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 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. You read these tables using Spark APIs. You write tables 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
- Partitioning data when writing a DataFrame

Related Information

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

Required properties

You must add several Spark properties through spark-2-defaults in Cloudera Manager to use the Hive Warehouse Connector for accessing data in Hive. Alternatively, configuration can be provided for each job using hive -\-conf.

- spark.sql.hive.hiveserver2.jdbc.url

  The URL for HiveServer2 Interactive
- spark.datasource.hive.warehouse.metastoreUri

  The URI for the metastore.
- spark.datasource.hive.warehouse.load.staging.dir

  The HDFS temp directory for batch writes to Hive, /tmp for example
- spark.hadoop.hive.llap.daemon.service.hosts

  The application name for LLAP service
- spark.hadoop.hive.zookeeper.quorum

  ZooKeeper hosts used by LLAP

Set the values of these properties as follows:

- spark.datasource.hive.warehouse.metastoreUri: Copy the value from hive.metastore.uris. In Hive, at the hive> prompt, enter set hive.metastore.uris and copy the output. For example, thrift://mycluster-1.com:9083.
- spark.hadoop.hive.llap.daemon.service.hosts: Copy the value from TBD > hive.llap.daemon.service.hosts.
- spark.hadoop.hive.zookeeper.quorum: Copy the value from TBDhive.zookeeper.quorum

Other HWC properties

The spark.datasource.hive.warehouse.write.path.strictColumnNamesMapping validates the mapping of columns against those in Hive to alert the user to input errors. Default = true.

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 Cloudera Manager, TBD.
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: Must use Spark ServiceCredentialProvider and set equal to a boolean, such as true
- Comment: true by default

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
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

As a Spark developer, you execute queries to Hive using the JDBC-style 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

For more information, see the Github project for the Hive Warehouse Connector (link below).

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()
```

Related Information

HiveWarehouseConnector Github project (select a feature branch)

HiveWarehouseConnector for handling Apache Spark data

Integrating Apache Hive with Apache Spark-\-Hive Warehouse Connector

Hive Warehouse Connector Use Cases

Catalog operations

Catalog operations include creating, dropping, and describing a Hive database and table from Spark.

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>)

Related Information

HiveWarehouseConnector Github project (select a feature branch)

HiveWarehouseConnector for handling Apache Spark data

Read and write operations

The API supports reading Spark tables from Hive. HWC supports push-downs of DataFrame filters and projections applied to executeQuery. 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.executeQuery("select * from web_sales")

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>).save()
```

Python:

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

Python:

```
df.write.format(HiveWarehouseSession().HIVE_WAREHOUSE_CONNECTOR).option("table",
 &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("partition", <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").start()
```

Python:

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

Related Information

HiveWarehouseConnector Github project (select a feature branch)

HiveWarehouseConnector for handling Apache Spark data

SPARK-20236

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

HiveWarehouseConnector Github project (select a feature branch)

HiveWarehouseConnector for handling Apache Spark data

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.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()
```

Related Information

HiveWarehouseConnector Github project (select a feature branch)

HiveWarehouseConnector for handling Apache Spark data

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
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);

//Helper for merge query
MergeBuilder mergeBuilder();
}
```

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
 matchExpr
MergeBuilder whenMatchedThenUpdate(String matchExpr, String...
 nameValuePairs);

//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

HiveWarehouseConnector Github project (select a feature branch)

HiveWarehouseConnector for handling Apache Spark data

Connecting Hive to BI tools using a JDBC/ODBC driver

To query, analyze, and visualize data stored within the Hortonworks Data Platform using drivers provided by Hortonworks, 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. You can download the drivers from the downloads page on the Cloudera web site.

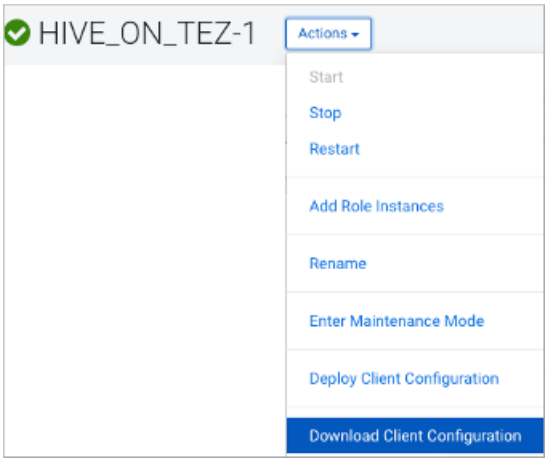
Before you begin

- Choose a Hive authorization model.
- Configure authenticated users for querying Hive through JDBC or ODBC driver by setting the value of the hive.server2.enable.doAs configuration property in the hive.site.xml file.

Procedure

1. Download the JDBC or ODBC driver.
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: put the driver in the classpath of your JDBC client.
3. Find the JDBC URL for HiveServer.

- In Cloudera Data Platform (CDP), from the hamburger menu of your computer cluster, click Copy JDBC URL.
- In Cloudera Manager (CM), click Clusters > HIVE_ON_TEZ_1 click Actions, and select Download Client Configuration.



Open beeline-site.xml, and copy the value of beeline.hs2.jdbc.url.HIVE_ON_TEZ-1 . 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.

Related Information

Locate the JDBC or ODBC driver

Specify the JDBC connection string

HiveWarehouseConnector Github project (select a feature branch)

Locate the JDBC or ODBC driver

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

About this task

Hortonworks provides the JDBC driver as part of the HDP distribution, and provides an ODBC driver as an add-on to the distribution for HDP support subscription customers.

Procedure

1. Get the driver.
   - In Ambari, navigate to **Services** > **Hive** > **Summary**and in Quick Links, click Jdbc Standalone Jar Download.
   - On a cluster node, navigate to /usr/hdp/current/hive-client/lib to locate the Hortonworks driver hive-jdbc.jar that HDP installed on your cluster.
   - Download the Hortonworks JDBC driver from the Hive JDBC driver archive.
   - Download the ODBC driver https://hortonworks.com/downloads/ > Addons. Skip the rest of the steps in this procedure and follow ODBC driver installation instructions.
2. Optionally, if you run a host outside of the Hadoop cluster, to use the JDBC driver in HTTP and HTTPS modes, give clients access to hive-jdbc-<version>-standalone.jar, hadoop-common.jar, and hadoop-auth.jar.

Specify the JDBC connection string

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

About this task

In embedded mode, HiveServer runs within the Hive client, not as a separate process. Consequently, the URL does not need a host or port number to make the JDBC connection. In remote mode, 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 transportMode and httpPath session configuration variables.
   jdbc:hive3://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:hive3://myserver:10000/default;principal=prin.dom.com@APRINCIPAL.DOM.COM

Related Information

HDP Addons: ODBC/JDBC Drivers

JDBC connection string syntax

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

jdbc:hive3://<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

JDBC Parameter	Description	Required
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 in remote mode. 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.

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:hive2://<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 differs from the format used by Sasl QOP.

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=<truststore_pas
<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.

```
jdbc:hive2://<host>:<port>/
<dbName>;principal=<HiveServer2_kerberos_principal>;saslQop=auth-
conf;<otherSessionConfs>?<hiveConfs>#<hiveVars>
```

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