sales from 2000 to 2001 and then adds that to the sales of 2022. While this is a simple calculation, there is an important thing to consider: many products have months with no sales in 2020 and 2021. We want to fill in blank values with the average sales value for the year (based on the months with actual sales). It converts currency values by country into US dollars. Finally, the query returns just the 2002 projection values.

The techniques used in this example are:

- By predefining all possible rows of data with the cross join ahead of the `MODEL` clause, we reduce the processing required by `MODEL`.
- The `MODEL` clause uses a reference model to perform currency conversion.
- By using the `CV` function extensively, we reduce the total number of rules needed to just three.
- The most interesting expression is found in the last rule, which uses a nested rule to find the currency conversion factor. To supply the country name needed for this expression, we define country as both a dimension `c` in the reference model, and a measure `cc` in the main model.

The way this example proceeds is to begin by creating a reference table of currency conversion factors. The table will hold conversion factors for each month for each country. Note that we use a cross join to specify the rows inserted into the table. For our purposes, we only set the conversion factor for one country, Canada.

```
CREATE TABLE currency (
   country         VARCHAR2(20),
   year            NUMBER,
   month           NUMBER,
   to_us           NUMBER);

INSERT INTO currency
(SELECT distinct
SUBSTR(country_name,1,20), calendar_year, calendar_quarter_number, 1
FROM sh.countries
CROSS JOIN times t
WHERE calendar_year IN (2020,2021,2022)
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
);
UPDATE currency set to_us=.74 WHERE country='Canada';
```

Here is the projection query. It starts with a `WITH` clause that has two subclauses. The first subclause finds the monthly sales per product by country for the years 2000, 2001, and 2002. The second subclause finds a list of distinct times at the month level.

```
WITH  prod_sales_mo AS       --Product sales per quarter for one country
(
SELECT country_name c, prod_id p, calendar_year  y,
   calendar_quarter_number  m, SUM(amount_sold) s
FROM sales s, customers c, times t, countries cn, promotions p, channels ch
WHERE  s.promo_id = p.promo_id AND p.promo_total_id = 1 AND
       s.channel_id = ch.channel_id AND ch.channel_total_id = 1 AND
       s.cust_id=c.cust_id  AND
       c.country_id=cn.country_id AND country_name='France' AND
       s.time_id=t.time_id  AND t.calendar_year IN  (2020, 2021,2022)
GROUP BY cn.country_name, prod_id, calendar_year, calendar_quarter_number
),
                    -- Time data used for ensuring that model has all dates
time_summary AS(  SELECT DISTINCT calendar_year cal_y, calendar_quarter_number cal_m
  FROM times
  WHERE  calendar_year IN  (2020, 2021, 2022)
)
                   --START: main query block
SELECT c, p, y, m, s,  nr FROM (
SELECT c, p, y, m, s,  nr
FROM prod_sales_mo s
                   --Use partitioned outer join to make sure that each combination
                   --of country and product has rows for all month values
  PARTITION BY (s.c, s.p)
     RIGHT OUTER JOIN time_summary ts ON
        (s.m = ts.cal_m
         AND s.y = ts.cal_y
        )
MODEL
  REFERENCE curr_conversion ON
      (SELECT country, year, month, to_us
      FROM currency)
      DIMENSION BY (country, year y,month m) MEASURES (to_us)
                                --START: main model
   PARTITION BY (s.c c)
   DIMENSION BY (s.p p, ts.cal_y y, ts.cal_m m)
   MEASURES (s.s s, CAST(NULL AS NUMBER) nr,
             s.c cc ) --country is used for currency conversion
   RULES (
                      --first rule fills in missing data with average values
      nr[ANY, ANY, ANY]
         = CASE WHEN s[CV(), CV(), CV()] IS NOT NULL
              THEN s[CV(), CV(), CV()]
              ELSE ROUND(AVG(s)[CV(), CV(), m BETWEEN 1 AND 12],2)
           END,
                      --second rule calculates projected values for 2002
      nr[ANY, 2022, ANY] = ROUND(
         ((nr[CV(),2021,CV()] - nr[CV(),2020, CV()])
          / nr[CV(),2020, CV()]) * nr[CV(),2021, CV()]
         + nr[CV(),2021, CV()],2),
                      --third rule converts 2002 projections to US dollars
      nr[ANY,y != 2022,ANY]
         = ROUND(nr[CV(),CV(),CV()]
           * curr_conversion.to_us[ cc[CV(),CV(),CV()], CV(y), CV(m)], 2)
)
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
ORDER BY c, p, y, m)
WHERE y = '2022'
ORDER BY c, p, y, m;
```

## 24.1.3 Business Intelligence Query Example 3: Customer Analysis by Grouping Customers into Buckets

One important way to understand customers is by studying their purchasing patterns and learning the profitability of each customer. This can help us decide if a customer is worth cultivating and what kind of treatment to give it. Because the `sh` sample schema data set includes many customers, a good way to start a profitability analysis is with a high level view: we will find data for a histogram of customer profitability, dividing profitability into 10 ranges (often called "buckets" for histogram analyses).For each country at an aggregation level of 1 month, we show:

- The data needed for a 10-bucket equiwidth histogram of customer profitability. That is, show the count of customers falling into each of 10 profitability buckets. This is just 10 rows of results, but it involves significant calculations.

For each profitability bucket, we also show:

- The median count of transactions per customer during the month (treating each day's purchases by 1 customer in 1 channel as a single transaction).
- The median transaction size (in local currency) per customer.
- Products that generated the most and least profit.
- Percent change of median transaction count and median transaction size versus last year.

The techniques used in this example illustrate the following:

- Using the `WITH` clause to clarify a query. By dividing the needed data into logical chunks, each of which is expressed in its own `WITH` subclause, we greatly improve readability and maintenance compared to nested inline views. The thorough use of `WITH` subclauses means that the main `SELECT` clause does not need to perform any calculations on the data it retrieves, again contributing to the readability and maintainability of the query.
- Using two analytic SQL functions, `width_bucket` equiwidth histogram buckets and `percentile_cont` to median transaction size and count.

This query shows us the analytic challenges inherent in data warehouse designs: because the `sh` data does not include entries for every transaction, nor a count of transactions, we are forced to make an assumption. In this query, we will make the minimalist interpretation and assume that all products sold to a single customer through a single channel on a single day are part of the same transaction. This approach inevitably undercounts transactions, because some customers will in fact make multiple purchases through the same channel on the same day.

Note that the query below should not be run until a materialized view is created for the initial query subfactor `cust_prod_mon_profit`. Before creating the materialized view, create two additional indexes. Unless these preparatory steps are taken, the query may require significant time to run.The two additional indexes needed and the main query are as follows:

```
CREATE BITMAP INDEX costs_chan_bix
      ON costs (channel_id)
      LOCAL NOLOGGING COMPUTE STATISTICS;

CREATE BITMAP INDEX costs_promo_bix
      ON costs (promo_id)
      LOCAL NOLOGGING COMPUTE STATISTICS;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
WITH cust_prod_mon_profit AS
-- profit by cust, prod, day, channel, promo
  (SELECT s.cust_id, s.prod_id, s.time_id,
          s.channel_id, s.promo_id,
          s.quantity_sold*(c.unit_price-c.unit_cost) profit,
          s.amount_sold dol_sold, c.unit_price price, c.unit_cost cost
   FROM sales s, costs c
   WHERE s.prod_id=c.prod_id
     AND s.time_id=c.time_id
     AND s.promo_id=c.promo_id
     AND s.channel_id=c.channel_id
     AND s.cust_id in (SELECT cust_id FROM customers cst
                       WHERE cst.country_id = 52770
     AND s.time_id IN (SELECT time_id FROM times t
                       WHERE t.calendar_month_desc = '2000-12'
   ),
-- Transaction Definition:  All products sold through a single channel to a
-- single cust on a single day are assumed to be sold in 1 transaction.
-- Some products in a transacton
-- may be on promotion
-- A customers  daily transaction amount is the sum of ALL products
-- purchased in the same channel in the same day
cust_daily_trans_amt  AS
(  SELECT cust_id, time_id, channel_id,  SUM(dol_sold) cust_daily_trans_amt
   FROM  cust_prod_mon_profit
   GROUP BY cust_id, time_id, channel_id
--A customers  monthly transaction count is the count of all channels
--used to purchase items in the same day, over all days in the month.
--It is really a count of the minimum possible number of transactions
cust_purchase_cnt  AS(  SELECT cust_id,  COUNT(*) cust_purchase_cnt
   FROM  cust_daily_trans_amt
   GROUP BY cust_id
),
--  Total profit for a customer over 1 month
cust_mon_profit AS
( SELECT cust_id, SUM(profit) cust_profit
       FROM  cust_prod_mon_profit
       GROUP BY cust_id
-- Minimum and maximum profit across all customer
-- sets endpoints for histogram data.
min_max_p AS
-- Note max profit + 0.1 to allow 10th bucket to include max value
(SELECT 0.1 + MAX(cust_profit) max_p, MIN(cust_profit) min_p
FROM cust_mon_profit),
-- Profitability bucket found for each customer
cust_bucket AS
(SELECT cust_id, cust_profit,
   width_bucket(cust_profit,
        min_max_p.min_p,
FROM cust_mon_profit,  min_max_p
-- Aggregated data needed for each bucket
histo_data AS
(  SELECT bucket,
     bucket*(( max_p-min_p) /10) top_end , count(*)  histo_count
   FROM  cust_bucket, min_max_p
   GROUP BY bucket, bucket*(( max_p - min_p) /10)
-- Median count of transactions per cust per month median_trans_count AS
-- Find median count of transactions per cust per month
(SELECT cust_bucket.bucket,
      PERCENTILE_CONT(0.5) WITHIN GROUP
          (ORDER BY cust_purchase_cnt.cust_purchase_cnt) median_trans_count
    FROM cust_bucket, cust_purchase_cnt
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
    WHERE cust_bucket.cust_id=cust_purchase_cnt.cust_id
    GROUP BY cust_bucket.bucket
-- Find Mmedian transaction size for custs by profit bucket
cust_median_trans_size AS
(  SELECT cust_bucket.bucket,
      PERCENTILE_CONT(0.5) WITHIN GROUP
          (ORDER BY cust_daily_trans_amt.cust_daily_trans_amt)
           cust_median_trans_ size
    FROM cust_bucket, cust_daily_trans_amt
    WHERE cust_bucket.cust_id=cust_daily_trans_amt.cust_id
    GROUP BY cust_bucket.bucket
-- Profitability of each product sold within each bucket
bucket_prod_profits AS
(  SELECT  cust_bucket.bucket, prod_id, SUM(profit) tot_prod_profit
   FROM  cust_bucket, cust_prod_mon_profit
   WHERE  cust_bucket.cust_id=cust_prod_mon_profit.cust_id
   GROUP BY cust_bucket.bucket, prod_id
),  -- Most and least profitable product by bucket
prod_profit AS
(  SELECT bucket, MIN(tot_prod_profit) min_profit_prod,
                  MAX(tot_prod_profit) max_profit_prod
   FROM bucket_prod_profits
   GROUP BY bucket
-- Main query block
SELECT  histo_data.bucket, histo_data.histo_count,
        median_trans_count.median_trans_count,
        cust_median_trans_size.cust_median_trans_size,
        prod_profit.min_profit_prod, prod_profit.max_profit_prod
FROM  histo_data, median_trans_count, cust_median_trans_size,
      prod_profit
WHERE  histo_data.bucket=median_trans_count.bucket
  AND  histo_data.bucket=cust_median_trans_size.bucket
  AND  histo_data.bucket=prod_profit.bucket;
```

## 24.1.4 Business Intelligence Query Example 4: Frequent Itemsets

Consider a marketing manager who wants to know which pieces of their firm's collateral are downloaded by users during a single session. That is, the manager wants to know which groupings of collateral are the most frequent itemsets. This is easy to do with the integrated frequent itemsets facility, as long as the Web site's activity log records a user ID and session ID for each collateral piece that is downloaded. For context, first we show a list of the aggregate number of downloads for individual white papers. (In our example data here, we use names of Oracle papers.)

```
White paper titles                                          #
----------------------------------------------------------  ----
Table Compression in Oracle Database 19C                     696
Field Experiences with Large Data Warehouses                 439
Key Data Warehouse Features: A Comparative Performance Analysis  181
Materialized Views in Oracle Database 19C                    167
Parallel Execution in Oracle Database 19C                    166
```

Here is a sample of the type of query that would be used for such analysis. The query uses `DBMS_FREQUENT_ITEMSET.FI_TRANSACTIONAL` as a table function. To understand the details of the query structure, see the *Oracle Database PL/SQL Packages and Types Reference*. The query returns the itemset of pairs of papers that were downloaded in a single session:

```
SELECT itemset, support, length, rnk
FROM
  (SELECT itemset, support, length,
   RANK() OVER (PARTITION BY length ORDER BY support DESC) rnk
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
FROM
(SELECT CAST(itemset AS fi_char) itemset, support, length, total_tranx
 FROM table(DBMS_FREQUENT_ITEMSET.FI_TRANSACTIONAL
  (CURSOR(SELECT session_id, command
          FROM web_log
          WHERE time_stamp BETWEEN '01-APR-2002' AND '01-JUN-2002'),
          (60/2600), 2, 2, CURSOR(SELECT 'a' FROM DUAL WHERE 1=0),
          CURSOR(SELECT 'a' FROM DUAL WHERE 1=0)))))
   WHERE rnk <= 10;
```

Here are the first three items of results:

```
White paper titles                                                    #
------------------------------------------------------------         -----
Table Compression in Oracle Database 19C                              115
Field Experiences with Large Data Warehouses

Data Warehouse Performance Enhancements with Oracle Database 19C      109
Oracle Performance and Scalability in DSS Environments

Materialized Views in Oracle Database 19C                             107
Query Optimization in Oracle Database 19C
```

This analysis yielded some interesting results. If one were to look at the list of the most popular single papers, one would expect the most popular pairs of downloaded papers would often include the white paper "Table Compression in Oracle Database 10*g*", because it was the most popular download of all papers. However, only one of the top three pairs included this paper.

By using frequent itemsets to analyze the Web log information, a manager can glean much more information than available in a simple report that only lists the most popular pieces of collateral. From these results, the manager can see that visitors to this Web site tend to search for information on a single topic area during a single session: visitors interested in scalability download white papers on compression and large data warehouses, while visitors interested in complex query capabilities download papers on query optimization and materialized views. For a marketing manager, this type of information is helpful in determining what sort of collateral should be written in the future; for a Web designer, this information can provide additional suggestions on how to organize the Web site.

See "Frequent Itemsets in SQL Analytics" for more information.


Copyright © 2001, 2025, Oracle and/or its affiliates.

# Part V
# Analytic Views

With analytic views you can easily create complex analytic queries on large amounts of hierarchical and dimensional data in database tables and views.

Analytic views are described in the following topics.


- Overview of Analytic Views
- Attribute Dimension and Hierarchy Objects
- Analytic View Objects




Copyright © 2001, 2025, Oracle and/or its affiliates.

25

# Overview of Analytic Views

Analytic views are metadata objects that enable the user to quickly and easily create complex hierarchical and dimensional queries on data in database tables and views.

General considerations of analytic views are described in the following topics.


- What Are Analytic Views?
- Privileges for Analytic Views
- Application Programming Interfaces for Analytic Views
- Compilation States of Analytic Views
- Validation of Data
- Classifications for Analytic Views
- Share Analytic Views with Application Containers
- Alter or Drop an Analytic View Object
- Data and Scripts for Examples


## 25.1 What Are Analytic Views?

Analytic views provide a fast and efficient way to create analytic queries of data stored in existing database tables and views.

Analytic views organize data using a dimensional model. They allow you to easily add aggregations and calculations to data sets and to present data in views that can be queried with relatively simple SQL.

Like standard relational views, analytic views:

- Are metadata objects (that is, they do not store data)
- Can be queried using SQL
- Can access data from other database objects such as tables, views, and external tables
- Can join multiple tables into a single view

Analytic views also:

- Organize data using a rich business model that has dimensional and hierarchical concepts
- Include system-generated columns with hierarchical data
- Automatically aggregate data
- Include embedded measure calculations that are easily defined using syntax based on the business model
- Include presentation metadata

The definition of an analytic view includes navigation, join, aggregation, and calculation rules, thus eliminating the need to include these rules in queries. Rather than having simple tables and complex `SELECT` statements that express joins, aggregations, and measure calculations,



Copyright © 2001, 2025, Oracle and/or its affiliates.

you can use simple SQL to query smart analytic views. This approach has several benefits, including:

- Simplified and faster application development; it is much easier to define calculations within analytic views than it is to write or generate complex `SELECT` statements
- Calculation rules can be defined once in the database and then be re-used by any number of applications; this provides end-users with greater freedom of choice in their use of reporting tools without concern for inconsistent results

Analytic views are especially useful for the following users:

- Data warehouse architect or designer
- Business Intelligence application developer
- Database analyst

For a data warehouse architect, analytic views are a tool for presenting data in a data warehouse to application developers and business users. Tools provided by the BI application generate a query, get the data, and present the result.

### Components of Analytic Views

Analytic view component objects consist of the following:

- Attribute dimensions, which are metadata objects that reference tables or views and organize columns into higher-level objects such as *attributes* and *levels*. Most metadata related to dimensions and hierarchies is defined in the attribute dimension object.
- Hierarchies, which are a type of view that reference attribute dimension objects and that organize data using hierarchical relationships. Data related to dimensions and hierarchies is selected from hierarchies.
- Analytic view objects, which are a type of view that presents fact data. Analytic views reference both fact tables and hierarchies. You can select both hierarchy and measure data from analytic views.
- Derived analytic views, which are defined in the `WITH` or `FROM` clause of a SELECT statement and are based on an existing analytic view.

Data dictionary views, such as `ALL_ANALYTIC_VIEW_COLUMNS`, contain the metadata and other information for the analytic view component objects.

The `DBMS_HIERARCHY` PL/SQL package contains functions for validating analytic view and hierarchy objects and a procedure that creates a table that you can use for logging messages generated by the validation functions.

### Data Sources for Analytic Views

Attribute dimensions and analytic views typically use star schema dimension tables and fact tables as data sources. For larger data sets, tables in the in-memory column store can offer the best query performance with analytic views. Analytic views can also be used with snowflake schemas, denormailized tables, external tables and remote tables.

You specify the data source with the `using_clause` in the attribute dimension or analytic view definition. You may specify an alias for the data source.

A database user who has the privileges required for access to the data sources can create analytic view objects. The creator defines the business model, which specifies how the data is queried, and implements the model by creating attribute dimensions, hierarchies, and analytic views.


Copyright © 2001, 2025, Oracle and/or its affiliates.

### Materialized Views and Analytic Views

Creating a materialized view over queries of an analytic view or a hierarchy is not supported. You may use a materialized view in a `MEASURE_GROUP` phrase of a `cache_clause` of an analytic view.

### Constraints for Analytic View Objects

For optimal query performance in queries of an analytic view, you should use the same constraints that you would typically use for querying a star schema. An attribute dimension or analytic view does not require that the source table or view have any particular constraints defined or enabled. Also, defining an attribute dimension or analytic view does not introduce any additional constraints on those tables or views. The PL/SQL functions `VALIDATE_HIERARCHY` and `VALIDATE_ANALYTIC_VIEW` are available for validating that the data in a table or view used by an attribute dimension in a hierarchy or used by an analytic view conforms to the logical constraints inherent in the metadata definitions.

### Naming Conventions for Analytic Views

The naming conventions for attribute dimensions, hierarchies, and analytic views, and components of them such as attributes, levels, and measures, follow standard database identifier rules. Double-quotes may be used to enclose identifiers, including extended characters and mixed-case; otherwise, the standard upper-case and limited character rules apply.

## 25.2 New Features for Analytic Views

Oracle AI Database 26ai includes these new features for analytic views.

New features for analytic views include the following:

- Base table query transformations
- Access to calculations through transparency views
- Remote source support
- Query-scoped base measures
- Autonomous caching
- Attribute dimension star caches
- Aggregation table support

### Base Table Query Transformations

The `ENABLE QUERY TRANSFORM RELY` clause in a `CREATE OR REPLACE ANALYTIC VIEW` statement enables the automatic creation of views that can improve the performance of queries. A query against a base table for the analytic view is automatically transformed into a query of the analytic view. This provides you with the performance improvements of analytic views without your needing to change to your SQL query.

### Access to Calculations Through Transparency Views

You can create transparency views using the `CREATE_VIEW_FOR_FACT_ROWS` and `CREATE_VIEW_FOR_STAR_ROWS` procedures of the `DBMS_HIERARCHY` package. When you query an analytic view and specify the `FACT ROWS` or `STAR ROWS` keywords in the `SELECT` statement, a transparency view is automatically created. The `FACT ROWS` keywords indicate that the analytic view should return rows as they are in the fact table, and the `STAR ROWS` keywords indicate that


Copyright © 2001, 2025, Oracle and/or its affiliates.

the analytic view should return rows for an attribute dimension. These keywords enable the analytic view to use base table query transformation.

One of the key features of analytic views is the hierarchy-aware analytic calculations used in creation of calculated measures. In a `SELECT` statement, you can specify the `AV_AGGREGATE` function to query a calculated measure

### Remote Source Support

When creating an analytic view or attribute dimension, you can specify the `REMOTE` keyword in the `USING` clause to include a remote table as a source for the object. The data dictionary tables `ALL_ATTRIBUTE_DIM_TABLES` and `ALL_ANALYTIC_VIEWS`, and their related `DBA_` and `USER_` tables, now have the `IS_REMOTE` column that indicates whether a source is remote.

### Query-Scoped Base Measures

In a query, you can add new base measures to a dynamic analytic view with the `FACT` and `AGGREGATE BY` keywords in the `ADD MEASURES` clause. Each base measure can have a different aggregation operator.

### Autonomous Caching

With procedures in the `DBMS_AVTUNE` PL/SQL package, you can enable the automatic creation of caches for an analytic view. These caches improve the performance of queries of the analytic view and other transformed SQL queries.

### Attribute Dimension Star Caches

A fact-based hierarchy is built over one or more columns of the fact table. When creating an attribute dimension, you can specify the creation of a cache for the star representation of a fact-based hierarchy. All hierarchies and analytic views based on that attribute dimension are able to share the single materialized star cache. In a query of the fact table, the cache eliminates the need to compute the distinct values of the hierarchy members.

### Aggregation Table Support

When creating an analytic view, you can specify an object, such as an aggregate table, a view, or a materialized view, to use in a level grouping cache in place of a materialized view. You can then refresh the aggregate table as desired, which allows you complete control of the aggregate results.

You specify the aggregation object with the `MATERIALIZED USING` keywords for a level in the measure group of the cache specification clause of the analytic view.

## 25.3 Privileges for Analytic Views

Describes the system and object privileges available for analytic views, attribute dimensions, and hierarchies.

### System Privileges

The following system privileges allow the user to create, alter, or drop analytic view component objects.

| System Privilege | Description |
|---|---|
| `CREATE ANALYTIC VIEW` | Create an analytic view in the grantee's schema. |


Copyright © 2001, 2025, Oracle and/or its affiliates.

| System Privilege | Description |
|---|---|
| CREATE ANY ANALYTIC VIEW | Create analytic views in any schema except SYS. |
| CREATE ATTRIBUTE DIMENSION | Create an attribute dimension in the grantee's schema. |
| CREATE ANY ATTRIBUTE DIMENSION | Create attribute dimensions in any schema except SYS. |
| CREATE HIERARCHY | Create a hierarchy in the grantee's schema. |
| CREATE ANY HIERARCHY | Create hierarchies in any schema except SYS. |
| ALTER ANY ANALYTIC VIEW | Rename analytic views in any schema except SYS. |
| ALTER ANY ATTRIBUTE DIMENSION | Rename attribute dimensions in any schema except SYS. |
| ALTER ANY HIERARCHY | Rename hierarchies in any schema except SYS. |
| DROP ANY ANALYTIC VIEW | Drop analytic views in any schema except SYS. |
| DROP ANY ATTRIBUTE DIMENSION | Drop attribute dimensions in any schema except SYS. |
| DROP ANY HIERARCHY | Drop hierarchies in any schema except SYS. |
| SELECT ANY TABLE | Query or view any analytic view or hierarchy in any schema. |

### Object Privileges

The following object privileges allow the user to query or rename analytic view component objects.

| Object Privilege | Operations Authorized |
|---|---|
| ALTER | Rename the analytic view, attribute dimension, or hierarchy. |
| READ | Query the object with the SELECT statement. |
| SELECT | Query the object with the SELECT statement. |

**Example 25-1 Granting System Privileges**

The following statements grant the CREATE system privilege to the user av_user.

```
GRANT CREATE ATTRIBUTE DIMENSION TO av_user;
GRANT CREATE HIERARCHY TO av_user;
GRANT CREATE ANALYTIC VIEW TO av_user;
GRANT SELECT ANY TABLE TO av_user;
```

**Example 25-2 Granting Object Privileges**

The following statements grant all object privileges to the user av_user2 and then revoke the ALTER privilege.

```
GRANT ALL ON "AV_USER".SALES_AV TO "AV_USER2";
REVOKE ALTER ON "AV_USER".SALES_AV FROM "AV_USER2";
```

## 25.4 Application Programming Interfaces for Analytic Views

The application programming interfaces for analytic views consist of SQL DDL statements, PL/SQL procedures and functions, and data dictionary views.


Copyright © 2001, 2025, Oracle and/or its affiliates.

These interfaces are listed in the following topics:

- SQL DDL Statements for the Creation and Management of Analytic Views
- PL/SQL Package for Analytic Views
- Data Dictionary Views for Analytic Views

### SQL DDL Statements for the Creation and Management of Analytic Views

You create and manage analytic view objects with the following SQL DDL statements:

- `CREATE ANALYTIC VIEW`
- `CREATE ATTRIBUTE DIMENSION`
- `CREATE HIERARCHY`
- `ALTER ANALYTIC VIEW`
- `ALTER ATTRIBUTE DIMENSION`
- `ALTER HIERARCHY`
- `DROP ANALYTIC VIEW`
- `DROP ATTRIBUTE DIMENSION`
- `DROP HIERARCHY`

For details about these statements, see `CREATE ANALYTIC VIEW` and the other statements in *Oracle AI Database SQL Language Reference*.

### SQL SELECT Statement Clauses for Filtered Facts and Added Measures

In the `WITH` and `FROM` clauses of a `SELECT` statement, you can define one or more transitory analytic views that filter the hierarchy members before the aggregation of measure values for the hierarchy. You can also define additional measures that participate in the query. The filtered facts and additional measures are based on an existing persistent analytic view, but they do not alter the definition of the persistent analytic view itself.

> **See Also**
>
> Analytic View Queries with Filtered Facts and Added Measures

### PL/SQL Package for Analytic Views

You can validate the data for analytic view and hierarchy objects with the following procedures in the `DBMS_HIERARCHY` package:

- `CREATE_VALIDATE_LOG_TABLE` procedure
- `VALIDATE_ANALYTIC_VIEW` function
- `VALIDATE_CHECK_SUCCESS` function
- `VALIDATE_HIERARCHY` function

For details about this package, see DBMS_HIERARCHY in *Oracle AI Database PL/SQL Packages and Types Reference*.


Copyright © 2001, 2025, Oracle and/or its affiliates.

### Data Dictionary Views for Analytic Views

The following data dictionary views contain information about analytic view objects. Only the views with the prefix `ALL` are listed. Each view also has a corresponding `DBA` and `USER` version.

#### Analytic View Views

- `ALL_ANALYTIC_VIEW_ATTR_CLASS`
- `ALL_ANALYTIC_VIEW_BASE_MEAS`
- `ALL_ANALYTIC_VIEW_CALC_MEAS`
- `ALL_ANALYTIC_VIEW_CLASS`
- `ALL_ANALYTIC_VIEW_COLUMNS`
- `ALL_ANALYTIC_VIEW_DIM_CLASS`
- `ALL_ANALYTIC_VIEW_DIMENSIONS`
- `ALL_ANALYTIC_VIEW_HIER_CLASS`
- `ALL_ANALYTIC_VIEW_HIERS`
- `ALL_ANALYTIC_VIEW_KEYS`
- `ALL_ANALYTIC_VIEW_LEVEL_CLASS`
- `ALL_ANALYTIC_VIEW_LEVELS`
- `ALL_ANALYTIC_VIEW_LVLGRPS`
- `ALL_ANALYTIC_VIEW_MEAS_CLASS`
- `ALL_ANALYTIC_VIEWS`

#### Attribute Dimension Views

- `ALL_ATTRIBUTE_DIM_ATTR_CLASS`
- `ALL_ATTRIBUTE_DIM_ATTRS`
- `ALL_ATTRIBUTE_DIM_CLASS`
- `ALL_ATTRIBUTE_DIM_JOIN_PATHS`
- `ALL_ATTRIBUTE_DIM_KEYS`
- `ALL_ATTRIBUTE_DIM_LEVEL_ATTRS`
- `ALL_ATTRIBUTE_DIM_LEVELS`
- `ALL_ATTRIBUTE_DIM_LVL_CLASS`
- `ALL_ATTRIBUTE_DIM_ORDER_ATTRS`
- `ALL_ATTRIBUTE_DIM_TABLES`
- `ALL_ATTRIBUTE_DIMENSIONS`

#### Hierarchy Views

- `ALL_HIER_CLASS`
- `ALL_HIER_COLUMNS`
- `ALL_HIER_HIER_ATTR_CLASS`
- `ALL_HIER_HIER_ATTRIBUTES`


Copyright © 2001, 2025, Oracle and/or its affiliates.

- ALL_HIER_JOIN_PATHS
- ALL_HIER_LEVEL_ID_ATTRS
- ALL_HIER_LEVELS
- ALL_HIERARCHIES

For details about these views, see ALL_ANALYTIC_VIEWS and the other views in *Oracle AI Database Reference*.

# 25.5 Compilation States of Analytic Views

When you create or alter an attribute dimension, a hierarchy, or an analytic view, Oracle AI Database ascertains the internal validity of the object's metadata.

The SQL DDL CREATE and ALTER statements for analytic views have FORCE and NOFORCE options, with NOFORCE as the default. The verification of metadata that depends on another object is optional and is determined by the FORCE and NOFORCE options.

If you specify NOFORCE and the compilation fails, then the CREATE or ALTER operation fails and an error is raised. If you specify FORCE, the CREATE or ALTER succeeds even if the compilation fails.

You can explicitly invoke a compilation by specifying the COMPILE keyword; a compilation is implicitly invoked as needed during a query. A query returns an error if an object is not compiled and cannot implicitly be compiled.

The compilation state is recorded in the COMPILE_STATE column in the ALL_ATTRIBUTE_DIMENSIONS, ALL_HIERARCHIES, and ALL_ANALYTIC_VIEWS data dictionary views (and the corresponding DBA and USER views). The state may be one of the following:

| Value | Description |
|---|---|
| VALID | The object has been compiled without error. |
| INVALID | Some change requires recompilation or the object has been compiled and errors have occurred. |

A SQL DDL operation on the analytic views object causes the state of dependent objects to change to INVALID. For example, a change to an attribute dimension causes any hierarchies that use that attribute dimension, and analytic views dimensioned by the attribute dimension, to change state to INVALID. Also, DDL changes to the tables or views used by attribute dimensions and analytic views cause the state for those objects to change to INVALID.

The ALL_OBJECTS data dictionary view has a STATUS column that may be VALID or INVALID. For attribute dimensions, hierarchies, and analytic views, the STATUS value correlates to the COMPILE_STATE. When COMPILE_STATE is VALID, the STATUS value is VALID. When COMPILE_STATE is INVALID, STATUS is INVALID.

# 25.6 Validation of Data

To ensure the accuracy of query results, the data of hierarchies and analytic views must be validated.

To validate the data for a hierarchy or analytic view, use the functions in the PL/SQL package DBMS_HIERARCHY. The VALIDATE_HIERARCHY and VALIDATE_ANALYTIC_VIEW functions validate the data and store the results in a table. An optional argument to the functions is the name of a table. The CREATE_VALIDATE_LOG_TABLE procedure creates a table that you can use for the


Copyright © 2001, 2025, Oracle and/or its affiliates.

purpose. If you do not specify a table, the VALIDATE_HIERARCHY and VALIDATE_ANALYTIC_VIEW functions create a table.

Any SQL DDL or DML changes made on the tables used by an associated attribute dimension or analytic view, or any DDL change to an attribute dimension, hierarchy, or analytic view itself, causes the state of a hierarchy to change to INVALID.

If any data security policies are applied to a hierarchy or analytic view, or any of the tables or views used by an associated attribute dimension, then the validation state cannot be determined and the VALIDATE_STATE is not set to VALID. An execution of the VALIDATE_HIERARCHY or VALIDATE_ANALYTIC_VIEW function indicates whether the hierarchy or analytic view is valid at that time and for that user.

If a SQL DML change to a table or view used by an attribute dimension occurs between the time you query the data dictionary or run the VALIDATE_HIERARCHY function and the time you execute a query of a hierarchy or analytic view, then the hierarchy may become invalid. To ensure that a hierarchy is valid for a query, you can establish a read-only transaction (for example, SET TRANSACTION READ ONLY), run the validation function, verify the success of the validation, execute queries, and then end the transaction with a COMMIT or ROLLBACK statement.

# 25.7 Classifications for Analytic Views

Classifications provide descriptive metadata for attribute dimensions, hierarchies, and analytic view objects, and for components of them such as attribute dimension keys, attributes, levels, and measures.

Applications can use classifications to present information about hierarchies and analytic views. Classifications are similar to comments on tables and columns, but a comment is a single value. You can specify any number of classifications for the same object. You can vary the values by language. A classification value is always a text literal and can have maximum length of 4000 bytes.

Classifications play no role in SQL queries, but are available in the data dictionary views for use by tools or applications. The CAPTION and DESCRIPTION classifications have DDL shortcuts for all objects that support classifications.

You may specify a language for a classification value. If you specify a language, it must be a valid NLS_LANGUAGE value. If you do not specify a language, then the language value for the classification is NULL and the default database language is used.

The DDL shortcuts for CAPTION and DESCRIPTION apply only to the NULL language. To specify a CAPTION and DESCRIPTION classification for a particular language, you must use the full CLASSIFICATION syntax.

SQL tools can interpret a NULL language value as a default. For example, suppose a tool is looking for the CAPTION for an attribute dimension. The tool might first look for the CAPTION having a language that matches the current NLS_LANGUAGE. If it finds one, it uses that CAPTION value. If not, it then looks for a CAPTION having a NULL language value and uses that. The SQL logic is up to the user, tool, or application.

To provide descriptive metadata that varies by language for a member of a hierarchy, use the hierarchical attributes MEMBER_NAME, MEMBER_CAPTION, and MEMBER_DESCRIPTION.

# 25.8 Share Analytic Views with Application Containers

You can share analytic views with application containers.


Copyright © 2001, 2025, Oracle and/or its affiliates.

In the definition of analytic view objects, you can use the `SHARING` clause to share attribute dimension, hierarchy, or analytic view metadata or objects with application containers. The values for the clause are the following:

| Value | Description |
|---|---|
| `NONE` | Do not share; this is the default value. |
| `METADATA` | Share metadata only. |
| `OBJECT` | Share the object, including data. |

If you specify `METADATA`, then only the definition of the object is shared with application containers.

If you specify `OBJECT`, then the attribute dimension, hierarchy, or analytic view object, including the data sources of the object, is shared with the application container.

# 25.9 Alter or Drop an Analytic View Object

With SQL DDL statements you can change the name of an object or you can drop it.

To alter any aspect of an analytic view object other than the name, use a `CREATE OR REPLACE` statement to replace the object with one that has the desired alterations.

**Example 25-3 Renaming an Attribute Dimension**

The following example renames an attribute dimension.

```
ALTER ATTRIBUTE DIMENSION product_attr_dim RENAME TO myproduct_attr_dim;
```

**Example 25-4 Dropping an Attribute Dimension**

The following example drops an attribute dimension.

```
DROP ATTRIBUTE DIMENSION myproduct_attr_dim;
```

# 25.10 Data and Scripts for Examples

This section describes the data on which the analytic views examples are based and contains SQL statements that create the analytic view component objects.

The data and the analytic view components are described in the following topics:

- About the Data and Scripts for Examples
- Create Attribute Dimension Statements
- Create Hierarchy Statements
- Create Analytic View Statements

## 25.10.1 About the Data and Scripts for Examples

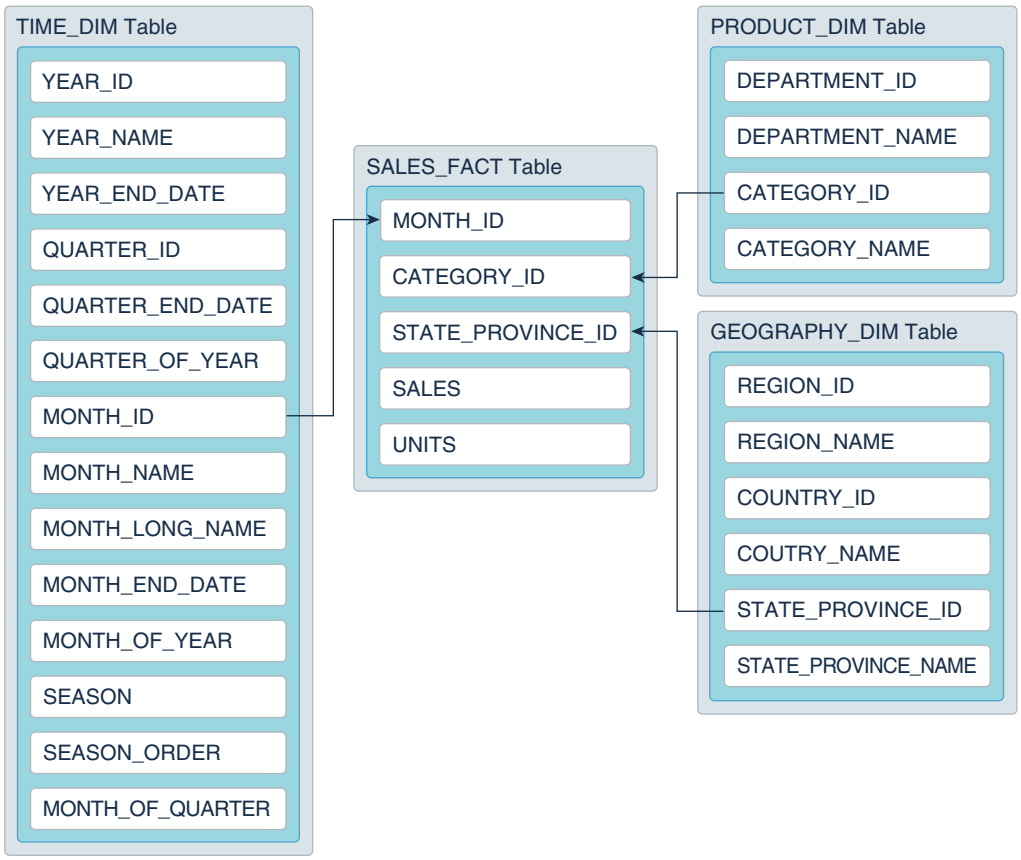
The data used by the examples consists of sales data in a single fact table and three dimension tables with time periods, products, and geographies.


Copyright © 2001, 2025, Oracle and/or its affiliates.

You can view and run the SQL scripts that create the tables, the analytic view component objects, and the queries used in the examples from the Oracle Live SQL website at https://livesql.oracle.com/apex/livesql/file/index.html.

The data is in the star schema tables shown in the following figure.

**Figure 25-1 Tables for Analytic View Examples**




In the SALES_FACT table, the MONTH_ID, DEPARTMENT_ID, and STATE_PROVINCE_ID columns are foreign keys to the TIME_DIM, PRODUCT_DIM, and GEOGRAPHY_DIM dimension tables, respectively.

In each dimension table, the _ID columns are used as keys and the _NAME columns are used as descriptors. Other columns may be used as attributes for sorting or reporting.

There are 1:1 relationships in data between _ID and _NAME columns. You can sort time periods by using the _END_DATE columns of the TIME_DIM table.


Copyright © 2001, 2025, Oracle and/or its affiliates.

## 25.10.2 Create Attribute Dimension Statements

This topic contains SQL statements that create the example attribute dimensions.

**Create the time_attr_dim Attribute Dimension**

The time_attr_dim attribute dimension is based on the TIME_DIM dimension table. The following statement creates the attribute dimension.

```
CREATE OR REPLACE ATTRIBUTE DIMENSION time_attr_dim
DIMENSION TYPE TIME
USING time_dim
ATTRIBUTES
 (year_id
   CLASSIFICATION caption VALUE 'YEAR_ID'
   CLASSIFICATION description VALUE 'YEAR ID',
  year_name
    CLASSIFICATION caption VALUE 'YEAR_NAME'
    CLASSIFICATION description VALUE 'Year',
  year_end_date
    CLASSIFICATION caption VALUE 'YEAR_END_DATE'
    CLASSIFICATION description VALUE 'Year End Date',
  quarter_id
    CLASSIFICATION caption VALUE 'QUARTER_ID'
    CLASSIFICATION description VALUE 'QUARTER ID',
  quarter_name
    CLASSIFICATION caption VALUE 'QUARTER_NAME'
    CLASSIFICATION description VALUE 'Quarter',
  quarter_end_date
    CLASSIFICATION caption VALUE 'QUARTER_END_DATE'
    CLASSIFICATION description VALUE 'Quarter End Date',
  quarter_of_year
    CLASSIFICATION caption VALUE 'QUARTER_OF_YEAR'
    CLASSIFICATION description VALUE 'Quarter of Year',
  month_id
    CLASSIFICATION caption VALUE 'MONTH_ID'
    CLASSIFICATION description VALUE 'MONTH ID',
  month_name
    CLASSIFICATION caption VALUE 'MONTH_NAME'
    CLASSIFICATION description VALUE 'Month',
  month_long_name
    CLASSIFICATION caption VALUE 'MONTH_LONG_NAME'
    CLASSIFICATION description VALUE 'Month Long Name',
  month_end_date
    CLASSIFICATION caption VALUE 'MONTH_END_DATE'
    CLASSIFICATION description VALUE 'Month End Date',
  month_of_quarter
    CLASSIFICATION caption VALUE 'MONTH_OF_QUARTER'
    CLASSIFICATION description VALUE 'Month of Quarter',
  month_of_year
    CLASSIFICATION caption VALUE 'MONTH_OF_YEAR'
    CLASSIFICATION description VALUE 'Month of Year',
  season
    CLASSIFICATION caption VALUE 'SEASON'
    CLASSIFICATION description VALUE 'Season',
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
  season_order
    CLASSIFICATION caption VALUE 'SEASON_ORDER'
    CLASSIFICATION description VALUE 'Season Order')
LEVEL month
  LEVEL TYPE MONTHS
  CLASSIFICATION caption VALUE 'MONTH'
  CLASSIFICATION description VALUE 'Month'
  KEY month_id
  MEMBER NAME month_name
  MEMBER CAPTION month_name
  MEMBER DESCRIPTION month_long_name
  ORDER BY month_end_date
  DETERMINES (month_end_date,
    quarter_id,
    season,
    season_order,
    month_of_year,
    month_of_quarter)
LEVEL quarter
  LEVEL TYPE QUARTERS
  CLASSIFICATION caption VALUE 'QUARTER'
  CLASSIFICATION description VALUE 'Quarter'
  KEY quarter_id
  MEMBER NAME quarter_name
  MEMBER CAPTION quarter_name
  MEMBER DESCRIPTION quarter_name
  ORDER BY quarter_end_date
  DETERMINES (quarter_end_date,
    quarter_of_year,
    year_id)
LEVEL year
  LEVEL TYPE YEARS
  CLASSIFICATION caption VALUE 'YEAR'
  CLASSIFICATION description VALUE 'Year'
  KEY year_id
  MEMBER NAME year_name
  MEMBER CAPTION year_name
  MEMBER DESCRIPTION year_name
  ORDER BY year_end_date
  DETERMINES (year_end_date)
LEVEL season
  LEVEL TYPE QUARTERS
  CLASSIFICATION caption VALUE 'SEASON'
  CLASSIFICATION description VALUE 'Season'
  KEY season
  MEMBER NAME season
  MEMBER CAPTION season
  MEMBER DESCRIPTION season
LEVEL month_of_quarter
  LEVEL TYPE MONTHS
  CLASSIFICATION caption VALUE 'MONTH_OF_QUARTER'
  CLASSIFICATION description VALUE 'Month of Quarter'
  KEY month_of_quarter;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

### Create the product_attr_dim Attribute Dimension

The product_attr_dim attribute dimension is based on the PRODUCT_DIM dimension table. The following statement creates the attribute dimension.

```
CREATE OR REPLACE ATTRIBUTE DIMENSION product_attr_dim
USING product_dim
ATTRIBUTES
 (department_id
    CLASSIFICATION caption VALUE 'DEPARTMENT_ID'
    CLASSIFICATION description VALUE 'DEPARTMENT ID',
  department_name
    CLASSIFICATION caption VALUE 'DEPARTMENT_NAME'
    CLASSIFICATION description VALUE 'Department',
  category_id
    CLASSIFICATION caption VALUE 'CATEGORY_ID'
    CLASSIFICATION description VALUE 'CATEGORY ID',
  category_name
    CLASSIFICATION caption VALUE 'CATEGORY_NAME'
    CLASSIFICATION description VALUE 'Category')
LEVEL DEPARTMENT
  CLASSIFICATION caption VALUE 'DEPARTMENT'
  CLASSIFICATION description VALUE 'Department'
  KEY department_id
  MEMBER NAME department_name
  MEMBER CAPTION department_name
  ORDER BY department_name
LEVEL CATEGORY
  CLASSIFICATION caption VALUE 'CATEGORY'
  CLASSIFICATION description VALUE 'Category'
  KEY category_id
  MEMBER NAME category_name
  MEMBER CAPTION category_name
  ORDER BY category_name
  DETERMINES(department_id)
ALL MEMBER NAME 'ALL PRODUCTS';
```

### Create the geography_attr_dim Attribute Dimension

The geography_attr_dim attribute dimension is based on the GEOGRAPHY_DIM dimension table. The following statement creates the attribute dimension.

```
CREATE OR REPLACE ATTRIBUTE DIMENSION geography_attr_dim
USING geography_dim
ATTRIBUTES
 (region_id
    CLASSIFICATION caption VALUE 'REGION_ID'
    CLASSIFICATION description VALUE 'REGION ID',
  region_name
    CLASSIFICATION caption VALUE 'REGION_NAME'
    CLASSIFICATION description VALUE 'Region',
  country_id
    CLASSIFICATION caption VALUE 'COUNTRY_ID'
    CLASSIFICATION description VALUE 'COUNTRY ID',
  country_name
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
    CLASSIFICATION caption VALUE 'COUNTRY_NAME'
    CLASSIFICATION description VALUE 'Country',
  state_province_id
    CLASSIFICATION caption VALUE 'STATE_PROVINCE_ID'
    CLASSIFICATION description VALUE 'STATE-PROVINCE ID',
  state_province_name
    CLASSIFICATION caption VALUE 'STATE_PROVINCE_NAME'
    CLASSIFICATION description VALUE 'State-Province')
LEVEL REGION
  CLASSIFICATION caption VALUE 'REGION'
  CLASSIFICATION description VALUE 'Region'
  KEY region_id
  MEMBER NAME region_name
  MEMBER CAPTION region_name
  ORDER BY region_name
LEVEL COUNTRY
  CLASSIFICATION caption VALUE 'COUNTRY'
  CLASSIFICATION description VALUE 'Country'
  KEY country_id
  MEMBER NAME country_name
  MEMBER CAPTION country_name
  ORDER BY country_name
  DETERMINES(region_id)
LEVEL STATE_PROVINCE
  CLASSIFICATION caption VALUE 'STATE_PROVINCE'
  CLASSIFICATION description VALUE 'State-Province'
  KEY state_province_id
  MEMBER NAME state_province_name
  MEMBER CAPTION state_province_name
  ORDER BY state_province_name
  DETERMINES(country_id)
ALL MEMBER NAME 'ALL CUSTOMERS';
```

## 25.10.3 Create Hierarchy Statements

This topic contains SQL statements that create the example hierarchies.

**Create Hierarchies Using time_attr_dim**

The following statements create hierarchies that use the time_attr_dim attribute dimension.

```
CREATE OR REPLACE HIERARCHY time_hier
  CLASSIFICATION caption VALUE 'CALENDAR'
  CLASSIFICATION description VALUE 'CALENDAR'
USING time_attr_dim
  (month CHILD OF
   quarter CHILD OF
   year);
--
CREATE OR REPLACE HIERARCHY time_season_hier
  CLASSIFICATION caption VALUE 'SEASONS'
  CLASSIFICATION description VALUE 'Seasons'
USING time_attr_dim
  (month CHILD OF
   season);
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
--
CREATE OR REPLACE HIERARCHY time_year_season_hier
USING time_attr_dim
  (month CHILD OF
   season CHILD OF
   year);
--
CREATE OR REPLACE HIERARCHY time_month_of_qtr_hier
  CLASSIFICATION caption VALUE 'MONTH_OF_QUARTER'
  CLASSIFICATION description VALUE 'Month of Quarter'
USING time_attr_dim
  (month CHILD OF
   month_of_quarter);
```

**Create a Hierarchy Using product_attr_dim**

The following statement creates a hierarchy that uses the product_attr_dim attribute dimension.

```
CREATE OR REPLACE HIERARCHY product_hier
  CLASSIFICATION caption VALUE 'PRODUCT'
  CLASSIFICATION description VALUE 'Product'
USING product_attr_dim
  (CATEGORY
   CHILD OF department);
```

**Create a Hierarchy Using geography_attr_dim**

The following statement creates a hierarchy that uses the geography_attr_dim attribute dimension.

```
CREATE OR REPLACE HIERARCHY geography_hier
  CLASSIFICATION caption VALUE 'GEOGRAPHY'
  CLASSIFICATION description VALUE 'Geography'
USING geography_attr_dim
  (state_province
   CHILD OF country
   CHILD OF region);
```

## 25.10.4 Create Analytic View Statements

This topic contains a SQL statement that creates the example analytic view.

**Create the sales_av Analytic View**

The following statement creates an analytic view that uses the SALES_FACT fact table.

```
CREATE OR REPLACE ANALYTIC VIEW sales_av
  CLASSIFICATION caption VALUE 'Sales AV'
  CLASSIFICATION description VALUE 'Sales Analytic View'
  CLASSIFICATION created_by VALUE 'Harold C. Ehrlicher'
USING sales_fact
DIMENSION BY
  (time_attr_dim
    KEY month_id REFERENCES month_id
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
    HIERARCHIES (
      time_hier DEFAULT,
      time_season_hier,
      time_year_season_hier,
      time_month_of_qtr_hier),
   product_attr_dim
    KEY category_id REFERENCES category_id
    HIERARCHIES (
      product_hier DEFAULT),
   geography_attr_dim
    KEY state_province_id
    REFERENCES state_province_id
    HIERARCHIES (
      geography_hier DEFAULT)
   )
MEASURES
 (sales FACT sales
    CLASSIFICATION caption VALUE 'Sales'
    CLASSIFICATION description VALUE 'Sales'
    CLASSIFICATION format_string VALUE '$9,999.99',
  units FACT units
    CLASSIFICATION caption VALUE 'Units'
    CLASSIFICATION description VALUE 'Units Sold'
    CLASSIFICATION format_string VALUE '9,999',
  sales_prior_period AS
    (LAG(SALES) OVER (HIERARCHY time_hier OFFSET 1))
      CLASSIFICATION caption VALUE 'Sales Prior Period'
      CLASSIFICATION description VALUE 'Sales Prior_Period'
      CLASSIFICATION format_string VALUE '$9,999.99',
  sales_chg_prior_period AS
    (LAG_DIFF(SALES) OVER (HIERARCHY time_hier OFFSET 1))
      CLASSIFICATION caption VALUE 'Sales Change Prior Period'
      CLASSIFICATION description VALUE 'Sales Change Prior Period'
      CLASSIFICATION format_string VALUE '$9,999.99',
  sales_qtr_ago AS
    (LAG(SALES) OVER (HIERARCHY time_hier OFFSET 1
      ACROSS ANCESTOR AT LEVEL quarter))
        CLASSIFICATION caption VALUE 'Sales Qtr Ago'
        CLASSIFICATION description VALUE 'Sales Qtr Ago'
        CLASSIFICATION format_string VALUE '$9,999.99',
  sales_chg_qtr_ago AS
    (LAG_DIFF(SALES) OVER (HIERARCHY time_hier OFFSET 1
      ACROSS ANCESTOR AT LEVEL quarter))
        CLASSIFICATION caption VALUE 'Sales Change Qtr Ago'
        CLASSIFICATION description VALUE 'Sales Change Qtr Ago'
        CLASSIFICATION format_string VALUE '$9,999.99',
  sales_pct_chg_qtr_ago AS
    (LAG_DIFF_PERCENT(SALES) OVER (HIERARCHY time_hier OFFSET 1
      ACROSS ANCESTOR AT LEVEL quarter))
        CLASSIFICATION caption VALUE 'Sales Percent Change Qtr Ago'
        CLASSIFICATION description VALUE 'Sales Percent Change Qtr Ago'
        CLASSIFICATION format_string VALUE '999.99',
  sales_yr_ago AS
    (LAG(SALES) OVER (HIERARCHY time_hier OFFSET 1
      ACROSS ANCESTOR AT LEVEL year))
        CLASSIFICATION caption VALUE 'Sales Year Ago'
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
      CLASSIFICATION description VALUE 'Sales Year Ago'
      CLASSIFICATION format_string VALUE '$9,999.99',
sales_chg_yr_ago AS
  (LAG_DIFF(SALES) OVER (HIERARCHY time_hier OFFSET 1
    ACROSS ANCESTOR AT LEVEL year))
      CLASSIFICATION caption VALUE 'Sales Change Year Ago'
      CLASSIFICATION description VALUE 'Sales Change Year Ago'
      CLASSIFICATION format_string VALUE '$9,999.99',
sales_pct_chg_yr_ago AS
  (LAG_DIFF_PERCENT(SALES) OVER (HIERARCHY time_hier OFFSET 1
    ACROSS ANCESTOR AT LEVEL year))
      CLASSIFICATION caption VALUE 'Sales Percent Change Year Ago'
      CLASSIFICATION description VALUE 'Sales Percent Change Year Ago'
      CLASSIFICATION format_string VALUE '999.99',
sales_qtd AS
  (SUM(sales) OVER (HIERARCHY time_hier
    BETWEEN UNBOUNDED PRECEDING AND CURRENT MEMBER
    WITHIN ANCESTOR AT LEVEL quarter))
      CLASSIFICATION caption VALUE 'Sales Quarter to Date'
      CLASSIFICATION description VALUE 'Sales Quarter to Date'
      CLASSIFICATION format_string VALUE '$9,999.99',
sales_ytd AS
  (SUM(sales) OVER (HIERARCHY time_hier
    BETWEEN UNBOUNDED PRECEDING AND CURRENT MEMBER
    WITHIN ANCESTOR AT LEVEL year))
      CLASSIFICATION caption VALUE 'Sales Year to Date'
      CLASSIFICATION description VALUE 'Sales Year to Date'
      CLASSIFICATION format_string VALUE '$9,999.99',
sales_2011 AS
  (QUALIFY (sales, time_hier = year['11']))
    CLASSIFICATION caption VALUE 'Sales CY2011'
    CLASSIFICATION description VALUE 'Sales CY2011'
    CLASSIFICATION format_string VALUE '$9,999.99',
sales_pct_chg_2011 AS
  ((sales - (QUALIFY (sales, time_hier = year['11']))) /
  (QUALIFY (sales, time_hier = year['11'])))
    CLASSIFICATION caption VALUE 'Sales Pct Change CY2011'
    CLASSIFICATION description VALUE 'Sales Pct Change CY2011'
    CLASSIFICATION format_string VALUE '999.99',
sales_share_time_parent AS
  (SHARE_OF(sales HIERARCHY time_hier PARENT))
    CLASSIFICATION caption VALUE 'Sales Share of Time Parent'
    CLASSIFICATION description VALUE 'Sales Share of Time Parent'
    CLASSIFICATION format_string VALUE '999.99',
sales_share_season_parent AS
  (SHARE_OF(sales HIERARCHY time_season_hier PARENT))
    CLASSIFICATION caption VALUE 'Sales Share of Season Parent'
    CLASSIFICATION description VALUE 'Sales Share of Season Parent'
   CLASSIFICATION format_string VALUE '999.99',
sales_share_prod_parent AS
  (SHARE_OF(sales HIERARCHY product_hier PARENT))
    CLASSIFICATION caption VALUE 'Sales Share of Product Parent'
    CLASSIFICATION description VALUE 'Sales Share of Product Parent'
    CLASSIFICATION format_string VALUE '999.99',
sales_share_dept AS
  (SHARE_OF(sales HIERARCHY product_hier LEVEL department))
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
      CLASSIFICATION caption VALUE 'Sales Share of Product Parent'
      CLASSIFICATION description VALUE 'Sales Share of Product Parent'
      CLASSIFICATION format_string VALUE '999.99',
  sales_share_geog_parent AS
   (SHARE_OF(sales HIERARCHY geography_hier PARENT))
      CLASSIFICATION caption VALUE 'Sales Share of Geography Parent'
      CLASSIFICATION description VALUE 'Sales Share of Geography Parent'
      CLASSIFICATION format_string VALUE '999.99',
  sales_share_region AS
    (SHARE_OF(sales HIERARCHY geography_hier LEVEL region))
      CLASSIFICATION caption VALUE 'Sales Share of Geography Parent'
      CLASSIFICATION description VALUE 'Sales Share of Geography Parent'
      CLASSIFICATION format_string VALUE '999.99'
  )
DEFAULT MEASURE SALES;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

26

# Attribute Dimension and Hierarchy Objects

Attribute dimensions reference data sources and specify attributes and levels; hierarchies organize levels hierarchically.

Attribute dimensions and hierarchies are described in the following topics:


- About Attribute Dimensions and Hierarchies
- Attributes and Hierarchical Attributes
- Order Levels
- Level Keys
- Determine Attribute Relationships


## 26.1 About Attribute Dimensions and Hierarchies

An attribute dimension specifies a data source, attributes, and levels; a hierarchy organizes the levels hierarchically.

An attribute dimension specifies the data source it is using and specifies columns of that source as its attributes. It specifies levels for some or all of the attributes and determines attribute relationships between levels.

A hierarchy defines the hierarchical relationships between the levels of an attribute dimension. Attribute dimensions and hierarchies provide the dimension members for analytic view objects.

Most metadata related to dimensions and hierarchies is defined in the attribute dimension. A hierarchy inherits all of the metadata of the attribute dimension it uses. This allows the metadata for attributes and levels to be reused in many hierarchies, promoting consistency and simplifying the definition of the hierarchy.

**About Attribute Dimensions**

An attribute dimension has the following characteristics:

- A data source, which is typically a star schema or snowflake schema dimension table but may be a denormalized table, a view or an external or remote table; each column of the dimension table may be presented in a hierarchy
- A dimension type, which is either `STANDARD` or `TIME`
- Attributes, which are columns from the data source
- Levels, which represent groups of values that are all at the same level of aggregation
- Hierarchical attributes, which are used by hierarchies to describe hierarchical relationships between levels
- An implicit ALL level with only one member, which is the highest level in any hierarchy that uses the attribute dimension
- Can be used by any number of hierarchies

An attribute dimension also has the following optional characteristics:



Copyright © 2001, 2025, Oracle and/or its affiliates.

- Can specify sharing its metadata or its metadata and data with an application container
- Can specify the ordering of level members
- Can specify classifications for the attribute dimension itself, its attributes, some of its hierarchical attributes, its levels, and the ALL member; the classifications provide metadata that an application can use in queries and in presenting query results

The attributes determined by the included levels specify the attributes that become columns in the hierarchy, and, therefore, of any analytic view that references the hierarchy.

### About Attribute Dimension and Level Types

An attribute dimension can be either a `STANDARD` or a `TIME` type. Functionally, the `STANDARD` and `TIME` type attribute dimensions are the same. However, each level of a `TIME` type attribute dimension must specify a level type, even though the values of the level members are not necessarily of that type. For example, a `TIME` type attribute dimension could have a level named SEASON that has a level type of `QUARTERS`, even though its values are the names of seasons. You can use the level types for whatever purpose you choose.

The levels of a `STANDARD` type attribute dimension are of type `STANDARD`. You do not need to specify a level type for the levels of a `STANDARD` type attribute dimension.

The levels of a `TIME` type attribute dimension must be one of the following level types:

- `YEARS`
- `HALF_YEARS`
- `QUARTERS`
- `MONTHS`
- `WEEKS`
- `DAYS`
- `HOURS`
- `MINUTES`
- `SECONDS`

### About Hierarchies

A hierarchy has the following characteristics:

- An attribute dimension
- A hierarchical ordering of levels of the attribute dimension
- Columns for each attribute, including determined attributes, of the levels
- Columns for its hierarchical attributes
- A row for each member of each level of the hierarchy and a row for an implicit ALL level, which represents a single top-level aggregate value
- Metadata it inherits from the attribute dimension
- May be used in the `FROM` clause of a SQL `SELECT` statement.

A hierarchy also has the following optional characteristics:

- Can specify sharing its metadata or its metadata and data with an application container
- Can specify classifications for itself and for its hierarchical attributes


Copyright © 2001, 2025, Oracle and/or its affiliates.

### Example 26-1 A Simple Attribute Dimension

An attribute dimension may be as simple as a list of attributes and levels defined only with key attributes. This example creates an attribute dimension that specifies as attributes only the YEAR_ID, QUARTER_ID, and MONTH_ID columns from the TIME_DIM table.

```
CREATE OR REPLACE ATTRIBUTE DIMENSION time_attr_dim
DIMENSION TYPE TIME
USING time_dim       -- References the TIME_DIM table
ATTRIBUTES           -- A list of table columns to be used as attributes
 (year_id,
  quarter_id,
  month_id)
LEVEL MONTH          -- A level
  LEVEL TYPE MONTHS  -- The level type
  KEY month_id       -- Attribute with unique values
LEVEL QUARTER
  LEVEL TYPE QUARTERS
  KEY quarter_id
LEVEL YEAR
  LEVEL TYPE YEARS
  KEY year_id;
```

For a description of the TIME_DIM table, see About the Data and Scripts for Examples.

Each of the _ID columns in the TIME_DIM table is included in the attribute list. By default, the name of the attribute is the dimension table column name. You can provide a different name for the attribute by using the `AS` *`alias`* clause in the definition.

Levels are created for each attribute using the `KEY` property, which is the only required property for a level.

### Example 26-2 A Simple Hierarchy

```
CREATE OR REPLACE HIERARCHY time_hier  -- Hierarchy name
USING time_attr_dim      -- Refers to the TIME_ATTR_DIM attribute dimension
 (month CHILD OF         -- Levels in the attribute dimension
  quarter CHILD OF
  year);
```

The hierarchy has columns for each attribute of the attribute dimension and for its hierarchical attributes.

```
SELECT column_name from ALL_HIER_COLUMNS WHERE HIER_NAME = 'TIME_HIER';
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

| | COLUMN_NAME |
|---|---|
| 1 | DEPTH |
| 2 | HIER_ORDER |
| 3 | IS_LEAF |
| 4 | LEVEL_NAME |
| 5 | MEMBER_CAPTION |
| 6 | MEMBER_DESCRIPTION |
| 7 | MEMBER_NAME |
| 8 | MEMBER_UNIQUE_NAME |
| 9 | MONTH_ID |
| 10 | PARENT_LEVEL_NAME |
| 11 | PARENT_UNIQUE_NAME |
| 12 | QUARTER_ID |
| 13 | YEAR_ID |

The following selects the attribute columns and some of the hierarchical columns from TIME_HIER when TIME_ATTR_DIM is the attribute dimension defined in Example 26-1.

```
SELECT year_id, quarter_id, month_id,
       member_name, member_unique_name
       member_caption, member_description
  FROM time_hier
  ORDER BY hier_order;
```

An excerpt from the query results are:

| | YEAR_ID | QUARTER_ID | MONTH_ID | MEMBER_NAME | MEMBER_UNIQUE_NAME | MEMBER_CAPTION | MEMBER_DESCRIPTION |
|---|---|---|---|---|---|---|---|
| 1 | (null) | (null) | (null) | ALL | [ALL].[ALL] | (null) | (null) |
| 2 | 11 | (null) | (null) | 11 | [YEAR].&[11] | (null) | (null) |
| 3 | 11 | 111 | (null) | 111 | [QUARTER].&[11]&[111] | (null) | (null) |
| 4 | 11 | 111 | Feb-11 | Feb-11 | [MONTH].&[11]&[111]&[Feb-11] | (null) | (null) |
| 5 | 11 | 111 | Jan-11 | Jan-11 | [MONTH].&[11]&[111]&[Jan-11] | (null) | (null) |
| 6 | 11 | 111 | Mar-11 | Mar-11 | [MONTH].&[11]&[111]&[Mar-11] | (null) | (null) |
| 7 | 11 | 211 | (null) | 211 | [QUARTER].&[11]&[211] | (null) | (null) |
| 8 | 11 | 211 | Apr-11 | Apr-11 | [MONTH].&[11]&[211]&[Apr-11] | (null) | (null) |
| 9 | 11 | 211 | Jun-11 | Jun-11 | [MONTH].&[11]&[211]&[Jun-11] | (null) | (null) |
| 10 | 11 | 211 | May-11 | May-11 | [MONTH].&[11]&[211]&[May-11] | (null) | (null) |
| 11 | 11 | 311 | (null) | 311 | [QUARTER].&[11]&[311] | (null) | (null) |
| 12 | 11 | 311 | Aug-11 | Aug-11 | [MONTH].&[11]&[311]&[Aug-11] | (null) | (null) |
| 13 | 11 | 311 | Jul-11 | Jul-11 | [MONTH].&[11]&[311]&[Jul-11] | (null) | (null) |
| 14 | 11 | 311 | Sep-11 | Sep-11 | [MONTH].&[11]&[311]&[Sep-11] | (null) | (null) |

# 26.2 Attributes and Hierarchical Attributes

Attribute dimension attributes typically reference columns from a source table or view. Hierarchical attributes provide information about the members of a hierarchy.

In an attribute dimension, attributes specify the columns of the source table or view to reference. The default name of the attribute is the name of the table column. You may provide a different name for an attribute by using syntax similar to SQL SELECT clause aliases. You define levels using attributes and you define the relationships between attributes using levels. Attributes appear as columns in hierarchies, depending on the levels that the hierarchy includes and on the defined attribute relationships of the levels.


Copyright © 2001, 2025, Oracle and/or its affiliates.

The hierarchical attributes are the following:

- `DEPTH` is the level depth of the hierarchy member; the ALL level is at depth 0 (zero)
- `HIER_ORDER` is the order of the member in the hierarchy
- `IS_LEAF` is a boolean value that indicates whether the member is at the lowest (leaf) level of the hierarchy
- `LEVEL_NAME` is the name of the level in the definition of the attribute dimension
- `MEMBER_NAME` is the name of the member in the definition of the attribute dimension
- `MEMBER_CAPTION` is `NULL` unless you specify values for it in the definition of the attribute dimension or the hierarchy
- `MEMBER_DESCRIPTION` is `NULL` unless you specify values for it in the definition of the attribute dimension or the hierarchy
- `MEMBER_UNIQUE_NAME` is a name that is guaranteed to be unique in the hierarchy; it is a concatenation of level name, ancestors, and key attribute values
- `PARENT_LEVEL_NAME` is the name of level that is the parent of the current member
- `PARENT_UNIQUE_NAME` is the `MEMBER_UNIQUE_NAME` of the parent of the current member

The hierarchical attribute value is composed of the level and the lineage. The lineage includes the member's key value. Each component of the lineage is enclosed in square brackets, and the components are separated by periods. If a component value contains a right square bracket, it is represented using two right square brackets.

**Example 26-3 Providing Values for Some Hierarchical Attributes**

This is the excerpt from the results of the query of the hierarchy based on the simple attribute dimension in About Attribute Dimensions and Hierarchies.

| | YEAR_ID | QUARTER_ID | MONTH_ID | MEMBER_NAME | MEMBER_UNIQUE_NAME | MEMBER_CAPTION | MEMBER_DESCRIPTION |
|---|---|---|---|---|---|---|---|
| 1 | (null) | (null) | (null) | ALL | [ALL].[ALL] | (null) | (null) |
| 2 | 11 | (null) | (null) | 11 | [YEAR].&[11] | (null) | (null) |
| 3 | 11 | 111 | (null) | 111 | [QUARTER].&[11]&[111] | (null) | (null) |
| 4 | 11 | 111 | Feb-11 | Feb-11 | [MONTH].&[11]&[111]&[Feb-11] | (null) | (null) |
| 5 | 11 | 111 | Jan-11 | Jan-11 | [MONTH].&[11]&[111]&[Jan-11] | (null) | (null) |
| 6 | 11 | 111 | Mar-11 | Mar-11 | [MONTH].&[11]&[111]&[Mar-11] | (null) | (null) |
| 7 | 11 | 211 | (null) | 211 | [QUARTER].&[11]&[211] | (null) | (null) |
| 8 | 11 | 211 | Apr-11 | Apr-11 | [MONTH].&[11]&[211]&[Apr-11] | (null) | (null) |
| 9 | 11 | 211 | Jun-11 | Jun-11 | [MONTH].&[11]&[211]&[Jun-11] | (null) | (null) |
| 10 | 11 | 211 | May-11 | May-11 | [MONTH].&[11]&[211]&[May-11] | (null) | (null) |
| 11 | 11 | 311 | (null) | 311 | [QUARTER].&[11]&[311] | (null) | (null) |
| 12 | 11 | 311 | Aug-11 | Aug-11 | [MONTH].&[11]&[311]&[Aug-11] | (null) | (null) |
| 13 | 11 | 311 | Jul-11 | Jul-11 | [MONTH].&[11]&[311]&[Jul-11] | (null) | (null) |
| 14 | 11 | 311 | Sep-11 | Sep-11 | [MONTH].&[11]&[311]&[Sep-11] | (null) | (null) |

While this hierarchy is functional, it lacks some important features. Note that the MEMBER_NAME column might not be easily readable, and the MEMBER_CAPTION and MEMBER_DESCRIPTION columns do not return data.

This new definition of the time_attr_dim attribute dimension includes the _NAME columns from the TIME_DIM table. In the definitions of the levels, it specifies attributes that contain values for the hierarchical attributes `MEMBER_NAME`, `MEMBER_CAPTION`, and `MEMBER_DESCRIPTION`. This definition provides a hierarchy that uses the attribute dimension with descriptive values for the level members.

```
CREATE OR REPLACE ATTRIBUTE DIMENSION time_attr_dim
DIMENSION TYPE TIME
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
USING time_dim
ATTRIBUTES
 (year_id,
  year_name,
  quarter_id,
  quarter_name,
  month_id,
  month_name,
  month_long_name)
LEVEL MONTH
  LEVEL TYPE MONTHS
  KEY month_id
  MEMBER NAME month_name
  MEMBER CAPTION month_name
  MEMBER DESCRIPTION month_long_name
LEVEL QUARTER
  LEVEL TYPE QUARTERS
  KEY quarter_id
  MEMBER NAME quarter_name
  MEMBER CAPTION quarter_name
  MEMBER DESCRIPTION quarter_name
LEVEL YEAR
  LEVEL TYPE YEARS
  KEY year_id
  MEMBER NAME year_name
  MEMBER CAPTION year_name
  MEMBER DESCRIPTION year_name;
```

This statement selects the attribute columns and some of the hierarchical columns from the TIME_HIER hierarchy.

```
SELECT year_id, quarter_id, month_id,
       member_name, member_unique_name,
       member_caption, member_description
  FROM time_hier
  ORDER BY hier_order;
```

An excerpt from the query results are:

| | YE... | QUARTER_ID | MONTH_ID | MEMBER_NAME | MEMBER_UNIQUE_NAME | MEMBER_CAPTION | MEMBER_DESCRIPTION |
|---|---|---|---|---|---|---|---|
| 1 | (null) | (null) | (null) | ALL | [ALL].[ALL] | (null) | (null) |
| 2 | 11 | (null) | (null) | CY2011 | [YEAR].&[11] | CY2011 | CY2011 |
| 3 | 11 | 111 | (null) | Q1CY2011 | [QUARTER].&[11]&[111] | Q1CY2011 | Q1CY2011 |
| 4 | 11 | 111 | Feb-11 | Feb-11 | [MONTH].&[11]&[111]&[Feb-11] | Feb-11 | February 2011 |
| 5 | 11 | 111 | Jan-11 | Jan-11 | [MONTH].&[11]&[111]&[Jan-11] | Jan-11 | January 2011 |
| 6 | 11 | 111 | Mar-11 | Mar-11 | [MONTH].&[11]&[111]&[Mar-11] | Mar-11 | March 2011 |
| 7 | 11 | 211 | (null) | Q2CY2011 | [QUARTER].&[11]&[211] | Q2CY2011 | Q2CY2011 |
| 8 | 11 | 211 | Apr-11 | Apr-11 | [MONTH].&[11]&[211]&[Apr-11] | Apr-11 | April 2011 |
| 9 | 11 | 211 | Jun-11 | Jun-11 | [MONTH].&[11]&[211]&[Jun-11] | Jun-11 | June 2011 |
| 10 | 11 | 211 | May-11 | May-11 | [MONTH].&[11]&[211]&[May-11] | May-11 | May 2011 |
| 11 | 11 | 311 | (null) | Q3CY2011 | [QUARTER].&[11]&[311] | Q3CY2011 | Q3CY2011 |
| 12 | 11 | 311 | Aug-11 | Aug-11 | [MONTH].&[11]&[311]&[Aug-11] | Aug-11 | August 2011 |
| 13 | 11 | 311 | Jul-11 | Jul-11 | [MONTH].&[11]&[311]&[Jul-11] | Jul-11 | July 2011 |
| 14 | 11 | 311 | Sep-11 | Sep-11 | [MONTH].&[11]&[311]&[Sep-11] | Sep-11 | September 2011 |


Copyright © 2001, 2025, Oracle and/or its affiliates.

The ordering of time periods is not yet correct for reporting on time series calculations; for example, February comes before January. For an example of specifying a sort order for a level, see Order Levels.

# 26.3 Order Levels

You can specify the order of attribute dimension level members.

You may use the `ORDER BY` clause of an attribute dimension level definition to specify an order for members of the level. By default, values of an attribute dimension level are sorted alphabetically by the `MEMBER_NAME` value. If you do not specify a member name, the level is ordered by its `KEY` attribute value.

The `ORDER BY` clause also specifies whether `NULL` values are first or last in the order. You may specify `MIN` or `MAX` *expression* if the attribute is not determined by the level, with the default being `MIN`.

**Example 26-4 Add End Dates**

This example adds end date attributes to the definition of the time_attr_dim attribute dimension.

```
CREATE OR REPLACE ATTRIBUTE DIMENSION time_attr_dim
DIMENSION TYPE TIME
USING time_dim
ATTRIBUTES
 (year_id,
  year_name,
  year_end_date,
  quarter_id,
  quarter_name,
  quarter_end_date,
  month_id,
  month_name,
  month_long_name,
  month_end_date)
LEVEL MONTH
  KEY month_id
  MEMBER NAME month_name
  MEMBER CAPTION month_name
  MEMBER DESCRIPTION month_long_name
  ORDER BY month_end_date
LEVEL QUARTER
  KEY quarter_id
  MEMBER NAME quarter_name
  MEMBER CAPTION quarter_name
  MEMBER DESCRIPTION quarter_name
  ORDER BY quarter_end_date
LEVEL YEAR
  KEY year_id
  MEMBER NAME year_name
  MEMBER CAPTION year_name
  MEMBER DESCRIPTION year_name
  ORDER BY year_end_date;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

This is the definition of the time_hier hierarchy.

```
CREATE OR REPLACE HIERARCHY time_hier
USING time_attr_dim
 (month CHILD OF
  quarter CHILD OF
  year);
```

This query includes the hierarchy order attribute.

```
SELECT year_id,
  quarter_id,
  month_id,
  member_name,
  hier_order
FROM time_hier
ORDER BY hier_order;
```

This is an excerpt from the query results.

| | YEAR_ID | QUARTER_ID | MONTH_ID | MEMBER_NAME | HIER_ORDER |
|---|---|---|---|---|---|
| 1 | (null) | (null) | (null) | ALL | 0 |
| 2 | 11 | (null) | (null) | CY2011 | 1 |
| 3 | 11 | 111 | (null) | Q1CY2011 | 2 |
| 4 | 11 | 111 | Jan-11 | Jan-11 | 3 |
| 5 | 11 | 111 | Feb-11 | Feb-11 | 4 |
| 6 | 11 | 111 | Mar-11 | Mar-11 | 5 |
| 7 | 11 | 211 | (null) | Q2CY2011 | 6 |
| 8 | 11 | 211 | Apr-11 | Apr-11 | 7 |
| 9 | 11 | 211 | May-11 | May-11 | 8 |
| 10 | 11 | 211 | Jun-11 | Jun-11 | 9 |
| 11 | 11 | 311 | (null) | Q3CY2011 | 10 |
| 12 | 11 | 311 | Jul-11 | Jul-11 | 11 |
| 13 | 11 | 311 | Aug-11 | Aug-11 | 12 |
| 14 | 11 | 311 | Sep-11 | Sep-11 | 13 |

The level members are now sorted by end dates.

# 26.4 Level Keys

A level key attribute specifies the data source of the level members.

An attribute dimension level specifies key and optional alternate key attributes that provide the members of the level.

A level must have a key, which is defined by a single attribute, or by multiple attributes for a compound key. Each distinct value for the key defines an attribute dimension member at that level.

A level can also have one or more alternate keys. An alternate key must have a one-to-one relationship with the level key: an attribute specified as an alternate key must have a unique value for every member of the level key attribute.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**Example 26-5 Create the PRODUCT_ATTR_DIM Attribute Dimension**

This example creates the product_attr_dim attribute dimension. The level clauses specify keys and alternate keys.

```
CREATE OR REPLACE ATTRIBUTE DIMENSION product_attr_dim
USING product_dim
ATTRIBUTES
 (department_id,
  department_name,
  category_id,
  category_name)
LEVEL DEPARTMENT
  KEY department_id
  ALTERNATE KEY department_name
  MEMBER NAME department_name
  MEMBER CAPTION department_name
  ORDER BY department_name
LEVEL CATEGORY
  KEY category_id
  ALTERNATE KEY category_name
  MEMBER NAME category_name
  MEMBER CAPTION category_name
  ORDER BY category_name
  DETERMINES(department_id)
ALL MEMBER NAME 'ALL PRODUCTS';
```

# 26.5 Determine Attribute Relationships

You can specify that an attribute of a level determines the values of other attributes.

You can use the `DETERMINES` clause of an attribute dimension level definition to specify a relationship between the level key attribute and other attributes. When there is only one value of an attribute for each value of another attribute, the value of one attribute *determines* the value of another. For example, there is only one value of QUARTER_ID for each value of MONTH_ID; MONTH_ID determines QUARTER_ID.

An attribute determined by a level is included in a hierarchy that uses the attribute dimension. An attribute specified in a `DETERMINES` clause can have the same value for different level members. A level implicitly determines its key and alternate key attributes, although, unlike the attributes in a `DETERMINES` clause, those attributes must have unique values.

The relationships specified by a `DETERMINES` clause can do the following:

- Change the number of rows returned by a hierarchy
- Control whether certain attributes return data for certain rows
- Simplify the SQL that is generated when an analytic view is queried

Specifying determined attributes helps a hierarchy or analytic view to determine a unique value for a member. If an attribute is determined by a level, you do not need to explicitly specify in a query the attribute value that identifies the relationship of the determined attribute to the hierarchy member. For example, a `QUALIFY` calculation requires a uniquely identified hierarchy member. If you omit attributes from a `DETERMINES` clause, then in an analytic view measure that uses a `QUALIFY` calculation, you must explicitly specify those attributes to identify the unique member.


Copyright © 2001, 2025, Oracle and/or its affiliates.

The relationship of determined attributes to key and alternate key attributes is not validated or enforced in an attribute dimension or in a hierarchy that uses the attribute dimension. To validate the relationship, use the PL/SQL procedure DBMS_HIERARCHY.VALIDATE_HIERARCHY, which inspects the data in the source table or view.

### Usage Notes

When using a DETERMINES clause, consider the following:

- Include in a DETERMINES clause the KEY attribute of a parent level in a hierarchy whenever the key of the lower level determines the value of the parent level. Lower levels inherit the determined attributes of ancestor levels; therefore, it is a good practice to include the key attribute value of the parent level in the DETERMINES clause of the lower level.
- Values of the MEMBER NAME, MEMBER CAPTION, MEMBER DESCRIPTION, and ORDER BY properties are assumed to be determined by the KEY attribute value. You do not need to include attributes for those properties in a DETERMINES clause. You should be sure, however, that the data for those attributes has only one value for each value of the KEY attribute.

### Example 26-6 Add DETERMINES Clauses

This example adds the DETERMINES clause to the levels of time_attr_dim.

```
CREATE OR REPLACE ATTRIBUTE DIMENSION time_attr_dim
DIMENSION TYPE TIME
USING time_dim
ATTRIBUTES
 (year_id,
  year_name,
  year_end_date,
  quarter_id,
  quarter_name,
  quarter_end_date,
  month_id,
  month_name,
  month_long_name,
  month_end_date)
LEVEL MONTH
  LEVEL TYPE MONTHS
  KEY month_id
  MEMBER NAME month_name
  MEMBER CAPTION month_name
  MEMBER DESCRIPTION month_long_name
  ORDER BY month_end_date
  DETERMINES (quarter_id)
LEVEL QUARTER
  LEVEL TYPE QUARTERS
  KEY quarter_id
  MEMBER NAME quarter_name
  MEMBER CAPTION quarter_name
  MEMBER DESCRIPTION quarter_name
  ORDER BY quarter_end_date
  DETERMINES (year_id)
LEVEL YEAR
  LEVEL TYPE YEARS
  KEY year_id
  MEMBER NAME year_name
  MEMBER CAPTION year_name
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
  MEMBER DESCRIPTION year_name
  ORDER BY year_end_date;
```

Select the LEVEL_NAME, _ID, and MEMBER_UNIQUE_NAME columns from the TIME_HIER hierarchy.

```
SELECT level_name,
  year_id,
  quarter_id,
  month_id,
  member_unique_name
FROM time_hier
ORDER BY hier_order;
```

The hierarchy now knows the relationship between the months, quarters, and years attributes, as shown in the following results of the preceding query. The `MEMBER_UNIQUE_NAME` values are now created from only the level name and the `KEY` attribute value; they no longer must include the full lineage as seen in Example 26-3.

| | LEVEL_NAME | YEAR_ID | QUARTER_ID | MONTH_ID | MEMBER_UNIQUE_NAME |
|---|---|---|---|---|---|
| 1 | ALL | (null) | (null) | (null) | [ALL].[ALL] |
| 2 | YEAR | 11 | (null) | (null) | [YEAR].&[11] |
| 3 | QUARTER | 11 | 111 | (null) | [QUARTER].&[111] |
| 4 | MONTH | 11 | 111 | Jan-11 | [MONTH].&[Jan-11] |
| 5 | MONTH | 11 | 111 | Feb-11 | [MONTH].&[Feb-11] |
| 6 | MONTH | 11 | 111 | Mar-11 | [MONTH].&[Mar-11] |
| 7 | QUARTER | 11 | 211 | (null) | [QUARTER].&[211] |
| 8 | MONTH | 11 | 211 | Apr-11 | [MONTH].&[Apr-11] |
| 9 | MONTH | 11 | 211 | May-11 | [MONTH].&[May-11] |
| 10 | MONTH | 11 | 211 | Jun-11 | [MONTH].&[Jun-11] |
| 11 | QUARTER | 11 | 311 | (null) | [QUARTER].&[311] |
| 12 | MONTH | 11 | 311 | Jul-11 | [MONTH].&[Jul-11] |
| 13 | MONTH | 11 | 311 | Aug-11 | [MONTH].&[Aug-11] |
| 14 | MONTH | 11 | 311 | Sep-11 | [MONTH].&[Sep-11] |


Copyright © 2001, 2025, Oracle and/or its affiliates.

27

# Analytic View Objects

An analytic view is a type of view that you can use to easily extend the content of a star schema, snowflake schema, or a flat (denormalized) fact table with aggregated data, measure calculations and descriptive metadata, and to simplify the SQL needed to access data.

Analytic views are described in the following topics.


- About Analytic Views
- Measures of Analytic Views
- Create Analytic Views
- Examples of Calculated Measures
- Attribute Reporting
- Analytic View Queries with Filtered Facts and Added Measures


## 27.1 About Analytic Views

Analytic views layer a hierarchical/dimensional model over data.
Analytic views are defined over the dimension tables and the fact table of a star or snowflake schema. You can also define an analytic view over a denormalized table, in which dimension attributes and fact data are in the same table. Hierarchies are defined over dimension tables. An analytic view references hierarchies and a fact table.

Even though an analytic view is defined over data modeled as a star schema, the data does not need to be stored in a star schema. You can use views to represent other forms of stored data to an analytic view. Generally, if the tables or views perform well with a star style query they work well with analytic views. Smaller data sets might work well with views. Larger data sets might perform better with tables in a star schema. The most performant schema is a star schema loaded into the in-memory column store, using the Oracle Database In-Memory Option.

When used with the in-memory column store, analytic views optimize the SQL execution plan to take advantage of In-Memory Aggregation (that is, the vector transform execution plan). Analytic views can take advantage of materialized views to further accelerate aggregate level queries (note that materialized views can be loaded into the in-memory column store).

The minimum requirements for an analytic view include the following:

- A dimension table (or view). This table should have a primary key that provides a unique list of values and that joins to the fact table.
- A fact table with at least one fact (measure) column and a key column that joins to the primary key of the dimension table.

More typically, an analytic view has the following characteristics:

- Is defined over using two or more dimension tables, which enables the ability to slice and dice data.
- One or more of the dimension tables contain data at different levels of aggregation (for example: days, months, quarters, and years).



Copyright © 2001, 2025, Oracle and/or its affiliates.

Analytic views comprise three types of objects: attribute dimensions, hierarchies, and analytic views.

An attribute dimension is a metadata object that references tables or views and organizes columns into higher-level objects such as attributes and levels. Most metadata related to dimensions and hierarchies is defined in the attribute dimension object.

A hierarchy is a type of view. Hierarchies reference attribute dimension objects. Hierarchies organize data using hierarchical relationships between the hierarchy members. Queries of a hierarchy return detail and aggregate-level keys ("hierarchy values") and attributes of those values.

An analytic view is a type of view that returns fact data. Analytic views reference both fact tables and hierarchies. Both hierarchy and measure data is selected from analytic views.

## 27.2 Measures of Analytic Views

Analytic view measures specify fact data and the calculations or other operations to perform on the data.

In an analytic view definition, you may specify one or more base measures and calculated measures.

### Base Measures

A base measure is a reference to a column in a fact table. You may optionally specify a `meas_aggregate_clause`, which overrides the default aggregation method of the analytic view. Each base measure may specify a default aggregation. The aggregation may be a simple operation like `SUM` or `AVG`, or a complex nesting of operations that vary by attribute dimension.

You can use the `default_aggregate_clause` to specify a default aggregation method for base measures that don't have a `meas_aggregate_clause`. The default value of the `default_aggregate_clause` is `SUM`.

### Calculated Measures

A calculated measure is an expression that can be a user-defined expression or one of the many pre-defined analytic calculations. A calculated measure expression may include other measures, row functions, and hierarchy functions. Hierarchy functions allow computations based on identifying and processing related members in a hierarchy. The expression may reference other measures in the analytic view, but may not reference fact columns. Because a calculation can refer to other measures, you can easily build complex calculations through nesting.

In defining a calculated measure expression, you may use any other measure in the analytic view, irrespective of the order in which you defined the measures of the analytic view. The only restriction is that no cycles may be introduced in the calculations.

In addition to using calculated measures in the definition of an analytic view, you can add calculated measures in a `SELECT` statement that queries an analytic view. To do so, you use the `ADD MEASURES` keywords in the `WITH` or `FROM` clauses of the statement. The syntax of a calculated measure is the same whether it is in the definition of the analytic view or in a `SELECT` statement.

Categories of calculated measure expressions are the following:

- Analytic view measure expressions
- Analytic view hierarchical expressions


Copyright © 2001, 2025, Oracle and/or its affiliates.

- Simple expressions
- Single row function expressions
- Compound expressions
- Datetime expressions
- Interval expressions

Analytic view measure expressions include the following operations:

- Lead and lag
- Qualified data reference (QDR)
- Rank
- Related member
- Share of
- Window calculations

**Related Topics**

- Analytic View Query with Added Measures

# 27.3 Create Analytic Views

In creating an analytic view, you specify one or more hierarchies and a fact table that has at least one measure column that can join to each hierarchy.

### Create a Simple Analytic View

An analytic view must have a reference to a fact table and a measure that can join to a hierarchy.

**Example 27-1 Creating a Simple Analytic View**

This analytic view uses the TIME_HIER hierarchy and the SALE_FACT table. It contains a single measure, SALES.

```
CREATE OR REPLACE ANALYTIC VIEW sales_av
USING sales_fact                       -- Refers to the SALES_FACT table
DIMENSION BY                           -- List of attribute dimensions
  (time_attr_dim                       -- TIME_ATTR_DIM attribute dimension
    KEY month_id REFERENCES month_id   -- Dimension key joins to fact column
    HIERARCHIES (                      -- List of hierarchies that use
      time_hier DEFAULT))              -- the attribute dimension
MEASURES                               -- List of measures
 (sales FACT sales)                    -- SALES measure references SALES column
DEFAULT MEASURE SALES;                 -- Default measure of the analytic view
```

A query that selects from an analytic view that does not include filters has the potential of returning large numbers of rows. However, in this query, the SALES_AV analytic view includes a single hierarchy that returns only 86 rows.

```
SELECT *
  FROM sales_av HIERARCHIES(time_hier)
  ORDER BY time_hier.hier_order;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

This is a excerpt of the returned values.

| MEMBER_DESCRIPTION | LEVEL_NAME | HIER_ORDER | DEPTH | IS_LEAF | PARENT_LEVEL_NAME | PARENT_UNIQUE_NAME | SALES |
|---|---|---|---|---|---|---|---|
| (null) | ALL | 0 | 0 | 0 | (null) | (null) | 36418586335.29 |
| CY2011 | YEAR | 1 | 1 | 0 | ALL | [ALL].[ALL] | 6755115980.73 |
| Q1CY2011 | QUARTER | 2 | 2 | 0 | YEAR | [YEAR].&[11] | 1625299627.35 |
| January 2011 | MONTH | 3 | 3 | 1 | QUARTER | [QUARTER].&[111] | 545626198.98 |
| February 2011 | MONTH | 4 | 3 | 1 | QUARTER | [QUARTER].&[111] | 516587219 |
| March 2011 | MONTH | 5 | 3 | 1 | QUARTER | [QUARTER].&[111] | 563086209.37 |
| Q2CY2011 | QUARTER | 6 | 2 | 0 | YEAR | [YEAR].&[11] | 1715160208.04 |
| April 2011 | MONTH | 7 | 3 | 1 | QUARTER | [QUARTER].&[211] | 556371561.43 |
| May 2011 | MONTH | 8 | 3 | 1 | QUARTER | [QUARTER].&[211] | 583962050.22 |
| June 2011 | MONTH | 9 | 3 | 1 | QUARTER | [QUARTER].&[211] | 574826596.39 |
| Q3CY2011 | QUARTER | 10 | 2 | 0 | YEAR | [YEAR].&[11] | 1691017692.94 |

### Add Another Base Measure

To add another base measure to an analytic view, include the measure in the `MEASURES` list.

**Example 27-2 Adding a Base Measure to an Analytic View**

```
CREATE OR REPLACE ANALYTIC VIEW sales_av
USING sales_fact
DIMENSION BY
  (time_attr_dim
    KEY month_id REFERENCES month_id
    HIERARCHIES (
      time_hier DEFAULT))
MEASURES
 (sales FACT sales,
  units FACT units)                    -- Add the UNITS base measure
DEFAULT MEASURE SALES;
```

Because a query of the analytic view could return a great many rows, a query typically uses filters to limit the results. In the `WHERE` clause, this query filters the time periods to those in the YEAR level, so it returns only SALES and UNITS data at that level.

```
SELECT time_hier.member_name as TIME,
 sales,
 units
FROM
 sales_av HIERARCHIES(time_hier)
WHERE time_hier.level_name = 'YEAR'
ORDER BY time_hier.hier_order;
```

These are the returned values.

| | TIME | SALES | UNITS |
|---|---|---|---|
| 1 | CY2011 | 6755115980.73 | 24462444 |
| 2 | CY2012 | 6901682398.95 | 24400619 |
| 3 | CY2013 | 7240938717.57 | 24407259 |
| 4 | CY2014 | 7579746352.89 | 24402666 |
| 5 | CY2015 | 7941102885.15 | 24475206 |


Copyright © 2001, 2025, Oracle and/or its affiliates.

### Add Hierarchies to an Analytic View

Typically, an analytic view has more than one hierarchy using one or more attribute dimensions.

**Example 27-3 Adding Hierarchies to an Analytic View**

This example adds attribute dimensions and hierarchies to the `DIMENSION BY` list of the analytic view.

```
CREATE OR REPLACE ANALYTIC VIEW sales_av
USING sales_fact
DIMENSION BY
  (time_attr_dim
    KEY month_id REFERENCES month_id
    HIERARCHIES (
      time_hier DEFAULT),
   product_attr_dim
    KEY category_id REFERENCES category_id
    HIERARCHIES (
      product_hier DEFAULT),
   geography_attr_dim
    KEY state_province_id
    REFERENCES state_province_id
    HIERARCHIES (
      geography_hier DEFAULT)
   )
MEASURES
 (sales FACT sales,
  units FACT units
  )
DEFAULT MEASURE sales;
```

The following query adds the PRODUCT_HIER and GEOGRAPHY_HIER hierarchies to the `HIERARCHIES` phrase of the `FROM` clause.

```
SELECT time_hier.member_name AS Time,
 product_hier.member_name AS Product,
 geography_hier.member_name AS Geography,
 sales,
 units
FROM
 sales_av HIERARCHIES (time_hier, product_hier, geography_hier)
WHERE time_hier.level_name in ('YEAR')
  AND product_hier.level_name in ('DEPARTMENT')
  AND geography_hier.level_name in ('REGION')
ORDER BY time_hier.hier_order,
  product_hier.hier_order,
  geography_hier.hier_order;
```

The query returns 50 rows. The following image shows only the first 20 rows.


Copyright © 2001, 2025, Oracle and/or its affiliates.

| | TIME | PRODUCT | GEOGRAPHY | SALES | UNITS |
|---|---|---|---|---|---|
| 1 | CY2011 | Cameras and Camcorders | Africa | 45634583.27 | 179017 |
| 2 | CY2011 | Cameras and Camcorders | Asia | 202690278.2 | 797358 |
| 3 | CY2011 | Cameras and Camcorders | Europe | 30943543.81 | 123376 |
| 4 | CY2011 | Cameras and Camcorders | North America | 74533750.43 | 292386 |
| 5 | CY2011 | Cameras and Camcorders | Oceania | 1475539.04 | 6015 |
| 6 | CY2011 | Cameras and Camcorders | South America | 86722190.7 | 340517 |
| 7 | CY2011 | Computers | Africa | 637720605.62 | 2021863 |
| 8 | CY2011 | Computers | Asia | 2841504727.29 | 8982201 |
| 9 | CY2011 | Computers | Europe | 440217216.49 | 1389905 |
| 10 | CY2011 | Computers | North America | 1043206792.26 | 3296076 |
| 11 | CY2011 | Computers | Oceania | 21628470.68 | 68639 |
| 12 | CY2011 | Computers | South America | 1212972617.95 | 3832038 |
| 13 | CY2011 | Portable Music and Video | Africa | 12006274.15 | 323900 |
| 14 | CY2011 | Portable Music and Video | Asia | 53059837.6 | 1434864 |
| 15 | CY2011 | Portable Music and Video | Europe | 8257455.15 | 222346 |
| 16 | CY2011 | Portable Music and Video | North America | 19477358.02 | 527553 |
| 17 | CY2011 | Portable Music and Video | Oceania | 398916.14 | 10999 |
| 18 | CY2011 | Portable Music and Video | South America | 22665823.93 | 613391 |
| 19 | CY2012 | Cameras and Camcorders | Africa | 46521568.18 | 178894 |
| 20 | CY2012 | Cameras and Camcorders | Asia | 206589367.58 | 795253 |

You can view and run SQL scripts that create the tables, the analytic view component objects, and the queries used in the examples from the Oracle Live SQL website at https://livesql.oracle.com/apex/livesql/file/index.html.

# 27.4 Examples of Calculated Measures

Calculated measures are expressions you add to a `MEASURES` clause of an analytic view in the form of *`measure_name`* `AS (`*`expression`*`)`.

### Add a LAG Expression

This example adds a calculated measure that uses a `LAG` operation to the SALES_AV analytic view.

**Example 27-4 Adding a LAG Expression**

```
CREATE OR REPLACE ANALYTIC VIEW sales_av
USING sales_fact
DIMENSION BY
  (time_attr_dim
    KEY month_id REFERENCES month_id
    HIERARCHIES (
      time_hier DEFAULT),
   product_attr_dim
    KEY category_id REFERENCES category_id
    HIERARCHIES (
      product_hier DEFAULT),
   geography_attr_dim
    KEY state_province_id REFERENCES state_province_id
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
    HIERARCHIES (
      geography_hier DEFAULT)
   )
MEASURES
 (sales FACT sales,
  units FACT units,
  sales_prior_period AS       -- Add a calculated measure.
    (LAG(sales) OVER (HIERARCHY time_hier OFFSET 1))
  )
DEFAULT MEASURE SALES;
```

Select the SALES and SALES_PRIOR_PERIOD measures at the YEAR and QUARTER levels.

```
SELECT time_hier.member_name as TIME,
 sales,
 sales_prior_period
FROM
 sales_av HIERARCHIES(time_hier)
WHERE time_hier.level_name IN ('YEAR','QUARTER')
ORDER BY time_hier.hier_order;
```

In this excerpt from the query results, note that the `LAG` expression returns prior periods within the same level.

| | TIME | SALES | SALES_PRIOR_PERIOD |
|---|---|---|---|
| 1 | CY2011 | 6755115980.73 | (null) |
| 2 | Q1CY2011 | 1625299627.35 | (null) |
| 3 | Q2CY2011 | 1715160208.04 | 1625299627.35 |
| 4 | Q3CY2011 | 1691017692.94 | 1715160208.04 |
| 5 | Q4CY2011 | 1723638452.4 | 1691017692.94 |
| 6 | CY2012 | 6901682398.95 | 6755115980.73 |
| 7 | Q1CY2012 | 1644857783.16 | 1723638452.4 |
| 8 | Q2CY2012 | 1752414181.93 | 1644857783.16 |
| 9 | Q3CY2012 | 1732373411.73 | 1752414181.93 |
| 10 | Q4CY2012 | 1772037022.13 | 1732373411.73 |
| 11 | CY2013 | 7240938717.57 | 6901682398.95 |
| 12 | Q1CY2013 | 1723571457.57 | 1772037022.13 |
| 13 | Q2CY2013 | 1840985832.41 | 1723571457.57 |

### SHARE OF Expressions

Share of measures calculate the ratio of a current row to a parent row, ancestor row, or all rows in the current level; for example, the ratio of a geography member to the parent of the member. Share of measures are specified using the `SHARE OF` expression.


Copyright © 2001, 2025, Oracle and/or its affiliates.

### Example 27-5 Using SHARE OF Expressions

This example adds calculated measures that use `SHARE OF` operations to the SALES_AV analytic view.

```
CREATE OR REPLACE ANALYTIC VIEW sales_av
USING sales_fact
DIMENSION BY
  (time_attr_dim
    KEY month_id REFERENCES month_id
    HIERARCHIES (
      time_hier DEFAULT),
   product_attr_dim
    KEY category_id REFERENCES category_id
    HIERARCHIES (
      product_hier DEFAULT),
   geography_attr_dim
    KEY state_province_id REFERENCES state_province_id
    HIERARCHIES (
      geography_hier DEFAULT)
   )
MEASURES
 (sales FACT sales,
  units FACT units,
  -- Share of calculations
 sales_shr_parent_prod AS
   (SHARE_OF(sales HIERARCHY product_hier PARENT)),
 sales_shr_parent_geog AS
   (SHARE_OF(sales HIERARCHY geography_hier PARENT)),
 sales_shr_region AS
   (SHARE_OF(sales HIERARCHY geography_hier LEVEL REGION))
  )
DEFAULT MEASURE SALES;
```

The SALES_SHR_PARENT_PROD measure calculates the ratio of a SALES value at the CATEGORY or DEPARTMENT level to SALES of the parent in the PRODUCT_HIER hierarchy, such as the ratio of SALES for Total Server Computers to Computers.

This query selects SALES and SALES_SHR_PARENT_PROD measure for CY2014 at each level of the PRODUCT_HIER hierarchy.

```
SELECT time_hier.member_name AS Time,
 product_hier.member_name    AS Product,
 product_hier.level_name     AS Prod_Level,
 sales,
 ROUND(sales_shr_parent_prod,2) AS sales_shr_parent_prod
FROM
 sales_av HIERARCHIES (time_hier, product_hier)
WHERE time_hier.year_name = 'CY2014'
AND time_hier.level_name = 'YEAR'
ORDER BY product_hier.hier_order;
```

The results of the query are:


Copyright © 2001, 2025, Oracle and/or its affiliates.

| | TIME | PRODUCT | PROD_LEVEL | SALES | SALES_SHR_PARENT_PROD |
|---|---|---|---|---|---|
| 1 | CY2014 | ALL PRODUCTS | ALL | 7579746352.89 | (null) |
| 2 | CY2014 | Cameras and Camcorders | DEPARTMENT | 496952312.98 | 0.07 |
| 3 | CY2014 | Camcorders and Accessories | CATEGORY | 154489927.29 | 0.31 |
| 4 | CY2014 | Cameras and Accessories | CATEGORY | 342462385.69 | 0.69 |
| 5 | CY2014 | Computers | DEPARTMENT | 6952712285.9 | 0.92 |
| 6 | CY2014 | All Computer Furniture | CATEGORY | 23214339.8 | 0 |
| 7 | CY2014 | Computer Printers and Supplies | CATEGORY | 1677409104.44 | 0.24 |
| 8 | CY2014 | PDAs | CATEGORY | 7747497.6 | 0 |
| 9 | CY2014 | Total Personal Computers | CATEGORY | 5133182348.24 | 0.74 |
| 10 | CY2014 | Total Server Computers | CATEGORY | 111158995.82 | 0.02 |
| 11 | CY2014 | Portable Music and Video | DEPARTMENT | 130081754.01 | 0.02 |
| 12 | CY2014 | Total iPlayer Family | CATEGORY | 130081754.01 | 1 |

The SALE_SHR_REGION measure calculates the share of SALES at the STATE or COUNTRY levels to SALES at the REGION level, for example, the ratio of SALES for California – US to SALES for North America.

This query returns the values for the SALES and SALES_SHR_REGION measures for year CY2014 and states in the United States.

```
SELECT time_hier.member_name AS Time,
 geography_hier.member_name  AS Geography,
 geography_hier.level_name   AS Geog_Level,
 sales,
 ROUND(sales_shr_region,2)   AS sales_shr_region
FROM
 sales_av HIERARCHIES (time_hier, geography_hier)
WHERE time_hier.year_name = 'CY2014'
AND time_hier.level_name = 'YEAR'
AND geography_hier.country_name = 'United States'
AND geography_hier.level_name = 'STATE_PROVINCE'
ORDER BY geography_hier.hier_order;
```

This is the result of the query.


Copyright © 2001, 2025, Oracle and/or its affiliates.

| | TIME | GEOGRAPHY | GEOG_LEVEL | SALES | SALES_SHR_REGION |
|---|---|---|---|---|---|
| 1 | CY2014 | California - US | STATE_PROVINCE | 10990458.69 | 0.01 |
| 2 | CY2014 | Florida - US | STATE_PROVINCE | 50867372.16 | 0.04 |
| 3 | CY2014 | Georgia - US | STATE_PROVINCE | 57369538.31 | 0.05 |
| 4 | CY2014 | Illinois - US | STATE_PROVINCE | 58648087.2 | 0.05 |
| 5 | CY2014 | Massachusetts - US | STATE_PROVINCE | 41954923.92 | 0.03 |
| 6 | CY2014 | Michigan - US | STATE_PROVINCE | 61579430.16 | 0.05 |
| 7 | CY2014 | Missouri - US | STATE_PROVINCE | 56495320.12 | 0.04 |
| 8 | CY2014 | Nevada - US | STATE_PROVINCE | 31457133.25 | 0.02 |
| 9 | CY2014 | New York - US | STATE_PROVINCE | 49942020.98 | 0.04 |
| 10 | CY2014 | Ohio - US | STATE_PROVINCE | 69715139.36 | 0.05 |
| 11 | CY2014 | Pennsylvania - US | STATE_PROVINCE | 54751342.31 | 0.04 |
| 12 | CY2014 | Rhode Island - US | STATE_PROVINCE | 28485913.48 | 0.02 |
| 13 | CY2014 | Tennessee - US | STATE_PROVINCE | 24783302.86 | 0.02 |
| 14 | CY2014 | Texas - US | STATE_PROVINCE | 44151509.32 | 0.03 |
| 15 | CY2014 | Virginia - US | STATE_PROVINCE | 28255742.07 | 0.02 |
| 16 | CY2014 | Washington - US | STATE_PROVINCE | 47650667.4 | 0.04 |

### QDR Expressions

A `qdr_expression` uses the `QUALIFY` keyword to limit the values of a measure to those for a single dimension member. An example is Sales for the year CY2011 or the percent difference in SALES between the current time period and CY2011. The `QUALIFY` expression refers to a `KEY` attribute value.

### Example 27-6 Using QUALIFY Expressions

Create the SALES_AV analytic view with the SALES_2011 and SALES_PCT_CHG_2011 measures.

```
CREATE OR REPLACE ANALYTIC VIEW sales_av
USING sales_fact
DIMENSION BY
  (time_attr_dim
    KEY month_id REFERENCES month_id
    HIERARCHIES (
      time_hier DEFAULT),
   product_attr_dim
    KEY category_id REFERENCES category_id
    HIERARCHIES (
      product_hier DEFAULT),
   geography_attr_dim
    KEY state_province_id REFERENCES state_province_id
    HIERARCHIES (
      geography_hier DEFAULT)
   )
MEASURES
 (sales FACT sales,
  units FACT units,
 -- Sales for CY2011
 sales_2011 AS
    (QUALIFY (sales, time_hier = year['11'])),
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
  -- Sales percent change from 2011.
  sales_pct_chg_2011 AS
    ((sales - (QUALIFY (sales, time_hier = year['11']))) /
    (QUALIFY (sales, time_hier = year['11'])))
  )
DEFAULT MEASURE SALES;
```

Regardless of filters in the query, the SALES_2011 measure always returns data for the year CY2011. The SALES_PCT_CHG_2011 measure calculates the percent difference between the current time period and CY2011.

This query selects SALES, SALES_2011 and SALES_PCT_CHG_2011 at the YEAR and REGION levels.

```
SELECT time_hier.member_name   AS Time,
 geography_hier.member_name    AS Geography,
 sales,
 sales_2011,
 ROUND(sales_pct_chg_2011,2) as sales_pct_chg_2011
FROM
 sales_av HIERARCHIES (time_hier, geography_hier)
WHERE time_hier.level_name = 'YEAR'
AND geography_hier.level_name = 'REGION'
ORDER BY geography_hier.hier_order,
 time_hier.hier_order;
```

This is an excerpt from the query results. Note that for each row SALES_2011 returns SALES for CY2011.

| | TIME | GEOGRAPHY | SALES | SALES_2011 | SALES_PCT_CHG_2011 |
|---|---|---|---|---|---|
| 1 | CY2011 | Africa | 695361463.04 | 695361463.04 | 0 |
| 2 | CY2012 | Africa | 715142588.19 | 695361463.04 | 0.03 |
| 3 | CY2013 | Africa | 746220583.3 | 695361463.04 | 0.07 |
| 4 | CY2014 | Africa | 781333432.78 | 695361463.04 | 0.12 |
| 5 | CY2015 | Africa | 818560024.79 | 695361463.04 | 0.18 |
| 6 | CY2011 | Asia | 3097254843.09 | 3097254843.09 | 0 |
| 7 | CY2012 | Asia | 3163782733.74 | 3097254843.09 | 0.02 |
| 8 | CY2013 | Asia | 3322778863.3 | 3097254843.09 | 0.07 |
| 9 | CY2014 | Asia | 3479067417.8 | 3097254843.09 | 0.12 |
| 10 | CY2015 | Asia | 3644177245.26 | 3097254843.09 | 0.18 |

# 27.5 Attribute Reporting

You can use any attribute of an attribute dimension in a hierarchy and aggregate data for it in an analytic view.

You can use attributes to filter data or to display in a report. You can also break out (aggregate) data by an attribute. You can create calculated measures in an analytic view using the attribute; the analytic view then provides the aggregate rows for the attribute.


Copyright © 2001, 2025, Oracle and/or its affiliates.

### Example 27-7 Using the SEASON Attribute

This example first creates an attribute dimension that has SEASON and SEASON_ORDER as attributes. This allows a hierarchy and an analytic view to reuse some metadata of those attributes and to relate the attributes to other levels. For example, SEASON is determined by MONTH values.

```
-- Create a time attribute dimension with a SEASON attribute.
CREATE OR REPLACE ATTRIBUTE DIMENSION time_attr_dim
DIMENSION TYPE TIME
USING time_dim
ATTRIBUTES
 (year_id,
  year_name,
  year_end_date,
  quarter_id,
  quarter_name,
  quarter_end_date,
  month_id,
  month_name,
  month_long_name,
  month_end_date,
  season,
  season_order)
LEVEL month
  LEVEL TYPE MONTHS
  KEY month_id
  MEMBER NAME month_name
  MEMBER CAPTION month_name
  MEMBER DESCRIPTION month_long_name
  ORDER BY month_end_date
  DETERMINES (quarter_id, season, season_order)
LEVEL quarter
  LEVEL TYPE QUARTERS
  KEY quarter_id
  MEMBER NAME quarter_name
  MEMBER CAPTION quarter_name
  MEMBER DESCRIPTION quarter_name
  ORDER BY quarter_end_date
  DETERMINES (year_id)
LEVEL year
  LEVEL TYPE YEARS
  KEY year_id
  MEMBER NAME year_name
  MEMBER CAPTION year_name
  MEMBER DESCRIPTION year_name
  ORDER BY year_end_date
LEVEL season
  LEVEL TYPE QUARTERS
  KEY season
  MEMBER NAME season
  MEMBER CAPTION season
  MEMBER DESCRIPTION season
  ORDER BY season_order;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

Create a hierarchy in which MONTH is a child of SEASON.

```
CREATE OR REPLACE HIERARCHY time_season_hier
USING time_attr_dim
  (month CHILD OF
   season);
```

Select data from the TIME_SEASON_HIER hierarchy.

```
SELECT member_name,
  member_unique_name,
  level_name,
  hier_order
FROM time_season_hier
ORDER BY hier_order;
```

In the results of the query, the TIME_SEASON_HIER hierarchy returns rows for the ALL level, SEASONS, and MONTHS. This image captures the first twenty of the rows returned.

| | MEMBER_NAME | MEMBER_UNIQUE_NAME | LEVEL_NAME | HIER_ORDER |
|---|---|---|---|---|
| 1 | ALL | [ALL].[ALL] | ALL | 0 |
| 2 | Spring | [SEASON].&[Spring] | SEASON | 1 |
| 3 | Mar-11 | [MONTH].&[Mar-11] | MONTH | 2 |
| 4 | Apr-11 | [MONTH].&[Apr-11] | MONTH | 3 |
| 5 | May-11 | [MONTH].&[May-11] | MONTH | 4 |
| 6 | Mar-12 | [MONTH].&[Mar-12] | MONTH | 5 |
| 7 | Apr-12 | [MONTH].&[Apr-12] | MONTH | 6 |
| 8 | May-12 | [MONTH].&[May-12] | MONTH | 7 |
| 9 | Mar-13 | [MONTH].&[Mar-13] | MONTH | 8 |
| 10 | Apr-13 | [MONTH].&[Apr-13] | MONTH | 9 |
| 11 | May-13 | [MONTH].&[May-13] | MONTH | 10 |
| 12 | Mar-14 | [MONTH].&[Mar-14] | MONTH | 11 |
| 13 | Apr-14 | [MONTH].&[Apr-14] | MONTH | 12 |
| 14 | May-14 | [MONTH].&[May-14] | MONTH | 13 |
| 15 | Mar-15 | [MONTH].&[Mar-15] | MONTH | 14 |
| 16 | Apr-15 | [MONTH].&[Apr-15] | MONTH | 15 |
| 17 | May-15 | [MONTH].&[May-15] | MONTH | 16 |
| 18 | Summer | [SEASON].&[Summer] | SEASON | 17 |
| 19 | Jun-11 | [MONTH].&[Jun-11] | MONTH | 18 |
| 20 | Jul-11 | [MONTH].&[Jul-11] | MONTH | 19 |

The example next creates an analytic view that provides aggregate data for SEASON.

```
CREATE OR REPLACE ANALYTIC VIEW sales_av
USING sales_fact
DIMENSION BY
  (time_attr_dim
    KEY month_id REFERENCES month_id
    HIERARCHIES (
      time_hier DEFAULT,
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

```
      time_season_hier),
   product_attr_dim
    KEY category_id REFERENCES category_id
    HIERARCHIES (
      product_hier DEFAULT),
   geography_attr_dim
    KEY state_province_id
    REFERENCES state_province_id
    HIERARCHIES (
      geography_hier DEFAULT)
   )
MEASURES
 (sales FACT sales,
  units FACT units
  )
DEFAULT MEASURE SALES;
```

You can now select SALES by YEAR and SEASON directly from the analytic view. This query selects from the TIME_HIER and TIME_SEASON_HIER hierarchies at the YEAR and SEASON levels.

```
SELECT  time_hier.member_name      AS Time,
 time_season_hier.member_name      AS Season,
 ROUND(sales)                      AS Sales
FROM sales_av HIERARCHIES (time_hier, time_season_hier)
WHERE time_hier.level_name = 'YEAR'
  AND time_season_hier.level_name = 'SEASON'
ORDER BY time_hier.hier_order,
  time_season_hier.hier_order;
```

This excerpt from the query results shows the first twelve rows returned.

| | TIME | SEASON | SALES |
|---|---|---|---|
| 1 | CY2011 | Spring | 1703419821 |
| 2 | CY2011 | Summer | 1708263225 |
| 3 | CY2011 | Fall | 1708053033 |
| 4 | CY2011 | Winter | 1635379902 |
| 5 | CY2012 | Spring | 1740590320 |
| 6 | CY2012 | Summer | 1749187977 |
| 7 | CY2012 | Fall | 1751494549 |
| 8 | CY2012 | Winter | 1660409552 |
| 9 | CY2013 | Spring | 1828526039 |
| 10 | CY2013 | Summer | 1835363103 |
| 11 | CY2013 | Fall | 1838698718 |
| 12 | CY2013 | Winter | 1738350857 |

You can view and run the SQL scripts that create the tables, the analytic view component objects, and the queries used in the examples from the Oracle Live SQL website at https://livesql.oracle.com/apex/livesql/file/index.html.


Copyright © 2001, 2025, Oracle and/or its affiliates.

# 27.6 Analytic View Queries with Filtered Facts and Added Measures

Queries that `SELECT` from analytic views may include the `FILTER FACT` keywords to filter the fact data accessed by the analytic view prior to any calculations and the `ADD MEASURES` keywords to define additional calculated measures for the query.

**Related Topics**

- Analytic View Query with Filtered Facts
- Analytic View Query with Added Measures
- Analytic View Query with Filtered Facts and Multiple Added Measures

## 27.6.1 Analytic View Query with Filtered Facts

In a query of an analytic view, you can filter the fact data before the analytic view aggregates the data for higher-level hierarchy members.

The values of aggregate records returned by an analytic view are determined by the hierarchies of the analytic view, the aggregation operators, and the rows contained in the fact table. A predicate in a `SELECT` statement that queries an analytic view restricts the rows returned by the analytic view but does not affect the computation of aggregate records.

By using the `FILTER FACT` keywords in a `SELECT` statement, you can filter fact records before the data is aggregated by the analytic view, which produces aggregate values only for the specified hierarchy members.

**Example 27-8 Queries With and Without Filter-Before Aggregation Predicates**

The following query selects hierarchy member names and sales values from the sales_av analytic view. The query predicate limits the hierarchy members to those in the YEAR level. The filtering does not affect the aggregation of the measure values.

```
SELECT time_hier.member_name, TO_CHAR(sales, '999,999,999,999') AS sales
  FROM sales_av HIERARCHIES(time_hier)
  WHERE time_hier.level_name = 'YEAR'
  ORDER BY time_hier.hier_order;
```

The result of the query is the following. The result includes the aggregated measure values for hierarchy members at the YEAR level.

```
MEMBER_NAME          SALES
-----------  -------------
CY2011       6,755,115,981
CY2012       6,901,682,399
CY2013       7,240,938,718
CY2014       7,579,746,353
CY2015       7,941,102,885
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

The following query defines an inline analytic view that the filters the hierarchy members before aggregation.

```
SELECT time_hier.member_name, TO_CHAR(sales, '999,999,999,999') AS sales
  FROM ANALYTIC VIEW (                      -- inline analytic view
  USING sales_av HIERARCHIES(time_hier)
  FILTER FACT (time_hier TO level_name = 'MONTH'
               AND TO_CHAR(month_end_date, 'Q') IN (1, 2)
               )
  )
  WHERE time_hier.level_name = 'YEAR')
  ORDER BY time_hier.hier_order;
```

The result of the query is the following. The `FILTER FACT` clause of the inline analytic view filters out all but the months that are in the first two quarters. The result includes the aggregated values at the YEAR level for those quarters. The aggregations do not include the third and fourth quarter values.

```
MEMBER_NAME          SALES
-----------  -------------
CY2011       3,340,459,835
CY2012       3,397,271,965
CY2013       3,564,557,290
CY2014       3,739,283,051
CY2015       3,926,231,605
```

**Related Topics**

- Analytic View Query with Filtered Facts and Multiple Added Measures

## 27.6.2 Analytic View Query with Added Measures

With the `ADD MEASURES` keywords, you can add measure calculations to a query of an analytic view.

**Example 27-9 Calculation Adding a Measure in the FROM Clause**

This example has an inline analytic view that adds the calculated measure share_sales to a query using the sales_av analytic view.

```
SELECT time_hier.member_name AS "Member",
       TO_CHAR(sales, '999,999,999,999') AS "Sales",
       ROUND(share_sales, 2) AS "Share of Sales"
  FROM ANALYTIC VIEW (
    USING sales_av HIERARCHIES (time_hier)
    ADD MEASURES (
      share_sales as (SHARE_OF(sales HIERARCHY time_hier PARENT))
    )
  )
  WHERE time_hier.level_name IN ('ALL', 'YEAR')
  ORDER BY time_hier.hier_order;
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

The following is the result of the query.

```
Member          Sales   Share of Sales
------  --------------  --------------
ALL     36,418,586,336
CY2011   6,755,115,981            0.19
CY2012   6,901,682,399            0.19
CY2013   7,240,938,718            0.2
CY2014   7,579,746,353            0.21
CY2015   7,941,102,885            0.22
```

**Example 27-10 Calculation Adding a Measure in the WITH Clause**

This example defines the same analytic view as in the previous example but it does so in the `WITH` clause of the `SELECT` statement.

```
WITH my_av ANALYTIC VIEW AS (
  USING sales_av HIERARCHIES (time_hier)
  ADD MEASURES (
    share_sales as (SHARE_OF(sales HIERARCHY time_hier PARENT))
  )
)
SELECT time_hier.member_name AS "Member",
       TO_CHAR(sales, '999,999,999,999') AS "Sales",
       ROUND(share_sales, 2) AS "Share of Sales"
  FROM my_av
  WHERE time_hier.level_name IN ('ALL', 'YEAR')
  ORDER BY time_hier.hier_order;
```

The result of the query are the same as the previous example.

```
Member          Sales   Share of Sales
------  --------------  --------------
ALL     36,418,586,336
CY2011   6,755,115,981            0.19
CY2012   6,901,682,399            0.19
CY2013   7,240,938,718            0.2
CY2014   7,579,746,353            0.21
CY2015   7,941,102,885            0.22
```

**Related Topics**

- Analytic View Query with Filtered Facts and Multiple Added Measures

## 27.6.3 Analytic View Query with Filtered Facts and Multiple Added Measures

In a query of an analytic view, you can specify pre-aggregation filters and added measures.

**Example 27-11 Query Using Filter Facts and Multiple Calculated Measures**

The analytic view in the `WITH` clause in this query is based on the sales_av analytic view. The my_av analytic view filters the time_hier hierarchy members to the first and second quarters of the QUARTER level and the geography_hier hierarchy members to the countries Mexico and


Copyright © 2001, 2025, Oracle and/or its affiliates.

Canada of the COUNTRY level. It adds calculated measures that compute sales for the prior period and the percent change of the difference between sales and the prior period sales.

```
WITH my_av ANALYTIC VIEW AS (
  USING sales_av HIERARCHIES (time_hier, geography_hier)
  FILTER FACT (time_hier TO level_name = 'QUARTER'
               AND (quarter_name LIKE 'Q1%' OR quarter_name LIKE 'Q2%'),
               geography_hier TO level_name = 'COUNTRY'
               AND country_name IN ('Mexico', 'Canada'))
  ADD MEASURES (sales_pp AS
                   (LAG(sales) OVER (HIERARCHY time_hier OFFSET 1)),
                sales_pp_pct_change AS
                  (LAG_DIFF_PERCENT(sales) OVER (HIERARCHY time_hier OFFSET
1)))
  )
 SELECT time_hier.member_name AS time,
   geography_hier.member_name AS geography,
   sales,
   sales_pp,
   ROUND(sales_pp_pct_change,3) AS "Change"
 FROM my_av HIERARCHIES (time_hier, geography_hier)
 WHERE time_hier.level_name IN ('YEAR') AND
       geography_hier.level_name = 'REGION'
 ORDER BY time_hier.hier_order;
```

The result is the following.

```
TIME    GEOGRAPHY           SALES    SALES_PP     Change
------  -------------  ------------  -----------  ------
CY2011  North America  229,884,616
CY2012  North America  233,688,485   229,884,616    .017
CY2013  North America  245,970,470   233,688,485    .053
CY2014  North America  256,789,511   245,970,470    .044
CY2015  North America  270,469,199   256,789,511    .053
```

**Related Topics**

- Analytic View Query with Filtered Facts
- Analytic View Query with Added Measures


Copyright © 2001, 2025, Oracle and/or its affiliates.

# Glossary

**additive**

Describes a fact (or measure) that can be summarized through addition. An additive fact is the most common type of fact. Examples include sales, cost, and profit. Contrast with nonadditive and semi-additive.

**advisor**

See SQL Access Advisor.

**aggregate**

Summarized data. For example, unit sales of a particular product could be aggregated by day, month, quarter and yearly sales.

**aggregation**

The process of consolidating data values into a single value. For example, sales data could be collected on a daily basis and then be aggregated to the week level, the week data could be aggregated to the month level, and so on. The data can then be referred to as aggregate data. The term aggregation is synonymous with summarization, and aggregate data is synonymous with summary data.

**analytic view**

A type of view that encapsulates aggregations, calculations, and joins of fact data. Analytic views organize data using a dimensional model. They allow you to easily add aggregations and calculations to data sets and to present data in views that can be queried with relatively simple SQL.

**ancestor**

A value at any level higher than a given value in a hierarchy. For example, in a Time dimension, the value 1999 might be the ancestor of the values Q1-99 and Jan-99.

**attribute**

A descriptive characteristic of one or more levels. For example, the product dimension for a clothing manufacturer might contain a level called item, one of whose attributes is color.


Copyright © 2001, 2025, Oracle and/or its affiliates.

Attributes represent logical groupings that enable end users to select data based on like characteristics.

Note that in relational modeling, an attribute is defined as a characteristic of an entity. In Oracle Database 10*g*, an attribute is a column in a dimension that characterizes each element of a single level.

### attribute dimension

Specifies a data source and the columns of the data source that are attributes of the attribute dimension. It specifies levels for its members and determines attribute relationships between levels. Attribute dimensions are used by hierarchies and analytic views.

### cardinality

From an OLTP perspective, this refers to the number of rows in a table. From a data warehousing perspective, this typically refers to the number of distinct values in a column. For most data warehouse DBAs, a more important issue is the degree of cardinality.

### child

A value at the level under a given value in a hierarchy. For example, in a Time dimension, the value Jan-99 might be the child of the value Q1-99. A value can be a child for more than one parent if the child value belongs to multiple hierarchies.

### cleansing

The process of resolving inconsistencies and fixing the anomalies in source data, typically as part of the ETL process.

### Common Warehouse Metadata (CWM)

A repository standard used by Oracle data warehousing, and decision support. The CWM repository schema is a standalone product that other products can share—each product owns only the objects within the CWM repository that it creates.

### cross product

A procedure for combining the elements in multiple sets. For example, given two columns, each element of the first column is matched with every element of the second column. A simple example is illustrated as follows:

```
Col1   Col2   Cross Product
----   ----   -------------
a      c      ac
b      d      ad
              bc
              bd
```


Copyright © 2001, 2025, Oracle and/or its affiliates.

Cross products are performed when grouping sets are concatenated, as described in SQL for Aggregation in Data Warehouses.

**data mart**

A data warehouse that is designed for a particular line of business, such as sales, marketing, or finance. In a dependent data mart, the data can be derived from an enterprise-wide data warehouse. In an independent data mart, data can be collected directly from sources.

**data source**

A database, application, repository, or file that contributes data to a warehouse.

**data warehouse**

A relational database that is designed for query and analysis rather than transaction processing. A data warehouse usually contains historical data that is derived from transaction data, but it can include data from other sources. It separates analysis workload from transaction workload and enables a business to consolidate data from several sources.

In addition to a relational database, a data warehouse environment often consists of an ETL solution, an analytical SQL engine, client analysis tools, and other applications that manage the process of gathering data and delivering it to business users.

**degree of cardinality**

The number of unique values of a column divided by the total number of rows in the table. This is particularly important when deciding which indexes to build. You typically want to use bitmap indexes on low degree of cardinality columns and B-tree indexes on high degree of cardinality columns. As a general rule, a cardinality of under 1% makes a good candidate for a bitmap index.

**denormalize**

The process of allowing redundancy in a table. Contrast with normalize.

**derived fact (or measure)**

A fact (or measure) that is generated from existing data using a mathematical operation or a data transformation. Examples include averages, totals, percentages, and differences.

**detail**

See: fact table.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**detail table**

See: fact table.

**dimension**

The term dimension is commonly used in two ways:

- A general term for any characteristic that is used to specify the members of a data set. The three most common dimensions in a sales-oriented data warehouse are time, geography, and product. Most dimensions have hierarchies.
- An object defined in a database to enable queries to navigate dimensions. In Oracle Database 10*g*, a dimension is a database object that defines hierarchical (parent/child) relationships between pairs of column sets. In Oracle Express, a dimension is a database object that consists of a list of values.

**dimension table**

Dimension tables describe the business entities of an enterprise, represented as hierarchical, categorical information such as time, departments, locations, and products. Dimension tables are sometimes called lookup or reference tables.

**dimension value**

One element in the list that makes up a dimension. For example, a computer company might have dimension values in the product dimension called LAPPC and DESKPC. Values in the geography dimension might include Boston and Paris. Values in the time dimension might include MAY96 and JAN97.

**drill**

To navigate from one item to a set of related items. Drilling typically involves navigating up and down through a level (or levels) in a hierarchy. When selecting data, you expand a hierarchy when you drill down in it, and you collapse a hierarchy when you drill up in it.

**drill down**

To expand the view to include child values that are associated with parent values in the hierarchy.

**drill up**

To collapse the list of descendant values that are associated with a parent value in the hierarchy.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**element**

An object or process. For example, a dimension is an object, a mapping is a process, and both are elements.

**enterprise data warehouse**

A data warehouse where raw data is consolidated in one storage location and is used as the center of the data warehousing architecture.

**entity**

Entity is used in database modeling. In relational databases, it typically maps to a table.

**ELT**

ELT stands for extraction, loading, transformation, and transportation. This is a more modern version of the old ETL.

**ETL**

ETL stands for extraction, transformation, and loading. ETL refers to the methods involved in accessing and manipulating source data and loading it into a data warehouse. The order in which these processes are performed varies.

Note that ETT (extraction, transformation, transportation) and ETM (extraction, transformation, move) are sometimes used instead of ETL.

**extraction**

The process of taking data out of a source as part of an initial phase of ETL.

**fact**

Data, usually numeric and additive, that can be examined and analyzed. Examples include sales, cost, and profit. Fact and measure are synonymous; fact is more commonly used with relational environments, measure is more commonly used with multidimensional environments. A derived fact (or measure) is generated from existing data using a mathematical operation or a data transformation.

**fact table**

A table in a star schema that contains facts. A fact table typically has two types of columns: those that contain facts and those that are dimension table foreign keys. The primary key of a fact table is usually a composite key that is made up of all of its foreign keys.

A fact table might contain either detail level facts or facts that have been aggregated (fact tables that contain aggregated facts are often instead called summary tables). A fact table usually contains facts with the same level of aggregation.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**fast refresh**

An operation that applies only the data changes to a materialized view, thus eliminating the need to rebuild the materialized view from scratch.

**file-to-table mapping**

Maps data from flat files to tables in the warehouse.

**hierarchy**

A logical structure that uses ordered levels as a means of organizing data. A hierarchy can be used to define data aggregation; for example, in a time dimension, a hierarchy might be used to aggregate data from the Month level to the Quarter level to the Year level. Hierarchies can be defined in Oracle as part of the dimension object. A hierarchy can also be used to define a navigational drill path, regardless of whether the levels in the hierarchy represent aggregated totals.

A hierarchy can also be a data dictionary object that is a type of view that defines the hierarchical relationships between the levels of an attribute dimension. Attribute dimensions and hierarchies provide the dimension members of an analytic view.

**level**

A position in a hierarchy. For example, a time dimension might have a hierarchy that represents data at the Month, Quarter, and Year levels.

**level value table**

A database table that stores the values or data for the levels you created as part of your dimensions and hierarchies.

**mapping**

The definition of the relationship and data flow between source and target objects.

**materialized view**

A pre-computed table comprising aggregated or joined data from fact and possibly a dimension table. Also known as a summary or aggregate table.

**materialized view log**

A log that records details about a given materialized view. Materialized view logs are required if you want to use fast refresh, with the exception of partition change tracking refresh.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**measure**

See fact.

**metadata**

Data that describes data and other structures, such as objects, business rules, and processes. For example, the schema design of a data warehouse is typically stored in a repository as metadata, which is used to generate scripts used to build and populate the data warehouse. A repository contains metadata.

Examples include: for data, the definition of a source to target transformation that is used to generate and populate the data warehouse; for information, definitions of tables, columns and associations that are stored inside a relational modeling tool; for business rules, discount by 10 percent after selling 1,000 items.

**model**

An object that represents something to be made. A representative style, plan, or design. A model can also be metadata that defines the structure of the data warehouse.

**nonadditive**

Describes a fact (or measure) that cannot be summarized through addition. An example includes Average. Contrast with additive and semi-additive.

**normalize**

In a relational database, the process of removing redundancy in data by separating the data into multiple tables. Contrast with denormalize.

The process of removing redundancy in data by separating the data into multiple tables.

**OLTP**

See: online transaction processing (OLTP).

**online transaction processing (OLTP)**

Online transaction processing. OLTP systems are optimized for fast and reliable transaction handling. Compared to data warehouse systems, most OLTP interactions will involve a relatively small number of rows, but a larger group of tables.

**parallel execution**

Breaking down a task so that several processes do part of the work. When multiple CPUs each do their portion simultaneously, very large performance gains are possible.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**parallelism**

Breaking down a task so that several processes do part of the work. When multiple CPUs each do their portion simultaneously, very large performance gains are possible.

**parent**

A value at the level above a given value in a hierarchy. For example, in a Time dimension, the value `Q1-99` might be the parent of the child value `Jan-99`.

**partition**

Very large tables and indexes can be difficult and time-consuming to work with. To improve manageability, you can break your tables and indexes into smaller pieces called partitions.

**partition change tracking (PCT)**

A way of tracking the staleness of a materialized view on the partition and subpartition level.

**pattern matching**

A way of recognizing patterns in a sequence of rows using the `MATCH_RECOGNIZE` clause.

**pivoting**

A transformation where each record in an input stream is converted to many records in the appropriate table in the data warehouse. This is particularly important when taking data from nonrelational databases.

**query rewrite**

A mechanism to use a materialized view (which is precomputed) to quickly answer queries.

**refresh**

The mechanism whereby a materialized view is changed to reflect new data.

**rewrite**

See: query rewrite.

**schema**

A collection of related database objects. Relational schemas are grouped by database user ID and include tables, views, and other objects. The sample schemas `sh` are used throughout this Guide. Two special types of schema are snowflake schema and star schema.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**semi-additive**

Describes a fact (or measure) that can be summarized through addition along some, but not all, dimensions. Examples include headcount and on hand stock. Contrast with additive and nonadditive.

**slice and dice**

This is an informal term referring to data retrieval and manipulation. We can picture a data warehouse as a cube of data, where each axis of the cube represents a dimension. To "slice" the data is to retrieve a piece (a slice) of the cube by specifying measures and values for some or all of the dimensions. When we retrieve a data slice, we may also move and reorder its columns and rows as if we had diced the slice into many small pieces. A system with good slicing and dicing makes it easy to navigate through large amounts of data.

**snowflake schema**

A type of star schema in which each dimension table is partly or fully normalized.

**source**

A database, application, file, or other storage facility from which the data in a data warehouse is derived.

**source system**

A database, application, file, or other storage facility from which the data in a data warehouse is derived.

**source tables**

The tables in a source database.

**SQL Access Advisor**

The SQL Access Advisor helps you achieve your performance goals by recommending the proper materialized view set, materialized view logs, partitions, and indexes for a given workload. It is a GUI in Oracle Enterprise Manager, and has similar capabilities to the `DBMS_ADVISOR` package.

**staging area**

A place where data is processed before entering the warehouse.

**staging file**

A file used when data is processed before entering the warehouse.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**star query**

A join between a fact table and a number of dimension tables. Each dimension table is joined to the fact table using a primary key to foreign key join, but the dimension tables are not joined to each other.

**star schema**

A relational schema whose design represents a multidimensional data model. The star schema consists of one or more fact tables and one or more dimension tables that are related through foreign keys.

**subject area**

A classification system that represents or distinguishes parts of an organization or areas of knowledge. A data mart is often developed to support a subject area such as sales, marketing, or geography.

**summary**

See: materialized view.

**Summary Advisor**

Replaced by the SQL Access Advisor.

**target**

Holds the intermediate or final results of any part of the ETL process. The target of the entire ETL process is the data warehouse.

**third normal form (3NF)**

A classical relational database modeling technique that minimizes data redundancy through normalization.

**third normal form schema**

A schema that uses the same kind of normalization as typically found in an OLTP system. Third normal form schemas are sometimes chosen for a large data warehouse, especially an environment with significant data loading requirements that is used to feed a data mart and execute long-running queries. Compare with snowflake schema and star schema.

**transformation**

The process of manipulating data. Any manipulation beyond copying is a transformation. Examples include cleansing, aggregating, and integrating data from multiple source tables.


Copyright © 2001, 2025, Oracle and/or its affiliates.

**transportation**

The process of moving copied or transformed data from a source to a data warehouse. Compare with transformation.

**unique identifier**

An identifier whose purpose is to differentiate between the same item when it appears in more than one place.

**update window**

The length of time available for updating a warehouse. For example, you might have 8 hours at night to update your warehouse.

**update frequency**

How often a data warehouse is updated with new information. For example, a warehouse might be updated nightly from an OLTP system.

**validation**

The process of verifying metadata definitions and configuration parameters.

**versioning**

The ability to create new versions of a data warehouse project for new requirements and changes.


Copyright © 2001, 2025, Oracle and/or its affiliates.

# Index

## Numerics

3NF, *2*

## A

about
    refresh statistics, *1*
accessing
    real-time materialized views, *16*
ADO, and IM column store, *12*
aggregates, *10*, *74*
    computability check, *10*
aggregation
    In-Memory Column Store, *22*
    VECTOR GROUP BY, *22*
alter analytic views, *10*
altering dimensions, *10*
alternate keys
    of an attribute dimension level, *8*
amortization
    calculating, *42*
analytic functions, *31*
    concepts, *2*
analytic processing
    materialized views, *7*
analytic views
    about, *1*
    alter or drop, *10*
    and materialized views, *8*
    APIs, *5*
    compilation states, *8*
    creating, *3*
    described, *1*
    example of creating, *16*
    example of filtering facts, *15*, *17*
    example of using added measures, *16*, *17*
    example scripts, *10*
    hierarchical attributes, *4*
    hierarchies, *1*
    measures, *2*
    privileges for, *4*
    sharing with application containers, *9*
    tables for examples, *10*
application containers, sharing analytic views
        with, *9*
applications
    decision support systems (DSS), *3*
APPROX_COUNT function, *31*
APPROX_RANK function, *9*
APPROX_SUM function, *31*
approximate aggregations, *28*
approximate query processing, *29*
approximate top-n query processing, *31*
approximate values
    percentile functions, *39*
architecture
    data warehouse, *5*
attribute clustering, *1*
    adding to existing table, *10*
    advantages, *4*
    data dictionary views, *13*
    dropping, *11*
    guidelines, *4*
    methods, *1*
    modifying, *11*
    privileges, *7*
    types, *2*
        interleaved ordering, *3*
        linear ordering, *2*, *7*, *8*
    using hints for DML, *11*
attribute dimensions
    about, *1*
    attributes, *4*
    examples of creating, *12*
    level keys, *8*
    order of level members, *7*
    privileges for, *4*
attribute-clustered tables, *1*
attributes, *2*
    attribute dimension, *4*
    creating hierarchies using, *11*
automatic big table caching
    about, *13*

## B

B-tree indexes, *8*
    bitmap indexes versus, *3*
base measures
    examples of, *3*


Copyright © 2001, 2025, Oracle and/or its affiliates.

bind variables
with query rewrite, 56
bitmap indexes, 1
nulls and, 2
on partitioned tables, 2
parallel query and DML, 3
bitmap join indexes, 5
build methods, 21
business intelligence, 1
queries, 1
business rules
violation of, 19

## C

calculated measures, 2
examples of, 6, 16, 17
CAPTION classification, 9
cardinality
degree of, 3
CASE expressions, 75
cell referencing, 11
classifications
analytic view, 9
columns
cardinality, 3
common joins, 5
common tasks
in data warehouses, 4
compilation states
of analytic views, 8
complete refresh, 3
complex queries
snowflake schemas, 10
composite
columns, 15
compression
See data segment compression, 21
concatenated groupings, 17
concatenated ROLLUP, 24
constraints, 10, 9
foreign key, 12
RELY, 13
states, 10
unique, 11
view, 14, 43
with partitioning, 14
with query rewrite, 74
cost-based rewrite, 2
CREATE DIMENSION statement, 4
CREATE MATERIALIZED VIEW statement, 17
enabling query rewrite, 3
creating
materialized views with approximate queries, 33
real-time materialized views, 16
creating (continued)
zone maps, 8
with attribute clustering, 6
CUBE clause, 7
partial, 8
when to use, 7
cubes
hierarchical, 8
materialized views, 8
CUME_DIST function, 11

## D

data
nonvolatile, 3
purging, 36
sufficiency check, 9
transformation, 8
transportation, 1
data compression, 18
See data segment compression, 21
data cubes
hierarchical, 19
data densification, 53
time series calculation, 59
with sparse data, 54
data error handling
using SQL, 20
data marts, 6
data rules
violation of, 19
data segment compression, 3
materialized views, 21
partitioning, 3
data transformation
multistage, 1
pipelined, 2
data warehouse, 1
architectures, 5
dimension tables, 5
fact tables, 5
physical design, 1
refresh tips, 12
data warehouses
common tasks, 4
key characteristics of, 3
database
staging, 1
date folding
with query rewrite, 42
DBMS_ADVISOR
TUNE_MVIEW procedure, 22
DBMS_ERROR package, 22
DBMS_MVIEW package, 8
EXPLAIN_MVIEW procedure, 37
EXPLAIN_REWRITE procedure, 67


Copyright © 2001, 2025, Oracle and/or its affiliates.

DBMS_SYNC_REFRESH package, *1*
decision support systems (DSS)
  bitmap indexes, *3*
degree of cardinality, *3*
DENSE_RANK function, *4*
densification
  data, *53*
DESCRIPTION classification, *9*
design
  logical, *1*
  physical, *1*
DETERMINES
  clause of an attribute dimension level, *9*
dimension levels
  skipping, *4*
dimension tables, *6*
  normalized, *7*
dimensions, *7*, *1*, *9*
  altering, *10*
  analyzing, *2*
  creating, *3*
  definition, *1*
  dimension tables, *5*
  dropping, *11*
  hierarchies, *8*
  multiple, *2*
  skipping levels, *4*
  validating, *9*
  with query rewrite, *74*
displaying
  real-time materialized views, *23*
drilling down, *1*
  hierarchies, *1*
drop analytic views, *10*
DROP MATERIALIZED VIEW statement
  prebuilt tables, *35*
dropping
  dimensions, *11*
  materialized views, *36*
  zone maps, *14*

## E

ELT, *1*
entity, *2*
error logging, *19*
  table, *21*
errors
  handling, *19*
ETL jobs
  monitoring, *3*
ETL. See extraction, transformation, and loading (ETL), *1*
examples
  analytic views, *10*
  tables for analytic view, *10*
examples *(continued)*
  zone maps
    join pruning, *26*
    partition and table scan pruning, *25*
EXCHANGE PARTITION statement, *14*
execution plans
  star transformations, *21*
EXPAND_GSET_TO_UNION hint, *52*, *76*
EXPLAIN PLAN statement, *67*
  star transformations, *21*
EXPLAIN_REWRITE procedure, *67*
exporting
  EXP utility, *6*
expression matching
  with query rewrite, *53*
external tables, *6*
extraction, transformation, and loading (ETL), *1*
  overview, *1*
  process, *10*
extractions
  data files, *5*
  distributed operations, *7*
  full, *2*
  incremental, *2*
  OCI, *6*
  online, *2*
  overview, *1*
  physical, *2*
  Pro*C, *6*
  SQL*Plus, *5*

## F

facts, *1*
fast refresh, *4*
  restrictions, *26*
  with UNION ALL, *23*
FETCH
  row_limiting_clause, *77*
files
  ultralarge, *2*
filling gaps
  with data, *58*
FIRST_VALUE function, *26*
FIRST/LAST functions, *34*
FOR loops, *23*
foreign key
  constraints, *12*
  joins
    snowflake schemas, *10*
frequent itemsets, *77*
functions
  analytic, *31*
  APPROX_COUNT, *31*
  APPROX_RANK, *9*
  APPROX_SUM, *31*


Copyright © 2001, 2025, Oracle and/or its affiliates.

functions *(continued)*
COUNT, *2*
CUME_DIST, *11*
DENSE_RANK, *4*
FIRST_VALUE, *26*
FIRST/LAST, *34*
GROUP_ID, *13*
GROUPING, *9*
GROUPING_ID, *12*
LAG/LEAD, *24*
LAST_VALUE, *26*
linear regression, *42*
LISTAGG function, *30*
NTH_VALUE, *27*
RANK, *4*
ranking, *4*
RATIO_TO_REPORT, *24*
REGR_INTERCEPT, *43*
REGR_SLOPE, *43*
reporting, *22*
ROW_NUMBER, *13*
WIDTH_BUCKET, *71*, *72*
window, *31*
windowing, *13*

## G

GROUP_ID function, *13*
grouping
compatibility check, *9*
conditions, *74*
GROUPING function, *9*
when to use, *12*
GROUPING_ID function, *12*
GROUPING_SETS expression, *14*

## H

hierarchical attributes
in a hierarchy, *4*
MEMBER_UNIQUE_NAME, *4*
hierarchical cubes, *8*
in SQL, *24*
hierarchies, *1*
analytic view, *1*
examples of creating, *15*
hierarchical attributes, *4*
how used, *8*
multiple, *6*
overview, *7*
privileges for, *4*
rolling up and drilling down, *1*
using any attribute, *11*
validating data of, *8*
hints
EXPAND_GSET_TO_UNION, *52*, *76*
hints *(continued)*
NOWRITE, *75*
query rewrite, *3*, *75*
REWRITE, *75*
REWRITE_OR_ERROR, *76*
histograms
creating with user-defined buckets, *76*
hypothetical rank, *41*

## I

IM column store, managing contents, *12*
In-Memory aggregation, *22*
In-Memory Column Store
aggregation, *22*
in-memory expressions, *11*
in-memory virtual columns, *12*
indexes
B-tree, *8*
bitmap indexes, *2*
bitmap join, *5*
cardinality, *3*
nulls and, *2*
partitioned tables, *2*
initialization parameters
QUERY_REWRITE_ENABLED, *3*
integrity constraints, *10*
invalidating
materialized views, *13*
itemsets
frequent, *77*

## J

join compatibility, *4*

## K

key lookups, *23*
keys, *6*
of an attribute dimension level, *8*

## L

LAG/LEAD functions, *24*
LAST_VALUE function, *26*
level relationships, *8*
purpose, *8*
levels, *7*, *8*
order of members in, *7*
types in an attribute dimension, *1*
levels in a dimension
skipping, *4*
limiting rows returned, *77*
linear regression functions, *42*


Copyright © 2001, 2025, Oracle and/or its affiliates.

LISTAGG function, *30*
local indexes, *2*, *3*
logging
  error, *19*
logical design, *1*
logs
  materialized views, *31*
lookup tables
  See dimension tables, *5*

## M

managing
  refresh statistics, *2*
manual
  refresh, *9*
manual refresh
  with DBMS_MVIEW package, *8*
MATCH_RECOGNIZE clause, *1*
materialized view logs, *31*
materialized view refresh
  online table redefinition, *14*
materialized views
  aggregates, *10*
  altering, *14*
  analytic processing, *7*
  based on hybrid partitioned tables, *20*
  build methods, *21*
  containing only joins, *13*
  creating, *17*
  cubes, *8*
  data segment compression, *21*
  delta joins, *7*
  dropping, *35*, *36*
  invalidating, *13*
  logs, *4*
  multiple, *30*
  naming, *20*
  nested, *15*
  ON STATEMENT refresh, *7*
  partition change tracking (PCT), *1*
  partitioned tables, *24*
  partitioning, *1*
  prebuilt, *17*
  query rewrite
    hints, *3*, *75*
    matching join graphs, *22*
    parameters, *3*
    privileges, *5*
  real-time materialized views, *15*
  refresh dependent, *11*
  refreshing, *24*, *1*
  refreshing all, *10*
  registration, *34*
  restrictions, *22*
materialized views *(continued)*
  rewrites
    enabling, *3*
  schema design, *6*
  schema design guidelines, *6*
  security, *11*
  set operators, *9*
  storage characteristics, *21*
  types of, *10*
  uses for, *1*
  with VPD, *12*
materialized viewsCOUNT(DISTINCT), *34*
measures, *6*
  analytic view, *2*
  base, *3*
  calculated, *6*
MERGE PARTITION operation, *18*
MERGE statement, *33*
MODEL clause, *1*
  cell referencing, *11*
  data flow, *3*
  keywords, *10*
  parallel execution, *36*
  rules, *12*
modeling
  logical design, *1*
  physical design, *1*
modifying
  zone maps, *13*
monitoring
  ETL jobs, *3*
  refresh, *15*
mortgage calculation, *42*
MOVE PARTITION operation, *18*
multiple hierarchies, *6*
multiple materialized views, *30*
MV_CAPABILITIES_TABLE table, *38*

## N

nested materialized views, *15*
  refreshing, *22*
  restrictions, *17*
net present value
  calculating, *40*
nonvolatile data, *3*
NOREWRITE hint, *3*, *75*
NTH_VALUE function, *27*
nulls
  indexes and, *2*

## O

OFFSET
  row_limiting_clause, *77*
ON COMMIT clause, *24*


Copyright © 2001, 2025, Oracle and/or its affiliates.

ON DEMAND clause, *24*
ON STATEMENT clause, *24*
online redefinition
  materialized views, *14*
optimizations
  query rewrite
    enabling, *3*
    hints, *3*, *75*
    matching join graphs, *22*
  query rewrites
    privileges, *5*
optimizer
  with rewrite, *1*
ORDER BY clause, *30*
  in an attribute dimension level, *7*
out-of-place refresh, *6*
outer joins
  with query rewrite, *74*

## P

packages
  DBMS_ADVISOR, *3*
  DBMS_DIMENSION, *8*
  DBMS_ERROR, *22*
  DBMS_ERRORLOG, *22*, *25*
  DBMS_MVIEW, *38*, *2*
  DBMS_SYNC_REFRESH, *1*
parallel DML
  bitmap indexes, *3*
parallel execution, *14*
parallel query
  bitmap indexes, *3*
parallelism, *14*
partition change tracking (PCT), *1*, *24*, *22*
  with Pmarkers, *28*
partitioned outer join, *53*
partitioned tables
  materialized views, *24*
partitioning, *4*
  materialized views, *1*
  prebuilt tables, *6*
partitions
  bitmap indexes, *2*
pattern matching, *1*
  keywords, *10*
patterns
  in SQL, *1*
percentile functions
  approximate results, *39*
physical design, *1*
  structures, *2*
pivoting, *26*, *47*
  operations, *47*
plans
  star transformations, *21*
Pmarkers
  with PCT, *28*
prebuilt materialized views, *17*
pruning
  examples, *25*, *26*
  using zone maps, *22*
purging data, *36*

## Q

queries
  running with approximate functions, *30*
query delta joins, *7*
query rewrite
  advanced, *57*
  checks made by, *4*
  controlling, *4*
  correctness, *4*
  date folding, *42*
  enabling, *2*, *3*
  hints, *3*, *75*
  matching join graphs, *22*
  materialized view with approximate queries, *62*
  methods, *1*
  parameters, *3*
  privileges, *5*
  real-time materialized views, *19*
  restrictions, *23*
  using equivalences, *57*
  using GROUP BY extensions, *49*
  using nested materialized views, *38*
  using PCT, *22*
  VPD, *12*
  when it occurs, *2*
  with bind variables, *56*
  with DBMS_MVIEW package, *67*
  with expression matching, *53*
  with partially stale materialized views, *53*
  with selfjoins, *40*
  with set operator materialized views, *46*
  with view constraints, *43*
QUERY_REWRITE_ENABLED initialization parameter, *3*

## R

range-partitioned table, *18*
RANK function, *4*
ranking functions, *4*
RATIO_TO_REPORT function, *24*
real-time materialized views, *15*
  accessing, *16*
  creating, *16*
  displaying, *23*
  for direct query access, *21*


Copyright © 2001, 2025, Oracle and/or its affiliates.

real-time materialized views *(continued)*
guidelines, *23*
query rewrite, *19*
restrictions, *16*
reference tables
See dimension tables, *5*
refresh
monitoring, *15*
options, *23*
out-of-place, *6*
scheduling, *18*
synchronous, *1*
with UNION ALL, *23*
refresh statistics
about, *1*, *4*, *6*
analyzing, *14*
collecting, *4*, *5*
data dictionary views, *2*
managing, *2*
modifying collection level, *5*
modifying retention period, *7*
purging, *9*
retaining, *6*
retention period, *7*
setting defaults, *5*, *7*
SQL statements, *13*
understanding, *14*
viewing basic, *10*
viewing change data, *12*
viewing detailed, *11*
viewing settings, *8*
refreshing
materialized views, *1*
materialized views based on approximate queries, *12*
nested materialized views, *22*
partitioning, *28*
zone maps, *21*
REGR_INTERCEPT function, *43*
REGR_R2 function, *43*
REGR_SLOPE function, *43*
RELY constraints, *13*
reporting functions, *22*
restrictions
fast refresh, *26*
nested materialized views, *17*
query rewrite, *23*
result set, *19*
REWRITE hint, *3*, *75*
REWRITE_OR_ERROR hint, *76*
rewrites
hints, *75*
parameters, *3*
privileges, *5*
query optimizations
hints, *3*, *75*
rewrites *(continued)*
query optimizations *(continued)*
matching join graphs, *22*
rolling up hierarchies, *1*
ROLLUP, *4*
concatenated, *24*
partial, *6*
when to use, *4*
root level, *8*
row_limiting_clause clause, *77*
ROW_NUMBER function, *13*
rules
in MODEL clause, *12*
in SQL modeling, *12*
order of evaluation, *15*

## S

schemas
3NF, *2*
design guidelines for materialized views, *6*
snowflake, *2*
star, *2*
set operators
materialized views, *9*
simultaneous equations, *41*
SKIP WHEN NULL clause, *4*
skipping levels in a dimension, *4*
snowflake schemas, *10*
complex queries, *10*
source systems, *1*
sparse data
data densification, *54*
SPLIT PARTITION operation, *18*
SQL modeling, *1*
cell referencing, *11*
keywords, *10*
order of evaluation, *15*
performance, *36*
rules, *12*
rules and restrictions, *34*
staging
areas, *6*
databases, *1*
files, *1*
star queries
star transformation, *19*
star schemas
dimensional model, *6*
star transformations, *19*
restrictions, *23*
statistics, *75*
storage
optimizing, *18*
summary management
components, *4*


Copyright © 2001, 2025, Oracle and/or its affiliates.

sync refresh, *1*
synchronous refresh, *1*

## T

tables
    attribute-clustered, *1*
    detail tables, *5*
    dimension tables (lookup tables), *5*
    external, *6*
    fact tables, *5*
tablespaces
    transportable, *3*, *2*, *5*
text match, *12*
    with query rewrite, *74*
Third Normal Form, *2*
time series calculations, *59*
timestamps, *4*
Top-N queries, *77*
transformations, *1*
    scenarios, *23*
    SQL and PL/SQL, *9*
    SQL*Loader, *4*
transportable tablespaces, *3*, *2*, *5*
transportation
    definition, *1*
    distributed operations, *2*
    flat files, *1*
triggers, *4*
types
    attribute clustering, *2*
    zone maps, *2*

## U

ultralarge files, *2*
unique
    constraints, *11*
    identifier, *2*, *1*
update frequencies, *9*
UPDATE statement, *16*
update windows, *9*
UPSERT ALL statement, *16*
UPSERT statement, *16*

## V

validating dimensions, *9*
validating hierarchy data, *8*
VECTOR GROUP BY
    aggregation, *22*
VECTOR GROUP BY aggregation
    optimizing star queries, *28*
view constraints, *14*, *43*
VPD
    and materialized views, *12*
    restrictions with materialized views, *13*

## W

WIDTH_BUCKET function, *71*, *72*
window functions, *31*
windowing functions, *13*

## Z

zone maps, *1*
    about, *1*
    about refreshing, *20*
    about staleness, *18*
    automatic refresh, *4*
    benefits, *2*
    compiling, *14*
    creating, *8*
        with attribute clustering, *6*
    data dictionary views, *27*
    dropping, *14*
    maintaining, *16*
    maintenance, *4*
    modifying, *13*
    privileges, *5*
    pruning, *22*
    refreshing, *21*
    types, *2*
    usage for SQL statements, *15*
    usage for SQL workloads, *15*
    usage scenarios, *3*
    with attribute clustering, *2*


Copyright © 2001, 2025, Oracle and/or its affiliates.