

## Integrating Apache Hive with Spark and BI

Date published: 2019-08-21

Date modified: 2021-03-18



# Legal Notice

© Cloudera Inc. 2024. All rights reserved.

The documentation is and contains Cloudera proprietary information protected by copyright and other intellectual property rights. No license under copyright or any other intellectual property right is granted herein.

Unless otherwise noted, scripts and sample code are licensed under the Apache License, Version 2.0.

Copyright information for Cloudera software may be found within the documentation accompanying each component in a particular release.

Cloudera software includes software from various open source or other third party projects, and may be released under the Apache Software License 2.0 (“ASLv2”), the Affero General Public License version 3 (AGPLv3), or other license terms. Other software included may be released under the terms of alternative open source licenses. Please review the license and notice files accompanying the software for additional licensing information.

Please visit the Cloudera software product page for more information on Cloudera software. For more information on Cloudera support services, please visit either the Support or Sales page. Feel free to contact us directly to discuss your specific needs.

Cloudera reserves the right to change any products at any time, and without notice. Cloudera assumes no responsibility nor liability arising from the use of products, except as expressly agreed to in writing by Cloudera.

Cloudera, Cloudera Altus, HUE, Impala, Cloudera Impala, and other Cloudera marks are registered or unregistered trademarks in the United States and other countries. All other trademarks are the property of their respective owners.

Disclaimer: EXCEPT AS EXPRESSLY PROVIDED IN A WRITTEN AGREEMENT WITH CLOUDERA, CLOUDERA DOES NOT MAKE NOR GIVE ANY REPRESENTATION, WARRANTY, NOR COVENANT OF ANY KIND, WHETHER EXPRESS OR IMPLIED, IN CONNECTION WITH CLOUDERA TECHNOLOGY OR RELATED SUPPORT PROVIDED IN CONNECTION THEREWITH. CLOUDERA DOES NOT WARRANT THAT CLOUDERA PRODUCTS NOR SOFTWARE WILL OPERATE UNINTERRUPTED NOR THAT IT WILL BE FREE FROM DEFECTS NOR ERRORS, THAT IT WILL PROTECT YOUR DATA FROM LOSS, CORRUPTION NOR UNAVAILABILITY, NOR THAT IT WILL MEET ALL OF CUSTOMER’S BUSINESS REQUIREMENTS. WITHOUT LIMITING THE FOREGOING, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CLOUDERA EXPRESSLY DISCLAIMS ANY AND ALL IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, NON-INFRINGEMENT, TITLE, AND FITNESS FOR A PARTICULAR PURPOSE AND ANY REPRESENTATION, WARRANTY, OR COVENANT BASED ON COURSE OF DEALING OR USAGE IN TRADE.

# Contents

|  |               |
|--|---------------|
| <b>Introduction to HWC.....</b>                                      | <b>5</b>      |
| Getting started with HWC.....  | 5             |
| HWC limitations.....   | 9             |
| Reading data through HWC.....  | 9             |
| Example of configuring and reading a Hive managed table.....         | 10            |
| Direct Reader mode introduction.....                                 | 11            |
| Using Direct Reader mode.....  | 12            |
| Direct Reader configuration properties.....                          | 13            |
| Direct Reader limitations.....                                       | 14            |
| JDBC read mode introduction.....                                     | 14            |
| Using JDBC read mode.....  | 16            |
| JDBC mode configuration properties.....                              | 17            |
| JDBC mode limitations.....   | 17            |
| Kerberos configurations for HWC.....                                 | 17            |
| Writing data through HWC.....  | 18            |
| Introduction to HWC and DataFrame APIs.....                          | 19            |
| HWC and DataFrame API limitations.....                               | 21            |
| HWC supported types mapping.....                                     | 22            |
| Catalog operations.....  | 23            |
| Read and write operations.....                                       | 24            |
| Commit transaction for Direct Reader.....                            | 25            |
| Close HiveWarehouseSession operations.....                           | 26            |
| Use HWC for streaming.....   | 26            |
| HWC API Examples.....  | 26            |
| Hive Warehouse Connector Interfaces.....                             | 27            |
| Submit a Scala or Java application.....                              | 29            |
| HWC integration with pyspark and Zeppelin.....                       | 30            |
| Submit a Python app.....   | 30            |
| Livy interpreter configuration.....                                  | 31            |
| Read and write Hive tables in Zeppelin.....                          | 32            |
| <br><b>Apache Hive-Kafka integration.....</b>                        | <br><b>34</b> |
| Create a table for a Kafka stream.....                               | 34            |
| Querying Kafka data.....   | 35            |
| Query live data from Kafka.....                                      | 36            |
| Perform ETL by ingesting data from Kafka into Hive.....              | 37            |
| Writing data to Kafka.....   | 38            |
| Write transformed Hive data to Kafka.....                            | 38            |
| Set consumer and producer properties as table properties.....        | 39            |
| Kafka storage handler and table properties.....                      | 39            |
| <br><b>Connecting Hive to BI tools using a JDBC/ODBC driver.....</b> | <br><b>41</b> |
| Getting the JDBC or ODBC driver.....                                 | 41            |
| Integrating Hive and a BI tool.....                                  | 42            |
| Specify the JDBC connection string.....                              | 43            |
| JDBC connection string syntax.....                                   | 44            |

|   |           |
|---|-----------|
| <b>Using JdbcStorageHandler to query RDBMS.....</b> | <b>46</b> |
| <b>Set up JDBCStorageHandler for Postgres.....</b>  | <b>46</b> |

## Introduction to HWC

HWC securely accesses Hive managed tables from Spark. You need to use Hive Warehouse Connector (HWC) software to query Apache Hive managed tables from Apache Spark.

To read Hive external tables from Spark, you do not need HWC. Spark uses native Spark to read external tables. If you configure HWC to work with managed tables, you can use the same configuration to work with external tables.

### Supported applications and operations

The Hive Warehouse Connector supports the following applications:

- Spark shell
- PySpark
- The spark-submit script
- Zeppelin with the Livy interpreter

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](#)

[Orc vs Parquet](#)

## Getting started with HWC

You need to know how to use the Hive Warehouse Connector (HWC) with different programming languages and build systems. You find out where HWC binaries are located in CDP parcels and how a Spark application consumes the binaries.

### Cloudera artifactory and HWC dependency

To pull the HWC dependency corresponding to a release, use the following artifactory:

```
https://repository.cloudera.com/artifactory/cloudera-repos
```

### Use with Maven

To use HWC with maven, define the cloudera artifactory as a repository.

```
<repository>
  <id>cloudera</id>
  <name>cloudera</name>
  <url>https://repository.cloudera.com/artifactory/cloudera-repos</url>
```

```
</repository>
```

In the pom.xml of the project, add the dependency as shown in the following example:

```
<dependency>
  <groupId>com.hortonworks.hive</groupId>
  <artifactId>hive-warehouse-connector_2.11</artifactId>
  <version>[***HWC VERSION***]</version>
  <scope>provided</scope>
</dependency>
```

### Use with Sbt

Add the Cloudera and Shibboleth repositories, and the HWC dependency to the build sbt as follows:

```
resolvers += "Cloudera repo" at "https://repository.cloudera.com/artifactory/cloudera-repos",

resolvers += "opensaml Repository" at "https://build.shibboleth.net/nexus/content/repositories/releases",

libraryDependencies += "com.hortonworks.hive" % "hive-warehouse-connector_2.11" % "[***HWC VERSION***]" % "provided",
```

### Dependency scope

Generally, you add HWC dependencies in provided scope unless there is a specific requirement to do otherwise. While running spark application, you can specify the HWC jar present in your distribution using the --jars option to spark-submit or spark-shell.

### HWC Binaries in CDP

HWC binaries are located in /opt/cloudera/parcels/CDH/lib/hive\_warehouse\_connector/. This directory contains HWC jar, a python zip, and the R package. Use these binaries to launch Spark applications in Scala, Java, Python, or R.

The following files are in /opt/cloudera/parcels/CDH/lib/hive\_warehouse\_connector/.

- hive-warehouse-connector-assembly-<version>.jar
- pyspark\_hwc-<version>.zip

### Working with different languages

You use HWC APIs to perform basic read and write operations. You need to understand how to use HWC APIs with different languages. The following examples show basic capabilities that are covered in detail later in this documentation.

#### Use with Scala

```
import com.hortonworks.hwc.HiveWarehouseSession
import org.apache.spark.sql.{SaveMode, SparkSession}

object HWCApp {

  def main(args: Array[String]): Unit = {
    val spark = SparkSession.builder.appName("HWCApp").enableHiveSupport.getOrCreate
    val hwc = HiveWarehouseSession.session(spark).build
    // create sample data
    val tvSeries = createSampleDataDf(spark)
```

```

val tableName = "tv_series"

hwc.dropTable(tableName, true, true)
println(s"=====Writing to hive table - $tableName via HWC=====")
// write to hive table via HWC
tvSeries.write.format(HiveWarehouseSession.HIVE_WAREHOUSE_CONNECTOR)
  .option("table", tableName)
  .mode(SaveMode.Append).save

println(s"=====Reading hive table $tableName via HWC=====")
// Read via HWC
hwc.sql(s"select * from $tableName").show(truncate = false)

hwc.close()
spark.stop
}

private def createSampleDataDf(spark: SparkSession) = {
  spark.sql("drop table if exists tv_series_dataset")
  spark.sql("create table tv_series_dataset(id int, name string, genres
string, rating double) using orc")
  spark.sql("insert into tv_series_dataset values " +
    "(1, 'Chernobyl', 'Drama|History|Tragedy|Science', 9.4), " +
    "(2, 'Westworld', 'Sci-fi', 8.6), (3, 'Sense8', 'Sci-fi', 8.3), " +
    "(4, 'Person of Interest', 'Drama|Sci-fi', 8.4), " +
    "(5, 'Its okay to not be okay', 'Drama', 8.7), " +
    "(6, 'Daredevil', 'Action|Sci-fi', 8.6), " +
    "(7, 'Money Heist', 'Drama|Thriller', 8.3), " +
    "(8, 'Breaking Bad', 'Crime|Drama', 9.5)")
  spark.sql("select * from tv_series_dataset")
}

```

#### Use with Java

The following Java code is equivalent to the scala code above.

```

import com.hortonworks.hwc.HiveWarehouseSession
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SaveMode;
import org.apache.spark.sql.SparkSession;

public class HWCApp {
  public static void main(String[] args) {
    SparkSession spark = SparkSession.builder().appName("HWCApp").enableHive
Support().getOrCreate();
    // HiveWarehouseSession creation
    HiveWarehouseSession hwc = HiveWarehouseBuilder.session(spark).build();
    // create sample data
    Dataset<Row> tvSeries = createSampleDataDf(spark);
    String tableName = "tv_series";
    hwc.dropTable(tableName, true, true);
    System.out.println("=====Writing to hive table - " + tableName + " via
HWC=====");
    // write data to hive table via HWC
    tvSeries.write().format(HiveWarehouseSession.HIVE_WAREHOUSE_CONNECTOR)
      .option("table", tableName)
      .mode(SaveMode.Append).save();

    System.out.println("=====Reading hive table - " + tableName + " via
HWC=====");
    // read hive table as dataframe using HWC
    hwc.sql("select * from " + tableName).show(false);
  }
}

```

```

    hwc.close();
    spark.stop();
}
private static Dataset<Row> createSampleDataDf(SparkSession spark) {
    spark.sql("drop table if exists tv_series_dataset");
    spark.sql("create table tv_series_dataset(id int, name string, genres string, rating double) using orc");
    spark.sql("insert into tv_series_dataset values " +
        "(1, 'Chernobyl', 'Drama|History|Tragedy|Science', 9.4), " +
        "(2, 'Westworld', 'Sci-fi', 8.6), (3, 'Sense8', 'Sci-fi', 8.3), " +
        "(4, 'Person of Interest', 'Drama|Sci-fi', 8.4), " +
        "(5, 'It's okay to not be okay', 'Drama', 8.7), " +
        "(6, 'Daredevil', 'Action|Sci-fi', 8.6), " +
        "(7, 'Money Heist', 'Drama|Thriller', 8.3), " +
        "(8, 'Breaking Bad', 'Crime|Drama', 9.5)");
    return spark.sql("select * from tv_series_dataset");
}
}

```

### Launching a Java or Scala app

After packaging the app in a jar, launch the app using standard Spark syntax for launching applications. Provide HWC jar from the distribution. The Spark application can be launched as follows:

```

spark-submit --jars /opt/cloudera/parcels/CDH/lib/hive_warehouse_connector/hive-warehouse-connector-assembly-<version>.jar \
--class com.cloudera.HWCApp \
...More spark/HWC confs...
...More spark/HWC confs...
/path-to-jar/hwc-app.jar

```

### Use with Python

```

from pyspark.sql import SparkSession
from pyspark_llap import HiveWarehouseSession

spark = SparkSession.builder.enableHiveSupport().appName("hwc-app").getOrCreate()
hwc = HiveWarehouseSession.session(spark).build()
tableName = "tv_series"
hwc.dropTable(tableName, True, True)

tvSeries = spark.createDataFrame([
    (1, "Chernobyl", "Drama|History|Tragedy|Science", 9.4),
    (2, "Westworld", "Sci-fi", 8.6),
    (3, "Sense8", "Sci-fi", 8.3),
    (4, "Person of Interest", "Drama|Sci-fi", 8.4),
    (5, "It's okay to not be okay", "Drama", 8.7),
    (6, "Daredevil", "Action|Sci-fi", 8.6),
    (7, "Money Heist", "Drama|Thriller", 8.3),
    (8, "Breaking Bad", "Crime|Drama", 9.5)
], ["id", "name", "genres", "rating"])

print("=====Writing to hive table - " + tableName + " via HWC=====")
# write to hive table via HWC
tvSeries.write.format(HiveWarehouseSession.HIVE_WAREHOUSE_CONNECTOR).option("table", tableName).mode("append").save()
print("=====Reading hive table - " + tableName + " via HWC=====")
# Read via HWC
hwc.sql("select * from " + tableName).show()

hwc.close()

```



```
spark.stop()
```

### Launching a Python app

After getting the python code ready, launch it using spark-submit. Provide the HWC jar and HWC python zip as follows:

```
spark-submit --jars /opt/cloudera/parcels/CDH/lib/hive_warehouse_connector/hive-warehouse-connector-assembly-<version>.jar \
--py-files /opt/cloudera/parcels/CDH/lib/hive_warehouse_connector/pyspark_hwc-<version>.zip \
...More spark/HWC confs...
...More spark/HWC confs...
/path-to-python-app/hwc-app.py
```

## HWC limitations

You need to be aware of HWC limitations, including Kerberos properties that are not allowed, and unsupported operations and connections. These limitations are in addition to Direct Reader mode, JDBC mode, and HWC and DataFrames API limitations.

General HWC limitations are:

- HWC supports reading tables in any format, but currently supports writing tables in ORC format only.
- The spark and livy thrift servers are not supported.
- The Hive Union types are not supported.

Workaround for using the Hive Warehouse Connector with Oozie Spark action

Hive and Spark use different Thrift versions and are incompatible with each other. Upgrading Thrift in Hive is complicated and may not be resolved in the near future. Therefore, Thrift packages are shaded inside the HWC JAR to make Hive Warehouse Connector work with Spark and Oozie Spark action. See the workaround in Cloudera Oozie documentation.

Secured cluster configurations

Set the following configurations in a secured cluster:

- ```
--conf "spark.security.credentials.hiveserver2.enabled=true"
```
- ```
--conf "spark.sql.hive.hiveserver2.jdbc.url.principal=hive/_HOST@ROOT.HWX.SITE"
```

The jdbc url must not contain the jdbc url principal and must be passed as shown here.

### Related Information

[Using Hive Warehouse Connector with Oozie Spark Action](#)

[Union Types](#)

## Reading data through HWC

You can configure one of the several HWC modes to read Apache Hive managed tables from Apache Spark. You need to know about the modes you can configure for querying Hive from Spark. Examples of how to configure the modes are presented.

### About this task

In this release, HWC configuration has been simplified.

You set the following configurations when starting the spark shell:

- `spark.sql.extensions="com.hortonworks.spark.sql.rule.Extensions"`
- `spark.datasource.hive.warehouse.read.mode=<mode>`

where <mode> is one of the following:

- `DIRECT_READER_V1`
- `JDBC_CLUSTER`
- `JDBC_CLIENT`

You can transparently read with HWC in different modes using just `spark.sql("<query>")`. You can specify the mode in the spark-shell when you run Spark SQL commands to query Apache Hive tables from Apache Spark. You can also specify the mode in `configuration/spark-defaults.conf`, or using the `--conf` option in `spark-submit`.

For backward compatibility, configuring `spark.datasource.hive.warehouse.read.mode` is the same as the following configurations.

- `--conf spark.datasource.hive.warehouse.read.jdbc.mode //deprecated`
- `--conf spark.sql.hive.hwc.execution.mode //deprecated`
- `--conf spark.datasource.hive.warehouse.read.via.llap //deprecated`

The old configurations are still supported for backward compatibility, but in a later release, support will end for these configurations and `spark.datasource.hive.warehouse.read.mode` will replace these configurations. HWC gives precedence to new configurations when old and new ones are encountered.

### Related Information

[Using Direct Reader mode](#)

[Using JDBC read mode](#)

## Example of configuring and reading a Hive managed table

### Before you begin

Set Kerberos for HWC.

### Procedure

1. Choose a read mode.
2. Start the Spark session using the following configurations.

For example, start the Spark session using Direct Reader and configure for kyro serialization:

```
spark-shell --jars ./hive-warehouse-connector-assembly-<version>.jar \
--master yarn \
--conf spark.sql.extensions="com.hortonworks.spark.sql.rule.Extensions" \
--conf spark.kryo.registrator=com.qubole.spark.hiveacid.util.HiveAcidKyroRegistrator \
--conf spark.sql.hive.hiveserver2.jdbc.url="jdbc:hive2://hwc-2.hwc.root.hwx.site:2181/default;retries=5;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2" \
--conf spark.sql.hive.hiveserver2.jdbc.url.principal=hive/_HOST@ROOT.HWX.SITE \
--conf spark.datasource.hive.warehouse.read.mode=DIRECT_READER_V2
```

For example, start the Spark session using the JDBC\_CLUSTER option:

```
spark-shell --jars ./hive-warehouse-connector-assembly-<version>.jar
--master yarn
--conf spark.sql.extensions="com.hortonworks.spark.sql.rule.Extensions"
--conf spark.kryo.registrator=com.qubole.spark.hiveacid.util.HiveAcidKyroRegistrator
```

```
--conf spark.sql.hive.hiveserver2.jdbc.url="jdbc:hive2://hwc-2.hwc.root.hwx.site:2181/default;retries=5;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2"
--conf spark.sql.hive.hiveserver2.jdbc.url.principal=hive/_HOST@ROOT.HWX.SITE
--conf spark.datasource.hive.warehouse.read.mode=JDBC_CLUSTER
```

You must start the Spark session after setting the Direct Read option, so include the configurations in the launch string.

### 3. Read Apache Hive managed tables.

For example:

```
scala> val hive = com.hortonworks.hwc.HiveWarehouseSession.session(spark)
      .build()

scala> hive.sql("select * from managedTable").show
```

## Direct Reader mode introduction

Direct Reader mode is a transparent connection that Hive Warehouse Connector (HWC) makes to Apache Hive metastore (HMS) to get transaction information. In Direct Reader mode, Spark reads the data directly from the managed table location using the transaction snapshot. You use this mode if you do not need production-level Ranger authorization. Direct Reader mode does not support Ranger authorization.

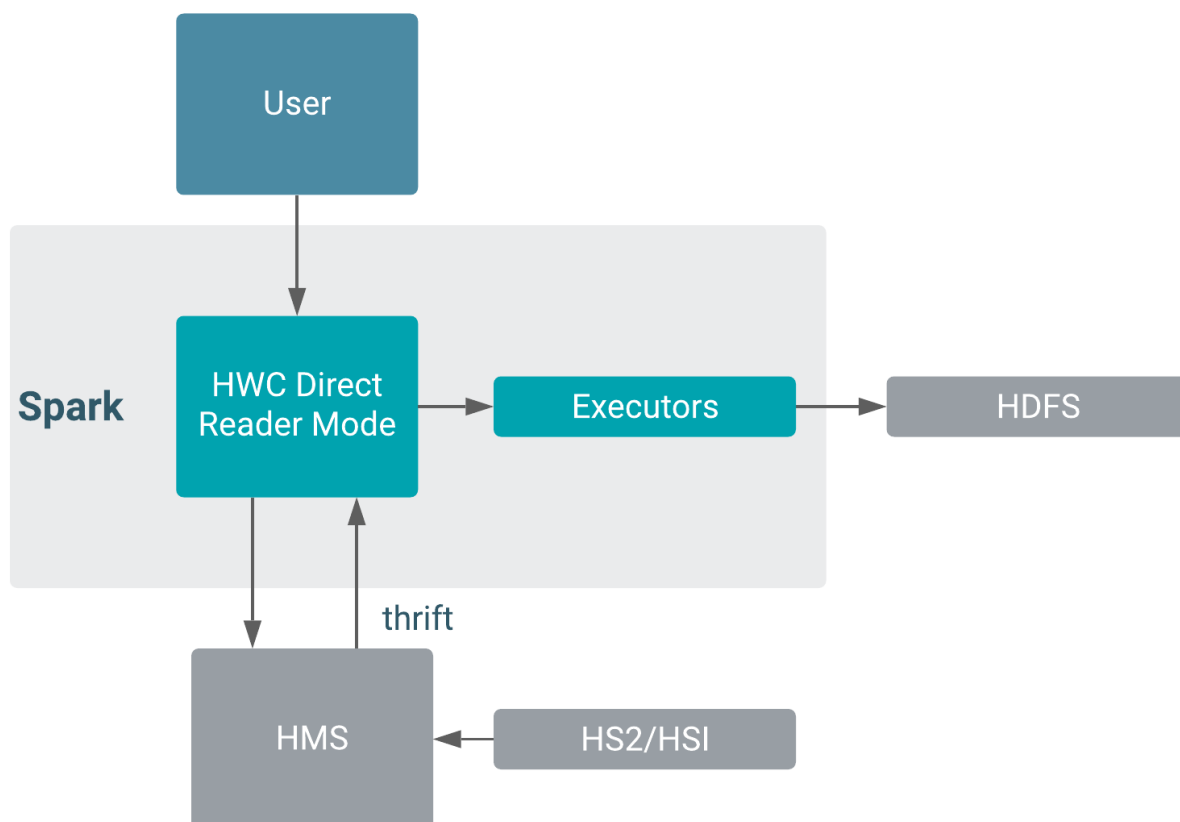
### Requirements and recommendations

Spark Direct Reader mode requires a connection to Hive metastore. A HiveServer (HS2) connection is not needed.

Spark Direct Reader for reading Hive ACID, transactional tables from Spark is supported for production use. Use Spark Direct Reader mode if your ETL jobs do not require authorization and run as super user.

### Component interaction

The following diagram shows component interaction ifor Direct Reader reads.



### Related Information

[Using Direct Reader mode](#)

[Using JDBC read mode](#)

## Using Direct Reader mode

In a few steps, you configure Apache Spark to connect to the Apache Hive metastore. An example shows how to configure Direct Reader reads while launching the Spark shell.

### About this task

This procedure assumes you require serialization and sets `spark.kryo.registrator=com.qubole.spark.hiveacid.util.HiveAcidKryoRegistrator`.

### Before you begin

For secure clusters, additional configurations will be needed by spark.

.

### Procedure

1. In Cloudera Manager, in Hosts > Roles, if Hive Metastore appears in the list of roles, copy the host name or IP address.

You use the host name or IP address in the next step to set the host value.

2. Launch the Spark shell and include the Direct Reader configurations.

For example:

```
spark-shell --jars ./hive-warehouse-connector-assembly-<version>.jar \
--master yarn \
--conf spark.sql.extensions="com.hortonworks.spark.sql.rule.Extensions" \
--conf spark.kryo.registrator=com.qubole.spark.hiveacid.util.HiveAcidKyroRegistrator \
--conf spark.sql.hive.hiveserver2.jdbc.url="jdbc:hive2://<domain name>:<port>/default;principal=hive/_HOST@ROOT.HWX.SITE;retries=5;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2"

--conf spark.datasource.hive.warehouse.read.mode=DIRECT_READER_V1
```

3. Read data in table customer.

View data in table customer.

```
scala>
spark.sql("select c_customer_sk, c_customer_id, c_last_name, c_birth_country from customer where c_birth_year=1983 limit 2 ").show()
21/02/08 11:03:31 INFO rule.HWCSwitchRule: using DIRECT_READER_V1 extension for reading
+-----+-----+-----+-----+
|c_customer_sk|c_customer_id|c_last_name|c_birth_country|
+-----+-----+-----+-----+
|          55634|AAAAAAACFJNAAAA|Campbell|THAILAND|
|          74213|AAAAAAAF0BCBAAA|Hudgins|KYRGYZSTAN|
+-----+-----+-----+-----+
```

## Direct Reader configuration properties

You need to know the property names and valid values for configuring Direct Reader mode.

### Options

In configuration/spark-defaults.conf, or using the --conf option in spark-submit/spark-shell set the following properties:

**Name: spark.sql.extensions**

Value: com.hortonworks.spark.sql.rule.Extensions

Required for using Spark SQL in auto-translate direct reader mode. Set before creating the spark session.

**spark.datasource.hive.warehouse.read.mode**

DIRECT\_READER\_V1

Required for performing Direct Reader reads.

**Name: spark.kryo.registrator**

Value: com.qubole.spark.hiveacid.util.HiveAcidKyroRegistrator

Set before the spark session. Required if serialization = kryo.

**Name: spark.hadoop.hive.metastore.uris**

Value: thrift://<host>:<port>

Hive metastore URI.

**Name: --jars**

Value: HWC jar

Pass the HWC jar to spark-shell or spark-submit using the `--jars` option while launching the application. For example, launch spark-shell as follows.

Example: Launch a spark-shell

```
spark-shell --jars \
/opt/cloudera/parcels/CDH/lib/hive_warehouse_connector/hive-warehouse-c
onconnector-assembly-<version>.jar \
--conf "spark.sql.extensions=com.hortonworks.spark.sql.rule.Extensions"
\
--conf "spark.datasource.hive.warehouse.read.mode=DIRECT_READER_V1
--conf "spark.kryo.registrator=com.qubole.spark.hiveacid.util.HiveAci
dKyroRegistrator" \
--conf "spark.hadoop.hive.metastore.uris=<metastore_uri>"
```

## Direct Reader limitations

You must understand the limitations of Direct Reader mode and what functionality is not supported.

### Limitations

- You cannot write data using HWC Direct Reader.
- Transaction semantics of Spark RDDs are not ensured when using Spark Direct Reader to read ACID tables.
- Supports only single-table transaction consistency. The direct reader does not guarantee that multiple tables referenced in a query read the same snapshot of data.
- Does not auto-commit transactions submitted by rdd APIs. Explicitly close transactions to release locks.
- Requires read and execute access on the hive-managed table locations.
- Does not support Ranger authorization.

You must configure read access to the HDFS, or other, location for managed tables. You must have Read and Execute permissions on hive warehouse location (`hive.metastore.warehouse.dir`).

- Blocks compaction on open read transactions.

The way Spark handles null and empty strings can cause a discrepancy between metadata and actual data when writing the data read by Spark Direct Reader to a CSV file.

### Unsupported functionality

Spark Direct Reader does not support the following functionality:

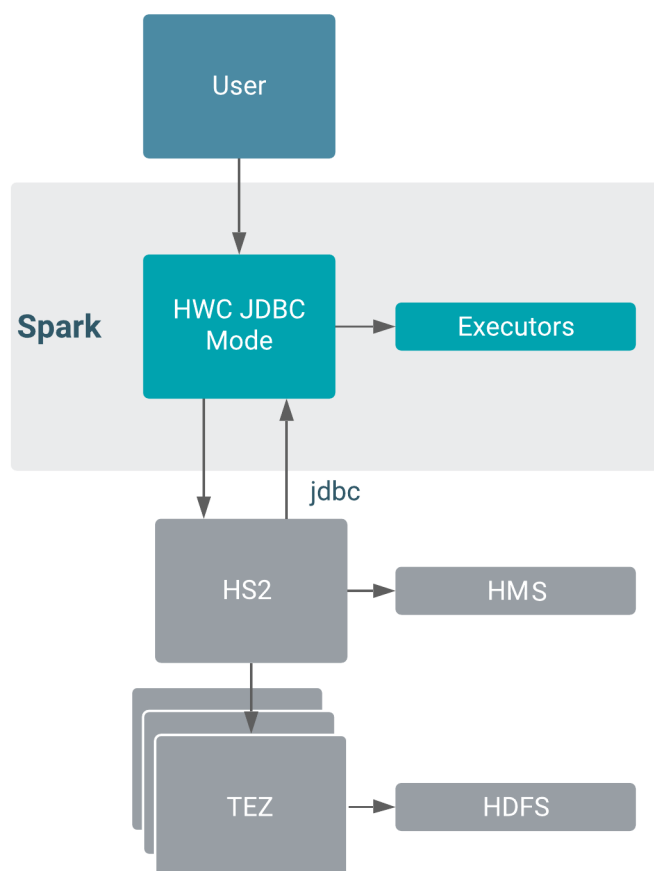
- Writes
- Streaming inserts
- CTAS statements

## JDBC read mode introduction

JDBC read mode is a connection that Hive Warehouse Connector (HWC) makes to HiveServer (HS2) to get transaction information. JDBC read mode is secured through Ranger authorization and supports fine-grained access control, such as column masking. You need to understand how you read Apache Hive tables from Apache Spark through HWC using the JDBC mode. The location where your queries are executed affects configuration. Understanding execution locations and recommendations help you configure JDBC reads for your use case.

### Component Interaction

Only one JDBC connection to HiveServer (HS2) is a potential bottleneck in data transfer to Spark. The following diagram shows interaction when you configure HWC in JDBC mode with Hive metastore (HMS), TEZ, and HDFS.



HWC does not use JDBC to write. HWC writes to an intermediate location from Spark, and then executes a LOAD DATA query to write the data. Using HWC to write data is recommended for production.

### Configuration

A JDBC read takes place in these locations:

- Driver: Using the Hive JDBC url, connects to Hive and executes the query on the driver side.
- Cluster: From Spark executors, connects to Hive through JDBC and executes the query.

Authorization occurs on the server.

A JDBC read occurs in the client or cluster:

- Client (Driver)

In the client, any failures to connect to HiveServer (HS2) will not be retried.

- Cluster (Executor)--recommended

In the cluster, any failures to connect to HS2 will be retried automatically.

JDBC reads are recommended for production for workloads having a data size of 1GB or less. Larger workloads are not recommended for JDBC reads in production due to slow performance.

Where your queries are executed affects the Kerberos configurations for HWC.

### Optimize reads using HWC session APIs

Using the HWC session API, you can use `hive.execute` to execute a fast read. This command processes queries through HWC to perform JDBC or Direct Reader reads.

## Using JDBC read mode

In a few steps, you configure Apache Spark to connect to HiveServer (HS2). Examples show how to configure JDBC Cluster and JDBC Client modes while launching the Spark shell.

### Before you begin

- Accept the default `spark.datasource.hive.warehouse.load.staging.dir` for the temporary staging location required by HWC.
- In `spark-defaults.conf`, check that `spark.hadoop.hive.zookeeper.quorum` is configured.
- In `spark-defaults.conf`, set Kerberos configurations for HWC, or for an unsecured cluster, set `spark.security.credentials.hiveserver2.enabled=false`.

### Procedure

1. Find the HiveServer (HS2) JDBC URL in `/etc/hive/conf.cloudera.HIVE_ON_TEZ-1/beeline-site.xml`. The value of `beeline.hs2.jdbc.url.HIVE_ON_TEZ-1` is the HS2 JDBC URL in this sample file.

```
...
<configuration>
  <property>
    <name>beeline.hs2.jdbc.url.default</name>
    <value>HIVE_ON_TEZ-1</value>
  </property>
  <property>
    <name>beeline.hs2.jdbc.url.HIVE_ON_TEZ-1</name>
    <value>jdbc:hive2://<domain name>:2181/?serviceDiscoveryMode=zooKeeper; \
      zooKeeperNamespace=hiveserver2;retries=5</value>
  </property>
</configuration>
```

2. Launch the Spark shell, including the configuration of the JDBC cluster option, and setting the Spark property to the value of the HS2 JDBC URL.

For example:

```
spark-shell --jars ./hive-warehouse-connector-assembly-<version>.jar \
--master yarn \
--conf spark.sql.extensions="com.hortonworks.spark.sql.rule.Extensions" \
--conf spark.kryo.registrator=com.qubole.spark.hiveacid.util.HiveAcidKyroRegistrator \
--conf spark.sql.hive.hiveserver2.jdbc.url="jdbc:hive2://hwc-2.hwc.root.hwx.site:2181/default;principal=hive/_HOST@ROOT.HWX.SITE;retries=5;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2"

--conf spark.datasource.hive.warehouse.read.mode=JDBC_CLUSTER
```

3. Read a hive table.

```
scala> sql("select * from managedTable").show
```

## Example of JDBC Client Mode

### Procedure

Launch the Spark shell in JDBC client mode.

```
spark-shell --jars ./hive-warehouse-connector-assembly-<version>.jar \
--master yarn \
--conf spark.sql.extensions="com.hortonworks.spark.sql.rule.Extensions" \
--conf spark.kryo.registrator=com.qubole.spark.hiveacid.util.HiveAcidKyroRegistrator \
```



```
--conf spark.sql.hive.hiveserver2.jdbc.url="jdbc:hive2://hwc-2.hwc.root.hwx.
site:2181/default;principal=hive/_HOST@ROOT.HWX.SITE;retries=5;serviceDiscov
eryMode=zooKeeper;zooKeeperNamespace=hiveserver2"

--conf spark.datasource.hive.warehouse.read.mode=JDBC_CLIENT
```

## JDBC mode configuration properties

You need to know the property names and valid values for configuring JDBC mode.

In configuration/spark-defaults.conf, or using the --conf option in spark-submit/spark-shell set the following properties:

**Name: spark.sql.extensions**

Value: com.hortonworks.spark.sql.rule.Extensions

**Name: spark.datasource.hive.warehouse.read.mode**

Value: JDBC\_CLUSTER or JDBC\_CLIENT

Configures the driver location.

**Name: spark.sql.hive.hiveserver2.jdbc.url**

Value:

The JDBC endpoint for HiveServer. For more information, see the Apache Hive Wiki (link below). For Knox, provide the HiveServer, not Knox, endpoint.

**Name: spark.datasource.hive.warehouse.load.staging.dir**

Value: Temporary staging location required by HWC. Set the value to a file system location where the HWC user has write permission.

**Name: spark.hadoop.hive.zookeeper.quorum**

## JDBC mode limitations

You must understand the limitations of JDBC mode and what functionality is not supported.

Keep the following limitations of JDBC mode in mind:

- JDBC\_CLUSTER and JDBC\_CLIENT are used for reads only, and are recommended for production workloads of 1 GB or less. With larger workload bottlenecks develop in data transfer to Spark.

Writes through HWC of any size are recommended for production. Writes do not use JDBC mode.

- In JDBC\_CLUSTER mode, HWC fails to correctly resolve queries that use the ORDER BY clause when run as hive.sql("<query> "). The query returns unordered rows of data even though the query contains an ORDER BY clause.

## Kerberos configurations for HWC

You learn how to configure and which parameters to set for a Kerberos-secure HWC connection for querying the Hive metastore from Spark.

The Hive Warehouse Connector (HWC) must connect to HiveServer (HS2) to execute writes or to execute reads in read modes other than Direct Reader. You need to set the following configuration properties to connect HWC to a Kerberos-enabled HiveServer:

- Property: spark.sql.hive.hiveserver2.jdbc.url.principal

Value: Set this value to the value of "hive.server2.authentication.kerberos.principal".

- Property: spark.security.credentials.hiveserver2.enabled

Value: Set this value to "true".

You do not need to explicitly provide other authentication configurations, such as auth type and principal. When Spark opens a secure connection to Hive metastore, Spark automatically picks the authentication configurations from the `hive-site.xml` that is present on the Spark app classpath. For example, to execute queries in direct reader mode through HWC, Spark opens a secure connection to Hive metastore and this authentication occurs automatically.

You can set the properties using the `spark-submit/spark-shell --conf` option.

## Writing data through HWC

A step-by-step procedure walks you through connecting to HiveServer (HS2) to perform batch writes from Spark, which is recommended for production. You configure HWC for the managed table write, launch the Spark session, and write ACID, managed tables to Apache Hive.

### Before you begin

- Accept the default `spark.datasource.hive.warehouse.load.staging.dir` for the temporary staging location required by HWC.
- Check that `spark.hadoop.hive.zookeeper.quorum` is configured.
- Set Kerberos configurations for HWC, or for an unsecured cluster, set `spark.security.credentials.hiveserver2.enabled=false`.

### About this task

Limitation: Only the ORC format is supported for writes.

The way data is written from HWC is not impacted by the read modes configured for HWC. For write operations, HWC writes to an intermediate location (as defined by the value of config `spark.datasource.hive.warehouse.load.staging.dir`) from Spark, followed by executing a "LOAD DATA" query in hive via JDBC. Exception: writing to dynamic partitions creates an intermediate temporary external table.

Using HWC to write data is recommended for production in CDP.

### Procedure

1. Open a terminal window, start the Apache Spark session, and include the URL for HiveServer.

```
spark-shell --jars /opt/cloudera/parcels/CDH/jars/hive-warehouse-connect
or-assembly-<version>.jar \
-- conf spark.sql.hive.hiveserver2.jdbc.url=<JDBC endpoint for HiveServer>
...
```

2. Include in the launch string a configuration of the intermediate location to use as a staging directory.  
Example syntax:

```
...
--conf spark.datasource.hive.warehouse.load.staging.dir=<path to directo
ry>
```

3. Write a Hive managed table.  
For example, in Scala:

```
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();
```

HWC internally fires the following query to Hive through JDBC:

```
LOAD DATA INPATH '<spark.datasource.hive.warehouse.load.staging.dir>' INTO
TABLE tpcds_bin_partitioned_orc_1000.newTable
```

4. Write to a statically partitioned, Hive managed table named t1 having two partitioned columns c1 and c2.

```
df.write.format(HIVE_WAREHOUSE_CONNECTOR).mode("append").option("partiti
on", "c1='val1',c2='val2'").option("table", "t1").save();
```

HWC internally fires the following query to Hive through JDBC after writing data to a temporary location.

```
LOAD DATA INPATH '<spark.datasource.hive.warehouse.load.staging.dir>' [O
VERWRITE] INTO TABLE db.t1 PARTITION (c1='val1',c2='val2');
```

5. Write to a dynamically partitioned table named t1 having two partitioned cols c1 and c2.

```
df.write.format(HIVE_WAREHOUSE_CONNECTOR).mode("append").option("partiti
on", "c1='val1',c2").option("table", "t1").save();
```

HWC internally fires the following query to Hive through JDBC after writing data to a temporary location.

```
CREATE TEMPORARY EXTERNAL TABLE db.job_id_table(cols...) STORED AS ORC
LOCATION '<spark.datasource.hive.warehouse.load.staging.dir>';

INSERT INTO TABLE t1 PARTITION (c1='val1',c2) SELECT <cols> FROM db.job
_id_table;
```

where <cols> should have comma separated list of columns in the table with dynamic partition columns being the last in the list and in the same order as the partition definition.

### Related Information

[Using Direct Reader mode](#)

[Using JDBC read mode](#)

## Introduction to HWC and DataFrame APIs

As an Apache Spark developer, you learn the code constructs for executing Apache Hive queries using the HiveWarehouseSession API. In Spark source code, you see how to create an instance of HiveWarehouseSession. You also learn how to access a Hive ACID table using DataFrames.

### Supported APIs

- Spark SQL

Supports native Spark SQL query read (only) patterns. Output conforms to native spark.sql conventions.

- HWC

Supports HiveWarehouse Session API operations using the HWC sql API.

- DataFrames

Supports accessing a Hive ACID table from Scala, or pySpark, directly using DataFrames. Use the short name HiveAcid. Direct reads and writes from the file are not supported.

#### Spark SQL Example

```
$ spark-shell <parameters to specify HWC jar and config settings>
scala> sql("select * from managedTable").show
scala> spark.read.table("managedTable").show
```

#### HWC API Example

```
scala> val hive = com.hortonworks.hwc.HiveWarehouseSession.session(spark
).build()
scala> hive.executeQuery("select * from emp_acid").show
scala> hive.executeQuery("select e.emp_id, e.first_name, d.name department
from emp_acid e join dept_ext d on e.dept_id = d.id").show
```

#### DataFrames Example

Hive ACID tables are tables in Hive metastore and must be formatted using DataFrames as follows:

Syntax:

```
spark.read.format("HiveAcid").option("table", "<table name>")
```

OR

```
val df = hive.sql("<SELECT query>")
```



**Note:** The `spark.read.format("HiveAcid")` method is deprecated and it is recommended that you use the `hive.sql()` method.

Example:

```
scala> val df = spark.read.format("HiveAcid").option(Map("table" -> "default.acidtbl")).load()
scala> df.collect()
```

OR

```
scala> val df = hive.sql("select * from managedTable where a=100")
scala> df.collect()
```

### 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 the cluster type you configured, either `JDBC_CLIENT` or `JDBC_CLUSTER`.
- `.execute()`
  - Required for executing queries if `spark.datasource.hive.warehouse.read.mode=JDBC_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.

Results are returned as a `DataFrame` to Spark.

## hive.sql vs. spark.sql

There are a number of important differences between the `hive.sql` and `spark.sql` functions:

- `hive.sql()` can only handle Apache Hive tables.
- `spark.sql()` selects HWC when you query an Apache Hive managed (ACID) table and falls back to native Spark for reading external tables.
- You can use the same Spark APIs, such as `spark.sql()`, to access either managed or external tables or temporary views. This is not the case with `hive.sql()`.
- The Direct Reader imposes the constraint that the Hive table must be transactional.

## Related Information

[HMS storage](#)

[Orc vs Parquet](#)

## HWC and DataFrame API limitations

These limitations are in addition to Direct Reader mode, JDBC mode, and HWC and DataFrames API limitations.

- Table stats (basic stats and column stats) are not generated when you write a `DataFrame` to Hive.

- 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.HiveWarehouseConnector"). \
mode("overwrite").option("table", "t1").save
```

## HWC supported types mapping

To create HWC API apps, you must know how Hive Warehouse Connector maps Apache Hive types to Apache Spark types, and vice versa. Awareness of a few unsupported types helps you avoid problems.

### 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](#)

## Catalog operations

Short descriptions and the syntax of catalog operations, which include creating, dropping, and describing an Apache Hive database and table from Apache Spark, helps you write HWC API apps.

### Methods

Three methods of executing catalog operations are supported: `.sql` (recommended), `.execute()` ( `spark.datasource.hive.warehouse.read.mode=JDBC_CLIENT`), or `.executeQuery()` for backward compatibility for LLAP reads.

- 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", "bigint").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

Brief descriptions of HWC API operations and examples cover how to read and write Apache Hive tables from Apache Spark. You learn how to 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>).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

[HMS storage](#)

[SPARK-20236](#)

## Commit transaction for Direct Reader

For Direct Reader operations, you need to know how to commit or abort transactions.

### About this task

A sql listener normally handles this task automatically when a dataframe operation or spark sql query finishes. In some cases when `.explain()`, `.rdd()`, or `.cache()` are invoked on a dataframe, the transaction is not automatically closed. In Spark Direct Reader mode, commit or abort a transaction as follows:

```
scala> com.qubole.spark.hiveacid.transaction.HiveAcidTxnManagerObject.commitTxn(spark)
scala> hive.commitTxn
```

Or, if you are using Hive Warehouse Connector with Direct Reader Mode enabled, you can invoke following API to commit transaction:

```
scala> hive.commitTxn
```

## Close HiveWarehouseSession operations

You need to know how to release locks that Apache Spark operations puts on Apache Hive resources. An example shows how and when to release these locks.

### About this task

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

## Use HWC for streaming

When using `HiveStreaming` to write a `DataFrame` to Apache Hive or an Apache Spark Stream to Hive, you need to know how 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](#)

## HWC API Examples

Examples of using the HWC API include how to create the `DataFrame` from any data source and include an option to write the `DataFrame` to an Apache Hive table.

### 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...`

When you write the `DataFrame`, the Hive Warehouse Connector creates the Hive table if it does not exist.

The following example uses Append mode.

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

val hive = com.hortonworks.hwc.HiveWarehouseSession.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 (not available
  in this release)
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();

// Closes the transaction started by the direct reader. The transaction is
  not committed if user
// uses rdd APIs.
void commitTxn();
}

```

### 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);

//Specify 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(String 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... 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

[HMS storage](#)

### Submit a Scala or Java application

A step-by-step procedure shows you how to submit an app based on the HiveWarehouseConnector library to run on Apache Spark Shell.

### Procedure

1. Choose a mode, for example JDBC, for your application.
2. Check that you meet the prerequisites for using JDBC read mode and make the HiveServer (HS2) connection as described for using the JDBC read mode.
3. Locate the hive-warehouse-connector-assembly jar in the /hive\_warehouse\_connector/ directory.  
For example, find hive-warehouse-connector-assembly-<version>.jar in the following location:

```
/opt/cloudera/parcels/CDH/jars
```

4. Add the connector jar and configurations to the app submission using the --jars option.  
Example syntax:

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

5. Add the path to app you wrote based on the HiveWarehouseConnector API.  
Example syntax:

```
<path to app>
```

For example:

```
spark-shell --jars /opt/cloudera/parcels/CDH/jars/hive-warehouse-connector-assembly-<version>.jar \  
--conf spark.sql.extensions=com.hortonworks.spark.sql.rule.Extensions \  
--conf spark.datasource.hive.warehouse.read.mode=JDBC_CLUSTER \  
--conf "spark.hadoop.hive.metastore.uris=thrift://172.27.74.137:9083" \  
--conf spark.datasource.hive.warehouse.load.staging.dir=<path to directory> \  
/home/myapps/myapp.jar
```

### Related Information

[Using Direct Reader mode](#)

[Using JDBC read mode](#)

[HMS storage](#)

## HWC integration with pyspark and Zeppelin

### Submit a Python app

A step-by-step procedure shows you how to submit a Python app based on the HiveWarehouseConnector library by submitting an application, and then adding a Python package.

### Procedure

1. Choose a read option, for example LLAP, for your application and check that you meet the configuration requirements, described earlier.
2. Configure a Spark-HiveServer connection, described earlier or, in your app submission include the appropriate --conf in step 4.
3. Locate the hive-warehouse-connector-assembly jar in the /hive\_warehouse\_connector/ directory.  
For example, find hive-warehouse-connector-assembly-<version>.jar in the following location:

```
/opt/cloudera/parcels/CDH/jars
```

4. Add the connector jar and configurations to the app submission using the `--jars` option.

Example syntax:

```
pyspark --jars <path to jars>/hive_warehouse_connector/hive-warehouse-co
nector-assembly-<version>.jar \
--conf <configuration properties>
```

5. Locate the `pyspark_hwc` zip package in the `/hive_warehouse_connector/` directory.

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

Example syntax:

```
--py-files <path>/hive_warehouse_connector/pyspark_hwc-<version>.zip
```

Example submission in JDBC execution mode:

```
pyspark --jars /opt/cloudera/parcels/CDH/jars/hive-warehouse-connector-a
ssembly-<version>.jar
--conf spark.sql.extensions=com.hortonworks.spark.sql.rule.Extensions \
--conf spark.datasource.hive.warehouse.read.mode=JDBC_CLUSTER \
--conf spark.datasource.hive.warehouse.load.staging.dir=<path to directory
> \
--py-files /opt/cloudera/parcels/CDH/lib/hive_warehouse_connector/pyspar
k_hwc-<version>.zip
```

### Related Information

[Using Direct Reader mode](#)

[Using JDBC read mode](#)

[HMS storage](#)

## Livy interpreter configuration

You can use the Hive Warehouse Connector in Zeppelin notebooks with the Livy interpreter by modifying or adding properties to your livy interpreter settings. The Livy interpreter accesses processing engines and data sources from the Zeppelin UI.

### Requirements

- Configurations require a livy prefix.
- A reference to the HWC jar on the local file system is required.

### Interpreter properties

- `livy.spark.hadoop.hive.metastore.uris` -

`thrift://<domain name>:<port>`

Example: `thrift://hwc-secure-1.hwc-secure.root.hwx.site:9083`

- `livy.spark.jars` -

`local:/opt/cloudera/parcels/<version>/jars/hive-warehouse-connector-assembly-<version>.jar`

Example: `local:/opt/cloudera/parcels/CDH-7.2.1-1.cdh7.2.1.p0.4847773/jars/hive-warehouse-connector-assembly-<version>.jar`

Use the local file ("local:").

- `livy.spark.kryo.registrator` - `com.qubole.spark.hiveacid.util.HiveAcidKryoRegistrator`
- `livy.spark.security.credentials.hiveserver2.enabled` - `true`
- `livy.spark.sql.extensions` - `com.qubole.spark.hiveacid.HiveAcidAutoConvertExtension`

- `livy.spark.sql.hive.hiveserver2.jdbc.url` -  
`jdbc:hive2://<domain name>:<port>/`  
`default;retries=5;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2`  
 Example: `jdbc:hive2://hwc-secure-1.hwc-secure.root.hwx.site:2181/`  
`default;retries=5;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2.`
- `livy.spark.sql.hive.hiveserver2.jdbc.url.principal` - `hive/_HOST@ROOT.HWX.SITE`

## Read and write Hive tables in Zeppelin

You can read and write Hive ACID tables from a Spark application using Zeppelin, a browser-based GUI for interactive data exploration, modeling, and visualization.

### Before you begin

You must be running spark application and have all the appropriate permissions to read the data from the hive warehouse directory for managed (ACID) tables.

### Procedure

1. Open the livy configuration file on the livy node and configure HWC Spark Direct Reader mode.

```
%livy2.conf
...
livy.spark.hadoop.hive.metastore.uris thrift://hwc-secure-1.hwc-secure.r
oot.hwx.site:9083
livy.spark.jars local:/opt/cloudera/parcels/CDH-7.2.1-1.cdh7.2.1.p0.484777
3/jars/hive-warehouse-connector-assembly-<version>.jar
livy.spark.kryo.registrator com.qubole.spark.hiveacid.util.HiveAcidKyroRe
gistrator
livy.spark.security.credentials.hiveserver2.enabled true
livy.spark.sql.extensions com.hortonworks.spark.sql.rule.Extensions
livy.spark.sql.hive.hiveserver2.jdbc.url jdbc:hive2://hwc-secure-1.hwc-se
cure.root.hwx.site:2181/default;retries=5;serviceDiscoveryMode=zooKeeper
;zooKeeperNamespace=hiveserver2
livy.spark.sql.hive.hiveserver2.jdbc.url.principal hive/_HOST@ROOT.HWX.S
ITE
...

```

2. In a Zeppelin notebook, read a Hive ACID table.

```
sql("show tables").show
```



```
sql("select * from hwc2").show
```

Zeppelin Notebook Job

## my-spark-notebook-1

```
spark.version

// Reading through HWC direct reader

sql("show tables").show

sql("select * from hwc2").show
```

Previous livy session is expired, new livy session is created.

| database | tablename | temporary |
|----------|-----------|-----------|
| default  | hwc1      | false     |
| default  | hwc2      | false     |
| default  | sample_07 | false     |
| default  | sample_08 | false     |
| default  | t1        | false     |
| default  | t2        | false     |
| default  | t3        | false     |
| default  | web_logs  | false     |

| c1 | c2  |
|----|-----|
| 1  | one |
| 2  | two |

### 3. Perform a batch write from Spark.

```
val hive = com.hortonworks.hwc.HiveWarehouseSession.session(spark).build()
val df = Seq(
  (1, "a", 1.1),
  (2, "b", 2.2),
  (3, "c", 3.3)
).toDF("col1", "col2", "col3")
val tableName = "t3"
hive.dropTable(tableName, true, true)
df.write.format("com.hortonworks.spark.sql.hive.llap.HiveWarehouseConnector").mode("append").option("table", tableName).save()
```

#### 4. Read data from table t3.

```
sql("select * from t3").show
```

Output is:

```
tableName: String = t3
+-----+-----+-----+
| col1 | col2 | col3 |
+-----+-----+-----+
|    1 |    a |   1.1 |
|    2 |    b |   2.2 |
|    3 |    c |   3.3 |
+-----+-----+-----+
```

## Apache Hive-Kafka integration

As an Apache Hive user, you can connect to, analyze, and transform data in Apache Kafka from Hive. You can offload data from Kafka to the Hive warehouse. Using Hive-Kafka integration, you can perform actions on real-time data and incorporate streamed data into your application.

You connect to Kafka data from Hive by creating an external table that maps to a Kafka topic. The table definition includes a reference to a Kafka storage handler that connects to Kafka. On the external table, Hive-Kafka integration supports ad hoc queries, such as questions about data changes in the stream over a period of time. You can transform Kafka data in the following ways:

- Perform data masking
- Join dimension tables or any stream
- Aggregate data
- Change the SerDe encoding of the original stream
- Create a persistent stream in a Kafka topic

You can achieve data offloading by controlling its position in the stream. The Hive-Kafka connector supports the following serialization and deserialization formats:

- JsonSerDe (default)
- OpenCSVSerde
- AvroSerDe

### Related Information

[Apache Kafka Documentation](#)

## Create a table for a Kafka stream

You can create an external table in Apache Hive that represents an Apache Kafka stream to query real-time data in Kafka. You use a storage handler and table properties that map the Hive database to a Kafka topic and broker. If the Kafka data is not in JSON format, you alter the table to specify a serializer-deserializer for another format.

### Procedure

1. Get the name of the Kafka topic you want to query to use as a table property.  
For example: "kafka.topic" = "wiki-hive-topic"
2. Construct the Kafka broker connection string.  
For example: "kafka.bootstrap.servers"="kafka.hostname.com:9092"

3. Create an external table named `kafka_table` by using `'org.apache.hadoop.hive.kafka.KafkaStorageHandler'`, as shown in the following example:

```
CREATE EXTERNAL TABLE kafka_table
(`timestamp` timestamp, `page` string, `newPage` boolean,
added int, deleted bigint, delta double)
STORED BY 'org.apache.hadoop.hive.kafka.KafkaStorageHandler'
TBLPROPERTIES
("kafka.topic" = "test-topic", "kafka.bootstrap.servers"="node2:9092");
```

4. If the default JSON serializer-deserializer is incompatible with your data, choose another format in one of the following ways:

- Alter the table to use another supported serializer-deserializer. For example, if your data is in Avro format, use the Kafka serializer-deserializer for Avro:

```
ALTER TABLE kafka_table SET TBLPROPERTIES ("kafka.serde.class"="org.apac
he.hadoop.hive.serde2.avro.AvroSerDe");
```

- Create an external table that specifies the table in another format. For example, create a table named that specifies the Avro format in the table definition:

```
CREATE EXTERNAL TABLE kafka_t_avro
(`timestamp` timestamp, `page` string, `newPage` boolean,
added int, deleted bigint, delta double)
STORED BY 'org.apache.hadoop.hive.kafka.KafkaStorageHandler'
TBLPROPERTIES
("kafka.topic" = "test-topic",
"kafka.bootstrap.servers"="node2:9092"
-- STORE AS AVRO IN KAFKA
"kafka.serde.class"="org.apache.hadoop.hive.serde2.avro.AvroSerDe");
```

### Related Information

[Apache Kafka Documentation](#)

## Querying Kafka data

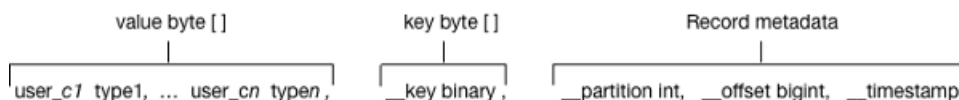
You can get useful information, including Kafka record metadata from a table of Kafka data by using typical Hive queries.

Each Kafka record consists of a user payload key (byte []) and value (byte[]), plus the following metadata fields:

- Partition int32
- Offset int64
- Timestamp int64

The Hive row represents the dual composition of Kafka data:

- The user payload serialized in the value byte array
- The metadata: key byte array, partition, offset, and timestamp fields



In the Hive representation of the Kafka record, the key byte array is called `__key` and is of type binary. You can cast `__key` at query time. Hive appends `__key` to the last column derived from value byte array, and appends the partition, offset, and timestamp to `__key` columns that are named accordingly.

### Related Information

[Apache Kafka Documentation](#)

## Query live data from Kafka

You can get useful information from a table of Kafka data by running typical queries, such as counting the number of records streamed within an interval of time or defining a view of streamed data over a period of time.

### Before you begin

This task requires Kafka 0.11 or later to support time-based lookups and prevent full stream scans.

### About this task

This task assumes you created a table named `kafka_table` for a Kafka stream.

### Procedure

1. List the table properties and all the partition or offset information for the topic.  
`DESCRIBE EXTENDED kafka_table;`
2. Count the number of Kafka records that have timestamps within the past 10 minutes.

```
SELECT COUNT(*) FROM kafka_table
  WHERE `__timestamp` > 1000 * to_unix_timestamp(CURRENT_TIMESTAMP - interval '10' MINUTES);
```

Such a time-based seek requires Kafka 0.11 or later, which has a Kafka broker that supports time-based lookups; otherwise, this query leads to a full stream scan.

3. Define a view of data consumed within the past 15 minutes and mask specific columns.

```
CREATE VIEW last_15_minutes_of_kafka_table AS SELECT `timestamp`, `user`,
  delta,
  ADDED FROM kafka_table
  WHERE `__timestamp` > 1000 * to_unix_timestamp(CURRENT_TIMESTAMP - interval '15' MINUTES) ;
```

4. Create a dimension table.

```
CREATE TABLE user_table (`user` string, `first_name` string , age int, gender string, comments string) STORED as ORC ;
```

5. Join the view of the stream over the past 15 minutes to `user_table`, group by gender, and compute aggregates over metrics from fact table and dimension tables.

```
SELECT SUM(added) AS added, SUM(deleted) AS deleted, AVG(delta) AS delta,
  AVG(age) AS avg_age , gender
  FROM last_15_minutes_of_kafka_table
  JOIN user_table ON `last_15_minutes_of_kafka_table`.`user` = `user_table`.`user`
  GROUP BY gender LIMIT 10;
```

6. Perform a classical user retention analysis over the Kafka stream consisting of a stream-to-stream join that runs adhoc queries on a view defined over the past 15 minutes.

```
-- Stream join over the view itself
-- Assuming l15min_wiki is a view of the last 15 minutes
SELECT COUNT( DISTINCT activity.`user`) AS active_users,
  COUNT(DISTINCT future_activity.`user`) AS retained_users
  FROM l15min_wiki AS activity
  LEFT JOIN l15min_wiki AS future_activity ON activity.`user` = future_activity.`user`
  AND activity.`timestamp` = future_activity.`timestamp` - interval '5' minutes ;

-- Stream-to-stream join
```

```
-- Assuming wiki_kafka_hive is the entire stream.
SELECT floor_hour(activity.`timestamp`), COUNT( DISTINCT activity.`user`)
  AS active_users,
  COUNT(DISTINCT future_activity.`user`) as retained_users
FROM wiki_kafka_hive AS activity
LEFT JOIN wiki_kafka_hive AS future_activity ON activity.`user` = future_
activity.`user`
AND activity.`timestamp` = future_activity.`timestamp` - interval '1' ho
ur
GROUP BY floor_hour(activity.`timestamp`);
```

### Related Information

[Apache Kafka Documentation](#)

## Perform ETL by ingesting data from Kafka into Hive

You can extract, transform, and load a Kafka record into Hive in a single transaction.

### Procedure

1. Create a table to represent source Kafka record offsets.

```
CREATE TABLE kafka_table_offsets(partition_id int, max_offset bigint, in
sert_time timestamp);
```

2. Initialize the table.

```
INSERT OVERWRITE TABLE kafka_table_offsets
SELECT `__partition`, min(`__offset`) - 1, CURRENT_TIMESTAMP
FROM wiki_kafka_hive
GROUP BY `__partition`, CURRENT_TIMESTAMP;
```

3. Create the destination table.

```
CREATE TABLE orc_kafka_table (partition_id int, koffset bigint, ktimestamp
  bigint,
  `timestamp` timestamp , `page` string, `user` string, `diffurl` string,
  `isrobot` boolean, added int, deleted int, delta bigint
) STORED AS ORC;
```

4. Insert Kafka data into the ORC table.

```
FROM wiki_kafka_hive ktable JOIN kafka_table_offsets offset_table
ON (ktable.`__partition` = offset_table.partition_id
AND ktable.`__offset` > offset_table.max_offset )
INSERT INTO TABLE orc_kafka_table
SELECT `__partition`, `__offset`, `__timestamp`,
  `timestamp`, `page`, `user`, `diffurl`, `isrobot`, added , deleted , del
ta
INSERT OVERWRITE TABLE kafka_table_offsets
SELECT `__partition`, max(`__offset`), CURRENT_TIMESTAMP
GROUP BY `__partition`, CURRENT_TIMESTAMP;
```

5. Check the insertion.

```
SELECT MAX(`koffset`) FROM orc_kafka_table LIMIT 10;

SELECT COUNT(*) AS c FROM orc_kafka_table
GROUP BY partition_id, koffset HAVING c > 1;
```

6. Repeat step 4 periodically until all the data is loaded into Hive.

## Writing data to Kafka

You can extract, transform, and load a Hive table to a Kafka topic for real-time streaming of a large volume of Hive data. You need some understanding of write semantics and the metadata columns required for writing data to Kafka.

### Write semantics

The Hive-Kafka connector supports the following write semantics:

- At least once (default)
- Exactly once

#### At least once (default)

The default semantic. At least once is the most common write semantic used by streaming engines. The internal Kafka producer retries on errors. If a message is not delivered, the exception is raised to the task level, which causes a restart, and more retries. The At least once semantic leads to one of the following conclusions:

- If the job succeeds, each record is guaranteed to be delivered at least once.
- If the job fails, some of the records might be lost and some might not be sent.

In this case, you can retry the query, which eventually leads to the delivery of each record at least once.

#### Exactly once

Following the exactly once semantic, the Hive job ensures that either every record is delivered exactly once, or nothing is delivered. You can use only Kafka brokers supporting the Transaction API (0.11.0.x or later). To use this semantic, you must set the table property "kafka.write.semantic"="EXACTLY\_ONCE".

### Metadata columns

In addition to the user row payload, the insert statement must include values for the following extra columns:

#### \_\_key

Although you can set the value of this metadata column to null, using a meaningful key value to avoid unbalanced partitions is recommended. Any binary value is valid.

#### \_\_partition

Use null unless you want to route the record to a particular partition. Using a nonexistent partition value results in an error.

#### \_\_offset

You cannot set this value, which is fixed at -1.

#### \_\_timestamp

You can set this value to a meaningful timestamp, represented as the number of milliseconds since epoch. Optionally, you can set this value to null or -1, which means that the Kafka broker strategy sets the timestamp column.

### Related Information

[Apache Kafka Documentation](#)

## Write transformed Hive data to Kafka

You can change streaming data and include the changes in a stream. You extract a Kafka input topic, transform the record in Hive, and load a Hive table back into a Kafka record.

## About this task

This task assumes that you already queried live data from Kafka. When you transform the record in the Hive execution engine, you compute a moving average over a window of one minute. The resulting record that you write back to another Kafka topic is named `moving_avg_wiki_kafka_hive`.

## Procedure

1. Create an external table to represent the Hive data that you want to load into Kafka.

```
CREATE EXTERNAL TABLE moving_avg_wiki_kafka_hive
(`channel` string, `namespace` string, `page` string, `timestamp` timestamp
, avg_delta double )
STORED BY 'org.apache.hadoop.hive.kafka.KafkaStorageHandler'
TBLPROPERTIES
  ("kafka.topic" = "moving_avg_wiki_kafka_hive",
   "kafka.bootstrap.servers"="kafka.hostname.com:9092",
   -- STORE AS AVRO IN KAFKA
   "kafka.serde.class"="org.apache.hadoop.hive.serde2.avro.AvroSerDe");
```

2. Insert data that you select from the Kafka topic back into the Kafka record.

```
INSERT INTO TABLE moving_avg_wiki_kafka_hive
SELECT `channel`, `namespace`, `page`, `timestamp`,
  AVG(delta) OVER (ORDER BY `timestamp` ASC ROWS BETWEEN 60 PRECEDING AND
    CURRENT ROW) AS avg_delta,
  null AS `__key`, null AS `__partition`, -1 AS `__offset`, to_epoch_milli
(CURRENT_TIMESTAMP) AS `__timestamp`
FROM l15min_wiki;
```

The timestamps of the selected data are converted to milliseconds since epoch for clarity.

## Related Information

[Query live data from Kafka](#)

## Set consumer and producer properties as table properties

You can use Kafka consumer and producer properties in the TBLPROPERTIES clause of a Hive query. By prefixing the key with `kafka.consumer` or `kafka.producer`, you can set the table properties.

## Procedure

For example, if you want to inject 5000 poll records into the Kafka consumer, use the following syntax.

```
ALTER TABLE kafka_table SET TBLPROPERTIES ("kafka.consumer.max.poll.records"
="5000");
```

## Kafka storage handler and table properties

You use the Kafka storage handler and table properties to specify the query connection and configuration.

### Kafka storage handler

You specify `'org.apache.hadoop.hive.kafka.KafkaStorageHandler'` in queries to connect to, and transform a Kafka topic into, a Hive table. In the definition of an external table, the storage handler creates a view over a single Kafka

topic. For example, to use the storage handler to connect to a topic, the following table definition specifies the storage handler and required table properties: the topic name and broker connection string.

```
CREATE EXTERNAL TABLE kafka_table
  (`timestamp` timestamp, `page` string, `newPage` boolean,
   added int, deleted bigint, delta double)
  STORED BY 'org.apache.hadoop.hive.kafka.KafkaStorageHandler'
  TBLPROPERTIES
    ("kafka.topic" = "test-topic", "kafka.bootstrap.servers"="localhost:90
    92");
```

You set the following table properties for with the Kafka storage handler:

**kafka.topic**

The Kafka topic to connect to

**kafka.bootstrap.servers**

The broker connection string

### Storage handler-based optimizations

The storage handler can optimize reads using a filter push-down when you run a query such as the following time-based lookup supported on Kafka 0.11 or later:

```
SELECT COUNT(*) FROM kafka_table
  WHERE `__timestamp` > 1000 * to_unix_timestamp(CURRENT_TIMESTAMP - int
  erval '10' MINUTES) ;
```

The Kafka consumer supports seeking on the stream based on an offset, which the storage handler leverages to push down filters over metadata columns. The storage handler in the example above performs seeks based on the Kafka record `__timestamp` to read only recently arrived data.

The following logical operators and predicate operators are supported in the WHERE clause:

Logical operators: OR, AND

Predicate operators: <, <=, >=, >, =

The storage handler reader optimizes seeks by performing partition pruning to go directly to a particular partition offset used in the WHERE clause:

```
SELECT COUNT(*) FROM kafka_table
  WHERE (`__offset` < 10 AND `__offset` > 3 AND `__partition` = 0)
  OR (`__partition` = 0 AND `__offset` < 105 AND `__offset` > 99)
  OR (`__offset` = 109);
```

The storage handler scans partition 0 only, and then read only records between offset 4 and 109.

### Kafka metadata

In addition to the user-defined payload schema, the Kafka storage handler appends to the table some additional columns, which you can use to query the Kafka metadata fields:

**\_\_key**

Kafka record key (byte array)

**\_\_partition**

Kafka record partition identifier (int 32)

**\_\_offset**

Kafka record offset (int 64)

**\_\_timestamp**



Kafka record timestamp (int 64)

The partition identifier, record offset, and record timestamp plus a key-value pair constitute a Kafka record. Because the key-value is a 2-byte array, you must use SerDe classes to transform the array into a set of columns.

### Table Properties

You use certain properties in the TBLPROPERTIES clause of a Hive query that specifies the Kafka storage handler.

| Property                            | Description   | Required | Default                                 |
|-------------------------------------|---|----------|---|
| kafka.topic                         | Kafka topic name to map the table to  | Yes      | null                                    |
| kafka.bootstrap.servers             | Table property indicating the Kafka broker connection string  | Yes      | null                                    |
| kafka.serde.class                   | Serializer and Deserializer class implementation  | No       | org.apache.hadoop.hive.serde2.JsonSerDe |
| hive.kafka.poll.timeout.ms          | Parameter indicating Kafka Consumer poll timeout period in milliseconds. (This is independent of internal Kafka consumer timeouts.) | No       | 5000 (5 Seconds)                        |
| hive.kafka.max.retries              | Number of retries for Kafka metadata fetch operations   | No       | 6                                       |
| hive.kafka.metadata.poll.timeout.ms | Number of milliseconds before consumer timeout on fetching Kafka metadata   | No       | 30000 (30 Seconds)                      |
| kafka.write.semantic                | Writer semantic with allowed values of NONE, AT_LEAST_ONCE, EXACTLY_ONCE  | No       | AT_LEAST_ONCE                           |

## Connecting Hive to BI tools using a JDBC/ODBC driver

To query, analyze, and visualize data stored in CDP, you use drivers provided by Cloudera to 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.

## Getting the JDBC or ODBC driver

You learn how to download the Cloudera Hive and Impala JDBC drivers to give clients outside the cluster access to your SQL engines.

### Procedure

1. Download the latest Hive JDBC driver for CDP from the [Hive JDBC driver download page](#).
2. Go to the [Impala JDBC driver](#) page, and download the latest Impala JDBC driver.
3. Follow JDBC driver installation instructions on the download page.

## Integrating Hive and a BI tool

After you download a ODBC driver or JDBC driver and give it to your client, you need to provide the information that the BI tool requires to connect to Hive in CDP.

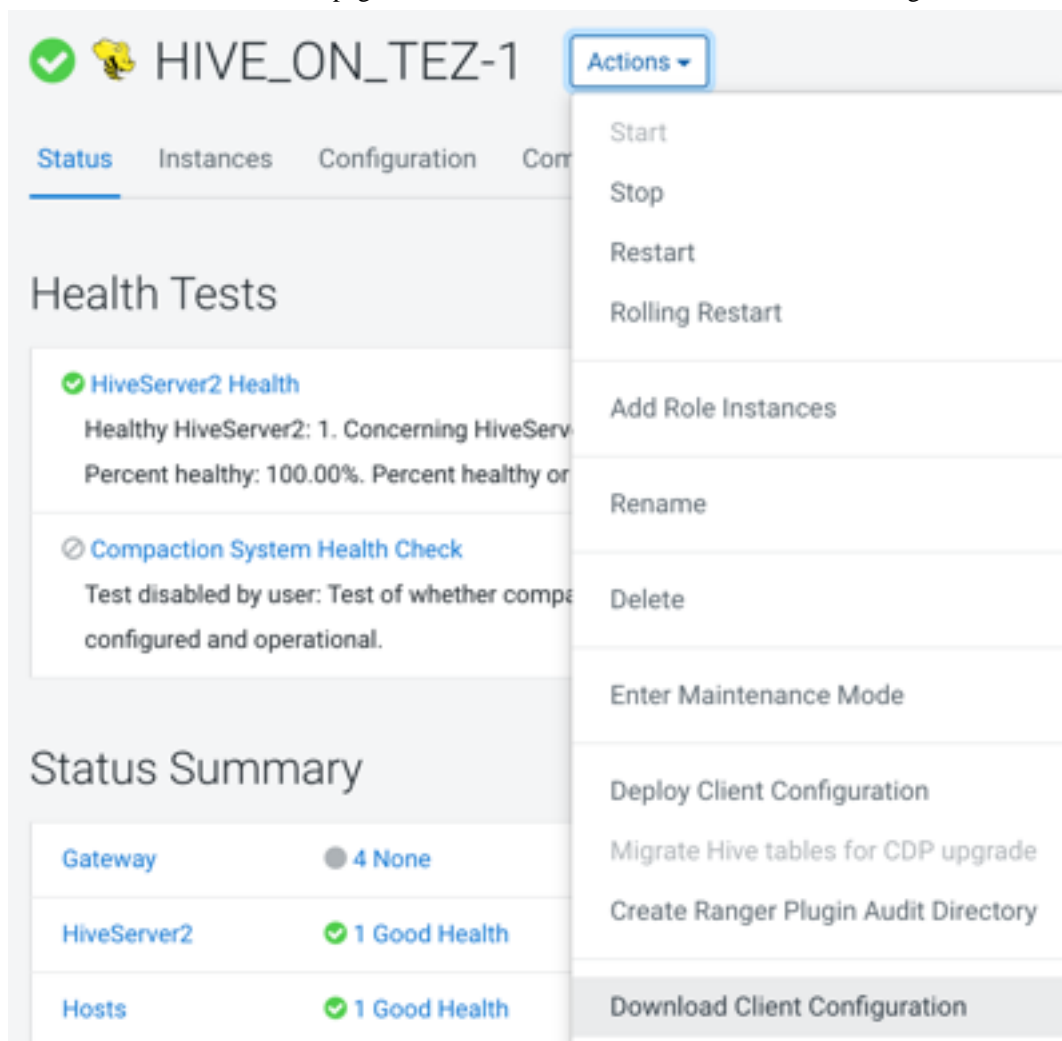
### Before you begin

You have downloaded the JDBC or ODBC driver.

### 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. For example, check the client documentation about where to put the driver.
2. In Cloudera Manager (CM), click `Clusters Hive on Tez` to go to the Hive on Tez service page.

- From the Hive on Tez service page, click Actions and select Download Client Configuration.



- Unpack `hive_on_tez-clientconfig.zip`, open the `beeline-site.xml` file, 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
```

- In the BI tool, configure the JDBC connection using the JDBC URL and driver class name, `com.cloudera.hive.jdbc.HS2Driver`.

## 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 transportMode 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>; ...<br>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>; ...<br>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, HiveServer2 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=<truststore_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 RDBMS

Using the `JdbcStorageHandler`, you can connect Apache 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;
```

## Set up JDBCStorageHandler for Postgres

If you use Enterprise PostgreSQL as the backend HMS database, you need to put the `JDBCStorageHandler` JAR in a central place.

### About this task

The Postgres Enterprise server comes with its own JDBC driver. The driver file is installed in the Hive lib directory. When you execute a query as a YARN application, the Class not found exception is thrown on worker nodes. The YARN container cannot include the jar file in the classpath unless you place the JAR in a central location.

Place the JAR in aux jars or provide the path to aux jars.

### Procedure

1. In CDP Private Cloud Base, click **Cloudera Manager Clusters** and select the Hive service, for example, HIVE.
2. Click **Configuration** and search for **Hive Auxiliary JARs Directory**.
3. Specify a directory value for the **Hive Aux JARs** property if necessary, or make a note of the path.
4. Upload the JAR to the specified directory on all HiveServer instances.