Integrating Apache Hive with Spark and BI

Date of Publish: 2019-04-15

https://docs.hortonworks.com/

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 need to understand how to use HWC to access Spark tables from Hive. You also export tables to Hive from Spark and vice versa using this connector.

You can use the HWC API to access any type of table in Hive from Spark. When you use SparkSQL, standard Spark APIs access tables Spark tables.

Using the HWC, you can export tables and extracts from the Spark to Hive and from Hive to Spark. You export tables and extracts from Spark to Hive by reading them using Spark APIs and writing them to Hive using the HWC. Conversely, you export tables and extracts from Hive to Spark by reading them using the Hive Warehouse Connector and writing them to Spark using Spark APIs.

Using the HWC, you can read and write Apache Spark DataFrames and Streaming DataFrames to and from Apache Hive. Apache Ranger and the HiveWarehouseConnector library provide row and column, fine-grained access to Spark data in Hive.

Limitations

- Currently, HWC supports tables in ORC format only.
- The spark thrift server is not supported.

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 for ORC-formatted data
- Selecting Hive data and retrieving a DataFrame
- Writing a DataFrame to Hive in batch
- Executing a Hive update statement
- Reading table data from Hive, transforming it in Spark, and writing it to a new Hive table
- Writing a DataFrame or Spark stream to Hive using HiveStreaming

Related Information

HiveWarehouseConnector Github project (select a feature branch)

HiveWarehouseConnector for handling Apache Spark data

Hortonworks Community Connection: Integrating Apache Hive with Apache Spark-\-Hive Warehouse Connector

Hive Warehouse Connector Use Cases

Apache Spark-Apache Hive connection configuration

In Spark, you can use the Hive Warehouse Connector for accessing ACID table data in Hive.

Prerequisites

You need to use the following software to connect Spark and Hive using the HiveWarehouseConnector library.

- Spark2
- Hive LLAP

  The Hive Warehouse Connector (HWC) and low-latency analytical processing (LLAP) are required for certain tasks, as shown in the following table:

Table 1: Spark Compatibility

Tasks	HWC Required	LLAP Required	Other Requirement/ Comments
Read Hive managed tables from Spark	Yes	Yes	Ranger ACLs enforced.
Write Hive managed tables from Spark	Yes	No	Ranger ACLs enforced.
Read Hive external tables from Spark	No	Only if HWC is used	Table must be defined in Spark catalog. Ranger ACLs not enforced.
Write Hive external tables from Spark	No	No	Ranger ACLs enforced.
Read Spark tables	Yes	Yes	Ranger ACLs enforced.
Write Spark tables	Yes	No	Ranger ACLs enforced.

You need low-latency analytical processing (LLAP) to read ACID, or other Hive-managed tables, from Spark. You do not need LLAP to write to ACID, or other managed tables, from Spark. 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 table above.

Spark on a Kerberized YARN cluster

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 Ambari, copy the value for this property from **Services** > **Hive** > **Configs** > **Advanced** > **Advanced hive-site** hive.server2.authentication.kerberos.principal.

Related Information

HiveWarehouseConnector Github project (select a feature branch)

HiveWarehouseConnector for handling Apache Spark data

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 .../hive_warehouse_connector/.
2. Add the connector jar to the app submission using --jars.

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

Related Information

HiveWarehouseConnector Github project (select a feature branch)
HiveWarehouseConnector for handling Apache Spark data

Submit a Hive Warehouse Connector Python app

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

Procedure

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

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

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

```
spark-shell --jars <path>/hive_warehouse_connector/hive-warehouse-
connector-assembly-1.0.0.jar
```

5. Add the Python package for the connector to the app submission.

```
pyspark --jars <path to jars>/hive_warehouse_connector/hive-warehouse-
connector-assembly-<version>.jar --py-files /usr/hdp/current/
hive_warehouse_connector/pyspark_hwc-<version>.zip
```

Related Information

HiveWarehouseConnector Github project (select a feature branch)
HiveWarehouseConnector for handling Apache Spark data

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 that you must manage.

Spark-Hive supported types mapping

The following types are supported for access through 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

Spark Type	Hive Type
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 to be in UTC time. 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. This is 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

HiveWarehouseConnector Github project (select a feature branch)

HiveWarehouseConnector for handling Apache Spark data

HiveWarehouseSession API operations

HiveWarehouseSession acts as an API to bridge Spark with Hive. In your Spark source code, you create an instance of HiveWarehouseSession. You use the language-specific code to create the HiveWarehouseSession.

Import statements and variables

The following string constants are defined by the API:

- HIVE_WAREHOUSE_CONNECTOR
- DATAFRAME_TO_STREAM
- STREAM_TO_STREAM

For more information, see the Github project for the Hive Warehouse Connector.

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()
```

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(100)
- 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",
 "bigint").column("ws_ship_date_sk", "bigint").create()
```

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

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

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

Read operations

Execute a Hive SELECT query and return a DataFrame.

hive.executeQuery("select * from web_sales")

Write operations

- Execute a Hive update statement

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

  Note: You can execute CREATE, UPDATE, DELETE, INSERT, and MERGE statements in this way.
- Write a DataFrame to Hive in batch (uses LOAD DATA INTO TABLE)

  Java/Scala:

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

  Python:

  ```
  df.write.format(HiveWarehouseSession().HIVE_WAREHOUSE_CONNECTOR).option("table",
   &tableName>).save()
  ```
- Write a DataFrame to Hive using HiveStreaming

  Java/Scala:

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

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

  Python:

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

  //Or, to write to 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").start()
  ```

  Python:

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

Write a DataFrame from Spark to Hive example

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. You should specify a Spark SaveMode, such as Append.

```
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.executeQuery("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()
```

HiveWarehouseSession interface

```
package com.hortonworks.hwc;

public interface HiveWarehouseSession {

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

//Execute Hive update statement
   boolean executeUpdate(String sql);

//Execute Hive catalog-browsing operation and return DataFrame
   Dataset<Row> execute(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);
}
```

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();
}
```

Related Information

HiveWarehouseConnector Github project (select a feature branch)

HiveWarehouseConnector for handling Apache Spark data

Hortonworks Community Connection: Integrating Apache Hive with Apache Spark-\-Hive Warehouse Connector

Hive Warehouse Connector Use Cases

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' = '\\');

Query a SQL data source using the JdbcStorageHandler

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

About this task

In this task 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;
```

Related Information

Apache Wiki: JdbcStorageHandler