

Oracle® Warehouse Builder

Transformation Guide

10g Release 1 (10.1)

Part No. B12151-02

September 2007

Oracle Warehouse Builder Transformation Guide, 10g Release 1 (10.1)

Part No. B12151-02

Copyright © 2007 Oracle. All rights reserved.

Primary Author: Jean-Pierre Dijcks

Contributing Author: Shirinne Alison, Kavita Nayar, Padmaja Potineni

The Programs (which include both the software and documentation) contain proprietary information; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. This document is not warranted to be error-free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose.

If the Programs are delivered to the United States Government or anyone licensing or using the Programs on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth in FAR 52.227-19, Commercial Computer Software--Restricted Rights (June 1987). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and we disclaim liability for any damages caused by such use of the Programs.

Oracle, JD Edwards, PeopleSoft, and Siebel are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

The Programs may provide links to Web sites and access to content, products, and services from third parties. Oracle is not responsible for the availability of, or any content provided on, third-party Web sites. You bear all risks associated with the use of such content. If you choose to purchase any products or services from a third party, the relationship is directly between you and the third party. Oracle is not responsible for: (a) the quality of third-party products or services; or (b) fulfilling any of the terms of the agreement with the third party, including delivery of products or services and warranty obligations related to purchased products or services. Oracle is not responsible for any loss or damage of any sort that you may incur from dealing with any third party.

Contents

Preface	ix
Purpose	ix
Documentation Accessibility	ix
Audience	x
How This Guide Is Organized.....	x
New in 10g Release 1 (10.1).....	xi
Added in Release 9.0.4.....	xiii
Conventions	xvii
Related Publications.....	xvii
Contacting Oracle.....	xviii
1 Introduction to Warehouse Builder Transformations	
Overview	1-1
Transforming Data with Warehouse Builder	1-2
SQL Standards	1-2
How SQL Works	1-2
SQL as the Common Language for Relational Databases	1-3
2 Transformations	
Regular SQL Operators	2-1
Deduplicator (DISTINCT).....	2-1
Filter (WHERE).....	2-2
Joiner (FULL OUTER JOIN)	2-4
Key Lookup.....	2-7
Pivot Operator	2-8
Example: Pivoting Sales Data.....	2-9
Sequence (CURRVAL, NEXTVAL)	2-10
Set (UNION, UNION ALL, INTERSECT, MINUS)	2-12
Sorter (ORDER BY)	2-13
Splitter (Multiple Table WHERE)	2-15
Table Function	2-18
Unpivot Operator.....	2-19
Example: Unpivoting Sales Data	2-19
Aggregator (GROUP BY, HAVING)	2-20
AVG	2-22

COUNT.....	2-22
MAX.....	2-23
MIN.....	2-23
STDDEV	2-24
STDDEV_POP	2-25
STDDEV_SAMP	2-25
SUM	2-26
VAR_POP.....	2-26
VAR_SAMP	2-27
VARIANCE.....	2-27
Constant.....	2-28
SYSDATE.....	2-28
SYSTIMESTAMP.....	2-28
Data Cleansing Operators.....	2-28
Name and Address	2-28
Match-Merge Operator.....	2-30
Example: Matching and Merging Customer Data	2-30
Designing Mappings with a Match-Merge Operator	2-30

3 SQL Transformations

Introduction.....	3-1
About Transformations	3-1
About Oracle Transformation Libraries	3-2
Global Shared Library	3-2
Oracle Library.....	3-2
Accessing Transformation Libraries	3-2
Importing PL/SQL Packages	3-3
Administrative Transformations	3-3
WB_ABORT	3-3
WB_ANALYZE_SCHEMA.....	3-3
WB_ANALYZE_TABLE.....	3-4
WB_COMPILE_PLSQL	3-4
WB_DISABLE_ALL_CONSTRAINTS	3-5
WB_DISABLE_ALL_TRIGGERS	3-6
WB_DISABLE_CONSTRAINT	3-6
WB_DISABLE_TRIGGER.....	3-7
WB_ENABLE_ALL_CONSTRAINTS	3-8
WB_ENABLE_ALL_TRIGGERS	3-9
WB_ENABLE_CONSTRAINT	3-9
WB_ENABLE_TRIGGER	3-10
WB_TRUNCATE_TABLE.....	3-11
Character Transformations	3-12
ASCII.....	3-12
ASCIISTR.....	3-12
CHARTOROWID.....	3-13
CHR.....	3-13
CONCAT	3-14

CONVERT	3-14
INITCAP	3-15
INSTR / INSTRB	3-15
LENGTH/LENGTHB	3-16
LOWER	3-17
LPAD	3-17
LTRIM	3-18
NLSSORT	3-18
NLS_INITCAP	3-19
NLS_LOWER	3-19
NLS_UPPER	3-20
REPLACE	3-20
RPAD	3-21
RTRIM	3-22
SOUNDEX	3-22
SUBSTR	3-23
TO_DATE	3-24
TO_MULTI_BYTE	3-24
TO_NUMBER	3-25
TO_SINGLE_BYTE	3-25
TRANSLATE	3-26
TRIM	3-26
UPPER	3-27
WB.LOOKUP_CHAR	3-27
WB.LOOKUP_CHAR	3-28
WB_IS_SPACE	3-29
Date Transformations	3-29
ADD_MONTHS	3-29
LAST_DAY	3-30
MONTHS_BETWEEN	3-30
NEW_TIME	3-30
NEXT_DAY	3-31
ROUND (date)	3-32
SYSDATE	3-32
TO_CHAR (datetime)	3-32
TRUNC (date)	3-33
WB_CAL_MONTH_NAME	3-33
WB_CAL_MONTH_OF_YEAR	3-34
WB_CAL_MONTH_SHORT_NAME	3-34
WB_CAL_QTR	3-35
WB_CAL_WEEK_OF_YEAR	3-35
WB_CAL_YEAR	3-36
WB_CAL_YEAR_NAME	3-36
WB_DATE_FROM_JULIAN	3-37
WB_DAY_NAME	3-37
WB_DAY_OF_MONTH	3-38
WB_DAY_OF_WEEK	3-38

WB_DAY_OF_YEAR.....	3-39
WB_DAY_SHORT_NAME.....	3-39
WB_DECADE.....	3-40
WB_HOUR12.....	3-40
WB_HOUR12MI_SS.....	3-41
WB_HOUR24.....	3-41
WB_HOUR24MI_SS.....	3-42
WB_IS_DATE.....	3-42
WB_JULIAN_FROM_DATE.....	3-43
WB_MI_SS.....	3-43
WB_WEEK_OF_MONTH.....	3-44
Number Transformations.....	3-44
ABS.....	3-45
ACOS.....	3-45
ASIN.....	3-45
ATAN.....	3-46
ATAN2.....	3-46
COS.....	3-46
COSH.....	3-47
CEIL.....	3-47
EXP.....	3-47
FLOOR.....	3-48
LN.....	3-48
LOG.....	3-48
MOD.....	3-49
POWER.....	3-49
ROUND (number).....	3-50
SIGN.....	3-50
SIN.....	3-50
SINH.....	3-51
SQRT.....	3-51
TAN.....	3-51
TANH.....	3-52
TO_CHAR (number).....	3-52
TRUNC (number).....	3-53
WB.LOOKUP_NUM (on a number).....	3-53
WB.LOOKUP_NUM (on a varchar2).....	3-54
WB_IS_NUMBER.....	3-55
OLAP Transformations.....	3-55
WB_OLAP_LOAD_CUBE.....	3-56
WB_OLAP_LOAD_DIMENSION.....	3-56
WB_OLAP_LOAD_DIMENSION_GENUK.....	3-57
XML Transformations.....	3-58
WB_XML_LOAD.....	3-58
WB_XML_LOAD_F.....	3-58
Conversion Transformations.....	3-59
CASE.....	3-59

NVL	3-60
Other Transformations	3-61
NLS_CHARSET_DECL_LEN	3-61
NLS_CHARSET_ID	3-62
NLS_CHARSET_NAME	3-62
UID	3-62
USER	3-63

A Using Slowly Changing Dimensions

About Slowly Changing Dimensions	A-1
Case Study Scenario.....	A-2
Source System	A-2
Target System	A-3
Using Type 1 Slowly Changing Dimensions	A-4
Step 1: Populate the Surrogate Key	A-5
Step 2: Configure the Target Properties.....	A-5
Step 3: Generate Code.....	A-6
Using Type 2 Slowly Changing Dimensions	A-6
Step 1: Detect a Match	A-8
Step 2: Split Join Results	A-8
Step 3: Determine Merge Rows	A-9
Step 4: Use the Expression UPDATE_DELTA_ROW	A-9
Step 5: Use the Expression MERGE_DELTA_ROW	A-9
Step 6: Populate Surrogate Keys	A-12
Step 7: Configure Target Properties	A-12
Step 8: Generate Code.....	A-12
Using Type 3 Slowly Changing Dimension.....	A-13
Step 1: Detect a Match	A-13
Step 2: Populate Current Values	A-14
Step 3: Populate Previous Value Columns by Expression	A-14
Step 4: Populate Surrogate Keys	A-14
Step 5: Configure Target Properties	A-14
Step 6: Generate Code.....	A-15
Deploying and Loading Slowly Changing Dimensions	A-15

Index

Preface

This preface includes the following topics:

- [Purpose](#) on page ix
- [Documentation Accessibility](#) on page ix
- [Audience](#) on page x
- [How This Guide Is Organized](#) on page x
- [New in 10g Release 1 \(10.1\)](#) on page xi
- [Added in Release 9.0.4](#) on page xiii
- [Conventions](#) on page xvii
- [Related Publications](#) on page xvii
- [Contacting Oracle](#) on page xviii

Purpose

Oracle Warehouse Builder is a comprehensive toolset for individuals who move and transform data, develop and implement business intelligence systems, perform metadata management, or create and manage Oracle databases and metadata. This guide describes the functions and procedures that characterize Warehouse Builder transformations.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at

<http://www.oracle.com/accessibility/>

Accessibility of Code Examples in Documentation

JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should

appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Audience

This guide is intended for data warehouse practitioners, including:

- Business Intelligence application developers
- Warehouse architects, designers, and developers—especially SQL and PL/SQL developers
- Data analysts and those who develop extract, transform, and load routines
- Developers of large-scale products based on data warehouses
- Warehouse administrators
- System administrators
- Other MIS professionals

In order to use the information in this guide, you need to be comfortable with the concepts of Relational Database Management Systems and Data Warehouse design. For information on data warehousing, refer to the *Oracle9i Data Warehousing Guide*. Also, you need to be familiar with Oracle's relational database software products such as Oracle Database, SQL*Plus, SQL*Loader, Oracle Enterprise Manager, and Oracle Workflow.

How This Guide Is Organized

The transformations described in this manual are grouped according to the data or function type. For example, all transformations working on dates are grouped together in one section. All functions in each section are listed alphabetically to make searching more efficient. Each transformation contains the syntax as it reads in Warehouse Builder, a description of the purpose of this transform, and examples or business use cases.

Warehouse Builder includes a set of custom functions, built using PL/SQL to make common activities easier for developers. These transformations start with the prefix WB_ and are listed at the end of the chapter.

The *Oracle Warehouse Builder Transformation Guide* contains the following chapters and appendixes.

- [Chapter 1, "Introduction to Warehouse Builder Transformations"](#) introduces SQL and PL/SQL transformations.
- [Chapter 2, "Transformations"](#) describes packaged transformations available in the Mapping Editor and how to define them within a mapping.
- [Chapter 3, "SQL Transformations"](#) provides a reference for the SQL procedures and functions available in Warehouse Builder.
- [Appendix A, "Using Slowly Changing Dimensions"](#) provides a brief introduction to the different types of Slowly Changing Dimensions. It also goes through a case

study scenario to demonstrate how to use Warehouse Builder to design and deploy different types of Slowly Changing Dimensions.

New in 10g Release 1 (10.1)

Enhancements to the Mapping Editor: Mapping Debugger

Warehouse Builder now provides you with extensive debugging capabilities for your mappings from within the Mapping Editor. Use the Mapping Debugger to locate logical design errors in your mappings. The new features allow you to step through the data flow of a mapping using comprehensive debugging functions such as setting breakpoints and watches and interactively changing test data.

Enhanced Support of Multiple Targets: Correlated Commit

This release introduces a new commit strategy for mappings with multiple targets. In previous releases, Warehouse Builder performed independent commits. That is, Warehouse Builder committed and rolled back each target separately and independently of other targets. In addition to this option, Warehouse Builder now also performs correlated commits. Warehouse Builder considers all targets collectively and commits or rolls back data uniformly across all targets. Use the correlated commit when it is important to ensure that every row in the source impacts all affected targets uniformly.

Direct Partition Exchange Loading

In previous releases, Warehouse Builder by default created a temporary table for mappings that required additional processing of source data before exchanging partitions. This occurred when the mapping contained remote sources or multiple sources joined together. Beginning in this release, you can now by-pass the creation of a temporary table and directly swap a source into a target. Use Direct PEL in a mapping to instantaneously publish fact tables that you loaded in a previously executed mapping.

Data Quality Features

- **Multiple Name and Address Software Providers:** Beginning in this release, Warehouse Builder is compatible with multiple certified Name and Address software providers. Third-party vendors can license Name and Address software directly to you for use with Warehouse Builder. This enables you to choose a name and address provider whose offering is the most appropriate for your project.
- **Name-Address Operator Wizard:** In previous releases, you defined the Name-Address operator using the mapping canvas and the operator Configuration Properties sheet. For improved usability, Warehouse Builder now enables you to use a wizard and Operator editor to create and edit the Name-Address operator.
- **Match-Merge Operator:** Warehouse Builder incorporates the data quality functionality formerly available in Oracle Pure Integrate. You can use the Match-Merge operator available in the Mapping Editor to define business rules for matching and merging records. The Match-Merge operator together with the Name-Address operator support householding, the process of identifying unique households in name and address data.

Metadata Change Management

In a previous release, you could perform metadata change management using the OMB Plus scripting utility. Beginning in this release, you can also access these functions for the Warehouse Builder client user interface. Metadata change management enables you to take snapshots of metadata objects and use them for backup and history management. Snapshots are supported for any object on the navigation tree and can store information about an object alone (such as a table or module), or the objects within it as well (such as the tables within a module).

Extending Oracle Warehouse Builder Functionality

- **Security:** Warehouse Builder now provides advanced repository security and auditing options that you can implement according to your security requirements. The advanced security options include the following:

Proactive Security: Warehouse Builder enables you to plug in a customized security PL/SQL implementation package in the Warehouse Builder repository to provide tailored access control to users according to the security rules defined by your organization.

Reactive Security: Warehouse Builder enables you to track audit information based on the metadata history and to determine security policies from such audit information.

Data Stewardship: Warehouse Builder enables an individual or a group of individuals to “own” portions of the metadata rather than the technical administrators. Metadata ownership thus becomes an important component of metadata security management.

- **RAC Support:** With 10g Release 1 (10.1), Warehouse Builder provides increased support for RAC features. Warehouse Builder now supports the use of net service names in the runtime. This enables you to plan maintenance of nodes in a cluster without having to reconfigure the runtime environment. Warehouse Builder also provides an increased availability in the runtime service. For example, if either the service instance or its associated node fails or is taken out of service, then the runtime service instance on a different node can take over. While the Warehouse Builder design repository can also be used in a RAC cluster, it will not take advantage of any failover features of RAC for this release.

Enhancements to Flat File Support

- **ZONED Data Type Support:** Warehouse Builder now enables you to load fixed format data files containing ZONED decimal data. In the Flat File Sample Wizard, specify the ZONED data type for a flat file you import. The format for ZONED data is a string of decimal digits, one for each byte, with the sign included in the last byte. (In COBOL, this is a SIGN TRAILING field.) The length of this field is equal to the precision (number of digits) that you specify. You may also specify a scale, which is the number of digits to the right of the decimal point.
- **DECIMAL Data Type Support:** DECIMAL data is in packed decimal format; two digits for each byte, except for the last byte, which contains a digit and sign. The DECIMAL data type includes precision and scale and therefore can represent fractional values.

Enhancements in Database Connectivity

Warehouse Builder now enables you to create public database links that can be shared across a database. Public database links can be created by repository owners, as well as any user with the `CREATE PUBLIC DATABASE LINK` privilege.

Warehouse Builder Available on HP-UX and AIX

Starting with this release, Warehouse Builder is available on HP-UX and AIX platforms. This new availability is an addition to the UNIX (Solaris and Linux), and Windows (NT, 2000, and XP) platforms, which have been available from previous releases. (Note that the OLAP Bridges feature is only available on Windows platforms and the Name and Address Server is only available on Windows and Solaris platforms.)

Public Application Programming Interface

Starting in this release, Warehouse Builder now includes a public application programming interface (API). To access the API, unzip and extract the following file to a folder on your local machine:

```
<owb home directory>\owb\lib\int\pubapi_javadoc.jar
```

Double click the file `index.html`. Select the Help link for information on how to use the API.

Added in Release 9.0.4

The following new features were introduced in Oracle Warehouse Builder:

Changes in the Warehouse Builder Console

- **Enhanced Navigation Tree:** The navigation tree that displays in the Warehouse Builder console has been enhanced to improve navigation between projects and facilitate direct access to metadata repository objects. All projects are now visible from the tree, whereas previously you could only see one project at a time. Now you can expand a project node to display the contents of the active project. The module tree no longer appears in a separate window.
- **Wizards, Editors, and Properties Sheets:** All Warehouse Builder wizards, editors, and properties sheets are now launched from the navigation tree.
- **Business Areas Renamed to Collections:** In previous releases, you could create business areas in warehouse modules to organize objects in Warehouse Builder and to export metadata to tools such as Oracle Discoverer. Starting in this release, *collections* replace *business areas* in all functions and introduce enhancements, such as the ability to import metadata into and export metadata from a collection.
- **Fact Tables Renamed to Cubes:** The terms *fact* and *fact table* have been replaced with *cube* in this release to be in line with OLAP industry standards.
- **Logical Names Renamed to Business Names:** All references to *logical names* of objects have changed to *business names* in this release.
- **Toolbars in the Warehouse Builder Console:** The utility drawer has been removed and the side and top toolbars in the Warehouse Builder console have been merged at the top to consolidate the most important functionality in one place.

Enhancements to Deployment

- **Addition of Deployment Management Objects:** This release introduces three object types to assist in managing connections to deployment sources and targets: Locations, Connectors, and Runtime Repository Connections. Locations define the physical location of the deployment. Connectors define relationships between locations. Runtime Repository Connections provide information about Runtime

Repositories. Using these objects, you can create multiple deployment targets for the same target design.

- **Single Deployment Management Interface:** The Deployment Manager provides a single interface for managing deployments of all objects, and executions of deployed mappings, transformations, and process flows. It also provides immediate access to the history of previously deployed objects. Not only does the Deployment Manager enable you to perform all these tasks from one interface, but Warehouse Builder now keeps track of runtime metadata, providing you with the history of what has previously been deployed.

Enhancements to Warehouse Builder Metadata Browser

- **Design Metadata Browsing:** The Warehouse Builder Design Browser has been enhanced to include all new exposed objects, such as external tables, locations and connectors. In addition, you can now launch the Design Browser as a standalone executable; it no longer requires Oracle Application Server to be installed for a single-user usage.
- **Runtime Metadata Browsing:** The Warehouse Builder Runtime Audit Viewer has been replaced by the Runtime Audit Browser, which provides web-based reporting. The Runtime Audit Browser provides a more extensive set of deployment and execution audit reports than was available in previous releases. This audit data comes from information stored in the Runtime Repository and includes both deployment and execution data.

Enhancements to Warehouse Builder Programmatic Access

- **Warehouse Builder Public APIs:** Starting with this release, Warehouse Builder offers this alternative for programmatic access to Oracle Warehouse Builder features: a full set of Java public APIs for application programmers who want to embed Warehouse Builder features and services in their own applications.
- **Warehouse Builder Scripting Language:** Oracle MetaBase (OMB) Scripting Language provides access to all Warehouse Builder functions without accessing the Warehouse Builder graphical user interface. Users can access Warehouse Builder metadata and functionality by using OMB Plus, Warehouse Builder's scripting utility. This gives developers the power of using Warehouse Builder programmatically and extending its functionality where required. For more information on OMB Scripting Language, refer to the *Oracle Warehouse Builder Scripting Reference*.

Enhancements to Metadata Management

- **Security:** Warehouse Builder now provides an optional repository security and auditing system that you can implement according to your security requirements. You can create a multiple user account system where multiple identifiable users can access the same Warehouse Builder repository. Warehouse Builder also enables you to plug in a customized security PL/SQL implementation package in the Warehouse Builder repository to provide tailored access control to users according to the security rules defined by your organization.
- **Metadata Change Management (Metadata Snapshots):** Starting in this release, you can take snapshots of metadata objects and use them for backup and history management. Snapshots are supported for any object on the navigation tree and can store information about an object alone (such as a table or module), or the objects within it as well (such as the tables within a module).

- **Multiple Language Support (MLS):** With this feature, you can store the displayed business names and descriptions in languages other than the base language of the repository. Your different translations of business names and descriptions can be used to deploy to an EUL in the language of the target user population.
- **Extensibility Through User-Defined Properties:** Users can define additional properties for any Warehouse Builder objects using the Warehouse Builder OMB Plus scripting utility. After you define user-defined properties through scripting, you can access them in the user interface, the Oracle MetaBase (OMB) Scripting Language, Warehouse Builder Java APIs, and Warehouse Builder Design Browser. This enhances the extensibility of Warehouse Builder and makes it easier to integrate it with other Business Intelligence products.
- **Metadata Loader (Import and Export) Flexibility Enhancements:** Two new features were added to enhance this area of the product. The first is the ability for you to export metadata directly from Collections. The second feature is available from the Metadata Loader command line utility. It provides you with flexibility to specify the type of actions you want to apply when you import a first-class object.

Process Flow Editor

Starting in this release, you can use the Process Flow Editor in Warehouse Builder to create and define process flows. External process operators that you previously defined in mappings are upgraded to user-defined processes and are contained within a process flow module. Process flows now integrate in the same Warehouse Builder design environment and no longer require you to use Oracle Workflow design client to perform these functions. The Warehouse Builder process flow modeler natively understands the semantic of your mappings and enables you to model activities such as FTP, email, and so on.

Performance Improvements

- **Mapping User Interface:** A new pre-defined display set, named Mapping, was added in this release. Selecting this display set causes the Mapping to only display columns that effectively are mapped, or used.
- **Mapping Compression:** This feature automatically detects unused connections between operators and attributes in any given mapping and eliminates them from the repository. This dramatically enhances the performance of loading and storing large mappings that represent significant data flows.
- **Metadata Loader (Import and Export):** Import and export functionality now takes advantage of the new compression feature available for each mapping. This means that the Metadata Loader now exports and imports only those mapping objects that are actually used.

Oracle Database Integration

- **OLAP Integration:** Warehouse Builder enables you to design, deploy, and load multidimensional OLAP objects as ROLAP or MOLAP models from different data sources. After the data is loaded, you can use BI tools and applications to run complex analytical queries that answer your business questions. Using Warehouse Builder, you can now create and manage both your relational and multidimensional objects from the same cube and dimension designs.
- **Advanced Queue (AQ) Integration:** Warehouse Builder enables you to import Advanced Queue definitions and to use AQs as data sources and targets while designing your data warehouse. Through Advanced Queue functionality coupled with the Messaging Gateways, Warehouse Builder enables you to support

messaging applications on MQ Series and Tibco as Warehouse Builder data sources. AQs also enable you to propagate change data capture from your source system to your target. The ability to integrate AQs lays the foundation for providing real time data warehousing in the future.

- **External Tables:** Starting in this release, you can use external tables to represent data from non-relational file sources in a relational, read-only format. You can import an existing external table from an Oracle database. Or you can create an external table in Warehouse Builder based on a flat file definition. Warehouse Builder will generate the right DDL for you to deploy your external table to an Oracle database.
- **Oracle Database Multiple Table Inserts:** Warehouse Builder takes advantage of Oracle Database functionality and generates a multiple-table insert statement when the target is Oracle Database. This enables you to optimize mappings to insert data into multiple tables in one operation.
- **Oracle Database Table Functions:** Warehouse Builder introduces the Table Function operator that enables you to improve performance when loading your target system. Use this operator to develop custom code that can manipulate a set of input rows and return another set of rows possibly of different cardinality. Unlike conventional functions, table functions output a set of rows that can be queried like a physical table.

Enhanced Support for Flat Files

- **Unbound Flat Files as Targets:** In this release, you can create a new, unbound flat file object as you create your mapping. Warehouse Builder creates a new comma-delimited, single-record-type flat file in the specified location. This feature makes it easier to load the contents of a relational object into a flat file.
- **Outbound Reconcile for Flat Files:** Outbound reconciliation makes it possible to create a new repository object from a mapping flat file. This results in a new, comma-delimited file to be created where specified, if the flat file is new to that repository. This feature makes it easier to "quickly dump" the contents of a relational object to a flat file.
- **Logical Records for Delimited Files:** The Flat File Sample Wizard has also been enhanced to display an improved user interface that enables you to define logical records for delimited files.
- **Position-Based Master-Detail Loading:** Position-based master-detail flat files are now easier to load with the use of additional mapping operators.
- **SQL Property Extensions:** You can now specify SQL properties for flat files you import into Warehouse Builder. This enables you to pre-define SQL property values for each flat file field. Thus, if mapping a flat file source to a relational target, the target column will default to these pre-defined SQL property values. These values will be used when building a relational target column or when creating an external table column.

Mapping Editor Enhancements

- **Mapping User Interface:** A new set of property tabs is now available for you to quickly create and edit mapping operators and attribute properties.
- **Pivot and Unpivot Operators:** Starting in this release, you can add a pivot operator or an unpivot operator to a mapping. The pivot operator enables you to transform a single row of attributes into multiple rows. The unpivot operator converts multiple input rows into one output row.

- **Name and Address Operator Enhancements:** The Name and Address operator has been enhanced to include new input roles and output attributes. The United States Postal Service Code Accuracy Support System (CASS) reporting is also supported starting with this release.

Warehouse Builder Is Now Available on UNIX Platforms

Starting with this release, Warehouse Builder is available on UNIX (Solaris, and Linux), as well as Windows (NT, 2000, and XP) platforms. This applies to all the components of Warehouse Builder, with the exception of the Name and Address libraries, which are not available on Linux in this release. (Note that the OLAP Bridges feature is only available on Windows platforms and the Name and Address Server is only available on Windows and Solaris platforms.)

Conventions

In this manual, Windows refers to the Windows NT, Windows 2000, and Windows XP operating systems. The SQL*Plus interface to Oracle Database may be referred to as SQL.

In the examples, an implied carriage return occurs at the end of each line, unless otherwise noted. You must press the Return key at the end of a line of input.

The following table lists the conventions used in this manual.

Convention	Meaning
. . . .	Vertical ellipsis points in an example mean that information not directly related to the example has been omitted.
...	Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted.
boldface text	Boldface type in text refers to interface buttons and links. Boldface type also serves as emphasis to set apart main ideas.
<i>italicized text</i>	Italicized text applies to new terms introduced for the first time. Italicized text also serves as an emphasis on key concepts.
unicode text	Unicode text denotes exact code, file directories and names, and literal commands.
<i>italicized unicode text</i>	Italicized unicode text refers to parameters whose value is specified by the user.
[]	Brackets enclose optional clauses from which you can choose one or none.

Related Publications

The Warehouse Builder documentation set includes these manuals:

- *Oracle Warehouse Builder User's Guide*
- *Oracle Warehouse Builder Installation and Configuration Guide*
- *Oracle Warehouse Builder Transformation Guide*
- *Oracle Warehouse Builder Scripting Reference*
- *Oracle Warehouse Builder Release Notes*

In addition to the Warehouse Builder documentation, you can refer to the *Oracle Data Warehousing Guide*.

Oracle provides additional information sources, including other documentation, training, and support services that can enhance your understanding and knowledge of Oracle Warehouse Builder.

- For more information on Oracle Warehouse Builder technical support, contact Oracle World Wide Support services at:

<http://www.oracle.com/support>

- For the latest information on, and downloads of, software and documentation updates to Oracle Warehouse Builder, visit MetaLink at:

<http://metalink.oracle.com>

- You can order other Oracle documentation at:

<http://oraclestore.oracle.com>

Contacting Oracle

OracleMetaLink

OracleMetaLink is the Oracle support Web site where you can find the latest product information, including documentation, patch information, BUG reports, and TAR entries. Once registered, you can access email, phone and Web resources for all Oracle products. *OracleMetaLink* is located at:

<http://metalink.oracle.com>

Check *OracleMetaLink* regularly for Warehouse Builder information and updates.

Documentation

You can order Oracle product documentation by phone or through the World Wide Web:

- **Phone:** Call 800-252-0303 to order documentation or request a fax listing of available Oracle documentation.

- **Oracle Documentation Sales Web site:**

<http://oraclestore.oracle.com>

- **Oracle Support Services Web site:**

<http://www.oracle.com/support>

Introduction to Warehouse Builder Transformations

Transforming data is one of the main functions of an Extract, Transformation, and Loading (ETL) tool. Data transformations can range from single-value-based arithmetic calculations to multiple data transformations, such as data manipulation and alterations. Oracle Warehouse Builder includes several pre-defined transformations, as well as a library of predefined functions and procedures, to transform data. Warehouse Builder also uses SQL and PL/SQL as the transformation languages, which enables it to fully utilize the power of the Oracle database engine.

This guide describes the functions and procedures that characterize Warehouse Builder transformations. Because this guide is not intended as a basic user manual, you must have an understanding of how to create mappings and how to add transformations and operators to mappings. Additionally, concepts such as the Global Shared Library are not discussed in this manual.

This chapter includes the following topics:

- [Overview](#) on page 1-1
- [Transforming Data with Warehouse Builder](#) on page 1-2
- [SQL Standards](#) on page 1-2
- [How SQL Works](#) on page 1-2
- [SQL as the Common Language for Relational Databases](#) on page 1-3

For more information, see the *Oracle Warehouse Builder User's Guide*.

Overview

Oracle offers a comprehensive ETL solution, and one of the major components in this solution is the ETL tool, Oracle Warehouse Builder. Warehouse Builder provides components that enable you to model and create an ETL process. Its intuitive user interface (UI) enables you to design and define objects that are stored in an open repository. After completing the initial design, you can deploy this design to the runtime platform. Because the runtime platform is the Oracle database, Warehouse Builder is used as a tool that generates code rather than an ETL engine-based tool. Warehouse Builder also provides reporting tools to run reports on your repository. And it also enables you to integrate with other Oracle query tools.

Because SQL and PL/SQL are versatile and proven languages widely used by many information professionals, the time and expense of developing an alternative transformation language is eliminated by using Warehouse Builder. With Warehouse

Builder, you can create solutions using existing knowledge and a proven, open, and standard technology.

Transforming Data with Warehouse Builder

The ETL processes designed with Warehouse Builder can be translated into PL/SQL packages. These PL/SQL packages are deployed to the Oracle database and stored as packages available for execution.

You can also use PL/SQL to transform data moving from sources to targets. To enable faster development of warehousing solutions, Warehouse Builder provides custom procedures and functions written in PL/SQL. Warehouse Builder enables you to reuse PL/SQL as well as to write your own PL/SQL transformations. Because the final process runs on the Oracle database, Warehouse Builder supports all constructs supported by the Oracle database.

You can also use the Warehouse Builder Mapping Editor to design data transformations using SQL code components. For example, activities such as joining disparate data sources or splitting data streams into multiple output streams can be implemented as SQL components. This enables Warehouse Builder to generate efficient SQL code to move data from source to target.

SQL Standards

Oracle Corporation strives to comply with industry-accepted standards and participates actively in SQL standards committees. Industry-accepted committees include the American National Standards Institute (ANSI) and the International Standards Organization (ISO), which is affiliated with the International Electrotechnical Commission (IEC). Both ANSI and the ISO/IEC have accepted SQL as the standard language for relational databases. When a new SQL standard is simultaneously published by these organizations, the names of these standards conform to conventions used by the organization. Although naming conventions may differ, it should be noted that the standards are technically identical.

The latest SQL standard was adopted in July 1999 and is often called SQL:99. The formal names of this standard are:

- ANSI X3.135-1999, "Database Language SQL", Parts 1 ("Framework"), 2 ("Foundation"), and 5 ("Bindings")
- ISO/IEC 9075:1999, "Database Language SQL", Parts 1 ("Framework"), 2 ("Foundation"), and 5 ("Bindings")

How SQL Works

SQL is a data sublanguage that provides an interface to a relational database such as Oracle. All SQL statements are instructions to the database. SQL benefits different types of users including application programmers, database administrators, managers, and end users.

SQL language:

- Processes sets of data as groups rather than as individual units.
- Provides automatic navigation to the data.
- Uses standalone statements that are complex and powerful.

Flow-control statements were not originally part of SQL, but they can be found in the recently accepted optional part of SQL, ISO/IEC 9075-5: 1996. Flow-control statements are commonly known as "persistent stored modules" (PSM), and Oracle's PL/SQL extension to SQL is similar to PSM.

SQL enables you work with data at the logical level. The implementation details are only required when you want to manipulate the data. For example, to retrieve a set of rows from a table, you can define a condition to filter the rows. All rows satisfying the condition are retrieved in a single step and can be passed as a unit to the user, to another SQL statement, or to an application. You do not have to filter the rows one by one, or manually store and retrieve the data. All SQL statements use the *optimizer*, a part of Oracle that determines the most efficient means of accessing the specified data. Oracle also provides techniques that you can use to improve the optimizer performance.

SQL provides statements for different tasks including:

- Querying data
- Inserting, updating, and deleting rows in a table
- Creating, replacing, altering, and dropping objects
- Controlling access to the database and its objects
- Guaranteeing database consistency and integrity

SQL unifies all of the preceding tasks in one consistent language.

SQL as the Common Language for Relational Databases

All major relational database management systems support SQL and all programs written in SQL can be moved from one database to another with very little modification.

This means that all the SQL knowledge in your organization is fully portable to Warehouse Builder. Warehouse Builder enables you to import and maintain any existing complex custom code. You can later use these custom transformations in Warehouse Builder mappings.

Transformations

The Warehouse Builder Mapping Editor includes a selection of pre-built transformation operators. These operators enable you to define common transformations when you define how the data will move from source to target.

Transformation Operators are pre-built PL/SQL functions, procedures, package functions, and package procedures. They take input data, perform operations on it, and produce output data.

This chapter contains the following topics:

- [Regular SQL Operators](#) on page 2-1
- [Data Cleansing Operators](#) on page 2-33

For related information, see the *Oracle Warehouse Builder User's Guide*.

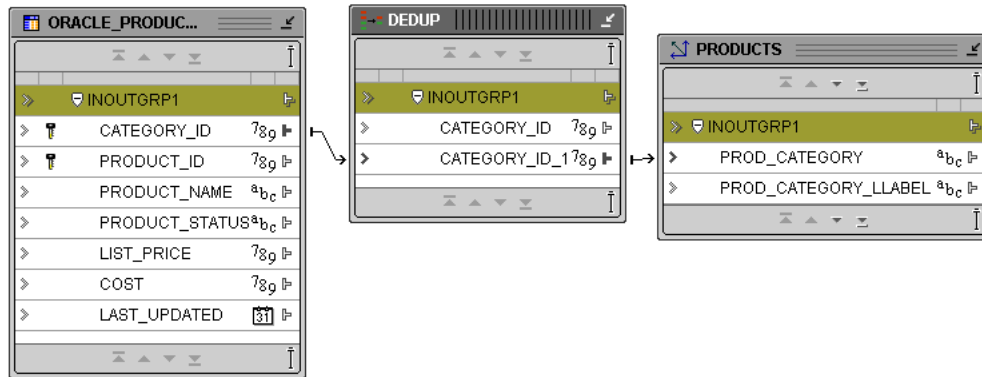
Regular SQL Operators

SQL is a powerful mechanism to extract and transform data. You can use SQL to join disparate sources into one data stream, transform the joined data, and then split it into multiple streams output to multiple targets. Warehouse Builder enables these activities by using regular SQL operations.

Deduplicator (DISTINCT)

In a large number of cases the source data contains duplicate values. For example, higher levels within a dimension are duplicated in the source because they are at a lower level of granularity. When selecting these rows, the goal is to only select one row for each level record, not multiple ones.

This can be achieved by adding a deduplicator operator to a mapping, as shown in [Figure 2-1](#). All rows that are required by the target must pass through the deduplicator. No row set can bypass the deduplicator and hit the target directly.

Figure 2–1 Deduplicator in a Mapping

A deduplicator results in a `DISTINCT` keyword in the generated extraction query, as shown in [Figure 2–2](#).

Figure 2–2 Translation to `DISTINCT` Keyword

```

Code Viewer: TRANSFORMATIONS [0 error(s), 3 warning(s)]
Code Edit Search View
39 (SELECT
40 /*+ DRIVING_SITE("DEDUP"."ORACLE_PRODUCT_INFO_DRUGSTORE") */
41 "DEDUP"."CATEGORY_ID_1&0" "CATEGORY_ID_1"
42 FROM
43 (SELECT
44 DISTINCT
45 NULL "CATEGORY_ID",
46 "ORACLE_PRODUCT_INFO_DRUGSTORE"."CATEGORY_ID" "CATEGORY_ID_1&0"
47 FROM
48 "ORACLE_PRODUCT_INFO"@("DRUGSTORE" "ORACLE_PRODUCT_INFO_DRUGSTORE") "DEDUP"
49 );
Line 39 Column 9 Read Only [PL/SQL] Set based Windows: CR/LF

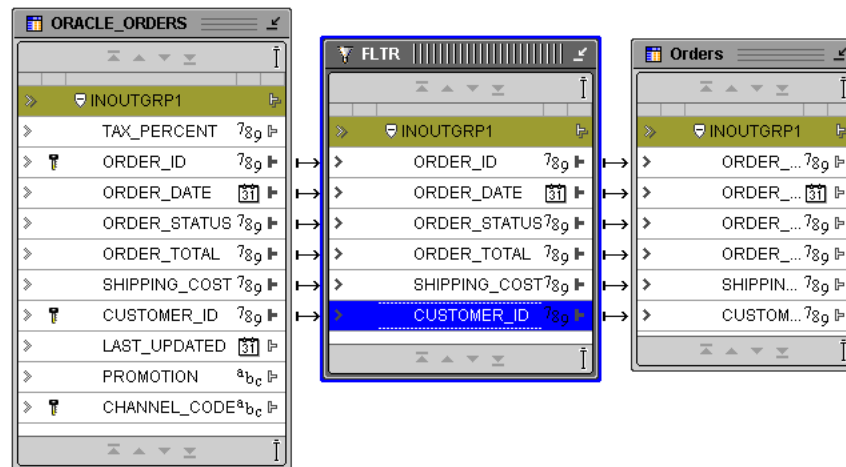
```

Filter (WHERE)

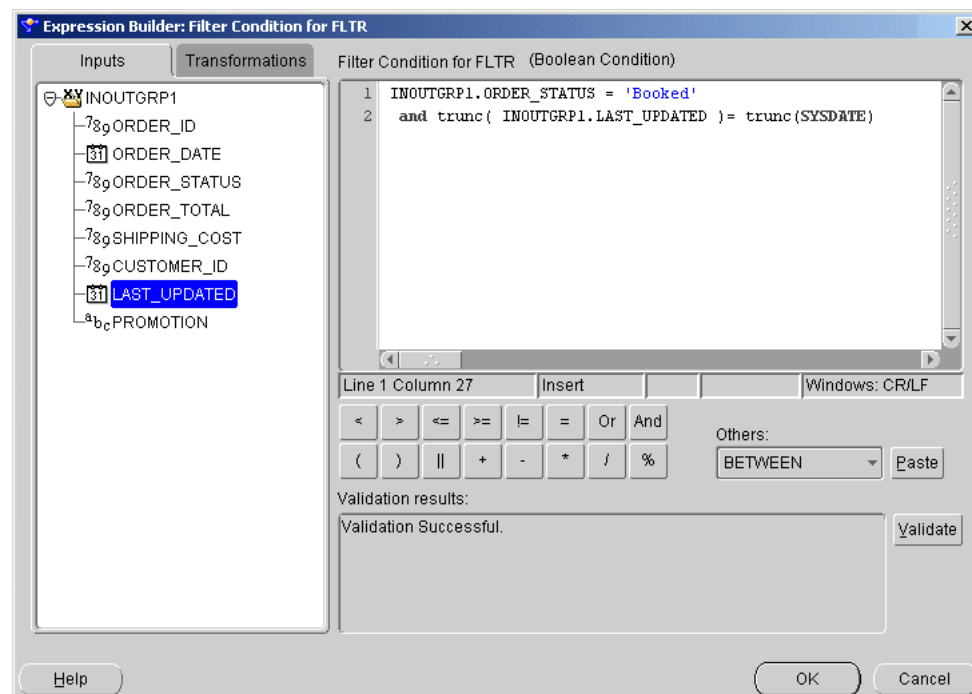
A filter enables you to restrict the data set selected from the query source. Filtering limits the number of rows to be extracted or processed based on a clause applied to a set of data. This filter clause can be based on all supported data types and can contain constants.

In [Figure 2–3](#), the orders are restricted using the order status. The orders must be booked and the last updated date must be the date of extraction. The result is truncated to ensure that matches are done on the date without the timestamp. Therefore, the result only loads the booked orders that were modified (or set to “booked”) on the day of loading. This is an easy way of implementing change data capture.

All rows that are required at the target must pass through the filter operator. No row set can bypass the filter and hit the target directly.

Figure 2–3 Filter in a Mapping

After defining a filter in the mapping, you must specify the filter clause using the Expression Builder, as shown in [Figure 2–4](#). Because filter conditions can be complex, you can use the Expression Builder to validate the filter clause before deploying it to any target system.

Figure 2–4 Expression Builder with the Filter Clause Defined

Warehouse Builder translates the filter clause into a WHERE clause in the extraction program, as shown in [Figure 2–5](#).

Figure 2–5 The Generated Code for Join operator

```

Code Viewer: TRANSFORMATIONS [INOUTGRP1] [0 error(s), 2 warning(s)]
Code Edit Search View
13 (SELECT
14 /*+ DRIVING_SITE("ORACLE_ORDERS_DRUGSTORE") */
15 "ORACLE_ORDERS_DRUGSTORE"."ORDER_ID" "ORDER_ID",
16 "ORACLE_ORDERS_DRUGSTORE"."ORDER_DATE" "ORDER_DATE",
17 "ORACLE_ORDERS_DRUGSTORE"."ORDER_STATUS" "ORDER_STATUS",
18 "ORACLE_ORDERS_DRUGSTORE"."ORDER_TOTAL" "ORDER_TOTAL",
19 "ORACLE_ORDERS_DRUGSTORE"."SHIPPING_COST" "SHIPPING_COST",
20 "ORACLE_ORDERS_DRUGSTORE"."CUSTOMER_ID" "CUSTOMER_ID",
21 "ORACLE_ORDERS_DRUGSTORE"."LAST_UPDATED" "LAST_UPDATED",
22 "ORACLE_ORDERS_DRUGSTORE"."PROMOTION" "PROMOTION"
23 FROM
24 "ORACLE_ORDERS"@ "DRUGSTORE" "ORACLE_ORDERS_DRUGSTORE"
25 WHERE
26 ( "ORACLE_ORDERS_DRUGSTORE"."ORDER_STATUS" = 'Booked' ) AND
27 ( trunc ( "ORACLE_ORDERS_DRUGSTORE"."LAST_UPDATED" ) = trunc ( SYSDATE ) )
28 )
Line 1 Column 1 Read Only [PL/SQL] Set based/Loading Windows: CR/LF

```

The generated extraction code contains the physical column names of the objects whereas the Expression Builder shows you the relative names. If you are using business names in the logical modeling phase, Warehouse Builder automatically translates them into physical names of actual database objects.

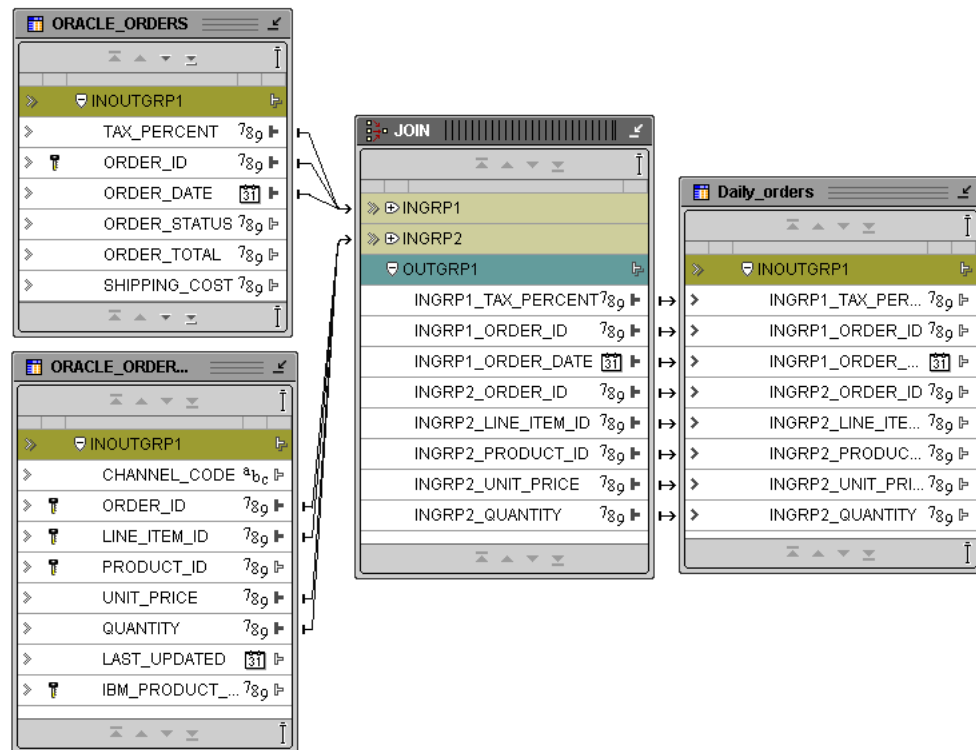
Joiner (FULL OUTER JOIN)

A join is a query that combines rows from two or more tables, views, or materialized views. Oracle performs a join whenever multiple tables appear in the query's FROM clause. The query's select list can select any columns from any of these tables.

Most join queries contain WHERE clause conditions that compare two columns, each from a different table. To execute a join, Oracle combines pairs of rows, each containing one row from each table, for which the join condition evaluates to TRUE. In addition to join conditions, the WHERE clause of a join query can also contain other conditions that refer to columns of only one table. These conditions can further restrict the rows returned by the join query.

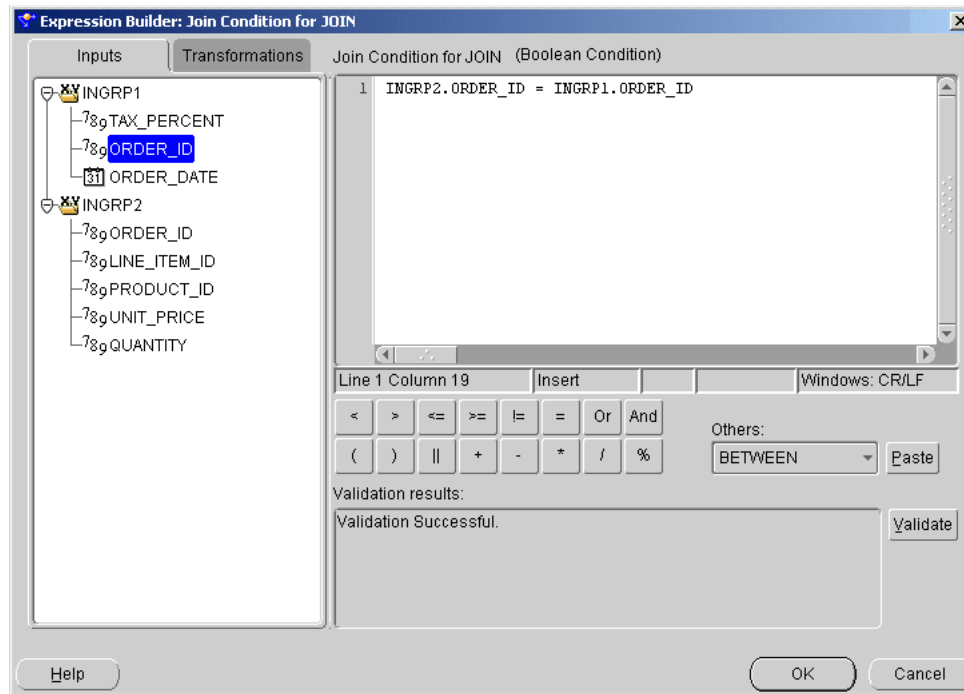
The joiner in Warehouse Builder also supports inner joins, outer joins, self joins, equijoins, and non-equijoins. When run on Oracle9i, Warehouse Builder supports full outer joins. The Key Lookup operator is an example of an outer join used in Warehouse Builder. For more information on joins, see the *Oracle SQL Reference*.

Figure 2–6 Joining Two Related Tables Into a Single Result Set

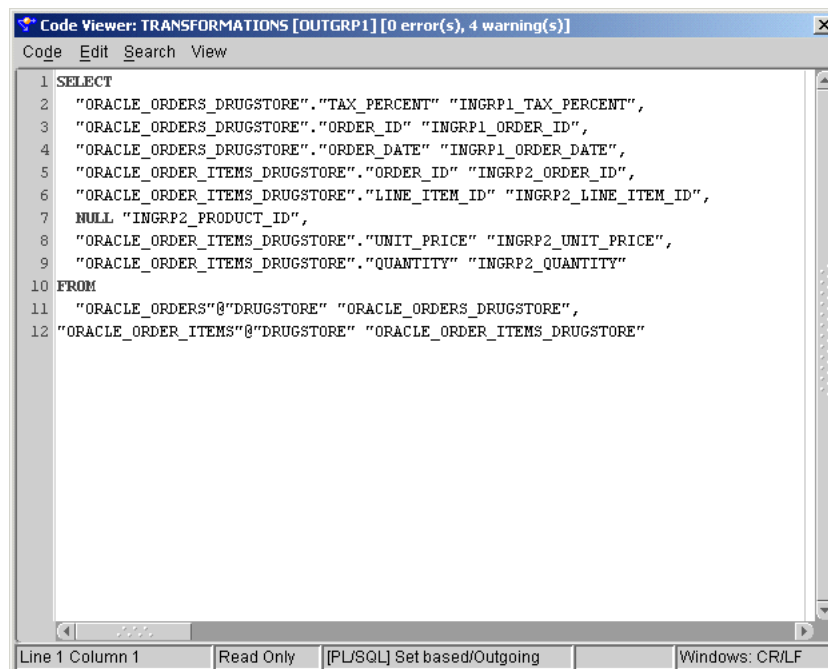


If the metadata in the Warehouse Builder repository already contains a primary to foreign key relationship, Warehouse Builder pre-populates the join condition based on that information. In [Figure 2–6](#), two source tables are joined to combine the data from a set of normalized order tables into one table. Because the order lines are typically of a higher cardinality (there are more records for each order in the lines table), the result set is also of that higher cardinality.

If two tables in a join query have no join condition, Warehouse Builder returns their Cartesian product and combines each row of one table with each row of the other table. Because a cartesian product always generates many rows, it may not be useful. For example, the Cartesian product of two tables each with 100 rows has 10,000 rows. You must always include a join condition unless you need a Cartesian product.

Figure 2-7 The Join Condition Based on the PK - FK Relationship

Using the Expression Builder, as shown in [Figure 2-7](#), you can define infinitely complex join clauses. Warehouse Builder translates these clauses into WHERE clauses in the generated SQL code, as shown in [Figure 2-8](#). In this example, the FROM clause contains both source tables and the WHERE clause joins these tables on the `order_id` columns.

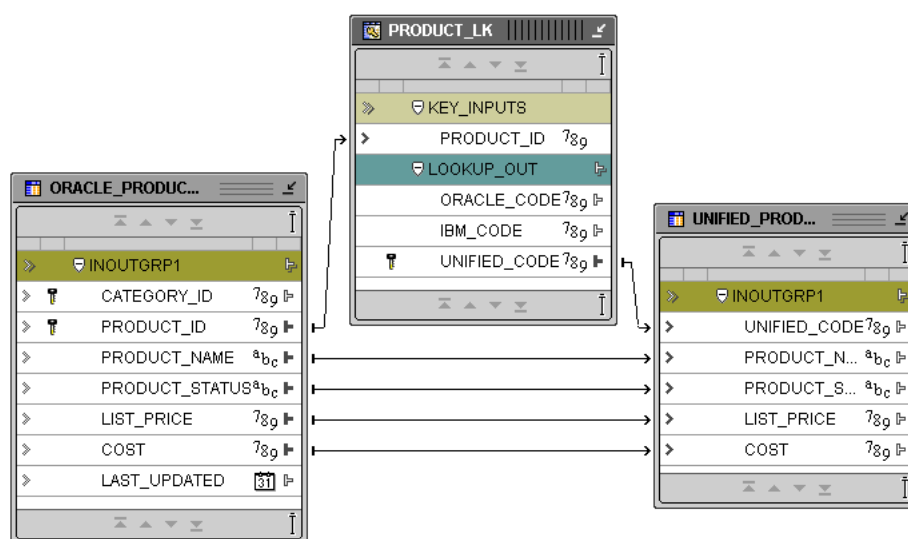
Figure 2-8 The Generated Code for Join operator

Key Lookup

A common design decision in a data warehouse is the use of surrogate keys. These keys, typically integers, are used to replace larger logical identifiers taken from the source systems. You can use surrogate keys to reduce the space used by tables linking to a primary key or to substitute source specific identifiers with common identifiers to enable reporting. For information on creating numerical surrogate keys, see "Sequence (CURRVAL, NEXTVAL)" on page 2-11.

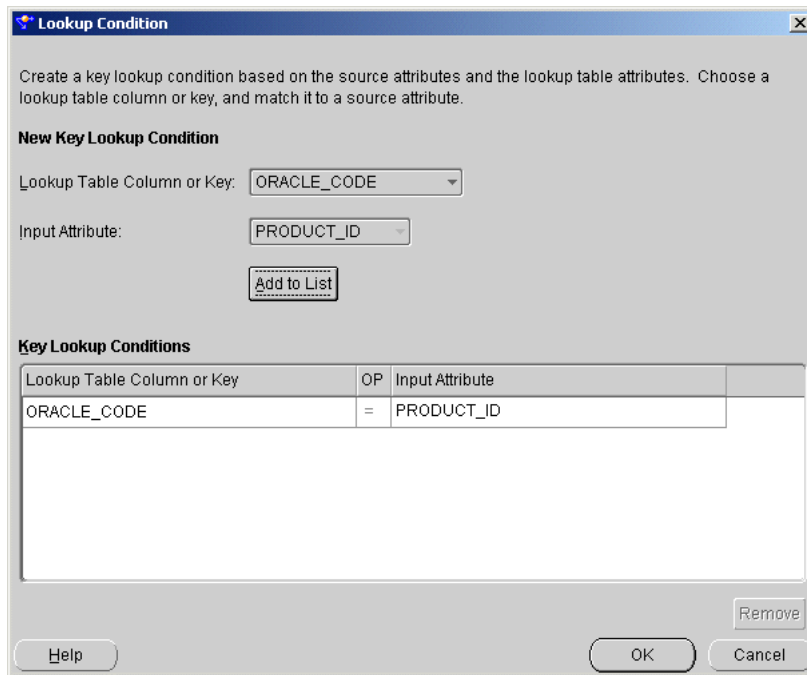
When you use surrogate keys, there is a mismatch between the primary identifier of a record in the source and in the target. To ensure that joins in the target are performed on the correct data, a key lookup needs to be performed. The key lookup transforms the logical key into its surrogate equivalent. [Figure 2-9](#) shows an example of a key lookup in a mapping.

Figure 2-9 Key Lookup in a Mapping

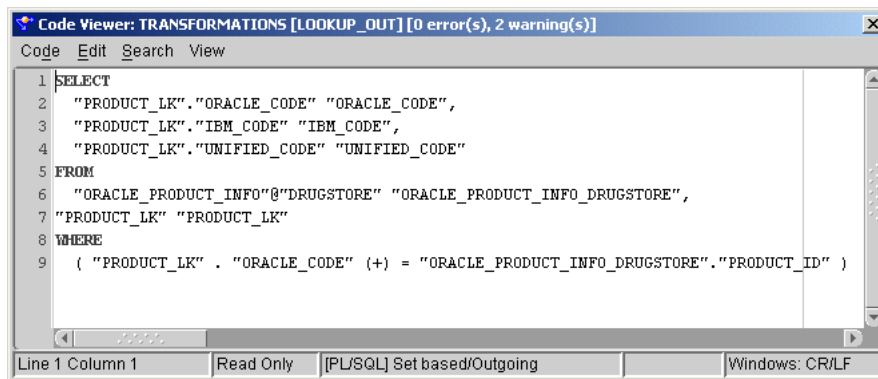


Warehouse Builder provides two ways of performing a key lookup: you can use pre-defined PL/SQL transformations or you can use a SQL operator. The SQL operator joins a lookup object, such as a table, view, materialized view, or dimension, to the table containing the original identifier.

For lookup conditions, you do not need to manually create the join clauses. A specialized UI enables you to specify these clauses, as shown in [Figure 2-10](#). For each output attribute on the key lookup operator, you can specify a default.

Figure 2–10 Defining a Lookup Condition

To avoid over-restriction (and missing source rows), the lookup table is outer joined by the operator to ensure that all source rows are part of the query. If no lookup value is available, the operator returns a NULL value. You can substitute this with a default value that uses NVL in the query to substitute the NULL with the default.

Figure 2–11 Generated Code With OUTER JOIN And NVL

In [Figure 2–11](#), the outer join is performed on the PRODUCT_LK table which is the lookup table. Rows returning NULL for the UNIFIED_CODE are substituted through use of the NVL function to reflect a -1 value. Thus, the default value does not interfere with the system-generated codes. For details, see [NVL](#) on page 3-61.

Pivot Operator

The pivot operator enables you to transform a single row of attributes into multiple rows. Use this operator in a mapping when you want to transform data that is contained across attributes instead of rows. For example, when you extract data from non-relational data sources, such as data in a crosstab format.

Example: Pivoting Sales Data

The external table `SALES_DAT` contains data from a flat file. This data contains one row for each sales representative and separate columns for each month. For more information about external tables, see *Oracle Warehouse Builder User's Guide*.

Figure 2–12 shows the data in `SALES_DAT` in a crosstab format.

Figure 2–12 SALES_DAT

ID	Reg	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12
0675	4	10.5	11.4	9.5	8.7	7.4	7.5	7.8	9.7				
0676	3	9.5	10.5	10.3	7.6	8.0	7.8	8.7	8.9				
0679	3	8.7	7.4	7.5	7.8	9.7	10.3	7.6	8.0				
0683	2	9.5	10.5	10.3	9.5	8.7	7.4	7.8	8.7				
0684	1	11.4	9.5	8.7	7.4	7.5	10.3	9.5	8.7				
0687	1	9.5	8.7	7.4	7.8	8.7	7.4	7.5	7.8				
0690	1	8.7	7.4	7.8	8.7	11.4	9.5	8.7	7.4				

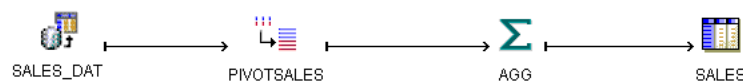
Table 2–1 shows a sample of the data after Warehouse Builder has performed a pivot operation. The data that was formerly contained across multiple columns (M1, M2, M3...) is now contained in a single attribute (`MONTHLY_SALES`). A single ID row in `SALES_DAT` corresponds to 12 rows in pivoted data.

Table 2–1 Pivoted Data

REP	MONTH	MONTHLY_SALES	REGION
0675	Jan	10.5	4
0675	Feb	11.4	4
0675	Mar	9.5	4
0675	Apr	8.7	4
0675	May	7.4	4
0675	Jun	7.5	4
0675	Jul	7.8	4
0675	Aug	9.7	4
0675	Sep	NULL	4
0675	Oct	NULL	4
0675	Nov	NULL	4
0675	Dec	NULL	4

To perform the pivot transformation in this example, you can create a mapping as shown in Figure 2–13.

Figure 2–13 Pivot Operator in a Mapping



In this mapping, Warehouse Builder reads the data from the external table, pivots the data, aggregates the data, and writes it to a target in set-based mode. It is not necessary

to load the data to a target directly after pivoting it. You can use the pivot operator in a series of operators before and after directing the data into the target operator. You can place operators such as a `FILTER`, `JOIN`, and `SET OPERATION` before the pivot operator. Because pivoted data in Warehouse Builder is not a row-by-row operation, you can also execute the mapping in set-based mode.

Sequence (CURRVAL, NEXTVAL)

A sequence enables you to generate sequential numbers from the database. It also enables you to create identifiers without a real semantic meaning. These identifiers are often called surrogate keys. You can then use Key Lookup within Warehouse Builder to deduce this generated key through a lookup on the original value.

When a sequence number is generated, the sequence is incremented independent of whether you commit or rollback the transaction. If two users concurrently increment the same sequence, the sequence numbers each user acquires may have gaps because the sequence numbers are also being generated by the another user. One user can never acquire the sequence number generated by the other user. After a user generates a sequence value, that user can continue to access that value whether the sequence is incremented by another user or not.

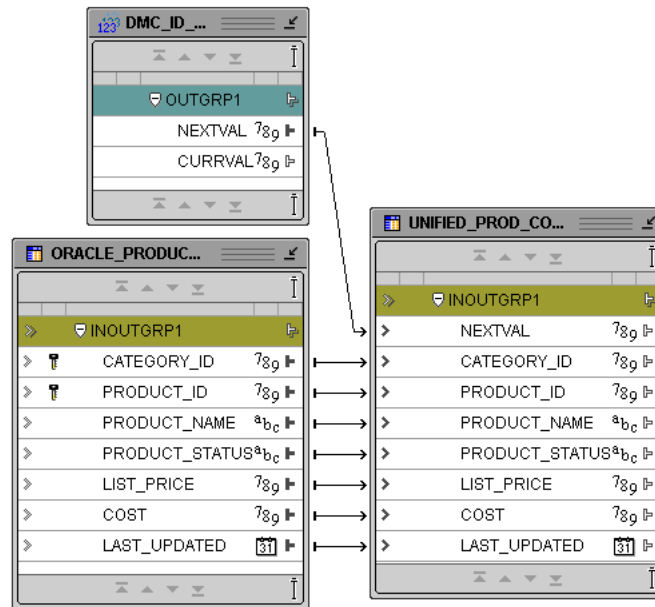
Because sequence numbers are generated independently of tables, the same sequence can be used for one or multiple tables. Individual sequence numbers may appear to be skipped if they are generated and used in a transaction that is later rolled back. Additionally, a single user may not realize that other users are drawing from the same sequence.

You can access the sequence values in SQL statements with the `CURRVAL` pseudocolumn (which returns the current value of the sequence) or the `NEXTVAL` pseudocolumn (which increments the sequence and returns the new value).

Note: To use the `CURRVAL` functionality, a `NEXTVAL` call has to be made first.

[Figure 2–14](#) shows you how to use sequences in a Warehouse Builder mapping.

Figure 2–14 Generating Surrogate Keys



In this example, the pseudocolumns CURRVAL and NEXTVAL are available in the sequence object in Warehouse Builder and you can choose the appropriate column to map. The NEXTVAL column is commonly used to generate the insert statement, as shown in Figure 2–15.

Figure 2–15 Generated Code for Sequence Operator

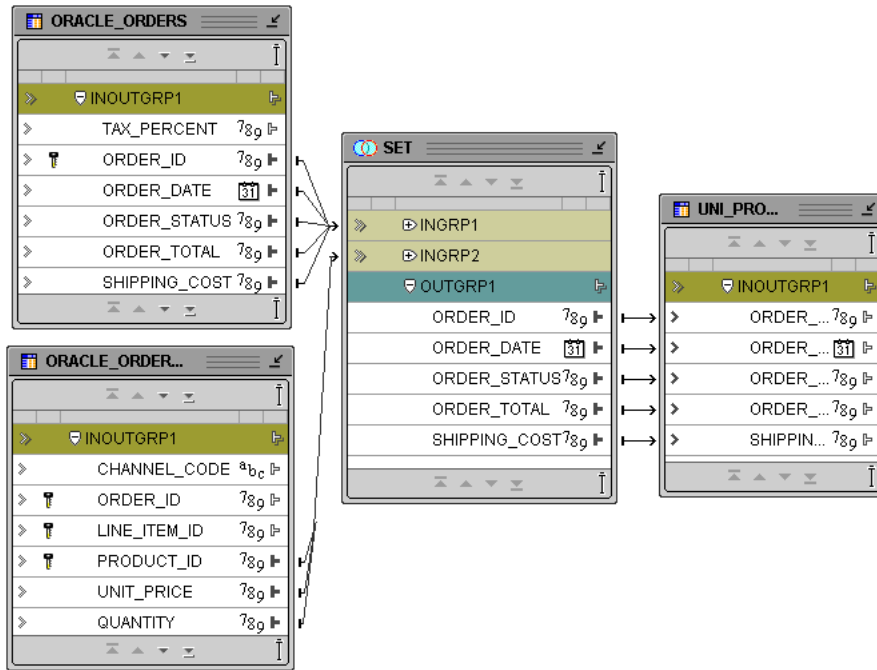
```
Code Viewer: TRANSFORMATIONS [INOUTGRP1] [0 error(s), 14 warning(s)]
Code Edit Search View
1 INSERT
2 /*+ APPEND PARALLEL("UNIFIED_PROD_CODES", DEFAULT, DEFAULT) */
3 INTO
4 "UNIFIED_PROD_CODES"
5 ("NEXTVAL_1",
6 "CATEGORY_ID",
7 "PRODUCT_ID",
8 "PRODUCT_NAME",
9 "PRODUCT_STATUS",
10 "LIST_PRICE",
11 "COST",
12 "LAST_UPDATED")
13 (SELECT
14 /*+ NO MERGE */
15 "DMC_ID_SEQ".NEXTVAL "NEXTVAL",
16 "ORACLE_PRODUCT_INFO_DRUGSTORE"."CATEGORY_ID" "CATEGORY_ID",
17 "ORACLE_PRODUCT_INFO_DRUGSTORE"."PRODUCT_ID" "PRODUCT_ID",
18 "ORACLE_PRODUCT_INFO_DRUGSTORE"."PRODUCT_NAME" "PRODUCT_NAME",
19 "ORACLE_PRODUCT_INFO_DRUGSTORE"."PRODUCT_STATUS" "PRODUCT_STATUS",
20 "ORACLE_PRODUCT_INFO_DRUGSTORE"."LIST_PRICE" "LIST_PRICE",
21 "ORACLE_PRODUCT_INFO_DRUGSTORE"."COST" "COST",
22 "ORACLE_PRODUCT_INFO_DRUGSTORE"."LAST_UPDATED" "LAST_UPDATED"
23 FROM
24 "ORACLE_PRODUCT_INFO"@DRUGSTORE "ORACLE_PRODUCT_INFO_DRUGSTORE"
```

Because sequences can be used by multiple sessions, you cannot depend on the numbers being consecutive. A sequence also caches a specific number of values, for example a range of 20 values, that are lost when you terminate the session.

Set (UNION, UNION ALL, INTERSECT, MINUS)

Set operators combine the results of two component queries into a single result. Unlike a joiner, set operators do not require WHERE clauses to tie result sets together. In set based operators, although the data is added to one output, the column lists are not mixed together to form one combined column list. While a joiner combines separate rows into one row, set operators combine all data rows in their universal row as shown in Figure 2–16.

Figure 2–16 Applying a Set-Based Transformation



Warehouse Builder supports UNION, UNION ALL, INTERSECT, and MINUS as modes for this operator. Table 2–2 shows the returns for the operator for all the modes. For details, see the *Oracle SQL Reference*.

Table 2–2 Set Operator Returns

Operator	Returns
UNION	All rows selected by either query
UNION ALL	All rows selected by either query, including all duplicates
INTERSECT	All distinct rows selected by both queries
MINUS	All distinct rows selected by the first query but not the second

The UNION (or UNION ALL) operator is most commonly used to combine, for example, product or customer lists from disparate sources. If the tables (often staging tables) match in format, a UNION can combine the records to one unified list of products. These than can be loaded into a warehouse or cleansed before storage in the warehouse. Figure 2–17 shows an example of a union on two product tables.

Figure 2–17 Performing a Union on Two Product Tables

```

Code Viewer: TRANSFORMATIONS [OUTGRP1] [1 error(s), 6 warning(s)]
Code Edit Search View
1 SELECT
2 "ORDER_ID" "ORDER_ID",
3 "ORDER_DATE" "ORDER_DATE",
4 "ORDER_STATUS" "ORDER_STATUS",
5 "ORDER_TOTAL" "ORDER_TOTAL",
6 "SHIPPING_COST" "SHIPPING_COST"
7 FROM
8 (SELECT
9 /*+ NO_MERGE */
10 "ORACLE_ORDERS_DRUGSTORE"."ORDER_ID" "ORDER_ID",
11 "ORACLE_ORDERS_DRUGSTORE"."ORDER_DATE" "ORDER_DATE",
12 "ORACLE_ORDERS_DRUGSTORE"."ORDER_STATUS" "ORDER_STATUS",
13 "ORACLE_ORDERS_DRUGSTORE"."ORDER_TOTAL" "ORDER_TOTAL",
14 "ORACLE_ORDERS_DRUGSTORE"."SHIPPING_COST" "SHIPPING_COST"
15 FROM
16 "ORACLE_ORDERS"@ "DRUGSTORE" "ORACLE_ORDERS_DRUGSTORE"
17 UNION
18 SELECT
19 "ORACLE_ORDER_ITEMS_DRUGSTORE"."PRODUCT_ID" "ORDER_ID",
20 "ORACLE_ORDER_ITEMS_DRUGSTORE"."UNIT_PRICE" "ORDER_DATE",
21 "ORACLE_ORDER_ITEMS_DRUGSTORE"."QUANTITY" "ORDER_STATUS"
22 FROM
23 "ORACLE_ORDER_ITEMS"@ "DRUGSTORE" "ORACLE_ORDER_ITEMS_DRUGSTORE")
Line 1 Column 1 | Read Only | [PL/SQL] Set based/Outgoing | Windows: CR/LF

```

If you change the mode on the operator (each operator can only perform one action), the generated code changes and the UNION keyword is substituted with the one you have chosen.

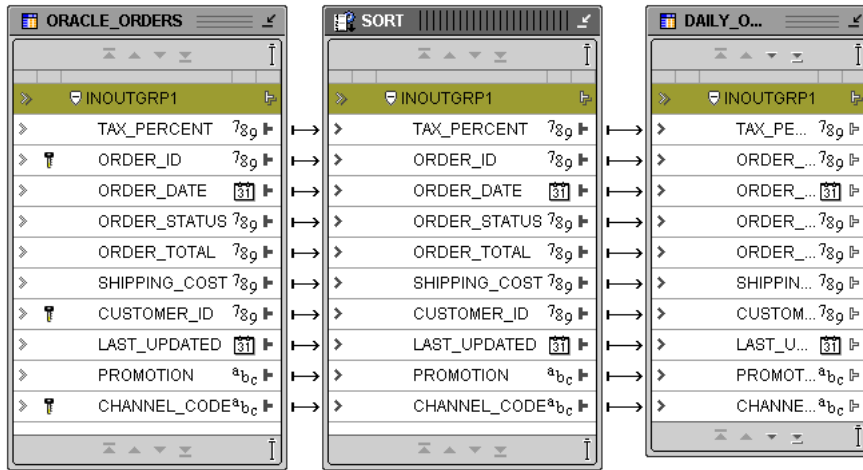
Sorter (ORDER BY)

Because most data in a data warehouse is typically loaded in batches, there can be problems in the loading routines. For example, a batch of orders might contain a single order number multiple times with each order line representing a different state of the order. The order might have gone from status "CREATED" to "UPDATED" to "BOOKED" during the day.

Because a SQL select statement does not guarantee any ordering by default, the inserts and updates on the target table can take place in the wrong order. If the "UPDATED" row is processed last, it becomes the final value for the day although the result should be status "BOOKED",

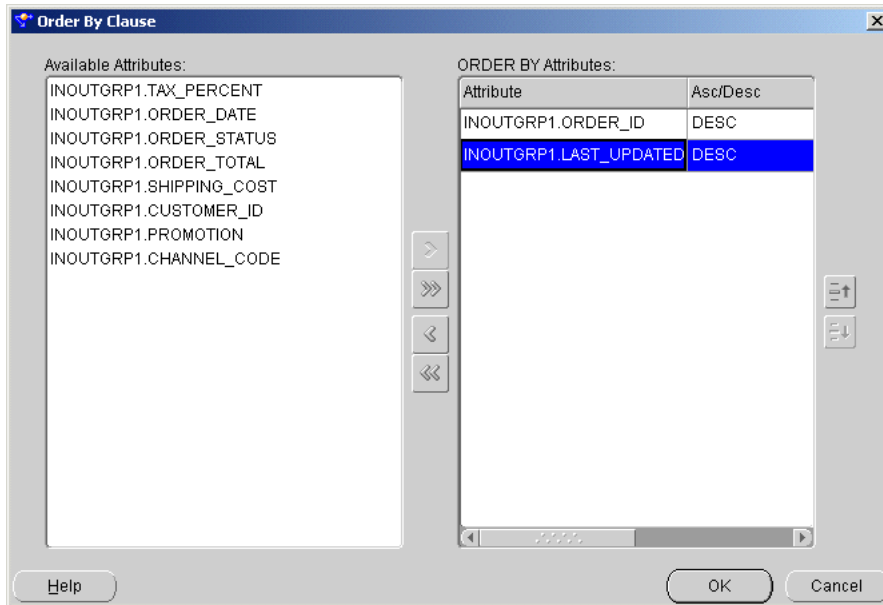
Warehouse Builder enables you to solve this problem by creating an ordered extraction query using the Sorter operator.

Figure 2–18 Ordering Data in a Mapping

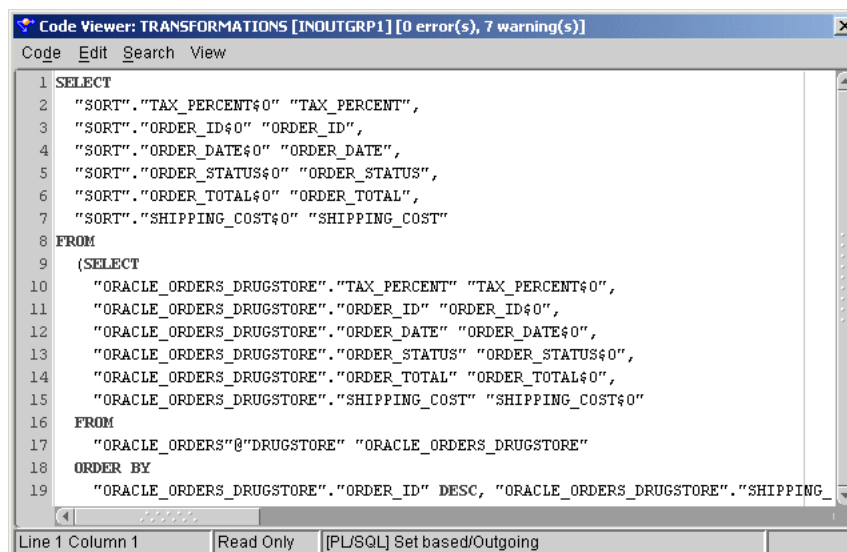


The sorter creates an `ORDER BY` clause in the SQL statement allowing ordering on the columns in the `ORDER BY` clause. Columns listed first in the `ORDER BY` clause take precedent over the ones listed later in the list. The first ordering is done on the order number, and within each group of order numbers, the ordering is done on the last updated date, as shown in Figure 2–18. The last change is the last update on the target reflecting the correct final status of the order in the target.

Figure 2–19 Determining Sorting and Sort Order



Ordering within the `GROUP BY` clauses can be done either in an `ASCENDING` or `DESCENDING` order. The default is `DESCENDING` order, as shown in Figure 2–19.

Figure 2–20 ORDER BY Clause in the Generated SQL


```

Code Viewer: TRANSFORMATIONS [INOUTGRP1] [0 error(s), 7 warning(s)]
Code Edit Search View
1 SELECT
2 "SORT"."TAX_PERCENT&0" "TAX_PERCENT",
3 "SORT"."ORDER_ID&0" "ORDER_ID",
4 "SORT"."ORDER_DATE&0" "ORDER_DATE",
5 "SORT"."ORDER_STATUS&0" "ORDER_STATUS",
6 "SORT"."ORDER_TOTAL&0" "ORDER_TOTAL",
7 "SORT"."SHIPPING_COST&0" "SHIPPING_COST"
8 FROM
9 (SELECT
10 "ORACLE_ORDERS_DRUGSTORE"."TAX_PERCENT" "TAX_PERCENT&0",
11 "ORACLE_ORDERS_DRUGSTORE"."ORDER_ID" "ORDER_ID&0",
12 "ORACLE_ORDERS_DRUGSTORE"."ORDER_DATE" "ORDER_DATE&0",
13 "ORACLE_ORDERS_DRUGSTORE"."ORDER_STATUS" "ORDER_STATUS&0",
14 "ORACLE_ORDERS_DRUGSTORE"."ORDER_TOTAL" "ORDER_TOTAL&0",
15 "ORACLE_ORDERS_DRUGSTORE"."SHIPPING_COST" "SHIPPING_COST&0"
16 FROM
17 "ORACLE_ORDERS"@ "DRUGSTORE" "ORACLE_ORDERS_DRUGSTORE"
18 ORDER BY
19 "ORACLE_ORDERS_DRUGSTORE"."ORDER_ID" DESC, "ORACLE_ORDERS_DRUGSTORE"."SHIPPING
Line 1 Column 1 Read Only [PL/SQL] Set based/Outgoing

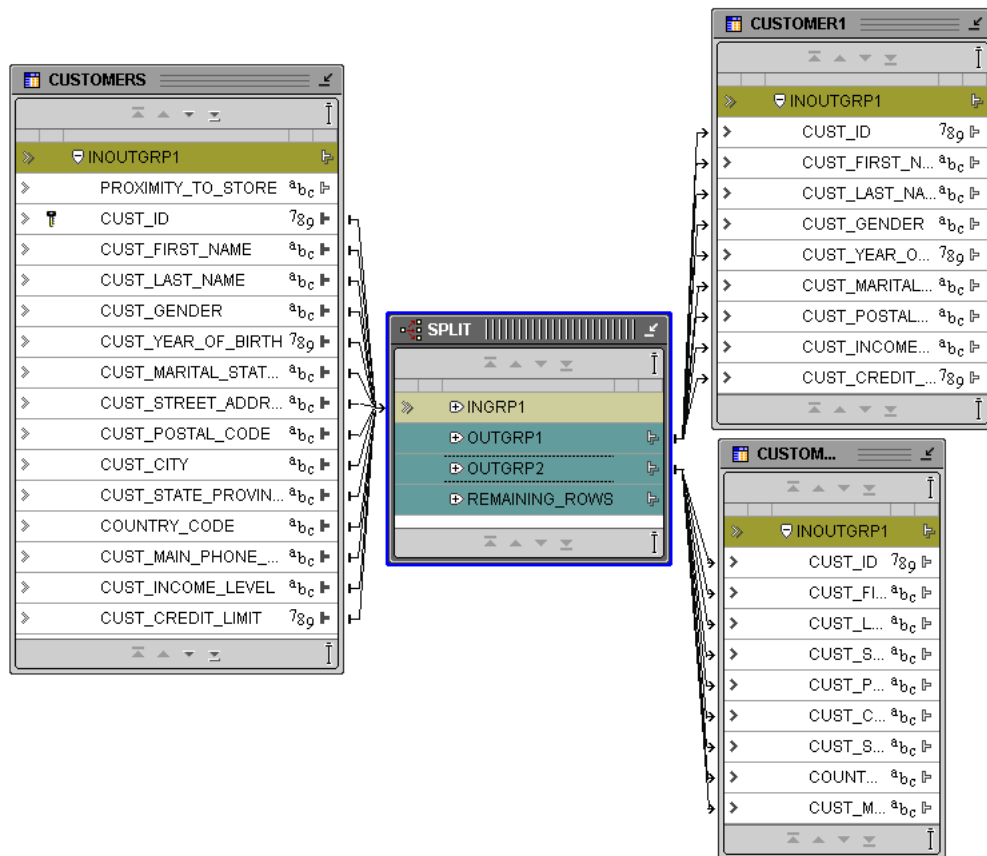
```

If you change the order of the attributes in the `ORDER BY` clause, it changes the order in the generated SQL and also the behavior of the mapping, as shown in [Figure 2–20](#).

Splitter (Multiple Table WHERE)

In a warehouse environment, you may require data to be moved to different targets based on a data driven condition. Instead of moving the data through multiple filters, Warehouse Builder enables you to use a splitter. This operator takes input data and outputs multiple flows of data based on the split conditions you specify.

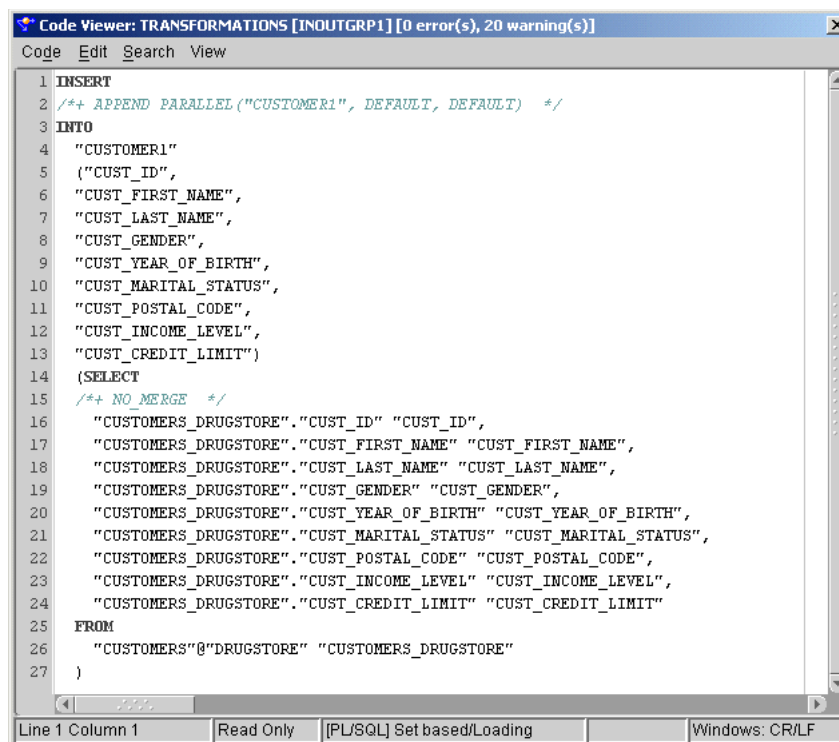
For example, if you want to split the `CUSTOMER` table into addresses and pure customer information, then one row must be inserted in two tables as shown in [Figure 2–21](#).

Figure 2–21 Performing an Unconditional Split

In this example, no split conditions are added to the splitter. Although OUTGRP1 and OUTGRP2 have no condition, a set of columns in OUTGRP1 are mapped to one target while a set of columns in OUTGRP2 are mapped to a different target. If you want to reduce the number of customers (based on the assumption that one customer has more addresses), you can add a deduplicator to the upper flow to obtain only one customer for each address. By mapping the `cust_id` to both targets, a relationship is maintained at all times.

Currently, the code generated is two separate streams of data. Each target is treated as a data recipient. If the data is inserted, two insert statements are generated in one package, as shown in [Figure 2–22](#) and [Figure 2–23](#).

Figure 2–22 Inserting the Customer Table

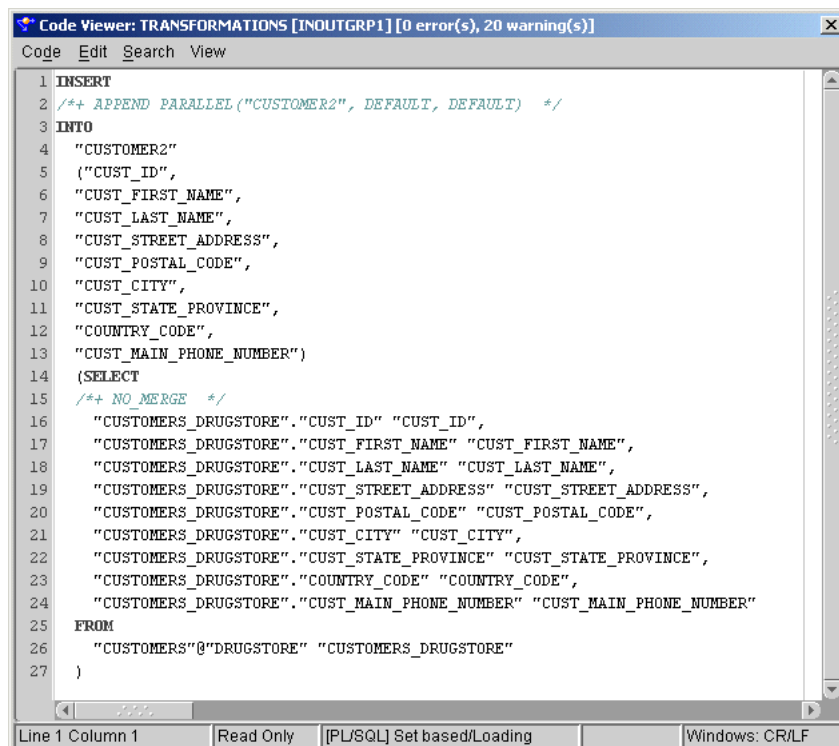


```

Code Viewer: TRANSFORMATIONS [INOUTGRP1] [0 error(s), 20 warning(s)]
Code Edit Search View
1 INSERT
2 /*+ APPEND PARALLEL("CUSTOMER1", DEFAULT, DEFAULT) */
3 INTO
4 "CUSTOMER1"
5 ("CUST_ID",
6 "CUST_FIRST_NAME",
7 "CUST_LAST_NAME",
8 "CUST_GENDER",
9 "CUST_YEAR_OF_BIRTH",
10 "CUST_MARITAL_STATUS",
11 "CUST_POSTAL_CODE",
12 "CUST_INCOME_LEVEL",
13 "CUST_CREDIT_LIMIT")
14 (SELECT
15 /*+ NO_MERGE */
16 "CUSTOMERS_DRUGSTORE"."CUST_ID" "CUST_ID",
17 "CUSTOMERS_DRUGSTORE"."CUST_FIRST_NAME" "CUST_FIRST_NAME",
18 "CUSTOMERS_DRUGSTORE"."CUST_LAST_NAME" "CUST_LAST_NAME",
19 "CUSTOMERS_DRUGSTORE"."CUST_GENDER" "CUST_GENDER",
20 "CUSTOMERS_DRUGSTORE"."CUST_YEAR_OF_BIRTH" "CUST_YEAR_OF_BIRTH",
21 "CUSTOMERS_DRUGSTORE"."CUST_MARITAL_STATUS" "CUST_MARITAL_STATUS",
22 "CUSTOMERS_DRUGSTORE"."CUST_POSTAL_CODE" "CUST_POSTAL_CODE",
23 "CUSTOMERS_DRUGSTORE"."CUST_INCOME_LEVEL" "CUST_INCOME_LEVEL",
24 "CUSTOMERS_DRUGSTORE"."CUST_CREDIT_LIMIT" "CUST_CREDIT_LIMIT"
25 FROM
26 "CUSTOMERS"@DRUGSTORE "CUSTOMERS_DRUGSTORE"
27 )
Line 1 Column 1 Read Only [PL/SQL] Set based/Loading Windows: CR/LF

```

Figure 2–23 Inserting the Address Data



```

Code Viewer: TRANSFORMATIONS [INOUTGRP1] [0 error(s), 20 warning(s)]
Code Edit Search View
1 INSERT
2 /*+ APPEND PARALLEL("CUSTOMER2", DEFAULT, DEFAULT) */
3 INTO
4 "CUSTOMER2"
5 ("CUST_ID",
6 "CUST_FIRST_NAME",
7 "CUST_LAST_NAME",
8 "CUST_STREET_ADDRESS",
9 "CUST_POSTAL_CODE",
10 "CUST_CITY",
11 "CUST_STATE_PROVINCE",
12 "COUNTRY_CODE",
13 "CUST_MAIN_PHONE_NUMBER")
14 (SELECT
15 /*+ NO_MERGE */
16 "CUSTOMERS_DRUGSTORE"."CUST_ID" "CUST_ID",
17 "CUSTOMERS_DRUGSTORE"."CUST_FIRST_NAME" "CUST_FIRST_NAME",
18 "CUSTOMERS_DRUGSTORE"."CUST_LAST_NAME" "CUST_LAST_NAME",
19 "CUSTOMERS_DRUGSTORE"."CUST_STREET_ADDRESS" "CUST_STREET_ADDRESS",
20 "CUSTOMERS_DRUGSTORE"."CUST_POSTAL_CODE" "CUST_POSTAL_CODE",
21 "CUSTOMERS_DRUGSTORE"."CUST_CITY" "CUST_CITY",
22 "CUSTOMERS_DRUGSTORE"."CUST_STATE_PROVINCE" "CUST_STATE_PROVINCE",
23 "CUSTOMERS_DRUGSTORE"."COUNTRY_CODE" "COUNTRY_CODE",
24 "CUSTOMERS_DRUGSTORE"."CUST_MAIN_PHONE_NUMBER" "CUST_MAIN_PHONE_NUMBER"
25 FROM
26 "CUSTOMERS"@DRUGSTORE "CUSTOMERS_DRUGSTORE"
27 )
Line 1 Column 1 Read Only [PL/SQL] Set based/Loading Windows: CR/LF

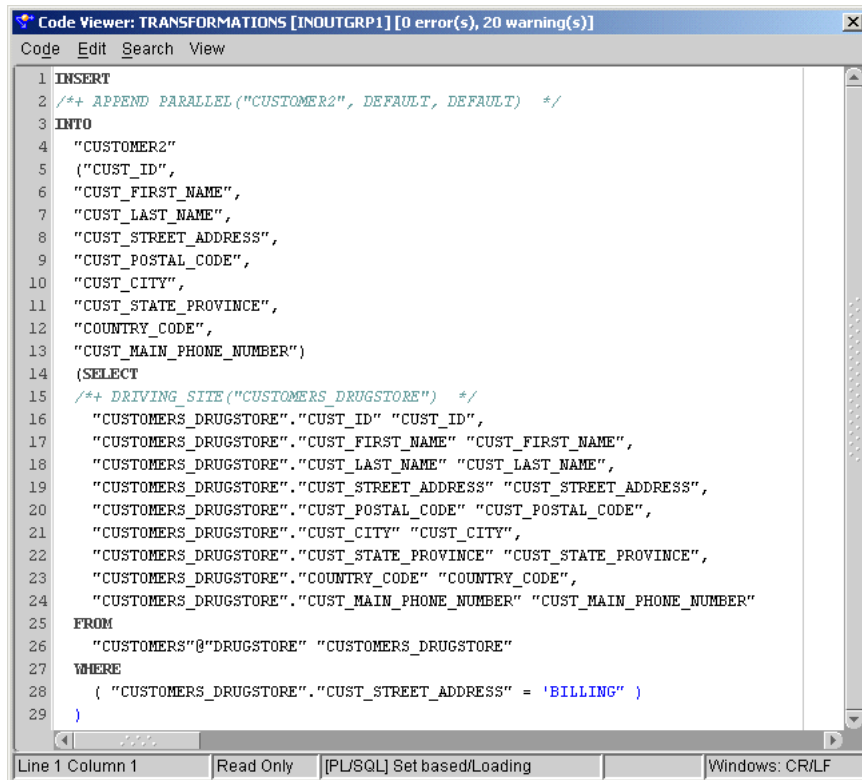
```

In the following example, the addresses may be split to hold only the billing addresses in the address table. You can add a condition to OUTGRP2 to select only these

addresses. In such a case, a WHERE clause is added to the code, as shown in [Figure 2–24](#).

If none of the clauses is met on the output groups, the data is added to the default group containing all data not held in any of the regular output groups.

Figure 2–24 Inserting Only “Bill To” Addresses



```

Code Viewer: TRANSFORMATIONS [INOUTGRP1] [0 error(s), 20 warning(s)]
Code Edit Search View
1 INSERT
2 /*+ APPEND PARALLEL ("CUSTOMER2", DEFAULT, DEFAULT) */
3 INTO
4 "CUSTOMER2"
5 ("CUST_ID",
6 "CUST_FIRST_NAME",
7 "CUST_LAST_NAME",
8 "CUST_STREET_ADDRESS",
9 "CUST_POSTAL_CODE",
10 "CUST_CITY",
11 "CUST_STATE_PROVINCE",
12 "COUNTRY_CODE",
13 "CUST_MAIN_PHONE_NUMBER")
14 (SELECT
15 /*+ DRIVING_SITE ("CUSTOMERS_DRUGSTORE") */
16 "CUSTOMERS_DRUGSTORE"."CUST_ID" "CUST_ID",
17 "CUSTOMERS_DRUGSTORE"."CUST_FIRST_NAME" "CUST_FIRST_NAME",
18 "CUSTOMERS_DRUGSTORE"."CUST_LAST_NAME" "CUST_LAST_NAME",
19 "CUSTOMERS_DRUGSTORE"."CUST_STREET_ADDRESS" "CUST_STREET_ADDRESS",
20 "CUSTOMERS_DRUGSTORE"."CUST_POSTAL_CODE" "CUST_POSTAL_CODE",
21 "CUSTOMERS_DRUGSTORE"."CUST_CITY" "CUST_CITY",
22 "CUSTOMERS_DRUGSTORE"."CUST_STATE_PROVINCE" "CUST_STATE_PROVINCE",
23 "CUSTOMERS_DRUGSTORE"."COUNTRY_CODE" "COUNTRY_CODE",
24 "CUSTOMERS_DRUGSTORE"."CUST_MAIN_PHONE_NUMBER" "CUST_MAIN_PHONE_NUMBER"
25 FROM
26 "CUSTOMERS"@ "DRUGSTORE" "CUSTOMERS_DRUGSTORE"
27 WHERE
28 ( "CUSTOMERS_DRUGSTORE"."CUST_STREET_ADDRESS" = 'BILLING' )
29 )
Line 1 Column 1 Read Only [PL/SQL] Set based/Loading Windows: CR/LF

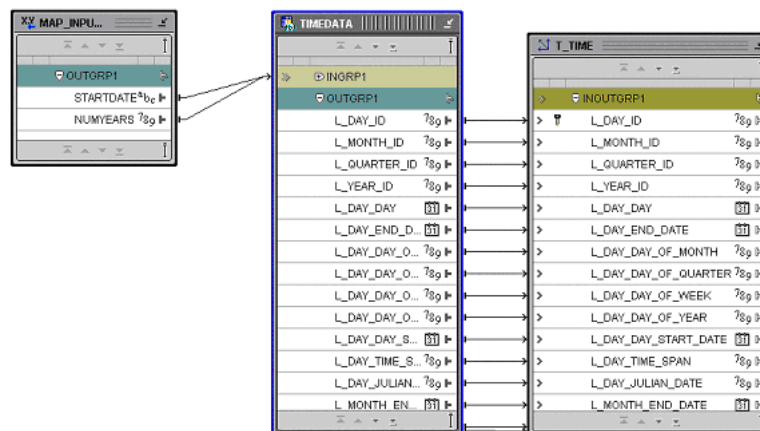
```

You can also use the splitter for conditional filtering. For example, you can use it split erroneous data from the main branch into separate error tables.

Table Function

While a regular function only works on one row at a time, a table function enables you to apply the same complex PL/SQL logic on a set of rows and increase your performance.

In Warehouse Builder, you can add a table function operator to a mapping and input a set of rows into it. This row set is then transformed using PL/SQL logic within the table function before it is output to the next operator.

Figure 2–25 Table Function in a Mapping

In [Figure 2–25](#), the Time dimension is loaded from a table function. This table function is added to the FROM clause of the select statement. The table function plays the role of a row set provider allowing complex calendar data generation and loading to be done in a single "insert as select" statement.

Unpivot Operator

The unpivot operator converts multiple input rows into one output row. It also enables you to extract once from a source, and then produce one row from a set of source rows that are grouped by attributes in the source data. Like the pivot operator, you can place the unpivot operator anywhere in a mapping.

Example: Unpivoting Sales Data

[Table 2–3](#) shows a representative sample of data from a relational table, SALES. In the crosstab format, the 'MONTH' column has 12 possible character values, one for each month of the year. And all sales figures are contained in one column, 'MONTHLY_SALES'.

Table 2–3 Data in Crosstab Format

REP	MONTH	MONTHLY_SALES	REGION
0675	Jan	10.5	4
0676	Jan	9.5	3
0679	Jan	8.7	3
0675	Feb	11.4	4
0676	Feb	10.5	3
0679	Feb	7.4	3
0675	Mar	9.5	4
0676	Mar	10.3	3
0679	Mar	7.5	3
0675	Apr	8.7	4
0676	Apr	7.6	3
0679	Apr	7.8	3

Figure 2–26 shows data from the relational table ‘SALES’ after Warehouse Builder unpivots the table. The data formerly contained in the ‘MONTH’ column, for example Jan, Feb, Mar, corresponds to 12 separate attributes (M1, M2, M3...). The sales figures formerly contained in the ‘MONTHLY_SALES’ are now distributed across the 12 attributes for each month.

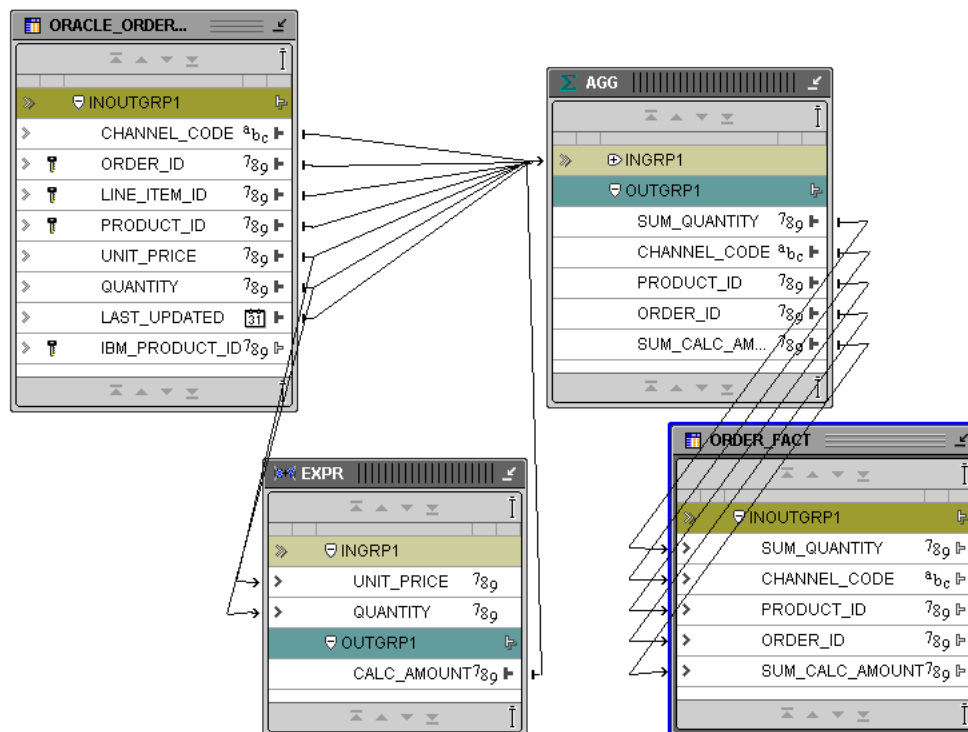
Figure 2–26 Data Unpivoted from Crosstab Format

ID	Reg	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12
0675	4	10.5	11.4	9.5	8.7	7.4	7.5	7.8	9.7				
0676	3	9.5	10.5	10.3	7.6	8.0	7.8	8.7	8.9				
0679	3	8.7	7.4	7.5	7.8	9.7	10.3	7.6	8.0				
0683	2	9.5	10.5	10.3	9.5	8.7	7.4	7.8	8.7				
0684	1	11.4	9.5	8.7	7.4	7.5	10.3	9.5	8.7				
0687	1	9.5	8.7	7.4	7.8	8.7	7.4	7.5	7.8				
0690	1	8.7	7.4	7.8	8.7	11.4	9.5	8.7	7.4				

Aggregator (GROUP BY, HAVING)

Aggregation of fact data is a common transformation operation. In Warehouse Builder, you can add one aggregator to a mapping to perform multiple aggregations. Warehouse Builder provides a separate editor to enable you to create complex aggregations. Although you can call a different aggregation function for each attribute in an aggregator, each aggregator supports only one GROUP BY and one HAVING clause. For example, you may want to aggregate orders over the Channel, Product, and Orders dimension, as shown in Figure 2–27.

Figure 2–27 Aggregating Order Information



If the target table in this example is allowed to take inserts and updates (updates are matched on the dimension key values or the aggregation points), then the following query is generated by Warehouse Builder, as shown in Figure 2–28.

Figure 2–28 Merging Aggregated Data

```

1 MERGE
2 /*+ APPEND PARALLEL("ORDER_FACT", DEFAULT, DEFAULT) */
3 INTO
4 "ORDER_FACT"
5 USING
6 (SELECT
7 /*+ DRIVING_SITE("AGG"."ORACLE_ORDER_ITEMS_DRUGSTORE") */
8 "AGG"."SUM_QUANTITY&0" "SUM_QUANTITY",
9 "AGG"."CHANNEL_CODE&0" "CHANNEL_CODE",
10 "AGG"."PRODUCT_ID&0" "PRODUCT_ID",
11 "AGG"."ORDER_ID&0" "ORDER_ID",
12 "AGG"."SUM_CALC_AMOUNT&0" "SUM_CALC_AMOUNT"
13 FROM
14 (SELECT
15 SUM("ORACLE_ORDER_ITEMS_DRUGSTORE"."QUANTITY") "SUM_QUANTITY&0",
16 "ORACLE_ORDER_ITEMS_DRUGSTORE"."CHANNEL_CODE" "CHANNEL_CODE&0",
17 "ORACLE_ORDER_ITEMS_DRUGSTORE"."PRODUCT_ID" "PRODUCT_ID&0",
18 "ORACLE_ORDER_ITEMS_DRUGSTORE"."ORDER_ID" "ORDER_ID&0",
19 SUM(("ORACLE_ORDER_ITEMS_DRUGSTORE"."UNIT_PRICE" * "ORACLE_ORDER
20 FROM
21 "ORACLE_ORDER_ITEMS"@ "DRUGSTORE" "ORACLE_ORDER_ITEMS_DRUGSTORE"
22 GROUP BY
23 "ORACLE_ORDER_ITEMS_DRUGSTORE"."CHANNEL_CODE", "ORACLE_ORDER_ITEMS
24 ) "MERGEQUERY_937"
25 ON (
26 )
27 WHEN NOT MATCHED THEN
28 INSERT

```

The statement can be created using the properties on the aggregator. Each attribute holds its own aggregation type, and the HAVING and GROUP BY clauses are modified on the operator, as shown in [Figure 2–29](#), [Figure 2–30](#), and [Figure 2–31](#).

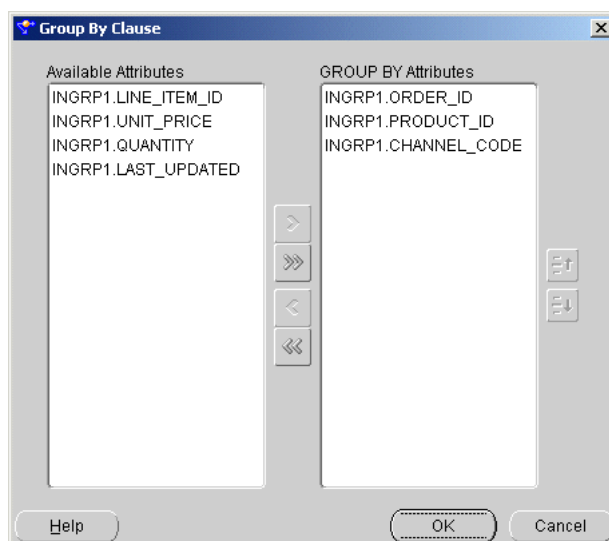
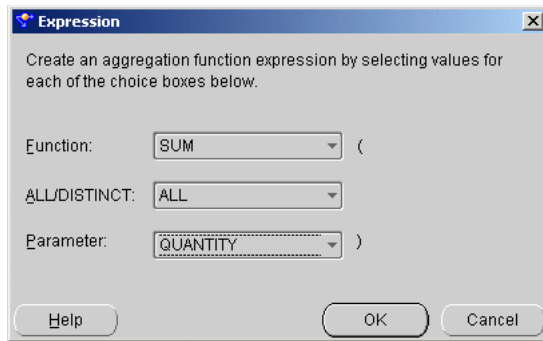
Figure 2–29 GROUP BY Clause

Figure 2–30 Aggregation Function Per Attribute

The Aggregator can use the following functions:

AVG

Syntax

```
avg ::= AVG (expr)
```

Purpose

AVG returns the average value of `expr` using the `GROUP BY` clause as specified on the operator. In Warehouse Builder, this means that the aggregator returns the average value of the data flowing into the operator as an output group attribute.

Example

The following example calculates the average salary of all employees in the `OE.EMPLOYEES` table:

```
SELECT AVG(salary) "Average" FROM employees;
```

```
Average
-----
6425
```

COUNT

Syntax

```
count ::= COUNT (expr)
```

Purpose

COUNT returns the number of rows in the query using the `GROUP BY` clause as specified on the operator. If you specify an `expr`, COUNT returns the number of rows where `expr` is not null. You can count all rows or only distinct values of `expr`. If you specify the asterisk (*), this function returns all rows, including duplicates and nulls. COUNT never returns null as a value on its own.

Example

The following example uses COUNT as an aggregate function:

```
SELECT COUNT(commission_pct) "Count" FROM employees;
```

```
Count
-----
```

35

MAX

Syntax

```
max ::= MAX(attribute)
```

Purpose

MAX returns the maximum value of `attribute` using the `GROUP BY` clause as specified on the operator. This means that the aggregator returns the maximum value of the data flowing into the operator `attribute` as an output group attribute.

Example

The following example determines the highest salary in the `HR.EMPLOYEES` table:

```
SELECT MAX(salary) "Maximum" FROM employees;
```

```
Maximum
-----
24000
```

MIN

Syntax

```
min ::= MIN(attribute)
```

Purpose

MIN returns the maximum value of `attribute` using the `GROUP BY` clause as specified on the operator. This means that the aggregator returns the maximum value of the data flowing into the operator `attribute` as an output group attribute.

Example

The following statement returns the earliest hiredate in the `HR.EMPLOYEES` table:

```
SELECT MIN(hire_date) "Earliest" FROM employees;
```

```
Earliest
-----
17-JUN-87
```

NONE

Syntax

```
none ::= Group By (attribute)
```

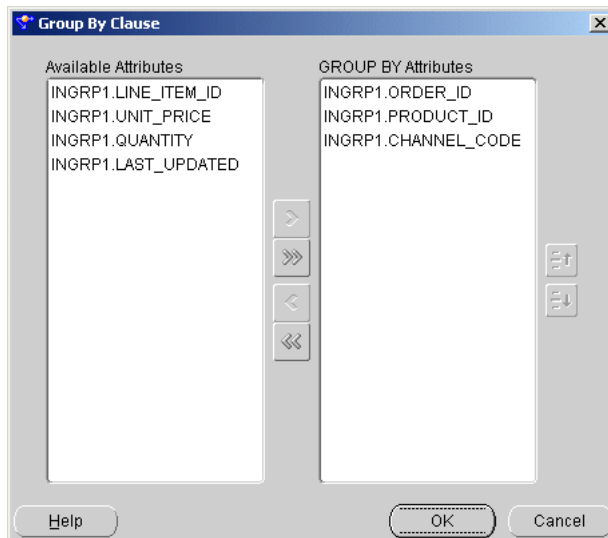
Purpose

NONE is used to identify the action used to aggregate on the attribute when this attribute is added to the `GROUP BY` clause. Specifying NONE in the aggregation operator for `attribute` automatically adds it to the `Group By` clause (and vice-versa). Using NONE does not lead to an aggregation in the SQL statement as the other functions do.

Example

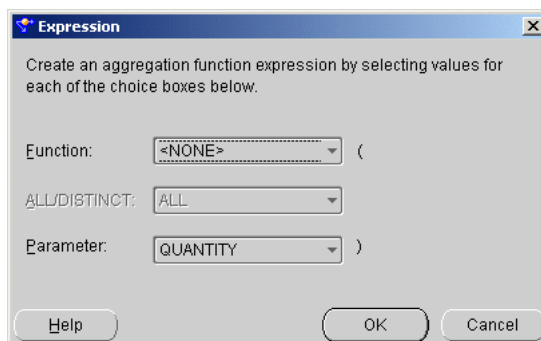
Figure 2–31 shows the GROUP BY attributes on the right side. Moving an attribute from the left side to the right side automatically switches the aggregation action to NONE.

Figure 2–31 Group By Clause Dialog



Conversely (as with the aggregation function dialog), selecting NONE moves the attribute to the GROUP BY clause and it appears on the right side.

Figure 2–32 Selecting No Aggregation for the Attribute



STDDEV

Syntax

```
stddev ::= STDDEV(attribute)
```

Purpose

STDDEV returns sample standard deviation of *attribute*: a set of numbers. The attributes in Warehouse Builder typically consist of a row fed into the aggregator.

STDDEV differs from STDDEV_SAMP because STDDEV returns zero when it has only one row of input data, and STDDEV_SAMP returns a null. Oracle calculates the standard deviation as the square root of the variance defined for the VARIANCE aggregate function.

Example

The following example returns the standard deviation of salary values in the sample HR.EMPLOYEES table:

```
SELECT STDDEV(salary) "Deviation"
FROM employees;
```

```
Deviation
-----
3909.36575
```

STDDEV_POP**Syntax**

```
stddev_pop::=STDDEV_POP(sttribute)
```

Purpose

STDDEV_POP computes the population standard deviation and returns the square root of the population variance into the output attribute of the aggregator.

The attribute is a number expression, and the function returns a value of type NUMBER.

This function is the same as the square root of the VAR_POP function. When VAR_POP returns null, this function also returns null.

Example

The following example returns the population and sample standard deviations of amount of sales in the sample table SH.SALES.

```
SELECT STDDEV_POP(amount_sold) "Pop",
       STDDEV_SAMP(amount_sold) "Samp"
FROM sales;
```

```
Pop          Samp
-----
944.290101   944.290566
```

STDDEV_SAMP**Syntax**

```
stddev_samp::=STDDEV_SAMP(attribute)
```

Purpose

STDDEV_SAMP computes the cumulative sample standard deviation and returns the square root of the sample variance into the output attribute of the aggregator.

The attribute is a number expression and the function returns a value of type NUMBER. This function is same as the square root of the VAR_SAMP function. When VAR_SAMP returns NULL, this function also returns NULL.

Example

The following example returns the population and sample standard deviations of amount of sales in the sample table SH.SALES.

```
SELECT STDDEV_POP(amount_sold) "Pop",
```

```
STDDEV_SAMP(amount_sold) "Samp"
FROM sales;
```

```
Pop          Samp
-----
944.290101  944.290566
```

SUM

Syntax

```
sum ::= SUM(attribute)
```

Purpose

SUM returns the summary of the values of *attribute*. The *attribute* in Warehouse Builder is typically a row set fed into the aggregator. It can contain expressions from previous transformations or from the source system.

Example

The following example calculates the sum of all salaries in the sample HR.EMPLOYEES table below:

```
SELECT SUM(salary) "Total"
FROM employees;
```

```
Total
-----
691400
```

VAR_POP

Syntax

```
var_pop ::= VAR_POP(attribute)
```

Purpose

After discarding the nulls in this set, VAR_POP returns the population variance of a set of numbers to the output attribute in the aggregator. The *attribute* is a number expression and the function returns a value of type NUMBER. If the function is applied to an empty set, it returns NULL.

The function makes the following calculation:

$$(SUM(attribute^2) - SUM(attribute)^2 / COUNT(attribute)) / COUNT(attribute)$$

Example

The following example returns the population variance of the salaries in the HR.EMPLOYEES table:

```
SELECT VAR_POP(salary) FROM employees;
```

```
VAR_POP(SALARY)
-----
15140307.5
```


VAR_SAMP

Syntax

```
var_samp ::= VAR_SAMP(attribute)
```

Purpose

After discarding the nulls in this set, `VAR_SAMP` returns the sample variance of a set of numbers to the output attribute in the aggregator. The expression is a number expression and the function returns a value of type `NUMBER`. If the function is applied to an empty set, it returns `NULL`.

The function makes the following calculation:

$$\frac{(\text{SUM}(\text{attribute}^2) - \text{SUM}(\text{attribute})^2 / \text{COUNT}(\text{attribute}))}{(\text{COUNT}(\text{attribute}) - 1)}$$

This function is similar to `VARIANCE`, except that given an input set of one element, `VARIANCE` returns 0 and `VAR_SAMP` returns null.

Example

The following example returns the sample variance of the salaries in the sample `HR.EMPLOYEES` table.

```
SELECT VAR_SAMP(salary) FROM employees;
```

```
VAR_SAMP(SALARY)
-----
15283140.5
```

VARIANCE

Syntax

```
variance ::= VARIANCE(attribute)
```

Purpose

`VARIANCE` returns the variance of `attribute` and delivers the result to the output attribute in the aggregator. Warehouse Builder calculates the variance of `attribute` as follows:

- 0 if the number of rows in `attribute` = 1
- `VAR_SAMP` if the number of rows in `attribute` > 1

Example

The following example calculates the variance of all salaries in the sample `HR.EMPLOYEES` table:

```
SELECT VARIANCE(salary) "Variance"
FROM employees;
```

```
Variance
-----
15283140.5
```

Constant

Many transformations require constant values. Warehouse Builder provides some constants as direct functions. You can use the Expression Builder to create constants in Warehouse Builder. The constants delivered as special functions are described in the following sections.

SYSDATE

Syntax

```
sysdate::=SYSDATE
```

Purpose

SYSDATE returns the current date and time and requires no arguments. In distributed SQL statements, this function returns the date and time on your local database. You cannot use this function in the condition of a CHECK constraint.

Example

The following example returns the current date and time:

```
SELECT TO_CHAR (SYSDATE, 'MM-DD-YYYY HH24:MI:SS') "NOW"  
FROM DUAL;
```

```
NOW  
-----  
04-13-2001 09:45:51
```

SYSTIMESTAMP

Syntax

```
systimestamp::=SYSTIMESTAMP
```

Purpose

The SYSTIMESTAMP function returns the system date, including fractional seconds and time zone of the database. The return type is `TIMESTAMP WITH TIME ZONE`.

Example

The following example returns the system date:

```
SELECT SYSTIMESTAMP FROM DUAL;
```

```
SYSTIMESTAMP  
-----  
28-MAR-00 12.38.55.538741 PM -08:00
```

Data Cleansing Operators

The Warehouse Builder Mapping Editor includes operators that perform data cleansing transformations. This section describes these operators.

Name and Address

Warehouse Builder includes an operator that enables name and address cleansing (as of the 9.0.2.56.0 version). The name and address operator supports parsing,

standardization, postal matching, and geocoding of name and address data. Name and Address parsing is the breakdown of non-discrete input into discrete name or address components. For example, an input address of:

```
Mr. Joe A. Smith Sr.
8500 Normandale Lake Blvd Suite 710
Bloomington MN 55438
```

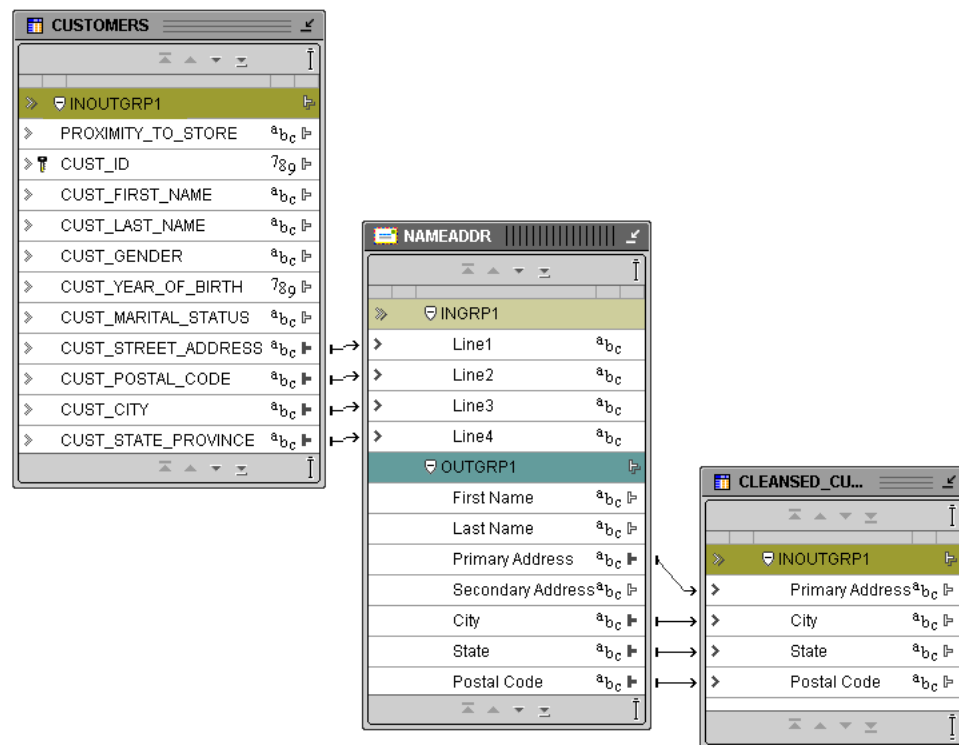
is parsed into the following abbreviated list of address components:

```
Pre name: MR
First name: JOE
First name standardized: JOSEPH
Post name: SR
Street name: NORMANDALE LAKE
Primary address: NORMANDALE LAKE BLVD
Secondary address: STE 710
Last Line: BLOOMINGTON MN 55437-3813
Latitude: 44.854876
```

Name and address standardization includes modification of components to a standard version acceptable to a postal service or suitable for record matching. In the preceding example, Suite is standardized to STE and Joe is standardized to JOSEPH.

Postal matching involves matching an input address with postal database entries to either verify an address or correct an address. In the preceding example, the postal code was corrected to 55437-3813.

Figure 2-33 Name and Address Operator



Geocoding (only available for the US) involves the collection of census and locational data. Latitude and Longitude are currently available in Warehouse Builder. Census

data such as minor census district, metropolitan statistical area, and FIPS code will be available in later versions.

Match-Merge Operator

The Match-Merge operator is a data quality operator that you can use to first match and then merge data.

When you match records, you determine through business rules which records in a table refer to the same data. When you merge records, you consolidate into a single record the data from the matched records.

This section includes information and examples on how to use the Match-Merge operator in a mapping. The Match-Merge operator together with the Name-Address operator support householding, the process of identifying unique households in name and address data.

Example: Matching and Merging Customer Data

Consider how you could utilize the Match-Merge operator to manage a customer mailing list. Use matching to find records that refer to the same person in a table of customer data containing 10,000 rows. For example, you can define a match rule that screens records that have similar first and last names. Through matching you may discover that 5 rows refer to the same person. You can merge those records into one new record. For example, you can create a merge rule to retain the values from the one of the five matched records with the longest address. The newly merged table now contains one record for each customer.

[Table 2–4](#) shows records that refer to the same person prior to using the Match-Merge operator.

Table 2–4 Sample Records

Row	FirstName	LastName	SSN	Address	Unit	Zip
1	Jane	Doe	NULL	123 Main Street	NULL	22222
2	Jane	Doe	111111111	NULL	NULL	22222
3	J.	Doe	NULL	123 Main Street	Apt 4	22222
4	NULL	Smith	111111111	123 Main Street	Apt 4	22222
5	Jane	Smith-Doe	111111111	NULL	NULL	22222

[Table 2–5](#) shows the single record for Jane Doe after using the Match-Merge operator. Notice that the new record retrieves data from different rows in the sample.

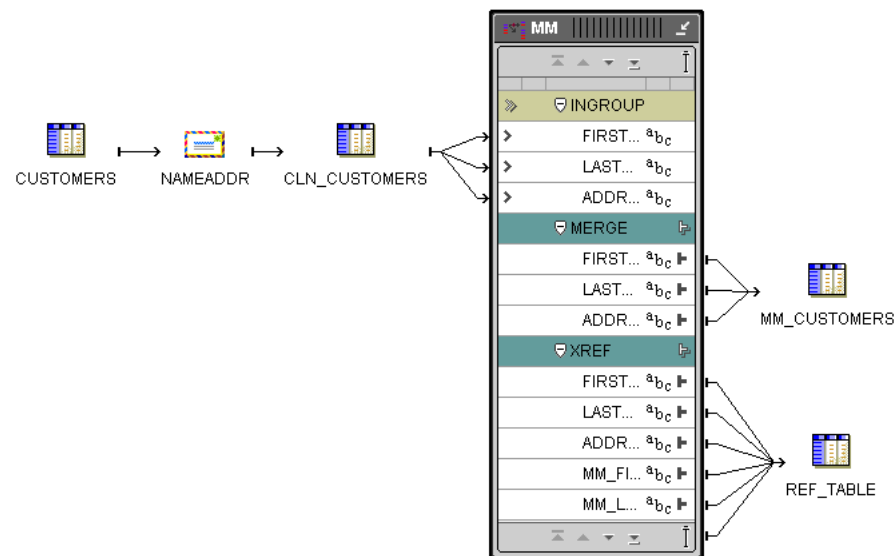
Table 2–5 Merged Record for Jane Doe

First Name	Last Name	SSN	Address	Unit	Zip
Jane	Doe	111111111	123 Main Street	Apt 4	22222

Designing Mappings with a Match-Merge Operator

[Figure 2–34](#) shows a mapping you can design using a Match-Merge operator. Notice that the Match-Merge operator is preceded by a Name-Address operator, NAMEADDR, and a staging table, CLN_CUSTOMERS. You can design your mapping with or without a Name-Address operator. Preceding the Match-Merge operator with a Name-Address operator is desirable when you want to ensure your data is clean and standardized before launching time consuming match and merge operations.

Figure 2–34 Match-Merge Operator in a Mapping



Whether you include a Name-Address operator or not, be aware of the following considerations as you design your mapping:

- **PL/SQL output:** The Match-Merge operator can generate two outputs, both PL/SQL outputs only. The MERGE group includes the merged data. The XREF group is an optional group you can design to document the merge process.
- **Row based operating mode:** When the Match-Merge operator matches records, it compares each row with the subsequent row in the source and generates row based code only. These mappings, therefore, can only run in row based mode.
- **SQL based operators before Match-Merge:** The Match-Merge operator generates only PL/SQL outputs. If you want to include operators that generate SQL code only, you must design the mapping such that they precede the Match-Merge operator. For example, operators such as the Join, Key Lookup, and Set operators must precede the Match-Merge operator. A mapping designed with operators that generate set based code after a Match-Merge operator is invalid and Warehouse Builder does not generate code for such mappings.
- **SQL input:** With one specific exception, the Match-Merge operator requires SQL input. If you want to precede a Match-Merge with an operator that generates only PL/SQL output such as the Name-Address operator, you must first load the data to a staging table.
- **Refining Data from Match-Merge operators:** To achieve greater data refinement, map the XREF output from one Match-Merge operator into another Match-Merge operator. This scenario is the one exception to the SQL input rule for Match-Merge operators. With additional design elements, the second Match-Merge operator accepts PL/SQL.

For more information on the Match Merge operator, see the *Oracle Warehouse Builder User's Guide*.

SQL Transformations

This chapter contains the following topics:

- [Introduction](#) on page 3-1
- [Administrative Transformations](#) on page 3-3
- [Character Transformations](#) on page 3-12
- [Date Transformations](#) on page 3-29
- [Number Transformations](#) on page 3-44
- [OLAP Transformations](#) on page 3-55
- [XML Transformations](#) on page 3-58
- [Conversion Transformations](#) on page 3-59
- [Other Transformations](#) on page 3-61

For related information, see:

- *Oracle SQL Reference*
- *Oracle Warehouse Builder User's Guide*

Introduction

The following sections describe the transformation libraries and introduce how to use custom transformations in Warehouse Builder.

About Transformations

Warehouse Builder supports the following transformation types:

- **User Transformation Package:** This category contains package functions and procedures that you define.
- **Predefined Transformations:** These categories exist in the Oracle Library and consist of built-in and seeded functions and procedures.
- **Functions:** The functions category is automatically created in every warehouse module. This category contains any standalone functions used as transformations. These functions can be defined by the user or imported from a database. A function transformation takes 0-*n* input parameters and produces a result value.
- **Procedures:** The procedures category is automatically created in every warehouse module. This category contains any standalone procedures used as transformations. These procedures can be defined by the user or imported from a

database. A procedure transformation takes 0-n input parameters and produces 0-n output parameters.

- **Imported Package:** This category is created by importing a PL/SQL package. Although you can modify the package body, you cannot modify the package header, which is the signature for the function or procedure. You can view the package in the transformation library property sheet.

About Oracle Transformation Libraries

Each time you create a warehouse module, Warehouse Builder creates a Transformation Library for that module containing transformation operations. This library contains the standard Oracle Library and an additional library for each warehouse module defined within the repository.

Transformation Libraries consist of the following types:

- **Global Shared Library:** a collection of reusable transformations categorized as functions and procedures defined within your repository.
- **Oracle Library:** a collection of predefined functions from which you can define procedures for your Global Shared Library.

When you create a custom transformation, add it to the Global Shared Library to share across warehouse modules. If the transformation is specific to one module, add it to the transformation library within that module.

Global Shared Library

The Global Shared Library stores transformations that are shared across a repository. The default categories are:

- **Functions:** This category stores standalone functions.
- **Procedures:** This category stores standalone procedures.

Oracle Library

The Oracle Library includes a set of standard transformations organized into categories including:

- Administration
- Character
- Conversion
- Date
- Numeric
- Other
- XML

Accessing Transformation Libraries

You can access the Transformation Libraries from the Expression Builder, the Add Transformation dialog, or the New Transformation Wizard. You can also access Transformation Libraries from the navigation tree in the Warehouse Builder console. Additionally, you can create your own transformation libraries to organize transformations according to your needs.

Importing PL/SQL Packages

Use the Import Wizard to import PL/SQL functions, procedures, and packages into a Warehouse Builder project.

When you use the imported PL/SQL:

- You can edit, save, and deploy the imported PL/SQL functions and procedures.
- You cannot edit imported PL/SQL packages.
- Wrapped PL/SQL objects are not readable.
- Imported packages can be viewed and modified in the category property sheet.
- You can edit the imported package body but not the imported package specification.

Use the transformation properties sheet to edit a transformation. Be sure to edit properties consistently. For example, if you change the name of a parameter, then you must change its name in the implementation code. You can view transformation properties from the Mapping Editor using the Operator Property sheet. These settings are read-only.

Administrative Transformations

Administration transformations, or functions, are actions that are regularly performed in ETL processes. The main focus of these transformations is in the DBA related areas or to improve performance. For example, it is common to disable constraints when loading tables and then to re-enable them after loading has completed. Warehouse Builder provides pre-built functionality for this purpose in the administration transformations.

The administration functions in Warehouse Builder are all custom functions and are listed in alphabetical order.

WB_ABORT

Syntax

```
WB_ABORT(p_code, p_message)
```

where *p_code* is the abort code, and must be between -20000 and -29999; and *p_message* is an abort message you specify.

Purpose

WB_ABORT enables you to abort the application from a Warehouse Builder component. You can run it from a post mapping process or as a transformation within a mapping.

Example

Use this administration function to abort an application. You can use this function in a post mapping process to abort deployment if there is an error in the mapping.

WB_ANALYZE_SCHEMA

Syntax

```
WB_ANALYZE_SCHEMA
```

No parameters are required for this function.

Purpose

After loading data into the warehouse, the statistics need to be refreshed to ensure optimal performance recommendations from the cost-based optimizer in the warehouse. `WB_ANALYZE_SCHEMA` calls `DBMS_DDL.ANALYZE_OBJECT` to analyze a schema to supply these statistics. It analyzes the entire schema, which may take some time depending on the number of tables and the number of rows in these tables.

Example

You can use this administration package to automatically run the analyze command on the schema you loaded. This can be done using the post mapping process of the last mapping in a dependency diagram. You can also deploy the procedure to the database schema and invoke the procedure from OEM using a SQL statement created in the client tool.

WB_ANALYZE_TABLE

Syntax

```
WB_ANALYZE_TABLE(p_name)
```

where *p_name* is the table on which the analyze command is executed.

Purpose

After loading data into the warehouse the statistics need to be refreshed to ensure optimal performance recommendations from the cost-based optimizer in the warehouse. `WB_ANALYZE_TABLE` calls `DBMS_DDL.ANALYZE_OBJECT` to analyze a specific table in the user schema and supply these statistics. It analyzes the table, which may take some time depending on the number of rows in this table.

Example

You can use this administration package to automatically run the analyze command on the table you have loaded. This can be done using a post mapping process of the mapping that loads data into the table.

WB_COMPILE_PLSQL

Syntax

```
WB_COMPILE_PLSQL(p_name, p_type)
```

where *p_name* is the name of the object that is to be compiled; *p_type* is the type of object to be compiled. The legal types are:

```
'PACKAGE'  
'PACKAGE BODY'  
'PROCEDURE'  
'FUNCTION'  
'TRIGGER'
```

Purpose

This program unit compiles a stored object in the database.

WB_DISABLE_ALL_CONSTRAINTS

Syntax

```
WB_DISABLE_ALL_CONSTRAINTS(p_name)
```

where *p_name* is the table name that determines which constraints are disabled.

Purpose

This program unit disables all constraints that are owned by the table as stated in the call to the program.

For faster loading of data sets, you can disable constraints on a table. The data is now loaded without validation. This is mainly done on relatively clean data sets.

Example

The following example shows the disabling of the constraints on the table OE.CUSTOMERS:

```
select constraint_name
,decode(constraint_type
      , 'C', 'Check'
      , 'P', 'Primary'
      ) Type
,      status
from user_constraints
where table_name = 'CUSTOMERS';
5 rows selected
```

CONSTRAINT_NAME	TYPE	STATUS
CUST_FNAME_NN	Check	ENABLED
CUST_LNAME_NN	Check	ENABLED
CUSTOMER_CREDIT_LIMIT_MAX	Check	ENABLED
CUSTOMER_ID_MIN	Check	ENABLED
CUSTOMERS_PK	Primary	ENABLED

Perform the following in Scalpels or Warehouse Builder to disable all constraints:

```
Execute WB_DISABLE_ALL_CONSTRAINTS('CUSTOMERS');
```

CONSTRAINT_NAME	TYPE	STATUS
CUST_FNAME_NN	Check	DISABLED
CUST_LNAME_NN	Check	DISABLED
CUSTOMER_CREDIT_LIMIT_MAX	Check	DISABLED
CUSTOMER_ID_MIN	Check	DISABLED
CUSTOMERS_PK	Primary	DISABLED

```
5 rows selected
```

Note: This statement uses a cascade option to allow dependencies to be broken by disabling the keys.

WB_DISABLE_ALL_TRIGGERS

Syntax

```
WB_DISABLE_ALL_TRIGGERS (p_name)
```

where *p_name* is the table name on which the triggers are disabled.

Purpose

This program unit disables all triggers owned by the table as stated in the call to the program. The owner of the table must be the current user (in variable USER). This action stops triggers and improves performance.

Example

The following example shows the disabling of all triggers on the table OE.OC_ORDERS:

Available triggers:

```
select trigger_name
,      status
from user_triggers
where table_name = 'OC_ORDERS';
```

TRIGGER_NAME	STATUS
ORDERS_TRG	ENABLED
ORDERS_ITEMS_TRG	ENABLED

Perform the following in Scalpels or Warehouse Builder to disable the specified constraint.

```
Execute WB_DISABLE_ALL_TRIGGERS ('OC_ORDERS');
```

TRIGGER_NAME	STATUS
ORDERS_TRG	DISABLED
ORDERS_ITEMS_TRG	DISABLED

WB_DISABLE_CONSTRAINT

Syntax

```
WB_DISABLE_CONSTRAINT(p_constraintname, p_tablename)
```

where *p_constraintname* is the constraint name to be disabled; *p_tablename* is the table name on which the specified constraint is disabled.

Purpose

This program unit disables the specified constraint that is owned by the table as stated in the call to the program. The user is the current user (in variable USER).

For faster loading of data sets, you can disable constraints on a table. The data is then loaded without validation. This reduces overhead and is mainly done on relatively clean data sets.

Example

The following example shows the disabling of the specified constraint on the table OE.CUSTOMERS:

```
select constraint_name
, decode(constraint_type
, 'C', 'Check'
, 'P', 'Primary'
) Type
, status
from user_constraints
where table_name = 'CUSTOMERS';
```

CONSTRAINT_NAME	TYPE	STATUS
-----	-----	-----
CUST_FNAME_NN	Check	ENABLED
CUST_LNAME_NN	Check	ENABLED
CUSTOMER_CREDIT_LIMIT_MAX	Check	ENABLED
CUSTOMER_ID_MIN	Check	ENABLED
CUSTOMERS_PK	Primary	ENABLED

5 rows selected

Perform the following in SQL*Plus or Warehouse Builder to disable the specified constraint.

```
Execute WB_DISABLE_CONSTRAINT('CUSTOMERS_PK', 'CUSTOMERS');
```

CONSTRAINT_NAME	TYPE	STATUS
-----	-----	-----
CUST_FNAME_NN	Check	ENABLED
CUST_LNAME_NN	Check	ENABLED
CUSTOMER_CREDIT_LIMIT_MAX	Check	ENABLED
CUSTOMER_ID_MIN	Check	ENABLED
CUSTOMERS_PK	Primary	ENABLED

5 rows selected

Note: This statement uses a cascade option to allow dependencies to be broken by disabling the keys.

WB_DISABLE_TRIGGER**Syntax**

```
WB_DISABLE_TRIGGER(p_name)
```

where *p_name* is the trigger name to be disabled.

Purpose

This program unit disables the specified trigger. The owner of the trigger must be the current user (in variable USER).

Example

The following example shows the disabling of a trigger on the table OE.OC_ORDERS:

```
select trigger_name, status
from user_triggers
where table_name = 'OC_ORDERS';
```

TRIGGER_NAME	STATUS
ORDERS_TRG	ENABLED
ORDERS_ITEMS_TRG	ENABLED

Perform the following in SQL*Plus or Warehouse Builder to disable the specified constraint.

```
Execute WB_DISABLE_TRIGGER ('ORDERS_TRG');
```

TRIGGER_NAME	STATUS
ORDERS_TRG	DISABLED
ORDERS_ITEMS_TRG	DISABLED

WB_ENABLE_ALL_CONSTRAINTS

Syntax

```
WB_ENABLE_ALL_CONSTRAINTS (p_name)
```

where *p_name* is the table name which determines which constraints are disabled.

Purpose

This program unit enables all constraints that are owned by the table as stated in the call to the program.

For faster loading of data sets, you can disable constraints on a table. After the data is loaded, you must enable these constraints again using this program unit.

Example

The following example shows the disabling of the constraints on the table OE.CUSTOMERS:

```
select constraint_name
, decode(constraint_type
, 'C', 'Check'
, 'P', 'Primary)
Type
, status
from user_constraints
where table_name = 'CUSTOMERS';
```

CONSTRAINT_NAME	TYPE	STATUS
CUST_FNAME_NN	Check	DISABLED
CUST_LNAME_NN	Check	DISABLED
CUSTOMER_CREDIT_LIMIT_MAX	Check	DISABLED
CUSTOMER_ID_MIN	Check	DISABLED
CUSTOMERS_PK	Primary	DISABLED

5 rows selected

Perform the following in SQL*Plus or Warehouse Builder to enable all constraints.

```
Execute WB_ENABLE_ALL_CONSTRAINTS('CUSTOMERS');
```

CONSTRAINT_NAME	TYPE	STATUS
-----	-----	-----
CUST_FNAME_NN	Check	ENABLED
CUST_LNAME_NN	Check	ENABLED
CUSTOMER_CREDIT_LIMIT_MAX	Check	ENABLED
CUSTOMER_ID_MIN	Check	ENABLED
CUSTOMERS_PK	Primary	ENABLED

```
5 rows selected
```

WB_ENABLE_ALL_TRIGGERS

Syntax

```
WB_ENABLE_ALL_TRIGGERS(p_name)
```

where *p_name* is the table name on which the triggers are enabled

Purpose

This program unit enables all triggers owned by the table as stated in the call to the program. The owner of the table must be the current user (in variable USER).

Example

The following example shows the enabling of all triggers on the table OE.OC_ORDERS:

```
select trigger_name
,      status
from user_triggers
where table_name = 'OC_ORDERS';
```

TRIGGER_NAME	STATUS
-----	-----
ORDERS_TRG	DISABLED
ORDERS_ITEMS_TRG	DISABLED

Perform the following in SQL*Plus or Warehouse Builder to enable the specified constraint.

```
Execute WB_ENABLE_ALL_TRIGGERS ('OC_ORDERS');
```

TRIGGER_NAME	STATUS
-----	-----
ORDERS_TRG	ENABLED
ORDERS_ITEMS_TRG	ENABLED

WB_ENABLE_CONSTRAINT

Syntax

```
WB_ENABLE_CONSTRAINT(p_constraintname, p_tablename)
```

where *p_constraintname* is the constraint name to be disabled and *p_tablename* is the table name on which the specified constraint is disabled.

Purpose

This program unit disables the specified constraint that is owned by the table as stated in the call to the program. The user is the current user (in variable `USER`). For faster loading of data sets, you can disable constraints on a table. After the loading is complete, you must re-enable these constraints. This program unit shows you how to enable the constraints one at a time.

Example

The following example shows the enabling of the specified constraint on the table `OE.CUSTOMERS`:

```
select constraint_name
,      decode(constraint_type
, 'C', 'Check'
, 'P', 'Primary'
) Type
,      status
from user_constraints
where table_name = 'CUSTOMERS';
```

CONSTRAINT_NAME	TYPE	STATUS
CUST_FNAME_NN	Check	DISABLED
CUST_LNAME_NN	Check	DISABLED
CUSTOMER_CREDIT_LIMIT_MAX	Check	DISABLED
CUSTOMER_ID_MIN	Check	DISABLED
CUSTOMERS_PK	Primary	DISABLED

5 rows selected

Perform the following in SQL*Plus or Warehouse Builder to enable the specified constraint.

```
Execute WB_ENABLE_CONSTRAINT('CUSTOMERS_PK','CUSTOMERS');
```

CONSTRAINT_NAME	TYPE	STATUS
CUST_FNAME_NN	Check	DISABLED
CUST_LNAME_NN	Check	DISABLED
CUSTOMER_CREDIT_LIMIT_MAX	Check	DISABLED
CUSTOMER_ID_MIN	Check	DISABLED
CUSTOMERS_PK	Primary	DISABLED

5 rows selected

WB_ENABLE_TRIGGER

Syntax

```
WB_ENABLE_TRIGGER(p_name)
```

where `p_name` is the trigger name to be enabled.

Purpose

This program unit enables the specified trigger. The owner of the trigger must be the current user (in variable `USER`).

Example

The following example shows the enabling of a trigger on the table OE.OC_ORDERS:

```
select trigger_name
       ,      status
from user_triggers
where table_name = 'OC_ORDERS';
```

TRIGGER_NAME	STATUS
ORDERS_TRG	DISABLED
ORDERS_ITEMS_TRG	ENABLED

Perform the following in SQL*Plus or Warehouse Builder to enable the specified constraint.

```
Execute WB_ENABLE_TRIGGER ('ORDERS_TRG');
```

TRIGGER_NAME	STATUS
ORDERS_TRG	ENABLED
ORDERS_ITEMS_TRG	ENABLED

WB_TRUNCATE_TABLE**Syntax**

```
WB_TRUNCATE_TABLE(p_name)
```

where *p_name* is the table name to be truncated.

Purpose

This program unit truncates the table specified in the command call. The owner of the trigger must be the current user (in variable USER). The command disables and re-enables all referencing constraints to enable the truncate table command. Use this command in a pre-mapping process to explicitly truncate a staging table and ensure that all data in this staging table is newly loaded data.

Example

The following example shows the truncation of the table OE.OC_ORDERS:

```
select count(*) from oc_orders;
```

COUNT(*)
105

Perform the following in SQL*Plus or Warehouse Builder to enable the specified constraint.

```
Execute WB_TRUNCATE_TABLE ('OC_ORDERS');
```

COUNT(*)
0

Character Transformations

Character transformations enable Warehouse Builder users to perform transformations on Character objects. These transformations are ordered alphabetically. The custom functions provided with Warehouse Builder are prefixed with `WB_`.

The following character transformations are available in Warehouse Builder.

ASCII

Syntax

```
ascii::=ASCII(attribute)
```

Purpose

ASCII returns the decimal representation in the database character set of the first character of `attribute`. An `attribute` can be of data type CHAR, VARCHAR2, NCHAR, or NVARCHAR2. The value returned is of data type NUMBER. If your database character set is 7-bit ASCII, this function returns an ASCII value. If your database character set is EBCDIC Code, this function returns an EBCDIC value. There is no corresponding EBCDIC character function.

Example

The following example returns the ASCII decimal equivalent of the letter Q:

```
SELECT ASCII('Q') FROM DUAL;  
ASCII('Q')  
-----  
81
```

ASCIISTR

Syntax

```
asciistr::=ASCIISTR(attribute)
```

Purpose

ASCIISTR uses a string in any character set as its argument and returns an ASCII string in the database character set. The value returned contains only characters that appear in SQL and a forward slash (/).

Example

The following example returns the ASCII string equivalent of the text string "flauwekul":

```
SELECT ASCIISTR('flauwekul') FROM DUAL;  
  
ASCIISTR('FLAUW  
-----  
\6<65\756<\6700
```

CHARTOROWID

Syntax

```
chartorowid::=CHARTOROWID(attribute)
```

Purpose

CHARTOROWID converts a value from CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to ROWID data type.

Example

The following example converts a character rowid representation to a rowid. The function returns a different rowid on different databases.

```
SELECT last_name FROM employees
WHERE ROWID = CHARTOROWID('AAAFYmAAFAAAAFEAAP');
LAST_NAME
-----
Greene
```

CHR

Syntax

```
chr::=CHR(attribute)
```

Purpose

CHR returns the character with the binary equivalent to the number specified in the *attribute* in either the database character set or the national character set.

If `USING NCHAR_CS` is not specified, this function returns the character with the binary equivalent to *attribute* as a VARCHAR2 value in the database character set. If `USING NCHAR_CS` is specified in the expression builder, this function returns the character with the binary equivalent to *attribute* as a NVARCHAR2 value in the national character set.

Examples

The following example is run on an ASCII-based machine with the database character set defined as WE8ISO8859P1:

```
SELECT CHR(67) || CHR(65) || CHR(84) "Dog" FROM DUAL;

Dog
---
CAT
```

To produce the same results on an EBCDIC-based machine with the WE8EBCDIC1047 character set, modify the preceding example as follows:

```
SELECT CHR(195) || CHR(193) || CHR(227) "Dog"
FROM DUAL;

Dog
---
CAT
```

The following example uses the UTF8 character set:

```
SELECT CHR (50052 USING NCHAR_CS) FROM DUAL;  
CH  
--  
Ä
```

CONCAT

Syntax

```
concat::=CONCAT(attribute1, attribute2)
```

Purpose

CONCAT returns *attribute1* concatenated with *attribute2*. Both *attribute1* and *attribute2* can be CHAR or VARCHAR2 data types. The returned string is of VARCHAR2 data type contained in the same character set as *attribute1*. This function is equivalent to the concatenation operator (||).

Example

This example uses nesting to concatenate three character strings:

```
SELECT CONCAT(CONCAT(last_name, ''s job category is '),  
job_id) "Job"  
FROM employees  
WHERE employee_id = 152;
```

```
Job
```

```
-----  
Hall's job category is SA_REP
```

CONVERT

Syntax

```
convert::=CONVERT(attribute, dest_char_set, source_char_set)
```

Purpose

CONVERT converts a character string specified in an operator *attribute* from one character set to another. The data type of the returned value is VARCHAR2.

- The *attribute1* argument is the value to be converted. It can be of the data types CHAR and VARCHAR2.
- The *dest_char_set* argument is the name of the character set to which *attribute1* is converted.
- The *source_char_set* argument is the name of the character set in which *attribute1* is stored in the database. The default value is the database character set.

Both the destination and source character set arguments can be either literals or columns containing the name of the character set. For complete correspondence in character conversion, the destination character set must contain a representation of all the characters defined in the source character set. When a character does not exist in the destination character set, it is substituted with a replacement character. Replacement characters can be defined as part of a character set definition.

Example

The following example illustrates character set conversion by converting a Latin-1 string to ASCII. The result is the same as importing the same string from a WE8ISO8859P1 database to a US7ASCII database.

```
SELECT CONVERT('Ä Ê Í Õ Ø A B C D E ', 'US7ASCII', 'WE8ISO8859P1')
FROM DUAL;
```

```
CONVERT('ÄÊÍÕØABCDE'
-----
A E I ? ? A B C D E ?
```

Common character sets include:

- US7ASCII: US 7-bit ASCII character set
- WE8DEC: West European 8-bit character set
- WE8HP: HP West European Laserjet 8-bit character set
- F7DEC: DEC French 7-bit character set
- WE8EBCDIC500: IBM West European EBCDIC Code Page 500
- WE8PC850: IBM PC Code Page 850
- WE8ISO8859P1: ISO 8859-1 West European 8-bit character set

INITCAP**Syntax**

```
initcap ::= INITCAP(attribute)
```

Purpose

INITCAP returns the content of the *attribute* with the first letter of each word in uppercase and all other letters in lowercase. Words are delimited by white space or by characters that are not alphanumeric. *Attribute* can be of the data types CHAR or VARCHAR2. The return value is the same data type as *attribute*.

Example

The following example capitalizes each word in the string:

```
SELECT INITCAP('the soap') "Capitals" FROM DUAL;
Capitals
-----
The Soap
```

INSTR / INSTRB**Syntax**

```
instr ::= INSTR(attribute1, attribute2, n, m)
instrb ::= INSTRB(attribute1, attribute2, n, m)
```

Purpose

INSTR searches *attribute1* beginning with its *n*th character for the *m*th occurrence of *attribute2*. It returns the position of the character in *attribute1* that is the first character of this occurrence. INSTRB uses bytes instead of characters.

If `n` is negative, Oracle counts and searches backward from the end of `attribute1`. The value of `m` must be positive. The default values of both `n` and `m` are 1, which means that Oracle begins searching the first character of `attribute1` for the first occurrence of `attribute2`. The return value is relative to the beginning of `attribute1`, regardless of the value of `n`, and is expressed in characters. If the search is unsuccessful (if `attribute2` does not appear `m` times after the `n`th character of `attribute1`), then the return value is 0.

Examples

The following example searches the string "CORPORATE FLOOR", beginning with the third character, for the string "OR". It returns the position in `CORPORATE FLOOR` at which the second occurrence of "OR" begins:

```
SELECT INSTR('CORPORATE FLOOR','OR', 3, 2) "Instring" FROM DUAL;
```

```
Instring
-----
14
```

The next example begins searching at the third character from the end:

```
SELECT INSTR('CORPORATE FLOOR','OR', -3, 2)
"Reversed Instring"
FROM DUAL;
```

```
Reversed Instring
-----
2
```

This example assumes a double-byte database character set.

```
SELECT INSTRB('CORPORATE FLOOR','OR',5,2) "Instring in bytes"
FROM DUAL;
```

```
Instring in bytes
-----
27
```

LENGTH/LENGTHB

Syntax

```
length::=LENGTH(attribute)
lengthb::=LENGTHB(attribute)
```

Purpose

The length functions return the length of `char`. `LENGTH` calculates the length using characters as defined by the input character set. `LENGTHB` uses bytes instead of characters. The `attribute` can be of the data types `CHAR` or `VARCHAR2`. The return value is of data type `NUMBER`. If `attribute` has data type `CHAR`, the length includes all trailing blanks. If `attribute` contains a null value, this function returns null.

Example

The following examples use the `LENGTH` function using single- and multibyte database character set.

```
SELECT LENGTH('CANDIDE') "Length in characters"
FROM DUAL;
```

```
Length in characters
-----
7
```

This example assumes a double-byte database character set.

```
SELECT LENGTHB ('CANDIDE') "Length in bytes"
FROM DUAL;
```

```
Length in bytes
-----
14
```

LOWER

Syntax

```
lower::=LOWER(attribute)
```

Purpose

LOWER returns *attribute*, with all letters in lowercase. The *attribute* can be of the data types CHAR and VARCHAR2. The return value is the same data type as that of *attribute*.

Example

The following example returns a string in lowercase:

```
SELECT LOWER('MR. SCOTT MCMILLAN') "Lowercase"
FROM DUAL;
```

```
Lowercase
-----
mr. scott mcmillan
```

LPAD

Syntax

```
lpad::=LPAD(attribute1, n, attribute2)
```

Purpose

LPAD returns *attribute1*, left-padded to length *n* with the sequence of characters in *attribute2*. *Attribute2* defaults to a single blank. If *attribute1* is longer than *n*, this function returns the portion of *attribute1* that fits in *n*.

Both *attribute1* and *attribute2* can be of the data types CHAR and VARCHAR2. The string returned is of VARCHAR2 data type and is in the same character set as *attribute1*. The argument *n* is the total length of the return value as it is displayed on your screen. In most character sets, this is also the number of characters in the return value. However, in some multibyte character sets, the display length of a character string can differ from the number of characters in the string.

Example

The following example left-pads a string with the characters "*.":

```
SELECT LPAD('Page 1',15,'*.') "LPAD example"
FROM DUAL;
```

```
LPAD example
-----
*.*.*.*Page 1
```

LTRIM

Syntax

```
ltrim::=LTRIM(attribute, set)
```

Purpose

LTRIM removes characters from the left of *attribute*, with all the left most characters that appear in *set* removed. *Set* defaults to a single blank. If *attribute* is a character literal, you must enclose it in single quotes. Warehouse Builder begins scanning *attribute* from its first character and removes all characters that appear in *set* until it reaches a character absent in *set*. Then it returns the result.

Both *attribute* and *set* can be any of the data types CHAR and VARCHAR2. The string returned is of VARCHAR2 data type and is in the same character set as *attribute*.

Example

The following example trims all of the left-most x's and y's from a string:

```
SELECT LTRIM('xyxXxyLAST WORD', 'xy') "LTRIM example"
FROM DUAL;
```

```
LTRIM example
-----
XxyLAST WORD
```

NLSSORT

Syntax

```
nlssort::=NLSSORT(attribute, nlsparam)
```

Purpose

NLSSORT returns the string of bytes used to sort *attribute*. The parameter *attribute* is of type VARCHAR2. Use this function to compare based on a linguistic sort of sequence rather than on the binary value of a string.

The value of *nlsparam* can have the form 'NLS_SORT = sort' where *sort* is a linguistic sort sequence or BINARY. If you omit *nlsparam*, this function uses the default sort sequence for your session.

Example

The following example creates a table containing two values and shows how the values returned can be ordered by the NLSSORT function:

```
CREATE TABLE test (name VARCHAR2(15));
INSERT INTO TEST VALUES ('Gaardiner');
INSERT INTO TEST VALUES ('Gaberd');
```



```

SELECT * FROM test ORDER BY name;

NAME
-----
Gaardiner
Gaberd

SELECT *
  FROM test
  ORDER BY NLSSORT(name, 'NLSSORT = XDanish');

Name
-----
Gaberd
Gaardiner

```

NLS_INITCAP

Syntax

```
nls_initcap::=NLS_INITCAP(attribute, nlsparam)
```

Purpose

NLS_INITCAP returns *attribute*, with the first letter of each word in uppercase, all other letters in lowercase. Words are delimited by white space or characters that are not alphanumeric.

The value of *nlsparam* can have the form 'NLS_SORT = *sort*', where *sort* is either a linguistic sort sequence or BINARY. The linguistic sort sequence handles special linguistic requirements for case conversions. These requirements can result in a return value of a different length than the *attribute*. If you omit 'nlsparam', this function uses the default sort sequence for your session.

Example

The following examples show how the linguistic sort sequence results in a different return value from the function:

```

SELECT NLS_INITCAP('ijsland') "InitCap"
  FROM dual;

InitCap
-----
Ijsland

SELECT NLS_INITCAP('ijsland', 'NLS_SORT=XDutch') "InitCap"
  FROM dual;

InitCap
-----
IJslan

```

NLS_LOWER

Syntax

```
nls_lower::=NLS_LOWER(attribute, nlsparam)
```

Purpose

NLS_LOWER returns *attribute*, with all letters lowercase. Both *attribute* and *nlsparam* can be any of the data types CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The string returned is of data type VARCHAR2 and is in the same character set as *attribute*. The value of *nlsparam* can have the form 'NLS_SORT = *sort*', where *sort* is either a linguistic sort sequence or BINARY.

Example

The following example returns the character string 'citta' using the XGerman linguistic sort sequence:

```
SELECT NLS_LOWER('CITTA', 'NLS_SORT=XGerman') "Lowercase"
       FROM DUAL;
```

```
Lowercase
-----
citta'
```

NLS_UPPER

Syntax

```
nl_upper::=NLS_UPPER(attribute, nlsparam)
```

Purpose

NLS_UPPER returns *attribute*, with all letters uppercase. Both *attribute* and *nlsparam* can be any of the data types CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The string returned is of VARCHAR2 data type and is in the same character set as *attribute*. The value of *nlsparam* can have the form 'NLS_SORT = *sort*', where *sort* is either a linguistic sort sequence or BINARY.

Example

The following example returns a string with all letters converted to uppercase:

```
SELECT NLS_UPPER('große') "Uppercase"
       FROM DUAL;
```

```
Uppercase
-----
GROßE
```

```
SELECT NLS_UPPER('große', 'NLS_SORT=XGerman') "Uppercase"
       FROM DUAL;
```

```
Uppercase
-----
GROSSE
```

REPLACE

Syntax

```
replace::=REPLACE(attribute, 'search_string', 'replace_string')
```

Purpose

REPLACE returns an attribute with every occurrence of `search_string` replaced with `replacement_string`. If `replacement_string` is omitted or null, all occurrences of `search_string` are removed. If `search_string` is null, attribute is returned.

Both `search_string` and `replacement_string`, as well as `attribute`, can be of the data types CHAR or VARCHAR2. The string returned is of VARCHAR2 data type and is in the same character set as `attribute`.

This function provides a superset of the functionality provided by the TRANSLATE function. TRANSLATE provides single-character, one-to-one substitution. REPLACE enables you to substitute one string for another, as well as to remove character strings.

Example

The following example replaces occurrences of "J" with "BL":

```
SELECT REPLACE('JACK and JUE','J','BL') "Changes"
FROM DUAL;
Changes
-----
BLACK and BLUE
```

RPAD**Syntax**

```
rpad::=RPAD(attribute1, n, attribute2)
```

Purpose

RPAD returns `attribute1`, right-padded to length `n` with `attribute2`, replicated as many times as necessary. `Attribute2` defaults to a single blank. If `attribute1` is longer than `n`, this function returns the portion of `attribute1` that fits in `n`.

Both `attribute1` and `attribute2` can be of the data types CHAR or VARCHAR2. The string returned is of VARCHAR2 data type and is in the same character set as `attribute1`.

The argument `n` is the total length of the return value as it is displayed on your screen. In most character sets, this is also the number of characters in the return value. However, in some multibyte character sets, the display length of a character string can differ from the number of characters in the string.

Example

The following example rights-pads a name with the letters "ab" until it is 12 characters long:

```
SELECT RPAD('MORRISON',12,'ab') "RPAD example"
FROM DUAL;
RPAD example
-----
MORRISONabab
```

RTRIM

Syntax

```
rtrim::=RTRIM(attribute, set)
```

Purpose

RTRIM returns *attribute*, with all the right most characters that appear in *set* removed; *set* defaults to a single blank. If *attribute* is a character literal, you must enclose it in single quotes. RTRIM works similarly to LTRIM. Both *attribute* and *set* can be any of the data types CHAR or VARCHAR2. The string returned is of VARCHAR2 data type and is in the same character set as *attribute*.

Example

The following example trims the letters "xy" from the right side of a string:

```
SELECT RTRIM('BROWNINGyxXxy', 'xy') "RTRIM e.g."
FROM DUAL;
```

```
RTRIM e.g.
-----
BROWNINGyxX
```

SOUNDEX

Syntax

```
soundex::=SOUNDEX(attribute)
```

Purpose

SOUNDEX returns a character string containing the phonetic representation of *attribute*. This function enables you to compare words that are spelled differently, but sound similar in English.

The phonetic representation is defined in *The Art of Computer Programming, Volume 3: Sorting and Searching*, by Donald E. Knuth, as follows:

- Retain the first letter of the string and remove all other occurrences of the following letters: a, e, h, i, o, u, w, y.
- Assign numbers to the remaining letters (after the first) as follows:
 - b, f, p, v = 1
 - c, g, j, k, q, s, x, z = 2
 - d, t = 3
 - l = 4
 - m, n = 5
 - r = 6
- If two or more letters with the same number were adjacent in the original name (before step 1), or adjacent except for any intervening h and w, omit all but the first.
- Return the first four bytes padded with 0.

Data types for `attribute` can be CHAR and VARCHAR2. The return value is the same data type as `attribute`.

Example

The following example returns the employees whose last names are a phonetic representation of "Smyth":

```
SELECT last_name, first_name
FROM hr.employees
WHERE SOUNDEX(last_name)
= SOUNDEX('SMYTHE');
```

```
LAST_NAME  FIRST_NAME
-----
Smith      Lindsey
```

SUBSTR

Syntax

```
substr::=SUBSTR(attribute, position, substring_length)
substrb::=SUBSTRB(attribute, position, substring_length)
```

Purpose

The substring functions return a portion of `attribute`, beginning at character `position`, `substring_length` characters long. SUBSTR calculates lengths using characters as defined by the input character set. SUBSTRB uses bytes instead of characters.

- If `position` is 0, it is treated as 1.
- If `position` is positive, Warehouse Builder counts from the beginning of `attribute` to find the first character.
- If `position` is negative, Warehouse Builder counts backward from the end of `attribute`.
- If `substring_length` is omitted, Warehouse Builder returns all characters to the end of `attribute`. If `substring_length` is less than 1, a null is returned.

Data types for `attribute` can be CHAR and VARCHAR2. The return value is the same data type as `attribute`. Floating-point numbers passed as arguments to SUBSTR are automatically converted to integers.

Examples

The following example returns several specified substrings of "ABCDEFGH":

```
SELECT SUBSTR('ABCDEFGH',3,4) "Substring"
FROM DUAL;
```

```
Substring
-----
CDEF
```

```
SELECT SUBSTR('ABCDEFGH',-5,4) "Substring"
FROM DUAL;
```

```
Substring
```

```
-----  
CDEF
```

Assume a double-byte database character set:

```
SELECT SUBSTRB('ABCDEFGF',5,4,2) "Substring with bytes"  
FROM DUAL;
```

```
Substring with bytes  
-----  
CD
```

TO_DATE

Syntax

```
to_date::=TO_DATE(attribute, fmt, nlsparam)
```

Purpose

TO_DATE converts *attribute* of CHAR or VARCHAR2 data type to a value of data type DATE. The *fmt* is a date format specifying the format of *attribute*. If you omit *fmt*, *attribute* must be in the default date format. If *fmt* is 'J', for Julian, then *attribute* must be an integer. The *nlsparam* has the same purpose in this function as in the TO_CHAR function for date conversion.

Do not use the TO_DATE function with a DATE value for the *attribute* argument. The first two digits of the returned DATE value can differ from the original *attribute*, depending on *fmt* or the default date format.

Example

The following example converts character strings into dates:

```
SELECT TO_DATE(  
'January 15, 1989, 11:00 A.M.',  
'Month dd, YYYY, HH:MI A.M.',  
'NLS_DATE_LANGUAGE = American')  
FROM DUAL;  
  
TO_DATE  
-----  
15-JAN-89
```

TO_MULTI_BYTE

Syntax

```
to_multi_byte::=TO_MULTI_BYTE(attribute)
```

Purpose

TO_MULTI_BYTE returns *attribute* with all of its single-byte characters converted to their corresponding multibyte characters; *attribute* can be of data type CHAR or VARCHAR2. The value returned is in the same data type as *attribute*. Any single-byte characters in *attribute* that have no multibyte equivalents appear in the output string as single-byte characters.

This function is useful only if your database character set contains both single-byte and multibyte characters.

Example

The following example illustrates converting from a single byte 'A' to a multi byte.

```
'A' in UTF8:
SELECT dump(TO_MULTI_BYTE( 'A')) FROM DUAL;
DUMP(TO_MULTI_BYTE('A'))
-----
Typ=1 Len=3: 239,188,161
```

TO_NUMBER**Syntax**

```
to_number ::= TO_NUMBER(attribute, fmt, nlsparam)
```

Purpose

TO_NUMBER converts *attribute* to a value of CHAR or VARCHAR2 data type containing a number in the format specified by the optional format *model_fmt*, to a value of NUMBER data type.

Examples

The following example converts character string data into a number:

```
UPDATE employees
SET salary = salary + TO_NUMBER('100.00', '9G999D99')
WHERE last_name = 'Perkins';
```

The *nlsparam* string in this function has the same purpose as it does in the TO_CHAR function for number conversions.

```
SELECT TO_NUMBER('-AusDollars100', 'L9G999D99',
' NLS_NUMERIC_CHARACTERS = ', '.')
NLS_CURRENCY = 'AusDollars'
) "Amount"
FROM DUAL;
```

```
Amount
-----
-100
```

TO_SINGLE_BYTE**Syntax**

```
to_single_byte ::= TO_SINGLE_BYTE(attribute)
```

Purpose

TO_SINGLE_BYTE returns *attribute* with all of its multibyte characters converted to their corresponding single-byte characters; *attribute* can be of data type CHAR or VARCHAR2. The value returned is in the same data type as *attribute*. Any multibyte characters in *attribute* that have no single-byte equivalents appear in the output as multibyte characters.

This function is useful only if your database character set contains both single-byte and multibyte characters.

Example

The following example illustrates going from a multibyte 'A' in UTF8 to a single byte ASCII 'A':

```
SELECT TO_SINGLE_BYTE( CHR(15711393)) FROM DUAL;
T
-
A
```

TRANSLATE

Syntax

```
translate::=TRANSLATE(attribute, from_string, to_string)
```

Purpose

TRANSLATE returns *attribute* with all occurrences of each character in *from_string* replaced by its corresponding character in *to_string*. Characters in *attribute* that are not in *from_string* are not replaced. The argument *from_string* can contain more characters than *to_string*. In this case, the extra characters at the end of *from_string* have no corresponding characters in *to_string*. If these extra characters appear in *attribute*, they are removed from the return value.

You cannot use an empty string for *to_string* to remove all characters in *from_string* from the return value. Warehouse Builder interprets the empty string as null, and if this function has a null argument, it returns null.

Examples

The following statement translates a license number. All letters 'ABC...Z' are translated to 'X' and all digits '012...9' are translated to '9':

```
SELECT TRANSLATE('2KRW229',
'0123456789ABCDEFGHIJKLMNQRSTUUVWXYZ',
'9999999999XXXXXXXXXXXXXXXXXXXXXXXXXXXX') "License"
FROM DUAL;
License
-----
9XXX999
```

The following statement returns a license number with the characters removed and the digits remaining:

```
SELECT TRANSLATE('2KRW229',
'0123456789ABCDEFGHIJKLMNQRSTUUVWXYZ', '0123456789') "Translate example"
FROM DUAL;

Translate example
-----
2229
```

TRIM

Syntax

```
trim::=TRIM(attribute)
```


Purpose

TRIM enables you to trim leading or trailing spaces (or both) from a character string. The function returns a value with data type VARCHAR2. The maximum length of the value is the length of *attribute*.

Example

This example trims leading and trailing spaces from a string:

```
SELECT TRIM ('   Warehouse   ') "TRIM Example"
FROM DUAL;
TRIM example
-----
Warehouse
```

UPPER**Syntax**

```
upper::=UPPER(attribute)
```

Purpose

UPPER returns *attribute*, with all letters in uppercase; *attribute* can be of the data types CHAR and VARCHAR2. The return value is the same data type as *attribute*.

Example

The following example returns a string in uppercase:

```
SELECT UPPER('Large') "Uppercase"
FROM DUAL;
Upper
-----
LARGE
```

WB.LOOKUP_CHAR**Syntax**

```
WB.LOOKUP_CHAR (table_name
, column_name
, key_column_name
, key_value
)
```

where *table_name* is the name of the table to perform the lookup on and *column_name* is the name of the VARCHAR2 column that will be returned. For example, the result of the lookup *key_column_name* is the name of the NUMBER column used as the key to match on in the lookup table, *key_value* is the value of the key column mapped into the *key_column_name* with which the match will be done.

Purpose

To perform a key lookup on a number that returns a VARCHAR2 value from a database table using a NUMBER column as the matching key.

Example

Consider the following table as a lookup table LKP1:

KEY_COLUMN	TYPE	COLOR
10	Car	Red
20	Bike	Green

Using this package with the following call:

```
WB.LOOKUP_CHAR ('LKP1'  
, 'TYPE'  
, 'KEYCOLUMN'  
, 20  
)
```

returns the value of 'Bike' as output of this transform. This output would then be processed in the mapping as the result of an inline function call.

Note: This function is a row-based key lookup. Set-based lookups are supported when you use the lookup operator.

WB.LOOKUP_CHAR

```
Syntax  
WB.LOOKUP_CHAR (table_name  
, column_name  
, key_column_name  
, key_value  
)
```

where `table_name` is the name of the table to perform the lookup on; `column_name` is the name of the VARCHAR2 column that will be returned, for instance, the result of the lookup; `key_column_name` is the name of the VARCHAR2 column used as the key to match on in the lookup table; `key_value` is the value of the key column, for instance, the value mapped into the `key_column_name` with which the match will be done.

Purpose

To perform a key lookup on a VARCHAR2 character that returns a VARCHAR2 value from a database table using a VARCHAR2 column as the matching key.

Example

Consider the following table as a lookup table LKP1:

KEYCOLUMN	TYPE	COLOR
ACV	Car	Red
ACP	Bike	Green

Using this package with the following call:

```
WB.LOOKUP_CHAR ('LKP1'  
, 'TYPE'  
, 'KEYCOLUMN'  
, 'ACP'  
)
```

returns the value of 'Bike' as output of this transformation. This output is then processed in the mapping as the result of an inline function call.

Note: This function is a row-based key lookup. Set-based lookups are supported when you use the lookup operator.

WB_IS_SPACE

Syntax

```
WB_IS_SPACE(attribute)
```

Purpose

Checks whether a string value only contains spaces. In mainframe sources, some fields contain many spaces to make a file adhere to the fixed length format. This function provides a way to check for these spaces. The function always returns a Boolean value.

Example

WB_IS_SPACE returns true if `attribute` contains only spaces.

Date Transformations

Date transformations provide Warehouse Builder users with functionality to perform transformations on date attributes. These transformations are ordered and the custom functions provided with Warehouse Builder are all in the format `WB_<function name>`.

All date transformations provided with Warehouse Builder are listed in alphabetical order in the following sections.

ADD_MONTHS

Syntax

```
add_months::=ADD_MONTHS(attribute, n)
```

Purpose

ADD_MONTHS returns the date in the `attribute` plus `n` months. The argument `n` can be any integer. This will typically be added from an `attribute` or from a constant.

If the date in `attribute` is the last day of the month or if the resulting month has fewer days than the day component of `attribute`, then the result is the last day of the resulting month. Otherwise, the result has the same day component as `attribute`.

Example

The following example returns the month after the `hire_date` in the sample table `employees`:

```
SELECT TO_CHAR(ADD_MONTHS(hire_date,1), 'DD-MON-YYYY') "Next month"
FROM employees
WHERE last_name = 'Baer';
Next Month
-----
07-JUL-1994
```

LAST_DAY

Syntax

```
last_day ::= LAST_DAY(attribute)
```

Purpose

LAST_DAY returns the date of the last day of the month that contains the date in attribute.

Examples

The following statement determines how many days are left in the current month.

```
SELECT SYSDATE,  
LAST_DAY(SYSDATE) "Last",  
LAST_DAY(SYSDATE) - SYSDATE "Days Left"  
FROM DUAL;  
SYSDATE    Last Days Left  
-----  
23-OCT-97  31-OCT-97  8
```

MONTHS_BETWEEN

Syntax

```
months_between ::= MONTHS_BETWEEN(attribute1, attribute2)
```

Purpose

MONTHS_BETWEEN returns the number of months between dates in attribute1 and attribute2. If attribute1 is later than attribute2, the result is positive; if earlier, then the result is negative.

If attribute1 and attribute2 are either the same day of the month or both last days of months, the result is always an integer. Otherwise, Oracle calculates the fractional portion of the result-based on a 31-day month and considers the difference in time components attribute1 and attribute2.

Example

The following example calculates the months between two dates:

```
SELECT MONTHS_BETWEEN  
(TO_DATE('02-02-1995', 'MM-DD-YYYY'),  
TO_DATE('01-01-1995', 'MM-DD-YYYY') ) "Months"  
FROM DUAL;  
  
Months  
-----  
1.03225806
```

NEW_TIME

Syntax

```
new_time ::= NEW_TIME(attribute, zone1, zone2)
```

Purpose

`NEW_TIME` returns the date and time in time zone `zone2` when date and time in time zone `zone1` are the value in `attribute`. Before using this function, you must set the `NLS_DATE_FORMAT` parameter to display 24-hour time.

The arguments `zone1` and `zone2` can be any of these text strings:

- AST, ADT: Atlantic Standard or Daylight Time
- BST, BDT: Bering Standard or Daylight Time
- CST, CDT: Central Standard or Daylight Time
- CST, EDT: Eastern Standard or Daylight Time
- GMT: Greenwich Mean Time
- HST, HDT: Alaska-Hawaii Standard Time or Daylight Time.
- MST, MDT: Mountain Standard or Daylight Time
- NST: Newfoundland Standard Time
- PST, PDT: Pacific Standard or Daylight Time
- YST, YDT: Yukon Standard or Daylight Time

Example

The following example returns an Atlantic Standard time, given the Pacific Standard time equivalent:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
SELECT NEW_TIME
(TO_DATE('11-10-99 01:23:45', 'MM-DD-YY HH24:MI:SS'),
'AST', 'PST') "New Date and Time"
FROM DUAL;
```

```
New Date and Time
-----
09-NOV-1999 21:23:45
```

NEXT_DAY

Syntax

```
next_day::=NEXT_DAY(attribute, attribute2)
```

Purpose

`NEXT_DAY` returns the date of the first weekday named by the string in `attribute2` that is later than the date in `attribute1`. The argument `attribute2` must be a day of the week in the date language of your session, either the full name or the abbreviation. The minimum number of letters required is the number of letters in the abbreviated version. Any characters immediately following the valid abbreviation are ignored. The return value has the same hours, minutes, and seconds component as the argument `attribute1`.

Example

This example returns the date of the next Tuesday after February 2, 2001:

```
SELECT NEXT_DAY('02-FEB-2001', 'TUESDAY') "NEXT DAY"
FROM DUAL;
```

```
      NEXT DAY
-----
06-FEB-2001
```

ROUND (date)

Syntax

```
round_date::=ROUND(attribute, fmt)
```

Purpose

ROUND returns the date in `attribute` rounded to the unit specified by the format model `fmt`. If you omit `fmt`, date is rounded to the nearest day.

Example

The following example rounds a date to the first day of the following year:

```
SELECT ROUND (TO_DATE ('27-OCT-00'), 'YEAR') "New Year"
FROM DUAL;
```

```
New Year
-----
01-JAN-01
```

SYSDATE

Syntax

```
sysdate::=SYSDATE
```

Purpose

SYSDATE returns the current date and time. The data type of the returned value is DATE. The function requires no arguments. In distributed SQL statements, this function returns the date and time on your local database. You cannot use this function in the condition of a CHECK constraint.

Example

The following example returns the current date and time:

```
SELECT TO_CHAR(SYSDATE, 'MM-DD-YYYY HH24:MI:SS') "NOW" FROM DUAL;
```

```
NOW
-----
04-13-2001 09:45:51
```

TO_CHAR (datetime)

Syntax

```
to_char_date::=TO_CHAR(attribute, fmt, nlsparam)
```

Purpose

TO_CHAR converts attribute of DATE data type to a value of VARCHAR2 data type in the format specified by the date format *fmt*. If you omit *fmt*, date is converted to a VARCHAR2 value in the default date format.

The *nlsparams* specifies the language in which month and day names and abbreviations are returned. This argument can have this form: 'NLS_DATE_LANGUAGE = language' If you omit *nlsparams*, this function uses the default date language for your session.

Example

The following example applies various conversions on the sysdate in the database:

```
select to_char(sysdate) no_fmt
from dual;
NO_FMT
-----
26-MAR-02

select to_char(sysdate, 'dd-mm-yyyy') fnted
from dual;

FMTED
-----
26-03-2002
```

TRUNC (date)**Syntax**

```
trunc_date::=TRUNC(attribute, fmt)
```

Purpose

TRUNC returns attribute with the time portion of the day truncated to the unit specified by the format model *fmt*. If you omit *fmt*, date is truncated to the nearest day.

Example

The following example truncates a date:

```
SELECT TRUNC(TO_DATE('27-OCT-92','DD-MON-YY'),'YEAR') "New Year"
FROM DUAL;

New Year
-----
01-JAN-92
```

WB_CAL_MONTH_NAME**Syntax**

```
WB_CAL_MONTH_NAME(attribute)
```

Purpose

The function call returns the full-length name of the month for the date specified in attribute.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_CAL_MONTH_NAME(sysdate)
from dual;

WB_CAL_MONTH_NAME(SYSDATE)
-----
March

select WB_CAL_MONTH_NAME('26-MAR-2002')
from dual;

WB_CAL_MONTH_NAME('26-MAR-2002')
-----
March
```

WB_CAL_MONTH_OF_YEAR**Syntax**

```
WB_CAL_MONTH_OF_YEAR(attribute)
```

Purpose

`WB_CAL_MONTH_OF_YEAR` returns the month (1-12) of the year for date in attribute.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_CAL_MONTH_OF_YEAR(sysdate) month
from dual;

      MONTH
-----
        3

select WB_CAL_MONTH_OF_YEAR('26-MAR-2002') month
from dual;

      MONTH
-----
        3
```

WB_CAL_MONTH_SHORT_NAME**Syntax**

```
WB_CAL_MONTH_SHORT_NAME(attribute)
```

Purpose

`WB_CAL_MONTH_SHORT_NAME` returns the short name of the month (for example 'Jan') for date in attribute.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_CAL_MONTH_SHORT_NAME (sysdate) month
from dual;
```

```
MONTH
-----
Mar
```

```
select WB_CAL_MONTH_SHORT_NAME ('26-MAR-2002') month
from dual;
```

```
MONTH
-----
Mar
```

WB_CAL_QTR**Syntax**

```
WB_CAL_QTR(attribute)
```

Purpose

`WB_CAL_QTR` returns the quarter of the Gregorian calendar year (for example Jan - March = 1) for date in `attribute`.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_CAL_QTR (sysdate) quarter
from dual;
```

```
QUARTER
-----
1
```

```
select WB_CAL_QTR ('26-MAR-2002') quarter
from dual;
```

```
QUARTER
-----
1
```

WB_CAL_WEEK_OF_YEAR**Syntax**

```
WB_CAL_WEEK_OF_YEAR(attribute)
```

Purpose

`WB_CAL_WEEK_OF_YEAR` returns the week of the year (1-53) for date `attribute`.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_CAL_WEEK_OF_YEAR (sysdate) w_of_y
from dual;
```

```
      W_OF_Y
-----
          13
```

```
select WB_CAL_WEEK_OF_YEAR ('26-MAR-2002') w_of_y
from dual;
```

```
      W_OF_Y
-----
          13
```

WB_CAL_YEAR**Syntax**

```
WB_CAL_YEAR(attribute)
```

Purpose

`WB_CAL_YEAR` returns the numerical year component for a date in `attribute`.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_CAL_YEAR (sysdate) year
from dual;
```

```
      YEAR
-----
      2002
```

```
select WB_CAL_YEAR ('26-MAR-2002') w_of_y
from dual;
```

```
      YEAR
-----
      2002
```

WB_CAL_YEAR_NAME**Syntax**

```
WB_CAL_YEAR_NAME(attribute)
```

Purpose

`WB_CAL_YEAR_NAME` returns the spelled out name of the year for the date in `attribute`.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_CAL_YEAR_NAME (sysdate) name
from dual;
```

```
NAME
```

```
-----
Two Thousand Two
```

```
select WB_CAL_YEAR_NAME ('26-MAR-2001') name
from dual;
```

```
NAME
```

```
-----
Two Thousand One
```

WB_DATE_FROM_JULIAN**Syntax**

```
WB_DATE_FROM_JULIAN(attribute)
```

Purpose

`WB_DATE_FROM_JULIAN` converts Julian date `attribute` to a regular date.

Example

The following example shows the return value on a specified Julian date:

```
select to_char(WB_DATE_FROM_JULIAN(3217345), 'dd-mon-yyyy') JDate
from dual;
```

```
JDATE
```

```
-----
08-sep-4096
```

WB_DAY_NAME**Syntax**

```
WB_DAY_NAME(attribute)
```

Purpose

`WB_DAY_NAME` returns the full name of the day for `date attribute`.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_DAY_NAME (sysdate) name
from dual;
```

```
NAME
```

```
-----
Thursday
```

```
select WB_DAY_NAME ('26-MAR-2002') name
from dual;
```

```
NAME
```

```
-----
Tuesday
```

WB_DAY_OF_MONTH

Syntax

```
WB_DAY_OF_MONTH(attribute)
```

Purpose

WB_DAY_OF_MONTH returns the day number within the month for date attribute.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_DAY_OF_MONTH (sysdate) num
from dual;
```

```
      NUM
-----
      28
```

```
select WB_DAY_OF_MONTH ('26-MAR-2002') num
from dual
```

```
      NUM
-----
      26
```

WB_DAY_OF_WEEK

Syntax

```
WB_DAY_OF_WEEK(attribute)
```

Purpose

WB_DAY_OF_WEEK returns the day number within the week for date attribute based on the database calendar.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_DAY_OF_WEEK (sysdate) num
from dual;
```

```
      NUM
-----
       5
```

```
select WB_DAY_OF_WEEK ('26-MAR-2002') num
from dual;
```

```

      NUM
-----
      3

```

WB_DAY_OF_YEAR

Syntax

```
WB_DAY_OF_YEAR(attribute)
```

Purpose

WB_DAY_OF_YEAR returns the day number within the year for the date attribute.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_DAY_OF_YEAR (sysdate) num
from dual;
```

```

      NUM
-----
      87

```

```
select WB_DAY_OF_YEAR ('26-MAR-2002') num
from dual;
```

```

      NUM
-----
      85

```

WB_DAY_SHORT_NAME

Syntax

```
WB_DAY_SHORT_NAME(attribute)
```

Purpose

WB_DAY_SHORT_NAME returns the three letter abbreviation or name for the date attribute.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_DAY_SHORT_NAME (sysdate) abbr
from dual;
```

```

ABBR
-----
Thu

```

```
select WB_DAY_SHORT_NAME ('26-MAR-2002') abbr
from dual;
```

```
NUM
-----
Tue
```

WB_DECADE

Syntax

```
WB_DECADE(attribute)
```

Purpose

WB_DECADE returns the decade number within the century for the date *attribute*.

Example

The following example shows the return value on the *sysdate* and on a specified date string:

```
select WB_DECADE (sysdate) dcd
from dual;
```

```
DCD
-----
2
```

```
select WB_DECADE ('26-MAR-2002') DCD
from dual;
```

```
DCD
-----
2
```

WB_HOUR12

Syntax

```
WB_HOUR12(attribute)
```

Purpose

WB_HOUR12 returns the hour (in a 12-hour setting) component of the date corresponding to *attribute*.

Example

The following example shows the return value on the *sysdate* and on a specified date string:

```
select WB_HOUR12 (sysdate) h12
from dual;
```

```
H12
-----
9
```

```
select WB_HOUR12 ('26-MAR-2002') h12
from dual;
```

```
H12
-----
```

12

Note: For a date not including the timestamp (in the second example), Oracle uses the 12:00 (midnight) timestamp and therefore returns 12 in this case.

WB_HOUR12MI_SS

Syntax

```
WB_HOUR12MI_SS(attribute)
```

Purpose

WB_HOUR12MI_SS returns the timestamp in *attribute* formatted to HH12:MI:SS.

Example

The following example shows the return value on the *sysdate* and on a specified date string:

```
select WB_HOUR12MI_SS (sysdate) h12miss
from dual;
```

```
H12MISS
```

```
-----
09:08:52
```

```
select WB_HOUR12MI_SS ('26-MAR-2002') h12miss
from dual;
```

```
H12MISS
```

```
-----
12:00:00
```

Note: For a date not including the timestamp (in the second example), Oracle uses the 12:00 (midnight) timestamp and therefore returns 12 in this case.

WB_HOUR24

Syntax

```
WB_HOUR24(attribute)
```

Purpose

WB_HOUR24 returns the hour (in a 24-hour setting) component of date corresponding to *attribute*.

Example

The following example shows the return value on the *sysdate* and on a specified date string:

```
select WB_HOUR24 (sysdate) h24
```

```
from dual;

      H24
-----
      9

select WB_HOUR24 ('26-MAR-2002') h24
from dual;

      H24
-----
      0
```

Note: For a date not including the timestamp (in the second example), Oracle uses the 00:00:00 timestamp and therefore returns the timestamp in this case.

WB_HOUR24MI_SS

Syntax

WB_HOUR24MI_SS(attribute)

Purpose

WB_HOUR24MI_SS returns the timestamp in *attribute* formatted to HH24:MI:SS.

Example

The following example shows the return value on the *sysdate* and on a specified date string:

```
select WB_HOUR24MI_SS (sysdate) h24miss
from dual;

H24MISS
-----
09:11:42

select WB_HOUR24MI_SS ('26-MAR-2002') h24miss
from dual;

H24MISS
-----
00:00:00
```

Note: For a date not including the timestamp (in the second example), Oracle uses the 00:00:00 timestamp and therefore returns the timestamp in this case.

WB_IS_DATE

Syntax

WB_IS_DATE(attribute, fmt)

Purpose

To check whether `attribute` contains a valid date. The function returns a Boolean value which is set to `true` if `attribute` contains a valid date. `fmt` is an optional date format. If `fmt` is omitted, the date format of your database session is used.

You can use this function when you validate your data before loading it into a table. This way the value can be transformed before it reaches the table and causes an error.

Example

`WB_IS_DATE` returns `true` in PL/SQL if `attribute` contains a valid date.

WB_JULIAN_FROM_DATE**Syntax**

```
WB_JULIAN_FROM_DATE(attribute)
```

Purpose

`WB_JULIAN_FROM_DATE` returns the Julian date of date corresponding to `attribute`.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_JULIAN_FROM_DATE (sysdate) jdate
from dual;
```

```
      JDATE
-----
      2452362
```

```
select WB_JULIAN_FROM_DATE ('26-MAR-2002') jdate
from dual;
```

```
      JDATE
-----
      2452360
```

WB_MI_SS**Syntax**

```
WB_MI_SS(attribute)
```

Purpose

`WB_MI_SS` returns the minutes and seconds of the time component in the date corresponding to `attribute`.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_MI_SS (sysdate) mi_ss
from dual;
```

```

MI_SS
-----
33:23

select WB_MI_SS ('26-MAR-2002') mi_ss
from dual;

MI_SS
-----
00:00

```

Note: For a date not including the timestamp (in the second example), Oracle uses the 00:00:00 timestamp and therefore returns the timestamp in this case.

WB_WEEK_OF_MONTH

Syntax

```
WB_WEEK_OF_MONTH(attribute)
```

Purpose

`WB_WEEK_OF_MONTH` returns the week number within the calendar month for the date corresponding to `attribute`.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```

select WB_WEEK_OF_MONTH (sysdate) w_of_m
from dual;

  W_OF_M
-----
      4

select WB_WEEK_OF_MONTH ('26-MAR-2002') w_of_m
from dual;

  W_OF_M
-----
      4

```

Number Transformations

These transforms are ordered alphabetically and the custom functions provided with Warehouse Builder are prefixed with `WB_`.

All numerical transformations provided with Warehouse Builder are listed in alphabetical order in the following sections.

ABS

Syntax

```
abs::=ABS(attribute)
```

Purpose

ABS returns the absolute value of `attribute`.

Example

The following example returns the absolute value of -15:

```
SELECT ABS(-15) "Absolute" FROM DUAL;
Absolute
-----
15
```

ACOS

Syntax

```
acos::= ACOS(attribute)
```

Purpose

ACOS returns the arc cosine of `attribute`. The argument `attribute` must be in the range of -1 to 1, and the function returns values in the range of 0 to pi, expressed in radians.

Example

The following example returns the arc cosine of .3:

```
SELECT ACOS(.3) "Arc_Cosine" FROM DUAL;

Arc_Cosine
-----
1.26610367
```

ASIN

Syntax

```
asin::=ASIN(attribute)
```

Purpose

ASIN returns the arc sine of `attribute`. The argument `attribute` must be in the range of -1 to 1, and the function returns values in the range of -pi/2 to pi/2, expressed in radians.

Example

The following example returns the arc cosine of .3:

```
SELECT ACOS(.3) "Arc_Sine" FROM DUAL;

Arc_Sine
-----
```

```
.304692654
```

ATAN

Syntax

```
atan::=ATAN(attribute)
```

Purpose

ATAN returns the arc tangent of *attribute*. The argument *attribute* can be in an unbounded range, and the function returns values in the range of $-\pi/2$ to $\pi/2$, expressed in radians.

Example

The following example returns the arc tangent of .3:

```
SELECT ATAN(.3) "Arc_Tangent" FROM DUAL;
```

```
Arc_Tangent  
-----  
.291456794
```

ATAN2

Syntax

```
atan2::=ATAN2(attribute1, attribute2)
```

Purpose

ATAN2 returns the arc tangent of *attribute1* and *attribute2*. The argument *attribute1* can be in an unbounded range, and the function returns values in the range of $-\pi$ to π , depending on the signs of *attribute1* and *attribute2*, and are expressed in radians. `ATAN2(attribute1, attribute2)` is the same as `ATAN2(attribute1/attribute2)`.

Example

The following example returns the arc tangent of .3 and .2:

```
SELECT ATAN2(.3,.2) "Arc_Tangent2" FROM DUAL;
```

```
Arc_Tangent2  
-----  
.982793723
```

COS

Syntax

```
cos::=COS(attribute)
```

Purpose

COS returns the cosine of *attribute* (an angle expressed in degrees).

Example

The following example returns the cosine of 180 degrees:

```
SELECT COS(180 * 3.14159265359/180) "Cosine" FROM DUAL;
```

```
Cosine
-----
      -1
```

COSH**Syntax**

```
cosh::=COSH(attribute)
```

Purpose

COSH returns the hyperbolic cosine of *attribute*.

Example

The following example returns the hyperbolic cosine of 0:

```
SELECT COSH(0) "Hyperbolic Cosine" FROM DUAL;
```

```
Hyperbolic Cosine
-----
                  1
```

CEIL**Syntax**

```
ceil::=CEIL(attribute)
```

Purpose

CEIL returns smallest integer greater than or equal to *attribute*.

Example

The following example returns the smallest integer greater than or equal to 15.7:

```
SELECT CEIL(15.7) "Ceiling" FROM DUAL;
```

```
Ceiling
-----
      16
```

EXP**Syntax**

```
exp::=EXP(attribute)
```

Purpose

EXP returns *e* raised to the *n*th power represented in *attribute*, where *e* = 2.71828183...

Example

The following example returns e to the 4th power:

```
SELECT EXP(4) "e to the 4th power" FROM DUAL;
```

```
e to the 4th power
-----
54.59815
```

FLOOR**Syntax**

```
floor::=FLOOR(attribute)
```

Purpose

FLOOR returns the largest integer equal to or less than the numerical value in attribute.

Example

The following example returns the largest integer equal to or less than 15.7:

```
SELECT FLOOR(15.7) "Floor" FROM DUAL;
```

```
Floor
-----
15
```

LN**Syntax**

```
ln::=LN(attribute)
```

Purpose

LN returns the natural logarithm of attribute, where attribute is greater than 0.

Example

The following example returns the natural logarithm of 95:

```
SELECT LN(95) "Natural Logarithm" FROM DUAL;
```

```
Natural Logarithm
-----
4.55387689
```

LOG**Syntax**

```
log::=LOG(attribute1, attribute2)
```

Purpose

LOG returns the logarithm, base `attribute1` of `attribute2`. The base `attribute1` can be any positive number other than 0 or 1 and `attribute2` can be any positive number.

Example

The following example returns the logarithm of 100:

```
SELECT LOG(10,100) "Log base 10 of 100" FROM DUAL;
```

```
Log base 10 of 100
```

```
-----
```

```
2
```

MOD**Syntax**

```
mod::=MOD(attribute1, attribute2)
```

Purpose

MOD returns the remainder of `attribute1` divided by `attribute2`. It returns `attribute1` if `attribute2` is 0.

Example

The following example returns the remainder of 11 divided by 4:

```
SELECT MOD(11,4) "Modulus" FROM DUAL;
```

```
Modulus
```

```
-----
```

```
3
```

POWER**Syntax**

```
power::=POWER(attribute1, attribute2)
```

Purpose

POWER returns `attribute1` raised to the `n`th power represented in `attribute2`. The base `attribute1` and the exponent in `attribute2` can be any numbers, but if `attribute1` is negative, then `attribute2` must be an integer.

Example

The following example returns three squared:

```
SELECT POWER(3,2) "Raised" FROM DUAL;
```

```
Raised
```

```
-----
```

```
9
```

ROUND (number)

Syntax

```
round_number::=ROUND(attribute1, attribute2)
```

Purpose

ROUND returns attribute1 rounded to attribute2 places right of the decimal point. If attribute2 is omitted, attribute1 is rounded to 0 places. Additionally, attribute2 can be negative to round off digits left of the decimal point and attribute2 must be an integer.

Examples

The following example rounds a number to one decimal point:

```
SELECT ROUND(15.193,1) "Round" FROM DUAL;
```

```
Round
```

```
-----
```

```
15.2
```

The following example rounds a number one digit to the left of the decimal point:

```
SELECT ROUND(15.193,-1) "Round" FROM DUAL;
```

```
Round
```

```
-----
```

```
20
```

SIGN

Syntax

```
sign::=SIGN(attribute)
```

Purpose

If attribute < 0, SIGN returns -1. If attribute = 0, the function returns 0. If attribute > 0, SIGN returns 1. This can be used in validation of measures where only positive numbers are expected.

Example

The following example indicates that the function's argument (-15) is <0:

```
SELECT SIGN(-15) "Sign" FROM DUAL;
```

```
Sign
```

```
-----
```

```
-1
```

SIN

Syntax

```
sin::=SIN(attribute)
```

Purpose

SIN returns the sine of attribute (expressed as an angle)

Example

The following example returns the sine of 30 degrees:

```
SELECT SIN(30 * 3.14159265359/180) "Sine of 30 degrees" FROM DUAL;
```

```
Sine of 30 degrees
-----
                    .5
```

SINH**Syntax**

```
sinh::=SINH(attribute)
```

Purpose

SINH returns the hyperbolic sine of attribute.

Example

The following example returns the hyperbolic sine of 1:

```
SELECT SINH(1) "Hyperbolic Sine of 1" FROM DUAL;
```

```
Hyperbolic Sine of 1
-----
                1.17520119
```

SQRT**Syntax**

```
sqrt::=SQRT(attribute)
```

Purpose

SQRT returns square root of attribute. The value in attribute cannot be negative. SQRT returns a "real" result.

Example

The following example returns the square root of 26:

```
SELECT SQRT(26) "Square root" FROM DUAL;
```

```
Square root
-----
5.09901951
```

TAN**Syntax**

```
tan::=TAN(attribute)
```

Purpose

TAN returns the tangent of attribute (an angle expressed in radians).

Example

The following example returns the tangent of 135 degrees:

```
SELECT TAN(135 * 3.14159265359/180) "Tangent of 135 degrees" FROM DUAL;
```

```
Tangent of 135 degrees
-----
                        -1
```

TANH**Syntax**

```
tanh::=TANH(attribute)
```

Purpose

TANH returns the hyperbolic tangent of *attribute*.

Example

The following example returns the hyperbolic tangent of 5:

```
SELECT TANH(5) "Hyperbolic tangent of 5" FROM DUAL;
```

```
Hyperbolic tangent of 5
-----
                        .462117157
```

TO_CHAR (number)**Syntax**

```
to_char_number::=to_char(attribute, fmt, nlsparam)
```

Purpose

TO_CHAR converts *attribute* of NUMBER data type to a value of VARCHAR2 data type, using the optional number format *fmt*. If you omit *fmt*, *attribute* is converted to a VARCHAR2 value exactly long enough to hold its significant digits. The *nlsparam* specifies these characters that are returned by number format elements:

- Decimal character
- Group separator
- Local currency symbol
- International currency symbol

This argument can have the following form:

```
'NLS_NUMERIC_CHARACTERS = 'dg''
NLS_CURRENCY = 'text'
NLS_ISO_CURRENCY = territory '
```

The characters *d* and *g* represent the decimal character and group separator, respectively. They must be different single-byte characters. Within the quoted string,

you must use two single quotation marks around the parameter values. Ten characters are available for the currency symbol.

If you omit `nlsparam` or any one of the parameters, this function uses the default parameter values for your session.

Examples

In this example, the output is blank padded to the left of the currency symbol.

```
SELECT TO_CHAR(-10000,'L99G999D99MI') "Amount"
FROM DUAL;
Amount
-----
$10,000.00-
SELECT TO_CHAR(-10000,'L99G999D99MI'
'NLS_NUMERIC_CHARACTERS = ','.'
NLS_CURRENCY = 'AusDollars' ') "Amount"
FROM DUAL;

Amount
-----
AusDollars10.000,00-
```

TRUNC (number)

Syntax

```
trunc_number::=TRUNC(attribute, m)
```

Purpose

TRUNC returns `attribute` truncated to `m` decimal places. If `m` is omitted, `attribute` is truncated to 0 places. `m` can be negative to truncate (make zero) `m` digits left of the decimal point.

Example

The following example truncates numbers:

```
SELECT TRUNC(15.79,1) "Truncate"
FROM DUAL;
Truncate
-----
15.7
SELECT TRUNC(15.79,-1) "Truncate"
FROM DUAL;
Truncate
-----
10
```

WB.LOOKUP_NUM (on a number)

Syntax

```
WB.LOOKUP_NUM (table_name
, column_name
, key_column_name
, key_value
)
```

where `TABLE_NAME` is the name of the table to perform the lookup on; `COLUMN_NAME` is the name of the `NUMBER` column that will be returned, for instance, the result of the lookup; `KEY_COLUMN_NAME` is the name of the `NUMBER` column used as the key to match on in the lookup table; `KEY_VALUE` is the value of the key column, for example, the value mapped into the `key_column_name` with which the match will be done.

Purpose

To perform a key look up that returns a `NUMBER` value from a database table using a `NUMBER` column as the matching key.

Example

Consider the following table as a lookup table LKP1:

KEYCOLUMN	TYPE_NO	TYPE
10	100123	Car
20	100124	Bike

Using this package with the following call:

```
WB.LOOKUP_CHAR('LKP1'  
, 'TYPE_NO'  
, 'KEYCOLUMN'  
, 20  
)
```

returns the value of 100124 as output of this transformation. This output is then processed in the mapping as the result of an inline function call.

Note: This function is a row-based key lookup. Set-based lookups are supported when you use the lookup operator.

WB.LOOKUP_NUM (on a varchar2)

Syntax:

```
WB.LOOKUP_CHAR(table_name  
, column_name  
, key_column_name  
, key_value  
)
```

where `TABLE_NAME` is the name of the table to perform the lookup on; `COLUMN_NAME` is the name of the `NUMBER` column that will be returned (such as the result of the lookup); `KEY_COLUMN_NAME` is the name of the `NUMBER` column used as the key to match on in the lookup table; `KEY_VALUE` is the value of the key column, such as the value mapped into the `key_column_name` with which the match will be done.

Purpose:

To perform a key lookup which returns a `NUMBER` value from a database table using a `VARCHAR2` column as the matching key.

Example

Consider the following table as a lookup table LKP1:

KEYCOLUMN	TYPE_NO	TYPE
ACV	100123	Car
ACP	100124	Bike

Using this package with the following call:

```
WB.LOOKUP_CHAR ('LKP1'
, 'TYPE'
, 'KEYCOLUMN'
, 'ACP'
)
```

returns the value of 100124 as output of this transformation. This output is then processed in the mapping as the result of an inline function call.

Note: This function is a row-based key lookup. Set-based lookups are supported when you use the lookup operator described in [Key Lookup](#) on page 2-7.

WB_IS_NUMBER

Syntax

```
WB_IS_NUMBER(attribute, fmt)
```

Purpose

To check whether *attribute* contains a valid number. The function returns a Boolean value, which is set to true if *attribute* contains a valid number. *fmt* is an optional number format. If *fmt* is omitted, the number format of your session is used.

You can use this function when you validate the data before loading it into a table. This way the value can be transformed before it reaches the table and causes an error.

Example

`WB_IS_NUMBER` returns true in PL/SQL if *attribute* contains a valid number.

OLAP Transformations

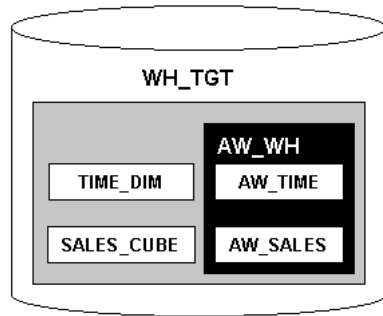
OLAP transformations enable Warehouse Builder users to load data stored in relational dimensions and cubes into an analytic workspace.

The OLAP transformations provided by Warehouse Builder are:

- [WB_OLAP_LOAD_CUBE](#) on page 3-56
- [WB_OLAP_LOAD_DIMENSION](#) on page 3-56
- [WB_OLAP_LOAD_DIMENSION_GENUK](#) on page 3-57

The `WB_OLAP_LOAD_CUBE`, `WB_OLAP_LOAD_DIMENSION`, and `WB_OLAP_LOAD_DIMENSION_GENUK` transformations are used for cube cloning in Warehouse Builder. Use these OLAP transformations only if your database version is Oracle Database 9i or Oracle Database 10g Release 1.

The examples used to explain these OLAP transformations are based on the scenario depicted in [Figure 3-1](#).

Figure 3–1 Example of OLAP Transformations

The relational dimension `TIME_DIM` and the relational cube `SALES_CUBE` are stored in the schema `WH_TGT`. The analytic workspace `AW_WH`, into which the dimension and cube are loaded, is also created in the `WH_TGT` schema.

WB_OLAP_LOAD_CUBE

Syntax

```
wb_olap_load_cube::=WB_OLAP_LOAD_CUBE(olap_aw_owner, olap_aw_name, olap_cube_
owner, olap_cube_name, olap_tgt_cube_name)
```

where `olap_aw_owner` is the name of the database schema that owns the analytic workspace; `olap_aw_name` is the name of the analytic workspace that stores the cube data; `olap_cube_owner` is the name of the database schema that owns the related relational cube; `olap_cube_name` is the name of the relational cube; `olap_tgt_cube_name` is the name of the cube in the analytic workspace.

Purpose

`WB_OLAP_LOAD_CUBE` loads data from the relational cube into the analytic workspace. This allows further analysis of the cube data. This is for loading data in an AW cube from a relational cube which it was cloned from. This is a wrapper around some of the procedures in the `DBMS_AWM` package for loading a cube.

Example

The following example loads data from the relational cube `SALES_CUBE` into a cube called `AW_SALES` in the `AW_WH` analytic workspace:

```
WB_OLAP_LOAD_CUBE('WH_TGT', 'AW_WH', 'WH_TGT', 'SALES_CUBE', 'AW_SALES')
```

WB_OLAP_LOAD_DIMENSION

Syntax

```
wb_olap_load_dimension::=WB_OLAP_LOAD_DIMENSION(olap_aw_owner, olap_aw_name, olap_
dimension_owner, olap_dimension_name, olap_tgt_dimension_name)
```

where `olap_aw_owner` is the name of the database schema that owns the analytic workspace; `olap_aw_name` is the name of the analytic workspace that stores the dimension data; `olap_dimension_owner` is the name of the database schema in which the related relational dimension is stored; `olap_dimension_name` is the name

of the relational dimension; `olap_tgt_dimension_name` is the name of the dimension in the analytic workspace.

Purpose

`WB_OLAP_LOAD_DIMENSION` loads data from the relational dimension into the analytic workspace. This allows further analysis of the dimension data. This is for loading data in an AW dimension from a relational dimension which it was cloned from. This is a wrapper around some of the procedures in the `DBMS_AWM` package for loading a dimension.

Example

The following example loads the data from the relational dimension `TIME_DIM` into a dimension called `AW_TIME` in the analytic workspace `AW_WH`:

```
WB_OLAP_LOAD_DIMENSION('WH_TGT', 'AW_WH', 'WH_TGT', 'TIME_DIM', 'AW_TIME')
```

WB_OLAP_LOAD_DIMENSION_GENUK

Syntax

```
wb_olap_load_dimension_genuk::=WB_OLAP_LOAD_DIMENSION_GENUK(olap_aw_owner, olap_
aw_name, olap_dimension_owner, olap_dimension_name, olap_tgt_dimension_name)
```

where `olap_aw_owner` is the name of the database schema that owns the analytic workspace; `olap_aw_name` is the name of the analytic workspace that stores the dimension data; `olap_dimension_owner` is the name of the database schema in which the related relational dimension is stored; `olap_dimension_name` is the name of the relational dimension; `olap_tgt_dimension_name` is the name of the dimension in the analytic workspace.

Purpose

`WB_OLAP_LOAD_DIMENSION_GENUK` loads data from the relational dimension into the analytic workspace. Unique dimension identifiers will be generated across all levels. This is for loading data in an AW dimension from a relational dimension which it was cloned from. This is a wrapper around some of the procedures in the `DBMS_AWM` package for loading a dimension.

If a cube has been cloned and if you select YES for the Generate Surrogate Keys for Dimensions option, then when you want to reload the dimensions, you should use the `WB_OLAP_LOAD_DIMENSION_GENUK` procedure. This procedure generates surrogate identifiers for all levels in the AW, because the AW requires all level identifiers to be unique across all levels of a dimension.

Example

Consider an example in which the dimension `TIME_DIM` has been deployed to the OLAP server by cloning the cube. The parameter generate surrogate keys for Dimension was set to true. To now reload data from the relational dimension `TIME_DIM` into the dimension `AW_TIME` in the analytic workspace `AW_WH`, use the following syntax.

```
WB_OLAP_LOAD_CUBE('WH_TGT', 'AW_WH', 'WH_TGT', 'TIME_DIM', 'AW_TIME')
```

XML Transformations

XML transformations provide Warehouse Builder users with functionality to perform transformations on XML objects. These transformations enable Warehouse Builder users to load and transform XML documents and Oracle AQs.

To enable loading of XML sources, Warehouse Builder provides access to the database XML functionality through custom functions, as detailed in this chapter.

WB_XML_LOAD

Syntax:

```
WB_XML_LOAD(control_file)
```

Purpose

WB_XML_LOAD extracts and loads data from XML documents into database targets. The `control_file`, an XML document, specifies the source of the XML documents, the targets, and any runtime controls. After the transformation has been defined, a mapping in Warehouse Builder calls the transformation as a pre-map or post-map trigger.

Example

The following example illustrates a script that can be used to implement a Warehouse Builder transformation that extracts data from an XML document stored in the file `products.xml` and loads it into the target table called `books`.

```
begin
wb_xml_load('<OWBXMLRuntime>'
||
'<XMLSource>'
||
' <file>\ora817\GCCAPPS\products.xml</file>'
||
'</XMLSource>'
||
'<targets>'
||
' <target XSLFile="\ora817\XMLstyle\GCC.xsl">books</target>'
||
'</targets>'
||
'</OWBXMLRuntime>'
);
end;
```

For more information on control files, see the *Oracle Warehouse Builder User's Guide*.

WB_XML_LOAD_F

Syntax

```
WB_XML_LOAD_F(control_file)
```

Purpose

WB_XML_LOAD_F extracts and loads data from XML documents into database targets. The function returns the number of XML documents read during the load. The

`control_file`, itself an XML document, specifies the source of the XML documents, the targets, and any runtime controls. After the transformation has been defined, a mapping in Warehouse Builder calls the transformation as a pre-map or post-map trigger.

Example

The following example illustrates a script that can be used to implement a Warehouse Builder transformation that extracts data from an XML document stored in the file `products.xml` and loads it into the target table `books`.

```
begin
wb_xml_load_f('<OWBXMLRuntime>'
|
|
| '<XMLSource>'
|
| ' <file>\ora817\GCCAPPS\products.xml</file>'
|
| '</XMLSource>'
|
| '<targets>'
|
| ' <target XSLFile="\ora817\XMLstyle\GCC.xsl">books</target>'
|
| '</targets>'
|
| '</OWBXMLRuntime>'
);
end;
```

For more information on the types handled and detailed information on `control_` files, see the *Oracle Warehouse Builder User's Guide*.

Conversion Transformations

The conversion transformations enable Warehouse Builder users to perform functions that allow conditional conversion of values. These functions achieve "if - then" constructions within SQL. For example, `NVL` provides functionality that substitutes NULL values with any value specified, or if `input = NULL` then `output = value`.

CASE

CASE expressions enable you to use "IF...THEN...ELSE" logic in SQL statements without invoking procedures. Use this statement instead of `decode`.

Syntax

```
case_expression::=CASE attributel WHEN inputvalue THEN outputvalue
[WHEN inputvalue THEN outputvalue]...
ELSE elsevalue
END
```

Purpose

In a simple CASE expression, Oracle searches for the first `WHEN . . . THEN` pair for which `attributel` is equal to `inputvalue` and returns `outputvalue`. If none of the `WHEN . . . THEN` pairs meet this condition, and an `ELSE` clause exists, then Oracle returns `elsevalue`. Otherwise, Warehouse Builder returns `null`.

All of the expressions (attribute1, inputvalue, and outputvalue) must be of the same data type, which can be CHAR or VARCHAR2.

Simple CASE Example

For each customer in the sample oe.customers table, the following statement lists the credit limit as "Low" if it equals \$100, "High" if it equals \$5000, and "Medium" if it equals anything else.

```
SELECT cust_last_name,
       CASE credit_limit WHEN 100 THEN 'Low'
                       WHEN 5000 THEN 'High'
                       ELSE 'Medium' END
FROM customers;
```

CUST_LAST_NAME	CASECR
...	
Bogart	Medium
Nolte	Medium
Loren	Medium
Gueney	Medium

Searched CASE Example

The following statement finds the average salary of the employees in the sample table oe.employees, using \$2000 as the lowest salary possible:

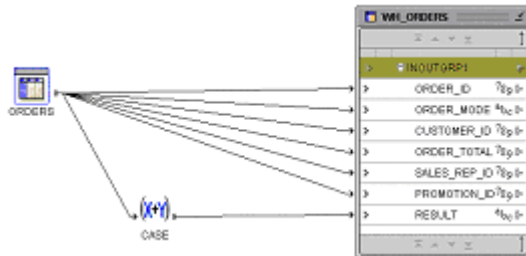
```
SELECT AVG(CASE WHEN e.salary > 2000 THEN e.salary
              ELSE 2000 END) "Average Salary" from employees e;
```

Average Salary
6461.68224

Warehouse Builder Example

In Warehouse Builder, you can use an expression to hold the CASE statement. The Expression Builder enables you to create the statement that is incorporated in the generated code. This example is shown in [Figure 3-2, "CASE Mapping Example"](#).

Figure 3-2 CASE Mapping Example



NVL

Syntax

```
nvl::=NVL(attribute1, attribute2)
```

Purpose

If `attribute1` is null, NVL returns `attribute2`. If `attribute1` is not null, then NVL returns `attribute1`. The arguments `attribute1` and `attribute2` can be any data type. If their data types are different, `expr2` is converted to the data type of `expr1` before they are compared. Warehouse Builder provides three variants of NVL to support all input values.

The data type of the return value is always the same as the data type of `attribute1`, unless `attribute1` is character data, in which case the return value data type is VARCHAR2, in the character set of `attribute1`.

Example

The following example returns a list of employee names and commissions, substituting "Not Applicable" if the employee receives no commission:

```
SELECT last_name, NVL(TO_CHAR(commission_pct), 'Not Applicable') "COMMISSION"
FROM employees
WHERE last_name LIKE 'B%';
```

LAST_NAME	COMMISSION
-----	-----
Baer	Not Applicable
Baida	Not Applicable
Banda	.11
Bates	.16
Bell	Not Applicable
Bernstein	.26
Bissot	Not Applicable
Bloom	.21
Bull	Not Applicable

Other Transformations

Other transformations included with Warehouse Builder enable you to perform various functions which are not restricted to certain data types. This section describes those types.

NLS_CHARSET_DECL_LEN**Syntax**

```
nls_charset_decl_len::=NLS_CHARSET_DECL_LEN(byte_count,charset_id)
```

Purpose

NLS_CHARSET_DECL_LEN returns the declaration width (in number of characters) of an NCHAR column. The `byte_count` argument is the width of the column. The `charset_id` argument is the character set ID of the column.

Example

The following example returns the number of characters that are in a 200-byte column when you are using a multibyte character set:

```
SELECT NLS_CHARSET_DECL_LEN(200, nls_charset_id('ja16eucfixed')) FROM DUAL;

NLS_CHARSET_DECL_LEN(200,NLS_CHARSET_ID('JA16EUCFIXED'))
-----
```

NLS_CHARSET_ID

Syntax

```
nls_charset_id ::= NLS_CHARSET_ID(text)
```

Purpose

NLS_CHARSET_ID returns the character set ID number corresponding to character set name *text*. The *text* argument is a run-time VARCHAR2 value. The *text* value 'CHAR_CS' returns the database character set ID number of the server. The *text* value 'NCHAR_CS' returns the national character set ID number of the server.

Invalid character set names return null.

Example

The following example returns the character set ID number of a character set:

```
SELECT NLS_CHARSET_ID('ja16euc') FROM DUAL;

NLS_CHARSET_ID('JA16EUC')
-----
                        830
```

NLS_CHARSET_NAME

Syntax

```
nls_charset_name ::= NLS_CHARSET_NAME(number)
```

Purpose

NLS_CHARSET_NAME returns the name of the character set corresponding to ID *number*. The character set name is returned as a VARCHAR2 value in the database character set.

If *number* is not recognized as a valid character set ID, then this function returns null.

Example

The following example returns the character set corresponding to character set ID number 2:

```
SELECT NLS_CHARSET_NAME(2) FROM DUAL;

NLS_CH
-----
WE8DEC
```

UID

Syntax

```
uid ::= UID()
```

Purpose

UID returns an integer that uniquely identifies the session user, such as the user who is logged on when running the session containing the transformation. In a distributed SQL statement, the UID function identifies the user on your local database.

Use this function when logging audit information into a target table to identify the user running the mappings.

Example

The following returns the local database user id logged into this session:

```
select uid from dual;
```

```

      UID
-----
      55

```

USER**Syntax**

```
user::=USER()
```

Purpose

USER returns the name of the session user (the user who logged on) with the data type VARCHAR2.

Oracle compares values of this function with blank-padded comparison semantics. In a distributed SQL statement, the UID and USER functions identify the user on your local database.

Use this function when logging audit information into a target table to identify the user running the mappings.

Example

The following example returns the local database user logged into this session:

```
select user from dual;
```

```

      USER
-----
OWB9I_RUN

```

Using Slowly Changing Dimensions

This appendix provides a brief introduction to the different types of Slowly Changing Dimensions. It also goes through a case study scenario to demonstrate how to use Warehouse Builder to design and deploy different types of Slowly Changing Dimensions. For additional information, refer to books that discuss data warehousing such as *The Data Warehouse Toolkit* by Ralph Kimball.

Note: Warehouse Builder only supports Slowly Changing Dimensions with Oracle9i Release 2 or later database servers.

This appendix contains the following topics:

- [About Slowly Changing Dimensions](#) on page A-1
- [Case Study Scenario](#) on page A-2
- [Using Type 1 Slowly Changing Dimensions](#) on page A-4
- [Using Type 2 Slowly Changing Dimensions](#) on page A-6
- [Using Type 3 Slowly Changing Dimension](#) on page A-13
- [Deploying and Loading Slowly Changing Dimensions](#) on page A-16

About Slowly Changing Dimensions

A Slowly Changing Dimension (SCD) is a well-defined strategy to manage both current and historical data over time in a data warehouse. You must first decide which type of slowly changing dimension to use based on your business requirements. [Table A-1](#) describes the three main types of SCDs.

Table A-1 *Types of Slowly Changing Dimensions*

Type	Use	Description	Preserves History?
Type 1	Overwriting	Only one version of the dimension record exists. When a change is made, the record is overwritten and no historic data is stored.	No
Type 2	Creating Another Dimension Record	There are multiple versions of the same dimension record, and new versions are created while old versions are still kept upon modification.	Yes

Table A-1 (Cont.) Types of Slowly Changing Dimensions

Type	Use	Description	Preserves History?
Type 3	Creating a Current Value Field	There are two versions of the same dimension record: old values and current values, and old values are kept upon modification on current values.	Yes

After selecting the type of SCD, proceed with the following steps to create the dimensions:

- Create new dimensions that store historic data.
- Create mappings that extract, transform, and load data from the source system to the pre-defined dimension target.
- Generate and deploy both dimensions and mappings to an Oracle9i Release 2 or later database.
- Execute the mappings.

Case Study Scenario

In order to use slowly changing dimensions, you must be using the following:

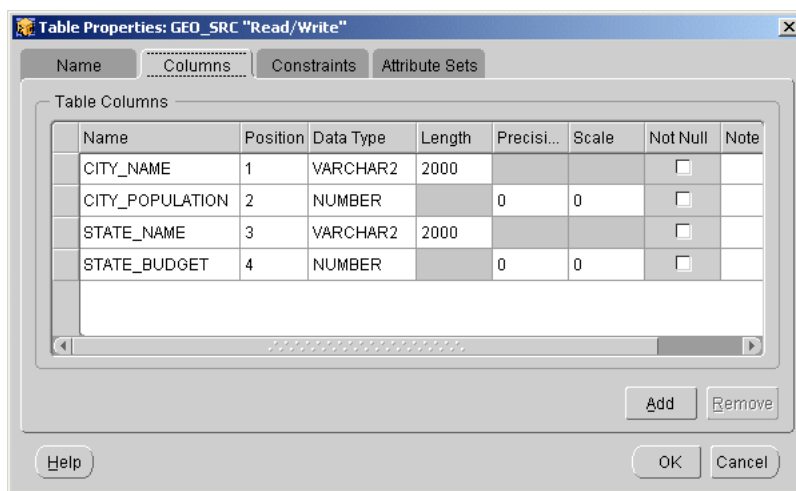
- Warehouse Builder 9.0.4 or later
- Oracle9i Release 2 Database or later

In this appendix, we will be demonstrating how to construct SCDs with the source and target systems described in the following sections. We will be using star schema to store data for all levels on the same dimension table target. This is one of the most commonly used strategy.

Source System

The geography data source table GEO_SRC will be used in our case study. [Figure A-1](#) shows the attributes of the GEO_SRC table.

Figure A-1 GEO_SRC Table Properties



Target System

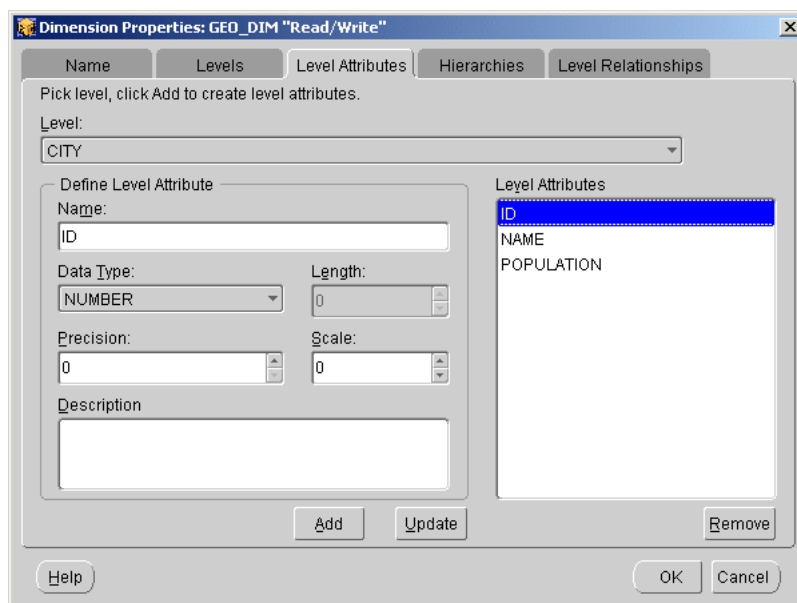
The target warehouse that will be created includes the following:

- The sequence DIM_ID that will be used to populate surrogate keys
- The dimension GEO_DIM that will be used as a Type 1 SCD
- A mapping to load data from GEO_SRC to GEO_DIM
- The dimension GEO_DIM_TYPE2 that will be used as a Type 2 SCD
- A mapping to load data from GEO_SRC to GEO_DIM
- The dimension GEO_DIM_TYPE3 that will be used as a Type 3 SCD
- A mapping to load data from GEO_SRC to GEO_DIM_TYPE3

A geography dimension will be used as our case study for illustration. Typically a geography dimension has two levels: city and state. A city level is the lowest level among the geography hierarchy, while a state level is the higher level. A simplified city level, shown in [Figure A-2](#), has the following attributes:

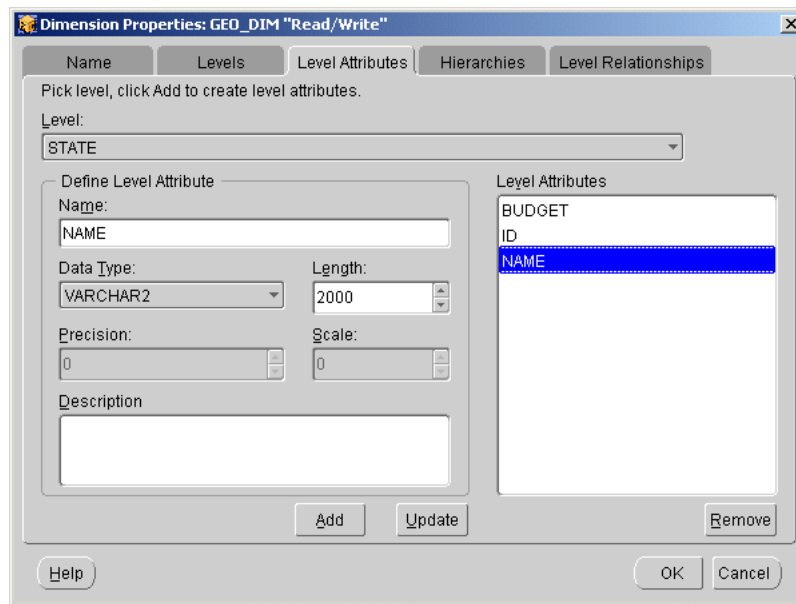
- **ID:** The surrogate key for city level.
- **NAME:** The natural key for city level.
- **POPULATION:** The population of the city.

Figure A-2 City Level Dimension Properties



A simplified state level, as shown in [Figure A-3](#), has the following attributes:

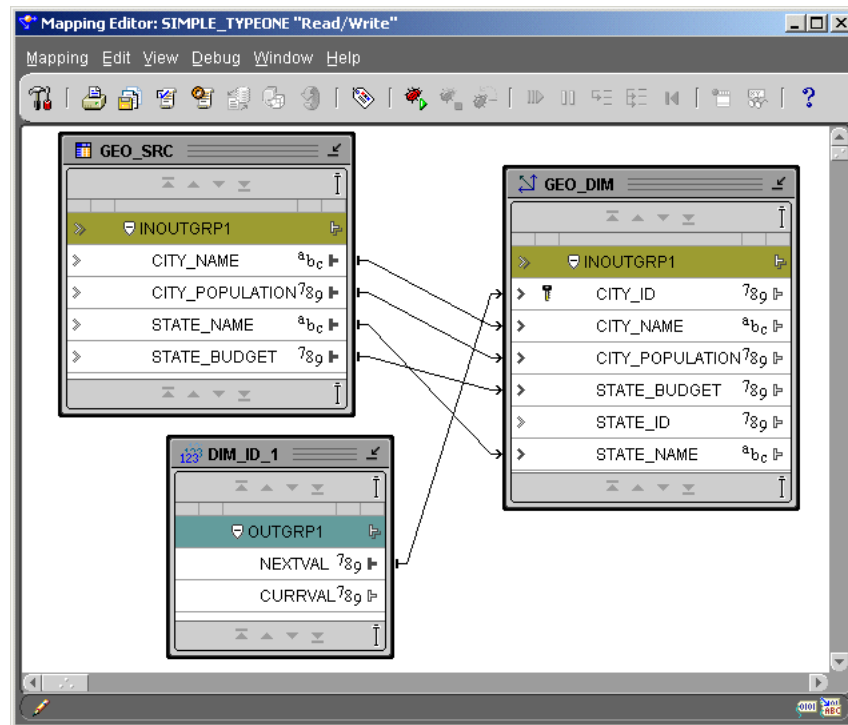
- **ID:** The surrogate key for state level.
- **NAME:** The natural key for state level.
- **BUDGET:** The budget of the state.

Figure A-3 State Level Dimension Properties

Using Type 1 Slowly Changing Dimensions

With Type 1 SCDs, you keep no history and only store the latest value of the dimension record. Once you define the dimension GEO_DIM, you can use it in your mapping to load data into it. To load Type1 slowly changing dimensions, you extract data from the source and then directly load them into the target. GEO_SRC is the source table from which data will be loaded into the dimension GEO_DIM. [Figure A-4](#) shows the mapping used in this example.

Figure A-4 Type 1 SCD Mapping



Use the following steps to finish creating the Type 1 SCD:

- [Step 1: Populate the Surrogate Key](#)
- [Step 2: Configure the Target Properties](#)
- [Step 3: Generate Code](#)

Step 1: Populate the Surrogate Key

To ensure that unique numbers are assigned for surrogate keys for new dimension records, a sequence operator is used to map to the surrogate key column of GEO_DIM, which is CITY_ID (lowest level key).

Step 2: Configure the Target Properties

You should configure the properties for GEO_DIM operator, as shown in [Figure A-5](#), to ensure data loads properly. First, you need to configure the loading type of GEO_DIM to be 'UPDATE/INSERT'.

Figure A-5 Mapping Dimension Properties

GEO_DIM	
Bound Name	GEO_DIM
Primary Source	No
Loading Type	UPDATE/INSERT
Conditional Loading	
Target Filter for Update	
Target Filter for Delete	
Match by constraint	No constraints
Keys (read-only)	
GEO_DIM_CITY_UK	
Key Name	GEO_DIM_CITY_UK
Key Columns	INOUTGRP1.CITY_ID
Key Type	UNIQUE
Referenced Keys	

You also need to configure each mapped column.

- CITY_ID is the surrogate key and is to be loaded only when inserting rows.
- CITY_NAME is the natural key and is to be loaded only when inserting rows. It is also to be matched when updating rows.
- CITY_POPULATION is to be loaded both when inserting and updating rows. STATE_NAME and STATE_BUDGET are configured in the same way.

Step 3: Generate Code

If your target database type, which is configurable from warehouse module configuration properties, is set to Oracle9i, the MERGE feature is ensured for you when you generate code.

Using Type 2 Slowly Changing Dimensions

With Type 2 SCD, you always create another version of dimension record and mark the existing version as history. To accommodate this, you need to create extra metadata for your dimension table, including an effective date column and an expiration date column. These columns are used to differentiate a current version from a historical version as follows:

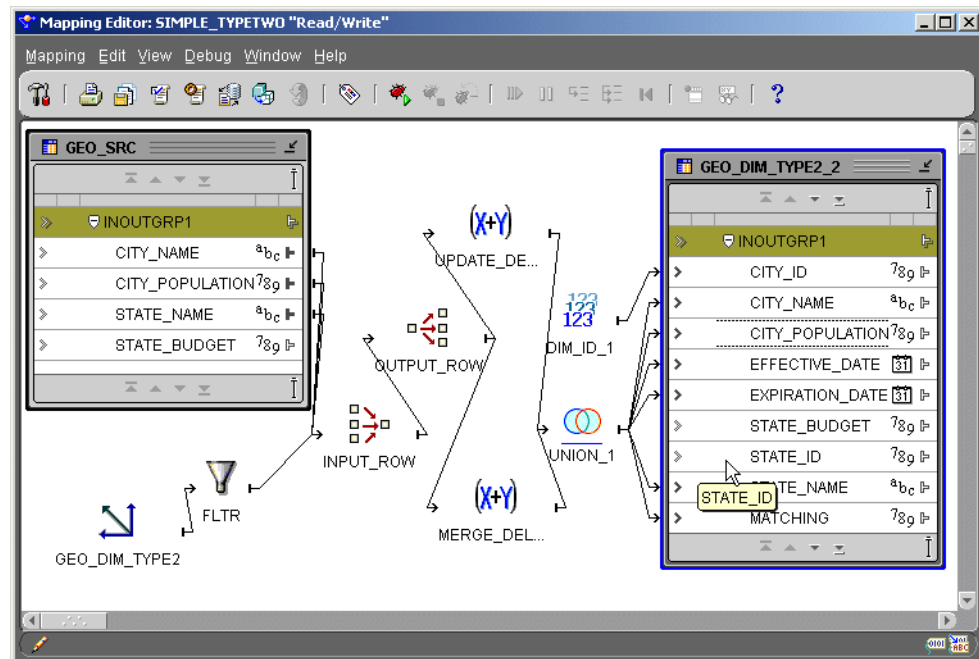
- Effective date column stores the effective date of the version; also known as start date
- Expiration date column stores the expiration date of the version; also known as end date
- Expiration date value of the current version is always set to NULL or a default date value

You also need to decide which columns you want to store historic data for when the values are to be changed. These columns are defined as trigger columns and should be described as part of your metadata.

Once you define your dimension GEO_DIM_TYPE2, you can use it in your mapping to load data into it. GEO_SRC is the sample source table here from which data are to be loaded into GEO_DIM_TYPE2.

To load Type 2 slowly changing dimension, you need to transform data extracted from the source properly before you load them into the target. You achieve this by creating a mapping, such as the one displayed in Figure A-6. In this mapping, data is first extracted from GEO_SRC, transformed by a series of operators, and finally loaded into GEO_DIM_TYPE2.

Figure A-6 Type 2 SCD Mapping



You must be very curious about how data are actually transformed. Warehouse Builder supports all operators you would need for Type 2 slowly changing dimension. With Warehouse Builder, the whole ETL process of Type 2 slowly changing dimension can be done in one single mapping. Let us take a look at how data are transformed in a step-by-step fashion.

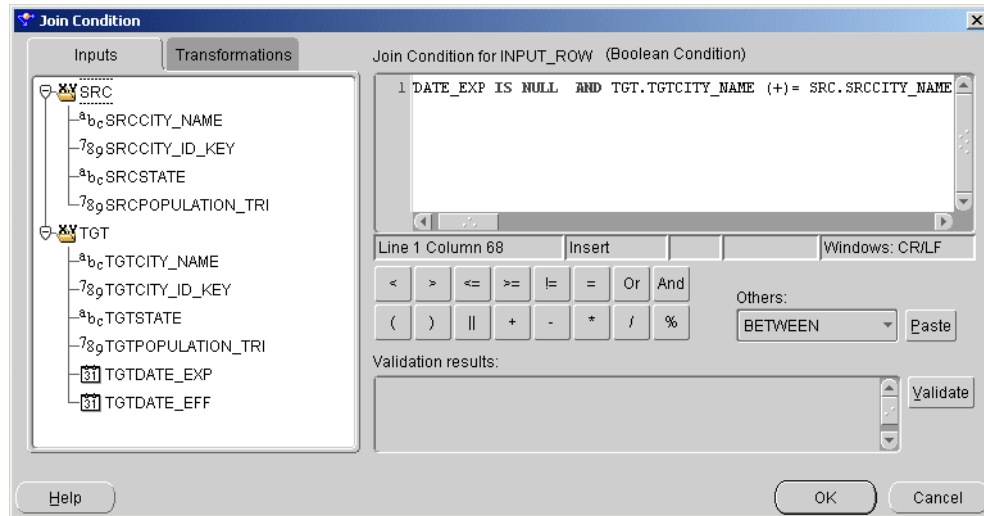
Use the following steps to create a Type 2 SCD:

- [Step 1: Detect a Match](#)
- [Step 2: Split Join Results](#)
- [Step 3: Determine Merge Rows](#)
- [Step 4: Use the Expression UPDATE_DELTA_ROW](#)
- [Step 5: Use the Expression MERGE_DELTA_ROW](#)
- [Step 6: Populate Surrogate Keys](#)
- [Step 7: Configure Target Properties](#)
- [Step 8: Generate Code](#)

Step 1: Detect a Match

First of all, for each source row from GEO_SRC, you need to figure out if it has matched a current dimension record in GEO_DIM_TYPE2. To do this, a Joiner is used to match GEO_SRC with GEO_DIM_TYPE2 exclusively using outer join by natural key columns as the join condition. Figure A-7 shows the expression used for this condition.

Figure A-7 Input_Row Expression



Also notice that GEO_SRC should only match current dimension records in GEO_DIM_TYPE2, rather than history dimension records. To do this, you apply a filter operator to filter out history records from matching.

Step 2: Split Join Results

After Joiner, the output data are now composing both the source data rows and the matched target rows. For each output row of Joiner, you need to categorize it into the following groups:

- OPEN_SET is defined to create a new version or overwrite a current version
- CLOSE_SET is defined to mark a current version as historical

Do this categorizing by splitting the Joiner output into OPEN_SET and CLOSE_SET groups using a Splitter.

A Joiner output row will be put into OPEN_SET group if it comes from a row in GEO_SRC that is either matching with any current version in GEO_DIM_TYPE2, or matching with no version. Do this by specifying the splitter condition for OPEN_SET group.

A Joiner output row will be put into CLOSE_SET group if both the following two condition are true:

- If it comes from a row in GEO_SRC that is matching with any current version in GEO_DIM_TYPE2, and
- If any trigger column from GEO_DIM_TYPE2 does not equal to that from GEO_SRC

Specify the splitter condition for CLOSE_SET group to AND the earlier two condition clauses.

Step 3: Determine Merge Rows

With OPEN_SET and CLOSE_SET, you compute the following two delta sets with which GEO_DIM_TYPE2 is to be loaded:

- From CLOSE_SET to update GEO_DIM_TYPE2; also known as UPDATE_DELTA_ROW
- From OPEN_SET to update/insert GEO_DIM_TYPE2; also known as MERGE_DELTA_ROW

You use Expressions to accomplish both tasks. UPDATE_DELTA_ROW and MERGE_DELTA_ROW are created as two separate Expression operators from output of CLOSE_SET and OPEN_SET, respectively. The output groups of both Expression operators are then UNION by utilizing a SetOp operator, whose output row set is ready to be mapped to GEO_DIM_TYPE2 directly.

Step 4: Use the Expression UPDATE_DELTA_ROW

UPDATE_DELTA_ROW represents the row set that the final target row is to be overwritten from in order to mark a current matched version as historical. Specifically, the target expiration timestamp need be updated with current system date value. This operation is also known as to close the current version. To accomplish this, you specify the expression of attribute DATE_EXP to be SYSDATE.

For the rest of the columns, you do not need to update them such that the original target column values are specified for the corresponding expressions.

Step 5: Use the Expression MERGE_DELTA_ROW

MERGE_DELTA_ROW represents the row set that the final target row is to be overwritten from in order to:

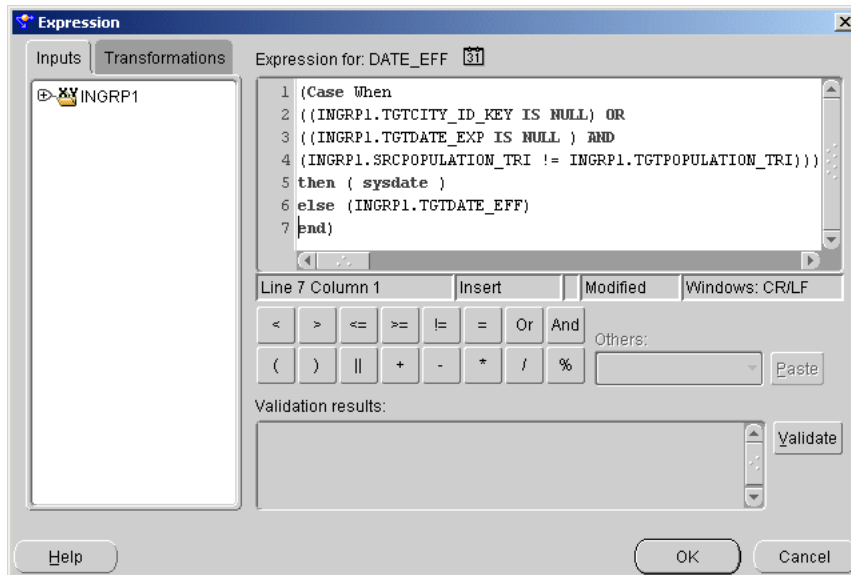
- Create another current version if there is either no current version matched in target, or the matched version has a different value in any of the trigger columns.
- Otherwise, update the matched version directly.

Specifically, you need to build the expression for each final target column to differentiate between the earlier two scenarios by instantiating a CASE expression, that is, 'Case When (...) Then (...) Else (...) End'. Fortunately, Warehouse Builder supports a user-friendly expression builder to accomplish this easily.

For DATE_EFF or any effective timestamp column, you specify the expression to:

- Either preserve the current system time (or SYSDATE) if it is to create another version,
- Or otherwise, preserve the effective timestamp value derived from target (that is, it is to update the matched version)

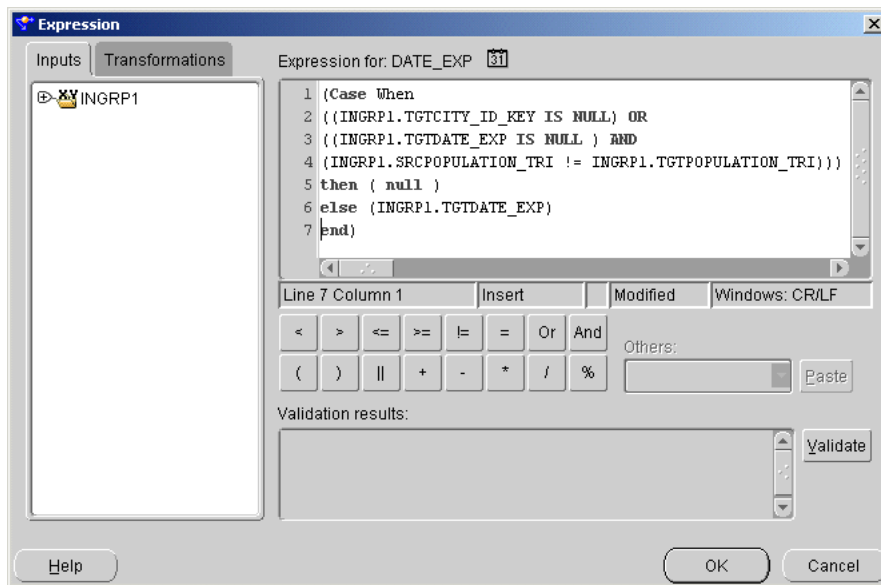
Figure A-8 shows an example of how you specify the expression for DATE_EFF or any effective timestamp column.

Figure A-8 Expression for DATE_EFF

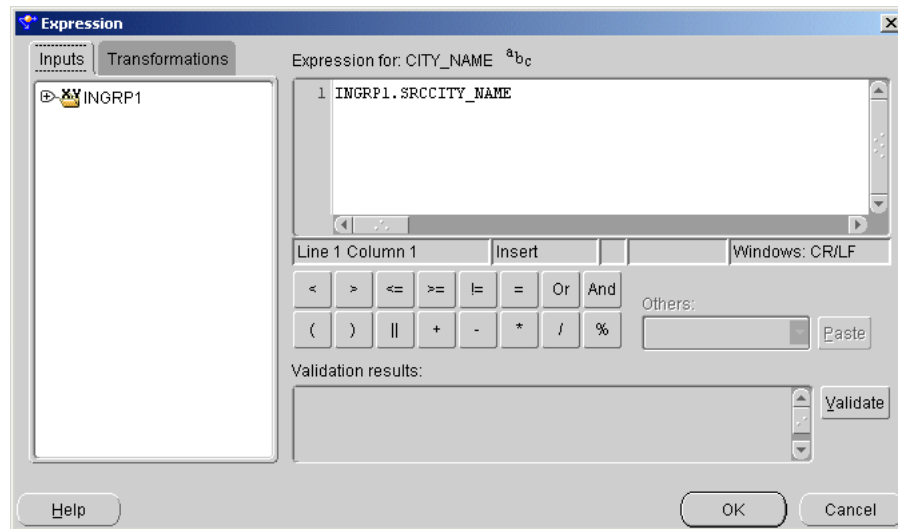
For DATE_EXP or any expiration timestamp column, you specify the expression to:

- Either preserve a default value (such as NULL or some future timestamp 01/01/2004) to mark any version as the current if it is to create another version,
- Or otherwise, preserve the expiration timestamp value derived from target (that is, it is to update the matched version)

Figure A-9 shows an example of how you specify the expression for DATE_EXP or any expiration timestamp column.

Figure A-9 Expression for DATE_EXP

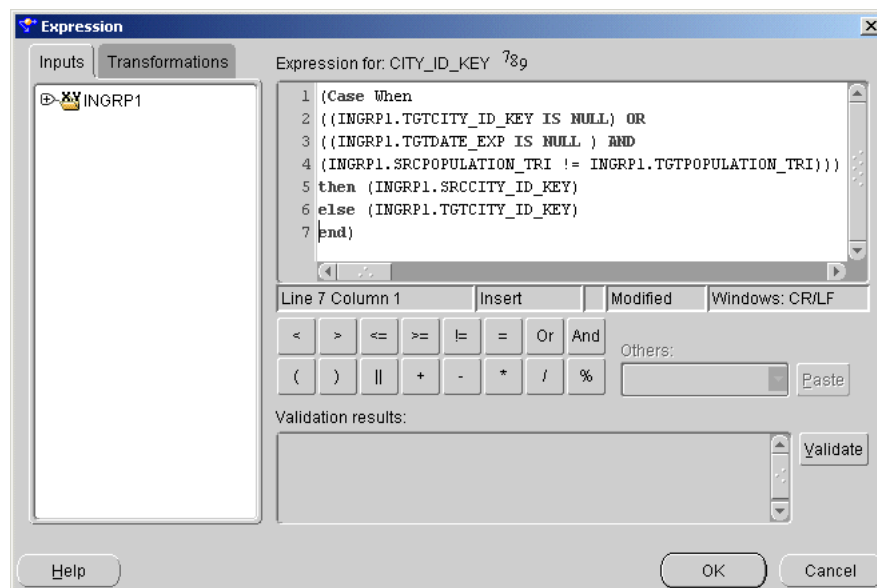
For CITY_NAME or any natural key column, you always overwrite with natural key value derived from source. Figure A-10 shows an example of how you specify the expression for CITY_NAME or any natural key column.

Figure A-10 Expression for CITY_NAME

For CITY_ID_KEY or any surrogate key column, you need to preserve the surrogate key value derived from target in order to:

- Match with the final target row to perform updating if it were to update the matched version.
- The derived target surrogate key would be NULL if it were to create a new version; a sequence number would be introduced later to ensure a unique surrogate key value is assigned for creating a dimension record.

Figure A-11 shows an example of how you specify the expression for CITY_ID_KEY or any surrogate key column.

Figure A-11 Expression for CITY_ID_KEY

For STATE_NAME or any non-trigger column, you always overwrite with the value derived from source.

For CITY_POPULATION or any trigger column, you always overwrite with the value derived from source.

Step 6: Populate Surrogate Keys

To ensure that unique numbers are assigned as surrogate keys for new dimension records, a sequence operator is used to insert the surrogate key column of GEO_DIM_TYPE2, which is CITY_ID.

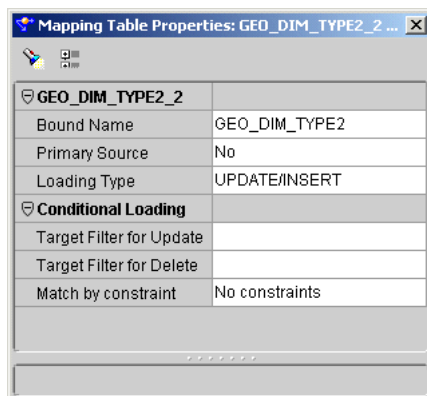
The derived target surrogate key from UNION would be used to match with the final target surrogate key during loading. To achieve this, you create an additional attribute MATCHING for the final target and then map from the derived target surrogate key CITY_ID_KEY to it.

MATCHING attribute stands for the unique key of the final target that is chosen to be the matching criteria to ensure data loads properly. Here you should use the final target surrogate key column CITY_ID as MATCHING attribute. You achieve this by setting the bound name to be the same as CITY_ID:

Step 7: Configure Target Properties

Figure A-12 shows an example of how you can configure the properties for GEO_DIM_TYPE2 operator to ensure that data loads properly. First of all, you need to configure the loading type of GEO_DIM_TYPE2 to be UPDATE/INSERT.

Figure A-12 Configuration using UPDATE/INSERT



You also need to configure each mapped column.

- CITY_ID is the surrogate key and is to be loaded only when inserting rows.
- CITY_NAME is the natural key and is to be loaded only when inserting rows.
- CITY_POPULATION is to be loaded both when inserting and updating rows. STATE_NAME, EFFECTIVE_DATE and EXPIRATION_DATE are configured in the same way.
- MATCHING is to be matched when updating rows.

Step 8: Generate Code

If your target database type (configurable from warehouse module configuration properties) is set to Oracle9i, the MERGE feature is ensured for you when you generate code.

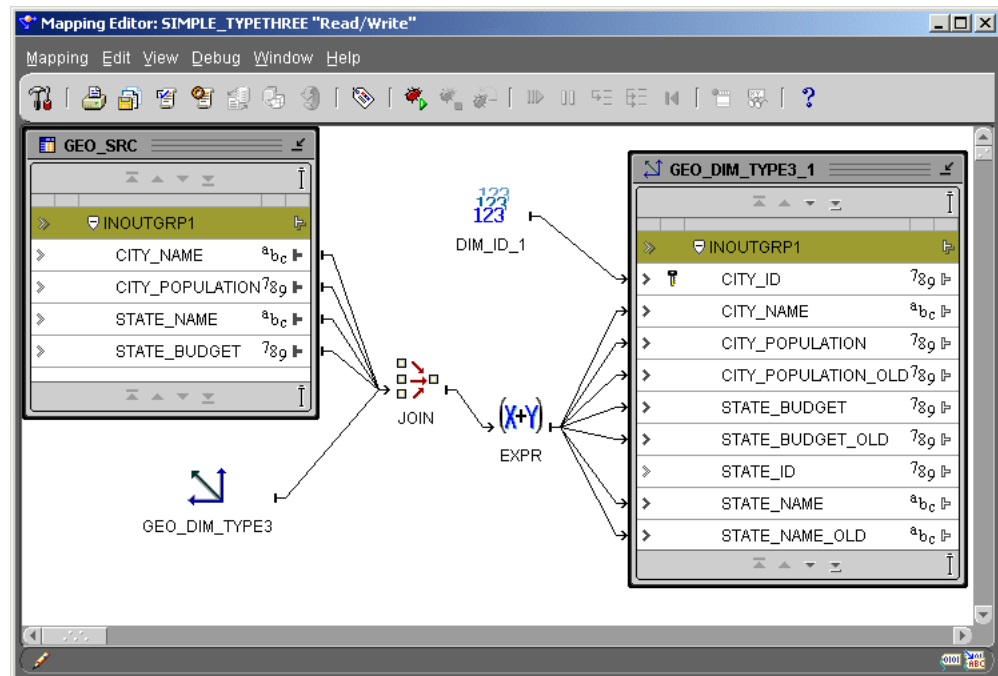
Using Type 3 Slowly Changing Dimension

With Type 3 SCD, you create a current value field to keep the current value of dimension record apart from its previous value. To achieve this, you need to create two columns for each data field, one for current value and the other for keeping previous value, respectively.

Once you define your dimension GEO_DIM_TYPE3, you can use it in your mapping to load data into it. GEO_SRC is the sample source table here from which data are to be loaded into GEO_DIM_TYPE3.

To load Type 3 slowly changing dimension, you extract data from the source and then transform them before directly load them into the target. You achieve this by the following mapping graph where data are first extracted from GEO_SRC, transformed by a series of operators, and finally loaded into GEO_DIM_TYPE3. Figure A-13 shows an example of this mapping.

Figure A-13 Type 3 SCD Mapping



Use the following steps to create a Type 3 SCD:

- Step 1: Detect a Match
- Step 2: Populate Current Values
- Step 3: Populate Previous Value Columns by Expression
- Step 4: Populate Surrogate Keys
- Step 5: Configure Target Properties
- Step 6: Generate Code

Step 1: Detect a Match

First of all, for each source row from GEO_SRC, you need to figure out if it has matched a current dimension record in GEO_DIM_TYPE3. To do this, a Joiner is used

to match GEO_SRC with GEO_DIM_TYPE3 exclusively (using outer join) by natural key columns as the join condition.

Step 2: Populate Current Values

For Type 3 SCD, you always overwrite current value columns of the target with that of the source. You accomplish this by creating mapping lines from Joiner output directly into the target, GEO_DIM_TYPE3.

Step 3: Populate Previous Value Columns by Expression

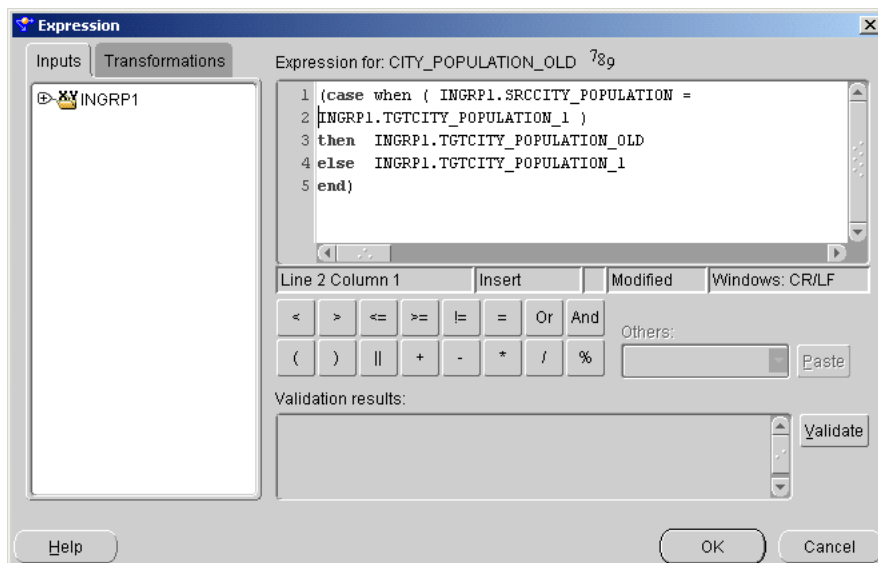
For Type 3 SCD, it matters to you when and how to overwrite previous value columns of the target, including CITY_POPULATION_OLD, CITY_STATE_BUDGET_OLD, and CITY_STATE_NAME_OLD.

Specifically you need to:

- Overwrite previous value column with current value column of the target, if current value of the target is different from that of the source; or
- Otherwise, no change is required for the previous value column.

To achieve this, build an Expression from the previous Joiner result and instantiate the expression using a CASE expression. [Figure A-14](#) shows an example of how to instantiate the expression using a CASE expression.

Figure A-14 Case Expression



Step 4: Populate Surrogate Keys

This is similar to what you have done for Type 1 SCDs. Refer to ["Step 1: Populate the Surrogate Key"](#) on page A-5 for details.

Step 5: Configure Target Properties

This is also similar to what you have done for Type 1 SCDs.

- You need to configure the loading type of GEO_DIM_TYPE3 to be UPDATE/INSERT.

- CITY_ID is the surrogate key and is to be loaded only when inserting rows.
- CITY_NAME is the natural key and is to be loaded only when inserting rows. It is also to be matched when updating rows.
- Others are to be loaded both when inserting and updating rows.

Step 6: Generate Code

If your target database type (configurable from warehouse module configuration properties) is set to Oracle9i, the MERGE feature is ensured for you when you generate code.

Deploying and Loading Slowly Changing Dimensions

Once you have constructed dimensions and mappings, you proceed to deploy and execute them through Deployment Manager. Run mappings using set-based mode with an Oracle9i database as your target to ensure optimal performance.

Index

A

ABS, 3-45
accessibility in Oracle documentation, ix
accessing
 transformation libraries, 3-2
ACOS function, 3-45
ADD_MONTHS, 3-29
administrative transformations, 3-3
aggregator operator, 2-20
Americans with Disabilities Act (ADA)
 compliance, ix
ASCII, 3-12
ASCIISTR, 3-12
ASIN function, 3-45
ATAN function, 3-46
ATAN2 function, 3-46
AVG, 2-22

B

books
 accessibility, ix

C

CEIL, 3-47
character transformations, 3-12
CHARTOROWID, 3-13
CHR, 3-13
CONCAT, 3-14
constant operator, 2-28
contacting Oracle
conventions
 used in this guide, xvii
CONVERT, 3-14
COS function, 3-46
COSH function, 3-47
COUNT, 2-22
CURRVAL, 2-10
 Sequence operator, 2-10

D

data cleansing operators, 2-28
data quality
 Match-Merge operator, 2-30

data transformation, 3-1
datatypes
 ZONED, xii
Date Transformations, 3-29
date transformations, 3-29
Deduplicator operator, 2-1
disabilities, documentation accessibility, ix
DISTINCT, 2-1
documentation
 ordering, for Warehouse Builder

E

EXP, 3-47

F

features
 new in this release
 new in this release *See also Oracle Warehouse Builder Release Notes*
Filter operator, 2-2
FLOOR, 3-48
FULL OUTER JOIN, 2-4
functions
 Global Shared Library, 3-2

G

geocoding, 2-29
Global Shared Library, 3-2
GROUP BY, 2-20
guides
 accessibility, ix
 available for Warehouse Builder, xvii

H

HAVING, 2-20
householding, 2-30
 Match-Merge operator, 2-30

I

importing
 PL/SQL Packages, 3-3

INITCAP, 3-15
installing Warehouse Builder *See Oracle Warehouse
Builder Installation and Configuration Guide*
INSTR / INSTRB, 3-15
INTERSECT, 2-12

J

Joiner operator, 2-4

K

key
 surrogate, 2-10
Key Lookup operator, 2-7

L

LAST_DAY, 3-30
LENGTH/LENGTHB, 3-16
LN function, 3-48
LOG function, 3-48
LOWER, 3-17
LPAD, 3-17
LTRIM, 3-18

M

manuals
 accessibility, ix
 available for Warehouse Builder, xvii
mapping operators
 Match-Merge operator, 2-30
 Pivot operator, 2-8
 Unpivot operator, 2-19
Match-Merge operator, 2-30
 design considerations, 2-30
 example, 2-30
MAX, 2-23
MetaLink
MIN, 2-23
MINUS, 2-12
MOD, 3-49
MONTHS_BETWEEN, 3-30

N

Name and Address operator, 2-28
new features
 See also Oracle9i Warehouse Builder Release Notes
NEW_TIME, 3-30
NEXT_DAY, 3-31
NEXTVAL, 2-10
 Sequence operator, 2-10
NLS_CHARSET_DECL_LEN function, 3-61
NLS_CHARSET_ID function, 3-62
NLS_CHARSET_NAME function, 3-62
NLS_INITCAP function, 3-19
NLS_LOWER function, 3-19
NLS_UPPER function, 3-20
NLSSORT function, 3-18

NONE, 2-23
Number Transformations, 3-44
number transformations, 3-44
NVL, 2-8

O

OLAP transformations, 3-55
operators
 aggregator, 2-20
 constant, 2-28
 data cleansing, 2-28
 Deduplicator operator, 2-1
 Filter, 2-2
 Joiner, 2-4
 Key Lookup operator, 2-7
 Match-Merge operator, 2-30
 Name and Address, 2-28
 Pivot operator, 2-8
 sequence, 2-10
 Sequence operator, 2-10
 Set operator, 2-12
 SQL, 2-1
 transformation operators, 2-1
 Unpivot operator, 2-19
Oracle
 accessibility
Oracle Library, 3-2
Oracle Transformation Libraries, 3-2
Oracle, contacting
OracleMetaLink
ORDER BY, 2-13
other (non-SQL) transformations, 3-61

P

Pivot operator
 about, 2-8
 example, 2-9
PL/SQL
 importing PL/SQL packages, 3-3
POWER, 3-49
procedures
 Global Shared Library, 3-2

R

ROUND, 3-32, 3-50
RPAD, 3-21
RTRIM, 3-22

S

Section 508 compliance, ix
sequence, 2-10
Sequence operator, 2-10
Set operator, 2-12
SIGN, 3-50
SIN function, 3-50
SINH function, 3-51
Sorter operator

- operators
 - Sorter operator, 2-13
- SOUNDEX, 3-22
- Splitter operator
 - operators
 - Splitter operator, 2-15
- SQL Operators, 2-1
- SQRT, 3-51
- STDDEV, 2-24
- STDDEV_POP, 2-25
- STDDEV_SAMP, 2-25
- SUBSTR, 3-23
- SUM, 2-26
- surrogate key, 2-10
- SYSDATE, 2-28
- SYSDATE function, 3-32
- SYSTIMESTAMP, 2-28

T

- Table Function operator, 2-18
- TAN function, 3-51
- TANH function, 3-52
- TO_CHAR, 3-32, 3-52
- TO_DATE, 3-24
- TO_MULTI_BYTE, 3-24
- TO_NUMBER, 3-25
- TO_SINGLE_BYTE, 3-25
- transformation libraries
 - about, 3-2
 - accessing, 3-2
 - Global Shared Library, 3-2
 - Oracle Library, 3-2
- Transformation Operators, 2-1
- transformations
 - about, 3-1
 - administrative, 3-3
 - character, 3-12
 - date, 3-29
 - functions, 3-1
 - imported package, 3-2
 - number, 3-44
 - OLAP, 3-55
 - operators, 2-1
 - other (non-SQL), 3-61
 - overview, 3-1
 - predefined, 3-1
 - procedures, 3-1
 - user transformation package, 3-1
 - XML, 3-58
- transformations *See also Oracle Warehouse Builder Transformation Guide*
- TRANSLATE, 3-26
- TRIM, 3-26
- TRUNC, 3-33, 3-53

U

- UID, 3-62
- UNION, 2-12

- UNION ALL, 2-12
- Unpivot operator
 - about, 2-19
 - example, 2-19
- updates
 - to Warehouse Builder documentation
- UPPER, 3-27
- USER, 3-63
- user manuals
 - accessibility, ix
- User Transformation Package, 3-1
- User's Guide*
 - conventions, xvii
 - related documents, xvii
 - to order documentation
 - updates available at

V

- VAR_POP, 2-26
- VAR_SAMP, 2-27
- VARIANCE, 2-27

W

- Warehouse Builder
 - documentation for, xvii
 - new features in this release
 - to order documentation
- WB_ABORT, 3-3
- WB_ANALYZE_SCHEMA, 3-3
- WB_ANALYZE_TABLE, 3-4
- WB_CAL_MONTH_NAME, 3-33
- WB_CAL_MONTH_OF_YEAR, 3-34
- WB_CAL_MONTH_SHORT_NAME, 3-34
- WB_CAL_QTR, 3-35
- WB_CAL_WEEK_OF_YEAR, 3-35
- WB_CAL_YEAR, 3-36
- WB_CAL_YEAR_NAME, 3-36
- WB_COMPILE_PLSQL, 3-4
- WB_DATE_FROM_JULIAN, 3-37
- WB_DAY_NAME, 3-37
- WB_DAY_OF_MONTH, 3-38
- WB_DAY_OF_WEEK, 3-38
- WB_DAY_OF_YEAR, 3-39
- WB_DAY_SHORT_NAME, 3-39
- WB_DECADE, 3-40
- WB_DISABLE_ALL_CONSTRAINTS, 3-5
- WB_DISABLE_ALL_TRIGGERS, 3-6
- WB_DISABLE_CONSTRAINT, 3-6
- WB_DISABLE_TRIGGER, 3-7
- WB_ENABLE_ALL_CONSTRAINTS, 3-8
- WB_ENABLE_ALL_TRIGGERS, 3-9
- WB_ENABLE_CONSTRAINT, 3-9
- WB_ENABLE_TRIGGER, 3-10
- WB_HOUR12, 3-40
- WB_HOUR12MI_SS, 3-41
- WB_HOUR24, 3-41
- WB_HOUR24MI_SS, 3-42
- WB_IS_DATE, 3-42

WB_IS_NUMBER, 3-55
WB_IS_SPACE, 3-29
WB_JULIAN_FROM_DATE, 3-43
WB_MI_SS, 3-43
WB_OLAP_LOAD_CUBE, 3-56
WB_OLAP_LOAD_DIMENSION, 3-56
WB_OLAP_LOAD_DIMENSION_GENUK, 3-57
WB_TRUNCATE_TABLE, 3-11
WB_WEEK_OF_MONTH, 3-44
WB_XML_LOAD, 3-58
WB_XML_LOAD_F, 3-58
WB.LOOKUP_CHAR, 3-27, 3-28
WB.LOOKUP_NUM, 3-53, 3-54
WHERE, 2-2, 2-15

X

XML Transformations, 3-58

Z

ZONED, xii