Cloudera Runtime 7.0.0

Integrating Apache Hive with Spark and BI

Date published: 2019-08-21
Date modified:

https://docs.cloudera.com/

Legal Notice

Contents

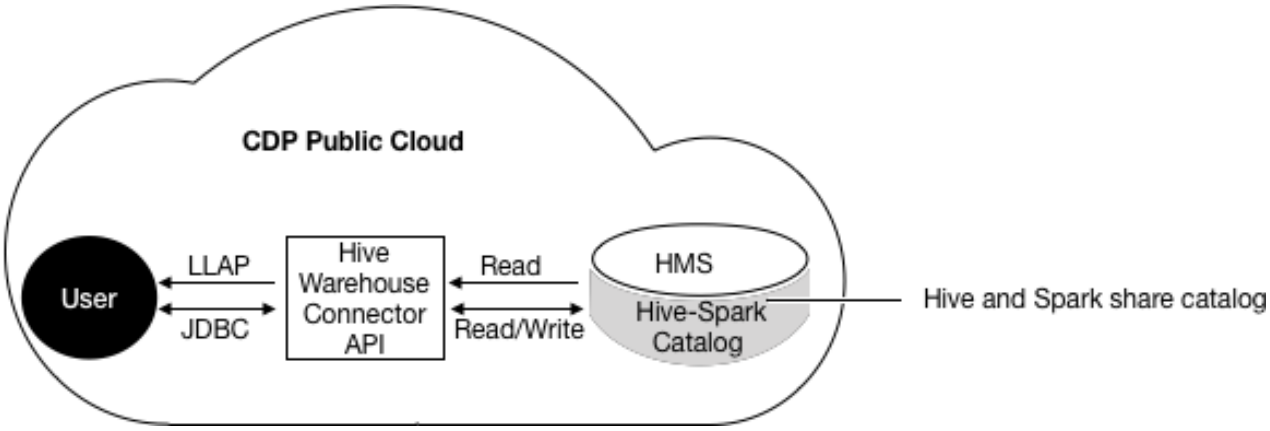
Hive Warehouse Connector for accessing Apache Spark data

The Hive Warehouse Connector (HWC) is a Spark library/plugin that is launched with the Spark app. You can use the Hive Warehouse Connector API to access any managed Hive table from Spark. Apache Ranger and the HiveWarehouseConnector library provide row and column, fine-grained access to the data.

In CDP Public Cloud and CDP Data Center, Spark and Hive share a catalog in the Hive metastore (HMS).

CDP Public Cloud Processing

In CDP Public Cloud, to read ACID, or other Hive-managed tables, from Spark using low-latency analytical processing (LLAP) is recommended.

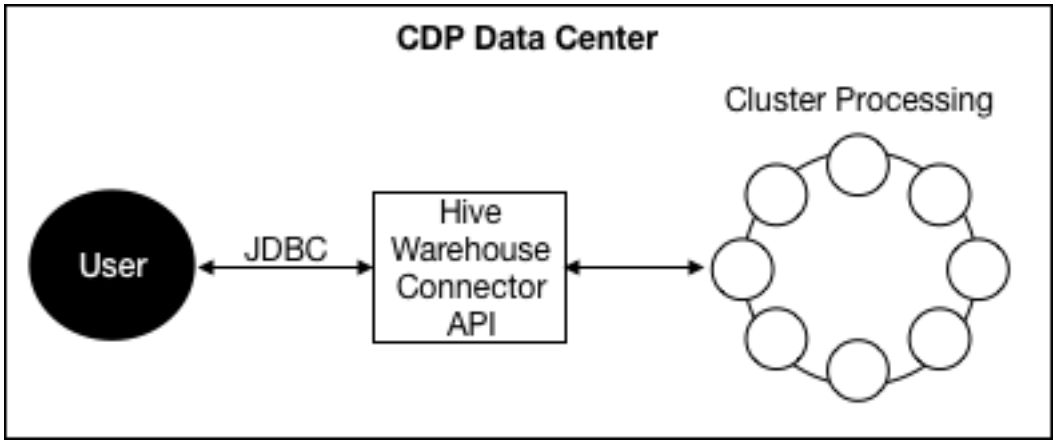


To write to ACID, or other managed tables, from Spark you must use JDBC. HWC is not required to access external tables from Spark. Spark can directly access Hive external tables using spark sql instead of HWC. However, HWC also supports reading external tables.

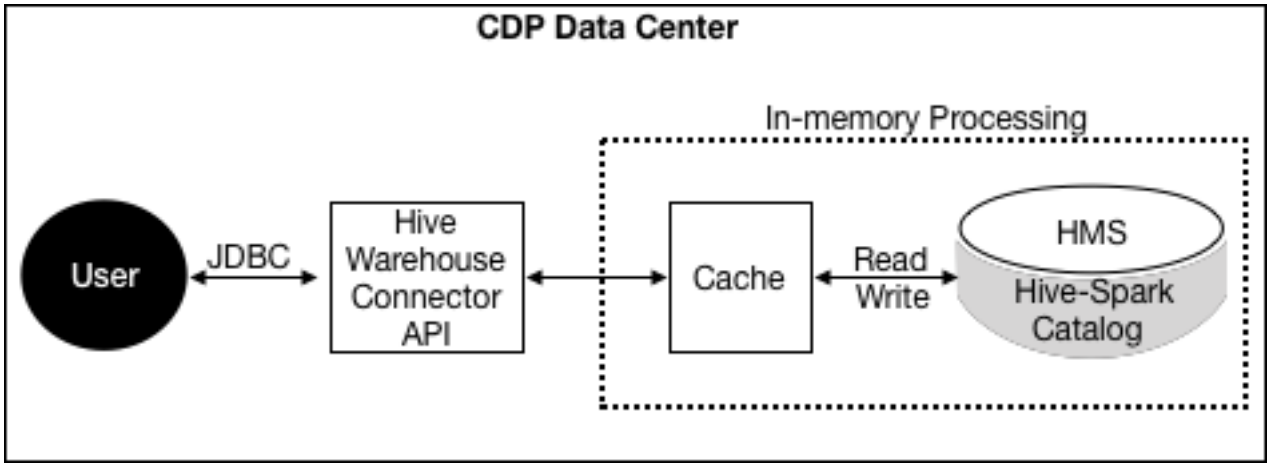
Using the HWC, you can read and write Apache Spark DataFrames and Streaming DataFrames.

CDP Data Center Processing

In CDP Data Center, HWC is available as a technical preview. HWC processes data through the Hive JDBC driver (download the latest 2.6.x version). LLAP is not available in CDP Data Center. You can choose cluster processing (recommended) or in-memory client processing, depending on the size of your resultset and latency requirements. Cluster processing, described later, is recommended for ingesting large data sets.



Configure the JDBC mode for in-memory processing. HWC stores the entire resultset in an in-memory cache for fast processing. The capacity of the in-memory cache is limited based on the capacity of the Spark driver/client application.



HWC Limitations

- HWC supports reading tables in any format, but currently supports writing tables in ORC format only.
- Table stats (basic stats and column stats)are not generated when you write a DataFrame to Hive.
- The spark thrift server is not supported.
- The Hive Union data type is not supported.
- Transaction semantics of Spark RDDs are not ensured when using Spark Direct Reader to read ACID tables.
- When the HWC API save mode is overwrite, writes are limited.

  You cannot read from and overwrite the same table. If your query accesses only one table and you try to overwrite that table using an HWC API write method, a deadlock state might occur. Do not attempt this operation.

  Example: Operation Not Supported

```
scala> val df = hive.executeQuery("select * from t1")
scala> df.write.format("com.hortonworks.spark.sql.hive.llap.HiveWarehouseC
onnector").mode("overwrite").option("table", "t1").save
```

Supported applications and operations

The Hive Warehouse Connector supports the following applications:

- Spark shell
- PySpark
- The spark-submit script

The following list describes a few of the operations supported by the Hive Warehouse Connector:

- Describing a table
- Creating a table in ORC using .createTable() or in any format using .executeUpdate()
- Writing to a table in ORC format
- Selecting Hive data and retrieving a DataFrame
- Writing a DataFrame to a Hive-managed ORC table in batch
- Executing a Hive update statement
- Reading table data, transforming it in Spark, and writing it to a new Hive table
- Writing a DataFrame or Spark stream to Hive using HiveStreaming
- Partitioning data when writing a DataFrame

Related Information

HMS storage

Apache Spark-Apache Hive connection configuration

In Spark, you can use the Hive Warehouse Connector (HWC) for accessing ACID table data in Hive. You need to understand the workflow and service changes involved in such access.

Configuring the HWC mode for reads

The HWC runs in the following modes for reading Hive-managed tables:

- LLAP
  - true (not supported in CDP Data Center)
  - false
- JDBC
  - cluster
  - client

You set the spark.datasource.hive.warehouse.read.via.llap property to true or false to configure LLAP mode on or off. You set spark.datasource.hive.warehouse.read.jdbc.mode to cluster or client to configure JDBC mode.

Configuring the HWC connection in CDP Public Cloud

Configure the following software and property settings for connecting Spark and Hive using the HiveWarehouseConnector library in CDP Public Cloud:

- Spark 2.4.x
- Low-latency analytical processing (LLAP) is recommended, or alternatively, use the Hive JDBC database driver
- Set spark.datasource.hive.warehouse.read.via.llap=true (recommended).

  Alternatively, for a non-llap cluster, set the following properties:
  - spark.datasource.hive.warehouse.read.via.llap=false
  - spark.datasource.hive.warehouse.read.jdbc.mode=cluster.

The Hive Warehouse Connector is required for certain tasks in CDP Public Cloud. Low-latency analytical processing (LLAP) is recommended for reading ACID, or other Hive-managed tables, from Spark. You do not need LLAP to write to ACID, or other managed tables, from Spark. CDP Data Center supports JDBC that you use with HWC in lieu of LLAP. JDBC mode works well for small datasets only.

The HWC library internally uses the Hive Streaming API and LOAD DATA Hive commands to write the data. You do not need LLAP to access external tables from Spark with caveats shown in the following table.

Table 1: Spark Compatibility: CDP Public Cloud

Tasks	HWC Required	Recommended HWC Mode	Other Requirement/Comments
Read Hive managed tables from Spark	Yes	LLAP mode=true	Ranger ACLs enforced.*
Write Hive managed tables from Spark	Yes	N/A	Ranger ACLs enforced.*
Read Hive external tables from Spark	No	N/A unless HWC is used, then LLAP mode=true	Ranger ACLs not enforced.
Write Hive external tables from Spark	No	N/A	Ranger ACLs enforced.

* Ranger column level security or column masking is supported for each access pattern when you use HWC.

Configuring the HWC connection in CDP Data Center

You configure the spark.datasource.hive.warehouse.read.jdbc.mode property as described below. The following software and property settings are required for connecting Spark and Hive using the HiveWarehouseConnector library in CDP Data Center:

- Spark 2.4.x
- Hive JDBC database driver

  Download the Hive JDBC database driver from the Cloudera Downloads page.
- Set spark.datasource.hive.warehouse.read.jdbc.mode=cluster (recommended). Alternatively, set this property to client if you expect your resultset to fit in memory.

Table 2: Spark Compatibility: CDP Data Center

Tasks	HWC Required	Recommended HWC Mode	Other Requirement/Comments
Read Hive managed tables from Spark	Yes	JDBC mode=cluster	Ranger ACLs enforced.*
Write Hive managed tables from Spark	Yes	N/A	Ranger ACLs enforced.*
Read Hive external tables from Spark	No	N/A	Ranger ACLs not enforced.
Write Hive external tables from Spark	No	N/A	Ranger ACLs enforced.

* Ranger column level security or column masking is supported for each access pattern when you use HWC.

Spark on a Kerberized YARN cluster in CDP Data Center

In Spark client mode on a kerberized Yarn cluster, set the following property: spark.sql.hive.hiveserver2.jdbc.url.principal. This property must be equal to hive.server2.authentication.kerberos.principal.

In Spark cluster mode on a kerberized YARN cluster, set the following property:

- Property: spark.security.credentials.hiveserver2.enabled
- Description: Use Spark ServiceCredentialProvider and set equal to a boolean, such as true
- Comment: true by default

Related Information

HMS storage

Submit a Hive Warehouse Connector Scala or Java application

You can submit an app based on the HiveWarehouseConnector library to run on Spark Shell, PySpark, and spark-submit.

Procedure

1. Locate the hive-warehouse-connector-assembly jar in the /hive_warehouse_connector/ directory.
2. Add the connector jar to the app submission using the --jars option.

```
spark-shell --jars <path to jars>/hive_warehouse_connector/hive-warehouse-connector-assembly-<version>.jar
```

Related Information

HMS storage

Submit a Hive Warehouse Connector Python app

You can submit a Python app based on the HiveWarehouseConnector library by submitting a Scala or Java application, and then adding a Python package.

Procedure

1. Locate the hive-warehouse-connector-assembly jar in the /hive_warehouse_connector/ directory.
2. Add the connector jar to the app submission using the --jars option.

```
pyspark --jars <path to jars>/hive_warehouse_connector/hive-warehouse-co
nnector-assembly-<version>.jar
```

3. Locate the pyspark_hwc zip package in the /hive_warehouse_connector/ directory.
4. Add the Python package for the connector to the app submission.

```
pyspark --jars <path to jars>/hive_warehouse_connector/hive-warehouse-co
nnector-assembly-<version>.jar --py-files <path>/hive_warehouse_connector/
pyspark_hwc-<version>.zip
```

Related Information

HMS storage

Hive Warehouse Connector supported types

The Hive Warehouse Connector maps most Apache Hive types to Apache Spark types and vice versa, but there are a few exceptions.

Spark-Hive supported types mapping

The following types are supported by the HiveWareHouseConnector library:

Spark Type	Hive Type
ByteType	TinyInt
ShortType	SmallInt
IntegerType	Integer
LongType	BigInt
FloatType	Float
DoubleType	Double
DecimalType	Decimal
StringType*	String, Varchar*
BinaryType	Binary
BooleanType	Boolean
TimestampType**	Timestamp**
DateType	Date
ArrayType	Array
StructType	Struct

Notes:

* StringType (Spark) and String, Varchar (Hive)

A Hive String or Varchar column is converted to a Spark StringType column. When a Spark StringType column has maxLength metadata, it is converted to a Hive Varchar column; otherwise, it is converted to a Hive String column.

** Timestamp (Hive)

The Hive Timestamp column loses submicrosecond precision when converted to a Spark TimestampType column because a Spark TimestampType column has microsecond precision, while a Hive Timestamp column has nanosecond precision.

Hive timestamps are interpreted as UTC. When reading data from Hive, timestamps are adjusted according to the local timezone of the Spark session. For example, if Spark is running in the America/New_York timezone, a Hive timestamp 2018-06-21 09:00:00 is imported into Spark as 2018-06-21 05:00:00 due to the 4-hour time difference between America/New_York and UTC.

Spark-Hive unsupported types

Spark Type	Hive Type
CalendarIntervalType	Interval
N/A	Char
MapType	Map
N/A	Union
NullType	N/A
TimestampType	Timestamp With Timezone

Related Information

HMS storage

HiveWarehouseSession API operations

As a Spark developer, you execute queries to Hive using the HiveWarehouseSession API that supports Scala, Java, and Python. In Spark source code, you create an instance of HiveWarehouseSession. Results are returned as a DataFrame to Spark.

Import statements and variables

The following string constants are defined by the API:

- HIVE_WAREHOUSE_CONNECTOR
- DATAFRAME_TO_STREAM
- STREAM_TO_STREAM

Assuming spark is running in an existing SparkSession, use this code for imports:

- Scala

```
import com.hortonworks.hwc.HiveWarehouseSession
import com.hortonworks.hwc.HiveWarehouseSession._
val hive = HiveWarehouseSession.session(spark).build()
```

- Java

```
import com.hortonworks.hwc.HiveWarehouseSession;
import static com.hortonworks.hwc.HiveWarehouseSession.*;
HiveWarehouseSession hive = HiveWarehouseSession.session(spark).build();
```

- Python

```
from pyspark_llap import HiveWarehouseSession
hive = HiveWarehouseSession.session(spark).build()
```

Executing queries

HWC supports three methods for executing queries:

- .sql()
  - Executes queries in any HWC mode.
  - Consistent with the Spark sql interface.
  - Masks the internal implementation based on cluster type.
- .execute()
  - Required for executing queries if spark.datasource.hive.warehouse.read.jdbc.mode = client (default = cluster).
  - Uses a driver side JDBC connection.
  - Provided for backward compatibility where the method defaults to reading in JDBC client mode irrespective of the value of JDBC client or cluster mode configuration.
  - Recommended for catalog queries.
- .executeQuery()
  - Executes queries, except catalog queries, in LLAP mode (spark.datasource.hive.warehouse.read.via.llap= true)
  - If LLAP is not enabled in the cluster, .executeQuery() does not work. CDP Data Center does not support LLAP.
  - Provided for backward compatibility.

Related Information

HMS storage

Catalog operations

Catalog operations include creating, dropping, and describing a Hive database and table from Spark.
Three methods of executing catalog operations are supported: .sql (recommended), .execute()
(spark.datasource.hive.warehouse.read.jdbc.mode = client), or .executeQuery() for backward compatibility in LLAP mode.

Catalog operations

- Set the current database for unqualified Hive table references

  hive.setDatabase(<database>)
- Execute a catalog operation and return a DataFrame

  hive.execute("describe extended web_sales").show()
- Show databases

  hive.showDatabases().show(100)
- Show tables for the current database

  hive.showTables().show(100)
- Describe a table

  hive.describeTable(<table_name>).show(100)
- Create a database

  hive.createDatabase(<database_name>,<ifNotExists>)

- Create an ORC table

```
hive.createTable("web_sales").ifNotExists().column("sold_time_sk", "bigi
nt").column("ws_ship_date_sk", "bigint").create()
```

  See the CreateTableBuilder interface section below for additional table creation options. You can also create Hive tables using hive.executeUpdate.
- Drop a database

  hive.dropDatabase(<databaseName>, <ifExists>, <useCascade>)
- Drop a table

  hive.dropTable(<tableName>, <ifExists>, <usePurge>)

Related Information

HMS storage

Read and write operations

The HiveWarehouseSession API supports reading and writing hive tables from spark. HWC supports writing to ORC tables only. You can update statements and write DataFrames to partitioned Hive tables, perform batch writes, and use HiveStreaming.

Read operations

Execute a Hive SELECT query and return a DataFrame.

hive.sql("select * from web_sales")

HWC supports push-downs of DataFrame filters and projections applied to .sql().

Alternatively, you can use .execute or .executeQuery as previously described.

Execute a Hive update statement

Execute CREATE, UPDATE, DELETE, INSERT, and MERGE statements in this way:

hive.executeUpdate("ALTER TABLE old_name RENAME TO new_name")

Write a DataFrame to Hive in batch

This operation uses LOAD DATA INTO TABLE.

Java/Scala:

```
df.write.format(HIVE_WAREHOUSE_CONNECTOR).option("table", <tableName>).s
ave()
```

Python:

```
df.write.format(HiveWarehouseSession().HIVE_WAREHOUSE_CONNECTOR).option("tab
le", &tableName>).save()
```

Write a DataFrame to Hive, specifying partitions

HWC follows Hive semantics for overwriting data with and without partitions and is not affected by the setting of spark.sql.sources.partitionOverwriteMode to static or dynamic. This behavior mimics the latest Spark Community trend reflected in Spark-20236 (link below).

Java/Scala:

```
df.write.format(HIVE_WAREHOUSE_CONNECTOR).option("table", <tableName>).optio
n("partition", <partition_spec>).save()
```

Python:

```
df.write.format(HiveWarehouseSession().HIVE_WAREHOUSE_CONNECTOR).option("tab
le", &tableName>).option("partition", <partition_spec>).save()
```

Where <partition_spec> is in one of the following forms:

- option("partition", "c1='val1',c2=val2") // static
- option("partition", "c1='val1',c2") // static followed by dynamic
- option("partition", "c1,c2") // dynamic

Depending on the partition spec, HWC generates queries in one of the following forms for writing data to Hive.

- No partitions specified = LOAD DATA
- Only static partitions specified = LOAD DATA...PARTITION
- Some dynamic partition present = CREATE TEMP TABLE + INSERT INTO/OVERWRITE query.

Note: Writing static partitions is faster than writing dynamic partitions.

Write a DataFrame to Hive using HiveStreaming

When using HiveStreaming to write a DataFrame to Hive or a Spark Stream to Hive, you need to escape any commas in the stream, as shown in Use the Hive Warehouse Connector for Streaming (link below).

Java/Scala:

```
//Using dynamic partitioning
df.write.format(DATAFRAME_TO_STREAM).option("table", <tableName>).save()

//Or, writing to a static partition
df.write.format(DATAFRAME_TO_STREAM).option("table", <tableName>).option("p
artition", <partition>).save()
```

Python:

```
//Using dynamic partitioning
df.write.format(HiveWarehouseSession().DATAFRAME_TO_STREAM).option("table",
 <tableName>).save()

//Or, writing to a static partition
df.write.format(HiveWarehouseSession().DATAFRAME_TO_STREAM).option("table",
<tableName>).option("partition", <partition>).save()
```

Write a Spark Stream to Hive using HiveStreaming

Java/Scala:

```
stream.writeStream.format(STREAM_TO_STREAM).option("table", "web_sales").sta
rt()
```

Python:

```
stream.writeStream.format(HiveWarehouseSession().STREAM_TO_STREAM).option("t
able", "web_sales").start()
```

Related Information

HMS storage
SPARK-20236

Close HiveWarehouseSession operations

Spark can invoke operations, such as cache(), persist(), and rdd(), on a DataFrame you obtain from running a HiveWarehouseSession .table() or .sql() (or alternatively, .execute() or .executeQuery()). The Spark operations can lock Hive resources. You can release any locks and resources by calling the HiveWarehouseSession close().

About this task

Calling close() invalidates the HiveWarehouseSession instance and you cannot perform any further operations on the instance.

Procedure

Call close() when you finish running all other operations on the instance of HiveWarehouseSession.

```
import com.hortonworks.hwc.HiveWarehouseSession
import com.hortonworks.hwc.HiveWarehouseSession._
val hive = HiveWarehouseSession.session(spark).build()
hive.setDatabase("tpcds_bin_partitioned_orc_1000")
val df = hive.sql("select * from web_sales")
. . .  //Any other operations
.close()
```

You can also call close() at the end of an iteration if the application is designed to run in a microbatch, or iterative, manner that does not need to share previous states.

No more operations can occur on the DataFrame obtained by table() or sql() (or alternatively, .execute() or .execute Query()).

Use the Hive Warehouse Connector for streaming

When using HiveStreaming to write a DataFrame to Hive or a Spark Stream to Hive, you need to escape any commas in the stream because the Hive Warehouse Connector uses the commas as the field delimiter.

Procedure

Change the value of the default delimiter property escape.delim to a backslash that the Hive Warehouse Connector uses to write streams to mytable.
ALTER TABLE mytable SET TBLPROPERTIES ('escape.delim' = '\\');

Related Information

HMS storage

Hive Warehouse Connector API Examples

You can create the DataFrame from any data source and include an option to write the DataFrame to a Hive table. When you write the DataFrame, the Hive Warehouse Connector creates the Hive table if it does not exist.

Write a DataFrame from Spark to Hive example

You specify one of the following Spark SaveMode modes to write a DataFrame to Hive:

- Append
- ErrorIfExists
- Ignore
- Overwrite

In Overwrite mode, HWC does not explicitly drop and recreate the table. HWC queries Hive to overwrite an existing table using LOAD DATA...OVERWRITE or INSERT OVERWRITE...

The following example uses Append mode.

```
df = //Create DataFrame from any source

val hive = com.hortonworks.spark.sql.hive.llap.HiveWarehouseBuilder.session
(spark).build()

df.write.format(HIVE_WAREHOUSE_CONNECTOR)
.mode("append")
.option("table", "my_Table")
.save()
```

ETL example (Scala)

Read table data from Hive, transform it in Spark, and write to a new Hive table.

```
import com.hortonworks.hwc.HiveWarehouseSession
import com.hortonworks.hwc.HiveWarehouseSession._
val hive = HiveWarehouseSession.session(spark).build()
hive.setDatabase("tpcds_bin_partitioned_orc_1000")
val df = hive.sql("select * from web_sales")
df.createOrReplaceTempView("web_sales")
hive.setDatabase("testDatabase")
hive.createTable("newTable")
.ifNotExists()
.column("ws_sold_time_sk", "bigint")
.column("ws_ship_date_sk", "bigint")
.create()
sql("SELECT ws_sold_time_sk, ws_ship_date_sk FROM web_sales WHERE ws_sold_
time_sk > 80000)
.write.format(HIVE_WAREHOUSE_CONNECTOR)
.mode("append")
.option("table", "newTable")
.save()
```

Related Information

HMS storage

Hive Warehouse Connector Interfaces

The HiveWarehouseSession, CreateTableBuilder, and MergeBuilder interfaces present available HWC operations.

HiveWarehouseSession interface

```
package com.hortonworks.hwc;

public interface HiveWarehouseSession {

//Execute Hive SELECT query and return DataFrame (recommended)
Dataset<Row> sql(String sql);
//Execute Hive SELECT query and return DataFrame in JDBC client mode
//Execute Hive catalog-browsing operation and return DataFrame
Dataset<Row> execute(String sql);

//Execute Hive SELECT query and return DataFrame in LLAP mode
Dataset<Row> executeQuery(String sql);

//Execute Hive update statement
```

```
boolean executeUpdate(String sql);

//Reference a Hive table as a DataFrame
Dataset<Row> table(String sql);

//Return the SparkSession attached to this HiveWarehouseSession
SparkSession session();

//Set the current database for unqualified Hive table references
void setDatabase(String name);

/**
* Helpers: wrapper functions over execute or executeUpdate
*/

//Helper for show databases
Dataset<Row> showDatabases();

//Helper for show tables
Dataset<Row> showTables();

//Helper for describeTable
Dataset<Row> describeTable(String table);

//Helper for create database
void createDatabase(String database, boolean ifNotExists);

//Helper for create table stored as ORC
CreateTableBuilder createTable(String tableName);

//Helper for drop database
void dropDatabase(String database, boolean ifExists, boolean cascade);

//Helper for drop table
void dropTable(String table, boolean ifExists, boolean purge);

//Helper for merge query
MergeBuilder mergeBuilder();

//Closes the HWC session. Session cannot be reused after being closed.
void close();
}
```

CreateTableBuilder interface

```
package com.hortonworks.hwc;

public interface CreateTableBuilder {

//Silently skip table creation if table name exists
CreateTableBuilder ifNotExists();

//Add a column with the specific name and Hive type
//Use more than once to add multiple columns
CreateTableBuilder column(String name, String type);

//Specific a column as table partition
//Use more than once to specify multiple partitions
CreateTableBuilder partition(String name, String type);

//Add a table property
//Use more than once to add multiple properties
```

```
CreateTableBuilder prop(String key, String value);

//Make table bucketed, with given number of buckets and bucket columns
CreateTableBuilder clusterBy(long numBuckets, String ... columns);

//Creates ORC table in Hive from builder instance
void create();
}
```

MergeBuilder interface

```
package com.hortonworks.hwc;

public interface MergeBuilder {

//Specify the target table to merge
MergeBuilder mergeInto(tring targetTable, String alias);

//Specify the source table or expression, such as (select * from some_table)
// Enclose expression in braces if specified.
MergeBuilder using(String sourceTableOrExpr, String alias);

//Specify the condition expression for merging
MergeBuilder on(String expr);

//Specify fields to update for rows affected by merge condition and match
Expr
MergeBuilder whenMatchedThenUpdate(String matchExpr, String... nameValuePa
irs);

//Delete rows affected by the merge condition and matchExpr
MergeBuilder whenMatchedThenDelete(String matchExpr);

//Insert rows into target table affected by merge condition and matchExpr
MergeBuilder whenNotMatchedInsert(String matchExpr, String... values);

//Execute the merge operation
void merge();
}
```

Related Information

HMS storage

Connecting Hive to BI tools using a JDBC/ODBC driver

To query, analyze, and visualize data stored in Data Hub or in the CDP Private Cloud Base using drivers provided by Cloudera, you connect Apache Hive to Business Intelligence (BI) tools.

About this task

How you connect to Hive depends on a number of factors: the location of Hive inside or outside the cluster, the HiveServer deployment, the type of transport, transport-layer security, and authentication. HiveServer is the server interface that enables remote clients to execute queries against Hive and retrieve the results using a JDBC or ODBC connection.

Before you begin

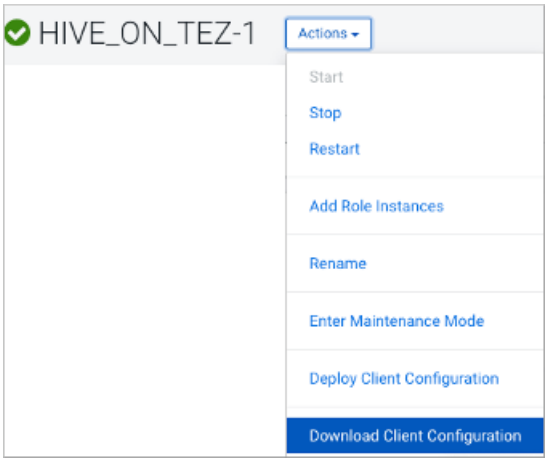
- Choose a Hive authorization model.

- Configure authenticated users for querying Hive through JDBC or ODBC driver. For example, set up a Ranger policy.

Procedure

1. Obtain the Hive database driver in one of the following ways:
   - For an ODBC or JDBC connection CDP Private Cloud Base: Get the Cloudera ODBC or JDBC driver from the Cloudera Downloads page.
   - For a JDBC connection in CDP Public Cloud: Using the CDW service, in a Virtual Warehouse in the CDW service, select Hive, and from the more options menu, click Download JDBC JAR to download to Apache Hive JDBC jar.

     Using the Data Hub service, download and extract the Apache Hive JDBC driver from /opt/cloudera/parcels/<version>/jars/hive-jdbc-3<version>-standalone.jar or use the <version> to look up the Apache Hive JDBC driver version and download it from the Central Maven repository.
2. Depending on the type of driver you obtain, proceed as follows:
   - ODBC driver: follow instructions on the ODBC driver download site, and skip the rest of the steps in this procedure.
   - JDBC driver: add the driver to the classpath of your JDBC client, such as Tableau. For example, check the client documentation about where to put the driver.
3. Find the JDBC URL for HiveServer using one of a number methods. For example:
   - Using the CDW service in a Virtual Warehouse, from the options menu of your Virtual Warehouse, click Copy JDBC URL.
   - In Cloudera Manager (CM), click Clusters Hive click Actions, and select Download Client Configuration.

     Unpack hive_on_tez-clientconfig.zip, open beeline-site.xml, and copy the value of beeline.hs2.jdbc.url.hive_on_tez . This value is the JDBC URL. For example

     ```
     jdbc:hive2://my_hiveserver.com:2181/;serviceDiscoveryMode=zooKeeper; \
        zooKeeperNamespace=hiveserver2
     ```
4. In the BI tool, such as Tableau, configure the JDBC connection using the JDBC URL and driver class name, org.apache.hive.jdbc.HiveDriver.

Get the JDBC or ODBC driver

You download the Apache Hive JDBC driver, navigate to the installed JDBC driver, or you download the Simba ODBC driver.

About this task

Cloudera provides the Apache Hive JDBC and Simba ODBC drivers as an add-on. When you start the Hive shell, the JDBC JAR version to use appears in the output:

```
. . .
Connected to: Apache Hive (version 3.1.2000.7.0.3.0-117)
Driver: Hive JDBC (version 3.1.2000.7.0.3.0-117)
Transaction isolation: TRANSACTION_REPEATABLE_READ
Beeline version 3.1.2000.7.0.3.0-117 by Apache Hive
0: jdbc:hive2://172.27.133.67:10000/default>
```

Procedure

1. Get the driver.
   - Using the version number of Apache Hive in your cluster, search for and download the standalone Apache Hive JDBC driver from the Central Maven repository.
   - Download the Simba ODBC driver for Hive in CDP from the Cloudera Downloads page. Skip the rest of the steps in this procedure and follow ODBC driver installation instructions on the same downloads page.
2. Give clients outside of the cluster who want to use the JDBC driver in HTTP and HTTPS modes access to required JARS:
   - /opt/cloudera/parcels/CDH-<version>/jars/hive-jdbc-<version>-standalone.jar
   - hadoop-common.jar

     For example, /opt/cloudera/parcels/CDH-7.0.0-1.cdh7.0.0.p0.1316065/lib/hadoop/hadoop-common.jar
   - hadoop-auth.jar

     Same location as hadoop-common.jar

Integrate Hive and a BI tool

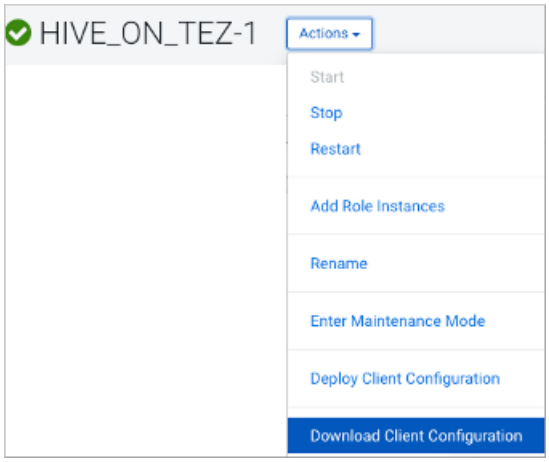
Before you begin

You have downloaded, or otherwise put, the JDBC/ODBC driver on your HiveServer cluster.

Procedure

1. Depending on the type of driver you obtain, proceed as follows:
   - ODBC driver: follow instructions on the ODBC driver download site, and skip the rest of the steps in this procedure.
   - JDBC driver: add the driver to the classpath of your JDBC client, such as Tableau. For example, check the client documentation about where to put the driver.

2. In Cloudera Manager (CM), click Clusters Hive click Actions, and select Download Client Configuration.



3. Unpack hive_on_tez-clientconfig.zip, open beeline-site.xml, and copy the value of beeline.hs2.jdbc.url.hive_on _tez. This value is the JDBC URL.
   For example

```
jdbc:hive2://my_hiveserver.com:2181/;serviceDiscoveryMode=zooKeeper; \
                        zooKeeperNamespace=hiveserver2
```

4. In the BI tool, such as Tableau, configure the JDBC connection using the JDBC URL and driver class name, org. apache.hive.jdbc.HiveDriver.

Specify the JDBC connection string

You construct a JDBC URL to connect Hive to a BI tool.

About this task

In CDP Private Cloud Base, if HiveServer runs within the Hive client (embedded mode), not as a separate process, the URL in the connection string does not need a host or port number to make the JDBC connection. If HiveServer does not run within your Hive client, the URL must include a host and port number because HiveServer runs as a separate process on the host and port you specify. The JDBC client and HiveServer interact using remote procedure calls using the Thrift protocol. If HiveServer is configured in remote mode, the JDBC client and HiveServer can use either HTTP or TCP-based transport to exchange RPC messages.

Procedure

1. Create a minimal JDBC connection string for connecting Hive to a BI tool.

   - Embedded mode: Create the JDBC connection string for connecting to Hive in embedded mode.
   - Remote mode: Create a JDBC connection string for making an unauthenticated connection to the Hive default database on the localhost port 10000.

   Embedded mode: "jdbc:hive://"
   Remote mode: "jdbc:hive://myserver:10000/default", "", "");

2. Modify the connection string to change the transport mode from TCP (the default) to HTTP using the transpor tMode and httpPath session configuration variables.
   jdbc:hive2://myserver:10000/default;transportMode=http;httpPath=myendpoint.com;

   You need to specify httpPath when using the HTTP transport mode. <http_endpoint> has a corresponding HTTP endpoint configured in hive-site.xml.

3. Add parameters to the connection string for Kerberos authentication.
   jdbc:hive2://myserver:10000/default;principal=prin.dom.com@APRINCIPAL.DOM.COM

JDBC connection string syntax

The JDBC connection string for connecting to a remote Hive client requires a host, port, and Hive database name. You can optionally specify a transport type and authentication.

jdbc:hive2://<host>:<port>/<dbName>;<sessionConfs>?<hiveConfs>#<hiveVars>

Connection string parameters

The following table describes the parameters for specifying the JDBC connection.

JDBC Parameter	Description	Required
host	The cluster node hosting HiveServer.	yes
port	The port number to which HiveServer listens.	yes
dbName	The name of the Hive database to run the query against.	yes
sessionConfs	Optional configuration parameters for the JDBC/ODBC driver in the following format: <key1>=<value1>;<key2>=<key2>...;	no
hiveConfs	Optional configuration parameters for Hive on the server in the following format: <key1>=<value1>;<key2>=<key2>; ... The configurations last for the duration of the user session.	no
hiveVars	Optional configuration parameters for Hive variables in the following format: <key1>=<value1>;<key2>=<key2>; ... The configurations last for the duration of the user session.	no

TCP and HTTP Transport

The following table shows variables for use in the connection string when you configure HiveServer. The JDBC client and HiveServer can use either HTTP or TCP-based transport to exchange RPC messages. Because the default transport is TCP, there is no need to specify transportMode=binary if TCP transport is desired.

transportMode Variable Value	Description
http	Connect to HiveServer2 using HTTP transport.
binary	Connect to HiveServer2 using TCP transport.

The syntax for using these parameters is:

```
jdbc:hive2://<host>:<port>/<dbName>;transportMode=http;httpPath=<http_endpoint>;<otherSessionConfs>?<hiveConfs>#<hiveVars>
```

User Authentication

If configured in remote mode, HiveServer supports Kerberos, LDAP, Pluggable Authentication Modules (PAM), and custom plugins for authenticating the JDBC user connecting to HiveServer. The format of the JDBC connection URL for authentication with Kerberos differs from the format for other authentication models. The following table shows the variables for Kerberos authentication.

User Authentication Variable	Description
principal	A string that uniquely identifies a Kerberos user.
saslQop	Quality of protection for the SASL framework. The level of quality is negotiated between the client and server during authentication. Used by Kerberos authentication with TCP transport.
user	Username for non-Kerberos authentication model.
password	Password for non-Kerberos authentication model.

The syntax for using these parameters is:

```
jdbc:hive://<host>:<port>/<dbName>;principal=<HiveServer2_kerberos_principal
>;<otherSessionConfs>?<hiveConfs>#<hiveVars>
```

Transport Layer Security

HiveServer2 supports SSL and Sasl QOP for transport-layer security. The format of the JDBC connection string for SSL uses these variables:

SSL Variable	Description
ssl	Specifies whether to use SSL
sslTrustStore	The path to the SSL TrustStore.
trustStorePassword	The password to the SSL TrustStore.

The syntax for using the authentication parameters is:

```
jdbc:hive2://<host>:<port>/<dbName>;\
ssl=true;sslTrustStore=<ssl_truststore_path>;trustStorePassword=<truststor
e_password>;\
<otherSessionConfs>?<hiveConfs>#<hiveVars>
```

When using TCP for transport and Kerberos for security, HiveServer2 uses Sasl QOP for encryption rather than SSL.

Sasl QOP Variable	Description
principal	A string that uniquely identifies a Kerberos user.
saslQop	The level of protection desired. For authentication, checksum, and encryption, specify auth-conf. The other valid values do not provide encryption.

The JDBC connection string for Sasl QOP uses these variables.

```
jdbc:hive2://FQDN.EXAMPLE.COM:10000/default;principal=hive/_H
OST@EXAMPLE.COM;saslQop=auth-conf
```

The _HOST is a wildcard placeholder that gets automatically replaced with the fully qualified domain name (FQDN) of the server running the HiveServer daemon process.

Using JdbcStorageHandler to query an RDBMS

Using the JdbcStorageHandler, you can connect Hive to a MySQL, PostgreSQL, Oracle, DB2, or Derby data source. You can then create an external table to represent the data, and query the table.

About this task

This task assumes you are a CDP Private Cloud Base user. You create an external table that uses the JdbcStorageHandler to connect to and read a local JDBC data source.

Procedure

1. Load data into a supported SQL database, such as MySQL, on a node in your cluster, or familiarize yourself with existing data in the your database.
2. Create an external table using the JdbcStorageHandler and table properties that specify the minimum information: database type, driver, database connection string, user name and password for querying hive, table name, and number of active connections to Hive.

```
CREATE EXTERNAL TABLE mytable_jdbc(
  col1 string,
  col2 int,
  col3 double
)
STORED BY 'org.apache.hive.storage.jdbc.JdbcStorageHandler'
TBLPROPERTIES (
  "hive.sql.database.type" = "MYSQL",
  "hive.sql.jdbc.driver" = "com.mysql.jdbc.Driver",
  "hive.sql.jdbc.url" = "jdbc:mysql://localhost/sample",
  "hive.sql.dbcp.username" = "hive",
  "hive.sql.dbcp.password" = "hive",
  "hive.sql.table" = "MYTABLE",
  "hive.sql.dbcp.maxActive" = "1"
);
```

3. Query the external table.

```
SELECT * FROM mytable_jdbc WHERE col2 = 19;
```