

HA300

SAP HANA 2.0 SPS06 - Modeling

**PARTICIPANT HANDBOOK
INSTRUCTOR-LED TRAINING**

Course Version: 18
Course Duration: 5 Day(s)
Material Number: 50160226

SAP Copyrights, Trademarks and Disclaimers

© 2022 SAP SE or an SAP affiliate company. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP SE or an SAP affiliate company.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP SE (or an SAP affiliate company) in Germany and other countries. Please see <https://www.sap.com/corporate/en/legal/copyright.html> for additional trademark information and notices.

Some software products marketed by SAP SE and its distributors contain proprietary software components of other software vendors.

National product specifications may vary.

These materials may have been machine translated and may contain grammatical errors or inaccuracies.

These materials are provided by SAP SE or an SAP affiliate company for informational purposes only, without representation or warranty of any kind, and SAP SE or its affiliated companies shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP SE or SAP affiliate company products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

In particular, SAP SE or its affiliated companies have no obligation to pursue any course of business outlined in this document or any related presentation, or to develop or release any functionality mentioned therein. This document, or any related presentation, and SAP SE's or its affiliated companies' strategy and possible future developments, products, and/or platform directions and functionality are all subject to change and may be changed by SAP SE or its affiliated companies at any time for any reason without notice. The information in this document is not a commitment, promise, or legal obligation to deliver any material, code, or functionality. All forward-looking statements are subject to various risks and uncertainties that could cause actual results to differ materially from expectations. Readers are cautioned not to place undue reliance on these forward-looking statements, which speak only as of their dates, and they should not be relied upon in making purchasing decisions.

Typographic Conventions

American English is the standard used in this handbook.

The following typographic conventions are also used.

This information is displayed in the instructor's presentation



Demonstration



Procedure



Warning or Caution



Hint



Related or Additional Information



Facilitated Discussion



User interface control

Example text

Window title

Example text

Contents

vii Course Overview

1 Unit 1: Calculation Views

3	Lesson: Introducing Calculation Views
15	Lesson: Understanding the Different Types of Views
23	Lesson: Working with Common View Design Features

43 Unit 2: Using Nodes in Calculation Views

45	Lesson: Using Projection Nodes
47	Lesson: Using Join Nodes
67	Lesson: Working with Data Sets
77	Lesson: Aggregating Data
81	Lesson: Creating CUBE with Star Join Calculation Views
83	Lesson: Extracting Top Values with Rank Nodes

99 Unit 3: Modeling Functions

101	Lesson: Creating Restricted and Calculated Columns
111	Lesson: Filtering Data
119	Lesson: Using Variables and Input Parameters
135	Lesson: Implementing Hierarchies
149	Lesson: Implementing Currency Conversion
157	Lesson: Defining Time-Based Dimension Calculation Views

167 Unit 4: Using SQL in Models

169	Lesson: Introducing SAP HANA SQL
185	Lesson: Query a Modeled Hierarchy Using SQL
189	Lesson: Working with SQLScript
197	Lesson: Creating and Using Functions
203	Lesson: Creating and Using Procedures

213 Unit 5: Persistence Layer

215	Lesson: Defining the Persistence Layer
223	Lesson: Loading Data into Tables
227	Lesson: Accessing Remote Data

231 Unit 6: Optimization of Models

- | | |
|-----|---|
| 233 | Lesson: Implementing Good Modeling Practices |
| 249 | Lesson: Implementing Static Cache |
| 253 | Lesson: Controlling Parallelization |
| 259 | Lesson: Implementing Union Pruning |
| 263 | Lesson: Using Tools to Check Model Performance |
| 271 | Lesson: Developing a Data Management Architecture |

283 Unit 7: Management and Administration of Models

- | | |
|-----|--|
| 285 | Lesson: Working with Modeling Content in a Project |
| 313 | Lesson: Creating and Managing Projects |
| 321 | Lesson: Enabling Access to External Data |
| 327 | Lesson: Working with GIT Within the SAP Web IDE |
| 349 | Lesson: Migrating Modeling Content |

359 Unit 8: Security in SAP HANA Modeling

- | | |
|-----|--|
| 361 | Lesson: Understanding Roles and Privileges |
| 373 | Lesson: Defining Analytic Privileges |
| 387 | Lesson: Defining Roles |
| 397 | Lesson: Masking Sensitive Data |
| 401 | Lesson: Anonymizing Data |

Course Overview

TARGET AUDIENCE

This course is intended for the following audiences:

- Database Administrator

Lesson 1

Introducing Calculation Views

3

Lesson 2

Understanding the Different Types of Views

15

Lesson 3

Working with Common View Design Features

23

UNIT OBJECTIVES

- Explain modeling objects
- Explain the types of views used in graphical modeling
- Use common features to design calculation views

Introducing Calculation Views

LESSON OVERVIEW

This lesson introduces the general concept of calculation views, and then describes the different types of calculation view.

Business Example

As part of an SAP HANA implementation, you want to use the modeling capabilities of SAP HANA to build flexible data models and easily report on your data.

Before getting into more detail, you want to understand what calculation views are. This is also an opportunity to review the basic concepts used in reporting; such as dimensions, measures, attributes, hierarchies, and so on.



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Explain modeling objects

Key Vocabulary of Modeling

Before introducing modeling in SAP HANA, you must become familiar with some key concepts that are frequently used when reporting on financial or operational data, for example:

- Measure
- Attribute
- Dimension
- Cube
- Star schema
- Hierarchy
- Semantics

Measure and Attribute

When you report on data, you have to distinguish between the following important concepts:

- Measure
- Attribute



Table 1: Measure Versus Attribute

	Measure	Attribute
Definition	A numeric value, such as a price, quantity, or volume, on which you can process arithmetic or statistics operations, such as sum, average, top N values, and calculations.	An element that is used to describe a measure.
Examples	<ul style="list-style-type: none"> • Number of products sold • Unit Price • Total Price 	<ul style="list-style-type: none"> • Product ID • Product Name • Customer ID • Customer Name • Sales Organization • Sales Org. Country • Sales Org. Region • Currency

Attributes are used to filter or aggregate the measures, in order to answer questions such as the following:

- What are the total sales originating from Sales Org. located in the EMEA region?
- What is the sales revenue generated by the product Cars?



Note:

One key objective of modeling in SAP HANA is to create a relevant association between attributes and measures to fulfill a particular reporting requirement.

Dimension

In a number of cases, analyzing the measures is easier if you group attributes together by dimension.

In the example below, the sales organization would be treated as a dimension, with the following associated attributes:

- Country
- Region

Similarly, a Product ID dimension could be associated with several attributes, such as product name, product category, or supplier.



Dimension 1	Dimension 2	Dimension n
Product	Sales Org.	Dim. name
Product Key	Sales Org. Key	Key
Product Name	Sales Org. Name	Name
Product Category	Country	Attribute x
Supplier	Region	Attribute y

Figure 1: Dimensions

Cube

A cube always contains at least one measure and several attributes. At run time, you choose a combination of the attributes to generate a data slice. The measures are usually (but not always) aggregated over the chosen attributes. Cubes do not include dimensions.



Cube
Attribute A
Attribute B
Attribute C
Attribute D
Attribute E
Measure 1
Measure 2
Measure...

Figure 2: Cube

Star Schema

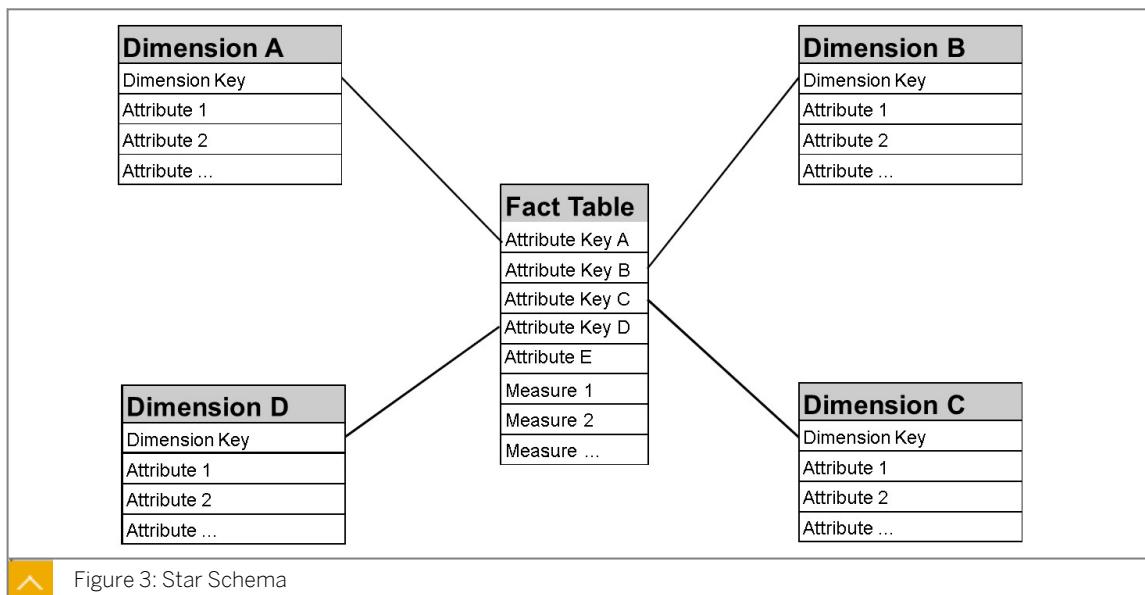


Figure 3: Star Schema

A star schema consists of one fact table that references one or several dimensions.

Each row of the fact table contains measures and attributes that describe a business event, such as a sale or purchase. Some of the attributes in the fact table are also used as keys to join to the dimensions. Attributes that are not used as keys still provide useful information about the fact, such as an invoice date or document status.

Each dimension contains attributes, such as the key of the dimension member (for example the product number), description, and other helpful attributes that can be used to further describe the facts. Dimension attributes can also be used to filter the facts or to build a hierarchy, and perform specific calculations.

In SAP HANA, a star schema is a *cube* that also incorporates one or more *dimensions*.



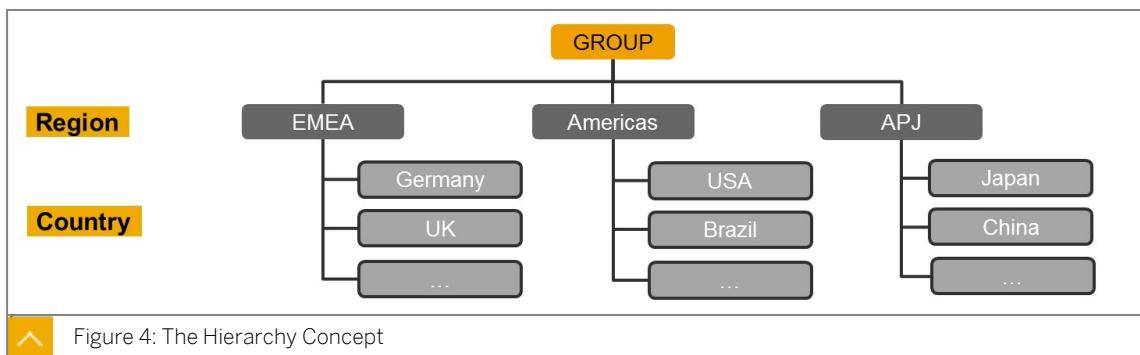
Note:

The term *fact table* is a well-known database modeling term but in SAP HANA calculation view modeling this term is not technically valid. A fact table in an SAP HANA calculation view is called the *Data Foundation* and is not necessarily built from a single table but often from multiple tables or calculation views.

Hierarchy

A hierarchy is a structured representation of an organization, a list of products, the time dimension, and so on, by levels.

It is used to navigate the entire set of members with more ease (the location of the company, the products, or the days, weeks, months, or years) when analyzing the data.



In the figure, The Hierarchy Concept, you see a geographical representation of the structure of a company by regions and countries. The level of detail (granularity) of the hierarchy depends on how the company wants to analyze its data, and its design too. For instance, the geographical hierarchy can be based on where the subsidiaries of a company are located.

In the figure, The Hierarchy Concept, the top node of the hierarchy, *GROUP*, represents the entire company.



Note:

Similarly, the list of products that a company sells could be organized into a hierarchy, by classifying the products by product area or product type.

Semantics

The term **semantics** is sometimes used to describe what a piece of data means, or relates to. A piece of numeric data that you report can be of different types. Here are a few examples:



- A monetary value

For example, the total amount of sales orders.

In this case, you might need a dimension specific to the currency (for example USD, EUR, or GBP), if it is not implicit.

- A number of items

For example, a number of sales orders, or a number of calls to support services.

- A weight, volume, distance, or a compound of these measures

For example, the payload-distance in freight transportation.

You often need to specify the unit in which the data is expressed.

- A percentage

For example, a discount rate, or a tax rate.



Note:

Even if the semantics are not always expressed explicitly in the data repository itself, it is very important to know and understand the semantics of any measure to avoid misinterpretation of the data or irrelevant calculation.

For example, before summing up the amount of sales orders, you must make sure that they are all expressed in the same currency.

As another example, you have to be careful with aggregations when using columns that contain ratios.

Calculation Views in SAP HANA

Calculation views are used in SAP HANA to create a Virtual Data Model (VDM), based on the data that resides in the SAP HANA database schemas.



Operational Reporting | Applications | Analytics

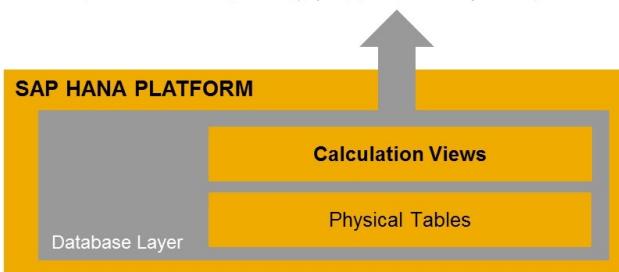


Figure 5: Calculation Views

The purpose of calculation views is to organize the data from the individual transactional tables, and to perform a variety of data calculations, in order to get a relevant and meaningful set of measures and dimensions or attributes to answer a specific reporting need. You can make the data more meaningful than it is in the source tables by customizing the column names, assigning label columns to key columns, and calculating additional attributes.

**Note:**

Calculation Views are a native feature of SAP HANA. In other contexts, a Virtual Data Model can also be implemented using another framework. A good example is SAP S/4HANA, where the Application Layer is based on the SAP Netweaver/ABAP framework: the Virtual Data Model *SAP S/4HANA Analytics* is developed using ABAP Core Data Services (CDS) and does not rely on SAP HANA native graphical calculation views.

Benefits of Calculation Views

The main benefits of using calculation views in SAP HANA are as follows:



- All calculations are performed on the fly within the database engines

No aggregates need to be pre-calculated, and the front-end reporting tools delegate most of the data processing workload (filtering, aggregation, calculations) to the SAP HANA in-memory engines in order to achieve very high performance.

- Reusability

Each calculation view can be used (referenced) by other calculation views.

- Flexibility

Calculation views provide a number of features that make them very flexible. For example, defining hierarchies, filtering data, generating prompts for variables and input parameters, and performing currency conversion.

- Adaptability

An SAP HANA calculation view can adapt its behavior to the list of columns that are selected or projected on top of it. For example, the granularity of a Rank node can be determined dynamically depending on whether you query results by country or by country and customer.

- Easy to transport

SAP HANA provides powerful tools to transport calculation views between different SAP HANA databases; for instance, to install SAP-delivered calculation views, or transport your own calculation views between your development, quality assurance, and production landscapes.

**Note:**

These benefits are illustrated in the following lessons of this unit, and in the following units.

Design-Time Versus Runtime Calculation Views

In SAP HANA, calculation views, like a large variety of development objects, can exist both as a design-time and runtime object.

- Design-Time View

The design-time view is the object that you create and modify with a graphical or text-based editor.

- Runtime View

The runtime view is a database catalog object, more specifically a column view, that is used when you preview the data within SAP HANA, execute a SQL query, or execute a query on top of the calculation view with an external tool, such as BI tools, Microsoft Excel, and so on.



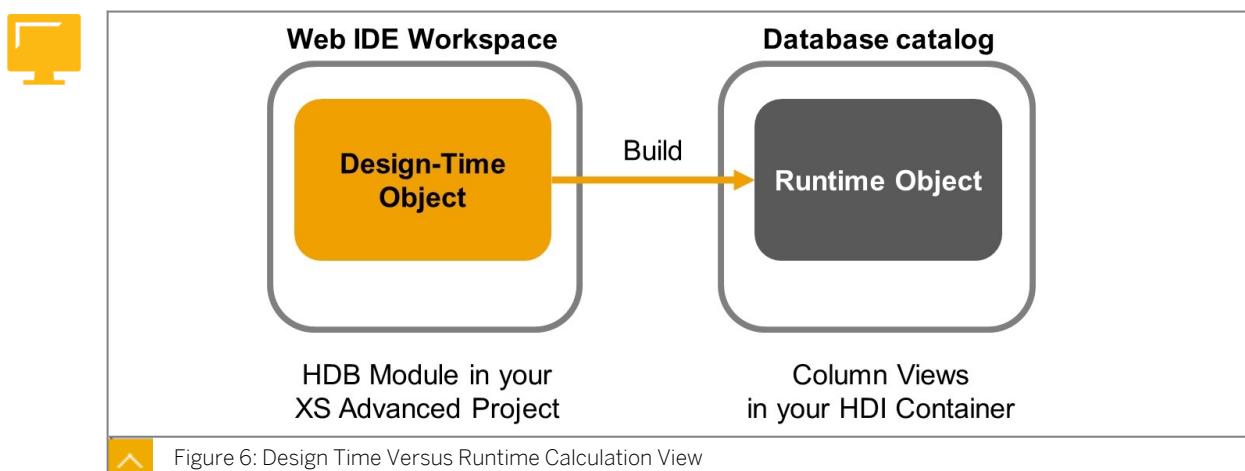
Caution:

A view cannot expose its data before a runtime version of this view is successfully created in the SAP HANA database catalog. Similarly, any change to a design-time view is only visible in a query after the runtime view has been successfully updated.

Building Calculation Views

In many cases, building a single design-time object creates several related runtime objects in the database schema. For example, in addition to the main column view itself, which contains the measures, you can find a column view with the list of all measures, and column views materializing the hierarchies.

Apart from column views, building a calculation view also creates the necessary metadata that makes this view consumable by external tools.



When you create or modify an calculation view with the SAP Web IDE for SAP HANA, the design-time object is located in a project within your workspace. This workspace is linked to your SAP Web IDE user and is not visible to other users in the same SAP HANA system.

When you build a file with the SAP Web IDE, SAP HANA creates the corresponding runtime objects in a container, which is an abstraction layer for a database schema. This container is specific to the HDB module of your project.



Note:

You will learn more about the information model's lifecycle and SAP HANA Deployment Infrastructure (HDI) later on, in a dedicated unit, Management and Administration of Models.

Analytical Versus Transactional Requirements

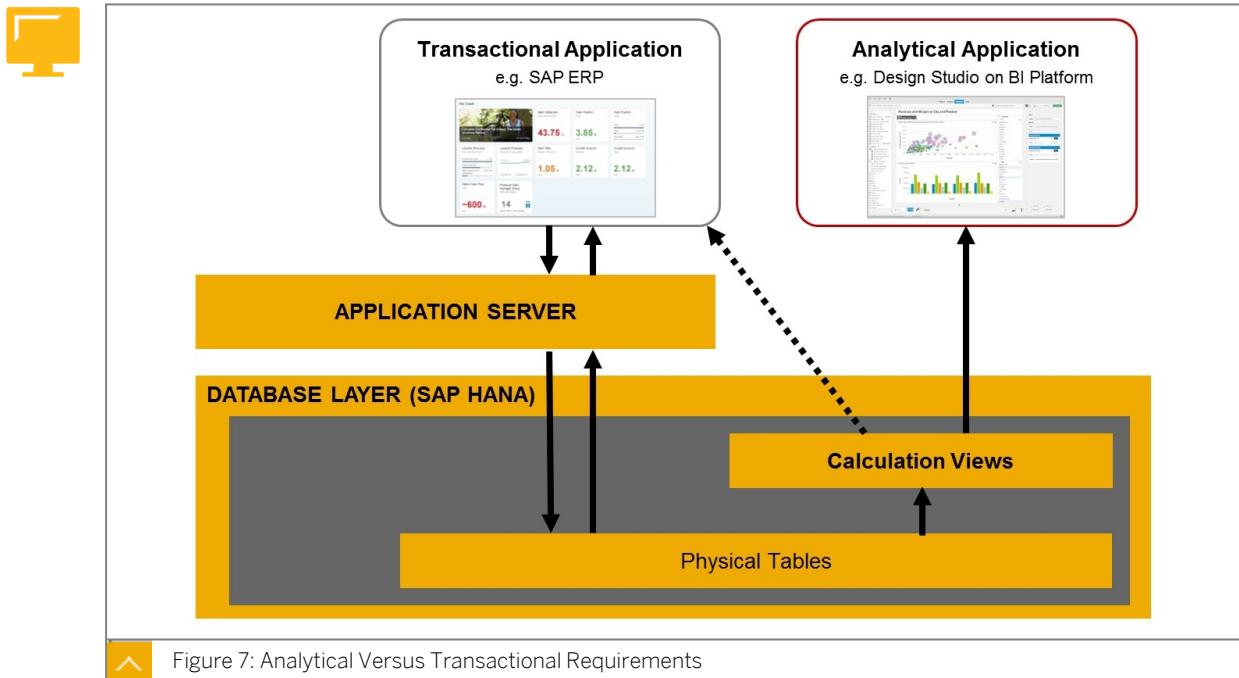


Figure 7: Analytical Versus Transactional Requirements

In transactional applications, such as SAP ERP, the underlying data (stored in physical tables) is generally handled by the application server. This layer is necessary to handle the business process, however complex it can be, while still ensuring data consistency at any time when updating multiple tables and checking the authorizations of the user.

On the other hand, calculation views are used only to retrieve data, without making any changes to the source transactional data.

For this reason, as shown in the figure, Analytical Versus Transactional Requirements, the analytical process can bypass the application server, which makes it even faster. The reporting tools directly query the SAP HANA calculation views, where data is calculated on-the-fly.



Note:

In some cases, the transactional application can also leverage the real-time capabilities of SAP HANA calculation views by displaying inside the transactional application (for example in the SAP Fiori launchpad) real-time analytical data that is relevant for the end user.

Calculation Views and the SAP HANA Engines

The index server of SAP HANA provides several engines to process queries against calculation views.

Main SAP HANA Engines for Executing Calculation Views



- Join Engine
- OLAP Engine
- Calculation Engine

The distribution of the overall view execution between these engines is complex, and out of the scope of this course.

However, you can keep in mind that optimization processes exist in order to delegate the execution of a calculation view to the most relevant engine. In a later unit, Optimization of Models, you will learn basic concepts about how to ensure the best possible optimization.

Querying Calculation Views Data

When building a calculation view, the SAP Web IDE for SAP HANA allows you to preview the data of the view or to execute a custom query on top of the view. It is important to understand the difference between these approaches.

The screenshot illustrates the SAP Web IDE interface for querying calculation views. At the top, a yellow bar reads "Standard Data Preview in Web IDE" with tabs for Analysis, Hierarchies, and Raw Data. Below this, a table shows rows of data with columns: COUNTRY, PRODUCT_ID, BP_COMPANY_N..., GROSS_AMOUNT, and RANK. A "Custom SQL Query" section is overlaid, featuring a "Raw Data" tab, a "Type to filter" input, and a "SQL" button highlighted with a red box and number 1. A modal window titled "Edit SQL Statement in SQL console" contains a SQL statement: "SELECT TOP 1000 [COUNTRY], [PRODUCT_ID], [BP_COMPANY_NAME]". Below the modal, a "Run" button is highlighted with a green box and number 3. The SQL code itself is numbered 2. To the right, a "Custom Result" table displays the same data as the preview, with one row highlighted and a "Check Results" button highlighted with a yellow box and number 4.

Row	COUNTRY	PRODUCT_ID	BP_COMPANY_N...	GROSS_AMOUNT	RANK
1	ZA	HT-1037	African Gold And Diamar	25676444.48	1
			African Gold And Diamar	4013066.8	2
			African Gold And Diamar	97435.28	3
			African Gold And Diamar	33698.32	4
			African Gold And Diamar	24223.84	5
			PicoBit	12838222.24	1

Row	COUNTRY	PRODUCT_ID	GROSS_AMOU..	RANK
AR	HT-1037	25676444.48	1	
AR	HT-1106	2441955.68	2	
AR	HT-1137	1869144.72	3	
AR	HT-1107	303284.64	4	
AR	HT-1107	5644.90	5	
AT	HT-1037	25676444.48	1	
AT	HT-1021	4013066.8	2	

Figure 8: Standard Preview or Custom Query

- Standard Data Preview

With the standard data preview, you select all the columns that are included in the semantics of the calculation view (provided that they are not hidden).

You can move the columns (using drag and drop), apply a temporary filter on one or several attribute columns, and order the result set by one (and only one) column.

- Custom SQL Query

An alternative to the standard data preview is to execute a custom SQL query. As shown in the figure above, you can modify the default SQL statement corresponding to the data preview in an SQL console, for example, change the selected columns or the GROUP BY clause, and order the result set by one or more columns.

This is particularly useful to perform a thorough test of calculation views with a complex scenario (stacked calculation views, counter measures, or join between several aggregation nodes with different GROUP BY columns).

**Note:**

Calculation views behave differently depending on which columns are selected, or whether you explicitly define a group by or not. You must ensure that a view does not give the wrong results if it is not correctly queried upon, or document the view so that it is correctly consumed by end users with reporting applications.

Standard Data Preview Features

Even though the Data Preview Editor is not a reporting tool, it still offers analysis functionality that can be useful during modeling or troubleshooting. It is comprised of the following tabs, each offering specific capabilities:

Table 2: Data Preview Tabs

Tab	Displays	Use Case
Raw Data	All data	Basic display of contents
Analysis	Selected attributes and measures in tables or graphs	Profiling and analysis
Hierarchies	Hierarchy tree with basic navigation	Navigating hierarchies in DIMENSION or CUBE Calculation Views without additional front-end tool

A setting in the SAP Web IDE for SAP HANA defines whether a filter must be set before fetching view or table data. To check or change this setting, choose *Tools → Preferences → Data Preview*.

If the option is selected, the default data preview is not executed immediately when you choose *Data Preview*.

Deferred Default Query Execution

Deferring the default query execution can be useful in the following situations:

- When the user wants to apply additional criteria to the data preview.
For example, to set an additional filter on one or several columns (measures or attributes).
- When the user wants to execute a custom query derived from the standard data preview query and does not need to execute the standard data preview.

**Note:**

Even with the option *Require that at least one filter be set before fetching view or table data selected*, it is still possible to execute the data preview without any filter. A dialog box is displayed to confirm that you want to fetch the data anyway.

Suppose you are testing a very complex view, and – just temporarily – you want to reduce the result set to one specific order ID. Executing the default query in this case is not useful, and might cause you to lose time. Instead you can perform the following steps:

1. In the SAP Web IDE for SAP HANA preferences, select the *Require that at least one filter be set before fetching view or table data* option.
2. Define a filter on the *ORDERID* column.
3. Click *Refresh Data*.



LESSON SUMMARY

You should now be able to:

- Explain modeling objects

Understanding the Different Types of Views



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Explain the types of views used in graphical modeling

Calculation Views

Before creating calculation views, you must fully understand the purpose of each type of calculation view so that you can choose the type that meets your requirements.

SAP HANA supports the following types of graphical calculation views:

Calculation View Type	Properties	Default Upper Node
SQL Access Only	For use as a modeled data source to other calculation views. No multidimensional support. Never exposed to any client tool. Can be accessed directly from SQL.	Projection
DIMENSION	Used to create re-usable master data views for standalone consumption or to be used in CUBE with Star Join types in order to add dimensions. No multidimensional support.	Projection
CUBE	Designed for data analysis with multidimensional reporting, but no dimensions allowed.	Aggregation
CUBE with Star Join	Similar to a CUBE dimension view, but the upper node is a Star Join where you join dimensions using calculation views of type DIMENSION.	Star Join

SQL Access Only Calculation View Type

A graphical calculation view defined with the SQL Access Only data category is meant to be used to generate intermediate views that are consumed by other calculation views (of any type). These calculation views are never exposed to client tools, so when a user is browsing a

list of available calculation views to use in their report, they will never see these types of calculation views.

SQL Access Only calculation views can contain both attributes and measures. You can force aggregation behavior for the measures using aggregation nodes within this calculation view type, but unlike CUBE or CUBE with star join, the resulting data set is not automatically aggregated to the columns requested in the external query. For example, if an external query requests only country and sales revenue but the result contains country, product, and sales revenue, a CUBE or CUBE with star join would automatically aggregate the sales revenue by country. An SQL Access Only calculation view would also return only country and sales revenue but would not aggregate the sales revenue for each country. However, it would produce a list of all sales in each country and for each product.

The main use case for this type of view is when you create views that will be reused in other views but do not need to be accessed by the end user.

In order to provide reporting tools with optimized views for hierarchical reporting, during the activation of a calculation view, multiple runtime column views are generated and activated. The generation and activation of these extra views can be time consuming. This means builds can take quite some time for complex models. If your calculation view will never be exposed to a reporting tool then the extra optimized views are not used. An SQL Access Only calculation view never generates these extra views and only generates the single view that is consumed internally by other calculation views.

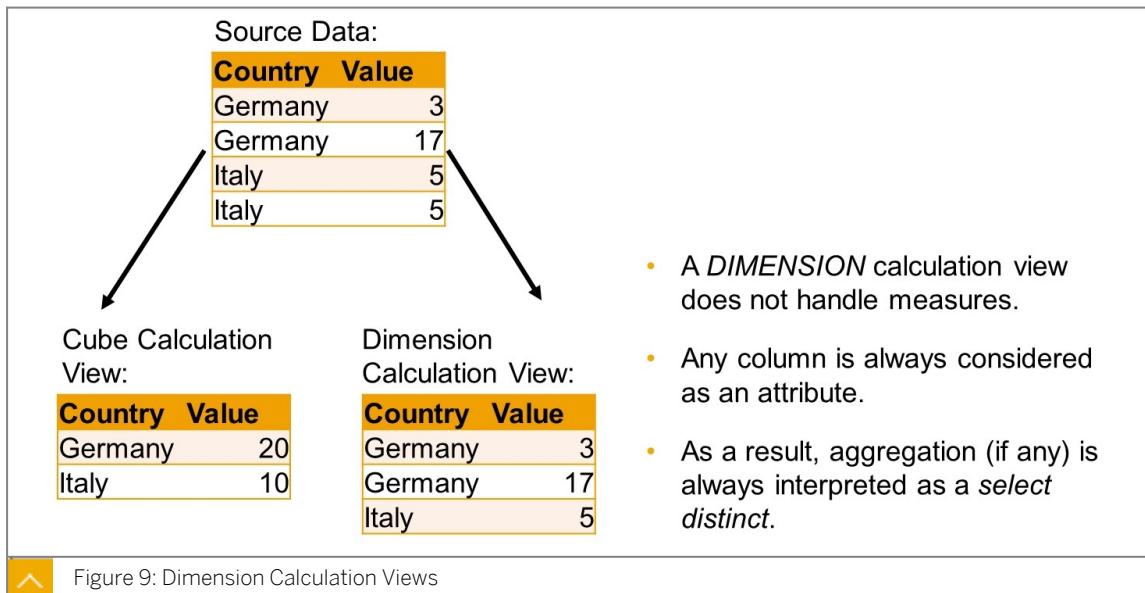
As the name suggests, it is possible to access SQL Access Only calculation views directly using SQL. For example, they can be accessed from a table function or procedure.



Note:

Prior to SAP HANA 2.0 SPS04, SQL Access Only calculation views were known as 'DEFAULT' type. The functionality did not change, only the name.

DIMENSION Calculation Views



A DIMENSION calculation view is used to expose master data. You can combine multiple data sources with filters, calculated attributes, and hierarchical models into a reusable model. This type of calculation view can be consumed directly by many clients, where simple master data

list reporting is required, but is most often used to provide dimensional information to a CUBE type of calculation view.

In a DIMENSION calculation view you cannot define a measure; any numeric columns or values are treated as attributes and are used to describe an entity, such as a *price* of a product, or age of a person.

You can specify aggregation on any attribute and this will produce a distinct list of values.

CUBE Calculation View

When you want to create a view that includes measures, you use a calculation view of the type CUBE. By default, the result of this type of view will always be aggregated by the attributes requested by the external query. So, even though the calculation view may be able to provide many attributes and measures, the measures are always automatically aggregated by the attributes that were requested by the query. This behavior is different to the SQL Access Only calculation view type.

This type of calculation view is optimized for OLAP style analysis, where slice and dice is required over the measures by any combination of attributes within the model.

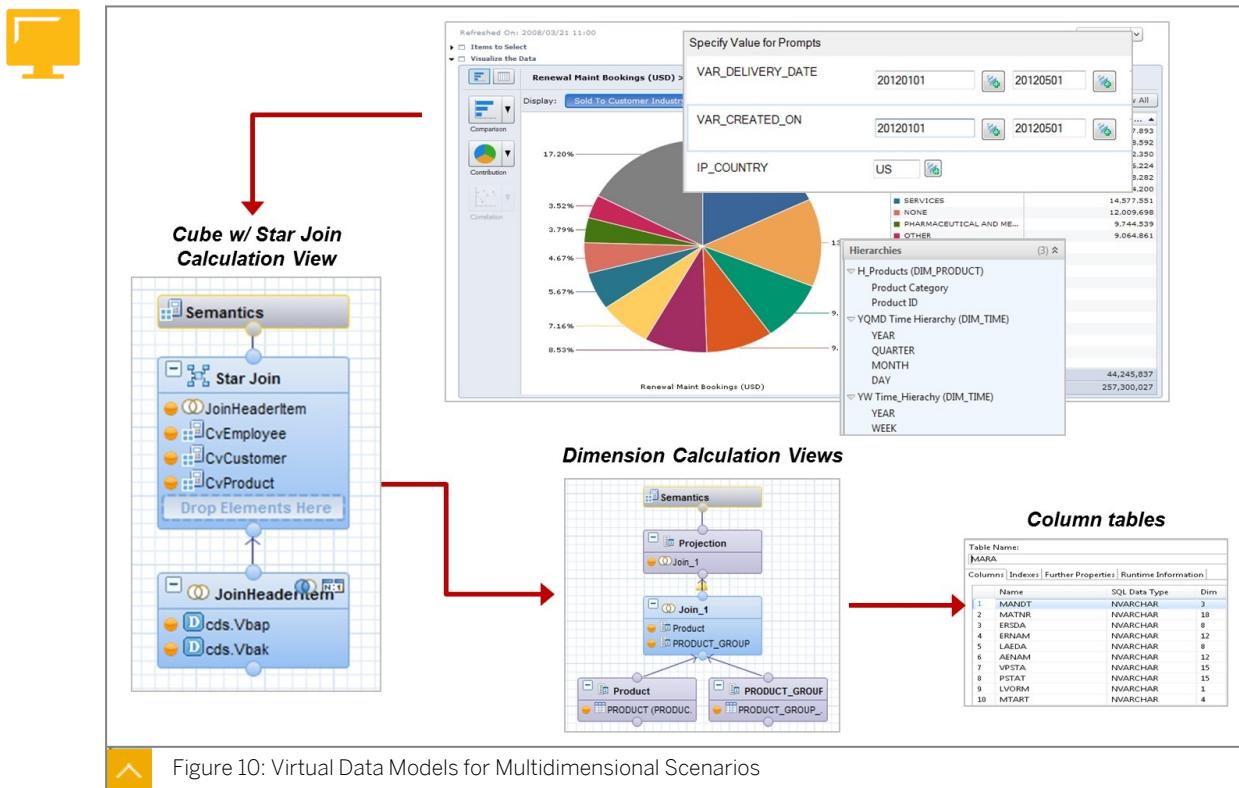
A CUBE calculation view does not allow dimensions (based on DIMENSION type calculation views) to be included. All attributes used in analysis come from within the defined model.

CUBE Calculation View with Star Join

An extension to the basic CUBE type of calculation view is the CUBE with Star Join.

In addition to the capabilities of the CUBE type of calculation view, a CUBE with Star Join calculation view allows you to join DIMENSION calculation views to the basic model so that you significantly expand the analysis capabilities. For example, if you create a basic sales cube which provides only limited attributes such as customer and product, you could then join the PRODUCT dimension calculation view to provide large numbers of useful attributes that further describe the customer or product. You could then aggregate the sales revenues of the cube by customer industry, product color, and so on, rather than by the dimensions.

Virtual Data Models for Multidimensional Scenarios



In this scenario, a Cube with Star Join calculation view enables a multidimensional reporting that leverages the source data (from the Vbap and Vbak tables) and DIMENSION calculation views created for the main dimensions, such as *Product*, *Product Group*.

Multidimensional tools support hierarchies for navigation, filtering and aggregation, as well as prompts (variables and input parameters) for efficient pre-filtering of data.

Supported Data Source Types in Graphical Calculation Views

The following is a list of the main data source types that are supported in SAP HANA calculation views.



Table 3: Supported Data Source Types in SAP HANA Calculation Views

Data Source Type
Column Table
Row Table
Calculation Views
SQL Views
Table Functions*
Virtual Tables

* Table functions must be located in the same database if using a multi-tenant system. All other data source types can be in other databases within the multi-tenant system.

**Caution:**

If a table has been created INSIDE an HDI container using an SQL statement (that is, without using a design-time file of type `.hdbtable` or `.hdbccls`), you might see this table as an available data source when creating a calculation view, but the build of the calculation view will fail.

As an aside, you will learn in the unit, Modeling the Persistence Layer, that this way of creating a table (without a design-time file) in an HDI container is not recommended at all.

Row or Column Table

To identify whether an existing table in SAP HANA is a column table or a row table, you have the following options:



- From the Database Explorer, connected to an HDI Container or the *Catalog* view of a classical schema:
 - For a table: check the table icon, or open its definition (right-click the table and choose *Open*) and check the *TYPE* field.
 - For a synonym: open its definition (right-click the synonym and choose *Open*). The definition mentions *For Table (Column Store) <target table name>* or *For Table (Row Store) <target table name>*
- From the SQL Console:
 - Query the system table `M_TABLES`.

Example:

```
SELECT "SCHEMA_NAME", "TABLE_NAME", "TABLE_TYPE" FROM "M_TABLES"
WHERE "SCHEMA_NAME" = 'TRAINING'
AND "TABLE_NAME" = 'SALES_DATA'
```

Table Functions

Table functions can be used to define complex data sources using SQL Script. These data sources can then be consumed by a calculation view.

As a general rule, you should always try to use the standard functionality of the calculation view using the graphical editors but sometimes you might need to revert to code to produce the desired outcome.

DIMENSION Calculation Views

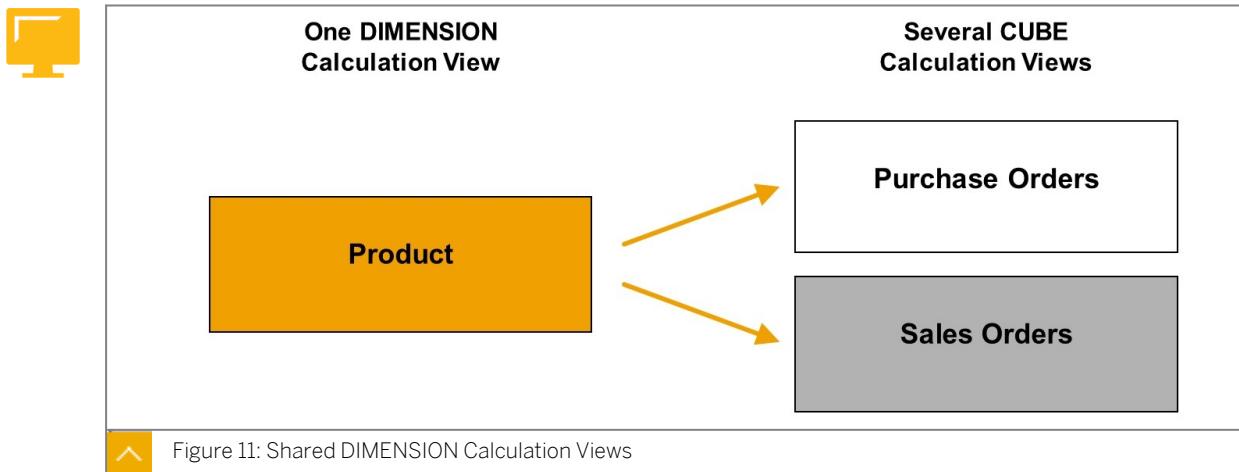
DIMENSION calculation views are used to provide context. This context is provided by master data tables which give meaning to data.

For example, if data in a car sales database only contains a numeric ID for each dealer, you can use a DIMENSION calculation view to provide information about each dealer. Using this method, you could then display the names and addresses of the car dealers and many other useful descriptive attributes that give context to the data.

DIMENSION calculation views are used to select a subset of columns and rows from master data tables. DIMENSION calculation views are not valid for measures. They can contain only attributes.

A DIMENSION calculation view does not need to be based on a single table. On the contrary, you can use them to join master data tables to each other to create complex views. For example, to join *Products* to *Product Categories*.

Shared DIMENSION Calculation Views



DIMENSION calculation views are reusable objects and can be shared between several CUBE calculation views.

For example, the product attribute view can be used both in a purchase order CUBE calculation view and in a sales order CUBE calculation view.

Supported Characters for Views Names and View Objects

SAP HANA supports all the Unicode characters in object names; that is, views, columns, input parameters, and so on.

However, a number of characters are forbidden in the object names. These characters are \ / : * ? " < > | . ; ' \$ % , ! # + and the **space** character.

Supported Types of Nodes

DIMENSION calculation views support most types of nodes, such as Projection, Join, Union, Aggregation, and Ranking. This allows a very flexible design of DIMENSION calculation views. The most commonly used nodes are Join nodes (to join master data tables) and Projection nodes (to filter data, select specific columns from the master data tables, and create calculated attribute columns).

Calculated Attributes

It is possible to create additional calculated columns in a DIMENSION calculation view.

For example, you have two columns containing the first and last name of the customer, but you would like to have all this information (first and last name) in a single column. You can do this by creating a calculated column based on string manipulations.

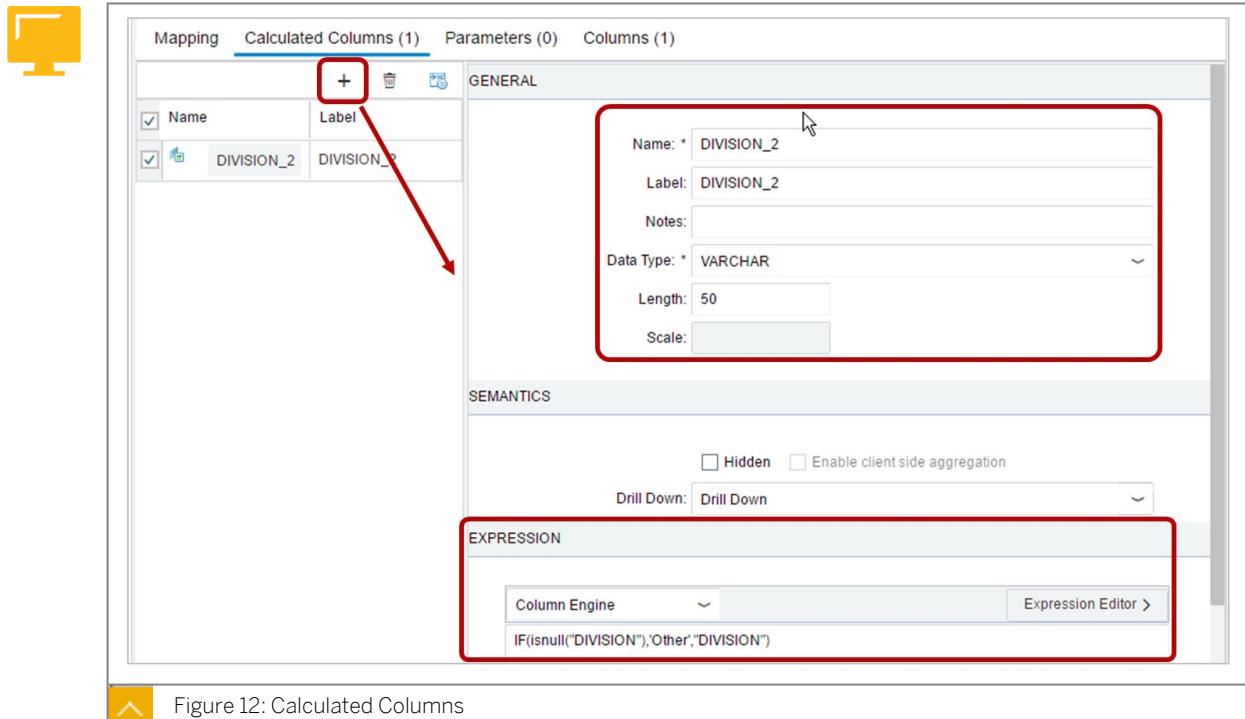


Figure 12: Calculated Columns

- The calculation can be an arithmetic or just a character string manipulation.
- Calculated columns also support non-measure attributes as part of the calculation.
- It is possible to nest calculated columns, so that one calculated column in turn is based on other calculated columns.

Measures in Calculation Views

Compared with attributes, measures bring an additional level of complexity to calculation views. In particular, you must define the relevant behavior of views when measures are aggregated based on a number of attributes.



Note:

Attributes are regular columns and, generally, they do not need to be aggregated. Aggregating data in a DIMENSION calculation view would just be like a *Select Distinct* statement.

In SAP HANA, two types of graphical modeling objects handle measures:

- Calculation views of type *CUBE*
- Calculation views of type *CUBE with Star Join*

In a calculation view, the **measures can originate from several data sources**, that is, several tables (local tables or via a synonym), calculation views, table functions, and so on.

For example, you can join two different tables and extract measures from both of them.

Standard Preview or Custom Query

When you design a calculation view, you have to keep in mind that the output data depends heavily on the query that the client tool executes on top of the view. Common criteria impacting the result set include the following:

- Selected attributes (and measures)
- Aggregate functions applied by the client query
- Ordering defined on one or several columns

The screenshot shows the SAP HANA Web IDE interface. At the top, there's a header bar with tabs for Analysis, Hierarchies, and Raw Data. Below it is a table titled "Standard Data Preview in Web IDE" showing rows of data with columns for COUNTRY, PRODUCT_ID, BP_COMPANY_NAME, GROSS_AMOUNT, and RANK. A red box highlights the "Raw Data" tab. In the center, there's a "Custom SQL Query" section with a "SQL" button highlighted with a red box. Below it is a SQL statement: "SELECT TOP 1000 [COUNTRY], [PRODUCT_ID], SUM([GROSS_AMOUNT]) AS [GROSS_AMOUNT], SUM([RANK]) AS [RANK] FROM [HA300_03_HDI_CONTAINER 1].[HA300::CVC_SO_RANK] GROUP BY [COUNTRY], [PRODUCT_ID] ORDER BY 1,4;". A yellow box labeled "Edit SQL Statement in SQL console" is overlaid on this area. On the left, there's a "Run" button with a green play icon highlighted with a red box. To the right, there's a "Custom Result" table showing the same data as the preview, with a red box highlighting the "Run" button. A yellow box labeled "Check Results" is overlaid on the result table.

RB	COUNTRY	RB	PRODUCT_ID	RB	BP_COMPANY_N...	12	GROSS_AMOU...	12	RANK
	1	ZA	HT-1037	African Gold And Diamar	25676444.48			1	
				African Gold And Diamar	4013066.8			2	
				African Gold And Diamar	97435.28			3	
				African Gold And Diamar	33698.32			4	
				African Gold And Diamar	24223.84			5	
				PicoBit	12838222.24			1	

RB	COUNTRY	RB	PRODUCT_ID	RB	GROSS_AMOU...	12	RANK
AR	HT-1037		25676444.48		1		
AR	HT-1106		2441955.68		2		
AR	HT-1137		1869144.72		3		
AR	HT-1107		303284.64		4		
AR	HT-1107		36644.96		5		
AT	HT-1037		25676444.48		1		
AT	HT-1021		4013066.8		2		

Figure 13: Standard Preview or Custom Query

To check the behavior of a calculation view that has a complex calculation scenario, we recommend that you create custom SQL queries on top of the view and test several scenarios. This is especially useful when some of the measures, or all of them, cannot be aggregated with the common Sum function, for example when working with ratios, averages, counter columns, and rank nodes.

This approach enables you to enhance the design of the view, if possible, or at least to understand how end users must query the view to get relevant results.



Caution:

Always keep in mind that calculation views in SAP HANA have a special behavior because they are instantiated at runtime, which means that their execution depends on the query that is run on top of them by the front-end tool (SQL console, reporting tool, and so on).



LESSON SUMMARY

You should now be able to:

- Explain the types of views used in graphical modeling

Working with Common View Design Features



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Use common features to design calculation views

Semantics Node

Every calculation view has a *Semantics* node. You do not add this, it is already present and will always be the top node, regardless of the data category of the calculation view. In this node, you assign semantics to each column in order to define its behavior and meaning. This is important information used by consuming clients so that they are able to handle the columns appropriately.

One of the most important settings for each column is its column type. You can choose between **attribute** or **measure**.

In the semantic node, you can also optionally assign a **semantic type** to each column. A *semantic type* describes the specific meaning of each column and this can be helpful to any client that consumes the calculation view by enabling it to then represent the columns in the appropriate format. For example, if you define a column as a date semantic type, the front-end application can then format the value with separators rather than a simple string. The key point is that it is the responsibility of the front-end application or consuming client to make use of this additional information provided by the semantic type.

Semantic Attributes

Semantic types for **attributes** can be defined as:

- Amount with Currency Code
- Quantity with Unit of Measures
- Currency Code
- Unit of Measure
- Date
- Date – Business Date From
- Date – Business Date To
- Geo Location - Longitude
- Geo Location - Latitude
- Geo Location - Carto ID
- Geo Location – Normalized Name

Semantic Measures

Semantic types for **measures** can be defined as:

- Amount with currency code
- Quantity with unit of measure

Apart from the *semantic type* there are other important settings that can be defined for each column such as the following:

- Assigning a description column to another column - for example, assigning the product id column to a product description column so a user sees a value that is more meaningful.
- Hiding a column - can be used if a column is only used in a calculation, or is an internal value that should not be shown, for example, hiding the unhelpful *product id* when we have assigned a description column that should be shown in its place.
- Assigning a variable - allowing a user to select a filter value at runtime for the attribute

One of the most frequently maintained values in the *Semantics* node is the name and label of the column. It is possible to define an alternative name and label to any column so that it will make more sense to a user than what was originally proposed from the data source. For example, who wants to see the word *MATNR* in a business report column heading when we really should be seeing the words *Material Number*?

Column Properties in the Semantics Node

In the *Semantics* node, you can define and review the properties assigned to columns that are in the output of the top node.

Table 4: Column Properties

Property Name	See Lesson
Aggregation Type	See lesson <i>Aggregation node</i>
Conversion Function	Conversion functions used for inbound/outbound input parameters values (optional)
Data Masking	See lesson <i>Masking Sensitive Data</i>
Data Type	See <i>Same Lesson</i>
Hidden	See <i>Same Lesson</i>
Label	Column Label
Label Column	See <i>Same Lesson</i>
Name	Column name
Related Attributes	
Semantics	See lesson <i>Conversion, Same Lesson</i>
Sort Direction	See <i>Same Lesson</i>
Type	See <i>Same Lesson</i>
Unconverted	See lesson <i>Conversion</i>
Variable	See lesson <i>Using variables and input parameters</i>

Property Name	See Lesson
Default Value	See lesson <i>Using variables and input parameters</i>
Display Folder	Organize attributes into folders for BI metadata (new in 2.0 SPS03)
Drill-Down Enablement	Property for calculated Columns
Hierarchy Default Member	See lesson <i>Using Hierarchies</i>
Info Object	
Keep Flag	See lesson <i>Aggregation node</i>
Key	Identifies key attributes, especially in DIMENSION calculation views
Mapping	Origin of the column (origin node name, column name)
Notes	Free text
Null Handling	Whether Null values in columns should be replaced with a specific default value
Presentation Scale	Whether a presentation scale must be applied (new in 2.0 SPS03)
Transparent Filter	See lesson <i>Aggregation node</i>

Base Table Alias



- When adding multiple instances of the same table to a view, an alias will be proposed.
- It is possible to modify the alias name in the data source properties (*Mapping* tab).

Figure 14: Adding Multiple Base Tables Using Aliases

In some cases, you need to use the same base table more than once in the same calculation view. In this case, you can define a table alias for any additional instance of the same source table.



Note:
The SAP Web IDE for SAP HANA automatically suggests an alias, but you can choose your own.

Hidden Columns



- To hide a column, select the *Hidden* checkbox of the column in the *Semantics* node.
- The column will not be exposed to reporting tools but can be used within the calculation view itself, for example in calculated columns.

Semantics

View Properties **Columns (9)** Hierarchies (0) Parameters (0)

Type	Name	Hidden	Data Type	Semantics	Conversion Function
LANGUAGE	LANGUAGE	<input type="checkbox"/>	NVARCHAR(1)		
FIRST_NAME	FIRST_NAME	<input checked="" type="checkbox"/>	NVARCHAR(40)		
LAST_NAME	LAST_NAME	<input checked="" type="checkbox"/>	NVARCHAR(40)		
FULL_NAME	FULL_NAME	<input type="checkbox"/>	VARCHAR(100)		

Figure 15: Hidden Columns

You can choose to hide the attributes that are not required for client consumption. This typically occurs in the following scenarios:

- When an attribute column is calculated using a complex computation of source columns or intermediary calculated columns, you can hide any column that you do not want to expose to the end user.

In the figure, *Hidden Columns*, for example, the *FULL_NAME* column is a concatenation of the first and last name (calculated column), and the source columns *FIRST_NAME* and *LAST_NAME* have been hidden.

- When a column that contains a description has been assigned to the corresponding code, name or ID column as a *Label Column*, you can hide this column. This action is particularly useful in scenarios where the reporting tools are able to interpret this code, name or ID to description assignment.

For example, SAP BusinessObjects Analysis and SAP Design Studio are able to expose *Label Columns*. The code, name, or ID, and the label, are generally referred to as *Key* and *Text*, respectively.



Note:
Only a column that is added to the output of the top node of a calculation view can be defined as a *Label Column*.

Label Columns and Hidden Attributes



- Label columns can refer to hidden attributes
- You define label columns and hidden attributes in the *Semantics*

Semantics

View Properties Columns (12) Hierarchies (1) Parameters (0)

Type	Name	Label Column	Hidden	Data Type	Semantics
BP_COMPANY_NAME	BP_COMPANY_NAME		<input checked="" type="checkbox"/>	NVARCHAR(80)	
BP_ID	BP_COMPANY_NAME		<input type="checkbox"/>	NVARCHAR(10)	
CLIENT			<input type="checkbox"/>	NVARCHAR(3)	
PARTNER_GUID			<input type="checkbox"/>	VARBINARY(16)	

When reporting on an information model containing a label column, with a tool that supports Label Columns:

- Label is displayed as "Text".
- The base column is displayed as "Key"

Figure 16: Label Columns and Hidden Attributes

Sorting the Data Set

SAP HANA 2.0 SPS03 introduces the possibility to sort the result set (output) of any calculation view. This enables you to define a sort order that will apply when none is specified by the client query that is executed on top of the calculation view, thus guaranteeing a stable result order.

It can also be useful when previewing the data for the purpose of testing your calculation views.



Option 1: Use the Sort Direction Property

Semantics

View Properties Columns (6) Hierarchies (0) Parameters (0)

Private

Type	Name	Sort Direction	Unconverted
CLIENT			
SO_ID		ASC	
SO_ITEM_POS		ASC	
PRODUCT_ID			
CURRENCY_CODE			
GROSS_AMOUNT		DESC	

Option 2: Use the Sort Result Set dialog box

Semantics

View Properties Columns (6) Hierarchies (0) Parameters (0)

Private

- 1 Click the **Sort** button in the toolbar.
- 2 Open the **Sort Result Set dialog box**.
- 3 Select the column to sort.
- 4 Set the sort direction (e.g., Ascending).
- 5 Close the dialog box.

Name	Sort Direction
SO_ID	Ascending
CLIENT	
SO_ID	
SO_ITEM_POS	
PRODUCT_ID	

Figure 17: Sorting the Result Set

Defining a sort order is done in the *Columns* tab of the *Semantics* node, and can be achieved in two different ways:

- In the *Sort Direction* property:

You can directly define a sort direction for one or several columns. In case you specify more than one column, the columns are taken into consideration in the order you defined them. For example, in the figure above, *SO_ID* first, and *SO_ITEM_POS* second.

- In the *Sort Result Set* dialog box:

You can add one or several sort columns and define the sort direction for each of them.

In addition, this dialog box allows you to modify the order of columns. For example, in the image above, decide if the sort order should be *PRODUCT_ID* then *SO_ID* or the other way round.



Note:

It is technically possible to include a hidden column in the list of columns used to sort a result set. However, this generally does not make sense.



Caution:

When the client query defines itself a sort order on different columns, or only a part of the sort columns used in the calculation view design to order the result set, you cannot be sure that the original order defined in the calculation view on other columns is applied. For example, if the calculation view sorts the result set by *SO_ID* then *PRODUCT_ID* and a front-end tool executes a query ordered by *PRODUCT_ID*, you will not necessarily have your result set actually sorted by *PRODUCT_ID* then *SO_ID*. To get this result, the front-end query needs to specify the two columns in the ORDER BY clause.

General Properties of Views

For each view, you can define a number of Properties in the *View Properties* tab of the *Semantics* node. Depending on the type of view, the list of available properties may vary.

The following table gives a list of these properties and a description. You will find more information on some of them in this course later on, and you can also consult the SAP HANA documentation, in particular the *SAP HANA Modeling Guide*.



Table 5: Properties of Views — General

Property	Description
Data Category	For calculation views, this determines whether the view supports multidimensional reporting.
Type	<i>Standard</i> or <i>Time</i>
Run With	Defines how to apply security when executing a Calculation View.
Default Client	Defines how to filter data by SAP CLIENT (aka MANDT).
Apply Privileges	Specifies whether analytic privileges must apply when executing a view (SAP Web IDE for SAP HANA modeling supports SQL analytic privileges only).
Default Member	Defines the default member to be used for all the hierarchies of this calculation view.

Property	Description
Deprecate	Identifies views that are not recommended for reuse, though still supported in SAP HANA Modeler.
Enable Hierarchies for SQL Access	In a calculation view of type CUBE with star join, determines whether the hierarchies defined in shared dimension views can be queried via SQL.
End User View	Can be used to define if the view should be available for reporting to the end user
Generate technical MDX hierarchies	Specifies whether to generate the technical hierarchies for MDX consumption
Enable History	The value of this property determines whether your calculation view supports time travel queries.
History Input Parameter	Specifies which input parameter must be used to specify the timestamp in time travel queries.



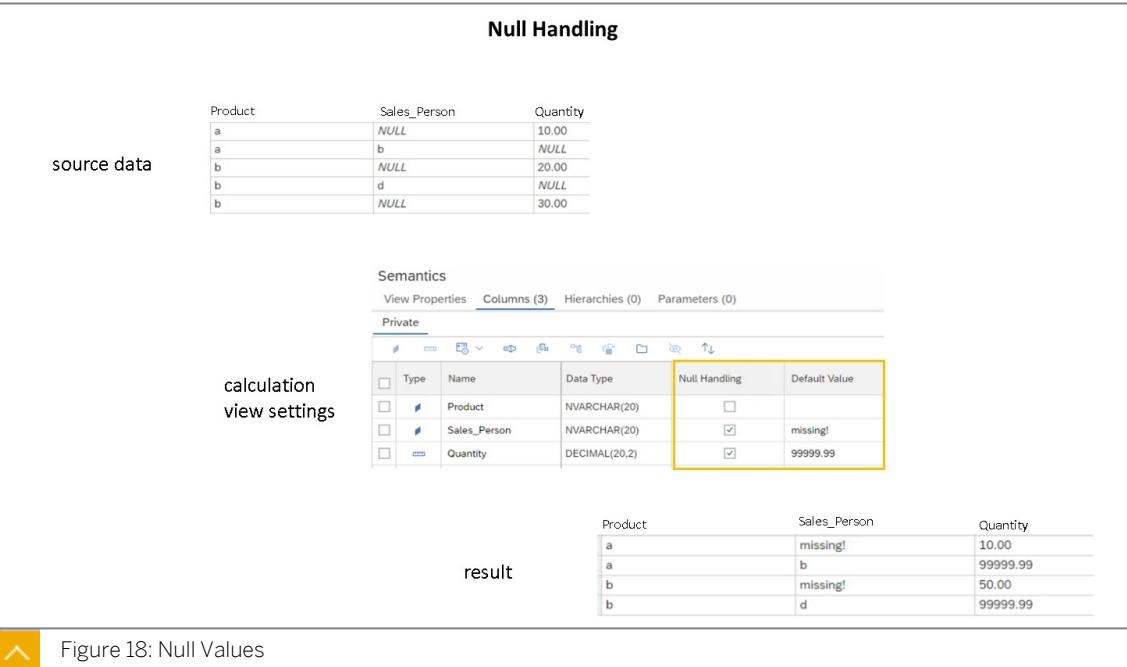
Table 6: Properties of Views — Advanced

Property	Description
Propagate instantiation to SQL Views	If the calculation view is referenced by an SQL view, determines whether the calculation view must be instantiated (for example, by pruning columns that are not selected by the above queries) or executed exactly as it is defined.
Cache	Defines whether the data retrieved by the view should be cached.
Analytic View Compatibility Mode	If this setting is activated, the join engine ignores joins with cardinality n..m defined in the star join node when no column at all is queried from one of the two joined tables.
Ignore Multiple Outputs For Filter	Optimization setting to push down filters even if a node is used in more than one other node.
Pruning Configuration Table	Identifies which table contains the settings to prune Union nodes.
Execute in	Determines whether the model must be executed by the SQL engine or column engine.
Cache Invalidation Period	If the view data are cached, this determines whether the cache must be deleted on a daily or hourly basis, or after each transaction that modifies any of the underlying tables.
Execution Hints	This property is used to specify how the SAP HANA engines must handle the calculation view at runtime.

Handling Null Values

Columns, both attributes and measures, can contain undefined values or null values. This means the output usually displays the word *null*.

However, in the calculation view you can choose a default value that replaces the null value in the output.



The screenshot illustrates the SAP Web IDE interface for defining null values in a calculation view. It is divided into three main sections:

- source data:** A table showing five rows of data with columns Product, Sales_Person, and Quantity. The data is as follows:

Product	Sales_Person	Quantity
a	NULL	10.00
a	b	NULL
b	NULL	20.00
b	d	NULL
b	NULL	30.00
- calculation view settings:** A semantic node configuration window titled "Null Handling". It shows three columns: Type, Name, and Data Type. The "Null Handling" and "Default Value" columns are highlighted with a yellow box. The data is as follows:

Type	Name	Data Type	Null Handling	Default Value
Product	NVARCHAR(20)		<input type="checkbox"/>	
Sales_Person	NVARCHAR(20)		<input checked="" type="checkbox"/>	missing!
Quantity	DECIMAL(20,2)		<input checked="" type="checkbox"/>	99999.99
- result:** A table showing the transformed data based on the settings. The Sales_Person column now contains "missing!" for rows where it was NULL in the source data.

Product	Sales_Person	Quantity
a	missing!	10.00
a	b	99999.99
b	missing!	50.00
b	d	99999.99

Figure 18: Null Values

To define null values you must first choose the *Null Handling* check-box which is found in the semantic node under *Columns*. You are then able to enter a custom value in the field *Default Value*.

If you have enabled null handling for columns, and if you have not provided any default value, 0 (zero) will be the default value for numerical data types (decimal, integers).

For character type columns of data type VARCHAR and NVARCHAR, if you have not defined a default value after enabling null handling, an empty string (blank) will be the default value.



Note:

If you do not see the settings *Null Handling* and *Default Value*, go to *Settings* and add these columns.

Specific Features to Enhance Flexibility of Calculation View Design

Because calculation views offer a lot of flexibility, in particular the possibility to have a large number of nodes in a view, the SAP Web IDE for SAP HANA provides advanced functionality to manage nodes and semantics. The following features are available:

Feature	Purpose
Switching Node Types	Switch one node type with another without losing the link to the data source or the upper and lower nodes.
Replacing a Data Source	Replace one node source with another without losing the output columns, calculated columns, and so on, throughout all of the upper nodes. You can also delete an intermediate node without losing the calculation logic and semantics between its ancestor and descendant.

Feature	Purpose
Extract Semantics	Apply the semantics from an underlying node or data source to the <i>Semantics</i> node of the calculation view.
Propagate to Semantics	Map columns to the output across all upper nodes, up to the top node of the calculation view.
Previewing the Output of Intermediate Nodes	To troubleshoot problems in view design, preview the data of an intermediate node rather than the whole calculation view.
Map Input Parameters between nodes	Map input parameters of a view to input parameters defined in underlying (source) views of other nodes.

Switching Between Certain Types of Node

It is possible to convert a *Projection* node into an *Aggregation* node and the other way around without losing the reference to the data sources of the node.



Note:

Before SAP HANA 2.0 SPS03, this was only possible for the top node of a calculation view (that is, the last node of the calculation logic).

For the top node, it is also possible to switch to a *Star Join* node.

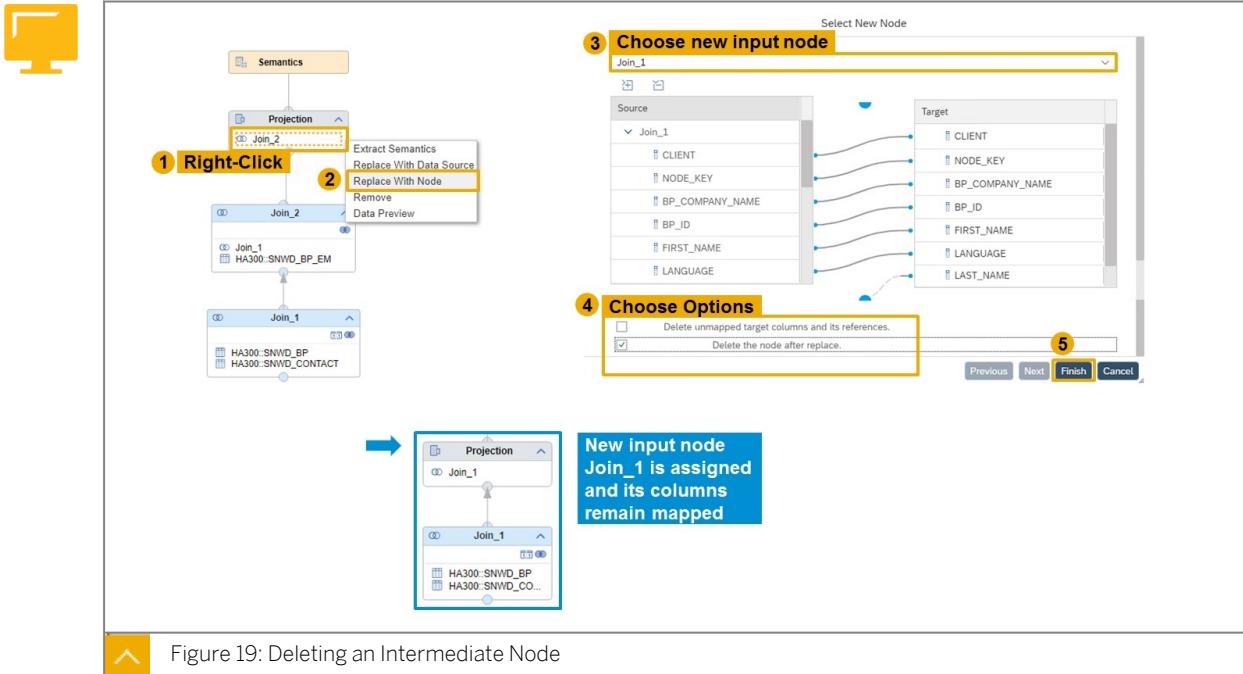
Replacing the Data Sources of a Node

Depending of the type of node and view configuration, you can replace the data source of a node (for example, a table, a Calculation view, or another node) by another data source (table, Calculation View).

This functionality is useful, for example, when you want to have the columns still propagated to all the upper nodes. You can also keep the existing calculated columns defined in this node working despite the change of data source.

When you do the change, a column mapping dialog box appears. It helps you assign the columns of the new data source to the output columns previously defined in the node.

When you want to **delete an intermediate node** and connect the direct descendant and ancestor, you proceed in a similar way after right-clicking the node's name in the direct descendant. You replace it with the direct ancestor node. This is shown in the figure, Deleting an Intermediate Node.



Extracting Semantics

It is possible to extract the semantics from the underlying data sources of the nodes, especially when these data sources are information views with rich semantics, and to propagate them to the semantics of the calculation view. In SAP HANA 2.0 SPS03, it is also possible to extract hierarchies and variable information from the included calculation views. To allow this, the corresponding columns must be available in the current view (the "extracting" view).

Propagate to Semantics

This feature can be used in views with multiple stacked nodes. The purpose is to map one or several columns, already in the output of a node, to the output of all upper nodes and up to the *Semantics* node, or in other words, the top node.

This is an alternative approach, faster than adding successively the same column(s) to the output of several stacked nodes.

To do this, select one or several columns in the output of a node. Then right-click the selection and choose *Propagate to Semantics*.

Previewing the Output of Intermediate Nodes

To understand and fine-tune the different steps of a calculation view, each node can be previewed independently from the calculation view itself.

To achieve this preview, the view must have already been successfully built.

Map Input Parameters between Nodes

You can map input parameters (and also variables) defined in a calculation view to the input parameters and variables defined in the source views. You can do this to pass the parameter value itself, but also to pass the relevant list of values to the input parameter dialog box at runtime (when the view is executed).

**Note:**

In the *Manage Mappings* view, you can even copy and map an input parameter from a source view.

That is, you do not need to create a new input parameter in the calculation view (and define all its settings) before mapping it. This task can be done automatically in just one operation.

Copy and Paste Part of a Scenario

It is possible to copy and paste a node within the *Scenario* pane of the same calculation view. If this node uses other nodes as its data sources, it is possible to decide whether you want to copy this node only, or the node and all nodes below, together with the connections between them.

Insert a Node between Two Other Nodes

In the Calculation View scenario, you can insert a new node between two existing nodes that are already linked. By doing this, you keep the columns consistent along the entire scenario.

Note that this is only available as a graphical feature: You must drag the new node from the palette and drop it exactly onto the existing link between the existing nodes.

For example, if you have started designing a CUBE calculation view where the aggregation node uses a join node as its data source. You have already mapped the required columns, defined the aggregate functions, semantics, and so on. Then you realize you need a projection to calculate a column before aggregating. In that case, by adding the new Projection between the Join node and the Aggregation node (instead of basically removing the Join node from the aggregation node and adding it to a new projection node), you keep most of the work you already made and save time.

Navigating to Source Calculation Views

With SAP HANA 2.0 SPS04, it is now possible to open a calculation view directly from calculation scenario of another view that consumes it. To do this, in the *Scenario* pane, right-click the data source (Calculation View) in the node that consumes it, and choose *Open*.

Working with Encrypted Columns

From SAP HANA 2.0 SPS05 onwards, Column Encryption is supported. In the model editors, in particular in the *Mapping* tab, the encrypted columns are flagged with a dedicated icon. This feature helps the modeler to make sure he does not define unsupported data processing (with regards to encryption) on these columns. It is also valuable to make sure security governance is applied in a relevant way during model development.

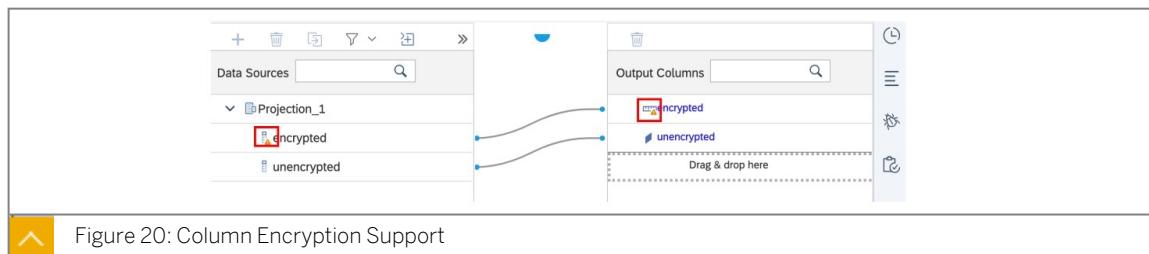
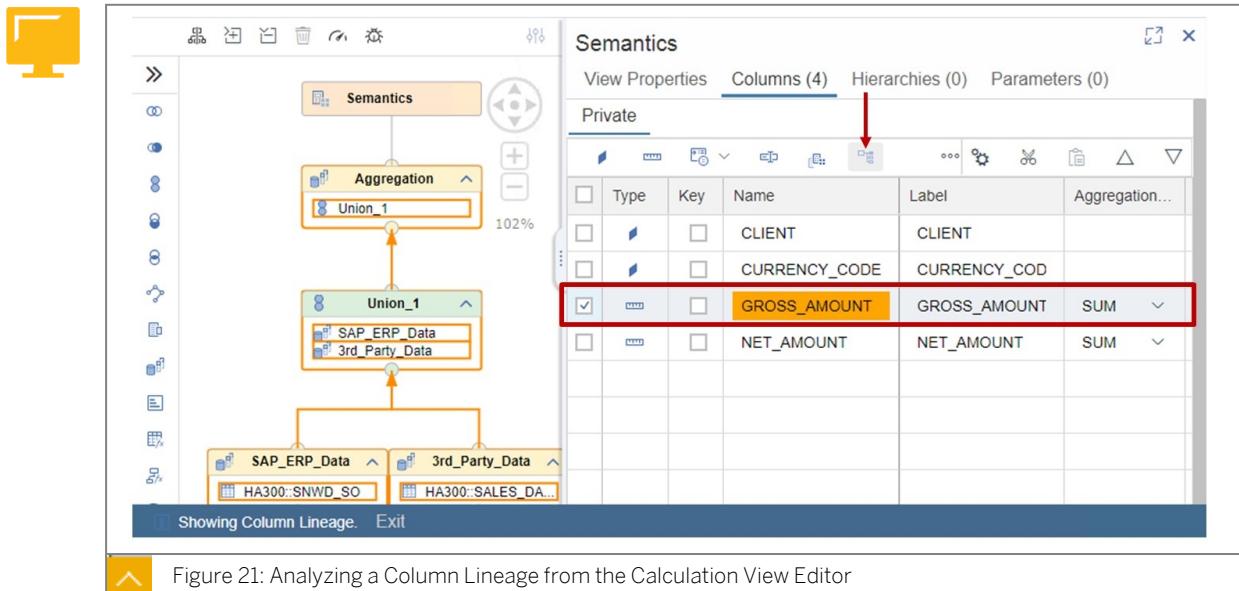


Figure 20: Column Encryption Support

To know more about column encryption, you can consult the SAP HANA Client-Side Data Encryption Guide on the SAP Help portal. Note that the client-side data encryption is not in

the scope of this course. Other techniques relating to data security are discussed in a later unit, Security in SAP HANA Modeling.

Analyzing a Column Lineage



The *Show Lineage* functionality enables you to track the origin of a column along the calculation scenario and down to the first node where it appears. This node can be a bottom-level node of the view where the column is selected from a source, such as another information view, a column table, and so on. It can also be an intermediate node where the column is calculated.

Show Lineage is a great feature for locating the origin of columns that might have been renamed during the flow.

Overview of the Possible Node Types

A calculation view can use as many nodes as you need. Each node has its own capabilities and behavior. The following figures show the types of nodes in the graphical calculation views and their use cases.



Icon	Node Type	Use Case
	Projection	To filter data or obtain a subset of required columns from a data source
	Aggregation	To summarize measures by grouping them together by attribute columns values
	Join	To query data from two or more data sources
	Non-Equi Join	To perform joins based on conditions >, <, and so on
	Union	To combine the data from two data sources
	Minus, Intersect	To analyze difference or intersection of two data sets
	Star Join	To join attributes to the very last node of a CUBE With Star Join Calculation view
	Rank	To order the data for a set of partition columns and select only the top 3/4/.../n elements

Figure 22: Types of Nodes in Graphical Calculation Views (1/2)



Icon	Node Type	Use Case
	Table Function	To use a table function and map parameters of <i>table</i> type
	Hierarchy Function	To include an SQL hierarchy function (generation or navigation)
	Anonymization	To conceal sensitive data with security algorithms
	Graph	To query a graph workspace with dedicated algorithms

Figure 23: Types of Nodes in Graphical Calculation Views (2/2)

The following unit is focusing on the main node types used in Calculation Views. Some of the nodes will be discussed later on, for example, the *Anonymization* node has a dedicated lesson in the unit, Security in SAP HANA Modeling. As for the *Graph* and *Hierarchy Function* nodes, they are discussed in another course, HA301 — SAP HANA Advanced Modeling.



LESSON SUMMARY

You should now be able to:

- Use common features to design calculation views

Learning Assessment

1. Which of the following are types of calculation views?

Choose the correct answers.

- A Cube
- B Composite
- C Cube with star join
- D Dimension

2. A dimension can have attributes and measures.

Determine whether this statement is true or false.

- True
- False

3. Which are true statements relating to calculation views of the type SQL access only?

Choose the correct answers.

- A They do not expose their meta data to reporting tools
- B They are mainly used for reusing inside other calculation views
- C They must include at least one measure
- D They include a star join node

4. Which are supported data sources for calculation view consumption?

Choose the correct answers.

- A Calculation views
- B Flat files
- C Virtual tables
- D Column tables
- E Row tables

5. What does SAP recommend you use to check your calculation view produces correct result?

Choose the correct answer.

- A Excel pivot tables
- B Custom SQL queries using SQL Console
- C Data preview function in the SAP Web IDE for SAP HANA

6. In a calculation view, which are valid output column types?

Choose the correct answers.

- A Measure
- B Attribute
- C Dimension
- D Cube

7. Why do you hide columns in a dimension calculation view?

Choose the correct answer.

- A When you want to hide a column that is not required or allowed for client consumption.
- B When you do not want to expose a sensitive column to a consuming calculation view.
- C To ensure they can only be displayed along with other attributes and not used for drill-down navigation or filtering.

Learning Assessment - Answers

1. Which of the following are types of calculation views?

Choose the correct answers.

- A Cube
- B Composite
- C Cube with star join
- D Dimension

Correct! — Cube, cube with star join and dimension are all types of calculation views. Composite is not a type of calculation view.

2. A dimension can have attributes and measures.

Determine whether this statement is true or false.

- True
- False

Correct — dimensions can have attributes but not measures. Measures are defined only in cube, cube with star join and SQL access only, types of calculation views.

3. Which are true statements relating to calculation views of the type SQL access only?

Choose the correct answers.

- A They do not expose their meta data to reporting tools
- B They are mainly used for reusing inside other calculation views
- C They must include at least one measure
- D They include a star join node

Correct! — SQL access only calculation views do not expose their meta-data to reporting tools. They are mainly used for reusing inside other calculation views. They can contain measures but measures are not mandatory and they cannot include star join nodes.

4. Which are supported data sources for calculation view consumption?

Choose the correct answers.

- A Calculation views
- B Flat files
- C Virtual tables
- D Column tables
- E Row tables

Correct — Calculation views, virtual tables, row tables, and column tables are valid data sources but flat files are not. You can load a row or column table with flat file data, or create a virtual table over a flat file but you cannot consume a flat file directly in a calculation view.

5. What does SAP recommend you use to check your calculation view produces correct result?

Choose the correct answer.

- A Excel pivot tables
- B Custom SQL queries using SQL Console
- C Data preview function in the SAP Web IDE for SAP HANA

Correct — Calculation views produce different results depending on the query that is calling the view so you need to create multiple custom queries, using SQL Console, to check the behavior of the calculation view under different query conditions. Data preview in the SAP Web IDE for SAP HANA and also XL pivot tables do not let you check all possible behaviors.

6. In a calculation view, which are valid output column types?

Choose the correct answers.

- A Measure
- B Attribute
- C Dimension
- D Cube

Correct — Measure and attribute are output column types. Dimension and cube are not output column types.

7. Why do you hide columns in a dimension calculation view?

Choose the correct answer.

- A** When you want to hide a column that is not required or allowed for client consumption.
- B** When you do not want to expose a sensitive column to a consuming calculation view.
- C** To ensure they can only be displayed along with other attributes and not used for drill-down navigation or filtering.

Correct — The reason you hide attributes in a dimension calculation view is to avoid exposing them to client applications. A hidden column is still exposed to consuming calculation views. Hiding a column means it is not exposed at all and does not simply switch off drill down and filtering capabilities.

UNIT 2

Using Nodes in Calculation Views

Lesson 1

Using Projection Nodes	45
------------------------	----

Lesson 2

Using Join Nodes	47
------------------	----

Lesson 3

Working with Data Sets	67
------------------------	----

Lesson 4

Aggregating Data	77
------------------	----

Lesson 5

Creating CUBE with Star Join Calculation Views	81
--	----

Lesson 6

Extracting Top Values with Rank Nodes	83
---------------------------------------	----

UNIT OBJECTIVES

- Use a projection node
- Use joins to combine data sources
- Join more than two tables in a single join node
- Work with non-equi joins
- Use a Dynamic Join
- Define Join Columns Optimization
- Use Union Nodes to combine data sets
- Use Set Operations: Minus and Intersect
- Use Aggregation Nodes

- Control the behavior of the Aggregation Node
- Use a Star Join in a CUBE calculation view
- Use a rank node to extract the top values of a data set

Using Projection Nodes



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Use a projection node

Projection Node

Using a Projection Node

A *Projection* node is typically used in the following scenarios:



- To filter measures based on attributes values.
- To extract only some columns from a data source.
- To define calculated columns, in particular when the calculation must occur BEFORE an aggregation.

Indeed, calculated columns in the *Aggregation* nodes of calculation views are always executed AFTER the aggregation.



Note:

With the most recent versions of SAP HANA, some capabilities such as filtering have been added to other types of nodes. So using projection nodes is probably a bit less frequent now than it was before. However, the use case about defining a calculation (for example on a table data source) that must be executed BEFORE an aggregation is still absolutely valid, and requires a projection node.



LESSON SUMMARY

You should now be able to:

- Use a projection node

Using Join Nodes



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Use joins to combine data sources
- Join more than two tables in a single join node
- Work with non-equi joins
- Use a Dynamic Join
- Define Join Columns Optimization

Connecting Tables with Join Nodes

One important activity when you create calculation views, is to express the relationships between the different data sources used by a model. Most often, this is done by using joins, a classical artifact in any relational database management system (RDBMS).

A specific node type, the *Join* node, is used in modeling to materialize joins between one or more data sources (tables, calculation views, and so on).

In addition, the *Star Join* node, which is used to model a star schema, also defines one or several joins between a main data sources (fact “table”) and DIMENSION calculation views.

For each *Join* node, you must define which columns of the two joined sources must participate in the join, as well as the join type, as discussed in the lesson Connecting Tables. You can also specify the cardinality, with the help of the *Propose Cardinality* feature, but only if you are sure that the cardinality you provide corresponds to how the data is actually organized.

Sample Business Case and Data

To illustrate the behavior of the different types of joins in SAP HANA, consider the following tables:

- Sales Order
- Customer
- State

The objective is to join these tables to retrieve the sales order amounts (facts) with the customer information, including the states in which the customers reside.



Sales Order

	ORDER_ID	C_ID	AMOUNT
1	1	1	100
2	2	1	100
3	3	2	100
4	4	4	100
5	8	77	100

Customer

	C_ID	CNAME	STATE	AGE
1	1	WERNER	MI	10
2	2	MARK	MI	11
3	3	TOM	TX	12
4	4	BOB	TX	13

We want to connect the **Sales Order table** to the **Customer table** linked to the **State table**.

State

	STATE	SNAME
1	MI	MICHIGAN
2	AL	ALABAMA

Figure 24: Sample Data for Business Example

To begin with, you can make the following observations:

- Sales Order 8 does not have a customer master record.
- Customer TOM does not have any orders.
- State TX does not have a description.
- No customer resides in Alabama.

Join Type Summary



Join type

Join type	Use when you want to report on	Be aware that
INNER	Facts with matching dimensions only	<ul style="list-style-type: none"> * Facts without a dimension will be excluded * Dimensions without a fact will be excluded * Join is always executed
LEFT OUTER	All posted facts whether there is a matching dimension or not	<ul style="list-style-type: none"> * Dimensions without a fact will be excluded * Best performance since join is ommissible
RIGHT OUTER	All dimension whether there are matching facts or not	<ul style="list-style-type: none"> * Facts without a dimension will be excluded * Right outer join is rarely used
FULL OUTER	All posted facts and all dimensions	* Combines the effects of LEFT and RIGHT OUTER join
REFERENTIAL	Facts with matching dimensions only where referential integrity is ensured	The join is only executed if columns from the right table are queried
TEXT	A multi-language table	<ul style="list-style-type: none"> * Requires a language column (SPRAS or equivalent) * Acts as a LEFT OUTER JOIN
TEMPORAL	A key date within a validity period	* Needs a temporal condition, from and to date.
SPATIAL	Geospatial data	* Executes specific computations on spatial data

Figure 25: Join Type Summary

There are different types of joins. The inner and outer joins are similar to other database management systems. Others are specific to SAP HANA and address specific requirements such as optimization (for example, the Referential Join), ease of use with typical SAP

Business Suite data model (for example, the Text join), and others. As an aside, some of the join types listed in the table consist of a specific option or configuration. For example, the Temporal Join is an Inner Join to which you add time-related conditions.

You can review the typical use cases for the different join types.

Note that the Spatial Join is discussed in more detailed in another course: HA301, SAP HANA 2.0 SPS03 Advanced Modeling.



Note:

This table builds on a scenario similar to the one just described (*Sample Data for Business Example*), assuming that a fact table (considered as the **left** table) is joined to a dimension table (considered as the **right** table).

Each join type will now be presented in detail.

Inner Join



Select * FROM "Customer" AS c -- table alias with explicit AS Inner Join "State" s -- table alias (simple form) On c.STATE = s.STATE
C_ID CNAME STATE AGE STATE SNAME 1 1 WERNER MI 10 MI MICHIGAN 2 2 MARK MI 11 MI MICHIGAN

STATE	SNAME
AL	ALABAMA
MI	MICHIGAN
TX	

✳ Customer (3 & 4) is not returned due to no corresponding entry (TX) in the state table.

Figure 26: Inner Join in a DIMENSION Calculation View

The Inner Join is the most basic of the join types. It returns rows when there is at least one match on both sides of the join.

Inner Join in a CUBE Calculation View



Select o.ORDER_ID, o.C_ID, sub1.CUSTOMER, sub1.AGE, sub1.STATE, sub1.STATETXT, o.AMOUNT FROM "SalesOrder" o Inner Join (Select c.C_ID, c.CNAME AS CUSTOMER, c.AGE, c.STATE, SNAME AS STATETXT FROM "Customer" c Inner Join "State" s On c.STATE = s.STATE) as sub1 On o.C_ID = sub1.C_ID ORDER BY o.ORDER_ID
ORDER_ID C_ID CUSTOMER AGE STATE STATETXT AMOUNT 1 1 1 WERNER 10 MI MICHIGAN 100 2 2 1 WERNER 10 MI MICHIGAN 100 3 3 2 MARK 11 MI MICHIGAN 100

✳ Inner Joins lose facts with fragmented dimensions. Order (4 & 8) lost due to no corresponding customer or state record.

AMOUNT	ORDER_ID	C_ID
100	4	4
100	8	77
100	1	1
100	2	1
100	3	2

STATE	SNAME
AL	ALABAMA
MI	MICHIGAN
TX	

Figure 27: Inner Join in a CUBE Calculation View

The figure, Inner Join in a CUBE Calculation View, shows the behavior of Inner Joins in a CUBE calculation view.

With the sample scenario data, some facts are not retrieved because customer information is missing.

Left Outer Join



<pre> Select c.C_ID, c.CNAME As CUSTOMER, c.AGE, c.STATE, SNAME as STATETXT FROM "Customer" c Left Outer Join "State" s On c.STATE = s.STATE </pre>				
C_ID	CUSTOMER	AGE	STATE	STATETXT
1	WERNER	10	MI	MICHIGAN
2	MARK	11	MI	MICHIGAN
3	TOM	12	TX	?
4	BOB	13	TX	?

STATE	SNAME
AL	ALABAMA
MI	MICHIGAN

STATE	SNAME
AL	ALABAMA
MI	MICHIGAN
TX	*
TX	*

* No matches for TX in the right table.

Figure 28: Left Outer Join in a DIMENSION Calculation View

A Left Outer Join returns all rows from the left table, even if there are no matches in the right table.

Left Outer Joins and Design Time Filters



CUSTOMER			
C_ID	CNAME	STATE	AGE
1	WERNER	MI	10
2	MARK	MI	11
3	TOM	TX	12
4	BOB	TX	13

Design time filter applied (AGE < 13) of left/central table

STATE	
STATE	SNAME
MI	MICHIGAN
AL	ALABAMA

Design time filter applied to (STATE = MI) on right table

RESULT		
C_ID	CNAME	STATE
1	WERNER	MICHIGAN
2	MARK	MICHIGAN
3	TOM	NULL

Filters are applied to both tables and then afterwards the join is executed.
Due to the left outer join TOM will be included in the result set even though he resides in TX.

Figure 29: Left Outer Joins and Design Time Filters

The figure, Left Outer Joins and Design Time Filters, shows the behavior of left outer joins with design time filters.

Left Outer Join in a CUBE Calculation View



```

Select
    o.ORDER_ID, o.C_ID, sub1.CUSTOMER, sub1.AGE,
    sub1.STATE, sub1.STATETXT, o.AMOUNT
    FROM "SalesOrder" o
    Left Outer Join
        (Select
            c.C_ID, c.CNAME As CUSTOMER,
            c.AGE, c.STATE, sname as STATETXT
            FROM "Customer" c
            Left Outer Join "State" s
            On c.STATE = s.STATE) as sub1
        On o.C_ID = sub1.C_ID
    ORDER BY o.ORDER_ID

```

	ORDER_ID	C_ID	CUSTOMER	AGE	STATE	STATETXT	AMOUNT
1	1	1	WERNER	10	MI	MICHIGAN	100
2	2	1	WERNER	10	MI	MICHIGAN	100
3	3	2	MARK	11	MI	MICHIGAN	100
4	4	4	BOB	13	TX	?	100
5	8	77	?	?	?	?	100

 Customer TOM is not returned due to no corresponding sale item record in sales order table.

AMOUNT	ORDER_ID	C_ID	STATE	SNAME
100	1	1	AL	ALABAMA
100	2	1	MI	MICHIGAN
100	3	2	MI	MICHIGAN
		3	TX	*
100	4	4	TX	
100	8	77		

 Figure 30: Left Outer Joins in a CUBE Calculation View

The figure, Left Outer Joins in a CUBE calculation view, shows the use of left outer joins in a CUBE calculation view.

Compared with the Inner join, all the sales order data (including those with no corresponding customer information) is retrieved, but still an analysis of sales by customer or state will return irrelevant data.

Right Outer Join



- Right Outer Join returns all the rows from the right table, even if there are no matches in the left table.

```

Select
    c.C_ID, c.CNAME As CUSTOMER,
    c.AGE, c.STATE, sname as STATETXT
    FROM "Customer" c
    Right Outer Join "State" s
    On c.STATE = s.STATE

```

	C_ID	CUSTOMER	AGE	STATE	STATETXT
1	1	WERNER	10	MI	MICHIGAN
2	2	MARK	11	MI	MICHIGAN
3	?	?	?	?	ALABAMA *

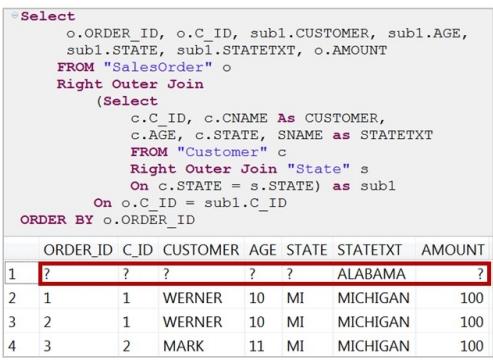
STATE	SNAME
AL	ALABAMA
MI	MICHIGAN

 Alabama is included in the result set, though there is no match in the left table.

 Figure 31: Right Outer Join in a DIMENSION Calculation View

A Right Outer Join returns all the rows from the right table, even if there are no matches in the left table.

Right Outer Join in a CUBE Calculation View

Right Outer Join results in NULL measure.

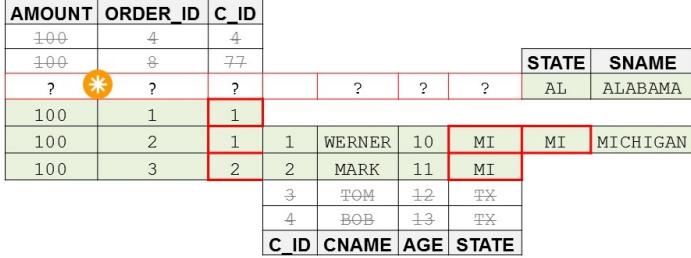


 Figure 32: Right Outer Join in a CUBE Calculation View

Full Outer Join



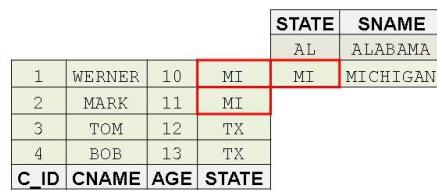


 Figure 33: Full Outer Join

A Full Outer Join combines the behaviors of the Left and Right Outer Joins.

The result set is composed of the following rows:

- Rows from both tables that match on joined columns
- Rows from the left table with no match in the right table
- Rows from the right table with no match in the left table

 Caution:
A Full Outer Join is supported by calculation views only, in the standard *Join* and *Star Join* nodes.
However, in a *Star Join* node, a full outer join can be defined only on one DIMENSION calculation view, and this view must appear last in the *Star Join* node.

Referential Join

SAP HANA offers a type of join that is optimized for performance: the Referential Join.

The key principle of a Referential Join is that, if referential integrity between two tables is ensured, then under some circumstances, the join between these tables will not be executed, which will save execution time.

The concept of **referential integrity** between two tables (A and B) means that, in the joined columns (1 or more columns from each table), there is always a match in table B for a row of column A, or the other way round, or both.

Let's take the example of a CUBE with Star Join calculation view that is defined with many DIMENSION calculation views, and the joins all have cardinality $n..1$ (meaning, a fact is connected to at most one member or each dimension). Let's assume that, by design, the source system ensures that all records in the fact table always have a match in all the DIMENSION calculation views (this is often the case).

So, when you execute a query on your CUBE calculation view, the calculation engine can prune any DIMENSION calculation view from which your query does not request any column. So the corresponding join will not be executed. This is where optimization occurs.



Table 7: Referential Join

Relies on Referential Integrity	Referential integrity is the fact that matches exist between the joined tables (in one direction, for example left to right, or right to left, or both).
Optimized for performance	Referential Join is not executed in circumstances where the result (without the join) will be the same as if the join was executed.
Like an Inner Join when join is executed	When a Referential Join is not pruned, it is executed as an Inner Join.

Defining a Referential Join

To define a Referential Join, you first add a *Join* node to the calculation view scenario, and assign two or more data sources. In the case of a CUBE with Star Join calculation view, you assign a lower node to the *Star Join* node and add one or more DIMENSION calculation view.

Then, you have to define the following settings:

- **Join Type**

This setting must be set to *Referential*.

- **Cardinality**

Cardinality must be specified. If it is not, the Referential Join that cannot be optimized.

- **Integrity Constraint**

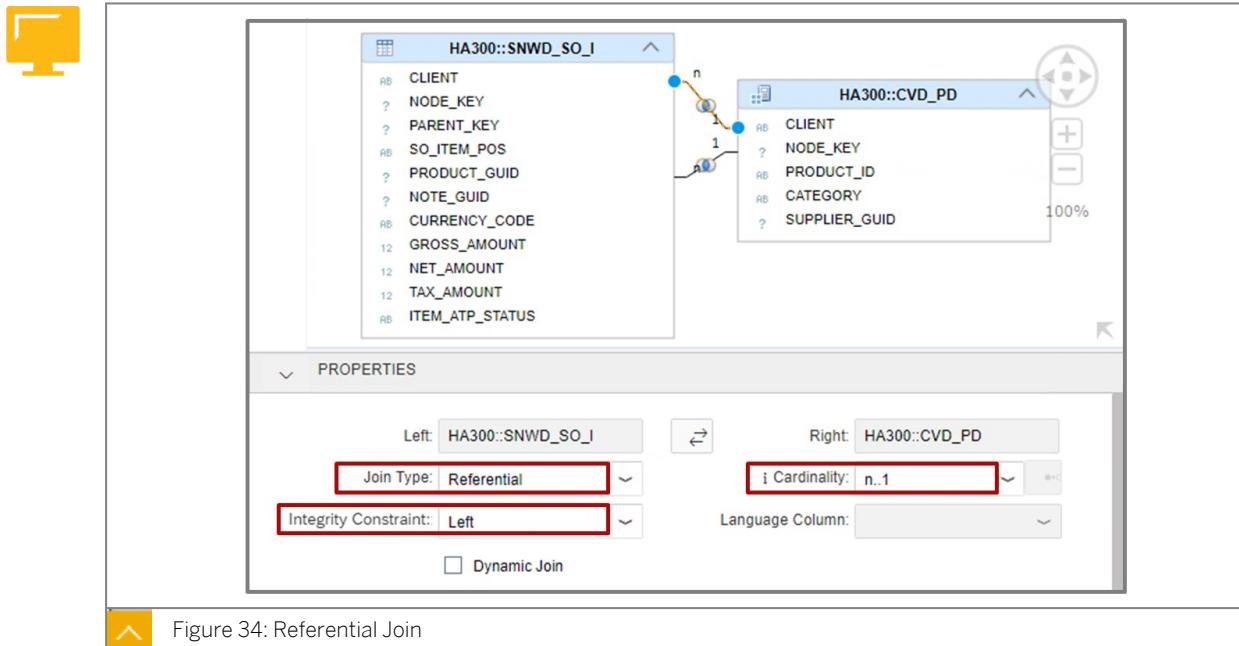
This setting defines in which direction the referential integrity is guaranteed.

- **Left:** Every entry in left table has at least one match in right table.

- **Right:** Every entry in right table has at least one match in left table.

- **Both:** Every entry in both tables has at least one match in the other table.

With the *Integrity Constraint* setting, you ensure that the optimization of the join occurs only when it is actually possible, because the cardinality alone is not enough. If *Integrity Constraint* is set to left, no join optimization is triggered if a query requests columns only from the right table.



Conditions for Referential Join Optimization

A join defined as a Referential Join between two tables or sources, A and B, is pruned (not executed) when all three following conditions are met:

- No field is requested from B.
- Integrity is placed on A.
- The cardinality on B side is :1.



Note:

When the cardinality on B side is not :1, the join will always be executed even if no column from B is requested. This is a requirement to get the correct number of rows in the output, which depends on the number of matching rows in B for each row in A.

Let's take an example, based on the figure, Referential Join.

The Join is defined as *Referential*, the *Integrity Constraint* (*Integrity*) is placed on the *Left "table"*. This means that any record from the SNWD_SO_I table has **at least** one match in the CVD_PD calculation view. However, the cardinality is *n..1*, which also tells us that this is **at most** one match.

So all in all, for each record of SNWD_SO_I, there is **exactly** one match in CVD_PD.

With this join definition, if a query selects NO column from the right "table" CVD_PD, then the join will not be executed.

**Note:**

The *Integrity Constraint* setting is a new feature of SAP HANA 2.0 SPS03. Up to version 2.0 SPS02, the referential optimization occurred only when columns from the right table were not selected and the cardinality was 1..1 or n..1. In other words, its behavior was the same as a Referential Join with Integrity Constraint = Left.

Referential Joins must be used with caution because they assume that referential integrity is ensured **at any time**. Using Referential Joins in a context where referential integrity is not ensured might lead to different results depending on whether or not you select columns from one of the two data sources.

**Note:**

If you consider a join between a fact table and its related attributes, keep in mind that facts without corresponding attributes violate referential integrity, while attributes without facts (for example, a customer without any order) do not.

Text Join

A Text Join enables SAP HANA to handle the translation of attribute labels in a way that corresponds to how translation texts are stored in the master data. In particular, this way to store texts in different languages is heavily used in SAP systems, such as the SAP Business Suite.




Figure 35: Text Joins

Technically, a Text Join behaves like a Left Outer Join, with cardinality 1:1, but in addition you specify a language column, typically called SPRAS in SAP systems tables.

During join execution, the language of the end user querying the calculation view is used to retrieve descriptions from the text table (here, MAKT) in the corresponding language, based on the language column.

**Note:**

In the back-end system, this design for master data tables is what allows to store a description for a single item (here, a given Material Number) in different languages in a dedicated master data tables (here, *MAKT*) without using a different column for each language, which would be more complicated to handle.

Text Join Example

- Text join is used when a translation for a dimension is available
- Designed for ERP tables (and typically SPRAS column)
- User language is used as a filter at runtime to find the right translation for that attribute

PRODUCT table

SPRAS	ID	DESC
E	1	Car
D	1	Auto
E	2	Motorbike
D	2	Motorrad

Text join applied on **ORDER.PR_ID = PRODUCT.ID**
using **SPRAS** as the language column

ORDER table

ORDER_ID	PR_ID	VALUE
00215753	1	1000
00237469	2	2000



Figure 36: Text Join Example

The figure, Text Join Example, is a simplified example of a Text Join. Depending on the session language of the end user, the *DESC* column displays the product description in English or in German.

Temporal Join

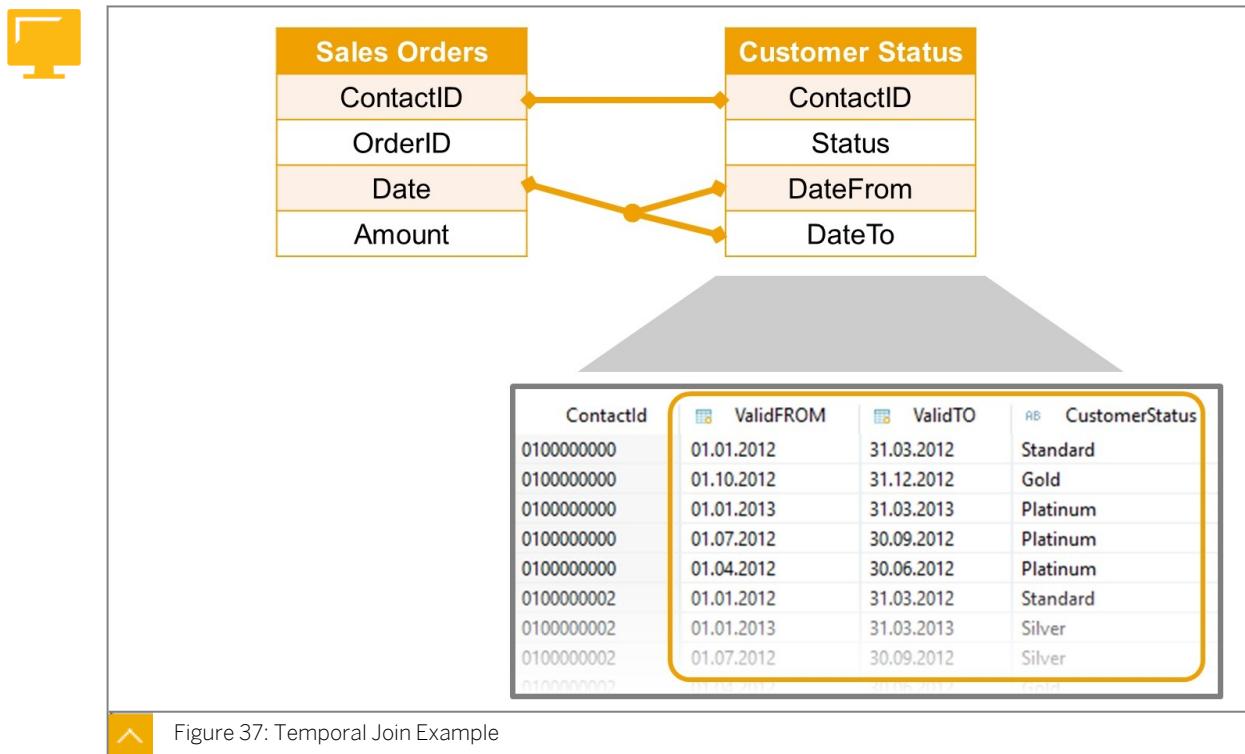
It is possible to add a temporal condition to a join in order to find matching records from two tables based on a date. The records are matched only if a date column of one table is within a time interval defined by two columns of the other table.

This is useful to manage time-dependent attributes.

**Caution:**

Temporal Joins are only supported in the *Star Join* of calculation views of the type CUBE with Star Join. The join type must be defined as *Inner*.

Temporal Join Example



In this example, the status of the customers can change over time, and this information is captured in a dedicated table (Customer Status). If you need to analyze the sales orders and include the status of each customer when they issued the order, you create an Inner Join on the *ContactID* column and add a temporal condition as follows:

- Temporal column: Date (Sales Orders)
- From Column: DateFrom (Customer Status)
- To Column: DateTo (Customer Status)
- Temporal Condition: Include Both

Note:

- Temporal conditions can be defined on columns of the following data types:
 - *timestamp*
 - *date*
 - *integer*
- Only columns already mapped to the output of the *Star Join* node can be defined as *Temporal Columns* in the temporal properties of the join.

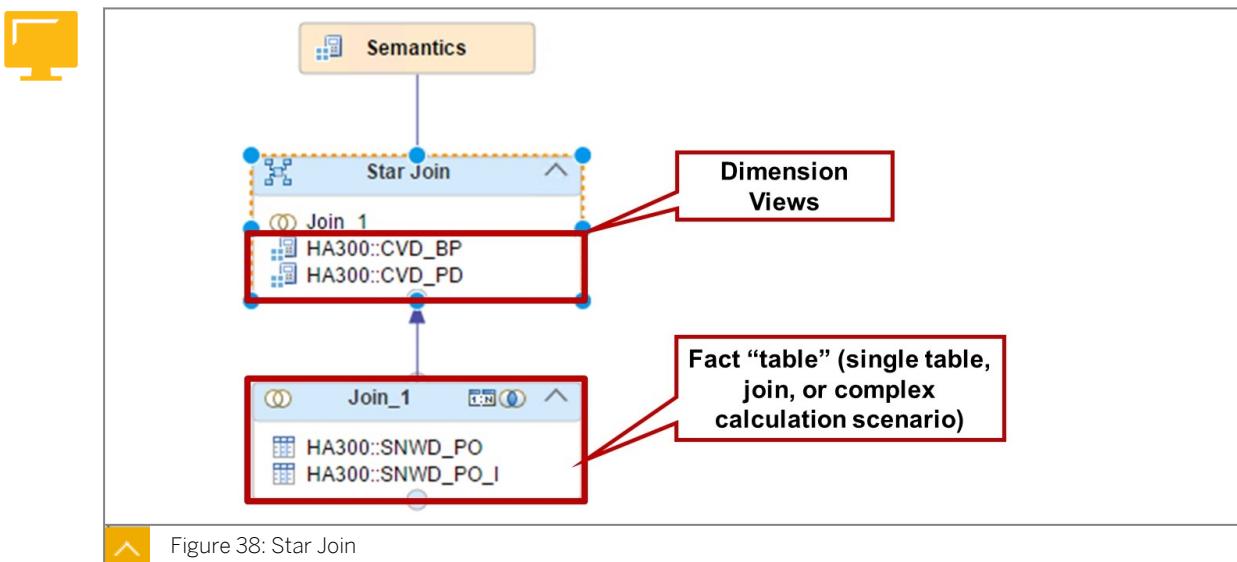
Star Join

The *Star Join* in calculation views of the type CUBE with Star Join is a **node type**, rather than a join type.

It is used to structure data in a star schema. The fact table (data source) of a *Star Join* can be any type of input node. However, only calculation views of the data category **DIMENSION** are allowed as input nodes for dimensions.

The type of joins between the fact and dimension tables within the star schema can be defined in the *Star Join* node. The available joins are as follows:

- Referential Join
- Inner Join
- Left Outer Join
- Right Outer Join
- Full Outer Join, with some specific restrictions (see above)
- Text Join



Shared Columns from DIMENSION Calculation Views

In a CUBE with Star Join Calculation View, the *Columns* tab of the Semantics separates columns into two categories:

- Private

Private columns are columns that are defined inside the calculation view itself. These can be measures or attributes. You have full control over these columns.

- Shared

Shared columns are columns that are defined "externally", in one or more **DIMENSION** Calculation Views that are referenced by your CUBE with Star Join Calculation View. On these columns, you have logically less control, because they are potentially "shared" with other CUBE with Star Join Calculation Views. Still, you can hide some of these columns to keep only the ones that you need.

Regarding the shared columns, their *Name* and *Label* properties cannot be changed, compared with a private column, but you can define an Alias Name and an Alias Label. Moreover, providing Alias Names is mandatory if column names from the underlying **DIMENSION** calculation views conflict with each other or conflict with the private column names.

Join Cardinality

The cardinality of a join defines how the data from two tables joined together are related, in terms of matching rows.

For example, if you join the *Sales Order* table (left table) with the *Customer* table (right table), you can define an *n..1* cardinality. This cardinality means that several sales orders can be related to the same customer, but the opposite is not possible (you cannot have a sales orders that relates to several customers).



Caution:

We recommend that you specify the cardinality only when you are sure of the content of the tables. If not, just leave the cardinality blank.

Validating a Join

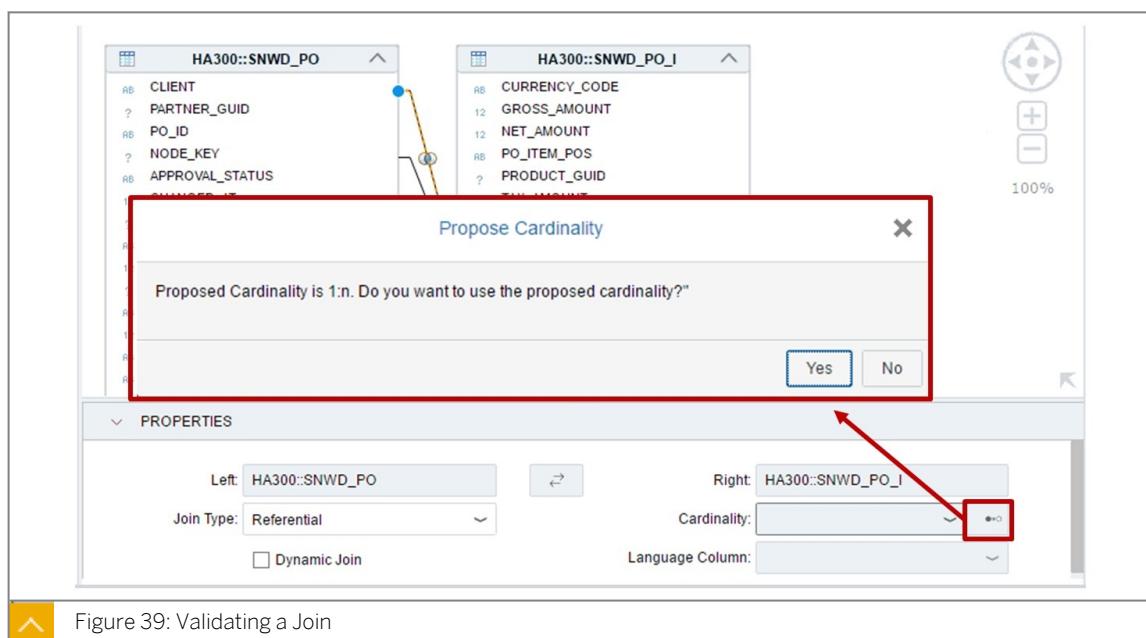


Figure 39: Validating a Join

A feature in the SAP Web IDE for SAP HANA suggests the recommended cardinality, based on an analysis of the tables that are joined together.



Caution:

This analysis of joined tables is performed at the moment you define the join. If the content of the table evolves after that, the cardinality you have defined might become incorrect.

For example, you are validating the join between the *Sales Order* and *Customer* tables, but your data contains only **one** sales order per customer. In this case, the join validation might suggest a *1..1* cardinality, which does not correspond to the expected scenario in real life.

Multi-Join

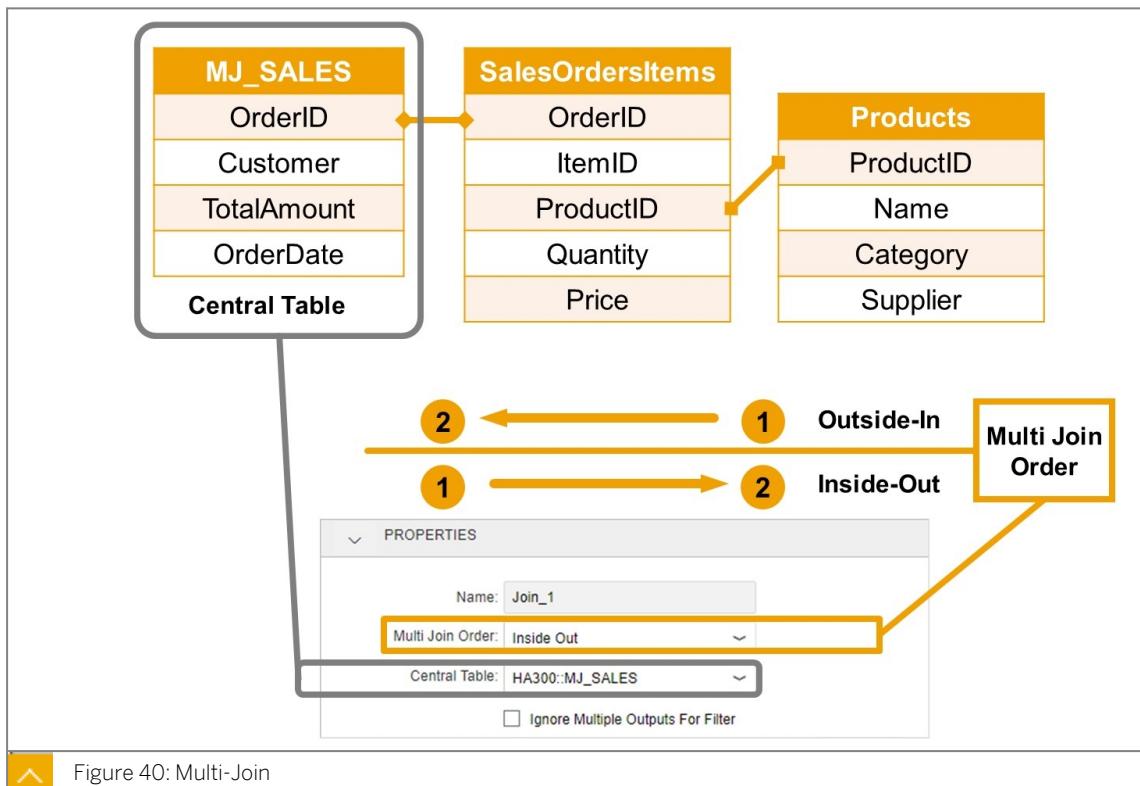
From version 2.0 SPS03 onwards, SAP HANA provides a new capability in join nodes, which is to allow more than two data sources in the same node.



Note:

Up to version 2.0 SPS02, a classical join node (not a *Star Join* node, which has a slightly different purpose) allowed only two data sources. Joining more than two tables always required cascading join nodes (successive join nodes where each intermediate node joins the output of the previous one with an additional data source).

In that case, joins are executed in the order that corresponds to the sequence of nodes, that is, bottom-up according to the default layout (starting with the nodes that are furthest from the top or default calculation view node).



When more than two data sources, for example, tables, other calculation views, or other nodes, are assigned to a join node, you must define a priority (precedence) for join execution. This is done in the *Properties* pane of the *Join Definition* tab.

You need to specify the following:

- The Multi-Join Order
- A Central Table

Based on this specification, the order of join execution is as follows:

- Outside-In:

The join that is furthest from the central table is executed first.

- Inside-out:

The join that is closest to the central table is executed first.

Note that the concept of central table does not relate to the fact that the table is joined to ("surrounded by") two or more tables. It is just the identification of a kind of pivot table for determining the join order.

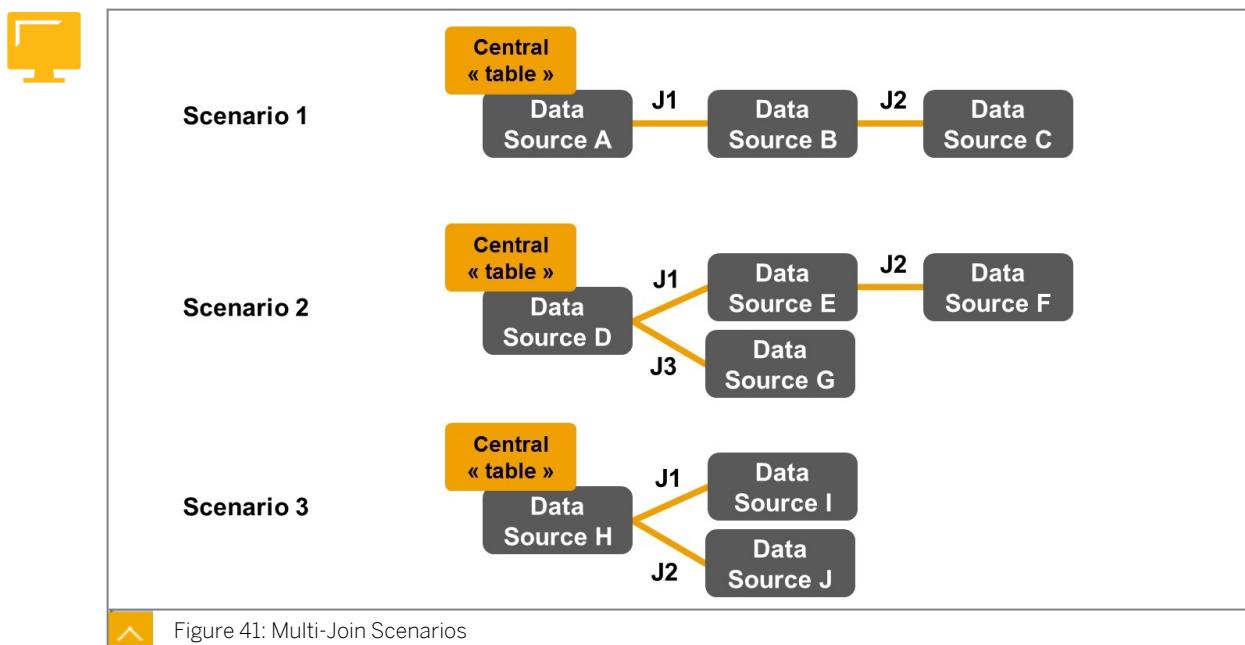


Figure 41: Multi-Join Scenarios

In the figure, Multi-Join Scenarios, the multi-join order property only applies in scenarios 1 and 2, and affects joins J1 and J2. The precedence between joins J1 and J3 (in scenario 2) or J1 and J2 (in scenario 3) is not controlled by the multi-join order setting.

When does Multi-Join Priority Affect the Join Node Results?

With more than two data sources feeding a *Join* node, the result sometimes depend on the join execution order, but this is not always the case. For example:

- When all joins are Inner Joins, the result set is generally the same regardless of the join execution order.
- With a mix of Inner and Left Outer Joins, the result set can vary based on the join execution order.

So, it is up to the Modeler to decide between joining more than two tables in a single node, or sticking to joining "only" two tables in a given join node, based on the potential differences in behavior, and which approach provides a better legibility of the Calculation View design in the calculation scenario and/or mapping tabs.

Non-Equi Join

A non-equi join is where the join condition is not represented by an = (**equal**) operator. For example, the value of column *CUSTOMER_ID* in the table *ORDERS* **equals** the value of column *ID* in table *CUSTOMERS*. Instead, the join condition is based on other comparison operators such as **Greater than** or **Non equal to**. This supports scenarios where there is not an exact match between column values but instead the values are higher or lower or in between.

You define a non-equi join using a dedicated node in the calculation view graphical editor.

Defining a non-equi Join condition is possible for the following types of joins:

- Inner
- Left Outer
- Right Outer
- Full Outer



Hint:

In a non-equi Join, the operator can be specified differently for each pair of joined columns, as illustrated in Example 1 below.



Join Definition

Join Definition Mapping Parameters (0) Columns (3)

ProductsToBeDelivered

SUBTASKS

Left: ProductsToBeDelivered Right: SUBTASKS

Left Column: dueOn Right Column: plannedDate

Join Type: Inner Cardinality: n..m

Operator: Less Than

ProductsToBeDelivered			
RB	id	RB task	◇ dueOn
1	car		2019-01-01
2	bicycle		2019-01-04
3	house		2019-05-05

SUBTASKS			
RB	id	RB subtask	◇ plannedDate
1	tires		2019-02-01
1	brakes		2019-12-31
1	windows		9999-12-12
2	tires		2019-04-01
2	brakes		2019-01-01
3	roof		9999-12-12
3	walls		2019-02-02
3	floor		2019-02-02

subtasks that will not finish in time

RB task	◇ dueOn	RB subtask	◇ plannedDate
bicycle	2019-01-04	tires	2019-04-01
car	2019-01-01	brakes	2019-12-31
car	2019-01-01	windows	9999-12-12
car	2019-01-01	tires	2019-02-01
house	2019-05-05	roof	9999-12-12

Figure 42: Non-Equi Join — Example 1

In Example 1, two tables contain a list of products to be delivered and a list of sub-tasks which provides the availability date of components.

The objective is to display, in a calculation view, which sub-tasks will not finish early enough to allow the company to meet the planned completion date for some products.

This requirement can be achieved with a Non-Equi Join with the following join conditions:

- ProductsToBeDelivered.id **Equal** SUBTASKS.id
- ProductsToBeDelivered.dueOn **Less Than** SUBTASKS.plannedDate

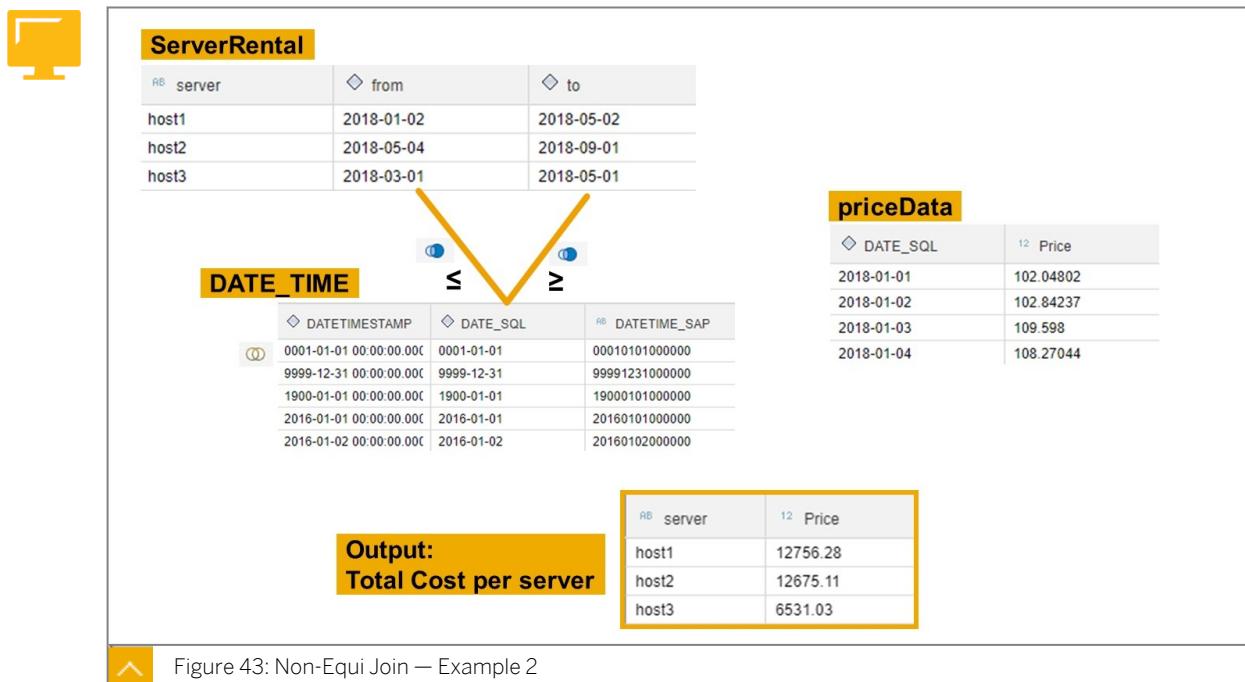


Figure 43: Non-Equi Join — Example 2

In Example 2, we assume that server rental prices vary from day to day.

To calculate the total cost for a rental period of several days, rental periods are "translated" into days by a non-equi Join involving the system date table *M_TIME_DIMENSION*. The result of this non-equi Join is then joined (classical join) with the table *priceData* that contains the rental price for each day.

As well as using the standard operators: **Less Than**, **Greater Than**, **Less Than Equal To**, **Greater Than Equal To**, **No Equal** it is also possible to define the non-equi join condition using an expression.

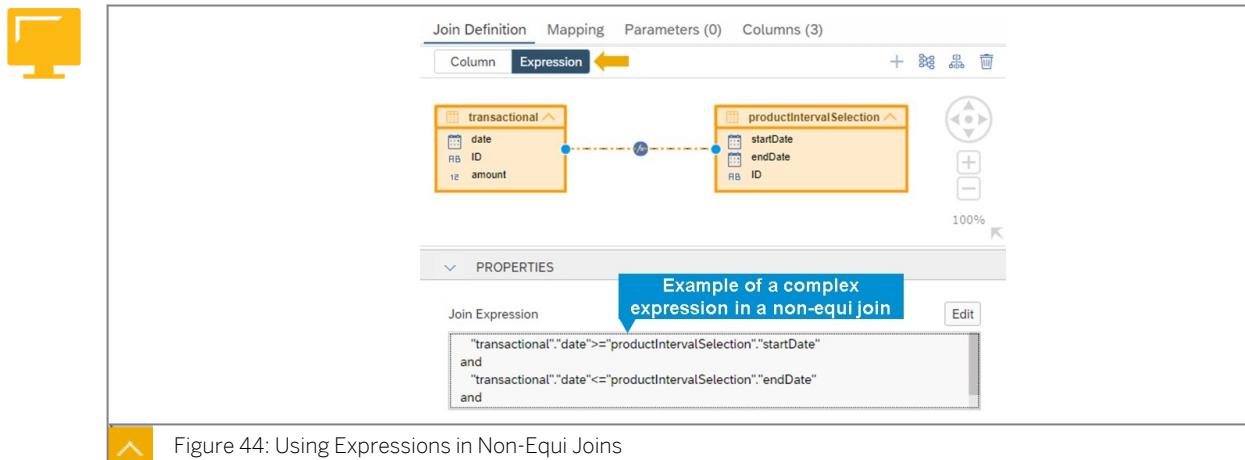


Figure 44: Using Expressions in Non-Equi Joins

There are some restrictions when using non-equi joins compared to equi joins. These are as follows:

- They cannot be used in text joins, referential joins, or dynamic joins.
- You cannot create calculated columns or any filter expressions in non-equi join nodes.

**Note:**

You cannot define a temporal join if you use *columns* to define the condition. However, you can define a temporal join using an expression.

Dynamic Join

Enhancing Model Flexibility with Dynamic Join

In some scenarios, you want to allow data analysis at different levels of granularity with the same calculation view.

This is generally possible in an *Aggregation* node when measures support the aggregation at different levels, that is, when it is possible to report measures grouped by one set of columns or another. For example, calculating the total sales by country and product in one case, and by region and product in another case.



Objective:
Show the Sales & Total Sales by Region & Product

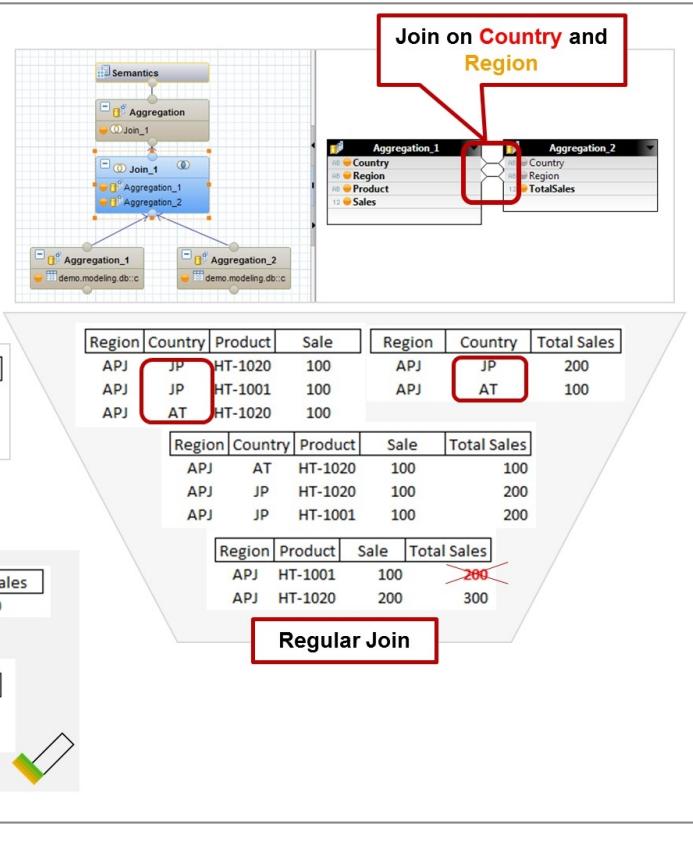


Figure 45: Dynamic Join

The figure, Dynamic Join, shows an example of a more complex scenario, where you want to present two different measures side by side, either by Country or Region:

- The sales by product
- The total sales (all products)

In this case, assuming that you model your calculation view with a Regular Join on *Country* and *Region*, you will get correct results if you analyze the data by country, but the results will be inconsistent if you analyze the data by region.

**Note:**

On further analysis of the example, you can see that the details of total sales by region and product are inconsistent for products that are not sold in all the countries of the region (HT-1001)... or (to be specific) in all the countries of the region that report sales.

Benefits of a Dynamic Join

With a Dynamic Join, only the join columns requested in the query are brought into context and play a part in the join execution. As a consequence, the same calculation view can be used for both purposes, that is, to analyze data by country or by region.

**Note:**

A Dynamic Join can be defined only with multi-column joins.

If we consider the behavior of the join from an aggregation perspective:

- In a Regular (static) Join, the aggregation is executed after the join.
- In a Dynamic Join, when a joined column is not requested by the client query, an aggregation is triggered to remove this column, and then the join is executed based only on the requested columns.

**Caution:**

With a Dynamic Join, if none of the joined columns are requested by the client query, you get a query runtime error.

Join Columns Optimization

In this lesson, you have already learned about a powerful optimization approach for joins in SAP HANA, which is to use Referential Joins. Compared with an Inner Join, a Referential Join can be omitted, in some circumstances, without affecting the calculation view output. These circumstances mainly relate to referential integrity, cardinalities, and which of the joined data sources the requested columns come from.

Still, in scenarios where one of the join partners can be pruned, another optimization question is whether the joined column(s) in the queried data source, the one that is not pruned, should be kept.

Background - The Default Behavior

Per default, the field on which a join is defined will be requested during query processing, regardless of whether the field is necessary from a query point of view.

The key objective is to guarantee a consistent aggregation behavior concerning the join field, even when the field is not requested by the query. So, the purpose of the default behavior of query execution is to get a stable aggregation behavior. If this aggregation behavior changes depending on which columns are requested, this could lead to a change of the resulting values of measures.

Optimize Join Columns Option

When you select the *Optimize Join Columns* checkbox, you tell the calculation engine that – in case a join partner is pruned – you do not expect changes to the result set depending on whether the joined column from the queried data source is used for aggregation or not. This depends heavily on the type of aggregations that are performed by the calculation view. In particular, the SUM aggregate function is not sensitive to the grouping level, whereas the MAX or MIN are.

When Does the Optimization (Join Pruning) Occur?

When the *Optimize Join Columns* option is active, pruning of join columns between two data sources, A and B, occurs when all four following conditions are met:

- The join type is Referential, Outer or Text (in particular, the calculation view cannot be built if join type is *Inner*).
- Only columns from one join partner, A, are requested.
- The join column from A is NOT requested by the query.
- The cardinality on B side (the side of the join partner from which no column is requested) is :1.



Caution:

As you see, the optimization heavily relies on cardinality. So, you must ensure that the cardinality is set according to the actual data model. If it is not, the *Optimize Join Columns* option will produce unstable (though sometimes faster) results.



LESSON SUMMARY

You should now be able to:

- Use joins to combine data sources
- Join more than two tables in a single join node
- Work with non-equi joins
- Use a Dynamic Join
- Define Join Columns Optimization

Unit 2

Lesson 3

Working with Data Sets



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Use Union Nodes to combine data sets
- Use Set Operations: Minus and Intersect

Union Node



- If you want to combine multiple result sets with identical structures into one result set, you can use a union node
- A mapping of the sources to the target is required and will allow you to adapt structural differences
- This can be done via a drag and drop interface.

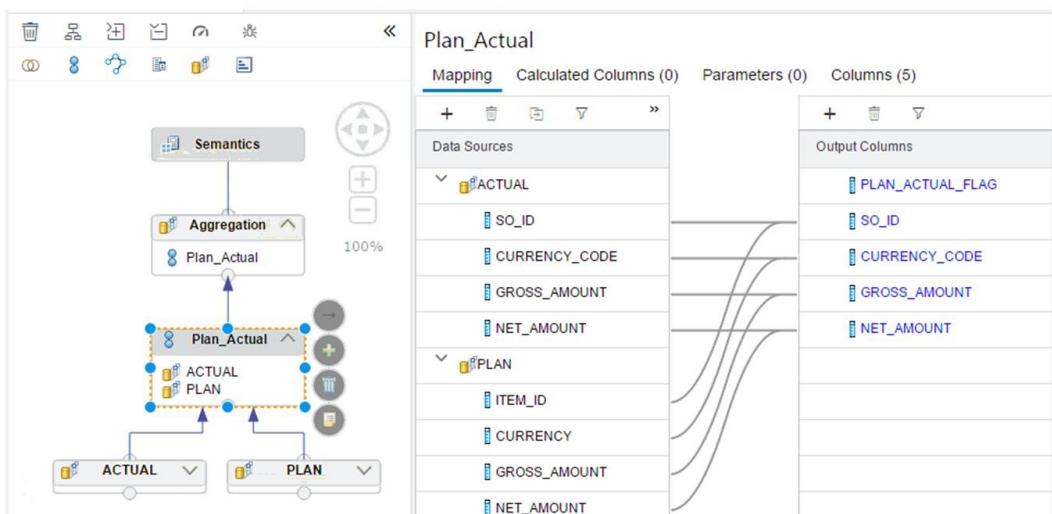


Figure 46: Union Node

A *Union* node is used to combine two or more data sets to a common set of columns.

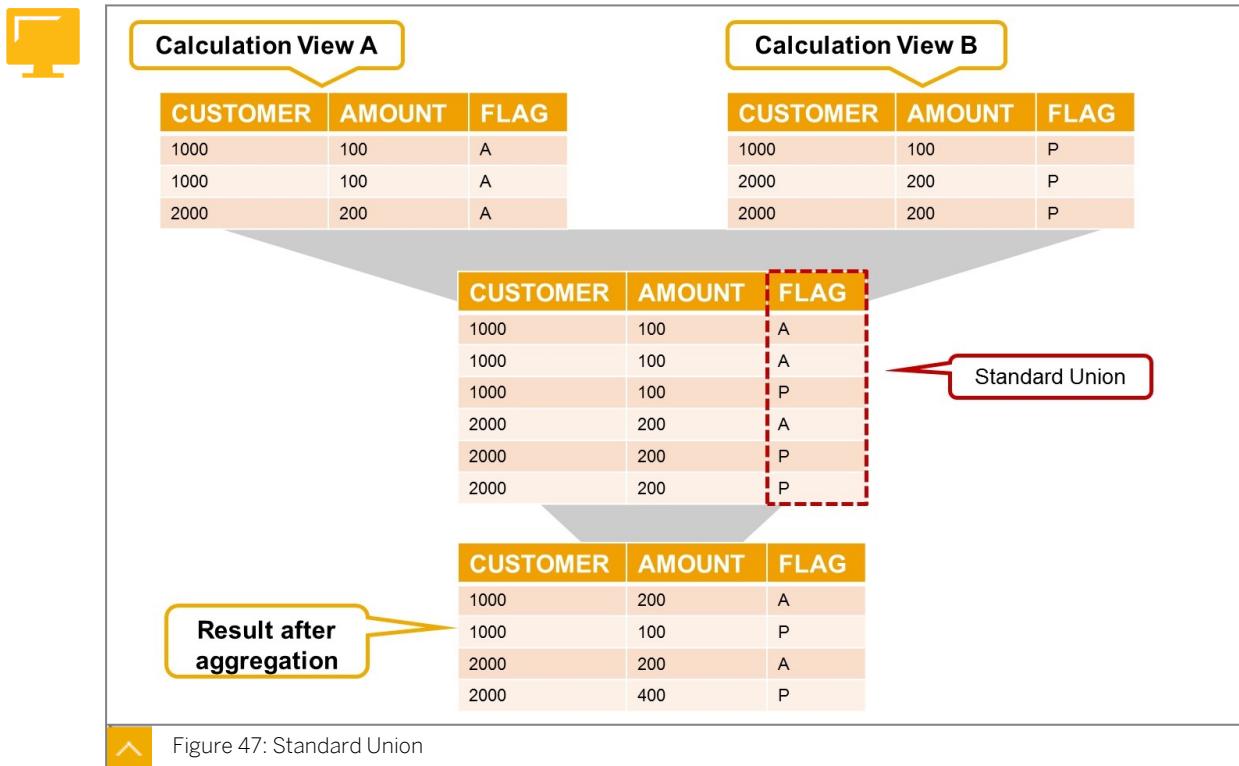
Depending on how different the column names are from the data source, you can map the columns automatically by name (columns with identical names will be mapped automatically) or define the mapping manually.

Standard Union

Depending on the requirement, you can use one of the following approaches:

- A standard union

- A union with constant values



A standard union is where, for each target column in the union, there is always one source column that is mapped. In the example, you see how both sources provide a mapped column to the union. The source column does not have to have the same name as the target column. For example, a source column *month* could be mapped to a target column *period*. Also, a standard union does not have to provide a mapping for each source column. In other words, there could be source columns that are left behind and play no part in the union. This scenario is useful when you want to combine measures from multiple sources into one column. Because the data sources can provide an attribute that describes the type of measure (for example, Plan or Actual) it means that we do not lose the meaning of each row.

Union with Constant Values

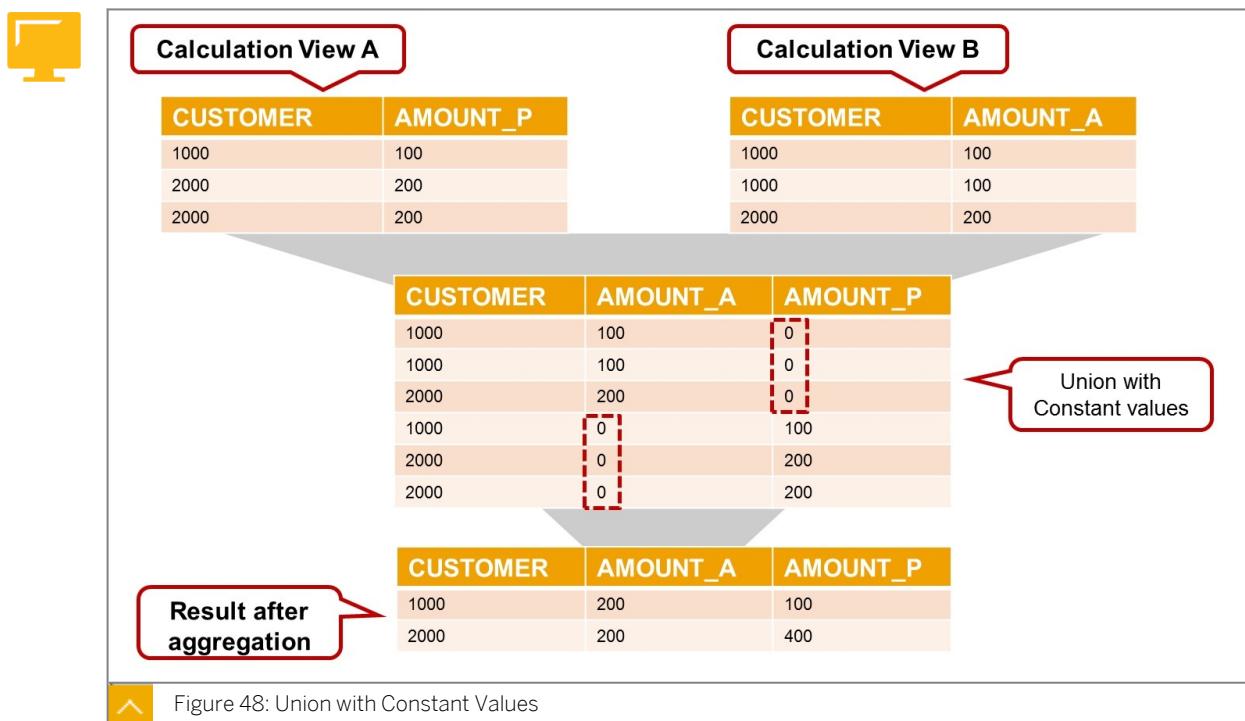


Figure 48: Union with Constant Values

A union with constant values is where you provide a fixed value that is used to fill a target column where the data source cannot provide a mapped column. For example, you have a data source that provides the actual amount and you also have a second data source that provides the planned amount. You decide to avoid combining these measures into one column as they would lose meaning, as there is no attribute to describe the meaning. In this case you would map the measure to a separate target column and provide a constant value '0' to the target column for the missing measure on each side.

The choice between a standard union and a union with constant values depends on the data you are using and the way you want the end users to report on data.

If it is more beneficial to present different measures in different columns, you can use a union with constant values so that you have a way to provide a value (probably zero) to a column where there is no source mapping. On the contrary, if it is easier to present measures in a single column and differentiate them with an attribute, such as an 'amount type', use the standard union.

Mapping Columns Based on their Names

It is possible to map columns in a *Union* node based on their names. This helps to save time when the data structures have some similarities in terms of column names.

The *Auto Map by Name* feature can be used in two different ways, depending on whether you make a selection in the *Data Source* area before clicking the icon.

Table 8: Two Options to Map Columns by Name Automatically

Scenario	Result
You have not selected any data source (default behavior)	All the columns of all data sources are added to the output. Columns with matching names are mapped together.
You have selected one or several data source(s)	Only the columns of the selected data source(s) are added to the output. The columns from the other data source(s) are added to the output only if they have a matching column in the selected data source(s). They are mapped to their matching columns.

**Note:**

Selecting a data source before triggering *Auto Map by Name* is useful when one or several other data sources have a lot of columns that you do not want to include in the output.

Unmapped Columns in a Union Node

There could be instances when a union needs to be performed between data sources that do not provide the same data structure.

- If column names are different but contain the same type of information (for example, *CURRENCY* versus *CURRENCY_CODE*, you must define the mapping manually. This can be done with drag and drop.
- If a column exists in only one data source but you need it in the target, you map it to the target. Then, for the other source, you must decide whether you allow the data in this column to be null or you can define a **constant** value that will be assigned to all the rows from this data source.

As an example, you have a data source that provides the customer status attribute and you have a second data source where the customer status attribute cannot be provided; there is no column for this. You know this second data source contains only customers with the status *Active*. So you simply fill the target column with a constant value for this data source to *Active* for every row.

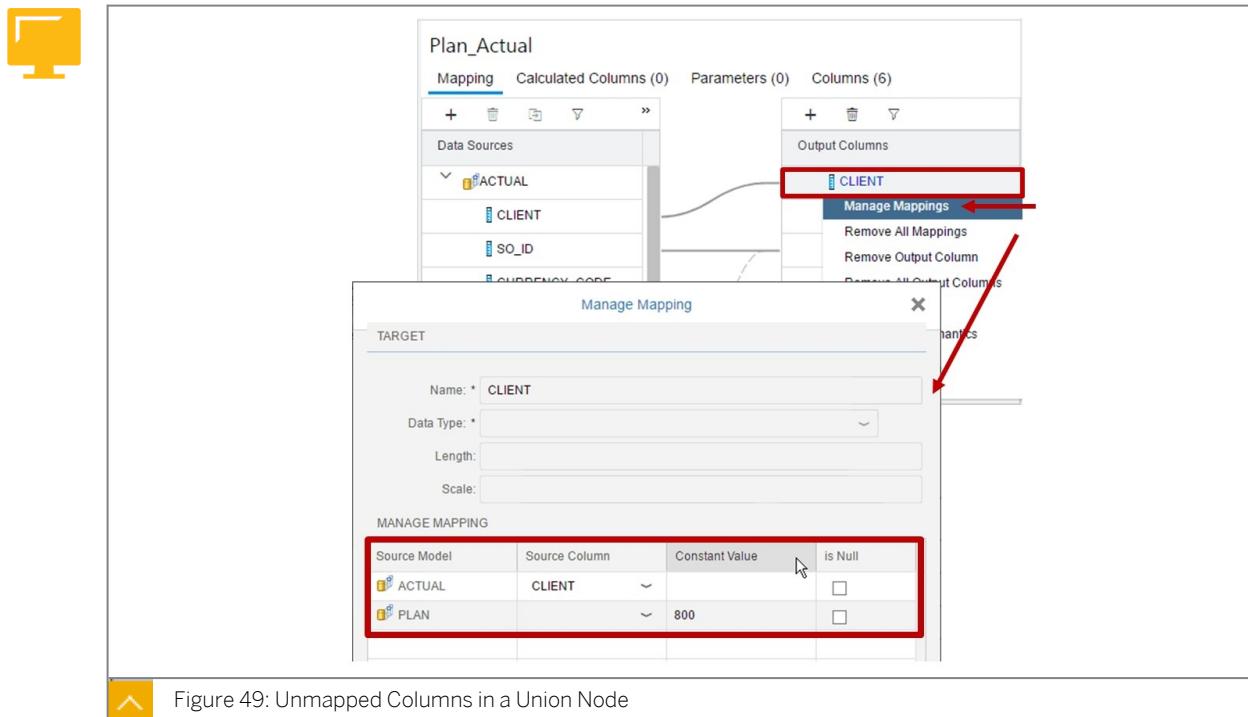
Even if you have a source column available for mapping, you can choose to use a constant value instead. This creates the effect of overriding the source value.

Manage Mapping of Unmapped Columns in a Union Node

Although we described how constants are often used when one of the data sources cannot, or should not, provide a value to the union target column, we can also use constants when **none** of the data sources can provide a value. To do this, we first create an empty column in the union target. We then define a constant value for each data source. For example, I would like to create a union between table A, that contains part-time employees' data, and another table B, that contains full-time employees' data. The employment status is very important for my analysis but neither table contains a column that indicates employment status. So I simply add a new column to the union target called 'Employment Status' and define a constant value

for one data source as *Full Time* and the other data source as *Part Time*. Basically, I have manually tagged each source with a fixed label that now appears in each row and I can use this new column for filtering, aggregation, and so on.

To set the constant value, right-click the target column and choose *Manage Mappings*.



Union Node Pruning

In some circumstances, the query that is executed on top of a *Union* node can ignore completely one or several of the data sources.

Think about a *Union* node that combines two data sources: *Cold* (data up to 2016) and *Hot* (data for 2017 and later).

If a select statement queries only the data for 2017, the *Cold* data source can be ignored from the start (this is called pruning), which will provide a better performance than if the query execution scans the entire *Cold* data source looking for records for 2017, and eventually does not find a single one.

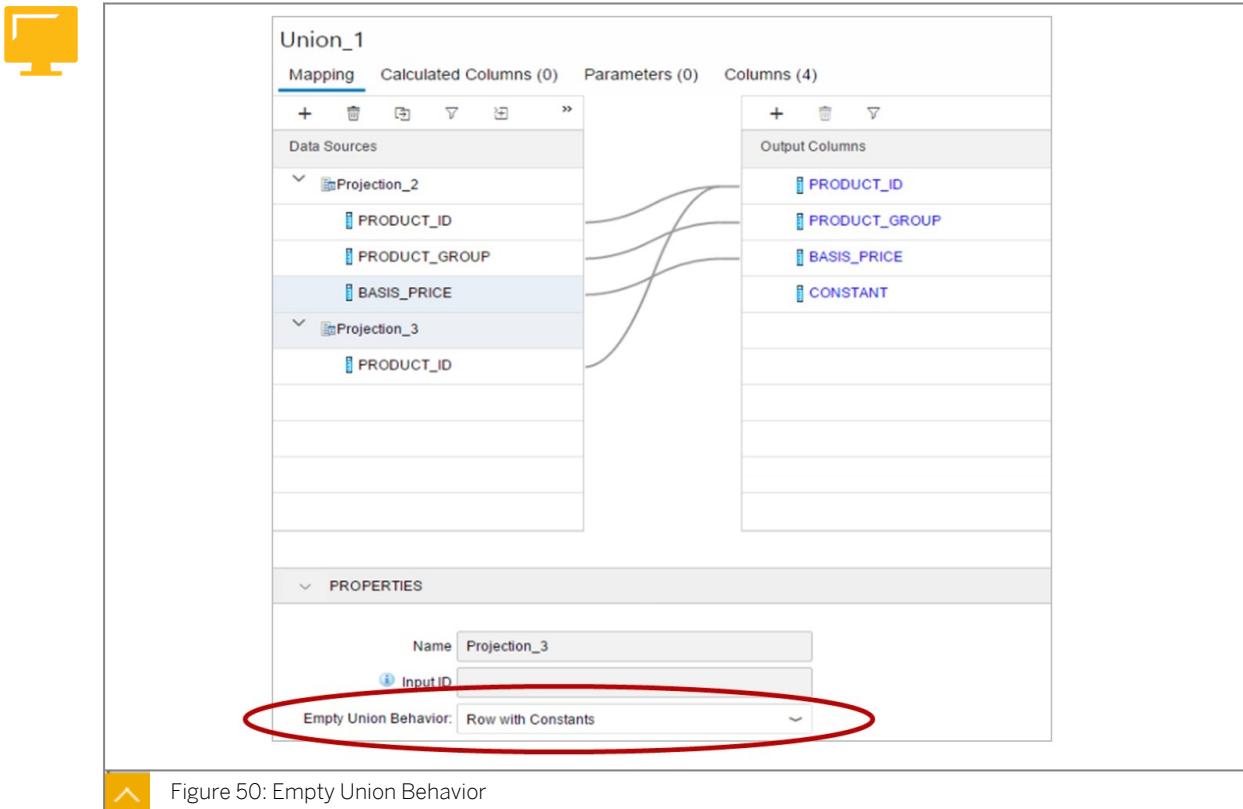


Note:

You will learn how to define *Union* node pruning in the unit *Optimization of Models*.

Empty Union Behavior

We know that the data sources to a union can provide different columns. So what happens when a query requests columns that are present in one source but not another source? Do you want to be made aware that one source was not able to provide a row? Or is this not important?



To illustrate this feature, imagine you have a data source, A, that contains the columns *Product* and *Product Group* and another data source, B, that contains only the column *Product*. If a query requests only the column *Product Group*, how does data source B respond when it doesn't have this column?

The answer depends on the setting in the property *Empty Union Behavior* which is set for each data source. The default behavior, as you might expect, is to provide *No Row* for data source B.

But there are times when you might prefer to know that no rows could be returned from data source B. In that case you should proceed as follows:

1. Add a constant column to the union output.
2. Provide a constant value for each data source with a suitable value that help you identify each source.
3. Change the property *Empty Union Behavior* to *Row with Constants*.

To test this, you simply consume the calculation view that contains the union, with a query that does not request any column from one of the data sources. In our case above, the query should request only the column *Product Group*. The results will then contain multiple rows from data source A, and also a single row with the constant value that you defined for the data source B and nulls will fill the empty columns that it could not provide.

Combining Several Data Sets: Union Versus Join

When you need to combine measures originating from more than one table or calculation view, you might want to create a join between these tables. However, under most circumstances, you must avoid using a join to address this requirement, because it is very costly from a computing standpoint.

It is more beneficial to use a *Union* node, which provides much better performance.



Note:
Technically, a union is not a join type.

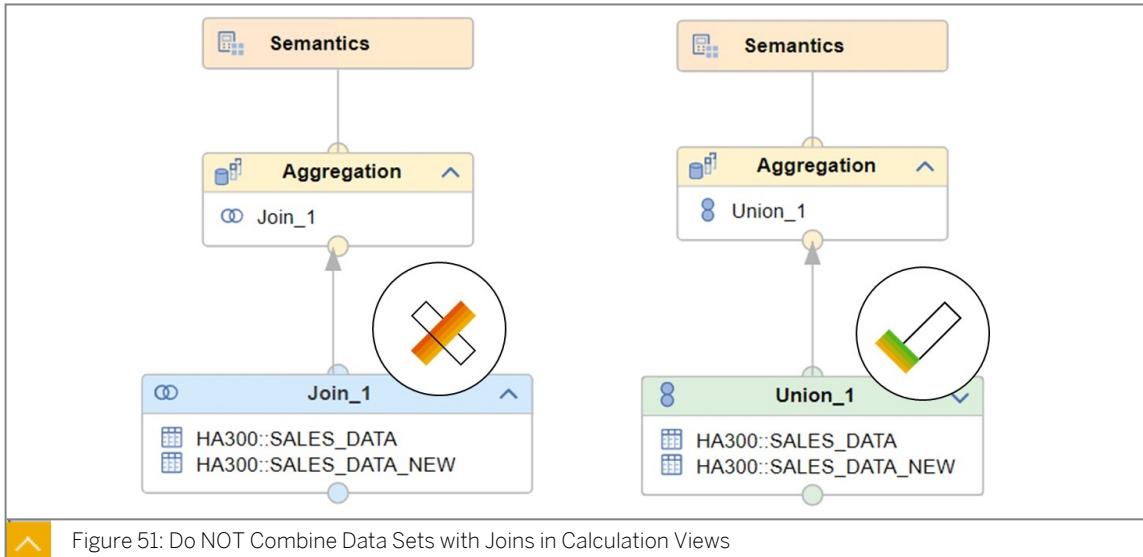


Figure 51: Do NOT Combine Data Sets with Joins in Calculation Views

Union All or Union Distinct

A *Union* node generates a list of all values from the input data sources, even if values are repeated. For example, if two data sources both include the same customer then the customer will appear twice in the output data set.

This is the equivalent of `UNION ALL` in SQL and is desirable in many cases.

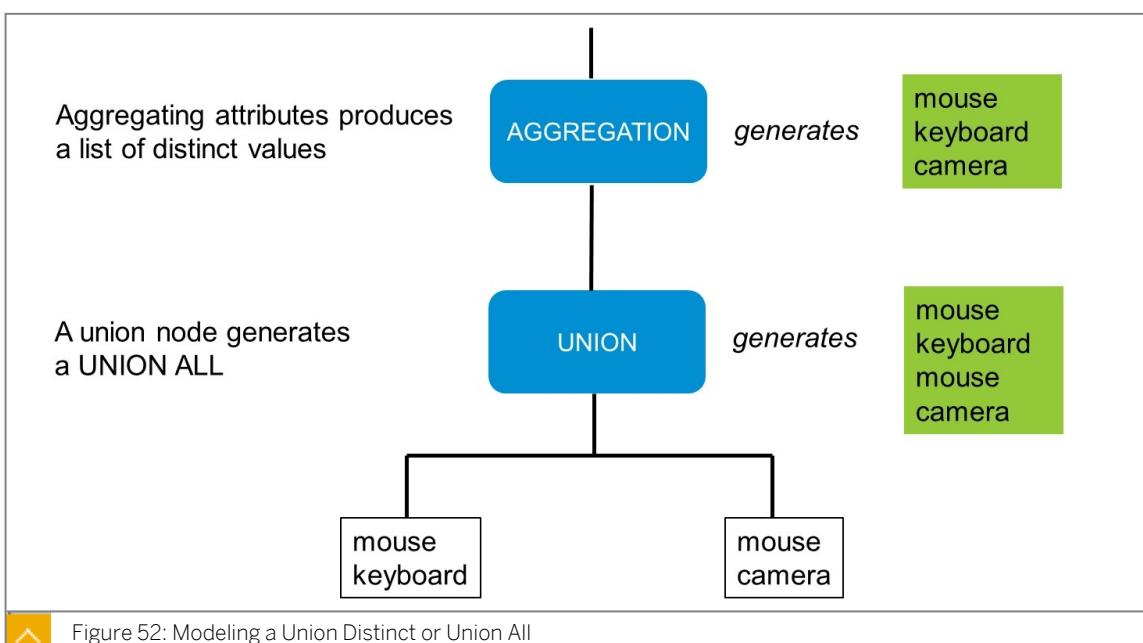
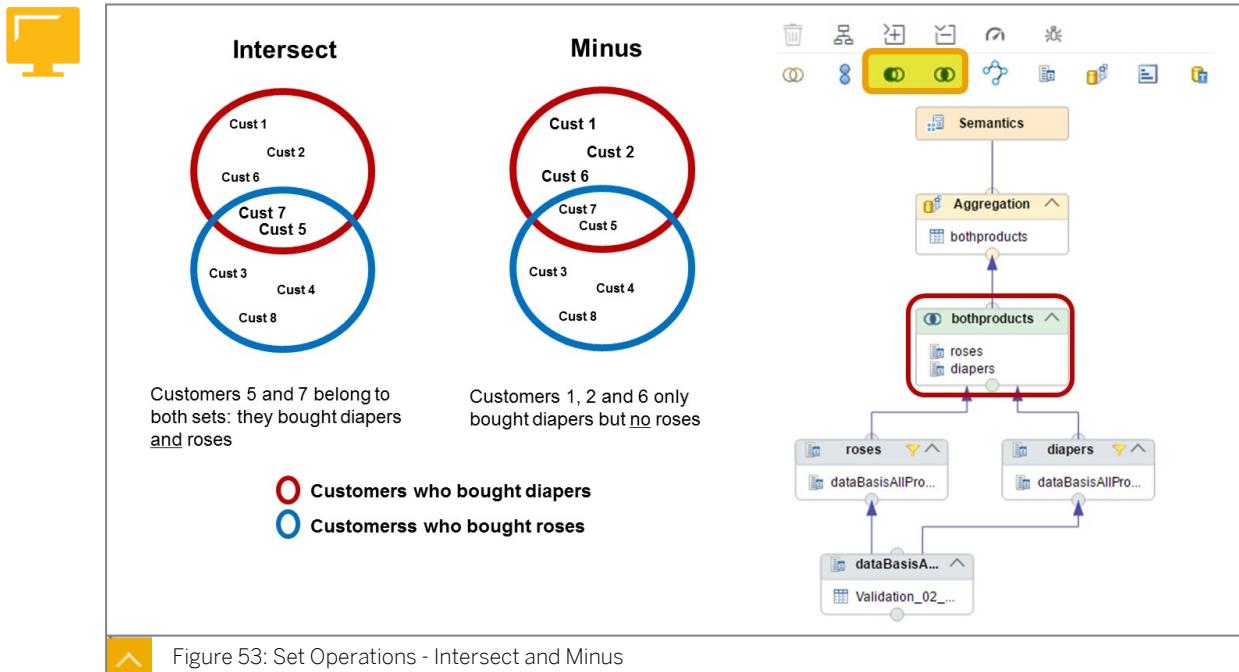


Figure 52: Modeling a Union Distinct or Union All

If repeating values are not required, then you should include an *Aggregation* node on top of the *Union* node to aggregate the attributes. We normally associate aggregation behavior with measures, such as SUM, AVE, and so on, but aggregation can also be performed on attributes. The effect of aggregating attributes is to simply produce a distinct list of values.

This is the equivalent of `UNION DISTINCT` (or just `UNION`) in SQL.

Set Operations Nodes – Intersect and Minus



SAP HANA 2.0 SPS01 introduced new types of node, dedicated to set operations: *Intersect* and *Minus*.

The purpose of these nodes is to easily filter members that belong to two data sets, *Intersect*, or members that belong to one data set and not another, *Minus*.



Note:

As per the figure, Set Operations - Intersect and Minus, you see that *Intersect* is symmetrical (it gives the same results regardless of the order of the two data sources) while *Minus* is not.

For the *Minus* node, the data sets are considered based on their order in the list of data sources for the node. So, the output contains items from the FIRST data source that are NOT in the SECOND data source. From SAP HANA 2.0 SPS04 onwards, it is possible to change the order of the data sources: just right click the *Minus* node and choose *Switch Order*.

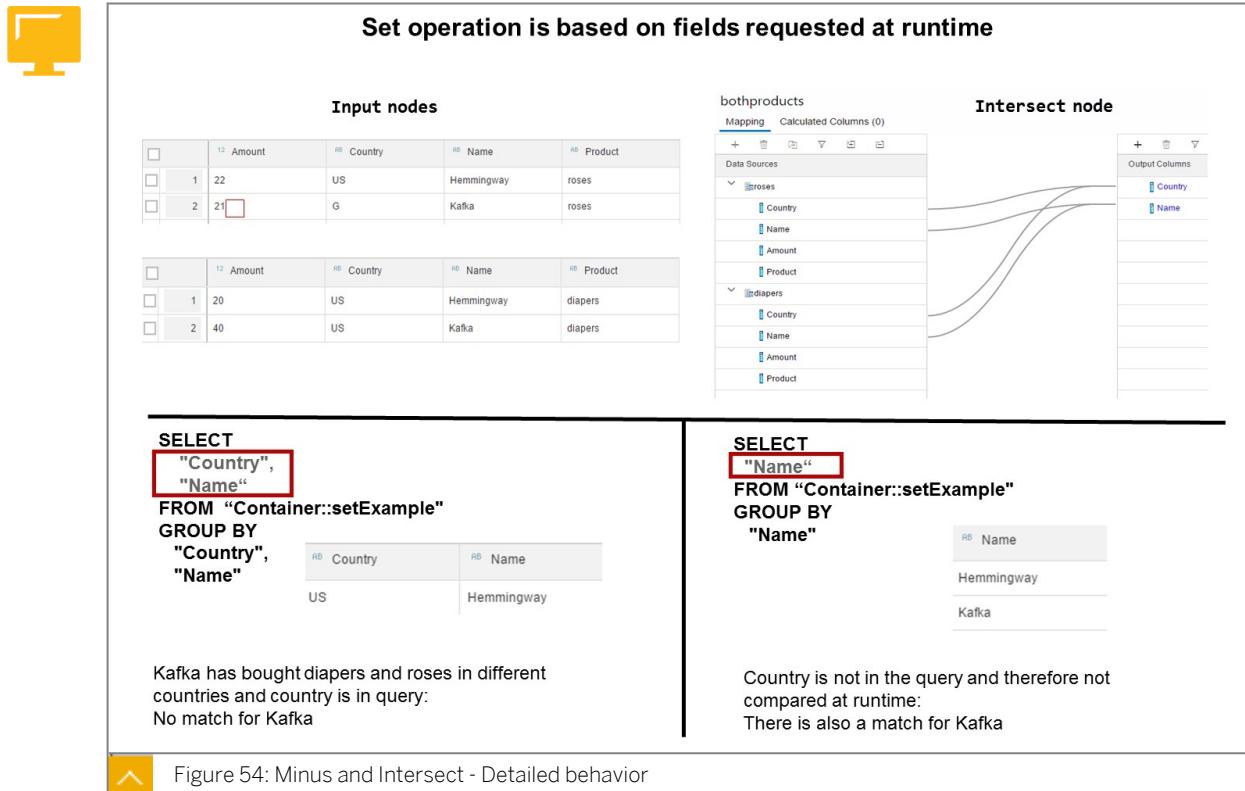


Figure 54: Minus and Intersect - Detailed behavior

Filtering relies on the list of attributes that are queried at runtime. In other words, a column that is provided by both source nodes, for example, *Country*, but is not queried at runtime, is ignored.



Note:

As of SAP HANA 2.0 SPS05, it is possible to swap the two data sources of a *Minus* node. Of course, this has an impact on the semantic.



LESSON SUMMARY

You should now be able to:

- Use Union Nodes to combine data sets
- Use Set Operations: Minus and Intersect

Aggregating Data



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Use Aggregation Nodes
- Control the behavior of the Aggregation Node

Aggregation Node

The purpose of an Aggregation node is to apply aggregate functions to measures based on one or several attributes.



Note:

This function acts similarly to a *GROUP BY* clause in an SQL query, where you must specify the aggregate function for each measure. Here, the *GROUP BY* clause is not materialized as such, but takes each and every attribute selected for the output.

A graphical calculation view can support *SUM*, *MIN*, *MAX*, and *COUNT*.

From version 1.0 SPS11, SAP HANA allows you to apply the following additional aggregate functions to the calculation views:

- Average
- Variance
- Standard deviation



Caution:

These aggregate functions should not be used in stacked scenarios.

In an Aggregation node, a calculated column is always computed AFTER the aggregate functions. If you need to calculate a column BEFORE aggregating the data from this column, you have to define the calculated column in another node, for example a *Projection* node, executed BEFORE the aggregation node in the calculation tree.

Attribute Versus Measure

When adding a column to the output of an Aggregation node, you define whether the column is used for grouping (*Add to Output*) or if it must be combined with an aggregate function (*Add as Aggregated Column*). In the second case, you must specify the aggregate function to apply (*Sum* is applied by default).

Defining an Aggregation in the Client Query

When you execute an SQL query on top of a CUBE Calculation View (with or without Star Join), the behavior depends on whether your SQL query includes a *GROUP BY* clause or not.

- If your SQL query does not include a *GROUP BY* clause, each requested measure is aggregated as specified in the Semantics of the Calculation View
- If your SQL query includes a *GROUP BY* clause, the aggregate function you specify for each requested measure overwrites the one defined in the Semantics of the Calculation View. This provides a lot of flexibility, but can be error-prone in some scenarios. It is recommended to use this with care.



Note:

The default data preview SQL statement for a CUBE calculation view (with or without Star Join) in the SAP Web IDE uses the second approach, thus showing you which aggregate function is applied to each measure.

Controlling the Behavior of the Aggregation Node

When you work with Aggregation nodes, the list of columns requested by the client query can influence the way the aggregation is executed, especially in complex scenarios.

The following features can help you control the aggregation of measures, in order to build more flexible models:

- *Keep Flag*
- *Transparent Filter*

Keep Flag

Let's consider a scenario where a data source contains the details of sales orders, by order ID. The only measures available are quantity and price.



Note:

Each order ID relates to one store, one customer, and only one product (in order to simplify the example).

You want to calculate the quantity and total sales for the product *Mouse* and the month of *February*.



Objective:
Show Mouse sales in February!

Order	Month	Product	Store	Customer	Quantity	Price	Results
1	2	Ipod	TigerDirect	John	20	20	
2	2	Mouse	TigerDirect	Susan	2	5	10
3	2	Mouse	TigerDirect	John	2	5	10
4	2	Mouse	Amazon	John	2	5	10
5	2	Headset	Amazon	Susan	3	5	
6	2	Headset	Ebay	John	2	5	
7	2	Ipad	Amazon	Susan	3	250	
8	2	Ipad	Ebay	Susan	3	250	
9	2	Mouse	Amazon	Peter	2	5	
10	3	Ipad	Amazon	John	3	250	10 40

Regular SQL

```
SQL Result
SELECT "Month", "Product", sum("Quantity"), sum("Price")
FROM "DEMO"."demo.modeling.db::cds.StoreOrders"
WHERE ("Month" = '2' AND "Product" = 'Mouse')
group by "Month", "Product"
```

Month	Product	SUM(Quantity)	SUM(Price)	Sales
1	2	Mouse	8	20

Output

Columns

- ORDER_ID: ORDERS.ORDER_ID
- PRODUCT: ORDERS.PRODUCT
- STORE: ORDERS.STORE
- CUSTOMER: ORDERS.CUSTOMER
- QUANTITY: ORDERS.QUANTITY
- UNIT_PRICE: ORDERS.UNIT_PRICE

Properties

General	Value
Name	ORDER_ID
Mapping	"TRAINING".ORD...
Length	
Scale	
Data Type	INTEGER
Filter	Keep Flag True
Transparent...	False

Solution:
Within the Calculation Model,
Set **Keep Flag = True** on **Order ID**

Figure 55: Keep Flag

Here, the issue is that the columns you request mandate a level of aggregation (*Month*, *Product*) that is generic. Hence, the total sales is calculated by multiplying the sum of quantities by the sum of unit prices.

Setting the *Keep Flag* property to true for the *Order ID* column forces the calculation to be triggered at the relevant level of granularity, even if this level is not the grouping level defined by the client query.

From SAP HANA 2.0 SPS02 onwards, the *Keep Flag* option can also be defined on the shared columns in the *Star Join* node of a CUBE with Star Join calculation view (shared columns are the ones defined in the underlying DIMENSION calculation views).

Transparent Filter

In some scenarios, using a filter (where clause) in a client query forces a column to be used in the *Group By* columns set.

In this scenario, for example, calculating the number of stores that sold mice to John or Susan triggers an intermediate calculation of the storecount sum by product and by customer, which makes the total by product irrelevant.

**Objective:**

Count the amount of stores that sold a Mouse to John or Susan

```
select product, sum(quantity), sum(storecount) from model
where Customer in (john, susan) and product = mouse group by product
```

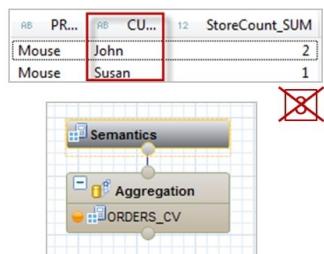
Order Month Product Store Customer Quantity Price

1	2	Ipod	TigerDirect	John	20	20
2	2	Mouse	TigerDirect	Susan	2	5
3	2	Mouse	TigerDirect	John	2	5
4	2	Mouse	Amazon	John	2	5
5	2	Headset	Amazon	Susan	3	5
6	2	Headset	Ebay	John	2	5
7	2	Ipad	Amazon	Susan	3	250
8	2	Ipad	Ebay	Susan	3	250
9	3	Ipad	Amazon	John	3	250
10	2	Mouse	Amazon	Peter	3	250

Results

Product	Quantity	StoreCount
Mouse	6	2

Stacked Calculation Model



Solution:
Set the **Transparent Filter** property for the column **Customer** to **True**, on all models and nodes that reference the Customer

Figure 56: Transparent Filter

Setting the *Transparent Filter* property to true for all models and nodes that reference the *Customer* column, will stop the *Customer* column from being unnecessarily used in the *Group By* clause.

This property is required in the following situations:

- When using stacked views where the lower views have distinct count measures.
- When queries executed on the upper calculation view contain filters on columns that are not projected.

**LESSON SUMMARY**

You should now be able to:

- Use Aggregation Nodes
- Control the behavior of the Aggregation Node

Creating CUBE with Star Join Calculation Views



LESSON OBJECTIVES

After completing this lesson, you will be able to:

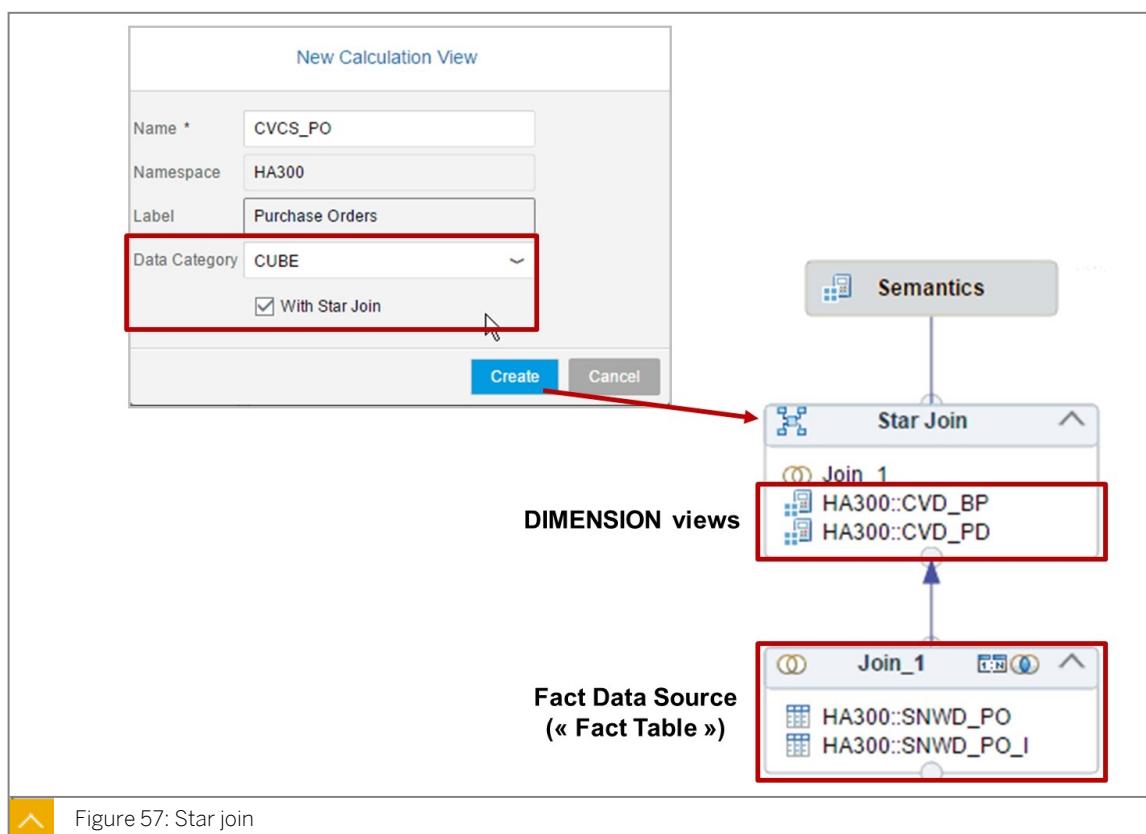
- Use a Star Join in a CUBE calculation view

Star Join Node

If your source data for a graphical calculation view is of a star schema type, you can create a calculation view of type *CUBE with Star Join*.

A *Star Join* node enables you to join the fact data with the descriptive data. The input allowed to the *Star Join* node includes the lower node and all the relevant calculation views of type **DIMENSION**.

This way, you are logically creating a star schema where the join is created from the central entity to the other entities.



During deployment, the *Star Join* node is always deployed with an *Aggregation* node on top of it. The *Star Join* node is deployed first with a series of joins, and then the aggregation logic is deployed. This allows the view to aggregate the measures dynamically, based on the attribute columns you include in the result set.

The *Star Join* node in calculation views supports the Referential Join type, which can speed up the execution of the calculation view when you do not query any column from one or several of the dimension calculation views assigned to the *Star Join* node, because the join will not be executed (at least for the cardinalities 1..1 and n..1).



Caution:

As always, using a Referential Join requires that the referential integrity of the joined tables is ensured.

As already said, in addition to the Facts Data Source (“Fact Table”), the only other possible sources for a *Star Join* node are calculation views of type DIMENSION. But DIMENSION views can also be used in classical CUBE calculation views (without star join) if relevant.



LESSON SUMMARY

You should now be able to:

- Use a Star Join in a CUBE calculation view

Extracting Top Values with Rank Nodes



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Use a rank node to extract the top values of a data set

Rank Node

The purpose of the *Rank* node is to enable the selection, within a data set, of the top or bottom 1, 2, ... n values for a defined measure, and to output these measures together with the corresponding attributes and, if needed, other measures.

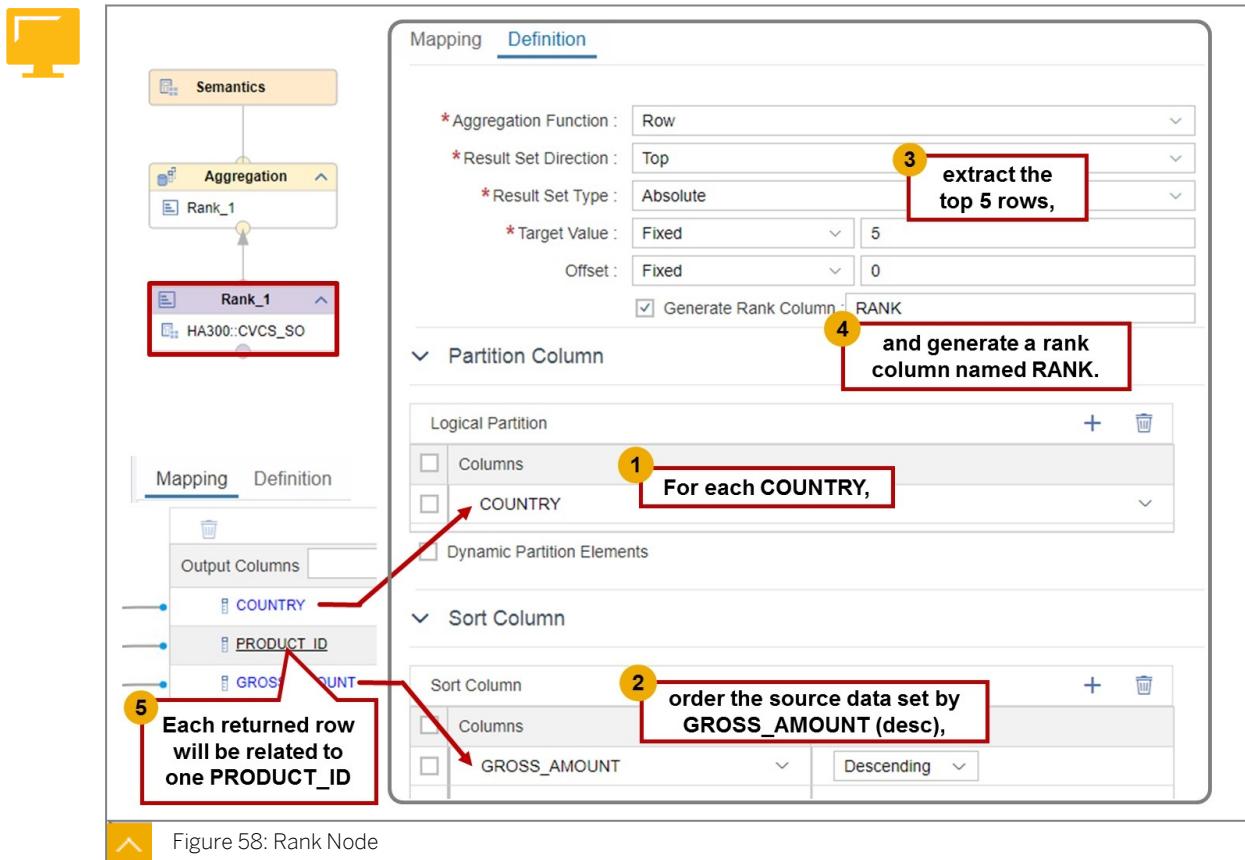
For example, with a *Rank* node, you can easily build a subset of data that gives you the five products that generated the biggest revenue (considering the measure *GROSS_AMOUNT*) in each country. The country, in this example, defines a Logical Partition of the source data set.



Note:

SAP HANA 2.0 SPS04 has introduced more advanced capabilities for *Rank* nodes compared with the previous versions. In particular, it implements several ways to define the number of returned rows for each partition, by combining the Aggregation Function, Result Set Type, Target Value, and Offset. This will be described in detail in the following sections.

The legacy aggregation functions *Top N* and *Bottom N* are just for compatibility purpose for the legacy models. They should not be used for new calculation views.



A rank node computes the source data set as follows:

- If specified, the source data set is partitioned by one or several columns (**Logical Partition**).
- Within each partition, the data is sorted by one or several columns (**Sort Column**).
- An **Aggregation Function** is applied, generating the rank information based on the **Result Set Type** (absolute or percentage).
- A subset of data is returned based on the **Target Value** (and the optional **Offset**) settings, starting from either from the top or bottom of the ordered set (**Result Set Direction**).

Optionally, the rank information can be included in the result set (**Rank Column**).



Caution:

The *Rank* node itself does not perform any type of aggregation on the source data set (this important topic is discussed later on in this lesson).

Main Settings of a Rank Node

The main settings of a *Rank* node are as follows:

Table 9: Main Settings of a Rank Node

Setting	Purpose
Aggregation Function	Define the key computation executed on the data set (see below)

Setting	Purpose
Result Set Direction	Decide whether to extract the Top or the Bottom (Down) rows from the ordered data set
Result Set Type	Absolute or Percentage
Target Value and Offset	Define the number of rows to return
Generate Rank Column	Indicate if you want to output a rank column and specify its name
Logical Partition	Partition the source data set by one or several columns, before executing the rank computation
Dynamic Partition Elements	Define whether the partition can be adjusted automatically based on the columns that are selected by an upper node or an upper view/query that you execute on top of the current one
Sort Column	The columns that are used to order the data set to execute the ranking

Partitioning the Source Data Set

The source data set can be partitioned by one or several columns. This means that the extraction rule you define, for example, return the top five total sales amount, will be executed in each partition, for example, for each Country and each Year.

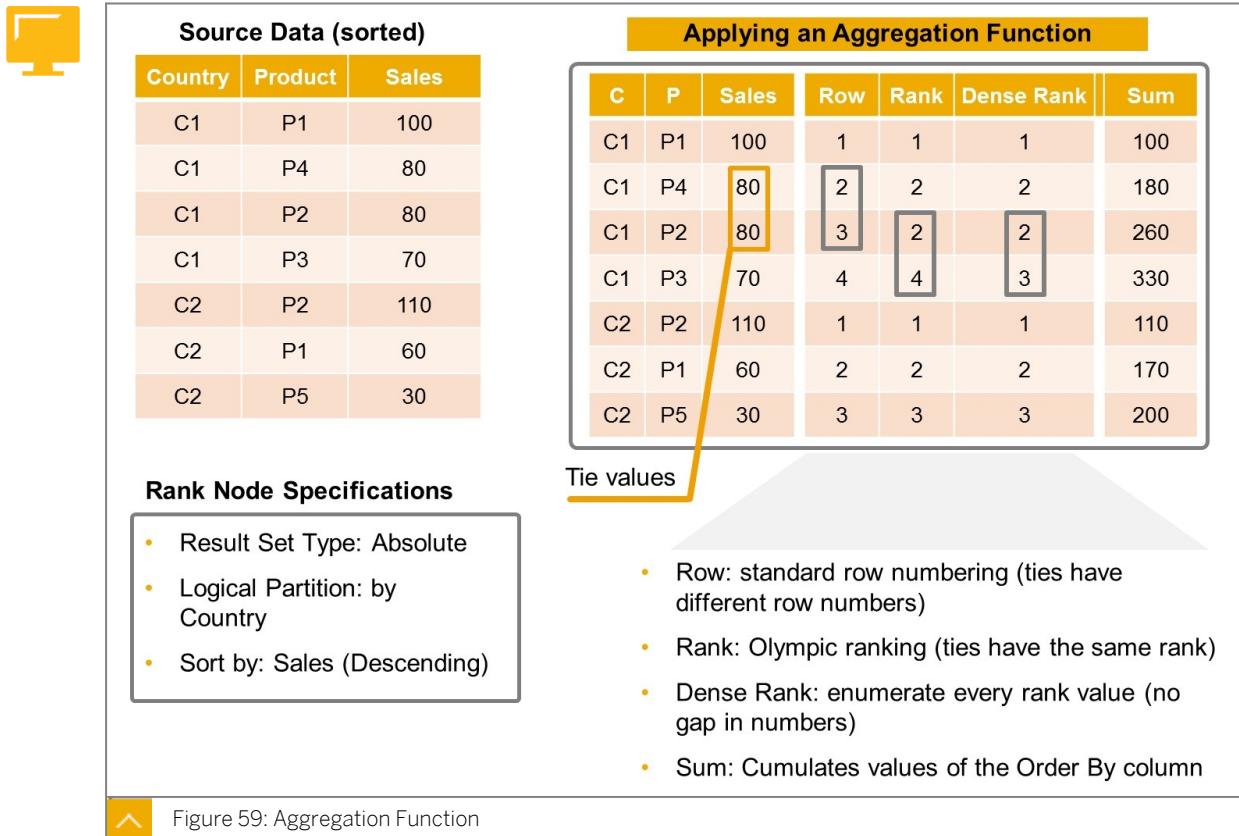
If you choose the *Dynamic Partition Element*, the columns listed in the Partition will be ignored if they are not requested by an upper node or top query. To follow the same example, you could return the top five total sales for each Country only, for each year only, just by excluding the column you do not need from your top query, and without redesigning your calculation view.

Choosing an Aggregation Functions

After partitioning the source data set by one or several columns and ordering it, an Aggregation Function is applied to the data set.

The *Rank* node offers four Aggregation Functions which can be classified into two categories:

- Three functions computing the row numbers. These are:
 - Row
 - Rank
 - Dense Rank
- One function computing the values of the sorted column, for example, a Sales amount. This is:
 - Sum



This diagram shows the behavior of the different aggregation functions on the same data set.

Row, *Rank*, and *Dense Rank* only differ in the way they deal with tie values (identical values in the sorted column). The *Sum* aggregation function generates a cumulative sum of the sorted column up to the current row.

Using Multiple Sort Columns

In scenarios where more than one sort column are defined, they are treated in sequence of appearance for the aggregation functions *Row*, *Rank*, and *Dense Rank*. This can be useful to better handle identical values in the first sort column.

The *Sum* aggregation function only uses the first Sort Column.

Generating the Result Set

The final stage is to extract the result set. This is done based on the *Target Value* setting.



C	P	Sales	Row	Rank	Dense Rank	Sum
C1	P1	100	1	1	1	100
C1	P4	80	2	2	2	180
C1	P2	80	3	2	2	260
C1	P3	70	4	4	3	330

Additional Rank Node Specifications

Aggr. Function	Row		Rank		Sum	
	Target Value	2		2		200

Diagram illustrating the result set definition:

- The first table shows the original data.
- The second table shows the rows selected by the Rank node (Rows 1 and 2).
- The third table shows the sum aggregation result (Sum = 200).

Note: Only the data for customer C1 is shown

Figure 60: Defining the Result Set

The setting must be a rank value (for example, extract the data up to the rank **2**), except for the *Sum* aggregation function where you set a cumulative value (for example, extract the data up to a cumulative Sales amount of **200**).

The target value can be fixed, that is, defined in the Rank Node definition, or it can be set at runtime by means of an input parameter.



Note:

The rank node returns the rows for which the computed ranking is **lower or equal to** the target value.

Working with Percentage Instead of Absolute Values

Instead of *Absolute*, which has been used up to now in our examples, you can set the *Result Set Type* to *Percentage*.

The figure, Percentage Result Type, shows the difference between the two methods.



Aggr. Function: Row					Aggr. Function: Rank				
C	P	Sales	Absolute	Percentage	C	P	Sales	Absolute	Percentage
C1	P1	100	1	.25	C1	P1	100	1	.25
C1	P4	80	2	.5	C1	P4	80	2	.5
C1	P2	80	3	.75	C1	P2	80	2	.5
C1	P3	70	4	1	C1	P3	70	4	1

Target Value: 0.5

Aggr. Function: Dense Rank					Aggr. Function: Sum				
C	P	Sales	Absolute	Percentage	C	P	Sales	Absolute	Percentage
C1	P1	100	1	.33	C1	P1	100	100	.30
C1	P4	80	2	.66	C1	P4	80	180	.55
C1	P2	80	2	.66	C1	P2	80	260	.79
C1	P3	70	3	1	C1	P3	70	330	1

Target Value: 0.5

Note: Only the data for customer C1 is shown

Figure 61: Percentage Result Set Type

With the *Percentage* Result Set Type, you can address requirements such as:

- Return the 50% best-selling products for each customer (with a *Row* aggregation function, for example).
- Return the best-selling products representing 30% of the total sales for each customer (with a *Sum* aggregation function).

The figure, Percentage Result Type, shows in a frame which rows would be included in the extracted data set.

Defining an Offset on the Result Set

It is possible to exclude a number of elements from the top of the sorted data set, by defining an offset.



Rank Node Specifications

- Result Set Type: Absolute
- Logical Partition: by Country
- Sort by: Sales (Descending)
- Target Value: 2
- Offset: 2

C	P	Sales	Row	Rank	Dense Rank
C1	P1	100	1	1	1
C1	P4	80	2	2	2
C1	P2	80	3	2	2
C1	P3	70	4	4	3

Additional Rank Node Specifications

Aggr. Function	Row	Rank	Dense Rank
C	P	Sales	Row
C1	P2	80	3
C1	P3	70	4

C	P	Sales	Rank
C1	P3	70	4

C	P	Sales	DenseRank
C1	P3	70	3

Note: Only the data for customer C1 is shown

Figure 62: Defining an Offset

In this example, the rows ranked up to (and including) 2 are excluded. Then the rank node returns the following rows based on the target value you have set.

Another example is when you want to assign members of a statistic collection to their inter-quartile interval. You could define a rank node as follows:

- Aggregation Function: Row
- Result Set Type: Percentage
- Target Value: 0.5
- Offset: 0.25

Is an Aggregation Needed Before Ranking?

As already discussed, the *Rank* node can partition the source data, but it does NOT perform any aggregation on the source data.

In other words, the way the source data is structured has a major impact on the way the *Rank* node will compute this data.

For example, if your source data contains a Sales Order ID column, you might not process directly a ranking of best-selling products in each country, because there might be several rows for the same Country and Product (but different Sales Orders).



Source Data Set

Country	Product	Sales Order ID	Sales
C1	P1	2	100
C1	P2	2	90
C1	P2	5	80
C1	P3	1	70
C2	P2	4	110
C2	P1	6	60
C2	P5	6	30

What if you do not want the Sales Order ID in your Calc View Output?

Option 1: Do not aggregate

Country	Product	Sales
C1	P1	100
C2	P2	110

" In C1, the biggest sales order detail (line item) is 100, and the corresponding product P1 "

Option 2: Aggregate data by Country and Product before ranking

Country	Product	Sales
C1	P2	170
C2	P2	110

" In C1, the best-selling product is P2 "

Figure 63: Rank Node and Aggregation

When the source data for the *Rank* node is a table, you must make sure that the data structure suits your modeling needs. If not, you might need to first aggregate data by adding an *Aggregation* node used as a data source by the *Rank* node.

When the source data for the *Rank* node is a CUBE calculation view, the aggregation defined in that calculation view is implicitly triggered based on the columns requested by the rank node, but a column that is mapped to the output of the *Rank* node but not consumed by the upper node will not be requested to the data source.



Caution:

Like for other types of nodes, some columns are passed to the upper nodes even when they are not requested. For example, this is the case when the source calculation view has a sort order defined in the *Semantics* node (all the columns used for sorting are passed to consuming views). This is also the case when special settings such as *Keep Flag* are used.

Assigning a Type to the Rank Column

Depending on how you want to use the ranking information, you can decide to assign to the rank column the type *Attribute* or *Measure* in the *Semantics* node.

Rank Column: Measure or Attribute?



- Attribute

Assigning the type *Attribute* is a simple approach.

It is probably a bit less error-prone, because you are not tempted to perform an irrelevant aggregation of ranking positions.

- Measure

With the type *Measure*, the rank column can provide some flexibility.

For example, if you set the default aggregate function to MAX, you can retrieve summarized data, such as the total sales generated by the five biggest orders in each country, while keeping the information about how many orders are actually totaled in each country. Indeed, there might be countries that have received less than five orders over the considered period, and this information could be of interest when analyzing the data.

Because the information in the rank column can differ a lot depending on the aggregation function and result set type you used, you might want to give it an explicit name if you find that *Rank_Column* (default column name) is too generic.



LESSON SUMMARY

You should now be able to:

- Use a rank node to extract the top values of a data set

Learning Assessment

1. What is the purpose of the *Projection* node?

Choose the correct answers.

- A To apply filters on the data
- B To extract only the required columns from a data source
- C To aggregate measures
- D To join data sources

2. Why do you use a Referential Join?

Choose the correct answer.

- A To improve performance of joins that do not need to check integrity if no column from the right table is selected by a query.
- B When you need to display only facts that have associated master data attributes.
- C When you want to display facts even when the associated master data attributes are missing.

3. A Text Join is a Right Outer Join used to join a text table containing multi-language descriptions, to an attribute in a dimension.

Determine whether this statement is true or false.

- True
- False

4. Identify the type of join that ensures records are only selected if they fit within a specified date range.

Choose the correct answer.

- A Left Outer Join
- B Right Outer Join
- C Temporal Join
- D Full Outer Join

5. In a union, you do not have to select all source columns and the source columns do not have to have the same name.

Determine whether this statement is true or false.

- True
- False

6. You have customers who appear in table A and sometimes they also appear in table B. You want to select the customers that appear ONLY in table A. Which node type do you use?

Choose the correct answer.

- A Intersect
- B Minus
- C Union

7. In an *Aggregation* node, a calculated column is always computed before the aggregate function.

Determine whether this statement is true or false.

- True
- False

8. Which flags are relevant to the control of the aggregation behavior?

Choose the correct answers.

- A Keep Flag
- B Transparent filter
- C Dynamic join

9. In a *Star Join* node, what do you join to the central entity (fact table)?

Choose the correct answer.

- A CUBE calculation views
- B Column tables
- C DIMENSION calculation views

10. Which are three valid aggregation functions for a rank node?

Choose the correct answers.

- A Row
- B Rank
- C Sum
- D Sort

Learning Assessment - Answers

1. What is the purpose of the *Projection* node?

Choose the correct answers.

- A To apply filters on the data
- B To extract only the required columns from a data source
- C To aggregate measures
- D To join data sources

Correct — Projection nodes are used to apply filters on the data and extract only the required columns from a data source. They are not used to aggregate or join data sources. There are dedicated nodes for those tasks.

2. Why do you use a Referential Join?

Choose the correct answer.

- A To improve performance of joins that do not need to check integrity if no column from the right table is selected by a query.
- B When you need to display only facts that have associated master data attributes.
- C When you want to display facts even when the associated master data attributes are missing.

Correct — Referential Joins are used to improve performance of joins that do not need to check integrity if no column from the right table is selected by a query.

3. A Text Join is a Right Outer Join used to join a text table containing multi-language descriptions, to an attribute in a dimension.

Determine whether this statement is true or false.

- True
- False

Correct — A Text Join is a **Left** Outer Join used to join a text table containing multi-language descriptions, to an attribute in a dimension.

4. Identify the type of join that ensures records are only selected if they fit within a specified date range.

Choose the correct answer.

- A Left Outer Join
- B Right Outer Join
- C Temporal Join
- D Full Outer Join

Correct — The Temporal Join is a special join that uses date range information from the left table in order to join the relevant data from the right table.

5. In a union, you do not have to select all source columns and the source columns do not have to have the same name.

Determine whether this statement is true or false.

- True
- False

Correct — Unions are very flexible You do not have to select all source columns and the source columns do not have to have the same name.

6. You have customers who appear in table A and sometimes they also appear in table B. You want to select the customers that appear ONLY in table A. Which node type do you use?

Choose the correct answer.

- A Intersect
- B Minus
- C Union

Correct — An *Intersect* node would select only customer who appear in both tables. A *Union* node would select all customers if they appeared in any table.

7. In an Aggregation node, a calculated column is always computed before the aggregate function.

Determine whether this statement is true or false.

- True
- False

Correct — In an *Aggregation* node, a calculated column is always computed **after** the aggregate function.

8. Which flags are relevant to the control of the aggregation behavior?

Choose the correct answers.

- A Keep Flag
- B Transparent filter
- C Dynamic join

Correct — Use **Keep Flag** and **Transparent Filter** to control the aggregation behavior.

Dynamic Join is used to optimize joins, not aggregations. —

9. In a *Star Join* node, what do you join to the central entity (fact table)?

Choose the correct answer.

- A CUBE calculation views
- B Column tables
- C DIMENSION calculation views

Correct — You are only able to join DIMENSION calculation views in a *Star Join* node.

10. Which are three valid aggregation functions for a rank node?

Choose the correct answers.

- A Row
- B Rank
- C Sum
- D Sort

Correct — Row, Rank and Sum are three valid aggregation functions for a rank node. Sort is not an aggregation function.

Lesson 1

Creating Restricted and Calculated Columns

101

Lesson 2

Filtering Data

111

Lesson 3

Using Variables and Input Parameters

119

Lesson 4

Implementing Hierarchies

135

Lesson 5

Implementing Currency Conversion

149

Lesson 6

Defining Time-Based Dimension Calculation Views

157

UNIT OBJECTIVES

- Create restricted and calculated columns
- Filter data
- Implement variables
- Define input parameters
- Define Value Help Views
- Map Variables and Input Parameters
- Define Hierarchies
- Explain the general principles of currency conversion
- Create a Time-Based Dimension Calculation View

Unit 3

Lesson 1

Creating Restricted and Calculated Columns

LESSON OVERVIEW

This lesson explains how to use different modeling capabilities on columns. In particular, you will learn how to create calculated columns (attributes or measures) and restricted columns (for attribute only).

Business Example

You want to ensure that you minimize the processing in the client tool, pushing down expressions to SAP HANA using calculated and restricted columns.



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Create restricted and calculated columns

Restricted Columns



What is a restricted column?

- As the name implies it is a column that does not give the complete picture of a column, it is restricted to a **subset** of the original column.
- The benefit of a restricted column is that it expands the modeling options in a view, giving the modeler the possibilities of creating objects that can be easily reported on or reused.

Country	Month	Amount	US Amount
DE	2010-01	12.345,00	0
DE	2010-02	15.678,00	0
DE	2010-03	25.814,00	0
DE	2010-04	21.586,00	0
DE	2010-05	21.861,00	0
DE	2010-06	11.258,00	0
DE	2010-07	12.387,00	0
DE	2010-08	13.589,00	0
DE	2010-09	12.345,00	0
DE	2010-10	15.678,00	0
DE	2010-11	25.814,00	0
DE	2010-12	21.586,00	0
US	2010-01	21.861,00	21.861,00
US	2010-02	11.258,00	11.258,00
US	2010-03	12.387,00	12.387,00
US	2010-04	13.589,00	13.589,00
US	2010-05	12.345,00	12.345,00
US	2010-06	15.678,00	15.678,00
US	2010-07	25.814,00	25.814,00
US	2010-08	21.586,00	21.586,00
US	2010-09	21.861,00	21.861,00
US	2010-10	11.258,00	11.258,00
US	2010-11	12.387,00	12.387,00
US	2010-12	25.814,00	25.814,00

Restricted column based on **Amount**, where **Country = US**

Figure 64: The Benefits of Restricted Columns

The restricted column is one of the column types available in SAP HANA models, which include the following:

- Columns
- Calculated columns
- Restricted columns

The restricted column is restricted based on one or more attributes. These columns can be anything in the base table or view that the modeler defines to help reporting or further modeling.



Note:

The restriction criteriacannot be based on a column defined as a *Measure* in the semantics. Only columns of type *Attribute* can be used.

Example Without Using a Restricted Column



- You have a transactional table with cost data items, with each cost type split on a different line.
- If you want to find out the shipping cost you could create an analytic view with Cost Type as an attribute, and Amount as a measure.
- You could then restrict your report by reporting on Cost Type, setting the column filtered on Cost Type = “Shipping Cost” in the reporting tool.

Cost Type	Amount
Purchasing Price	€ 1 200
Shipping Cost	€ 80
VAT	€ 346
Processing Cost	€ 150
Margin	€ 300



Figure 65: Example Without Using a Restricted Column

With this data, you could restrict your report by filtering by *Cost Type* = *Shipping Cost*. If your reporting tool is SAP Business Objects, you could create a data provider with a query restriction where *Cost Type* is filtered to only display *Shipping Cost*.

Creating Restricted Columns



- A restricted column can be created in an Aggregation or Star Join node of a calculation view.
- You assign a measure to the restricted column, and also which columns define the restriction.
Example: restricted column based on Gross Amount, restricted to certain product categories.

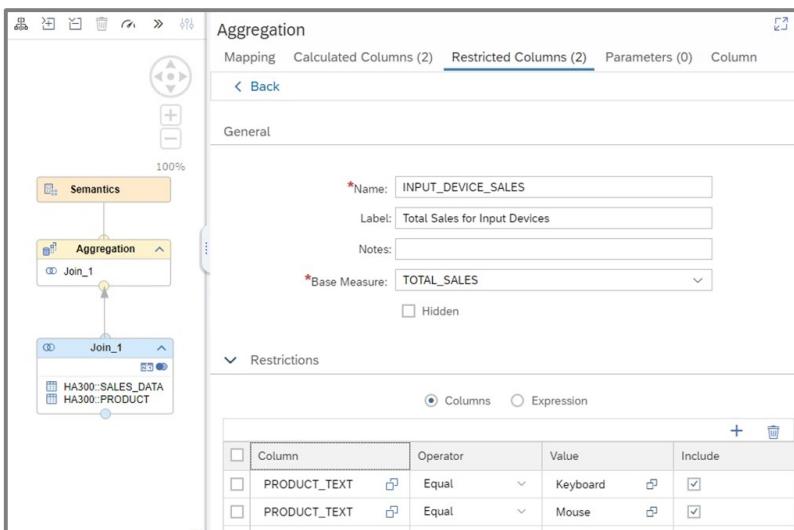


Figure 66: Creating Restricted Columns

Setting the Restriction

Continuing with the example of using SAP Business Objects for reporting, when you have access to this restricted column, you can report on both the total gross amount and the gross amount for flat screens, in the same data provider or query.



- The restrictions for a restricted column do not have to be limited to one single Column: you can filter based on multiple columns depending on your reporting requirements.
- Multiple operators can be used for the filter restrictions, such as Equal, Greater Than, Less Than and others.
- You have the option to hide the restricted column, for example to reuse it in a calculated column and thereby make it unavailable for reporting.

Column	Operator	Value	Include
PRODUCT_TEX	Equal	Keyboard	<input checked="" type="checkbox"/>
PRODUCT_TEX	Equal	Mouse	<input checked="" type="checkbox"/>
	Between		
	Equal		
	GreaterEqual		
	GreaterThan		
	Is Not Null		
	IsNull		
	LessEqual		
	LessThan		

Figure 67: Setting the Restriction

If there are different lines in the restriction, all the lines defined on the same column are combined with the logical operator *OR*, and then, all the sets of restrictions for different columns are combined with the logical operator *AND*.



Note:
This is the case regardless of the order in which you define the lines.

Displaying and Editing Restriction Expressions

Another option to define or edit the restrictions is to use an Expression. This provides more flexibility when the standard operators for the column-based, "graphical", restrictions do not fulfill your needs. The expression can use column names, operators, input parameters. From SAP HANA 2.0 SPS05 onwards, the restricted column expression can also include Functions.



Note:
For example, extracting the year **YYYY** from a date column like **YYYY-MM-DD** with a string or date function is now possible.



- Several restrictions on the same column are combined with an **OR** operator.
- Restrictions on different columns are combined with an **AND** operator.
- To visualize the expression, select the *Columns* tab.
- You can also edit the expression to enrich it.

In this case, the column-based definition can no longer be edited.

Figure 68: Displaying and Editing Restriction Expressions

You can visualize the expression corresponding to your restrictions. This can be useful, for example, when you want to check the precedence of logical operators.

It is also possible to modify this expression, especially in complex scenarios when the features offered in the *Column* tab do not fulfill your requirements.

As of SPS11, the expression can be written using SQL, in addition to the Column Engine syntax. However, for expressions in SQL, SAP HANA modeling supports only a limited list of SQL functions.



Note:

The general SQL support within expressions is not specific to restricted measure expressions. SQL can also be used in the following expressions:

- Calculated columns (since SAP HANA SPS10)
- Filters (new in SPS11)
- Default values for variables and input parameters (new in SPS11)

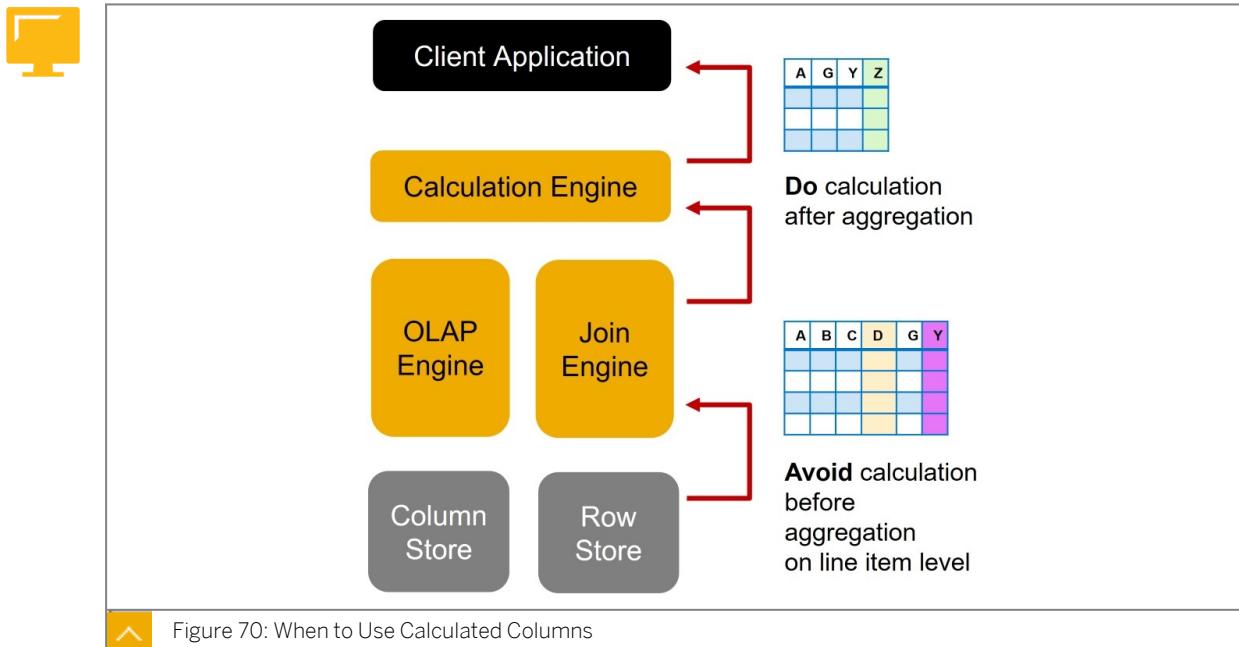
You will learn about these different types of modeling functions later.

Calculated Columns



Figure 69: When to Use Calculated Columns

When to Use Calculated Columns



Hint:

It is recommended to create calculated columns only when there is a specific reporting need. Indeed, even if the calculation engine benefits from regular improvements in terms of performance, processing a calculated column can sometimes be costly, especially when it is complex or involves a lot of data.

The Calculated Columns Wizard

When creating a calculated column, the first step is to choose the column type, measure, or attribute.



- A calculated column is defined within an information model and can use the string functions, mathematical functions etc. available in the editor.
- You can define a calculated column as measure or attribute.

The screenshot shows the SAP BusinessObjects Data Services Studio interface. On the right, the 'Expression Editor (INPUT_DEVICE_SALES_SHARE)' is open with the expression `"INPUT_DEVICE_SALES" / "TOTAL_SALES"`. On the left, the 'Aggregation' dialog box is displayed, showing a semantic model diagram with nodes like 'INPUT_DEVICE', 'HA200 SALES', and 'HA200 PROD'. The 'Calculated Columns (1)' tab is selected, listing the column 'INPUT_DEVICE_SALES_SHARE' with a label 'Percentage Share of Input'. The 'GENERAL' tab shows the SQL expression `"INPUT_DEVICE_SALES"/"TOTAL_SALES"`.

- Double-click or drag-and-drop Elements, Operators and Functions to build the expression
- When you type, use Ctrl+Space to autocomplete the name of functions, columns, input parameters.

Figure 71: Calculated Columns Wizard

Consider Granularity When Creating Calculated Columns



Product	Units	Price	Total Sales (Units * Price)
Keyboards	100	€ 40	€ 4 000
LCD Screens	50	€ 300	€ 15 000
Network Switches	75	€ 90	€ 6 750
VOIP Telephones	200	€ 65	€ 13 000
Servers	30	€ 2 000	€ 60 000
SUM:		455	€ 1 135 225

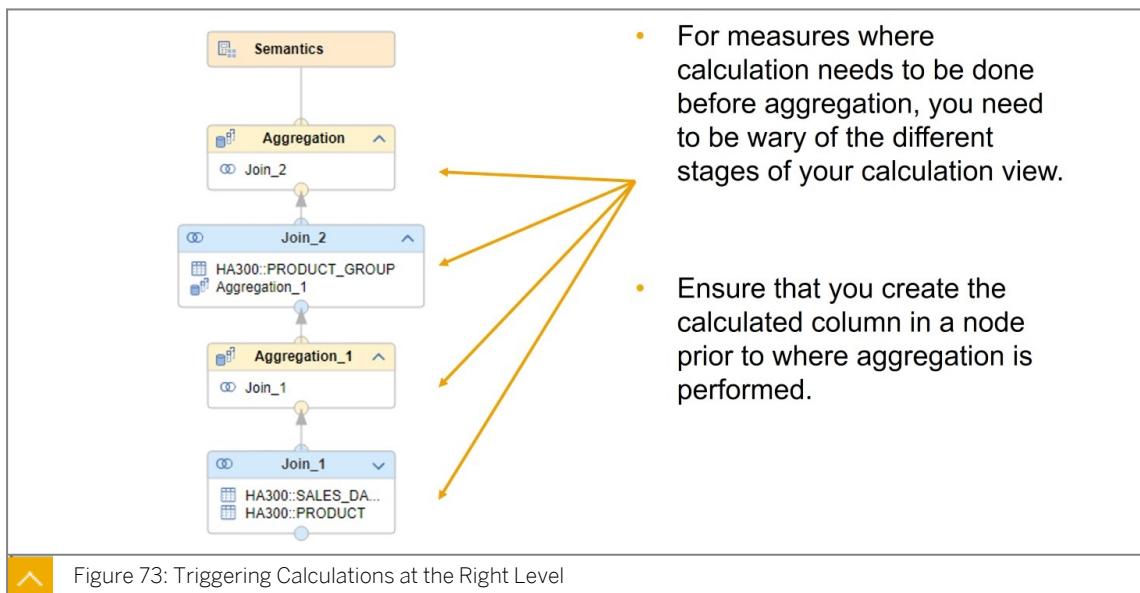


- For certain measure it is not possible to perform the calculations when the measures are already aggregated. The aggregated granularity of, for example, Price does not mean anything.

Figure 72: Consider Granularity When Creating Calculated Columns

In the figure, Consider Granularity when Creating Calculated Columns, in the sum line highlighted in red, the units as well as the price have been aggregated. Multiplying these to aggregates does not give a meaningful result.

Triggering Calculations at the Right Level



By analyzing your reporting requirements, you can arrive at a decision at which stage the calculation should be performed.



Caution:

Try to minimize calculations before aggregation; for example, when calculations include multiplication or division.

If the calculations are just additions or subtractions, it is not required. It will also slow down the execution of the view.

Calculation Before Aggregation



Figure 74: Calculation Before Aggregation

Product	Units	Price	Total Sales (Units * Price)
Keyboards	100	€ 40	€ 4 000
LCD Screens	50	€ 300	€ 15 000
Network Switches	75	€ 90	€ 6 750
VOIP Telephones	200	€ 65	€ 13 000
Servers	30	€ 2 000	€ 60 000
SUM:			€ 98 750

- This way you can be sure that you end up with a correct sum as the calculations are performed on the correct granular level.

Calculated Columns and Persistence

Creating calculated columns in an SAP HANA calculation view does not mean the calculation is persisted; it is simply projected through the generated column view.

The calculated column is available only during the runtime of the model and can be displayed in a report, or consumed by another view.

By contrast, if you need to keep the result of a calculation over time, you can apply one of the following methods:

- Create an ad-hoc SQL artifact that updates and inserts the calculated column into a table.
You can use a write-enabled procedure.
- Use the transformation and calculation features provided by an ETL tool, such as SAP Landscape Transformation (SLT) or SAP Data Services.
You can pre-calculate columns during the data provisioning phase.

Expression Language and Validation

As of SAP HANA SPS10, you can specify which of the two following languages the expression of a calculated column uses:



- SQL
The expression only uses plain SQL.
- Column Engine
The expression uses any valid SQLScript expression.

Validating an Expression Against a Specific Language: Key Benefits



- It helps to validate the syntax of calculated columns according to the specified language.
For example, an expression with an IF condition (supported only with SQLScript) will not validate if it is defined as SQL.
- In turn, it helps the modeler to optimize the calculation execution.
Indeed, plain SQL expressions enable a better optimization process, compared to SQLScript. So, by validating the expression against SQL, you can make sure that it is fully optimized.



Note:

SAP HANA SPS11 has introduced the same distinction for filters, restricted columns, and default value expressions for variables and input parameters.

Client Side Aggregation

If you want to create a calculated measure and enable client side aggregation for the calculated measure, select the *Enable client side aggregation* checkbox.

This allows you to propose the aggregation that the client needs to perform on calculated measures.

The screenshot shows the 'Aggregation' configuration screen. In the 'Calculated Columns (1)' tab, there is one entry: 'INPUT_DEVICE_SHARE' with the label 'Percentage Share of Input'. The 'GENERAL' tab displays the following details:

- Name:** INPUT_DEVICE_SHARE
- Label:** Percentage Share of Input Devices out of the Total Sales
- Notes:** (empty)
- Data Type:** DECIMAL
- Length:** 4
- Scale:** 2

The 'SEMANTICS' tab shows:

- Column Type:** Measure
- Aggregation type:** MAX (circled in red)
- Hidden:** (unchecked)
- Enable client side aggregation:** (checked)

The 'EXPRESSION' tab contains the SQL expression: "'INPUT_DEVICE_SHARE' / 'TOTAL_SALES'".

Figure 75: Client Side Aggregation

Consider an example where you have created the measure MAXIMUM_GROSS_AMOUNT, giving you the maximum value of the gross amount. By selecting the *Enable client side aggregation* checkbox, you propose to the reporting client to also apply a maximum aggregation on the client side as defined in the *Client Aggregation* drop-down list.



LESSON SUMMARY

You should now be able to:

- Create restricted and calculated columns

Filtering Data

LESSON OVERVIEW

Most of the time, users do not need the whole data perimeter stored in the source system. In these instances, filters can be used to display only relevant data. Moreover, by decreasing data volume, performance can be improved.

Business Example

You notice that a large amount of data stored in base tables are never used for reporting, because users decide to apply WHERE clause filters in reporting tools.

In order to speed up query execution you decide to introduce filters into information models.



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Filter data

Using Filter Operations

Filtering data is often required when analyzing data, in order to reduce the result set.

Typically, you might want to retrieve the sales details for a particular country or region, for a particular range of products, or specific customers (new customers, customers who haven't ordered any product for more than one year). These three examples are all about data filtering based on attributes.

On the other hand, you might want to filter data based on measures. For example, you might want to list only the sales order for which the total amount exceeds a threshold.

The two approaches can also be combined, for example, when you want to retrieve the list of US customers who have posted more than 10 orders during the past month. In this case, filtering the data by customer country (US) can be done as early as possible, but you must compute the total number of orders per customer (for last month) before applying the threshold (10).

So, filtering is sometimes a question of mitigating between performance (filtering as early as possible) and the consistency of the result you get (filtering too early without care can lead to wrong results). This is particularly true when data is aggregated.

Filter Criteria: Hard-coded or Flexible?

From a functional perspective, a key question when defining filters is whether they are hard-coded in the calculation view, or provide a more flexible way to define the filter criteria at runtime. In a number of cases, it is possible to execute a query with a client tool on top of a calculation view, including filtering criteria.

A primary goal during modeling is to minimize data transfers between engines. This statement holds true both internally, that is, in the database between engines, and also between SAP HANA and the end user client application. For example, an end user will never

need to display a million rows of data. Such a large amount of information just cannot be consumed in a meaningful way.

Whenever possible, data should be aggregated and filtered to a manageable size before it leaves the data layer.

When deciding which records should be reported upon, a best practice is to think at a set level, not a record level.

A set of data can be aggregated by a region, a date, or some other group, to minimize the amount of data passed between views.

Filters in Calculation View Nodes

In Calculation Views, filters can be applied to many node types. These include the following:

- Projection
- Union
- Join
- Rank
- Aggregation
- Star Join

As a consequence, you generally do not need to use a Projection Node for the sole purpose of filtering data.



Note:

Defining filter expressions on Aggregation and Star Join nodes is a new capability of SAP HANA 2.0 SPS03.

To define a filter, you display the *Filter Expression* tab and define a valid SQL or Column Engine expression. It is recommended to use SQL expressions whenever possible, because they most often allow better optimization.

An example of filter expression is "`COUNTRY='US'`". The expression can combine several columns and logical operators; for example, "`COUNTRY='US' OR COUNTRY='CA'`".



Caution:

When a column from a data source, for example `COUNTRY`, is mapped to the node output with a different name, for example, `COUNTRY_CODE`, the **output column name** must be used in the expression. In this case, the correct expression would be something like "`COUNTRY_CODE='US'`". If you enter the expression "`COUNTRY='US'`", the expression might validate but building the calculation view will fail.

The name of a column that is used in a filter expression cannot be changed. This is to avoid creating inconsistencies. If you need to change a column name, first remove the filter expression, and set the (adapted) filter expression back after the name has been changed.

Where and When is the Filter Applied?

The behavior of filter expressions depends on the node where the filter is defined.

- Filters defined in the top node

In the **top node**, a filter expression is generally applied **before** the data processing defined in this node. For example, in an Aggregation or Star Join node, the filter is applied to the data **before** the aggregation is executed.

- Filters defined in other nodes
- In **other nodes** (NON top nodes), the filter expression is generally applied to the output of the node.

In some cases, a filter expression can be defined in one or another node of the calculation scenario without impacting its output. In other cases, however, particularly when defining filters on measures, the position of the filter expression in the calculation scenario can affect the result set.

Filtering the Data Set During Data Preview

When you query a Calculation View, all the filters are applied. Some of them can be pushed down to lower levels by the optimizer; in that case, the optimizer tries to ensure data consistency. There are additional ways to further restrict the result set at runtime:

- Define filters graphically, with the *Add Filter* button in the ribbon of the *Raw Data* tab
When applying the filters, the query against the entire data set is re-executed
In the corresponding query, you can, in turn, edit some filters or add new ones (see below)
- Edit the default Data Preview SQL Query to add filters and execute the modified query.
A filter on an attribute is written as a *WHERE* clause. A filter on a measure is written as a *HAVING* clause (filtering based on an aggregate column) when the query include a *GROUP BY* clause; otherwise as a *WHERE* clause.
- An additional filtering capability is offered in the *Raw Data* tab, in the dropdown list at the right of each column header.

This is a text filter (matching the entered text string against the column's values) and it applies to the current data set: the query is not re-executed against the source calculation view to find all matching rows. So, this filtering functionality is more suited for activities such as basic testing.

Ignore Multiple Output for Filters

Nodes that are consumed by multiple nodes in a calculation view prevent filter push-down from consuming nodes. The purpose of this is to ensure semantic correctness.

However, if push-down does not violate semantic correctness, it most often makes the calculation view execution time shorter.

So, the flag *Ignore Multiple Outputs For Filter* can be set to enforce filter push-down. Its impact is local to the node where it is set.

**Note:**

A similar setting can be defined at the View Level, in the *Advanced Properties* tab of the *Semantics* node. It has the same effect, but applies to scenarios where one calculation view is consumed multiple times by another.

You will learn more about Filter push-down in the lesson, *Implementing Good Modeling Practices*.

Using Variables and Input Parameters for filtering purposes

Variables are also typically used in calculation views to filter data. They are a powerful filtering method, because you can define the filter condition at runtime; that is, when executing the view.

However, compared with filter expressions, variables allow you to filter a data set only by attributes, not by measures. To implement filtering on measures columns at runtime, you will need to use an input parameter and use the parameter value in a filter expression.

You will learn about Variables and Input Parameters later, in a dedicated lesson, *Using Variables and Input Parameters*.

Client-Dependent Views

The *CLIENT* (sometimes also called SAP Client to avoid any confusion with the concept of Customer) is a general concept in SAP Systems such as SAP Business Suite and SAP Business Warehouse. The main purpose is to isolate different types of data (for example, development versus test data) based on a specific column (generally named *CLIENT* or *MANDT*, which stands for *Mandant* (a German word for *Client*)). The values in this column are three-digit numbers, such as 001, 200, 800.

Almost all tables in the SAP Business Suite and SAP S/4HANA data models are client-dependent. Only a small number of tables do not have the client as the first primary key field.

SAP HANA handles the SAP Client column to enable client filtering when you work on SAP Applications such as the SAP ERP.

In the SAP Web IDE for SAP HANA, you have to explicitly define, in the properties of a source table, which column holds the SAP Client information.

**Note:**

In SAP HANA Studio, the client columns handling is different. In particular, a column named *CLIENT* or *MANDT* is automatically considered as an SAP Client column, and filtering settings can be defined for each Calculation View without the need to specify explicitly which column holds the SAP Client info in each data source.

Whenever you create SAP HANA models on SAP data sources that include a Client column, we recommend that you always define joins on this column. Otherwise, your view might return inconsistent results if there is data for several clients. For example, you might join data for clients 200 and 800 (cartesian product), which in general would not make sense.

You should also pay attention to aggregations when the SAP Client is not part of the requested columns, because this might result in an irrelevant aggregation.

Defining Client-Specific Filtering in Calculation Views

Client-specific filtering makes sense only when some (or all) of the source tables have an SAP Client column.

1 Define the Client Column in the properties of the table

The screenshot shows the SAP Web IDE interface. A red box highlights the 'HA300::CITIES' entry in the 'Data Sources' list under the 'Mapping' tab. An arrow points from this box to a callout box labeled '1 Define the Client Column in the properties of the table'. Another callout box shows the 'PROPERTIES' dialog for the 'HA300::CITIES' node, where the 'Client Column' is set to 'MANDT'.

2 Set the Default Client property to Session Client

The screenshot shows the 'Semantics' node of the Calculation View. A red box highlights the 'Default Client' dropdown, which is set to 'Session Client'. An arrow points from this box to a callout box labeled '2 Set the Default Client property to Session Client'.

3 In the Security (e.g. SAP HANA Studio), assign a Session Client to the user.

The screenshot shows the SAP HANA Studio security configuration screen. A red box highlights the 'Session Client' field, which is set to '800'. An arrow points from this box to a callout box labeled '3 In the Security (e.g. SAP HANA Studio), assign a Session Client to the user.'

When the user runs a query based on this model, the user's Session Client (in this example: 800) is used to filter the source data.

Figure 76: Creating a Client-Dependent View

By default, a calculation view defined in the SAP Web IDE returns all the values, regardless of the SAP Client value. To implement client-based filtering, you proceed as follows:

- Define, for all relevant data source tables, the SAP client column.

This is done in the *Mapping* tab of the node where the source table is referenced, with the *Client Column* property.

- Define how you want to retrieve data.

This is done in the *Semantics* node of the Calculation View. You choose one of the three following options for the *Default Client* setting:

- **Cross-Client:** All the data is retrieved regardless of the client number.
- **Fixed Client:** You specify an SAP Client number (for example, 200) and the data sources are automatically filtered to include only the rows with this client number.
- **Session Client:** The source tables are filtered dynamically based on a client value that is specified for each user in the *USERS* table of SAP HANA.

The *Session Client* is the default setting applied to a new view.

Let's consider a simple table that contains data for two different clients, 200 and 800. This information is stored in a column named *MANDT*. You create and build a calculation view based on this table, and then query the data with your user. The following table shows what data is retrieved depending on both the default client setting for the calculation view and

whether or not the *MANDT* column has been defined as the (SAP) Client Column in the view properties.

Table 10: Default Client Setting Effect

Default Client Setting	Effect when NO Client Column is specified	Effect when <i>MANDT</i> is defined as the Client Column
Cross-Client	All rows	All rows
Fixed Client: 200	All rows	Only rows for which <i>MANDT</i> =200
Session Client (use case 1: the user has no default client assigned)	All rows	No rows at all
Session Client (use case 2: the user has default client 800)	All rows	Only rows for which <i>MANDT</i> =800

Important Note Regarding the Technical User in the SAP Web IDE

In the SAP Web IDE for SAP HANA, you connect with a regular XS Advanced user, but most of the tasks you perform with defining a calculation view and previewing its data are executed on your behalf by a technical user.

Because the technical user has no default client assigned, if you specify a *Column Client* property for a calculation view and set the *Default Client* to *Session Client*, the data preview in the Developer perspective will not retrieve any data.

There are two alternative approaches that can be used to bypass the technical user. These are as follows:

- You can query the data from an external tool with your regular user (for example, *STUDENT##*)
- You can display the column view data in the *Database Explorer* perspective from a connection to the classic database (for example, in the training environment, *H00 DB*). A query from the classic database connection does not involve the technical user.

This can be done either by opening the column view or by executing an SQL query on top of it.

Session-Dependent Functions

To dynamically select table data in calculation views, you can use several functions. These functions return client or locale information for the connected user, and are as follows:



- `SESSION_CONTEXT('CLIENT')`
Returns the client's value based on the current user profile.
- `SESSION_CONTEXT('LOCALE')`
Returns the session's language in POSIX format (set by 'locale' parameter of JDBC/ODBC/ODBO connection, for example, en_en, de_de, de_at).
- `SESSION_CONTEXT('LOCALE_SAP')`

Returns the session's language following the SAP internal format (like the SPRAS column in the text tables of Master Data).

Example

```
SELECT field1, field2 FROM my_table WHERE mandt =  
SESSION_CONTEXT('CLIENT') AND spras = SESSION_CONTEXT('LOCALE_SAP')
```

It is also possible to refer to the system variable `$$client$$` in a filter expression of a Calculation View. This variable picks up the client value stored in the user ID.



LESSON SUMMARY

You should now be able to:

- Filter data

Using Variables and Input Parameters

LESSON OVERVIEW

This lesson explains how you can customize your calculation views by creating variables and input parameters.

These artifacts will result in dedicated dialog boxes that will appear when the view is executed, requesting the end user to pass parameters to the query to modify the result set or trigger parameter-based calculations.

These artifacts provide the flexibility to use the same calculation view with parameters entered at runtime, instead of hard-coding such parameters in different separate views during the design. It also helps restrict the set of data that is transferred between SAP HANA and the reporting tools.

You can also compute calculated columns based on a parameter entered by the user at runtime instead of hard-coding it in the calculation view itself.



Note:

For course participants familiar with SAP BusinessObjects terminology, a variable or an input parameter is often referred to as a *prompt* in Business Object tools.

Business Example

You want to build a calculation view to report sales by product, but only for the country (or region) the user specifies when executing the view.

Additionally, you would like to create in your calculation view additional calculated values based on user-defined parameters.



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Implement variables
- Define input parameters
- Define Value Help Views
- Map Variables and Input Parameters

Variables

You define a **variable** in a calculation view when you want to **pass dynamic values** for **filtering on attribute columns**. They provide an excellent way of filtering data and there are many options that make implementing variables easy and flexible.

Upon execution of the Calculation View, variables are passed inside the SQL query using a **WHERE clause**, which is easily understood by most applications that call calculation views. For example:

```
SELECT <columns>
FROM <calculation_view_name>
WHERE ((COUNTRY = 'JP'))
GROUP BY <group_by columns>;
```

 The following types of Variables are supported:

Type	Description
Single Value	Use this to filter data on a single value of the column
Interval	Use this where you want the user to specify an interval between two values of the column.
Range	Use this when you want the end user to use operators such as "Greater Than" or "Less Than".

Range Variable Example:

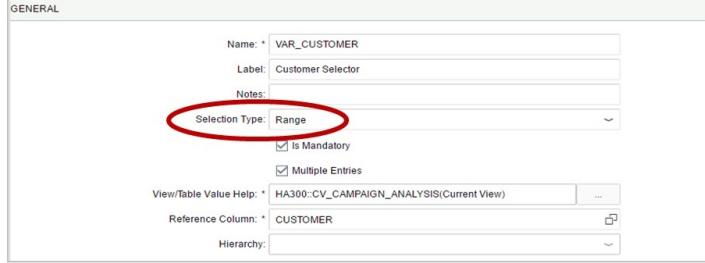


 Figure 77: Variables Types

Creating Variables

You define variables in the *Semantics* node of a Calculation View, in the *Parameters* tab.



Figure 78: Creating Variables (1)

A variable definition includes:

- **View/Table for Value Help and Attribute:** These settings define which view/table and which attribute from this view/table is used as a reference to provide a list of values at runtime
- **Selection Type:** Whether selections should be based on intervals, ranges or single values.
- **Multiple Entries:** Whether multiple occurrences of the selection type are allowed.
- You can also define whether specifying the variable at runtime is **Mandatory** and/or if it should have a **Default Value**.
- You define which **attribute(s)** of the current view the variable should be applied to.

The *View/Table Value Help* setting is used to define in which table/view SAP HANA will fetch the data to show in the prompt. The *Reference Column* defines which (unique) column in this table/view will provide the possible values for the variable.

By default, the *View/Table Value Help* setting refers to the current calculation view. This means that the values of the specified attribute column, for example *CUSTOMER* (customer ID), will be scanned in order to propose the various values in the prompt.

A different approach is to define a dedicated source table/view for the variable. For example, you could define the master data table *CUSTOMERS* as the Value Help source for the variable, and -of course- choose the column from this master data table that matches the column of your calculation view you want to filter on: the Customer ID in our example.

Once the Value Help table and one column are chosen, you assign the *APPLY FILTER* attribute. That is, you identify which attribute column will be filtered at runtime, based on the variable values chosen by the user.



Note:

When the Value Help source table/view is the calculation view itself, the reference column you define is automatically assigned to the *APPLY FILTER* section as a filter attribute. But you can define another attribute if needed, provided that it is consistent with the reference column.

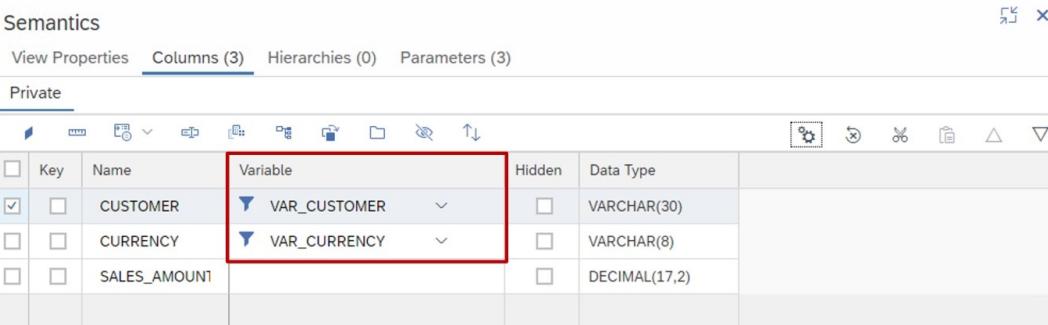
You will learn more about Value Help later on in this lesson.

The behavior of variables at runtime depends on whether an entry is required for the variable or not. These scenarios can be defined as follows:

- If a variable is defined as mandatory, the user needs to provide the values, ranges, or intervals at runtime.
- For non-mandatory variables, if nothing is specified at runtime, all the data for the corresponding attributes is returned by the view without filtering.

Creating Variables (2)

 Figure 79: Creating Variables (II)

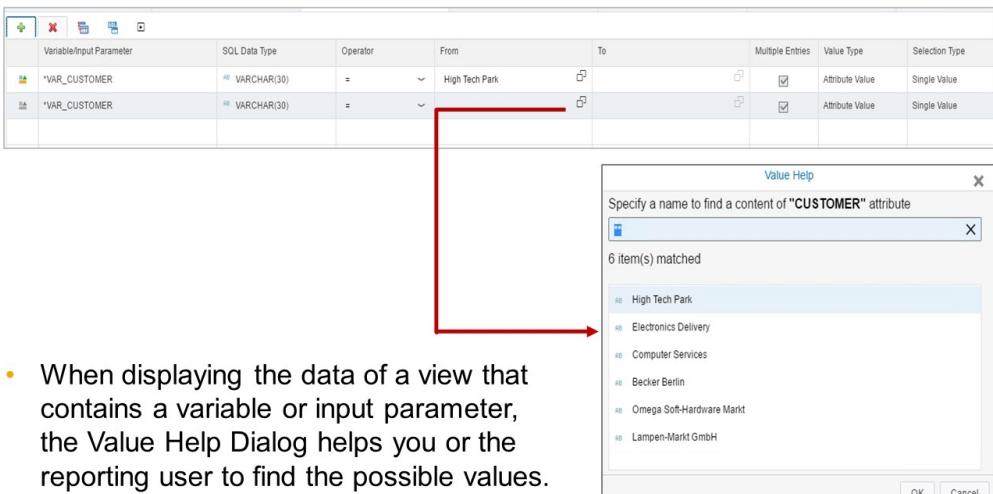


Key	Name	Variable	Hidden	Data Type
<input checked="" type="checkbox"/>	CUSTOMER	VAR_CUSTOMER	<input type="checkbox"/>	VARCHAR(30)
<input type="checkbox"/>	CURRENCY	VAR_CURRENCY	<input type="checkbox"/>	VARCHAR(8)
<input type="checkbox"/>	SALES_AMOUNT		<input type="checkbox"/>	DECIMAL(17,2)

- In the semantics of a view, you can see, and also define, which variable is assigned to which attribute.
- Note that one variable can be assigned to multiple attributes.

Creating Variables (3)

 Figure 80: Creating Variables (III)



Variable/Input Parameter	SQL Data Type	Operator	From	To	Multiple Entries	Value Type	Selection Type
*VAR_CUSTOMER	VARCHAR(30)	=	High Tech Park		<input checked="" type="checkbox"/>	Attribute Value	Single Value
*VAR_CUSTOMER	VARCHAR(30)	=			<input checked="" type="checkbox"/>	Attribute Value	Single Value

Value Help
Specify a name to find a content of "CUSTOMER" attribute
6 item(s) matched

- When displaying the data of a view that contains a variable or input parameter, the Value Help Dialog helps you or the reporting user to find the possible values.

More than one value can be chosen for a variable when you select the *Multiple Entries* checkbox.



Note:

In the data preview, *From* and *To* are displayed in the *Variable Values* dialog even when a variable has not been defined as *range*.

Value Help for Variables

When a dialog box appears to the user they must make a selection. However, rather than an empty field appearing and the user having to guess valid values or figure out the format for a value (for example, is the country code UK or GB?), we can have a dialog box populated with the run-time values available. To do this, you make a selection in the setting *View/Table for value help*. The default entry is the calculation view where the variable is being created. This means that you present **all possible values** from the column that is assigned to the *Attribute* setting in the variable definition.

While this might seem like a great idea, remember that the list may be huge and would be difficult for the user to navigate. Imagine presenting a list of every employee in a very large organization? If the list should be restricted to offer limited values (such as employees in your line of business or country) then you should reference an external calculation view or table that exposes a restricted list. It is also good practice, from a performance perspective, to refer to an external calculation view or table. This is because it means you are burdening the main calculation view with the task of providing value help for unfiltered columns.

Value Help Based on Hierarchies

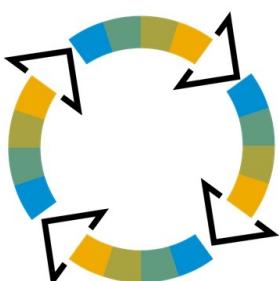
When creating a variable on an attribute that is associated with one or several hierarchies, you can specify one of the hierarchies in the variable definition. With this option, the user can navigate the hierarchy, rather than a flat list, to select the values in the value help. This makes navigation much easier when there are a lot of values; imagine being able to first select your country, then your department, before the list of employees appears? You can use either parent-child or level hierarchies.

The following are some of the basic rules that apply when implementing value help based on hierarchies:

- If you refer to a parent-child hierarchy, the variable attribute column must be defined as the **parent** attribute and not the child.
- If you refer to a level hierarchy, the variable attribute column must be defined at the **leaf level**; that is, the bottom level.

Input Parameters

You can use another type of artifact to specify values to pass to the calculation view at runtime: an **Input Parameter**. Compared with a variable, which is only able to filter a data set, an input parameter can fulfill additional requirements.



- The purpose of Variables is to filter a data set.
- You might want to take input from the user and process it, for example execute a calculation based on the user input/selection.
- Input Parameters make this possible.

Figure 81: Input Parameter Use Cases

You can use input parameters to define the internal parameterization of the view. Input parameters defined in a Calculation View can be used in different places, in particular Calculated Columns, Restricted Columns and Filter Expressions. Here are a few examples:

- Calculated Column

You want to execute an ad-hoc calculation of the forecast sales for next year, based on the current year and an overall sales volume increase rate (example: 1.05) entered by the user at runtime.

- Restricted Column

You want a calculation view to return the sales for a year entered by the user at runtime (example: 2019) and the previous year, using two restricted columns.

- Filter

You want to filter the data set for sales orders that have a total amount greater than \$ 10.000 (threshold entered at runtime).

You want to filter sales data for products that have a name containing the string 'GTR' (entered at runtime).

You want to control specifically where a filter expression based on a user entry is applied inside the calculation scenario.

Input Parameters can also be used when you have scalar or table functions in your Calculation View. The parameters of these functions can be fed with values that you enter at runtime, when querying your calculation view. This requires that you map the parameters, which will be discussed later on in this lesson.

Input Parameter Types



The following types of Input Parameters are supported:

Type	Use Cases
Direct: Currency	For currency conversion, when you want the end user to specify a source or target currency.
Direct: Date	To retrieve data based on a date entered by the end user (or chosen in a calendar type input box).
Direct: Unit of Measure	To retrieve data based on a unit of measure chosen by the end user
Static List	To provide the end user with a predefined list of values in which he/she chooses one or several items.
Column	To provide the end user with a list of values from a column of the information model
Derived From Table	When you want the end user to have a set list of values from a table (not necessarily included in the view)
Derived From Procedure	When you want the parameter value to be passed to the information model based on the scalar output of a stored procedure
Direct (without semantic type)	When none of the above applies and/or when you want the user to enter a parameter without choosing it from a predefined list.



Figure 82: Input Parameter Types

The figure, Input Parameter Types, shows the different types of Input Parameters that can be defined.

The *Direct* parameter type can be combined with a semantic type such as *Date*, *Currency*, or *Unit of measure*. This means that the value help will be based on these types of values. For

example, if you specify *Date*, then a pop-up calendar will appear for the user prompt. If you specify *Currency*, then a list of valid currencies will be presented in the value help. This allows us to provide flexible input for the user but also allows us to control the type of values that are allowed.

Currency and Unit of Measure Semantic Types

For the *Currency* and *Unit of measure* semantic types, the list of proposed values will be created based on the corresponding reference tables in SAP HANA. This setup requires that the default schema assigned to the view contains the reference tables.

Input parameters support multiple values, which means that, at runtime, the end user has the possibility to provide several values to the parameter. Some examples of use cases include the following:

- Applying filters of the types *List of values* and *Not in List*
- Expression of calculated columns and expression of filters in projection nodes, provided that the expression requires a multivalue input.

When you define an input parameter of the type *Derived from Procedure/Scalar Function*, it is possible to map parameters to the input of the scalar function or procedure.



Note:

Input parameters of the types *Derived from table* and *Derived from Procedure/Scalar Function* do not generate a prompt for the end user (they pass the parameter values directly), except if you select the *Input Enabled* option. In this case, the values returned by the table, procedure, or scalar function, can be modified by the end user.

Upon execution of the Calculation View, Input Parameters are passed inside the SQL query using a **PLACEHOLDER clause**. For example:

```
SELECT <columns>
FROM <calculation_view_name>
  (placeholder."$$IP_YEAR$$">'2019')
GROUP BY <group_by_columns>;
```

Not all applications are able to pass values to the PLACEHOLDER.



Note:

Once you have defined an input parameter, you must figure out how to use it in an expression; otherwise it is ignored.

Creating Input Parameters

As discussed already, unlike a variable, an input parameter can be used in a conditional expression. For example, we can use an input parameter to determine which measure should be displayed in a particular column.

To illustrate this, we create a calculated column called *AMOUNT* that can be filled with either the gross amount or the net amount, depending on the value that the user chooses when querying the view.

In our example, we have chosen to use an input parameter of the type *Static List*. This means that we predefine the allowed value that can be chosen by the user in a list. This is fine for

short lists, but when the list becomes large it becomes cumbersome to manage, as you would have to edit the calculation view and rebuild it each time. Of course you could choose the type *Direct*, which would mean the user could input anything. But that would mean, apart from the user not having any guidance, the user could also mistype the value, or enter the value in the wrong format (perhaps adding leading zeros when they were not required).

A good solution would be to define the input parameter with the type *Column* and then, in the *View / Table Value Help*, enter the name of a table or view where the allowed entries are presented. This also means that this list can be used by multiple input parameters and encourages central maintenance of the consistent, allowed values list.

Input Parameters

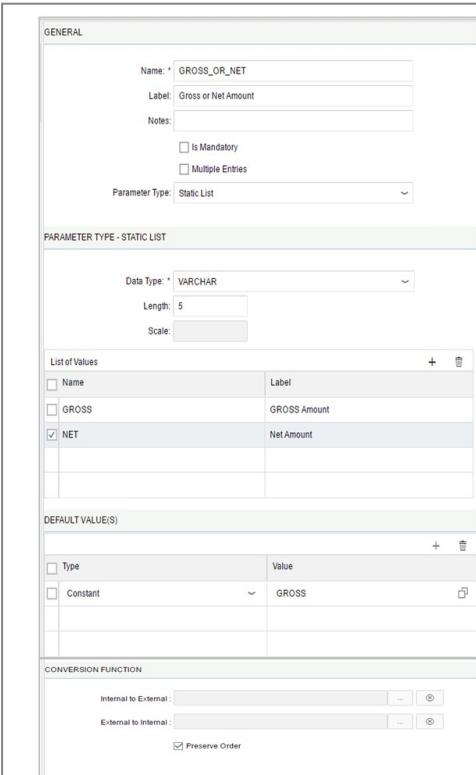


Figure 83: Creating Input Parameters

- If we want the end user to decide whether Gross or Net amount should be shown in a view, the first step is to create an input parameter that will be used in a calculation.
- The Input Parameter can be of any suitable type, for example a Static List type.
- In this example, the user will be able to choose either "Gross Amount" or "Net Amount".
- Default value GROSS will be assigned to the input parameter if the user does not specify anything.

An input parameter used within a formula does not necessarily have to be of the type *Static List*. For example, it can also be a *Direct* numeric value used in multiplication or any other calculation type.

Calling an Input Parameter in a Calculation



GENERAL

Name: * AMOUNT
Label: Gross or Net Amount depending on the Input Parameter
Notes:
Data Type: * DECIMAL
Length: 13
Scale: 2

SEMANTICS

Column Type: Measure
Aggregation Type:
Hidden Enable client side aggregation
Drill Down:

EXPRESSION

Column Engine Expression Editor >
if(\$\$GROSS_OR_NET\$\$='GROSS','GROSS_AMOUNT','NET_AMOUNT')

- The second step is to use the Input Parameter in a Calculated column.
- This is done by calling it within single quotes and double dollar signs.
- In this example, the input parameter is used in the condition of an IF expression

Expression Editor (AMOUNT)

Language: Column Engine ✓ Validate Syntax

if(''GROSS_OR_NET''='GROSS','GROSS_AMOUNT','NET_AMOUNT')

Elements Functions Operators

Columns	if	+
CUSTOMER	Conversion Functions	-
CURRENCY	String Functions	*
SALES_AMOUNT	Mathematical Functions	**
SALE_DATE	Date Functions	()
Calculated Columns	Spatial Functions	=
Restricted Columns	• ST_Difference()	>
Parameters	• ST_SymDifference()	<
GROSS_OR_NET	• Spatial Predicates	>=
	• Misc Functions	<=
		isNull
		not
		and
		or
		in
		match

if(''GROSS_OR_NET''='GROSS','GROSS_AMOUNT','NET_AMOUNT')

Figure 84: Calling an Input Parameter in a Calculation

In the example in the figure, Calling an Input Parameter in a Calculation, if the user selects GROSS, the calculated column (of type Measure) will display the GROSS_AMOUNT measure in the AMOUNT column. Any other selection will result in NET_AMOUNT being displayed.

Input Parameter Using Dates



GENERAL

Name: * <input type="text" value="INP_CONV_DATE"/>
Label: <input type="text" value="INP_CONV_DATE"/>
Notes: <input type="text"/>
<input checked="" type="checkbox"/> Is Mandatory
<input type="checkbox"/> Multiple Entries
Parameter Type: <input type="text" value="Static List"/>

PARAMETER TYPE - STATIC LIST

Data Type: * <input style="outline: 2px solid red;" type="text" value="DATE"/>
Length: <input type="text"/>
Scale: <input type="text"/>

List of Values

Name	Label
20121231	31/12/2012
20131231	31/12/2013

DEFAULT VALUE(S)

Type	Value
Constant	<input type="button" value="□"/>



Figure 85: Input Parameter Using Dates

- An Input Parameter type of type "Direct" with a semantic type "Date" can be useful when you want to create calculations based on a date specified by the reporting user.

- You can create a date range by creating a pair of input parameters (for example, "Date From" and "Date To")

- Note that the Data Type must be set to "DATE".



Using a Calendar Dialog for Date Input Parameters

Variable/Input Parameter

Variable/Input Parameter: <input type="text" value="INP_CONV_DATE"/>	SQL Data Type: <input type="text" value="DATE"/>	Operator: <input type="text" value="#"/>	From: <input type="text"/>	To: <input type="text"/>	Multiple Entries: <input type="checkbox"/>	Value Type: <input type="text" value="StaticList"/>	Selection Type: <input type="text" value="Single"/>
--	--	--	----------------------------	--------------------------	--	---	---

Value Help

July 2017

Sun	Mon	Tue	Wed	Thu	Fri	Sat
26	27	28	29	30	1	
27	2	3	4	5	6	7
28	9	10	11	12	13	14
29	16	17	18	19	20	21
30	23	24	25	26	27	28
31	30	31	1	2	3	4

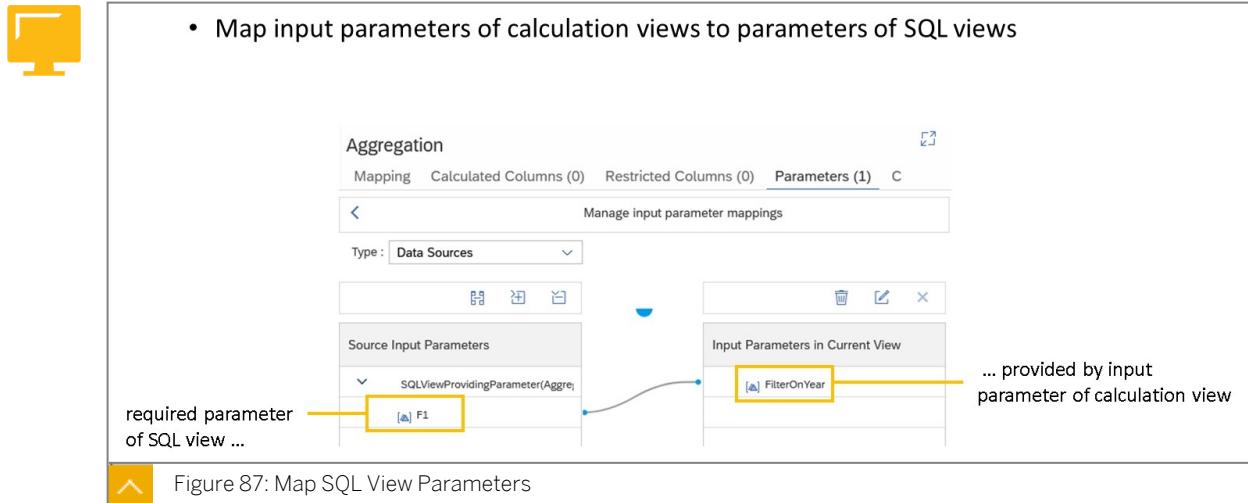


Figure 86: Using a Calendar Dialog for Date Input Parameters

In the figure, Using a Calendar Dialog for Date Input Parameters, the user is asked for a single value. Dates can also be selected as ranges.

Mapping Input Parameters to SQL View Parameters

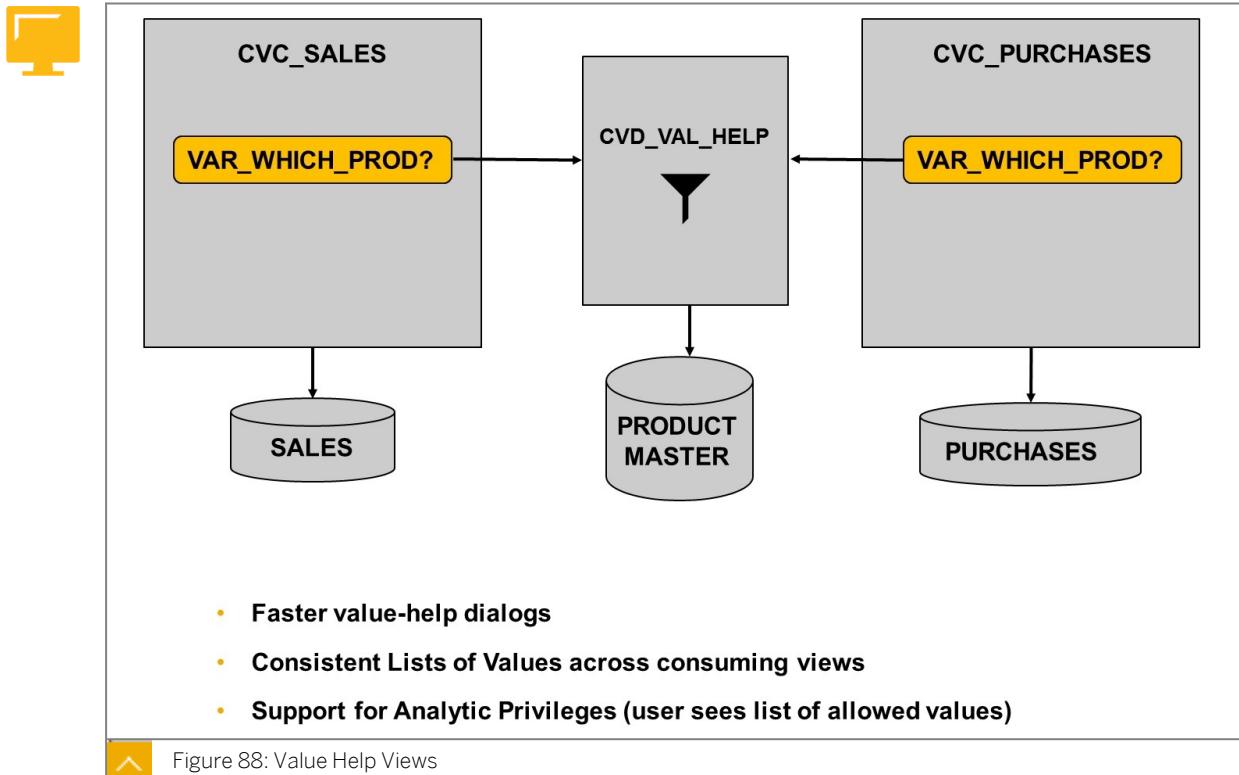
If a data source for your calculation view is an SQL view and the SQL view requires parameters, you can provide the values from input parameters of the consuming calculation view.



This technique is often used to push down the filters down to the SQL view in order to reduce the workload of the calculation view.

Value Help Views

When you define a variable you must choose an attribute that the variable is based on. But the user prompt provides all possible values of the chosen attribute that come from the data source. In some cases this can produce a very long list of values, for example, a list of materials. As well as being overwhelming, you also might not want some values to appear in the list. So you need to restrict the list of available attributes that appear in the prompt.



To restrict the values that are offered to a user in a prompt, you define a **value help view**.

A value help view is usually a dimension calculation view that includes filters. The filters can either be fixed in the view (this would mean the same values are always provided) or they can be provided from an assigned analytic privilege (so each user would have a personal list of values).

The value help view is then referenced in a calculation view that defines the variable.

Mapping of Variables and Input Parameters

Pushing Down Input Parameters and Variables to Lower Level Calculation Views

In many cases, calculation views use other calculation views as data sources. This is not necessarily confined to two levels; we can go on and layer the calculation views to create a **stacked model**. When you execute a calculation view in which variables or input parameters are defined, it is possible to pass their values (entered by the end user at runtime) to the lower level calculation views. In fact, the input parameters and variables at the lower levels are usually ignored unless you define input parameters and variables at the top level and map them to the input parameters and variables in the lower levels. This is called **parameter mapping** and is an important feature of SAP HANA calculation view modeling.

The screenshot shows the SAP BW Semantics view with the 'Parameters (1)' tab selected. A red arrow points from the text below to the left pane, which lists 'Source Input Parameters' for 'HA300::CVD_LIMITED_PRODUCTS'. Another red arrow points from the text below to the right pane, which lists 'Input Parameters in Current View' for '(A)IP_PROD_GROUP'. A note at the bottom suggests using the 'auto map' button to generate IP with the same name.

Input Parameters defined in source Calculation View appear on left side

Input Parameters defined in consuming Calculation View appear on right side *

Use 'auto map' button to generate IP with same name to avoid having to create each one manually

Figure 89: Mapping Parameters in Calculation Views

To enable parameter mapping, you must use the *Input Parameter/Variables Mapping* feature. You can find this feature in the *Parameters* tab in the calculation view.

When you open the mapping pane you must first select the type of mapping you want to work with, using the drop-down selector.

Parameter Mapping Types

There are four types of parameter mapping and you choose the type from the *Manage Mapping* pane.

Value	Description
Data Sources	Map input parameters of the underlying data source to input parameters of the calculation view
Procedures/Scalar Functions For input parameters	If you are using input parameters of type procedure/scalar function, and if you want to map the input parameters defined in the procedure or scalar function to the input parameters of the calculation view
Views for value help for variables/input parameters	If you are using input parameters or variables, which refer to external views for value help references and if you want to map input parameters or variables of external views with the input parameters or variables of the calculation view.
Views for value help for attributes	If you are creating a calculation view, and for the attributes in the underlying data sources of this calculation view, if you have defined a value help view or a table that provides values to filter the attribute at runtime.

Figure 90: Parameter Mapping Types

Once you make your type selection, you will then see, on the left side, the input parameters and variables that are defined in the calculation views from all lower layers in the stack, which are related to the mapping type you selected. On the right side, you will see the input parameters and variables that are defined in the current calculation view (the one you are editing).



Note:

You can only map Variables to Variables and Input Parameters to Input Parameters. Cross-Mapping (such as an Input Parameter to a Variable) is not possible.

You simply drag a line between the left and right side to map them. There is also an auto-map feature which means that if the names are the same, the mapping is done with a single click. The auto-map feature generates the input parameters or variables for the current view with the same name as the source variables and also maps them. This means that you don't have to manually create the input parameters or variables in the current view.

Pushing filters down to the source views using parameter mapping is a common scenario. To enable this, choose the type *Data Sources* from the drop-down list in the *Manage Mapping* dialog.

Mapping input parameters of the current view to the input parameters of the underlying data sources moves the filters down to the underlying data sources during run-time. This is a great way to improve performance.

Another common scenario is when you want to push parameters down from the main calculation view to a calculated column in a lower view to support a calculation. Again, this would be the type *Data Sources*.

Note that In the type of mapping *Data Source*, you only map input parameters to input parameters. In other words, a Variable defined in an underlying Calculation View cannot be mapped to a Variable defined in the current Calculation View. However, it is possible to access these variables from the *Extract Semantics* feature and copy them to the current view. To do that, you right-click the data source in the calculation scenario and choose *Extract Semantics*. Then choose the *Variables* tab and select the ones you want to copy to the semantics of your Calculation View.



Note:

From SAP HANA 2.0 SPS06 onwards, it is possible to map input parameters of the current calculation view to parameters defined in an underlying SQL View.

This allows to better control filters when reporting on top of a SQL View.

Mapping for External Value Help Views

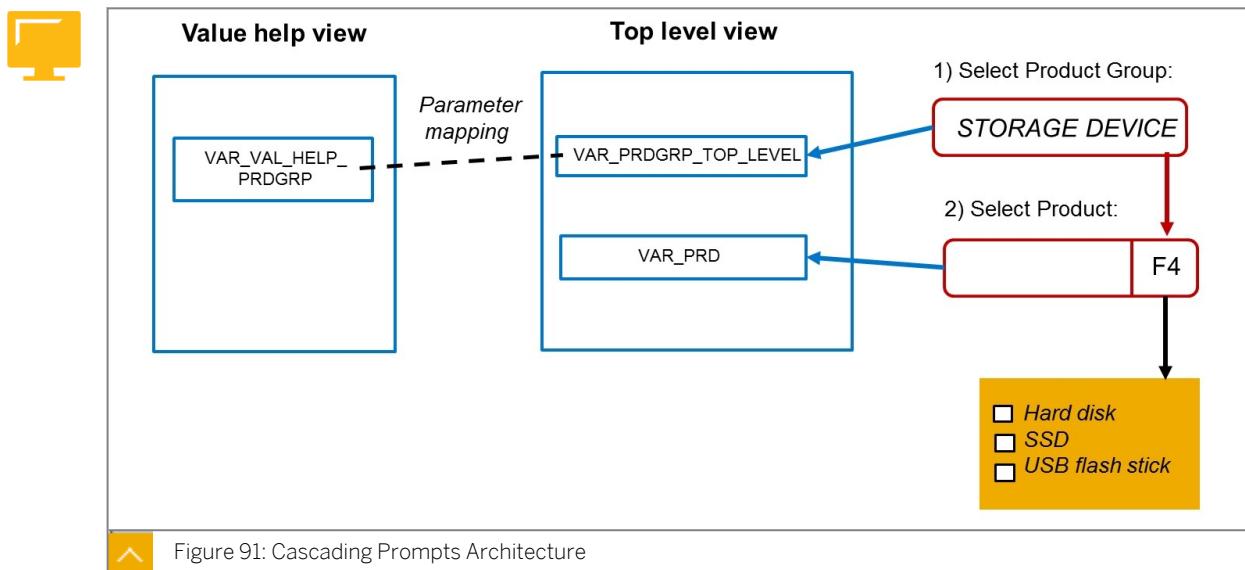
Another important use case for mapping input parameters and variables is to enable dynamic **value help views**.

When you define input parameters and variables, the default data source that generates the **value help** list is taken from the calculation view itself. So, essentially you are getting an unrestricted list of all possible values to choose from. However, you can also redirect the value help to use a list from another table or view. The main reason we do this is to expose a restricted value help list.

This is also good practice for performance because the value help is not competing with the main calculation view for data. For example, you could create a calculation view on a table that contains all possible cities. Here, your calculation view would include a fixed filter expression that restricts the cities to a specific country. This means that the value help list presents only cities of a specific country to the user.

What if you wanted to change the country? You could go back to the calculation view and change the fixed filter expression, but this would be inefficient.

Cascading Prompts Architecture



What we should do is replace the fixed value in the filter expression with a variable based on country. Then, we should map this variable to a variable we define in the main calculation view for the country. This means that when a user is prompted for a country in the main view, the value chosen is passed through the mapping to the value help calculation view, so that the cities are filtered by the country that was chosen. The list of cities is then presented as the value help for the *cities* column. This is also known as **cascading prompts**. Cascading is not restricted to two levels; you can also cascade prompts across multiple levels. For example you could prompt for *Continent*, which then restricts the list of *Countries*, which in turn restricts the list of *Cities*, and so on.

To implement value help parameter mapping, you must select the option *Views for value help for variables/input parameters* from the drop-down list in the *Manage Mapping* dialog.



Note:

An external view based on a hierarchy could also be considered as a value help cascading solution, and might be more visually appealing to the user.



LESSON SUMMARY

You should now be able to:

- Implement variables
- Define input parameters
- Define Value Help Views

- Map Variables and Input Parameters

Implementing Hierarchies

LESSON OVERVIEW

This lesson explains how to implement hierarchies in information models using the SAP Web IDE for SAP HANA.

Business Example

Hierarchies are usually used in business intelligence reporting to display characteristics across aggregated nodes.

For example, you may have business requirement to display customers in a geographical organization with country, state, and city in a hierarchy.



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Define Hierarchies

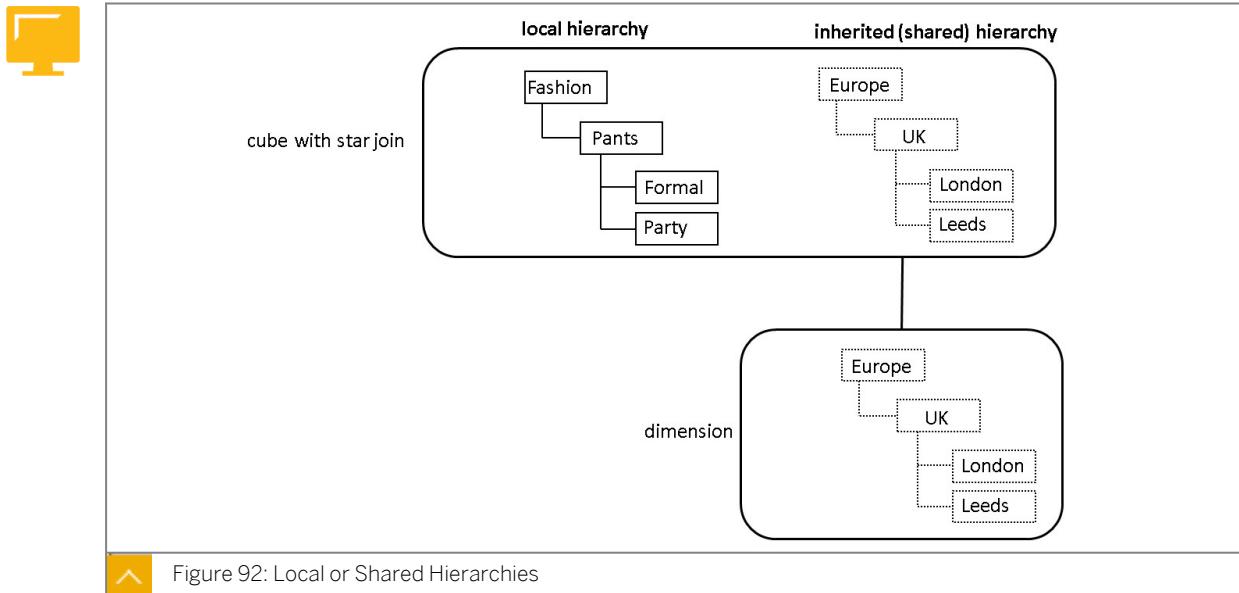
Hierarchies in SAP HANA Modeling

Many of the attributes that are included in your calculation views can be usefully organized using hierarchies. For example, a product hierarchy, team hierarchy, or an organization hierarchy.

A hierarchy can be used to support various modeling requirements including:

- Provide a helpful drill-down sequence in a report
- Calculate measures at various levels of aggregation
- Filter data according to levels of a hierarchy
- Prevent users from displaying data at specific levels of a hierarchy

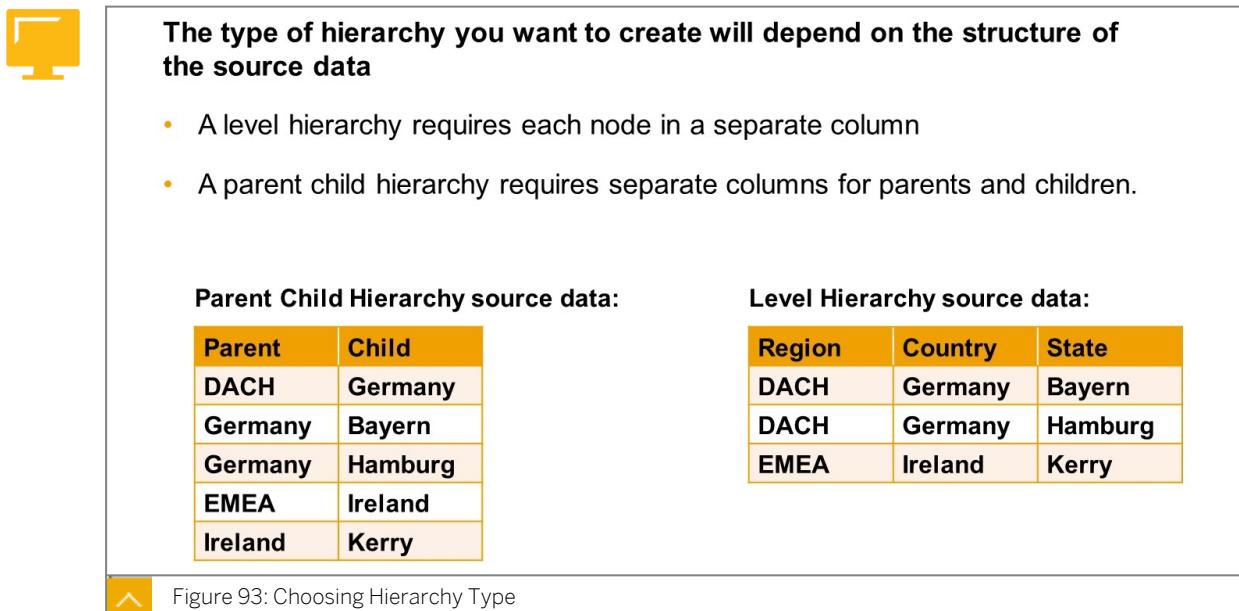
In SAP HANA calculation view modeling you can define one or more hierarchies based on the attributes that are available in the calculation view.



Hierarchies can be defined in any type of calculation view (dimension, cube etc.). However, it is good practice to define common hierarchies in dimension calculation views so they can be re-used in other calculations views. A hierarchy that is inherited from an underlying dimension calculation view is known as a *shared hierarchy*. In addition to the shared hierarchies, you can define additional hierarchies in the top-level calculation view. These hierarchies are known as *local hierarchies*.

In SAP HANA modeling, there are two types of hierarchy:

- Level hierarchy
- Parent-Child hierarchy





Note:

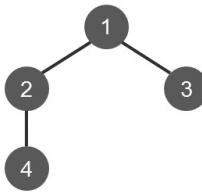
Parent-child hierarchy columns usually contain IDs or key fields instead of plain text.

Hierarchy Comparison



Parent-Child Hierarchy

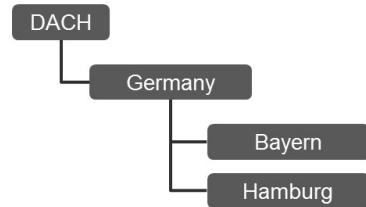
Parent	Child	X	Y	Z
1	2			
1	3			
2	4			



- Distinct fields define the parent-child relation
- Parent and child fields usually have the same data type
- Recursive data structure defines the hierarchy

Level Hierarchy

Region	Country	State	X	Y
DACH	Germany	Bayern		
DACH	Germany	Hamburg		
EMEA	Ireland	Kerry		



- Heterogeneous fields (possibly with different data types) are combined in a hierarchy

Figure 94: Hierarchy Comparison

Level Hierarchies

A level hierarchy is rigid in nature, and the root and child nodes can only be accessed in a defined order.

To implement level hierarchies, use the following procedure:

• Select the source table(s) for the view:

Figure 95: Implement Level Hierarchies (I)

Level Hierarchies (2)

• Select the columns that should be part of the view, including any columns required for the hierarchy:

Figure 96: Implement Level Hierarchies (II)

Level Hierarchies (3)



- In the Semantics node, select the Hierarchies tab and click the '+' button in the Hierarchy pane:

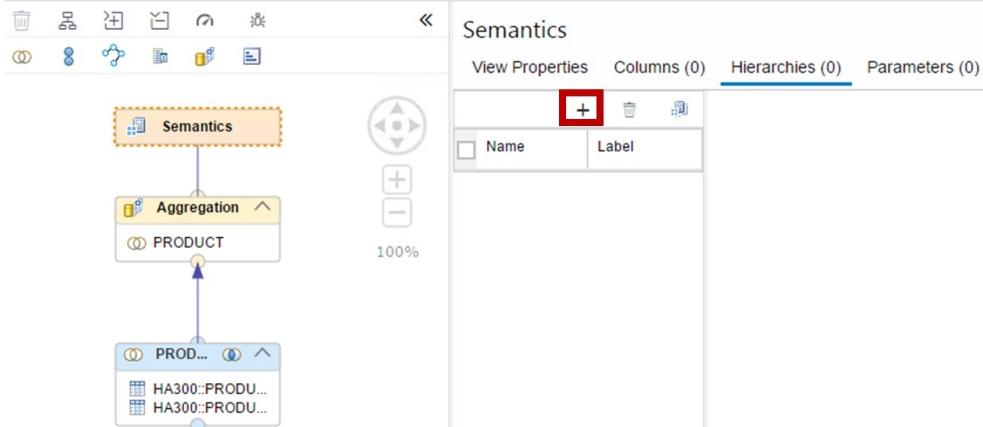
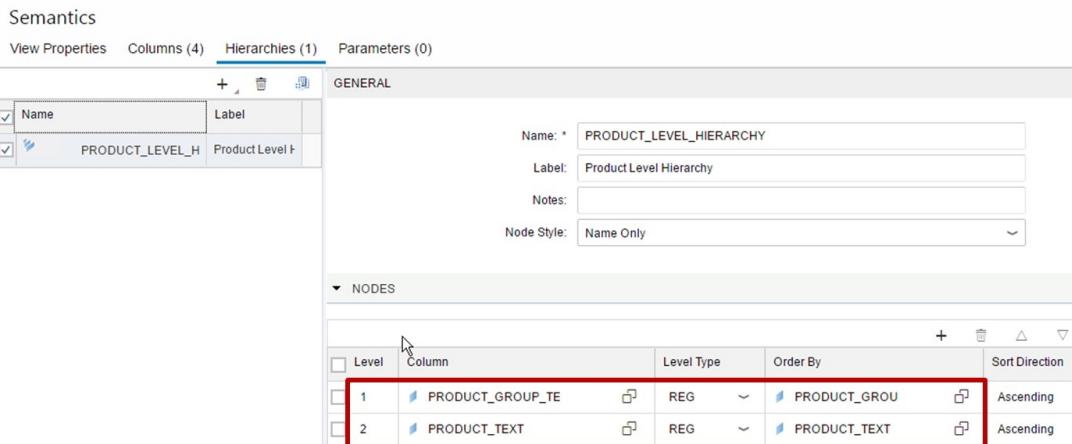


Figure 97: Implement Level Hierarchies (III)

Level Hierarchies (4)



- Add the columns to the hierarchy in the correct level order from top to bottom, with the lowest granularity at the lowest level of the hierarchy.
- Additionally, you can define an ascending or descending sort direction per level.

Figure 98: Implement Level Hierarchies (IV)



Note:

When you preview a calculation view containing hierarchies using the SAP Web IDE for SAP HANA, you will not be able to see the hierarchy in the same way that it is displayed when you use a reporting tool supporting hierarchies.

Node Styles

Node styles are used to define the output format of a node ID.

Using a fiscal hierarchy example, the following table demonstrates the different node styles:



Table 11: Node Styles

Level Style	Output	Example
Level Name	Level and node name	MONTH.JAN
Name Only	Node name only	JAN
Name Path	Node name and its ancestors	FISCAL_2015.QUARTER_1.JAN

Level Types

A level type specifies the semantics for the level attributes. For example, the level type *TIMEMONTHS* indicates that the attributes are months such as January, February, or March.

The level type *REGULAR* indicates that the level does not require any special formatting.

Hierarchy Member Order

Using the *Order By* drop-down list, an attribute can be selected for ordering the hierarchy members in the order specified in the *Sort Direction* column.

Orphan Nodes

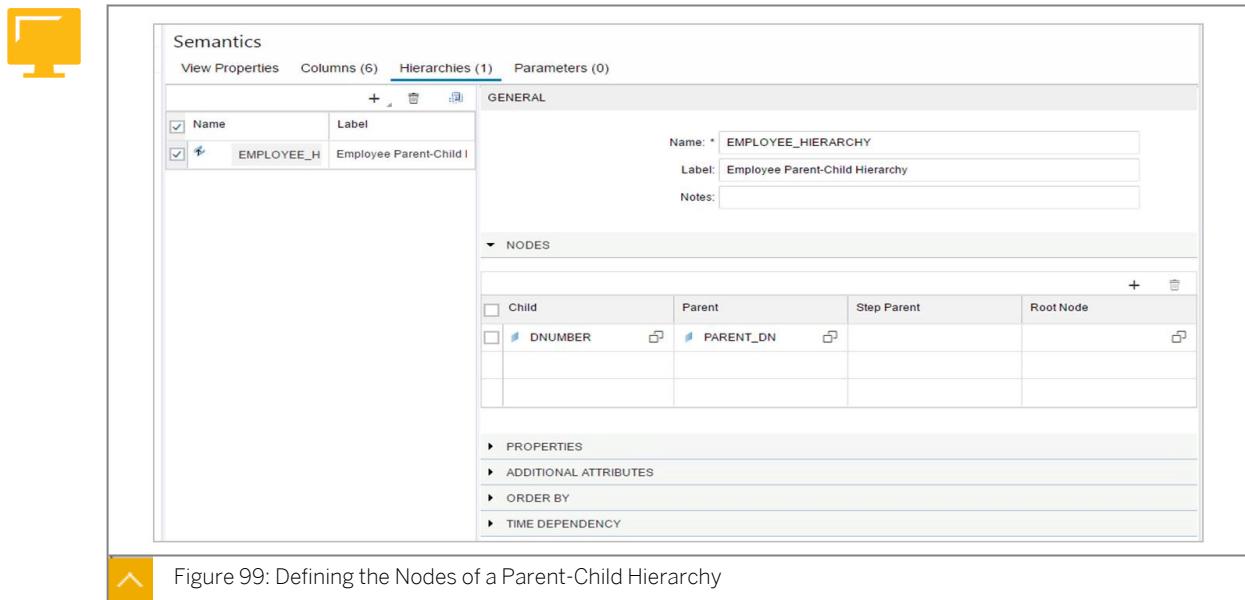
An orphan node in a hierarchy is a member that has no parent member.

Level hierarchies offer four different ways to handle orphan nodes. They are as follows:

- Root Nodes
Any orphan node will be defined as a hierarchy root node.
- Error
When encountering an orphan node, the view will throw an error.
- Ignore
Orphan nodes will be ignored.
- Step Parent
Orphan nodes are assigned to a step parent you specify.

Parent-Child Hierarchies

When creating a parent-child hierarchy, the first step is to define the nodes that make up the hierarchy.



The screenshot shows the SAP Semantics interface for defining hierarchies. In the top navigation bar, 'Hierarchies (1)' is selected. The main area is titled 'GENERAL' and contains fields for 'Name' (EMPLOYEE_HIERARCHY) and 'Label' (Employee Parent-Child Hierarchy). Below this is a 'NODES' section with columns for 'Child' (DNUMBER) and 'Parent' (PARENT_DN). At the bottom, there are sections for 'PROPERTIES', 'ADDITIONAL ATTRIBUTES', 'ORDER BY', and 'TIME DEPENDENCY'.

Figure 99: Defining the Nodes of a Parent-Child Hierarchy

The *Child* column contains the attribute used as the child within the hierarchy, whereas the *Parent* column contains the attribute used for its parent.

You can define multiple parent-child pairs to support the compound node IDs. For example:

- *CostCenter* → *ParentCostCenter*
- *ControllingArea* → *ParentControllingArea*

The preceding list of parent-child pairs constitutes a compound parent-child definition to uniquely identify cost centers.

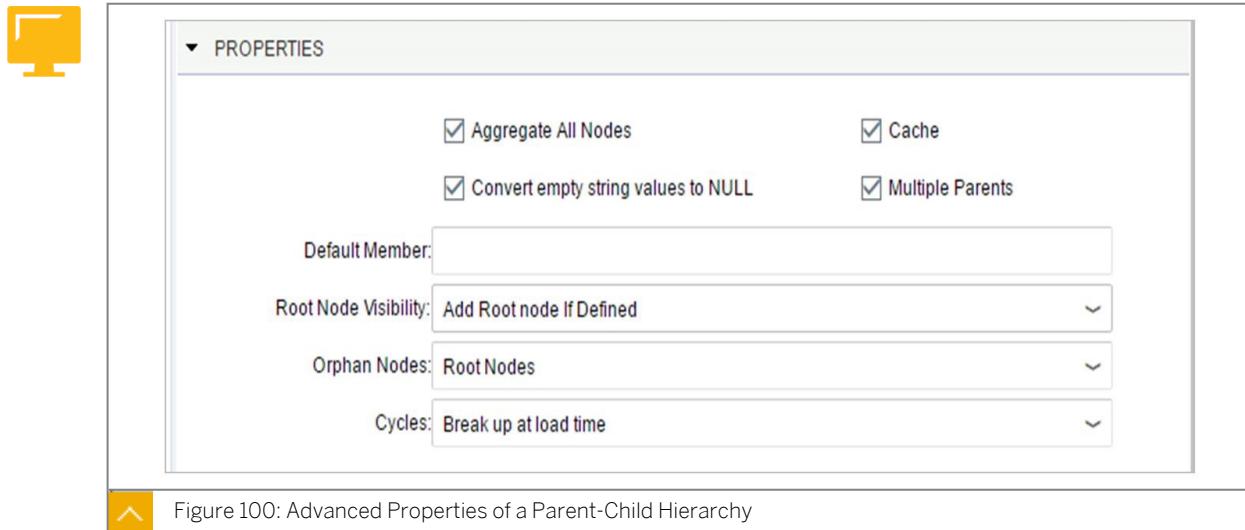


Caution:

Multiple parents and compound parent-child definitions are not supported by MDX.

Advanced Properties of a Parent-Child Hierarchy

Additional attributes can also be added to the hierarchy, making it easier to report on.



The screenshot shows the 'PROPERTIES' section of the SAP Semantics interface. It includes checkboxes for 'Aggregate All Nodes' (checked), 'Cache' (checked), 'Convert empty string values to NULL' (checked), and 'Multiple Parents' (checked). There are also dropdowns for 'Default Member', 'Root Node Visibility' (set to 'Add Root node If Defined'), 'Orphan Nodes' (set to 'Root Nodes'), and 'Cycles' (set to 'Break up at load time').

Figure 100: Advanced Properties of a Parent-Child Hierarchy

Aggregate All Nodes

The *Aggregate All Nodes* property defines whether the values of intermediate nodes of the hierarchy should be aggregated to the total value of the hierarchy's root node. If you are sure that there is no data posted on aggregate nodes, you should set the option to *False*. The engine then executes the hierarchy faster.



Note:

The value of the *Aggregate All Nodes* property is interpreted only by the SAP HANA MDX engine.

Default Member

The *Default Member* value helps identify the default member of the hierarchy. If you do not provide any value, all members of the hierarchy are default members.

Orphan Nodes

In a parent-child hierarchy, you might encounter orphan nodes without a parent. The *Orphan Nodes* property defines how these should be handled.



Note:

If you choose to assign an orphan node to a step parent, the following rules apply:

- The step parent node must be already defined in the hierarchy at the ROOT level.
- The step parent ID must be entered according to the node style defined in the hierarchy.

Root Node Visibility

The *Root Node Visibility* property is used to define whether an additional root node needs to be added to the hierarchy.

Cycles

Cycles are typically not desirable in a hierarchy.

In such cases, the *Cycles* property is used to define how these should be broken when encountered, or whether an error should be thrown.

Time-Dependent Hierarchies

Time dependency is supported for more complex hierarchy data, such as human resources applications with their organizations, or material management systems with BOMs where information is reliant on time.



Caution:

Defining a time dependency is only possible in calculation views, and for parent-child hierarchies.

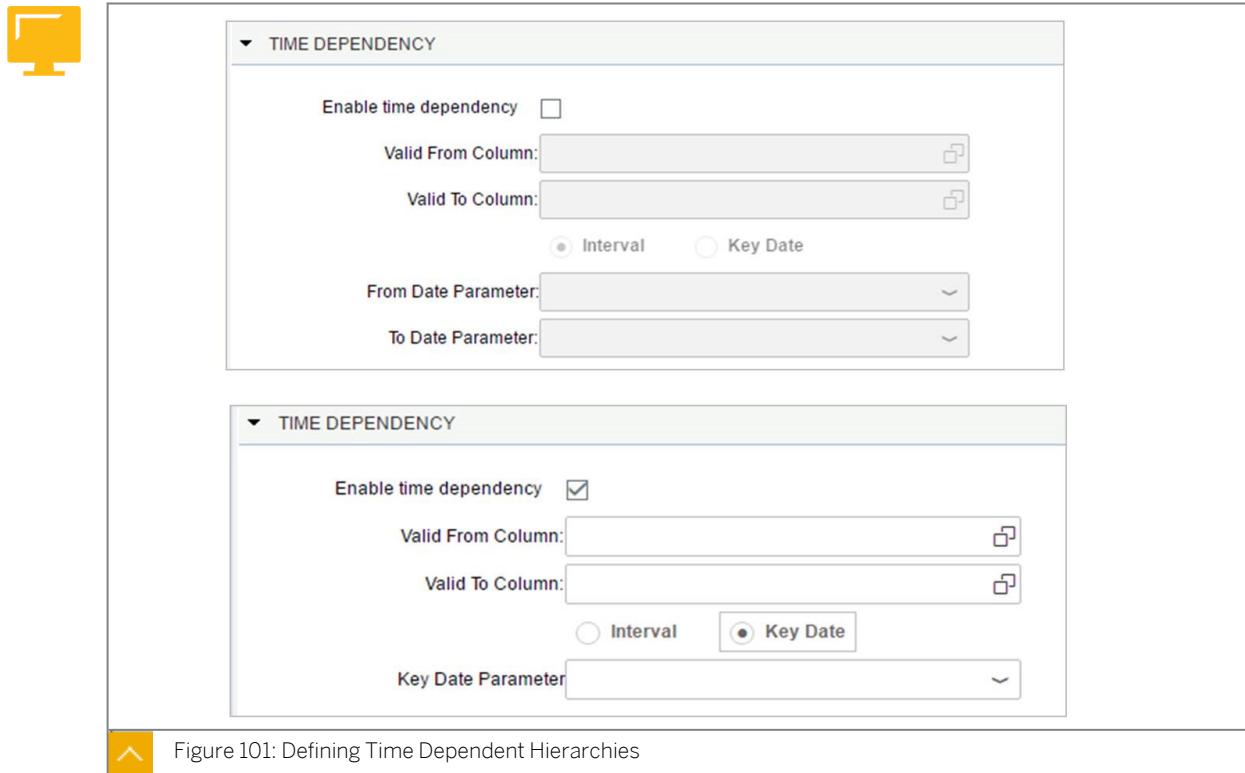


Figure 101: Defining Time Dependent Hierarchies

Enabling time dependency supports hierarchies based on changing elements valid for specific time periods. This allows displaying different versions of a hierarchy.

Your source data needs to contain definition columns consisting of a *Valid From* and a *Valid To* column.

Determining the Validity Period

When a hierarchy needs to show elements from an interval, you have to define two input parameters; a *From Parameter*, and a *To Parameter*. If the hierarchy needs to show elements valid on a specific date, you need one input parameter defined as the *Key Date*.

Drill Down Enablement

By default, MDX only shows key fields, which are governed by an output field property of the dimension calculation view.

If the *Drill Down Enablement* property is left as default for a non-key field, this field will not show up for reporting in an MDX client such as Microsoft Excel.



- By setting Drill Down Enablement to 'Drill Down with flat hierarchy (MDX)', you can make sure that the column can be used for reporting using MDX, even though it is not a key field.

Semantics				
		View Properties	Columns (9)	Hierarchies (0)
Type	Key	Name		Drill Down Enablement
<input type="checkbox"/>	<input checked="" type="checkbox"/>	CLIENT		Drill Down
<input type="checkbox"/>	<input checked="" type="checkbox"/>	NODE_KEY		Drill Down
<input checked="" type="checkbox"/>	<input type="checkbox"/>	BP_COMPANY_NAME		Drill Down
<input type="checkbox"/>	<input type="checkbox"/>	BP_ID		Drill Down
<input type="checkbox"/>	<input type="checkbox"/>	FIRST_NAME		Drill Down
<input type="checkbox"/>	<input type="checkbox"/>	LANGUAGE		Drill Down with flat Hierarchy (MDX)

Figure 102: Drill Down Enablement in MDX

When *Drill Down Enablement* is set to *Drill Down with flat hierarchy (MDX)* the attribute is enabled for drill down and an additional flat hierarchy is generated.

In the generated flat hierarchy, all the distinct attribute values make up the first and only level of the hierarchy.

Enabling Hierarchies for SQL Access

A hierarchy defined in a dimension calculation view is available (as a shared hierarchy) in any calculation view of the type *Cube with Star Join* that references the dimension view in its star join. SAP HANA enables you to include a specific column for this hierarchy, which you can query with SQL. This column will display the different nodes of the hierarchy, which will enable filtering or aggregation in SQL queries executed against the *Cube with Star Join* view.



Note:

This column is only available via SQL. It is not exposed to the graphical information models that consume the view.

For a given *Cube with Star Join* calculation view, the following two options enable SQL access to the shared hierarchies defined in the underlying dimension views:



- Option 1: Enable all shared hierarchies at once.

You can enable SQL access to all of the shared hierarchies defined in the various dimension calculation views.

In the *View Properties* tab of the Semantics node, select the *Enable Hierarchies for SQL* access checkbox.

- Option 2: Select which shared hierarchies you want to enable.

You can specify which of the shared hierarchies you want to enable for SQL access.

In the *Hierarchies* tab, select a shared hierarchy and in the *SQL Access* area, select the *Enable SQL access* checkbox.

You will learn how to consume the hierarchy columns with SQL later, in the *SQL Script and Procedures* unit.

Types of Generated Hierarchies

When you define a hierarchy in an SAP HANA Calculation View, whether it is a level or parent-child hierarchy, the hierarchy is materialized upon build/deployment by various tables and/or views in the HDI Container schema (column views or classic SQL views) or in other locations, especially the SAP HANA Analytic Catalog (*BIMC** tables and views in the *_SYS_BI* schema). These objects enable, among others, data preview in the SAP Web IDE for SAP HANA and the consumption of the hierarchy by front-end tools.

A key setting, introduced in SAP HANA 2.0 SPS06, allows you to influence the way SAP HANA translates the defined hierarchy into technical tables and views. This is the **Hierarchy Type** setting, which you can define in the *Semantics* of the Calculation View, in the *View Properties → General* tab.

The setting can take three values:

- **Auto** (default value)

Upon build, SAP HANA generates MDX hierarchies, including the meta-data defining the hierarchy, as well as column views materializing the MDX hierarchy.



Note:

This value corresponds to the previous behavior up to SAP HANA 2.0 SPS05, when the (new) setting did not exist

- **SQL Hierarchy Views**

Upon build, in addition to hierarchy meta-data and MDX hierarchies (column views), classic SQL views are also generated to materialize the SQL hierarchy.

- **No Hierarchy Views**

Upon build, the meta-data of the hierarchies is generated (*_SYS_BI.BIMC** tables) but the hierarchies themselves (detailed list of members, and so on) are NOT generated.

This setting is relevant when the consuming front-end tool itself can generate the set of hierarchy members and their relationships, based on the meta-data defined in the SAP HANA Analytic Catalog.



Caution:

When MDX hierarchies are generated, another setting in the General properties of Calculation Views, *Generate technical MDX hierarchies*, governs the generation of additional Column Views for each and every dimension of a CUBE or DIMENSION Calculation View.

These technical MDX hierarchies are required in some scenarios, such as previewing the hierarchy in the SAP Web IDE for SAP HANA, or querying the hierarchy from Excel with the SAP HANA MDX provider.

The table below summarizes which objects and references are generated based on the chosen option.

Table 12: Hierarchy Type

	Auto	SQL Hierarchy Views	No Hierarchy View
Hierarchy Meta-data (records in the <i>BIMC*</i> tables in schema _SYS_BI)	Yes	Yes	Yes
MDX Hierarchy (Column Views in the Container schema)	Yes	Yes	No
SQL Hierarchy (classic SQL View in the container schema)	No	Yes	No

**Note:**

With the *Auto* setting, NO SQL Hierarchy view is generated in HANA On-Premise. However, later on, if you migrate the Calculation View to SAP HANA Cloud, and because SAP HANA Cloud does not provide support for MDX, the SQL hierarchy (SQL views) will be generated instead of the MDX ones without changing the setting. This makes migration to SAP HANA Cloud simpler. For more detail, you can consult SAP Note [3139372](#).

Name of a Calculation Views and its Associated Hierarchy Views

The following table shows an example of how the various objects related to a Calculation View are named in the corresponding HDI Container schema.

In this example, one level hierarchy, *PROD_LEV_HIER*, has been modeled within the Calculation View *HA300::CVC_SALES_HIER_00*. The Calculation View has three attributes and two measures.

Base Calculation View	HA300::CVC_SALES_HIER
MDX HIERARCHY (Column View)	HA300::CVC_SALES_HIER/PROD_LEV_HIER/hier/PROD_LEV_HIER
SQL HIERARCHY (SQL View)	HA300::CVC_SALES_HIER/PROD_LEV_HIER/sqlh/PROD_LEV_HIER
Additional technical MDX hierarchies (Columns Views)	<ul style="list-style-type: none"> • HA300::CVC_SALES_HIER/Measures/hier/Measures • HA300::CVC_SALES_HIER/PRODUCT_ID/hier/PRODUCT_ID • HA300::CVC_SALES_HIER/PROD_GROUP/hier/PROD_GROUP • HA300::CVC_SALES_HIER/PROD_NAME/hier/PROD_NAME



LESSON SUMMARY

You should now be able to:

- Define Hierarchies

Implementing Currency Conversion

LESSON OVERVIEW

This lesson describes how to set up currency conversion for measures.

Business Example

For a worldwide company, sales are conducted in a lot of different currencies. However, you want to display data and amounts with one single currency to be able to aggregate data. So you need to set up a currency conversion.

You want to know more about the native functionality of SAP HANA to implement currency conversion in your information models.



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Explain the general principles of currency conversion

Currency Conversion in Data Reporting Rationale



- As most front-end tools do not allow defining or switching reporting currency in the UI, and as there might not be such information in master data, we have to convert the possibly many monetary document currencies into just a few.
- SAP HANA has the necessary functions needed to achieve currency conversion during data modeling.

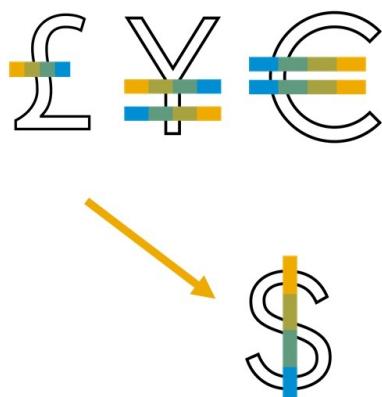


Figure 103: Currency Conversion

When you build information models, the source data is often expressed in a single currency. Typically, for sales data, this would be the transaction currency at the date of sale.

For reporting purposes however, it is often necessary to convert the currencies. The purposes can include (but are not limited to) the following:

- Corporate Reporting

When using a global reporting currency in corporate reporting, all values should be displayed in the global reporting currency.

- Regional Reporting

For example, the US region might want to see European figures in USD.

- P&L Reporting

You might want to analyze the effects of currency gains and losses.

- Accounting

Data conversion is a strong requirement in multicurrency general ledger transactions.

Currency Conversion



As currency exchange rates fluctuate constantly in the global markets, when converting it is necessary not only to define the **source** and **target** currencies when converting, but also to define the **time** when currency conversion should take place.

Examples could be:

- Billing Date
- Posting Date
- Financial Year End
- Today's Date



Figure 104: Currency Conversion

Due to the permanent fluctuation of currency exchange rates, you have to define a smart conversion process and, in particular, define which date must be considered to define the conversion rate to apply.

This requirement is even stronger when the source or target currency is volatile.

Native Currency Conversion in SAP HANA Information Models

SAP HANA offers an elaborate conversion mechanism that is based on the following building blocks:

- A set of technical tables to store master data about currencies, exchange rate types, and the exchange rate values.
- The concept of **semantic type**, which allows a flag to measure with the *Amount with Currency* semantic type.
- An interface to define, for each amount measure, how the conversion should be processed by the information model (which rate and conversion dates should be applied, where to find the source and target currency).

Currency Conversion Approaches in SAP HANA

In SAP HANA, data conversion can be implemented in both graphical and script-based views.

Graphical calculation views in SAP HANA provide the easiest way to convert currencies because the conversion modeling can be done using the graphical interface.

Alternatively, in case of constraints in the master data or because of the complexity of the reporting requirements, you can model the currency conversion within a scripted view.

However, this feature is based on a SQLScript function, *CONVERT_CURRENCY*, which is based on column engine plan operators (or CE functions), which are now deprecated.



Note:

If you have to adapt an existing implementation of this *CONVERT_CURRENCY* function, you can find more information in the *SAP HANA SQL and System Views Reference guide*. This is available at <http://help.sap.com/hana>.

Applying Conversion to Lower Aggregation Nodes

From SAP HANA SPS12 onwards, it is possible to define a currency conversion on any aggregation node of a calculation view. This enhancement allows you to maintain a currency conversion at an intermediate level of your calculation view. For example, when you need to combine two different data sets with a Union node, and only one of the data sets needs to be converted.



Note:

Up to SPS11, currency conversion could only be defined at the uppermost level of the calculation view, either the upper Aggregation node of a cube calculation view, or the star join node of a cube with star join calculation view.

TCUR Schema

One of the key building blocks to enable data conversion in SAP HANA is a set of tables to define currencies and exchange rate types, and to store the conversion rates.

Table 13: Required Conversion Tables

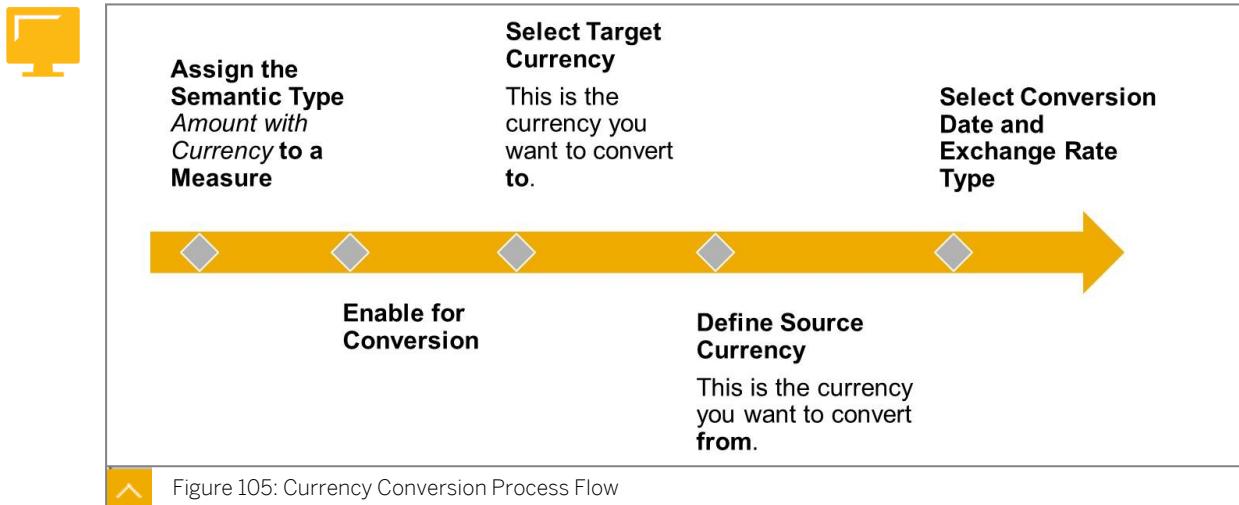
Table Name	Description
TCURC	Currency codes
TCURR	Exchange rates
TCURV	Exchange rate types for currency translation
TCURF	Conversion factors
TCURN	Quotations
TCURX	Decimal places in currencies

These tables exist in most SAP systems (in particular, SAP Business Suite and SAP Business Warehouse).

To enable conversion in SAP HANA, these tables must be available in the SAP HANA database; for example, in a dedicated schema for currencies and exchange rates.

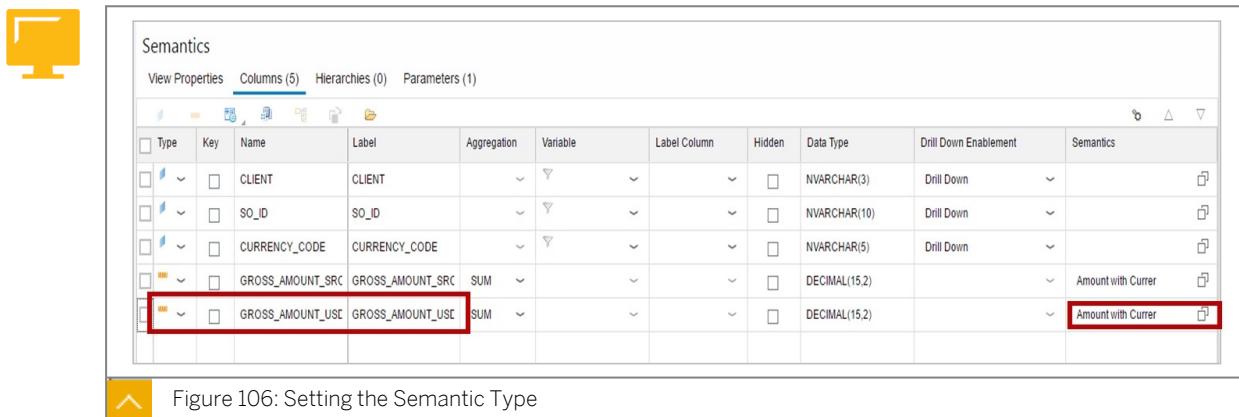
If the models are based on data replicated from another system, the replication process should also include these tables so that they are always synchronized with the ones located on the source system.

Implementing Conversion in Calculation Views



The figure, Currency Conversion Process Flow, depicts the process flow for currency conversion within a calculation view.

Setting the Semantic Type



By default, a measure has no semantic type.

When a measure contains an amount, and if you want to enable conversion, you need to change its semantic type to *Amount with Currency Code*. You can do this in the following ways:

- In the *Output* pane of the Aggregation node, select an aggregated column and, in the *General Properties* pane, define the semantic type. You can also double-click the column. This can also be done on the Star Join node of a cube with star join calculation view.
- Alternatively, when the currency conversion is implemented at the uppermost level of the calculation view, select the Semantics node, and choose *Assign Semantics*.

**Note:**

The semantic type *Amount with Currency Code* can be used even when conversion is not actually used.

In this case, you only have to set the currency property to identify the currency (or currencies) in which the amounts are expressed.

Defining Currency Conversion Settings



Assign Semantics For GROSS_AMOUNT_USD

Semantic Type	Amount with Currency Code		
Application Type	ERP		
<input type="checkbox"/> Reuse Semantics	<input type="checkbox"/>	<input checked="" type="radio"/> Reference	<input type="radio"/> Copy
*Display Currency	Fixed	USD	<input type="checkbox"/>
<input checked="" type="checkbox"/> Conversion <input checked="" type="checkbox"/> Decimal Shifts <input type="checkbox"/> Rounding <input type="checkbox"/> Shift Back <input type="checkbox"/> Reverse Look Up			
Conversion Tables	Definition		
*Client	Fixed	800	<input type="checkbox"/>
*Source Currency	Column	CURRENCY_CO...	<input type="checkbox"/>
*Conversion Date	Input Parameter	INP_CONV_DATE	<input type="checkbox"/>
<input type="checkbox"/> *Data Type	DECIMAL	15	2
<input type="checkbox"/> Generate	<input type="checkbox"/>		
*Exchange Type Fixed EZB <input type="checkbox"/> *Target Currency Fixed USD <input type="checkbox"/> Exchange Rate Select Column <input type="checkbox"/> Upon Failure Fail <input type="checkbox"/> Accuracy Intermediate rounding <input type="checkbox"/>			
<input type="button" value="OK"/> <input type="button" value="Cancel"/>			

Figure 107: Currency Conversion Settings

After assigning the semantic type *Amount with Currency Code*, you can enable conversion and define the main parameters used for conversion.

Key Settings for Currency Conversion

Table 14: Key Settings for Currency Conversion

Setting	Description	Options
Client	The client (MANDT) to use to filter the TCUR* tables content	Session Client / Fixed Client Number / Column / Input parameter
Source Currency	The currency in which the amounts to convert are expressed	Fixed / Column
Target Currency	The currency in which the amounts must be converted	Fixed / Column / Input Parameter
Exchange Type	The type of rate used to convert amounts. Example: Spot rate, average rate...	Fixed / Column / Input Parameter

Setting	Description	Options
Conversion Date	The date used to match an amount and the corresponding conversion rate	<i>Fixed / Column / Input Parameter</i>
Exchange Rate	(optional) A column from the source data that contains the exchange rate to be used	
Data Type	The data type of the converted measure (overrides the data type of the converted column)	Example: Decimal (15,2)
Generate	If selected, this option creates a column that indicates for each converted amount the (target) currency in which it is expressed.	
Upon Failure	Specifies the behavior if the conversion cannot be executed (for example, if the rate	<i>Fail</i> (a query on the view generates an error), <i>NULL</i> (the column is not populated), <i>Ignore</i> (keeps the source amount without converting it)
Accuracy	Defines how the conversion must be performed	Intermediate rounding / Retain all possible digits

It is important to carefully define how the exceptions must be handled when converting data. In addition, to reduce the risk of conversion failure, make sure that the currency conversion tables TCUR* in your SAP HANA system are updated on a regular basis, in particular in a side-by-side scenario where they should always be in sync with the data imported from the remote SAP system.



Note:

The result currency column is never exposed to client tools. It is only available to other SAP HANA views, where it can be used in additional calculations.

Decimal Shift and Rounding

By default, the precision of all values is two digits in SAP ERP tables.

As some currencies require accuracy in value, decimal shift moves the decimal points according to the settings in the TCURX currency table. If you want to round the result value after currency conversion to the number of digits of the target currency, select the *Rounding* checkbox.

Decimal shift back is necessary if the results of the calculation views are interpreted in ABAP. The ABAP layer, by default, always executes the decimal shift. In such cases, decimal shift back helps avoid wrong numbers due to a double shift.

Reusing Currency Conversion Settings between Columns

From SAP HANA 2.0 SPS02, it is possible to reuse the currency conversion settings for other measures in the same node of a Calculation View. This reduces manual definition and allows

for more consistency, by avoiding mistakes. The settings can be applied to several measures at the same time.

The currency conversion settings can be reused in two different ways:



- Reference

The settings defined in a measure are applied **as is** in the other measures that you select, and cannot be modified in the other measures.

In other words, the settings will always remain consistent and only the source measure for currency conversion setting can be changed, thus impacting the ones that reference it.

- Copy

The setting defined in a measure is just copied to the other measures, but they are not bound to each other. The currency conversion settings of the other measures can be freely modified.

Using an Input Parameter for the Currency

The input parameter can also be described as a prompt, in that it asks the user what currency to use.



Figure 108: Creating an Input Parameter

If you want to define the currency at runtime, when the view is executed, you can create an input parameter.

VARCHAR (5) is the way that the currency code is defined in the TCUR* tables, so to be consistent, we recommend that you define the input parameter with the same data type.



LESSON SUMMARY

You should now be able to:

- Explain the general principles of currency conversion

Defining Time-Based Dimension Calculation Views



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Create a Time-Based Dimension Calculation View

Time-Based Dimension Calculation Views

When you create a dimension calculation view you can choose either **Standard** or **Time** type. The default is **Standard**.

The tables containing measures generally have a date or date-time column to clearly define when an event occurred, or which time period a measure relates to, such as a year, month or a quarter. The purpose of the *time* type DIMENSION calculation view is to provide additional time attributes to compliment what is already provided in the source data. For example, if the data source provides a date, the time dimension calculation view can provide the corresponding week, month, quarter, year, and financial period.

With these time related attributes you can define a time hierarchy and aggregate measures over different granularities of time.

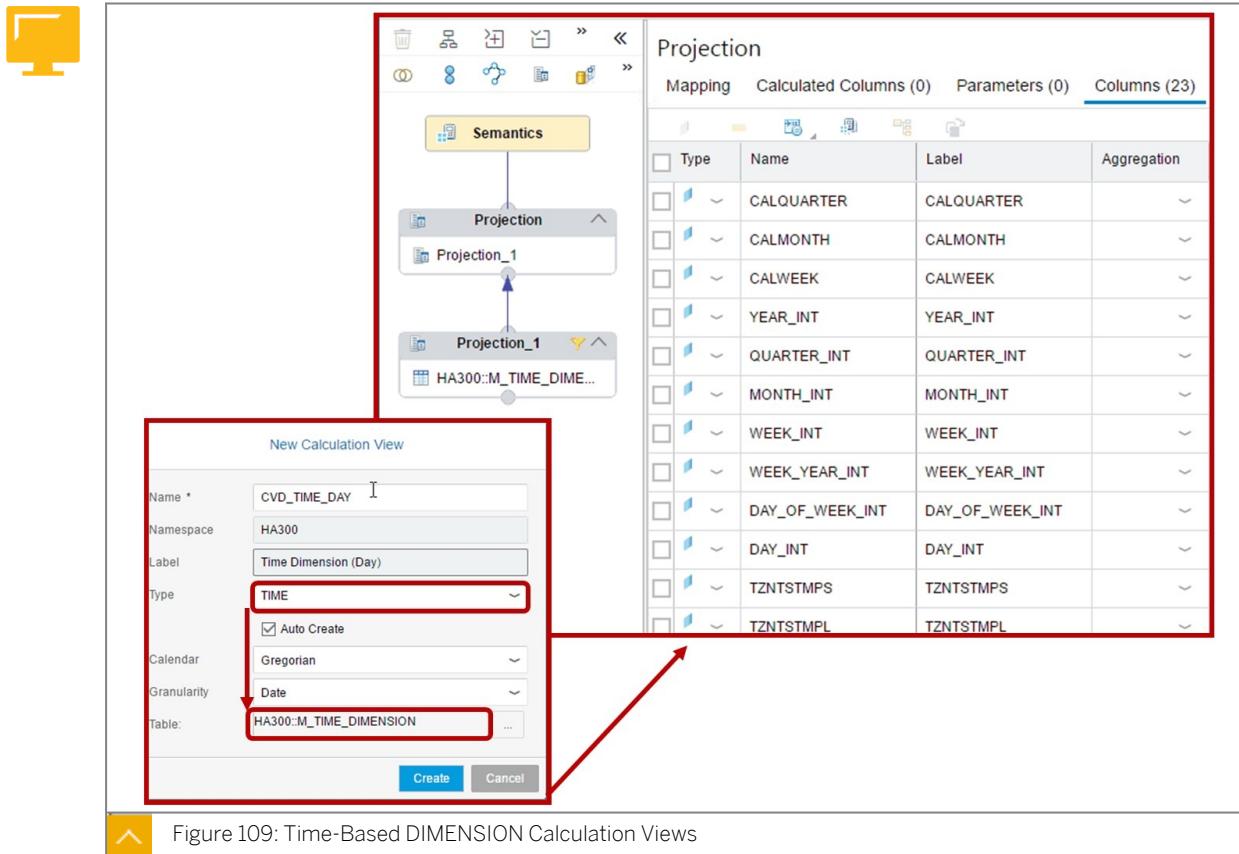


Figure 109: Time-Based DIMENSION Calculation Views

Different Calendar Types in Time Calculation View

The following are the different calendar types available in time calculation views:



- Gregorian

The Gregorian calendar is made up of years, months, and days. You can adjust the level of granularity, down to hours, minutes, or seconds.

- Fiscal

The fiscal calendar is organized into fiscal years and fiscal periods. Several fiscal year variants can be defined depending on your reporting needs.



Note:

The fiscal calendar is especially useful to display data in your calculation views according to the fiscal calendar tables available in your SAP ERP system.

Data Source for Time Data Used in Time DIMENSION Calculation Views

The time attributes are not calculated at run-time but are generated periodically and stored in source tables. It is possible to define and populate the source tables for Time DIMENSION calculation views inside your local HDI container, using an assistant available in the *Development* perspective. This allows you to customize the definition of time granularity, fiscal periods, time range, and so on, for your specific HANA Database Module, regardless of what is defined for the entire database in the system tables *M_TIME_DIMENSION_xxx* and *M_FISCAL CALENDAR* (schema *SYS_BI*).

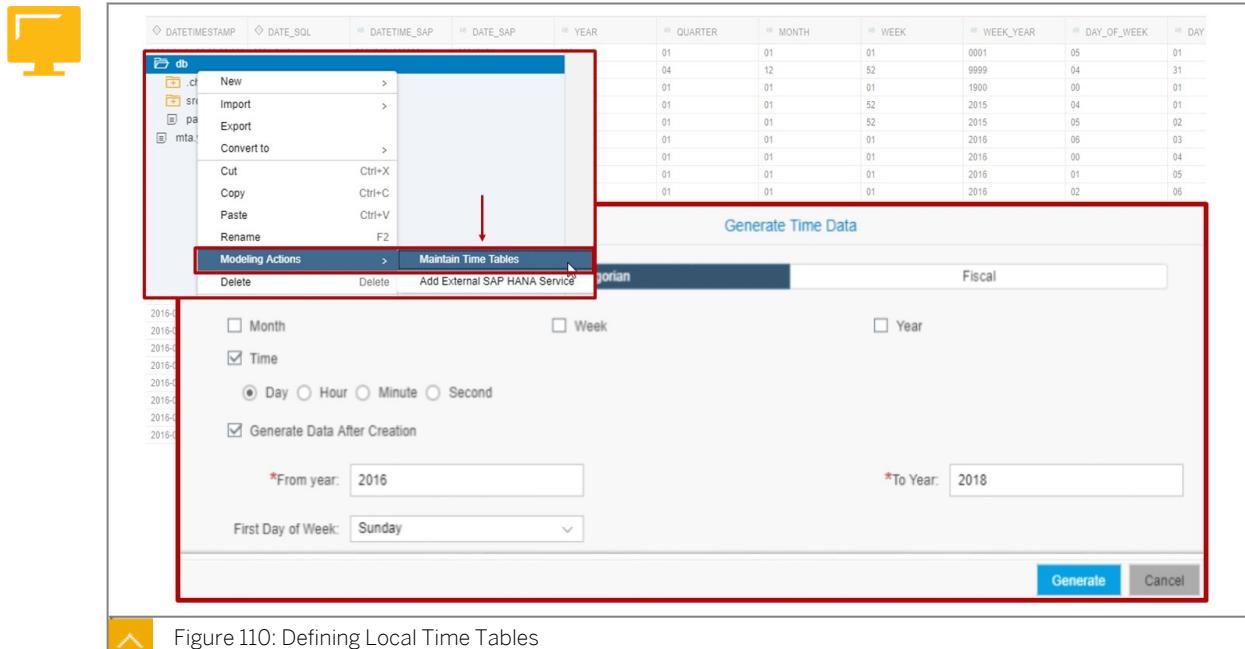


Figure 110: Defining Local Time Tables

The assistant can be manually run to generate new time attribute values, in other words, more records for future dates.

From HANA 2.0 SPS06 onwards, it is also possible to automate the generation of new records by using the SAP provided procedure "SYS"."UPDATE_TIME_DIMENSION".



Note:



LESSON SUMMARY

You should now be able to:

- Create a Time-Based Dimension Calculation View

Learning Assessment

1. A restricted column provides a subset of the original column filtered by attribute values.

Determine whether this statement is true or false.

- True
 False

2. When is the filter expression applied in a calculation view?

Choose the correct answers.

- A Before a query on the calculation view begins to execute
 B On the final result of a query, similar to a 'where' clause
 C Before any joins are executed within the calculation view

3. When the *Default Client* property of a calculation view is set to *Session Client*, the data is filtered dynamically based on the CLIENT assigned to the user.

Determine whether this statement is true or false.

- True
 False

4. Why do you use a variable?

Choose the correct answer.

- A To provide a missing value to a formula
 B To filter results

5. In a calculated column, which do you use as a placeholder for a dynamic value in an expression?

Choose the correct answer.

- A Variable
 B Input parameter

6. In a hierarchy, what is an orphan node?

Choose the correct answer.

- A A node that has no parent node
- B A node that has no child nodes
- C A node that contains only one member

7. What are two types of modeled hierarchy you can create using calculation views?

Choose the correct answers.

- A Parent-Child
- B Referential
- C Level

8. While converting currencies, in addition to source and target currencies, which is also needed?

Choose the correct answer.

- A Date
- B Country
- C VAT rate

9. What is a time-based dimension calculation view?

Choose the correct answer.

- A A view that is only accessible by users for a limited time
- B A view that stores the values of attributes across time

Learning Assessment - Answers

1. A restricted column provides a subset of the original column filtered by attribute values.

Determine whether this statement is true or false.

True

False

Correct — Indeed, a restricted column provides a subset of the original column filtered by attribute values. For example — a column that shows only female employees (the column is based on all employees but a restriction is defined using a filter expression on attribute Gender = female).

2. When is the filter expression applied in a calculation view?

Choose the correct answers.

A Before a query on the calculation view begins to execute

B On the final result of a query, similar to a 'where' clause

C Before any joins are executed within the calculation view

Correct — A filter is applied before a query on the calculation view begins to execute and before any joins are executed within the calculation view

3. When the *Default Client* property of a calculation view is set to *Session Client*, the data is filtered dynamically based on the CLIENT assigned to the user.

Determine whether this statement is true or false.

True

False

Correct — Indeed, when the *Default Client* property of a calculation view is set to *Session Client*, the data is filtered dynamically based on the CLIENT assigned to the user

4. Why do you use a variable?

Choose the correct answer.

- A To provide a missing value to a formula
 B To filter results

Correct — A variable is used to filter results.

5. In a calculated column, which do you use as a placeholder for a dynamic value in an expression?

Choose the correct answer.

- A Variable
 B Input parameter

Correct — You use an Input parameter as a placeholder for a dynamic value in an expression and not a variable.

6. In a hierarchy, what is an orphan node?

Choose the correct answer.

- A A node that has no parent node
 B A node that has no child nodes
 C A node that contains only one member

Correct — An orphan node has no parent node.

7. What are two types of modeled hierarchy you can create using calculation views?

Choose the correct answers.

- A Parent-Child
 B Referential
 C Level

Correct — Parent-Child and Level are valid hierarchies. Referential is not a type of hierarchy but refers to a type of join.

8. While converting currencies, in addition to source and target currencies, which is also needed?

Choose the correct answer.

- A Date
- B Country
- C VAT rate

Correct — Date is required in addition to the source and target currencies.

9. What is a time-based dimension calculation view?

Choose the correct answer.

- A A view that is only accessible by users for a limited time
- B A view that stores the values of attributes across time

Correct — A time based calculation view stores the values of attributes across time and is not a view that is only accessible by users for a limited time

UNIT 4

Using SQL in Models

Lesson 1

Introducing SAP HANA SQL

169

Lesson 2

Query a Modeled Hierarchy Using SQL

185

Lesson 3

Working with SQLScript

189

Lesson 4

Creating and Using Functions

197

Lesson 5

Creating and Using Procedures

203

UNIT OBJECTIVES

- Describe SAP HANA SQL
- Query a modeled hierarchy using SQL
- Develop Skills using SQLScript
- Work with functions
- Create and use procedures

Unit 4

Lesson 1

Introducing SAP HANA SQL

LESSON OVERVIEW

This lesson provides an introduction to the SQL language in SAP HANA, and covers the following topics:

- Overview of SQL language elements
- Identifiers
- SQL data types
- Predicates and operators
- Functions and expressions
- SQL statement examples
- Defining and loading tables using source files

Business Example

As an SAP HANA consultant, you should learn how to perform tasks such as querying source data out of a table or a data model, copying tables, or altering table column types. Knowledge of SQL will let you perform these tasks. Also, in order to develop sophisticated models, you sometimes need to write SQL in order to enhance calculation views with custom code.



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Describe SAP HANA SQL

SQL in SAP HANA Modeling

Structured Query Language (SQL) is a hugely popular and standardized **database language**, defined by the American National Standards Institute (ANSI). SQL is used to build a database and its tables, to read, update, and insert data, and to define security to relational databases.

Importance of SQL in SAP HANA Modeling

SQL plays an important role in SAP HANA modeling and can be implemented in different SAP HANA modeling objects, including the following:

- Expressions in calculation views
- Procedures
- Input parameters
- Scalar and table functions
- Analytic privileges

Also, to debug and trace calculation views, it is essential that you understand how to interpret the SQL that is generated from the calculation view instantiation process.



Caution:

SQL should not be considered as an alternative approach to graphical modeling. For example, you should NOT create an SQL-based object (function or procedure) to achieve a result that a graphical calculation view can achieve (even if you have advanced SQL knowledge). Graphical calculation views are highly optimized at run-time and are dynamically pruned to the conditions of the calling query.



Where can we implement SQL in SAP HANA modeling?

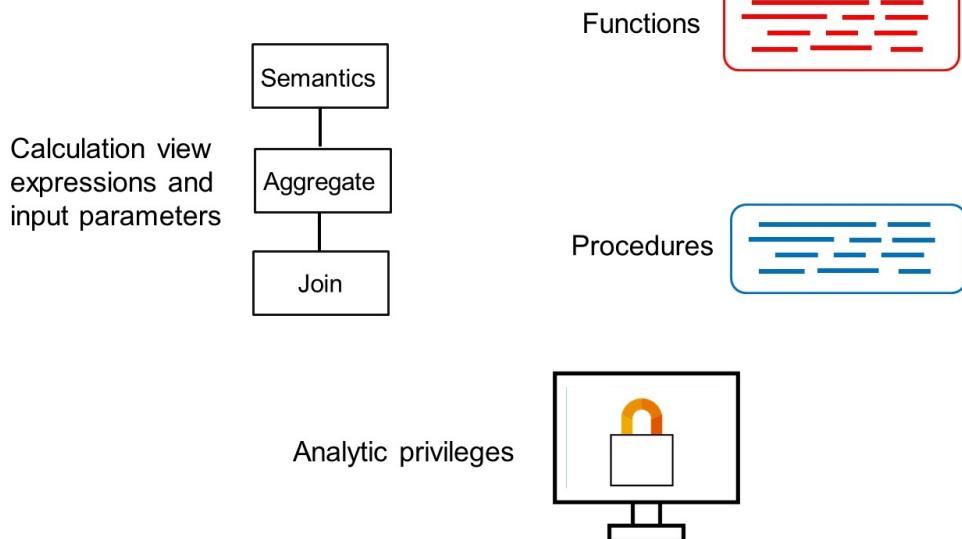


Figure 111: Where SQL is used in HANA modeling



Note:

In this course, we will not go into the language of SQL in depth, but will focus more on where and how we implement SQL in SAP HANA modeling. For a deep dive on the SQL and SQLScript language, refer to the SAP course *HA150 – SAP HANA SQL..*

Implementing SQL in Calculation Views

As the figure, Use of SQL in Calculation View Expressions, shows, SQL can be used to create expressions in calculation views to define the following:

- Calculated columns
- Filters
- Restricted columns

In addition, SQL can be used to return values for input parameters. This is very powerful because input parameters are used in many places throughout calculation views as dynamic placeholders for filters, ranking, user prompts, calculations and more.



Note:

Variables do not make use of SQL to return values.



Use of SQL in Calculation View expressions

The screenshot shows the SAP HANA Studio interface with the title "Use of SQL in Calculation View expressions". The main area is titled "Expression Editor (Age_of_order)" and contains the SQL code: `DAYS_BETWEEN("SALES_DATE", CURRENT_DATE)`. Below the editor are three panels: "Elements" (Columns, Calculated Columns, Restricted Columns, Parameters), "Functions" (Conversion Functions, String Functions, Mathematical Functions, Date Functions, including ADD_DAYS(), ADD_MONTHS(), ADD_SECONDS(), ADD_WORKDAYS(), CURRENT_DATE, CURRENT_TIME, DAYNAME(), DAYOFMONTH(), DAYOFYEAR()), and "Operators" (+, -, *, **, /, %, =, !=, >, <, >=, <=).

Figure 112: Use of SQL in Calculation View Expressions

It is essential for modelers to grasp the basics of SQL, because they will encounter SQL often throughout their modeling activities. Pay particular attention to the long list of available SQL functions within expressions. These functions can provide significant, additional options when you are trying to develop some logic where data must be manipulated. For example, SQL provides many string manipulation functions that are useful for re-formatting a field, or extracting some characters from it. Also, there are many predefined data functions available in SQL that can help you to calculate with dates, for example, to find the number of days between two dates.

You may be left wondering if you could even abandon calculation views completely and instead write the data models using only SQL. To a large extent this would be possible, but remember, calculation views carry a lot of valuable metadata including many flags and settings that provide the optimizer with run-time hints for optimization and ensure accurate results under different query conditions. Calculation views also carry metadata that is consumed by reporting tools to indicate how results should be interpreted and presented.

In addition, there are graphical tools for lineage and impact analysis when using calculation views, to support developers who need to trace a complex data model. And of course, who wants to read through lines of code in order to figure out what a model is actually doing?



Note:
SQL expressions in calculation views use standard SQL and not the special SAP HANA version of SQL called SQLScript.

Classifications of SQL Statements

For those who are new to SQL, let's learn the basics.



Note:
If you already have SQL skills, you can skip this theory section and go straight to the exercise.

SQL statements are grouped into the following classifications:

Data definition language (DDL)

DDL is a group of SQL statements to create, modify, and delete database objects such as tables, views, and procedures. You use DDL statements to define the physical layer (create tables) and logical layer (create views).

Data manipulation language (DML)

DML is a group of SQL statements to insert, update, delete, and read the data records; in other words, working on the data and not the database objects.

Data control language (DCL)

DCL is a group of SQL statements to define the database users, roles, and to grant security clearance to the database objects. (For example, who can access a particular table?)

Your SQL code can mix statements from any of the three classifications. For example, in one block of SQL, you could create statements that do the following:

1. Create a table (DDL)
2. Fill the table with records (DML)
3. Read the records (DML)
4. Grant permissions to other users to read the table (DCL)

SQL is a descriptive, or sometimes called declarative, language. With SQL you describe using code, what you would like to achieve (the intent), but not how to explicitly achieve it. When executed, the SQL is optimized automatically to achieve the best performance, which is always its main goal. Examples of optimization are, deciding on the sequence of operations or whether data can be processed in parallel if there are no dependencies between operations.

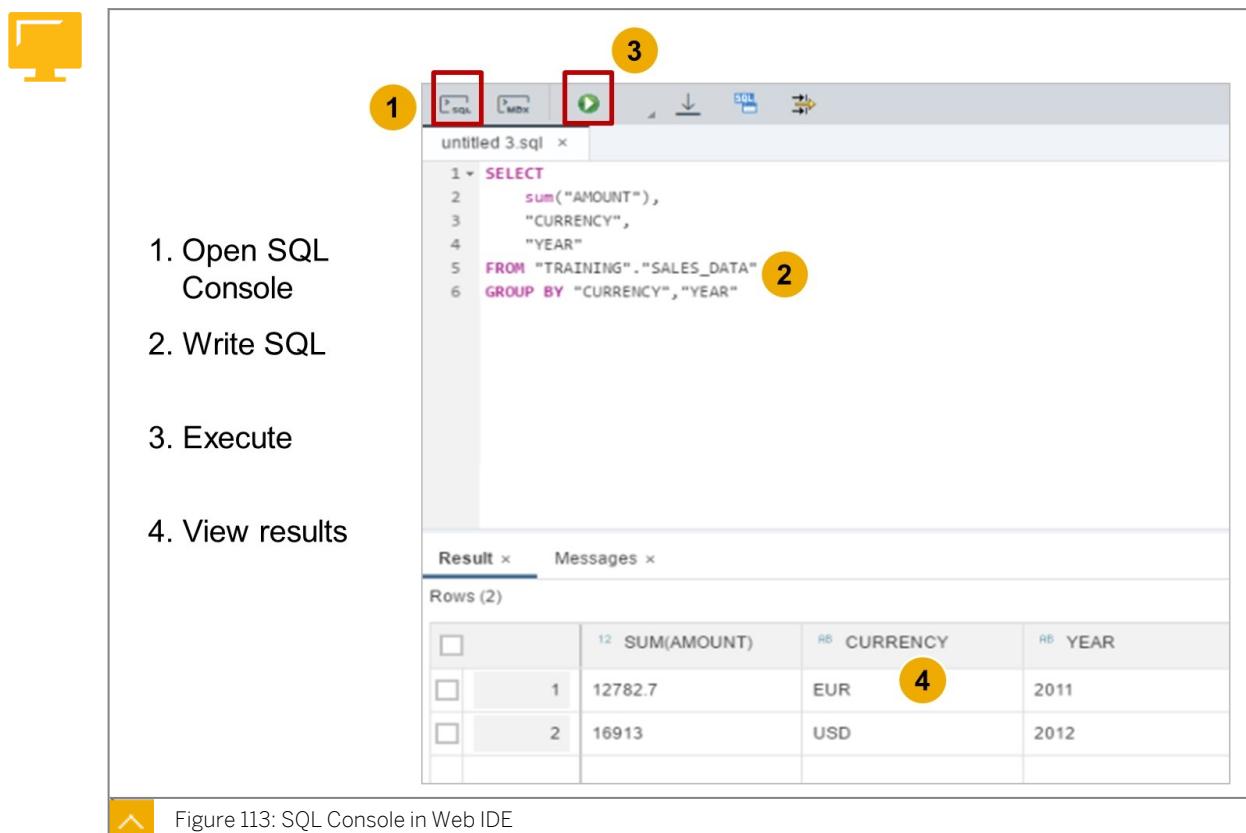
Depending on the growth of data and even the reconfiguration of the hardware (for example, partitioning, scale-out), the optimization may change over time. But your SQL code does not usually need to be changed.

In contrast, procedural languages provide opportunities to tightly control the flow of execution by explicitly stating how something should be achieved. In other words, you are taking over the control of optimization and leaving little or nothing for the optimizer to work on. Procedural languages are sometimes referred to as imperative languages. These are often associated with application development languages such as C++, JAVA or ABAP.

SAP HANA - SQL Console

Most databases include an SQL Editor or Console. SAP HANA includes an SQL Console and it is a built-in tool of Web IDE and can be used to write SQL statements and also directly execute them on the SAP HANA database.

Although the SQL Console is ideal for executing one-time instructions, such as creating a temporary table or executing a test query, it is not recommended for generating reusable development objects, such as tables, views, indexes, and so on. For this, we recommend you use design-time objects, because they are managed in a *Project* and can easily be deployed when needed.



You can launch the SQL Console from the Database Explorer of SAP Web IDE. Each time you launch the SQL console a new tab appears. You can open as many tabs as you wish.

When you launch an SQL Console in SAP Web IDE from the Database Explorer view, you need to be aware of which database connection you are using because the executed SQL will run against that connection and also the design-time errors will be based on the connection you are using. The database connection for your console can be selected in a number of ways:

- Highlight an entry in the database connection list and then press *Open SQL Console* button in the toolbar.
- Right-click an entry in the database connection list and then choose *Open SQL Console* in the context menu.
- Expand a database connection and locate a database object, then right-click the object and choose an SQL expression from the context menu such as *select*.

You can clearly see which database connection you are using because this is displayed at the top of the screen.

Regardless of which connection you choose, you can easily swap the connection by using the button in the toolbar *Connect this SQL console to a different database*.

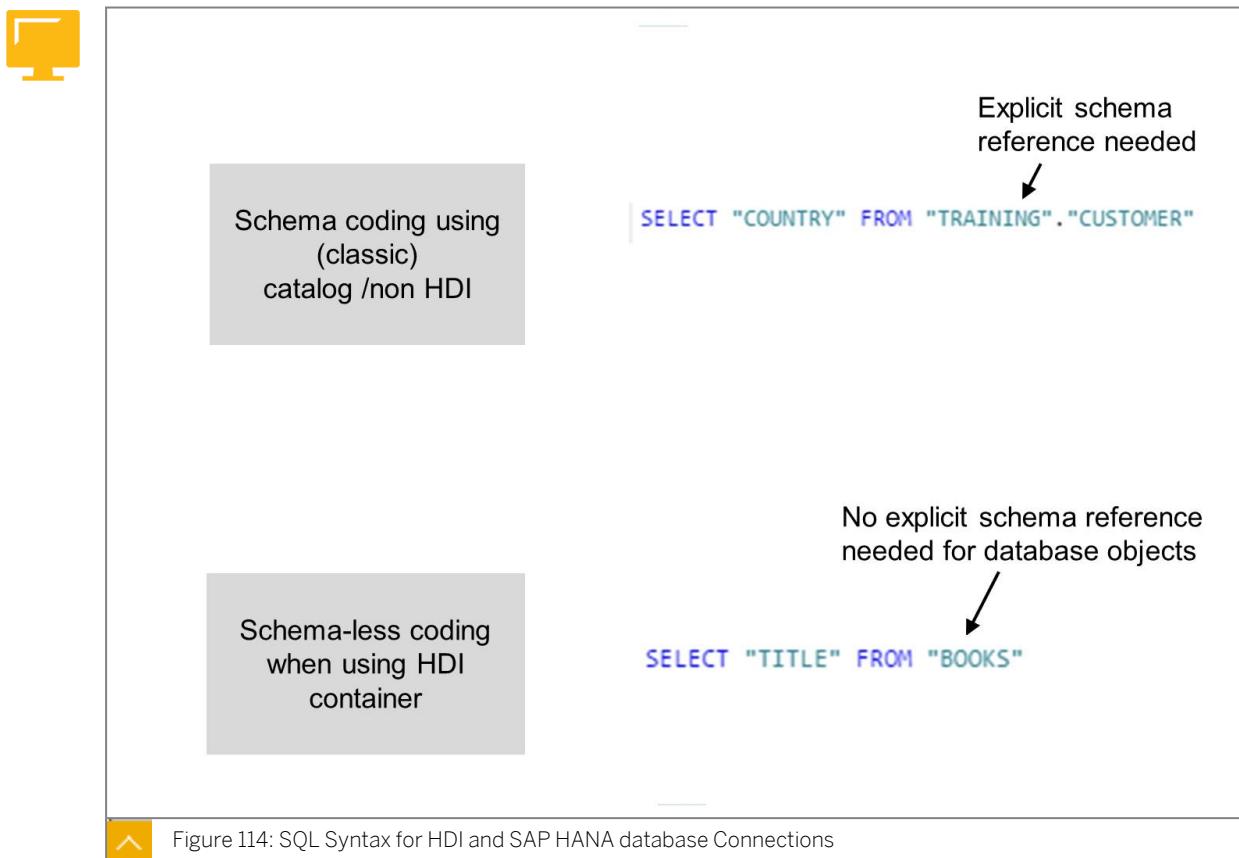
HDI or Classic Database Connections

There are two main types of database connection:

- HDI Container (used in XS Advanced projects)
- SAP HANA database (a classic, catalog type connection to one or more schemas)

Both of these provide access to database objects. Many developers will be already be familiar with the SAP HANA database catalog type connection. The HDI Container type connection is specific to SAP HANA and requires a little more investigation.

It is important that you understand which type you are using for your SQL Console because this affects how you write the SQL code.



When using a HDI Container type, your SQL does need to include the database schema. This is called schema-less coding. Of course, the SQL is always executed against a database schema, but the schema is determined automatically from the HDI container on which you are executing your SQL.

If you are using a connection that is an SAP HANA database type, then you must explicitly reference the schema, unless you are using the user default schema.

SQL Console Auto-Complete and Other Productivity Aids

There are a number of productivity aids built into SAP Web IDE.

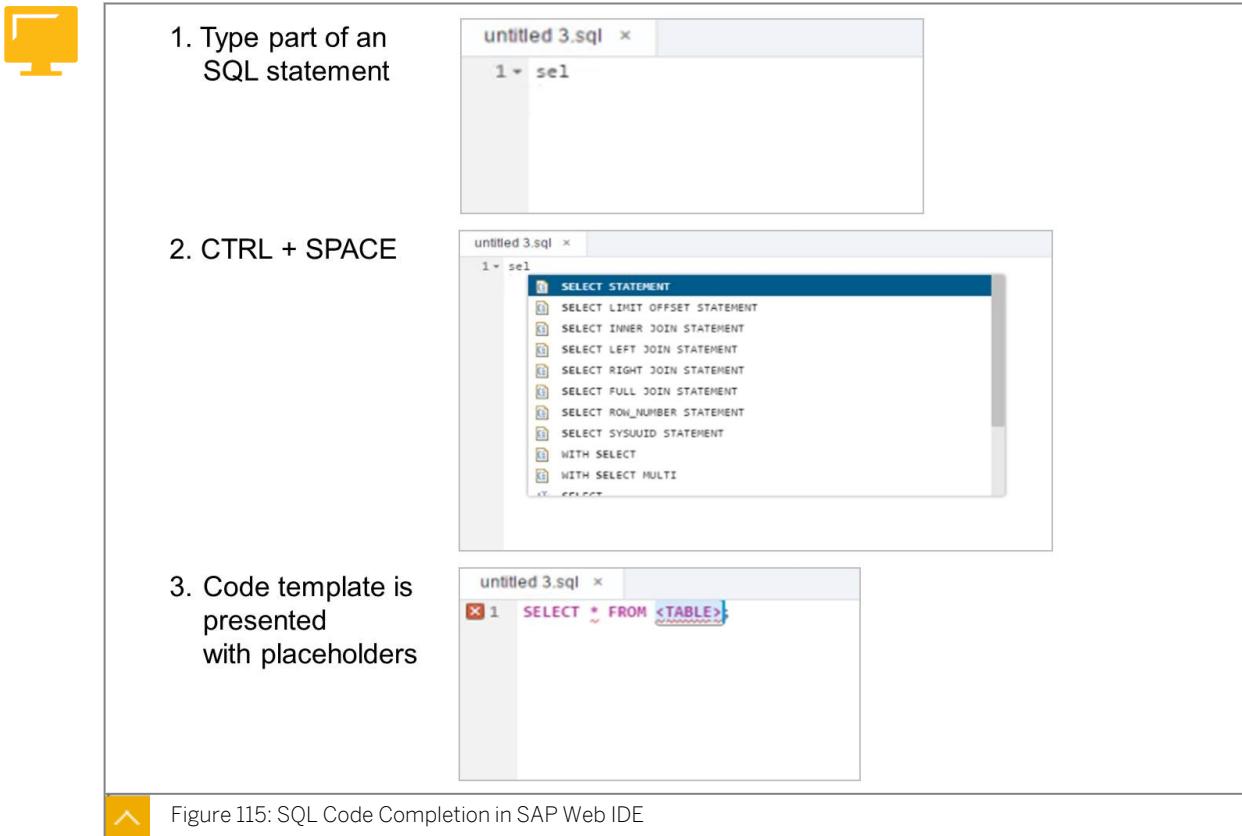


Figure 115: SQL Code Completion in SAP Web IDE

As the figure, SQL Code Completion in SAP Web IDE, shows, the SQL Console includes an auto-completion feature, which can be activated by pressing **Ctrl + SPACE** while entering a command.

The auto-complete feature is also available in the code editor when you create functions and procedures. This speeds up the writing of SQL by building the basic statement structure for you, and ensures that you include all important parameters. Simply enter an expression such as *select* or a function such as *daysbetween* and then press **Ctrl + SPACE** to use auto-complete so that you see the placeholders that represent the required parameters. You don't even need to type the entire word – for example if you type the letters *le* the selection list will include multiple options including *Left()*, *Left Outer Join*, *Length()* and so on.

There is no validation button or menu option in the SQL Console because SQL code is constantly checked in real time for correctness as you type. If there are design-time errors, a red alert icon appears on the left of the line in error. You can hover over the icon and read the error text. Simply fix the problem and the red alert icon disappears. For example, a missing bracket would cause a syntax design-time error. The console is even smart enough to know that a statement, although syntactically correct, is still invalid if it refers to a table that does not exist. Also, even if there are no design-time errors, you may still receive run-time errors when you execute the SQL statement. For example, you may have forgotten to add an attribute to the GROUP BY clause that is used in a SELECT statement which contains a *sum* on a measure.

At run-time, if errors are encountered, at the first error you will be prompted and given three options to choose how you wish to proceed:

1. Ignore this statement and continue to the next statement
2. Ignore this statement and all other statements with errors

3. Abort the execution immediately



Hint:

Sometimes you only want to execute a snippet of code or a single statement in the console. If so, then simply highlight a section of SQL code and then hit F8 or use the execute button in the toolbar. Then, only that highlighted code is executed. If you do not highlight any code, then all code in the SQL console is processed when you execute. This is a great way of testing single statements.



Hint:

When you need to include the name of a database object in your code, rather than type it in manually, just drag and drop it from the database explorer connection on the left panel.

Example of an SQL Statement



Note:

The following section is not included in the instructor presentation and is provided for self study and is aimed at those who are new to SQL.



-- Find a list of SAP courses about HANA

```
SELECT name, description, cost
FROM courses
WHERE company = 'SAP'
      AND description LIKE '%HANA%';
/*
```

Look at the comments, SELECT statement, identifiers (name, description, cost, company), predicates (= and LIKE), and operators (AND)

```
*/
```



Figure 116: Example of an SQL Statement

Let's have a look the example SQL statement in the figure, Example of an SQL Statement:

- All text after a double hyphen (--) or between the /* and */ text is seen as comments. In this case, you specified in the top line what the statement does.
- The SELECT statement is the most used statement when writing SQL code. It allows you to read data from a data source, as follows:

- You first specify which columns (fields) you want to read from the data source. In this case, you want to read the name of the course, the course description, and the cost of the course. You can read ALL the columns (fields) from a data source by writing `SELECT *`. This is not recommended, especially for columnar databases like SAP HANA because it can degrade performance.
- You can specify the name of the data source in the `FROM` clause, which in this case is a table named `courses`.
- You can then restrict the amount of data returned by specifying a `WHERE` clause. In this case you only want to see courses from SAP.
- You can further restrict the data returned by additional clauses using the `AND` and `OR` operators.
- Finally, you can specify that you only want SAP courses where the description field contains the word HANA somewhere in the text. You do this using the `LIKE` predicate.
- The field names in this statement can be referred to as identifiers.

SQL Language Elements

The table SQL Language Elements defines the elements that make up SQL.

Table 15: SQL Language Elements

Element	Description
Identifiers	Used to represent object names used in SQL statements such as table names or column names
Data type	Specifies the characteristics of a data value such as NVARCHAR or DECIMAL
Expressions	Clause that can be evaluated to return values for example SELECT or GROUP BY
Operators	Used in expressions to perform operations for example <code>>=</code> , <code> </code> , <code>*</code> , <code><></code>
Predicates	Combines one or more expressions to return either TRUE, FALSE or UNKNOWN. Use predicates such as BETWEEN, IN, CONTAINS, LIKE
Functions	Used in expressions to return information from the database using preprogrammed logic for example <code>daysbetween()</code> .

Comments and Code Page



Comments

```
-- Everything after the double hyphen
-- until the end of a line
-- is considered by the SQL parser
-- to be a comment

Set schema "Training"; --end comment

/*
<.....>
*/
```

Examples of use:

```
select revenue from sales_table -- where country = FR

-- select revenue from sales_table where country = FR

select revenue from sales_table where country = FR -- this filter added Jan 2019 by Tito
```

Figure 117: Comments and Code Page

As the figure, Comments and Code Page, shows, comments are delimited in SQL statements as follows:

- Everything from a double hyphen (--) to the end of a line is parsed as a comment.
- Everything between /* and */ is also parsed as a comment. This style of commenting is used to place comments on multiple lines.

The SAP HANA database supports Unicode to allow use of all languages in the Unicode Standard and 7 Bit ASCII code page without restriction.

Identifiers

You use identifiers to represent names used in SQL statements. These include the following examples:

- Table names
- View names
- Column names
- Synonym names
- Index names
- Function names
- Procedure names
- User names
- Role names

Delimited and Undelimited Identifiers



To delimit or not to delimit identifiers?		
Undelimited Identifiers	<ul style="list-style-type: none"> must start with a letter cannot contain any symbols other than digits or an underscore "_" 	TRAINING Training (treated with upper case!) 1TRAINING TRAINING%
Delimited Identifiers	<ul style="list-style-type: none"> enclosed in the delimiter double quotes like this "<identifier>" can contain any character including special characters. 	"TRAINING" "Training" (case sensitive) "1Training" "Training%"

Figure 118: Identifiers

As the figure, Identifiers, shows, there are two kinds of identifiers; undelimited and delimited.

Undelimited table and column names must start with a letter and cannot contain any symbols other than digits or an underscore (_).



Note:

Undelimited identifiers are implicitly treated as upper case. When quoting identifiers, it is possible to use spaces and lower or mixed case in identifiers.

As the figure shows, delimited identifiers are enclosed in the delimiter double quotes, and the identifier can contain any character, including special characters. For example, →AB\$%CD→ is a valid identifier name.



Note:

Delimited identifiers are case sensitive. SAP HANA is case sensitive on database objects such as the names of tables and fields. For example, you can have a table called "DATA" and another called "data". These will be treated as two different tables.

Delimiters can be delimited by single quotation marks, or double quotation marks, as follows:

Single quotation mark

Single quotation marks are used to delimit string literals and a single quotation mark itself can be represented using two single quotation marks.

Double quotation mark

Double quotation marks are used to delimit identifiers and double quotation mark itself can be represented using two double quotation marks.

You can see the different delimiters in the following example:

```
select 'String', 'SAP''s current offering' from "DUMMY"
```

Data Types

Data types are used to specify the characteristics of the data stored in the database. For example, a string can be stored as a NVARCHAR data type. The table, SQL Data Types, lists some of the data types available in SQL. Data types specify the characteristics of a data value.

Table 16: SQL Data Types

Classification	Standard ANSI-92 Data Type
Date/Time	DATE, TIME, TIMESTAMP
Numeric	DECIMAL, INTEGER, SMALLINT, FLOAT, REAL, DOUBLE
Character String	CHAR, NCHAR, VARCHAR, NVARCHAR
Binary	BINARY, VARBINARY



Note:

A special value of NULL is included in every data type to indicate the absence of a value.

Predicates

Predicates are typically used in the WHERE clause of a SELECT statement.



Comparison Predicates	<code><expression> { = != <> > < >= <= } [ANY SOME ALL] { <expression_list> <subquery> }</code>
BETWEEN Predicate (range)	<code><expression1> [NOT] BETWEEN <expression2> AND <expression3></code>
IN Predicate	<code><expression> [NOT] IN { <expression_list> <subquery> }</code>
EXISTS Predicate	<code>[NOT] EXISTS (<subquery>)</code>
LIKE Predicate	<code><expression1> [NOT] LIKE <expression2> [ESCAPE <expression3>]</code>
NULL Predicate	<code><expression> IS [NOT] NULL</code>



Figure 119: SQL Predicates

A predicate is specified by combining one or more expressions or logical operators and returns one of the following logical or truth values:

TRUE, FALSE, or UNKNOWN.

The figure, SQL Predicates, shows the correct syntax for predicates.

The LIKE Predicate

The LIKE predicate is used for string comparisons. A value, expression1, is tested for a pattern, expression2. Wildcard characters (%) and (_) may be used in the comparison

string expression2. `LIKE` returns true if the pattern specified by expression2 is found. The percentage sign (%) matches zero or more characters and underscore (_) matches exactly one character. To match a percent sign or underscore in the `LIKE` predicate, an escape character must be used. Using the optional argument `ESCAPE expression3`, you can specify the escape character that will be used so that the underscore (_) or percentage sign (%) can be matched.

Operators



Unary	operator operand	unary plus operator(+) unary negation operator(-) logical negation(NOT)
Binary	operand1 operator operand2	multiplicative (*, /), additive (+, -) comparison operators (=, !=, <, >, <=, >=) logical operators (AND, OR)
Arithmetic Operators	< expression > < expression > operator < expression >	Negation, Addition, Subtraction Multiplication, Division
String Operator	< expression > < expression >	String concatenation
Comparison Operators	<expression> operator <expression>	>= Greater or equal to <= Less than or equal to !=, <> Not equal
Logical Operators	Search conditions can be combined using AND or OR operators. You can also negate them using the NOT operator.	AND, OR NOT
Set Operators	Set operators perform operations on the results of two or more queries.	UNION, UNION ALL, INTERSECT, EXCEPT



Figure 120: Operators

You can perform operations in expressions by using operators. Operators are grouped into categories, such as *Comparison Operators* or *Logical Operators*. Operators can be used for calculation, value comparison, or to assign values.

SQL Functions



**-- Find a list of SAP HANA courses
-- for the next 4 weeks (28 days)**

```
SELECT name, description, cost, date,  
DAYS_BETWEEN(CURRENT_DATE, Date) AS  
in_x_days  
FROM courses  
WHERE company = 'SAP'  
AND description LIKE '%HANA%'  
AND DAYS_BETWEEN(CURRENT_DATE, Date) <= 28;
```



Figure 121: An Example of SQL Functions

The figure, An Example of SQL Functions, gives an example of a simple SQL function.

SQL Functions

Table 17: Functions

Classification of the Functions	Examples
Data type conversion	CAST, TO_APLHANUM, TO_BIGINT, ...
Date and Time	ADD_DAYS, ADD_MONTHS, ADD_YEARS, DAYS_BETWEEN, DAYNAME, CURRENT_DATE, ...
Number	ABS, ACOS, ASIN, ATAN, COS, SIN, TAN, ...
String	CONCAT, LEFT, LENGTH, TRIM, ...
Miscellaneous	IFNULL, CURRENT_SCHEMA, ...

Functions provide a reusable method to evaluate expressions and return information from the database. They are allowed anywhere an expression is allowed. Functions use the same syntax conventions used by SQL statements. The table, Functions, lists the main functions available in SQL.

Expressions

An expression is a clause that can be evaluated to return values, as shown in the figure, An Example of an SQL Expression Using COUNT.



-- Find out how many unique SAP HANA courses
-- are available

```
SELECT COUNT(DISTINCT name) as num_courses
FROM courses
WHERE company = 'SAP'
AND description LIKE '%HANA%';
```

Figure 122: An Example of an SQL Expression Using COUNT

Types of Expressions in SQL

The table, Expressions, lists the types of expressions used in SQL.

Table 18: Expressions

Expression Type	Description
Case Expressions	Allows the user to use IF ... THEN ... ELSE logic without using procedures in SQL statements.
Function Expressions	Allows SQL built-in functions to be used as an expression.

Expression Type	Description
Aggregate Expressions	Uses an aggregate function to calculate a single value from the values of multiple rows in a column.
Subqueries in expressions	A SELECT statement enclosed in parentheses. The SELECT statement can contain only one select list item. When used as an expression, a scalar subquery is allowed to return only zero or one value.

Expressions: Examples



- **Case expression**

You can use IF ... THEN ... ELSE logic without using procedures in SQL statements.

```
<expression> ::= CASE <expression>
                  WHEN <expression>
                  THEN <expression>, ...
                  [ ELSE <expression> ]
                  { END | END CASE }
```

- **Aggregate Expressions**

```
<aggregate_expression> ::= COUNT(*) | <agg_name> ( [ ALL | DISTINCT ] <expression> )
<agg_name> ::= COUNT | MIN | MAX | SUM | AVG | STDDEV | VAR
```

Figure 123: Examples of Expressions

As the figure, Examples of Expressions shows, in a case expression, if the expression following the CASE statement is equal to the expression following the WHEN statement, then the expression following the THEN statement is returned. Otherwise the expression following the ELSE statement is returned, if it exists.

As the figure also shows, an aggregate expression can contain the following functions:

- COUNT: Counts the number of rows returned by a query. COUNT (*) returns the number of rows, regardless of the value of those rows and including duplicate values, whereas COUNT (<expression>) returns the number of non-NULL values for that expression returned by the query.
- MIN: Returns the minimum value of an expression.
- MAX: Returns the maximum value of an expression.
- SUM: Returns the sum of an expression.
- AVG: Returns the arithmetical mean of an expression.

- `STDDEV`: Returns the standard deviation of a given expression as the square root of the `VARIANCE` function.
- `VAR`: Returns the variance of an expression as the square of a standard deviation.



LESSON SUMMARY

You should now be able to:

- Describe SAP HANA SQL

Query a Modeled Hierarchy Using SQL



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Query a modeled hierarchy using SQL

Query Hierarchies using SQL

A hierarchy that is defined in a calculation view can also be accessed directly by SQL statements. For example, you may need to return the total number of absence days for the Service line of business that can only be found by rolling up many departments and sub-departments in a company hierarchy that is modeled in a calculation view. A simple SQL `select sum(days)` statement with a `where hier_node = 'service'` clause could easily provide this value.

But before you can write the SQL query, you need to work on the following setup:

1. The hierarchy you wish to query must be defined in a **dimension** calculation view, which then must be consumed in the **star join** node of a **cube** calculation view. This is because only **shared** hierarchies can be read by SQL.
2. The calculation view of type cube with star join must allow the shared hierarchy to be exposed to SQL by setting a special parameter in the *Properties*.
3. You should check the name that is given to the node column because you will need to refer to this in SQL.

Reading hierarchies using SQL is supported for both *level* and also *parent-child* types.

Let's look in detail at the settings in the calculation view type cube with star join.

In the *View Properties* tab of the *Semantics* node you need to set the flag *Enable Hierarchies for SQL Access* in order to give permission for this calculation view to expose its shared hierarchies to SQLScript.

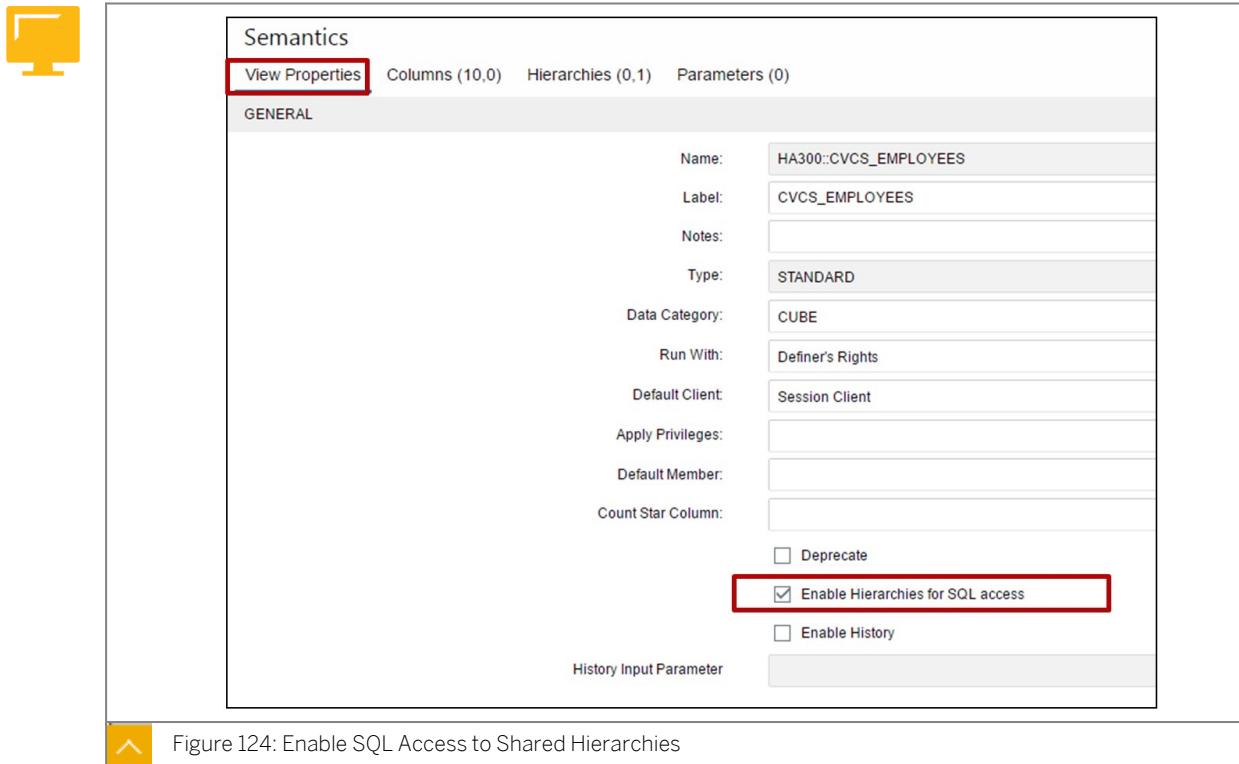


Figure 124: Enable SQL Access to Shared Hierarchies

Once this is set, you can then review the settings in the *Hierarchies* tab of the *Semantics* node.

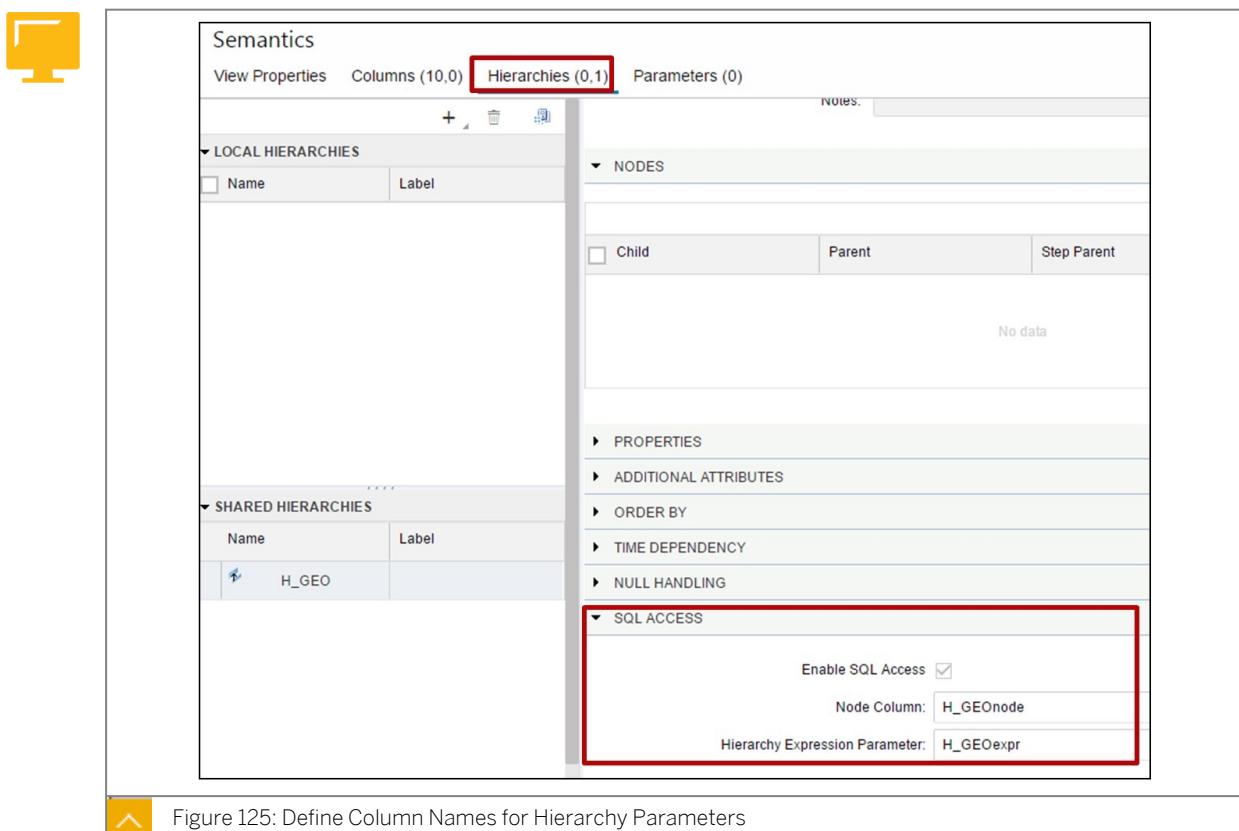


Figure 125: Define Column Names for Hierarchy Parameters

Here you will see, for each shared hierarchy, the node column name that is proposed. This name can be overwritten so that a more meaningful name can be supplied.



Note:

The *Hierarchy Expression Parameter* is not used in the current release of SAP HANA. Please ignore this setting.

Aggregating Values in a Hierarchy Via SQL

The figure, Aggregating Values in a Hierarchy Via SQL, shows how the SQL is written and how the result would appear. Simply refer to the generated hierarchy node column name as if it were a regular column in the data source.



Aggregation in a Hierarchy via SQL

The node column can be used in the GROUP BY clause

SQL		
	SQL	Result
1	select "H_GEONode", sum("SALARY_AMOUNT") from "_SYS_BIC"."STUDENT03.HIER/CVC_EMPLOYEES" group by "H_GEONode"	
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		

H_GEONode	SUM(SALARY_AMOUNT)
1 (all)	3,565,994
2 [DE]	124,618
3 [DE].[Berlin]	124,618
4 [EN]	478,449
5 [EN].[London]	478,449
6 [FR]	0
7 [FR].[Paris La Défense Cedex]	0
8 [US]	2,962,927
9 [US].[Antioch, Illinois]	1,767,002
10 [US].[New York]	619,509
11 [US].[San Francisco]	576,416

The result shows the aggregated amount for each node of the hierarchy

Figure 126: Aggregating Values in a Hierarchy via SQL



LESSON SUMMARY

You should now be able to:

- Query a modeled hierarchy using SQL

Working with SQLScript

LESSON OVERVIEW

This lesson will give you a general introduction to SQLScript. You will learn about the benefits of SQLScript and how you can use it in the context of SAP HANA modeling which includes functions and procedures.

Business Example

You have created graphical calculation views, but you now require additional functionality that can only be found in SQLScript.



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Develop Skills using SQLScript

Why Do We Need SQLScript?

SQLScript is a database language developed by SAP that takes standard SQL, and adds additional capabilities to exploit the features of SAP HANA. The figure, SQLScript: Concept, lists some of the key features of SQLScript.



- **SQLScript** extends ANSI-92 SQL. Just like other database vendors extend SQL in their databases, SAP HANA extends SQL using SQLScript.
- **SQLScript** enables you to access SAP HANA-specific features like column tables, parameterized information views, delta buffers, working with multiple result sets in parallel, built-in currency conversions at database level, fuzzy text searching, spatial data types, and predictive analysis libraries.
- **SQLScript** allows developers to push data intensive logic into the database
- **SQLScript** encourages developers to maintain algorithms using a set-oriented paradigm, instead of a one record at a time paradigm
- **SQLScript** does however allow looping through results and using if-then-else logic
- **SQLScript** allows you to break complex queries into multiple smaller statements, thereby simplifying your SQL coding, and enhancing parallelism via the optimizer.



Figure 127: SQLScript: Concept

Additional features of SQLScript include working with SAP HANA column tables, passing parameters between calculation views and SQLScript, working with spatial data types, and predictive analysis libraries.

SQLScript Extends SQL

In traditional client-server approaches, the business logic is defined and executed in the application server using application programming languages such as ABAP or C++. With SAP HANA, much of this logic can be defined using SQLScript which allows the developer to define and execute all data processing logic in the database in order to achieve improved performance.

SQLScript allows you to use variables to break a large complex SQL statement into smaller, simpler statements. This makes the code much easier to understand. It also helps with SQL HANA's performance because many of these smaller statements can be run in parallel.



Store Interim Results Using SQLScript Variables

```
books_per_publisher = SELECT publisher, COUNT (*) AS num_books FROM BOOKS GROUP BY publisher;
```

```
publisher_with_most_books = SELECT * FROM :books_per_publisher WHERE num_books >= (SELECT MAX (num_books) FROM :books_per_publisher);
```

- Easier to understand
- Re-use the variable multiple times
- Improve parallelization



Figure 128: SQLScript Variables

Let's have a look at an example.

```
books_per_publisher = SELECT publisher, COUNT (*) AS num_books FROM BOOKS GROUP BY publisher;
```

```
publisher_with_most_books = SELECT * FROM :books_per_publisher WHERE num_books >= (SELECT MAX (num_books) FROM :books_per_publisher);
```

You normally write this example as a single SQL statement using a temporary table, or by repeating a sub-query multiple times. In this example, however, you break this into two smaller SQL statements by using table variable. Notice how the table variable does not even need to be declared in advance; its definition is taken from the output result.

The first statement calculates the number of books each publisher has and stores the entire result set into the table variable called `books_per_publisher`.

This variable, containing the entire result set, is used twice in the second statement.

Notice that the table variable is prefixed in SQLScript with a colon (':') to indicate that this is used as an input variable. All output variables just have the name, and all input variables have a colon prefix.

The second statement uses `:books_per_publisher` as inputs, and uses a nested SELECT.

The SQLScript compiler and optimizer determine how to best execute these statements, whether by using a common sub-query expression with database hints or by combining this into a single complex query. The code becomes easier to write and understand, more maintainable, and developer productivity increases.

By breaking the SQLScript into smaller statements and filling table variables, you also mirror the way in which you have learned to build your calculation views. Just like when you start building calculation views in layers, starting from the bottom up, you do the same with your SQLScript code. Keep in mind that the precise sequence of your SQLScript steps is not necessarily the runtime order, because the optimizer will always decide on the best execution plan. However, the optimizer will never alter your desired outcome. This is also true for calculation views.

SQLScript Data Types

SQLScript also adds extra data types to help address missing features from standard SQL; for example, spatial data types and text data types. These data types are listed in the figure, ANSI-92 and SQLScript Data Types.



Classification	Standard ANSI-92 Data Types	SQLScript-specific Data Types
Date/Time	DATE, TIME, TIMESTAMP	SECONDDATE
Numeric	DECIMAL, INTEGER, SMALLINT, FLOAT, REAL, DOUBLE	TINYINT, BIGINT, SMALLDECIMAL
Character string	CHAR, NCHAR, VARCHAR, NVARCHAR	ALPHANUM, SHORTTEXT
Binary	BINARY, VARBINARY	
Large Object		BLOB, CLOB, NCLOB, TEXT, BINTEXT
Boolean		BOOLEAN
Spatial		ST_POINT, ST_GeOMETRY

Figure 129: ANSI-92 and SQLScript Data Types

The Use of Variables

The use of variables in SQL statements is not specified by the ANSI-92 SQL. In SAP HANA, SQLScript implements the concept of variable as an extension to ANSI-92 SQL.

Variables can be declared, assigned, and reused within a script.

Using Variables in SQLScript



- Declare a variable

```
DECLARE <variable_name> <type> [NOT NULL] [= <value>]
```

Example 1: `DECLARE a int;`

Example 2: `DECLARE b int = 0;` (the value is assigned when creating the variable)

- Assign a variable (define the value of a variable)

```
<variable_name> = <value> or <expression>
```

Example 1: `a = 3;`

Example 2: `b = select count(*) from baseTable;`

- Use a variable in an expression

`:<variable_name>` (with a colon) returns the current value of the variable

Example: `b = :a + 5;` (assigns the value of $a + 5$ to b)

**Note:**

These examples are simplified and do not cover the complete SQLScript syntax for variables. You can find more information in the SAP HANA SQLScript Reference Guide; available at <http://help.sap.com/hana>

Processing Input Parameters Defined in Calculation Views with SQLScript

When you use a *select* statement in SQLScript to query a calculation view that has input parameters, you use the *PLACEHOLDER* keyword with the name of the input parameter wrapped in \$\$ symbols, and the value you want to pass to the input parameter. For example:

Using Input Parameters in SQLScript

Use the *PLACEHOLDER* keyword to pass input parameters to the calculation view in a *select* statement:

```
SELECT "PRODUCT", sum("TAXED_AMOUNT")
      FROM "_SYS_BIC"."MyPackage/SalesResults"
        ('PLACEHOLDER' = ('$$Tax_Amount$$', '17.5'))
GROUP BY "PRODUCT"
```

Within SQLScript you refer to the input parameter using a colon::

```
Sales with tax = Sales Revenue * :Tax_Amount
```

It is also possible to pass output parameters generated from an SQLScript, into a calculation view using the parameter mapping feature.

Imperative Logic

In general, SQL code is written in a declarative way. This means you state the **intent** of the action and do not dictate **how** the data processing steps should be carried out in a particular order. We do this so that you allow the SQL optimizer to figure out the best execution plan based on many dynamic factors at run-time.

However, sometimes you need to control the flow logic of the SQL so that steps are performed in a particular order. For example, when you need to perform a step only if a certain condition is met, or you need to loop a number of times. In these cases, you will use imperative logic.

This is (mostly) not available in ANSI SQL, and therefore SAP HANA provides declarative logic language as part of SQLScript to provide flow control.

A lack of imperative logic language is what often causes the developer to move the processing of data to the application layer. When an application runs on HANA, you really should push as much data processing to the database. SAP HANA SQLScript imperative logic enables this so less data handling needs to be done in the application layers and more can be done in the database.

**Hint:**

Remember that imperative logic cannot have the same performance as declarative logic. This is because the SQL optimizer is not free to decide on the sequence of steps and has to take care to follow your logic, which might destroy optimizations. If you require the best performance, try to avoid imperative logic.



```

IF <bool-expr1>
  THEN
    {then-stmts1}
  {ELSEIF <bool-expr2>}
  THEN
    {then-stmts2}
  {ELSE}
    {else-stmts3}
  END IF

```

Example:

```

SELECT count(*) INTO found
FROM books WHERE isbn = :v_isbn;

IF :found IS NULL THEN

  CALL ins_msg_proc
  ('result of count(*) cannot be NULL');

ELSE

  CALL ins_msg_proc
  ('result of count(*) not NULL - as expected');

END IF;

```

Figure 130: The IF Statement

The figure, The IF Statement, shows an example of this statement, which consists of a Boolean expression `bool-expr1`. If this expression evaluates to true, then the statement `then-stmts1` in the mandatory `THEN` block is executed. The `IF` statement ends with `END IF`. The remaining parts are optional.

If the Boolean expression `bool-expr1` does not evaluate to true, then the `ELSE` branch is evaluated. In most cases this branch starts with `ELSE`. The statement `else-stmts3` is executed without further checks. After an `else` branch, no further `ELSE` branch or `ELSEIF` branch is allowed.

Alternatively, when `ELSEIF` is used instead of `ELSE`, another Boolean expression `bool-expr2` is evaluated. If it evaluates to true, the statement `then-stmts2` is executed. You can add an arbitrary number of `ELSEIF` clauses in this way.

This statement can be used to simulate the switch-case statement known from many programming languages.

WHILE and FOR Loops

The figure, WHILE and FOR Loops, shows the syntax of these two loops.



```

WHILE <bool-stmt> DO
  {stmts}
END WHILE

```

```

FOR <loop-var> IN {REVERSE} <start> .. <end> DO
  {stmts}
END FOR

```

Figure 131: WHILE and FOR Loops

The WHILE loop

The WHILE loop executes the statement `stmts` in the body of the loop as long as the Boolean expression at the beginning `bool-stmt` of the loop evaluates as true.

The FOR loop

The FOR loop iterates a range of numeric values – denoted by start and end in the syntax – and binds the current value to a variable (`loop-var`) in ascending order. Iteration starts with value start and is incremented by one until the `loop-var` is larger than end.

Therefore, if start is larger than end, the body loop will not be evaluated. For each enumerated value of the loop variable the statements in the body of the loop are evaluated. The optional keyword REVERSE specifies to iterate the range in descending order.

Dynamic SQL

Sometimes there are elements of the SQL code that cannot be known until run-time. For example: the name of a table or a column may not be known until a session variable is filled, or a look-up is performed on current data. In that case, dynamic SQL can be used.

The EXEC Statement

The EXEC statement helps to create dynamic SQL statements. For example, let's say that you want to find the travel arrangements for users. A web form is requesting the user's passport number. You insert this passport number into a partially prepared SQL statement, and then execute this SQL statement using the EXEC statement to get the travel plans.

```
EXEC '<SQL statement goes here>'
```

Dynamic SQL statements are used to construct SQL statements at runtime in a procedure.

The EXEC statement executes the SQL statement passed in a string argument. This statement allows for dynamically constructing an SQL statement at execution time of a procedure.

For example, a SELECT statement could reference a table whose name is not known until runtime of the procedure. Perhaps the table name is retrieved from a lookup based on a condition.

Therefore, on the positive side, dynamic SQL allows the use of variables where they might not be supported in SQLScript, or more flexibility in creating SQL statements.

On the negative side, dynamic SQL comes with the following additional costs at run-time:

- Opportunities for optimizations are limited.
- The statement is potentially recompiled every time the statement is executed.
- It is not possible to use SQLScript variables in the SQL statement (but when constructing the SQL statement string).
- It is not possible to bind the result of a dynamic SQL statement to a SQLScript variable.
- SQL injection bugs could harm the integrity or security of the database.



Note:

SAP HANA SPS11 introduced features to secure Dynamic SQL. However, we recommend, in general, that you avoid dynamic SQL because it might have a negative impact on security or performance.

Security of Dynamic SQL

Built-In Procedures to Secure Dynamic SQL

For variables containing an SQL string literal, use the following: `ESCAPE_SINGLE_QUOTES (string_var)`

For variables containing a delimited SQL identifier, use the following:

`ESCAPE_DOUBLE_QUOTES (string_var)`

To check that a variable contains safe, simple SQL identifiers (up to num_tokens, where the default is 1), use the following: `IS_SQL_INJECTION_SAFE(string_var[, num_tokens])`

You can find details on these procedures in the SAP HANA SQLScript Reference Guide; available at <http://help.sap.com/hana>.



LESSON SUMMARY

You should now be able to:

- Develop Skills using SQLScript

Creating and Using Functions



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Work with functions

Introducing Functions

Functions, or User Defined Functions (UDF), as they are more formally known, allow developers to capture a reusable block of SQLScript code that provides a result after working through some data processing logic. Often, functions allow input parameters to pass key information to the processing logic so that the function can be reused in different scenarios.

Functions are relevant to SAP HANA modeling because they offer a way to extend the standard capabilities provided with graphical calculation views to reach more complex data processing possibilities. Functions are written in SQLScript and are built using design-time files. After a successful build, a runtime object is generated in the SAP HANA database. When executed, functions always run in the in-memory SAP HANA database, and so performance is optimized.



Note:

Functions are relevant for classic repository or XS development and also the newer HDI or XSA development. In this lesson we focus on the HDI or XSA approach.

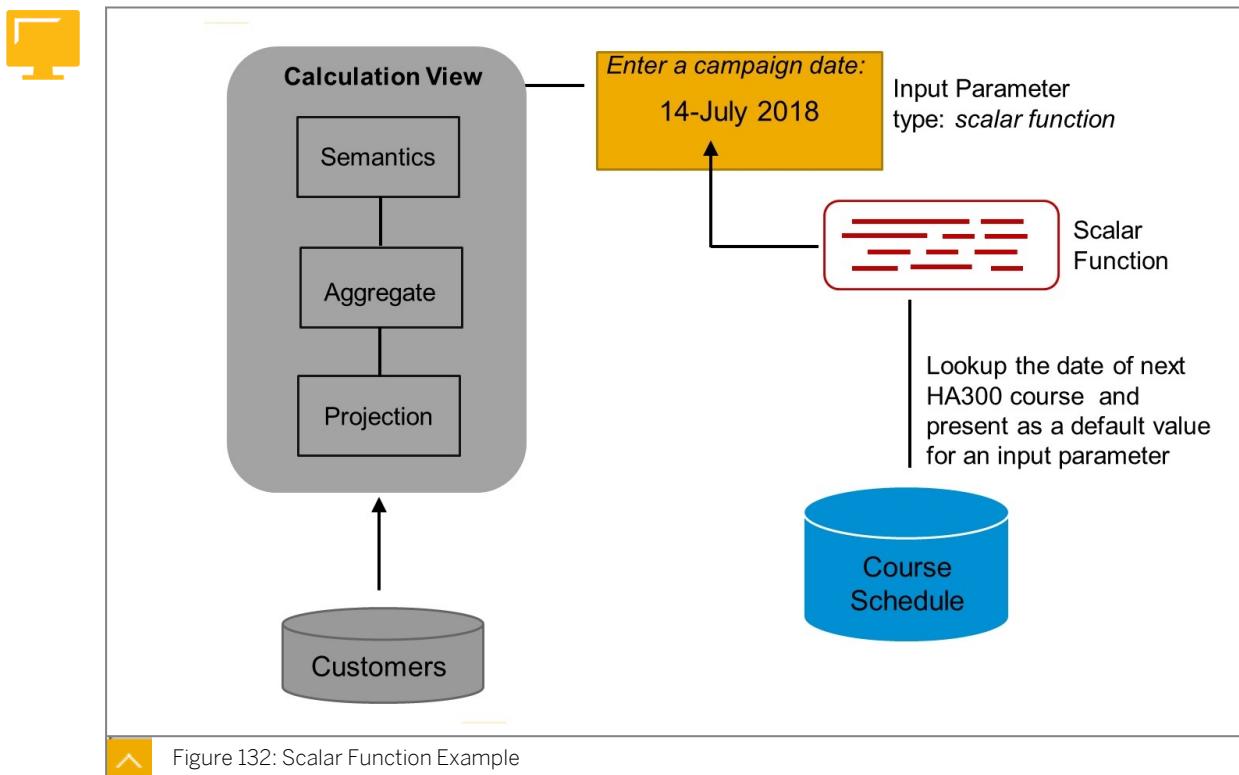
SAP HANA supports **scalar** and **table** functions. Table and scalar functions are created as design-time objects in an SAP Web IDE project folder. The design-time object has the extension **.hdbfunction** and the same extension is used for table and scalar functions.

Functions, both scalar and table, are always **read-only**. It is not possible to alter the database by using data definition statements such as *create table*, or data manipulation statements such as *update*. If you need to do such things, then consider the use of *Procedures*.

Functions can call other functions which encourages a modular approach to ensure high reuse.

Scalar Function

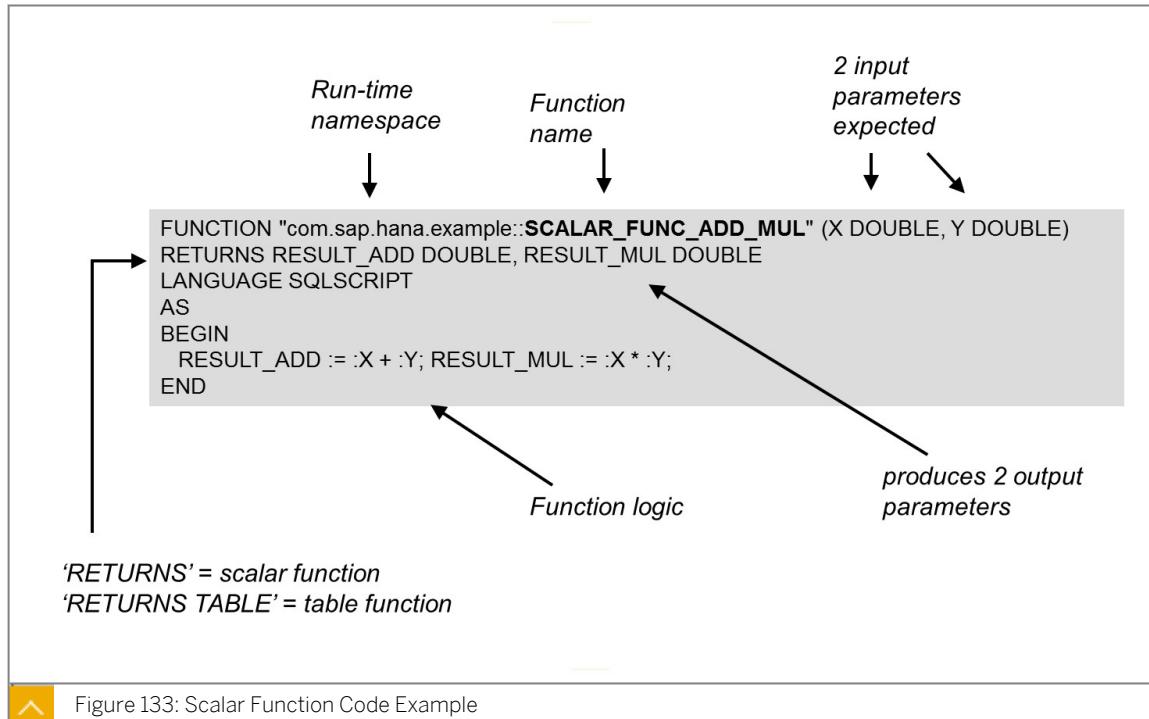
One of the common uses of a scalar function in a calculation view, is to derive values for input parameters, either as a default proposal in a popup that can be overwritten by a user, or as a fixed value that cannot be changed by a user.



A simple example of a scalar function could be to return the current date. However, scalar functions can be more complex and can read tables and call other database objects (including other user defined functions).

A scalar function returns one or more parameters but each parameter always has only a single value. Scalar functions often have one or more input values, but these are not mandatory. For example, returning the current date requires no input parameter, whereas if you wanted to return the date that was x days earlier than today, then you would need to provide x as the input parameter.

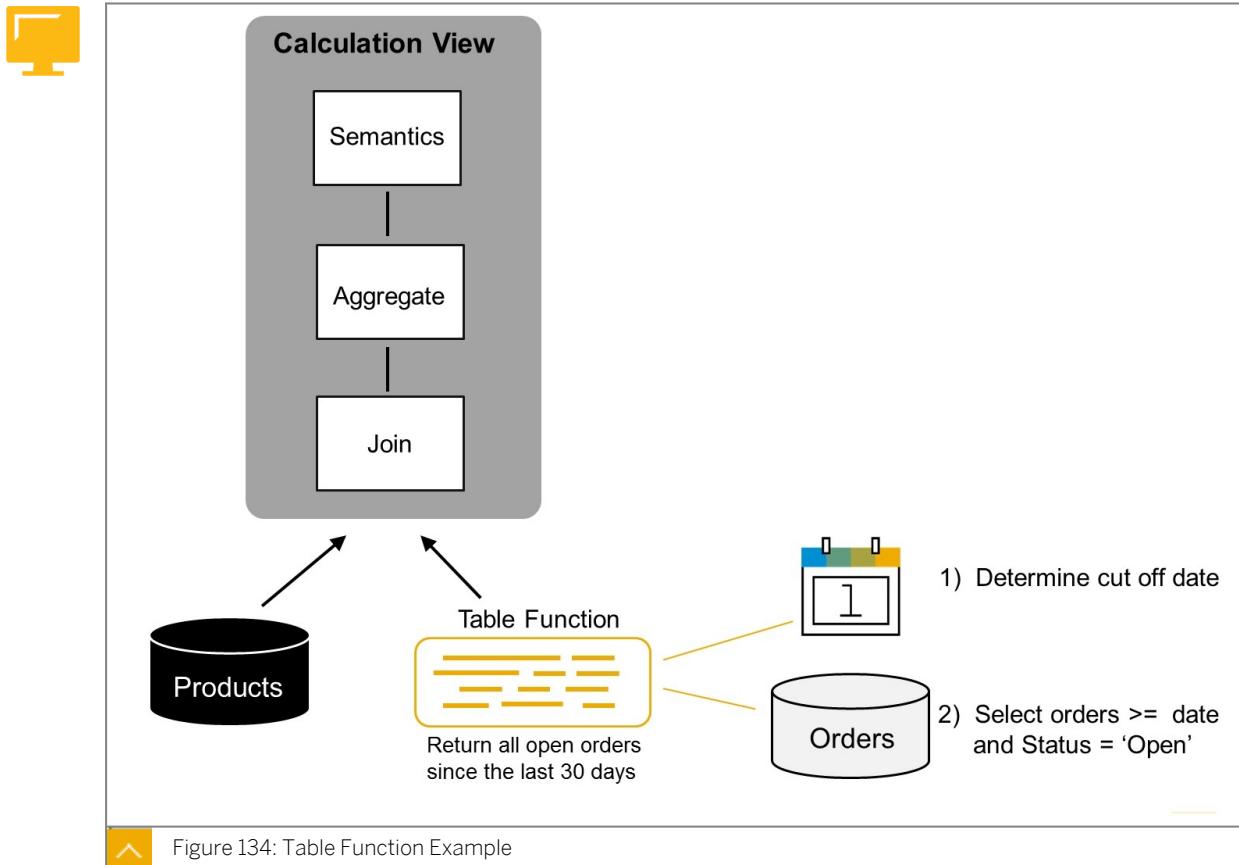
The following is a breakdown of a scalar function's code:



Notice the use of the keyword **RETURNS** for scalar functions. The keyword **RETURNS TABLE** is used for table functions.

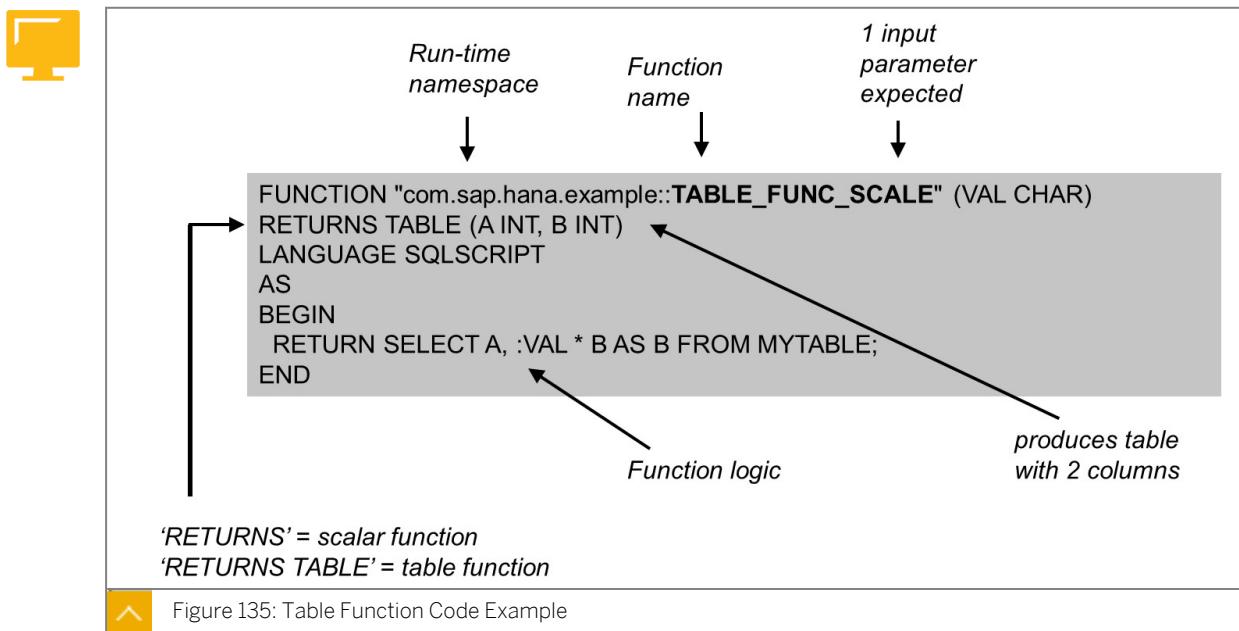
Table Functions

Table functions are used to return a **tabular** data set as opposed to a single value parameter. A tabular data set can consist of one or more columns and usually has multiple values (rows). Table functions are commonly used by modelers to define a customized data source in a calculation view. Refer to the function wherever a data source can be defined, such as in a calculation view *Projection* node or *Join* node just as you would include a table. In fact, it might help to think of them as dynamic tables that are created at run time.



As an example of a table function, consider that you may need to read through a table that contains sales orders so that you can find open orders that have been recently created. The date used for the determination of the required orders is based on an input parameter passed from the calculation view. The resulting open orders are passed to the calculation views as a data source.

The following is a breakdown table function's code:



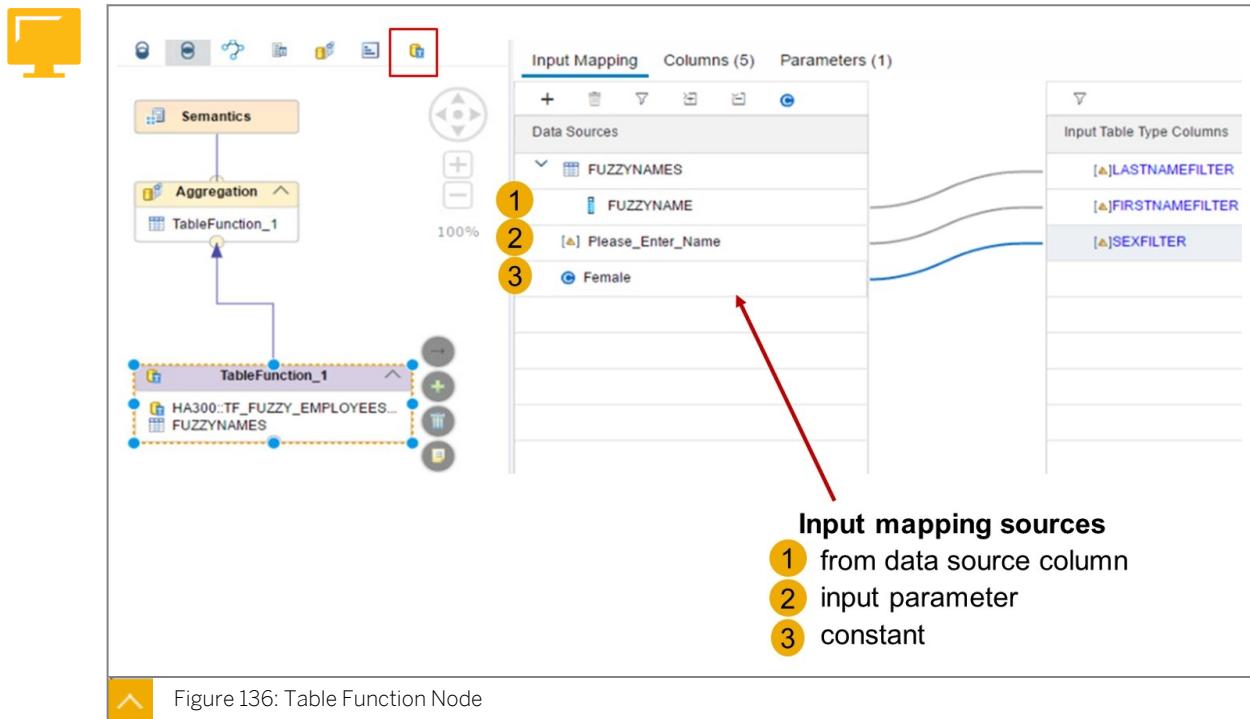
Notice the use of the keyword **RETURNS TABLE** for table functions. The keyword **RETURNS** is used for scalar functions.

Note:

Introduced in the first release of SAP HANA, there was a type of calculation view that could be built using 100% SQLScript instead of using the graphical approach. They were called *Scripted Calculation Views*. It is still possible to create these using SAP HANA Studio under the classic/repository framework. However, SAP no longer recommends these because they do not provide the best performance optimizations. You should avoid these and instead build your SQLScript into functions and then consume the functions in graphical calculation views. SAP provides a migration tool in SAP HANA Studio to convert scripted calculation views to functions, and a wrapper graphical calculation view is generated to consume the functions. Thus, the outcome is the same.

All output parameters of the table function are offered as columns to the calculation view node. If your table function requires input parameters you can first define input parameters in the calculation view and then use the input parameter mapping function to map them.

Since SAP HANA 2.0 SPS01, a new calculation view node is available: **Table Function**. It is specially dedicated to the handling of the different types of mapping of the input parameters in a table function. It was possible to achieve the same outcome using other node types, but this new node makes things easier to define the parameter mapping.



There are three types of input mapping available in the *Table Function* node. They are as follows:

- **Data Source Column**

First, add a data source to the Table Function node. Then, map one of its columns to one or more table function input parameters.

- **Input Parameter**

Choose an input parameter from the calculation view and map it to one or more table function input parameters.

- **Constant**

Manually define a single fixed value and map it to one or more table function input parameters.



LESSON SUMMARY

You should now be able to:

- Work with functions

Creating and Using Procedures

LESSON OVERVIEW

In this lesson, you will get an introduction to procedures and learn how to define a procedure and how to call the procedure.

Business Example

As part of your SAP HANA implementation project, you have some business requirements for reporting that cannot be fulfilled with only analytic or graphical calculation views.

You have decided to use SQL Calculation Views, but you would like to get the best possible flexibility by creating procedures that you can easily reuse in several calculation views.

You want to know more about procedures and how to use them in the most relevant way.



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Create and use procedures

Introduction to Procedures

A procedure is a reusable block of code that defines data processing logic. A procedure can optionally include input parameters which can be of the type scalar or table. Procedures can have one or more output parameters which can be of the type scalar or table . The language used is typically SQLScript, but other languages, such as R, are possible.

Procedures (often referred to as database procedures to differentiate them from application-language procedures) are a popular artifact with application developers who need to lock in database processing logic into a modular design. With SAP HANA, application developers are encouraged to move their data processing code down to the database. Using the powerful SQLScript language of SAP HANA it is possible for developers to write complex logic that would previously have been written in the application layer using an application language such as ABAP.

Procedures are also used in modeling when more flexibility is needed to be able to define complex data processing logic that can only be expressed in SQLScript.

There are two main places where procedures can be used in SAP HANA modeling. They are as follows:

- Input Parameters
- Analytic Privileges



- Reusable block of data processing logic
- Add complex processing logic to calculation views
- Determine dynamic analytic privileges
- Read and or write capable
- Can contain DDL statements
- One or more inputs (table or scalar types)
- One or more outputs (table or scalar types)
- Written in SQLScript, R or L
- Procedures can call other procedures or functions

```

PROCEDURE "HA300::P_BOOKS_VAT_00"
(IN tax DECIMAL(5,2),
OUT out_with_tax table
(ISBN VARCHAR(20),
TITLE VARCHAR(50),
PUBLISHER INT,
PRICE DECIMAL(33,2),
PRICE_VAT DECIMAL(33,2),
CURRENCY VARCHAR(3)
)
)
LANGUAGE SQLSCRIPT
SQL SECURITY DEFINER
READS SQL DATA AS

/********** Begin Procedure Script *****/
BEGIN

out_with_tax = SELECT ISBN,
TITLE,
PUBLISHER,
PRICE,
PRICE * :tax AS PRICE_VAT,
CRCY AS CURRENCY
FROM "HA300::BOOKS";
END

```



Figure 137: Procedures – Key Features

Procedures and functions (table functions and scalar functions) have broadly similar features and achieve similar aims. They can both be written in SQLScript and are used to create reusable blocks of complex processing logic, receiving input parameters and producing output parameters.

So what are the main differences between procedures and functions, and why do we need both in modeling?

A key difference is how they are invoked. This involves the following:

- You can define a table function as a data source in a calculation view, just like a table, but you cannot add a procedure as a data source.
- You can use a table function in a FROM clause of a SELECT statement, but not a procedure.
- You can include a scalar function in the column list of a SELECT or WHERE statement, but not a procedure.
- You call a procedure using a CALL statement in SQLScript, but you do not CALL a function.
- A procedure can be used to return values for input parameters in a calculation view but so too can a function (but only a scalar function).
- A procedure can be used to return the data access permissions used in dynamic analytic privileges, but a function cannot be used to achieve the same.

The following are other important differences:

- A powerful feature of a procedure is its ability to write data to the database as well as read it. Functions cannot write data, they can only read data. This is often referred to as "side-effect free". This simply means that the database does not change from the use of a function.
- Procedures can contain SQL statements that are forbidden in functions, for example, data definition language such as CREATE TABLE, TRUNCATE, ALTER and so on.

- Procedures can produce multiple tabular output structures containing different mixtures of columns. A function can only produce one output structure.
- Procedures support multiple languages including SQLScript, R and L ('L' is an SAP internal language not released to customers). Functions can only be written in SQLScript.

Procedures can be called from table functions as long as they are read-only procedures.



Note:

In general terms, data modelers are more likely to be working with scalar and table functions than procedures. This is because the primary data modeling object of SAP HANA is the calculation view and calculations views make more use of functions than procedures. Procedures are more likely to be used by application developers who need to write as well as read data and are not limited to developing code for data modeling purposes only but may also be creating code for application development and general administration, such as creating routines that grant and revoke privileges or call data flow-graphs to load data to SAP HANA.

Procedures in Calculation Views

Procedures can be used to derive results for input parameters. For example, you could define a procedure to determine the date of the next campaign or to calculate the number of target days to use in a calculation.

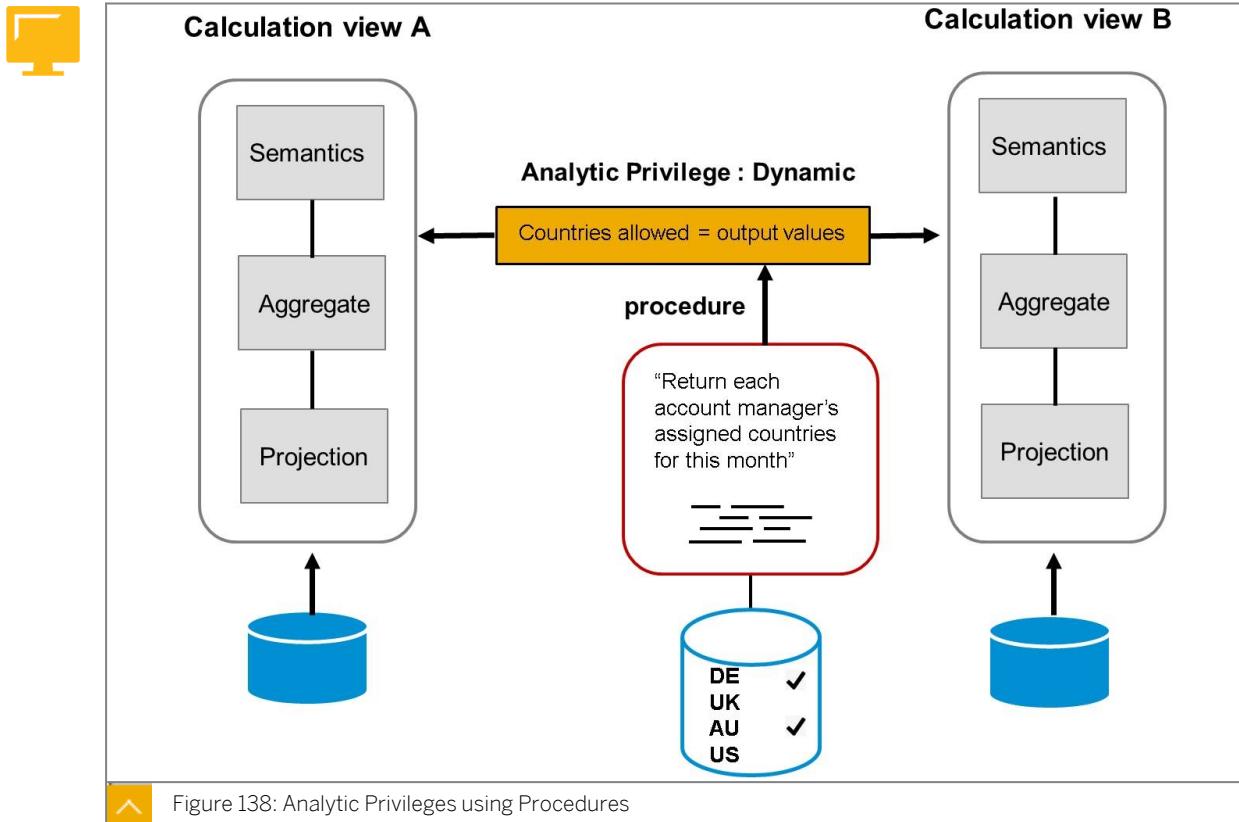


Note:

It is also possible to define a scalar function to return the values for input parameters.

Procedures in Analytic Privileges

Analytic privileges are a type of security artifact used to define data access conditions for calculation views.



When you define an analytic privilege you must decide whether you will hard-code the allowed values of attributes, or whether you will use a procedure to dynamically derive the allowed values.

A procedure is used to provide the logic that determines the allowed values at run-time. A typical example would be to evaluate the user's id or user's role when a calculation view is queried to determine the current allowed value that should be displayed. This means only one analytic privilege needs to be created that will be used by many users to determine automatically the countries allowed to be displayed. This is ideal when access values change often and you want to avoid a lot of maintenance of hard-coded analytic privileges.

Writing a Procedure



P_BOOKS_VAT_00.hdbproced... ◀ Source file name

```
1 PROCEDURE "HA300::P_BOOKS_VAT_00"  
2 (IN tax DECIMAL(5,2),  
3 OUT out_with_tax table  
4 (ISBN VARCHAR (20),  
5 TITLE VARCHAR (50),  
6 PUBLISHER INT,  
7 PRICE DECIMAL (33,2),  
8 PRICE_VAT DECIMAL (33,2),  
9 CURRENCY VARCHAR (3)  
10 )  
11 )  
12 LANGUAGE SQLSCRIPT  
13 SQL SECURITY DEFINER  
14 READS SQL DATA AS  
15  
16 /****** Begin Procedure Script *****/  
17 BEGIN  
18  
19 out_with_tax = SELECT ISBN,  
20 | TITLE,  
21 | PUBLISHER,  
22 | PRICE,  
23 | PRICE * :tax AS PRICE_VAT,  
24 | CRCV AS CURRENCY  
25 FROM "HA300::BOOKS";  
26  
27  
28 END;  
29 /****** End Procedure Script *****/
```

Run time name

Input

Output

Language

Use permissions of definer, not invoker

Read only

SQLScript

Figure 139: Example of a Procedure

You create a procedure by writing your SQLScript code in a dedicated source file in SAP Web IDE, which has the extension **.hdbprocedure**. You then build it to create the runtime object in the HDI container. It is also possible to create procedures directly using the `CREATE PROCEDURE` statement in the SQL console. However, this is not recommended because this does not follow the HDI approach.

Testing a Procedure

During development of a procedure, it is helpful to test it using the SQL console. You can either write a simple 'CALL' statement in the SQL console of Web IDE, specify the procedure, and manually add input parameters. Or from the development view of Web IDE, you can right-click the source file of the procedure and directly launch it in the Database Explorer so that you don't even need to know how to write the CALL statement, it is written for you. You just fill in the parameters using the placeholders and hit F8.



Note:

Functions and procedures and advanced SQLScript topics are covered in detail in the course HA150 – SOLScript for SAP HANA.



LESSON SUMMARY

You should now be able to:

- Create and use procedures

Learning Assessment

1. Why is knowledge of SQL important to an SAP HANA modeler?

Choose the correct answers.

- A So that they can choose to create calculation views using only SQL language
- B So that they can understand how to implement a function or procedure and consume this within a calculation view
- C So that they can extend the capabilities of a calculation view using SQL expressions

2. You are using the SQL console of the Web IDE. When are schema references not required in your SQL?

Choose the correct answer.

- A When writing SQL against a database connection of the type HDI Container
- B When writing SQL against a database connection of the type SAP HANA Database.

3. Why should I define my SAP HANA tables using source files rather than using the SQL statement 'CREATE TABLE' in the SQL Console of Web IDE?

Choose the correct answer.

- A The syntax is simpler
- B Performance of the table is better.
- C The table definition is easily transported as part of the overall application to ensure all project artifacts are kept together.

4. Which are valid source file types used to define the persistence layer of SAP HANA?

Choose the correct answers.

- A .hdbtable
- B .sqltable
- C .hdbdropcreatetable

5. What types of user-defined SQL functions can you include as a data source in a calculation view?

Choose the correct answers.

- A Table Functions
- B Column Engine Functions
- C Scalar Functions

6. Why would you use SQL functions when modeling in calculation views?

Choose the correct answer.

- A As an alternative to developing a graphical calculation view when you prefer to use script
- B When you need to write results back to a table from a calculation view
- C When you need to define custom data processing logic that the graphical calculation view cannot provide

Learning Assessment - Answers

1. Why is knowledge of SQL important to an SAP HANA modeler?

Choose the correct answers.

- A So that they can choose to create calculation views using only SQL language
- B So that they can understand how to implement a function or procedure and consume this within a calculation view
- C So that they can extend the capabilities of a calculation view using SQL expressions

Correct — SQL is important for modelers because they need to know how to read the logic written in SQL inside functions and procedures and also how they might extend the calculation view capabilities, for example, by writing expressions that call standard SQL functions.

2. You are using the SQL console of the Web IDE. When are schema references not required in your SQL?

Choose the correct answer.

- A When writing SQL against a database connection of the type HDI Container
- B When writing SQL against a database connection of the type SAP HANA Database.

Correct — You never specify a schema when writing SQL against a HDI Container type database connection.

3. Why should I define my SAP HANA tables using source files rather than using the SQL statement 'CREATE TABLE' in the SQL Console of Web IDE?

Choose the correct answer.

- A The syntax is simpler
- B Performance of the table is better.
- C The table definition is easily transported as part of the overall application to ensure all project artifacts are kept together.

Correct — Source file syntax is not simpler than the standard SQL 'Create Table' statement. A table that is created with source files does not perform better than one created with the standard SQL 'Create Table' statement. The table definition is easily transported as part of the overall application to ensure all artifacts are kept together.

4. Which are valid source file types used to define the persistence layer of SAP HANA?

Choose the correct answers.

- A .hdbtable
- B .sqltable
- C .hdbdropcreatetable

Correct — Valid source types are .hdbtable and .hdbdropcreatetable.

5. What types of user-defined SQL functions can you include as a data source in a calculation view?

Choose the correct answers.

- A Table Functions
- B Column Engine Functions
- C Scalar Functions

Correct - Table functions are supported as data sources in SAP HANA calculation views.

6. Why would you use SQL functions when modeling in calculation views?

Choose the correct answer.

- A As an alternative to developing a graphical calculation view when you prefer to use script
- B When you need to write results back to a table from a calculation view
- C When you need to define custom data processing logic that the graphical calculation view cannot provide

Correct — Functions are not a replacement for graphical calculation views to provide more flexibility. Functions are read only and do not allow writing to the database. Functions allow you to write custom data processing logic that the graphical calculation view cannot provide.

Lesson 1

Defining the Persistence Layer

215

Lesson 2

Loading Data into Tables

223

Lesson 3

Accessing Remote Data

227

UNIT OBJECTIVES

- Understand Why We Need a Persistence Layer
- Define the Persistence Layer
- Load a Table Using SQL Statements
- Load a Table Using Project Source Files
- Load a Table with Data from a Local File
- Explain Virtual Tables

Defining the Persistence Layer



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Understand Why We Need a Persistence Layer
- Define the Persistence Layer

What is the Persistence Layer?

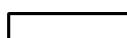
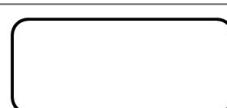
A data model consists of many layers. Layers can either generate data on the fly - these are referred to as virtual layers - or they can permanently store data - these are referred to as persistent layers.

The virtual layers of the SAP HANA modeling stack are built using calculation views and the data is generated at query run-time. The persistence layer of the SAP HANA modeling stack is built using database tables.

Positionally, the persistence layer sits at the bottom of the modeling stack. The final layer (top) of the stack is the consumption layer where we find the applications that consume the virtual layer.



consumption layer (apps)



persistence layer (tables)

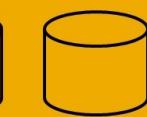
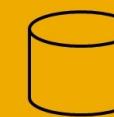
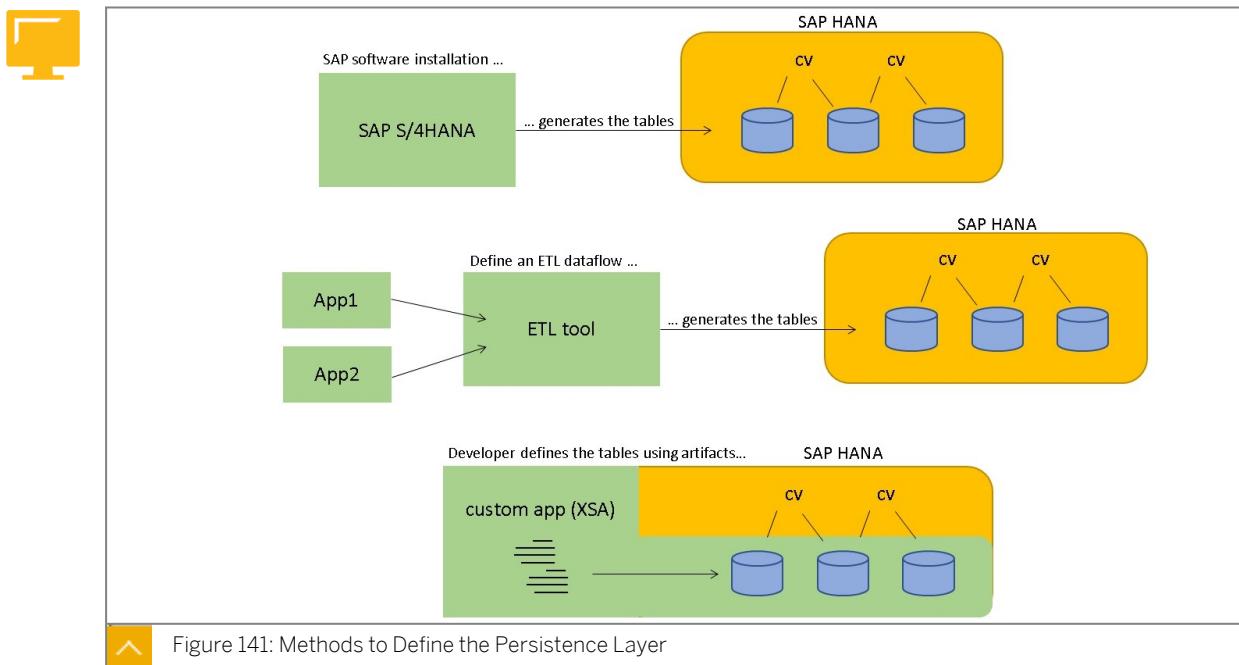


Figure 140: Modeling Layers

This course is aimed at modelers and focuses on the development of the virtual layer using calculation views. But we should take time to consider how the persistence layer is built.



A modeler is usually not responsible for the creation and on-going maintenance of the persistence layer. This is often taken care of by database administrators or application developers who might provide the tables when they are developing the application code. Another possibility is that the tables are provided by the tools that are used to extract, transform and load (ELT) data to SAP HANA, such as SAP Data Services. For SAP applications such as SAP S/4HANA and SAP BW/4HANA, the tables that support those solutions are actually created during the software installation process.

But it is also possible to create tables and even load them with data by defining source code artifacts within an SAP HANA project. SAP provide native artifacts for table creation and also data loading.



Note:

The persistence layer is sometimes located outside of SAP HANA. It is possible to consume data in the tables of external systems by defining a special type of SAP HANA table called a *virtual table*. We will cover that topic later. For now we will focus on standard tables where the data is located inside the SAP HANA database.

Define the Persistence Layer in SAP HANA

Table Types in SAP HANA

The figure, Create Table – Table Types, lists the types of table that you can create in SAP HANA.



COLUMN	COLUMN-based storage should be used if the majority of access is through a large number of tuples but with only a few selected attributes.
ROW	ROW-based storage is preferable if the majority of access involves selecting a few records and with all attributes selected.
SYSTEM-VERSIONED TABLE *	Pair of column tables designed to store previous versions of a table's records to allow retrieval of historical values based on a specific system time-stamp.
APPLICATION TIME PERIOD TABLE *	Pair of column tables designed to store previous versions of a table's records to allow retrieval of historical values based on a specific date/time (NOT necessarily the system's timestamp).
GLOBAL TEMPORARY	Table definition is globally available while data is visible only to the current session. The table is visible in the catalog and is dropped at the end of the session.
LOCAL TEMPORARY	The table definition and data is visible only to the current session. The table is not visible in the catalog and is dropped at the end of the session. Table name must begin with the hash (#) symbol.
VIRTUAL TABLE	Creates an empty table in SAP HANA that reads data from a remote data source when accessed. Requires a remote connection to be configured before the table can be created.

* The former HISTORY COLUMN table type is deprecated.

Figure 142: Create Table – Table Types

The default table type is column. You do not have to specify the type explicitly. If you require a row table then you must specify this explicitly.

Column tables are optimized for very fast read access, especially when large amounts of data need to be aggregated. Within the modeling scenario, column tables are the most popular of all table types.

Row tables are optimized for individual record-level processing. Row tables are efficient for transactional application that require access to all columns of individual records, but they are not optimized for fast read access.

To allow time-travel queries, you can use two temporal properties of COLUMN tables (not supported in ROW tables):

- **System-Versioned Table**, where historical data is stored based on system time-stamps
- **Application-Time Period Tables**, where historical data is stored based on application-specific time periods

In both cases, instead of just updating a table's record when you edit it and thereby losing the previous values, SAP HANA keeps the previous values in a corresponding history table. This allows you to go back in time and see what the table looked like at a specific point in the past. In future, you can expect to see many more features in the business systems where SAP will leverage this functionality; for example, for slowly changing dimensions or month-end processes.



Note:
The former table type *HISTORY COLUMN* is kept for backward compatibility purposes, but is now deprecated and replaced with system-versioned and application-time period tables.

A global temporary table definition is globally available but data that is created by a session is visible only to that session. Data for a session is automatically dropped when the session is ended. A global temporary table can be created based on column or row storage.

A local temporary table definition and also the data is only visible to the current session. The table is automatically dropped when the session is ended. A local temporary table can be created based on column or row storage.

A virtual table is defined by referring to a remote table or view that is located in an external database.

Approaches to Creating Tables in SAP HANA

There are two approaches for creating tables directly in the SAP HANA database:

1. Execute `create table` SQL statements in the SQL console of Web IDE.
2. Describe the table in a source file in your XS Advanced project, then build the HDB module which creates the table.



Two approaches to creating tables

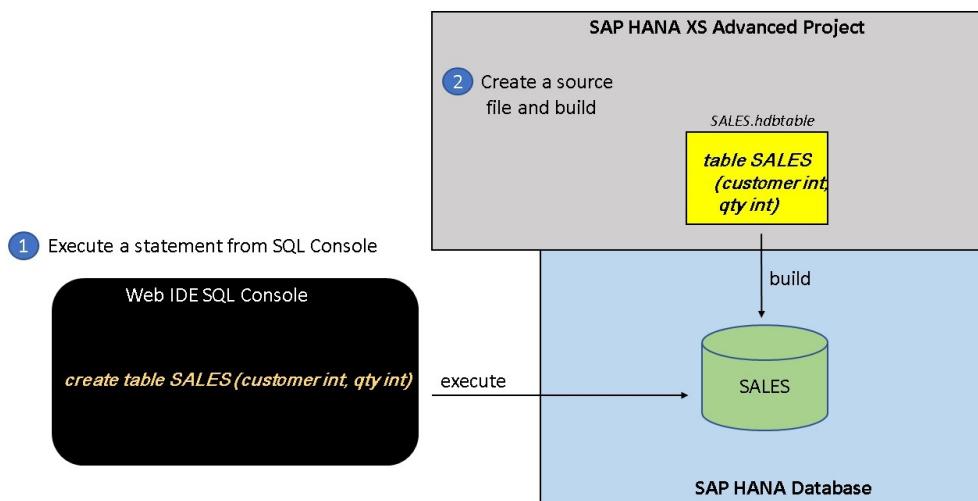


Figure 143: Approaches to Creating Tables In SAP HANA

Create Tables Using the SQL Console

To create a table using a standard SQL statement in the SQL Console, open the SQL Console connected to either a HDI container or a classic catalog connection and type: `create table <tablename> (<col1> <type>, <col1> <type>)`

The default table type in SAP HANA is column store. If you prefer a row store table, simply use the `create row table` statement.



SQL

```

1  -- Preparation
2  -- Replace ## with your group number (e.g. STUDENT01)
3  SET SCHEMA STUDENT##;
4
5
6  -- STEP 1
7  drop table publishers;
8
9  CREATE COLUMN TABLE publishers
10  (
11      pub_id INTEGER PRIMARY KEY,
12      name VARCHAR (50),
13      street VARCHAR (50),
14      post_code VARCHAR (10),
15      city VARCHAR (50),
16      country VARCHAR (50));
17
18  drop table "publishers";
19
20  CREATE COLUMN TABLE "publishers"
21  (
22      pub_id INTEGER PRIMARY KEY,
23      name VARCHAR (50),
24      street VARCHAR (50),
25      post_code VARCHAR (10),
26      city VARCHAR (50),
27      country VARCHAR (50));

```

Table Elements

Table Type

Column Constraint

Column Definition

Data Type

Figure 144: Create Table – Syntax Elements

This approach will be familiar to many developers, however for SAP HANA XS Advanced projects, using the SQL Console to execute statements that create tables is the wrong approach.

Tables that are created directly in the SQL Console have no source code representation. This means the table definition is missing from the source code of your project. When the project is migrated and re-deployed in another target system, the tables would be missing. In this case you would have to manually re-execute the SQL statements in the target system to re-create the tables. This is clearly not efficient as the re-execution of the SQL statements might lead to inconsistencies. Even if you store the statements in text files, the files might get separated from the project and cause version issues.

Another major issue with this approach, is that the development artifacts in your project, such as a calculation view, cannot directly access a table that is created in the SQL Console. This is because there is no source file that provides the table definition in your project. All tables that are referenced by your development artifacts must have a source file that represents them. If there is no source file that provides the table, the calculation view build will fail.

So We Should Never Use the SQL Console Approach to Create a Table?

There are times when using the SQL Console to create tables is acceptable. For example, when you need a temporary table for testing some SQL or SQLScript code. In this case, creating a project, then a database module and then the source files to define the table and then fill it, would be cumbersome. Using a simple `create table` statement and then an `insert into` statement to load some records using the SQL Console is quick and easy. Plus, you can easily drop the table using the SQL `drop` statement when you are finished. You might also want to test a cross-schema scenario where tables are located outside of your container. In this case you can use the SQL Console to easily create a new schema and some tables, and then define a cross-schema service to access those tables from your calculation view.

Create Tables Using Source Files

Create a Table Using a Source File Development Artifacts

We recommend that all tables that are part of an XS Advanced project are defined directly in SAP HANA database using source files, otherwise known as development artifacts. We do not recommend using the SQL Console.

Persistence objects should be part of an SAP Web IDE development project so that they are tied closely with other objects in the same project. For example, we should define the tables, calculations views, functions and all other development artifacts in a single project. By doing this we ensure better integrity as all dependent objects are checked and built together as a single unit. HDI works on the principle of *all or nothing*. That means if even one object is broken, then nothing is built, including all the valid objects.

There are multiple development artifacts to choose from to create a table:

- .hdbtable
- .hdbdropcreatetable
- .hdbmigrationtable
- .hdcds

To delete a table defined using the provided source files, you should not directly use the DROP TABLE statement in the SQL Console. Instead, you should delete the source file. When you next build the folder where the source file existed, the run-time object (the table) will be dropped from the database.



Note:

It is good practice to re-build the project folder soon after you delete source files from it, so the run-time objects are dropped. Be careful not to delete the folder before you re-build it, because then you may lose track of the run-time objects that exist and do not have a source file. If that happened you should build the HDB module so all folders are part of the build.

Defining Tables using .hdbtable

The .HDBTABLE file format uses a well-known DDL-style syntax, which is equivalent to the SQL syntax in the *CREATE TABLE* SQL command, but without the leading *create* keyword. If you change the table definition the data is automatically migrated to the new table. For very large tables this can be costly.

Defining Tables using .hdbdropcreatetable

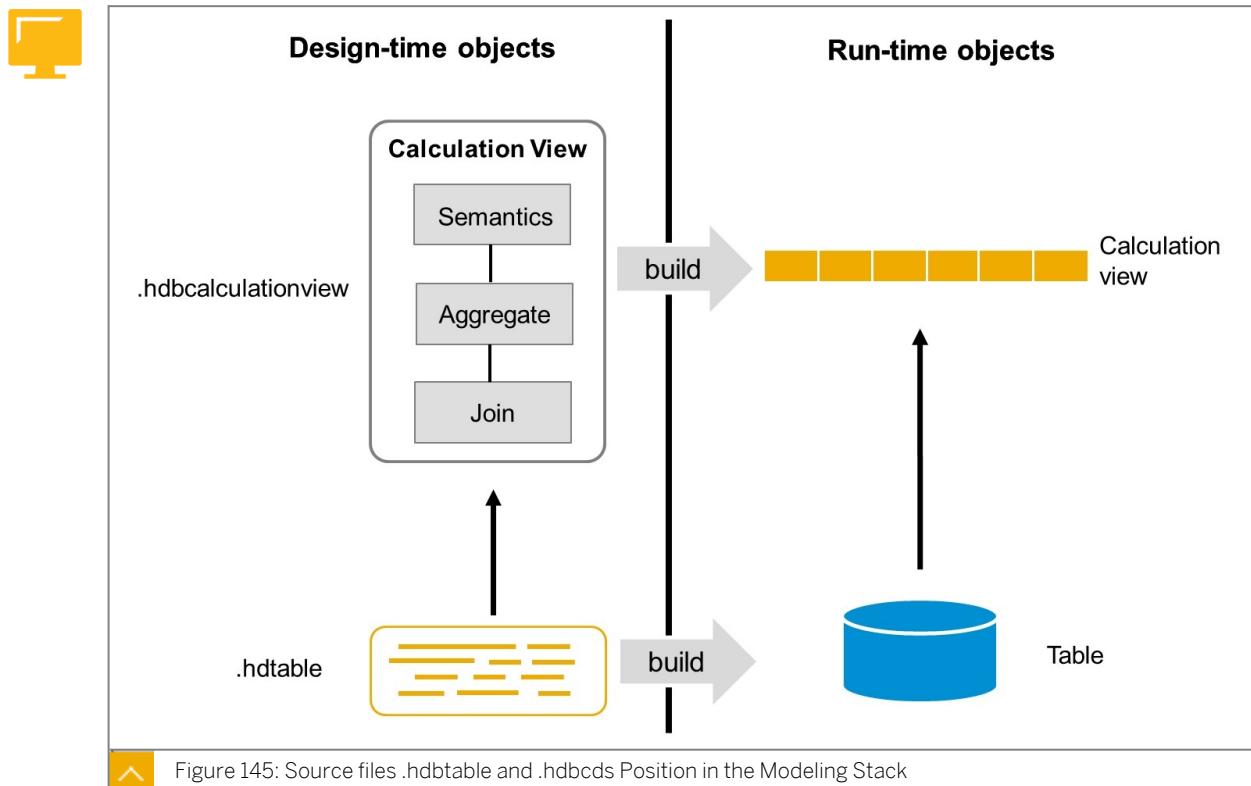
The drop-create-table format does not provide any data migration. If you change a table definition, the build will drop an already deployed version of the table and then recreate the table. If the table contains data, the build is not allowed, since the data would be lost. You first have to delete the data in the table before the build.

Defining Tables using .hdbmigrationtable

In contrast to the .hdbtable, the migration-table format .hdbmigrationtable uses explicit versioning and migration tasks. This means that the modifications of the database table are explicitly specified by the developer in the source file using *alter* statements and are carried out on the database table exactly as specified in the statements, without incurring the cost of

an internal table-copy operation such as the one performed by the .hdbtable format. This behavior makes the .hdbmigrationtable especially useful for tables that contain a lot of data. When a new version of an already existing table is deployed, the .hdbmigrationtable file performs the migration statements that are missing for the new version.

Persistence Definition Source Files in the Modeling Stack



The source files that provide the data persistency layer must be created **before** you begin working on the source files that define the modeling layer. This is because the modeling layer usually refers to the objects of the persistency layer.

For example, a calculation view requires tables to provide the data sources. Those tables must already be provided in source files . If you try to execute a build of a calculation view, and the tables that are being referred to do not have source files that describe them, the build will fail.



LESSON SUMMARY

You should now be able to:

- Understand Why We Need a Persistence Layer
- Define the Persistence Layer

Loading Data into Tables



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Load a Table Using SQL Statements
- Load a Table Using Project Source Files
- Load a Table with Data from a Local File

Load a Table Using SQL Statements

One of the easiest and quickest ways to add records to a table is to execute the *INSERT INTO* statement in the SQL Console.



```
INSERT INTO <table_name> [ ( column_name, ... ) ]
    { VALUES (expr, ... ) | <subquery> };
```

```
15@insert into publishers VALUES
16      ( 1, 'Oldenburg Wissenschaftsverlag GmbH',
17      'Rosenheimer Strasse 145',
18      '81671',
19      'Muenchen',
20      'Germany');
21@insert into publishers VALUES
22      ( 2, 'Pearson Education Deutschland GmbH',
23      'Martin-Kollar-Strasse 10-12',
24      '81829',
25      'Muenchen',
26      'Germany');
```

```
insert
into "DOM_STATV"
select
domvalue_1
from dd071
where domname = 'STATV'
and as4local = 'A';
```

Figure 146: SQL Statement: Insert

If the columns in the *insert* statement correspond to the column sequence of the table, the column name does not need to be specified. Otherwise, the column name must precede the value.

You can also fill a table using a query to read records from another table and include filters to extract only the records that you require.

Although inserting records using the SQL Console would not be suitable for mainstream data provisioning, it might be useful for testing purposes or if you need to add some minimal information to support your application, such as settings, where the data rarely changes.

**Note:**

To selectively remove records from a table you use a *DELETE* statement. Remember to include a *WHERE* clause to define the criteria for the records to be deleted. For example: `delete from "CurrDecimals" where "CURRKEY" = 'AFA'`. If you don't include a *WHERE* clause then all records are deleted.

**Note:**

If you want to remove all records from a table, you should use the SQL statement *TRUNCATE* and not a *DELETE* without a *WHERE* clause. This is because a truncate executes faster than delete because logging is not used. For example: `truncate table "CurrDecimals"`

Load a Table Using Source Files

SAP HANA tables are usually updated with SQL insert, update and delete statements that are generated from application code. For example, in S/4HANA when a customer master record is created, multiple HANA tables receive an *insert into* statement generated from the ABAP code.

Tables can also be updated from instructions issued by data loading and replication tools, such as SAP Data Services, SAP LT and SAP BW.

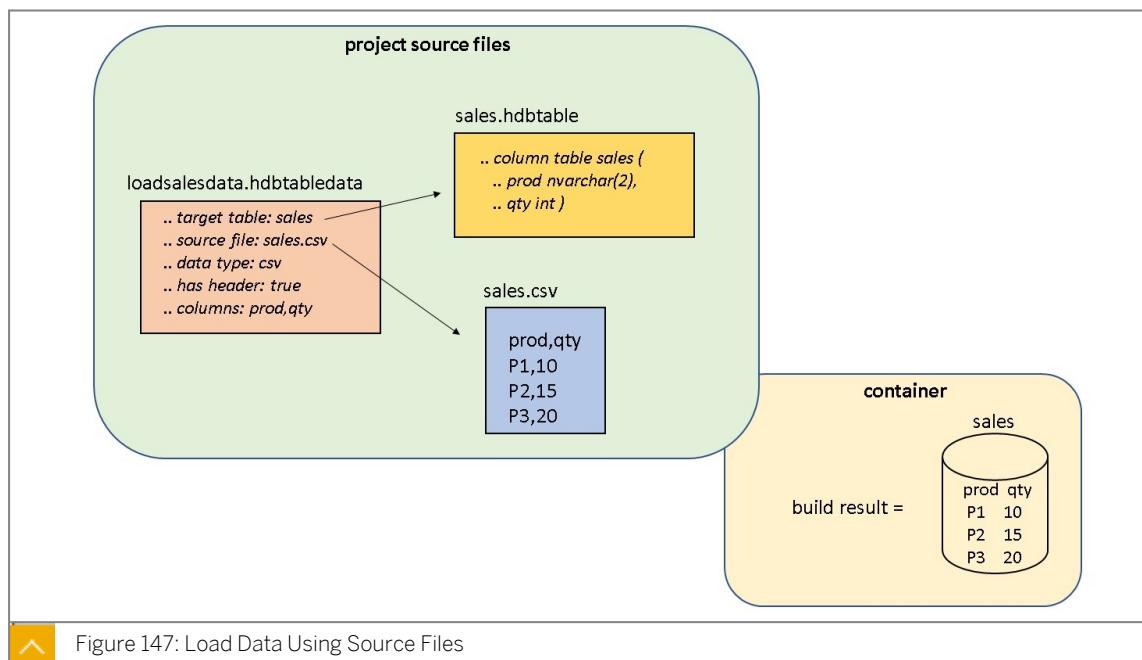


Figure 147: Load Data Using Source Files

But it is also possible for data to be loaded from local files defined within a project, using SAP provided development artifacts.

You can provide the data using a simple csv file. The source file `.hdbtabledata` is then used to provide the loading instructions including pointing to the source file where the data is found, and the target table where data is to be loaded.

Whilst this approach would not be suitable for most scenarios where data is either bulky and/or fast moving, it does provide a simple way to insert local, domain data into tables within

an application, when the application cannot or should not provide this data. For example you might want to store information related to the version of the application, that could be part of the report output. This means the support team could check the version number of the application if issues were found with the report. Only the development team could update the application version information when they next work on the project, by simply updating the internal csv file.

Using the `.hdbtabledata` source file plus a csv file is commonly used to provide the data to a test or demo scenario so that the data that must be used, is included with the other development artifacts. You could add export these files together so that a zip file can be passed to team members who could then import the files into their system so that tables and their data are quickly setup.

Loading data Using the Wizard

Data from local files can be imported directly into tables using a tool that is included in Web IDE.

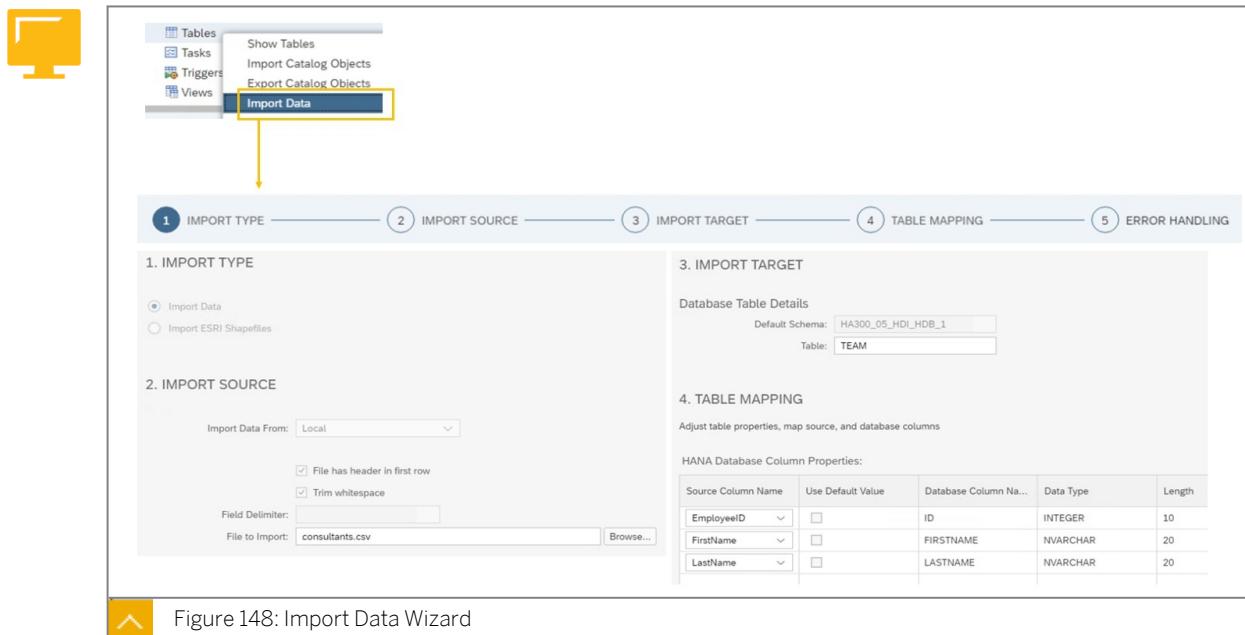


Figure 148: Import Data Wizard

There are no data transformation capabilities.

When you are importing data to a table in a container, the target table must already exist. If you are importing data to a table in a catalog schema, it is possible for the import wizard to also create the target table during the import.

The import wizard is a useful tool to quickly load testing data into a table with very little effort. The import settings are not persisted so they cannot be repeated. You would have to start again each time. The tool is not meant to be used for on-going operational use.



LESSON SUMMARY

You should now be able to:

- Load a Table Using SQL Statements
- Load a Table Using Project Source Files
- Load a Table with Data from a Local File

Accessing Remote Data



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Explain Virtual Tables

Virtual Tables

SAP HANA can access data that is stored in external systems, for example tables or views in databases, files on servers or local machines, cloud sources and much more.

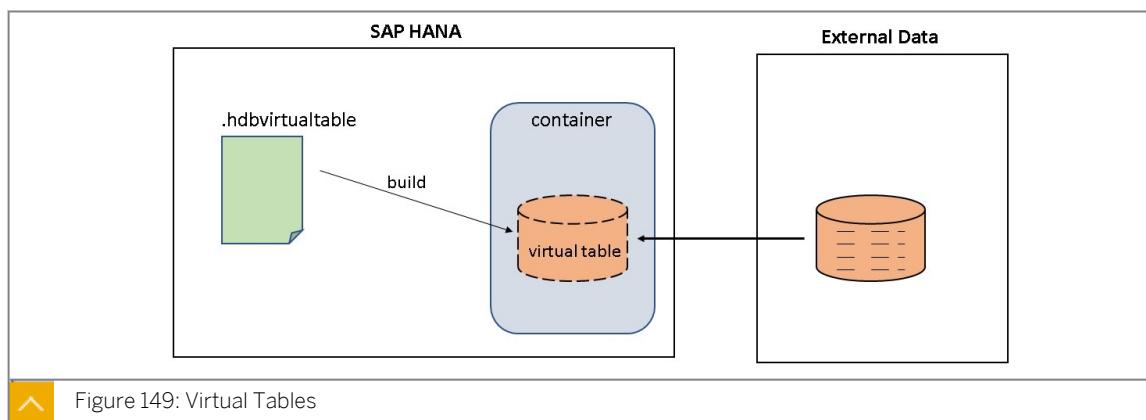


Figure 149: Virtual Tables

To access data in external systems, often referred to as remote data, the SAP HANA administrator should setup a remote source connection from SAP HANA to an external system. This establishes the basic connectivity between SAP HANA and the external system. The developer then creates a special type of SAP HANA table called a *virtual table*. SAP provide a database artifact for the creation of virtual tables. The source file extension is *.hdbvirtualtable* and this file can be added to any folder within the database module of a project so that when the module is built, the virtual tables are part of the build. Within the *.hdbvirtualtable* file you define the specific source object from where the data should be read.

A virtual table can be used wherever a standard table is used, for example in a calculation view or in a table function. You can mix virtual tables and standard tables within a data model. In other words a calculation view can include data sources based on both types of table.

Of course, it should be understood that read performance of virtual tables, especially on large data sets, will probably not be as good as a standard SAP HANA table. It is recommended to investigate pruning options when using virtual tables. Often, virtual tables are used to access older data that might be stored in external archive systems. If a particular user's query does need to access older data and requires only recent data from standard table in SAP HANA, you should ensure that the remote sources are pruned. See the section of this course where we cover pruning techniques.

When the virtual tables is queried, a connection is made to the remote system and the query executes against the remote data source. Data is passed into the virtual table before being read by the calling query. Data is not persistent in the virtual table. This is one of the main reasons for implementing virtual tables : no persistent data footprint in SAP HANA database.



Note:

Virtual table technology is part of Smart Data Access (SDA) which is a standard feature of SAP HANA. For details see the course HA550 - Data Provisioning with SAP HANA

A virtual table can also be created using the SQL Console using `create virtual table`.



LESSON SUMMARY

You should now be able to:

- Explain Virtual Tables

Learning Assessment

1. SAP recommends always create tables using SQL statements in the SQL console.

Determine whether this statement is true or false.

- True
- False

2. Which is the source file type that you use to load a table?

Choose the correct answer.

- A .hdbtable
- B .hdbdropcreatetable
- C .hdbtabledata

3. Virtual tables usually provide superior performance compared to standard tables.

Determine whether this statement is true or false.

- True
- False

Learning Assessment - Answers

- SAP recommends always create tables using SQL statements in the SQL console.

Determine whether this statement is true or false.

True

False

Correct - You should avoid creating tables using SQL statements and instead define the tables using source files as part of your development project.

- Which is the source file type that you use to load a table?

Choose the correct answer.

A .hdbtable

B .hdbdropcreatetable

C .hdbtabledata

Correct - .hdbtabledata is the source file type used to load data to a table

- Virtual tables usually provide superior performance compared to standard tables.

Determine whether this statement is true or false.

True

False

Correct - Compared to standard tables, virtual tables are not expected to perform as well due to source the data being located outside of SAP HANA database.

UNIT 6

Optimization of Models

Lesson 1

Implementing Good Modeling Practices

233

Lesson 2

Implementing Static Cache

249

Lesson 3

Controlling Parallelization

253

Lesson 4

Implementing Union Pruning

259

Lesson 5

Using Tools to Check Model Performance

263

Lesson 6

Developing a Data Management Architecture

271

UNIT OBJECTIVES

- Implementing good modeling practices
- Implement Static Cache to Improve Calculation View Performance
- Controlling Parallelization
- Implement Union Pruning
- Use tools to check model performance
- Implement Good Data Management Architecture

Implementing Good Modeling Practices

LESSON OVERVIEW

There are many choices to make when developing SAP HANA models. The wrong choice can severely affect the function and performance of your model. In this lesson, we will guide you towards good modeling approaches.



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Implementing good modeling practices

Understanding Optimization and Instantiation

Before we dive into the detail of how to improve calculation view performance, it is helpful to learn a little about the way a calculation view responds to a query that calls it.

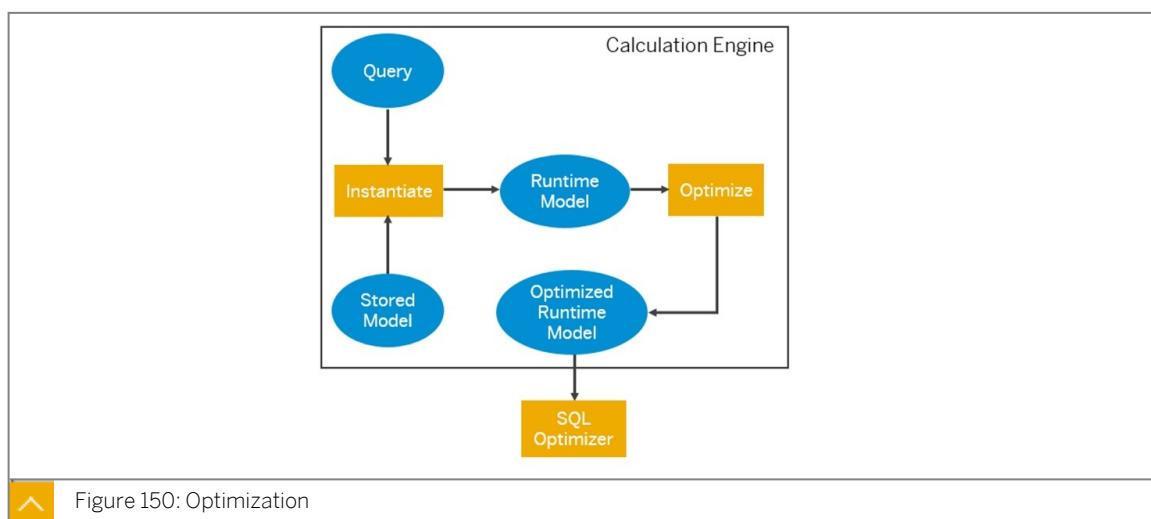
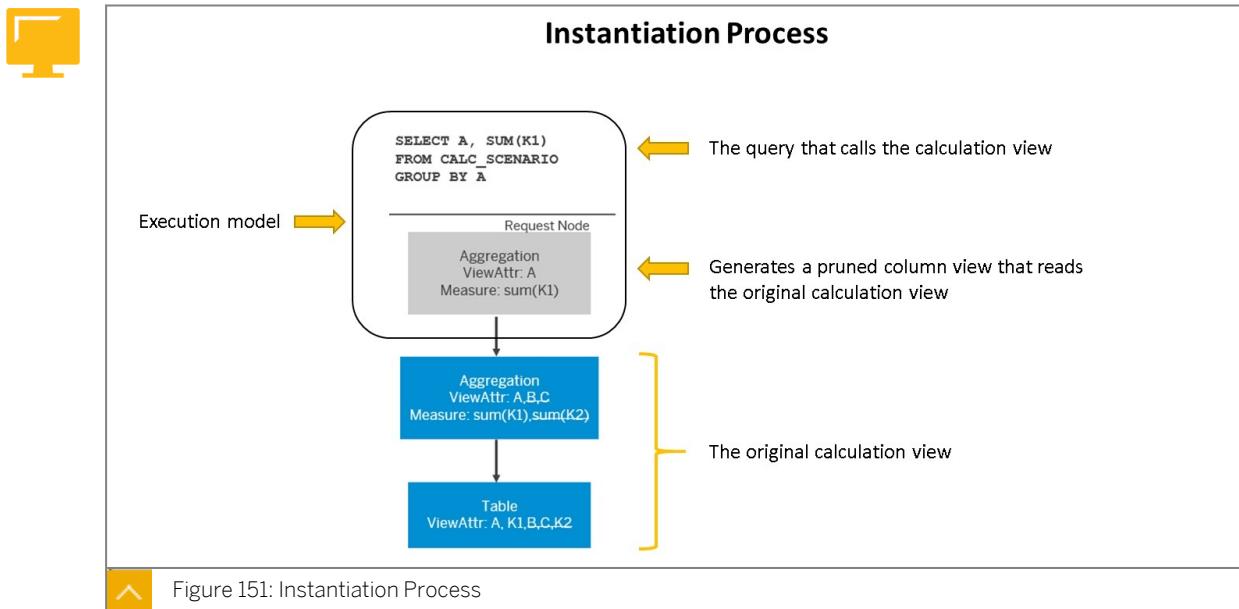


Figure 150: Optimization

The calculation engine pre-optimizes queries before they are worked on by the SQL optimizer.

A calculation view is instantiated at run-time when a query is executed. During the instantiation process, the calculation engine simplifies the calculation view into a model that fulfills the requirements of the query. This results in a reduced model that can be further optimized by the calculation engine. For example, it considers settings such as dynamic join, join cardinality and union node pruning

After this, the SQL optimizer applies further optimizations and determines the query execution plan, for example, it determines the optimal sequence of operations, and also those steps that could be run in parallel.



The instantiation process transforms an original (one that you define) calculation view into an execution model based on a query that is run on top of a calculation view. The generated view is pruned of unnecessary elements from the original calculation view and is technically a column view that references one specific node of the original calculation view.

During the instantiation process, the query and the original calculation model are combined to build the optimized, execution calculation model.

Best Practices for Modeling

Throughout this course we have implemented a design approach that provides a basis for good performance.

In this lesson, we will re-emphasize some of those approaches and also introduce some new recommendations for good design practices. We will also introduce some special modeling settings that might help to improve the performance of your calculation views.

General Design Recommendations

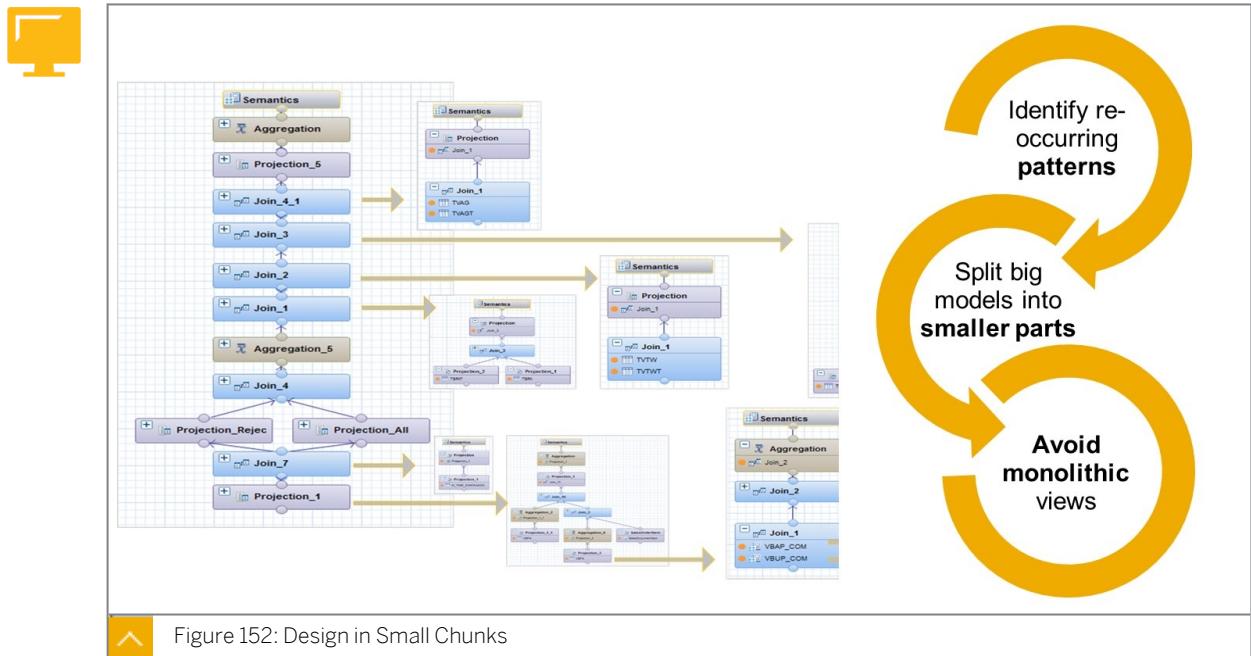


Figure 152: Design in Small Chunks

You should try to break down large models into individual calculation views so that they are not only easier to understand, but also allow you to define reusable components, thus avoiding redundancy and duplicated maintenance as a consequence.

Try to develop calculation views in layers as this promotes a high level of re-useability. Each layer consumes its lower layers. Any changes to the lower layers immediately impacts the higher layers. The lowest layers should provide the foundation calculation views. The higher layers add increasing semantics to provide more business meaning to the views.

Do not be concerned that breaking up large calculation views into multiple objects will damage overall run-time performance. At run time, the SQL compiler attempts to flatten the entire stack of calculation views into a single executable model. Remember the calculation view definitions, however many there are in the stack, are simply a logical view of the data flow and provide the intent. How the compiler puts together the final query execution can never be known at the design stage by simply observing the calculation view definition.

Note:

Be aware that when there are a large number of calculation views in a stack, **build** times usually increase due to dependencies. This has nothing to do with run-time.

Check Query Unfolding



Aim for query unfolding

Context

- Queries are generated in a way that makes query objects **directly** accessible to the SQL optimizer on a detailed level

Impact

- Unfolded plans should be faster on average
- Compilation time of query might increase



Figure 153: Performance Recommendations 3

At run time, your calculation view is analyzed by the SQL Processor. A key aim is to convert each calculation view operation into a corresponding SQL operator in order to convert the complete data flow graph into a single SQL statement. This is called **unfolding**. A benefit of unfolding to SQL is that only the SQL engine is needed and no other SAP HANA engine is called, such as the Calculation Engine or the specialist engines such as Spatial or Graph. However, due to your design, there might be complex steps that cannot be translated into SQL operators. If these steps cannot be translated, they will be processed in the specialist SAP HANA engines and one or more column views will then be materialized for the SQL to read as an interim data source. That means the SQL doesn't read directly from the database objects e.g. tables, functions, but instead from materialized column views. This ultimately means the SQL optimizer is not free to work with plain SQL but has to accept that some of the processing will be done by other engines. Mixing engines would result in a partially unfolded query that might not perform as well as a completely unfolded query due to limited SQL optimizations.

So what limits unfolding?

Some operators cannot be unfolded. The list of operators gets shorter with each SAP HANA release, and in fact there are very few operators that cannot be unfolded in simple calculation views. See SAP NOTE 2441054 for details.

It should be emphasized that although unfolding is generally desirable, sometimes unfolding can actually damage overall performance. For example, query compilation time can increase when the query has to be unfolded each time the calculation view is called with new selections. This is because each query is usually different and would need to be broken down into individual operations and converted to plain SQL steps each time it is used. Sometimes, it is better to have a folder, or partially folder query.

It is possible to set a SQL **hint** that prevents unfolding for specific calculation views. This is found under *Semantic Node → Properties → Advanced*.

You can also set a global parameter to prevent unfolding for all calculation views.

You can check if your calculation view is unfolded by using the *Explain Plan* option in the SQL Console, where you can see the list of operators and data sources that are accessed.



Unfolded Query – accesses SQL objects					
Rows (5)					
	OPERATOR_NAME	OPERATOR_DETAILS	EXECUTION	TABLE_NAME	TABLE_TYPE
1	PROJECT	STUDENTSENRICHMENT.SITE, STUDENTSENFTHEX	NULL	NULL	
2	LIMIT	NUM RECORDS: 1000	HEX	NULL	NULL
3	AGGREGATION	GROUPING: STUDENTSENRICHMENT.SITE, ST	HEX	NULL	NULL
4	INDEX JOIN	INDEX JOIN CONDITION: STUDENTS.ID = STUD	HEX	STUDENTS	COLUMN TABLE
5	COLUMN TABLE		HEX	STUDENTSENRICHMENT	COLUMN TABLE

Folded Query – accesses column views					
Rows (4)					
	OPERATOR_NAME	OPERATOR_DETAILS	EXECUTION	TABLE_NAME	TABLE_TYPE
1	COLUMN SEARCH	exampleNoKAnonymity.SITE, exampleNoK	COLUMN	NULL	NULL
2	LIMIT	NUM RECORDS: 1000	COLUMN	NULL	NULL
3	AGGREGATION	GROUPING: exampleNoKAnonymity.SITE,	COLUMN	NULL	NULL
4	COLUMN VIEW		COLUMN	exampleNoKAnonymity	CALCULATION VIEW

Figure 154: Check Unfolding

If a *column view* is called, then you know this is not an unfolded query. What you are looking for are accesses only to SQL objects, such as tables and functions. If that is the case, then you have a completely unfolded query.

Also check the query compilation time versus the execution time to see how unfolding is affecting the *overall* run time.

It is possible to identify how many queries were not unfolded. Run a query on system view **M_FEATURE_USAGE** as follows:

```
SELECT
    FEATURE_NAME,
    CALL_COUNT
FROM
    "M_FEATURE_USAGE"
WHERE
    COMPONENT_NAME = 'CALCENGINE'
AND
    FEATURE_NAME
IN
    ('QUERY TOTAL', 'QUERY UNFOLDING')
```

Set Join Cardinalities

 **Always set join cardinalities**

Context

- Based on join cardinality setting some joins can be omitted

Join Type:

Cardinality:

Impact

- Join pruning can improve runtime with large models
- Wrong setting can lead to unexpected results

Consequences

- Use join cardinality setting whenever possible
- Use cardinality proposal where applicable (only tables as sources)
- Ensure continued correctness of join cardinality setting
(also in future and in target systems of transport)

 Figure 155: Set Join Cardinalities

When you define a join, you can optionally set the cardinality to describe the relationship between the two data sources, for example, 1:1 ,1:m, n:m.

We recommend that you always set join cardinalities so the optimizer can decide if joins can be omitted at run time based on the query that is being sent to the calculation view.

If tables are used as the data sources, and not calculation views, then you can use the *Propose Cardinality* option.

 Caution:

The cardinality setting is proposed based on the content of the joined tables at the time you click the *Proposal* button. The cardinality setting does not automatically update as the table content changes. You might need to come back and change the cardinality setting if the relationship in the data changes. Also, remember that you need to use realistic data if you plan to use the proposal. It is no good proposing cardinality on a few test records that later do not make sense on live data.



Prerequisite for join pruning

- a) no field is requested from the to-be-pruned table
- b) the join type is outer, referential, or text
- c) the join cardinality is either "..1" for the to-be-pruned table or only measures with count distinct aggregation or no measures at all are requested

Join

Join Definition	Mapping	Calculated Columns (0)	Parameters (0)	Columns (5)	Filter Expression												
+ <input type="button" value="Data Sources"/> <input type="button" value="New..."/> <input type="button" value="Edit..."/> <input type="button" value="Delete..."/> <input type="button" value="Import..."/> <input type="button" value="Export..."/>	<input type="button" value="Output Columns"/> <input type="button" value="New..."/> <input type="button" value="Edit..."/> <input type="button" value="Delete..."/>																
<table border="1"> <tr> <td><input type="checkbox"/> salesOrder</td> <td><input type="checkbox"/> employee_1</td> <td><input type="checkbox"/> amount</td> <td><input type="checkbox"/> employee</td> <td><input type="checkbox"/> manager</td> <td><input type="checkbox"/> amount</td> </tr> <tr> <td><input type="checkbox"/> employee</td> <td><input type="checkbox"/> salesOrder</td> <td><input type="checkbox"/> employee_1</td> <td><input type="checkbox"/> manager</td> <td><input type="checkbox"/> salesOrder</td> <td><input type="checkbox"/> employee</td> </tr> </table>						<input type="checkbox"/> salesOrder	<input type="checkbox"/> employee_1	<input type="checkbox"/> amount	<input type="checkbox"/> employee	<input type="checkbox"/> manager	<input type="checkbox"/> amount	<input type="checkbox"/> employee	<input type="checkbox"/> salesOrder	<input type="checkbox"/> employee_1	<input type="checkbox"/> manager	<input type="checkbox"/> salesOrder	<input type="checkbox"/> employee
<input type="checkbox"/> salesOrder	<input type="checkbox"/> employee_1	<input type="checkbox"/> amount	<input type="checkbox"/> employee	<input type="checkbox"/> manager	<input type="checkbox"/> amount												
<input type="checkbox"/> employee	<input type="checkbox"/> salesOrder	<input type="checkbox"/> employee_1	<input type="checkbox"/> manager	<input type="checkbox"/> salesOrder	<input type="checkbox"/> employee												

SELECT
"salesOrder",
"employee_1",
SUM("amount")
FROM
...
GROUP BY
"salesOrder",
"employee_1"

Figure 156: Prerequisites for Join Pruning

One of the main benefits of having the correct cardinality set is that the SQL optimizer can prune entire data source from a join if certain prerequisites are met.



When only fields from left table are requested....

Join Type: Left Outer Join Type: Left Outer Join Type: Right Outer

Cardinality: 1..1 Cardinality: 1..1 Cardinality: 1..1

Figure 157: Example of Join Pruning

Referential Join Type



Use referential join where applicable

Table for which referential integrity holds can be selected

Integrity Constraint:

- *Left*: every entry in left table has at least one match in right table
- *Right*: every entry in right table has at least one match in left table
- *Both*: every entry in both tables has at least one match in the other table

Allows for **join pruning**, if

- no field is requested from to-be-pruned table
- integrity is placed on the not-to-be-pruned table
- the to-be-pruned table has join cardinality :1

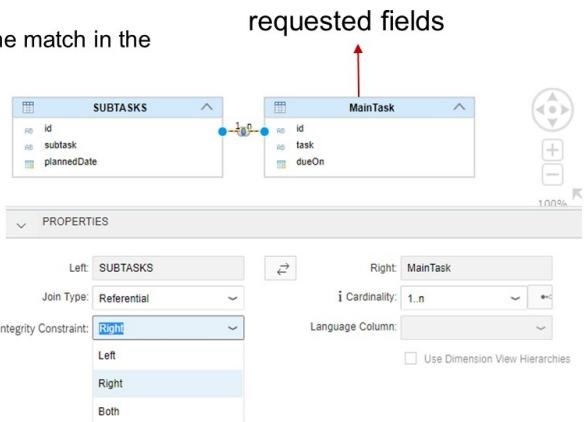


Figure 158: Referential Join Type

Consider using the join type *Referential* if your joins do not have to ensure integrity of the data, and are defined to provide access to additional columns where you already trust the data integrity (e.g. the sales item will always have a header).

It might be possible for the SQL optimizer to prune the data source from the join if it is not needed, if certain conditions are met.

Unblock Filter Push-down



Consider using flag – Ignore Multiple Outputs for Filter

- By default, nodes that are consumed by multiple nodes prevent filter push-down from the consuming nodes to ensure semantic correctness
- if push-down does not violate semantic correctness flag “Ignore Multiple Outputs For Filter“ can be set to enforce filter push-down

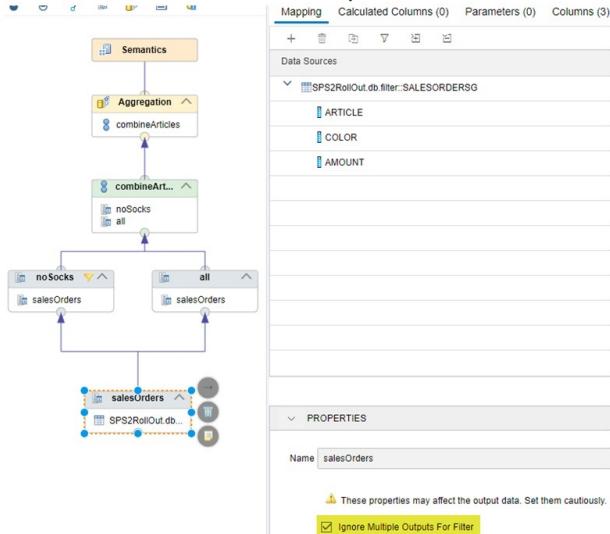


Figure 159: Unblock Filter Push-down

In your calculation view, if you have defined a node that is a source to more than one other node (in other words the data flow splits), then you need to be aware that this can potentially block filter push-down. The SQL optimizer does not want to break the semantics of the query and so by default it keeps the filter at the node at which it is defined (and this node might be high up in the stack causing large data sets to travel up the stack).

You can set the flag *Ignore Multiple Outputs for Filter* so that the blockage of filter push down is removed and the optimizer is free to push the filter down to the lowest level to get the best performance, but be very careful to check the query semantics are not broken and wrong results are produced.

General Join Recommendations

 **Using Joins**

Joins

- Always maintain the cardinality of the joins.
- Try to use n:1 and 1:1 left outer joins or 1:n and 1:1 right outer joins
- Try to reduce number of join fields
- Avoid joining on calculated columns

 Performance validation warnings

> In Join_1 Node, the calculated column "CC_1" is used in the join definition of the input

- Avoid type conversions at runtime



- Check whether features of Dynamic Join and Optimize Join Columns can be used.

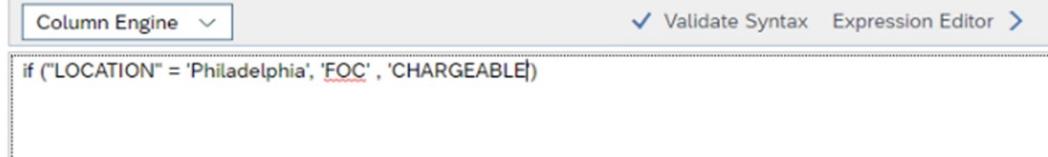
 Figure 160: General Join Recommendations

Joins are a source of potential performance improvements. Make sure you consider all the recommendations when defining joins to ensure you get the best performance from your calculation views.

Use Plain SQL not CE Language in Expressions

Write Calculation View Expressions in SQL, not Column Engine Language

 **Avoid column engine language**



When you can achieve the same outcome with SQL language

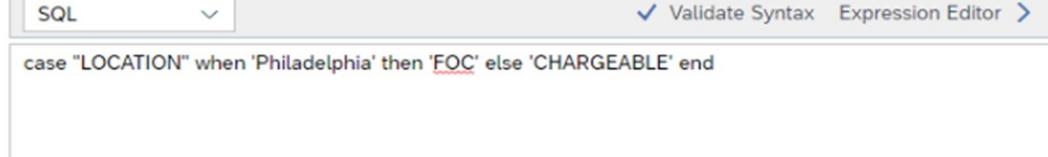


 Figure 161: Use SQL for Expressions

Expressions are used in many places within calculation views including calculated columns, restricted measures, and filter expressions. When you build expressions, you can choose to use either the **Column Engine** expression language or **SQL** expression language. The language is chosen using a drop-down selector in the Expression Editor.

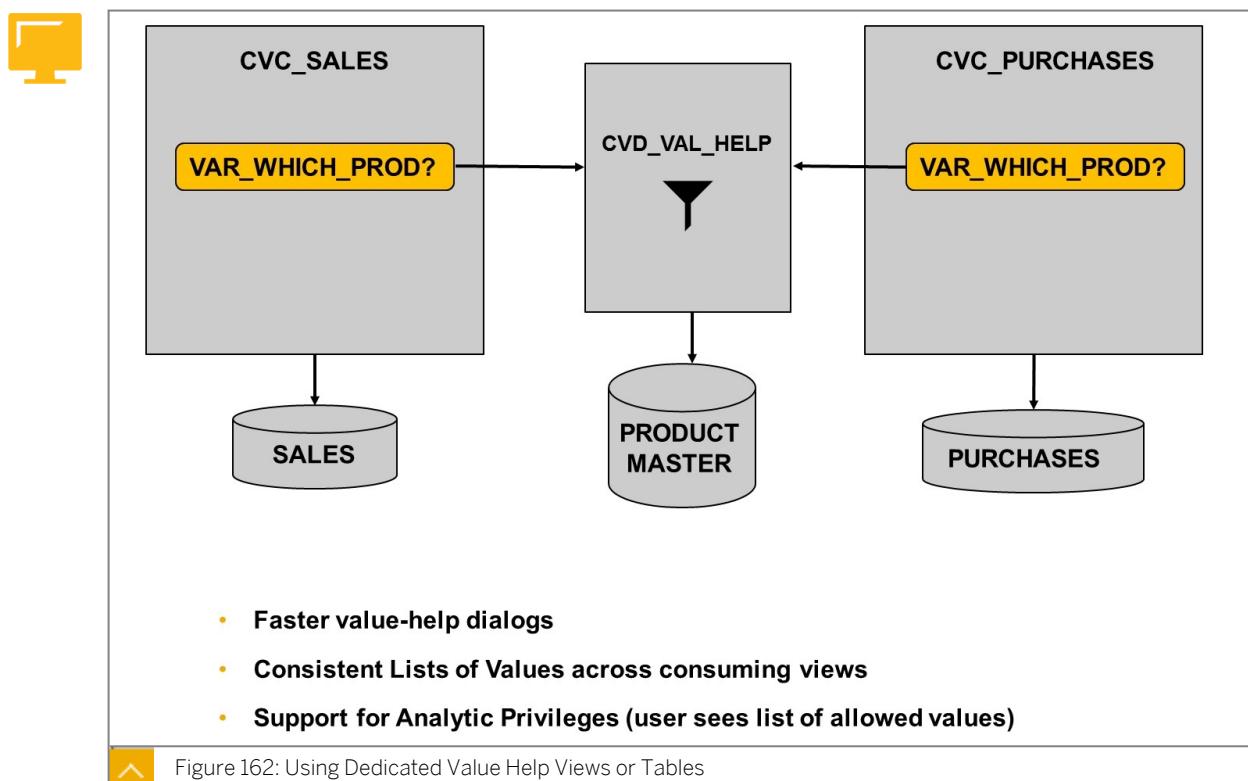
SAP recommends that you use SQL expressions wherever possible. SAP will continue to develop more functions using SQL only and it will become the default expression language going forward. Whether you choose to use SQL or column engine expressions makes no difference to the expression result. At run time, column engine expressions are converted to SQL expressions. We recommend that you write the expression using SQL where possible for best performance and future compatibility. SAP only includes the column engine language to ensure backward compatibility with older models that may still have expressions written in that language.

You should be able to recreate most, if not all column engine expressions to plain SQL. For example, the *If - Then - Else* column engine expression can easily be replaced with the standard SQL expression CASE to achieve the same outcome.

Even if you choose to use the column engine language, SAP HANA converts this to plain SQL at run time so that more of the query can be unfolded for better performance. Working with SQL means the conversion does not have to take place and performance might be improved.

Use Dedicated Views or Tables for ‘List of Values’

For performance issues, we generally recommend that you create dedicated calculation views or tables to generate the list of help values for variables and input parameters.



This approach enables a faster display of the list of values, without reading the entire view or table on which the calculation view is based.

This is also best practice so that the list of values is reusable and consistent from one calculation view to another, and users see the same selection values.

The list can be driven by simple fixed filters in the help calculation view or even by analytic privileges that offer a dynamic list restricted to what each user is allowed to choose from.

Column Pruning

Don't create large intermediate result sets

Column Pruning

- Make sure that you understand which attributes are requested on each operation node and why
- Verify that the “intermediate” sets are as small as possible

Type	Name	Label	Aggregation T...	Variable
□	PartnerId	PartnerId	~	~
□	TITLE	TITLE	~	~
□	FIRST_NAME	FIRST_NAME	~	~
□	LAST_NAME	LAST_NAME	~	~
□	GENDER	GENDER	~	~
□	PHONE_NUMBER	PHONE_NUMBER	~	~
□	AddressId	AddressId	~	~
□	BillingStatus	BillingStatus	~	~
□	CreatedAt	CreatedAt	~	~
□	CreatedBy	CreatedBy	~	~
□	Currency	Currency	~	~
□	DeliveryStatus	DeliveryStatus	~	~
□	SalesOrderId	SalesOrderId	~	~
□	NetAmount	NetAmount	SUM	~

Figure 163: Column Pruning

Make sure you do not map columns that are not needed. This might sound like a fairly obvious recommendation, but it is very common for developers to simply map all columns without considering if, and why, there are used. They are often mapped 'just in case' they are needed. Too many attributes can lead to very large, granular data sets especially when only aggregated data is needed.

Optimal Filtering and Calculated Columns



Good Design Practices

Filtering

- Verify that the filters are pushed down to the base tables
- Try to avoid filters on calculated columns

Column Calculations

- If the calculations are coming from a single table and they are row based calculations – try to materialize them as “generated as” columns
- Try to move calculations to the highest node

Aggregations

- Aggregate early as possible in the data flow
- For star-join scenarios always use a star-join model to invoke the OLAP engine - avoid modeling star-join scenarios as a sequence of relational joins.



Figure 164: Filtering and Defining Calculated Columns

Always try to filter as early as possible in the stack. The objective is to prune data as early as possible to avoid large data sets that can harm performance.

Avoid including calculated columns in filter expressions. It's better to first create a new column based on the calculation, and then add a projection node on top to filter the result set based on the column (which is no longer calculated).

Instead of creating a calculated column in a calculation view, use the SQL *generated as* expression to calculate values. This can only be done if the components of the calculation come from one table and from a single row of the table.

Calculate as high as you can in the stack so that you avoid calculating on granular data and instead calculate on aggregated data.

Always reduce the data set by introducing aggregations early in the modeling stack.

If you consume a calculation view that produces attributes, do not change them to measure and vice-versa.

Make sure you use the features of the star join for dimensional OLAP models. Do not try to create these using standard join nodes. These may not perform as well as the built-in star join node.

Best Practices for Writing SQL

One of the benefits of using graphical modeling tools is that you don't have to concern yourself with writing SQL code. However, there are many times when you do need to write the SQL code directly, such as when you are developing a function or a procedure.

Writing the SQL directly offers the most flexibility to control exactly what you want to achieve and how you want to achieve it, but you may write code that is suboptimal for SAP HANA. Even experienced SQL coders should pay attention to the following guidelines to avoid applying techniques that do not work well with SAP HANA. SAP has extended standard SQL

with many extensions to the language, which can potentially lead to poor performance of calculation views if not handled well.

The official SAP Help documentation provides more details, but here are some of the basic guidelines:



Reduce complexity of SQL statements

- Use variables to break up statements

```
books_per_publisher = SELECT publisher, COUNT (*) AS cnt
FROM :books GROUP BY publisher;
largest_publishers = SELECT * FROM :books_per_publisher
WHERE cnt >= (SELECT MAX (cnt))
```



Figure 165: SQL Best Practices – Table Variables

You can reduce the complexity of SQL statements by using table variables. Table variables are used to capture the interim results of each SQL statement and can then be used as the inputs for subsequent steps. This makes understanding the code much easier and does not sacrifice performance. Using variables also means the calculated data set can be referred to multiple times within the script, thus avoiding repeating the same SQL statement. So we highly recommend that you use table variables.



Reduce dependencies

- Avoid imperative statements

```
IF <bool_expr1>
THEN
    <then_stmts1>
[ {ELSEIF <bool_expr2>
THEN
    <then_stmts2>}... ]
[ ELSE
    <else_stmts3>]
END IF
```



Figure 166: SQL Best Practices — Dependencies

You can reduce dependencies by ensuring that your SQL steps are independent from each other. When you wrap SQL expressions in looping constructs or use IF/THEN/ELSE expressions to determine the flow, you inhibit the ability of the SQL Optimizer to produce the best plan. This is because you have declared a specific flow that the SQL optimizer cannot break. Also, using expressions that control the logic makes it harder to create a plan that can be parallelized and therefore achieve the best performance. So, always consider if you can achieve the same results using only declarative language, and try to control the flow with imperative language, such as loops and if/then/else.



Avoid reading records one at a time

- Avoid cursors

```
CREATE PROCEDURE foreach_proc LANGUAGE SQLSCRIPT AS
Reads SQL DATA
BEGIN
    DECLARE val decimal(34,10) = 0;
    DECLARE CURSOR c_cursor1 FOR
        SELECT isbn, title, price FROM books;
    FOR r1 AS c_cursor1 DO
        val = :val + r1.price;
    END FOR;
END;
```



```
SELECT sum(price) into val FROM books;
```



Figure 167: SQL Best Practices — Avoid Cursors

Cursors are a powerful feature of SQL Script and can be useful when you need to read a table one record at a time.

However, when writing SQL Script that will be consumed by calculation views, you should avoid cursors as this slows things down. Think about writing the code using declarative language.



LESSON SUMMARY

You should now be able to:

- Implementing good modeling practices

Implementing Static Cache



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Implement Static Cache to Improve Calculation View Performance

Caching View Results

Complex calculation views place a heavy load on the system. To reduce this, it is possible to cache the results of a calculation view in order to speed up the time needed to get the results when executing a query against the view. This feature is useful when you have complex scenarios or large amounts of data so that the results do not have to be recalculated each time the query is executed. The benefits are not just the reduced time for the results to appear for the user of the query, but it also means CPU and memory consumption is reduced leading to better performance for other tasks.



Static Cache – Overview

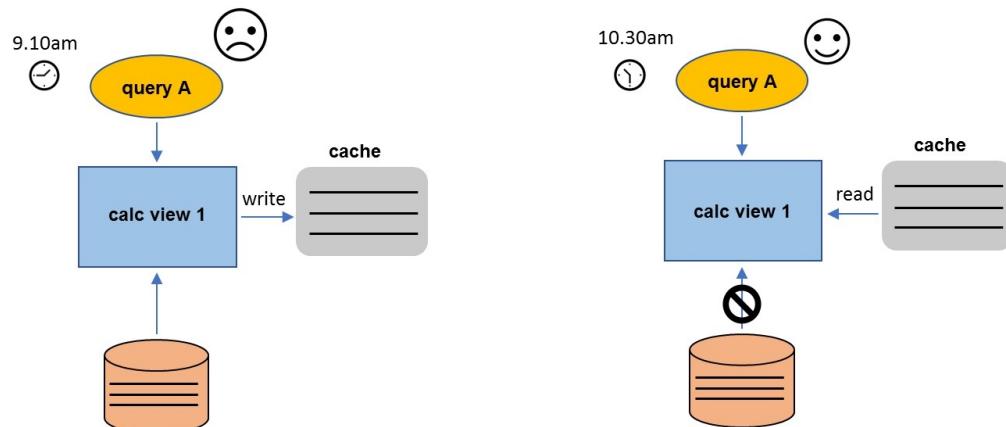


Figure 168: Calculation View Cache Overview

If the cache cannot be used, due to a query requesting a different data set than what is held in cache, a query does not fail. It means that the query will need complete processing.

Caching should not be applied to raw data but to data that has been highly processed and reduced through aggregations and filtering. In practice, this means applying caching to the top-most nodes of a calculation view.

Cache can only be used for calculation views that do not check for analytic privileges. This means analytic privileges should be defined on the top-most view only, in a calculation view stack. The optimal design would be to define the cache setting at the highest calculation view possible in the stack, but not at the very top where analytic privileges are checked. This

means that the user privileges are always checked against the cached data, if the cache is useable.

Static Cache – Cached Columns

Semantics

View Properties Columns (4) Hierarchies (0) Parameters (0)

General Advanced Static Cache

Enable Cache Force

* Retention Period

Columns

Column1 Column2 Column3 Column4

Subset of columns are cached

cache

query

- ✓ column1, column2, column4
- ✗ column1, column3
- ✓ column1, column4

Figure 169: Calculation View Cache - Columns

It is possible to fine-tune the calculation view cache using column and filter settings.

Firstly, calculation view caching can be defined at the column level. This means queries that use only a sub-set of the cached columns can be served by the cache. It is recommended to cache only the columns that are frequently requested in order to reduce memory consumption and speed up cache refresh. If you do not specify columns in the cache settings, then all columns are cached.

Static Cache – Filter

Columns

Column1 Column2 Column3 Column4

Filters

Column3 = X

cache

query

- ✗ column1, column2, column4 WHERE COLUMN3=X
- ✓ column1, column2, column4 WHERE COLUMN3='X' AND COLUMN2='Y'
- ✗ column1, column2, column4 WHERE COLUMN3='X' OR COLUMN2='Y'
- ✗ column1, column2, column4 WHERE COLUMN3='Y'

Figure 170: Calculation View Cache - Filters

The cache size can be further reduced by defining filters in the cache settings. Queries that use either exactly the cache filters, or a subset of the filters, are served by the cache. Cache is only used if the query filter matches exactly the filter that is defined for the cache, or reduces the data further.

Figure 171: Calculation View Cache - Retention

It is possible to define a retention period of the cache so that the cached data expires. The value entered is in minutes.



Note:

It is currently not possible to define that cache should be invalidated if new data is loaded to the underlying tables. This feature is expected to come later. For now we just have a time-based expiry (in minutes).

In order to use the calculation view static cache there are some important prerequisites that must be met.

a) Enable Cache is selected

b) Calculation view can be unfolded

c) No granularity tracking calculations that prevent cache-use (to avoid unexpected results)

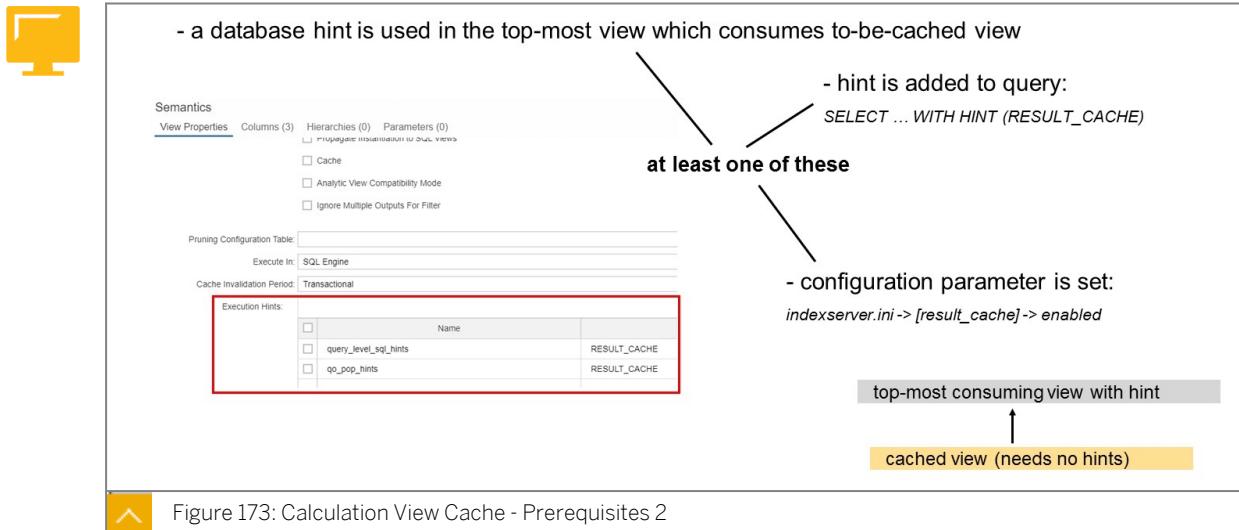
Figure 172: Calculation View Cache - Prerequisites 1

Firstly, the basic setting *Enable Cache* must be set in the calculation view to be cached.

Then, the query must be able to be fully-unfolded. This means that the entire query must be fully translatable into a plain SQL statement and not require the calculation of interim results. You can use the *Explain Plan* feature of the SQL Analyzer to check if unfolding can occur.

There must be no calculations that depend on a specific level of calculated granularity as the cache is stored at the level of granularity requested by the first query. This may not align to the aggregation level required by subsequent queries and could cause incorrect results.

Even with these prerequisites met, the cache feature is disabled by default and is only considered when explicitly invoked by using one of the following RESULT_CACHE hints:



If the prerequisites for cache usage are not met, it might still be possible to force the cache to be used. You can use the setting *Force* so that the cache is used. However, you should test extensively to ensure that correct results are returned.



LESSON SUMMARY

You should now be able to:

- Implement Static Cache to Improve Calculation View Performance

Controlling Parallelization



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Controlling Parallelization

Controlling Parallelization in a Data Flow

SAP HANA always attempts to automatically apply parallelization to queries in order to optimize performance. Following good modeling practices will ensure the queries that run on your calculation views have the best chance of being parallelized. However, there are cases when the optimizer will not apply parallelization in a complex, long-running data processing step as it is not able to understand the business semantics and is concerned it might break the results if it tried to parallelize. So, instead, it cautiously leaves the processing step as sequential even though it might harm performance.

However, if you are able to ensure the step could be safely run in a parallel way by splitting up the data into semantic partitions, then you can dictate when parallelization should occur.

Within a calculation view, it is possible to force parallelization of data processing by setting a flag *Partition Local Execution* to mark the start and also the end of the section of a data flow where parallelization is required. The reason you do this is to improve the performance of a calculation view by generating multiple processing threads at specific positions in the data flow where performance bottlenecks can occur.

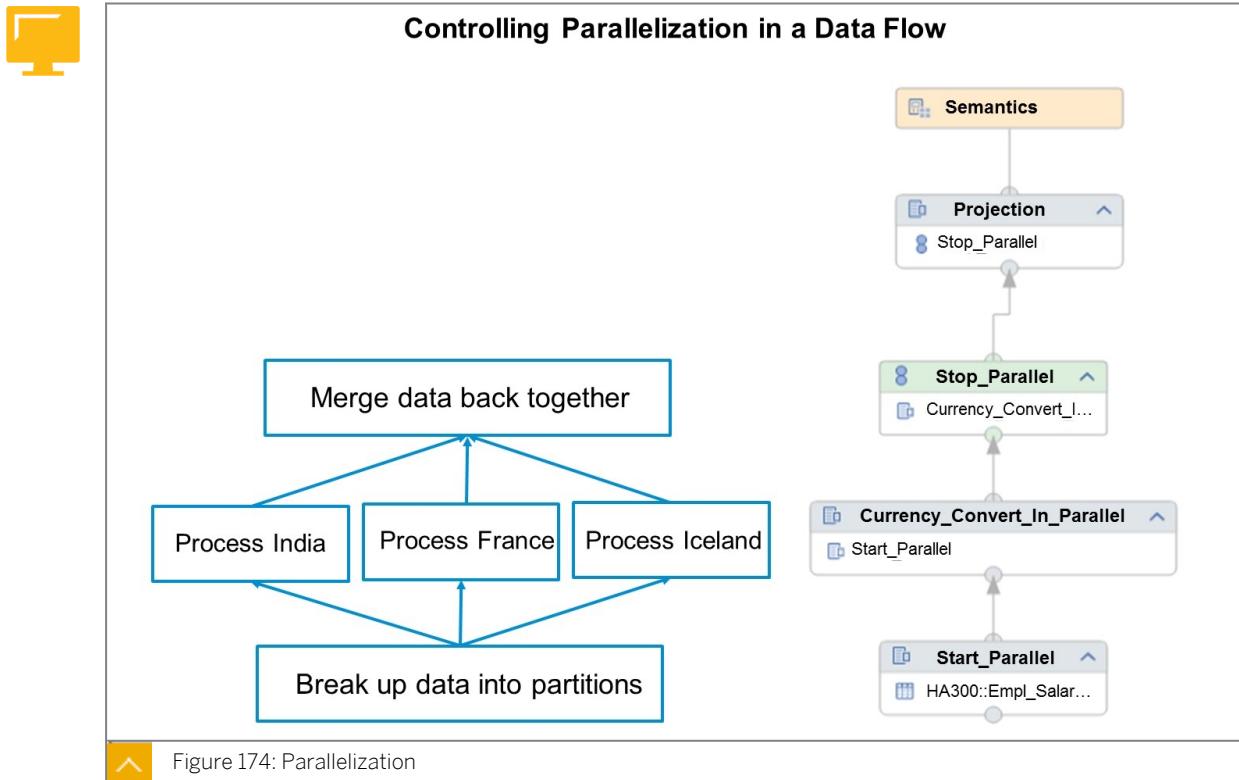


Figure 174: Parallelization

The parallelization block begins with a calculation view node that is able to define a table as a data source. This node could be a projection node or an aggregation node. It is not possible to use any other type of data source such as a function or a view.

In the *Properties* of the chosen start node, a flag *Partition Local Execution* is set to signal that multiple threads should be generated from this point onwards. It is possible to define a source column as a partitioning value. This means that a partition is created for each distinct value of the selected column. For example, if the column chosen was *COUNTRY*, then a processing thread is created for each country. Of course it makes sense to look for partitioning columns where the data can be evenly distributed. The partitioning column is optional. If it is not selected then the partitioning defined for the table is used.

If you don't explicitly define the partitioning column, then the partitioning rules of the underlying table are applied (assuming the table is partitioned).

To end the parallelization block you use a union node. But unlike a regular union node that would always have at least two data sources, a union used to terminate the parallel processing block is fed only from one data source. The union node combines the multiple generated threads but the multiple inputs are not visible in the graphical editor and so the union node appears to have only one source from the node below. You cannot combine any other data sources in the union node that is used to terminate the parallelization. In the *Properties* of the union node, a flag '*Partition Local Execution*' is set to signal the ending of the parallelization block.



Controlling Parallelization - Restrictions 1/2

*start of parallelization
only in nodes with
table data source
(lowermost node)*

*no
parallelization
across views*

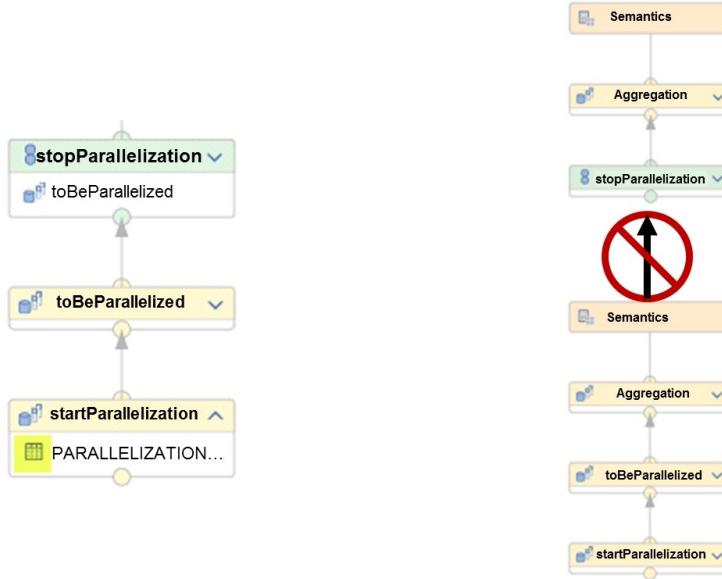


Figure 175: Controlling Parallelization Restrictions 1/2

There are some restrictions that you should be aware of.

The parallelization block always starts with a node that includes a data source that must be a **table**. You can use a table defined in the local container or a synonym which points to an external table. You cannot use another calculation view as the data source or a table function.



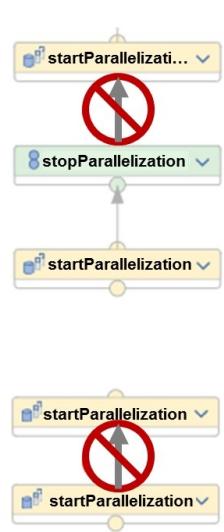
Note:

Support for synonyms was not available in the first release of the feature which was SAP HANA 2.0 SPS04, only tables were supported.



Controlling Parallelization - Restrictions 2/2

Only one parallelization block per query



Only if parallelization column is defined

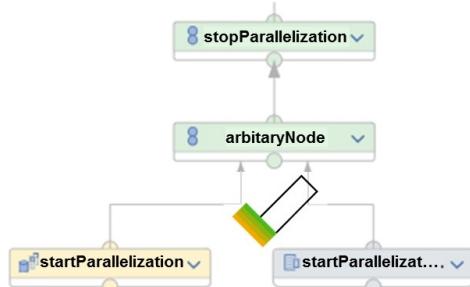


Figure 176: Controlling Parallelization Restrictions 2/2

Only one parallelization block can be defined per query. This means you cannot stop and then start another block either in the same calculation view or across calculation views in the complete stack. You cannot nest parallelization blocks, e.g. start a parallelization block then start another one inside the original parallelization block.

It is possible to create multiple start nodes with different partitioning configurations, but to do this explicitly define the partitioning column. If you create multiple start nodes then all threads that were generated are combined in a single ending union node.



Note:

In addition to defining logical partitions using this technique, always remember that SAP HANA will attempt to apply additional parallelization to the logical partitions.

To check the partitioning of the data you can define a calculated column within the parallelization block with the simple column engine expression `partitionid()`. In the resulting data set you will then see a partition number generated for each processing thread.

Other techniques to monitor parallelization:

- You can also collect trace information by adding `WITH PARAMETERS ('PLACEHOLDER' = ('$$CE_SUPPORT$$',))` to the end of the query statement.
- Use SQL Analyzer navigate to the nodes that should be parallelized and you should see the nodes are duplicated according to the number of partitions.



Caution:

It is important to not overuse this feature as over-parallelizing can lead to poor overall performance where other processes are affected.



LESSON SUMMARY

You should now be able to:

- Controlling Parallelization

Unit 6

Lesson 4

Implementing Union Pruning



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Implement Union Pruning

Union Pruning

What is Union Pruning?

Make sure you are aware of the various pruning techniques when using union nodes. There are opportunities to avoid accessing entire data sources that are not required in a union node by using either explicit or implicit pruning rules.

Efficient pruning can improve performance considerably and so is worth investigating.

Explicit Pruning



Prune unions data sources

Explicit pruning

CREATE TARGET

Name: *	temperature
Data Type: *	VARCHAR
Length:	3
Scale:	

MANAGE MAPPING

Source Model	Source Column	Constant Value	is Null
olderData		old	<input type="checkbox"/>
currentData		new	<input type="checkbox"/>

Implicit pruning

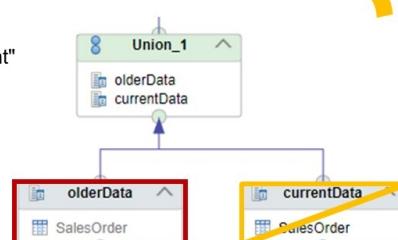
#	SCHEMA	CALC_SCE...	INPUT	COL...	OPTION	LOW_VALUE
1	HDISCHEMA	pruning:tablePrun1	olderData	ChangedAt <		2012-10-19
2	HDISCHEMA	pruning:tablePrun1	currentData	ChangedAt >=		2012-10-19

Each row in the pruning configuration table defines a condition on a column, which all data of the input meets.

Ex: in olderData, the ChangedAt column is always lower than 2012-10-19

```
SELECT
    "BillingStatus",
    SUM("NetAmount") AS "NetAmount"
FROM
    "constantPruning"
WHERE
    "temperature"='old'
```

currentData is pruned



```
SELECT
    "BillingStatus",
    SUM("NetAmount") AS "NetAmount"
FROM
    "tablePruning"
WHERE
    "ChangedAt" < ('2012-09-30')
```

Figure 177: Union Pruning

For each data source in a union, you can define an additional column and fill it with a constant value to explicitly tag the meaning of the data source. For example, plan or actual, hot or cold. If a query does not match the constant value for a specific source of a union, then the source

is ignored. This is called **explicit pruning**. Explicit pruning helps performance by not accessing data sources that are not needed. A constant value is typed in manually to the union node and for every source in the union.

You can also use existing source columns for the constant column. Or for example, if each data source contains only one country and there was an incoming source column called *Country* in all data sources, you could define a constant value for each data source according to the country. Then for each data source in the union, simply specify the country code for each data source.

In the example in the figure, Union Pruning, the union node has a new column defined in the union node Manage Mapping pane, and this column is named *temperature*. For each of the data sources a value is assigned. In this case either *old* or *new*. This is a simple way to filter out sources to unions where the rules are simple and rarely change. Although a union node usually has just one constant column, you can create and map multiple constant columns and use AND/OR conditions in your query. But a key point is that if you don't **explicitly** refer to the constant column in the sending query, then the data source is always included in the union and is not pruned.

Explicit pruning is a very simple way to implement union pruning.

Implicit Pruning

Implicit pruning is implemented by defining one or more pruning rules in a dedicated table called the **Pruning Configuration Table**. This pruning configuration table allows you to describe in much more detail the rules that determine when each source of a union is valid. This implicit pruning approach does not need to provide constant values in the union node for each data source. With a union pruning configuration table, you can define multiple single values per column that are treated with 'OR' logic. You can also define values across multiple columns in a rule. Values across columns are treated with 'AND' logic. This gives much more flexibility over explicit pruning where you can only enter one value per data source. It also means you can easily update the pruning rules as these are simply records in a table.

The first time you create a pruning configuration table, you will need to provide a table name. You then provide the pruning rules using various operators:

- The possible operators are **=**, **<**, **>**, **<=**, **>=** and **BETWEEN**.
- The **BETWEEN** operator refers to a closed interval (including the LOW and HIGH values).
- If several pruning conditions exist for the same column of a view, they are combined with **OR**.
- On different columns, the resulting conditions are combined with **AND**.

As of SAP HANA 2.0 SPS05, the conditions in a Pruning Configuration Table can be defined on columns of the following data types:

- INT
- BIGINT
- VARCHAR
- NVARCHAR
- Date

For example, you could define that a source to a union should only be accessed if the query is requesting the column YEAR = 2008 **OR** YEAR = 2010 **OR** YEAR between 2012 – 2014 **AND** column STATUS = 'A' **OR** STATUS = 'X'.

This means that if the requested data is 2008 and status 'C', then the source is ignored. If the request is for 2013 and status 'A', then the source is valid.

We can use input parameters to direct the query to use only the union sources that contain the data required.

Whichever technique you choose, you are providing the optimizer with important information that it uses when making run time decisions regarding the union sources. This ensures the best possible performance by avoiding processing sources that are not needed.



LESSON SUMMARY

You should now be able to:

- Implement Union Pruning

Using Tools to Check Model Performance



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Use tools to check model performance

Introducing the Performance Analysis mode

When building a calculation view, it is possible to have SAP HANA automatically highlight areas that might lead to poor performance. To do this, you use the Performance Analysis mode. This tool is launched from inside the calculation view editor in the toolbar (the icon looks like a speedometer). You do not need to activate or build a Calculation View in order to use this tool. The basic idea is that you are able to react at **design time** to warnings and other information that is being passed to you that might lead to poor performance so that you can consider alternative approaches.



Note:

Some optimization tools require you to first build the calculation view. This is not true for Performance Analysis mode, which analyzes the design-time object.

Performance Analysis – Detailed Information

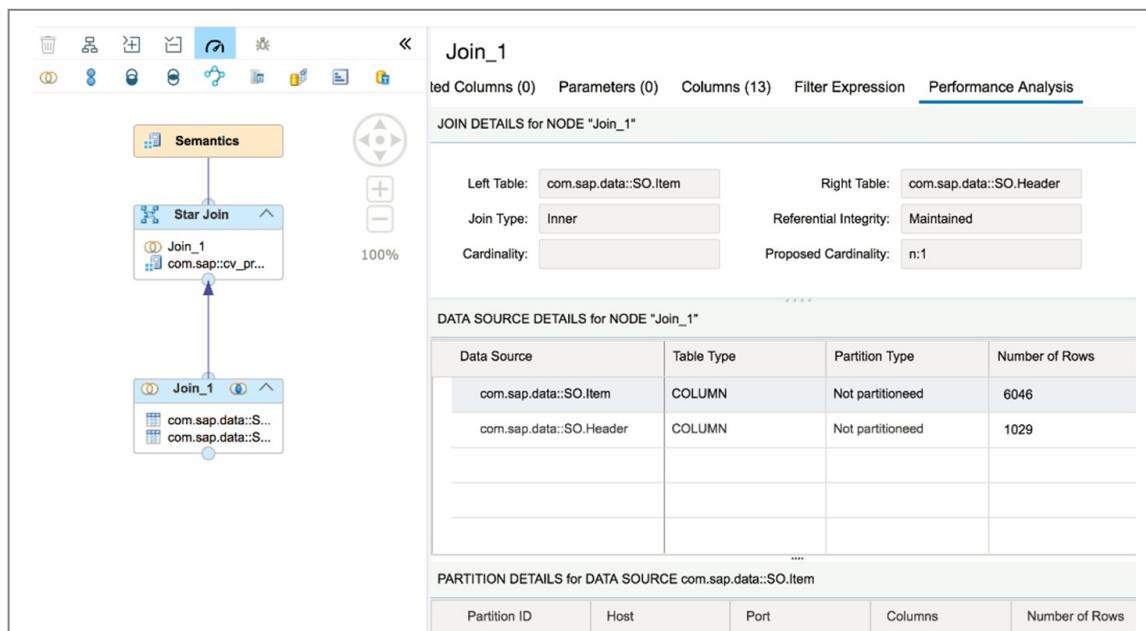


Figure 178: Performance Analysis – Detailed Information

You switch your calculation view to Performance Analysis mode by pressing the button that resembles a speedometer in the toolbar. Once you do this, the *Performance Analysis* tab appears for all nodes except the semantics node. Choose this tab to view detailed information about your calculation view, in particular the settings and values that can have a big impact on performance.

Examples of the information presented on the Performance Analysis mode tab include:

- The type of tables used (row or column)
- Join types used (inner, outer, and so on)
- Join cardinality of data sources selected (n:m and so on)
- Whether tables are partitioned and also how the partitions are defined and how many records each partition contains



Note:

In SAP Web IDE Preferences, you can set a flag that causes all calculation views to open in Performance Analysis mode by default. This can be useful to ensure design violation warnings are not missed by the modeler.

When you leave Performance Analysis mode, the *Performance Analysis* tab disappears from all nodes.

Performance Analysis Validation Warnings

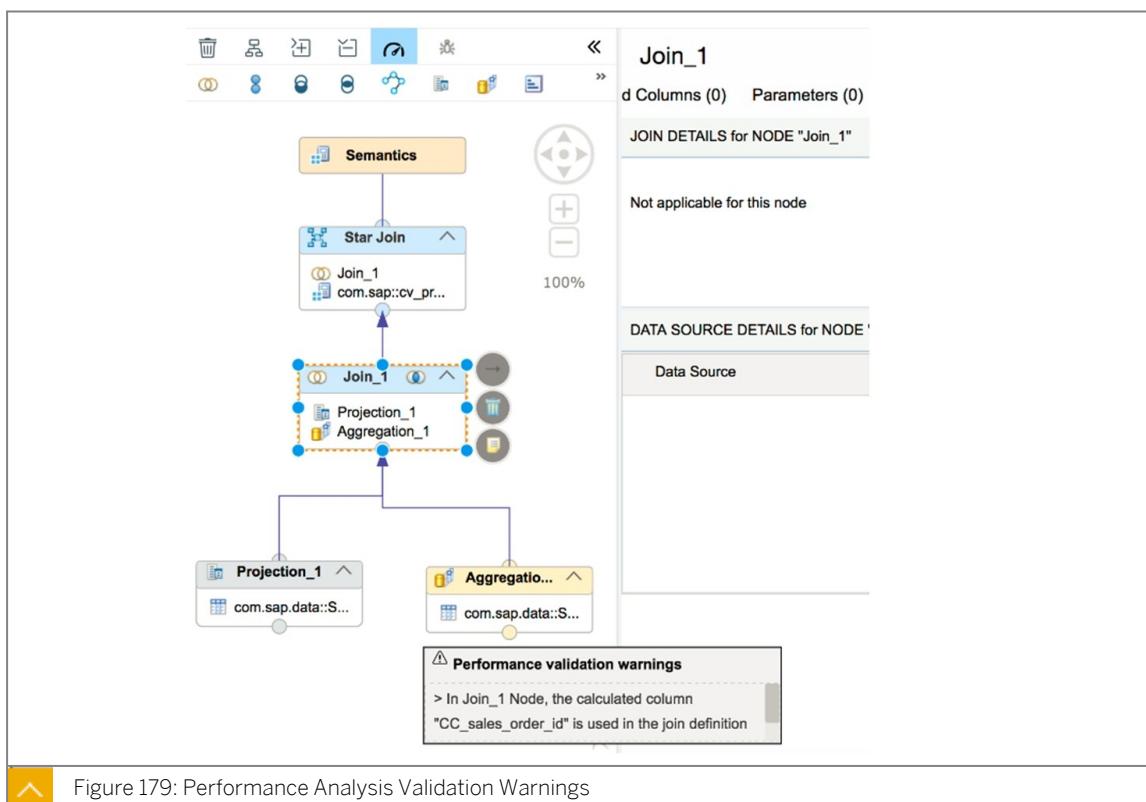


Figure 179: Performance Analysis Validation Warnings

As well as information about the calculation view, Performance Analysis mode also provides you with warnings such as the following:

- The size of tables with clear warnings highlighting the very large tables
- Missing cardinalities or cardinalities that do not align with the current data set
- Joins based on calculated columns
- Restricted columns based on calculated columns

This information enables you to make good decisions that support high performance. For example, if you observe that a table you are consuming is extremely large, you might want to think about adding some partitions and then apply filters so you process only the partitions that contain data you need.



Note:

The tool does not support performance analysis for join view nodes that contain multiple join definitions.

Setting Number of Rows Threshold



Figure 180: Setting Number of Rows Threshold

In the figure, Setting Number of Rows Threshold, you set the threshold in the Modeler preferences of SAP Web IDE.

Introducing the Debug Query Mode

To examine the behavior of a calculation view when it is called by a query, you use the Debug Query mode. In Debug Query mode, you can interrogate each node in the calculation view to see how the SQL is generated.

When you switch to Debug Query mode, a generic query is automatically created to call the calculation view. This generated query simulates a reporting tool or application that is sending a query to the calculation view. The generated query requests all columns of the calculation view and does not have filters. In other words, the generated query requests everything from the calculation view. But you can modify the generated query if you wish, for example, to remove columns or add filters to simulate a user requesting different views of the data.

Once you execute the generated query on the calculation view, you can then view the SQL that is generated for each node of the calculation view and even execute the SQL at each node if you wish to see the interim result for the node. This helps to ensure that your calculation view is behaving as you'd expect, or perhaps to investigate a performance or output issue that has been reported. By stepping through the nodes one by one, examining the SQL, you can check if the expected query is generated. For example, you could see how a WHERE clause is

added to the query when a filter is needed, or check how a PLACEHOLDER is behaving when using an input parameter to pass a value.

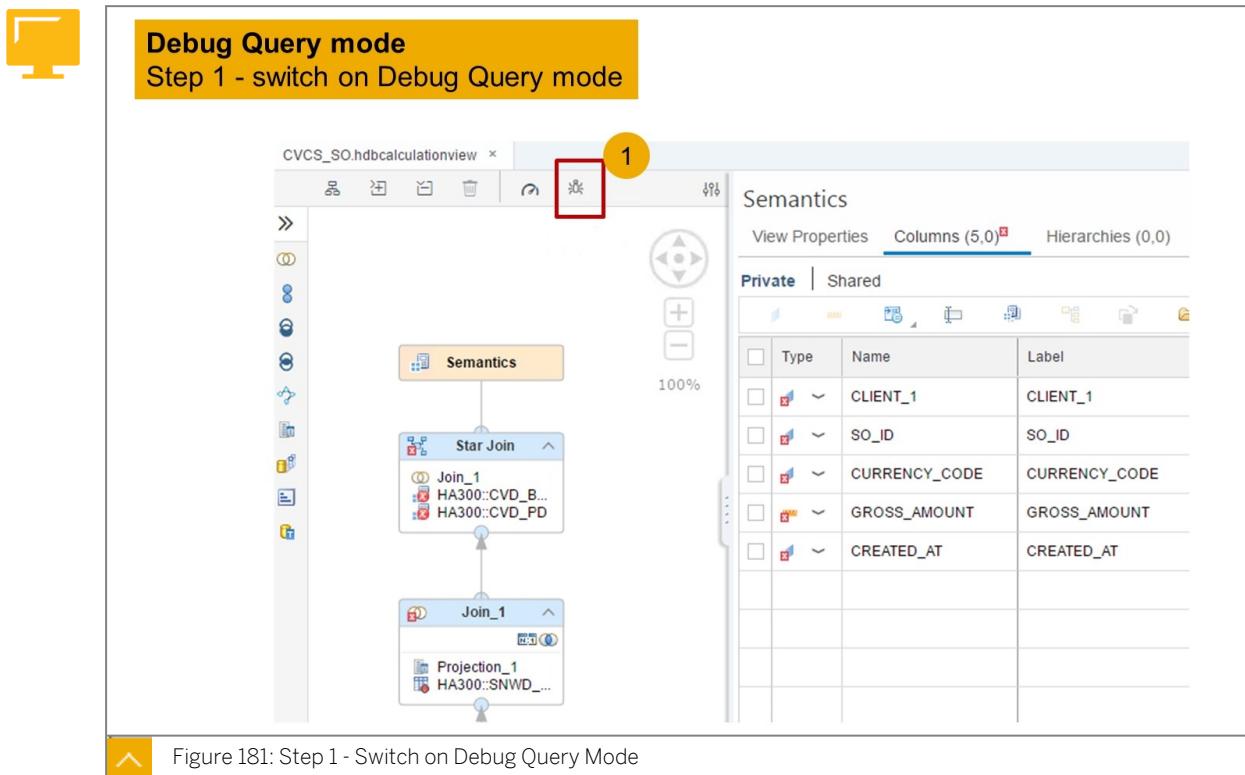
Using Debug Query mode, you can drill down to the lowest level of the entire modeled stack using the graphical editor of the calculation view. This means you do not have to run the Debug Query mode in separate calculation views, but you can start the debugging at the top of the model and continue debugging to the lowest level in the same flow.

One of the most useful things you can discover when using Debug Query mode is to see how each node is pruned of columns and even complete data sources under various query conditions. For example, this mode is a great way of testing if union pruning is working as you would expect by running queries that request different data sets.

Debug Query mode

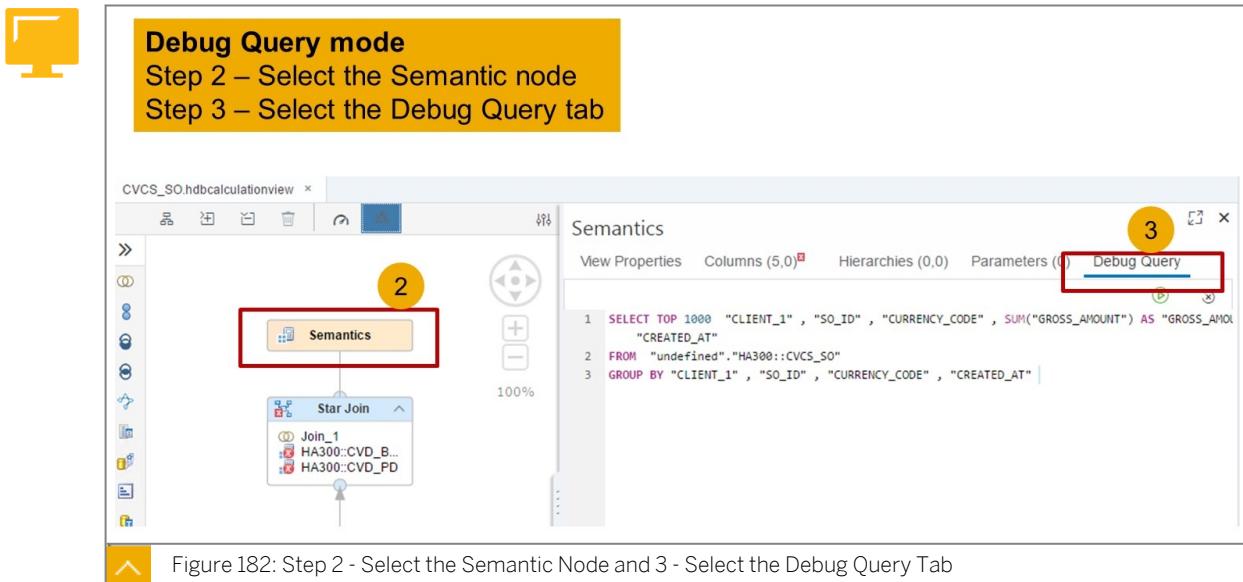
To get started with Debug Query mode:

Open a calculation view, and from the toolbar, choose the button that resembles a bug. This switches the calculation view to Debug Query mode. When a calculation view is in Debug Query mode, it is read-only. This means changes to the calculation view definition cannot be made. If you want to make changes you must switch off Debug Query mode.



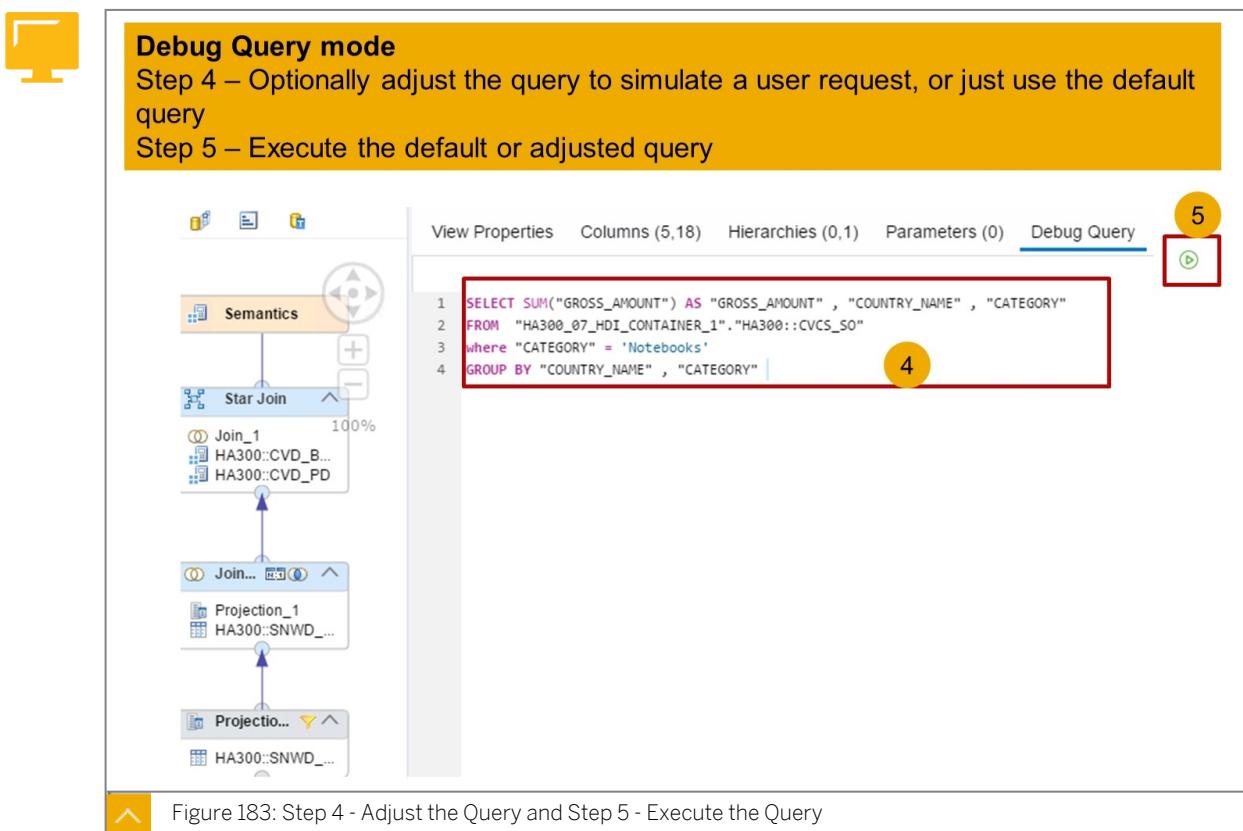
Select the *Semantics* node.

Select the *Debug Query* tab. You will see the generated default query which simulates an unfiltered query coming from a reporting tool that requests all attributes and measures and uses a TOP 1000 constraint just to stop things going too crazy.



You can edit this query if you wish, to simulate a particular query execution from a reporting tool that might pass filters in a WHERE clause or remove some columns you do not need.

To run the query, choose the *Execute* button.



You will now see the generated SQL for each node.

You can now run the generated query, or adjust the query to fine-tune it.

View the results for each node.

Debug Query mode

Step 6 – View the generated SQL at each node
 Step 7 – Optionally execute (or adjust) the generated query at each node
 Step 8 - View result set for the node

The screenshot shows the Data Studio interface with a query tree on the left and a results table on the right. The tree includes nodes like Semantics, Star Join, Join_1, Projection_1, and Projectio... The results table displays a list of countries with their category and gross amount. A yellow box highlights the results table, and numbered circles 6, 7, and 8 point to specific elements: 6 points to the SQL code in the top bar, 7 points to the 'Debug Query' button, and 8 points to the results table.

COUNTRY_NAME	CATEGORY	GROSS_AMOUNT
Germany	Notebooks	1288773.12
France	Notebooks	182324.16
Spain	Notebooks	290930.72
United Kingdom...	Notebooks	131146.88
Mexico	Notebooks	131146.88
Canada	Notebooks	290930.72
Brazil	Notebooks	131146.88
United States of...	Notebooks	342108

Figure 184: Step 6 - View the SQL, Step 7 - Execute the Query at Each Node and Step 8 - View the Result

You can drill down to lower level calculation views, to view the generated SQL at the lower level nodes and also execute the SQL to view node results.

Debug Query mode

Step 9 – Optionally drill down to lower views

The screenshot shows the Data Studio interface with a query tree on the left. A red box highlights a 'Join_1' node in the tree. A red arrow points from this node to a detailed view of the lower-level nodes on the right, which include 'Join_2', 'Join_3', 'Join_4', and 'Join_5'. Numbered circle 9 points to the 'Join_1' node in the tree.

Figure 185: Step 9 - Drill down to Lower Views

Click each node to see how columns have been pruned based on your query. The pruned columns are greyed out. You can also check the push down of filters by checking the SQL to see if the WHERE clauses are present in the lower nodes.



Debug Query mode

Step 10 – Select each node to view the pruning behavior (if you adjusted the default query)

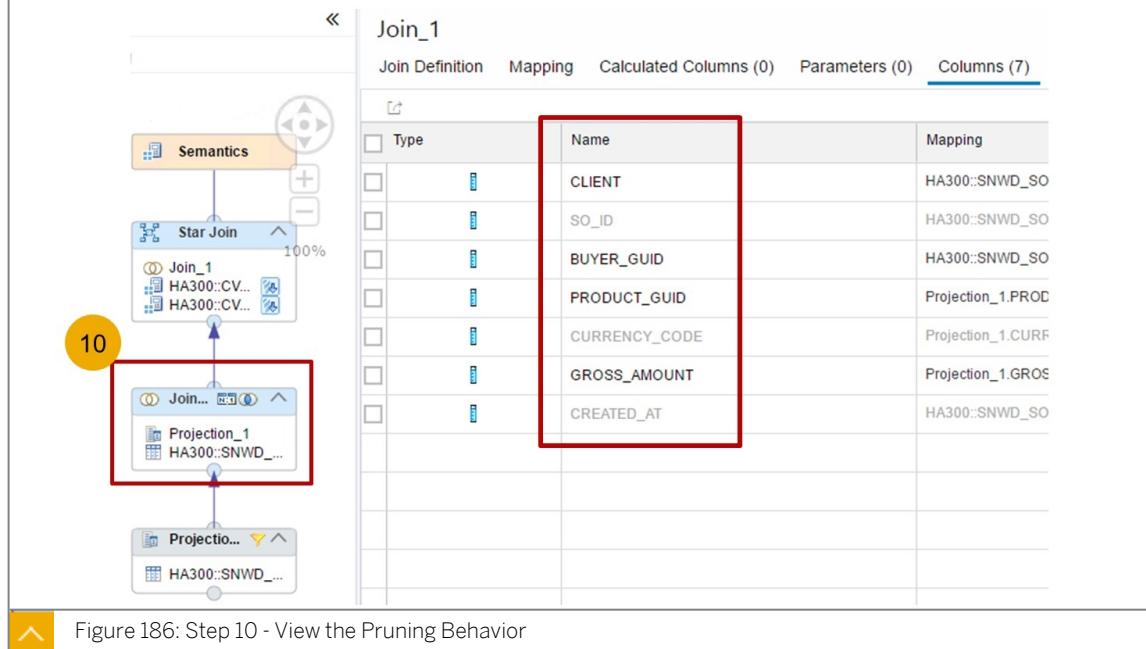


Figure 186: Step 10 - View the Pruning Behavior

You can reset the top level debug query in the *Semantics* node at any time using the reset button, which is adjacent to the *Execute* button.



Note:

Remember that the Debug Query mode calls the **active** version of the calculation view. This means that if you have made changes to the calculation view but not yet rebuilt it, then you switch on Debug Query mode, the debug results will not reflect those latest changes.

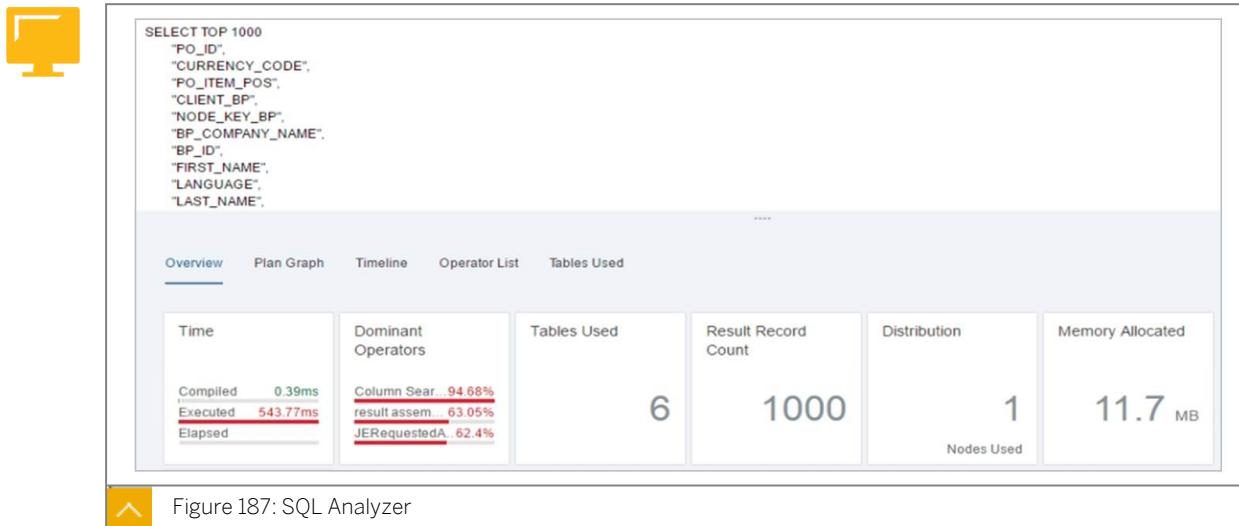
Introducing the SQL Analyzer

Although your calculation view may appear well designed, ultimately it is the performance of the calculation view that really counts. You have seen how to display the generated SQL in a calculation view using the Debug Query mode. While this is helpful to see how filters are pushed down and data is pruned, this does not tell us how the SQL actually performs. We really need to know the answers to the following questions:

- What was the total execution time?
- How long did each step take?
- Are there any steps that are taking significantly longer than other steps?
- Are there any bottlenecks?
- How many records did each step process?
- In what sequence are the steps carried out?
- Which operators are being called?

- To what extent was SAP HANA able to parallelize the query?
- Which processing engines are being invoked?
- Were all the query plan steps unfolded?

To help us learn more about the performance of the SQL, we can use the SAP Web IDE tool **SQL Analyzer**, as shown in the figure, SQL Analyzer.



The SQL Analyzer can be launched from various places in SAP HANA including from the Database Explorer of the SAP Web IDE.

In addition to analyzing the SQL of calculation views, the SQL Analyzer can be used wherever SQL is defined, for example, in procedures or functions.

To get started with SQL Analyzer, take the following steps:

1. Go to the Database Explorer view of SAP Web IDE.
2. Locate the column view that corresponds to the calculation view that you want to analyze. You will find this by expanding the runtime container that corresponds to the project where the calculation view was built and then choose *Column Views*. The column views then appear in the lower pane.
3. From the context menu of the column view, choose *Generate SELECT statement*.
4. Right-click anywhere in the SQL code and select the option *Analyze SQL*. A new tab appears with the SQL analysis results.



LESSON SUMMARY

You should now be able to:

- Use tools to check model performance

Developing a Data Management Architecture



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Implement Good Data Management Architecture

Partitioning Tables

Table Partitioning

Data in column store tables is separated into individual columns to support high performance on queries (query pruning) and also for better memory management (load only required column to memory). But it is also possible to subdivide the rows of column tables into separate blocks of data. We call this **table partitioning**. If column separation is vertical subdivision, think of table partitioning as horizontal subdivision.



Partition tables for better load balancing and performance

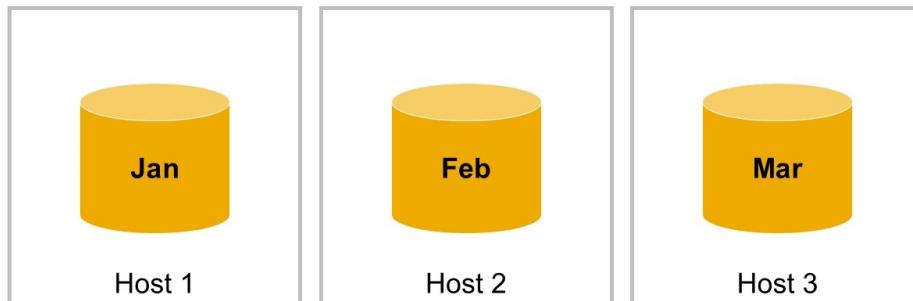


Figure 188: Partition Tables for Better Load Balancing and Performance

Reasons for Partitioning a Column Store

There are many reasons for partitioning a column store table including:

- **Load balancing in a distributed system** – Spread the data equally over nodes so that each server processes a partition in parallel to improve performance
- **Parallelization** – Partitioning allows operations to be parallelized by using several execution threads for each table to improve query performance
- **Pruning to improve query performance** – A query only hits the partitions where the requested data resides. This improves query performance.
- **Overcoming the size limitation of column-store tables** – Non-partitioned tables have a 2-billion row limit, partitions also have a 2-billion row limit but you can define up to 16,000 partition per table.

**Note:**

Partitioning tables also helps with more efficient delta merge management as you can perform merging on partitions.

Table partitioning is typically used in multiple-host systems, but it may also be beneficial in single-host systems especially on very large tables where only small sets of data are requested from large tables. Table partitioning is typically evaluated in the wider context of overall SAP HANA data management. Often modelers will not be responsible for creating and maintaining partitions but their input is incredibly important to the decisions being made about partitioning. That is why modelers must be aware of partitioning.

Table Partitioning

SAP HANA supports three types of table partitioning:

- **Range** – Define partitions explicitly using single values or ranges based on values found in one or more columns. For example, 2010–2013, 2014–2017, 2018. You need to know the table data very well to determine the best columns to use. The table can have primary key or no primary key. If the table has a primary key, then the partitioning columns must be part of that primary key.
- **Hash** – Allow SAP HANA to build partitions; you simply choose one or more columns on which SAP HANA computes the hash values, and SAP HANA distributes the data evenly. Hash partitioning does not require an in-depth knowledge of the actual content of the table. The table can have a primary key or no primary key. If there is a primary key, then all partition columns must be part of that key.
- **Round Robin** – New rows are assigned to partitions on a rotation basis. You do not specify the columns to use. The table must not have primary keys.

For all these partitioning types, the number of partitions can be either explicitly defined or can be automatically determined based on the number of configured servers that host the database.

**Note:**

Be careful not to confuse table partitioning with table distribution. The latter is where **entire tables** are carefully distributed across hosts for better load balancing. Table partitioning is where we distribute the data from one table across partitions in single or multiple hosts systems.

How do we partition a column-store table? There are a number of methods to create table partitions:

- SQL statement
- CDS definition

Specifying table partitions with SQL



Specifying table partitions with SQL

```
CREATE COLUMN TABLE MY_TABLE (a INT, b INT, c INT, PRIMARY KEY (a,b))
PARTITION BY RANGE (a)
(PARTITION 1 <= VALUES < 5,
PARTITION 5 <= VALUES < 20,
PARTITION VALUE = 44, PARTITION OTHERS)
```



Figure 189: Specifying table partitions with SQL

Table partitions can be defined, altered, and deleted using SQL.

Specifying Table Partitions with CDS



Specifying table partitions with CDS

```
entity MyEntity {
    key id : Integer;
    a : Integer;
    b : Integer;
    t : String(100);
} technical configuration {
    partition by hash (id) partitions 2,
        range (a) (partition 1 <= values < 10,
                    partition values = 10,
                    partition others);
}
```



Figure 190: Specifying Table Partitions with CDS

Table partitions can also be defined using Core Data Services (CDS). This is the recommended approach because the partitioning information is saved in a source file inside a Web IDE project. It can then easily be transported and redeployed many times.

Distribution of Records over Partitions

Partitioning is transparent for SQL query definitions. This means the SQL code does not refer to partitions. However, the key purpose of partitioning tables from a modeler perspective is to improve performance of the queries. Knowing how a table is partitioned is essential for the modeler. With this knowledge they can then carefully design calculation views, functions, or procedures to exploit the way the partitions are formed. For example, a modeler that is aware of partitions should ensure that filters are applied as early as possible on the partitioned column. This means that only data for the partition is loaded to memory.

Partitions are usually defined when a column-store table is created, but it is also possible to partition a column-store table that was created but never partitioned. You do not have to drop the table to create partitions. You can partition an existing table that is empty or already contains data.

As well as creating partitions, it is also possible to:

- Re-partition an already partitioned table, for example, to change the partitioning specification (hash to round-robin and so on) or to change the partitioning columns or to increase or decrease the number of partitions.
- Merge partitions back to one table.
- Add/delete individual partitions.
- Move partitions to other hosts.

Multi-Level Partitions

An easy way to spot if a table is partitioned is to observe the icon alongside the table name. You will see that the icon looks similar to a column table icon but the columns are separated in a partitioned column table. You can also open the *Runtime Information* tab of the table metadata, and in the lower part of the screen you can click the *Parts* tab and expand the table node to see the partitions (if they exist).

Finally, we should briefly mention **multi-level partitions**. With multi-level partitions, you can create additional partitions on each partition. It might help to think of this as a hierarchy of partitions. This is typically used when you first want to distribute large tables over multiple hosts (use hash partitioning to load balance), and then for each of those partitions you apply the next level of range partitions (break up by year). So now you have host/year partitioning. Also, multi-level partitioning can be used to overcome the limitation of single-level hash partitioning and range partitioning, that is, the limitation of only being able to use key columns as partitioning columns when you are dealing with keyed tables. Multi-level partitioning makes it possible to partition by a column that is not part of the primary key.



Note:

You cannot define partitions on a row-store table



Note:

We have covered only the very basics of table partitioning here. Refer to the official SAP Help documentation for full information.

Data Tiering

What is Data Tiering?

In pursuit of the best performance for analytics, modelers should be aware of all the features provided by SAP HANA that can help to achieve this aim. Data management architecture can have a significant impact on the performance of analytics.

We know that SAP HANA is an in-memory database and data processing platform. We have an opportunity to keep more and more data in memory than ever before. It is possible to size hardware so that even the largest enterprise databases could be stored completely in memory. However, it does not make sense to store old, infrequently accessed data in expensive memory. Memory should not be used as a data archive and should be reserved for active data where instant access is needed. There are better solutions for managing data that is infrequently accessed, at a much lower cost than memory, but still providing good read performance.

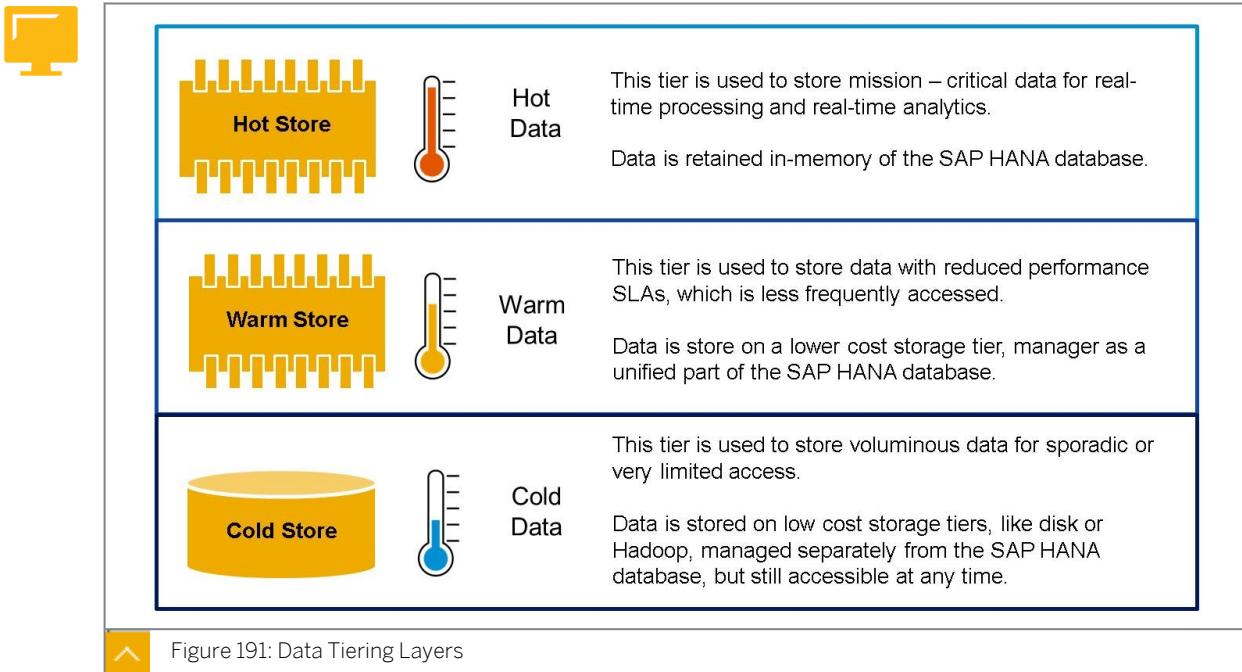


Figure 191: Data Tiering Layers

We classify data into temperatures. Typically three temperatures are used: hot, warm and cold.

Hot data requires super-fast access but also comes at the highest cost. Warm data loses some of the performance but the cost reduction can be significant. Finally, cold data provides access speeds that might only be acceptable in some scenarios, but usually the storage capacity is huge and costs are relatively low.

SAP provides technical solutions to manage data at all three layers. The layering of data is known as data tiering, with each tier representing a data temperature. For all three temperatures, there are multiple technical solutions available within SAP HANA and each solution has its own strengths and weaknesses.

Tiering Solutions

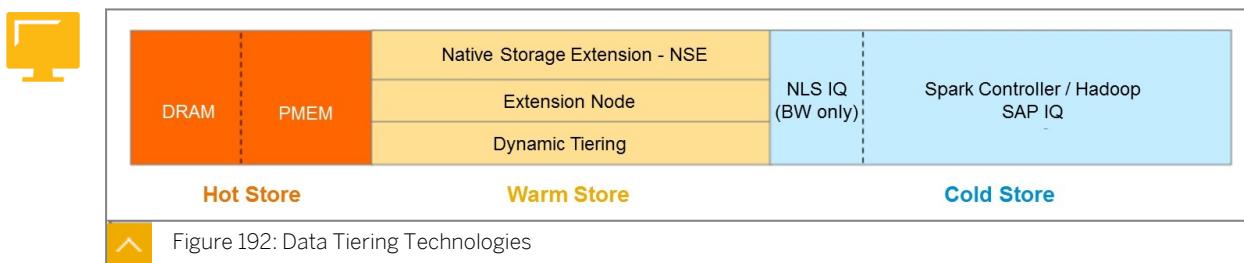
Hot data is managed in memory. However, with the release of SAP HANA 2.0, SPS04, a new type of memory called persistent memory (PMEM) is available to provide non-volatile storage. PMEM sits alongside the existing volatile memory (DRAM) and can provide much larger capacities than DRAM and at a cheaper cost than DRAM. PMEM also enables fast database restarts because data does not have to be reloaded from disk to memory and because data never leaves memory during a restart. The downside of PMEM is that it is not as fast as DRAM but it still provides acceptable read access response and is still considered part of the hot store tier. Customers are encouraged to size their memory using a combination of DRAM and PMEM to create a hot tier memory-based solution. A modeler does not need to concern themselves with the differences between DRAM and PMEM when developing calculation views as the allocation of data between DRAM and PMEM is managed internally by SAP HANA and a modeler does not direct the calculation view to use PMEM or DRAM. The main interest for a modeler is that a combination of DRAM and PMEM means very large memory is available and that means more data can be classified as hot so even larger data volumes can be processed by calculation views whilst maintaining good performance. This might mean modelers might relax their filtering rules to provide access to more data for business users, knowing that performance would be acceptable.

For **warm data** there are three possible solutions:

- Native Storage Extension (NSE) is the latest warm data storage solution to be offered by SAP (as of SAP HANA 2.0 SPS04) and should be considered before the other warm data solutions, which for now, continue to be offered. NSE uses disk to store tables that are less frequently accessed, or where very fast read performance is not necessary. NSE is part of the native SAP HANA database and shares the same persistence layer used by memory for unloading tables when memory becomes full and also for logging/backups. When tables are first created they can be assigned to NSE either at the full table level, individual column level, or at a partition level. NSE uses an intelligent buffer in memory to move pages of data between disk store and memory.
- Extension Node solution makes use of a worker node in a scale-out landscape to store warm data in memory. Memory sizes of the worker node are relaxed compared to the primary node and this means that more data can be stored in memory. Currently, this warm data solution provides the fastest read access to data (data comes from memory not disk), but the downside it requires more hardware resources and therefore increases cost.
- Dynamic Tiering uses a separate column store disk based database that works side-by-side with the SAP HANA in memory database. Warm data is stored primarily on the disk component. Similar to NSE, when tables are created they can be assigned to disk. A key strength of this solution is it is currently the largest volume warm data solution available. The downside is that Dynamic Tiering is the slowest of the warm storage solution. Similar to NSE, a single table can be assigned to memory or disk based on individual columns and partitions. Tables that are split in this way using Dynamic Tiering are called *multistore* tables.

For all three warm store solutions, the modeler should simply be aware of the tables, their columns and partitions that are classified as warm store and consider using techniques such as developing filters, pruning rules etc to avoid unnecessary warm store access in order to maintain very high model performance.

Cold data is managed using a separately installed Spark Controller on Hadoop (or other big data storage solution). The cold storage solution manages data that is infrequently accessed or where read performance does not have to be optimal. Benefits are huge data volumes are possible, but the downside is the cost of setting up the infrastructure and also the speed of data retrieval. But for many organizations whose data volume growth is a concern, this is a good solution to ensure that hot, and especially warm data stores, do not become data archives. As with warm data, the modeler simply needs to be aware of the tables that are managed in cold store so that they can apply the modeling techniques that mean they avoid unnecessary access to tables that are not needed.



Considerations for a Data Modeler

Why should a data modeler care about data management?

Although it is unlikely that a data modeler will be involved in the technical implementation of the tiering solutions, the decisions made around how data is tiered might have an influence on the design of their data models. For example, if the modeler is aware that historical data is

managed in the cold store, they might implement union pruning rules to ensure the cold store is not accessed unless needed by the query. This will ensure that the query does not scan data sources in archives when it is known that the requested data does not reside there.

Even before data models are created, a well-thought-out data management tiering architecture is an essential foundation on which to build. Modelers are important stakeholders in the design of the data management architecture and can heavily influence the solution with their detailed knowledge of data modeling within SAP HANA.



LESSON SUMMARY

You should now be able to:

- Implement Good Data Management Architecture

Learning Assessment

1. Why would you define a pruning configuration table?

Choose the correct answer.

- A To define the conditions that determine when a complete data source can be pruned from a union.
- B To define the optimal path that should be used to prune data sources from a union .

2. Which type of node is used to end the parallelization of a data flow?

Choose the correct answer.

- A Projection
- B Union
- C Aggregation

3. To work with calculation view debug query, you first need to build the calculation view?

Determine whether this statement is true or false.

- True
- False

4. What is the purpose of the SQL Analyzer?

Choose the correct answers.

- A To identify the longest running SQL statements.
- B To investigate the generated SQL for each node in my calculation view.
- C To highlight syntax errors in my SQL code.

5. Which are valid table partition types in SAP HANA?

Choose the correct answers.

- A Range
- B Hash
- C Round Robin
- D Column

6. To which storage tier does Native Store Extension (NSE) belong?

Choose the correct answer.

- A Cold
- B Hot
- C Warm

Learning Assessment - Answers

1. Why would you define a pruning configuration table?

Choose the correct answer.

- A To define the conditions that determine when a complete data source can be pruned from a union.
- B To define the optimal path that should be used to prune data sources from a union .

Correct — A pruning configuration table is used to define the conditions that determine when a complete data source can be pruned from a union.

2. Which type of node is used to end the parallelization of a data flow?

Choose the correct answer.

- A Projection
- B Union
- C Aggregation

Correct — A union node with only one source must be used to end the parallelization of a data flow.

3. To work with calculation view debug query, you first need to build the calculation view?

Determine whether this statement is true or false.

- True
- False

Correct — You first need to build the calculation view to use the Query Debug mode. The Query Debug mode calls the runtime object of the calculation view and that is only available when you first build the calculation view.

4. What is the purpose of the SQL Analyzer?

Choose the correct answers.

- A To identify the longest running SQL statements.
- B To investigate the generated SQL for each node in my calculation view.
- C To highlight syntax errors in my SQL code.

Correct — You can use SQL Analyzer to identify the longest running SQL statements and many other issues that might contribute to poor run-time performance, such as use of cache, query unfolding, etc. You use the Query Debug mode to investigate the generated SQL for each node in your calculation view. Syntax errors in SQL code are highlighted in the SQL Console of Web IDE.

5. Which are valid table partition types in SAP HANA?

Choose the correct answers.

- A Range
- B Hash
- C Round Robin
- D Column

Correct! — Range, Hash, Round Robin are valid table partition types. Column is not a table partition type. For more detail see HA300 Unit 6 Lesson 3

6. To which storage tier does Native Store Extension (NSE) belong?

Choose the correct answer.

- A Cold
- B Hot
- C Warm

Correct! — Native Store Extension (NSE) belongs to the warm storage tier. For more details see HA300 Unit 6 Lesson 3

UNIT 7

Management and Administration of Models

Lesson 1

Working with Modeling Content in a Project

285

Lesson 2

Creating and Managing Projects

313

Lesson 3

Enabling Access to External Data

321

Lesson 4

Working with GIT Within the SAP Web IDE

327

Lesson 5

Migrating Modeling Content

349

UNIT OBJECTIVES

- Analyze and document information models
- Explain the structure of a project
- Build modeling content
- Modify and move modeling content
- Translate Descriptions
- Define the key settings of a project
- Manage the lifecycle of a project
- Set up access to external data
- Use the Native Git Integration of the SAP Web IDE
- List the deprecated modeling artifacts
- Explain how to migrate modeling content

Working with Modeling Content in a Project

LESSON OVERVIEW

In this lesson, you will learn about several features related to the lifecycle of an information model, such as validation, version comparison, documentation, and performance analysis.

Business Example

During the development phase of your project, you are involved in troubleshooting of some *AnalyticViews*.

You want to use different types of model validation rules to narrow down the list of issues and perform a more specific analysis.



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Analyze and document information models
- Explain the structure of a project
- Build modeling content
- Modify and move modeling content
- Translate Descriptions

Auditing Dependencies Between Information Models

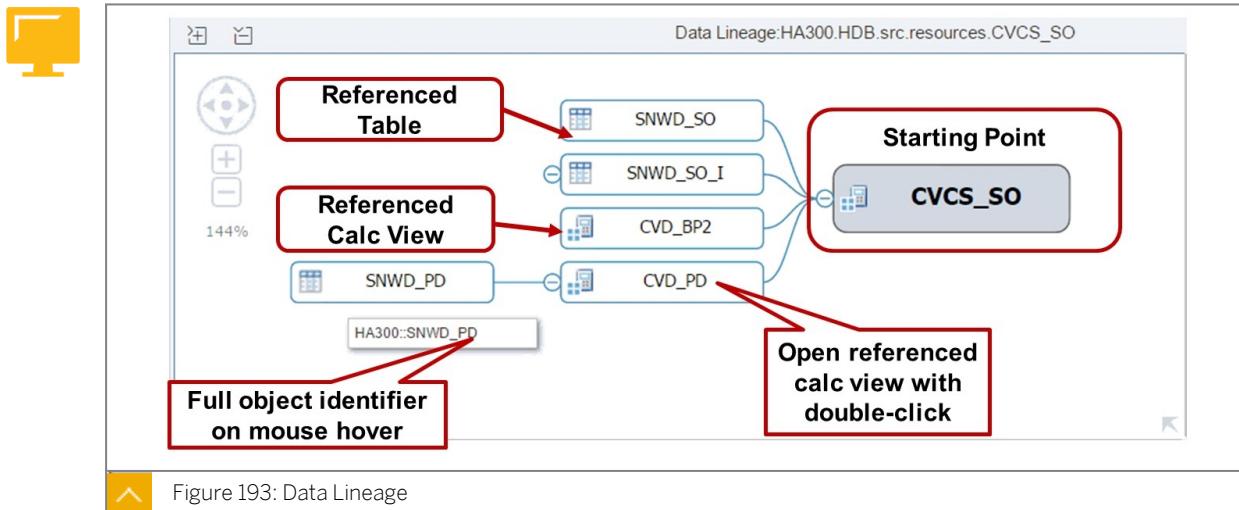
Two features are available in the SAP Web IDE to analyze modeling content within a project. These are:

- Data lineage
- Impact analysis

They work in a symmetrical way. For any calculation view you choose, you can show the dependencies with other modeling content.

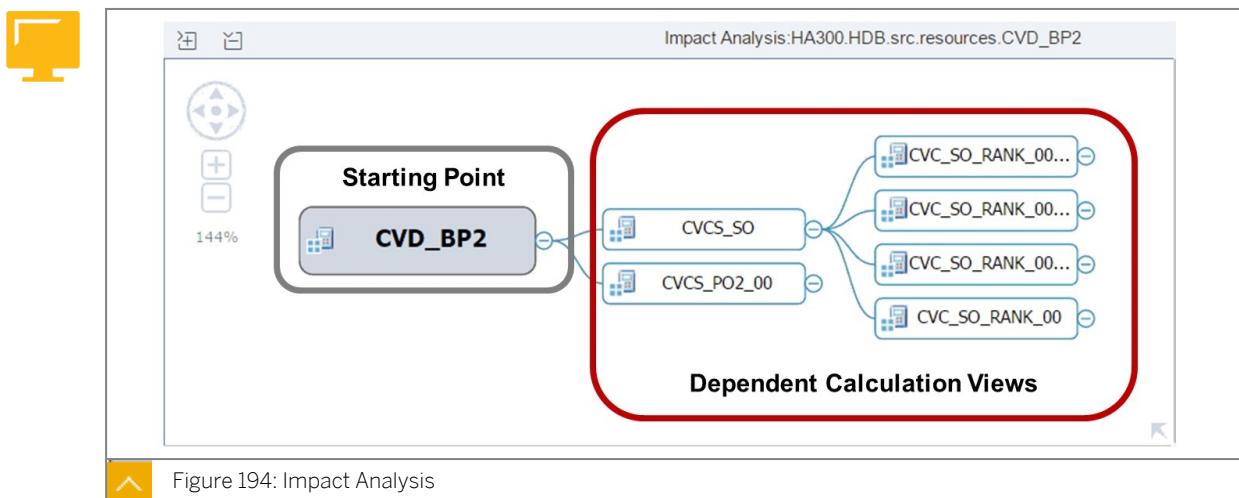
To use these features, right-click a calculation view and choose *Other Action → Data Lineage* or *Other Action → Impact Analysis*.

Data Lineage



Data lineage shows all the calculation views and source tables on which a given calculation view depends.

Impact Analysis



The purpose of impact analysis is to show all the chain of calculation views that depend on a given calculation view.



Hint:
From the *Data Lineage* or *Impact Analysis* view, you can directly open a calculation view from the dependency tree by double-clicking it.

Tracing the Origin of a Column in a Calculation View Scenario

Another powerful auditing feature is available within a calculation view to show the origin of a column.

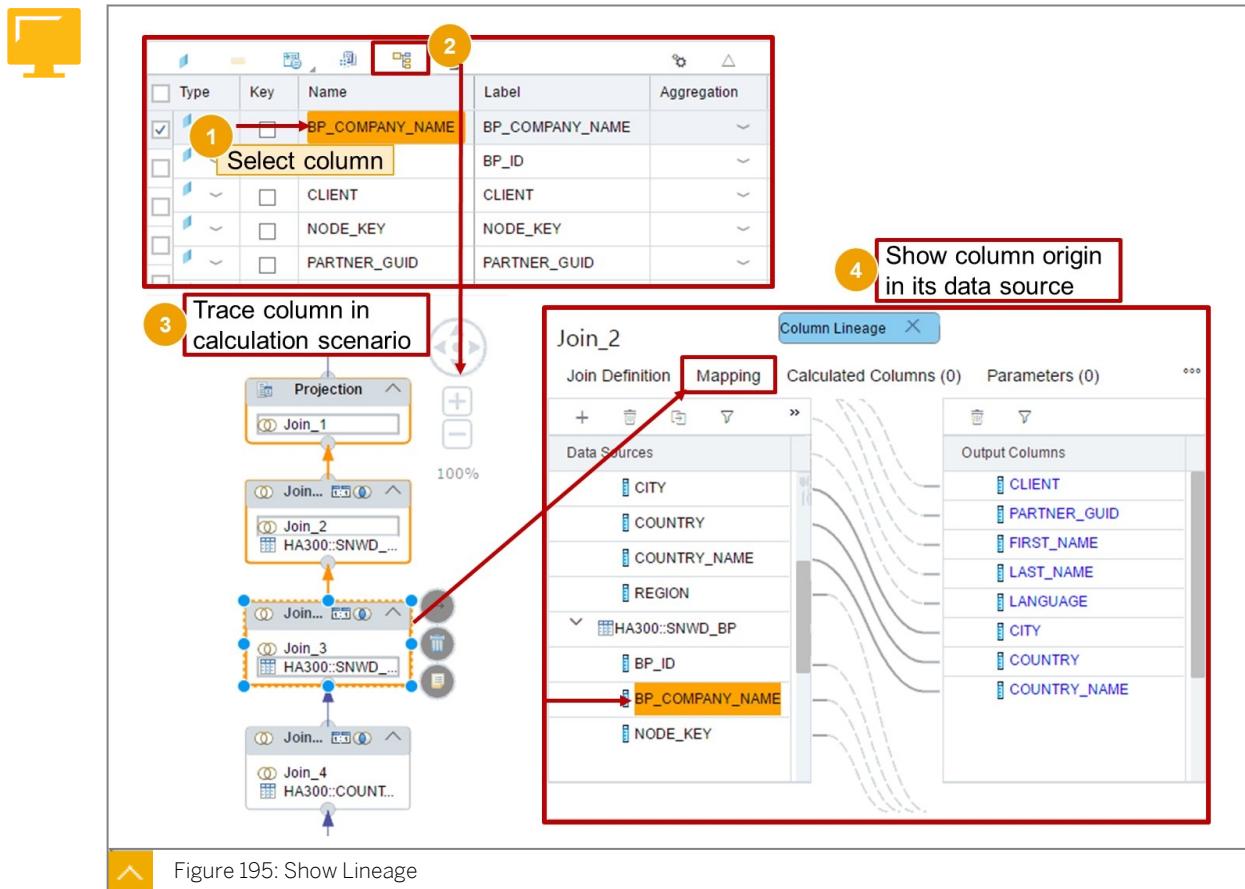


Figure 195: Show Lineage

From the Semantics node, you can choose a column and trace its origin with *Column Lineage*.

The column lineage shows, within the scenario of the opened calculation view, all the nodes where this column exists. At the bottom of the calculation view scenario, you find the origin of the column. Opening the *Mapping* tab of this node allows you to identify the source column name in the data source.



Note:

In a cube with star join calculation view, the *Show Lineage* feature only works for private columns, not shared columns from dimension calculation views.

Column Lineage

Column lineage is very useful when columns are renamed within the calculation scenario, or when you want to see quickly where a calculated or restricted column originates from.

It also helps you to avoid mixing up columns with the same name, but not necessarily the same data that are present in several nodes of a calculation view.

For example, the *NODE_KEY* column in an SAP ERP system is used in many tables to join the master data-defining attributes. You might want to make sure that the *NODE_KEY* column in the output of a dimension calculation view originates from the correct table (for example, the table *SNWD_BP* containing business partners) and not from another table, also used in the calculation view, in which a *NODE_KEY* column is present but does not identify business partners (for example, the *NODE_KEY* in the table *SNWD_AD* containing Contact addresses).

The Outline View

The *Outline* view offers another way to navigate the structure of a calculation view and jump easily to a given column in a given node. It is accessible in the toolbar on the right of the Web IDE

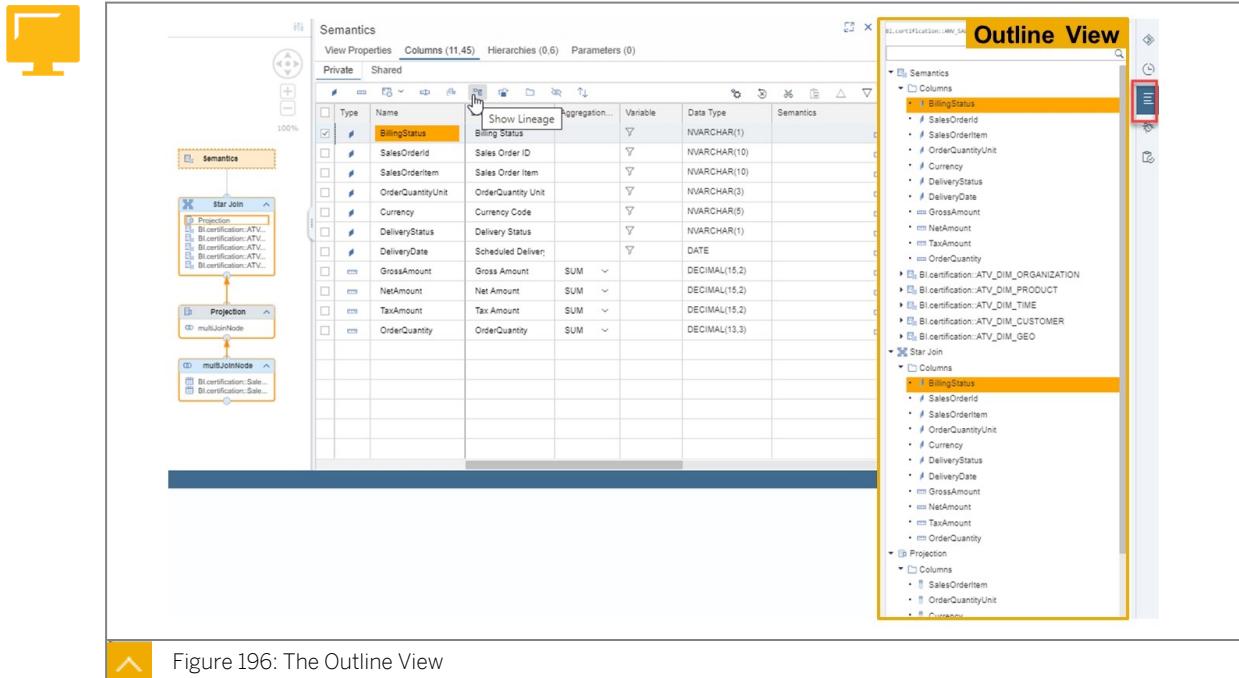


Figure 196: The Outline View



Note:

As of HANA 2.0 SPS05, the *Outline* view also highlights the relevant columns when column lineage is active.

Deprecated Calculation Views

In the SAP Web IDE for SAP HANA, it is possible to flag a Calculation View as **deprecated**. To do this, in the *Semantics* of the Calculation View, display the *General Properties* tab and check the *Deprecate* checkbox.

The purpose of this *Deprecate* flag is to identify Calculation Views that are still valid from a technical perspective, but should not be used, for a variety of reasons: for example, because they are replaced with newer ones that are designed in a way that provides better performance, and you want to make modelers aware that they should not be used any longer in consuming views.

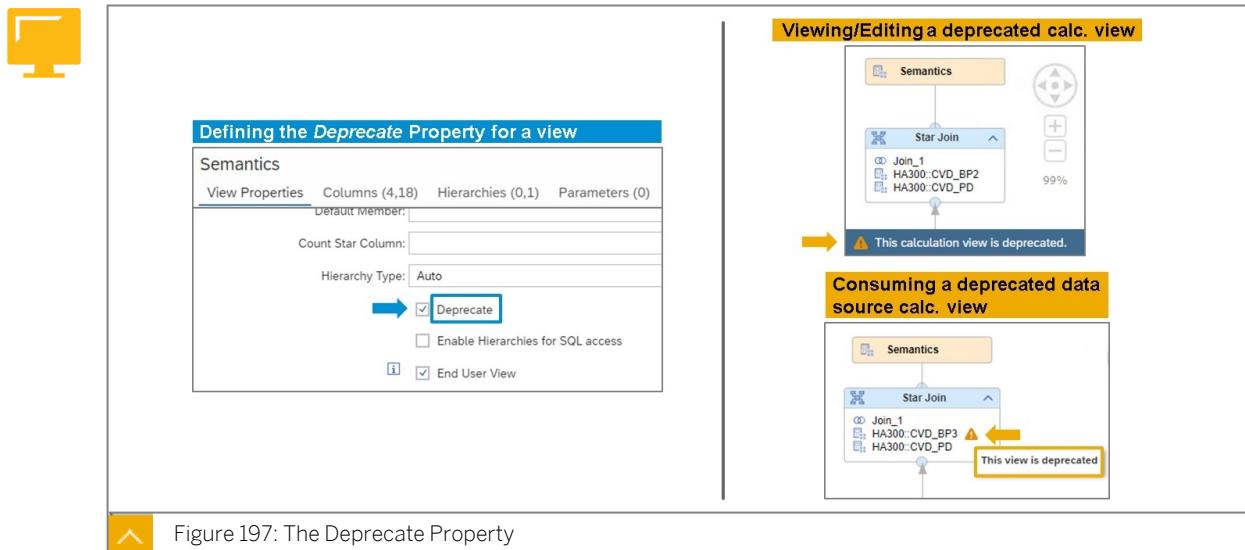


Figure 197: The Deprecate Property

When viewing and editing Calculation Views, the *Deprecate* property of the open view and/or any of its underlying data sources is materialized graphically.



Caution:

The *Deprecate* property of a view has NO impact on the way the view is exposed to the end-user and front-end tools. In particular, a Calculation View that is flagged as *Deprecated* returns the same data, columns, can be accessed by the same users, and so on.

As an aside, the flag is NOT included in the SAP HANA Analytical Catalog, materialized by the *BIMC_** views and tables. So the *Deprecate* feature cannot be used as a replacement for a proper view access through *SELECT* privileges.

Workspace, Projects, and Modules

The workspace in the SAP Web IDE is a structure where you can work on one or several projects. Each user of the SAP Web IDE for SAP HANA has their own workspace.

The workspace of a user can contain as many projects as needed, but each project must have a different name.

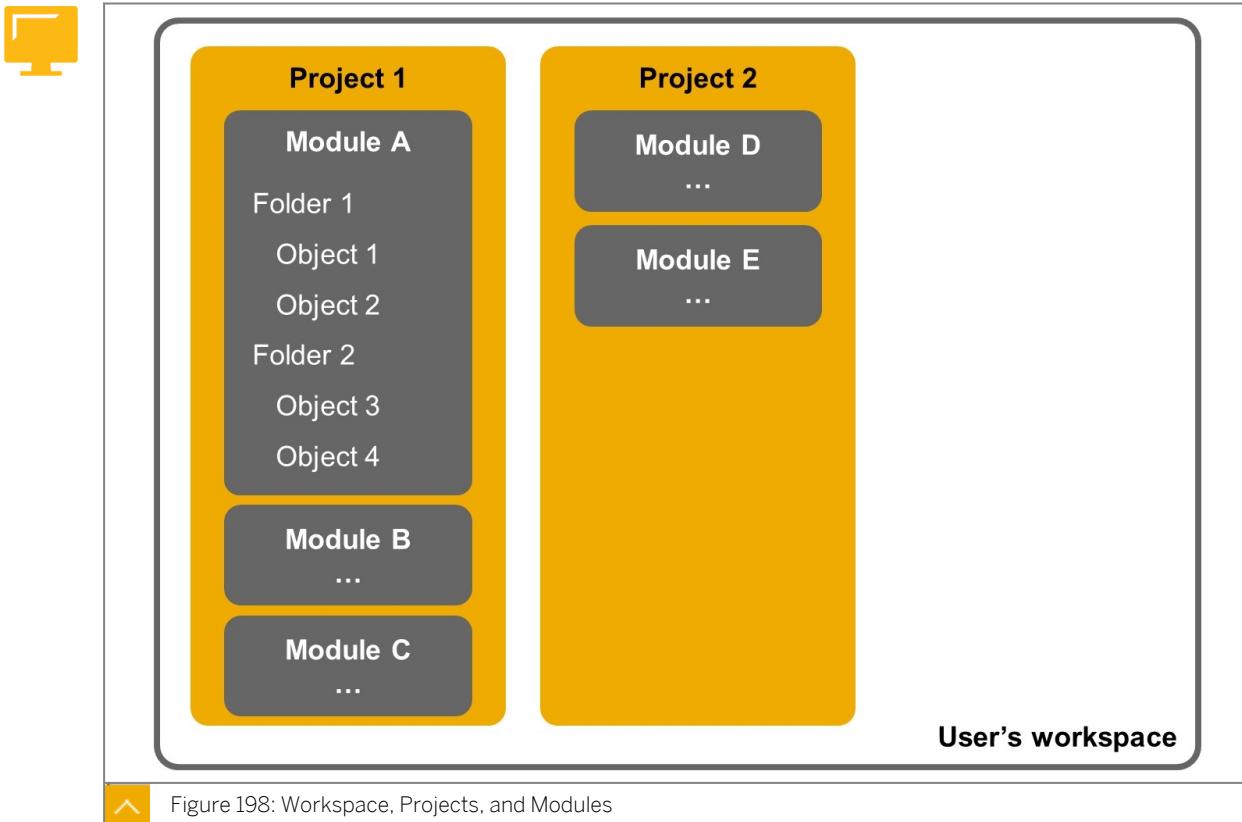


Figure 198: Workspace, Projects, and Modules

When you create a new project, the creation wizard will ask you to choose a project template. The Web IDE for SAP HANA comes with several templates and you choose the one most suitable for the type of application you want to create. Depending on the template you choose, you will be prompted for various settings which are then used to create the very basic project shell.

In this course, we focus on the Multi-Target Application (MTA) project template. An MTA is comprised of multiple components (modules) created with different technologies and deployed to different targets, but with a single common lifecycle.



Note:

The SAP HANA Database Application template could also have been chosen, and actually, this template automates a few of the steps needed to fully prepare the project to begin development, specifically of the database artifacts. The MTA template does not automate these steps and you are left to work through these manually. The outcome is the same, but we felt that rather than automating the steps, we would show you later how to work through these steps manually so you would learn more about the setup of a project.

Main Types of Modules

The source code for your application is stored inside various modules of your project.

The first step is to create a module of the type you require and then begin developing the source code in that module. You can create as many modules as you need for your project, for example, you may choose to create two database modules: one for tables and one for views, if that helps with better organization of source code. You should provide a helpful name for the

module so that it is easy to identify the module type and what it contains. Do not create empty modules that you will not use.

Table 19: Main Types of Modules

Module Type	Key Role in Application
SAP HANA Database (HDB) module	Definition of database objects that provide persistence layer, virtual data model, data access, data processing (functions, procedures)
Java module	Business logic
Node.js module	Business logic
Basic HTML5	User interface (UI)
SAPUI5 module	UI based on the SAPUI5 standard
SAP Fiori launchpad site module	UI: SAP Fiori launchpad site with entry point (host) and site's content and configuration
SAP Fiori master-detail module	UI: Typical split-screen layout (one of SAP Fiori design patterns)

In this course, the focus is exclusively on the HDB module.



Note:

The other types of modules are introduced in another course, *HA450 – Application Development for SAP HANA*.

HDB Modules

A HDB module contains all the design-time files that define the database objects that must be deployed together with the application.

Main Database Artefacts Defined in a HDB Module



- Tables
- Table data
- Calculation views
- CDS views
- Table functions
- Procedures
- Analytic privileges
- Synonyms

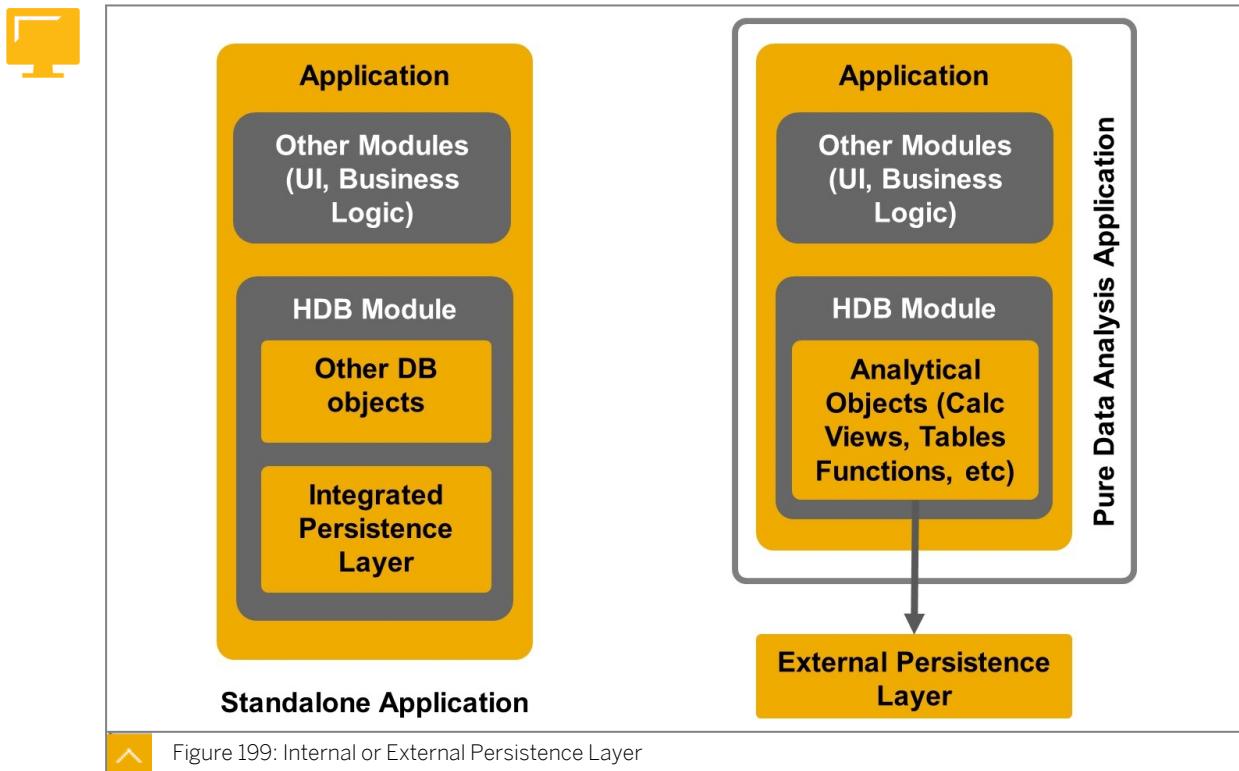
A project can contain several HDB modules. For example, the SHINE for XSA application (SHINE stands for SAP HANA Interactive Education) is comprised of several modules of different types (HDB, node.js, and so on), including two HDB modules.

Persistence Layer: Inside the HDB Module or Outside?

Depending on the type of application, the main data managed by the application can be stored in different places.

A major design option for an application is to decide whether this data is stored in a persistence layer defined by the application itself, or if it is stored in another location, such as a classical database schema. Let's consider the two following examples to illustrate these approaches:

Internal or External Persistence Layer



- **Standalone Application**

With the SAP Web IDE and XSA-based application development, it is very easy to define the entire set of components of an application, including the persistence layer. With this approach, this persistence layer will be defined in the HDB module, mainly by tables defined as CDS entities.

- **Pure Data Analysis Application**

In this scenario, the data sources are not defined by the application itself, but located elsewhere, for example in a classical database schema. A number of application artefacts defined in the HDB module, such as calculation views, table functions, and so on, consume these external data sources. For example, when building a virtual data model on top of an existing application such as an ERP, the data persistence layer is managed by the ERP application layer, so you do not define it in your reporting application.

**Note:**

There is sometimes a lightweight persistence layer built inside the application, in order to manage some application-related information, such as metadata associated with the virtual model itself, or information on the way users consume the views. For example, the history of views queried by each user, or a favorite flag/personal flags that users can assign to the views they use on a regular basis.

Objects Identifier and Namespace

In XS Advanced, each runtime object has an object identifier, which is used to reference this object within the container/schema. To provide a simple example, if the calculation view CV1 references the calculation view CV2, the execution of CV1 will trigger a “call” to view CV2 by using its object identifier.

A typical structure of a runtime object identifier is as follows:

```
<Namespace>::<Runtime Object Name>
```

- The **Runtime Object Name** is mandatory.
- The **Namespace** is an optional part of the object identifier, and is mainly used to organize logically the runtime objects within the container.

When you work with XS Advanced projects, the naming of runtime objects is distinct from the way design-time files are organized. The main advantage of this approach is that you can more easily relocate design-time objects from one folder to another without impacting the corresponding runtime object name.

**Note:**

In this course, the namespace option we have chosen for the main *HA300_##* application is to have a namespace prefix **HA300**, applied uniformly to all objects, regardless of the folder location of design-time objects. In other words, the namespace is the same for all the runtime objects defined in our HDB module. So, these objects all have the identifier pattern **HA300 : <Runtime Object Name>**.

Runtime Objects Identifiers

The object identifiers (including the namespace) are always specified in the design-time objects. That is, the design-time objects must define without ambiguity how the runtime objects will be identified.

Let's take an example of a modeling object, a calculation view *CVC_SalesOrders*, with its design-time version in the SAP Web IDE workspace, and its runtime version (created during build) in the container schema.

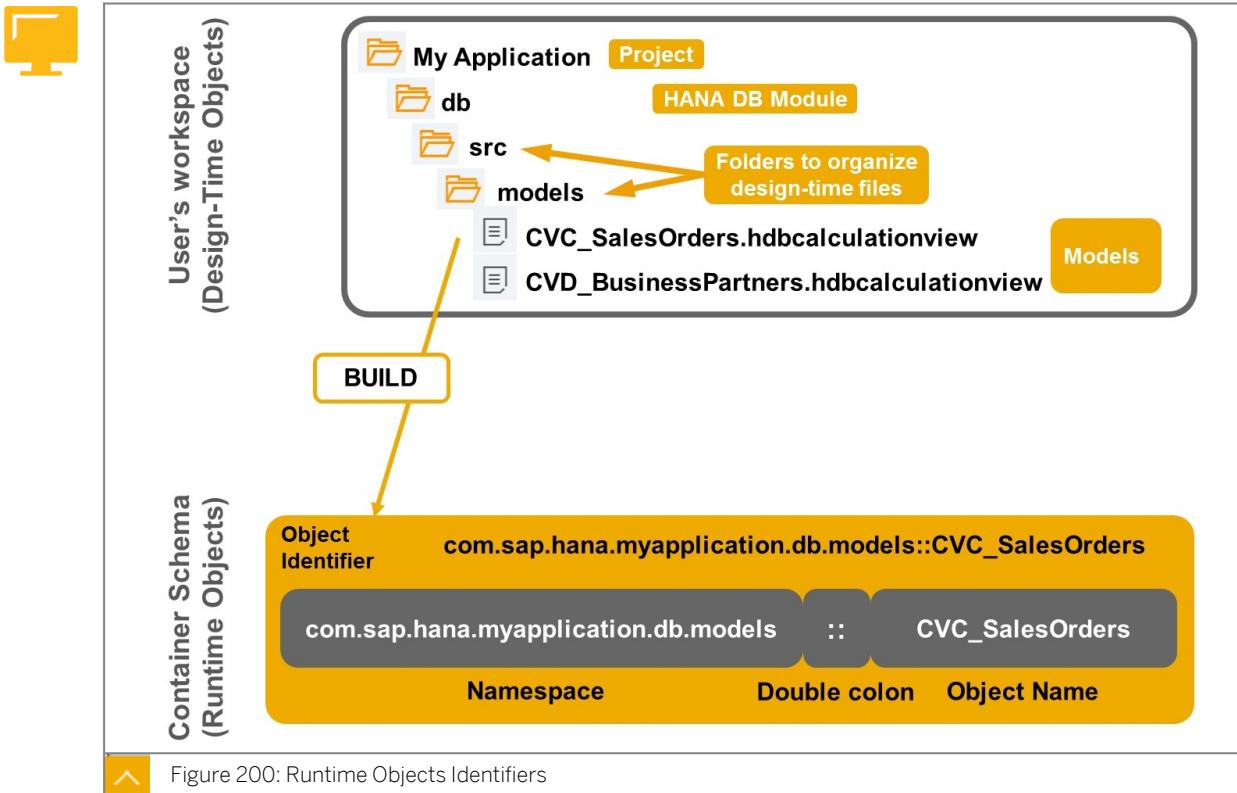


Figure 200: Runtime Objects Identifiers

The figure, Runtime Objects Identifiers, explains how folders are used to organize the design-time files in the HDB module of your application, and shows what the runtime object identifier could be. In this scenario, the calculation view identifier has a namespace. Let's now discuss the main rules and possible options to define the namespace.

Namespace

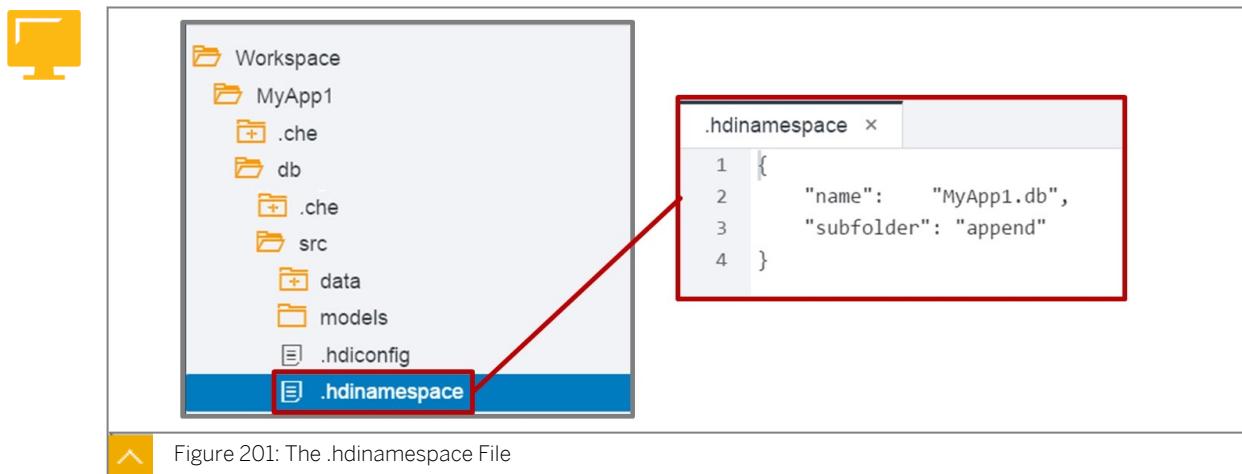
The following rules apply to the namespace:

- The namespace is optional. Some objects can be defined with a namespace in their identifier, and others without.
- A HDB module can have no namespace defined at all, or can specify any number of different namespaces.
- The namespace is always identical for all the design-time objects stored in the same folder.
- The namespace must always be explicitly mentioned inside the design-time files, and must correspond to the namespace defined explicitly in the containing folder, or implicitly cascaded from the parent folder(s).

The .hdinamespace File

The definition of a namespace is possible in the src folder of an HDB module, and also in any of its subfolders. This is done with a file called **.hdinamespace**.

When you create an HDB module, a **.hdinamespace** file is created automatically in the folder `<hdbmodulename> → src` and contains a default value `"<application_name>.<HDB_module_name>"`. You can modify this file to apply different rules than the default to your project's namespace(s).



Elements of .hdinamespace File

The content of any .hdinamespace file is always comprised of two elements, *name* and *subfolder*.

Element	Possible values	Default Value
"name"	<ul style="list-style-type: none"> "<Any syntactically correct namespace of your choice>" "" [no namespace specified] 	"<application_name>.<HDB_module_name>"
"subfolder"	<ul style="list-style-type: none"> "append" "ignore" 	• "append"

To structure your runtime modeling content in a way that suits your needs, you can add a .hdinamespace file to any sub-folder of the *src* folder. The "subfolder" setting determines whether the sub-folder names should be added "implicitly" to the namespace for objects located in sub-folders (append) or not (ignore).



Caution:

When you modify the specifications of the .hdinamespace file (or create a new .hdinamespace file in a folder), the new namespace rules are taken into consideration only after the .hdinamespace file has been built.

As a general recommendation, the namespace rules should be defined before you create modeling content. Indeed, if you modify these rules after creating modeling content, you have to adjust the object identifiers (which includes the namespace) before you can build your models. And this is even more complicated if dependencies exist between models.

Object Name

The way to define a runtime object name in design-time files is governed by one of the two following rules, depending on the object type.

Table 20: Defining the runtime object names

Type of modeling object	Number of objects per design-time files	Naming rule
Calculation views, table functions, procedures, analytic privileges	Each design-time file defines only one runtime object (*)	The runtime object name is defined inside the object content and should match the design-time file name (without file extension).
Synonyms, CDS tables, CDS views, roles	Each design-time file can define one or several runtime object(s)	The runtime object name is defined only in the object content and is not linked to the design-time file name.

(*) Here, we mean, only one “core” runtime object. As you might have observed, even a simple calculation view generates a core column view, plus other “child” column views for the attributes, hierarchies, and so on.



Note:

In this table, we are not covering the entire typology of models, but only the main ones that are discussed in this course.

Let's take two examples, one for each of the two approaches.

Object Name Example: Calculation View



**Design-time file
(Web IDE workspace)**

```
CVC_SO.hdbcalculationview <?
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Calculation:scenario ..... id="HA300:CVC_SO" applyP...
3  <Descriptions defaultDescription="Summary of Sales Orders" ...
4  <localVariables/>
5  <variableMappings/>
6  <DataSource id="HA300::SALES_DATA">
7  <resourceUri>HA300::SALES_DATA</resourceUri>
8  </DataSource>
9
10 <!-->
11 <!-->
12 <!-->
```

**Recommendation:
Filename and Runtime
Object name should match**

**Runtime object
(Database Explorer)**

- My HA300_03 Container
- Public Synonyms
- Column Views**
- DataStores
- Functions
- Indexes
- Procedures
- Sequences
- Synonyms

Search ColumnViews

HA300:CVC_SO

BUILD

Figure 202: Object Name Example: Calculation View

Technically, the design-time file name for a calculation view can be different from the runtime object name.

296

© Copyright. All rights reserved.

However, we recommend that you always keep these names in sync to make it easier to find a design-time object based on the runtime object name.

Object Name Example: Synonyms

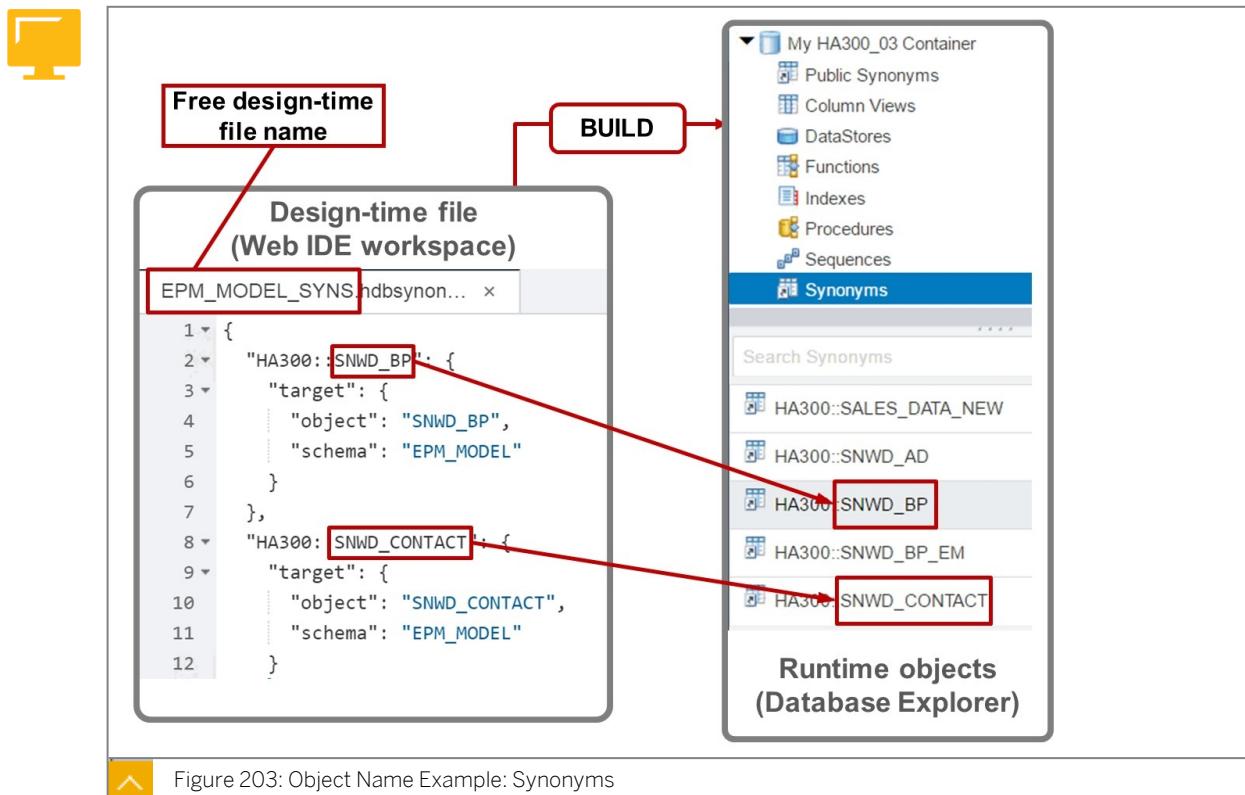


Figure 203: Object Name Example: Synonyms

This second configuration example shows that a design-time file for synonyms can define several synonyms. In that case, there is no specific rule or recommendation for the design-time file name.



Note:

In your *HA300_##* project, we have chosen an approach where each .hdbsynonym file contains synonyms for objects located in the same schema (for example, *EPM_MODEL* or *TRAINING*).

The SAP HANA Deployment Infrastructure

The SAP HANA Deployment Infrastructure (HDI) is a set of services specific to XS Advanced that allows the independent deployment of database objects. It relies on HDI containers.

HDI containers are a special type of database schema that manage their contained database objects.

These objects are described in the HDB modules of XSA projects, in design-time artifacts that are deployed by the HDI Deployer application. HDI takes care of dependency management and determines the order of activation; HDI also provides support for upgrading existing runtime artifacts when their corresponding design-time artifacts are modified.

One key concept of HDI containers is the fact that the entire lifecycle (creation, modification, and deletion) of database objects is performed exclusively by the HDI. Therefore, you cannot

create database artifacts, such as tables or views, with Data Definition Language (DDL) statements in SQL.

Indeed, if you create a table in a container schema with a plain SQL statement, this means there is no design-time object for this table. So, upon deployment of the container, this table will not be recreated. Similarly, deleting a database object from your container with SQL will not remove its design-time counterpart, so the object will be recreated upon build and/or deployment.



Note:

This is very different from other types of applications that use different persistence frameworks that rely on plain schemas. These frameworks can directly execute DDL statements.

Data Sources for Calculation Views

When you design calculation views, the data sources must exist as design-time files in the container of your HDB module. So they must be of one of the following types:

- "Real" database objects (table, view, table function, virtual tables, and so on) existing inside the container and having a corresponding design-time file in the HDB module
- Synonyms referencing database objects located in any other schema than the container itself. You will learn about synonyms later on.



Caution:

If a Calculation View uses as a data source table that was created by plain SQL in your container (so, without a design-time file) cannot be built. The build fails and issues an error message, such as:

The file requires "db://HA300::TABLE_SQL" which is not provided by any file

Key Properties of HDI Containers



- A container is equal to a database schema.
- Database objects are deployed into the schema.
- Definitions must be written in a schema-free way.
The name of the container schema is determined only when deploying the container.
- Direct references to external schema objects are not allowed.
These objects must be referenced using database synonyms created within the container.



Note:

You will learn more about synonyms later on in this unit.

HDI Container Configuration File

Within each HDB module of a project you will find a very important and mandatory file that exists under the `src` folder with the suffix: `.hdiconfig`. This is the HDI container configuration file that is used to bind the design-time files of your project to the corresponding installed build plug-in.

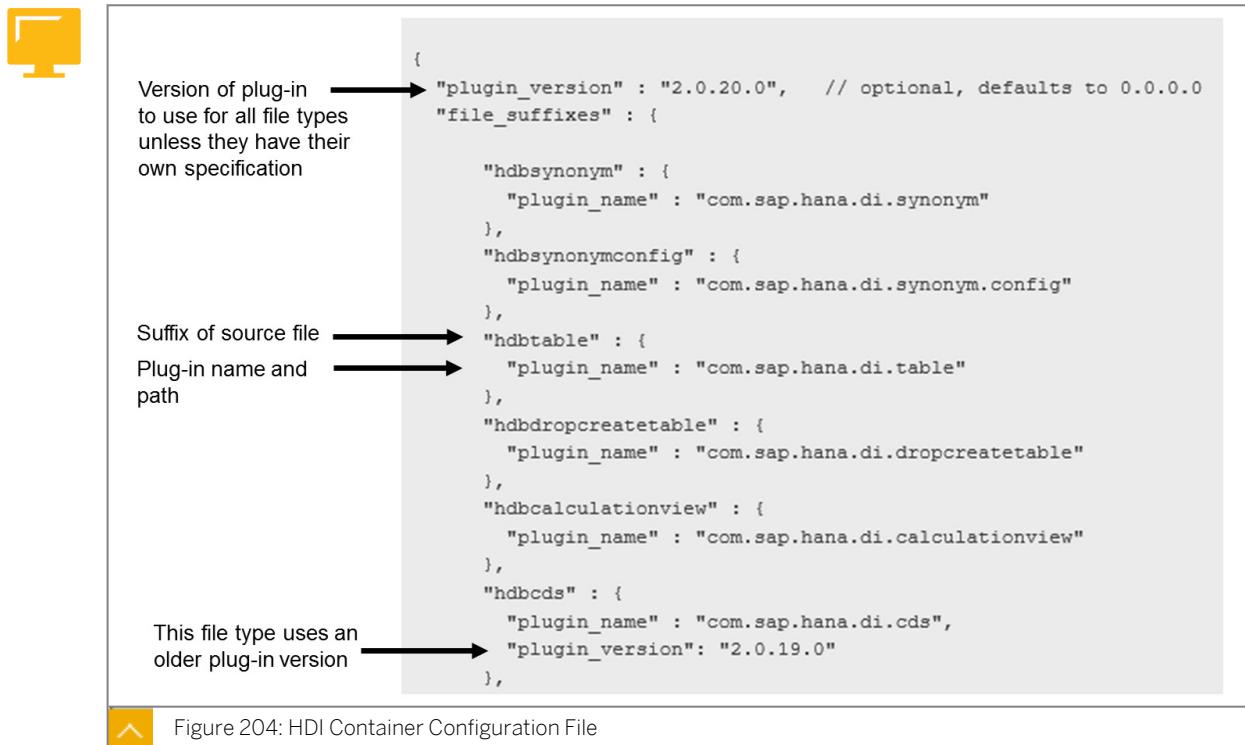


Figure 204: HDI Container Configuration File

Without the build plug-in it is not possible to build the run-time objects from the design-time files. This configuration file directs each source file in your project to the correct build plug-in.

Note:

The `.hdiconfig` file is hidden by default in the Web IDE, so you may need to first expose the file (press the button that looks like an 'eye' above the source files tree to show the hidden files).

Usually there is only one `.hdiconfig` file in a HDB module, and this must be located in the root folder of the HDB module. This file contains a list of all bindings for all source file types. But it is also possible to create additional `.hdiconfig` files lower down in the project folder hierarchy which would then expand or restrict the definitions at a certain point in the hierarchy. Because the configuration file(s) are specific to each project, this means you could work on multiple projects in the Web IDE, each using different versions of the same build plug-ins. This could be useful when working on the next phases of development of an application that require the latest build plug-ins but at the same time needing to support older versions of the application which require the older build plug-ins.

Inside the container configuration file you will find, for each source file type, three entries:

- suffix name, for example, "hdbcalculationview"
- plug-in name, for example, "com.sap.hana.di.calculationview"

- plug-in version, for example, "2.0.50.0"

The version number definition is optional, but it is a good idea to include this so that you are sure to be binding the correct version of the plug-in to the source files in your project. You can have multiple versions of the same build plug-in installed in your landscape and the plug-ins are often updated with each HANA release to handle the new features. For plug-ins that share the same version number you can specify the version number once at the start of the file and not for each suffix. Then, only specify the version number against the suffixes that do not follow the global version number defined at the top of the file.

It's very important to remember than when you import a project into your landscape, it brings with it its own *.hdiconfig* file that refers to the plug-ins versions that were used when it was first developed. If you then plan to update the source file using newer features of SAP HANA (for example, you want to add a new feature to a calculation view that just became available with the newer version of SAP HANA) you will not see the new feature in the source editor if the *.hdiconfig* file has not been adjusted to use the later version of the build plug-ins. In other words, just because you install the new build plug-ins does not mean that it is automatically used for all projects. Always check the *.hdiconfig* file each time to be sure. For new projects where the *.hdiconfig* file is first created, the latest plug-in version will be referenced in the configuration file.



Caution:

The HDI container configuration file is an unnamed file and must only consist of the suffix *.hdiconfig*. If you try to add a name, the build of the *.hdiconfig* file will fail. Leave it as it is.

The Database Explorer

Inside the SAP Web IDE for SAP HANA, you can use the *Database Explorer* perspective to view the database artifacts (tables, views, procedures, column views) of one or several containers that you add to the list.

When you do this, even if you are logged on to the SAP Web IDE with your usual SAP Web IDE user, access to the container is done by a technical SBSS (Service Broker Security Support) user that is created transparently when you add the container to the Database Explorer. This technical user interacts with the database objects on your behalf, for example, to view the content of a table or preview the data of a calculation view.



Note:

Until SAP HANA 2.0 SPS02, the HDI container technical users' names had an *SBSS_* prefix.

As of 2.0 SPS03, to enhance readability, the technical user's name starts with the corresponding schema name (container). For example
HA300_01_HDI_HDB_1_5TODRJKQCIJYG7T2H9076YFG6_RT

If your container consumes data from an external schema, the technical user must also be granted authorizations to the external schema objects, for example, to view the data of an external table referenced by a synonym. You will learn more about this in the unit, *Security in SAP HANA Modeling*.



Note:

The Database Explorer also allows you to connect to classic database schemas, not managed by the HANA Deployment Infrastructure, to show the database objects and the content of tables, execute queries in a SQL console, and so on.

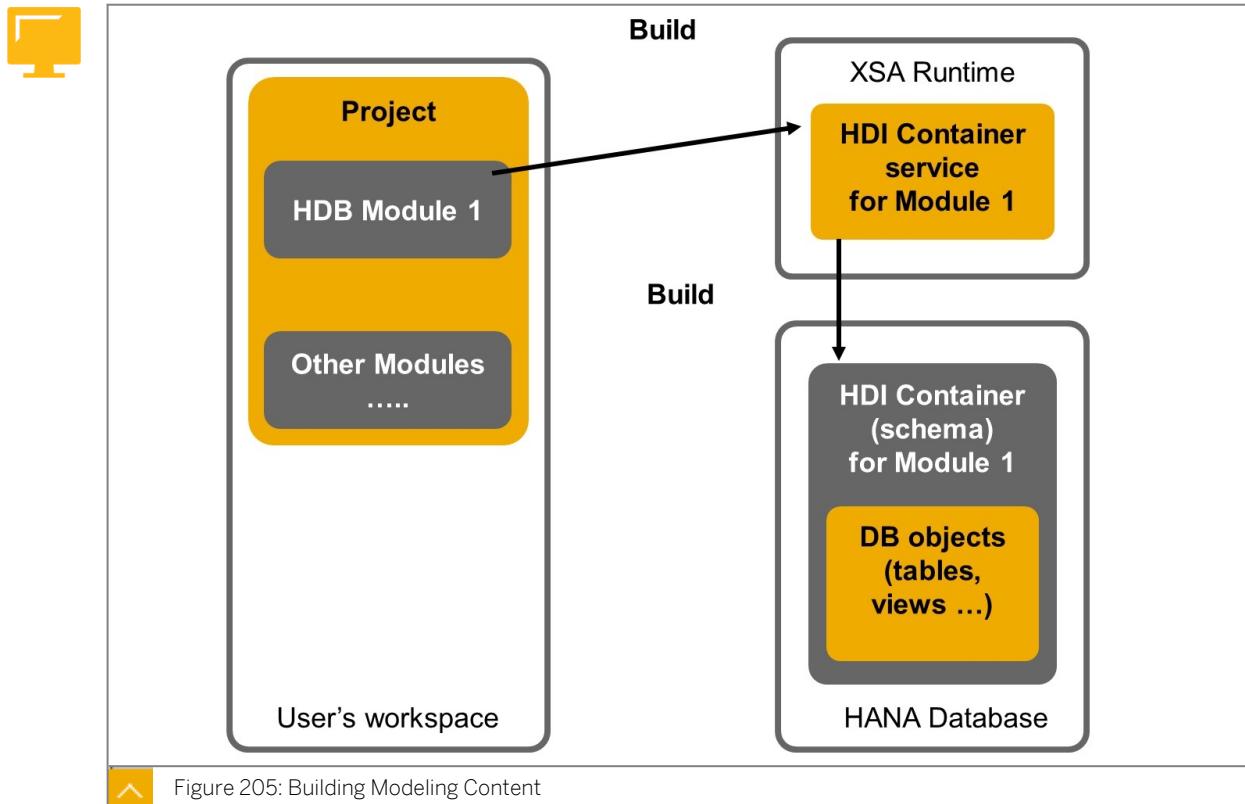
Building Modeling Content

Building modeling content means creating the runtime objects from the design-time objects, and deploy them to the container (schema).

A container service is also created in XS Advanced to manage the interaction between the XSA environment (in particular the SAP Web IDE, the HANA Deployment Infrastructure, and so on) and the SAP HANA database.

The first build of an HDB module (or some of its design-time content, such as CDS entities defining the persistence) also creates the corresponding container schema.

Structure of Building Modeling Content



A build operation can be executed at the following levels of a project structure:

- An entire HDB module
- A sub-folder of the HDB module
- One or several individual objects

**Caution:**

Building the entire XSA Project, that is, triggering the build at the project level, does NOT build the HDB module(s) contained inside the project. It does not affect the database objects within your container. Building an XSA project is mainly used to prepare the transport and deployment of the multi-target application.

Regardless of what you build (HDB module, sub-folder, or single object), there is always an end-to-end dependency check applied to each object that is part of the build scope. So, even if you build a single object, there will be a recursive check of all the objects it references to make sure these objects already exist as runtime objects or are defined in a design-time object and built at the same time.

Build Errors

When you build modeling content defined in a HDB module, you might come across different issues which are described in the build log that you can consult in the console. Some are related to the dependency checks we have just discussed, others have to do with the consistency of design-time objects (for example, the namespace settings of design-time folders). Another classical build error is when two different design-time files provide a definition for the same runtime object.

Let's try to list a number of frequent root causes for build errors.

Most Common Errors during Build Operations



- The project has just been imported but is not yet assigned to a space.

Indeed, a build always triggers the deployment to a container. A condition is that the space in which to deploy the container service must be specified. Your SAP Web IDE user must have the *Developer* role in the space.

- The definition of an object on which another object depends is not provided, or it is provided but this object has not been built yet.
- The definition of a runtime object is provided by several design-time files.
- There is a namespace inconsistency between the content of a design-time file and the namespace property of the folder in which it is located.
- The service to access external schemas is not defined and running in the target space.
- The synonyms for external schema access are not built yet.
- There is an object-specific design error.

For example, no measure is defined in a Cube calculation view.

- There is an inconsistency between the object you build and a referenced object.

For example, there is a mismatch in a column name between two objects with a dependency.

Putting a Design File Aside to Workaround Build Issues

When a Calculation View cannot be built for whatever reason, it generally prevents the build of its containing folder and any upper folder. There are several approaches you can use to solve

this issue temporarily and go on working on the rest of your models, without losing completely the existing design-time file.

- Export the Calculation View to a safe place, and then delete the Calculation View from your workspace.

When executing the build, the removed Calculation View won't cause any issue. When you want to resume working on this view, you just need to import it back to its original location.

- Add `.txt` to the design-time file. For example, `SALES.hdbculationview` will become `SALES.hdbculationview.txt`

In this scenario, you do not remove the file from the workspace, but modify its extension so that it is (temporarily) not considered as a Calculation View design-time file but a plain text file. No specific check is performed on a `.txt` file upon build, so it will have the same effect as removing the file, but it is then much easier to undo: you just need to remove the additional `.txt` extension.



Note:

You could also replace the extension; but on the one hand, adding an extension works fine –it is recommended for temporary use only– and on the other hand, it is easier, faster, and less error prone as you keep trace of the original extension.

Note that, with both approaches, dependencies between objects is what can makes things a bit more complicated. For example, if the Calculation View you stash away was consumed by other Calculation Views, you might also have to rename the consuming Calculation Views in a similar way.

Runtime Calculation Views Materialization

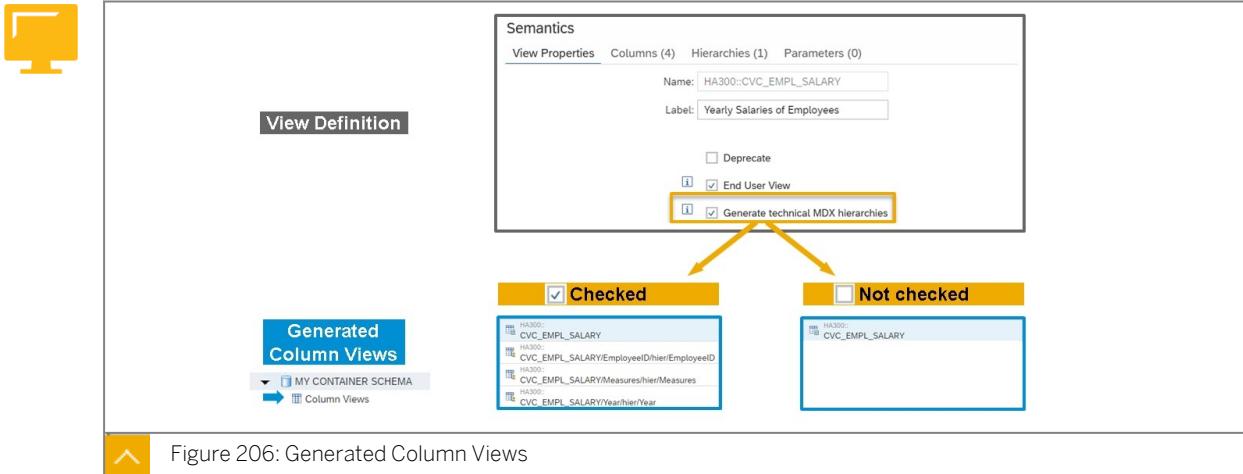
When you build a Calculation View, as discussed earlier, the **corresponding runtime object** is materialized by one or several **Columns Views** within the HDI Container schema. In the Web IDE for SAP HANA, and more specifically in the Database Explorer, you can connect to the HDI Container and display the list of existing columns views.



Note:

You can also navigate and execute the HDI-based Calculation View runtime objects from the classic Database Access (that is, connected to the entire database instead of a specific HDI Container schema), but then a prerequisite is that your user has privileges on the HDI Container and/or some of its objects.

In the *Column Views* section of the catalog, the runtime objects corresponding to the Calculation View are listed. At minimum –if the build is successful– you find one object, which is the materialization of the calculation view. For example, the `CVC_EMPL_SALARY` Column View.



In addition, if you have checked the option *Generate technical MDX hierarchies*, an additional series of Column Views (technical hierarchies) are generated. These Column Views support MDX reporting on top of the Calculation View. So depending on how the end-users will consume the view, you decide whether to activate this option. When the option is deselected, this allows a faster build/deployment, because the hierarchy views for single columns (attributes or measures) are not generated.



Note:

This is a new option in SAP HANA 2.0 SPS06. In earlier versions, the technical MDX hierarchies were systematically generated.



Caution:

This concept of *Technical MDX hierarchy* exists regardless of whether or not your Calculation View includes modeled hierarchies, that is, Level or Parent-Child Hierarchies that you define in the *Hierarchies* tab of the *Semantics* node. That is, even a basic calculation view with two attribute columns and a single measure, but no modeled hierarchy, will still generate, upon build, a main Column view plus three technical MDX hierarchy views if the option is selected.

The SAP HANA Analytic Catalog

Together with runtime objects generated during the build of Calculation Views, the HDI Builder also generates a series of tables and (mostly) views in the *_SYS_BI* schema, to implement an **Analytic Catalog** that front-end tools can use to expose the views and provide information about their content. All these objects have their names prefixed with **BIMC_**.

The BIMC tables and views have their own reference documentation on the SAP Help Portal for SAP HANA Platform, in the *Use* section. You can also search for **BIMC Views**.

The End User View Property

The new *End User View* property has been introduced in SAP HANA 2.0 SPS06 in order to facilitate the maintenance of a list of views to expose to the end-user. You set this property in the *Semantics* node, in the *View Properties* → *General* tab. It is especially useful in scenarios where a number of views are reused by other views but not necessarily meant to be exposed to the end user.

Note that this flag does not replace a relevant assignment of privileges (especially the *SELECT* privilege on the Column Views generated by the build operation), but it allows the modeler to list the views that are relevant to the end-user from within the Calculation View Editor.

Technically, this property is included in the SAP HANA Analytic Catalog (in the *BIMC_ALL_CUBES* and *BIMC_ALL_AUTHORIZED_CUBES* views), so some reporting tools can make use of it when browsing the *BIMC** tables. It has no "other" technical impact: the view will be built and instantiated upon *SELECT* in the same way, regardless of whether the flag is selected or not.



Note:

The *End User View* property is selected by default when you create a new Calculation Views, and also –to avoid additional effort when upgrading SAP HANA– for any existing Calculation View developed in SAP HANA up to version 2.0 SPS05, that is, when the property did not exist.

Import and Export of Modeling Content

The SAP Web IDE for SAP HANA offers import and export features within the workspace.

These features can be used to keep several versions of your modeling content, and re-import all or part of this content if needed.

They are also useful for quick sandboxing. Indeed, even in your own workspace, you cannot copy/paste an entire project. Instead, you export this project and import it. You will only need to change the imported project name in the *Import to* field of the *Import* dialog box.

Additionally, import/export can be used to share in a simple way your modeling content with another developer. For example, when a series of synonyms to external database objects have been defined in your project, you can easily share the synonyms definitions with another developer by sending the relevant design-time files for synonyms.

Correspondingly, you can import an individual file or an archive. In that case, most of the time, the archive needs to be unzipped.

Exporting Modeling Content

An export of modeling content can be executed at different levels of a project structure, and applied to either a single file or a folder.

- A single design-time file is exported as is, with the source filename and extension.
You cannot export a multi-selection of individual files.
- A folder within the project structure (up to the project folder itself) is always exported as a .zip archive.
The *.rar* standard is not supported for project content export/import. This standard is used in *.mtar* (MTA archive) files, but these files are used for application deployment, which is different from design-time content export/import.

Importing Modeling Content

Corresponding to the export, you can either import an individual file or a .zip archive. In the latter case, most of the time, the archive needs to be unzipped.

Two specific cases always require an intermediate step, after the import, before you can build the imported objects:

- When you import an entire XSA project, you must assign the project to a space before building the project or any of its components (in particular the HDB module).

In the workspace, right-click the project and choose *Projects Settings*. In the *Space* section, assign a space and choose *Save*.



Note:

Only the spaces to which you are assigned as a developer are listed.

- If you import a HDB module into a new project (for example, after exporting this HDB module from another project), you must first identify the imported content as a HDB module in the project.

In the workspace, right-click the new (imported) module —it is not yet known as an HDB module even if its name is something like *HDB*, *dbmodule* or *db*— and choose *Convert to → HDB Module*.

The module entry is added to the MTA descriptor file *mta.yaml* and, from this point on, is considered as a HDB module.

Copy, Rename, Move, and Delete Modeling Content

The structure of a project in the SAP Web IDE for SAP HANA looks like a classical file structure, but the changes you make to this structure by copying, renaming, moving, or deleting files (or folders) can have strong impact on the consistency of your project. This impact does not necessarily show up immediately during the file structure modification. It generally appears when you build the modeling content.

Whenever you copy, rename, move, or delete modeling content, you must keep in mind the key rules that govern the build of database artifacts:

Main Rules for a Consistent Management of Modeling Content Files



- In an entire HDB module, the definition of a given runtime object (`<namespace>::<object_name>`) cannot be provided more than once.
- The namespace defined in the design-time file of a database object must correspond to the namespace setting applied to the folder in which it is located
- A build operation always checks the end-to-end dependency between modeling content across all the HDB module, but only builds the design-time files you have selected for the build operation.
- During a build operation, the checks apply to all the runtime objects that are already built, and also all the objects included in the build scope (that is, in case of a partial build, the design-time files you have selected).

A design-time file that has never been built and is not part of the build operation is ignored.

Copying Modeling Content

When you copy/paste a design-time file within the same folder, you must provide a different name for the copy. This is not the case if you copy/paste a file to a different folder where there is not a file with the same name.

However, in any case, the content of the design-time file is not modified at all. Which means you have in your HDB module two design-time files with identical content.

At least, you must ensure that you rename all the runtime objects defined in the design-time file to make these objects unique in the whole HDB module. And, if needed, you must also change the namespace according to the target folder namespace settings.



Note:

If you copy a design-time file that was already opened in a code or graphical editor, keep in mind that copy/pasting does not open the new file (the copy). The file visible in the editor is still the original one, when - in many circumstances - you might want to actually modify the copy.

Moving Modeling Content

When moving modeling content, one key rule that you must think about is the namespace setting of the target folder.

- If the namespace settings for source and target folder are the same, moving the file has no particular impact on objects that reference the moved object.

However, a build of the moved object can be successful only if the build scope includes the source folder, in order to execute/acknowledge the “deletion” of the moved file in its source folder. If not, the build will fail.

- If the namespace settings for source and target folder are different, you must at least adapt the namespace. In this case, you must first perform an impact analysis to check whether some existing objects reference the one you are about to move.

Renaming Modeling Content

For the HDI builder, renaming a design-time file is like deleting a design-time file and creating a new one with the same content. You must be aware of the following rules and recommendations:

- A design-time file that creates only one “core” runtime object (for example, a calculation view or a table function) should always have the same name as the runtime object it defines. For example, after renaming the design-time of a calculation view, it is recommended to also change its ID in the code by opening the ID of the runtime object.

This is not a technical requirement, but rather a best practice to keep your design-time content readable.

- You must check if database objects reference the object you plan to rename. Generally speaking, renaming objects that have dependencies requires modification to the references to this object in other objects. Ideally, you should do this only on exception, because it is error-prone, or limit this practice to test objects, typically when you copy – and rename – an existing object in order to test changes you make to this object while keeping the source object.
- You must be particularly careful with partial builds. If you rename an object that was already built and had dependencies, the renamed file is seen by the HDI Builder as a new design-time. The original design-time file will be seen as deleted by the HDI Builder only when you build either the entire HDB module, or any of its sub-folders that contains the renamed file.

SAP HANA 2.0 SPS02 has introduced a new feature, *Refactor*. This allows the following operations:

The Refactor Functionality

- Rename a runtime object (for example, a calculation view).
- Check which other calculation views reference the renamed object, and adjust the reference accordingly.

This does not work in artefacts where the references are located in SQL or SQL Script code (for example in a procedure).

There are two ways to invoke this functionality:

- From the workspace

Right-click the object you want to rename and choose *Rename* or press **F2**. After modifying the design-time file name, a dialog box appears to ask you whether you want to also rename the view (change the runtime object identifier) and adjust the references, that is, make sure that other views ("impacted views") referencing the renamed one will still point to the corresponding (renamed) runtime object.

- From the calculation view editor

In the *View Properties* tab of the *Semantics* node, click the *Edit* icon next to the runtime object *Name* and enter the new runtime object identifier. Similarly, you get a dialog box offering to adjust the reference in impacted objects.

Deleting Modeling Content

Deleting modeling content can have surprising effects, especially in case of partial builds. Indeed, the deleted design-time file is only seen as actually deleted by the HDI Builder when you build the entire HDB module, or any of the sub-folders that contained the deleted file.

In other words, the smallest build scope you can think of, in order to validate the deletion of a design-time file, is the folder in which the deleted file was located. So, even if it is empty, this folder is extremely precious.

Indeed, if you delete this folder as well, you can only validate the deletion of the design-time files it contained by building a parent folder. Suppose this parent folder is the *src* folder and it contains one (or several) models that you know cannot be built at the moment (for whatever reason).

Therefore, when deleting modeling content, you should apply the following best practices:

- Before deleting a design-time file, always perform an impact analysis.
- Before deleting a folder, always perform a partial build at this folder level. If the build is successful, it means that all the runtime objects that were defined in design-time files from this folder have actually been (successfully) deleted.



Caution:

As already mentioned, you should NEVER delete a database object located in an HDI container with a plain SQL statement.

Using the Search/Replace Feature

The SAP Web IDE for SAP HANA offers several find/search and replace features that are useful to manage the renaming of objects and the code adjustments that must be performed manually.

Within the Code Editor, you can use Find (**Ctrl+F**) or Replace (**Ctrl+H**) to locate easily (and replace if needed) the ID of an object.

On the right toolbar, you can also use the Advanced Repository Search to look for a particular text string, either in the name of design-time files (within a folder, a project, or the entire workspace) or within the content of the design-time files. You can directly open a design-time file from the search results.

Rename Columns

In the SAP Web IDE for SAP HANA, it is possible to rename one or several columns of a Calculation View from the Semantics node.

Renaming a Column in an Intermediate Node

It is possible to rename a column in any node of the calculation scenario.



Note:

The case of the top node is specific and will be dealt with later on.

Inside a calculation view node, you can rename the output columns only, not the source columns. As a consequence, a column must first be added to the output in the *Mapping* tab before you can rename it. Then you have two main options to rename the column:

- From the *Mappings* tab

Select the column in the *Output Columns* area. Then, in the *PROPERTIES* area, modify the *Name* field.

- From the *Columns* tab

The *Name* column allows you to change the name of several columns at once.

Note that the impact of renaming a column on the upper nodes of the calculation view depends on the upper mapping status.

- If the column was not yet mapped in upper nodes, the new column name will be used when adding it to the output of upper nodes, up to the top node.
- If the column was already mapped in upper nodes, renaming the column applies only locally: at the next level, the column is mapped "back" to a column with the original name.

Renaming a Column at the Top Node Level

This case is different from the one above, because the column names in the output of the top node (similar to the ones shown in the Semantics) are used to consume the calculation view by front-end tool, SQL queries... but also by other calculation views.

Technically, columns of the top node can be renamed in the same way as discussed before: from the *Mapping* or *Columns* tabs. In addition, the *Semantics* allow you to maintain a number of column properties, including the *Name* and *Label*.



Caution:

Calculation Views exposed externally must be handled with care in order to guarantee they can be used without disruption. This includes, among others, column names.

For a Calculation Views that is consumed by another Calculation View, a change in column name is likely to prevent the build of the consuming calculation view.

Rename and Adjust References

To enable column renaming in stacked scenarios, the *Semantics* includes a feature that allows you to rename one or several columns and to adjust them in other Calculation Views where these columns are consumed.

You can perform a search on column names (with operators contains and equals, or regular expressions), and adjust the column names manually. A dedicated button allows you to copy the current name to the new name for one or several columns, as a first step, to make things easier. You can also use a find and replace approach, including regular expressions if needed.

After renaming, the dependencies are analyzed and a report shows the impacted views. These views can then be refactored so that they reference the new column names. Building the view with renamed columns will fail until you have adjusted all the consuming views.



Note:

Some objects, such as table functions or procedures, might not be supported by the refactoring mechanism. In this case, they must be adapted manually to match the new column names in the underlying calculation views.

Translate Descriptions

For a calculation view, you can generate a properties file that contains the names and descriptions of all calculation view objects such as columns, input parameters and variables.



Table: _SYS_BI.BIMC_DESCRIPTIONS

SCHEMA_NAME	QUALIFIED_NAME	ID	LANG
1 HA300_05_HDI_HDB_1	HA300::CVC_CAMPAIGN_ANALYSIS_00	&&VIEW_NAME&&	Email Campaign Analysis
2 HA300_05_HDI_HDB_1	HA300::CVC_CAMPAIGN_ANALYSIS_00	CUSTOMER	CUSTOMER
3 HA300_05_HDI_HDB_1	HA300::CVC_CAMPAIGN_ANALYSIS_00	CURRENCY	CURRENCY
4 HA300_05_HDI_HDB_1	HA300::CVC_CAMPAIGN_ANALYSIS_00	SALES_AMOUNT	SALES_AMOUNT
5 HA300_05_HDI_HDB_1	HA300::CVC_CAMPAIGN_ANALYSIS_00	SALE_DATE	SALE_DATE
6 HA300_05_HDI_HDB_1	HA300::CVC_CAMPAIGN_ANALYSIS_00	DAYSELAPSED	Days between Sale Date and e-mail
7 HA300_05_HDI_HDB_1	HA300::CVC_CAMPAIGN_ANALYSIS_00	INPEMAILDATE	The date when the e-mail was sent

Semantics

- Name: HA300::CVC_CAMPAIGN_ANALYSIS_00
- Label: Email Campaign Analysis

Properties File Content:

```

{
  "Semantics": {
    "Columns": [
      {"ID": "CUSTOMER", "Label": "Customer", "Type": "Text", "Lang": "EN-US", "Value": "CUSTOMER", "LangList": ["EN-US"]},
      {"ID": "CURRENCY", "Label": "Currency", "Type": "Text", "Lang": "EN-US", "Value": "CURRENCY", "LangList": ["EN-US"]},
      {"ID": "SALES_AMOUNT", "Label": "Sales Amount", "Type": "Text", "Lang": "EN-US", "Value": "SALES_AMOUNT", "LangList": ["EN-US"]},
      {"ID": "SALE_DATE", "Label": "Sale Date", "Type": "Text", "Lang": "EN-US", "Value": "SALE_DATE", "LangList": ["EN-US"]},
      {"ID": "DAYSELAPSED", "Label": "Days Elapsed", "Type": "Text", "Lang": "EN-US", "Value": "DAYSELAPSED", "LangList": ["EN-US"]},
      {"ID": "INPEMAILDATE", "Label": "Inp Email Date", "Type": "Text", "Lang": "EN-US", "Value": "INPEMAILDATE", "LangList": ["EN-US"]}
    ],
    "Parameters": [
      {"ID": "INPEMAILDATE", "Label": "Inp Email Date", "Type": "Text", "Lang": "EN-US", "Value": "INPEMAILDATE", "LangList": ["EN-US"]}
    ],
    "Aggregation": [
      {"ID": "DAYSELAPSED", "Label": "Days Elapsed", "Type": "Text", "Lang": "EN-US", "Value": "DAYSELAPSED", "LangList": ["EN-US"]}
    ]
  }
}

```

Figure 207: Properties File

You first generate the properties file for the calculation view using the menu option *Modeling* → *Generate Properties File*. You then must build the SAP HANA Database Module that contains the generated properties file so that the names and description values are then added to the *_SYS_BI.BIMC_DESCRIPTION* table.

The key reason for doing this is to enable translation of the name and description values in a calculation view of all objects to multiple languages by updating the `_SYS_BI.BIMC_DESCRIPTION` table.

Client tools can read the `BIMC_DESCRIPTION` table and look-up the descriptions in the user's local language.

If you adjust the names or labels in a calculation view, you must regenerate and build the properties file so that the `_SYS_BI.BIMC_DESCRIPTION` table is updated.

The properties file contains only descriptions and names of all objects defined in a calculation view. It is not meant to be used as documentation of a calculation view. Consider using the menu option *Modeling → Generate Document* if you need to generate a file that includes all settings of a calculation view.



Note:

SAP HANA does not provide tools to translate the descriptions.



LESSON SUMMARY

You should now be able to:

- Analyze and document information models
- Explain the structure of a project
- Build modeling content
- Modify and move modeling content
- Translate Descriptions

Creating and Managing Projects

LESSON OVERVIEW

In this lesson, you will learn about how to create a brand new project and how to define the key settings of a project. You will also get the main information on how to consume data from external data sources.

Like in the entire course, we will only focus on modeling within an HDB Module, so we will not cover any other type of module (Java, Node.js, HTML5, UI5, and so on).

Business Example

You are working as a Modeler at a client site and need to create a project that will contain the modeling content. You need to understand the key options that you need to define to set up the project.

Your project will mainly consist in Calculation Views modeling based on data located in an external schema. So you want to understand the key steps needed to enable access to this external data.

Finally, you also want to understand the key difference between developing/building an application in a space, and deploying a ready-made application to a specific target space for testing or production.



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Define the key settings of a project
- Manage the lifecycle of a project

MTA Project

A Multi-Target Application (MTA) project is comprised of one or several modules of different types. You learned about module types in a previous lesson. Here, we focus on the SAP HANA Database (HDB) module type.

However, it is important to understand some key settings of an MTA project. Some of them do not have to do only with HDB modules, but are related to the lifecycle of a project in general.

When you create a new project, you have to define a few parameters. Let's review these parameters.



Table 21: Key Settings of an MTA Project

Setting	Description	Example
Project name	The name given to the new project in your workspace	

Setting	Description	Example
Application ID	The identifier of the multi-target application, used during deployment. It must be unique in the target environment, and uses the reverse-URL dot-notation.	com.sap.mta.sample
Application version	A three-level application version number (Major.Minor.Patch), used to handle successive deliveries of the same application to runtime environments.	1.0.3
Description	A free text describing the purpose of the MTA.	
Space	The space where you want to build your project content during the development phase.	DEV

All these settings are stored in the project descriptor file, which is named *mta.yaml*. This file can be edited with the code editor, or with a new MTA editor available from SAP HANA 2.0 SPS01 onwards.

HANA Database Module

As you have already seen, a Multi-Target Application can include one or several modules of the type HDB (SAP HANA Database). In the context of modeling, a HDB module can define a local persistence (a persistence layer that is defined and deployed in the application itself). When you build a virtual data model based on calculation views - even if the source data is stored outside of the application - the HDB module contains the database artifacts that define this virtual data model.

When you create a new HDB module, you must provide a number of properties for the module. Most of these properties are stored in the descriptor file of the application: (*mta.yaml*), except for a few. In particular, the namespace property of the module is not stored in the descriptor file but is materialized by the *.hdinamespace* file located in the *src* folder.



Table 22: Key Settings of an HDB Module

Setting	Description	Example
Name	The name of the module. It should be unique within the entire descriptor file (<i>mta.yaml</i>).	core-database-module
Type	hdb is the reserved module type for an HDB module.	hdb
Path	The relative location of the module root within the file structure of the project.	db

Setting	Description	Example
Schema Name	A specific schema name for the HDB module container, used during build and deployment.	SAMPLE_SCHEMA
SAP HANA Database Version	The SAP HANA database version against which the artefacts of the HDB module must be validated.	_schema-version: "2.0"

Default Schema Name of the HDB Module Container

As you might have noticed in the *HA300_##* project, specifying a schema name is optional.

From SAP HANA 2.0 SPS01 onwards, the default schema name for the container generated during the build of a project is the following string, after capitalization and replacement of some special characters (for example, any hyphen “-” is replaced with an underscore “_”):

```
<project-name>_<hdi-container-service-name>_<numeric_increment>
```



Note:

The default schema name generated during a deployment with the XSA Command-Line Interface (CLI), that is, outside of the SAP Web IDE, is different: *<project-name>* is replaced with *<application-ID>* (because a project name makes sense only in the context of the SAP Web IDE). It is also possible to specify the schema name at deployment time with an mta extension descriptor (a *.mtaext* file).

Specifying a Schema Name for the HDB Module Container

You can specify a schema name for your HDB module. This is done in the descriptor file *mta.yaml* of your project, and more specifically in the parameters of the hdi-container service that is defined as a resource used by the HDB module.

The following sample shows the additional parameters section of the *mta.yaml* file where the schema name is defined:

```
resources:
- name: hdi-cont
parameters:
  config:
    schema: <YOUR_SCHEMA_NAME>
properties:
  hdi-container-name: ${service-name}
type: com.sap.xs.hdi-container
```

Then, the schema generated during the first build of the HDB module is named as follows:

```
<YOUR_SCHEMA_NAME>_<numeric_increment>
```

**Caution:**

It is possible to force the schema name to be exactly what you specify in the `mta.yaml` file, by adding as follows:

```
config:
  schema: <YOUR_SCHEMA_NAME>
  makeUniqueName : false
```

However, this can generate schema naming conflicts in case several projects use the same schema name.

- 1
- 2
- 3

To Find the Actual Container Service Name and Schema Name of an HDB Module

You have built a HDB module from the SAP Web IDE for SAP HANA, and want to know the actual name of the HDI container service, and the name of the corresponding database schema.

1. Add the container to the Database Explorer.
 - a) Choose *Tools* → *Database Explorer*.
 - b) Choose the + (*Add a Database to the Database Explorer*) icon.
 - c) In the *Add Database* dialog box, select the database type *HDI Container* and enter the project name (or a part of it) in the search field.
The corresponding container(s) displays.
 - d) Select the relevant container.
The entry in the *Name* column corresponds to the actual XS Advanced service that manages the container.
 - e) In the *Name to Show in Display* field, enter a meaningful display name for the container.
 - f) Choose *OK*.
2. Open an SQL console connected to the container.
 - a) If needed, select the container in the top-left list (it should be already selected after the previous step).
 - b) Choose *Open SQL Console*.
3. Execute the following SQL query:


```
select CURRENT_SCHEMA from DUMMY
```

 - a) Enter the SQL statement.
 - b) Choose *Run* (or press **F8**).
The schema name of the container is displayed in the SQL query results.

Lifecycle of a Modeling Project in XS Advanced

You have already learnt how to manage some important stages of a modeling project's lifecycle.

- Create a project and define some key settings.
- Import and export modeling content, including an entire project.

- Build selected objects or an entire HDB module.

You also got an overview of how to collaborate on projects with repositories such as GitHub.

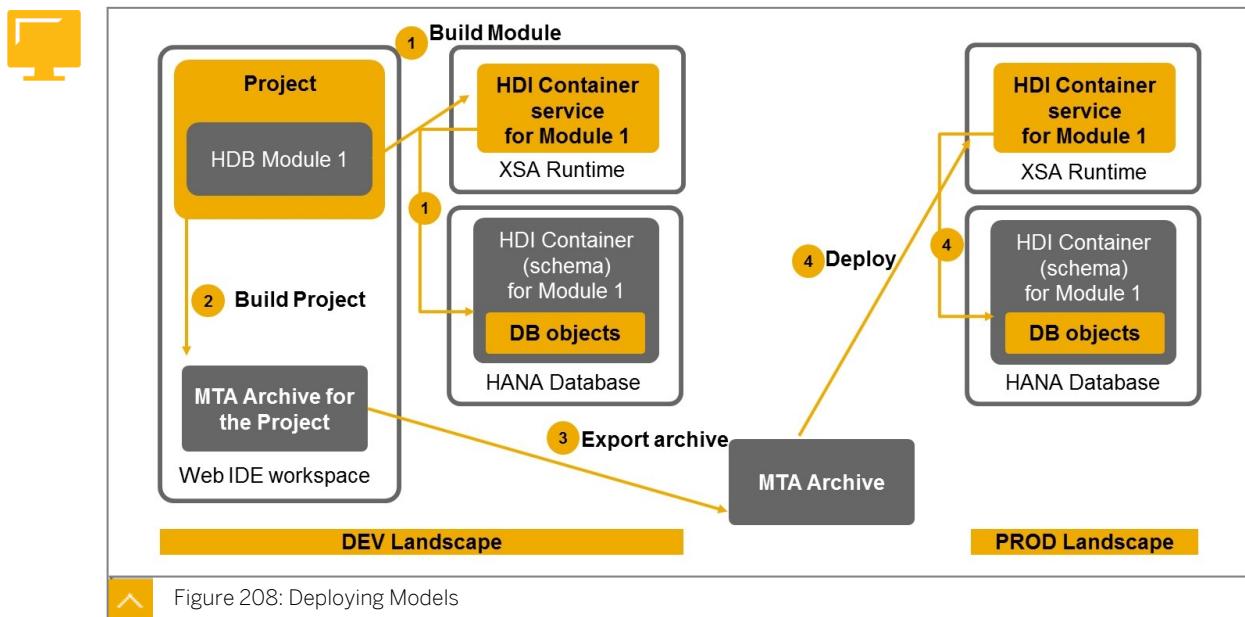
Let's now discuss the deployment of an application.

Application Deployment

We will only consider the deployment of a Multi-Target Application made up of one or several HDB modules. That is, we will not discuss the other tiers such as the user interface or application logic.

Deploy Versus Build

Let's first make a clear difference between building and deploying an application.



When you **work on a project**, you build the modeling content on a regular basis in order to test it, check dependencies, and so on. This is something you trigger from the SAP Web IDE for SAP HANA. To do that, you need to have a modeler role in the space to which you want to assign your XSA project.

When you build an HDB module (or some of its design-time content), if the build is successful, the corresponding runtime objects deploy to the corresponding space.



Note:

Remember that triggering a build at the project level does not build the HDB module(s) that the project contains.

On the other hand, **deploying an application** is something you do in a target space at a specific point in time, for example to test the application in a QA environment or set the application to production. Deploying an application can be done outside of the SAP Web IDE and the XS Advanced user who deploys the application does not need to have a developer role in the target space.

To deploy your modeling content to a target environment, such as QA or PROD, you must do the following:

1. Build the HDB module(s).
2. Build the entire Project in order to generate MTA archive file (.mtar).
3. Export the MTA archive.
4. Deploy the MTA archive in XS Advanced.

Between steps 3 and 4, you must of course make the MTA archive file “available” to the target landscape, for example by copying the file to a specific folder or using a dedicated transport tool.

**Note:**

As you see from the figure, Deploying Models, the Web IDE for SAP HANA is not involved during the deployment (step 4) on the target landscape. In particular, you do not need to create and build the project in the target landscape.

Source Files for Deployment

The key source file for application deployment is generated by a build operation at the project level in the SAP Web IDE. This creates a *.mtar* (MTA archive) file, which is technically a *.rar* compressed file. This file is named *<Application ID>_<version>.mtar* and can be found in your workspace, in the folder *mta_archives*. This archive contains the file *mtad.yaml*, which is an equivalent of the *mta.yaml* file but specific to application deployment.

Another important (though optional) file used for deployment is the *.mtaext* file. This is an “extension” file in which the space administrator who deploys the application can specify additions or modifications to the *mtad.yaml* file in order to pass the relevant parameters for the target environment during deployment. Typical examples of what you can specify in the extension file include the following:

- The name you want to give to the HDI container schema
- The actual name of a user-provided service in the target environment
- The actual name of another HDI container service in the target environment

The name of the HDI container(s) created during the deployment of your application is also an important parameter. This is the name of the HDI container resource declared in the *mta.yaml* file (by default, when you create the HDB module, *hdi-container*). You probably want to make it specific to your application, in order to identify it more easily among in the XS services.

For example, the SHINE application installed in your training environment defines its two container services as follows:

- *shine-container*
for the HDB module *core-db*
- *shine-user-container*
for the HDB module *user-db*

Deleting a Project

When you delete a project from the workspace in the SAP Web IDE for SAP HANA, the corresponding HDI container is not removed. This means there is still a service running in the XS Advanced runtime, and the corresponding schema still exists in the database.

To clean up the XSA runtime and the database, you must execute an `xs delete` command in the XS Advanced Command-Line Interface (XSA CLI).



LESSON SUMMARY

You should now be able to:

- Define the key settings of a project
- Manage the lifecycle of a project

Enabling Access to External Data



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Set up access to external data

External Data Access Setup

The SAP HANA Deployment Infrastructure (HDI) relies on a strong isolation of containers and a schema-free definition of all the modeling artifacts.

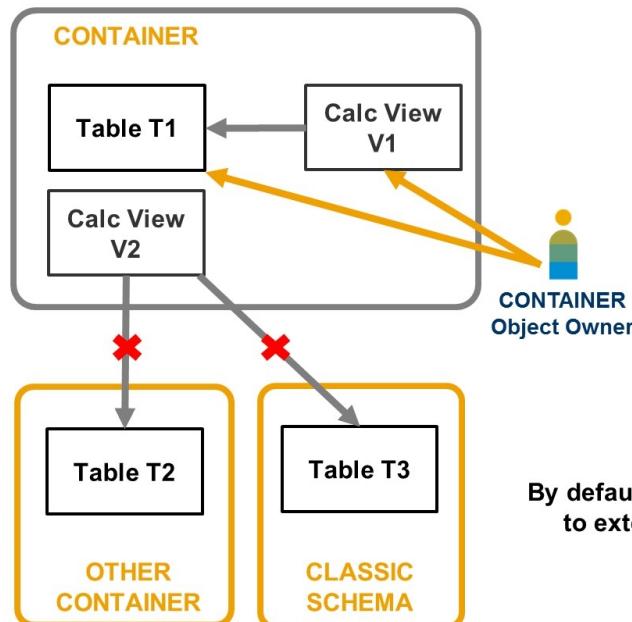
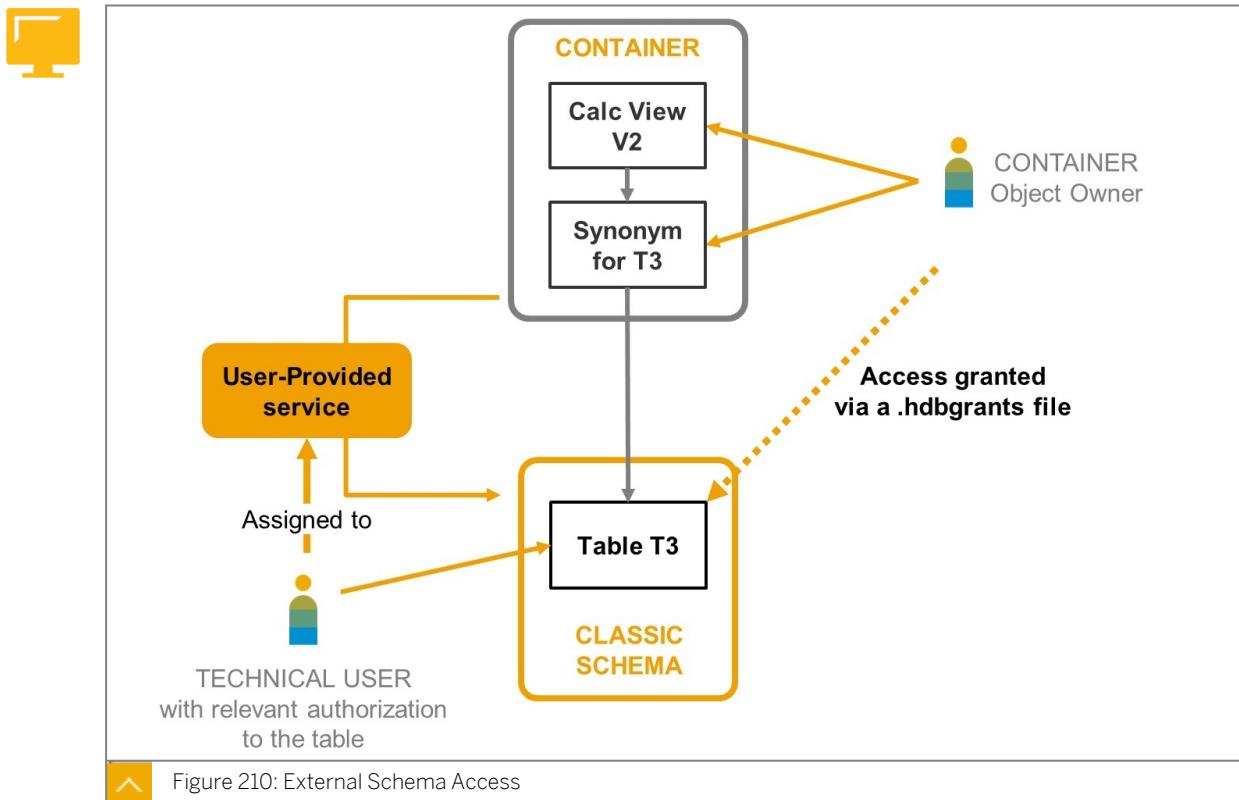


Figure 209: Container Isolation

By default, a container (that is, the database materialization of an HDB module) is fully aware of the database objects it contains (tables, calculation views, and so on), but has no access at all to other database schemas, whether it is another container or a classic database schema.

Defining Access to an External Schema – End-to-End Scenario

Let's consider the end-to-end setup of an access to an external schema.



First of all, you need a **user-provided service**. It is an XS Advanced service that an administrator must set up in the space to which your project is assigned. The main properties of this service are the following:

Properties of a User-Provided Service for External Schema Access

Parameter	Description
host	The identifier of the SAP HANA system in which the target schema is located Example: wdf1bmt7215.wdf.sap.corp
port	The port used to access the database (SQL port of the indexserver) Example: 30015
user	The name of the technical user assigned to the user-provided service
password	The password of this technical user
driver	com.sap.db.jdbc.Driver
tag	hana
schema	(optional) The name of a schema to which the user-provided service will give access. If blank, the service can be used to provide access to several schemas of the target database.

Technical User

The **technical user** assigned to the user-provided service must have the relevant authorizations to the target objects—and with the grant option—so that some (or all of) these authorizations can be granted to the object owner of your container.



Note:

The authorizations of the technical user define the maximum set of authorization that can be, in turn, granted to the object owner and application user.

The next step is to include a **reference to this service in the project descriptor file *mta.yaml***. This is done in the *resources* section, with the following code:

```
resources:
- name: <logical_name_of_user_provided_service>
  type: org.cloudfoundry(existing-service)
  parameters:
    service-name: <actual_name_of_user_provided_service>
```



Note:

It is possible to omit the *parameters* section if you put in the *name*: key the actual name of the user-provided service that is running in your project space.

You must also, in the *mta.yaml* file, **register the user-provided service as a dependency of the HDB module**. This is done in the *requires* section of the HDB module definition, with the following code:

```
requires:
- name: <logical_name_of_user_provided_service>
```

Then, you **can grant the relevant authorizations** to the object owner and application user. This is done in a *.hdbgrants* file.



Note:

You will learn more about this in the unit, Security in SAP HANA Modeling.

Creating Synonyms

The final step is to **create synonyms to the target objects** of the external schema. The synonyms declaration is done in a *.hdbsynonym* file. There are two ways to provide the definition of the synonyms:

- In the same *.hdbsynonym* file (like in the example below),
- In a dedicated *.hdbsynonymconfig* file

These file types can be edited either with the code editor, as in the example below, or a dedicated synonym editor.

Extract of an EPM_MODEL.hdbsynonym File



```
{
  "HA300:::SNWD_BP": {
    "target": {
```

```

        "database": "H00",
        "schema": "EPM_MODEL"
        "object": "SNWD_BP",
    }
},
"HA300:::SNWD_PO": {
    "target": {
        "database": "H00",
        "schema": "EPM_MODEL"
        "object": "SNWD_PO",
    }
}
}

```

The "database" field is optional, needed only in cross-database access scenarios.



Caution:

Public synonyms CANNOT be used in XSA projects to access external objects such as tables. Local synonyms must be defined INSIDE the container to access the data.

When adding a data source to a node, there are mainly two possible scenarios:

- Your data source is defined locally in your container or a synonym for an external data source (in another schema or container) already exists in your local container
In the *Find Data Source* dialog box, select the existing data source and choose *Finish*.
- You need to access an external data source that is not yet referenced by a local synonym.
You must first define the synonym (as described above) in a *.hdbsynonym*, and maybe adjust the *.hdbgrants* file to make sure the external object will actually be accessible.

From SAP HANA 2.0 SPS02 onwards, the synonym can be created automatically at the time you need to access the external data source from the Calculation View editor. To do that, in the *Find Data Source* dialog box, you must first select the external service and then enter a search criteria. Then, select the object and choose *Next*. You can adjust the synonym name (without changing the namespace). Choose *Finish*.

Two files, *grantor.hdbgrants* and *synonyms.hdbsynonym*, are created (if needed) or adjusted in the same folder where the consuming calculation view is located. An entry in the *.hdbgrants* file is created for each external object.



Note:

The location of the automatically generated *.hdbgrants* and *.hdbsynonym* files might not suit your requirements, in particular if you prefer to keep all synonyms defined in a more central place (for example, a dedicated folder). In that case, we recommend that you go on creating the synonyms manually.

Dynamic Target Schema Identification

If you use a *.hdbsynonymconfig* file, this gives you additional possibilities to provide the actual target schema name on deployment time. There are two different ways to achieve this:

- You can retrieve the schema name based on what is specified in the user-provided service (or an existing HDI container in case of cross-container access)

For example, you can replace "schema": "EPM_MODEL" with "schema.configure": "<logical_name_of_user_provided_service>/schema"

- You can use a logical schema name instead of a real schema name.

For example, you can replace "schema": "EPM_MODEL" with "logical_schema": "<logical_schema_name>"

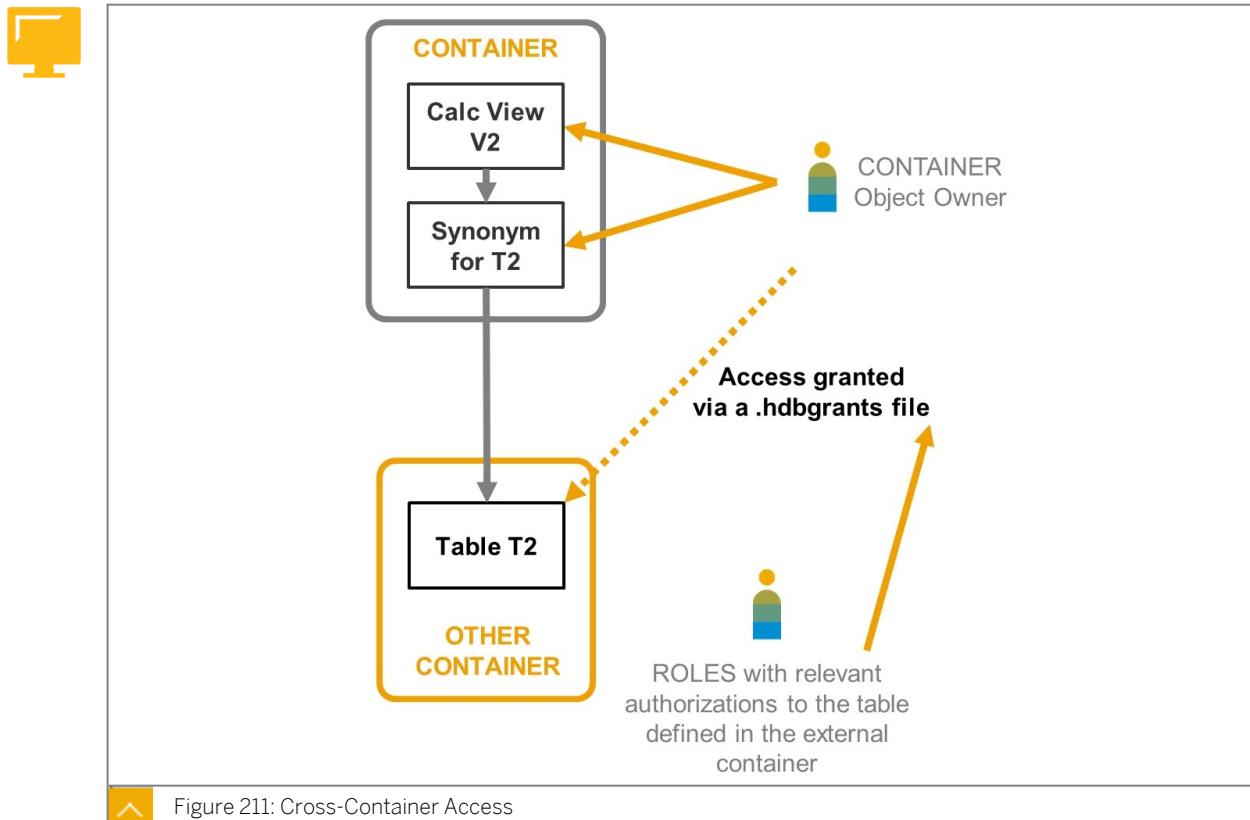
The mapping is done in a `.hdblogicalschema` file.



Note:

You can learn more about these two techniques in the SAP HANA Developer Guide for SAP HANA XS Advanced Model. Search respectively for *Templates for HDI Configuration Files* and *hdblogicalschema*.

Cross-Container Access



When you need to access data from another HDI container, the setup is relatively similar to what you have just learned for an external (classic) database schema. Let's point out the main differences:

- There is no need for a user-provided service if the external container service is running in the same space as the one your project is assigned to.

Indeed, you can directly reference in your `mta.yaml` file the external HDI container service.

- You must create relevant roles inside the external container.

- The *.hdbgrants* file does not refer to database object authorizations of the technical user assigned to the user-provided service, but to the dedicated roles created inside the external container (see the previous point).

If you want to learn more about synonyms, external schema and cross-container access, in addition to the SAP HANA documentation, you might want to consult a series of three detailed blog posts, starting at the following URL: <https://blogs.sap.com/2017/01/06/synonyms-in-hana-xs-advanced-introduction/>.



Note:

The blog posts are about SAP HANA version 2.0 SPS00, but most of the information they provide is still relevant for more recent versions of SAP HANA.



LESSON SUMMARY

You should now be able to:

- Set up access to external data

Working with GIT Within the SAP Web IDE

LESSON OVERVIEW



LESSON OBJECTIVES

After completing this lesson, you will be able to:

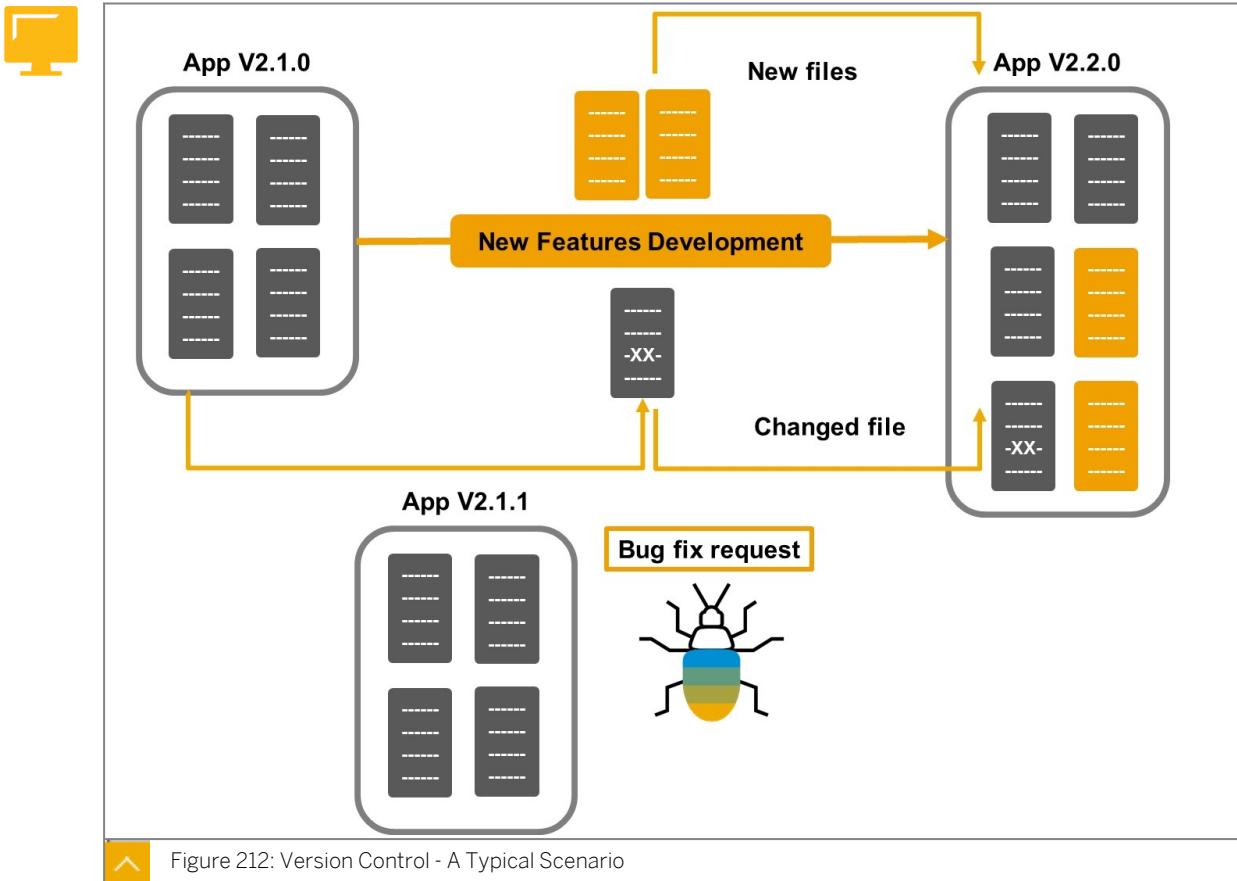
- Use the Native Git Integration of the SAP Web IDE

Overview of Git

Why Should You Use a Version Control System?

Think about an XSA application. This application, with its different modules (SAP HANA Database Modules, UI, and control flow modules), consists of a number of files, organized in different folders.

Now, suppose the last released version of the application is V2.1.0, and you're currently preparing new features for the upcoming minor release V2.2.0, which is scheduled in a few weeks. You might have imported the MTA project in your SAP Web IDE workspace and started developing the new features that are planned for V2.2.0. This development means essentially new files, as well as modifications to existing files. Maybe the deletion of a few ones as well.



At some point, you receive an important e-mail from support saying that a bug in V2.1.0 needs an urgent fix. This cannot wait until the next minor version, and requires a patch (V2.1.1).

So, how do you handle this request? At the moment, your current development is not finished, not tested, but you need to patch your version 2.1.0. Of course, you want to start from the last release (you do not want to include any part of the future functionality into the patch), but your patch might affect some files that you already modified as part of the development of new features.

This is where a version control system comes into play. It allows you to keep a complete change history by using kind of milestones during the development of your code, at a very fine-grained level if needed. And you can also branch your code, which means, you can develop and test different features in different parallel development threads (branches). If a feature branch is good to go, you can merge this branch with the main branch. If it is not, you can continue your development, or even get rid of this branch if you realize that a development option for a feature was not relevant and you need to think about it again.

Key Benefits of a Version Control System



- Source code backup
- A complete change history
- Branching and merging capabilities
- Traceability (for example, connecting a change to project management software)

To learn more about version control systems, you can have a look at this page: <https://www.atlassian.com/git/tutorials/what-is-version-control>.

Git in a Nutshell

Git is a Distributed Version Control System (D-VCS) used to manage source code in software development, or more generally to manage the lifecycle of any set of files.

It allows one or several developers to work locally with their own copy of the Git repository, which contrasts with a traditional client/server architecture.



Note:

Git can also be used even out of a collaboration context, to help you control the development of a project on which you work alone, thanks to a number of capabilities.

The architecture of Git is distributed. It is somewhere between purely local and purely central. A local architecture would make it difficult for several developers to collaborate. A centralized one allows collaboration, but in some cases, a developer need to block a piece of code on the central repository while working on it.

Instead of this, Git is designed so that every developer can keep the entire history of a project (or only a part of it, depending of their needs), locally on their computer.

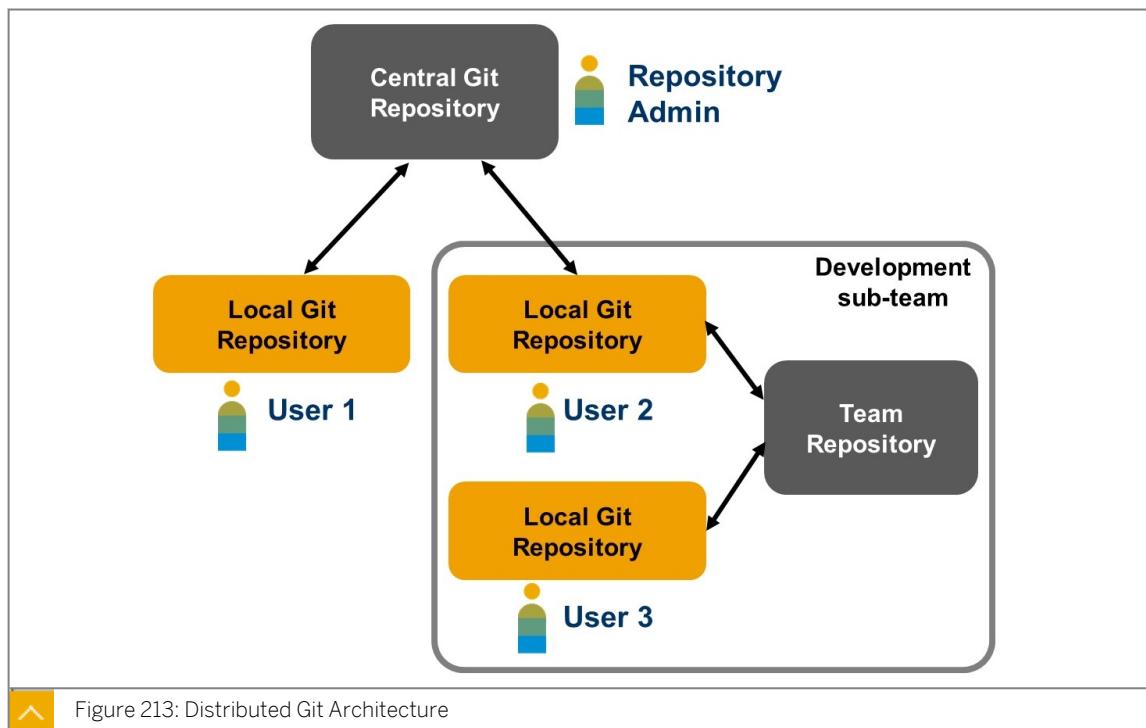
Git is a free software distributed under GNU GPL (General Public License).



Note:

Note that the initial purpose of Git is NOT code review or automated deployment of applications. For this, other tools exist –often with advanced connectivity to Git– such as **Gerrit** for code review, or **Jenkins** for automated integration and deployment.

Git Architecture



In a classical Git architecture, developers work in a local repository on their computer, and connect on a regular basis to a central repository to share their contribution and get the contribution from other developers on the project.

Git also supports easily several remote repositories for a single project. If needed, it is possible for a sub-team of developers to have their own sub-team repository to collaborate, and synchronize their changes with the central repository when needed.



Note:

In Git, it is even possible for a developer to define the local repository of another developer as a remote repository and to synchronize his development. This requires a network access and the relevant authorizations.

The shared Git repositories can be hosted on the own company's IT infrastructure, or on Git hosting services such as GitHub –one of the most popular–, Helix (Perforce), Bitbucket (Atlassian), and many more. A lot of companies offering Git hosting services also provide additional services such as code review or issue tracking.

Key Benefits of Git

Git has been designed with Security, Flexibility and Performance in mind. Almost every operation is performed locally. The branching model provides an extreme flexibility.

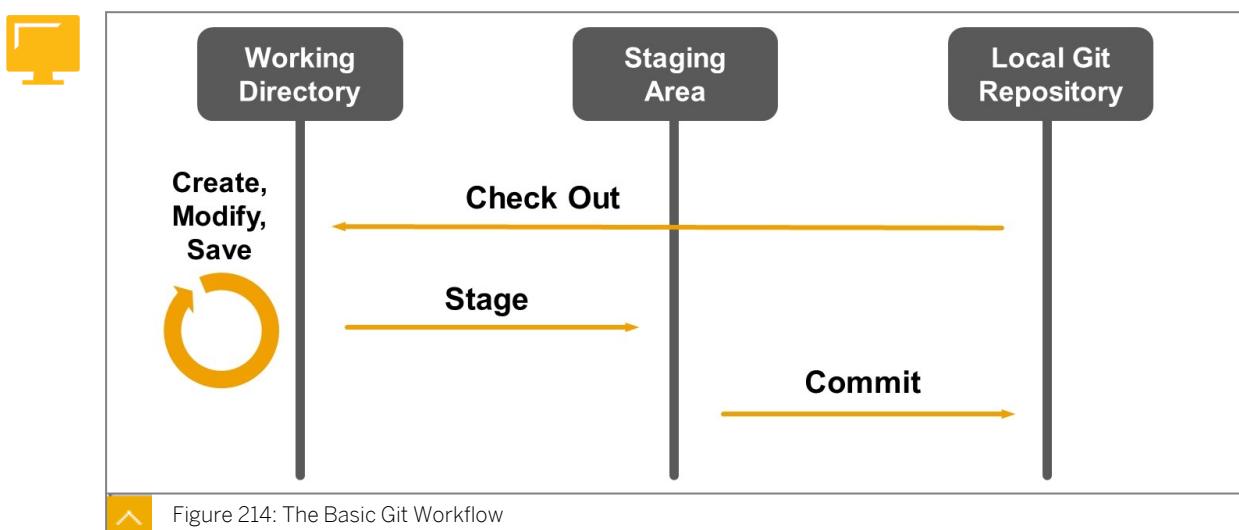
To learn more about Git, go to: <https://git-scm.com/>.

The Lifecycle of Files in Git

When you work with files locally in Git (the case of remote Git repositories will be discussed later on), this involves three major “logical” areas.

- The Working Directory
- The Staging Area, also known as INDEX.
- The (local) Git Repository

These are not all real areas, in the sense that a given file is not necessarily materialized in each of them. Let's explain this with a diagram.



The way to manage files in a local Git repository is very straightforward, relying on a small number of actions.

When you create a file, this file is initially stored in your **Working directory**. You can also modify (or even delete, if needed) a file that you have previously exposed in your **Working Directory** with a **Check Out** command.

At this stage, your changes – even if they are saved in your working directory – are not yet part of the Git history. To update the Git history, you must perform what is called a **Commit**. This is what will create a new point in Git history (a so-called Commit), referencing a number of changes since the previous commit.

However, do you want to include all of the exact changes in your next commit? This is where the **Staging Area** comes into play. By staging a file, you mark this new or modified file so that it is included in the next Commit. You can of course stage several files (this is very common), or even stage all the current changes of your working directory.

Let's put it another way: The staging area is the “virtual” place where you put all the modifications that you want to include in your next commit.

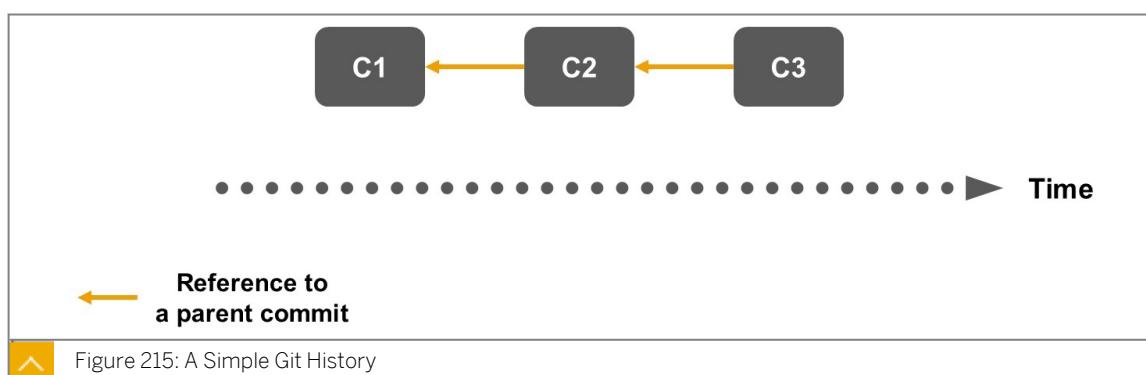


Note:

When your working directory contains changes that you do NOT want to include in your next commit, you just need to make sure you do NOT stage the corresponding files before committing.

The Git History

Git is very good at representing the history of a project in a simple way, as the succession of commits.



Over time, all the commits you execute in your project are added to the history, and each commit (except the initial one) references its parent commit.



Note:

Actually, you will see later on that in the case of a *Merge* operation, a commit can have two parent commits.

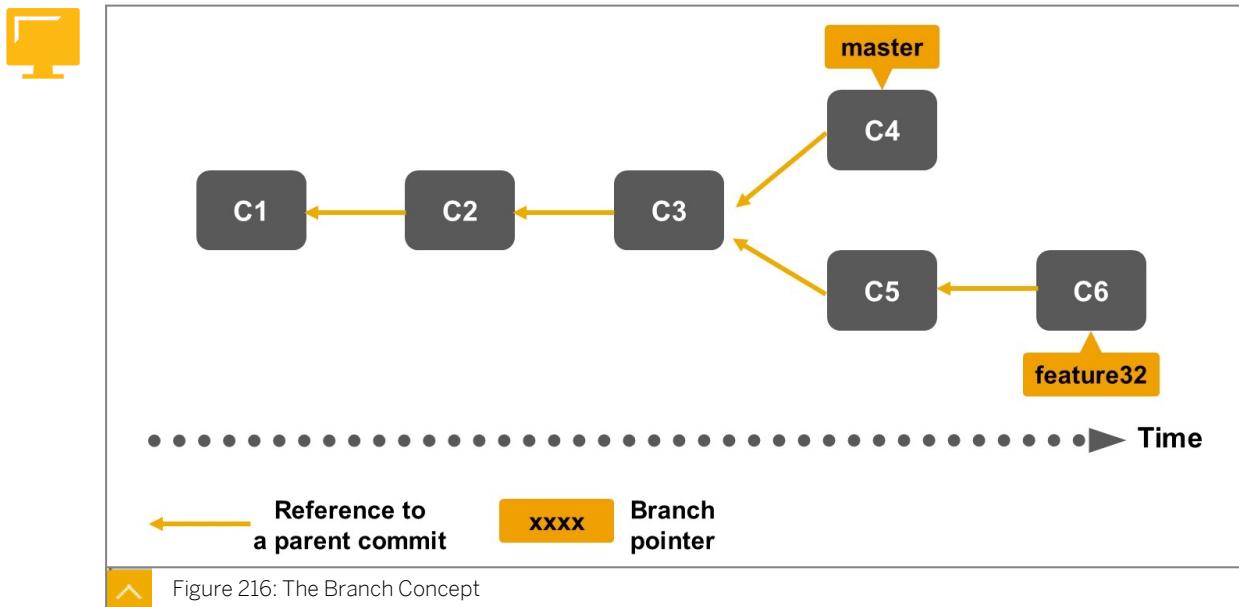
With each commit, Git keeps record of the commit date, the identity of the developer who executed it, and useful information about which files were affected (added, modified, or deleted).

The Branch Concept in Git

Branching is one of the core concepts of Git, which provides a huge flexibility at a very low cost. So what is a branch?

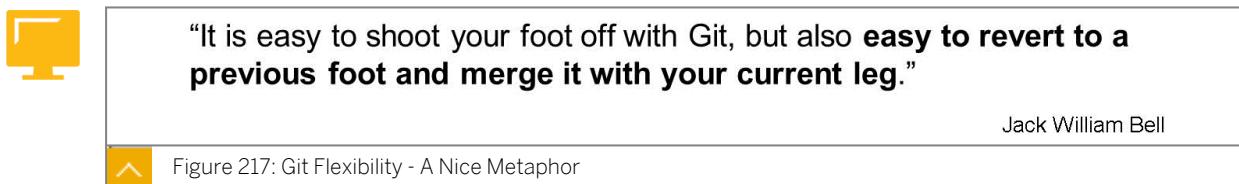
From a conceptual standpoint, a branch is a series of commits. Technically, a branch is just a pointer that references a particular commit of the project history.

Each Git repository always has at least one branch. The default branch is generally called **master**.



For many different purposes, you can create a new branch and commit your changes to an existing branch.

Let's describe the figure, The Branch Concept. After commit C3, a new branch *Feature32* has been created to support the development of a new feature. Two commits, C5 and C6, have been made to this branch. In the meantime, additional changes have been committed in the *master* branch (commit C4).



Git Integration within the SAP Web IDE for SAP HANA

Git Clients

Since its creation in 2005, Git provides its complete set of features through the command line. Over time, a number of Graphical User Interfaces have been developed. They generally include a subset of the Git functionality available through the command line.

The SAP Web IDE for SAP HANA provides an embedded Graphical User Interface for Git.

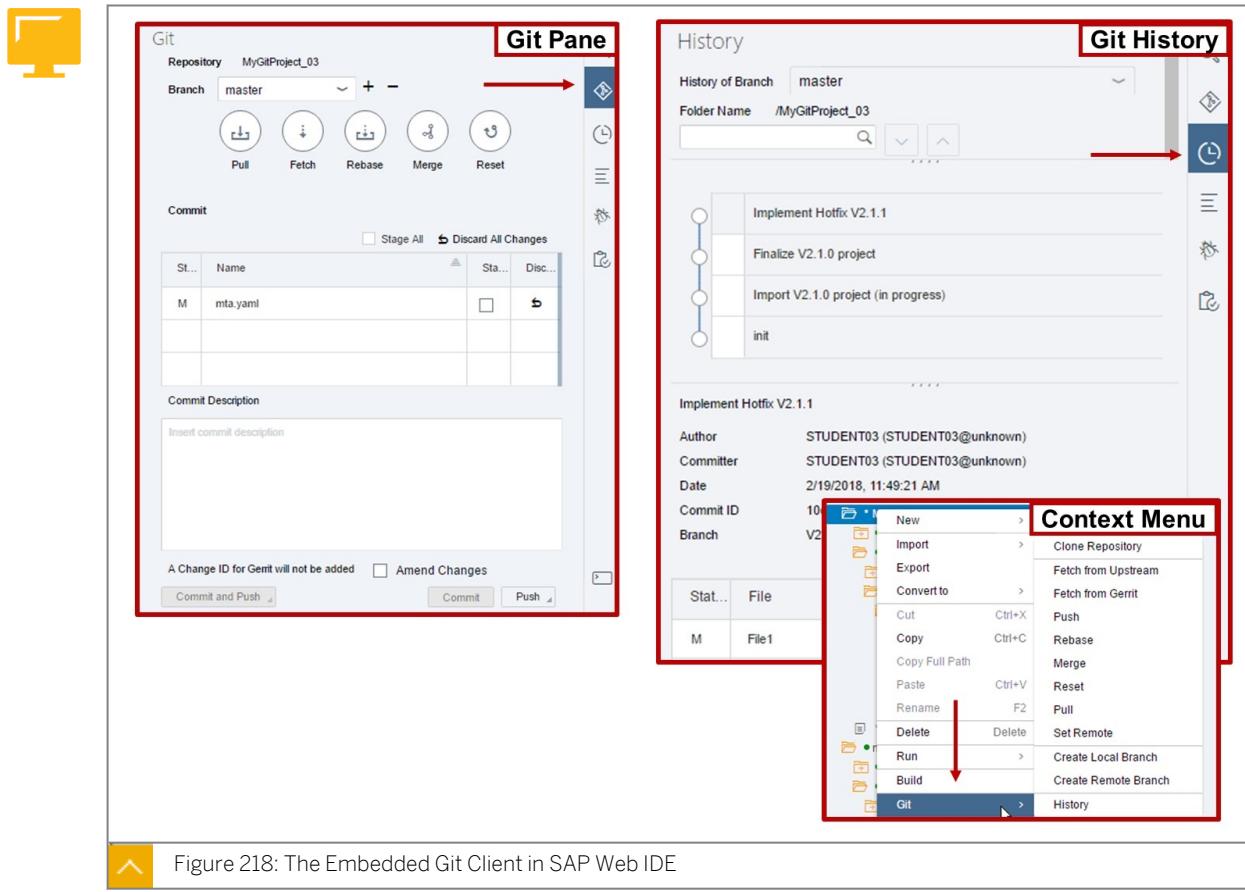


Figure 218: The Embedded Git Client in SAP Web IDE

This client includes two main Git panes, accessible from the right toolbar of the *Development* perspective, or the *View* menu, as well as a context menu and a set of icons displayed next to the files and folders in the workspace.

In addition, in the *Project Settings*, a dedicated section *Git repository configuration* displays the entries of the Git configuration file. These entries include, among others, the Git user name and e-mail, remote repositories (if any), and so on.



Note:

It is possible to edit the Git configuration file manually, but this requires advanced Git knowledge. In many cases, this is done automatically, based on the actions you perform in the GUI.

The SAP Web IDE includes the core features of Git (commit, history, management of local and remote branches, and so on). The support of additional Git features is included on a regular basis in some new releases of the SAP Web IDE. For example, the *Git Stash* feature (allowing to set aside uncommitted changes temporarily) will be included in the SAP Web IDE for SAP HANA 2.0 SPS03.

Starting to Use Git in the SAP Web IDE

Working with Git in the SAP Web IDE is something you decide project by project. A project might have Git functionality enabled, when another project does not.

If you decide to activate the Git functionality in the SAP Web IDE, there are different approaches. You will choose the one that is best suited to your needs... keeping in mind that in some use cases, they get to the same result.

- Clone a Remote Git Repository

This is an easy way to copy an existing project stored on a Git server (or Git hosting service), and if needed, start contributing to the project.

In some cases, you will not contribute to the project, but you just want to get a copy of a project that a development team made available to you, either because you have read access to the remote repository, or because it is available publicly. This is the case of the SHINE for XSA project, which anybody can clone without any credentials.

- Initialize a Local Repository

By default, a project that you create or import in the SAP Web IDE does not have the Git functionality activated. This is the case, for example, for the *HA300_##* project you've worked on so far.

However, activating Git for a project is extremely quick and simple: via the context menu of the project, you just tell the SAP Web IDE to define the folder where your project is stored as a Git repository. Basically, this adds to this folder a few files that Git will then use to track the history, store snapshots of changes to the repository, and manage settings.

Initializing a local repository is what you need when you want to work with Git, but alone (locally), on a project. That is, without sharing it or having other developers contribute to it (at least at some point). Indeed, you do not need a remote Git repository at all, in this case.



Note:

When you initialize a local repository (which results in an empty initial commit), all your project files are flagged as new and are staged. You are ready to add all your project files to the local Git repository by executing a commit.

- Set Remote Repository

This is what you need to associate your local Git repository (project) with a remote Git repository. For example, when you have already developed content locally and would like to share this content with other developers.



Caution:

This is NOT needed when you clone a remote repository, because in that case the local repository (the clone) is automatically associated to the remote repository you cloned from.

Whenever you work with a remote Git repository, you need the Git Clone URL. In addition, you might need credentials to access the repository if it is not public.

Working with Files in a Git Project

Basically, modifying your project content (with or without Git) means creating, modifying, and deleting files. With Git, all these changes are tracked in your working directory as soon as they are done –for example, when you save a file you have just modified– and listed in the *File Status* area.

Then, for each file, you have the following possibilities:

- Stage the modification so that it will be included in the next commit

- Leave the modification unstaged (it will not be included in the next commit)
- Discard the change

Staging or Discarding can also be done for the entire set of modifications.

The next step is to commit your changes, which will add a next commit to the history of the branch.



Note:

The concept of branches in Git, already introduced, will be discussed in more detail later on. For now, let's just consider that you are working on a single local branch, for example, *master*.



Working
Directory

Create,
Modify,
Save



Staging
Area

Check Out

Stage

Local Git
Repository

Commit

Figure 219: The Basic Git Workflow

To materialize this workflow, each file is assigned a Git status. This status is represented by an icon in the SAP Web IDE workspace and/or the *File Status* area of the *Git Pane*. The list of possible file statuses is as follows:



Workspace Icon *	Meaning	File Status in the Status Table
	New, not staged	Untracked (U)
	Modified, not staged	Modified (M)
	New/modified and staged	New (N)
		Modified (M)
	Deleted (staged or unstaged)	Deleted (D)
	Committed	[File is no longer listed]
	Conflicting	Conflict (C) – Only during a merge or rebase operation

* Only one icon is displayed for folders that contain files with different statuses
** The icon identifies folders from which files have been deleted



Figure 220: Git Status of Files in the SAP Web IDE



Note:
The Conflict (C) status will be discussed later on.

Committing Changes

When you commit changes, you add these changes to the Git history and provide a commit description. From this moment on, the *Git History* pane will include this commit, and provide the identity of who committed, the date, the commit description, and details on which files were affected by the commit. Note that committing changes does not modify your working directory. The committed files are still available for further modification, and the new or modified files you haven't committed yet are still here for an upcoming commit. You can also discard the changes to these uncommitted files.

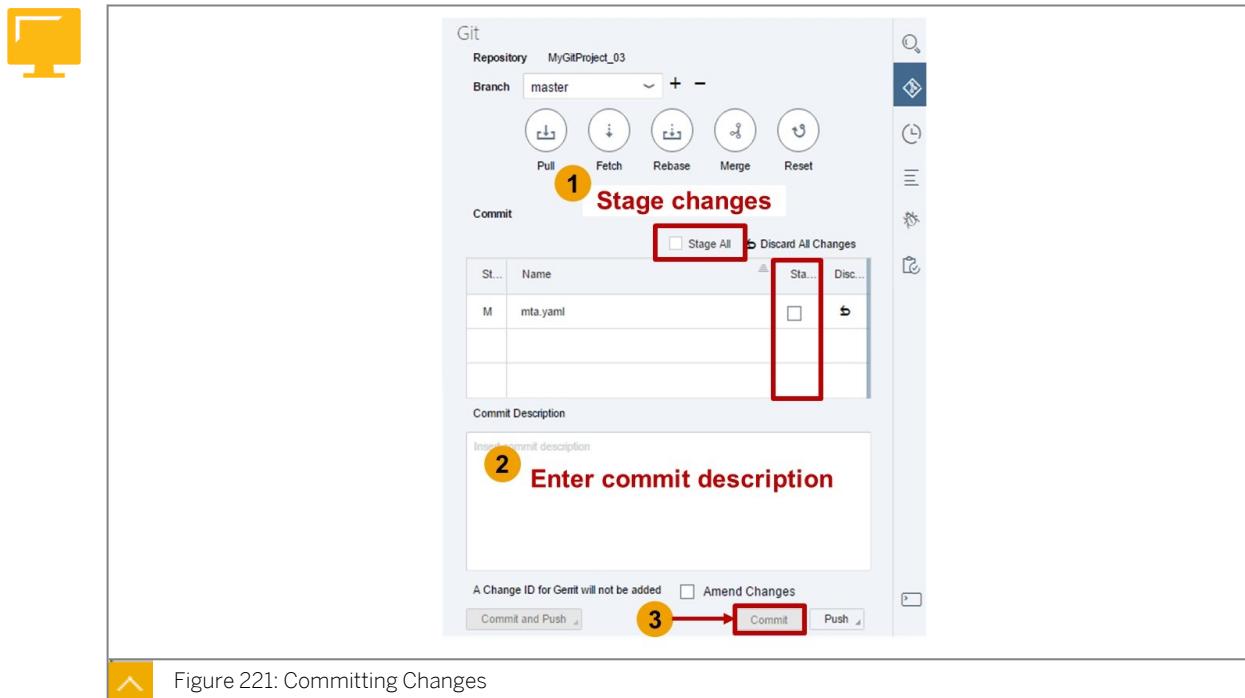


Figure 221: Committing Changes

The commit description is an important info to allow the readability of your changes, in particular when you collaborate with other developers. You can find a number of blogs and tutorials on how to write a good commit message. Here are a few recommendations:

- Start with a relatively short subject (50 characters max), using imperative mood
- Separate the subject and body with a blank line
- Provide info about what the change does, and why (for example, what issue it solves)
- If relevant, provide the URL of a related issue or specification in another system (for example, JIRA)

It is possible to **amend** a previous commit. This is a way to replace the very last commit by a new one, after you staged additional files. The original commit description displays so that you can modify it before committing again.



Caution:

You must not amend commits that have been shared with other developers, because this would modify a (shared) history on which others might have already based their work.

Working with Git Branches in the SAP Web IDE

Local or Remote Branch

Git allows you to create both local and remote branches. So how to choose?

One key principle is that only remote branches are visible to others: your local branches are for you and you are the only one who works in these branches.

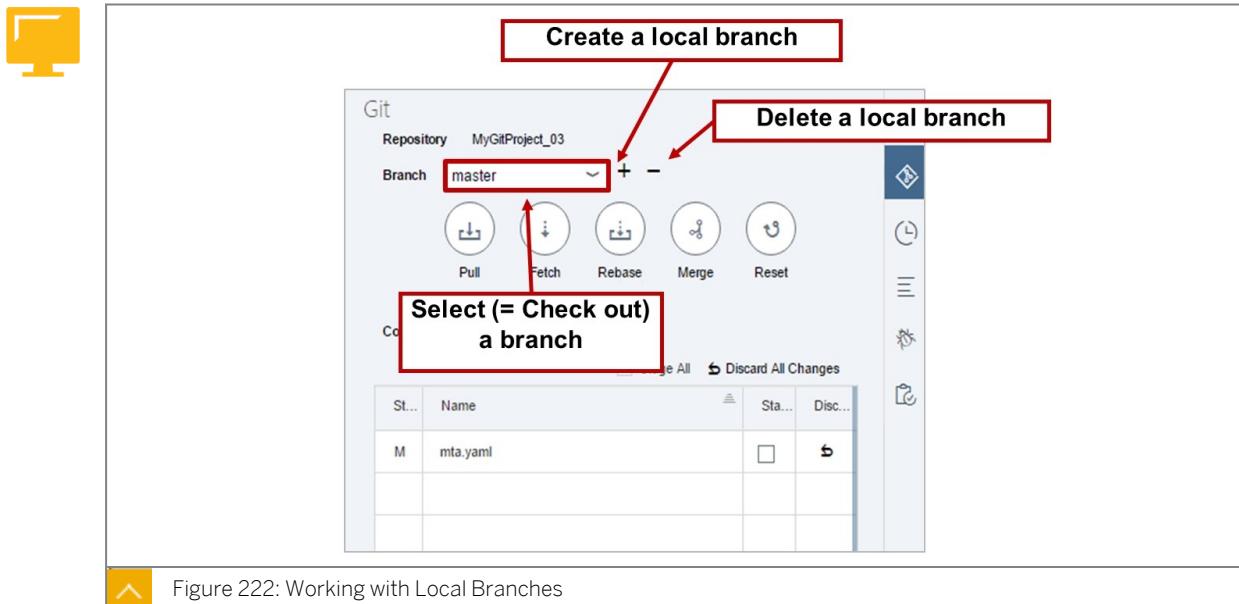
Another important principle in Git is that you never commit changes directly in a remote branch. Instead, you always commit changes in one of your local branches first. Then, if needed, you can transport the commits to a remote branch.

Let's take an example. You are working on a project with other developers. At some point, you need to test something on your own, without sharing it with other developers. Then you will create a local branch for that. If, later on, you need to share this with others, you have the possibility to create a remote branch based on your local branch.

We will come back to remote branches later on. But for now, let's discuss how you work with local branches.

Working with Local Branches

You can create a new local branch from the *Git Pane* or the Git context menu. The only thing you need is to define a source branch (which could be remote or local) and give the new branch a name. Just after a new branch is created, it is absolutely identical to the source branch.



When you have several branches in your project, you must carefully decide to which branch you want to commit your changes. To do this, you select a branch in the *Git Pane*. In Git terms, this is known as *Check Out*.

Checking out a branch also updates your working directory to reflect the exact current state of files in this branch. In other words, your working directory is likely to change when you switch from a branch to another, except if these branches are identical (that is, if they point to the same commit).



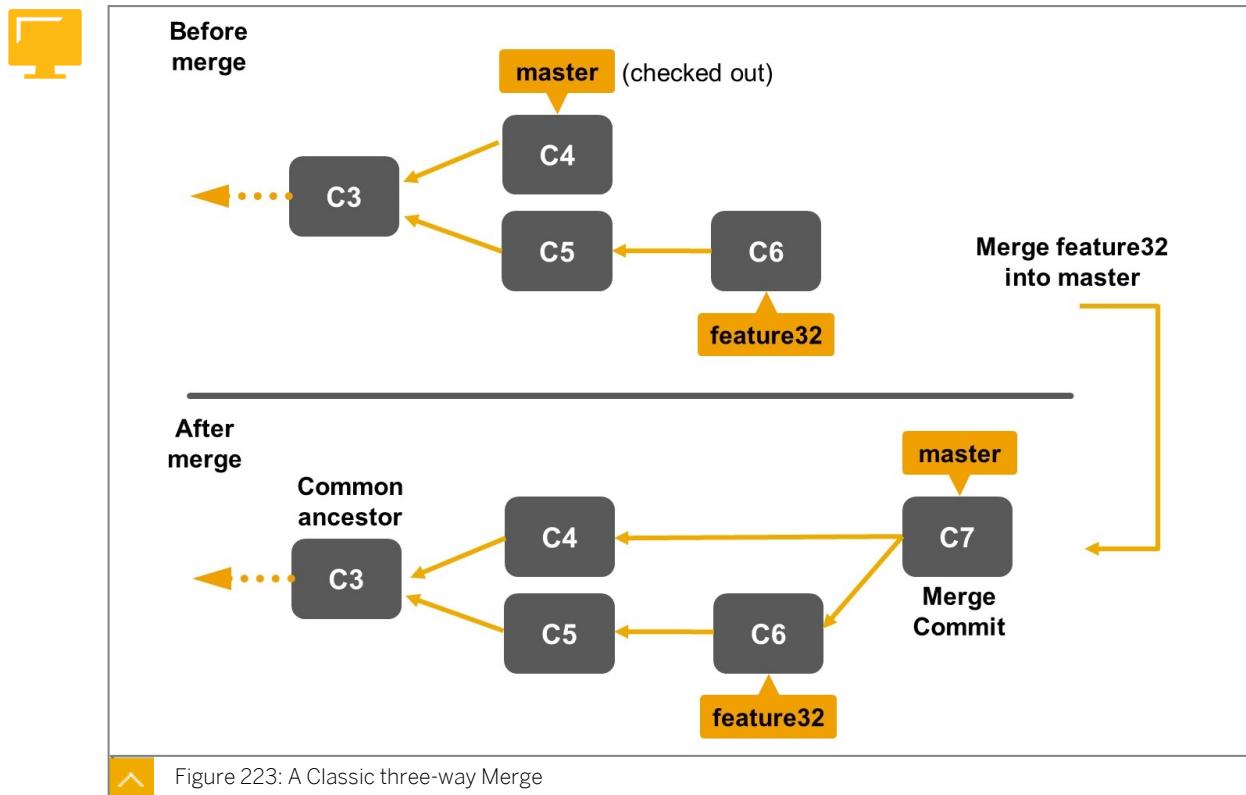
Caution:

Issues might occur when you have changes that are not committed yet and you try to check out another branch. For example, you have made a modification to a file that exists in both branches. To avoid this, the best way is to have a clean working directory without uncommitted modifications before you check out a branch.

Combining Changes from Two Branches

When changes affect several local branches, you might want to combine these changes. For example, to include a hotfix or a new feature in your master.

Let's assume that commits have been executed in two different branches, *master* and *feature32*.



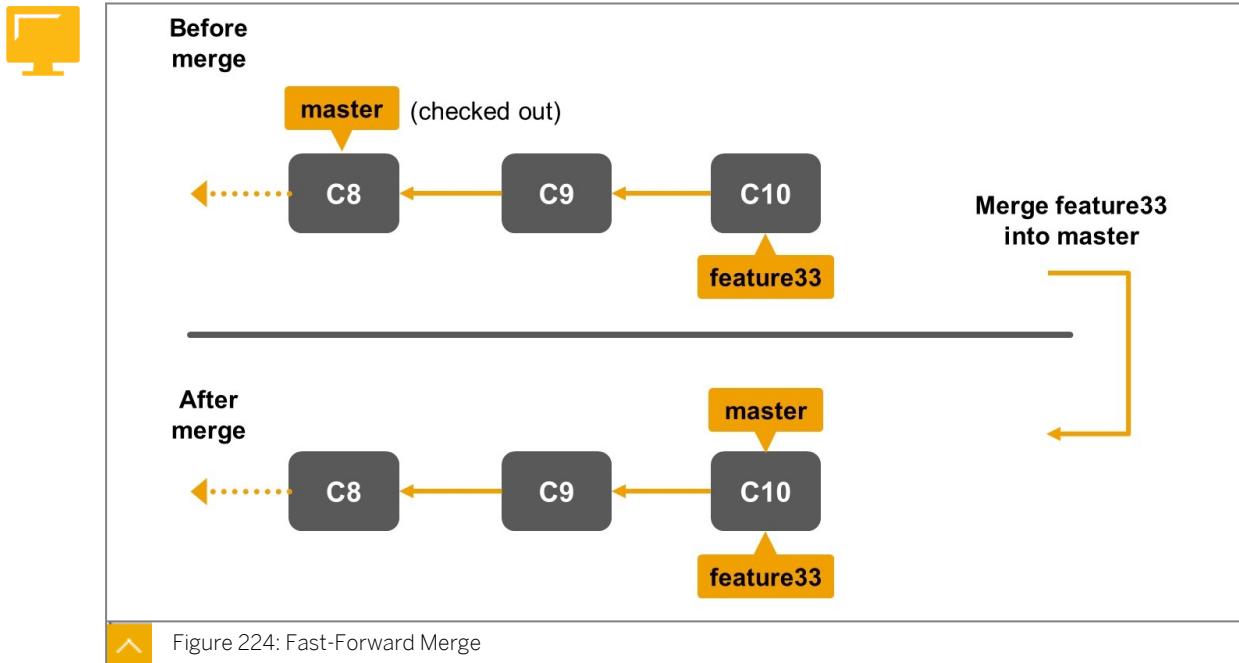
A Git merge operation affects the branch that is currently checked out. Git creates a new commit (merge commit) that will combine the changes of another branch with the changes of the checked out branch since their history diverged. As you see in the figure, A Classic three-way Merge, the merge commit has two parents, which are the last commits of the two branches that you merged together.



Note:

After the merge, the two branches are NOT identical. In particular, *feature32* does not include C4.

A special case for Git merge is when the branch you want to merge into (the checked out branch) points to a commit that exists in the branch you want to merge. Meaning, no additional commit has been made to that branch.



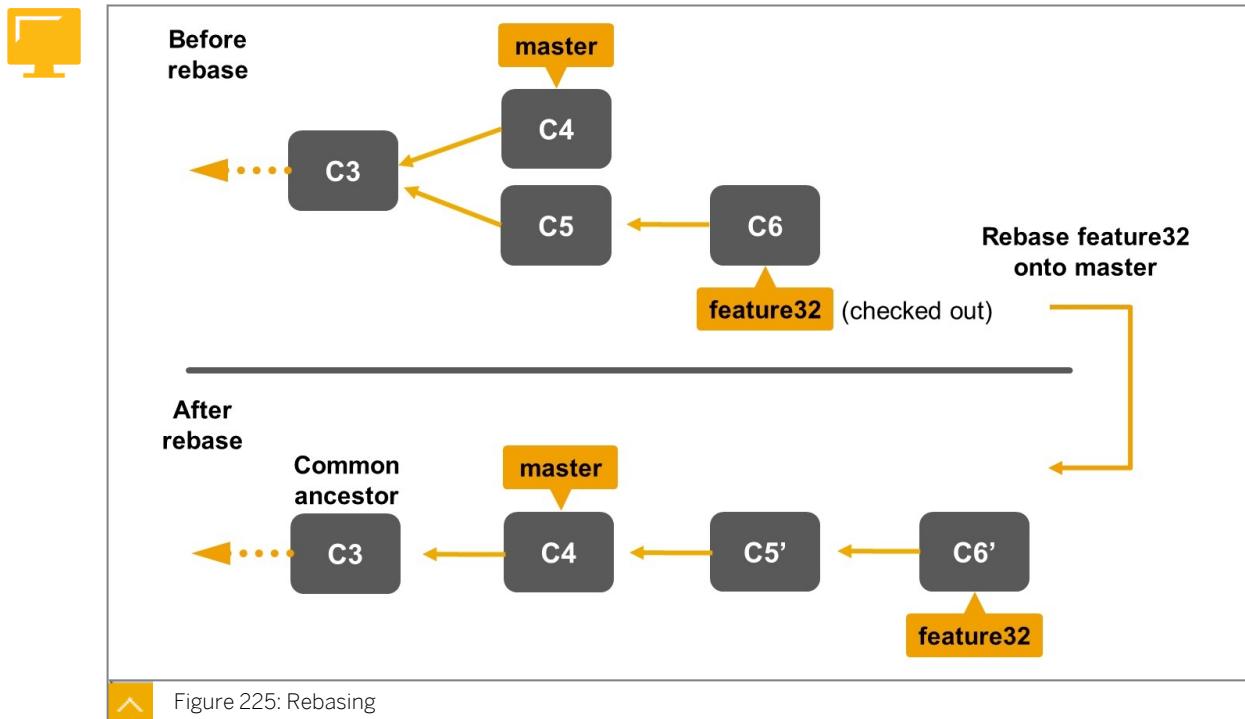
In this case, there is no need for an additional merge commit. Indeed, the pointer of the checked out branch is just pushed forward to the last commit of the other branch. Git calls this a Fast-Forward Merge.



Note:

A fast-forward merge only has effect if you check out the branch that is behind in the commit history. In the scenario of the figure, Fast-Forward Merge, if you check out the branch *feature33* and execute a “merge *master* into *feature33*”, no change will be made at all.

Now, let's introduce an alternative to merging when you want to combine changes from two branches. It is called “Rebase”.



When you rebase a branch B2 (here, *feature32*) onto branch B1 (here, *master*), you basically take the history of commits in B2 and replay these commits on top of B1. Then, you can execute a fast-forward merge on B1 (*master*) so that it points to the same commit as B2 (*feature32*).

As you see, compared with the classic three-way merge we discussed earlier, this keeps the history of the *master* branch linear. The final content of the *master* branch is identical, but the history looks different.

Rebasing is sometimes useful when you work on a purely local project, and also when you want to prepare a clean commit history that will flow naturally on top of a branch to which you contribute.



Caution:

You should always keep in mind the golden rule to use rebase: **You can only rebase changes that you have NOT shared with anyone**. Indeed, a commit that is communicated externally (for example, by pushing changes to a remote Git server) might be the starting point of some work from other developers. If you rebase your changes and push them again to this Git server, this will alter the history and might cause a lot of issues.

Managing Merge / Rebase Conflicts

A conflict occurs during a merge or rebase when one or several files are affected by concurrent modifications from both branches.

In this case, you must analyze the files and determine how to solve the conflict. Which means, decide how to produce a code that suits the requirements that the changes in each branch addressed.

Each conflicting file is identified with a dedicated red icon >< in the workspace (project tree). To solve a conflict, you must open the file, in which the content from the two branches are

presented together, and contains a special markup to identify which changes come from which branch.

Example of a Merge Conflict in a Table Function

```
select
"FIRST_NAME",
"LAST_NAME",
"SEX",
"LANGUAGE"
from "HA300::SNWD_EMPLOYEES"
<<<<< HEAD
where contains("LAST_NAME", :lastNameFilter, FUZZY(0.55));
=====
where contains("LAST_NAME", :lastNameFilter, FUZZY(0.6));
>>>>> hotfix
```

HEAD is the pointer to the checked out branch (the one you're merging INTO) and, here, hotfix is the name of the branch you're merging.

Example of a Rebase Conflict in a Table Function

```
select
"FIRST_NAME",
"LAST_NAME",
"SEX",
"LANGUAGE"
from "HA300::SNWD_EMPLOYEES"
<<<<< Upstream, based on master
where contains("LAST_NAME", :lastNameFilter, FUZZY(0.55));
=====
where contains("LAST_NAME", :lastNameFilter, FUZZY(0.6));
>>>>> 93f6499 Change fuzziness threshold for employee search function
```

In a rebase, the branch on top of which you want to rebase is identified with *Upstream, based on <branch name>* and the specific commit from the branch that you are rebasing is listed, with its description.

In both cases, you must adapt the code, remove the specific markup, and sometimes include comment lines in your code to explain how you solved the conflict, and then save the file.

When this has been done for each conflicting file, you can then proceed as follows:

- For a Merge: stage the modified file(s), enter a commit description and commit your changes.

This description will replace the default merge commit description (the one that you would get if no conflict occurred).

- For a Rebase: choose *Continue*.

In the rebased branch history, the conflict resolution is stored in the “replayed” commit. As if there had not been a concurrent change, or to put it another way, as if you had made your change in the *hotfix* branch based on the modified version from the *master* branch.

Working with Remote Branches

Working with remote branches is not fundamentally different from what we have discussed for local branches. First of all, because the changes you make to your files are always committed to a local branch, and also because combining the changes from a local branch and a remote one works in a similar way as combining the changes from two local branches. However, there are a few differences:

- You need to connect to a remote Git server (or a Git hosting service)

- At some point, you will share your changes (commits) with others, and also receive changes (commits) from others.

Connecting to a Remote Git Repository

As discussed earlier, in the section, Starting to Use Git in the SAP Web IDE, there are two ways to connect to a remote Git repository:

- Clone from a remote Git repository
- Set a remote (link your local Git repository to a remote one)

An essential piece of information you need for that is the **remote repository URL**.

In addition, cloning, setting a remote, and then, most of the regular interactions with a remote repository, require **credentials**.

However, some repositories are public, which means that they can be accessed anonymously, without credentials. For example, this allows you to clone in your repository some code that the owners have decided to share publicly.

For security purposes, on the remote Git server side, there is a repository administrator who manages users and authorizations. For example, you might be allowed to see a repository but not to write to it (Guest role). Or some branches might be protected, and only a user with role "Branch Master" can write to these branches.

Interacting with a Remote Git Repository

- Clone from a remote Git repository

If you clone a remote repository, your local Git automatically knows about the branches it contains. By default, it creates the local branch *master* which –at first– corresponds to the remote *master* branch.

- Set a remote (link your local Git repository to a remote one)

In this case, the first thing you need to do is to tell Git to Fetch info from the remote repository. For now, let's say that a Fetch allows the Git client to know about the branches of the remote repository.

In situations where you connect to a brand new remote repository, you should be aware that some Git hosting services allow the repository creator to include an initial commit (an empty one, or one with just a *readme* file) or not. And connecting to an empty repository will generally work better if there is an initial commit.

Now, let's discuss what you might want to do.

Copying Branches

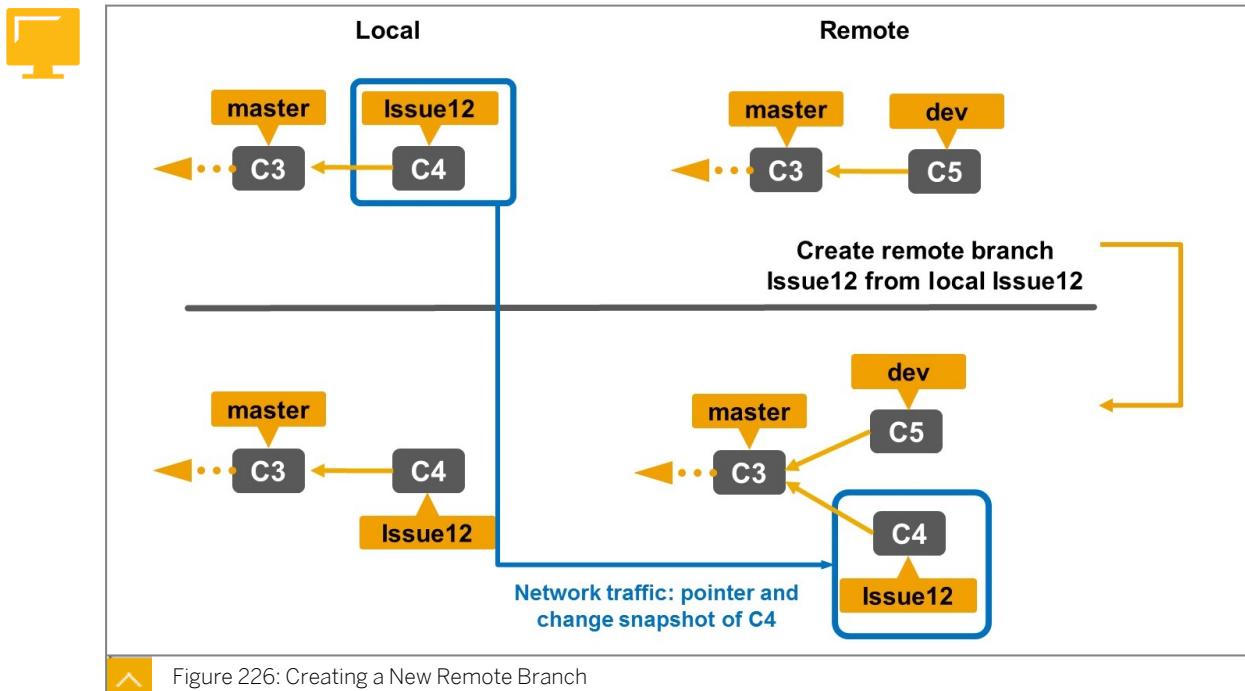
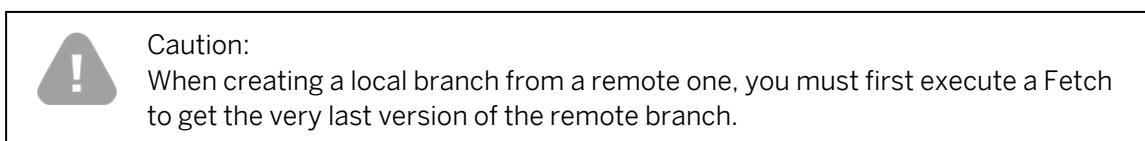
You can create a local or remote branch. In any case, you need to specify a source branch.

Table 24: Creating Branches from Other Branches

Task	Purpose / Example	Steps
Create a local branch from a remote branch	You want to contribute to a project, starting from an existing remote branch you don't have.	<i>Git Pane: Create local branch.</i> Choose the remote branch (source) and define the name of the local branch
Create a remote branch from a local branch	You have created code that you want to share with others developers but without including it in an existing remote branch.	<i>Workspace: Git Context Menu: Create Remote Branch.</i> Choose the local branch (source) and define the name of the new remote branch.

Task	Purpose / Example	Steps
Create a remote branch from a remote branch	You want to keep in a branch V315 the current status of the dev branch, which has just been approved for release	Workspace: Git Context Menu: Create Remote Branch. Choose the source remote branch (dev) and define the name of the new remote branch (V315).

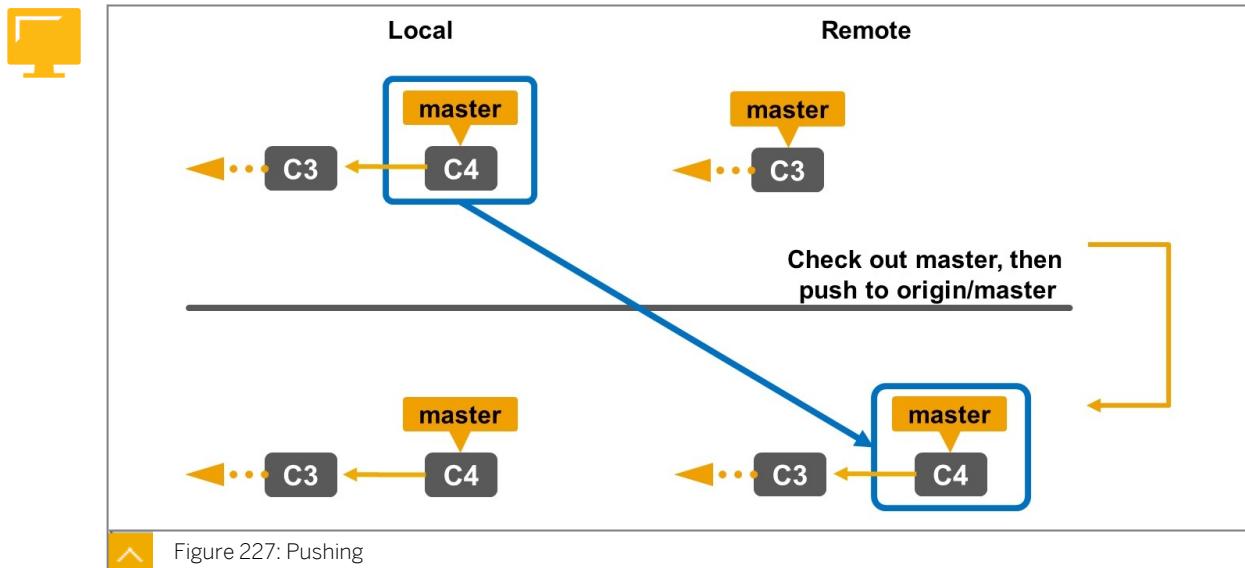
Creating a new branch is a bit like “copying a branch”, but technically, you do not duplicate the entire set of code that the source branch contains. You only create a new pointer, and if needed, Git only transports the changes that correspond to commits that the repository of the new branch does not know about.



In the figure, Creating a New Remote Branch, you have been asked to develop a hotfix for issue #12. You need to develop the fix on top of the productive version (master). When the fix is ready, or even before, if you need to discuss it with a colleague, you can copy the corresponding branch /Issue12 to the remote repository.

Pushing and Fetching

Now, let's discuss Fetch and Push. To keep it simple, let's say that these commands only work between a local repository and a remote one. These are the two essential commands to send local content to a remote repository (push) and get remote content into your local repository (fetch).



Pushing means that you send commits you have in a local branch to a remote one.



Note:

These might be commits of changes that you have made, or commits from another developer that you have just merged into your local branch after validating them.

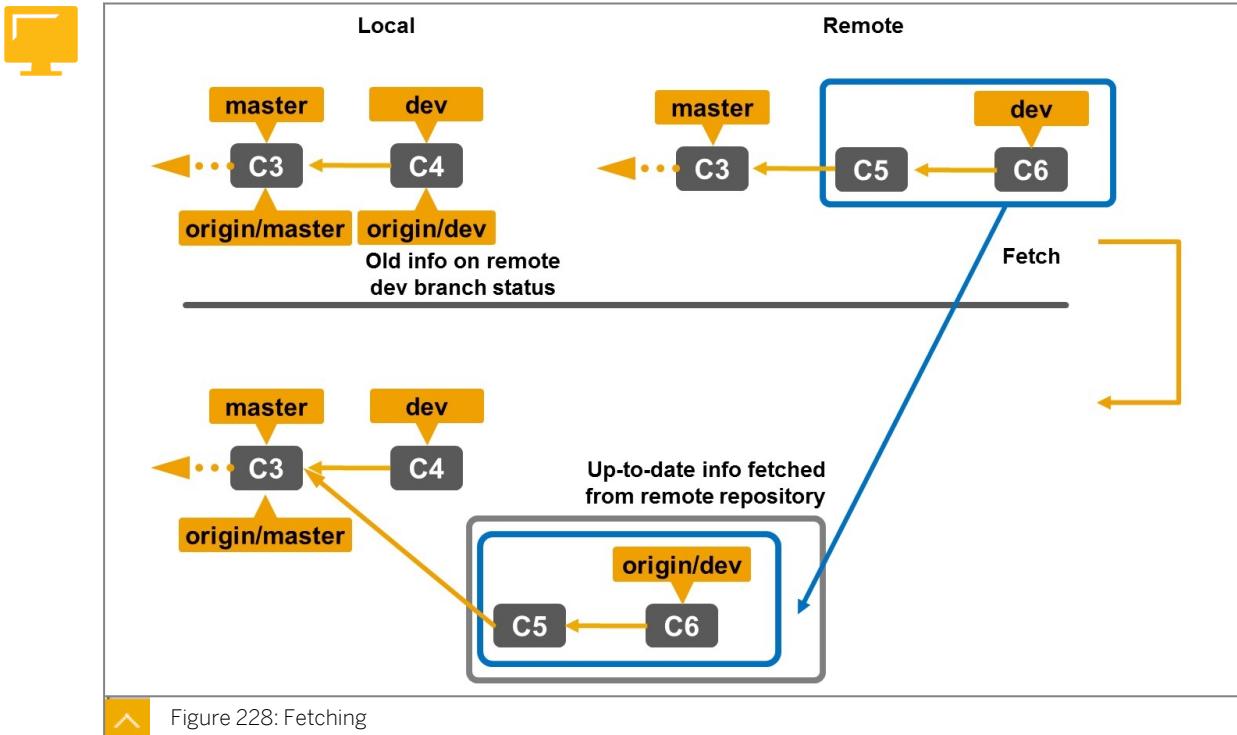
To do that, you check out the branch you want to push changes from, and you define which branch you want to push to.

By default, Git allows you to push only if the last commit of the remote branch is already in your local branch (this allows the equivalent of a fast-forward merge of your change into the remote repository). If not, you will have to Fetch the remote branch history, and then rebase your changes onto the remote branch. Only then, will you be able to perform a successful Push.



Note:

The *Commit and Push* button in the SAP Web IDE triggers first a commit to your current (checked out) local branch, and then a push to a remote branch.



With a Fetch, Git downloads from a remote repository the commits and branch pointers. This does not change your working directory or your local branches, but it creates (or adjusts) locally references and change snapshots from the remote branches. This allows you to inspect the commits that occurred on remote branches (date, description, who committed) and, later on, to merge these changes into a local branch, or rebase a local branch onto the remote branch.



Note:

Fetching is also required if you want to get the list of branches available on a remote repository. In particular, if someone else created a new branch on the remote repository, you cannot push commits to this branch before you know it exists.

In the figure, Fetching, we introduce two additional branch pointers: *origin/dev* and (for the sake of consistency) *origin/master*. These are the local pointers to remote branches. They are used to store locally the state of remote branches when you fetch. Then, any merge, rebase, or new branch creation involving a remote branch is executed locally based on these pointers (and the commits they reference). They materialize what Git calls *Remote-Tracking Branches*. That is, local branches that you cannot commit changes into, but are aimed at keeping locally, offline, a recent status of remote branches.

By default, for a given repository that your local repository knows about, Git will create, (or update) during a fetch, one (local) remote-tracking branch per remote branch. The default naming convention for a remote repository is *origin*, and if a branch in this remote is called for instance *dev*, the corresponding local-tracking branch will be called *origin/dev*. If you want to work on a local copy of this branch, Git will propose the default name *dev* but you can choose another one.

After you fetch, you can see how many commits you are behind of the current remote branch commit. This displays in a box next to the current branch name. For example, you might have

made 2 commits locally after cloning the remote repository (you do not need to fetch to know this), while someone has pushed 3 commits to the same remote (you will need to fetch to get this info). In this case, you would say you are “2 commits ahead and 3 commits behind”.

Pulling

Pulling consists of a Fetch, immediately followed by a Merge. It can be useful in some cases.

Resetting a Branch

In the SAP Web IDE, it is possible to reset a local branch, which means, to revert this branch to the state of another local or remote branch. In case you reset based on a remote branch, always Fetch before resetting, to make sure you get the very last changes of the remote branch in your local repository.

Two reset modes are proposed:

- MIXED (HEAD and index updated)

This reset mode keeps your working directory as is and unstages the changes. Which means, you can stage and commit again all the changes (or part of them) that the reset rolled back.

- HARD (HEAD, index and working directory updated)

With this reset mode, the commits that the reset rolled back are discarded, even in the working directory.



LESSON SUMMARY

You should now be able to:

- Use the Native Git Integration of the SAP Web IDE

Migrating Modeling Content

LESSON OVERVIEW

In this lesson, you will learn about deprecated information model types in SAP HANA Studio and how to convert them into their replacement object types within SAP HANA Studio.

You will also get an overview about the SAP HANA XS Advanced Migration Tool which is used, among others, to migrate SAP HANA Studio-based modeling content (classic repository) to the XS Advanced/HDI infrastructure. The migrated content can then be maintained with the SAP Web IDE for SAP HANA.

Business Example

You work on a project where SAP HANA Studio has been used to create modeling content, including deprecated objects.

As part of a migration project to SAP HANA 2.0 where the SAP Web IDE will be used for modeling, you want to understand the key steps to migrate your existing information views.



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- List the deprecated modeling artifacts
- Explain how to migrate modeling content

Deprecated Modeling Objects in SAP HANA Studio

In the context of SAP HANA Studio modeling, several types of graphical calculation views existed in the past, namely attribute views and analytic views. Historically, calculation views were used only when attribute or analytic views could not fulfill the requirement.

Over time, the successive SAP HANA releases have brought a lot of enhancements to the functional coverage and performance of graphical calculation views. As a result, the use of attribute and analytic views is no longer recommended. All in all, from SAP HANA SPS12 onwards, most use cases should see an equal or better performance of calculation views (even if there can still be a few exceptions).

Besides, it was possible, up to SAP HANA SPS11, to create scripted calculation views. This type of view is now deprecated, and replaced by table functions, which offer better optimization possibilities.

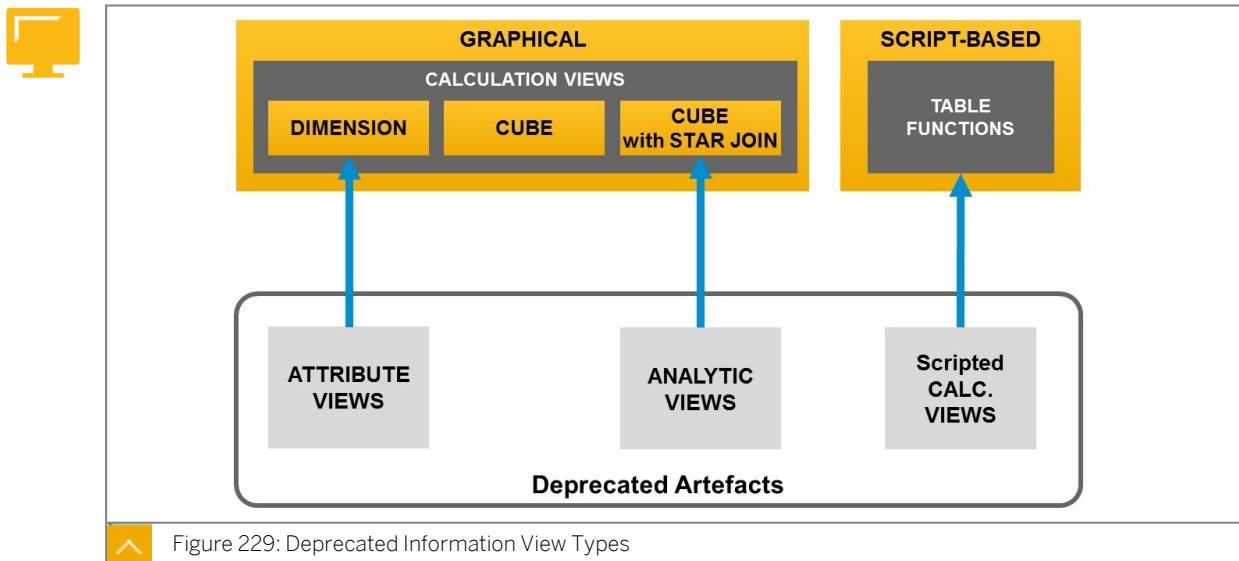


Figure 229: Deprecated Information View Types

The figure, Deprecated Information View Types, shows the deprecated object types and what replaces them.

In addition, to secure the access to the data of information models, it was historically possible to create two different types of Analytic Privileges: XML-based and SQL Analytic Privileges. Now, XML-based (also called “classic”) Analytic Privileges are deprecated.

Migrating Modeling Content

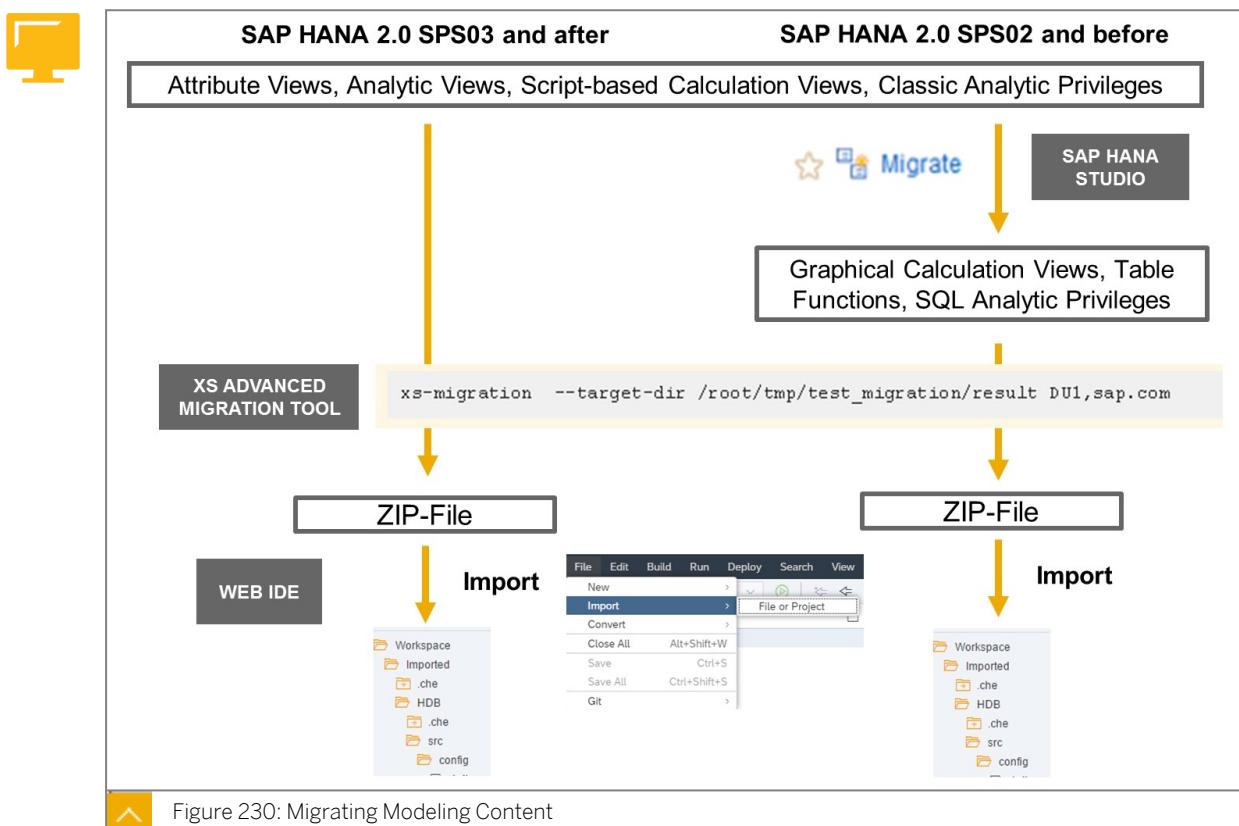


Figure 230: Migrating Modeling Content

A tool to migrate information models to the XS Advanced environment is available. It is called the XS Advanced Migration Assistant. This assistant is invoked from the command line at the

OS level. It generates a template of an XS Advanced project and places the migrated objects in a specific folder. In addition, a migration report is generated and lists the problems that occurred during the migration. They need to be solved before you trigger the migration assistant again.

Once the content has been prepared, the migration assistant generates an MTA archive that you can import into your workspace in the SAP Web IDE for SAP HANA. After importing the project, a number of additional tasks must be performed, within the XSA project, the XSA space, as well as the classical schema objects such as users and roles, in order to finalize security definition, access to external schema, and so on.



Note:

Up to SAP HANA 2.0 SPS02, a step had to be executed before using the XS Advance Migration Assistant. This step is also materialized in the figure, Migrating Modeling Content.

It was executed in the *SAP HANA Modeler* of SAP HANA Studio, and aimed at replacing all the deprecated objects types with more recent ones.

The XS Advanced Migration Assistant has been enhanced in 2.0 SPS03, and now supports the complete migration process.

To learn more about the Modeling Content Migration

- The SAP training HOHAM1 (future code: HA310e), Delta from SAP HANA 1.0 to SAP HANA 2.0 for Modelers, offers a nice opportunity to practice the migration of modeling content from SAP HANA Studio, including preparation in SAP HANA Studio, technical migration, import and post-migration adaptations in the SAP Web IDE for SAP HANA.
- The XS Advanced Migration Assistant has its own guide, since SAP HANA 2.0 SPS01. Additional information can be found in SAP Note [2465027](#)
- The SAP HANA Studio migrate tool is documented in the SAP HANA Modeling Guide for SAP HANA Studio
You can also find additional information in SAP Note [2325817](#), and [2236064](#) in BW contexts.



LESSON SUMMARY

You should now be able to:

- List the deprecated modeling artifacts
- Explain how to migrate modeling content

Learning Assessment

1. Which of the following tools is used to analyze dependencies between models?

Choose the correct answers.

- A Data lineage
- B Column lineage
- C Impact analysis

2. In a full stack application development project using Web IDE XS Advanced, which are provided by an HDB module?

Choose the correct answers.

- A Persistence layer
- B User interface
- C Business logic
- D Data modeling layer

3. An XS Advanced project can contain multiple HDB modules.

Determine whether this statement is true or false.

- True
- False

4. The database objects defined in design-time files located in the same project folder can have different namespaces.

Determine whether this statement is true or false.

- True
- False

5. A HDB module always corresponds to a physical database schema.

Determine whether this statement is true or false.

- True
 False

6. You have successfully built a project that you imported into the SAP Web IDE workspace, but you cannot see the corresponding runtime database objects in the Database Explorer. What could be the reason?

Choose the correct answer.

- A The project is not assigned to a Space.
 B Building a project does not generate the database objects defined in its HDB modules.
 C You do not have the *Developer* role in the assigned Space.
7. A calculation view CV1 has been created in your project folder but has never been built. You perform a build of another calculation view CV2 that consumes CV1. The CV1 file will be included in the build of CV2.

Determine whether this statement is true or false.

- True
 False

8. Which database artifact do you use to access external data that is outside your project?

Choose the correct answer.

- A Calculation view
 B Synonym
 C Logical schema
 D User-provided service

9. The Git architecture relies on a central server that stores a unique version of your source code and includes a check-out process that prevents several users from modifying the same source file at the same time.

Determine whether this statement is true or false.

- True
 False

Learning Assessment - Answers

1. Which of the following tools is used to analyze dependencies between models?

Choose the correct answers.

- A Data lineage
- B Column lineage
- C Impact analysis

Correct — Data Lineage and Impact Analysis are tools used to analyze the dependencies between models, whereas Column Lineage only shows the origin of a column within a single model.

2. In a full stack application development project using Web IDE XS Advanced, which are provided by an HDB module?

Choose the correct answers.

- A Persistence layer
- B User interface
- C Business logic
- D Data modeling layer

Correct — The persistence layer and data modeling layer are defined in an HDB module. The user interface typically uses HTML5 or UI5 modules, and the business logic uses Java or Node.js modules.

3. An XS Advanced project can contain multiple HDB modules.

Determine whether this statement is true or false.

- True
- False

Correct — A project can contain more than one HDB module.

4. The database objects defined in design-time files located in the same project folder can have different namespaces.

Determine whether this statement is true or false.

True

False

Correct — The rules defining namespaces are defined at the folder level (at any level of the folder structure). This means all database objects created within a folder always inherit the same namespace provided by the folder.

5. A HDB module always corresponds to a physical database schema.

Determine whether this statement is true or false.

True

False

Correct — When an HDB module is built for the first time, a schema is created in the SAP HANA database catalog and managed by the corresponding HDI container service.

6. You have successfully built a project that you imported into the SAP Web IDE workspace, but you cannot see the corresponding runtime database objects in the Database Explorer. What could be the reason?

Choose the correct answer.

A The project is not assigned to a Space.

B Building a project does not generate the database objects defined in its HDB modules.

C You do not have the *Developer* role in the assigned Space.

Correct — Building a project generates an MTA archive for deployment, but does NOT trigger a build of the HDB module(s). If the project were not assigned to a space, the build would fail. And you can only assign a project to a space where you have the *Developer* role.

7. A calculation view CV1 has been created in your project folder but has never been built. You perform a build of another calculation view CV2 that consumes CV1. The CV1 file will be included in the build of CV2.

Determine whether this statement is true or false.

True

False

Correct — A design-time file that has never been built is ignored by the dependency check performed by the HDI builder if it is not part of the build scope.

8. Which database artifact do you use to access external data that is outside your project?

Choose the correct answer.

- A Calculation view
- B Synonym
- C Logical schema
- D User-provided service

Correct — Calculation views can consume external data, but they need a synonym to access this data. A logical schema can be used to define synonyms more dynamically, but this is something defined in XS Advanced, it is not a database artifact. As for the user-provided service, it is required to access data from an external schema but it is not a database artifact.

9. The Git architecture relies on a central server that stores a unique version of your source code and includes a check-out process that prevents several users from modifying the same source file at the same time.

Determine whether this statement is true or false.

- True
- False

Correct. The Git architecture is not centralized but distributed, which means that several contributors to a project can have the complete project code on their own computer. There is no check-in / check-out process, compared with other types of control-version systems.

UNIT 8

Security in SAP HANA Modeling

Lesson 1

Understanding Roles and Privileges

361

Lesson 2

Defining Analytic Privileges

373

Lesson 3

Defining Roles

387

Lesson 4

Masking Sensitive Data

397

Lesson 5

Anonymizing Data

401

UNIT OBJECTIVES

- Understand roles and privileges
- Define analytic privileges
- Create a design-time role
- Restrict access to columns containing sensitive data within a View
- Protect sensitive data with anonymization

Understanding Roles and Privileges

LESSON OVERVIEW

This lesson will give you an overview of how security is implemented in SAP HANA.

Business Example

After the SAP HANA System has been installed, you need to define security and give the relevant authorizations to users so that they can administrate the system, provision data, and, in the case of the Modeler role, start creating models.

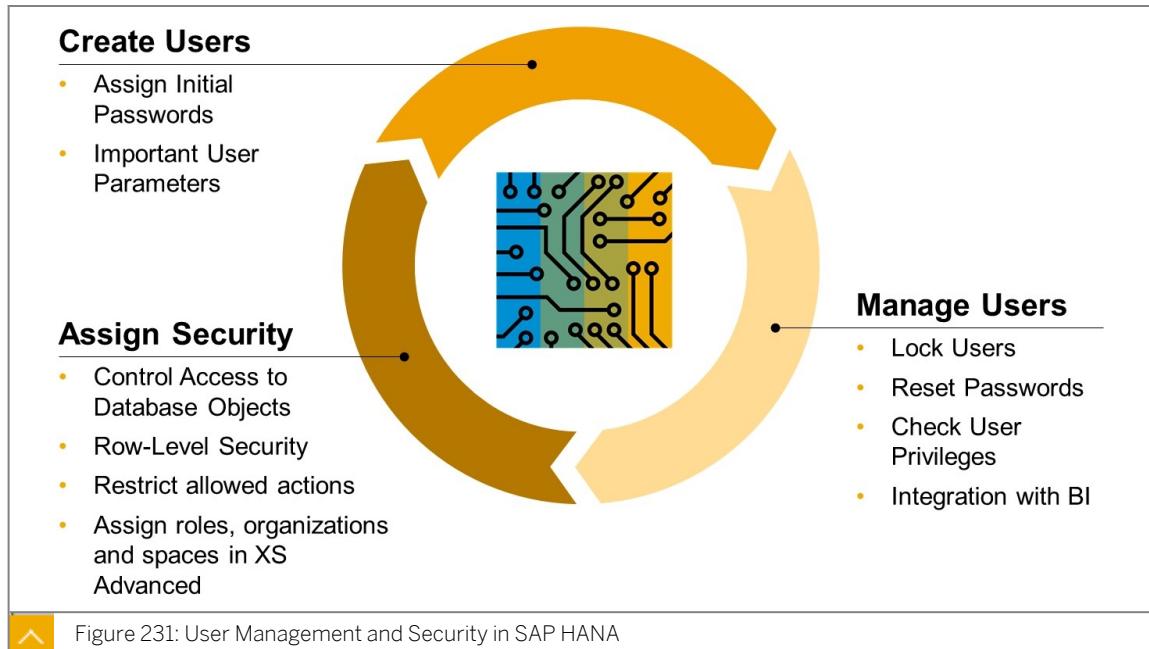


LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Understand roles and privileges

Overview of User Management and Security



Like any other database system, SAP HANA provides features to define and maintain security.

The reasons for implementing security are as follows:

- Restrict database administration to skilled and empowered employees only
- Ensure that each database user has relevant authorization to manipulate database objects
- Separate duties

- Manage access to the system from a variety of front-end tools
- Restrict access to the data stored in the system based on role, geographic, or organizational responsibilities

XS Advanced and SAP HANA Database Security

When you work in SAP HANA with the SAP Web IDE (so, in XS Advanced), there are two aspects of security that you must consider.

- XS Advanced Security

XS Advanced provides its own runtime, which is separate (though heavily connected) to the database. The XS Advanced environment has its own way to implement security. This security is defined in the XS Advanced Administration Tools. For example, when you log on to the Web IDE, a dedicated XS Advanced service checks whether you have the role *Web IDE developer*, and also which space(s) – for example the *DEV* space used for this training – you are allowed to develop in.

- SAP HANA Database Security

The database security is what governs general authorizations on the database system, access to objects (tables, views, procedures), fine-grained access to data with analytic privileges, and so on.

Most of the security artifacts that you build in an HDB module are classic HDB security objects, such as roles, analytic privileges, and so on. Once the roles are built, you can assign them to classic database users.



Note:

This is different when you work in SAP HANA Studio and/or XS Classic, because in this case all the security is relying on classical security artifacts: system privileges, object privileges (schemas and/or objects they contained), analytic privileges, and also, for example, package privileges (for SAP HANA Studio-based modeling) and application privileges (for XS Classic development).

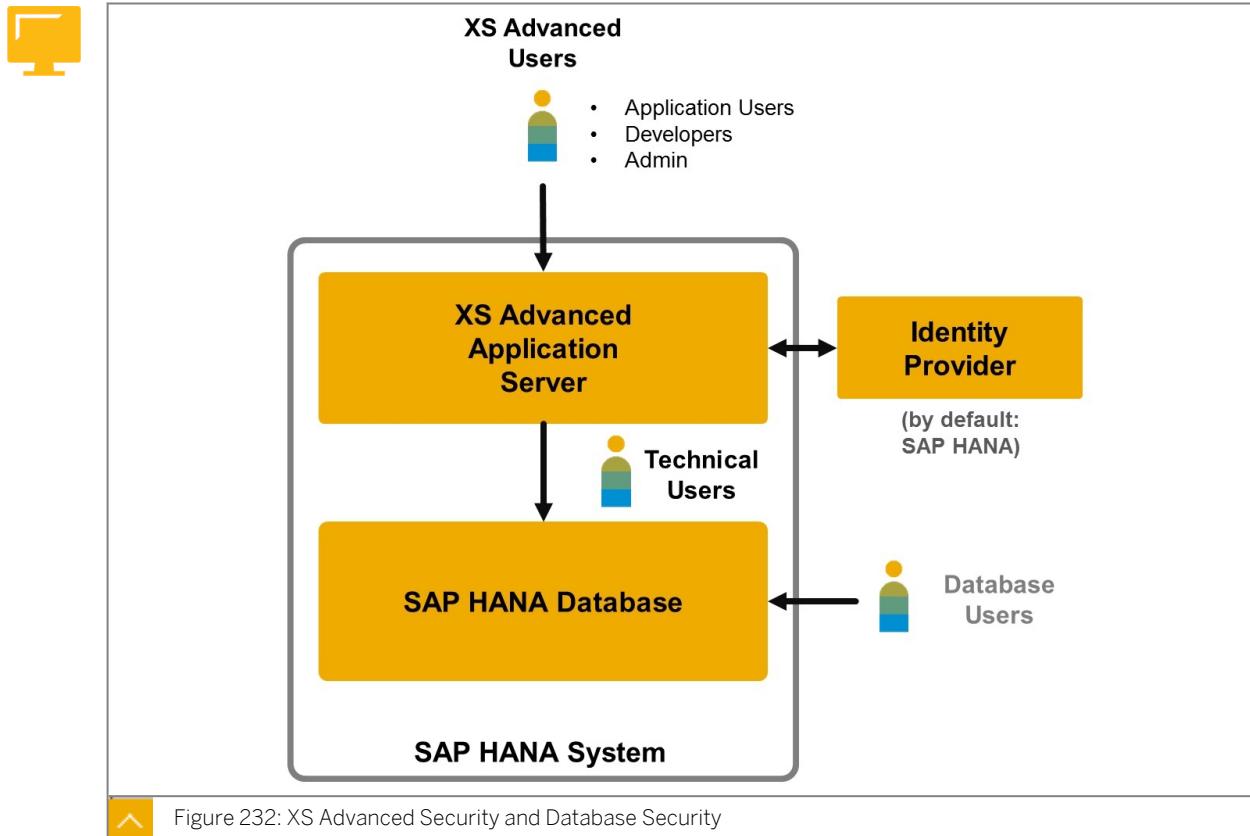


Figure 232: XS Advanced Security and Database Security

The interaction between the XS Advanced Application Server and the SAP HANA database involves a number of technical users. For example, each and every HDI container deployed in the database comes with technical users to manage the ownership of the corresponding schema and the database objects (tables, column views) as well as the key security artifacts such as analytic privileges, roles, and so on.

When you work on a project in the SAP Web IDE for SAP HANA, almost everything you do, for example, access tables from your container or from an external schema, build a calculation view, preview its data, and so on, is done on behalf of your Web IDE user by a technical user that is specific to your container, which is granted the necessary authorizations to the container's schema. In the Database Explorer, viewing the objects of a container's schema is done by this technical user.

On the contrary, when you add a classic database to the Database Explorer and specify a user and password, the queries you execute on this database are executed by this user. In this case, no technical user is involved.

XS Advanced Users

The XS Advanced (XSA) users are listed in the SAP HANA database like classic database users. The main difference is that additional user properties, specific to XSA user management, are used to define the security for these users.

When you create a user in the XSA environment, this is done in the *XS Advanced Administration Tool*. There are two different possibilities:

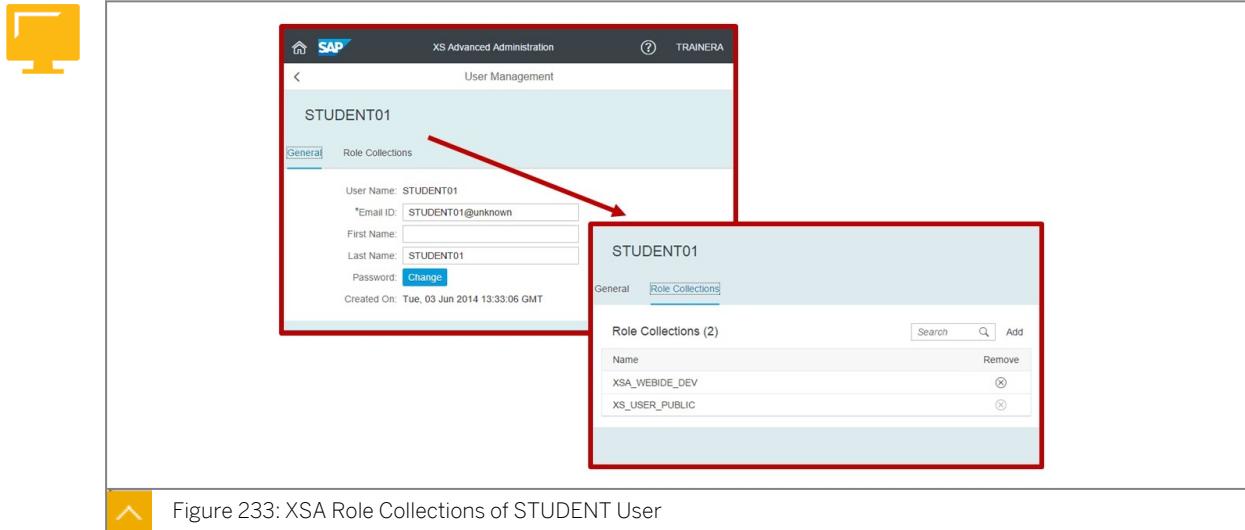
- Create a new user.
With this approach, you create a brand new user.
- Reuse an existing database user for XSA.

In this case, you choose an existing database user and enable this user for XSA.

When a user is enabled for XSA, you assign this user a role collection, which contains one or several XSA application roles.

Let's take the following examples:

- Your *STUDENT##* user is assigned the role collection *XSA_WEBIDE_DEV* that allows this user to access the Web IDE and create applications.
- The *XSA_ADMIN* user (super-admin user in the XSA environment) is assigned other role collections to manage users, and so on.

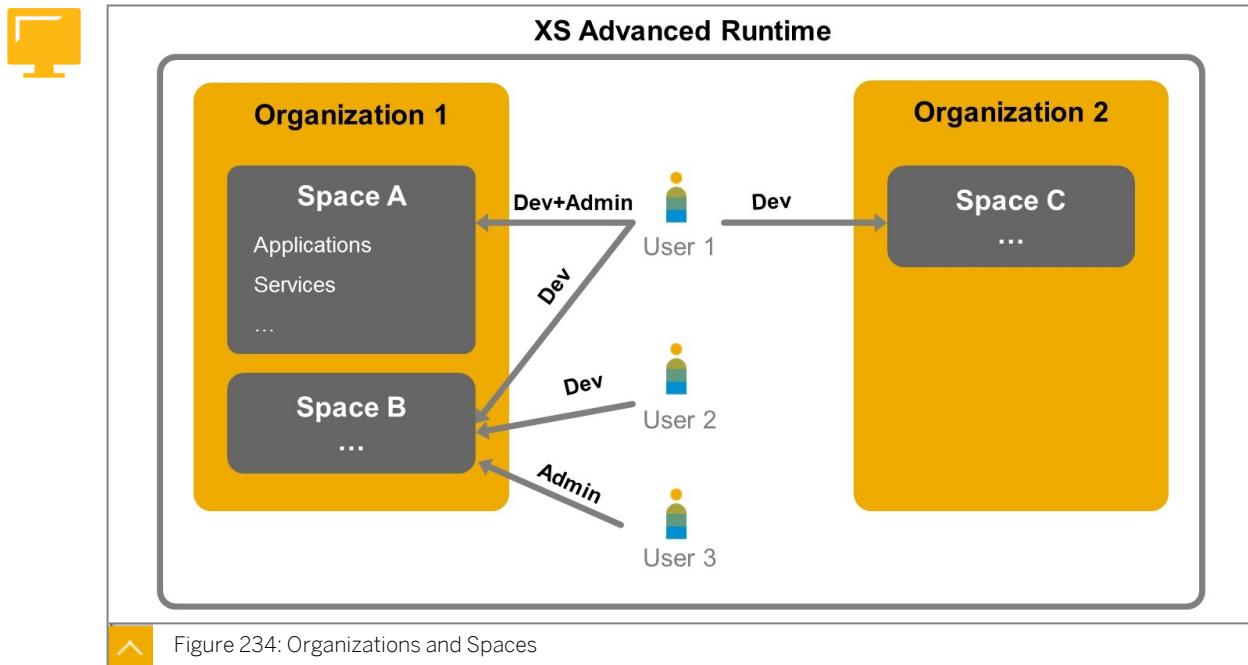


Note:

During Exercise 1 of this course, you also granted your *STUDENT##* user specific authorizations to your container. This is what allows your user to directly consume the container's tables and column views with front-end tools such as MS Excel or SAP BusinessObjects Analysis for Excel.

Organizations and Spaces

In addition to the role collections, the security in XSA relies on organizations and spaces.



- **Organizations**

Organizations enable developers to collaborate by sharing resources, services, and applications. Access to the shared resources, services, and applications is controlled by roles, for example, *Org Manager* or *Org Auditor*; the role defines the scope of the permissions assigned to the named user in the organization.

For example, an *Org Manager* can add new users to organizations; create, modify, or delete organizational spaces; and add domains to the organization.

- **Spaces**

In an organization, spaces enable users to access shared resources that can be used to develop, deploy, and maintain applications.

Access to the resources is controlled by the following roles: *Space Manager*, *Space Developer*, and *Space Auditor*. The role defines the scope of the permissions assigned to the named user in the organizational space. For example, a *Space Developer* can deploy and start an application.

Organization and Space of STUDENT User

The figure, Organization and Space of STUDENT User, shows that the STUDENT## users are assigned as Developers in the DEV space of the SAP organization.

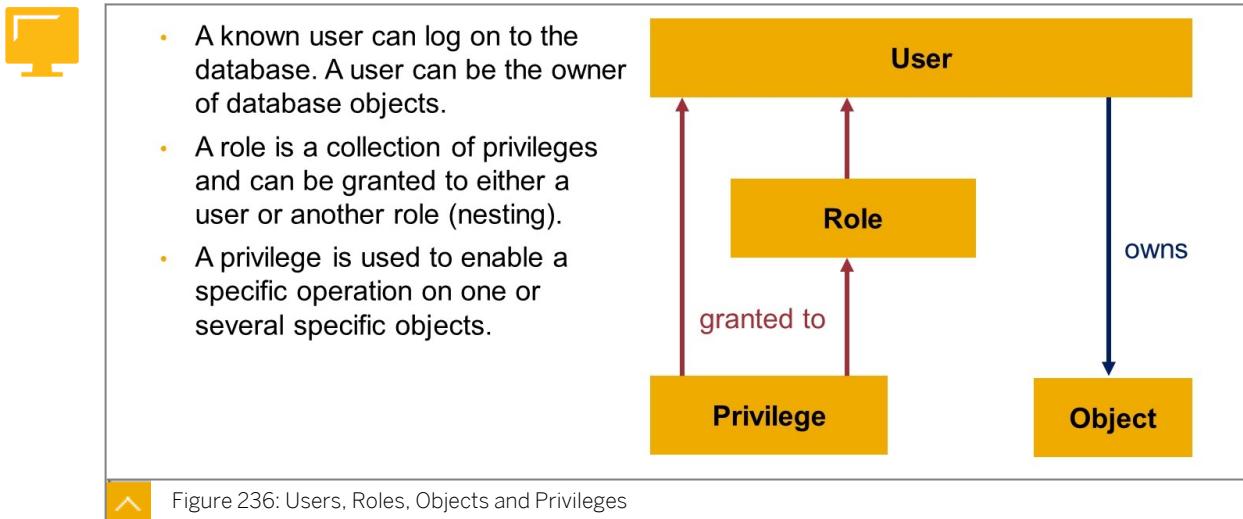
In XSA, a user can be assigned to several spaces, with different roles, in one or several organizations.

The SAP Space

The SAP space is created by default when you install XS Advanced, and this is where most of the applications and services for XSA Developers and Administrators are located. For example, the SAP Web IDE, the XS Advanced Administration Tools, and many others, such as the back-end services used by the Database Explorer, the User Authentication and Authorization services, and the HANA Deployment Infrastructure (HDI) applications used to build your modeling content.

Caution:
For security reasons, it is highly recommended to keep all these applications and services separate from the applications you create and to strictly restrict the authorizations granted to the SAP space.

Database Security: Users, Roles and Privileges



The figure, Users, Roles, Objects, and Privileges, shows how users and roles can be granted privileges.

- Privileges can be assigned to users either directly, or indirectly by using roles.
- Roles help you to structure the access control scheme and model reusable business roles. They can be nested, enabling the implementation of a hierarchy of roles.



Hint:

It is highly recommended that you manage authorizations for users by using roles. Assigning a privilege directly to a user is not a good practice.

- All the privileges granted directly or indirectly to a user are combined.
Whenever a user tries to access an object, the system performs an authorization check based on the user's roles and directly allocated privileges (if any).
- It is not possible to explicitly deny privileges.
In particular, the system does not need to check all the users roles. As soon as all the privileges required for a specific operation on a specific object have been found, the system ends the check and allows the operation.
- Several predefined roles exist in the SAP HANA database.
Some of them are templates (and need to be customized), and others can be used as they are.
- As a best practice, users should only be given the smallest set of privileges required for their role.

Defining Roles

In the SAP HANA database, there are two ways to create roles:

- As pure runtime objects that follow classic SQL principles (**Catalog Roles**)

- By means of design-time files that you create in the HDB module of a project (**Design-Time Roles**)



Table 25: Catalog Roles vs. Design-Time Roles

Feature	Catalog Roles	Design-Time Roles
Transportability	No	Yes
Version Management	No	Yes
Relationship to database users	Each role is owned by the database user that created it.	The roles created when building the HDB Module are owned by the technical user that owns the container's content.

In general, repository roles are recommended, as they offer more flexibility. In particular, they can easily be transported between SAP HANA systems via the build/deployment of XSA Applications.

Design-time roles are created as *.hdbrole* files within a project, with the SAP Web IDE.



Note:

A typical use case where catalog roles can be used is when security is managed in another tool, and the SAP HANA database is only accessed by a very limited number of technical users. But in the context of XSA-based modeling, design-time roles are definitely recommended, so the authorizations stay in sync with the objects they refer to.

How to Assign Privileges

The following list outlines the ways to maintain users and roles in SAP HANA:

- In SAP HANA Studio, by using the dedicated folder *Security* of the *Systems* view.
- From an SQL Console, in SAP HANA Studio or the Database Explorer of the SAP Web IDE, by executing SQL statements.

Figure 237: Assigning Roles by SQL or HANA Studio UI

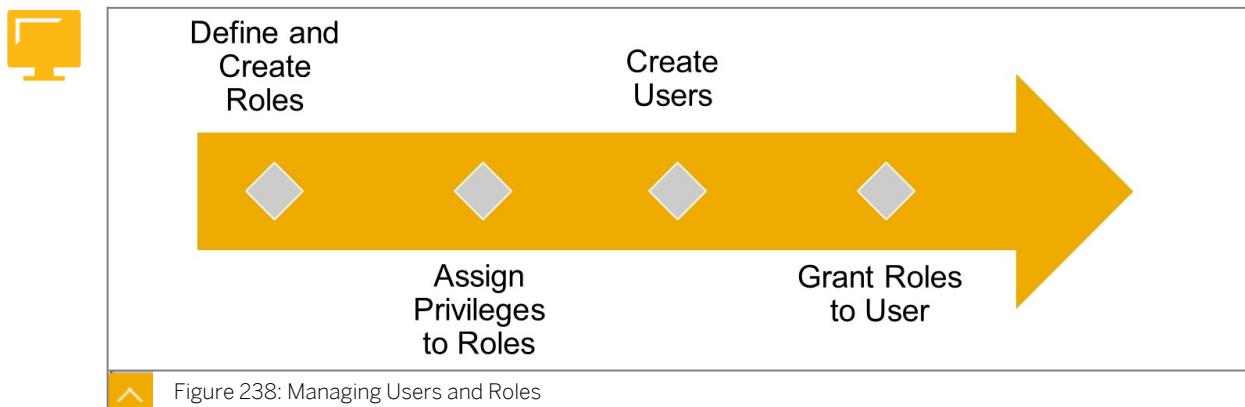
The figure, *Assigning Roles by SQL or HANA Studio UI*, shows how to grant the generic access role of a container to your *TRAINING_ROLE_##*. This is what you did as part of the first exercise.



Note:

This generic access role is also granted to the technical user who works “on your behalf” when you are logged on to the Web IDE and working on an XSA project.

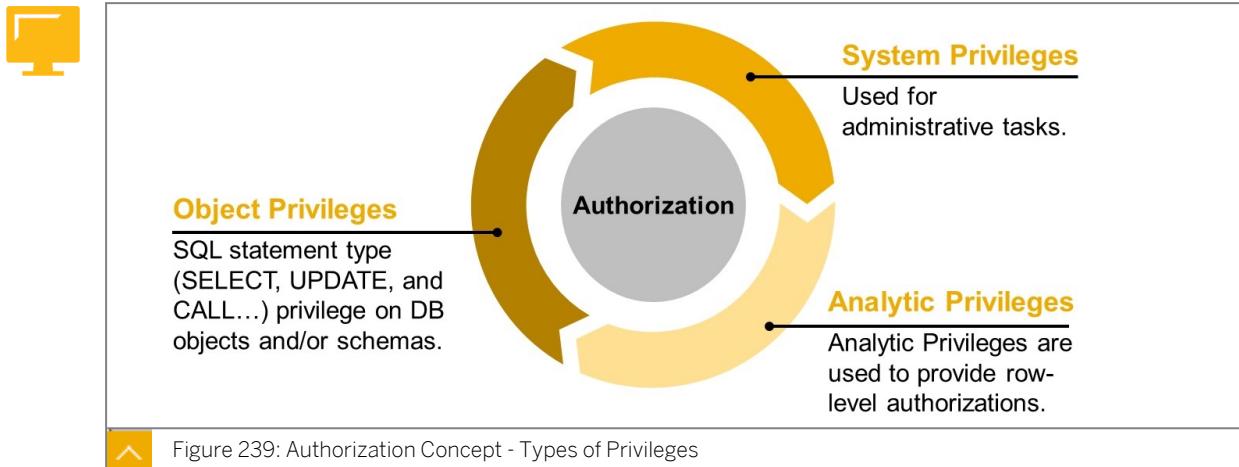
Managing Users and Roles



The figure, *Managing Users and Roles*, shows the general process to define users and roles and assign privileges. In a typical SAP HANA modeling scenario with the Web IDE, defining the roles and assigning privileges (or other roles) to roles is done in design-time files.

Types of Privileges

Let's have a quick overview of the different types of privileges that can be defined in the database.



When modeling or developing applications in XSA, two types of privileges are not used at all:

- Package privileges
- Application privileges

System Privileges

System privileges are used to control general system activities. They include general administrative actions, such as creating or deleting schemas, managing users and roles, performing backup, monitoring and tracing, and so on.

System privileges also include several privileges directly related to the modeling activities in SAP HANA Studio (not in the SAP Web IDE), such as maintaining, exporting, and importing delivery units.

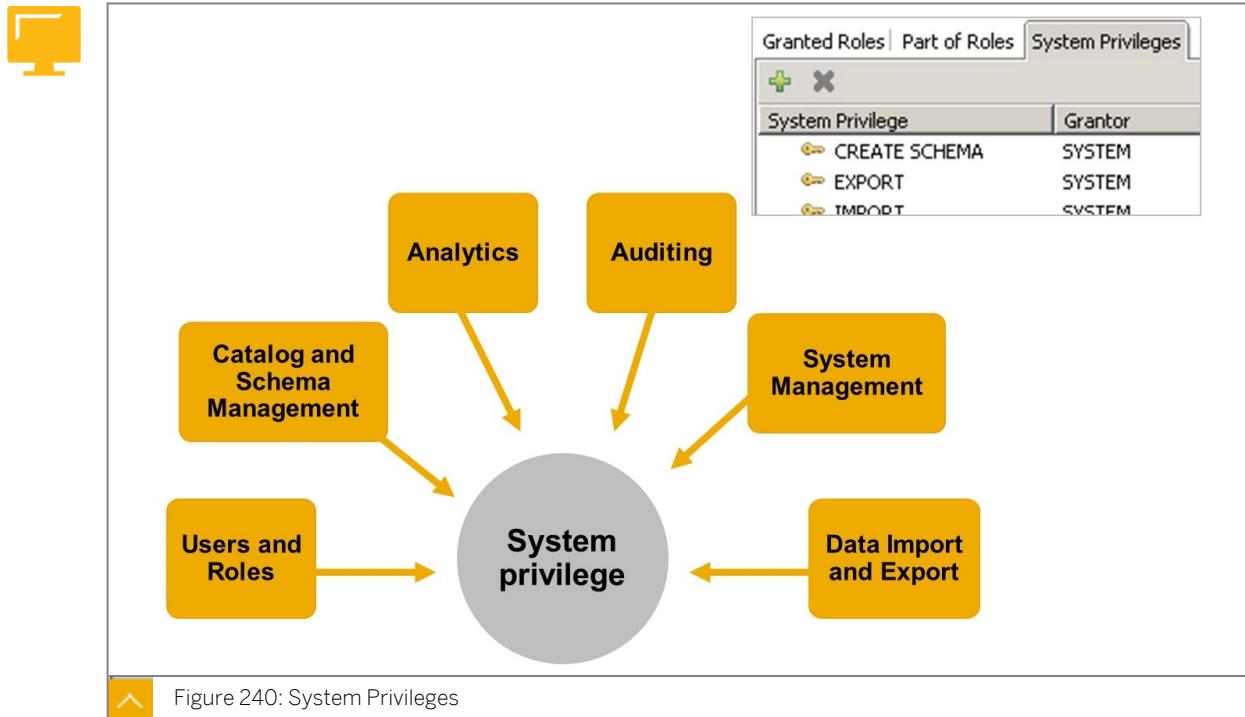


Figure 240: System Privileges

Object Privileges

Object privileges are used to allow access to and modification of database objects.

Each object privilege is related to one object and, depending of the type of object (schema, table, and procedure) includes the different SQL statement types, which you can grant separately; for example, CREATE, ALTER, DROP, SELECT, INSERT, UPDATE, DELETE, and EXECUTE.



Note:

Most of the object privileges provide an option defining whether the grantee (user or role) has the right to grant it to other users.

Object Privileges for Modular Role

For the modeler role, object privileges are a key building block to define security, in particular because they control the way users read and modify data, both in the Container of a deployed application, but also in external schema or other containers.



- Object privileges are bound to an object, for example, to a database table or schema, and enable object-specific control activities, such as SELECT, UPDATE, or DELETE.

The screenshot shows a table with columns "SQL Object" and "Grantor".

SQL Object	Grantor
_SYS_BI	SYSTEM
_SYS_BIC	SYSTEM

The screenshot shows a table with columns "Grantable to Others" and checkboxes for various SQL statements.

Grantable to Others	
<input type="checkbox"/> CREATE ANY	<input type="radio"/> Yes <input checked="" type="radio"/> No
<input checked="" type="checkbox"/> ALTER	<input type="radio"/> Yes <input checked="" type="radio"/> No
<input type="checkbox"/> DROP	<input type="radio"/> Yes <input checked="" type="radio"/> No
<input checked="" type="checkbox"/> EXECUTE	<input type="radio"/> Yes <input checked="" type="radio"/> No
<input checked="" type="checkbox"/> SELECT	<input type="radio"/> Yes <input checked="" type="radio"/> No
<input type="checkbox"/> INSERT	<input type="radio"/> Yes <input checked="" type="radio"/> No

Figure 241: Object Privilege

Analytic Privileges

Analytic privileges are used to enable data access to different portions of data in calculation views, by filtering the attribute values.

Each information view has a dedicated property indicating whether the execution of the view must be restricted by analytic privileges or not. You will learn more about this distinction later.



Note:

An analytic privilege is evaluated when a user executes a query against one of the information views to which the privilege applies.

Related Information

You can find more information about security in the following guide, available on the SAP Help Portal. Go to <http://help.sap.com/hana>:

- SAP HANA Security Guide

You can also refer to the training course **HA240 - SAP HANA Security & Authorization**. Go to <http://training.sap.com> for more details.



LESSON SUMMARY

You should now be able to:

- Understand roles and privileges

Defining Analytic Privileges

LESSON OVERVIEW

This lesson will describe how to create Analytic Privileges in the SAP Web IDE for SAP HANA, in order to control the access to data based on attributes. Note that two different flavors of Analytic Privileges existed in SAP HANA Studio modeling, but in XSA-based modeling with the SAP Web IDE, only SQL Analytic Privileges are supported. The focus will be put on this type of Analytic Privileges.

Business Example

You work as a Modeler on an SAP HANA Project, and you have been asked to design the data access security. You need to learn about how to define and assign Analytic Privileges.

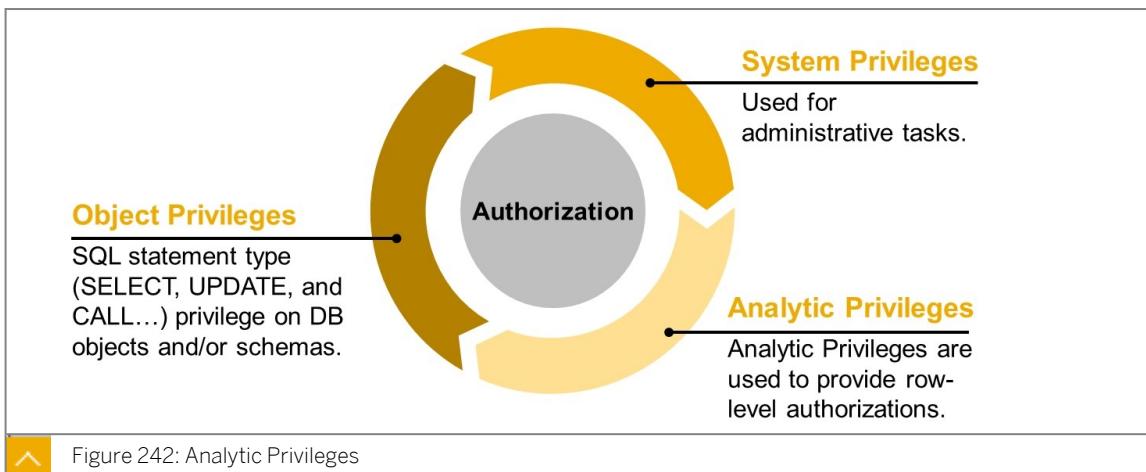


LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Define analytic privileges

Analytic Privileges



Analytic privileges are used to enable data access to different portions of data in calculation views, by filtering the data based on the values of one or more attributes.

The rationale for analytic privileges is to allow the use of calculation views by different users who might not be allowed to see the same data.

For example, different regional sales managers who are only allowed to see sales data for their regions could use the same calculation view, but you would have to define and assign analytic privileges to the view so that each manager sees only the data for their region.

**Note:**

This is different from the behavior of SAP Business Warehouse (BW). While the concept is similar, SAP BW will forward an error message if you execute a query that returns values you are not authorized to see.

With SAP HANA, the query would be executed and, depending on your authorizations, only values that you can see would be returned.

Types of Analytic Privileges

To avoid any confusion, let's explain that there were historically two different types of analytic privileges

Two Flavors of Analytic Privileges



- SQL analytic privileges

This type of analytic privilege provides the highest flexibility in scenarios where filter conditions are complex. In addition, it can be used on a large variety of objects, including calculation views, CDS views, and catalog views.

- Classical analytic privileges (also called XML-based analytic privileges)

This is the historical data access restriction type, and it is NOT supported in XSA-based modeling (in SAP Web IDE).

**Note:**

A migration utility is available in SAP HANA Studio to convert classical analytic privileges to SQL analytic privileges.

In the following sections, we only cover SQL analytic privileges.

SQL Analytic Privileges — The End-to-End Scenario

To secure a calculation view with SQL analytic privilege, the main steps are as follows:

To Create and Assign an Analytic Privilege



1. Start the analytic privilege creation wizard.
2. Assign the calculation view(s) that you want to secure with this analytic privilege.
3. Choose the type of restrictions you want to use and define the restrictions.
4. Set the secured calculation views to check SQL analytic privileges.
5. Save and build the analytic privilege.
6. Assign the analytic privilege to a role.
7. Assign the role to a user.

1. Start Creation Wizard

- Select a folder in your project.

- Choose New → Analytic Privilege.
- Provide a name and description and choose Create.

2. Select Information Models

An analytic privilege definition contains the list of information models to which it will apply. You can choose one or several of the following objects:

- Calculation views of type DIMENSION and CUBE (with or without a star join)
- CDS views
- Catalog views



Note:

Defining the list of secured information models is a prerequisite if you want to create restrictions based on attributes explicitly chosen among the actual attributes defined in these models. You will learn more about restriction types later.

The analytic privilege editor lists, at any time, all the models that are secured by the analytic privilege.

Analytic Privilege - Select Information Models



Select applicable Information Models

The views selected to define an Analytic Privilege determine:

- Which views the Analytic Privilege will apply to
- What attributes you can choose to define restrictions

You can add other views to an existing Analytic Privilege at any time

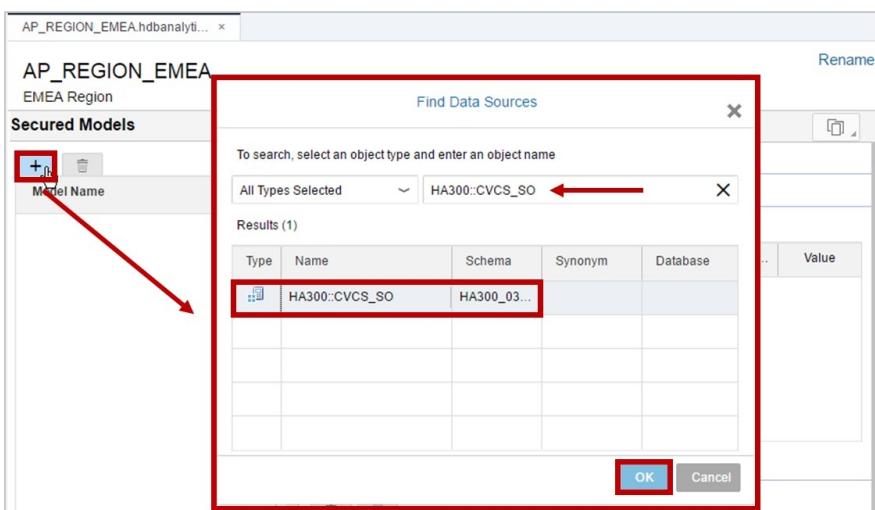


Figure 243: Analytic Privilege - Select Information Models

3. Define the Restrictions

There are three methods available to define the restrictions in an analytic privilege. These methods are exclusive; that is, in one analytic privilege, you use only one of these methods.



Table 26: Restriction Types in Analytic Privileges

Restriction type	How to Use	Restriction Example
Attribute	With the restriction editor, select one or several attributes from the secured views. For each of them, define restrictions.	<ul style="list-style-type: none"> • REGION: EMEA • YEAR: Between 2015 and 2017
SQL Expression	Create a valid static SQL expression that refers to the attributes and the authorized values. This is useful when the <i>Attribute</i> restriction type does not fulfill the requirement.	(“REGION”=’EMEA’ AND “YEAR”=’2015’) (valid SQL expression)
Dynamic	Use a procedure to derive a dynamic SQL expression to restrict the data set. This expression must be similar to a WHERE clause in a select statement.	P_DYNAMIC_AP_FOR_REGION (name of the procedure)

You can change between restriction types, depending on your needs.

Changing Restriction Type



Figure 244: Changing Restriction Type

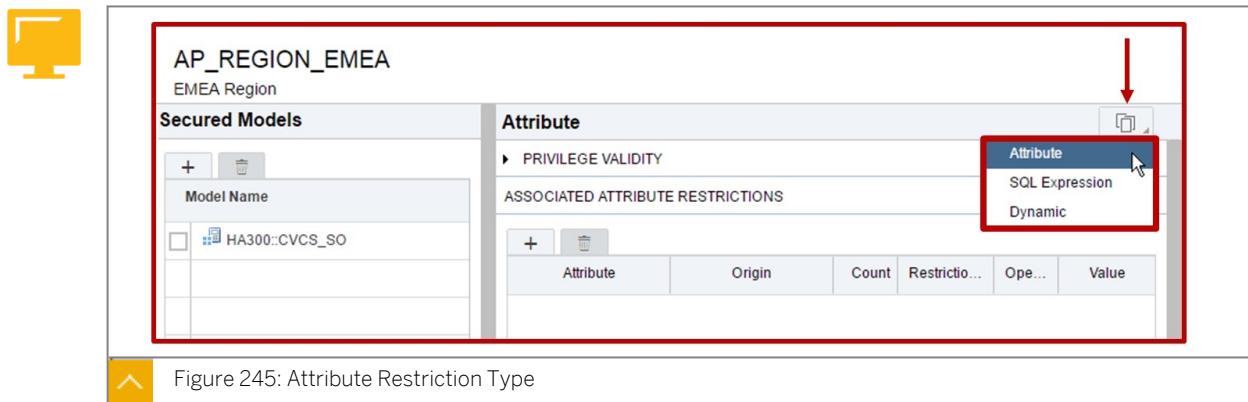


Note:

A restriction of type *Attribute* can be automatically converted into an *SQL Expression* restriction, but the other way round is NOT possible.

To define a restriction of type *Attribute*, you can select among the list of shared and private attributes from the secured calculation views. You can define as many restrictions as needed to define the correct data set.

Defining a Restriction of Type Attribute



Defining Values in the Restriction Filter

To define values in the restriction filter, you can use the following operators:

- **Between <scalar_value_1> <scalar_value_2>**
- **ContainsPattern <pattern with *>**
- Comparison operators: =, <=, <, >, >= with **<scalar value>**
- **IsNull** and **Is Not Null**

Note:

All filter operators, except **IsNull** and **Is Not Null**, accept empty strings (" ") as filter operands. For example:

- *In (" ", "A", "B")*
- *Between (" ", "XYZ")* (as lower limit in comparison operators)

Only columns of type *Attribute* (NOT *Measure*) can be specified in dimension restrictions.

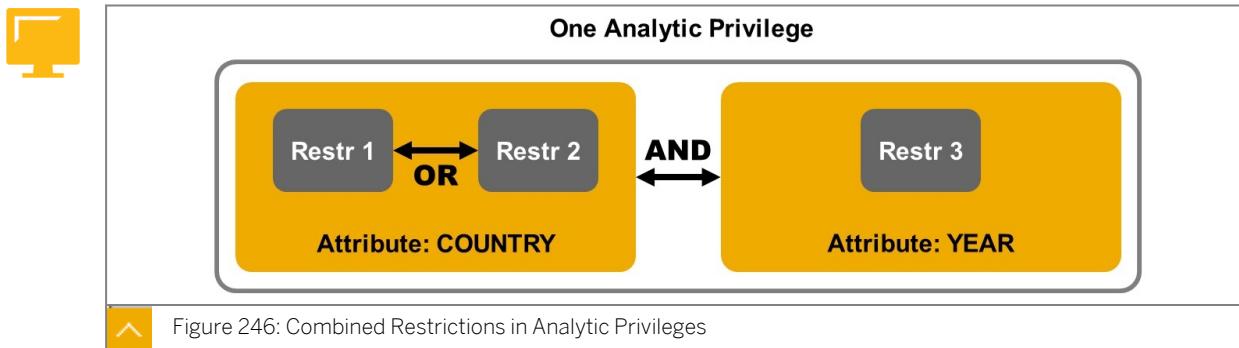
If a DIMENSION calculation view contains a parent-child hierarchy and the hierarchy is enabled for SQL access, it is also possible to define the restriction on a hierarchy node. This is discussed in a dedicated section later.

Combining Several Attribute Restrictions

Several restriction filters within an analytic privilege are combined in the following way:

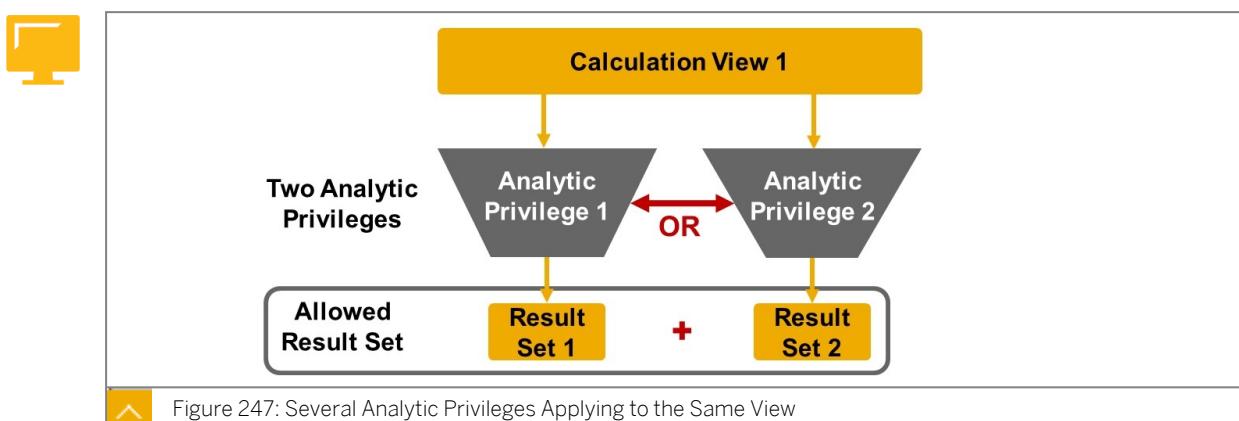
- Within one attribute column, several restrictions are combined with a logical OR.
- Within one analytic privilege, all dimension restrictions are combined with a logical AND.

For example, if an analytic privilege includes two restrictions on two different attributes (YEAR=2017, COUNTRY=US), the user is allowed to see only data fulfilling the compound condition YEAR=2017 AND COUNTRY=US.



Several Analytic Privileges Applying to the Same View

If two analytic privileges (or more) are defined to apply to the same view, SAP HANA combines the corresponding conditions with a logical “OR”.



Caution:

If the two restrictions from the previous example were defined in two different analytic privileges applying to this user and this view, the user would see more data. Namely all the rows for which `YEAR=2017 OR COUNTRY=US` (that is, any year for `COUNTRY=US` and any country for `YEAR=2017`).

Defining a Validity Period

You can decide to make an analytic privilege of the type *Attribute* valid for only a certain period of time. This restriction applies to the date when users are querying the secured view (`CURRENT_DATE`).



Caution:

This is not related to the time attributes (year, month, date, and so on) defined in your views. To limit the access to the data based on such attributes, you use a classic attribute-based restriction, for example on a `DATE` or `YEAR` column.

Restricting Value with an SQL Expression

This second type of restriction allows you to define a filtering expression that cannot be obtained with restrictions of type *Attribute*. For example, when the precedence of OR and AND logical operators does not correspond to what you want to define.

Suppose you want to allow a user to view all the data from the country he's responsible for (US), but also all the historical data for any country. You could create a restriction of type *SQL Expression* as follows:

```
("COUNTRY" = 'US' OR "YEAR" <= '2016')
```



Hint:

Instead of writing your SQL expression from a blank page, you can start defining a restriction of the type *Attribute*, and then convert the restriction type to *SQL expression*. The corresponding SQL code will be already generated for you, and you will just need to adjust it to your requirements.

Defining a Validity Period

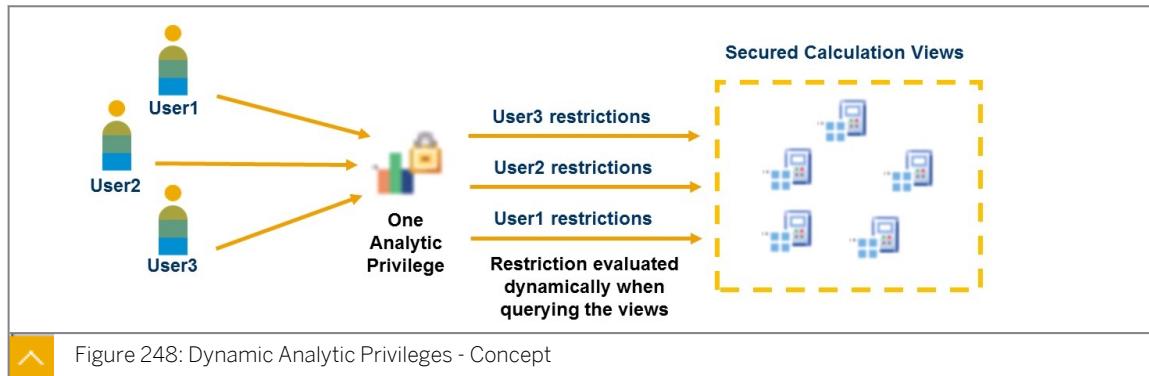
Restrictions of the type *SQL Expression* also support a validity period, by adding to the SQL expression the relevant SQL code to filter the *CURRENT_DATE*.

For example:

```
(CURRENT_DATE BETWEEN '2017-07-28 00:00:00.000' AND '2017-07-28  
23:59:59.999')  
AND < SQL expression to define the restriction >
```

Creating a Dynamic Restriction Type

In an analytic privilege, in addition to static values filtering conditions, it is possible to determine the filtering conditions in a dynamic way, which means that the filtering condition is not defined once for all, but is evaluated dynamically when the view is executed. This is called *Dynamic Analytic Privilege*.



With a dynamic restriction, the filtering conditions that apply for a specific user are determined at runtime. This allows a more scalable approach, where the same analytic privilege can be granted to several users who have different authorization requirements.

For example, the COUNTRY attribute in one or several calculation views can be filtered dynamically based on the actual list of countries that each is allowed to access, depending on his position in the geographical organizational structure.

Technically, to implement a dynamic restriction, you assign to the analytic privilege one procedure, which returns a SQLScript expression to filter data, like in the WHERE clause of an SQL statement.

For example, a procedure could return ("COUNTRY"='US') for *User1* and ("COUNTRY"='UK' OR "COUNTRY"='FR') for *User2*.

This procedure must have the following properties:

Dynamic Restriction - Procedure Properties



- Procedure must be read-only
- Security mode must be *DEFINER*
- No input parameters
- Only one scalar output parameter of type VARCHAR(256) or NVARCHAR(256)

4. Set the Secured Calculation Views to Check SQL Analytic Privileges

To actually secure a calculation view with an analytic privilege, you must set the *Apply Privileges* property of the calculation view to *SQL Analytic Privileges*.



Figure 249: Apply Privileges Property

5. Save and Build the Analytic Privilege

During the build of calculation views and analytic privileges, a specific dependency check is triggered to avoid errors that would lead to unsecured calculation views. You get a build error in the following cases:

- If you try to build a calculation view after activating the check for SQL analytic privileges but NO analytic privilege has this view in its Secured Models list.
- If you try to build an analytic privilege but some of the (runtime) calculation views it secures do NOT have the property *Apply Privileges* set to *SQL Analytic Privileges*.
- If you try to build a calculation view after deactivating the check for SQL analytic privileges but there is still one or several (runtime) analytic privileges that have this view in their *Secured Models* list.



Note:

The best approach is to build in parallel analytic privileges and the calculation views that they secure.

When the analytic privilege is built, its runtime version is created in your container schema.

6. Assign the Analytic Privilege to a Role

Once an analytic privilege is built, the calculation views it applies to cannot be viewed until the privilege is granted to the end user.

To do so, in your project, you create a design-time role and grants the new analytic privilege to this role.

7. Assign the Role to a User

The last step is to grant the role to the end user. This can be done in the SAP HANA Studio or with an SQL statement.

Summary of the Analytic Privilege Evaluation During a Query on a Calculation View



The Analytic Privilege Check evaluates Analytical Privileges:

- Granted to the User
- Applicable to the queried Information Model
- Currently valid (validity dates)
- With Attribute restrictions covering attributes of the view

If no Analytic Privilege for the user can be found, user queries are rejected with a “...not authorized” error message.

Detailed info can be found in the trace file of the Index Server (Administration → Diagnosis Files)

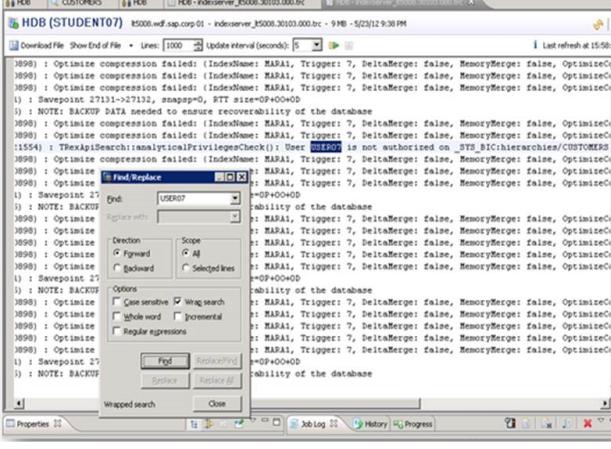


Figure 250: Analytic Privilege Check

The figure, Analytic Privilege Check, summarizes the analytic privilege check process.

If you need to solve issues related to the behavior of analytic privileges, you can find detailed information in the trace file of the index server:

- In the context menu of the SAP HANA system, choose *Administration* and select the *Diagnosis Files* tab.
- Locate the file *index server_<host name>.<port>.<counter>.trc* and double-click to open it.
- Choose *Show End of File* and search for the user name (press *Ctrl+F*).

The trace log displays the analytical privilege check errors.

Securing CDS Views with SQL Analytic Privileges

From SAP HANA 1.0 SPS11 onwards, it is possible to apply SQL analytic privileges to CDS views.

This offers a new possibility in implementation scenarios where CDS views are used and exposed to the end user.

To Secure a CDS View with an Analytic Privilege

1. Enable the CDS view for SQL analytic privilege check.

You include in the .hdbcards design-time file of the view the following annotation:

```
@WithStructuredPrivilegeCheck: true
```

2. Create an SQL analytic privilege.
3. Add the CDS view to the list of secured models.
4. Create an *Attribute* restriction type.
If needed, convert the restriction to an SQL expression.

Definition of Restrictions on Hierarchy Nodes in SQL Analytic Privileges

SAP HANA supports attribute restrictions based on a hierarchy node (rather than a list of values, and interval) in SQL analytic privileges.

This is useful in scenarios where a user who is enabled to a certain node of the hierarchy must also be enabled to all the descendants of this node.

With a geographical hierarchy, for example, the manager of the North America area will see all the countries in his area if the analytic privilege sets the attribute restriction to the NA node. You do not have to list all the countries from this area.

To Use a Hierarchy Node in an Attribute Restriction

1. Enable the calculation view for SQL analytic privilege check and SQL access to hierarchies.
2. Create an SQL analytic privilege.
3. Add the calculation view to the list of secured models.
4. Create an *Attribute* restriction type.
5. In the restriction, choose *Hierarchy Node*.
6. Select one of the available hierarchies.
7. Choose a hierarchy node.



Caution:

This feature is supported:

- With calculation views of the type *CUBE With Star Join* that are designed to check SQL analytic privilege and enable SQL access to hierarchy.
- Only for parent-child Hierarchies.

Defining Data Access Security with Nested Calculation Views

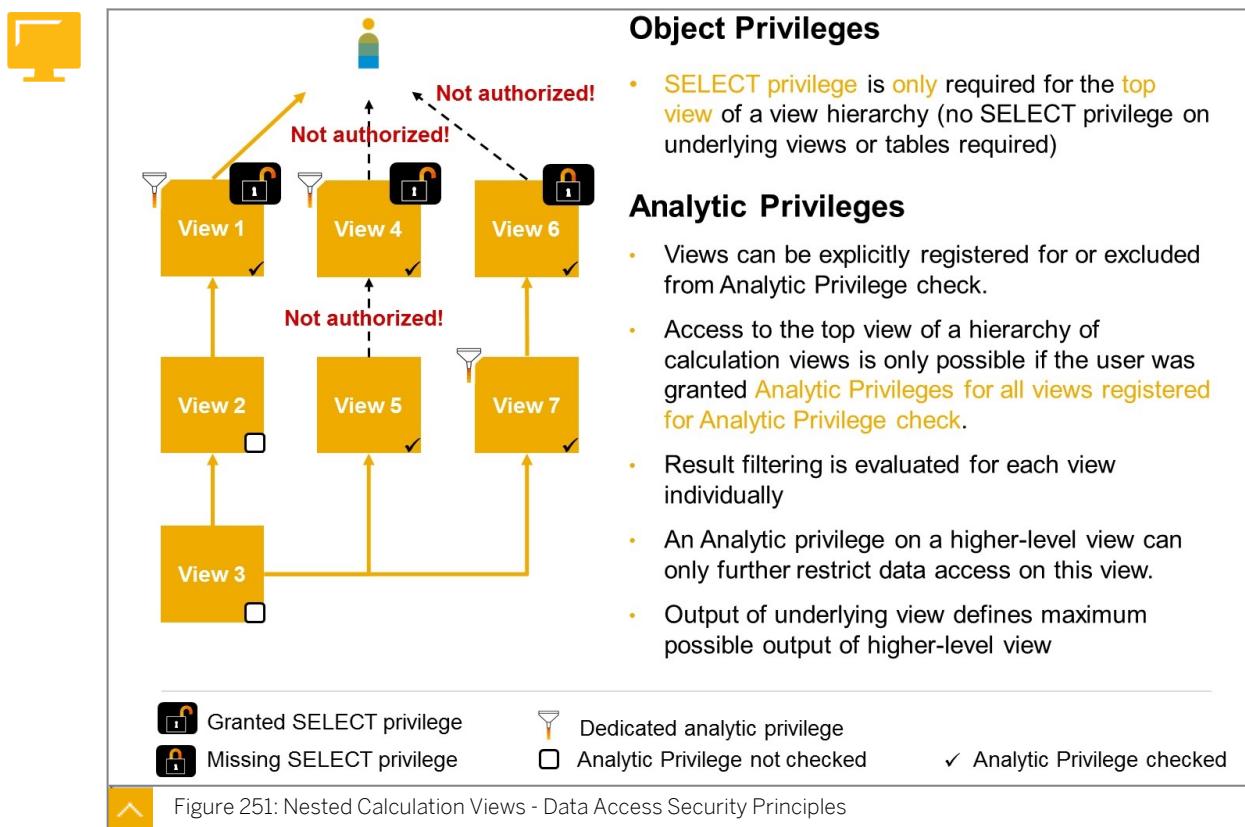
There are many business cases where calculation views contain references to one another.

The figure, Nested calculation views - Data Access Security Principles, explains how data access is handled when calculation views are nested.

For each view, the following criteria are considered:

- Does the user have `SELECT` privilege on the column view in the container schema?
- Is the view designed to check analytic privileges?
- Is the user granted analytic privileges for the view?

Nested Calculation Views - Data Access Security Principles



The key rules that govern the access to data are as follows:

- Object privileges

There is no need to grant **SELECT** privileges on the underlying views or tables. The end user only needs to be granted **SELECT** privileges on the top view of the view hierarchy.

- Analytic Privileges

The analytic privileges logic is applied through all the view hierarchy.

Whenever the view hierarchy contains at least one view that is checked for analytic privileges but for which the end user has no analytic privilege, no data is retrieved (not authorized).



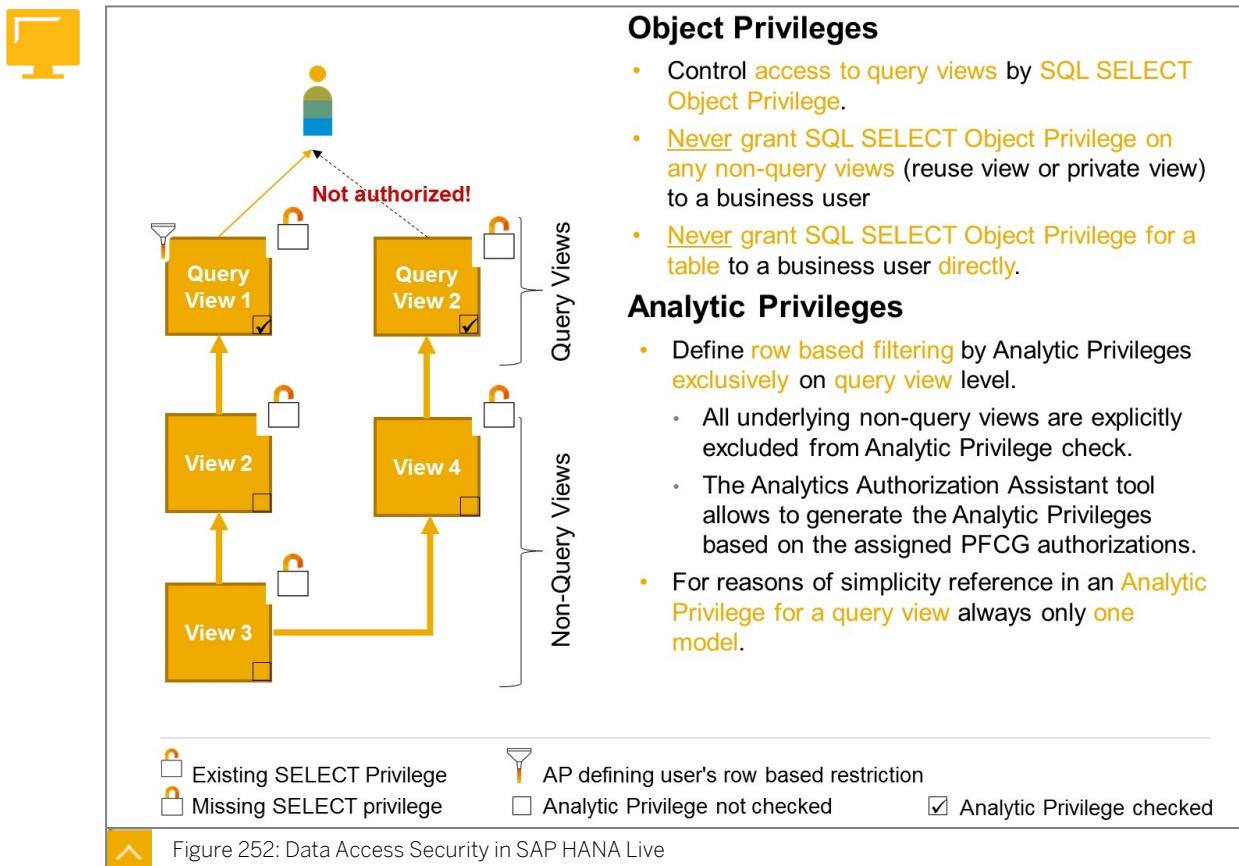
Note:

In the figure, Nested Calculation Views - Data Access Security Principles, this is the case for View 5, so View 4 will not retrieve any data.

This is also the case for View 6, and in addition, the end user is not granted a **SELECT** privilege on View 6. So View 6 will retrieve no data (not authorized).

Note that the end user always needs an explicit **SELECT** privilege on a calculation view to be able to query its data. That is, granting an Analytic Privilege to this user does not grant him an implicit **SELECT** authorization on the views that this Analytic Privilege secures.

Data Access Security in SAP HANA Live



The SAP HANA Live virtual data model relies heavily on views nesting, and in addition, classifies views either as query views (exposed to the end user) or as non-query views (namely, private and reuse Views) which should not be accessed directly by the end user.

The data access security of SAP HANA Live is implemented as follows:

- The end users are granted the object privilege SELECT exclusively on query views. They are never granted any privilege on non-query views or database tables.
- By design, query views perform a check for analytic privileges, and analytic privileges must be defined on these views to enable end users to see the data.

The non-query views do not perform any check for analytic privileges, which is only possible from a security perspective because end-users cannot access them (missing SELECT privileges on the views).

Enabling Access to an External Schema or Another Container

By default, a HDB module is only aware of the objects that are created locally, in the corresponding container schema – for example, tables created with design-time files .hdbcds, calculation views, and so on – and cannot access external data.

This is sufficient when the entire application manages the data persistence inside its own HDB module. However, there are cases where your need to read – or even modify – data located in an external location, such as classic database schema or another container. As an example, all the calculation views you built during this course are using data sources from two main external schemas: *EPM_MODEL* and *TRAINING*.



LESSON SUMMARY

You should now be able to:

- Define analytic privileges

Defining Roles

LESSON OVERVIEW

You have already learnt different types of objects that you can create inside the HDB module of an XS Advanced project. For example, Calculation Views, CDS tables for data persistence, Procedures, Analytic Privileges, and so on.

Another object type that you can also create with design-time files, and deploy inside your HDI container, is the Role. As you know, roles are used to distribute in a relevant way authorizations to database objects and data, either located in classic schemas or in the container(s) created as part of the deployment of an HDB Module.

One key benefit of working with design-time roles is that they can be easily transported along with the XSA Project, where pure catalog roles cannot.

Business Example

You work as a Modeler on an SAP HANA Implementation and want to create roles inside your project, in order to easily transport these roles to the production environment and assign them to end-users who will consume your information models.

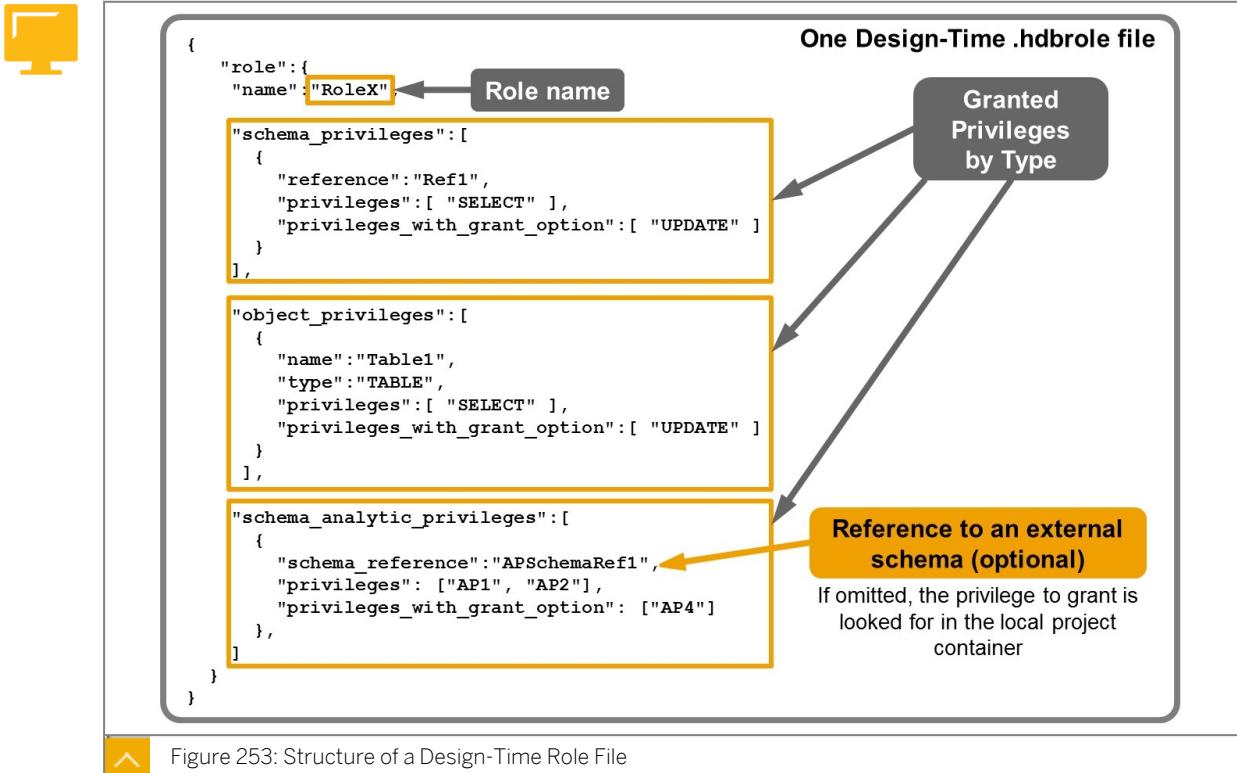


LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Create a design-time role

Design-Time Roles in an HDB Module



The design-time files used to create roles use a Java Simple Object Notation (JSON) syntax, and must have the extension `.hdbrole` in order to be recognized as design-time role files by the HDI build plug-in.



Note:

Each role must be defined in its own `.hdbrole` design-time file.

It is not possible to create several roles within the same `.hdbrole` file.

The role ID (including a valid namespace if applicable) must be unique in the HDB module, as for any other object (calculation view, synonym, and so on).

Types of Privileges in a Design-Time Role

A `.hdbrole` design-time file can include one or several of the following privileges or roles:

Table 27: Types of Privileges in a `.hdbrole` Design-Time File

Type of Role and Notation in the <code>.hdbrole</code> file	Description
Global Role <code>"global_roles"</code>	Global roles, including a number of built-in roles, defined without a schema, for example: PUBLIC, RESTRICTED_ROLE.
Schema Role (*) <code>"schema_roles"</code>	Schema-local roles defined in your container schema or another schema.

Type of Role and Notation in the .hdbrole file	Description
System Privilege "system_privileges"	System privileges, such as USER ADMIN, ROLE ADMIN.
Privilege on a Schema (*) "schema_privileges"	Privilege that applies to an entire schema, such as SELECT, UPDATE, DELETE, and so on. When granted, these privileges apply to all the objects of the schema (for example, all tables, all views).
Object Privilege "object_privileges"	Privileges on objects defined within the local container of your project, such as a table, synonym, procedure or function.
Schema Analytic Privilege (*) "schema_analytic_privileges"	Analytic privileges defined in the local container of your project, or in another container or schema.

(*) For these types of roles or privileges, a reference to a schema must be provided if you want to specify from which external schema the role or analytic privilege must be granted, or to which external schema a schema privilege applies.

If this reference is not specified, these privileges are looked for in the local container (schema) of your HDB module.



Caution:

The .hdbrole file cannot contain references to real schema names, but only logical references that are resolved in another type of design-time file: the .hdbroleconfig file.

The .hdbroleconfig File

A .hdbroleconfig file must be created to “resolve” the name of external schemas (classic database schemas or other database containers) that are referenced in the .hdbrole file.

The purpose is to maintain the actual name of the external schemas in dedicated files, instead of having many occurrences of the schema names in the .hdbrole files themselves. It makes the maintenance of a project easier when you need to change the references to external schemas.

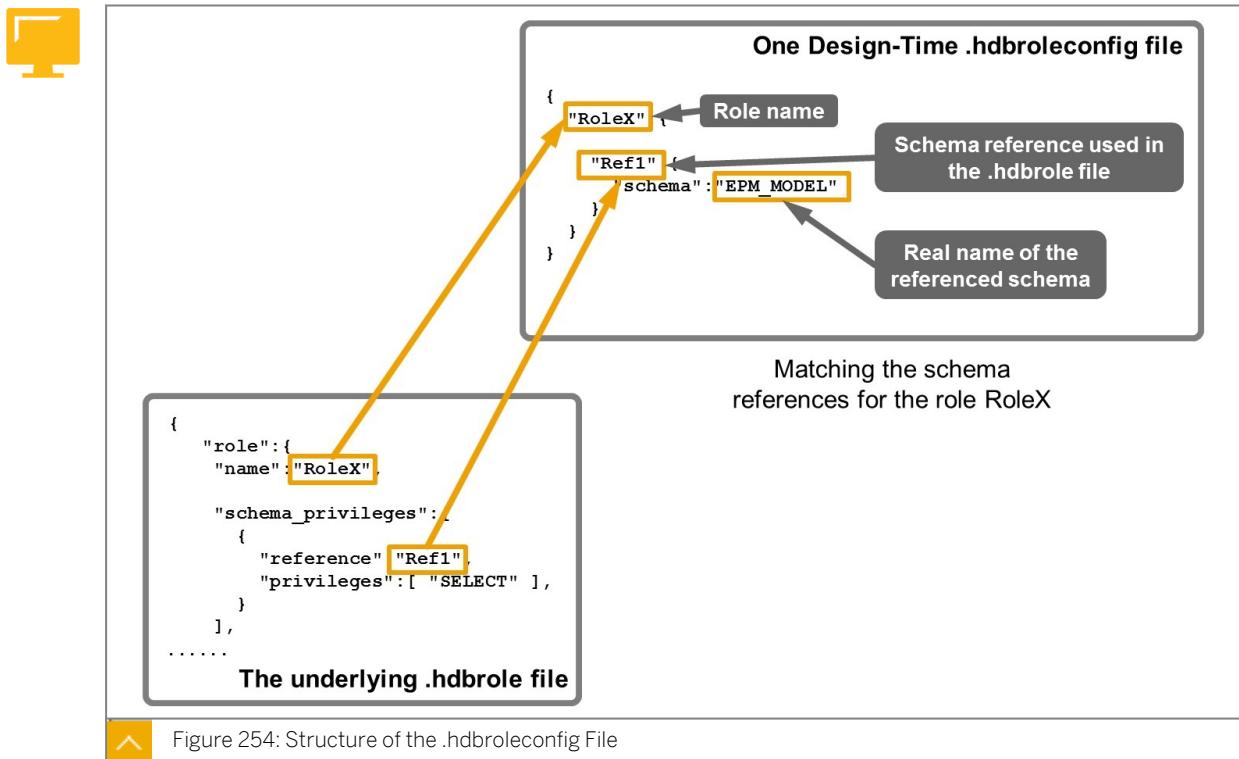


Figure 254: Structure of the .hdbroleconfig File

Example of a Design-Time Role File

The following is an example of a .hdbrole file with the different types of privileges.

```
{
  "role": {
    "name": "RoleX",
    "global_roles": [
      "MODELING",
      "DATA ADMIN"
    ],
    "schema_roles": [
      {
        "schema_reference": "RoleSchemaRef1",
        "names": [ "RoleA", "RoleB" ]
      },
      {
        "schema_reference": "RoleSchemaRef2",
        "names": [ "RoleB", "RoleC" ]
      }
    ],
    "system_privileges": [
      "BACKUP ADMIN",
      "USER ADMIN"
    ],
    "schema_privileges": [
      {
        "reference": "Ref1",
        "privileges": [ "SELECT" ],
        "privileges_with_grant_option": [ "UPDATE" ]
      }
    ],
    "object_privileges": [
      {
        "name": "Table1",
        "type": "TABLE",
        "privileges": [
          {
            "privilege": "SELECT"
          },
          {
            "privilege": "UPDATE"
          }
        ]
      }
    ]
  }
}
```

```

        "privileges": [ "SELECT" ],
        "privileges_with_grant_option": [ "UPDATE" ]
    }
],
"schema_analytic_privileges": [
{
    "schema_reference": "APSchemaRef1",
    "privileges": [ "AP1", "AP2" ],
    "privileges_with_grant_option": [ "AP4" ]
},
]
}
}

```

**Note:**

The complete syntax of the .hdbrole design-time file can be found in the SAP HANA Developer Guide for SAP HANA XS Advanced Model, on the SAP Help Portal. Go to <http://help.sap.com/hana>.

Granting a Privilege With Grant Option

When a privilege is granted (to a user or to a role), there are two possible options:

- “Simple” Grant

In turn, the grantee is NOT allowed to grant the privilege to another user or role.

- Grant With Grant Option

In this case, the grantee is allowed to grant the privilege to another user or role. So this approach gives more “power” to the grantee.

Privilege With Grant Option — Example



```
{
    "role": {
        "name": "RoleABC#",
        "schema_analytic_privileges": [
            {
                "privileges": [ "AP1", "AP2" ],
                "privileges_with_grant_option": [ "AP4" ]
            },
        ]
    }
}
```

In the example, *RoleABC#* includes three Analytic Privileges: AP1, AP2, and AP4.

A user who is granted *RoleABC#* will be allowed to grant AP4 to another user, but not AP1 or AP2.

**Caution:**

A role name MUST always end with the # sign (for example, **RoleABC#**) if the role includes privileges WITH GRANT OPTION.

If this naming rule is not applied, the build of the role fails and the error is reported in the build log.

**Note:**

As you can see from the example, the three analytic privileges AP1, AP2, and AP4 must exist in the local container, because there is no schema reference.

Using the Role Editor

SAP HANA 2.0 SPS05 comes with a new graphical editor for roles. It is invoked by default when you create or edit a .hdbrole file, but you can always use the code editor if you prefer.

The role editor supports the definition of roles with and without .hdbroleconfig files, and the following categories of roles:

- Object Privileges
- Schema Privileges
- Analytic Privileges
- System Privileges

It also allows you to assign one or several roles to the current role (roles nesting).

Note that the objects or synonyms defined inside your container do not need a schema specification. This is materialized in the *Object Privileges* tab (no *Schema* column).

Roles and Privileges Relevant for the Modeler Role

Among the different types of privileges we discussed before, the most relevant for the Modeler role are the following ones:

Privilege or Role	Use Case for the Modeler Role
Schema Analytic Privilege	Assign the analytic privileges defined in your project to roles.
Object Privilege	Control accurately the access to the database objects defined in your local container (schema), such as tables, views, procedures, and so on.
Privilege on a Schema	Grant to a role some schema privileges that were given by the technical user configured in a user-provided service (for cross schema access). For example, SELECT on the external schema.
Schema Role	Group the roles defined in your project together in order to have a relevant nesting of roles.

**Note:**

The Object Privilege section can also be used to grant authorization to specific database objects located in external schemas that you access via synonyms. However, in the context of calculation view modeling, in most cases, you do not need to grant SELECT privileges on the underlying tables (even if they are located in an external schema), but only SELECT privileges on the calculation views that are defined in your HDB Module and located in the HDI container schema.

Default Roles Created in a Container

Whenever you build a HDB module, two roles are created by default in the corresponding container schema. These roles are named as follows:

- <CONTAINER_SCHEMA_NAME>::access_role
- <CONTAINER_SCHEMA_NAME>::external_privileges_role

For example, when you built the HDB module of your HA300_## project, during Exercise 1, it created the two default roles mentioned above. You also granted the role HA300_##_HDI_HDB_1::access_role to the TRAINING_ROLE_##.

The ::external_privileges_role does not contain any privileges by default.

The ::access_role contains the following privileges:

Privileges Included in the <CONTAINER_SCHEMA_NAME>::access_role

- The following schema privileges on the local container schema: CREATE TEMPORARY TABLE, DELETE, EXECUTE, INSERT, SELECT, SELECT CDS METADATA, UPDATE
- The privileges on external schema objects granted via the .hdbgrants file (see below).



Caution:

The <CONTAINER_SCHEMA_NAME>::access_role can be assigned to some developers during the development phase of your project, but it provides a number of authorizations that might not be needed/relevant for the application users. For example, select/update/insert/delete/execute on the entire container schema, which means, all the tables, views, procedures, and so on.

For this reason, it is recommended to define your own roles and adjust the granted privileges to the exact access requirements of each category of users.

If you do not want the default access role <CONTAINER_SCHEMA_NAME>::access_role to grant these extended privileges, you can even provide your own definition for this role. This is done by means of a specific file `default_access_role.hdbrole` located in a specific folder `<HDB module name>/src/defaults/`. All the default permissions above will be revoked, and only the ones provided in your `default_access_role.hdbrole` file will be included in the default access role, in addition to the authorization to external schemas granted by the grantor user (for example, in our landscape, TRAIN_TU).

Enabling Access to an External Schema

When your application requires access to an external schema, an administrator must define a dedicated user-provided service. A technical user is assigned to this service, and brings its own authorizations to the database objects.

As part of the security implementation in your project, it is necessary to define which of these authorizations will be granted to the different roles. For example, some users might need insert/update/delete privileges on a particular set of tables, when other users only need select privileges.

**Note:**

When creating calculation views, the main authorization you need is only a SELECT privilege on the data sources.

But other types of applications must write data to an external persistence layer, and in that case privileges such as INSERT/UPDATE/DELETE on the corresponding tables must be granted to some users.

The .hdbgrants File

A dedicated file, with extension *.hdbgrants*, is used to define the set of authorizations that will be given to two specific users, the *Object Owner* and the *Application User*.

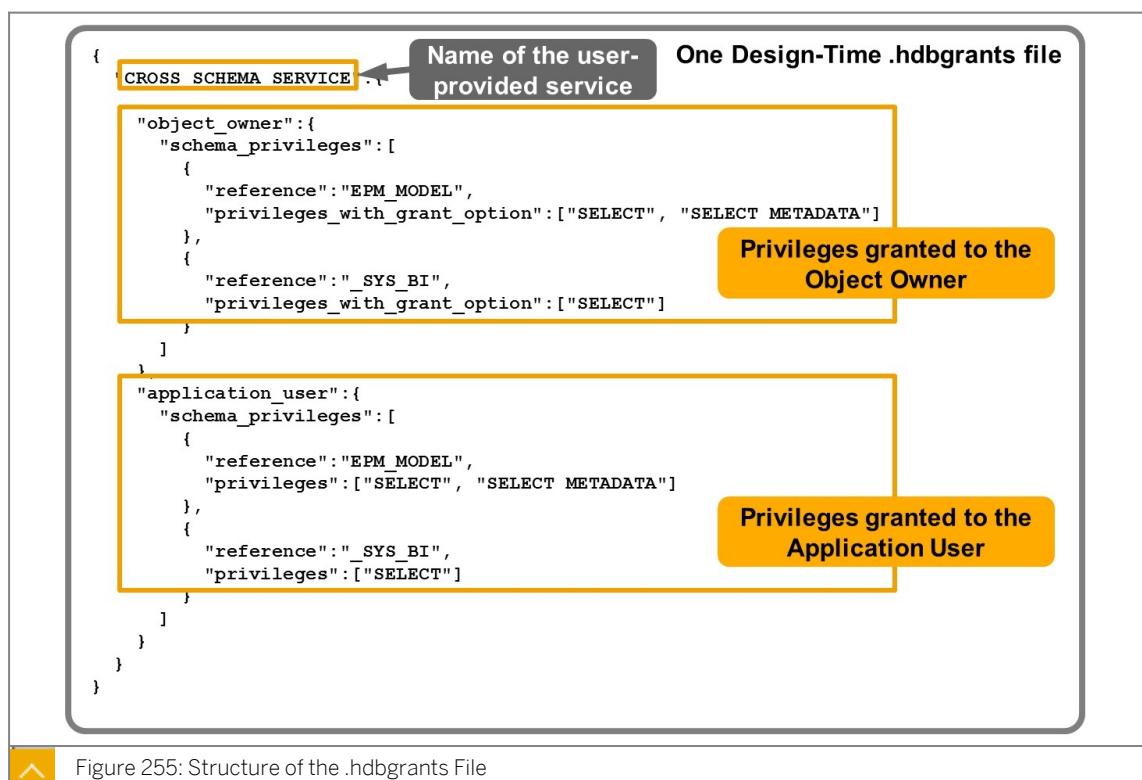


Figure 255: Structure of the .hdbgrants File

The *<filename>.hdbgrants* file is structured into three levels:

- The name of the user-provided service
- The users to whom the privileges are granted

There are two possible “values”:

- *object_owner* is the technical user that owns all the objects of the container schema.
- *application_user* represent the users who are bound to the application modules.

When you create models in the SAP Web IDE, for example, the application user is the technical user that works “on your behalf” inside the container and accesses the external schemas, if any.

- The set of privileges granted

The syntax of this third level is very similar to the syntax of what you find in a *.hdbrole* file.



Note:

A single .hdbgrants file can list authorizations from more than one user-provided services.

It is essential to give the *application_user* a correct set of authorizations. For example, if a SELECT privilege on an external table is granted to the *object_owner*, this will allow the creation of a synonym for this table. But if the same SELECT privilege is not granted to the *application_user*, you won't be able to display the content of the target table.



LESSON SUMMARY

You should now be able to:

- Create a design-time role

Unit 8

Lesson 4

Masking Sensitive Data



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Restrict access to columns containing sensitive data within a View

Column Masking

When calculation views might expose sensitive data to the end-user, it is possible to define a mask on the corresponding column(s) so that only authorized users will be allowed to see the actual data. For other users, the mask will be applied in order to hide part or all of the column when previewing the calculation view data or querying it with a front-end tool.

The key building blocks of column masking are as follows:

- A mask expression, defined in the *Semantics* node of a calculation view.
- An authorization on the view to visualize unmasked data. If this authorization is not granted, the column's data is masked according to the mask expression.



RB	CUSTOME	RB	FIRSTNAME	RB	LASTNAME	RB	ACCOUNTNUMBER
	001		Franz		Kafka		XXXXXXXXXX
	002		Ernest		Hemingway		XXXXXXXXXX
	003		Thomas		Mann		XXXXXXXXXX
	004		Stanislaw		Lem		XXXXXXXXXX

Bob does not have **UNMASKED** privilege for the view

RB	CUSTOME	RB	FIRSTNAME	RB	LASTNAME	RB	ACCOUNTNUMBER
	001		Franz		Kafka		123456789
	002		Ernest		Hemingway		987654321
	003		Thomas		Mann		555555555
	004		Stanislaw		Lem		444444444

Alice has **UNMASKED** privilege for the view

RB	CUSTOME	RB	FIRSTNAME	RB	LASTNAME	RB	ACCOUNTNUMBER
	001		Franz		Kafka		123456789
	002		Ernest		Hemingway		987654321
	003		Thomas		Mann		555555555
	004		Stanislaw		Lem		444444444

Masked column



Figure 256: Column Masking



Note:

In addition to graphical calculation views, data masking can also be defined on the columns of other objects, namely: tables and SQL views.

Defining a Mask Expression

Only columns of certain data types can be masked in a calculation view:

- *VARCHAR*
- *NVARCHAR*
- *CHAR*
- *SHORTTEXT*



Note:

Masking is supported for both table types (ROW tables and COLUMN tables).

The mask expression can be either a constant string (for example: **xxx-xxx-xxx**) or an SQL expression using string functions and/or data from the source column.

For example, you can mask the middle part of a credit card number stored in column *credit_card* with the following mask expression:

```
LEFT(credit_card, 4) || '-XXXX-XXXX-' || RIGHT(credit_card, 4)
```

With this expression, the credit card number 1111-2222-3333-4444 will be masked as 1111-XXXX-XXXX-4444.

Authorizing Access to a Masked Column

When a column of a calculation view is assigned a mask expression, the data of this column is masked unless the user has the *UNMASKED* privilege for this calculation view. In the context of XS Advanced modeling, this privilege is generally included in one or several roles defined in the HDB module, and this role is in turn granted either to the end-user or, most often, to a role assigned to the end-user.

The *UNMASKED* privilege can be granted at two different levels, which corresponds to two different role definition syntaxes:

- At the schema level

The authorization to view unmasked data (the actual value) is given for an entire schema. In particular, if no schema is specified, the authorization affects all the calculation views defined in the container schema where the role is defined, which is the most common case.

```
{
  "role": [
    {
      "name": "db::UNMASK_ENTIRE_SCHEMA",
      "schema_privileges": [
        {
          "privileges": ["UNMASKED"]
        }
      ]
    }
  ]
}
```

Note that the authorization at the schema level allows to unmask columns in all types of objects that support masks, namely: tables, SQL views and graphical calculation views.

- At the object level:

The authorization to view unmasked data is given for a specific object. You have to specify the object type and object runtime name (with namespace).

```
{
  "role": [
    {
      "name": "db::UNMASK_EMPLOYEES_PAYMENT",
      "object_privileges": [
        {
          "type": "VIEW",
          "name": "db::CVD_EMPLOYEES_PAYMENT",
          "privileges": ["UNMASKED"]
        }
      ]
    }
  ]
}
```

Once the role is built, it can be assigned to a user or role with the following SQL query:

```
GRANT "<container_schema_name>.<unmasking_role_name>" TO <user_name>
| <role_name>;
```

For example:

```
GRANT "HA300_01_HDI_HDB_1"."HA300::UNMASK_EMPLOYEE_PAYMENT" TO
TRAINING_ROLE_01;
```

This query must be executed by a user that is allowed to grant the relevant roles, or the UNMASKED privilege on the relevant objects, to the end-user.



Note:

Like for any other calculation view, the end user will receive an authorization error if he does not have a *SELECT* authorization on the view. In other words, the *UNMASKED* privilege does not by-pass the *SELECT* privilege which is mandatory to display a calculation view's data.

Current Limitations to Column Masking

Two important limitations apply to column masking.

- Nesting of Calculation Views

When two calculation views are nested, a column masking defined in an underlying calculation view does not affect the calculation view that consumes it. As a consequence, you must make sure that the calculation views that you expose to the end users contain the relevant mask definitions.

- Calculated Columns Use Unmasked Data

When the expression of a calculated column refers to a column that has a column mask assigned, the calculation uses unmasked data, regardless of whether the user has the *UNMASKED* privilege for the view or not.



Caution:

Make sure to keep these two limitations in mind when creating calculation views, to avoid security issues.



LESSON SUMMARY

You should now be able to:

- Restrict access to columns containing sensitive data within a View

Unit 8

Lesson 5

Anonymizing Data



LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Protect sensitive data with anonymization

Data Anonymization

Anonymization is the process of removing specific details from data, so as to disassociate sensitive data from identifiable individuals or entities.

Let's take an example based on a very simple data set.



Name	Birth	City	Weight	Illness
Paul	07-1975	Walldorf	82 kg	AIDS
Martin	10-1975	Hamburg	110 kg	Lung Cancer
Nils	01-1975	Munich	70 kg	Flu
Annika	09-1987	Berlin	58 kg	Multiple Sclerosis



Name	Birth	Location	Weight	Illness
0c4a67	1975	Germany	~ 96 kg	AIDS
df89aa	1975	Germany	~ 96 kg	Lung Cancer
305be2	19**	Germany	~ 64 kg	Flu
7422c2	19**	Germany	~ 64 kg	Multiple Sclerosis

Identifiers *Quasi-Identifiers* *Sensitive*

Figure 257: Anonymizing Data — Example

In this sample data set, the date of birth, location and weight can be considered as quasi-identifiers. Indeed, suppose that someone knows that the analyzed group of people contains Annika, who is born in 1987. Then, if the analysis results show that someone suffering from multiple sclerosis was born in 1987, as a consequence, you can guess that this is actually Annika. In this case, grouping this date of birth together with other dates (for example, 19**) helps conceal the identity of the person.

Similarly, knowing the City where people live and showing the city details in an analysis would enable you to guess which diseases people suffer from. Similarly, replacing year/month with only year helps make Paul and Martin “similar” with regards to the date of birth attribute.

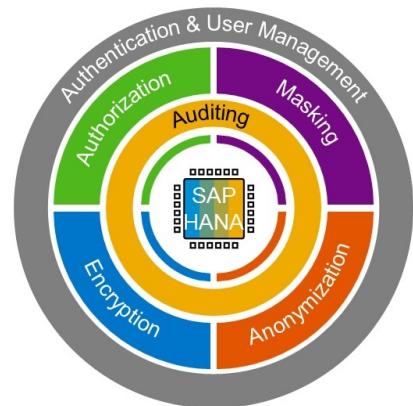
Now, if you consider the City, because in our very simple data set each person lives in a different city, the city allows you to identify people if you know where they live. Indeed, the only person who has lung cancer lives in Munich, and nobody else in the analyzed data set lives in Munich. In that case, replacing the city with the country solves the privacy issue, at least if you consider that having four different people located in Germany is "enough" to prevent anybody from identifying them.



Masking and anonymization address different use cases

Masking

- Selectively hide sensitive information from DBAs and power users with broad access
- Display/hide sensitive information depending on the user role, e.g. for call center employees



Anonymization

- Structured approach to protect the privacy of individuals in complex data sets
- Real-time analytics on anonymized data
- Enables insights from data that could not be leveraged before due to regulations



Figure 258: Anonymization vs. Masking

Unlike masking, which hides data, anonymization serves a different purpose - namely, to allow analysis of data in the aggregate, without consideration of individual records.

Anonymization Methods in SAP HANA

There are two kinds of anonymization supported in SAP HANA, both built on standard mathematical models:

- k-Anonymity
- Differential privacy

Note that the following examples are approximate models only. In order to meet standards, both are much more complex than what's shown here.



Name	Birth	Location	Weight	Illness
0c4a67	1975	Germany	~ 96 kg	AIDS
df89aa	1975	Germany	~ 96 kg	Lung Cancer
305be2	19**	Germany	~ 64 kg	Flu
7422c2	19**	Germany	~ 64 kg	Multiple Sclerosis

Identifiers

Quasi-Identifiers

Sensitive



Figure 259: k-Anonymity

k-Anonymity works by removing details from data, by grouping similar data into sets where the individual values are not easily identified. Swapping the detailed values of a column to a higher level of information is known as 'generalizing'.

For example, we could refer to a hierarchy in order to remove the lowest level of detail and instead refer to data from the next highest level.

- An individual listed as living in a particular city could instead be listed as living in a province or state, with many other individuals.
- A value represented by an integer (income, age, and so on) could be represented by ranges. For example, the individual might be listed as being born in "the seventies".
- For the individual values to be truly anonymized, there must be a sufficiently large grouping so that it becomes unlikely to guess the individual record.
- If only one person in a transnational company answering a survey is employed in a given country, then grouping by city, province, or even country is not enough to provide anonymity.
- On the other hand, if there are 5-10 employees in a given business group, but more than 100 in a given office, the "office level" would provide sufficient anonymity to make it unlikely to identify a given employee's response.



Name	Birth	City	Weight	Salary
0c4a67	07-1975	Walldorf	82 kg	65k + $x_1 = 12k$
df89aa	10-1975	Hamburg	110 kg	34k + $x_2 = -30k$
305be2	01-1975	Munich	70 kg	75k + $x_3 = 140k$
7422c2	09-1987	Berlin	58 kg	105k + $x_4 = 80k$



Figure 260: Differential Privacy

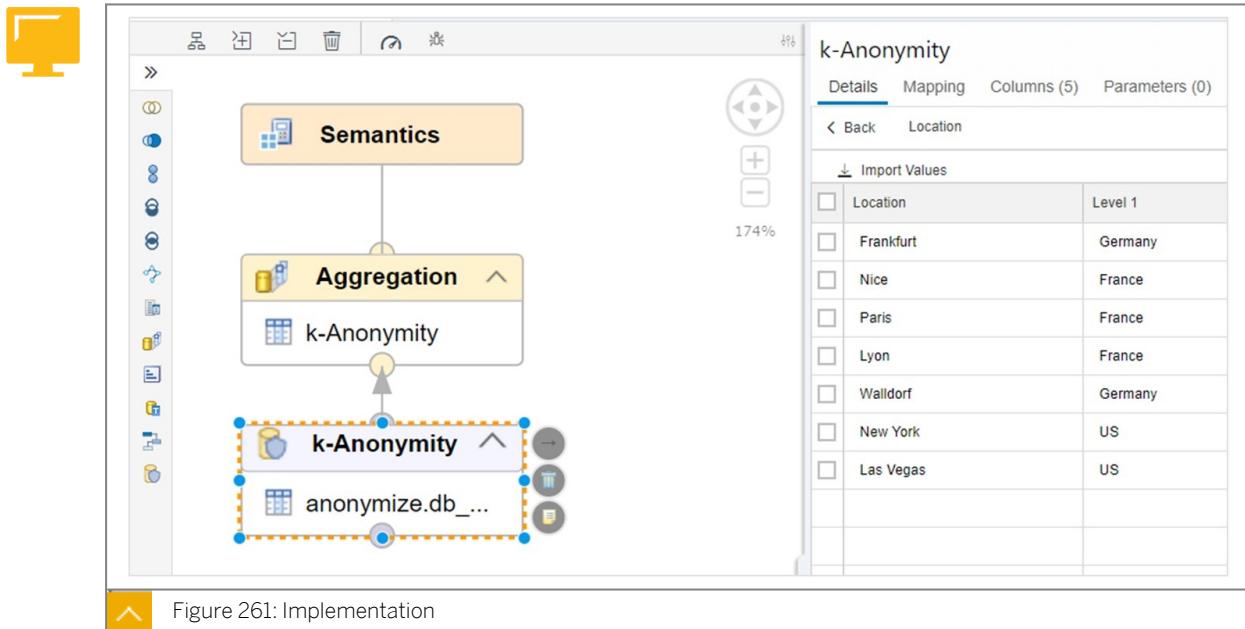
Differential privacy works by introducing specific values of error into the data. The values of a column anonymized with this algorithm are no longer displayed as their actual values.

However, the error is introduced such that aggregation of the "noisy" data will be very close to aggregated value of the actual data.

For example, a salary column could be obscured by randomly offsetting its true value by either -2000, -1000, +1000, or +2000.

- If each of the offsets is used approximately the same number of times, an average of the offset values will be very close -or even the same- as the average of the true values.
- With a low count of values to be offset, however, it's possible that an unequal number of the offsets could be used, introducing error into the final result.
- With a high count of values to be offset, the overall impact of any single offset will be small; especially when the aggregated values can be rounded off to a modest level of accuracy (for example, total units sold across a product line to the nearest multiple of 1000), the error may be negligible.

Implementing Anonymization



Anonymization is implemented in graphical Calculation Views through a dedicated node type. Once you have selected the data source, you must define the setting for the Anonymization node.

This starts by the choice of the anonymization method (k-Anonymity or Differential Privacy).

Table 28: K-Anonymity Settings

Setting	Definition
Sequence Column	Identify a sequence column (SEQ) among the columns of type INT or BIGINT included in the output (mapping) of the node. This SEQ columns will be used to identify each row of the source data.
k	Minimum number of distinct values in the generalization hierarchy of a quasi-identifying information column.
Loss	The loss parameter helps specify the proportion (expressed as %) of records that can be dropped from the results in order to avoid generalizing whole columns due to a very small number of records that would cause generalization and therefore lead to a loss of detailed information. Of course you lose whole records in this process but the idea is that you accept losing a few records for the benefit of analysis of the remainder at a decent level of detail.
Quasi Columns	List of your data source columns that contain the quasi identifying information. Only VARCHAR and NVARCHAR data types are supported.
Generalization hierarchies	For each quasi-identifying column, you can define the hierarchy manually (this is called 'embedded') within the anonymity node hierarchy editor. To avoid redundancy you can use an external calculation view to define the generalization hierarchies (good for reuse and centralized maintenance). The calculation view you refer to must contain a hierarchy function node or a table function. A modelled (reporting) hierarchy defined in the semantic node is not valid.

Setting	Definition
Weight	By weighting one or more columns in your definition of k-anonymity, you can adjust the relative importance of different attributes. This allows you to influence the data quality after generalization in the quasi-identifying columns. The value is enter between 0–1, where 1 is the stronger preference to keep the original column values. This allows you to attempt to hold on to the columns that give the most meaning but losing the other less valuable column to a higher level of detail.
Min Level	In addition to the main k anonymity parameters, you can also define the number of distinct values that must be reached before the column value is shown in the results. This setting can be used to ensure that a column must ALWAYS be generalized for example, gender regardless of the outcome of the other k anonymity rules.
Max Level	This is the opposite of the above and can be used to determine that a column should never be generalized regardless of the outcome of the other k anonymity rules..

Table 29: Differential Privacy Settings

Setting	Definition
Epsilon	This setting controls the influence of individual values on statistics. If influence is too high, privacy might be broken. Lower values achieve better privacy, but also less reliable statistics. It should be set by security experts. Typical values are between 0.01 and 0.1
Sensitivity	This setting needs to be set high enough so that privacy can be guaranteed. If set too high, statistics become less reliable. Should be set to the difference between the max. and min. values to be concealed.
Noised Column	The column to which noise must be added. Only FLOAT and DOUBLE data types are supported.



LESSON SUMMARY

You should now be able to:

- Protect sensitive data with anonymization

Learning Assessment

1. What is a dynamic analytic privilege?

Choose the correct answer.

- A A reusable analytic privilege that can be used for several users who need to access different data.
- B An analytic privilege that takes its filter values from variables defined in the calculation view
- C A temporary analytic privilege that has a defined time validity set for its use

2. What is data masking?

Choose the correct answer.

- A Obscuring column values by hiding some or all characters with replacement characters
- B Replacing column values with a higher level group such as replacing a person id with team name
- C Removing complete rows of data that should not be seen by users

3. Why do you implement data anonymization?

Choose the correct answer.

- A To hide sensitive data using alternative characters such as 'XXXXX'.
- B To hide sensitive data by adding noise at the record level so that aggregated values still make sense but individual records are not exposed.
- C To remove complete records so that they cannot be seen by unauthorized users

Learning Assessment - Answers

1. What is a dynamic analytic privilege?

Choose the correct answer.

- A A reusable analytic privilege that can be used for several users who need to access different data.
- B An analytic privilege that takes its filter values from variables defined in the calculation view
- C A temporary analytic privilege that has a defined time validity set for its use

Correct — A dynamic privilege is a reusable analytic privilege that can be assigned to several users who need to access different data.

2. What is data masking?

Choose the correct answer.

- A Obscuring column values by hiding some or all characters with replacement characters
- B Replacing column values with a higher level group such as replacing a person id with team name
- C Removing complete rows of data that should not be seen by users

Correct! — Data masking is obscuring column values by hiding some or all characters with replacement characters, such as a bank card xxxx-xxxx-xxxx-3476.

3. Why do you implement data anonymization?

Choose the correct answer.

- A To hide sensitive data using alternative characters such as 'XXXXX'.
- B To hide sensitive data by adding noise at the record level so that aggregated values still make sense but individual records are not exposed.
- C To remove complete records so that they cannot be seen by unauthorized users

Correct! — data anonymization uses algorithms to hide sensitive data by adding noise at the record level so that aggregated values still make sense but individual records are not exposed.