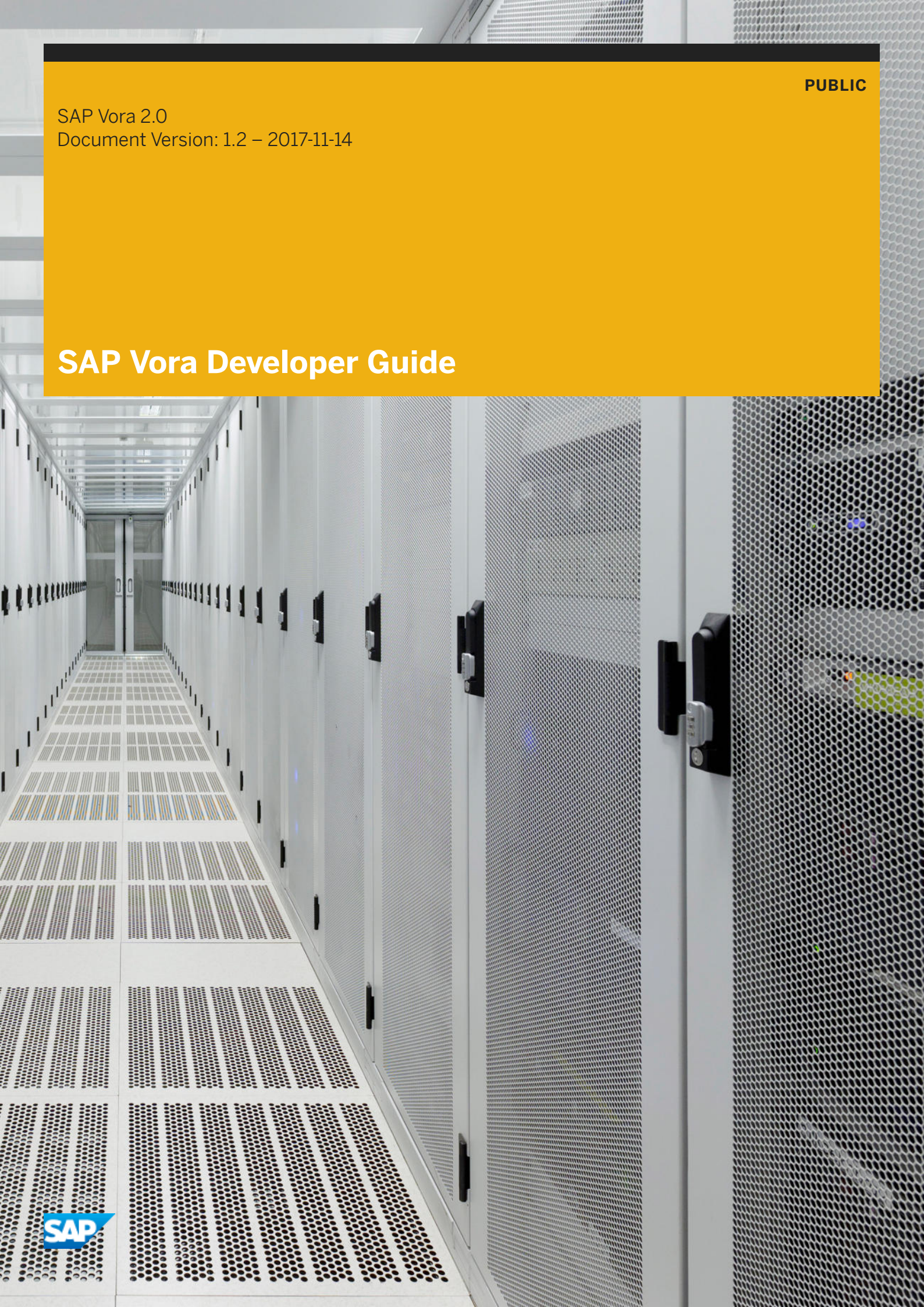
PUBLIC

SAP Vora 2.0
Document Version: 1.2 – 2017-11-14

SAP Vora Developer Guide



Content

1 Introduction to SAP Vora

SAP Vora is a distributed database system for big data processing. SAP Vora can run on a cluster of commodity hardware compute nodes and is built to scale with the size of the data by scaling up the compute cluster.

SAP Vora Engine Architecture

SAP Vora comes with support for various data types, such as relational data, graph data, collections of JSON (Java Script Object Notation) documents, and time series. Each of these data types is managed by a specialized engine, which has tailored internal data structures and algorithms to natively support and efficiently process that type of data.

- The relational in-memory store allows you to load relational data into main memory for fast access using code generation for query processing.
- The relational disk engine provides relational data processing for data sets that do not fit into main memory.
- The time series engine allows you to compress time series data using various compression techniques, while it provides algorithms like cross correlation or histogram computation on the compressed data.
- The graph engine lets you perform commonly used graph operations on its data and is particularly suited for handling complex read-only analytical queries on very large graphs.
- The document store supports rich query processing over JSON data.

SAP Vora can load and index data from external distributed data stores, such as HDFS and Amazon S3. The data is either kept in main memory for fast processing, or, in the case of the relational disk engine, it is indexed and stored on the hard disks which are locally attached to the compute nodes. Data loaded to SAP Vora can be partitioned by user-defined partitioning schemes, such as range, block, or hash partitioning. SAP Vora contains a distributed query processor, which can evaluate queries on the partitioned data. Metadata (that is, table schemas, partition schemes, and so on) is stored in SAP Vora's own catalog, which persists the catalog entries using SAP Vora's Distributed Log (DLog) infrastructure.

Apache Spark Integration

SAP Vora 2.0 ships both a Spark 1.6.x and Spark 2.x integration. The Spark 2.x integration emphasizes the separation between the SAP Vora database commands and Spark commands for a clearer and more intuitive usage of the SAP Vora functionality.

> **Note**
>
> Spark 1.6.x is deprecated and will be removed in a future version.

The following example shows how Spark 2.x can typically be used with SAP Vora. The example creates a table, adds a file to it, loads it into the database, and finally queries it from the Spark side:

Sample Code

```
import org.apache.spark.sql._
import sap.spark.vora._
val spark = new SparkSession(sc)
// This requires that your Spark Configuration points to a valid
// transaction coordinator, as explained in the Spark 2 section
val client = PublicVoraClientUtils.createClient(spark)

// Create table in Vora
client.execute("""CREATE TABLE "testTable" (name string, score int)""")
// Mark file to be loaded in Vora
client.execute("""ALTER TABLE "testTable" ADD DATASOURCE
               \|HDFS('hdfs://<host>:<port>/path/to/file.csv')""".stripMargin)
// Load files to table
client.execute("""LOAD TABLE "testTable"""")
// Create Dataset pointing to table in Vora
val table = spark.read.format("sap.spark.vora").option("table",
"testTable").load()
// Show contents of the table
table.show()
```

For Spark 1.6.x, the SAP Vora application programming interface (API) for applications running on SAP Vora is tightly integrated into Apache Spark SQL. Thus, to create, query, or modify database objects (tables, collections, graphs), you can write Spark programs using the SAP Vora Spark extension.

A typical example looks like this:

Sample Code

```
We assume a CSV file on HDFS that contains some data:
John,10
Jane,20
John,20
Jane,40
*/
import org.apache.spark.sql._
val sqlc = new SapSQLContext(sc)
sqlc.sql(
s"""
               CREATE TABLE testTable (name string, score integer)
               USING com.sap.spark.engines.relational
               OPTIONS (
                  files "/path/to/file.csv",
                  hdfsnamenode "<host>:<port>"
    )""".stripMargin)
/* query for entries with score > 10 */
sqlc.sql("SELECT name, age from testTable where score > 10").show
/* query for average score */
sqlc.sql("SELECT name, AVG(score) AS avgScore from testTable GROUP BY
name").show
```

In this example, a new table "testTable" is created in the SAP Vora relational in-memory engine and data from a CSV (comma-separated values) file is loaded into the table. The created table is also known to Spark. Therefore, you can make regular use of Spark SQL to query the table. These Spark queries are either evaluated using Spark mechanisms (after transferring data from the SAP Vora engines to Spark workers), or - whenever possible - are pushed for fast processing into the respective SAP Vora engine(s). The SAP Vora Spark Extension (implemented in the `SapSQLContext` or the `SapHiveContext`) takes care of analyzing Spark plans and pushing the query (or parts of it) into the SAP Vora engines.

Spark applications can interact with SAP Vora by referencing the SAP Vora Spark Extension JAR file delivered with SAP Vora through a Spark submit operation. External tools that make use of the Java Database Connectivity (JDBC) protocol can interact with SAP Vora through the Spark/Hive Thriftserver shipped with SAP Vora.

Spark Extensions

SAP Vora implements various Spark extensions that are relevant for business applications:

- Views: SAP Vora allows you to create views with a CREATE VIEW statement. The view's SQL statement is stored in the SAP Vora catalog.
- Annotations: Columns of views can be annotated with arbitrary key-value pairs. This additional metadata can be used to add further semantic information to a view column, for example, the information whether a column represents a dimension or a measure in an OLAP cube.
- Hierarchies: SAP Vora allows you to represent the content of tables as hierarchies (for example, organizational or location hierarchies) and to define queries along the relationship of the nodes in the hierarchy.
- Currency Conversion: SAP Vora can convert currencies using a user-defined Spark function.
- OLAP Models: Using hierarchies, it is possible to define OLAP models on data stored in Hadoop.
- Vora Tools: SAP Vora comes with a graphical UI that contains an SQL console, a graphical view modeler, and a data browser.

SAP Vora - SAP HANA Integration

SAP Vora integrates with SAP HANA. There are two approaches to combine or federate queries across SAP HANA and SAP Vora:

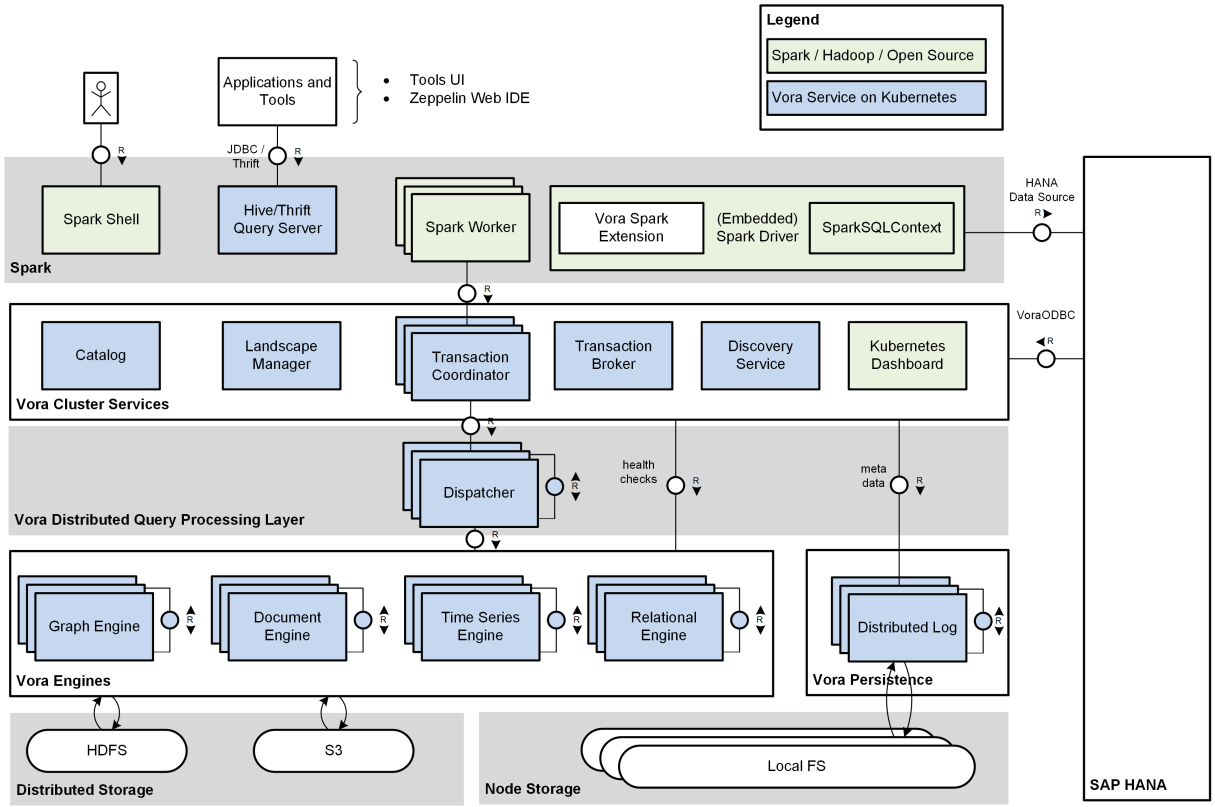
1. From SAP Vora to SAP HANA via the SAP HANA data source Spark extension
   The SAP Vora Spark extension contains a Spark data source implementation that allows you to interact with SAP HANA systems. Therefore, Spark applications can combine data from both SAP HANA and SAP Vora.
2. From SAP HANA to SAP Vora via SAP HANA SDA and the VoraODBC protocol
   SAP HANA SDA allows you to register SAP Vora as a remote source. Tables stored in SAP Vora can be exposed to the SAP HANA catalog as virtual tables. SAP HANA queries that involve virtual tables from the SAP Vora system lead to query execution on the SAP Vora system and an exchange of intermediate data with the SAP HANA system.

Kubernetes and Hadoop

SAP Vora is deployed to and runs in Kubernetes clusters. All SAP Vora services are containerized using Docker. Based on a cluster description specification (defined in a configuration file), SAP Vora clusters can be booted up in a set of Docker containers and run in a Kubernetes environment. Hardware and software failures are mitigated by failover mechanisms. The Hadoop-related parts of SAP Vora (that is, the SAP Vora Spark Extension) are deployed to Hadoop clusters.

1.1 System Architecture

The main components used in the SAP Vora Spark development environment are shown in the figure below.



Related Information

Component Overview [page 10]

1.2 Component Overview

The main components used in the SAP Vora Spark development environment are described briefly in the table below.

Component	Description
Tools UI	Runs on the AppServer.
Hive/Thrift Query Server	The Hive/Thrift Query Server handles incoming JDBC requests for Spark SQL queries. It is in charge of creating and handling the Spark context required to run Spark SQL queries.

Component	Description
Zeppelin	Apache Zeppelin is a web-based notebook that enables interactive data analytics. SAP Vora provides a Zeppelin interpreter that allows the SAP Vora Spark extensions to be used from Zeppelin.
Spark Shell	A command-line tool (Scala shell extension) for running Spark programs on the cluster.
Spark Driver	The Spark component provides a Spark SQLContext and an SQL API for client programs. It contains the Spark SQL query processor, which compiles SQL queries into RDDs (resilient distributed data sets). It also submits jobs to process RDDs on the cluster to YARN so that they are executed on the Spark executors. The Spark component contains the SAP Vora extension.
SAP Vora Spark Extension	The SAP Vora extension enhances Spark with additional functionality, such as DDL/SQL parsers, hierarchies, and OLAP modeling, and adds the semantics for persistent tables managed by the SAP Vora engines. The SAP Vora extension communicates with the following components: • Transaction Coordinator: To execute queries on data stored in the various SAP Vora engines. • Catalog Server: To store or fetch table metadata for tables and views created on the SAP Vora engines. • SAP HANA: To read table metadata (SAP HANA data source)
Catalog Server	The SAP Vora catalog provides a distributed metadata store. It stores changes to the metadata in the DLog Server. The catalog keeps metadata about the database objects (tables, time series, graphs, document collections) stored in the various Vora engines. It also stores view metadata for Spark views and engine views.
Transaction Coordinator	The transaction coordinator controls the execution of queries on the graph, disk, document, and time series engines on behalf of the SAP Vora Spark extension.
Transaction Broker	The transaction broker provides a read-write lock mechanism for DDL statements. This ensures that both the catalog and instances of the query engine keep a consistent state at all times.
Landscape Manager	The landscape manager controls data partitioning and placement across database engines.
DLog (Distributed Log)	The distributed log manager provides metadata persistence for the SAP Vora catalog.
Discovery Service	The Discovery Service manages the service endpoints in the cluster. These include the SAP Vora catalog, SAP Vora engines, DLog, and others. It runs health checks on all registered services at pre-defined intervals.
Spark Worker	The Spark worker runs Spark programs or parts of the programs and pushes query processing to the SAP Vora engines, provided data is loaded on the SAP Vora engines and SQL is supported.
SAP Vora Engines	The SAP Vora Engines provide specialized storage and processing capabilities for relational, graph, time series, and document data. The engines communicate with each other during data partitioning and query processing.
Dispatcher	The dispatcher runs the distributed query processing across the engines based on a query evaluation plan generated by the Transaction Coordinator.

Component	Description
HDFS	The Hadoop Distributed File System is a virtualized file system with replication and block-wise storage.
S3	The Amazon Simple Storage Service provides an object store, from which the Vora engines can read data.
Kubernetes Dashboard	The Kubernetes dashboard is used to run all Vora services on a cluster of compute nodes. It allows you to deploy, maintain, and monitor the Vora services.

Related Information

System Architecture [page 10]

2 Getting Started with Application Development

Set up your own Spark project on top of SAP Vora and run your applications as described below. The techniques shown here can also be used to customize the SAP Vora examples shipped with the SAP Vora data source.

> **Note**
>
> Spark 1.6.x is deprecated and will be removed in a future version.

Using SAP Vora Through the Spark Shell

The easiest way to interact with SAP Vora through standard Spark tooling is the Spark Shell.

You can start the Spark Shell with the SAP Vora Spark extensions as follows:

1. Make sure the `VORA_SPARK_HOME` environment variable points to the installation directory of the SAP Vora Spark extension (default: `/opt/vora-spark`), for example:

   ```
   export VORA_SPARK_HOME=/opt/vora-spark
   ```

2. Run the Spark Shell (the SAP Vora Spark extensions ship a wrapper script that makes the SAP Vora Spark extensions available in the Spark Shell):

   ```
   $> VORA_SPARK_HOME/bin/start-spark-shell.sh
   ```

3. Get access to the `SapSQLContext` that contains the SAP Vora Spark extensions:

   ```
   $> import org.apache.spark.sql._
   $> val sqlc = new SapSQLContext( sc )
   ```

Setting Up an SAP Vora Spark Project

For new projects with SAP Vora, we recommend using Maven as a build tool. As dependencies, you need at least `spark-sql_2.10` and `spark-sap-parent` (publicly available Spark extensions provided by SAP).

You can create a Maven artifact for `spark-sap-parent` using SAP Vora packaged as a JAR file. You can make use of the `maven-install-plugin` to install the JAR file in your local Maven repository and reference it in your project's `pom.xml` file.

Running Your SAP Vora Spark Project

After packaging your project as a JAR, you can run it on your cluster as follows:

Sample Code

```
spark-submit --jars $VORA_SPARK_HOME/lib/spark-sap-datasources.jar
--class org.you.YourMainClass yourapp.jar
```

Accessing Data Sources

SAP Vora provides the following data sources:

- `com.sap.spark.engines` for the specific SAP Vora engines (relational, graph, time series, document store, and disk). See *Working with Relational Tables and Views* and *Working with Graphs, Documents, and Time Series*.
- `com.sap.spark.hana` for the SAP HANA data source. See *Connectivity Between SAP HANA and SAP Vora*.

You can use a data source in Spark by creating a table or database object, such as graph, time series, or collection, using the Spark SQL command `CREATE TABLE` or `CREATE <object>`, together with the keyword `USING` and the respective data source.

Before you can create a table or object in Spark, you need to instantiate an `SapSQLContext` or an `SapHiveContext`. Both the `SapSQLContext` and the `SapHiveContext` are based on a `SparkContext` object.

Executing Queries

You can execute queries in the same way as in Spark SQL. SQL queries are compiled into RDDs (resilient distributed data sets), which support all functions needed for working with SQL, such as filters, projections, joins, and aggregations.

You can join tables regardless of their origin, but you should bear in mind that there may be differences in performance. For example, if you join a table from an SAP HANA data source with one from an SAP Vora engine data source, this might require data to be offloaded to Spark.

Related Information

Working with Relational Tables and Views [page 23]
Working with Graphs, Documents, and Time Series [page 58]
Connectivity Between SAP HANA and SAP Vora [page 146]

2.1 Samples

The SAP Vora data source is shipped with the code samples shown below. The samples are packaged in the `spark-sap-datasources.jar` file and can be executed using `spark-submit`.

The source code of the samples is also part of the SAP Vora distribution. You can find the code in `$VORA_SPARK_HOME/examples`.

List of Samples

Sample	Class
Load CSV into Spark and SAP Vora	com.sap.spark.vora.examples.LoadDataIntoVora
Load data into SAP HANA	com.sap.spark.vora.examples.LoadDataIntoHana
Load and join ORC and Parquet data	com.sap.spark.vora.examples.JoinOrcAndParquetData
Parent-child hierarchy	com.sap.spark.vora.examples.Hierarchies
Load CSV data from an Amazon S3 bucket	com.sap.spark.vora.examples.LoadDataFromS3
Hash partitioning	com.sap.spark.vora.examples.HashPartitioning
Hash partitioning with a fixed number of partitions	com.sap.spark.vora.examples.FixedHashPartitioning
Range-interval partitioning	com.sap.spark.vora.examples.RangeIntervalPartitioning
Document store engine	com.sap.spark.vora.examples.DocStoreEngine
Disk engine	com.sap.spark.vora.examples.DiskEngine
Graph engine	com.sap.spark.vora.examples.GraphEngine
Time series engine	com.sap.spark.vora.examples.TimeSeriesEngine

Related Information

Run the Samples [page 16]

2.1.1 Run the Samples

Execute the code samples as described below.

Prerequisites

- To run the samples, you need to have added the `spark-submit` script to your `$PATH` variable. The `spark-submit` script comes as part of the Spark installation.
- You have located the `spark-sap-datasources.jar` file. It is usually in `$VORA_SPARK_HOME/lib`. You need to pass it to the `spark-submit` script (see the steps below).

Procedure

1. Copy the sample data to the storage backend:

   ```
   spark-submit --class
   com.sap.spark.vora.examples.tools.CopyExampleFilesToHdfs /path/to/datasource/
   jarfile
   ```

   Depending on your cluster setup, you need to choose the correct master URL and storage backend. For example, if you are running the SAP Vora Developer Edition, you have a single node setup and should execute:

   ```
   spark-submit --class com.sap.spark.vora.examples.tools.CopyExampleFilesToHdfs
   --conf spark.vora.storagebackend=hdfs --master local /path/to/datasource/
   jarfile
   ```

   Note that since HDFS is the default storage backend, you don't actually have to set it.
2. Run the individual examples, for example as follows:

   ```
   spark-submit --class com.sap.spark.vora.examples.LoadDataIntoVora --master
   local /path/to/datasource/jarfile
   ```

   Again, you may need to specify the storage backend and the master URL.
3. Alternatively, open the source code and copy and paste the single commands one by one into the Spark shell.

Related Information

Samples [page 15]

2.2 Enabling Kryo Serialization Reference Tracking

By default, SAP Vora uses Kryo data serialization. The `spark.kryo.referenceTracking` parameter determines whether references to the same object are tracked when data is serialized with Kryo.

To avoid running into stack overflow problems related to the serialization or deserialization of too much data, you need to set the `spark.kryo.referenceTracking` parameter to true in the Spark configuration, for example, in the `spark-defaults.conf` file:

```
spark.kryo.referenceTracking=true
```

This setting is required irrespective of whether you are using the Spark shell or Thrift server. If you decide to use another serializer, this configuration is no longer needed, since it only influences the behavior of Kryo.

3 Working with the Spark Catalogs and the SAP Vora Catalog

Spark can connect to various systems from which to read and process data, for example, HDFS, Hive, SAP Vora, and so on. Each of these components has its own means of metadata management. Some care has to be given to managing metadata with Spark and SAP Vora.

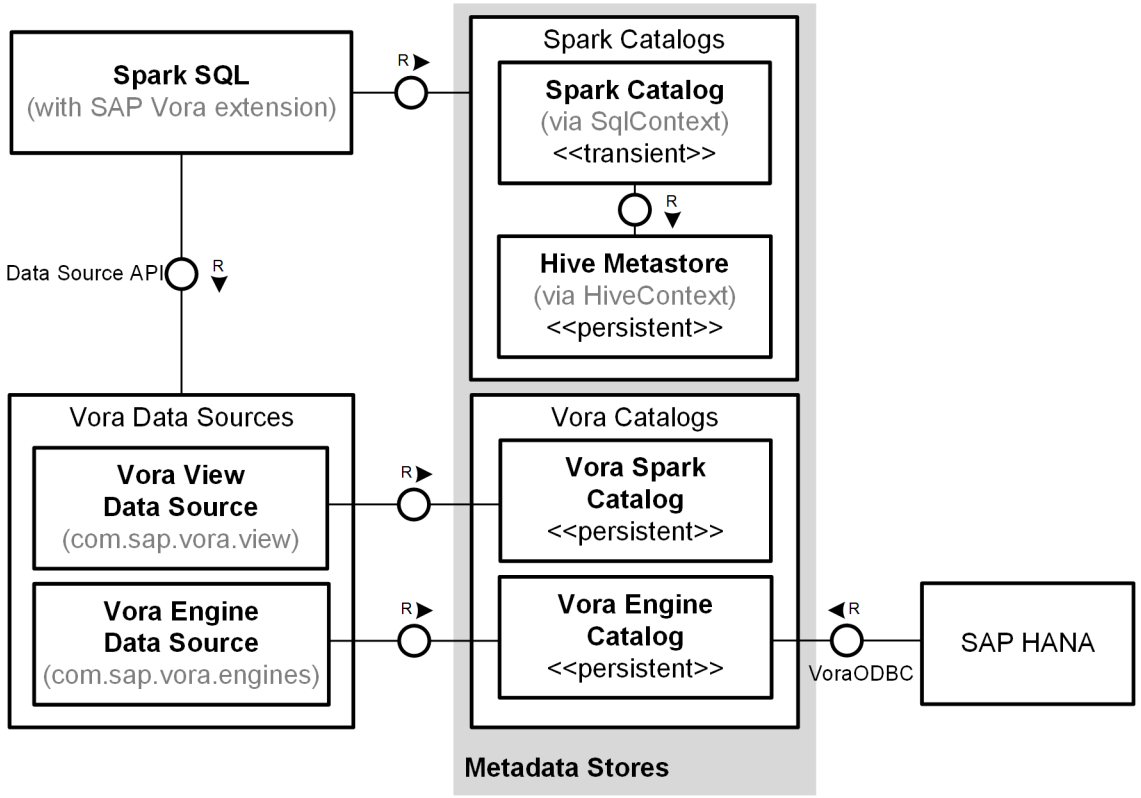
Working with Spark SQL requires an instance of an SQL context. Spark provides two SQL contexts: the standard `SQLContext` and the `HiveContext`. See *Working with Hive Tables and the SapHiveContext*. Both contexts differ in functionality. For example, the `HiveContext` supports the HiveQL language instead of the standard Spark SQL dialect. Please refer to the Spark documentation for more information.

One major difference between the two contexts is that the `SQLContext` makes use of a transient catalog to store metadata, while the `HiveContext` supports persistent metadata storage in the Hive Metastore (see the figure below). If you therefore create a table using the `SQLContext`, the table metadata goes out of scope at the end of the Spark session, and if a new session is started the table needs to be created again. In contrast, the `HiveContext` persists table metadata in the Hive metastore. Therefore, tables that have been created with the HiveContext "survive" session restarts.

The SAP Vora Spark extension provides two specializations for the Spark contexts: the `SapSQLContext` and the `SapHiveContext`. Both contexts support the respective feature set. They also follow Spark's behavior with respect to metadata storage: the `SapSQLContext` provides session storage and `SapHiveContext` provides persistent storage in the Hive Metastore.

In addition, when working with the SAP Vora data sources (`com.sap.spark.view` and `com.sap.spark.engines(.*)`), data is stored in SAP Vora's own persistent catalog.

An overview of the catalogs is shown below:



For more information about using the Spark SQL context with the SAP Vora data sources, see *Spark SQL Context and Data Source Combinations*.

Note that when consuming SAP Vora artifacts from SAP HANA using the methods described in *Accessing SAP Vora from SAP HANA* (with the VoraODBC implementation), this only applies to artifacts stored in the Vora engine catalog. This is because SAP HANA communicates directly with the SAP Vora Transaction Coordinator and not through the SAP Vora Spark extension (see *System Architecture*).

Related Information

Spark SQL Context and Data Source Combinations [page 20]
Catalog Example [page 22]
Working with Hive Tables and the SapHiveContext [page 31]
Accessing SAP Vora from SAP HANA [page 155]
System Architecture [page 10]

3.1 Spark SQL Context and Data Source Combinations

Metadata is stored differently depending on the Spark SQL context chosen and the data source used. Your choice therefore has implications that potentially affect how the created artifacts are consumed.

The options available are shown in the following table. Each option is described in more detail further below:

	SQL Context	Data Source	Metadata Storage
1	`SapSQLContext` `SapHiveContext`	`com.sap.spark.view` (Vora views)	Spark catalog (transient) and Vora Spark catalog (persistent) Note: View metadata is not persisted in the Hive metastore.
2	`SapSQLContext`	`com.sap.spark.engines.relational` `com.sap.spark.engines.disk`	Spark catalog (transient) and Vora engine catalog (persistent)
3	`SapSQLContext`	`com.sap.spark.engines` via raw SQL interface (time series, graph, document data)	Vora engine catalog (persistent)
4	`SapHiveContext`	`com.sap.spark.engines.relational` `com.sap.spark.engines.disk`	Hive Metastore and Vora engine catalog
5	`SapHiveContext`	`com.sap.spark.engines` via raw SQL interface (time series, graph, document data)	Vora engine catalog

1. `SapSQLContext` or `SapHiveContext` with `com.sap.spark.view` (Vora views)
   When working with the SAP Vora view extension, the view SQL is stored in the Spark catalog (transient) and the Vora Spark catalog during the view creation, for example:

   ```
   // Metadata is written to the Spark Catalog and the Vora Spark Catalog
   CREATE VIEW myView AS SELECT * from table1 using com.sap.spark.view
   Because the view metadata is stored in the Spark Catalog, you can use plain
   Spark SQL queries to access the view, for example:
   // Spark fetches view metadata from the Spark Catalog
   SELECT * from myView
   When the session goes out of scope, the Spark Catalog is deleted. In a new
   session, the view can be retrieved using a REGISTER TABLE call, for example:
   // Vora fetches the View metadata from the Vora Spark catalog
   REGISTER TABLE myView using com.sap.spark.view
   ```

   For more information, see *Creating Views* and *Registering Tables in Spark*.
   Since Vora views are not persisted in the Vora engine catalog, they cannot be queried from SAP HANA using the VoraODBC remote source adapter.
2. `SapSQLContext` with `com.sap.spark.engines.relational` or `com.sap.spark.engines.disk`
   The metadata for relational tables (either stored on disk or in the relational in-memory engine) is stored in the Spark catalog (transient) and the Vora engine catalog (persistent).
   Thus, these tables can be used in standard Spark SQL (through the Spark catalog), for example:

   ```
   SELECT * from myTable; // Spark resolves metadata using the Spark Catalog
   ```

When the session goes out of scope, the Spark catalog is deleted. In a new session, the table can be retrieved from the Vora engine catalog using a REGISTER TABLE call.
Since Vora relational tables are persisted in the Vora engine catalog, they can be queried from SAP HANA using the VoraODBC remote source adapter.

3. `SapSQLContext` with `com.sap.spark.engines` via raw SQL interface (time series, graph, document data)
   Document collections, time series, and graphs are accessible in SAP Vora through the raw SQL syntax and the parsed DDL syntax (see *Raw SQL Syntax and Parsed DDL Syntax*). Metadata for these database artifacts is kept solely in the Vora engine catalog.
   For example:

   ```
   // Both statements store metadata in the Spark Engine Catalog only
   // Parsed DDL Syntax to create a collection
   CREATE COLLECTION C1 using com.sap.spark.engines
   // Raw SQL Syntax to create a collection
   ``CREATE COLLECTION C2`` using com.sap.spark.engines
   ```

   Spark does not have a notion of these artifacts. Therefore, it is not possible to use the respective names in Spark queries directly:

   ```
   // Results in an error, because "C1" is not known to the Spark Catalog
   SELECT * FROM C1
   ```

   Rather, the raw SQL syntax must be used to query the artifacts:

   ```
   // Vora retrieves metadata for C1 from the Vora Engine Catalog
   ``SELECT * FROM C1`` using com.sap.spark.engines
   ```

   Although the metadata for time series, graphs and document collections is stored in the Vora engine catalog, these objects are not accessible via the VoraODBC remote source adapter.
4. `SapHiveContext` with `com.sap.spark.engines.relational` or `com.sap.spark.engines.disk`
   When using the `SapHiveContext` for relational data (relational or disk), the metadata is stored in the Hive metastore and the Vora engine catalog.
   These tables do not require an explicit REGISTER TABLE call when a new session is started because Spark can read the metadata from the Hive metastore.
   Note that like the plain `HiveContext`, the `SapHiveContext` requires a proper setup of the Hive Metastore to work correctly. Please consult your Spark documentation for further details.
5. `SapHiveContext` with `com.sap.spark.engines` via raw SQL interface (time series, graph, document data)
   When using the raw SQL syntax or the parsed DDL syntax with the `SapHiveContext`, the same applies as described in option 3. No data is kept in the Hive Metastore.

Related Information

Creating Views [page 37]
Registering Tables in Spark [page 35]
Raw SQL Syntax and Parsed DDL Syntax [page 58]

3.2 Catalog Example

The following example illustrates how metadata is handled by SAP Vora.

Sample Code

```
import org.apache.spark.sql._
// Create a Spark SQLContext with SAP Vora extensions
val sqlc = new SapSQLContext(sc)
// Use the SAP Vora data source extension 'com.sap.spark.engines.relational'
to create a table.
// The table metadata will be stored in the Vora Engine Catalog and
// in the Spark Catalog (session-only lifetime).
// The resulting table is available to Spark as a DataFrame, which can be
queried.
sqlc.sql(s"""CREATE TABLE testTableName (column1 string, column2 integer)
USING com.sap.spark.engines.relational
OPTIONS ( files "/path/to/file.csv")""".stripMargin)
// Execute a standard Spark SQL query on the table stored in SAP Vora.
sqlc.sql(s"""SELECT count(*) FROM testTableName""").show
// Close and re-start the session with a new SapSQLContext
[new example code area for new session]
import org.apache.spark.sql._
// Create a Spark SQLContext with SAP Vora extensions
val sqlc = new SapSQLContext(sc)
// The following query shows the tables known to the Spark Catalog.
// It should return an empty result because the session was closed
// and no tables have been created in the Spark catalog.
sqlc.sql(s"""show tables""").show
// The following query shows the tables persisted in the SAP Vora Engine
catalog.
// It should return the table created in the previous session.
sqlc.sql(s"""SHOW TABLES USING com.sap.spark.engines.relational""" ).show
// Register a table from the SAP Vora catalog.
// This command fetches metadata from the SAP Vora catalog into the Spark
catalog.
sqlc.sql(s"""REGISTER TABLE testTableName using
com.sap.spark.engines.relational""")
// The following query should return data about 'testTableName'.
// The table can now be used as an ordinary Spark DataFrame again.
sqlc.sql(s"""show tables""").show
```

4 Working with Relational Tables and Views

The SAP Vora Spark extension allows you to improve Spark performance by using SAP Vora as an in-memory database or disk index.

> **Note**
>
> Spark 1.6.x is deprecated and will be removed in a future version.

The SAP Vora Spark extension defines certain Spark SQL language extensions that allow you to:

- Create tables in SAP Vora
- Load data from distributed files systems such as HDFS or S3 into tables
- Partition data
- Query the data loaded to SAP Vora

To support these operations, SAP Vora extends the Spark Datasource API and the Spark SQL dialect.

Internally, the tables stored in SAP Vora are exposed to Spark as standard Spark DataFrames. Thus, once created, you can query tables stored in SAP Vora using standard Spark SQL. In addition, you can use the Spark SQL extensions (for hierarchies, annotations, currency conversion) provided by SAP Vora. The SAP Vora Spark extension analyzes Spark SQL queries for opportunities to push the execution of parts of the query, or the complete query if possible, to the SAP Vora engines.

A Spark SQL query can be executed on multiple SAP Vora engines concurrently and the execution result returned back to the Spark runtime as a data frame. The returned data frame can be further processed using the standard Spark functionality.

SAP Vora provides the following data sources to manage relational data: `com.sap.spark.engines.relational` (the relational in-memory engine) and `com.sap.spark.engines.disk` (the disk engine). These data sources support the following:

- Appending data to existing tables using files. Note that in SAP Vora you use APPEND instead of INSERT.
- Listing all tables persisted in the relational in-memory database and disk engine
- Registering persisted tables in the Spark SQLContext for data processing
- Dropping tables from the Spark SQLContext as well as from SAP Vora. Note that SAP Vora does not support DELETE or TRUNCATE.
- Partitioning tables. When using `com.sap.spark.engines.relational` or `com.sap.spark.engines.disk`, the data is always assigned to a single host in the cluster unless partitioning is applied.

Related Information

Creating Tables in the Relational Engines [page 24]
Creating Partitioned Tables in the Relational Engines [page 26]
Code Examples [page 27]

Disk Engine and Relational Engine Data Source API [page 53]

4.1 Creating Tables in the Relational Engines

A CREATE TABLE statement registers the table in the Spark SQLContext and creates a table in the SAP Vora relational in-memory or disk engine.

You need to provide a table name, the fully qualified name of the SAP Vora data source package (for example, `com.sap.spark.engines.relational`), and a set of options required by the data source. For example, using SQL:

Sample Code

```
CREATE TABLE testTableName (column1 string, column2 integer)
USING com.sap.spark.engines.relational
OPTIONS (
 files "/path/to/file1.csv,/path/to/file2.csv"
)
```

Note that you can also create the table programmatically with Scala. For more information, see the code examples.

Note

The code examples work on the relational in-memory engine and the disk engine. Adapt the code examples by replacing the provider accordingly: `com.sap.spark.engines.relational` or `com.sap.spark.engines.disk`.

Remember

Tables created with `com.sap.spark.engines.relational` or `com.sap.spark.engines.disk` are not automatically partitioned.

Semantics

To allow a table that has already been created to be registered with the Spark SQLContext, the following commands are available:

- CREATE TABLE IF NOT EXISTS statement with the same table name as an already existing table.
  In this case, the provided schema and metadata are not considered and will be the same as those of the already existing table. If the table with the given name does not yet exist, the statement will create a new table.
- CREATE TABLE statement, where the table name and schema are the same as those of an already existing table.
  If a table with the same name but different schema already exists, an exception is thrown. If the table with the given name does not yet exist, the statement will create a new table.

- CREATE TABLE statement with the same table name as an already existing table, but without a schema. In this case, the schema and metadata are the same as those of the already existing table. If the table with the given name does not yet exist, the statement will fail and the table will not be registered.
- REGISTER TABLE statement with the same table name as an already existing table.
  For more information, see *Registering Tables in Spark*.

Example

The sample code below shows how a table can be created from the Spark Scala shell using the SAP Vora relational in-memory data source:

Sample Code

```
import org.apache.spark.sql._
val sqlc = new SapSQLContext(sc)
sqlc.sql(
s"""CREATE TABLE testTableName (column1 string, column2 integer)
USING com.sap.spark.engines.relational
OPTIONS (
files "/path/to/file.csv",
hdfsnamenode "namenode.example.com:8020")""".stripMargin)
```

Note the following:

- An SapSQLContext has been instantiated based on a SparkContext object.
- Data is loaded into the table from a file in HDFS. The location of the file is specified using the `files` option. The `hdfsnamenode` parameter is required when reading data from HDFS and the parameter has not been set in the Spark default configuration.
- All options can be set in the SparkConf or in the SQLContext configuration. For more information about the supported options, see *Disk Engine and Relational Engine Data Source API*.

Related Information

Creating Partitioned Tables in the Relational Engines [page 26]
Appending Data to Existing Tables [page 32]
Registering Tables in Spark [page 35]
Disk Engine and Relational Engine Data Source API [page 53]

4.2 Creating Partitioned Tables in the Relational Engines

When you create tables using `com.sap.spark.engines.relational` or `com.sap.spark.engines.disk`, the data is always assigned to a single node in the cluster unless partitioning is applied.

You can create tables that support partition functions and partition schemes as shown in the example below. First use the engine data source to create the partition function and partition scheme:

Sample Code

```
CREATE PARTITION FUNCTION PF1 ( c INTEGER )
AS RANGE BOUNDARIES( 5 ) MIN PARTITIONS 5
USING com.sap.spark.engines;
CREATE PARTITION SCHEME PS1 USING PF1
USING com.sap.spark.engines;
```

Then create a table on the disk engine or relational in-memory engine with partition scheme `PS1(a1)` as follows:

Sample Code

```
CREATE TABLE TAB001(a1 int, a2 double, a3 date, a4 string)
USING com.sap.spark.engines.disk
OPTIONS(ps "PS1(a1)",
        files "f1.csv",
        csvdelimiter "|",
        dateformat "a3: 'YYYY-MM-DD'");
```

For the relational in-memory engine, replace `USING com.sap.spark.engines.disk` with `USING com.sap.spark.engines.relational`.

Related Information

Partitioning Tables [page 140]

4.3 Code Examples

The following code examples show how a table can be created and queried in Spark using the SAP Vora relational in-memory data source.

> **Note**
>
> The code examples work on the relational in-memory engine and the disk engine. Adapt the code examples by replacing the provider accordingly: `com.sap.spark.engines.relational` or `com.sap.spark.engines.disk`.

Code Example: Create and Query a Table with SQL

Sample Code

```
We assume a CSV file on HDFS that contains some data:
John,10
Jane,20
John,20
Jane,40
*/
import org.apache.spark.sql._
val sqlc = new SapSQLContext(sc)
sqlc.sql(
s"""
               CREATE TABLE testTable (name string, score integer)
               USING com.sap.spark.engines.relational
               OPTIONS (
                  files "/path/to/file.csv",
                  hdfsnamenode "<host>:<port>"
    )""".stripMargin)
/* query for entries with score > 10 */
sqlc.sql("SELECT name, age from testTable where score > 10").show
/* query for average score */
sqlc.sql("SELECT name, AVG(score) AS avgScore from testTable GROUP BY
name").show
```

Code Example: Create and Query a Table Programmatically

This example uses Spark DataFrames. For information about programming with DataFrames, see the *Spark SQL and DataFrame Guide*.

Sample Code

```
import org.apache.spark.sql._
import org.apache.spark.sql.types._
/*
Csv file that just contains:
```

```
John,10
Jane,20
John,20
Jane,40
*/
val stds1 = "..." // Some path to a csv file
val sqlc = new SapSQLContext(sc)
/* Source package needed to use the Vora source */
val source = "com.sap.spark.engines.relational"
/* Table schema */
val schema = StructType(
 StructField("name", StringType, nullable = true) ::
 StructField("age", IntegerType, nullable = true) :: Nil
)
val options = Map(
 /* Comma-separated CSV file paths in HDFS*/
 "files" -> stds1,
 /* Table name */
 "tablename" -> "voraTable"
)
/* Creating the new table */
val voraTable = sqlc.read.format(source).schema(schema).options(options).load()
/*
Jane,20
John,20
Jane,40
*/
val queryResult = voraTable.select("name", "age").where(voraTable("age") > 10)
queryResult.collect().foreach(println)
/* We need to import this to use the different sql functions like MAX, MIN or
AVG */
import org.apache.spark.sql.functions._
/*
John,15.0
Jane,30.0
*/
val aggregationResult = voraTable.select("name",
"age").groupBy("name").agg(avg("age").as("avgAge"))
aggregationResult.collect().foreach(println)
```

Related Information

Spark SQL and DataFrame Guide

4.4 Loading Data into Tables

You can load data into tables from files stored in HDFS or Amazon S3. You can load the files at the time of table creation using the CREATE TABLE statement, or at a later stage using the APPEND TABLE statement.

> **Note**
>
> The primary persistence of the data is the original files in HDFS, Amazon S3, and so on. The original files must therefore not be deleted. SAP Vora does not currently provide a primary persistence.

Prerequisites

If your cluster runs behind a proxy, your proxy settings need to be set up correctly. Otherwise the SAP Vora engine or Spark might not be able to read files from Amazon S3 due to missing proxy information.

Files

You use the `files` option to specify the fully qualified names of the files to be uploaded, separated by commas. The example below shows how to load data into a table from two files stored in HDFS (Hadoop Distributed File System). The storage backend is, by default, HDFS and the file format CSV:

Sample Code

```
CREATE TABLE testTableName (column1 string, column2 integer)
USING com.sap.spark.engines.relational
OPTIONS (
 files "/path/to/file1.csv,/path/to/file2.csv",
 hdfsnamenode "namenode.example.com:8020"
)
```

The `hdfsnamenode` parameter is required when reading data from HDFS. You always need to set it when creating tables with the SQL editor in the SAP Vora tools, even if it is set in the Spark default configuration.

If an option is specific to a single file (such as `storagebackend`, `format`, `csvdelimiter`, and so on), it can be specified per file using the following notation:

```
<FILE>[|<OPTKEY>=<OPTVAL>][,<FILE>[|<OPTKEY>=<OPTVAL>]]
```

where:

- <FILE> is the fully qualified file name
- <OPTKEY> is the option name (such as `storagebackend`)
- <OPTVAL> is the option value (such as `hdfs`)

For example, the following `files` option specifies two files with different storage backend and format options:

Sample Code

```
"/path/to/file1.orc|format=orc|storagebackend=hdfs,/path/to/file2.csv|
storagebackend=s3"
```

If separation characters occur in either the file name, option key, or option value, they need to be quoted with single quotes.

If you need to specify a single quote (for example, as the csv quote character), it must be doubled. For example, the following `files` option specifies a csv file separated by '|' and the single quote as the csv quote character:

Sample Code

```
"/path/to/file1.csv|storagebackend=s3|csvdelimiter='|'|csvquote=''''
```

For the disk engine, replace `USING com.sap.spark.engines.relational` with `USING com.sap.spark.engines.disk`.

For more information about the file options, including default values and examples, see *Disk Engine and Relational Engine Data Source API*.

Amazon S3 Data

To load a data file from Amazon S3 (Simple Storage Service) into a table in SAP Vora, you need to specify the key ID and secret key ID of your Amazon S3 account, as well as the Amazon S3 endpoint to be contacted:

Sample Code

```
CREATE TABLE testTableName (column1 string, column2 integer)
USING com.sap.spark.engines.relational
OPTIONS (
 files "/S3_BUCKET/data.csv",
 csvdelimiter "|",
 storagebackend "s3",
 s3accesskeyid "S3_KEY_ID",
 s3secretaccesskey "S3_KEY_SECRET",
 s3endpoint "S3_ENDPOINT",
 )
```

Parameter	Description	More Information
files	Fully qualified names of the files to be uploaded to SAP Vora	SAP Vora accepts Amazon S3 file names in the following format: `/<bucket_name>/<file_path>` For example: `/examples/data.csv`
storagebackend	Storage backend ("s3" or "hdfs")	Set to "s3" to load files from Amazon S3
s3accesskeyid	Amazon S3 access key	You can get the key ID from the Amazon console
s3secretaccesskey	Amazon S3 secret access key	You can get the secret access key from the Amazon console
s3endpoint	Amazon S3 endpoint	You can find information about your endpoint at: http://docs.aws.amazon.com/general/latest/gr/rande.html#s3_region

Note that as with other parameters, you can also configure the Amazon S3 parameters in the `spark-defaults.conf` file. For security reasons, we recommend that you configure the Amazon S3 secret key in the `spark-defaults.conf` file to avoid having to enter it in the Spark shell. The `spark-defaults.conf` file is located at `<spark-installation>/conf/` (for example: `/opt/spark/conf`).

Note

Data files can currently be loaded in one direction only, from Amazon S3 into SAP Vora.

Related Information

Disk Engine and Relational Engine Data Source API [page 53]

4.5 Working with Hive Tables and the SapHiveContext

The SAP Vora Spark extensions also come with proper Hive integration. The Hive context has all the features provided by the `SapSQLContext` as well as some additional Hive-dependent ones.

Initialization

You can instantiate an `SapHiveContext` as follows:

```
import org.apache.spark.sql.hive.SapHiveContext
val hiveContext = new SapHiveContext(sparkContext)
```

If you open a Spark shell and a `HiveContext` already exists, this will lead to an exception because there can only be one metastore session.

In Spark you usually use `HiveContext.newSession` to create a second `HiveContext` from the existing one. The problem however is that `newSession` only returns a plain `HiveContext`. So for this use case there is the `SapHiveContext.newSessionFrom` method, which is implemented on the companion object of `SapHiveContext`:

```
val newSessionSapHiveContext = SapHiveContext.newSessionFrom(hiveContext)
```

Dialects

By default, the `SapHiveContext` runs with the `sapsql` dialect. This dialect is the one that includes features such as view creation, hierarchies, and system tables.

To switch to the standard Hive SQL dialect you have to set `spark.sql.dialect` to `hiveql`. This can be done either programmatically using the following:

```
hiveContext.setConf("spark.sql.dialect", "hiveql")
```

Or you can use SQL as follows:

```
SET spark.sql.dialect=hiveql;
```

Hive Import

The Hive import feature allows you to import a table from Hive into `com.sap.spark.engines.relational` or `com.sap.spark.engines.disk`. During a CREATE TABLE, you can specify that a Hive table is to be used as the basis for creating the table. To do so, you need to specify at least the `hivetable` option together with the table to be imported. If you also want to import files from a Hive table, you have to specify the files relative to the directory that Hive stores them in using the `hivefiles` parameter.

A complete statement could therefore look something like this:

```
CREATE TABLE VORA_IMPORTED
USING com.sap.spark.engines.relational
OPTIONS (
  hivetable "myhivetable",
  hivefiles "part.csv"
)
```

You can also adjust the database where the table to be imported is located using the `hivedb` parameter.

4.6 Appending Data to Existing Tables

You can add more data to tables using the APPEND TABLE command.

Sample Code

```
APPEND TABLE testTableName OPTIONS (files "path1/to/file/file1.csv,path2/to/
file/file2.csv")
```

APPEND TABLE supports the same options as CREATE TABLE, as well as the following option:

Option	Description	Default Value	Example
override	Overrides the content of the table with the new data. To do this, the former table is truncated before the new files are added. Note: This is not an atomic operation (like all save operations in Spark SQL). If append fails, the truncated table will not be recovered.	false	true

Note

If you append files with the same path (with identical or different content), their content will be added to the table each time, even if the content of the file has not changed.

Related Information

Creating Tables in the Relational Engines [page 24]

4.7 Listing Tables and Views

You can list all relations persisted in the SAP Vora in-memory database and disk engine using the `SHOW TABLES` statement.

The syntax of the SHOW TABLES statement is shown in the examples below.

List tables in the relational in-memory engine or disk engine:

Sample Code

```
SHOW TABLES
USING com.sap.spark.engines.relational
OPTIONS (
<some options>
)
+----------+------------+----------+
|TABLE_NAME|IS_TEMPORARY|KIND      |
+----------+------------+----------+
|    TAB002|       FALSE|RELATIONAL|
|    TAB001|       FALSE|RELATIONAL|
+----------+------------+----------+
SHOW TABLES
USING com.sap.spark.engines.disk
OPTIONS (
<some options>
)
+----------+------------+----+
|TABLE_NAME|IS_TEMPORARY|KIND|
+----------+------------+----+
|    TAB002|       FALSE|DISK|
|    TAB001|       FALSE|DISK|
+----------+------------+----+
```

Note that the following query on the system table SYS.TABLES provides the same information as above.

Sample Code

```
SELECT * FROM SYS_TABLES USING com.sap.spark.engines.disk;
```

List views in the relational in-memory engine and disk engine:

Sample Code

```
SHOW TABLES
USING com.sap.spark.view
OPTIONS (
<some options>
```

The returned result contains the name of the relation as well as the following information, as shown in the example below:

TABLE_NAME	IS_TEMPORARY	KIND
Table1	FALSE	TABLE
MyView	FALSE	VIEW
MyDimensionView	FALSE	DIMENSION
MyCubeView	FALSE	CUBE

Note that if you use the SHOW TABLES command without specifying the SAP Vora datasource, only the tables available in the Spark catalog are shown:

```
// show only tables registered in the Spark catalog
SHOW TABLES
+---------+-----------+
|   TAB002|      false|
|   TAB001|      false|
+---------+-----------+
```

You can also display the original SQL statement used to create a view by executing the DESCRIBE TABLE command, for example:

Sample Code

```
DESCRIBE TABLE MyView USING com.sap.spark.engines.relational
DESCRIBE TABLE MyView USING com.sap.spark.engines.disk
```

Result:

TABLE_NAME	DDL_STMT
MyView	CREATE VIEW MyView AS SELECT * FROM Table1 USING com.sap.spark.engines.relational OPTIONS ()

4.8 Registering Tables in Spark

Since tables created in Spark exist only for the lifetime of a particular Spark SQLContext, you can use the REGISTER TABLE or REGISTER ALL TABLES statement to import tables from the SAP Vora data sources into a new Spark context.

REGISTER TABLE

You can use the REGISTER TABLE statement to register a specific table already created in the SAP Vora data source into the Spark catalog. No additional metadata or schema information is needed to perform the registration:

```
// table in the new relational in-memory engine
REGISTER TABLE <table name>
USING com.sap.spark.engines.relational
[OPTIONS <options>] [IGNORING CONFLICTS]
// table in the disk engine
REGISTER TABLE <table name>
USING com.sap.spark.engines.disk
[OPTIONS <options>] [IGNORING CONFLICTS]
// view in the new in-memory engine or disk engine
REGISTER TABLE <table name>
USING com.sap.spark.view
[OPTIONS <options>] [IGNORING CONFLICTS]
```

An error is thrown if the table already exists in the Spark catalog, but you can force the registration by using the `IGNORING CONFLICTS` clause, which then overwrites that table in the Spark catalog.

Alternatively you can use the `IF NOT EXISTS` clause to skip the registration if the table is already registered.

REGISTER ALL TABLES

You can use the REGISTER ALL TABLES statement to register all tables already created in the SAP Vora data source into the Spark catalog. No additional metadata or schema information is needed to perform the registration:

Sample Code

```
// tables in the new relational in-memory engine
REGISTER ALL TABLES
USING com.sap.spark.engines.relational
[OPTIONS <options>] [IGNORING CONFLICTS]
// tables in the disk engine
REGISTER ALL TABLES
USING com.sap.spark.engines.disk
[OPTIONS <options>] [IGNORING CONFLICTS]
// views in the new in-memory engine and disk engine
REGISTER ALL TABLES
USING com.sap.spark.view
[OPTIONS <options>] [IGNORING CONFLICTS]
```

An error is thrown if any of the tables already exist in the Spark catalog, but you can force the registration by using the `IGNORING CONFLICTS` clause, which then overwrites those tables in the Spark catalog.

Alternatively you can use the `IF NOT EXISTS` clause to skip the registration of any tables that are already registered.

4.9 Dropping Tables

Use the DROP TABLE command to drop a table from both Spark and SAP Vora.

Sample Code

```
DROP TABLE testTableName
```

If you are not sure whether the table exists, you can add `IF EXISTS` to the clause as follows:

Sample Code

```
DROP TABLE IF EXISTS tableName
```

An exception will not be thrown if the table does not exist.

If the specified table is referenced more than once in the SAP Vora catalog, the drop table action will fail. This could happen if, for example, the table is used in a number of views.

To drop both the table and every entry in the catalog that references it, add the `CASCADE` suffix to the `DROP TABLE` statement as follows:

Sample Code

```
DROP TABLE tableName CASCADE
```

Related Information

Dropping Views [page 38]

4.10 Creating Views

Views are virtual tables that contain the result of an SQL query execution. You can create views and use them later to construct complex queries.

When you create a view in Spark, it is deleted when the corresponding SQLContext is closed. However, views can also be persisted if the data source supports it.

Note that the data source itself has no notion of a view's behavior. When a view is stored in a data source, it is only its SQL string that is stored.

Plain Spark Views

You can create a view that resides only in the SQLContext catalog by issuing a CREATE VIEW statement. The basic syntax of a CREATE VIEW statements is as follows:

```
CREATE (DIMENSION | CUBE)? VIEW <view name>
AS <view query>
```

Sample Code

```
CREATE VIEW MyView
AS SELECT * FROM Table1
```

Note that in the current release these views act exactly like normal views.

Persisted Views

To create a view in the current catalog and store it additionally in a given datasource, you can issue a CREATE VIEW ... USING statement. Use `com.sap.spark.view` for the relational in-memory engine and disk engine. The syntax of this statement is as follows:

```
CREATE (DIMENSION | CUBE)? VIEW <view name>
AS <view query>
USING <datasource name>
[OPTIONS <options>]
```

Sample Code

```
CREATE VIEW MyView
AS SELECT * FROM Table1
USING com.sap.spark.view
```

When the `com.sap.spark.view` data source is used to store a view, the view's SQL statement is stored inside the SAP Vora catalog.

To retrieve a view that was once stored using a data source, you can use the REGISTER TABLE <view name> statement.

Related Information

Dropping Views [page 38]
Listing Tables and Views [page 33]

4.11 Dropping Views

To drop a view use the DROP VIEW statement.

For plain Spark views, the DROP VIEW statement will only remove the view from the Spark catalog. For plain engine views, the DROP VIEW statement will only remove the view from the engine catalog. If views are persisted in a data source, they will also be removed from that data source.

The syntax of the DROP VIEW statement is as follows:

```
DROP VIEW <view name>
```

Sample Code

```
DROP VIEW MyPersistedView
```

Before dropping a view, make sure that it exists in the current catalog. Alternatively, you can add `IF EXISTS` to the clause as follows:

Sample Code

```
DROP VIEW IF EXISTS MyView
```

An exception will not be thrown if the view does not exist.

If the view has dependencies, you have to drop the referencing relations as well. To do so, add the `CASCADE` suffix to the `DROP VIEW` statement as follows:

Sample Code

```
DROP VIEW MyView CASCADE
```

Related Information

Creating Views [page 37]
Listing Tables and Views [page 33]
Dropping Tables [page 36]

4.12 Adding Annotations

You can define annotations on columns of views. An annotation is a key/value pair that allows external tools, such as modelers and visual SQL editors, to store additional information about the view's columns, such as default aggregations or UI formatting tips. This makes integration with SAP Vora and Spark easier.

Adding Annotations to Views

You can create a view with annotations as follows:

Sample Code

```
CREATE VIEW view1 AS SELECT col1 @ (foo = 'bar'), col2 @ (baz = 'buzz') FROM
table1 USING com.sap.spark.view
```

Note

This feature is only supported by the `com.sap.spark.view` data source.

In this example, the view `view1` has been defined with two columns, `col1` and `col2`, with the following annotations:

- `col1`: annotation `foo`, value 'bar'
- `col2`: annotation `baz`, value 'buzz'

You can define any number of annotations, but note the following:

- Annotations keys must be unique for a given column.
- Value types can be either string, long, double, or string array.

View `view1` can be queried just like any other relation in Spark.

Viewing Annotations

To view the annotations defined on a table, you can use the table-valued function DESCRIBE_TABLE as follows:

Sample Code

```
SELECT * FROM DESCRIBE_TABLE(SELECT * FROM view1)
```

The result contains all columns and annotations defined on the columns:

TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME	ORDINAL_POSITION	IS_NULLABLE	DATA_TYPE	NUMERIC_PRECISION	NUMERIC_SCALE	ANNOTATION_KEY	ANNOTATION_VALUE
t1	table1	col1	0	true	INTEGER	null	null	null	null
t1	table1	col2	1	true	INTEGER	null	null	null	null
t1	view1	col1	0	true	INTEGER	null	null	foo	bar
t1	view1	col2	1	true	INTEGER	null	null	baz	buzz

Note that you can also retrieve this information from the SYS.SCHEMAS system table. For more information, see *System Tables*.

Annotation Propagation

You can also define annotations on nested views. In this case, they are propagated from the nested view to the outer one.

For example:

Sample Code

```
CREATE VIEW annotatedView AS SELECT col1 AS col3 @(extraKey = 'extraValue'),
col2 AS col4 @(baz = 'overridenValue') FROM view1
```

The resulting view `annotatedView` contains two columns, `col1` and `col2`:

- `col1`: `col3` is an alias of `col1`. This column has two annotations:
  - `extraKey` with the value 'extraValue'
  - The original annotation defined for `col1`: `foo` with the value 'bar'
- `col2`: `col4` is an alias of `col2`. This column has an annotation `baz` with the value 'overridenValue'. Note that the annotation originally defined for `col2` is overridden because `col4` defines a new annotation with the same key. This feature is referred to as annotation propagation.

Related Information

System Tables [page 43]

4.12.1 Supported CDS Annotation Types

SAP Vora supports the following CDS annotation types for views (all types), dimensions, and cubes.

Annotation	Scope	Example
Views		
EndUserText.heading	Entity, View, Element	@(EndUserText.heading = 'new heading')
EndUserText.label (not yet supported by the SAP Vora modeler)	Entity, View, Element	@(EndUserText.label = 'new UI label')
ObjectModel.text.element	Element	@(ObjectModel.text.element='COLUMN_NAME')
Semantics.type (amount; quantity; currencyCode; unitOfMeasure; date; url)	Element	@(Semantics.type = 'currencyCode')
Semantics.amount.currencyCode	Element	@(Semantics.amount.currencyCode = 'COLUMN_NAME')
Semantics.quantity.unitOfMeasure	Element	@(Semantics.quantity.unitOfMeasure = 'COLUMN_NAME')
Dimensions		
Analytics.dimensionType (STANDARD; TIME)	Entity, View	@(AnalyticsDetails.dimensionType='TIME')
Analytics.Details. drillDownEnablement (NONE; DRILL_DOWN)	Element	@(AnalyticsDetails.drillDownEnablement='DRILL_DOWN')
Cubes		
DefaultAggregation (NONE; SUM; MIN; MAX)	Element	@(DefaultAggregation='SUM')

The `EndUserText` and `Semantics` annotations are used in the following examples.

Sample Code

Dimension: MINI.PRODUCT_DIMENSION

```
create dimension view MINI.PRODUCT_DIMENSION as select
key PRODUCT as "ProductOrGroup" @(EndUserText.heading = 'Product / Product
Group') ,
    PRODUCT as "Product" @(EndUserText.heading = 'Product'),
    PRICE   as "Price" @(Semantics.amount.currencyCode = 'Currency',
AnalyticsDetails.drillDownEnablement = 'NONE', EndUserText.heading = 'Retail
Price'),
    PRODUCT_CURRENCY as "ProductCurrency" @(Semantics.currencyCode),
    null             as "ProductGroup" @(EndUserText.heading = 'Product
Group'),
    PRODUCT_GROUP    as "ParentProductGroup"
   from mini.product
union all select
    PRODUCT_GROUP as "ProductOrGroup",
```

```
    null as "Product",
    null   as "Price",
    null   as "ProductCurrency",
    PRODUCT_GROUP as "ProductGroup",
    PARENT_PRODUCT_GROUP as "ParentProductGroup"
   from "MINI"."PRODUCT_GROUP_HIERARCHY"
```

Sample Code

Cube: MINI.SALES_CUBE

```
create cube view MINI.SALES_CUBE as select
    F."Revenue" @(Semantics.amount.currencyCode = 'Currency',
DefaultAggregation = 'SUM', EndUserText.heading = 'Revenue'),
    F."Currency" @(Semantics.currencyCode),
    F."Revenue" / P."Price" as "AverageQuantity"@(DefaultAggregation =
'FORMULA', EndUserText.heading = 'Average Quantity')
    --columns from Customer dimension
    C."Customer",
    C."Country",
    C."Continent",
    --columns from employee dimension
    E."Employee" as "SalesRep" @(EndUserText.heading = 'Sales Representative'),
    E."Manager",
    E."FullName"  --is this really needed? (display attribute)
    --all columns from product dimension
    P.*,
from mini.V_SALES_FACT as F
inner join MINI.CUSTOMER_DIMENSION as C on F."Customer" = C."Customer"
outer join MINI.EMPLOYEE_DIMENSION as E on F."SalesRep" = E."Employee"
inner join MINI.PRODUCT_DIMENSION as P on F."Product" = P."Product"
```

Note

You can assign semantics and associate columns with label columns (`@ObjectModel.text.element`) in the SAP Vora modeler.

Related Information

Assign Semantics [page 242]
Associate Columns with Label Columns [page 244]

4.13 Table-Valued Functions

Table-valued functions are functions that can be used in SQL statements that take logical plans as their input and return a table as their output.

The `describe_table` function takes one logical plan and analyzes all used relations, together with their columns. For example:

Sample Code

```
SELECT * FROM describe_table(SELECT * FROM t)
```

This returns a table with the following fields:

field name	type	nullable	description
TABLE_SCHEMA	string	true	Schema the relation resides in
TABLE_NAME	string	false	Name of the table the column belongs to
COLUMN_NAME	string	false	Name of the column
ORDINAL_POSITION	int	false	Ordinal position in the relation
DATA_TYPE	string	false	Data type of the column
NUMERIC_PRECISION	int	true	Numeric precision for numeric data types
NUMERIC_PRECISION_RADIX	int	true	Numeric precision radix for numeric data types
NUMERIC_SCALE	int	true	Numeric scale for numeric data types
ANNOTATION_KEY	string	true	Key of an annotation
ANNOTATION_VALUE	string	true	Value of an annotation

An exception is thrown if the target relation does not exist. To return an empty result set if the target relation does not exist, use the `describe_table_if_exists` function instead, which has exactly the same output.

4.14 System Tables

System tables provide information about the objects in Spark and the data source system. They can be queried on standalone Spark, on a given data source, or both. Where this is supported is individually defined for each system table.

Note that you can use "SYS." or "SYS_" to access system tables.

SYS.TABLES

The SYS.TABLES system table provides information about the tables contained in either the Spark catalog or the specified data source. You can query SYS.TABLES on a given data source as follows:

```
SELECT * FROM SYS.TABLES USING com.sap.spark.engines.relational OPTIONS ()
SELECT * FROM SYS.TABLES USING com.sap.spark.engines.disk OPTIONS ()
```

The OPTIONS clause is optional.

You can query the same system table on Spark as follows:

```
SELECT * FROM SYS.TABLES
```

The structure of the output is as follows:

Field	Description
TABLE_NAME	Name of the relation
IS_TEMPORARY	Indicates whether the table ceases to exist when the current Spark context is closed
KIND	Either TABLE or VIEW
PROVIDER	Data source where the relation is stored (or null if it cannot be inferred or there is none)

A sample result looks like this:

TABLE_NAME	IS_TEMPORARY	KIND	PROVIDER
persons	FALSE	TABLE	com.sap.spark.engines.relational
pets	FALSE	TABLE	com.sap.spark.engines.relational
pet_owners	TRUE	VIEW	null

SYS.OBJECT_DEPENDENCIES

The object dependencies system table can currently only be queried on Spark. It shows a relation's directly dependent objects as well as the type of dependency. This system table has the following fields:

Field	Description
BASE_SCHEMA_NAME	Schema of the base object, usually null
BASE_OBJECT_NAME	Name of the base object, for example, the view or table name
BASE_OBJECT_TYPE	Type of base object. Currently this is either TABLE or VIEW.
DEPENDENT_SCHEMA_NAME	Schema of the dependent object, usually null
DEPENDENT_OBJECT_NAME	Name of the dependent object
DEPENDENT_OBJECT_TYPE	Type of dependent object. Currently this is either TABLE or VIEW.
DEPENDENCY_TYPE	ID of the dependency type. Currently only the referential dependency with ID 0 is supported.

You can query the SYS.OBJECT_DEPENDENCIES table as follows:

```
SELECT * FROM SYS.OBJECT_DEPENDENCIES
```

A sample result looks like this:

BASE_SCHEMA_NAME	BASE_OBJECT_NAME	BASE_OBJECT_TYPE	DEPENDENT_SCHEMA_NAME	DEPENDENT_OBJECT_NAME	DEPENDENT_OBJECT_TYPE	DEPENDENCY_TYPE
null	persons	table	null	pets	table	0
null	persons	table	null	english_speakers	view	0
null	english_speakers	view	null	presenters	view	0

SYS.TABLE_METADATA

The metadata system table is for providers only. It allows providers to list provider-specific metadata for a table. It can be queried as follows:

```
SELECT * FROM SYS.TABLE_METADATA USING com.sap.spark.engines
SELECT * FROM SYS.TABLE_METADATA USING com.sap.spark.engines.disk
```

This retrieves a table with the following fields:

Field	Description
TABLE_NAME	Name of the table the metadata is provided for
METADATA_KEY	Key of the metadata key-value pair
METADATA_VALUE	Value of the metadata key-value pair

Both METADATA_KEY and METADATA_VALUE are strings. The SAP Vora data source returns JSON strings for some metadata values.

A sample result is shown below:

TABLE_NAME	METADATA_KEY	METADATA_VALUE
employees	parent	null
employees	version	1.2.5

SYS.SCHEMAS

The SYS.SCHEMAS system table is available on both SAP Vora and providers that implement the DatasourceCatalog interface. It queries all table schemas of the target provider. A SQL statement looks like this:

```
SELECT * FROM SYS.SCHEMAS [ USING com.sap.spark.engines.relational ]
```

Note that "[...]" denotes optional.

This returns a table with the following fields:

Field	Description
TABLE_SCHEMA	Schema the table resides in
TABLE_NAME	Name of the table
COLUMN_NAME	Name of the column
ORDINAL_POSITION	Ordinal position of the column in the table schema
IS_NULLABLE	A boolean indicating whether the column value is nullable
DATA_TYPE	Data type of the given table. It may be a data source-dependent native type.
SPARK_TYPE	The corresponding Spark type for the data type (above). It might be null if there is none.
NUMERIC_PRECISION	Numeric precision of the Spark data type, if it is a numeric type
NUMERIC_PRECISION_RADIX	Numeric precision radix of the Spark data type, if it is a numeric type
NUMERIC_SCALE	Numeric scale of the Spark data type, if it is a numeric type
ANNOTATION_KEY	Key of a column annotation
ANNOTATION_VALUE	Value of a column annotation
Comment	A comment attached to the column

A sample result is shown below:

TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME	ORDINAL_POSITION	IS_NULLABLE	DATA_TYPE	SPARK_TYPE	NUMERIC_PRECISION	NUMERIC_PRECISION_RADIX	NUMERIC_SCALE	ANNOTATION_KEY	ANNOTATION_VALUE	Comment
null	persons	name	1	false	VARCHAR(20)	string	null	null	null	meta	data	comment1
null	animals	age	1	false	INTEGER	int	32	2	0	null	null	

INFER SCHEMA

The `INFER SCHEMA` command can be used to retrieve the schema of an ORC or Parquet file. The syntax of the command is as follows:

```
INFER SCHEMA OF "<path to file>" [ AS ( PARQUET | ORC ) ]
```

The path to the file is an HDFS path.

Sample Code

Executed on the cluster (the HDFS client needs to correctly configured):

```
INFER SCHEMA OF "/persons.orc"
```

Sample Code

Executed in SAP Vora Tools:

```
INFER SCHEMA OF "hdfs://<namenode_address>/pers.orc"
```

If you do not specify the `AS ...` clause, the file extension is used to infer the file type. If the file type cannot be inferred, an exception is thrown.

The command returns a table with the following fields:

field name	type	nullable	description
COLUMN_NAME	string	false	Name of the inferred column
ORDINAL_POSITION	int	false	Ordinal position of the column
DATA_TYPE	string	false	Data type of the column
NUMERIC_PRECISION	int	true	Numeric precision for numeric data types
NUMERIC_PRECISION_RADIX	int	true	Numeric precision radix for numeric data types
NUMERIC_SCALE	int	true	Numeric scale for numeric data types

4.15 Supported Data Types

The following table shows the supported data type mappings between Spark and the SAP Vora engine.

Category	Spark Type	Vora Type	Comments
String	CHAR(X)	VARCHAR(*)	
	VARCHAR(X)	VARCHAR(*)	
	STRING	VARCHAR(*)	
Numeric	TINYINT	TINYINT	Range: -127 to 126
	SMALLINT	SMALLINT	Range: -32767 to 32766
	INTEGER	INTEGER	Range: -2147483647 to 2147483646
	INT	INTEGER	Range: -2147483647 to 2147483646
	LONG	BIGINT	Range: -2^63 + 1 to 2^63 - 1

Category	Spark Type	Vora Type	Comments
	BIGINT	BIGINT	Range: -2^63 + 1 to 2^63 - 1
	FLOAT	REAL	32-bit floating point
	DOUBLE	DOUBLE	64-bit floating point
	DECIMAL	DECIMAL(10, 0)	Default decimal with precision 10 and scale 0
	DECIMAL(p, s)	DECIMAL(p, s)	Decimal with precision p and scale s. Precision and scale have to be 32-bit numbers.
Date and Time	DATE	DATE	
	Not supported	TIME	
	TIMESTAMP	TIMESTAMP	
Logical	BOOLEAN	BOOLEAN	
Other	BINARY	Not supported	
	ARRAY	Not supported	
	STRUCT	Not supported	

In some cases Spark does implicit data type conversions. For example, the VARCHAR(X) type is internally mapped to STRING without any length information. As a result, the VARCHAR(*) type (also without any length information) is used in SAP Vora as the Vora type.

If you want to control the data type used for SAP Vora, you can specify the type using the `tableschema` option for the disk engine (`com.sap.spark.engines.disk`) and relational in-memory engine (`com.sap.spark.engines.relational`), or the `schema` option for the SAP HANA data source (`com.sap.spark.hana`). For more information, see the respective API documentation.

For example:

Sample Code

```
CREATE TABLE tableWithSpecialSchema
USING com.sap.spark.engines.relational
OPTIONS (
  tableschema "name varchar(255), age smallint"
)
```

As a result, VARCHAR(255) and SMALLINT are used in SAP Vora.

Note that you do not need to specify the schema in the CREATE TABLE statement. It is sufficient to specify it in the OPTIONS clause.

Related Information

Disk Engine and Relational Engine Data Source API [page 53]
SAP HANA Data Source API [page 154]

Up Casting [page 49]

4.15.1 Up Casting

The behavior of SAP Vora in overflow situations is based on that of Apache Spark. That means, in particular, that the data contained in ORC, Parquet, or CSV files must not exceed the size allowed by the data types specified in the table schema.

The resulting data type of any binary operation is determined by the data type of the larger of the two input parameters (that is, the higher data type). For example, the multiplication of an INTEGER and a BIGINT results in the data type BIGINT.

If your expression might overflow, you can prevent errors by explicitly casting the data types to higher data types. You can do this by using the `cast` operator as follows: `cast(expression as type)`

Example

Assume `a` and `b` are two integer columns with numbers that might lead to an overflow during multiplication. To avoid an overflow, you apply the cast function to the select statement. The query should look something like this:

```
select cast(a as bigint) * cast(b as bigint) from table
```

4.16 Supported Expressions

The expressions that are explicitly supported in the SAP Vora relational engines (in-memory and disk) are listed below.

Context

Spark SQL describes its operations in the form of logical plans. Logical plans are tree structures and can contain nodes such as projections (SELECT a, b...), aggregations (GROUP BY together with SUM...), filters (WHERE a = 1...), and many other node types. Some are tied to actual SQL operations, while others are not.

When you issue a Spark SQL query where a supported data source table, such as an SAP Vora table, is involved, the pushdown process comes into play and the plan tree is scanned for subnodes that can be executed using the data source. In the SAP Vora-Spark integration, to see if a node is suitable for pushdown, the node is traversed and checked for support. If a node such as a projection contains expressions like, for example, `` `SELECT 1, a` ``, this means that the expressions `` `1` `` and `` `a` `` also have to be checked.

Note

Expressions that are not listed are not supported. Only the listed expressions are explicitly supported.

For expressions involving child nodes (such as `And` and `Not`) the support information only applies to the type of node. A node is only fully supported if all its sub-nodes are also supported.

Logical Expressions

- And: Supported
- Or: Supported
- Not: Supported

Aggregate Functions

- AggregateExpression: Supported when not distinct
- Average: Supported
- Count: Supported
- Max: Supported
- Min: Supported
- Sum: Supported
- VarianceSamp: Supported
- VariancePop: Supported
- StddevSam: Supported
- StddevPop: Supported
- First: Not Supported
- Last: Not Supported

Casts

Casts are supported for following data types:

- Long
- Date
- Double
- Integer
- Timestamp
- String
- Boolean

Null Handling

- IsNull: Supported
- IsNotNull: Supported
- Coalesce: Supported

String Operations

- Lower: Supported
- Upper: Supported
- Substring: Supported
- StringTrim: Supported
- StringTrimLeft: Supported
- StringTrimRight: Supported
- Replace: Supported
- Concat: Supported
- StringReverse: Supported
- Like: Supported
- Contains: Supported
- EndsWith: Supported
- StartsWith: Supported
- RLike: Not Supported
- Length: Supported
- StringLocate: Supported

Date Operations

- DateAdd: Supported
- DateSub: Supported
- AddMonths: Supported
- AddYears: Supported
- CurrentDate: Supported
- DateDiff: Supported
- DayOfMonth: Supported
- Month: Supported
- Year: Supported
- Hour: Supported
- Minute: Supported
- Second: Supported

Comparison Operators

- LessThan: Supported
- LessThanOrEqual: Supported
- GreaterThan: Supported
- GreaterThanOrEqual: Supported
- EqualTo: Supported

Arithmetic Operations

- Add: Supported
- Subtract: Supported
- Multiply: Supported
- Divide: Supported
- Remainder: Supported
- Sqrt: Supported
- UnaryMinus: Supported
- BitwiseAnd: Supported
- BitwiseOr: Supported
- BitwiseXor: Supported
- BitwiseNot: Supported
- Abs: Supported
- Logarithm: Supported
- Log: Supported
- Pow: Supported
- Ceil: Supported
- Floor: Supported
- Round: Supported
- Signum: Supported
- MaxOf: Not Supported
- Sin: Not Supported
- Cos: Not Supported
- Tan: Not Supported
- Asin: Not Supported
- Acos: Not Supported
- Atan: Not Supported

Hierarchy Expressions

- IsRoot: Supported
- Level: Supported

- PreRank: Supported
- PostRank: Supported
- IsLeaf: Supported
- IsDescendant: Supported
- IsDescendantOrSelf: Supported
- IsParent: Supported
- IsSibling: Supported
- IsSiblingOrSelf: Supported
- IsFollowing: Supported

Set Operations

- In
  - When the value to check for is Boolean: Not supported
  - Otherwise: Supported
- InSet: Supported

Miscellaneous

- SortOrder: Supported
- CaseWhen: Supported
- Literal: Supported
- AttributeReference: Supported
- Alias: Supported

4.17 Disk Engine and Relational Engine Data Source API

The disk engine and relational engine data source API has the following configurable options.

Name	Description	Default Value[1]	Example Values
files	Comma-separated list of the fully qualified names of the files to be uploaded to SAP Vora.To specify file-specific options (such as `storagebackend`, `format`, and so on) please use pipe-separated notation as explained in *Loading Data into Tables*.	-	path/to/file1.csv,path/to/file2.csv
tableschema	Table schema used to create the SAP Vora table. This parameter is only recommended if you want to use special SAP Vora data types that are not directly supported in Spark.	-	name varchar(*), age integer

Name	Description	Default Value[1]	Example Values
dateformat	Specification of global and per-column date formats. The first value without a column name is the global one.	-	• col1: 'YYYY_MM_DD', col2: 'DD:MM:YYYY' • 'MM/DD/YYYY' • 'MM/DD/YYYY', col1: 'YYYY_MM_DD', col2: 'DD:MM:YYYY'
format	Format used to read the data. SAP Vora supports "csv", "orc", and "parquet".	csv	csv,orc,parquet
storagebackend	Backend where the data to be loaded is stored. This might require additional parameters to be set depending on the storage type. Possible values are "hdfs", "local", or "s3".	hdfs	hdfs, local, s3
ps	Partition scheme used to partition a table	-	ps(a)
pscasesensitive	Indicates if the partition scheme is case sensitive	false	true
null	Specification of global and per-column null values to parse NULL fields in CSV files. The first value without a column name is the global one. All values have to be enclosed in single quotes.	Absent values in varchar/char columns and absent values or "" in columns of other types are interpreted as NULL.	• '', col1: 'null', col2: 'null' • 'null', col1: 'x' • '' (default) • col1: 'null', col2: '?'
ignoreinvalid	Continues to load even if some data points cannot be formatted according to the column specification. If the column is nullable, invalid fields are set to NULL, otherwise the column's default value is assigned.	false	true
csvdelimiter	Delimiter used to parse CSV files	,	;
csvquote	Quote character in CSV files	"	,
csvskip	Skips the first n lines in CSV files. The number must be non-negative.	0	10
csvthousandsdelimiter	Delimiter for thousands	,	:
csvdecimaldelimiter	Delimiter for decimals	.	@
csvescape	Character used for escaping strings in the CSV file	\	#
csvnoquote	True if no quotes should be used when loading csv files, false otherwise. If it is set to true, `csvquote` is ignored.	false	true
columns	Comma-separated list of column indexes to be loaded to SAP Vora. For example, "4,1,5" means that the 4th, 1st, and 5th columns of the input file are to be loaded. The number of columns in the schema definition must match the number of column indexes provided by this option.	-	4,1,5

Name	Description	Default Value[1]	Example Values
s3accesskeyid	Amazon S3 access key	-	somes3keyid
s3secretaccesskey	Amazon S3 secret access key	-	somes3accesskey
s3endpoint	Amazon S3 endpoint	-	s3endpoint.example.com
hdfsnamenode	HDFS Namenode from where data is to be loaded. If a default namenode is not set in the Spark default configuration, it needs to be set when reading from HDFS. When working with the SAP Vora tools, it always needs to be set when creating a table.	-	namenode.example.com:8020
host	Transaction coordinator host	-	txc.example.org
port	Transaction coordinator port	-	2202
catalog.host	Catalog host	-	catalog.example.org
catalog.port	Catalog port	-	1234
catalog.timeout	Timeout for catalog	3	5
schema	Schema to be used for operations on SAP Vora	VORA	CUSTOMSCHEMA
tabletype[2]	Type of the underlying table	datasource	streaming
maxvarcharlength	Maximum varchar length in a CREATE TABLE call. If this option is set, all columns of Spark type String (in the create call) are mapped to the SAP Vora type varchar(maxvarcharlength).	*	5000, 100, *
overwrite	Only considered during APPEND statements. This option overrides the content of the table with the new data. To do this, the former table is truncated before the new files are added. Note: This is not an atomic operation (like all save operations in Spark SQL). If append fails, the truncated table will not be recovered.	false	true
hivedb	Name of the Hive database to look up for metadata operations	default	myhivedb
hivetable	Name of the Hive table to import	-	myhivetable
hivefiles	Files to import into SAP Vora, relative to the folder where they are stored	-	/relative/file1,/relative/file2

[1] An empty cell indicates that there is no default value.

[2] Only supported by the disk engine.

> **Note**
>
> You can set all properties globally in the `spark-defaults.conf` file by adding the prefix `spark.vora.engines`.

Related Information

Loading Data into Tables [page 28]
Creating Partitioned Tables in the Relational Engines [page 26]

4.17.1 Date and Time Formats

The SAP Vora data source API option `dateformat` allows you to specify custom date and time formats.

The format specified using the `dateformat` option is not interpreted by Spark, but passed directly to the SAP Vora engines. The format specifiers are listed below:

Format Specifier	Description
YYYY	The year including the century. Range = [0001,9999].
YY	The year within the century. Range = [00,99]. Values in the range [69,99] refer to years 1969 to 1999 inclusive, and values in the range [00,68] refer to years 2000 to 2068 inclusive.

RRRR	The year with or without the century. Range = [0001,9999] or [00,99]. If the century is provided, the behavior is the same as for YYYY. If the century is not provided, the behavior is as follows:
	• The specified two-digit year is 00 to 49:
	  ○ When the last two digits of the current year are 00 to 49, the returned year has the same century as the current year.
	  ○ When the last two digits of the current year are 50 to 99, the century of the returned year is 1 greater than the century of the current year.
	• The specified two-digit year is 50 to 99:
	  ○ When the last two digits of the current year are 00 to 49, the century of the returned year is 1 less than the century of the current year.
	  ○ When the last two digits of the current year are 50 to 99, the returned year has the same century as the current year.

RR	The year without the century. Range = [00,99]. The behavior is the same as for RRRR when the century is omitted.
MM	The month number. Range = [01,12].
MON	Abbreviated month name (for example, DEC).
MONTH	Full month name (for example, DECEMBER).
DD	The day of the month. Range = [01,31].
DDD	The day number of the year. Range = [001,366].
HH	The hour in 12 or 24 hour format (depending on whether AP appears as well or not). Range = [00,23] or [01,12].
HH12	The hour in 12 hour format. Range = [01,12]. AP needs to appear as well.
HH24	The hour in 24 hour format. Range = [00,23].

Format Specifier	Description
AP	Meridian indicator. Range = {AM,PM}.
MI	The minute of the hour. Range = [00,59].
SS	The second of the minute. Range = [00,59].
SSSSS	The second of the day. Range = [00,86399].
FF[1-7]	The fraction of the second in 100 nanoseconds. Range = [0000000,9999999]. The precision specifier is optional; the default is 7. Leading zeros are recognized, for example, 0010000 yields 1 millisecond.
SF	The second of the minute with optional fraction (separated by a '.'), that is, SF is equal to SS[.FF7].

Note the following points:

- Format strings are case insensitive. For example, the format `yYyY-MM-dD FOObar` matches `2015-01-01 fooBAR`.
- For all numeric fields, leading zeros are permitted but not required (note the special behavior for `FF`).
- The default value for year, month, and day is 1. The default value for hour, minute, second, and fraction is 0.
- Time specifiers can occur in date formats and will be ignored, and vice versa.

Note the following restrictions:

- Only one month specifier can occur in a format.
- `DDD` cannot occur together with `DD` or any month specifier in a format.
- Only one hour specifier (`HH`, `HH12`, or `HH24`) can occur in a format. If `HH24` occurs, `AP` must not occur as well. If `HH12` occurs, `AP` needs to occur as well.
- `SSSSS` cannot occur together with `MI` or any hour specifier in a format.
- The given date expression must match the given format (trailing spaces are ignored if the expression is of type CHAR and either the format is omitted or it is also of type CHAR).
- The number of digits in a numeric field cannot exceed the length of the format specifier.
- The given date, time, and timestamp expression must be a valid date, time, and timestamp.

5 Working with Graphs, Documents, and Time Series

SAP Vora provides a document store, graph engine, and time series engine, which are integrated into Spark using a raw data source, `com.sap.spark.engines`.

Since `com.sap.spark.engines` is a raw data source, the SQL syntax extensions provided by these engines are not fully integrated into Spark SQL, which means that you need to use the following syntax instead:

- Raw SQL syntax for SELECT statements
- Parsed DDL syntax for DDL statements

Note that tables created with `com.sap.spark.engines` are always assigned to a single host in the cluster unless partitioning is applied.

Related Information

Raw SQL Syntax and Parsed DDL Syntax [page 58]
Processing Graph Data [page 60]
Analyzing Time Series Data [page 82]
Working with Collections (Document Store) [page 103]
Data Source Options [page 122]
Partitioning Tables [page 140]

5.1 Raw SQL Syntax and Parsed DDL Syntax

The raw data source `com.sap.spark.engines` supports the raw SQL syntax for SELECT statements and parsed DDL syntax for DDL statements .

Raw SQL Syntax

SAP Vora extends Spark SQL with a raw SQL interface. The raw SQL interface exposes the SQL dialect of the SAP Vora engines using the following syntax:

```
``<sqlString>`` using com.sap.spark.engines
```

Sample Code

```
``select * from TAB_TS`` using com.sap.spark.engines
```

The <sqlString> contained in the double backticks (``) is directly passed from Spark SQL to the SAP Vora engines without any further Spark processing (that is, without parsing, plan generation, and so on). The SAP Vora engines execute the SQL statement and return the result back to Spark by populating a Spark data frame.The raw SQL extension calculates the schema of the data frame based on the result of the query. If the schema is known in advance, it can be specified as follows:

```
``<sqlString>`` using com.sap.spark.engines AS (<schemaString>)
```

The <schemaString> follows the table schema syntax, for example:

```
``select col1, col2 from TAB_TS`` using com.sap.spark.engines AS (col1 INTEGER,
col2 String)
```

All engines (disk, relational, document store, graph, and time series) support the raw SQL interface.

Note

For the document store, graph engine, and the time series engine, raw SQL is the only interface.

Parsed DDL Syntax

DDL statements are parsed by the Spark SQL parser and do not need backticks:

```
<DDL_statement> using com.sap.spark.engines OPTIONS (<comma-separated-pairs-
list>)
```

Sample Code

```
create collection c partition by ps1(x) using com.sap.spark.engines options
(files "/path/to/file")
```

CREATE statements specify the files to be loaded and other options. The files are automatically added as data sources to the created database object (graph, collection, or time series) and are loaded directly.

You can add further files to the database object using the APPEND statement:

```
APPEND [COLLECTION | SERIES TABLE] <name> using com.sap.spark.engines OPTIONS
(<comma-separated-pairs-list>)
```

As with the CREATE statement, the specified files are added as data sources and are loaded directly.

Related Information

Raw SQL Syntax: Implications [page 60]

5.1.1 Raw SQL Syntax: Implications

Spark SQL is case sensitive while the SAP Vora engines are not case sensitive. This has the following implications when you mix Spark SQL with raw SQL for the disk engine and the relational engine.

- When using SAP Vora raw SQL, identifiers are capitalized when nonquoted and case sensitive when quoted (double quotation marks).
- When using Spark they are always case sensitive.

Consider the following cases:

- A case-insensitive identifier (for example, `table1`) in SAP Vora raw SQL is equal to the uppercase form (`TABLE1`) and is therefore referenced in Spark SQL in upper case (as `TABLE1`).
- A case-sensitive identifier (for example, `"table1"`) in SAP Vora raw SQL is case sensitive and is therefore referenced in Spark SQL as is, but without quotation marks (as `table1`).

These rules need to be considered when referencing existing SAP Vora tables in raw SQL commands. Otherwise, the tables might not be correctly found in the SAP Vora catalog.

Related Information

Raw SQL Syntax and Parsed DDL Syntax [page 58]

5.2 Processing Graph Data

The SAP Vora graph engine is a distributed in-memory graph engine that supports graph processing and allows you to execute typical graph operations on data stored in SAP Vora.

The graph engine uses a native, node-centric graph store for high performance analytical query processing. It supports directed and undirected graphs and has an underlying property graph model. Properties can currently be specified on nodes only, not on edges.

> **Note**
>
> Note the following restrictions:
>
> - Only block partitioning functions can be used for graphs.
> - SELECT * is not supported.
> - JOIN and UNION of graph queries with other (graph) queries are not supported.

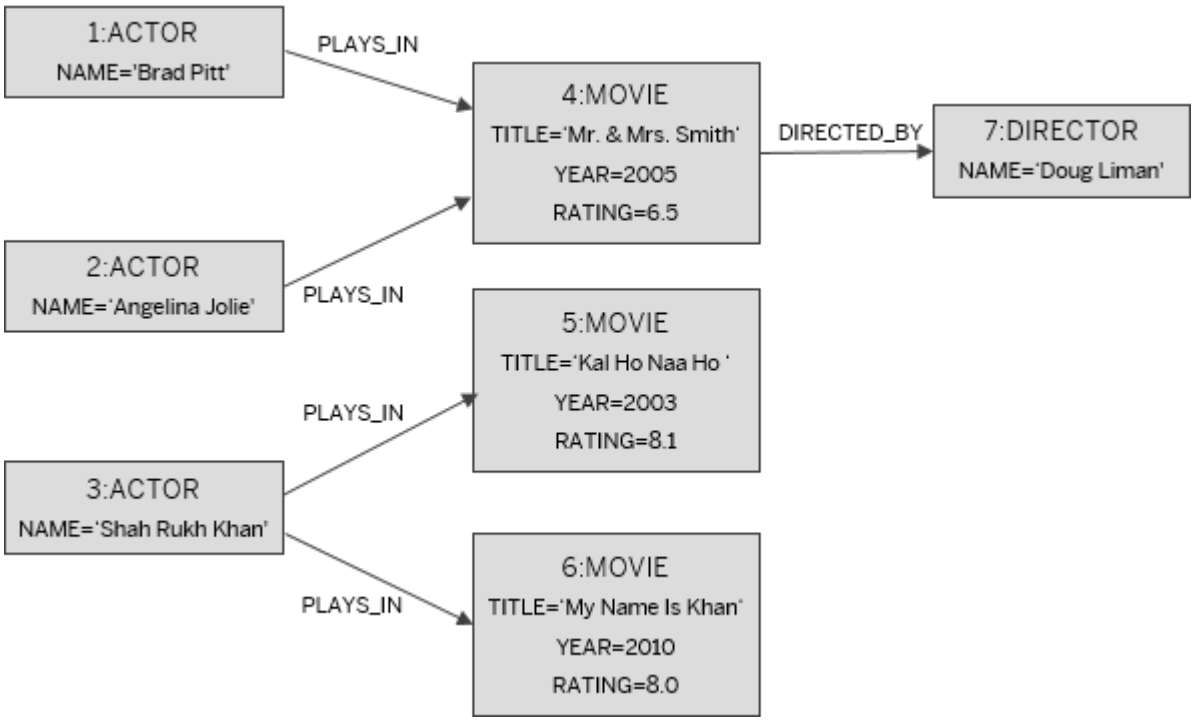
5.2.1 SAP Vora Graph Data Model

The underlying data model used in SAP Vora graphs are typed graphs with node properties. The SAP Vora graph engine supports both directed and undirected graphs, however, you cannot mix directed and undirected edges in one graph.

A graph consists of a set of nodes and accompanying metadata. A node has a type, a set of primitive-typed properties, and a set of outgoing edges. In undirected graphs, outgoing and incoming edges are equivalent. Like nodes, edges and properties are also typed. A distinction is made between the sets of node types, edge types, and property types. Therefore it is possible, but not recommended, to use the same names for node, property, and edge types. Edge properties are not supported.

The following figure shows a directed graph. This graph is used as an example in other topics in this section. It has the following:

- Three node types: ACTOR, MOVIE, DIRECTOR
- Four property types: NAME, TITLE, YEAR, RATING
- Two edge types: PLAYS_IN, DIRECTED_BY



The following primitive data types are supported in SAP Vora graphs: VARCHAR, BIGINT, DOUBLE.

All primitive types can have NULL values. Accessing a non-existent property of a node yields NULL.

5.2.2 Linked CSV Files

Graphs can be loaded using tabular data stored in CSV files. Multiple CSV files with different schemas can be combined to load data into one graph. The import supports fixed and variable node, property, and edge types. Both 1:n and n:m relationships can be directly imported as edges.

To load CSV data into a graph, an import definition needs to be specified in JSON format. This import definition has the following general structure:

- General import options
- Node import definitions
- Property import definitions
- Edge import definitions

5.2.2.1 General Import Options

General import options include the name of the graph, an undirected flag, and a list of property types to be indexed for fast lookup. All of these fields are optional. Node and edge import definitions are specified in the `nodes` and `edges` fields.

An overview of all general import definition fields is shown below:

Field	Type	Required	Description
name	String	No	Name of the graph
undirected	Boolean	No	Flag indicating whether the graph is undirected
property_indexes	List	No	Properties to be indexed for fast lookup
nodes	Object	No	Node import definitions
edges	Object	No	Edge import definitions

5.2.2.2 Shared Node, Property, and Edge Import Fields

Node, property, and edge import definitions share a common set of options.

These options include fields for specifying the path of the CSV file to be imported, the storage plugin to be used, a CSV separator, and a row offset. The type of nodes, properties, and edges can be either a fixed value per CSV file or derived from a column.

An overview of all shared node, property, and edge import definition fields is shown below:

Field	Type	Required	Description
file	String	Yes	Path of the CSV file to be imported
plugin_type	String	No	Import plugin to be used; the possible values are "hdfs" and "s3"

Field	Type	Required	Description
separator	String	No	CSV separator character; the default value is ","
row_offset	Integer	No	Number of rows to be skipped from the CSV file; the default value is 0
type	String	No	Fixed node/edge type name to be used; if not present, `type_col` must be set
type_col	Integer	No	Index of the node/edge type column; if not present, `type` must be set

5.2.2.3 Node Import Definitions

In addition to the shared import fields, node definitions must contain a primary key column index. Optional node import fields include a flag for ignoring primary key violations, and a field containing property import definitions.

An overview of the node-specific import definition fields is shown below:

Field	Type	Required	Description
primary_key_col	Integer	Yes	Index of the node's primary key column
ignore_primary_key_violations	Boolean	No	If set to true, primary key constraint violations are ignored
properties	Object	No	Property import definitions

5.2.2.4 Property Import Definitions

In addition to the shared import fields, property import definitions must contain the SQL type of the property to be imported and either a fixed value to be used for this property or the column index of the column containing the property values.

An overview of the property-specific import definition fields is shown below:

Field	Type	Required	Description
sql_type	String	Yes	SQL type of the property; the possible values are "VARCHAR", "BIGINT", and "DOUBLE"
value	String or numeric	No	Fixed property value to be used; if not present, `value_col` must be set
value_col	Integer	No	Index of the property value column; if not present, `value` must be set

5.2.2.5 Edge Import Definitions

In addition to the shared import fields, edge import definitions must contain fields for source or target node import definition names and foreign key columns.

An overview of the edge-specific import definition fields is shown below:

Field	Type	Required	Description
source_nodes	String or list	Yes	Name or list of names of source node import definitions
source_col	Integer	Yes	Index of the column containing the source node primary keys
target_nodes	String or list	Yes	Name or list of names of source node import definitions
target_col	Integer	Yes	Index of the column containing the target node primary keys

5.2.2.6 Example

The following example consists of three CSV files and an import definition given in JSON format.

```
persons.csv:
   "ACTOR","Brad Pitt"
   "ACTOR","Angelina Jolie"
   "ACTOR","Shah Rukh Khan"
   "DIRECTOR","Doug Liman"
```

```
movies.csv:
   "Mr. & Mrs. Smith",2005,6.5,"Doug Liman"
   "Kal Ho Naa Ho",2003,8.1,
   "My Name is Khan",2010,8.0,
```

```
plays_in.csv:
   "Brad Pitt","Mr. & Mrs. Smith"
   "Angelina Jolie","Mr. & Mrs. Smith"
   "Shah Rukh Khan","Kal Ho Naa Ho"
   "Shah Rukh Khan","My Name is Khan"
```

```
graph.json:
   {
     "name":"MOVIES",
     "import_type":"CSV",
     "property_indexes":["NAME","TITLE"],
     "nodes":{
       "persons":{
         "file":"persons.csv",
         "separator":",",
         "type_col":1,
         "primary_key_col":2,
         "properties":[
           {"type":"NAME","value_col":2,"sql_type":"VARCHAR"}
         ]
       },
       "movies":{
```

```
        "file":"movies.csv",
        "type":"MOVIE",
        "primary_key_col":1,
        "properties":[
          {"type":"TITLE","value_col":1,"sql_type":"VARCHAR"},
          {"type":"YEAR","value_col":2,"sql_type":"BIGINT"},
          {"type":"RATING","value_col":3,"sql_type":"DOUBLE"}
        ]
      }
    },
    "edges":{
      "plays_in":{
        "file":"plays_in.csv",
        "type":"PLAYS_IN",
        "source_nodes":"persons",
        "source_col":1,
        "target_nodes":"movies",
        "target_col":2
      },
     "directed_by":{
        "file":"movies.csv",
        "type":"DIRECTED_BY",
        "source_nodes":"movies",
        "source_col":1,
        "target_nodes":"persons",
        "target_col":4
      }
    }
  }
```

5.2.3 JSG Files

The supported file format for loading graphs is JSG. JSG files are files with a line-based JSON format and a JSG extension. In the JSG file format, each line is a JSON array that describes one node of the graph.

A node array consists of the following elements:

- Node type (string, upper case recommended)
- Node ID (unsigned 64-bit integer, strictly greater than zero)
- Properties (object of type-value pairs)
- Edges (object of type-value pairs)

Additional metadata can be specified in an optional header. The header must be prefixed by a hash sign and consists of a JSON object. The following table summarizes the supported fields of the header object:

Field	Type	Required	Description
name	String	No	Name of the graph
undirected	Boolean	No	Flag indicating whether the graph is undirected
property_indexes	List	No	Properties to be indexed for fast lookup

For example:

Sample Code

movies.jsg

```
# {"name":"MOVIES","property_indexes":["NAME","TITLE"]}
["ACTOR",1,{"NAME":"Brad Pitt"},{"PLAYS_IN":[4]}]
["ACTOR",2,{"NAME":"Angelina Jolie"},{"PLAYS_IN":[4]}]
["ACTOR",3,{"NAME":"Shah Rukh Khan"},{"PLAYS_IN":[5,6]}]
["MOVIE",4,{"TITLE":"Mr. & Mrs Smith","YEAR":2005,"RATING":6.5},{"DIRECTED_BY":
[7]}]
["MOVIE",5,{"TITLE":"Kal Ho Naa Ho","YEAR":2003,"RATING":8.1},{}]
["MOVIE",6,{"TITLE":"My Name is Khan","YEAR":2010,"RATING":8.0},{}]
["DIRECTOR",7,{"NAME":"Doug Liman"},{}]
```

Note that the JSG format supported in SAP Vora 1.x uses arrays to store the properties and edges of nodes. This format is also supported in SAP Vora 2.x.

Related Information

Creating Graphs [page 66]

5.2.4 Creating Graphs

Graphs can be created and loaded in SAP Vora using the CREATE GRAPH statement. Files can be loaded from local file systems, HDFS, and Amazon S3. Both linked CSV and JSG files can be used for graph creation.

Creating Non-Partitioned Graphs

You can create and load a non-partitioned graph using the following syntax (note that there is no default partitioning):

Sample Code

```
CREATE GRAPH MOVIES
  USING com.sap.spark.engines
  OPTIONS (
    files 'graph.json',
    storagebackend "hdfs"
  );
```

Creating Partitioned Graphs

To create a partitioned graph, you need to define a partition function and partition scheme. You can create the partition function and scheme as follows:

Sample Code

```
CREATE PARTITION FUNCTION PF(X BIGINT) AS BLOCK(X) PARTITIONS 10 BLOCKSIZE
1000 USING com.sap.spark.engines;
CREATE PARTITION SCHEME PS USING PF USING com.sap.spark.engines;
```

Note that you need to use block partition functions to partition graphs. Block partition functions require you to specify the number of partitions and the block size. The set of graph nodes is partitioned by assigning nodes to partitions in round-robin style, based on the following formula:

```
P(NODEID) = (NODEID / BLOCKSIZE) % PARTITIONS
```

The BLOCKSIZE and PARTITIONS parameters must both be strictly greater than zero. The recommended values for these two parameters are shown below:

Parameter	Recommended Value
PARTITIONS	Number of workers
BLOCKSIZE	1000

The partition function must be defined with exactly one parameter of type BIGINT. When creating the graph, you need to instantiate this parameter with the NODEID property, as shown in the example below:

Sample Code

```
CREATE GRAPH MOVIES
  PARTITION BY PS(NODEID)
  USING com.sap.spark.engines
  OPTIONS (
    files 'graph.json',
    storagebackend "hdfs"
  );
```

Related Information

Linked CSV Files [page 62]
JSG Files [page 65]

5.2.5 Dropping Graphs

You can remove a graph using the DROP GRAPH command.

Sample Code

```
DROP GRAPH MOVIES USING com.sap.spark.engines;
```

5.2.6 Graph Query Language

The graph query language is closely aligned with the SQL standard. It uses, where possible, SQL primitives and extends them with graph-specific features. Some of these extensions can be also found in a similar form in the query language of SAP HANA Core Data Services (CDS).

Keywords

Following the SQL standard, keywords used in the query language are case-insensitive. Keywords are recognized relative to their position in the query. Therefore it is in principle possible, but not recommended, to also use keywords as identifiers.

Identifiers

Identifiers are non-empty character strings, starting with a letter or underscore followed by a possibly empty sequence of letters, digits, or underscores:

```
identifier = ( letter | "_" ) ( letter | digit | "_" )*
```

Identifiers are case-insensitive and are automatically converted to uppercase. For example, the identifier `Actor` is interpreted as `ACTOR`. However, specific cases can be enforced by surrounding the identifiers with double quotes, for example, `"Actor"`.

Note that during graph creation using node, edge, or mapping tables, identifiers are case-sensitive. This means that in the process of graph creation, you are advised to use only uppercase identifiers for node, edge, and property types. Otherwise, the exact case will have to be used in the query language and the identifiers surrounded with double quotes.

Literals

Literals are constants used in the query language. The following literals are supported:

Primitive Type	Example Literal
String	'Hello world'
Integer	42
Float	8.5

The syntax for literals is defined as follows:

```
literal        = stringliteral | integerliteral | floatliteral
stringliteral  = "'" non-single-quote-character* "'"
integerliteral = [ "-" | "+" ] digit+
floatliteral   = [ "-" | "+" ] digit+ "." digit+
```

Paths

Similar to the query language of SAP HANA Core Data Services, the graph query language extends SQL with path expressions. A path is a sequence of identifiers separated by dots:

```
path = identifier ( "." identifier )*
```

The length of a path is defined as the number of identifiers occurring in it.

5.2.7 Graph Query Structure

Graph queries are read-only operations on a given target graph and can include pattern matching, traversals, and other graph or relational operations.

The result of executing a query is always a table containing primitive-type values. Graph queries can therefore in principle be embedded in and combined with standard SQL queries.

Syntactically, graph queries follow the structure of SELECT statements in SQL. Specifically, graph queries have the following structure:

```
graphquery = selectclause [ whereclause ] [ orderbyclause ] [ limitclause ]
```

The ORDER BY and LIMIT clauses follow standard SQL syntax. The syntax and semantics of the SELECT and WHERE clauses are explained in separate topics.

> **Note**
>
> Use the raw SELECT syntax for SELECT statements (this is not shown in the code examples).

Related Information

SELECT Clauses [page 70]
WHERE Clauses [page 74]
Raw SQL Syntax and Parsed DDL Syntax [page 58]

5.2.7.1 SELECT Clauses

SELECT clauses start with the SELECT keyword followed by a list of features, the FROM keyword, a variable list, and finally the USING GRAPH clause, as shown below.

```
selectclause = SELECT featurelist FROM variablelist [ USING GRAPH graphname ]
graphname    = [ identifier "." ] identifier
```

- USING GRAPH
  The USING GRAPH clause is mandatory and indicates that the query is a graph query. It is used to determine the graph on which the query should be executed. The graph name consists of one or two identifiers separated by a dot. If two identifiers are used, the first identifier is the name of a schema in the database catalog. The second identifier is the name of a graph in the database catalog. If no schema is specified, the schema currently set is used.
- FROM
  The FROM clause consists of a list of variables. A variable is a declaration of a node, which consists of a node type and an optional variable alias. As in SQL, the variable alias can be preceded by the optional AS keyword.
  Syntax:

  ```
  variablelist  = variable ( "," variable )*
  variable      = nodetype [ [ AS ] variablealias ]
  nodetype      = identifier
  variablealias = identifier
  ```

- SELECT
  The SELECT clause consists of a list of features. A feature can be the star symbol or an expression, which in turn is a literal, a path, an aggregation, or a function call. For a list of supported aggregations and functions, and information about how to use them, see the *Functions* section.
  Syntax:

  ```
  featurelist    = "*" | ( feature ( "," feature )* )
  feature        = ( expression | aggregation ) [ AS identifier ]
  expression     = literal | path | functioncall
  functioncall   = identifier "(" ( functionflag )* expression ( ","
  expression )* ")"
  functionflag   = identifier
  aggregation    = identifier "(" ( "*" | expression ) ")"
  ```

  The most important kind of feature are paths. The first element of every used path must be the alias of a variable defined in the FROM clause or, if only one variable is defined, an edge or property type. The middle part of a path is a possibly empty sequence of edge types. A property type can be specified as the last element of a path feature. Thus, the structure of path features can be formally defined as follows:

  ```
  featurepath  = ( variablealias ( "." edgetype)* [ propertytype ] ) |
  ```

```
                ( edgetype ( "." edgetype)* [ propertytype ] ) |
                ( propertytype )
edgetype     = identifier
propertytype = identifier
```

If there are name conflicts between variable aliases, edge types, or property types, the following priorities are used to resolve conflicts:

- Variable aliases have precedence over property types
- Property types have precedence over edge types

Related Information

Path Semantics [page 71]

5.2.7.2 Path Semantics

The semantics for evaluating feature lists with simple node property paths is the same as in standard SQL.

For every variable declared in the FROM clause, nodes in the target graph are matched and their properties used to evaluate the features defined in the SELECT clause. For every feature in the feature list, a corresponding column is created in the result table.

If paths with edge types are used, the standard SQL semantics is extended by interpreting paths as bounded traversals of edges in the graph. In relational terminology, the semantics of such a bounded traversal is equivalent to a (repeated) left outer join.

In graph terminology, the evaluation semantics of paths relies on the notion of a "match". A match is a mapping of the variables in the FROM clause to nodes in the target graph such that the conditions in the WHERE clause of the query are met. For every match of a query, each feature in the SELECT clause is evaluated and elements are added to the columns of the result table.

Evaluating a path in a graph corresponds to a traversal, starting at the matched node and looking up the requested property in the final node. If for a matched node the path cannot be completely evaluated or the requested property was not found, NULL values are added to the result.

If multiple path features are used in a query, the result for every match is calculated as a cross join of the respective feature results for that match.

Example

Query:

```
SELECT NAME, PLAYS_IN.TITLE FROM ACTOR USING GRAPH MOVIES
```

Result:

NAME	PLAYS_IN.TITLE
'Brad Pitt'	'Mr. & Mrs. Smith'

NAME	PLAYS_IN.TITLE
'Angelina Jolie'	'Mr. & Mrs. Smith'
'Shah Rukh Khan'	'Kal Ho Naa Ho'
'Shah Rukh Khan'	'My Name Is Khan'

Three different actor nodes are matched by this query. The first two actors each play in one movie, which is also the same movie. The third actor plays in two movies and therefore generates two rows in the result table.

Example

Query:

```
SELECT A.NAME, A.PLAYS_IN.DIRECTED_BY.NAME FROM ACTOR A USING GRAPH MOVIES
```

Result:

A.NAME	A.PLAYS_IN.DIRECTED_BY.NAME
'Brad Pitt'	'Doug Liman'
'Angelina Jolie'	'Doug Liman'
'Shah Rukh Khan'	NULL
'Shah Rukh Khan'	NULL

Four (partial) paths are found and corresponding entries generated in the result table. In those cases where the paths are incomplete, NULL values are added to the result. Note that a variable alias is used in this example.

5.2.7.3 Wildcard Node and Edge Types

The reserved keyword ANY can be used to specify that there are no restrictions on a node type or edge type.

The keyword can be used in the FROM clause of a query to match a node of any node type. In paths used in the SELECT or WHERE clauses of a query, the ANY keyword can be used to refer to edges of any type.

Example

Query:

```
SELECT NAME FROM ANY USING GRAPH MOVIES
```

Result:

NAME
'Brad Pitt'

NAME
'Angelina Jolie'
'Shah Rukh Khan'
'Doug Liman'
NULL
NULL
NULL

This query matches all nodes of the graph irrespective of their type and inspects the NAME property. Note that for the two MOVIE nodes, this property is not defined and therefore NULL values are returned.

5.2.7.4 Predefined Properties

All graph nodes have the following implicitly defined properties.

Node IDs

Graph nodes are identified internally by means of a unique node ID. This node ID is given by a strictly positive integer and can be inspected using the implicitly defined property NODEID.

If a feature path in a SELECT statement does not contain a property type as its last element, the implicitly defined NODEID property is used as the default property.

Node Types

To access and constrain the node type of a variable declared using the ANY keyword, the implicitly defined property NODETYPE can be used. This property is of type string and contains the name of the node type.

Properties

The predefined property PROPERTIES enables all properties of a given node to be accessed as a JSON object. This JSON object consists of type-value pairs and is equivalent to the properties object in the JSG file format.

Edges

The predefined property EDGES enables all properties of a given node to be accessed as a JSON object. This JSON object consists of type-value pairs and is equivalent to the edges object in the JSG file format. The values are arrays consisting of target node IDs.

Example

Query:

```
SELECT NODETYPE, NODEID, PROPERTIES, EDGES FROM ANY USING GRAPH MOVIES;
```

Possible result:

NODETYPE	NODEID	PROPERTIES	EDGES
'ACTOR'	1	'{"NAME":"Brad Pitt"}'	'{"PLAYS_IN":[4]}'
'ACTOR'	2	'{"NAME":"Angelina Jolie"}'	'{"PLAYS_IN":[4]}'
'ACTOR'	3	'{"NAME":"Shah Rukh Khan"}'	'{"PLAYS_IN":[5,6]}'
'MOVIE'	4	'{"TITLE":"Mr. & Mrs Smith","YEAR":2005,"RATING":6.5}'	'{"DIRECTED_BY":[7]}'
'MOVIE'	5	'{"TITLE":"Kal Ho Naa Ho","YEAR":2003,"RATING":8.1}'	'{}'
'MOVIE'	6	'{"TITLE":"My Name is Khan","YEAR":2010,"RATING": 8.0}'	'{}'
'DIRECTOR'	7	'{"NAME":"Doug Liman"}'	'{}'

5.2.7.5 WHERE Clauses

WHERE clauses can be used to apply filtering as in standard SQL, but also to define graph patterns that restrict the set of matches to be found. This is particularly useful if more than one variable is used.

The syntax of WHERE clauses is shown below:

```
whereclause  = WHERE condition
condition    = andcondition ( OR andcondition )*
andcondition = term ( AND term )*
term         = ( NOT term ) | predicate | comparison | ( "(" condition ")" )
predicate    = path ( ( IN path ) | ( IS [ NOT ] NULL ) | ( LIKE
stringliteral ) )
comparison   = ( path | functioncall ) ("=" | "<>" | "<" | ">" | "<=" | ">=")
expression
```

Conditions in the WHERE clause can be constructed using the standard logical operators OR, AND, and NOT. The IS NULL and LIKE keywords are available for node properties and can be used as in standard SQL.

Comparisons are allowed between features, that is, paths, literals, or function calls. The specified paths must refer to properties of matched nodes. This is achieved using paths of length two, consisting of a variable alias and a property type.

For graph pattern matching, the IN keyword can be used to check whether edges exist between matched nodes. Note that this use of the IN keyword differs from the standard SQL syntax and semantics. The syntax for edge checks is as follows.

```
inpredicate    = targetvariable IN sourcevariable.edgetype
sourcevariable = identifier
targetvariable = identifier
```

Example

Query:

```
SELECT A.NAME FROM ACTOR A, MOVIE M USING GRAPH MOVIES WHERE M.TITLE='Mr. &
Mrs. Smith' AND M IN A.PLAYS_IN
```

Result:

A.NAME
'Brad Pitt'
'Angelina Jolie'

This query finds all actors who played in the movie with the title 'Mr. & Mrs. Smith'. The condition (M IN A.PLAYS_IN) asserts that there must be an edge of type PLAYS_IN between the matched actor and the matched movie. In other words, the matched movie must be reachable from the matched actor by an edge of type PLAYS_IN.

Example

Query:

```
SELECT A.NAME, B.NAME FROM ACTOR A, ACTOR B, MOVIE M USING GRAPH MOVIES WHERE
(M IN A.PLAYS_IN) AND (M IN B.PLAYS_IN)
```

Result:

A.NAME	B.NAME
'Brad Pitt'	'Brad Pitt'
'Brad Pitt'	'Angelina Jolie'
'Angelina Jolie'	'Brad Pitt'
'Angelina Jolie'	'Angelina Jolie'
'Shah Rukh Khan'	'Shah Rukh Khan'
'Shah Rukh Khan'	'Shah Rukh Khan'

This query finds all triples of actor pairs and movies where both actors have an outgoing edge to the matched movie. Parentheses are used in this query to improve the readability. Note that there are symmetric matches and non-distinct matches.

Symmetric and non-distinct matches can be avoided by defining an order on the matched nodes. For performance reasons, it is advised to use the implicitly defined NODEID property for this purpose.

Example

Query:

```
SELECT A.NAME, B.NAME FROM ACTOR A, ACTOR B, MOVIE M USING GRAPH MOVIES WHERE
M IN A.PLAYS_IN AND M IN B.PLAYS_IN AND A<B
```

Result:

A.NAME	B.NAME
'Brad Pitt'	'Angelina Jolie'

No properties are specified on the left-hand side or right-hand-side of the condition A<B. Therefore the node ID properties are used by default and the constraint is treated as A.NODEID<B.NODEID. This additional comparison between the node IDs of the matched nodes ensures that there are no symmetric or non-distinct matches.

5.2.8 Graph Functions

Functions can be used in the features of the SELECT clause and conditions of the WHERE clause in a query.

Aggregation Functions

Standard SQL aggregation functions can be used in the SELECT clause. The following table summarizes the supported aggregation functions:

Aggregation Function	Description
COUNT	Number of result rows
MIN	Minimum value of a result set column
MAX	Maximum value of a result set column
SUM	Sum of the values in a result set column
AVG	Average (arithmetic mean) of the values in a result set column

Graph Functions

A number of additional graph-specific functions are supported. An overview of the available graph functions is shown in the table below:

Graph Function	Description	SELECT Clause	WHERE Clause
DEGREE	Node degree	Yes	Yes
DISTANCE	Node distance	Yes	Yes
CONNECTED_COMPONENT	(Strongly) connected component	Yes	No

These functions are described in separate topics.

5.2.8.1 Degree

The degree of a node is its number of incoming and outgoing edges. The in-degree of a node is its number of incoming edges, while the out-degree of a node is its number of outgoing edges. The DEGREE function can be used to calculate these measures.

The DEGREE function has an optional flag, which can be any one of the keywords IN, OUT, or INOUT, indicating whether the in-degree, out-degree, or degree should be computed. The default value of this flag is INOUT. In addition, the function expects the variable of the query for which the degree should be calculated to be passed as an argument.

For any node x in a directed graph, it holds that DEGREE(INOUT X) = DEGREE(IN X) + DEGREE(OUT X). For undirected graphs, in contrast, the function variants are equivalent and therefore the equality DEGREE(INOUT X) = DEGREE(IN X) = DEGREE(OUT X) holds.

The degree function can be used in both the SELECT and WHERE clauses of queries. When using the degree function in the WHERE clause, only comparisons with integer literals (constants) are allowed.

Example

Query:

```
SELECT M.TITLE, DEGREE(IN M) AS DI, DEGREE(OUT M) AS DO, DEGREE(INOUT M) AS
DIO FROM MOVIE M USING GRAPH MOVIES
```

Result:

M.TITLE	DI	DO	DIO
'Mr. & Mrs. Smith'	2	1	3
'Kal Ho Naa Ho'	1	0	1
'My Name Is Khan'	1	0	1

Example

Query:

```
SELECT M.TITLE FROM MOVIE M USING GRAPH MOVIES WHERE DEGREE(IN M)>1
```

Result:

M.TITLE
'Mr. & Mrs. Smith'

5.2.8.2 Distance

The DISTANCE function calculates the distance of the shortest directed or undirected path between two nodes. A path is a non-empty sequence of edges connecting a source node with a target node.

In a directed path, all edges point in the direction of the path. In an undirected path, edges can point in arbitrary directions.

The DISTANCE function expects two arguments that must be variables of the query. These variables are the source and the target nodes, respectively.

The optional flag DIRECTED or UNDIRECTED can be specified for the distance function to indicate whether directed or undirected paths should be considered. In addition, the flag EDGETYPE followed by the name of an edge type or ANY can be specified to restrict the edge types to be considered for evaluating the paths.

If there is no (undirected) path between the two nodes, the DISTANCE function returns NULL. Note that DISTANCE(DIRECTED a,b) calculates the distance of the shortest path from a to b, but not from b to a.

The DISTANCE function can be used in the SELECT and WHERE clauses of a query. In the WHERE clause of a query, the DISTANCE function can be used only in comparisons with integer literals (constants).

Example

Query:

```
SELECT A.NAME, B.NAME, DISTANCE(DIRECTED A,B) AS D FROM ACTOR A, DIRECTOR B
USING GRAPH MOVIES
```

Result:

A.NAME	B.NAME	D
'Brad Pitt'	'Doug Liman'	2
'Angelina Jolie'	'Doug Liman'	2
'Shah Rukh Khan'	'Doug Liman'	NULL

Example

Query:

```
SELECT B.NAME, DISTANCE(UNDIRECTED A,B) AS D FROM ACTOR A, ACTOR B USING GRAPH
MOVIES WHERE A.NAME='Brad Pitt'
```

Result:

B.NAME	D
'Brad Pitt'	0
'Angelina Jolie'	2
'Shah Rukh Khan'	NULL

Example

Query:

```
SELECT A.NAME, B.NAME FROM ACTOR A, DIRECTOR B USING GRAPH MOVIES WHERE
DISTANCE(A,B)=2
```

Result:

A.NAME	B.NAME
'Brad Pitt'	'Doug Liman'
'Angelina Jolie'	'Doug Liman'

5.2.8.3 Connected Components

Connected components and strongly connected components can be computed using the CONNECTED_COMPONENT function. This function returns an ID for the (strongly) connected component of a particular node.

The CONNECTED_COMPONENT function expects one argument that must be a variable of the query and represents the matched nodes for which the connected component should be calculated. In the result set, a column of type integer is generated containing the IDs of the calculated (strongly) connected components of the matched nodes.

An optional flag can be specified with either the keyword STRONG or WEAK. If set to STRONG, strongly connected components are computed (taking into account the direction of edges). If set to WEAK, connected components are computed (ignoring the direction of edges). Additionally, the optional flag EDGETYPE followed by the name of an edge type or the keyword ANY can be specified to restrict the connections allowed within a (strongly) connected component to a specific edge type. If the ANY keyword is used, there is no restriction on the edge types used.

The connected component IDs are derived from the node IDs. The ID of a connected component is determined by the smallest node ID that occurs in it.

The function for calculating connected components is available only in the SELECT clause of a query.

Example

Query:

```
SELECT A.NAME, CONNECTED_COMPONENT(STRONG A) AS SCC, CONNECTED_COMPONENT(WEAK
a) AS WCC FROM ACTOR A USING GRAPH MOVIES
```

Result:

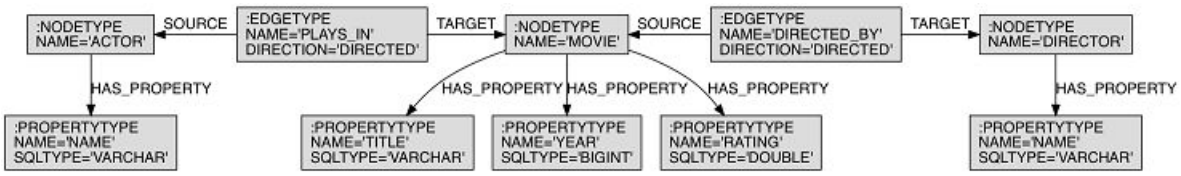
A.NAME	SCC	WCC
'Brad Pitt'	1	1
'Angelina Jolie'	2	1
'Shah Rukh Khan'	3	3

Each actor node forms its own strongly connected component, since there are no two actors that are mutually reachable from each other by directed paths. However, the nodes 'Brad Pitt' and 'Angelina Jolie' are in the same (not strongly) connected component because there is an undirected path between these two nodes.

5.2.9 Graph Metafunction: Type Information

Type information about the nodes, edges, and properties of a graph can be obtained in the form of a type graph. You can access the information in a type graph using SQL queries.

The type graph for the example graph is shown below:



You can list all node types in the graph as follows:

Example

Query:

```
SELECT NAME FROM NODETYPE USING GRAPH TYPE_INFO(MOVIES);
```

Result:

NAME
ACTOR
MOVIE
DIRECTOR

You can use similar queries for edges. The following query returns the type information for each edge type in the graph. Note that the DIRECTION of an edge can be either DIRECTED or UNDIRECTED. This depends on the

type of graph, that is, for a directed graph all edges are directed, and for an undirected graph all edges are undirected:

Example

Query:

```
SELECT SRC.NAME,E.NAME,TGT.NAME,E.DIRECTION FROM NODETYPE SRC, EDGETYPE E,
NODETYPE TGT USING GRAPH TYPE_INFO(MOVIES) WHERE SRC IN E.SOURCE AND TGT IN
E.TARGET;
```

Result:

SRC.NAME	E.NAME	TGT.NAME	E.DIRECTION
ACTOR	PLAYS_IN	MOVIE	DIRECTED
MOVIE	DIRECTED_BY	DIRECTOR	DIRECTED

You can list all property types present in the graph as shown below:

Example

Query:

```
SELECT V.NAME,P.NAME,P.SQLTYPE FROM NODETYPE V, PROPERTYTYPE P USING GRAPH
TYPE_INFO(MOVIES) WHERE P IN V.HAS_PROPERTY;
```

Result:

V.NAME	P.NAME	P.SQLTYPE
ACTOR	NAME	VARCHAR
MOVIE	TITLE	VARCHAR
MOVIE	YEAR	BIGINT
MOVIE	RATING	DOUBLE
DIRECTOR	NAME	VARCHAR

5.3 Analyzing Time Series Data

The SAP Vora time series engine enhances the SAP Vora in-memory engine by enabling time series data to be efficiently analyzed in distributed environments. It supports highly compressed time series storage and time series analysis algorithms that work directly on top of the compressed data.

5.3.1 Data Definition Statements

Data definition statements are used to create and define time series. This includes definitions of the respective columns as well as basic definitions of the rules the series complies with.

In addition, you can define a partitioning of the time series into multiple storage objects that can be distributed over multiple nodes.

> **Note**
>
> The time series engine currently only supports a single multivariate time series per table.

5.3.1.1 CREATE Statement

Use the CREATE TABLE statement together with the SERIES clause to define a time series table. Use the PARTITION BY clause in addition to create a partitioned time series table.

Syntax

```
<create series> ::= CREATE TABLE <table_name>
<table_content>
<series_clause>
[<table_type>]
[<partition_clause>]
<spark_engine_clause>
<spark_options_clause>
```

`<table_name>`

The table name is a simple identifier in the form of a string literal. It is used to identify a time series.

```
<table_name> ::= <identifier>
<identifier> ::= string literal
```

<table_content>

You can create a series table by describing the columns and their data types, including the time stamp column.

```
<table_content> ::=
'(' <table_elements>, ... ')'
```

<table_element>

A table element defines a time series data column with its name, data type, and definitions for default values and automatic value generation.

```
<table_element> ::= <column_definition>
<column_definition> ::= <column_name> <data_type>
```

<column_name>

Defines the name of the column.

```
<column_name> ::= <identifier>
```

<data_type>

```
<data_type> ::= TINYINT | SMALLINT | INTEGER | BIGINT |
REAL | DOUBLE | TIMESTAMP
```

Currently only a single time column is allowed for time series managed in the time series engine. This column should be configured as the period column.

<series_clause>

The series clause defines the basic properties of a series table, including time-specific parameters and compression settings.

```
<series_clause> ::=
    SERIES '('
        <series_clause_parameter>, ...
        [<compression_clause>]
    ')'
```

<series_clause_parameter>

The series clause parameter list defines series-specific parameters, such as period, begin and end timestamps, equidistance parameters, and series keys. The parameters can be provided in an arbitrary order, but duplicated clauses are not allowed. At a minimum, the period clause as well as the range expression have to be defined.

```
<series_clause_parameter> ::=
      <series_key_clause>
    | <series_period>
    | <range_expression>
    | <equidistance_definition>
```

<series_key_clause>

The series key clause identifies the tuple of columns that make up the series key. Together with the period columns, they form the primary key constraint of the table.

```
<series_key_clause> ::= SERIES KEY '(' <identifier>, ... ')'
```

Defining the series key clause allows you to combine an arbitrary number of time series in one table, each identified by a series key. Each time series is physically separated when stored.

Table functions such as GRANULIZE or CORRELATION can be calculated on separated time series only. Using a series key clause therefore forces you to use a PARTITION BY clause for these table functions.

<series_period>

The period defines the column or tuple that is used as the identifying, counting reference for the series. For instance-based series, you select a single period column.

```
<series_period> ::= PERIOD FOR SERIES '('
<column_identifier>, ... ')'
```

Note that the syntax without parentheses is deprecated and might be removed in the future: `PERIOD FOR SERIES <column>`

<range_expression>

The range expression defines the time range of a series. The range definition is defined as a right half-open interval, that is, [start,end). The start time is considered to be a strict limit, so no data before that time should be added to the time series. The end time is considered a soft definition, where the end time is automatically increased if data later than this end time is added.

```
<range_expression> ::=
[<series_start>] [<series_end>] [<population_clause>]
<series_start> ::=
    START <time_rep> '<date>'
  | MINVALUE <time_rep> '<min_value>'
<series_end> ::=
    END TIMESTAMP '<date>'
  | MAXVALUE TIMESTAMP '<max_value>'
```

After create, an equidistant time series will be pre-initialized with NULL values for the defined range.

> **Note**
>
> Specifing a range expression is, by design, not compatible with auto-partitioning, such as interval partitioning or key-time partitioning. This is because when START and END are specified, an equidistant series is pre-initialized. When using auto-partitioning, however, partitions are created as needed during ingestion and therefore cannot be pre-initialized during create.

<equidistance_definition>

Defines whether a series is equidistant, that is, the distance between all adjacent time stamps comply with a given interval. The interval value is defined by an `<interval`

const>, meaning a number literal plus a respective time unit. The following time units are supported: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, MILLISECOND.

```
<equidistance_definition> ::=
NOT EQUIDISTANT
| EQUIDISTANT INCREMENT BY <interval const>
```

<compression_clause>

Defines a compression strategy for the time series. A compression strategy comprises a compression technique that is used for a specific column of a time series. The given error bound is calculated based on the median of all values contained in a single partition. The median is then multiplied with the given percentage error and the result is used as the error bound for each single value in the positive and negative direction. If a column is not named in the compression clause, no compression is applied.

```
compression_clause> ::= [<default_compression>]
<compression_definition>[ <compression_definition>...]
<default_compression> ::= DEFAULT COMPRESSION USE
(<compression_type>)
<compression_definition> ::= COMPRESSION ON <column_def>
<comp_def>
<column_def> ::=
  <column>
| (<column_name_list>)
<comp_def> ::= USE <compression_identifier> [<error_bound>]
<error_bound> ::= ERROR exact_numeric_literal PERCENT
<compression_identifier> ::= APCA | SDT | SPLINE
<column_name_list> ::= <list_entry> [, <list_entry>, ...]
<list_entry> ::= <column> | <column> .. <column>
```

<table_type>

Tables can be designated as streaming tables. Streaming tables persist their content in DLog, which enables the cluster to recover the data after a restart or failure.

```
<table_type> ::= TYPE STREAMING
```

Streaming tables allow interval partitioning to be used. They do not support period range predicates or range expressions.

<partition_clause>

The time series engine uses SAP Vora's standard partitioning syntax, which consists of the definitions of a partition function and a partition scheme. The partition function defines how partitions are created. It is important to note that the time series engine does not support all partitioning types that other engines might support.

The partition clause is the reference added to the CREATE TABLE syntax. The columns provided need to be defined in the list of column definitions. For range and interval partitioning, the period column has to passed.

For hash partitioning, only key columns can be used.The same applies for key-time partitioning, which is a combination of hash and interval partitioning.

```
<partition_function> ::=
  CREATE PARTITION FUNCTION <identifier> '(' <pf_parameter>, ... ')' AS
<pf_definition>;
<pf_parameter> ::= <identifier> <type>
<pf_definition> ::=
      <pf_definition_range_partitioning>
    | <pf_definition_hash_partitioning>
    | <pf_definition_interval_partitioning>
    | <pf_definition_key_time_partitioning>
<pf_definition_range_partitioning> ::= RANGE '(' <literal>, ... ')'
<pf_definition_hash_partitioning> ::= HASH '(' <identifier>, ... ')'
    MIN PARTITIONS <unsigned_integer>
    [ MAX PARTITIONS <unsigned_integer> ]
<pf_definition_interval_partitioning> ::= INTERVAL '(' <identifier> ')'
    <interval_string> <interval_qualifier>
<pf_definition_key_time_partitioning> ::=
    <pf_definition_hash_partitioning>
    <pf_definition_interval_partitioning>
<partition_scheme> ::= CREATE PARTITION SCHEME <identifier> USING <identifier>;
<partition_clause> ::= PARTITION BY '(' <identifier>, ... ')'
```

See also ISO/IEC 9075-2:2011, Clause 10.1 <interval qualifier>.

<range partitioning>

The data is partitioned by the period column, by dividing it at the specified borders. The number of partitions is the number of borders plus one. For example, given two borders A and B, the partitions will be defined as follows: [-inf, A), [A,B), [B, +inf)

```
CREATE PARTITION FUNCTION RANGE_PARTITIONING_FUNC ( ts
TIMESTAMP )
    AS RANGE BOUNDARIES(
        TIMESTAMP '2007-08-01 01:10:30',
        TIMESTAMP '2007-08-01 01:15:30'
    );
CREATE PARTITION SCHEME RANGE_PARTITIONING USING
RANGE_PARTITIONING_FUNC;
CREATE TABLE series_key_tbl ( key_column BIGINT, ts TIMESTAMP,
j BIGINT, d double )
SERIES (
    SERIES KEY( key_column )
    EQUIDISTANT INCREMENT BY 1 MINUTE
    START TIMESTAMP '2007-08-01 01:00:30'
    END TIMESTAMP '2007-08-01 01:30:30'
    PERIOD FOR SERIES ( ts )
) PARTITION BY RANGE_PARTITIONING( ts );
```

<hash partitioning>

The data is partitioned by key such that all data for a key is included in the same partition. Data for two keys, `key1` and `key2`, will be contained in the same partition if `hash( key1 ) == hash( key2 )`.

```
CREATE PARTITION FUNCTION HASH_PARTITIONING_FUNC ( key_column
varchar(*) )
    AS HASH( key_column ) MIN PARTITIONS 3;
CREATE PARTITION SCHEME HASH_PARTITIONING USING
HASH_PARTITIONING_FUNC;
CREATE TABLE series_key_tbl ( key_column BIGINT, ts TIMESTAMP,
j BIGINT, d double )
SERIES (
    SERIES KEY( key_column )
```

```
    EQUIDISTANT INCREMENT BY 1 MINUTE
    START TIMESTAMP '2007-08-01 01:00:30'
    END TIMESTAMP '2007-08-01 01:30:30'
    PERIOD FOR SERIES ( ts )
) PARTITION BY HASH_PARTITIONING( key_column );
```

<interval partitioning>

The data is partitioned by the period column, by dividing the time series after the defined interval relative to the Unix epoch: 1970-01-01 00:00:00.0.

Interval partitioning requires tables of type STREAMING.

```
CREATE PARTITION FUNCTION INTERVAL_PARTITIONING_FUNC( ts
TIMESTAMP )
    AS INTERVAL( ts ) '15' MINUTE;
CREATE PARTITION SCHEME INTERVAL_PARTITIONING USING
INTERVAL_PARTITIONING_FUNC;
CREATE TABLE series_key_tbl ( key_column BIGINT, ts TIMESTAMP,
j BIGINT, d double )
SERIES (
    SERIES KEY( key_column )
    EQUIDISTANT INCREMENT BY 1 MINUTE
    START TIMESTAMP '2007-08-01 01:00:30'
    END TIMESTAMP '2007-08-01 01:30:30'
    PERIOD FOR SERIES ( ts )
) TYPE STREAMING PARTITION BY INTERVAL_PARTITIONING( ts );
```

<key-time partitioning>

The data is partitioned by key, as described in hash partitioning. In addition, data for a key is split further, as described in interval partitioning.

This results in a set of segments, where each segment refers to a single key and a single time range.

Key-time partitioning requires tables of type STREAMING, due to the fact that interval partitioning also requires them.

```
CREATE PARTITION FUNCTION
KEY_TIME_PARTITIONING_FUNC( key_column VARCHAR(*), ts
TIMESTAMP  )
    AS HASH( key_column ) MIN PARTITIONS 3
       INTERVAL( ts ) '15' MINUTE;
CREATE PARTITION SCHEME KEY_TIME_PARTITIONING USING
KEY_TIME_PARTITIONING_FUNC;
CREATE TABLE series_key_tbl ( key_column BIGINT, ts TIMESTAMP,
j BIGINT, d double )
SERIES (
    SERIES KEY( key_column )
    EQUIDISTANT INCREMENT BY 1 MINUTE
    START TIMESTAMP '2007-08-01 01:00:30'
    END TIMESTAMP '2007-08-01 01:30:30'
    PERIOD FOR SERIES ( ts )
) TYPE STREAMING PARTITION BY
KEY_TIME_PARTITIONING( key_column, ts );
```

<spark_engine_clause>

The Spark engine clause tells the SAP Vora Spark layer to execute the statement using the data source for the SAP Vora engines (graph, time series, document store, and disk):

```
<spark_engine_clause> ::= using com.sap.spark.engines
```

<spark_options_clause>

The Spark options allow you to define the source of the data import as well as further options, such as delimiters and so on.

```
<spark_options_clause> ::= OPTIONS '('
    <import_path>,
    <csv_option_list>,
    <storage_backend_info>
  )
  <import_path> ::= files '"'<path_string_literal>'"'
  <csv_option_list> ::= <csv_option> [, <csv_option>, ...]
  <csv_option> ::=
    <csv_skip>
  | <csv_delimiter>
  <csv_skip> ::= csvskip '"'<numeric literal>'"'
  <csv_delimiter> ::= csvdelimiter '"'<ansi_character>'"'
  <storage_backend_info> ::= storagebackend '"'<storage_backend>'"'
  <storage_backend> ::=
    local
  | hdfs
```

For the full list of <csv_options> as well as the supported storage backends, see the general Spark options and CSV options overview page: *Raw SQL Data Source Options*.

Related Information

Data Source Options [page 122]

5.3.1.2 Examples

Basic

This example creates a simple series with one timestamp and two value columns. The timestamp column is the chosen period.

Sample Code

```
CREATE TABLE name (
    ts TIMESTAMP,
    value1 INTEGER,
    value2 DOUBLE
) SERIES (
    PERIOD FOR SERIES ts
    START TIMESTAMP '2010-01-01 00:00:00'
    END TIMESTAMP '2010-12-31 00:00:00'
    EQUIDISTANT INCREMENT BY 1 HOUR
) using com.sap.spark.engines;
```

Compression Configuration

This example creates a series with a simple compression definition on all columns without ranges.

Sample Code

```
CREATE TABLE name (
    ts TIMESTAMP,
    value1 INTEGER,
    value2 DOUBLE
) SERIES(
    PERIOD FOR SERIES ts
    START TIMESTAMP '2010-01-01 00:00:00'
    END TIMESTAMP '2010-12-31 00:00:00'
    DEFAULT COMPRESSION USE (AUTO ERROR 1.2 PERCENT)
    COMPRESSION ON value1
    USE (APCA ERROR 1.2 PERCENT)
) using com.sap.spark.engines;
```

Compression Configuration with Column List

This example creates a series with a simple compression definition on all columns without ranges.

Sample Code

```
CREATE TABLE name (
    ts TIMESTAMP,
    value1 INTEGER,
    value2 DOUBLE
) SERIES(
    PERIOD FOR SERIES ts
    START TIMESTAMP '2010-01-01 00:00:00'
    END TIMESTAMP '2010-12-31 00:00:00'
    DEFAULT COMPRESSION USE (AUTO ERROR 1.2 PERCENT)
    COMPRESSION ON (value1..value2)
    USE (APCA ERROR 1.2 PERCENT)
) using com.sap.spark.engines;
```

Range Definition

This example creates a strictly equidistant series, including a range definition.

Sample Code

```
CREATE TABLE name (
    ts TIMESTAMP,
    value1 INTEGER,
    value2 DOUBLE )
SERIES (
    PERIOD FOR SERIES ts
    START TIMESTAMP '2010-01-01 00:00:00'
    END TIMESTAMP '2010-12-31 00:00:00'
    EQUIDISTANT INCREMENT BY 1 HOUR
) using com.sap.spark.engines;
```

Partitioning

This example partitions the series in the ranges given above.

Sample Code

```
CREATE PARTITION FUNCTION PF1( C TIMESTAMP )
AS RANGE BOUNDARIES(
    TIMESTAMP '2010-04-01 09:00:00.0000',
    TIMESTAMP '2010-09-01 09:00:00.0000'
);
CREATE PARTITION SCHEME PS1 USING PF1;
CREATE TABLE name (
    ts TIMESTAMP,
    value1 INTEGER,
    value2 DOUBLE
) SERIES (
    PERIOD FOR SERIES ts
    START TIMESTAMP '2010-01-01 00:00:00'
    END TIMESTAMP '2010-12-31 00:00:00'
    EQUIDISTANT INCREMENT BY 1 HOUR
) PARTITION BY PS1(ts)
using com.sap.spark.engines;
```

Load Table from HDFS

This example creates a table and loads the data from a CSV file (fully qualified file name). The CSV contains data that is delimited by a semicolon and where the first line will be skipped. The data is appended to the current table.

Sample Code

```
CREATE TABLE name (
```

```
    ts TIMESTAMP,
    value1 INTEGER,
    value2 DOUBLE
) SERIES (
    PERIOD FOR SERIES ts
    START TIMESTAMP '2010-01-01 00:00:00'
    END TIMESTAMP '2010-12-31 00:00:00'
    EQUIDISTANT INCREMENT BY 1 HOUR
) PARTITION BY PS1(ts)
using com.sap.spark.engines
OPTIONS (
    files "<path>",
    csvskip "1",
    csvdelimiter ";",
    storagebackend "hdfs"
)
```

Load Table from CSV Using Raw SQL

Reload the table `name` after a restart of the system using raw SQL.

Sample Code

```
``LOAD TABLE name`` using com.sap.spark.engines;
```

5.3.2 Data Query Statements

These statements are used to retrieve time series from the SAP Vora time series engine. They are also the basic components for aggregation and analysis invocation.

Syntax

```
<data query statement> ::=
    <select statement>
```

Note

Use the raw SELECT syntax for SELECT statements (this is not shown in the code examples).

Related Information

Raw SQL Syntax and Parsed DDL Syntax [page 58]

5.3.2.1 Simple SELECT

SELECT statement for retrieving values from a time series.

Syntax

```
<select statement> :==
    SELECT <column_expression_list>, ...
    FROM <series_table_reference>
    [<where_clause>]
    [<group_by>]
```

<column_expression_list>

Selects time series columns or columns from a table function result. In addition, you can add column functions for time series columns using a column function expression. Column and table function identifier and signature comply with the functions that are currently supported.

The column function_expression refers to calling standard aggregations, advanced aggregations, and column-based functions. Table functions are part of the FROM clause and are explained in the *Table Functions* section.

```
<column_expression_list> ::=
      <column_expression>, ...
    | <asterisk>
<column_expression> ::=
      <column_name>
    | '"' <column_name> '"'
    | <column_function_expression>
```

<series_table_reference>

The series table reference marks an execution on top of a series by introducing the reserved keyword SERIES.

```
<series_table_reference> ::= [{[SERIES] [TABLE]}]
<table_name> [[AS] <alias>]
| <table_function> [[AS] <alias>]
```

The following are alternatives to using SERIES as the reserved keyword:

- SELECT * FROM series TABLE <name>
- SELECT * FROM <name>
  When this option is used, the SAP Vora semantic analyzer automatically searches for the type within the internal catalog.

Table functions provided as part of the FROM clause are also supported. In the case of a table function, the SELECT clause is used to select columns from the table function result.

<where_clause>

The where clause allows you to filter by key and time. The result of the query is limited to the rows for which the predicate evaluates to TRUE.

```
<where_clause> ::= WHERE <series_predicate>
<series_predicate> ::=
      <period_predicate>
    | <period_range_predicate>
    | <series_key_predicate>
    | <series_predicate> AND <series_predicate>
    | <series_predicate> OR <series_predicate>
    | '(' <series_predicate> ')'
```

<period_predicate>

The projection is restricted to the specified time range.

```
<period_predicate> ::=
      <period_range_predicate>
    | <identifier> <comparison_operator> <literal>
    | <literal> <comparison_operator> <identifier>
    | <identifier> [NOT] IN '(' <literal>, ... ')'
    | <identifier> [NOT] BETWEEN <literal> AND <literal>
```

`<identifier>` must refer to a series period column.

`<literal>` must be a literal of the same type as the period column.

The BETWEEN clause `X BETWEEN A AND B` refers to `[A,B]`, which means that B will be included.

Please note that the BETWEEN clause cannot be arbitrarily combined with a conjuction. It might need to be surrounded with parentheses.

Sample Code

Simple select with a filter on period, filtering for January and March 2000 using the standard SQL BETWEEN and the time series engine-specific PERIOD BETWEEN. Two time intervals are queried: [`2000-01-01 00:00:00`;`2000-01-31 59:59:59.9999999`], [`2000-03-01 00:00:00`;`2000-04-01 00:00:00`)

```
SELECT * FROM series_tbl WHERE
       ts_column BETWEEN TIMESTAMP `2000-01-01 00:00:00` AND
`2000-01-31 59:59:59.9999999`
    OR PERIOD BETWEEN TIMESTAMP `2000-03-01 00:00:00` AND
`2000-04-01 00:00:00`;
```

Sample Code

Simple select with a filter on period, filtering for January and March 2000 using comparison operators

```
SELECT * FROM series_tbl WHERE
       ( ts_column >= TIMESTAMP `2000-01-01 00:00:00` AND
ts_column <= `2000-01-31 59:59:59.9999999` )
    OR ( ts_column >= TIMESTAMP `2000-03-01 00:00:00` AND
ts_column < `2000-04-01 00:00:00` );
```

Sample Code

Simple select with a filter on period, filtering for three specific dates

```
SELECT * FROM series_tbl WHERE ts_column IN (
    TIMESTAMP `2000-01-01 00:00:00`,
    TIMESTAMP `2000-02-25 00:00:00`,
    TIMESTAMP `2000-03-16 00:00:00`
);
```

<period_range_predicate>

The projection is restricted to the provided range. This clause is a special version of restricting ranges with a where clause.

```
<period_range_predicate> ::=
    PERIOD AS OF <time_literal>
  | BETWEEN <time_literal> AND <time_literal>
<time_literal> ::=
    TIMESTAMP '<timestamp_string>'
  | DATE '<date_string>'
  | TIME '<time_string>'
```

The BETWEEN clause `PERIOD BETWEEN A AND B` refers to `[A,B)`, which means that B will be excluded.

Please note that the BETWEEN clause cannot be arbitrarily combined with a conjuction. It might need to be surrounded with parentheses.

Sample Code

Example: Simple select with filter on period, filtering for March 2000 and all data from year 2001 and later

```
SELECT * FROM series_tbl WHERE
       PERIOD BETWEEN TIMESTAMP `2000-03-01 00:00:00` AND
`2000-04-01 00:00:00`
    OR PERIOD AS OF TIMESTAMP `2001-01-01 00:00:00`
```

Note

`<period_range_predicate>` cannot be used when the table is of type STREAMING.

Note

The period range clause is deprecated and could be removed in the future. Please use comparison or a standard SQL BETWEEN clause instead:

- `PERIOD AS OF <A>` is equal to `<period column> >= <A>`.
- `PERIOD BETWEEN <A> AND <B>` is equal to `<period column> >= <A> AND <period column> < <B>`.

<series_key_predicate>

The projection is restricted to the specified series key.

```
<series_key_predicate> ::=
      <identifier> <comparison_operator> <literal>
    | <literal> <comparison_operator> <identifier>
    | <identifier> [NOT] IN '(' <literal>, ... ')'
    | <identifier> [NOT] BETWEEN <literal> AND <literal>
```

`<identifier>` must refer to a series key column.

`<literal>` must be a literal of the same type as the series key column.

Please note that the BETWEEN clause cannot be arbitrarily combined with a conjuction. It might need to be surrounded with parentheses.

Sample Code

Example: Simple select with filter on the series key, filtering for sensors of a string type

```
SELECT * FROM series_tbl WHERE
          sensor_id = 'SensorA'
       OR sensor_id IN ( 'SensorB', 'SensorC' )
```

Sample Code

Simple select with a filter on the series key, filtering for sensors of an integer type

```
SELECT * FROM series_tbl WHERE (
           sensor_id = 1
        OR sensor_id IN (2, 3)
        OR ( sensor_id > 5 AND sensor_id < 10 )
        OR ( sensor_id BETWEEN 20 AND 29 )
        OR sensor_id > 200
    ) AND sensor_id <> 202
      AND sensor_id NOT IN ( 25, 27 )
      AND sensor_id NOT BETWEEN 220 and 229;
```

<group_by>

```
<group_by> ::= GROUP BY <series_key_column>, ...
```

`<series_key_column>, ...` must be the series key or a subset of the series key. No other group by clauses are supported.

If specified, aggregations are calculated per series key. The result will contain as many rows as the series table contains distinct series keys.

Sample Code

Simple select with a GROUP BY key clause

```
SELECT SUM(val1) FROM SERIES name
```

```
GROUP BY key;
```

5.3.2.2 Examples

Basic

Simple select on three columns of a time series: PERIOD, value1, value2.

Sample Code

```
SELECT timestamp, value1, value2 FROM SERIES <table_name>
```

Range Limit

Selects data from the years 2000 and 2001.

Sample Code

```
SELECT "Period", "value1", "value2" FROM SERIES <table_name> WHERE PERIOD
BETWEEN TIMESTAMP '2000-01-01 00:00:00' AND TIMESTAMP '2001-12-31 00:00:00'
```

Selects all data starting from the 1st January 2002.

Sample Code

```
SELECT timestamp, value1, value2 FROM SERIES TABLE <table_name> WHERE PERIOD
AS OF DATE '2002-01-01';
```

5.3.3 Aggregations and Column Functions

A function expression can be a standard aggregation, advanced aggregation, or a column function.

```
<function expression> ::= <std aggregation> '(' <column> ')'
| <column function> '(' <column> ')'
<column> ::= <column_identifier> | <asterisk>
<column_identifier> ::= {<column_name>|'"'<column_name>'"'}
<column function> ::=
| <trend function>
| <mode function>
```

```
| <median function>
```

5.3.3.1 Standard Aggregation

Calculates a standard aggregation for a value column of a time series.

```
<std aggregation> ::= SUM | AVG | MIN | MAX | COUNT
```

Sample Code

```
SELECT SUM(val1) FROM SERIES name WHERE PERIOD AS OF timestamp '2015-09-01
00:00:00'
```

5.3.3.2 TREND_SLOPE

TREND_SLOPE derives the slope of a linear regression function.

In general, a linear regression can be described as follows: `y = a*x + b`, where `y` is a measured variable sorted by time and represented by the column identifier, `x` is the order of values in `y`, `a` is a slope, and `b` is an intercept.

```
<trend_slope function> ::= TREND_SLOPE '(' <column_identifier> ')'
```

Sample Code

TREND_SLOPE aggregation including a range filter

```
SELECT TREND_SLOPE(val1)
FROM SERIES name
WHERE PERIOD BETWEEN timestamps '2015-09-01 00:00:00'
                 AND timestamp '2015-10-01 00:00:00';
```

5.3.3.3 MEDIAN

Calculates the median for a value column of a time series.

```
<median function> ::= MEDIAN '(' <column_identifier> ')'
```

Sample Code

Simple MEDIAN with a range filter

```
SELECT MEDIAN(val1)
```

```
FROM SERIES name
WHERE PERIOD BETWEEN timestamps '2015-09-01 00:00:00'
AND timestamp '2015-10-01 00:00:00'
```

5.3.3.4 MODE

Calculates the modal value for a value column of a time series.

```
<median function> ::= MODE '(' <column_identifier> ')'
```

Sample Code

Simple MODE with a range filter

```
SELECT MODE(val1)
FROM SERIES name
WHERE PERIOD BETWEEN timestamps '2015-09-01 00:00:00'
AND timestamp '2015-10-01 00:00:00'
```

5.3.4 Table Functions

Table functions allow you to calculate autocorrelation and cross-correlation coefficients, create histograms, and increase or decrease the granularity of a time series.

5.3.4.1 AUTO CORRELATION

Calculates the autocorrelation coefficient for a given column and a given lag.

Note

This feature is deprecated. AUTO_CORR and CROSS_CORR have been replaced by CORRELATION and will be removed in a future version.

The provided maxTimeLag defines the maximum lag. A correlation value is calculated for all lags up to maxTimeLag. The ORDER BY clause is optional. If it is not provided, the period column of the time series is used for ordering. When explicitly selecting a column, only time stamp columns can be selected for ORDER BY.

```
<auto_corr_table_function> ::= AUTO_CORR '('
    <series_table_reference>
    [ORDER BY <ts_column>],
    <maxTimeLag>,
    DESCRIPTOR(<column_identifier>)
')'
```

Sample Code

```
SELECT * FROM AUTO_CORR ( SERIES name, 10, DESCRIPTOR(val1) ) AC;
```

5.3.4.2 CROSS CORRELATION

Calculates the correlation between two columns.

Note

This feature is deprecated. AUTO_CORR and CROSS_CORR have been replaced by CORRELATION and will be removed in a future version.

The provided starting time stamps indicate the start of the considered sample for each of the provided columns. The first time stamp corresponds to the first column, while the second corresponds to the second column. The <numRows> clause defines the number of elements within the samples. Both columns are required to have the same number of rows. If you do not define any time stamps or number of rows, the entire column is considered.

```
<cross_corr_table_function> ::= CROSS_CORR '('
    <series_table_reference> [ORDER BY <ts_column>],
    <maxTimeLag>,
    [<numRows>],
    DESCRIPTOR(<column_identifier>, <column_identifier>)
')'
<starting_time> ::= <timestamp>
```

Sample Code

```
SELECT * FROM CROSS_CORR ( SERIES name, 10,
DESCRIPTOR(val1, val2) ) CC;
SELECT * FROM CROSS_CORR ( SERIES name, 10, 31, DESCRIPTOR(val1, val2) ) CC;
```

5.3.4.3 CORRELATION

The correlation table function calculates correlation coefficients of a given type for columns specified in the correlation descriptors. It provides a unified way of using correlation functions.

```
<correlation_table_function> ::= CORRELATION '('
  <series_table_reference>
  [PARTITION BY '(' <series_key_column>, ... ')' ]
  [ORDER BY '(' <ts_column>, ... ')' ],
  <correlation_parameters>,
  <correlation_descriptors>
')'
<correlation_parameters> ::=
    <correlation_type>
    ',' <correlation_matrix_type>
```

```
    [',' <maxTimeLag>]
    [',' <numRows>]
<correlation_type> ::= SIGNAL
<correlation_matrix_type> ::= AUTO_CORR | CROSS_CORR
<correlation_descriptors> ::=
    COLUMN => DESCRIPTOR( <identifier>, ... )
```

In general, it returns a correlation matrix consisting of the correlation coefficients for all combinations of defined columns and keys. The results are returned in a coordinate format: keys from PARTITION BY if specified, lag, x-coordinates of the keys' types, y-coordinates of the keys' types, matrix coordinates as column names, and a correlation coefficient as a double. The ORDER BY is optional. If it is not provided, the period column of the time series is used for ordering. When explicitly selecting a column, only timestamp columns can be selected for ORDER BY. The PARTITION BY clause is mandatory for tables with defined series keys. Only series keys can be defined in this clause.

Note

To ensure compatibility with the SAP HANA syntax, ORDER BY without brackets is also supported.

The correlation parameters define the type and settings of a correlation function:

- The specified `maxTimeLag` defines the maximum lag. When this optional setting is specified, an extra column CORR_LAG is added to the output, which contains the lags for `-lag` to `+lag`.
  A correlation value is calculated for all lags up to `maxTimeLag`.
- `numRows` defines how many column rows are considered. If fewer rows are available, an exception is thrown.
- `correlation_type` defines which correlation function is used. Currently, only SIGNAL is supported.
- `correlation_matrix_type` describes which correlation type is calculated:
  - AUTO_CORR computes autocorrelation for a specific column.
  - CROSS_CORR derives a correlation coefficient between two columns.

The correlation descriptors define the input of a table function. The column clause describes which columns should be used from the time series. The key columns define the matrix coordinates.

Sample Code

Simple autocorrelation with a lag of 5

```
SELECT * FROM CORRELATION ( SERIES series_tbl,
    SIGNAL, AUTO_CORR, 5, COLUMN => DESCRIPTOR( j ) );
```

Sample Code

Simple cross correlation

```
SELECT * FROM CORRELATION ( SERIES series_tbl,
    SIGNAL, CROSS_CORR, COLUMN => DESCRIPTOR( col1, col2 ) );
```

5.3.4.4 HISTOGRAM

Creates a histogram of the values for the selected column. You need to provide the number of bins that the histogram should create. A default value can be configured in the system preferences.

```
<histogram_table_function> ::= HISTOGRAM '('
    SERIES [TABLE] <series_name>
    [PARTITION BY '(' <series_key_column>, ... ')' ]
    [ORDER BY '(' <ts_column> ')' ],
    <numberOfBins>,
    DESCRIPTOR(<column_identifier>)
')'
```

Sample Code

```
SELECT * FROM HISTOGRAM ( SERIES name, 5,
DESCRIPTOR(<column_identifier>) ) HIST;
```

5.3.4.5 GRANULIZE

Returns a new series based on an existing series by changing the interval between adjacent time stamps.

You can increase or decrease the time granularity. The new series contains the columns selected in the SELECT clause. The columns in the SELECT clause have to match the columns provided as function parameters. An asterisk in the SELECT clause is supported and all columns provided as function parameters are returned.

```
<granulize_table_function> ::= GRANULIZE '('
    SERIES [TABLE] <series_name>
    [PARTITION BY '(' <series_key_column>, ... ')' ]
    [ORDER BY '(' <ts_column> ')' ],
    <interval const> <offset>,
    <rounding_mode>,
    <granulize_column_list>
    ')'
<granulize_column_list> ::=
    <granulize_column> [, <granulize_column>, ...],
<granulize_column> ::=
    <granulize_mode> => DESCRIPTOR(<column_list>)
<granulize mode> ::= EVEN | SAME | AVG | SUM
<column_list> ::= <column_identifier> [, ...]
<offset> ::= OFFSET <interval const>
<rounding_mode> ::=
      ROUND_HALF_UP
    | ROUND_HALF_DOWN
    | ROUND_HALF_EVEN
    | ROUND_DOWN
    | ROUND_UP
```

The provided interval is the target interval. It is provided by an <interval const> clause, which combines a numeric value and a unit. The units currently supported are YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and MILLISECOND. The granulize function always disaggregates or rounds to a full interval. Deviations can be provided by adding an offset that needs to be smaller than the target interval.

The granulize mode determines how the values of the selected value columns are rounded up (decreasing the granularity) or disaggregated (increasing the granularity). The following modes are supported:

Granulize Mode	Description
EVEN	Disaggregation operation. Values are evenly distributed to the target time intervals. Counter mode to SUM rounding. If more values are distributed to fewer values, some values are lost.
SAME	Disaggregation operation. The same value is provided to all instances of the target interval. Counter mode to AVG rounding. If more values are distributed to fewer values, some values are lost.
AVG	Rounding operation. Assigns the average of all connected values to the instances of the new interval. Counter mode to SAME. If fewer values are aggregated to more values, NULLs are inserted.
SUM	Rounding operation. Uses the SUM of all connected values to compute the values for the instances of the new interval. Counter mode to EVEN. If fewer values are aggregated to more values, NULLs are inserted.

The rounding mode defines how values that fall in between the target interval are assigned to the instances of the new interval. The supported rounding modes are:

Rounding Mode	Description
ROUND_HALF_UP	Rounds away from zero to the nearest interval instance. Halfway values are rounded up to the greater interval instance.
ROUND_HALF_DOWN	Rounds away from zero to the nearest interval instance. Halfway values are rounded down to the smaller interval instance.
ROUND_HALF_EVEN	Rounds away from zero to the nearest interval instance. Halfway values are rounded to the instances that comply with an even index.
ROUND_UP	Always rounds away from zero to the greater interval instance.
ROUND_DOWN	Always rounds away from zero to the smaller interval instance.
ROUND_CEILING	Always rounds in a positive direction to the greater interval instance. For time stamps this is reflected with ROUND_UP, because no negative time stamps exist.
ROUND_FLOOR	Always rounds in a negative direction to the smaller interval instance. For time stamps this is reflected with ROUND_DOWN, because no negative time stamps exist.

Sample Code

```
SELECT * FROM GRANULIZE
( SERIES name, 1 MONTH, SUM => DESCRIPTOR(val1) ) AS G;
SELECT * FROM GRANULIZE
( SERIES name, 15 MINUTE, EVEN => DESCRIPTOR(val1), SAME =>
DESCRIPTOR(val2,val3) ) AS G;
SELECT * FROM GRANULIZE
( SERIES name, 1 HOUR OFFSET 15 MINUTES, SUM => DESCRIPTOR(val1) ) AS G;
SELECT * FROM GRANULIZE
( SERIES name, 1 MONTH, ROUND_UP, AVG => DESCRIPTOR(val1,val2) ) G;
```

5.4 Working with Collections (Document Store)

The SAP Vora document store is a distributed in-memory JSON document store that supports rich query processing over JSON data.

The document store uses a special binary JSON format and a highly optimized, parallel, and NUMA-aware execution engine to provide high performance on analytical workloads. It also combines JSON with most of the regular SQL features, allowing you to use SQL for JSON as well.

5.4.1 Documents and Collections

The document store provides flexible storage for documents using the standard JSON format. Documents are stored in collections, which are schema-less.

Documents

A document is a JSON object. JSON documents can consist of strings, integers, arrays, objects, and any other data types supported by JSON. The following example shows a document that consists of different JSON types:

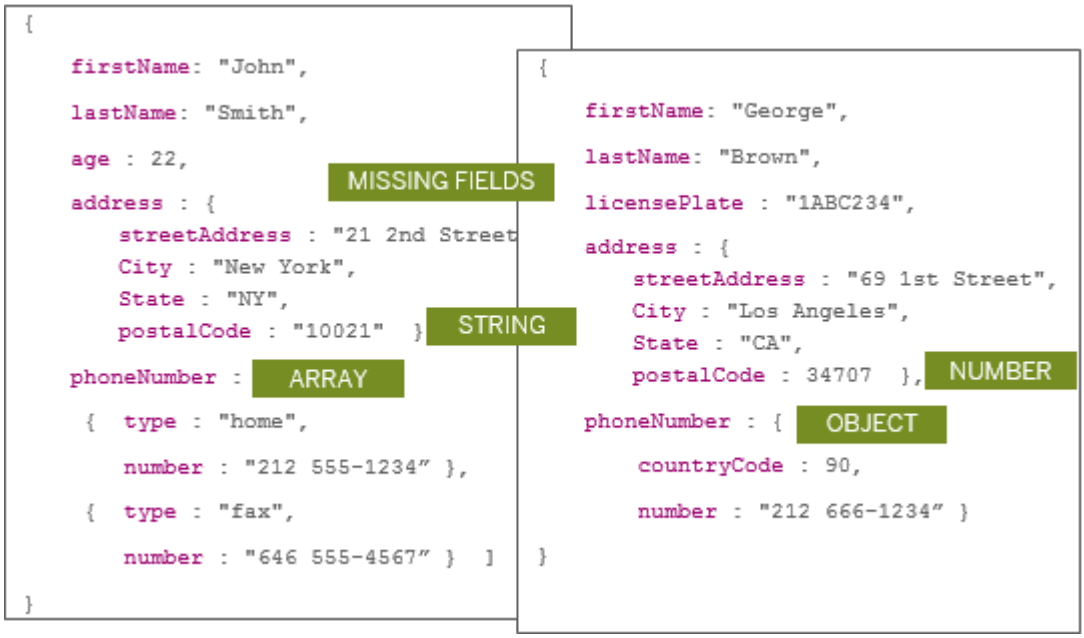
```
{
    firstName: "John",                                  STRING
    lastName: "Smith",
    age : 22,                                           NUMBER
    address : {                                         OBJECT
        streetAddress : "21 2nd Street",
        City : "New York",
        State : "NY",
        postalCode : "10021"  },
    phoneNumber : [                                     ARRAY
              {  type : "home", number : "212 555-1234" },
              {  type : "fax", number : "646 555-4567" }  ],
    married : false,                                    BOOLEAN
    spouse : null                                       NULL
}
```

Collections

A JSON document is stored in a collection. Since collections are schema-less, the documents they contain are not required to have identical fields, although their structures are generally similar. Therefore:

- Fields present in one document can be missing in another
- Different documents within a collection can have identically named fields, but with different data types

For example, the two documents below are contained in the same collection but have different fields as well as identically named fields with different data types:



5.4.2 Creating Collections

You can create and load collections in SAP Vora using the CREATE COLLECTION statement. You can create a partitioned collection by adding the PARTITION BY clause.

You can create a collection using the following syntax:

```
CREATE COLLECTION <collection_identifier>
  [PARTITION BY <partition_scheme_identifier>]
  USING com.sap.spark.engines
  OPTIONS (<list_of_options>);
```

- The PARTITION BY clause is optional.
- The following options are supported:
  - `files '/path/to/file/to/load/some.json'`
  - `storagebackend "hdfs"`

Creating Partitioned Collections

To create a partitioned collection, you first need to define a partition function and partition scheme.

You can create a partition function and scheme as follows. Note that the document store currently only supports HASH partitioning:

```
CREATE PARTITION FUNCTION <partition_function_identifier>(<list_of_fields>)
    AS HASH(<list_of_fields>) MIN PARTITIONS <num> MAX PARTITIONS <num>
    USING com.sap.spark.engines;
CREATE PARTITION SCHEME <partition_scheme_identifier>
    USING <partition_function_identifier>
    USING com.sap.spark.engines;
```

Sample Code

```
CREATE PARTITION FUNCTION PF("_id") AS HASH("_id")
    MIN PARTITIONS 3 MAX PARTITIONS 3
    USING com.sap.spark.engines;
CREATE PARTITION SCHEME PS
    USING PF
    USING com.sap.spark.engines;
```

Then add a reference to the partition scheme in the CREATE COLLECTION statement, as shown in the example below:

Sample Code

```
CREATE COLLECTION T
  PARTITION BY PS("state")
  USING com.sap.spark.engines
  OPTIONS (files 'path/to/file/to/load/some.json');
```

Loading Data

You can load data from files stored in HDFS. The supported file format is JSON. To load data from HDFS, you also need to specify the `storagebackend` option:

Sample Code

```
CREATE COLLECTION T
  PARTITION BY PS2("state")
  USING com.sap.spark.engines
  OPTIONS (
    files 'path/to/file/to/load/some.json'
    storagebackend "hdfs"
  );
```

The following restrictions apply when loading files:

- Only `.json` files are supported
- Newline should be used as a separator between documents (therefore `\n` cannot be used inside a document)
- JSON keys are restricted as follows:
  - They are not allowed to start with '$' (for example, `{"$year" : "1986"}` is invalid)

- They are not allowed to contain '[' or ']' (for example, `{"key[0]": 1 }` is invalid)

5.4.3 Dot Notation

Since JSON documents can be nested, dot notation is used to access nested elements.

To access `street` in the following document, for example, you would use `author.address.street` as an identifier for the values:

Sample Code

```
{
    "author": {
        "address": {
            "street": "..."
        },
        ...
    },
    ...
}
```

To access elements in an array, you use the index operator as known from other programming languages. So to access the street of the second address, you would use `author.addresses[2].street`:

Sample Code

```
{
    "author": {
        "addresses": [{
            "street": "First Street"
        },{
            "street": "Second Street"
        }]
    },
    ...
}
```

The index positions start at 1, so the first element has position 1. This notation is used wherever you can access a value in this manner, such as in sort, filter, and projection. If there is no value in the document for a given identifier, a NULL value is returned.

5.4.4 SELECT Clauses

A simple query consists of three parts:

- SELECT: Parts of the document to return
- FROM: The data bucket or data store to work with
- WHERE: Conditions the document must satisfy

Note

Use the raw SELECT syntax for SELECT statements (this is not shown in the code examples).

Sample Code

```
SELECT * FROM <collection> WHERE fname = 'Ian';
```

A query only requires a SELECT clause. The wildcard * selects all parts of the document. Queries can return a collection of different document structures or fragments. However, they will all match the conditions in the WHERE clause.

```
SELECT <collection> FROM <collection>
```

The SELECT clause behaves like SELECT * if the projection field is the same as the collection name.

Projection

You can apply projections to selection queries using the following syntax:

```
select { <resulting_field> : <queried_field> } from <collection>;
```

For example, you have the following two documents:

Sample Code

```
{ _id: 1, name: "joe", int1: 3, int2: 2, float: 6.0, obj: { int3: 10 } }
{ _id: 2, name: "jane", int1: 4, int2: 5, float: 7.0, obj: { int3: 11 } }
```

Compare the following queries and results:

Example

Query:

```
select { _id: _id, res: int1 + int2 } from a order by _id asc;
```

Result:

```
{ _id: 1, res: 5 }
{ _id: 2, res: 9 }
```

Example

Query:

```
select { a: { b: int1, c: { d : int2 } } } from a;
```

Result:

```
{ a: { b: 3, c: { d : 2 } } }
{ a: { b: 4, c: { d : 5 } } }
```

Related Information

Raw SQL Syntax and Parsed DDL Syntax [page 58]

5.4.4.1 Filters

You can use filters in the same way as in standard SQL. In addition, you can use the dot notation to filter on nested values.

The following query, for example, only returns documents that contain the field `author.address.street` with the value 'First Street':

Sample Code

```
SELECT * FROM <collection> WHERE author.address.street = 'First Street'
AND ...;
```

FILTER: IS MISSING

Since there is no schema in the document store, you can have different fields in different documents. If you want to get all documents which don't have certain fields set, you can use IS MISSING.

The following query, for example, returns every document for which `author` has no `address` assigned:

Sample Code

```
SELECT * FROM <collection> WHERE author.address IS MISSING;
```

FILTER: IS NULL

`IS NULL` behaves as in standard SQL. In addition, `IS NULL` returns true for missing fields.

FILTER: Array Elements

You can filter by checking whether the value of the array in an index equals a specific value.

For example, you have the following three documents:

Example

For example, you have the following three documents:

```
{ name:"john", arr: [10, 20] }
{ name:"mike", arr: [10, 20, 40] }
{ name:"eva", arr: [10, 20, 30, 40, 50] }
```

You execute the following query:

```
select * from collection where arr[3]=40;
```

The result is as follows:

```
{ name:"mike", arr: [10, 20, 40] }
```

Note that the first document is also not included in the result since the requested index is out of the array's bounds.

5.4.4.2 Limit Result

You can limit the result using the same syntax as in SQL:

```
SELECT * FROM <collection> FETCH FIRST <num> ROWS ONLY;
```

In addition, the LIMIT syntax is supported:

```
SELECT * FROM <collection> LIMIT <num>;
```

5.4.4.3 Skip Rows

To ignore the first <num> rows, you can use the `OFFSET` syntax:

```
SELECT * FROM <collection> OFFSET <num>;
```

This also works together with `LIMIT`:

```
SELECT * FROM <collection> LIMIT <limit> OFFSET <offset>;
```

5.4.4.4 Order By

You use the same syntax as in SQL:

```
SELECT * FROM <collection> ORDER BY <field> [ASC | DESC];
```

`<field>` can be any valid identifier in dot notation.

5.4.4.5 JSON Unnest

The JSON_UNNEST function returns a new collection containing a new document for each member of the specified array fields. The output documents contain the members of the arrays. The syntax is similar to SQL unnest but a collection is required.

```
SELECT * FROM JSON_UNNEST(<collection>, <field>...)
[PRESERVE_NULLS_AND_EMPTY_ARRAYS];
```

Based on the collection below with three documents, a simple example is as follows:

Sample Code

```
    {"A": [1, 2, 3], "B": [10, 11]}
    {"A": [], "B": [12]}
    {"A": null, "B": [13]}
```

Sample Code

```
SELECT * FROM JSON_UNNEST(<collection>, A);
```

Sample Code

```
    {"A": 1, "B": [10, 11]}
    {"A": 2, "B": [10, 11]}
    {"A": 3, "B": [10, 11]}
```

By default, no documents are produced for inputs where a specified field evaluates to null or an empty array. However, you can apply the option PRESERVE_NULLS_AND_EMPTY_ARRAYS to preserve those documents, as shown below:

Sample Code

```
SELECT * FROM JSON_UNNEST(<collection>, A) PRESERVE_NULLS_AND_EMPTY_ARRAYS;
```

Sample Code

```
    {"A": 1, "B": [10, 11]}
```

```
{"A": 2, "B": [10, 11]}
{"A": 3, "B": [10, 11]}
{"B": [12]}
{"A": null, "B": [13]}
```

In case of expansion with multiple fields, the number of documents will match the field that produces the largest number of documents and the missing ones will be set to null:

Sample Code

```
SELECT * FROM JSON_UNNEST(<collection>, A, B);
```

Sample Code

```
{"A": 1, "B": 10}
{"A": 2, "B": 11}
{"A": 3, "B": null}
{"B": 12}
{"A": null, "B": 13}
```

Experimental: You can also use json_unnest in a nested manner:

```
SELECT * FROM JSON_UNNEST((SELECT * FROM JSON_UNNEST(<collection>, A))
<an_alias_for_derived_table>, B);
```

Sample Code

```
{"A": 1, "B": 10}
{"A": 1, "B": 11}
{"A": 2, "B": 10}
{"A": 2, "B": 11}
{"A": 3, "B": 10}
{"A": 3, "B": 11}
```

5.4.5 Expressions

cardinality(<array>)

Returns the size of an array field.

Sample Code

```
SELECT { field: cardinality([10, 15, 20]) } FROM <collection>;
```

is_<type>(<expr>)

Returns true if the `<expr>` is of type `<type>`.

Sample Code

```
SELECT { field: is_boolean(true) } FROM <collection>;
```

to_bigint(<string>)

Converts the `<string>` to integer.

Sample Code

```
SELECT { field: to_bigint('15034112311') } FROM <collection>;
```

to_varchar(<expr>, [<format>])

Converts the `<expr>` in number, date, time, or timestamp types to string. The optional `<format>` parameter is used when the first parameter is a datetime type

Sample Code

```
SELECT { field: to_varchar(dt, 'YYYY/MM/DD') } FROM <collection>;
SELECT { field: to_varchar(15.4) } FROM <collection>;
```

to_double(<string>)

Converts the `<string>` to double.

Sample Code

```
SELECT { field: to_double('130.9531') } FROM <collection>;
```

concat(<str1>, <str2>)

Concatenates two strings and returns the result.

Sample Code

```
SELECT { field: concat(field_as_string, field2_as_string) } FROM <collection>;
```

length(<str>)

Returns the length of a string. Note that Unicode strings are not yet supported.

Sample Code

```
SELECT { field: length(field_as_string) } FROM <collection>;
```

replace(<str>, <find>, <replace>)

Replaces all occurrences of `<find>` with `<replace>` in `<str>`.

Sample Code

```
SELECT { field: replace(field_as_string, 'aaa', 'bbb') } FROM <collection>;
```

upper(<str>)

Converts a string to uppercase. The string can be either the result of an expression or a constant.

```
SELECT { field: UPPER(field_as_string), constant: UPPER('sTrinG_liTeRal') } FROM
<collection>;
```

lower(<str>)

Converts a string to lowercase. The string can be either the result of an expression or a constant.

```
SELECT { field: LOWER(field_as_string), constant: LOWER('sTrinG_liTeRal') } FROM
<collection>;
```

In Predicate

Returns true if the <expr> is found in a set of expressions.

Sample Code

The following query selects documents that have a name field that is 'joe' or 'jane':

```
SELECT {name: name} FROM col WHERE name IN ('joe', 'jane');
```

5.4.6 Mathematical Expressions

The following mathematical expressions are supported.

Supported Mathematical Expressions
abs
acos
asin
atan
atan2
cos
cosh
cot
ln
log
mod
power
round
sin
sinh
tan
tanh

5.4.7 Aggregation

Aggregate functions are used to calculate a single result per group of documents.

Group By

The GROUP BY clause groups the elements according to specified field names.

```
SELECT { _id: _id, aggr: <aggregatefunc> } FROM <collection> GROUP BY <field>;
```

The `_id` field of the new documents is set as the field in the GROUP BY clause. Multiple fields can be specified in a GROUP BY clause. In this case, documents are grouped by the combination of field names.

```
SELECT { _id: _id, aggr: <aggregatefunc> } FROM <collection> GROUP BY <field1>,
<field2>;
```

The `_id` field of the new documents is an object of grouped field names: `( _id: {field1:value, field2:value})`

If the GROUP BY clause is not specified, aggregate functions are calculated on all documents and one single result is returned. The `_id` of this element is set to null.

In case of duplicate fields in the GROUP BY clause, only the first one is effective.

In case of overlapping fields in the GROUP BY clause, the outermost field is effective.

Sum

Returns the sum.

```
SELECT { _id: _id, sum: sum(<field>) } FROM <collection> GROUP BY <field>;
```

Avg

Returns the average value.

```
SELECT { _id: _id, avg: avg(<field>) } FROM <collection> GROUP BY <field>;
```

Count

COUNT(<field>)

Returns the number of values of the specified field. Null values are not counted.

```
SELECT { _id: _id, count: count(<field>) } FROM <collection> GROUP BY <field>;
```

COUNT(*)

Returns the number of returned documents. Note that null values are also counted.

```
SELECT { _id: _id, countall: count(*) } FROM <collection> GROUP BY <field>;
```

Max

Returns the largest value.

```
SELECT { _id: _id, max: max(<field>) } FROM <collection> GROUP BY <field>;
```

Min

Returns the smallest value.

```
SELECT { _id: _id, min: min(<field>) } FROM <collection> GROUP BY <field>;
```

> **Note**
>
> Sum, Avg, Count, Max, and Min ignore null values. Count(*) does not.

5.4.8 Time-Related Features

There are four types of objects: Time, Timestamp, Date, and Interval.

The following three documents are used in the examples below:

> **Sample Code**

```
{ _id: 1, g: timestamp '2005-10-10 10:10:10', i: 1, f: 3.0, dt: date
'2010-10-10', t: time '10:10:10', null: null, str: "hi", h: timestamp
'2006-02-02 10:10:10'}
{ _id: 2, g: timestamp '2006-10-10 10:10:10', i: 1, f: 3.0, dt: date
'2010-10-07', t: time '09:10:10', null: null, str: "hi", h: timestamp
'2006-02-02 10:10:10'}
```

```
{ _id: 3, g: timestamp '2007-10-10 10:10:10', i: 1, f: 3.0, dt: date
'2011-10-07', t: time '11:10:10', null: null, str: "hi", h: timestamp
'2006-02-02 10:10:10'}
```

Date

This type holds a string of year to month in the following format: YYYY-MM-DD

Sample Code

```
select * from a where dt <= date '2010-10-09';
```

Time

This type holds a string of hour to second in the following format: hh:mm:ss

Sample Code

```
select * from a where t <= time '10:10:09';
```

Timestamp

This type holds a string of year to second in the following format: YYYY-MM-DD hh:mm:ss

Sample Code

```
select * from a where g < timestamp '2006-02-02 10:10:10' - interval '2' month;
```

Interval

This type is used for addition and subtraction operations covering all time-related types. The following options are available:

- year

Sample Code

```
select * from a where date '2017-02-28' = date '2016-02-29' + interval '1'
year;
```

- month

Sample Code

```
select * from a where date '2016-02-29' = date '2016-03-30' - interval '1'
month;
```

- day

Sample Code

```
select * from a where date '2016-02-29' = date '2016-03-01' - interval '1'
day;
```

- year to month

Sample Code

```
select * from a where timestamp '2004-9-10 10:10:10' = g - interval '1-1'
year to month;
```

- day to second

Sample Code

```
select * from a where timestamp '2005-10-12 00:10:10' = g + interval '1
14:00:00' day to second;
```

- hour to second

Sample Code

```
select * from a where time '00:10:10' = t + interval '14:00:00' hour to
second;
```

Related Information

Time-Related Expressions [page 119]

5.4.8.1 Time-Related Expressions

add_days(<dt>, <d>)

Adds <d> days onto date <dt> and returns the new date.

Sample Code

```
SELECT { field: add_days(date '2010-10-10', 10) } FROM <collection>;
```

add_months(<dt>, <m>)

Adds <m> months onto date <dt> and returns the new date.

Sample Code

```
SELECT { field: add_months(date '2010-10-10', 5) } FROM <collection>;
```

add_years(<dt>, <y>)

Adds <y> years onto date <dt> and returns the new date.

Sample Code

```
SELECT { field: add_years(date '2010-10-10', 2) } FROM <collection>;
```

dayname(<dt>)

Returns the name of the day on the date <dt>.

Sample Code

```
SELECT { field: dayname(date '2002-05-05') } FROM <collection>;
```

days_between(<d1>, <d2>)

Returns the number of days between the dates <d1>, <d2> (d2 - d1).

Sample Code

```
SELECT { field: days_between(date '2002-05-05', date '2001-05-05') } FROM
<collection>;
```

seconds_between(<d1>, <d2>)

Returns the number of seconds between the dates <d1>, <d2> (d2 - d1).

Sample Code

```
SELECT { field: seconds_between(date '2002-05-05', date '2001-05-05') } FROM
<collection>;
```

datediff_<unit>(<dt1>, <dt2>)

Returns the number of <unit>s between two datetime types <dt2> and <dt1>. The <dt1> and <dt2> types have to be the same. Supported units: nanosecond, microsecond, millisecond, second, minute, hour, day.

Sample Code

```
SELECT { field: datediff_nanosecond(date '2002-05-05', date '2005-01-01') }
FROM <collection>;
SELECT { field: datediff_day(timestamp '2002-05-05 15:02:10', date '2005-01-01
16:06:01') } FROM <collection>;
SELECT { field: datediff_millisecond(time '11:02:30', time '16:20:10') } FROM
<collection>;
```

weekday(<d>)

Returns the day number of the week on date <d>. The return value is 0-6 starting from Monday.

Sample Code

```
SELECT { field: weekday(date '2002-05-05') } FROM <collection>;
```

lastday(<d>)

Returns the last day of the month on date <d>.

Sample Code

```
SELECT { field: lastday(date '2005-01-01') } FROM <collection>;
```

to_date(<string>, <format>)

Converts the <string> to a date type according to <format>.

Sample Code

```
SELECT { field: to_date('20150505', 'YYYYMMDD') } FROM <collection>;
SELECT { field: to_date('2015/05/05', 'YYYY/MM/DD') } FROM <collection>;
```

to_time(<string>, <format>)

Converts the <string> to a time type according to <format>.

Sample Code

```
SELECT { field: to_time('20:15:10', 'HH24:MI:SS') } FROM <collection>;
```

to_timestamp(<string>, <format>)

Converts the <string> to a timestamp type according to <format>.

Sample Code

```
SELECT { field: to_timestamp('20150505 20:15:10', 'YYYYMMDD HH24:MI:SS') }
FROM <collection>;
```

5.5 Data Source Options

The `com.sap.spark.engines` data source has the following configurable options. Note that most of the options are taken into account only in the parsed DDL syntax case

Name	Description	Default Value[1]	Example Values
files	Comma-separated list of the fully qualified names of the files to be uploaded to SAP Vora.To specify file-specific options (such as `storagebackend`, `format`, and so on) please use pipe-separated notation as explained in *Loading Data into Tables*.	-	path/to/file1.csv,path/to/file2.csv
storagebackend	Backend where the data to be loaded is stored. This might require additional parameters to be set depending on the storage type. Possible values are "hdfs", "local", or "s3".	hdfs	hdfs, local, s3
format[2]	Format used to read the data. SAP Vora supports "csv", "orc", and "parquet".	csv	csv,orc,parquet
dateformat[2]	Specification of global and per-column date formats. The first value without a column name is the global one.	-	• col1: 'YYYY_MM_DD', col2: 'DD:MM:YYYY' • 'MM/DD/YYYY' • 'MM/DD/YYYY', col1: 'YYYY_MM_DD', col2: 'DD:MM:YYYY'
null[2]	Specification of global and per-column null values to parse NULL fields in CSV files. The first value without a column name is the global one. All values have to be enclosed in single quotes.	Absent values in varchar/char columns and absent values or "" in columns of other types are interpreted as NULL.	• '', col1: 'null', col2: 'null' • 'null', col1: 'x' • '' (default) • col1: 'null', col2: '?'
csvdelimiter [2]	Delimiter used to parse CSV files	,	;
csvquote[2]	Quote character in CSV files	"	,
csvskip[2]	Skips the first n lines in CSV files. The number must be non-negative.	0	10
csvthousandsdelimiter[2]	Delimiter for thousands	,	:
csvdecimaldelimiter[2]	Delimiter for decimals	.	@
csvescape[2]	Character used for escaping strings in the CSV file	\	#
csvnoquote[2]	True if no quotes should be used when loading csv files, false otherwise. If it is set to true, `csvquote` is ignored.	false	true

Name	Description	Default Value[1]	Example Values
columns[2]	Comma-separated list of column indexes to be loaded to SAP Vora. For example, "4,1,5" means that the 4th, 1st, and 5th columns of the input file are to be loaded. The number of columns in the schema definition must match the number of column indexes provided by this option.	-	4,1,5
tabletype[2]	Type of the underlying table	datasource	streaming
s3accesskeyid	Amazon S3 access key	-	somes3keyid
s3secretaccesskey	Amazon S3 secret access key	-	somes3accesskey
s3endpoint	Amazon S3 endpoint	-	s3endpoint.example.com
hdfsnamenode[3]	HDFS Namenode from where data is to be loaded. If a default namenode is not set in the Spark default configuration, it needs to be set when reading from HDFS. When working with the SAP Vora tools, it always needs to be set when creating a table.	-	namenode.example.com:8020
host[3]	Transaction coordinator host	-	txc.example.org
port[3]	Transaction coordinator port	-	2202
catalog.host[3]	Catalog host	-	catalog.example.org
catalog.port[3]	Catalog port	-	1234
catalog.timeout(3)	Timeout for catalog	3	5
schema[3]	Schema to be used for operations on SAP Vora	VORA	CUSTOMSCHEMA

[1] An empty cell indicates that there is no default value.

[2] Only works for the time series engine.

[3] Also works with the raw SQL syntax.

> **Note**
>
> You can set all properties globally in the `spark-defaults.conf` file by adding the prefix `spark.vora.engines`.

5.6 Raw SQL Scalar Functions

Supported Functions

- Numeric Functions [page 124]

Legend

integer	Specifies a parameter of type TINYINT, SMALLINT, INTEGER, or BIGINT
numeric	Specifies a parameter of numeric type (TINYINT, SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, DOUBLE)
string	Specifies a parameter of string type (CHAR or VARCHAR)
date	Specifies a parameter of type DATE
time	Specifies a parameter of type TIME
timestamp	Specifies a parameter of type TIMESTAMP
boolean	Specifies a parameter of type BOOLEAN

5.6.1 Numeric Functions

abs (numeric)

Returns the absolute value of a numeric argument.

acos (numeric)

Returns the arc-cosine, in radians, of the numeric argument.

asin (numeric)

Returns the arc-sine, in radians, of the numeric argument.

atan (numeric)

Returns the arc-tangent, in radians, of the numeric argument.

bitand (integer, integer)

Performs a bitwise AND operation on the two integer arguments.

bitnot (integer)

Performs a bitwise NOT operation on the argument.

bitor (integer, integer)

Performs a bitwise OR operation on the two integer arguments.

bitxor (integer, integer)

Performs a bitwise XOR operation on the two integer arguments.

ceil (numeric)

Returns the smallest integer that is greater than or equal to the specified value.

cos (numeric)

Returns the cosine of the angle, in radians, for the specified argument.

floor (numeric)

Returns the largest integer that is not greater than the specified numeric argument.

mod (<dividend>: numeric, <divisor>: numeric)

Returns the remainder of <dividend> divided by <divisor>.

ln (numeric)

Returns the natural logarithm of the specified argument.

log (numeric)

Returns the logarithm of base 10 of the specified number.

power (<base>: numeric, <exponent>: numeric)

Calculates a specified base number raised to the power of a specified argument.

round (<number>: numeric, <scale>: integer)

Rounds the specified argument to the specified number of digits after the decimal point.

sin (numeric)

Returns the sine of an angle expressed in radians.

sqrt (numeric)

Returns the square root of the specified argument.

tan (numeric)

Returns the tangent of a specified number, where the argument is an angle expressed in radians.

5.6.2 String Functions

concat (string,string)

Returns the combined string consisting of two specified strings.

length (string)

Returns the number of characters in a string.

locate (<haystack>: string, <needle>: string [, <reverse_flag>: integer])

Returns the first location of string `<needle>` in string `<haystack>`.

If `<reverse_flag>` is 1, the last location is returned.

If `<needle>` is not found, 0 is returned.

Examples:

- locate('abca', 'a') -> 1
- locate('abca', 'a', 1) -> 4
- locate('abca', 'd') -> 0

lower (string)

Converts all characters in a string to lowercase.

ltrim (<str>: string [, <remove_set>: string])

Returns a string, trimmed of all leading spaces.

If `<remove_set>` is specified, ltrim removes all the characters contained in this set from the start of string `<str>`.

replace (<original_string>: string, <search_string>: string, <replace_string>: string)

Searches a string for all occurrences of a specified string and replaces them with another specified string.

reverse (string)

Reverses the characters in a string.

rtrim (<str>: string [, <remove_set>: string])

Returns a string, trimmed of all trailing spaces.

If <remove_set> is specified, rtrim removes all the characters contained in this set from the end of string `<str>`.

sign (numeric)

Returns the sign (positive or negative) of the specified numeric argument.

substring (<str>: string, <start_position>: integer [, <string_length>: integer])

Returns a substring of `<str>` starting from `<start_position>` within the string.

When `<string_length>` is not specified, the entire substring starting from `<start_position>` is returned. When `<string_length>` is specified, the returned string consists of `<string_length>` characters starting from `<start_position>`. If `<string_length>` is greater than the length of the remaining part of `<str>`, then the remaining part is returned without blank padding.

trim ([[LEADING | TRAILING | BOTH] <trim_char>: string FROM] <str>:string)

Returns a string after removing leading and trailing spaces.

upper (string)

Converts all characters in a string to uppercase.

5.6.3 Datetime Functions

add_days (date, integer)

Computes the specified date plus the specified days.

add_hours (time | timestamp, integer)

Return a value of type TIME and TIMESTAMP, respectively, where increment hours are added to the given time or timestamp.

If the specified time is of type TIME, the operation is done modulo 24 hours such that the result always is a time between 00:00:00.0 and 23:59:59.9999999.

add_minutes (time | timestamp, integer)

Adds the specified number of minutes to the specified time or timestamp.

add_months (date, integer)

Adds the specified number of months to the specified date.

add_seconds (time | timestamp, integer)

Adds the specified number of seconds to the specified time or timestamp.

add_years (date, integer)

Adds the specified number of days to the specified date.

current_date

Returns the current local system date.

current_time

Returns the current local system time.

current_timestamp

Returns the current local system timestamp.

datediff ({NANOSECOND | MICROSECOND | MILLISECOND | SECOND | MINUTE | HOUR}, <datetime1>: date | time | timestamp, <datetime2>: date | time | timestamp)

Gives the signed number of the datepart boundaries crossed between the startdate `<datetime1>` and enddate `<datetime2>`.

Note that DATE and TIMESTAMP can not be compared with TIME.

When comparing DATE with TIMESTAMP the time 00:00:00 is assumed for the specified DATE.

Note that when the datepart is NANOSECOND, the returned number is always a multiple of 100 (since times are internally stored up to a precision of 100 nanoseconds).

dayname (date)

Returns the weekday in English for the specified date.

dayofmonth (date)

Returns the day of the month for the specified date.

dayofyear (date)

Returns an integer representation of the day of the year for the specified date.

days_between (date, date)

Computes the number of days between two dates.

hour (time | timestamp)

Returns an integer representation of the hour for the specified time or timestamp. The result is given in 24 hour format.

last_day (date)

Returns the date of the last day of the month that contains the specified date.

minute (time | timestamp)

Returns an integer representation of the minute for the specified time or timestamp.

month (date | timestamp)

Returns the number of the month from the specified date or timestamp.

monthname (date)

Returns the name of the month in English for the specified date.

now ()

Returns the current timestamp.

quarter (date | timestamp)

Returns the numerical year quarter of the specified date or timestamp.

The result depends only on the month of the date as follows:

- January, February, March -> 1
- April, May, June -> 2
- July, August, September -> 3
- October, November, December -> 4

second (time | timestamp)

Returns a value of the seconds for a given time or timestamp.

year (date | timestamp)

Returns the year number of a specified date or timestamp.

week (date)

Returns the week number of the specified date.

weekday (date)

Returns an integer representation for a given date.

Return values range from 1 to 7, representing Monday(1) to Sunday(7).

5.6.4 Data Type Conversion Functions

cast (<expression> as <data_type>)

Converts the expression to the target data type.

```
<data_type> ::=
BOOLEAN
| TINYINT
| SMALLINT
| BIGINT
| DECIMAL
| REAL
| DOUBLE
| CHAR
| VARCHAR
| DATE
| TIMESTAMP
| TIME
```

to_boolean (<expression>)

Converts the expression to type BOOLEAN.

to_bigint (<expression>)

Converts the expression to type BIGINT.

to_char (<expression>: date | time | timestamp, <format>: string)

Converts the specified date, time, or timestamp to data type CHAR. The `format` must be a constant CHAR or VARCHAR and specifies how to format the supplied value. For a full list of date and time format specifiers, see *Date and Time Format Specifiers*.

The to_char function only supports the conversion of types DATE, TIME, or TIMESTAMP.

Use the `cast (<expression> as CHAR(integer))` syntax to convert other types to CHAR and to explicitly specify the length of the resulting CHAR type.

to_date (<expression>: string | date | timestamp [, <format>: string])

Converts expressions of type VARCHAR, CHAR, or TIMESTAMP into a value of type DATE. If the input is of type DATE, then the input is returned unchanged.

If present, `format` must be a constant CHAR or VARCHAR. If `format` is omitted, "YYYY-MM-DD" is used as the format. For a full list of date and time format specifiers, see *Date and Time Format Specifiers*.

Timestamps are converted to DATE by dropping the time information.

Examples of correct use:

- TO_DATE('2010-05-13') -> 2010-05-13
- TO_DATE('2010-05-13', 'YYYY-MM-DD') -> 2010-05-13
- TO_DATE('February 28', 'MONTH DD') -> 0001-02-28
- TO_DATE('135/09', 'DDD/YY') -> 2009-05-15
- TO_DATE('14 8','YY MM') -> 2014-08-01
- TO_DATE('2008 FEB 29', 'YYYY MON DD') -> 2008-02-29
- TO_DATE('2008 366', 'YYYY DDD') -> 2008-12-31
- TO_DATE('08 03 24', 'YY MM DD') -> 2008-03-24
- TO_DATE('08 03 24', 'YYYY MM DD') -> 0008-03-24
- TO_DATE(TO_TIMESTAMP('2015-02-04 11:44:35.1234')) -> 2015-02-04

Examples of incorrect use:

- TO_DATE('2010-05-13-MAY', 'YYYY-MM-DD-MON') (invalid format)
- TO_DATE('February 28 143', 'MONTH DD DDD') (invalid format)
- TO_DATE('135/09', 'DDD-YY') (expression does not match format)
- TO_DATE('14 8 20', 'YY MM') (expression does not match format)
- TO_DATE('14 0008','YY MM') (number of digits of month exceeds 2)
- TO_DATE('2010-05-32', 'YYYY-MM-DD') (invalid date)
- TO_DATE('2009-02-29', 'YYYY-MM-DD') (invalid date)
- TO_DATE('2009 366', 'YYYY DDD') (invalid date)

to_decimal (<expression>, <precision>: integer, <scale>: integer)

Converts a value to data type DECIMAL that has the specified precision and scale.

`<precision>` and `<scale>` must be constant literals.

to_double (<expression>)

Converts the expression to type DOUBLE.

to_integer (<expression>)

Converts the expression to type INTEGER.

to_real (<expression>)

Converts a value to data type REAL.

to_smallint (<expression>)

Converts the expression to type SMALLINT.

to_time (<expression>: string | time | timestamp [, <format>: string])

Converts expressions of type VARCHAR, CHAR or TIMESTAMP into a value of type TIME. If the input is of type TIME, then the input is returned unchanged.

If present, `format` must be a constant CHAR or VARCHAR. If `format` is omitted "HH:MI:SS" is used as the format. For a full list of date and time format specifiers, see *Date and Time Format Specifiers*.

TIMESTAMPS are converted to TIME by dropping the date information.

Examples of correct use:

- TO_TIME('11:12:13') -> 11:12:13.0000000
- TO_TIME('11-12-13', 'HH-MI-SS') -> 11:12:13.0000000
- TO_TIME('11:12', 'HH24:MI') -> 11:12:00.0000000
- TO_TIME('3:15 PM', 'HH12:MI AP') -> 15:15:00.0000000
- TO_TIME('07:12:32 AM','HH:MI:SS AP') -> 07:12:32.0000000
- TO_TIME('15:16:17.0234', 'HH:MI:SS.FF4') -> 15:16:17.0234000
- TO_TIME(TO_TIMESTAMP('2015-02-04 11:44:35.1234')) -> 11:44:35.1234

Examples of incorrect use:

- TO_TIME('14:13', 'HH-MI') (expression does not match format)
- TO_TIME('14:13', 'HH:MI:SS') (expression does not match format)
- TO_TIME('24:13', 'HH:MI') (invalid time)
- TO_TIME('22:12:37 am', 'HH12:MI:SS AP') (invalid time)
- TO_TIME('14:13:12', 'HH:MI:HH') (invalid format)
- TO_TIME('4:13:12 pm', 'HH12:MI:HH') (invalid format, AP missing)
- TO_TIME('4:12345', 'HH:SSSSS') (invalid format)

to_timestamp (<expression>: string | date | time | timestamp [, <format>: string])

Converts expressions of type VARCHAR, CHAR, DATE, or TIME into a value of type TIMESTAMP. If the input is of type TIMESTAMP, then the input is returned unchanged.

If present, `format` must be a constant CHAR or VARCHAR. If `format` is omitted "YYYY-MM-DD HH:MI:SS.FF7" is used as the format. For a full list of date and time format specifiers, see *Date and Time Format Specifiers*.

If the input is of type TIME, the date part of the timestamp is filled with the current date.

If the input is of type DATE, the time part of the timestamp is filled with zero.

Examples of correct use:

- TO_TIMESTAMP(' ', ' ') -> 01-01-0001 00:00:00.0000000
- TO_TIMESTAMP('2015-02-04 11:44:35.1234') -> 2015-02-04 11:44:35.1234000
- TO_TIMESTAMP('1990-200 12345.01', 'YYYY-DDD SSSSS.FF2') -> 1990-07-19 03:25:45.0100000
- TO_TIMESTAMP('1954-03-02 2:42:21.7654321 pm', 'YYYY-MM-DD HH12:MI:SS.FF7 AP') -> 1954-03-02 14:42:21.7654321
- TO_TIMESTAMP(TO_TIME('11:12:13')) -> 20xx-xx-xx 11:12:13.0000000 (xx = current date)
- TO_TIMESTAMP(TO_DATE('2010-05-13')) -> 2010-05-13 00:00:00.0000000

Examples of incorrect use:

- TO_TIMESTAMP('2015:02:04 10-38-15', 'YYYY-MM-DD HH:MI:SS') (expression does not match format)
- TO_TIMESTAMP('2015-02-04', 'YYYY-MM-DD HH24:MI:SS.FF6' (expression does not match format; no time given)
- TO_TIMESTAMP('2009-366 11:11:11', 'YYYY-DDD HH:MI:SS') (invalid date)
- TO_TIMESTAMP('2015-02-09 22:12:37 am', 'YYYY-MM-DD HH12:MI:SS AP') (invalid time)

to_tinyint (<expression>)

Converts the expression to type TINYINT.

to_varchar (<expression> [, <format>: string])

Converts a given value to data type VARCHAR.

`<format>` must be a constant CHAR or VARCHAR. It can only be used if `<expression>` is of type DATE, TIME, or TIMESTAMP; it specifies how to format the supplied value. For a full list of date and time format specifiers, see *Date and Time Format Specifiers*.

The to_varchar function only supports the conversion of types DATE, TIME, or TIMESTAMP.

Use the `cast (<expression> as VARCHAR(integer))` to explicitly specify the length of the resulting VARCHAR type.

Related Information

Date and Time Format Specifiers [page 138]

5.6.5 Miscellaneous Functions

coalesce (<expression1>, <expression2>, ...)

Returns the first non-NULL expression from a list.

nullif (<expression1>, <expression2>)

Compares both expressions. If both expressions are equal, NULL is returned. Otherwise, `<expression1>` is returned.

5.6.6 Like Predicate

```
<str> [NOT] LIKE <pattern> [ ESCAPE <escape> ]
```

Returns true if `<pattern>` can be matched to `<str>` and false otherwise. If `NOT` is present, the return value is reversed. The match is case sensitive.

`<string>`, `<pattern>`, and `<escape>` are of type CHAR or VARCHAR. If present, `<escape>` must have length 1.

`<pattern>` may include the following wildcards:

- An underscore ('_') matches any single character.
- A percent sign ('%') matches zero or more characters.
- If `<escape>` is specified, then `<pattern>` is valid if any occurrences of `<escape>` in `<pattern>` are followed by either '_', '%', or `<escape>`. In this case, the character that follows a non-escaped `<escape>` character represents an occurrence of the literal character in `<pattern>`. If both `<str>` and `<pattern>` are of type CHAR, then spaces at the end of both parameters are ignored.

Examples:

- 'Hello' LIKE 'Hello' -> true
- 'Hello' LIKE 'Yellow' -> false
- 'Hello' LIKE 'hello' -> false

- 'Hello' LIKE '' -> false
- '' LIKE '' -> true
- 'Hello' LIKE '_ello' -> true
- 'Hello World' LIKE 'Hello%' -> true
- 'Dr. John Smith' LIKE '%John%' -> true
- 'Dr. John Smith' LIKE '%John%' ESCAPE '\' -> true
- 'Dr. John Smith' LIKE '%John\%' ESCAPE '\' -> false
- 'Dr. John%' LIKE '%John\%' ESCAPE '\' -> true
- 'Dr. John% Smith' LIKE '%John_%' ESCAPE '_' -> false
- 'Dr. John% Smith' LIKE '%%John%' ESCAPE '%' -> false
- 'Hello World!' LIKE '%World!!' ESCAPE '!' -> true
- 'Hello World!' LIKE '%World!' ESCAPE '!' -> Invalid escape sequence
- 'Hello ' LIKE 'Hello' -> true (if parameters of type CHAR)
- 'Hello' LIKE 'H% _ %' -> true (if parameters of type CHAR)

5.6.7 Date and Time Format Specifiers

Possible format specifiers are:

- YYYY: The year including the century. Range = [0001,9999].
- YY: The year within the century. Range = [00,99]. Values in the range [69,99] refer to years 1969 to 1999 inclusive, and values in the range [00,68] refer to years 2000 to 2068 inclusive.
- RRRR: The year with or without the century. Range = [0001,9999] or [00,99]. If the century is provided, the behavior is the same as for YYYY. If the century is not provided and :
  - The specified two-digit year is 00 to 49:
    - If the last two digits of the current year are 00 to 49, then the returned year has the same century as the current year.
    - If the last two digits of the current year are 50 to 99, then the century of the returned year is 1 greater than the century of the current year.
  - And the specified two-digit year is 50 to 99:
    - If the last two digits of the current year are 00 to 49, then the century of the returned year is 1 less than the century of the current year.
    - If the last two digits of the current year are 50 to 99, then the returned year has the same century as the current year.
- RR: The year without the century. Range = [00,99]. Same behavior as for RRRR when century is omitted.
- MM: The month number. Range = [01,12].
- MON: Abbreviated month name (for example, DEC).
- MONTH: Full month name (for example, DECEMBER).
- DD: The day of the month. Range = [01,31].
- DDD: The day number of the year. Range = [001,366].
- HH: The hour in 12 or 24 hours format (depending on if AP appears as well or not). Range = [00,23] or [01,12].
- HH12: The hour in 12 hours format. Range = [01,12]. AP needs to appear as well.
- HH24: The hour in 24 hours format. Range = [00,23].

- AP: Meridian indicator. Range = {AM,PM}.
- MI: The minute of the hour. Range = [00,59].
- SS: The second of the minute. Range = [00,59].
- SSSSS: The second of the day. Range = [00,86399].
- FF[1-7]: The fraction of the second in 100 nanoseconds. Range = [0000000,9999999]. The precision specifier is optional; the default is 7. Leading zeros are recognized, for example, 0010000 yields 1 millisecond.
- SF: The second of the minute with an optional fraction (separated by a '.'), that is, SF is equal to SS[.FF7].

Note the following:

- Format strings are case insensitive, that is, the format 'yYyY-MM-dD FOObar' matches '2015-01-01 fooBAR'.
- For all numeric fields, leading zeros are permitted but not required (note the special behavior for FF).
- The default value for year, month, and day is 1. The default value for hour, minute, second and fraction is 0.
- Time specifiers can occur in date formats and are ignored, and vice versa.

Restrictions:

- Only one month specifier can occur in `format`.
- DDD cannot occur together with DD or any month specifier in `format`.
- Only one hour specifier (HH, HH12, or HH24) can occur in `format`. If HH24 occurs, AP must not occur as well. If HH12 occurs, AP needs to occur as well.
- SSSSS cannot occur together with MI or any hour specifier in `format`.
- The given date expression must match the given format (trailing spaces are ignored if the expression is of type CHAR and either `format` is omitted or `format` is also of type CHAR).
- The number of digits in a numeric field cannot exceed the length of the format specifier.
- The given date, time, and timestamp expression must be a valid date, time and timestamp, respectively.

6 Partitioning Tables

Tables created with `com.sap.spark.engines.relational`, `com.sap.spark.engines.disk`, and `com.sap.spark.engines` are not automatically partitioned, which means that they are not, by default, distributed over all available nodes in the landscape. Unless partitioning is applied, they are always assigned to a single host in the cluster.

To partition tables, you use partition functions and partition schemes. Partition functions allow you to define how tables should be partitioned, while partition schemes are derived from partition functions and are used to apply the partition functions to the tables.

6.1 Partition Function

A partition function allows you to define how the partitions of a table are created. The supported partition function types are range partitioning, hash partitioning, and block partitioning.

You can create a partition function using the CREATE PARTITION FUNCTION command. The general syntax is shown below:

```
<partition_function> ::=
  CREATE PARTITION FUNCTION <name> <pf_definition>
<pf_definition> ::= <column_type_list> AS <partition_function_type>
<column_type_list> ::= '(' {<identifier> <type>},... ')'
<partition_function_type> ::= {
    <range_partitioning>
 || <hash_partitioning>
 || <block_partitioning>
 || <system_partitioning>
 }
<range_partitioning> ::= RANGE [ '('<col_identifier>')' ]
    [<boundary_def>]
    [ MIN PARTITIONS <numeric_literal> ]
    [ MAX PARTITIONS <numeric_literal ]
<hash_partitioning> ::= HASH
  [ '(' <col_list> ')'] [ MIN PARTITIONS <numeric_literal> ]
  [ MAX PARTITIONS <numeric_literal> ]
<block_partitioning> ::= BLOCK
    '(' <col_list> ')'
    PARTITIONS <numeric_literal>
    BLOCKSIZE <numeric_literal>
<system_partitioning> ::= AUTO
<col_list> ::= <col_identfier>, ...
```

The partition function determines how many partitions are created. Since each partition is placed on one host, you should ensure that the number of partitions is equal to or greater than the number of hosts over which you want to distribute the table.

For example, if you have 100 nodes and want to distribute a table over all of them, you should create a partition function that results in at least 100 partitions. For range partitioning, you should define at least 99 boundaries (number of boundaries equal to or greater than n - 1, where n is the target number of nodes). For hash

partitioning, you should specify 100 as the minimum number of partitions (minimum partition number equal to the target number of nodes).

Related Information

RANGE Partitioning [page 141]
HASH Partitioning [page 141]
BLOCK Partitioning [page 142]

6.1.1 RANGE Partitioning

Range partitioning allows you to define certain values of a specified column as the boundaries for dividing the data into separate partitions.

For example, given a value range from [0..10), you could specify the values 3 and 6 to create 3 partitions. The first partition would include the values [0..3), the second partition the values [3..6), and the third partition the values [6..10). The definition of a minimum number of partitions and a maximum number of partitions gives the system the flexibility to create further partitions not specified as boundaries. It is important that the maximum number of partitions is not smaller than the number of boundaries + 1.

Sample Code

```
CREATE PARTITION FUNCTION PF1 ( X INT )
  AS RANGE BOUNDARIES( 2, 4 )
  MIN PARTITIONS 3;
```

6.1.2 HASH Partitioning

Using HASH partitioning, the values of the specified columns are hashed and added to a partition based on their hash values.

You can define the number of partitions available by giving a minimum and/or maximum number of partitions allowed. Within the given boundaries, the system is allowed to freely choose the exact number of partitions. For example, given a minimum partition number of 4, you could use the `modulo` function as a hash function and compute the hash value accordingly.

Sample Code

```
CREATE PARTITION FUNCTION PF1 ( X INTEGER )
  AS HASH(X)
  MIN PARTITIONS 5
  MAX PARTITIONS 5;
```

6.1.3 BLOCK Partitioning

Block partitioning is a special type of partitioning that is designed for partitioning graphs. It divides the nodes of a graph into different blocks and buckets.

You define a block size and the number of partitions. Blocks and buckets are computed as follows. Given a block size `S` and a number of partitions `N`, the `block id` is computed as `node id / S`. These blocks are then assembled into buckets, where buckets are defined as `block id / C`. Each host is responsible for exactly one bucket.

Sample Code

```
CREATE PARTITION FUNCTION PF1_GRAPH(X BIGINT)
  AS BLOCK(X)
  PARTITIONS 4
  BLOCKSIZE 1000;
```

6.2 Partition Scheme

Use a partition scheme to apply and instantiate a partition function.

First create a partition scheme using the following syntax:

```
<partition_scheme> ::=
  CREATE PARTITION SCHEME <name> <pf_reference>
<pf_reference> ::=
  USING <partition_function_identifier>
```

Sample Code

```
CREATE PARTITION SCHEME PS1 USING PF1;
```

Then apply the partition scheme as part of the CREATE table statement by simply adding the PARTITION BY clause to the end of it. The PARTITION BY clause contains a reference to the respective partition scheme. In addition, the partition scheme and partition function are instantiated by providing the concrete partitioning columns as function parameters to the partition scheme. The syntax is as follows:

```
<partition_clause> ::=
  PARTITION BY <partition_scheme_id> '('<col_list>')'
```

Sample Code

```
CREATE TABLE ExampleTable (
  key1 integer,
  val1 integer
) PARTITION BY PS1(key1);
```

> **Note**
>
> The PARTITION BY clause is not supported by the disk engine and relational in-memory engine. Instead, you specify the partition scheme by using the `ps` option in the CREATE TABLE statement.

Related Information

Disk Engine and Relational Engine Data Source API [page 53]
Creating Partitioned Tables in the Relational Engines [page 26]

6.3 Engine Compatibility Overview

The individual SAP Vora engines support the partition function types as shown below.

Engine	RANGE	HASH	BLOCK
Disk Engine	x	x	-
Relational In-Memory Engine	x	x	-
Graph Engine	-	-	x
Document Store	-	x	-
Time Series Engine	x(Period)	x	-

7 Reloading Engine Tables After a Cluster Restart

You can explicitly ensure that tables, collections, and graphs are loaded into the (main) memory of the respective engine, for example, after a cluster restart.

Tables, collections, and graphs can be loaded in the following ways:

- Individually using Spark SQL
- Individually using raw SQL
- All together in one run using Spark code

Loading tables individually using Spark SQL

- For disk engine tables:

```
LOAD TABLE <TABLENAME> USING com.sap.spark.engines
```

- For time series tables:

```
LOAD SERIES TABLE <SERIESNAME> USING com.sap.spark.engines
```

- For document collections:

```
LOAD COLLECTION <COLLECTIONNAME> USING com.sap.spark.engines
```

- For graphs:

```
LOAD GRAPH <GRAPHNAME> USING com.sap.spark.engines
```

Loading tables individually using raw SQL

- For disk engine tables:

```
``LOAD TABLE <TABLENAME>`` USING com.sap.spark.engines AS ()
```

- For time series tables:

```
``LOAD TABLE <SERIESNAME>`` USING com.sap.spark.engines AS ()
```

- For document collections:

```
``LOAD COLLECTION <COLLECTIONNAME>`` USING com.sap.spark.engines AS ()
```

- For graphs:

```
``LOAD GRAPH <GRAPHNAME>`` USING com.sap.spark.engines AS ()
```

Loading all tables, graphs, and collections using Spark code

You can reload all tables, graphs, and collections together using the following Spark code:

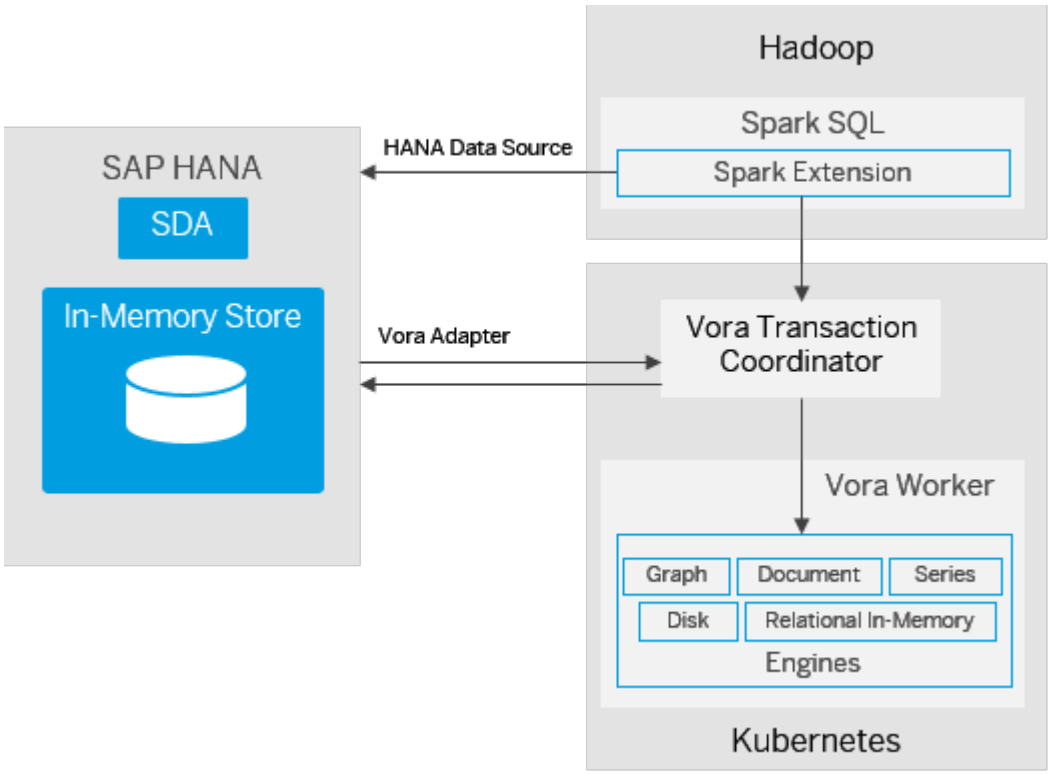
```
import com.sap.spark.engines.client.EngineClient
import com.sap.spark.engines.config.EngineConfiguration
val engineConfig = EngineConfiguration(sqlContext, Map.empty[String, String])
val client = EngineClient(sqlContext, engineConfig)
val loaded = client.reloadTables()
```

You can use `Map.empty[String, String]` if the configuration has been defined in the `spark-defaults.conf` file or using `sqlContext.setConf()`. If not, specify the configuration in the map, for example, `Map("spark.vora.engines.host" -> "vorahost.example.com")`.

8 Connectivity Between SAP HANA and SAP Vora

The SAP HANA data source provides a pluggable mechanism for accessing data stored in SAP HANA from a Spark-based environment through Spark SQL. It includes an enhanced data source API implementation that supports predicate pushdown for all predicates that SAP HANA can process.

While the SAP HANA data source supports read and write access from SAP Vora to SAP HANA, read and write access from SAP HANA to SAP Vora requires a connection through SAP HANA smart data access (SDA), as shown below:



Related Information

Creating Tables Using the SAP HANA Data Source [page 147]
Code Example [page 149]
Pushing Down SAP HANA UDFs [page 153]
SAP HANA Data Source API [page 154]
Accessing SAP Vora from SAP HANA [page 155]

8.1 Creating Tables Using the SAP HANA Data Source

A CREATE TABLE statement registers the table in the Spark SQLContext and creates a table in the SAP HANA database.

Creating Tables

You need to provide a table name, the fully qualified name of the SAP HANA data source package, `com.sap.spark.hana`, and a set of options required by the data source.

Sample Code

```
CREATE TABLE $tableName
USING com.sap.spark.hana
OPTIONS (
tablepath "$tableName",
dbschema "$dbSchema",
host "$host",
instance "$instance",
user "$user",
passwd "$passwd"
)
```

Semantics

Tables from the SAP HANA catalog can be referenced from the Spark session catalog (see *Working with the Spark Catalog and SAP Vora Catalog*). Essentially, the table metadata is copied between the catalogs.

To reflect an SAP HANA table in the Spark session catalog, you can use a CREATE TABLE statement that references an existing SAP HANA table. The table metadata is copied from the SAP HANA catalog into the Spark session catalog. The table in SAP HANA has to exist prior to importing it into SAP Vora.

You can also create new tables in SAP HANA using the CREATE TABLE command. A prerequisite is that the table does not yet exist in SAP HANA.

Consider the following cases:

- CREATE TABLE with table schema information
  - If the table with the given name does not yet exist in the SAP HANA database, the table is created in SAP HANA and registered in the Spark context. The table metadata is stored in the SAP Vora catalog.

    Note

    When a table is created that does not yet exist in the SAP HANA database, it will only be persisted if data is inserted into it.

  - If the table with the given name already exists in the SAP HANA database and the specified table schema matches that of the existing table, the table is registered in the Spark context as well as in the

SAP Vora catalog. If these conditions are not met, the statement will fail and the table will not be registered.

- CREATE TABLE without table schema information
  - If the table with the given name does not yet exist in the SAP HANA database, the statement will fail and the table will not be registered.
  - If the table with the given name already exists in the SAP HANA database, the table is registered in the Spark context as well as in the SAP Vora catalog.

> **Note**
>
> You can also reflect SAP HANA tables in the Spark catalog using the REGISTER TABLE command. For more information, see *Registering Tables in Spark*.

> **Note**
>
> Identifiers for tables, schemas, and columns are case sensitive in Spark. In SAP HANA, identifiers are case insensitive by default, and case sensitive when surrounded by quotation marks (double quotes). Consider the following consequences:
>
> - A case-insensitive identifier (for example, `table1`) in SAP HANA is referenced in Spark in upper case (as `TABLE1`).
> - A case-sensitive identifer (for example `"table1"`) in SAP HANA is referenced in Spark as is, but without quotation marks (as `table1`).
>
> These rules need to be taken into account when referencing existing SAP HANA tables in CREATE TABLE commands in SAP Vora. Otherwise, the tables might not be correctly found in the SAP HANA catalog. When you create tables from SAP Vora in the SAP HANA database, the identifiers are stored in the case-sensitive form.

Inserting Data

Data can be loaded into a table in the SAP HANA database from a DataFrame in Spark as follows:

> **Sample Code**

```
dataFrame.write.format("com.sap.spark.hana").mode(SaveMode.Append).options(tabl
eConf).save()
```

In general for all save modes, if the table does not yet exist in SAP HANA, a new table is created in the SAP HANA database with the DataFrame's schema and data is inserted into that table. If the table already exists in SAP HANA, the behavior is as follows:

SaveMode.Append	Data is appended to the existing table.
SaveMode.Overwrite	Data of the current table is dropped and new data is inserted.
SaveMode.ErrorIfExists	The statement fails and no changes are made to the existing table.
SaveMode.Ignore	The statement doesn't fail and no changes are made to the existing table.

Related Information

Code Example [page 149]
SAP HANA Data Source API [page 154]
Working with the Spark Catalogs and the SAP Vora Catalog [page 18]
Registering Tables in Spark [page 35]

8.2 Code Example

The following code example shows how a table can be created and queried in Spark using the SAP HANA data source.

Sample Code

```
import org.apache.spark.sql._
import org.apache.spark.sql.types._
val nameNodeHostAndPort = "name.node.mycompany.corp:8020"
val pathToCsvFile = "/path/to/file.csv"
val sqlc = new SapSQLContext(sc)
// this table name holds for HANA as well as for the Vora instance
val tableName = "people_test"
//Database Schema Name
val dbSchema = "dbschema"
// HANA Host instance
val host = "hana.host1.com"
// HANA instance ID
val instance = "02"
//User name and password
val user = "myuser"
val passwd = "mzpassword"
/* Creating the new table */
sqlc.sql(
 s"""CREATE TABLE $tableName (name string, age integer)
 USING com.sap.spark.hana
 OPTIONS (
 tablepath "$tableName",
 dbschema "$dbSchema",
 host "$host",
 instance "$instance",
 user "$user",
 passwd "$passwd")""".stripMargin)
val dataRDD = sc.textFile("hdfs://$nameNodeHostAndPort$pathToFile")
val rowRDD = dataRDD.map(_.split(",")).map(p =>
org.apache.spark.sql.Row(p(0),p(1).toInt))
val schema =
StructType(Array(StructField("name",StringType,false),StructField("age",Integer
Type,false)))
val dataframe = sqlc.createDataFrame(rowRDD, schema)
val configuration = Map(("host"-> host), ("instance"-> instance), ("user"->
user), ("passwd"-> passwd))
val writeOptions = configuration + ("tablepath" -> tableName) + ("dbschema" ->
dbSchema)
dataFrame.write.format("com.sap.spark.hana").mode(SaveMode.ErrorIfExists).optio
ns(writeOptions).save()
val queryResult = sqlc.sql("SELECT name, age from people_test where age > 10")
queryResult.collect().foreach(println)
```

8.3 Listing Tables in SAP HANA

Tables created in Spark exist only for the lifetime of a particular Spark SQLContext. The corresponding data source tables, however, are persisted in the SAP HANA backend database and can be listed using the SHOW TABLES statement, as shown in the example below.

Sample Code

```
SHOW TABLES USING com.sap.spark.hana
OPTIONS (
host "$host",
instance "$instance",
user "$user",
passwd "$passwd",
dbschema "$dbSchema",
tablePattern "$pattern"
)
```

The optional parameters `dbschema` and `tablePattern` allow you to specify SQL-LIKE patterns for the SAP HANA schema from which you want to display the tables, and for the names of the source tables.

For example, `dbschema` set to "spark" and `tablePattern` set to "%8" will display all tables with the digit "8" at the end of their names created within the schema "spark". If `dbschema` is not specified, then the default schema equal to the value of the `user` parameter will be used. The default `tablePattern` value is "%", which matches all tables. It is therefore recommended that you use these parameters to avoid displaying too many tables.

Note

The SHOW DATASOURCETABLES command, which can be used in the same way, has been deprecated and should therefore not be used. Please use SHOW TABLES instead.

8.4 Registering SAP HANA Tables in Spark

When you instantiate a new Spark SQLContext, the tables you worked with in a previous session no longer exist.

REGISTER TABLE

You can use the REGISTER TABLE statement to register a specific table already created in the SAP HANA data source into the Spark catalog. No additional metadata or schema information is needed to perform the registration:

```
REGISTER TABLE tablename USING com.sap.spark.hana
OPTIONS (
```

```
dbschema "$dbSchema",
host "$host",
instance "$instance",
user "$user",
passwd "$passwd"
) [IGNORING CONFLICTS]
```

An error is thrown if the table already exists in the Spark catalog, but you can force the registration by using the `IGNORING CONFLICTS` clause, which then overwrites that table in the Spark catalog.

Alternatively you can use the `IF NOT EXISTS` clause to skip the registration if the table is already registered.

REGISTER ALL TABLES

You can use the REGISTER ALL TABLES statement to register all tables already created in the SAP HANA data source into the Spark catalog. No additional metadata or schema information is needed to perform the registration:

```
REGISTER ALL TABLES USING com.sap.spark.hana
OPTIONS (
host "$host",
instance "$instance",
user "$user",
passwd "$passwd",
[dbschema "$dbSchema"],
[tablePattern "$pattern"]
) [IGNORING CONFLICTS]
```

The optional parameters `dbschema` and `tablePattern` allow you to specify SQL-LIKE patterns for the SAP HANA schema from which you want to register the tables, and for the names of the source tables.

For example, `dbschema` set to "spark" and `tablePattern` set to "%8" will register all tables with the digit "8" at the end of their names created within the schema "spark". If `dbschema` is not specified, then the default schema equal to the value of the `user` parameter will be used. The default `tablePattern` value is "%", which matches all tables. It is therefore recommended that you use these parameters to avoid registering too many tables within the Spark context.

An error is thrown if any of the tables already exist in the Spark catalog, but you force the registration by using the the `IGNORING CONFLICTS` clause, which then overwrites those tables in the Spark catalog.

Alternatively you can use the `IF NOT EXISTS` clause to skip the registration of any tables that are already registered.

8.5 Dropping SAP HANA Tables

The `DROP TABLE` command drops the specified table in the Spark context. If the table is a non-virtual table, it also deletes the corresponding in-memory SAP HANA table (provided it exists and the SAP HANA user is allowed to perform the action).

Sample Code

```
DROP TABLE testTableName
```

If you are not sure whether the table exists, you can add `IF EXISTS` to the clause as follows:

Sample Code

```
DROP TABLE IF EXISTS testTableName
```

An exception will not be thrown if the table does not exist.

If the specified table is referenced more than once in the SAP Vora catalog, the drop table action will fail. To drop both the table and every entry in the catalog that references it, add the `CASCADE` suffix to the `DROP TABLE` statement as follows:

Sample Code

```
DROP TABLE testTableName CASCADE
```

8.6 Displaying Table Metadata

You can display table metadata using the DESCRIBE TABLE statement.

```
DESCRIBE TABLE tablename USING com.sap.spark.hana
OPTIONS (
dbschema "$dbSchema",
host "$host",
instance "$instance",
user "$user",
passwd "$passwd"
)
```

8.7 Pushing Down SAP HANA UDFs

The SAP HANA data source allows you to use UDFs that are implemented solely in SAP HANA (that is, they do not exist in Spark). You can do this by using the "$" prefix.

The following example shows how to push down a unit of measure conversion:

Sample Code

```
import org.apache.spark.sql._
val sqlc = new SapSQLContext(sc)
lazy val configuration = Map(("host"->"hana.host1.com"),
 ("instance"->"02"),
 ("user"->"myuser"),
 ("passwd"->"secret"))
lazy val sampleInputConf =
 configuration + ("tablepath" -> "SAMPLE_INPUT") + ("dbschema" -> "SAPCCH")
sampleInputRelation =
 sqlc.read.format("com.sap.spark.hana").options(sampleInputConf).load()
sampleInputRelation.registerTempTable("SAMPLE_INPUT")
var query = sqlc.sql("Select $CONVERT_UNIT" +
 "(QUANT,SOURCE_UNIT,'SAPCCH',TARGET_UNIT,'000') as converted " +
 "FROM SAMPLE_INPUT")
queryResult.collect().foreach(println)
```

8.8 Using SAP HANA Secure Store

The SAP HANA secure user store (hdbuserstore) is a tool installed with the SAP HANA client. It is used to store the credentials of SAP HANA systems securely on the client so that client applications can connect to SAP HANA without users having to enter this information.

The SAP HANA data source is able to use the SAP HANA secure user store to connect to SAP HANA. To that end, the SAP HANA secure user store files need to be distributed to the same folder on all nodes on which Spark runs. The default value of this folder is `$HOME/.hdb`, where `$HOME` is the home folder of the corresponding Spark user. If a different folder path needs to be used, the path can be set using the Spark parameter `securestorehanapath`. The SAP HANA data source reads the credentials from secure store files indexed with a key, which is specified with the Spark parameter `securestorekey`.

For example:

```
SHOW TABLES USING com.sap.spark.hana
OPTIONS (
securestorekey "mykey",
securestorehanapath "/home/vora/.hdb",
dbschema "$dbSchema",
tablePattern "$pattern"
)
```

Note

The secure store files, `SSFS_HDB.DAT` and `SSFS_HDB.KEY`, need to be protected with strict file permissions so that only the user who runs the Spark job can access them. If Spark jobs are run in YARN modes, the user

permission needs to be handled accordingly. For instance, Spark jobs are run with the YARN user in the executors unless YARN is Kerberized and configured accordingly.

8.9 SAP HANA Data Source API

The SAP HANA data source API provides a set of table options that can added to Spark SQL statements or specified in the Spark configuration file (`spark-defaults.conf`). The options are shown below.

Name	Description	Default Value[1]	Example
host	Host name of the SAP HANA server		hana.host1.com
instance	This is a double digit number which computes the local port used to access the SAP HANA server. The local port is derived from the local instance number as 3<instance number>15. For example, if the instance number is 02, then the local port will be 30215.		02
port	Port suffix of the JDBC connection string. For example, if the instance number is 02 and the port suffix is set to 13, the connection port will be 30213. By modifying this suffix, it is possible to connect directly to a specific SAP HANA database in a multitenant SAP HANA installation.	13 if a tenant database is specified, otherwise 15	41, 44
tenantdatabase	Name of the target tenant database in mutliten-ant SAP HANA installations (translated to the databaseName parameter in the SAP HANA connection string).		myDb
tablepath	SAP HANA database table or view		tableName
dbschema	SAP HANA database schema of the table specified in the parameter above	SYSTEM	mySchema
user	SAP HANA database user		user
passwd	Password for the SAP HANA database user specified in the parameter above		passwd
schema	SAP HANA schema to be used to create the backend table		name varchar(200), age integer
batchsize	Batch size for loading partitioned data frames	1000	1000
columnStore	Should be set to true if a column table is to be created in the SAP HANA backend	false	true
virtual	Should be set to false if the table is a non-virtual table. Non-virtual tables are deleted from the SAP HANA backend when they are dropped from Spark. Virtual tables (default) can be dropped only from Spark.	true	false

Name	Description	Default Value[1]	Example
partitioningColumn	Column used for table partitioning		age
numberOfPartitions	Number of partitions on the table. It is recommended that the number of partitions should be close to the number of Spark workers in the landscape.		10
maximumPartitionSize	Maximum number of rows		1000
securestorekey	The key of the SAP HANA credentials stored in the SAP HANA secure store. If this parameter is set, the SAP HANA data source uses the credentials of the corresponding key to connect to SAP HANA.		mykey
securestorehanapath	The path of the folder that stores the SAP HANA secure store files	$HOME/.hdb/	/home/vora/.hdb

[1] An empty cell indicates that there is no default value.

> **Note**
>
> You can set all properties globally in the `spark-defaults.conf` file by adding the prefix `spark.hana`, for example, `spark.hana.host` or `spark.hana.instance`.

8.10 Accessing SAP Vora from SAP HANA

You can connect to and access data in SAP Vora from SAP HANA using SAP HANA smart data access (SDA).

SAP Vora Remote Source Adapter

You can create an SDA remote source connection directly to the SAP Vora cluster using the SAP Vora remote source adapter `voraodbc`.

You can use the SAP Vora remote source adapter as follows:

1. Ensure that the SAP HANA Wire protocol is enabled. See Enable the SAP HANA Wire for Smart Data Access [page 156].
2. Create a remote source using the `voraodbc` remote source adapter. See Create an SAP Vora Remote Source [page 156].
3. Create virtual tables that represent the tables you want to access in the SAP Vora remote source. See Create Virtual Tables [page 158].
4. Alternatively, directly execute queries on the remote source. See Executing Remote Queries [page 162].
5. Optionally reroute stored procedures from SAP HANA to SAP Vora, so that they run directly on the applicable objects. See Reroute Stored Procedures [page 163].

> **Note**
>
> - The `voraodbc` SDA adapter is delivered with SAP HANA 1.0 SPS 12 revision 122.05 and higher.
> - You can currently only create virtual tables based on tables in the SAP Vora disk engine or relational in-memory engine.

8.10.1 Enable the SAP HANA Wire for Smart Data Access

SAP Vora supports the SAP HANA Wire protocol, which allows a direct connection to be established from SAP HANA to SAP Vora using SAP HANA smart data access (SDA).

Context

The SAP HANA Wire is implemented in the SAP Vora transaction coordinator and is enabled by default.

Procedure

1. Open the Kubernetes Dashboard.
2. Select your namespace, if applicable, and choose *Deployments*.
3. In the list on the *Deployments* screen, find *vora-tx-coordinator*. Click the *Actions* menu and choose *View/edit YAML*.
4. In the *spec/template/spec/containers/0/args* section, make the following settings:
   - Set *tc_hana_wire_enabled* to **`true`**.
   - In the *tc_hana_wire_instance_number* field, enter the instance number of the Vora cluster. This number will be used to derive the port number of the Vora transaction coordinator for the remote source connection.
5. Choose *Update* to apply your changes.

8.10.2 Create an SAP Vora Remote Source

Create an SDA remote source connection directly to the SAP Vora cluster using the SAP Vora remote source adapter `voraodbc`.

Prerequisites

You have enabled the SAP HANA Wire. See Enable the SAP HANA Wire for Smart Data Access [page 156].

Procedure

On the SAP HANA instance, create a remote source using the following SQL statement:

> **Note**
>
> The SAP HANA Studio remote source editor (UI) does not currently support the SAP Vora remote source adapter.

```
CREATE REMOTE SOURCE <Name> ADAPTER "voraodbc"
CONFIGURATION  'ServerNode=<TC Server>:<TC HANA Wire Port>;Driver=libodbcHDB'
WITH CREDENTIAL TYPE 'PASSWORD' USING 'user=my_user;password=my_password';
```

Replace the variables as follows:

- <Name>: Name of the remote source to be displayed in the SAP HANA Studio.
- <TC Server>: IP address/domain name of the server in the SAP Vora cluster on which the transaction coordinator is running.
- <TC HANA Wire Port>: SAP HANA Wire port of the transaction coordinator, which is determined as 3XX15, where XX is the instance number of the SAP Vora cluster as configured in the Kubernetes cluster manager. (Note that it is not the number of the SAP HANA instance adding the SAP Vora cluster as a remote source). For example, when the instance number of the SAP Vora cluster is configured as 25, the SAP HANA Wire port is 32515.

> **Note**
>
> SAP Vora does not currently check credentials. You can use any user and password.

Results

The remote source is now listed under *Provisioning* > *Remote Sources*. Expand the remote source to see the users and tables.

Related Information

Enable the SAP HANA Wire for Smart Data Access [page 156]
CREATE REMOTE SOURCE

8.10.3 Create Virtual Tables

Virtual tables represent the tables you want to access in the SAP Vora remote source. You can add one or more remote objects as virtual tables.

Prerequisites

A remote source connection has been created using the `voraodbc` adapter. See Create an SAP Vora Remote Source [page 156].

Context

You can access data stored in the SAP Vora disk engine and SAP Vora relational in-memory engine. You can also query the time series engine and document store using remote query execution (see *Executing Remote Queries*). You cannot currently access data in the graph engine.

Procedure

Add the table you want to access in the remote source as a virtual table:

Option	Steps
SAP HANA Studio Provisioning UI	1. Expand the remote source to see the users and tables: *Provisioning* > *Remote Sources* > *<remote-source>* > *<user>*. 2. Browse the tables and select the table you want to access. 3. From the context menu, choose *Add as Virtual Table*. 4. Enter a table name, select the schema where the virtual table should be stored on your SAP HANA instance, and choose *Create*.

SQL Command	`CREATE VIRTUAL TABLE <local schema name>.<local table name>`
	`AT "<remote source>"."<NULL>"."<remote schema>"."<remote table>"`
	
	Replace the variables as follows:
	○ <local schema name>: Name of the schema on the SAP HANA instance in which the virtual table should be created.
	○ <local table name>: Name to be assigned to the virtual table.
	○ <remote source>: Name of the remote source in which the remote table is located.
	○ <remote schema>: Name of the schema in the remote source in which the remote table is located.

Option	Steps
	○ <remote table>: Name of the table to be added as a virtual table from the remote source. Note that <NULL> is the NULL database item displayed when you browse the remote source in the SAP HANA studio.

> **Note**
>
> SAP HANA imposes a maximum length of 256 characters for the names of schemas, tables, and columns. If an SAP Vora table does not meet these requirements, it cannot be added as a virtual table.

Results

The new virtual table is now listed under *Catalog* > *<schema>* > *Tables*. You can run SQL queries on virtual tables in the same way as with normal SAP HANA tables.

Related Information

Managing Virtual Tables
Data Type Restrictions [page 159]
Executing Remote Queries [page 162]

8.10.4 Data Type Restrictions

The mappings between the SAP HANA and SAP Vora data types are shown below.

Vora SQL Type	HANA SQL Type	Differences/Limitations
CHAR	CHAR	Maximum length in SAP HANA: 2000 characters. An SAP Vora table cannot be added as a virtual table if one of its columns exceeds the limit (in this case the error message in SAP HANA is inconclusive). Note: CHAR(*) in SAP Vora maps to 2bn (which is larger than the 2000 limit). Note: The CHAR type does not currently support high Unicode characters. Use VARCHAR to store high Unicode special characters.

Vora SQL Type	HANA SQL Type	Differences/Limitations
VARCHAR	VARCHAR1	Maximum length in SAP HANA: 5000 characters. An SAP Vora table cannot be added as a virtual table if one of its columns exceeds the limit. Note: Certain character UDFs behave differently when the target string contains Unicode codepoints > 10K. For STRLEN, for example, which returns the number of UTF-8 characters, SAP HANA counts each codepoint twice after 10K, while SAP Vora counts it once. This leads to different results when STRLEN is evaluated on SAP Vora and SAP HANA.
TINYINT/INT8	TINYINT	The following value is used to represent `null` in SAP Vora: minimum integer value.
SMALLINT/INT16	SMALLINT	
INTEGER/INT32	INTEGER	
BIGINT/INT64	BIGINT	
BOOLEAN	TINYINT	
DOUBLE	DOUBLE	The following value is used to represent `null` in SAP Vora: negative infinity value.
REAL	REAL	
DECIMAL	DECIMAL	Maximum precision in SAP Vora: 18 digits
DATE	DAYDATE	Ancient date values are not currently supported, for example, earlier than the year 1500.
TIME	SECONDTIME	SAP Vora has split seconds, SAP HANA only has full seconds. On SELECT, split seconds are cut off or rounded down. When using the TIME column in predicates, use for example >= 12:00:00 AND < 12:00:01 to avoid incomplete query results due to comparing split-second with full-second values.
TIMESTAMP	LONGDATE	Ancient date values are currently not supported (see above).

Related Information

Handling VARCHAR Length [page 160]

8.10.5 Handling VARCHAR Length

The `tableSchema` and `maxvarcharlength` options provide ways in which you can explicitly manage the VARCHAR length of tables in SAP Vora.

- `tableSchema`

You can use the `tableSchema` option to specify column names and SQL types. Otherwise, SAP Vora ignores VARCHAR lengths and uses VARCHAR(*), which is 2bn, so SAP HANA will not be able to read the table due to its VARCHAR 5000 character limit. For example:

Sample Code

```
CREATE TABLE example_table (col1 VARCHAR(5000), col2 VARCHAR(1000), col3
integer, col4 double)
USING com.sap.spark.engines.disk
OPTIONS (
files "/example/example.csv",csvdelimiter ",",
format "csv",
tableName "CLICKS_CLEANSED_DE",
tableSchema "col1 VARCHAR(5000), col2 VARCHAR(1000), col3 integer, col4
double",
storagebackend "hdfs"
)
```

Note that the column specifications in the Spark line can be omitted when they are specified in the `tableSchema` option.

- `maxvarcharlength`
  The `maxvarcharlength` option allows you to specify the maximum VARCHAR length to be used for all Spark-to-Vora STRING mappings during table creation. For example:

Sample Code

```
CREATE TABLE X (col1 String, ...) USING ... options (maxvarcharlength
"5000")
```

Related Information

Data Type Restrictions [page 159]
Disk Engine and Relational Engine Data Source API [page 53]
Supported Data Types [page 47]

8.10.6 SQL Queries

The following types of SQL queries are supported through the `voraodbc` adapter.

- SELECT queries
- Queries based on the REMOTE_EXECUTE_QUERY function
- Joins between SAP HANA and SAP Vora tables

In general, you should avoid SQL queries with excessively large result sets, for example, SELECT * without any WHERE conditions on a table with 10^7 rows. This applies equally when you execute SQL queries directly in SAP Vora.

In addition, it's a good idea to close sessions frequently and open a new session. Query results that are not completely fetched by the SAP HANA client are not automatically freed by a timeout, so you could have a resource leak when a session is left open for a long time.

8.10.6.1 Executing Remote Queries

You can use the REMOTE_EXECUTE_QUERY function to pass a query directly to a `voraodbc` remote source for execution. This allows SAP HANA to push the query down to the SAP Vora data source without parsing it first.

By passing unparsed strings, SAP HANA allows you to use SQL constructs in SAP Vora which are not supported by the SAP HANA version set up to work with SAP Vora.

The following example shows how REMOTE_EXECUTE_QUERY can be used to perform a selection on MyTable on the remote source `remote_vora`:

Sample Code

```
SELECT X.A FROM REMOTE_EXECUTE_QUERY ('remote_vora', 'select A from MyTable
where B = ?', COLUMNS ('A' integer)) as X;
```

- `remote_vora` is the name of the remote source and is given in single quotes.
- The SQL string is also enclosed in single quotes. It can contain Vora-specific SQL that is not supported by SAP HANA.
- COLUMNS defines an integer column called A, which is the same column returned by the SQL string. The number and names of the columns defined in the COLUMNS clause must exactly match the number and names of the columns returned by the SQL string.

Note

- REMOTE_EXECUTE_QUERY only supports SELECT statements.
- REMOTE_EXECUTE_QUERY is particularly useful for document store and time series tables, which are exposed as virtual tables.
- Remote queries cannot be pushed down to the graph engine.
- The REMOTE EXECUTE privilege is required to use this function.
- This function is supported as of SAP HANA 2.0.

8.10.6.2 Reroute Stored Procedures

You can reroute the execution of simple stored procedures from SAP HANA to SAP Vora. In order to do so, the stored procedure must be defined in both SAP HANA and SAP Vora.

Prerequisites

A remote source connection has been created using the `voraodbc` adapter. See Create an SAP Vora Remote Source [page 156].

Procedure

1. Create the stored procedure in both SAP HANA and SAP Vora as follows:
   a. SAP HANA:

   ```
   CREATE PROCEDURE ( [ <ParameterMode><ParameterIdentifier><ParameterType>
   {, <ParameterMode><ParameterIdentifier><ParameterType> } ] ))
   READS SQL DATA AS BEGIN <Statement>; END;
   ```

   b. SAP Vora:

   ```
   CREATE PROCEDURE ( [ <ParameterMode><ParameterIdentifier><ParameterType>
   {, <ParameterMode><ParameterIdentifier><ParameterType> } ] ))
   AS BEGIN <Statement>; END;
   ```

   Note the following:
   - <ParameterMode>: IN (only IN parameters are currently supported)
   - <ParameterType>: SQL parameter type. Only primitive types are currently supported (not dates, timestamps, or blobs). For example:
     - CHAR
     - VARCHAR
     - INTEGER
     - REAL
     - DOUBLE
2. Register or deregister a rerouting from SAP HANA to the SAP Vora remote source as follows:

Option	Steps
Register a rerouting	`alter procedure <ProcedureName>` `add route to remote source <VoraRemoteSourceName>;`

Deregister a rerouting	`alter procedure <ProcedureName>`
	`drop route to remote source`
	`<VoraRemoteSourceName>;`

3. Optionally check which routes are registered in SAP HANA as follows:

```
select * from PROCEDURE_ROUTES;
```

4. Call a rerouted procedure as follows:

```
call <ProcedureName>(<Parameters>);
```

9 Using Hierarchies

Hierarchical data structures define a parent-child relationship between different data items, providing an abstraction that makes it possible to perform complex computations on different levels of data.

An organization, for example, is basically a hierarchy where the connections between nodes (for example, manager and employee) are determined by the reporting lines that are defined by that organization.

Since it is very difficult to use standard SQL to work with and perform analysis on hierarchical data, Spark SQL has been enhanced to provide missing hierarchy functionality. Extensions to Spark SQL support hierarchical queries that make it possible to define a hierarchical DataFrame and perform custom hierarchical UDFs on it. This allows you, for example, to define an organization's hierarchy and perform complex aggregations, such as calculating the average age of all second-level managers or the aggregate salaries of different departments.

Since the SAP Vora execution engine supports hierarchies, hierarchical queries can be pushed down to SAP Vora using the data source implementation. Level-based hierarchies, however, are currently only supported in Spark and cannot be pushed down to SAP Vora.

> **Note**
>
> To calculate a hierarchy binary UDF, the Spark implementation requires a cross join to be evaluated, which can be very expensive in the case of large hierarchy tables.

Related Information

Creating Adjacency-List Hierarchies [page 165]
Creating Level-Based Hierarchies [page 169]
Hierarchy Example (h_src) [page 167]
Using Hierarchies with Views [page 170]
Hierarchy UDFs [page 171]

9.1 Creating Adjacency-List Hierarchies

You can build a hierarchy using an adjacency list that defines the edges between hierarchy nodes. The adjacency list is read from a source table where each row of the source table becomes a node in the hierarchy.

The hierarchy SQL syntax allows you to define the adjacency list and tweak it. It also allows you to determine how the hierarchy is created by controlling the order of the children of each node and by explicitly determining the roots of the hierarchy.

For example, you have a table called `h_src` that includes the two columns `pred` and `succ`, showing predecessors and successors respectively. You can create and query the hierarchy using the following statement:

Sample Code

```
SELECT position, IS_ROOT(node) FROM HIERARCHY (
USING h_src AS v
JOIN PRIOR u ON v.pred = u.succ
ORDER SIBLINGS BY ord ASC
START WHERE pred IS NULL
SET node
) AS H
```

The clauses in the statement are used as follows:

- `JOIN PRIOR <alias> <equality_expression>`
  Defines how the adjacency list is constructed. In this example it states that any two rows in `h_src` have an edge between them in the hierarchy if the child row's `pred` column is equal to the parent row's `succ` column. This clause is mandatory.
- `ORDER SIBLINGS BY <order_by_expression>`
  Determines the order of the children when the hierarchy is being constructed. The order is relevant for some UDFs, such as `IS_PRECEDING` and `IS_FOLLOWING`, and can be defined by the user. This clause is optional. If it is not defined and UDFs are used that depend on the order of the children, the results returned are undefined.
- `START WHERE <condition>`
  Defines the roots of the hierarchy. Any row that matches the specified condition is considered a root in the hierarchy forest. This clause is also optional. If omitted, the roots of the hierarchy are determined by scanning all source table rows and identifying those that do not have a parent.
- `SET <node_column_name>`
  Names the newly created node column. The new hierarchy relation is created with the same schema as that of `h_src` but also contains the additional node column (in the query above it is called `node`), which is internal and used for hierarchy operations. For each row in the hierarchy relation, this column contains the information necessary to specify its location in the hierarchy. Note that the column must not be used in a top-level query unless it is inside a hierarchy UDF.

Related Information

Hierarchy Example (h_src) [page 167]
Hierarchy UDFs [page 171]

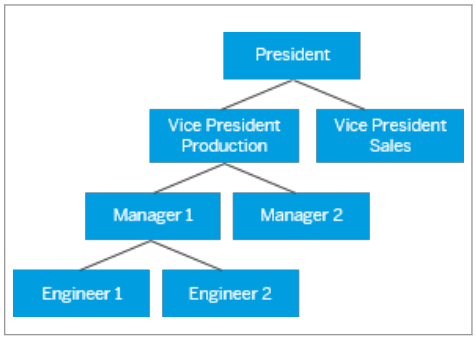
9.2 Hierarchy Example (h_src)

In this example, the table `h_src` represents a simple organizational hierarchy.

Table h_src

The table `h_src` below includes the two columns `pred` and `succ`, which show the predecessors and successors in the hierarchy. Based on equality between `pred` and `succ`, the table `h_src` represents the hierarchy shown graphically on the right:

position	pred	succ	ord
President	None	1	1
Vice President Production	1	2	1
Vice President Sales	1	3	2
Manager 1	2	4	1
Manager 2	2	5	2
Engineer 1	4	6	1
Engineer 2	4	7	2



Employees Table

A table called `employees` contains the names of the employees in the organization:

name	position
Denise Smith	Manager 3
Donna Moore	Manager 1
John Miller	Engineer 1
Julie Armstrong	President
Michael Adams	Vice President Production

Query Examples

A hierarchy table is a regular SQL table that can be used just like any other table, for example, as shown below.

Inner Join

You can get the positions, names, and levels of the employees in the tree as follows:

Sample Code

```
SELECT B.position, A.name, B.level
FROM
(SELECT position, LEVEL(node) AS level FROM HIERARCHY (
USING h_src AS v
JOIN PRIOR u ON v.pred = u.succ
ORDER SIBLINGS BY ord ASC
START WHERE pred IS NULL
SET node)
AS H) B, employees A
WHERE B.position = A.position
```

Result:

position	name	level
President	Julie Armstrong	1
Vice President Production	Michael Adams	2
Manager 1	Donna Moore	3
Engineer 1	John Miller	4

Left Outer Join

You can do a left outer join to get all positions in the organization, including those not currently occupied:

Sample Code

```
SELECT A.position, B.name, A.level
FROM
SELECT position, LEVEL(node) AS level FROM HIERARCHY (
USING h_src AS v
JOIN PRIOR u ON v.pred = u.succ
ORDER SIBLINGS BY ord ASC
START WHERE pred IS NULL
SET node)
AS H) A LEFT OUTER JOIN employees B
ON A.position = B.position
```

Result:

position	name	level
President	Julie Armstrong	1
Vice President Production	Michael Adams	2
Vice President Sales	null	2
Manager 1	Donna Moore	3
Manager 2	null	3
Engineer 1	null	4

position	name	level
Engineer 2	John Miller	4

Related Information

Creating Adjacency-List Hierarchies [page 165]
Creating Level-Based Hierarchies [page 169]

9.3 Creating Level-Based Hierarchies

An alternative way of creating a hierarchy is by mapping hierarchy levels to source table columns. This type of hierarchy creation is particularly useful when the source table is a flattened hierarchy where all hierarchy paths are encoded as rows.

For example, you have a source table with the following schema:

Sample Code

```
h_src(col1, col2, col3, col4)
```

`col1` represents the first level, `col2` the second level, and so on. Using this table, you can create a level-based hierarchy as follows:

Sample Code

```
SELECT position, IS_ROOT(node) FROM HIERARCHY (
USING h_src WITH LEVELS (col1, col2, col3, col4)
MATCH PATH
ORDER SIBLINGS BY ord ASC
SET node) AS H) AS H
```

The clauses in the statement are used as follows:

- `WITH LEVELS (col1, col2, col3, col4)`
  Determines the levels of the hierarchy. In this example it means that `col1` is used for the first level, `col2` for the second level, and so on.
- `MATCH PATH`
  Determines the way identical nodes are handled across the same and across different columns.
- `ORDER SIBLINGS BY`
  This clause has the same semantics as in adjacency-list hierarchies.
- `SET`
  This clause also has the same semantics as in adjacency-list hierarchies.

When a level-based hierarchy is constructed, all paths of the hierarchy are enumerated as rows, which can potentially result in a much larger number of rows in the hierarchy table than in the source table. In the worst case this could be C*R, where C is the number of columns and R the number of rows in the source table.

Example

A source table has three columns, `col1`, `col2`, and `col3`, and one row containing `value1`, `value2`, and `value3`. The hierarchy table will have three rows enumerating all paths starting with `value1`:

```
value1, null, null
value1, value2, null
value1, value2, value3
```

Note

Although you can work with the resulting hierarchy as with adjacency-list hierarchies, bear in mind that level-based hierarchies are currently only supported in Spark and cannot be pushed down to SAP Vora.

Related Information

Creating Adjacency-List Hierarchies [page 165]

9.4 Using Hierarchies with Views

Views make is easier to work with hierarchies, in particular, when using self-joins to calculate binary predicates, such IS_PARENT and IS_FOLLOWING.

To create a view of a hierarchy that uses the table `h_src`, for example, you can issue the following SQL statement:

Sample Code

```
CREATE VIEW HV AS SELECT * FROM HIERARCHY (
USING h_src AS v
JOIN PRIOR u ON v.pred = u.succ
ORDER SIBLINGS BY ord ASC
START WHERE pred IS NULL
SET Node)
```

The above command creates a view named HV that wraps a hierarchy. From now on, the view name can be used in an SQL query and will be replaced with the underlying hierarchy statement.

In the following example, the hierarchy view is joined to itself in order to get the positions of the children of the root:

Sample Code

```
SELECT Children.position
FROM HV Children, HV Parents
WHERE IS_ROOT(Parents.Node) AND IS_PARENT(Parents.Node, Children.Node)
```

To select the names of all the descendants of the second-level employees, you need to do a two-level join:

- The inner join calculates the descendants of the second-level employees.
- The outer join joins the result to the `employees` table and gathers the positions and names.

Sample Code

```
SELECT A.position, B.name
FROM
  (SELECT Descendants.position AS position
   FROM HV Parents, HV Descendants
   WHERE IS_DESCENDANT(Descendants.Node, Parents.Node) AND LEVEL(Parents.Node)
= 2
  ) A, employees B
WHERE A.position = B.position
```

Related Information

Hierarchy Example (h_src) [page 167]

9.5 Hierarchy UDFs

This list shows the user-defined functions (UDFs) that can be used with hierarchies.

UDF	Description
LEVEL(u)	Returns the level of the node in the tree
IS_ROOT(u)	True if the node is a root, otherwise false
IS_DESCENDANT(u,v)	True if node u is a descendant of node v
IS_DESCENDANT_OR_SELF(u,v)	Node u equals node v or is_descendant(u,v)
IS_ANCESTOR(u,v)	True if node u is an ancestor of node v
IS_ANCESTOR_OR_SELF(u,v)	Node u equals node v or is_ancestor(u,v)
IS_PARENT(u,v)	Node u is a parent of node v
IS_CHILD(u,v)	Node u is a child of node v
IS_SIBLING(u,v)	Node u is a sibling of node v
IS_FOLLOWING(u,v)	Node u follows node v in preorder and is not a descendant of v

UDF	Description
IS_PRECEDING(u,v)	Node u precedes node v in preorder and is not a descendant of v
NODE(node)	Returns the textual name of the node

10 Currency Conversion

SAP Vora supports two forms of currency conversion: standard currency conversion and ERP currency conversion.

- Standard currency conversion: This conversion method uses a single rates table to perform the conversion and provides a standard set of options to adapt the conversion process. It is available in the open source extension library.
- ERP currency conversion: This conversion method is compatible with the ERP conversion method implemented in SAP HANA. In addition to a rates table, this conversion method uses a rates prefactor table, a currency precision table, and a client settings table, to adapt the conversion process to different applications as used in SAP BW and SAP ERP.

Related Information

Standard Currency Conversion [page 173]
ERP Currency Conversion [page 176]

10.1 Standard Currency Conversion

The standard currency conversion is based on a given rates table.

The schema of the rates table must be as follows:

```
( SOURCE_CUR string,
  TARGET_CUR string,
  REF_DATE string,
  RATE double )
```

The names of the columns can be arbitrary. Their semantics are:

SOURCE_CUR	The source currency as a string identifier (for example, USD)
TARGET_CUR	The target currency as a string identifier (for example, EUR)
REF_DATE	A string in the YYYY-MM-DD format
RATE	The conversion rate in arbitrary precision (normally 5 digits)

A rates table might look like this:

SOURCE_CUR	TARGET_CUR	REF_DATE	RATE
EUR	USD	2015-01-01	1.32113

SOURCE_CUR	TARGET_CUR	REF_DATE	RATE
EUR	USD	2015-01-02	1.30121
EUR	USD	2015-01-03	1.30121
EUR	USD	2015-01-04	1.31763
USD	EUR	2015-01-01	0.89464
USD	EUR	2015-02-01	0.86297
USD	GBP	2015-01-01	0.68960
USD	GBP	2015-02-01	0.70544

The following `orders` table is used in the examples:

TID	USERID	CURRENCY	AMOUNT	ORDERDATE
100	user1	USD	120.10	2014-12-15
101	user1	USD	24.99	2015-01-01
102	user5	EUR	24.11	2015-01-02
103	user3	GBP	542.00	2015-01-02
104	user5	EUR	11.99	2015-01-03
...	...	...	...	...

Conversion Function

The standard currency conversion function is defined as follows:

```
CC( AMOUNT Double,
    SOURCE_CURRENCY String,
    TARGET_CURRENCY String,
    REF_DATE String )
```

Given the `orders` table above, the conversion function can be used as follows:

Sample Code

```
SELECT TID,
       USERID,
       ORDERDATE,
       CC( AMOUNT, CURRENCY, "USD", ORDERDATE )
FROM ORDERS
```

For all arguments of the function, the values can either come from a column or can be set explicitly in the statement. For example, the conversion for a fixed reference date would be:

Sample Code

```
SELECT TID,
       USERID,
```

```
        ORDERDATE,
        CC( AMOUNT, CURRENCY, "USD", "2015-01-01" )
FROM ORDERS
```

Conversion Method

The standard conversion method is defined as follows:

```
CC(AMOUNT, SOURCE_CURRENCY, TARGET_CURRENCY, REF_DATE) :=
   AMOUNT * RATE_LOOKUP(SOURCE_CURRENCY, TARGET_CURRENCY, REF_DATE)
RATE_LOOKUP(SOURCE_CURRENCY, TARGET_CURRENCY, DATE) :=
   if REF_DATE <= DATE exists:
     The RATE value where SOURCE_CURRENCY and TARGET_CURRENCY
     match and (TIME - REF_TIME) is the minimum
   else: NULL
```

For conversions with missing rates, the result is either NULL or the method throws an exception (based on the `error_handling` option).

Example

For `TID = 102` in the `orders` table, the conversion to `USD` based on the rates table would be: `24.11 * 1.28042 = 30.8709262`.

Example

For `TID = 100` in the `orders` table, the conversion to `USD` would fail or be NULL (depending on the currency options) since there is no valid entry in the rates table.

Related Information

Standard Currency Conversion Options [page 175]

10.2 Standard Currency Conversion Options

The standard conversion function can be adapted by setting options using the `spark.sql.currency.basic.*` prefix.

You can set an option:

- In the `spark-defaults.conf` file
- During startup of Spark by setting the configuration parameters on the command line

- Within a SparkSQL session by using the SET statement, for example:

```
SET spark.sql.currency.basic.OPTIONNAME = OPTIONVALUE;
```

For example, you can set the rates table as follows:

Sample Code

```
SET spark.sql.currency.basic.table = new_rates_table;
```

The following options are available:

Name	Description	Default Value	Example
table	Name of a Spark SQL table with the conversion rates	RATES	CONVERSION_RATES
allow_inverse	If inverse lookup is allowed, the conversion will try to do a regular rate lookup first. If a matching rate cannot be found, it tries to lookup the inverse rate (SOURCE and TARGET switched during lookup) and performs the conversion using AMOUNT / RATE.	false	true
do_update	This option triggers an update of the currency function before the next method call. By doing this, new values in the rates table are considered in the conversion. The option is automatically set to false after the update has been performed.	false	true
error_handling	Specifies how the function should behave if a conversion rate could not be found: • fail_on_error: Aborts the Spark job and throws an exception • set_to_null: Returns NULL for that row • keep_unconverted: Uses the input amount	fail_on_error	set_to_null

10.3 ERP Currency Conversion

The ERP currency conversion is based on a set of rates, currency, and configuration tables. They have a standard format in the ERP system and are managed by particular tools.

The ERP currency conversion logic is able to consume these ERP tables and perform a currency conversion that is compatible with the ERP conversion in SAP HANA.

The ERP table schemas are:

TCURX: Currency precision

Column	Type	Description	Example
CURRKEY	String	Alphabetic currency code (ISO 4217)	USD, EUR, NOK
CURRDEC	Int	Number of digits after decimal point (ISO 4217)	2 (for EUR), 0, 3

TCURV: Conversion settings

Column	Type	Description	Example
MANDT	String	Client	'001', '002'
KURST	String	Effective conversion type	'M', 'EUREX'
XINVR	String	Calculation allowed with inverted exchange rate	'0', '1'
BWAER	String	Reference currency for currency translation	
XBWRL	String	Base currency is "from" currency in the exchange rate table	'0', '1'
GKUZU	String	Exchange rate type of average rate used to determine buying rate	
BKUZU	String	Exchange rate type of average rate used to determine selling rate	
XFIXD	String	Exchange rate type uses fixed exchange rates	'0', '1'
XEURO	String	Exchange rate type uses special translation model	'0', '1'

TCURF: Currency factors

Column	Type	Description	Example
MANDT	String	Client ID	'000', '001', ...
KURST	String	Exchange rate type	'M', 'EURX'
FCURR	String	Source currency	'EUR'
TCURR	String	Target currency	'USD'
GDATU	String	SAP legacy date (99999999 - YYYYMMDD)	'79839875' (for 2016/01/24)
FFACT	Decimal	Source factor	1, 20
TFACT	Decimal	Target factor	1, 20
ABWCT	String	Alternative (exchange rate type)	'M', 'EURX'
ABWGA	String	Date from which ABWCT is valid	'79839875' (for 2016/01/24)

TCURR: Currency rates

Column	Type	Description	Example
MANDT	String	Client	'001'

Column	Type	Description	Example
KURST	String	Conversion type	'M', 'EURX'
FCURR	String	Source currency	'EUR'
TCURR	String	Target currency	'USD'
GDATU	String	Date (legacy format)	'79839875' (for 2016/01/24)
UKURS	Decimal	Exchange rate. Note that a negative rate always means that the inverse of the rate has to be taken, so if `R < 0` then `R := -1/R` instead.	1234.12345, 0001.41500, -0021.33212
FFACT	Decimal	Source factor	
TFACT	Decimal	Target factor	

To use the EPR currency conversion, the tables needs be be available as Spark SQL tables. By default, the function expects the tables with names TCURX, TCURV, TCURF, TCURR to exist.

Conversion Method

The method is defined as follows:

```
CONVERT_CURRENCY( CLIENT String
                  CONVERSION_TYPE String,
                  AMOUNT Decimal,
                  SOURCE_CURRENCY String,
                  TARGET_CURRENCY String,
                  REF_DATE String )
```

The semantics of the arguments are:

CLIENT	The client ID for which the conversion settings have been defined in the ERP tables
CONVERSION_TYPE	A conversion type that has been defined in the ERP tables. A conversion type of 'M' represents a default regular lookup. Other types can be defined in the tables. For more information, see the `Options` section.
AMOUNT	The amount to be converted
SOURCE_CURRENCY	The currency of the given amount
TARGET_CURRENCY	The currency to convert to
REF_DATE	The date on which the conversion should take place

Given a table `FINANCE` with client, amount, currency, and date columns, the function can be called as follows:

Sample Code

```
SELECT client,
       date,
       CONVERT_CURRENCY( client, 'M', amount, currency, 'USD', date )
FROM FINANCE
```

Conversion Steps

The currency conversion is based on the following steps:

```
(1) Shift input AMOUNT according to settings in TCURX
(2) Get conversion settings according to (CLIENT, CONVERSION_TYPE,
SOURCE_CURRENCY, TARGET_CURRENCY)
(3) Perform the conversion based on the retrieved settings:
    - regular: Look up the rate in ``TCURR`` and do the conversion or NULL if no
rate found
    - allow_inverse: Do the conversion as with regular. If NULL, reverse the
conversion and apply the inverse rate
    - base_currency: Use a base currency BASE and to the transformation by
(SOURCE -> BASE) and (TARGET -> BASE)
(4) Round the result according to the currency settings in ``TCURX``
```

The steps can be set by the steps option, as described in the `Options` section.

Related Information

ERP Currency Conversion Options [page 179]

10.4 ERP Currency Conversion Options

The ERP conversion function can be adapted by setting the options using the `spark.sql.currency.erp.*` prefix as described below.

You can set an option:

- In the `spark-defaults.conf` file
- During startup of Spark by setting the configuration parameters on the command line
- Within a SparkSQL session by using the SET statement, for example:

  ```
  SET spark.sql.currency.erp.OPTIONNAME = OPTIONVALUE;
  ```

  For example, you can enable inverse lookups as follows:

  Sample Code

  ```
  SET spark.sql.currency.erp.allow_inverse = true;
  ```

The following options are available:

Name	Description	Default Value	Example
table_prefix	A prefix for ERP tables. For example, a value of A01_ will look for the table set in tcurx by A01_TCURX. This makes it easier to switch between different sets of ERP tables.	""	A01_
tcurx	Table name of the TCURX table	TCURX	TCURX_V2
tcurv	Table name of the TCURV table	TCURV	TCURV_V2
tcurf	Table name of the TCURF table	TCURF	TCURF_V2
tcurr	Table name of the TCURR table	TCURR	TCURR_V2
lookup	Can be regular or reverse. If the lookup type is reverse, the conversion will try to do a regular rate lookup first. If a matching rate cannot be found, it tries to look up the reverse rate (SOURCE and TARGET switched during lookup) and performs the conversion using AMOUNT / RATE.	regular	reverse
do_update	This option triggers an update of currency function before the next method call. By doing this, new values in the ERP tables are considered in the conversion. The option is automatically set to false after the update has been performed.	false	true
date_format	Specifies how the date string should be interpreted: • normal: String in the YYYY-MM-DD or YYYYMMDD format • inverted: String in the format string(99999999 - int(YYYYMMDD)) • auto_detect: Tries to auto-detect the format	auto_detect	inverted
steps	The conversion process contains the steps (1) shift, (2) convert, (3) round. This option specifies how far the result should be computed (mainly used to enable rounding and for debugging).	shift,convert	shift
error_handling	Specifies how the function should behave if a conversion rate could not be found: • fail_on_error: Aborts the Spark job and throws an exception • set_to_null: Returns NULL for that row • keep_unconverted: Uses the input amount	fail_on_error	set_to_null

SAP HANA Data Source Support

The ERP conversion function is designed to be API compatible with the SAP HANA SQL function CONVERT_CURRENCY and can therefore be pushed down when SAP HANA is used as a data source. Since there is no notion of SCHEMA in Spark SQL, the schema holding the rates tables needs to be set by an SQL option as follows:

```
SET spark.sql.currency.erp.schema = rates_table_schema;
```

All other options can also be used with the SAP HANA data source.

11 System Tables

System tables provide information about the SAP Vora system, such as the existing tables in the system. System tables have a hard-coded schema and can be used to programmatically get the schema of other tables.

The system tables currently available reflect the data in the database catalog. They also reflect updates whenever there is a change to the database catalog. For example, if the user creates a new table, the system tables will immediately show this new table.

The contents of the system tables are versioned and directly correspond to a version of the database catalog. Therefore, concurrent queries of system tables might get different results if there is a concurrent modification of the database catalog.

Note that there needs to be at least one relational in-memory engine in the cluster to execute queries on system tables.

Related Information

Available System Tables [page 182]
Restrictions [page 184]

11.1 Available System Tables

The following system tables are currently available.

SYS.SCHEMAS

SYS.SCHEMAS contains information about all schemas in the current database. Note that schemas that do not contain any tables are listed in the SYS.SCHEMAS table, but not in the SYS.TABLES table.

SYS.SCHEMAS has the following columns:

Name	Type	Description
SCHEMA_NAME	varchar(256)	Name of the schema
SCHEMA_OID	bigint	Object ID of the schema

SYS.TABLES

SYS.TABLES contains information about all tables in the current database. It has the following columns:

Name	Type	Description
SCHEMA_NAME	varchar(256)	Name of the schema
TABLE_NAME	varchar(256)	Name of the table
TABLE_TYPE	varchar(16)	Table type: GRAPH, COLLECTION, TIMESERIES, IN-MEMORY, or DISK

SYS.TABLE_COLUMNS

SYS.TABLE_COLUMNS contains information about all columns of all tables in the current database. It has the following columns:

Name	Type	Description
SCHEMA_NAME	varchar(256)	Name of the schema
TABLE_NAME	varchar(256)	Name of the table
COLUMN_NAME	varchar(256)	Name of the column
DATA_TYPE_NAME	varchar(256)	String-representation of the SQL type of the column

SYS.VIEWS

SYS.VIEWS contains information about views. It has the following columns:

Name	Type	Description
SCHEMA_NAME	varchar(256)	Name of the schema
VIEW_NAME	varchar(256)	Name of the view
DEFINITION	varchar(256)	View definition string

SYS.SYNONYMS

SYS.SYNONYMS contains information about synonyms. It has the following columns:

Name	Type	Description
SCHEMA_NAME	varchar(256)	Name of the schema

Name	Type	Description
SYNONYM_NAME	varchar(256)	Name of the synonym
OBJECT_SCHEMA	varchar(256)	Schema name of the referenced object
OBJECT_NAME	varchar(256)	Name of the referenced object
OBJECT_TYPE	varchar(32)	Type of the referenced object
IS_VALID	varchar(5)	Specifies whether the synonym is valid or not

11.2 Restrictions

System tables reside in the SYS schema of each database and reflect only the information about that database. Because of forward compatibility, modification of the SYS schema by the user, for example, by adding new tables, is prohibited. Similarly, modification of the contents of system tables is prohibited.

These prohibitions are currently not enforced. Breaking them, that is, attempting to modify the SYS schema or any of the objects in the SYS schema results in undefined behavior.

In future version of SAP Vora, the schema of the system tables might change. We recommend that your use of system tables does not rely on the number of columns in system tables, nor the number of tables in the SYS schema.

In any case, the SYS schema should be considered reserved; you should not attempt to modify it even in the absence of system tables.

12 Working with the SAP Vora Tools

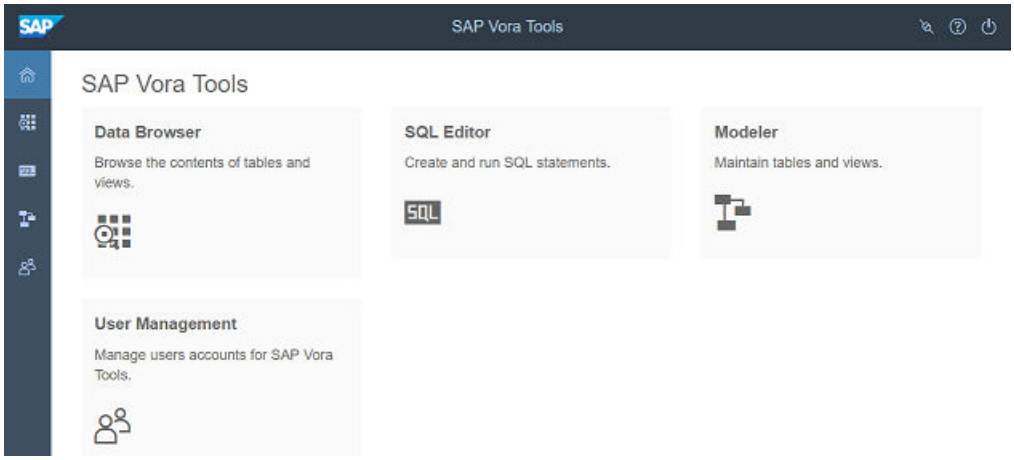
The SAP Vora Tools provide a data browser for viewing and exporting data in tables and views, an SQL editor for creating and running SQL scripts, and a modeler for creating data models.

Prerequisites

The SAP Vora Thriftserver and SAP Vora Tools services are up and running.

Initial Screen

The initial screen of the SAP Vora Tools shows the four main work areas: the data browser, SQL editor, modeler, and user management tool. Each tool can also be accessed from the panel on the left:



Related Information

Show and Export Data using the Data Browser [page 186]
Execute SQL Scripts Using the SQL Editor [page 186]
SQL Editor Keyboard Shortcuts [page 188]
Data Modeling in SAP Vora [page 190]

12.1 Show and Export Data using the Data Browser

The data browser allows you to quickly display data in tables and views in SAP Vora without having to write a query.

Procedure

1. In the data browser, select a table or view from the navigation pane.
   The first 1000 rows are displayed in a tabular data view.
2. Filter or sort the data displayed.
3. Hide columns using the *Settings* menu.
4. Export the data displayed as a CSV file.

12.2 Execute SQL Scripts Using the SQL Editor

You can use the SQL editor to execute a single SQL statement or a sequence of statements in SAP Vora. The editor supports syntax highlighting, code completion, and syntax checks as you write your SQL code.

Context

Note the following:

- You can execute SQL only. It is not possible to execute Scala code from the editor.
- When you use the SQL editor to create a table that loads data from HDFS, you need to set the parameter `hdfsnamenode`. For example:

  Sample Code

  ```
  CREATE TABLE table1(col1 int)
  USING com.sap.spark.engines.relational
  OPTIONS (hdfsnamenode "namenode.example.com:8020", files "/path/to/
  file1.csv", storagebackend "hdfs")
  ```

- The editor content is saved in the local storage of the browser and therefore cannot be shared between different browsers.
- The content is lost if you delete the local storage of your browser.

Procedure

- **Execute a single statement**
  a. In the SQL editor, place the cursor inside a statement.
  b. Choose *Execute Statement*.
  The output of the execution is shown. If the statement returns data, the result set is displayed in the result window.
  c. You can filter the data displayed or export it as CSV.
- **Execute multiple statements**
  a. Select the statements to be executed in the editor. Multiple statements are delimited by a semicolon.
  b. Choose *Execute Statement*.
  The output of the execution is shown. The result set of the last statement is displayed in the result window.
  c. Select the output of another statement to show its result set.
- **Execute all statements**
  a. Choose *Execute All*.
  The output of the execution is shown. The result set of the last statement is displayed in the result window.
  b. Select the output of another statement to show its result set.
- **Exclude code from execution** by enclosing it with a comment:

Sample Code

```
-- SELECT * FROM v; single line comment, will not be executed
```

Sample Code

```
/* SELECT * FROM v;
   SELECT * FROM t; multiline comment, will not be executed */
```

- **Apply code completion**

  Code completion assists you when writing SQL code. It offers code snippets for common SQL statements, completions for keywords, locally used words, database tables, views, and columns.

  Trigger code completion by pressing `CTRL` + `SPACEBAR` or by typing a dot in a qualified column name.
- **Show token tooltips**

  The token tooltip displays metadata about database tables, views, and columns.

  Hover the mouse over a identifier or query to show the tooltip.

Related Information

SQL Editor Keyboard Shortcuts [page 188]

12.3 SQL Editor Keyboard Shortcuts

The following common shortcut key codes are available in the SQL editor.

Windows/Linux	Mac	Action
Execution		
CTRL + R	Command + R	Execute statement/Execute selection
CTRL + SHIFT + R	Command + SHIFT + R	Execute all
Find/Replace		
CTRL + F	Command + F	Find
CTRL + H	Command + Option + F	Replace
CTRL + K	Command + G	Find next
CTRL + SHIFT + K	Command + SHIFT + G	Find previous
Other		
CTRL + SPACEBAR	Command + SPACEBAR	Start code completion
CTRL + L	Command + L	Go to line
TAB	TAB	Indent
SHIFT + TAB	SHIFT + TAB	Outdent
CTRL + Z	Command + Z	Undo
CTRL + SHIFT + Z, CTRL + Y	Command + SHIFT + Z, Command + Y	Redo
CTRL + / CTRL + # (German keyboard)	Command + /	Toggle line comment
CTRL + SHIFT + / CTRL + SHIFT + # (German keyboard)	Command + /	Toggle block comment
CTRL + SHIFT + U	CTRL + SHIFT + U	Change to lower case
CTRL + U	CTRL + U	Change to upper case
CTRL + A	Command + A	Select all
Line Operations		
CTRL + D	Command + D	Remove line
ALT + SHIFT + DOWN ARROW	Command + Option + DOWN ARROW	Copy lines down
ALT + SHIFT + UP ARROW	Command + Option + UP ARROW	Copy lines up
ALT + DOWN ARROW	Option + DOWN ARROW	Move lines down
ALT + UP ARROW	Option + UP ARROW	Move lines up
ALT + DEL	CTRL + K	Remove to line end

Windows/Linux	Mac	Action
ALT + BACKSPACE	Command + BACKSPACE	Remove to line start
CTRL + BACKSPACE	Option + BACKSPACE, CTRL + Option + BACKSPACE	Remove word left
CTRL + DEL	Option + DEL	Remove word right

13 Data Modeling in SAP Vora

Data modeling refers to an activity of refining or slicing data in database tables by creating views that depict a business scenario. You can query these views and use the query result for any reporting and decision-making purposes.

The SAP Vora data modeler provides capabilities that helps you to create and edit views.

The SAP Vora data modeler tool supports the following types of views:

- SQL Views
- Dimensions
- Cubes

Who Should Read this Guide

This guide is intended for data modelers, business or data analysts, application developers who are building database models, or database experts who are involved in defining data models, views, primary keys, indexes, partitions, and all other aspects of layout and interrelationship of data.

The guide is organized as follows.

Title	Description
Creating Tables in SAP Vora	Explains the steps for creating tables in the SAP Vora relational engine, appending data to existing tables in the SAP Vora relational in-memory engine, and creating tables using objects from SAP HANA.
Working with other SAP Vora Engines	Explains the steps for creating tables for time series data, tables on the disk, creating graphs, and creating collections.
Create Partition Schemes	Explains the steps for creating partition schemes using SAP Vora Modeling tool.
Creating Views in SAP Vora	Explains the steps for creating views, defining SQL clauses, creating view with star joins, creating views with collections, and creating views with graphs.
Preview Output of Views	Explains the steps for previewing output of views created in the SAP Vora modeler tool.
Visualizing and Analyzing the Data	Explains the steps for using the SAP Vora data browser tool to visualize and analyze the data in tables and graphs.
Additional Functionality for Views	Explains the steps for creating parent-child hierarchies in views, creating level-based hierarchies in views, creating table functions for time series, creating calculated columns, using calculated columns for currency conversions, and other additional functions for views.

Related Information

Create Views [page 210]
Creating Tables in SAP Vora [page 191]
Defining SQL Clauses [page 212]
Working with Other SAP Vora Engines [page 196]
Preview Output of Views [page 224]
Additional Functionality for Views [page 231]
Create Partition Schemes [page 208]
Visualizing and Analyzing the Data [page 226]

13.1 Creating Tables in SAP Vora

Use the graphical data modeling tools in SAP Vora to create tables that you can then use, for example, as data sources when modeling views.

You can also register tables from SAP HANA system as data sources in the SAP Vora relational in-memory engine.

You can create tables in different SAP Vora engines. For example, you can create time series tables in the SAP Vora time series engine, and create tables on disk using the SAP Vora disk engine. After creating the tables in any of the SAP Vora engines, you can use them as data sources when modeling views.

Related Information

Create Tables in the SAP Vora Relational Engine [page 191]
Create Tables Using Objects from SAP HANA [page 195]
Working with Other SAP Vora Engines [page 196]

13.1.1 Create Tables in the SAP Vora Relational Engine

Use the graphical data capabilities in the SAP Vora data modeler tool to create tables in the SAP Vora relational engine. You can then use the tables, for example, as data sources to model views in SAP Vora.

Procedure

1. Start the SAP Vora tool in a Web browser.

2. In the home page, select *Modeler*.
3. In the navigation pane, choose + (Create).
4. Select the *Create Relational Table* menu option.
5. Provide a name for the table.
6. Select *Vora* as the data source type.
7. In the *Engine* dropdown list, select *Relational* as the engine type.
8. (Optional) Provide details for loading data.

   You use files in HDFS or S3 to load data into the table. For loading data into the table, provide the following additional information:

   a. In the *File Type* dropdown list, select the required file type.

      The tool supports CSV, ORC, AVRO, and Parquet file types.

   b. If you have selected the file type as CSV, specify the delimiter that SAP Vora must use to parse the `CSV` files and also a CSV skip value.

      CSV skip value means that the first 'n' (n is the CSV skip value) lines in the `.CSV` file are skipped when loading data.

   c. In the *File System* dropdown list, select a value.

      The tool supports HDFS and S3 files systems.

File System	Description
HDFS	Create tables and load data to the tables from files in HDFS. If you select HDFS as the file system, provide the HDFS NameNode details.
S3	Create tables and load data to the tables from files in S3. If you select S3 as the file system, provide the details such as S3 Access Key ID, S3Secret Access Key, and S3 EndPoint.

   d. In the *File Path*, enter the fully qualified path to the file.

> **Note**
>
> In the *File Path*, you can enter the fully qualified path of more than one file. After entering a fully qualified path, press the return key to enter the full qualified path of another file. The files can be from the same or different file systems.

9. (Optional) Provide file-specific details.

   You can load data from multiple files, and these files can exist in different file systems. For each file, you can provide file specific details.

   a. In the *File Path*, select a file.
   b. Select *Enable File specifc options*.
   c. In the *File System* dropdown list, select the file system in which the file is available.
   d. If file is a .csv file, you can provide CSV-specific details such as CSV skip value, delimiter, columns to load, CSV quote value, and more.

10. Choose *Next*.
11. Define columns:

    a. In the *Name* field, enter the name of the column.
    b. In the *Data Type*, dropdown list, select the required data type.

c. Provide the *Length* and *Scale* depending on the selected data type.
d. (Optional) To create additional columns, choose + (Add Column).

12. (Optional) Define a partition scheme.

    You can use a partition scheme to apply and instantiate a partition function.

    a. In the right pane, select the *Partition Scheme* section.
    b. Choose + (Add).
    c. If you want to use an existing partition scheme, select the *Partition Scheme* and choose *Select*.

    > **Note**
    >
    > For creating tables in the SAP Vora relational engine, you can use partition schemes defined with RANGE or HASH partition function types only.

    d. If want to create and use a new partition scheme, in the *Partition Scheme* dialog box, choose + (Add New Scheme) to create a new partition scheme.
    e. In the *Partition Parameter* text field, enter the required partition parameter.

    When creating a table in the SAP Vora relational engine, instantiate this parameter with a column from the table.

13. (Optional) If you want to provide additional information to configure the table or to load data in the table, in the details pane specify the configuration details.

    For more information about the configuration options, see Disk Engine and Relational Engine Data Source API [page 53].

14. Choose *Finish*.

Related Information

Append Data to Existing Tables in the SAP Vora Relational In-Memory Engine [page 193]
Create Views [page 210]

13.1.1.1 Append Data to Existing Tables in the SAP Vora Relational In-Memory Engine

Use the SAP Vora data modeler tool to append data to existing tables in the SAP Vora relational in-memory engine.

Procedure

1. Start the SAP Vora tool in a Web browser.
2. In the home page, select *Modeler*.
3. From the navigation pane, select the required table in *Vora Relational Engine*.

The tool opens the schema definition of the selected table.

4. In the menu bar, choose  (Append Data Files).
5. (Optional) Provide details for loading data.

   You use files in HDFS and S3 to load data into the table. In the *Append File* dialog box, provide details of the file from which you want to append data.

   a. In the *File Type* dropdown list, select the required file type.

      The tool supports CSV, ORC, AVRO, and Parquet file types.

   b. If you have selected the file type as CSV, specify the delimiter that SAP Vora must use to parse the `CSV` files and also a CSV skip value.

      CSV skip value means that the first 'n' (n is the CSV skip value) lines in the `.CSV` file are skipped when loading data.

   c. In the *File System* dropdown, list select a value.

      The tool supports HDFS and S3 files systems.

File System	Description
HDFS	Create tables and load data to the tables from files in HDFS. If you select HDFS as the file system, provide the HDFS NameNode details.
S3	Create tables and load data to the tables from files in S3. If you select S3 as the file system, provide the details such as S3 Access Key ID, S3Secret Access Key, and S3 EndPoint.

   d. In the *File Path*, enter the fully qualified path to the file in the selected file system.

> **Note**
>
> You can enter the fully qualified path of more than one file. After entering a fully qualified path, press the return key to enter the full qualified path of another file. The files can be from the same or different file systems.

6. (Optional) Provide file-specific details

   You can load data from multiple files, and these files can exist in different file systems. For each file, you can provide file specific details.

   a. In the *File Path*, select a file.
   b. Select *Enable File specifc options*.
   c. In the *File System* dropdown list, select the file system in which the file is available.
   d. If file is a .csv file, you can provide CSV-specific details such as CSV skip value, delimiter, columns to load, CSV quote value, and more.
7. Choose *OK*.

Related Information

Create Tables in the SAP Vora Relational Engine [page 191]

13.1.2 Create Tables Using Objects from SAP HANA

You can create a table in the SAP Vora engine, by referencing the table definition to an object (table or view) in the SAP HANA system.

Procedure

1. Start the SAP Vora tool in a Web browser.
2. In the home page, select *Modeler*.
3. In the navigation pane, choose + (Create).
4. Select the *Create HANA Table* menu option.
5. Provide a name for the table.
6. In the *Data Source Type* dropdown list, select *HANA* as the data source type.
7. Browse the required SAP HANA object.
   a. In the *Find & Add HANA Object* dialog box, search and select the required SAP HANA object.
   b. Choose *OK*.
8. Choose *Next*.

   The tool displays the table schema for the selected SAP HANA object.
9. Choose *Finish*.

   The tool automatically registers the table in the SparkContext.

> **Note**
>
> Register Tables
>
> Register tables in the SparkContext every time you restart the SAP Vora Thriftserver. In the navigation pane, select *HANA* and, in the context menu, choose *Register All*.

Results

The table definitions are stored in the SAP Vora engine, and you can access them in the navigation pane (under the *HANA* section).

13.2 Working with Other SAP Vora Engines

In addition to the SAP Vora relational in-memory engine, SAP Vora also provides a document store, graph engine, time series engine, and disk engine.

Users can, for example, use the graphical editor in the SAP Vora data modeler tool to create time series tables in the SAP Vora time series engine. This engine helps handle time series data more efficiently. Similarly, users can store tables on the disk using the SAP Vora disk engine.

This section describes how users can use the SAP Vora data modeler tool when working with the SAP Vora engines.

Related Information

Create Tables for Time Series Data [page 196]
Create Tables on the Disk [page 200]
Create Graphs [page 203]
Create Collections [page 205]

13.2.1 Create Tables for Time Series Data

Use the graphical data modeling tools to efficiently create tables with time series data in SAP Vora time series engine. After creating the tables, use them, for example, as data sources to model views in SAP Vora.

Procedure

1. Start the SAP Vora tool in a Web browser.
2. In the home page, select *Modeler*.
3. In the navigation pane, choose + (Create).
4. Select the *Create Series Table* menu option.
5. Provide a name for the table.
6. Select *Vora* as the data source type.

   > **Note**
   >
   > When the data source type is *HANA*, the tables in SAP Vora engine are not time series tables, but instead standard SAP Vora tables.

7. In the *Engine* dropdown list, select *Time Series* as the engine type.
8. (Optional) Provide details for loading data.

You use files in HDFS or S3 to load data into the table. For loading data into the table, provide the following additional information:

a. In the *Delimiter* text field, define the delimiter value.
b. In the *CSV Skip Value* text field, enter a value.

   CSV skip value means that the first 'n' (n is the CSV skip value) lines in the `.csv` file are skipped when loading data.
c. In the *File System* dropdown list, select a value.

   The tool supports HDFS and S3 files systems.

File System	Description
HDFS	Create tables and load data to the tables from files in HDFS. If you select HDFS as the file system, provide the HDFS NameNode details.
S3	Create tables and load data to the tables from files in S3. If you select S3 as the file system, provide the details such as S3 Access Key ID, S3Secret Access Key, and S3 EndPoint.

d. In the *File Path*, enter the fully qualified path to the file.

> **Note**
>
> In the *File Path*, you can enter the fully qualified path of more than one file. After entering a fully qualified path, press the return key to enter the full qualified path of another file. The files can be from the same or different file systems.

9. (Optional) Provide file-specific details

   You can load data from multiple files, and these files can exist in different file systems. For each file, you can provide file specific details.

   a. In the *File Path*, select a file.
   b. Select *Enable File specifc options*.
   c. In the *File System* dropdown list, select the file system in which the file is available.
   d. If file is a .csv file, you can provide CSV-specific details such as CSV skip value, delimiter, columns to load, CSV quote value, and more.
10. Choose *Next*.
11. Define columns.

    a. In the *Name* field, enter the name of the column.
    b. In the *Data Type* dropdown list, select *Timestamp* as the data type.
    c. Select *Period* for columns that you want to specify as period columns.

       The period columns are used as the identifying reference for the series. For instant-based series, select a single period column. For an interval-based time series, select a column that represents the start of an interval and another column that represents the end of the interval.

> **Note**
>
> You cannot explicitly define primary key columns because for a series the columns selected as *PERIOD* are always the primary key. The data type for period columns can only be *Timestamp*.

    d. To create additional columns, in the menu bar, choose + (Add Columns).

12. (Optional) Define a partition scheme.

    You can use a partition scheme to apply and instantiate a partition function.

    a. In the right pane, select the *Partition Scheme* section.
    b. Choose + (Add).
    c. If you want to use an existing partition scheme, select the *Partition Scheme* and choose *Select*.

    > **Note**
    >
    > For creating tables in the SAP Vora time series engine, you can use partition schemes defined with partition function type as RANGE and data type as TIMESTAMP only.

    d. If want to create and use a new partition scheme, in the *Partition Scheme* dialog box, choose + (Add New Scheme) to create a new partition scheme.
    e. In the *Partition Parameter* text field, enter the required partition parameter.

       For a time series table, instantiate this parameter using the period column from the time series.

13. (Optional) Define a range expression.

    The range expression defines and restricts the time range of a series.

    a. In the *Range Expression* dropdown list, select a range expression.
    b. Provide additional details based on the selected range expression type.

14. (Optional) Define whether a series is equidistant.

    This allows you to define whether the distances between all adjacent timestamp in a series comply with a given interval.

    a. In the *Equidistant Options* dropdown list, select an equidistant value.
    b. For equidistant time series, in the *Missing Elements* dropdown list, select the required missing element.

       This allows you to specify whether missing timestamps are allowed in the series. The default configuration is that the missing values are not allowed. r
    c. For equidistant time series, in the *Load Exception* dropdown list, select the required load exception.

       This allows you to define whether deviations from the provided interval are automatically rounded to the series definition when data is loaded into the table.

15. (Optional) Define column compressions.

    A compression strategy is a compression technique that is used for a given range or on a specific column of a time series.

    a. In the *Compression Definition* section, choose +.
    b. In the *Compression Type* dropdown list, select the required compression technique.
    c. In the *Error Bound* dropdown list, provide an error bound value in percentage.
    d. Select the columns or a range of columns that you want to use to define the compressions.
    e. Choose *OK*.

16. Choose *Finish*.

Related Information

Append Data to Existing Time Series Tables [page 199]

13.2.1.1 Append Data to Existing Time Series Tables

Use the SAP Vora data modeler tool to append data to existing time series tables in the time series engine.

Procedure

1. Start the SAP Vora tool in a Web browser.
2. In the home page, select *Modeler*.
3. From the navigation pane, select the required time series table in *Other Vora Engines*.
   The tool opens the schema definition of the selected table.
4. In the menu bar, choose  (Append Data Files).
5. (Optional) Provide details for loading data.
   You use files in HDFS and S3 to load data into the table. In the *Append File* dialog box, provide details of the file from which you want to append data.
   a. In the *Delimiter* text field, define the delimiter value.
   b. In the *File System* dropdown list, select a value.
      The tool supports HDFS and S3 files systems.

File System	Description
HDFS	Create tables and load data to the tables from files in HDFS. If you select HDFS as the file system, provide the HDFS NameNode details.
S3	Create tables and load data to the tables from files in S3. If you select S3 as the file system, provide the details such as S3 Access Key ID, S3Secret Access Key, and S3 EndPoint.

   c. In the *File Path*, enter the fully qualified path to the file in the selected file system.

> **Note**
>
> You can enter the fully qualified path of more than one file. After entering a fully qualified path, press the return key to enter the full qualified path of another file. The files can be from the same or different file systems.

6. (Optional) Provide file-specific details
   You can load data from multiple files, and these files can exist in different file systems. For each file, you can provide file specific details.
   a. In the *File Path*, select a file.
   b. Select *Enable File specifc options*.
   c. In the *File System* dropdown list, select the file system in which the file is available.
   d. If file is a .csv file, you can provide CSV-specific details such as CSV skip value, delimiter, columns to load, CSV quote value, and more.
7. Choose *OK*.

13.2.2 Create Tables on the Disk

Use the graphical data modeling tools to create tables on the disk using the SAP Vora disk engine. After creating the tables, use them, for example, as data sources to model views in SAP Vora.

Procedure

1. Start the SAP Vora tool in a Web browser.
2. In the home page, select *Modeler*.
3. In the navigation pane, choose + (Create).
4. Select the *Create Disk Table* menu option.
5. Provide a name for the table.
6. In the *Data Source Type* dropdown list, select *Vora* as the data source type.
7. In the *Engine* dropdown list, select *Disk* as the engine.
8. (Optional) Provide details for loading data.

   You use files in HDFS or S3 to load data into the table. For loading data into the table, provide the following additional information:

   a. In the *File Type* dropdown list, select the required file type.

      The tool supports CSV, ORC, AVRO, and Parquet file types.

   b. If you have selected the file type as CSV, specify the delimiter that SAP Vora must use to parse the `CSV` files and also a CSV skip value.

      CSV skip value means that the first 'n' (n is the CSV skip value) lines in the `.CSV` file are skipped when loading data.

   c. In the *File System* dropdown list, select a value.

      The tool supports HDFS and S3 files systems.

File System	Description
HDFS	Create tables and load data to the tables from files in HDFS. If you select HDFS as the file system, provide the HDFS NameNode details.
S3	Create tables and load data to the tables from files in S3. If you select S3 as the file system, provide the details such as S3 Access Key ID, S3Secret Access Key, and S3 EndPoint.

   d. In the *File Path*, enter the fully qualified path to the file.

> **Note**
>
> In the *File Path*, you can enter the fully qualified path of more than one file. After entering a fully qualified path, press the return key to enter the full qualified path of another file. The files can be from the same or different file systems.

9. (Optional) Provide file-specific details

   You can load data from multiple files, and these files can exist in different file systems. For each file, you can provide file specific details.

a. In the *File Path*, select a file.
b. Select *Enable File specifc options*.
c. In the *File System* dropdown list, select the file system in which the file is available.
d. If file is a .csv file, you can provide CSV-specific details such as CSV skip value, delimiter, columns to load, CSV quote value, and more.

10. Choose *Next*.
11. Define columns.
    a. In the *Name* field, enter the name of the column.
    b. In the *Data Type*, dropdown list, select the data type.
    c. Provide the *Length* and *Scale* depending on the selected data type.
    d. To create additional columns, choose + (Add Column).
12. (Optional) Define a partition scheme.

    You can use a partition scheme to apply and instantiate a partition function.
    a. In the right pane, select the *Partition Scheme* section.
    b. Choose + (Add).
    c. If you want to use an existing partition scheme, select the *Partition Scheme* and choose *Select*.

    > **Note**
    >
    > For creating tables in the SAP Vora disk engine, you can use partition schemes defined with RANGE or HASH partition function types only.

    d. If want to create and use a new partition scheme, in the *Partition Scheme* dialog box, choose + (Add New Scheme) to create a new partition scheme.
    e. In the *Partition Parameter* text field, enter the required partition parameter.

    When creating a table on disk, instantiate this parameter with a column from the disk table.
13. (Optional) To configure the table or to load data in the table, specify additional configuration details in the details pane.

    For more information about the configuration options, see Disk Engine and Relational Engine Data Source API [page 53].
14. Choose *Finish*.

Related Information

Append Data to Existing Files in the Disk Engine [page 202]

13.2.2.1 Append Data to Existing Files in the Disk Engine

Use the SAP Vora data modeler tool to append data to existing tables in the SAP Vora disk engine.

Procedure

1. Start the SAP Vora tool in a Web browser.
2. In the home page, select *Modeler*.
3. From the navigation pane, select the required table in *Vora Disk Engine*.
   The tool opens the schema definition of the selected table.
4. In the menu bar, choose  (Append Data Files).
5. (Optional) Provide details for loading data.
   You use files in HDFS and S3 to load data into the table. In the *Append File* dialog box, provide details of the file from which you want to append data.
   a. In the *File Type* dropdown list, select the required file type.
      The tool supports CSV, ORC, AVRO, and Parquet file types.
   b. If you have selected the file type as CSV, specify the delimiter that SAP Vora must use to parse the `CSV` files and also a CSV skip value.
      CSV skip value means that the first 'n' (n is the CSV skip value) lines in the `.CSV` file are skipped when loading data.
   c. In the *File System* dropdown list, select a value.
      The tool supports HDFS and S3 files systems.

File System	Description
HDFS	Create tables and load data to the tables from files in HDFS. If you select HDFS as the file system, provide the HDFS NameNode details.
S3	Create tables and load data to the tables from files in S3. If you select S3 as the file system, provide the details such as S3 Access Key ID, S3Secret Access Key, and S3 EndPoint.

   d. In the *File Path*, enter the fully qualified path to the file in the selected file system.

> **Note**
>
> You can enter the fully qualified path of more than one file. After entering a fully qualified path, press the return key to enter the full qualified path of another file. The files can be from the same or different file systems.

6. (Optional) Provide file-specific details
   You can load data from multiple files, and these files can exist in different file systems. For each file, you can provide file specific details.
   a. In the *File Path*, select a file.

b. Select *Enable File specifc options*.
c. In the *File System* dropdown list, select the file system in which the file is available.
d. If file is a .csv file, you can provide CSV-specific details such as CSV skip value, delimiter, columns to load, CSV quote value, and more.

7. Choose *OK*.

13.2.3 Create Graphs

Use the graphical capabilities in the SAP Vora data modeler tool to create and load graphs in SAP Vora graph engine.

Context

You can load data into graphs from JSG files (files with a line-based JSON format) or from CSV files. Linked CSV files allow multiple CSV files with different schemas to be used in combination to load tabular data into one graph. To use linked CSV files, an import definition needs to be specified in JSON format.

Procedure

1. Start the SAP Vora tool in a Web browser.
2. In the home page, select *Modeler*.
3. In the navigation pane, choose + (Create).
4. Select the *Create Graph* menu option.
5. In the *Name* field, provide a name for the graph.
6. In the *Data Source Type* dropdown list, select *VORA*.
7. In the *Engine* dropdown list, select *Graph* as the engine type.
8. (Optional) Provide details for loading data.

   You use files in HDFS or S3 to load data into the table. For loading data into the table, provide the following additional information:

   a. In the *File System* dropdown list, select a value.

      The tool supports HDFS and S3 files systems.

File System	Description
HDFS	Create tables and load data to the tables from files in HDFS. If you select HDFS as the file system, provide the HDFS NameNode details.

File System	Description
S3	Create tables and load data to the tables from files in S3. If you select S3 as the file system, provide the details such as S3 Access Key ID, S3Secret Access Key, and S3 EndPoint.

b. In the *File Path*, enter the fully qualified path to the file in the selected file system.

> **Note**
>
> You can enter the fully qualified path of more than one file. After entering a fully qualified path, press the return key to enter the full qualified path of another file. The files can be from the same or different file systems.

9. (Optional) Create graph with partitioned data.

   You can use a partition scheme to apply and instantiate a partition function.

   a. In the *Partition Scheme* dropdown list, select the required partition scheme.

      The tool populates the dropdown list only with partition schemes that you can use in the SAP Vora graph engine.

   b. If you want to create and add a new partition scheme, choose *More*.

> **Note**
>
> For creating tables in the SAP Vora graph engine, you can use partition schemes defined with partition function type as BLOCK and data type as BIGINT only.

   c. In the *Partition Parameter* text field, enter the required partition parameter.

> **Note**
>
> Partition parameter is a node ID of a graph node. The system displays the data type of the node ID to enter as a hint in the text field. When creating the graph, instantiate this parameter with the NODEID property.

10. (Optional) Provide file-specific details

    You can load data from multiple files, and these files can exist in different file systems. For each file, you can provide file specific details.

    a. In the *File Path*, select a file.
    b. Select *Enable File specifc options*.
    c. In the *File System* dropdown list, select the file system in which the file is available.

11. Choose *Finish*.

Results

The system creates a graph in the SAP Vora engine, and also lists the same under the *Other Data Sources* category (in the navigation pane). You can use this graph as a data source when you are modeling views.

Related Information

Analyze Graph Data [page 229]
Create Views with Graphs [page 221]

13.2.4 Create Collections

Use the graphical capabilities in the SAP Vora data modeler tool to create and load collections in SAP Vora document store. The SAP Vora document store is a distributed in-memory JSON document store that supports rich query processing over JSON data.

Context

A JSON document is stored in a collection, and collections are schema-less. This means that the documents stored in collections are not required to have identical fields, although their structures are similar. You can create both partitioned and nonpartitioned collections.

Procedure

1. Start the SAP Vora tool in a Web browser.
2. In the home page, select *Modeler*.
3. In the navigation pane, choose + (Create).
4. Select the *Create Collection* menu option.
5. In the *Name* field, provide a name for the collection.
6. In the *Data Source Type* dropdown list, select *Vora*.
7. In the *Engine* dropdown list, select the *Document Store* engine type.
8. (Optional) Provide details for loading data.

   You use JSON files in HDFS or S3 to load data into the table. For loading data into the table, provide the following additional information:

   a. In the *File System* dropdown list, select a value.

      The tool supports HDFS and S3 files systems.

File System	Description
HDFS	Create tables and load data to the tables from files in HDFS. If you select HDFS as the file system, provide the HDFS NameNode details.

File System	Description
S3	Create tables and load data to the tables from files in S3. If you select S3 as the file system, provide the details such as S3 Access Key ID, S3Secret Access Key, and S3 EndPoint.

b. In the *File Path*, enter the fully qualified path to the file in the selected file system.

> **Note**
>
> You can enter the fully qualified path of more than one file. After entering a fully qualified path, press the return key to enter the full qualified path of another file. The files can be from the same or different file systems.

9. (Optional) Create collections with partitioned data.

   You can use a partition scheme to apply and instantiate a partition function.

   a. In the *Partition Scheme* dropdown list, select the required partition scheme.

      The tool populates the dropdown list only with partition schemes that you can use in the SAP Vora document store.

   b. If you want to create and add a new partition scheme, choose *More*.

   > **Note**
   >
   > For creating tables in the SAP Vora document store, you can use partition schemes defined with HASH partition function type only.

   c. In the *Partition Parameter* text field, enter the required partition parameter.

   > **Note**
   >
   > The partition parameter is a key value or a property from the document. The system displays the data type of the key value or property to enter as a hint in the text field.

10. (Optional) Provide file-specific details

    You can load data from multiple files, and these files can exist in different file systems. For each file, you can provide file specific details.

    a. In the *File Path*, select a file.
    b. Select *Enable File specifc options*.
    c. In the *File System* dropdown list, select the file system in which the file is available.

11. Select a file to load data.

    a. For *File Path*, browse to the required JSON file in HDFS.
    b. Choose *OK*.

12. Choose *Finish*.

Results

The system creates a collection in the SAP Vora document store, and also lists the same under the *Other Data Sources* category (in the navigation pane). You can use this collection as a data source when modeling views.

Related Information

Append Data to Existing Collections [page 207]
Create Views with Collections [page 220]

13.2.4.1 Append Data to Existing Collections

Use the SAP Vora data modeler tool to append data to existing collections in SAP Vora document store.

Procedure

1. Start the SAP Vora tool in a Web browser.
2. In the home page, select *Modeler*.
3. From the navigation pane, select an existing collection in *Other Vora Engines*.
   The tool opens the schema definition of the selected collection.
4. In the menu bar, choose  (Append Data Files).
5. (Optional) Provide details for loading data.
   You use files in HDFS and S3 to load data into the table. In the *Append File* dialog box, provide details of the file from which you want to append data.
   a. In the *File System* dropdown list, select a value.
      The tool supports HDFS and S3 files systems.

File System	Description
HDFS	Create tables and load data to the tables from files in HDFS. If you select HDFS as the file system, provide the HDFS NameNode details.
S3	Create tables and load data to the tables from files in S3. If you select S3 as the file system, provide the details such as S3 Access Key ID, S3Secret Access Key, and S3 EndPoint.

   b. In the *File Path*, enter the fully qualified path to the file in the selected file system.

> **Note**
>
> You can enter the fully qualified path of more than one file. After entering a fully qualified path, press the return key to enter the full qualified path of another file. The files can be from the same or different file systems.

6. (Optional) Provide file-specific details
   You can load data from multiple files, and these files can exist in different file systems. For each file, you can provide file specific details.

a. In the *File Path*, select a file.
b. Select *Enable File specifc options*.
c. In the *File System* dropdown list, select the file system in which the file is available.

7. Choose *OK*.

13.3 Create Partition Schemes

Use the SAP Vora data modeler tool to create partition schemes. The partition schemes help to apply and instantiate a partition function.

Context

To partition tables, you use partition functions and partition schemes. Partition functions allow you to define how tables should be partitioned, while partition schemes are derived from partition functions and are used to apply the partition functions to the tables.

Procedure

1. Start the SAP Vora tool in a Web browser.
2. In the home page, select *Modeler*.
3. In the navigation pane, choose + (Create).
4. Select the *Create Partition Scheme* menu option.
5. Provide a name for the partition scheme.
6. In the *Partition Type* dropdown list, select a value.

Partition Function Type	Description
RANGE	Range partitioning allows you to define certain values of a specified column as the boundaries for dividing the data into separate partitions. Provide the required data type, boundary values, minimum partitions, and maximum partitions.
HASH	Using HASH partitioning, the values of the specified columns are hashed and added to a partition based on their hash values. Provide the required data type, minimum partitions, and maximum partitions.
BLOCK	Block partitioning is a special type of partitioning that is designed for partitioning graphs. It divides the nodes of a graph into different blocks and buckets. Provide the required block size and number of partitions.

7. Choose *Add*.
8. Choose *Close*.
9. (Optional) Manage partition schemes.

   If you want to view all the available partition schemes or if you want to delete a partition scheme,

   a. In the navigation pane, choose ... (Settings) and select the *Manage Partition Scheme* menu option.

      In the *Partition Scheme* dialog box, the tool displays the list of partition schemes available in your system.

   b. If you want to delete a partition scheme, select the partition scheme and choose  (Delete).
   c. Choose *Close*.

Related Information

Engine Compatibility Overview [page 143]

13.4 Creating Views in SAP Vora

Views are virtual tables that contain the result of an SQL query execution. You can create views and use them later to construct complex queries necessitated by business requirements.

Users can use the graphical data modeling capabilities in the SAP Vora data modeler tool to create views on top of data sources, or create views with star joins, and so on. After creating a view, users can also define SQL operation such as unions, joins, and other SQL operations for data sources within the view.

> ⚠ Restriction
>
> To access views in the Spark Shell, the tables and columns in the view must have uppercase names only.

Related Information

Create Views [page 210]
Create Views with Star Joins [page 217]
Create Views with Collections [page 220]
Create Views with Graphs [page 221]
Supported View Types in SAP Vora Modeler [page 224]

13.4.1 Create Views

Use the graphical data modeling tools in the SAP Vora to create simple SQL views, or dimensions, or cubes that depict real business scenarios. These views can include layers of calculation logic and can contain data from one or more data sources.

Procedure

1. Start the SAP Vora tool in a Web browser.
2. In the home page, select *Modeler*.
3. In the navigation pane, choose + (Create).
4. Choose the *Create View* menu option.
5. In the *Create View* dialog box, provide a name for the view.
6. In the *Type* dropdown list, select a view type.
7. Choose *OK*.

   The tool, after validating the view name, opens the view in a view editor where you can define the view.
8. Add data sources.
   a. In the menu bar, choose + (Add).
   b. In the *Find & Add Data Source* dialog box, search for the name of the data source.
   c. Select the required data source from the search list.

      You can use the wildcard character (*) to list all available data sources. For views with time series tables, you can use only one time series table as a data source.
   d. Choose *OK*.

      > **Note**
      >
      > The *Find & Add Data Source* dialog box displays only data sources that are available in the system. You can select tables or views from the same system when modeling views.

   e. From the context menu of the data source, choose *Add Alias* to provide an alias name to the data source.

   > **Note**
   >
   > You can also drag and drop the data sources from the navigation pane to the view editor.
9. Define output columns.
   a. Select the data source on the view editor.
   b. Select the columns from the data source that you want to add to the view output.
   c. In the context menu, choose *Add to Output*.

> **Note**
>
> To add all columns of the data source to the output, select the data source and choose (Star).
>
> Similarly, to add columns of all data sources, select (Star) in the *Columns* section of the view details pane.

10. Group by and Order by result set.

    If you want to use one or more columns to perform a SQL `GROUP BY` operation or to perform a SQL `ORDER BY` operation.

    a. Select the data source in the view editor.
    b. Select the required column.
    c. In the context menu, choose *Add to Group By* or *Add to Order By* depending on whether you are using the column to perform a `Group By` or `Order By` operation.

11. After you have modeled a view, choose the (Save) icon in the menu bar to save changes.

> **Note**
>
> In the menu bar, choose (Show SQL) to view the equivalent SQL statement that the tool automatically generates for the view.

Next Steps

Modeling views for complex business scenarios can include multiple layers of calculation logic. In such cases, you may need to perform additional tasks to obtain the desired output.

The following table lists some important additional tasks that you can perform to enrich the view.

Requirement	Task to Perform
Query data from two data sources and combine records from both the data sources based on a join condition.	Create Joins
Combine the results of two or more data sources.	Create Unions
Assign semantic types to provide more meaning to columns in the view.	Assign Semantics
Create an SQL query and use it within the main query to restrict the data to be retrieved from the main query.	Create Subselects
Create new output columns and calculate its values at runtime using an expression. You can also create calculated columns to perform standard currency conversion or ERP currency conversion.	Create Calculated Columns
Create level-based hierarchies to organize data in reporting tools.	Create Level-Based Hierarchies
Create parent-child hierarchies to organize data in reporting tools.	Create Parent-Child Hierarchies

Requirement	Task to Perform
Preview the output of a view or any of its intermediate nodes, or preview the SQL statement that the tool generates for the view.	Preview Output of Views

Related Information

Defining SQL Clauses [page 212]
Create Views with Star Joins [page 217]
Create Views with Collections [page 220]
Create Views with Graphs [page 221]
Supported View Types in SAP Vora Modeler [page 224]
Creating Hierarchies [page 231]
Create Table Functions for Time Series [page 236]
Create Calculated Columns [page 238]
Preview Output of Views [page 224]

13.4.1.1 Defining SQL Clauses

When modeling views, based on a business requirement, you may need to define certain supported SQL clauses for the data sources in the view to obtain the desired view output.

The SAP Vora data modeler supports several SQL clauses, including joins and unions, WHERE, HAVING, and LIMIT clauses.

Related Information

Create Joins [page 213]
Create Unions [page 214]
Create SQL WHERE Clause [page 216]
Create SQL HAVING Clause [page 216]
Define SQL LIMIT Clause [page 217]

13.4.1.1.1 Create Joins

Use joins in views to query data from two data sources. You can either limit the number of records, or combine records from both data sources so that they appear as one record in the query results.

Procedure

1. From the navigation pane, select the view in which you want to include a `JOIN` clause.

   The tool opens the view in the view editor.
2. Join two data sources.
   a. Select a data source.
   b. Choose  (Join).
   c. Drag the cursor to the required data source.

   > **Note**
   >
   > To join a data source to a hierarchy in the same view, or to join two hierarchies, first add the hierarchy to a *SubSelect* node. From the context menu of the hierarchy, choose *Move to SubSelect*. You can now join the SubSelect node with any other data source.
   >
   > Similarly, to join a time series table with other time series tables (or with any other data source), add the time series tables within the *SubSelect* node.
3. Join two columns in the data sources.

   If you want to create a join between a specific column in one data source to another column in a different data source,
   a. Select the required column.
   b. Drag the cursor to the required column in the other data source.
4. Define join conditions.

   In the *Join Definition* dialog box, select a join type and define the join conditions.
   a. In the *Type* dropdown list, select the required join type.
   b. In the *Condition* text field, enter a valid join condition.
   c. Choose *More Options* and select the required columns, functions, or operators to build your expression.

      For example: `S_NATIONKEY = supplier_nation.N_NATIONKEY`.

   > **Note**
   >
   > To use the join condition proposed by the tool, based on the selected columns in the two data sources, choose *Propose Condition*.
5. Choose *OK*.

   This operation creates a join path for the two data sources.

> **Note**
>
> To edit the join definition, from the context menu of the join path, select *Edit*.

Related Information

Supported Join Types [page 214]

13.4.1.1.1.1 Supported Join Types

Specify the join type when you create a join between two tables. The following table lists the supported join types in the SAP Vora data modeler.

Join Type	Description
Inner	Returns all rows when there is at least one match in both the database tables
Left Outer	Returns all rows from the left table, and the matched rows from the right table
Right Outer	Returns all rows from the right table, and the matched rows from the left table
Cross	Returns the cartesian product of the two tables
Full Outer	Results from both left and right outer joins and returns all (matched or unmatched) rows from the tables on both sides of the join clause

13.4.1.1.2 Create Unions

You can use the SAP Vora data modeler to create views with unions, allowing you to combine the result sets of two or more data sources.

Prerequisites

You can create unions only if the view has two or more data sources (or subselects).

Procedure

1. From the navigation pane, select the view in which you want to include a `UNION` clause.

   The tool opens the view in the view editor.
2. Select the data source.
3. Choose  (Union).
4. Drag the cursor to the required data source.
5. Add more data sources to the union.

   If you want to union the result sets of multiple data sources (more than two), perform the union operation on the result sets. This means that, you include the data source in a *ResultSet* node, and add the output of the data source to the result set.

   a. In the menu bar, select  (ResultSet).
   b. Drag and drop the *ResultSet* node to the view editor.
   c. Select the *ResultSet* node.
   d. Choose  (ResultSetLink).
   e. Drag the cursor to the required data source.

   > **Note**
   >
   > To create a union between a data source and a hierarchy in the same view, or to perform a union operation between hierarchies, first add the hierarchy to a *SubSelect* node. Select the hierarchy and in the context menu choose, *Move to SubSelect*. You can now create a union between the *Subselect* and any other data sources.

   f. Select the *ResultSet* (not the default result set or the latest *ResultSet* node you included).
   g. Choose  (Union).
   h. Drag the cursor to the required result set.

      This operation creates a union path for the data sources.
6. Add columns to the union output.

   The columns you add to the *Default ResultSet* node are the output columns of the union.

   a. Select the data source.
   b. Select the required columns from the data source that you want to add to the output.
   c. In the context menu, choose *Add to Output*.

   > **Note**
   >
   > A union operation combines the result sets from two or more `SELECT` statements. Each result set (`SELECT` statements) must have the same number of columns and similar data types. The columns in each result set must also be in the same order.

7. (Optional) Include all records from the result sets without removing any duplicate records from the result set:

a. Select the union path.
b. In the context menu, choose *Switch to Union All*.

13.4.1.1.3 Create SQL WHERE Clause

Define a SQL `WHERE` clause in a view to filter and retrieve records from the output of a data source based on a specified condition.

Procedure

1. From the navigation pane, select the view in which you want to include the SQL `WHERE` clause.
   The tool opens the view in the view editor.
2. In the details pane, select *Where*.
3. Choose + (Add).
4. In the expression editor, define the SQL `WHERE` clause condition.
5. (Optional) Choose *More Options* and select the required columns, functions, or operators to build your expression.
6. Choose *OK*.

13.4.1.1.4 Create SQL HAVING Clause

Define a SQL `HAVING` clause in a view to retrieve records from the output of a data source, only when the aggregate values satisfy a defined condition.

Procedure

1. From the navigation pane, select the view in which you want to include the SQL `HAVING` clause.
   The tool opens the view in the view editor.
2. In the details pane, select *Having*.
3. Choose + (Add).
4. In the expression editor, define the SQL `HAVING` clause condition.
5. (Optional) Choose *More Options* and select the required columns, functions, or operators to build your expression.
6. Choose *OK*.

13.4.1.1.5 Define SQL LIMIT Clause

Define SQL `LIMIT` clause on a view to limit the result set of the view.

Procedure

1. From the navigation pane, select the view for which you want to limit the result set.
2. In the details pane, select *Limit*.
3. Use the arrow keys to define the limit value.

   For example, if you set a limit value of 10, the result set includes only the first 10 rows.

13.4.2 Create Views with Star Joins

Star joins in views let you join a central fact table with other dimensional data.

Context

A fact table can contain data that represents business facts, such as price, discount values, number of units sold, and more. Dimension tables represent different ways to organize data, such as geography, time intervals, contact names, and more.

Procedure

1. Start the SAP Vora tool in a Web browser.
2. In the home page, select *Modeler*.
3. In the navigation pane, choose + (Create).
4. Choose the *Create View* menu option.
5. In the *Create View* dialog box, provide a name for the view.
6. In the *Type* dropdown list, select *CUBE* as the view type.

   You can create views with star joins only if the view type is cube.
7. Choose *OK*.

   The tool validates the view name and opens the view editor with the *Facts* node already open. Use this editor to add a star join to the view.
8. Define the central fact table.
   a. Select the *Facts* node.

b. In the context menu, choose *Open*.

c. In the menu bar, choose  (Add) to add the required data sources.

d. In the *Find & Add Data Source* dialog box, search for the name of the data source.

e. Select the data source from the search list.

You can use the wildcard character (*) to list all available data sources.

> **Note**
>
> You can also drag and drop the data sources from the navigation pane to the view editor.

f. Choose *OK*.

g. Continue modeling the view with a cube structure. You can add more data sources, define SQL clauses, create joins, unions, define the output columns, and more.

9. Define the dimensional data.

a. Use the breadcrumb navigation to return to the main view.

b. In the context menu, choose *Open*.

c. In the menu bar, choose + (Add) to add the required data sources.

d. In the *Find & Add Data Source* dialog box, search for the name of the data source.

> **Note**
>
> You can also drag and drop the data sources from the navigation pane to the view editor.

e. Select the required data source from the search list.

You can use the wildcard character (*) to list all available data sources.

> **Note**
>
> You can add only dimension data; for example, views with a data category of *DIMENSION*.

f. Choose *OK*.

10. Create joins.

Create joins between all dimension data sources and the central fact table.

a. Select the required data source.

b. Choose  (Join).

c. Drag the cursor to the *Facts* node.

11. Join two columns in the data sources.

If you want to create a join between a specific column in the dimensional data source with another column in the *Facts* node,

a. Select the required column.

b. Drag the cursor to the required column in the other data source.

12. Define join conditions.

In the *Join Definition* dialog box, select a join type and define the join conditions.

a. In the *Type* dropdown list, select the join type.

b. In the *Condition* text area, enter a valid join condition.

c. Choose *More Options*.
d. Select the required columns, functions, or operators to build your expression.

   For example: `S_NATIONKEY = supplier_nation.N_NATIONKEY.`
e. Choose *OK*.

13. Define output columns.
    a. Select the data source in the view editor.
    b. Select the columns from the data source that you want to add to the view output.
    c. In the context menu, choose *Add to Output*.

> **Note**
>
> To add all columns of the data source to the output, select the data source and choose  (Star).
>
> Similarly, to add columns of all data sources, select  (Star) in the *Columns* section of the view details pane.

14. Group by and Order by result set.

    If you want to use one or more columns to perform a SQL `GROUP BY` operation or to perform a SQL `ORDER BY` operation.
    a. Select the data source in the view editor.
    b. Select the column.
    c. In the context menu, choose *Add to Group By* or *Add to Order By* depending on whether you are using the column to perform a `Group By` or `Order By` operation.

15. After you have modeled a view with star join, choose the  (Save) icon in the menu bar to save changes.

> **Note**
>
> In the menu bar, choose  (Show SQL) to view the equivalent SQL statement that the tool automatically generates for the view.

Results

This operation creates a star join that connects the central fact table with other dimensional data.

> **Note**
>
> To edit the join definition, select *Edit* from the context menu of the join path.

Related Information

Defining SQL Clauses [page 212]
Supported View Types in SAP Vora Modeler [page 224]
Create Views [page 210]

13.4.3 Create Views with Collections

Use the collections from the document store as data sources when modeling a view with the SAP Vora data modeler tool.

Procedure

1. Start the SAP Vora tool in a Web browser.
2. In the home page, select *Modeler*.
3. In the navigation pane, choose + (Create).
4. Choose the *Create View* menu option.
5. In the *Create View* dialog box, provide a name for the view.
6. In the *Type* dropdown list, select a view type.
7. Choose *OK*.

   After validating the view name, the tool opens the view in an editor where you can define it.
8. Add a collection as a data source.
   a. In the menu bar, choose + (Add).
   b. In the *Find & Add Data Source* dialog box, search for the name of the collection.
   c. Select the required collection from the search list.

      You can use the wildcard character (*) to list all available data sources.
   d. Choose *OK*.

   > **Note**
   >
   > You can also drag and drop the collection from the navigation pane to the view editor.
9. Define output columns.
   a. Select the keys from the collection that you want to include as output columns to the view.
   b. In the context menu, choose *Add to Output*.
10. (Optional) Define JSON output.
    a. In the *Columns* section of view details pane, choose (Switch to JSON) to switch to JSON output.
    b. In the *JSON* section, choose + (Add).
    c. In the *JSON Definition* dialog, provide a valid JSON expression.
    d. Choose *More Options* and select the required columns, functions, or operators to build your expression.

e. Choose *OK*.

The tool executes the expression at runtime to identify the JSON output for the view.

11. Group by and Order by result set.

    If you want to use one or more key values from the collection to perform a SQL `GROUP BY` operation or to perform a SQL `ORDER BY` operation.

    a. Select the collection in the view editor.
    b. Select the required key values.
    c. In the context menu, choose *Add to Group By* or *Add to Order By* depending on whether you are using the key value to perform a `Group By` or `Order By` operation.

12. After you have modeled a view with collections, choose the  (Save) icon in the menu bar to save changes.

    In the menu bar, choose  (Show SQL) to view the equivalent SQL statement that the tool automatically generates for the view.

Related Information

Defining SQL Clauses [page 212]
Supported View Types in SAP Vora Modeler [page 224]

13.4.4 Create Views with Graphs

Use graphs from the SAP Vora graph engine as data sources when modeling a view with the SAP Vora data modeler tool.

Procedure

1. Start the SAP Vora tool in a Web browser.
2. In the home page, select *Modeler*.
3. In the navigation pane, choose + (Create).
4. Choose the *Create View* menu option.
5. In the *Create View* dialog box, provide a name for the view.
6. In the *Type* dropdown list, select a view type.
7. Choose *OK*.

   After validating the view name, the tool opens the view in an editor where you can define it.
8. Add a graph as a data source.

   a. In the menu bar, choose + (Add).

b. In the *Find & Add Data Source* dialog box, search for the name of the graph.
c. Select the required graphs from the search list.

   You can use the wildcard character (*) to list all available data sources.
d. Choose *OK*.

The tool displays the metadata of the selected graph in the metadata editor. A*Any* node is also available in the metadata, which contains properties and edges from all nodes in the graph.

> **Note**
>
> You can also drag and drop the graph from the navigation pane to the view editor.

9. Select the required nodes.

   The properties and edges from the selected nodes are available for defining the view.
10. Choose *Close*, to close the metadata editor.

    To open the metadata editor, select  (Graph Metadata) in the view details pane.
11. Define output columns.
    a. Select the properties from nodes, which you want to include as output columns for the views.
    b. (Optional) Select an edge, and use the *Graph Data Path* builder to traverse and select a property from other connected nodes.

       You can also use the *Expression Editor* in the *Graph Data Path* builder to define the graph path expression.
12. (Optional) Create graph functions.

    You can create graph functions and use it as output columns for the view.
    a. In the *Columns* section, choose + (Add).
    b. Select *Add Graph Function*.
    c. In the *Name* text field, provide a name for the graph function.
    d. In the *Function* dropdown list, select the graph function.
    e. In the *Node* dropdown list, selected the required node from the graph.
    f. Provide additional details based on the selected graph function type.
    g. Choose *OK*.
13. Continue modeling the view.

    If you want to join or union the graph data source with other data sources, move the graph data source within a *SubSelect* node. In the view details pane, choose  (Move to Subselect) to move the graph data source into a *SubSelect* node.
14. Group by and Order by result set.

    If you want to use one or more properties to perform a SQL `GROUP BY` operation or to perform a SQL `ORDER BY` operation.
    a. Select the node in the view editor.
    b. Select the required property.
    c. In the context menu, choose *Add to Group By* or *Add to Order By* depending on whether you are using the property to perform a `Group By` or `Order By` operation.

15. After you have modeled a view with graphs, choose the  (Save) icon in the menu bar to save changes.

> **Note**
>
> In the menu bar, choose  (SQL) to view the equivalent SQL statement that the tool automatically generates for the view.

Related Information

Defining SQL Clauses [page 212]
Supported View Types in SAP Vora Modeler [page 224]
Supported Graph Functions [page 223]

13.4.4.1 Supported Graph Functions

SAP Vora supports a number of graph-specific functions. An overview of the available graph functions is shown in the following table:

Function	Description
Degree	The degree of a node is its number of incoming and outgoing edges. "In-degree" is a node's number of incoming edges, while "out-degree" is its number of out-going edges. You can use the DEGREE function to calculate these measures.
Distance	Use the DISTANCE function to calculate the distance of the shortest directed or undirected path between two nodes. A path is a non-empty sequence of edges connecting a source node with a target node.
Connected_Component	Connected components and strongly connected components can be computed using the CONNECTED_COMPONENT function. This function returns an ID for the (strongly) connected component of a particular node.

13.4.5 Supported View Types in SAP Vora Modeler

The SAP Vora modeler classifies and supports several view types. The following table provides more information on each of these view types.

Data Category	Description
SQL	SQL views, which are SQL queries represented as database objects. Just like executing queries on tables, users can execute queries on these SQL views.
Cubes	Cubes, which are used for multidimensional reporting. You can create views based on other SQL views, dimensions, or cubes. For cubes, an additional column property is available to define the default aggregation types for columns.
Dimensions	Standard dimensions, which you can reuse to model cubes (for multidimensional reporting). Dimension views let you drill down on attribute columns in reporting tools.

13.5 Preview Output of Views

After modeling a view in SAP Vora data modeler tool, you can preview the output, and the SQL statement that the tool generates for a selected view.

Context

The tool provides multiple preview options. You can preview output data in simple tabular format, or you can preview output data in graphical representations, such as bar graphs, area graphs, and pie charts. If your view uses hierarchal data structures, the tool allows you to preview output data in hierarchical representations.

Procedure

1. Start the SAP Vora tool in a Web browser.
2. In the home page, select *Modeler*.
3. (Optional) Preview the SQL statement that is been generated for a view:
   a. From the navigation pane, select the required view or table.
      The tool opens the view in a view editor.
   b. In the menu bar, choose  (Show SQL).
4. Preview output of intermediate nodes.

If you want to preview output of any of the data source in the selected view, or the output of any of its intermediate nodes, such as the SubSelect node or ResultSet node,

a. From the navigation pane, select the required view or table.

   The tool opens the view in a view editor.

b. Right-click the required node or data source in the view and choose *Data Preview*.

   The tool opens a new data preview pane in the editor, where you can preview the output of the intermediate node in tabular or graph format, or both.

5. Preview raw data output from the selected view.

   a. In the navigation pane, right-click the view for which you want to preview the output data.

      The tool opens a new data preview pane in the editor. The tool, by default, displays the raw data output in tabular format.

   b. Use the *TABLE* tab to preview the raw data output.

   c. To export the raw data to a `.csv` file, choose (Download as CSV).

6. Preview output of view in a variety of formats, including bar graph, area graph, pie chart, and table chart.

   a. In the navigation pane, right-click the view for which you want to preview the output data.

      The tool opens a new data preview pane in the editor. The tool, by default, displays the raw data output in tabular format.

   b. Choose the *CHARTS* tab.

   c. In the *Configure Chart* pane, configure the required X and Y axis values.

   d. To filter the output results, add, and define the required filter conditions in the *Filter* section.

   e. Choose *Apply*.

      For the defined axis values and filter conditions, the tool displays the output of the view in the selected graphical representation. Use the icons in the menu bar to change the graph format.

7. Preview output of views with hierarchies

   If you have defined views with hierarchies, you can preview the output of such views in hierarchical representations.

   a. In the navigation pane, right-click the view for which you want to preview the output data.

      The tool opens a new data preview pane in the editor. The tool, by default, displays the raw data output in tabular format.

   b. Use the *HIERARCHY* tab to preview the output in hierarchical representations.

      The tool, by default, displays the output in tabular format.

   c. In the menu bar, choose  (Create Hierarchy) to preview output in hierarchical graphs.

      > **Note**
      >
      > This is currently supported only for parent-child hierarchies.

   d. Hover your cursor over each node for more details.

   e. To search for elements in the hierarchy, configure the columns shown in the preview, or define aggregations, choose  (Settings) in the menu bar.

   f. Define your configurations.

   g. Choose *Apply*.

Related Information

Creating Hierarchies [page 231]
Create Views [page 210]

13.6 Visualizing and Analyzing the Data

After creating and loading data in the tables, graphs, and collections in the respective SAP Vora engines, use the SAP Vora data browser tool to visualize and analyze the data. You can use various visualizations including bar graph, area graph, pie chart, and table chart to analyze the data.

This section describes the steps involved in analyzing the data in tables and graphs.

Task	Description
Analyze Time Series Data	Explains the steps required to visualize and analyze the time series data.
Analyze Graphs	Explains the steps required to visualize and analyze graph data.
Analyze data in tables	Explains the steps required to visualize and analyze the data in SAP Vora tables (relational engine) or disk tables (disk engine).

Related Information

Analyze Data in Tables [page 226]
Analyze Time Series Data [page 227]
Analyze Graph Data [page 229]

13.6.1 Analyze Data in Tables

Use the data browser tool in SAP Vora to graphically visualize and analyze the data in tables. The tables can exist in the SAP Vora relational engine or in the disk engine.

Context

You can use the SAP Vora data modeler tool to create tables in the SAP Vora relational engine or SAP Vora disk engine. After creating the table and loading data to the tables, you can use the data browser tool to graphically visualize and analyze data.

Procedure

1. Start the SAP Vora tool in a Web browser.
2. In the home page, select *Data Browser*.

   This operation opens the data browser tool, where you can visualize and analyze the time series data.
3. In the navigation pane, expand the *Vora Relational Engine* or *Vora Disk Engine* section.

   Select depending on whether you want to analyze the tables in the SAP Vora relational engine or the SAP Vora disk engine.
4. Select the table that you want to graphically visualize.

   The tool displays the data in the selected table in tabular format. You can export and download the table in .CSV format. Choose (Download as CSV) to download the data.
5. Analyze data in a variety of formats, including bar graph, area graph, pie chart, and table chart.
   a. Choose the *CHARTS* tab.
   b. In the *Configure Chart* pane, configure the required X and Y axis values.
   c. To filter the output results, add, and define the required filter conditions in the *Filter* section.
   d. Choose *Apply*.

      For the defined axis values and filter conditions, the tool displays the data in the selected graphical representation. Use the icons in the menu bar to change the graph format.
   e. (Optional) To view the SQL statement that the tool generates for this configuration, choose  (View SQL).
   f. (Optional) To create a SQL view for this chart configuration, in the configuration pane, choose  (Create View).

13.6.2 Analyze Time Series Data

Use the data browser tool in SAP Vora to graphically visualize and analyze time series tables created in the SAP Vora time series engine.

Context

You can use the SAP Vora data modeler tool to create time series tables in the SAP Vora time series engine. After creating the table and loading time series data to the table, you can use the data browser tool to graphically visualize and analyze the time series data.

Procedure

1. Start the SAP Vora tool in a Web browser.

2. In the home page, select *Data Browser*.

   This operation opens the data browser tool, where you can visualize and analyze the time series data.
3. In the navigation pane, expand the *Other Vora Engines* section.
4. Select the time series table from the SAP Vora time series engine that you want to graphically visualize.

   The tool displays the data in selected time series table in tabular format. You can export and download the table in `.csv` format. Choose  (Download as CSV) to download the time series data.
5. Analyze the time series data using the various charts.
   a. In the header, choose the *Charts* tab.
   b. In the *Configuration Chart* pane, choose + (Add New Table Function).

Menu Option	Description
Basic	To use a basic query to configure the chart and analyze the time series data
Histogram	To use a histogram function to configure the chart and analyze the time series data
Granulize	To use a granulize function to configure the chart and analyze the time series data

> **Note**
>
> You can also stack conditions to configure the chart and analyze the time series data. For example, you can use both a histogram function and a granulize function at the same time.

6. (Optional) Analyze the time series data with basic query.
   a. In the *Configuration Chart* pane, choose + (Add New Table Function) and select the *Basic* menu option.
   b. Choose + (Add Column) to define a query with one or more descriptor column and provide values.
   c. Select *From* and *To* time values.

      The tool displays the time series data for this time period.
   d. Choose *Apply*.
   e. (Optional) To view the SQL statement that the tool generates for this configuration, choose  (View SQL).
   f. (Optional) To create a SQL view for this chart configuration, in the configuration pane, choose  (Create View).

   The tool displays a table chart for the defined configuration. In the header bar, choose a required graphical representation to analyze the time series data.
7. (Optional) Analyze time series data with granulize table function.
   a. In the *Configuration Chart* pane, add a granulize table function.
   b. In the *Period Column* dropdown list, select the period column.
   c. In the *Descriptor* section, choose + to add a descriptor column.

      You can configure the granulize table function with more than one descriptor column.
   d. In the *Rounding Mode* dropdown list, select the required value.

      The rounding mode defines how values that fall in between the target interval are assigned to the instances of the new interval.

e. Select *Enable Aggregation*, to configure the table function with one or more standard aggregations.
f. Select *From* and *To* time values.

   The tool displays the time series data for this time period.
g. Choose *Apply*.
h. (Optional) To view the SQL statement that the tool generates for this configuration, choose (View SQL).
i. (Optional) To create a SQL view for this chart configuration, in the configuration pane, choose (Create View).

The tool displays a table chart for the defined configuration. In the header bar, choose a required graphical representation to analyze the time series data. If you have enabled aggregations in your configuration, you can analyze the time series data with a pie chart and bar graph. If you have not enabled aggregations, you can analyze the time series data using a line chart and an area chart. For line charts, you can use a slider at the bottom of the chart to increase or decrease the area resolution.

8. Analyze time series data with histogram table function
   a. In the *Configuration Chart* pane, add a histogram table function.
   b. In the *Descriptor* dropdown list, select the required value.
   c. In the *No. Of Bins* text field, enter the number of bins that the histogram must create.
   d. Choose *Apply*.
   e. (Optional) To view the SQL statement that the tool generates for this configuration, choose (View SQL).
   f. (Optional) To create a SQL view for this chart configuration, in the configuration pane, choose (Create View).

   The tool displays a table chart for the defined configuration. In the header bar, choose a required graphical representation to analyze the time series data. For analyzing time series data with histograms, the tool supports bar graph and pie chart graphical representations.

13.6.3 Analyze Graph Data

Use the data browser tool in SAP Vora for a graphical visualization of graphs created in the SAP Vora graph engine.

Context

You can use the SAP Vora data modeler tool to create graphs in the SAP Vora graph engine. After creating the graph and loading data to the graph, use the data browser tool to obtain a graphical summary of the nodes and edges in the graph.

Procedure

1. Start the SAP Vora tool in a Web browser.

2. In the home page, select *Data Browser*.

   This operation opens the data browser tool, which you can use to obtain a summary of graphs.
3. In the navigation pane, expand the *Other Vora Engines* section.
4. Select the graph that you want to graphically visualize.

   When you select a graph, the tool loads the summary of the graph. In the *SUMMARY* tab, you can obtain information on the distinct number of nodes, total number of nodes and edges, and more.
5. In the *Nodes* tile, you can change the graphical representation (pie charts, bar graphs, or table charts) to view the distinct number of nodes in the graph.
6. In the *Node Details* section, use the *Node Type* dropdown list to filter and view nodes based on the node type value.
7. Graphically visualize the graph.
   a. In the menu bar, select the *VISUALIZATION* tab.

      When you select the *VISUALIZATION* tab, you maybe prompted to first filter the result set. You must filter because the graph contained more than seven hundred (700) nodes. Filtering helps obtain a better visualization, and you can add one or more filter conditions.
   b. If you are prompted to apply filters, in the menu bar, choose  (Add Filter) to filter the graph based on the node type.
   c. Select a node in the visualization.

      In the *GENERAL* tab (see the right pane), the tool provides more details on the different nodes and edges in the graph. In the *Selection* section, see more details about the selected node.
   d. Select an edge.

      In the *Selection* section, see more details on the selected edge.
   e. In the menu bar, choose  (Add Filter) to filter the graph based on the node type.

      Filtering helps restrict the number of nodes in the visualization.
   f. If you want to view the SQL statement that the tool generates for fetching the data for visualization, in the menu bar, choose  (Show SQL).

      If you have defined any filter condition, the tool generates the SQL statement accordingly. Choose *Copy* to copy the SQL statement to the clipboard.
   g. (Optional) To create a view for the selected graph visualization and filter conditions, in the menu bar, choose  (Create View).
   h. To view all first-level neighborhood nodes for a selected node, right-click the node in the visualization and choose *Load Neighboring Nodes*.
   i. After loading the neighboring nodes, if you want to view the properties of the neighboring nodes, right-click the node and choose *View Neighboring Node Properties*.
   j. If you want to search the neighboring nodes based on the node property, right-click the node and choose *Search Nieghboring Nodes*.
8. (Optional) Configure visualization.

   You can configure the visualization, for example, the node and edge style the tool must use in the visualization.
   a. In the *CONFIG* tab, define the required configurations.
9. (Optional) Create new style rules.

In the *CONFIG* tab, you can create new style rules and apply these style rules for specific nodes in the visualization. The tool uses the new style rule for such nodes, and applies the default node style for all other nodes in the visualization.

a. In the *Style Rules* section, choose + (Add Styles) to create new style rules.
b. In the *Name* field, provide a name for the new style.
c. In the *Property* dropdown list, select the required node property.
d. In the *Value* field, provide the required value.

   The tool applies the new style rule to all nodes that use the selected node property value.
e. In the *Node Styles* section, define the required node style.
f. Choose *OK*.

13.7 Additional Functionality for Views

For some business scenarios, you may need to perform certain additional functions to improve the efficiency of the view or to obtain the desired output. You can perform these additional functions while you design the view.

Related Information

Create Calculated Columns [page 238]
Create Subselects [page 240]
Assign Semantics [page 242]
Enable Attributes for Drilldown in Reporting Tools [page 243]
Associate Columns with Label Columns [page 244]
Add Alias Names to Columns [page 244]
Creating Hierarchies [page 231]
Create Table Functions for Time Series [page 236]
Import or Export Views [page 245]

13.7.1 Creating Hierarchies

Hierarchical data structures define a parent-child relationship between different data items, making it possible to perform complex computations on different levels of data.

An organization, for example, is a hierarchy where the connections between nodes (for example, manager and employee) are determined by the reporting lines that are defined by that organization. The SAP Vora data modeler tool supports parent-child hierarchies and level-based hierarchies.

Related Information

Create Parent-Child Hierarchies in Views [page 232]
Create Level-Based Hierarchies in Views [page 234]

13.7.1.1 Create Parent-Child Hierarchies in Views

A parent-child hierarchy between data items lets you perform complex computations on different levels of data.

Context

If data sources contain data that represent parent-child relationships, you can create hierarchies. For example, an organization is a hierarchy where the connections between nodes (for example, manager and developer) are determined by the reporting lines that are defined by that organization. Creating a hierarchy lets you perform complex aggregations, such as calculating the average age of all second-level managers, the average salaries of different departments, and more.

> **Note**
>
> For more information about the functions you can use with hierarchies, see Hierarchy UDFs [page 171]

Procedure

1. From the navigation pane, select the view in which you want to create the hierarchy.
   The tool opens the view in the view editor.
2. In the menu bar, select  (Create Hierarchy).
3. Drop the *Create Hierarchy* node onto the view editor.
4. In the *Create Hierarchy* dialog box, provide a name for the hierarchy.
5. In the *Type* dropdown list, select *Parent Child* as the hierarchy type.
6. In the *Node Column Name* text field, enter a name for the node column.
   You can use the node column to perform hierarchy operations, for example, a SQL `GROUP BY` operation. For each row in the hierarchy relation, the node column contains information that specifies its location in the hierarchy.
7. Choose *OK*.
8. Add a table, a view, a subselect, or another hierarchy (nested hierarchies) as the data source.
   a. In the menu bar, choose + (Add).

b. In the *Find & Add Data Source* dialog box, search for the name of the data source.

   You can use the wildcard character (*) to list all available data sources.

c. Select the data source from the search list.

d. Choose *OK*.

> **Note**
>
> The *Find & Add Data Source* dialog box displays only data sources that are available in the system. You can thus select only tables or views from the same system when modeling views.

9. Define the parent and child.

   In the hierarchy definition pane, provide additional details for the hierarchy.

   a. In the *Parent* dropdown list, select the column from the data source as the parent node.

   b. In the *Child* dropdown list, select the column from the data source as a child node.

10. (Optional) Define the hierarchy root.

    In the hierarchy definition pane, define the hierarchy root.

    a. In the *Start Where* section, choose + (Add).

    b. In the *Start Expression* editor, enter an expression.

       Any row that matches the result of evaluating the expression is considered as root in the hierarchy.

    c. Choose *More Options*, dropdown list, select the required columns, functions, or operators to build your expression.

    d. Choose *OK*.

    > **Note**
    >
    > If you have not defined a root hierarchy, the tool automatically determines the root for the hierarchy by scanning all source table rows and identifying those hierarchies that do not have a parent.

11. (Optional) Define order by column.

    If you want to use one or more columns to perform a SQL order by operation,

    a. In the view editor, select the data source.

    b. Select the required column.

    c. In the context menu, choose *Add to Order Siblings By*.

    > **Note**
    >
    > The *Order Siblings By* columns determine the order of the children when the hierarchy is constructed. The order is relevant for hierarchy functions such as `IS_PRECEDING` and `IS_FOLLOWING`. For example, if you do not define the *Order Siblings By* column, but use the hierarchy functions in expressions that depend on the order of the children, the system returns undefined results.

12. Use the breadcrumb navigation to return to the main view and to continue modeling the view.

13. Continue modeling the view.

    You can add more data sources, create joins, unions, define SQL clauses, define the output columns, and more.

> **Note**
>
> Move the hierarchy to a *SubSelect* node to perform union or join SQL operations between two hierarchies or with any other data source.

a. Select the hierarchy.
b. In the context menu, choose *Move to SubSelect*.

14. Group by and Order by result set.

    If you want to use one or more columns to perform a SQL `GROUP BY` operation or to perform a SQL `ORDER BY` operation.

    a. Select the data source in the view editor.
    b. Select the required column.
    c. In the context menu, choose *Add to Group By* or *Add to Order By* depending on whether you are using the column to perform a `Group By` or `Order By` operation.

15. Choose the  (Save) icon in the menu bar to save changes.

> **Note**
>
> In the menu bar, choose  (Show SQL) to view the equivalent SQL statement that the tool automatically generates for the view.

Related Information

Create Views [page 210]

13.7.1.2 Create Level-Based Hierarchies in Views

In level-based hierarchies, each level represents a position in the hierarchy, and you can map each level to source table columns. For example, you can create a level hierarchy with the columns, MONTH, QUARTER, and YEAR as levels in the hierarchy.

Procedure

1. From the navigation pane, select the view in which you want to create the hierarchy.

   The tool opens the view in the view editor.

2. In the menu bar, select  (Create Hierarchy).

3. Drop the *Create Hierarchy* node onto the view editor.
4. In the *Create Hierarchy* dialog box, provide a name for the hierarchy.
5. In the *Type* dropdown list, select *Level* as the hierarchy type.
6. In the *Node Column Name* text field, enter the name for the node column.

   You can use the node column to perform hierarchy operations, for example, a SQL `GROUP BY` operation. For each row in the hierarchy relation, this column contains information necessary to specify its location in the hierarchy.
7. Choose *OK*.
8. Add a data source.

   You can add a table, a view, subselect or another hierarchy (nested hierarchies) as data source.
   a. In the menu bar, choose + (Add).
   b. In the *Find & Add Data Source* dialog box, search for the name of the data source.
   c. Select the data source from the search list.

      You can use the wildcard character (*) to list all available data sources.
   d. Choose *OK*.
9. Define levels.
   a. Select the data source.
   b. Select the required column that you want to map to a level in the hierarchy.
   c. In the context menu, choose *Add to Levels*.
10. Define order by column.

    If you want to use one or more columns to perform a SQL order by operation,
    a. In the view editor, select the data source.
    b. Select the required column.
    c. In the context menu, choose *Add to Order Siblings By*.

> **Note**
>
> The *Order Siblings By* columns determine the order of the children when the hierarchy is constructed. The order is relevant for hierarchy functions such as `IS_PRECEDING` and `IS_FOLLOWING`. For example, if you do not define the *Order Siblings By* column, but use the hierarchy functions in expressions that depend on the order of the children, the system returns undefined results.

11. Use the breadcrumb navigation to return to the main view and to continue modeling the view.
12. Continue modeling the view.

    You can add more data sources, create joins, unions, define SQL clauses, define the output columns, and more.

> **Note**
>
> Move the hierarchy to a *SubSelect* node to perform union or join SQL operations between two hierarchies or with any other data source.

    a. Select the hierarchy.
    b. In the context menu, choose *Move to SubSelect*.
13. Group by and Order by result set.

If you want to use one or more columns to perform a SQL `GROUP BY` operation or to perform a SQL `ORDER BY` operation.

a. Select the data source in the view editor.
b. Select the required column.
c. In the context menu, choose *Add to Group By* or *Add to Order By* depending on whether you are using the column to perform a `Group By` or `Order By` operation.

14. Choose the  (Save) icon in the menu bar to save changes.

> **Note**
>
> In the menu bar, choose  (Show SQL) to view the equivalent SQL statement that the tool automatically generates for the view.

Related Information

Create Views [page 210]

13.7.2 Create Table Functions for Time Series

Use the graphical data modeling capabilities in the SAP Vora data modeler tool to create table functions for time series.

Context

Table functions let you calculate auto-correlation and cross-correlation coefficients, create histograms, and increase or decrease the granularity of a time series.

Procedure

1. In the navigation pane, select the view in which you want to create a table function for the time series data.
2. In the menu bar, choose  (Create Time Series Table Function).
3. Drop the time series table function node onto the view editor.
4. In the *Add Time Series Table Function* dialog box, provide a name for the table function.
5. In *Type* dropdown list, select the table function type.
6. Choose *OK*.

7. Add data source.

   You can add a time series table as a data source.

   a. In the menu bar, choose + (Add) to add the required data sources.

      > **Note**
      >
      > You can also drag and drop the data sources from the navigation pane to the view editor.

   b. In the *Find & Add Data Source* dialog box, search for the name of the data source.
   c. Select the required data source from the list.

      You can use the wildcard character (*) to list all available data sources.
   d. Choose *OK*.
8. Define the table function.

   Based on the selected table function type, in the *Time Series Table Function* definition pane, define the parameters for the table function. For example, if you have selected a table function of type auto correlation, then provide a maximum time lag value and the description value.
9. If you have selected the table function type as *Granulize*, then in addition to the Interval Constant, Offset, and Rounding Values, also define the granulize columns.
   a. In the table function details pane, choose + (Add) to add granulize columns.
   b. In the *Granulize Mode* dropdown list, select an appropriate value for the granulize mode.
   c. In the *Descriptor* dropdown list, select a descriptor value.

      Descriptor values are column values from the time series.
   d. Choose *OK*.
10. Use the breadcrumb navigation to return to the main view and to continue modeling the view.
11. Continue modeling the view.

    You can add more data sources, create joins, unions, define SQL clauses, define the output columns, and so on.
12. Choose the  (Save) icon in the menu bar to save changes.

> **Note**
>
> In the menu bar, choose  (Show SQL) to view the equivalent SQL statement that the tool automatically generates for the view.

Related Information

Supported Table Function Types for Time Series Data [page 238]
Table Functions [page 98]
Create Views [page 210]

13.7.2.1 Supported Table Function Types for Time Series Data

SAP Vora data modeler allows you to use graphical modeling tools to create table functions for time series. The following table describes the supported table function types.

Type	Description
Auto-Correlation	Calculates the auto-correlation coefficient for a given column and a given lag time. Provide the maximum time lag value and select the required description value.
Cross- Correlation	Calculates the correlation between two columns. Provide the maximum time lag value and select the descriptor values. Optionally, you can also specify the number of elements with the samples.
Histogram	Creates a histogram of the values for the selected column. Provide the number of bins that the histogram should create and select a descriptor value.
Granulize	Returns a new series based on an existing series by changing the interval between adjacent timestamps. You can increase or decrease the time granularity for a target interval (which combines a unit value), provide an offset value, select a rounding mode, and define granulize columns and granulize mode.

13.7.3 Create Calculated Columns

Create new output columns for a view and calculate the column values at runtime based on the result of an expression. You can use other column values, functions, or constants when defining the expression.

Context

For example, you can create a calculated column DISCOUNT using the expression CASE WHEN PRODUCT='NOTEBOOK' THEN DISCOUNT*0.10 END AS DISCOUNT. This expression uses the function CASE, the column PRODUCT, and the operator * to obtain values for the calculated column, DISCOUNT.

Procedure

1. From the navigation pane, select the view in which you want to create the calculated column.
   The tool opens the view in the view editor.
2. Select the data source in which to create the calculated column.
3. In the view details pane, select *Columns*.
4. Choose + (Add).
5. In the *Add Calculated Column* dialog box, provide a name for the calculated column.

6. In the *Expression* editor, enter a valid expression.
7. Choose *More Options* and select the required columns, operators, or functions to define your expression.

   The SAP Vora data modeler tool computes this expression at runtime to obtain values of the calculated column.
8. Choose *OK*.

Related Information

Use Calculated Columns for Currency Conversions [page 239]

13.7.3.1 Use Calculated Columns for Currency Conversions

Perform standard currency conversions or ERP currency conversions by creating calculated columns using currency conversion functions. The calculated columns store the result of currency conversion, when you execute the view.

Prerequisites

- You have the rates table in the same cluster, for performing standard currency conversion.
- You have the standard SAP currency conversion tables (rates table, rates prefactor table, a currency precision table, and a client settings table) in the same cluster.

Context

Create calculated columns and use the built-in predefined currency conversion functions in SAP Vora data modeler to perform standard or ERP currency conversions. For a standard currency conversion, the function uses a rates table to perform the conversion and provides a standard set of options. Similarly, for an ERP currency conversion, the function uses a rates table, rates prefactor table, a currency precision table, and a client settings table.

Procedure

1. From the navigation pane, select the view in which you want to create the calculated column.

   The selected view opens in the view editor.
2. Select the data source in which you want to create the calculated column.

3. In the view details pane, select *Columns*.
4. Choose + (Add).
5. Define the calculated column.
   a. In the *Name* field, provide a name for the calculated column.
   b. Choose *More Options*.
   c. In the *Function* pane, expand *Currency Conversion*.
   d. Select the required currency conversion function.

      Use the function `cc` to perform standard currency conversion and use the function `convert_currency` for ERP currency conversion.
   e. In the *Expression* editor, provide parameter values for the currency conversion function.
6. Choose *OK*.

Results

The SAP Vora data modeler tool computes this expression at runtime to obtain values of the calculated column.

Related Information

Standard Currency Conversion [page 173]
ERP Currency Conversion [page 176]

13.7.4 Create Subselects

To model complex business scenarios, you may need to create nested SQL queries or subqueries to obtain the desired output. You can achieve the desired output in the SAP Vora data modeler tool by creating subselects within the target view.

Context

Create a subselect within a view to filter the data retrieved by the target view. You can create multiple subselects within a single view.

Procedure

1. From the navigation pane, select the view in which you want to create subselects.
   The tool opens the view in the view editor.
2. In the menu bar, select  (Create SubSelect).
3. Drop the *SubSelect* node onto the view editor.
4. In the *Add Alias* dialog box, provide a *Alias* name.
   If you are creating multiple subselects within the same view, then provide alias names to each of subselects.
5. Choose *OK*.
6. Define the subselect.
   Defining a subselect is similar to defining a view. You can add data sources, create unions or joins, define SQL clauses, define column properties, or you can create another subselect within it.

   > **Note**
   >
   > You can use the breadcrumb navigation in the menu bar to switch between subselects or the target view.

7. Use the breadcrumb navigation to return to the main view and to continue modeling the view.
8. Continue modeling the view.
   You can add more data sources, create joins, unions, define SQL clauses, define the output columns, and more.
9. Choose the  (Save) icon in the menu bar to save changes.

   > **Note**
   >
   > In the menu bar, choose  (Show SQL) to view the equivalent SQL statement that the tool automatically generates for the view.

Related Information

Create Views [page 210]

13.7.5 Assign Semantics

Assign semantics to provide meaning to the output columns and to define output structure of the view.

Context

For example, if you use an attribute column, PRICE, which contains product price data, and another attribute column, CODE, which contains the currency format, you can then assign the semantic type Currency Code to CODE and assign the semantic type Amount with Currency Code to PRICE.

Procedure

1. From the navigation pane, select the required view.
   The tool opens the view in the view editor.
2. In the view details pane, select *Columns*.
3. Choose  (Edit).
4. Assign Semantics
   a. For each column, in the *Column Editor* > *Semantics Type* column, choose *Add*.
   b. In the *Semantic Type* dropdown list, select a semantic type.

      For the semantic types *Amount with Currency Code* or *Quantity with Unit of Measure*, you can also specify the columns that contains the relevant currency code value or unit of measure value, respectively.
5. Choose *OK*.

   The system internally stores values of semantic type properties as annotations.

Related Information

Supported Semantics Types [page 242]
Adding Annotations [page 39]

13.7.5.1 Supported Semantics Types

Client tools use semantic types to represent data in an appropriate format.

You can assign any of the following supported semantic types to columns.

- Amount with Currency Code
- Quantity with Unit of Measures
- Currency Code
- Unit of Measure
- Date
- URL

13.7.6 Enable Attributes for Drilldown in Reporting Tools

If your business requirements need it, you can enable drilldown of attribute columns in reporting tools. Drilldown is applicable only for columns in dimensional views.

Context

By default, the attribute columns of dimensional views are available for drilldown in reporting tools. If you do not want to drill down columns in reporting tools, disable the behavior.

Procedure

1. From the navigation pane, select the required view.
   The tool opens the view in the view editor.

   > **Note**
   > You can enable drilldown for columns in views of type dimension only.

2. In the view details pane, select *Columns*.
3. Choose (Edit).
4. In *Column Editor* > *DrillDown Enablement*, select a value for each column.

Value	Description
None	To disable drilldown property for a column
<blank>	The default drilldown property for columns. It signifies that the column is available for drilldown in the reporting tools, but that the tool does not create annotations internally for this property.
Drill_Down	To enable drilldown and to internally create annotations for this property.

5. Choose *OK*.

Related Information

Adding Annotations [page 39]

13.7.7 Associate Columns with Label Columns

Associate an attribute column that contains descriptions (or texts) with a label column. The label column then acts as a pointer to the attribute column that contains the descriptions for a given column.

Procedure

1. From the navigation pane, select the required view.
   The tool opens the view in the view editor.
2. In the view details pane, select *Columns*.
3. Choose  (Edit).
4. In *Column Editor* > *Label Column*, select a column value from the dropdown list for each column.
5. Choose *OK*.
   The system internally stores the values of the label column property as annotations.

Related Information

Adding Annotations [page 39]

13.7.8 Add Alias Names to Columns

If you are using the same column more than once in a data source, assign an alias name to the column to avoid duplicate column names.

Procedure

1. From the navigation pane, select the required view.
   The tool opens the view in the view editor.
2. In the view details pane, expand the *Columns* section.

3. Select the column for which you want to provide an alias name.
4. In the context menu, choose *Edit*.
5. In the *Edit Column Properties* dialog box, enter the alias name.
6. In the *Aggregate Function* dropdown list, select an aggregate function.
7. Choose *OK*.

13.7.9 Import or Export Views

Export the SQL statement that the tool generates for a view created in the SAP Vora data modeler tool. You can then use the file in other capacities, for example, import it to other clusters to use the same SQL statement there.

Context

The SQL statement for a view is exported as a text file to the local system. You can import the text file to other clusters. You can import or export more than one view at a time.

Procedure

1. From the navigation pane, select the view for which you want to export the SQL statement.
   This operation opens the view in the view editor.
2. In the menu bar, choose (Export Views).
3. (Optional) Exporting more than one view.
   a. From the navigation pane, choose (Export Views).
      The tool displays the list of views that you can export.
   b. In the *Export Views* dialog box, select the views for which you want to export SQL statements.
   c. Choose *OK*.
   The system automatically downloads a text file, which contains the SQL statements of the selected views.
4. Import the exported text files.
   a. From the navigation pane, choose (Import Views).
   b. Browse the text file, which contains the SQL statements you want to import.

14 SAP Vora Integration with Spark 2

The integration between SAP Vora and Spark 2 differs significantly compared to SAP Vora's integration with Spark 1.6.

The two main differences are as follows.

- You can no longer issue DDL statements using Spark SQL or Spark methods.
  For general custom code, however, you can now create and alter tables in an easier and more consistent way.
- Spark code is no longer altered. This means that `SQLContext` and `SparkSession` are not modified in any way.

These changes enable SAP Vora to be more easily integrated with existing third-party tools. Since the Spark 1.6 integration altered Spark SQL and some of its APIs, this meant that not all external tools could operate well with SAP Vora's altered Spark version.

Structure

The SAP Vora integration with Spark 2 is structured into two main packages, `sap.client` and `sap.spark`, where each has its own subpackage for both SAP Vora (`sap.client.vora` and `sap.spark.vora`) and SAP HANA (`sap.client.hana` and `sap.spark.hana`).

Spark Integration

There are two separate data sources for SAP Vora and SAP HANA. Although their parameters are largely the same, they differ in terms of the specific characteristics of the target database systems.

Note that in general the following documentation provides an overview of the functionality and characteristics of the SAP Vora and SAP HANA data sources.

For a complete Spark or Spark SQL reference, please see the respective Spark documentation.

14.1 Clients

Clients are the main components for accessing and invoking SAP Vora or SAP HANA functionality. There are several ways in which you can instantiate a client.

For both SAP HANA and SAP Vora, the companion object of the clients (that is, `sap.client.vora.PublicVoraClient` and `sap.client.hana.PublicHanaClient`) expose `apply` methods that accept the parameters that are required to instantiate a client. For example, for the

`PublicVoraClient`, these are the `host` and the `port` of the transaction coordinator instance you want to connect to:

```
import sap.client.vora
val client = PublicVoraClient("txchost.example.org", 2202)
```

Since it is tedious to extract the SAP Vora configuration from the Spark defaults, there is a helper package in the `sap.spark.vora` or `sap.spark.hana` package called `PublicVoraClientUtils` or `PublicHanaClientUtils`. The helper package exposes a method called `createClient` that accepts either a `SparkSession` or an `SQLContext` and which instantiates a `PublicVoraClient` or `PublicHanaClient` by extracting the necessary values from the configuration of the given `SparkSession` or `SQLContext`.

The SAP HANA and SAP Vora clients both maintain a single connection. This is why you need to `close` them after using them. The two main interface methods of the clients are `query` and `execute`:

- `query` sends the given SQL query to the target database engine and then efficiently fetches the result:

  ```
  val result = client.query("SELECT * FROM SYS.TABLES")
  result.foreach(println)
  ```

  The resulting values are returned in the form of `HanaRow`s or `VoraRow`s. A `HanaRow/VoraRow` contains the values as an `IndexedSeq` of `Option[Any]`. For typed access, you can call the `get...` methods that do the necessary type checks and value retrievals for you.
- `execute` works like `query` although it does not yield a result. It should be used in the context of DDL operations:

  ```
  client.execute("CREATE TABLE created_by_client(a int, b int)")
  ```

14.2 SAP Vora Data Source

The SAP Vora data source allows data to be queried from the SAP Vora engines in two main ways; through table registration with the `table` option or by submitting ad-hoc queries using the `query` option. It also allows data to be inserted in the standard Spark manner.

14.2.1 Configuring the SAP Vora Data Source

The SAP Vora data source supports a layered configuration approach. This allows configuration values to be specified in several locations at the same time, while an order of precedence, based on location, is used to determine which values are actually applied.

This is particularly useful if you want to provide sensible default values for all your `SparkSession`s, but don't want to re-enter them every time.

You can enter configuration values in the following locations, listed in ascending order of precedence (that is, the last location in the list represents the highest layer and overwrites all preceding layers):

- `SparkContext` configuration through your `spark-defaults.conf` file and through the `SparkConf` object during the construction of your `SparkSession`.

This configuration has to be prefixed with the SAP Vora Spark configuration prefix `spark.sap.vora`. For example, in your `spark-defaults.conf` file:

```
spark.sap.vora.host = txc.example.org
```

- `SparkSession` through `SparkSession.conf`. These values also have to be prefixed in the same way as described above.
- Per query (if supported) by using `.option` or `OPTIONS` (Scala and SQL API). These values should not be prefixed.

Note that the SAP Vora data source always requires these two parameters:

- `host`: The host the transaction coordinator is running on
- `port`: The port the transaction coordinator is listening on

By default, the SAP Vora Spark extensions installer sets the transaction coordinator host and port in your `spark-defaults.conf` file, so you don't have to specify them for each query. If you need to override them, either alter the layer at which they are specified, or choose a higher layer to provide your custom configuration.

For the sake of simplicity, this documentation assumes that both the `host` parameter and the `port` parameter have been set in either the `SparkContext` or `SparkSession`, so they don't need to be repeated for each command.

Related Information

SAP Vora Data Source Parameters [page 251]

14.2.2 Registering Tables

To register a table that already exists in SAP Vora, use the `table` option.

By using this option, the metadata of the table concerned is fetched from SAP Vora and a `DataFrame` pointing to that table is returned, as shown below:

```
val table1 =
  spark.read.format("sap.spark.vora")
    .option("table", "t")
    .load()
```

> **Note**
>
> Bear in mind that only relational tables are supported. This means that for SAP Vora graphs, for example, you either have to use the `query` option and run a relational query on top of the graph, as shown in the next section, or you have to create a Vora view on top of the graph that relationalizes that graph and then consume that view instead.

If the table concerned is located in a different namespace, you can specify the target namespace using the `namespace` option. By default, the namespace is `VORA`. For example:

```
val table2 =
  spark.read.format("sap.spark.vora")
    .option("table", "t")
    .option("namespace", "custom")
    .load()
```

After obtaining a `DataFrame`, you can either directly consume or transform it by calling `collect`, `show`, `map` or any other of the available methods on it, or you can also register it in the Spark catalog by using `createOrReplaceTempView`.

If your only interface is SQL (for instance, through a Thriftserver or third-party BI tool), you can do the reading and registering in the Spark catalog in one step, as follows:

```
CREATE TEMPORARY VIEW x USING sap.spark.vora
OPTIONS (
  table "t"
);
```

After that, you can access the table using the regular Spark SQL methods.

14.2.3 Querying Ad-hoc Views

When you want to extract specific information from SAP Vora without creating dedicated views, or when you want to run queries with Vora-specific SQL, you can use the `query` option.

The `query` option allows you to specify an ad-hoc query that is pushed down to SAP Vora. The SQL you specify in the `query` option has to be a valid query. DML statements are not supported.

For example:

```
val tables =
  spark.read.format("sap.spark.vora")
    .option("query", "SELECT * FROM SYS.TABLES")
    .load()
```

Again, if your only interface is SQL, you can create and register the ad-hoc view as follows:

```
CREATE TEMPORARY VIEW x USING sap.spark.vora
OPTIONS (
  query "SELECT * FROM SYS.TABLES"
);
```

14.2.4 Partitioning Result Sets

Some tables or queries tend to produce large result sets. If the result set is handled on a single Spark executor, this leads to high memory consumption and a computational bottleneck. By applying partitioning, you can spread the computation on your Spark cluster more efficiently.

The SAP Vora data source supports three partitioning strategies:

- Maximum partition size partitioning
- Maximum number of partitions partitioning
- Equidistant boundary range partitioning

Maximum partition size partitioning

To use this type of partitioning, you specify the maximum number of rows that you want to have per partition. If you have a query or table that yields 100 rows and you specified that you want to have 25 rows per partition, you will get 4 partitions.

In order to have a deterministic result set, you also have to specify one or more columns on which the resulting partitions are ordered.

The example belows loads a table with maximum partition size partitioning:

```
val maximumPartitionSizePartitioned =
  spark.read.format("sap.spark.vora")
    .option("table", "t")
    .option("maximumpartitionsize", 25)
    .option("orderby", "a") // 'a' is a column of the specified table 't'
    .load()
```

Maximum number of partitions partitioning

To use this partitioning strategy, you specify the maximum number of partitions you want to have the result set partitioned into. If you have a query with 100 rows and you specify that you want to have 4 partitions, those 100 rows will be split into 4 partitions with 25 rows each.

If the number of partitions is greater than the number of rows, the number of partitions will be set to the number of rows, so you'll have one partition per row.

As in the case of maximum partition size partitioning, you also need to specify one or more order columns to get deterministic partitions.

You can use this partitioning strategy as shown in the example below:

```
val maximumNumberOfPartitionsPartitioned =
  spark.read.format("sap.spark.vora")
    .option("table", "t")
    .option("numberofpartitions", 4)
    .option("orderby", "a")
    .load()
```

Equidistant boundary range partitioning

To use equidistant boundary range partitioning, you specify a column, an upper bound, and a lower bound, as well as the number of partitions you want to have.

By default, the specified column is also used to order the partitions, but you can also specify other columns for ordering purposes by using the `orderby` option.

If it is not possible to create the specified number of partitions, the number of partitions will be decreased accordingly. An example of such a case is when you specify 5 as the lower bound, 10 as the upper bound, and you want to have 8 partitions. Since the minimum stride is 1, the maximum possible number of partitions is 5.

You can use equidistant boundary range partitioning as shown in the following example:

```
val equidistantBoundaryRangePartitioned =
  spark.read.format("sap.spark.vora")
```

```
        .option("table", "t")
        .option("partitioningcolumn", "a")
        .option("numberofpartitions", "4")
        .option("lowerbound", 2)
        .option("upperbound", 10)
        .load()
```

14.2.5 Inserting Data into SAP Vora

The SAP Vora data source allows you to insert data into a table that already exists. The two main ways in which you can do this involve using either the `DataFrameWriter` API or the `InsertableRelation` trait.

DataFrameWriter API

To insert data using the `DataFrameWriter` API, you need to have a `DataFrame` that you want to insert. Once you have obtained one, you can call the `write` method on it, specifying the target data source, the `SaveMode`, and other options.

For SAP Vora, such an insert could look like this:

```
// Obtain CSV dataframe from already existing file
val dataFrame = spark.read.csv("/tmp/foo.csv")
dataFrame.write.format("sap.spark.vora").option("table", "t").save()
```

The target table has to exist, otherwise an exception is thrown.

The `SaveMode` is handled as follows:

- `SaveMode.Append`: Executes with the SQL INSERT statement
- `SaveMode.Overwrite`: Executes with the SQL UPSERT statement
- `SaveMode.Ignore`: Does nothing
- `SaveMode.ErrorIfExists`: Errors with an exception that the table exists

Insert Method

The other way to insert data is by either calling the `insert` method directly on a `VoraTableRelation` or by issuing the `INSERT INTO` Spark SQL statement. This requires the target table to be registered in the current `SparkSession`'s catalog. A workflow around `INSERT INTO` is shown in the example below:

```
CREATE TEMPORARY VIEW csv USING csv OPTIONS ( path "/tmp/foo.csv" );
CREATE TEMPORARY VIEW t USING sap.spark.vora OPTIONS ( table "t" );
INSERT INTO TABLE t SELECT * FROM csv;
```

14.2.6 SAP Vora Data Source Parameters

The overview below lists the parameters supported by the SAP Vora data source.

Parameter Name	Usage	Default	Example
host	Transaction coordinator host	50000	txc.example.org

Parameter Name	Usage	Default	Example
port	Transaction coordinator port	VORA	2202
namespace	Namespace for tables on SAP Vora	-	CUSTOM
table	Name of the table to be retrieved	-	t
query	Query to be executed	-	SELECT 1
numberofpartitions	Number of partitions	-	3
maximumpartitionsize	Maximum partition size	-	3
orderby	Columns to be used to order the partitions	-	foo, bar
partitioningcolumn	Partitioning column		foo
lowerbound	Lower partitioning bound		0
upperbound	Upper partitioning bound		10

Note that all parameters are case insensitive.

14.3 SAP HANA Data Source

Like the SAP Vora data source, the SAP HANA data source allows data to be queried in two ways; through table registration with the `table` option or by submitting ad-hoc queries using the `query` option. In addition, you can insert data into SAP HANA using the Spark APIs.

14.3.1 Configuring the SAP HANA Data Source

Like the SAP Vora data source, the SAP HANA data source supports a layered configuration approach, where different layers overwrite each other depending on their priority.

You can enter configuration values in the following locations, listed in ascending order of precedence (that is, the last location in the list represents the highest layer and overwrites all preceding layers):

- `SparkContext` configuration through your `spark-defaults.conf` file and through the `SparkConf` object during the construction of your `SparkSession`.
  This configuration has to be prefixed with the SAP HANA Spark configuration prefix `spark.sap.hana`. For example, in your `spark-defaults.conf` file:

  ```
  spark.sap.hana.host = hana.example.org
  ```

- `SparkSession` through `SparkSession.conf`. These values also have to be prefixed in the same way as described above.
- Per query (if supported) by using `.option` or `OPTIONS` (Scala and SQL API). These values should not be prefixed.

Note that the SAP HANA data source requires either the `securestorehanapath` and the`securestorekey` options pointing to a valid secure store if you have a secure store set up, or the `host`, `instance`, `port`, `user`, and `password` options.

Optionally, for the latter configuration (no secure store set up), you can also specify the tenant database that you want to connect to using the `tenantdatabase` option.

For the sake of simplicity, this documentation assumes that one of either of these basic configurations has been set in the `SparkContext` or `SparkSession`, so it doesn't need to be repeated for each command.

Related Information

SAP HANA Data Source Parameters [page 256]

14.3.2 Registering Tables

To register an existing SAP HANA table in Spark, you need to use the `table` option. By using this option, the metadata of the target table is fetched from SAP HANA and a `DataFrame` is created that points to that table.

The Spark code in Scala would look like this:

Sample Code

```
val hanaTable =
  spark.read.format("sap.spark.hana")
    .option("table", "t")
    .load()
```

If the table is located in a different schema or namespace, you can specify it using the `namespace` option, as follows:

```
val hanaTableFromDifferentNameSpace =
  spark.read.format("sap.spark.hana")
    .option("table", "t")
    .option("namespace", "data")
    .load()
```

After obtaining a `DataFrame`, you can either directly consume or transform it by calling `collect`, `show`, `map` or any other of the available methods on it, or you can also register it in the Spark catalog by using `createOrReplaceTempView`.

If your only interface is SQL (for instance, through a Thriftserver or third-party BI tool), you can do the reading and registering in the Spark catalog in one step, as follows:

```
CREATE TEMPORARY VIEW x USING sap.spark.hana
OPTIONS (
  table "t"
);
```

After that, you can access the table using the regular Spark SQL methods.

14.3.3 Querying Ad-hoc Views

When you want to extract specific information from SAP HANA without creating dedicated views, or when you want to run queries with HANA-specific SQL, you can use the `query` option.

The `query` option allows you to specify an ad-hoc query that is pushed down to SAP HANA. The SQL you specify in the `query` option has to be a valid query. DML statements are not supported.

For example:

```
val tables =
  spark.read.format("sap.spark.hana")
    .option("query", "SELECT * FROM SYS.TABLES")
    .load()
```

Again, if your only interface is SQL, you can create and register the ad-hoc view as follows:

```
CREATE TEMPORARY VIEW x USING sap.spark.hana
OPTIONS (
  query "SELECT * FROM SYS.TABLES"
);
```

14.3.4 Partitioning Result Sets

Some tables or queries tend to produce large result sets. If the result set is handled on a single Spark executor, this leads to high memory consumption and a computational bottleneck. By applying partitioning, you can spread the computation on your Spark cluster more efficiently.

The SAP HANA data source supports three partitioning strategies:

- Maximum partition size partitioning
- Maximum number of partitions partitioning
- Equidistant boundary range partitioning

Maximum partition size partitioning

To use this type of partitioning, you specify the maximum number of rows that you want to have per partition. If you have a query or table that yields 100 rows and you specified that you want to have 25 rows per partition, you will get 4 partitions.

The example belows loads a table with maximum partition size partitioning:

```
val maximumPartitionSizePartitioned =
  spark.read.format("sap.spark.hana")
    .option("table", "t")
    .option("maximumpartitionsize", 25)
    .load()
```

Maximum number of partitions partitioning

To use this partitioning strategy, you specify the maximum number of partitions you want to have the result set partitioned into. If you have a query with 100 rows and you specify that you want to have 4 partitions, those 100 rows will be split into 4 partitions with 25 rows each.

You can use this partitioning strategy as shown in the example below:

```
    val maximumNumberOfPartitionsPartitioned =
      spark.read.format("sap.spark.hana")
        .option("table", "t")
        .option("numberofpartitions", 4)
        .load()
```

Equidistant boundary range partitioning

To use equidistant boundary range partitioning, you specify a column, an upper bound, and a lower bound, as well as the number of partitions you want to have.

You can use equidistant boundary range partitioning as shown in the following example:

```
    val equidistantBoundaryRangePartitioned =
      spark.read.format("sap.spark.hana")
        .option("table", "t")
        .option("partitioningcolumn", "a")
        .option("numberofpartitions", "4")
        .option("lowerbound", 2)
        .option("upperbound", 10)
        .load()
```

14.3.5 Inserting Data into SAP HANA

The SAP HANA data source allows you to insert data into a table that already exists. The two main ways in which you can do this involve using either the `DataFrameWriter` API or the `InsertableRelation` trait.

DataFrameWriter API

To insert data using the `DataFrameWriter` API, you need to have a `DataFrame` that you want to insert. Once you have obtained one, you can call the `write` method on it, specifying the target data source, the `SaveMode`, and other options.

For SAP HANA, such an insert could look like this:

```
// Obtain CSV dataframe from already existing file
val dataFrame = spark.read.csv("/tmp/foo.csv")
dataFrame.write.format("sap.spark.hana").option("table", "t").save()
```

The target table has to exist, otherwise an exception is thrown.

The `SaveMode` is handled as follows:

- `SaveMode.Append`: Executes with the SQL INSERT statement
- `SaveMode.Overwrite`: Executes with the SQL UPSERT WITH PRIMARY KEY statement
- `SaveMode.Ignore`: Does nothing
- `SaveMode.ErrorIfExists`: Errors with an exception that the table exists

Insert Method

The other way to insert data is by either calling the `insert` method directly on a `HanaTableRelation` or by issuing the `INSERT INTO` Spark SQL statement. This requires the target table to be registered in the current `SparkSession`'s catalog. A workflow around `INSERT INTO` is shown in the example below:

```
CREATE TEMPORARY VIEW csv USING csv OPTIONS ( path "/tmp/foo.csv" );
CREATE TEMPORARY VIEW t USING sap.spark.hana OPTIONS ( table "t" );
INSERT INTO TABLE t SELECT * FROM csv;
```

14.3.6 Using SAP HANA Secure Store

The SAP HANA secure user store (hdbuserstore) is a tool installed with the SAP HANA client. It is used to store the credentials of SAP HANA systems securely on the client so that client applications can connect to SAP HANA without users having to enter this information.

The SAP HANA data source is able to use the SAP HANA secure user store to connect to SAP HANA. To that end, the SAP HANA secure user store files need to be distributed to the same folder on all nodes on which Spark runs. The default value of this folder is `$HOME/.hdb`, where `$HOME` is the home folder of the corresponding Spark user. If a different folder path needs to be used, the path can be set using the Spark parameter `securestorehanapath`. The SAP HANA data source reads the credentials from secure store files indexed with a key, which is specified with the Spark parameter `securestorekey`.

For example:

```
SHOW TABLES USING com.sap.spark.hana
OPTIONS (
securestorekey "mykey",
securestorehanapath "/home/vora/.hdb",
dbschema "$dbSchema",
tablePattern "$pattern"
)
```

> **Note**
>
> The secure store files, `SSFS_HDB.DAT` and `SSFS_HDB.KEY`, need to be protected with strict file permissions so that only the user who runs the Spark job can access them. If Spark jobs are run in YARN modes, the user permission needs to be handled accordingly. For instance, Spark jobs are run with the YARN user in the executors unless YARN is Kerberized and configured accordingly.

14.3.7 SAP HANA Data Source Parameters

The overview below lists the parameters supported by the SAP HANA data source.

Parameter Name	Usage	Default	Example
host	SAP HANA host	-	hana.example.org

Parameter Name	Usage	Default	Example
port	Port for connecting to SAP HANA	-	01
instanceid	SAP HANA instance ID	-	00
user	User on the SAP HANA instance	-	foo
password	Password on the SAP HANA instance	-	secret
securestorekey	Secure store key	-	key
securestorehanapath	Path to the SAP HANA secure store	-	/path/to/store
namespace	Namespace for tables on SAP Vora	-	CUSTOM
table	Name of the table to be retrieved	-	t
query	Query to be executed	-	SELECT 1
numberofpartitions	Number of partitions	-	3
maximumpartitionsize	Maximum partition size	-	3
partitioningcolumn	Partitioning column	-	foo
lowerbound	Lower partitioning bound	-	0
upperbound	Upper partitioning bound	-	10

Note that all parameters are case insensitive.

Important Disclaimers and Legal Information

Coding Samples

Any software coding and/or code lines / strings ("Code") included in this documentation are only examples and are not intended to be used in a productive system environment. The Code is only intended to better explain and visualize the syntax and phrasing rules of certain coding. SAP does not warrant the correctness and completeness of the Code given herein, and SAP shall not be liable for errors or damages caused by the usage of the Code, unless damages were caused by SAP intentionally or by SAP's gross negligence.

Accessibility

The information contained in the SAP documentation represents SAP's current view of accessibility criteria as of the date of publication; it is in no way intended to be a binding guideline on how to ensure accessibility of software products. SAP in particular disclaims any liability in relation to this document. This disclaimer, however, does not apply in cases of willful misconduct or gross negligence of SAP. Furthermore, this document does not result in any direct or indirect contractual obligations of SAP.

Gender-Neutral Language

As far as possible, SAP documentation is gender neutral. Depending on the context, the reader is addressed directly with "you", or a gender-neutral noun (such as "sales person" or "working days") is used. If when referring to members of both sexes, however, the third-person singular cannot be avoided or a gender-neutral noun does not exist, SAP reserves the right to use the masculine form of the noun and pronoun. This is to ensure that the documentation remains comprehensible.

Internet Hyperlinks

The SAP documentation may contain hyperlinks to the Internet. These hyperlinks are intended to serve as a hint about where to find related information. SAP does not warrant the availability and correctness of this related information or the ability of this information to serve a particular purpose. SAP shall not be liable for any damages caused by the use of related information unless damages have been caused by SAP's gross negligence or willful misconduct. All links are categorized for transparency (see: https://help.sap.com/viewer/disclaimer).

go.sap.com/registration/contact.html