Oracle® Analytics

Building Semantic Models in Oracle Analytics Server

F91008-01
March 2024

Oracle Analytics Building Semantic Models in Oracle Analytics Server,

F91008-01

Primary Author: Stefanie Rhone

Contributing Authors: Shounak Ganguly

Contents

3 Get Started with Semantic Modeling

4 Develop Semantic Models in a Collaborative Environment

5 Work with Data Sources

6 Migrate From Model Administration Tool

7 Create a Semantic Model

8 Build a Semantic Model's Physical Layer

9 Build a Semantic Model's Logical Layer

10 Build a Semantic Model's Presentation Layer

11 Work with Logical Hierarchies

12 Manage Logical Table Sources

13 Create and Use Variables in a Semantic Model

Preface

Learn how to use Semantic Modeler to build and deploy semantic models for use in workbooks, analyses, and dashboards.

Audience

This guide is intended for data modelers and business intelligence analysts who use Oracle Analytics Server:

- **Data Modelers** use the Semantic Modeler to create, design, edit, and deploy semantic models to Oracle Analytics Server.
- **Analysts** use the deployed semantic model's subject areas to model enterprise data and create workbooks, analyses, and dashboards for consumers. Analysts can select interactive visualizations and create advanced calculations to reveal data insights.

Documentation Accessibility

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

Access to Oracle Support

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

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Conventions

The following text conventions are used in this document:

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

Part I
Before You Begin

This part contains information about understanding and designing your semantic models.

Topics:

1

Introduction to Semantic Models

This chapter introduces you to Oracle Analytics semantic models.

Topics:

What Is a Semantic Model?

A semantic model is a metadata model that contains physical database objects that are abstracted and modified into logical dimensions. A semantic model is designed to present data for analysis according to the structure of the business.

After deployment, the semantic model is presented to users as subject areas, which are made up of tables, columns, and hierarchies. In the semantic model, these are mapped to the data sources that provide data to the workbooks, analyses, and dashboards that users create and consume.

A semantic model acts like a translation layer between your application and your underlying data structures. You can use this metrics-oriented data layer that the semantic model exposes directly with APIs, with embedded visualizations, or from other analytics tools to support your enterprise's advanced analytics applications.

A well-designed semantic model meets the business requirements of the stakeholders without them needing to understand the complexity of the underlying data structure. And a well-designed semantic model enable analysts to design workbooks, analyses, and dashboards to query data in the same intuitive way that users think about their business and ask business questions.

A semantic model enables you to structure data in a business-friendly way. It enables you to add business semantics to provide meaning to the data and the governance rules that secure data access.

About a Semantic Model's Architecture

The semantic model contains three layers of metadata that build on each other and prepare the data source's data for users to query and analyze.

Physical Layer

This is the first layer of the semantic model.

The physical layer defines the objects and relationships that the Oracle Analytics query engine needs to write native queries against each physical data source. You create this layer by importing tables from your data sources into physical databases and then creating alias tables to obfuscate the actual database tables from end users and to control access and updates to the actual data.

Separating the logical behavior of the application from the physical model provides the ability to federate multiple physical sources to the same logical object, enabling aggregate navigation and partitioning, as well as dimension conformance and isolation from changes in the physical sources.

The physical layer can contain: subject areas, folders, localization configuration information, role-based permissions, row-level security, variables, governance rules, and mappings.

See What is the Physical Layer?

Logical Layer

This is the second layer of the semantic model.

The logical layer defines the logical model of the data and specifies the mapping between the logical model and the physical schemas. This layer determines the analytic behavior seen by users, and defines the superset of objects and relationships available to users. The logical layer hides the complexity of the source data models.

Each column in the logical layer maps to one or more columns in the physical layer. At runtime, the Oracle Analytics query engine evaluates Logical SQL requests against the logical layer, and then uses the mappings to determine the best set of physical tables and files for generating the necessary physical queries. The mappings often contain calculations and transformations, and might combine multiple physical tables.

The logical layer can contain: business models, business entities, levels, measures, filters, aggregates, table sources, hierarchies, level-based measures, calculations, aggregates by dimension, and fragmented or federated logic.

See What is the Logical Layer?

Presentation Layer

This is the third layer of the semantic model.

The presentation layer provides a way to present customized, secure, role-based views of a logical layer to users. It adds a level of abstraction over the logical layer and provides the view of the data seen by users building requests in Oracle Analytics and other clients. The presentation layer allows users to easily query data without having to understand the underlying data source.

You can create multiple subject areas in the presentation layer that map to a single logical layer, effectively breaking up the logical layer into manageable pieces.

The presentation layer can contain: tables, data sources, aliases, joins, connections, fragmented sources, and federated sources.

See What is the Presentation Layer?

How Does a Semantic Model Query Data?

The Oracle Analytics query engine interprets Logical SQL queries and generates optimized Physical SQL queries to data sources as specified in a semantic model.

Oracle Analytics Query Engine

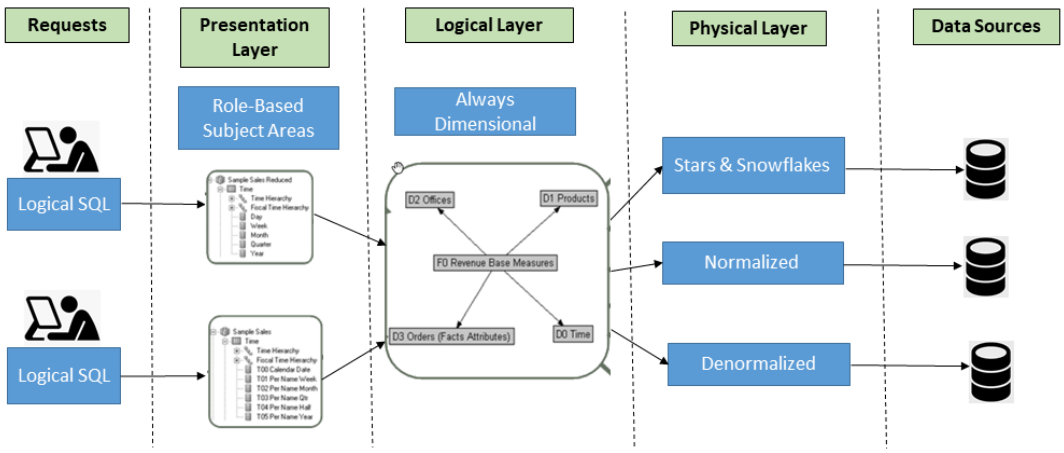
The Oracle Analytics query engine is the backbone of Oracle Analytics' governed and self-service analytics functionality. The query engine provides centralized data access, computes calculations, and enables data governance by creating a pipeline through which anyone can consume information specific to their application roles across their enterprise. The query engine is central to data visualizations, dashboards, ad-hoc queries, mobile access, enterprise reporting, data flows, and more. The semantic model functions as the brain of the query engine.

The Oracle Analytics query engine maintains the logical data model and provides client access to the model using ODBC connectivity or native APIs, such as OCI for the Oracle Database.

Logical SQL Queries

The Oracle Analytics query engine uses the semantic model's metadata to translate Logical SQL queries from workbooks, dashboards, and analyses into physical SQL queries against the mapped data sources that supply the data. The Oracle Analytics query engine also transforms and combines the physical result sets and perform final calculations.

This diagram shows how a Logical SQL query traverses the layers of a semantic model to query the data sources.



Logical Request Transformation Example

This example shows how the Oracle Analytics query engine interprets and converts a Logical SQL query to a Physical SQL query.

The Oracle Analytics query engine receives the following simple client request:

```
SELECT
"D0 Time"."T02 Per Name Month" saw_0,
"D4 Product"."P01 Product" saw_1,
"F2 Units"."2-01 Billed Qty (Sum All)" saw_2
FROM "Sample Sales"
ORDER BY saw_0, saw_1
```

The Oracle Analytics query engine converts the Logical SQL query into a Physical SQL query:

```
WITH SAWITH0 AS (
select T986.Per_Name_Month as c1, T879.Prod_Dsc as c2,
   sum(T835.Units) as c3, T879.Prod_Key as c4
from
   Product T879 /* A05 Product */ ,
   Time_Mth T986 /* A08 Time Mth */ ,
   FactsRev T835 /* A11 Revenue (Billed Time Join) */
where ( T835.Prod_Key = T879.Prod_Key and T835.Bill_Mth = T986.Row_Wid)
group by T879.Prod_Dsc, T879.Prod_Key, T986.Per_Name_Month
)
select SAWITH0.c1 as c1, SAWITH0.c2 as c2, SAWITH0.c3 as c3
from SAWITH0
order by c1, c2
```

What Is SMML?

The Semantic Modeler Markup Language (SMML) is a JSON-based markup language that describes the design-time semantic model's objects. SMML provides a grammar, syntax, and structure for defining semantic models.

Each SMML file represents an object in the semantic model. You can use a semantic model's SMML files for metadata migration, programmatic metadata generation and manipulation, metadata patching, and other functions.

SMML allows developers to use their semantic model editor of choice. Developers can use the Semantic Modeler user interface and its diagramming capabilities to create models, or use the native SMML editor or their preferred external text editor to create and modify the semantic model.

And because SMML uses JSON files, Semantic Modeler can integrate with any Git-compatible repository, such as GitHub, GitLab, or Git on Oracle Visual Builder, to provide a seamless, efficient multi-user development environment and source control. With full support for branching, merging, pull, push, and commit from within Semantic Modeler, multiuser development becomes much less complicated. With Git integration, you have full visibility to a complete change history and the ability to publish to multiple targets.

Other benefits of SMML include:

- File granularity is at the table level (not the column level), which reduces the number of files to manage.
- SMML files are human readable.
- Object references are easy to define with fully-qualified object names.
- SMML object names match the names used by the Semantic Modeler user interface.

- Attributes order matches the attribute order used by the Semantic Modeler interface.
- The semantic model's SMML files can be exported as data model archive (.mar) files.

For more information about SMML, see SMML Schema Reference for Oracle Analytics Cloud.

Oracle Analytics Data Modeling Tools

Oracle Analytics offers several data modeling tools that you can use to create enterprise semantic models and self-service datasets.
Use this topic to learn the differences between the data modeling tools and which tool to use based on what you want to create.

Tool	Use to create	Description
Semantic Modeler	Governed data models	A browser-based modeling tool that developers use for creating, building, and deploying the semantic model to an .rpd file. The Semantic Modeler editor is a fully-integrated Oracle Analytics component. Because the Semantic Modeler generates Semantic Model Markup Language (SMML) to define semantic models. developers have the choice of using the Semantic Model editor, the native SMML editor, or another editor to develop semantic models. Semantic Modeler provides full Git integration to support multi-user development. You can use the Semantic Modeler to create semantic models from the data sources that it supports. Use the Model Administration Tool to create semantic models from data sources that Semantic Modeler doesn't support. See What Is Oracle Analytics Semantic Modeler? and Supported Data Sources.

Tool	Use to create	Description
Model Administration Tool Administration Tool Oracle BI Administration Tool	Governed data models	A mature, longstanding, heavyweight, developer-focused modeling tool that provides complete governed data modeling capabilities. Developers use the Model Administration Tool to define rich business semantics, data governance, and data interaction rules to fetch, process, and present data at different granularity from disparate data systems. Oracle recommends that you use Semantic Modeler to create semantic models from the data sources Semantic Modeler supports, and that you use Model Administration Tool to create semantic models from any data source that Semantic Modeler doesn't support. See About Creating Semantic Models with Model Administration Tool and Supported Data Sources. The Model Administration Tool is a Windows-based application that isn't integrated into the Oracle Analytics interface. You download the Model Administration Tool and install it onto and use it from your computer. If you previously modeled your business data with Oracle BI Enterprise Edition, you don't have to start from scratch in Oracle Analytics. You can use the Model Administration Tool to upload a complete semantic model .rpd file to Oracle Analytics Server and immediately start using your subject areas in visualizations, dashboards, and analyses. Optionally, can use the Model Administration Tool to download, edit, and upload your semantic model .rpd files to Oracle Analytics.
Data Model Editor (For Pixel-Perfect Reports)	XML data structure for Pixel-Perfect Reports	The Data Model editor enables you to combine data from multiple datasets into a single XML data structure for pixel-perfect reports. See Build Data Models for Pixel-Perfect Reports.
Dataset Editor	Self-service data models	A user-friendly data modeling and data preparation tool that data analysts and business analysts use to create datasets containing multiple tables with joins. A dataset can contain data from local and remote files, including more than 50 connections and subject areas. The Dataset editor is available from the Oracle Analytics interface and enables business users to create self-service data models on top of existing governed semantic models. See What Are Datasets?

What Is Oracle Analytics Semantic Modeler?

Oracle Analytics Semantic Modeler is a browser-based data modeling tool with a modern user interface. It provides a streamlined user experience for creating semantic models (governed data models), and is a fully-integrated Analytics Cloud component. Key benefits include:

- Provides a modern alternative to the Model Administration Tool.
- Includes complete semantic modeling capabilities for most data sources. Support for more data sources to be included in future releases.

- Contains complete semantic modeling capabilities, including physical diagrams, logical diagrams, and lineage diagrams.
- Includes streamlined search integration that seamlessly shows relationships among the semantic model's objects.
- Includes a lineage viewer to show the mapping of physical, logical, and presentation layers.
- Integrates with any Git-based platform, such as GitHub, GitLab, or Git on Oracle Visual Builder, to provide a seamless, efficient multi-user development environment and source control.
- Allows developers to perform most common Git operations from within the Semantic Modeler user interface.
- Transparently generates Semantic Model Markup Language (SMML), which uses Javascript Object Notation (JSON) to define semantic models.
- Allows developers to use their semantic model editor of choice. Developers can use the Semantic Modeler user interface and its diagramming capabilities to create models, or display and use the native SMML editor or use their preferred external text editor to create or modify the semantic model's source code.
- Ability to validate calculations and advanced expressions from within the SMML editor.

2

Plan a Semantic Model

This topic contains information to help you design a semantic model.

Topics:

Understand a Semantic Model's Requirements

Before you can begin modeling data, you must understand your semantic model's requirements.

When creating a semantic model, the key objective is to design a model that presents business information in the way that users understand their business' structure. A well designed semantic model allow users to query data in the same way that they would ask business questions.

Use this list of questions to help you analyze a semantic model's requirements:

- What kinds of business questions are business analysts trying to answer?
- What are the measures required to understand business performance?
- What are all the dimensions the business operates under? Or, in other words, what are the dimensions used to break down the measurements and provide headers for the reports?
- Are there hierarchical elements in each dimension, and what types of relationships define each hierarchy?

Answering these questions makes it easier to identify and define the semantic model's objects.

Components of a Semantic Model

Fact tables, dimension tables, joins, and hierarchies are a semantic model's key components.

Component	Description
Fact Tables	Fact tables contain measures (columns) that have aggregations built into their definitions. Measures aggregated from facts must be defined in a fact table. Measures are typically calculated data such as dollar value or quantity sold, and they can be specified in terms of hierarchies. For example, you might want to determine the sum of dollars for a given product in a given market over a given time period. Each measure has its own aggregation rule such as SUM, AVG, MIN, or MAX. A business might want to compare values of a measure and need a calculation to express the comparison.
Dimension Tables	A business uses facts to measure performance by well-established dimensions, for example, by time, product, and market. Every dimension has a set of descriptive attributes. Dimension tables contain attributes that describe business entities (like Customer Name, Region, Address, or Country). Dimension table attributes provide context to numeric data, such as being able to categorize Service Requests. Attributes stored in this dimension might include Service Request Owner, Area, Account, or Priority. Dimension tables in the model are conformed. In other words, even if there are three different source instances of a particular Customer table, the model only has one table. To achieve this, all three source instances of Customer are combined into one using database views.
Joins	Joins indicate relationships between fact tables and dimension tables in the model. When you create joins, you specify the fact table, dimension table, fact column, and dimension column you want to join. Joins allow queries to return rows where there is at least one match in both tables. Tip: Analysts can use the option **Include Null Values** when building reports to return rows from one table where there’re no matching rows in another table.
Hierarchies	Hierarchies are sets of top-down relationships between dimension table attributes. In hierarchies, levels roll up from lower levels to higher levels. For example, months can roll up into a year. These rollups occur over the hierarchy elements and span natural business relationships.

Plan the Physical Layer

Use the topics in this section to determine the physical layer's content.

Topics:

About Physical Schema Types

When you model data sources, you can break down the model of any physical source into overlapping dimensional subsets.

Each physical model mirrors the shape of the source. For example, snowflake or normalized.

- **Star Schemas**

  A star schema is a set of dimensional schemas (stars) that each have a single fact table with join relationships to several dimension tables. When you map a star to the business model, you first map the physical fact columns to one or more logical fact tables. Then, for each physical dimension table that joins to the physical fact table for that star, you map the physical dimension columns to the appropriate conformed logical dimension tables.

- **Snowflake Schemas**

  A snowflake schema is similar to a star schema, except that each dimension is made up of multiple tables joined together. Like star schemas, you first map the physical fact columns to one or more logical tables. Then, for each dimension, you map the snowflake physical dimension tables to a single logical table. You can achieve this by either having multiple logical table sources, or by using a single logical table source with joins.

- **Normalized Schemas**

  Normalized schemas distribute data entities into multiple tables to minimize data storage redundancy and optimize data updates. Before mapping a normalized schema to the business model, you need to understand how the distributed structure is understood in terms of facts and dimensions.

  After analyzing the structure, you pick a table that has fact columns and then map the physical fact columns to one or more logical fact tables. Then, for each dimension associated with that set of physical fact columns, you map the distributed physical tables containing dimensional columns to a single logical table. Like with snowflake schemas, you can achieve this by having multiple logical table sources, or by using a single logical table source with joins. Mapping normalized schemas is an iterative process because you first map a certain set of facts, then the associated dimensions, and then you move on to the next set of facts.

  When a single physical table has both fact and dimension columns, you may need to create a physical alias table to handle the multiple roles played by that table.

- **Fully Denormalized Schemas**

  This type of dimensional schema combines the facts and dimensions as columns in one table, and is mapped differently than other types of schemas. When you map a fully denormalized schema to the star-shaped business model, you map the physical fact columns from the single physical fact table to multiple logical fact tables in the business model. Then, you map the physical dimension columns to the appropriate conformed logical dimension tables.

Identify the Data Source Table Structure

When you build a semantic model, you map logical tables to the underlying physical tables in your data sources. Before you can map the tables, you need to identify the contents of the physical data sources as it relates to your business model.

Identify the following contents of the physical data source:

- Identify the contents of each table.
- Identify the detail level for each table.
- Identify the table definition for each aggregate table. This lets you set up the aggregate navigation. The Oracle Analytics query engine requires the following:
  - The columns the tables are grouped by (the aggregation level).
  - The type of aggregation: `SUM`, `AVG`, `MIN`, `MAX`, or `COUNT`.
- Identify the contents of each column.
- Identify how each measure is calculated.
- Identify the joins defined in the database.

To find this information, go to any available documentation that describes the data elements created when the data source was implemented. Or you could work with the DBA for each data source to gather this information.

To fully understand all of the data elements, you could ask the people who use or own the data, or the developers of the applications that create the data.

Physical Layer Design Tips

Use the information in this topic to help you design the semantic model's physical layer.

The most common way to create the schema in the Physical layer to import metadata from databases and other data sources. If you import metadata, many of the properties are configured automatically based on the information gathered during the import process. You can also define other attributes of the physical data source, such as join relationships, that might not exist in the data source metadata.

For each data source, there is at least one corresponding connection pool. The connection pool contains data source name (DSN) information used to connect to a data source, the number of connections allowed, timeout information, and other connectivity-related administrative details.

Use these tips when designing the physical layer:

- You should use table aliases in the physical layer to eliminate extraneous joins, including the following:
  - Eliminate all physical joins that cross dimensions (inter-dimensional circular joins) by using aliases.
  - Eliminate all circular joins (intra-dimensional circular joins) in a logical table source in the physical layer by creating physical table aliases.

    A circular join involves using different joins from the same table to get results. For example, suppose you have a Customer table that's used to look up ship-to addresses, and you use a different join to the Customer table to look up bill-to addresses. You can avoid the circular joins by creating an alias table in the physical layer so that only one table instance is used for each purpose, with separate joins.

  If you don't eliminate circular joins, you could get erroneous report results. Also, query performance is negatively impacted by circular joins.

- You should use alias tables to create separate physical joins when you need the join to perform as an inner join in one logical table source, and as an outer join in another logical table source.
- You might import some tables into the physical layer that you might not use right away, but that you don't want to delete. To identify tables that you do want to use right away in the logical layer, you can assign aliases to physical tables before mapping them to the logical layer.
- Use a SELECT statement only if there is no other solution to your modeling problem. You should create a physical table or a materialized view. SELECT statements prevent the Oracle Analytics query engine from generating optimized SQL because SELECT statements contain fixed SQL statements that are sent to the underlying data source.
- Decide if you want to set up row-level security controls in the database or in the semantic model. This decision determines if you share connection pools and cache, and may limit the number of separate source databases you want to include in your deployment.

Plan the Logical Layer

Use the topics in this section to determine the logical layer's content.

Topics:

Guidelines for Identifying the Logical Layer's Content

Use this sequence to determine what content to include in your semantic model's logical layer.

1. Identify the logical columns that users need to query.
2. Identify each column's role as either a measure column or a dimensional attribute.
3. Arrange the logical columns in a dimensional model based on the relevant roles, relationships between columns, and logic.

Businesses are analyzed by relevant dimensional criteria, and the business model is developed from these relevant dimensions. These dimensional models form the basis of the valid business models to use with the Oracle Analytics query engine.

Although not all dimensional models are built around a star schema, it's a best practice to use a simple star schema in the business model layer. In other words, the dimensional model should represent some measurable facts that are viewed in terms of various dimensional attributes.

After you analyze your business model requirements, you need to identify the specific logical tables and hierarchies that you need to include in your business model.

Identify the Logical Fact Tables

The semantic model's logical layer contains logical fact tables containing measures with aggregations built into their definitions.

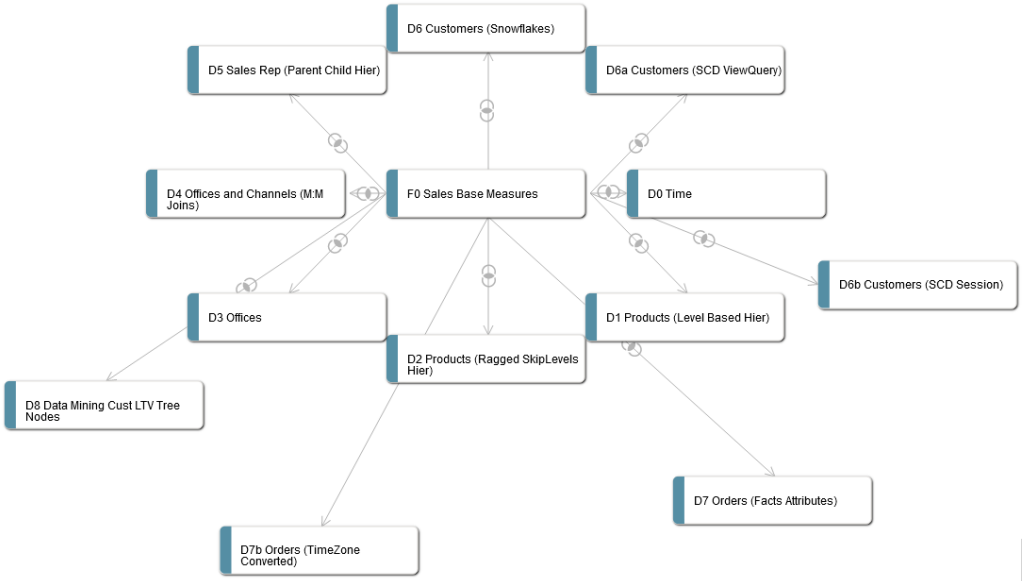
Logical fact tables are different from physical fact tables in relational models. Physical tables in relational models define facts at the lowest possible grain. Logical fact table can contain measures of different grains,

You must define measures aggregated from facts in a logical fact table. Measures are calculated data such as dollar value or quantity sold. You can specify measures in terms of dimensions. For example, you might want to determine the sum of dollars for a given product in a given market over a given time period.

Each measure has its own aggregation rule such as `SUM`, `AVG`, `MIN`, or `MAX`. A business might want to compare values of a measure and need a calculation to express the comparison. You can specify aggregation rules to specific dimensions. You can define complex, dimension-specific aggregation rules in the semantic model.

You don't explicitly label tables in the logical layer as fact tables or dimension tables. The Oracle Analytics query engine treats tables at the *one* end of a join as dimension tables, and tables at the *many* end of a join as fact tables.

The image shows the many-to-one joins to a fact table in a logical diagram. In the logical diagram, all joins have an arrow, indicating the *one* side, pointing away from the fact table. No joins are pointing to it.



Identify the Logical Dimension Tables

Dimension tables contain attributes that describe business entities such as Customer Name, Region, Address and Country.

A business uses facts to measure performance by established dimensions such as by time, product, and market. Every dimension has a set of descriptive attributes. Dimension tables contain primary keys that identify each member.

Dimension table attributes provide context to numeric data, for example, by providing the ability to categorize Service Requests. Attributes stored in a service requests dimension table could include Service Request Owner, Area, Account, and Priority.

Dimensions in the business model are conformed dimensions. For example, if a specific data source has five different instances of a specific Customer table, the business model should only have one Customer table. To achieve conformance, all five physical source instances of Customer are mapped to a single Customer logical table, with transformations in the logical table source as necessary. Conformed dimensions hide the complexity of the physical layer from users, and enable combining data from multiple fact sources at different grains. Conformed dimensions enable combining multiple data sources.

The business model uses business keys for a dimension and level keys instead of generated surrogate keys. For example, you would use *Customer Name* with values like *Oracle* instead of *Customer Key* with values like *1076823*. Using business keys in the business model ensures that all sources for that dimension can conform to the same logical dimension table with the same logical key and level key.

Generated surrogate keys can exist in the physical layer where the keys are useful for their query performance advantages on joins. The logical layer doesn't have surrogate key columns.

Identify Dimensions

Dimensions are categories of attributes that define your business.

Common dimensions are time periods, products, markets, customers, suppliers, promotion conditions, raw materials, manufacturing plants, transportation methods, media types, and time of day. There are many attributes within a dimension. For example, the time period dimension can contain the attributes day, week, month, quarter, and year. Exactly what attributes a dimension contains depends on the way the business is analyzed.

Dimensions contain hierarchies that are sets of top-down relationships between members within a dimension. There are two types of hierarchies:

- **Level-based hierarchies (structure hierarchies) -** In these hierarchies, members of the same type occur only at a single level, while members in parent-child hierarchies all have the same type. Oracle Analytics supports a time dimension level-based hierarchy that provides functionality for modeling time series data.

  In level-based hierarchies, levels roll up from a lower level to higher level, for example, months can roll up into a year. These roll ups occur over the hierarchy elements and span natural business relationships.
- **Parent-child hierarchies (value hierarchies) -** In these hierarchies, the business relationships occur between different members of the same real-world type such as the manager-employee relationship in an organizational hierarchy tree. Parent-child hierarchies don't have explicitly named levels. There isn't a limit to the number of implicit levels in a parent-child hierarchy.

To define your hierarchies, you specify the **contains** relationships in your business to drive roll up aggregations in all calculations, as well as drill-down navigation in reports and dashboards. For example, if month rolls up into year and an aggregate table exists at the

month level, you can use the table to answer questions at the year level by adding up all of the month-level data for a year.

To determine the correct hierarchy type for your modeling needs, consider the following:

- Are all the members of the same type such as employee, assembly, or account, or are they different types that naturally fall into levels such as year-quarter-month, continent-country-state/province, or brand-line-product?
- Do members have the same set of attributes? For example, in a parent-child hierarchy like *Employees*, all members might have a Hire Date attribute. In a level-based hierarchy like Time, the Day type might have a Holiday attribute, while the Month type doesn't have the Holiday attribute.
- Are the levels fixed at design time (year-quarter-month), or can runtime business transactions add or subtract levels? For example, if you can add a level when the current lowest-level employee hires a subordinate, who then is the new lowest level.
- Are there constraints in your primary data source that require a certain hierarchy type? If your primary data source is modeled in one way, you might need to use the same hierarchy type in your business model, regardless of other factors.

Dimensions can contain multiple hierarchies. Dimensions with multiple hierarchies must always end with the same column. For example, time dimensions often have one hierarchy for the calendar year, and another hierarchy for the fiscal year.

Identify Lookup Tables

When you need to display translated field information from multilingual schemas, you create a logical lookup table that corresponds to a lookup table in the physical layer.

A lookup table stores multilingual data corresponding to rows in the base tables. Before using a specific logical lookup table, you must designate the table as a lookup table in the logical table's lookup tables.

You can use lookup tables to display one set of values to users, while using a different corresponding set of values in the physical query. You can use the lookup table to provide human-readable values that are looked up in a different data source.

Logical Layer Design Tips

The logical layer organizes information by business model. In this layer, each business model is effectively a separate application.

The logical schema defined in each business model must contain at least two logical tables. You must define relationships between all the logical tables.

When designing the logical layer:

- Create the business model with one-to-many logical joins between logical dimension tables and the fact tables wherever possible. The business model should resemble a simple star schema in which each fact table is joined directly to its dimensions.
- Join every logical fact table to at least one logical dimension table. When the source is a fully denormalized table, you must map its physical fact columns to one

or more logical fact tables, and its physical dimension columns to logical dimension tables.

- Define relationships between dimension attributes by creating hierarchies within a logical dimension.
- Map all appropriate fact sources map to the appropriate level in the hierarchy using data aggregation when creating level-based measures.
- Create aggregate sources as separate logical table sources.
- Create a unique level key for each dimension level in a hierarchy. Each logical dimension table must have a unique primary key. The key is also used as the level key for the lowest hierarchy level.
- Ensure that each logical level of a dimension hierarchy contains the correct value. Fact sources are selected on a combination of the fields selected as well as the levels in the dimensions to which they map. By adjusting these values, you can alter the fact source selected by the Oracle Analytics query engine.

Logical Fact Tables

- Logical fact tables can contain measures of different grains. Don't use the grain as a reason to split up logical fact tables.
- Logical fact tables shouldn't contain any keys, except when you need to send Logical SQL queries against the Oracle Analytics query engine from a client that requires keys. In this case, you need to expose those keys in both the logical fact tables, and in the presentation layer.
- All columns in logical fact tables are aggregated measures, except for keys required by external clients, or dummy columns used as a divider. Other non-aggregated columns should exist in a logical dimension table.
- You can use multiple logical fact tables in a single business model. For Logical SQL queries, the multiple logical fact tables behave as if they're one table. Reasons to have multiple logical fact tables include: to automatically create small subject areas in the presentation layer, and to organize and simplify them within the business model.

Calculations

You can define calculations in the following ways:

- Before the aggregation, in the logical table source. For example:

```
sum(col_A *( col_B))
```

- After the aggregation, in a logical column derived from two other logical columns. For example:

```
sum(col A) * sum(col B)
```

You can also define post-aggregation calculations in workbooks, dashboards, analyses, or in Logical SQL queries.

Model Outer Joins

Use this information to model outer joins.

- Queries that use outer joins are usually slower. To avoid performance issues, define outer joins only when necessary. Where possible, use ETL techniques to eliminate the need for outer joins in the reporting SQL.

- Outer joins are always defined in the logical layer. Physical layer joins don’t specify inner or outer.
- You can define outer joins by using logical table joins, or in logical table sources. Which type of outer join you use is determined by whether the physical join maps to a business model join, or to a logical table source join.
- If you must define an outer join, try to create two separate dimensions: one that uses the outer join and one that doesn’t. Make sure to name the dimension with the outer join in a way that clearly identifies it, so that client users can use it as little as possible.
- Avoid using more than one outer join. Instead, to achieve the same effect as a logical outer join, Oracle recommends that the logical join be an inner join and that the analysis designer at design time selects the **Include Null Value** option in the corresponding analysis.

Plan the Presentation Layer

The presentation layer is where you set up the user view of the business model. After you deploy the semantic model, the presentation layer is displayed as subject areas.

The names of folders and columns in the presentation layer can appear in localized language translations. The presentation layer is the appropriate layer in which to set user permissions.

In this layer, you can do the following:

- You can show fewer columns than exist in the logical layer. For example, you can exclude the key columns because they have no business meaning.
- You can organize columns using a different structure from the table structure in the logical layer.
- You can display column names that are different from the column names in the logical layer.
- You can set permissions to grant or deny users access to individual subject areas, tables, and columns.
- You can export logical keys to ODBC-based query and reporting tools.
- You can create multiple subject areas for a single business model.
- You can create a list of alternative names for presentation objects that are used in Logical SQL queries. Alternative names allows you to change presentation column names without breaking existing reports.

The following is a list of tips to use when designing the presentation layer:

- Because there isn't an automatic way to synchronize all changes between the logical layer and the presentation layer, it's best to wait until the logical layer is relatively stable before adding customizations in the presentation layer.
- There are many ways to create subject areas, such as dragging and dropping the entire business model, dragging and dropping incremental pieces of the model, or automatically creating subject areas based on logical stars or snowflakes. Dragging and dropping incrementally works well if certain parts of your business model are still changing.
- For better maintainability, it's a best practice to rename objects in the logical layer rather than the presentation layer Assigning user-friendly names to logical objects

rather than presentation objects ensures that you can use the names in multiple subject areas. Also, it ensures that the names persist even when you need to delete and re-create subject areas to incorporate changes to your business model.

- Members in a presentation hierarchy aren't visible in the presentation layer. You can see hierarchy members in the Workbook editor or in the Analysis editor.
- When setting up data access security for a large number of objects, consider setting object permissions by role rather than setting permissions for individual columns.

Part II
Create and Build Your Model

Topics:

3

Get Started with Semantic Modeling

This chapter provides information to help you understand the general steps required to create and build a semantic model. It also provides information about how to navigate the Semantic Modeler tool, name and search for your semantic model's objects, and manage multi-user development.

Topics:

Workflow to Build a Semantic Model

Here are the common tasks for creating and building a semantic model.

Task	Description	More Information
Understand Semantic Modeler	Use Semantic Modeler to create a semantic model and build its physical, logical, and presentation layers.	Introduction to Semantic Models Plan a Semantic Model
Request Permissions to Use Semantic Modeler	Ask your administrator to give you the BI Data Model Author application role. To check you if have permission to use Semantic Modeler, navigate to the Home page, click **Create**, and look for the **Semantic Model** option. If you don't see the Semantic Model option, then you don't have the BI Data Model Author application role. If you plan to set up one or more data source connections for your semantic model, ask your administrator to give you the DV Content Author application role.	About Application Roles

Task	Description	More Information
Confirm that your data source is supported	Understand which data sources Semantic Modeler supports. Semantic Modeler only supports relational data sources. Before you import a semantic model from Model Administration Tool or Data Modeler, confirm that Semantic Modeler supports the model's data source. Be sure to remove or replace any unsupported data sources in the semantic model before migration. The import fails if a semantic model contains an unsupported data source.	Data Sources Available for Data Modeling
Create the Semantic Model	Create the semantic model in one of the following ways: • Create an empty semantic model. • Import an exported semantic model (.rpd file), an archived semantic model (.zip file), or an .rpd file from Model Administration Tool. • Load the semantic model deployed to Oracle Analytics. • Clone a Git repository to your development environment.	Create a Semantic Model Develop Semantic Models in a Collaborative Environment
Build the Physical Layer	Define the semantic model's data sources and the relationships between physical databases and other data sources that the Oracle Analytics query engine uses to process multiple data source queries. Define other attributes of the physical data sources, such as join relationships, that might not exist in the data source metadata. Tasks include: • Importing metadata • Creating physical tables and columns • Creating alias tables • Creating joins	Build a Semantic Model's Physical Layer

Task	Description	More Information
Build the Logical Layer	Define one or more business model objects that contain the business model definitions and the mappings from logical to physical tables. Create logical tables (fact and dimension) containing logical columns. Tasks include: • Examine logical joins • Examine logical table sources • Rename logical objects • Delete unnecessary logical objects • Create simple measures	Build a Semantic Model's Logical Layer Work with Logical Hierarchies
Build the Presentation Layer	Structure the logical layer's objects to be presented to users as subject areas that they'll use to build visualizations and analyses. Tasks include: • Create presentation tables • Create presentation columns • Rename and reorder presentation columns	Build a Semantic Model's Presentation Layer
Test and Validate the semantic model	Check the semantic model for errors. Deploy the semantic model. Test the semantic model by creating visualizations and analyses and verifying the results. Tasks include: • Check consistency • Deploy • Create and run visualizations and analyses	Check Consistency and Deploy a Semantic Model

Semantic Model Object Naming Requirements

Use these guidelines and requirements to name the semantic model's objects, for example tables and columns.

- Names can contain multi-byte characters.
- Names must be 128 characters or less.
- Names can't contain leading or trailing spaces.
- Names can't contain characters such as single quotes, hash marks, question marks, or asterisks.

Edit Semantic Model Objects Using the SMML Editor

You can use the SMML editor to view and edit the JSON SMML schema file of an object in your semantic model.

The SMML editor displays a semantic model object's text-based JSON SMML schema file based on the object-type JSON schema. If you are viewing or editing an invalid file, syntax and semantic errors are marked on the relevant line of text.

For more information about SMML, see SMML Schema Reference for Oracle Analytics Cloud.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. In the semantic model's left pane, select a layer.
4. Locate the object you want to edit.
5. Right-click the object and then select **Open in SMML Editor**.
6. Edit the SMML schema file and click **Save** to save the semantic model.

About Command-Line Utilities and Semantic Modeler

You can use some of the Oracle Analytics Server command-line utilities with the semantic models that you create in Semantic Modeler.

For information about the command-line utilities, see About the Oracle BI Server Command-Line Utilities.

4

Develop Semantic Models in a Collaborative Environment

This chapter contains information to help you understand how multiple developers can work on and deploy the same model.

Topics:

About Collaborative Semantic Model Development

More than one developer can work on and deploy a semantic model. You can use Git repositories or permissions to allow and manage concurrent semantic model development.

Git repositories and permissions both ensure that you and other developers or teams of developers work on the latest version of the semantic model. You can also learn about the changes that other developers made to the semantic model.

These are the ways that you and your team can collaborate on a semantic model:

- **Git** - Oracle recommends that you use Git in collaborative environments. Implement and use Git to allow the semantic model developers to create and work in branches to add, update, and commit the files on their individual development environments and then push the commits to the remote repository. To learn about how to use Git with semantic models, see About Using Git with Semantic Model Development.
- **Permissions** - Use permissions to provide members of a small development team access to the semantic model. Permissions determine which roles and users can access and develop the semantic model. When you use permissions, only one developer at a time can work on the semantic model. See Use Permissions for Collaborative Semantic Model Development.

Use Permissions for Collaborative Semantic Model Development

The owner of a semantic model can assign access permissions to other semantic model developers in a concurrent development environment.

Use permissions to share a semantic model within a small development team. When you use permissions to share a semantic model, only one developer at a time can work on the semantic model.

The **Share using Permissions** option is only available for semantic models that aren't using Git. When you use Git to share a semantic model, you can't change it to use permission to share it. But if you use permissions to share a semantic model, you can later update sharing to use Git.

These are the permissions that you can assign to the semantic model's users and roles:

- Full Control - Choose this option to give the corresponding users and roles the ability to access and modify the semantic model, and the ability to add and assign users and roles and permissions to the semantic model.
- Read-Write - Choose this option to give the corresponding users and roles the ability to access and modify the semantic model.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. Locate the semantic model that you want share, click **Actions**, and then click **Inspect**.
3. Click the **Sharing** tab.
4. In the Sharing tab, click **Share using Permissions**.
5. Optional: To add users and roles, click the **Add** field and type the name of the user or role that you want to add. Select the user or role from the search results list to add it, and click the permission that you want to assign it.
6. Optional: To modify permissions, locate a user or role and click the permission that you want to assign to it.
7. Optional: To delete a user or role, hover over it and click **Delete**.
8. Click **Save**.

About Using Git with Semantic Model Development

You can use Git to enable sharing and concurrent semantic model development. You can use any Git service in a public cloud that Oracle Analytics can access.

Examples of Git services that you can use are: Oracle Visual Builder Studio, GitHub, Bitbucket, GitLab, and Azure DevOps.

To learn about other ways of sharing semantic models, see About Collaborative Semantic Model Development.

A semantic model is comprised of a set of SMML files. When you create and develop a semantic model locally, the model's SMML files are stored in Oracle Cloud. To make

a semantic model's SMML files available for other development team members to work on, the semantic model's owner creates a Git repository, initializes it with HTTPS or SSH, and uploads the semantic model's SMML files to the repository. Each developer creates a semantic model and uses HTTPS or SSH to connect to and clone the semantic model's SMML files to their Git repository.

When working on a cloned semantic model, the development team creates and works in branches to add, update, and commit the files on their computers and then pushes the commits to the remote repository.

To effectively create and contribute to a semantic model Git repository, you must have a basic understanding of Git and how to work with branches. If you're new to Git and want to learn more about Git repositories and Git basics, such as remote repositories, cloning, commits, pushes, and branches, then read the Git documentation. See https://git-scm.com/book/ and http://git-scm.com/doc.

Upload a Semantic Model to a Git Repository Using HTTPS

An HTTPS connection uses your Git user name and password to initialize and upload a semantic model to an empty Git repository.

Before you can use HTTPS to connect to, initialize, and upload a semantic model to an empty Git repository, you must:

- Go to your Git provider and create an empty repository.
- Copy the empty repository's URL needed to initialize the Git repository.
- Know your Git user name and password to create the Git profile to authenticate to the Git repository. If you're using Github, then instead of a Git user password, you need to know your personal access token. Or choose a profile that you use with other semantic model Git repositories. See View and Manage Git Profiles.
- Create and save the semantic model to upload to the Git repository.

After you've uploaded the semantic model to the Git repository, provide the URL to your development team members. Developers use the URL to clone the Git repository to their development environments.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Toggle Git Panel** to open the Git pane.
4. In the Git pane, click **Start**.
5. In Initialize Git, enter the repository's URL using the following format: `https://gitserver.com/myorg/myproject.git`. Click **Continue**.
6. Click **Git profile** and either select a Git profile that you've already used to initialize or clone a Git repository, or select **New Profile** and enter a profile name, your Git user name, and your password to create a profile. If you're using Github, then instead of entering a Git user password, enter your personal access token.
7. Click **Initialize Git**.

Upload a Semantic Model to a Git Repository Using SSH

An SSH connection uses a key that you generate in Oracle Analytics and copy into your Git account to create an SSH key. You use this key to initialize and connect to a Git repository without needing to supply a Git user name and password.

Before you can use SSH to connect to, initialize, and upload a semantic model to an empty Git repository, you must:

- Go to your Git provider and create an empty repository.
- Copy the empty repository's URL needed to initialize the Git repository.
- Decide whether to create a Git profile or use an existing profile to authenticate to the Git repository. An existing profile is a profile that you use with other semantic model Git repositories. See View and Manage Git Profiles.
- Create and save the semantic model to upload to the Git repository.

After you've uploaded the semantic model to the Git repository, provide the URL to your development team members. Developers use this URL to clone the Git repository to their development environments.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Toggle Git Panel** to open the Git pane.
4. In the Git pane, click **Start**.
5. In Initialize Git, enter the repository's URL using the following format: `git@gitserver.com:myorg/myproject.git`. Click **Continue**.
6. Click **Git profile**.
7. Optional: If you want to use a Git profile that you've already used to initialize or clone a Git repository, then select an existing profile.
8. Optional: If you want to create a profile, then select **New Profile** and click **Generate Key**.
9. If you completed the previous step to create a profile and generate a key, then click **Copy Key**, go to your Git account, and use the copied key to create an SSH key. Then return to the Oracle Analytics Initialize Git wizard.
10. Click **Initialize Git**.

Work With Branches

By default a semantic model's Git repository has one default main branch. However, you can add more branches to the repository for development purposes.

Branching lets you work on different features and updates at any time without affecting the original source code. You can create branches for feature development work and for things like urgent product fixes.

Before you start working on a new feature or update major portions of the source code, it’s considered a good practice to create a local branch and commit your changes to the local branch. This way your changes don’t affect the original source code and are safe to test and review.

Oracle assumes that you understand the concepts of working with branches in Git repositories. See https://git-scm.com/book/en/v2/ to learn more about the Git branch workflow.

The Oracle Analytics Git pane contains tabs that correspond to Git's standard repository development functionality like push, pull, and merge. This table describes how to use each tab.

Tab Name	Icon	Description
Status		Use this tab to view a list of and manage your unstaged, staged, and committed changes. This tab also displays files with merge conflicts. You can stage and commit some or all changes. When you commit a change, it moves the change into the branch you're working on.
Pull		Use this tab to pull the committed changes made by other developers on the remote branch into your local branch. You use pull to ensure that you're working with the latest code.
Push		Use this tab to push your staged and committed changes to the remote branch. The changes you push to the remote branch are available to the other developers using the branch.
Merge		Use this tab to merge the contents of the selected branch into your current branch and resolve any resulting conflicts. See Understand and Resolve Merge Conflicts.
Switch Branch		Use this tab to change from one branch to another. The name of the branch you're working on is displayed in the Semantic Modeler heading.
Create Local Branch		Use this tab to create a local branch from the branch that you select. You work on your local branch instead of working on another public branch. On your local branch, you make your unstaged, uncommitted changes. Other developers can't access and update this branch. Only you can update this branch.
Delete Branch		Use this tab to select and delete your local branches after you've finished working on them.
Manage Git Profiles		Use this tab to regenerate or copy profile keys, update a profile's Git user name and password, or delete a profile. See View and Manage Git Profiles.

View and Manage Git Profiles

A Git profile contains your Git user name and password or the SSH key that you use to access semantic model Git repositories. You create a Git profile or use an existing profile when you create or clone a Git repository.

Use the **Manage Git Profiles** tab to regenerate or copy profile keys, update a profile's Git user name and password, or delete a profile. In most cases you won't need to update your profiles or regenerate your SSH key.

> **Note:**
>
> Deleting a profile can be destructive. Before you delete a profile, confirm that it's no longer used by any semantic mode Git repositories. After you delete a profile, you can no longer access the semantic model Git repositories initialized with the profile.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.

3. Click **Toggle Git Panel** to open the Git pane.
4. Click the **Manage Git Profiles** tab.
5. Optional: To check or update a Git user name or password associated with a profile, expand the profile and update the credentials. Click **Save**.
6. Optional: To check, copy, or regenerate an SSH key, expand the profile and either click **Copy Key** to copy the key, or click **Regenerate Key** to regenerate the SSH key.
7. Optional: To delete a profile, click its **Delete profile** icon.

Understand and Resolve Merge Conflicts

This topic describes what you need to know to understand and resolve the merge conflicts that Git can't automatically resolve.

Topics:

What are Merge Conflicts?

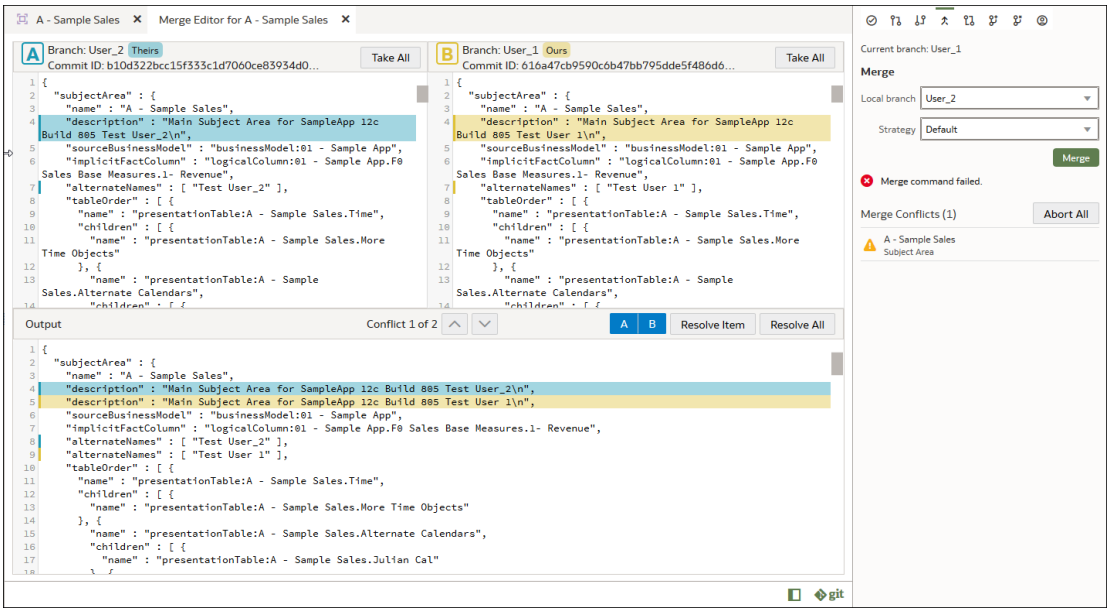
Merge conflicts happen when Git can't automatically determine how to resolve conflicting code changes between commits from two different branches. You need to manually resolve merge conflicts.

In Git, a merge is when users combine commits from different branches. In most cases Git uses the merge strategy that you specified in the Git pane's Merge to resolve the differences between two commits. But in some cases, such as where users have updated the same line of code differently, Git doesn't know which code change is correct. These situations create merge conflicts that you manually resolve by telling Git which code changes to keep and which to discard.

After you manually resolve the merge conflicts, you can successfully commit the changes to the repository.

About the Merge Editor

Use the Merge editor to locate, understand, and resolve merge conflicts in the selected file. The Merge editor provides the same Git functionality that you can access from the command line Git interface.

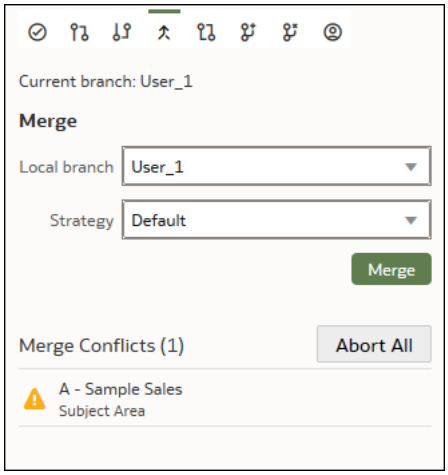


Each Merge editor feature and how you can use it is explained here:

- **Merge Tab**

  The Merge tab displays a drop down list of the branches you're working with and the merge strategy Git uses to resolve most merge conflicts. See Change Git's Merge Strategy. A semantic model is comprised of many SMM files, and any merge conflicts that Git can't resolve are listed by file name in the Merge Conflicts pane.

  You can use the Merge tab to cancel all of the merge conflicts that Git couldn't resolve. See Cancel All Merge Conflicts.



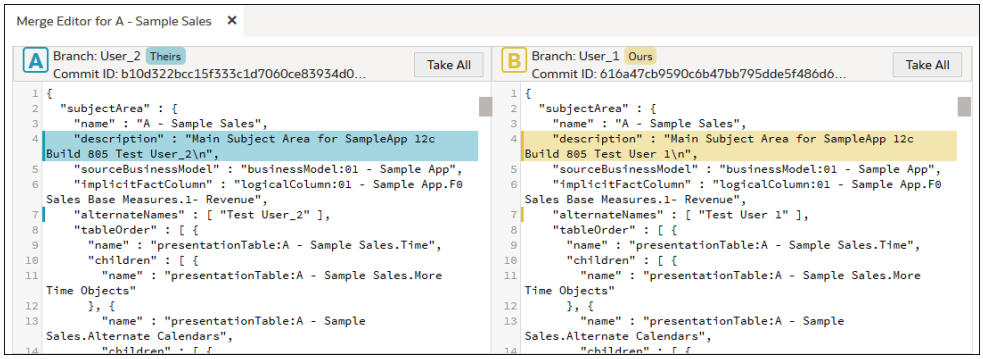
- **Branch Panes**

  The Branch panes highlight the conflicts between Branch A and Branch B.

  – **Branch A** - This area highlights the conflicting code from the source (Theirs) branch, which is the branch you're merging from.

  – **Branch B** - This area highlights the conflicting code from the target (Ours) branch, which is the branch you're merging into.

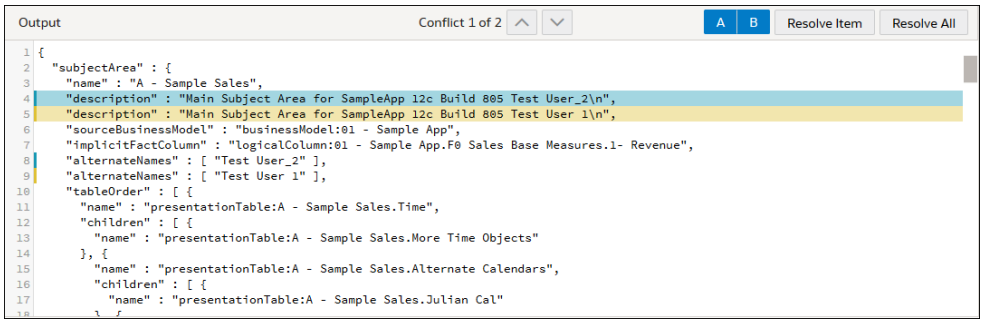
Scroll in either pane to locate and review the highlighted conflicts and decide how to resolve them. You can click the branch's **Take All** button and then go to the Output pane and click **Resolve All** to use the branch as the source of truth to resolve all conflicts. See the "Output Pane" section below for more information about the **Resolve All** button.



- **Output Pane**

  The Output pane stacks and highlights the conflicting code so that you can compare and select which code to use to resolve the conflict. The highlight colors correspond to branches A (Theirs) and B (Ours) displayed in the Branch panes.

  Use the **Conflict** up and down buttons to locate and review the highlighted conflicts.



  This section describes the buttons you use to resolve the merge conflicts.

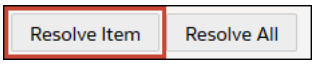
  – **A and B Buttons**

    How you deselect and select the **A** and **B** toggle buttons depends on how you need to resolve the conflict. You can resolve an individual conflict by selecting **A** or **B**, or by re-ordering a code sequence conflict by specifying the sequence (for example, branch B's code should be located before branch A's code). See Understand How to Resolve Conflicts.

  – **Resolve Item Button**

    Use this button to resolve one conflict at a time. After you use the **A** or **B** button to specify how to resolve the highlighted conflict, click **Resolve Item** to

mark the item as resolved. After you click **Resolve Item**, the Output pane navigates to the next conflict. See Resolve Individual Merge Conflicts.

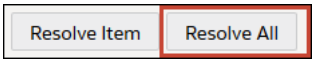


- **Resolve All Button**

  You can use this button in the following ways:

  In the Output pane, use the **A** and **B** buttons to navigate to and specify a resolution for each conflict, and then click **Resolve All** to resolve all conflicts.

  In the Branch pane, click the **Take All** button in the **A** (Ours) or **B** (Theirs) Branch pane and then click **Resolve All** to resolve all conflicts using the branch you chose. See Resolve All Merge Conflicts.



Understand How to Resolve Conflicts

This topic explains how to use the Merge editor's buttons to resolve a file's merge conflicts one at a time or all at the same time.

See About the Merge Editor.

Resolve an Individual Conflict

In the Output pane, use the **A** and **B** buttons to specify how to resolve merge conflicts one at a time.

Confirm that the button corresponding to the branch that you want to use to resolve the conflict is highlighted. By default both **A** and **B** are selected, so to select **A**, you must deselect **B**.

After you set the **A** and **B** buttons to indicate how to resolve the conflict, click the **Resolve Item** button to resolve the issue and navigate to the next conflict.

Reorder a Code Change to Resolve an Individual Conflict

In the Output pane, use the **A** and **B** buttons to reorder a code change to resolve a conflict.

Select a sequence for a conflict where you need to reorder (or stack) a code change. To specify a sequence, click the **A** button and then the **B** button to deselect them. Then click the buttons in either the A before B (so A's code before B's code) or B before A (so B's code before A's code) sequence to specify how to stack the changes. Then click the **Resolve Item** button to resolve the issue and navigate to the next conflict.

Resolve All Conflicts at the Same Time

> **Note:**
>
> You *can't* use the **Resolve All** button with the Output pane's **A** and **B** buttons to resolve all conflicts at the same time.

In the Branch pane, click the **Take All** button for the **A** (Ours) or **B** (Theirs) branch that you want to use as the single source of truth to resolve all conflicts, and then click **Resolve All**.

Mark Individual Conflicts and then Resolve All Conflicts as Marked

In the Output pane, use the **Conflict** up and down buttons to navigate to each highlighted conflict and use the **A** and **B** buttons to specify how to resolve each highlighted conflict. After you've specified how to resolve each item, click **Resolve All** to resolve all conflicts.

Change Git's Merge Strategy

You can choose how you want Git to automatically resolve the merge conflicts that it finds in your branches. In most cases Git can use the merge strategy to resolve the differences between branches.

You can choose from the following Git merge strategies:

Semantic Merge - Use this option to use Semantic Modeler's merge strategy. Semantic Merge merges the model's objects and not just text. Oracle recommends that you use this merge strategy.

Git Merge - Use this option to use Git's default merge strategy. Git's default merge strategy uses a three way algorithm. In cases where there is more than one common ancestor, Git creates a merged tree of the common ancestors and uses it to determine the three way merge.

Ours - Use this option to have Git resolve conflicts by favoring code changes from the branch that you are merging into. This is branch B or the target branch. If you select this option, Git won't use **Ours** to resolve all merge conflicts, but only when it can't use its default merge strategy to resolve specific conflicts.

Theirs - Use this option to resolve conflicts by favoring code from the branch that you are merging from. This is Branch A or the source branch. If you select this option, Git won't use **Theirs** to resolve all merge conflicts, but only when it can't use its default merge strategy to resolve specific conflicts.

If Git can't automatically resolve the merge conflicts, then the Merge Conflicts pane is displayed and lists the conflicts that you must resolve manually. See Resolve All Merge Conflicts and Resolve Individual Merge Conflicts.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Toggle Git Panel** to open the Git pane.

4. Click the Merge tab and go to the **Strategy** field and select a merge strategy to use when you merge branches.
5. Click **Merge**.

Cancel All Merge Conflicts

You can cancel the merge process and reconstruct the pre-merge state of the branches.

If your semantic model contained uncommitted changes when you started the merge, then Git might not be able to reconstruct the pre-merge changes.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Toggle Git Panel** to open the Git pane.
4. Click the Merge tab and go to the Merge Conflicts pane.
5. Click **Cancel All**.

Resolve All Merge Conflicts

You can choose one branch as the source of truth to resolve all merge conflicts in the selected file.

See Understand How to Resolve Conflicts.

The Branch panes highlight the conflicts between Branch A and Branch B.

- **Branch A** - This pane highlights the conflicting code in the Their branch, which is the source branch or the branch you're merging from.
- **Branch B** - This pane highlights the conflicting code in the Ours branch, which is the target branch or the branch you're merging into.

> **Note:**
>
> You *can't* resolve all conflicts by clicking the Output pane's **A** or **B** button and then clicking the **Resolve All** button.

Resolving merge conflicts doesn't fix inconsistencies in your semantic model. You must run the consistency check to detect and fix inconsistencies in your semantic model.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Toggle Git Panel** to open the Git pane.
4. Click the Merge tab and go to the Merge Conflicts pane.
5. Right-click a file and select **View Conflicts**.
6. In the Branch panes, click the **Take All** button corresponding to the branch that you want to use to resolve all merge conflicts in the file.
7. In the Output pane, click **Resolve All**.

8. Go to the Merge Conflicts pane and confirm that the file that contained the conflicts was removed from the list.
9. In the Merge tab, click **Merge**.

Resolve Individual Merge Conflicts

You can review and resolve each of the selected file's merge conflicts one at a time.

The Output pane displays and highlights the conflicting code side-by-side so that you can compare and select which code to use to resolve the conflict. The highlight colors correspond to branches A (Theirs) and B (Ours) displayed in the Branch panes.

For each conflict, use the **A** and **B** buttons to specify which branch's code you want to use to resolve the highlighted conflict. By default both buttons are selected, so to select **A**, you must deselect **B**. Or use the **A** and **B** buttons to specify the order (or stacking) of the code changes. See Understand How to Resolve Conflicts.

Resolving merge conflicts doesn't fix any inconsistencies in your semantic model. You must run the consistency check to detect and fix inconsistencies in your semantic model.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Toggle Git Panel** to open the Git pane.
4. Click the Merge tab and go to the Merge Conflicts pane.
5. Right-click a file and select **View Conflicts**.
6. In the Output pane, locate the highlighted merge conflict.
7. Use the **A** and **B** buttons to specify how to resolve the conflict. Click **Resolve Item**.
8. Click **Conflict Navigate Down** to highlight the next merge conflict.
9. Use the **A** and **B** buttons to specify how to resolve the conflict. Click **Resolve Item**.
10. Navigate to each conflict and resolve it.
11. After you've resolved all conflicts, go to the Merge Conflicts pane and confirm that the file that contained the conflicts was removed from the list.
12. In the Merge tab, click **Merge**.

5

Work with Data Sources

This chapter contains information to help you understand how to connect to data sources to use when creating semantic models.

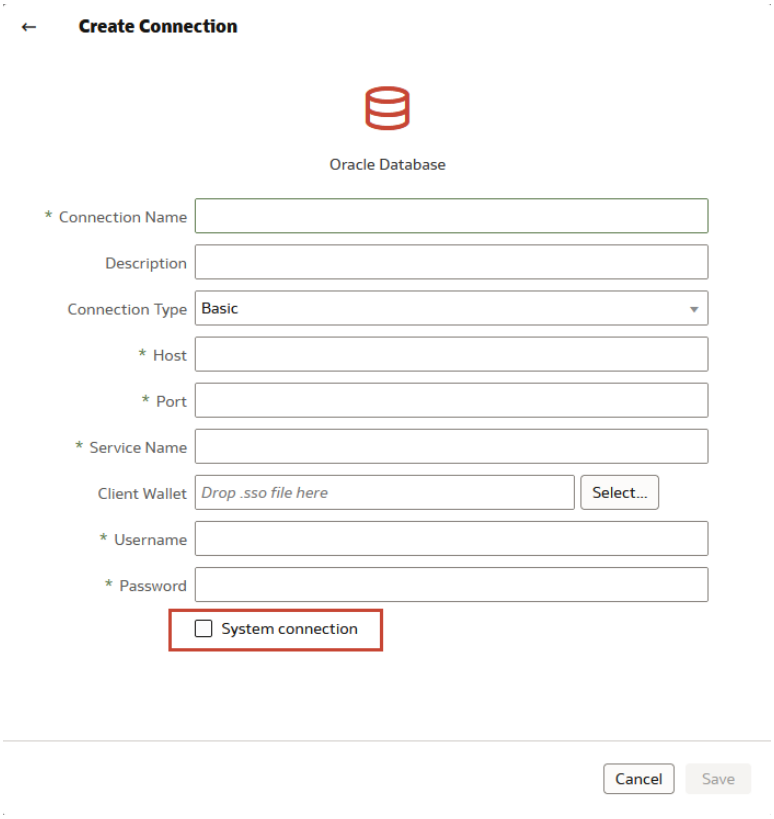
Topics:

About Connections for Semantic Models

A connection must have the **System connection** field selected to make it available for semantic model development.

In Oracle Analytics, you need DV Content Author permissions to create connections to data sources.

To make a data source connection available to semantic models, you must select the **System Connection** checkbox when you create the connection. You can't edit the data source's **System Connection** option after setting up the connection.



Go to the Oracle Analytics Home Page and click **Create**, then **Connection** to connect to each database that you're modeling. Make sure to click the **System connection** checkbox..

To view a list of data sources that you can use in a semantic model, see Data Sources Available for Data Modeling.

Data Sources Available for Data Modeling

Before you create or migrate a semantic model, it is important to understand which data sources are supported by Semantic Modeler and which are supported by Model Administration Tool.

To find out which databases Semantic Modeler supports, look for a Yes in the 'Data Models - Semantic Modeler' column in the supported data sources list. See Certification - Supported Data Sources.

View Available Data Source Connections

A data source connection accesses and supplies tables and data to a semantic model.

Your list contains the connections that you built and the connections that you have permission to access and use.

Semantic models can only use connections with the **System connection** field selected. This field must be selected when the connection is created because it can't be changed after the connection is saved.

If you need to create a connection, see About Connections for Semantic Models.

1. In the Home Page, click **Navigator** and then click **Data**.
2. Click the **Connections** tab to view your connections list.

Semantic Modeler Data Source Limitations

Use the information linked to in the topic to understand the data source limitations that can impact your semantic models.

For information about Apache Hive limitations, see Limitations on the Use of Apache Hive.

For information about Teradata limitations, see Avoid Spool Space Errors for Queries Against Teradata Data Sources.

Import Metadata from Data Sources

You can import metadata for supported data source by selecting the appropriate connection type from the Connections tab in the Semantic Model pane.

To import metadata, you must have all database connections created. To create a connection, see About Connections for Semantic Models.

When you import physical tables, be careful to import only those tables that contain data that are likely to be used in the business models you create. You can search and select the tables that you want to add. Adding large numbers of extraneous tables and other objects adds unnecessary complexity and increases the size of the semantic model.

When you import metadata for most data sources, the default is to import tables and primary keys.

You can also import database views, aliases, synonyms, and system tables. Import these objects only if you want the Oracle Analytics query engine to generate queries against them.

After you import metadata, you should check to ensure that your database and connection pool settings are correct.

In rare cases, the Oracle Analytics query engine can't determine the exact database type during import and instead assigns an approximate type to the database object.

6

Migrate From Model Administration Tool

This chapter contains information to help you migrate a semantic model from Model Administration Tool to Semantic Modeler. You can migrate a semantic model from an .rpd file or from the deployed model.

Topics:

Plan Your Migration From Model Administration Tool to Semantic Modeler

Use this information to understand the steps required to migrate your semantic model from Model Administration Tool to Semantic Modeler.

Step	Description
Check the model's data source	Confirm that the model you want to migrate uses a data source that Semantic Modeler supports. Semantic Modeler only supports relational data sources. Be sure to remove or replace any unsupported data sources in the semantic model before migration. See Data Sources Available for Data Modeling.
Go to your Oracle Analytics development or test environment	Perform the semantic model migration in a non-production environment, such as an existing development or test environment, before making changes to your production environment.
Back up your environment	Use the Console to take a full snapshot of your development or test environment. You can use the snapshot to restore the environment if you discover issues after deploying the imported model. See Take a Snapshot.
Understand the differences between Model Administration Tool and Semantic Modeler	Learn about the functionality and features differences between Model Administration Tool and Semantic Modeler. See Understand the Differences Between Model Administration Tool and Semantic Modeler.
Prepare the semantic model in Model Administration Tool	Check and update the semantic model to ensure successful migration. See Prepare the Semantic Model for Migration to Semantic Modeler.

Step	Description
Import the model	Use the Semantic Modeler Create option to migrate the model. When you create the model, you have the option of importing the .rpd file into the new model, or importing the model deployed from Model Administration Tool. See Import the Semantic Model From the Model Administration Tool .rpd File or Import the Semantic Model Deployed From Model Administration Tool.
Modify the imported model and check consistency	Use Semantic Modeler to modify the migrated model and run the advanced consistency check. Learn about and try Semantic Modeler's many features. See Update the Semantic Model After Migration From Model Administration Tool and Check the Consistency of a Semantic Model.
Deploy the model	If the model is working as expected and passes the advanced consistency check, then deploy the model from Semantic Modeler. See Deploy a Semantic Model.
Revert to Model Administration Tool if necessary	If you deployed the model from Semantic Modeler and discover issues such as visualizations not displaying the correct data, then use the Console to restore your environment to the state when the snapshot was taken. If you revert your environment, any changes you made on your environment since you took the snapshot are lost. See Restore from a Snapshot.

Understand the Differences Between Model Administration Tool and Semantic Modeler

Semantic Modeler offers most of the same functionality as Model Administration Tool with some exceptions. Before you migrate the semantic model, use this topic to understand some of the differences between Semantic Modeler and Model Administration Tool.

Functionality Differences

Item	Description
Initialization block deferred execution	In Semantic Modeler, when you create an initialization block, by default its deferred execution property is set to on. See Defer Session Variable Processing.
Allow Unmapped Table property	This field is included in Model Administration Tool logical table properties user interface, but isn't include in the Semantic Modeler logical table properties user interface. When you migrate a logical table source from Model Administration Tool or add a logical table source in Semantic Modeler, this property is internally set to on.

Terminology Differences

Model Administration Tool Term	Semantic Modeler Term
aggregation content	data aggregation
BI Server	Oracle Analytics query engine
Business Model and Mapping Layer	logical layer
common enterprise information model, RPD queries	semantic model
complex join	join expression
data model	semantic model
dynamic variable	global variable
execution precedence	dependencies
flat file	This term and concept not used in Semantic Modeler.
foreign keys	join
governed data model	semantic model
ignore (Query Limit field option)	inherit
logical dimensions	logical hierarchies
Logical Table Source (user interface elements and labels)	Sources tab (fact and dimension logical tables)
Logical Table Source dialog box's Content tab	Data Granularity section located on a logical table's Sources tab
metadata repository	semantic model
opaque view	SELECT statement
Oracle BI repository	repository
Presentation layer aliases	alternative names
query limit restrictions allow and disallow	available and unavailable
repository	semantic model
repository variable	semantic model variable
Row-wise initialization (Session Variable Initialization Block Variable Target field)	Query Returns field with Variable names and values selected
RPD	semantic model .rpd file
single join	join
Status Max Rows (Query Limits field)	Row Limit
Status Max Time (Query Limits field)	Max Time
translation	localization

Prepare the Semantic Model for Migration to Semantic Modeler

Check and prepare the semantic model to ensure a successful migration.

To learn more about how Semantic Modeler handles the migration and the types of issue you might need to fix after the migration, see Update the Semantic Model After Migration From Model Administration Tool.

Item	Description
Data sources	Confirm that the semantic model uses a data sources that Semantic Modeler supports. Semantic Modeler supports only relational data sources. Be sure to remove or replace any unsupported data sources in the semantic model before migration. See Data Sources Available for Data Modeling.
Logical dimension (hierarchy) based on two logical tables	Check that any logical dimension (hierarchy) is based on only one logical table (dimension table). Semantic Modeler doesn't support logical dimensions (hierarchies) based on two logical tables. For example, a dimension table and a dimension extension logical table. To fix this issue before migration, go to Model Administration Tool and combine the two logical tables (for example, dimension table and dimension extension table) into one logical table.
Logical foreign key joins	Check if the semantic model contains logical foreign key joins. Logical foreign key joins don't exist in Semantic Modeler and won't be included with the migration. Before you migrate the model, be sure to delete the logical foreign key joins and replace them with logical joins.
Primary keys	Check that all of the semantic model's logical levels contain primary keys.
Consistency check	The semantic model must pass consistency check before migration. In Model Administration Tool, run consistency check on the model and fix any errors before migration.

Import the Semantic Model From the Model Administration Tool .rpd File

You can migrate the semantic model created in Model Administration Tool by importing the model's .rpd file into your semantic modeler development environment.

Before you begin the migration, be sure that you perform the steps required to prepare and fully migrate the model. See Plan Your Migration From Model Administration Tool to Semantic Modeler and Prepare the Semantic Model for Migration to Semantic Modeler.

Because of the difference between Model Administration Tool and Semantic Modeler, after migration you might need to update the model to ensure that it passes the advanced consistency check and functions properly. See Update the Semantic Model After Migration From Model Administration Tool.

1. On the Home page, click **Create** and then click **Semantic Model**.
2. In Create Semantic Model enter a name for the semantic model. Click **Create**.

3. In the Create Semantic Model options page, click **Import a File**.
4. In the **File Upload** dialog, browse for the .rpd file to upload. Click **Open**.
5. On the Import File window enter the password required to import the file.
6. Click **Import**.

Import the Semantic Model Deployed From Model Administration Tool

You can migrate the semantic model created in and deployed from Model Administration Tool into your semantic modeler development environment.

Before you begin the migration, be sure that you perform the steps required to prepare and fully migrate the model. See Plan Your Migration From Model Administration Tool to Semantic Modeler and Prepare the Semantic Model for Migration to Semantic Modeler.

Because of the difference between Model Administration Tool and Semantic Modeler, after migration you might need to update the model to ensure that it passes the advanced consistency check and functions properly. See Update the Semantic Model After Migration From Model Administration Tool.

1. On the Home page, click **Create** and then click **Semantic Model**.
2. In Create Semantic Model enter a name for the semantic model. Click **Create**.
3. In the Create Semantic Model options page, click **Import the Deployed Model**.

Update the Semantic Model After Migration From Model Administration Tool

Use this topic to learn how Semantic Modeler handles model migration and the types of issue you might need to fix after the migration from Model Administration Tool to Semantic Modeler.

See Understand the Differences Between Model Administration Tool and Semantic Modeler.

Item	Description
Display keys in logical dimension (hierarchy) levels	If the semantic model you migrated contained display keys for logical levels, the migrated model includes only the first display key and the remaining keys aren't included.
Data connections	Semantic Modeler only supports Data Connections. If you migrating a semantic model with a defined connection or that uses an external Console Connection, after migration you need to create a data connection and assign it to the connection pool in Semantic Modeler. See About Connections for Semantic Models and Work with Connection Pools.
Logical foreign key joins	Although Model Administration Tool doesn't support logical foreign key joins, there is a chance that your semantic model contains them. Any foreign key joins aren't included in the migrated model. To fix this issue, after migration create the needed logical joins in Semantic Modeler. See Work with Logical Joins.

Item	Description
Primary keys	Semantic Modeler requires that each logical level contains a primary key. If the migrated model doesn't contain primary keys, then add them. See Identify the Primary Key for a Dimension Level.
Consistency check	After fixing the post-migration items in this table, run the Semantic Modeler's advanced consistency check on the migrated model. Note that Semantic Modeler's consistency check enforces more data modeling best practices and checks additional rules than the Model Administration Tool consistency check. Even though the model was consistent before you migrated it, the Semantic Modeler consistency check might find errors and issues. See Work with Check Consistency.

7

Create a Semantic Model

This chapter contains information to help you understand how to create a semantic model. After you create the semantic model you can begin adding or modifying its layers.

Topics:

Create an Empty Semantic Model

Create an empty semantic model when you want to choose and import table definitions and manually build each semantic model layer and its objects.

> **Note:**
>
> You must have BI Data Model Author permissions to access and use Semantic Modeler. If you go to the **Home** page, click **Create**, and don't see the **Semantic Modeler** option, then ask your administrator to assign the BI Data Model Author permissions to you.

When you create an empty semantic model, the Semantic Modeler editor opens and an empty database is displayed in the **Physical Layer** tab. The **Connections** tab is populated with the database connections made available for semantic models.

1. On the Home page, click **Create** and then click **Semantic Model**.
2. In Create Semantic Model enter a name for the semantic model. Click **Create**.
3. In the Create Semantic Model options page, click **Start with an Empty Model**.

Import a File to Create a Semantic Model

You can import an .rpd file (exported semantic model) or .zip file (archived semantic model) file to create a semantic model in your semantic modeler development environment.

You can import an .rpd file from Model Administration Tool to create a semantic model. See Import the Semantic Model From the Model Administration Tool .rpd File.

If you're working in a Windows or Linux environment, you can use the rpdtojson utility to import a semantic model. See Generate JSON/SMML from an .rpd File.

> **Note:**
>
> You must have BI Data Model Author permissions to access and use Semantic Modeler. If you go to the **Home** page, click **Create**, and don't see the **Semantic Modeler** option, then ask your administrator to assign the BI Data Model Author permissions to you.

An imported model contains a connection, data table definitions, and semantic model layer objects that you can modify as needed.

After you import a file, the Semantic Modeler editor opens and you must:

- Review the imported model's metadata, objects, and properties to confirm that they were populated correctly.
- Confirm that the semantic model accesses the required connection. You or another user can create or share the imported semantic model's connection before or after file import. See Manage Connections to Data Sources.
- Add any needed connection pools or assign a connection to each imported connection pool. See Set a Connection Pool's Connection Property

1. On the Home page, click **Create** and then click **Semantic Model**.
2. In Create Semantic Model enter a name for the semantic model. Click **Create**.
3. In the Create Semantic Model options page, click **Import a File**.
4. In the **File Upload** dialog, browse for an exported (.rpd file) or archived (.zip file) semantic model to upload. Click **Open**.
5. Optional: If you chose an exported (.rpd file) semantic model, then in the Import File window enter the password required to import the file.
6. Click **Import**.

Import the Deployed Model to Create a Semantic Model

You can import the deployed semantic model from Oracle Analytics to create a semantic model in your semantic modeler development environment.

You can use this option when:

- You don't have access to the deployed semantic model's source files but need them to perform troubleshooting work from the Semantic Modeler editor.
- You need to migrate the semantic model created and deployed from Model Administration Tool to Semantic Modeler. For specific information, see Import the Semantic Model Deployed From Model Administration Tool.

The new semantic model contains a connection, data table definitions, and semantic model layer objects that you can modify as needed. After you import the deployed semantic model, the Semantic Modeler editor opens and you should do the following:

- Review the imported model's metadata, objects, and properties to confirm that they were populated correctly.
- Confirm that the semantic model accesses the required connection. You or another user can create or share the imported semantic model's connection before or after file import. See Manage Connections to Data Sources.

- Review the connection pool for each database and assign a data system connection to each imported connection pool. See Set a Connection Pool's Connection Property.

1. On the Home page, click **Create** and then click **Semantic Model**.
2. In Create Semantic Model enter a name for the semantic model. Click **Create**.
3. In the Create Semantic Model options page, click **Import the Deployed Model**.

Clone a Git Repository Using HTTPS

An HTTPS connection uses your Git user name and password to clone the Git repository to your development environment.

> **Note:**
>
> You must have BI Data Model Author permissions to access and use Semantic Modeler.
> If you go to the **Home** page, click **Create**, and don't see the **Semantic Modeler** option, then ask your administrator to assign the BI Data Model Author permissions to you.

Before you can use HTTPS to clone a Git repository, you must:

- Get the semantic model Git repository's URL from the developer who initialized and uploaded it.
- Know your Git user name and password to create the Git profile to authenticate to the Git repository. If you're using Github, then instead of a Git user password, you need to know your personal access token. Or choose a profile that you use with other semantic model Git repositories. See View and Manage Git Profiles.

1. On the Home page, click **Create** and then click **Semantic Model**.
2. In Create Semantic Model enter a name for the semantic model. Click **Create**.
3. In the Create Semantic Model options page, click **Clone a Git Repository**.
4. In Clone a Git Repository, enter the repository's URL. The URL must have this format: `https://gitserver.com/myorg/myproject.git`. Click **Continue**.
5. Click **Git profile** and specify the profile you want to use to clone the repository.
   - Select a Git profile that you've already used to initialize or clone a Git repository.
   - Select **New Profile** and enter a profile name and your Git user name and password to create a profile. If you're using Github, then instead of entering a Git user password, you enter your personal access token.
6. Click **Clone**.

   The Semantic Modeler editor opens and the new semantic model's layers are populated with the cloned model's metadata, objects, and properties.

Clone a Git Repository Using SSH

An SSH connection uses a deploy key that you generated in Oracle Analytics and copy into your Git account to create an SSH key. You use this key to connect to and clone a Git repository without needing to supply a Git user name and password.

> **Note:**
>
> You must have BI Data Model Author permissions to access and use Semantic Modeler.
> If you go to the **Home** page, click **Create**, and don't see the **Semantic Modeler** option, then ask your administrator to assign the BI Data Model Author permissions to you.

Before you can use SSH to clone a Git repository, you must:

- Get the semantic model Git repository's URL from the developer who initialized and uploaded it.
- Decide whether to create a Git profile or use an existing profile to authenticate to the Git repository. An existing profile is a profile that you use with other semantic models stored in Git repositories. See View and Manage Git Profiles.

1. On the Home page, click **Create** and then click **Semantic Model**.
2. In Create Semantic Model enter a name for the semantic model. Click **Create**.
3. In the Create Semantic Model options page, click **Clone a Git Repository**.
4. In Clone a Git Repository, enter the repository's URL. The URL must have this format: `git@gitserver.com:myorg/myproject.git`. Click **Continue**.
5. Click **Git profile** and specify the profile you want to use to clone the repository.
   - Select a Git profile that you've already used to initialize or clone a Git repository.
   - Select **New Profile** and enter a profile name and click **Generate Key**.
6. If you created a new Git profile and generated a key, then click **Copy Key**, go to your Git account, and use the copied key to create an SSH key. Then return to the Oracle Analytics Initialize Git wizard.
7. Click **Clone**.

   The Semantic Modeler editor opens and the new semantic model's layers are populated with the cloned model's metadata, objects, and properties.

8

Build a Semantic Model's Physical Layer

This chapter contains information to help you understand how to build a semantic model's physical layer.

Topics:

What is the Physical Layer?

The semantic model's physical layer contains objects representing physical data constructs from the back-end data sources that provide data for visualizations and reports.

The physical layer defines the objects and relationships available to the Oracle Analytics query engine for writing physical queries. This layer encapsulates data source dependencies to enable portability and federation.

Typically, each of the semantic model's data sources has its own discrete physical model in the physical layer. The top-level object in the physical layer is a database, and the type of database determines which features and rules apply to that physical model.

Physical tables, joins, and other objects in the physical layer are typically created automatically when you import metadata from the data sources. After these objects have been imported, you can perform tasks such as create additional join paths that aren't in the data source or create alias tables for physical tables that need to serve in different roles.

Create a Database and Add Tables to the Physical Layer

A database is the physical layer's highest level object and minimally contains one schema and a subset of tables and metadata from the semantic model's data source.

When you drag a table from the connection and drop it into an empty physical database, Oracle Analytics creates a physical schema and puts the table into the schema.

Dragging and dropping a table from the data source connection to the database populates some of the database's features, the query features, and connection pools. You can adjust these settings as needed. You can add query limits for application roles to the database.

Before you start adding tables to the database, you can add catalogs and schemas structure to organize the database's tables. See Add a Catalog to a Database and Add a Schema to a Database or Catalog.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**.
4. In the Physical Layer pane click **Create** and then click **Create Database**.
5. In Create Database, go to the **Name** field and type a name. Click **OK**.

   The new database's **Tables** tab is displayed.
6. Click **Connections** and in the connections pane, browse or search for the table to add to the physical database.
7. Drag the table to the physical database and drop it into the Tables list.
8. Click **Save**.

Add a Catalog to a Database

You can add catalogs to the physical layer's databases to group a semantic model's schemas and tables to mirror how the data source is organized.

You can't rearrange the catalog and schema groupings that you add. For example, if you added a schema and tables directly into a database, then you can't later create a catalog and move them to the catalog.

See Add a Schema to a Database or Catalog.

You can use a variable to dynamically populate the catalog's name. See Use a Variable to Dynamically Name a Catalog or Schema.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**.
4. Click **Create** and then click **Create Catalog**.
5. In Create Catalog, locate the **Name** field and type a catalog name.
6. In **Location**, click the menu and specify which database to add the catalog to.
7. Click **OK**.

Add a Schema to a Database or Catalog

You can add schemas to the physical layer's databases or catalogs to mirror how the data source is organized.

You can't rearrange the catalog and schema groupings that you add. For example, if you added a schema and tables directly into a database, then you can't later create a catalog and move them to the catalog.

See Add a Catalog to a Database.

You can use a variable to dynamically populate the schema's name. See Use a Variable to Dynamically Name a Catalog or Schema.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**.
4. Click **Create** and then click **Create Schema**.
5. In Create Schema, locate the **Name** field and type a schema name.
6. In **Location**, click the menu and specify which database or catalog to add the schema to.
7. Click **OK**.

Use a Variable to Dynamically Name a Catalog or Schema

You can use variables to dynamically name the physical layer's catalogs and schemas.

For example, suppose you have data for multiple clients and you structured the data source so that data for each client is in a separate catalog. In this case, you can initialize a session variable named Client that dynamically sets the name for the catalog object when a user signs into Oracle Analytics.

See About Session Variables.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**.
4. Browse for and double-click the catalog or schema that you want to add the variable to.

5. In the item's General pane click **Select**.
6. In Select Variables, locate and click the variable that you want to use. Click **Select**.
7. Click **Save**.

Change a Database Object's Database Type

When you import the physical schema into the physical layer, the database type is assigned automatically. In some cases you need to change the database type.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**.
4. Locate and double-click a database.
5. In the database's tab, click **General**.
6. Click the **Database Type** field and select a database type.
7. Click **Save**.

Modify a Database's Data Source Properties and Supported Query Features

This topic provides information about how to modify a database object's properties.

Topics:

Add or Modify a Database's Data Source Properties

Use this topic to understand and specify a database's data source properties.

These are the data source properties that you can assign to a database:

- **Virtual Private Database** - Select to identify the database source as a virtual private database (VPD). When a VPD is used, returned data results are contingent on the user's authorization credentials. Therefore, it's important to identify these sources. These data results affect the validity of the query result set that's used with caching. See About Row-Level Security.

  If you select this option, then you also should select the **Security Sensitive** option in the session variable's Variables tab.
- **Siebel CRM Database** - Select to indicate that the definition of physical tables and columns for Siebel CRM tables was derived from the Siebel metadata dictionary.
- **Allow direct database requests by default** - If this property is configured incorrectly, it can expose sensitive data to an unintended audience.

Select to allow all users to run physical queries. The Oracle Analytics query engine sends unprocessed, user-entered, physical SQL directly to an underlying database. The returned results set can be rendered in the Oracle Analytics query engine and then charted, rendered, and treated as an Oracle Analytics request.

If you want most but not all users to be able to run physical queries, select this option and use the **Query Limits** tab to limit queries for specific application roles.

- **Allow populate queries by default** - Select to allow everyone to run `POPULATE SQL`. If you want most, but not all, users to be able to run `POPULATE SQL`, select this option and then limit queries for specific users or groups.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer** and locate and double-click a database.
4. In the database's tab, click **Advanced**.
5. Go to the Data Source Properties section of the Features table and specify the database's data source properties.
6. Click **Save**.

What Are Supported Query Features?

The Oracle Analytics query engine uses the specified query features settings to determine how to query the data source. The supported query features are automatically populated with values appropriate for your semantic model's data source.

Query features are the SQL expressions, statements, function, operations, and other features that you can run against the data source such as a query that uses an `ISDESCENDANT` statement. Operations such as `ADD` or `SQRT` (square root) operations are supported.

When a supported query feature is selected or a value is specified, the data source supports the feature and the Oracle Analytics query engine pushes the function or calculation to the data source for improved performance.

When a supported query feature is deselected or no value is specified, then it isn't supported in the data source and the calculation or processing is performed in the Oracle Analytics query engine.

In most cases, you should keep the default selections and values. If you enable or change query features that the data source doesn't support, your query may return errors and unexpected results. If you disable supported SQL features, the server could issue less efficient SQL to the data source. Before you change any of the defaults, confirm that the query feature is supported by the data source.

See Modify a Database's Supported Query Features.

These are some reasons why you would update a database's query feature settings:

- If you're upgrading to a newer version of a data source. In this case, you can tailor the query features for the data source to see if the updated feature is reflected in the Oracle Analytics query engine defaults.
- If a data source supports a particular feature such as left outer join queries but you want to prohibit the Oracle Analytics query engine from sending such queries to a particular data source.

- If you have federated data sources that run functions differently. In this case, you can disable the appropriate functions so that Oracle Analytics query engine performs calculations consistently and produce correct query results.
- If you're troubleshooting a query or other operation that isn't working as expected.

Modify a Database's Supported Query Features

You can view and modify how the database's supported query features are set. The data source determines how the default query feature values are set.

In most cases, you should keep the default selections and values. If you enable or change query features that the data source doesn't support, your query may return errors and unexpected results. If you disable supported SQL features, the server could issue less efficient SQL to the data source. Before you change any of the defaults, confirm that the query feature is supported by the data source. For more information, see What Are Supported Query Features?

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer** and locate and double-click a database.
4. In the database's tab, click **Advanced**.
5. Go to the Supported Query Features section of the Features table and modify the database's query features as needed.
6. Click **Save**.

Work with Connection Pools

This topic provides information about how to create and modify a database's connection pools.

Topics:

What Are Connection Pools?

The semantic model's physical layer contains at least one connection pool for each database. These connection pools are configured to enhance the execution of commands between the Oracle Analytics query engine and the semantic model's database data source.

A connection pool is automatically created when you import tables into a database object in the physical layer. You can add and configure multiple connection pools for each database. Connection pools allow multiple concurrent data source requests (queries) to share a single database connection, reducing the overhead of connecting to a database. Oracle recommends that you create a dedicated connection pool for initialization blocks. See About Connection Pools for Initialization Blocks.

For each connection pool, you must specify the maximum number of concurrent connections allowed. After this limit is reached, the connection request waits until a connection becomes available.

Increasing the allowed number of concurrent connections can potentially increase the load on the underlying database accessed by the connection pool. Test and consult with the database administrator to make sure the data source can handle the number of connections specified in the connection pool. Also, if the data sources have a charge back system based on the number of connections, you might want to limit the number of concurrent connections to keep the charge-back costs down.

In addition to the potential load and costs associated with the database resources, the Oracle Analytics query engine allocates shared memory for each connection upon server startup. This raises the number of connections and increases the Oracle Analytics query engine memory usage.

About Connection Pools for Initialization Blocks

You should create a dedicated connection pool for initialization blocks. Don't use the connection pools that you create for initialization blocks for data queries.

You should isolate the connections pools for different types of initialization blocks. By isolating the connection pools, you can ensure that authentication and login-specific initialization blocks don't slow down the login process. The following types of initialization blocks should have separate connection pools:

- All initialization blocks that set session variables.
- All initialization blocks that set semantic model variables. Run initialization blocks that set variables using credentials with administrator privileges.

  Be aware of the number of these initialization blocks, their scheduled refresh rate, and when they're scheduled to run. It would take an extreme case for this scenario to affect resources. For example, refresh rates set in minutes, greater than 15 initialization blocks that refresh concurrently, and a situation in which either of these scenarios could occur during prime user access time frames.

It's more efficient and less resource intensive to set as many variables as possible in an initialization block. For example, suppose you have one initialization block that contains five variables. In this case, the initialization string makes one call to the back-end tables. Creating five initialization blocks that contain one variable each results in five calls to the back-end tables.

If an initialization block fails for a particular connection pool, no more initialization blocks using that connection pool are processed. Instead, the connection pool is denied and subsequent initialization blocks for that connection pool are skipped. This behavior ensures that Oracle Analytics continues to work, even when a connection pool has a large number of associated initialization blocks or variables.

If this issue occurs, a message similar to the following is displayed in the server log:

```
[OracleBIServerComponent] [ERROR:1] [43143] Blacklisted connection
pool name_of_connection_pool
```

If you see this error, check the initialization blocks for the given connection pool to ensure they're correct.

Connection Pool General Properties

The topic describes the connection pool properties in the Connection Pool's tab General pane. These properties are common among most connection types.

Use the information in this topic to help you create or modify a connection pool. See Set a Connection Pool's General Properties.

Property	Description
Connection	Displays the connections available to semantic models. A connection's **System Connection** property must be selected for the connection to display in this list. See Manage Connections to Data Sources. Oracle Analytics doesn't always automatically assign the connection pool's connection, so sometimes you must manually assign one. You're not able to preview a physical table's data until it's database's connection pool connection is assigned.
Remote Connection	Identifies if the database connection uses remote data connectivity. This field isn't automatically selected if the database uses remote data connectivity. You must set this field manually. If the database uses remote data connectivity and this field isn't selected then you'll receive an error when you run consistency check.
Max Connections	Specifies the maximum number of connections allowed for this connection pool. The default is 10. You can determined the value by the database make and model and the configuration of the hardware for the computer where the database runs, and the number of concurrent users who require access. For deployments with Oracle BI Interactive Dashboards pages, consider estimating this value at 10% to 20% of the number of simultaneous users multiplied by the number of requests on a dashboard. You can adjust the number based on usage. Define the total number of all connections in the semantic model to less than 800. To estimate the maximum connections needed for a connection pool dedicated to an initialization block, you might use the number of users concurrently logged on during initialization block processing.
Timeout	Specifies the amount and increment of time that a connection remains open after a request completes. During this time, new requests use this connection rather than open a new one up to the number specified for the maximum connections. The time is reset after each completed connection request.

Property	Description
Isolation Level	Specifies the value sets the transaction isolation level on each connection to the back-end database. For ODBC gateways only. Controls the default transaction locking behavior for all statements issued by a connection. You can only set one at a time. It remains set for that connection until it's explicitly changed. The options are: **Dirty read** - Implements dirty read, isolation level 0 locking. This is the least restrictive isolation level. When this option is set, it's possible to read uncommitted or dirty data, change values in the data, and have rows appear or disappear in the data set before the end of the transaction. Dirty data is data to clean before running a query to obtain correct results. For example, duplicate records, records with inconsistent naming conventions, or records with incompatible data types. **Committed read** - Specifies that shared locks are held while the data is read to avoid dirty reads. You can change the data before the end of the transaction, resulting in non-repeatable reads or phantom data. **Repeatable read** - Places locks on all data that's used in a query, preventing other users from updating the data. You can insert new phantom rows into the data set by another user and are included in later reads in the current transaction. **Serializable** - Places a range lock on the data set, preventing other users from updating or inserting rows into the data set until the transaction is complete. This is the most restrictive of the four isolation levels. Because concurrency is lower, use this option only if necessary.

Property	Description
Require fully qualified table names	When selected, specifies that all requests sent from the connection pool use fully qualified names to query the underlying database. Select this option if the database or database configuration requires fully qualified table names. This option isn't available for some connection types. The fully qualified names are based on the physical object names in the semantic model. If you're querying the same tables that the physical layer metadata was imported from, then you can safely select this option. If you've migrated your semantic model from one physical database to another physical database that has different database and schema names, the fully qualified names are invalid in the newly migrated database. In this case, if you don't select this option, the queries succeed against the new database objects. For some connections, fully qualified names are a safer because they guarantee that the queries are directed to the desired tables in the desired database. For example, if the RDBMS supports an original database concept, a query against a table named Customer first looks for that table in the original database, and then looks for it in the specified database. If the table named Customer exists in the original database, that table is queried, not the table named Customer in the specified database. You might need to select this option when you're using an Oracle Database and you're accessing the database with a user that isn't the owner of the schema containing the tables. When the Oracle Database interprets table names in SQL, it assumes that the user that made the query is the owner if the table name isn't fully qualified in the query. This can result in an incorrect qualified name. For example, if the user SAMPLE creates a table called CUSTOMER, the fully qualified table name is SAMPLE.CUSTOMER. When the SAMPLE user references the CUSTOMER table in a query, the Oracle Database assumes the fully qualified table name is SAMPLE.CUSTOMER, and the access is successful. However, if the JANEDOE user references the CUSTOMER table in a query, the Oracle Database assumes the fully qualified table name is JANEDOE.CUSTOMER, and a Table or view not found error can result. To enable access for JANEDOE, you must select **Require fully qualified table names** in the connection pool so that the Oracle Analytics query engine specifies SAMPLE.CUSTOMER in all queries.
Use multithreaded connections	When selected, specifies that the Oracle Analytics query engine terminates idle physical queries (threads). When not selected, one thread is tied to one database connection, number of threads = maximum connections. Even if threads are idle, they consume memory.
Parameters supported	When selected, indicates that the database features table supports parameters and special code runs that allows the Oracle Analytics query engine to push filters (or calculations) with parameters to the database. The Oracle Analytics query engine does this by simulating parameter support within the gateway/ adapter layer by sending extra SQLPrepare calls to the database.
Enable connection pooling	Allows a single database connection to remain open for the specified time for use by future query requests. Connection pooling saves the overhead of opening and closing a new connection for every query. If you don't select this option, each query sent to the database opens a new connection.

Set a Connection Pool's General Properties

General properties include Max Connections, Timeout, Isolation Level, and so on.

For description of the general properties and how to set them, see Connection Pool General Properties.

The properties listed in the General tab vary according to the data source type.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer** and locate and double-click a database.
4. In the database's tab, click **Connection Pools**.
5. In the connection pools list table, click a connection pool to select it and then click **Detail view** to open the Properties pane.
6. Go to the General section of the Properties pane and modify the connection pool's properties.
7. Click **Save**.

Set a Connection Pool's Connection Property

A connection pool's connection must be correctly assigned before you can preview a physical table's data.

When you import a .rpd or .zip file or load the deployed semantic model to create a semantic model, Oracle Analytics doesn't always automatically assign the connection pool's connection. In such cases you must manually assign one.

The **Connections** field displays the data source connections that you can use with semantic models. You must create or have been given access to the needed semantic model's data source connection. See Manage Connections to Data Sources.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer** and locate and double-click a database.
4. In the database, click **Connection Pools** and click **Detail view**.
5. Go to the connections pool list and click a connection pool to display its details.
6. In the connection pool's details, scroll to the **Connection** field, click it, and select a connection.
7. Click **Save**.

Add Connection Scripts to a Connection Pool

You can add one or more connection scripts and set them to run before the connection is established, before a query is run, after a query is run, or after the connection is disconnected.

For example, you can create a connection script that on connect inserts the name of the user and the connection time into a table.

Connection scripts can contain any commands accepted by the database, such as a command to turn on quoted identifiers. This enables mainframe environments to maintain security in one central location.

Because the connection script is sent directly to the data source, you must write the script in native SQL. Don't write the script in Oracle Analytics Logical SQL because the data source won't understand it.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer** and locate and double-click a database.
4. In the database's tab, click **Connection Pools**.
5. In the connection pools list table, click a connection pool to select it and then click **Detail view** to open the properties pane.
6. Scroll to Connection Scripts, click **Add Script**, and select when to run the script.

   A subsection is added for the selected connection script type.
7. Write the script using native SQL or a language that the data source understands.
8. Optional: Click **Enable** so that the script runs before the connection is established.
9. Click **Save**.

About Setting the Bulk Insert Buffer Size and Transaction Boundary Settings

For write back, if each row size in a result set is 1 KB and the buffer size is 20 KB, then the maximum array size is 20 KB.

If there are 120 rows, there are 6 batches with each batch size limited to 20 rows.

If you set **Transaction boundary** to 3, the server commits twice. The first time, the server commits after row 60 (3 * 20). The second time, the server commits after row 120. If there is a failure when the server commits, the server only rolls back the current transaction. For example, if there are two commits and the first commit succeeds but the second commit fails, the server only rolls back the second commit.

For optimum performance, consider setting the buffer size to 128 and the transaction boundary to 1000.

Set up Write Back in a Connection Pool

A connection pool's write back requirements include a temporary table and bulk insert properties.

See About Setting the Bulk Insert Buffer Size and Transaction Boundary Settings.

The table describes the properties in the Write Back tab of the Connection Pool dialog.

Property	Description
Database supports unicode	Select when the columns are of an explicit Unicode data type, such as `NCHAR`, in a Unicode database. This makes sure that the binding is correct and that data is inserted correctly. Different database vendors provide different character data types and different levels of Unicode support. Use these guidelines to determine when to set this option: • On a database where `CHAR` data type supports Unicode and there isn't a separate `NCHAR` data type, don't select this option. • On a database where `NCHAR` data type is available, it's recommended to select this option. • On a database where `CHAR` and `NCHAR` data type are configured to support Unicode, it's option to select this option. Unicode and non-Unicode data types can't coexist in a single non-Unicode database. For example, mixing the `CHAR` and `NCHAR` data types in a single non-Unicode database environment isn't supported.
Temporary Table - Prefix	Enter the first two characters in the temporary table name that the Oracle Analytics query engine creates.
Temporary Table - Owner	Enter the table owner name used to qualify a temporary table name in a SQL statement. For example, `owner.tablename`. If left blank, the user name specified in the writeable connection pool is used to qualify the table name.
Bulk Insert - Buffer size (KB)	Enter the maximum number of bytes inserted into a database table. For optimum performance, set this parameter to 10240.
Bulk Insert - Transaction boundary	Enter the batch size for an insert in a database table. For optimum performance, set this parameter to 1000.

Use these steps to specify your database's write back properties.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer** and locate and double-click a database.
4. In the database's tab, click **Connection Pools**.
5. In the connection pools list table, click a connection pool to select it and then click **Detail view** to open the properties pane.
6. Scroll to Write Back and specify the write back properties.
7. Click **Save**.

Set a Connection Pool's Permissions

A connection pool's permissions specify which application roles have read-write, read-only, and no access permissions to use the connection pool. For example, you can set up the users in the DV Content Author application role to have their own connection pool.

By default, all roles have read-only access to the connection pool. Add applications roles and assign permissions to limit who can use the connection pool.

Don't use connection pool permissions to determine data access security. For example, connection pool permissions don't protect cache entries.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer** and locate and double-click a database.
4. In the database's tab, click **Connection Pools**.
5. In the connection pools list table, click a connection pool to select it and then click **Detail view** to open the properties pane.
6. Scroll to Permissions.
7. Click in the search field and type the name of a role, or enter * (an asterisk) to see the full list of roles. From the results list, click the role that you want to add to the permissions table.
8. Specify the role's permission.
9. Click **Save**.

About Physical Tables

A physical table is an object that represents a data source's table. You can configure tables as physical tables, select tables, or stored procedure tables.

You can drag and drop tables from the data source into the physical layer's database. These imported tables contain metadata and joins from the data source. This metadata enables the Oracle Analytics query engine to use a SQL request to access the corresponding data source table.

You can also create virtual physical tables. Virtual tables provide the Oracle Analytics query engine and the underlying data sources with the proper metadata to perform advanced query requests.

You can store a virtual physical table as a stored procedure or a `SELECT` statement. You can define a table from a select statement and deploy it in the data source to create a deployed view.

For more information about the table source types you assign to physical tables, see What Are a Physical Table's General Properties?

What Are a Physical Table's General Properties?

This topic contains information about the properties that you assign to your imported, added, or aliased physical tables.

See Create a Physical Table and Populate Physical Columns with a Stored Procedure or Select Statement.

Property	Description
Source	Specifies how the physical table's columns get their data. For an alias table, you can't change the table's source option, but you can click **Replace...** to change the source table. For an imported or newly added table, select **Table** if you need to add columns to match those in a corresponding data source table. You might use this options when you need to add physical columns because a data source's table isn't finalized and available for import, or if the administrator has added more columns to a data source's table. After you select this option, you use the **Columns** tab to create the needed columns. For an imported or added table, select **Stored Procedure** or **Select Statement** to use a stored procedure or select statement to populate the physical table's columns. After you select this option, you use the **Columns** tab to write the default or database-specific stored procedure or select statement and to create the needed columns. See Populate Physical Columns with a Stored Procedure or Select Statement.
Dynamic Name	Displays the name of the session variable used to name the table. This option is available if you selected Table in the **Source** field. Available for imported or added tables. You can choose **Use Dynamic Name** to select between primary and shadow tables that are valid at different times in the ETL cycle. In both cases, you can assign session variables to dynamically select the appropriate table.
Caching	Specifies if and how the table's data is cached. Typically you cache data when the table doesn't need to be accessed in real time. Select **Same cache setting as source** so that the alias table uses the same caching preference as its source table. If you select this option then the source's caching option is displayed next to the field. For example, (Cache forever). Select **Do not cache** to not cache the table. Select **Cache forever** so that the table entry cache doesn't automatically expire. This option is useful when a table is important to a large number of queries that users might run. For example, if most queries have a reference to an account object, keeping it cached indefinitely could actually improve performance rather than compromise it. Selecting this option doesn't mean that an entry always remains in the cache. Other invalidation techniques, such as manual purging, LRU (Least Recently Used) replacement, metadata changes, or use of the cache polling table can result in entries being removed from the cache. Select **Cache for** to specify how long the table entries are persisted in the query cache. Setting a cache persistence time is useful for data sources that are updated frequently. For example, you could set this option to refresh the underlying physical tables daily for a particular workbook or dashboard. If a query references multiple physical tables with different persistence times, the cache entry for the query exists for the shortest persistence time set for any of the tables referenced in the query. This makes sure that no subsequent query gets a cache hit from an expired cache entry.
SQL Hint	Contains instructions that tell the data source query optimizer the most efficient way to run the SQL statement. See About Hints in SQL Statements.
Join Keys	Displays the table's keys that are used in joins to other physical tables. Join keys are automatically created when you import joined data source tables and when you create or modify physical table joins. Use the physical diagram to update joins. See About Physical Joins.
Additional Keys	Displays a list of keys that, in addition to the join keys, defines identifier columns for the table.

Disable Auto Joins Creation in the Physical Layer

Disable the **Automatically create joins if tables added to the physical layer have foreign keys** user preference to prevent Oracle Analytics from automatically creating physical joins.

> **Note:**
>
> Deselecting **Automatically create joins if tables added to the physical layer have foreign keys** disables the option for all semantic models that you work with, not just the semantic model that was open when you set this user preference.

By default the **Automatically create joins if tables added to the physical layer have foreign keys** user preference is set to on. When you add data source tables to the physical layer, any foreign keys defined in the data source automatically create joins between the corresponding tables in the physical layer.

In some cases you might want to manually build the physical layer's joins. For example, if adding data source tables creates unneeded or incorrect joins in the physical layer and it's time consuming for you to delete these joins. In such cases you deselect the **Automatically create joins if tables added to the physical layer have foreign keys** checkbox to turn off automatic joins.

Deselecting **Automatically create joins if tables added to the physical layer have foreign keys** doesn't remove the physical layer's existing joins, so you must remove those joins manually.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Page Menu** and then click **Preferences**.
4. In the Users Preferences dialog box, scroll to **Physical Layer** and then click **Automatically create joins if tables added to the physical layer have foreign keys** to deselect it.
5. Click **Apply**.

Create a Physical Table

Manually create a physical table when you can't import a data source table. You create a table to mirror a table in the data source, or to use a select statement or stored procedure to populate the physical columns you specify.

For information about how to set the table properties, see What Are a Physical Table's General Properties?

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**.

4. In the Physical Layer pane click **Create** and then click **Create Physical Table**.
5. In Create Physical Table, go to the **Name** field and enter a table name. Then go to **Location** and browse for and select the new table's location.
6. Click **OK**.
7. In the new table's tab, click **General** and specify the table's properties.
8. Click **Columns** and depending on the source you specified, add the columns or enter a stored procedure or select statement to populate the columns.
9. Click **Save**.

Create or Modify a Physical Column

You can create a new column in a physical table, or update an existing or imported column's properties. Physical columns store data within tables in the physical database.

Remember that if you create, modify, or delete the physical table's columns, then any alias tables that use the physical table as its source reflect those changes.

When you create or update a physical table, you can set its **Source** property to use a stored procedure or select statement to populate columns. See Populate Physical Columns with a Stored Procedure or Select Statement.

Use this information to help you set a column's properties:

- **Type** - Indicates the column's data type. Use caution when changing the data type. Setting the values to data types that are incorrect in the underlying data source might cause unexpected results. If there are any data type mismatches, correct them in the semantic model or reimport the columns that have mismatched data types.

  If you reimport columns, you also need to remap any logical column sources that reference the remapped columns. The data type of a logical column in the business model must match the data type of its physical column source. The Oracle Analytics query engine passes these logical column data types to client applications.

  `Longvarchar` and `longvarbinary` data types are supported for writing complete Logical SQL statements into usage tracking tables for debugging purposes. They aren't supported for general-purpose queries, and can't be displayed in Oracle BI Server.
- **Nullable** - Specifies whether null values are allowed for the column. If null values can exist in the underlying table, you need to select this option. This allows null values to be returned to the user, which is expected with certain functions and with outer joins. It's generally safe to change a non-nullable value to a nullable value in a physical column.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**.
4. In the database pane, browse for and double-click the table where you want to add or modify a column.
5. In the physical table, click the **Columns** tab.
6. Click **Add Column** and select **Create New Column** to create a new column in the physical table. Or in the table, click a column to highlight it, and then click it again to enable the fields to be updated.

7. Optional: To duplicate a column, hover over the column you want to duplicate and click its **Row Menu** and click **Duplicate**.
8. Specify the column's properties and then click off of the column.
9. Click **Save**.

Populate Physical Columns with a Stored Procedure or Select Statement

If you chose the stored procedure or select statement as the physical table's source type, then you create physical columns and write a stored procedure or select statement to populate them.

Preview isn't available for columns populated by a stored procedure.

Use this information to help you write a stored procedure or select statement:

- **Stored procedure** - Provides a default stored procedure and database-specific stored procedures. Requests for this table call the stored procedure. The default initialization string is run when the queried database type doesn't have a corresponding database-specific string defined.

  Stored procedures within an Oracle Database might not return result sets. You can't initiate stored procedures from within Analytics Cloud. You need to rewrite the procedure as an Oracle function, use the Oracle function in a `SELECT` statement in the initialization block, and associate the Oracle function with the appropriate session variables.

  The following example shows a SQL initialization string using the `GET_ROLES` function that's associated with the `USER`, `GROUP`, and `DISPLAYNAME` variables. The function takes a user Id as a parameter and returns a semicolon-delimited list of group names:

  ```
  SELECT user_id, get_roles(user_id), first_name || ' ' || last_name
  FROM csx_security_table
  WHERE user_id = ':USER' and password = ':PASSWORD'
  ```

- **Select statement** - Provides a default `SELECT` statement and a `SELECT` statement for any databases that you select. You need to manually create the table columns. The column names must match the ones specified in the `SELECT` statement. Column aliases are required for advanced SQL functions, such as aggregates and `CASE` statements.

  The default `SELECT` statement is run when the queried database type doesn't have a corresponding database-specific `SELECT` statement defined.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**.
4. In the database pane, browse for and double-click the table where you want to add the stored procedure or select statement.
5. In the physical table, click the **General** tab and in the **Source** field, select either **Stored Procedure** or **Select Statement**.
6. Click the **Columns** tab.

7. Click **Add Column** and select **Create New Column** to add the new physical column needed to store the data. Add more columns as needed.
8. Depending on the source you selected, go to the **DEFAULT** field and enter your default stored procedure or select statement.
9. Optional: To add stored procedures or select statements written for specific database types, click **Specify query for additional databases** and then click **Add additional databases** and click to select the needed databases.
10. Optional: In the list of databases, click a database and enter its required stored procedure or select statement.
11. Click **Save**.

About Physical Alias Tables

An alias table is a physical table that uses an alternative name to references another physical table as its source. Creating alias tables lets you to reuse an existing table more than once so you don't have to import the table into the physical layer several times.

The primary reasons to use alias tables are:

- To set up multiple tables, each with different keys, names, or joins, when a single data source table needs to serve in different semantic roles. Setting up alias tables in this case is a way to avoid triangular or circular joins.

  For example, suppose you have a fact table in which the order date and shipping date both point to the same column in the time dimension data source table. In this case you can alias the dimension table so that each role is presented as a separately labeled alias table with a join. These separate roles carry over into the business model, so that *Order Date* and *Ship Date* are part of two different logical hierarchies. If a single logical query contains both columns, the physical query uses aliases in the SQL statement so that it can include both of them.

  You can also use aliases to enable a data source table to play the role of both a fact table and a dimension table that joins to another fact table, often called a *fan trap*.

- To include best practice naming conventions for physical table names. For example, you can prefix the alias table name with the table type such as fact, dimension, or bridge, and not change the original physical table names. Some organizations create alias tables for all physical tables to enforce best practice naming conventions. In this case, all mappings and joins are based on the alias tables rather than the original tables.

Alias table names appear in physical SQL queries. Using alias tables to provide meaningful table names can make SQL queries referencing those tables easier to read. For example:

```
WITH
SAWITH0 AS (select sum(T835.Dollars) as c1
from
     FactsRevT835/*AllRevenue(Billed Time Join)*/)
select distinct 0 as c1,
     D1.c1 as c2
from
     SAWITH0 D1
order by c1
```

In this query, the meaningful alias table name *A11 Revenue (Billed Time Join)* has been applied to the terse original physical table name *FactsRev*. In this case, the alias table name

provides information about which role the table was playing each time it appears in SQL queries.

Alias tables can have cache properties that differ from their original tables.

Because the alias table's columns are automatically synchronized with the original table, you can't add, delete, or modify columns in an alias table. Synchronization ensures that the original tables and their related alias tables have the same column definitions. For example, if you delete a column in the original table, the column is automatically removed from the alias table.

You can't delete an original table unless you delete all of its alias tables first. Alternatively, you can select the original table and all its alias tables and delete them at the same time.

You can change an alias table's original table if a new original table is a superset of the current original table. However, this could result in an inconsistent semantic model if changing the original table deletes columns that are being used. Running consistency check identifies orphaned aliases.

Alias tables inherit some properties from their original tables. Some of the properties that the alias table gets from the original table can't be changed in the alias table. Such properties are grayed out in the alias table properties. If the original table changes its value for a grayed out property, the same property change displays for the alias table.

Create an Alias Table

An alias table uses another physical table as its source. You can use an alias table to create an alternative name for its source table. For example, if you imported a table named SAMP_TIME_DAY from the data source, you can create an alias table named Time Day Grain.

You can change some of the source table properties that the alias table inherits, such as its source table and caching method. You can also use the alias table to create joins. See About Physical Alias Tables and What Are a Physical Table's General Properties?

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**.
4. In the Physical Layer pane click **Create** and then click **Create Physical Table Alias**.
5. In Create Physical Table Alias, go to the **Name** field and enter a table name. Then go to **Source** and browse for and select the source table you want the alias table to use.
6. Click **OK**.
7. In the new alias table, click **General** and updated the table's properties.
8. Click **Save**.

Open the Physical Diagram from the Physical Layer

The physical diagram provides a graphic view of the physical table or tables that you selected. From the diagram you can view a table's columns, show direct joins, and add, modify, or delete physical joins.

You can select what you want the diagram to contain:

- **Selected Tables Only** - Displays only the selected physical tables. Physical joins display only if they exist between the tables that you selected.
- **Selected Tables and Direct Joins** - Displays the selected physical tables and any physical tables that join to the table or tables that you selected.

You can open the physical diagram from the physical layer or the logical layer. Opening the physical diagram from the logical layer helps you understand the model's logical-to-physical mappings, and shows you the physical objects that are associated with a particular logical object.

For information about adding or modifying joins from the physical diagram, see Add and Define Physical Joins.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**.
4. Locate and right-click a table, or use Shift click or Ctrl click to select more than one table and right-click.
5. Hover over **Show Physical Diagram** and click **Selected Tables Only** or **Selected Tables and Direct Joins**.
6. In the physical diagram, double-click a table to view a list of column names with column type icons. Double-click the list to collapse it.
7. Right-click a table and select **Show Direct Joins** to display the tables that join to the table.
8. Click a join to access the Edit Physical Join dialog box to view or modify the join's properties.
9. Click a table to select it and on the right side of the table, grab its handle and drag to another table to create a physical join and specify its properties.
10. Click **Add**.
11. Click **Save**.

Delete a Physical Table

Deleting a physical table also deletes its dependent objects. For example, columns, keys, and joins.

You can't delete a table that an alias table is using as its source. You must first delete the alias table.

Instead of browsing to the physical table that you want to delete, you can search for it.

1. On the Home page, click **Navigator** and then click **Semantic Models**.

2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**.
4. In the physical layer pane, locate and right-click the table you want to delete. Click **Delete**.
5. When prompted, click **Save All**.

Delete a Physical Column

Deleting a physical column deletes it from any physical alias tables that include the column.

Instead of browsing to the physical column that you want to delete, you can search for it.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**.
4. In the database pane, browse for and double-click the table with the column you want to delete.
5. In the physical table, click the **Columns** tab.
6. Locate the column that you want to delete, click **Row Menu** and click **Delete**.

Work with Physical Joins

This topic provides information about how to create and modify physical joins.

Topics:

About Physical Joins

Physical joins indicate relationships between physical tables and tell the Oracle Analytics query engine how to retrieve data from the tables.

Creating a physical join automatically creates a join key for the identifier column and adds it to the primary table's properties. To view a table's join keys, open the physical table and then go to the **General** tab. You can view join keys, but not edit them. To find more information about a join, open the Physical Diagram and click a join to find out the tables it's joined to, join cardinality, join type, join condition, and so on.

You must explicitly define joins in each data source or between data sources to express relationships between tables in the physical layer. You can create a join can based on a join condition, or on an expression that you provide. For most data sources, join conditions are preferred for performance reasons. Joins based on expressions usually don't perform as well because they don't use key column relationships to form the join.

Joins that are defined to enforce referential integrity constraints can result in specifying incorrect joins in queries. For example, joins between a multipurpose lookup table and several other tables can result in unnecessary or invalid circular joins in the SQL queries issued by the Oracle Analytics query engine.

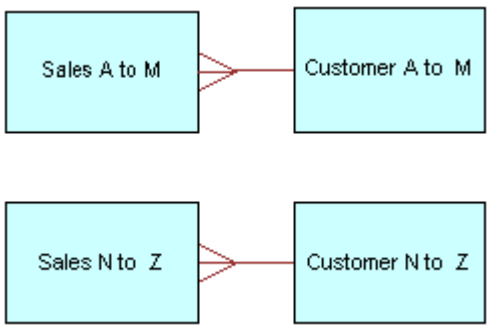
You can define a join from one metadata database object to another metadata database object. This is called a multi-database join.

While the Oracle Analytics query engine has several strategies for optimizing the performance of multi-database joins, these joins are significantly slower than joins between tables within the same database. As a result of the negative performance impact, you should avoid using multi-database joins whenever possible.

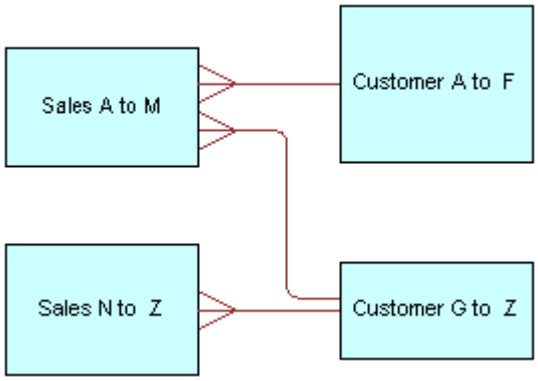
About Joining Fragmented Data

Fragmented data is data for a single entity that is split between multiple tables.

For example, a data source might store sales data for customers with last names beginning with the letter A through M in one table and last names from N through Z in another table. With fragmented tables, you need to define all of the join conditions between each fragment and all the related tables. The figure shows the physical joins with a fragmented sales table and a fragmented customer table where the data are fragmented the same way (A through M and N through Z).



You could have a fragmented fact table and a fragmented dimension table with fragments across different values. You create the joins to the fragmented table and define a one-to-many join, as shown in the Customer A to F and from Customer G to Z to Sales A to M example.



> **Note:**
>
> Avoid adding join conditions where they aren't necessary, for example, between Sales A to M and Customer N to Z. Extra join conditions can cause performance degradations.

Add and Define Physical Joins

You can define joins between physical tables in the same data source, or you can define joins between physical tables in different data sources. You use the Physical Diagram to add and define joins.

Semantic Modeler determines what type of join to create based on the selected object types and the join expression.

See About Physical Joins and Use Hints in SQL Statements.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**.
4. Click a table, right click, and select **Show Physical Diagram** and select **Selected Tables Only**.
5. Drag and drop additional tables to the Physical Diagram.
6. Working in the Physical Diagram, hover over the first table in the join (the table representing many in the one-to-many join.), grab its handle, and drag to the table that you want to join to (the table representing one in the one-to-many join).

   A box is displayed around the table that you are joining to.
7. In the Add Physical Join dialog box, specify one or more join conditions, or click **Use Join Expression** to specify the join properties and enter the join expression.

   If using Unknown cardinality, then you only need to select *Unknown* for one side of the join. For example, choosing unknown-to-1 is equivalent to unknown-to-unknown and appears as such the next time you open the dialog box for this join.
8. Click **Add**.

Use Hints in SQL Statements

This topic provides information about how to add SQL hints to physical tables and physical joins.

Topics:

- Create Physical Join Hints

About Hints in SQL Statements

Hints are instructions that you add to a SQL statement that tell the data source query optimizer the most efficient way to run the statement.

Hints override the optimizer's processing plan, so you can use hints to improve performance by forcing the optimizer to use a more efficient plan. Hints are only supported for Oracle Database data sources.

You can add hints to a physical table or a join expression. When the object associated with the hint is queried, the Oracle Analytics query engine inserts the hint into the SQL statement.

For physical tables, SQL hints you specify for tables with the source type **Table** are supported, but SQL hints that you specify for tables with the source type **Stored Procedure** or **Select Statement** are ignored. For tables with the source type of **Select Statement**, you can provide the hint text as part of the SQL statement you enter in the **DEFAULT** field.

About the Index Hint

An Index hint explains how the optimizer scans a specified index rather than a table.

See Oracle hints in the SQL reference guide for the version of the Oracle Database that you use.

This is the syntax for the Index hint:

```
index(table_name,index_name)
```

For example, suppose queries against the `ORDER_ITEMS` table are slow and you've reviewed the processing plan of the query optimizer and discovered that the `FAST_INDEX` wasn't used. You can create an Index hint to force the optimizer to scan the `FAST_INDEX` rather than the `ORDER_ITEMS` table.

For this example, you add this syntax to the physical table's **SQL Hints** properties field:

```
index(ORDER_ITEMS, FAST_INDEX)
```

About the Leading Hint

A Leading hint forces the optimizer to build the join order of a query with a specific table.

See Oracle hints in the SQL reference guide for the version of the Oracle Database that you use.

This is the syntax for the Leading hint:

```
leading(table_name)
```

For example, suppose you have a join between the Sales Fact table and the Products table and want to force the optimizer to begin the join with the Products table.

For this example, you add this syntax to the physical join's **Include Hint** field:

```
leading(Products)
```

Performance Considerations for SQL Statement Hints

Well researched and planned SQL statement hints can result in significantly better query performance. However, hints can also negatively affect performance if they result in a suboptimal processing plan.

Follow these guidelines to create hints to optimize query performance:

- Only add hints to a semantic model after you've tried to improve performance in the following ways:
  - Adding physical indexes or other physical changes to the Oracle Database.
  - Making modeling changes within the server.
- Avoid creating hints for physical table and join objects that are queried often. If you drop or rename a physical object that's associated with a hint, you must also update the hints accordingly.

Create Physical Table Hints

You can add SQL hints to a physical table. You can't add hints to an alias table, but only to its source table.

Although hints are identified using SQL comment markers (/* or --), don't type SQL comment markers when you type the text of the hint. The Oracle Analytics query engine inserts the comment markers when the hint is run.

For a description of available Oracle hints and hint syntax, see SQL reference for the version of the Oracle Database that you use.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**.
4. In the database pane, browse for and double-click the table where you want to add a SQL hint.
5. Click the General tab, and in the **SQL Hints** field enter the SQL hint.
6. Click **Save**.

Create Physical Join Hints

You can add SQL hints to a physical join in an alias table.

Although hints are identified using SQL comment markers (/* or --), don't type SQL comment markers when you type the text of the hint. The Oracle Analytics query engine inserts the comment markers when the hint is run.

For a description of available Oracle hints and hint syntax, see SQL reference for the version of the Oracle Database that you use.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**.

4. Hover over a table and right-click. Select **Show Physical Diagram** and the diagram type you want to work from.
5. Double-click a join.
6. In Edit Physical Join, click the **Include Hint** field and enter the SQL hint.
7. Click **Save**.

Preview Data in Physical Tables

You can preview a physical table's data.

Before you can preview a table's data, the database that contains the table must have a connection pool connection assigned to it. See Set a Connection Pool's General Properties.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**.
4. In the database pane, browse for and double-click the table that you want to preview data for.
5. In the physical table, click the **Preview** tab.

9

Build a Semantic Model's Logical Layer

This chapter contains information to help you understand how to build a semantic model's logical layer.

Topics:

What is the Logical Layer?

The semantic model's logical layer defines the dimensional business model of the data and specifies the mapping between the business model and the physical layer schemas.

The logical layer determines the analytic behavior seen by users, and defines the superset of objects and relationships available to users. The logical layer hides the complexity of the source data models.

The logical layer can contain more than one business model, but ideally a single, integrated business model is preferred to provide common dimensions used to analyze data across subject areas. Business models are always dimensional, unlike objects in the physical layer, which reflect the organization of the data sources. Each business model contains logical tables, logical columns, logical table sources, and logical hierarchies.

Even though similar terminology is used for logical table and physical table objects, such as tables and joins, objects in the logical layer have their own set of rules that differ from those of relational models. For example, logical joins can represent many possible physical joins.

Tables, joins, mappings, and other objects in the logical layer are typically created automatically when you drag and drop objects from the physical layer to a business model. After these objects have been created, you can perform tasks like create additional logical

joins, perform calculations and transformations on columns, and add and remove joins from dimension and fact tables.

Each column in the logical layer maps to one or more columns in the physical layer. At run time, the Oracle Analytics query engine evaluates Logical SQL requests against the logical layer, and then uses the mappings to determine the best set of physical tables and files for generating the necessary physical queries. The mappings often contain calculations and transformations, and might combine multiple physical tables.

Automatically Rename Logical Layer Objects

Use the **Automatically rename objects when added to logical layer** user preference to automatically rename the physical tables and columns that you add to the logical layer.

> **Note:**
>
> Setting **Automatically rename objects when added to logical layer** preferences applies to all semantic models that you work with, not just the semantic model that was open when you set this preference.

When you select this option, you also specify how you want the tables and column renamed. For example, choose the **Change each underscore(_) to a space** rename action and then choose the **All lowercase** rename action. So now when you drag SAMP_CUSTOMER_D to the logical layer, its name is changed to samp customer d.

By default the **Automatically rename objects when added to logical layer** user preference is set to off. So when you drag and drop tables and columns from the physical layer to the logical layer, the resulting logical table and column names match the corresponding physical table and column names.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Page Menu** and then click **Preferences**.
4. In the Users Preferences dialog box, scroll to **Logical Layer** and then click the **Automatically rename objects when added to logical layer** checkbox to enable the user preference.
5. Select which physical objects you want to automatically rename when you drag them to the logical layer.
6. In Rename Actions, click **Add rules** and select a rule.
7. Optional: Click **Add rules** to add another rule.
8. Click **Apply**.

Create a Business Model in the Logical Layer

A business model contains logical tables, logical columns, logical table sources, and logical hierarchies. The business model also contains mappings from logical to physical tables.

A business model can map to more than one data source. Data mapping can also come from different granularity within data sources.

The logical layer can contain one or more business models.

To add tables to the business model, you can drag tables from the physical layer to the business model. Or you can create new fact, dimension, and lookup tables within the business model.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. In the Logical Layer pane click **Create** and then click **Create Business Model**.
5. In Create Business Model, go to the **Name** field and type a name. Click **OK**.

   The new business model's tab is displayed and opens to **Logical Tables**.

About Logical Tables

Logical tables provide a dimensional view of your business' data.

A logical table is sourced from one or more physical tables through mappings. There are three types of logical tables: fact, dimension, and lookup. The logical schema defined in each business model must contain at least two logical tables, and you must define relationships between the logical tables.

Each logical table is associated with one or more logical columns and one or more logical table sources. You can add a new logical table source, edit or delete an existing table source, create or change mappings to the table source, or define when to use logical table sources.

In most cases when you need to create a logical table, you drag and drop tables from the physical layer to the logical layer. In some situations you need to create a logical table manually. See Create a Fact, Dimension, or Lookup Logical Table.

When you drag and drop physical tables from the physical layer to the logical layer, the columns in the table are also added to the logical table along with join relationships. Primary keys and joins are created that mirror the keys and joins in the physical layer. After adding objects to the logical layer, you can modify the objects in the logical table without affecting the objects in the physical layer.

If you create tables manually or drag tables from the physical layer to the logical layer, then you must create the logical mappings between the new or newly dragged tables and the previously dragged tables.

After you add a logical table, you can perform tasks such as change a table's name, reorder the logical table sources, and configure logical joins.

Create a Fact, Dimension, or Lookup Logical Table

Manually create a logical table when you can't drag and drop a table from the physical layer to the logical layer. For example, if a table you need doesn't exist in your physical schema, then you create a logical table manually.

When you create a table, you choose the table type:

Fact table - Contains measures. For example, revenue and cost.

Dimension table - Contain data to be analyzed by measures. For example, products and customers.

Lookup table - Contains multilingual data corresponding to rows in the base tables. For more information see What Is Multilingual Data Support?

After creating a logical table, you must add columns and logical table sources to it. And you must join it to other logical tables within the business model.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. In the Logical Layer pane click **Create** and then click **Create Logical Table**.
5. In Create Logical Table, go to the **Name** field and enter a table name. Then go to **Type** and select the new table's type.
6. Go to **Business Model** and select the business model to add the table to.
7. Click **OK**.

Work with Logical Columns

This topic provides information about how to create and modify logical columns.

Topics:

About Logical Columns

Each logical table contains one or more logical columns. A logical column can be an attribute or a measure that is mapped or calculated.

In most cases you create logical columns by dragging tables from the physical layer to the logical layer. The logical columns you create in this way map to one or more physical columns and they inherit the physical column's data types.

You can also manually create logical columns that are derived from calculations based on other logical columns.

For example, you have a dimension table with two mapped attribute columns: a First Name column and a Last Name column. In the dimension table you can also have a calculated column named Full Name that is calculated by concatenating the Last Name column with the First Name column.

In the same example, you have a fact table with two mapped measure columns: a Revenue measure column with an aggregation of Sum, and a Billed Quantity measure column with an aggregation of Count. In the fact table you can also have a calculated measure column named Actual Unit Price that is calculated by dividing Revenue by Billed Quantity.

Add or Modify a Logical Column

You can add a new column to a logical table, or update an imported column's properties.

In most cases you create logical columns by dragging a physical table from the physical layer to the logical layer. This action creates a logical table with logical columns based on the physical table and its physical columns.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. In the Logical Layer pane, browse for and double-click the table where you want to add or modify a column.
5. In the logical table, click the **Columns** tab.
6. Click **Add Column** and specify how you want to add the column:
   - Click **Create New Column** to create an empty column.
   - Click **Add Physical Column** and in Select Physical Column browse for and select one or more columns. Click **Select**.
7. Optional: To duplicate a column, in the column table, hover over the column you want to duplicate and click its **Row Menu** and click **Duplicate**.
8. In the column table, click a column to highlight it, and then click **Detail view** to view or modify its properties.
9. Specify the column's properties.
10. Click **Save**.

Delete a Logical Column's Logical Table Source

Adding a logical table source to a logical table automatically adds logical columns and maps them to physical columns. You can delete a column's logical table source.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.

4. In the Logical Layer pane, browse for and double-click the table with the column you want to modify.
5. In the logical table, click the **Columns** tab.
6. In the columns list, click a column to select it and then click **Detail view** to open the properties pane.
7. Scroll to the Sources section, click the logical table source that you want to remove, and click **Delete**.
8. Click **Save**.

Base a Logical Column's Sort Order on a Different Column

Change the sort order of a logical column when you don't want to order a column's values alphabetically (lexical order).

In a lexical order sort, columns are ordered by their alphabetic spelling and not divided into a separate group. For example, if you sorted on month (using a column such as `MONTH_NAME`), the results return February, January, March in their lexicographical sort order.

Suppose you want to sort months in chronological order, so January, February, and March. Your table needs to have a month key such as `MONTH_KEY` with values of 1 (January), 2 (February), 3 (March) to achieve the chronological sort order. You set the Sort order column field for the `MONTH_NAME` column to the `MONTH_KEY` and then a request to order by `MONTH_NAME` would return January, February, and March.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. In the Logical Layer pane, browse for and double-click the table with the logical column you want to change the sort order for.
5. In the logical table, click the **Columns** tab.
6. In the column table, click the column to highlight it, and then click **Detail view** to view its properties.
7. In the logical column's **General** properties, click **Sort By** and select the column that you want to sort by.
8. Click **Save**.

Add Double Column Support

Double column support allows you to associate two columns. One column provides the display and description values such as the description of an item. The second column provides a descriptor ID or code column.

For example, you can use the actual column to provide the project list, and hide the ID column associated with the first column, as in Clinic and Clinic ID. Only the Clinic description is displayed to the user.

Using double columns can help improve performance because filtering is done on the ID column, which is numeric and indexed.

When multilingual columns are based on a lookup function, you can specify the non-translated lookup key column as the descriptor column of the translated column. You can use double column support to defining language-independent filters. For example, in analyses users see the display column, but the query filters on the hidden descriptor ID column.

See Support Multilingual Data.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. In the Logical Layer pane, browse for the table with the logical column you want to use as a double column.
5. In the logical table, click the **Columns** tab.
6. In the column table, click the column to highlight it, and then click **Detail view** to view its properties.
7. In the logical column's **General** properties, click the **Descriptor Column** field and select the column you want to use.
8. Click **Save**.

Create Derived Columns

Columns can be derived from other logical columns as a way to apply post-aggregation calculations to measures. You use the Expression Editor to specify the derived column expression.

You can use a derived column to create a lookup function to display data from multilingual database schemas. See Create Logical Lookup Columns.

The Oracle Analytics query engine prevents errors in divide-by-zero situations. The Oracle Analytics query engine creates a divide-by-zero prevention expression using `nullif()` or a similar function when it writes the physical SQL. Because of this, you don't have to use `CASE` statements to avoid divide-by-zero errors.

To optimize performance and avoid errors on the aggregation level, don't define aggregations in Expression Editor. Instead, set the logical column's **Aggregation Rule** field. See Set Aggregation Rules for a Measure Column.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. In the Logical Layer pane, browse for the table with the logical column you want to add the derived column expression to.
5. In the logical table, click the **Columns** tab.
6. In the column table, click the column to highlight it, and then click **Detail view** to view its properties.
7. In the logical column's **Sources** properties, pane, click **Logical Expression**, and then click **Open Expression Editor** and create and validate the expression.
8. In the Expression Editor, click **Save**.

Configure Logical Columns for Multicurrency Support

You can configure logical columns to allow users to select the currency that they want to display their visualizations, analyses, and dashboards currencies columns in.

You can set up this feature so that all users see the same static list of currency options, or you can provide a dynamic list of currency options that changes based on a Logical SQL statement you specify.

When you use session variables in an expression, you must use this format: `VALUEOF(NQ_SESSION.var_name)`. Edit any logical columns that display currency values to use the appropriate conversion factor using the `PREFERRED_CURRENCY` session variable.

See Create an Initialization Block and Create a Session Variable.

The following logical column expression uses the value of the `NQ_SESSION.PREFERRED_CURRENCY` variable to switch between different currency columns. The currency columns are expected to have the appropriate converted values.

```
INDEXCOL( CASE VALUEOF(NQ_SESSION.PREFERRED_CURRENCY) WHEN 'gc1' THEN 0
WHEN 'gc2' THEN 1 WHEN 'orgc' THEN 2 WHEN 'lc1' THEN 3 ELSE 4 END,
"Paint"."Sales Facts"."USDCurrency",
"Paint"."Sales Facts"."DEMCurrency" ,
"Paint"."Sales Facts"."EuroCurrency" ,
"Paint"."Sales Facts"."JapCurrency" ,
"Paint"."Sales Facts"."USDCurrency" )
```

An Administrator must configure user-preferred currency options to enable multicurrency support. For information about this configuration, see Define User-Preferred Currency Options.

1. Click **Variables**.
2. Click **Create**, click **Create Initialization Block**, and create the session variable's initialization block.
3. Create a session variable and name it *PREFERRED_CURRENCY*. Make sure to select the **Enable any user to set the value** field for the session variable.
4. Click **Save**.
5. Click **Logical Layer**.
6. In the Logical Layer pane, browse for the table with the logical column you want to configure for multicurrency.
7. In the logical table, click the **Columns** tab.
8. In the column table, click the column to highlight it, and then click **Detail view** to view its properties.
9. In the logical column's **Sources** properties, click the **Logical Expression** field and the click **Open Expression Editor**.
10. In the Expression Editor, create and validate a derived expression that uses the *PREFERRED_CURRENCY* variable.
11. Click **Save** to save the expression.

12. Optional: To provide a dynamic list of currency options, create a table in your data source that provides the entries you want to display for the user-preferred currency. This table must include the following columns:
    - The first column contains the values used to set the session variable `PREFERRED_CURRENCY`. Each value in this column is a string that uniquely identifies the currency (for example, gc2).
    - The second column contains currency tags from the Currencies XML system setting. The `displayMessage` values for each tag are used to populate the Currency box and currency prompts, for example, `int:euro-1`.
    - You can provide a third column that contains the values used to set the presentation variable `currency.userPreference`. Each value in this column is a string that identifies the currency, for example, Global Currency 2. If you omit this column, then the values for the `displayMessage` attributes for the corresponding currency tags located in the Currencies XML system setting are used.

    Sample user-preferred currency entries:
    - UserPreference: orgc1, CurrencyTag: loc:en-BZ, UserPreferenceName: Org currency
    - UserPreference: gc2, CurrencyTag: int:euro-1, UserPreferenceName: Global currency 2
    - UserPreference: lc1, CurrencyTag: int:DEM, UserPreferenceName: Ledger currency
    - UserPreference: gc1, CurrencyTag: int:USD, UserPreferenceName: Global Currency 1

Specify a Logical Table's Primary Key

After creating logical tables and adding columns to them, you specify a primary key for each dimension table.

Logical dimension tables must have a logical primary key. A logical primary key can be a composite key, which is composed of one or more logical columns. Oracle advises against specifying logical keys for logical fact tables.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer** and locate and double-click the logical table that you want to add a primary key to.
4. In the logical table, click the **General** tab.
5. Click the **Primary Key** field and select the column that you want to use as the table's primary key.
6. Optional: If the primary key is a composite key, then click the **Primary Key** field and select another column.
7. Click **Save**.

Work with Logical Joins

This topic provides information about how to create and modify logical joins.

Topics:

About Logical Joins

Logical joins define relationships between logical tables.

Logical joins are conceptual joins and not physical joins. Logical joins don't join to specific keys or columns. A single logical join can correspond to many possible physical joins.

A key property of a logical join is cardinality. Cardinality expresses how rows in one table are related to rows in the table that it's joined to. A one-to-many cardinality means that for every row in the first logical dimension table there are 0, 1, or many rows in the second logical table. Semantic Modeler considers a table to be a logical fact table if it's at the Many end of all logical joins that connect it to other logical tables.

Specifying the logical table joins is required so that the Oracle Analytics query engine can have the necessary metadata to translate a logical request against the business model to SQL queries against the data sources. The logical join information provides the Oracle Analytics query engine with the many-to-one relationships between the logical tables. This logical join information is used when the Oracle Analytics query engine generates queries against the underlying data source.

You don't need to create logical joins in the logical layer if both of the following statements are true:

- You create the logical tables by dragging and dropping all required physical tables to the logical layer.
- The logical joins are the same as the joins in the physical layer.

You might need to create some logical joins in the logical layer because you can't drag and drop all physical tables simultaneously except in very simple models.

You use the logical diagram to create joins. When you create a join expression in the physical layer, you can specify expressions and the specific columns on which to create the join. When you create a logical join in the logical layer, you can't specify expressions or columns to create joins on. A join in the physical layer doesn't require a matching join in the logical layer.

What Are Driving Tables?

Driving tables optimize how the Oracle Analytics query engine processes cross-database joins when one table is very small and the other table is very large.

Specifying driving tables leads to query optimization in cases where the number of rows being selected from the driving table is much smaller than the number of rows in the table that it's joined to.

When you specify a driving table, the Oracle Analytics query engine uses the driving table if the query plan determines that the table's use can optimize query processing. The small table (the driving table) is scanned, and parameterized queries are issued to the large table to select matching rows. The other tables, including other driving tables, are then joined together.

You can use driving tables with inner joins, and for outer joins when the driving table is the left table for a left outer join, or the right table for a right outer join. Driving tables aren't used for full outer joins.

Note the following information when deciding to set a driving table:

- Specify a driving table when the driving table is extremely small (less than 1000 rows).
- Specify a driving table only when multi-database joins are going to occur.
- If large numbers of rows are being selected from the driving table, specifying a driving table could lead to significant performance degradation or, if the `MAX_QUERIES_PER_DRIVE_JOIN` limit is exceeded the query terminates.

There are two entries in the database features table that control and tune driving table performance:

- `MAX_PARAMETERS_PER_DRIVE_JOIN`

  This is a performance tuning parameter. The larger its value, the fewer parameterized queries are generated. Values that are too large can result in parameterized queries that fail due to back-end database limitations. Setting the value to 0 (zero) turns off drive table joins.

- `MAX_QUERIES_PER_DRIVE_JOIN`

  This is used to prevent runaway drive table joins. If the number of parameterized queries exceeds its value, then the query is terminated and an error message is returned to the user.

What Determines Join Trimming?

This topic describes how the Oracle Analytics query engine determines which joins it can trim from a physical query.

These are the join trimming rules for tables within a logical table source:

- Join Outerness (Inner, Left Outer, Right Outer, or Full Outer).
- Join Cardinality (There are nine join cardinality combinations excluding those with Unknown cardinality on at least one side of the join.).
- Whether the logical table source contains a WHERE clause filter.
- Whether the physical join is a join or join expression.

The Oracle Analytics query engine uses the following criteria to trim a join:

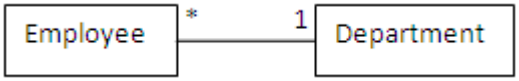
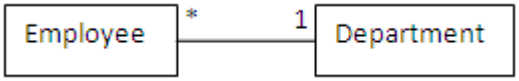
- No references to the trimmed table can exists anywhere in the query such as in the projected list of columns or in the WHERE clause.

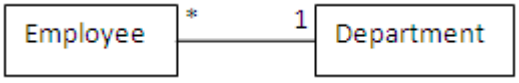
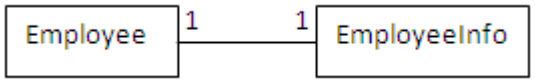
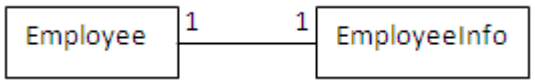
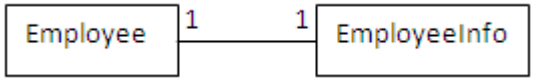
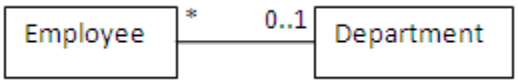
- The trimmed table must not cause the cardinality of the result set to change. If removing a join could potentially change the number of rows selected, then the Oracle Analytics query engine doesn't trim it.

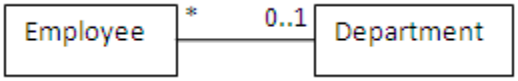
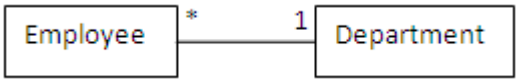
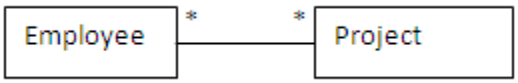
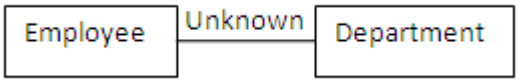
  A join is considered to have the potential to change the number of rows in the result set if any of the following conditions are true. If any of these conditions are true, then the join isn't trimmed from the query:

  - The join is a full outer join, only inner joins, left outer joins, and right outer joins are candidates for trimming
  - The join cardinality is unknown on either side
  - The table to trim is on the many side of a join, in other words, the detail table is never trimmed in a primary-detail relationship
  - The table to trim has a 0..1 cardinality and the join is an inner join. 0..1 cardinality implies that a possible matching row in the table. A join with 0..1 cardinality on one side is effectively like a filter. The Oracle Analytics query engine can't trim the table without changing the number of rows selected.
  - The table to trim is on the left side of a left outer join or on the right side of a right outer join, the row-preserving table is never trimmed. There is an exception to this rule for queries that select only attributes in which a DISTINCT clause is added to the query. Because of the DISTINCT clause, trimming the row-preserving table doesn't affect the number of rows returned from the null-supplying table. In the special case of distinct queries on attributes, you can trim the row-preserving table from an outer join.

The following table provides examples of when the Oracle Analytics query engine trims joins from the query.

Scenario	Result
 Employee INNER JOIN Department	The Oracle Analytics query engine can trim Department because it's on the one side of an inner join. The Oracle Analytics query engine can't trim Employee because it's on the many side of an inner join.
 Employee LEFT OUTER JOIN Department	The Oracle Analytics query engine can trim Department because it's on the one side of the join and it's on the right side of a LEFT OUTER JOIN, the null supplying table. The Oracle Analytics query engine can't trim Employee because it's on the many side, and because it's on the left side of a LEFT OUTER JOIN, the row preserving table.

Scenario	Result
 Employee RIGHT OUTER JOIN Department	The Oracle Analytics query engine can't trim Department because it's on the right side of a RIGHT OUTER JOIN, the row preserving table. The Oracle Analytics query engine can't trim Employee because it's on the many side of the join.
 Employee INNER JOIN EmployeeInfo	The Oracle Analytics query engine can trim either side because both tables are on the one side of an inner join.
 Employee LEFT OUTER JOIN EmployeeInfo	The Oracle Analytics query engine can trim EmployeeInfo since it's on the one side of the join, and it's on the right side of a LEFT OUTER JOIN, the null supplying table. The Oracle Analytics query engine can't trim Employee because it's on the left side of a LEFT OUTER JOIN, the row preserving table.
 Employee RIGHT OUTER JOIN EmployeeInfo	The Oracle Analytics query engine can trim EmployeeInfo because it's on the right side of a RIGHT OUTER JOIN, the row preserving table. You can trim Employee because it's on the "one" side of the join, and it's on the left side of a RIGHT OUTER JOIN, the null supplying table.
 Employee INNER JOIN Department	The Oracle Analytics query engine can trim Department because it's on the 0..1 side of an inner join. The Oracle Analytics query engine can trim Employee because it's on the many side of an inner join.

Scenario	Result
 Employee LEFT OUTER JOIN Department	The Oracle Analytics query engine can trim Department because it's on the 0..1 side of an outer join, and it's on the right side of a LEFT OUTER JOIN, the null supplying table. The Oracle Analytics query engine allows trimming the null supplying table on the 0..1 side of an outer join, because in this case, trimming Department from the query wouldn't change the number of rows selected from the Employee table. The Oracle Analytics query engine can trim Employee since it's on the many side of an outer join.
 Employee FULL OUTER JOIN Department	The Oracle Analytics query engine can't trim either side because the join is a FULL OUTER JOIN.
 Employee MANY TO MANY Project	The Oracle Analytics query engine can't trim either side because the join is many-to-many.
 Employee UNKNOWN Department	The Oracle Analytics query engine can't trim either side because the join has unknown cardinality.

Add and Define Logical Joins

You use the Logical Diagram to add and define joins between logical tables.

When you drag multiple tables from the physical layer to the logical layer, corresponding logical tables and logical columns are created, and a logical join is

automatically created for each physical join. You can modify these joins as needed, or add new joins.

See About Logical Joins.

> **Note:**
>
> Use caution when specifying a driving table. Driving tables are used for query optimization only under rare circumstances and when the driving table is small (fewer than 1000 rows). Choosing a driving table incorrectly can lead to severe performance degradation. See What Are Driving Tables?

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click a table, right click, and select **Show Logical Diagram** and select **Selected Tables Only**.
4. Drag and drop additional tables to the Logical Diagram.
5. Working in the Logical Diagram, hover over the first table in the join (the table representing many in the one-to-many join.), grab its handle, and drag to the table that you want to join to (the table representing one in the one-to-many join).

   A box is displayed around the table that you are joining to.
6. In Add Join, modify the values in the **Cardinality**, **Driving Table**, and **Join Type** fields as needed.
7. Click **Add**.

Identify the Physical Tables That Map to Logical Tables

The Physical Diagram shows the physical tables that map to the selected logical table or tables. The diagram also shows the physical joins between each physical table.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. Click a table or Crtl click multiple tables, right click, and select **Show Physical Diagram** and select **Selected Tables and Direct Joins**.

Open the Logical Diagram

The logical diagram provides a graphic view of the logical table or tables that you selected. From the diagram you can view a logical table's columns, show joins, and add, modify, or delete logical joins.

The logical model diagram doesn't display other logical objects, such as business models, dimensions, or hierarchies.

You can select what you want the diagram to contain:

- **Selected Tables Only** - Displays only the selected logical tables. Logical joins display only if they exist between the tables that you selected.

- **Selected Tables and Direct Joins** - Displays the selected logical tables and any logical tables that join to the table or tables that you selected.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. Locate and right-click a table, or use Shift click or Ctrl click to select more than one table and right-click.
5. Hover over **Show Logical Diagram** and click **Selected Tables Only** or **Selected Tables and Direct Joins**.
6. In the physical diagram, double-click a table to view a list of column names with column type icons. Double-click the list to collapse it.
7. Right-click a table and select **Show Direct Joins** to display the tables that join to the table.
8. Double-click a join to access the Edit Join dialog box to view or modify the join's properties.
9. Click a table to select it and on the right side of the table, grab its handle and drag to another table to create a logical join and specify its properties.

Open the Physical Diagram from the Logical Layer

Opening the physical diagram from the logical layer helps you understand the model's logical-to-physical mappings, and shows you the physical objects that are associated with a particular logical object.

You can add or modify physical joins from the physical diagram. See Add and Define Physical Joins.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. Locate and right-click the table that you want logical-to-physical mappings information about, or use Shift click or Ctrl click to select more than one table and right-click.
5. Hover over **Show Physical Diagram** and click **Selected Tables Only** or **Selected Tables and Direct Joins**.
6. In the physical diagram, double-click a table to view a list of column names with column type icons. Double-click the list to collapse it.
7. Right-click a table and select **Show Direct Joins** to display the tables that join to the table.
8. Double-click a join to access the Edit Physical Join dialog box to view or modify the join's properties.
9. Click a table to select it and on the right side of the table, grab its handle and drag to another table to create a physical join and specify its properties.

Work with Logical Column Aggregation

This topic provides information about the different ways that you can set up logical column aggregation.

Topics:

About Levels of Aggregation

Only perform aggregations on measure columns. Measure columns should exist only in logical fact tables.

You can select different aggregation rules for different dimensions that are associated with a logical column. Suppose someone queries the aggregate column along with one dimension, you may want to use one type of aggregation rule, whereas with another dimension, you may want to use a different aggregation rule. For example, number of employees is a count on all dimensions except on the time dimension where the aggregation rule would be last.

When the default aggregation rule is **Count Distinct**, you can specify an override aggregation expression for specific logical table sources. For example, you may want to specify override aggregation expressions when you're querying different logical table sources that already contain some level of aggregation.

You can choose the **EVALUATE_AGGR** aggregation rule to enable queries to call custom functions in the data source. Use this aggregation rule when the aggregation must be done in an external data source.

By default, data is considered sparse. However, you might have a logical table source with dense data. A logical table source is considered to have dense data if it has a row for every combination of its associated dimension levels. When setting up aggregate rules for a measure column, you can specify that data is dense only if all the logical table sources to which it's mapped are dense.

For measures where the aggregation rule is the same in all dimensions, select one of the aggregate functions from the **Aggregation Rule** list. The function you select is always applied when a user or an application requests the column in a query, unless an override aggregation expression has been specified. When you select **Count Distinct** as the default aggregation rule, you can specify an override aggregation expression for specific logical table sources. Choose this option when you have more than one logical table source mapped to a logical column and you want to apply a different aggregation rule to each source.

Set Aggregation Rules for a Measure Column

You need to specify aggregation rules for mapped logical columns that are measures.

If your measure has different aggregation rules for different dimensions, for semi-additive measures, then you choose **Based on dimensions** as the measure's aggregation rule. See Set an Aggregation Level Based on a Dimension for a Measure Column.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. In the Logical Layer pane, browse for the table with the logical column that you want to add an aggregation rule to.
5. In the logical table, click the **Columns** tab.
6. In the column table, click the column to highlight it, and then click **Detail view** to view its properties.
7. In the logical column's **Aggregation** properties, click the **Aggregation Rule** field and select an aggregation rule.
8. Click **Add Aggregation by Level** and in the **Dimension** field select a table source.
9. Click the **Logical Level** field and select the level of aggregation. What you choose is the minimum level of aggregation, and the measure won't be aggregated below the level you choose.
10. Click **Save**.

Set an Aggregation Level Based on a Dimension for a Measure Column

The majority of measures have the same aggregation rule for each dimension. Some measures can have different aggregation rules for different dimensions.

For information about setting up dimension hierarchies, see About Level-Based Hierarchies and About Parent-Child Hierarchies.

For example, a bank could calculate account balances averages over a specific time, but calculated averages on individual accounts with a simple summation for a period. You can configure dimension-specific aggregation rules. You can specify one aggregation rule for a given dimension and specify other rules to apply to other dimensions.

Choose **Based on dimensions** as the measure column's aggregation rule if your measure has different aggregation rules for different dimensions, for semi-additive measures.

When setting up the aggregation, selecting the **Data is dense** option indicates that all sources that the column is mapped to have a row for every combination of dimension levels that they represent. Incorrect results are returned if you select this option and the measure column's table source doesn't contain dense data.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. In the Logical Layer pane, browse for the table with the logical column that you want to add an aggregation rule to.
5. In the logical table, click the **Columns** tab.
6. In the column table, click the column to highlight it, and then click **Detail view** to view its properties.

7. In the logical column's **Aggregation** properties, click the **Aggregation Rule** field and select **Based on dimensions**.
8. Click **Add Aggregate by Dimension** and in the dimension field select a dimension.
9. Click the **Formula** list and select an aggregation rule, or click the **Expression Builder** icon to use the Expression Editor to create the aggregation rule.
10. If all the logical table sources that the column is mapped to are dense, then select the **Data is dense** field.
11. Click **Save**.

Associate an Attribute with a Logical Level in Dimension Tables

You can associate attributes with a logical level.

You can associate measures with levels from multiple dimensions and aggregate to the levels specified. A measure is associated to a level is called a level-based measure. A level-based measure is computed at that grain, even when the query context has a lower grain. For example, if yearlySales is associated to year level, it's computed at the yearly level in the following query: `Select month, yearlySales`.

Dimensions are displayed in the Dimensions list. If the attribute is associated with a logical level, the level appears in the Levels list.

Another way to associate a measure with a level in a dimension is to expand the dimension tree in the logical layer, and then use drag-and-drop the column on the target level. See Level-Based Measure Calculations.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. In the Logical Layer pane, browse for the table with the logical column that you want to add an aggregation level to. Double-click the column.
5. In the logical table, click the **Columns** tab.
6. In the column table, click the column to highlight it, and then click **Detail view** to view its properties.
7. In the logical column's **Aggregation** properties, click the **Aggregation Rule** field and select an aggregation rule.
8. Click **Add Aggregation by Level** and in the **Dimension** field select a logical table source.
9. Click the **Logical Level** field and select a level.
10. Click **Save**.

Enable Write Back On Columns

You can configure individual columns so that analyses and dashboard users can update column data and write the changes back to the data source.

Enabling write back is a three step process where you disable caching on the corresponding physical table so that users can immediately see data updates, modify the logical column

Writeable field, and assign application roles and users the **Read/Write** permission to the corresponding presentation column.

You must perform additional tasks to enable write back in Oracle Analytics. See Enable Write Back in Analyses and Dashboards.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**.
4. In the Physical Layer pane, browse for and double-click the physical table with the column that you want to enable write back for.
5. In the physical table, click the **Columns** tab.
6. Click the **Caching** field and select **Do not cache**.
7. To locate a list of logical tables that use the physical table as their source, go to the Physical Layer pane, right-click the physical table, click **Show Related**, then **Logical**, and then **Logical Table**.
8. In the table list, locate and click the logical table that contains the logical column that you want to set for write back.
9. In the logical table, click the **Columns** tab.
10. In the columns list, click a column to select it and then click **Detail view** to open the properties pane.
11. In the logical column's **General** properties click the **Writeable** field.
12. In the Semantic Modeler's left pane, click Presentation Layer and locate and double-click the logical table that contains the logical column that you want to set up for write back.
13. In the columns list, click the presentation column that corresponds to the logical column that you're setting up write back for, and then click **Detail view** to open the properties pane.
14. Scroll to **Permissions** and deselect the **Same permission as table** field,
15. In the Permissions table, specify which roles have Read-Write permissions.
16. Click **Save**.

Work with Bridge Tables

This topic provides information about when to use bridge tables and how to model them.

Topics:

About Bridge Tables

Use a bridge table (or intermediate table) to resolve many-to-many relationships between tables.

For example, suppose there is an Employee table that contains information about employees, and a Jobs table that contains information about the jobs the employees perform. An organization's employees can have multiple jobs, and the same job can be performed by multiple employees. This results in a many-to-many relationship between the Employees table and the Jobs table.

For this scenario, you create a bridge table named Assignments to resolve the many-to-many relationship. Each row in the Assignments table is unique, representing one employee doing one job. If an employee has several jobs, there are several rows in the Assignments table for that employee. If a job is performed by several employees, there are several rows in the Assignments table for that job. The primary key of the Assignments table is a composite key, made up of a column containing the employee ID and a column containing the job ID.

By acting as a bridge table between the Job and Employee tables, the Assignments table enables you to resolve the many-to-many relationship between Employees and Jobs into:

- A one-to-many relationship between Employees and Assignments
- A one-to-many relationship between Assignments and Jobs

You should include Weight Factor as an additional column in the bridge table, and to calculating during ETL for efficient query processing.

Create Joins in the Physical Layer for Bridge and Associated Dimension Tables

To model bridge tables in the physical layer, create joins between the bridge table and the associated dimension tables.

After you've completed creating joins in the physical layer, you then add the needed associated dimension tables to the logical layer and model them in either a single dimension or separate dimensions.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**.
4. In the Physical Layer pane, browse for and Ctrl click the fact, bridge, and associated dimension tables.
5. Right-click, hover over **Show Physical Diagram**, and click **Selected Tables Only**.
6. From the bridge table, click and drag to draw a join line to a dimension table.
7. Add joins from the bridge table to the other associated dimension tables.
8. Confirm that one of the associated dimension tables is joined to the fact table.
9. Click **Save**.

Model the Associated Dimension Tables in a Single Dimension

In the logical layer, you can choose to model the two dimension tables associated with a bridge table in a single dimension, or in two separate dimensions.

Before you perform this task, you need to create the required physical joins. See Create Joins in the Physical Layer for Bridge and Associated Dimension Tables.

To model the associated dimension tables in a single dimension, create a second logical table source that maps to the bridge table and to the other dimension table, and then add columns from the other dimension table. Don't add the bridge table and the associated dimension table that isn't joined to the fact table to the logical layer. For the example described in About Bridge Tables, you add the Jobs table (dimension table joined to the fact table), but not the Assignment table (bridge table) and Employee table (dimension table not joined to the fact table).

Providing two separate logical table sources makes queries more efficient because it ensures that queries against a single dimension table don't involve the bridge table.

It's a good practice to use the bridge table name as the name of the source.

You can create dimensions based on your logical tables, including the logical table with the bridge table source.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. In the Logical Layer pane, click **Create** and then **Create Logical Table** to create the needed logical dimension table. Repeat this step to create the logical fact table.
5. In the logical layer, double-click the dimension table that is joined to the fact table and in the table, click the **Sources** tab.
6. Click **Add Physical Table** and then **Create Logical Table Source** and in the table source's **Name** field, enter the name of the bridge table that you created in the physical layer.
7. Click **Add Physical Table** and then **Create Logical Table Source** and in the table source's **Name** field, enter the name of the associated dimension table that isn't joined to the fact table.
8. Click the **Columns** tab to navigate to the table's column list.
9. Click **Add Column** and then click **Create New Column**.
10. In the new column's **Name** field, enter the name of a column from the dimension table that isn't joined to the fact table. Repeat this step to add the required columns.
11. Click **Save**.

Model the Associated Dimension Tables in Separate Dimensions

Instead of modeling the two dimension tables associated with a bridge table in a single dimension, you can choose to model them in separate dimensions.

Before you perform this task, you need to create the required physical joins. See Create Joins in the Physical Layer for Bridge and Associated Dimension Tables.

To model the associated dimension tables in separate dimensions, create a logical join between the fact table and the dimension table that isn't physically joined to the fact table, and then modify the dimension table's logical table source to add the other table mappings. Don't add the bridge table to the logical layer, but add all dimension tables associated with the bridge table. For the example described in About Bridge Tables, you add the Jobs table (dimension table joined to the fact table) and Employee table (dimension table not joined to the fact table), but you won't add the Assignment table (bridge table).

You can create dimensions based on your logical tables, including both logical tables associated with the bridge table.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. In the Logical Layer pane, click **Create** and then **Create Logical Table** to create a logical dimension table. Repeat this step to create the required logical dimension tables and to create the logical fact table.
5. In the Logical Layer pane, Crtl click the fact table and associated dimension tables.
6. Right-click, hover over **Show Logical Diagram**, and click **Selected Tables Only**.
7. From the fact table, click and drag to draw a join line to the dimension table not joined to the fact table.
8. In the Logical layer pane, double-click the dimension table you joined to the fact table and in the table, click the **Sources** tab.
9. Click **Add Physical Table** and then **Create Logical Table Source** and in the new table source's **Name** field, enter the name of the bridge table that you created in the physical layer.
10. Click **Add Physical Table** and then **Create Logical Table Source** and in the new table source's **Name** field, enter the name of the associated dimension table. Repeat this step to add other dimension table.
11. Click **Save**.

10

Build a Semantic Model's Presentation Layer

This chapter contains information to help you understand how to build a semantic model's presentation layer.

Topics:

What is the Presentation Layer?

The presentation layer provides users with customized, secure, role-based views of a business model.

Role-based views provide object security and also provide a way to hide some of the complexity of the business model.

In the presentation layer, you can set an implicit fact column. The primary function of the presentation layer is to provide custom names, dictionary entries, organization, and security for different groups of users.

Presentation layer views are called subject areas. Subject areas contain presentation tables, columns, hierarchies, and levels. You can create a subject area that's identical to your business model, or you can create role-based subject areas that show a single subject or that supports a specific business role. Subject areas aren't abstract views. You should create subject areas that organize your content in a way that benefits your users.

You can use a JDBC connection to query subject areas externally. When you access subject areas in this way, the subject areas are displayed as catalogs.

Even though the Logical SQL requests from visualizations, analyses, and other clients query the presentation tables and columns, the logic for entities, relationships, and joins is in the logical layer.

There is no automatic way to synchronize all changes between the logical layer and the presentation layer. For example, if you add logical columns to an existing logical table, or edit existing columns, you must manually update the corresponding presentation layer table and columns.

In some cases, if there are many changes to a logical table or even to an entire business model, it's easiest to delete the corresponding presentation table or subject area, and then and drag and drop the updated logical objects to the presentation layer. For this reason, it's

best to wait until the logical layer is relatively stable before adding customizations in the presentation layer.

About Alternative Names for Presentation Objects

Use alternative names to help track an object's name changes and to prevent SQL queries that include the object's previous name from failing.

Semantic Modeler *doesn't* create an alias when you change a presentation object's name. If you need to track an object's previous names, then Oracle recommends that you create and manage alternative names for the object. When you rename a presentation object, you can create alternative names for the object to prevent breaking references that any existing requests have to the old names, including requests from workbooks, analyses, dashboards, reports, or other Logical SQL clients.

For example, suppose you have a subject area called *Sample Sales Reduced* that contains a presentation table called *Facts Other*. If you rename the table's presentation column *# of Customers* to *Number of Customers*, any requests that use *# of Customers* fail. However, if you add a *# of Customers* alternative name to the *Number of Customers* column object, then queries containing both *# of Customers* and *Number of Customers* succeed and return the same results.

Because presentation layer objects are often deleted and then re-created during the semantic model development process, it's best to wait until your logical business model is relatively stable before renaming and creating alternative names for presentation objects.

Note the following information when working with alternative names:

- Alternative names for presentation objects aren't displayed in the subject areas that users access to create visualizations and analyses. Also, alternative names aren't displayed in other query clients used to create queries. End users and users who write queries only use the assigned names of subject areas, hierarchies, levels, tables, and columns.
- Alternative names work differently than SQL aliases or the alias feature in the physical layer. Alternative names provides synonyms for object names, much like synonyms in SQL.
- You can't rename a presentation object to a name that's already in use as an alias for an object of the same type.
- You can use alternative names in Logical SQL queries.

Work with Subject Areas

This topic provides information about how to create and modify the presentation layer's subject areas.

Topics:

About Creating Subject Areas

There are several ways to create subject areas in the presentation layer.

Oracle recommends that you drag and drop a business model from the logical layer to the presentation layer, and then modify the presentation layer based on what you want users to see. You can move columns between presentation tables, remove columns that don't need to be seen by the users, or even present all of the data in a single presentation table. You can create presentation tables to organize and categorize measures in a way that makes sense to your users.

You can also duplicate an existing subject area and its corresponding business model. Or you can create an empty subject area.

Although each subject area must be populated with contents from a single business model, you can create multiple subject areas for one business model. Creating multiple subject areas for one business model makes it easier for the users to work with the content and create queries that span multiple subject areas.

There are many ways to create multiple subject areas from a single business model. One method is to drag a business model to the presentation layer multiple times, then edit the properties or objects of the resulting subject areas.

For example, suppose you have a business model called `ABC` that contains the `Geography` and `Products` dimensions. When you drag it to the presentation layer twice two subject areas are created with the default names `ABC` and `ABC#1`. You then edit the subject areas as follows:

- Rename the `ABC` subject area to `DEF`, then delete the `Geography` presentation hierarchy
- Rename the `ABC#1` subject area to `XYZ`, then delete the `Products` presentation hierarchy

Users can then run queries that span the `DEF` subject area containing the `Products` hierarchy, and the `XYZ` subject area containing the `Geography` hierarchy.

When you query a single subject area, all the columns exposed in that subject area are compatible with all the dimensions exposed in the same subject area. However, when you combine columns and dimensions from multiple subject areas, you must ensure that you don't include combinations of columns and dimensions that are incompatible with one another.

For example, a column in one subject area might not be dimensioned by Project. If columns from the Project dimension from another subject area are added to the request along with columns that aren't dimensioned by Project, then the query might fail to return results, or cause the error, "`No fact table exists at the requested level of detail: XXXX.`"

About the Implicit Fact Column

For each subject area in the presentation layer, you can set an implicit fact column.

The subject area's specified implicit fact column is added to a query when it contains columns from two or more dimension tables and no measures. The column isn't visible in the results. It's used to specify a default join path between dimension tables when there are several possible alternatives or contexts.

If an implicit fact isn't configured, then the Oracle Analytics query engine uses any fact table source to answer dimension-only subrequest that contains multiple dimensional references but no fact reference.

The Oracle Analytics query engine can also use any fact table source if the configured implicit fact column isn't relevant to the dimensions that are joined. This could happen, for example, when implicit fact column is a level based measure at a level higher than the dimensional only subrequest.

You use the Implicit Fact Column field in the subject area's General tab to add, remove, or replace the implicit fact column. See Create a Subject Area.

Create a Subject Area

Manually create a subject area when you can't drag and drop a business model from the logical layer to the presentation layer.

You use logical content from a single business model to build the subject area. Subject areas can't span business models.

The subject area's specified implicit fact column is added to a query when it contains columns from two or more dimension tables and no measures. The column isn't visible in the results. It's used to specify a default join path between dimension tables when there are several possible alternatives or contexts.

You can select the **Hide if** field and write an expression to hide a subject area. See Write an Expression to Hide a Presentation Object.

Semantic Modeler *doesn't* create an alias when you change a subject area's name. If you need to track a subject area's previous names, then Oracle recommends that you use the **Alternative Names** field to create and manage alternative names for the subject. For more information about alternative names, see About Alternative Names for Presentation Objects.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Presentation Layer**.
4. Click **Create** and then click **Create Subject Area**.
5. In Create Subject Area, go to the **Name** field and enter a subject area name.
6. Go to **Business Model** and select the business model to associate with the subject area.
7. Click **OK**.
8. In the subject area, click **General** to set the subject area's general properties.
9. Optional: In the **Description** field, enter a description that is displayed when a user creating visualizations or analyses hovers over the subject area in the data sources list.
10. Optional: In the **Implicit Fact Column** field and click **Select** to browse for and select the fact column you want to use.
11. In the **Alternative Names** field, enter an alternative name for the subject area.

12. Optional: Select the **Hide if** field and provide an expression that controls whether the subject area is available to users when they create visualizations and analyses.
13. Click **Save**.

Work with Presentation Tables and Columns

This topic provides information about how to create and modify the presentation layer's tables and columns.

Topics:

About Presentation Tables

Subject areas contain presentation tables, and presentation tables contain presentation columns and presentation hierarchies. Presentation tables function as intuitive categories that collocate the columns and data users need to create visualizations and analyses.

In most cases, you create presentation tables by dragging and dropping logical tables from the logical layer into a presentation layer's subject area. The names and object properties of the presentation tables are independent of the logical table properties.

In most cases, a presentation table contains columns from its corresponding logical table. But you can build a presentation table containing columns from a different logical table within the same business model. In such cases, be aware that building a presentation table in this way can cause query errors when a user selects these columns.

Create a Presentation Table

Manually create a presentation table when you can't drag and drop a table from the logical layer to the presentation layer.

You can select the **Hide if** field and write an expression to hide a presentation table. See Write an Expression to Hide a Presentation Object.

Semantic Modeler *doesn't* create an alias when you change a presentation table's name. If you need to track a table's previous names, then Oracle recommends that you use the **Alternative Names** field to create and manage alternative names for the table. For more information about alternative names, see About Alternative Names for Presentation Objects.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Presentation Layer**.
4. In the Presentation Layer pane click **Create** and then click **Create Presentation Table**.

5. In Create Presentation Table, go to the **Name** field and enter a presentation table name.
6. Go to **Subject Area** and select which subject area to add the table to.
7. Click **OK**.
8. In the presentation table, click **General** to set the presentation table's general properties.
9. Optional: In the **Description** field, enter a description that is displayed when a user creating visualizations or analyses hovers over the table in the subject area.
10. Optional: In the **Alternative Names** field, enter an alternative name for the presentation table.
11. Optional: Select the **Hide if** field and provide an expression that controls whether the presentation table is available to users when they create visualizations and analyses.
12. Click **Save**.

About Presentation Columns

Presentation columns provide the data users need to create visualizations and analyses.

Presentation columns can be either attribute columns or measure columns. Attribute columns contains columns from dimension logical tables with a GROUP BY and DISTINCT clause applies, and measure columns which are numeric columns are from fact logical tables with an aggregation function like SUM applied.

In most cases, you create presentation columns by dragging and dropping logical columns from the logical layer to a presentation table. Columns that you drag and drop must have unique names. But in some cases when you can't drag and drop logical columns to a presentation table, you can manually create presentation columns within a presentation table.

You can drag and drop a column from a logical table into multiple presentation tables. For example, you can create several presentation tables that contain different classes of measures such as one containing volume measures, one containing share measures, and one containing measures from a year ago.

Create a Presentation Column

Manually create a presentation column when you can't drag and drop a column from the logical layer to the presentation layer.

You can select the **Hide if** field and write an expression to hide a presentation column. See Write an Expression to Hide a Presentation Object.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Presentation Layer**.
4. In the Presentation Layer pane locate and double-click the presentation table that you want to add a column to.
5. In the presentation table, click **Columns** and then click **Add Column**.

6. In the New Column table row, enter a name for the column and press Enter.
7. Confirm that the new column is highlighted in the table, and then click **Detail view**.
8. Optional: In the **Description** field, enter a description that is displayed when a user creating visualizations or analyses hovers over the column in the subject area.
9. Go to **Logical Column**, click **Select**, and select which logical column to associate the presentation column with.
10. Optional: Select the **Hide if** field and provide an expression that controls whether this column is available to users when they create visualizations and analyses.
11. Click **Save**.

Modify a Presentation Column Name

When you drag and drop a logical table or logical columns to the presentation layer, by default the resulting presentation columns have the same names as the logical columns they're based on. You can change a presentation column's default name to a more user-friendly name.

Semantic Modeler *doesn't* create an alias when you change a presentation column's name. If you need to track a column's previous names, then Oracle recommends that you use the **Alternative Names** field to create and manage alternative names for the column. For more information about alternative names, see About Alternative Names for Presentation Objects.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Presentation Layer**.
4. In the Presentation Layer pane locate and expand the presentation table containing the column to rename. Double-click the column.
5. In **Name**, enter a new name for the column.
6. Optional: In **Alternative Names**, type an alternative name and press Enter. Repeat this step to add more alternative names for the column.
7. Click **Save**.

Delete a Presentation Column

Remove presentation columns that don't provide meaningful content to the users who create visualization and analyses.

Consider removing these types of presentation columns from presentation tables:

- Key columns that have no business meaning.
- Columns that users don't need to view. For example, a product code column when its text descriptions exist in another column.
- Columns that users aren't authorized to read.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Presentation Layer** and locate and double-click the presentation table that you want to remove columns from.

4. In the presentation table, click **Columns**.
5. Click the column that you want to remove, click **Row menu**, and then click **Delete**.
6. Click **Save**.

Reorder and Nest Tables for End Users

You can specify the order and levels that a subject area's tables display to users who access the subject area to create visualization and analyses.

You can move a presentation table up or down to change its position in the subject area, and you can change the table's relationship to the other tables in the subject area by moving the table right or left. Moving tables right and left provides the end user with nested folders in the subject area.

Changing the table order doesn't change the order that the presentation tables display in the Semantic Modeler's Presentation Layer pane. The table order and nesting you specify only appears in the subject area's **Tables** tab and the subject area that end users access. The tables aren't actually nested in drill-down, and the qualified names of the objects remain the same.

When you run consistency check on the subject area, the consistency check detects any circularity in parent-child presentation table assignments. It also detects and reports project definitions that include child presentation tables without parent presentation tables.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Presentation Layer**.
4. In the Presentation Layer pane locate and double-click the subject area where you want to reorder the tables for end users.
5. In **Tables** click a table to select it and use the **Move Up**, **Move Down**, **Move Right**, and **Move Left** buttons to change its position.
6. Click **Save**.

Work with Presentation Hierarchies and Levels

This topic provides information about how to create and modify the presentation layer's hierarchies and levels.

Topics:

About Presentation Hierarchies and Levels

Use presentation hierarchies and presentation levels to provide multidimensional models. Presentation hierarchies and levels display to users as roll-up information in subject areas they use to create visualizations and analyses.

In most cases, you create presentation hierarchies by dragging and dropping logical dimensions from a logical table into a presentation table. If you drag and drop a logical hierarchy containing more than one hierarchy to a presentation table, then Semantic Modeler creates a separate presentation hierarchy for each of the logical hierarchy's hierarchies. See About Adding Logical Hierarchies with Multiple Hierarchies to the Presentation Layer.

You can also manually browse for and add hierarchies to a presentation table. After you've added a logical hierarchy to a presentation level, you can apply fine-grained access control to the presentation hierarchy and its levels.

A presentation hierarchy's members aren't visible in the presentation layer. But when creating visualizations and analyses, users can view hierarchy members in subject areas.

Users can create hierarchy-based queries using objects in presentation hierarchies and levels. Presentation hierarchies expose analytic functionality such as member selection, custom member groups, and asymmetric queries.

About Creating Presentation Hierarchies

Oracle recommends that to create a presentation hierarchy, you add a logical dimension hierarchy from the logical layer to a presentation table.

In the logical layer, logical dimensions are peer objects to tables. In the presentation layer, a presentation hierarchy is always located in a presentation table. Presentation hierarchies are displayed within their associated tables in the subject areas users access to create visualizations and analyses. This structure provides a conceptually simpler model.

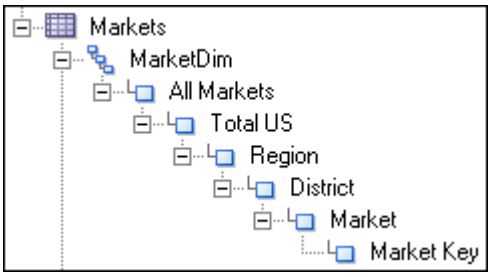
If a logical dimension spans multiple logical tables in the logical layer, then it's a best practice to model the separate logical tables as a single presentation table in the presentation layer.

There are different ways to create presentation hierarchies:

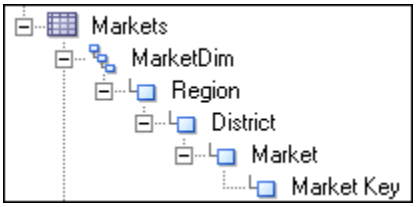
- Drag an entire business model from the logical layer to the presentation layer. Semantic Modeler automatically creates the presentation hierarchies and constituent levels are created automatically when you drag an entire model.
- Drag a logical dimension table from the logical layer to the presentation layer. Semantic Modeler automatically creates presentation hierarchies and levels based on the dimensions.
- Open a presentation table and in the **Hierarchies** tab, click **Add Hierarchy** to browse for and select the hierarchy to add to the table.

You can also drag an individual logical level from the logical layer to a presentation table to create a presentation hierarchy that's a subset of the logical dimension hierarchy.

For example, suppose a logical dimension has the levels All Markets, Total US, Region, District, Market, and Market Key. Dragging and dropping the entire logical dimension to the corresponding presentation table is displayed like this:



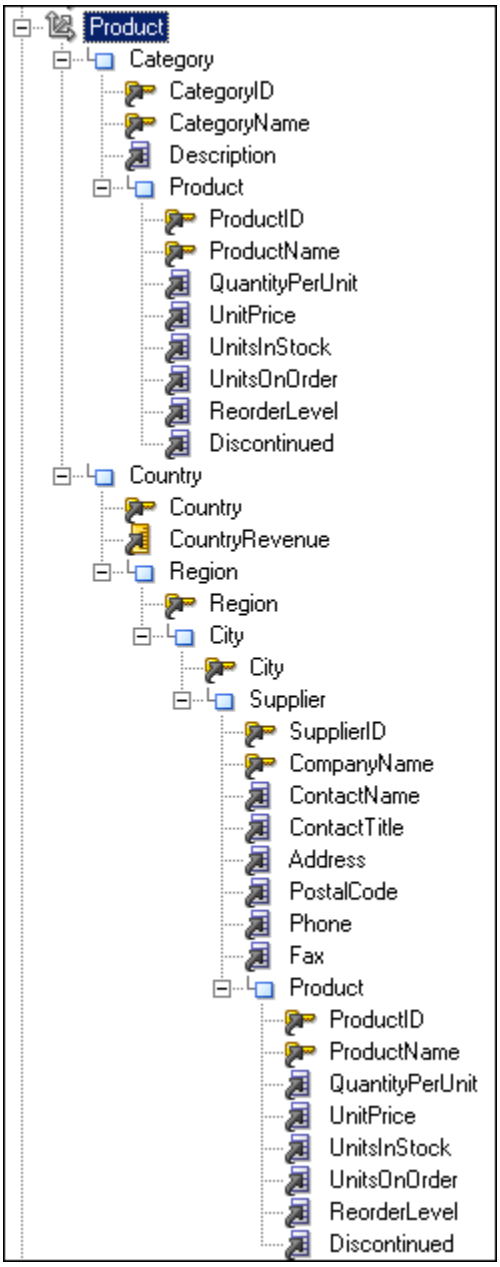
However, dragging and dropping the Region level to the same presentation table is displayed like this:



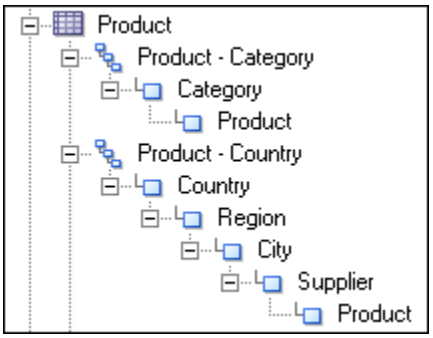
About Adding Logical Hierarchies with Multiple Hierarchies to the Presentation Layer

If you drag and drop a logical hierarchy containing more than one hierarchy to a presentation table, then Semantic Modeler creates a separate presentation hierarchy for each of the logical hierarchy's hierarchies.

For example, suppose your model contains a logical hierarchy called Product and it contains the two hierarchies named Category and Country:



In the logical layer, Semantic Modeler models this logical hierarchy as a single dimension object that contains multiple hierarchies. In the presentation layer, Semantic Modeler models this dimension as two separate objects: one that displays the drill path through the Category level, and another that shows the drill path through the Country level:



Add a Presentation Hierarchy to a Presentation Table

Manually add a presentation hierarchy when you can't drag and drop a logical hierarchy from the logical layer to the presentation layer.

You can edit a presentation hierarchy's properties to set permissions and apply role-based access control and add data filters. If you're adding or editing a level-based hierarchy, then you can add or delete levels and modify a level's properties. See Add and Modify Presentation Hierarchy Levels.

You can select the **Hide if** field and write an expression to hide a presentation hierarchy. See Write an Expression to Hide a Presentation Object.

Semantic Modeler *doesn't* create an alias when you change a presentation hierarchy's name. If you need to track a hierarchy's previous names, then Oracle recommends that you use the **Alternative Names** field to create and manage alternative names for the hierarchy. For more information about alternative names, see About Alternative Names for Presentation Objects.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Presentation Layer**.
4. In the Presentation Layer pane locate and double-click the presentation table that you want to add a hierarchy to.
5. In the presentation table, click the **Hierarchies** tab and then click **Add Hierarchy**. Browse for and select the hierarchy to add it to the presentation table. Confirm that the added hierarchy is highlighted in the Hierarchies pane.
6. Optional: In the **Description** field, enter a description that is displayed when a user creating visualizations or analyses hovers over the table in the subject area.
7. Optional: Select the **Hide if** field and provide an expression that controls whether the hierarchy is available to users when they create visualizations and analyses.
8. Click **Save**.

Add and Modify Presentation Hierarchy Levels

You can manually add a level to a level-based presentation hierarchy. Presentation levels are displayed within hierarchical columns in the corresponding subject area end users access to create visualization and analyses.

You can specify one or more display columns for a level. Display columns define the columns used for display for that level on drill-down. For example, if two columns called *Name* and *ID* are at the same level, you can choose to display *Name* because it's more user-friendly. The presentation level's available display columns are based on which key columns for the corresponding logical level have the **Use for display** option selected.

Semantic Modeler *doesn't* create an alias when you change a presentation hierarchy level's name. If you need to track a hierarchy level's previous names, then Oracle recommends that you use the **Alternative Names** field to create and manage alternative names for the hierarchy level. For more information about alternative names, see About Alternative Names for Presentation Objects.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Presentation Layer**.
4. In the Presentation Layer pane expand the presentation table that contains the hierarchy you want to work with, and from the table's list locate and double-click the hierarchy.
5. In the presentation table's page, click the **Hierarchies** tab and then in the Hierarchies pane locate the hierarchy you want to work with.
6. Optional: To add a level to the hierarchy, click **Level** to select and add a level. You can select and add a level from any available hierarchy.
7. Optional: To modify an existing level, click the level to display its properties pane.
8. Optional: In the **Description** field, enter a description that is displayed when a user creating visualizations or analyses hovers over the table in the subject area.
9. Optional: In the **Logical Level** field, click **Select** and select a logical level for the presentation level.
10. In the **Display Columns** field, select the columns to use for display for the hierarchy level on drill-down.
11. Click **Save**.

Write an Expression to Hide a Presentation Object

You can write an expression to hide a subject area, presentation table, presentation column, or hierarchy. These expressions determine if or when hidden objects are visible to users when they create visualizations and analyses.

The presentation object's **Hide if** field controls a presentation object's visibility and doesn't affect the ability to access the object. For example, you can query a hidden presentation object using a tool such as `nqcmd`.

You can specify three different types of expressions in the **Hide If** field:

- **Constant** - Use any non-zero constant in the field to hide the object. Use zero (0) or leave the field blank to display the object.

- **Session variable** - You can use a session variable in the expression to control whether the object is hidden. If the expression evaluates to a non-zero value, the object is hidden. If the expression evaluates to zero, is empty, or has no value definition, the object is displayed. The session variable must be populated using a session initialization block or a row-wise initialization block. You must properly populate the session variable to control the visibility.

  The SQL for the `init` block can use `CASE` statements to control whether to return zero or a non-zero number in the session variable. For example:

  ```
  VALUEOF(NQ_SESSION."VISIBLE")
  ```

  Session variable names that include periods must be enclosed in double quotes.

- **Session variable comparison** - You can use an equality or inequality comparison to control whether the object is hidden, using the following form:

  ```
  'session_variable_expression' '=|<>' 'constant'
  ```

  If the expression evaluates to zero, null, or empty, the object is displayed. If the expression evaluates to a non-zero value, the object is hidden. For example:

  ```
  NQ_SESSION."VISIBLE" = 'ABC'
  NQ_SESSION."VISIBLE" <> 'ABC'
  ```

  You must enclose session variable names that include periods in double quotes.

  You can use any scalar function supported by Oracle Analytics in the `Hide object if` expression. Scalar functions include any function that accepts a simple value for each of its arguments and returns a single value. You can use the functions listed, except for functions that return non-deterministic results like `RAND`, `NOW`, `CURRENT_DATE`, `CURRENT_TIMESTAMP`, and `CURRENT_TIME`.

  - All String Functions.
  - Math Functions, except RAND.
  - Calendar Date/Time Functions, except NOW, and CURRENT_DATE.
  - Conversion Functions such as CAST, IFNULL, TO_DATETIME, and VALUEOF.

  For example, this expression checks to see if the *NQ_SESSION.VISIBLE* session variable begins with the letters *ABC*:

  ```
  LEFT(VALUEOF(NQ_SESSION."VISIBLE"), 3) = 'ABC'
  ```

  The following expression checks to see if the given variable begins with *ExtnAttribute*:

  ```
  Left(VALUEOF(NQ_SESSION."ADF_LABEL_ORACLE.APPS.CRM.MODEL.ANALYTICS.
  APPLICATIONMODULE.CRMANALYTICSAM_CRMANALYTICSAMLOCAL_CRMANALYTICSAM.
  OPPORTUNITYAM.OPPORTUNITY_EXTNATTRIBUTECHAR001"), 13) = 'ExtnAttribute'
  ```

Run **Check Consistency** to find any inconsistencies in the visibility filter expression.

Work with Localization

This topic provides information about localizing presentation objects.

Topics:

- Clear All Name and Description Variables
- Generate Localization Keys and Name and Description Variables
- Externalize Strings for a Subject Area
- Externalize Strings for All Subject Areas
- Translate Strings

Modify or Delete Individual Localization Keys and Variables

Semantic Modeler automatically creates localization keys and name variables for all subject areas and presentation objects. You can modify or delete these keys and variables. You can also manually add description variables.

Localization keys are assigned when presentation objects are created. When externalized and used to translate strings, a localization key serves to map the object's name with its translated value. Be careful when modifying localization keys because it can break this mapping.

If you modify or manually add variables, then you must include VALUEOF(NQ_SESSION.CN) in name variables (for example, VALUEOF(NQ_SESSION.CN_A_-_Sample_Sales_Offices) and VALUEOF(NQ_SESSION.CD) in description variables (for example, VALUEOF(NQ_SESSION.CD_A_-_Sample_Sales_Offices).

When the localization keys and variables are finalized, you can externalize strings for the subject area. See Externalize Strings for a Subject Area.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Presentation Layer**.
4. In the Presentation Layer pane locate and double-click a subject area or the presentation table that you want to modify keys and variables for.
5. In the subject area or presentation table page, click the **Localization** tab.
6. Optional: To update a localization key, do one of the following:
   - For a subject area, click the **Localization Key** field and delete or modify its value.
   - For a presentation table, locate an object and double-click its table row. Click the object's **Localization Key** field and delete or modify its value.
7. Optional: To update a description value, do one of the following:
   - For a subject area, click the **Name Variable** or **Description Variable** field and delete or modify its value.
   - For a presentation table, locate an object and double-click its table row. Click the object's **Localization Key** field or **Description Variable** and modify or delete its value.
8. Click **Save**.

Clear All Name and Description Variables

You can clear name and description variables for a subject area only, for a subject area and all of its child objects, or for a presentation table and all of its objects.

1. On the Home page, click **Navigator** and then click **Semantic Models**.

2. In the Semantic Models page, click a semantic model to open it.
3. Click **Presentation Layer**.
4. In the Presentation Layer pane locate and double-click a subject area or the presentation table that you want to clear variables for.
5. In the subject area or presentation table, click the **Localization** tab.
6. To clear variables, do one of the following:
   - For a subject area, click **Clear Variables** and select which variables to clear, and then specify if you want to clear variables for the subject area object only or the subject area and all of its child objects.
   - For a presentation table, click **Clear Variables** and select which variables to clear.
7. Click **Clear**.
8. Click **Save**.

Generate Localization Keys and Name and Description Variables

You can generate localization keys and name and description variables for a subject area only, for a subject area and all of its child objects, or for a presentation table and all of its objects.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Presentation Layer**.
4. In the Presentation Layer pane locate and double-click a subject area or the presentation table that you want to generate localization keys and variables for.
5. In the subject area or presentation table, click the **Localization** tab.
6. To generate variables, do one of the following:
   - For a subject area, click **Generate Variables** and select which variables to generate, and then specify if you want to generate variables for the subject area object only or the subject area and all of its child objects.
   - For a presentation table, click **Clear Variables** and select which variables to generate.
7. Click **Generate**.
8. Click **Save**.

Externalize Strings for a Subject Area

You can externalize strings for a subject area, its presentation tables, hierarchies, columns, and their descriptions. Externalizing strings for the subject area outputs a CSV file.

After you externalized the strings for the subject area and its objects, you use the resulting files to translate the strings. See Translate Strings.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.

3. Click **Presentation Layer**.
4. In the Presentation Layer pane locate and right-click the subject area you want to externalize strings for.
5. Click **Externalize Strings**.
6. In the **Name** field, enter a name for the outputted CSV file.
7. Click **Externalize**.

Externalize Strings for All Subject Areas

You can externalize strings for all of the presentation layer's subject area, their presentation tables, hierarchies, columns, and their descriptions. Externalizing strings for all subject areas outputs a CSV file.

After you externalized the strings for the subject area and its objects, you use the resulting files to translate the strings. See Translate Strings.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Presentation Layer**.
4. In the header, click **Page Menu** and then click **Externalize Strings**.
5. In the **Name** field, enter a name for the outputted CSV file.
6. Click **Externalize**.

Translate Strings

After you externalized translation keys and strings, you can use the resulting files to translate the strings for the presentation objects.

Note the contents of the output file:

- The first column contains the actual semantic model object names with a prefix indicating its object type.
- The second column contains the session variables that correspond to the name of each object or description, with a default prefix of CN_ for custom names and CD_ for custom descriptions.
- The third column contains the translation keys that correspond to the name of each object.

1. Open each string file Add a fourth column called Language. In this column, specify the code for the language in which the name was translated, such as de.
2. Load each string file into a database table.
3. In the Semantic Modeler, import the table into the physical layer.
4. Load the translated strings using row-wise initialization blocks. Ensure that you set the target of the initialization block to **Row-wise initialization** and that the execution precedence is set correctly. For example:

a. Create a session initialization block that has the data source from a database, using a SQL statement such as the following one:

```
SELECT 'VALUEOF(NQ_SESSION.WEBLANGUAGE)' FROM DUAL
```

b. In the Session Variable Initialization Block dialog box for SET Language, specify the LOCALE session variable for the Variable Target. This specification ensures that whenever a user signs in, the WEBLANGUAGE session variable is set. Then this variable sets the LOCALE variable using the initialization block.

c. Create another session initialization block that creates a set of session variables using a database-specific SQL statement such as the following one in the Session Variable Initialization Block Data Source dialog box:

```
select SESSION_VARIABLE, TRANSLATION from external where
LANGUAGE =
'VALUEOF(NQ_SESSION.LOCALE)'
```

This block creates all the variables whose language matches the language that the user specified during sign-in.

d. In the Session Variable Initialization Block Variable Target dialog box, set the target of the initialization block to **Row-wise initialization**.

e. In the Execution Precedence area of the Session Variable Initialization Block dialog box, specify the previously created initialization block, so that the first block that you created earlier is run first.

5. Click **Save**.

11

Work with Logical Hierarchies

This chapter contains information to help you understand how to create and manage logical hierarchies.

Topics:

About Working with Logical Hierarchies

In the logical layer, a dimension object represents a hierarchical organization of logical columns (attributes).

You can associate one or more logical dimension tables with one dimension object.

Common dimensions include time periods, products, markets, customers, suppliers, promotion conditions, raw materials, manufacturing plants, transportation methods, media types, and time of day. Dimensions exist in the logical layer and in the presentation layer.

In each dimension, you organize logical columns into the structure of the hierarchy. The structure represents the organization rules and reporting needs required by your business and provides the metadata the Oracle Analytics query engine uses to drill into and across dimensions to get detailed views of the data.

There are two types of logical hierarchies:

- **Dimensions with level-based hierarchies** - These are also called structure hierarchies. In level-based hierarchies, members are of several types, and members of the same type, such as employee or assembly occur only at a single level.
- **Dimensions with parent-child hierarchies** - These are also called value hierarchies. In parent-child hierarchies, members all have the same type.

Semantic Modeler also supports a special type of level-based dimension called a time dimension that provides special functionality for modeling time series data.

You can expose logical hierarchies to workbooks and analyses by creating presentation hierarchy objects that are based on particular logical hierarchies. Creating hierarchies in the presentation layer enables users to create hierarchy-based queries. See Work with Presentation Hierarchies and Levels.

You can also expose hierarchies by adding one or more columns from each hierarchy level to a subject area in the presentation layer.

Create and Manage Level-Based Hierarchies

This topic provides information to help you understand and create a level-based hierarchy and its dimensions.

Topics:

About Level-Based Hierarchies

Each business model can have one or more dimensions, each dimension can have one or more logical levels, and each logical level has one or more attributes (columns) associated with it.

When you create logical levels, first create a Grand Total level and then create child levels, working down to the lowest level.

The following are the parts of a dimension:

Grand Total level

The Grand Total level represents the sum of all totals for a dimension. Each dimension can have just one Grand Total level. The Grand Total level doesn't contain dimensional attributes and doesn't have a level key. You can associate measures with a Grand Total level. The aggregation level for those measures is the grand total for the dimension. The Grand Total level can exist without any columns.

Level

Levels must have at least one column. You don't need to explicitly associate all of the columns from a table with logical levels. Any column that you don't associate with a logical level is automatically associated with the lowest level in the dimension that corresponds to that dimension table. You must associate all logical columns in the same dimension table with the same dimension.

A dimension can have an unlimited number of levels.

Hierarchy

Each dimension contains one or more hierarchies. All hierarchies must have a common leaf level. For example, a time dimension might contain a fiscal hierarchy and a calendar hierarchy, with a common leaf level of Day. In this example, Day has two named parent levels, Fiscal Year and Calendar Year that are both children of the All root level.

Unlike hierarchies in the presentation layer, in the logical layer logical hierarchies aren't defined as independent metadata objects. Logical hierarchies exist implicitly through the relationships between levels.

You can define intermediate levels in your hierarchies to avoid having very large numbers of members at one level. For example, if you're creating a Product dimension for an automotive company that tracks data on 500 different car models, you might create some finer-grained hierarchical levels such as SUVs, Subcompacts, and Midsize Sedans. You could improve query performance and make reports and diagrams easier to read and navigate. See Create Logical Levels in a Logical Dimension Table .

Level keys

Each logical level, except the Grand Total level, must have one or more attributes that compose a level key. The level key defines the unique elements in each logical level. You must associate the dimension table logical key with the lowest level of a dimension.

A logical level can have multiple level keys. When a logical level has multiple level keys, specify a key as the primary key for the level. All dimension sources that have aggregate content at a specified level need to contain the column that's the primary key of that level. Each logical level should have one level key that's displayed when a user selects the object to drill down. You can use any level key to provide user access to the level.

You must create a unique level key. To create a unique level key with month, include the year attribute as part of the key.

Ensure that your level key is unique by including higher-level attributes to prevent queries from returning unexpected results. For example, when the Oracle Analytics query engine needs to combine result sets from multiple physical queries, the results might exclude expected rows that aren't unique according to the level key definition.

Create meaningful level keys using common business keys such as *Month_name='2022 July'*, rather than generated surrogate keys such as *time_key='1023793'*. The generated surrogate keys are physical artifacts that only apply to a single instance of a source table. A business key can map to any physical instance for that logical column, for example, *month_name* might map to a detailed table, an aggregate table from an aggregate star, or a column in a federated data source. The physical layer can use surrogate keys in the joins but Oracle recommends using business keys.

Time dimensions and chronological keys

You can identify a dimension as a time dimension. Use the following guidelines when setting up and using time dimensions:

- At least one level of a time dimension must have a chronological key. See Select and Sort Chronological Keys in a Time Dimension.
- All time series measures using the `AGO`, `TODATE`, and `PERIODROLLING` functions are in time levels. `AGO`, `TODATE`, and `PERIODROLLING` aggregates are created as derived logical columns.
- `AGO`, `TODATE`, and `PERIODROLLING` functionality isn't supported either on fragmented dimensional logical table sources, or on fact sources fragmented on the same time

dimension. Fact sources may be fragmented on other dimensions. See Work With Logical Table Source Data Fragmentation.

See About Time Series Functions.

About Hierarchy Structures

A logical hierarchy can have a balanced, ragged, or skip-level structure.

Balanced hierarchy

A balanced hierarchy's structure contains members that descend to the same level and where each member's parent is immediately above it.

Unbalanced or ragged hierarchy

An unbalanced or ragged hierarchy is a hierarchy where the leaves (members with no children) might not have the same depth. For example, a site can choose to have data for the current month at the day level, previous month's data at the month level, and the previous five years' data at the quarter level.

User applications can use the `IS_LEAF` function to determine whether to allow moving down from any particular member.

A missing member is implemented in the data source with a null value for the member value. All computations treat the null value as a unique child within its parent. Level-based measures and aggregate-by calculations group all missing nodes together.

Unbalanced hierarchies aren't necessarily the same as parent-child hierarchies. Parent-child hierarchies are unbalanced by nature. Unbalanced level-based hierarchies are possible.

Skip-level hierarchy

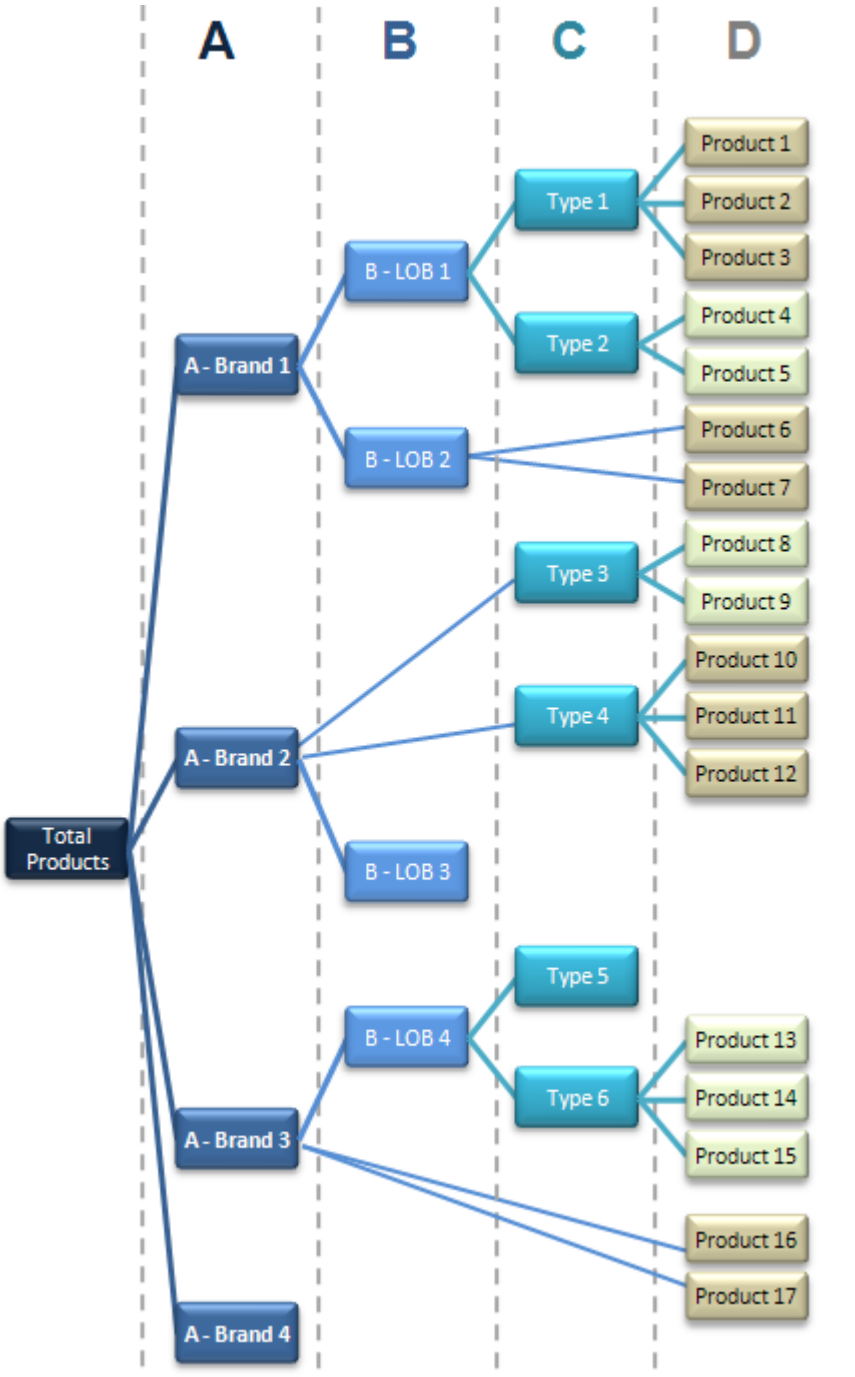
A skip-level hierarchy is a hierarchy where there are members that don't have a value for a particular ancestor level. For example, in a Country-State-City-District hierarchy, the city *Washington D.C.* doesn't belong to a State. In this case, you can drill down from the Country level (USA) to the City level (Washington D.C.) and below.

In a query, skipped levels aren't displayed, and don't affect computations. When sorted hierarchically, members appear under their nearest ancestors.

A missing member at a particular level is implemented in the data source with a null value for the member value. All computations treat the null value as a unique child within its parent. Level-based measures and aggregate-by calculations group all skip-level nodes together.

Example of hierarchy containing ragged and skip-level

The image shows a hierarchy with both ragged and skip-level characteristics. For example, A-Brand 4, B-LOB 3, and Type 5 are unbalanced branches, while skips are present between A-Brand 2 and Type 3, B-LOB 2 and Product 6, and others.



About Using Dimension Hierarchy Levels in Level-Based Hierarchies

Learn how to use dimension hierarchical levels.

Dimension hierarchical levels can be used to perform the following actions:

- Set up aggregate navigation.
- Configure level-based measure calculations. See Level-Based Measure Calculations.
- Determine what attributes are displayed when users drill down in their data requests.

Automatically Create Dimensions with Level-Based Hierarchies

You can set up a dimension automatically from a logical dimension table if a dimension for that table doesn't exist.

To create a dimension with a level-based hierarchy automatically, a semantic model examines the logical table sources and the column mappings in those sources and uses the joins between physical tables in the logical table sources to determine logical levels and level keys. As a best practice, create a dimension table with a level-based hierarchy after all the logical table sources have been defined for a dimension table.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer** and locate and double-click a logical dimension table that isn't associated with any dimension.
4. In the logical table, click the **Hierarchy** tab.
5. In the **Hierarchy Type** field, select **Level-Based** or **Parent-Child**.
6. Click **Save**.

Manually Create Dimensions in Level-Based Hierarchies

You can associate each dimension with attributes (columns) from one or more logical dimension tables and level-based measures from logical fact tables.

It's a best practice to ensure that the physical hierarchy type set in the physical layer matches the dimension properties you select in the logical layer. Also, be sure that you set the Ragged and Skipped Levels dimension properties correctly so that the queries work properly.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. In the Logical layer, right-click a business model and select **Create Logical Table**.
4. In **Name**, type a name for the logical table. Click the **Type** field and select **Dimension**.
5. Click **OK**.
6. In the new logical table's tabs click the **Hierarchy** tab.
7. Click the **Hierarchy Type** field and either select **Level-Based**, or if the dimension is a time dimension, select **Time**.

   The **Default root level** field is automatically populated after you associate logical columns with a dimension level.
8. If the hierarchy type is Level-Based, click either **Ragged** or **Skipped Levels**.
9. Click **Save**.

Create Logical Levels in a Logical Dimension Table

When you create logical levels in a logical dimension table, you also create the hierarchy by identifying the type of level and defining child levels.

If you're defining the level as a `Grand Total level`, the default value is 1.

The number doesn't have to be exact, but ratios of numbers from one logical level to another should be accurate. You can retrieve the row count for the level key and use that number as the number of elements.

The Oracle Analytics query engine uses this number when selecting which aggregate source to use. For example, when aggregate navigation is used, multiple fact sources exist at different grains. The Oracle Analytics query engine multiplies the number of elements at each level for each qualified source as a way to estimate the total number of rows for that source. The Oracle Analytics query engine then compares the result for each source and selects the source with the lowest number of total elements to answer the query. The source with the lowest number of total elements is assumed to be the fastest.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. In the Logical layer, double-click a logical table, and in the logical table's tabs click **Hierarchies**.
4. Click the **Hierarchy Type** field and either select **Level-Based**, or if the dimension is a time dimension, select **Time**.
5. Click **New Level**.
6. Add and configure the Grand Total level and Detail level.
7. Optional: Rename the Grand Total level and Detail level. For example, Products Total or Products Detail.
8. To add and define child logical levels, select the hierarchy and click **New Level**.
9. In the **Elements at this level** field, specify the number of elements that exist at this logical level.
10. If measure values at a particular level fully constitute aggregated measures at its parent level, select **Supports rollup to higher level**.
11. For all levels except Total, select the **Primary Key**.
12. For all levels except Total, select the **Display Key**
13. Click **Save**.

Associate a Logical Column and Its Table with a Dimension Level

After you create all logical levels within a dimension, you associate one or more columns from the logical dimension table to each logical level except the Grand Total level.

The first time you add a column to a dimension it associates the logical table to the dimension. The drag and drop action associates the logical column with that level of the dimension. To associate the logical level with that logical column, drag a column from one logical level to another.

You must associate the logical column or columns that comprise the logical key of a dimension table with the lowest level of the dimension.

After you associate a logical column with a dimension level, the tables where these columns exist are displayed in the **Tables** tab of the Dimensions dialog box.

For examples, see: About Level-Based Measure Calculations and Grand Total Dimensional Hierarchy Example.

For time dimensions, ensure that all time-related logical columns in the source table are defined in the time dimension. For example, if a time-related logical table contains the columns Month Name and Month Code, you must ensure that both columns are defined at the appropriate level within the dimension.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer** and locate and double-click the logical table that you want to associate with a dimension level.
4. Click the **Columns** tab.
5. Click to select the logical column, and then click **Detail View**.
6. Click the **Level** field and select a logical level, making sure not to select the Grand Total Level.
7. Click **Save**.

About Level-Based Measure Calculations

A level-based measure is a column whose values are always calculated to a specific level of aggregation.

You can set up columns to measure CountryRevenue, RegionRevenue, and CityRevenue. For example, a company might want to measure its revenue based on the country, region, and city.

When a query containing a presentation hierarchy includes a level-based measure column, and the query grain is higher than the level of aggregation specific to the column, the query results return null. If the request only contains ordinary columns and no hierarchical columns, the level-based measure isn't replaced with null.

You can create an AllProductRevenue measure as a level-based measure at the Grand Total level. Level-based measures allow a single query to return data at multiple levels of aggregation. Level-based measures are also useful in creating share measures, calculated by taking some measure and dividing it by a level-based measure to calculate a percentage. For example, you can divide salesperson revenue by regional revenue to calculate the share of the regional revenue each salesperson generates.

For example, to set up these calculations, you need to build a dimensional hierarchy in your semantic model that contains the Grand Total, Country, Region, and City levels. This hierarchy contains the metadata that defines a one-to-many relationship between Country and Region and a one-to-many relationship between Region and City. For each country, there are many regions, but each region is in only one country. Similarly, for each region, there are many cities, but each city is in only one region.

After building a dimensional hierarchy, you need to create three logical columns one each for CountryRevenue, RegionRevenue, and CityRevenue. The columns use the

Revenue logical column as its source. The Revenue column has a default aggregation rule of `SUM` and has sources in the underlying databases.

Assign the CountryRevenue, RegionRevenue, and CityRevenue columns to the Country, Region, and City levels, respectively. Each query that requests one of these columns returns the revenue aggregated to its associated level.

Grand Total Dimensional Hierarchy Example

Use this example to learn how to use a grand total dimensional hierarchy with revenue.

If your product dimensional hierarchy contains TotalProducts (Grand Total level), Brands, and Products levels, and a Revenue column defined with a default aggregation rule of Sum, you can then create an AllProductRevenue logical column. The AllProductRevenue column uses Revenue as its source. Associate the AllProductRevenue column to the Grand Total level. Each query that includes the AllProductRevenue column returns the total revenue for all products. The value is returned regardless of any constraints on Brands or Products.

If you have constraints on columns in other tables, the grand total is limited to the scope of the query. For example, if the scope of the query asks for data from 2000 and 2021, the grand total product revenue is for all products sold in 2000 and 2021.

If you have three products, A, B, and C with total revenues of 100, 200, and 300 respectively, then the grand total product revenue is 600, the sum of each product's revenue. If you have set up a semantic model as described in this example, the following query produces the results listed:

```
SELECT product, productrevenue, allproductrevenue
FROM sales_subject_area
WHERE product IN 'A' or 'B'
```

The results are as follows:

```
PRODUCT;;PRODUCTREVENUE;;ALLPRODUCTREVENUE
A;;;;;;;;100;;;;;;;;;;;;;600
B;;;;;;;;200;;;;;;;;;;;;;600
```

The AllProductRevenue column always returns a value of 600, regardless of the products on which the query constrains.

Identify the Primary Key for a Dimension Level

Use a logical dimension table's **Hierarchy** tab and **Primary Key** field to identify the column to use as the dimension level's primary key.

You can't use a derived logical column that's the result of a `LOOKUP` function as part of a primary logical level key. This limitation exists because the `LOOKUP` operation is applied after aggregates are computed, but level key columns must be available before the aggregates are computed because they define the granularity at which the aggregates are calculated.

You can use a derived logical column that's the result of a `LOOKUP` function as a secondary logical level key.

If the level is in a time dimension, you can select chronological keys and sort the keys by name.

To help manage primary keys, you can go to the logical table's **Columns** tab, locate the column used as a primary key, and add information to its **Description** field.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. In the Logical layer, right-click on a logical dimension table, then click **Edit**.
4. Click **Logical Layer** and locate and double-click the logical table with the dimension level that you want to add a primary key to.
5. Click the **Hierarchy** tab.
6. Click to select a level below the Grand Total level.
7. Click the **Primary Key** field and select a level key from the list and save changes. If only one key exists, it is the primary key by default.
8. Click **Save**.

Select and Sort Chronological Keys in a Time Dimension

At least one level of a time dimension must have a chronological key. You can select one or more chronological keys for any level and then sort keys in the level, but Oracle Analytics uses only the first chronological key.

Pay attention to the column order in a chronological key with many columns. You set the column order using a SQL `ORDER BY` clause on the columns to reflect the real-world chronological order in the **Chronological Key** field. Since the range for quarters is 1 to 4 with 4 quarters in a year, using an `ORDER BY` clause with the Quarter before the Year (*Quarter*, *Year*) is incorrect. The incorrect order shows all first quarters across all years, before displaying any second quarters, and so on. To correct the results, use (*Year*, *Quarter*) in the `ORDER BY` clause.

For information about creating a time dimension, see Manually Create Dimensions in Level-Based Hierarchies.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. In the Logical layer, double-click a logical table, and in the logical table's tabs click **Hierarchies**.
4. Click a logical level below the Grand Total level.
5. Click the **Chronological Key** field and select a chronological key.
6. Click **Save**.

Add a Dimension Level to the Preferred Drill Path

You can use the Preferred Drill Path field to identify the drill path to use when users drill down in their data requests.

You should use a preferred drill path only to specify a drill path that's outside the normal drill path defined by the dimensional level hierarchy. A drill path is most commonly used to drill from one dimension to another. You can delete a logical level from a drill path or reorder a logical level in the drill path.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.

3. Click **Logical Layer** and locate and double-click the logical table that you want to add a drill path to.
4. In the logical table, click the **Hierarchy** tab.
5. Click a logical level, go to the **Preferred Drill Path** field, and click **Add Table**.
6. In Select Logical Level, search for and select a logical level, and then click **Select**.

   You can select logical levels from the current dimension or from other dimensions.
7. Click **Save**.

Create and Manage Parent-Child Hierarchies

This topic provides information to help you understand and create a parent-child hierarchy.

Topics:

About Parent-Child Hierarchies

A parent-child hierarchy is a hierarchy of members that all have the same type. For example, employee or assembly.

This contrasts with level-based hierarchies, where members of the same type occur only at a single level of the hierarchy.

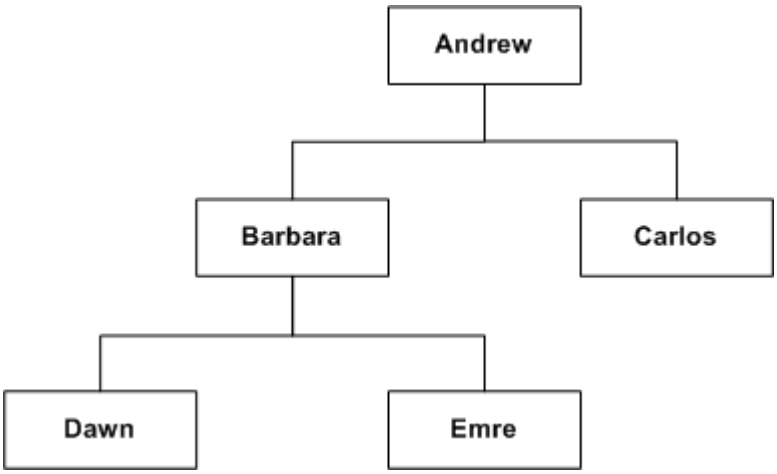
A common real-life parent-child hierarchy occurrence is an organizational reporting hierarchy chart. In an organizational reporting hierarchy chart, the following can apply:

- Each individual in the organization is an employee.
- Each employee, apart from the top-level managers, reports to a single manager.
- The reporting hierarchy has many levels.

These conditions illustrate the basic features that define a parent-child hierarchy, namely:

- A parent-child hierarchy is based on a single logical table, for example, the *Employees* table.
- Each row in the table contains two identifying keys, one to identify the member itself, the other to identify the parent of the member, for example, *Emp_ID* and *Mgr_ID*.

The image shows an example of a multi-level parent-child hierarchy.



The following table shows how this parent-child hierarchy could be represented by the rows and key values in an Employees table.

Emp_ID	Mgr_ID
Andrew	*null*
Barbara	Andrew
Carlos	Andrew
Dawn	Barbara
Emre	Barbara

You can expose logical parent-child hierarchies to users by creating presentation hierarchies that are based on particular logical hierarchies. Creating hierarchies in the presentation layer enables users to create hierarchy-based queries.

See Work with Presentation Hierarchies and Levels .

About Levels and Distances in Parent-Child Hierarchies

All the dimension members of a parent-child hierarchy occur in a single logical column.

In a parent-child hierarchy, the parent of a member is in another row in the same logical column, pointed to by the parent key. In a level-based hierarchy, the parent of a member is in a different logical column in the same row. Navigation in a parent-child hierarchy follows data values, while navigation in a level-based hierarchy follows the metadata structure.

In level-based hierarchies, each level is named, and occupies a position in the hierarchy that corresponds to a real-world attribute or category useful for analysis. In level-based hierarchies the number of levels is fixed at design time. There is no limit to the depth of a parent-child hierarchy, and the depth can change at run time due to new data.

Every parent-child hierarchy has two system-generated entities, Total and Detail, that are automatically defined when the logical hierarchy is created. The Detail entity

contains all the hierarchy members. These two system-generated entities are different from the implicit, inter-member levels between ancestors and descendants in a parent-child hierarchy. The implicit levels are referred to as parent-child hierarchical levels.

Closely associated with levels is the concept of distance in parent-child hierarchies. The distance of one member from another is the number of parent-child hierarchical levels from the member to an ancestor or to a descendant. For example, the distance from a member to its parent is always 1. See About Parent-Child Hierarchies for an example.

About Parent-Child Relationship Tables

In a relational table, the relationships between different members in a parent-child hierarchy are implicitly defined by the identifier key values in the associated base table.

For each parent-child hierarchy defined in a relational table, you must also explicitly define the inter-member relationships in a separate parent-child relationship table.

The parent-child relationship table must include four columns:

- A column that identifies the member
- A column that identifies an ancestor of the member

  An ancestor is the parent of the member, or a higher-level ancestor.
- A relationship distance column that specifies the number of parent-child hierarchical levels from the member to the ancestor
- A leaf node column that indicates if the member is a leaf member (`1`=*Yes*, `0`=*No*)

The column names can be user-defined. The data types of the columns must satisfy the following conditions:

- The member and ancestor identifier columns have the same data type as the associated columns in the logical table that contains the hierarchy members.
- The distance and leaf columns are `INTEGER` columns.

For the rows in a parent-child relationship table:

- Each member must have a row pointing at itself, with distance zero.
- Each member must have a row pointing at each of its ancestors. For a root member, this is a termination row with null for the parent and distance values.

The example shown in the table uses text strings for readability, but you normally use integer surrogate keys for `member_key` and `ancestor_key`, if they exist in the source dimension table.

The table shows an example of a parent-child relationship table with rows that represent the inter-member relationships for the employees. See the figure in About Parent-Child Hierarchies.

Member_Key	Ancestor_Key	Distance	Isleaf
Andrew	Andrew	0	0
Barbara	Barbara	0	0
Carlos	Carlos	0	0
Dawn	Dawn	0	0
Emre	Emre	0	0

Member_Key	Ancestor_Key	Distance	Isleaf
Andrew	null	null	0
Barbara	Andrew	1	0
Carlos	Andrew	1	1
Dawn	Barbara	1	1
Dawn	Andrew	2	1
Emre	Barbara	1	1
Emre	Andrew	2	1

You must generate the parent-child relationship table and then import it into the physical layer before associating it with the parent-child hierarchy. You use the Hierarchy Tab's **Generate Relationship Table** functionality to generate scripts that are run to create and populate the parent-child relationship table.

When you generate the relationship table, two scripts are created: one script to create the table, and the other script to load the table. Note the following information about the create and load scripts:

- You run the create script only once, to create the parent-child relationship table in the data source.
- You must run the load script after each time the data changes in the dimension table. Because of this, you typically call the load script in your ETL processing. The load script reloads the entire parent-child relationship table; it isn't incremental.

For information about generating the relationship table scripts, see Generate Scripts to Create a Parent-Child Relationship Table.

Create Dimensions with Parent-Child Hierarchies

The key elements that you must define for a parent-child hierarchy are the identifier columns for the member and the parent of the member.

This basic principle applies to all parent-child hierarchies, regardless of the data source that the hierarchy is derived from.

Parent-child hierarchies based on relational tables must have an accompanying parent-child relationship table. See About Parent-Child Relationship Tables and Define Parent-Child Relationship Tables.

In the Semantic Model pane, click **Logical Layer**, then double-click on any logical dimension table to view their primary keys and the other columns in the table.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**, click **Create**, and then select **Create Logical Table**.
4. In Create Logical Table, enter a name in the **Name** field.
5. In the **Type** field, select **Dimension**.
6. In the **Business Model** field, select a business model. Click **OK**.

7. In the Logical Table's tabs, click **Sources** and add a table source in one of the following ways:
   - Click **Add Physical Table**, and then select **Add Physical Table** to select a physical table source.
   - Click **Create Logical Table Source** to create and add a new logical table source.
8. In the Logical Table's tabs, click **Hierarchy**, click the **Hierarchy Type** field and select **Parent-Child**.
9. Scroll to the **Relationship Table** field, click **Select** and choose a physical table.
10. Go to the **Member Key**, **Display Key**, and **Parent Key** fields and select columns.
11. Click **Save**.

If the logical table is from a relational table source, you must continue the dimension definition process by setting up the parent-child relationship table for the hierarchy.

Generate Scripts to Create a Parent-Child Relationship Table

From the logical table's **Hierarchy** tab you can generate SQL scripts to generate and load the parent-child relationship tables into your data source.

After you run the scripts and generate the parent-child relationship table, you add them to the semantic model's physical layer to make them available to use in the parent-child hierarchy.

See About Parent-Child Relationship Tables and Add the Parent-Child Relationship Table to the Semantic Model.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer** and locate and double-click the logical table with the parent child hierarchy that you want to generate a relationship table for.
4. In the Logical Table's tabs, click **Hierarchy**, click the **Generate Relationship Table**.
5. In the Generate Relationship Table Scripts dialog, confirm the column name in the **Member Column** field and in the **Parent Column** field choose a column. Confirm the other fields in the dialog.
6. Click **Download Script**.
7. Run the downloaded scripts to create the parent-child relationship tables.

Add the Parent-Child Relationship Table to the Semantic Model

For measures in fact tables that are aggregated by rolling up the facts from lower-level members, you must edit physical layer joins to include the parent-child relationship table.

You need to add the parent-child relationship table to the appropriate logical table source.

For fact tables containing pre-aggregated data for a parent-child hierarchy or for individual contribution measures, you should join the parent-child dimension table directly with the fact table rather than joining through the parent-child relationship table.

Joining the parent-child dimension table directly with the fact table ensures that the pre-aggregated value or individual contribution value is returned, rather than rolling up all the descendants. When pre-aggregated data is populated for all members, don't add the parent-child relationship table to the logical table source to avoid over counting.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**. Right-click a physical table, select **Show Physical Diagram**, and select **Selected Tables Only and Direct Joins**.
4. Right-click each direct join from the dimension table to each of the fact tables and select **Delete Join**.
5. Click **Logical layer** and locate and double-click the logical table source for the logical fact table that's used in your parent-child hierarchy.
6. In the Logical Table's tabs, click **Joins** and create a join from the parent-child relationship table to the dimension table using the ancestor key.
7. Create joins from the fact tables to the parent-child relationship table using the member key.
8. In the Logical Table's tabs, click **Sources** to edit the logical table source for the logical fact table that's used in your parent-child hierarchy.
9. In the **Sources** tab, click **Add Physical Table**.
10. Locate the parent-child relationship table in the Physical layer, and then click **Select**.
11. Click **Save**.

Define Parent-Child Relationship Tables

The parent-child relationship table must have at least four columns that describe how the inter-member relationships are derived in the logical table selected for the hierarchy.

When you configure the parent-child relationship table, you can select an existing relationship table. Or you can create, generate, and import the required relationship table into the physical layer. For information about how to generate the table, see Generate Scripts to Create a Parent-Child Relationship Table.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. In the Logical layer, double-click a logical table, and then click the logical table's **Hierarchy** tab.
5. In the **Hierarchy Type** field and select **Parent-Child**.
6. Scroll to **Relationship Table**, click **Select** to select and add a logical table source, and then in the **Relationship Table** field, click **Add**.
7. In Select Physical Table, click a Physical Table to act as the parent-child relationship table for your hierarchy, and then click **Select**.
8. Map the **Member Key**, **Ancestor Key**, **Relationship Distance**, and **Leaf Node Identifier** column fields to the physical parent-child relationship table.
9. Click **Save**.

About Modeling Aggregates for Parent-Child Hierarchies

Fact tables in level-based hierarchies might only contain facts for a single level of the hierarchy.

Facts for higher-level dimension members can be calculated by aggregating the facts from the lower-level fact table or from a higher-level summary table.

In contrast, parent-child hierarchies require data modelers to make some additional decisions related to how to store the base facts in the fact table and how to aggregate the base facts to obtain the facts for higher-level members of the parent-child hierarchy.

About Storing Facts for Parent-Child Hierarchies

You can store facts in the fact table for only the leaf members of the parent-child hierarchy, or for members at any level of the parent-child hierarchy, including non-leaf members.

Storing facts for only the leaf members of the parent-child hierarchy

Use this option when facts for the non-leaf members of the parent-child hierarchy can be derived entirely from the facts of the leaf members. For example, if you've a parent-child product hierarchy where the actual product members appear only as leaf members of the hierarchy, then it makes sense for a revenue fact table to only record revenue facts for the leaf members of this product hierarchy. The revenue figures for the non-leaf members of the product hierarchy such as the product categories can be derived entirely by aggregating the facts for the leaf product members at the bottom of the hierarchy.

The image shows example data for a situation where facts are stored only for leaf members in a parent-child hierarchy.

The following table shows example data for the dimension table PRODUCT_DIM:

MemberKey	Name	ParentKey
P1	Product1	C1
P2	Product2	C1
C1	Category1	C2
C2	Category2	C3
C3	Category3	-

The following table shows example data for the fact table REVENUE_FACTS:

ProductKey	YearKey	Revenue
P1	2020	100,000
P1	2021	105,000
P2	2020	75,000
P2	2021	80,000

Store facts for members at any level of the parent-child hierarchy, including non-leaf members

In this option, facts are stored for members at any level of the parent-child hierarchy. This is necessary when the facts for the non-leaf members aren't completely derived from facts of the leaf members. A good example is a sales person hierarchy where a sales person might report to a manager who is also a sales person. Each individual sales person, including the manager, could have a different revenue figure stored in the fact table.

The following table shows example data for the dimension table SALES_REP_DIM:

MemberKey	Name	ParentKey
101	Phillip	201
102	Vivian	201
201	Jacob	301
202	Audrey	301
301	Ryan	-

The following table shows example data for the fact table REVENUE_FACTS:

SalesRepKey	YearKey	Revenue
101	2021	1,200,000
102	2021	1,100,000
201	2021	250,000
202	2021	1,400,000

Storing facts for both leaf and non-leaf members is also appropriate when the rules for aggregating the parent-child hierarchy are complex, or when aggregating the hierarchy at query time is expensive and would lead to unacceptably long query response times. In this case, the fact table would store preaggregated facts for the non-leaf members in addition to the facts stored for the leaf members.

About Aggregating Parent-Child Hierarchies

You must determine how to aggregate the stored facts to calculate the aggregated facts for higher level members of the parent-child hierarchy.

In addition to choosing the correct aggregation function for the measure, you must decide if you need to roll up the fact values recorded for lower-level members to calculate the values for higher-level members. In some cases, rolling up the facts of lower-level members of the parent-child hierarchy makes sense. In other cases such as with a pre-aggregated fact table or a measure that's intended to show each member's individual contribution, rolling up the facts from lower-level members of the parent-child hierarchy is incorrect.

Rolling up facts from lower-level members of a parent-child hierarchy

If a fact table only stores facts for the leaf members of a parent-child hierarchy or if the fact table only records each member's individual contribution, then most likely the

values stored in the fact table must be rolled up to obtain the correct aggregated value for higher-level members of the parent-child hierarchy. Rolling up the facts along a parent-child hierarchy is achieved by joining the fact table to the dimension table through the parent-child relationship table, see Add the Parent-Child Relationship Table to the Semantic Model.

For a fact table that stores facts only for the leaf members such as the product revenue fact table, this modeling technique calculates aggregate values that correctly summarize all the facts for the leaf-level members.

For a fact table that stores the individual contribution of both leaf members and non-leaf members, this technique computes a hierarchical aggregate that summarizes the individual contributions of the member and all its members.

Modeling individual contribution measures

To report the individual contribution of each member, in addition, to reporting the summarized hierarchical aggregate that rolls up the individual contributions of multiple members, you must create two separate fact logical table sources. One fact logical table source maps the base fact table and the parent child relationship table. This is the logical table source for the hierarchical aggregate measure. The second fact logical table source maps only an alias of the fact table. This fact table alias should join directly with the dimension table rather than joining indirectly through the parent-child relationship table. This is the logical table source for the individual contribution measure.

Modeling pre-aggregated measures

Some fact tables contain pre-aggregated data that's populated for all members of the parent-child hierarchy. For example, the fact value for a root member might be populated with the aggregation of the data for all of its descendent members. It's important to ensure that queries don't aggregate the members from this dimension to avoid erroneous results.

To correctly model this type of parent-child hierarchy, you must create a parent-child relationship table to support hierarchical filter functions like `IsAncestor` and `IsDescendant`. You can join the parent-child dimension table directly with the fact table rather than joining through the parent-child relationship table to ensure that the pre-aggregated member value is returned, rather than rolling up all the descendants.

Don't modify the parent-child relationship table script to only include the *self* rows, because doing so would break the `IsAncestor` and `IsDescendant` functions.

To achieve the correct aggregation for dimensions of this type, you must determine what you want to see as a grand total when the parent-child hierarchy is aggregated. For example, assume that your hierarchy contains a single root member, and you want to display the pre-aggregated value for this root member. You must first create an additional fact logical table source mapped at the Total level of the parent-child hierarchy. Next, in the logical table source, create a `WHERE` clause filter that selects only the root member.

With this model in place, for queries that are at the Total level of the parent-child hierarchy, the Oracle Analytics query engine selects the aggregate logical table source and applies the root member `WHERE` clause filter. For queries that are at the Detail level, the Oracle Analytics query engine selects the detailed logical table source and returns the pre-aggregated member values. In either case, it doesn't matter how the aggregation rule is set, because there is a pre-aggregated source at each level.

Use this approach only if the queries are at the Total or Detail level of the parent-child dimension. For queries that group by some non-unique attribute of the parent-child dimension, the aggregation might not be correct. For example, if an Employee dimension has a Location attribute, and a query groups by *Employee.Location*, then double counting is likely

because an employee often reports to other employees at the same location. Because of this, when fact tables contain pre-aggregated member values, you should avoid grouping by non-unique attributes of the parent-child dimension. If grouping by those attributes is unavoidable, then you should model them as separate dimensions.

Maintain Parent-Child Hierarchies Based on Relational Tables

For parent-child hierarchies based on relational tables, you must ensure that the data in the parent-child relationship table accurately reflects the inter-member relationships in the dimension.

If you wrote scripts to create and populate the parent-child relationship table, you must run these scripts, adapting them to guarantee the integrity of the parent-child relationships in the hierarchy. You should add the Populate script to your extract-transform-load (ETL) process so that the script runs after the dimension table is updated. For more information on scripts, see Generate Scripts to Create a Parent-Child Relationship Table.

Model Time Series Data

This topic provides information to help you understand and use functions to model time series data.

Topics:

About Time Series Functions

Time series functions operate on time-oriented dimensions. You use them to compare business performance with previous time periods, allowing you to analyze data that spans multiple time periods.

For example, time series functions enable you to compare current sales to sales from one year ago or one month ago.

Because SQL doesn't provide a direct way to make time comparisons, you must model time series data in the semantic model. First, set up time dimensions based on the period table in your data warehouse. Then, you can define measures that take advantage of this time dimension to use the `AGO`, `TODATE`, and `PERIODROLLING` functions. At query time, the Oracle Analytics query engine generates highly optimized SQL that pushes the time offset processing to the database whenever possible, resulting in the best performance and functionality.

To use time series functions on a particular dimension, you must designate the dimension as a Time dimension and set one or more keys at one or more levels as chronological keys. These keys identify the chronological order of the members within a dimension level.

Use Expression Editor to call a logical function to perform time series calculations instead of aliasing physical tables and modeling logically. The time series functions calculate `AGO`, `TODATE`, and `PERIODROLLING` functions based on the calendar tables in your data warehouse, not on standard SQL date manipulation functions.

This example shows a sample report that includes several measures derived using time series functions.

	2008 Q1			2008 Q2			2008 Q3			2008 Q4		
	2008 / 01	2008 / 02	2008 / 03	2008 / 04	2008 / 05	2008 / 06	2008 / 07	2008 / 08	2008 / 09	2008 / 10	2008 / 11	2008 / 12
Dollars	100	200	300	101	202	303	110	220	330	444	555	666
Dollars Qago				100	200	300	101	202	303	110	220	330
Dollars QTD	100	300	600	101	303	606	110	330	660	444	999	1,665
Dollars 3-Period Rolling Sum	100	300	600	601	603	606	615	633	660	994	1,329	1,665
Dollars 3-Period Rolling Avg	33.3	100.0	200.0	200.3	201.0	202.0	205.0	211.0	220.0	331.3	443.0	555.0

You can use several different grains, such as:

- **Query grain** - The lowest time grain of the request.
- **Time Series grain** - The time series grain indicates the aggregation or offset is requested for the `AGO` and `TODATE` functions. In the above example, the time series grain is Quarter. Time series queries are valid only if the time series grain is at the query grain or higher. The `PERIODROLLING` function doesn't have a time series grain, so instead you specify a start and end period in the function.
- **Storage grain** - You can generate the report shown in the above example from daily sales or monthly sales. The grain of the source is called the storage grain. A chronological key must be defined at this level for the query to work, but performance is generally much better if a define a chronological key at the query grain.

Queries against time series data must exactly match to access the query cache.

About the AGO Function

The AGO function offsets the time dimension to display data from a past period.

This function is useful for comparisons such as *Dollars* compared to *Dollars a Quarter Ago*. The value of *Dollars Qago* for month 2008/08 equals the value of *Dollars* for month 2008/05.

This example shows values for the Dollars and Dollars Qago measures.

	2008 Q1			2008 Q2			2008 Q3			2008 Q4		
	2008 / 01	2008 / 02	2008 / 03	2008 / 04	2008 / 05	2008 / 06	2008 / 07	2008 / 08	2008 / 09	2008 / 10	2008 / 11	2008 / 12
Dollars	100	200	300	101	202	303	110	220	330	444	555	666
Dollars Qago				100	200	300	101	202	303	110	220	330

In the above example, the Dollars Qago measure is derived from the Dollars measure.

In Expression Builder, the `AGO` function has the following template:

```
Ago(<<Measure>>, <<Level>>, <<Number of Periods>>)
```

`<<Measure>>` represents the logical measure column that you want to derive from. In this example, you select the measure "Dollars" from your existing logical fact tables.

`<<Level>>` is the optional time series grain you want to use. In this example, you select "Quarter" from your time dimension.

`<<Number of Periods>>` is the size of the offset, measured in the grain you provided in the `<<Level>>` argument. For example, if the `<<Level>>` is Quarter and the `<<Number of Periods>>` is 2, the function displays dollars from two quarters ago.

Use this function template to create an expression for a One Quarter Ago measure, as follows:

```
Ago("Sales"."Base Measures"."Dollars" , "Sales"."Time MonthDim"."Quarter" , 1)
```

The `<<Level>>` parameter is optional. If you don't want to specify a time series grain in the `AGO` function, the function uses the query grain as the time series grain.

For example, you could define Dollars_Ago as `Ago(Dollars, 1)`. Then, you could perform the following logical query:

```
SELECT Month, Dollars, Dollars_Ago
```

The result is the same as if you defined Dollars_Ago as `Ago(Dollars, Month, 1)`, or you could perform the following logical query:

```
SELECT Quarter, Dollars, Dollars_Ago
```

The result is the same as if you defined Dollars_Ago as `Ago(Dollars, Quarter, 1)`.

See Logical SQL Reference Guide for Oracle Business Intelligence Enterprise Edition.

About the TODATE Function

The TODATE function accumulates the measure from the beginning of the time series grain period to the current displayed query grain period.

This example shows a report with the measure *Dollars QTD*, the Quarter To Date version of the *Dollars* measure.

	2008 Q1			2008 Q2			2008 Q3			2008 Q4		
	2008 / 01	2008 / 02	2008 / 03	2008 / 04	2008 / 05	2008 / 06	2008 / 07	2008 / 08	2008 / 09	2008 / 10	2008 / 11	2008 / 12
Dollars	100	200	300	101	202	303	110	220	330	444	555	666
Dollars QTD	100	300	600	101	303	606	110	330	660	444	999	1,665

In the example, Dollars QTD for Month 2008/05 is the sum of Dollars for 2008/04 and 2008/05. Dollars QTD is the sum of the values for all the query grain periods (month) for the current time series grain period (quarter). The accumulation starts over for the next quarter.

In the example, the Dollars QTD measure is derived from the Dollars measure.

In Expression Builder, the `TODATE` function uses the following format:

```
ToDate(<<Measure>>, <<Level>>)
```

<<*Measure*>> represents the logical measure column that you want to derive from. In this example, you select the measure *Dollars* from your existing logical fact tables.

<<*Level*>> is the time series grain you want to use. In this example, you select *Quarter* from your time dimension.

Using this function format, you can create the following expression for the measure:

```
ToDate("Sales"."Base Measures"."Dollars" , "Sales"."Time MonthDim"."Quarter" )
```

The query grain is specified in the query itself at run time. For example, this measure can display *Quarter To Date* at the *Day* grain, and accumulates up to the end of the Quarter.

See Logical SQL Reference Guide for Oracle Business Intelligence Enterprise Edition.

About the PERIODROLLING Function

The PERIODROLLING function lets you perform an aggregation across a specified set of query grain periods, rather than within a fixed time series grain.

The most common use is to create rolling averages such as a 13-week Rolling Average.

The `PERIODROLLING` function doesn't have a time series grain, the length of the rolling sequence is determined by the query grain. For example, the `Dollars 3-Period Rolling Average` calculates the mean of values from the last 3 months if the query grain is Month, but calculates the mean of the last 3 years if the query grain is Year.

The image shows a report with these two measures.

	2008 Q1			2008 Q2			2008 Q3			2008 Q4		
	2008 / 01	2008 / 02	2008 / 03	2008 / 04	2008 / 05	2008 / 06	2008 / 07	2008 / 08	2008 / 09	2008 / 10	2008 / 11	2008 / 12
Dollars	100	200	300	101	202	303	110	220	330	444	555	666
Dollars 3-Period Rolling Sum	100	300	600	601	603	606	615	633	660	994	1,329	1,665
Dollars 3-Period Rolling Avg	33.3	100.0	200.0	200.3	201.0	202.0	205.0	211.0	220.0	331.3	443.0	555.0

In the example above , the Dollars 3-Period Rolling Sum and Dollars 3-Period Rolling Avg measures are derived from the Dollars measure.

In Expression Editor, the `PERIODROLLING` function has the following format:

```
PeriodRolling(<<Measure>>, <<Starting Period Offset>>, <<Ending Period Offset>>)
```

`<<Measure>>` represents the logical measure column from which you want to derive. To create the measure *Dollars 3-Period Rolling Sum*, you select the measure, *Dollars* from your existing logical fact tables.

`<<Starting Period Offset>>` and `<<Ending Period Offset>>` identify the first period and last period used in the rolling aggregation. The integer is the relative number of periods from the displayed period. In this example, the query grain is month, and the 3-month rolling sum starts 2 periods in the past and includes the current period, that is, for month 2008/07, the rolling sum includes 2008/05, 2008/06 and 2008/07. To create the measure, `Dollars 3-Period Rolling Sum`, the integers to indicate these offsets are -2 and 0.

Using this function format, you can create the following expression for the measure:

```
PeriodRolling("Sales"."Base Measures"."Dollars" , -2, 0)
```

The example also shows a 3-month rolling average. To compute this measure, you can divide the rolling sum that you previously created by 3 to get a 3-period rolling average. The assumption to divide the rolling sum by 3 results from the `<<Starting Period Offset>>` and `<<Ending Period Offset>>` fields for the rolling sum that are -2 and 0.

The expression for the 3-month rolling average is:

```
PeriodRolling("Sales"."Base Measures"."Dollars" , -2, 0) /3
```

Don't use the `AVG` function to create a rolling average. The `AVG` function computes the average of the database rows accessed at the storage grain. To perform the rolling average, you need an average where the denominator is the number of rolling periods at the query grain.

The `PERIODROLLING` function includes a fourth optional hierarchy argument that lets you specify the name of a hierarchy in a time dimension such as `yr, mon, day`, that you want to use to compute the time window. This option is useful when there are multiple hierarchies in a time dimension, or when you want to distinguish between multiple time dimensions.

See Logical SQL Reference Guide for Oracle Business Intelligence Enterprise Edition.

About Creating Logical Time Dimensions

Creating time dimensions requires selecting a Time hierarchy type and designating a chronological key for every level of every dimension hierarchy.

Use these guidelines when modeling time series data:

- Use a time series function when the data source contains history. A data source that contains history might use a star or snowflake schema with an explicit time dimension table. A normalized, historical database might include a time hierarchy with levels in a schema similar to a snowflake. A simple date field isn't adequate for use with a time series function.
- Oracle Analytics Server requires the time dimension physical table or set of normalized tables that are separate from its related physical fact table.

  A common source schema pattern is a fully denormalized table that has time dimension columns are in the same table as facts and other dimensions. This common source schema pattern can't qualify as a time dimension, because the time dimension table is combined with the fact table. Because you can't change the source model, you can create a SELECT statement of the physical table containing the time columns to act as the distinct physical time dimension table. You must join the SELECT statement time dimension to the physical table that contains the facts.
- In the physical layer, the time dimension table or lowest-level table in the normalized/snowflake must join directly to the fact table without any intervening tables.
- The tables in the physical model containing the time dimension can't join to other data sources, except at the most detailed level.
- A member value must be physically present for every period at every hierarchy level. They must not contain rows that are skipped in the sequence. You don't need a fact data for every period. Only the dimension data must be complete.

- You must model each unit of distance between members such as *month*, *half*, or *year*, in a separate hierarchy level.

See Create the Logical Time Dimension.

About Setting Chronological Keys

The chronological keys you set identify the member order within the time dimension level.

The chronological keys must be comparable with the standard SQL `ORDER BY` clause. The `ORDER BY` clause on the chronological key must reflect the real world chronological order of the time dimension members represented by the key. For example, if the time dimension members are: Jan-3-2022, Jan-4-2022, Jan-5-2022 then the following chronological keys can be assigned to them in the same order: 1, 5, 9. However, assigning chronological keys such as 2,1,3 would result in Jan-4-2022, Jan-3-2022, Jan-5-2022, which is an incorrect chronological order.

The Oracle Analytics query engine uses the chronological key to create mathematically correct time series predictions, such as Jan + 2 months = Mar. You should set a chronological key for every level, except for the Grand Total level, so that you can perform time series operations on all levels with good performance. This enables you to use an `AGO`, `TODATE`, or `PERIODROLLING` function for any level of any time dimension hierarchy, such as fiscal month ago, calendar year ago, and day ago.

Theoretically, time series functions operate correctly if only the bottom level key in the logical hierarchy is chronological. In practice, however, this causes performance problems because it forces the physical query to use the lowest grain, causing joins of orders of magnitude more rows, for example, 365 times more rows for a "year ago" joining at the "day" grain.

As with any level key, be sure the key is unique at its level. For example, a column containing simple month names such as "January" isn't unique unless it's concatenated to a column containing year names.

Create the Logical Time Dimension

To enable the time series functions on the dimension, select the **Time** hierarchy type in the logical dimension table and then designate a chronological key for every level of each dimension hierarchy.

See About Creating Logical Time Dimensions and About Setting Chronological Keys.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. In the Logical layer, double-click the logical table where you want to enable time series functions and in the logical table's tabs click **Hierarchy**.
4. Click the **Hierarchy Type** field and select Time.
5. Click a hierarchy level and in its details click **Chronological Key** and choose a key.
6. For each hierarchy level that you need to set a key for, go the level's details, click the **Chronological Key**, and choose a key.
7. Click **Save**.

Create AGO, TODATE, and PERIODROLLING Measures

You can build time series measures by creating derived expressions from base measures.

Follow these guidelines when modeling time series functions:

- You can't derive time series functions from measures that use the fragmentation form of federation. This rule prevents some complex boundary conditions and cross-source assumptions in the query generation and result merging, such as the need to join some time dimension rows from one source to some of the fact rows in a different source. To reduce maintenance and increase accuracy, it's best to create a single base measure, and then derive a family of time series measures from it. For example, start with a base measure, then define variations for month-ago, year-ago, and month-to-date.
- You must define the unit as a level of the time dimension, so that it can take advantage of the chronological keys built in the time dimension.

For information about how to use time series functions in expressions, see About the AGO Function, About the TODATE Function, and About the PERIODROLLING Function.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. In the Logical layer, double-click the logical table containing the logical column that you want to add a time series function to. Click **Columns**.
4. Locate and click the column and click **Detail View**. Scroll to **Sources** and click **Logical Expression**.
5. Click **Open Expression Editor**.
6. In Expression Builder, go to the Function panel and scroll to **Time Series Calculations** and use these functions to build the expression.
7. Click **Save**.

12

Manage Logical Table Sources

This chapter contains information to help you understand how to create and manage logical table sources.

Topics:

What are Logical Table Sources?

Logical table sources define the mappings from a single logical table to one or more physical tables.

Use the physical to logical mapping to specify transformations that occur between the physical layer and the logical layer and to enable aggregate navigation and fragmentation.

In the logical layer, you can open a fact or dimension table and use the **Sources** tab to open a list of its table sources. From that list, you can select a specific table to view its properties such as table mapping, joins, and column mapping.

Logical tables can have many physical table sources. A single logical column might map to many physical columns from multiple physical tables, including aggregate tables that map to the column such as if a query asks for the appropriate level of aggregation on that column.

How Are Fact Logical Table Sources Selected to Answer a Query?

Oracle Analytics uses a specific criteria to select a fact logical table's sources to answer a query.

Every column in a query is sourced from a single logical table source based on the below criteria. Queries aren't load-balanced across multiple logical table sources.

After the fact logical table sources are selected, Oracle Analytics selects the best dimensional logical table sources to answer a query. See How Are Dimension Logical Table Sources Selected to Answer a Query?

The fact logical table source selection criteria is listed from the highest precedence to the lowest precedence:

- **Logical table source priority group** - A higher priority fact logical table source group is used before a lower priority fact logical table source group, even if the higher priority source is at a more detailed grain. A lower group number indicates a higher priority. See About Assigning Logical Table Sources Priority Order.
- **The grain of the logical table source** - If all fact table sources have the same priority number, then a higher-grain logical table source is used before a lower-grain logical table source.
- **Logical table source list order** - If all other criteria are equal, then the first logical table source in the fact table's sources list is selected. This list is displayed in the logical table's **Sources** tab.

How Are Dimension Logical Table Sources Selected to Answer a Query?

After the fact logical table sources are selected, Oracle Analytics selects the best dimensional logical table sources to answer a query.

See How Are Fact Logical Table Sources Selected to Answer a Query?

Oracle Analytics uses the following criteria to select the dimension logical table source. The criteria are listed from the highest precedence to the lowest precedence:

- **Logical table source priority group** - A higher priority dimension logical table source group is used before a lower priority dimension logical table source group. A lower group number indicates higher priority. See About Assigning Logical Table Sources Priority Order.
- **Lower join cost** - If all dimension table sources have the same priority assigned to them, then the dimension logical table source with the lowest join cost is selected before dimension logical tables sources with higher join costs.
- **Higher level** - If the priority group and join cost are the same, then the higher level logical table source is used because that logical table source could require joining fewer rows.
- **Number of elements at this level setting** - If the grains aren't comparable, then the number specified for the **Number of elements at this level** field is considered.

  For example, suppose you've the following two logical table sources with grains that aren't comparable: LTS1(year, city) and LTS2(month, state). If you've 10 years, 100 cities, 120 months, and 9 states, the worst case size of LTS1 is 10 x 100 = 1000, and the worst case size of LTS2 is 120 x 9 = 1080. In this scenario, LTS1 is selected because the source with the lowest estimated number of total elements is assumed to be the fastest.

Change the Default Selection Criteria for Dimension Logical Table Sources

You can change the default logical table source selection criteria to favor dimension logical table sources that are at the same level as the fact logical table source before considering the higher level logical table source.

Create a session variable and name it `DIMENSION_LTS_JOIN_RESTRICTIONS`. Set this session variable to `PREFER_SAME_LEVEL`.

If a suitable dimension logical table source at the same level as the fact logical table source doesn't exists, then the Oracle Analytics query engine selects the highest level dimension logical table source that's joinable to the fact. These factors are only considered after priority group and join cost.

The `PREFER_SAME_LEVEL` value for the `DIMENSION_LTS_JOIN_RESTRICTIONS` session variable sets the following criteria for selecting the dimension logical table source to answer the query:

- Logical table source priority group
- Lower join cost
- Same level as the fact logical table source
- Higher level than other dimension logical table sources if no other logical table source is at the same level as the fact logical table source

When `DIMENSION_LTS_JOIN_RESTRICTIONS` is set to *NONE*, the default value, you can join fact logical table sources to a higher level dimension logical table source even if there is another joinable dimension logical table source at the same level as the fact.

About Consistency Among Data in Multiple Table Sources

It's important to confirm that your table sources' data is consistent.

For example, the year-level logical table source and the month-level logical table source for your time dimension should cover the same time period.

You might see consistency issues in table source data for queries that override null suppression. For example, some aggregate tables might not include the dimension records that correspond to the null fact values such as a yearly sales aggregate table that doesn't include years with no sales. All years in the year dimension must exist for the null values to be included in the result.

Add Logical Table Sources

A logical table's physical source is included when you drag and drop a table from the physical layer to the logical layer. You can add logical table sources to the logical tables that you create by dragging and dropping, or to logical tables that you create manually.

Add logical table sources when you need multiple physical tables to source the logical table's data. For example:

- You have three different business units each with its own order system where you get revenue information.

- You periodically summarize revenue from an orders system or a financial system and use this table for high-level reporting.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. In the Logical Layer pane, browse for and double-click the table where you want to add a table source.
5. In the logical table, click the **Sources** tab.
6. Optional: To browse for and add a physical table source, click **Add Physical Table**, select **Add Physical Table**, and browse for and select a physical table.
7. Optional: To add a new source, click **Add Physical Table**, select **Create New Source**, and enter a name for the source table.
8. From the table's source list, click a table source and then click **Detail view**.
9. Specify the data source's general properties, table mapping, joins, column mapping, and data granularity.
10. Click **Save**.

Enable or Disable a Logical Table Source

You can enable or disable one or more of a logical table's source.

You can use this setting to test your queries. Any table that is disabled isn't considered during query generation.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. In the Logical Layer pane, browse for and double-click the table with the source that you want to enable or disable.
5. In the logical table, click the **Sources** tab.
6. In the sources table, click the source table that you want to enable or disable and then click **Detail view** to open the properties pane.
7. Click the **Enabled** field to select (enable) the table source, or click the field to clear (disable) the table source.
8. Click **Save**.

Work With Logical Table Source Priorities

This topic provides information to help you understand and assign logical table source priorities.

Topics:

- Reverse the Table Source Priority Ranking at Query Time

About Assigning Logical Table Sources Priority Order

Priority numbers determine which logical table source is used to answer a query.

For example, you might have user queries that are fulfilled by both a data warehouse and an OLTP source. Often access to an operational system is expensive, while access to a data warehouse is cheap. In this situation, you can assign a higher priority to the data warehouse to ensure that all queries are fulfilled by the data warehouse if possible.

Although the logical table source priority is the metric that the Oracle Analytics query engine considers before any other cost metric, the table source's priority group doesn't always determine that a particular query is fulfilled by that source. The Oracle Analytics query engine uses other factors to determine which logical table source to use for a query. See How Are Fact Logical Table Sources Selected to Answer a Query? and How Are Dimension Logical Table Sources Selected to Answer a Query?

To assign priority group numbers, you rank your logical table sources in numeric order, with 0 being the highest-priority source. You can assign the same number to multiple sources to create a priority group. For example, you can have two logical table sources in priority group 0, two logical table sources in priority group 1, and so on. In most cases only two priority groups (0 and 1) are needed.

Assigning priority groups is optional.

It's important that you don't use priority groups as a method of fine tuning the choice of logical table sources used to answer queries. The Oracle Analytics query engine tries to automatically use the most optimal logical table sources, but only within the same priority group. When you set a different priority group to each logical table source, it might cause the Oracle Analytics query engine to use suboptimal logical table sources.

Set the Logical Table Sources Priority Order

Sometimes a logical table contains more than one table source that can be used in a query. In such cases, you can set priority numbers to determine which logical table source is used in a query.

You can assign the same priority number to more than one source table to create a priority group.

See About Assigning Logical Table Sources Priority Order.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. In the Logical Layer pane, browse for and double-click the table with the sources that you want to assign priority groups to.
5. In the logical table, click the **Sources** tab.
6. In the sources list table, click the source table that you want to assign query groups to and then click **Detail view** to open the properties pane.
7. Scroll to General and click the **Priority** field and enter a priority group number.
8. In the sources list table, click another source table that you want to assign query groups to and then click **Detail view** to open the properties pane.

9. Scroll to General and then click the **Priority** field and enter a priority group number.
10. Click **Save**.

Reverse the Table Source Priority Ranking at Query Time

You can use session variables and request variables with logical table source priority groups to reverse the logical table source priorities at query time. This method provides a way to dynamically select a source at run time, depending on user preference.

1. To enable the dynamic selection, first create the `REVERSIBLE_LTS_PRIORITY_SA_VEC` session variable in the semantic model. Create this variable as a string vector session variable that uses a row-wise session initialization block. `REVERSIBLE_LTS_PRIORITY_SA_VEC` should list the subject areas for which you want to allow users to reverse the logical table source priority ranking. You must define this variable to enable priority ranking reversal.
2. After you've defined the set of subject areas where you want to allow priority ranking reversal, users can include the request variable `REVERSE_LTS_PRIORITY` with their queries to reverse the logical table source priority ranking. You can set this request variable to 1 to reverse the logical table source priority, or 0 to keep the normal logical table source priority.
3. As an alternative to using a request variable at query time, you can define a predetermined set of subject areas for which the logical table source priority is permanently reversed. To do this, create the session variable `REVERSED_LTS_PRIORITY_SA_VEC`. Create this variable as a string vector session variable that uses a row-wise session initialization block. `REVERSED_LTS_PRIORITY_SA_VEC` should list the subject areas where you want the logical table source priority set to permanently reversed.
4. You could create a table called SA_TABLE that contains two columns: SUBJECT_AREA_NAME and REVERSIBLE. This table could contain rows mapping subject area names to their reversible values (1 or 0), as follows:
   - SUBJECT_AREA_NAME - my_sa_1; REVERSIBLE - 1
   - SUBJECT_AREA_NAME - my_sa_2; REVERSIBLE - 0
5. Then, create a string vector session variable called REVERSIBLE_LTS_PRIORITY_SA_VEC with a row-wise session initialization block. The initialization string for this initialization block is similar to the following:
   ```
   SELECT 'REVERSIBLE_LTS_PRIORITY_SA_VEC', SUBJECT_AREA_NAME FROM
   SA_TABLE WHERE REVERSIBLE=1
   ```

Modify a Logical Table Source's Logical Column to Physical Column Mappings

Semantic Modeler automatically maps logical columns to physical columns when you drag and drop a table from the physical layer to the logical layer, or when you add

additional logical table sources to a logical table. You can modify these default column mappings.

You must include a physical table source to make its column available for use in column mapping. See Add Logical Table Sources.

Logical to physical column mapping can also be used to specify transformations that occur between the physical layer and the logical layer. The transformations can be simple, such as changing an integer data type to a character, or more complex, such as applying a formula to find a percentage of sales per unit of population. Applying these transformations is typically referred to as creating calculated items.

The data type of a logical column is determined by its logical table source mappings. For example, if a logical column has one physical source with a data type of `VARCHAR(50)` *not-nullable*, and another physical source with a `VARCHAR(20)` data type, *nullable*, then the data type of the logical column is `VARCHAR(50)` *nullable*. This final type is called a promoted type. Because of the rules governing logical table source mappings, you can't map physical sources with data types that are promotable such as an `INT` with a `VARCHAR`.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. In the Logical Layer pane, browse for and double-click the table with the column mappings you want to modify.
5. In the logical table, click the **Sources** tab.
6. In the logical table sources list table, click a logical table source to select it and then click **Detail view** to open the properties pane.
7. Scroll to Column Mapping, locate and double-click the column mapping that you want to change.
8. Click the physical column field's dropdown button and browse for and select the name of the physical column to map the corresponding logical column to.
9. Click **Save**.

Map a Logical Table Source's Logical Column to a Calculated Item

Create a calculated item when you need to derive the logical column's data from two or more physical columns or tables.

You can create calculated items where formulas are applied pre-aggregation. These are two examples:

- Create the measure *tons sold* using the columns *units_sold* and *unit_weight*, you apply a pre-aggregation formula (`fact.units_sold*product.unit_weight`), and then apply the aggregation rule `SUM` in the measure object.
- Use `CAST` to transform a column of type `TIMESTAMP` to type `DATE` for faster display in Answers and other clients, for example, `CAST("DB"."TABLE"."COL" AS DATE)`.

You can also change data sources by creating expressions that perform transformations on physical data. For example, you can use the `CAST` function to transform a column with a character data type to an integer data type to match data coming from a second logical table

source. Other examples include using `CONCATENATE` or math functions to make similar transformations on physical data.

You must include a physical table source to make its columns available for use in a calculation. See Add Logical Table Sources.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. In the Logical Layer pane, browse for and double-click the table with the column mappings you want to modify.
5. In the logical table, click the **Sources** tab.
6. In the logical table sources list table, click a logical table source to select it and then click **Detail view** to open the properties pane.
7. Go to the pane's Column Mapping section, locate and double-click the column mapping that you want to create a calculated item for.
8. Click **Open Expression Editor** and create and validate the physical item's calculation. In the Expression Editor click **Save**.
9. Click **Save** to save the semantic model.

Work With Data Granularity

This topic provides information to help you understand and define logical table sources' data granularity.

Topics:

About Data Granularity

Data granularity indicates a logical table source's level of detail. When a query is issued, the Oracle Analytics query engine uses the logical table source's data granularity to find the required level of detail for the requested data.

The logical table's logical dimensions and hierarchies determine the granularity levels that you can assign to a logical table source. For example, years, months, weeks, days, or hours.

You need to specify data granularity for each fact table's logical table sources. This granularity defines at what level of granularity the data is stored in the fact table. You also need to define granularity for each logical table that joins to the fact table. The Oracle Analytics query engine assumes that if no logical table sources level is specified, then the most detailed level should be used. A data modeling best practice is to assign data granularity for each table source.

About Aggregate Tables

Aggregate tables are physical tables that store precomputed results from measures that have been aggregated over a set of dimensional attributes.

You must join the aggregate fact and dimension tables. See About Aggregate Table Joins.

Each aggregate table column contains data at a given set of levels. For example, a monthly sales table might contain a precomputed sum of the revenue for each product in each store during each month.

When you create a logical table source for an aggregate fact table, you should create corresponding logical dimension table sources at the same levels of aggregation.

You need to have at least one logical dimension table source for each level of aggregation. If the sources at each level already exist, you don't need to create additional sources.

For example, you might have a monthly sales fact table containing a precomputed sum of the revenue for each product in each store during each month. You need to have the following three dimension sources, one for each of the logical dimension tables referenced in the example:

- A source for the Product logical table with one of the following content specifications:
  - By logical level: ProductDimension.ProductLevel
  - By column: Product.Product_Name
- A source for the Store logical table with one of the following content specifications:
  - By logical level: StoreDimension.StoreLevel
  - By column: Store.Store_Name
- A source for the Time logical table with one of the following content specifications:
  - By logical level: TimeDimension.MonthLevel
  - By column: Time.Month

At query time, the Oracle Analytics query engine first determines which sources have enough detail to answer the query. Out of these sources, the Oracle Analytics query engine chooses the most aggregated source to answer the query, because it's assumed to be the fastest. The most aggregated source is the one with the lowest multiplied number of elements.

See Create Logical Levels in a Logical Dimension Table to learn how to specify the number of elements at each level.

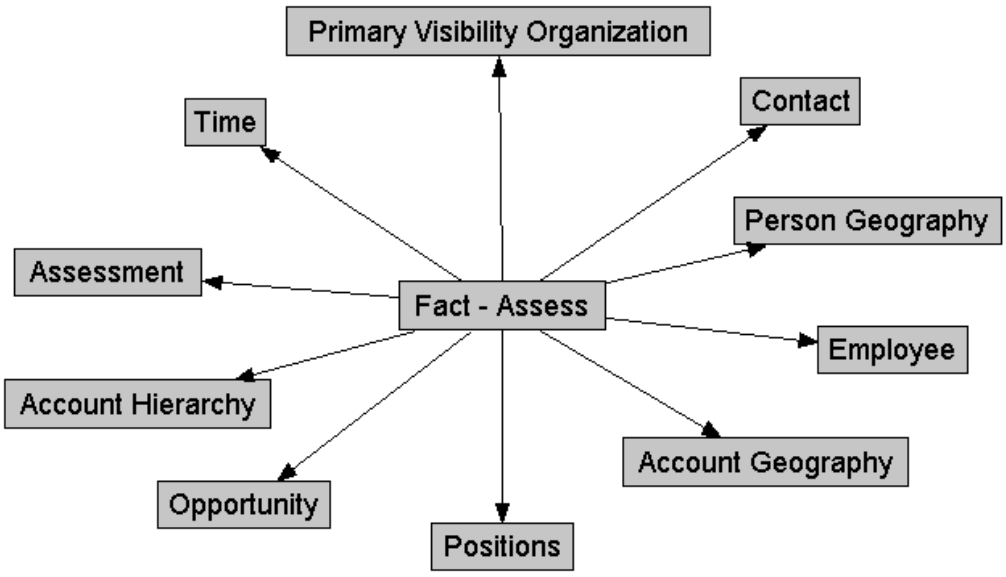
About Aggregate Table Joins

You must create physical joins between the aggregate fact tables and the aggregate dimension tables.

Joins tells the Oracle Analytics query engine where to send queries for physical aggregate fact tables joined to and constrained by values in the physical aggregate dimension tables.

You can verify joins by opening the fact logical table's logical diagram. The diagram displays only the dimension logical tables that are directly joined to the fact logical table. The diagram doesn't display dimension tables if the same physical table is used in logical fact and dimension sources.

The image shows the Fact - Assess fact table's logical diagram.



The table contains a list of the logical level for each dimension table that's directly joined to the Fact - Assess fact table.

Dimension	Logical Level
Account Geography	Postal Code Detail
Person Geography	Postal Code Detail
Time	Day Detail
Account Organization	Account Detail
Opportunity	Opty Detail
Primary Visibility Organization	Detail
Employee	Detail
Assessment	Detail
Contact (W_PERSON_D)	Detail
FINS Time	Day
Positions	Details

About the Logical Table Source's Parent-Child Settings

When a logical table is part of a dimension with a parent-child hierarchy that's based on relational tables, the logical table includes both a physical source and a source for the parent-child relationship table required by the parent-child hierarchy.

Parent-child relationship tables explicitly define the inter-member relationships for parent-child hierarchies.

You can view details for the parent-child relationship table source in logical table's Hierarchy tab.

- **Relationship Table** - The name of the parent-child relationship table that the source is based on.
- **Member Key** - The name of the column in the parent-child relationship table that identifies the member.
- **Parent Key** - The name of the column in the parent-child relationship table that identifies an ancestor of the member.
- **Relationship Distance** - The name of the column in the parent-child relationship table that specifies the number of parent-child hierarchical levels from the member to the ancestor.
- **Leaf Node Identifier** - The name of the column in the parent-child relationship table that indicates if the member is a leaf member (1=Yes, 0=No).

See Create Dimensions with Parent-Child Hierarchies.

Define Logical Table Source Data Granularity

Define granularity for the dimension tables and dimension and level information for the fact table joined to the dimension tables. The Oracle Analytics query engine uses this information to find the required level of detail for the requested data.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. In the Logical Layer pane, browse for and double-click the dimension table with the table source you want to define data granularity for.
5. In the table's tabs, click **Sources**.
6. In the table sources list, click the logical table source that you want to define data granularity for and then click **Detail view** to open the properties pane.
7. Scroll to Data Granularity, click the **Defined by** field, and choose a level. Repeat this step for other dimension tables as needed.
8. In the Logical Layer pane, browse for and double-click the fact table joined to the dimension tables that you defined data granularity for.
9. In the table's tabs, click the **Sources** tab.
10. In the logical table sources list table, click the logical table source that you want to define data granularity for and then click **Detail view** to open the properties pane.
11. Scroll to the pane's Data Granularity section and click **Add Level**.
12. In the new level's **Dimension** field, click the dropdown button and choose a dimension table. In the **Level** field, click the dropdown button and choose a granularity level. Repeat this step for other dimension tables joined to the fact table.
13. Click **Save**.

Work With Logical Table Source Data Fragmentation

This topic provides information to help you understand and define data fragmentation.

Topics:

About Data Fragmentation

A logical table can include table sources that have the same level of detail, but each contains a specific range of values (or fragments of data). These tables are called fragmented tables.

When you use fragmented tables as logical table sources, you must write an expressions for each table source to indicate its range of values. The Oracle Analytics query engine uses the expression to determine which table to use to find the data requested by the query.

Fragmented logical table sources must have physical joins to the appropriate tables so that when the Oracle Analytics query engine uses the fragment, it still joins to the appropriate table sources.

Sometimes the data needed for a query overlaps between fragmented logical table sources. In these cases you might need to select the **This source should be combined with other sources at this level** option. Consider the following examples of how to use this option:

- **Example 1** - Suppose your logical table uses a fragmented logical table source containing all sales for years 2000 to the current year (2022), and another fragmented logical table source containing current year sales and its table source fragmentation expression is set to year = 2022. In this case the table fragments overlap and you shouldn't select the **This source should be combined with other sources at this level** option. In this case, the Oracle Analytics query engine can use any single fragment based on query predicate or fragmentation predicate compatibility.
- **Example 2** - Suppose your logical table uses a fragmented logical table source that contains all sales for years 2000 to 2021, and another fragmented logical table source containing sales for year 2022. In this case you should select the **This source should be combined with other sources at this level** option because the fragments don't overlap. In this case, the Oracle Analytics query engine creates a union of all the logical table sources on this level that can't be disqualified based on query predicate or fragmentation predicate compatibility.

If a logical table is sourced from a set of fragmented tables, then each fragmented table doesn't have to map the same set of columns. However, the Oracle Analytics query engine returns different answers depending on how the columns are mapped.

For the best query results, Oracle recommends that all the fragments map to the same set of columns.

- If the logical table is sourced from fragmented tables that map the same set of columns, then the Oracle Analytics query engine considers the set of fragmented sources to be a complete set of logical table sources. This means that measure aggregations can be calculated based on the set of fragments.
- If the set of mapped columns differs between the fragmented tables, then the Oracle Analytics query engine assumes that the set of logical table sources is incomplete, and because some fragments are missing, won't calculate aggregate rollups. In this case, the server returns NULL as measure aggregates.

About Global Variables and Logical Table Source Fragmentation

You can use global variables in a logical table source's fragmentation expression to automatically modify a fragment's content.

For example, suppose you have two sources for information about orders where one source contains recent orders and the other source contains historical data. You need to update the global variable to use the recent orders and move the historical order data to a different view. Without using global variables, you would describe the content of the source containing recent data with an expression such as:

```
Orders.OrderDates."Order Date" >= TIMESTAMP '2001-06-02 00:00:00'
```

This content statement becomes invalid as new data is added to the recent source and older data is moved to the historical source. To accurately reflect the new content of the recent source, you would have to modify the fragmentation content description manually. Instead you can define global variables to automatically modify the content.

Define Data Fragmentation for a Logical Table Source

A fragmented table contains a portion of the data at a specific aggregation level. When you use fragmented tables, you must write an expressions for each table source to indicate its range of values.

See About Data Fragmentation.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. In the Logical Layer pane, browse for and double-click the table with the table source you want to define data fragmentation for.
5. In the table's tabs, click **Sources**.
6. In the logical table sources list table, click the logical table source that you want to define data fragmentation for and then click **Detail view** to open the properties pane.
7. Go to the pane's Data Fragmentation section and click **Data is fragmented**.
8. Click **Open Expression Editor** and create and validate the fragment expression. In the Expression Editor click **Save**.
9. Optional: If the data needed for a query is located in more than one fragmented table, then click **Combine with other fragmented sources** to sum the data.

10. Optional: Click **Enable Data Driven Fragment Selection** to improve the performance of the logical table source.
11. Click **Save**.

Improve the Performance of Fragmented Logical Table Sources

You can use data driven fragment selection to improve the performance of fragmented logical table sources.

Data driven fragment selection is disabled by default.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. In the Logical Layer pane, browse for and double-click the table with the fragmented source that you want to improve performance for.
5. In the table's tabs, click **Sources**.
6. In the table sources list, click the logical table source that you want to improve performance for then click **Detail view** to open the properties pane.
7. Scroll to Data Granularity and click **Data is fragmented** to display the expression that you created.
8. Click **Enable Data Driven Fragment Selection**.
9. Click **Save**.

Work With Fragmentation for Aggregate Navigation

This topic contains examples that provide techniques and rules for specifying data fragmentation.

See Define Data Fragmentation for a Logical Table Source.

Topics:

- Specify Fragmentation for Single Column, Value-Based Predicates
- Specify Fragmentation for Single Column, Range-Based Predicates

Specify Fragmentation for Single Column, Value-Based Predicates

You can replace the `IN` predicates with either an equality predicate or multiple equality predicates separated by the `OR` connective.

Fragment 1:

```
logicalColumn IN <valueList1>
```

Fragment n:

```
logicalColumn IN <valueListN>
```

Specify Fragmentation for Single Column, Range-Based Predicates

Use >= and < predicates to ensure that the fragment content descriptions don't overlap. For each fragment, you must express the upper value as <. An error occurs if you use <=. You can't use the `BETWEEN` predicate to describe fragment range content.

Fragment 1:

```
logicalColumn >= valueof(START_VALUE) AND logicalColumn < valueof(MID_VALUE1)
```

Fragment 2:

```
logicalColumn >= valueof(MID_VALUE1) AND logicalColumn < valueof(MID_VALUE2)
```

Fragment n:

```
logicalColumn >= valueof(MID_VALUEN-1) AND logicalColumn < valueof(END_VALUE)
```

Pick your start point, midpoints, and endpoint carefully.

The `valueof` referenced here is the value of a semantic model variable. If you use semantic model values in your expression, the following construct doesn't work for Fragment 2:

```
logicalColumn >= valueof(MID_VALUE1)+1 AND logicalColumn < valueof(MID_VALUE2)
```

Use another semantic model variable instead of `valueof(MID_VALUE1)+1`.

The same variables, for example, `valueof(MID_VALUE1)`, aren't required to appear in the content of both fragments. You could set another variable, and create statements of the following form:

Fragment 1:

```
logicalColumn >= valueof(START_VALUE) AND logicalColumn < valueof(MID_VALUE1)
```

Fragment 2:

```
logicalColumn >= valueof(MID_VALUE2) AND logicalColumn < valueof(MID_VALUE3)
```

Specify Multicolumn Content Descriptions

An arbitrary number of predicates on different columns can be included in each content filter. Each column predicate can be value-based or range-based.

Fragment 1:

```
<logicalColumn1 predicate> AND <logicalColumn2 predicate > ... AND <logicalColumnM
predicate>
```

Fragment n:

```
<logicalColumn1 predicate> AND <logicalColumn2 predicate > ... AND <logicalColumnM
predicate>
```

Ideally, all fragments have predicates on the same M columns. If there is no predicate constraint on a logical column, The Oracle Analytics query engine assumes that the fragment contains data for all values in that logical column.

Specify Parallel Content Descriptions

Use the parallel OR to handle dates that cross logical columns such as across years, or across months in a date range.

Use the `parallel OR technique` to handle the multiple hierarchical relationships across logical columns such as from year to year month to date, and from month to year month to date. For example, consider fragments delineated by different points in time such as year and month. Constraining sufficiently far back in a year is enough to drive the selection of just the historical fragment. The `parallel OR technique` supports this.

This example assumes that the snapshot month was April 1, 12:00 a.m. in the year 2022.

Fragment 1 (Historical):

```
EnterpriseModel.Period."Day" < VALUEOF("Snapshot Date") OR
EnterpriseModel.Period.MonthCode < VALUEOF("Snapshot Year Month") OR
EnterpriseModel.Period."Year" < VALUEOF("Snapshot Year") OR
EnterpriseModel.Period."Year" = VALUEOF("Snapshot Year") AND
  EnterpriseModel.Period."Month in Year" < VALUEOF("Snapshot Month") OR
EnterpriseModel.Period."Year" = VALUEOF("Snapshot Year") AND
  EnterpriseModel.Period."Monthname" IN ('Mar', 'Feb', 'Jan')
```

Fragment 2 (Current):

```
EnterpriseModel.Period."Day" >= VALUEOF("Snapshot Date") OR
EnterpriseModel.Period.MonthCode >= VALUEOF("Snapshot Year Month") OR
EnterpriseModel.Period."Year" > VALUEOF("Snapshot Year") OR
EnterpriseModel.Period."Year" = VALUEOF("Snapshot Year") AND
  EnterpriseModel.Period."Month in Year" >= VALUEOF("Snapshot Month") OR
EnterpriseModel.Period."Year" = VALUEOF("Snapshot Year") AND
  EnterpriseModel.Period."Monthname" IN ('Dec', 'Nov', 'Oct', 'Sep', 'Aug',
'Jul',
  'Jun', '', 'Apr')
```

If the logical model doesn't go down to the date level of detail, then omit the predicate on `EnterpriseModel.Period."Day"` in the preceding example.

Notice the use of the `OR` connective to support parallel content description tracks.

Specify Unbalanced Parallel Content Descriptions

In an order entry application, time-based fragmentation between historical and current fragments is insufficient.

For example, records might still be volatile, even though they're historical records entered into the database before the snapshot date.

For the following example, assume that open orders can be directly updated by the application until the order is shipped or canceled. After the order has shipped, however, the only change that can be made to the order is to type a separate compensating return order transaction.

There are two parallel tracks in the following content descriptions. The first track uses the multicolumn, parallel track techniques described in the preceding section. Notice

the parentheses nesting the parallel calendar descriptions within the Shipped-or-Canceled order status multicolumn content description.

The second parallel track is present only in the Current fragment and specifies that all Open records are in the Current fragment only.

Fragment 1 (Historical):

```
Marketing."Order Status"."Order Status" IN ('Shipped', 'Canceled') AND
  Marketing.Calendar."Calendar Date" <= VALUEOF("Snapshot Date") OR
Marketing.Calendar."Year" <= VALUEOF("Snapshot Year") OR
Marketing.Calendar."Year Month" <= VALUEOF("Snapshot Year Month")
```

Fragment 2 (Current):

```
Marketing."Order Status"."Order Status" IN ('Shipped', 'Canceled') AND
  Marketing.Calendar."Calendar Date" > VALUEOF("Snapshot Date") OR
Marketing.Calendar."Year" >= VALUEOF("Snapshot Year") OR
Marketing.Calendar."Year Month" >= VALUEOF("Snapshot Year Month") OR
Marketing."Order Status"."Order Status" = 'Open'
```

The overlapping Year and Month descriptions in the two fragments don't cause a problem because overlap is permissible when there are parallel tracks. The rule is that at least one of the tracks has to be non-overlapping. The other tracks can have overlap.

Examples of Parallel Content Descriptions

These examples explain how to use labels with fragment content statements.

The Track number labels in the examples are shown to help relate the examples to the discussion that follows. You wouldn't include these labels in the actual fragmentation content statement.

Fragment 1 (Historical)

```
Track 1  EnterpriseModel.Period."Day" < VALUEOF("Snapshot Date") OR
Track 2  EnterpriseModel.Period.MonthCode < VALUEOF("Snapshot Year Month") OR
Track 3  EnterpriseModel.Period."Year" < VALUEOF("Snapshot Year") OR
Track 4  EnterpriseModel.Period."Year" = VALUEOF("Snapshot Year") AND
  EnterpriseModel.Period."Month in Year" < VALUEOF("Snapshot Month") OR
Track 5  EnterpriseModel.Period."Year" = VALUEOF("Snapshot Year") AND
  EnterpriseModel.Period."Monthname" IN ('Mar', 'Feb', 'Jan')
```

For example, consider the first track on `EnterpriseModel.Period."Day."` In the historical fragment, the < predicate tells the Oracle Analytics query engine that any queries that constrain on Day before the Snapshot Date fall within the historical fragment. Conversely, the >= predicate in the current fragment on Day indicates that the current fragment doesn't contain data before the Snapshot Date.

The second track on `MonthCode`, for example, 202112, is similar to Day. It uses the < and >= predicates, as there is a non-overlapping delineation on month because the snapshot date is April 1. The key rule to remember is that each additional parallel track must reference a different column set. You can use common columns, but the overall column set must be unique. The Oracle Analytics query engine uses the column set to select the most appropriate track.

The third track on `Year`, < in the historical fragment and > in the current fragment, tells the Oracle Analytics query engine that optimal (single) fragment selections can be made on queries that just constrain on year. For example, a logical query on Year IN (2019, 2020)

should only hit the historical fragment. Likewise, a query on Year = 2022 should only hit the current fragment. However, a query that hits the year 2021 can't be answered by the content described in this track, and therefore hits both fragments, unless additional information can be found in subsequent tracks.

The fourth track describes the fragment set for Year and Month in Year (month integer). Notice the use of the multi-column content description technique, described previously. Notice the use of < and >= predicates, as there is no ambiguity or overlap for these two columns.

The fifth track describes fragment content in terms of Year and Month name. It uses the value-based `IN` predicate technique.

As an embellishment, suppose the snapshot date fell on a specific day within a month: therefore, multi-column content descriptions on just year and month would overlap on the specific snapshot month. To specify this ambiguity, <= and >= predicates are used.

Fragment 1 (Historical):

```
EnterpriseModel.Period."Day" < VALUEOF("Snapshot Date") OR
EnterpriseModel.Period.MonthCode <= VALUEOF("Snapshot Year Month") OR
EnterpriseModel.Period."Year" < VALUEOF("Snapshot Year") OR
EnterpriseModel.Period."Year" = VALUEOF("Snapshot Year") AND
  EnterpriseModel.Period."Month in Year" <= VALUEOF("Snapshot Month") OR
EnterpriseModel.Period."Year" = VALUEOF("Snapshot Year") AND
  EnterpriseModel.Period."Monthname" IN ('Apr', 'Mar', 'Feb', 'Jan')
```

Fragment 2 (Current):

```
EnterpriseModel.Period."Day" >= VALUEOF("Snapshot Date") OR
EnterpriseModel.Period.MonthCode >= VALUEOF("Snapshot Year Month") OR
EnterpriseModel.Period."Year" > VALUEOF("Snapshot Year") OR
EnterpriseModel.Period."Year" = VALUEOF("Snapshot Year") AND
  EnterpriseModel.Period."Month in Year" >= VALUEOF("Snapshot Month") OR
EnterpriseModel.Period."Year" = VALUEOF("Snapshot Year") AND
  EnterpriseModel.Period."Monthname" IN ('Dec', 'Nov', 'Oct', 'Sep', 'Aug',
'Jul',
  'Jun', '', 'Apr')
```

Work With Aggregate Table Fragments

This topic contains a use case that provides techniques and rules for working with aggregate table fragments.

See Define Data Fragmentation for a Logical Table Source.

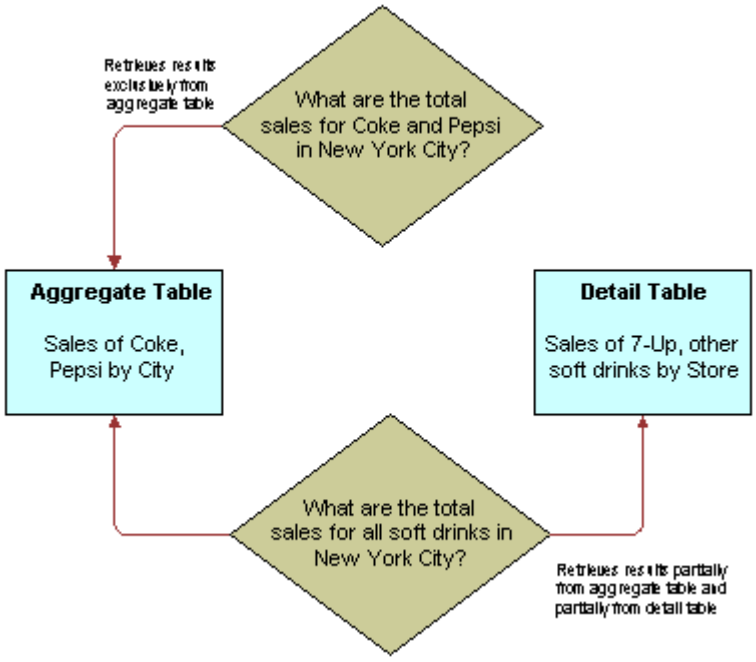
Topics:

- About Aggregate Table Fragments
- Specify the Aggregate Table Content
- Define a Physical Layer Table with a Select Statement to Complete the Domain
- Specify the SQL Virtual Table Content
- Create Physical Joins for the Virtual Table

About Aggregate Table Fragments

Data at an aggregation level can be stored in multiple physical tables. In such cases, you need to specify which logical table source contains which fragment of the data so that the Oracle Analytics query engine chooses the correct source for the query.

For example, suppose you have a database that tracks the sales of soft drinks in all stores. The detail level of data is at the store level. Aggregate information is stored at the city level for the sales of Coke and Pepsi, but there is no aggregate information for the sales of 7-Up or other sodas.



The goal of this type of configuration is to maximize the use of the aggregate table. If a query asks for sales figures for Coke and Pepsi, the data should be returned from the aggregate table. If a query asks for sales figures for all soft drinks, the aggregate table should be used for Coke and Pepsi and the detail data for the other brands.

The Oracle Analytics query engine handles this type of partial aggregate navigation. To configure a semantic model to use aggregate fragments for queries whose domain spans multiple fragments, you need to define the entire domain for each level of aggregate data, even if you must configure an aggregate fragment as being based on a less summarized physical source.

Specify the Aggregate Table Content

You configure the aggregate table navigation in the logical table source's fragmentation expression.

In the soft drink example, the aggregate table contains data for Coke and Pepsi sales at the city level.

Its data fragmentation expression should be similar to the following:

```
SoftDrinks.Products.Product IN ('Coke', 'Pepsi')
```

This expression tells the Oracle Analytics query engine that the source table has data at the city and product level for two of the products.

Because this source is a fragment of the data at this level, you must select the **Combine with other fragmented sources** field to indicate that the source combines with other sources at the same level.

Define a Physical Layer Table with a Select Statement to Complete the Domain

The data for the rest of the domain (the other types of sodas) is all stored at the store level.

To define the entire domain at the aggregate level, for example city and product, you need to have a source that contains the rest of the domain at this level. Because the data at the store level is at a lower, more detailed level than at the city level, it's possible to calculate the city and product level detail from the store and product detail by adding up the product sales data of all of the stores in a city. You can use a query involving the store and product level table.

One way to do this is to define a table in the physical layer with a Select statement that returns the store level calculations. To define the table, go to the physical layer on the physical schema object and create a table on the physical schema object that the `SELECT` statement uses for the query. Choose **Select** from the **Table Type** list, and type the SQL statement in the **Default Initialization String** box.

The SQL statement must define a virtual table that completes the domain at the level of the other aggregate tables. In this case, there is one existing aggregate table, and it contains data for Coke and Pepsi by city. Therefore, the SQL statement has to return all of the data at the city level, except for the Coke and Pepsi data.

Specify the SQL Virtual Table Content

Create a logical table source for the Sales column that covers the remainder of the domain at the city and product level.

This source contains the virtual table created in the previous section. Map the Dollars logical column to the US Dollars physical column in this virtual table.

The aggregate content specification for this source is:

Group by logical level:

```
GeographyDim.CityLevel, ProductDim.ProductLevel
```

This tells the Oracle Analytics query engine that this source has data at the city and product level.

The fragmentation content specification might be:

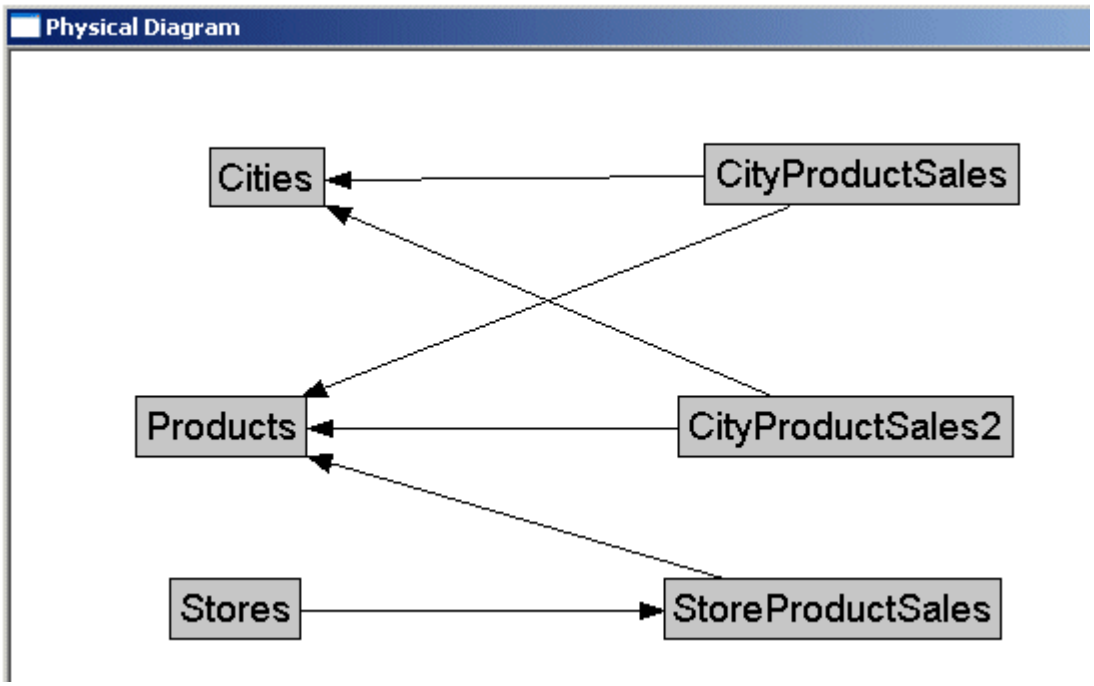
```
SoftDrinks.Products.Product = '7-Up'
```

Additionally, because it combines with the aggregate table containing the Coke and Pepsi data at the city and product level to complete the domain, you need to select the **Combine with other fragmented sources** field.

Create Physical Joins for the Virtual Table

This topic provides an example that shows you how to construct physical joins for the virtual table.

Construct the correct physical joins for the virtual table. Notice that CityProductSales2 joins to the Cities and Products tables.



In this example, the two sources comprise the whole domain for soda sales. A domain can have many sources. The sources have to all follow the rule that each level must contain sources that, when combined, comprise the whole domain of values at that level. Setting up the entire domain for each level helps ensure that queries asking for Coke, Pepsi, and 7-Up don't leave out 7-Up. It also helps ensure that queries requesting information that has been precomputed and stored in aggregate tables can retrieve that information from the aggregate tables, even if the query requests other information that isn't stored in the aggregate tables.

Work With Logical Table Source Data Filters

This topic provides information to help you understand and add logical table source data filters.

Topics:

About Logical Table Source Data Filters

A logical table source's data filter limits the data returned from the physical table.

Each logical table source should contain data at a single intersection of aggregation levels. For example, you wouldn't want to create a source that has sales data at both the Brand and Manufacturer levels. If the physical tables include data at multiple levels, then add an appropriate data filter to constraint to values to a single level.

Add a Data Filter to a Logical Table Source

Add a data filter to limit the data returned from the logical table source's corresponding physical table.

For example, if the physical table contains data for all global regions, but you only need data for North America, then add a filter that returns data for North America only.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer**.
4. In the Logical Layer pane, browse for and double-click the table with the table source you want to add a data filter to.
5. In the table's tabs, click **Sources**.
6. In the table sources list, click the logical table source that you want to add a data filter to and then click **Detail view** to open the properties pane.
7. Scroll to Data Filter and click **Open Expression Editor**.
8. Create and validate the filter expression. Click **Save** to save the expression.
9. Optional: If the logical source table contains duplicate data, then click **Select distinct values**.
10. Click **Save**.

13

Create and Use Variables in a Semantic Model

This chapter contains information to help you understand how to create, manage, and use global, session, and static variables.

Topics:

About Semantic Model Variables

A semantic model variable contains a single value at any point in time. You use a variable instead of a literal or constant in data filters and expressions.

Types of Variables

You can create three types of semantic model variables:

- **Global variables** - A global variable has the same value for all users. You can let the system set the value, or specify a schedule to set the value. For example, you can define current period or current year as a global variable.
- **Session variables** - A session variable's value is specific to a user's session and is usually set when a user logs into Oracle Analytics, for example, user department or sales region.
- **Static variables** - A static variable holds a value that doesn't change, for example, minimum credit score or preferred credit score.

About Initialization Blocks and Variables

A global or session variable's initialization block contains a default initialization query that is run to initialize or refresh the variables defined in the initialization block. The initialization query references the data source's tables that supply the variable values. A query can populate values into several variables, that is one variable for each column in the query. You must also specify a connection pool that the initialization query uses to access the data source and return the variable values.

In addition to the default initialization query, you can choose to create data source-specific initialization queries for the data sources that your company is using (for example, Snowflake or DB2). If you define an additional initialization query, then Oracle Analytics uses it rather than the default initialization query to instantiate the initialization block and populate the variables.

For example, suppose you're an Oracle Fusion Cloud Applications customer and your data source is Snowflake. The semantic model delivered with Oracle Fusion Cloud Applications contains variables with their initialization queries written for Oracle Autonomous Database. To ensure that the variables included in the semantic model delivered with Oracle Fusion Cloud Applications work properly in your instance, you must modify the initialization blocks containing the variables to include an additional initialization query written for the Snowflake data source.

A static variable's initialization block doesn't contain an initialization query. To define a static variable, you specify the variable name and value.

After you define the variables in the initialization block, the variables are then available for you to include in data filters and expressions.

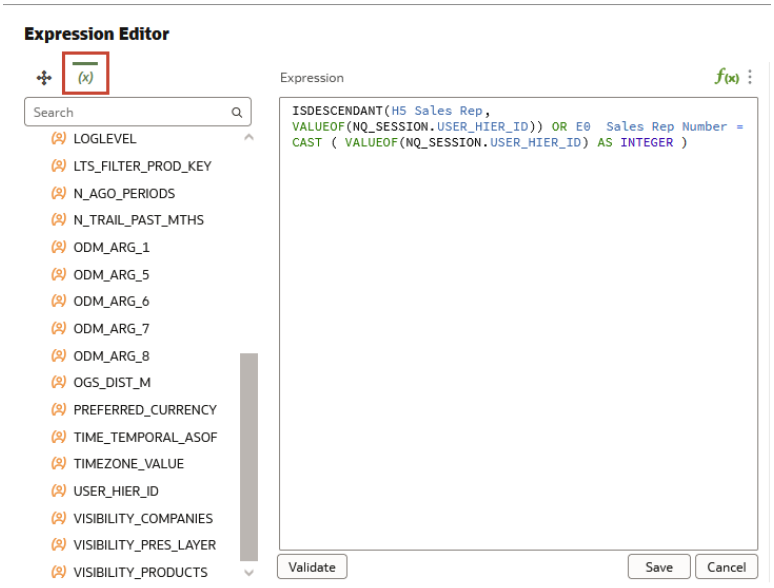
About Variables In Semantic Model Data Filters and Expressions

Variables are available for you to include in data filters and expressions. You use variables instead of literals or constants in expressions. You can't use variables to represent columns or other semantic model objects.

Variables are arguments of the function `VALUEOF()`. For example:

```
CASE WHEN "Hour" >= VALUEOF("prime_begin")AND "Hour" <
VALUEOF("prime_end") THEN 'Prime Time' WHEN ... ELSE...END
```

The Semantic Modeler displays the Variables tab in the Expression Editor. The Variables tab lists the variables that you can use in a data filter or expression. When you double-click a variable, the Expression Editor inserts the `VALUEOF()` function and the variable name.



Create and Configure Initialization Blocks

This topic describes what you need to know to understand, create, and configure initialization blocks.

Create an Initialization Block

Create an initialization block specifically for the type of variable that you want to create.

For a global or session variable, its initialization block contains the variable definition and the initialization query that supplies the variable with its value. For a static variable, its initialization block contains the variable definition, including the variable's default value. A static variable's initialization block doesn't contain an initialization query.

For information about how to define an initialization block for a specific type of variable, see Create a Global Variable, Create a Session Variable, or Create a Static Variable.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Variables**.
4. Click **Create** and then click **Create Initialization Block**.
5. In Create Initialization Block, go to **Name** and enter an initialization block name.
6. Go to the **Type** field and select the type of variable that you want to create. Click **OK**.

Open an Initialization Block

Open an initialization block to view or update its configuration, and to view, update, or add variable definitions.

For information about how to add variable definitions to an initialization block, see Create a Global Variable, Create a Session Variable, or Create a Static Variable.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Variables**.
4. In the Variables pane, browse or search for the initialization block that you want to open and double-click it.

Defer Session Variable Processing

To decrease logon time and save system resources, you can defer the processing of an initialization block containing many session variables.

Any new initialization blocks that you create are set to deferred execution by default. When you import a model created in Data Modeler or Model Administration Tool, the deferred execution property set in its variables are imported into Semantic Modeler.

If you defer the processing of a session variable initialization block, then any variable included in the initialization block is processed when it's accessed for the first time during the session rather than at logon time. And Oracle Analytics doesn't run initialization blocks that contain session variables that aren't used during the session.

The deferred run of an initialization block also triggers the processing of all unprocessed predecessor initialization blocks. All associated variables of the initialization block and its unprocessed predecessors are updated with the values returned from the deferred processing.

A message is displayed when you can't defer the processing of a session variable initialization block. See When You Can't Defer Session Variable Processing.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Variables**.
4. In the Variables pane, browse or search for the initialization block that you want to defer execution for and double-click it.
5. In the Initialization Block, click the **General** tab and then click **Allow deferred execution**.
6. Click **Save**.

When You Can't Defer Session Variable Processing

This topic explains when you can't defer session variable initialization block processing and the example messages that Semantic Modeler displays.

You can't defer variable initialization block processing when:

- The **Variable names and values** option is selected in the **Query Returns** field and the variables haven't been declared explicitly with default values.

  Example message: `"The execution of init block 'A_blk' cannot be deferred as it is using row-wise initialization."`

- The initialization block contains variables with the **Security Sensitive** option selected.

  Example message: `"The execution of init block 'A_blk' cannot be deferred as it is used by session variable 'A' which is security sensitive."`

- The initialization block is a predecessor to another initialization block that doesn't have the **Allow deferred execution** option selected.

Example message: "One of the successors for init block 'A_blk' does not have "Allow deferred execution" flag set. Init block 'B_blk' does not have "Allowed deferred execution" flag set.

About Dynamically Creating Session Variables and Setting Their Values

In a session variable's initialization block, you can select the **Variable names and values** option in the **Query Returns** field to create session variables dynamically and set their values when the session begins.

The names and values of the session variables reside in an external data source that you access through a connection pool. The variables receive their values from the initialization query that you provide.

Example 1 - Initialization Query With a Single Value Variable

Suppose you want to create session variables using values contained in a table named RW_SESSION_VARS. This table contains three columns:

- USERID - Contains values that represent the unique identifiers of the users.
- NAME - Contains values that represent session variable names.
- VALUE - Contains values that represent session variable values.

This table shows the example columns and their values.

USERID	NAME	VALUE
JOHN	LEVEL	4
JOHN	STATUS	FULL-TIME
JANE	LEVEL	8
JANE	STATUS	FULL-TIME
JANE	GRADE	AAA

To implement this example, create a session variable initialization block and select **Variable names and values** in the **Query Returns** field. Then in the **Select Statement** field, enter this initialization query:

```
SELECT NAME, VALUE
FROM RW_SESSION_VARS
WHERE USERID='VALUEOF(NQ_SESSION.USERID)'
```

NQ_SESSION.USERID is a system session variable that Oracle Analytics initializes for each user when they log on.

When initialized, this example creates the following session variables:

- When John connects to Oracle Analytics, his session contains two session variables: LEVEL containing the value 4, and STATUS containing the value FULL_TIME.
- When Jane connects to Oracle Analytics, her session contains three session variables: LEVEL containing the value 8, STATUS containing the value FULL-TIME, and GRADE containing the value AAA.

Example 2 - Initialization Query With a Multiple Value Variable

Suppose you want to create session variables using values contained in a table named `RW_SESSION_VARS`. This table contains three columns:

- `ROLE_NAME` - Contains values that represent user roles.
- `NAME` - Contains values that represent session variable names.
- `VALUE` - Contains values that represent session variable values.

This table shows the example columns and their values.

ROLE_NAME	NAME	VALUE
Role1	LEVEL	4
Role1	STATUS	FULL-TIME
Role2	GRADE	AAA

To implement this example, create a session variable initialization block and select **Variable names and values** in the **Query Returns** field. Then in the **Select Statement** field, enter this initialization query:

```
SELECT NAME, VALUE
FROM RW_SESSION_VARS
WHERE ';' || 'valueof(NQ_SESSION.ROLES)' || ';' like '%;' || ROLE_NAME || ';%'
```

If a user is assigned Role1 and Role 2, then `valueof(NQ_SESSION.ROLES)` returns the value `Role1;Role2`.

`NQ_SESSION.ROLES` is a system session variable that Oracle Analytics initializes for each user when they log on.

When initialized, this example creates the following session variables:

- When users assigned to Role1 connect to Oracle Analytics, their sessions contains two session variables: `LEVEL` containing the value `4`, and `STATUS` containing the value `FULL-TIME`.
- When users assigned to Role2 connect to Oracle Analytics, their sessions contain one session variable `GRADE` containing the value `AAA`.
- When users assigned to Role 1 and Role2 connect to Oracle Analytics, their sessions contain three session variables: `LEVEL` containing the value `4`, `STATUS` containing the value `FULL-TIME`, and `GRADE` containing the value `AAA`.

Use a List of Values to Initialize a Session Variable

You can configure a session variable's initialization block to initialize a session variable with a list of values.

To configure an initialization block to initialize a session variable with a list of values, in the initialization block configuration you must select **Variable names and values** in the **Query Returns** field.

When you select the **Variable names and values** field, the **Query Returns** field is displayed. Selecting the **Cache Query result** option puts the query's results in a main memory cache. The Oracle Analytics query engine uses the cached results for

subsequent sessions. This can reduce session startup time. However, the cached results might not contain the most current session variable values. If every new session needs the most current set of session variables and their corresponding values, you should clear this option.

The information and example in this topic pertain to Logical SQL. If you're using Physical SQL to initialize a variable with a list of values, then use the `VALUELISTOF` function.

For example, to get the customers assigned to the user names in the variable *LIST_OF_USERS*, use the following SQL statement in the initialization query:

```
SELECT 'LIST_OF_USERS', USERID
FROM RW_SESSION_VARS
WHERE NAME='STATUS' AND VALUE='FULL-TIME'
```

This SQL statement populates the variable `LIST_OF_USERS` with a colon-separated list of the values `JOHN` and `JANE` (for example, `JOHN:JANE`). You can then use this variable in a filter, as shown in the following `WHERE` clause:

```
WHERE TABLE.USER_NAME = valueof(NQ_SESSION.LIST_OF_USERS)
```

The variable `LIST_OF_USERS` contains a list of one or more values. The physical `IN` clause replaces the logical `WHERE` clause as shown in the following statement:

```
WHERE TABLE.USER_NAME IN ('JOHN', 'JANE')

Select 'LIST_OF_CUSTOMERS', Customer_Name from RW_CUSTOMERS where
RW.CUSTOMERS.USER_NAME in (VALUELISTOF(NQ_SESSION.LIST_OF_USERS))
```

To filter by specific values in a list, use `ValueNameof`. The first value is 0, not 1. For example:

```
Select 'LIST_OF_CUSTOMERS', Customer_Name from RW_CUSTOMERS where
RW.CUSTOMERS.USER_NAME in '(ValueNameOf(0,NQ_SESSION.LIST_OF_USERS))
```

Create a Schedule to Update Global Variable Values

You can schedule how often an initialization block's global variables' values are updated.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Variables**.
4. In the Variables pane, browse or search for the initialization block that you want to schedule and double-click it.
5. In the Initialization Block, click the **General** tab.
6. Go to the **Run every** field and specify how frequently you want to refresh the initialization block's variable values.
7. Go to the **Starting On** field and select the date and time when you want the initialization block's refresh schedule to begin.
8. Click **Save**.

Add an Additional Database Query to an Initialization Block

An initialization block must have a default initialization query. You can specify additional initialization queries specific to the data sources that your company is using (for example, Oracle Snowflake or DB2).

When you define an additional initialization query and the corresponding variable is used in a data filter or expression, Oracle Analytics skips the initialization block's default query. The data source-specific query bypasses the Oracle Analytics query engine to instantiate the initialization block and populate the variables.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Variables**.
4. In the Variables pane, browse or search for the initialization block that you want to add database query to and double-click it.
5. In the Initialization Block, click the **Variables** tab and then click **Specify query for additional databases**.
6. Click **Add database-specific query** and then select the database that you want to write a query for.
7. In the **Select Statement: <*your_database*>** field, write a select statement.
8. Click **Save**.

Initialization Queries Used in Variables to Override Selection Steps

For analyses that contain hierarchical columns, global variables or session variables can override selection steps.

Global and session variables intended for this purpose must use valid JSON syntax.

Using JSON, you must define type, column, and members with the following syntax.

```
{
  "type": "Hierarchy",
  "column": {
    "subject_area":"your_subject_area",
    "hier_id":"your_hier_id",
    "dim_id":"your_dim_id",
    "table_name":"your_table_name"
  },
  "members": [
    {
      "level_id":"your_level_id",
      "values": [
        your_value,
        your_value
      ]
    },
    {
      "level_id":"your_level_id",
      "values": [
        your_value
      ]
```

```
    }
  ]
}
```

Where:

`"type"` indicates hierarchy type.

`"column"` indicates the hierarchy column's information such as subject area and table name.

`"dim_id"` is the logical hierarchy name.

`"members"` indicates which hierarchy level and which member ID.

`"level_id"` is the presentation level name.

Example of Standard Hierarchy Syntax

```
{
  "type": "Hierarchy",
  "column": {
    "subject_area": "A - Sample Sales",
    "hier_id": "H2 Offices",
    "dim_id": "H3 Offices",
    "table_name": "Offices"
  },
  "members": [
    {
      "level_id": "Company",
      "values": [
        10001,
        10002
      ]
    },
    {
      "level_id": "Organization",
      "values": [
        1005
      ]
    }
  ]
}
```

Example of Parent-Child Hierarchy Syntax

```
{
  "type":"Hierarchy",
  "column":{
    "subject_area":"A - Sample Sales",
    "hier_id":"Sales Rep Hierarchy",
    "dim_id":"H5 Sales Rep",
    "table_name":"Sales Person"
  },
  "members":[
    {
      "level_id":"Grand Total",
      "values":[
        27,
        24,
        18,
        16
```

```
        ]
      }
    ]
}
```

Test an Initialization Block's Query

Test the initialization block's initialization query to confirm that the connection pool is working properly, the query returns the expected values, and the values are correctly assigned to the variables that you defined.

It is best practice to create and use a dedicated connection pool for initialization blocks. See About Connection Pools for Initialization Blocks.

If an initialization block fails because of a particular connection pool, then no more initialization blocks using that connection pool are processed. Instead, the connection pool is blocked and subsequent initialization blocks for that connection pool are skipped.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Variables**.
4. In the Variables pane, browse or search for the initialization block that you want to test and double-click it.
5. In the Initialization Block, click the **Variables** tab and then click **Test Query**.

Change the Order of Variables in an Initialization Block

If you test an initialization block's query and the variables are populated with the wrong values, then you might need to change the order of the variables.

The initialization query's column order and the variable order specified in the initialization block determines the column value assigned to each variable. When you associate variables with an initialization block, the first column specified in the query is assigned to the first variable in the list. If the initialization query's column order doesn't match the variables' order, then the variables are populated with the wrong values.

The number of associated variables could differ from the number of columns retrieved. If there are fewer variables than columns, extra column values are ignored. If there are more variables than columns, the additional variables aren't refreshed and the variables retain their original values.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Variables**.
4. In the Variables pane, browse or search for the initialization block that you want to disable or enable and double-click it.
5. In the Initialization Block, click the **Variables** tab.
6. Go to the variables list and click to highlight the variable that you want to move. Click **Move Up** or **Move Down**.
7. Click **Save**.

Add Dependencies to an Initialization Block

When a semantic model has multiple initialization blocks, you can set the order that the blocks are initialized in.

If you don't set dependencies, then Oracle Analytics runs all initialization blocks at the same time. This results in null values because the variable values aren't returned in the necessary order.

To add dependencies, you first open the initialization block that you want to be run last and then add the initialization blocks that you want to be run before the block you've opened. For example, suppose a semantic model has two initialization blocks, A and B. You open initialization Block B, and then specify that Block A runs before Block B. If you're setting dependencies for session initialization blocks that include schedules, then Block A runs according to Block B's schedule in addition to its own schedule.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Variables**.
4. In the Variables pane, browse or search for the initialization block that you want to add a dependency to.
5. In the Initialization Block, click the **Dependencies** tab.
6. Click **Add Initialization Block** and browse for and select an initialization block to add it to the Dependencies list.
7. Click **Save**.

Disable or Enable an Initialization Block

You can disable or enable any global or session initialization blocks. By default, initialization blocks are enabled.

You might disable or enable an initialization block for testing purposes.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Variables**.
4. In the Variables pane, browse or search for the initialization block that you want to disable or enable and double-click it.
5. In the Initialization Block, click the **General** tab and then click **Disable** to disable the initialization block, or clear **Disable** to enable the initialization block.
6. Click **Save**.

Define Global Variables

This topic describes what you need to know to understand and define global variables for use in data filters and expressions.

Topics:

About Global Variables

Use a global variable when you need a variable's value to be the same for all users.

To define a global variable, you create or use an existing initialization block to contain one or more global variables. The initialization block contains an initialization string and connection pool to access the data source and return results to populate the global variables that you define. The global variables are then available for you to add to data filters or expressions.

For global variables, you can scheduled the initialization block to refresh variable values as needed.

A common use of global variables is to set filters in logical table sources. See About Global Variables and Logical Table Source Fragmentation.

Create a Global Variable

Create a global variable when you need the variable's value to be the same for all users, for example, current period or current year.

After you define and save global variables, they're available for you to add to the semantic model's data filters or expressions.

In a semantic model, you create and define a global variable within an initialization block. You can't create and define a standalone global variable and then later associate it with an initialization block.

Consider the following information when creating a global variable:

- If you add more than one variable to the initialization block, then the variables must match the column order in the initialization query. This is so each variable receives the proper value when the initialization query runs.
- The initialization query used to populate the variables must reference the data source tables needed for the variable values. You don't have to include the data source tables that supply the variable values in the semantic model's physical layer.
- If you're creating a variable to override selection steps in a hierarchy column, then use JSON syntax to write the initialization query. See Initialization Queries Used in Variables to Override Selection Steps.
- Because object permissions don't apply to variables, the values in variables aren’t secure and anyone who knows or can guess the name of the variable can use it in

an expression. Because of this, Oracle recommends that you don't put sensitive data like passwords in variables.

Follow these steps to create a global variable:

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Variables**.
4. Click **Create** and then click **Create Initialization Block**.
5. In Create Initialization Block, go to the **Name** field and enter an initialization block name.
6. Go to the **Type** field and select **Global**. Click **Add**.
7. Confirm that the **Variables** tab is displayed, and then go to the **Select Statement: DEFAULT** field and enter the initialization query.
8. Go to **Connection Pool** and click **Select** to browse for and select a connection pool specifically for use in initialization blocks.
9. Click **Add Variable** and enter a unique name.
10. Use one of the following options to specify a default value.
    - Go to the **Value** field and enter a default value.
    - Leave the **Value** blank if you want null as the default value.
    - Click **Detail View** and click **Open Expression Editor** to create an expression that determines the default value.
11. Optional: Click **Add Variable** to add another variable, and use the **Move Up** and **Move Down** to position the variables in the correct order.
12. Optional: Click **Test Query** to review the variable values returned by the initialization query and to confirm that the variables are receiving the correct values.
13. Click **Save**.

Define Session Variables

This topic describes what you need to know to understand and define session variables for use in data filters and expressions.

About Session Variables

Use a session variable when you need a variable with a value specific to a user's session and is set when a user logs into Oracle Analytics. Use session variables to set filters and permissions for the session.

Session variables dynamically modify metadata content to adjust to a changing data environment. For example, suppose User1 belongs to Department1 and User2 belongs to Department2. These users must access only the data for their respective departments. In this

case you can create and use the DEPARTMENT_NUMBER variable to store the appropriate values for User1 and User2. You can then use this variable to filter data by Department2 for User1 and Department2 for User2.

To define a session variable, you create or use an existing initialization block to contain one or more session variables. The initialization block contains a default initialization query and connection pool to access the data source and return results to populate the session variables that you define. The session variables are then available for you to add to the semantic model's data filters or expressions.

Unlike global variables, the initialization of session variables isn't scheduled. When a user begins a session, Oracle Analytics creates new instances of session variables and initializes them. Session variable values remain unchanged for the duration of the session.

There are as many instances of a session variable as there are active sessions on Oracle Analytics. You can initialize each instance of a session variable to a different value.

Initialization blocks that contain many session variables can slow performance. You can defer the processing of session variable initialization blocks during session logon until their associated session variables are actually accessed within the session. See Defer Session Variable Processing.

About Multi-Source Session Variables

Create and use multi-source session variables when you need a variable to provide values from more than one data source. You can use multi-source session variables in data filters and expressions.

There is no restriction to the number of values that the multi-source session variable can hold.

In a session initialization block, you use the following format to create a session variables for each source. This format contains four underscore characters as the separator between the variable name and the source.

<ms_variable_name>____<source>

The multi-source system variable definitions that you created are listed in the saved session initialization block's definition (for example, *MVCOUNTRY____ORCL* and *MVCOUNTRY____SNFL*). But when you create expressions that include the multi-source session variable name, the Expression Editor's Variables tab displays the variable name (for example, *MVCOUNTRY*).

For an example of how to create multi-source session variables, see Create a Multi-Source Session Variable.

You can add values to the multi-source session variable from other component initialization blocks that return values. The multi-source session variable fails if all of the component initialization blocks return null values.

You can set processing dependencies and deferred processing for multi-source session variables, similar to regular session variables.

Create a Session Variable

Create a session variable when you need the variable's value to be specific to a user's session and set when a user logs into Oracle Analytics. For example, user department or sales region.

After you define and save session variables, they're available for you to add to data filters and expressions.

In a semantic model, you create and define a session variable within an initialization block. You can't create and define a standalone session variable and then later associate it with an initialization block.

Consider the following information when creating a session variable:

- If you add more than one variable to the initialization block, then the variables must match the column order in the initialization query. This is so each variable receives the proper value when the query is run.
- The initialization query used to populate the variables must reference the physical tables needed for the variable values. You don't have to include the physical tables that supply the variable values in the semantic model's physical layer.
- The **Enable any user to set the value** option allows any user to set the variable's value in an analyses or dashboard (for example, a What If analysis). The user-specified variable value is passed to the Oracle Analytics query engine and used in the underlying calculation.
- The **Security Sensitive** option identifies the variable as sensitive to security when using a row-level database security strategy such as Virtual Private Database (VPD). When this option and the database's **Virtual Private Database** data source property are selected, then the Oracle Analytics query engine matches a list of security-sensitive variables to each prospective cache hit. Cache hits occur only on cache entries that include and match all security-sensitive variables.
- If you're creating a variable to override selection steps in a hierarchy column, then use JSON syntax to write the initialization query. See Initialization Queries Used in Variables to Override Selection Steps.
- Because object permissions don't apply to variables, the values in variables aren’t secure and anyone who knows or can guess the name of the variable can use it in an expression. Because of this, Oracle recommends that you don’t put sensitive data like passwords in variables.

Follow these steps to create a session variable:

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Variables**.
4. Click **Create** and then click **Create Initialization Block**.
5. In Create Initialization Block, go to the **Name** field and enter an initialization block name.
6. Go to the **Type** field and select **Session**. Click **Add**.
7. Confirm that the **Variables** tab is displayed, and then go to the **Select Statement: DEFAULT** field and enter the initialization query.

8. Go to **Connection Pool** and click **Select** to browse for and select a connection pool specifically for use in initialization blocks.
9. Click **Add Variable** and enter a unique name.
10. Use one of the following options to specify a default value.
    - Go to the **Value** field and enter a default value.
    - Leave the **Value** blank if you want null as the default value.
    - Click **Detail View** and click **Open Expression Editor** to create an expression that determines the default value.
11. Optional: Select **Enable any user to set the value** to allow the user to set the variable's value in an analyses or dashboard (for example, a What If analysis).
12. Optional: Select **Security Sensitive** to identify the variable as sensitive to security when using a row-level database security strategy, such as a Virtual Private Database (VPD).
13. Optional: Click **Add Variable** to add another variable, and use the **Move Up** and **Move Down** to position the variables in the correct order.
14. Optional: Click **Test Query** to review the variable values returned by the initialization query and to confirm that the variables are receiving the correct values.
15. Click **Save**.

Example - Create and Use a Multi-Source Session Variable

This topic provides an example of how to create and use the MVCOUNTRY multi-source session variable.

Topics:

Create a Multi-Source Session Variable

This topic explains how to create the MVCOUNTRY multi-source session variable. When you add the MVCOUNTRY variable to an expression or data filter, it returns data from the Oracle and Snowflake data sources.

After you create a multi-source session variable, the variables definitions are listed in the initialization block's definition (for example, *MVCOUNTRY____ORCL* and *MVCOUNTRY____SNFL*). But in the Expression Editor, the multi-source session variable name is displayed (for example, *MVCOUNTRY*).

See About Multi-Source Session Variables.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Create the first variable.
    a. Click **Variables**.

b. Click **Create** and then click **Create Initialization Block**.

c. In Create Initialization Block, go to the **Name** field and enter `mvcountry_orcl_init`.

d. Go to the **Type** field and select Session. Click **Add**.

e. Confirm that the **Variables** tab is displayed, and then go to the **Query Returns** field and select **Variable names and values**.

f. Go to the **Select Statement: DEFAULT** field and enter the following initialization query, using four underscores between the variable name (MVCOUNTRY) and the source (ORCL):

```
select distinct 'MVCOUNTRY____ORCL', country from oracle_table
```

g. Go to **Connection Pool** and click **Select** to browse for and select a connection pool.

h. Click **Add Variable** and enter the name `MVCOUNTRY____ORCL`. You need to add four underscores between the variable name and the source name.

i. Click **Save**.

4. Create the second variable.

a. Click **Variables**.

b. Click **Create** and then click **Create Initialization Block**.

c. In Create Initialization Block, go to the **Name** field and enter `mvcountry_snfl_init`.

d. Go to the **Type** field and select Session. Click **Add**.

e. Confirm that the **Variables** tab is displayed, and then go to the **Query Returns** field and select **Variable names and values**.

f. Go to the **Select Statement: DEFAULT** field and enter the following initialization query, using four underscores between the variable name (MVCOUNTRY) and the source (SNFL):

```
select distinct 'MVCOUNTRY____SNFL', country from snowflake_table
```

g. Go to **Connection Pool** and click **Select** to browse for and select a connection pool.

h. Click **Add Variable** and enter the name `MVCOUNTRY____SNFL`. You need to add four underscores between the variable name and the source name.

i. Click **Save**.

Use a Multi-Source Session Variable in an Expression

After you create the MVCOUNTRY multi-source session variable, you can use it in an expression.

For information about the MVCOUNTRY session variable used in this example, see Create a Multi-Source Session Variable.

This is an example of how to use the multi-source session variable in an expression:

```
select lastName, firstName, country from employee
where country=VALUEOF(NQ_SESSION.MVCOUNTRY)
```

Use a Multi-Source Session Variable in a Data Filter

After you create the MVCOUNTRY multi-source session variable, you can use it in a data filter.

The MVCOUNTRY multi-source session variable is displayed in the Expression Editor's Variables tab. For information about the MVCOUNTRY session variable used in this example, see Create a Multi-Source Session Variable.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Browse for and open a logical or presentation table.
4. Click the **Data Filters** tab.
5. Go to the **Add** field, enter the application role that you want to set the data filter for, click **Search by Role Name**, and from the list select the application role.
6. In the Role Name list, click the role you added to highlight it, and then click **Open Expression Editor**.
7. Enter this expression:

   ```
   Country=VALUEOF(NQ_SESSION.MVCOUNTRY)
   ```

8. Optional: Click **Validate**.
9. Click **Save** to save the expression.
10. Click **Save** to save the semantic model.

Define Static Variables

This topic describes what you need to know to understand and define static variables for use in data filters and expressions.

Topics:

- About Static Variables
- Create a Static Variable

About Static Variables

Use a static variable when you need a variable with a fixed value.

For example, suppose you want to create an expression to group times of day into different day segments. If Prime Time were one of those segments and corresponded to the hours between 5:00 PM and 10:00 PM, you could create a `CASE` statement like the following:

```
CASE WHEN "Hour" >= 17 AND "Hour" < 23 THEN 'Prime Time' WHEN... ELSE...END
```

Hour is a logical column, mapped to a timestamp physical column using the date-and-time `Hour(<<timeExpr>>)` function.

Instead of entering the numbers 17 and 23 into this expression as constants, you could create and use a static variable named `prime_begin`, set the variable's value to

`17`, and then create another variable named `prime_end` and set the variable's value to `23`.

When you create a static variable, you must include a default value. You can set the **Value** field with a number, character, date, time, or timestamp value. Or you can use the Expression Editor to insert the Date, Time, and TimeStamp constants into an expression.

Create a Static Variable

Create a static variable when you need a variable with a value that doesn't change, for example, minimum credit score or preferred credit score.

After you define and save static global variables, they're available for you to add to the semantic model's data filters or expressions.

In a semantic model, you create and define a static variable within an initialization block. You can't create and define a standalone static variable and then later associate it with an initialization block.

A static variable must have a default value that is a numeric, character, date, time, or timestamp value. If you initialize a static variable using a character string, enclose the string in single quotes ('). You can use the Expression Editor to insert the Date, Time, or Timestamp constants.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Variables**.
4. Click **Create** and then click **Create Initialization Block**.
5. In Create Initialization Block, go to the **Name** field and enter an initialization block name.
6. Go to the **Type** field and select **Static**. Click **Add**.
7. Confirm that the **Variables** tab is displayed and then click **Add Variable** and enter a unique name.
8. Go to the **Value** field and enter a static value. Or, click **Detail View** and click **Open Expression Editor** to create an expression that determines the static value.
9. Click **Save**.

14

Support Multilingual Data

Oracle Analytics supports several language translations. This chapter section describes how you configure field information to display in multiple languages

Topics:

What Is Multilingual Data Support?

Multilingual data support is the ability to display data from database schemas in multiple languages.

Oracle Analytics supports multilingual schemas by simplifying the administration and improving query performance for translations. Multilingual schemas typically store translated fields in separate tables called lookup tables. Lookup tables contain translations for descriptor columns in several languages, while the base tables contain the data in the base language. Descriptor columns provide a textual description for a key column where there is a logical one-to-one relationship between the descriptor column and the key column. An example of a descriptor column might be Product_Name, which provides textual descriptions for a Product_Key column.

What is Lookup?

Lookup is when a query joins the base table and lookup table to obtain the translated values for each row in the base table.

Lookup tables might be dense and sparse in nature. A dense lookup table contains translations in all languages for every record in the base table. A sparse lookup table contains translations for only for some records in the base tables. Sometimes it is also possible that lookup tables are both dense and sparse. For example, a lookup table might contain complete translation for the Product Description field but only partial translation for the Product Category field. Dense and Sparse are types of lookup operation rather than being a table property. You configure lookup tables using the Semantic Modeler.

What Is Double Column Support?

Double column support is the ability to associate two columns (a descriptor ID column and a descriptor column) in the logical layer, and can help you to define language independent filters.

When the user creates a filter based on a descriptor column, the query tool displays a list of values to the user that are selected from the descriptor column.

This descriptor column technique is also useful when dealing with queries that involve LOB data types such as CLOBs and BLOBs and aggregate functions such as `COUNT` or `SUM`. Some data sources don't allow LOB columns to be used in the `GROUP BY` clause. So, instead of adding the LOB column to the `GROUP BY`, it's necessary to group by some other column that has a one-to-one relationship with the LOB column and then in join the LOB column after the aggregates have been computed.

See Add Double Column Support.

Design Translation Lookup Tables in Multilingual Schema

The two common techniques of designing translation lookup tables: design a lookup table for each base table, and design a lookup table for each translated field.

Lookup Table for Each Base Table

There is often a separate lookup table for each base table. The lookup table contains a foreign key reference to records in the base table, and contains the values for each translated field in a separate column.

Assuming a completely dense lookup table, the number of rows in the lookup table for a particular language equals the number of rows in the base table.

The example in the figure below shows each record in the lookup table matching only one row in the base table.

Base Table

Key	Code	Description	Category_Code	Category
1	A123	Bread	D45	Groceries
2	B234	Marmalade	D45	Groceries
3	C345	Milk	D45	Groceries

Lookup Table

Key	Language_Key	Description	Category
1	DE	Brot	Lebensmittelgeschaft
1	IT	Pane	Drogheria
2	DE	Marmelade	Lebensmittelgeschaft
2	IT	Marmeleta di agrumi	Drogheria
3	DE	Milch	Lebensmittelgeschaft
3	IT	Latte	Drogheria

Lookup Table for Each Translated Field

The alternative approach to using one lookup table for each base table involves a separate lookup table for each translated field, as shown in the figure below.

Getting the translated value of each field requires a separate join to a lookup table. In practice there is often just one physical table that contains translations for multiple fields. When a single table contains translations for multiple fields, you must place a filter on the lookup table to restrict the data to only those values that are relevant to a particular column in the base table.

Base Table

Key	Code	Description	Category_Code	Category
1	A123	Bread	D45	Groceries
2	B234	Marmalade	D45	Groceries
3	C345	Milk	D45	Groceries

Lookup Table

Field_Key	Value_Key	Language_Key	Translation
Description	A123	DE	Brot
Description	A123	IT	Pane
Description	B234	DE	Marmelade
Description	B234	IT	Marmeleta di agrumi
Description	C345	DE	Milch
Description	C345	IT	Latte
Category	D45	DE	Lebensmittelgeschaft
Category	D45	IT	Drogheria

Create Logical Lookup Tables and Logical Lookup Columns

This section describes creating logical lookup tables and lookup columns.

Topics:

Create Logical Lookup Tables

You create a logical lookup table object in the business model to define the necessary metadata for a translation lookup table.

A lookup table is a logical table with a property that designates it as a lookup table, as described in Designate a Logical Table as a Lookup Table. The figure below provides an example of a lookup table.

Product_Translation Table

Product_Code	Language_Key	Description
A123	DE	Brot
A123	IT	Pane
B234	DE	Marmelade
B234	IT	Marmelde di agrumi
C345	DE	Milch
C345	IT	Latte

- Each of the lookup table's primary keys are considered together as a Lookup Key and perform the lookup. The lookup can be performed only when the values for all lookup key columns are provided. For example, in the figure above, the combined Product_Code and Language_Key form the primary key of this lookup table.
- A lookup key is different from a logical table key because lookup key columns are order sensitive. For example, Product_Code and Language_Key are considered a different lookup key to Language_Key and Product_Code. All columns of the lookup key must be joined in the lookup function.
- A lookup table has a combination of lookup keys.
- A lookup table has at least one value column. In the figure above, the value column is Description, and it contains the translated value for the product description.
- There must be a functional dependency from a lookup key to each value column. That is the lookup key can identify the value column. The lookup key and value column should both belong to the same physical table.
- A lookup table is standalone without joining to any other logical tables.

  The consistency check rules are relaxed for lookup tables, such that if a table is designated as a lookup table, it need not be joined with any other table in the subject area (logical tables would normally be joined to at least one table in the subject area).
- The aggregation results when using lookup columns should match the results from the base column. For example, the following code

  ```
  SELECT productname_trans.productname, sales.revenue FROM
  snowflakesales;
  ```

  should return the same results as

  ```
  SELECT product.productname, sales.revenue FROM snowflakesales;
  ```

  If the lookup table productname_trans in this example uses the lookup key ProductID and LANGUAGE, then both queries return the same aggregation results.

  If the lookup key contains a column with a different aggregation level to productname, then the query grain changes and this affects the aggregation.

Designate a Logical Table as a Lookup Table

A logical table must be designated as a lookup table (using the Semantic Modeler) before you can use it as a lookup table.

To designate a logical table as a lookup table, you must first import the lookup table into the physical layer and drag and drop it into the logical layer.

The order in which the columns are specified in the lookup table primary key determines the order of the corresponding arguments in the `LOOKUP` function.

For example, if the lookup table primary key consists of the RegionKey, CityKey, and LanguageKey columns, then the matching arguments in the `LOOKUP` function must be specified in the same order. You use the Semantic Modeler to change the order of primary key columns.

About the LOOKUP Function Syntax

A `LOOKUP` function is typically used in the logical layer, as an expression in a translated logical table column.

The syntax of the `LOOKUP` function is as follows:

```
Lookup ::= LookUp([DENSE] value_column, expression_list ) | LookUp(SPARSE
value_
column, base_column, expression_list )

expression_list ::= expr {, expression_list }

expr ::= logical_column | session_variable | literal
```

For example:

```
LOOKUP( SPARSE SnowflakeSales.ProductName_TRANS.ProductName,
SnowflakeSales.Product.ProductName, SnowflakeSales.Product.ProductID,
VALUEOF(NQ_SESSION."LANGUAGE"))

LOOKUP( DENSE SnowflakeSales.ProductName_TRANS.ProductName,
SnowflakeSales.Product.ProductID, VALUEOF(NQ_SESSION."LANGUAGE"))
```

Note the following:

- A `LOOKUP` function is either dense or sparse, and is specified using the keyword `DENSE` or `SPARSE`. The default behavior is dense lookup, if neither `DENSE` or `SPARSE` is specified. For `DENSE` lookup, the translation table is joined to the base table through an inner join, while for `SPARSE` lookup, a left outer join is performed.
- The first parameter (the parameter after the `DENSE` or `SPARSE` keyword) must be a valid value column from a valid lookup table that's defined in the logical layer.
- If the `SPARSE` keyword is given, then the second parameter must be a column that provides the base value of the value_column. For `DENSE` lookup, this base column isn't required.

- The number of expressions in the expression_list must be equal to the number of the lookup key columns that are defined in the lookup table, which is defined by the value_column. The expression that's specified in the expression list must also match the lookup key columns one by one in order.

  For example:

  – The lookup key for lookup table ProductName_TRANS is both Product_code and Language_Key

  – The expressions in expression_list are SnowflakeSales.Product.ProductID and VALUEOF(NQ_SESSION."LANGUAGE")

  – The meaning of the lookup is:

    return the translated value of ProductName from the translation table with the condition of Product_code = SnowflakeSales.Product.ProductID and Language_Key = VALUEOF(NQ_SESSION."LANGUAGE")

Create Logical Lookup Columns

You use the Expression Builder in the Semantic Modeler to create a logical column that includes the lookup function.

The value of the logical column depends on the language that is associated with the current user.

You create a logical column using a derived column expression in the column properties pane located in the logical table's Columns tab. See Create Derived Columns.

For example to get the translated product name:

```
INDEXCOL( VALUEOF(NQ_SESSION."LAN_INT"), "Translated Lookup
Tables"."Product". "ProductName",
LOOKUP( DENSE "Translated Lookup Tables"."Product Translations".
"ProductName", "Translated Lookup Tables"."Product"."ProductID",
VALUEOF(NQ_SESSION."WEBLANGUAGE")))
```

`LAN_INT` is a session variable that's populated by the session initialization block MLS and represents either the base language or other languages:

- 0 for base language (for example, en - English)
- 1 for other language codes (for example, fr - French, or cn - Chinese)

`WEBLANGUAGE` is a session variable that is initialized automatically, based on the language selected when a user logs in.

The `INDEXCOL` function helps to select the appropriate column. In the preceding example, the expression returns the value of the base column (ProductName) only if the user language is the base language (that is, when the value of session variable `LAN_INT` is 0). If the user language isn't the base language (when the value of the session variable `LAN_INT` is 1), then the expression returns the value from the lookup table of the language that's passed in the `WEBLANGUAGE` session variable.

When you use the `DENSE` function (shown in the previous example), if there's no value for a column in the translated language, then the lookup function displays a blank entry.

When you use the `SPARSE` function (shown in the following example), and there is no value for a column in the translated language, then the lookup function displays a corresponding value in the base language.

```
INDEXCOL( VALUEOF(NQ_SESSION."LAN_INT"), "Translated Lookup
Tables"."Product".
"ProductName", LOOKUP( SPARSE "Translated Lookup Tables"."Product
Translations".
"ProductName", "Translated Lookup Tables"."Product"."ProductName",
"Translated
Lookup Tables"."Product"."ProductID", VALUEOF(NQ_SESSION."WEBLANGUAGE")))
```

Create Physical Lookup Tables and Physical Lookup Columns

You can create physical lookup table objects in the logical layer to define the necessary metadata for translation lookup tables. Physical lookup tables are similar to logical lookup tables in both semantics and usage.

Physical lookup tables address the following scenarios that logical lookup tables can't handle:

- The lookup table source is fragmented. In this case, use multiple physical lookup tables to hold the values. For example, translation values for fragmented product name data can be distributed in two physical lookup tables called productname_trans_AtoM and productname_trans_NtoZ.
- Different levels of translation tables are available. It's preferable to use the same source as the base query.

Unlike logical lookup tables, you configure physical lookup tables by constructing lookup functions in the logical table source mapping.

For example, suppose that you have the following physical tables:

- A base table called Categories, with columns such as categoryid and categoryname.
- A translation table called Categories_Trans, with columns such as categoryid, language_key, and categoryname. The translated value of categoryname is determined through a combination of the categoryid and language_key columns.

Suppose that you have a logical table called Categories. In that table, you add a new logical column called categoryname_p, which is a translation column that depends on the current language. The column isn't derived from any other logical column (unlike logical lookup columns).

The following procedure explains how to configure a physical lookup translation column using the previous example.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer** and browse for and double-click the table (for example, Categories) where you want to add a logical column.
4. In the logical table, click **Columns** and then click **Add Column**. In the New Column_1 row enter a column name (for example, categoryname_p). Press Enter.
5. Click **Sources**.

6. In the logical table sources list table, click the logical table source and then click **Detail view** to open the properties pane.
7. Scroll to Column Mapping, click the **Show** field, and select **Unmapped**.
8. Locate and click the new logical column (for example, categoryname_p) to select it, and then click it again to display the expression field. Click the expression editor icon and create an expression similar to the following:

```
INDEXCOL(VALUEOF(NQ_SESSION."LAN_INT"),
"DB_Name"."My_Category"."My_Schema"."Categories"."CategoryName",
LOOKUP(SPARSE
"DB_Name"."My_Category"."My_Schema"."CATEGORIES_TRANS"."CATEGORYNAME
",
"DB_Name"."My_Category"."My_Schema"."Categories"."CategoryName",
"DB_Name"."My_Category"."My_Schema"."Categories"."CategoryID",
VALUEOF(NQ_SESSION."LANGUAGE")))
```

9. In Logical Table Source, click **OK**.

The Categories_trans physical translation table doesn't need to be incorporated into the logical table source. The `INDEXCOL` function checks that if the `LAN_INT` session variable is 0, then the categoryname column is fetched from the base table. Note the following about the `LOOKUP` function:

- The physical `LOOKUP` function works the same as a logical `LOOKUP` function. The only difference is that all the references to logical tables and columns are replaced by physical tables and columns.
- The first column of the `LOOKUP` function is a value column, which is a translation value column from a translation table. The second column is the base value column, if a sparse lookup exists. The remaining columns are columns or values to be joined to the physical translation table, which is the table that's implied by the value column of the `LOOKUP` function.

Because you can't use a dialog box to configure a physical lookup table, you must ensure that the order of the join columns and values is compatible with the column sequence displayed in the Additional Keys section of the physical table's General tab for the physical translation table. For example, the Additional Keys section for the Categories_trans table indicates that the primary key is composed of the CategoryID and Language_Key columns.

The columns that are specified in the `LOOKUP` function correspond to these columns:

- The following line:

  ```
  "DB_Name"."My_Category"."My_Schema"."Categories"."CategoryID"
  ```

  corresponds to the Categories_trans.CategoryID column.
- The following line:

  ```
  valueof(NQ_SESSION."LANGUAGE")
  ```

  corresponds to the Categories_trans.Language_key column.

See Create Logical Lookup Columns for information about lookup concepts like the `LAN_INT` and `LANGUAGE` session variables and full syntax information for the `LOOKUP` function.

Enable Lexographical Sorting

Lexicographical sorting is the ability to sort data in alphabetical order.

Most data sources support lexicographical sorting. However, if you notice that lexicographical sorting isn't working properly for a particular data source, then you can configure the Oracle Analytics query engine to perform the sort rather than the back-end data source. To perform this configuration, you need to confirm or deselect the semantic model database's ORDERBY_SUPPORTED supported query feature. See Modify a Database's Supported Query Features.

Note that disabling ORDERBY_SUPPORTED in the database can have a very large performance impact because consequently many joins aren't pushed down to the data source. In many cases, the performance impact is significant enough that ORDERBY_SUPPORTED can still be enabled in the data source regardless of the impact on the lexicographical sorting functionality.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer** and locate and double-click a database.
4. In the database, click **Advanced**.
5. Go to the Supported Query Features section of the Features table and locate the **ORDERBY_SUPPORTED** feature.
6. Confirm that the **ORDERBY_SUPPORTED** feature is deselected.
7. Click **Save**.

15

Apply Data Access Security to Semantic Model Objects

This chapter explains the data access security types and how to implement them in your semantic model.

Topics:

About Data Access Security

After developing your semantic model, you need to set up a data security architecture to control source data access.

Set up data access security to meet data security requirements such as:

- Protect business data from unauthorized access.
- Protect your semantic model's metadata such as measure definitions.
- Prevent individual users from damaging overall system performance.

You can set up three types of data security: row-level security, object permissions, and query limits (governors).

In the semantic model, you set up object permissions and query limits, which are then enforced by the Oracle Analytics query engine. You can add row-level data security, which is also enforced by the Oracle Analytics query engine, to both the semantic model and the database.

It's best practice to implement row-level security in the database and object permissions and query limits in the semantic model. Although it's possible to provide database-level object restrictions on individual tables or columns, objects that users don't have access to are still visible in all clients even though queries against them fail. It's better to set up object permissions in the semantic model so that objects that users don't have access to are hidden in all clients.

To control user access to workbooks, dashboards, or analyses, set up access and read and write permissions at the workbook, dashboard, or analyses object level.

If you implement security only in workbooks, dashboards, or analyses, then the deployed semantic model and database are exposed to SQL injection hacker attacks and other security vulnerabilities. Implementing object-level security and data and row-level security in the semantic model prevents these attacks and vulnerabilities. This security applies to all incoming clients.

Work With Row-Level Security

This topic provides information to help you understand and define semantic model row-level security.

Topics:

About Row-Level Security

Some data sources apply row-level security policies to determine what data can be queried by an individual user.

Data security is described using various terms such as row-level security, data-level security, or Virtual Private Database (VPD) policies. This document uses the term row-level security.

Some data sources support connections using a privileged user that can impersonate the end user running a query. Connection pools allow parameterization of connection string information, and on-connection and on-query scripts that run prior to data queries. When Oracle Analytics connects to a data source by using a privileged user that can impersonate the actual end user, the data source’s data security policies apply to the end user queries.

In addition to the connection string and query script configuration, Oracle Analytics provides a Virtual Private Database (VPD) data source property for each database in the semantic model's physical layer. When you enable the Virtual Private Database (VPD) option, you can prevent sharing of query cache between users because each user needs to retrieve only the data they are permitted to query.

You must define the users, permissions, and security policies in the database. Refer to your database documentation for more information.

You can use a connection script to achieve the same row-level security for Oracle Database data sources.

Where to Set Up Row-Level Security

You can set up row-level security in the semantic model or in the database.

Implementing row-level security in the semantic model provides benefits such as:

- All users share the same database connection pool for better performance.
- All users share cache for better performance.

- Security rules can be defined and maintained to apply across many federated data sources.

Implementing row-level security in the database is good for situations where multiple applications share the same database. When you design and implement row-level security in the database, you should also define and apply object permissions in the semantic model.

Although it's possible to set up row-level security in both the semantic model and in the database, you typically don't enforce row-level security in both places unless there is a specific need to do so.

Set Up Row-Level Security in the Database

Implement row-level security in the database when multiple applications share the same database.

If you configured the database to use the Virtual Private Database (VPD) feature, then perform this task to make database queries through a semantic model.

Selecting the **Virtual Private Database** field in the physical database's Advanced properties ensures that the Oracle Analytics query engine protects cache entries for each user. Oracle Analytics query engine matches a list of security-sensitive variables to each prospective cache hit. Cache hits would only occur on cache entries that included and matched all security-sensitive variables.

After you set up row-level security in the database, you can set up object permissions in the semantic model for the presentation layer or other objects. You can also set query limits (governors). See Set Up Presentation Object Permissions and Limit the Number of Rows in a Database Query .

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**, and then double-click the database you want to edit.
4. Click the **Advanced** tab.
5. In Data Source Properties, select **Virtual Private Database**.
6. Click **Save**.

About Data Filters and Row-Level Security

Define data filters on semantic model objects for specific application roles.

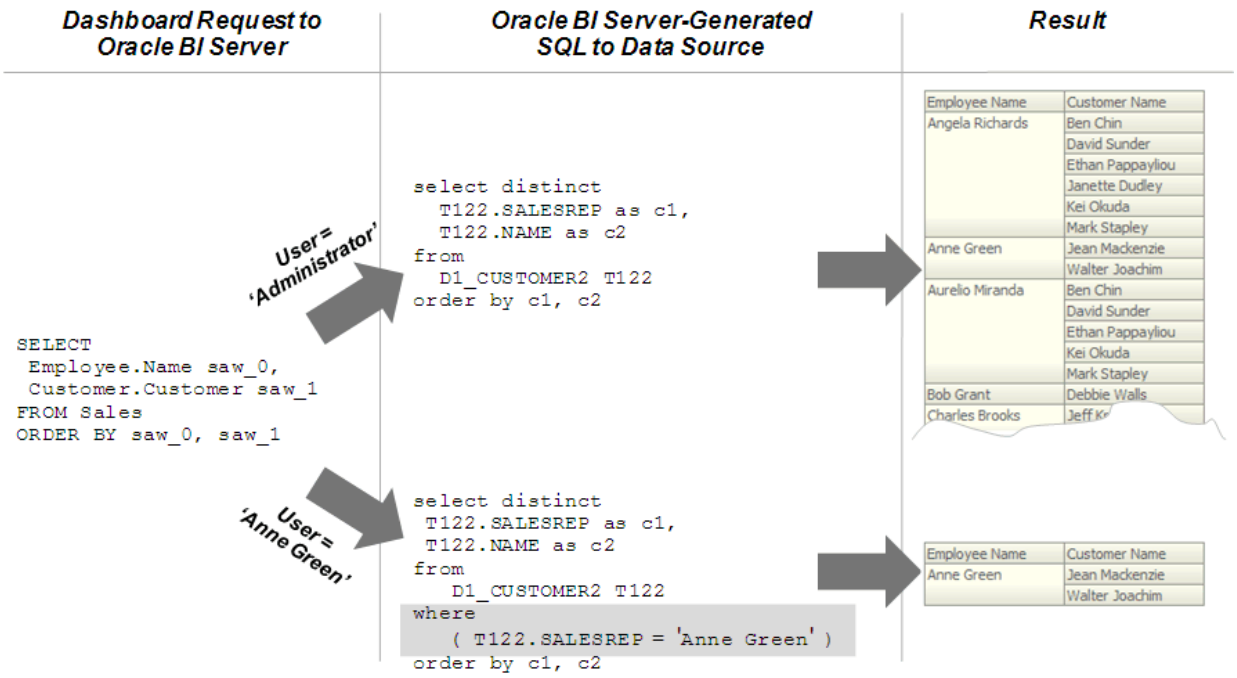
Typically you don't set up data filters if you've implemented row-level security in the database. Row-level security policies are enforced by the database and not by Oracle Analytics.

You can set data filters for objects in the logical layer and the presentation layer. Applying a filter on a logical object impacts all presentation layer objects that use the object. If you set a filter on a presentation layer object, it's applied to the object along with any other filters that are set on the underlying logical objects.

The image shows how data filter rules are enforced in the Oracle Analytics query engine. The security rules are applied to all incoming clients and can't be breached, even when the Logical SQL query is modified.

In this example, a filter has been applied to an application role. When Anne Green, who is a member of that role, sends a request, the return results are limited based on the filter.

Because no filters have been applied to the application roles for the Administrator user, all results are returned. The Oracle Analytics query engine-generated SQL takes into account any data filters that have been defined.



Set Up Data Filters in the Semantic Model

You can assign data filters for specific application roles to enforce row-level security rules in the semantic model.

To create filters, you select objects from subject areas where you want to apply the filters and then you provide the filter expression information for the individual objects. For example, you might want to define a filter like `"Sample Sales"."D2 Market"."M00 Mkt Key" > 5` to restrict results based on a range of values for another column in the table.

You can also use semantic model and session variables in filter definitions.

When a semantic model object such as a logical fact table is accessed by multiple application roles with different levels of access, you can create functional groups to prevent application roles from viewing data restricted from view by that specific application role.

For example, suppose you want your regional sales associates to see the revenue for a quarter in their assigned region, but to avoid exposing sensitive information you want to prevent your regional sales associates to see to total segment sales for all of the regions, In this scenario you create functional groups with different levels of access as appropriate for the specific application role to the filter. See Specify a Functional Group for a Data Filter's Application Role.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.

3. Click **Logical Layer** or **Presentation Layer** and double-click the table where you want to set up data filters.
4. Click the **Data Filters** tab.
5. In **Add**, search for and select the application role that you want to set the data filter for.
6. Click **Open Expression Editor**.
7. In the Expression Editor, define the condition using the semantic model objects and operators.
8. Click **Save**.

About Specifying Functional Groups for Application Roles in Data Filters

When a semantic model object such as a logical fact table is accessed by multiple application roles with different levels of access, you can specify functional groups to prevent application roles from viewing data restricted from view by that specific application role.

When there are no functional groups defined, all the security filters applied to a given table, regardless of the associated role, and are combined using the `OR` operator. Using the `OR` operator works in most cases because a role can view a union of all the rows selected by the security filters. For example, consider the following filters:

Role A is assigned the filter, `Product = 'Camera'`

Role B is assigned the filter, `Product = 'Monitor'`

If an application role is given Role A and Role B, then the role can view data for both the Camera and Monitor products.

When the two security filters from the same table are combined in the query, the filter conditions are combined using the `OR` operator, this is appropriate for most security filters defined on dimension tables, for example:

```
Product = 'Camera' OR Product = 'Monitor'
```

Using functional groups is necessary when securing a single fact table, using data filters from different dimensions.

In this example, a fact table is secured using the following filters:

Role A is assigned the filter, `Product = 'Camera'`

Role B is assigned the filter, `Product = 'Monitor'`

Role C is assigned the filter, `Region = 'Southwest'`

If you don't use functional groups, a user with roles A, B, and C would have all three filter conditions combined in the query using the `OR` operator, for example:

```
(Product = 'Camera' OR Product = 'Monitor' OR Region = 'Southwest')
```

Combining the results of Role A, B, and C doesn't make sense because Product and Region are independent dimensions. Combining data filters from different dimensions using `OR` operator provides the application roles access to more data values than the roles should view.

In this example, the application role can see data for all products within the Southwest region as well as data for all regions within the Camera and Monitor products.

To get the expected behavior, that is allowing the application role to see data only for the Camera and Monitor products within the Southwest region, you need to change the filter to combine the product filters with the region filter using the `AND` operator, for example:

```
(Product = 'Camera' OR Product = 'Monitor') AND (Region = 'Southwest')
```

To achieve this using functional groups, assign the security filters to functional groups as follows:

Role A is assigned the filter, `Product = 'Camera' with functional group "Product"`

Role B is assigned the filter, `Product = 'Monitor' with functional group "Product"`

Role C is assigned the filter, `Region = 'Southwest' with functional group "Region"`

All the filters in the same functional group are combined using the `OR` operator and all sets of filters in different functional groups are combined using the `AND` operator. By choosing the functional groups associated with each security filter, you can control how the filters are combined using the `OR` and `AND` operators.

Specify a Functional Group for a Data Filter's Application Role

You can specify a functional group for an application role with different data access filters on the same semantic model object, usually a logical fact table.

See About Specifying Functional Groups for Application Roles in Data Filters and Set Up Data Filters in the Semantic Model.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Logical Layer** or **Presentation Layer** and double-click the table containing the data filter that you want to specify a functional group for.
4. Click the **Data Filters** tab.
5. Select the filter that you want to specify the functional group for.
6. Click the **Functional Group** field and select an existing group or enter the name of a new group.
7. Click **Save**.

Work With Object Permissions

This topic provides information to help you understand and set up semantic model object permissions.

Topics:

- About Object Permissions
- About Permission Inheritance for Application Roles
- Set Up Presentation Object Permissions

About Permission Inheritance for Application Roles

Application roles can have permissions granted through membership in other application roles.

Permissions granted explicitly to an application role take precedence over any permissions granted through other application roles.

If there are multiple application roles acting on an application role at the same level with conflicting security attributes, then the application role is granted the least restrictive security attribute. Oracle currently requires that the application role with access to an object also have access to the object's container. For example, if ApplicationRole 1 has permission to access Column A, which is part of Table B, then ApplicationRole1 must also have permission to access Table B.

Set Up Presentation Object Permissions

Add application roles and permissions to secure a presentation object.

The permissions that you set for an object are inherited by its child objects. You can change the child object's permissions to override its parent object's permissions. For example, if you set permissions on a subject area, then you can set permissions on a table or column to override the corresponding subject area's permissions.

These are the role permissions that you can set for a presentation object:

- **Read-Write** - Provides both read and write access to the object.
- **Read Only** - Allows only read access to the object.
- **No Access** - Denies all access to the object.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Presentation Layer**.
4. In the Presentation Layer pane locate and double-click the object that you want to assign permissions to.
5. Click the **Permissions** tab.
6. In **Add**, search for and select the application role that you want to set permissions for.
7. Choose a permission for the application role.
8. Click **Save**.

About Object Permissions

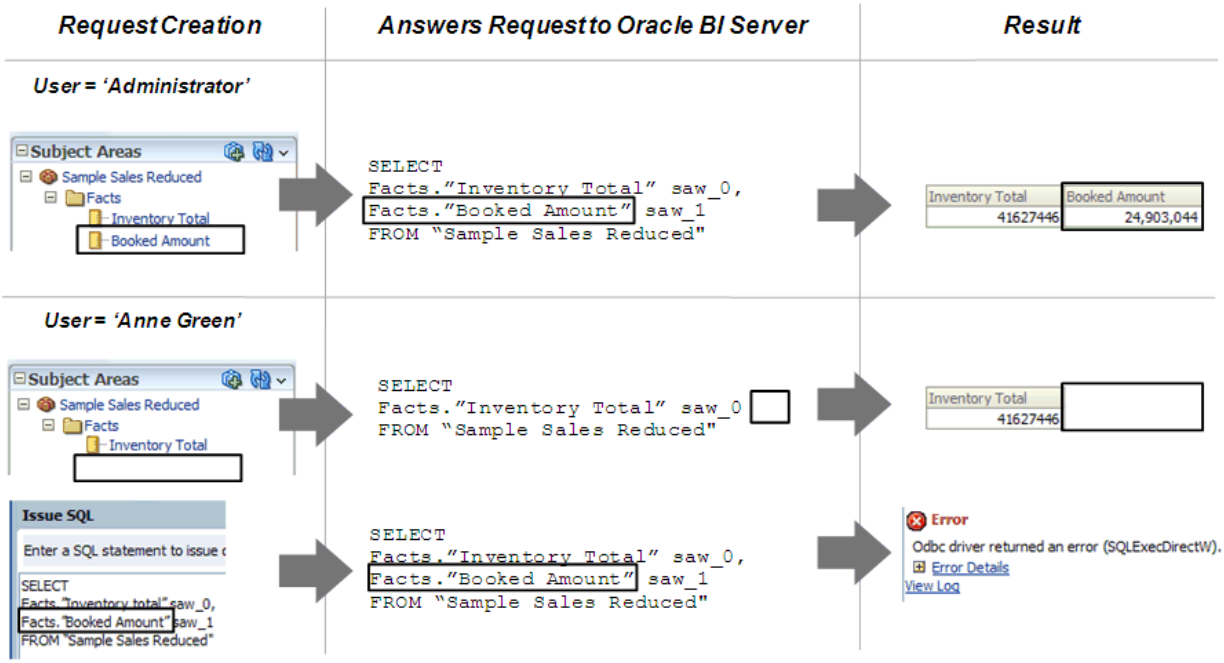
You can use object permissions to configure data filters for objects in the logical layer by using functional groups for multiple application roles.

The object permissions you set determine the security rules that Oracle Analytics applies to client queries. These permissions can't be breached, even when the Logical SQL query is modified.

To set up object permissions:

- Select individual objects such as subject areas, presentation tables, or presentation columns in the presentation layer and assign data access for specific application roles.
- Select individual objects in the logical layer and use data filters to specify functional groups when multiple application roles have different levels of access to the same object.

Set up object permissions for application roles when you want to define data access permissions for a set of objects that are common to users assigned the specific application role.



- If an application role has permissions on an object from multiple sources, for example, explicitly and through one or more additional application roles, the permissions are applied based on the order of precedence.
- If you explicitly deny access to an object that has child objects, application roles are denied access to the child objects. For example, if you explicitly deny access to a particular logical table, you're implicitly denying access to all of the logical columns associated with that table.
- It's best practice to *not* put sensitive data like passwords in session or semantic model variables. Object permissions don't apply to semantic model and session variables, so values in these variables aren't secure. Anyone who knows or can guess the name of the variable can use it in an expression in Answers or in a Logical SQL query.
- The AuthenticatedUser is the default application role associated with new semantic model objects, which means that any authenticated user has read access to new semantic model objects.

  The AuthenticatedUser application role is internal to the semantic model and doesn't display in the semantic modeler user interface. You can override the AuthenticatedUser application role's access at the object level. For example, in a subject area's permissions.

Work With Query Limits

This topic provides information to help you understand and set up semantic model query limits.

Topics:

Limit the Number of Rows in a Database Query

You can control runaway queries for an application role assigned to a physical database by limiting queries to a specific number of rows.

Any query limits you set should exceed the Presentation Server settings for Maximum Number of Rows Processed when Rendering a Table View and Maximum Number of Rows to Download by at least 500 to avoid error messages. When you specify a max row query limit, then those users assigned to the application role may receive Max Row Limit Exceeded messages.

You can override the row limit that you set for an application role. See Override an Application Role's Query Limits.

The options for row limit are:

- **Enable** - Limits the number of rows to the value specified. If the number of rows exceeds the **Max Rows** value, the query is terminated.
- **Disable** - Disables any limits set in the **Max Rows** field.
- **Warn** - Logs queries that exceed the set limit in the Query log. This option doesn't enforce limits.
- **Inherit** - Inherits limits from the parent application role. If there is no row limit to inherit, no limit is enforced.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**.
4. In the Physical Layer pane, locate and double-click the database that you want to assign query limits to.
5. Click the **Query Limits** tab.
6. Locate the role name that you want to limit, double-click its **Max rows** field, and enter the maximum number of rows that members of the application role can retrieve from the source database object.
7. Double-click the **Row Limit** field and select a row limit.
8. Click **Save**.

Limit Database Queries by Maximum Run Time

You can specify the maximum time a query can run on a physical database for a particular application role.

You can override the time queries that you set for an application role. See Override an Application Role's Query Limits.

The options for time limit are:

- **Enable** - This limits the time to the value specified.
- **Disable** - Disables any limits set in the **Max Time** field.
- **Warn** - Logs queries that exceed the set time limit in the Query log. This option doesn't enforce the time limits.
- **Inherit** - Inherits limits from the parent application role. If there is no time limit to inherit, no limit is enforced.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**.
4. In the Physical Layer pane, locate and double-click the database that you want to assign query limits to.
5. Click the **Query Limits** tab.
6. Locate the role name that you want to limit, double-click its **Max time (minutes)** field, and enter the maximum number of minutes rows that want queries to run on each database object.
7. Double-click the **Time Limit** field and select a time limit.
8. Click **Save**.

Allow or Disallow Direct Database Requests

You can specify if you want an application role to be able to run direct database requests.

What you specify in the Query Limits **Execute Direct Database Requests** field overrides what you selected in the **Allow direct database requests by default** field in the physical database's Advanced tab.

The options for the **Execute Direct Database Requests** field are:

- **Allow** - Grants the ability to run direct database requests for this database.
- **Disallow** - Denies the ability to run direct database requests for this database.
- **Inherit** - Inherits limits from the parent application role. If there is no limit to inherit, then direct database requests are allowed or disallowed based on the `Allow direct database requests by default` property for the database.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**.

4. In the Physical Layer pane, locate and double-click the database that you want to assign query limits to.
5. Click the **Query Limits** tab.
6. Locate the role name that you want to specify direct database requests for, go to the **Execute Direct Database Requests** field, click it, and select an option.
7. Click **Save**.

Override an Application Role's Query Limits

You can specify when and how you want to override a database's query limits for a specific application role.

For each application role in a physical database, you can choose to override the row and time limits and the run direct database requests setting. You can also specify how many seconds to limit the application role's logical queries to.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**.
4. In the Physical Layer pane, locate and double-click the database that you want to override assign query limits for.
5. Click the **Query Limits** tab.
6. Locate the click the role name with the query limits that you want to override.
7. Click **Detail View**.
8. In the day and time grid, click to select one or more day and time that you want the override to occur.
9. In the **Row Limit**, **Time Limit**, **Execute Direct Database Requests**, and **Limit logical queries to fields**, specify how to override the application role's query limit settings.
10. Click **Available**.
11. Click **Save**.

Pause an Application Role's Query Limits

You can specify when you want to pause a database's query limits for a specific application role.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Physical Layer**.
4. In the Physical Layer pane, locate and double-click the database that you want to override assign query limits for.
5. Click the **Query Limits** tab.
6. Locate the click the role name with the query limits that you want to pause.
7. Click **Detail View**.

8. In the day and time grid, click to select one or more week days and times that you want to pause the query limits.
9. Click **Unavailable**.
10. Click **Save**.

16

Check Consistency and Deploy a Semantic Model

This chapter contains information to help you understand and how to use the Check Consistency feature, deploy a semantic model, and access a deployed semantic model's log files.

Topics:

Work with Check Consistency

This topic describes how to use the Check Consistency and Advanced Checker features to check a semantic model for errors and warnings.

Topics:

About Check Consistency

Use the Check Consistency feature to validate a semantic model object or the entire semantic model. Check Consistency locates and helps you fix issues that cause query generation to fail at runtime.

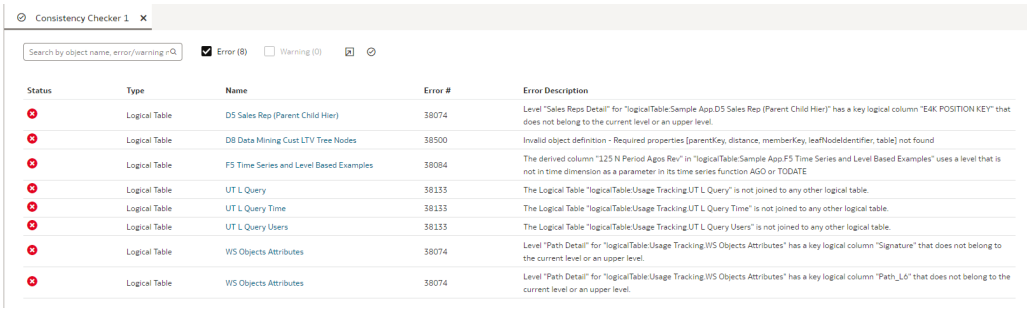
Check Consistency provides the following types of messages:

- **Errors** - Error messages describe errors that you must fix. Use the information in the message to correct the inconsistency, then run the Check Consistency again to confirm that you've fixed the error.
- **Warnings** - These messages indicate conditions that you might need to fix. For example, a warning message about a missing display key in a logical hierarchy level. Other

messages warn of inconsistent values, or feature table changes that don't match the defaults.

For examples of error and warning messages, see Common Consistency Check Messages.

The consistency check report displays in a tab and contains information that you can use to understand, navigate to, and fix the objects listed in the report. Each error or a warning is identified by its name and object type (for example, Logical Table or Initialization Block). You can also search for objects in the list by name, error message number, and so on.



Passing the consistency check doesn't guarantee that a semantic model is constructed correctly, but it does rule out many common problems.

The consistency check doesn't check the validity of objects outside the metadata using the connection. It only checks the consistency of the metadata and not the mapping to the physical objects outside the metadata. If the connection isn't working or objects were deleted in the database, the consistency check doesn't report these errors.

If you use lookup tables to store localized field names with multilingual schemas, the consistency check rules are relaxed for the lookup tables.

Sometimes when you check the semantic model's consistency after an Oracle Analytics upgrade, you might see errors that weren't included in previous consistency checks.

Types of Semantic Model Consistency Checks

There are two consistency checks that you can run: Check Consistency and Advanced Check.

Check Consistency

Check Consistency examines individual semantic objects and their relationships to other objects and finds certain kinds of errors and inconsistencies. You can run Check Consistency on an individual object, a group of selected objects, or the whole semantic model. Run Check Consistency and resolve all errors before you run Advanced Check.

Here are some examples of what running Check Consistency does:

- Finds any logical tables that don't have logical sources configured.
- Finds any logical columns that aren't mapped to physical sources.
- Checks for undefined logical join conditions.
- Determines if any physical tables referenced in a business model aren't joined to the other tables referenced in the business model.

- Checks if each business model has a corresponding subject area.

While you're developing the semantic model, Oracle recommends that you run Check Consistency on objects and the whole model to find and fix any errors. See Check the Consistency of a Semantic Model.

Advanced Check

Run Advanced Check before you deploy the semantic model.

To successfully deploy a semantic model, it must pass the Advanced Check. However, to troubleshoot migration issues, a semantic model doesn't have to pass the Advanced Check. You can convert a semantic model that passed only Check Consistency to a .rpd file, but be aware that it might contain issues that prevent query generation.

The Advanced Check does the following:

- Runs Check Consistency to check the whole model. The consistency check must be error-free before Semantic Modeler can perform the advanced checks.

  If the Advanced Check runs Check Consistency and finds any errors, then only Check Consistency errors are displayed. Advanced Check errors aren't found and displayed until all Check Consistency errors are resolved.
- Converts the semantic model from SMML to a .rpd file and runs advanced checks on the .rpd. If the model contains Check Consistency errors, Advanced Check can't convert it from SMML to a.rpd file and perform advanced checks.
- Looks for scenarios with query generation navigation space errors.

Common Consistency Check Messages

Use this topic to help you understand and fix some of the most common consistency check warnings and errors.

This topic provides a partial list of check consistency messages, and doesn't describe all possible warnings and errors.

Error or Warning Example	Error or Warning	More Information
[14031] The content filter of a source for logical table: FACT_TABLE_NAME references multiple dimensions.	Error	Indicates that the logical table has a logical table source with a WHERE clause filter that references multiple dimensions. A WHERE clause with multiple dimensions is invalid.
[38126] 'Logical Table' '"Technology - WFA"."Fact WFA WO "' has name with leading or trailing space(s).	Error	Identifies an object with leading or trailing spaces in the object name. Leading spaces in object names can cause query and reporting issues.
[38012] Logical column DIM_Start_Date.YEAR_QUARTER_NBR doesn't have a physical data type mapping, nor is it a derived column. [38001] Logical column DIM_Start_Date.YEAR_QUARTER_NBR has no physical data source mapping.	Error	Indicates that logical columns aren't mapped to any logical table source. These mappings are invalid and cause queries to fail. Both of the validation rules relate to the same issue.

Error or Warning Example	Error or Warning	More Information
[39062] Initialization Block 'Authorization' uses Connection Pool '"My_DB". "My_CP"' which is used for report queries. This may impact query performance.	Warning	Indicates that the same connection pool is used for both queries and for initialization blocks. This configuration isn't recommended. To fix this issue, create a dedicated connection pool for initialization blocks. Otherwise, query performance may suffer, or user logins can stop responding if authorization initialization blocks can't run.
[39028] The features in Database 'MyDB' don't match the defaults. This can cause query problems.	Warning	Indicates that some database feature defaults were changed in this release of Oracle Analytics. Unless you've specific customizations to your feature set, it's recommended that you reset your database features to the new defaults.
[39003] Missing functional dependency association for column: DIM_Offer_End_Date.CREATE_DT.	Warning	Indicates that the given column is only mapped to logical table sources that are disabled. This warning prompts you to decide if you want to use the default behavior.
[39059] Logical dimension table MY_DIM has a source MY_DIM_DAILY at level Daily that joins to a higher level fact source MY_FACT_SUM.MTHLY_SUM	Warning	Indicates that the fact logical table source has an aggregate grain set in this dimension, but either no join was found that connects to any logical table source in this dimension or an invalid join was found. This means that either no join exists at all, or it does exist but is potentially invalid because it connects a higher-level fact source to a lower-level dimensional source. Such joins are potentially invalid because if followed, they might lead to double counting in query results. For example, consider Select year, yearlySales. Even if a join exists between monthTable and yearlySales table on yearId, it shouldn't be used because such a join would overstate the results by a factor of 12 (the number of months in each year). If you get a 39059 warning after upgrading, verify that the join is as intended and doesn't result in incorrect double counting. If the join is as intended, then ignore the 39059 warning.
[39055] Fact table "HR"."FACT - HC Budget" isn't joined to tables in logical dimension "HR"."DIM - HR EmployeeDim". This can cause problems when extracting project(s).	Warning	Indicates that there's a physical join between the given fact and dimension sources, but there isn't a corresponding logical join between the fact table and the dimension table.
[39057] There are physical tables mapped in Logical Table Source ""HR"."Dim - Schedule"."SCH_DEFN"" that aren't used in any column mappings or expressions.	Warning	Indicates that the given logical table source contains tables that aren't used in any mapping. This situation shouldn't cause any errors.

Check the Consistency of a Semantic Model

Run Check Consistency to validate all of the semantic model's objects and the relationships between objects. Check Consistency displays a list of syntax or semantic errors and warnings that can cause runtime query generation to fail.

Oracle recommends that you check consistency to find and fix any errors while you develop the semantic model. Oracle also recommends that you run Check Consistency and fix any errors before you run Advanced Check and deploy the model.

You can deploy a model with warnings, but not with errors. You can export a semantic model that contains errors.

For more information about Check Consistency, see About Check Consistency.

You can also run Check Consistency on a specific semantic model object. For example, a database, business model, subject area, or table. See Check Consistency of One or More Semantic Model Objects.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. If you've updated the semantic model, then click **Save** to save it before running Check Consistency.
4. In the toolbar, click the **Check Consistency** icon and then select either **Errors and Warnings** or **Errors Only** to specify the results you want.

   The Check Consistency tab is displayed and contains a list of errors and warnings.
5. From the Check Consistency tab, click an object's link to locate and open it.
6. Fix the object and then click **Save** to save the semantic model.
7. Click the Check Consistency tab and click **Check** to re-run Check Consistency and refresh the Check Consistency tab.

Check Consistency of One or More Semantic Model Objects

Use Check Consistency to validate a specific semantic model object or the objects that you choose. Check Consistency displays a list of syntax or semantic errors and warnings that may cause runtime queries to fail.

For more information about Check Consistency, see About Check Consistency.

For information about running Check Consistency for the whole semantic model, see Check the Consistency of a Semantic Model.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. If you've updated the semantic model, then click **Save** to save it before running Check Consistency.
4. Click the semantic model layer containing the object or objects to check and browse for or use **Search** to find the object or objects that you want to run consistency check on.
5. To check one object, right-click it and select **Check Consistency**.

   To check multiple objects, hold down the **Ctrl** key and click the objects that you want to check, right-click, and select **Check Consistency**

The Check Consistency tab is displayed and contains a list of errors and warnings.

6. From the Check Consistency tab, click the object's link to locate and open it.
7. Fix the issue and then click **Save** to save the semantic model.
8. Click the Check Consistency tab and click **Check** to re-run the consistency check and to refresh the Check Consistency tab.

Run the Advanced Consistency Check Before Deploying a Semantic Model

Run the Advanced Check to confirm that your semantic model is ready for deployment and successful query generation.

The consistency check must be error-free before Semantic Modeler can perform the advanced checks and display any advanced check errors. Running Advanced Check won't show you a list of both consistency check and advanced check errors.

When you run Advanced Check, it first runs Check Consistency to check the whole model. If it finds any consistency errors, it displays them in a tab and you must resolve the errors before you can re-run the Advanced Check.

After Advanced Check runs Check Consistency successfully, it converts the semantic model from SMML to a .rpd file and then checks it for query generation navigation space errors.

After Advanced Check completes, it displays the Advanced Check tab and contains a list of syntax or semantic errors and warnings that it found.

Oracle Analytics runs Advanced Check in the background.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Update the semantic model as needed and click **Save** to save the semantic model before running Advanced Check.
4. In the toolbar, click the down arrow next to the **Check Consistency** icon and then select **Advanced Check**.
5. Go to the Oracle Analytics footer and view the status of the Advanced Check. If the footer displays "Consistency errors found," then click it and select **View Results** to display the Advanced Check tab containing the list of errors.
6. From the Advanced Check tab, click an object's link to locate and open it.
7. Fix the object and then click **Save** to save the semantic model.
8. In the toolbar, click the down arrow next to the **Check Consistency** icon and then select **Advanced Check** to re-run Advanced Check.

Find and View Advanced Check History

You can find and view the results of the five previous advanced checks that you or other users ran on a semantic model.

You can refer to the information in previous Advance Check records to help you understand and troubleshoot model errors and warnings.

See Why Are the Advanced Check Records in a Different Language?

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. In the toolbar, click the down arrow next to the **Check Consistency** icon.
4. Hover over **Advanced Check History**, and click the Advanced Check history record that you want to view.

Why Are the Advanced Check Records in a Different Language?

When you run the advanced consistency check, the language you selected when you logged into Oracle Analytics determines the language used to output and store the records containing the error and warning messages.

For example, if you selected Deutsch when you logged in and then ran the advanced consistency check, Oracle Analytics generates and stores the records in German.

When a user logs in and opens the advanced consistency check records, the records are displayed in the language used to generate them. For example, if the records were generated in German and another user selects English when logging in, the records are displayed in German.

To work around this issue, the user who opens the records must rerun the advanced consistency check so that the records are generated and stored using the language they selected when logged in.

Show or Hide the Advanced Check Warning Message

When you run Advanced Check, by default Oracle Analytics displays a message containing information about the Advanced Check. You can hide this message in your instance, or after you hide the message, you can choose to display it again.

Selecting the **Do not show me this warning message again** field in the Advanced Check dialog turns the **Show Advanced Check warning message** user preference off.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Click **Page Menu** and then click **Preferences**.
4. In the Users Preferences dialog box, scroll to **Consistency Check** and then clear the **Show Advanced Check warning message** checkbox to turn the message off, or click to select the checkbox to turn the message on.
5. Click **Apply**.

Export Consistency Check Results to a CSV File

After you run Check Consistency or Advanced Check, you can export the results to a CSV file.

Exporting consistency check results exports all results and not only the results displayed in the Check Consistency tab. For example, if you ran Check Consistency on a semantic model for both errors and warnings, and in the Check Consistency tab you click **Error** to display only error messages, when you click **Export to CSV**, Oracle Analytics produces a CSV file containing all errors and warnings.

See About Check Consistency.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. If you've updated the semantic model, then click **Save** to save it before running Check Consistency or Advanced Check.
4. Run Check Consistency on the model or model object, or Advanced Check on the model:
5. In the Check Consistency tab or the Advanced Check tab, click **Export to CSV**.

Other Semantic Model Finalization Tasks

After you check the model's consistency and before you deploy the model, you can use nqcmd to test the repository. After you deploy the model, you can enable end user client applications to connect to the deployed model.

For information about how to use nqcmd, see Use nqcmd to Test and Refine the Repository.

For information about creating data source connections for client applications, see Create Data Source Connections to the Oracle BI Server for Client Applications.

Deploy a Semantic Model

Deployment moves the semantic model from the design-time environment to the runtime environment.

The users see the deployed semantic model as subject areas that they query from workbooks, dashboards, and analyses. The Oracle Analytics query engine uses the semantic model's metadata to write physical queries against the data source and to transform and combine the physical result set and perform final calculations.

Before you deploy a semantic model, make sure that you run Check Consistency and Advanced Check and resolve any errors. You can't successfully deploy a semantic model that contains errors. See Check the Consistency of a Semantic Model and Run the Advanced Consistency Check Before Deploying a Semantic Model.

After you begin deploying a semantic model, the deployment's status is displayed at the bottom of the Semantic Modeler editor. You can click the drop-down button and click **Cancel** to cancel the deployment.

If your semantic model deployment fails, then Oracle Analytics creates log files that you can use to understand why the deployment failed. See View a Semantic Model's Logs.

If the subject areas don't display or aren't updated after you successfully deployed the semantic model, then log out and log back into Oracle Analytics. In some cases, an Administrator may need to go to the Console and click the Session and Query Cache's **Reload Files and Metadata** link to reload the subject areas.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. In the toolbar, click **Page Menu** and then click **Deploy**.

17

Manage Semantic Models

This chapter contains information to help you understand how you can manage your semantic models. For example, export and import models, find and download a model's log files, and view information about when a model was deployed or imported.

Topics:

Export a Semantic Model

Export a semantic model from your development environment to a .rpd or .zip file when you want to backup the semantic model, or share it with another developer.

A semantic model must pass Check Consistency before you can export it. A semantic model doesn't need to pass Advanced Check before you can export it. See Check the Consistency of a Semantic Model.

Oracle recommends that you use export and import to share files with other developers, but not for collaborative semantic model development. Exporting and importing a model creates a copy of the model, and developers who import the model aren't working on the same version of the model. For collaborative model development use Git or permissions. See About Collaborative Semantic Model Development.

If you're working in a Windows or Linux environment, you can use the jsontorpd utility to export a semantic model. See Generate an .rpd file from JSON/SMML.

- **Exporting to .zip** - Choose to export to a .zip file if you want to back up the semantic model, or share the semantic model's SMML files and not a compiled .rpd file. Oracle Analytics doesn't generate historic information or log files when you export to a .zip file.

  You can open the .zip file locally and work with the SMML files to perform design time tasks such as search and replace. After you're finished with your tasks, you can export the files to a .zip file and upload it to create a semantic model in your environment.
- **Exporting to .rpd** - Choose to export to an .rpd file to back up the semantic model, or if you want the users you share the file with to only open the semantic model in Semantic Modeler.

When you export to a .rpd file, Oracle Analytics generates log files and stores the export file with the semantic model's history. See Download an Exported .rpd File.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. In the toolbar, click **Page Menu** and then click **Export**.
4. In the **Name** field, enter a name for the file.
5. Select the type of file to export to. If you choose **Repository Document File (.rpd)**, then specify a password that is eight or more characters and includes at least one numeric character and one non-numeric character.
6. Click **Export**.

Generate an .rpd file from JSON/SMML

Use the jsontorpd utility to generate an .rpd file from JSON/SMML. Running this utility is similar to using Semantic Modeler to export a semantic model.

You can run jsontorpd using a JSON folder to generate an equivalent .rpd file for the input JSON.

After you install the Client Tools, the location of the jsontorpd utility is *BI_DOMAIN*/bitools/bin

Syntax

The `jsontorpd` utility takes the following parameters:

```
jsontorpd [-P repository_password] {-R repository_path} {-D
target_json_path}
```

Where the required flags are:

repository_password is the password for the semantic model.

repository_path is the path and name of the semantic model. You must provide the full path names to the input and output repository files if they're located in different directories.

target_json_path is the path and name for the target JSON/SMML. You must provide the full path names to the input and output repository files if they're located in different directories.

Where the optional flags are:

Specify `-O` to generate an output log file at the path you specify.

Specify `-H` or `-?` to display usage information and exit.

Examples

> **Note:**
>
> For all examples, the full path names to the input and output repository files are required if they're located in different directories.

This example generates a semantic model with a password in `/project/target.rpd` based on the input JSON `/project/json`:

```
jsontorpd -P password -R /project/source.rpd -D /project/json
```

This example generates a semantic model with a password in `/project/target.rpd` based on the input JSON `/project/json`. It also generates an additional log file for the conversion:

```
jsontorpd -P password -R /project/source.rpd -D /project/json -O /project/
logs/jsontorpd.log
```

Download an Exported .rpd File

When you export a semantic model to an .rpd file, you can't specify where to save the file. Instead you need to download the exported file.

See Export a Semantic Model.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, locate the semantic model you exported, click **Actions**, and click **Inspect**.
3. Click **Job History** and locate and hover over the export job with the semantic model .rpd file that you want to download.
4. Click **Download RPD**.

Import an .rpd or .zip File Into Your Semantic Model

Another developer can export a semantic model as an .rpd or .zip that you then upload into the semantic model on your environment.

When you import an .rpd or .zip file, you can choose to replace all content or add content and replace matching objects in the semantic model on your environment.

Oracle recommends that you use export and import to share files with other developers, but not for collaborative semantic model development. Exporting and importing a model creates a copy of the model, and developers who import the model aren't working on the same version of the model. For collaborative model development use permissions or Git. See About Collaborative Semantic Model Development.

1. On the Home page, click **Navigator** and then click **Semantic Models**.

2. In the Semantic Models page, click a semantic model to open it.
3. In the semantic model editor, click **Page Menu**, and then select **Import**.
4. In Import From File, choose if you want to replace all content or add content and replace matching objects in the semantic model on your environment.
5. Click **Select** to browse for and select the .zip or .rpd file to import.
6. Click **Import**.
7. Click **Save**.

Import the Deployed Model Into Your Semantic Model

You can import the deployed model into the semantic model on your environment when you need to learn about or troubleshoot the deployed model.

Importing the deployed model into your environment is the only way that you can examine the deployed model. When you import the deployed model, everything in your environment is replaced with the deployed model's contents.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. In the semantic model editor, click **Page Menu**, and then select **Import Deployed Model**.
4. Click **Load**.
5. Click **Save**.

Generate JSON/SMML from an .rpd File

Use the rpdtojson utility to generate JSON/SMML from an .rpd file. Running this utility is similar to using Semantic Modeler to import a semantic model.

The location of the rpdtojson utility is *BI_DOMAIN*/bitools/bin

Syntax

The `rpdtojson` utility takes the following parameters:

```
rpdtojson [-P repository_password] {-R repository_path} {-D
target_json_path} [-8] [-O log_pathname] [-C]
```

Where the required flags are:

repository_password is the password for the semantic model.

repository_path is the path and name of the semantic model. You must provide the full path names to the input and output repository files if they're located in different directories.

target_json_path is the path and name for the target JSON/SMML. You must provide the full path names to the input and output repository files if they're located in different directories.

Where the optional flags are:

Specify `-O` to generate an output log file at the path you specify.

Specify `-8` to assure that the semantic model to JSON conversions supports all UTF-8 encoding.

Specify `-C` to use compatibility mode to generate additional metadata for backward compatibility in legacy semantic models.

Specify `-H` or `-?` to display usage information and exit.

Examples

> **Note:**
>
> For all examples, the full path names to the input and output repository files are required if they're located in different directories.

This example generates JSON/SMML with ASCII encoding in `/project/json` based on the input repository source.rpd:

```
rpdtojson -P password -R /project/source.rpd -D /project/json
```

This example generates JSON/SMML with UTF-8 encoding in `/project/json` based on the input repository source.rpd:

```
rpdtojson -P password -R /project/source.rpd -D /project/json -8
```

This example uses compatability mode to generate JSON/SMML in `/project/json` based on the input repository source.rpd. It also generates an additional log file for the conversion.

```
rpdtojson -P password -R /project/source.rpd -D /project/json -O /project/
logs/rpdtojson.log -C
```

View a Semantic Model's Logs

You can view and copy logs for the semantic models you deployed, imported, loaded, or exported.

Logs aren't available for semantic models exported to ZIP.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, locate the semantic model to download, click **Actions**, and click **Inspect**.
3. Click **Job History** and locate and hover over the job that you want logs for.
4. Click **Logs**.

View a Semantic Model's Job History

You can view job history information for the semantic models you deployed, imported, loaded, or exported.

Job history information includes start time, job type, status, duration, and model size. You can also view and copy log files for the semantic models you deployed, imported, loaded, or exported. See View a Semantic Model's Logs.

Job history information isn't available for semantic models exported to ZIP.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, locate the semantic model that you want history information for, click **Actions**, and click **Inspect**.
3. Click **Job History** and locate and hover over the job that you want history information for.

Generate Indexes for a Semantic Model

The **Generating Indexes** option is for use with support-related issues. Use this option only when the Oracle Support team directs you to.

Part III
Reference

This part provides reference information.

Chapters:

18

Design Tips

This chapter contains reference information to help you design semantic models.

Topics:

Business Model Design

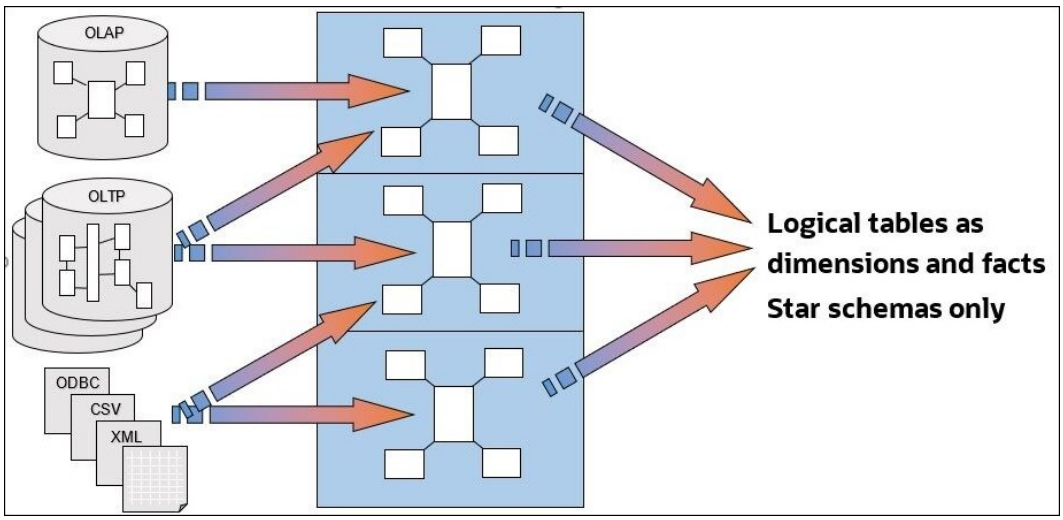
In Oracle Analytics, you can configure the logical layer in your semantic model in many different ways. Oracle recommends that you follow the best practices described here, so you can avoid several errors at runtime and significantly decrease your maintenance workload.

Best Practice

- **Use star schemas in business models**

  Data structure in the physical layer may come in many different forms. Having star schemas in the physical layer is useful for performance but it isn't mandatory. However, no matter the structure of the physical layer, your business model should *always* be star schemas.

  Each logical table can include multiple physical tables, either in the same logical table source or split across multiple logical table sources.



- **Use a separate dimension logical table for each dimension**
- **Don't combine or merge dimensions into one logical table**

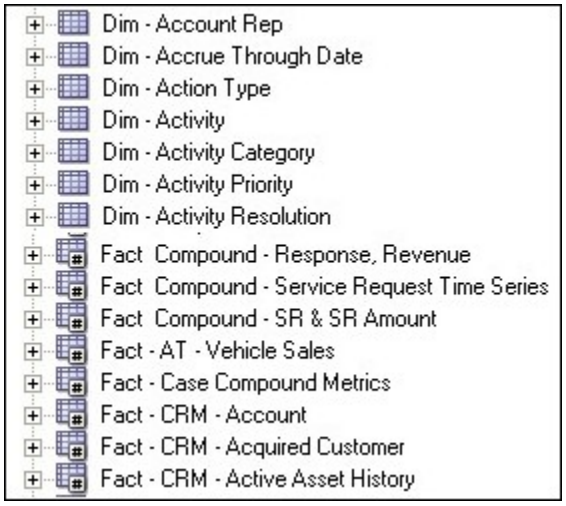
- **Use a separate fact logical table for each fact**

  The same goes for facts, you don't want to end up with a single fact logical table called "Facts – Stuff"!

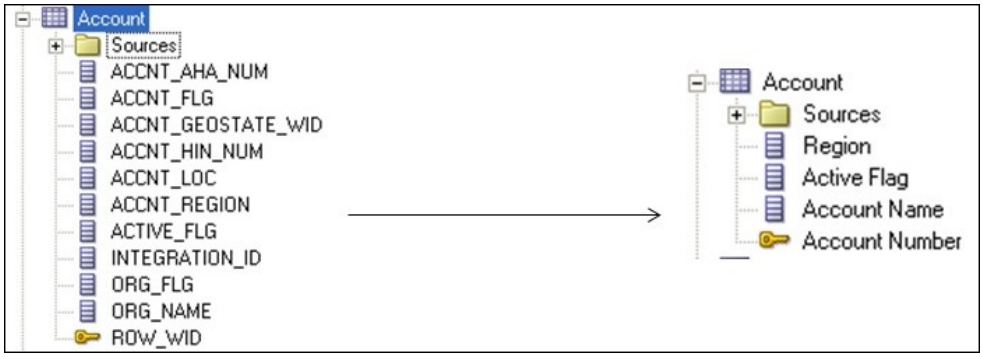
- **Use a separate logical table for "Compound" facts**

  A compound fact is the place where you put derived expressions that combine metrics from multiple fact tables. For example, if you have Fact Order and Fact Opportunity, include a calculation with the formula "# Opportunities/ # Orders" in separate logical table Fact Compound Opportunity & Order.

- **Prefix logical table names with Dim – or Fact – or Fact Compound –**



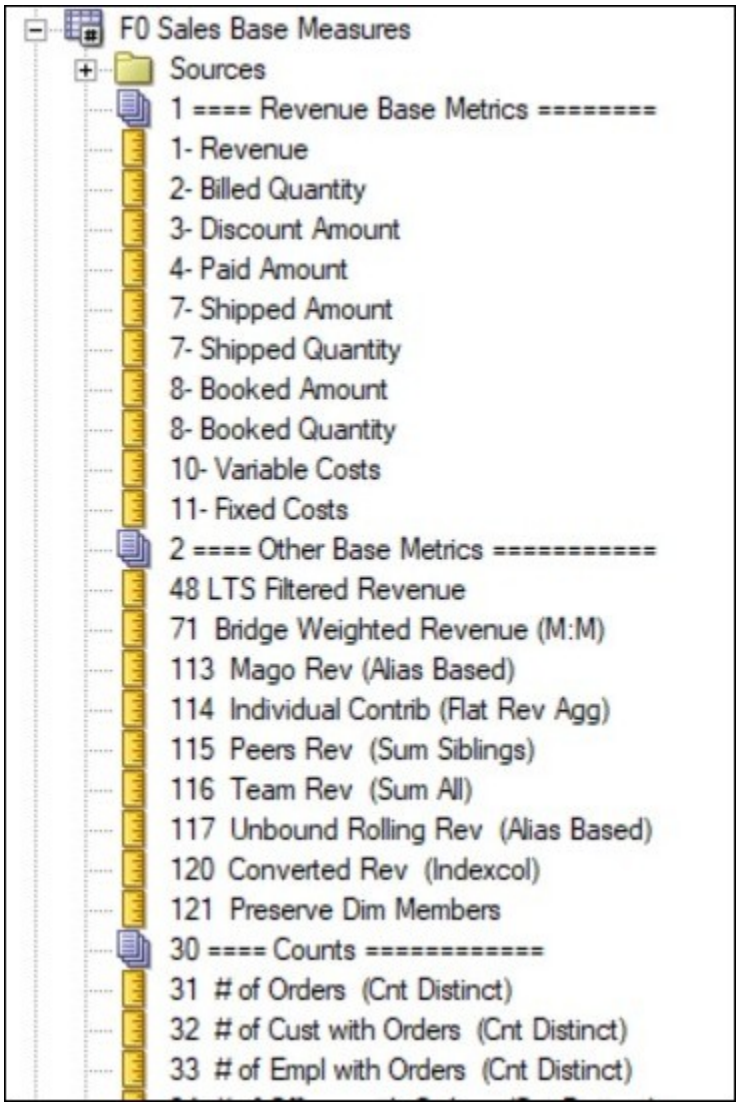
- **Assign unique business columns as dimension primary keys wherever possible**
- **Rename logical columns to use presentation names**
- **Keep only used columns in the logical layer**



- **Don't assign logical primary keys on logical fact tables**

  Logical primary keys are needed only on dimension tables.

- **Create "dummy" measures to separate facts into various groups, if required**



- **Ensure almost every fact logical column has an aggregation rule set.**

Time Dimension Design

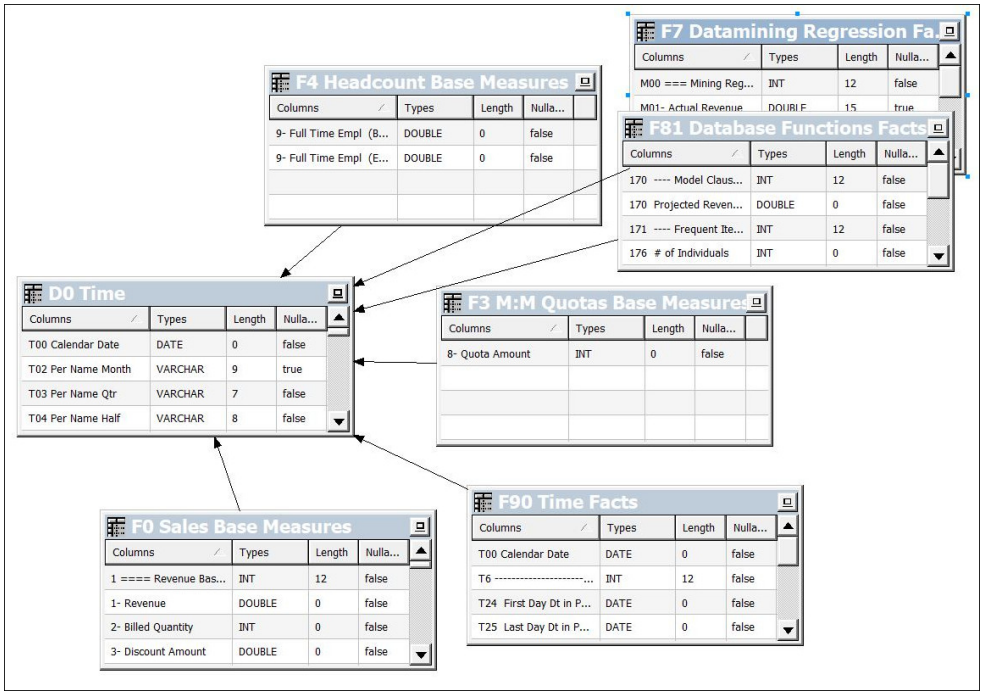
In Oracle Analytics, fact tables often include many dates, and therefore many potential time dimensions. Oracle recommends that you follow the most efficient way to handle and configure time dimensions in your semantic model as described here.

Best Practices

- **Create a single generic time dimension for each fact table**

  For each fact table, identify one date which is the used the most with this fact. To help you identify the best date, ask yourself questions like, “if I select this amount for June, what does ‘June’ mean?”.

  After you’ve identified a specific date for each fact table, use that date to join each fact to a generic time dimension. A generic time dimension is required for reports that includes multiple facts and is much easier for end-users than creating a separate time dimension for each fact table.

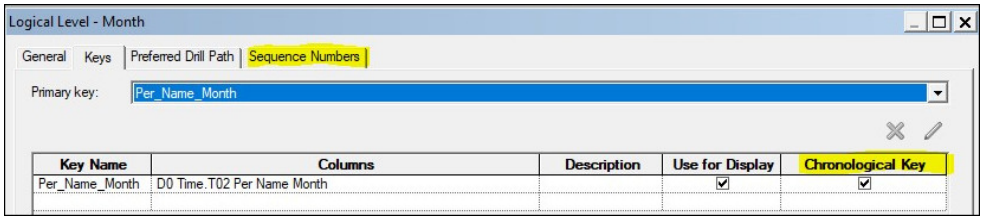


- **Only create secondary time dimensions if needed**

  A time dimension is only useful for a date if you want to simplify user selection at the date level (such as the year or quarter) or if you need to drill down the date hierarchy at runtime. For both these cases, we recommend that you create a secondary time dimension with joins to only the specific fact tables. Otherwise, displaying the date itself as a single attribute in the Presentation layer is often enough.

- **Configure your time dimension for time series**

  Time series functions like `Ago` or `ToDate` are often used to easily calculate metrics such as Year-Ago or YearToDate. These time series functions are available only if you configure chronological keys for the corresponding time dimension. These chronological keys must be unique at each level of the dimension hierarchy.



You can use time series functions after chronological keys are defined. However, these time series functions can have an impact on performance. One way you can minimize the performance impact is to define sequence numbers on the time dimension. Sequence numbers are optional and used only for performance reasons.

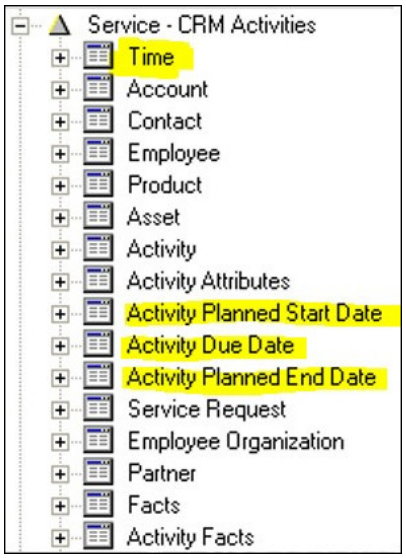
There are two types of sequence:

  - **Absolute:** Integer chronological keys with values that increment by 1 and range from N to M. For example, a 4-digit year.
  - **Relative:** Integer values with values that increment by 1 and range from 1 to N. These values represent the chronological order within a higher level. For example, month number (1 – 12).

    As absolute and relative sequences cover different use cases, you can define both types of sequence. Logical columns must be at the appropriate level in the hierarchy to be available for the sequence definition.

- **In the Presentation layer, always display the generic time dimension as the first folder**

  You can place secondary time dimensions as folders or sub-folders next to their corresponding dimension attributes.



Physical Table Alias

In Oracle Analytics, you can create aliases for tables in the physical layer of your semantic model. Table aliases are useful when a single table has several different roles.

Best Practices

A single table often has multiple roles. Sometimes a table is used as a dimension, sometimes as a fact table, sometimes to extend another dimension to retrieve a specific attribute, and sometimes as a helper table to join two other tables together.

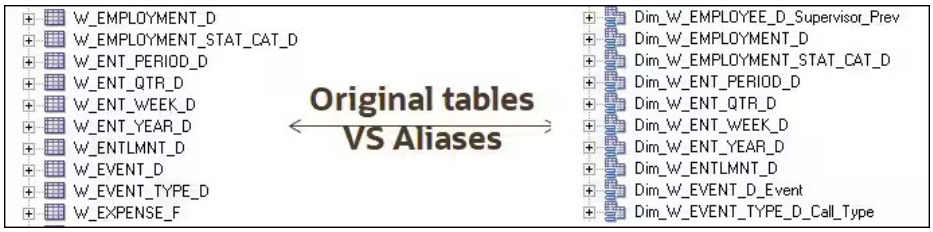
Often, each role comes with a different set of physical joins. If you configure all the joins on a single instance of the table, it results in data integrity issues. You can avoid such issues, if you use table aliases and follow some basic rules.

- **Use a consistent naming convention for aliases**

  The alias name should include both the name of the original table, and some indication of the role of the alias. This way on first sight, developers immediately know which table is being used and understands the purpose of the alias.

- **Don’t define any physical joins on the original table**

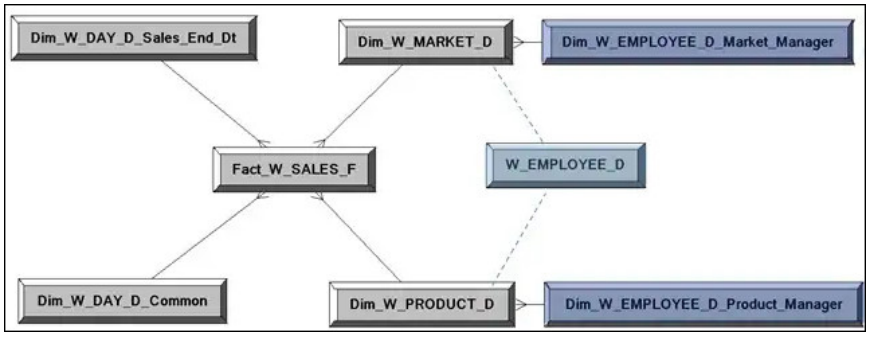
  Start by creating an alias. Each physical table should always have at least one alias. Only the alias will be used, not the original table. This way if you need new instances of the same table for other roles in the future, it’s easy to identify the differences and roles of each alias.



- **Create additional aliases when you need different physical joins depending on the context in which a table is used**

  Here are two common examples

  Example 1



  Example 1 shows an implementation of the Employee table. Table `W_MARKET_D` includes the key of the employee who is the `Market Manager`. Table `W_PRODUCT_D` includes the key of the employee who is the `Product Manager`. Without any alias, table `W_EMPLOYEE_D` joins to both `W_MARKET_D` and `W_PRODUCT_D`. If you create a report that selects the name of both the `Market Manager` and `Product Manager`, the `WHERE` clause generated in the physical SQL would include the following statements: `W_MARKET_D.EMP_ID=W_EMPLOYEE_D.ID` and `W_PRODUT_D.EMP_ID=W_EMPLOYEE_D.ID`

  This means that the ID of the employee must at the same time equal the `Market Manager` ID and `Product Manager` ID. This isn’t possible because these managers are two different employees, so the query doesn’t return any records.

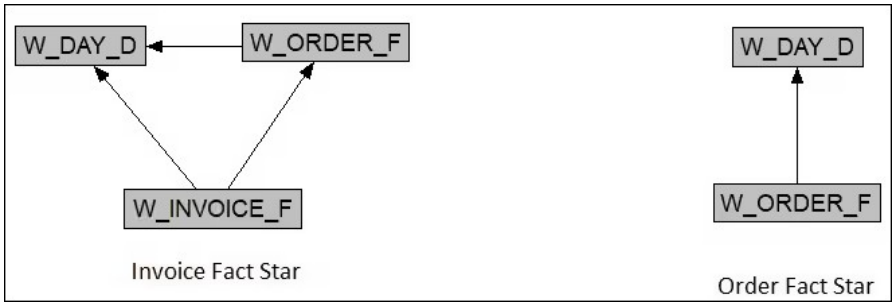
  Instead, as described on the diagram above, the solution is to have two aliases of the `Employee` table. One alias is joined with the `Market` table and the other is joined to the `Product` table. These two aliases are considered as if they are two different tables, completely independent from each other. By using two aliases, there is no conflict between the two joins.

  Example 2

  Example 2 shows three tables. Table `W_ORDER_F` is used as a fact table for order metrics, a dimension for order attributes, and it includes the `Order Date`. There is also calendar table `W_DAY_D`, and invoice table `W_INVOICE_F` that includes `Order ID` and `Invoice Date`. The invoice table is joined to the order table to retrieve order attributes as a dimension for `Invoice Fact` metrics. Note that Oracle

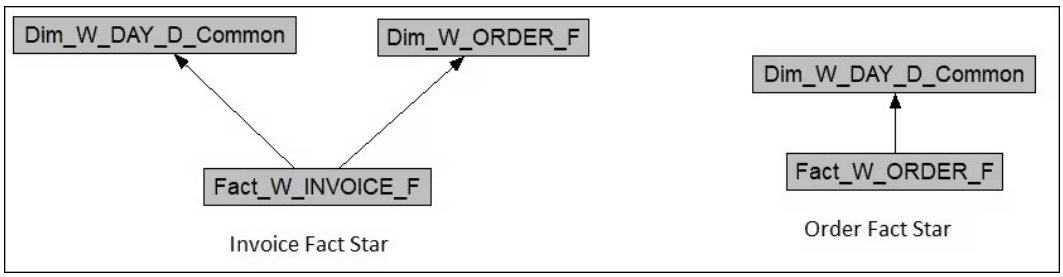
Analytics generates separate sub-queries for each fact table. Therefore, we must consider `Order Fact Star` and `Invoice Fact Star` separately, as shown in the diagram.

Without any aliases, the diagrams look like this:



This configuration causes similar data integrity issues to the Employee example, that is, the `Order Date` is not equal to `Invoice Date` but they are both joined to the same date column on the calendar table.

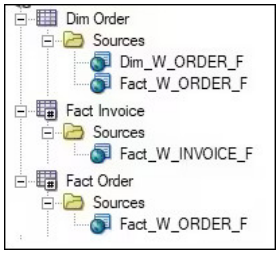
The solution is to create two aliases for the Order table, one alias for the fact and the second alias for the dimension. With aliases, the diagrams look like this:



Now there's no conflict between the joins, as the dimension alias of the Order table is not joined to the calendar dimension.

Also note that there's no need to join `Fact_W_ORDER_F` with `Dim_W_ORDER_D`. Except for rare specific situations, you should never join two aliases of the same table together. While doing so doesn't impact data integrity, it does impact performance and it's useless.

Instead, create two logical table sources in the `Order Dimension` in the business model as shown here. Use one logical table source for the `Invoice Fact Star` and the other for the `Order Fact Star`.



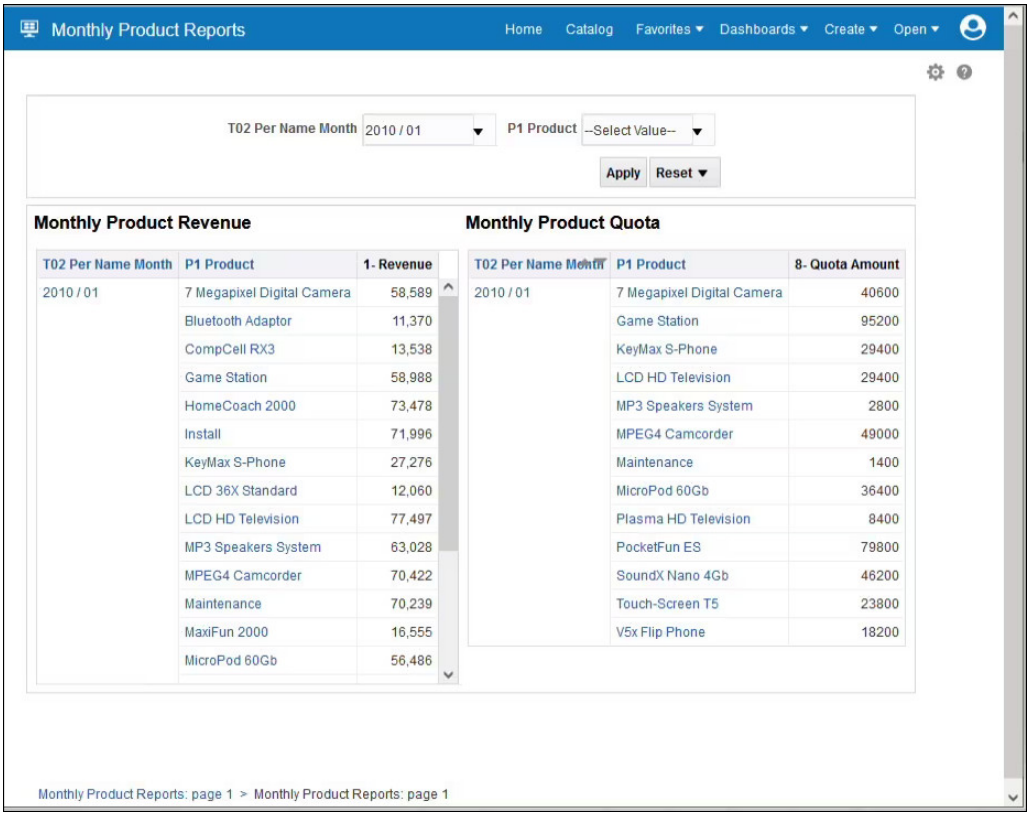
Summary

- Always create at least one alias for each physical table.
- If needed, create additional aliases based on the different roles of the table in your model and the different types of joins you require.
- Although there are exceptions, in most cases you shouldn't join two aliases of the same table together.

Implicit Facts in Subject Areas

You can set an implicit fact in a subject area so that Oracle Analytics always uses a predictable fact source when a query contains only dimensions. This way, you can ensure that query results always match your expectations.

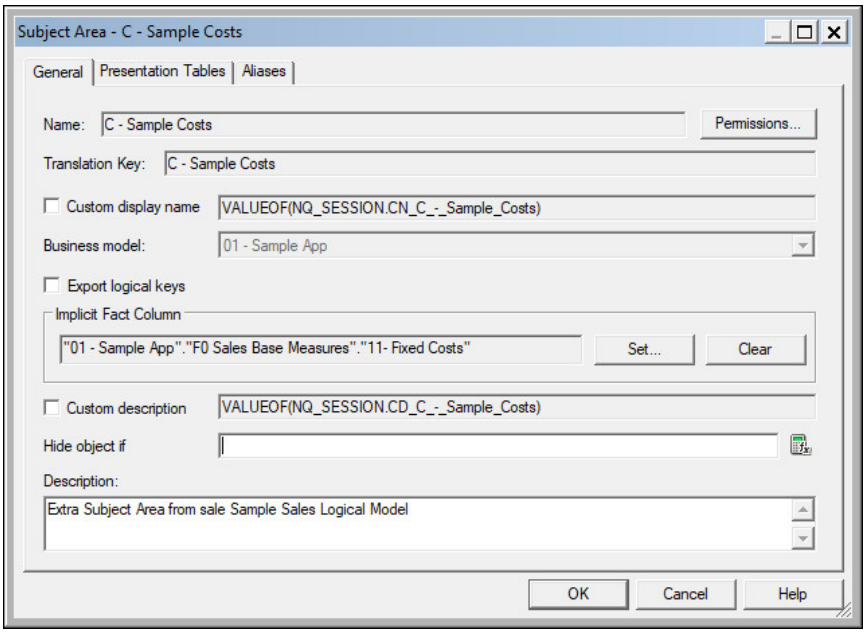
Different fact tables within the same semantic model often result in a different set of elements for the same query filters. For example, the list of products for Revenue or Quota Amount for the month of January.



The values returned from the query `Select Month, Product from subject area A where month = 'Jan'` depends on which fact table is used to run the query.

Most queries contain a mixture of facts and dimensions, so the sources used are predictable and the results match expectations. When a query contains only dimensions, Oracle Analytics must choose a fact table using the best information available, and this might yield results that don't match your expectations.

In this scenario, there's an option to assign an implicit fact for your subject area. This implicit fact is automatically included for any query that only includes dimensions from that subject area. This ensures that Oracle Analytics always uses a predictable fact source, and query results match your expectations.



For example, this session log shows the implicit fact added to the logical query.

```
SELECT
 0 s_0,
 "C - Sample Costs"."Products"."P1 Product" s_1,
 "C - Sample Costs"."Time"."T02 Per Name Month" s_2,
 DESCRIPTOR_IDOF("C - Sample Costs"."Products"."P1 Product") s_3
FROM "C - Sample Costs"
WHERE("Time"."T02 Per Name Month" = '2010 / 01')
ORDER BY
 3 ASC NULLS LAST,
 2 ASC NULLS LAST,
 4 ASC NULLS LASTFETCH FIRST 500001 ROWS ONLY
-------------------- Logical Request (before navigation): [[
RqList [1,2,3]
 0 as c1 GB,
 D1 Products (Level Based Hier).P1 Product as c2 GB,
 D0 Time.T02 Per Name Month as c3 GB,
 D1 Products (Level Based Hier).P0 Product Number as c4 GB,
 11- Fixed Costs:[DAggr(F0 Sales Base Measures.11- Fixed Costs by [ D1
Products (Level Based Hier).P0 Product Number, D1 Products (Level Based
Hier).P1 Product, D0 Time.T02 Per Name Month] )] as c5 GB
DetailFilter: D0 Time.T02 Per Name Month = '2010 / 01'
OrderBy: c3 asc NULLS LAST, c2 asc NULLS LAST, c4 asc NULLS LAST
```

Dimensional Hierarchies, Level Keys and Content Levels

In Oracle Analytics, dimensional hierarchies, level keys, and content levels together form the basis of navigation. This topic describes how you can set up dimensional hierarchies to enhance the capabilities of Oracle Analytics.

Level keys are used to define the levels in a dimensional hierarchy. In turn, these levels are used to set content levels or the level of aggregation of a logical table source. Oracle

Analytics uses content levels to navigate to the most optimized logical table source for a given query.

Dimensional hierarchies are also required to create level-based measures and to set up drilling for analyses.

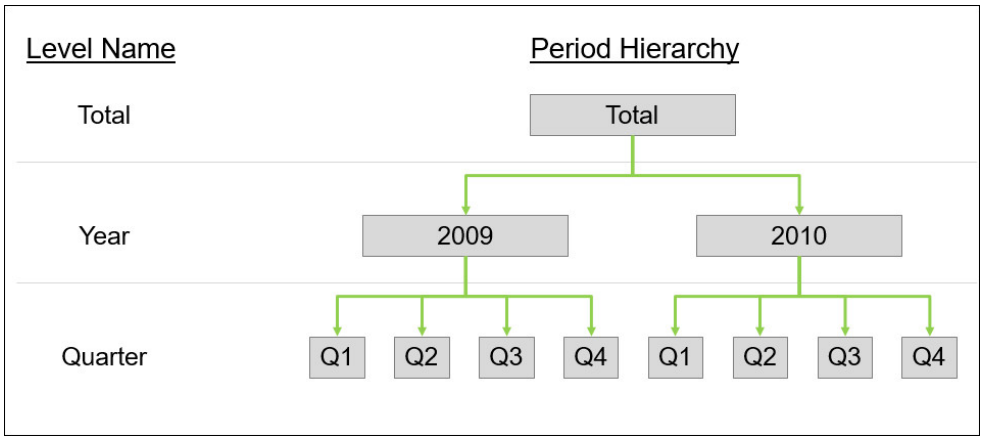
Dimensional Hierarchies

Always create dimensional hierarchies, even when there's only one level. We recommend that you do this for many reasons:

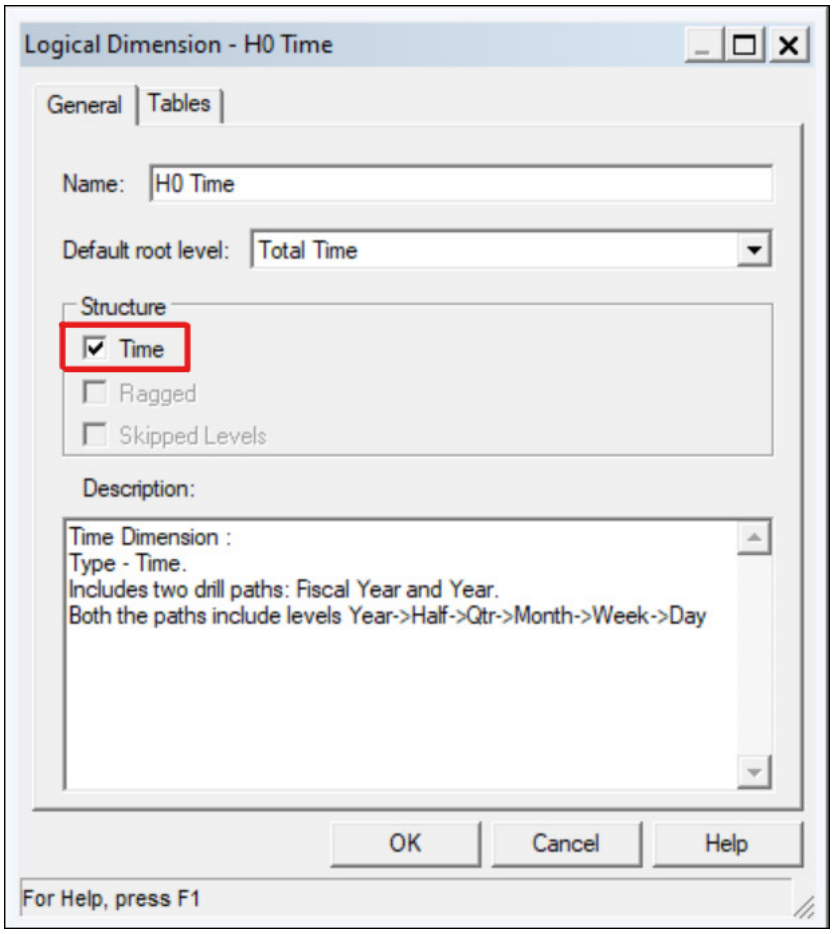
- Oracle Analytics uses dimensional hierarchies to select the most optimized logical table source by way of content levels.
- Dimensional hierarchies are required to drill up and down between levels. Sometimes drilling is intuitive. For example, if you're analyzing a brand, you'll probably want to drill down to its corresponding Universal Product Codes (UPC). Other types of drill-downs might not be obvious but still useful. For example, you might want to drill down from a contact type to a contact name.
- Dimensional hierarchies are useful when Oracle Analytics joins two result sets. For example, if you combine two fact tables in the same report.
- Dimensional hierarchies are required to create level-based measures.
- Time dimensions are required in some time series calculations. For example, where the calculation is based on a specific level, such as the year.
- When you define dimensional hierarchies and content levels for a logical table source, it improves the capabilities of the consistency checker to identify issues with the semantic model.

Types of Dimensional Hierarchies

- A *balanced level-based hierarchy* is the most common type of hierarchy used in Oracle Analytics. In all level-based hierarchies, the detail levels roll up into higher levels. In a balanced level-based hierarchy, all members of the hierarchy have ancestors at all levels as shown here.

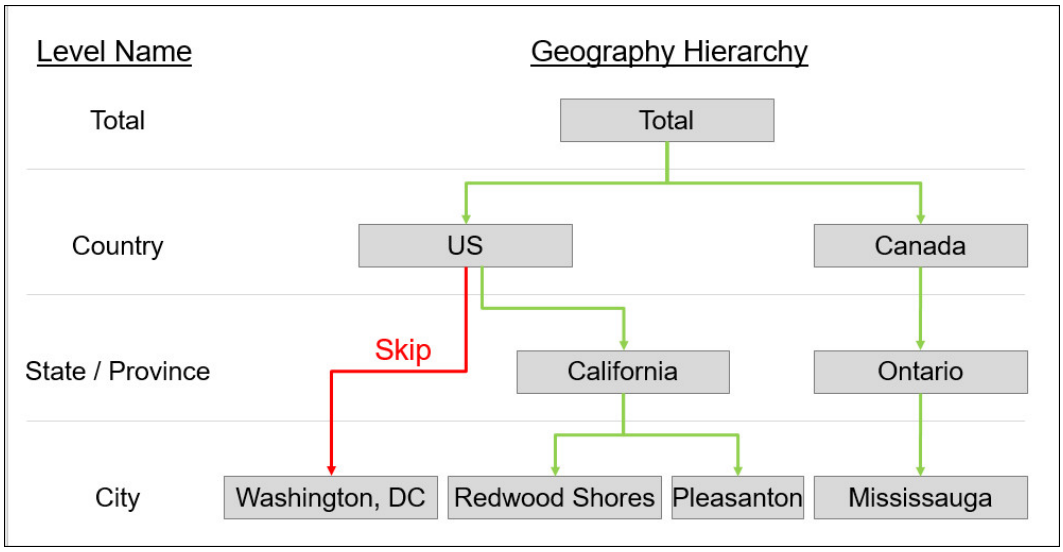


- A *time dimension* is a special level-based hierarchy that is used specifically for time-based hierarchies. A time dimension is required if you want to use time series calculations such as `AGO` and `TODATE`. To define a time dimension, you select **Time** in the properties of the dimensional hierarchy.



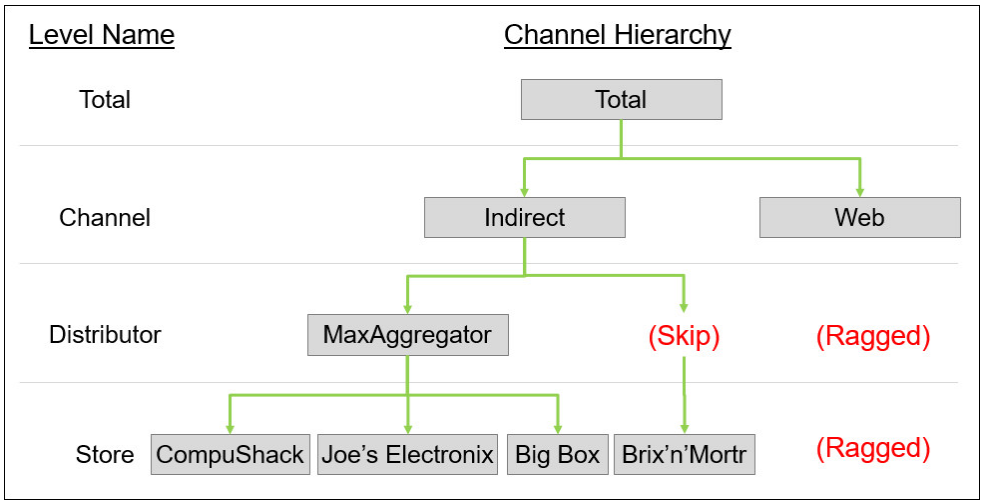
- A *skipped-level hierarchy* is a special level-based hierarchy where not all members of the hierarchy have ancestors at all levels. To define a skipped-level hierarchy, you select **Skipped Levels** in the properties of the hierarchy.

  This example shows a skipped-level hierarchy where Washington DC doesn't belong to a state, so the state/province level is skipped.

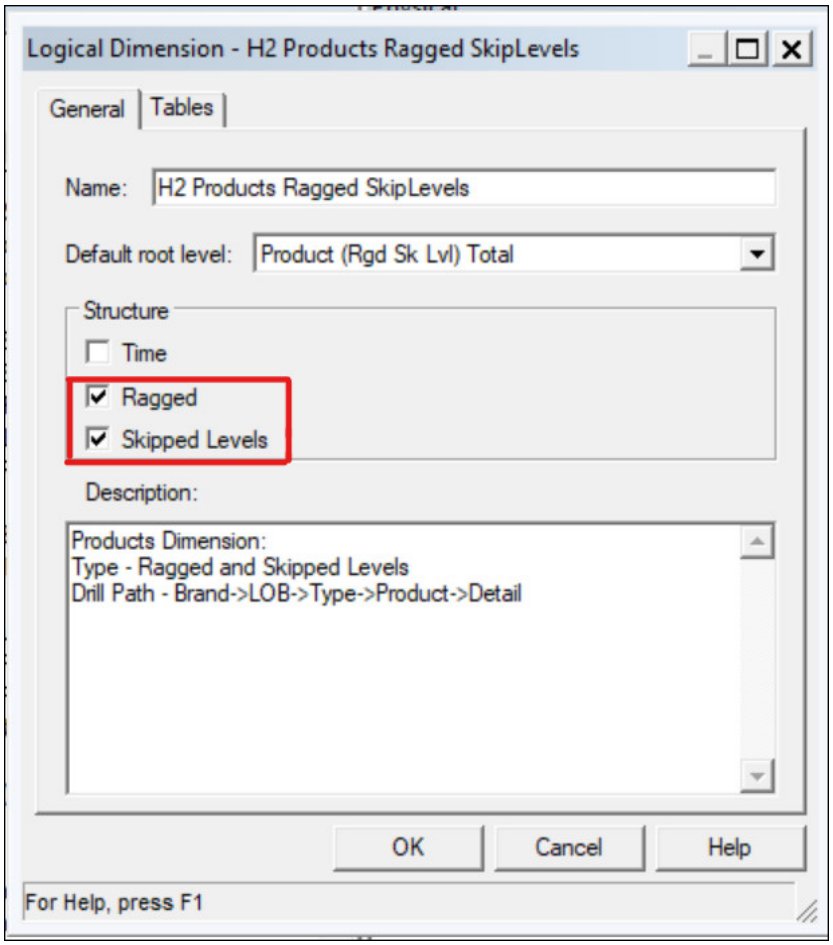


- A *ragged or unbalanced hierarchy* is another special level-based hierarchy where not all the data is present at all levels of the hierarchy. To define a ragged hierarchy, you select **Ragged** in the properties of the hierarchy.

  This example shows a ragged and skipped-level hierarchy where the Distributor and Store levels are missing from the Web branch of the hierarchy.



Here, you select both **Ragged** and **Skipped Levels** in the properties of the hierarchy.

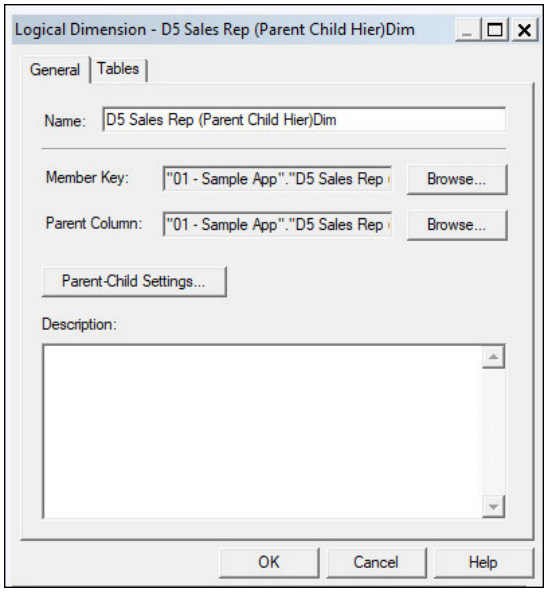


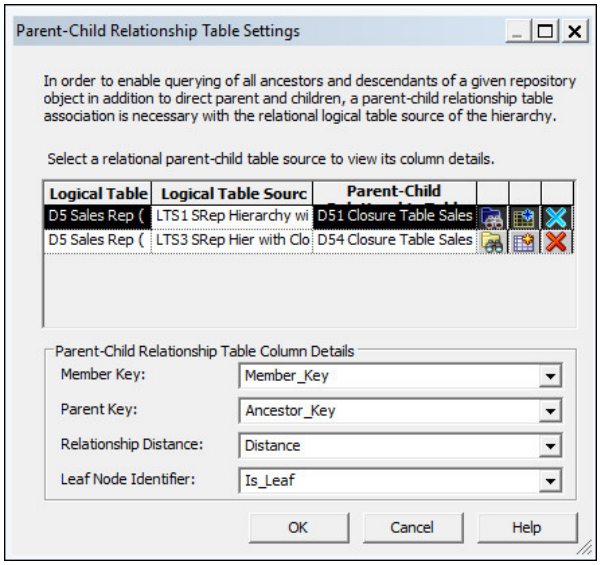
- A *parent-child hierarchy* is a type of hierarchy often associated with an organization. For example, employees rolling up to a manager. In a parent-child hierarchy, each child member rolls up to a single parent member. At the lowest level, each member has no child members. At the highest level, there is a single parent with no further parent levels. In between, each member is both a parent and a child.

  Parent-child hierarchies are based on a specialized parent-child relationship table made up of four columns:

  – Member
  – Ancestor or ancestors of a member
  – Number of levels between the member and the ancestor
  – If the member is a leaf member, that is, at the lowest level

  To define a parent-child hierarchy, you select **Dimension with Parent-Child Hierarchy** when you create a new hierarchy, and then select **Parent-Child Settings** to set up the parent-child hierarchy as shown here.

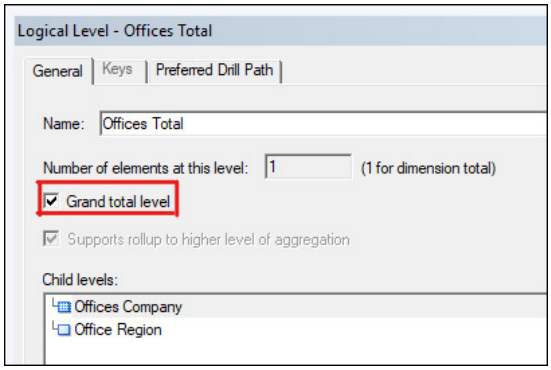




Rules When You Create a Dimensional Hierarchy

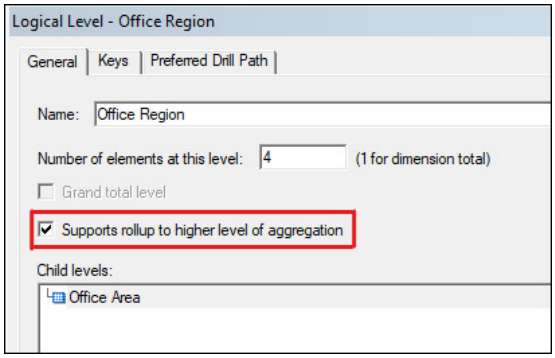
- A dimensional hierarchy can contain only one *grand total*.
- If a dimensional hierarchy has multiple branches, all branches typically have a common beginning point and a common end point.
- To define the grand total-level, you must select **Grand total level** in the properties for the level.

  This example shows **Grand total level** selected and **Number of elements at this level** set to 1.

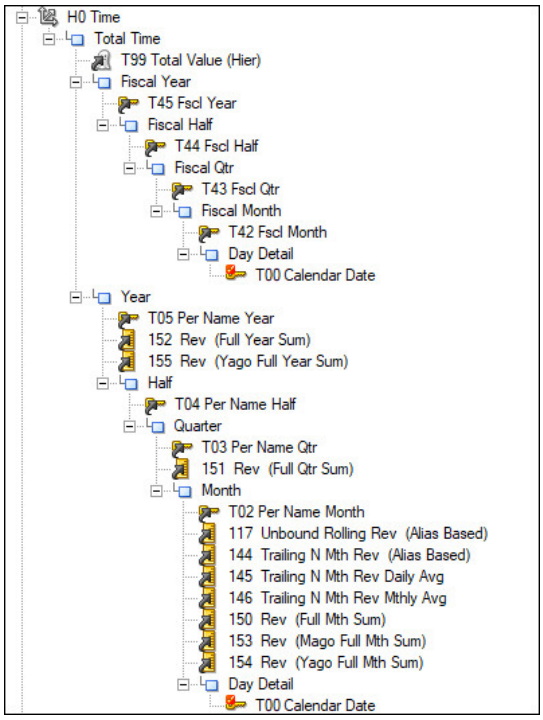


- To define a non grand total levels, you must select **Supports rollup to higher level of aggregation** in the properties for the level. After you define a grand total level, Oracle Analytics automatically sets **Supports rollup to higher level of aggregation** for all other levels in the hierarchy.

  For all non grand total levels, **Number of elements at this level** is always higher than 1.

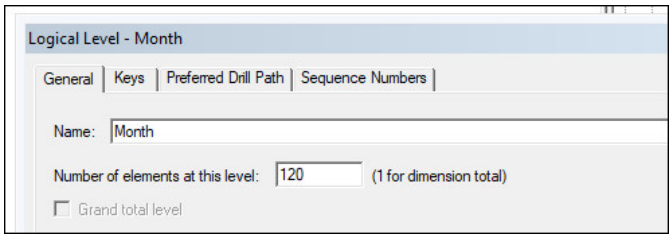


This example shows a dimensional hierarchy with multiple branches starting and ending at a common point. The shared starting point is the grand total level and the shared end point is the detail level.



- When you define a dimensional hierarchy, always specify the number of elements per level. Oracle Analytics uses the number of elements to identify aggregate tables and mini dimensions. This number doesn't need to be exact, a rough estimate is sufficient. In navigation, a cross product is calculated across all the content levels in a logical table source. This is used as a tie-breaker when navigating between otherwise equal sources.
- The number of elements per level is hierarchical with the lowest number of elements at the top (1 for a grand total). Higher levels in a dimensional hierarchy should have fewer elements than lower levels. Note that the consistency checker warns you if a parent level has a greater number of elements than a child level.

  This example shows a logical level **Month** with **120** elements defined.



- After you create a dimensional hierarchy for a logical table, all columns in that table are part of the hierarchy. By default, a column not explicitly associated with a higher level is considered to be part of the lowest or detail level.

Level Keys

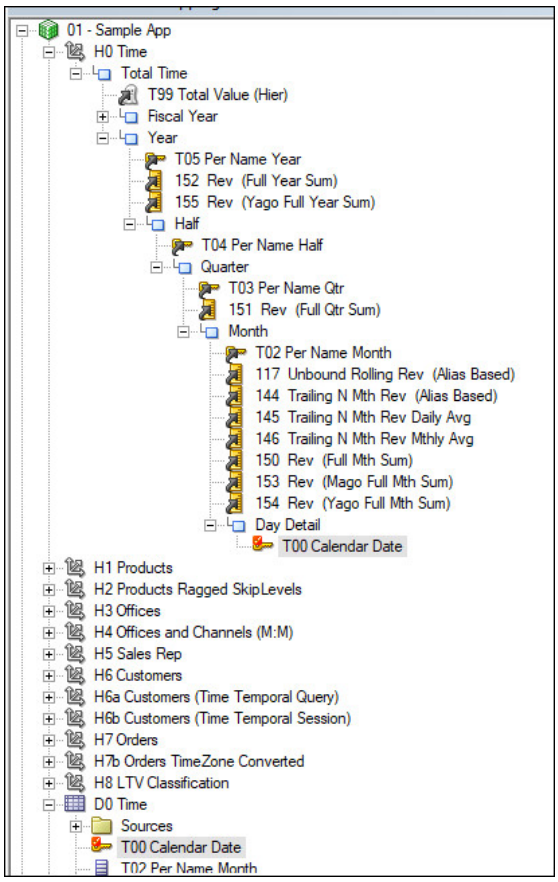
You use level keys to identify a given level.

- The primary key of each level must be unique.
- If a single column used as a primary key of a level is not unique, you must combine it with additional columns to form a unique, composite level key. For example, consider the case where *month number* or *month name* is used for a level key. The month number for October is 10 but both October and 10 are not unique, as every year has a month

number 10 and a month named October. To form a unique level key, you must combine month number or month name with year to form a composite level key. In this example, the composite level key is month number 10 and year 2021 or month name October and year 2021.

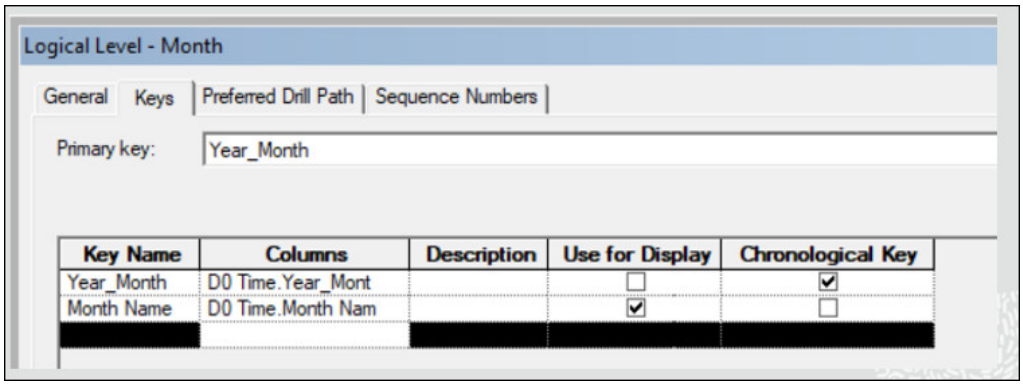
- The primary key of the lowest or detail level of a dimensional hierarchy must match the primary key of the logical dimension table that the hierarchy is based on.

  This example shows a logical dimension table and dimensional hierarchy with common primary keys.



- Grand total levels don't have a level key associated with them.
- If a column is a primary level key or part of a primary level key, you must assign that column to that level. If a column is a level key in a parent level and part of a composite level key in a child level, you assign that column to the parent level. Using the earlier example, month number and month name should be assigned to the month level and year to the year level.
- Display keys are a level key with **Use for Display** selected. A display key is the column that is shown in an analysis when you drill down on an object.
- If a dimensional hierarchy is for a time dimension, at least one of the levels must have a chronological key which specifies the sort order of the periods from the oldest to the newest. Often the primary key of the detail level is also the chronological key.

  This example shows a time dimension level key with the same primary key and chronological key.

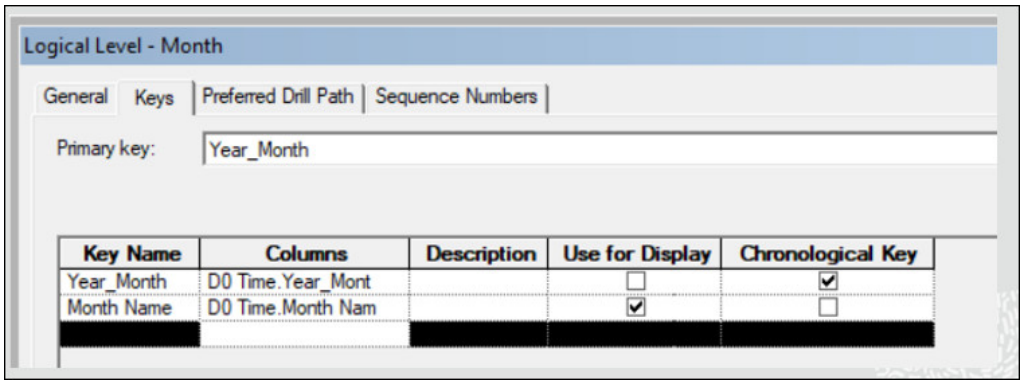


Content Levels

You use content levels to define the level of aggregation of a logical table source in both facts and dimensions.

- Content levels allow Oracle Analytics to select the most optimized logical table source for a query.
- Content levels help the consistency checker find issues with your semantic model configuration and this can prevent runtime errors.
- If you specify content levels for a fact logical table source, you must specify content levels for all the dimensions that join to that logical fact table. If you join a dimension table with no content level set to a fact table but content levels exist for other dimensions, Oracle Analytics doesn't perform a join between that dimension table and the fact table. In this case, Oracle Analytics assumes that the dimension table doesn't join to the fact table. Queries involving dimensions with no content levels specified but which do join to a fact source return the error *Unable to navigate requested expression*.

  This example shows a fact logical table source showing level of aggregation or content levels for a fact table. Note that this fact logical table source joins to only 4 of 12 attribute tables.



19

Miscellaneous Reference Information

This chapter contains reference information to help you understand and use Semantic Modeler.

Topics:

Keyboard Shortcuts for Semantic Modeler

You can use keyboard shortcuts to navigate and to perform actions in Semantic Modeler.

Using Keyboard Shortcuts in Semantic Modeler

Use these general keyboard shortcuts for working with Semantic Modeler.

Task	Keyboard Shortcut
Close active tab	Ctrl+Shift+X (Windows) Command+Shift+X (Mac)
Copy	Ctrl+C (Windows) Command+C (Mac)
Cut	Ctrl+X (Windows) Command+X (Mac)
Delete	Backspace (Windows) Delete (Mac)
Edit	Enter (Windows) Enter (Mac)
Global Check Consistency	Ctrl+K (Windows) Command+K (Mac)
Global Search	Ctrl+Shift+F (Windows) Command+Shift+F (Mac)
Go to Connection Tab in Finger pane	Ctrl+Alt+0 (Windows) Command+Option+0 (Mac)
Go to Invalid Tab in Finger pane	Ctrl+Alt+5 (Windows) Command+Option+5 (Mac)
Go to Logical Tab in Finger pane	Ctrl+Alt+2 (Windows) Command+Option+2 (Mac)
Go to Physical Tab in Finger pane	Ctrl+Alt+1 (Windows) Command+Option+1 (Mac)
Go to Presentation Tab in Finger pane	Ctrl+Alt+3 (Windows) Command+Option+3 (Mac)

Task	Keyboard Shortcut
Go to Variables Tab in Finger pane	Ctrl+Alt+4 (Windows) Command+Option+4 (Mac)
Paste	Ctrl+V (Windows) Command+V (Mac)
Save the Model	Ctrl+S (Windows) Command+S (Mac)
Show/Hide Finger pane	Ctrl+Alt+H (Windows) Command+Option+H (Mac)
Show/Hide Git pane	Ctrl+Alt+G (Windows) Command+Option+G (Mac)
View Lineage	Ctrl+Shift+L (Windows) Command+Shift+L (Mac)
View Source (SMML)	Shift+Enter (Windows) Shift+Enter (Mac)

Model Binary Large Object (BLOB) Data and Character Large Object (CLOB) Data

Learn how to model binary large object (BLOB) data and character large object (CLOB) data in a semantic model.

CLOB data is a large plain text document in any character set. The supported BLOB image types are: GIF, PNG, TIFF, JPEG, and BMP. BLOB formats not supported are: PDF, audio, or video.

The default data type for BLOB columns after the import is LongVarBinary, while for CLOB columns it's LongVarChar. The column for the BLOB or CLOB can't exceed the `MaxFieldSize` limit of 32 KB.

When configuring the physical joins, create a physical join between the tables using the primary key when the primary key is used as a join in the other table.

1. On the Home page, click **Navigator** and then click **Semantic Models**.
2. In the Semantic Models page, click a semantic model to open it.
3. Import the physical table containing the BLOB or CLOB data from the data source into the physical layer.
4. After import, open the physical column for the BLOB or CLOB column, and change the **Length** field.
5. Configure physical joins.
6. Drag the BLOB or CLOB column to the logical layer to generate a logical column.
7. Configure a physical lookup for the logical column to ensure that the Oracle Analytics query engine doesn't generate a group by or order by on the logical column.
8. In the logical column's General tab, configure the **Descriptor ID column** to ensure that Presentation Services uses the correct column when generating filters.

9. Configure the **Sort order column**, configure the sort order column to ensure that the Oracle Analytics query engine orders column as expected.
10. Save the changes.

20

Data Types Supported by Oracle Analytics Cloud

This topic provides information about supported data types and semantic models.

Topics:

Data Types Supported by Oracle Analytics

This topic contains a list by category of the data types that you can use with Oracle Analytics.

For more information about supported base data types, see Supported Data Types.

Binary Data

The supported binary data types are:

- BIT
- BINARY
- LONGVARBINARY
- VARBINARY

Date and Time Data

The supported date and time data types are:

- DATE
- TIME
- TIMESTAMP

Numeric Data

For information about DOUBLE and FLOAT, see Floating Point Limitations and Data Type Limitations.

The supported numeric data types are:

- BIGINT
- DECIMAL
- DOUBLE
- FLOAT
- INTEGER
- NUMERIC
- REAL
- SMALLINT
- TINYINT

Textual Data

The supported textual data types are:

- CHAR
- LONGVARCHAR
- VARCHAR

Data Type Limitations

This topic lists the supported data types, their descriptions, and their limitations.

An administrator or semantic model developer can use this information to evaluate whether a particular data type is suitable for a given column or set of values, and to determine whether the data type is capable of representing all the required values.

For example, the `INTEGER` column in the Oracle database supports a very large range of values, up to 38 decimal digits, but the `INTEGER` data type in Oracle Analytics is a 32-bit binary integer type that's capable of holding up to nine digits without encountering data overflow (truncation) issues. If the column holds values in the range of [-2,147,483,648, 2,147,483,647], then you should use the `INTEGER` data type. However, if the column stores values larger than this range, then you should use another data type such as `NUMERIC` or `VARCHAR`.

Choose the smallest, in bytes, data type that's capable of representing the column's expected range of values. Choosing a data type in this way reduces the amount of memory and disk space consumed by the Oracle BI Server for cache files, temp files, and so on.

Data Type	Limitations
BIG INT	JDBC and the Semantic Modeler don't support this type; therefore, Oracle Analytics doesn't fully support the BIG INT type. The BIG INT type is intended to be same as the C int64 data type.
BINARY	Oracle Analytics doesn't fully support the BINARY type. Oracle Analytics supports only the fetching of columns whose data type is BINARY. The Oracle Analytics query engine doesn't support the BINARY type in bind parameters or insert statements.

Data Type	Limitations
BIT	Oracle Analytics doesn't fully support the BIT type. Instead, you should use either the INT or CHAR type to represent Boolean data.
CHAR	The CHAR type's values are always padded with ending spaces that can equal up to the length specified by the data type. The CHAR type supports Unicode values.
DATE	The DATE type represents only year, month, and day components. DATE type doesn't represent hours, minutes, or seconds like the Oracle DATE data type.
DECIMAL	The DECIMAL type is the same as the NUMERIC type.
DOUBLE	The DOUBLE type is the same as the IEEE 754 64-bit double-precision binary floating-point data type. The internal storage is eight bytes. The significand occupies 53 bits (including the sign bit). Therefore, the precision is limited to approximately 16 decimal digits. The exponent occupies 11 bits. The range of the exponent is approximately ±307 as a base 10 decimal value. See Floating Point Limitations.
INTEGER	The INTEGER type is a signed binary integer data type occupying four bytes. The maximum value that can be represented is 2,147,483,647, and the minimum value is -2,147,483,648.
FLOAT	The FLOAT type is the same as the IEEE 754 32-bit single-precision binary floating-point data type. The internal storage is four bytes. The significand occupies 24 bits (including the sign bit). Therefore, the precision is limited to approximately 7 decimal digits. The exponent occupies eight bits. The range of the exponent is approximately ±38 as a base 10 decimal value. See Floating Point Limitations.
LONGVARBINARY	The LONGVARBINARY type supports up to 32,678 bytes.
LONGVARCHAR	The LONGVARCHAR type supports up to 32,678 bytes. Both the LONGVARCHAR type and the VARCHAR type support Unicode values.
NUMERIC	The NUMERIC type is a true decimal data type occupying 22 bytes. The internal representation and limitations are the same as the Oracle NUMBER data type. The NUMERIC type supports positive numbers in the range of 1 x 10^-130 to 9.999...9 x 10^125 with up to 38 significant digits. The precision and scale aren't stored in the semantic model. The scale is assumed to be 10.
REAL	The REAL type has the same description and limitations as the FLOAT type.
SMALLINT	The SMALLINT type is represented as the INTEGER type internally in the Oracle Analytics query engine and has the same limitations as the INTEGER data type.
TIME	The TIME type represents only hour, minute, and second components.
TIMESTAMP	The TIMESTAMP type represents year, month, day, hour, minute, and second components. For some data sources on some platforms, it can also support fractions of a second.

Data Type	Limitations
TINYINT	The TINYINT type is represented as an INTEGER internally in the Oracle Analytics query engine. The TINYINT type and INTEGER type have the same limitations.
VARBINARY	The VARBINARY type is interchangeable with the LONGVARBINARY type. The VARBINARY type and the LONGVARBINARY type have the same limitations.
VARCHAR	The VARCHAR type is interchangeable with the LONGVARCHAR type. The VARCHAR type and LONGCARCHAR type have the same limitations. Semantic Modeler allows users to enter a maximum character length of 2,147,483,647. However, the actual maximum length supported is 32,678.

Floating Point Limitations

You can't represent some numbers exactly with binary floating point data types such as FLOAT and DOUBLE.

When converting decimal numbers to and from binary floating point representations, often there are rounding errors because of the representational limitations of binary floating point formats. For example, a decimal number such as 1.365 might be represented as 1.364999999999999 when converted to the DOUBLE type. When this number is rounded to 3 digits after the decimal point, the result is 1.365. However, if the number is rounded to 2 decimal digits, then the result is 1.36 and not 1.37.

To avoid the limitations of the FLOAT and DOUBLE types, Oracle suggests that you update the FLOAT and DOUBLE data types to the NUMERIC type. There is no workaround to fix the inherent limitations with binary floating point data types, other than switching to the NUMERIC data type.

Use the NQSGetSQLDataTypes Procedure to Access Data Type Information

To access a list of data types supported by Oracle Analytics Server, use the Oracle BI Server nqcmd utility to run the NQSGetSQLDataTypes procedure.

For example: `call NQSGetSQLDataTypes(0);`

When you run this procedure, the results contain a list of supported data types and information specific to each data type such as case sensitivity and the ability to search.

See Use nqcmd to Test and Refine the Repository.

SQL Identifier Character Limitation

In addition to the data type limitations, 128 characters is the default maximum length of all SQL identifiers that Oracle Analytics can process.

See Data Type Limitations.

Other Oracle BI Server Limitations

Learn about other data type limitations such as table name and column name length.

In addition to the data type limitations, Oracle BI Server has the following limitations:

- The default maximum length of all fields in Oracle BI Server is 32,678 bytes. This default limit can be changed by setting the environment variable `OBIS_MAX_FIELD_SIZE`.
- The default maximum length of all SQL identifiers, for example, table names and column names, is 128 characters.

Data Type Mapping in Oracle Database and Oracle Analytics

When you import metadata from an Oracle Database, the Semantic Modeler uses these mappings to determine the corresponding data type in the Oracle Analytics query engine for each imported column.

The mapping of data types from the Oracle Database to the Oracle Analytics query engine might differ depending on the database.

Oracle Database Data Type	Oracle Analytics Data Type
CHAR	CHAR
NCHAR	CHAR
VARCHAR2	VARCHAR
NVARCHAR2	VARCHAR
NUMBER (precision, scale)	INT if scale = 0 and 1 <= precision <= 9; otherwise, same as NUMBER
BINARY_FLOAT	FLOAT
BINARY_DOUBLE	DOUBLE
DATE	DATETIME
TIMESTAMP	TIMESTAMP
TIMESTAMP WITH TIME ZONE	TIMESTAMP
TIMESTAMP WITH LOCAL TIME ZONE	TIMESTAMP
BLOB	LONGVARBINARY
CLOB	LONGVARCHAR
NCLOB	LONGVARCHAR
BFILE	Not supported
LONG	LONGVARCHAR
LONG RAW	Not supported
ROWID	CHAR
XML Type	LONGVARBINARY
UriType	Not supported

21

Expression Editor Reference

This part describes the expression elements that you can use in the Expression Editor

Topics:

SQL Operators

You use SQL operators to specify comparisons and arithmetic operations between expressions.

You can use various types of SQL operators.

Operator	Example	Description	Syntax
BETWEEN	`"COSTS"."UNIT_COST" BETWEEN 100.0 AND 5000.0`	Determines if a value is between two non-inclusive bounds. `BETWEEN` can be preceded with `NOT` to negate the condition.	`BETWEEN [LowerBound] AND [UpperBound]`
IN	`"COSTS"."UNIT_COST" IN(200, 600, 'A')`	Determines if a value is present in a set of values.	`IN ([Comma Separated List])`
IS NULL	`"PRODUCTS"."PROD_NAME" IS NULL`	Determines if a value is null.	`IS NULL`
LIKE	`"PRODUCTS"."PROD_NAME" LIKE 'prod%'`	Determines if a value matches all or part of a string. Often used with wildcard characters to indicate any character string match of zero or more characters (%) or any single character match (_).	`LIKE`
+	`(FEDERAL_REVENUE + LOCAL_REVENUE) - TOTAL_EXPENDITURE`	Plus sign for addition.	`+`

Operator	Example	Description	Syntax
-	`(FEDERAL_REVENUE + LOCAL_REVENUE) - TOTAL_EXPENDITURE`	Minus sign for subtraction.	`-`
* *or* X	`SUPPORT_SERVICES_EXPENDITURE * 1.5`	Multiply sign for multiplication.	`*` `X`
/	`CAPITAL_OUTLAY_EXPENDITURE/ 1.05`	Divide by sign for division.	`/`
%		Percentage	`%`
||	`STATE|| CAST(YEAR AS CHAR(4))`	Character string concatenation.	`||`
(	`(FEDERAL_REVENUE + LOCAL_REVENUE) - TOTAL_EXPENDITURE`	Open parenthesis.	`(`
)	`(FEDERAL_REVENUE + LOCAL_REVENUE) - TOTAL_EXPENDITURE`	Close parenthesis.	`)`
>	`YEAR > 2000 and YEAR < 2016 and YEAR <> 2013`	Greater than sign, indicating values higher than the comparison.	`>`
<	`YEAR > 2000 and YEAR < 2016 and YEAR <> 2013`	Less than sign, indicating values lower than the comparison.	`<`
=		Equal sign, indicating the same value.	`=`
>=		Greater than or equal to sign, indicating values the same or higher than the comparison.	`>=`
<=		Less than or equal to sign, indicating values the same or lower than the comparison.	`<=`
<>	`YEAR > 2000 and YEAR < 2016 and YEAR <> 2013`	Not equal to, indicating values higher or lower, but different.	`<>`
,	`STATE in ('ALABAMA','CALIFORNIA')`	Comma, used to separate elements in a list.	`,`

Conditional Expressions

You use conditional expressions to create expressions that convert values.

The conditional expressions described in this section are building blocks for creating expressions that convert a value from one form to another.

Follow these rules:

- In `CASE` statements, `AND` has precedence over `OR`.
- Strings must be in single quotes.

Expression	Example	Description	Syntax

CASE (If)	`CASE`
	`WHEN score-par < 0 THEN 'Under Par'`
	`WHEN score-par = 0 THEN 'Par'`
	`WHEN score-par = 1 THEN 'Bogey'`
	`WHEN score-par = 2 THEN 'Double Bogey'`
	`ELSE 'Triple Bogey or Worse'`
	`END`
		Evaluates each `WHEN` condition and if satisfied, assigns the value in the corresponding `THEN` expression.
		If none of the `WHEN` conditions are satisfied, it assigns the default value specified in the `ELSE` expression. If no `ELSE` expression is specified, the system automatically adds an `ELSE NULL`.
		Note: See *Best Practices for using CASE statements in Analyses and Visualizations*.
			`CASE WHEN request_condition1 THEN expr1 ELSE expr2 END`

CASE (Switch)	`CASE Score-par`
	`WHEN -5 THEN 'Birdie on Par 6'`
	`WHEN -4 THEN 'Must be Tiger'`
	`WHEN -3 THEN 'Three under par'`
	`WHEN -2 THEN 'Two under par'`
	`WHEN -1 THEN 'Birdie'`
	`WHEN 0 THEN 'Par'`
	`WHEN 1 THEN 'Bogey'`
	`WHEN 2 THEN 'Double Bogey'`
	`ELSE 'Triple Bogey or Worse'`
	`END`
		Also referred to as `CASE (Lookup)`. The value of the first expression is examined, then the `WHEN` expressions. If the first expression matches any `WHEN` expression, it assigns the value in the corresponding `THEN` expression.
		If none of the `WHEN` expressions match, it assigns the default value specified in the `ELSE` expression. If no `ELSE` expression is specified, the system automatically adds an `ELSE NULL`.
		If the first expression matches an expression in multiple `WHEN` clauses, only the expression following the first match is assigned.
		Note See *Best Practices for using CASE statements in Analyses and Visualizations*.
			`CASE expr1 WHEN expr2 THEN expr3 ELSE expr4 END`

Expression	Example	Description	Syntax
IfCase > ELSE	-	-	`ELSE [expr]`
IfCase > IFNULL	-	-	`IFNULL([expr], [value])`
IfCase > NULLIF	-	-	`NULLIF([expr], [expr])`
IfCase > WHEN	-	-	`WHEN [Condition] THEN [expr]`
IfCase > CASE	-	-	`CASE WHEN [Condition] THEN [expr] END`
SwitchCase > ELSE	-	-	`ELSE [expr]`
SwitchCase >IFNULL	-	-	`IFNULL([expr], [value])`
SwitchCase > NULLIF	-	-	`NULLIF([expr], [expr])`
SwitchCase > WHEN	-	-	`WHEN [Condition] THEN [expr]`

Functions

There are various types of functions that you can use in expressions.

Topics:

Aggregate Functions

Aggregate functions perform operations on multiple values to create summary results.

The following list describes the aggregation rules that are available for columns and measure columns. The list also includes functions that you can use when creating calculated items for analyses.

- **Default** — Applies the default aggregation rule as in the semantic model or by the original author of the analysis. Not available for calculated items in analyses.
- **Server Determined** — Applies the aggregation rule that's determined by the Oracle BI Server (such as the rule that is defined in the semantic model). The aggregation is performed within Oracle BI Server for simple rules such as Sum, Min, and Max. Not available for measure columns in the Layout pane or for calculated items in analyses.
- **Sum** — Calculates the sum obtained by adding up all values in the result set. Use this for items that have numeric values.
- **Min** — Calculates the minimum value (lowest numeric value) of the rows in the result set. Use this for items that have numeric values.
- **Max** — Calculates the maximum value (highest numeric value) of the rows in the result set. Use this for items that have numeric values.
- **Average** — Calculates the average (mean) value of an item in the result set. Use this for items that have numeric values. Averages on tables and pivot tables are rounded to the nearest whole number.
- **First** — In the result set, selects the first occurrence of the item for measures. For calculated items, selects the first member according to the display in the Selected list. Not available in the Edit Column Formula dialog box.
- **Last** — In the result set, selects the last occurrence of the item. For calculated items, selects the last member according to the display in the Selected list. Not available in the Edit Column Formula dialog box.
- **Count** — Calculates the number of rows in the result set that have a non-null value for the item. The item is typically a column name, in which case the number of rows with non-null values for that column are returned.
- **Count Distinct** — Adds distinct processing to the Count function, which means that each distinct occurrence of the item is counted only once.
- **None** — Applies no aggregation. Not available for calculated items in analyses.
- **Server Complex Aggregate** — Applies the aggregation rule that is determined by the Oracle BI Server (such as the rule that is defined in the semantic model). The aggregation is performed by the Oracle BI Server, rather than within Presentation Services. Not available for calculated items in analyses.
- **Report-Based Total (when applicable)** — If not selected, specifies that the Oracle BI Server should calculate the total based on the entire result set, before applying any filters to the measures. Not available in the Edit Column Formula dialog box or for calculated items in analyses. Only available for attribute columns.

Function	Example	Description	Syntax
AGGREGATE AT	`AGGREGATE(sales AT year)`	Aggregates columns based on the level or levels in the data model hierarchy you specify. • *measure* is the name of a measure column. • `level` is the level at which you want to aggregate. You can optionally specify more than one level. You can't specify a level from a dimension that contains levels that are being used as the measure level for the measure you specified in the first argument. For example, you can't write the function as `AGGREGATE(yearly_sales AT month)` if *month* is from the same time dimension used as the measure level for *yearly_sales*.	`AGGREGATE(measure AT level [, level1, levelN])`
AGGREGATE BY	`AGGREGATE(sales BY month, region)`	Aggregates a measure based on one or more dimension columns. • *measure* is the name of a measure column to aggregate. • `column` is the dimension column at which you want to aggregate. You can aggregate measures based more than one column.	`AGGREGATE(measure BY column [, column1, columnN])`
AVG	`Avg(Sales)`	Calculates the average (mean) of a numeric set of values.	`AVG(expr)`
AVGDISTINCT		Calculates the average (mean) of all distinct values of an expression.	`AVG(DISTINCT expr)`
BIN	`BIN(revenue BY productid, year WHERE productid > 2 INTO 4 BINS RETURNING RANGE_LOW)`	Classifies a given numeric expression into a specified number of equal width buckets. The function can return either the bin number or one of the two end points of the bin interval. numeric_expr is the measure or numeric attribute to bin. BY grain_expr1,..., grain_exprN is a list of expressions that define the grain at which the numeric_expr is calculated. BY is required for measure expressions and is optional for attribute expressions. WHERE a filter to apply to the numeric_expr before the numeric values are assigned to bins INTO number_of_bins BINS is the number of bins to return BETWEEN min_value AND max_value is the min and max values used for the end points of the outermost bins RETURNING NUMBER indicates that the return value should be the bin number (1, 2, 3, 4, etc.). This is the default. RETURNING RANGE_LOW indicates the lower value of the bin interval RETURNING RANGE_HIGH indicates the higher value of the bin interval	`BIN(numeric_expr [BY grain_expr1, ..., grain_exprN] [WHERE condition] INTO number_of_bins BINS [BETWEEN min_value AND max_value] [RETURNING {NUMBER | RANGE_LOW | RANGE_HIGH}])`

Function	Example	Description	Syntax
BottomN		Ranks the lowest n values of the expression argument from 1 to n, 1 corresponding to the lowest numerical value. *expr* is any expression that evaluates to a numerical value. integer is any positive integer. Represents the bottom number of rankings displayed in the result set, 1 being the lowest rank.	`BottomN(expr, integer)`
COUNT	`COUNT(Products)`	Determines the number of items with a non-null value.	`COUNT(expr)`
COUNTDISTINCT		Adds distinct processing to the COUNT function. *expr* is any expression.	`COUNT(DISTINCT expr)`
COUNT*	`SELECT COUNT(*) FROM Facts`	Counts the number of rows.	`COUNT(*)`
First	`First(Sales)`	Selects the first non-null returned value of the expression argument. The `First` function operates at the most detailed level specified in your explicitly defined dimension.	`First([NumericExpression)]`
Last	`Last(Sales)`	Selects the last non-null returned value of the expression.	`Last([NumericExpression)]`
MAVG		Calculates a moving average (mean) for the last n rows of data in the result set, inclusive of the current row. *expr* is any expression that evaluates to a numerical value. integer is any positive integer. Represents the average of the last n rows of data.	`MAVG(expr, integer)`
MAX	`MAX(Revenue)`	Calculates the maximum value (highest numeric value) of the rows satisfying the numeric expression argument.	`MAX(expr)`
MEDIAN	`MEDIAN(Sales)`	Calculates the median (middle) value of the rows satisfying the numeric expression argument. When there are an even number of rows, the median is the mean of the two middle rows. This function always returns a double.	`MEDIAN(expr)`
MIN	`MIN(Revenue)`	Calculates the minimum value (lowest numeric value) of the rows satisfying the numeric expression argument.	`MIN(expr)`
NTILE		Determines the rank of a value in terms of a user-specified range. It returns integers to represent any range of ranks. NTILE with numTiles=100 returns what is commonly called the "percentile" (with numbers ranging from 1 to 100, with 100 representing the high end of the sort). *expr* is any expression that evaluates to a numerical value. numTiles is a positive, nonnull integer that represents the number of tiles.	`NTILE(expr, numTiles)`

Function	Example	Description	Syntax
PERCENTILE		Calculates a percentile rank for each value satisfying the numeric expression argument. The percentile rank ranges are between 0 (0th percentile) to 1 (100th percentile). *expr* is any expression that evaluates to a numerical value.	`PERCENTILE(expr)`
RANK	`RANK(chronological_key, null, year_key_columns)`	Calculates the rank for each value satisfying the numeric expression argument. The highest number is assigned a rank of 1, and each successive rank is assigned the next consecutive integer (2, 3, 4,...). If certain values are equal, they'reare assigned the same rank (for example, 1, 1, 1, 4, 5, 5, 7...). *expr* is any expression that evaluates to a numerical value.	`RANK(expr)`
STDDEV	`STDDEV(Sales)` `STDDEV(DISTINCT Sales)`	Returns the standard deviation for a set of values. The return type is always a double.	`STDDEV(expr)`
STDDEV_POP	`STDDEV_POP(Sales)` `STDDEV_POP(DISTINCT Sales)`	Returns the standard deviation for a set of values using the computational formula for population variance and standard deviation.	`STDDEV_POP([NumericExpression])`
SUM	`SUM(Revenue)`	Calculates the sum obtained by adding up all values satisfying the numeric expression argument.	`SUM(expr)`
SUMDISTINCT		Calculates the sum obtained by adding all of the distinct values satisfying the numeric expression argument. *expr* is any expression that evaluates to a numerical value.	`SUM(DISTINCT expr)`
TOPN		Ranks the highest n values of the expression argument from 1 to n, 1 corresponding to the highest numerical value. *expr* is any expression that evaluates to a numerical value. integer is any positive integer. Represents the top number of rankings displayed in the result set, 1 being the highest rank.	`TOPN(expr, integer)`

Analytics Functions

Analytics functions allow you to explore data using models such as trendline and cluster.

Function	Example	Description	Syntax
TRENDLINE	`TRENDLINE(revenue, (calendar_year, calendar_quarter, calendar_month) BY (product), 'LINEAR', 'VALUE')`	Oracle recommends that you apply a Trendline using the **Add Statistics** property when viewing a visualization. See Adjust Visualization Properties. Fits a linear, polynomial, or exponential model, and returns the fitted values or model. The *numeric_expr* represents the Y value for the trend and the *series* (time columns) represent the X value.	`TRENDLINE(numeric_expr, ([series]) BY ([partitionBy]), model_type, result_type)`
CLUSTER	`CLUSTER((product, company), (billed_quantity, revenue), 'clusterName', 'algorithm=k-means;numClusters=%1;maxIter=%2;useRandomSeed=FALSE;enablePartitioning=TRUE', 5, 10)`	Collects a set of records into groups based on one or more input expressions using K-Means or Hierarchical Clustering.	`CLUSTER((dimension_expr1 , ... dimension_exprN), (expr1, ... exprN), output_column_name, options, [runtime_binded_options])`
OUTLIER	`OUTLIER((product, company), (billed_quantity, revenue), 'isOutlier', 'algorithm=kmeans')`	Classifies a record as Outlier based on one or more input expressions using K-Means or Hierarchical Clustering or Multi-Variate Outlier detection Algorithms.	`OUTLIER((dimension_expr1 , ... dimension_exprN), (expr1, ... exprN), output_column_name, options, [runtime_binded_options])`
REGR	`REGR(revenue, (discount_amount), (product_type, brand), 'fitted', '')`	Fits a linear model and returns the fitted values or model. This function can be used to fit a linear curve on two measures.	`REGR(y_axis_measure_expr, (x_axis_expr), (category_expr1, ..., category_exprN), output_column_name, options, [runtime_binded_options])`

Date and Time Functions

Date and time functions manipulate data based on `DATE` and `DATETIME`.

Function	Example	Description	Syntax
CURRENT_Date	`CURRENT_DATE`	Returns the current date. The date is determined by the system in which the Oracle BI is running.	`CURRENT_DATE`

Function	Example	Description	Syntax
CURRENT_TIME	`CURRENT_TIME(3)`	Returns the current time to the specified number of digits of precision, for example: HH:MM:SS.SSS If no argument is specified, the function returns the default precision.	`CURRENT_TIME(expr)`
CURRENT_TIMESTAMP	`CURRENT_TIMESTAMP(3)`	Returns the current date/timestamp to the specified number of digits of precision.	`CURRENT_TIMESTAMP(expr)`
DAYNAME	`DAYNAME(Order_Date)`	Returns the name of the day of the week for a specified date expression.	`DAYNAME(expr)`
DAYOFMONTH	`DAYOFMONTH(Order_Date)`	Returns the number corresponding to the day of the month for a specified date expression.	`DAYOFMONTH(expr)`
DAYOFWEEK	`DAYOFWEEK(Order_Date)`	Returns a number between 1 and 7 corresponding to the day of the week for a specified date expression. For example, 1 always corresponds to Sunday, 2 corresponds to Monday, and so on through to Saturday which returns 7.	`DAYOFWEEK(expr)`
DAYOFYEAR	`DAYOFYEAR(Order_Date)`	Returns the number (between 1 and 366) corresponding to the day of the year for a specified date expression.	`DAYOFYEAR(expr)`
DAY_OF_QUARTER	`DAY_OF_QUARTER(Order_Date)`	Returns a number (between 1 and 92) corresponding to the day of the quarter for the specified date expression.	`DAY_OF_QUARTER(expr)`
HOUR	`HOUR(Order_Time)`	Returns a number (between 0 and 23) corresponding to the hour for a specified time expression. For example, 0 corresponds to 12 a.m. and 23 corresponds to 11 p.m.	`HOUR(expr)`
MINUTE	`MINUTE(Order_Time)`	Returns a number (between 0 and 59) corresponding to the minute for a specified time expression.	`MINUTE(expr)`
MONTH	`MONTH(Order_Time)`	Returns the number (between 1 and 12) corresponding to the month for a specified date expression.	`MONTH(expr)`
MONTHNAME	`MONTHNAME(Order_Time)`	Returns the name of the month for a specified date expression.	`MONTHNAME(expr)`
MONTH_OF_QUARTER	`MONTH_OF_QUARTER(Order_Date)`	Returns the number (between 1 and 3) corresponding to the month in the quarter for a specified date expression.	`MONTH_OF_QUARTER(expr)`
NOW	`NOW()`	Returns the current timestamp. The `NOW` function is equivalent to the `CURRENT_TIMESTAMP` function.	`NOW()`
QUARTER_OF_YEAR	`QUARTER_OF_YEAR(Order_Date)`	Returns the number (between 1 and 4) corresponding to the quarter of the year for a specified date expression.	`QUARTER_OF_YEAR(expr)`
SECOND	`SECOND(Order_Time)`	Returns the number (between 0 and 59) corresponding to the seconds for a specified time expression.	`SECOND(expr)`

Function	Example	Description	Syntax
TIMESTAMPADD	`TIMESTAMPADD(SQL_TSI_MONTH, 12,Time."Order Date")`	Adds a specified number of intervals to a timestamp, and returns a single timestamp. Interval options are: *SQL_TSI_SECOND*, *SQL_TSI_MINUTE*, *SQL_TSI_HOUR*, *SQL_TSI_DAY*, *SQL_TSI_WEEK*, *SQL_TSI_MONTH*, *SQL_TSI_QUARTER*, *SQL_TSI_YEAR*	`TIMESTAMPADD(interval, expr, timestamp)`
TIMESTAMPDIFF	`TIMESTAMPDIFF(SQL_TSI_MONTH, Time."Order Date",CURRENT_DATE)`	Returns the total number of specified intervals between two timestamps. Use the same intervals as TIMESTAMPADD.	`TIMESTAMPDIFF(interval, expr, timestamp2)`
WEEK_OF_QUARTER	`WEEK_OF_QUARTER(Order_Date)`	Returns a number (between 1 and 13) corresponding to the week of the quarter for the specified date expression.	`WEEK_OF_QUARTER(expr)`
WEEK_OF_YEAR	`WEEK_OF_YEAR(Order_Date)`	Returns a number (between 1 and 53) corresponding to the week of the year for the specified date expression.	`WEEK_OF_YEAR(expr)`
YEAR	`YEAR(Order_Date)`	Returns the year for the specified date expression.	`YEAR(expr)`

Date Extraction Functions

These functions calculate or round-down timestamp values to the nearest specified time period, such as hour, day, week, month, and quarter.

You can use the calculated timestamps to aggregate data using a different grain. For example, you might apply the `EXTRACTDAY()` function to sales order dates to calculate a timestamp for midnight on the day that orders occur, so that you can aggregate the data by day.

Function	Example	Description	Syntax
Extract Day	`EXTRACTDAY("Order Date")` • `2/22/1967 3:02:01 AM` returns `2/22/1967 12:00:00 AM`. • `9/2/2022 10:38:21 AM` returns `9/2/2022 12:00:00 AM`.	Returns a timestamp for midnight (12 AM) on the day in which the input value occurs. For example, if the input timestamp is for 3:02:01 AM on February 22nd, the function returns the timestamp for 12:00:00 AM on February 22nd.	`EXTRACTDAY(expr)`
Extract Hour	`EXTRACTHOUR("Order Date")` • `2/22/1967 3:02:01 AM` returns `2/22/1967 3:00:00 AM`. • `6/17/1999 11:18:30 PM` returns `6/17/1999 11:00:00 PM`.	Returns a timestamp for the start of the hour in which the input value occurs. For example, if the input timestamp is for 11:18:30 PM, the function returns the timestamp for 11:00:00 PM.	`EXTRACTHOUR (expr)`

Function	Example	Description	Syntax
Extract Hour of Day	`EXTRACTHOUROFDAY("Order Date")` • `2014/09/24 10:58:00` returns `2000/01/01` **`10`**`:00:00`. • `2014/08/13 11:10:00` returns `2000/01/01` **`11`**`:00:00`	Returns a timestamp where the hour equals the hour of the input value with default values for year, month, day, minutes, and seconds.	`EXTRACTHOUROFDAY(expr)`
Extract Millisecond	`EXTRACTMILLISECOND("Order Date")` • `1997/01/07 15:32:02.150` returns `1997/01/07 15:32:02.`**`150`**. • `1997/01/07 18:42:01.265` returns `1997/01/07 18:42:01.`**`265`**.	Returns a timestamp containing milliseconds for the input value. For example, if the input timestamp is for 15:32:02.150, the function returns the timestamp for 15:32:02.150.	`EXTRACTMILLISECOND(expr)`
Extract Minute	`EXTRACTMINUTE("Order Date")` • `6/17/1999 11:18:00 PM` returns `6/17/1999 11:18:00 PM`. • `9/2/2022 10:38:21 AM` returns `9/2/2022 10:38:00 AM`.	Returns a timestamp for the start of the minute in which the input value occurs. For example, if the input timestamp is for 11:38:21 AM, the function returns the timestamp for 11:38:00 AM.	`EXTRACTMINUTE (expr)`
Extract Month	`EXTRACTMONTH("Order Date")` • `2/22/1967 3:02:01 AM` returns `2/1/1967 12:00:00 AM`. • `6/17/1999 11:18:00 PM` returns `6/1/1999 12:00:00 AM`.	Returns a timestamp for the first day in the month in which the input value occurs. For example, if the input timestamp is for February 22nd, the function returns the timestamp for February 1st.	`EXTRACTMONTH(expr)`

Function	Example	Description	Syntax
Extract Quarter	`EXTRACTQUARTER("Order Date")` • `2/22/1967 3:02:01 AM` returns `1/1/1967 12:00:00 AM`, the first day of the first fiscal quarter. • `6/17/1999 11:18:00 PM` returns `4/1/1999 12:00:00 AM`, the first day of the second fiscal quarter. • `9/2/2022 10:38:21 AM` returns `7/1/2022 12:00:00 AM`, the first day of the third fiscal quarter. **Tip:** Use QUARTER (expr) to calculate just the ordinal quarter from the returned timestamp.	Returns a timestamp for the first day in the quarter in which the input value occurs. For example, if the input timestamp occurs in the third fiscal quarter, the function returns the timestamp for July 1st.	`EXTRACTQUARTER(expr)`
Extract Second	`EXTRACTSECOND("Order Date")` • `1997/01/07 15:32:02.150` returns `1997/01/07 15:32:02`. • `1997/01/07 20:44:18.163` returns `1997/01/07 20:44:18`.	Returns a timestamp for the input value. For example, if the input timestamp is for 15:32:02.150, the function returns the timestamp for 15:32:02.	`EXTRACTSECOND(expr)`
Extract Week	`EXTRACTWEEK("Order Date")` • `2014/09/24 10:58:00` returns `2014/09/21`. • `2014/08/13 11:10:00` returns `2014/08/10`.	Returns the date of the first day of the week (Sunday) in which the input value occurs. For example, if the input timestamp is for Wednesday, September 24th, the function returns the timestamp for Sunday, September 21st.	`EXTRACTWEEK(expr)`
Extract Year	`EXTRACTYEAR("Order Date")` • `1967/02/22 03:02:01` returns `1967/01/01 00:00:00`. • `1999/06/17 23:18:00` returns `1999/01/01 00:00:00`.	Returns a timestamp for January 1st for the year in which the input value occurs. For example, if the input timestamp occurs in 1967, the function returns the timestamp for January 1st, 1967.	`EXTRACTYEAR (expr)`

Conversion Functions

Conversion functions convert a value from one form to another.

Function	Example	Description	Syntax
CAST	`CAST(hiredate AS CHAR(40)) FROM employee`	Changes the data type of an expression or a null literal to another data type. For example, you can cast a *customer_name* (a data type of `CHAR` or `VARCHAR`) or *birthdate* (a datetime literal). Use `CAST` to change to a *Date* data type. Don't use `TODATE`.	`CAST(expr AS type)`
IFNULL	`IFNULL(Sales, 0)`	Tests if an expression evaluates to a null value, and if it does, assigns the specified value to the expression.	`IFNULL(expr, value)`
INDEXCOL	`SELECT INDEXCOL(VALUEOF (NQ_SESSION.GEOGRAPHY_LEVEL), Country, State, City), Revenue FROM Sales`	Uses external information to return the appropriate column for the signed-in user to see.	`INDEXCOL([integer literal], [expr1] [, [expr2], ?-])`
NULLIF	`SELECT e.last_name, NULLIF(e.job_id, j.job_id) "Old Job ID" FROM employees e, job_history j WHERE e.employee_id = j.employee_id ORDER BY last_name, "Old Job ID";`	Compares two expressions. If they're equal, then the function returns NULL. If they're not equal, then the function returns the first expression. You can't specify the literal NULL for the first expression.	`NULLIF([expression], [expression])`
To_DateTime	`SELECT To_DateTime ('2009-03-0301:01:00' , 'yyyy-mm-dd hh:mi:ss') FROM sales`	Converts string literals of *DateTime* format to a *DateTime* data type.	`To_DateTime([expression], [literal])`
VALUEOF	`SalesSubjectArea.Customer.Region = VALUEOF("Region Security"."REGION")`	References the value of a semantic model variable in a filter. Use *expr* variables as arguments of the `VALUEOF` function. Refer to static semantic model variables by name.	`VALUEOF(expr)`

Display Functions

Display functions operate on the result set of a query.

Function	Example	Description	Syntax
BottomN	`BottomN(Sales, 10)`	Returns the *n* lowest values of expression, ranked from lowest to highest.	`BottomN([NumericExpression], [integer])`
FILTER	`FILTER(Sales USING Product = 'widget')`	Computes the expression using the given preaggregate filter.	`FILTER(measure USING filter_expr)`

Function	Example	Description	Syntax
MAVG	`MAVG(Sales, 10)`	Calculates a moving average (mean) for the last *n* rows of data in the result set, inclusive of the current row.	`MAVG([NumericExpression], [integer])`
MSUM	`SELECT Month, Revenue, MSUM(Revenue, 3) as 3_MO_SUM FROM Sales`	Calculates a moving sum for the last *n* rows of data, inclusive of the current row. The sum for the first row is equal to the numeric expression for the first row. The sum for the second row is calculated by taking the sum of the first two rows of data, and so on. When the *n* th row is reached, the sum is calculated based on the last *n* rows of data.	`MSUM([NumericExpression], [integer])`
NTILE	`NTILE(Sales, 100)`	Determines the rank of a value in terms of a user-specified range. It returns integers to represent any range of ranks. The example shows a range from 1 to 100, with the lowest sale = 1 and the highest sale = 100.	`NTILE([NumericExpression], [integer])`
PERCENTILE	`PERCENTILE(Sales)`	Calculates a percent rank for each value satisfying the numeric expression argument. The percentile rank ranges are from 0 (1st percentile) to 1 (100th percentile), inclusive.	`PERCENTILE([NumericExpression])`
RANK	`RANK(Sales)`	Calculates the rank for each value satisfying the numeric expression argument. The highest number is assigned a rank of 1, and each successive rank is assigned the next consecutive integer (2, 3, 4,...). If certain values are equal, they're assigned the same rank (for example, 1, 1, 1, 4, 5, 5, 7...).	`RANK([NumericExpression])`
RCOUNT	`SELECT month, profit, RCOUNT(profit) FROM sales WHERE profit > 200`	Takes a set of records as input and counts the number of records encountered so far.	`RCOUNT([NumericExpression])`
RMAX	`SELECT month, profit, RMAX(profit) FROM sales`	Takes a set of records as input and shows the maximum value based on records encountered so far. The specified data type must be one that can be ordered.	`RMAX([NumericExpression])`
RMIN	`SELECT month, profit, RMIN(profit) FROM sales`	Takes a set of records as input and shows the minimum value based on records encountered so far. The specified data type must be one that can be ordered.	`RMIN([NumericExpression])`
RSUM	`SELECT month, revenue, RSUM(revenue) as RUNNING_SUM FROM sales`	Calculates a running sum based on records encountered so far. The sum for the first row is equal to the numeric expression for the first row. The sum for the second row is calculated by taking the sum of the first two rows of data, and so on.	`RSUM([NumericExpression])`
TOPN	`TOPN(Sales, 10)`	Returns the *n* highest values of expression, ranked from highest to lowest.	`TOPN([NumericExpression], [integer])`

Evaluate Functions

Evaluate functions are database functions that can be used to pass through expressions to get advanced calculations.

Embedded database functions can require one or more columns. These columns are referenced by %1 ... %N within the function. The actual columns must be listed after the function.

Function	Example	Description	Syntax
EVALUATE	`SELECT EVALUATE('instr(%1, %2)', address, 'Foster City') FROM employees`	Passes the specified database function with optional referenced columns as parameters to the database for evaluation.	`EVALUATE([string expression], [comma separated expressions])`
EVALUATE_AGGR	`EVALUATE_AGGR('REGR_SLOPE(%1, %2)', sales.quantity, market.marketkey)`	Passes the specified database function with optional referenced columns as parameters to the database for evaluation. This function is intended for aggregate functions with a `GROUP BY` clause.	`EVALUATE_AGGR('db_agg_function(%1...%N)' [AS datatype] [, column1, columnN])`

Mathematical Functions

The mathematical functions described in this section perform mathematical operations.

Function	Example	Description	Syntax
ABS	`ABS(Profit)`	Calculates the absolute value of a numeric expression. *expr* is any expression that evaluates to a numerical value.	`ABS(expr)`
ACOS	`ACOS(1)`	Calculates the arc cosine of a numeric expression. *expr* is any expression that evaluates to a numerical value.	`ACOS(expr)`
ASIN	`ASIN(1)`	Calculates the arc sine of a numeric expression. *expr* is any expression that evaluates to a numerical value.	`ASIN(expr)`
ATAN	`ATAN(1)`	Calculates the arc tangent of a numeric expression. *expr* is any expression that evaluates to a numerical value.	`ATAN(expr)`
ATAN2	`ATAN2(1, 2)`	Calculates the arc tangent of *y* /*x*, where *y* is the first numeric expression and *x* is the second numeric expression.	`ATAN2(expr1, expr2)`

Function	Example	Description	Syntax
CEILING	`CEILING(Profit)`	Rounds a non-integer numeric expression to the next highest integer. If the numeric expression evaluates to an integer, the `CEILING` function returns that integer.	`CEILING(expr)`
COS	`COS(1)`	Calculates the cosine of a numeric expression. *expr* is any expression that evaluates to a numerical value.	`COS(expr)`
COT	`COT(1)`	Calculates the cotangent of a numeric expression. *expr* is any expression that evaluates to a numerical value.	`COT(expr)`
DEGREES	`DEGREES(1)`	Converts an expression from radians to degrees. *expr* is any expression that evaluates to a numerical value.	`DEGREES(expr)`
EXP	`EXP(4)`	Sends the value to the power specified. Calculates *e* raised to the n-th power, where *e* is the base of the natural logarithm.	`EXP(expr)`
ExtractBit	`Int ExtractBit(1, 5)`	Retrieves a bit at a particular position in an integer. It returns an integer of either 0 or 1 corresponding to the position of the bit.	`ExtractBit([Source Number], [Digits])`
FLOOR	`FLOOR(Profit)`	Rounds a non-integer numeric expression to the next lowest integer. If the numeric expression evaluates to an integer, the `FLOOR` function returns that integer.	`FLOOR(expr)`
LOG	`LOG(1)`	Calculates the natural logarithm of an expression. *expr* is any expression that evaluates to a numerical value.	`LOG(expr)`
LOG10	`LOG10(1)`	Calculates the base 10 logarithm of an expression. *expr* is any expression that evaluates to a numerical value.	`LOG10(expr)`
MOD	`MOD(10, 3)`	Divides the first numeric expression by the second numeric expression and returns the remainder portion of the quotient.	`MOD(expr1, expr2)`
PI	`PI()`	Returns the constant value of pi.	`PI()`
POWER	`POWER(Profit, 2)`	Takes the first numeric expression and raises it to the power specified in the second numeric expression.	`POWER(expr1, expr2)`
RADIANS	`RADIANS(30)`	Converts an expression from degrees to radians. *expr* is any expression that evaluates to a numerical value.	`RADIANS(expr)`
RAND	`RAND()`	Returns a pseudo-random number between 0 and 1.	`RAND()`

Function	Example	Description	Syntax
RANDFromSeed	`RAND(2)`	Returns a pseudo-random number based on a seed value. For a given seed value, the same set of random numbers are generated.	`RAND(expr)`
ROUND	`ROUND(2.166000, 2)`	Rounds a numeric expression to *n* digits of precision. *expr* is any expression that evaluates to a numerical value. *integer* is any positive integer that represents the number of digits of precision.	`ROUND(expr, integer)`
SIGN	`SIGN(Profit)`	Returns the following: • 1 if the numeric expression evaluates to a positive number • -1 if the numeric expression evaluates to a negative number • 0 if the numeric expression evaluates to zero	`SIGN(expr)`
SIN	`SIN(1)`	Calculates the sine of a numeric expression.	`SIN(expr)`
SQRT	`SQRT(7)`	Calculates the square root of the numeric expression argument. The numeric expression must evaluate to a nonnegative number.	`SQRT(expr)`
TAN	`TAN(1)`	Calculates the tangent of a numeric expression. *expr* is any expression that evaluates to a numerical value.	`TAN(expr)`
TRUNCATE	`TRUNCATE(45.12345, 2)`	Truncates a decimal number to return a specified number of places from the decimal point. *expr* is any expression that evaluates to a numerical value. *integer* is any positive integer that represents the number of characters to the right of the decimal place to return.	`TRUNCATE(expr, integer)`

Running Aggregate Functions

Running aggregate functions perform operations on multiple values to create summary results.

Function	Example	Description	Syntax
MAVG		Calculates a moving average (mean) for the last *n* rows of data in the result set, inclusive of the current row. *expr* is any expression that evaluates to a numerical value. *integer* is any positive integer. Represents the average of the last *n* rows of data.	`MAVG(expr, integer)`

Function	Example	Description	Syntax
MSUM	`select month, revenue, MSUM(revenue, 3) as 3_MO_SUM from sales_subject_area`	Calculates a moving sum for the last n rows of data, inclusive of the current row. *expr* is any expression that evaluates to a numerical value. *integer* is any positive integer. Represents the sum of the last n rows of data.	`MSUM(expr, integer)`
RSUM	`SELECT month, revenue, RSUM(revenue) as RUNNING_SUM from sales_subject_area`	Calculates a running sum based on records encountered so far. *expr* is any expression that evaluates to a numerical value.	`RSUM(expr)`
RCOUNT	`select month, profit, RCOUNT(profit) from sales_subject_area where profit > 200`	Takes a set of records as input and counts the number of records encountered so far. *expr* is an expression of any datatype.	`RCOUNT(expr)`
RMAX	`SELECT month, profit,RMAX(profit) from sales_subject_area`	Takes a set of records as input and shows the maximum value based on records encountered so far. *expr* is an expression of any datatype.	`RMAX(expr)`
RMIN	`select month, profit,RMIN(profit) from sales_subject_area`	Takes a set of records as input and shows the minimum value based on records encountered so far. *expr* is an expression of any datatype.	`RMIN(expr)`

Spatial Functions

Spatial functions enable you to perform geographical analysis when you model data. For example, you might calculate the distance between two geographical areas (known as shapes or polygons).

> **Note:**
>
> You can't use these spatial functions in custom calculations for visualization workbooks.

Function	Example	Description	Syntax
GeometryArea	`GeometryArea(Shape)`	Calculates the area that a shape occupies.	`GeometryArea(Shape)`
GeometryDistance	`GeometryDistance(TRIP_START, TRIP_END)`	Calculates the distance between two shapes.	`GeometryDistance(Shape 1, Shape 2)`

Function	Example	Description	Syntax
GeometryLength	`GeometryLength(Shape)`	Calculates the circumference of a shape.	`GeometryLength(Shape)`
GeometryRelate	`GeometryRelate(TRIP_START, TRIP_END)`	Determines whether one shape is inside another shape. Returns TRUE or FALSE as a string (varchar).	`GeometryRelate(Shape 1, Shape 2)`
GeometryWithinDistance	`GeometryWithinDistance(TRIP_START, TRIP_END, 500)`	Determines whether two shapes are within a specified distance of each other. Returns TRUE or FALSE as a string (varchar).	`GeometryWithinDistance(Shape1, Shape2, DistanceInFloat)`

String Functions

String functions perform various character manipulations. They operate on character strings.

Function	Example	Description	Syntax
ASCII	`ASCII('a')`	Converts a single character string to its corresponding ASCII code, between 0 and 255. If the character expression evaluates to multiple characters, the ASCII code corresponding to the first character in the expression is returned. *expr* is any expression that evaluates to a character string.	`ASCII(expr)`
BIT_LENGTH	`BIT_LENGTH('abcdef')`	Returns the length, in bits, of a specified string. Each Unicode character is 2 bytes in length (equal to 16 bits). *expr* is any expression that evaluates to a character string.	`BIT_LENGTH(expr)`
CHAR	`CHAR(35)`	Converts a numeric value between 0 and 255 to the character value corresponding to the ASCII code. *expr* is any expression that evaluates to a numerical value between 0 and 255.	`CHAR(expr)`
CHAR_LENGTH	`CHAR_LENGTH(Customer_Name)`	Returns the length, in number of characters, of a specified string. Leading and trailing blanks aren't counted in the length of the string. *expr* is any expression that evaluates to a character string.	`CHAR_LENGTH(expr)`
CONCAT	`SELECT DISTINCT CONCAT ('abc', 'def') FROM employee`	Concatenates two character strings. *exprs* are expressions that evaluate to character strings, separated by commas. You must use raw data, not formatted data, with `CONCAT`.	`CONCAT(expr1, expr2)`

Function	Example	Description	Syntax
INSERT	`SELECT INSERT('123456', 2, 3, 'abcd') FROM table`	Inserts a specified character string into a specified location in another character string. *expr1* is any expression that evaluates to a character string. Identifies the target character string. *integer1* is any positive integer that represents the number of characters from the beginning of the target string where the second string is to be inserted. *integer2* is any positive integer that represents the number of characters in the target string to be replaced by the second string. *expr2* is any expression that evaluates to a character string. Identifies the character string to be inserted into the target string.	`INSERT(expr1, integer1, integer2, expr2)`
LEFT	`SELECT LEFT('123456', 3) FROM table`	Returns a specified number of characters from the left of a string. *expr* is any expression that evaluates to a character string *integer* is any positive integer that represents the number of characters from the left of the string to return.	`LEFT(expr, integer)`
LENGTH	`LENGTH(Customer_ Name)`	Returns the length, in number of characters, of a specified string. The length is returned excluding any trailing blank characters. *expr* is any expression that evaluates to a character string.	`LENGTH(expr)`
LOCATE	`LOCATE('d' 'abcdef')`	Returns the numeric position of a character string in another character string. If the character string isn't found in the string being searched, the function returns a value of 0. *expr1* is any expression that evaluates to a character string. Identifies the string for which to search. *expr2* is any expression that evaluates to a character string. Identifies the string to be searched.	`LOCATE(expr1, expr2)`
LOCATEN	`LOCATEN('d' 'abcdef', 3)`	Like LOCATE, returns the numeric position of a character string in another character string. LOCATEN includes an integer argument that enables you to specify a starting position to begin the search. *expr1* is any expression that evaluates to a character string. Identifies the string for which to search. *expr2* is any expression that evaluates to a character string. Identifies the string to be searched. *integer* is any positive (nonzero) integer that represents the starting position to begin to look for the character string.	`LOCATEN(expr1, expr2, integer)`

Function	Example	Description	Syntax
LOWER	`LOWER(Customer_Name)`	Converts a character string to lowercase. *expr* is any expression that evaluates to a character string.	`LOWER(expr)`
OCTET_LENGTH	`OCTET_LENGTH('abcdef')`	Returns the number of bytes of a specified string. *expr* is any expression that evaluates to a character string.	`OCTET_LENGTH(expr)`
POSITION	`POSITION('d', 'abcdef')`	Returns the numeric position of *strExpr1* in a character expression. If *strExpr1* isn't found, the function returns 0. *expr1* is any expression that evaluates to a character string. Identifies the string to search for in the target string. For example, "d". *expr2* is any expression that evaluates to a character string. Identifies the target string to be searched. For example, "abcdef".	`POSITION(expr1, expr2)`
REPEAT	`REPEAT('abc', 4)`	Repeats a specified expression *n* times. *expr* is any expression that evaluates to a character string *integer* is any positive integer that represents the number of times to repeat the character string.	`REPEAT(expr, integer)`
REPLACE	`REPLACE('abcd1234', '123', 'zz')`	Replaces one or more characters from a specified character expression with one or more other characters. *expr1* is any expression that evaluates to a character string. This is the string in which characters are to be replaced. *expr2* is any expression that evaluates to a character string. This second string identifies the characters from the first string that are to be replaced. *expr3* is any expression that evaluates to a character string. This third string specifies the characters to substitute into the first string.	`REPLACE(expr1, expr2, expr3)`
RIGHT	`SELECT RIGHT('123456', 3) FROM table`	Returns a specified number of characters from the right of a string. *expr* is any expression that evaluates to a character string. *integer* is any positive integer that represents the number of characters from the right of the string to return.	`RIGHT(expr, integer)`
SPACE	`SPACE(2)`	Inserts blank spaces. *integer* is any positive integer that indicates the number of spaces to insert.	`SPACE(expr)`

Function	Example	Description	Syntax
SUBSTRING	`SUBSTRING('abcdef' FROM 2)`	Creates a new string starting from a fixed number of characters into the original string. *expr* is any expression that evaluates to a character string. *startPos* is any positive integer that represents the number of characters from the start of the left side of the string where the result is to begin.	`SUBSTRING([SourceString] FROM [StartPostition])`
SUBSTRINGN	`SUBSTRING('abcdef' FROM 2 FOR 3)`	Like SUBSTRING, creates a new string starting from a fixed number of characters into the original string. *SUBSTRINGN* includes an integer argument that enables you to specify the length of the new string, in number of characters. *expr* is any expression that evaluates to a character string. *startPos* is any positive integer that represents the number of characters from the start of the left side of the string where the result is to begin.	`SUBSTRING(expr FROM startPos FOR length)`
TrimBoth	`Trim(BOTH '_' FROM '_abcdef_')`	Strips specified leading and trailing characters from a character string. *char* is any single character. If you omit this specification (and the required single quotes), a blank character is used as the default. *expr* is any expression that evaluates to a character string.	`TRIM(BOTH char FROM expr)`
TRIMLEADING	`TRIM(LEADING '_' FROM '_abcdef')`	Strips specified leading characters from a character string. *char* is any single character. If you omit this specification (and the required single quotes), a blank character is used as the default. *expr* is any expression that evaluates to a character string.	`TRIM(LEADING char FROM expr)`
TRIMTRAILING	`TRIM(TRAILING '_' FROM 'abcdef_')`	Strips specified trailing characters from a character string. *char* is any single character. If you omit this specification (and the required single quotes), a blank character is used as the default. *expr* is any expression that evaluates to a character string.	`TRIM(TRAILING char FROM expr)`
UPPER	`UPPER(Customer_Name)`	Converts a character string to uppercase. *expr* is any expression that evaluates to a character string.	`UPPER(expr)`

System Functions

The `USER` system function returns values relating to the session. For example, the user name you signed in with.

Function	Example	Description	Syntax
DATABASE		Returns the name of the subject area to which you're logged on.	`DATABASE()`
USER		Returns the user name for the semantic model to which you're logged on.	`USER()`

Time Series Functions

Time series functions enable you to aggregate and forecast data based on time dimensions. For example, you might use the AGO function to calculate revenue from one year ago.

Time dimension members must be at or below the level of the function. Because of this, one or more columns that uniquely identify members at or below the given level must be projected in the query.

Function	Example	Description	Syntax
AGO	`SELECT Year_ID, AGO(sales, year, 1)`	Calculates the aggregated value of a measure in a specified time period in the past. For example, to calculate monthly revenue one year ago, use `AGO(Revenue, Year, 1, SHIP_MONTH)`. To calculate quarterly revenues in the last quarter, use `AGO(Revenue, Quarter, 1)`.	`AGO(EXPR, TIME_LEVEL, OFFSET)` Where: • EXPR = the measure to calculate, for example, revenue. • TIME_LEVEL = the time interval, which must be Year, Quarter, Month, Week, or Day. • OFFSET = the number of time intervals to calculate back to, for example, 1 for one year.

Function	Example	Description	Syntax
PERIODROLLING	`SELECT Month_ID, PERIODROLLING (monthly_sales, -1, 1)`	Computes the aggregate of a measure over the period starting *x* units of time and ending *y* units of time from the current time. For example, `PERIODROLLING` can compute sales for a period that starts at a quarter before and ends at a quarter after the current quarter.	`PERIODROLLING(measure, x [,y])` Where: • *measure* = the name of a measure column. • *x* is an integer that specifies the offset from the current time. • *y* specifies the number of time units over which the function computes. • *hierarchy* is an optional argument that specifies the name of a hierarchy in a time dimension, such as *yr*, *mon*, *day*, that you want to use to compute the time window.
TODATE	`SELECT Year_ID, Month_ID, TODATE (sales, year)`	Calculates the aggregated value of a measure from the start of a time period to the latest time period, for example, year to date calculations. For example, to calculate Year to Date Sales, use `TODATE(sales, year)`.	`TODATE(EXPR, TIME_LEVEL)` Where: • EXPR = an expression that references at least one measure column, for example, sales. • TIME_LEVEL = the time interval, which must be Year, Quarter, Month, Week, or Day.

FORECAST Function

Creates a time-series model of the specified measure over the series using Exponential Smoothing (ETS) or Seasonal ARIMA or ARIMA. This function outputs a forecast for a set of periods as specified by the *numPeriods* argument.

Syntax `FORECAST(numeric_expr, ([series]), output_column_name, options, [runtime_binded_options])])`

Where:

- *numeric_expr* indicates the measure to forecast, for example, revenue data.
- *series* indicates the time grain used to build the forecast model. The series is a list of one or more time dimension columns. If you omit series, then the time grain is determined from the query.
- *output_column_name* indicates the valid column names of *forecast*, *low*, *high*, and *predictionInterval*.
- *options* indicates a string list of name/value pairs separated by a semi-colon (;). The value can include *%1 ... %N* specified in `runtime_binded_options`.
- *runtime_binded_options* indicates a comma separated list of columns and options. Values for these columns and options are evaluated and resolved during individual query execution time.

FORECAST Function Options The following table list available options to use with the `FORECAST` function.

Option Name	Values	Description
numPeriods	Integer	The number of periods to forecast.
predictionInterval	0 to 100, where higher values specify higher confidence	The confidence level for the prediction.
modelType	ETS (Exponential Smoothing) SeasonalArima ARIMA	The model to use for forecasting.
useBoxCox	TRUE FALSE	If *TRUE*, then use Box-Cox transformation.
lambdaValue	Not applicable	The Box-Cox transformation parameter. Ignore if NULL or when `useBoxCox` is *FALSE*. Otherwise the data is transformed before the model is estimated.
trendDamp	TRUE FALSE	This is specific to the Exponential Smoothing model. If *TRUE*, then use damped trend. If FALSE or NULL, then use non-damped trend.
errorType	Not applicable	This is specific to the Exponential Smoothing model.
trendType	N (none) A (additive) M (multiplicative) Z (automatically selected)	This is specific to the Exponential Smoothing model
seasonType	N (none) A (additive) M (multiplicative) Z (automatically selected)	This is specific to the Exponential Smoothing model
modelParamIC	ic_auto ic_aicc ic_bic ic_auto (this is the default)	The information criterion (IC) used in the model selection.

Revenue Forecast by Day Example

This example selects revenue forecast by day.

```
FORECAST("A - Sample Sales"."Base Facts"."1- Revenue" Target,
("A - Sample Sales"."Time"."T00 Calendar Date"),'forecast',
'numPeriods=30;predictionInterval=70;') ForecastedRevenue
```

Revenue Forecast by Year and Quarter Example

This example selects revenue forecast by year and quarter.

```
FORECAST("A - Sample Sales"."Base Facts"."1- Revenue",
("A - Sample Sales"."Time"."T01 Year" timeYear, "A - Sample Sales"."Time"."T02
```

```
Quarter" TimeQuarter),'forecast', 'numPeriods=30;predictionInterval=70;')
ForecastedRevenue
```

Constants

You can use constants to include specific fixed dates and times in workbooks and reports.

Constant	Example	Description	Syntax
DATE	`DATE '2026-04-09'`	Creates a specific date in a calculation or expression.	`DATE 'yyyy-mm-dd'`
TIME	`TIME '12:00:00'`	Creates a specific time in a calculation or expression.	`TIME 'hh:mi:ss'`
TIMESTAMP	`TIMESTAMP '2026-04-09 12:00:00'`	Creates a specific time-stamp in a calculation or expression.	`TIMESTAMP 'yyyy-mm-dd hh:mi:ss'`

Types

You can use data types, such as `CHAR`, `INT`, and `NUMERIC` in expressions.

For example, you use types when creating `CAST` expressions that change the data type of an expression or a null literal to another data type.

Variables

Variables are used in expressions.

You can use a variable in an expression.

See Advanced Techniques: Reference Stored Values in Variables.