Cloudera Runtime 7.2.11

Integrating Apache Hive with Spark and Kafka

Date published: 2019-08-21
Date modified: 2021-09-08

https://docs.cloudera.com/

Legal Notice

Contents

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. However, you must know that accessing external tables through HWC is slower as compared to accessing external tables through native Spark libraries.

Supported applications and operations

The Hive Warehouse Connector supports the following applications:

- Spark 3
- Spark shell
- PySpark
- The spark-submit script
- sparklyr
- 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 pre-existing or new table in Parquet, ORC, AVRO, or Textfile formats
- 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

Set up

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 Cloudera 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/co
ntent/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 Spark 3 parcel

HWC supports Spark 3 through Cloudera Runtime.

The HWC binaries are located in /opt/cloudera/parcels/CDH/lib/hwc_for_spark3/. This directory contains HWC jar and a python zip. Use these binaries to launch Spark applications in Scala, Java, or Python.

The following files are in /opt/cloudera/parcels/CDH/lib/hwc_for_spark3/.

- hive-warehouse-connector-spark3-assembly-<version>.jar
- pyspark_hwc-spark3-<version>.zip

HWC configurations

Learn about the configurations that are required by Spark when using HWC. Setting HWC configurations has been simplified. As a cluster administrator, you can specify the required configurations in Cloudera Manager and then enable HWC by setting spark.cloudera.useHWC=true. You can either add this property to spark-defaults.conf to enable HWC for all Spark jobs or enable HWC for a specific Spark job by using the --conf option in spark-shell or spark-submit.

Add the following configurations in the spark-defaults.conf file by going to Clusters Spark 3 Configuration , search for spark-defaults.conf in Cloudera Manager. This is a one-time activity that is performed by a cluster administrator.

- spark.sql.hive.hiveserver2.jdbc.url="jdbc:hive2://<jdbc-url>"

- spark.datasource.hive.warehouse.read.mode=<mode>
- spark.security.credentials.hiveserver2.enabled=true
- spark.sql.hive.hiveserver2.jdbc.url.principal=<principal>/_HOST@ROOT.HWX.SITE
- spark.datasource.hive.warehouse.load.staging.dir=<hdfs://>/tmp/staging/hwc

Setting spark.cloudera.useHWC=true also adds all the jar files present in /opt/cloudera/parcels/CDH/lib/hwc_for_spark3/ to the Spark classpath along with adding the following configurations to Spark configuration:

- spark.kryo.registrator=com.qubole.spark.hiveacid.util.HiveAcidKyroRegistrator
- spark.sql.extensions=com.hortonworks.spark.sql.rule.Extensions

Note: You can choose to override the configuration values during runtime by using the --conf option while submitting a Spark job.

The following example shows how you can enable HWC when launching the Spark shell:

```
[root@sim-hwc-1 hive_warehouse_connector]# spark-shell  --conf spark.clouder
a.useHWC=true  --master yarn

Spark context available as 'sc' (master = yarn, app id = application_17325
48085468_0001).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.4.1.7.3.1.0-178
      /_/

Using Scala version 2.12.17 (OpenJDK 64-Bit Server VM, Java 1.8.0_232)
Type in expressions to have them evaluated.
Type :help for more information.

scala> val hwc = com.hortonworks.hwc.HiveWarehouseSession.session(spark).bui
ld()
hive: com.hortonworks.spark.sql.hive.llap.HiveWarehouseSessionImpl = com.ho
rtonworks.spark.sql.hive.llap.HiveWarehouseSessionImpl@5ac9ae05
scala> hwc.sql("select * from acidtable").show
+---+-----+----+-----+----+-----+
| id| name|col3| col4|col5| col6|
+---+-----+----+-----+----+-----+
|  1|name1|   2|name2|   3|name3|
|  4|name4|   5|name5|   6|name6|
+---+-----+----+-----+----+-----+
```

Optional HWC configurations

Optionally, you can set the following properties:

- spark.datasource.hive.warehouse.write.path.strictColumnNamesMapping — Validates the mapping of columns against those in Hive to alert the user to input errors. Default = true.
- spark.sql.hive.conf.list — Propagates one or more configuration properties from the HWC to Hive. Set properties on the command line using the --conf option. For example:

  ```
  --conf spark.sql.hive.conf.list="hive.vectorized.execution.filesink.arro
  w.native.enabled=true;hive.vectorized.execution.enabled=true"
  ```

Do not attempt to set spark.sql.hive.conf.list programmatically.

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.g
etOrCreate
    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")
  }
```

Important: Spark 3 does not support automatic table creation using the SaveMode.Append and SaveMode .Overwrite modes. The table that you are writing to must exist in order to append or overwrite data in it. Instead, you can use the SaveMode.ErrorIfExists or SaveMode.Ignore modes.

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 st
ring, 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)");
    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/hwc_for_spark3/hive-wareho
use-connector-spark3-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").getOrCre
ate()
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).optio
n("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/hwc_for_spark3/hive-wa
rehouse-connector-spark3-assembly-<version>.jar \
    --py-files /opt/cloudera/parcels/CDH/lib/hwc_for_spark3/pyspark_hwc-<v
ersion>.zip \
    ...More spark/HWC confs...
    ...More spark/HWC confs...
    /path-to-python-app/hwc-app.py
```

Use with Sparklyr

You can access Hive tables through R by loading the sparklyr library along with the SparklyrHWC package available in /opt/cloudera/parcels/CDH/lib/hwc_for_spark3/, which can be used to trigger HWC APIs from R.

```
library(sparklyr)
library(SparklyrHWC, lib.loc = c(file.path(“<path to SparklyrHWC>")))
#Set env variables
Sys.setenv(SPARK_HOME = "/opt/cloudera/parcels/CDH/lib/spark/")
Sys.setenv(HADOOP_HOME = "/opt/cloudera/parcels/CDH/lib/hadoop")

#Configurations needed to use spark-acid and related configurations.
config <- spark_config()
config$spark.sql.hive.hiveserver2.jdbc.url="jdbc:hive2://<url>:10000/defau
lt"
config$spark.datasource.hive.warehouse.user.name="hive"
config$spark.hadoop.hive.metastore.uris="thrift://<url>:9083"
config$spark.sql.extensions="com.hortonworks.spark.sql.rule.Extensions"
config$spark.kryo.registrator="com.qubole.spark.hiveacid.util.HiveAcidKyro
Registrator"
config$spark.datasource.hive.warehouse.read.mode="DIRECT_READER_V2"

#Build HWC session
hs <- build(HiveWarehouseBuilder.session(sc))

#Use database
```

```
sparklyr::sdf_sql(sc,"use test")
#Reading a managed table using spark acid direct-reader
intDf <- sparklyr::spark_read_table(sc, 'emp_hwc')
#Converts SparkDataframe to R dataframe
 sparklyr::sdf_collect(intDf1)
#Writing into a managed table
#Read first table
intDf <- sparklyr::spark_read_table(sc, 'emp_hwc')
#read second table
intDf1 <-  sparklyr::spark_read_table(sc, 'emp_overwrite')
#Commit transaction if read using spark-acid
commitTxn(hs)
#Append the second table, to the first.
SparklyrHWC::spark_write_table('emp_hwc',intDf1,'append')
#Overwrite the first table with the second table.
SparklyrHWC::spark_write_table('emp_hwc',intDf1,'overwrite')

#Using HWC Api's
#create a table from existing table
SparklyrHWC::executeUpdate(hs,"create table hwc1 as select * from 'emp_hwc'
")
#Execute query
hwcDf <- SparklyrHWC::executeQuery(hs, "select * from hwc1")
#convert into R dataframe.
hwcSdf <- sparklyr::sdf_copy_to(sc, hwcDf)
```

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, Secure access mode, and HWC and DataFrames API limitations.

General HWC limitations are:

- The spark and livy thrift servers are not supported.
- Spark UDFs 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.

Hive Warehouse Connector is optimized for sequential query execution. Running multiple jobs concurrently can lead to execution failures. Hive Warehouse Connector functions best when queries are executed one after another.

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.HW
  X.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 or DIRECT_READER_V2
- JDBC_CLUSTER
- SECURE_ACCESS

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

Apache Spark executor task statistics

Using secure access 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.HiveAcidKy
roRegistrator \
--conf spark.sql.hive.hiveserver2.jdbc.url="jdbc:hive2://hwc-2.hwc.root.
hwx.site:2181/default;retries=5;serviceDiscoveryMode=zooKeeper;zooKeeper
Namespace=hiveserver2" \
--conf spark.sql.hive.hiveserver2.jdbc.url.principal=hive/_HOST@ROOT.HW
X.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.HiveAcidKy
roRegistrator
--conf spark.sql.hive.hiveserver2.jdbc.url="jdbc:hive2://hwc-2.hwc.root.hw
x.site:2181/default;retries=5;serviceDiscoveryMode=zooKeeper;zooKeeperNa
mespace=hiveserver2"
--conf spark.sql.hive.hiveserver2.jdbc.url.prinicpal=hive/_HOST@ROOT.HW
X.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
```

Note: HWC supports the hive.sql() API for executing queries. The .execute() and .executeQuery() methods are deprecated and it is recommended that you use the hive.sql() method.

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. You use this mode if you do not need production-level Ranger authorization. Direct Reader mode does not support Ranger authorization. Direct Reader does support Spark-consistent timestamps in this release and later.

In Direct Reader mode, Spark reads the data directly from the managed table location using the transaction snapshot.

Requirements and recommendations

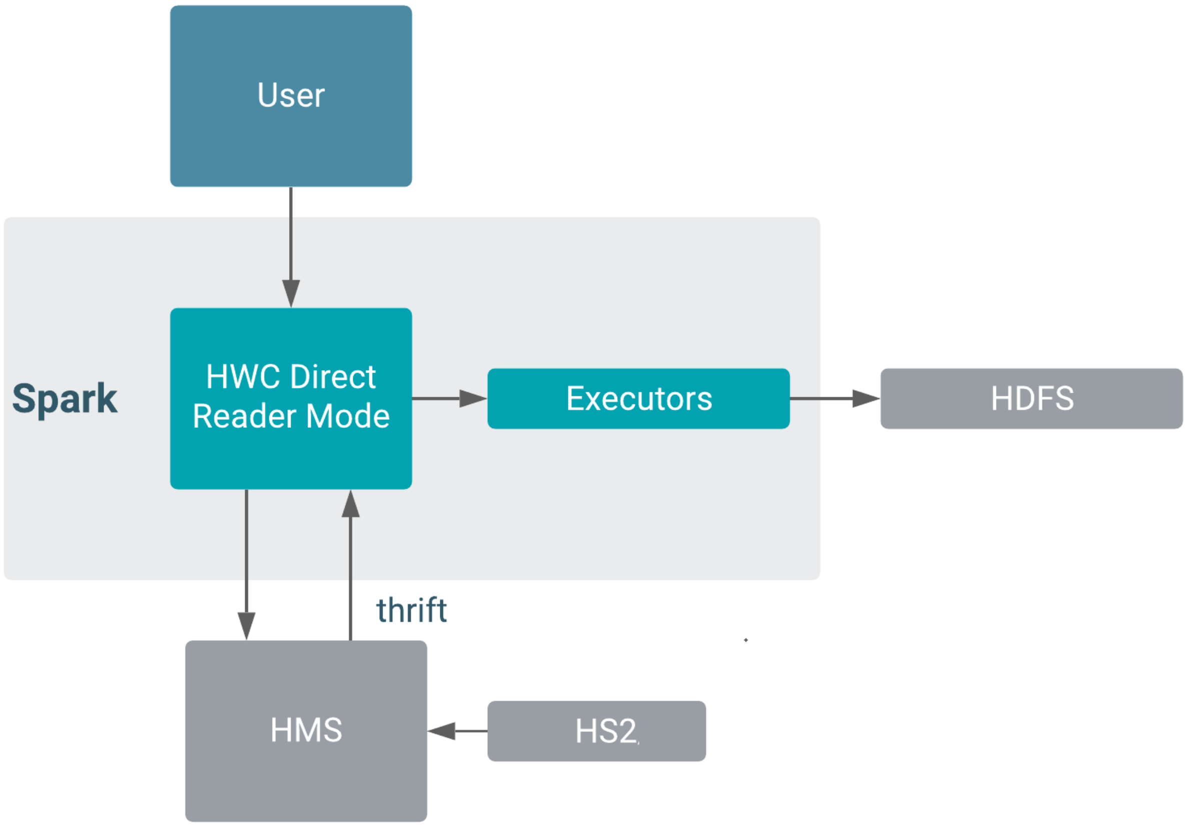
Spark Direct Reader mode requires a connection to Hive metastore. A HiveServer (HS2) connections is not needed.

The user running the Spark session must have access (read access on data and execute access on folder structure) to the underlying data of the table on the filesystem and to HMS.

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 for Direct Reader reads.



Using the timestamp type

When reading tables with Direct Reader, the behavior of the timestamp type is consistent with Spark. If you use the same storage format to store a timestamp value in an external table and a managed, transactional table, you see exactly the same value using the native Spark reader as you see using Direct Reader.

When querying a Hive table using Beeline, the behavior of the timestamp type is also consistent with Spark, assuming the Spark JVA and session time zones are consistent with the following exception: tables stored in Parquet format. Spark timestamp values differ depending on the value of spark.sql.hive.convertMetastoreParquet even for the external tables. Direct Reader reads of tables stored in Parquet format yields results as if spark.sql.hive.convertMetastoreParqu et is false.

This design ensures that the timestamp values remain consistent across all the formats when read by Direct Reader. In the following section, examples show how Spark and Direct Reader behave with timestamps. The examples query external and transactional tables in different formats. All the tables were created using Hive beeline. A timestamp value “1989-01-05 00:00:00” was inserted. The Spark native reader reads external tables. Direct Reader reads transactional tables. A union of all the dataframes make the result more presentable and comparable.

With spark.sql.hive.convertMetastoreParquet = true (default in spark)

```
+-------------------------+-------------------+
|type                     |timestamp          |
+-------------------------+-------------------+
|External parquet         |1989-01-04 16:00:00|
|Transactional parquet    |1989-01-05 00:00:00|
|External orc             |1989-01-05 00:00:00|
|Transactional orc        |1989-01-05 00:00:00|
|External avro            |1989-01-05 00:00:00|
|Transactional avro       |1989-01-05 00:00:00|
|External text           |1989-01-05 00:00:00|
|Transactional text      |1989-01-05 00:00:00|
+------------------------+-------------------+
```

With spark.sql.hive.convertMetastoreParquet = false

```
+------------------------+-------------------+
|type                    |timestamp          |
+------------------------+-------------------+
|External parquet        |1989-01-05 00:00:00|
|Transactional parquet   |1989-01-05 00:00:00|
|External orc            |1989-01-05 00:00:00|
|Transactional orc       |1989-01-05 00:00:00|
|External avro           |1989-01-05 00:00:00|
|Transactional avro      |1989-01-05 00:00:00|
|External text           |1989-01-05 00:00:00|
|Transactional text      |1989-01-05 00:00:00|
+------------------------+-------------------+
```

Direct Reader is aligned to Spark for all the storage formats, including convertMetastoreParquet.

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.HiveAcidKyroRegistrator.

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.HiveAcidKy
roRegistrator \
--conf spark.sql.hive.hiveserver2.jdbc.url="jdbc:hive2://<domain name>:<
port>/default;principal=hive/_HOST@ROOT.HWX.SITE;retries=5;serviceDiscov
eryMode=zooKeeper;zooKeeperNamespace=hiveserver2"

--conf spark.datasource.hive.warehouse.read.mode=DIRECT_READER_V2
```

3. Read data in table customer.
View data in table customer.

```
scala>
hive.sql("select c_customer_sk, c_customer_id, c_last_name, c_birth_count
ry from customer where c_birth_year=1983 limit 2 ").show()

21/02/08 11:03:31 INFO rule.HWCSwitchRule: using DIRECT_READER_V2 extens
ion for reading
+-------------+---------------+-----------+---------------+

|c_customer_sk|   c_customer_id|c_last_name|c_birth_country|
+-------------+---------------+-----------+---------------+
|        55634|AAAAAAAACFJNAAAA|   Campbell|       THAILAND|
|        74213|AAAAAAAAFOBCBAAA|    Hudgins|     KYRGYZSTAN|
+-------------+---------------+-----------+---------------+
```

Direct Reader configuration properties

You need to know the property names and valid values for configuring Direct Reader mode. The advantage of using Direct Reader V2 over Direct Reader V1 is its ability to process ORC data using vectorization, which improves performance.

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.

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

Value: DIRECT_READER_V1 or DIRECT_READER_V2

Required for using Direct Reader with the Spark Data Source API V1 or V2 (recommended), respectively.

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

Value: The JDBC endpoint for HiveServer2

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
onnector-assembly-<version>.jar \
    --conf "spark.sql.extensions=com.hortonworks.spark.sql.rule.Extensions"
 \
    --conf "spark.datasource.hive.warehouse.read.mode=DIRECT_READER_V2" \
    --conf "spark.kryo.registrator=com.qubole.spark.hiveacid.util.HiveAcid
KyroRegistrator" \
    --conf "spark.hadoop.hive.metastore.uris=<metastore_uri>" \
    --conf "spark.sql.hive.hiveserver2.jdbc.url= <jdbc_endpoint_for_hs2>"
```

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.
- This mode does not require a Hive Server (HS2) connection, therefore, the audit event is generated by HMS, which captures just the type of access (for example, SELECT) and does not capture all the details (about columns). Also, the audit event does not log the actual query (SQL).
- Does not auto-commit transactions submitted by dataframe or rdd APIs. Explicitly close transactions to release locks.

  Some of the operations, such as df.cache, df.persist, and df.rdd open transactions but do not close them. This is expected and the transactions are closed automatically after the end of the spark application. If such transactions have to be closed immediately, then it is recommended to execute the following in Spark:

  ```
  com.qubole.spark.hiveacid.transaction.HiveAcidTxnManagerObject.commitTxn
  (spark)
  ```
- 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.
- Does not support stoarge-handlers.

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

Secure access mode introduction

Use Hive Warehouse Connector (HWC) secure access mode if you want fine-grained access control (FGAC) column masking and row filtering to secure managed (ACID); or external, Hive table data that you read from Spark. If you have large workloads, low-latency requirements, and require fine-grained access control, secure access mode is recommended over the Direct Reader mode.

As an administrator, you set up Ranger FGAC, consisting of column masking and row filtering, to secure the data. You select a staging location in your cloud storage service, such as S3 or ADLS. After you configure secure access mode, HWC creates external tables on your designated staging location when users launch Spark. HWC uses CREATE TABLE AS SELECT (CTAS) to create the tables. No code refactoring is necessary. Ranger policies are applied and FGAC is enforced during the CTAS operation. Users read external tables in the secure staging location.

Requirements

- As an administrator, you set up Ranger policies.
- As a user, you launch Spark and query Hive tables in secure access mode.

Considerations

- Intermediate data is generated every time the Spark job runs even if it is running on the same underlying data.
- Intermediate data is automatically cleared when the Spark session ends.
- Additional storage requirements depend on concurrent Spark jobs running.

Related Information

Setting up secure access mode

Using secure access mode

Setting up secure access mode in Cloudera Data Hub

Learn how to set up Ranger policies on a staging location. This location is used to temporarily store Hive files that users need to read from Spark using the HWC secure access mode.

About this task

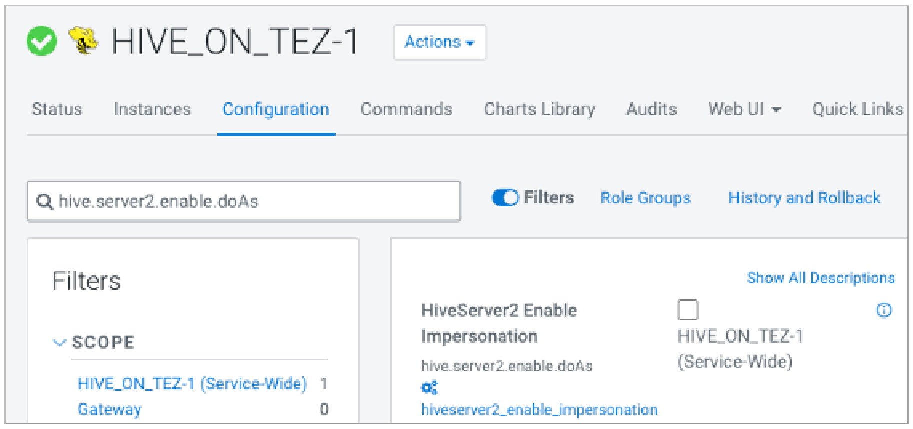
Before using secure access mode to read Hive data, you must set up two Ranger S3 access policies on the staging location for the Hive and Spark session users, and then set up a Hive URL authorization policy on the staging location.

As an example, let us consider S3 as the cloud storage service and assume the S3 staging location to be s3a://s3-hwc/stagingHWC/. In the following procedure, as an administrator, you set up Ranger policies on files and directories in the staging location. These policies secure managed ACID as well as external tables.

Procedure

1. In Cloudera Manager, click Hive on Tez Configuration , search for hive.server2.enable.doAs, and, if necessary, clear the selection to disable.

   This step is required by Hive for managing ACID tables.



hive.server2.enable.doAs

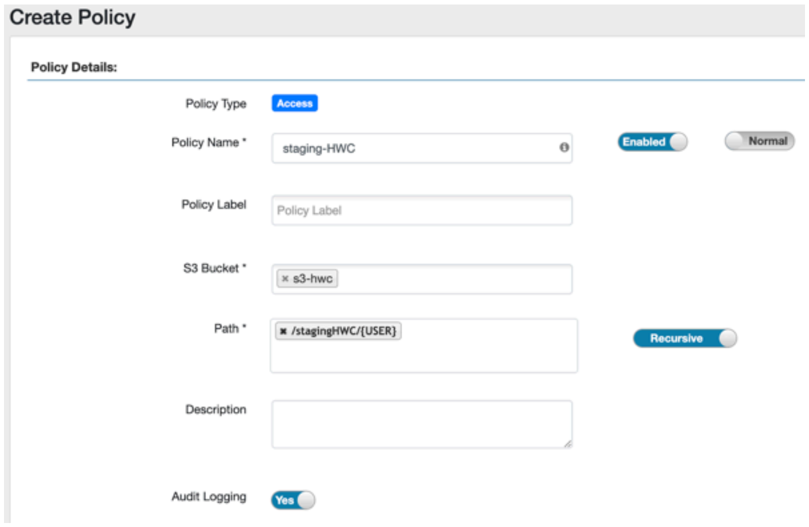
HiveServer2 Enable Impersonation ☐ HIVE_ON_TEZ-1 (Service-Wide)

hive.server2.enable.doAs

hiveserver2_enable_impersonation

2. Log into the Ranger Admin UI and in the **Service Manager** page, click the S3 preloaded resource-based service, for example, cm_s3.
3. Click Add New Policy to create an access policy for the end user who launches the spark-shell and initiates a Spark session.

4. In the **Create Policy** page, specify a policy name, S3 bucket name, and path of the staging location.

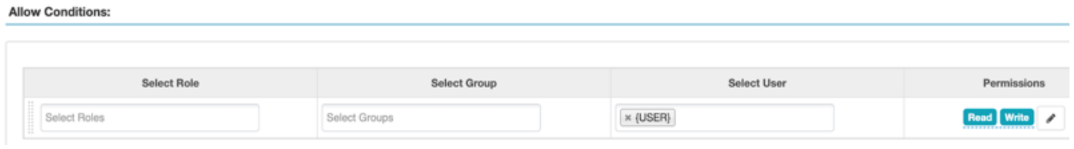


Create Policy

Policy Details:

Policy Type	Access
Policy Name *	staging-HWC (Enabled, Normal)
Policy Label	Policy Label
S3 Bucket *	s3-hwc
Path *	/stagingHWC/{USER} (Recursive)
Description	
Audit Logging	Yes

5. In the Allow Conditions section, select {USER} and add the read and write permissions, and then click Add to create the policy.



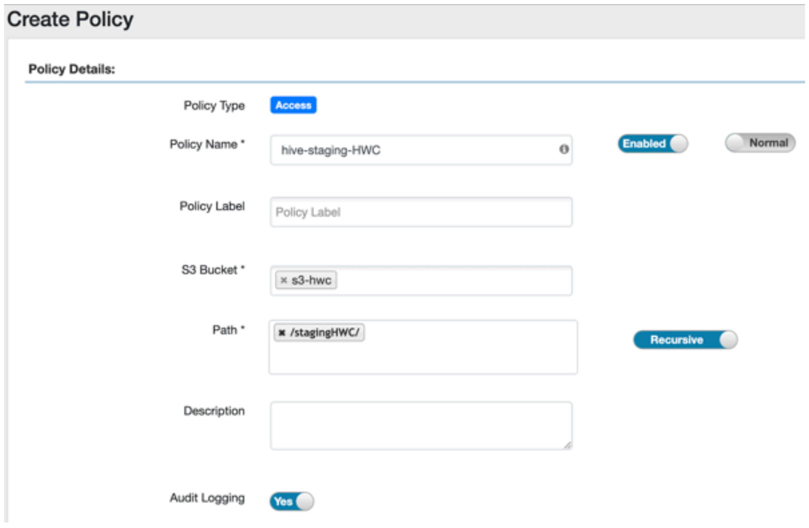
Allow Conditions:

Select Role	Select Group	Select User	Permissions
Select Roles	Select Groups	{USER}	Read Write

The Spark session users must have access to the staging location. This action sets a single Ranger policy for all users.

The policy is created and displayed in the list of available S3 policies.

6. Click Add New Policy again to add another S3 policy that grants the Hive user additional privileges on the staging path.
7. In the **Create Policy** page, specify the policy name, S3 bucket name, and path of the staging location.

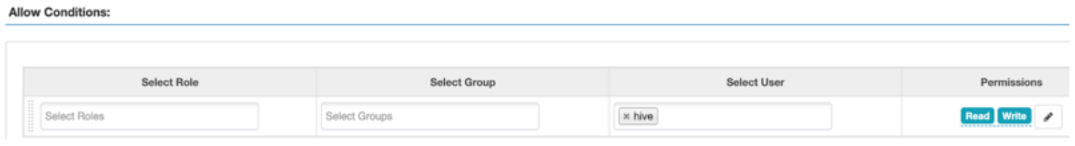


Create Policy

Policy Details:

Policy Type	Access
Policy Name *	hive-staging-HWC (Enabled, Normal)
Policy Label	Policy Label
S3 Bucket *	s3-hwc
Path *	/stagingHWC/ (Recursive)
Description	
Audit Logging	Yes

8. In the Allow Conditions section, select hive as the user and add the read and write permissions, and then click Add to create the policy.

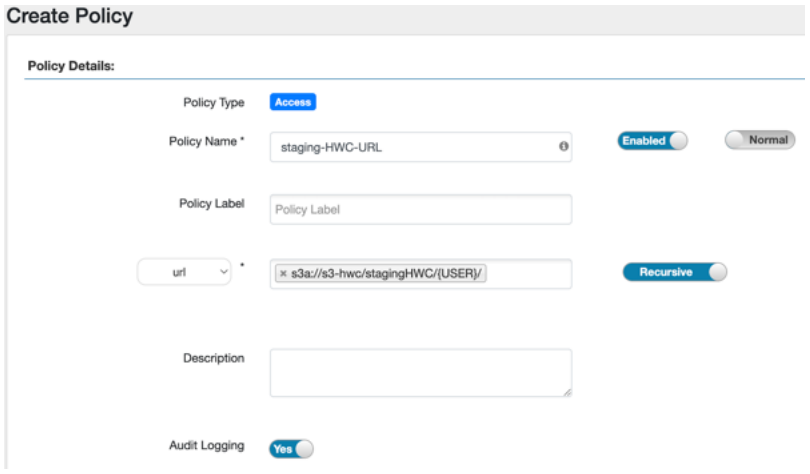


Allow Conditions:

Select Role	Select Group	Select User	Permissions
Select Roles	Select Groups	× hive	Read Write

The policy is created and displayed in the list of available S3 policies.

9. Click the Service Manager link in the breadcrumb trail and then click the Hadoop SQL preloaded resource-based service.
10. Click Add New Policy to create the Hive URL authorization policy.
11. In the **Create Policy** page, select url from the drop-down list, and specify a policy name and the S3 staging location.



Create Policy

Policy Details:

Policy Type: Access

Policy Name *: staging-HWC-URL — Enabled — Normal

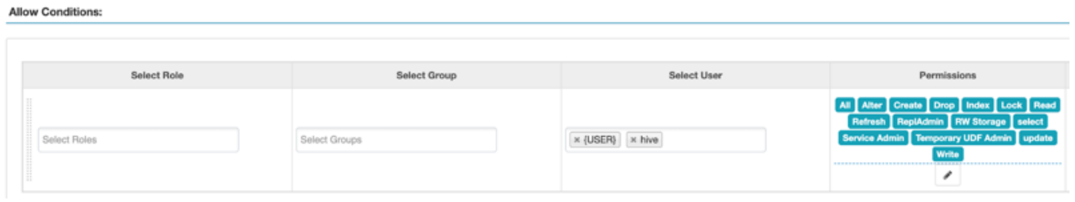
Policy Label: Policy Label

url *: × s3a://s3-hwc/stagingHWC/{USER}/ — Recursive

Description:

Audit Logging: Yes

12. In the Allow Conditions section, select {USER} and add all permissions, and then click Add to create the policy.



Allow Conditions:

Select Role	Select Group	Select User	Permissions
Select Roles	Select Groups	× {USER} × hive	All Alter Create Drop Index Lock Read Refresh ReplAdmin RW Storage select Service Admin Temporary UDF Admin update Write

The policy is created and displayed in the list of available Hadoop SQL policies.

What to do next

You can use the HWC secure access mode to securely read Hive data from Spark.

Related Information

Configure a resource-based policy: S3

Configure a resource-based policy: HadoopSQL

Using secure access mode

Learn how to use HWC secure access mode that offers fine-grained access control (FGAC) column masking and row filtering to secure managed (ACID), or external Hive table data that you read from Spark.

About this task

In this task, you enter the following configuration options when you launch the Spark shell.

- spark.datasource.hive.warehouse.load.staging.dir=<path of the staging location in your cloud storage service>, for example, s3a://s3-hwc/stagingHWC/. The path represents the temporary subdirectories per user, per session that are created under this directory. This is the temporary location where the output is staged for the duration of the session. Make sure to provide the absolute path, or fully qualified name (FQDN), not the relative path of the staging directory.

- spark.datasource.hive.warehouse.read.mode=secure_access starts using the staging output mode with fine-grained access control (FGAC). No code refactoring is required. You can use hive.sql() in your queries.

Note: Using Spark UDFs on HWC is a limitation. The workaround is to use Hive UDFs instead, as they are compatible and provide similar functionality within the Hive Warehouse Connector framework.

Before you begin

- Your administrator has set up ranger permissions for using secure access mode on Hive tables.
- Your administrator has granted you permission to the staging location in your cloud storage service, such as S3 or ADLS.

Procedure

1. Launch the Spark shell, configuring secure access mode.
For example, use the Spark session user name livy.

```
[cloudbreak@hwc-dh-master0 hwc]$ sudo -u livy spark-shell
--jars  /opt/cloudera/parcels/CDH/jars/hive-warehouse-connector-assembly-
<version>-SNAPSHOT.jar
--master yarn spark.sql.extensions="com.hortonworks.spark.sql.rule.Extens
ions"
--conf spark.kryo.registrator=com.qubole.spark.hiveacid.util.HiveAcidKyro
Registrator
--conf spark.sql.hive.hiveserver2.jdbc.url="jdbc:hive2://<domain name>:<
port>
/default;httpPath=cliservice;retries=5;serviceDiscoveryMode=zooKeeper;ssl=
true;transportMode=http;zooKeeperNamespace=hiveserver2"
--conf spark.datasource.hive.warehouse.read.mode=secure_access
--conf spark.datasource.hive.warehouse.load.staging.dir="s3a://s3-hwc/sta
gingHWC/"
--conf spark.security.credentials.hiveserver2.enabled=true
--conf spark.sql.hive.hiveserver2.jdbc.url.principal=hive/_HOST@ROOT.HW
X.SITE
```

Output is:

```
Setting default log level to "WARN".

Spark context Web UI available at http://<domain name>:<port>
Spark context available as 'sc' (master = yarn, app id = application_163
7657444149_0024).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\
        /_/

Using Scala version 2.11.12 (OpenJDK 64-Bit Server VM, Java 1.8.0_292)
Type in expressions to have them evaluated.
Type :help for more information.
```

2. Read a Hive table.

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

```
hive: com.hortonworks.spark.sql.hive.llap.HiveWarehouseSessionImpl = com.h
ortonworks.spark.sql.hive.llap.HiveWarehouseSessionImpl@159b4611

scala> hive.sql("select id, name from test.acidtable limit 3").show

+---+--------------------+
| id|                name|
+---+--------------------+
|  1|   namex x xxxx xxxx|
|  4|namex xxxx xxxxxx...|
|  1|   namex x xxxx xxxx|
+---+--------------------+
```

Example

To illustrate how fine-grained access control column masking and row filtering works, consider the following schema of a managed Hive table, test.acidtable:

```
+--------+---------+-------+
|col_name|data_type|comment|
+--------+---------+-------+
|      id|      int|       |
|    name|   string|       |
|    col3|      int|       |
|    col4|   string|       |
|    col5|      int|       |
|    col6|   string|       |
+--------+---------+-------+
```

The Spark session user, 'livy' has access to only the id and name columns and there is a masking policy on the name column.

```
scala> val hive = com.hortonworks.hwc.HiveWarehouseSession.session(spark).bu
ild()
hive: com.hortonworks.spark.sql.hive.llap.HiveWarehouseSessionImpl = com.h
ortonworks.spark.sql.hive.llap.HiveWarehouseSessionImpl@6e12d8be

scala> hive.sql("select col4, col5 from test.acidtable").show
```

Output is:

```
22/01/01 08:31:48 WARN conf.HiveConf: HiveConf of name hive.masking.algo doe
s not exist
java.lang.RuntimeException: Error while compiling statement: FAILED: HiveAcc
essControlException Permission denied: user [livy] does not have [SELECT] pr
ivilege on [test/acidtable/col4,col5]
```

The exception indicates that the user, 'livy' does not have access to columns — col4 and col5.

Related Information

Introduction to secure access mode

Setting up secure access mode

Configuring caching for secure access mode

You can enable or disable caching for the Hive Warehouse Connector (HWC) secure access mode to have finer control over read queries and ensure that the content updated outside of a Spark session is considered during reads. Caching is enabled by default because queries that run with caching enabled tend to run faster.

Before you begin

You must ensure that you are using the HWC secure access mode for reads: spark.datasource.hive.warehouse.read .mode=secure_access

Procedure

Caching is enabled by default, however, you can choose to disable caching by setting the spark.hadoop.secure.acce ss.cache.disable property to true either at a global-level, session-level, or at a runtime-level when running a query. Queries that run with caching disabled tend to run slower.

- Global-level — Specify the property in the spark-defaults.conf file.
- Session-level — Specify the property using the --conf option in spark-submit:

```
.bin/spark-submit \
--conf "spark.hadoop.secure.access.cache.disable=true"
```

- Runtime-level — Specify the property just before running your queries:

```
scala> spark.conf.set("spark.hadoop.secure.access.cache.disable","true")
scala> hive.sql("select * from anytable").show
```

Note: The value specified for the configuration at a runtime-level is valid for all subsequent queries until it is modified.

The order of preference for configuration is as follows:

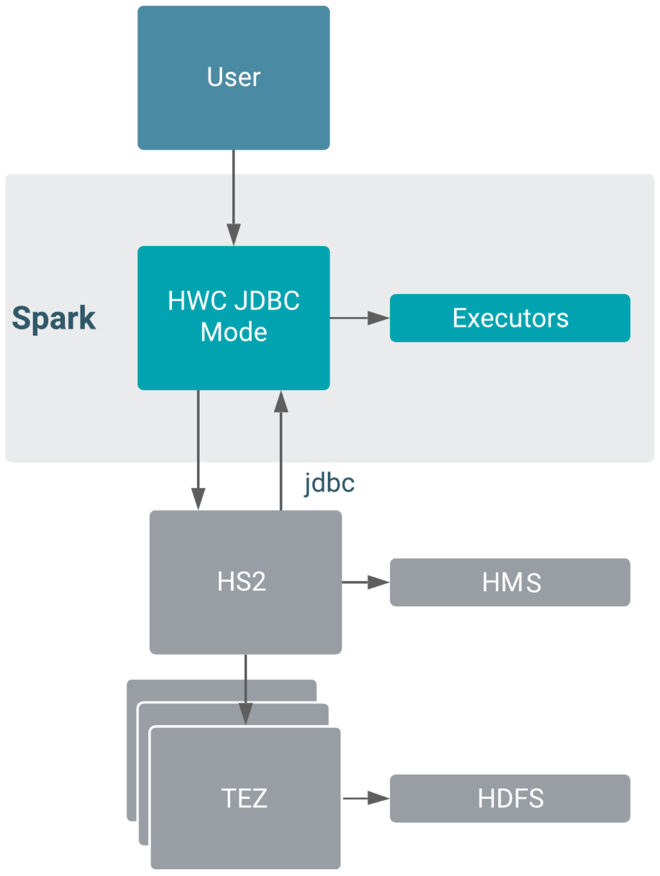
a. Property passed at a run-time level
b. Property passed at a Spark session-level
c. Property set in the spark-defaults.conf file

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 the cluster — From Spark executors, connects to Hive through JDBC and executes the query. Authorization occurs on the server and 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.sql 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 mode 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.crede ntials.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.HiveAcidKy
roRegistrator \
--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;servi
ceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2"

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

3. Read a hive table.

```
scala> hive.sql("select * from managedTable").show(1, false)
OR
scala> spark.sql("select * from managedTable").show(1, false)
```

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

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 is used for reads only, and is recommended for production workloads of 1 GB or less. With larger workload, bottlenecks develop in data transfer to Spark. Use only for running DML to extract data into an external table for direct spark.sql access.

  Writes through HWC of any size are recommended for production. Writes do not use JDBC mode.
- Data transfer is through a single JDBC connection. Does not scale and is slow for large volumes of data.
- Executors require access to HiveServer (HS2).
- 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.
- In JDBC read mode, a query of a table having a column of a complex type, such as ARRAY, STRUCT and MAP, incorrectly represents the type as String in the returned DataFrame.
- Spark fetches only the first x rows from the connection. This is to prevent HS2 timeouts which might otherwise cause Spark to hit Session/Operation Handle exceptions.

Kerberos configurations for HWC

You learn how to set up HWC for Kerberos, or not. You set properties and values depending on the JDBC_CLUSTER or JDBC_CLIENT option you configure.

In Cloudera Base on premises, you need to configure HWC options in configuration/spark-defaults.conf, depending on the read option you select. Alternatively, you can set the properties using the spark-submit/spark-shell --conf option.

Secured cluster configuration

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

On a kerberized YARN cluster, set the following property:

- JDBC_CLUSTER option in a secured cluster
  - Property: spark.security.credentials.hiveserver2.enabled
  - Value: true
  - Comment: true by default
- JDBC_CLIENT option in a secured cluster
  - Property: spark.security.credentials.hiveserver2.enabled
  - Value: false

Unsecured cluster configuration

In an unsecured cluster, set the following property:

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

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.enabl ed=false.

About this task

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 and intermediate temporary external table.

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

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.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_so
   ld_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

Apache Spark executor task statistics

Apache Spark executor task statistics

You can view Spark executor task statistics, such as the read or write metrics when you use Hive Warehouse Connector (HWC) to query Hive managed tables from Spark. These metrics enable you to view information about running Spark executors and help in troubleshooting performance issues.

For example, you can view the write metrics — .bytesWritten and .recordsWritten to understand the amount of data and number of records written to Spark. You can choose to view these metrics either at a granular task level or at an aggregated level.

For the complete list of Spark Executor Task metrics, see Apache Spark documentation.

You can view these metrics through one of the following ways:

- Using the Spark UI
- Using Spark REST APIs
- Using Spark Listeners

Using the Spark UI

You can use the Web UI that is available for each Spark context to monitor the status and resource consumption of your Spark cluster. You can navigate to the Stages tab of the Spark UI to view the current state of all stages of all jobs in the Spark application, and view the list of executor task metrics.

The Details for Stage pane displays the metrics at an overall Stage level and the Aggregated Metrics by Executor table displays metrics at an individual task level.

Using Spark REST APIs

Apart from viewing metrics through the Web UI, you can use the appropriate Spark REST APIs that returns a result set containing metrics in JSON format. You can then use the JSON to create new visualizations and monitoring tools for your Spark application.

The metrics in this JSON can be viewed either at a task level or at higher levels, such as stage or jobs. For example, the following API displays a list of all tasks for a specific stage of a YARN application:

https://<spark-ui>:port/proxy/[app-id]/api/v1/applications/[app-id]/stages/[stage-id]

The API returns a JSON that contains a summary of the statistics aggregated at a stage level and individual statistics for each task. A sample extract of the JSON is provided below:

```
"executorSummary" : {
    "driver" : {
      "taskTime" : 309,
      "failedTasks" : 0,
      "succeededTasks" : 4,
      "killedTasks" : 0,
      "inputBytes" : 0,
      "inputRecords" : 0,
      "outputBytes" : 1412,
      "outputRecords" : 16,
      "shuffleRead" : 0,
      "shuffleReadRecords" : 0,
      "shuffleWrite" : 0,
      "shuffleWriteRecords" : 0,
      "memoryBytesSpilled" : 0,
      "diskBytesSpilled" : 0,
      "isBlacklistedForStage" : false
    }
  },
```

The outputBytes and outputRecords metrics in the JSON extract correspond to the amount of data and number of records written by all tasks in a specified stage.

Using Spark Listeners

You can add Spark Listeners to a required event (completion of job or stage) to view metrics and to monitor your Spark application while the application is still running. Listeners intercept events from the Spark Scheduler to give you useful information at the end of each event.

The following code snippet shows how you can add a Spark Listener to view the bytesWritten and recordsWritten metrics at the end of a task:

```
import org.apache.spark.scheduler.{SparkListener, SparkListenerTaskEnd}
var recordsWrittenCount = 0L
var bytesWrittenCount = 0L
sc.addSparkListener(new SparkListener() {
override def onTaskEnd(taskEnd: SparkListenerTaskEnd) {
 synchronized {
recordsWrittenCount += taskEnd.taskMetrics.outputMetrics.recordsWritten
 bytesWrittenCount += taskEnd.taskMetrics.outputMetrics.bytesWritten
```

```
  }
}})
```

Related Information

Apache Spark documentation

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 built-in Spark SQL query read (only) patterns. Output conforms to built-in spark.sql conventions.

  Example

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

  Supports HiveWarehouse Session API operations using the HWC sql API. The .execute() and .executeQuery() methods are deprecated.

  Example

  ```
  scala> val hive = com.hortonworks.hwc.HiveWarehouseSession.session(spark
  ).build()
  scala> hive.sql("select * from emp_acid").show
  scala> hive.sql("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
  ```

  Note: hive.sql() is used to execute read operations and does not support write operations, such as INSERT, UPDATE, and DELETE.
- DataFrames

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

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

  Syntax

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

  Example

  ```
  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 the hive.sql() API for executing queries. You can also use the Spark SQL to query Hive managed tables, however, it is recommened that you use the HWC sql method.

- .sql()
  - Executes queries in all the read modes — Direct Reader, JDBC, and Secure access modes.
  - Consistent with the Spark sql interface.
  - Masks the internal implementation based on the cluster type you configured.
  - Used to execute read operations and does not support write operations, such as INSERT, UPDATE, and DELETE.

Results are returned as a DataFrame to Spark.

Note: The .execute() and .executeQuery() methods are deprecated and it is recommended that you use the hive.sql() method.

Support of HWC read modes on Hive tables or views

The following table represents the different Hive tables or views that are supported by the various HWC read modes:

Mode vs Table	Full ACID table	Insert-only ACID table	View (created on a managed table)	Materialized view (created on a managed table)
DIRECT_READER_V1	Yes	Yes	Yes	Yes
DIRECT_READER_V2	Yes	Yes	Yes	Yes
JDBC_CLUSTER	Yes	Yes	Yes	Yes
SECURE_ACCESS	Yes	Yes	Yes	Yes

It is recommended that you do not use Managed non-transactional tables. Such tables should ideally be converted to external tables.

Support of HWC read modes on table formats

The following table formats are supported by HWC while reading a table:

Mode	ORC	Parquet	Avro	Textfile
DIRECT_READER_V1	Yes	Yes	Yes	Yes
DIRECT_READER_V2	Yes	Yes	Yes	Yes
JDBC_CLUSTER	Yes	Yes	Yes	Yes
SECURE_ACCESS	Yes	Yes	Yes	Yes

hive.sql vs. spark.sql

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

- hive.sql() is explicitly defined in HWC and can be used across all read modes to query Apache Hive managed tables (full ACID and insert-only ACID tables).
- spark.sql() can also be used across all the read modes to query an Apache Hive managed table. However, it is recommended that you use hive.sql() over spark.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, Secure access 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.HiveWarehouseC
  onnector"). \
  mode("overwrite").option("table", "t1").save
  ```
- Automatic table creation during HWC write operation is not supported in Spark 3.

  If you are using Spark 3 and performing a DataFrame write using HWC with the SaveMode.Append or SaveMode .Overwrite modes, the write operation fails if the table you are writing to does not exist. These modes can save a DataFrame only if the table you are writing to, already exists.

  For example, the following statement tries to append to a table, "ex2" that does not exist:

  ```
  df.write.format(HIVE_WAREHOUSE_CONNECTOR).mode("append").option("table",
   "ex2").save()
  ```

  The statement results in the following exception:

  ```
  22/05/26 12:01:12 ERROR llap.HiveWarehouseConnector: Failed to connect to
   HS2
  org.apache.hive.service.cli.HiveSQLException: Error while compiling statem
  ent: FAILED: SemanticException [Error 10001]: Line 1:29 Table not found
  'ex2'
   at org.apache.hive.jdbc.Utils.verifySuccess(Utils.java:373)
   at org.apache.hive.jdbc.Utils.verifySuccessWithInfo(Utils.java:359)
   at org.apache.hive.jdbc.HiveStatement.runAsyncOnServer(HiveStatement.ja
  va:334)
  ```

```
at org.apache.hive.jdbc.HiveStatement.execute(HiveStatement.java:275)
    ...
    ...
```

Alternatively, you can use the SaveMode.ErrorIfExists or SaveMode.Ignore modes to save DataFrames to a table.

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

Spark Type	Hive Type
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

.sql is the recommended method for executing catalog operations.

- Set the current database for unqualified Hive table references

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

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

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

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

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

  hive.createDatabase(<database_name>,<ifNotExists>)
- Create an ORC table

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

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

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

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

Related Information

HMS storage

Read and write operations

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

Run 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().

Run a Hive update statement

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

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

Write a DataFrame to Hive in batch

This operation uses LOAD DATA INTO TABLE.

Java/Scala:

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

Python:

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

Write a DataFrame to Hive, specifying partitions

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

Java/Scala:

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

Python:

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

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

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

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

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

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

Write a DataFrame to Hive using HiveStreaming

When using HiveStreaming to write a DataFrame to Hive or a Spark Stream to Hive, you need to escape any commas in the stream.

Java/Scala:

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

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

Python:

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

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

Write a Spark Stream to Hive using HiveStreaming

Java/Scala:

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

Python:

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

Related Information

HMS storage

SPARK-20236

Committing a 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.commit
Txn(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
```

Closing 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(). 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().

Using 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

Hive Warehouse Connector streaming for transactional tables

Learn how to use Hive Warehouse Connector to stream data from Apache Spark into transactional Hive tables.

Running HWC Streaming

Hive Warehouse Connector supports streaming data from Apache Spark into Hive tables, enabling real-time data ingestion. However, the target Hive table must be transactional to use this feature. Use the steps below to set up and run streaming with Hive Warehouse Connector.

1. Create the Hive Table

   Before writing streaming data, pre-create the target Hive table. This table must be transactional and accessible to the Spark session. Example:

   ```
   CREATE TABLE spark_rate_source(`timestamp` STRING, value BIGINT);
   ```

2. Configure and Write Streaming Data from Spark

The following example demonstrates streaming data to the Hive table using Spark's inbuilt rate source, ideal for testing.

- Spark streaming requires a checkpoint location to manage state.

```
spark.conf.set("spark.sql.streaming.checkpointLocation", "/tmp/spark_che
ckpoint");
```

- Use Spark's rate source to generate rows at a configurable rate.

```
val rateDF = spark.readStream.format("rate").option("rowsPerSecond", 1).
load
```

- Use Hive Warehouse Connector's streaming mode (STREAM_TO_STREAM) to write data directly to the Hive table.

```
import org.apache.spark.sql.streaming.Trigger
import com.hortonworks.hwc.HiveWarehouseSession
import com.hortonworks.hwc.HiveWarehouseSession._
val query = rateDF.writeStream .format(STREAM_TO_STREAM).outputMode("app
end").option("metastoreUri", "thrift://<metastore-host>:9083").option("d
atabase", "default").option("table", "spark_rate_source").trigger(Trigge
r.ProcessingTime("1 seconds")).start
```

Key Requirements for Streaming

- Transactional Tables: The Hive table must be transactional and pre-created.
- HMS Connection: Hive Warehouse Connector connects to the Hive Metastore Service to initiate transactions, obtain write IDs, and fetch file locations.

File System Access: The Spark session must have write access to the table's file system location.

Note: The managed tables are typically restricted to the Hive user to maintain ACID v2 consistency.

Managing streaming with Hive Warehouse Connector

Understand how Hive Warehouse Connector uses HMS for transaction management and directly writes ORC files to Hive table locations without relying on HiveServer2.

Hive Warehouse Connector does not rely on HiveServer2 for streaming. Instead, it interacts with HMS for transaction management and writes ORC bucket files directly to the table's location.

An example of using the DATAFRAME_TO_STREAM method for non-streaming writes:

```
myDF.write.format(DATAFRAME_TO_STREAM)
  .option("metastoreUri", "thrift://jkovacs-1.jkovacs.root.hwx.site:9083")
  .option("metastoreKrbPrincipal", "hive/_HOST@AD.HALXG.CLOUDERA.COM")
  .option("database", "default")
  .option("table", "hwctest")
  .save()
```

Important Notes:

- Always pre-create the Hive table before writing to it.
- Ensure that the Spark session user has appropriate permissions for the table's file system location.
- Verify that the Hive Metastore URI is correctly configured in the options.

By following these steps, you can leverage Hive Warehouse Connector to efficiently stream data into Hive tables using Spark.

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.

Important: Spark 3 does not support automatic table creation using the Append and Overwrite modes. The table that you are writing to must exist in order to append or overwrite data in it. Instead, you can use the ErrorIfExists or Ignore modes.

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 (depre
cated)
//Execute Hive catalog-browsing operation and return DataFrame (deprecated)
Dataset<Row> execute(String sql);

//Execute Hive SELECT query and return DataFrame in LLAP mode (not available
 in this release)(deprecated)
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 n
ot 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);

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

//Add a table property
//Use more than once to add multiple properties
CreateTableBuilder prop(String key, String value);

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

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

MergeBuilder interface

```
package com.hortonworks.hwc;

public interface MergeBuilder {

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

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

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

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

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

//Insert rows into target table affected by merge condition and matchExpr
MergeBuilder whenNotMatchedInsert(String matchExpr, String... values);
//Execute the merge operation
void merge();
}
```

Related Information

HMS storage

Submitting 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-connect
   or-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 directo
   ry> \
   /home/myapps/myapp.jar
   ```

Related Information

Using Direct Reader mode

Using JDBC read mode

HMS storage

Examples of writing data in various file formats

Hive Warehouse Connector (HWC) enables you to write to tables in various formats, such as Parquet, ORC, AVRO, and Textfile. You see by example how to write a Dataframe in these formats, to a pre-existing or new table.

Initialize

import com.hortonworks.hwc.HiveWarehouseSession import com.hortonworks.hwc.HiveWarehouseSession._

val hive = HiveWarehouseSession.session(spark).build()

Create a Dataframe

val df = Seq((1, "bat"), (2, "mouse"), (3, "horse")).toDF("id", "name")

df.show

Table does not exist

Examples of a Dataframe write to new tables:

- Dataframe write in Parquet

  ```
  df.write.format(HIVE_WAREHOUSE_CONNECTOR).mode("append").option("table",
      "parquet_table").option("fileformat","parquet").save()
  ```
- Dataframe write in ORC

  ```
  df.write.format(HIVE_WAREHOUSE_CONNECTOR).mode("append").option("table",
      "orc_table").option("fileformat","orc").save()
  ```
- Dataframe write in Avro

  ```
  df.write.format(HIVE_WAREHOUSE_CONNECTOR).mode("append").option("table",
      "avro_table").option("fileformat","avro").save()
  ```
- Dataframe write in Textfile
  - With Default Field Delimiter (,)

    ```
    df.write.format(HIVE_WAREHOUSE_CONNECTOR).mode("append").option("table",
        "text_table1").option("fileformat","textfile").save()
    ```
  - With Custom Field Delimiter (*)

    ```
    df.write.format(HIVE_WAREHOUSE_CONNECTOR).mode("append").option("table",
        "text_table2").option("fileformat","textfile").option("sep","*").sa
    ve()
    ```

Important: Spark 3 does not support automatic table creation using the Append and Overwrite modes. The table that you are writing to must exist in order to append or overwrite data in it. Instead, you can use the ErrorIfExists or Ignore modes.

Table already exists

If you already have a table, you do not need to specify a file format, but you can as shown in the following examples:

Without file format specification

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

With file format specification

Results differ depending on whether the file format specification matches that of the table or not. If there is a mismatch in the file format, an exception is displayed.

A match succeeds

```
df.write.format(HIVE_WAREHOUSE_CONNECTOR).mode("append").option("table",
"parquet_table").option("fileformat","parquet").save()
```

Throws exception if there is a mismatch in the file format

```
df.write.format(HIVE_WAREHOUSE_CONNECTOR).mode("append").option("table",
"parquet_table").option("fileformat","orc").save()
```

Default Table Format is ORC

If you do not specify a file format for the table, the table is created in ORC format.

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

To change the default format, use the set method:

```
sql("set spark.datasource.hive.warehouse.default.write.format=parquet")
```

You can specify properties as options as follows:

.option("compression", "SNAPPY")

.option("transactional", "false")

HWC integration pyspark, sparklyr, and Zeppelin

Submitting 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
   nnector-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

Reading and writing Hive tables in R

The Hive Warehouse Connector (HWC) supports reads and writes to Apache Hive managed ACID tables in R. Cloudera provides an R package SparklyrHWC that includes all HWC methods, such as hive.sql, and a spark_write_table method to write to managed tables. The native sparklyr spark_write_table method supports writes to external tables only.

Support

HWC should work with Sparklyr 1.0.4. Versions later than 1.0.4 should also work if interfaces are not changed by sparklyr. However, sparklyr isn't supported by Cloudera. We will support any issues around using HWC from sparklyr.

Downloading SparklyrHWC

You can download the SparklyrHWC R package that includes HWC methods from your Cloudera Cluster. Go to /opt/cloudera/parcels/CDH/lib/hive_warehouse_connector/. Copy the SparklyrHWC to your download location.

About this task

To read and write Hive tables in R, you need to configure an HWC execution mode. You can configure either one of the following HWC execution modes in R:

- JDBC mode
    - Suitable for writing production workloads.
    - Suitable for reading production workloads having a data size of 1 GB or less.
    - Use this mode for reading if latency is not an issue.
- Spark-ACID mode
    - Suitable for reading production workloads.
    - Does not support writes

Reading and writing managed tables

You can read Hive managed tables using either JDBC or Spark-ACID mode. The mode you configure affects the background process. You use the same R code regardless of the mode with one exception: You do not need to call the commitTxn(hs) when using JDBC mode.

To write to Hive managed tables, you must connect to HWC in JDBC mode.

Reading and writing external tables

You can read and write Hive external tables in R using the sparklyr package. HWC is not required.

In the following procedure, you configure Spark-Acid execution mode to read tables on a production cluster. You use the native sparklyr spark_read_table and spark_load_table to read Hive managed tables in R.

Reading a Hive managed table example

Procedure

1. Read tables using spark-acid - direct_reader_v2 mode.
   For example:

   ```
   library(sparklyr)

   // Configurations needed  to use spark-acid
   config <- spark_config()
   config$sparklyr.jars.default <- "<path to hwc jar>"
   config$spark.sql.hive.hiveserver2.jdbc.url="jdbc:hive2://<url>:10000/defau
   lt"
   config$spark.datasource.hive.warehouse.user.name="hive"
   config$spark.hadoop.hive.metastore.uris="thrift://<url>:9083"
   config$spark.sql.extensions="com.hortonworks.spark.sql.rule.Extensions"
   config$spark.kryo.registrator="com.qubole.spark.hiveacid.util.HiveAcidKyro
   Registrator"
   config$spark.datasource.hive.warehouse.read.mode="DIRECT_READER_V2"

   //Set env variables

   Sys.setenv(SPARK_HOME = "/opt/cloudera/parcels/CDH/lib/spark/")
   Sys.setenv(HADOOP_HOME = "/opt/cloudera/parcels/CDH/lib/hadoop")
   // Spark context can be set to local or yarn mode
   sc <- spark_connect(master = "local",config = config)
   ```

2. Read Hive using sparklyr methods spark_read_table() or spark_load_table().
   For example:

   ```
   // reading a managed table using spark acid direct-reader

   > intDf <- sparklyr::spark_read_table(sc, 'emp_hwc')
   >  sparklyr::sdf_collect(intDf1) // Converts SparkDataframe to R dataframe
   ```

3. Explicitly commit your code. For example, use SparklyrHWC::commitTxn(hs), where hs is the HiveWarehouseSessionImpl object.

   You need to commit your code using commitTxn() due to sparklyr calling specific cache methods internally.

Writing a Hive managed table in R

About this task

You configure JDBC mode and use and SparklyrHWC::spark_write_table() to write to a managed table.

Procedure

1. In R, configure JDBC mode by setting the following configs and values:

   ```
   > config$spark.sql.extensions="com.hortonworks.spark.sql.rule.Extensions"
   > config$spark.datasource.hive.warehouse.read.via.llap="false"
   > config$spark.datasource.hive.warehouse.read.jdbc.mode="cluster"
   ```

2. Include the sparklyr library.

   ```
   > library(sparklyr)  // Load sparklyr library
   ```

3. Write to a managed table using SparklyrHWC::spark_write_table.

   ```
   > library(sparklyr)  // Load sparklyr library
   ```

```
> Sys.setenv(HADOOP_HOME = "/opt/cloudera/parcels/CDH/lib/hadoop") // set
environment variables
> Sys.setenv(SPARK_HOME = "/opt/cloudera/parcels/CDH/lib/spark/")
> sc <- spark_connect(master = "yarn", config = config) // Create a spark
 Connection.

> intDf <- sparklyr::spark_read_table(sc, 'emp_hwc') // read first table
> intDf1 <-  sparklyr::spark_read_table(sc, 'emp_overwrite') // read sec
ond table
> commitTxn(hs) // Commit transaction if read using spark-acid
> SparklyrHWC::spark_write_table('emp_hwc',intDf1,'append') // Append the
 second table, to the first.
> SparklyrHWC::spark_write_table('emp_hwc',intDf1,'overwrite') // Overwri
te the first table with the second table.
```

You do not need to call commitTxn(hs) when using JDBC mode.

Supported HWC APIs in R

About this task

The SparklyrHWC package supports the following HWC APIs in Scala.

- hive.sql() (recommended), execute(), and executeQuery() APIs to run read SQL statements

  Note: It is recommended that you use the hive.sql() method because execute() and executeQuery() methods will soon be deprecated.

- executeUpdate() API to run write SQL statements
- API call for Hive CTASK operations (create table as select .....)
- Other HWC APIs, such as dropTable, dropDatabase, showTable

Any HWC API in Scala can be used in R. The behavior of these APIs in R and HWC is similar.

Procedure

1. Create a table using another existing table.

   ```
   SparklyrHWC::executeUpdate(hs,"create table hwc1 as select * from hwc")
   ```

2. Read the new table.

   ```
   hwcDf <- SparklyrHWC::sql(hs, "select * from hwc1")
   ```

3. Convert the table to a SparkDataframe.

   ```
   hwcSdf <- sparklyr::sdf_copy_to(sc, hwcDf)
   ```

4. Write the table to a new table.

   ```
   SparklyrHWC::spark_write_table('hwcNew',hwcSdf,'OVERWRITE')
   ```

5. Append the table data to the new table.

   ```
   SparklyrHWC::spark_write_table('hwcNew',hwcSdf,'APPEND')
   ```

6. Insert data into the table using HWC API.

   ```
   SparklyrHWC::executeUpdate(hs,"insert into hwcNew values(2,'B')")
   ```

7. Read the newly created table using HWC API.

```
SparklyrHWC::sql(hs, "select * from hwcNew")
```

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 prefilx.
- 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.HiveAcidKyroRegistrator
- 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://<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

Reading and writing 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.

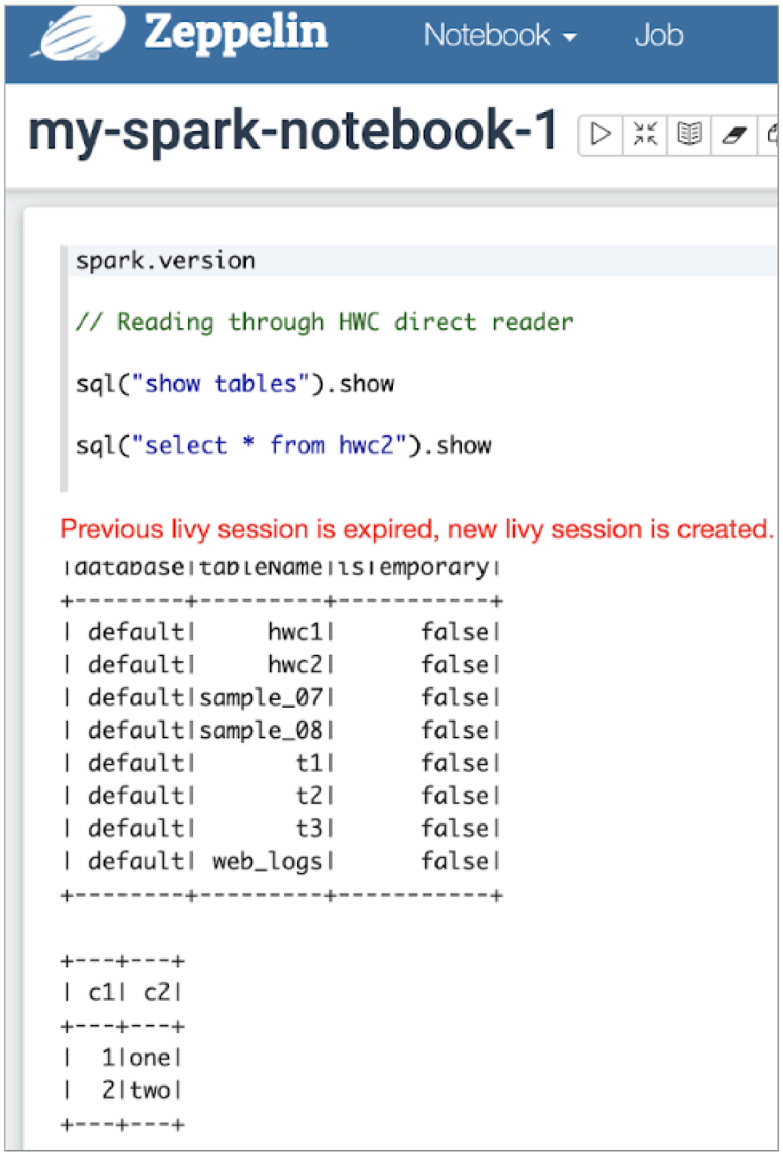
Procedure

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
```



```
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|isTemporary|
+--------+---------+-----------+
| 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.HiveWarehouseConnect
or").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

Creating 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

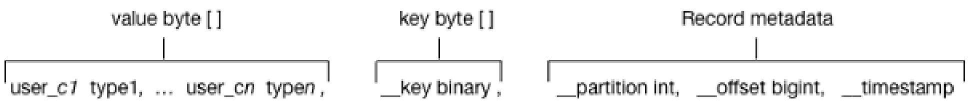
You can get useful information, including Kafka record metadata from a table of Kafka data by using typical Hive queries.

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


value byte [ ]
key byte [ ]
Record metadata
user_c1 type1, ... user_cn typen ,
__key binary ,
__partition int, __offset bigint, __timestamp

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

Querying 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 - inte
rval '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 - i
nterval '15' MINUTES) ;
```

4. Create a dimension table.

```
CREATE TABLE user_table (`user` string, `first_name` string , age int, g
ender 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_tab
le`.`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_activ
ity.`user`
```

```
AND activity.`timestamp` = future_activity.`timestamp` - interval '5' min
utes ;

--  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

Writing 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

Querying live data from Kafka

Setting consumer and producer 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 forwith 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