



SAP HANA™ Database – Development Guide Beta Preview

– How to Use SQL and SQLScript for Data Modeling

■ SAP HANA™ Appliance Software

Target Audience

- Consultants
- Application Programmers

Disclaimer

This document is a preview version. No statement in this document is meant to be an official statement by SAP AG be it about any of SAP's products, strategy, or future development. The information in this document may not be complete or correct. SAP makes no assumptions, gives no warranty and accepts no liability as to the usefulness of this document for your business purposes or any damages created by making use of it. Use of the documentation is subject to you having a license for your use of SAP HANA and is subject to its terms governing your use of SAP HANA.

Document version 1.0 beta – 10/21/2011



© Copyright 2011 SAP AG. 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 AG. The information contained herein may be changed without prior notice.

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

Microsoft, Windows, Outlook, and PowerPoint are registered trademarks of Microsoft Corporation.

IBM, DB2, DB2 Universal Database, System i, System i5, System p, System p5, System x, System z, System z10, System z9, z10, z9, iSeries, pSeries, xSeries, zSeries, eServer, z/VM, z/OS, i5/OS, S/390, OS/390, OS/400, AS/400, S/390 Parallel Enterprise Server, PowerVM, Power Architecture, POWER6+, POWER6, POWER5+, POWER5, POWER, OpenPower, PowerPC, BatchPipes, BladeCenter, System Storage, GPFS, HACMP, RETAIN, DB2 Connect, RACF, Redbooks, OS/2, Parallel Sysplex, MVS/ESA, AIX, Intelligent Miner, WebSphere, Netfinity, Tivoli and Informix are trademarks or registered trademarks of IBM Corporation.

Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.

Adobe, the Adobe logo, Acrobat, PostScript, and Reader are either trademarks or registered trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Oracle is a registered trademark of Oracle Corporation.

UNIX, X/Open, OSF/1, and Motif are registered trademarks of the Open Group.

Citrix, ICA, Program Neighborhood, MetaFrame, WinFrame, VideoFrame, and MultiWin are trademarks or registered trademarks of Citrix Systems, Inc.

HTML, XML, XHTML and W3C are trademarks or registered trademarks of W3C®, World Wide Web Consortium, Massachusetts Institute of Technology.

Java is a registered trademark of Sun Microsystems, Inc.

JavaScript is a registered trademark of Sun Microsystems, Inc., used under license for technology invented and implemented by Netscape.

SAP, R/3, SAP NetWeaver, Duet, PartnerEdge, ByDesign, SAP BusinessObjects Explorer, and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP AG in Germany and other countries.

Business Objects and the Business Objects logo, BusinessObjects, Crystal Reports, Crystal Decisions, Web Intelligence, Xcelsius, and other Business Objects products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of Business Objects Software Ltd. in the United States and in other countries.

Sybase and Adaptive Server, iAnywhere, Sybase 365, SQL Anywhere, and other Sybase products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of Sybase, Inc. Sybase is an SAP company.

All other product and service names mentioned are the trademarks of their respective companies. Data contained in this document serves informational purposes only. National product specifications may vary.

These materials are subject to change without notice. These materials are provided by SAP AG and its affiliated companies ("SAP Group") for informational purposes only, without representation or warranty of any kind, and SAP Group shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP Group 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.

Disclaimer

Some components of this product are based on Java™. Any code change in these components may cause unpredictable and severe malfunctions and is therefore expressly prohibited, as is any decompilation of these components.

Any Java™ Source Code delivered with this product is only to be used by SAP's Support Services and may not be modified or altered in any way.

Documentation in the SAP Service Marketplace

You can find this documentation at the following address:
<http://service.sap.com/hana>

Terms for Included Open Source Software

This SAP software contains also the third party open source software products listed below. Please note that for these third party products the following special terms and conditions shall apply.

1. This software was developed using ANTLR.
2. gSOAP

Part of the software embedded in this product is gSOAP software. Portions created by gSOAP are Copyright (C) 2001-2004 Robert A. van Engelen, Genivia inc. All Rights Reserved.

THE SOFTWARE IN THIS PRODUCT WAS IN PART PROVIDED BY GENIVIA INC AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

3. SAP License Agreement for STLport SAP License Agreement for STLPort between SAP Aktiengesellschaft Systems, Applications, Products in Data Processing Neurtstrasse 16 69190 Walldorf, Germany (hereinafter: SAP) and you (hereinafter: Customer)

a) Subject Matter of the Agreement

A) SAP grants Customer a non-exclusive, non-transferrable, royalty-free license to use the STLport.org C++ library (STLport) and its documentation without fee.

B) By downloading, using, or copying STLport or any portion thereof Customer agrees to abide by the intellectual property laws, and to all of the terms and conditions of this Agreement.

C) The Customer may distribute binaries compiled with STLport (whether original or modified) without any royalties or restrictions.

D) Customer shall maintain the following copyright and permissions notices on STLport sources and its documentation unchanged:

Copyright 2001 SAP AG

E) The Customer may distribute original or modified STLport sources, provided that:

- o The conditions indicated in the above permissions notice are met;
- o The following copyright notices are retained when present, and conditions provided in accompanying permission notices are met:

Copyright 1994 Hewlett-Packard Company

Copyright 1996,97 Silicon Graphics Computer Systems Inc.

Copyright 1997 Moscow Center for SPARC Technology.

Copyright 1999,2000 Boris Fomitchev

Copyright 2001 SAP AG

Permission to use, copy, modify, distribute and sell this software and its documentation for any purposes is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Hewlett-Packard Company makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Silicon Graphics makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

Permission to use, copy, modify, distribute and sell this software and its documentation for any purposes is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Moscow Center for SPARC makes no

representations about the suitability of this software for any purpose. It is provided “as is” without express or implied warranty.

Boris Fomitchev makes no representations about the suitability of this software for any purpose. This material is provided "as is", with absolutely no warranty expressed or implied.

Any use is at your own risk. Permission to use or copy this software for any purpose is hereby granted without fee, provided the above notices are retained on all copies.

Permission to modify the code and to distribute modified code is granted, provided the above notices are retained, and a notice that the code was modified is included with the above copyright notice.

Permission to use, copy, modify, distribute and sell this software and its documentation for any purposes is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. SAP makes no representations about the suitability of this software for any purpose. It is provided with a limited warranty and liability as set forth in the License Agreement distributed with this copy.

SAP offers this liability and warranty obligations only towards its customers and only referring to its modifications.

b) Support and Maintenance SAP does not provide software maintenance for the STLport. Software maintenance of the STLport therefore shall be not included.

All other services shall be charged according to the rates for services quoted in the SAP List of Prices and Conditions and shall be subject to a separate contract.

c) Exclusion of warranty

As the STLport is transferred to the Customer on a loan basis and free of charge, SAP cannot guarantee that the STLport is error-free, without material defects or suitable for a specific application under third-party rights. Technical data, sales brochures, advertising text and quality descriptions produced by SAP do not indicate any assurance of particular attributes.

d) Limited Liability

A) Irrespective of the legal reasons, SAP shall only be liable for damage, including unauthorized operation, if this (i) can be compensated under the Product Liability Act or (ii) if caused due to gross negligence or intent by SAP or (iii) if based on the failure of a guaranteed attribute.

B) If SAP is liable for gross negligence or intent caused by employees who are neither agents or managerial employees of SAP, the total

liability for such damage and a maximum limit on the scope of any such damage shall depend on the extent to which its occurrence ought to have anticipated by SAP when concluding the contract, due to the circumstances known to it at that point in time representing a typical transfer of the software.

C) In the case of Art. 4.2 above, SAP shall not be liable for indirect damage, consequential damage caused by a defect or lost profit.

D) SAP and the Customer agree that the typical foreseeable extent of damage shall under no circumstances exceed EUR 5,000.

E) The Customer shall take adequate measures for the protection of data and programs, in particular by making backup copies at the minimum intervals recommended by SAP. SAP shall not be liable for the loss of data and its recovery, notwithstanding the other limitations of the present Art. 4 if this loss could have been avoided by observing this obligation.

F) The exclusion or the limitation of claims in accordance with the present Art. 4 includes claims against employees or agents of SAP.

4. Adobe Document Services Adobe, the Adobe logo, Acrobat, PostScript, and Reader are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and / or other countries. For information on Third Party software delivered with Adobe document services and Adobe LiveCycle Designer, see SAP Note 854621

SAP HANA Database – Development Guide

– How to Use SQL and SQLScript for Data Modeling

TABLE OF CONTENTS

| | |
|--|----|
| 1 Introduction..... | 7 |
| 1.1 What is SAP HANA? | 7 |
| 1.2 Related Documentation | 8 |
| 2 SAP HANA Database Concepts | 9 |
| 2.1 Basic Concepts | 9 |
| 2.1.1 Impact of Modern Hardware on Database System Architecture | 9 |
| 2.1.2 Columnar and Row-Based Data Storage | 10 |
| 2.1.3 Advantages of Columnar Tables | 12 |
| 2.2 Architecture Overview | 15 |
| 2.3 SAP HANA Database Concepts: Tables, Models and View Processing | 17 |
| 2.3.1 Tables, Views, and Star Schemas..... | 17 |
| 2.3.2 SAP HANA Modeling Views | 18 |
| 2.3.3 SAP HANA View Processing | 19 |
| 3 Tutorials..... | 24 |
| 3.1 Using the SAP HANA Studio..... | 24 |
| 3.2 How to Use the SAP HANA Information Modeler | 31 |
| 3.2.1 How to Create Analytic Views | 32 |
| 3.2.2 Inspecting the Tables..... | 32 |
| 3.2.3 Creating Analytic Views | 34 |
| 3.2.4 MANDT Attributes | 37 |
| 3.2.5 Applying Filters | 38 |
| 3.2.6 Creating Attribute Views | 38 |
| 3.2.7 Creating a simple Star Schema | 40 |
| 3.2.8 Calculated Attributes..... | 41 |
| 3.2.9 Graphical Calculation Views | 43 |
| 3.2.10 How to Create a Scripted Calculation View..... | 46 |
| 3.3 Using SAP HANA Studio to Execute SQL and SQLScript Statements | 49 |

| | |
|--|----|
| 3.4 How to Display the Query Plan..... | 52 |
| 3.4.1 Columns in the Query Plan | 54 |
| 3.4.2 OPERATOR_NAME Column in Query Plans | 55 |
| 3.5 Using the JDBC Driver | 56 |
| 3.5.1 Installing the Driver | 57 |
| 3.5.2 Prerequisites..... | 57 |
| 3.5.3 Integration of the JDBC Driver..... | 57 |
| 3.5.4 Loading the JDBC Driver | 57 |
| 3.5.5 Connection URL | 57 |
| 3.5.6 JDBC 4.0 Standard Extension API..... | 58 |
| 3.5.7 JDBC Trace | 58 |
| 3.5.8 Mapping SQL and Java Types | 60 |
| 4 Best Practices | 61 |
| 4.1 Features of the Column Storage Engine..... | 61 |
| 4.2 SQL Query Cost Estimation..... | 63 |
| 4.2.1 Row Search Cost Models | 63 |
| 4.2.2 Column Search Cost Models..... | 64 |
| 4.3 SQL Query Tuning Tips for the Column Engine | 64 |
| 4.3.1 Expressions | 65 |
| 4.3.2 Join..... | 66 |
| 4.3.3 EXISTS / IN Predicate | 69 |
| 4.3.4 Set Operations | 70 |
| 4.4 SQLScript Recommended Practices..... | 71 |
| 4.4.1 Reduce Complexity of SQL Statements | 71 |
| 4.4.2 Identify Common Sub-Expressions..... | 71 |
| 4.4.3 Multi-level Aggregation..... | 71 |
| 4.4.4 Understand the Costs of Statements | 72 |
| 4.4.5 Exploit Underlying Engine | 73 |
| 4.4.6 Reduce Dependencies | 74 |
| 4.4.7 Simulating Function Calls in SQL Statements | 74 |
| 4.4.8 Avoid Mixing Calculation Engine Plan Operators and SQL Queries | 74 |
| 4.4.9 Avoid Using Cursors..... | 75 |
| 4.4.10 Avoid using Dynamic SQL | 76 |
| 4.4.11 Tracing and Debugging..... | 77 |

1 Introduction

1.1 What is SAP HANA?

SAP HANA is an exciting new technology brought to you by SAP. At its core it uses an innovative in-memory technique to store your data that is particularly suited for handling very large amounts of tabular, or relational, data with unprecedented performance. Common databases store tabular data row-wise, e. g. all data describing an address are stored adjacent to each other in memory. As long as your requirements are to access a single address your application will run quickly as all the required data is stored contiguously. However, consider the situation that your application requires a count of how many of the addresses stored relate to a particular country, city or ZIP code? In this case it would have to scan through the full table, select each row, and check for the country or city that is required. As all mass-storage devices, e. g. hard disks, access stored data in full blocks that can be quite large in comparison to the data of interest, e. g. 512 Bytes for a hard disk, it is likely that a mass-storage device has to read one or more rows just to check for a couple of characters, e. g. "Brazil" or "San Francisco". Tables of business data often contain many data fields, or columns, which are rarely used, e. g. data relating to other tables, or data fields controlling how other fields are to be used. Can you imagine the increase in efficiency if your application could just read around those unwanted fields and access the information that is really required?

If this style of data storage were used you would experience a significantly faster response from your database or application. SAP HANA allows you to read around unwanted data by organizing tables in this efficient columnar manner. In addition to the common row-oriented storage schema a column-oriented data storage layout can be used. This means your application does not have to wait for the database to fetch data that it does not need as all the data in a table column is stored in an adjacent manner. So, in our example of an address table, scanning through the country or city column is much faster than with row-oriented storage layout.

But what if your database system already caches all data in RAM, in fast accessible main memory close to the CPU? Would a column-oriented memory layout still speed access up? Measurements conducted at SAP and at the Hasso Plattner Institute in Potsdam have proven that reorganizing the data in memory column-wise brings a tremendous speed increase when accessing a subset of the data in each table row. As SAP HANA caches all data in memory, hard disks are rarely used in the system they are only needed to record changes to the database for permanent persistency. SAP HANA keeps the number of changes to your dataset as small as possible by recording every change as delta to your original dataset. Data is not modified in place but inserted or appended to a table column. This provides several advantages, not just speed of access. As all of the old data is retained, your applications can effectively "time-travel" through data providing views of the data as it has changed over time.

Contemporary database applications separate data management and applications in two distinct architectural layers, the database and application layer. This separation forces data to travel from the database to the application before it can be analyzed or modified. Sometimes very large amounts of data have to travel from one layer to another. SAP HANA avoids this common bottleneck by locating data intensive application logic to where the data is, which is in the database itself. To enable this embedding of application logic in the database SAP has invented an extension to standard SQL (Structured Query Language) named SQLScript. SQLScript allows programming of data intensive

operations in a way such that they can be executed in the database layer. SQLScript allows you to extend SQL queries to contain high level calculations thereby extending the data processing capabilities of the database.

This document explains how you can make use of SQLScript to achieve efficient data intensive processing in the SAP HANA database.

1.2 Related Documentation

There are other related documents which tell the tiny details of the programmers' tools and the programming languages used. They include the following:

- [SAP HANA Database – Administration Guide](#)
– How to use the SAP HANA studio and how to administrate the SAP HANA database.
- [SAP HANA Modeling Guide](#)
– How to use the HANA Modeler to create OLAP analytic views and calculation views which are based on SQLScript programs.
- [SAP HANA Database – SQL Reference Guide](#)
– Full reference to the query language used in SAP HANA.
- [SAP HANA Database – SQLScript Guide](#)
– A tutorial about how to use SQLScript to program and use stored procedures in SAP HANA including its use from ABAP programs.

2 SAP HANA Database Concepts

The SAP HANA database conceptually is about increasing speed, increasing execution speed of database queries via the use of in-memory data storage, and also increasing speed of application development. With the SAP HANA database, queries can be executed rapidly and in parallel. This means that complex coding techniques, e. g. pre-calculation of values (materialized aggregates), that were required by traditional databases to maintain performance are no-longer needed. This is because you can use the HDB to query huge datasets in real-time. When this developmental complexity is removed, applications can be created and modified in a straight forward and clear manner, thus enabling faster development times.

SAP HANA is also very competent at more traditional database data storage and access, e. g. row-oriented tables are also available. This combination of both classical and innovative technologies allows the developer to choose the best technology for their application and, where necessary, use both in parallel.

2.1 Basic Concepts

2.1.1 Impact of Modern Hardware on Database System Architecture

Historically database systems were designed to perform well on computer systems with limited RAM, this had the effect that slow disk I/O was the main bottleneck in data throughput. Consequently the architecture of those systems was designed with a focus on optimizing disk access, e. g. by minimizing the number of disk blocks (or pages) to be read into main memory when processing a query.

Computer architecture has changed in recent years. Now multi-core CPUs (multiple CPUs on one chip or in one package) are standard, with fast communication between processor cores enabling parallel processing. Main memory is no-longer a limited resource, modern servers can have 2TB of system memory and this allows complete databases to be held in RAM. Currently server processors have up to 64 cores, and 128 cores will soon be available. With the increasing number of cores, CPUs are able to process increased data per time interval. This shifts the performance bottleneck from disk I/O to the data transfer between CPU cache and main memory (see figure. 1).

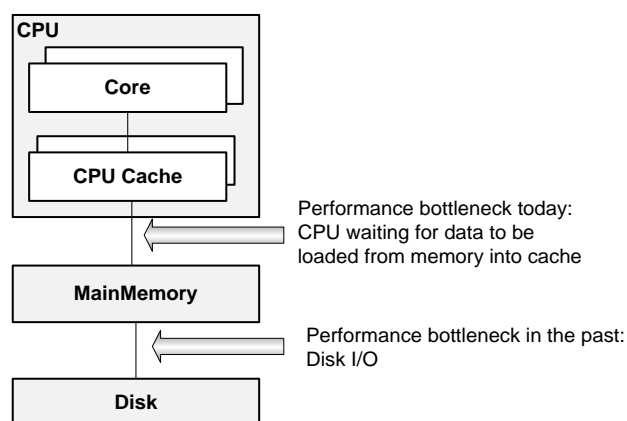


Figure 1: Hardware architecture: Current and past performance bottlenecks.

Traditional databases for online transaction processing (OLTP) do not use current hardware efficiently. It was shown in 1999 by Alamaki et al. that when databases have all data loaded into main

memory the CPU spends half of its execution time in stalls, i. e. waiting for data to be loaded from main memory into the CPU cache.

So, what are the ideal characteristics of a database system running on modern hardware?

In-memory database. All relevant data is available in main memory. This characteristic avoids the performance penalty of disk I/O. With all data in memory, techniques to reduce disk I/O like disk-based indexes are no longer needed. Disk storage is still required for permanent persistency, for example in the event of a power failure. This is not a performance issue however, as the required disk write operations can happen asynchronously as a background task.

Cache-aware memory organization. The design must minimize the number of cache misses and avoid CPU stalls because of memory access. A general mechanism to achieve this is to maximize the spatial locality of data, i. e. data that should be accessed consecutively should be stored contiguously in memory. For example search operations in tabular data can be accelerated by organizing data in columns instead in rows.

Support for parallel execution. Higher CPU execution speeds are nowadays achieved by adding more cores to a CPU package. Earlier improvements resulted from applying higher packing densities on the chip and optimizing electronic current paths. The speed advancements available using these techniques have, for the most part, been exhausted. Multiple CPUs call for new parallel algorithms to be used in databases in order to fully utilize the computing resources available.

2.1.2 Columnar and Row-Based Data Storage

As mentioned above columnar storage organization in certain situations leads to fewer cache misses and therefore less CPU stalls. This is particularly useful when the CPU needs to scan through a full column. For example this happens when a query is executed that cannot be satisfied by index searches or when aggregate functions are to be calculated on the column, e. g. the sum or average.

Indexed rows are the classical means to speed up row-based table access. This works well for column stores, but index trees have a low spatial locality and therefore increase cache misses significantly. In addition to this, indexes have to be reorganized whenever data is inserted into a table. Therefore database programmers have to understand the advantages and disadvantages of both storage techniques in order to find a suitable balance.

Conceptually, a database table is a two-dimensional data structure with cells organized in rows and columns. Computer memory however is organized as a linear structure. To store a table in linear memory, two options exist as shown in figure 2. A row-oriented storage stores a table as a sequence of records, each of which contain the fields of one row. Conversely, in a column store the entries of a column are stored in contiguous memory locations.

| Table | | | Row Store | | Column Store | | |
|---------|---------|-------|-----------|-------|--------------|---------|-------|
| Country | Product | Sales | | | | | |
| US | Alpha | 3.000 | Row 1 | US | Country | US | |
| | | | | Alpha | | US | |
| | | | | 3.000 | | | |
| US | Beta | 1.250 | | US | | JP | |
| JP | Alpha | 700 | Row 2 | Beta | | UK | |
| | | | | 1.250 | | Product | Alpha |
| | | | | JP | | | Beta |
| | | | | | Alpha | | |
| | | | Row 3 | Alpha | | Alpha | |
| | | | | 700 | | | 3.000 |
| | | | | UK | | | 1.250 |
| | | | Row 4 | Alpha | Sales | 700 | |
| | | | | 450 | | | 450 |
| | | | | | | | |

Figure 2: Row and column-based storage

The concept of columnar data storage has been used for quite some time. Historically it was mainly used for analytics and data warehousing where aggregate functions play an important role. Using column stores in OLTP applications requires a balanced approach to insertion and indexing of column data to minimize cache misses.

The SAP HANA database allows the developer to specify whether a table is to be stored column-wise or row-wise. It is also possible to alter an existing table from columnar to row-based and vice versa. See the *SAP HANA SQL Reference* for details.

Column-based tables have advantages in the following circumstances:

- Calculations are typically executed on single or a few columns only.
- The table is searched based on values of a few columns.
- The table has a large number of columns.
- The table has a large number of rows and columnar operations are required (aggregate, scan, etc.).
- High compression rates can be achieved because the majority of the columns contain only few distinct values (compared to number of rows).

Row based tables have advantages in the following circumstances:

- The application needs to only process a single record at one time (many selects and/or updates of single records).
- The application typically needs to access a complete record (or row).
- The columns contain mainly distinct values so that the compression rate would be low.
- Neither aggregations nor fast searching are required.
- The table has a small number of rows (e. g. configuration tables).

To enable fast on-the-fly aggregations, ad-hoc reporting, and to benefit from compression mechanisms it is recommended that transaction data is stored in a column-based table. The SAP HANA database allows joining row-based tables with column-based tables. However, it is more efficient to join tables that are located in the same row or column store. For example, master data that is frequently joined with transaction data should also be stored in column-based tables.

2.1.3 Advantages of Columnar Tables

If the criteria listed above for columnar tables are fulfilled, column-based tables have several advantages that are explained in this section. This should not lead to the impression that column based tables are always the better choice. There are also situations in which row based tables are advantageous.

Advantage: Higher Data Compression Rates

The goal of keeping all relevant data in main memory can be achieved with less cost if data compression is used. Columnar data storage allows highly efficient compression. Especially if a column is sorted, there will normally be several contiguous values placed adjacent to each other in memory. In this case compression methods, such as run-length encoding, cluster coding or dictionary coding can be used. This is especially promising for business applications as many of the table columns contain only a few distinct values compared to the number of rows. Extreme examples are codes such as country codes or ZIP codes, other valid examples include customer and account numbers, region codes, sales channel codes, or status codes. Many of the latter are used as foreign keys in other tables in the database, e. g. tables containing data for customer orders and accounting records. This high degree of redundancy within a column allows for effective compression of the data. In row-based storage, successive memory locations contain data of different columns, so compression methods such as run-length encoding cannot be used. In column stores a compression factor of 10 can typically be achieved compared to traditional row-oriented storage systems.

Compression techniques discussed briefly below:

Run-length encoding. If data in a column is sorted there is a high probability that two or more elements contain the same values. Run-length encoding counts the number of consecutive column elements with the same values. To achieve this, the original column is replaced with a two-column list. The first column contains the values as they appear in the original table column and the second column contains the counts of consecutive occurrences of the respective value. From this information the original column can easily be reconstructed.

Cluster encoding. This compression technique works by searching for multiple occurrences of the same sequence of values within the original column. The compressed column consists of a two-column list with the first column containing the elements of a particular sequence and the second column containing the row numbers where the sequence starts in the original column. Many popular data compression programs use this technique to compress text files.

Dictionary encoding. Table columns which contain only a comparably small number of distinct values can be effectively be compressed by enumerating the distinct values and storing only their numbers. This technique requires that an additional table, the dictionary, is maintained which in the first column contains the original values and in the second one the numbers representing the values. This technique leads to high compression rates, is very common, e. g. in country codes or customer numbers, but is seldom regarded as a compression technique.

Advantage: Higher Performance for Column Operations

With columnar data organization operations on single columns, such as searching or aggregations, can be implemented as loops over an array stored in contiguous memory locations. Such an operation has high spatial locality and can efficiently be executed in the CPU cache. With row-oriented

storage, the same operation would be much slower because data of the same column is distributed across memory and the CPU is slowed down by cache misses.

Assume that we want to aggregate the sum of all sales amounts in the example in figure 2 using a row-based table. Data transfer from main memory into CPU cache happens always in blocks of fixed size called "cache lines" (for example 64 bytes). With row-based data organization it may happen that each cache line contains only one "sales" value (stored in 4 bytes) while the remaining bytes are used for the other fields of the data record. For each value needed for the aggregation a new access to main memory would be required. This demonstrates that with row-based data organization the operation will be slowed by cache misses that cause the CPU to wait until the required data is available. With column-based storage, all sales values are stored in contiguous memory, so the cache line would contain 16 values that are all needed in the calculation of the sum. In addition, the fact that columns are stored in contiguous memory allows memory controllers to pre-fetch, which further minimizes the number of cache misses.

As explained above, columnar data organization also allows highly efficient data compression. This not only saves memory but also increases speed for the following reasons:

- Compressed data can be loaded into the CPU cache faster. This is because the limiting factor is the data transport between memory and CPU cache, and so the performance gain will exceed the additional computing time needed for decompression.
- With dictionary encoding, the columns are stored as sequences of bit-coded integers. That means that a check for equality can be executed on the integers (for example during scans or join operations). This is much faster than comparing, for example, string values.
- Compression can speed up operations such as scans and aggregations if the operator is aware of the compression. Given a good compression rate, computing the sum of the values in a column will be much faster if the column is run length coded and many additions of the same value can be replaced by a single multiplication. A recent scientific study reported a performance gain of factor 100 to 1000 when comparing operations on integer coded compressed values with operations on uncompressed data.

Advantage: Elimination of Additional Indexes

Columnar storage, in many cases, eliminates the need for additional index structures. Storing data in columns is functionally similar to having a built-in index for each column. The column scanning speed of the in-memory column store and the compression mechanisms – especially dictionary compression – allow read operations with very high performance. In many cases it will not be required to have additional indexes. Eliminating additional indexes reduces complexity and eliminates effort for defining and maintaining metadata.

Advantage: Parallelization

Column-based storage also makes it easy to execute operations in parallel using multiple processor cores. In a column store data is already vertically partitioned. This means that operations on different columns can easily be processed in parallel. If multiple columns need to be searched or aggregated, each of these operations can be assigned to a different processor core. In addition operations on one column can be parallelized by partitioning the column into multiple sections that can be processed by different processor cores.

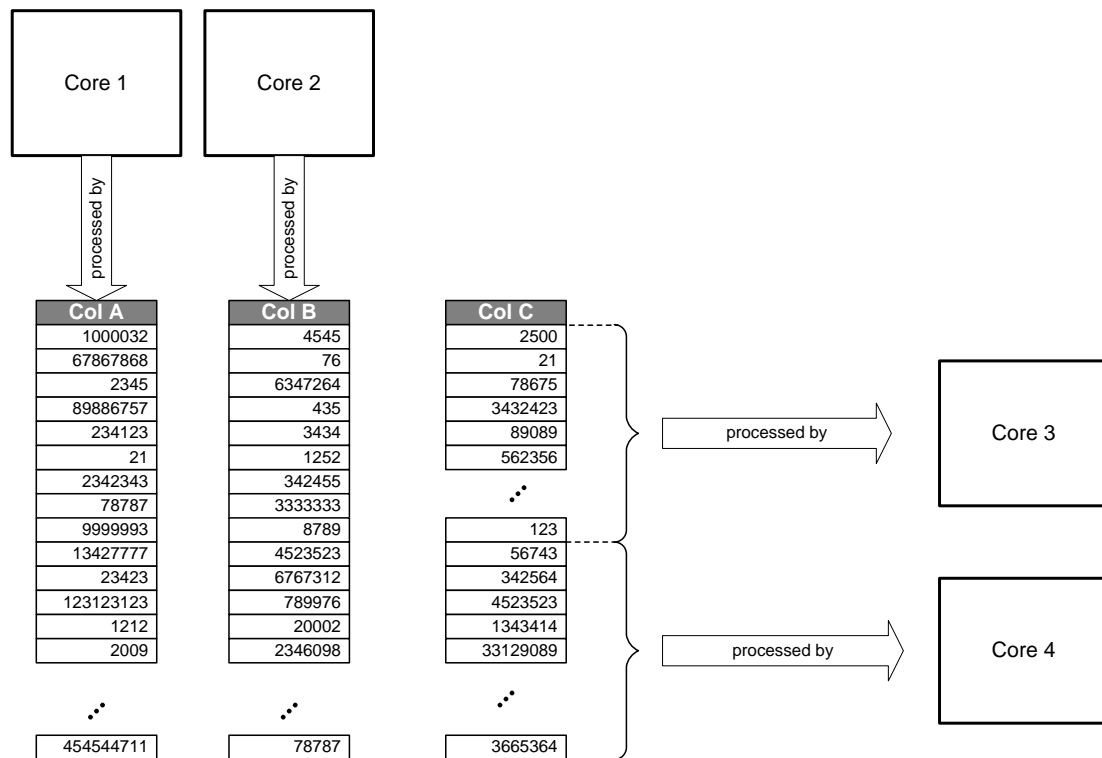


Figure 3: Example for parallelization in a column store

Advantage: Elimination of Materialized Aggregates

Traditional business applications use materialized aggregates to increase read performance. This means that the application developers define additional tables in which the application redundantly stores the results of aggregates (for example sums) computed on other tables. The materialized aggregates are computed and stored either after each write operation on the aggregated data, or at scheduled times. Read operations read the materialized aggregates instead of computing them each time they are required.

With a scanning speed of several megabytes per millisecond, in-memory column stores make it possible to calculate aggregates on large amounts of data on the fly with high performance. This is expected to eliminate the need for materialized aggregates in many cases.

In financial applications, different kinds of totals and balances are typically persisted for the different ledgers as materialized aggregates. For example general ledger, accounts payable, accounts receivable, cash ledger, material ledger, etc. With an in-memory column store, these materialized aggregates can be eliminated as all totals and balances can be computed on the fly from accounting document items with high performance.

Eliminating materialized aggregates has several advantages:

Simplified data model. With materialized aggregates, additional tables are needed which make the data model more complex. In SAP Business ByDesign financials application, for example, persisted totals and balances are stored with a star schema (see section 2.3.1). Specific business objects are introduced for totals and balances, each of which is persisted with one fact table and several dimen-

sion tables. All these tables can be removed if totals and balances are computed on the fly. A simplified data model makes development more efficient, removes sources of programming errors and increases maintainability.

Simplified application logic. The application either needs to update the aggregated value after an aggregated item was added, deleted or modified. Alternatively special aggregation runs need to be scheduled that update the aggregated values at certain time intervals (for example once a day). By eliminating persisted aggregates, this additional logic is no longer required.

Higher level of concurrency. With materialized aggregates after each write operation a write lock needs to be acquired for updating also the aggregate. This limits concurrency and may lead to performance issues. Without materialized aggregates only the document items are written, and this can be done concurrently without any locking.

Contemporaneousness of aggregated values. With on-the fly aggregation, the aggregate values are always up-to-date while materialized aggregates are sometimes updated only at scheduled times.

2.2 Architecture Overview

The block diagram in figure 4 (see below) gives an overview of the conceptual architecture of the SAP HANA database. At the top there is the *connection and session management* component which creates and manages sessions and connections for the database clients. For each session a set of parameters is maintained such as, for example, auto-commit settings or the current transaction isolation level. Once a session is established, database clients typically use SQL statements to communicate with the SAP HANA database.

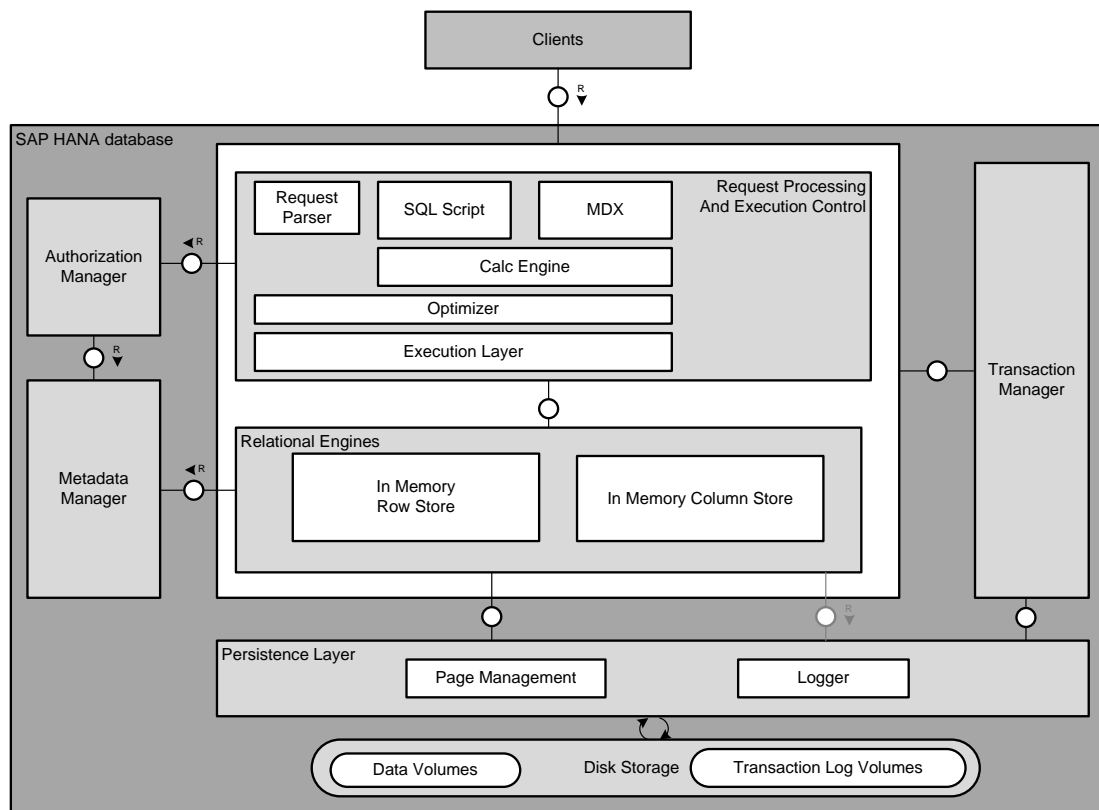


Figure 4: SAP HANA Database High Level Architecture

In the SAP HANA database, each statement is processed in the context of a transaction. New sessions are implicitly assigned to a new transaction. The *transaction manager* is the component that coordinates database transactions, controls transactional isolation and keeps track of running and closed transactions. When a transaction is committed or rolled back, the transaction manager informs the involved engines about this event so they can execute necessary actions. The transaction manager cooperates with the persistence layer to achieve atomic and durable transactions.

The client requests are analyzed and executed by the set of components summarized as *Request Processing And Execution Control*. Figure 4 shows the main functions of this layer: The request parser analyses the client request and dispatches it to the responsible component. Transaction control statements are, for example, forwarded to the transaction manager. Data definition statements are dispatched to the metadata manager and object invocations are forwarded to object store. Data manipulation statements are forwarded to the optimizer which creates an optimized execution plan that is given to the execution layer. The execution layer acts as the controller that invokes the different engines and routes intermediate results to the next execution step.

The SAP HANA database also has built-in support for domain-specific models (such as for financial planning) and it offers scripting capabilities that allow application-specific calculations to run inside the database. The SAP HANA database has its own scripting language named *SQLScript* that is designed to enable optimizations and parallelization. SQLScript is based on side effect free functions that operate on tables using SQL queries for set processing.

The SAP HANA database also contains a component called the *planning engine* that allows financial planning applications to execute basic planning operations in the database layer. One such basic operation is to create a new version of a dataset as a copy of an existing one while applying filters and transformations. For example: Planning data for a new year is created as a copy of the data from the previous year. This requires filtering by year and updating the time dimension. Another example for a planning operation is the disaggregation operation that distributes target values from higher to lower aggregation levels based on a distribution function.

The SAP HANA database features such as SQLScript and planning operations are implemented using a common infrastructure called the *calc engine*.

Metadata can be accessed via the *metadata manager*. The SAP HANA database metadata comprises of a variety of objects, such as definitions of relational tables, columns, views, and indexes, definitions of SQLScript functions and object store metadata. Metadata of all these types is stored in one common catalog for all SAP HANA database stores (in-memory row store, in-memory column store, object store, disk-based). Meta-data is stored in tables in row store. The SAP HANA database features described in the following chapters such as transaction support, multi-version concurrency control, are also used for metadata management. In distributed database systems central metadata is shared across servers. How metadata is actually stored and shared is hidden from the components that use the meta-data manager.

The *column store* is the SAP HANA database column-based in-memory data engine. Parts of it originate from TREX (SAP NetWeaver Search and Classification). For the SAP HANA database this proven technology was further developed into a full relational column-based data store. In addition the TREX

text analysis and search capabilities are still available, even though they are not covered in this document.

As row-based tables and columnar tables can be combined in one SQL statement, the corresponding engines must be able to consume intermediate results created by each other. A main difference between the two engines is the way they process data: Row store operators process data in a row-at-a-time fashion using iterators. Column store operations (such as scan, aggregate and so on) require that the entire column is available in contiguous memory locations. To exchange intermediate results, row store can provide results to column store materialized as complete rows in memory while column store can expose results using the iterator interface needed by row store.

The *persistence layer* is responsible for durability and atomicity of transactions. It ensures that the database is restored to the most recent committed state after a restart and that transactions are either completely executed or completely undone. To achieve this goal in an efficient way the persistence layer uses a combination of write-ahead logs, shadow paging and savepoints. The persistence layer offers interfaces for writing and reading data. It also contains SAP HANA's logger that manages the transaction log. Log entries can be written implicitly by the persistence layer when data is written via the persistence interface or explicitly by using a log interface.

The *authorization manager* is invoked by other SAP HANA database components to check whether the user has the required privileges to execute the requested operations. SAP HANA allows granting of privileges to users or roles. A privilege grants the right to perform a specified operation (such as create, update, select, execute, and so on) on a specified object (for example a table, view, SQLScript function, and so on). The SAP HANA database supports analytic privileges that represent filters or hierarchy drilldown limitations for analytic queries. Analytic privileges grant access to values with a certain combination of dimension attributes. This could, for example, be used to restrict access to a cube with sales data to values with dimension attributes of region = 'US' and year = '2010'. As analytic privileges are defined on dimension attribute values and not on metadata, they are evaluated dynamically during query execution.

Users are authenticated either by the SAP HANA database itself (login with user and password) or authentication can be delegated to an external authentication providers such as an LDAP directory.

2.3 SAP HANA Database Concepts: Tables, Models and View Processing

SAP HANA database is a very capable database system but it requires some understanding, and to be used correctly, to obtain good performance. In this section we will explore some key concepts of the SAP HANA database. Also, an analysis of how you can approach modeling to achieve good results is presented.

2.3.1 Tables, Views, and Star Schemas

The SAP HANA database allows you to model your data as tables and views. Tables are tabular data structures, each row identifying a particular entity, and each column having a unique name. The data fields of one row are called the *attributes* of the entity. The word “attribute” is used with different meanings in this document. It may denote a table column, a particular data field of a table row, or the contents of such a data field. The respective meaning will be clear from the context.

Views are combinations and selections of data from tables modeled to serve a particular purpose. Views always appear like readable tables, i. e. database operations which read from tables can also be used to read data from views. For instance, you can create views which simply select some columns from a table, or views which select some columns and some rows according to some filter pattern. SAP HANA distinguishes several types of views.

In analytics applications *star schemas* are a general pattern: *Fact tables* are lists of business transactions while the linked-in data are typically master data. These linked-in tables are often called *dimension tables*. A fact table surrounded by its linked-in dimension tables is often called a *star schema* because of the geometry of its graphical model. In SAP HANA a star schema can be created using tables or views by surrounding an analytic or calculation view with attribute views.

2.3.2 SAP HANA Modeling Views

Before we continue to discuss how to model in SAP HANA, first it is important to understand each of the differing SAP HANA modeling view types and their capabilities.

Attribute Views

Attribute views are used to give master data tables context. This context is provided by text tables which give meaning to the master data. For example, if our fact table or analytic view only contains some numeric ID for each car dealer then we can link in information about each dealer using an attribute view. We could then display the dealers' names and addresses instead of their IDs thus providing the context for the master data table.

Attribute views are used to select a subset of columns and rows from a data table. As it is of little use to sum up attributes from master data tables there is no need to define measures or aggregates for attribute views.

You can also use attribute views to join master data tables to each other, e. g. joining "Plant" to "Material".

Analytic Views

Analytic views are used to build a data foundation based on transactional tables. You can create a selection of measures (sometimes referred to as key figures), add attributes and join attribute views.

Analytic views leverage the computing power of SAP HANA to calculate aggregate data, e. g. the number of sold cars per country, or the maximum power consumption per day. They are defined on at least one *fact table*, i. e. a table which contains e. g. one row per sold car or one row per power meter reading, or more generally speaking, some form of business transaction records. Fact tables can be joined to allow access to more detailed data using a single analytic view. Analytic views can be defined on a single table, or joined tables.

Analytic views can contain two types of attributes (or columns), so-called *measures* and *normal attributes*. Measures are attributes for which an aggregation must be defined. If analytic views are used in SQL statements (see below) then the measures have to be aggregated e. g. using the SQL functions SUM(<column name>), MIN(<column name>), or MAX(<column name>). Normal attributes can be handled as regular columns. For them there is no need to be aggregated.

Calculation Views

Calculation views are used to provide composites of other views. They are essentially a view which is based on the result of an SQLScript. These scripts can join or union two or more data flows or invoke built in Calculation engine or generic SQL functions.

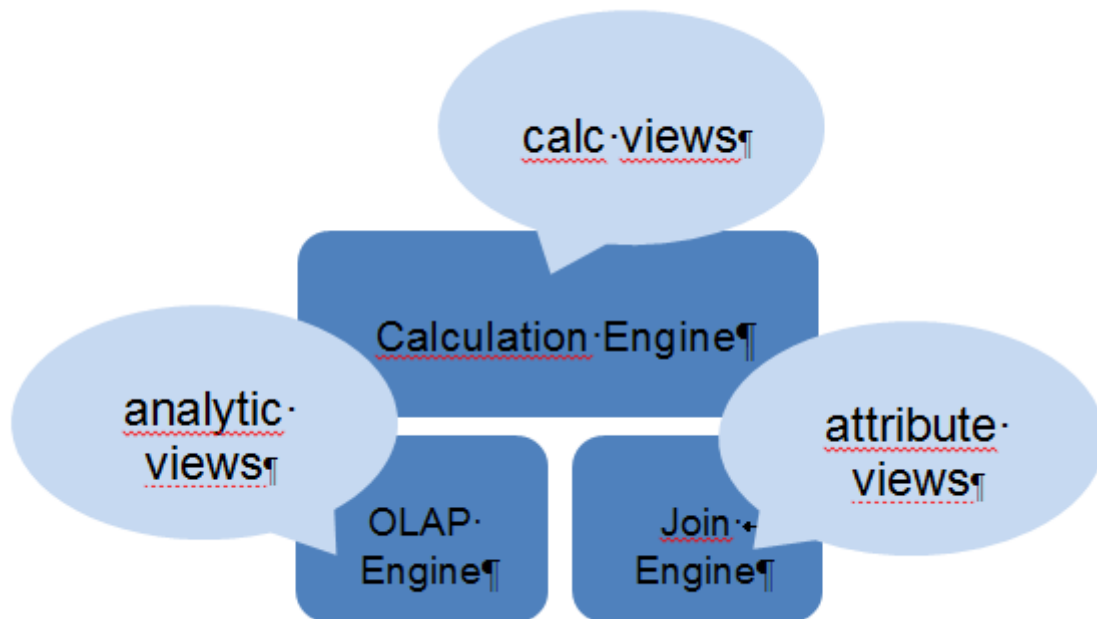
Calculation views are defined as either *graphical views* or *scripted views* depending on how they are created. They can be used in the same way as analytic views, however, in contrast to analytic views it is possible to join several fact tables in a calculation view. Calculation views always have at least one measure.

Graphical views can be modeled using the graphical modeling features of the SAP HANA Information Modeler. Scripted views are created as sequences of SQLScript statements. In essence they are SQLScript procedures with certain properties.

2.3.3 SAP HANA View Processing

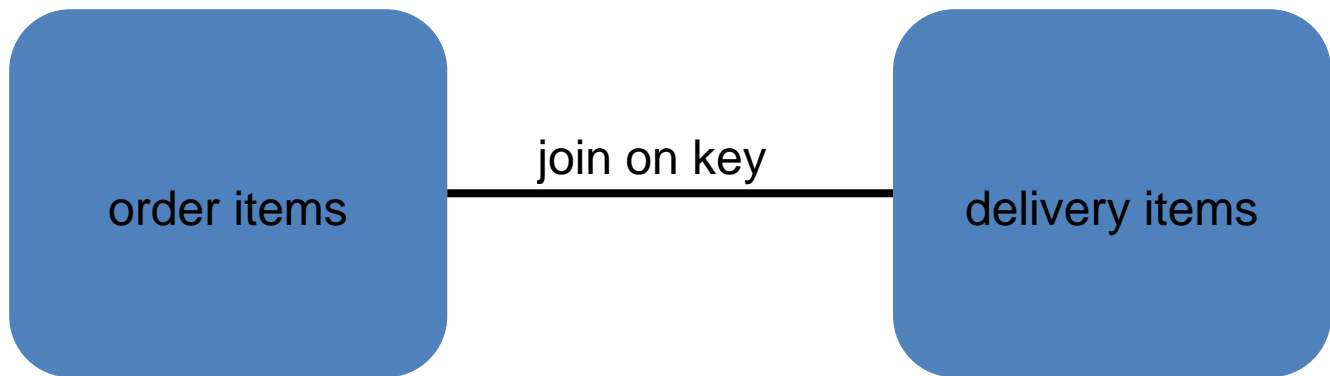
A basic understanding how SAP HANA processes views is required so that you can ensure that data transfer within the database is minimized.

A simplified view of the system is illustrated by the diagram below.

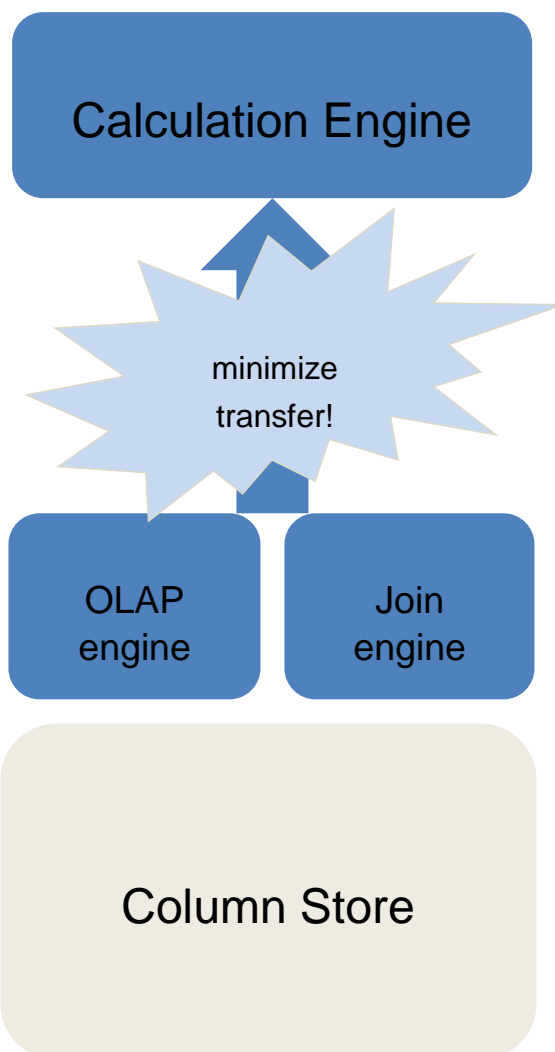


From the diagram we can see that SAP HANA has three types of engines that are used based on the views required by the model.

- Calculation engine – used on top of OLAP engine and/or join engine for complex calculations that cannot be achieved using the join or OLAP engines.
- OLAP engine – used for calculation and aggregation “based on star schema” or similar
- Join engine – Engine used for all type of joins



An SQL optimizer decides the best way to call the engines function based on the models and queries involved.



This diagram is somewhat simplified. For example an analytic view with a calculated attribute or that includes an attribute view containing a calculated attribute, will become a calculation view. This should be taken into consideration during modeling because it can have a large impact on the performance of the data model.

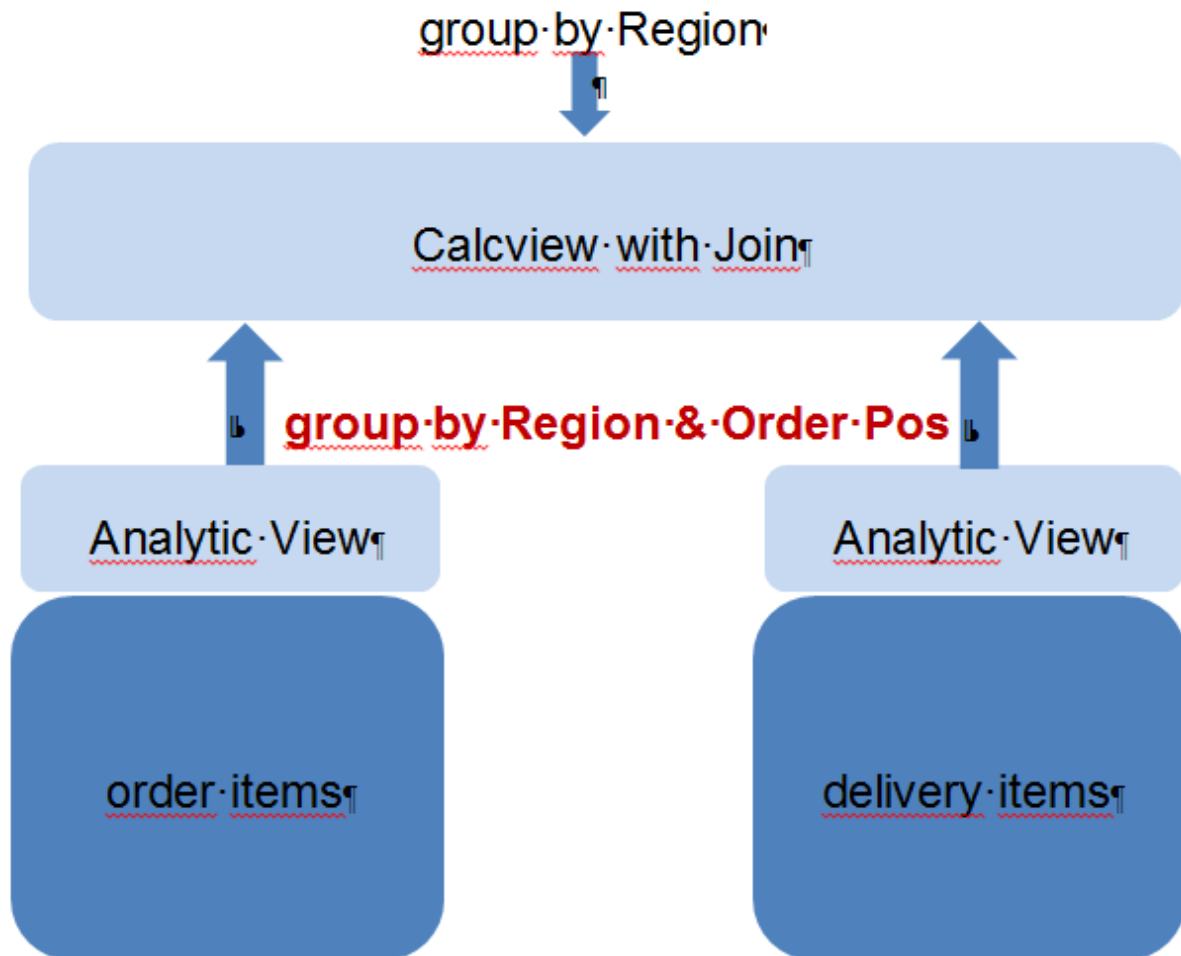
Modeling Guidance – Minimize Data Transfers

As has been stressed many times already in this document, a primary goal during modeling is to minimized Data transfers. This ternate holds true both internally, between the SAP HANA database engines, and also between SAP HANA and the end user application. For example, the end-user will never need to see 1 million rows of data. They would never be able to understand so much information or consume it in some meaningful way. This means that Data, wherever possible, should be aggregated and filtered to a manageable size before it leaves the datalayer.

When deciding upon the records that should be reported upon, a best practice approach is to think at a “set level” not a “record level”. A set of data can be aggregated by a region, a date, or some other group in order to minimize the amount of data passed between engines.

Basic Modeling Rules to Follow – an Example

For this example, the requirement is that the two fact tables are joined by their keys. This drives how the query of the data should be carried out.

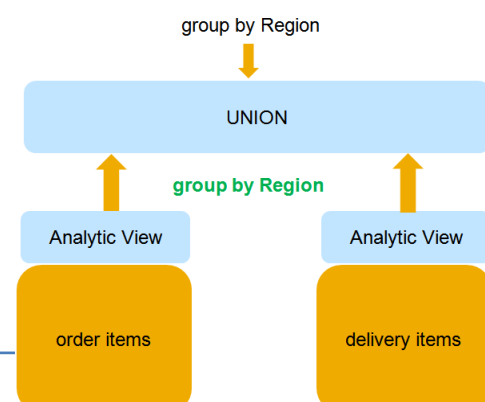


As mentioned before, always think in sets of data not rows (or records) of data. The image below depicts a poor design as it will join on the keys of the tables. In doing so, it will have to touch each individual record during the query. This is costly during run time due to the use of joining the data in the calculation view.

In the case when both analytic views will generate a lot of records, this join must be avoided and replaced with a union. A join can be used when one of the analytic views generates a small number of records (less than one million) into a calculation view.

Before the operation which transfers data from the OLAP engine to the calculation engine, the data must be minimized - this can be done using projections. Using projections, the number of records generated by an analytic view can be reduced by grouping and by applying filter criteria. These observations can be utilized during the creation of calculation views to avoid performance issues.

The image to the right shows a more efficient way of constructing the view. Note that the two separate analytic views are initially grouped by region and then brought together via a union. This is more efficient than joining at the row level.



Use Set Processing for Mass Data

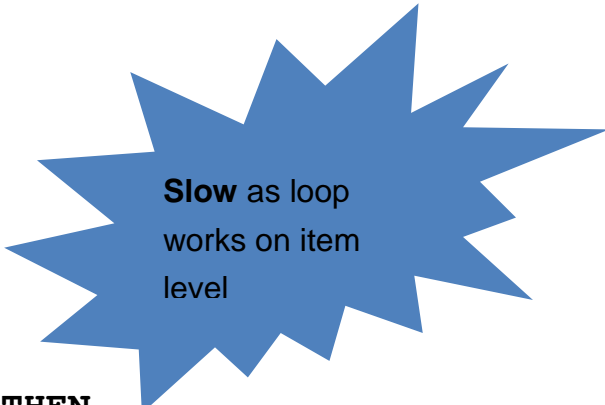
As mentioned in the previous section, using a “Set approach” rather than a “Row approach” to processing data will always yield better performance. In this section we will examine this further.

Both of the queries below will eventually access the same data. The first query sets a qualifier in the first line, then must once again query for all of the rest of the data in the set.

The code below is an **example of a poor query**:

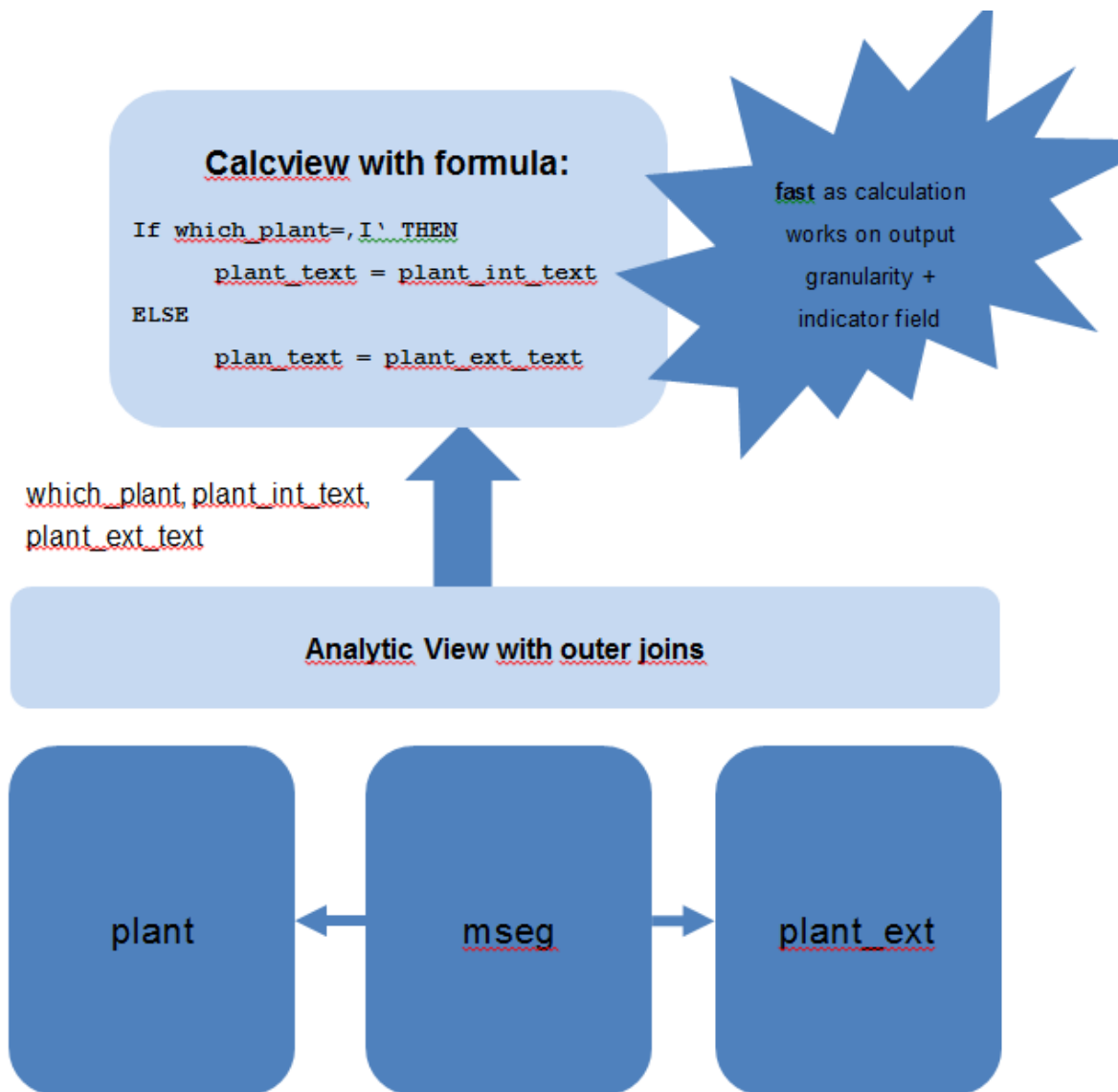
```
matmoves = SELECT * FROM MSEG
FOR EACH matmove in matmoves
    IF matmove.whichPlant = „I“ THEN
        plant_text = SELECT plant_text FROM WERKS
WHERE id=matmove.plant
ELSE
```

[pseudo code]



Slow as loop
works on item
level

The query below is **an example of a more efficient way** of getting the same data. It is more efficient as it is querying against set of data, rather than all of the data.



It is very important to avoid loops when modeling as this will cause very poor performance. This is especially true when the amount of data in the table that will be looped upon is very high (hundreds of millions of data entries).

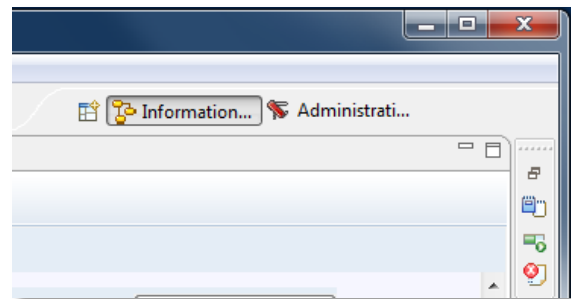
3 Tutorials

3.1 Using the SAP HANA Studio

SAP HANA Studio is a tool for developers who wish to create data models and stored procedures, or manipulate tables and issue queries interactively. In the following tutorial we assume the following:

- You have access to a SAP HANA appliance running on a host named "hana.mynetwork.com" from your workstation via TCP/IP.
- The instance number is 99.
- Your workstation has the SAP HANA Studio is installed. To use SAP HANA Studio you need to know the name of your SAP HANA user account and its respective password. This information can be obtained from your local database administrator. Please note that access to some schemas (catalogs of data models and objects) is restricted.

Please start SAP HANA studio on your workstation. The SAP HANA studio offers two main areas of operation, known as perspectives, that can be selected in the upper right corner of the Studio window. The perspective you will be using most frequently is the *Information Modeler perspective*. The *Administration Console perspective* is required less often. Readers familiar with the Eclipse integrated development environment will note that both perspectives have been implemented in the Eclipse framework.



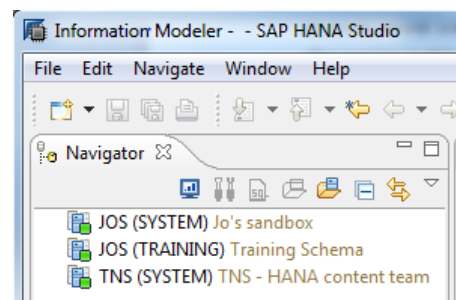
Please switch to the Information Modeler perspective by clicking on the respective field in the upper right-hand corner, or alternatively by selecting *Window ► Open Perspective ► Information Modeler* from the pull-down menu. This perspective offers features to create and edit data models and stored procedures that can be used by the Business Objects Explorer analytics application and certified products which support MDX, SQL or BICS. The artifacts created by the Modeler are stored directly in the SAP HANA tables as XML documents ("design-time objects").

The Administration Console perspective is useful when you wish to manipulate the data dictionary of SAP HANA, create or drop schemas, import or export data, or interactively query the database using SQL.

Registering a SAP HANA Appliance with SAP HANA Studio.

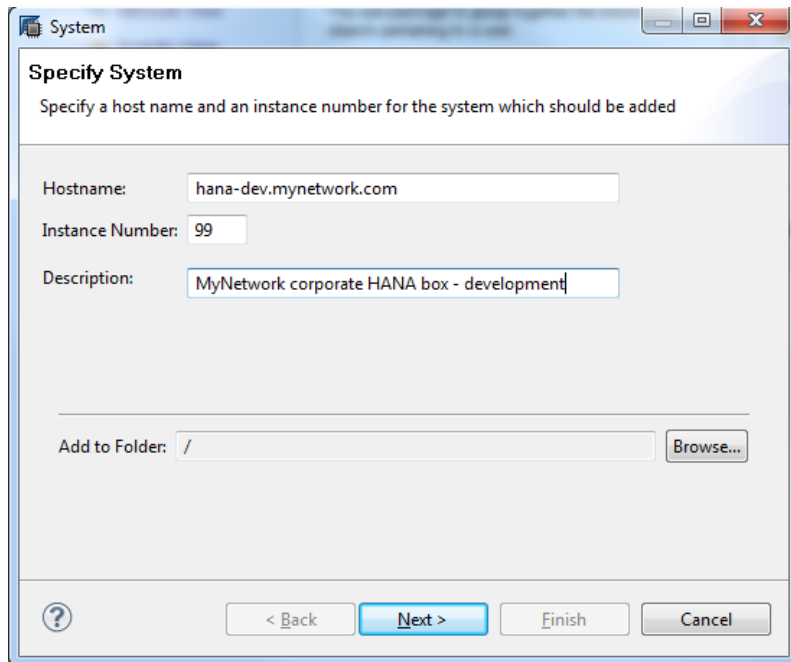
Within the SAP HANA studio main window there are a number of sub-windows called views. The leftmost one is labeled *Navigator*.

Right-click into the Navigator view to bring up a context menu. Select *Add System* from that menu. In the dialog window that appears, please enter the following information:

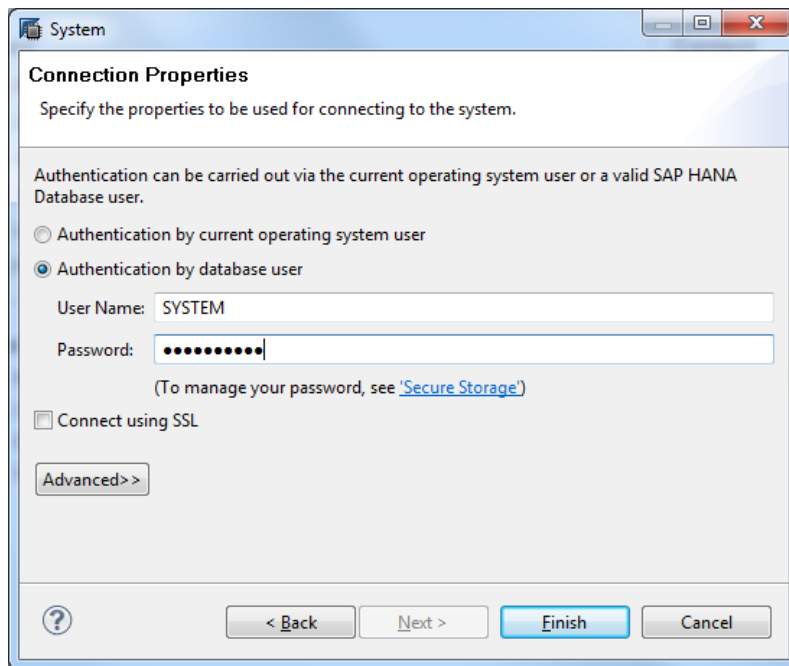


- Hostname: hana-dev.mynetwork.com,
- Instance Number: 99
- Description: MyNetwork corporate SAP HANA box – development

Click *Next* in the dialog window to continue.

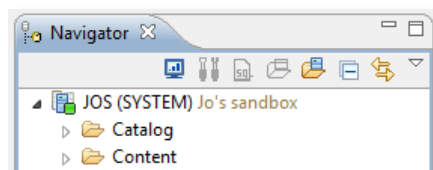


Please enter the name of the system or personal user name you have been given by your system administrator, the respective password and then click *Finish*. If the data entered is correct, a half expanded tree structure will appear in the Navigator view with the name of the system as its top node. Please double-click on the Catalog. Under that node you will find the nodes *Authorization* and *Public Synonyms* as well as a number of entries which represent database schemas.



In the Navigator view you should see a new node labeled *HANA (SYSTEM) MyNetwork corporate SAP HANA box – development*. Click on the white triangle to see the system contents. There are two sub-nodes, *Catalog* and *Content*. The first one represents SAP HANA's data dictionary, i. e. all data structures, tables, and data which can be used. The node *Contents* represents the design-time repository which holds all models created with the Information Modeler. Physically these models are stored in database tables which are also visible under *Catalog*. The *Contents* node just provides a different view on the same physical data.

In the following screen shots we will use a different system. Its Navigator node is labeled *JOS (SYSTEM) Jo's sandbox*.



Loading example data into SAP HANA

In the following chapters we will use data models and data which are available for download from

<https://sapmats-de.sap-ag.de/download/download.cgi?id=Z0UGTDG927WEEZDB9DXMYV6TSA9SZCLP3SV5NV6X9UU0XJ6SFA>

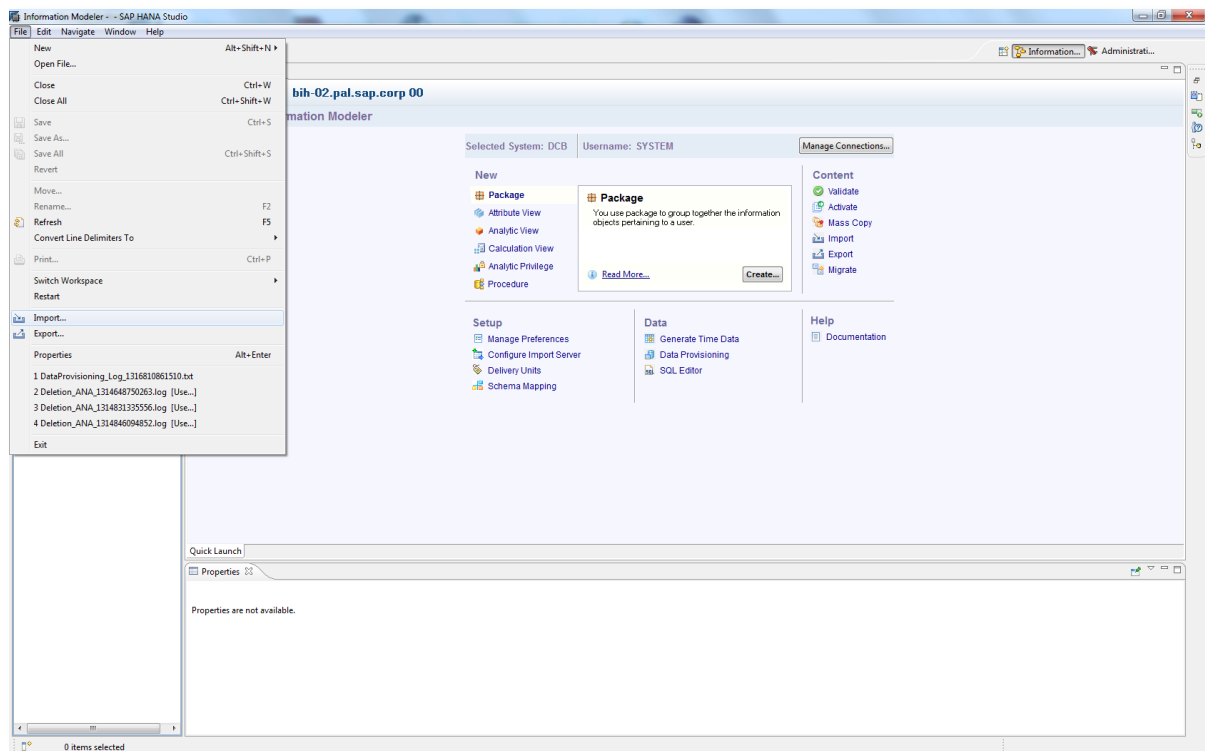
Please download the file into a directory on the SAP HANA appliance, e. g. `/home/guest/sflight` and unzip it using the command

```
unzip sflight.zip
```

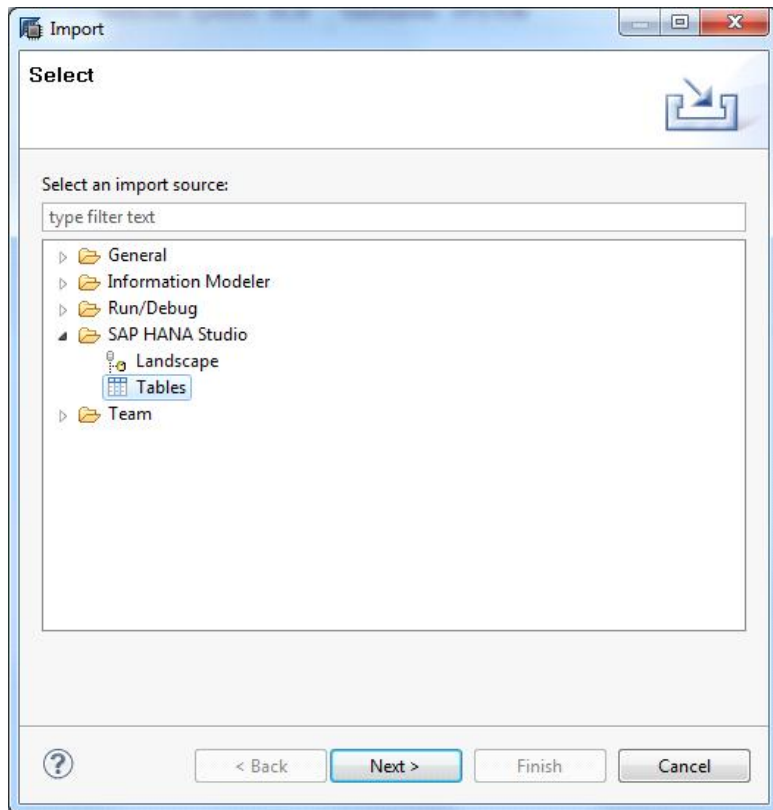
You should now see a new directory named `index`.

```
Terminal — ssh — 80x24
inflating: index/SFLIGHT/ST/STRAVELAG/RuntimeData
inflating: index/SFLIGHT/ST/STRAVELAG/freeUdivStore
creating: index/SFLIGHT/ST/STRAVELAG/attributes/
inflating: index/SFLIGHT/ST/STRAVELAG/attributes/attribute_1.bin
inflating: index/SFLIGHT/ST/STRAVELAG/attributes/attribute_201.bin
inflating: index/SFLIGHT/ST/STRAVELAG/attributes/attribute_202.bin
inflating: index/SFLIGHT/ST/STRAVELAG/attributes/attribute_203.bin
inflating: index/SFLIGHT/ST/STRAVELAG/attributes/attribute_204.bin
inflating: index/SFLIGHT/ST/STRAVELAG/attributes/attribute_205.bin
inflating: index/SFLIGHT/ST/STRAVELAG/attributes/attribute_206.bin
inflating: index/SFLIGHT/ST/STRAVELAG/attributes/attribute_207.bin
inflating: index/SFLIGHT/ST/STRAVELAG/attributes/attribute_208.bin
inflating: index/SFLIGHT/ST/STRAVELAG/attributes/attribute_209.bin
inflating: index/SFLIGHT/ST/STRAVELAG/attributes/attribute_210.bin
inflating: index/SFLIGHT/ST/STRAVELAG/attributes/attribute_211.bin
inflating: index/SFLIGHT/ST/STRAVELAG/attributes/attribute_212.bin
inflating: index/SFLIGHT/ST/STRAVELAG/attributes/attribute_213.bin
inflating: index/SFLIGHT/ST/STRAVELAG/attributes/attribute_7.bin
inflating: index/SFLIGHT/ST/STRAVELAG/attributes/attributeStore.js
inflating: index/SFLIGHT/ST/STRAVELAG/table.xml
inflating: index/SFLIGHT/ST/STRAVELAG/create.sql
guest@hanasvr-02:~/sflight> ls
index  sflight.zip
guest@hanasvr-02:~/sflight>
```

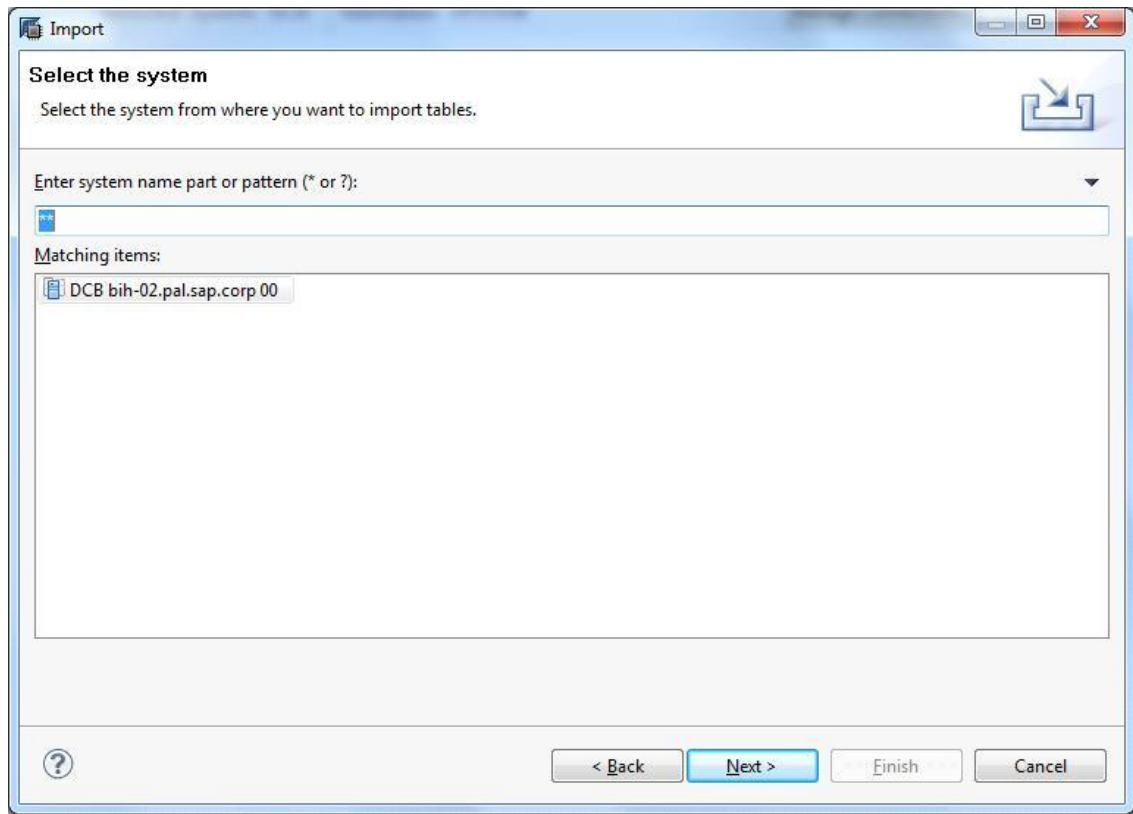
Open your SAP HANA studio and choose **File ► Import** from the main menu or the Quick Launch view.



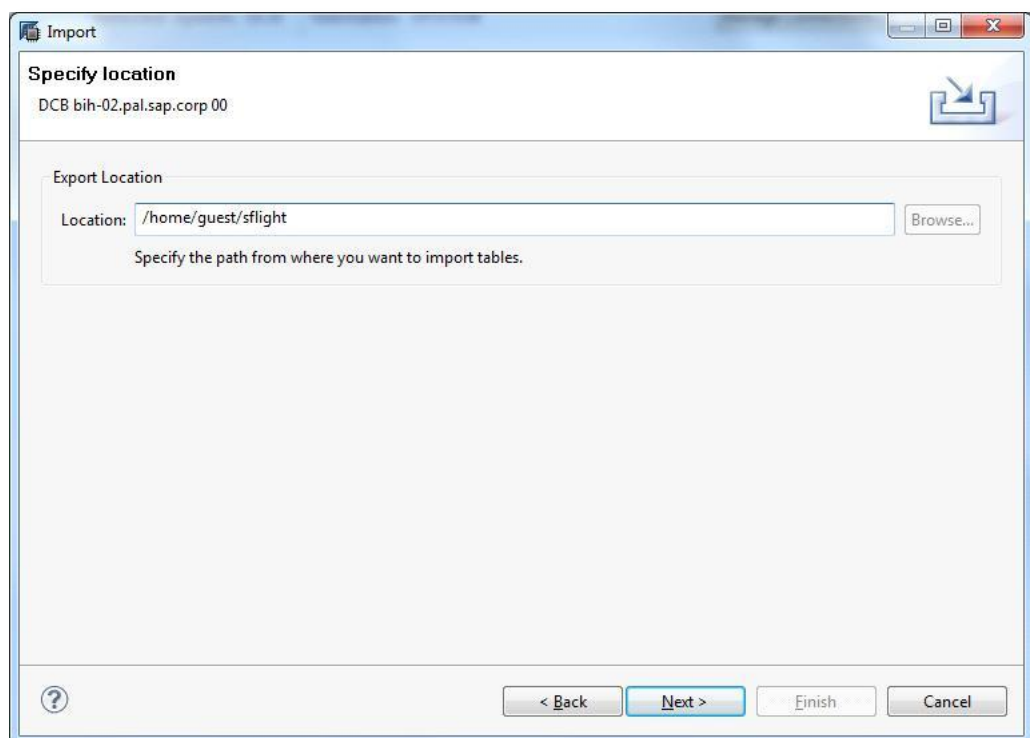
On the popup, navigate to *SAP HANA Studio* ► *Tables* and click *Next* .



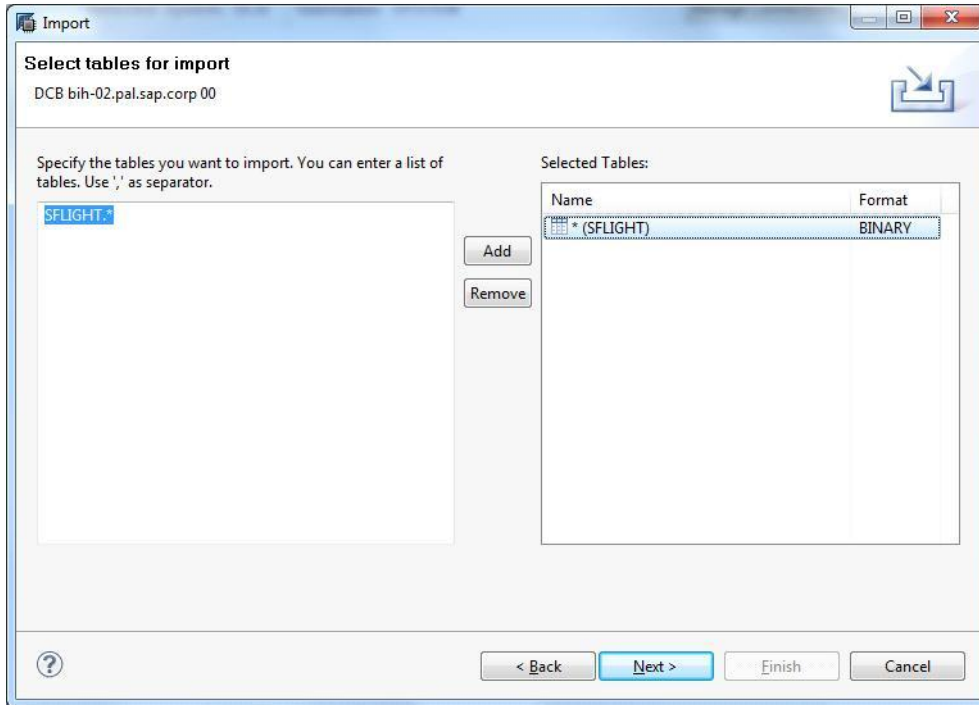
You may be prompted to choose the SAP HANA DB system to import into. Select the relevant system and click *Next*.



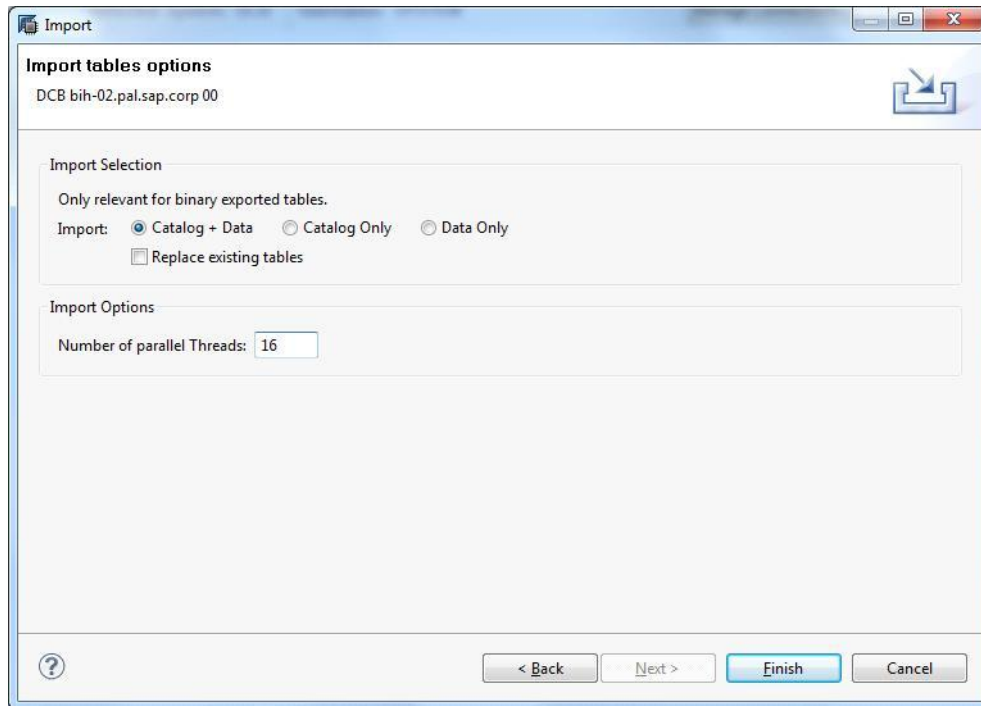
Enter the path to your upload directory, e. g. `/home/guest/sflight`, and click *Next* . – the <SID>adm user of the SAP HANA DB system needs at least read permissions for this directory.



On the following screen, enter “SFLIGHT.*” (as in “all tables in the import directory that belong to schema SFLIGHT”) into the left text area and click the *Add* button. A new entry * (SFLIGHT) appears in the table to the right. In the format column of that new row, choose *BINARY* from the drop down menu. Click *Next* .



Finally, leave the import Radio Button “Catalog + Data” checked and enter a number of parallel threads for the import. The optimal number is equal or less than the number of cores on your SAP HANA server, but also depends on how many users you are sharing the system with. The more threads you choose, the faster your upload will be (and the more you will block other users).



After just a few seconds (on 16 threads it should even take less than a second), you should be able to see a new schema SFLIGHT with the imported tables.

We will later learn which tables and data were imported and how to display them.

3.2 How to Use the SAP HANA Information Modeler

In the following section we are going to learn how to use the SAP HANA Information Modeler to create various views based on our imported data set. We will start with a simple example of an analytic view and proceed to an attribute view that will be combined with an analytic view to form a star schema. We will learn how to focus on data from a single client/tenant, filter data from attributes and add attributes which are calculated from other attributes.

To learn about the full functionality of the SAP HANA Information modeler, consult the [SAP HANA Modeling Guide](#).

Four steps to a high-performance data model. The ordering of the examples has been chosen to reflect the correct development path to follow to address more complex scenarios. Whenever possible you should follow the sequence presented here to produce a model with optimal performance.

In general you should start with simple analytic views. If this does not offer enough features add attribute views to create a simple star schema. If this does not fit either try calculation views, which are more flexible but also more complex. After this, try graphical views before finally using scripted views. For more details please see chapter 4.3.5.

Our scenario is based on flight bookings. The questions we will deal with include:

- How much revenue was made by each travel agency?

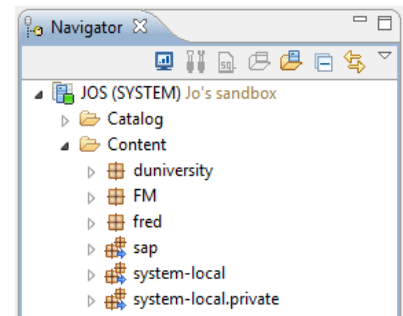
- How can we combine the revenue per travel agency with more information about the agencies?
- How much revenue would have been made if a discount had been given for bookings in a particular currency?

To continue with our first example please start up SAP HANA Studio on your workstation and switch to the Information Modeler perspective.

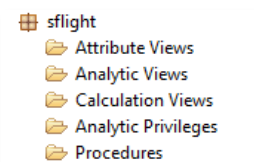
3.2.1 How to Create Analytic Views

The next steps will teach you how to use the Information Modeler to create analytic data models. Let us start with defining a space in the repository where our models will be stored. First click on the *Content* node to expand it. In the content node you should be able to see a number of package nodes.

Right-click on the *Content* node and choose *New ► Package*. In the pop-up window please enter a name and description for your new model package. Click OK to save it.



Under *Content* there is a new package node labeled *sflight*. Click on the little triangle left of it to display its inner structure. Some of the sub-nodes are obvious: the *Attribute Views* folder is used to store attribute views, *Analytic Views* is used to store analytic views, and *Calculation Views* is used to store calculation views. These three view types have been discussed earlier.



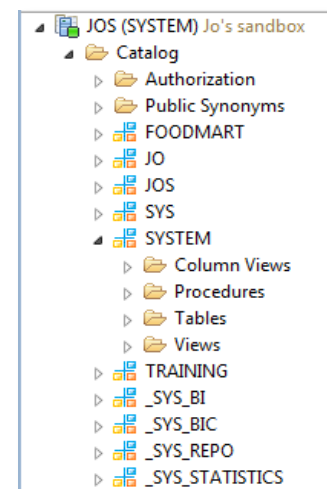
Analytic Privileges are used to bundle several packages and models and make them accessible using the same set of credentials. Node *Procedures* is used to store SQLScript procedures.

Now we are prepared to model our first view. But before we do that we have to learn about the tables we are going to work with.

3.2.2 Inspecting the Tables

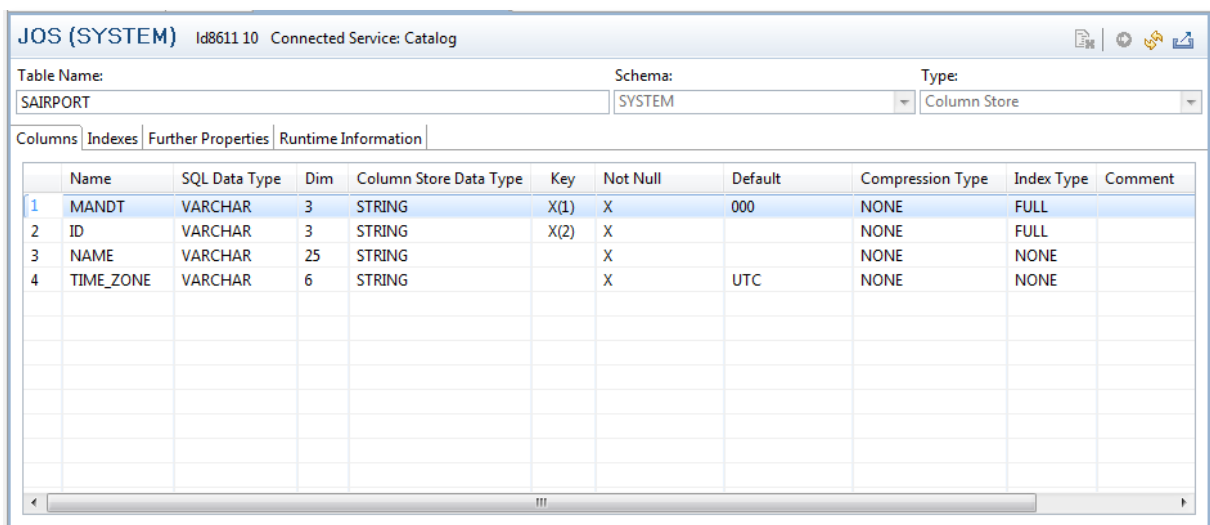
The tables we loaded earlier contain data about flights, travel agencies, customers, tickets, bookings, and on-board meals. The tables and their contents were taken from the SFLIGHT educational example which is available in every SAP IDES system, a preconfigured, preloaded SAP ERP system for educational purposes. Please note that only a subset of the tables of SFLIGHT is included in the download package.

To see the tables please click in the navigator view on the triangle near *Catalog* to see the different schemas which are available. Please click on the triangle next to *SYSTEM*. This is the schema node where the tables were imported. There are other important schemas. *_SYS_REPO* is used to store the design-time models which appear under *Content*. *_SYS_BIC* under *Catalog* keeps the run-time objects generated from these models. Please click on the triangle next to *Tables* to see a list of all tables in schema *SYSTEM*. The following SFLIGHT tables are available:



- SAIRPORT – master data table with name and time zone information for airports
- SAPLANE – master data table with information about plane types, e. g. number of seats, weight, tank capacity, and operating speed
- SBOOK – fact table containing all flight bookings and their details like date of flight, flight number, price, customer ID, and ID of the travel agency that sold the ticket.
- SBUSPART – master data table with information about business partners
- SCARR – master data table with some airline information (name, currency, URL)
- SCURR – table with currency conversion factors
- SCURX – per-currency number of decimals used to display currency values
- SCUSTOM – master data table containing information about passengers
- SDESSERT – master data table with information about the desserts available during flights
- SFLIGHT – master data table containing per-flight information, e. g. date, price, plane type, seats
- SMACOURSE – master data table with information about the main courses available during flights
- SMEAL – master data table with information about the meals available during flights
- SMEALT – master data table with information about the meals in various languages
- SMENU – master data table with information about the available combinations of meals
- SSTARTER – master data table with information about the main courses available during flights
- STICKET – fact table which contains information about the tickets issued at the airports
- STRAVELAG – master data table containing information about travel agencies

Right-click on the table node SAIRPORT and choose *Open Definition* to show the structure of the table. In the main view you will see a tabular display with the technical details of the table. From the display we can learn the table name (SAIRPORT), the schema (SYSTEM), and the storage type (column store). In addition to this, the different columns or attributes are described. To learn about the full functionality of this view please consult the [SAP HANA Modeling Guide](#).



| | Name | SQL Data Type | Dim | Column Store Data Type | Key | Not Null | Default | Compression Type | Index Type | Comment |
|---|-----------|---------------|-----|------------------------|------|----------|---------|------------------|------------|---------|
| 1 | MANDT | VARCHAR | 3 | STRING | X(1) | X | 000 | NONE | FULL | |
| 2 | ID | VARCHAR | 3 | STRING | X(2) | X | | NONE | FULL | |
| 3 | NAME | VARCHAR | 25 | STRING | | X | | NONE | NONE | |
| 4 | TIME_ZONE | VARCHAR | 6 | STRING | | X | UTC | NONE | NONE | |

Many of the tables have a MANDT column which contains a client or tenant number. This column is used in SAP ERP systems to distinguish data belonging to different companies, e. g. data for a holding

and its subsidiaries. In addition to that our airport table has a column with the three-letter IATA codes of the airports, the airports' names, and their time zones. The primary index of the table includes the key attributes MANDT and ID (column *Key*). The data set used in these examples contains data from clients 200 and 800. Please check your SAP HANA installation and SAP ERP system for the clients your system has.

To preview data in the table right-click on the table node *SAIPOINT* and choose *Open Content*. In the main view you will see a new tab appearing which shows a couple of lines of the table and the SQL statement which was used to access the data. You can copy & paste the SQL statement into other applications or into the SQL Editor view which will be described later.

Please take some time to inspect the SFLIGHT tables. We will be using some of them soon.

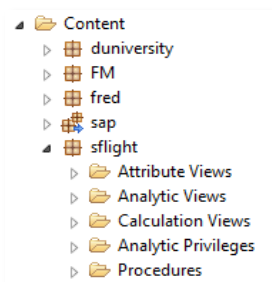
JOS (SYSTEM) Id8611 10 Connected Service: Catalog

```
SELECT TOP 1000 * FROM "SYSTEM"."SAIPOINT"
```

| | MANDT | ID | NAME | TIME_ZONE |
|----|-------|-----|-----------------------------|-----------|
| 1 | 800 | ITM | Osaka Itami Apt, Japan | UTC+9 |
| 2 | 800 | JFK | New York JF Kennedy, USA | UTC-5 |
| 3 | 800 | JKT | Jakarta, Indonesia | UTC+7 |
| 4 | 800 | KIX | Osaka Kansai Int., Japan | UTC+9 |
| 5 | 800 | KUL | Kuala Lumpur, Malaysia | UTC+8 |
| 6 | 800 | LAS | Las Vegas Mc Carran, USA | UTC-8 |
| 7 | 800 | MUC | Munich, FRG | UTC+1 |
| 8 | 800 | NRT | Tokyo Narita, Japan | UTC+9 |
| 9 | 800 | ORY | Paris Orly Apt, France | UTC+1 |
| 10 | 800 | PID | Nassau Paradise IS, Baha... | UTC-5 |
| 11 | 800 | RTM | Rotterdam Apt, NL | UTC+1 |
| 12 | 800 | SEL | Seoul Kimpo Int, ROK | UTC+9 |
| 13 | 800 | TYO | Tokyo, JAPAN | UTC+9 |
| 14 | 800 | VCE | Venice Marco Polo Apt, I | UTC+1 |

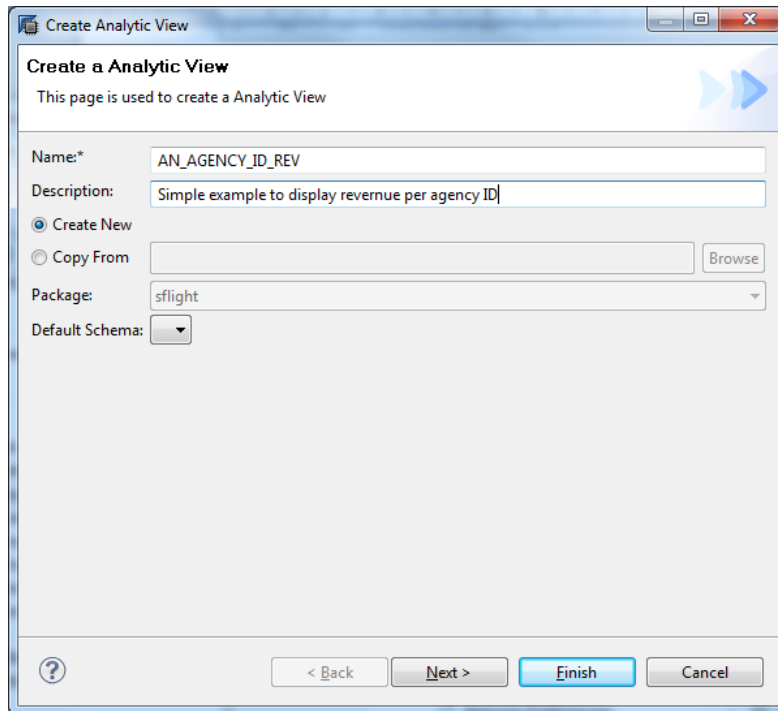
3.2.3 Creating Analytic Views

As our first example we are going to create an analytic view on table SBOOK. Let us assume that we want to report the summed up revenue per travel agency. To keep things simple we will display the travel agencies' numerical IDs instead of their full names. We will deal with how to obtain the textual names later.



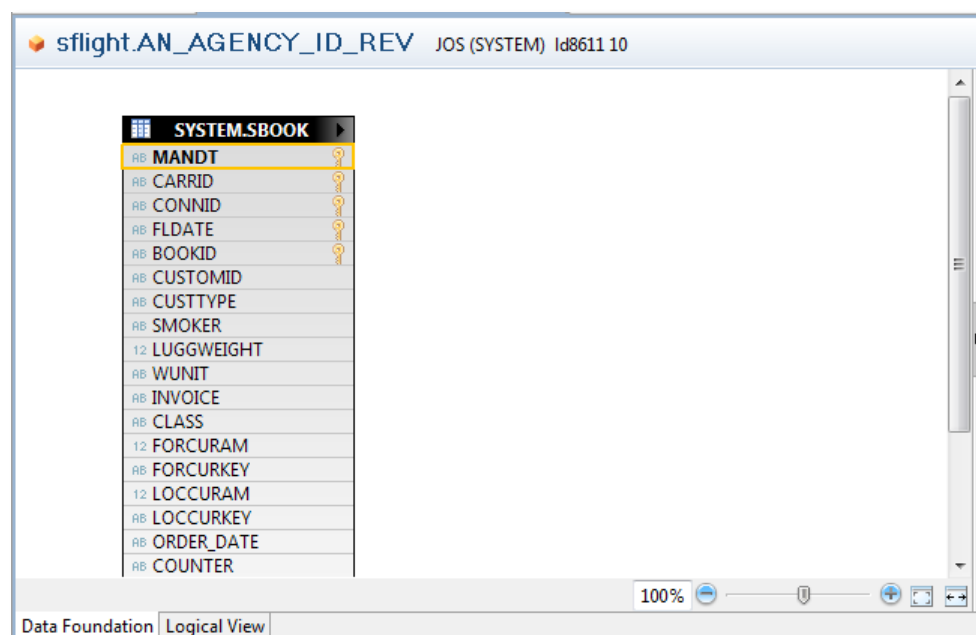
In the navigator view please expand the *Content* node for your database. Then expand package node *sflight*. Right-click on *Analytic Views* and choose *New* ► *Analytic View*.

In the appearing pop-up window enter a name and description of your first analytic view. Here we use name *AN_AGENCY_ID_REV*. Click on *Finish* to save your analytic view.

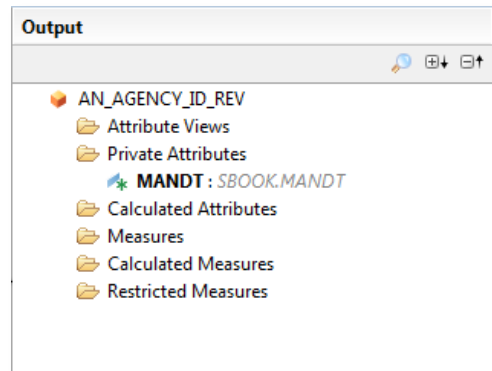


In the main view the Modeler canvas labeled *sflight.AN_AGENCY_ID_REV* appears. The canvas is empty because we did not define any tables to be used in our analytic view yet. Lets now add a table.

In the Navigator view find the *SBOOK* table node under *Catalog/SYSTEM/Tables*. Drag and drop the *SBOOK* node on the empty Modeler canvas. The Modeler canvas now displays a rectangular box containing the table's name and all of its attributes and columns.



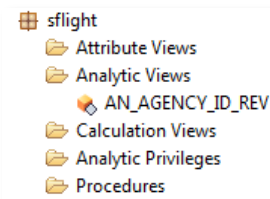
On The bottom of the Modeler view we see two tabs, *Data Foundation* and *Logical View*. To the right there is a slider to change the scaling of the displayed model. Try the slider to get acquainted with its behavior. Then click on *Logical View*. There is a small rectangle labeled *Data Foundation*. Its attribute list is empty. Switch back to the *Data Foundation* tab and right-click on attribute *MANDT* to add it to the logical view. From the pop-up menu choose *Add as Attribute*. To the right of the Modeler canvas you see the output table for the analytic view. This table contains all attributes and measures defined. After adding *MANDT* you will see that a new node is created in the output table with the same label. Switch to tab *Logical View* to see the correspondence of attributes chosen from the data foundation to be available in the analytic view. The logical view can be thought of as a black-box view of the model.



Please add the attributes *FURCURKEY* (numeric key of foreign currency) and *AGENCYNUM* (travel agency number) to the output table in the same way as you did for *MANDT*.

Right-click on attribute *FURCURAM* (foreign currency amount) and choose *Add as Measure*. Check the output table and logical view to see the result of your actions.

Save your model by choosing *File ► Save* from the top pull-down menu. Alternatively you can press *Ctrl-S*. Your new analytic view is visible in the Navigator view as a node in package *sflight*. Its icon



is marked with a gray diamond. This means that the model has been saved in the repository but has not been activated yet, i. e. no run-time object in Schema *_SYS_BIC* has been created. Activate the analytic view by right-clicking on Navigator node *AN_AGENCY_ID_REV* and choosing *Activate*. The gray diamond will disappear to signal a successful activation.

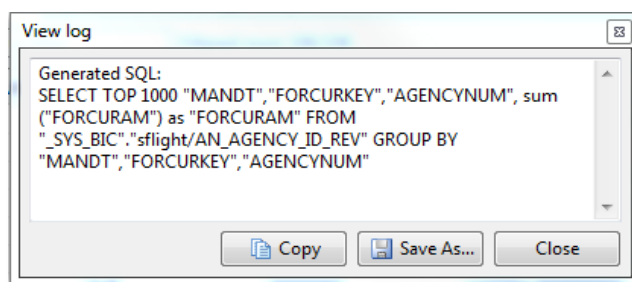
Congratulations! You have just created your first SAP HANA analytic view, but does it work? Right-click on *AN_AGENCY_ID_REV* and choose *Data Preview*. In the main view you will see a tabular display of the data retrieved using the analytic view that you just created.


106 rows (1404 ms) Max rows: 1000 Refresh

Raw Data Distinct values Analysis

Enter your filter Filtered rows: 106/106 Add filter Save as file

| AB | MANDT | AB | FORCURKEY | AB | AGENCYNUM | 12 | FORCURAM |
|-----|-------|-----|-----------|----------|-----------|--------------------|----------|
| 200 | | EUR | | 00000000 | | 4368689.9599999655 | |
| 200 | | JPY | | 00000000 | | 1326851.1 | |
| 200 | | SGD | | 00000000 | | 2637250.2599999974 | |
| 200 | | USD | | 00000000 | | 3201395.7200000063 | |
| 200 | | USD | | 00000055 | | 1256243.0599999987 | |
| 200 | | EUR | | 00000061 | | 1421322.5799999996 | |
| 200 | | EUR | | 00000087 | | 1437704.0899999996 | |
| 200 | | EUR | | 00000093 | | 1365469.4699999993 | |
| 200 | | EUR | | 00000100 | | 1384822.3900000001 | |
| 200 | | EUR | | 00000101 | | 1588129.2099999995 | |
| 200 | | AUD | | 00000102 | | 1022163.3600000006 | |
| 200 | | SGD | | 00000103 | | 2248706.5300000017 | |
| 200 | | EUR | | 00000104 | | 1381710.5500000003 | |
| 200 | | EUR | | 00000105 | | 1368267.6499999994 | |
| ... | ... | ... | ... | ... | ... | ... | ... |



Click on the info icon  in the upper right area of the view to display the SQL statement used to retrieve the data displayed. Now let's dissect the statement for a minute. For readability we have omitted all quotes which are redundant.

```
SELECT TOP 1000 MANDT, FORCURKEY, AGENCYNUM, SUM(FORCURAM) AS FORCURAM
FROM _SYS_BIC."sflight/AN_AGENCY_ID_REV"
GROUP BY MANDT, FORCURKEY, AGENCYNUM
```

The statement selects data from view sflight/AN_AGENCY_ID_REV which resides in schema _SYS_BIC (the place for all generated run-time objects). The attributes MANDT, FORCURKEY, and AGENCYNUM are selected. In addition to this, the sum of attribute FORCURAM is calculated and displayed as column FORCURAM. Grouping is done over attributes MANDT, FORCURKEY, and AGENCYNUM.

So what does the data preview display? For each travel agency (more specifically its numeric ID) the revenue per currency is displayed. We can learn that agency no. 00000000 contributed to the revenue in Euros (EUR), Yens (JPY), and Singapore Dollars (SGD).

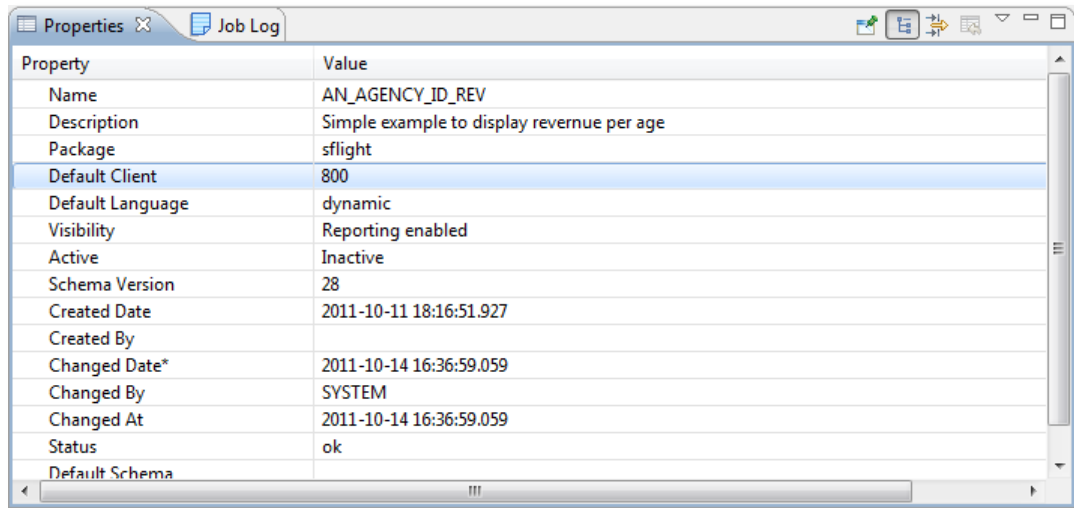
3.2.4 MANDT Attributes

Our table displays entries with all possible values of the MANDT attribute. This is not useful because different MANDT values distinguish different companies operating a single SAP ERP system, e. g. a holding and its subsidiaries it is useless to sum up revenues of different companies.

Our data set contains data for two MANDT values 200 and 800. So let's only select data from the latter. Below the Modeler there is another SAP HANA studio view which displays certain properties of

our analytic view. The *Default Client* attribute value is *dynamic* by default. Double-click on the value field to edit it. Please set the value to 800.

Save the analytic view and activate it. The data preview now displays only rows with MANDT = 800.



| Property | Value |
|------------------|---|
| Name | AN_AGENCY_ID_REV |
| Description | Simple example to display revenue per age |
| Package | sflight |
| Default Client | 800 |
| Default Language | dynamic |
| Visibility | Reporting enabled |
| Active | Inactive |
| Schema Version | 28 |
| Created Date | 2011-10-11 18:16:51.927 |
| Created By | |
| Changed Date* | 2011-10-14 16:36:59.059 |
| Changed By | SYSTEM |
| Changed At | 2011-10-14 16:36:59.059 |
| Status | ok |
| Default Schema | |

3.2.5 Applying Filters

Let us investigate only the revenue made in Euros. This can be done by setting a filter on attribute FORCURKEY. It is possible to apply filters to the attributes of analytic and attribute views.

In the *Data Foundation* tab right-click on FORCURKEY and select *Apply Filter*. The pop-up dialog allows you to select among several operator, e. g. *Between* or *Equal*. Please choose *Equal* and enter the value *EUR*. Save your model and activate it. Now the data preview only displays rows with FORCURKEY = EUR.

3.2.6 Creating Attribute Views

Now back to our problem: What revenue was made by each travel agency? We could USE what we have if we could remember the IDs of all the travel agencies. On the other hand, there is table STRAVELAG which contains data for each travel agency so we can use this to make the travel agency names human readable.

JOS (SYSTEM) Id8611 10 Connected Service: Catalog

Table Name: STRAVELAG Schema: SYSTEM Type: Column Store

Columns | Indexes | Further Properties | Runtime Information

| | Name | SQL Data Type | Dim | Column Store Data Type | Key | Not Null | Default |
|----|-----------|---------------|-----|------------------------|------|----------|----------|
| 1 | MANDT | NVARCHAR | 3 | STRING | X(1) | X | 000 |
| 2 | AGENCYNUM | NVARCHAR | 8 | STRING | X(2) | X | 00000000 |
| 3 | NAME | NVARCHAR | 25 | STRING | | X | |
| 4 | STREET | NVARCHAR | 30 | STRING | | X | |
| 5 | POSTBOX | NVARCHAR | 10 | STRING | | X | |
| 6 | POSTCODE | NVARCHAR | 10 | STRING | | X | |
| 7 | CITY | NVARCHAR | 25 | STRING | | X | |
| 8 | COUNTRY | NVARCHAR | 3 | STRING | | X | |
| 9 | REGION | NVARCHAR | 3 | STRING | | X | |
| 10 | TELEPHONE | NVARCHAR | 30 | STRING | | X | |
| 11 | URL | NVARCHAR | 255 | STRING | | X | |
| 12 | LANGU | NVARCHAR | 1 | STRING | | X | |
| 13 | CURRENCY | NVARCHAR | 5 | STRING | | X | |

This is a master data table. So let us create an attribute view to select only those attributes we are interested in, for instance NAME, CITY, COUNTRY, CURRENCY.

In the Navigator view open package *sflight* and right-click on node Attribute Views. Choose **New ▶ Attribute View**. In the pop-up dialog window enter a name, e. g. AT_AGENCY_DESCRIBED, and a description. Click on Finish.

Create Attribute View

New Attribute View

Enter a Attribute View name and Description..

Name*: AT_AGENCY_DESCRIBED

Description: Agency names, addresses, and currencies

Package: sflight

In the Navigator view browse to *Catalog / SYSTEM / Tables*. Click on table node *STRAVELAG*. Drag and drop it onto the Modeler canvas labeled *sflight.AT_AGENCY_DESCRIBED*. Please define MANDT, NAME, CITY, COUNTRY, and CURRENCY as view attributes. Add AGENCYNUM as a key attribute.

Change view property *Default Client* from *dynamic* to 800 in order to not mix up agencies handled by different companies. Please save the new attribute view and activate it. Now there is a new node *Content / sflight / AT_AGENCY_DESCRIBED* in the Navigator view. Right-click on it and choose **Data Preview** to see a list of all travel agencies in client 800.

50 rows (1155 ms) Max rows: 1006 Refresh

Raw Data Distinct values Analysis

Enter your filter Filtered rows: 50/50 Add filter Save as file

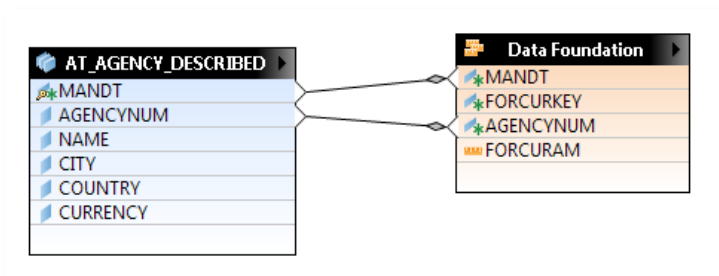
| RB | MANDT | RB | AGENCYNUM | RB | NAME | RB | CITY | RB | COUNTRY | RB | CURRENCY |
|-----|-------|----------|-----------|----|-----------------------|----|--------------|----|---------|----|----------|
| 800 | | 00000055 | | | Sunshine Travel | | Rochester | | US | | USD |
| 800 | | 00000061 | | | Fly High | | Duesseldorf | | DE | | EUR |
| 800 | | 00000087 | | | Happy Hopping | | Berlin | | DE | | EUR |
| 800 | | 00000093 | | | Pink Panther | | Frankfurt | | DE | | EUR |
| 800 | | 00000100 | | | Your Choice | | Nuernberg | | DE | | EUR |
| 800 | | 00000101 | | | Bella Italia | | Roma | | IT | | EUR |
| 800 | | 00000102 | | | Hot Socks Travel | | Sydney | | AU | | AUD |
| 800 | | 00000103 | | | Burns Nuclear | | Singapore | | SG | | SGD |
| 800 | | 00000104 | | | Honauer Reisen GmbH | | Neumarkt | | AT | | EUR |
| 800 | | 00000105 | | | Travel from Walldorf | | Berlin | | DE | | EUR |
| 800 | | 00000106 | | | Voyager Enterprises | | Stockholm | | SE | | SEK |
| 800 | | 00000107 | | | Ben McCloskey Ltd. | | Birmingham | | GB | | GBP |
| 800 | | 00000254 | | | Cap Travels Ltd. | | Johannesb... | | ZA | | ZAR |
| 800 | | 00000284 | | | Rainy, Stormy, Cloudy | | Stuttgart | | DE | | EUR |
| 800 | | 00000291 | | | Women only | | Mainz | | DE | | EUR |

3.2.7 Creating a simple Star Schema

We are now ready to join both the analytic and attribute view into a very simple star schema. We will use our attribute view as one “arm” of the star and table SBOOK as its center. We will create the star schema from the analytic view we created in the last section.

Please create a new analytic view in package sflight with the name AN_AGENCY_REV. In the create dialog please choose *Copy From* and select *sflight / Analytic Views / AN_AGENCY_ID_REV*.

Switch to the tab *Logical View* below the Modeler canvas. Now drag and drop attribute view AT_AGENCY_DESCRIBED onto the Modeler canvas. Do not be surprised by the warning message. There is a name clash between the attributes AGENCYNUM from both tables. SAP HANA Information Modeler creates an alias for the newly introduced attribute in order to avoid this clash.



Click on attribute AGENCYNUM in the Data Foundation box and drag and drop it onto the attribute AGENCYNUM in the box representing our new attribute view. Connect the MANDT attributes in the same manner. This action creates a join between both views

Output

AN_AGENCY_ID_REV

Attribute Views

AT_AGENCY_DESCRIBED

- NAME : STRAVELAG.NAME
- CITY : STRAVELAG.CITY
- COUNTRY : STRAVELAG.COUNTRY
- CURRENCY : STRAVELAG.CURRENCY
- AGENCYNUM(AT_AGENCY_DESCRIBED_AGENCYNUM) : STRAVELAG.AGENCYNUM

Private Attributes

- MANDT : SBOOK.MANDT
- FORCURKEY : SBOOK.FORCURKEY
- AGENCYNUM : SBOOK.AGENCYNUM

Calculated Attributes

Measures

- FORCURAM : SBOOK.FORCURAM

Calculated Measures

Restricted Measures

on attributes MANDT and AGENCYNUM.

Open node *Attribute Views* / *AT_AGENCY_DESCRIBED* in the Output structure to the right of the Modeler canvas. You can see that all attributes from the attribute view are included, and how the name clash on AGENCYNUM has been resolved. Save our analytic view and activate it.

Please invoke the data preview on the analytic view *AN_AGENCY_REV*. You will see that our attribute view has been linked into the attributes of table SBOOK which was used initially to create the analytic view *AN_AGENCY_ID_REV* and later *AN_AGENCY_REV*.

22 rows (3042 ms) Max rows: 1006 Refresh

Raw Data Distinct values Analysis

Enter your filter Filtered rows: 22/22 Add filter Save as file

| RB | NAME | RB | CITY | RB | COUNTRY | RB | CURRENCY | RB | AT_AGENCY_DESCRIBED_AGENCYNUM | RB | MANDT | RB | FORCURKEY | 12 | FORCURAM |
|----|-----------------------|----|-----------|----|---------|----|----------|----|-------------------------------|----|-------|----|-----------|----|--------------------|
| | Pink Panther | | Frankfurt | | DE | | EUR | | 00000093 | | 800 | | EUR | | 1364241.729999999 |
| | Your Choice | | Nuernberg | | DE | | EUR | | 00000100 | | 800 | | EUR | | 1355369.5100000002 |
| | Travel from Walldorf | | Berlin | | DE | | EUR | | 00000105 | | 800 | | EUR | | 1353477.35 |
| | Bella Italia | | Roma | | IT | | EUR | | 00000101 | | 800 | | EUR | | 1588944.2700000005 |
| | Honauer Reisen GmbH | | Neumarkt | | AT | | EUR | | 00000104 | | 800 | | EUR | | 1369901.1100000006 |
| | Caribbean Dreams | | Emden | | DE | | EUR | | 00000117 | | 800 | | EUR | | 1367463.8800000008 |
| | Everywhere | | Muenchen | | DE | | EUR | | 00000119 | | 800 | | EUR | | 1377253.8799999969 |
| | Special Agency Peru | | Stuttgart | | DE | | EUR | | 00000116 | | 800 | | EUR | | 1368918.8399999973 |
| | Bavarian Castle | | Dresden | | DE | | EUR | | 00000110 | | 800 | | EUR | | 1359611.2000000014 |
| | Happy Holiday | | Bremen | | DE | | EUR | | 00000120 | | 800 | | EUR | | 1374914.7099999979 |
| | Maxitrip | | Wiesbaden | | DE | | EUR | | 00000294 | | 800 | | EUR | | 1368047.5599999991 |
| | Fly Low | | Dresden | | DE | | EUR | | 00000122 | | 800 | | EUR | | 1597717.3299999994 |
| | No Return | | Koeln | | DE | | EUR | | 00000115 | | 800 | | EUR | | 1349118.0299999996 |
| | Up 'n' Away | | Hannover | | DE | | EUR | | 00000124 | | 800 | | EUR | | 1353964.2099999997 |
| | Fly & Smile | | Frankfurt | | DE | | EUR | | 00000188 | | 800 | | EUR | | 1090183.2800000028 |
| | Rainy, Stormy, Cloudy | | Stuttgart | | DE | | EUR | | 00000284 | | 800 | | EUR | | 1618936.1199999996 |

Our new analytic view can be used to display the revenue in Euros per agency but in a more comprehensible form than our first analytic view which could only be used to learn about the travel agency IDs instead of their names etc.

Please check the properties of the new analytic view. The default client should be set to 800 so that only data from this client is combined in our new star schema.

3.2.8 Calculated Attributes

In our next example we will do some calculation in our analytic view. Let's assume that we want to calculate the revenue as before. However, now we want to calculate it under the assumption that a 10 percent discount has been given on all bookings paid in Euros.

Please create a new analytic view in package *sflight* with the name *AN_AGENCY_EUR_DISCOUNT*. In the create dialog please choose *Copy From* and select *sflight / Analytic Views / AN_AGENCY_REV*.

Remember that we restricted the results to revenue made in Euros in our last example. We have to remove this restriction. Therefore right-click on attribute *FORCURKEY* in the Modeler pane and choose *Remove Filter*.

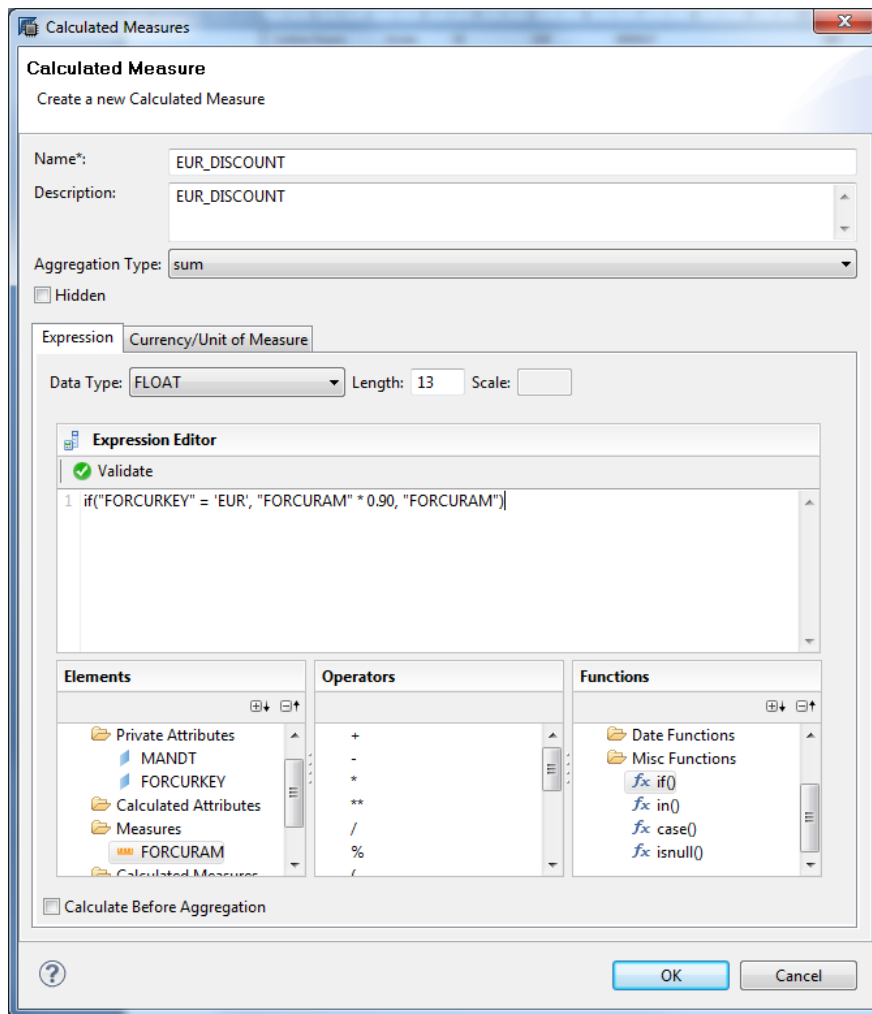
In the *Output* window on the right hand side of the Modeler canvas please right-click on *Calculated Measures* and select *New*. In the dialog window please fill in the following: Name = *EUR_DISCOUNT*, Data Type = *FLOAT*. In the Expression Editor please enter

```
if("FORCURKEY" = 'EUR', "FORCURAM" * 0.90, "FORCURAM")
```

This is equivalent to: If attribute FORCURKEY equals the string 'EUR' then attribute FORCURAM multiplied by 0.90 should be returned. Otherwise the original value of attribute FORCURAM should be used.

The expression entered manually can also be assembled by dragging and dropping the expression elements from the menus below the editor window.

Click on OK to save the expression.



Please save and activate the analytic view.

The data preview now displays an additional column labeled EUR_DISCOUNT according to our calculation expression.

49 rows (1919 ms) Max rows: 1006 Refresh

Raw Data Distinct values Analysis

Enter your filter Filtered rows: 49/49 Add filter Save as file

| RB | NAME | RB | CITY | RB | COUN... | RB | CURRE... | RB | AT_AGENCY... | RB | MANDT | RB | FORCURKEY | 12 | FORCURAM | 12 | EURO_DISCOUNT |
|----|--------------------|----|---------------|----|---------|----|----------|----|--------------|----|-------|----|-----------|----|--------------------|----|--------------------|
| | Kangeroos | | London | | GB | | GBP | | 00000109 | | 800 | | GBP | | 974418.130000008 | | 974418.130000008 |
| | All British Air... | | Vineland | | US | | USD | | 00000319 | | 800 | | USD | | 1242095.8800000064 | | 1242095.8800000064 |
| | Submit and ... | | Palo Alto | | US | | USD | | 00000304 | | 800 | | USD | | 1248131.5800000047 | | 1248131.5800000047 |
| | Fly Now, Pay... | | New York | | US | | USD | | 00000229 | | 800 | | USD | | 993660.4999999979 | | 993660.4999999979 |
| | Caribbean Dre... | | Emden | | DE | | EUR | | 00000117 | | 800 | | EUR | | 1367463.8800000008 | | 1230717.4920000008 |
| | Rainy, Storm... | | Stuttgart | | DE | | EUR | | 00000284 | | 800 | | EUR | | 1618936.1199999966 | | 1457042.507999997 |
| | Ben McClosk... | | Birmingham | | GB | | GBP | | 00000107 | | 800 | | GBP | | 854437.9400000063 | | 854437.9400000063 |
| | No Name | | Aachen | | DE | | EUR | | 00000121 | | 800 | | EUR | | 1358637.8900000008 | | 1222774.1010000007 |
| | Maxitrip | | Wiesbaden | | DE | | EUR | | 00000294 | | 800 | | EUR | | 1368047.5599999991 | | 1231242.8039999993 |
| | No Return | | Koeln | | DE | | EUR | | 00000115 | | 800 | | EUR | | 1349118.0299999996 | | 1214206.2269999997 |
| | Special Agen... | | Stuttgart | | DE | | EUR | | 00000116 | | 800 | | EUR | | 1368918.8399999973 | | 1232026.9559999977 |
| | Travel from ... | | Berlin | | DE | | EUR | | 00000105 | | 800 | | EUR | | 1353477.35 | | 1218129.615 |
| | Hitchhiker | | Issy-les-M... | | FR | | EUR | | 00000224 | | 800 | | EUR | | 1092759.0500000017 | | 983483.1450000015 |
| | Fly High | | Duesseldorf | | DE | | EUR | | 00000061 | | 800 | | EUR | | 1403469.9699999999 | | 1263122.9729999999 |
| | Ali's Bazar | | Boston | | US | | USD | | 00000111 | | 800 | | USD | | 1258801.9800000009 | | 1258801.9800000009 |

3.2.9 Graphical Calculation Views

There are several ways to create views in the SAP HANA database. We just learned about analytic and attribute views and how to combine them into a star schema which forms an analytic view itself. These views are completely processed by the column store. For more complex views we need to leverage the power of the calculation engine. These views are called calculation views.

As our next example we will fully recreate the example AN_AGENCY_REV as a graphical calculation view. Open package node *Content / sflight* in the Navigator view. Right-click on *Calculation Views* and choose *New ► Calculation View*. In the dialog window please enter the view name *GC_AGENCY_REV*. Click on *Next*.

Create Calculation View

Create a Calculation View

JOS (SYSTEM) Id8611 10

Name:* GC_AGENCY_REV

Description Description

☒ Create New

☐ Copy From Browse

View Type

☒ Graphical

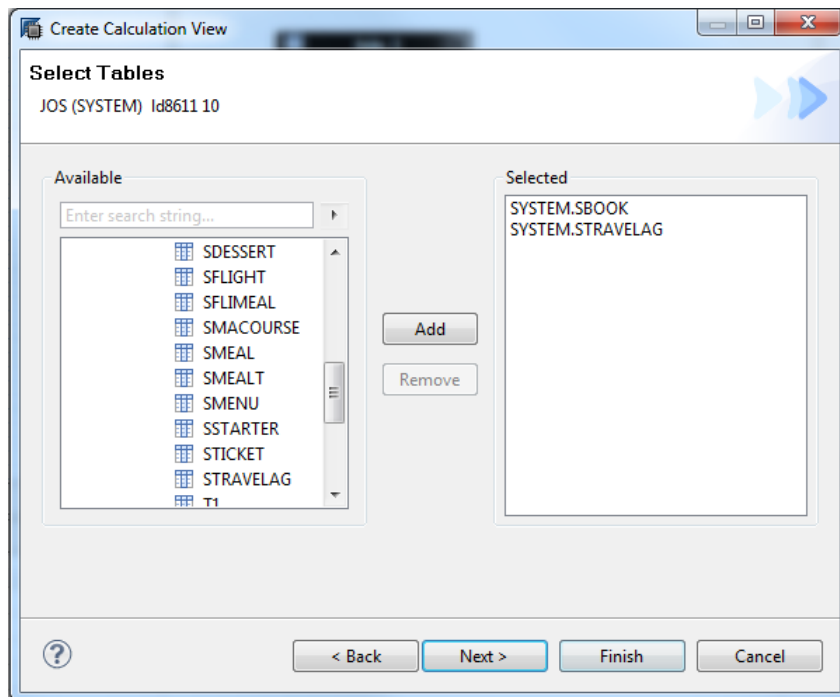
☐ SQL Script

Package sflight

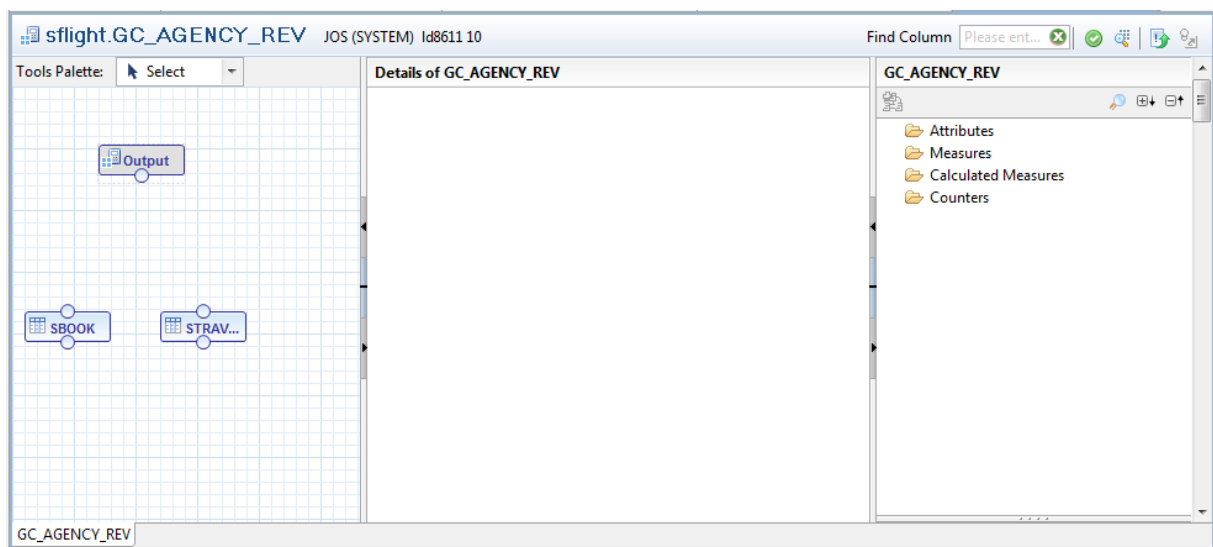
Default Schema

Run With Definer's Rights

Select tables *SBOOK* and *STRAVELAG* from the list of available tables and add these to the list of selected tables by marking them and clicking on the *Add* button. Then click on *Finish*.

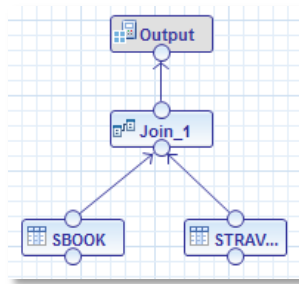


The modeler canvas will appear and it is split into two areas. The left one is used to connect tables with the output node by means of various operations which can be picked from the *Tools Palette* on top of that area. When the graphical calculation view modeler first starts up the *Output* node is selected. The selected node determines what is displayed in the canvas areas to the right.



In the left area we see three boxes or nodes labeled SBOOK, STRAVELAG, and Output. They represent the two tables used and the output from this graphical view. To remodel our previous example AN_AGENCY_REV we will have to join the two tables and connect the resulting attribute set with node Output.

Let's add the join operation from the tools palette. Please select Join and move the new node labeled Join_1 among the other three nodes SBOOK, STRAVELAG, and Output.

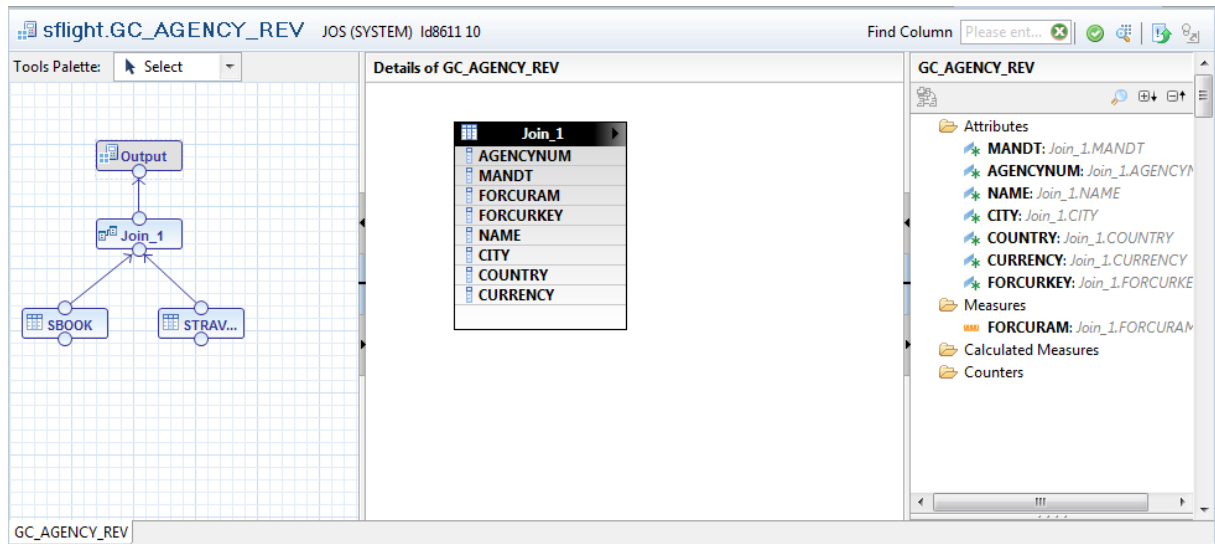


Now connect the four nodes in the following way. To connect SBOOK and Join_1 please click on the circle on top of SBOOK and draw a line to the circle in the bottom of Join_1. Connect the other nodes as shown in the image to the right.

Now click on Join_1 to display the details of the join in the middle area of the canvas. You will see the attributes of both tables SBOOK and STRAVELAG. These tables have to be joined on the common attribute AGENCYNUM. Click on AGENCYNUM in SBOOK and draw a connection to AGENCYNUM in STRAVELAG. Create another connection between the MANDT attributes of both tables.

Right-click on attribute MANDT in SBOOK and add it to the output of the join. Do the same with attributes FORCURAM and FORCURKEY in SBOOK and NAME, CITY, COUNTRY, and CURRENCY in STRAVELAG.

We have just defined the columns or attributes exported by the join operation. In the next step we have to assign them to the types of attributes which are exported by the Output node or result node of our calculation view. Click on *Output* in the left area of the canvas. In the middle area please select MANDT, AGENCYNUM, NAME, CITY, COUNTRY, CURRENCY, and FORCURKEY as attributes, and FORCURAM as a measure.



Save the calculation view and activate it.

Please note that the default client property which is common to every modeled view is not honored by calculation views. To compare the results of analytic view AN_AGENCY_REV and our new view CG_AGENCY_REV we therefore have to apply the respective filter in the data preview. Click on *Add filter* and select MANDT=800 in the *Column filters* pane.

98 rows (1280 ms) Max rows: 1006 Refresh

Raw Data Distinct values Analysis

Enter your filter Filtered rows: 49/98 Add filter Save as file

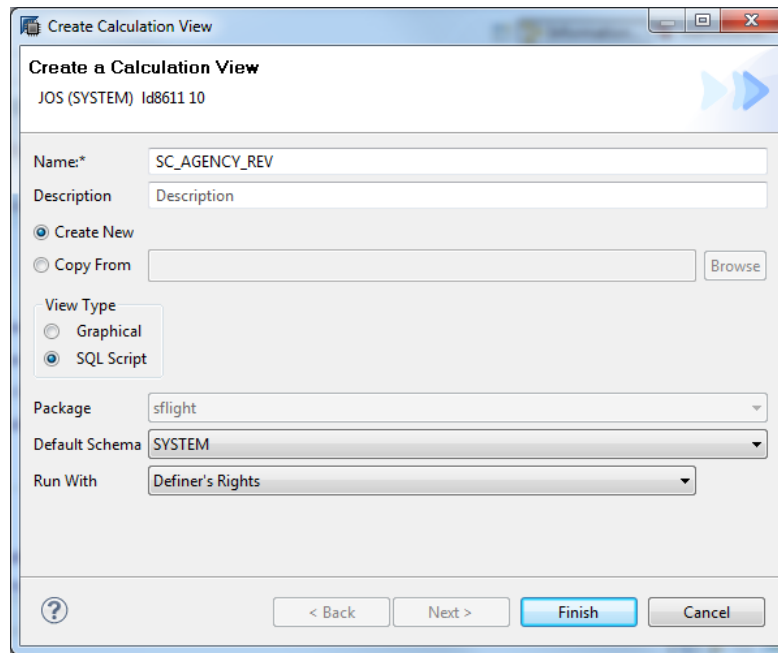
| RB | MANDT | RB | AGENCYNUM | RB | NAME | RB | CITY | RB | COU... | RB | CUR... | RB | FORC... | 12 | FORCURAM |
|-----|----------|----|----------------------|----|-------------|----|------|----|--------|----|--------|----|---------|----|-------------|
| 800 | 00000055 | | Sunshine Travel | | Rochester | | US | | USD | | USD | | | | 1243959,20 |
| 800 | 00000061 | | Fly High | | Duesseldorf | | DE | | EUR | | EUR | | | | 1403469,97 |
| 800 | 00000087 | | Happy Hopping | | Berlin | | DE | | EUR | | EUR | | | | 1414573,75 |
| 800 | 00000093 | | Pink Panther | | Frankfurt | | DE | | EUR | | EUR | | | | 1364241,73 |
| 800 | 00000100 | | Your Choice | | Nuernberg | | DE | | EUR | | EUR | | | | 1355369,51 |
| 800 | 00000101 | | Bella Italia | | Roma | | IT | | EUR | | EUR | | | | 1588944,27 |
| 800 | 00000102 | | Hot Socks Travel | | Sydney | | AU | | AUD | | AUD | | | | 1027026,94 |
| 800 | 00000103 | | Burns Nuclear | | Singapore | | SG | | SGD | | SGD | | | | 2252937,74 |
| 800 | 00000104 | | Honauer Reisen GmbH | | Neumarkt | | AT | | EUR | | EUR | | | | 1369901,11 |
| 800 | 00000105 | | Travel from Walldorf | | Berlin | | DE | | EUR | | EUR | | | | 1353477,35 |
| 800 | 00000106 | | Voyager Enterprises | | Stockholm | | SE | | SEK | | SEK | | | | 13218578,39 |
| 800 | 00000107 | | Ben McCloskey Ltd. | | Birmingham | | GB | | GBP | | GBP | | | | 854437,94 |
| 800 | 00000108 | | Pillepalle Trips | | Zuerich | | CH | | CHF | | CHF | | | | 2019691,05 |
| 800 | 00000109 | | Kangeroos | | London | | GB | | GBP | | GBP | | | | 974418,13 |
| 800 | 00000110 | | Bavarian Castle | | Dresden | | DE | | EUR | | EUR | | | | 1359611,20 |

Column filters: MANDT (1/2) - Equal 800

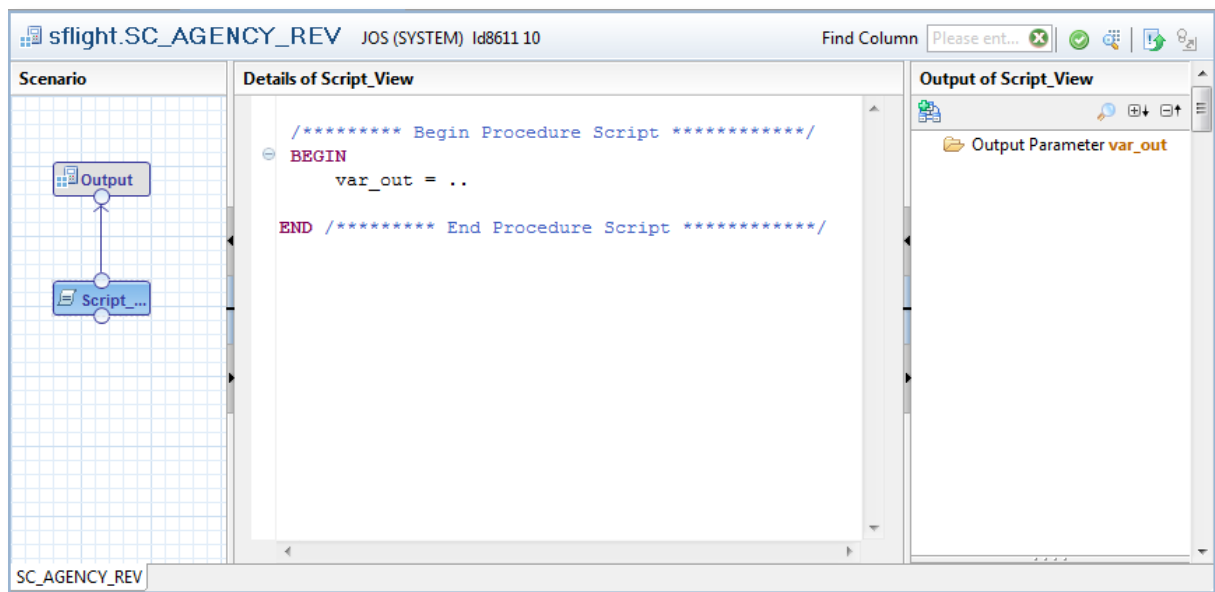
3.2.10 How to Create a Scripted Calculation View

Scripted calculation views are the last category of modeled views that we are going to discuss in this tutorial. The example is very similar to the previous one as we will create another implementation of the analytic view AN_AGENCY_REV but this time using SQLScript. For details about SQLScript please read the [SAP HANA SQLScript documentation](#).

Open package node *Content / sflight* in the Navigator view. Right-click on *Calculation Views* and choose *New ► Calculation View*. In the dialog window please enter the view name *SC_AGENCY_REV*. Choose *View Type SQLScript* and click on *Finish*.



Now click on the model node *Script_View* to open the script editor window labeled *Details of Script_View*.



The editor displays a fragment of SQLScript code which has to be completed by you. Please type the following into the editor window between BEGIN and END:


```
var_out = SELECT SBOOK.MANDT, SBOOK.AGENCYNUM,
           SUM(SBOOK.FORCURAM) as FORCURAM,
           FORCURKEY, NAME, COUNTRY, CURRENCY
FROM SYSTEM.SBOOK, SYSTEM.STRAVELAG
WHERE SBOOK.AGENCYNUM = STRAVELAG.AGENCYNUM
       AND SBOOK.MANDT = STRAVELAG.MANDT
GROUP BY SBOOK.MANDT, SBOOK.AGENCYNUM,
```

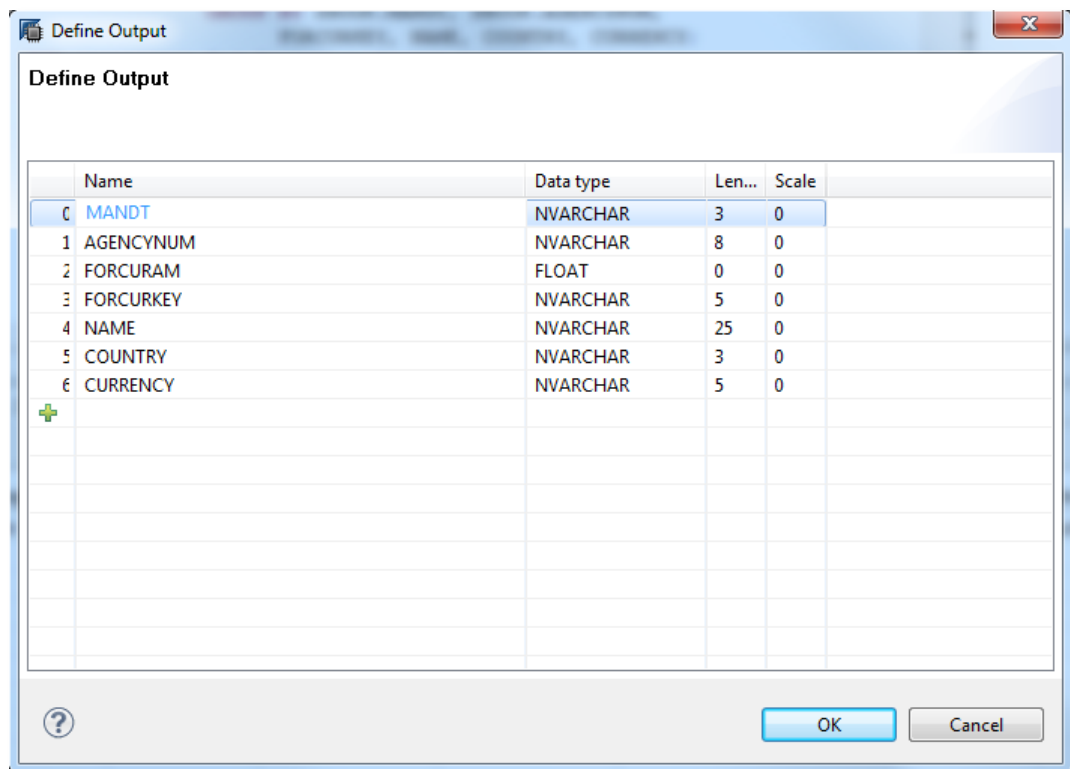
FORCURKEY, NAME, COUNTRY, CURRENCY;

This statement selects exactly the same data as in views AN_AGENCY_REV and GC_AGENCY_REV.

The attributes which are available in the return variable var_out are

- SBOOK.MANDT,
- SBOOK.AGENCYNUM,
- FORCURAM (sum aggregate),
- SBOOK.FORCURKEY,
- STRAVELAG.NAME,
- STRAVELAG.COUNTRY, and
- STRAVELAG CURRENCY.

These attributes have to be declared manually as output attributes of the Script View node. Please click on the icon  in the *Output of Script View* window and enter the attributes and their data types according to the next image. Click on the green plus sign to add another line. Click *OK* to save the settings.

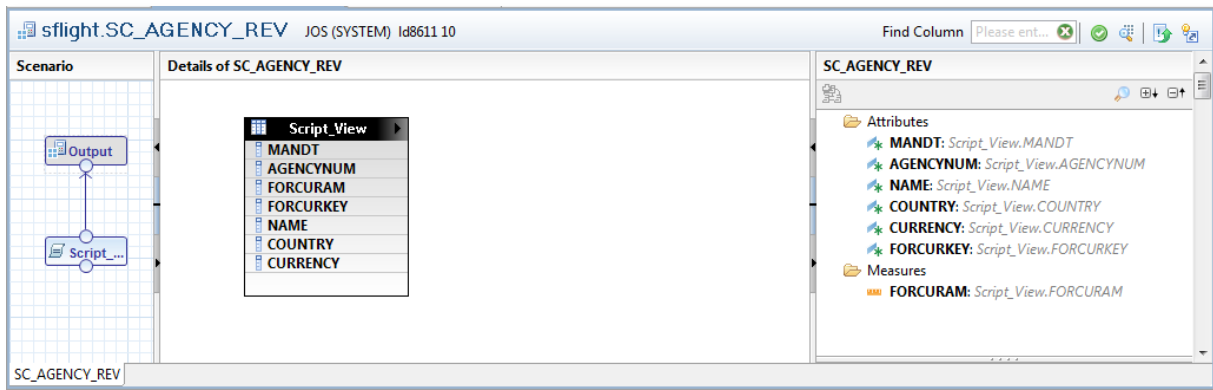


The image shows a 'Define Output' dialog box with a table for defining output attributes. The table has five columns: 'Name', 'Data type', 'Len...', and 'Scale'. There are seven rows in total, with the first six rows containing data and the seventh row being a blank row with a green plus sign in the first column. The data in the table is as follows:

| | Name | Data type | Len... | Scale |
|---|-----------|-----------|--------|-------|
| 0 | MANDT | NVARCHAR | 3 | 0 |
| 1 | AGENCYNUM | NVARCHAR | 8 | 0 |
| 2 | FORCURAM | FLOAT | 0 | 0 |
| 3 | FORCURKEY | NVARCHAR | 5 | 0 |
| 4 | NAME | NVARCHAR | 25 | 0 |
| 5 | COUNTRY | NVARCHAR | 3 | 0 |
| 6 | CURRENCY | NVARCHAR | 5 | 0 |
| + | | | | |

At the bottom of the dialog box, there is a question mark icon on the left, and 'OK' and 'Cancel' buttons on the right.

Now that the node Script View exports these attributes, we have to assign them to the Output node. Please select the Script View attributes MANDT, AGENCYNUM, NAME, COUNTRY, CURRENCY, and FORCURKEY as attributes and FURCURAM as a measure. Save the view and activate it.



Please note again that the default client property common to every modeled view is not honored by calculation views. In the data preview click on *Add filter* and select MANDT=800 in the *Column filters* pane.

98 rows (780 ms)

Max rows: 1006

Refresh

Raw Data

Distinct values

Analysis

Enter your filter

Filtered rows: 49/98

Add filter

Save as file

| AB | MANDT | AB | AGENCYNUM | AB | NAME | AB | COUNTRY | AB | CURRENCY | AB | FORCURKEY | 12 | FORCURAM |
|-----|-------|----|-----------|----|----------------------|----|---------|----|----------|----|-----------|----|-------------|
| 800 | | | 00000055 | | Sunshine Travel | | US | | USD | | USD | | 1243959,20 |
| 800 | | | 00000061 | | Fly High | | DE | | EUR | | EUR | | 1403469,97 |
| 800 | | | 00000087 | | Happy Hopping | | DE | | EUR | | EUR | | 1414573,75 |
| 800 | | | 00000093 | | Pink Panther | | DE | | EUR | | EUR | | 1364241,73 |
| 800 | | | 00000100 | | Your Choice | | DE | | EUR | | EUR | | 1355369,51 |
| 800 | | | 00000101 | | Bella Italia | | IT | | EUR | | EUR | | 1588944,27 |
| 800 | | | 00000102 | | Hot Socks Travel | | AU | | AUD | | AUD | | 1027026,94 |
| 800 | | | 00000103 | | Burns Nuclear | | SG | | SGD | | SGD | | 2252937,74 |
| 800 | | | 00000104 | | Honauer Reisen G... | | AT | | EUR | | EUR | | 1369901,11 |
| 800 | | | 00000105 | | Travel from Walldorf | | DE | | EUR | | EUR | | 1353477,35 |
| 800 | | | 00000106 | | Voyager Enterprises | | SE | | SEK | | SEK | | 13218578,39 |
| 800 | | | 00000107 | | Ben McCloskey Ltd. | | GB | | GBP | | GBP | | 854437,94 |
| 800 | | | 00000108 | | Pillepalle Trips | | CH | | CHF | | CHF | | 2019691,05 |
| 800 | | | 00000109 | | Kangeroos | | GB | | GBP | | GBP | | 974418,13 |
| 800 | | | 00000110 | | Bavarian Castle | | DE | | EUR | | EUR | | 1359611,20 |
| 800 | | | 00000111 | | Ali's Bazar | | US | | USD | | USD | | 1258801,98 |
| 800 | | | 00000112 | | Super Agency | | GB | | GBP | | GBP | | 851754,52 |
| 800 | | | 00000113 | | Wang Chong | | RU | | USD | | USD | | 1252591,94 |

Column filters

MANDT (1/2) - Equal

800

Analytic and attribute views are restricted to simple data models, e. g. it is not possible to join two tables as a single data foundation. In these cases script views are the solution as it is possible to perform complex calculations in SQLScript. Please understand that the complexity of a calculation determines the response time of SAP HANA. In chapter 4 we will discuss various approaches to tune the SAP HANA views for high performance.

3.3 Using SAP HANA Studio to Execute SQL and SQLScript Statements

Many other database systems distinguish among databases, catalogs, and schemas, with a database forming a physical storage (typically in files), catalogs and schemas being merely namespaces (catalogs contain schemas). Often a private schema is created with every user account. The SAP HANA database does not use databases. This is not an issue as there is no need to distinguish between different physical storage because everything is kept in memory. A SAP HANA appliance consists of exactly one database which in turn has exactly one catalog. Schemas are separate namespaces; i. e. the same table or view name may appear in multiple schemas. It is possible to grant authorizations on schema, table or view level. Authorizations granted for a schema are not automatically propagated to all existing objects in a schema. However, if a user creates new object in a schema it inherits the authorizations the respective user has for the schema.

If a new user has been created for you, you should see a schema with the same name as your user name. This is not the case if you connected using the SYSTEM user. Please right-click on the respective schema node and select SQL Editor from the context menu. This opens a new view you can use to type in SQL statements and send them to your SAP HANA database. Please expand the respective schema node in the Navigator view, to view the objects contained in the schema.

You are now going to create your first table in your SAP HANA database. Please enter the following SQL statement into the SQL Editor view, either on one line or on several lines as shown:

```
create table local_climate (  
    location nvarchar(60),  
    temperature_low int,  
    temperature_high int,  
    annual_precipitation float  
);
```

To execute the statement please click on the green circle with the arrow above the text area. You should see a success message in the area under the SQL text area. This message also tells you the execution time of your SQL statement. You can see how easy it is to measure execution times which are necessary to tune your database schema and queries for highest performance. If you have access to an on-line version of this document you might copy SQL statements from the document to the SQL Editor view using the cut & paste function of your operating system.

Now we will populate our table with some real data. Please execute the following SQL statements. You can enter multiple statements separating them by the semicolon character (";") and execute them with one mouse click on the green circle.

```
insert into local_climate values ('Berlin', -2, 24, 570.7);  
insert into local_climate values ('Heidelberg', 3, 20, 745.0);  
insert into local_climate values ('Biel', -2, 25, 1203.0);  
insert into local_climate values ('Dublin', 5, 15, 769.0);  
insert into local_climate values ('Milano', -2, 29, 943.2);  
insert into local_climate values ('Vancouver', -1, 22, 1181.5);
```

To check what is in the table please execute

```
select * from local_climate;
```

You should now get a second tab labeled "Result (1)" in the SQL Editor view that displays the data you entered. Data in column ANNUAL_PRECIP for locations Berlin and Milano display many digits. This is due to the fact that, in general, real values cannot be stored with unlimited precision. SAP HANA chooses the most precise internal representation in these cases.

Please click into the Navigator view and press F5. This will refresh the list. Then, please click on the small triangle before node Tables in the schema you have chosen. You should see a new node labeled *LOCAL_CLIMATE*. It represents the table you just created. Double-click on the tree node. This will bring up another view in the same screen area as the SQL Editor view. It displays what SAP HANA knows about the table: its columns with names and types as well as the internal data types which were selected automatically. From this we learn that SQL data types are mapped to internal SAP HANA data types that have been chosen to allow for efficient storage in memory. Above the tabular

display there are three fields labeled Table Name, Schema, and Type. From the latter we learn that your table LOCAL_CLIMATE was created in the row store which is the default.

Let us now delete the table and recreate it in the column store. To do that, execute the following statements:

```
drop table local_climate;  
create column table local_climate (  
    location nvarchar(60),  
    temperature_low int,  
    temperature_high int,  
    annual_precipitation float  
);
```

Repeat the steps above to populate the table with data and to display its structure. You will see in the structure view that the table type is now Column Store. Please drop the table, recreate it in the row store, and populate it with data as described earlier. We will need this table in the row store for further demonstrations.

You may have noticed that in the Navigator view the new table was labeled LOCAL_CLIMATE while in the SQL statements you used "local_climate". SAP HANA SQL does not distinguish between upper and lower case in SQL keywords and unquoted names. If you wish to make use of mixed-case names please put those names in double quotes, e. g. "LOCAL_CLIMATE" and "Local_Climate" will be distinct names.

Now for some SQLScript, which is the SQL extension language built into SAP HANA for fast in-memory computing. Please execute the following statement which creates a new table named PRECIPITATION with two columns with the same name and type as the respective columns in table CLIMATE. Table PRECIPITATION will be used as a template later.

```
create table precipitation (  
    location nvarchar(60),  
    annual_precipitation real  
);
```

The next statement to execute is actually some SQLScript code. It declares a procedure which is called by some statements further on in the code (**call** ...). The procedure has two input parameters, LOW_TEMP_MIN and LOW_TEMP_MAX, and one output parameter, PRCPT. Both input parameters are of type integer while the output parameter has the same structure as table PRECIPITATION. The procedure selects from the columns LOCATION and ANNUAL_PRECIP of table LOCAL_CLIMATE, only those rows with a value in column LOW_TEMP in the interval [LOW_TEMP_MIN ... LOW_TEMP_MAX] and puts the result of this query into output parameter PRCPT.

```
create procedure "getColdPrecipitation" (  
    in low_temp_min integer,  
    in low_temp_max integer,  
    out prcpt precipitation  
)  
language sqlscript reads sql data as  
begin  
    prcpt = select location, annual_precipitation from local_climate  
    where :low_temp_min <= temperature_low
```

```
and temperature_low <= :low_temp_max;  
end;
```

Please execute the following statement which creates a temporary table named PRC using table PRECIPITATION as a template:

```
create global temporary column table prc as table precipitation;
```

Now we call the procedure defined earlier to fill table PRC:

```
call "getColdPrecipitation" (0, 5, prc) with overview;
```

You will notice that a new tab appears in the SQL Editor view. Click on it to see what is returned by getColdPrecipitation.

SQLScript procedures return a table with the local output parameter names in the first column and the assigned variables or tables in the second column.

Finally, please execute

```
select * from prc;
```

You will observe that the only rows of table CLIMATE to be selected and transferred into table PRC had values in column LOW_TEMP between 0 and 5. As described earlier you may enter all statements at once into the SQL Editor and execute them with just one click on the green circle. If you mark a portion of the text and click on the green circle, only the marked text will be sent to the SAP HANA database for execution.

To learn more about SQLScript please refer to the *SAP HANA Database SQLScript* documentation.

3.4 How to Display the Query Plan

The query plan of an SQL statement explains which basic operations have to be performed in order to execute that statement. Basic operations are e. g. table scans, index usage, application of filters, and calculation of joins. In the following section we will use a join of tables CLIMATE and PRC to demonstrate the query plan explanation feature of SAP HANA.

Please execute the following statement:

```
select local_climate.* from climate, prc  
where local_climate.location = prc.location;
```

This statement selects and displays all complete rows from CLIMATE with a value in column LOCATION that also appears in the corresponding column in table PRC. Clicking on the tab labeled "Result (1)" displays the query result which contains the two rows for locations Heidelberg and Dublin.

Please mark the above statement in the SQL editor, right-click on the marked text, and select "Explain Plan". This recreates tab "Result (1)" with a list of operators executed and some details. Basically it is the call stack of the SAP HANA SQL processor displayed in reverse. This example is not very instructive. Let us redo these actions with a filter applied to the select statement. Please execute and explain the following statement

```
select local_climate.* from local_climate, prc
where local_climate.location = prc.location
      and prc.location = 'Dublin';
```

After execution the "Result (1)" tab displays a single row for location Dublin. How this data is calculated is explained by the query plan. Right-clicking on the statement and executing "Explain Plan" fills the tab "Result (1)" with lots of data.

The first operation listed is a row search over a number of rows or tuples with the fields LOCAL_CLIMATE.LOCATION, LOCAL_CLIMATE.LOW_TEMP, LOCAL_CLIMATE.TEMPERATURE_HIGH, LOCAL_CLIMATE.ANNUAL_PRECIPITATION. These are the fields required by the "select local_climate.*" part of the above statement. This operation assembles the tuples for display or other consumption that were calculated by the next operation in the list.

The next operation is a hash join. Rows selected from both tables are joined using a hash function to compare the LOCATION values. Using hash values instead of the original character strings takes less time especially for large data sets. So this comparison type is chosen by default.

Operation no. 3 is a table scan over the LOCAL_CLIMATE table to find all rows with location Dublin. A table scan is necessary since no index was created for column LOCATION. Therefore each row of table CLIMATE has to be checked against the filter condition "LOCAL_CLIMATE.LOCATION = 'Dublin'". Why is table LOCAL_CLIMATE checked against location Dublin, did we not specify that table PRC shall be checked? If you look two lines below you will find that table PRC is checked, too. The query optimizer of SAP HANA has found that in order to correctly calculate the join the filter condition has to be applied to both tables. Otherwise rows from table LOCAL_CLIMATE would be included which contain data for locations other than Dublin.

Operation no. 4 is a column search on column PRC.LOCATION. Here we note a difference when we compare this to operation no. 3. Since LOCAL_CLIMATE was (re-)created in the row store and PRC resides in the column store different operations were chosen to select the respective tuples from both tables.

Operation no. 5 is simply the application of the filter condition that, in the row store, appears separated from the column search operation while the row store (see operation no. 3) lists both the scan operation and the use of the filter condition as one operation.

The last operation looks up the structure of table PRC in the data dictionary.

If you scroll the operator table to the right you will find that in column PARENT_OPERATOR_ID operations no. 3 and 4 were both executed as part of operation no. 2 which is the calculation of the join. This is reasonable since a join combines data from several tables.

Let us now create an index on column LOCAL_CLIMATE.LOCATION. Please execute

```
create index climate_loc on local_climate (location);
```

and then again execute and explain the statement.

```
select local_climate.* from local_climate, prc
where local_climate.location = prc.location
      and prc.location = 'Dublin';
```

If you now look at the query plan you will see that the operations no. 2 (hash join) and 3 (table scan) were replaced by a Coherent Pattern Biclustering (CPB) tree index join. This operation uses data indexed in a particular fashion (CPB) to select the respective tuples and calculate the join in one step.

In general it is a good idea to use indexes for tables as this allows for precise selection of data. This is particularly important when calculating joins of large tables. Then less tuples from the respective tables have to be joined which makes for a faster calculation.

3.4.1 Columns in the Query Plan

The table below describes the columns in the query plan display.

| Column name | Description |
|--------------------|---|
| OPERATOR_NAME | Name of an operator. Details are described in the following section. |
| OPERATOR_DETAILS | Details of an operator. Predicates and expressions used by the operator are shown here. |
| OBJECT_NAME | Name of database tables and views accessed by an operator. |
| SUBTREE_COST | Estimated cost of executing the subtree starting from an operator. The unit of this value is time in seconds, but the absolute value would not match the actual execution time. This value is only for relative comparison between different subtrees. |
| INPUT_CARDINALITY | Estimated input row count of an operator. This is available only for operators accessing tables and views directly. For column store, estimation is done in row granularity, but for row store, estimation is done in page granularity and there can be large error for small tables. |
| OUTPUT_CARDINALITY | Estimated output row count of an operator. |
| OPERATOR_ID | Unique ID of an operator. IDs are integers starting from 1. |
| PARENT_OPERATOR_ID | OPERATOR_ID of the parent of an operator. The shape of a query plan is a tree and the topology of the tree can be reconstructed using OPERATOR_ID and PARENT_OPERATOR_ID. PARENT_OPERATOR_ID of the root operator is shown as 0. |
| LEVEL | Level from the root operator. Level of the root operator is 1, level of a child of the root operator is 2 and so on. This can be utilized for output indentation. |
| POSITION | Position in the parent operator. Position of the first child is 0, position of the second child is 1 and so on. |

| | |
|-----------|---|
| TIMESTAMP | Date and time when the EXPLAIN PLAN command was executed. |
|-----------|---|

3.4.2 OPERATOR_NAME Column in Query Plans

Below is a list of column engine operators shown in the OPERATOR_NAME column of a query plan.

| Operator name | Description |
|---------------|---|
| COLUMN SEARCH | Starting position of column engine operators. Listed in OPERATOR_DETAILS are projected columns. |
| LIMIT | Operator for limiting number of output rows |
| ORDER BY | Operator for sorting output rows |
| HAVING | Operator for filtering with predicates on top of grouping & aggregation |
| GROUP BY | Operator for grouping & aggregation |
| DISTINCT | Operator for duplicate elimination |
| FILTER | Operator for filtering with predicates |
| JOIN | Operator for joining input relations |
| COLUMN TABLE | Information on accessed column table |
| MULTIPROVIDER | Operator for producing union-all of multiple grouping & aggregations |

Below is a list of **row engine** operators shown in the OPERATOR_NAME column.

| Operator name | Description |
|-------------------|--|
| ROW SEARCH | Starting position of row engine operators. Listed in OPERATOR_DETAILS are projected columns. |
| LIMIT | Operator for limiting number of output rows |
| ORDER BY | Operator for sorting output rows |
| HAVING | Operator for filtering with predicates on top of grouping & aggregation |
| GROUP BY | Operator for grouping & aggregation |
| MERGE AGGREGATION | Operator for merging results of multiple parallel grouping & aggregations |
| DISTINCT | Operator for duplicate elimination |

| | |
|---------------------------|--|
| FILTER | Operator for filtering with predicates |
| UNION ALL | Operator for producing union-all of input relations |
| MATERIALIZED UNION ALL | Operator for producing union-all of input relations with intermediate result materialization |
| BTREE INDEX JOIN | Operator for joining input relations through B-Tree index searches. Join type suffix can be added. For example, B-Tree index join for left outer join is shown as BTREE INDEX JOIN (LEFT OUTER). Join without join type suffix means inner join. |
| CPBTREE INDEX JOIN | Operator for joining input relations through CPB tree index searches. A join type suffix can be added. |
| HASH JOIN | Operator for joining input relations through probing hash table built on the fly. A join type suffix can be added. |
| NESTED LOOP JOIN | Operator for joining input relations through nested looping. A join type suffix can be added. |
| MIXED INVERTED INDEX JOIN | Operator for joining an input relation of row store format with a column table without format conversion using the inverted index of the column table. A join type suffix can be added. |
| BTREE INDEX SEARCH | Table access through B-Tree index search |
| CPBTREE INDEX SEARCH | Table access through CPB-Tree index search |
| TABLE SCAN | Table access through scanning |
| AGGR TABLE | Operator for aggregating base table directly |
| MONITOR SEARCH | Monitoring view access through search |
| MONITOR SCAN | Monitoring view access through scanning |

COLUMN SEARCH is a mark for the starting position of column engine operators and ROW SEARCH is a mark for the starting position of row engine operators.

For more information on the subject please refer to the *SAP HANA SQL Guide*.

3.5 Using the JDBC Driver

To access SAP HANA from Java programs you use a JDBC driver which can be embedded into your own programs. It has the following properties:

- "100 percent pure Java"
- Supports JDBC 4.0 Standard Extension API (DataSource and ConnectionPoolDataSource)

- Driver of type 4 (the connection to the database instance is established using the SAP HANA-specific network protocol)

3.5.1 Installing the Driver

The driver can be installed separately by the command

```
hdbinst.exe -a client (Microsoft Windows) or hdbinst -a client (Linux, UNIX).
```

After the installation the driver can be found at `C:\Program Files\sap\hdbclient\ngdbc.jar` on Windows platforms and at `/usr/sap/hdbclient/ngdbc.jar` on Linux and UNIX platforms.

Since the SAP HANA Studio was written in Java, the JDBC driver is installed along with the SAP HANA Studio and can be obtained from this installation. It can be found in

`<SAP HANA Studio installation directory>/plugins/com.sap.ndb.studio.jdbc_<version>.jar`

on Linux or UNIX platforms and on Windows platforms in

`<SAP HANA Studio installation directory>\plugins\com.sap.ndb.studio.jdbc_<version>.jar`

3.5.2 Prerequisites

You have installed a Java platform:

- Sun Java JRE 1.4, or
- Sun Java JRE 1.6 or, for the application development Sun Java JDK 1.6 (for JDBC 4.0)

3.5.3 Integration of the JDBC Driver

Please add the path for the driver file `ngdbc.jar` to the `CLASSPATH` environment variable so that the Java platform can find and load the JDBC driver.

3.5.4 Loading the JDBC Driver

Please use the `forName` method of the `Class` class in the Java platform:

```
Class.forName("com.sap.db.jdbc.Driver");
```

For more information about the `forName` method, see the documentation for the Java platform at sun.java.com.

When executing the Java application, the Java platform loads the JDBC driver and registers it automatically in the JDBC driver manager. If an exception of the class `java.lang.ClassNotFoundException` is triggered, the Java platform could not find the JDBC driver.

3.5.5 Connection URL

In the connection URL, you define the properties of the connection to the database instance. You can find more detailed information in the documentation on the Java platform, see sun.java.com.

Syntax

```
jdbc:sap://<database_computer>[:<port>][/?<option1>[&<option2>]...]
```

| | |
|-------------------------|--|
| <database_computer> | Name of the database computer |
| <port> | Port of the SAP in-memory computing engine instance. |
| <option1>,<option2>,... | Properties of the connection |

Example

jdbc:sap://localhost:30115/?autocommit=false

Using this connection URL, you set up a connection to the database instance on the local computer. The port number used by the SAP in-memory computing engine is 30115. The connection options select the auto commit mode to false.

3.5.6 JDBC 4.0 Standard Extension API

Provides the API for server side data source access and processing from the Java™ programming language. The java API specification can be found at javax.sql.

These classes implement the `javax.sql.DataSource` and related APIs.

| class name | comment |
|---|--|
| <code>com.sap.db.jdbcext.DataSourceSAP</code> | an implementation of the <code>javax.sql.DataSource</code> |
| <code>com.sap.db.jdbcext.ConnectionPoolDataSourceSAP</code> | an implementation of the <code>javax.sql.ConnectionPoolDataSource</code> |
| <code>com.sap.db.jdbcext.PooledConnectionSAP</code> | an implementation of the <code>javax.sql.PooledConnection</code> |

3.5.7 JDBC Trace

You can activate the JDBC trace to find errors while your application is connected to a database via JDBC.

Features

When the JDBC trace is activated, the system logs the following information:

- JDBC API calls called by the JDBC application
- JDBC API call parameters
- Executed SQL statements and their results

Prerequisites

You are logged on as the operating system user who started (or will start) the JDBC application.

Note

- You always activate the JDBC trace for all JDBC applications that the current operating system user has started.

- Configuration changes have an effect on all JDBC applications that the current operating system user has started.

Procedure

Variant 1

1. Enter the following command on the command line: `java -jar <installation_path>\ngdbc.jar`
2. Select *Trace enabled*.
3. Choose your trace options.

SAP MaxDB JDBC Trace:

| Option | Command Line Option | Description |
|-------------------|---|---|
| Trace File Folder | - | Directory where the system writes the trace file When you do not configure a directory, the system writes the trace files to the temporary operating system directory (Microsoft Windows: %TEMP%, Unix and Linux: \$TMP) |
| Trace File Name | TRACE FILENAME [<path>]<file_name> | Sets the name of the trace file The system assigns each trace file an additional unique ID <id>: <file_name>_<id>.prt Default value for <file_name>: jdbctrace |
| Limit File Size | TRACE SIZE <size> [KB MB GB] | Limits the size of the trace data to <size> |
| - | TRACE SIZE UNLIMITED | Removes all size limitations for the trace file |
| Stop on Error | TRACE STOP ON ERROR <error_code> TRACE STOP ON ERROR OFF | Stops writing the JDBC trace when the error <error_code> appears or the first error occurs |

4. Choose *OK*.

Variant 2

5. To display the current configuration, enter `java --jar <installation_path>\sap\hdbclient\ngdbc.jar SHOW`
6. Enter your trace options: `java -jar <installation_path>\sap\hdbclient\ngdbc.jar <command_line_option>`
7. Enter `java -jar <installation_path>\sap\hdbclient\ngdbc.jar TRACE ON`

Variant 3

Specify the following option in the connection URL:

`trace=<file_name>`

More information: Connection URL

Result

The system writes the results of the JDBC trace to files in the trace directory.

3.5.8 Mapping SQL and Java Types

Conversions by setObject

| | TINYINT | SMALLINT | INTEGER | BIGINT | REAL | FLOAT | DOUBLE | DECIMAL | CHAR | VARCHAR | NCHAR | NVARCHAR | BINARY | VARBINARY | DATE | TIME | TIMESTAMP | BLOB | CLOB |
|----------------------|---------|----------|---------|--------|------|-------|--------|---------|------|---------|-------|----------|--------|-----------|------|------|-----------|------|------|
| String | ✓ | ✓ | ✓ | | | | | | | | | | | | | | | | |
| java.math.BigDecimal | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | | | | | | |
| Boolean | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | | | | | | |
| Byte | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | | | | | | |
| Short | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | | | | | | |
| Integer | ✓ | | | | | | | | | | | | | | | | | | |
| Long | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | | | | | | |
| Float | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | | | | | | |
| Double | | | | | | | | | | | | | | | | | | | |
| byte[] | | | | | | | | | | | | | ✓ | ✓ | | | | | |
| java.sql.Date | | | | | | | | | ✓ | ✓ | ✓ | ✓ | | | ✓ | | ✓ | | |
| java.sql.Time | | | | | | | | | ✓ | ✓ | ✓ | ✓ | | | | ✓ | | | |
| java.sql.Timestamp | | | | | | | | | ✓ | ✓ | ✓ | ✓ | | | ✓ | ✓ | ✓ | | |
| Blob | | | | | | | | | | | | | | | | | | ✓ | |
| Clob | | | | | | | | | | | | | | | | | | | ✓ |

The mark ✓ means that the given Java object type may be converted to the given SQL type. This table shows the possible values for the parameter specifying a target JDBC type that is passed to the method `PreparedStatement.setObject` or `RowSet.setObject`. Note that some conversions may fail at run time if the value presented is invalid. Labels parameters.

4 Best Practices

The power of the SAP HANA database lies in its unprecedented performance and easy administration. Users of the analytic functions of the SAP HANA database will appreciate the very short response times. The short response times allow users to query the database in a highly interactive manner without the need for predefined reports. Best practices therefore all strive to minimize response times.

4.1 Features of the Column Storage Engine

Both the row and column storage engine each have a built-in query engine optimized for the respective storage method. Consequently each of the storage engines has its strengths when used for differing tasks. The row engine supports all features provided by SAP HANA SQL. The column engine, however, has been optimized for analytics and supports a narrower set of SQL features. These features are much faster than the corresponding functions of the row store. Since not all features provided by SAP HANA SQL are natively supported by the column engine, the query optimizer sometimes creates an execution plan that involves both storage engines even if the data reside completely in the column store. On the other hand it is sometimes desirable to transfer intermediate results from the row store to the column store because subsequent calculation steps can be processed much faster there. As a general rule of thumb we recommend to use the "Explain Plan" feature of the SAP HANA studio to explore execution of critical SQL statements and optimize the data model and SQL statements accordingly.

The following list explains the features of the column query engine. The column engine can natively execute queries that are logically equivalent to queries that can be expressed in the following format.

```
SELECT <column engine expressions separated by comma>
FROM <column engine FROM specifications separated by comma>
  [ WHERE <column engine filter condition> ]
  [ GROUP BY <column engine expressions separated by comma> ]
  [ HAVING <column engine filter condition> ]
  [ ORDER BY <column engine expressions with optional
                                     ASC | DESC separated by comma> ]
  [ LIMIT <any expression> [OFFSET <any expression> ] ]
```

A column engine FROM specification can be one of the following:

- <table reference>
- (**SELECT** * **FROM** <table reference> **WHERE** <column engine filter condition>)
- <column engine FROM specification> **INNER JOIN** <column engine FROM specification> **ON** <column engine join condition>
- <column engine FROM specification> **LEFT OUTER JOIN** <column engine FROM specification> **ON** <column engine join condition>
- <column engine FROM specification> **RIGHT OUTER JOIN** <column engine FROM specification> **ON** <column engine join condition>
- <column engine FROM specification> **FULL OUTER JOIN** <column engine FROM specification> **ON** <column engine join condition>

The column engine supports following join conditions.

- <column reference from one side> = <column reference from the other side>
 - Data types of two columns should be the same. Decimals with different precisions or scales are also treated as different data types.
- AND operations between column engine join conditions
 - For outer joins, cyclical joining is not supported. For example, $T.a = S.a$ AND $T.b = R.b$ is not supported if there is already a join condition between S and R.

The column engine supports the following filter conditions

- Binary comparison (=, <>, <, >, <= and >=) between column engine expressions
- LIKE, IS NULL and IN predicate on a column engine expression
- AND, OR, NOT operations over column engine filter conditions

The column engine supports the following scalar expressions.

- Constant values
- Binary arithmetic operation (+, -, * and /) between column engine expressions
- <column engine expression>
- String concatenation (||) between column engine expressions
- CASE <column engine expression> WHEN <column engine expression> THEN <column engine expression> ... [ELSE <column engine expression>] END
- CASE WHEN <column engine filter condition> THEN <column engine expression> ... [ELSE <column engine expression>] END
- Functions listed below with column engine expressions as arguments
 - TO_DECIMAL, TO_NUMBER, TO_BIGINT, TO_REAL, TO_DOUBLE, TO_CHAR, TO_NCHAR, TO_DATE, TO_TIMESTAMP and BINTOHEX/HEXTOBIN
 - Format strings are not supported for TO_CHAR, TO_NCHAR, TO_DATE and TO_TIMESTAMP
 - LTRIM | RTRIM | TRIM, LENGTH, SUBSTR, INSTR and LOWER | UPPER
 - WEEKDAY, DAYS_BETWEEN, SECONDS_BETWEEN, ADD_DAYS, UTCTOLOCAL, LOCALTOUTC, ISOWEEK and QUARTER
 - LN | LOG, EXP | POWER | SQRT, SIN | COS | TAN, ASIN | ACOS | ATAN, SINH | COSH and FLOOR | CEIL
 - NULLIF, NVL, NVL2 and COALESCE
 - EXTRACT(YEAR | MONTH FROM <column engine expression>)
- Functions without arguments are evaluated by the SQL parser. The evaluated results are passed to the column engine as constant values.
 - CURRENT_CONNECTION, CURRENT_SCHEMA, CURRENT_USER, SYSUID, CURRENT_DATE/ CURRENT_TIME/ CURRENT_TIMESTAMP and CURRENT_UTCDATE/ CURRENT_UTCTIME/ CURRENT_UTCTIMESTAMP
- Expressions equivalent to one of the listed above (e.g. CAST function)

The column engine supports the following aggregation expressions.

- MIN | MAX | COUNT | SUM | AVG ([DISTINCT] <column engine expression>)

Since SQL provides multiple possible ways to formulate a query, an SQL query that is not in the above format can be processed natively by the column engine if the query is equivalent to a query that can be represented in the correct format. Below are examples of such equivalence. Equivalence is denoted as \leftrightarrow .

```
SELECT * FROM T, S WHERE T.a = S.a
 $\leftrightarrow$ .SELECT * FROM T INNER JOIN S ON T.a = S.a

SELECT DISTINCT a, b, c FROM T
 $\leftrightarrow$ .SELECT a, b, c FROM T GROUP BY a, b, c

SELECT * FROM T WHERE EXISTS (SELECT * FROM S WHERE T.a = S.a)
 $\leftrightarrow$ .SELECT DISTINCT T.* FROM T INNER JOIN S ON T.a = S.a
```

Equivalent only if table T has a primary key.

```
SELECT * FROM T WHERE NOT EXISTS (SELECT * FROM S WHERE T.a =
S.a)
 $\leftrightarrow$ .SELECT T.* FROM T LEFT OUTER JOIN S ON T.a = S.a WHERE S.a
IS NULL
```

4.2 SQL Query Cost Estimation

The SQL optimizer weighs possible alternatives for query execution using the following cost estimation formula. The following tables list formulas to calculate the relative execution time for the search operations supported by the two storage engines. The estimated value is highly correlated with the actual execution time. Please note that this table cannot be used to calculate actual execution times which depend e. g. on hardware parameters like the number of CPU cores or CPU and bus clock rate. However, the tables can be used for relative comparison between alternatives methods to calculate an optimum search strategy or data model.

4.2.1 Row Search Cost Models

| Operation | Model | Coefficients |
|----------------------|--|--|
| Projection | $C \times \text{<output size>}$ | $C = 0.821 \mu\text{s}$ |
| Limit | $C \times \text{<output size>}$ | $C = 0.276 \mu\text{s}$ |
| Grouping/aggregation | $C_1 \times \text{<input size>} + C_2 \times \text{<output size>}$ | $C_1 = 2.063 \mu\text{s},$ $C_2 = 0.410 \mu\text{s}$ |
| Selection | $C_1 \times \text{<input size>} + C_2 \times \text{<output size>}$ | $C_1 = 0.521 \mu\text{s},$ $C_2 = 0.102 \mu\text{s}$ |
| Nested-loop join | $C_1 \times \text{<left input size>} \times \text{<right input size>} + C_2 \times \text{<output size>}$ | $C_1 = 0.474 \mu\text{s},$ $C_2 = 0.410 \mu\text{s}$ |
| Hash join | $C_1 \times \text{<left input size>} + C_2 \times \text{<right input size>} + C_3 \times \text{<output size>}$ | $C_1 = 0.728 \mu\text{s},$ $C_2 = 2.063 \mu\text{s},$ |

| | | |
|---------------------------|--|---|
| | | $C_3 = 0.675 \mu s$ |
| Index join | $C_1 \times \text{<left input size>} + C_2 \times \text{<left input size>} \times \log(\text{<right input size>})$ | $C_1 = 0.368 \mu s$, $C_2 = 0.028 \mu s$ |
| Mixed inverted index join | $C_1 \times \text{<left input size>} + C_2 \times \text{<left input size>} \times \log(\text{<right input size>}) + C_3 \times \text{<output size>}$ | $C_1 = 0.423 \mu s$, $C_2 = 0.251 \mu s$, $C_3 = 0.190 \mu s$ |
| Index search | $C_1 \times \text{<output size>} + C_2$ | $C_1 = 0.496 \mu s$, $C_2 = 2.145 \mu s$ |
| Table scan | $C \times \text{<input size>}$ | $C = 0.410 \mu s$ |
| Remote access | $C \times \text{<output size>}$ | $C = 8.986 \mu s$ |

4.2.2 Column Search Cost Models

| Operation | Model | Coefficients |
|--|--|--|
| Join | $\text{Weight} \times \text{<column engine cost estimation>}$ | $\text{Weight} = 0.0573368$ |
| Result materialization | $C \times \text{<output size>}$ | $C = 0.481 \mu s$ |
| Dictionary build | $C \times \text{<input size>}$ | $C = 5.461 \mu s$ |
| Grouping/aggregation | $C_1 \times \text{<input size>} + C_2 \times \text{<output size>}$ | $C_1 = 0.173 \mu s$, $C_2 = 0.194 \mu s$ |
| Grouping/aggregation (multi-column grouping) | $C_1 \times \text{<input size>} + C_2 \times \text{<output size>}$ | $C_1 = 0.236 \mu s$, $C_2 = 0.796 \mu s$ |

4.3 SQL Query Tuning Tips for the Column Engine

The SAP HANA column store engine has been optimized for the most frequent pattern of OLAP queries in the form of single-block SPJG (select, project, join and group by). This section lists high-cost features that are best avoided when formulating SQL queries to get the best performance out of the column engine.

Note that using high-cost features will not cause problems for small tables or small intermediate results extracted from big tables. Special care is needed when operations are executed on large tables or large intermediate results sets. Workarounds suggested here are just examples and possible workarounds will differ from application to application.

Many of the tips here suggest avoiding operations that are not natively supported by column engine. Please refer to the previous chapter to check the natively supported operations.

All tables used by examples in this chapter are column tables.

4.3.1 Expressions

Calculation. If a calculation is used that is not supported by the column engine, the operation based on the calculation is executed by the row engine which can lead to transfer of large intermediate result sets from the column engine to the row engine.

For example, the TO_DATE function is only supported by the column engine without a format string. The TO_DATE function can parse strings in the form of 'YYYYMMDD' and 'YYYY-MM-DD' by default. If a user knows that the source strings can be parsed by the TO_DATE function, it is better to use the TO_DATE function without a format string. Otherwise the function call and its parameters have to be transferred to the row engine to be processed. In the example below, date_string is a VARCHAR column.

| | |
|--------------|---|
| Slower query | <pre>SELECT * FROM T WHERE TO_DATE(date_string, 'YYYYMMDD') = CURRENT_DATE;</pre> |
| Faster query | <pre>SELECT * FROM T WHERE TO_DATE(date_string) = CURRENT_DATE;</pre> |

Using calculations that are natively supported by the column engine is faster than using none native calculations. However, operations with calculations are still slower than operations without calculations and it is better to avoid calculations at all if possible. Below is an example how to avoid calculations.

| | |
|--------------|---|
| Slower query | <pre>SELECT * FROM T WHERE (CASE WHEN a = 1 THEN 1 ELSE 2 END) = 2;</pre> |
| Faster query | <pre>SELECT * FROM T WHERE a <> 1 OR a IS NULL;</pre> |

If a calculation cannot be avoided by changing the SELECT statement, it may still be possible to avoid a calculation during query processing. This can be achieved by adding an automatically calculated column as shown in the following example. Adding an automatically generated column improves the query performance with the expense of increased insertion and update cost.

| | |
|----------------------|--|
| Slower query | <pre>SELECT * FROM T WHERE b * c = 10;</pre> |
| Faster query | <pre>SELECT * FROM T WHERE bc = 10;</pre> |
| DDL for faster query | <pre>ALTER TABLE T ADD (bc INTEGER GENERATED ALWAYS AS b * c);</pre> |

Please note that the specification of columns is needed for data insertions after adding generated column. This is required to avoid setting value for the generated column as shown in the following example.

| | |
|--|--|
| Insertion before adding generated column | <pre>INSERT INTO T VALUES (1, 2, 3);</pre> |
|--|--|

| | |
|---|---|
| Insertion after adding generated column | <pre>INSERT INTO T(a, b, c) VALUES (1, 2, 3);</pre> |
|---|---|

Implicit type casting. SAP HANA can perform type casts implicitly even if the user did not explicitly specify a type cast operation. For example, if there is a comparison between a VARCHAR value and a DATE value, the database system performs an implicit type cast operation to convert the VARCHAR value into a DATE value. Implicit type casting is done from lower-precedence types to higher-precedence types. You can find type precedence rules in the [SAP HANA SQL Reference](#).

Implicit type casting is a kind of calculation and has the same effect on performance as other calculations. For this reason, it is better to avoid implicit type casts if possible. If two columns are frequently compared by queries, it is better to create both columns with the same data type. One way to avoid the cost of implicit type casts is to use explicit type casts. If a VARCHAR column is to be compared with a DATE value and the developer knows that casting the DATE value into a VARCHAR value produces the desired result. It is better to save the cost of changing the type of the column by changing type of the value. For example, if the VARCHAR column contains only date values in the form of 'YYYYMMDD', it could be compared with a string generated from a DATE value in the form of 'YYYYMMDD'. In the example below, date_string is a VARCHAR column. Please note that comparison between strings is produces a different result than between dates in general. They are identical only in some specific cases.

| | |
|--------------|--|
| Slower query | <pre>SELECT * FROM T WHERE date_string < CURRENT_DATE;</pre> |
| Faster query | <pre>SELECT * FROM T WHERE date_string < TO_CHAR(CURRENT_DATE, 'YYYYMMDD');</pre> |

If implicit type casting cannot be directly avoided, adding a generated column at the expense of increased insertion and update cost may offer a more efficient work around. For example, a user can find '1', '1.0' and '1.00' stored in a VARCHAR column using the following queries.

In the example below, s is a VARCHAR column.

| | |
|----------------------|---|
| Slower query | <pre>SELECT * FROM T WHERE s = 1;</pre> |
| Faster query | <pre>SELECT * FROM T WHERE n = 1;</pre> |
| DDL for faster query | <pre>ALTER TABLE T ADD (n DECIMAL GENERATED ALWAYS AS s);</pre> |

4.3.2 Join

Non-Equijoin Predicate. The column engine does not natively support join predicates other than the equality condition. In other words, the column engine supports only equijoins natively. Join predicates connected by OR, Cartesian products, comparisons, and joins without a join predicate are not natively supported. In these cases the row engine will be invoked and the intermediate data is

transferred to that engine. If only non-equi-join predicates are used the row engine executes the join operation using a nested-loop algorithm. The nested-loop algorithm, as well as reformatting of intermediate data can be costly if the intermediate results set is large.

For outer joins, if equi-join predicates are connected by AND with non-equi-join predicates, they are processed in the same way as non-equi-join predicates. For inner joins, if equi-join predicates are connected by AND with non-equi-join predicates, only equi-join predicates are used for join, and non-equi-join predicates are applied as filter predicates after joining. In this case, even if the join predicates as a whole are selective enough, the join operation may take considerable time if the equi-join predicates are not selective enough.

An example of rewriting a non-equi-join predicate into an equi-join predicate is shown below. In the example, M is a table containing first and last dates of months of interest.

| | |
|--------------|--|
| Slower query | SELECT M.year, M.month, SUM(T.ship_amount) FROM T JOIN M ON T.ship_date BETWEEN M.first_date AND M.last_date GROUP BY M.year, M.month; |
| Faster query | SELECT M.year, M.month, SUM(T.ship_amount) FROM T JOIN M ON EXTRACT(YEAR FROM T.ship_date) = M.year AND EXTRACT(MONTH FROM T.ship_date) = M.month GROUP BY M.year, M.month; |

Predicates Over Calculated Columns. Equi-join predicates over calculated columns are not natively supported by the column engine. If the calculation is natively supported by the column engine, the intermediate result from the child operation containing the calculation is reformatted and consumed by the column engine again. If the calculation is not natively supported by the column engine, the intermediate results from both child operations are reformatted and consumed by the row engine. In either case, there can be a performance impact if the amount of intermediate data is large. A possible way of avoiding such calculations is by using generated columns.

Below is an example of avoiding calculated join columns using generated columns.

| | |
|----------------------|---|
| Slower query | SELECT M.year, M.month, SUM(T.ship_amount) FROM T JOIN M ON EXTRACT(YEAR FROM T.ship_date) = M.year AND EXTRACT(MONTH FROM T.ship_date) = M.month GROUP BY M.year, M.month; |
| Faster query | SELECT M.year, M.month, SUM(T.ship_amount) FROM T JOIN M ON T.year = M.year AND T.month = M.month GROUP BY M.year, M.month; |
| DDL for faster query | ALTER TABLE T ADD (year INTEGER GENERATED ALWAYS AS EXTRACT(YEAR FROM T.ship_date)); ALTER TABLE T ADD (month INTEGER GENERATED ALWAYS AS EXTRACT(MONTH FROM T.ship_date)); |

Filter Predicate Inside Outer-Join Predicate. The column engine does not natively support filter predicates inside outer-join predicates. Filter predicates over the right-hand side of a left outer join and filter predicates over the left-hand side of a right outer join are exceptions because shifting those predicates to the selections invoked by the join operation produces the same results.

If filter predicates are used inside outer join predicates and they cannot be shifted below the join operation, the row engine executes the join operation after reformatting intermediate results from both child selections. An example of rewriting such filter predicates into equijoin predicates using a generated column is shown below.

| | |
|----------------------|--|
| Slower query | SELECT * FROM T LEFT JOIN S ON T.a = S.a AND T.b = 1; |
| Faster query | SELECT * FROM T LEFT JOIN S ON T.a = S.a AND T.b = S.one; |
| DDL for faster query | ALTER TABLE S ADD (one INTEGER GENERATED ALWAYS AS 1); |

Cyclic Join. The column engine does not natively support join trees that have cycles in join edges if an outer join is involved in the cycle. If there is such a cycle involving an outer join, the result from a child of the join that completes the cycle is materialized (reformatted) to break the cycle. Cyclic inner joins are natively supported by the column engine, but it is better to avoid them because their performance is inferior to acyclic inner joins. One way of breaking such cycles is to move some of the columns involved in the join to different tables by changing the schema. Below is an example. For the acyclic join in the example, the NATION column of the SUPPLIER table is moved to the LINE_ITEM table.

| | |
|--------------|---|
| Cyclic join | SELECT * FROM supplier S, customer C, line_item L WHERE L.supp_key = S.key AND L.cust_key = C.key AND S.nation = C.nation; |
| Acyclic join | SELECT * FROM supplier S, customer C, line_item L WHERE L.supp_key = S.key AND L.cust_key = C.key AND L.supp_nation = C.nation; |

Filter Predicates Below Outer Join Accessing Multiple Tables. Filter predicates over multiple tables are not natively supported by the column engine if they appear under an outer join. If such filter predicate exists, the result from the child including the predicate is materialized before executing the join. Filter predicates over the left child of a left outer join and filter predicates over the right child of a right outer join are exceptions because moving those predicates upward to the outer join produces the same results. Such move is automatically done by SQL optimizer.

Below is an example of a filter predicate that triggers materialization of intermediate results. One way to avoid such materialization in the example would be maintaining a PRIORITY column in the LINE_ITEM table instead of in the ORDERS table.

```
SELECT * FROM customer C
LEFT JOIN (SELECT * FROM orders O
           JOIN lineitem L ON O.order_key = L.order_key
           WHERE L.shipmode = 'AIR' OR O.priority = 'URGENT'
) ON C.cust_key = L.cust_key
```

Projection of Constant or Calculated Values Below Outer Join. The column engine does not natively support constant or calculated value projection below outer joins. If such constant or calculated value projection exists, the result from the child including the projection is materialized before executing the join. Constant or calculated value projection over left child of left outer join or right child of right outer join are exceptions. This is because moving such projections above the join produces the same results and such move is automatically carried out by the SQL optimizer. A possible workaround to avoid materialization of intermediate results is by adding a generated column for the constants or calculated values.

Implicit Creation of Concatenated Column. When multiple columns are involved in a join, the column engine tries to create a concatenated column on the fly to process the join. The concatenated column is a new internal column that contains all information of the involved columns. The concatenated column is not created if the query is issued by an update transaction or if there is a lock conflict on concatenated column creation with another update transaction. When the concatenated column is not created, a less efficient execution strategy has to be used. To avoid run-time overhead of concatenated column creation or a sub-optimal execution strategy due to concatenated column creation failure, it is recommended to create the required columns beforehand. Below is an example of query that needs concatenated columns and DDLs to create the needed concatenated columns.

| | |
|---|---|
| Query that needs concatenated columns | SELECT M.year, M.month, SUM(T.ship_amount) FROM T JOIN M ON T.year = M.year AND T.month = M.month GROUP BY M.year, M.month; |
| DDL to create the needed concatenated columns | CREATE INDEX T_year_month ON T(year, month); CREATE INDEX M_year_month ON M(year, month); |

4.3.3 EXISTS / IN Predicate

Disjoint EXISTS Predicate. When an EXISTS or NOT EXISTS predicate is connected with other predicates through OR, it is internally mapped to a left outer join. As left outer join processing is generally more expensive than inner join processing, it is recommended to avoid such disjoint EXISTS predicates whenever possible. Below is an example how to avoid disjoint EXISTS predicates.

| | |
|--------------|---|
| Slower query | SELECT * FROM T WHERE EXISTS (SELECT * FROM S WHERE S.a = T.a AND S.b = 1) OR EXISTS (SELECT * FROM S WHERE S.a = T.a AND S.b = 2); |
| Faster query | SELECT * FROM T WHERE EXISTS (SELECT * FROM S |

| | |
|--|--|
| | WHERE S.a = T.a AND (S.b = 1 OR S.b = 2)); |
|--|--|

Filter Predicate Inside NOT EXISTS Predicate Accessing Multiple Tables. Since a NOT EXISTS predicate is processed by a left outer join, the tuning tip for outer joins also applies to the NOT EXISTS predicate.

Filter Predicate Inside a Disjoint EXISTS Predicate Accessing Multiple Tables. Since a disjoint EXISTS predicate is processed by a left outer join, the tuning tip for outer joins also applies to the disjoint EXISTS predicate.

NOT IN Predicate. Since the NOT IN predicate is much more expensive to process than NOT EXISTS, it is recommended to use NOT EXISTS instead of NOT IN whenever possible. Below is an example how to avoid a NOT IN predicate. Please note that the transformation in the example is not valid in general. It is valid only if there are no null values in the columns of interest. The transformation is automatically applied by the SQL optimizer for all columns of interest if NOT NULL constraints have been declared explicitly.

| | |
|---|--|
| NOT IN query | SELECT * FROM T WHERE a NOT IN (SELECT a FROM S); |
| Possibly equivalent query in some cases | SELECT * FROM T WHERE NOT EXISTS (SELECT * FROM S WHERE S.a = T.a); |

4.3.4 Set Operations

UNION ALL / UNION / INTERSECT / EXCEPT. Since UNION ALL, UNION, INTERSECT and EXCEPT are not natively supported by the column engine, avoiding them may improve performance. Below are examples how to avoid UNION, INTERSECT and EXCEPT. Please note that the transformations in the examples are not valid in general. They are valid only if there are no null values in the columns of interest.

| | |
|---|---|
| UNION query | SELECT a, b FROM T UNION SELECT a, b FROM S; |
| Possibly equivalent query in some cases | SELECT DISTINCT COALESCE(T.a, S.a) a, COALESCE(T.b, S.b) b FROM T FULL OUTER JOIN S ON T.a = S.a AND T.b = S.b; |

| | |
|---|---|
| INTERSECT query | SELECT a, b FROM T INTERSECT SELECT a, b FROM S; |
| Possibly equivalent query in some cases | SELECT DISTINCT T.a a, T.b b FROM T JOIN S ON T.a = S.a AND T.b = S.b; |

| | |
|---|---|
| EXCEPT query | SELECT a, b FROM T EXCEPT SELECT a, b FROM S; |
| Possibly equivalent query in some cases | SELECT DISTINCT T.a a, T.b b FROM T WHERE NOT EXISTS (SELECT * FROM S WHERE T.a = S.a AND T.b = S.b); |

4.4 SQLScript Recommended Practices

4.4.1 Reduce Complexity of SQL Statements

Variables in SQLScript enable you to arbitrarily break up a complex SQL statement into many simpler ones. This makes a SQLScript procedure easier to comprehend. To illustrate this point, consider the following query:

```
books_per_publisher = SELECT publisher, COUNT (*) AS cnt
                      FROM :books GROUP BY publisher;
largest_publishers = SELECT * FROM :books_per_publisher
                      WHERE cnt >= (SELECT MAX (cnt)
                                     FROM :books_per_publisher);
```

Writing this query as a single SQL statement either requires the definition of a temporary view (using WITH) or repeating a sub query multiple times. The two statements above break the complex query into two simpler SQL statements that are linked via table variables. This query is much easier to comprehend because the names of the table variables convey the meaning of the query and they also break the complex query into smaller logical pieces.

The SQLScript compiler will combine these statements into a single query or identify the common sub-expression using the table variables as hints. The resulting application program is easier to understand without sacrificing performance.

4.4.2 Identify Common Sub-Expressions

The query examined in the previous sub section contained common sub-expressions. Such common sub-expressions might introduce expensive repeated computation that should be avoided. For query optimizers it is very complicated to detect common sub-expressions in SQL queries. If you break up a complex query into logical sub queries it can help the optimizer to identify common sub-expressions and to derive more efficient execution plans. If in doubt, you should employ the EXPLAIN plan facility for SQL statements to investigate how the HDB treats a particular statement.

4.4.3 Multi-level Aggregation

Computing multi-level aggregation can be achieved using grouping sets. The advantage of this approach is that multiple levels of grouping can be computed in a single SQL statement.

```
SELECT publisher, name, year, SUM(price)
FROM :it_publishers, :it_books
WHERE publisher=pub_id AND crcy=:currency
GROUP BY GROUPING SETS ((publisher, name, year), (year))
```

To retrieve the different levels of aggregation the client typically has to examine the result repeatedly, e.g. by filtering by NULL on the grouping attributes.

In the special case of multi-level aggregations, SQLScript can exploit results at a finer grouping for computing coarser aggregations and return the different granularities of groups in distinct table variables. This could save the client the effort of reexamining the query result. Consider the above multi-level aggregation expressed in SQLScript.

```
books_ppy = SELECT publisher, name, year, SUM(price)
              FROM :it_publishers, :it_books
              WHERE publisher = pub_id AND crcy = :currency
              GROUP BY publisher, name, year;
books_py = SELECT year, SUM(price)
              FROM :books_ppy
              GROUP BY year;
```

It is a matter of developer choice and also the application requirements as to which alternative is the best fit for the purpose.

4.4.4 Understand the Costs of Statements

It is important to keep in mind that even though the SAP HANA database is an in-memory database engine and that the operations are fast, each operation has its associated costs and some are much more costly than others.

As an example, calculating a UNION ALL of two result sets is cheaper than calculating a UNION of the same result sets because of the duplicate elimination the UNION operation performs. The calculation engine plan operator CE_UNION_ALL (and also UNION ALL) basically stacks the two input tables over each other by using references without moving any data within the memory. Duplicate elimination as part of UNION, in contrast, requires either sorting or hashing the data to realize the duplicate removal, and thus a materialization of data. Various examples similar to these exist. Therefore it is important to be aware of such issues and, if possible, to avoid these costly operations. In general the HDB provides a statement to examine the costs of a SQL statement (see the SQL Reference Documentation for details):

You can get the query plan from the view SYS.QUERY_PLANS. The view is shared by all users. Here is an example of reading a query plan from the view.

```
EXPLAIN PLAN [ SET PLAN_ID = <plan_id> ] FOR <dml_stmt>

SELECT lpad(' ', level) || operator_name AS operator_name,
       operator_details, object_name, subtree_cost,
       input_cardinality, output_cardinality, operator_id,
       parent_operator_id, level, position
FROM sys.query_plans
WHERE PLAN_ID = <plan_id> ORDER BY operator_id;
```

Sometimes alternative formulations of the same query can lead to faster response times. Consequently reformulating performance critical queries and examining their plan may lead to better performance.

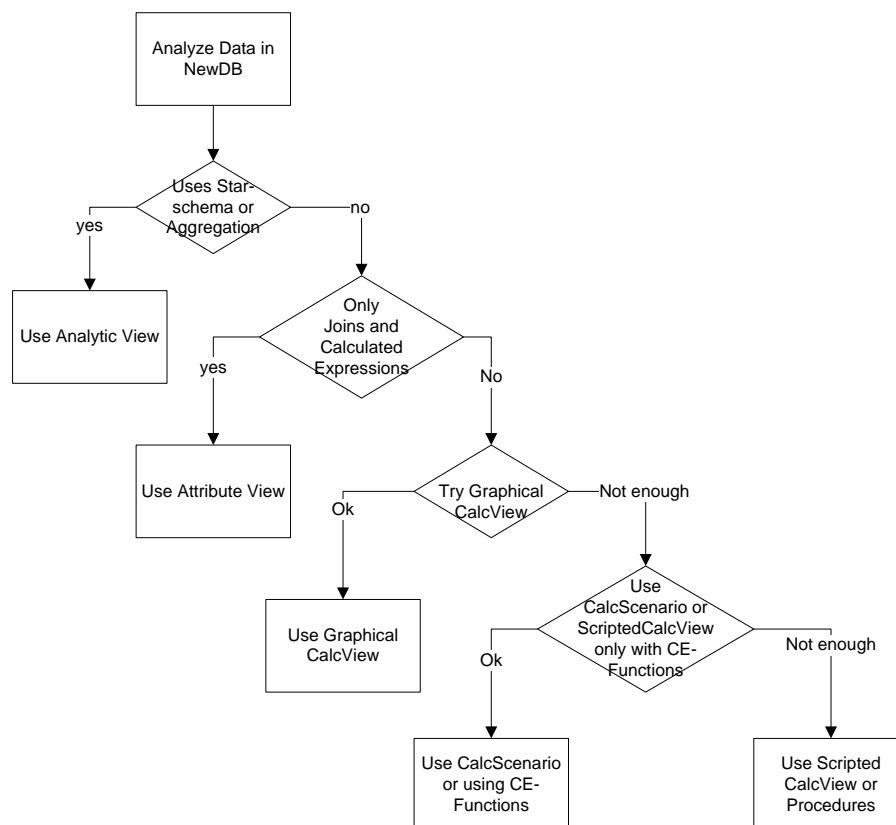
The HDB provides a library of application-level functions which handle frequent tasks, e.g. currency conversions. These functions can be expensive to execute, so it makes sense to reduce the input as much as possible prior to calling the function.

4.4.5 Exploit Underlying Engine

SQLScript can exploit the specific capabilities of the OLAP- and JOIN-Engine. For instance, if our data model is a star schema, it makes sense to model the data as an OLAP view. This allows the HDB to exploit the star schema when computing joins producing much better performance.

Similarly, if the application involves complex joins, it might make sense to model the data as a join view. Again, this conveys additional information on the structure of the data which is exploited by the HDB for computing joins.

Below you can find a proposal how to select amongst the various ways to implement logic. In the figure SQLScript appears only at the bottom leaf nodes. Using CE functions only or alternatively SQL statements only in a procedure still allows for many optimizations in the underlying engines. But when SQLScript procedures using imperative constructs are called by other programs, e.g. predicates to filter data early can no longer be applied. The performance impact of using these constructs must be carefully analyzed when performance is critical.



Finally, note that not assigning the result of an SQL query to a table variable will return the result of this query directly to the client as a result set. In some cases the result of the query can be streamed (or pipelined) to the client. This can be very effective as this result does not need to be materialized on the server before it is returned to the client.

4.4.6 Reduce Dependencies

One of the most important methods for speeding up processing in the HDB is a massive parallelization of executing queries. In particular, parallelization is exploited at multiple levels of granularity: For example, the requests of different users can be processed in parallel, and also single relational operators within a query are executed on multiple cores in parallel. It is also possible to execute different statements of a single SQLScript in parallel if these statements are independent of each other. Remember that SQLScript is translated into a dataflow graph, and independent paths in this graph can be executed in parallel.

From an SQLScript developer perspective, we can support the database engine in its attempt to parallelize execution by avoiding unnecessary dependencies between separate SQL statements, and also by using declarative constructs if possible. The former means avoiding variable references, and the latter means avoiding imperative features, e.g. cursors.

4.4.7 Simulating Function Calls in SQL Statements

The CALL statement cannot be used within a SQL statement. So, whenever we call a procedure on data that results from a transformation of input tables, the intermediate result has to be bound to a table variable.

In the special case of a read-only procedure that returns a single table you can create a procedure with WITH RESULT VIEW. It is then possible to embed a call to the procedure in a SQL SELECT statement as follows (see also the detailed discussion above in this document for details):

```
CREATE PROCEDURE ProcWithResultView(IN id INT, OUT o1 CUSTOMER)
LANGUAGE SQLSCRIPT READS SQL DATA WITH RESULT VIEW ProcView AS
BEGIN
    o1 = SELECT * FROM CUSTOMER WHERE CUST_ID = :id;
END;
```

It is then possible to query the result of a procedure as part of a SQL statement:

```
SELECT * FROM ProcView WITH PARAMETERS
        ('placeholder' = ('$$id$$', 5))
```

HDB in SPS2 offers limited ways to define scalar user defined functions which can be called from a SELECT or GROUP BY clause. Please ask your SAP contact for details.

4.4.8 Avoid Mixing Calculation Engine Plan Operators and SQL Queries

The semantics of relational operations as used in SQL queries and calculation engine operations are different. In the calculation engine operations will be instantiated by the query that is executed on top of the generated data flow graph. Therefore the query can significantly change the semantics of the data flow graph.

For example consider a calculation view that is queried using attribute publisher (but not year) that contains an aggregation node (i.e. CE_AGGREGATION) which is defined on publisher and year. The grouping on year would be removed from the grouping. Evidently this reduces the granularity of the grouping, and thus changes the semantics of the model. On the other hand, in a nested SQL query

containing a grouping on publisher and year this aggregation-level would not be changed if an enclosed query only queries on publisher.

Because of the different semantics outlined above the optimization of a mixed data flow using both types of operations is currently limited. Hence, one should avoid mixing both types of operations in one procedure.

4.4.9 Avoid Using Cursors

While the use of cursors is sometime required, they imply row-at-a-time processing. As a consequence, opportunities for optimizations by the SQL engine are missed. So you should consider replacing the use of cursors with loops by SQL statements as follows:

Read-Only Access

For read-only access to a cursor consider using simple selects or join:

```
CREATE PROCEDURE foreach_proc LANGUAGE SQLSCRIPT AS
    val decimal(34,10) := 0;
    CURSOR c_cursor1 FOR
        SELECT isbn, title, price FROM books;

BEGIN

    FOR r1 AS c_cursor1 DO
        val := :val + r1.price;
    END FOR;

END;
```

This sum can also be computed by the SQL engine:

```
SELECT sum(price) into val FROM books;
```

Computing this aggregate in the SQL engine may result in parallel execution on multiple CPUs inside the SQL executor.

Updates and Deletes

For updates and deletes, consider using the WHERE CLAUSE

```
CREATE PROCEDURE foreach_proc LANGUAGE SQLSCRIPT AS
    val INT := 0;
    CURSOR c_cursor1 FOR
        SELECT isbn, title, price FROM books;

BEGIN

    FOR r1 AS c_cursor1 DO
        IF r1.price > 50
        THEN
            DELETE FROM Books WHERE isbn = r1.isbn;
        
```

```
        END IF;  
    END FOR;  
  
END;
```

This delete can also be computed by the SQL engine:

```
DELETE FROM Books  
WHERE isbn IN (SELECT isbn FROM books WHERE price > 50);
```

Computing this in the SQL engine reduces the calls through the runtime stack of HDB and potentially benefits from internal optimizations like buffering or parallel execution.

Insertion into Tables

```
CREATE PROCEDURE foreach_proc LANGUAGE SQLSCRIPT AS  
    val INT := 0;  
    CURSOR c_cursor1 FOR  
        SELECT isbn, title, price FROM books;  
  
BEGIN  
  
    FOR r1 AS c_cursor1 DO  
        IF r1.price > 50  
        THEN  
            INSERT INTO ExpensiveBooks VALUES(..., r1.title, ...);  
        END IF;  
    END FOR;  
  
END;
```

This delete can also be computed by the SQL engine:

```
SELECT ..., title, ... FROM Books WHERE price > 50  
    INTO ExpensiveBooks;
```

Similar to updates and deletes, computing this statement in the SQL engine reduces the calls through the runtime stack of HDB and potentially benefits from internal optimizations like buffering or parallel execution.

4.4.10 Avoid using Dynamic SQL

Dynamic SQL is a very powerful way to express application logic. It allows for constructing SQL statements at execution time of a procedure. However, executing dynamic SQL is slow because compile time checks and query optimization must be done for every invocation of the procedure. So when there is an alternative to dynamic SQL using variables, this should be used instead.

Another related problem is security because constructing SQL statements without proper checks of the variables used might create a security vulnerability, e.g. via SQL injection. Using variables in SQL statements avoids these problems because type checks are performed at compile time, and parameters cannot inject arbitrary SQL code.

Summarizing potential use cases for dynamic SQL are:

| Feature | Proposed Solution |
|--|---|
| Projected attributes | Dynamic SQL |
| Projected literals | SQL + variables |
| FROM clause | SQL + variables; result structure must remain unchanged |
| WHERE clause values | SQL + variables |
| WHERE clause – attribute names & Boolean operators | Dynamic SQL |

4.4.11 Tracing and Debugging

To examine problems when creating SQLScript procedures, it is possible to increase the trace level for the indexserver. In this case HDB will not only trace error messages but will produce more detailed information.

For SQLScript the following trace information (i.e. diagnosis files) are relevant:

- Component Indexserver
calcengine, calcengineinstantiate, llang, or sqlscript
This traces information into the file indexserver_<host>.<port>.<xyz>.trc
- Component SQL
This traces information of SQL commands executed into the file
sqltrace_<host>_<port>_<xyz>.py
- Performance Trace

Furthermore the indexserver.ini configuration can be used to obtain more detailed information on the instantiated calculation engine models which are the runtime representation for SQLScript.

- Section: calcengine
 - Entry: tracemodels = yes
If set to yes, the json model before and after optimization is printed to trace file
 - Entry: show_intermediate_results = yes or topXXX
TopXXX will write the first XXX rows of the result set, yes writes all results
- Section: trace
 - Entry: row_engine = warning
This entry sets the trace level for the row engine of HDB which enables warnings for the SQLScript compiler including deprecation warnings; the default level is error.
- Runtime statistics in the performance tab of the administration view in the HDB studio.