

# **HA150**

**SAP HANA 2.0 SPS06 - SQLScript  
for SAP HANA**

**PARTICIPANT HANDBOOK  
INSTRUCTOR-LED TRAINING**

Course Version: 18  
Course Duration: 3 Day(s)  
Material Number: 50159857

# 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	<b>Course Overview</b>
<b>1</b>	<b>Unit 1: Fundamentals</b>
3	Lesson: What is the Difference Between SQL and SQLScript?
11	Lesson: Explaining SAP HANA XS Advanced and SAP HANA Deployment Infrastructure (HDI)
17	Lesson: Understanding HDI and Working with the Database Explorer
21	Lesson: Describing the SAP HANA Database Module
25	Lesson: Working with the SAP Web IDE for SAP HANA
37	Lesson: Understanding the Course Data
<b>47</b>	<b>Unit 2: SQL - Refreshing the Essentials</b>
49	Lesson: Understanding Motivation and Basic Concepts
59	Lesson: Using Data from a Table or View
83	Lesson: Understanding NULL Values
87	Lesson: Aggregating Data
95	Lesson: Understanding Unions and Joins
115	Lesson: Changing Data Stored in Tables
<b>127</b>	<b>Unit 3: SQL Logic Containers</b>
129	Lesson: User-Defined Functions
137	Lesson: Creating Database Procedures
143	Lesson: Trapping Errors in SQLScript
151	Lesson: Creating User-Defined Libraries
<b>161</b>	<b>Unit 4: Declarative Logic</b>
163	Lesson: Using Declarative Logic
<b>173</b>	<b>Unit 5: Imperative Logic</b>
175	Lesson: Implementing Imperative Logic
<b>193</b>	<b>Unit 6: Temporal Tables</b>
195	Lesson: Working with Temporal Tables
<b>203</b>	<b>Unit 7: OLAP Operations</b>
205	Lesson: Using OLAP Analytic Features
<b>223</b>	<b>Unit 8: Hierarchies</b>
225	Lesson: Working with Hierarchies

**233      Unit 9:    Troubleshooting and Best Practices**

- |     |  |
|-----|--|
| 235 | Lesson: Understanding the Tools and Views to Analyze and<br>Optimize SQL |
| 243 | Lesson: Applying Tools and Views to Optimize SQL Performance             |
| 255 | Lesson: Further Tools for Troubleshooting                                |

**261      Unit 10:    Appendix - SQL Fast Track**

- |     |  |
|-----|--|
| 263 | Lesson: Using Sub Queries                |
| 275 | Lesson: Defining How Data Is Stored      |
| 285 | Lesson: Using Views for Data Access      |
| 293 | Lesson: Defining Data Access             |
| 301 | Lesson: Explaining Database Transactions |

# Course Overview

## TARGET AUDIENCE

This course is intended for the following audiences:

- Application Consultant
- Development Consultant
- Technology Consultant
- Database Administrator
- Developer



# UNIT 1

# Fundamentals

## Lesson 1

What is the Difference Between SQL and SQLScript?

3

## Lesson 2

Explaining SAP HANA XS Advanced and SAP HANA Deployment Infrastructure (HDI)

11

## Lesson 3

Understanding HDI and Working with the Database Explorer

17

## Lesson 4

Describing the SAP HANA Database Module

21

## Lesson 5

Working with the SAP Web IDE for SAP HANA

25

## Lesson 6

Understanding the Course Data

37

## UNIT OBJECTIVES

- Understand SQL
- Understand how SQLScript extends SQL
- Understand XS advanced and HDI
- Understand HDI and work with the Database Explorer
- Describe the SAP HANA Database Module
- Introduce the SAP Web IDE for SAP HANA
- Introduce the SQL Console of SAP Web IDE for SAP HANA
- Understand the sample database used throughout the course



# What is the Difference Between SQL and SQLScript?



## LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Understand SQL
- Understand how SQLScript extends SQL

## What is SQL?

### A Brief Introduction to SQL

Structured Query Language (SQL) is a standardized language for communicating with a relational database.

SAP HANA database is a relational database and fully supports SQL.

SQL is used to retrieve, store, or manipulate data in the SAP HANA database.



### What is SQL?

In almost every business application scenario, the data is managed using database systems.

The most significant are database systems based on the relational data model and using SQL (Structured Query Language) as a database language.

SQL is a widely-established powerful, standardized database language many application programmers have experience in.

There is (so far) no other database language that has all the advantages mentioned.

- SAP HANA is a relational database management system
- SAP HANA supports SQL



Figure 1: What is SQL?

SQL is a declarative language which simply means that you write statements that describe **what** you want to achieve and not precisely **how** the system should achieve it.

For example, when you request data from multiple tables, you do not define the sequence in which tables should be opened. You leave that decision to the database so that it can decide on the most optimal approach. Depending on the size of tables, the location, and the dependencies on other parts of the SQL statement, the system may read the tables in any order, or perhaps even in parallel. A key goal of SQL is to execute your statements using the

best performance possible. Consider that landscapes can change and so the precise execution of your SQL may change over time. What does not change is your **intent**.



### SQL language elements can be divided into the following categories:

Data Manipulation Language (DML) statements	DML statements insert, update and delete the data in a database. DML statements also select data that you want to read.
Data Definition Language (DDL) statements	DDL statements create, alter and drop databases, tables and other database objects in the server.
Data Control Language (DCL) statements	DCL statements grant and revoke permissions from database users.

Using SQL, the following tasks can be performed on a relational database

- Schema Definition and Manipulation
- Data Manipulation
- System Management
- Session Management
- Transaction Management



Figure 2: Types of SQL Statements

SQL is written in a series of statements. There are many statements which can be used with SQL to interact with a database. Statements are grouped into three families:

- DDL – data definition language
  - Creation (CREATE) of database table by defining the structure and the table columns
  - Modify or ALTER the table definition in the database
  - Remove or DROP the table from the database
  - Others like Schemas, Indexes, Views, Sequences, Triggers can also be managed using this subset
- DML – data manipulation language
  - “SELECT” data from the database including filtering, joining, and so on
  - “INSERT” data into the database
  - “UPDATE” data from the database
  - “DELETE” data from the database
  - Others like Truncate table data, Load, Unload tables into memory, and so on
- DCL – data control language
  - “CREATE”, “ALTER” and “REMOVE” users, user groups, roles in the database
  - “GRANT” or “REVOKE” data access in a table to a specific user or set of users
  - “GRANT” or “REVOKE” execute access on DDL / DML commands on the database to a specific user or set of users

- “COMMIT” and “ROLLBACK” of the changes made in the database
- Others like create, manage audit policies, remote source, and so on



**SQL building blocks**

<b>Data Types</b> <i>Binary, Boolean, Character, Datetime, LOB, Numeric etc.,</i>
<b>Operators</b> <i>Perform operations on the data. For e.g., Arithmetic, String, Comparison etc.,</i>
<b>Expressions</b> <i>Can be evaluated to return values. For e.g. CASE, Aggregate, SQL Functions etc.,</i>
<b>Predicates</b> <i>Specified by combining expressions, or logical operators. For e.g. LIKE, BETWEEN, CONTAINS etc.,</i>
<b>Functions</b> <i>Returns an output based on the manipulation on the data. For e.g., Convert data type, String manipulation etc.,</i>

Figure 3: SQL Building Blocks

## Data Types

The database can store many different data types data like below. Data type defines the characteristics of a data value. The data types are classified based on their characteristics, as shown in the following figure:



These data types are supported by SAP HANA SQL  
Notice the special SAP HANA data types (TEXT, ST\_Geometry, ...)

Classification	Data Type
Datetime types	DATE, TIME, SECONDDATE, TIMESTAMP
Numeric types	TINYINT, SMALLINT, INTEGER, BIGINT, SMALLDECIMAL, DECIMAL, REAL, DOUBLE
Boolean type	BOOLEAN
Character string types	VARCHAR, NVARCHAR, ALPHANUM, SHORTTEXT
Binary types	VARBINARY
Large Object types	BLOB, CLOB, NCLOB, TEXT
Multi-valued types	ARRAY
Spatial types	ST_Geometry, ST_Point

Figure 4: SAP HANA SQL Data Types

## Operators

Operators perform arithmetic, string, comparison operations in expressions.

Some of the examples are Unary operators like NOT, + (unary positive), - (unary negative), or Logical operators like NOT, OR, AND, or Comparison operators like ‘=’, ‘<’, ‘<=’ and so on.

## Expressions

An expression is a clause that can be evaluated over a set of inputs to return values as designed.

For example, CASE expression below will evaluate the input and return a value based on the input value.

```
CASE
WHEN 'APPLE'
THEN 'FRUITS'
WHEN 'CABBAGE'
THEN 'VEGETABLE'
ELSE 'UNKNOWN'
END
```

Just like CASE expressions, there are others like SQL built-in function expression (COUNT, MIN, MAX, SUM and so on).

## Predicates

Predicate is specified by combining one or more expressions, or logical operators, and returns one of the following logical values: TRUE, FALSE, or UNKNOWN.

An example is as follows:

```
SELECT * FROM <table_1> WHERE itemtype IN ('FRUIT', 'VEGETABLE');
```

## Functions

There are many built-in functions that are provided with SAP HANA SQL standard. They can be used to convert data from one data type to another, manipulation or return information of a string, or perform mathematical operations to return a value after manipulation.

```
TO_VARCHAR, TO_DATE, LENGTH, SUBSTR_AFTER, SQRT, etc.
```

## Comments

You can add comments to improve the readability and maintainability of your SQL statements. Comments are delimited in SQL statements as follows:

- Double hyphens "--". Everything after the double hyphen until the end of a line (not statement) is ignored by the SQL parser.
- "/\*" and "\*/". This style of commenting is used to wrap comments that appear on multiple lines. All text between the opening "/\*" and closing "\*/" is ignored by the SQL parser.



### Note:

If you are new to SQL, then we recommend that you now go to the APPENDIX of this course and study the basic SQL statements. It is important that everyone understands basic SQL before beginning the journey through SQLScript.

## What is SQLScript?

### SQLScript Extends Standard SQL

SQLScript is a set of extensions on top of standard SQL that employs the unique features of SAP HANA.



## What is SQLScript?

**Based on ANSI-92 SQL but adds extensions to exploit SAP HANA features:**

- Data Type Extensions
- Additional Security
- Logic Containers (procedures, functions)
- Implicitly and Explicitly Defined Table Variables
- Adds Imperative Logic
- Orchestration Logic to control both imperative and declarative statements

Figure 5: What is SQLScript?

By using these extensions, it allows much more pushdown of the data-intensive processing to the SAP HANA database, which otherwise would have to be done at the application level.

Applications benefit most from the potential of SAP HANA when they perform as many data-intensive computations in the database as possible. This avoids loading large amounts of data into an application server separate from SAP HANA, and leverages fast column operations, query optimization, and parallel execution. This can be achieved to a certain extent if the applications use advanced SQL statements, but sometimes you may want to push more logic into the database than possible using individual SQL statements, or make the logic more readable and maintainable.

SQLScript has been introduced to assist with this task.

### SQLScript Definition and Goal



- SQLScript is defined as follows:
  - The language to write stored procedures and user-defined functions in SAP HANA
  - An extension of ANSI SQL
- The main goal is to allow the execution of data-intensive calculations inside SAP HANA. This is helpful for the following reasons:
  - Eliminate data transfer between database and application tiers
  - Execute calculations in the database layer to benefit from fast column operations, query optimization, and parallel execution

### SQLScript Advantages

Compared to plain SQL, SQLScript provides the following advantages:



- Using SQLScript, complex logic can be broken down into smaller chunks of code. This encourages a modular programming style, which means better code reuse. SQL only allows the definition of SQL views to structure complex queries, and SQL views have no parameters.
- SQLScript supports local variables for staging intermediate results with implicitly-defined types. With standard SQL, it would be required to define globally visible views even for intermediate steps.

- SQLScript has flow control logic such as "if-then-else" clauses and looping "constructs" that are not available in SQL.
- Stored procedures can return multiple results with different tabular structures, while an SQL query returns only one result set.
- SQLScript supports input parameters so that code can be re-used in different scenarios by passing different parameter values. SQL does not support parameters and so SQL is usually limited in its use.

Pushing the processing to SAP HANA is a good thing because there are lots of opportunities for SAP HANA to optimize the execution with in-memory, parallel processing.

Standard SQL does not provide sufficient syntax to push many calculations to the database and, as a result, the application layer has to take on this duty. This means huge amounts of data must be copied between the database server and the application server.



- These primitive data types are supported by SQLScript
- SQLScript currently allows a length of 8,388,607 characters for the NVARCHAR and the VARCHAR data types, unlike SQL where the length of that data type is limited to 5,000.

Numeric types	TINYINT DECIMAL REAL	SMALLINT SMALDECIMAL DOUBLE	INT	BIGINT
Character String Types	VARCHAR NVARCHAR ALPHANUM			
Date-Time Types	TIMESTAMP SECONDDATE DATE TIME			
Binary Types	VARBINARY			
Large Object Types	CLOB NCLOB BLOB			
Spatial Types	ST_GEOOMETRY			
Boolean Type	BOOLEAN			



Figure 6: SQLScript Data Types



SQL	SQLScript
Offload very limited functionality into the database using SQL. Most of the application logic is normally executed on an application server	SQLScript uses the SQL extensions (for the SAP HANA database), allowing developers to push data-intensive logic to the database, better performance
SQL views cannot be parameterized, which limits their reuse	Re-usable
Do not have features to express business logic (for example a complex currency conversion)	Provide superior optimization possibilities
SQL query can only return one result at a time	Implement algorithms using a set-oriented paradigm and not using a one record at a time paradigm. (Imperative logic is required like iterative approximation algorithms)
SQL is a declarative language	It is possible to mix imperative constructs (procedural language) known from stored procedures with declarative ones

Figure 7: SQL vs. SQLScript

It is important to remember that SQLScript is not a full application programming language such as ABAP or C++. But SQLScript can significantly reduce the need to program data-intensive tasks in the application layer by providing imperative language in the database. Imperative language allows the developer to add very precise control flow logic, for example, to read tables one record at a time and process each record, then return to read the next record. Standard SQL does not allow this type of processing and only provides a way to write set-based logic that works on complete data sets from an instruction. That is why standard SQL needed to be extended with SQLScript to add more programmatic control in the database layer. This is especially necessary for transaction-based applications that operate at the record level.

### Declarative vs Imperative

SQL is a descriptive, or sometimes called declarative, language.



Declarative Logic	VS.	Imperative Logic
<p><b>What do you want to achieve?</b></p> <p><b>SQL is optimized automatically to achieve the best performance</b></p>		<p><b>How do you want to achieve?</b></p> <p><b>Programmer controls the execution flow, and taking over optimization, and not leaving with the optimizer</b></p>

Figure 8: SQLScript Adds Imperative Logic

SQLScript is written in logic containers such as **procedures** or **functions**. These are XS advanced source objects that are part of a complete SAP HANA application along with other related application components such as Java, HTML, and so on.

But the cost of using the SQLScript imperative language elements is that you potentially break the automatic optimization of SAP HANA. This is because you introduce dependencies into your logic. For example, before you can apply a discount, you first need to read through all customer sales records line by line and check if they are eligible records, and then sum the total sales to then look up a discount table based on spend amount. With SQLScript, you are able to take over the control of the logic flow to make sure each step happens in the right order. But you seriously limit the parallelization potential if queries that should otherwise be able to execute independently get driven by this type of strictly sequenced logic.



## LESSON SUMMARY

You should now be able to:

- Understand SQL
- Understand how SQLScript extends SQL

## Explaining SAP HANA XS Advanced and SAP HANA Deployment Infrastructure (HDI)



### LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Understand XS advanced and HDI

### What Is XS advanced and HDI?

#### XS Advanced



**SAP HANA XS Classic** is a web application server with a server-side JavaScript engine embedded in the database

**SAP HANA XS Advanced (and Cloud Foundry)** is a platform for microservices based polyglot web applications on premise (and in the cloud)

**SAP HANA Deployment Infrastructure (HDI)** provides a service that enables you to deploy database development artifacts to containers.

**SAP HANA HDI container** consists of a design-time container and a corresponding run-time container



Figure 9: XS Advanced

SAP HANA extended application services (SAP HANA XS) provide applications and application developers with access to the SAP HANA database using a consumption model that is exposed via HTTP.

SAP HANA functions as a comprehensive platform for the development and execution of native data-intensive applications that run efficiently in SAP HANA, taking advantage of its in-memory architecture and parallel execution capabilities.

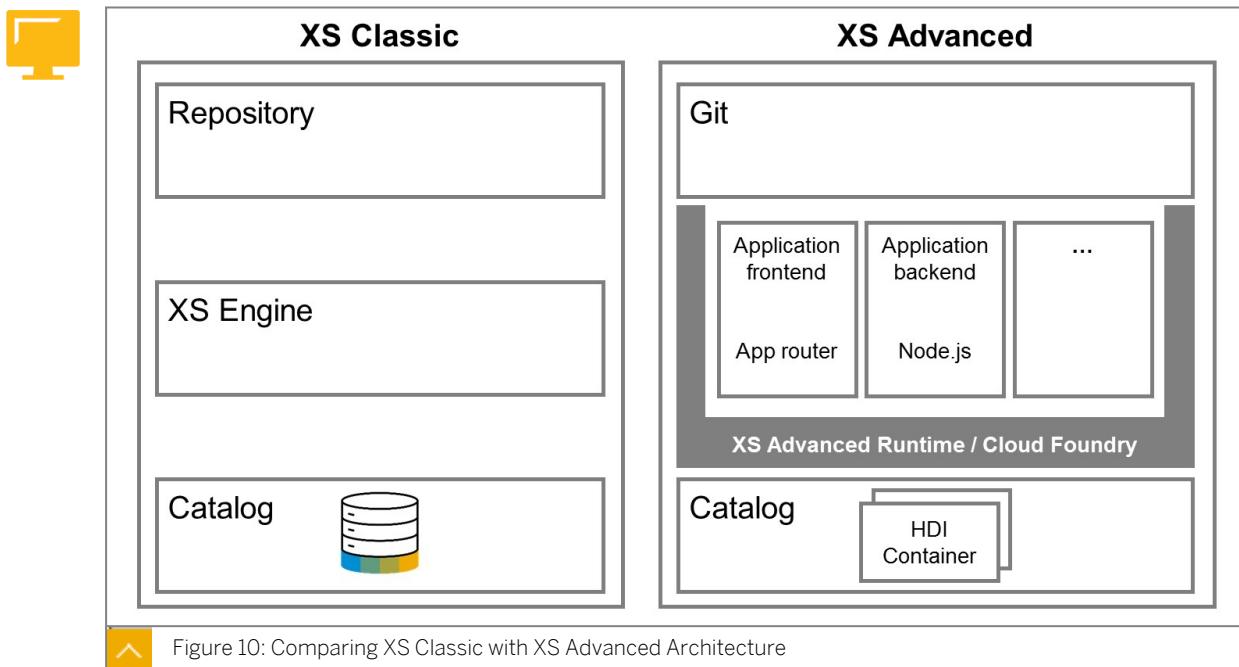


Figure 10: Comparing XS Classic with XS Advanced Architecture

Structured accordingly, applications can profit from the increased performance provided by SAP HANA due to the integration with the data source.

XS advanced is a polyglot application platform that supports several programming languages and execution environments, for example, Java and Node.js.

In comparison, the classic XS JavaScript (XSJS) is supported by a framework running in the Node.js run time, and uses the old SAP HANA repository in the SAP HANA database.

In simple terms, XS advanced is basically the Cloud Foundry open-source Platform-as-a-Service (PaaS) with many tweaks and extensions provided by SAP. These SAP enhancements include amongst other things:

- Integration with the SAP HANA database
- OData support
- Compatibility with XS classic model
- Improved application security

XS advanced also provides support for business applications that are composed of multiple micro-services, which are implemented as separate Cloud Foundry applications, and when combined are also known as multi-target applications (MTA). A multi-target application includes multiple so-called “modules” which are the equivalent of Cloud Foundry applications.

### HDI

The SAP HANA Deployment Infrastructure (HDI) provides a service that enables you to deploy database development artifacts to so-called containers. This service includes a family of consistent design-time artifacts for all key SAP HANA platform database features which describe the target (runtime) state of SAP HANA database artifacts, for example: tables, views, or procedures. These artifacts are modeled, staged (uploaded), built, and deployed into SAP HANA.

An SAP HANA HDI container consists of a design-time container and a corresponding runtime container.

SAP HANA HDI uses containers to store design-time artifacts and the corresponding deployed runtime (catalog) objects. The SAP HANA Deployment Infrastructure (HDI) strictly separates design-time and runtime objects by introducing the following distinct container types:

- Design-time container – An isolated environment for design-time files
- Runtime container – Stores the deployed objects built according to the specification stored in the corresponding design-time artifacts

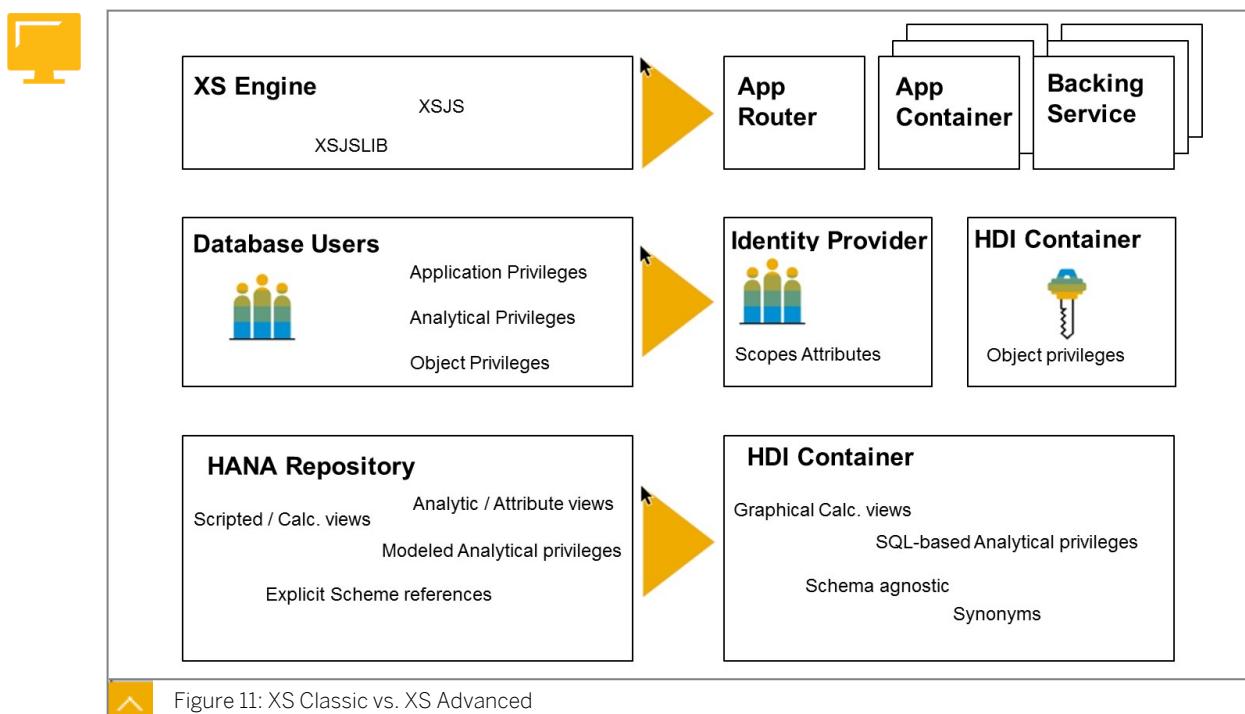


#### Note:

HDI uses so-called Build Plug-ins to create runtime objects from the corresponding design-time files, for example, a procedure object in the database catalog from a design-time definition of a stored procedure.

### XS Classic vs. XS Advanced

SAP HANA XS classic model enables you to create database schema, tables, views, and sequences as design-time files in the SAP HANA repository. Repository files can be read by applications that you develop.



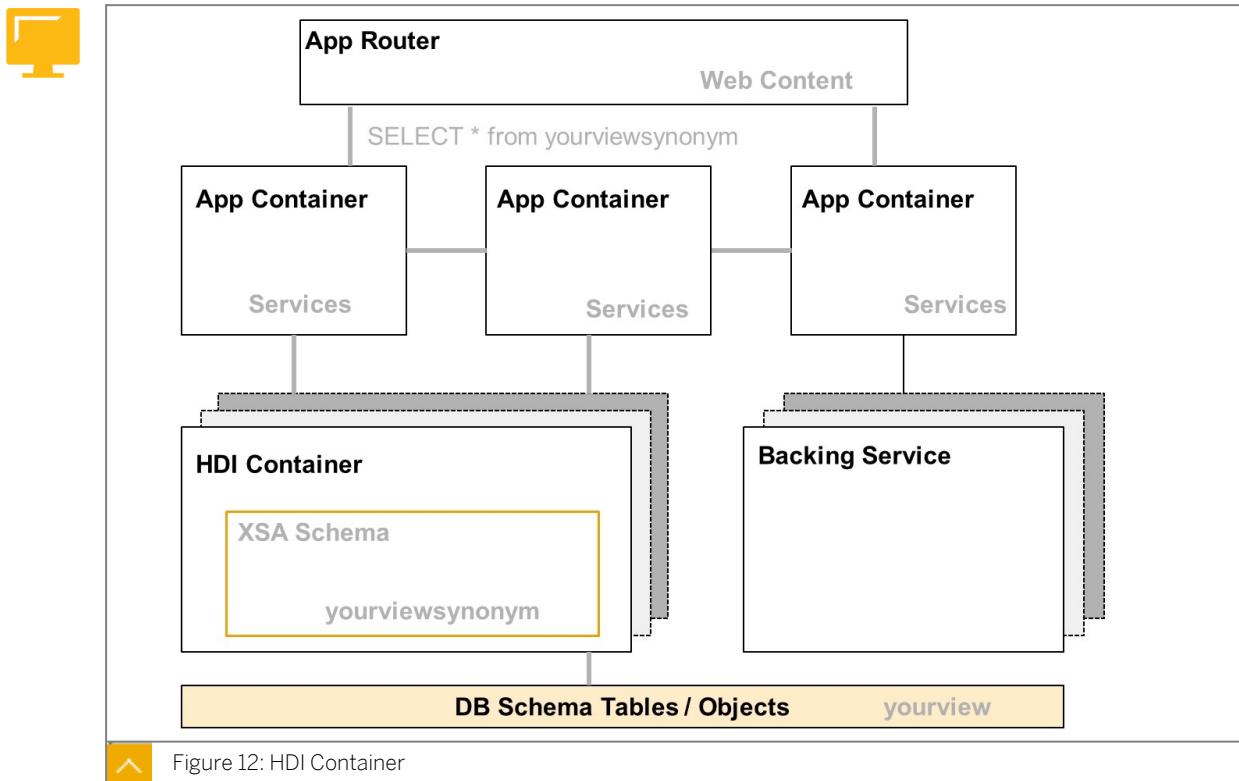
In XS advanced, application development takes place in the context of a project. The project brings together individual applications in a so-called Multi-Target Application (MTA), which includes a module in which you define and store the database objects required by your data model.

Basically, your project allows you to do the following:

- Define the data model as design time object
- Configure the SAP HANA deployment infrastructure

- Deploy and generate the run time objects (active objects) in the database catalog
- Consume the model

### Components of XS Advanced



### HDI Container

HDI containers are a special type of database schemata that manage their contained database objects. These objects are described 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.

### Synonyms

Synonyms in XS advanced enable access to objects that are not in the same schema or application container. Synonyms are most commonly used to hide the real object names from consumers or to give a database object a more convenient name.

Synonyms play a more important role in XS advanced and HDI than they do in the schemas used in XS classic. In XS advanced, using synonyms is the designated method to enable access to objects in other schemas, for example, a schema owned by another XS advanced application container, provided the access is granted by the other container.

### Backing Services

Backing services provide the technology on which XS advanced application are built, for example:

- Persistence services (databases, key-value stores, ...)
- Communication services (message queues, e-mail gateways, ...)

- Authentication services (User Account and Authentication service (UAA))

### Application Containers

During application deployment, the build pack ensures that this run time is provided to the application and that the appropriate data sources for the SAP HANA HDI container are bound to the corresponding application container.

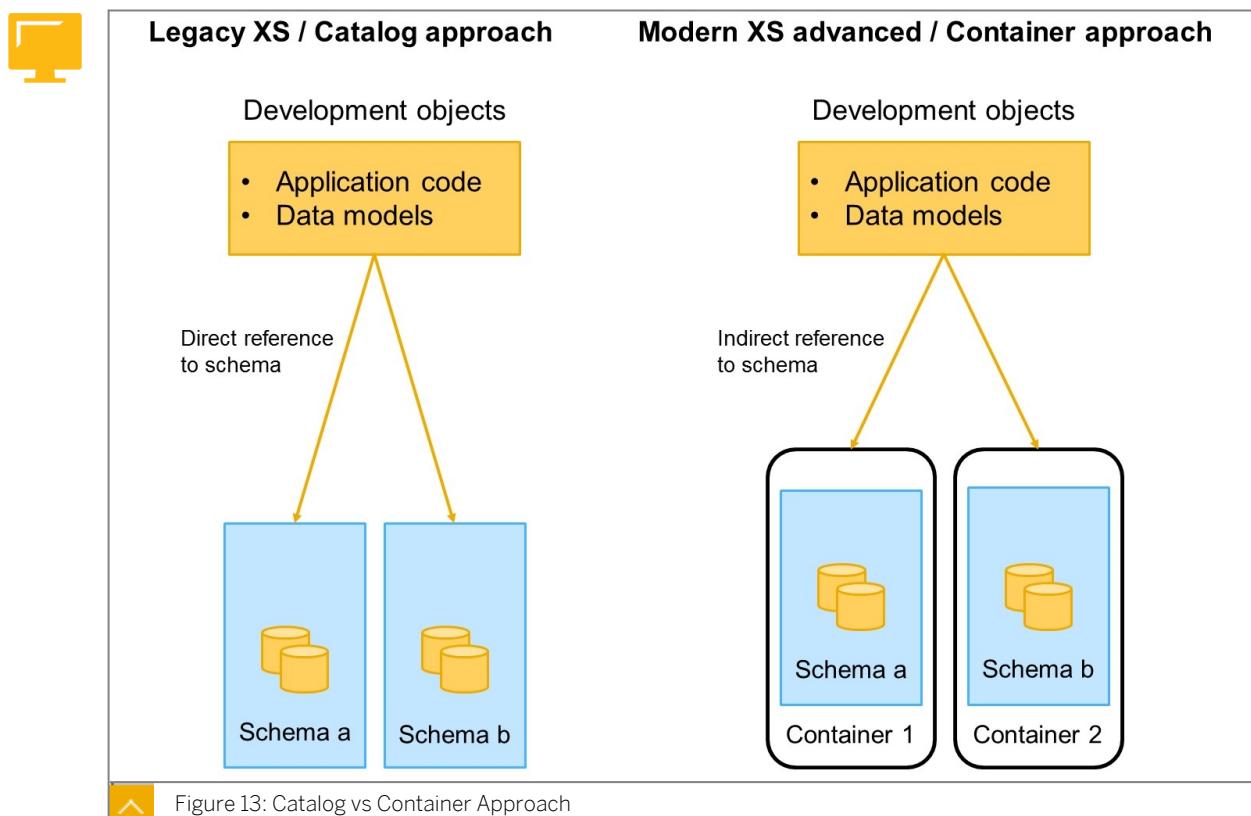
### Application Router

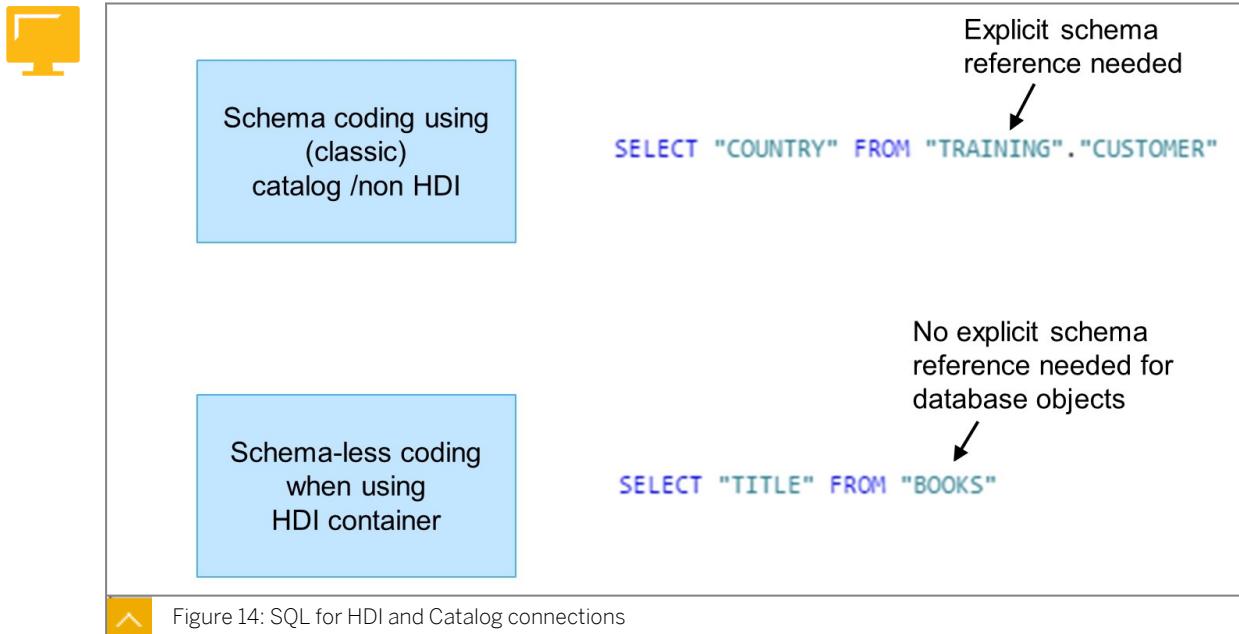
The application router is the single-entry point for the (business) application and receives all the requests. It serves static content, propagate and authenticates users, rewrites URLs, and routes proxy requests to other micro services, as well as propagating user information.

### Comparing XS Classic / Repository to XS Advanced / HDI

SQLScript is relevant for both the XS Class / Repository approach (essentially SAP HANA 1.0) and also XS Advanced / HDI approach (essentially SAP HANA 2.0).

You should not mix development across both approaches and SAP recommends using XS advanced / HDI, as the XS Classic / Repository approach is officially deprecated and will eventually disappear from the software.





When using containers, you do not explicitly reference a schema name when referring to database object in your SQLScript. As long as the development objects are in the same container, they will find each other. Only when you need to reference database objects in other external containers or catalog schemas will you need to take action. In that case, you first need to build synonyms in your container that point to the external database object you need access to.



### LESSON SUMMARY

You should now be able to:

- Understand XS advanced and HDI

# Understanding HDI and Working with the Database Explorer



## LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Understand HDI and work with the Database Explorer

## SAP HANA Deployment Infrastructure

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 XS advanced 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 that there is no design-time object for this table. So, upon deployment of the container, this table will not be recreated.



### Note:

This is very different from other types of applications that use different persistency frameworks that rely on plain schemas. These frameworks can directly execute DDL statements.

## 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*.



```
{
    "plugin_version" : "2.0.20.0", // optional, defaults to 0.0.0.0
    "file_suffixes" : {

        "hdbsynonym" : {
            "plugin_name" : "com.sap.hana.di.synonym"
        },
        "hdbsynonymconfig" : {
            "plugin_name" : "com.sap.hana.di.synonym.config"
        },
        "hdbtable" : {
            "plugin_name" : "com.sap.hana.di.table"
        },
        "hdbdropcreatetable" : {
            "plugin_name" : "com.sap.hana.di.dropcreatetable"
        },
        "hdbcalculationview" : {
            "plugin_name" : "com.sap.hana.di.calculationview"
        },
        "hdbcds" : {
            "plugin_name" : "com.sap.hana.di.cds",
            "plugin_version": "2.0.19.0"
        }
    }
}
```

Version of plug-in to use for all file types unless they have their own specification

Suffix of source file

Plug-in name and path

This file type uses an older plug-in version



Figure 15: 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 (choose 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 files 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 need 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, "2.0.40.0"

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 SAP 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 that 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, it 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 Service Broker Security Support (SBSS) 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 contents 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 SAP HANA Deployment Infrastructure, to show the database objects and the content of tables, execute queries in a SQL console, and so on.



### LESSON SUMMARY

You should now be able to:

- Understand HDI and work with the Database Explorer

## Describing the SAP HANA Database Module

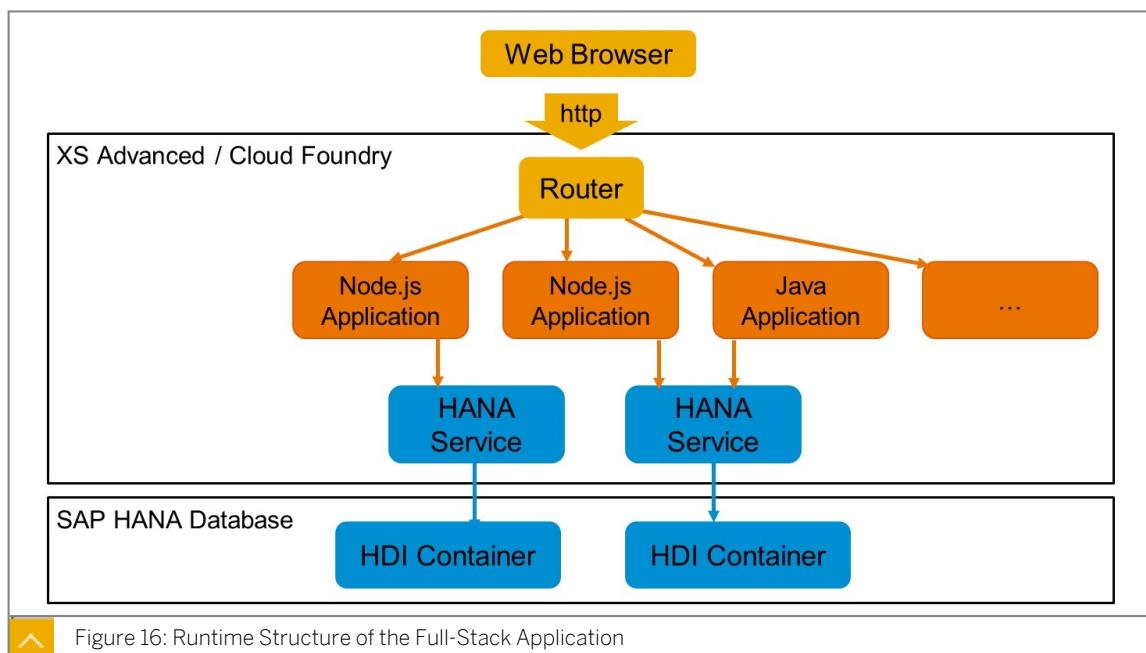


### LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Describe the SAP HANA Database Module

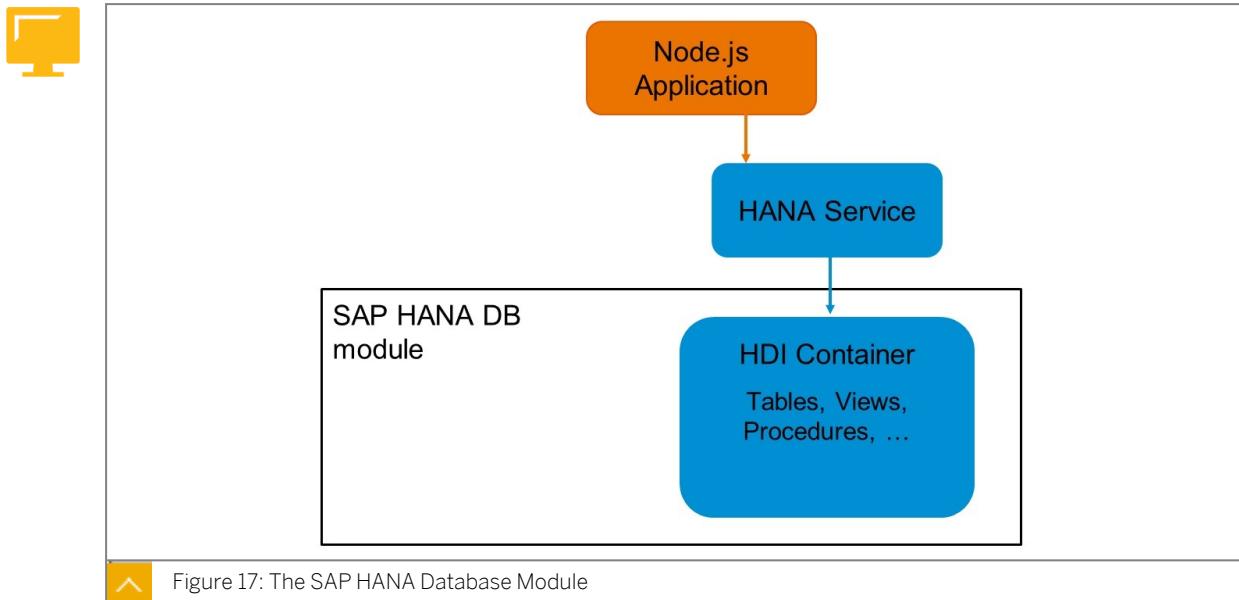
### The SAP HANA Database Module



At runtime, the full-stack MTA is commonly made in the following way:

1. The user interface is written using HTML5 and rendered/executed in a web browser.
2. The HTML5 program communicates with the router via the HTTP protocol. The router is a program running on the server. It does not execute business logic, it is just a dispatcher of HTTP messages.
3. The router dispatches the HTTP message to the proper application, running on the server. This application is built out of the corresponding Node.js (Java, Python, and so on) module, coded by the developer within the MTA project in the SAP Web IDE for SAP HANA.
4. The Node.js application communicates with the database via a SAP HANA service, that is connected to a SAP HANA Deployment Infrastructure (HDI) container stored in the database.
5. Database objects (tables, views, procedures, and so on) are stored in the HDI container.

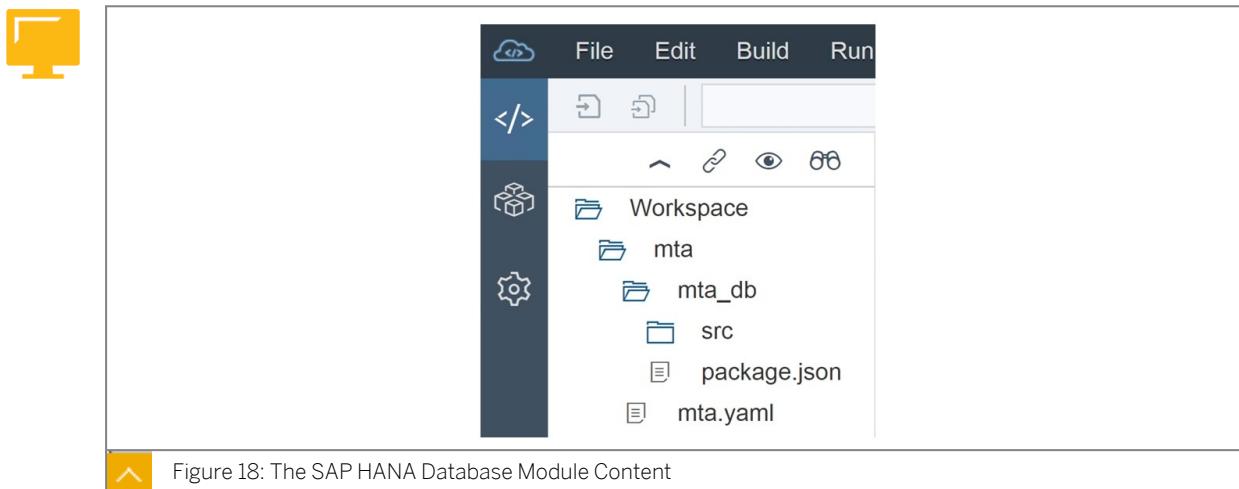
6. The procedures in the database use the SQLScript (alternatively the R) language, that is specialized in highly parallel and data-intensive processing.



At design time, in the SAP Web IDE for SAP HANA, an MTA project is created.

The database content of the MTA is defined using the SAP HANA database module.

This module contains the design time definitions of all the objects to be created in the database, for example, tables and views, calculation views, procedures, and so on.



A minimal SAP HANA database module contains the following:

- An src folder, made to store the design-time definitions of the database objects.
- A minimal package.json file for configuration, for example, to set the version and the options of the deploy program.

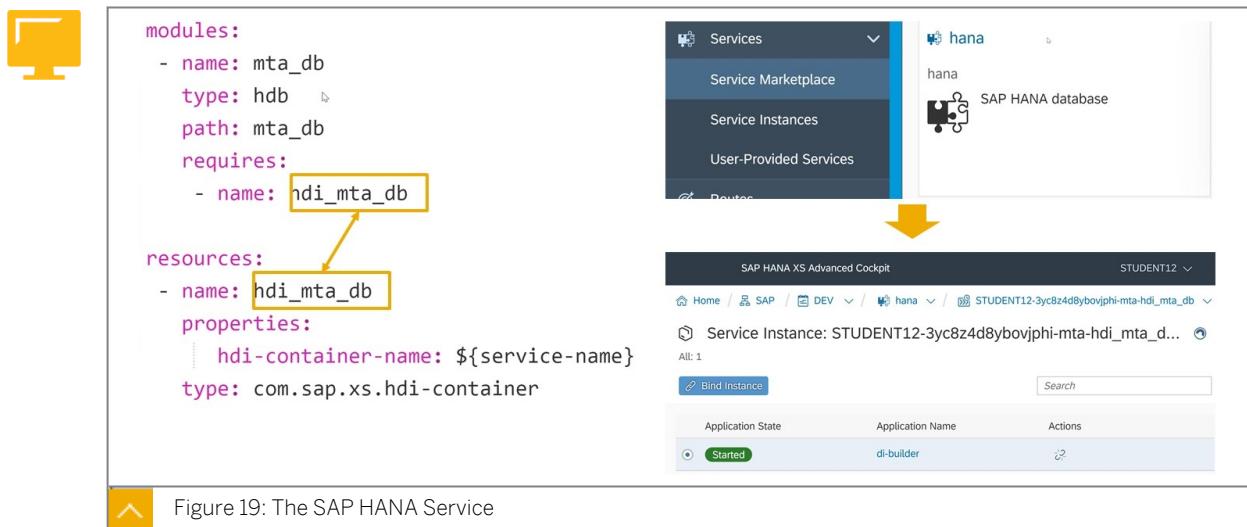


Figure 19: The SAP HANA Service

A service of type HANA (technical name com.sap.xs.hdi-container) needs to exist for the HDI container to be accessible from XS advanced / Cloud Foundry. When you create a new SAP HANA database module, a service is also added by default to the mta.yaml file, together with the configuration required for the service to be accessible by the module.

When the module is built, a service instance (of type HANA) is created in XS advanced. This service instance is connected to the HDI container that is created in the SAP HANA database.



## LESSON SUMMARY

You should now be able to:

- Describe the SAP HANA Database Module



# Unit 1

## Lesson 5

# Working with the SAP Web IDE for SAP HANA



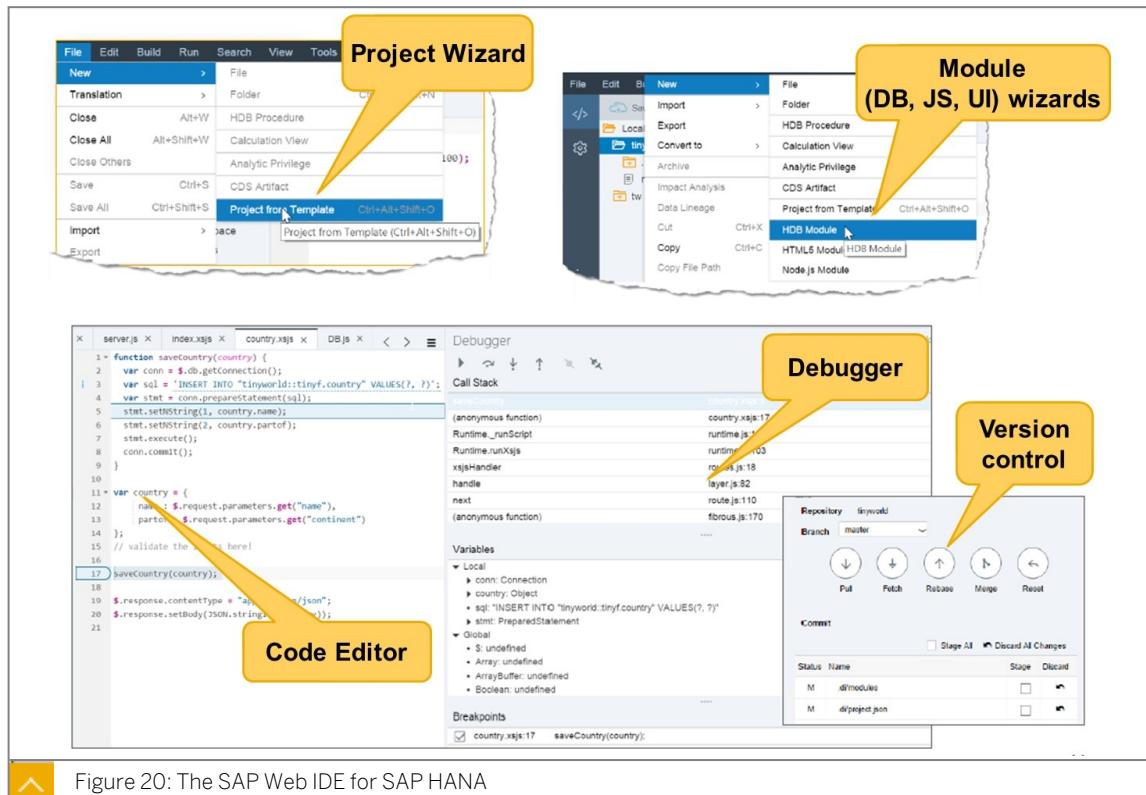
## LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Introduce the SAP Web IDE for SAP HANA
- Introduce the SQL Console of SAP Web IDE for SAP HANA

## What is the SAP Web IDE for SAP HANA?

The SAP Web IDE for SAP HANA is a browser-based software development tool and is a key component supplied with SAP HANA.



The SAP Web IDE is aimed at developers who build SAP HANA applications using a variety of development languages. This includes SQL and SQLScript.

**Caution:**

Do not confuse SAP Web IDE for SAP HANA with SAP Web IDE. SAP Web IDE is used to build SAP Fiori and SAPUI5 based applications. SAP Web IDE for SAP HANA is used to build XS advanced applications on SAP HANA.

The predecessor to SAP Web IDE for SAP HANA (we'll just call it SAP Web IDE from now on), is SAP HANA Studio. It is very important to remember that any developments using SAP HANA Studio do not create XS advanced applications. They create XS Classic applications. Only SAP Web IDE creates XS advanced applications and should be used for all developments going forward.

**Note:**

SAP provides migration tooling to move XS Classic applications to XS advanced.

SAP Web IDE includes many tools for development, including the following:

- Project Wizard – Used to set up a development project which is required as the highest level wrapper of all source objects.
- Module Wizards – Used to create the language-specific folders that will contain the source files of a project, such as Java, HTML and HDB.
- Code Editor – Used to write the code and includes language-specific syntax checking.
- Code Version Control – Used to present the Git interface with SAP Web IDE to manage code branching, commits and fetches, and so on.
- Debugger – Language-specific code debugging tools.

When working with SAP Web IDE, you need to decide which view you need to use. There are two views, as follows:

- Development View
- Database Explorer View

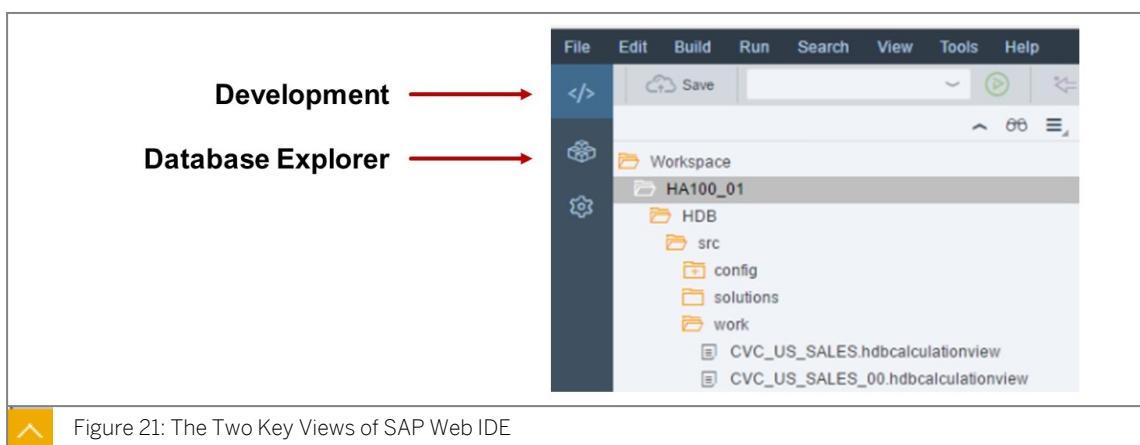


Figure 21: The Two Key Views of SAP Web IDE

The Development view (shown in the figure) is used to develop the source code in files. The files are organized into folders. You can freely define as many folders and sub-folders as you wish.

The top-level folder (under **Workspace**) is called the **project**. The project is defined using a SAP Web IDE wizard.

Within the project, you then create development **modules**. Each module defines the type of objects allowed, such as Java and HTML. Every development project will need an **HDB** (SAP HANA database) module. The HDB module is where all SQL and SQLScript source code is written and stored. Each source file has an extension. You will write your SQLScript in source files with one of these extensions:

- .hdbfunction
- .hdbprocedure

The Database Explorer view provides access to the run-time database objects, such as tables, views and procedures.

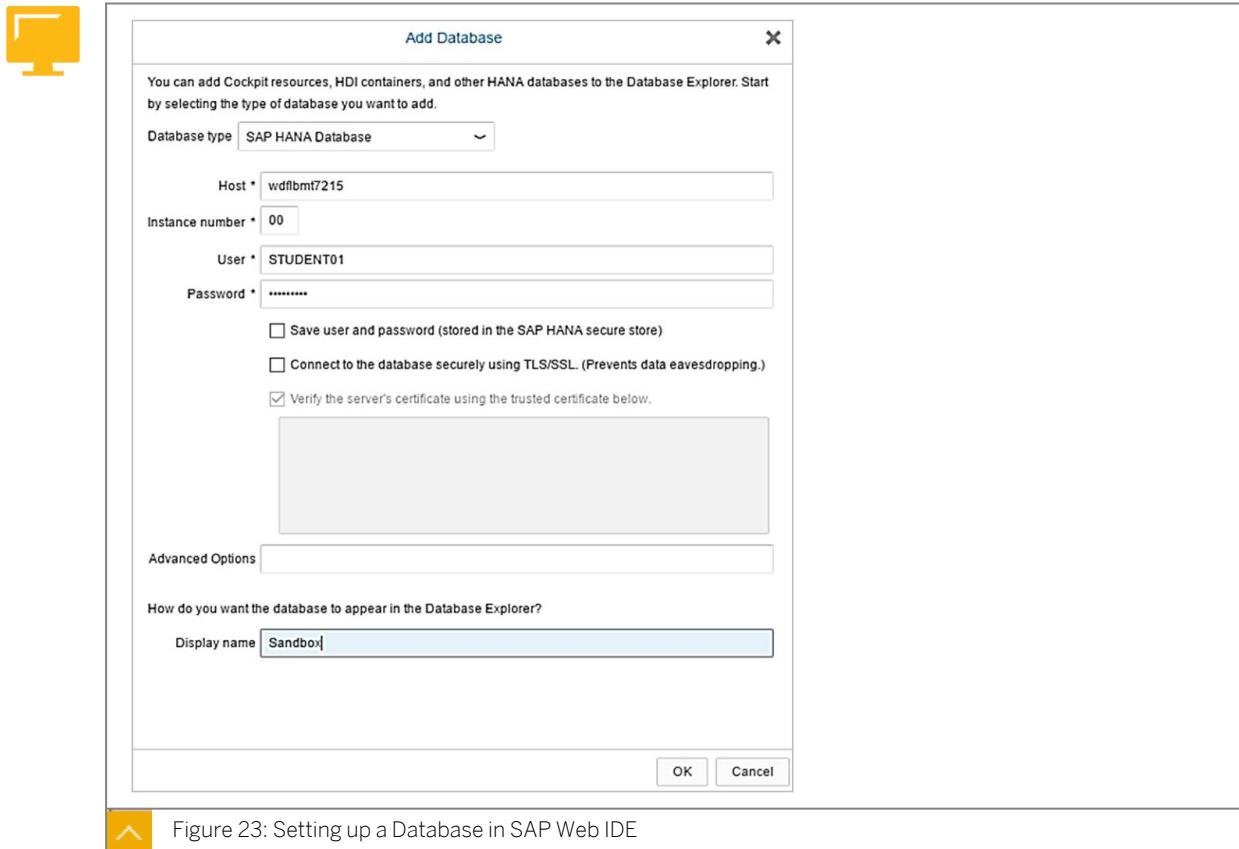
Figure 22: Database Explorer of SAP Web IDE

There are two options for displaying the database objects, as follows:

- Catalog / schema view
- Container view

You can chose to show the objects in either view, or even in both views side by side. When working with XS advanced development, you will work with containers, and so this is the most useful view for you. However, it might also be helpful to see the same database objects presented in their native schemas. Having access to schemas is also helpful especially if you plan to access objects that are in external schemas and want to first examine them before creating synonyms that point to them from your container.

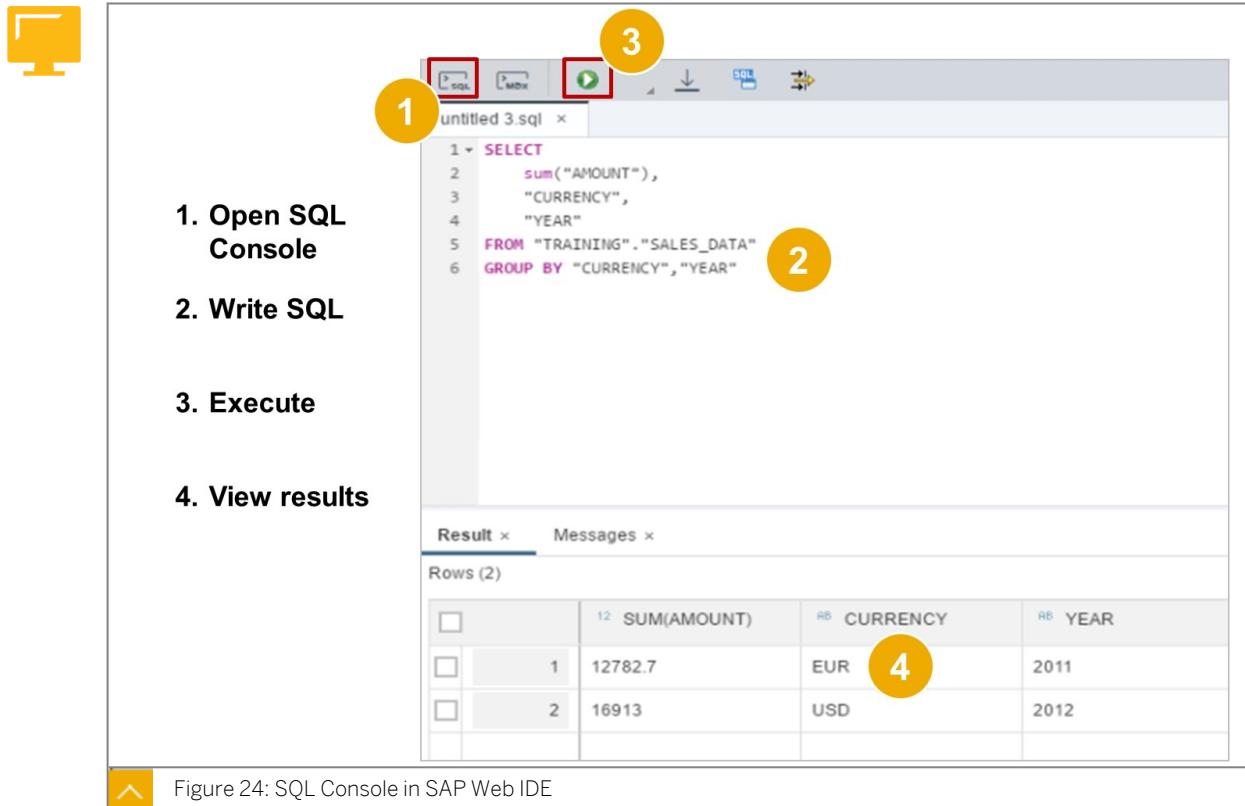
Before you can view the database objects in the Database Explorer, you must first create a database connection.



The most important parameter is the first one: Database Type.

This is where you specify whether you require the container or catalog / schema type of view.

In the Database Explorer you will find the **SQL Console**.



The SQL Console can be launched by right-clicking directly on a database object, or you can open the SQL Console from the toolbar.

If you open the SQL Console from the database object, it places the selected object in the correct part of the SQL statement. For example, if you launch the console by right-clicking on a table, the table is placed after the *FROM* in the SQL statement with all columns selected.

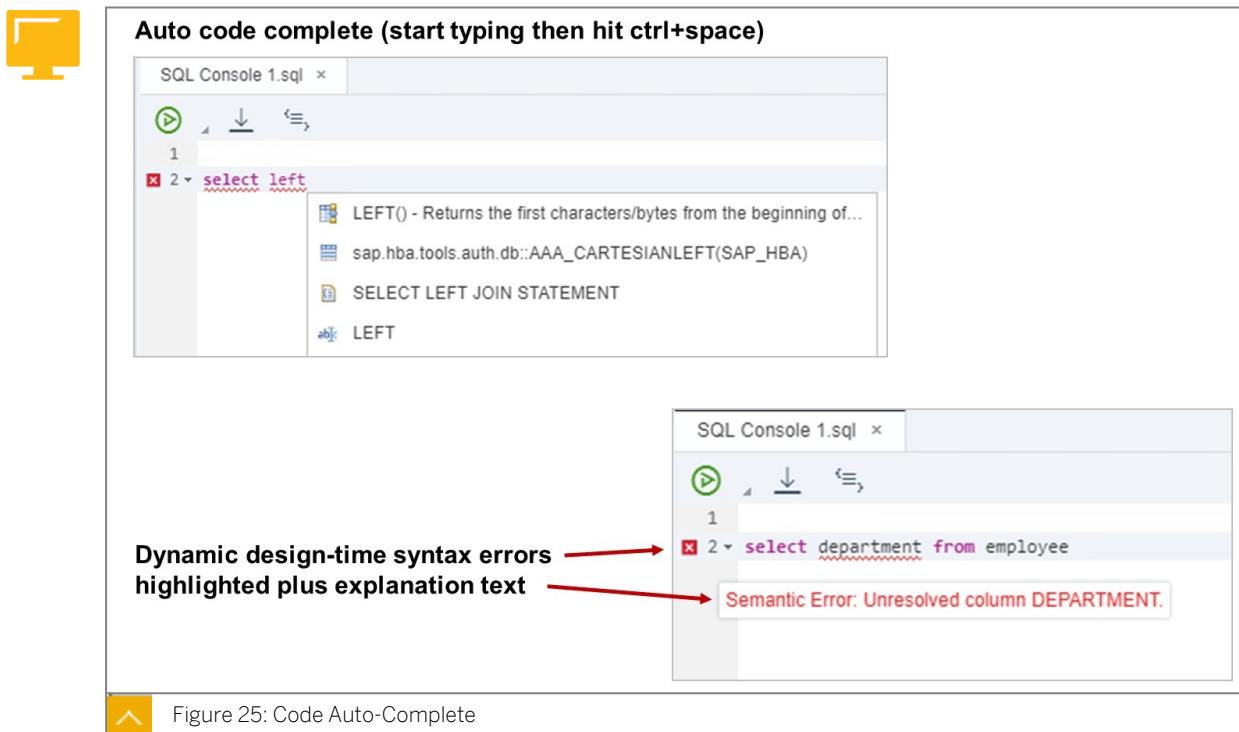
If you selected a procedure, the SQL console launches and the selected procedure is placed after the *CALL* followed by its empty parameters. You can even have the parameters prompted in a tidy dialog pane using the menu option *Call Procedure and Prompt for Values*.

The results of each SQL query is displayed beneath the code and you can display multiple outputs at any time side by side.

Before you execute your SQL statements, make sure that the SQL Console is connected to the correct database. Use the buttons in the toolbar to switch database if needed. The SQL statement is executed against the selected database. If you launched the SQL Console directly from a database object, then the correct database will already be selected. The connection is always displayed in the header area of the SQL Console so don't forget to check this.

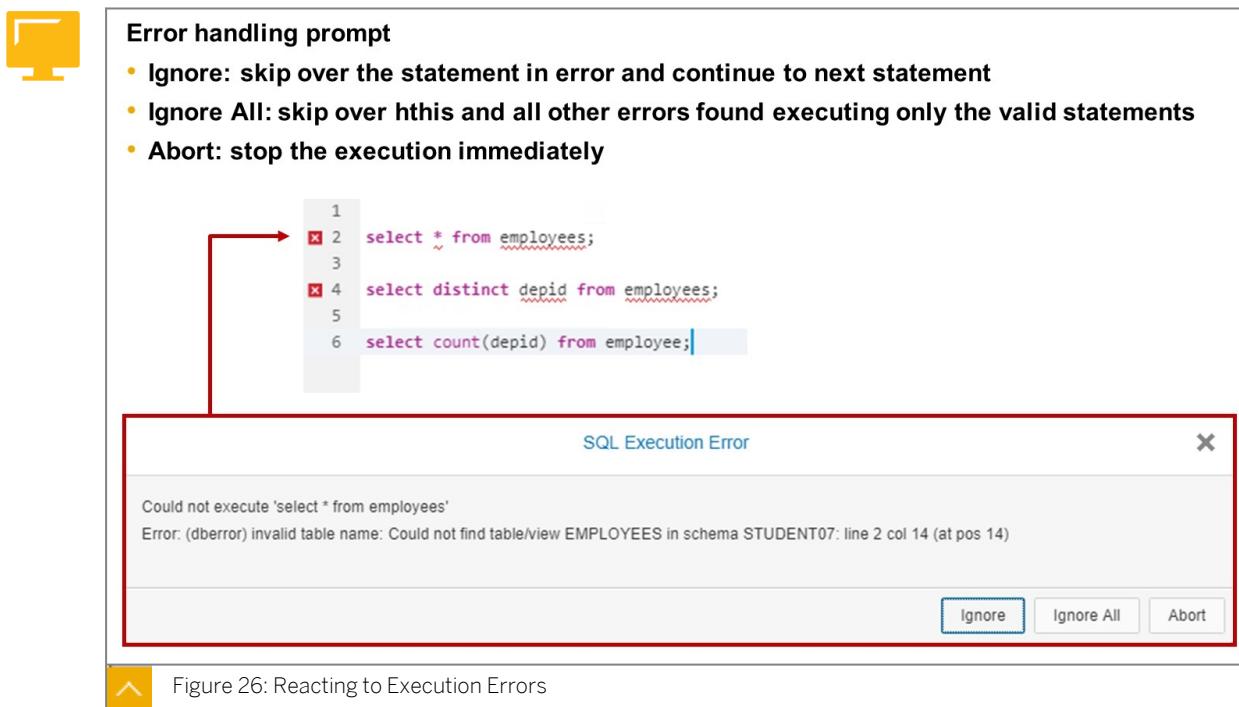
### The SQL Console of Web IDE Database Explorer

There are a number of tools and aids provided with the Web IDE SQL Console that you should familiarize yourself with so that you can improve your productivity when writing SQL and SQLScript.



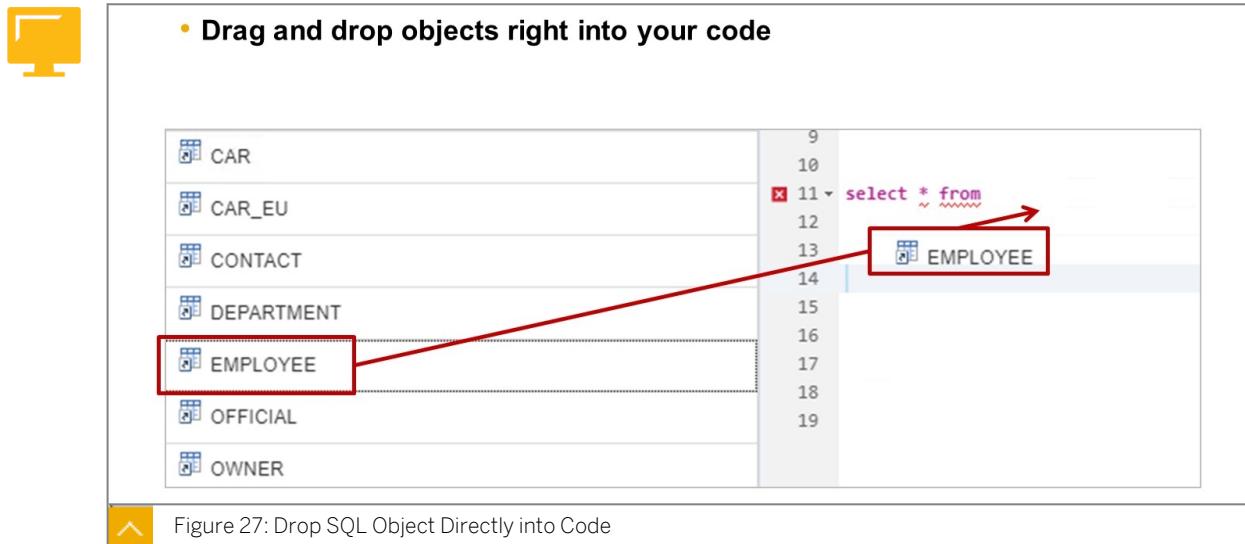
Instead of trying to remember the full syntax of common SQL statements and functions, simply begin typing and then hit **ctrl+space** anytime, and a dropdown list appears where you can select your entry. The remaining code is then added to the console. You simply insert your parameters in the placeholders.

As you enter your SQL code, the SQL console will highlight errors immediately. Look out for the red cross to the left of the code to know which lines have errors. Common errors include using invalid table name or column names, and syntax errors such as missing parentheses. If you see the red cross, also look in the code line to see which part is underlined in red. Finally, hover over the error to see the explanation text.

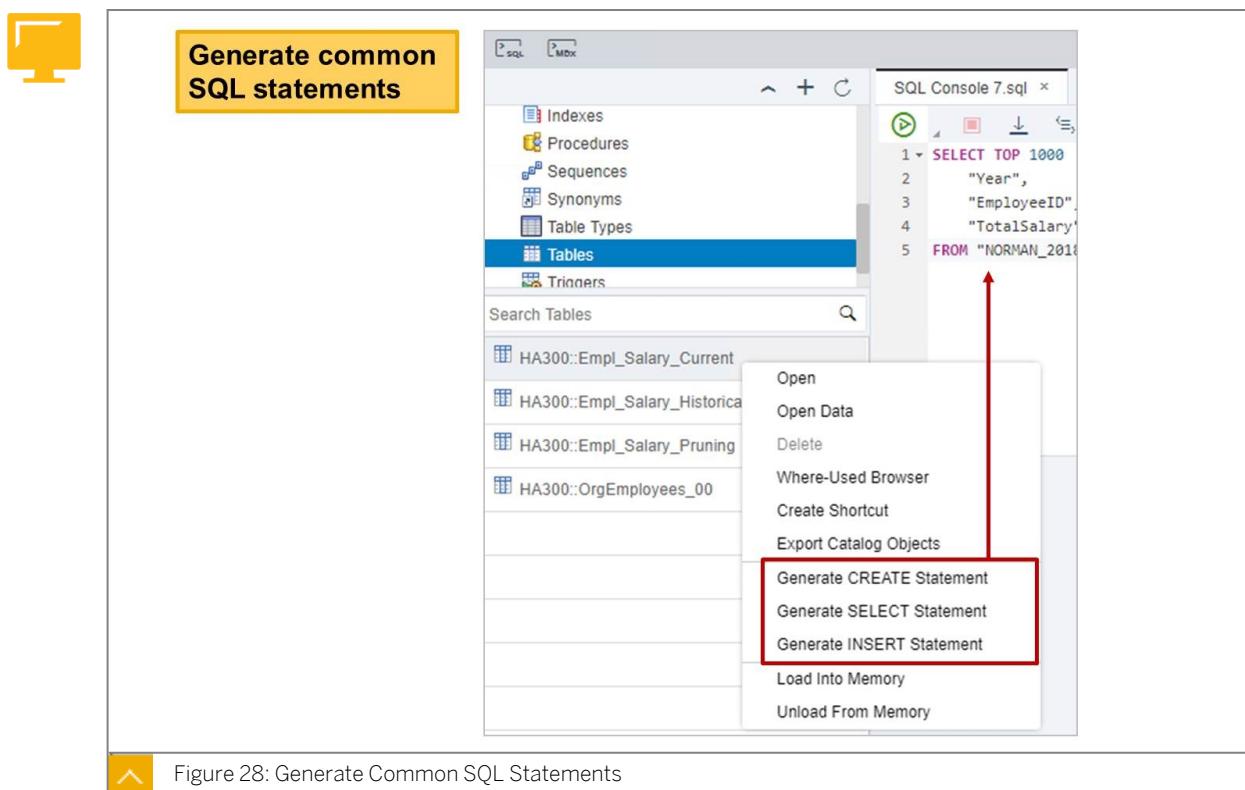


If errors are found during execution of your SQL code when there are multiple statements, a pop-up appears and you can decide how to proceed.

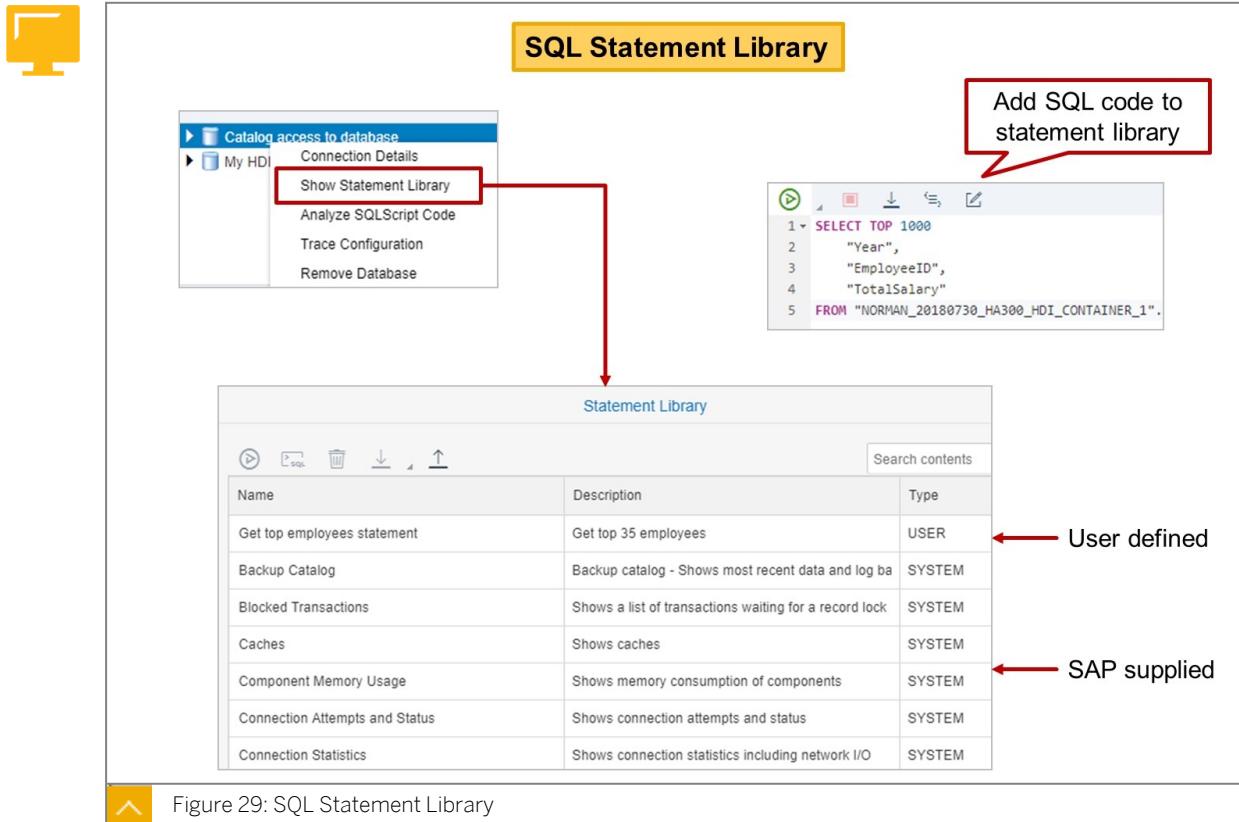
You can choose to skip the statement in error and try the next statement. You can choose to skip all other statements in error (in other words, do not keep popping up for each error). You can also decide to stop the execution immediately.



Instead of typing each table name, function, or procedure into the SQL console, you can simply drag the object and drop it directly into the code. As well as being simpler, it also eliminates errors because all delimiters are added (quotes) and fully qualified names including namespaces used in HDI artifacts.



You can generate common SQL statements by simply clicking on the SQL object name in the left pane. An SQL Console will automatically open with the corresponding database connection. In the console, the requested SQL code will appear, ready to modify or execute.



SAP supply a library of SQL code that can be used to monitor the SAP HANA database. This is called the **SQL Statement Library** and is a feature of the SAP Web IDE for SAP HANA.



#### Note:

Be careful not to confuse the *SQL Statement Library* with the *User Defined Library*. The latter is a library that contains procedures and functions that belong together.

Here you will find prepared queries that you will find useful.

You can also add your own statements to the library. When you add your own statement, you will be asked if you would like to add a short description to the code.

A *DESCRIPTION* tag is automatically added to the top of your code and you can enter some words. This will help when other developers find your code and want to understand its purpose.

```
/*
[DESCRIPTION]
This is where you add your own description (don't touch the tag above!)
*/
```

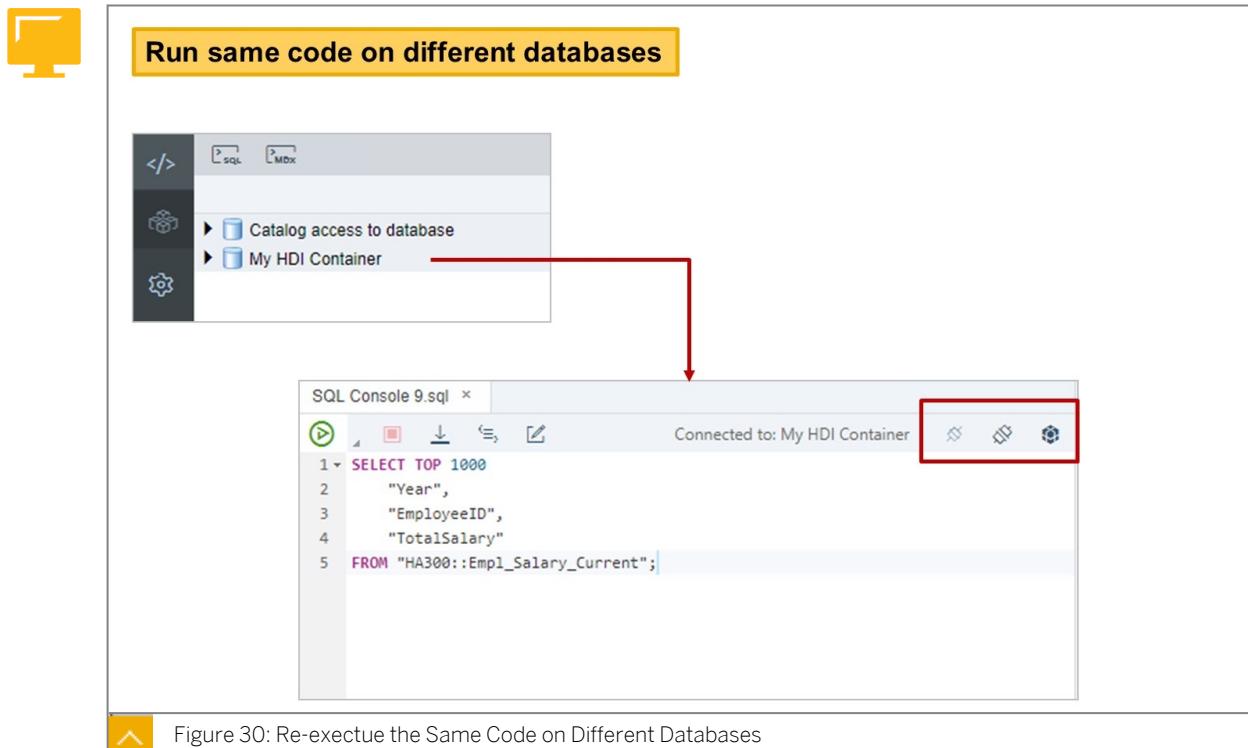


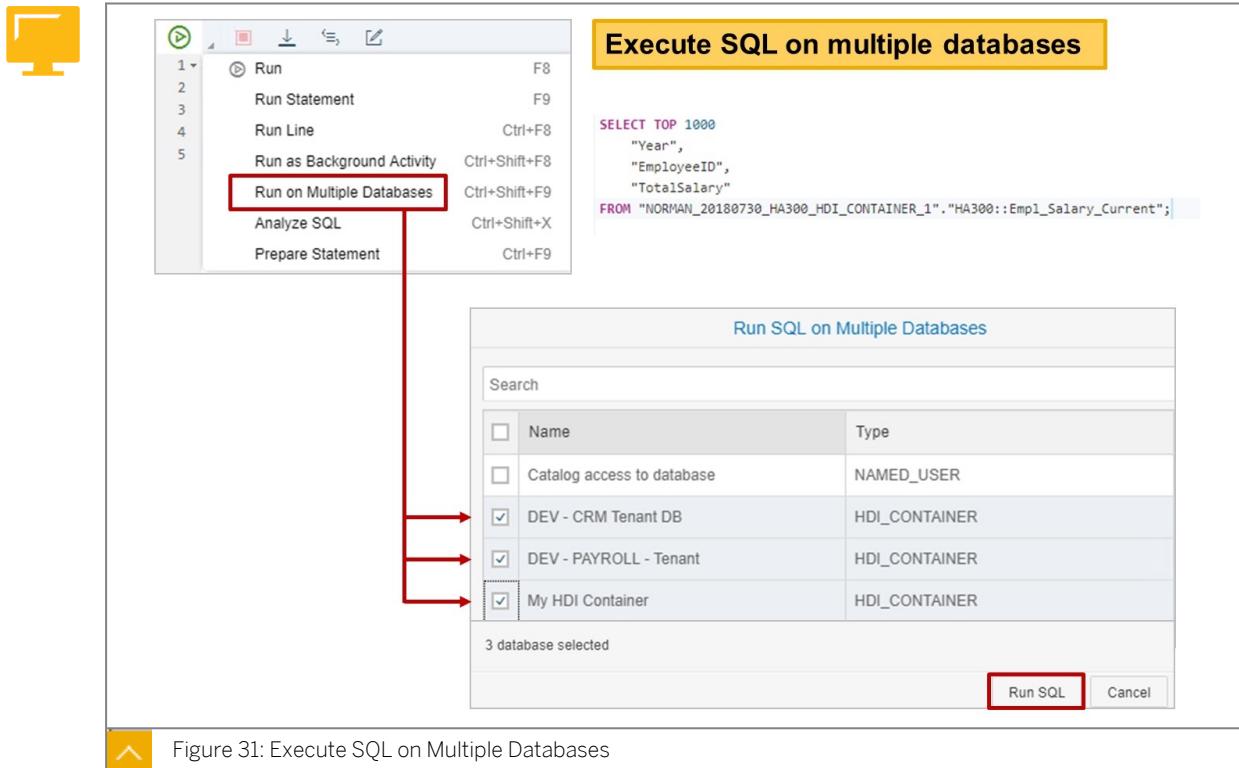
Figure 30: Re-execute the Same Code on Different Databases

When you open an SQL Console in SAP Web IDE, you need to specify one of the database connections that you defined earlier so that the code you write is executed against that database. There might be times when you want to swap the database connection so that the same code can be executed on a different database.

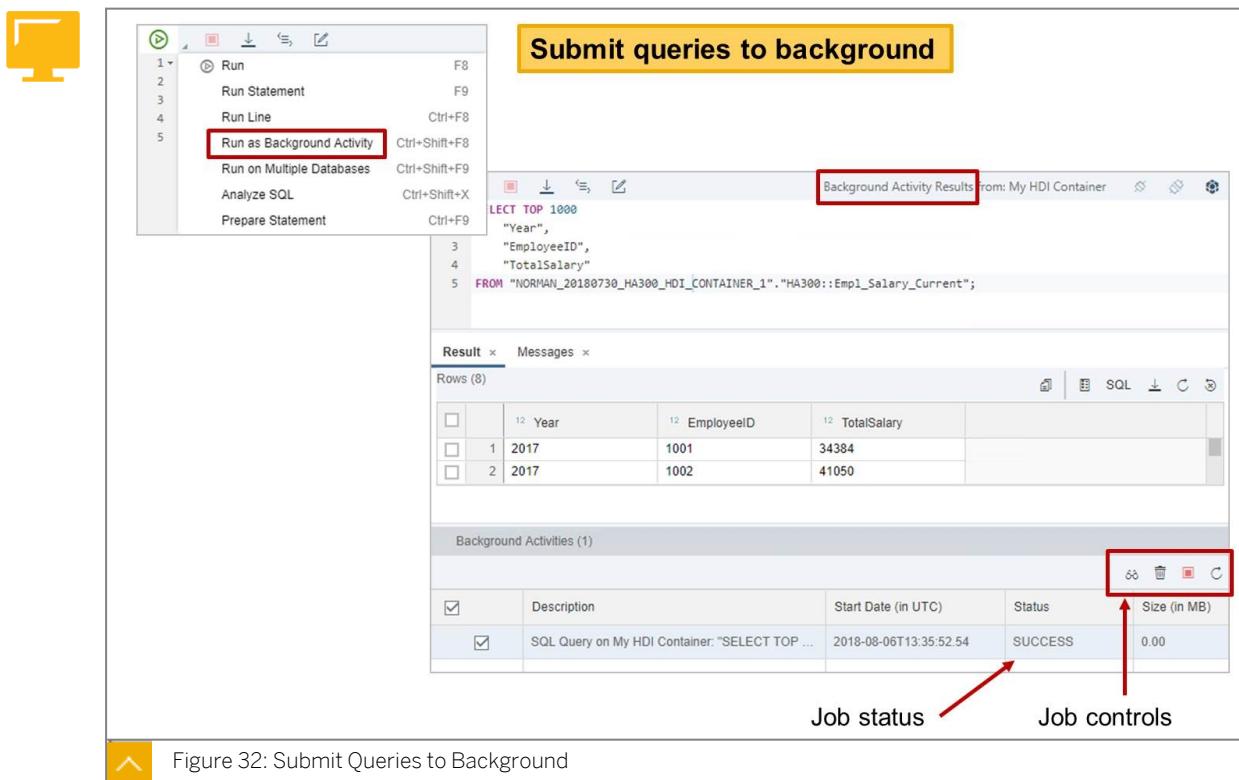


Hint:

Always check the name of the database you are connected to before executing your code, especially if you frequently jump between database connections.



It is possible to execute your SQL code on multiple databases at the same time. For example, you may wish to execute a statement from the SAP supplied library, in order to monitor the database across multiple tenants. Each tenant would be a separate connection. Simply choose the connections from the pop-up and execute the code against all selected database connections.



A great feature of SAP Web IDE is that you can execute queries in the background. This means you submit a query and you can then disconnect from the database and close your SAP Web IDE interface. The query will still run in the background and its results will be waiting for you in the **Background Activity Monitor** the next time you connect to the Database Explorer.



### LESSON SUMMARY

You should now be able to:

- Introduce the SAP Web IDE for SAP HANA
- Introduce the SQL Console of SAP Web IDE for SAP HANA



# Unit 1

## Lesson 6

# Understanding the Course Data

## LESSON OVERVIEW

This lesson explains the sample data models and data used throughout lessons and exercises of this course.



## LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Understand the sample database used throughout the course

## The Sample Data Model

The examples used in the lessons and on the figures of this course are based on a fictional car registration office.

### Example: Registration Office

- The database of a fictional registration office serves as the basis for further explanations.
- The tables in this database have been specifically tailored to the SQL course and are not an example of good database design.
- The officials working in the fictional registration office have a manager.
- Each vehicle is registered for exactly one owner (or is unregistered).
- There is a list of vehicles that have been reported stolen.
- Owners, who have at least three vehicles registered, are assigned to one or multiple contacts.

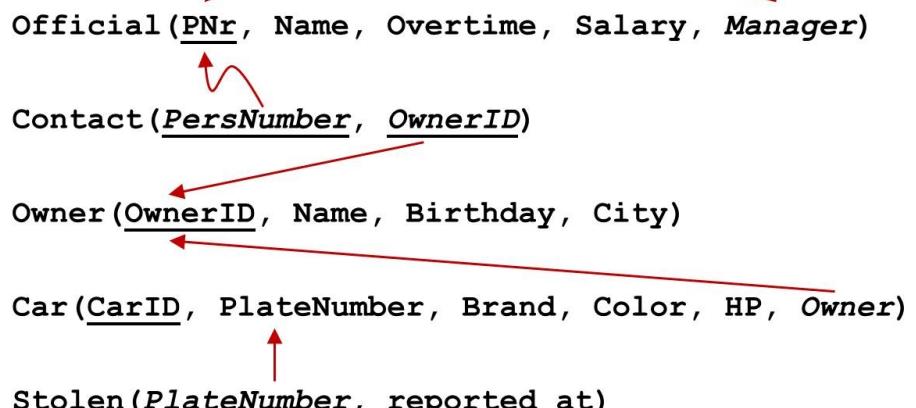


Figure 33: Registration Office Example



Table 1: The Official Table

The following shows the content of the *Official* database table as an example.

Official	PNr	Name	Overtime	Salary	Manager
	P01	Mr A	10	A09	P04
	P02	Mr B	10	A10	P04
	P03	Ms C	20	A09	P04
	P04	Ms D	NULL	A12	P09
	P05	Mr E	10	A08	P08
	P06	Mr F	18	A09	P09
	P07	Ms G	22	A11	P08
	P08	Ms H	NULL	A13	P09
	P09	Mr I	NULL	A14	NULL



Table 2: The Owner Table

The *Owner* database table contains the information about the car owners. Note that owners who are people have a birthday, but owners that are companies do not.

Owner	OwnerID	Name	Birthday	City
	H01	Ms T	20.06.1934	Wiesloch
	H02	Ms U	11.05.1966	Hockenheim
	H03	SAP AG	NULL	Walldorf
	H04	HDM AG	NULL	Heidelberg
	H05	Mr V	21.04.1952	Leimen
	H06	Ms W	01.06.1957	Wiesloch
	H07	IKEA	NULL	Walldorf
	H08	Mr X	30.08.1986	Walldorf
	H09	Ms Y	10.02.1986	Sinsheim
	H10	Mr Z	03.02.1986	Ladenburg



Table 3: The Contact Table

The *Contact* database table relates a car owner owning more than two cars to the officials who are the respective contact persons.

Contact	PersNumber	OwnerID
	P01	H03
	P01	H04
	P01	H07



Contact	PersNumber	OwnerID
	P04	H03
	P04	H04
	P08	H04
	P08	H07
	P09	H03



Car	CarID	PlateNumber	Brand	Color	HP	Owner
	F01	HD-V 106	Fiat	red	75	H06
	F02	HD-VW 4711	VW	black	120	H03
	F03	HD-JA 1972	BMW	blue	184	H03
	F04	HD-AL 1002	Mercedes	white	136	H07
	F05	HD-MM 3206	Mercedes	black	170	H03
	F06	HD-VW 1999	Audi	yellow	260	H05
	F07	HD-ML 3206	Audi	blue	116	H03
	F08	HD-IK 1002	VW	black	160	H07
	F09	HD-UP 13	Skoda	red	105	H02
	F10	HD-MT 507	BMW	black	140	H04
	F11	HD-MM 208	BMW	green	184	H02
	F12	HD-XY 4711	Skoda	red	105	H04
	F13	HD-IK 1001	Renault	red	136	H07
	F14	HD-MM 1977	Mercedes	white	170	H03
	F15	HD-MB 3030	Skoda	black	136	H03
	F16	NULL	Opel	green	120	NULL
	F17	HD-Y 333	Audi	orange	184	H09
	F18	HD-MQ 2006	Renault	red	90	H03
	F19	HD-VW 2012	VW	black	125	H01
	F20	NULL	Audi	green	184	NULL

Figure 34: Cars



Stolen	PlateNumber	reported_at
	HD-VW 1999	20.06.2012
	HD-V 106	01.06.2012
	HD-Y 333	21.05.2012

Figure 35: Stolen Cars



- A new (fictional) European Union directive requires that the information about which vehicle is registered to which owner has to be stored in a central transnational database.
- The vehicle identification number (CarID) is unique across the EU, but not the Owner ID.

`Owner_EU(Country, OwnerID, Name, Birthday, City)`  
  
`Car_EU(CarID, PlateNumber, Brand, Color, HP, Country, Owner)`



Figure 36: Registration Office and EU



Owner_EU	Country	OwnerID	Name	Birthday	City
D	H01	Ms T		20.06.1934	Wiesloch
D	H02	Ms U		11.05.1966	Hockenheim
D	H03	SAP AG		<i>NULL</i>	Walldorf
D	H04	HDM AG		<i>NULL</i>	Heidelberg
D	H05	Mr V		21.04.1952	Leimen
D	H06	Ms W		01.06.1957	Wiesloch
D	H07	IKEA		<i>NULL</i>	Walldorf
D	H08	Mr X		30.08.1986	Walldorf
D	H09	Ms Y		10.02.1986	Sinsheim
D	H10	Mr Z		03.02.1986	Ladenburg
A	H01	Ms O		21.05.1977	Wien
A	H02	Mr P		02.08.1977	Salzburg
E	H01	Señor Q		18.02.1925	Madrid
E	H02	Señora R		27.02.1927	Barcelona



Figure 37: Owner (EU-Wide)



Car_EU	CarID	PlateNumber	Brand	Color	HP	Country	Owner
F01	HD-V 106	Fiat	red	75	D	H06	
F02	HD-VW 4711	VW	black	120	D	H03	
F03	HD-JA 1972	BMW	blue	184	D	H03	
F04	HD-AL 1002	Mercedes	white	136	D	H07	
F05	HD-MM 3206	Mercedes	black	170	D	H03	
F06	HD-VW 1999	Audi	yellow	260	D	H05	
F07	HD-ML 3206	Audi	blue	116	D	H03	
F08	HD-IK 1002	VW	black	160	D	H07	
F09	HD-UP 13	Skoda	red	105	D	H02	
F10	HD-MT 507	BMW	black	140	D	H04	
F11	HD-MM 208	BMW	green	184	D	H02	
F12	HD-XY 4711	Skoda	red	105	D	H04	
F13	HD-IK 1001	Renault	red	136	D	H07	
F14	HD-MM 1977	Mercedes	white	170	D	H03	
F15	HD-MB 3030	Skoda	black	136	D	H03	
F16	NULL	Opel	green	120	NULL	NULL	
F17	HD-Y 333	Audi	orange	184	D	H09	
F18	HD-MQ 2006	Renault	red	90	D	H03	
F19	HD-VW 2012	VW	black	125	D	H01	
F20	NULL	Audi	green	184	NULL	NULL	
F21	W-302 ML	Mercedes	black	170	A	H01	
F22	S-215 MM	VW	silvern	184	A	H02	
E22	1100 EMM	Audi	blau	116	E	UN1	

Figure 38: Cars (EU-Wide)

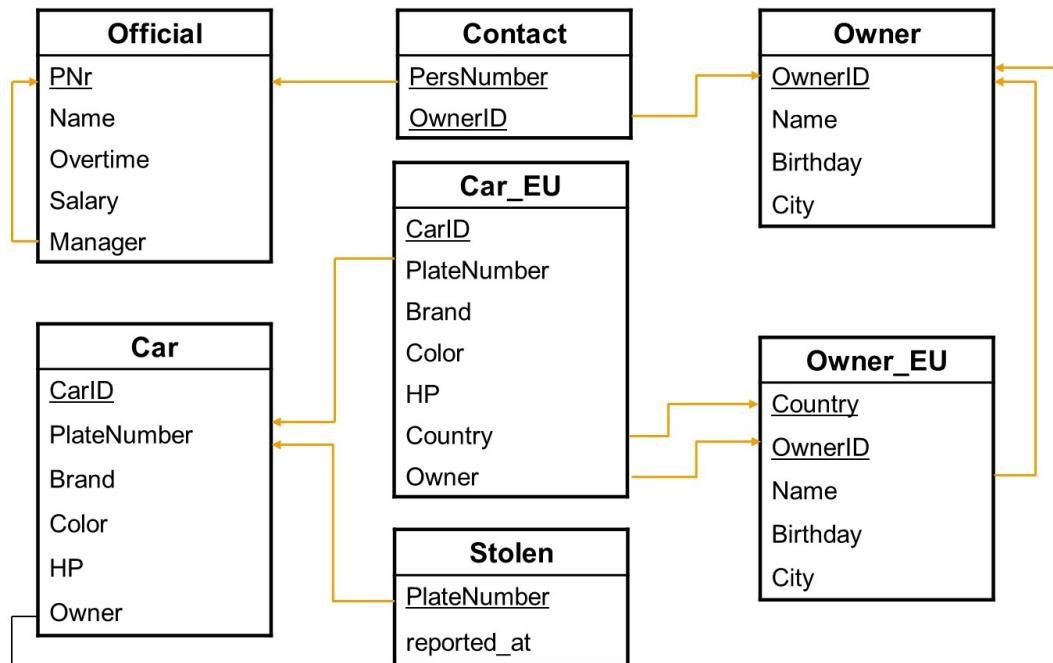


Figure 39: Relationships Between the Tables



## LESSON SUMMARY

You should now be able to:

- Understand the sample database used throughout the course



## Learning Assessment

1. In the SAP Web IDE for SAP HANA, when you create a new SAP HANA database module, what is added to the mta.yaml file?

*Choose the correct answers.*

- A A module of type HDB
- B A route to the SAP HANA database
- C A service of type com.sap.xs.hdi-container
- D A service of type OData

2. What is SQLScript?

*Choose the correct answer.*

- A A replacement for SQL that provides a more powerful language for developers
- B A set of extensions on top of standard SQL that leverage the specific features of SAP HANA
- C An open source version of SQL that is supported by most well-known databases

3. What is the SAP HANA Deployment Infrastructure (HDI)?

*Choose the correct answer.*

- A An isolated environment for storing SAP HANA database runtime objects
- B A source code versioning framework for SAP HANA source files
- C A type of application that is built from components developed in multiple languages

4. The SQL Console is found in the Development view of the Web IDE.

*Determine whether this statement is true or false.*

- True
- False

## Learning Assessment - Answers

1. In the SAP Web IDE for SAP HANA, when you create a new SAP HANA database module, what is added to the mta.yaml file?

*Choose the correct answers.*

- A A module of type HDB
- B A route to the SAP HANA database
- C A service of type com.sap.xs.hdi-container
- D A service of type OData

Correct! In the SAP Web IDE for SAP HANA, when you create a SAP HANA database module, an SAP HANA database module and a *com.sap.xs.hdi-container* service are added to the mta.yaml file.

2. What is SQLScript?

*Choose the correct answer.*

- A A replacement for SQL that provides a more powerful language for developers
- B A set of extensions on top of standard SQL that leverage the specific features of SAP HANA
- C An open source version of SQL that is supported by most well-known databases

Correct! SQLScript is a set of extensions on top of standard SQL that leverage the specific features of SAP HANA. It does not replace SQL and is not an open source version of SQL. See HA150 Unit 1 for details.

3. What is the SAP HANA Deployment Infrastructure (HDI)?

*Choose the correct answer.*

- A** An isolated environment for storing SAP HANA database runtime objects
- B** A source code versioning framework for SAP HANA source files
- C** A type of application that is built from components developed in multiple languages

Correct! HDI is an isolated environment for storing SAP HANA database runtime objects. HDI is not a source code versioning framework; that is Git. HDI is not an application that is built from components developed in multiple languages. See HA150 Unit 1 for details.

4. The SQL Console is found in the Development view of the Web IDE.

*Determine whether this statement is true or false.*

- True
- False

Correct! The SQL Console is found in the Database Explorer view of the Web IDE. See HA150 Unit 1 for details.



**Lesson 1**

Understanding Motivation and Basic Concepts

49

**Lesson 2**

Using Data from a Table or View

59

**Lesson 3**

Understanding NULL Values

83

**Lesson 4**

Aggregating Data

87

**Lesson 5**

Understanding Unions and Joins

95

**Lesson 6**

Changing Data Stored in Tables

115

**UNIT OBJECTIVES**

- Understand the motivation for and foundation of the relational model
- Understand SQL and its relationship to the relational model
- Understand database tables as the most important database objects
- Write simple database queries using SQL's SELECT statement and project columns in and out of queries using the SELECT clause
- Calculate column values, use built-in functions and the CASE expression in column lists
- Avoid duplicates in SELECT statement result sets
- Limit results sets to a given number of rows and browse through result sets
- Ensure a specific order in result sets

- Restrict the result set using the WHERE clause
- Interpret NULL values in databases and understand why their presence can lead to unexpected query results
- List the most important aggregate functions supported by HANA and use them to determine aggregated values on table columns using a single SELECT statement
- Determine aggregated values for groups of rows, using the GROUP BY clause
- Filter groups using the HAVING clause
- Read data from multiple tables
- List the various types of JOIN constructs and use the appropriate JOIN construct to combine data from several tables using a single query
- Add rows to database tables using SQL
- Change existing rows of a database table
- Remove existing rows from a database table

# Understanding Motivation and Basic Concepts

## LESSON OVERVIEW

This lesson explains why it may be important to learn about SAP HANA SQL and the database model from which it originates.



## LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Understand the motivation for and foundation of the relational model
- Understand SQL and its relationship to the relational model
- Understand database tables as the most important database objects

## Terms: Database Management System, Database, and Database System

The terms Database, Database System, and Database Management System are frequently used incorrectly and interchangeably:



- “Which **Database Management System** do you use?”
- “We are using SAP HANA as our **Database Management System**. ”
- In everyday life, these terms are usually used incorrectly ...

Figure 40: Terms DB, DBS, and DBMS

The terms in fact refer to slightly different concepts:

### Database (DB)

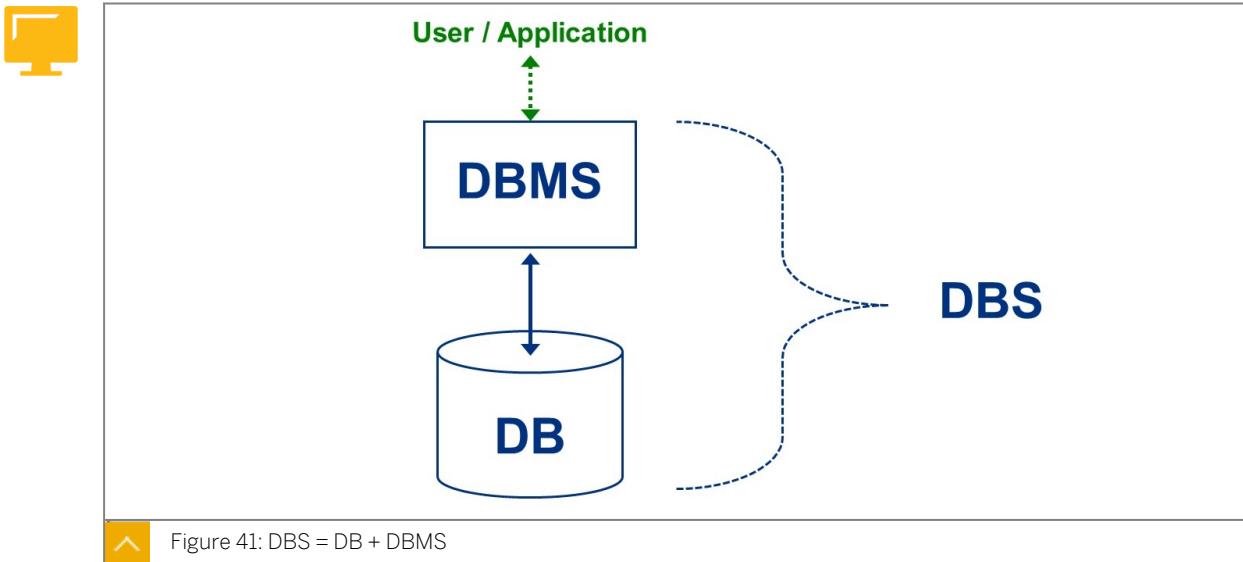
A database is a concrete structured collection of “records”.

### Database Management System (DBMS)

A database management system is software that manages databases. Every access to a database (create, read, insert, update, delete) goes exclusively through the DBMS. The DBMS exercises complete control over the databases it manages.

### Database System

A database system is the combination of the implementation of a DBMS and the databases it manages.



### Goals of the Relational Database Model

The way that data records are structured and how DBMSs are implemented are referred to as database models. The following are some of the database models that have been developed over the past decades:

- Hierarchical Database Model
- Network Database Model
- Relational Database Model
- Object-relational Database Model
- Object-oriented Database Model
- XML-based Database Model

### Design Goals of the Relational Database Model

The **Relational Database Model** was invented in the late 1960s by Edgar F. Codd (1923–2003) while working at IBM Almaden Research Lab in San José, California. The Relational Database Model was designed to meet the following specific goals:

- 
- Provide a simple yet mathematically profound and precise database model.
  - Make monitoring data integrity largely a responsibility of the DBMS.
  - Separate the conceptual schema from the internal schema.
  - Make data storage and retrieval the sole responsibility of the DBMS.
    - A descriptive data access language removes the need to traverse data links.
    - Database optimizers choose the optimal execution strategy for a given query.
  - Provide a simple database access language with semantics described with mathematical precision.
    - Easy to learn.
    - Equivalence of different queries can be proved.

The goal behind separating the conceptual schema from the internal schema is to allow for a three-level architecture so that changes on a lower level do not impact higher levels.



### Objective: Changes at a lower level should not affect a higher level (if possible)

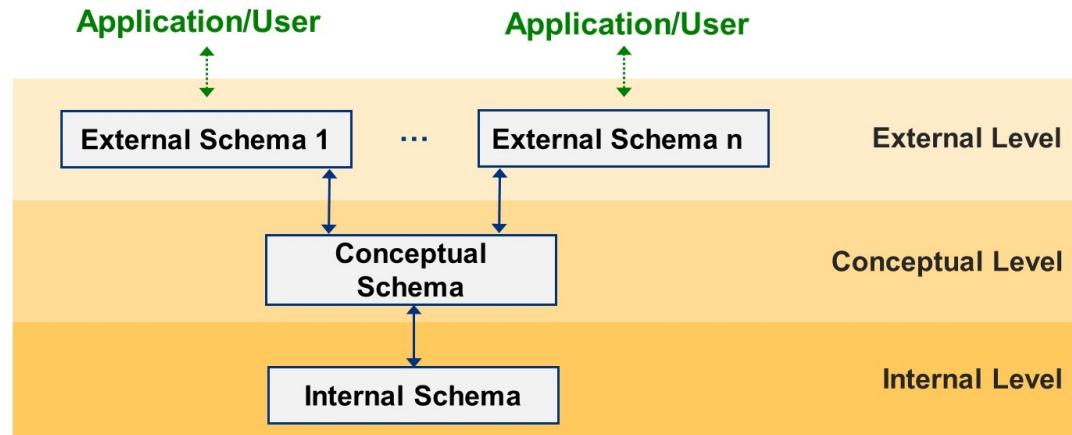


Figure 42: Three-Level Schema Architecture

The **External Schema** defines how data is presented to the user:

- (Partial) views of the data as required by applications or users
- In the case of a relational database system, implemented via views

The **Conceptual Schema** defines what is stored:

- Overall presentation of the data model at the logical, (if possible) DBMS, and application-independent level
- For example, in a relational representation, or even higher level of abstraction (for example, E/R model)

The **Internal Schema** defines how and where it is stored:

- Describes (DBMS-specific) the internal, physical representation of data
- How and where exactly the data is stored, internal record format, access paths, and so on



- **The address book should not display salary data**
  - **External Level (VIEW)**
- **An Employee has a D-Number, name, and salary and is assigned to a department**
  - **Conceptual Level (TABLE)**
- **The board mostly accesses employee data according to descending order of salary (which has to be very fast)**
  - **Internal Level (INDEX)**

Figure 43: Three Level Schema Architecture Overview

## The Role of SQL

### Languages for the Relational Database Model

The following languages are available to interact with a relational DBMS:



- Relational Algebra
  - Formal basis for DBMS internal query optimization
  - 6 basic operations: Selection, Projection, Cartesian Product, Union, Difference, and Rename
- Relational Calculus
  - Tuple variables and quantifiers
- SQL
  - Standardized and used in practice



<p>In almost every business application scenario, the data is managed using database systems.</p> <p>The most significant are database systems based on the relational data model and using SQL (Structured Query Language) as a database language.</p>	<p>SQL is a widely-established, powerful, standardized database language many application programmers have experience in.</p> <p>There is (so far) no other database language that has all the advantages mentioned.</p>
<ul style="list-style-type: none"> <li>• SAP HANA is a relational database management system</li> <li>• SAP HANA supports SQL</li> </ul>	
<span style="color: #FFDAB9;">↗</span> Figure 44: Why SQL?	

## ANSI SQL

SQL is a computer language that is used to define, manipulate, and control relational databases. It is defined by the American Standards Institute (ANSI).

SQL is designed as a descriptive rather than procedural language. Using SQL you express what data you want, not exactly how to retrieve it. This design allows the DBMS to first parse each statement and then optimize it. In theory, the optimizer determines the best execution plan.

## SQL Language Elements



Table 4: SQL Language Elements

SQL language elements can be divided into the following three categories:

Data Manipulation Language (DML) statements	DML statements <code>insert</code> , <code>update</code> , and <code>delete</code> the data in a database. DML statements also <code>select</code> data that you want to read.
---	--

Data Definition Language (DDL) statements	DDL statements <code>create</code> , <code>alter</code> , and <code>drop</code> databases, tables and other database objects in the server.
Data Control Language (DCL) statements	DCL statements <code>grant</code> and <code>revoke</code> permissions from database users.

## SQL and Relational Model

SQL is the most important database language for the relational database model. However, it deviates in important points from the “purely relational” model.

Because SQL is designed to work with data stored in tables, in other words with sets, it is **multi-set** oriented and not single record-based. Using a single SQL statement, you can read multiple table rows, modify, or delete them in one go.



### SQL is Not Fully Relational

SQL deviates from the purely relational model in the following ways:



- Results of SQL queries can contain duplicates (identical rows). This makes SQL **multi-set** oriented and not set oriented.
- SQL allows for NULL values, leading to a three-valued logic.
- SQL language is not closed.

## SAP HANA SQL and SQLScript

SAP HANA SQL, including SQLScript, is the computer language created by SAP and used in the SAP HANA platform. It is built on a foundation of standard ANSI SQL, and it extends the standard considerably, in part to better support the column and in-memory store nature of SAP HANA.

### SAP HANA SQL and SQLScript Elements

The following is a list of selected elements of SAP HANA SQL:



- Standard (ANSI) SQL statements
- SAP extensions to SQL statements for SAP HANA, for example:
  - DML extensions, for example `SELECT INTO`, `UPSERT`
  - DDL extensions, for example to allow creating column and row store tables
  - Functions
  - Triggers
- Declarative SQLScript statements, for example variable assignments
- Imperative SQLScript statements (flow control statements)

## Database Objects



- Tables are the primary database object, but not the only one. Apart from tables, a database usually contains the following:

- Views to simplify and limit data access
- Indexes to speed up (certain) read accesses
- Constraints to ensure data consistency
- Stored procedures for more complex tasks
- Triggers to selectively respond to particular events



**A table consists of columns (structure) and rows (data instances)**

- A table can therefore be represented two-dimensionally

**A (database) table represents a relation**

- A table represents an (unordered) multi-set of points in n-dimensional space
- Each of the n dimensions is equivalent to a table column

Employee	PersNumber	Name	HiringYear
	P036407	Paul	2001
	P040824	Ben	2008
	P052867	Raja	2010



```
Employee = { (P036407, Paul, 2001), (P040824, Ben, 2008),
             (P052867, Raja, 2010) }
```



Figure 45: Database Tables

## Components of Database Tables



**Table-Name**                    **Primary Key**                    **Column Name**

Employee	PersNumber	Name	HiringYear
	P036407	Paul	2001
	P040824	Ben	2008
	P052867	Raja	2010

**Table Row** →

**(Table Column) value**

**(Table Column)**



Figure 46: Components of Database Tables



**Key =** is a set of columns which serves to uniquely identify any row in the table.

The ability to uniquely identify rows must apply in principle (and not only for the rows existing at a certain point in time).

Employee	PersNumber	Name	HiringYear
	P036407	Paul	2001
	P040824	Ben	2008
	P052867	Raja	2010

Figure 47: Keys



### A key can consist of multiple columns

Employee	Name	Country	City	BuildingNr	Block	Floor	Room	SeatNr
Mr A	DE	WDF	03	A	5	29	1	
Ms B	DE	WDF	03	A	5	29	2	
Ms C	DE	WDF	03	A	5	29	3	
...	...	...	...	...	...	...	...	...

- In this example the key consists of 7 columns
- 1 Key (not 7 keys)!

Figure 48: Multi-Column Keys



### There may be more than one key

Employee	PNumber	Name	Tax ID	Passport ID	Plate Number	Chasis Nr	CarLicense ID	SWIFT	IBAN
P000815	...	...	...	...	HD-MM 815	...	...	...	...
P004711	...	...	...	...	HD-ML 4711	...	...	...	...
P012345	...	...	...	...	HD-OI 2345	...	...	...	...
...	...	...	...	...	...	...	...	...	...

- Here are 7 keys (provided there are no joint accounts)
- Just 1 key is selected as Primary Key

Figure 49: Multiple Keys



**Foreign-Key** = set of columns, which is a (primary) key in an(other) table

- The foreign-key can refer to its own table
- It is not necessary to share the same names
- The foreign-key can contain only those values that occur as a (primary) key value in another table (in addition and if applicable, NULL values are allowed)
- The foreign-key is usually not a key!

### Foreign-key Relationship

Employee	DNumber	Name	DepartmentNR	Department	DNr	Function
	D010000	Mr A	A01		A01	HANA-Development
	D010001	Ms B	A01		A02	HANA-Sales
	D010002	Mr C	A02		A03	HANA-Training
	D010003	Ms D	A02		A04	ABAP-Development
	D010004	Mr E	A02			
	D010005	Ms F	A03			

Figure 50: Foreign Keys



**A Foreign-Key can consist of multiple columns**

Publication	PaperID	Topic	Uni	StudentID	Student	Uni	StudentID	Name	FirstName
	P001	In-Memory	HPI	12345		HPI	12345	A	B
	P002	Column Store	HPI	12345		HPI	77777	C	D
	P003	Row Store	HPI	77777		FSU	12345	E	F
	P004	SQL & XML	FSU	12345		FSU	77777	G	H
	P005	XML & SQL	FSU	12345					
	P006	DB Recovery	FSU	77777					

- This example shows 1 Foreign-Key built on 2 columns

Figure 51: Multi-Column Foreign Keys



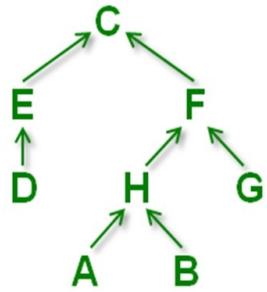
**There can be multiple Foreign-Keys**

Vendor	VID	Name	Delivery	VID	PNr	Year	Quantity	Product	PNr	Description
	L1	Heidelberg Paper		L1	A1	2010	320		A1	Sticky Notes
	L2	Wiesloch Paper		L1	A1	2011	570		A2	Printing Paper
	L3	Walldorf Paper		L1	A1	2012	925		A3	Envelopes
	...	...		L1	A2	2012	102		...	...
				L2	A1	2011	577			
				L2	A2	2011	100			
				L3	A1	2012	999			
				...	...	...	...			

Figure 52: Multiple Foreign Keys



### The foreign-key can refer to its own table



Employee	DNumber	Name	DNumberManager
D010000	Mr A	D010007	
D010001	Ms B	D010007	
D010002	Mr C		
D010003	Ms D	D010004	
D010004	Mr E	D010002	
D010005	Ms F	D010002	
D010006	Mr G	D010005	
D010007	Ms H	D010005	



Figure 53: Self-Referencing Foreign Keys



### LESSON SUMMARY

You should now be able to:

- Understand the motivation for and foundation of the relational model
- Understand SQL and its relationship to the relational model
- Understand database tables as the most important database objects



## Using Data from a Table or View

### LESSON OVERVIEW

The lesson covers the foundation of how to retrieve data from a database.



### LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Write simple database queries using SQL's SELECT statement and project columns in and out of queries using the SELECT clause
- Calculate column values, use built-in functions and the CASE expression in column lists
- Avoid duplicates in SELECT statement result sets
- Limit results sets to a given number of rows and browse through result sets
- Ensure a specific order in result sets
- Restrict the result set using the WHERE clause

### Overview of the SELECT Statement

The `SELECT` statement is an SQL statement that is used to read data from a database table or view. The `SELECT` statement is the central construct for read access to data, of the Data Manipulation Language.

A `SELECT` statement must contain a `SELECT` and a `FROM` clause and can contain a `WHERE`, `GROUP BY`, `HAVING`, and `ORDER BY` clause.



```
SELECT Column, Column, COUNT(*)
      FROM Table
      WHERE Condition
      GROUP BY Column, Column
      HAVING Group_Condition
      ORDER BY Column ASC, Column DESC;
```

Figure 54: The `SELECT` Statement: Mandatory and Optional Clauses

### The `SELECT` Clause in its Most Simple Form

The `SELECT` clause specifies the **Projection List**. The Projection List defines which columns to read from the data or which columns to include in the result set of the `SELECT` statement.



You can specify multiple columns in the projection list:

```
SELECT PNr, Name, Salary
FROM Official;
```

PNR	NAME	Salary
P01	Mr A	A09
P02	Mr B	A10
P03	Ms C	A09
P04	Ms D	A12
P05	Mr E	A08
P06	Mr F	A09
P07	Ms G	A11
P08	Ms H	A13
P09	Mr I	A14



Figure 55: Multiple Columns in Projection List



The sequence of the columns in the projection list is relevant:

```
SELECT PNr, Name
FROM Official;
```

```
SELECT Name, PNr
FROM Official;
```

PNR	NAME
P01	Mr A
P02	Mr B
P03	Ms C
P04	Ms D
...	...

NAME	PNR
Mr A	P01
Mr B	P02
Ms C	P03
Ms D	P04
...	...



Figure 56: Sequence of Columns in the Projection List



An asterisk (\*) in the projection list represents “all columns”:

```
SELECT *
FROM Official;
```

PNR	NAME	OVERTIME	SALARY	MANAGER
P01	Mr A	10	A09	P04
P02	Mr B	10	A10	P04
P03	Ms C	20	A09	P04
P04	Ms D	?	A12	P09
...	...	...	...	...



Figure 57: All Columns Asterisk in Projection List

The following options are also supported:

- You can include a single column in the projection list.
- You can include the same column in the projection list repeatedly.

- You can combine using the asterisk and explicitly named columns.
- You can use the asterisk repeatedly in the projection list.

We do not recommend using the last three options because each makes the result set difficult to understand and work with.

### Naming Columns of the Result Set

It is possible to rename the columns of the result set so that they differ from the column names in the table.



Note:

Renaming columns in the `SELECT` clause has no effect on the table from which they are selected. It only changes the result set returned.



#### You can rename result columns:

```
SELECT PNr AS PersNumber,
       Salary AS "Salary Group"
  FROM Official;
```

PERSNUMBER	Salary Group
P01	A09
P02	A10
P03	A09
P04	A12
...	...



Figure 58: Renaming Columns in the Projection

Column names not enclosed in quotation marks are converted to upper case. A column name has to be enclosed in quotation marks if one of the following applies:

- The column name should not be converted to upper case but be kept exactly as spelled in your statement.
- The column name contains a space, a punctuation mark, a colon, or any other special character.



Note:

This rule also applies to the names of the original database tables and their columns; the SQL parser converts names not enclosed in quotation marks to upper case. It then performs a case-sensitive search for the corresponding database objects.

The keyword `AS` is optional.

You can use column names of the table selected from to rename other columns of the result set, for example, to exchange column names.



You can use existing column names for (re)naming of result columns:

```
SELECT PNr Salary, Salary PNr
  FROM Official;
```

SALARY	PNR
P01	A09
P02	A10
P03	A09
P04	A12
P05	A08
P06	A09
P07	A11
P08	A13
P09	A14

Figure 59: Reusing Existing Column Names for Other Columns

### The SELECT Clause: Calculated Columns

The SELECT clause can also contain columns with values that are computed dynamically, typically based on the values of other columns.

In the most simple form, these calculated columns can be literal values. Both string and numerical literals are supported.



You can generate additional “literal” result columns.

The additional result column can have a string or a numeric type:

```
SELECT 'The working week of', Name, 'amounts to', 40,
'hours.'
  FROM Official;
```

'The working week of'	NAME	'amounts to'	40	'hours.'
The working week of	Mr A	amounts to	40	hours.
The working week of	Mr B	amounts to	40	hours.
The working week of	Ms C	amounts to	40	hours.
The working week of	Ms D	amounts to	40	hours.
The working week of	Mr E	amounts to	40	hours.
The working week of	Mr F	amounts to	40	hours.
The working week of	Ms G	amounts to	40	hours.
The working week of	Ms H	amounts to	40	hours.
The working week of	Mr I	amounts to	40	hours.

Figure 60: Literal Values in the SELECT Clause



Note:

String values are enclosed in single quotes. This can lead to confusion if column names are enclosed in quotation marks as well.

Numerical values are not enclosed in quotes.

The projection list can even comprise one or more literal values only. Note that the result set still contains as many records, all comprising the same literal values only, as the table select

from contains. If your intent is to select a single “literal record”, you can use the built-in table DUMMY.



If the projection list only contains artificial result columns, you can use the special table “Dummy” as reference:

```
SELECT 'Good Morning!'
FROM Dummy;
```

```
'Good Morning!'
-----
Good Morning!
```

The “DUMMY” table contains one column and one row:

```
SELECT *
FROM Dummy;
```

DUMMY
X

Figure 61: Built-In Table DUMMY



The projection list can have computed columns.

If a NULL value is used in an arithmetic operation, the result is also a NULL value:

```
SELECT Name, Overtime * 60
      FROM Official;
```

NAME	OVERTIME*60
Mr A	600
Mr B	600
Ms C	1200
Ms D	?
Mr E	600
Mr F	1080
Ms G	1320
Ms H	?
Mr I	?

Figure 62: Calculated Columns



You can explicitly name computed result columns:

```
SELECT Name, Overtime * 60 AS Overminutes
  FROM Official;
```

NAME	OVERMINUTES
Mr A	600
Mr B	600
Ms C	1200
Ms D	?
...	...



Figure 63: Naming Calculated Columns

SAP HANA supports using functions to calculate columns of the projection list. Different functions are available for different data types of the input columns and values.



When were vehicle owners first allowed to drive a car in Germany?

```
SELECT Name, ADD_YEARS(Birthday, 18)
      AS "18th Birthday"
  FROM Owner;
```

NAME	18th Birthday
Ms T	1952-06-20
Ms U	1984-05-11
SAP AG	?
HDM AG	?
Mr V	1970-04-21
Ms W	1975-06-01
IKEA	?
Mr X	2004-08-30
Ms Y	2004-02-10
Mr Z	2004-02-03



Figure 64: Functions



### Function calls can be nested

- On which day is the owner's 18<sup>th</sup> birthday?

```
SELECT Name,
       DAYNAME
      ( ADD_YEARS
        (Birthday,
         ROUND(ABS(-18.2))
        )
      ) AS Weekday
FROM Owner;
```

NAME	WEEKDAY
Ms T	FRIDAY
Ms U	FRIDAY
SAP AG	?
HDM AG	?
Mr V	TUESDAY
Ms W	SUNDAY
IKEA	?
Mr X	MONDAY
Ms Y	TUESDAY
Mr Z	TUESDAY

Figure 65: Function Calls can be Nested



Table 5: Selection of Functions Supported by SAP HANA

SAP HANA supports a rich list of built-in function. The following is only a selection.

Function	Explanation
ADD_YEAR (<date>, <number>)	Adds a <number> of years to the specified <date>
CURRENT_DATE	Determines the current date
ABS (<number>)	Determines the absolute value
ROUND (<number>)	Rounds the <number>
SQRT (<number>)	Determines the square root of <number>
UPPER (<string>)	Converts the <string> to upper case
SUBSTR (<string>, <start>, <len>)	Extracts a substring of <string>
CONCAT (<string>, <string>)	Concatenates two strings. Equivalent to the binary operator   .
COALESCE (<expressions>)	Returns the result of the first element of the list of <expressions> that does not evaluate to NULL.

See the SAP HANA SQL Reference available on the SAP Help Portal for the full list of supported built-in functions. [http://help.sap.com/hana\\_platform#section6](http://help.sap.com/hana_platform#section6)

You can also create scalar User-Defined Functions and use them in the same way as built-in functions.

### The CASE Expression

A special way of calculating column values is using a CASE expression. This expression allows you to determine a column value based on conditions, in other words, to include an IF-THEN-ELSE type of logic in the projection list.



The projection list can contain columns that are based on a CASE expression

- These columns can be named explicitly:

```
SELECT *,  
      CASE  
        WHEN HP < 120 THEN 'low'  
        WHEN HP >= 120 AND HP < 180 THEN 'medium'  
        ELSE 'high'  
      END AS Rating  
FROM Car;
```

Figure 66: Case Expression: Basic Form



CARID	PLATENUMBER	BRAND	COLOR	HP	OWNER	RATING
F01	HD-V 106	Fiat	red	75	H06	low
F02	HD-VW 4711	VW	black	120	H03	medium
F03	HD-JA 1972	BMW	blue	184	H03	high
F04	HD-AL 1002	Mercedes	white	136	H07	medium
F05	HD-MM 3206	Mercedes	black	170	H03	medium
F06	HD-VW 1999	Audi	yellow	260	H05	high
F07	HD-ML 3206	Audi	blue	116	H03	low
F08	HD-IK 1002	VW	black	160	H07	medium
F09	HD-UP 13	Skoda	red	105	H02	low
F10	HD-MT 507	BMW	black	140	H04	medium
F11	HD-MM 208	BMW	green	184	H02	high
F12	HD-XY 4711	Skoda	red	105	H04	low
F13	HD-IK 1001	Renault	red	136	H07	medium
F14	HD-MM 1977	Mercedes	white	170	H03	medium
F15	HD-MB 3030	Skoda	black	136	H03	medium
F16	?	Opel	green	120	?	medium
F17	HD-Y 333	Audi	orange	184	H09	high
F18	HD-MQ 2006	Renault	red	90	H03	low
F19	HD-VW 2012	VW	black	125	H01	medium
F20	?	Audi	green	184	?	high

Figure 67: Case Expression: Result



If a CASE expression does not correspond to any of the listed cases, a NULL value is returned:

```
SELECT CarID,
       CASE Color
           WHEN 'red'      THEN 'FF0000'
           WHEN 'green'    THEN '00FF00'
           WHEN 'yellow'   THEN 'FFFF00'
           WHEN 'blue'     THEN '0000FF'
           WHEN 'white'    THEN 'FFFFFF'
           WHEN 'black'    THEN '000000'
       END AS Color
FROM Car;
```

CARID	COLOR
F01	FF0000
F02	000000
F03	0000FF
F04	FFFFFF
F05	000000
F06	FFFF00
F07	0000FF
F08	000000
F09	FF0000
F10	000000
F11	00FF00
F12	FF0000
F13	FF0000
F14	FFFFFF
F15	000000
F16	00FF00
F17	?
F18	FF0000
F19	000000
F20	00FF00

Figure 68: Case Expression: Case not Handled



Both the THEN and the ELSE branch can contain references to table columns:

```
SELECT CarID,
       CASE
           WHEN PlateNumber IS NOT NULL THEN PlateNumber
           ELSE 'The Car is not registered!'
       END AS PlateNumber,
       CASE Brand
           WHEN 'Mercedes' THEN 'Mercedes-Benz'
           WHEN 'VW'        THEN 'Volkswagen'
           ELSE Brand
       END AS Brand,
       Color, HP
FROM Car;
```

Figure 69: Case Expression: Using Columns in the THEN and ELSE Branches



CARID	PLATENUMBER	BRAND	COLOR	HP
F01	HD-V 106	Fiat	red	75
F02	HD-VW 4711	Volkswagen	black	120
F03	HD-JA 1972	BMW	blue	184
F04	HD-AL 1002	Mercedes-Benz	white	136
F05	HD-MM 3206	Mercedes-Benz	black	170
F06	HD-VW 1999	Audi	yellow	260
F07	HD-ML 3206	Audi	blue	116
F08	HD-IK 1002	Volkswagen	black	160
F09	HD-UP 13	Skoda	red	105
F10	HD-MT 507	BMW	black	140
F11	HD-MM 208	BMW	green	184
F12	HD-XY 4711	Skoda	red	105
F13	HD-IK 1001	Renault	red	136
F14	HD-MM 1977	Mercedes-Benz	white	170
F15	HD-MB 3030	Skoda	black	136
F16	The Car is not registered!	Opel	green	120
F17	HD-Y 333	Audi	orange	184
F18	HD-MQ 2006	Renault	red	90
F19	HD-VW 2012	Volkswagen	black	125
F20	The Car is not registered!	Audi	green	184

Figure 70: Case Expression: Result

## Eliminating Duplicates in Result Sets

An important difference between SQL and the Relational Database Model is that SQL is multi-set oriented, which allows for duplicate entries in result sets. However, you won't notice this as long as the database table you select from has a primary key, and once you select all of the columns.



### A table with an enforced primary key does not contain duplicates

```
SELECT *
FROM Car;
```

CARID	PLATENUMBER	BRAND	COLOR	HP	OWNER
...	...	...	...	...	...
F04	HD-AL 1002	Mercedes	white	136	H07
F05	HD-MM 3206	Mercedes	black	170	H03
...	...	...	...	...	...
F09	HD-UP 13	Skoda	red	105	H02
...	...	...	...	...	...
F14	HD-MM 1977	Mercedes	white	170	H03
F15	HD-MB 3030	Skoda	black	136	H03
...	...	...	...	...	...

Figure 71: No Duplicates with Primary Key

If you project a table and don't include the full primary key, duplicate entries can result.



Duplicates can occur when a key column is not included in the projection list:

```
SELECT Brand  
FROM Car;
```

BRAND
-----
...
Mercedes
Mercedes
...
Skoda
...
Mercedes
Skoda
...



Figure 72: Duplicates by Projection

In most cases, this is not intended or practical, as there is no way to distinguish the duplicate entries from each other in the result set. This is why duplicate entries can be removed explicitly using keyword **DISTINCT**. Note that duplicates are *not* removed by default to speed up data retrieval.



The keyword **DISTINCT** ensures that the result table contains no duplicates:

```
SELECT DISTINCT Brand  
FROM Car;
```

BRAND
-----
Fiat
VW
BMW
Mercedes
Audi
Skoda
Renault
Opel



Figure 73: Duplicate Elimination using DISTINCT



NULL values are treated in duplicate elimination as "normal" values. The result table contains (at most) one row that consists entirely of NULL values:

```
SELECT DISTINCT Overtime  
FROM Official;
```

OVERTIME
-----
10
20
?
18
22



Figure 74: Duplicate Elimination: Treatment of NULL Values



If a projection list contains multiple columns, **DISTINCT** always refers to the combination of all these columns:

```
SELECT DISTINCT Brand, Color
FROM Car;
```

BRAND	COLOR
Mercedes	white
Mercedes	black
Skoda	red
Skoda	black
...	...

Figure 75: DISTINCT Eliminates Duplicates in Multi-Column Result Set

In principle, you could always include the **DISTINCT** keyword in the **SELECT** statement to be sure you never get duplicates, since **DISTINCT** is compatible with using an asterisk in the projection list. Note, however, that using **DISTINCT** comes at a small performance penalty.

### The TOP N Clause

You can use the **TOP N** clause as part of the **SELECT** clause to specify that the result set should contain at most N rows.



You can limit the number of rows to be included in the query result to a maximum number.

- What are the ten most powerful vehicles (based on horse power)?

```
SELECT TOP 10 CarID, Brand, Color, HP
FROM Car
ORDER BY 4 DESC;
```

CARID	BRAND	COLOR	HP
F06	Audi	yellow	260
F20	Audi	green	184
F17	Audi	orange	184
F11	BMW	green	184
F03	BMW	blue	184
F14	Mercedes	white	170
F05	Mercedes	black	170
F08	VW	black	160
F10	BMW	black	140
F15	Skoda	black	136

Figure 76: Limiting the Result Set to a Maximum Number of Rows



You can combine the Top-n Clause with the keyword DISTINCT.

- What are the 7 highest HP values?

```
SELECT TOP 7 DISTINCT HP
FROM Car
ORDER BY 1 DESC;
```

HP
---
260
184
170
160
140
136
125

Figure 77: TOP N Clause can be Combined with DISTINCT



No error is thrown if you request more rows than available.

The result set will not be filled with additional, “artificial” rows.

- 570 different colors are requested:

```
SELECT TOP 570 DISTINCT Color
FROM Car;
```

COLOR
-----
red
black
blue
white
yellow
green
orange

Figure 78: TOP N Clause with Fewer Rows than N



It is possible to request 0 result rows.

In this case the result set does not contain any row.

- Zero colors are requested:

```
SELECT TOP 0 Color
FROM Car;
```

COLOR
-----

Figure 79: TOP N Clause with N=0

## The LIMIT and OFFSET Clauses

The TOP N clause allows you to limit the result set to a specified maximum number of rows, but it does not allow you to retrieve the “next” N rows. To browse through a result in a page-by-page way, you can use the LIMIT and OFFSET clauses.



You can use the Limit clause as an alternative to the Top-n clause.

The Limit clause comes at the very end of the SELECT statement.

- What are the 5 most powerful vehicles (based on horse power)?

```
SELECT *
FROM Car
ORDER BY HP DESC
LIMIT 5;
```

CARID	PLATENUMBER	BRAND	COLOR	HP	OWNER
F06	HD-VW 1999	Audi	yellow	260	H05
F20	?	Audi	green	184	?
F17	HD-Y 333	Audi	orange	184	H09
F11	HD-MM 208	BMW	green	184	H02
F03	HD-JA 1972	BMW	blue	184	H03

Figure 80: The LIMIT Clause as an Alternative to TOP N



The Limit clause can be combined with the Offset clause to skip records.

This allows you to read result sets “page by page”.

- What are the next 5 most powerful vehicles (based on horse power)?

```
SELECT *
FROM Car
ORDER BY HP DESC
LIMIT 5
OFFSET 5;
```

CARID	PLATENUMBER	BRAND	COLOR	HP	OWNER
F78	HD-MT 2510	?	green	260	?
F77	HD-MT 2509	?	red	184	?
F14	HD-MM 1977	Mercedes	white	184	H03
F05	HD-MM 3206	Mercedes	black	184	H03
F07	HD-IK 1002	VW	black	184	H07

Figure 81: Combine LIMIT and OFFSET to Browse through a Result Set



#### Note:

The LIMIT and OFFSET clauses are not part of the SELECT clause, but are the last clauses of a SELECT statement.

### The ORDER BY Clause

By default, the result returned by a query in SAP HANA has no defined and no reliable sort order. Executing the same query repeatedly can result in different sort orders.



Note:

Query results in the SAP HANA database are not sorted by primary key by default. There is no defined and no reliable default sort order.

You can use the `ORDER BY` clause to explicitly specify the sort order of the result set and this way get a reliable sort order.



**The result table can be sorted by a specific column:**

```
SELECT Brand, Color
  FROM Car
 ORDER BY Brand;
```

BRAND	COLOR
Audi	yellow
Audi	blue
Audi	orange
Audi	green
BMW	blue
BMW	black
BMW	green
Fiat	red
Mercedes	white
Mercedes	black
Mercedes	white
Opel	green
Renault	red
Renault	red
Skoda	red
Skoda	red
Skoda	black
VW	black
VW	black
VW	black



Figure 82: The ORDER BY Clause



**A descending sorting is possible:**

```
SELECT Brand, Color
  FROM Car
 ORDER BY Brand DESC;
```

BRAND	COLOR
VW	black
VW	black
VW	black
Skoda	black
Skoda	red
Skoda	red
Renault	red
Renault	red
Opel	green
Mercedes	white
Mercedes	black
Mercedes	white
Fiat	red
BMW	green
BMW	black
BMW	blue
Audi	green
Audi	orange
Audi	blue
Audi	yellow



Figure 83: Descending Sort Order



To sort ascending apply the optional keyword **ASC**:

```
SELECT Brand, Color
FROM Car
ORDER BY Brand ASC;
```

BRAND	COLOR
Audi	yellow
Audi	blue
Audi	orange
Audi	green
BMW	blue
BMW	black
BMW	green
Fiat	red
Mercedes	white
Mercedes	black
Mercedes	white
Opel	green
Renault	red
Renault	red
Skoda	red
Skoda	red
Skoda	black
VW	black
VW	black
VW	black



Figure 84: Ascending Sort Order



The sorting can be applied using a column that does not appear in the projection list:

```
SELECT Brand, Color
FROM Car
ORDER BY PlateNumber;
```

BRAND	COLOR
Opel	green
Audi	green
Renault	red
Mercedes	white
VW	black
BMW	blue
Skoda	black
Audi	blue
Mercedes	white
BMW	green
Mercedes	black
Renault	red
BMW	black
Skoda	red
Fiat	red
Audi	yellow
VW	black
VW	black
Skoda	red
Audi	orange



Figure 85: Sorting by a Column not in the Result



You can sort using a combination of columns:

```
SELECT Brand, Color
FROM Car
ORDER BY Brand ASC, Color DESC;
```

BRAND	COLOR
Audi	yellow
Audi	orange
Audi	green
Audi	blue
BMW	green
BMW	blue
BMW	black
Fiat	red
Mercedes	white
Mercedes	white
Mercedes	black
Opel	green
Renault	red
Renault	red
Skoda	red
Skoda	red
Skoda	black
VW	black
VW	black
VW	black



Figure 86: Sorting by Multiple Columns



Instead of the column names in the ORDER BY clause, the column numbers (based on the projection list) can be used:

```
SELECT Brand, Color
FROM Car
ORDER BY 1 ASC, 2 DESC;
```

BRAND	COLOR
Audi	yellow
Audi	orange
Audi	green
Audi	blue
BMW	green
BMW	blue
BMW	black
Fiat	red
Mercedes	white
Mercedes	white
Mercedes	black
Opel	green
Renault	red
Renault	red
Skoda	red
Skoda	red
Skoda	black
VW	black
VW	black
VW	black



Figure 87: Referencing Columns by Position



If result columns are named explicitly,  
you can refer to the new name for sorting:

```
SELECT Brand AS Manufacturer, Color
  FROM Car
 ORDER BY Manufacturer ASC, Color DESC;
```

MANUFACTURER	COLOR
Audi	yellow
Audi	orange
Audi	green
Audi	blue
BMW	green
BMW	blue
BMW	black
Fiat	red
Mercedes	white
Mercedes	white
Mercedes	black
Opel	green
Renault	red
Renault	red
Skoda	red
Skoda	red
Skoda	black
VW	black
VW	black
VW	black



Figure 88: Sorting and Column Renaming



You can sort based on calculated values.

- Sorting criteria:  
How much is the power below 200 kW?

```
SELECT CarID, Brand, HP
  FROM Car
 ORDER BY 200 - HP / 1.36 ASC;
```

CARID	BRAND	HP
F06	Audi	260
F03	BMW	184
F11	BMW	184
F17	Audi	184
F20	Audi	184
F05	Mercedes	170
F14	Mercedes	170
F08	VW	160
F10	BMW	140
F04	Mercedes	136
F13	Renault	136
F15	Skoda	136
F19	VW	125
F02	VW	120
F16	Opel	120
F07	Audi	116
F09	Skoda	105
F12	Skoda	105
F18	Renault	90
F01	Fiat	75



Figure 89: Sorting by Calculated Columns

You can also use functions in the expression in the ORDER BY clause.

### The WHERE Clause

The WHERE clause is used to select a subset of rows from the data source based on a specified condition. Only the rows matching the condition are returned.



**The WHERE clause is used to filter rows.**

**It is used to extract only those rows that fulfill a specified criterion.**

```
SELECT PlateNumber, Brand, Color
FROM Car
WHERE Brand = 'BMW' ;
```

PLATENUMBER	BRAND	COLOR
HD-JA 1972	BMW	blue
HD-MT 507	BMW	black
HD-MM 208	BMW	green



Figure 90: WHERE Clause



**You can reference a column in the WHERE clause that is not included in the projection list.**

```
SELECT Brand, Color
FROM Car
WHERE HP >= 170 ;
```

BRAND	COLOR
BMW	blue
Mercedes	black
Audi	yellow
BMW	green
Mercedes	white
Audi	orange
Audi	green



Figure 91: WHERE Clause

Condition expressions in SQL are very similar to condition expressions in programming languages such as ABAP, Java, and so on. The columns of the data source take over the roles of variables used in these languages.

- A condition can refer to string-like or to numeric columns. The columns used in the condition do not have to be included in the projection list.
- Atomic conditions can be combined to complex and nested conditions using the usual logical operators AND, OR, and NOT, and parentheses to override operator precedence rules.
- A condition can make use of functions, including nested functions.

An atomic condition typically contains a comparison operator. The usual comparison operators =, <>, <, <=, >, and >= are supported. But SQL supports a few additional comparison operators.



You can check for the presence of **NULL** values in the WHERE clause:

```
SELECT CarID, Brand, Color
FROM Car
WHERE PlateNumber IS NULL;
```

CARID	BRAND	COLOR
F16	Opel	green
F20	Audi	green

Figure 92: Comparison Operator IS NULL



You can check for the absence of **NULL** values in the WHERE clause:

```
SELECT CarID, Brand, Color
FROM Car
WHERE PlateNumber IS NOT NULL;
```

CARID	BRAND	COLOR
F01	Fiat	red
F02	VW	black
F03	BMW	blue
F04	Mercedes	white
F05	Mercedes	black
F06	Audi	yellow
F07	Audi	blue
F08	VW	black
F09	Skoda	red
F10	BMW	black
F11	BMW	green
F12	Skoda	red
F13	Renault	red
F14	Mercedes	white
F15	Skoda	black
F17	Audi	orange
F18	Renault	red
F19	VW	black

Figure 93: Comparison Operator IS NOT NULL



You can check if values are included **IN** a value-list.

```
SELECT Brand, Color
FROM Car
WHERE Color IN ('red', 'blue', 'orange');
```

BRAND	COLOR
Fiat	red
BMW	blue
Audi	blue
Skoda	red
Skoda	red
Renault	red
Audi	orange
Renault	red

Figure 94: Comparison Operator IN



You can check if values are included in an interval.

```
SELECT PlateNumber, Brand, Color, HP
FROM Car
WHERE HP BETWEEN 140 AND 170;
```

PLATENUMBER	BRAND	COLOR	HP
HD-MM 3206	Mercedes	black	170
HD-IK 1002	VW	black	160
HD-MT 507	BMW	black	140
HD-MM 1977	Mercedes	white	170

Figure 95: Comparison Operator BETWEEN ... AND



You can use search patterns in the WHERE clause.

```
SELECT PlateNumber, Brand, Color, HP
FROM Car
WHERE PlateNumber LIKE '%MM%';
```

PLATENUMBER	BRAND	COLOR	HP
HD-MM 3206	Mercedes	black	170
HD-MM 208	BMW	green	184
HD-MM 1977	Mercedes	white	170

Figure 96: Comparison Operator LIKE



The wildcard character % (percentage sign) represents any string containing zero, one, or multiple characters.

The wildcard character \_ (underscore) represents any single character.

<b>WHERE MyColumn LIKE 'M%'</b>	String starting with "M"
<b>WHERE MyColumn LIKE 'M_'</b>	String of two characters starting with "M"
<b>WHERE MyColumn LIKE '%M'</b>	String ending with "M"
<b>WHERE MyColumn LIKE '%M%</b>	String containing "M"
<b>WHERE MyColumn LIKE '___'</b>	String of three characters
<b>WHERE MyColumn LIKE '____T_M%</b>	String with "T" in the fifth and "M" in the seventh position

Figure 97: Wildcards in the LIKE Predicate



If you want to search for the percentage sign (%) or underscore (\_) itself, you have to place an **ESCAPE** character in front.  
You can choose the **ESCAPE** character (with some restrictions).

<code>LIKE '\$%%' ESCAPE '\$'</code>	String starting with a percentage sign
<code>LIKE '\$__' ESCAPE '\$'</code>	String of two characters starting with an underscore
<code>LIKE '%\$%' ESCAPE '\$'</code>	String ending with a percentage sign
<code>LIKE '%\$%' ESCAPE '\$'</code>	String containing a percentage sign
<code>LIKE '\$\$_\$%' ESCAPE '\$'</code>	String containing an underscore followed by a percentage sign
<code>LIKE '____\$%\$_%%' ESCAPE '\$'</code>	"%" in the fifth and "_" in the seventh position



Figure 98: Escaping Wildcards in the LIKE Predicate



- The license plate started with "HD" for Heidelberg
- I'm sure I saw an "M"
- I clearly remember the digits "2" and "6"
- The "2" was definitely left of the "6"
- Between the "2" and the "6" was exactly one digit
- It was neither Skoda, nor a VW
- The car was blue or green

CARID	PLATENUMBER	BRAND	COLOR	HP	OWNER
-----	-----	-----	-----	---	-----
F07	HD-ML 3206	Audi	blue	116	H03

```

SELECT *
  FROM Car
 WHERE PlateNumber LIKE 'HD-%M% __ 6%'
   AND Brand <> 'Skoda' AND Brand <> 'VW'
   AND (Color = 'blue' OR Color = 'green');
  
```



Figure 99: A Complex WHERE Clause



**The operators have the precedence indicated by the table below:**

Precedence	Operator	Explanation
Highest	( )	parentheses
	-	unary minus
	*, /	multiplication, division
	+, -	addition, subtraction
	=, <, <=, >, >=, <>, IS NULL, LIKE, BETWEEN	comparison
	NOT	logical negation
	AND	conjunction
Lowest	OR	disjunction

Figure 100: Operator Precedence



## LESSON SUMMARY

You should now be able to:

- Write simple database queries using SQL's SELECT statement and project columns in and out of queries using the SELECT clause
- Calculate column values, use built-in functions and the CASE expression in column lists
- Avoid duplicates in SELECT statement result sets
- Limit results sets to a given number of rows and browse through result sets
- Ensure a specific order in result sets
- Restrict the result set using the WHERE clause



# Understanding NULL Values

## LESSON OVERVIEW

This lesson explains what NULL values are and why dealing with them correctly is important.



## LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Interpret NULL values in databases and understand why their presence can lead to unexpected query results

## NULL Values: Interpretation and Origin

A NULL value can be used to express the following:



- The corresponding value exists in principle but is unknown temporarily or permanently for the row at hand (such as a birthday of a person).
- The corresponding value does not exist, for example the owner of an unregistered car.
- NULL is a special value reserved by the DBMS.
  - For numeric data types, NULL does not equal 0.
  - For string-based data types, NULL does not equal an empty string or a space character.

## NULL Values: Origin

NULL values can result in the following cases:



- When reading rows (`SELECT` statement)
  - If the table queried from already contains NULL values
  - When using outer joins
- When inserting or updating rows (`INSERT` and `UPDATE` statements)
  - If no value for a column is provided, NULL values are allowed and no default value is defined
  - If a NULL value is explicitly specified for a column and NULL values are allowed
- When adding a column to a table (`ALTER TABLE` statement)
  - If the table has at least one row, NULL values are allowed for the added column and no default value is defined for the added column



**By allowing NULL values, a third truth value is necessary.  
In addition to “true” and “false” there is also the truth value  
“unknown” in SQL .**

**Any comparison with a NULL value results in the truth value  
“unknown”.**

- This applies to all comparison operators:  $=$ ,  $>$ ,  $\geq$ ,  $<$ ,  $\leq$ ,  $\neq$

A	A = 7	A $\neq$ 7	A $\geq$ 6
5	false	true	false
7	true	false	true
NULL	unknown	unknown	unknown

Figure 101: NULL Values Lead to Trivalent Logic



**Two value logic is a subset of three value logic.**

**When NULL values are eliminated, the AND test simplifies.**

X	Y	X AND Y
true	true	true
true	false	false
false	true	false
false	false	false
true	unknown	unknown
false	unknown	false
unknown	true	unknown
unknown	false	false
unknown	unknown	unknown

Figure 102: Trivalent Logic Extends Logical Operators. Example: AND



**Two value logic is a subset of three value logic.**

**When NULL values are eliminated, the OR test simplifies.**

X	Y	X OR Y
true	true	true
true	false	true
false	true	true
false	false	false
true	unknown	true
false	unknown	unknown
unknown	true	true
unknown	false	unknown
unknown	unknown	unknown

Figure 103: What Makes Trivalent Logic Difficult?

The introduction of the logical value “unknown” and the way expressions are evaluated by first evaluating their atomic sub expressions leads to an unexpected result: Logical expressions that always evaluate to true in usual programming languages and according to common sense no longer evaluate to true in all cases. The following list shows some examples:

- $A = A$
- $(A < 5) \text{ OR } (A \geq 5)$
- $0 * A = 0$
- $2 * A = A + A$
- $A \text{ OR } (\text{NOT } A)$
- $\text{MAX}(\dots) \geq \text{ALL } (\text{<set of all values>})$
- $\text{MIN}(\dots) \leq \text{ALL } (\text{<set of all values>})$



**Do the following two SQL queries have the same result?**

```
SELECT Name, Overtime
  FROM Official
 WHERE Overtime <= 10 OR Overtime > 10;
```

```
SELECT Name, Overtime
  FROM Official;
```

Figure 104: Do these Queries have the Same Result?



The two SQL queries have **different results!**

- Only officials where Overtime IS NOT NULL applies:

```
SELECT Name, Overtime
  FROM Official
 WHERE Overtime <= 10 OR Overtime > 10;
```

NAME	OVERTIME
Mr A	10
Mr B	10
Ms C	20
Mr E	10
Mr F	18
Ms G	22

- All officials (including those with: Overtime IS NULL)

```
SELECT Name, Overtime
  FROM Official;
```

NAME	OVERTIME
Mr A	10
Mr B	10
Ms C	20
Ms D	?
Mr E	10
Mr F	18
Ms G	22
Ms H	?
Mr I	?



Figure 105: The Queries have Different Results!



## LESSON SUMMARY

You should now be able to:

- Interpret NULL values in databases and understand why their presence can lead to unexpected query results

## Aggregating Data

### LESSON OVERVIEW

This lesson covers how you can perform certain calculations on a database table column across multiple rows of the database table.



### LESSON OBJECTIVES

After completing this lesson, you will be able to:

- List the most important aggregate functions supported by HANA and use them to determine aggregated values on table columns using a single SELECT statement
- Determine aggregated values for groups of rows, using the GROUP BY clause
- Filter groups using the HAVING clause

### Calculations Across Multiple Rows: Aggregation

Function expressions allow the values of columns to be calculated based on the values of other columns on a row-by-row basis. But in many cases, you may want to calculate a single value across the values of multiple rows in one or more columns. You can use aggregate expressions for this purpose.



Table 6: Aggregate Expressions Supported by SAP HANA

The following table lists a selection of commonly-used aggregate functions supported by SAP HANA. For other, less-commonly used aggregate functions, please refer to SAP HANA documentation.

Aggregate Name	Description
COUNT	Returns the number of rows.
MIN, MAX	Returns the minimum or maximum value of an input column with a numeric data type.
SUM, AVG	Returns the sum or the arithmetic mean value of an input column with a numeric data type.



## Count (\*)

You can calculate the number of rows in the result set using **COUNT**.

Rows containing only **NULL** values are included.

- What is the quantity of cars with the brand “Audi”?

```
SELECT COUNT(*)
  FROM Car
 WHERE Brand = 'Audi';
```

```
COUNT(*)  
-----  
4
```

Figure 106: COUNT(\*) Expression



## Count (<column>)

You can calculate the number of values within a single column.

NULL values are not included.

You can only use a single column as parameter of **COUNT**.

- How many cars are registered to an owner?
- This does not calculate the number of different owners!

```
SELECT COUNT(Owner)
  FROM Car;
```

```
COUNT (OWNER)  
-----  
18
```

Figure 107: COUNT(<column>) Expressions



## Count (DISTINCT <column>)

You can calculate the number of distinct, non-NULL values of a certain column.

You can use only a single column as parameter for **COUNT DISTINCT**.  
NULL values are not included.

- How many different owners have a registered car?

```
SELECT COUNT(DISTINCT Owner)
  FROM Car;
```

```
COUNT (DISTINCT OWNER)  
-----  
8
```

Figure 108: COUNT(DISTINCT <column>) Expression



<code>SELECT * FROM T;</code>	<code>S — ? ? ? X X</code>
<code>SELECT S FROM T;</code>	
<code>SELECT COUNT(*) FROM T;</code>	<code>COUNT(*) ----- 5</code>
<code>SELECT COUNT(S) FROM T;</code>	<code>COUNT(S) ----- 2</code>
<del><code>SELECT COUNT(DISTINCT *) FROM T;</code></del>	
<del><code>SELECT COUNT(DISTINCT S) FROM T;</code></del>	<code>COUNT(DISTINCT S) ----- 1</code>

Figure 109: Differences between the COUNT Expressions

The DISTINCT modifier may also be used with the SUM or AVERAGE aggregates, but not with the MAX or MIN aggregates.



### MIN/MAX (<column>)

You can calculate the minimum or maximum value in a column.

- What is the horsepower range of the registered cars?

```
SELECT MIN(HP) , MAX(HP)  
FROM Car;
```

MIN (HP)	MAX (HP)
75	260

Figure 110: MIN and MAX Expressions



You can combine aggregate expressions and “normal” functions.

- In which year was the youngest owner born?

```
SELECT MAX(YEAR(Birthday)) AS Year  
FROM Owner;
```

```
SELECT YEAR(MAX(Birthday)) AS Year  
FROM Owner;
```

YEAR
1986

Figure 111: Combining Aggregate and Function Expressions

Note that, in general, the order in which functions and aggregate expressions are nested matters. As an example, the following two SELECT statements lead to different results:

```
SELECT ABS( MAX(0-HP) ) FROM Car; -- Result: 75  
SELECT MAX( ABS(0-HP) ) FROM Car; -- Result: 260
```



**The WHERE clause may be included with aggregate functions.**

**Aggregates are evaluated after the WHERE condition**

- What is the horsepower range of BMWs?

```
SELECT MIN(HP) , MAX(HP)
  FROM Car
 WHERE Brand = 'BMW';
```

MIN (HP)	MAX (HP)
140	184



Figure 112: Aggregate Expressions



### SUM/AVG (<column>)

You can sum up the values in a column or determine their arithmetic mean.

The WHERE clause is evaluated before aggregation.

Individual NULL values contained in the column do not result in NULL for the sum or average (as long as there is at least one numeric value).

- What is the sum and the mean of horsepower for all Audis?

```
SELECT SUM(HP) , AVG(HP)
  FROM Car
 WHERE Brand = 'Audi';
```

SUM (HP)	AVG (HP)
744	186



Figure 113: SUM(<column>) and AVG(<column>) Expressions

## The GROUP BY Clause

The examples, so far, all calculated a single value across all rows of a table that satisfied the WHERE condition of the SELECT statement. Sometimes, you may need such aggregations, not for all rows in total, but to produce different values for different subsets. This is where grouping and the GROUP BY clause comes into play.



Note:

In SAP HANA SQL, you have to explicitly include each column in the GROUP BY clause that is not used in an aggregate expression.



**NULL values of the GROUP BY column are treated as “normal” values creating a single group.**

- How often does each unique overtime value occur?

```
SELECT Overtime, COUNT(*) AS Frequency
  FROM Official
 GROUP BY Overtime;
```

OVERTIME	FREQUENCY
10	3
20	1
→ ?	3
18	1
22	1

Figure 114: Treatment of NULL Values



**You can combine GROUP BY with ORDER BY for sorting.**

```
SELECT Brand, MAX(HP)
  FROM Car
 GROUP BY Brand
 ORDER BY 2 DESC, 1 ASC;
```

BRAND	MAX (HP)
Audi	260
BMW	184
Mercedes	170
VW	160
Renault	136
Skoda	136
Opel	120
Fiat	75

Figure 115: Grouping and Sorting



**You can use functions in the GROUP BY clause.**

- What is the number of owners per year of birth?

```
SELECT YEAR(Birthday), COUNT(*)
  FROM Owner
 GROUP BY YEAR(Birthday)
 ORDER BY 2 DESC, 1 ASC;
```

YEAR(BIRTHDAY)	COUNT (*)
?	3
1986	3
1934	1
1952	1
1957	1
1966	1

Figure 116: Combining Functions and Grouping



You can use a combination of columns in the GROUP BY clause.

```
SELECT Brand, Color, COUNT(*)
  FROM Car
GROUP BY Brand, Color;
```

BRAND	COLOR	COUNT (*)
Fiat	red	1
VW	black	3
BMW	blue	1
Mercedes	white	2
Mercedes	black	1
Audi	yellow	1
Audi	blue	1
Skoda	red	2
BMW	black	1
BMW	green	1
Renault	red	2
Skoda	black	1
Opel	green	1
Audi	orange	1
Audi	green	1



Figure 117: Grouping by Several Columns

## The HAVING Clause

When using grouping, you can discard some of the resulting groups in a way similar to how you exclude rows from a non-aggregated result set. The corresponding keyword is **HAVING**. The **HAVING** clause behaves similar to the **WHERE** clause, but affects the resulting groups instead of the rows when calculating the aggregate values.



Using the **HAVING** clause you can specify which conditions a group must meet to be included in the result set.

- Note that the **HAVING** clause is evaluated after the aggregate groups are calculated.
- Which combinations of brand and color occur at least twice?

```
SELECT Brand, Color, COUNT(*)
  FROM Car
GROUP BY Brand, Color
HAVING COUNT(*) >= 2;
```

BRAND	COLOR	COUNT (*)
VW	black	3
Mercedes	white	2
Skoda	red	2
Renault	red	2



Figure 118: HAVING



- What is the number of cars per brand that are black or red?
- Rename the Brand column to “Manufacturer”.
- Display only those manufacturers with at least 2 cars.
- Sort the result set first – descending by number of cars.
- Sort the result set second – ascending by manufacturer.

```
SELECT c.Brand AS "Manufacturer", COUNT(*)
  FROM Car c
 WHERE Color IN ('black', 'red')
GROUP BY Brand
 HAVING COUNT(*) >= 2
ORDER BY 2 DESC, Brand ASC;
```

Manufacturer	COUNT(*)
Skoda	3
VW	3
Renault	2

Figure 119: SELECT statement



## LESSON SUMMARY

You should now be able to:

- List the most important aggregate functions supported by HANA and use them to determine aggregated values on table columns using a single SELECT statement
- Determine aggregated values for groups of rows, using the GROUP BY clause
- Filter groups using the HAVING clause



# Understanding Unions and Joins

## LESSON OVERVIEW

This lesson covers how data can be retrieved and combined from several tables in a single statement, using UNIONs and JOINs.



## LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Read data from multiple tables
- List the various types of JOIN constructs and use the appropriate JOIN construct to combine data from several tables using a single query

## UNIONs and JOINs Overview

The SELECT statement is more flexible than has been shown so far. It can be used to combine data from one or more tables. This may involve combining rows or columns in novel ways.

- You can combine the results of several partial queries to form the union of the results, or determine their intersection or difference:
  - UNION ALL and UNION merge the results of partial queries to form their union.
  - INTERSECT is used to determine the elements contained in all partial query results.
  - EXCEPT is used to determine the elements contained in one partial result, but not the second partial result.
- You can use JOIN to combine data from several tables in common rows of a result set:
  - CROSS JOIN, INNER JOIN, and OUTER JOIN are the three types of JOINs. They are explained in the following sections.
  - The two syntax options, implicit JOIN and explicit JOIN, are available.

## UNION and UNION ALL



You can combine the result sets of multiple queries using UNION [ALL].

- The individual result sets must have the same number of columns.
- The corresponding result columns must have compatible data types.
- The column names of the resulting output table are based on the first SELECT statement.

```
SELECT PNr, Name
  FROM Official
 WHERE Salary = 'A09'
UNION ALL
SELECT OwnerID, Name
  FROM Owner
 WHERE Birthday >= '1977-05-21';
```

PNR	NAME
P01	Mr A
P03	Ms C
P06	Mr F
H08	Mr X
H09	Ms Y
H10	Mr Z



Figure 120: UNION [ALL]



If the results of multiple partial queries overlap, the UNION ALL operator includes duplicates.

```
SELECT PlateNumber, Brand, Color
  FROM Car
 WHERE Brand = 'Mercedes'
UNION ALL
SELECT PlateNumber, Brand, Color
  FROM Car
 WHERE Color = 'white';
```

PLATENUMBER	BRAND	COLOR
HD-AL 1002	Mercedes	white
HD-MM 3206	Mercedes	black
HD-MM 1977	Mercedes	white
HD-AL 1002	Mercedes	white
HD-MM 1977	Mercedes	white

Duplicates



Figure 121: UNION ALL Can Lead to Duplicates



You use UNION instead of UNION ALL to eliminate duplicates.

```
SELECT PlateNumber, Brand, Color
  FROM Car
 WHERE Brand = 'Mercedes'
UNION
SELECT PlateNumber, Brand, Color
  FROM Car
 WHERE Color = 'white';
```

PLATENUMBER	BRAND	COLOR
HD-AL 1002	Mercedes	white
HD-MM 3206	Mercedes	black
HD-MM 1977	Mercedes	white

Figure 122: Avoiding Duplicates Using UNION



You can use UNION [ALL] to combine result tables of multiple queries.

```
SELECT PlateNumber, Brand, Color
  FROM Car
 WHERE Brand = 'BMW'
UNION
SELECT PlateNumber, Brand, Color
  FROM Car
 WHERE Color = 'yellow'
UNION
SELECT PlateNumber, Brand, Color
  FROM Car
 WHERE Color = 'orange';
```

PLATENUMBER	BRAND	COLOR
HD-JA 1972	BMW	blue
HD-MT 507	BMW	black
HD-MM 208	BMW	green
HD-VW 1999	Audi	yellow
HD-Y 333	Audi	orange

Figure 123: UNION [ALL] Can Handle More Than Two SELECTs

## INTERSECT and EXCEPT



**INTERSECT** returns records that exist in all query results.

```
SELECT PlateNumber, Brand, Color
      FROM Car
     WHERE Brand = 'Mercedes'
INTERSECT
SELECT PlateNumber, Brand, Color
      FROM Car
     WHERE Color = 'white';
```

PLATENUMBER	BRAND	COLOR
HD-AL 1002	Mercedes	white
HD-MM 1977	Mercedes	white



Figure 124: INTERSECT [DISTINCT]



**EXCEPT** returns results from the first query that are NOT available in subsequent queries.

```
SELECT PlateNumber, Brand, Color
      FROM Car
     WHERE Brand = 'Mercedes'
EXCEPT
SELECT PlateNumber, Brand, Color
      FROM Car
     WHERE Color = 'white';
```

PLATENUMBER	BRAND	COLOR
HD-MM 3206	Mercedes	black



Figure 125: EXCEPT [DISTINCT]

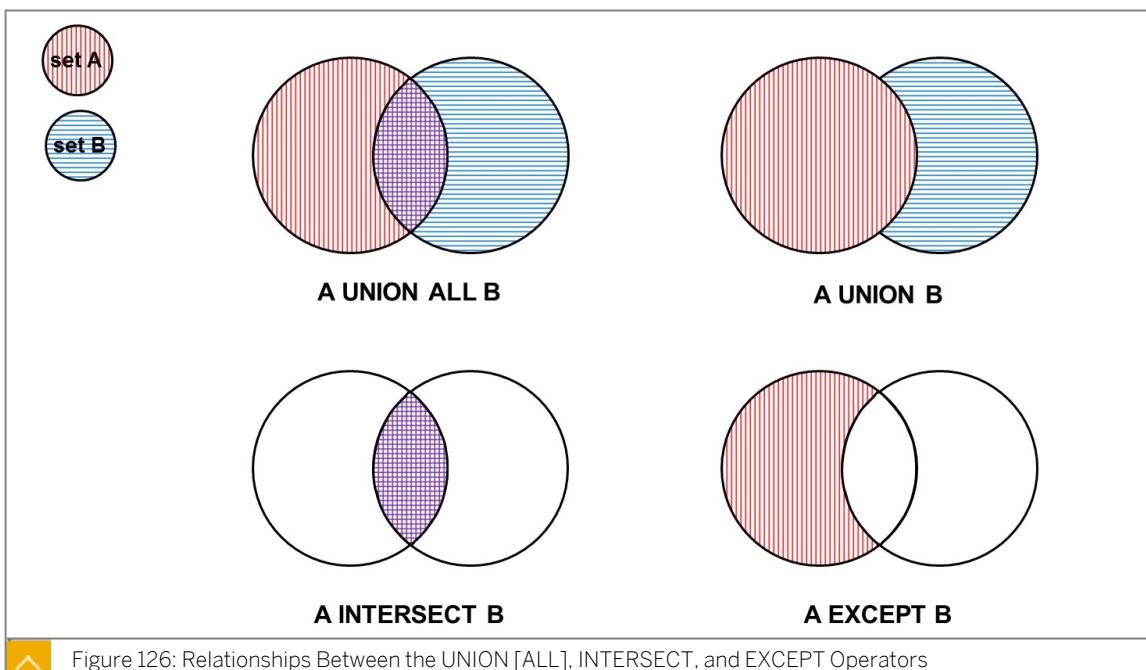


Figure 126: Relationships Between the UNION [ALL], INTERSECT, and EXCEPT Operators

### Cross Join

You can use a **cross join**, also called **Cartesian product**, to create a result set in which each row of each partial query involved is combined with each row of the other partial queries involved.



#### Note:

Cross join results are frequently not meaningful by themselves. For example, it is not helpful to list out every car as being registered simultaneously by every owner.



- Each row of the left table is connected to each row of the right table.
- No logical condition is required with the CROSS JOIN.

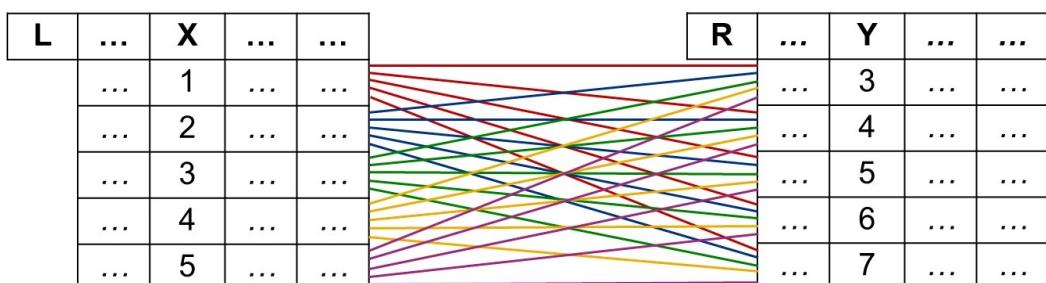


Figure 127: CROSS JOIN: Explanation

Joins may be specified with two different syntaxes: implicit and explicit. Although you will probably encounter implicit joins, the explicit syntax allows for more control, and is considered more correct. While a few implicit examples will be shown, you should focus on the explicit syntax.



**Each row of a table is connected to each row of the other table.**

- The result set contains  $10 * 20 = 200$  rows

```
SELECT *
  FROM Owner, Car;
```

OWNERID	NAME	BIRTHDAY	CITY	CARID	PLATENUMBER	BRAND	COLOR	HP	OWNER
H01	Ms T	1934-06-20	Wiesloch	F01	HD-V 106	Fiat	red	75	H06
H02	Ms U	1966-05-11	Hockenheim	F01	HD-V 106	Fiat	red	75	H06
H03	SAP AG	?	Walldorf	F01	HD-V 106	Fiat	red	75	H06
...	...	...	...	...	...	...	...	...	...
H01	Ms T	1934-06-20	Wiesloch	F02	HD-VW 4711	VW	black	120	H03
H02	Ms U	1966-05-11	Hockenheim	F02	HD-VW 4711	VW	black	120	H03
H03	SAP AG	?	Walldorf	F02	HD-VW 4711	VW	black	120	H03
...	...	...	...	...	...	...	...	...	...
H01	Ms T	1934-06-20	Wiesloch	F03	HD-JA 1972	BMW	blue	184	H03
H02	Ms U	1966-05-11	Hockenheim	F03	HD-JA 1972	BMW	blue	184	H03
H03	SAP AG	?	Walldorf	F03	HD-JA 1972	BMW	blue	184	H03
...	...	...	...	...	...	...	...	...	...
H10	Mr Z	1986-02-03	Ladenburg	F20	?	Audi	green	184	?

Figure 128: CROSS JOIN with Implicit Syntax



**Each row of a table is connected to each row of the other table.**

- The result set contains  $10 * 20 = 200$  rows

```
SELECT *
  FROM Owner CROSS JOIN Car;
```

OWNERID	NAME	BIRTHDAY	CITY	CARID	PLATENUMBER	BRAND	COLOR	HP	OWNER
H01	Ms T	1934-06-20	Wiesloch	F01	HD-V 106	Fiat	red	75	H06
H02	Ms U	1966-05-11	Hockenheim	F01	HD-V 106	Fiat	red	75	H06
H03	SAP AG	?	Walldorf	F01	HD-V 106	Fiat	red	75	H06
...	...	...	...	...	...	...	...	...	...
H01	Ms T	1934-06-20	Wiesloch	F02	HD-VW 4711	VW	black	120	H03
H02	Ms U	1966-05-11	Hockenheim	F02	HD-VW 4711	VW	black	120	H03
H03	SAP AG	?	Walldorf	F02	HD-VW 4711	VW	black	120	H03
...	...	...	...	...	...	...	...	...	...
H01	Ms T	1934-06-20	Wiesloch	F03	HD-JA 1972	BMW	blue	184	H03
H02	Ms U	1966-05-11	Hockenheim	F03	HD-JA 1972	BMW	blue	184	H03
H03	SAP AG	?	Walldorf	F03	HD-JA 1972	BMW	blue	184	H03
...	...	...	...	...	...	...	...	...	...
H10	Mr Z	1986-02-03	Ladenburg	F20	?	Audi	green	184	?

Figure 129: CROSS JOIN with Explicit Syntax



A WHERE clause may be specified to reduce the rows returned from a join.

```
SELECT *
  FROM Owner, Car
 WHERE HP > 250;
```

```
SELECT *
  FROM Owner CROSS JOIN Car
 WHERE HP > 250;
```

OWNERID	NAME	BIRTHDAY	CITY	CARID	PLATENUMBER	BRAND	COLOR	HP	OWNER
H01	Ms T	1934-06-20	Wiesloch	F06	HD-VW 1999	Audi	yellow	260	H05
H02	Ms U	1966-05-11	Hockenheim	F06	HD-VW 1999	Audi	yellow	260	H05
H03	SAP AG	?	Walldorf	F06	HD-VW 1999	Audi	yellow	260	H05
H04	HDM AG	?	Heidelberg	F06	HD-VW 1999	Audi	yellow	260	H05
H05	Mr V	1952-04-21	Leimen	F06	HD-VW 1999	Audi	yellow	260	H05
H06	Ms W	1957-06-01	Wiesloch	F06	HD-VW 1999	Audi	yellow	260	H05
H07	IKEA	?	Walldorf	F06	HD-VW 1999	Audi	yellow	260	H05
H08	Mr X	1986-08-30	Walldorf	F06	HD-VW 1999	Audi	yellow	260	H05
H09	Ms Y	1986-02-10	Sinsheim	F06	HD-VW 1999	Audi	yellow	260	H05
H10	Mr Z	1986-02-03	Ladenburg	F06	HD-VW 1999	Audi	yellow	260	H05

Figure 130: CROSS JOIN and WHERE Clause



Adding tables to the CROSS JOIN quickly increases the number of result rows.

- The result set contains  $10 * 20 * 3 = 600$  rows

```
SELECT *
  FROM Owner,
    Car,
      Stolen;
```

```
SELECT *
  FROM Owner CROSS JOIN
    Car CROSS JOIN
      Stolen;
```

OWNERID	NAME	BIRTHDAY	CITY	CARID	PLATENUMBER	BRAND	COLOR	HP	OWNER	PLATENUMBER	REGISTERED_AT
H01	Ms T	1934-06-20	Wiesloch	F01	HD-V 106	Fiat	red	75	H06	HD-VW 1999	2012-06-20
H01	Ms T	1934-06-20	Wiesloch	F01	HD-V 106	Fiat	red	75	H06	HD-V 106	2012-06-01
H01	Ms T	1934-06-20	Wiesloch	F01	HD-V 106	Fiat	red	75	H06	HD-Y 333	2012-05-21
...	...	...	...	...	...	...	...	...	...	...	...
H01	Ms T	1934-06-20	Wiesloch	F02	HD-VW 4711	VW	black	120	H03	HD-VW 1999	2012-06-20
H01	Ms T	1934-06-20	Wiesloch	F02	HD-VW 4711	VW	black	120	H03	HD-V 106	2012-06-01
H01	Ms T	1934-06-20	Wiesloch	F02	HD-VW 4711	VW	black	120	H03	HD-Y 333	2012-05-21
...	...	...	...	...	...	...	...	...	...	...	...
H02	Ms U	1966-05-11	Hockenheim	F01	HD-V 106	Fiat	red	75	H06	HD-VW 1999	2012-06-20
H02	Ms U	1966-05-11	Hockenheim	F01	HD-V 106	Fiat	red	75	H06	HD-V 106	2012-06-01
H02	Ms U	1966-05-11	Hockenheim	F01	HD-V 106	Fiat	red	75	H06	HD-Y 333	2012-05-21
...	...	...	...	...	...	...	...	...	...	...	...
H10	Mr Z	1986-02-03	Ladenburg	F20	?	Audi	green	184	?	HD-Y 333	2012-05-21

Figure 131: CROSS JOIN

You can also cross-join a table with itself.



Note:

In general, a cross join can quickly lead to a very large result set because its cardinality (number of rows) equals the product of the cardinalities of the queries of which it is comprised. It can be easy to produce a cross join by mistake, because its syntax looks very similar to that of the inner join.

## Table Aliases

Different tables can have columns with the same name. This can lead to ambiguities when joining such tables, for example when a column name that appears in two or more of the

tables involved is used in the projection list, the WHERE condition, or in a JOIN condition. You have to resolve such ambiguities by qualifying the non-unique column name, that is, by specifying the table that the column belongs to. You can do this using the full table name as qualifier.



**You can qualify the column name by adding the table name:**

```
SELECT Official.Name
FROM Official;
```

NAME
Mr A
Mr B
Ms C
Ms D
Mr E
Mr F
Ms G
Ms H
Mr I

Figure 132: Qualifying a Column Name Using the Table Name

Having to repeat the full table name as qualifier each time can be cumbersome. This is why you can use **table aliases** to introduce abbreviated table names and use the abbreviated form as qualifier in the same SELECT statement.



**You can use table aliases in the projection list.  
Table aliases are defined in the FROM clause.**

```
SELECT o.Name, o.PNr
FROM Official o;
```

NAME	PNR
Mr A	P01
Mr B	P02
Ms C	P03
Ms D	P04
Mr E	P05
Mr F	P06
Ms G	P07
Ms H	P08
Mr I	P09

**You can use the (optional) keyword AS  
in the definition of a table aliases:**

```
SELECT o.Name, o.PNr
FROM Official AS o;
```

Figure 133: Table Aliases



If a table alias is defined in the **FROM** clause, you are not allowed to use the corresponding table name for qualification of a column name:

~~SELECT Official.Name  
FROM Official o;~~

**SELECT Official.Name  
FROM Official;**

**SELECT o.Name  
FROM Official o;**

~~SELECT o.Name  
FROM Official;~~

**SELECT Name  
FROM Official o;**

**SELECT Name  
FROM Official;**

Figure 134: Supported Combinations of Qualified and Unqualified Column Names

### Inner Join

An **inner join** is used to combine information from one table with corresponding information from another table, and to only include such rows in the result for which corresponding information is available.



One row of the left table and one row of the right table are always joined to a common result row - provided that the **JOIN condition** is fulfilled.

- JOIN Condition:  $L.X = R.Y$

L	...	X	...	...
...	1		...	...
...	2		...	...
...	3		...	...
...	4		...	...
...	5		...	...

R	...	Y	...	...
...	3		...	...
...	4		...	...
...	5		...	...
...	6		...	...
...	7		...	...

Figure 135: Inner Join Explanation



**One row of a table and one row of another table are always connected to a common result row provided that the JOIN condition is fulfilled.**

**The implicit JOIN condition is part of the WHERE clause**

**The explicit JOIN condition is independent of the WHERE clause**

- **To whom which car is registered?**

```
SELECT *
FROM Owner, Car
WHERE OwnerID = Owner;
```

```
SELECT *
FROM Owner JOIN Car
ON OwnerID = Owner;
```

OWNERID	NAME	BIRTHDAY	CITY	CARID	PLATENUMBER	BRAND	COLOR	HP	OWNER
H06	Ms W	1957-06-01	Wiesloch	F01	HD-V 106	Fiat	red	75	H06
H03	SAP AG	?	Walldorf	F02	HD-VW 4711	VW	black	120	H03
H03	SAP AG	?	Walldorf	F03	HD-JA 1972	BMW	blue	184	H03
H07	IKEA	?	Walldorf	F04	HD-AL 1002	Mercedes	white	136	H07
H03	SAP AG	?	Walldorf	F05	HD-MM 3206	Mercedes	black	170	H03
...	...	...	...	...	...	...	...	...	...

Figure 136: Inner Join Implicit Syntax



#### Note:

The following examples show the explicit syntax option only. Using the implicit syntax is more error-prone because the join condition cannot be distinguished from additional parts of a WHERE condition. This makes it easy to forget the join condition and produce a cross join by mistake.



**Both “JOIN” and “INNER JOIN” are acceptable syntax variants**

**The INNER JOIN is the most important JOIN variant (and therefore the default)**

- **To whom is which car registered?**

```
SELECT *
FROM Owner INNER JOIN Car ON OwnerID = Owner;
```

OWNERID	NAME	BIRTHDAY	CITY	CARID	PLATENUMBER	BRAND	COLOR	HP	OWNER
H06	Ms W	1957-06-01	Wiesloch	F01	HD-V 106	Fiat	red	75	H06
H03	SAP AG	?	Walldorf	F02	HD-VW 4711	VW	black	120	H03
H03	SAP AG	?	Walldorf	F03	HD-JA 1972	BMW	blue	184	H03
H07	IKEA	?	Walldorf	F04	HD-AL 1002	Mercedes	white	136	H07
H03	SAP AG	?	Walldorf	F05	HD-MM 3206	Mercedes	black	170	H03
...	...	...	...	...	...	...	...	...	...

Figure 137: Keyword INNER is Optional



## You can specify columns in the projection list of the JOIN.

- To whom is which car registered?
- ```
SELECT Name, Brand, Color
FROM Owner JOIN Car
ON OwnerID = Owner;
```

| NAME   | BRAND    | COLOR  |
|--------|----------|--------|
| Ms W   | Fiat     | red    |
| SAP AG | VW       | black  |
| SAP AG | BMW      | blue   |
| IKEA   | Mercedes | white  |
| SAP AG | Mercedes | black  |
| Mr V   | Audi     | yellow |
| SAP AG | Audi     | blue   |
| IKEA   | VW       | black  |
| Ms U   | Skoda    | red    |
| HDM AG | BMW      | black  |
| Ms U   | BMW      | green  |
| HDM AG | Skoda    | red    |
| IKEA   | Renault  | red    |
| SAP AG | Mercedes | white  |
| SAP AG | Skoda    | black  |
| Ms Y   | Audi     | orange |
| SAP AG | Renault  | red    |
| Ms T   | VW       | black  |

Figure 138: Joins Can be Combined with Projections



## You can use table aliases in the projection list of the JOIN.

The JOIN condition can also refer to table aliases.

If column names are not unique, you must qualify them with the table aliases.

- To whom is which car registered?

```
SELECT o.Name, c.Brand, c.Color
FROM Owner o JOIN Car c
ON o.OwnerID = c.Owner;
```

| NAME   | BRAND    | COLOR  |
|--------|----------|--------|
| Ms W   | Fiat     | red    |
| SAP AG | VW       | black  |
| SAP AG | BMW      | blue   |
| IKEA   | Mercedes | white  |
| SAP AG | Mercedes | black  |
| Mr V   | Audi     | yellow |
| SAP AG | Audi     | blue   |
| IKEA   | VW       | black  |
| Ms U   | Skoda    | red    |
| HDM AG | BMW      | black  |
| Ms U   | BMW      | green  |
| HDM AG | Skoda    | red    |
| IKEA   | Renault  | red    |
| SAP AG | Mercedes | white  |
| SAP AG | Skoda    | black  |
| Ms Y   | Audi     | orange |
| SAP AG | Renault  | red    |
| Ms T   | VW       | black  |

Figure 139: Table Aliases Can be Handy for Inner Joins



You do not need to include a column from every involved table in the projection list.

In addition to the JOIN condition you can specify a WHERE clause.

- To whom is a black car registered (at least one)?

```
SELECT DISTINCT o.Name
  FROM Owner o JOIN Car c
    ON o.OwnerID = c.Owner
   WHERE c.Color = 'black';
```

| NAME   |
|--------|
| -----  |
| Ms T   |
| SAP AG |
| HDM AG |
| IKEA   |

Figure 140: Projections Can Reduce to One of the Tables Involved



You can build the JOIN condition on multiple columns.

- To whom within the EU is a black car registered (at least one)?

```
SELECT DISTINCT o.Name AS "Owners' name"
  FROM Owner_EU o JOIN Car_EU c
    ON o.Country = c.Country
   AND o.OwnerID = c.Owner
   WHERE c.Color = 'black'
  ORDER BY o.Name;
```

| Owners' name |
|--------------|
| -----        |
| HDM AG       |
| IKEA         |
| Ms O         |
| Ms T         |
| SAP AG       |
| Señora R     |

Figure 141: The JOIN Condition Can be Complex



You can join a table to itself.

So-called “self joins” will require table aliases to be used.

- Who is the manager of which employee?

```
SELECT e.Name AS Employee, m.Name AS Manager
  FROM Official e JOIN Official m
    ON e.Manager = m.PNr;
```

| EMPLOYEE | MANAGER |
|----------|---------|
| -----    | -----   |
| Mr A     | Ms D    |
| Mr B     | Ms D    |
| Ms C     | Ms D    |
| Ms D     | Mr I    |
| Mr E     | Ms H    |
| Mr F     | Ms H    |
| Ms G     | Ms H    |
| Ms H     | Mr I    |

Figure 142: Implicit INNER JOIN



**Combining more than two tables will require additional join conditions.**

To avoid cross joins, you must supply a valid join condition for each pairwise combination of tables in the join.

- To whom is which stolen car registered?

```
SELECT o.Name, c.Brand, c.Color, c.PlateNumber
  FROM Owner o
    JOIN Car c ON o.OwnerID = c.Owner
    JOIN Stolen s ON c.PlateNumber = s.PlateNumber;
```

| NAME | BRAND | COLOR  | PLATENUMBER |
|------|-------|--------|-------------|
| Ms W | Fiat  | red    | HD-V 106    |
| Mr V | Audi  | yellow | HD-VW 1999  |
| Ms Y | Audi  | orange | HD-Y 333    |

Figure 143: Combining Two Tables



You can use different comparison operators other than EQUAL in the JOIN condition.

- Which owner is older than which other owner?

```
SELECT o.Name AS "older", y.Name AS
"younger"
  FROM Owner o JOIN Owner y
    ON o.Birthday < y.Birthday;
```

| older | younger |
|-------|---------|
| Ms T  | Ms U    |
| Ms T  | Mr V    |
| Ms T  | Ms W    |
| Ms T  | Mr X    |
| Ms T  | Ms Y    |
| Ms T  | Mr Z    |
| Ms U  | Mr X    |
| Ms U  | Ms Y    |
| Ms U  | Mr Z    |
| Mr V  | Ms U    |
| Mr V  | Ms W    |
| Mr V  | Mr X    |
| Mr V  | Ms Y    |
| Mr V  | Mr Z    |
| Ms W  | Ms U    |
| Ms W  | Mr X    |
| Ms W  | Ms Y    |
| Ms W  | Mr Z    |
| Ms Y  | Mr X    |
| Mr Z  | Mr X    |
| Mr Z  | Ms Y    |

Figure 144: Use of Different Comparison Operators



You can use calculations and functions in the JOIN condition.

- Which owner was born in the same year as which other owner?
- ```
SELECT o1.Name, o1.Birthday, o2.Name, o2.Birthday
  FROM Owner o1 JOIN Owner o2
 WHERE YEAR(o1.Birthday) = YEAR(o2.Birthday)
   AND o1.OwnerID < o2.OwnerID;
```

NAME	BIRTHDAY	NAME	BIRTHDAY
Mr X	1986-08-30	Ms Y	1986-02-10
Mr X	1986-08-30	Mr Z	1986-02-03
Ms Y	1986-02-10	Mr Z	1986-02-03

Figure 145: Calculations and Functions in the JOIN Condition

## Join types

The result set of an inner join does not contain the rows of any of the involved tables for which no corresponding row can be found in the other tables involved according to the join condition. Sometimes you may want to include such rows in the result set, to get the full set of rows of one table and augment the data with data from other tables as far as available. In addition to the inner join, three types of **outer joins** exist for this purpose:

- **Left outer join:** includes all entries of the first (**left**) table even if no corresponding entries are found in the second (**right**) table.
- **Right outer join:** includes all entries of the second table even if no corresponding entries are found in the first table.
- **Full outer join:** is the union of left and right outer join.

In all three cases, the following applies:

- A NULL value results for each column and row for which no corresponding data can be retrieved according to the join condition.
- SAP HANA only supports the explicit syntax option.



### Left Outer Joins

- One row of a table and one row of another table are always connected to a common result row - provided the JOIN condition is fulfilled.
- In addition, rows of the left table without matching row in the right table are copied to the query result. The missing values (from the right table) are filled with NULL values.

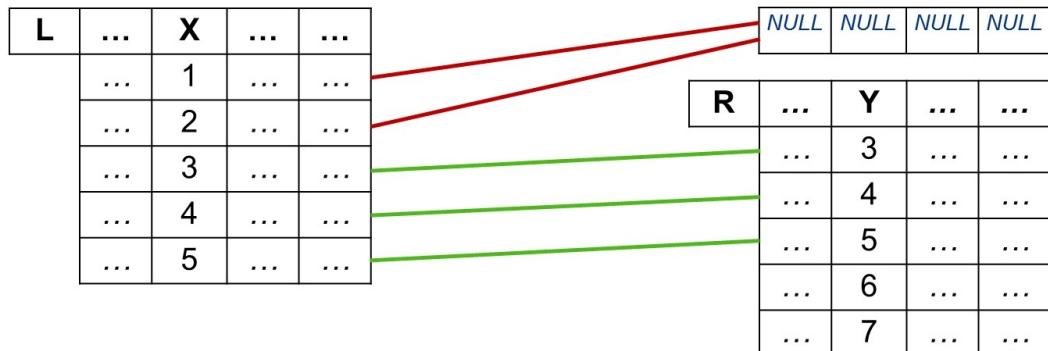


Figure 146: LEFT OUTER Join: Explanation



- Which car is registered to which individual or company?
- Individuals and companies that currently do not have a car registered should be included.
- Cars without a registration should not be included in the result.

```
SELECT Name, CarID, Brand
FROM Owner LEFT OUTER JOIN Car
ON OwnerID = Owner;
```

NAME	CARID	BRAND
Ms T	F19	VW
Ms U	F09	Skoda
Ms U	F11	BMW
SAP AG	F02	VW
SAP AG	F03	BMW
SAP AG	F18	Renault
SAP AG	F05	Mercedes
SAP AG	F15	Skoda
SAP AG	F07	Audi
SAP AG	F14	Mercedes
HDM AG	F12	Skoda
HDM AG	F10	BMW
Mr V	F06	Audi
Ms W	F01	Fiat
IKEA	F13	Renault
IKEA	F08	VW
IKEA	F04	Mercedes
→ Mr X	?	?
→ Ms Y	F17	Audi
→ Mr Z	?	?



Figure 147: LEFT OUTER Join: Example



### Right Outer Joins

- One row of a table and one row of another table are always connected to a common result row, provided that the JOIN condition is fulfilled.
- In addition, rows of the right table without matching row in the left table are copied to the query result. The missing values (from the left table) are filled with NULL values.
- Note that reversing the order of tables in the join condition AND switching the direction of the join ( $\text{LEFT} \leftrightarrow \text{RIGHT}$ ) produces the same result set.

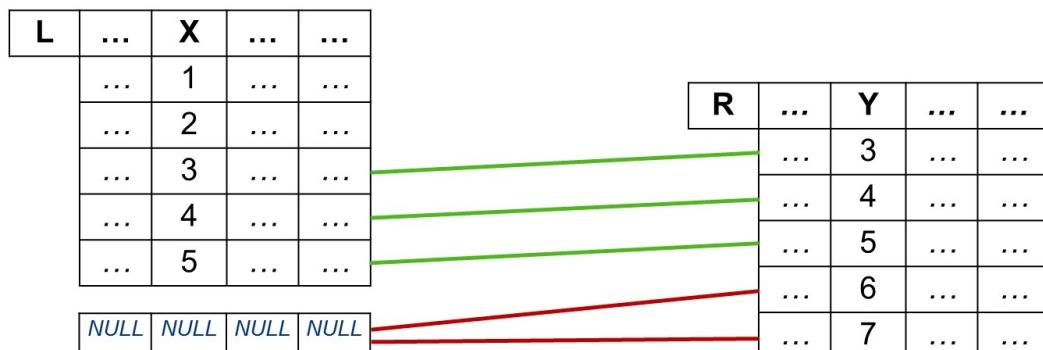


Figure 148: RIGHT OUTER Join: Explanation



- Which car is registered to which individual or company?
- Cars without a registration should be included.
- Individuals and companies that currently do not have a car registered should not be included in the result.

```
SELECT Name, CarID, Brand
  FROM Owner RIGHT OUTER JOIN Car
    ON OwnerID = Owner;
```

NAME	CARID	BRAND
Ms W	F01	Fiat
SAP AG	F02	VW
SAP AG	F03	BMW
IKEA	F04	Mercedes
SAP AG	F05	Mercedes
Mr V	F06	Audi
SAP AG	F07	Audi
IKEA	F08	VW
Ms U	F09	Skoda
HDM AG	F10	BMW
Ms U	F11	BMW
HDM AG	F12	Skoda
IKEA	F13	Renault
SAP AG	F14	Mercedes
SAP AG	F15	Skoda
→ ?	F16	Opel
Ms Y	F17	Audi
SAP AG	F18	Renault
Ms T	F19	VW
→ ?	F20	Audi

Figure 149: RIGHT OUTER Join: Example

## Full Outer Join



- One row of a table and one row of another table are always connected to a common result row, provided that the JOIN condition is fulfilled.
- In addition, rows of both tables without matching records are copied to the query result. The missing values (from the other table) are filled with NULL values.

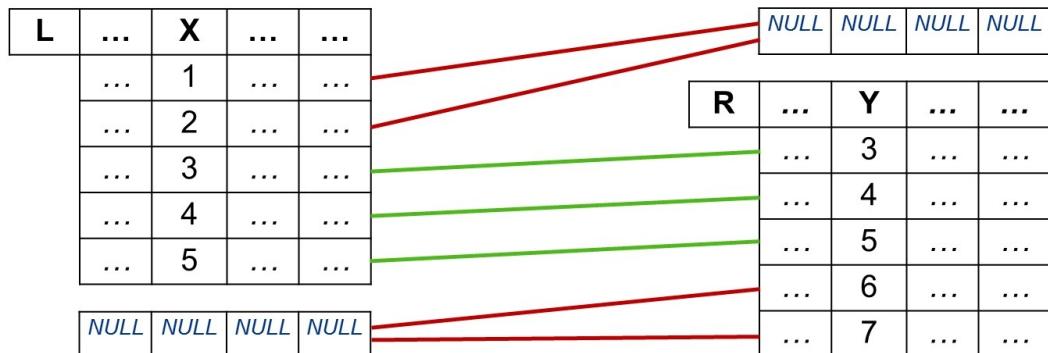


Figure 150: FULL OUTER Join: Explanation



- Which car is registered to which individual or company?
- Individuals and companies that currently do not have a car registered should be included.
- Cars without a registration should be included in the result.

```
SELECT Name, CarID, Brand
FROM Owner FULL OUTER JOIN Car
ON OwnerID = Owner;
```

NAME	CARID	BRAND
Ms W	F01	Fiat
SAP AG	F02	VW
SAP AG	F03	BMW
IKEA	F04	Mercedes
SAP AG	F05	Mercedes
Mr V	F06	Audi
SAP AG	F07	Audi
IKEA	F08	VW
Ms U	F09	Skoda
HDM AG	F10	BMW
Ms U	F11	BMW
HDM AG	F12	Skoda
IKEA	F13	Renault
SAP AG	F14	Mercedes
SAP AG	F15	Skoda
→ ?	F16	Opel
Ms Y	F17	Audi
SAP AG	F18	Renault
Ms T	F19	VW
→ ?	F20	Audi
→ Mr X	?	?
→ Mr Z	?	?

Figure 151: FULL OUTER Join: Example

Each type of outer join supports joining a table with itself. The following shows the example of a left outer join.



### Outer Joins can operate on a single table

- As with inner self joins, table aliases must be used.
- Who is the manager of which employee?
- Employees without a manager should be included.
- Only managers with employees should be included.

EMPLOYEE	MANAGER
Mr A	Ms D
Mr B	Ms D
Ms C	Ms D
Ms D	Mr I
Mr E	Ms H
Mr F	Ms H
Ms G	Ms H
Ms H	Mr I
→ Mr I	?

```
SELECT e.Name AS Employee, m.Name AS Manager
FROM Official e LEFT OUTER JOIN Official m
ON e.Manager = m.PNr;
```

Figure 152: LEFT OUTER JOIN

Replacing keyword `LEFT` with `RIGHT` or `FULL` in the example works and leads to different result sets.

- Using `RIGHT OUTER JOIN` would lead to a result including each official name in the `MANAGER` column and with only the names of the officials who have a manager assigned in the `EMPLOYEE` column.
- `USING FULL OUTER JOIN` would lead to a result including each official name in the `MANAGER` column and with each official name in the `EMPLOYEE` column.

Similar to what is supported for `CROSS JOINS` and `INNER JOINS`, you can also do the following with `OUTER JOINS`:

- Use any comparison operator in the join condition.
- Join more than two tables, and join a table with itself.
- Project the result set to a subset of columns. You do not have to include a column of every table involved in the projection list.
- Rename columns.
- Eliminate duplicates using `DISTINCT`.
- Order the result set, use aggregate expressions, a `GROUP BY` clause, or a `HAVING` clause.
- Use table aliases and reference them in the projection list and join condition. You have to qualify column names with the table names or use aliases if the column names are not unique.

More generally, any join may combine with any of the query operations already seen: column naming, results ordering, aggregates and grouped aggregates, functions and expressions, and so on. Query operations are independent and may be arbitrarily combined to produce as simple or complex a result as desired.



### LESSON SUMMARY

You should now be able to:

- Read data from multiple tables

- List the various types of JOIN constructs and use the appropriate JOIN construct to combine data from several tables using a single query



# Changing Data Stored in Tables

## LESSON OVERVIEW

This lesson covers how you can add data to a database, and change and delete data.



## LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Add rows to database tables using SQL
- Change existing rows of a database table
- Remove existing rows from a database table

## Modifying Data — Overview

In previous lessons, we examined how you can read data from a database.

The following list provides an overview of the statements used to modify data:



### INSERT

The `INSERT` statement adds rows to a database table.

### UPDATE

The `UPDATE` statement changes values of existing rows in a table.

### DELETE

The `DELETE` statement permanently removes existing rows from a table.

### UPSERT

The `UPSERT` statement changes existing rows of a table if found, otherwise it inserts the new rows. It is a combination of `INSERT` and `UPDATE`.

`REPLACE` is a synonym for `UPSERT`.

## Inserting Data

The `INSERT` statement is used to add new rows to a database table. If the table already contains rows with the same primary key values or if you try to add several rows with the same primary key values, the DBMS checks and, in such cases, raises an exception.

You can use three different syntax versions to insert data. The first two options allow you to insert single rows only, the third option allows you to insert several rows with a single statement.



### Using INSERT you can specify a value for each column.

If the sequence of the values corresponds to the sequence of columns; you do not need to list the column names.

The order of the columns is set implicitly when you create the table.

Note that unlike some SQL variants, HANA SQL requires the INTO keyword.

- Insert a new row into the Official table:

```
INSERT INTO Official
    VALUES ('P10', 'Ms J', 20, 'A10', 'P04');
```

PNR	NAME	OVERTIME	SALARY	MANAGER
---	---	-----	-----	-----
...	...	...	...	...
P10	Ms J	20	A10	P04

Figure 153: Single Row Insert, Pass Values by Column Order



### Using INSERT you can specify a value for each column.

If the sequence of the values does not match the sequence of the corresponding columns, you must list the column names in the correct sequence of the values.

- Insert a new row into the Official table:

```
INSERT INTO Official (Salary, Overtime, Name, PNr, Manager)
    VALUES ('A12', 50, 'Mr K', 'P11', 'P09');
```

PNR	NAME	OVERTIME	SALARY	MANAGER
---	---	-----	-----	-----
...	...	...	...	...
P11	Ms K	50	A12	P09

Figure 154: Single Row Insert, Pass Values by Name



### You do not have to specify a value for each column of the INSERT.

You must list the column names in the correct sequence for each column, where you specify a value.

Missing values will be replaced by NULL or the default value of the column

- Insert a new row into the Official table:

```
INSERT INTO Official (PNr, Name) VALUES ('P12', 'Ms L');
```

PNR	NAME	OVERTIME	SALARY	MANAGER
---	---	-----	-----	-----
...	...	...	...	...
P12	Ms L	?	A06	?

Figure 155: Omitting Values on Insert



You can **INSERT** data read from another table.

- **INSERT all owner IDs and names from city Wiesloch into the Official table:**

```
INSERT INTO Official(PNr, Name)
    SELECT OwnerID, Name
        FROM Owner
    WHERE City = 'Wiesloch';
```

PNR	NAME	OVERTIME	SALARY	MANAGER
---	---	-----	-----	-----
...	...	...	...	...
H01	Ms T	?	A06	?
H06	Ms W	?	A06	?

Figure 156: Inserting Multiple Rows

## Updating Data

The fundamental statement to update existing rows of a table is the **UPDATE** statement. The basic syntax of the **UPDATE** statement is as follows:

```
UPDATE <table>
    SET <column> = <value> [,,
        <column> = <value>, ...]
    [WHERE <condition>]
```



You can use the **UPDATE** statement to change only certain rows in a table by adding a **WHERE** clause.

- The overtime value of officials in salary group “A09” should be doubled:

```
UPDATE Official
    SET Overtime = Overtime * 2
    WHERE Salary = 'A09';
```

PNR	NAME	OVERTIME	SALARY	MANAGER
---	---	-----	-----	-----
P01	Mr A	20	A09	P04
P02	Mr B	10	A10	P04
P03	Ms C	40	A09	P04
P04	Ms D	?	A12	P09
...	...	...	...	...

Figure 157: Update Example



Note:

The **WHERE** clause is optional. However, omitting the **WHERE** clause updates **all** rows of the table, which may not be the intention.

In practice, the WHERE clause often specifies the primary key values of the rows to be updated. Note that “missing rows” do not lead to an exception, they lead to no row being updated.



### You can use the UPDATE statement to change values of multiple columns.

- The overtime value of officials in salary group “A09” should be tripled. In addition their salary group should be changed to “A10”.

```
UPDATE Official
    SET Overtime = Overtime * 3,
        Salary = 'A10'
WHERE Salary = 'A09';
```

PNR	NAME	OVERTIME	SALARY	MANAGER
---	---	-----	-----	-----
P01	Mr A	30	A10	P04
P02	Mr B	10	A10	P04
P03	Ms C	60	A10	P04
...	...	...	...	...

Figure 158: Updating Several Columns

The WHERE clause supports the same features as the SELECT statement, for example sub queries.



### You can use an uncorrelated sub query on the WHERE clause of the UPDATE statement.

- Officials who are contacts for the car owners are moved to salary group “A15”.

```
UPDATE Official
    SET Salary = 'A15'
WHERE PNr IN (SELECT PersNumber
        FROM Contact);
```

PNR	NAME	OVERTIME	SALARY	MANAGER
---	---	-----	-----	-----
...	...	...	...	...
P07	Ms G	22	A11	P08
P08	Ms H	?	A15	P09
P09	Mr I	?	A15	?

Figure 159: The WHERE Clause Works as Usual

You can also use correlated sub queries, such as functions in the WHERE clause.



Usually a sub query is part of the WHERE clause.

You can also use a (correlated or uncorrelated) sub query in the SET clause.

For the following SQL statement, we assume that the table, Car, has an additional column, Ownername.

```
UPDATE Car c
SET Ownername = (SELECT o.Name
                  FROM Owner o
                  WHERE o.OwnerID =
                        c.Owner);
```

CARID	PLATENUMBER	BRAND	COLOR	HP	OWNER	OWNERNAME
F01	HD-V 106	Fiat	red	75	H06	Ms W
F02	HD-VW 4711	VW	black	120	H03	SAP AG
F03	HD-JA 1972	BMW	blue	184	H03	SAP AG
...	...	...	...	...	...	...

Figure 160: Sub Queries also Work in the SET Clause

## The UPSERT/REPLACE Statement

In addition to the INSERT and the UPDATE statements, SAP HANA also supports an UPSERT statement. This statement either updates rows in a table or inserts new rows.

### UPSERT Statement Syntax

Due to its support for inserting data, the UPSERT statement comes in three syntax variants similar to the INSERT syntax variants:

- UPSERT Table(Column List) VALUES( Value List ) WHERE Condition

If the condition evaluates to true, the matching rows are updated. If not, a new row with the values provided is inserted.

- UPSERT Table(Column List) VALUES( Value List ) WITH PRIMARY KEY

Updates the row with the primary key contained in the value list if present. If not, a new row with the values provided is inserted.

- UPSERT Table(Column List) SELECT ... FROM ... WHERE ...

For each row returned by the sub query, it checks if the table contains a row with the same primary key value. If yes, the existing row is updated with the row returned by the sub query. If not, a new row with the values provided is inserted.

The first variant looks similar to the INSERT statement, with the key word changed to UPSERT and a WHERE clause added.

The third variant looks like the INSERT statement in its set variant and behaves in much the same way.

Alternatively, you can also use the REPLACE keyword. UPSERT and REPLACE are truly synonyms.



**UPSERT statements may act as either an INSERT or an UPDATE, depending on the existing data in the table.**

The Employee table originally contains the rows:

DNUMBER	NAME	DATEOFBIRTH	REMAINDERDAYS	DEPID
D100001	Mr A	Jan 31, 1972	10	A01
D100002	Ms B	Feb 29, 1972	15	A01
D100003	Mr C	Mar 31, 1972	20	A01
D100004	Ms D	Apr 30, 1972	10	A02
D100005	Ms E	May 31, 1972	10	A02
D100006	Mr F	Jun 30, 1972	40	A03
D100007	Ms G	?	?	?
D100008	Mr H	?	?	?
D100009	Ms I	Apr 1, 1972	10	A04
D100010	Ms J	Apr 30, 1972	0	A04
D100011	Mr K	May 1, 1972	20	A04
D100012	Mr L	May 31, 1972	20	A04



Figure 161: UPSERT Statements



**UPSERT Employee**

```
VALUES ('D100001', 'Mr X', '1975-01-31', 10, 'A01')
WHERE DNumber = 'D100001';
```

**UPSERT Employee**

```
VALUES ('D100099', 'Ms X', '1975-01-31', 10, 'A01')
WHERE DNumber = 'D100099';
```

DNUMBER	NAME	DATEOFBIRTH	REMAINDERDAYS	DEPID
D100001	Mr X	Jan 31, 1975	10	A01
D100002	Ms B	Feb 29, 1972	15	A01
D100003	Mr C	Mar 31, 1972	20	A01
D100004	Ms D	Apr 30, 1972	10	A02
...	...	...	...	...
D100012	Mr L	May 31, 1972	20	A04
D100099	Ms X	Jan 31, 1975	10	A01



Figure 162: UPSERT Values

## Deleting Data

The fundamental statement to delete existing rows of a table is the **DELETE** statement. The basic syntax for the **DELETE** statement is as follows:

```
DELETE FROM <table>
[WHERE <condition>]
```

**You can DELETE selected rows from a table:**

- Delete all cars with horsepower less than 180:

```
DELETE
  FROM Car
 WHERE HP < 180;
```

CARID	PLATENUMBER	BRAND	COLOR	HP	OWNER
F20	?	Audi	green	84	?

Figure 163: Deleting Selected Rows

**Note:**

The WHERE clause is optional. However, omitting the WHERE clause deletes **all** rows of the table, which may not be intended.

In practice, the WHERE clause often specifies the primary key values of the rows to be updated. Note that “missing rows” do not lead to an exception, but they lead to no row being updated.

The WHERE clause supports the same features as for the SELECT statement, for example sub queries.

**You can use an uncorrelated sub query in the DELETE statement.**

- Delete all cars that are reported as stolen.

```
DELETE
  FROM Car
 WHERE PlateNumber IN (SELECT PlateNumber
                           FROM Stolen);
```

CARID	PLATENUMBER
F02	HD-VW 4711
F03	HD-JA 1972
F04	HD-AL 1002
F05	HD-MM 3206
F07	HD-ML 3206
F08	HD-IK 1002
F09	HD-UP 13
F10	HD-MT 507
F11	HD-MM 208
F12	HD-XY 4711
F13	HD-IK 1001
F14	HD-MM 1977
F15	HD-MB 3030
F16	?
F18	HD-MQ 2006
F19	HD-VW 2012
F20	?

Figure 164: The WHERE Clause Works as Usual

You can also use correlated sub-queries, such as functions in the WHERE clause.

**LESSON SUMMARY**

You should now be able to:

- Add rows to database tables using SQL
- Change existing rows of a database table

- Remove existing rows from a database table

# Learning Assessment

1. For string-based data types, NULL is equal to an empty string or a space character.

*Determine whether this statement is true or false.*

- True
- False

2. Which of the following is correct when combining the result sets of multiple queries using UNION?

*Choose the correct answers.*

- A The individual result sets must have the same number of columns.
- B The corresponding result columns must have compatible data types.
- C The individual result sets must have the same number of rows.
- D The column names of the resulting output table are based on the first SELECT statement.

3. In an Left Outer Join, what happened to the rows of the left table without a matching row in the right table?

*Choose the correct answer.*

- A Rows of the left table without a matching row in the right table are copied to the query result and the missing values from the right table are filled with NULL values.
- B Rows of the left table without a matching row in the right table are NOT copied to the query result.
- C Rows of the left table without a matching row in the right table are copied to the query result and the missing values from the right table are filled with a blank if the data type is character or with a zero if it is numeric.
- D This will result as an error.

# Learning Assessment - Answers

1. For string-based data types, NULL is equal to an empty string or a space character.

*Determine whether this statement is true or false.*

True

False

Correct! For string-based data types, NULL does not equal to an empty string or a space character. For more details, see HA150 Unit 2.

2. Which of the following is correct when combining the result sets of multiple queries using UNION?

*Choose the correct answers.*

A The individual result sets must have the same number of columns.

B The corresponding result columns must have compatible data types.

C The individual result sets must have the same number of rows.

D The column names of the resulting output table are based on the first SELECT statement.

Correct! The individual result sets must have the same number of columns. The corresponding result columns must have compatible data types. Also, the column names of the resulting output table are based on the first SELECT statement. For more details, see HA150 Unit 2.

3. In an Left Outer Join, what happened to the rows of the left table without a matching row in the right table?

*Choose the correct answer.*

- A Rows of the left table without a matching row in the right table are copied to the query result and the missing values from the right table are filled with NULL values.
- B Rows of the left table without a matching row in the right table are NOT copied to the query result.
- C Rows of the left table without a matching row in the right table are copied to the query result and the missing values from the right table are filled with a blank if the data type is character or with a zero if it is numeric.
- D This will result as an error.

Correct! Rows of the left table without a matching row in the right table are copied to the query result and the missing values from the right table are filled with NULL values. For more details, see HA150 Unit 2.



**Lesson 1**

User-Defined Functions

129

**Lesson 2**

Creating Database Procedures

137

**Lesson 3**

Trapping Errors in SQLScript

143

**Lesson 4**

Creating User-Defined Libraries

151

**UNIT OBJECTIVES**

- Create and use scalar and table user-defined functions
- Create and use database procedures in SAP HANA
- Describe the need to trap errors
- Define customized error conditions
- Control program flow to deal with errors
- Create User-Defined Libraries



# User-Defined Functions

## LESSON OVERVIEW

This lesson introduces SQLScript and explains how to create and use user-defined functions.



## LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Create and use scalar and table user-defined functions

## Scalar User-Defined Functions

The most simple database objects for which you can make use of SQLScript are **Scalar User-Defined Functions (Scalar UDFs)**.

Scalar UDFs allow you to define functions that take a number of input parameters and return scalar values. Only expressions are allowed in the body of the UDF, so no table operations, CE functions, or array operations.

Scalar UDFs have the following features:



- They can have any number of **scalar** input parameters (primitive SQL types).
- They can return multiple **scalar** values.
- They can contain expressions within their body. Table and array operations are not supported.
- They can be used in the field list or the WHERE clause of SELECT statements – like built-in functions.
- They are callable via direct assignment in other user-defined functions or stored procedures (`x := my_scalar_func()`).
- They must be free of side-effects and do not support any type of SQL statement in their body.

UDFs can be managed in several ways in SAP HANA.

## Options Used to Define UDFs in SAP HANA

The following are some options used to define UDFs in SAP HANA:



- Executing direct SQL **statements** in the SQL Console of SAP Web IDE to generate the run time object — this is **not recommended** as this approach provides no support for managing the lifecycle of functions, for example, for adjusting the function code of an existing function. It also means the function is not considered as part of a complete set of development files that should be built together as a complete unit.

- Defining the UDF in a development source file of the type **.hdbfunction** in the SAP Web IDE Development view, then building it to generate the runtime object — this is the **recommended** approach and how we will proceed in this class with our exercises. For the diagrams we still refer to the direct SQL method (using the 'Create FUNCTION' statement). But as we are simply trying to illustrate the code, this approach is acceptable.

The SQL statement to define UDFs is `CREATE FUNCTION`. The basic syntax to define a scalar UDF using the direct SQL method is as follows:

### Basic Syntax to Define a Scalar UDF



```
CREATE FUNCTION <function name>
(<list of input parameters with type>)
RETURNS <scalar result parameter name and type>
AS
BEGIN
    <function body>
END;
```

The figure, A Simple Scalar UDF, gives an example of using scalar user-defined functions like built-in functions.

UDFs can also be deleted using an SQL statement. The syntax is as simple as follows:

### Deleting UDFs



```
DROP FUNCTION <function name>;
```



#### Note:

The only way to change the body of an existing UDF using SQL statements is to delete the function and re-create it.

You can also use **imperative logic** in scalar UDFs, to the extent that this does not conflict with the statements above. Imperative language constructs allow the developer to control data and control flow, for example loops, scalar variables and `IF-THEN-ELSE` statements.

### Imperative Logic in SQLScript

- 
- Imperative logic allows you to control the flow of the logic:
    - Scalar variable manipulation
    - Branching logic, for example using `IF-THEN-ELSE`
    - Loops — `WHILE` and `FOR`
    - `DDL` statements and `INSERT`, `UPDATE`, and `DELETE` statements
  - Imperative logic is executed exactly as scripted and is procedural. It prevents parallel processing.



Note:

DDL and DML are not supported in scalar user-defined functions anyway. They can be used in table UDFs and database procedures.

You can see an example of this logic in the figure Imperative Logic in a Scalar UDF.

As we will be using the recommended approach using a source file to define our functions, let's take a look at this now.

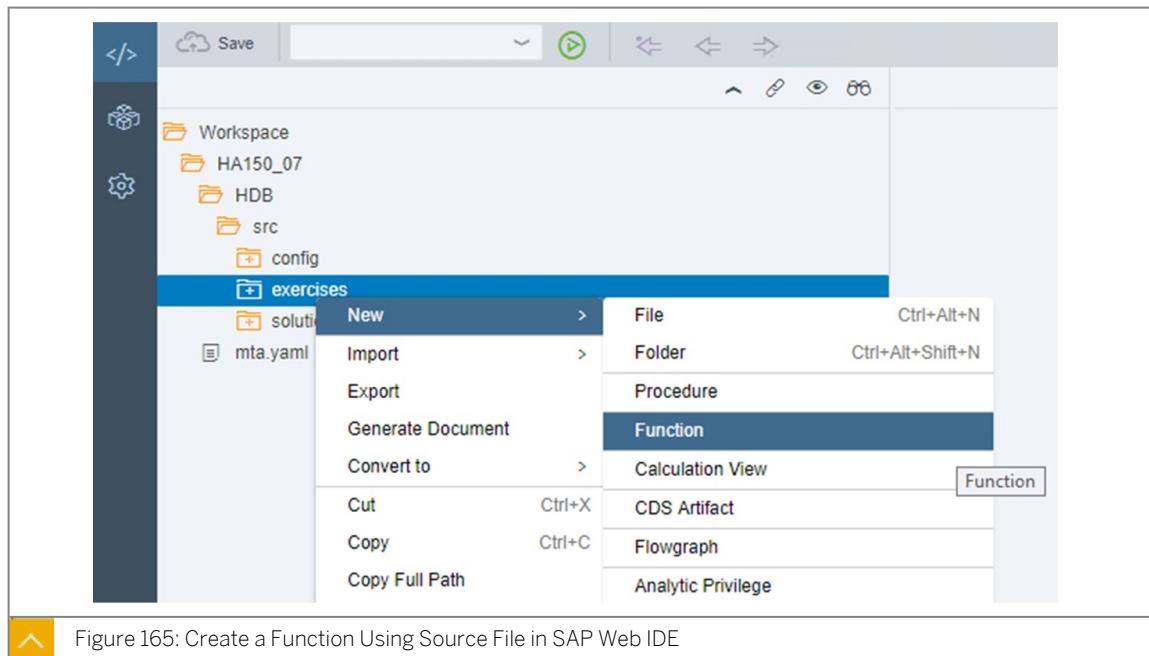


Figure 165: Create a Function Using Source File in SAP Web IDE

A function should always be created as a source file in the Development view of SAP Web IDE. Click a project folder that belongs to a HDB module and use the menu option **New → Function**. You don't need to specify the extension **.hdbfunction** as this is automatically appended to the name you provide for the file.

The file opens with basic shell code for you to complete.

When you have finished writing the code, you simply save then build the object. Assuming you have no errors, a runtime function is created in the container that belongs to the project. You can then open the SQL Console against the container and write some SQL to test your function.

If the function does not behave the way you expected, you can then enjoy the benefit of having a source file where you simply modify the code in the source file, re-save and re-build before re-testing. If you were using SQL statements directly to create the function, you would have to hope you saved the SQL code somewhere, otherwise you would have to begin again and re-enter it all over.

### Table User-Defined Functions

In addition to scalar UDFs, SAP HANA supports table UDFs.

Table UDFs have the following features:



- They can have any number of input parameters.

- They return **exactly one** table.
- They allow you to perform table operations within the body.
- They are used in the FROM clause of SELECT statements.
- They must be free of side-effects, that is, DDL statements or the DML statements INSERT, UPDATE, and DELETE cannot be used in function bodies.

Like scalar user-defined functions, the SQL statement to define table user-defined functions is also CREATE FUNCTION. However, table user-defined functions use the keyword TABLE for the return parameter, and assign the value of the sole return parameter using the keyword RETURN.

### Basic Syntax to Define a Table UDF

```
 CREATE FUNCTION <function name> (<list of input parameters with type>)
    RETURNS TABLE [table type](<list of table column definitions>) ]
AS
BEGIN
    <function body>
    RETURN <expression to set return table>
END;
```

 You can create table user-defined functions for use in **FROM clauses**.

```
CREATE FUNCTION Convert_OfficialsHours
(im_to VARCHAR(1) )
RETURNS TABLE ( PNr      NVARCHAR(3) ,
                Name     NVARCHAR(20) ,
                Overtime DEC(5,2))
AS
BEGIN
    RETURN SELECT PNr, Name,
                Convert_Hours(Overtime,:im_to) AS Overtime
        FROM Official;
END;

SELECT *
    FROM Convert_OfficialsHours('d');
```

 Figure 166: Example of a Table UDF

The figure, Example of Table UDF, shows an example of this syntax in use.

An example showing how this can be used in a table UDF is as follows.

### Dynamic Filtering

When defining table UDFs or stored procedures, a useful feature is **dynamic filtering** using the built-in table function APPLY\_FILTER. This function allows you to apply a dynamic WHERE clause to a database table or table variable and assign the result to a table variable, as shown in the figure, Dynamic Filtering Argument.

By providing parameters for the tests, you can apply a different WHERE to the same table on subsequent calls of the function, as shown in the figure Dynamic Filtering Argument WHERE Clause.

The figure, A Simple Table UDF, shows an example of this type of function.

## Execution Errors

There are a variety of methods to keep an error from execution of a procedure, function, or SQLScript block from causing an uncontrolled abort. The first method of dealing with these situations is the EXIT HANDLER. In general, when an EXIT HANDLER is declared, the following behavior is followed:

1. An error is encountered in procedure execution
2. Statements in the same execution block following the error are skipped
3. The actions specified in the EXIT HANDLER are executed

The DECLARE EXIT HANDLER command can be used to react to specific error codes. It can also be defined as generic and used to respond to any error.



You can raise exceptions in functions and with SQLScript.

```
CREATE PROCEDURE MYPROC AS
BEGIN
    DECLARE MYCOND CONDITION FOR SQL_ERROR_CODE 10001;
    DECLARE EXIT HANDLER FOR MYCOND
        SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE
        FROM DUMMY;

        SIGNAL MYCOND SET MESSAGE_TEXT = 'my error';
        INSERT INTO MYTAB VALUES (1); -- will not be reached
END;
```



Figure 167: Syntax Exit Handler

- The SQLEXCEPTION condition is generic and is invoked by any failure.
- SQL\_ERROR\_CODE values describe specific error states.
- Named conditions encapsulate specific SQL error codes and can define custom error messages.

SQL\_ERROR\_CODE values are described in the *SQL Reference* guide of the SAP HANA documentation set. Values range from the general (1: general warning; 2: general error) to the extremely specific (464: SQL internal parse tree depth exceeds its maximum). The EXIT HANDLER reacts to the following error code types:

- ERR\_SQL\_\* (series 200 above 255, series 300, most of series 400, some of series 500, and some of series 600)
- ERR\_SQLSCRIPT\_\* (series 1200, 1300, and 2800)

The following are some of the more significant errors:

- 256: SQL processing error
- 257: SQL syntax error
- 258: Insufficient privilege
- 259: Invalid table name

- 260: Invalid column name
- 263: Invalid alias name
- 264: Invalid datatype
- 266: Inconsistent datatype
- 267: Specified length too long for its datatype
- 276: Missing aggregation or grouping
- 284: Join field does not match
- 285: Invalid join condition
- 287: Cannot insert or update to NULL
- 407: Invalid date format



**In this example, the specified text string is returned when any error occurs.**

```
CREATE FUNCTION ...
AS
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
        SELECT <TEXT_STRING> FROM DUMMY;
    ...
END;
```

**In this example, the text string is returned when a datatype error occurs.**

```
CREATE FUNCTION ...
AS
BEGIN
    DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 264,
                           SQL_ERROR_CODE 266
        SELECT <TEXT_STRING> FROM DUMMY;
    ...
END;
```



Figure 168: Examples Exit Handler

If desired, a BEGIN ... END pair can be used to indicate a block of code to execute when the EXIT HANDLER is invoked. Similarly, an EXIT HANDLER inside a BEGIN...END block indicates a section of code to be skipped if there is not an exception encountered.

If a function is called without the im\_filter value supplied, the following events occur:

- The DECLARE lv\_filter command results in an SQL error.
- The exit handler catches the error and runs the two commands in the innermost BEGIN...END pair. The value of lv\_filter is set to an empty string and lt\_official populated from the APPLY\_FILTER expression. The line between the two END statements is skipped.

If the lv\_filter declaration is valid, the script skips to the line after the innermost END.

In every case, the script performs the RETURN statement, which is now guaranteed to have a valid value of lt\_official to draw on.

## Arrays in SQLScript

SQLScript supports the definition of an array as an indexed collection of elements of a single data type.

### With arrays in SQLScript, you can do the following



- Set and access elements by index using `<array>[<index>]`
- Add and remove elements from the ends of an array
- Concatenate two arrays using `CONCAT` or `||`
- Convert arrays into a table using `UNNEST`
- Convert a table column into an array using `ARRAY_AGG`
- Determine the length of the array using `CARDINALITY`

In the figure, Arrays in SQLScript, the syntax performs the following tasks:

- The `UNNEST` function converts the referenced arrays into a table, with specified column names.
- In this case, the unit and factor arrays become columns called `Unit` and `Factor` in a table called `lt_conversion`.
- The `COALESCE` function returns the first non-null value from its input arguments.
- In this case, the `:im_to` parameter is searched in the `lt_conversion` table and the appropriate value of `Factor` is returned.
- If the `:im_to` value is invalid, no rows are returned and `Factor` is null.
- If no rows are returned and `Factor` is null, `COALESCE` skips the value of `Factor` and returns a conversion ratio of 1.0.



## LESSON SUMMARY

You should now be able to:

- Create and use scalar and table user-defined functions



## Creating Database Procedures

### LESSON OVERVIEW

This lesson expands SQLScript and explains how to create and call stored procedures in SAP HANA.



### LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Create and use database procedures in SAP HANA

### Database Procedures

Database procedures are the most flexible mechanism to implement data-intensive calculations in SAP HANA, and for which SQLScript can be used.

SAP HANA offers you several ways to manage database procedures.

### Options to Define Database Procedures in SAP HANA

The following are some options to define database procedures in SAP HANA:



- Use a source file with the extension **.hdbprocedure** in the development view of the SAP Web IDE — this is the recommended approach.
- Typing the SQL statement 'CREATE PROCEDURE ...' directly in the SQL console — this is **not** the recommended approach.

### Basic Syntax to Define a Stored Procedure using the 'Create Procedure..' Statement



```
CREATE PROCEDURE <procedure_name>
[(<parameter_clause>) ]
[LANGUAGE <lang>]
[SQL SECURITY <mode>]
[DEFAULT SCHEMA <default_schema_name>]
[READS SQL DATA]
AS
BEGIN [SEQUENTIAL EXECUTION]
<procedure body>
END;
```

### Elements of a Procedure Header

The following are elements of a procedure header:



- LANGUAGE specifies which programming language the procedure is implemented in. SQLScript is the default.
- SQL SECURITY specifies which user's privileges apply when the procedure is executed. Default is DEFINER.

- `DEFAULT SCHEMA` specifies the database schema to be used for unqualified database object accesses within the procedure body.
- `READS SQL DATA` specifies that the procedure is free of side effects, that is, does not modify data in the database. You need to remove this line if your procedure uses DDL statements, such as `CREATE TABLE`.
- `SEQUENTIAL EXECUTION` forces sequential execution of the procedure logic. No parallelism takes place.

The following example illustrates the basic syntax:



**A procedure can have scalar and table output parameters. Their value is set using assignments.**

```
CREATE PROCEDURE Convert_OfficialsHours(
    IN im_filter VARCHAR(1000),
    IN im_to      VARCHAR(1),
    OUT ex_official TABLE (PNr      nvarchar(3),
                           Name     nvarchar(20),
                           Overtime dec(5,2)),
    OUT ex_message NVARCHAR(200) )
LANGUAGE SQLSCRIPT
SQL SECURITY INVOKER
READS SQL DATA
AS BEGIN
...
    ex_official = SELECT PNr, Name,
                        Convert_Hours(Overtime,:im_to) AS Overtime
        FROM :lt_official;
END;
```

Figure 169: Input and Output Parameters In Procedures

Database procedures can have several output parameters, and a mix of parameters of scalar and table types is possible.

They can also have input parameters and parameters which are both input and output. This is why the keywords `IN`, `OUT`, and `INOUT` are necessary in the definition of the signature to indicate the kind of parameter. `IN` is the default.

Several optional clauses can be used in the procedure header. The most important one is `READS SQL DATA`.

`READS SQL DATA` prevents the use of constructs in the procedure body that could lead to side effects, for example `INSERT` statements or dynamic SQL using `EXEC`. Remove this if you wish to use statements that will modify the database (insert records, drop table, and so on).

### Declarative Logic in SQLScript

Using declarative logic only, which implies side-effect free programming, has a very big advantage:

- The procedure developer defines **what** data to select using `SELECT` or `CE` functions.
- The database system determines the **how** and executes accordingly.
- Data selection using declarative statements can be parallelized massively.



**You can create stored procedures using SQLScript. Procedures can have multiple output parameters.**

```

CREATE PROCEDURE CarOwner_Count (OUT ex_company_cars INTEGER,
                                OUT ex_private_cars INTEGER )
  LANGUAGE SQLSCRIPT
  SQL SECURITY INVOKER
  READS SQL DATA
AS BEGIN
  lt_companies = SELECT OwnerID FROM Owner WHERE Birthday IS NULL;
  lt_persons   = SELECT OwnerID FROM Owner
                 WHERE Birthday IS NOT NULL;
  lt_cars = SELECT * FROM Car c WHERE PlateNumber
            NOT IN (SELECT PlateNumber FROM Stolen);
  SELECT COUNT(*) INTO ex_company_cars
    FROM :lt_cars INNER JOIN :lt_companies ON Owner = OwnerID;
  SELECT COUNT(*) INTO ex_private_cars
    FROM :lt_cars INNER JOIN :lt_persons ON Owner = OwnerID;
END;

CALL CarOwner_Count (?,?);

```

Figure 170: A Procedure Using Declarative Logic Only

The fact that the procedure "CarOwner\_Count" uses declarative logic only allows executing the first three queries and assignments in parallel because their results are mutually independent.

### Recap: Imperative Logic in SQLScript

The following is a recap of the imperative logic used in SQLScript:



- Imperative logic allows you to control the flow of the logic in the following ways:
  - Scalar variable manipulation
  - Branching logic, for example using IF-THEN-ELSE
  - Loops — WHILE and FOR
  - DDL statements and INSERT, UPDATE, and DELETE statements
- Imperative logic is executed exactly as scripted, which is procedural, and prevents parallel processing.

Database procedures can be deleted using an SQL statement if they have been created directly using SQL.

### Deleting Stored Procedures

The syntax is as simple as follows:



```
DROP PROCEDURE <procedure name>;
```

### Error Reporting

Procedures can have table and scalar output parameters, which can be used to return more readable error messages in case of exceptions. The figure, Using SQL Errors in Exceptions, illustrates an example, and shows how the original SQL error code and text can be accessed

using built-in variables `:SQL_ERROR_CODE` and `:SQL_ERROR_MESSAGE` (note the additional colon).



### You can access SQL error information using built-in parameters `:SQL_ERROR_CODE` and `:SQL_ERROR_MESSAGE`

```
CREATE PROCEDURE Convert_OfficialsHours( ... ) ...
AS
BEGIN
    DECLARE lv_filter NVARCHAR(1000) := :im_filter;
    BEGIN
        DECLARE EXIT HANDLER FOR SQLEXCEPTION
        BEGIN
            lv_filter := '';
            lt_official = APPLY_FILTER(Official,:lv_filter);

            ex_message := 'Result not filtered because of SQL ERROR '
                || ::SQL_ERROR_CODE || ' ' || ::SQL_ERROR_MESSAGE;
        END;
        lt_official = APPLY_FILTER(Official,:lv_filter);
    END;
    ...
END;
```

Figure 171: Using SQL Errors In Exceptions

In the example shown, an error trap similar to the `DECLARE EXIT HANDLER` for the table function is used. The main difference is that this time, the error message returned to the user incorporates the values of the built-in variables `:SQL_ERROR_CODE` and `:SQL_ERROR_MESSAGE`.

## Executing Stored Procedures

To call a procedure, use the `CALL` statement. There are different options depending on where you call the procedure and for passing parameters, as shown in the figure Calling Procedures.

Arguments for IN parameters of table type can either be physical tables, views or table variables. The actual value passed for tabular OUT parameters must be '?' when calling a procedure in the SQL Console.



## You can call a procedure using the `CALL` statement

### In the SQL Console:

Passing parameters by position

```
CALL ConvertOfficialsHours(NULL, 'd', ?, ?)
```

Passing parameters by name – allows ignoring their order

```
CALL ConvertOfficialsHours( im_filter=>NULL, im_to=>'d',
                            ex_official=>?, ex_message=>? )
```

### In the another procedure

```
CALL ConvertOfficialsHours( im_filter=>NULL, im_to=>'d',
                            ex_official=>lt_official,
                            ex_message=>lv_message )
```

### Using `WITH OVERVIEW`

Figure 172: Calling Procedures

Calling a procedure `WITH OVERVIEW` will return one result set that holds the information of which table contains the result of a particular table output parameter. Scalar outputs are represented as temporary tables with only one cell. When you pass existing tables to the output parameters, `WITH OVERVIEW` inserts the result set tuples of the procedure into the provided tables. When you pass '?' to the output parameters, temporary tables holding the result sets are generated. These tables are dropped automatically when the database session is closed.

We will create a procedure but we will use the recommended approach which is to develop a source file with the extension `.hdbprocedure` in the development view of SAP Web IDE.

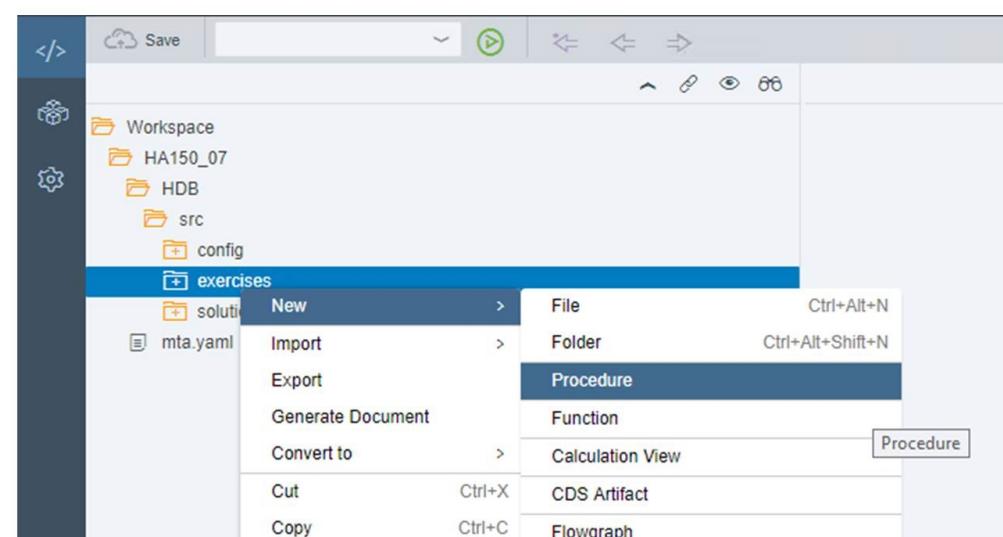


Figure 173: Create a Procedure Using a Source File

You do not need to specify the extension `.hdbprocedure` because this is added automatically to the name you provide.

When you first open the source file you will see the basic outline of the procedure code. You simply add your own code, save, and then build.

As with a function built using a source file, you do not drop the procedure directly using a *Drop* statement in SQL. If you wish to remove the procedure you simply delete the source file and in the next build of the folder in which it was stored, the run-time object is then deleted.



### LESSON SUMMARY

You should now be able to:

- Create and use database procedures in SAP HANA

## Trapping Errors in SQLScript



### LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Describe the need to trap errors
- Define customized error conditions
- Control program flow to deal with errors

### Trapping Errors

In addition to the sorts of exceptions previously described, there are circumstances in which values may meet the technical requirements of SQLScript, but can still lead to undesired or nonsensical results. In such cases, it is helpful to trap the problematic values (sometimes the result of bad inputs, sometimes resulting in undesired results) and lead the SQLScript code to a clean exit, preferably one that informs the end user about what problems occurred and how to avoid them in the future.

The first step to resolving errors is to determine situations in which they may occur.

### Causes of Errors

The following are the situations in which errors occur:



- Missing parameters
- Mismatched datatypes
- Invalid or undesired values
- Misleading results

In earlier examples, EXIT HANDLERS reacted to any exceptions in SQLScript code, but not every error that we care about comes from an exception. The following example describes various outcomes to a simple procedure:



**Consider a procedure that is supposed to return a total number of stolen cars, based on Brand and Color as input parameters:**

**Call the procedure: are both parameters present?**

No: **Error – missing parameter(s)**

Yes: Are both parameters the correct datatype?

Not: **Error – mismatched type(s)**

Yes: Do the Brans and Color values actually exist?

No: **Misleading ouptut: cars cannot exist, so zero stolen?**

Yes: Are there any cars that match the Brand and Color?

No: **Misleading output: cars do not exist, so zero stolen**

Yes: **Join the PlateNumber to the Stolen table, return total**

- **The two errors will result in an abort if left unaddressed.**
- **If fully executed, two misleading output states would both report a misleading total of zero stolen cars.**
- **Only the last case provides a meaningful total (including zero total).**



Figure 174: Errors to Trap

In this example, the procedure should have several possible exit points:

- The two errors should lead to messages to the user, indicating an unsuccessful execution due to bad inputs, and showing the required parameters/types.
- The first misleading output, if joined, will report zero stolen cars. However, this is misleading because the calculation is done with nonexistent values of Brand and/or Color. This branch of the logic tree should exit without joining and inform the user that the supplied Brand and/or Color is not valid. A list of unique values may be returned to the user to show valid values.
- The second misleading output, if joined, will also report zero stolen cars. However, this is because no cars exist in the Brand/Color combination. Even though every value supplied is valid, it is not meaningful to indicate a total of zero stolen cars. This branch of the logic tree should exit without joining and inform the user no such cars exist.
- Only if a valid and existing combination of Brand/Color is supplied should PlateNumber values be joined against the Stolen table, and a count(\*) performed. Note that zero totals in this case are meaningful.

### Customized Error Messages

You may declare a CONDITION to customize error messages and outcomes.



- Declaring a CONDITION variable allows you to name SQL error codes or define a user-defined condition.
- Using CONDITION variables for SQL error codes makes the procedure/function code more readable. For example, instead of using the SQL error code 304 (division by zero), you can describe it with a meaningful name.

**Syntax:**

```
DECLARE <condition name>
    CONDITION [ FOR SQL_ERROR_CODE <ERROR_CODE> ];
```

**Examples:**

```
DECLARE invalid_input;

DECLARE invalid_input;
    CONDITION FOR SQL_ERROR_CODE 10001;

DECLARE divide_by_zero;
    CONDITION FOR SQL_ERROR_CODE 304;
```

**Note that user-defined error codes must be within the range of 10000 to 19999**

Figure 175: Trapping Errors Condition

The CONDITION declaration may be assigned to a specific error code:

- System error codes act as a description of that system code.
- User-defined error codes are not tied to specific error codes.

When a CONDITION has an error code assigned to it, it may be returned to the user when the CONDITION is raised. This is useful, because it allows the returned SQL\_ERROR\_CODE to indicate not just that a block of code has finished executing, but the exit point at which it finished. CONDITIONS may be referenced by EXIT HANDLERS, as shown in the code fragments:

**Example 1**

```
DECLARE MYCOND CONDITION FOR SQL_ERROR_CODE 301;

DECLARE EXIT HANDLER FOR MYCOND
    SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE
    FROM DUMMY;
```

**Example 2**

```
DECLARE MYCOND CONDITION FOR SQL_ERROR_CODE 1001;

DECLARE EXIT HANDLER FOR MYCOND
    SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE
    FROM DUMMY;
```

**Example 3**

```
DECLARE MYCOND CONDITION;

DECLARE EXIT HANDLER FOR MYCOND
    SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE
    FROM DUMMY;
```

Figure 176: Exit Handler with Condition

These examples have different behaviors:

- Example 1 returns the error code (304) and associated text.
- Examples 2 and 3 do not produce the error code and error message when the procedure exits.
  - In example 2, no error 10001 will ever be encountered.
  - In example 3, the EXIT HANDLER does not know which error to react to.
  - In examples 2 and 3, no error message text is supplied to be returned.
  - In example 3, no error number is supplied to be returned.

However, custom error text can be associated with the third example, via the SIGNAL or RESIGNAL commands.

Thinking back to the example of the stolen cars, it's good for the user to know what the outcome of the procedure represents, but it may be even more important for the SQLScript/procedure that called the stolen cars stored procedure to know.

Depending on the meaning of the output, the calling code might disregard the result, prompt the user to try different values, or call further procedures and pass the result. This can be accomplished by intentionally declaring a CONDITION or custom error code to be in effect, also by means of the SIGNAL or RESIGNAL commands.

## Raising Exceptions

It is possible to explicitly declare an exception to be in effect with the SIGNAL command.



### Syntax:

```
SIGNAL (<USER_DEFINED_CONDITION> | SQL_ERROR_CODE <INT> )
[SET MESSAGE_TEXT = '<MESSAGE_STRING>']
```

### Examples:

```
SIGNAL SQL_ERROR_CODE 10001;

SIGNAL invalid_input;

SIGNAL invalid_input;
SET MESSAGE_TEXT = 'Invalid input arguments';
```

Figure 177: Signal Syntax

The first example assigns a custom error code to the value of: SQL\_ERROR\_CODE (which must fall in the previously-discussed range of 10000 to 19999). In the second example, the execution is declared to be in the state of invalid\_input, which would have been set by a DECLARE CONDITION statement.

In both examples, there is no value of: SQL\_ERROR\_MESSAGE. In the third example, a value is assigned to SQL\_ERROR\_MESSAGE and may be returned.



### You can raise exceptions in functions and with SQLScript.

```

CREATE FUNCTION Convert_Hours
(im_hours INTEGER, im_to VARCHAR(1)
RETURNS ex_result DEC(5,2)
AS
BEGIN
    DECLARE UNKNOWN_UNIT
        CONDITION FOR SQL_ERROR_CODE 10001;

    ...IF      :im_to = 'm' THEN ex_result := :im_hours * 60;
    ...ELSEIF :im_to = 'd' THEN ex_result := :im_hours / 24;
    ...ELSEIF :im_to = 'h' THEN ex_result := :im_hours;
    ...ELSE SIGNAL UNKNOWN_UNIT
        SET message_text = 'Target unit' || :im_to ||
                           'not supported';
    END IF;

END

```



Figure 178: Signal Example

In this example, the UNKNOWN\_UNIT exception is defined as having error code 10001. If a non-valid value of: im\_to is supplied, the UNKNOWN\_UNIT exception is declared to be in effect (setting the current value of: SQL\_ERROR\_CODE to 10001), and the message\_text string assigned to: SQL\_ERROR\_MESSAGE. The code and string are incorporated into the system-generated error that is returned to the user.

When combined with an EXIT HANDLER, the SIGNAL can control which portions of SQLScript are executed.



### You can raise exceptions in procedures and with SQLScript.

```

CREATE PROCEDURE MYPROC AS
BEGIN
    DECLARE MYCOND CONDITION FOR SQL_ERROR_CODE 10001;
    DECLARE EXIT HANDLER FOR MYCOND
        SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE
        FROM DUMMY;

    .....

    SIGNAL MYCOND SET MESSAGE_TEXT = 'my error';
    INSERT INTO MYTAB VALUES (1); -- will not be reached

END;

```



Figure 179: Signal Exit Handler

In this example, the MYCOND exception is defined as having error code 10001, and an EXIT HANDLER is declared: in the event of MYCOND being true, the values of: SQL\_ERROR\_CODE (10001) and: SQL\_ERROR\_MESSAGE (undefined) will be returned to the user via a SELECT.

The SIGNAL command then declares MYCOND to be in effect and supplies a message text so that: SQL\_ERROR\_MESSAGE is no longer undefined. When called, the stored procedure stops

before reaching the INSERT command. Querying the MYTAB table will reveal it to not include the value of 1.

### Resignal

The RESIGNAL command is similar to SIGNAL, but is used exclusively in the EXIT HANDLER. As such, it does not declare a particular exception to be in effect. It can be used to pass along a condition name/error code number, or message text.



#### RESIGNAL is attached to the EXIT HANDLER:

```
CREATE PROCEDURE MYPROC2
(IN in_var INTEGER, OUT outtab TABLE (1 INTEGER) )
AS

BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    RESIGNAL SET MESSAGE_TEXT =
        'for the input parameter in_var = '
        || :in_var || ' exception was raised ';
    outtab = SELECT 1/:in_var as I FROM DUMMY;
END;
```

Alternatively:

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION
RESIGNAL SET MESSAGE_TEXT =
    'for the input parameter in_ar = '
    || :in_var || ' exception was raised '
    || ::SQL_ERROR_MESSAGE;
```



Figure 180: Resignal Example

In the first example, an integer is supplied as input, and its reciprocal integer returned as output. If a zero value is supplied, a divide-by-zero error will be raised and caught by the EXIT HANDLER as a SQLEXCEPTION. In this case, the RESIGNAL command is used to place customized text in the error message – the string for the input parameter in\_var = is concatenated with the value of: in\_var and the string exception was raised.



Note:

Unlike SIGNAL, RESIGNAL does not supply a value for: SQL\_ERROR\_MESSAGE.

In the second (partial) example, the RESIGNAL command uses the same custom text as in the first example, then further concatenates the original value of: SQL\_ERROR\_MESSAGE, which is preserved.

### Continue Handler

The EXIT handler in SQLScript already offers a way to process exception conditions in a procedure or a function during execution. The CONTINUE handler not only allows you to handle the error but also to continue with the execution after an exception has been thrown. More specifically, SQLScript execution continues with the statement following the exception-throwing statement right after catching and handling the exception.



With the new **Continue Handler** it is now possible to handle the error thrown during execution of a procedure or function and also to continue the execution afterwards from the next statement after the error-throwing statement as well.

```

1 PROCEDURE "test_continue_handler"( in im_var int,
2                                     out ex_var int,
3                                     out ex_message
4                                     table (error_code nvarchar(20),
5                                            | message nvarchar(250) )
6 LANGUAGE SQLSCRIPT
7 SQL SECURITY INVOKER
8 READS SQL DATA AS
9 BEGIN
10
11  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION -- Catch the exception
12    BEGIN
13      ex_message.error_code[1] = ::SQL_ERROR_CODE;
14      ex_message.message[1] = ::SQL_ERROR_MESSAGE;
15    END;
16
17    ex_var = :im_var / 0; -- An exception will be thrown
18    ex_var = :im_var; -- Execution continues
19
20 END

```

```

1 CALL "test_continue_handler"
2 IM_VAR => 99,
3 EX_VAR => ?
4 EX_MESSAGE => ?
5 );

```

Result1		Result2	Messages
Rows (1)			SQL
	ERROR_CODE	MESSAGE	↓
1	304	division by zero undefined: cannot be divided by zero	

Result1		Result2	Messages
Rows (1)			SQL
		EX_VAR	↓
1	99		

Figure 181: Continue Handler

Please note the following in regards to the CONTINUE HANDLER statement:

- CONTINUE HANDLER statements are not supported in any procedure or function placed in a parallel execution block statement (that is, 'BEGIN PARALLEL EXECUTION ... END;') as it is not possible to determine the next statement (as all procedures and functions are executing in parallel).
- If there is an error in a conditional statement for an IF, a WHILE, or a FOR block, the whole block will be skipped after handling the error because the condition is no longer valid.
- The value of the variable remains as it was before the execution of the statement that returns an exception.

## Catching Errors



```

21
22  DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 131
23  BEGIN
24    ex_message := 'Error Code: ' || ::SQL_ERROR_CODE || ' ' || ::SQL_ERROR_MESSAGE;
25  END;

```

- As of SPS04, it is now possible to catch the following transaction errors
  - ERR\_TX\_ROLLBACK\_LOCK\_TIMEOUT(131)
  - ERR\_TX\_ROLLBACK\_DEADLOCK(133)
  - ERR\_TX\_SERIALIZATION(138)

In addition to **ERR\_SQL\_\*** and **ERR\_SQLSCRIPT\_\*** errors

Figure 182: Catching Specific Errors

There are (as of SPS04) 478 distinct errors capable of being thrown by the SQL Engine Runtime. All of them have a unique number assigned by SAP along with a unique name.

The number assigned for each distinct error ranges from 131 to 2891 (not all numbers are used). The name uses one of four prefixes and is either:

- ‘ERR\_SQL\_’ (321 different errors)
- ‘ERR\_TX\_’ (4 different errors)
- ‘ERR\_SQLSCRIPT\_’ (151 different errors)
- ‘ERR\_API\_’ ( 2 different errors).

All errors with prefix ‘ERR\_SQL\_’, ‘ERR\_SQLSCRIPT\_’, and (as of SPS04 only) ‘ERR\_TX\_’ can be caught by the application developer.

Using the code in the figure, Catching Specific Errors, we have (on line 22) the unique error number 131 which has the unique name ERR\_TX\_ROLLBACK\_LOCK\_TIMEOUT.



## LESSON SUMMARY

You should now be able to:

- Describe the need to trap errors
- Define customized error conditions
- Control program flow to deal with errors

# Unit 3

## Lesson 4

# Creating User-Defined Libraries



## LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Create User-Defined Libraries

## User-Defined Libraries

You have built many functions, procedures, and variables during your project implementation. If you think some of these objects can be reused in some other areas of the project or outside the project, they can be organized into a custom **library**.



A library is a set of related variables, procedures and functions

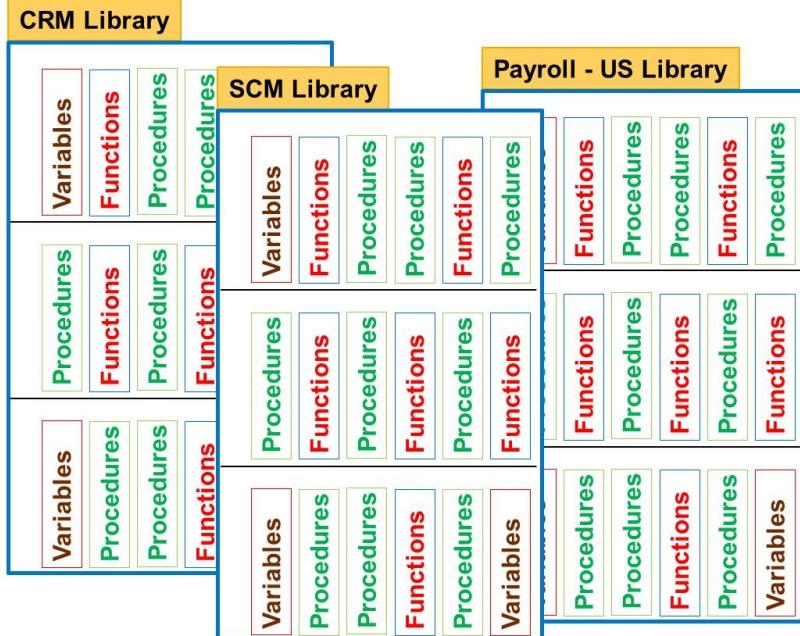


Figure 183: SQLScript Libraries

A library is a group of variables, procedures, and functions that are related. SAP HANA has some built-in libraries, but you can also create custom libraries.

The members of the library can be re-used in other procedures or functions.



<p><b>Types of library</b></p> <ul style="list-style-type: none"> <li>• User defined libraries</li> <li>• Built-in libraries</li> </ul> <p><b>For functions and procedures</b></p> <p><b>Private</b> functions and procedures</p> <ul style="list-style-type: none"> <li>• For internal use within the library</li> <li>• Cannot be called from outside the library</li> </ul> <p><b>Public</b> functions and procedures</p> <ul style="list-style-type: none"> <li>• Can be used / called from outside the library like a non-library function or procedure</li> <li>• Need the EXECUTE privilege on the library</li> </ul>	<p><b>Library members</b></p> <ul style="list-style-type: none"> <li>• Variables</li> <li>• Functions</li> <li>• Procedures</li> </ul> <p><b>Views to display all libraries and members (SYS schema)</b></p> <ul style="list-style-type: none"> <li>• LIBRARIES</li> <li>• LIBRARY_MEMBERS</li> </ul>
--	---

Figure 184: SQLScript Libraries – Key Points

The library members functions and procedures can be defined as **PRIVATE** or **PUBLIC**, by defining their access mode. There may be some specialist library members that only makes sense to be used with some projects, and these can be defined as private. These will not be available for or other procedures or functions. Generally useful functions and procedures can be defined as public for any users who have the EXECUTE permission on the library.

Although libraries and their members can be defined directly in the SQL Console using the `CREATE LIBRARY` statement, this is not recommended for development projects and source artifacts should be defined instead using the Development view of SAP Web IDE.

The source file type is **.HDBLIBRARY**. There is no dedicated menu option for this artifact, as there is for procedures and functions. You should use *New > File* and be sure to specify the file extension after the file name.

You use the supplied system views to list all possible libraries and their members they contain once they are built:

- SYS.LIBRARIES
- SYS LIBRARY\_MEMBERS



**Note:**

Libraries are designed to be used only in SQLScript procedures or functions and are not available outside these objects.



### Create a library and some members

```
create library mylib as begin

    public variable maxval constant int = 100;

    public function bound_with_maxval(i int) returns x int as begin
        x = case when :i > :maxval then :maxval else :i end;
    end;

    public procedure get_data2(in size int, out result table(col1 int)) as begin
        result = select top :size col1 from data_table;
    end;
end;
```

### Use system view LIBRARY\_MEMBERS to list available members

LIBRARY_NAME	LIBRARY_OID	MEMBER_NAME	MEMBER_TYPE	ACCESS_MODE	DEFINITION
MYLIB	1023255	BOUND_WITH_MAXVAL	FUNCTION	PUBLIC	function bound_with_maxval(i int) returns x int a
MYLIB	1023255	GET_DATA2	PROCEDURE	PUBLIC	procedure get_data2(in size int, out result table(c
MYLIB	1023255	MAXVAL	VARIABLE	PUBLIC	variable maxval constant int = 100

Figure 185: Creating a Library and its Members

Each member in a library does not have its own individual metadata object. Only the entire library is a metadata object and all members in the library are considered part of that single metadata object. There is very often a dependency on the individual members of a library.

If one member becomes invalid, it might have an effect on the other members in the library object.

Because library members do not have their own metadata object, it is possible to have library members that share the same name. SAP recommends that you always use fully qualified names when defining the library members.



### Procedure using library members

```
create procedure myproc (in inval int) as begin

    using mylib as mylib;

    declare var1 int = mylib:bound_with_maxval(:inval);

    if :var1 > mylib:maxval then
        select 'unexpected' from dummy;
    else
        declare tv table (col1 int);
        call mylib:get_data2(:var1, tv);
        select count(*) from :tv;
    end if;
end;
```

Figure 186: Using Library Members

To delete a library, you simply use the `DROP LIBRARY` statement if you have created this directly with the SQL Console using the `CREATE LIBRARY` statement. If you have created this library using the source file `.HDBLIBRARY`, then simply delete the source file and rebuild the parent folder to remove the runtime object.

SAP HANA comes with a few built-in libraries for handling a few of the special functions that can be handled with efficient performance.



### Built-In Libraries

#### **SQLSCRIPT\_SYNC**

- Offers functions for sleeping and waking up with performance efficiency

#### **SQLSCRIPT\_STRING**

- Offers simple ways for manipulating strings

#### **SQLSCRIPT\_PRINT**

- Makes it possible to print strings or even whole tables

RB SCHEMA_NAME ▾	RB LIBRARY_NAME	RB MEMBER_NAME	RB MEMBER_TYPE
SYS	SQLSCRIPT_SYNC	SLEEP_SECONDS	PROCEDURE
SYS	SQLSCRIPT_SYNC	WAKEUP_CONNECTION	PROCEDURE
SYS	SQLSCRIPT_STRING	FORMAT	FUNCTION
SYS	SQLSCRIPT_STRING	FORMAT_TO_ARRAY	FUNCTION
SYS	SQLSCRIPT_STRING	FORMAT_TO_TABLE	FUNCTION
SYS	SQLSCRIPT_STRING	SPLIT	FUNCTION
SYS	SQLSCRIPT_STRING	SPLIT_REGEXPR	FUNCTION
SYS	SQLSCRIPT_STRING	SPLIT_REGEXPR_TO_ARRAY	FUNCTION
SYS	SQLSCRIPT_STRING	SPLIT_REGEXPR_TO_TABLE	FUNCTION
SYS	SQLSCRIPT_STRING	SPLIT_TO_ARRAY	FUNCTION
SYS	SQLSCRIPT_STRING	SPLIT_TO_TABLE	FUNCTION
SYS	SQLSCRIPT_STRING	TABLE_SUMMARY	FUNCTION
SYS	SQLSCRIPT_PRINT	PRINT_LINE	PROCEDURE
SYS	SQLSCRIPT_PRINT	PRINT_TABLE	PROCEDURE

Figure 187: Built-In Libraries

For example, you might want to let certain processes wait for a while. By different user's design, implementing such waiting manually may lead to "busy waiting" and to the CPU performing unnecessary work during this waiting time.

To avoid this, SQLScript offers a built-in library `SYS.SQLSCRIPT_SYNC` containing the procedures `SLEEP_SECONDS` and `WAKEUP_CONNECTION`.

Similar to this library, there are libraries for manipulating strings (`SYS.SQLSCRIPT_STRING`) and for some special `PRINT` functions including table printing (`SYS.SQLSCRIPT_PRINT`).

Beware of the current limitation when using libraries:

- The usage of library variables is currently limited. For example, it is not possible to use library variables in the `INTO` clause of a `SELECT INTO` statement and in the `INTO` clause of dynamic SQL. This limitation can be easily circumvented by using a normal scalar variable as intermediate value.
- It is not possible to call library procedures with hints.
- Since session variables are used for library variables, it is possible (provided you have the necessary privileges) to read and modify arbitrary library variables of (other) sessions.
- Variables cannot be declared by using `LIKE` for specifying the type.

- Non-constant variables cannot have a default value yet.
- The table type library variable is not supported.
- A library member function cannot be used in queries.



### LESSON SUMMARY

You should now be able to:

- Create User-Defined Libraries



## Learning Assessment

1. Table User-Defined Functions can accept any number of input parameters and return any number of table results.

*Determine whether this statement is true or false.*

- True
- False

2. Stored procedures must utilize only scalar parameters, or only table parameters.

*Determine whether this statement is true or false.*

- True
- False

3. Which two statements are correct regarding table functions?

*Choose the correct answers.*

- A They can have any number of input parameters.
- B They can return multiple tables.
- C They must be free of side effects.
- D They are used in the WHERE clause of a SELECT statement.

4. What is the recommended approach for creating procedures?

*Choose the correct answer.*

- A Define a source file *.hdbprocedure* in the development view of SAP Web IDE.
- B Use the *CREATE PROCEDURE* statement in the SQL Console.

5. What is a user-defined library?

*Choose the correct answer.*

- A** A group of variables, procedures and functions, which are related.
- B** A repository for storing custom SQL code that is used frequently.
- C** A collection of frequently used input parameters that form part of a test script

## Learning Assessment - Answers

1. Table User-Defined Functions can accept any number of input parameters and return any number of table results.

*Determine whether this statement is true or false.*

- True  
 False

Correct! Answer is False. While Table User-Defined Functions can accept any number of input parameters, they must return exactly one table.

2. Stored procedures must utilize only scalar parameters, or only table parameters.

*Determine whether this statement is true or false.*

- True  
 False

Correct! Answer is False. Stored procedure parameters may be a mix of scalar and table.

3. Which two statements are correct regarding table functions?

*Choose the correct answers.*

- A They can have any number of input parameters.  
 B They can return multiple tables.  
 C They must be free of side effects.  
 D They are used in the WHERE clause of a SELECT statement.

Correct! A table function is used in a FROM clause, not a WHERE clause, and can return only one table. A table function can have any number of input parameters and must be free of side effects. For more details, see HA150 unit 3.

4. What is the recommended approach for creating procedures?

*Choose the correct answer.*

A Define a source file *.hdbprocedure* in the development view of SAP Web IDE.

B Use the *CREATE PROCEDURE* statement in the SQL Console.

Correct! You should always build a procedure by defining a source file *.hdbprocedure* in the development view of SAP Web IDE. In fact, all database objects should be created using source files in this way. Never create database objects using *CREATE* statements in the SQL Console. For more details, see HA150 unit 3.

5. What is a user-defined library?

*Choose the correct answer.*

A A group of variables, procedures and functions, which are related.

B A repository for storing custom SQL code that is used frequently.

C A collection of frequently used input parameters that form part of a test script

Correct! A user-defined library is a group of variables, procedures and functions, which are related and can be reused by other developers in their code. A repository for storing custom SQL code that is used frequently is called the SQL Statement Library. There is no such library that can store frequently used input parameters that form part of a test script. For more details see HA150 unit 3.

### Lesson 1

Using Declarative Logic

163

#### UNIT OBJECTIVES

- Use declarative logic



# Using Declarative Logic



## LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Use declarative logic

## Declarative Logic

### What is declarative logic in SQLScript?

The aim of declarative logic in SQLScript is to enable optimization of the execution of data-intensive computations by carefully writing SQL code that declares only **intent** of the request and not the **how** to process.



### Declarative Logic

**Allows the developer to describe the data flow declare the data selection via SELECT statements**

- Used for efficient execution of data-intensive computations
- This logic is internally represented as a data flow graph with nodes that might execute in parallel
- Each statement is bound to a variable to be passed as input to the next statement
- With no changes to the database, the declarative logic is side-effect free
- Can only use a limited set of SQL Script language



Figure 188: SQLScript Declarative Logic Overview

You implement declarative logic by coding in SQLScript following specific rules in order to preserve the optimization. Once you break the declarative coding rules, you begin to limit the optimization possibilities.

Declarative logic in SQLScript maximizes the parallelization possibilities, and as SAP HANA is built for high levels of parallel processing, declarative programming is highly recommended.

The main requirement of declarative coding is that the operations in a data-flow graph must be **free of side effects**, meaning the statements must not change any global state, in the database, or at the application level. Or put simply, the SQLScript must execute read-only statements and any writing should be restricted to session variables and parameters which

only exist during the execution and do not have a permanent effect on the database, such as changing a record in a table or creating a persistent SQL object.

Parallelization and free of side effects can only be achieved by allowing changes to the data set that is passed as inputs to the operator, and by allowing only a limited subset of language features to express the logic of the operator.

When writing SQLScript code, careful planning and design is needed when specifying the statements in order to write the flow logic. But if you do this well, then the SAP HANA database has the freedom to figure out the data flow step dependencies and then optimize the data flow, which may result in better performance.

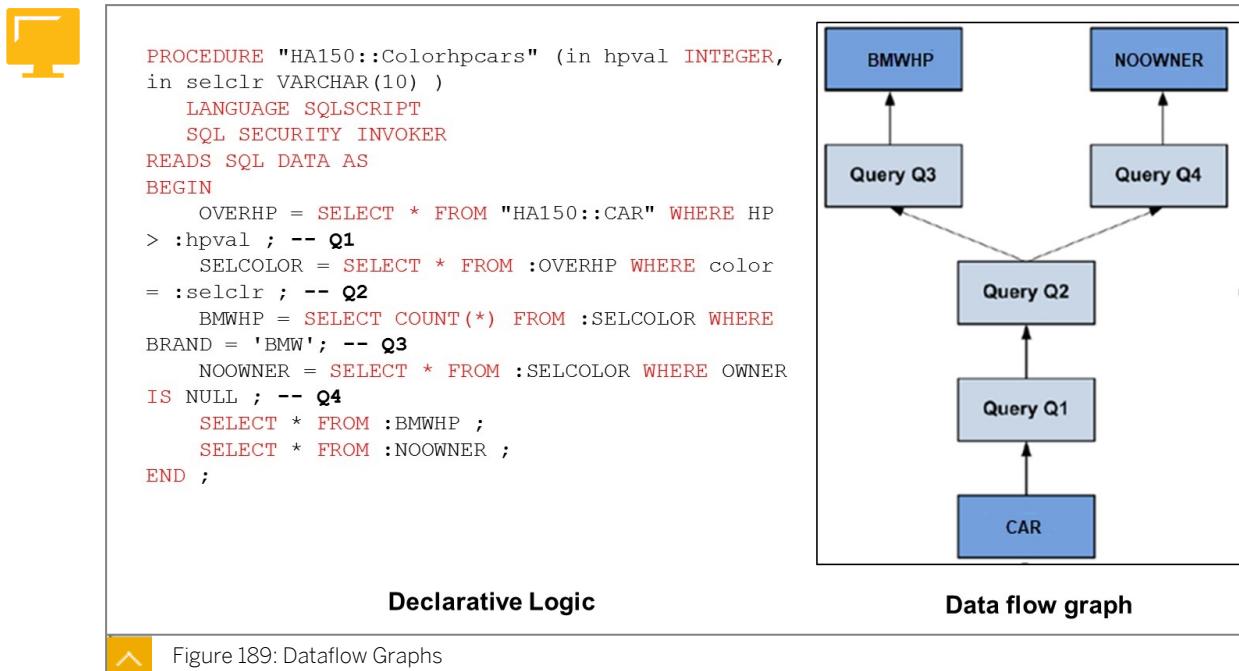


Figure 189: Dataflow Graphs

A procedure in SQLScript that only uses declarative constructs can be completely translated into an acyclic (does not loop) dataflow graph, where each node represents a data transformation.

This example in the figure defines a read-only procedure that has two scalar input parameters (IN hpval INTEGER, IN selclr VARCHAR(3)).

Query Q1, identifies the cars that have a HP over a certain value of any color (using the input parameter 'hpval').

Query Q2, identifies cars from Q1 that meet the specified color (using the input parameter 'selclr').

Finally, this information is aggregated in two different ways:

- Query Q3 is a count of the BMW cars from Q2
- Query Q4 returns cars that have no owner but any make. This selection is also from Q2
- These resulting tables constitute the output tables of the function. There are **no side-effects**

As you can see after Query Q2, there is parallelization happening for both queries Q3 and Q4.

All data flow operations have to be side-effect free, that is they must not change any global state either in the database or in the application logic, which will be optimized for parallel execution by the optimizer.



## Table Parameters

You can define table parameters in one of the two ways.

**Define directly in the signature of the procedure or function:**

- (IN inTab TABLE1(I INT), OUT outTab TABLE2 (I INT, J DOUBLE))
- Advantage: Define directly and not have to manage the object
- Disadvantage: It cannot be reused

**Use a table type that was already defined:**

- (IN inTab tableType1, OUT outTab tableType2)
- Advantage: It can be reused in other procedures and functions
- Disadvantage: Need to take care of the object lifecycle

Figure 190: Table Parameters

Table parameters are used to define inputs and outputs for procedures and functions. Table parameters that are defined in the procedure or function signature are either declared as input or output. They must be typed explicitly. This can be done either by using a table type previously defined with the CREATE TYPE command, or by writing it directly in the signature without any previously defined table type.

With table parameters, conceptually, this enables a temporary table. A reserved and locally visible schema is introduced with the table parameters. All objects within this special schema are only visible for the current transaction.



## Use of same table variable name in two separates coding blocks

```

PROCEDURE "HA150::LCL_DEF" ( )
  LANGUAGE SQLSCRIPT
  SQL SECURITY INVOKER
  READS SQL DATA AS|
BEGIN
DECLARE A CAR ;
    A = select * from CAR ; -- \$local.A_Scope1
    BEGIN
        DECLARE A OFFICIAL ;
        A = select * from OFFICIAL ; -- \$local.A_Scope2
    END;
    select * from :A;
END;

```

Figure 191: Table Variables

A table variable is an intermediate variable and appears in the body of a procedure or a table function and can be either derived from a query result, or declared explicitly.

If the table variable has derived its type from the SQL query, the SQLScript compiler determines its type from the first assignments of the variable thus providing a lot of flexibility. One disadvantage of this approach is that it also leads to many type conversions in the background because sometimes the derived table type does not match the typed table parameters in the signature. This can lead to additional unnecessary conversions. Another disadvantage is the unnecessary internal statement compilation to derive the types. To avoid this unnecessary effort, you can declare the type of a table variable explicitly.



## Table Variable Type

**DECLARE keyword is used to define a table variable type in the SQLScript.**

Again, you can define table variables by two methods.

**Define the table variable explicitly**

- `DECLARE local_table TABLE (n int);`

**Reference a type that was already defined**

- `DECLARE local_table MY_TABLE_TYPE;`

**Default value assignment**

- `DECLARE temp MY_TABLE_TYPE = UNNEST (:arr) as (i);`

**Default value assignment and a READ-ONLY table variable (CONSTANT)**

- `DECLARE temp CONSTANT MY_TABLE_TYPE DEFAULT SELECT * FROM TABLE;`



Figure 192: Table Variables

A declared table variable is always initialized with empty content. The name of the table variable must be unique among all other scalar variables and table variables in the same code block. However, you can use names that are identical to the name of another variable in a different code block. Additionally, you can reference those identifiers only in their local scope.



## Binding and Referencing – Table Variable

**Local table variables (for e.g. local\_table) can be bound and referenced after they are assigned.**

**Binding:**

- ```
lt_expensive_books =
    SELECT title, price, crcy FROM :it_books
    WHERE price > :minPrice AND crcy = :currency;
```

**Results of the SQL statement on the right side, is bound to the table variable**

- `lt_expensive_books`

**Referencing:**

- The `<:it_books>` variable in the FROM clause of the statement, refers to an IN parameter of a table type or a previously bound variable.



Figure 193: Binding Table Variables

Table variables are bound using an equals (=) operator. The results of a *SELECT* are passed to the intermediate table variable or to the output table parameter.

Table variables are referred to using the colon (:) symbol that prefixes the name of the table variables. Table variables can be referred to multiple times.



### Note:

If a table's variable is not consumed by a subsequent statement, then it is ignored by the optimizer.



## MAP\_MERGE

**MAP\_MERGE operator for evaluating each row of a tabular input to a mapper function in parallel and union all intermediate results**

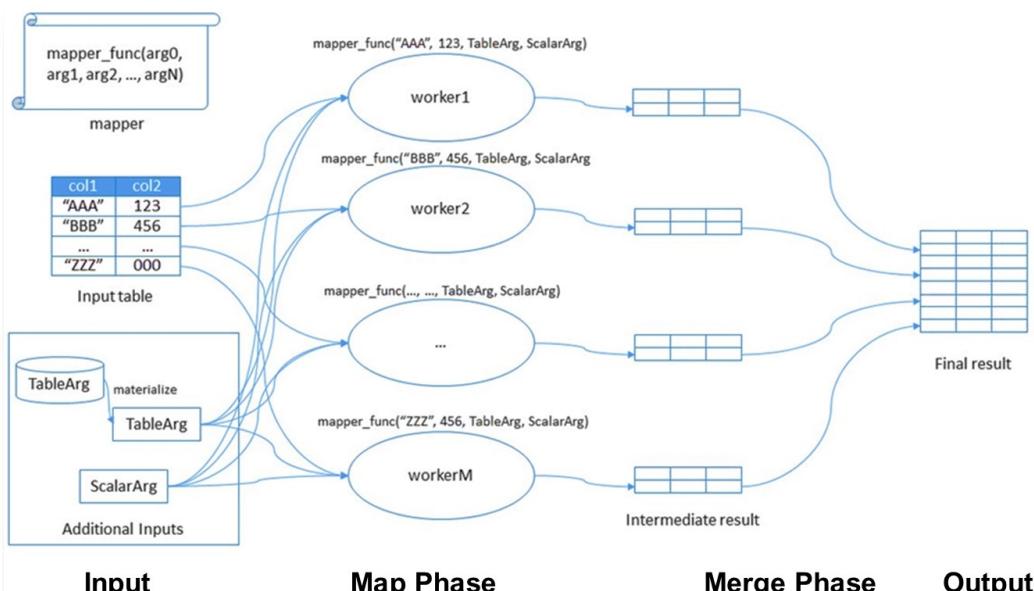


Figure 194: Map Merge

The MAP\_MERGE operator is used to apply each row of the input table to the mapper function and unite all intermediate result tables. The purpose of the operator is to replace sequential FOR-loops and union patterns, like in the example below, with a parallel operator to significantly improve performance.

The mapper procedure is a read-only procedure with only one output that is a tabular output.



### MAP\_MERGE operator example

Replace this code with ...

```
DO (OUT ret_tab TABLE(col_a nvarchar(200))=>?)
BEGIN
    DECLARE i int;
    DECLARE varb nvarchar(200);
    t = SELECT * FROM tab;
    FOR i IN 1 .. record_count(:t) DO
        varb = :t.col_a[:i];
        CALL mapper(:varb, out_tab);
        ret_tab = SELECT * FROM :out_tab
        UNION SELECT * FROM :ret_tab;
    END FOR;
END;
```

Parallel MAP\_Merge Operator Version

```
DO (OUT ret_tab TABLE(col_a nvarchar(200))=>?)
BEGIN
    t = SELECT * FROM tab;
    ret_tab = MAP_MERGE(:t, mapper(:t.col_a));
END;
```

Figure 195: MAP MERGE operator example

As an example, let us rewrite this example to leverage the parallel execution of the MAP\_MERGE operator. We need to transform the procedure into a table function, because MAP\_MERGE only supports table functions as <mapper\_identifier>.



### Map Reduce Example

Input

| ID | STRING |
|----|--------|
| 1  | A,B    |
| 2  | B,E,F  |
| 3  | F,C    |
| 4  | A,A    |
| 5  | A,B,B  |
| 6  | E,F    |

→→→ Required output

| AB | VAL | 12 STMT_FREQ | 12 TOTAL_FREQ |
|----|-----|--------------|---------------|
|    | A   | 3            | 4             |
|    | B   | 3            | 4             |
|    | C   | 1            | 1             |
|    | E   | 2            | 2             |
|    | F   | 3            | 3             |

Figure 196: Map Reduce

MAP\_REDUCE is a programming model introduced by Google that allows easy development of scalable parallel applications for processing big data on large clusters of commodity machines.

The MAP\_REDUCER operator is a specialization of the MAP\_MERGE operator.



## MAP\_REDUCE

MapReduce is a programming model which allows easy development of scalable parallel applications to process big data on large clusters of commodity machines.

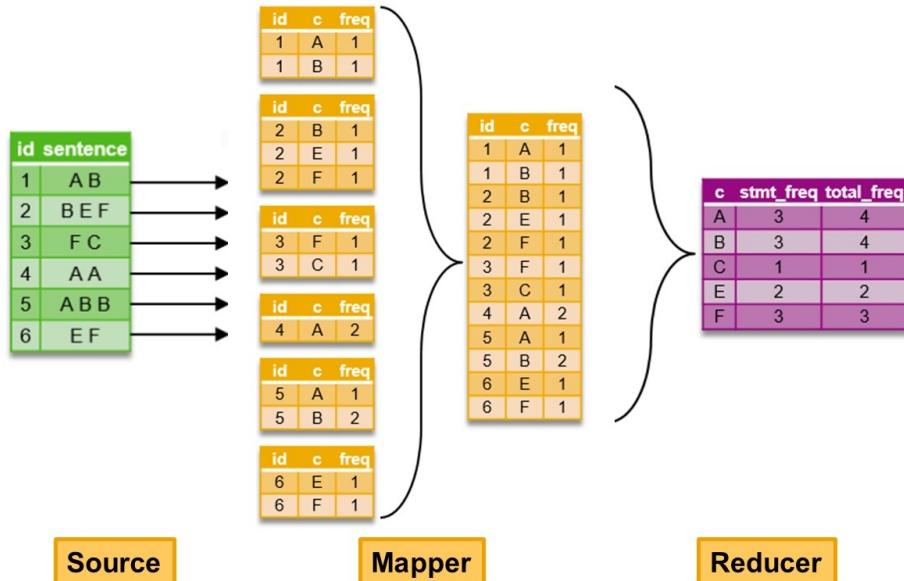


Figure 197: Map Reduce



## MAP\_REDUCE Example

### 1) Build the Mapper

#### Mapper Function

```
create function mapper(in id int, in sentence varchar(5000))
returns table (id int, c varchar, freq int) as begin
    using sqlscript_string as lib;
    declare tv table(result varchar);
    tv = lib:split_to_table(:sentence, ' ');
    return select :id as id, result as c, count(result) as freq from :tv group by result;
end;
```

### 2) Build the Reducer

#### Reducer Function

```
create function reducer(in c varchar, in values table(id int, freq int))
returns table (c varchar, stmt_freq int, total_freq int) as begin
    return select :c as c, count(distinct(id)) as stmt_freq, sum(freq) as total_freq from :values;
end;
```

#### Final Code

```
do begin
    declare result table(c varchar, stmt_freq int, total_freq int);
    result = MAP_REDUCE(tab, mapper(tab.id, tab.sentence) group by c as X,
                        reducer(X.c, X));
    select * from :result order by c;
end;
```

Figure 198: MAP REDUCE Example



### LESSON SUMMARY

You should now be able to:

- Use declarative logic

# Learning Assessment

1. Declarative Logic can only use a limited set of SQLScript language.

*Determine whether this statement is true or false.*

- True
- False

2. Why do we use MAP\_MERGE operators?

*Choose the correct answer.*

- A To replace sequential FOR-loops and union patterns with a parallel operator to improve performance
- B To push back complex conditional logic to the source database when using remote data connections

## Learning Assessment - Answers

1. Declarative Logic can only use a limited set of SQLScript language.

*Determine whether this statement is true or false.*

True

False

Correct! Declarative Logic can only use a limited set of SQLScript language, essentially the language that does not 'control' the data flow or cause side effects. For more details, see HA150 Unit 4.

2. Why do we use MAP\_MERGE operators?

*Choose the correct answer.*

A To replace sequential FOR-loops and union patterns with a parallel operator to improve performance

B To push back complex conditional logic to the source database when using remote data connections

Correct! The MAP\_MERGE operator replaces sequential FOR-loops and union patterns with a parallel operator to improve performance. For more details, see HA150 Unit 4.

## Lesson 1

Implementing Imperative Logic

175

### UNIT OBJECTIVES

- Implement imperative logic



# Implementing Imperative Logic



## LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Implement imperative logic

## Coding with Imperative Statements

### What Is Imperative Logic?

Let's begin by reminding ourselves that declarative logic is side-effect-free. That means there is no change of status of the underlying objects either at the database or at the application level. Or put another way, declarative logic work on a "read-only" basis. By focusing purely on read-only operations, SAP HANA is able to fully optimize the whole processing logic and parallelize as much of the execution as possible.

However, writing code using only declarative language limits what you can achieve and sometimes you need to **control** the data flow logic using imperative (sometimes called orchestration) language. SQLScript supports imperative logic.



### Allows developer to control the flow of the logic within SQLScript

- Scalar variable manipulation
- DDL/DML logic
- FOR & WHILE loops
- Emptiness checks
- Branching logic based on some conditions, for example IF/ELSE
- Executed procedurally, but the SQLScript engine will make all efforts to optimize for parallelization



Figure 199: Features of Imperative Logic

SQLScript supports the use of imperative language.

Imperative language is used to implement data-flow and control-flow logic using constructs such as loops and conditionals. When you include these constructs, the code then executes procedurally and follows the logic of the loop or conditional. You should bear in mind that this reduces the possibilities for SAP HANA to apply optimizations to the code.

Examples of imperative logic:

- Scalar variable manipulation
- DDL/DML logic

- FOR & WHILE loops
- Emptiness checks
- Branching logic based on some conditions, for example IF/ELSE
- Executed procedurally, but the SQLScript engine will still continue to make all efforts to optimize for parallelization

**Note:**

The imperative logic is usually represented as a single node in the dataflow graph; thus, it is executed sequentially, and so the opportunities for optimizations are limited.

## Variables



### Scalar variables (local to procedures / functions)

#### Optionally be initialized with their declaration

- `DECLARE local_var int;`
- `DECLARE C int = 5;`
- `DECLARE a int DEFAULT 0;`

#### Assignment done using “=”

It can be referenced by using ":" as the prefix in the variable

- `local_var = :C + :a + 3 ;`



Figure 200: Scalar Variables

The local scalar and table variables are declared in the functions or the procedures by using the `DECLARE` keyword and are optionally initialized in the declaration.

The variables can also be flagged as read-only by using the `CONSTANT` keyword, and the consequence is that you cannot override the variable anymore. Also, if you use the `CONSTANT` keyword, it should have a default value and cannot be `NULL`.

When you want to assign a value to another variable, use `=` with the left side containing the variable whose value will be updated, as below:

```
local_var = :C + :a + 3 ;
```

It is also possible to define local variables inside `LOOP` / `WHILE` / `FOR` / `IF-ELSE` control structures.



## Table variables (local to procedures / functions)

### Defined by using the DECLARE keyword

#### Optionally be initialized with their declaration

- `DECLARE temp TABLE (n int);`
- `DECLARE outTab MY_TABLE_TYPE;`
- `DECLARE inTab MY_TABLE_TYPE = UNNEST (:arr) as (i);`
- `DECLARE temp CONSTANT MY_TABLE_TYPE DEFAULT SELECT * FROM TABLE;`

#### Assignment done using “=”

It can be referenced by using ":" as the prefix in the variable

- `outTab = Select * from :inTab;`

Figure 201: Table Variables

Assignment is possible more than once, overwriting the previous value stored in the scalar variable. At the same time, the SQLScript supports local variable declaration in a nested block. Local variables are only visible in the scope of the code block in which they are declared and defined, meaning the variable value inside the nested block is assigned inside the nested block. For example, if the variable only declared outside the nested block, but also assigned inside the nested block, the inside assignment will override the value assigned earlier outside of that nested block.



## Global session variables

**Used to share a scalar value between procedures and functions that are running only in the same session.**

**Not visible from another session.**

**Set in a procedure or a function**

- `SET 'MY_SES_VAR' = :new_value ;`

**Retrieve the session variable using SESSION\_CONTEXT function**

- `var = SESSION_CONTEXT('MY_SES_VAR');`

**Default value is NULL**

**Can be RESET to NULL using UNSET**

- `UNSET 'MY_VAR' ;`

**Cannot be used in READ ONLY procedures or functions**

Figure 202: Global Session Variables

If there is a need to pass the value of a variable between procedures and functions that are running in the same session, global session variables can be used in SQLScript. This variable value is only available in that session and not visible from another session. The session variables are of type STRING, and if it is not of type STRING, it will implicitly be converted of type STRING.

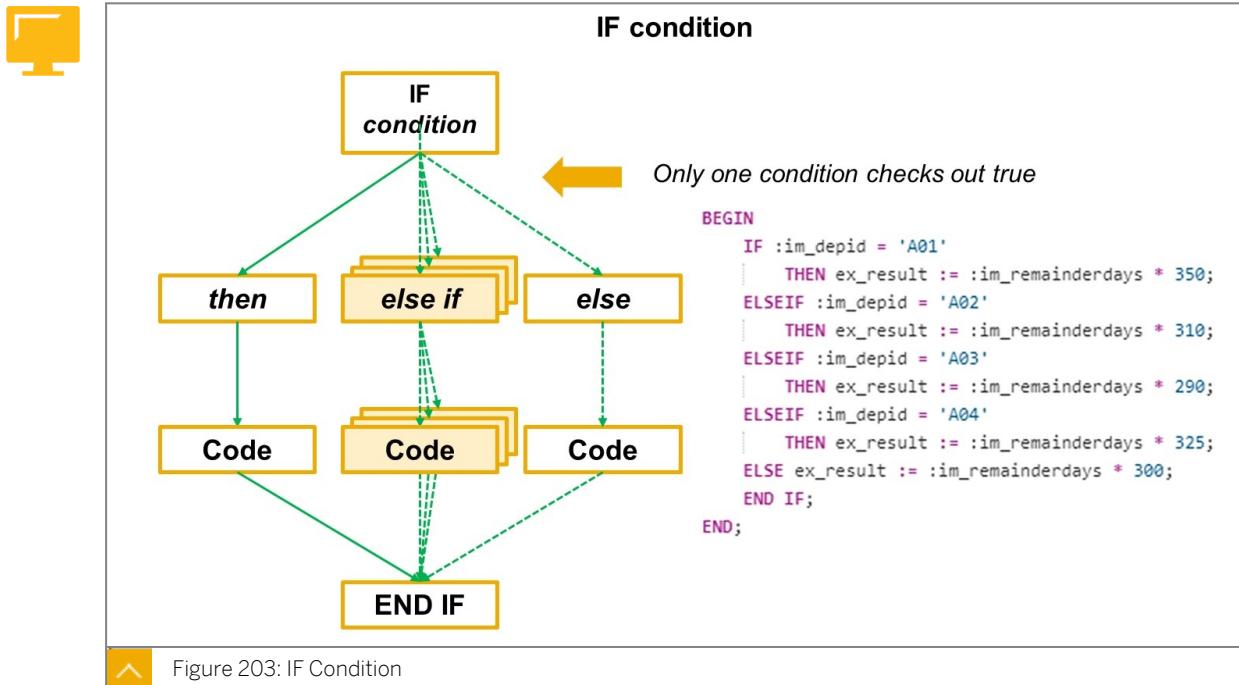
The global session variables are, by default, set to NULL values. They can be set and unset (reset to NULL) using the SET and UNSET command inside the procedures and/ or functions.

To retrieve the session variable, function SESSION\_CONTEXT (<key>) can be used. For example, the following function retrieves the value of session variable MY\_SES\_VAR.

```
var = SESSION_CONTEXT ('MY_SES_VAR');
SELECT SESSION_CONTEXT(<key>) "MY_VAR" FROM DUMMY;
SELECT SESSION_CONTEXT ('APPLICATION') "session context" FROM DUMMY;
```

Here the key is 'APPLICATION' and it returns the value for that specific key.

### Conditional Logic



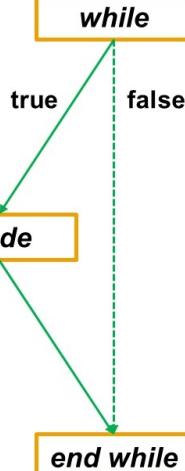
For the imperative logic, there are control statements like IF control statement (IF – ELSEIF – ELSE – END IF), and loop statements like WHILE (WHILE – DO – END WHILE) and FOR (FOR – DO – END FOR).

The IF control statement, as shown in the figure, will have one TRUE condition and will be executed, while all others will be FALSE and skipped.

The IF statement checks on the Boolean expression condition. If it is TRUE, the statements in the following THEN block are executed. The whole condition checking IF statement ends now with END IF, as all the other optional block of code is skipped. If the condition check on the first IF is FALSE, and if there is an ELSEIF or ELSE condition code is present, the process continues until one check evaluates TRUE. If nothing evaluates TRUE, then the code in the ELSE block, if present, will be executed.



## WHILE Loop



```

WHILE :v_index1 < 5
DO
  v_index2 := 0;
  WHILE :v_index2 < 5
    DO
      v_index2 := :v_index2 + 1;
    END WHILE;
    v_index1 := :v_index1 + 1;
  END WHILE;

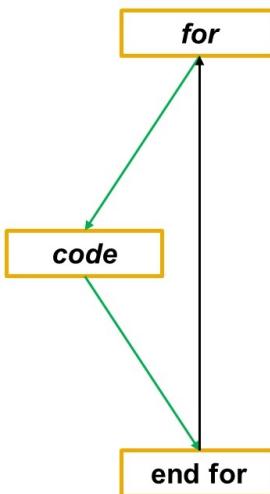
```

Figure 204: WHILE Loop

The WHILE loop executes the block of code in the body of the loop, as long as the Boolean expression at the condition check in the beginning of the loop evaluates to TRUE. If the condition evaluates FALSE, the whole block of code is skipped.



## FOR Loop



```

BEGIN
  FOR v_index1 IN -2 .. 2
    DO
      FOR v_index2 IN REVERSE 0 .. 5
        DO
          CALL ins_msg_proc(:v_index1 || '~' || :v_index2);
        END FOR;
    END FOR;

```

**REVERSE** will loop for the for clause in descending order

Figure 205: FOR Loop

The FOR loop iterates over a range of numeric values (like 1..10, or ten times one for each numeric value) and binds the current value to a variable in ascending order. Iteration starts with the value of starting value (1 in this case) and is incremented by one until the variable is greater than ending value (10 in this case).

You can add an optional keyword REVERSE, with this the FOR loop sequence to occur in a descending order.

If the start value is greater than the ending value in the range of numeric values, the loop will not be evaluated.



## BREAK and CONTINUE

```

FOR x IN 0 .. 10 DO
    IF :x < 3
    THEN
        ... CONTINUE ;
    END IF ;

    IF :x = 5
    THEN
        ... BREAK ;
    END IF ;
END FOR ;

```

### CONTINUE

Stop processing the current iteration in the loop, and start processing the next iteration immediately

### BREAK

Stop processing the loop, and exit it



Figure 206: BREAK and CONTINUE

The BREAK and CONTINUE statements provide internal control functionality for loops.

BREAK is used when you need to exit out of the current loop completely and move to the next code, if there is any, following the loop. The current loop will stop being processed.

CONTINUE is for skipping the current iteration of the loop and proceed processing the next iteration in the same loop. Unlike BREAK, the loop iteration is continued to be executed, if the condition of the loop evaluates TRUE.

### Working with Cursors

Cursors are used to fetch single rows from the result set returned by a query. When a cursor is declared, it is bound to the query. It is possible to parameterize the cursor query.



- Allows developers to iterate through a result set and perform row-based processing and calculations.
- Cursors can be defined either after the signature of the procedure and before the procedure's body or at the beginning of a block with the DECLARE statement
- The cursor is defined with a name, optionally a list of parameters, and an SQL SELECT statement

```

12  declare v_new_price decimal(15,2);
13  declare CURSOR c_products FOR
14      SELECT PRODUCTID, CATEGORY, PRICE
15      from "SAP_HANA_EPM_NEXT"."sap.hana.democontent.epmNext.data::MD.Products";
16
17  FOR cur_row as c_products DO
18
19      if :im_direction = 'INCREASE' then
20          v_new_price := cur_row.PRICE + (cur_row.PRICE * :im_rate);
21      elseif :im_direction = 'DECREASE' then
22          v_new_price := cur_row.PRICE - (cur_row.PRICE * :im_rate);
23      end if;
24
25      UPDATE "SAP_HANA_EPM_NEXT"."sap.hana.democontent.epmNext.data::MD.Products"
26      | SET PRICE = v_new_price where PRODUCTID = cur_row.PRODUCTID;
27
28  END FOR;
29
30  ex_products = select * from "SAP_HANA_EPM_NEXT"."sap.hana.democontent.epmNext.data::MD.Products"
31
32 END;

```

Figure 207: Working with Cursors

Cursors can be defined either after the signature of the procedure and before the procedure's body or at the beginning of a block with the DECLARE token. The cursor is defined with a name, optionally a list of parameters, and an SQL SELECT statement.

The cursor provides the functionality to iterate through a query result row by row. With SAP HANA 2.0 SPS03, an updatable cursor is available, which helps to change a record directly on the row to which the cursor is currently pointing.



Note:

Avoid using cursors when it is possible to express the same logic with SQL.  
Cursors cannot be optimized the same way SQL can.

## Working with Arrays



**Allows the developer to define and construct arrays within SQLScript**

**An array is an indexed collection of elements of a single data type**

**Operations supported:**

- Set elements
- Return elements
- Remove elements
- Concatenate two arrays
- Turn array into a table
- Turn a column of a table into an array
- Return cardinality of an array

```

CREATE PROCEDURE STUDENT01."PRODUCT_ARRAY"(
    |   OUT output_table STUDENT01."TT_PRODUCT_SALES"
)
LANGUAGE SQLSCRIPT
SQL SECURITY INVOKER
AS
*****BEGIN PROCEDURE SCRIPT *****
BEGIN
    DECLARE productid VARCHAR(20) ARRAY;
    DECLARE category VARCHAR(20) ARRAY;
    DECLARE price DECIMAL(15,2) ARRAY;

    productid[1] := 'ProductA';
    productid[2] := 'ProductB';
    productid[3] := 'ProductC';

    category[1] := 'CategoryA';
    category[2] := 'CategoryB';
    category[3] := 'CategoryC';

    price[1] := 19.99;
    price[2] := 29.99;
    price[3] := 39.99;
    |
    |
    output_table =
    UNNEST(:productId, :category, :price, :saleprice)
        AS ("PRODUCTID", "CATEGORY", "PRICE", "SALEPRICE");
END;

```

Figure 208: Working with Arrays

An array is an indexed collection of elements of a single data type. In the following section, we explore the varying ways to define and use arrays in SQLScript. The array can be used to set, return, or remove elements in an array, concatenate two arrays into a single array, turn arrays from multiple arrays into a table using the UNNEST function, and so on.

As shown in the figure, the UNNEST function converts one or many arrays into a table. The result table includes a row for each element of the specified array. The result of the UNNEST function needs to be assigned to a table variable.

ARRAY\_AGG function converts a column of a single table into an array. While converting the column of a table using the ARRAY\_AGG function, the columns can be placed in the array using ORDER BY to do it in ascending or descending order and, also place the NULLs (null values) in the beginning or in the end using NULLS FIRST or NULLS LAST.

```
ARRAY_AGG(<table_name>.<column_name> ORDER BY <column_name> DESC NULLS FIRST) ;
```



**Allows the developer to access any cell (read/write) of an intermediate table variable or table parameter directly**

- Access via <table>.<column>[<index>] notation

```

1  PROCEDURE "HA150::build_emp_list" (
2      out ex_emp table (emp_id nvarchar(10),
3                          "name.first" nvarchar(10),
4                          "name.last" nvarchar(10),
5                          salamount integer)
6  )
7  LANGUAGE SQLSCRIPT
8  SQL SECURITY INVOKER
9  -- DEFAULT SCHEMA "STUDENT01"
10 READS SQL DATA AS
11 BEGIN
12     ex_emp.emp_id[1] = '10001';
13     ex_emp."name.first"[1] = 'Mickey';
14     ex_emp."name.last"[1] = 'Mouse';
15     ex_emp.salamount[1] = '2000';
16
17     ex_emp.emp_id[2] = '10002';
18     ex_emp."name.first"[2] = 'Don';
19     ex_emp."name.last"[2] = 'Duck';
20     ex_emp.salamount[2] = '3000';
21 END
22 ;
23 
```

The screenshot shows the SAP SQLScript interface. On the left, the code for the stored procedure 'HA150::build\_emp\_list' is displayed. On the right, the 'Result' tab shows the output of the 'CALL' command, which displays two rows of data from the table variable 'ex\_emp'.

|   | EMP_ID | name.first | name.last | SALAMOUNT |
|---|--------|------------|-----------|-----------|
| 1 | 10001  | Mickey     | Mouse     | 2000      |
| 2 | 10002  | Don        | Duck      | 3000      |

Figure 209: Access any Cell

The index-based cell access allows you random access (read and write) to each cell of table variable.

<table\_variable>.<column\_name>[<index>]

The index is a number which can be a number between 1 to 2^31 or an SQL expression, or a Scalar UDF that returns a number.

Reading and writing can be performed on a cell of a table variable using the index-based cell access.

When writing a value to the table variable, use the following syntax:

```
ex_emp.emp_id[1] = '10001';
```

When reading, you must have preceded the table variable with a ":" as illustrated below:

```
outvar = :intab.B[100];
```

Restrictions:

- Physical tables cannot be accessed.
- Not applicable in SQL queries like SELECT :MY\_TABLE\_VAR.COL[55] AS A FROM DUMMY. You need to assign the value to be used to a scalar variable first.

### Commit and Rollback

The COMMIT and ROLLBACK commands are supported natively in SQLScript.



- Supported in procedures only, not supported for scalar or table UDFs
- COMMIT statement commits the current transaction and all changes before the COMMIT statement
- ROLLBACK statement rolls back the current transaction and undoes all changes since the last COMMIT
- Transaction boundary is not tied to the procedure block, so if there are nested procedures that contain COMMIT/ROLLBACK then all statements in the top-level procedure are affected.
- If dynamic SQL was used in the past to execute COMMIT and ROLLBACK statements, it is recommended to replace all occurrences with the native command because they are more secure.



Figure 210: Commit and Rollback

The COMMIT command commits the current transaction and all changes before the COMMIT command are written to persistence.

The ROLLBACK command rolls back the current transaction and undoes all changes since the last COMMIT.

The RESIGNAL statement is used to pass on the exception that is handled in the exit handler. Besides passing on the original exception by simply using RESIGNAL, you can also change some information before passing it on.

The RESIGNAL statement raises an exception on the action statement in exception handler. If the error code is not specified, RESIGNAL will throw the caught exception:

```
BEGIN
DECLARE MYCOND CONDITION FOR SQL_ERROR_CODE 10001;
DECLARE EXIT HANDLER FOR MYCOND RESIGNAL;

INSERT INTO MYTAB VALUES (1);

SIGNAL MYCOND SET MESSAGE_TEXT = 'my error';

END; CALL MYPROC;
```



Note:

RESIGNAL statement can only be used in the exit handler.

## Dynamic SQL



**Dynamic SQL allows you to construct an SQL statement during the execution time of a procedure.**

**Dynamic SQL allows you to use variables**

**Provides more flexibility when creating SQL statements**

### EXEC

- EXEC 'SELECT A FROM T1' INTO A\_COPY ;

### EXECUTE IMMEDIATE

- EXECUTE IMMEDIATE 'SELECT \* FROM tab ORDER BY i';

### APPLY\_FILTER

- Dynamic filter on a table or a table variable



Figure 211: Dynamic SQL

EXEC executes the SQL statement <sql-statement> passed in a string argument. EXEC does not return any result set if <sql\_statement> is a SELECT statement. You have to use EXECUTE IMMEDIATE for that purpose.

If the query returns a single row, you can assign the value of each column to a scalar variable by using the INTO clause.

EXECUTE IMMEDIATE executes the SQL statement passed in a string argument. The results of queries executed with EXECUTE IMMEDIATE are appended to the procedures result iterator.

The APPLY\_FILTER function applies a dynamic filter on a table or table variable. Logically, it can be considered a partial dynamic SQL statement.

Some of the disadvantages of the dynamic SQL are:

- Opportunities for optimizations are limited.
- Statement is potentially recompiled every time the statement is executed.
- Cannot use SQLScript variables in the SQL statement.
- Cannot bind the result of a dynamic SQL statement to an SQLScript variable.
- Must be very careful to avoid SQL injection bugs that might harm the integrity or security of the database.

SQL Injection bugs: If SQLScript procedure needs execution of dynamic SQL statements where the parts of it are derived from untrusted input (for example, user interface), there is a danger of an SQL injection attack.

The following functions can be utilized to prevent it:

- ESCAPE\_SINGLE\_QUOTES - used for variables containing a SQL string literal
- ESCAPE\_DOUBLE\_QUOTES - used for variables containing a delimited SQL identifier
- IS\_SQL\_INJECTION\_SAFE - used to check that a variable contains safe simple SQL identifiers

## EXISTS and IN Predicates

Both the **EXISTS** keyword as well as the **IN** keyword are “predicate” keywords, meaning that in combination with other keywords, the developer is able to create an expression to determine whether something is TRUE, FALSE, or UNKNOWN.



- **EXISTS** allows the developer to evaluate if a result set has any entries.
- **IN** allows the developer to evaluate if a given value matches one of a enumerated set of values.

```

7
8 - IF EXISTS (SELECT * FROM "MD.Products" WHERE category = :im_product_cat ) THEN
9
10 - IF :im_product_cat IN ('Notebooks', 'PC', 'Handheld') THEN
11   ex_product_class = 'Asset';
12 - ELSEIF :im_product_cat IN ('Mice', 'Keyboard', 'Speakers') THEN
13   ex_product_class = 'Other';
14 END IF;
15
16 END IF;
17

```



Figure 212: EXISTS and IN Predicates

In the case of an expression using EXISTS, the expression will return TRUE if the sub-query contained in the expression returns a minimum of one record (in other words, not an empty result set). Otherwise, the expression will return FALSE (in other words, an empty result set). EXISTS can be used in both IN and WHILE expressions.

In the code in the figure, a product category (the :im\_product\_catvariable) is used in a where clause for a SELECT statement from the MD.Productstable. If there is at least one record in the table assigned to this category, then the EXISTS (and therefore the IF also) will evaluate to TRUE and to FALSE otherwise.

In the case of an expression containing an IN, the expression will return TRUE if the value on the left side of the IN matches at least one of a defined list of values on the right side of the IN. Otherwise, the expression will return FALSE. The value on the left side of the IN can be the result of a sub query. In addition, a tuple (multiple values to match) can be used that is, (city, country) IN ((‘Los Angeles’, ‘UnitedStates’), (‘London’, ‘England’));. IN can be used in expressions formulated with IF and WHILE as well as WHERE clauses on SELECT expressions.

In the code in the figure, a product category (the :im\_product\_catvariable) will have a value. The IF evaluates whether the product category is equal to any of a defined set of values. If that is the case then the IF will evaluate to TRUE. Otherwise, the IF evaluates to FALSE and the ELSEIF will evaluate the product category against a different set of values. If there is a match the ELSEIF will evaluate to TRUE. Otherwise, the ELSEIF evaluates to FALSE and the inner IF above concludes.

## Transactional Savepoints

SQLScript supports transactional savepoints.



- Transactional Savepoints, that allow the rollback of a transaction to a defined point allowing partial rollback.
    - **SAVEPOINT <name>** - Definition of a Savepoint
    - **ROLLBACK TO SAVEPOINT <name>**
      - Rollback to a specific Savepoint
    - **RELEASE SAVEPOINT <name>** - Release a Savepoint
  - Supports multiple savepoints. Rollback to a specific savepoint will release all following (or nested) savepoints. Commit or Rollback releases all savepoints.

```
5  create column table myProducts( productid nvarchar(20) );
6
7  create procedure "test_save_points"
8      out ex_products table (productid nvarchar(20));
9  LANGUAGE SQLSCRIPT
10  SQL SECURITY INVOKER AS
11 begin
12
13     insert into myProducts values('HT-1000');
14     insert into myProducts values('HT-1100');
15     insert into myProducts values('HT-1200');
16     SAVEPOINT mySavepoint;
17
18     insert into myProducts values('HT-2000');
19     insert into myProducts values('HT-2100');
20     insert into myProducts values('HT-2200');
21     ROLLBACK TO SAVEPOINT mySavepoint;
22
23     ex_products = select * from myProducts;
24     RELEASE SAVEPOINT mySavepoint;
25 end;
26
27 call "test_save_points"();
```

| Rows (3) |         | PRODUCTID |
|----------|---------|-----------|
| 1        | HT-1000 |           |
| 2        | HT-1100 |           |
| 3        | HT-1200 |           |

Figure 213: Transactional Savepoints

A savepoint is a way of implementing sub-transactions (also known as nested transactions) within a relational database management system by indicating a point within a transaction that can be "rolled back to" without affecting any work done in the transaction before the savepoint was created.

Savepoints are useful for implementing complex error recovery in database applications. If an error occurs in the midst of a multiple-statement transaction, the application may be able to recover from the error (by rolling back to a savepoint) without needing to abort the entire transaction.

All statements for transactional savepoints must be implemented within a procedure or library. For library implementations, transactional savepoint statements can only go within procedures contained within the library. Scalar UDFs or Table UDFs cannot contain any transactional savepoint statements.

To define a savepoint, the developer will use the “`SAVEPOINT <name>`” SQL statement. The “`<name>`” parameter cannot have spaces but otherwise can be any alphanumeric string and serves to identify that particular savepoint. Multiple savepoints are permitted provided that the names are unique. The `<name>` parameter is not case sensitive.

The “ROLLBACK TO SAVEPOINT <name>” statement rolls back all statements starting with the first statement issued after the “SAVEPOINT <name>” statement (wherever the developer first mentioned it) and ending with the last statement before the “ROLLBACK TO SAVEPOINT <name>” statement itself was issued.

The “RELEASE SAVEPOINT <name>” removes that savepoint from future consideration. Any reference to the savepoint (with a ROLLBACK statement) after the release statement triggers an error. All transactional savepoints are automatically released when the relevant procedure ends.



## LESSON SUMMARY

You should now be able to:

- Implement imperative logic



## Learning Assessment

1. Why do we use imperative logic?

*Choose the correct answer.*

- A To take control of the data flow using conditional logic
- B To improve performance of data-intensive tasks
- C To define reusable modules of code

2. As well as reading records one by one, using cursors allows you to change the record, too.

*Determine whether this statement is true or false.*

- True
- False

3. What states can be returned using a combination of EXISTS and IN predicates?

*Choose the correct answers.*

- A False
- B Incorrect
- C True
- D Unknown

4. A transactional savepoint allows you to perform a partial rollback within a complex transaction.

*Determine whether this statement is true or false.*

- True
- False

## Learning Assessment - Answers

1. Why do we use imperative logic?

*Choose the correct answer.*

- A To take control of the data flow using conditional logic  
 B To improve performance of data-intensive tasks  
 C To define reusable modules of code

Correct! Imperative logic allows the developer to take control of the data flow using conditional logic. It actually reduces performance and does not improve it.

2. As well as reading records one by one, using cursors allows you to change the record, too.

*Determine whether this statement is true or false.*

- True  
 False

Correct! Since SPS03, it is possible to change data using cursor processing.

3. What states can be returned using a combination of EXISTS and IN predicates?

*Choose the correct answers.*

- A False  
 B Incorrect  
 C True  
 D Unknown

Correct! States true, false, and unknown can be returned using a combination of EXISTS and IN predicates.

4. A transactional savepoint allows you to perform a partial rollback within a complex transaction.

*Determine whether this statement is true or false.*

True

False

Correct! A transactional savepoint allows you to roll back a complex transaction only partially without having to abort the entire transaction.



## Lesson 1

Working with Temporal Tables

195

### UNIT OBJECTIVES

- Work with temporal tables



# Working with Temporal Tables



## LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Work with temporal tables

## Temporal Tables



SAP HANA already supports **History Tables**.



- Allow time-based queries on previous versions of a record.
- Write-operations don't physically overwrite an existing record, but insert a new version.
- Each version has a validity period, which is **based on commit IDs**.
- **Proprietary** implementation. Doesn't match SQL standard.

## Challenges

The temporal information of a record ...

- ... is not externally accessible, e.g., by SQL.
- ... is specific to the database instance.
- ... has to be mapped to an accessible time-format, i.e., a **TIMESTAMP**.

Figure 214: Transactional tables extend functions of History Tables

For a long time, HANA SQL has supported the creation of history tables. This was done by the developer issuing a "CREATE HISTORY COLUMN TABLE <table\_name>" SQL statement where <table\_name> was the name of the table being created. As a result of this statement, a log of all transactions executed against a history table would be permanently logged in system tables. The log contained the unique commit ID of the transaction (one is created for each and every commit SQL statement executed in SQL) along with the UTC timestamp of when that commit executed. The commit ID and/or the UTC timestamp allowed a developer, when selecting records from a table, to retrieve not just the current version of the record but any and all prior versions of that record that existed at any point in the past (known as a time travel query).

To retrieve prior versions of a record, the developer would execute the "SET HISTORY SESSION TO <when>" statement where <when> would be set to "1" for the unique commit ID, "2" for the UTC timestamp of when the commit occurred, or "3" using the SQL keyword "NOW" (indicating to only retrieve current versions of the record).

While useful, there were several limitations to using the history table approach. The first was that the commit ID was HANA-instance-specific and could not be migrated to other HANA instances. In addition, the commit ID was not externally accessible (from ABAP, for

example). Finally, the design of history tables was not based off of any SQL standard (such as SQL 92, for example).

SQL:2011 or [ISO/IEC 9075:2011](#) was officially adopted in December 2011. Among other areas, it established a standardized way of designing temporal tables in database systems. SQL as of SPS03 now supports the creation of temporal tables in accordance with the SQL:2011 specification.

A temporal table gives the developer not only the ability to execute “time travel” queries, but also to define validity periods for a record.



**Temporal Tables** are tables whose records are associated with one or more **temporal periods**.



**System-Versioned Tables** allow **change tracking** on database tables.

- Validity periods are automatically **maintained by the database** whenever a record is changed.
- Retrospective changes to the validity period of a record are strictly prohibited. A new version is inserted for every change to preserve a record's history.
- Temporal information must have a **very high granularity**

**Application-Time Period Tables** capture the time, in which a record is **valid in the real world**.

- Validity periods are **determined by the application**.
- Application can **update validity period** of a record, e.g., to correct errors.
- Temporal information can arbitrarily reflect the **past, present or future at any granularity**.

**Bitemporal Tables** combine system-versioned tables and application-time period tables.

**Temporal periods** are based on already existing SQL data types.

- Two columns, defining START and END.
- Temporal data type, e.g., TIMESTAMP.

**temporal period**

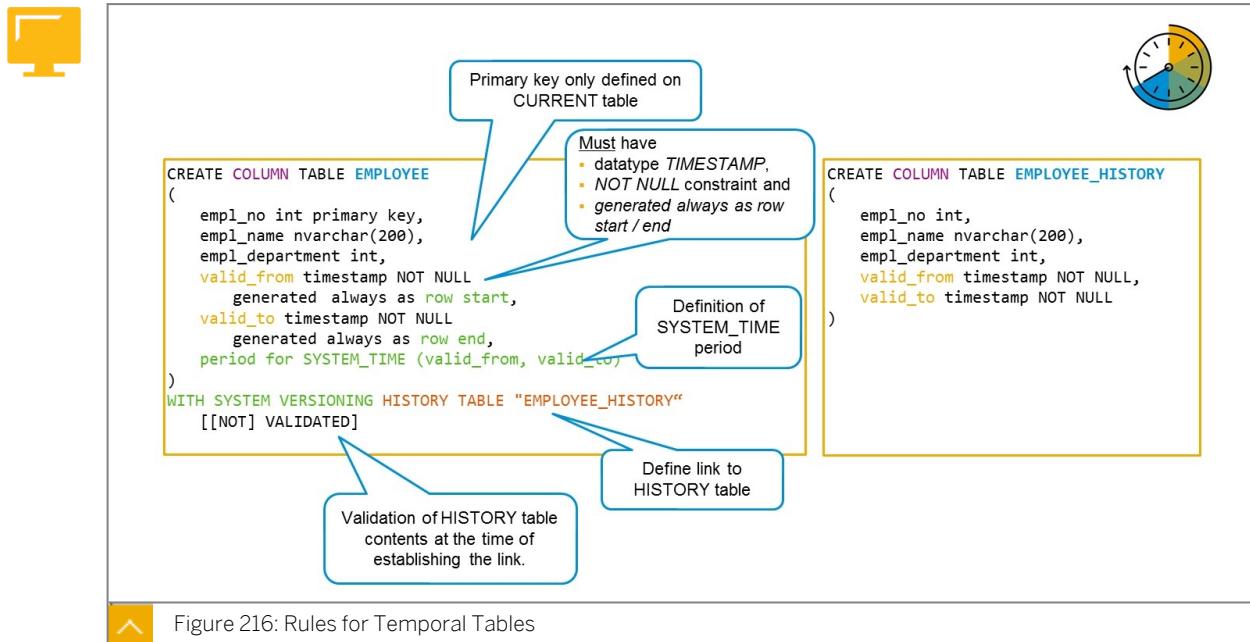
| ITEM_ID | STOCK | VALID_FROM          | VALID_TO            |
|---------|-------|---------------------|---------------------|
| 304     | 112   | 2017-04-28 10:32:44 | ∞                   |
| 305     | 5423  | 2017-04-28 10:31:32 | 2017-04-28 10:32:26 |
| 306     | 345   | 2017-04-28 10:31:17 | ∞                   |
| 305     | 5422  | 2017-04-28 10:32:26 | ∞                   |

Figure 215: Introducing Temporal Tables

When creating temporal tables, the developer has several choices.

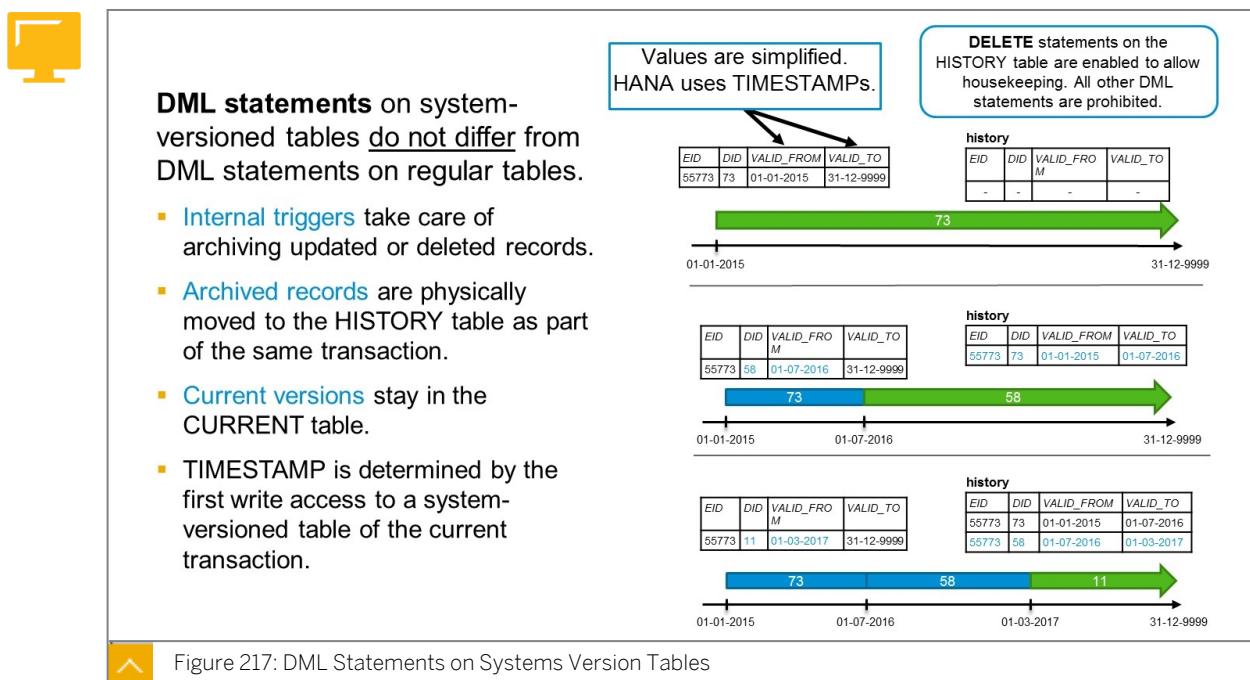
1. The first option is to create **system-versioned tables**, which will result in any and all changes to records in the table being stored in an associated history table (not to be confused with history tables discussed previously). With system-versioned tables, specific validity period columns will be added to the table and the type of those columns will be the built-in HANA SQL type TIMESTAMP. With system-versioned tables, any and all changes to a record result in a new version (and thus the start of a new validity period) of the record being created. Under no circumstances with system-versioned tables can the validity period of a record be retroactively adjusted. Therefore, every version of the record that has ever existed will be preserved with a corresponding start and end validity period.
2. The second option is to create **application-time period tables**. With this option, the developer will create their own validity period columns (with whatever names and SQL data types preferred) for a record that is freely modifiable based on business logic needs of record maintenance. Therefore, the start and end validity periods are adjustable based on business needs. In addition, the developer can issue transactions that change certain fields of a record without necessarily changing the validity period of the record. These validity period columns will be ignored for all system-versioned table operations.
3. The final option that the developer has in regard to temporal table definition is to apply both approaches. This is known as a **bitemporal table** definition. In this case, both system-versioned validity period fields and application-time period validity period fields will be

created, allowing the developer to implement both change tracking and business logic validity in the table design.



Column tables are the only type supported for system-versioned tables. In addition, only the HANA SQL type `TIMESTAMP` is supported for the “valid\_from” and “valid\_to” fields.

The structure of the history table must match the structure of the column table for which it is designated as the history table. The “VALIDATED” HANA SQL keyword can be used to confirm that the structure of the history table is correct, but is not required.



When transactions execute that result in changes to a system-versioned table, the timestamp is determined by the first write access to a system-versioned table of the current transaction.

Current versions stay in the CURRENT table. Internal triggers take care of archiving updated or deleted records and archived records are physically moved to the HISTORY table as part of the same transaction.



### Time travel



... FOR SYSTEM\_TIME AS OF <timestamp>

- Returns all records that were active right at <timestamp>.

| ITEM_ID | STOCK | VALID_FROM          | VALID_TO            |
|---------|-------|---------------------|---------------------|
| 306     | 345   | 2017-04-28 10:31:17 | 2017-04-28 10:35:04 |
| 305     | 5423  | 2017-04-28 10:31:32 | 2017-04-28 10:32:26 |
| 305     | 5422  | 2017-04-28 10:32:26 | 2017-04-28 10:35:26 |
| 304     | 112   | 2017-04-28 10:32:44 | ∞                   |
| 306     | 545   | 2017-04-28 10:35:04 | ∞                   |
| 305     | 5421  | 2017-04-28 10:35:26 | 2017-04-28 10:37:46 |
| 305     | 5410  | 2017-04-28 10:37:46 | ∞                   |

```
SELECT ITEM_ID, STOCK
FROM STOCKS
FOR SYSTEM_TIME AS OF "2017-04-28 10:36:00"
```

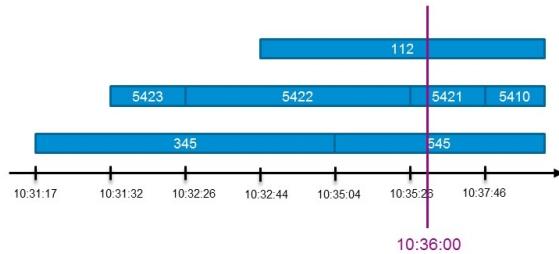


Figure 218: Time Travel



### Reading ranges of versions



... FOR SYSTEM\_TIME FROM <t1> TO <t2>

- Returns all records that were active any time within the specified time range.
- Records that became active right at <t2> are not included.

```
SELECT ITEM_ID, STOCK
FROM STOCKS
FOR SYSTEM_TIME
FROM "2017-04-28 10:33:00" TO "2017-04-28
10:37:46"
```

| ITEM_ID | STOCK | VALID_FROM          | VALID_TO            |
|---------|-------|---------------------|---------------------|
| 306     | 345   | 2017-04-28 10:31:17 | 2017-04-28 10:35:04 |
| 305     | 5423  | 2017-04-28 10:31:32 | 2017-04-28 10:32:26 |
| 305     | 5422  | 2017-04-28 10:32:26 | 2017-04-28 10:35:26 |
| 304     | 112   | 2017-04-28 10:32:44 | ∞                   |
| 306     | 545   | 2017-04-28 10:35:04 | ∞                   |
| 305     | 5421  | 2017-04-28 10:35:26 | 2017-04-28 10:37:46 |
| 305     | 5410  | 2017-04-28 10:37:46 | ∞                   |

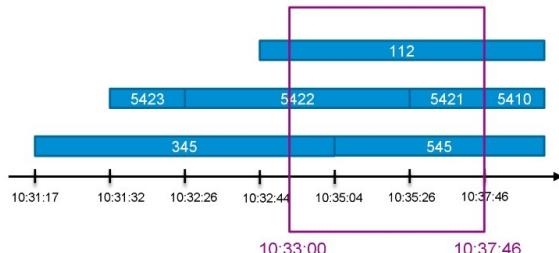


Figure 219: Ranges — From



### Reading ranges of versions

... FOR SYSTEM\_TIME BETWEEN <t1> AND <t2>

- Returns all records that were active any time within the specified time range.
- Records that became active right at <t2> are included as well.

| ITEM_ID | STOCK | VALID_FROM          | VALID_TO            |
|---------|-------|---------------------|---------------------|
| 306     | 345   | 2017-04-28 10:31:17 | 2017-04-28 10:35:04 |
| 305     | 5423  | 2017-04-28 10:31:32 | 2017-04-28 10:32:26 |
| 305     | 5422  | 2017-04-28 10:32:26 | 2017-04-28 10:35:26 |
| 304     | 112   | 2017-04-28 10:32:44 | ∞                   |
| 306     | 545   | 2017-04-28 10:35:04 | ∞                   |
| 305     | 5421  | 2017-04-28 10:35:26 | 2017-04-28 10:37:46 |
| 305     | 5410  | 2017-04-28 10:37:46 | ∞                   |

included as well →

```
SELECT ITEM_ID, STOCK
FROM STOCKS
FOR SYSTEM_TIME
BETWEEN "2017-04-28 10:33:00" AND "2017-04-28
10:37:46"
```

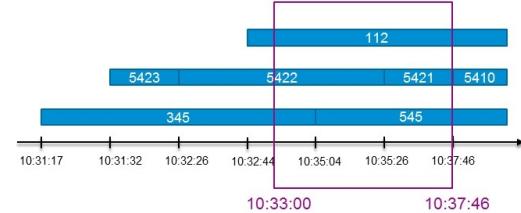


Figure 220: Ranges — Between



**Language enhancements** according to the SQL:2011 standard include

- **Time period definitions** based on two regular table columns, representing start and end time with close-open semantics.
- Query language extensions using **temporal predicates**, e.g.,
  - CONTAINS,
  - OVERLAPS,
  - EQUALS,
  - PRECEDES,
  - SUCCEEDS, ...

Figure 221: Language Enhancements

These are additional keywords that can be used to formulate queries against temporal tables.



### LESSON SUMMARY

You should now be able to:

- Work with temporal tables



## Learning Assessment

1. What are temporal tables?

*Choose the correct answer.*

- A Tables that persist only during a session
- B Tables that support time travel queries

## Learning Assessment - Answers

1. What are temporal tables?

*Choose the correct answer.*

- A Tables that persist only during a session
- B Tables that support time travel queries

Correct! Temporal tables support queries that request data as of a specific date and time, in other words, time travel.

## Lesson 1

Using OLAP Analytic Features

205

### UNIT OBJECTIVES

- Introduce OLAP analytic features
- Use SQL Group By features
- Use window framing in SQL



## Using OLAP Analytic Features



### LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Introduce OLAP analytic features
- Use SQL Group By features
- Use window framing in SQL

### OLAP Analytic Features

SQL is a set-oriented language. It is particularly effective when accessing a set of rows, as opposed to a single row at a time.

In most languages, complex data analysis requires that data be accessed in a loop, processing a row at a time. However, ANSI SQL-99 and 2003 include the following features that enable complex data analysis within a SELECT statement:

- ANSI SQL-99: GROUP BY enhancements
- ANSI SQL-2003: window framing

SAP HANA supports these SQL features.

### GROUP BY Enhancements

Online Analytical Processing (OLAP) GROUP BY enhancements allow application developers to write complex SQL statements to generate valuable results:



- To generate multiple groupings of data in a single SELECT statement, use GROUP BY GROUPING SETS.
- To create a sparse multi-dimensional result set that contains multiple levels of aggregation, use GROUP BY ROLLUP.
- To create a multi-dimensional cube as a result set, use GROUP BY CUBE.

These features are standard with ANSI SQL-99 and are supported by SAP HANA.

### Group By Grouping Sets

The GROUP BY GROUPING SETS parameter generates multiple groupings of data in a single statement.

The following is an example of the syntax to use with the GROUP BY GROUPING SETS parameter:



```
SELECT BRAND, COLOR, AVG(HP)
FROM TRAINING.CAR
```

```
WHERE OWNER = 'H03'
GROUP BY GROUPING SETS ((BRAND), (COLOR), (BRAND, COLOR));
```

This query returns the following result:



Table 7: GROUP BY GROUPING SETS Result

|    | BRAND    | COLOR | AVG(HP) |
|----|----------|-------|---------|
| 1  | Audi     | ?     | 116     |
| 2  | BMW      | ?     | 184     |
| 3  | Mercedes | ?     | 170     |
| 4  | Renault  | ?     | 90      |
| 5  | Skoda    | ?     | 136     |
| 6  | VW       | ?     | 120     |
| 7  | ?        | black | 142     |
| 8  | ?        | blue  | 150     |
| 9  | ?        | red   | 90      |
| 10 | ?        | white | 170     |
| 11 | Audi     | blue  | 116     |
| 12 | BMW      | blue  | 184     |
| 13 | Mercedes | black | 170     |
| 14 | Mercedes | white | 170     |
| 15 | Renault  | red   | 90      |
| 16 | Skoda    | black | 136     |
| 17 | VW       | black | 120     |

Rows 1 to 6 of the output are grouped by BRAND.

Rows 7 to 10 of the output are grouped by COLOR.

Rows 11 to 17 of the output are grouped by BRAND and COLOR.

### Group By Rollup

The GROUP BY ROLLUP parameter generates multiple levels of aggregation in a single statement.

The following is an example of the syntax to use with the GROUP BY ROLLUP parameter:



```
SELECT BRAND, COLOR, AVG (HP)
FROM TRAINING.CAR
WHERE OWNER = 'H03'
GROUP BY ROLLUP (BRAND, COLOR);
```

This query returns the following result:



Table 8: GROUP BY ROLLUP Result

|    | BRAND    | COLOR | AVG(HP)    |
|----|----------|-------|------------|
| 1  | Audi     | blue  | 116        |
| 2  | BMW      | blue  | 184        |
| 3  | Mercedes | black | 170        |
| 4  | Mercedes | white | 170        |
| 5  | Renault  | red   | 90         |
| 6  | Skoda    | black | 136        |
| 7  | VW       | black | 120        |
| 8  | Audi     | ?     | 116        |
| 9  | BMW      | ?     | 184        |
| 10 | Mercedes | ?     | 170        |
| 11 | Renault  | ?     | 90         |
| 12 | Skoda    | ?     | 136        |
| 13 | VW       | ?     | 120        |
| 14 | ?        | ?     | 140.857142 |

Rows 1 to 7 of the output are grouped by BRAND and COLOR.

Rows 8 to 13 of the output are grouped by BRAND.

Row 14 of the output is the grand total.

### Group By Cube

The GROUP BY CUBE parameter generates multiple levels of aggregation in every dimension.

The following is an example of the syntax to use with the GROUP BY CUBE parameter:



```
SELECT BRAND, COLOR, AVG(HP)
FROM TRAINING.CAR
WHERE OWNER = 'H03'
GROUP BY CUBE (BRAND, COLOR);
```

This query returns the following result:



Table 9: GROUP BY CUBE Result

|   | BRAND    | COLOR | AVG(HP) |
|---|----------|-------|---------|
| 1 | Audi     | blue  | 116     |
| 2 | BMW      | blue  | 184     |
| 3 | Mercedes | black | 170     |
| 4 | Mercedes | white | 170     |
| 5 | Renault  | red   | 90      |

|    | BRAND    | COLOR | AVG(HP)    |
|----|----------|-------|------------|
| 6  | Skoda    | black | 136        |
| 7  | VW       | black | 120        |
| 8  | ?        | black | 142        |
| 9  | ?        | blue  | 150        |
| 10 | ?        | red   | 90         |
| 11 | ?        | white | 170        |
| 12 | Audi     | ?     | 116        |
| 13 | BMW      | ?     | 184        |
| 14 | Mercedes | ?     | 170        |
| 15 | Renault  | ?     | 90         |
| 16 | Skoda    | ?     | 136        |
| 17 | VW       | ?     | 120        |
| 18 | ?        | ?     | 140.857142 |

Rows 1 to 7 of the output are grouped by BRAND and COLOR.

Rows 8 to 11 of the output are grouped by COLOR.

Rows 12 to 17 of the output are grouped by BRAND.

Row 18 is the grand total.

## Window Partitioning

### Window Framing

Window framing is an important SQL construct for OLAP operations. It enables users to perform the following tasks:



- Divide query result sets into groups of rows called partitions.
- Determine subsets of rows to aggregate with respect to the current row.

Window framing is standard with ANSI SQL:2003 and SAP HANA supports a variety of aggregate functions within window framing.

### Window Frame Components

Window frames have the following three components:



- Partitioning: the division of a user-specified result set (input rows) using the PARTITION BY clause.
- Ordering: the arrangement of results (rows) within each partition using the WINDOW ORDER clause.
- Framing: the definition of a moving frame of rows using the BETWEEN clause.

These components are defined within a SELECT statement using the OVER() clause.

## Window Partitioning

Window partitioning enables the user to divide user-specified input rows using a PARTITION BY clause.

The partition is defined by one or more value expressions separated by commas. The expression list can consist of items such as columns, operations, and functions. The partition data is implicitly sorted and the default sort order is ascending.

Partitioning is optional. If a window partition clause is not specified, the input rows are treated as a single partition.

The following is an example of the syntax to use with the PARTITION BY clause:

```
PARTITION BY <WINDOW PARTITION EXPRESSION LIST>
```

The following is an example of the syntax to use with the PARTITION BY clause within a SELECT statement:

```
SELECT ...
RANK() OVER (PARTITION BY customer_key ...)
```

## Window Ordering

Window ordering is the arrangement of results (rows) within each window partition using a WINDOW ORDER clause.

The expression list can consist of one or more expressions separated by commas. They consist of items such as columns, operations, and functions.

Ordering is optional. If a window ordering clause is not specified, the input rows can be processed in an arbitrary order.

The following is an example of the syntax to use with the WINDOW ORDER clause:

```
<WINDOW ORDER CLAUSE> :: = <ORDER SPECIFICATION>
```

The following are examples of the syntax to use with the WINDOW ORDER clause within a SELECT statement:

```
SELECT ...
RANK() OVER (ORDER BY sum(amount) desc ...)

SELECT ...
OVER (ORDER BY customer_key ...)
```

## Window Ordering Example

The following is an example of the syntax to use with the WINDOW ORDER clause:



```
SELECT product_id, SUM(amount),
       RANK() OVER ( ORDER BY sum(amount) desc) AS RANK
FROM TRAINING.SALES_DATA
GROUP BY product_id
ORDER BY RANK
```

This query returns the following result:



Table 10: WINDOW ORDER Result

|   | PRODUCT_ID | SUM(AMOUNT) | RANK |
|---|------------|-------------|------|
| 1 | 20002      | 12,813.8    | 1    |
| 2 | 20008      | 7,840       | 2    |

|   | PRODUCT_ID | SUM(AMOUNT) | RANK |
|---|------------|-------------|------|
| 3 | 20007      | 4,896.3     | 3    |
| 4 | 20006      | 3,189.6     | 4    |
| 5 | 20004      | 483.5       | 5    |
| 6 | 20003      | 472.5       | 6    |

In this example, the data is sorted as follows:

- Select rows from the SALES\_DATA table.
- Group rows by PRODUCT\_ID.
- Calculate the rank of each group based on the SUM(AMOUNT) aggregate.
- Order the rows by SUM(AMOUNT) descending.
- List the PRODUCT\_ID, SUM(AMOUNT), and RANK.

### Window Ordering and Partitioning Example

The following is an example of the syntax to use with window ordering and partitioning:



```
SELECT customer_id, product_id, SUM(amount),
       RANK() OVER (PARTITION BY customer_id
                    ORDER BY SUM(amount) DESC) AS RANK
FROM TRAINING.SALES_DATA
GROUP BY customer_id, product_id
ORDER BY customer_id, rank
```

This query returns the following result:



Table 11: Window Ordering and Partitioning Result

|    | CUSTOMER_ID | PRODUCT_ID | SUM(AMOUNT) | RANK |
|----|-------------|------------|-------------|------|
| 1  | 1000        | 20002      | 2,876.4     | 1    |
| 2  | 1000        | 20006      | 900         | 2    |
| 3  | 1000        | 20007      | 564.3       | 3    |
| 4  | 1000        | 20003      | 112.5       | 4    |
| 5  | 1000        | 20004      | 67.5        | 5    |
| 6  | 1001        | 20008      | 4,792       | 1    |
| 7  | 1001        | 20007      | 1,672       | 2    |
| 8  | 1001        | 20002      | 1,598       | 3    |
| 9  | 1001        | 20004      | 150         | 4    |
| 10 | 1001        | 20003      | 50          | 5    |
| 11 | 3000        | 20002      | 7,322.4     | 1    |
| 12 | 3000        | 20006      | 2,289.6     | 2    |

|    | CUSTOMER_ID | PRODUCT_ID | SUM(AMOUNT) | RANK |
|----|-------------|------------|-------------|------|
| 13 | 3000        | 20007      | 1,596       | 3    |
| 14 | 3000        | 20003      | 279         | 4    |
| 15 | 3000        | 20004      | 171         | 5    |
| 16 | 3001        | 20008      | 3,048       | 1    |
| 17 | 3001        | 20007      | 1,064       | 2    |
| 18 | 3001        | 20002      | 1,017       | 3    |
| 19 | 3001        | 20004      | 95          | 4    |
| 20 | 3001        | 20003      | 31          | 5    |

In this example, the data is sorted as follows:

- Select rows from the SALES\_DATA table.
- Partition data by CUSTOMER\_ID.
- When the CUSTOMER\_ID value changes, reset the RANK calculation.
- Within each partition, calculate the rank of each month based on the SUM(AMOUNT).
- Within each partition, order the rows by SUM(AMOUNT), in descending order.
- List the CUSTOMER\_ID, revenue, and RANK.
- Order the rows by CUSTOMER\_ID and RANK.

## Window Framing

Window framing defines the beginning and the ending of the window relative to the current row.

This OLAP function is computed with respect to the contents of a moving frame rather than the complete contents of the input data. Depending on its definition, a partition has a start row and an end row, and the window frame slides from the starting point to the end of the partition.

Window framing can be used for all windows functions except ranking aggregates.

## Defining a Window Frame

There are two ways to define a window frame:

- By rows: specify a number of rows before and/or after the current row.
- By range: use data values before and/or after the current row.

A window frame uses the BETWEEN keyword to specify the beginning and the end of the frame.

The following is an example of the syntax to use when defining a window frame by rows:

```
ROWS BETWEEN UNBOUNDED PRECEDING and CURRENT ROW
```

The following is an example of the syntax to use when defining a window frame by range:

```
RANGE BETWEEN UNBOUNDED PRECEDING and CURRENT ROW
```

The row clause refers to the specific number of rows to be included in the window frame. The range clause refers to the data values of the column in the ORDER BY clause of the window. In both cases, specific values or keywords may be used.

### Range and Row Extension

Ranges and rows can extend between the following:



- Current row, meaning the current row in the query's result set
- Unbounded preceding, meaning all rows back to the beginning of the partition
- Unbounded following, meaning all rows forward to the end of the partition
- N preceding, where n is the number of rows or range of values before the current row
- N following, where n is the number of rows or range of values after the current row

### Default Window Frame

If the framing clause is omitted, the default window frame is as follows:



- When the window specification includes an ORDER BY clause, the rows are between unbounded preceding and current row.
- When the window specification does not include an ORDER BY clause, the rows are between unbounded preceding and unbounded following.
- If the window frame extent specifies only one of these two values, the other value defaults to current row.

The following is an example of window framing:



Table 12: Window Framing Example

| Column Value | Rows                | Range               |
|--------------|---------------------|---------------------|
| 10           | Unbounded Preceding | Unbounded Preceding |
| 20           | 1 Preceding         | 10 Preceding        |
| 30           | CURRENT ROW         |                     |
| 40           | 1 Following         | 10 Following        |
| 50           | Unbounded Following | Unbounded Following |

The table summarizes the result returned for each framing clause:



Table 13: Window Framing Example Result

| Framing Clause | Result                              |
|----------------|-------------------------------------|
| Rows between   | unbounded preceding and current row |
|                | 1 preceding and current row         |
|                | 1 preceding and 1 following         |

| Framing Clause |                                             | Result         |
|----------------|---------------------------------------------|----------------|
|                | unbounded preceding and unbounded following | All rows       |
|                | 1 preceding and 1 preceding                 | 20             |
| Range between  | unbounded preceding and current row         | 10, 20, and 30 |
|                | 1 preceding and current row                 | 30             |
|                | 10 preceding and current row                | 20 and 30      |
|                | 10 preceding and 10 following               | 20, 30, and 40 |
|                | current row and current row                 | 30             |

### Cumulative Total Example

The following is an example of the syntax to use to return a cumulative total:



```
SELECT product_id, amount, SUM(amount)
   OVER (ORDER BY product_id
         ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
      AS "CUMULATIVE TOTAL"
  FROM TRAINING.SALES_DATA
 ORDER BY product_id;
```

This query returns the following result:



Table 14: Cumulative Total Example Result

|    | PRODUCT_ID | AMOUNT  | CUMULATIVE TOTAL |
|----|------------|---------|------------------|
| 1  | 20002      | 1,017   | 1,017            |
| 2  | 20002      | 7,322.4 | 8,339.4          |
| 3  | 20002      | 1,598   | 9,937.4          |
| 4  | 20002      | 2,876.4 | 12,813.8         |
| 5  | 20003      | 31      | 12,844.8         |
| 6  | 20003      | 50      | 12,894.8         |
| 7  | 20003      | 112.5   | 13,007.3         |
| 8  | 20003      | 279     | 13,286.3         |
| 9  | 20004      | 171     | 13,769.8         |
| 10 | 20004      | 150     | 13,598.8         |
| 11 | 20004      | 67.5    | 13,448.8         |
| 12 | 20004      | 95      | 13,381.3         |
| 13 | 20006      | 2,289.6 | 16,059.4         |
| 14 | 20006      | 900     | 16,959.4         |
| 15 | 20007      | 1,596   | 18,555.4         |

|    | PRODUCT_ID | AMOUNT | CUMULATIVE TOTAL |
|----|------------|--------|------------------|
| 16 | 20007      | 1,672  | 20,227.4         |
| 17 | 20007      | 564.3  | 20,791.7         |
| 18 | 20007      | 1,064  | 21,855.7         |
| 19 | 20008      | 4,792  | 26,647.7         |
| 20 | 20008      | 3,048  | 29,695.7         |

This example calculates the cumulative total of amounts for all products. For each row, the amount column is summed from the first row (unbounded preceding) to the current row. For example, the cumulative total for row 2 (8,339.4) is calculated by adding the amount for row 2 (1,017) to the amount for row 1 (7,322.4).

### Cumulative Total and Partitioning Example

The following is an example of the syntax to use to return a cumulative total with partitioning:



```
SELECT customer_id, product_id, amount, SUM(amount)
OVER (PARTITION BY customer_id
      ORDER BY product_id
      ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
      AS "CUMULATIVE TOTAL"
FROM TRAINING.SALES_DATA
ORDER BY customer_id, product_id;
```

This query returns the following result:



Table 15: Cumulative Total With Partitioning Example Result

|    | CUSTOMER_ID | PRODUCT_ID | AMOUNT  | CUMULATIVE TOTAL |
|----|-------------|------------|---------|------------------|
| 1  | 1000        | 20002      | 2,876.4 | 2,876.4          |
| 2  | 1000        | 20003      | 112.5   | 2,988.9          |
| 3  | 1000        | 20004      | 67.5    | 3,056.4          |
| 4  | 1000        | 20006      | 900     | 3,956.4          |
| 5  | 1000        | 20007      | 564.3   | 4,520.7          |
| 6  | 1001        | 20002      | 1,598   | 1,598            |
| 7  | 1001        | 20003      | 50      | 1,648            |
| 8  | 1001        | 20004      | 150     | 1,798            |
| 9  | 1001        | 20007      | 1,672   | 3,470            |
| 10 | 1001        | 20008      | 4,792   | 8,262            |
| 11 | 3000        | 20002      | 7,322.4 | 7,322.4          |
| 12 | 3000        | 20003      | 279     | 7,601.4          |
| 13 | 3000        | 20004      | 171     | 7,772.4          |
| 14 | 3000        | 20006      | 2,289.6 | 10,062           |

|    | CUSTOMER_ID | PRODUCT_ID | AMOUNT | CUMULATIVE TOTAL |
|----|-------------|------------|--------|------------------|
| 15 | 3000        | 20007      | 1,596  | 11,658           |
| 16 | 3001        | 20002      | 1,017  | 1,017            |
| 17 | 3001        | 20003      | 31     | 1,048            |
| 18 | 3001        | 20004      | 95     | 1,143            |
| 19 | 3001        | 20007      | 1,064  | 2,207            |
| 20 | 3001        | 20008      | 3,048  | 5,255            |

This example calculates the cumulative total for the first three months on a customer-by-customer basis. When the customer ID value increments, the cumulative total is reset.

### Moving Average Example

The following is an example of the syntax to use to return a moving average:



```
SELECT order_id, amount,
       ROUND(AVG(amount) OVER (ORDER BY order_id
                                ROWS BETWEEN 2 PRECEDING AND CURRENT ROW))
          AS "MOVING AVG"
FROM TRAINING.BII_ORDERS
ORDER BY order_id;
```

This query returns the following result:



Table 16: Moving Average Example Result

|    | ORDER_ID | AMOUNT  | MOVING AVG |
|----|----------|---------|------------|
| 1  | 29,401   | 2,452   | 2,452      |
| 2  | 29,402   | 3,372.7 | 2,912      |
| 3  | 29,403   | 7,266   | 4,364      |
| 4  | 29,404   | 1,135   | 3,925      |
| 5  | 29,405   | 327     | 2,909      |
| 6  | 29,406   | 3,539   | 1,667      |
| 7  | 29,407   | 2,078   | 1,981      |
| 8  | 29,408   | 1,285   | 2,301      |
| 9  | 29,409   | 2,668   | 2,010      |
| 10 | 29,410   | 3,954   | 2,636      |
| 11 | 29,411   | 4,880   | 3,834      |
| 12 | 29,412   | 2,612   | 3,815      |
| 13 | 29,413   | 6,712   | 4,735      |
| 14 | 29,414   | 7,033   | 5,452      |

|    | ORDER_ID | AMOUNT | MOVING AVG |
|----|----------|--------|------------|
| 15 | 29,415   | 1,344  | 5,030      |

This example calculates the moving average of the amounts for the previous two orders and the current order. This is a three-order moving average.

For rows 1 and 2, there are not yet three orders to average together. In row 2, the average of row 1 and 2 is calculated. It is only from row 3 on that the average over three rows is calculated.



#### Note:

The values in the MOVING AVG column have been rounded up to avoid unsightly decimal places.

### Difference Example

The following is an example of the syntax to use to return a difference:



```
SELECT stock, period, stock_price,
       stock_price - sum(stock_price)
    OVER (ORDER BY period
          ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING))
      AS "DIFFERENCE"
  FROM TRAINING STOCK_PRICES
 WHERE stock = 'SAP'
 ORDER BY stock, period;
```

This query returns the following result:



Table 17: Difference Example Result

|    | STOCK | PERIOD  | STOCK_PRICE | DIFFERENCE |
|----|-------|---------|-------------|------------|
| 1  | SAP   | 1/2017  | 92.52       | ?          |
| 2  | SAP   | 2/2017  | 94.64       | 2.12       |
| 3  | SAP   | 3/2017  | 98.17       | 3.53       |
| 4  | SAP   | 4/2017  | 100.18      | 2.01       |
| 5  | SAP   | 5/2017  | 108.60      | 8.42       |
| 6  | SAP   | 6/2017  | 104.67      | -3.93      |
| 7  | SAP   | 7/2017  | 107.36      | 2.69       |
| 8  | SAP   | 8/2017  | 105.34      | -2.02      |
| 9  | SAP   | 9/2017  | 109.65      | 4.31       |
| 10 | SAP   | 10/2017 | 114.91      | 5.26       |
| 11 | SAP   | 11/2017 | 111.17      | -3.74      |
| 12 | SAP   | 12/2017 | 112.58      | 1.41       |

This example displays the current stock price and the difference between the current price and the previous price.

The difference is calculated by subtracting the stock price of the previous row from the stock price of the current row. A positive value indicates an increase and a negative value indicates a decrease.

In row 1, as there is no previous row value to subtract, a null value (?) is displayed in the DIFFERENCE column.



### LESSON SUMMARY

You should now be able to:

- Introduce OLAP analytic features
- Use SQL Group By features
- Use window framing in SQL



## Learning Assessment

1. What is the difference between GROUP BY CUBE and GROUP BY ROLLUP?

*Choose the correct answer.*

- A GROUP BY CUBE is not part of the ANSI SQL-99 standard.
- B GROUP BY CUBE aggregates in every dimension.
- C GROUP BY ROLLUP cannot be used in a SELECT statement.

2. In a SELECT with an OVER() clause, which of the following statements are true?

*Choose the correct answers.*

- A PARTITIONED BY specifies the order of the rows in the result set.
- B ORDER BY specifies the order of the rows in the result set.
- C An OVER clause with ORDER BY cannot contain a PARTITIONED BY.
- D The SELECT statement cannot return NULL values.
- E The PARTITION BY clause is optional.

3. If the current row has a value of 5000 and the three preceding rows have a value (in order) of 1000, 1500, and 3000, which rows would be included in ROWS BETWEEN 2 PRECEDING AND 1 PRECEDING?

*Choose the correct answer.*

- A 1500
- B 1000 and 1500
- C 1500 and 3000
- D 1000, 1500, 3000, and 5000

4. If the current row has a value of 5000 and the three preceding rows have a value (in order) of 1000, 1500, and 3000, which rows would be included in RANGE BETWEEN 3500 PRECEDING AND CURRENT ROW?

*Choose the correct answer.*

- A 1000, 1500, and 3000
- B 1000, 1500, 3000, and 5000
- C 1500, 3000, and 5000
- D 3000 and 5000
- E 5000

## Learning Assessment - Answers

1. What is the difference between GROUP BY CUBE and GROUP BY ROLLUP?

*Choose the correct answer.*

- A GROUP BY CUBE is not part of the ANSI SQL-99 standard.
- B GROUP BY CUBE aggregates in every dimension.
- C GROUP BY ROLLUP cannot be used in a SELECT statement.

Correct! GROUP BY CUBE aggregates in every dimension.

2. In a SELECT with an OVER() clause, which of the following statements are true?

*Choose the correct answers.*

- A PARTITIONED BY specifies the order of the rows in the result set.
- B ORDER BY specifies the order of the rows in the result set.
- C An OVER clause with ORDER BY cannot contain a PARTITIONED BY.
- D The SELECT statement cannot return NULL values.
- E The PARTITION BY clause is optional.

Correct! ORDER BY specifies the order of the rows in the result set and the PARTITION BY clause is optional.

3. If the current row has a value of 5000 and the three preceding rows have a value (in order) of 1000, 1500, and 3000, which rows would be included in ROWS BETWEEN 2 PRECEDING AND 1 PRECEDING?

*Choose the correct answer.*

- A 1500
- B 1000 and 1500
- C 1500 and 3000
- D 1000, 1500, 3000, and 5000

Correct! The rows would be 1500 and 3000.

4. If the current row has a value of 5000 and the three preceding rows have a value (in order) of 1000, 1500, and 3000, which rows would be included in RANGE BETWEEN 3500 PRECEDING AND CURRENT ROW?

*Choose the correct answer.*

- A 1000, 1500, and 3000
- B 1000, 1500, 3000, and 5000
- C 1500, 3000, and 5000
- D 3000 and 5000
- E 5000

Correct! The rows would be 1500, 3000, and 5000.

## Lesson 1

Working with Hierarchies

225

### UNIT OBJECTIVES

- Understand the basics of hierarchies



# Working with Hierarchies



## LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Understand the basics of hierarchies

## SAP HANA SQL Hierarchies



### SAP HANA hierarchies

- Simple SQL interface
- Separates generation from navigation
- Parent-child hierarchies
- Descendants, ancestors, siblings
- Caching for performance optimization
- tightly integrated in SAP HANA operations (import/export, security, backup etc.)

### Benefits

- Easy to use for developers
- Navigate hierarchies based on real-time data

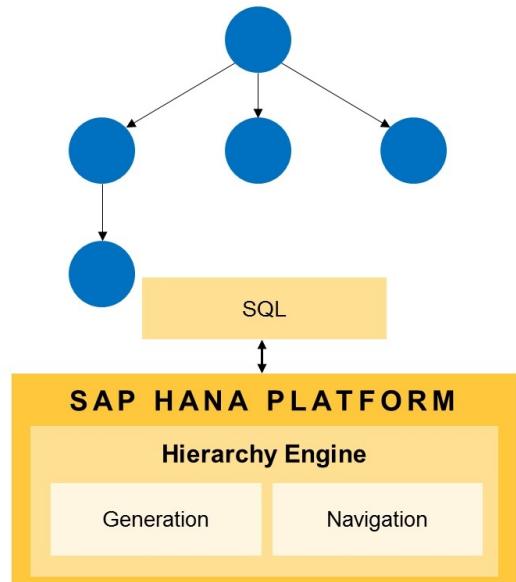
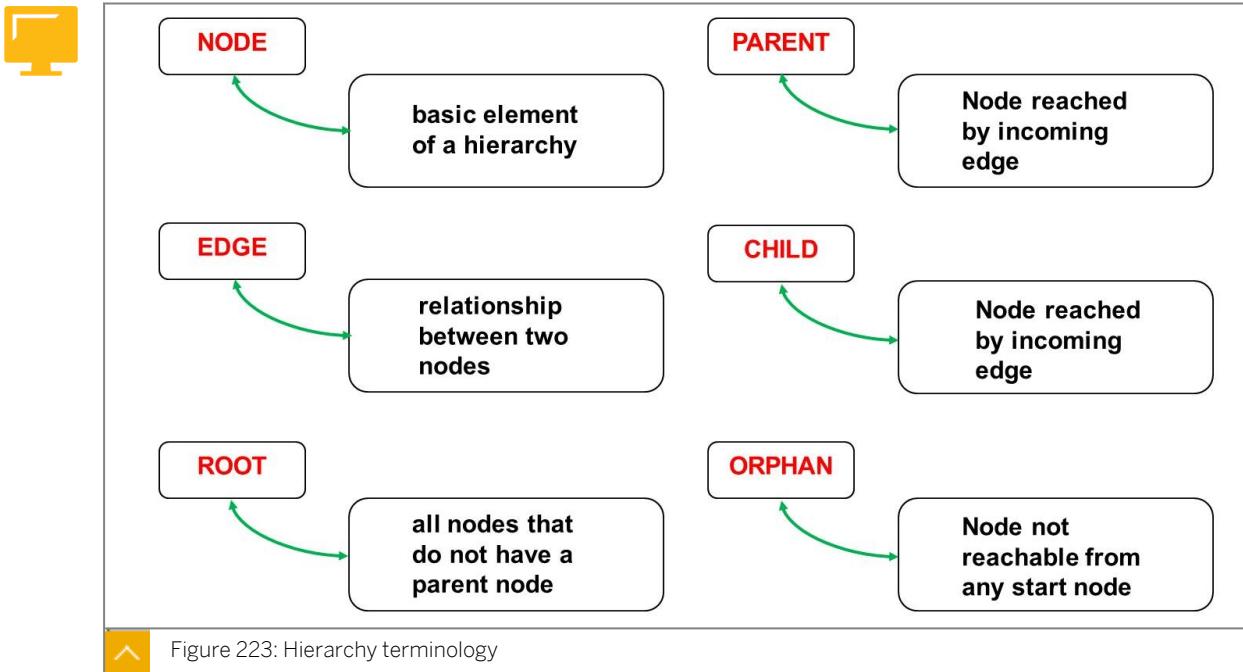


Figure 222: Hierarchies Introduction

Hierarchy functions are an integral part of SAP HANA core functionality. Hierarchies expand the SAP HANA platform with native support for hierarchy attribute calculation and navigation, and allow the execution of typical hierarchy operations directly on relational data, seamlessly integrated into the SQL query.

Hierarchy functions explicitly target the multitude of hierarchical data that already exist in real-world scenarios, company reporting or department structures, hierarchy roles or even plain address data. Hierarchy functions enable users to efficiently query this data in a consistent and structured way, even though the layout of the source data can be very diverse.



Let's work on the terminology used in SQL hierarchies:

- **Hierarchy** — A hierarchy contains a set of standard attributes with fixed types. The topology of the generated hierarchy is strictly a tree or forest where each result row corresponds to a single graph node and edge. The computed index enables the navigation functions to traverse the hierarchy tree efficiently. Hierarchy is the result of the hierarchy generator function.
- **Node** — The basic element of a hierarchy. Each node is defined as a row of a hierarchy generator function result. The primary identifier of a node is its preorder rank provided by the HIERARCHY\_RANK attribute.
- **Edge** — The basic relationship between two nodes in a hierarchy. Each node has either 0 (root) or 1 incoming edge. In addition, a node may have 0 (leaf) or more (branch) outgoing edges. Since a hierarchy is always a strict tree, edges are not independent entities like in generic graphs.
- **Parent** — A node reached by an incoming edge is called a parent node of that node. A node has either one or no parent node.
- **Child** — A node reached by an outgoing edge is called a child node of that node.
- **Root** — Root nodes are all nodes that do not have a parent node.
- **Orphan** — Orphaned nodes are all nodes of a hierarchy that are not reachable from a set of user defined start nodes. In a recursive parent-child hierarchy, two sorts of orphans exist. On the one hand, there are records that do not match the START condition and does not have a parent. On the other hand, there are orphaned islands made up of nodes, which build a cycle that is not connected to any root.



**The type of hierarchy that you will create depend on the structure of the source data**

**Parent Child hierarchy requires separate columns for parents and children**

- Parent and Child usually are of the same data type

**Level hierarchy requires each node in a separate column**

- Possibly different data types can be combined in a hierarchy

**Parent Child Hierarchy source data:**

| Parent  | Child   |
|---------|---------|
| DACH    | Germany |
| Germany | Bayern  |
| Germany | Hamburg |
| EMEA    | Ireland |
| Ireland | Kerry   |

**Level Hierarchy source data:**

| Region | Child   | State   |
|--------|---------|---------|
| DACH   | Germany | Bayern  |
| DACH   | Germany | Hamburg |
| EMEA   | Ireland | Kerry   |

**Note:**

The hierarchies can be time-dependent, where the structure will contain additional columns VALID\_FROM\_DATE and VALID\_TO\_DATE with unique couple of values

Figure 224: Types of Hierarchy

The most common types are parent-child hierarchies and leveled hierarchies.

In parent-child hierarchies, each record references a parent record, defining the hierarchical structure, like in an organization, where each employee record references another person as manager.

In leveled hierarchies, each record contains complete individual path information through the hierarchy. A common example is address data, where each record typically consists of country, state, city, street, and street number data items, which may also be interpreted as a geographical hierarchy.



### Three types of Hierarchy Functions

#### Hierarchy Generator functions

- Generates hierarchy using source table data as input

#### Hierarchy Navigation functions

- Returns specific set of nodes, or node aggregations

#### Hierarchy Scalar functions

- Concatenates multicolumn tuple-like node identifiers into single scalar values

Figure 225: Hierarchy Functions

Hierarchy functions are an integral part of SAP HANA core functionality. Hierarchies expand the SAP HANA platform with native support for hierarchy attribute calculation and navigation, and allow the execution of typical hierarchy operations directly on relational data, seamlessly integrated into the SQL query.



### Public SQL interface for hierarchies

- Parent-child hierarchy generation
- Navigation exposed as table functions
- Massive query acceleration via optional caching

```
WITH "HIER" AS (
    SELECT * FROM HIERARCHY (SOURCE (SELECT
        "NODE_ID", "PARENT_ID", "NAME" FROM "T_GEO"
        ORDER BY "HIERARCHY_RANK")
)
SELECT * FROM HIERARCHY_DESCENDANTS (
    SOURCE "HIER"
    START WHERE "NODE_ID" = 'OC'
);
```

| HIERARCHY_RANK | HIERARCHY_TREE_SIZE | HIERARCHY_PARENT_RANK | HIERARCHY_LEVEL | HIERARCHY_IS_CYCLE | HIERARCHY_IS_ORPHAN | NODE_ID | PARENT_ID | NAME        | HIERARCHY_DISTANCE | START_RANK |
|----------------|---------------------|-----------------------|-----------------|--------------------|---------------------|---------|-----------|-------------|--------------------|------------|
| 7              | 4                   | 1                     | 2               | 0                  | 0                   | 0 OC    | WO        | Oceania     | 0                  | 7          |
| 8              | 2                   | 7                     | 3               | 0                  | 0                   | 0 NZ    | OC        | New Zealand | 1                  | 7          |
| 9              | 1                   | 8                     | 4               | 0                  | 0                   | 0 CO    | NZ        | Coromandel  | 2                  | 7          |
| 10             | 1                   | 7                     | 3               | 0                  | 0                   | 0 AUS   | OC        | Australia   | 1                  | 7          |

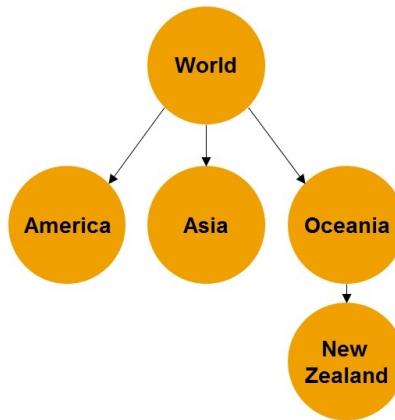


Figure 226: Public SQL Interface

Hierarchy functions explicitly target the multitude of hierarchical data that already exist in real-world scenarios, company reporting or department structures, hierarchy roles or even plain address data. Hierarchy functions enable users to efficiently query this data in a consistent and structured way, even though the layout of the source data can be very diverse.



### Generator functions

#### HIERARCHY Generator

```
HIERARCHY (
<hierarchy_genfunc_source_spec>
[<hierarchy_genfunc_start_cond>]
<hierarchy_genfunc_order_spec>
[<hierarchy_genfunc_depth_spec>]
[<hierarchy_genfunc_multiparent_spec>]
[<hierarchy_genfunc_orphan_spec>]
[<hierarchy_genfunc_cycle_spec>]
[<hierarchy_genfunc_cache_spec>]
[<hierarchy_genfunc_load_spec>]
)
```

#### HIERARCHY\_LEVELED Generator

```
HIERARCHY_LEVELED (
<hierarchy_genfunc_source_spec>
[<hierarchy_genfunc_level_spec>]
<hierarchy_genfunc_order_spec>
[<hierarchy_genfunc_cache_spec>]
) Syntax
```

#### HIERARCHY\_SPANTREE Generator

```
HIERARCHY_SPANTREE (
<hierarchy_genfunc_source_spec>
[<hierarchy_genfunc_start_cond>]
<hierarchy_genfunc_order_spec>
[<hierarchy_genfunc_depth_spec>]
[<hierarchy_genfunc_multiparent_spec>]
[<hierarchy_genfunc_cache_spec>]
[<hierarchy_genfunc_load_spec>])
```

#### HIERARCHY\_TEMPORAL Generator

```
HIERARCHY_TEMPORAL (
<hierarchy_genfunc_source_spec>
[ <hierarchy_genfunc_start_cond> ]
<hierarchy_genfunc_order_spec>
<hierarchy_genfunc_validity_spec>
[ <hierarchy_genfunc_depth_spec> ]
[<hierarchy_genfunc_multiparent_spec>]
[<hierarchy_genfunc_cycle_spec>]
[ <hierarchy_genfunc_cache_spec> ]
[ <hierarchy_genfunc_load_spec> ] )
```

Figure 227: Generator Functions



## Generator functions

- HIERARCHY Generator

**Source**

|    | PARENT_ID | NODE_ID | ORD | AMOUNT |
|----|-----------|---------|-----|--------|
| 1  | NULL      | A1      | 1   | 1      |
| 2  | A1        | B1      | 1   | 2      |
| 3  | A1        | B2      | 2   | 4      |
| 4  | B1        | C1      | 1   | 1      |
| 5  | B1        | C2      | 2   | 3      |
| 6  | B2        | C3      | 3   | 1      |
| 7  | B2        | C4      | 4   | 2      |
| 8  | C3        | D1      | 1   | 2      |
| 9  | C3        | D2      | 2   | 3      |
| 10 | C4        | D3      | 3   | 1      |

**SELECT**

```

hierarchy_rank AS rank,
hierarchy_tree_size AS tree_size,
hierarchy_parent_rank AS parent_rank,
hierarchy_level AS level,
hierarchy_is_cycle AS is_cycle,
hierarchy_is_orphan AS is_orphan,
parent_id,
node_id

```

**FROM**

```

HIERARCHY (
SOURCE "STUDENT01"."SRC_TABLE_PARENT_CHILD"
SIBLING
ORDER BY ord CACHE FORCE )
ORDER BY hierarchy_rank
;
```

**Output: View / Persisted**

| RANK | TREE_SIZE | PARENT_RANK | LEVEL | IS_CYCLE | IS_ORPHAN | PARENT_ID | NODE_ID |
|------|-----------|-------------|-------|----------|-----------|-----------|---------|
| 1    | 10        | 0           | 1     | 0        | 0         | NULL      | A1      |
| 2    | 3         | 1           | 2     | 0        | 0         | A1        | B1      |
| 3    | 1         | 2           | 3     | 0        | 0         | B1        | C1      |
| 4    | 1         | 2           | 3     | 0        | 0         | B1        | C2      |
| 5    | 6         | 1           | 2     | 0        | 0         | A1        | B2      |
| 6    | 3         | 5           | 3     | 0        | 0         | B2        | C3      |
| 7    | 1         | 6           | 4     | 0        | 0         | C3        | D1      |
| 8    | 1         | 6           | 4     | 0        | 0         | C3        | D2      |
| 9    | 2         | 5           | 3     | 0        | 0         | B2        | C4      |
| 10   | 1         | 9           | 4     | 0        | 0         | C4        | D3      |

Figure 228: Hierarchy Generator



## Generator functions

- HIERARCHY\_LEVELED Generator

**Source**

| LEVEL_1 | LEVEL_2 | ATTR_2 | LEVEL_3 | LEVEL_4 | ATTR_4 |
|---------|---------|--------|---------|---------|--------|
| L1      | M1      | 1      | N1      | O1      | 10     |
| L1      | M1      | 2      | NULL    | O2      | 20     |
| L1      | M2      | 3      | NULL    | NULL    | NULL   |
| L1      | NULL    | NULL   | N2      | O3      | 30     |

**HIERARCHY\_LEVELED  
Generator function**

**Output: View / Persisted**

| RANK | TREE_SIZE | PARENT_RANK | LEVEL | PARENT_ID | NODE_ID | LEVEL_NAME | L1 | L2   | A2   | L3   | L4   | A4   |
|------|-----------|-------------|-------|-----------|---------|------------|----|------|------|------|------|------|
| 1    | 14        | 0           | 1     | NULL      | L1      | LEVEL_1    | L1 | NULL | NULL | NULL | NULL | NULL |
| 2    | 8         | 1           | 2     | L1        | M1      | LEVEL_2    | L1 | M1   | NULL | NULL | NULL | NULL |
| 3    | 4         | 2           | 3     | M1        | 1       | ATTR_2     | L1 | M1   | 1    | NULL | NULL | NULL |
| 4    | 3         | 3           | 4     | 1         | N1      | LEVEL_3    | L1 | M1   | 1    | N1   | NULL | NULL |
| 5    | 2         | 4           | 5     | N1        | O1      | LEVEL_4    | L1 | M1   | 1    | O1   | NULL | NULL |
| 6    | 1         | 5           | 6     | O1        | 10      | ATTR_4     | L1 | M1   | 1    | N1   | O1   | 10   |
| 7    | 3         | 2           | 3     | M1        | 2       | ATTR_2     | L1 | M1   | 2    | NULL | NULL | NULL |
| 8    | 2         | 7           | 5     | 2         | O2      | LEVEL_4    | L1 | M1   | 2    | NULL | O2   | NULL |
| 9    | 1         | 8           | 6     | O2        | 20      | ATTR_4     | L1 | M1   | 2    | NULL | O2   | 20   |
| 10   | 2         | 1           | 2     | L1        | M2      | LEVEL_2    | L1 | M2   | NULL | NULL | NULL | NULL |
| 11   | 1         | 10          | 3     | M2        | 3       | ATTR_2     | L1 | M2   | 3    | NULL | NULL | NULL |
| 12   | 3         | 1           | 4     | L1        | N2      | LEVEL_3    | L1 | NULL | NULL | N2   | NULL | NULL |
| 13   | 2         | 12          | 5     | N2        | O3      | LEVEL_4    | L1 | NULL | NULL | N2   | O3   | NULL |
| 14   | 1         | 13          | 6     | O3        | 30      | ATTR_4     | L1 | NULL | NULL | N2   | O3   | 30   |

Figure 229: Hierarchy Leveled Generator

Specialized hierarchy generator functions translate the diverse relational source data into a generic and normalized tabular format that, for the sake of brevity, is termed as hierarchy. The structure of such a hierarchy is always the same, regardless of the original format of the source data, and consists of an ordered list of its nodes. The nodes themselves are represented by a minimal set of orthogonal hierarchy topology attributes plus a projection of the original source attributes.

Many basic properties of the hierarchy and its elements can be determined by directly querying this tabular hierarchy representation using the usual SQL filtering and expression facilities.

More complex calculations on the hierarchy involving a start set of nodes and potentially multiple steps beyond their immediate neighborhood are termed navigations. SAP HANA provides built-in navigation functions that use optimized algorithms on the hierarchy attributes to efficiently compute ancestors, descendants, or siblings of nodes.



**Materialized Temporary table from function**

```
CREATE LOCAL TEMPORARY COLUMN TABLE <TEMP_TABLE_NAME> AS ( SELECT
    *
    FROM HIERARCHY ( SOURCE <SOURCE_TABLE_NAME> SIBLING
        ORDER BY ord )
    ORDER BY hierarchy_rank )
```

**View from function**

```
CREATE VIEW <VIEW_NAME> AS SELECT
    *
    FROM HIERARCHY ( SOURCE <SOURCE_TABLE_NAME> SIBLING
        ORDER BY ord CACHE FORCE )
    ORDER BY hierarchy_rank ;
```

Figure 230: Using Temporary Tables with Hierarchies

Very often, multiple queries might be executed using the same hierarchy generator function result set. Therefore, it is recommended to either materialize the hierarchy generator function output into a temporary table or define a view over it.

Materializing the hierarchy generator function output into a temporary table guarantees consistent and stable navigation results for the entire lifecycle of the temporary table and eliminates the hierarchy generation cost for subsequent queries: At the same time, materializing a hierarchy into a temporary or persistent table cuts all connections between the hierarchy and its sources.

If a query should return the most current state of the data, it is recommended to rather create a view over a hierarchy generator function, which guarantees that hierarchy navigation results always correctly reflect the current transactional view of the source data. If a hierarchy source is fully deterministic (that means that multiple query executions over the same data set always return the same result), it is allowed to cache the hierarchy generator function output. Caching avoids the overhead of recalculating the hierarchy if the sources do not change between subsequent navigation and at the same time guarantees a correct transaction result.



## LESSON SUMMARY

You should now be able to:

- Understand the basics of hierarchies

## Learning Assessment

1. What are the three types of hierarchy function?

*Choose the correct answers.*

- A Navigation
- B Generation
- C Scalar
- D Table

2. The results of a hierarchy generator function can be used only once.

*Determine whether this statement is true or false.*

- True
- False

## Learning Assessment - Answers

1. What are the three types of hierarchy function?

*Choose the correct answers.*

- A Navigation
- B Generation
- C Scalar
- D Table

Correct! Generation, navigation, and scalar are the three types of hierarchy function. Table is not a hierarchy function. See HA150 Unit 8 for more details.

2. The results of a hierarchy generator function can be used only once.

*Determine whether this statement is true or false.*

- True
- False

Correct! You should use a temporary table or a view to materialize the results so they can be reused. See HA150 Unit 8 for more details.

# UNIT 9

# Troubleshooting and Best Practices

## Lesson 1

Understanding the Tools and Views to Analyze and Optimize SQL

235

## Lesson 2

Applying Tools and Views to Optimize SQL Performance

243

## Lesson 3

Further Tools for Troubleshooting

255

## UNIT OBJECTIVES

- Understand the tools and views for analyzing and optimizing SQL in SAP HANA
- Apply Views and Tools to optimize SQL Performance
- Usage of further tools for troubleshooting SQLScript



# Understanding the Tools and Views to Analyze and Optimize SQL



## LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Understand the tools and views for analyzing and optimizing SQL in SAP HANA

## Suboptimal SQL Symptoms



- Slow generation of reports in the backend
- SAP HANA System not responding anymore while applications are running
- Error messages: Out of memory situations (OOM) in the traces
- Lack of memory according to SAP HANA Cockpit and Monitoring tools

Figure 231: Suboptimal SQL - Potential Symptoms

When facing performance problems on a SAP HANA system, it is not always right away clear what the reason is. It might be caused by code of an application or by other reasons. For instance an unload of data from the cache in SAP HANA can be due to contention of parallel very long lasting SQL. It also could be the result of a memory that size does not allow processing appropriately - maybe due to heavy data growth over time. So when problems on SAP HANA occur, the first question to answer is: what is the root cause for it? If the answer is at least partly due to supposedly suboptimal SQL, then the following described methods can be used to analyze in detail what is or are the reasons for it and how to solve it in the next step.

## Proactive Activities



- Revise the data model for SAP HANA (in-memory, column based)
- Stay trained on using in-memory methods and functions in SQL
- Stay current with new SAP HANA releases (new SQL Features)
- Create easy to understand and transferrable problem / solution documentation
- Provide of access to systems for backup teams (absences)
- Workload Classes for the most urgent and important SQL
- Create, use and maintain a suitable naming convention for SQL objects
- Use a meaningful error messaging concept to find suboptimal SQL easier



Figure 232: Proactive Activities for Optimal SQL

The best way to prevent creating suboptimal SQL is the training on how to create optimal SQL right from the start. Attending trainings before creating optimal SQL is not always feasible or knowledge on how to do it the best way may have become inactive. Nevertheless it appears to be a good investment in knowledge if you think about the following costs of making up SQL errors in the code (detect, evaluate, improve, test and bring to production again ...). On top experience reveals that not seldom suboptimal code shows up in inconvenient moments of projects.

The data model is a core and important factor in creating optimal SQL. If the data model does not allow efficient code, optimization probably will fail or at least only bring less results. Also the data modeler knowledge over time should be kept up-to-date to learn new features. Those new features may allow to solve a SQL performance problem in the code that could not be solved that way before. The Release Bulletins that come up with every SPS (Support Package Stack) that may offer new features of SQL. New hints can be mentioned in this context also.

In-memory column based technology differs from the legacy RDBMS. Context switches between SAP HANA engines can cause poor performance but can be fixed. In the best case, context switches are not even allowed through proactive testing and monitoring. Also, organizational factors form a proactive approach to prevent suboptimal SQL from being written. Creating a list of best practices and how to access the system is another proactive measure. Also, assigning Workload classes with higher or lower priority can push optimal execution of SQL in SAP HANA.

Last but not least, naming conventions for stored procedures, tables, and so on, connected with custom created error messaging contribute to finding and improving SQL with poor performance.

## Role Responsibilities



- Developer has no privilege to grant privileges on system configuration
- Administrator is not permitted to enter a HDI Container deploying optimal code
- Data Modeler is not permitted to recompile a Stored Procedure executing the code in sub optimal way
- Problem Analyzer Role would be beneficial to have – e.g. setup in role rotation

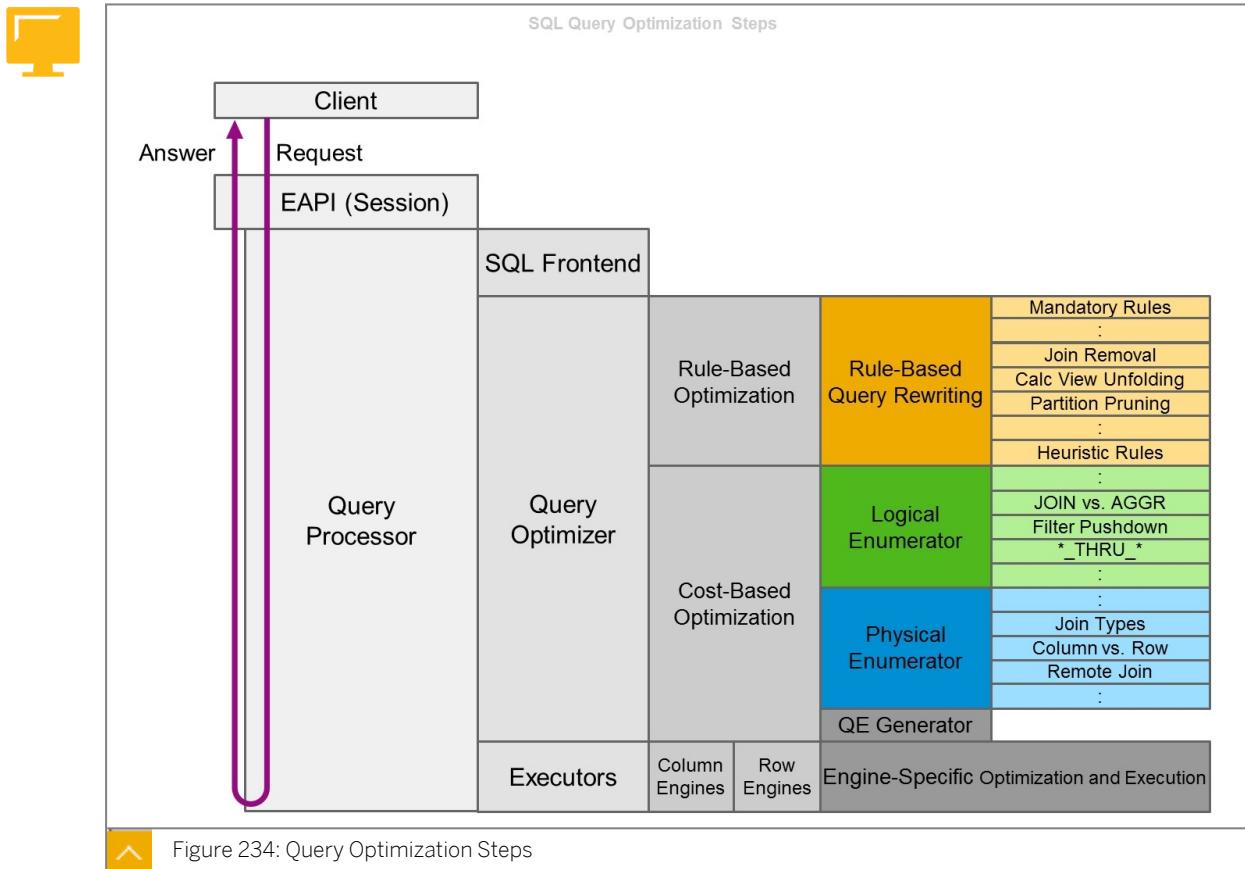
Figure 233: Role Responsibilities for Suboptimal SQL

In real world projects, the question about who is responsible for taking care of the performance issues sometimes comes up, especially if it is SQL code that was handed over or inherited from third parties. This first barrier is a non-technical but an organizational issue that needs to be addressed. Roles like Administrator or Data modeler or the Developer are among the possible roles that are potentially capable of solving suboptimal SQL challenges. Someone has to take over the responsibility to analyze the problem and it is not clear which role will finally work on finding the solution for it. This could be time consuming and it is worthwhile to assign clearly who does what and when.

In case of obvious inefficient system setups, the Administrator is usually responsible. If the execution of application code clearly causes slow SQL execution, the responsible role will be the author of this code and it is often the Developer who has to do some optimizing changes on the SQL Code if possible. If the data model happens to be the problem, e.g. if migration from other database manufacturers towards SAP HANA took place, then the Data modeler appears to be the right person to find a solution and so on.

The role in-between that helps to find out what is happening and whom to assign the responsibility for finding the problem and then creating a solution can be described as the "Problem Analyzer" role. Surely this could be set up to combine with existing roles like Architect, Developer, Administrator or other roles. Part time job rotation role is also an idea too. The role owner of "Problem Analyzer" pre-evaluates the root cause, e.g. poor performing Stored Procedure in Container XYZ and may assign further steps to the appropriate trained person or the author to work on a solution and to transport it back to production.

## Query Optimization Steps

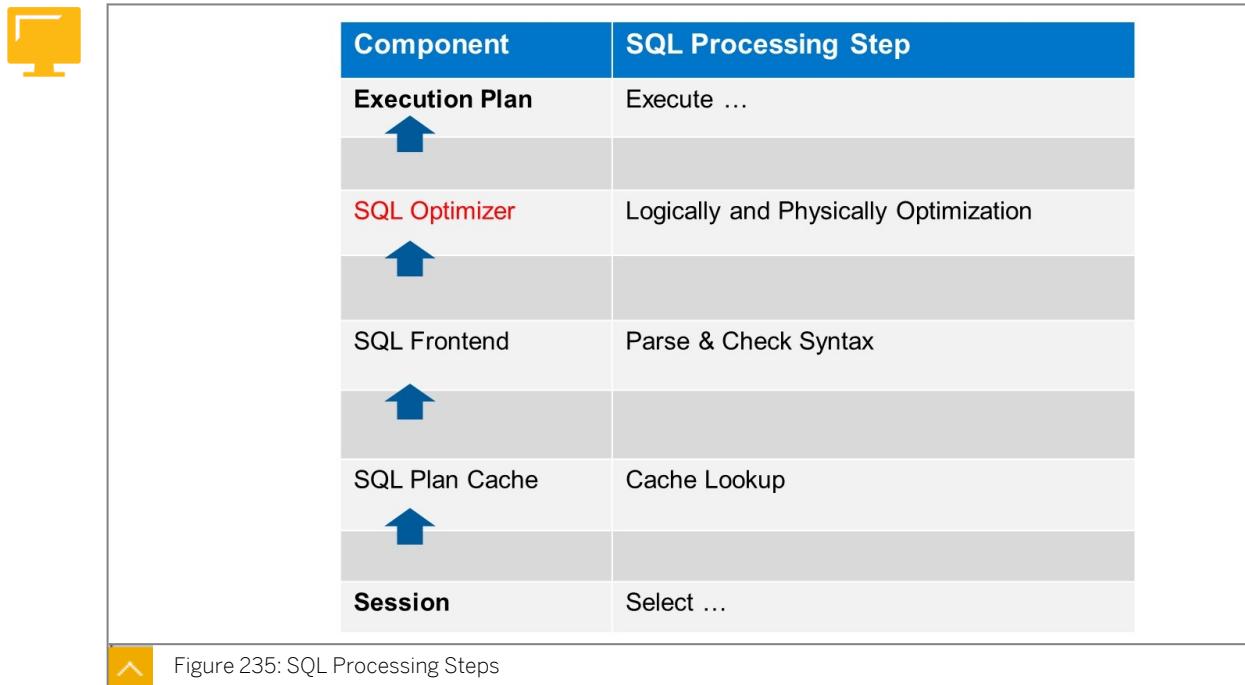


The key to solving a performance issue is to find the root cause of the problem. This is the point in which the compilation process no longer leads to a result that matches performance expectations, though it may run correctly. This could be perceived as a contradiction (correct execution plan but poor performance). For this situation, SAP provides a great variety of tools. To be able to identify the underlying issues, you require a thorough understanding of the query optimization steps, as you see in the figure.

On the way from Request to Answer, the *Query Optimizer* coordinates the *Query Optimizer* until all detail decisions have been made. There is an automatic *Cost-Based Optimization* process within SAP HANA that can be over-ruled by the *Rule-Based Optimizer*.

Executors like the *SAP HANA Extended SQL Executor* (ESX), which is a front-end execution engine execute the SQL queries. The Enumerator phase (Rule Based vs. Cost-Based) ends with the selection of Column Engines, Row Engines or an orchestration of those. Then either a rewriting (Rule-Based) of the execution steps happen or logical or physical enumerators (Cost-based) are applied until the execution plan has been created successfully. This process can be repeated and influenced by rules, e.g. *Filter Pushdown* replacing *Joins* could be a logical change that often appears as a solution. Hints provided by SAP can be used to test and change decisions of the optimizer and finally optimize the execution plan.

## Basic SQL Processing



The basic SQL query undergoes several steps on its way to its execution plan. In order to analyze and finally optimize SQL in SAP HANA, it is necessary to understand firstly the process how SQL is executed on the basis of an execution plan.

An execution plan is the compiled, ready to execute and selected translation of what the code intends to achieve so optimization can focus on this plan. Before this execution plan is created, several other steps have to be undertaken by SAP HANA.

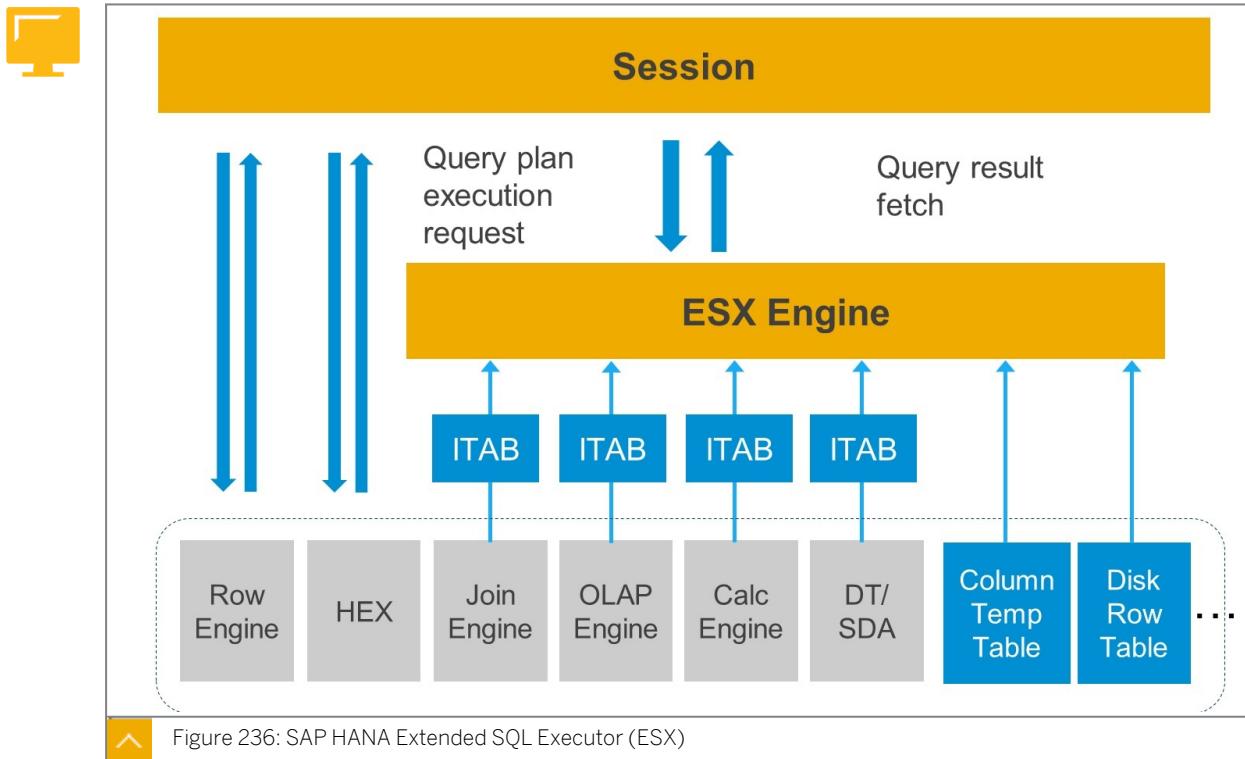
These steps are executed by means of different components and in parallel if possible . After accepting a SQL request within a Session, a syntax check and parsing in context of sufficient object and system privileges take place in SQL Frontend.

Then the Query Optimizer, a very important concept in SQL Execution process, will evaluate the "best" and "most efficient" track for the result set which leads to the execution plan. Logical and physical optimization iterations are undertaken fully automatically, this happens under the condition of cost-based.

In test or development systems, sometimes rule-based execution is used temporarily which makes parallelism either impossible or harder to apply. The regular cost based evaluation triggers the comparison of enumerations and then leads to the best approach for that particular moment, which is represented by the execution plan to be used.

This plan can be reused to save compilation time in the future and cut down redundant operation steps (like syntax checks). The appropriate SAP HANA Engines then are selected and will work on the generation of the result sets with the generated code according to the execution plan.

## SAP HANA Extend SQL Executor



Since SAP HANA 2.0 SPS 02, there are two additionally processing engines in SAP HANA to execute SQL queries. They were introduced to offer even better performance under the precondition not to affect the functionality of SAP HANA.

The new engines are active by default and are considered by the SQL optimizer during query plan generation.

*SQL Executor (ESX)* and

*SAP HANA Execution Engine (HEX)*

The one we want to shortly point out in this Performance and Tuning introduction is the *SAP HANA Extended SQL Executor (ESX)*.

The *SAP HANA Extended SQL Executor (ESX)* is a frontend execution engine that replaces the row engine in part, but not completely. It retrieves database requests at session level and delegates them to lower-level engines like the *Join Engine* and *Calculation Engine*. Communication with other engines is simplified by using ITABs (internal tables) as the common exchange format.

The other engine, the *SAP HANA Execution Engine (HEX)* is a query execution engine that will replace other SAP HANA engines such as the *Join Engine* and *OLAP Engine* in the long run by combining all in one single engine. It connects the SQL layer with the column store by creating an appropriate SQL plan during the prepare phase. Queries that are not supported by HEX or where an execution is not considered beneficial are automatically routed to the legacy engine.

## Unexpected SQL Query Performance Deterioration



### Top reasons:

- Upgrades of SAP HANA Platform with new functions and new default values
- Evolution of the data model – ( Optimal Code may become Suboptimal )
- Memory Size exceeded (no temperature model and thereof size problems)
- Mass processing of data at the same time (“Traffic Jam effect”)

Figure 237: Unexpected SQL Query Performance Deterioration

Optimal SQL code may become suboptimal over time. Therefore, it is possible that applications may show a stagnation or deterioration in performance over time due to changing data metrics, changing data formats or changed size relations between database artifacts through joins associated table evaluations. The cost based optimizer for this reason is flexible and tries to optimize the execution plan as much as possible at creation time. Though there are situations where this is not possible anymore and suddenly or subsequently a down-fall in execution takes place. The reason for this is not only connected to different size metrics of objects like tables but could also be caused by new functionalities of SAP HANA after an upgrade. SAP HANA is a fast growing technology and comprises, in its Support Package Stacks, large sets of new functionalities in each release. If new functionalities, e.g. when new execution engines are introduced, it can have an impact on how the optimizer chooses to create the execution plan. Previously solid and long used execution plans might become suboptimal.

Working through SAP HANA Release Bulletin documents shipped with each new SAP HANA upgrade is highly recommended. The steps are to be undertaken proactively on SAP HANA databases in order not to experience suboptimal code execution. Another advantage of establishing this routine is that new SAP HANA functionalities bring improvements that may have great potential for improvements in other work areas.

Too much growing data can lead to poor SQL performance and to the cache unloading data over time. Concepts and technologies like the temperature model or data tiering can protect from this deterioration and should be taken into account. Additionally applying the flow of data into SAP HANA and out of SAP HANA gives back control over what to do with less important and less urgent data over time. Concepts and technologies like the well-known temperature model or data tiering can protect from ever-growing data that might lead to unloads of data from cache over time and as a consequence to poor SQL performance.

Last but not least, system performance deterioration can also have its origin in external factors like the configuration of the overall system. Configuration (default vs. needed configuration) can be done in a way that performance of SQL execution within SAP HANA is not possible and activities on the Application level would turn out to be ineffective. A systematic solution process is recommended between the SAP HANA system roles and shared objectives.

## Analyzing Tools for SQL



- Explain Plan – Lightweight tool
- SQL Analyzer – Main tool
- Graph Analyzer ex. Visualized Plan (PlanViz)
- SQL Plan Cache
- SQL Traces (Trace, Step Debug, (Time Debug))
- System Views (e.g. Exensive Statement, Plan\_Cache)
- Mini Checks (also using System Views)
- Hints (Rule vs. Cost-based optimizations)



Figure 238: Analyzing Tools for Suboptimal SQL

To find and focus on the suboptimal SQL is the very first step to solve poor performance. For this reason, there is a wide range of analyzing tools provided to the SAP HANA Developers who use SAP HANA. Some of those tools can help the developers to understand the group of objects involved, for instance, within the execution respective call of a stored procedure, or simply to understand the operators being used.

The hierarchy of operators can be visualized by the *Graph Plan* tool. We can see the excluded or included parts within the processing of SQL and find the place where time is consumed to a high degree.

*Explain Plan*, on the other hand, provides an ex-ante possibility within testing scenarios to evaluate how the cost based optimizer within SAP HANA will decide which way to take in order to create the result set. For instance, if an index was ignored by the optimizer, one can test using a hint in the code to force index usage and see if the SQL runs as expected.

*SQL Traces* show what happened and give a granular way of controlling what is intended to be traced. *System Views* or *Mini Checks*, which are integrated in the Statement Library, can be used to evaluate information about suboptimal SQL. In the next lessons, we will learn where to find and how to use these tools in more detail.



## LESSON SUMMARY

You should now be able to:

- Understand the tools and views for analyzing and optimizing SQL in SAP HANA

## Applying Tools and Views to Optimize SQL Performance



### LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Apply Views and Tools to optimize SQL Performance

### SAP HANA System Views for SQL Monitoring



#### SAP HANA comprises 3 types of built-in system views :

##### Monitoring Views

- provide actual SAP HANA runtime data including SQL DML statements
- Very useful for monitoring and troubleshooting SAP HANA performance
- Examples: **M\_SQL\_PLAN\_CACHE, M\_EXPENSIVE\_STATEMENTS**

##### System Views

- Main focus area is Administration and system state
- Examples: **CS\_JOIN\_TABLES, EFFECTIVE\_PRIVILEGES**

##### Embedded Statistics Service Views

- Tables and views from the \_SYS\_STATISTICS schema.
- Examples: **HOST\_CS\_UNLOADS, HOST\_RS\_INDEXES**

Figure 239: SAP HANA System Views for SQL Monitoring

#### 1. Monitoring Views (e.g. M\_SQL\_PLAN\_CACHE, M\_EXPENSIVE\_STATEMENTS)

These views are in Schema 'SYS' and can be identified by the **M\_** at the beginning of the view name. SAP HANA includes a set of useful runtime views called monitoring views. These views are useful for monitoring and troubleshooting SQL performance. They provide actual SAP HANA runtime data, including statistics and status information related to the execution of DML statements. At the time of writing there are 385 Monitoring Views out of 592 Views in Total (as of SPS06) customers. The data in monitoring views is not stored. The View is calculated when you execute a select on it. Nevertheless the result sets can be materialized in tables if needed. Further information can be looked up in the SAP HANA SQL Reference Guide.

#### 2. System Views (Main focus SAP HANA Administration )

System views are stored in the SYS schema. In tenant databases, each database has a SYS schema with system views that contain information focused on the tenant. System views allow you to evaluate information about the system state itself and not the

application code like SQL. The System Privileges CATALOG READ or DATABASE ADMIN are needed to use them.

### 3. Embedded Statistics Service Views

Tables and views are stored in the \_SYS\_STATISTICS schema. It is implemented by a set of tables and SQLScript procedures in the master index server and by the statistics scheduler thread that runs in the master name server. The statistics service is continuously collecting and evaluating information about status, performance, and resource usage from all components of the SAP HANA database.

#### Monitor View M\_SQL\_PLAN\_CACHE

The screenshot shows the SAP HANA Studio interface. On the left, the object browser displays various catalog objects under 'My Catalog (SYSTEM)'. In the center, a results table shows 1000 rows of data. A context menu is open over the table, with the option 'Generate SELECT Statement' highlighted. At the bottom, a table titled 'First 1000 rows' shows the detailed data for the first 8 rows.

|   | HOST        | PORT  | VOLUME_ID | Statement                                                                                                                                                         |
|---|-------------|-------|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1 | wdfibmt7215 | 30003 | 3         | DELETE FROM "com.sap.di.job.JobEntity" WHERE (((TYPE = 'asyncJob') AND ((MODIFIED IS NOT NULL) AND ((MODIFIED + 3600000) < 164613115))                            |
| 2 | wdfibmt7215 | 30003 | 3         | DELETE FROM "com.sap.di.job.JobEntity" WHERE (((TYPE = 'runTask') AND (((STATE = 'CANCELLED') AND ((MODIFIED IS NOT NULL) AND ((MODIFIED + 3600000) < 164613115)) |
| 3 | wdfibmt7215 | 30003 | 3         | DELETE FROM "com.sap.di.job.JobEntity" WHERE (((TYPE = 'runTask') AND (((STATE = 'CANCELLED') AND ((MODIFIED IS NOT NULL) AND ((MODIFIED + 3600000) < 164613115)) |
| 4 | wdfibmt7215 | 30003 | 3         | DELETE FROM "com.sap.di.job.JobEntity" WHERE (((TYPE = 'runTask') AND (((STATE = 'CANCELLED') AND ((MODIFIED IS NOT NULL) AND ((MODIFIED + 3600000) < 164613115)) |
| 5 | wdfibmt7215 | 30003 | 3         | DELETE FROM "com.sap.di.job.JobEntity" WHERE (((TYPE = 'runTask') AND (((STATE = 'CANCELLED') AND ((MODIFIED IS NOT NULL) AND ((MODIFIED + 3600000) < 164613115)) |
| 6 | wdfibmt7215 | 30003 | 3         | select SCHEMA_NAME, ACCESSED_OBJECT_NAMES, HOST_EXECUTED, PORT_EXECUTED, ACCESSED_PARTITION from _SYS_SQL_ANALYZER                                                |
| 7 | wdfibmt7215 | 30003 | 3         | select distinct HOST_EXECUTED, PORT_EXECUTED from _SYS_SQL_ANALYZER.OPERATOR_STATISTICS where STATEMENT                                                           |
| 8 | wdfibmt7215 | 30003 | 3         | Select USED_MEMORY_SIZE, EXECUTED_OUTPUT_RECORD_COUNT from _SYS_SQL_ANALYZER.STATEMENT_STATISTICS where STATEMENT                                                 |

Figure 240: Monitor View: M\_SQL\_PLAN\_CACHE

The M\_SQL\_PLAN\_CACHE view shows statistics for individual execution plans. It shows which part of the execution is dominant. For each plan in the cache, it delivers statistics from execution and technical details such as related objects and object IDs, updated objects, and much more.

M\_SQL\_PLAN\_CACHE shows views that can be reset. This is beneficial to only see results after the point in time when it was reset.

To reset the view, execute the following statement: `ALTER SYSTEM RESET MONITORING VIEW SYS.M_SQL_PLAN_CACHE_RESET;`

In the version of SAP HANA SPS06, this view has 108 columns. When you scroll with the bar at the bottom to the right, you can see more or you can choose to let the tool create the select statement on the view so as to select the information you would like to see.

## Monitor View M\_EXPENSIVE\_STATEMENTS

The screenshot shows the SAP HANA Database Explorer interface. On the left, there is a tree view of database objects under the 'SYS' schema, with 'M\_EXPENSIVE\_STATEMENTS' highlighted. A context menu is open over this object, with 'Open Data' selected. Below the tree view, the main workspace displays a table titled 'M\_EXPENSIVE\_STATEMENTS'. The table has four rows of data, each containing information such as HOST, PORT, CONNECTION\_ID, TRANSACTION\_ID, UPDATE\_TRANSACTION\_ID, STATEMENT\_ID, STATEMENT\_HASH, DB\_USER, and APP\_USER.

|   | HOST        | PORT  | CONNECTION_ID | TRANSACTION_ID | UPDATE_TRANSACTION_ID | STATEMENT_ID     | STATEMENT_HASH                   | DB_USER         | APP_USER  |
|---|-------------|-------|---------------|----------------|-----------------------|------------------|----------------------------------|-----------------|-----------|
| 1 | wdfibmt7215 | 30003 | 300164        | 20             | 483283007             | 1289196178762474 | d6fd6678833f9a2e25e7b53239c50e9a | _SYS_STATISTICS |           |
| 2 | wdfibmt7215 | 30003 | 300160        | 15             | 483283038             | 1289179454689086 | d6fd6678833f9a2e25e7b53239c50e9a | _SYS_STATISTICS |           |
| 3 | wdfibmt7215 | 30003 | 300162        | 18             | 483280460             | 128918798791573  | d6fd6678833f9a2e25e7b53239c50e9a | _SYS_STATISTICS |           |
| 4 | wdfibmt7215 | 30003 | 362774        | 55             | 0                     | 1558106547988095 | 04d85e71b606ccfec5fb4123f43a5af  | STUDENT01       | STUDENT01 |

Figure 241: Monitor View: M\_EXPENSIVE\_STATEMENTS

### Expensive Statements Trace Information

If you have the TRACE ADMIN privilege, then you can view expensive statements trace information in the following ways:

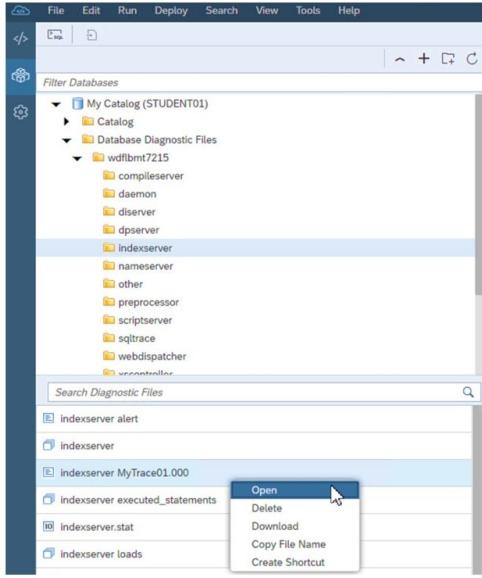
- In the Expensive Statements app of the SAP HANA cockpit
- In the Statement Library in SAP HANA database explorer by searching for Expensive Statements Analysis
- In the M\_EXPENSIVE\_STATEMENTS system view

In the version of SAP HANA SPS06, this view has 43 columns.

Expensive statements are SQL statements with execution times exceeding a configured threshold. So, the user decides "what" is expensive. The thresholds should be selected that way that these thresholds are reached only in critical situations that are abnormal. Never set the threshold too high that it can never be reached. Threshold values could be found and fixed when testing applications. The expensive statements trace records information about these statements for further analysis and is inactive by default.

If, in addition to activating the expensive statements trace, you can also enable per statement memory tracking, the expensive statements trace will show the peak memory size used to execute the expensive statements.

## SQL Optimization Step Debug Trace



The screenshot shows the SSMS interface with the title 'SQL Optimization Step Debug Trace'. In the Object Explorer, under 'My Catalog (STUDENT01)', there is a 'Database Diagnostic Files' node. Inside it, there is a folder named 'wdflibmrt7215' which contains several sub-folders: 'complexserver', 'daemon', 'diserver', 'dsrver', 'indexserver', 'nameserver', 'other', 'preprocessor', 'scriptsrvr', 'sqrltrace', 'webdispatcher', and 'wmoncontroller'. A context menu is open over a file named 'Indexserver MyTrace01.000'. The menu options are: Open, Delete, Download, Copy File Name, and Create Shortcut. The 'Open' option is highlighted.

**SqlOptStep debug trace logs the query optimization (QO) tree**

**Two types of optimization exist:**

- Rule-based rewriting
- Cost-based enumeration

**Rule Based shows:**

- QO tree after applying the rules (before and after applying) allows conclusion

**Cost Based shows:**

- logical tree of the final execution plan
- physical tree of the final execution plan
- Type of operators chosen

Figure 242: SQL Optimization Step Debug Trace

The SQL optimization step debug trace (SqlOptStep) logs the query optimization (QO) tree in each step of the optimization.

There are two main categories of optimization, rule-based rewriting and cost-based enumeration. In the case of rule-based rewriting, the SQL optimization step trace shows what the QO tree looks like after each rule has been applied. For example, you can see that the QO trees are different before and after calculation view unfolding. In the case of cost-based enumeration, the SQL optimization step trace shows the logical tree and the physical tree of the final execution plan. While the logical execution plan indicates which operators are located where, the physical execution plan tells you which types of operators were selected.



SQL Console 8.sql x indexserver\_wdfibmt7215.3000... x SQL Con: >

Showing: Lines From End of File Lines shown: 1488 Loaded File Size:

```

SET 'APPLICATIONUSER' = 'STUDENT01'
[13976]{310464}[40/-1] 2022-03-04 14:52:22.659585 d SqlOptTime
TraceContext.cpp(01347) : User
=STUDENT01, ApplicationName=S
StatementHash=feb722a616995b9fb998ac60e252039a
=8444378828134
[13976]{310464}[40/-1] 2022-03-04 14:52:22.659585 d SqlOptTime
TraceManager.cc(00247) : [Query Compile] Compiled Query String
=====
statement_hash: feb722a616995b9fb998ac60e252039a
call "STUDENT01"."my_sql_challenge" ('01.01.2015','23.02.2022',?)I
-----
* Parsing time: 0.157 ms
* Preprocessing time: 0.139 ms
* Checking time: 0.469 ms
* QP to QC conversion time: 0.068 ms
* Code generation time: 6.424 ms
* Total query compilation time: 8.370 ms

```

challenge  
All  
+ 1 of 4 .\* Aa \b S

Figure 243: SQL Optimization Step Debug Trace

The trace also provides more detailed information at the debug level. This includes estimated sizes, estimated subtree costs, applied rule names, applied enumerator names, partition information, column IDs, relation IDs, and much more.

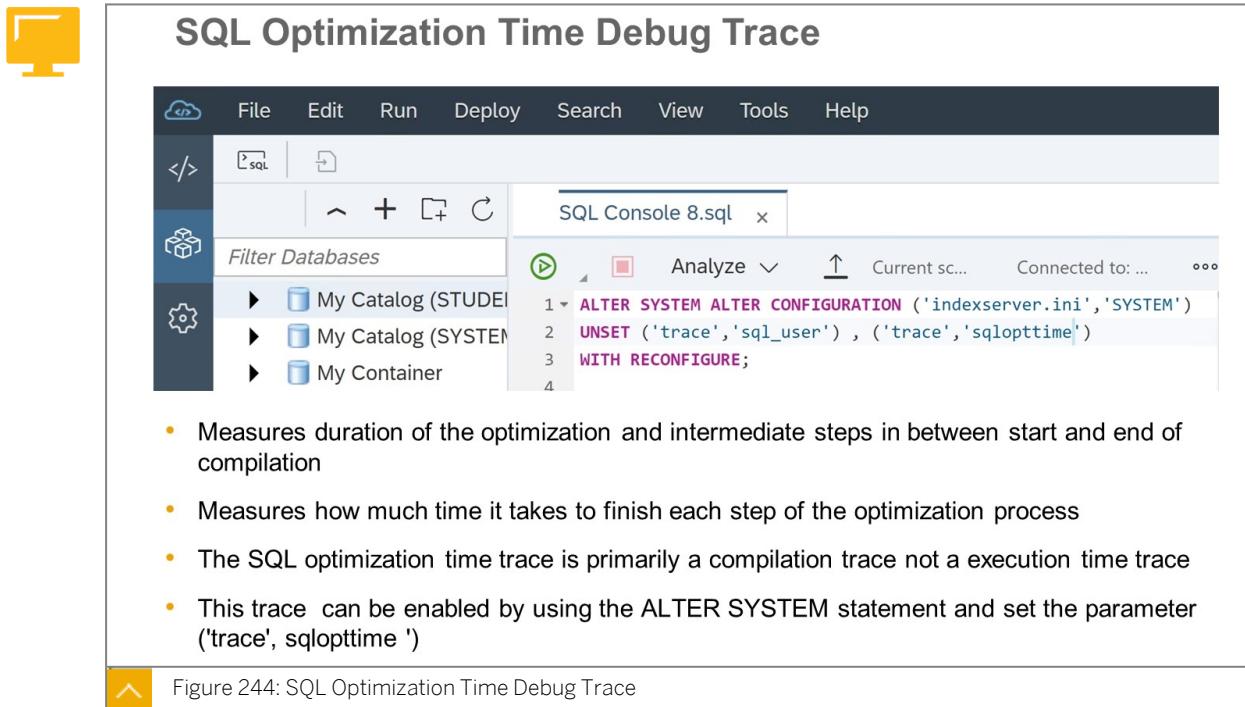
The fastest and most effective way to learn about the trace is to use it yourself. This means working through all the activities required to collect the trace and analyze it.



#### Note:

Tip: If you don't know the thread number or the transaction ID associated with the expensive statement in question you can search for the schema name in the log which narrows as a first approach.

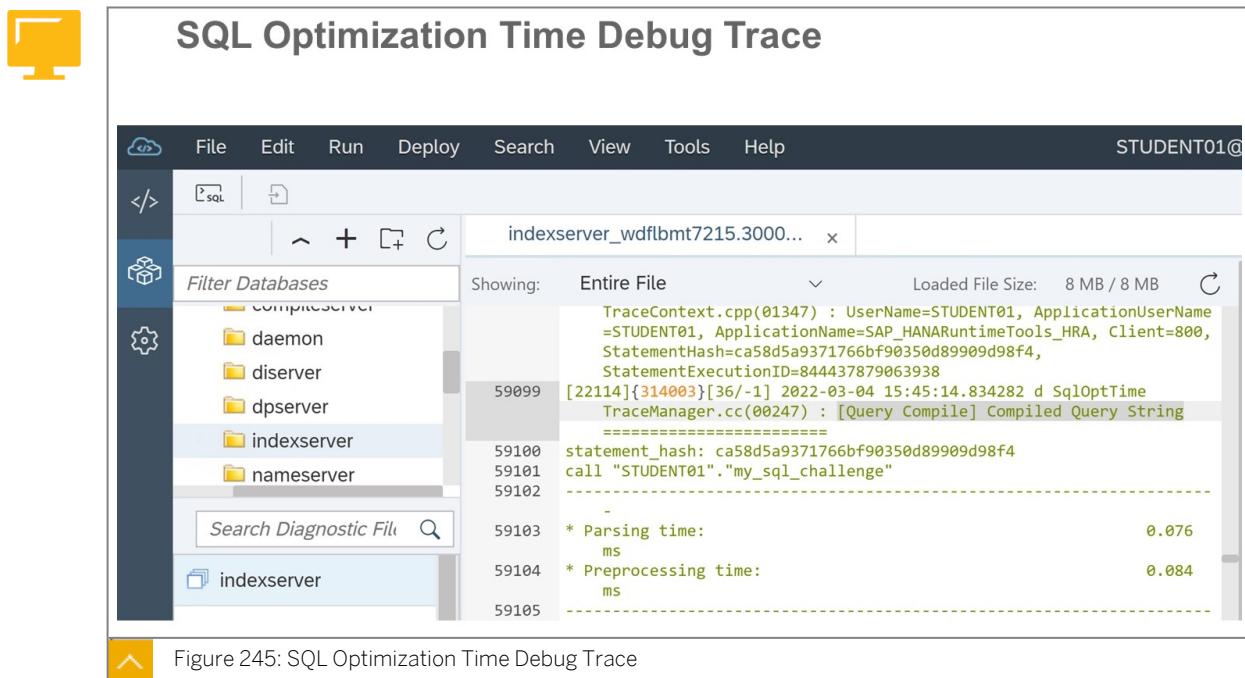
## SQL Optimization Time Debug Trace



The SQL optimization time trace (SqlOptTime) can be compared to a stopwatch. It measures how much time it takes to finish each step of the optimization process. It records not only the time required for the overall rule-based or cost-based optimization process but also the time consumed by each writing rule in rule-based optimization and the number of repeated enumerators during cost-based optimization.

The stopwatch for the SQL optimization time trace starts when compilation begins and ends when it finishes. It is primarily a compilation trace rather than an execution trace and can even be obtained by compiling a query and cancelling it immediately before execution. There is no point in collecting this trace if your aim is to improve execution time, particularly when this involves an out of memory condition or long running thread caused by a large amount of materialization. The SQL optimization time trace is helpful when there is a slow compilation issue.

With regard to execution time, there could be cases where long execution times can only be reduced at the expense of longer compilation times, so the total time taken from compilation to execution remains the same irrespective of which one you choose to reduce. Although these are execution time-related performance issues, you would still need the SQL optimization time trace to investigate whether the compilation time could be improved. Many cases involving long execution times could be closely related to compilation, such as when the optimizer reaches the enumeration limit, which causes it to stop and generate a plan. Or there might simply be a slow compilation issue where the purpose of the investigation is to reduce the compilation time.

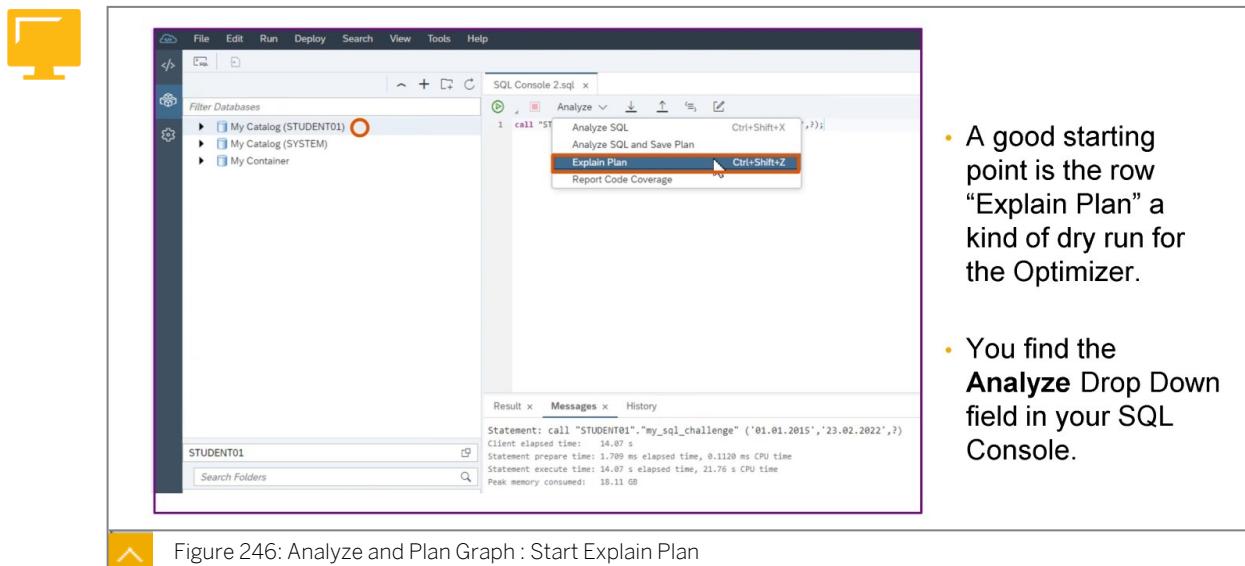


The SQL optimization time trace can also be used in a general sense to get a summarized report of the rules and enumerators used to construct the final plan. The SQL optimization step trace is usually very long, so a brief version that just tells you what choices the optimizer made rather than how they were made might be preferable. In that case, the SQL optimization time trace would be the appropriate tool.

But, Execution time and compilation time can be mutually dependent and closely connected to execution time.

The SQL optimization time trace is used in as a general summary report of the rules and enumerators used in the execution plan.

### Analyze and Plan Graph Start



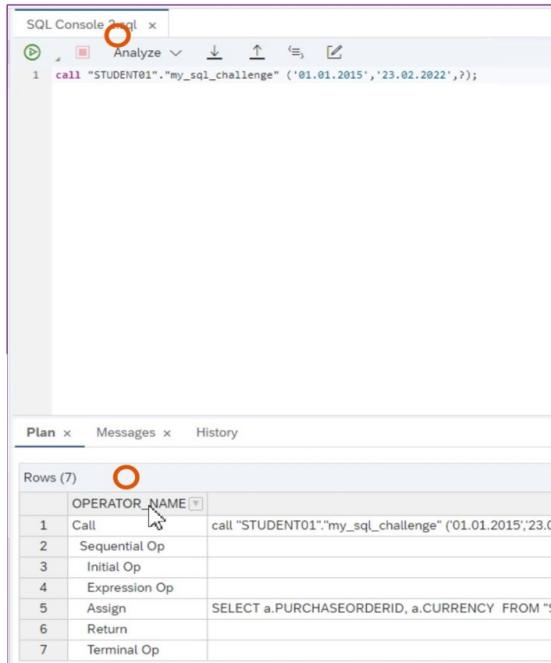
The Application Database Explorer offers the Analysis of the performance of pure SQL statements, SQLScript procedures and user defined scalar and tabular functions.

Do not mix it up with the SQLScript Code Analyzer. This tool checks code and searches proactively for patterns indicating potential problems with Checking rules. The Analyze functionality let you drill down to a particular piece of SQL Code analyzing the execution plan and Operators.

SAP Note 2373065

## Analyze and Plan Graph Result





- Explain Plan delivers a rough Overview of the Execution Plan chosen from the Optimizer
- In the result set of Explain Plan, the Operators are listed in a hierarchy and can be read from bottom up

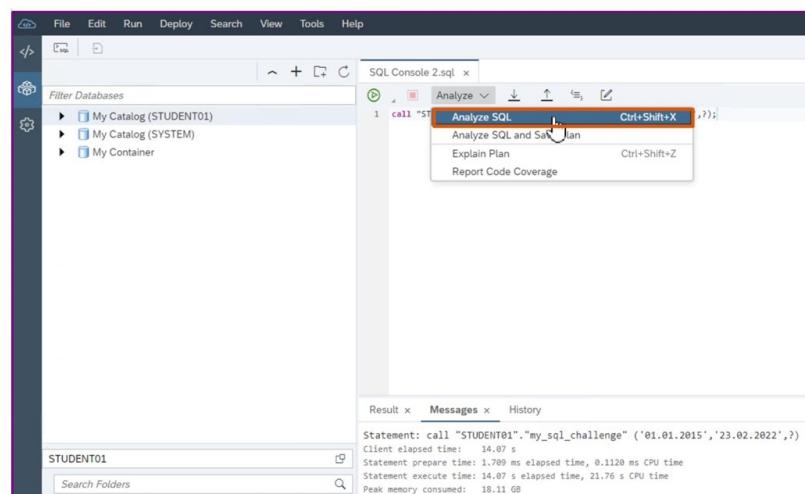
 Figure 247: Analyze and Plan Graph : Result Explain Plan

Explain Plan provides information about the compiled plan of a given procedure. Explain Plan generates the plan information by using the given SQLScript Engine Plan structure. It traverses the plan structure and records each information corresponding to the current SQLScript Engine Operator.

In the case of invoking another procedure inside of a procedure, Explain Plan inserts the results of the invoked procedure (callee) under the invoke operator (caller), although the actual invoked procedure is a sub-plan which is not located under the invoke operator.

Another case is the else operator. Explain Plan generates a dummy else operator to represent alternative operators in the condition operator.

## Analyze and Plan Graph Creating



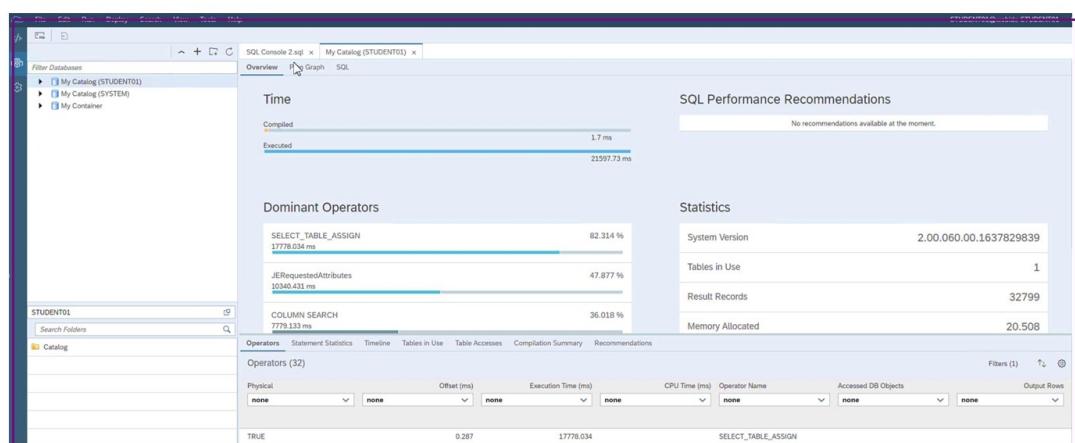
The screenshot shows the SAP Studio interface. In the top menu bar, the 'Analyze' dropdown is open, with 'Analyze SQL' highlighted. Below the menu, a SQL statement is being run in the 'SQL Console'. The results show the execution time and memory usage for the statement. A context menu is open over the statement, with 'Analyze SQL' also highlighted.

- **Analyze SQL** creates a Plan Graph in a new tab within the Database Explorer.
- With this Plan you can start analyzing your SQL

Figure 248: Analyze and Plan Graph : Creating Plan Graph

This Plan lets you thoroughly analyze your SQL, detect details like involved tables, materializations, execution engine usage, and a convenient timeline of execution steps.

## Analyze and Plan Graph Exploring



The screenshot shows the SAP Studio interface with the Plan Graph open. The main area displays three tabs: Overview, Graph, and SQL. The Overview tab is selected, showing a timeline of execution steps (Compiled and Executed) and a list of Dominant Operators. The Graph tab likely contains a visual representation of the query plan. The SQL tab shows the original SQL statement and its execution details.

- The Plan Graph provides three main tabs and seven sub-tabs
- The Overview tab provides a good starting point for exploring the Plan Graph
- On sub-tabs Operators, Tables in Use and more can be filtered

Figure 249: Analyze and Plan Graph : Exploring Plan Graph

In the sub-tab Operators the sum of involved Operators are presented and their system.

## Analyze and Plan Graph Analyzing

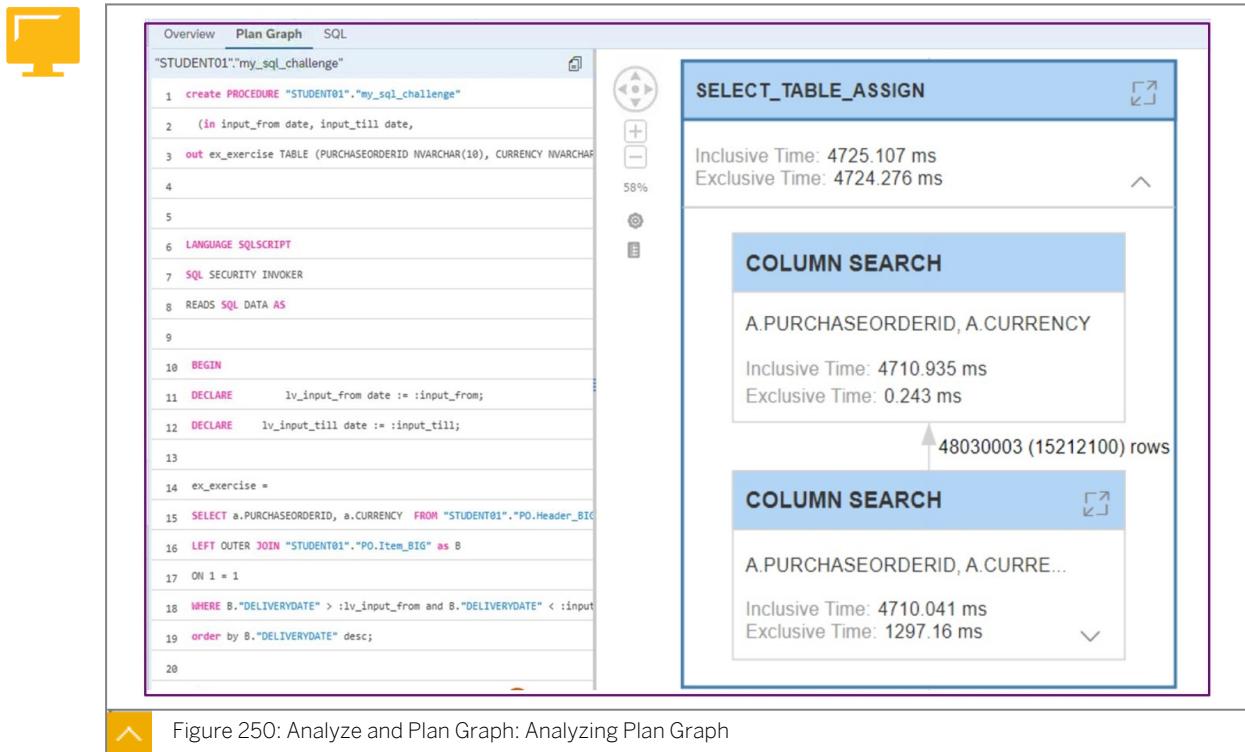


Figure 250: Analyze and Plan Graph: Analyzing Plan Graph

The Plan Graph shows each execution step with inclusive and exclusive time measures in microsecond units. One can read this plan from bottom up. It is shown initially folded but each step can be unfolded to analyze processing of the Optimizer sub steps. The Engine Usage chosen by the Optimizer can be detected and in parenthesis, you can compare the estimated rows with the real rows that were evaluated.

You can revise the execution plan of a SQL statement. It displays a visualization of a critical path based on inclusive execution time of operators and allows you to identify the most expensive path in a query execution plan.

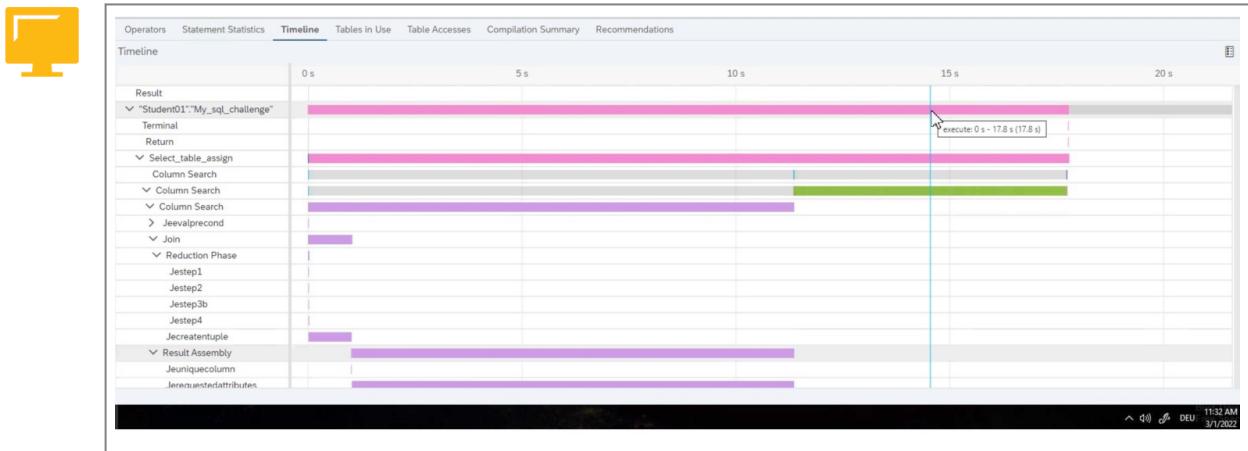
In case of a SQLScript, the Plan Graph displays its complete definition. To retrieve the information in a text format, you can copy the definition by clicking the copy icon.

In the Plan Graph tab, you can open the Detail Properties view by clicking one of the operators. This view offers detailed information on the operator such as name, location, ID, summary.

You can open the edge information detail by clicking one of the links between the operators. This view offers further information on the edge values, such as target, source, output cardinality, fetch call count, and estimated output cardinality.

Furthermore, you can configure plan graph settings. You can set the color of the nodes by type or location, and choose to show either physical or logical inner plans.

## Analyze and Plan Graph Timeline



- The **Timeline sub-Tab** shows each execution step with its duration
- Each row represents on execution step with its duration
- The result hierarchy can be unfolded to drill down

Figure 251: Analyze and Plan Graph: Timeline

- The Timeline can be read from the left to the right. The Result steps on the left side (Operator tree table) are organized hierarchically. You get a complete overview of the execution plan based on the visualization of sequential time-stamps. The operator tree table displays hierarchical parent-child relationships and container-inner plan relationships. Based on the operators, the timeline chart shows the operations executed at different time intervals
- The Operator tree table initially is shown folded but each step can be unfolded to analyze the processing of the Optimizer sub steps so as to find out reasons for performance and starting point for tuning activities.

## Analyze and Plan Graph Saving

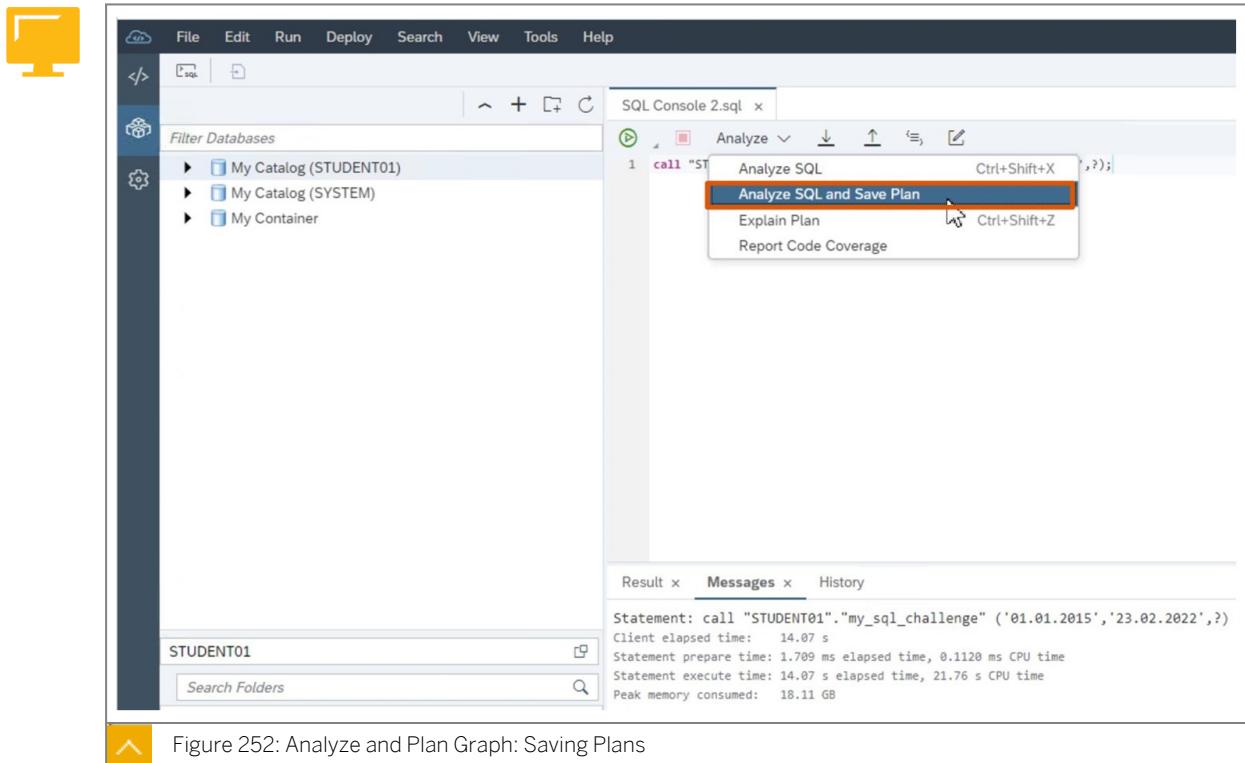


Figure 252: Analyze and Plan Graph: Saving Plans

Generated plans can be saved (suffix: .plv) for instance on the client and loaded and analyzed into another application like SAP Business Application Studio, which is a cloud tool.

In that way different execution plans on different environments (e.g. hybrid scenarios) can be compared to reveal and solve performance breaks within SAP HANA.



### LESSON SUMMARY

You should now be able to:

- Apply Views and Tools to optimize SQL Performance

## Further Tools for Troubleshooting



### LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Usage of further tools for troubleshooting SQLScript

### SQLScript Code Analyzer

The SQLScript Code Analyzer is a tool that is used to scan the code of **functions** and **procedures** to search for patterns indicating potential problems.



#### SQLScript Code Analyzer

- Scan **functions** and **procedures** statements and searching for patterns indicating problems in code quality, security, and performance.
- In the Web IDE Database Explorer choose *Analyze SQLScript Code* from the context menu of a database, container, schema, procedure, or function.

#### SQLScript Code Analysis Results

Analyzed all functions and procedures in database 08 Container HA150 for security and consistency issues.

Database: 08 Container HA150

Objects Analyzed: 14

Include Objects without Issues

| Object Name                  | Issue Description                                                                                | Rule Category |
|------------------------------|--------------------------------------------------------------------------------------------------|---------------|
| HA150::PlateNumber_Temp_proc |                                                                                                  |               |
|                              | Variable V_PLATENUMBER has no influence on the function/procedure outcome                        | CONSISTENCY   |
|                              | Variable CUR_ROW has no influence on the function/procedure outcome                              | CONSISTENCY   |
|                              | Assigned value of variable CUR_ROW was not used                                                  | CONSISTENCY   |
|                              | Assigned value of variable V_PLATENUMBER was not used                                            | CONSISTENCY   |
|                              | Found DML statement in a loop. DML statements in loops can have a negative impact on performance | PERFORMANCE   |

Figure 253: SQLScript Code Analyzer

The tool looks for issues in the code relating to the following:

- Consistency
- Style
- Security
- Performance



### SQL Script Code Analyzer – 12 Checking Rules

| RULE_NAME                         | CATEGORY    |
|-----------------------------------|-------------|
| COMMIT_OR_ROLLBACK_IN_DYNAMIC_SQL | STYLE       |
| DML_STATEMENTS_IN_LOOPS           | PERFORMANCE |
| ROW_COUNT_AFTER_DYNAMIC_SQL       | BEHAVIOR    |
| ROW_COUNT_AFTER_SELECT            | BEHAVIOR    |
| SINGLE_SPACE_LITERAL              | CONSISTENCY |
| UNCHECKED_SQL_INJECTION_SAFETY    | SECURITY    |
| UNNECESSARY_VARIABLE              | CONSISTENCY |
| UNUSED_VARIABLE_VALUE             | CONSISTENCY |
| USE_OF_CE_FUNCTIONS               | PERFORMANCE |
| USE_OF_DYNAMIC_SQL                | PERFORMANCE |
| USE_OF_SELECT_IN_SCALAR_UDF       | PERFORMANCE |
| USE_OF_UNASSIGNED_SCALAR_VARIABLE | CONSISTENCY |

Figure 254: SQLScript Code Analyzer Rules

There are currently (SAP HANA 2.0 SPS05) twelve rules and these are defined in a system view `SQLSCRIPT_ANALYZER_RULES`.

- **UNNECESSARY\_VARIABLE** – Each variable is tested to identify if it is used by any output parameter of the procedure or if it influences the outcome of the procedure. Relevant statements for the outcome could be DML statements, implicit result sets, conditions of control statements.
- **UNUSED\_VARIABLE\_VALUE** – If a value assigned to a variable is not used in any other statement, the assignment can be removed. In case of default assignments in `DECLARE` statements, the default is never used.
- **SINGLE\_SPACE\_LITERAL** – This rule searches for string literals consisting of only one space. If ABAP VARCHAR MODE is used, such string literals are treated as empty strings. In this case, CHAR(32) can be used instead of ''.
- **USE\_OF\_UNASSIGNED\_SCALAR\_VARIABLE** – detects variables which are used but were never assigned explicitly. Those variables will still have their default value when used, which might be undefined. We recommend that you assign a default value (can be NULL) to be sure that you get the intended value when you read from the variable.
- **USE\_OF\_CE\_FUNCTIONS** – checks whether Calculation Engine Plan Operators (CE functions) are used. Since they make optimization more difficult and lead to performance penalties, they should be avoided.
- **DML\_STATEMENTS\_IN\_LOOPS** – detects the following DML statements inside of loops: INSERT, UPDATE, DELETE, REPLACE/UPSERT. Sometimes it is possible to rewrite the loop and use a single DML statement to improve performance instead.
- **USE\_OF\_SELECT\_IN\_SCALAR\_UDF** – detects if SELECT is used within a scalar UDF which can lower the performance. If table operations are really needed, procedures or Table UDFs should be used instead.

- **UNCHECKED\_SQL\_INJECTION\_SAFETY** – Parameters of string type should always be checked for SQL injection safety if they are used in dynamic SQL. This rule checks if for any such parameter the function `is_sql_injection_safe` was called. For a simple conditional statement like `IF is_sql_injection_safe(:var) = 0 THEN...`, the control flow in the true branch is checked. The procedure should either end (by returning or by throwing an error) or the unsafe parameter value should be escaped with the functions `escape_single_quotes` or `escape_double_quotes`, depending on where the value is used. If the condition is more complex (for example, more than one variable checked in one condition), a warning will be displayed, as it could only be checked if any execution of the dynamic SQL has passed the SQL injection check.
- **COMMIT\_OR\_ROLLBACK\_IN\_DYNAMIC\_SQL** – detects dynamic SQL which use the COMMIT or ROLLBACK statement. Since COMMIT and ROLLBACK can be used directly in SQLScript without the need of dynamic SQL, it is recommended to use COMMIT and ROLLBACK directly.

The SQLScript Code Analyzer is launched in the SAP Web IDE Database Explorer from the context menu of:

- **Database** — To scan all procedures and function in the database (launched from the catalog DB connection type)
- **Schema** — To scan all procedures and function in a specific schema (launched from the catalog DB connection type)
- **Container**
  - To scan all procedures and function in a specific HDI container (launched from the HDI container connection type)
- **Procedure**
  - To scan a specific procedure
- **Function**
  - To scan a specific function

You can download the results of a code analysis to a .CSV file. Use the download button in the results screen.

## SQL Analyzer

The *SQL Analyzer* is a tool that is used to check runtime performance of your SQL statements. The tool can help you evaluate potential bottlenecks and optimizations for these queries.

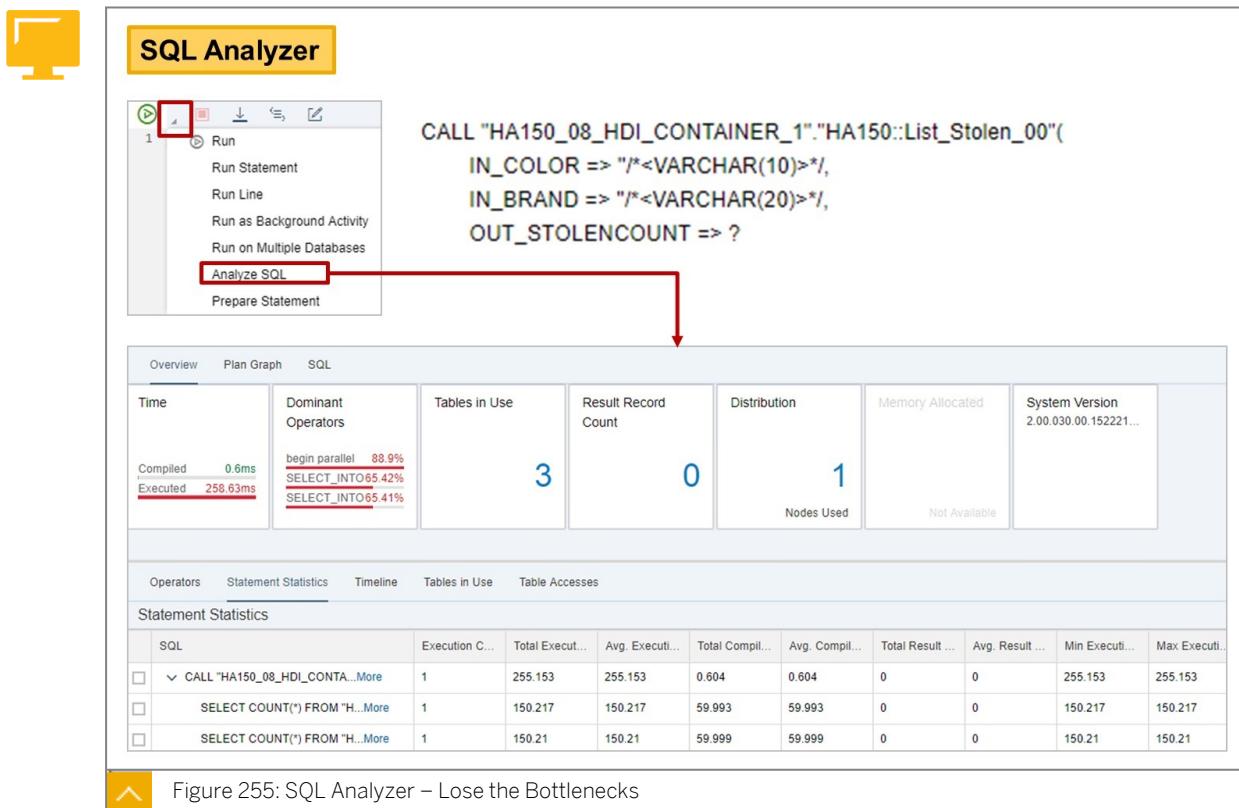


Figure 255: SQL Analyzer – Lose the Bottlenecks

You launch the *SQL Analyzer* by using the drop down menu from the *Execute* button.



#### Note:

Be careful not to confuse *SQL Analyzer* with *SQL Code Analyzer*.

Once launched you can navigate the various views that contain the following information:

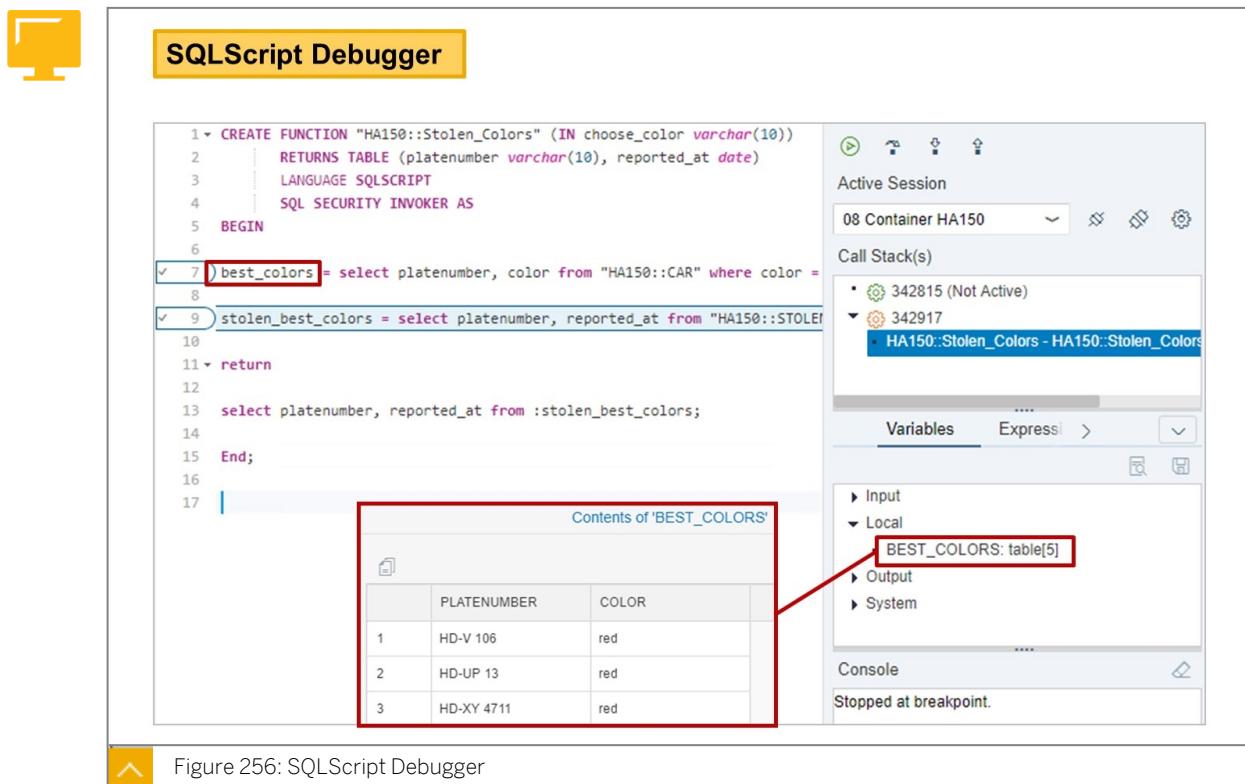
- **Overview** — Displays tiles that expose a summary of the key information. Useful for a quick glance.
- **Plan Graph** — This view provides graphical guidance to help you understand and analyze the execution plan of an SQL statement.
- **Timeline** — This view provides a complete overview of the execution plan based on the visualization of sequential time stamps.
- **Tables Used** — This view provides a list of tables used during query execution and includes further details on tables, which can be used for further analysis. It can be used to understand which tables are needed to fulfill a given SQL statement.
- **Operator List** — This view provides a list of operators used during query execution and includes additional details about each operator, which can be used for analysis.
- **Statement Statistics** — This view is only displayed in the case of an SQLScript. This view provides details on the statistics of the executed statements in the SQLScript.
- **Table Accesses** — Number of records returned and filter values used, and so on.

Based on the analysis, if needed, the following changes can be performed for optimization:

- Change the SQL statement
- Change the data model
- Change the physical design
- Add SQL hints to statements

## SQL Debugger

You can step through the code and examine the run time behavior of your **procedures** and **functions** using the *SQLScript Debug tool*.



You call the debugger from the *Database Explorer* view of SAP Web IDE by right-clicking any procedure or function and choosing *Open for Debugging*.

You must then attach a debug session to the procedure or function. This is where you choose a database connection on which to run the debug session.

Then you set the breakpoints in your code by double-clicking on the code line where you then see a breakpoint marker.

You can optionally set conditional breakpoints where an expression is entered that must be true to stop the code.

Finally, right-click anywhere in the code and use the option *Call Function* or *Call Procedure*. You will be presented with the SQL calling statement where you can insert any input variables if applicable. Launch the procedure or function using F8 and the code will stop at the first breakpoint where you can examine the contents of any variables or parameters.

Use the control buttons to step through the code.



Note:  
Don't forget to check the debug session time-out value in the *Web IDE Preferences*.



### LESSON SUMMARY

You should now be able to:

- Usage of further tools for troubleshooting SQLScript

## Lesson 1

Using Sub Queries

263

## Lesson 2

Defining How Data Is Stored

275

## Lesson 3

Using Views for Data Access

285

## Lesson 4

Defining Data Access

293

## Lesson 5

Explaining Database Transactions

301

## UNIT OBJECTIVES

- Read data using sub queries
- List the most important data types SAP HANA supports
- Create new database tables in HANA
- Change tables by adding, removing or renaming columns
- Describe the use cases for and advantages of using database views, define database views and use them in queries
- Understand database schemas and access tables in other schemas
- Explain when database indexes make sense in SAP HANA and create and delete indexes using SQL
- Explain database transactions and the ACID requirements
- Finish database transactions in SAP HANA using SQL statements
- Describe issues that arise if transactions are not mutually isolated

- Understand and control isolation levels of transactions and how SAP HANA handles concurrency

# Using Sub Queries

## LESSON OVERVIEW

This lesson covers how sub queries can be used to retrieve data from several tables in a single statement.



## LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Read data using sub queries

## Sub Queries Overview

In addition to UNION and JOIN, **sub queries** or **nested queries** also provide the option of reading from multiple tables and views.

Nested queries contain what is called a “sub query”.



- A **sub query** is a query that is used in SQL statements; most commonly, this is another query.
- The query containing the sub query is called the “outer query”.
- In this context, the sub query is also referred to as the “inner query”.

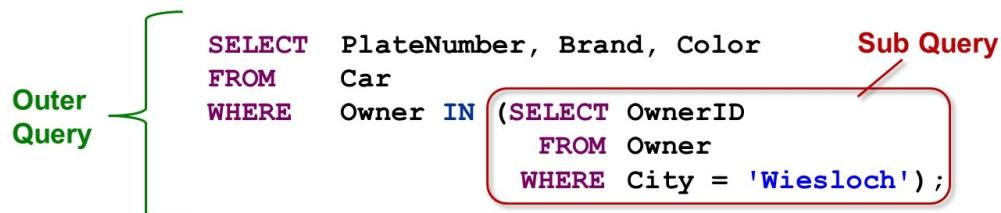


Figure 257: Sub Query: Explanation

A sub query can be used to achieve the following:

- Make a SELECT statement more “readable”.
- Improve the performance of a SELECT statement, however, with a “perfect” optimizer, this should not be necessary.
- Formulate a SELECT statement that cannot be formulated (or only with difficulty) without a sub query. For example, it is not possible to put aggregate expressions in the WHERE clause of a SELECT statement, unless they are part of a sub query.

There are different types of sub queries:

- Expression sub queries return exactly one row; quantified predicate sub queries may return zero or multiple rows.

- Uncorrelated sub queries are complete SELECT statements that may be run on their own; correlated sub queries are incomplete and must refer to the outer query.

These two determinations (expression versus quantified predicate; uncorrelated versus correlated) are independent. This gives four possible types of sub queries.



**Note:**

It is sometimes possible to get the same result from a correlated sub query and an uncorrelated sub query. In these cases, the uncorrelated sub query will almost certainly involve fewer calculation costs than the correlated sub query.



**An expression sub query returns exactly one row (CarID values are unique).**

- Boolean expressions may be used to compare against the sub query.

```
SELECT  *
FROM    Owner
WHERE   OwnerID = (SELECT Owner
                    FROM  Car
                    WHERE CarID = 'F08') ;
```

A quantified predicate subquery may return zero or many rows (there may be any number of owners in the city of Wiesloch).

- Boolean expressions may not be used because one value cannot be equal to (or greater, or less than) multiple values. IN and EXISTS may be used instead.

```
SELECT  *
FROM    Car
WHERE   Owner IN  (SELECT OwnerID
                    FROM  Owner
                    WHERE City = 'Wiesloch') ;
```



Figure 258: Differences Between Existence and Quantified Predicate Subqueries



An uncorrelated sub query makes no reference to the outer query.

```
SELECT *
  FROM Car
 WHERE Owner IN (SELECT OwnerID
                  FROM Owner
                 WHERE City = 'Wiesloch');
```

A correlated sub query refers to the outer query.

```
SELECT *
  FROM Car c
 WHERE EXISTS (SELECT *
                  FROM Owner o
                 WHERE o.OwnerID = c.Owner
                   AND o.City = 'Wiesloch');
```

Figure 259: Uncorrelated Versus Correlated Sub Query

## Uncorrelated Sub Queries



This query is uncorrelated because the inner query is complete.

- The inner query, if run on its own, would return the OwnerID values H01 and H06.
- The outer query is equivalent to one using the logical test

```
WHERE Owner IN ('H01' , 'H06')
```

- Which cars are registered to an owner from Wiesloch?

```
SELECT *
  FROM Car
 WHERE Owner IN (SELECT OwnerID
                  FROM Owner
                 WHERE City = 'Wiesloch');
```

| CARID | PLATENUMBER | BRAND | COLOR | HP  | OWNER |
|-------|-------------|-------|-------|-----|-------|
| F01   | HD-V 106    | Fiat  | red   | 75  | H06   |
| F19   | HD-VW 2012  | VW    | black | 125 | H01   |

Figure 260: Frequent Case: Combining IN with a Sub Query



You can use `= ANY`, if the outer value should match any result value of the sub query.

- Which cars are registered to an owner from Wiesloch?

```
SELECT *
FROM Car
WHERE Owner = ANY ( SELECT OwnerID
                      FROM Owner
                      WHERE City = 'Wiesloch');
```

| CARID | PLATENUMBER | BRAND | COLOR | HP  | OWNER |
|-------|-------------|-------|-------|-----|-------|
| F01   | HD-V 106    | Fiat  | red   | 75  | H06   |
| F19   | HD-VW 2012  | VW    | black | 125 | H01   |



Figure 261: The ANY Expression



`= ANY` is equivalent to `IN`,  
but you can use `ANY` with other comparison operators:

|                           |                                                                            |
|---------------------------|----------------------------------------------------------------------------|
| <code>= ANY</code>        | The external value matches any value of the sub query.                     |
| <code>&lt; ANY</code>     | The external value is less than any value of the sub query.                |
| <code>&lt;= ANY</code>    | The external value is less than or equal to any value of the sub query.    |
| <code>&gt; ANY</code>     | The external value is greater than any value of the sub query.             |
| <code>&gt;= ANY</code>    | The external value is greater than or equal to any value of the sub query. |
| <code>&lt;&gt; ANY</code> | The external value is different to any value of the sub query.             |



Figure 262: Uses of the ANY Expression



You can use `> ANY`, if the external value should be greater than any value of the sub query.

- Which officials have more than the minimum overtime hours?

```
SELECT Name, Overtime
FROM Official
WHERE Overtime > ANY (SELECT Overtime
                        FROM Official);
```

| NAME | OVERTIME |
|------|----------|
| Ms C | 20       |
| Mr F | 18       |
| Ms G | 22       |



Figure 263: Example for > ANY



You can use `>` instead of `> ANY`, if the sub query results in only a single value.

- Which officials have more than the minimum overtime hours?

```
SELECT Name, Overtime
FROM Official
WHERE Overtime > (SELECT MIN Overtime
                   FROM Official);
```

| NAME | OVERTIME |
|------|----------|
| Ms C | 20       |
| Mr F | 18       |
| Ms G | 22       |

Figure 264: Special Case: Single-Row Sub Query



You can use `< ALL`, if the outer value should be less than all result values of the sub query (other comparison operators are possible)

|                           |                                                                                                                                                  |
|---------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>= ALL</code>        | The external value matches each result value of the sub query.<br>(This option is quite useful, as the sub query result can contain duplicates.) |
| <code>&lt; ALL</code>     | The external value is less than all values of the sub query.                                                                                     |
| <code>&lt;= ALL</code>    | The external value is less than or equal to all values of the sub query.                                                                         |
| <code>&gt; ALL</code>     | The external value is greater than all values of the sub query.                                                                                  |
| <code>&gt;= ALL</code>    | The external value is greater than or equal to all values of the sub query.                                                                      |
| <code>&lt;&gt; ALL</code> | The external value is different to all values of the sub query.                                                                                  |

Figure 265: The ALL Expression



The ANY and ALL operators allow for comparisons against quantified predicate subqueries.

- Which officials have the fewest overtime hours?

```
SELECT Name, Overtime
FROM Official
WHERE Overtime <= ALL (SELECT Overtime
                       FROM Official
                       WHERE Overtime IS NOT NULL);
```

| NAME | OVERTIME |
|------|----------|
| Mr A | 10       |
| Mr B | 10       |
| Mr E | 10       |

Figure 266: <=ALL Example



However, the ANY and ALL operators are frequently misunderstood.

If the quantified predicate subquery can be made into an expression subquery, ANY and ALL can be avoided.

The MAX and MIN aggregates can be used to convert the subquery.

- Which officials have the fewest overtime hours?
- For the example below = would be possible instead of `<=` as well.

```
SELECT Name, Overtime
FROM Official
WHERE Overtime <= (SELECT MIN Overtime
                     FROM Official);
```

| NAME | OVERTIME |
|------|----------|
| Mr A | 10       |
| Mr B | 10       |
| Mr E | 10       |



Figure 267: Special Case: Single-Row Sub Query

## Correlated Sub Queries



A correlated sub query refers to the outer query.

Using `EXISTS` you can check that the query result of the sub query is not empty.

- Which vehicles are registered to an owner from Wiesloch?

```
SELECT *
  FROM Car c
 WHERE EXISTS (SELECT *
                  FROM Owner o
                 WHERE o.City = 'Wiesloch' AND o.OwnerID = c.Owner);
```

| CARID | PLATENUMBER | BRAND | COLOR | HP  | OWNER |
|-------|-------------|-------|-------|-----|-------|
| F01   | HD-V 106    | Fiat  | red   | 75  | H06   |
| F19   | HD-VW 2012  | VW    | black | 125 | H01   |



Figure 268: Correlated Sub Query with EXISTS



**Even with a correlated sub query, you can omit table aliases if the column names are unique.**

- What vehicles are registered to an owner from Wiesloch?

```
SELECT *  
FROM Car  
WHERE EXISTS (SELECT *  
              FROM Owner  
              WHERE City = 'Wiesloch' AND OwnerID = Owner);
```

| CARID | PLATENUMBER | BRAND | COLOR | HP  | OWNER |
|-------|-------------|-------|-------|-----|-------|
| F01   | HD-V 106    | Fiat  | red   | 75  | H06   |
| F19   | HD-VW 2012  | VW    | black | 125 | H01   |

Figure 269: Table Aliases can be Optional



**Using NOT EXISTS you can check if the result of the sub query is empty.**

- Which cars are not reported as stolen?

```
SELECT CarID  
      FROM Car c  
 WHERE NOT EXISTS (SELECT *  
                      FROM Stolen s  
                      WHERE s.PlateNumber =  
                            c.PlateNumber);
```

| CARID |
|-------|
| ----- |
| F02   |
| F03   |
| F04   |
| F05   |
| F07   |
| F08   |
| F09   |
| F10   |
| F11   |
| F12   |
| F13   |
| F14   |
| F15   |
| F16   |
| F18   |
| F19   |
| F20   |

Figure 270: Correlated Sub Query with NOT EXISTS



You can use the same table in both the **FROM** clause of the outer query and the **FROM** clause of the correlated sub query.

- Which vehicle has the most HP?

```
SELECT *
  FROM Car c1
 WHERE NOT EXISTS (SELECT *
                      FROM Car c2
                     WHERE c2.HP > c1.HP);
```

| CARID | PLATENUMBER | BRAND | COLOR  | HP  | OWNER |
|-------|-------------|-------|--------|-----|-------|
| F06   | HD-VW 1999  | Audi  | yellow | 260 | H05   |

Figure 271: Correlated Sub Query Using the Same Table



Even when using a correlated sub query, you can for example use **>= ALL**

- How much horsepower has the most powerful car of each brand?

```
SELECT DISTINCT Brand, HP
  FROM Car c1
 WHERE c1.HP >= ALL (SELECT c2.HP
                      FROM Car c2
                     WHERE c2.Brand = c1.Brand);
```

| BRAND    | HP  |
|----------|-----|
| Fiat     | 75  |
| BMW      | 184 |
| Mercedes | 170 |
| Audi     | 260 |
| VW       | 160 |
| Renault  | 136 |
| Skoda    | 136 |
| Opel     | 120 |

Figure 272: Correlated Sub Query with **>= ALL**



The WHERE clause of a correlated sub query can contain an additional correlated sub query.

- To whom is (at least) one stolen car registered?

```
SELECT o.Name
  FROM Owner o
 WHERE EXISTS
    (SELECT *
      FROM Car c
     WHERE c.Owner = o.OwnerID)
      AND EXISTS (SELECT *
                   FROM Stolen s
                  WHERE s.PlateNumber = c.PlateNumber)
;
```

Figure 273: Correlated Sub Queries can be Nested



You can also use a correlated sub query in the SELECT clause.

- How much horsepower is each car lacking compared to the most powerful car of the same brand?

```
SELECT CarID, Brand, (SELECT MAX (HP)
                        FROM Car c2
                       WHERE c2.Brand = c1.Brand) - HP
AS Difference
FROM Car c1;
```

| CARID | BRAND    | DIFFERENCE |
|-------|----------|------------|
| F01   | Fiat     | 0          |
| F02   | VW       | 40         |
| F03   | BMW      | 0          |
| F04   | Mercedes | 34         |
| F05   | Mercedes | 0          |
| F06   | Audi     | 0          |
| F07   | Audi     | 144        |
| ...   | ...      | ...        |
| F20   | Audi     | 76         |

Figure 274: Correlated Sub Query in the SELECT Clause

Using nested queries you can also perform the following:

- Restrict the projection list to certain columns.
- Explicitly rename the result columns.
- Sort the query result.
- Use grouping.
- Include aggregate expressions.
- Eliminate duplicates using DISTINCT.
- Involve the same table multiple times.

- Involve more than two tables.

## Extending Sub Queries



You can use functions in a sub query.

- Which owners were born in the same year as the youngest owner?

```
SELECT *
  FROM Owner
 WHERE YEAR(Birthday) = (SELECT MAX(YEAR(Birthday))
                           FROM Owner);
```

| OWNERID | NAME | BIRTHDAY   | CITY      |
|---------|------|------------|-----------|
| H08     | Mr X | 1986-08-30 | Walldorf  |
| H09     | Ms Y | 1986-02-10 | Sinsheim  |
| H10     | Mr Z | 1986-02-03 | Ladenburg |



Figure 275: Sub Queries Can Use Functions



You can use multiple columns in the sub query.

- To whom within the EU is (at least) one blue car registered?
- ```
SELECT Name
  FROM Owner_EU
 WHERE (Country, OwnerID) IN (SELECT Country, Owner
                                FROM Car_EU
                               WHERE Color = 'blue');
```

NAME
SAP AG
Señor Q



Figure 276: Multiple Columns in a Sub Query



### The WHERE clause of a sub query can contain another sub query.

- Who are the owners of stolen cars?

```
SELECT Name
  FROM Owner
 WHERE OwnerID IN
    (SELECT Owner
      FROM Car
     WHERE PlateNumber IN
          (SELECT PlateNumber FROM Stolen)
    );
```

NAME
-----
Mr V
Ms W
MS Y

Figure 277: The Where Clause of a Sub Query



### You can combine joins and sub queries.

- To whom is the most powerful car per brand registered?
- ```
SELECT DISTINCT Brand, Name
  FROM Owner RIGHT OUTER JOIN Car
    ON OwnerID = Owner
 WHERE (Brand, HP) IN
    ( SELECT Brand, MAX (HP)
      FROM Car
     GROUP BY Brand )
 ORDER BY Brand ASC, Name ASC;
```

| BRAND    | NAME   |
|----------|--------|
| -----    | -----  |
| Audi     | Mr V   |
| BMW      | Ms U   |
| BMW      | SAP AG |
| Fiat     | Ms W   |
| Mercedes | SAP AG |
| Opel     | ?      |
| Renault  | IKEA   |
| Skoda    | SAP AG |
| VW       | IKEA   |

Figure 278: Combine Joins and Sub Queries



**A sub query is usually part of the WHERE clause.**  
**But you can also use a sub query in other clauses.**

- Which blue cars have less than 120 HP?

```
SELECT *
  FROM (SELECT PlateNumber, HP AS Power
        FROM Car
       WHERE Color = 'blue')
 WHERE Power < 120;
```

| PLATENUMBER | POWER |
|-------------|-------|
| HD-ML 3206  | 116   |



Figure 279: Sub Queries Are Usually Part of the Where Clause



**A sub query is usually part of the WHERE clause.**  
**But you can also use a sub query in other clauses.**

- What is the HP deviation of each car when compared to the most powerful yellow car?

```
SELECT CarID, Brand,
       (SELECT MAX (HP)
        FROM Car
       WHERE Color = 'yellow') - HP AS Deviation
     FROM Car
    ORDER BY 3 DESC;
```

| CARID | BRAND   | DEVIATION |
|-------|---------|-----------|
| F01   | Fiat    | 185       |
| F18   | Renault | 170       |
| F09   | Skoda   | 155       |
| ...   | ...     | ...       |
| F06   | Audi    | 0         |



Figure 280: Other Clauses



## LESSON SUMMARY

You should now be able to:

- Read data using sub queries

## Defining How Data Is Stored

### LESSON OVERVIEW

This lesson covers how you can create and change database tables to define how data is stored.



### LESSON OBJECTIVES

After completing this lesson, you will be able to:

- List the most important data types SAP HANA supports
- Create new database tables in HANA
- Change tables by adding, removing or renaming columns

### Managing Tables in SAP HANA — Overview

Before data can be inserted into database tables, the tables themselves have to be defined and created.

#### Some Options to Manage Database Tables in SAP HANA

SAP HANA supports several means of managing tables, including:



- Using SQL DDL Statements. This is the only option covered in this course.
- Using a form-based editor in the SAP HANA Tools.
- Using Core Data Services and the HANA Development Perspective.



#### Note:

SQL and the form-based editor provide no support for managing the lifecycle of database tables, for example for keeping track of how a table was defined or changed so that changes can be transported to another system without losing data in the target system.

Nevertheless, this course only covers the SQL option. Understanding it can be helpful when, for example, prototyping or analyzing issues.

### SAP HANA SQL Types

When managing database tables on the SQL level, it is important to know the data type system provided by SAP HANA on the SQL level. These are outlined in the following table:



Table 18: Numerical Data Types

The following table lists the numerical SQL data types supported by SAP HANA.

|              | Data Type    | Remark                                                                                                                                                                   |
|--------------|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Integers     | TINYINT      | unsigned, 8-bit integer values                                                                                                                                           |
|              | SMALLINT     | signed, 16-bit integer values                                                                                                                                            |
|              | INTEGER      | signed, 32-bit integer values                                                                                                                                            |
|              | BIGINT       | signed, 64-bit integer values                                                                                                                                            |
| Real numbers | DECIMAL(p,s) | Fixed-point decimal number with total number of digits p (precision) and s (scale) digits right of the decimal point (fractional digits).                                |
|              | SMALLDECIMAL | Floating point number with precision varying dynamically between 1 and 16 and scale between -369 and 368, respectively between 1 and 34 and scale between -6111 and 6167 |
|              | DECIMAL      | 32-bit floating point number                                                                                                                                             |
|              | REAL         | 64-bit floating point number                                                                                                                                             |
|              | FLOAT(N)     | 32-bit or 64-bit real number with N significant bits. 1 <= N <= 53.                                                                                                      |

**Note:**

DECIMAL (**p,s**) is suitable for storing fixed-point decimal numbers with **p** digits in total and **s** fractional digits, whereas DECIMAL (without precision and scale) is used to store floating point numbers.

Since floating point numbers often cannot be represented exactly in the binary system and are rounded, you should use them with care in database tables. Especially avoid using them in WHERE clauses.

The number 1.0000259 has precision 8 and scale 7. The number 3.1415 requires at least precision 5 and scale 4 to be stored without loss of precision. Using DECIMAL (5, 4) it is stored as 3.1415, using DECIMAL (8, 7) as 3.1415000.



Table 19: String/Character Data Types

The following table lists the string-like SQL data types supported by SAP HANA.

| Data Type    | Remark                                                                                                          |
|--------------|-----------------------------------------------------------------------------------------------------------------|
| VARCHAR(n)   | Variable-length ASCII character string with maximum length n (n ≤ 5,000)                                        |
| NVARCHAR(n)  | Variable-length unicode character string with maximum length n (n ≤ 5,000)                                      |
| ALPHANUM(n)  | Variable-length alphanumeric character string with maximum length n (n ≤ 127)                                   |
| SHORTTEXT(n) | Variable-length unicode character string based on NVARCHAR(n) with support for text- and string-search features |



Table 20: Data Types for Date and Time

The following table lists the string-like SQL data types supported by SAP HANA.

| Data Type  | Remark                                                             |
|------------|--------------------------------------------------------------------|
| DATE       | Consists of year, month, day '2012-05-21'                          |
| TIME       | Consists of hour, minute, second '18:00:57'                        |
| SECONDDATE | Combination of data and time '2012-05-21 18:00:57'                 |
| TIMESTAMP  | Precision: ten millionth of a second '2012-05-21 18:00:57.1234567' |



Table 21: Data Types for Binary Data and Large Objects

The following table lists the string-like SQL data types supported by SAP HANA.

|               | Data Type    | Remark                                                                                                        |
|---------------|--------------|---------------------------------------------------------------------------------------------------------------|
| Binary Data   | VARBINARY(n) | Binary data, maximum length n Bytes (n ≤ 5,000)                                                               |
| Large Objects | BLOB         | Large binary data (maximum 2 GB)                                                                              |
|               | CLOB         | Long ASCII character string (maximum 2 GB)                                                                    |
|               | NCLOB        | Long unicode character string (maximum 2 GB)                                                                  |
|               | TEXT         | Long unicode character string (maximum 2 GB) based on NCLOB with support for text- and string-search features |

Data type VARBINARY can be used to store short values in binary form, for example UUIDs (Universally Unique Identifiers):

```
INSERT INTO MyTable VALUES( ... TO_BINARY('Walldorf') ...);
-- Value stored in DB: 57616C6C646F7266
INSERT INTO MyTable VALUES( ... TO_BINARY(x'00075341500700FF') ...);
-- x'..' is a hexadecimal literal value
```

- The term LOB data type (LOB = large object) is used as a generic term for data types such as CLOB (character large object) or BLOB (binary large object).
- When using LOB data types it is important to note the following:
  - LOB columns cannot be part of the primary key.
  - LOB columns cannot be used in the ORDER BY clause.
  - LOB columns cannot be used in the GROUP BY clause.
  - LOB columns may not be part of the JOIN condition (explicit JOIN).
  - LOB columns cannot be used as an argument for an aggregate function.
  - LOB columns cannot be used in the SELECT DISTINCT clause.
  - LOB columns cannot occur in a UNION statement.
  - LOB columns cannot be part of a database index.

## Creating Database Tables

In ANSI SQL, the statement to create new database tables is `CREATE TABLE ...`. SAP HANA supports two types of table stores: row store and column store. As a result, SAP HANA SQL extends the ANSI syntax to control whether a new table is created in the row store or in the column store. This leads to the following basic syntax:

### Basic CREATE TABLE syntax

```
CREATE [ROW|COLUMN] TABLE <Table Name> (
    <Column Name> <SQL Data Type> [PRIMARY KEY],
    <Column Name> <SQL Data Type> [NULL|NOT NULL],
    <Column Name> <SQL Data Type> [DEFAULT <Default Value>],
    [PRIMARY KEY (<column list>)] [UNIQUE (<column list>)]
)
```



#### Note:

You do not have to specify which table store to use by selecting the keywords `ROW` or `COLUMN`. The `ROW` store is the default for reasons of compatibility, but the `COLUMN` is the option recommended in most cases.

This is why the following examples all include the keyword `COLUMN`.



- Unless quoted, table and column names are treated as all uppercase, regardless of how they are typed.
- Case sensitivity requires the use of double quotes.
- We recommend leaving table and column names unquoted when creating.

```
CREATE COLUMN TABLE "Official"
("PNr"          VARCHAR(3),
 "Name"         VARCHAR(20),
 "Overtime"     INTEGER,
 "Salary"        VARCHAR(3),
 "Manager"       VARCHAR(3));
```



Figure 281: Create Column Table



You can define a table without a primary key.

If you do not define a key, the table can contain duplicates.

```
CREATE COLUMN TABLE Official
(PNr          VARCHAR(3),
 Name         VARCHAR(20),
 Overtime     INTEGER,
 Salary        VARCHAR(3),
 Manager       VARCHAR(3));
```



Figure 282: Defining a Table without a Primary Key



You can specify that no NULL values are allowed by adding **NOT NULL**

- For primary key columns, NULL values are implicitly prohibited.

You can define a default value for a column by adding **DEFAULT <value>**.

- The default value is used on **INSERT** if no value is specified for the column.

NULL values are not excluded when setting a default value. **NOT NULL** can also be useful for columns with a default value.

- The order of the keywords **DEFAULT** and **NOT NULL** is irrelevant.

```
CREATE COLUMN TABLE Official
  (PNr      VARCHAR(3),
   Name     VARCHAR(20) NOT NULL,
   Overtime INTEGER,
   Salary   VARCHAR(3) DEFAULT 'A06' NOT NULL,
   Manager  VARCHAR(3),
   PRIMARY KEY(PNr));
```

Figure 283: Default Values and Prohibiting NULL Values



You can define a primary key.

If a primary key is defined, the table may not contain duplicates in the key columns.

You can use the primary key definition as part of the column definition, if the primary key consists of only one column.

```
CREATE COLUMN TABLE Official
  (PNr      VARCHAR(3) PRIMARY KEY,
   Name     VARCHAR(20),
   Overtime INTEGER,
   Salary   VARCHAR(3),
   Manager  VARCHAR(3));
```

Figure 284: Define A Primary Key



You must use a separate **PRIMARY KEY** clause to define a multi-column primary key.

For all columns of the **PRIMARY KEY**, NULL values are implicitly prohibited.

```
CREATE COLUMN TABLE Owner_EU
  (Country   VARCHAR(3),
   OwnerID  VARCHAR(3),
   Name     VARCHAR(20),
   Birthday DATE,
   City     VARCHAR(20),
   PRIMARY KEY (Country, OwnerID));
```

Figure 285: Primary Key



You can specify that column(s) must not contain duplicate values by adding a **UNIQUE** constraint.

In contrast the the **PRIMARY KEY**, **UNIQUE** allows NULL values.

- Since NULL values do not equal each other, they are technically unique.

If a **UNIQUE** column also specifies **NOT NULL**, it is functionally equivalent to **PRIMARY KEY**.

Multiple **UNIQUEs** are allowed in one table, but **PRIMARY KEY** is limited to one.

#### **CREATE COLUMN TABLE Car**

```
(CarID      VARCHAR(3) PRIMARY KEY,
PlateNumber VARCHAR(10), UNIQUE,
Brand       VARCHAR(20),
Color       VARCHAR(10),
HP          INTEGER,
Owner       VARCHAR(3));
```



Figure 286: Unique Constraints



It is possible to prohibit duplicate values for more than one column.

- Using **UNIQUE** two single-column keys are defined:

#### **CREATE COLUMN TABLE Car**

```
(CarID      VARCHAR(3) PRIMARY KEY,
PlateNumber VARCHAR(10),
Brand       VARCHAR(20) UNIQUE,
Color       VARCHAR(10) UNIQUE,
HP          INTEGER,
Owner       VARCHAR(3));
```



Figure 287: Multiple Simple Uniqueness Constraints



You must use a separate **UNIQUE** clause to define a multi-column key.

- Using **UNIQUE** a two-column key is defined:

#### **CREATE COLUMN TABLE Car**

```
(CarID      VARCHAR(3) PRIMARY KEY,
PlateNumber VARCHAR(10),
Brand       VARCHAR(20),
Color       VARCHAR(10),
HP          INTEGER,
Owner       VARCHAR(3),
UNIQUE(Brand, Color));
```



Figure 288: Composite Uniqueness Constraints

## Changing Database Tables

Once a table has been defined, you can modify it in certain ways using the `ALTER TABLE` statement. The statement is always followed by the name of the table to be changed, and one of the following additional keyword to indicate the type of change:

- To add one or more columns, use `ADD`.
- To remove a column and its data, use `DROP`.
- To change the properties of columns such as their SQL data type and default value, use `ALTER`.



You can `ADD` columns to an existing table.

You can use `NOT NULL` only if the table is empty or if you define a `DEFAULT` value for the added columns.

Newly added columns are filled with `NULL` values or with the corresponding default value.

```
ALTER TABLE Official
  ADD (RemainderDays      INTEGER,
       AnnualLeave        INTEGER NOT NULL DEFAULT 30);
```



Figure 289: Adding Columns



You can `DROP` columns from an existing table.

You cannot drop columns that are part of the primary key.

Also for tables without a primary key at least one column must be left.

```
ALTER TABLE Official
  DROP (Salary,
        Manager,
        Name);
```



Figure 290: Removing Columns



**You can specify a default value for an existing column.**

- The changed default value only affects newly inserted rows.

**You can allow NULL values for columns for which they were prohibited so far.**

**You can change the data type of existing columns. Type compatibility must be ensured.**

**VARCHAR (20) → VARCHAR (25) is allowed, but not**

**VARCHAR(25) → VARCHAR(20)**

```
ALTER TABLE Official
    ALTER Overtime   INTEGER DEFAULT 7,
    ALTER Salary     VARCHAR(5) DEFAULT 'A01',
    ALTER Name       VARCHAR(25) Null;
```



Figure 291: Changing Column Properties

To remove a default value, use `ALTER TABLE ... ALTER` and set the default value to `NULL`.



**You can delete the limitations imposed by the primary key.**

**Only the primary key constraint is deleted.**

**The corresponding columns are not deleted.**

```
ALTER TABLE Official
    DROP PRIMARY KEY;
```



Figure 292: Removing the Primary Key



**You can subsequently define a primary key for a table without one.**

**The corresponding columns must not contain NULL values.**

**The uniqueness requirement must not be violated by the existing data.**

```
ALTER TABLE Official
    ADD PRIMARY KEY (PNr);
```



Figure 293: Adding a Primary Key

## Renaming a Column

You can also rename an existing table column. This is not achieved using an `ALTER TABLE` statement, but using a `RENAME COLUMN` statement.



**You can rename a column in a table.**

**The respective table can be empty or can contain data.**

**The new column name must be different from the old column name, and may not duplicate an existing column name in the same table.**

- **Column “PNr” or Table “Official” is renamed into “PersNumber”:**

```
RENAME COLUMN Official.PNr TO PersNumber;
```



Figure 294: Renaming a Column

## Renaming a Table

You can also rename a database table, using a `RENAME TABLE` statement.



**You can rename an existing table.**

**The table you want to rename can be empty or can contain data.**

**The new table name must be different from the old table name.**

```
RENAME TABLE Official TO Employee;
```



Figure 295: Renaming a Table

## Removing a Table

You can also remove an existing table, including its data.



**You can `DROP` an existing table.**

**The table you want to drop can be empty or can contain data.**

**Both the table content and the table itself will be deleted.**

```
DROP TABLE Official;
```



Figure 296: Removing a Table

## Row and Column Store Tables

SAP HANA has the ability to create different kinds of tables for different uses; for now, the only distinction that we will focus on is whether the table is **row store** or **column store**. The choice of row or column store will affect the structure of the table, the costs to query it, and the sorts of analysis that can be performed against it.

Row store tables are most appropriate when:

- The table has a limited number of columns and rows
- The table will not see frequent data change
- All of the columns of the table will be retrieved together
- Aggregation and complex analytics are avoided

Column store tables are most appropriate when:

- The table has a large number of columns and rows

- The table sees frequent data change
- A limited number of columns are retrieved at any one time
- Aggregation and complex analytics are performed
- High storage compression rates are desired

In general, a table that should have been column store but which was defined as row store is more damaging to performance than a table that should have been row store but which was defined as column store. It is better to assume all tables should be column store unless definite reasons can be presented to the contrary.

As shown at the start of this unit, to distinguish between the table types, add the word COLUMN or ROW between the words CREATE and TABLE in the create syntax. Note that if you omit the COLUMN or ROW qualifier, the default behavior is to create a row store table.



## LESSON SUMMARY

You should now be able to:

- List the most important data types SAP HANA supports
- Create new database tables in HANA
- Change tables by adding, removing or renaming columns

# Using Views for Data Access

## LESSON OVERVIEW

This lesson covers database views.



## LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Describe the use cases for and advantages of using database views, define database views and use them in queries

## Database Views — Overview

The following views are supported by SAP HANA:



- You can define database views using SQL **DDL Statements**.

**This is the only option covered in this course.**

- You can define **Calculation Views** or, more generally, information models using SAP HANA Tools.
- You can define views using **Core Data Services** and the HANA Development Perspective in the SAP HANA Tools.



### Note:

Calculation Views and Core Data Services may be more powerful than SQL views, or may simplify the development process.

Nevertheless, this course only covers SQL views. Understanding them is the foundation for understanding the other types of views and can be helpful for prototyping or analyzing issues.

## Database Views: Advantages

Database views provide the following advantages:



- De-couple applications from lower levels (relative to the three level schema architecture).
  - You can use database views to change to the conceptual schema (for example, add columns to a table) without causing side effects on applications.
- Create views tailored to the users (applications) and their needs.
  - The application programmers do not have to know the full database structure but only the excerpt that is relevant to them.
- Simplify queries.

- Complex queries referencing views are easier to formulate, provided that the view provides an appropriate pre-selection of data.
- Limit access to data.
  - You can use views to realize value-dependent access privileges or to make sure certain columns are never visible for certain users.



### Example of a view containing only red cars:

```
CREATE VIEW redCars AS
    SELECT CarID, Color, PlateNumber, HP AS Power
        FROM Car
    WHERE Color = 'red';
```

| CARID | COLOR | PLATENUMBER | POWER |
|-------|-------|-------------|-------|
| F01   | red   | HD-V 106    | 75    |
| F09   | red   | HD-UP 13    | 105   |
| F12   | red   | HD-XY 4711  | 105   |
| F13   | red   | HD-IK 1001  | 136   |
| F18   | red   | HD-MQ 2006  | 90    |



Figure 297: A Sample View: Red Cars



### You can select data from a view in the same way you select from a “normal” table.

- Which red cars have more than 100 HP?

```
SELECT CarID, PlateNumber, Power
    FROM redCars
    WHERE Power > 100
    ORDER BY Power DESC, CarID ASC;
```

| CARID | PLATENUMBER | POWER |
|-------|-------------|-------|
| F13   | HD-IK 1001  | 136   |
| F09   | HD-UP 13    | 105   |
| F12   | HD-XY 4711  | 105   |



Figure 298: Using Views in Queries

## Managing Database Views

You can create database views using the **CREATE VIEW** statement as follows:

```
CREATE VIEW <Name of the view> [<list of column names>] AS
    SELECT ...
```

You can use nearly any valid **SELECT** statement as the **SELECT** clause for a **CREATE VIEW** statement.



You can restrict a view to specific columns and rows of the underlying base table.

```
CREATE VIEW Audis AS
SELECT Brand, Color, HP
FROM Car
WHERE Brand = 'Audi' ;
```

| BRAND | COLOR  | HP  |
|-------|--------|-----|
| Audi  | yellow | 260 |
| Audi  | blue   | 116 |
| Audi  | orange | 184 |
| Audi  | green  | 184 |



Figure 299: CREATE VIEW

You can also use the following in the SELECT part of the CREATE VIEW statement:

- Calculations and functions
- Any type of join

The need to formulate the same complex join condition frequently in application development is a typical case where views can be used to simplify.

- A TOP N clause to limit the number of rows returned
- DISTINCT



Note:

A view can contain a projection leading to potential duplicates in result sets.

- Aggregate expressions, GROUP BY and HAVING clauses
- Sub queries, uncorrelated or correlated
- UNION
- An ORDER BY clause to achieve a default sort order of results



Note:

This order can be overridden when SELECTING from the view. For this reason, the use of ORDER BY is questionable and not recommended.



If you rename columns in both the CREATE VIEW clause and in the SELECT clause, the name of the CREATE VIEW clause “wins“.

We recommend renaming columns only in the SELECT clause.

```
CREATE VIEW Audis (Manufacturer, Color, Power) AS
    SELECT Brand AS Description, Color, HP AS
HorsePower
    FROM Car
    WHERE Brand = 'Audi' ;
```

| MANUFACTURER | COLOR  | POWER |
|--------------|--------|-------|
| Audi         | yellow | 260   |
| Audi         | blue   | 116   |
| Audi         | orange | 184   |
| Audi         | green  | 184   |



Figure 300: Precedence Rule for Column Naming

You can also define views based on another view.



You can define a view based on another view.

```
CREATE VIEW Volkswagen AS
    SELECT CarID, PlateNumber, Color, HP
        FROM Car
    WHERE Brand = 'VW' ;

CREATE VIEW blackVolkswagen AS
    SELECT CarID, PlateNumber, HP
        FROM Volkswagen
    WHERE Color = 'black' ;
```

| CARID | PLATENUMBER | HP  |
|-------|-------------|-----|
| F02   | HD-VW 4711  | 120 |
| F08   | HD-IK 1002  | 160 |
| F19   | HD-VW 2012  | 125 |



Figure 301: View on View



You can define a view based on a join.

```
CREATE VIEW OwnerInfo AS
    SELECT Name, Birthday, PlateNumber, Brand, Color
    FROM Owner JOIN Car
        ON Owner.OwnerID = Car.Owner
    WHERE Birthday = IS NOT NULL;
```

| NAME | BIRTHDAY     | PLATENUMER | BRAND | COLOR  |
|------|--------------|------------|-------|--------|
| Ms T | Jun 20, 1934 | HD-VW 2012 | VW    | black  |
| Ms U | May 11, 1966 | HD-UP 13   | Skoda | red    |
| Ms U | May 11, 1966 | HD-MM 208  | BMW   | green  |
| Mr V | Apr 21, 1952 | HD-VW 1999 | Audi  | yellow |
| Ms W | Jun 1, 1957  | HD-V 106   | Fiat  | red    |
| Ms Y | Feb 10, 1986 | HD-Y 333   | Audi  | orange |

Figure 302: Define View



- You can drop an existing view.
- Only the view definition is deleted.
- The data in the tables underlying the view is not deleted.

```
DROP VIEW redCars;
```

Figure 303: Removing a View

## Modifying Data with a View

In principle, an application developer should not be able to tell if a database table or a view is accessed when looking at an SQL statement. This largely holds true as far as SELECT statements / querying data is concerned.

To a certain extent, this statement even holds true when considering data modifications. Depending on how a database view is defined, you can insert, update or delete database records using the view.



You can **INSERT** new rows into a view.

- The new rows are not inserted into the view, but into the underlying base table!

```
INSERT INTO redCars
    VALUES ('F77', 'red', 'HD-MT 2509', 170);
SELECT * FROM Car;
```

| CARID | PLATENUMBER | BRAND   | COLOR | HP  | OWNER |
|-------|-------------|---------|-------|-----|-------|
| F01   | HD-V 106    | Fiat    | red   | 75  | H06   |
| F02   | HD-VW 4711  | VW      | black | 120 | H03   |
| F03   | HD-JA 1972  | BMW     | blue  | 184 | H03   |
| ...   | ...         | ...     | ...   | ... | ...   |
| F18   | HD-MQ 2006  | Renault | red   | 90  | H03   |
| F19   | HD-VW 2012  | VW      | black | 125 | H01   |
| F20   | ?           | Audi    | green | 184 | ?     |
| → F77 | HD-MT 2509  | ?       | red   | 170 | ?     |



Figure 304: Inserting through a View



You can **UPDATE** rows in a view.

- The rows are not changed in the view, but in the underlying base table!

```
UPDATE redCars
    SET Power = 120
    WHERE Power < 120;

SELECT * FROM Car;
```

| CARID | PLATENUMBER | BRAND   | COLOR | HP  | OWNER |
|-------|-------------|---------|-------|-----|-------|
| → F01 | HD-V 106    | Fiat    | red   | 120 | H06   |
| ...   | ...         | ...     | ...   | ... | ...   |
| F07   | HD-ML 3206  | Audi    | blue  | 116 | H03   |
| → F08 | HD-IK 1002  | VW      | black | 160 | H07   |
| → F09 | HD-UP 13    | Skoda   | red   | 120 | H02   |
| F10   | HD-MT 507   | BMW     | black | 140 | H04   |
| F11   | HD-MM 208   | BMW     | green | 184 | H02   |
| → F12 | HD-XY 4711  | Skoda   | red   | 120 | H04   |
| F13   | HD-IK 1001  | Renault | red   | 136 | H07   |
| ...   | ...         | ...     | ...   | ... | ...   |
| → F18 | HD-MQ 2006  | Renault | red   | 120 | H03   |
| ...   | ...         | ...     | ...   | ... | ...   |



Figure 305: Updating through a View



You can **DELETE** rows from a view.

- The rows are not deleted from the view, but from the underlying base table!

**DELETE**

```
FROM redCars
WHERE Power < 125;
```

**SELECT \* FROM Car;**

| CARID | PLATENUMBER | BRAND   | COLOR | HP  | OWNER |
|-------|-------------|---------|-------|-----|-------|
| F01   | HD-V 106    | Fiat    | red   | 120 | H06   |
| ...   | ...         | ...     | ...   | ... | ...   |
| F07   | HD-ML 3206  | Audi    | blue  | 116 | H03   |
| F08   | HD-IK 1002  | VW      | black | 160 | H07   |
| → F09 | HD-UP 13    | Skoda   | red   | 120 | H02   |
| F10   | HD-MT 507   | BMW     | black | 140 | H04   |
| F11   | HD-MM 208   | BMW     | green | 184 | H02   |
| → F12 | HD-XY 4711  | Skoda   | red   | 120 | H04   |
| F13   | HD-IK 1001  | Renault | red   | 136 | H07   |
| ...   | ...         | ...     | ...   | ... | ...   |
| → F18 | HD-MQ 2006  | Renault | red   | 120 | H03   |
| ...   | ...         | ...     | ...   | ... | ...   |



Figure 306: Deleting through a View

Such data modifications through database views are not always possible. You **cannot** modify data through a view if the view definition contains any of the following:

- A calculation or function in the projection list
- A sub query in the projection list
- DISTINCT
- The TOP N clause
- Aggregate expressions
- A GROUP BY clause
- UNION



## LESSON SUMMARY

You should now be able to:

- Describe the use cases for and advantages of using database views, define database views and use them in queries



## Defining Data Access

### LESSON OVERVIEW

In this lesson, you will learn how to manage data access permission on the SQL level, and how to speed up data access using database indexes.



### LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Understand database schemas and access tables in other schemas
- Explain when database indexes make sense in SAP HANA and create and delete indexes using SQL

### Database Schemas

Throughout this course you created database tables or views in exercises. You may wonder why this did not lead to naming collisions with other database users creating tables or views with the same names. You can consider the following situation:

- User 1 creates table “Car”.
- User 2 also creates table “Car”.

Why are there no naming collision?

The reason is that the name of the database object (table, view, and so on) implicitly contains a schema name as prefix: <schema name>. <given Name>. Thus the two Car tables are not the same: USER1 . Car ≠ USER2 . Car

### Features of Database Schemas

The following are features of database schemas:



- A **database schema** is the container that holds database objects such as tables or database views.
- Each database objects belongs to a schema.
- A database schema acts as namespace for the database objects it contains.
- A table or other objects' presence in a schema does not indicate ownership; the object is owned by its creator.

You can see a simple example of user schemas in the figure User Schemas.



**When creating a user an identically named schema is created implicitly.**  
**By default a user works in his own schema (and namespace).**  
**You can explicitly specify the schema name if necessary.**

- **Which brand has (at least) one black car?**

```
SELECT DISTINCT User1.Car.Brand
  FROM User1.Car
 WHERE User1.Car.Color = 'black'
ORDER BY User1.Car.Brand;
```

| BRAND    |
|----------|
| -----    |
| BMW      |
| Mercedes |
| Skoda    |
| VW       |



Figure 307: User Schemas

You can see an example of querying across schemas in the figure, Queries Across Schemas.



**Queries across multiple schemata are possible.**

- **To whom is (at least) one black car registered?**

```
SELECT DISTINCT Schema1.Owner.Name
  FROM Schema1.Owner, Schema2.Car
 WHERE Schema1.Owner.OwnerID = Schema2.Car.Owner
   AND Schema2.Car.Color = 'black';
```

| NAME   |
|--------|
| -----  |
| SAP AG |
| IKEA   |
| HDM AG |
| Ms T   |



Figure 308: Queries Across Schemas

## Data Access Control



### Note:

The SAP HANA platform is more than a relational database management system, so it provides a richer authorization model extending beyond the SQL level. This model includes, for example, analytic privileges, as well as application, development and administration permissions and roles. SAP HANA therefore also provides more advanced means to manage user permissions and roles than this course covers, for example using the SAP HANA Tools user and role management.

This course only covers the SQL level and how permissions can be managed using SQL statements. This can be helpful for prototyping, for example.

Relational database management systems, including SAP HANA, support the following basic ways for controlling which database user can access which data using SQL:

- Create database views that represent a portion of the data.
  - Example: You provide a view that contains only the owners of CityA to a user for whom the owners outside of CityA are not relevant.
- Grant specific access permissions to selected users using the Data Control Language of SQL.
  - Example: You grant read-only permissions to a user, who should be able to read the owner data, but not change it.

## DCL Statements

The following are the two DCL Statements and their basic syntax:



### **GRANT**

The GRANT statement is used to selectively grant access permissions to a user or role. Its basic syntax is:

```
GRANT <list of privileges> ON <database object> TO <user or role> [WITH GRANT OPTION].
```

### **REVOKE**

The REVOKE statement is used to selectively withdraw access permissions from a user or role. Its basic syntax is:

```
REVOKE <list of privileges> ON <database object> FROM <user or role>.
```

The figure, Example: Granting Read Permission, shows options for granting read permission.



**You can define which user is allowed to SELECT from which table.**

**By adding the WITH GRANT OPTION the recipient of the privilege (grantee) is allowed to grant this privilege to other users (without losing it).**

- User1 is allowed to read (SELECT) from table *Official* of schema *Schema2*.
- User1 is allowed to grant this SELECT privilege to other users.

```
GRANT SELECT
    ON Schema2.Official
    TO User1
    WITH GRANT OPTION;
```



Figure 309: Example: Granting Read Permission

The figure Granting Several Privileges in One Statement lists the privileges you can grant in a single statement, including SELECT, INSERT, UPDATE, and DELETE.



**You can assign multiple privileges for the same database object by using a single `GRANT` statement.**

- User1 is allowed to `SELECT, INSERT, UPDATE, DELETE` on table `Contact` of schema `Schema2`.
- User1 is allowed to grant these privileges to other users (in whole or in part).

```
GRANT SELECT, INSERT, UPDATE, DELETE
    ON Schema2.Contact
    TO User1
    WITH GRANT OPTION;
```

Figure 310: Granting Several Privileges in One Statement



Note:

There is no inherent dependency between privileges. For example, an `INSERT` privilege does not imply a `SELECT` or an `UPDATE` privilege.

The figure, Granting Access to all Objects of a Schema, shows how you can do this with a single `GRANT` statement.



**You can assign privileges for all tables and views of a schema by using a single `GRANT` statement.**

- User1 is allowed to `SELECT` from all tables and views of schema `Schema2`.
- User1 is allowed to grant this privilege to other users (for all or selected tables or views).

```
GRANT SELECT
    ON SCHEMA Schema2
    TO User1
    WITH GRANT OPTION;
```

Figure 311: Granting Access to all Objects of a Schema

Selectively revoking privileges can also be a simple matter, as shown in the figure, Revoking a Privilege.



**Using the `REVOKE` statement you can selectively revoke access privileges from a user.**

- User1 will no longer be allowed to read from table `Official` of schema `Schema2` (assuming that this privilege is not granted by other privileges)

```
REVOKE SELECT
    ON Schema2.Official
    FROM User1;
```

Figure 312: Revoking a Privilege

Just as you can grant multiple privileges with a single statement, you can also revoke them, as shown in the figure Revoking Multiple Privileges.



**You can revoke multiple privileges for the same database object by using a single REVOKE statement**

- User1 will no longer be allowed to **SELECT, INSERT, UPDATE, DELETE** from table **Contact** of schema **Schema2** (assuming that these privileges are not granted by other privileges)

```
REVOKE SELECT, INSERT, UPDATE, DELETE
  ON Schema2.Contact
  FROM User1;
```

Figure 313: Revoking Multiple Privileges

The figure, No Hierarchy of Privileges, is a reminder that access privileges operate independently of each other.



**Access privileges on different hierarchical levels of database objects are independent.**

- Despite the **REVOKE** statement User1 still has read access to the table **Official** of schema **Schema2**, because he still has a (not withdrawn) read permission on the entire schema **Schema2**.

```
GRANT SELECT
  ON SCHEMA Schema2
  TO User1;

REVOKE SELECT
  ON Schema2.Official
  FROM User1;
```

Figure 314: No Hierarchy of Privileges

## Database Indexes

A database index has the following properties:



- It can be considered an access path that helps the DBMS locate rows faster.
- It can speed up searching and sorting as follows:
  - It reduces the need for “full table scans” or “full column scans”.
  - It reduces the need for non-unique indexes in SAP HANA to exceptional cases only.
- It has no influence on the result of queries, only on how fast the result is returned.
- It can slow down INSERTS, UPDATES and DELETES (especially non-unique indexes).
- Coming up with a good balance of indexes is an iterative process.

The figure, SAP HANA: Why Indexes on Column Store, gives an example of when you would use indexes on a column store.



- SAP HANA column store tables store keep column values together, not row values
- Columns are stored as
  - Sorted dictionary of values
  - (Bit-)Vector of value IDs

**Are indexes necessary to speed up data access?**

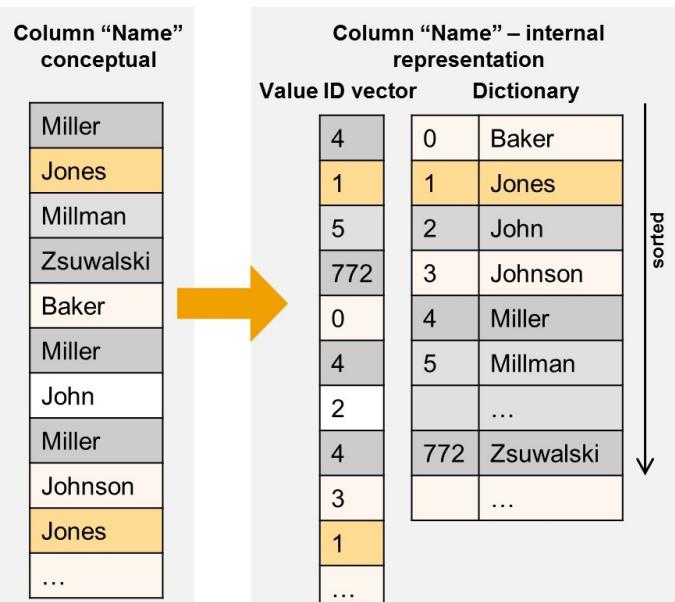


Figure 315: SAP HANA: Why Indexes on Column Store?

The figure, SAP HANA: Table Scan with Value ID, gives an example of this type of scan.



- Value ID lookup is fast
  - binary search on sorted dictionary
- Row ID lookup can still involve “full column scan”
  - No issue for most value ID vectors, even less if sorted
  - But may be slow for very large value ID vectors

**WHERE Name = 'Zsuwalski'**

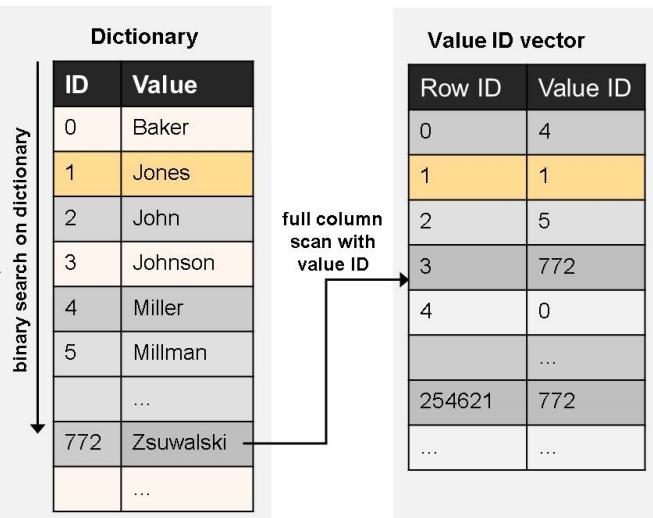


Figure 316: SAP HANA: Table Scan with Value ID

The figure, SAP HANA: Inverted Index on Column Store, gives an example of this type of index.



- An inverted index can be reasonable in exceptional cases if column scan performance is not sufficient

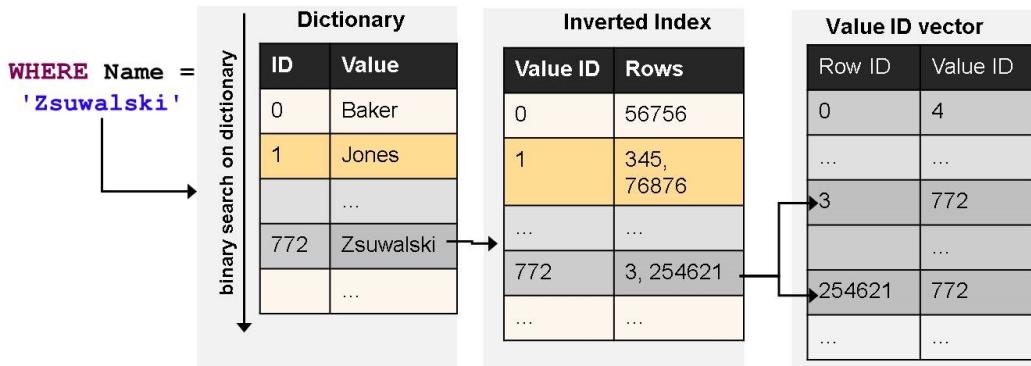


Figure 317: SAP HANA: Inverted Index on Column Store

## Database Index Management

The syntax of the statements you use to manage indexes in SQL is as follows:



### CREATE INDEX

This statement is used to create an index. Its basic syntax is:

```
CREATE [UNIQUE] INDEX <index name> ON <table name> ( <list of columns> ).
```

### DROP INDEX

This statement is used to delete an index. The underlying database table and its data are not deleted. The basic syntax is:

```
DROP INDEX <index name>.
```

You can also create an index on a table column, as shown in the figure, Example: Creating a Non-Unique Index.



To speed up the read access you can create an index on a table column.  
The respective table can be empty or contain data.

- An index on column *PlateNumber* of table *Car* will be created:

```
CREATE INDEX PlateNumberIndex  
ON Car (PlateNumber);
```

Figure 318: Example: Creating a Non-Unique Index



You can create a **UNIQUE INDEX** to ensure that the corresponding column cannot contain duplicate values.

Unlike **PRIMARY KEY**, **UNIQUE INDEX** does not prohibit NULL values.

A **UNIQUE INDEX** can only be created when the column contains no duplicate values.

- In table **Car** no duplicated plate numbers are allowed:

```
CREATE UNIQUE INDEX PlateNumberIndex
    ON Car (PlateNumber);
```

Figure 319: Example: Creating a Unique Index

As the figure, Example: Creating a Unique Index, shows creating a unique index is equivalent to adding a **UNIQUE** constraint to the definition of the database table. Each such constraint automatically leads to a unique index. In addition, a unique index is created automatically for the primary key.

### Index Creation Options

You can also create indexes as follows:



- You can create multiple indexes on the same table.
- You can create multi-column indexes, provided you observe the following guidelines:
  - Multi-column non-unique indexes are not recommended.
  - The memory cost of multi-column indexes is substantial.
  - Creating a non-unique index on the most important column of a multi-column set is sufficient to achieve good performance.



You can drop an existing index.

Only the access path (index) is deleted.

The underlying data of the index is not deleted.

**DROP INDEX** does not syntactically distinguish between **NON-UNIQUE** and **UNIQUE** indexes.

```
DROP INDEX PlateNumberIndex;
```

Figure 320: DROP INDEX

The figure, DROP INDEX, shows the syntax and guidelines for this statement.



### LESSON SUMMARY

You should now be able to:

- Understand database schemas and access tables in other schemas
- Explain when database indexes make sense in SAP HANA and create and delete indexes using SQL

# Explaining Database Transactions

## LESSON OVERVIEW

This lesson covers the necessity for database transactions and how SAP HANA treats them.



## LESSON OBJECTIVES

After completing this lesson, you will be able to:

- Explain database transactions and the ACID requirements
- Finish database transactions in SAP HANA using SQL statements
- Describe issues that arise if transactions are not mutually isolated
- Understand and control isolation levels of transactions and how SAP HANA handles concurrency

## Database Transactions

One of the strengths of relational database management systems (DBMSs) is their support for database transactions. The figure, The Purpose of Database Transactions, lists the purpose of these transactions.



- **Transaction** = Sequence of associated database operations (SQL statements) for which the ACID requirements must be met.
  - All database operations always take place within transactions (in the extreme case, each operation has its own transaction).
- 
- **A = Atomicity**
  - **C = Consistency**
  - **I = Isolation**
  - **D = Durability**

Figure 321: The Purpose of Database Transactions

The figure The Necessity of Transactions shows one practical application of database transactions.



### Assume you are moving money between bank accounts

#### UPDATE SAVINGS

```
SET BALANCE = BALANCE - 200  
WHERE ACCOUNT = <accountnumber>;
```

#### UPDATE CHECKING

```
SET BALANCE = BALANCE + 200  
WHERE ACCOUNT = <accountnumber>;
```

If a power failure occurs after the first UPDATE and before the second, what has become of your money?



Figure 322: The Necessity of Transactions

The figure, Database Transactions Multiple Statements, shows how database transactions help to prevent inconsistencies.

The figure Database Transaction ACID Definition lists the requirements for database transactions.

### Transaction Starts and Ends

The following rules apply to starting a transaction in SAP HANA:

#### Starting a Transaction in SAP HANA



- SAP HANA does not provide an SQL statement to explicitly start a transaction.
- A transaction is started implicitly with the first DML statement, before execution of the statement begins.

#### Ending a Transaction in SAP HANA

A transaction in SAP HANA can be ended with the following statements:

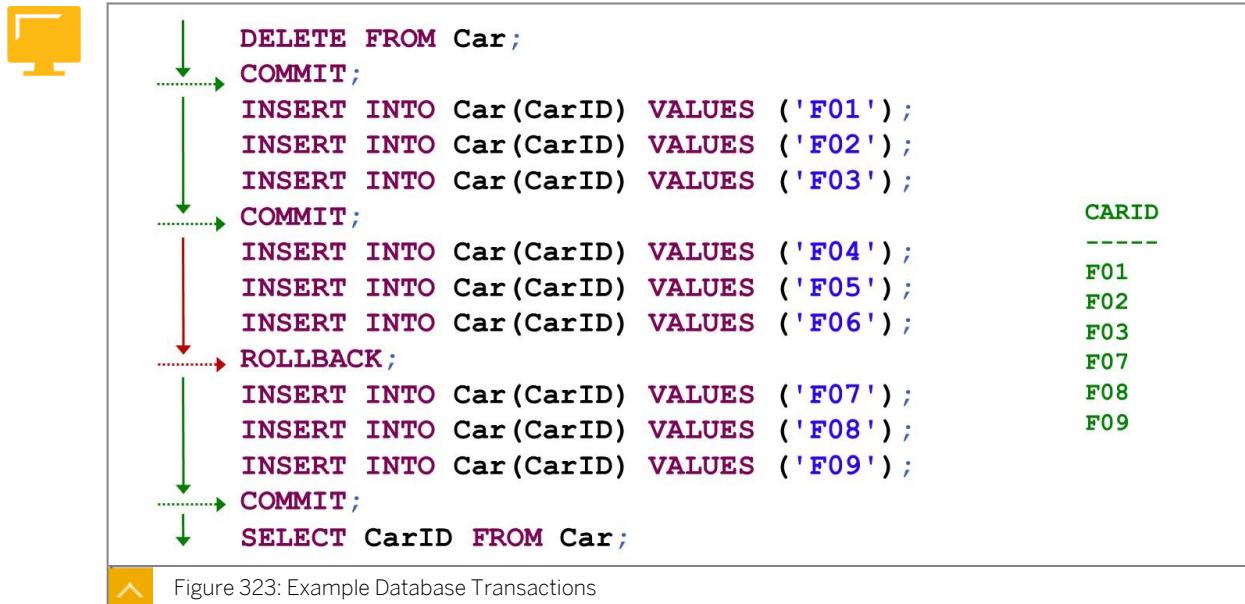
##### COMMIT [WORK]

This statement ends a transaction successfully, making modifications performed during the transaction durable.

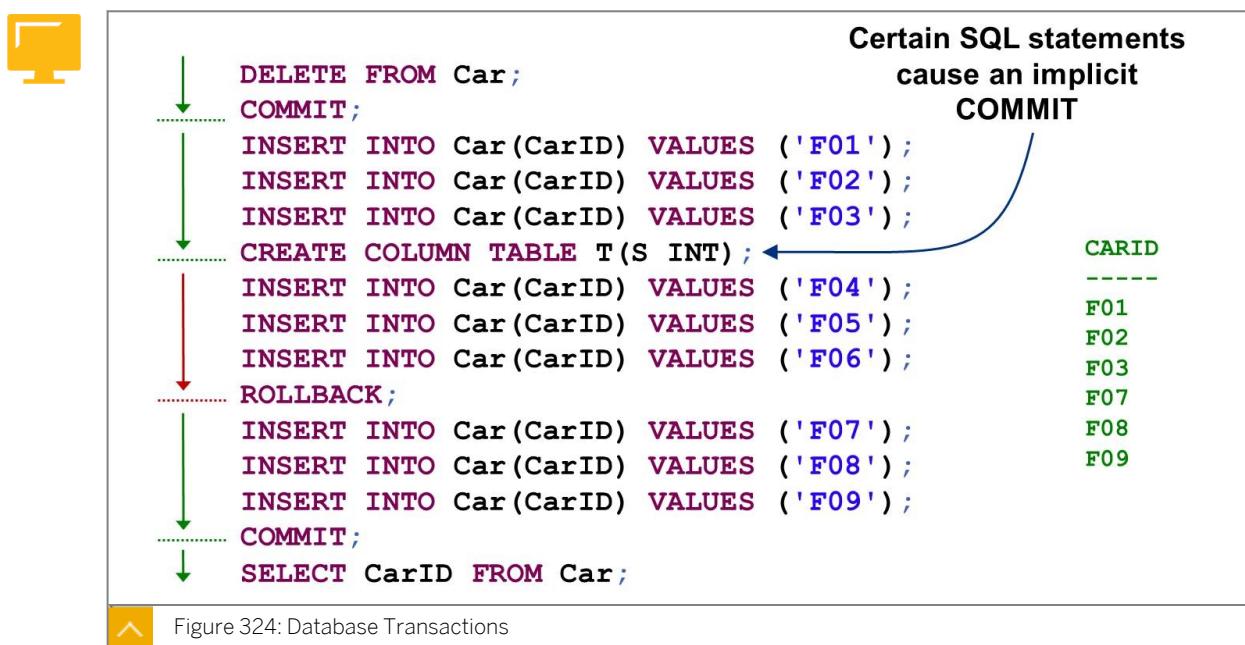
##### ROLLBACK [WORK]

This statement ends a transaction by discarding all modifications performed during the transaction. A rollback cancels a transaction.

The figure Example Database Transactions shows how you use COMMIT and ROLLBACK statements.



As the figure Database Transactions shows, Data Definition Language (DDL) statements automatically end the transaction they belong to successfully, creating an implicit COMMIT.



## Isolation Challenges

The isolation requirement becomes relevant when considering concurrent access to a database by several users or several sessions.

Support for such multi-user mode is absolutely necessary. Executing all transactions strictly in sequence would be a huge waste of resources, for example when waiting for user input or the applications using the DBMS. Furthermore, a strict serialization of all transactions would not be accepted by the end users.

Therefore, several transactions must run time-shared. This makes the isolation requirement important, because without certain control mechanisms, problems can occur, as illustrated in the following Isolation Challenge figures.

The figure Isolation Challenge 1: Inconsistent Reads illustrates the problem of this apparent inconsistency.



### Inconsistent Analyses

**A transaction sees an apparent inconsistency, because it refers to two different consistent database states.**

| Point in time | Transaction 1                                         | Transaction 2                                |
|---------------|-------------------------------------------------------|----------------------------------------------|
| t1            | Read marital status of Mandy<br>(Result: "unmarried") |                                              |
| t2            |                                                       | Change marital status of Mandy to "married"  |
| t3            |                                                       | Change marital status of Thomas to "married" |
| t4            | Read marital status of Thomas<br>(Result: "married")  |                                              |

Figure 325: Isolation Challenge 1: Inconsistent Reads

The figure Isolation Challenge 2: Lost Updates, shows how changes can be lost when they are overwritten.



### Lost-update Problem

**The changes of a transaction are overwritten by another transaction and therefore lost.**

| Point in time | Transaction 1   | Transaction 2 |
|---------------|-----------------|---------------|
| t1            | Read x          |               |
| t2            |                 | Read x        |
| t3            | x := x + 5,000; |               |
| t4            |                 | x := x - 100; |
| t5            | Write x         |               |
| t6            |                 | Write x       |

Figure 326: Isolation Challenge 2: Lost Updates

A phantom read occurs when a transaction sees a new row as an apparent inconsistency, as illustrated by the figure Isolation Challenge 3: Phantom Reads.



### Phantom Problem (“malicious”, because it cannot be prevented by row-locking)

A transaction sees an apparent inconsistency, since another transaction inserted a new row in the meantime.

| Point in time | Transaction 1                                                                                                                    | Transaction 2                       |
|---------------|----------------------------------------------------------------------------------------------------------------------------------|-------------------------------------|
| t1            | Read all SAP employees                                                                                                           |                                     |
| t2            |                                                                                                                                  | Insert Mr Z as a new SAP employee   |
| t3            |                                                                                                                                  | Register Mr Z as course participant |
| t4            | Read all course participants<br>(Mr Z is a participant of an internal course, though he apparently seems to be no SAP employee!) |                                     |

Figure 327: Isolation Challenge 3: Phantom Reads

A non-repeatable read occurs when a repeated reading of the same facts in the same transaction gives different results, as shown in the figure Isolation Challenge 4: Non-Repeatable Reads.



### Non-repeatable read

During the same transaction different results are obtained for repeated reading of the same facts.

| Point in time | Transaction 1       | Transaction 2        |
|---------------|---------------------|----------------------|
| t1            | Read salary of Mr A |                      |
| t2            |                     | Raise salary of Mr A |
| t3            | Read salary of Mr A |                      |

Figure 328: Isolation Challenge 4: Non-Repeatable Reads

The figure Isolation Challenge 5: Dirty Reads shows the result of a transaction seeing and using a state that never officially existed.



### Dependency on uncommitted changes (Dirty reads)

A transaction sees and uses a state that officially never existed.

| Point in time | Transaction 1                       | Transaction 2        |
|---------------|-------------------------------------|----------------------|
| t1            | Raise salary to € 10,000,000.-      |                      |
| t2            |                                     | Read salary          |
| t3            |                                     | Donate € 5,000,000.- |
| t4            |                                     | COMMIT;              |
| t5            | ROLLBACK;<br>(transaction canceled) |                      |



Figure 329: Isolation Challenge 5: Dirty Reads

Different applications can tolerate different degrees of inconsistency. It may be cheaper to allow a dirty read to occur and then correct any bad decisions than to wait for a transaction to finish and then be absolutely certain of a value. In contrast, at times you may need to prioritize complete confidence in the correctness of a value read, even if that means delaying other processing or spending extra resources.

Improving one necessarily means reducing the other.

### Isolation Levels and MVCC

DBMSs, including SAP HANA, allow you to control the extent to which problems are avoided for each transaction by supporting different isolation levels. Isolation levels define whether rows that are read, inserted, updated, or deleted by a transaction can be accessed by concurrent transactions and whether the current transaction should be able to access rows that were read, inserted, updated, or deleted by other transactions.

In summary, isolation levels control which types of problems that come from the multi-user mode are allowed and which ones are prevented. The isolation levels defined by the SQL standard are listed in the figure Isolation Levels.



**4 isolation levels are defined by the SQL Standard in such a way that each level defines which problems may occur and which problems must be prevented by the DBMS.**

| Isolation Level |                  | Uncommitted Dependency (Dirty Reads) | Non-repeatable Reads | Phantom Problem |
|-----------------|------------------|--------------------------------------|----------------------|-----------------|
| 0               | READ UNCOMMITTED | possible                             | possible             | possible        |
| 1               | READ COMMITTED   | impossible                           | possible             | possible        |
| 2               | REPEATABLE READ  | impossible                           | impossible           | possible        |
| 3               | SERIALIZABLE     | impossible                           | impossible           | impossible      |



Figure 330: Isolation Levels



Note:

SAP HANA does not support all four levels of isolation defined in the SQL standard.

Typical DBMSs support a single version of each persistent data record only and use both read and write locks (set automatically by the DBMS) to implement the isolation levels. The isolation level then impacts the granularity of read and write locks.

### Consistency of Read Access in SAP HANA

In contrast, SAP HANA implements Multi-Version Concurrency Control (MVCC) and works with snapshot isolation. This means that consistency of write access is ensured using exclusive write locks, as follows:



- No shared (read) locks are necessary.
- A `SELECT` statement sees a consistent version of the database state: a snapshot.
  - By default, this is the version valid immediately before the statement execution: statement level snapshot isolation.  
This roughly corresponds to isolation level `REPEATABLE READ`.
  - This can be changed to the version that was valid when the transaction to which the statement belongs began: transaction level snapshot isolation.  
This roughly corresponds to isolation level `READ COMMITTED`.
- Write access creates a new version of the database state, which is not visible to other transactions that started already.

### Controlling the Isolation Level

The following statements apply to controlling the isolation level:



- The higher the isolation level, the more types of problems are prevented.
- The higher the isolation level, the higher the burden on the system, and the less concurrency.
- The isolation level can be controlled as follows:

```
SET TRANSACTION ISOLATION LEVEL
```

The figure, Isolation Levels Supported by SAP HANA, lists the levels that SAP HANA supports:



According to the SQL Standard isolation level **SERIALIZABLE** is the default.

In SAP HANA the isolation level **READ COMMITTED** is the default.

It is not possible to set isolation level **READ UNCOMMITTED** in SAP HANA.

- You can set the remaining three isolation levels as follows:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Figure 331: Isolation Levels Supported by SAP HANA

You can see an example of MVCC at work in the figure, MVCC – An Example.



Two transactions act on a value in a table, one after the other, at statement level isolation:

- T1 changes the value from 3 to 6
- T2 changes the value from 6 to 4

Three queries retrieve the value from the table:

- At time t1, Q1 sees the committed value 3.
- At time t2, Q2 sees the committed value 6. Q1 still sees the value 3 – the version for Q1 locked down on first read, and Q1 will continue to see the value 3 until its transaction ends.
- At time t3, Q3 sees the committed value 4, but Q1 and Q2 remain at their earlier versions, locked at first read (3 and 6, respectively).

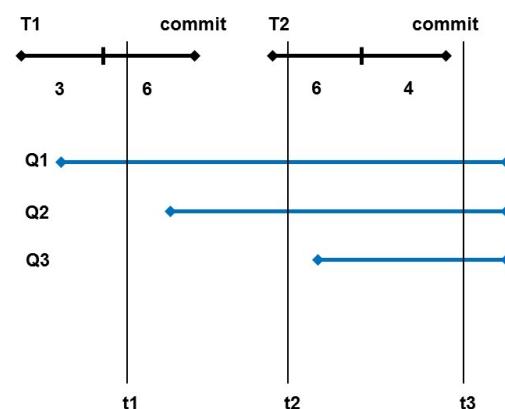


Figure 332: MVCC - An Example

In the figure, MVCC Transaction Level Isolation, you can see how this isolation level affects the handling of queries.



**Two transactions act on a value in a table, one after the other, at transaction level isolation:**

- T1 changes the value from 3 to 6
- T2 changes the value from 6 to 4

**Three queries retrieve the value from the table:**

- At time t1, Q1 sees the committed value 3.
- At time t2, Q2 also sees the committed value 3.
- At time t3, Q3 sees the committed value 6.
- At the higher isolation level, the version is established not at the time of first read, but rather at the time of transaction start.
- As before, once locked, the values remain the same until transaction end.

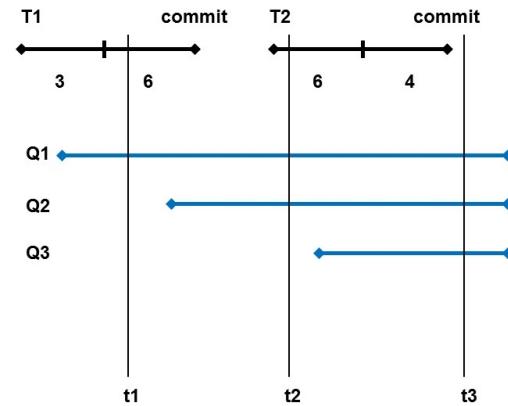


Figure 333: MVCC Transaction Level Isolation

There is an important implication in version-based transaction control: multiple users can issue the same query in the same database at the same time and receive different results. Each of these users is receiving a correct answer, although this answer differs from user to user. This is because each user is looking at a different version of the underlying table representing a different point in time.



## LESSON SUMMARY

You should now be able to:

- Explain database transactions and the ACID requirements
- Finish database transactions in SAP HANA using SQL statements
- Describe issues that arise if transactions are not mutually isolated
- Understand and control isolation levels of transactions and how SAP HANA handles concurrency