Cloudera Runtime 7.1.0

Integrating Apache Hive with Spark and BI

Date published: 2019-08-21
Date modified:

Legal Notice

Contents

Hive Warehouse Connector for accessing Apache Spark data

The Hive Warehouse Connector (HWC) is a Spark library/plugin that is launched with the Spark app to access any managed Hive table from Spark. You explicitly use HWC by calling the HiveWarehouseConnector API for writes. You can implicitly use HWC for reads by simply running a Spark SQL query on a managed table. Apache Ranger and the HiveWarehouseConnector library provide row and column, fine-grained access to the data.

Spark and Hive share a catalog in the Hive metastore (HMS). The shared catalog simplifies use of HWC. To read the Hive external table from Spark, you do not need to define the table redundantly in the Spark catalog. Also, HMS detects the type of client interacting with HMS, for example Hive or Spark, and compares the capabilities of the client with the Hive table requirement. A resulting HMS translation occurs that makes sense given the client and other factors.

HWC Limitations

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

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

  Example: Operation Not Supported

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

Supported applications and operations

The Hive Warehouse Connector supports the following applications:

- Spark shell
- PySpark
- The spark-submit script

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

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

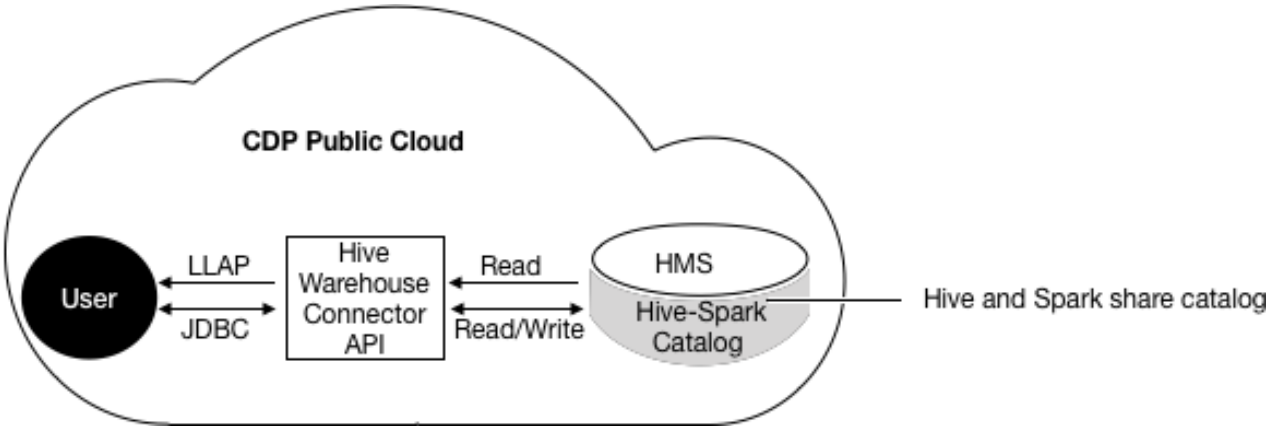
Related Information

HMS storage

Configuring HWC in CDP Public Cloud

You use the Hive Warehouse Connector (HWC) either transparently through spark sql or by using HWC API commands. You configure HWC processing to use LLAP or not, based on your use case.

To read ACID, or other Hive-managed tables, from Spark using low-latency analytical processing (LLAP) is recommended.



Low-latency analytical processing (LLAP) is recommended for reading ACID, or other Hive-managed tables, from Spark. You do not need LLAP to write to ACID, or other managed tables, from Spark. You do not need LLAP to access external tables from Spark.

Configuring the HWC mode for reads

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

- LLAP
  - true
  - false
- JDBC
  - cluster
  - client

You need to configure the following properties in configuration/spark-defaults.conf. Alternatively, you can set the properties using the spark-submit/spark-shell --conf option.

- spark.datasource.hive.warehouse.read.via.llap

  Configures LLAP mode on or off. Values: true or false
- spark.datasource.hive.warehouse.read.jdbc.mode

  Configures JDBC mode. Values: cluster or client
- spark.sql.hive.hiveserver2.jdbc.url

  The Hive JDBC url in /etc/hive/conf/beeline-site.xml.
- spark.datasource.hive.warehouse.metastoreUri

  URI of Hive metastore. In Cloudera Manager, click Clusters Hive-1 Configuration , search for hive.metastore.uris, and use that value.
- spark.datasource.hive.warehouse.load.staging.dir

  Temporary staging location required by HWC.

  Set the value to a file system location where the HWC user has write permission.

Table 1: Spark Compatibility

Tasks	Use HWC	Recommended HWC Mode
Read Hive managed tables from Spark	Yes	LLAP mode=true
Write Hive managed tables from Spark	Yes	N/A
Read Hive external tables from Spark	Ok, but unnecessary	N/A
Write Hive external tables from Spark	Ok, but unnecessary	N/A

Related Information

HMS storage

Submit an HWC Scala or Java application

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

Procedure

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

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

Related Information

HMS storage

Submit an HWC Python app

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

Procedure

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

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

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

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

Related Information

HMS storage

HWC supported types

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

Spark-Hive supported types mapping

The following types are supported by the HiveWareHouseConnector library:

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

Notes:

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

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

** Timestamp (Hive)

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

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

Spark-Hive unsupported types

Spark Type	Hive Type
CalendarIntervalType	Interval
N/A	Char
MapType	Map
N/A	Union
NullType	N/A

Spark Type	Hive Type
TimestampType	Timestamp With Timezone

Related Information

HMS storage

HiveWarehouseSession API operations

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

Import statements and variables

The following string constants are defined by the API:

- HIVE_WAREHOUSE_CONNECTOR
- DATAFRAME_TO_STREAM
- STREAM_TO_STREAM

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

- Scala

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

- Java

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

- Python

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

Executing queries

HWC supports three methods for executing queries:

- .sql()
  - Executes queries in any HWC mode.
  - Consistent with the Spark sql interface.
  - Masks the internal implementation based on cluster type.
- .execute()
  - Required for executing queries if spark.datasource.hive.warehouse.read.jdbc.mode = client (default = cluster).
  - Uses a driver side JDBC connection.
  - Provided for backward compatibility where the method defaults to reading in JDBC client mode irrespective of the value of JDBC client or cluster mode configuration.
  - Recommended for catalog queries.

- .executeQuery()
  - Executes queries, except catalog queries, in LLAP mode (spark.datasource.hive.warehouse.read.via.llap= true)
  - If LLAP is not enabled in the cluster, .executeQuery() does not work. CDP Data Center does not support LLAP.
  - Provided for backward compatibility.

Related Information

HMS storage

Catalog operations

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

Catalog operations

- Set the current database for unqualified Hive table references

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

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

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

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

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

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

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

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

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

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

Related Information

HMS storage

Read and write operations

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

Read operations

Execute a Hive SELECT query and return a DataFrame.

hive.sql("select * from web_sales")

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

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

Execute a Hive update statement

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

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

Write a DataFrame to Hive in batch

This operation uses LOAD DATA INTO TABLE.

Java/Scala:

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

Python:

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

Write a DataFrame to Hive, specifying partitions

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

Java/Scala:

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

Python:

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

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

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

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

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

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

Write a DataFrame to Hive using HiveStreaming

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

Java/Scala:

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

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

Python:

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

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

Write a Spark Stream to Hive using HiveStreaming

Java/Scala:

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

Python:

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

Related Information

HMS storage

SPARK-20236

Close HiveWarehouseSession operations

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

About this task

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

Procedure

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

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

```
. . .  //Any other operations
.close()
```

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

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

Use the Hive Warehouse Connector for streaming

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

Procedure

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

Related Information

HMS storage

Hive Warehouse Connector API Examples

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

Write a DataFrame from Spark to Hive example

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

- Append
- ErrorIfExists
- Ignore
- Overwrite

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

The following example uses Append mode.

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

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

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

ETL example (Scala)

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

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

```
hive.setDatabase("testDatabase")
hive.createTable("newTable")
.ifNotExists()
.column("ws_sold_time_sk", "bigint")
.column("ws_ship_date_sk", "bigint")
.create()
sql("SELECT ws_sold_time_sk, ws_ship_date_sk FROM web_sales WHERE ws_sold_
time_sk > 80000)
.write.format(HIVE_WAREHOUSE_CONNECTOR)
.mode("append")
.option("table", "newTable")
.save()
```

Related Information

HMS storage

Hive Warehouse Connector Interfaces

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

HiveWarehouseSession interface

```
package com.hortonworks.hwc;

public interface HiveWarehouseSession {

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

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

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

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

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

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

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

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

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

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

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

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

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

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

//Helper for merge query
MergeBuilder mergeBuilder();

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

CreateTableBuilder interface

```
package com.hortonworks.hwc;

public interface CreateTableBuilder {

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

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

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

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

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

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

MergeBuilder interface

```
package com.hortonworks.hwc;

public interface MergeBuilder {

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

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

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

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

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

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

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

Related Information

HMS storage

Apache Hive-Kafka integration

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

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

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

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

- JsonSerDe (default)
- OpenCSVSerde
- AvroSerDe

Related Information

Apache Kafka Documentation

Create a table for a Kafka stream

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

Procedure

1. Get the name of the Kafka topic you want to query to use as a table property.
   For example: "kafka.topic" = "wiki-hive-topic"

2. Construct the Kafka broker connection string.
   For example: "kafka.bootstrap.servers"="kafka.hostname.com:9092"
3. Create an external table named kafka_table by using 'org.apache.hadoop.hive.kafka.KafkaStorageHandler', as shown in the following example:

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

4. If the default JSON serializer-deserializer is incompatible with your data, choose another format in one of the following ways:
   - Alter the table to use another supported serializer-deserializer. For example, if your data is in Avro format, use the Kafka serializer-deserializer for Avro:

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

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

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

Related Information

Apache Kafka Documentation

Querying Kafka data

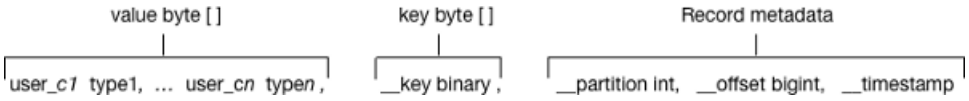
You can get useful information, including Kafka record metadata from a table of Kafka data by using typical Hive queries.

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

- Partition int32
- Offset int64
- Timestamp int64

The Hive row represents the dual composition of Kafka data:

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



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

Related Information

Apache Kafka Documentation

Query live data from Kafka

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

Before you begin

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

About this task

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

Procedure

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

```
SELECT COUNT(*) FROM kafka_table
  WHERE `__timestamp` >  1000 * to_unix_timestamp(CURRENT_TIMESTAMP - 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

Write transformed Hive data to Kafka

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

About this task

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

.

Procedure

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

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

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

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

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

Related Information

Query live data from Kafka

Set consumer and producer properties as table properties

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

Procedure

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

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

Kafka storage handler and table properties

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

Kafka storage handler

You specify 'org.apache.hadoop.hive.kafka.KafkaStorageHandler' in queries to connect to, and transform a Kafka topic into, a Hive table. In the definition of an external table, the storage handler creates a view over a single Kafka topic. For example, to use the storage handler to connect to a topic, the following table definition specifies the storage handler and required table properties: the topic name and broker connection string.

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

You set the following table properties 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 execute a query such as the following time-based lookup supported on Kafka 0.11 or later:

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

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

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

Logical operators: OR, AND

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

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

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

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

Kafka metadata

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

__key

Kafka record key (byte array)

__partition

Kafka record partition identifier (int 32)

__offset

Kafka record offset (int 64)

__timestamp

Kafka record timestamp (int 64)

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

Table Properties

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

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

Connecting Hive to BI tools using a JDBC/ODBC driver

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

About this task

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

Before you begin

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

Get the JDBC or ODBC driver

You download the Cloudera Hive JDBC or ODBC driver.

Procedure

1. Get the driver from the Cloudera Downloads page.
2. Following instructions to install the driver in documentation on the same page.

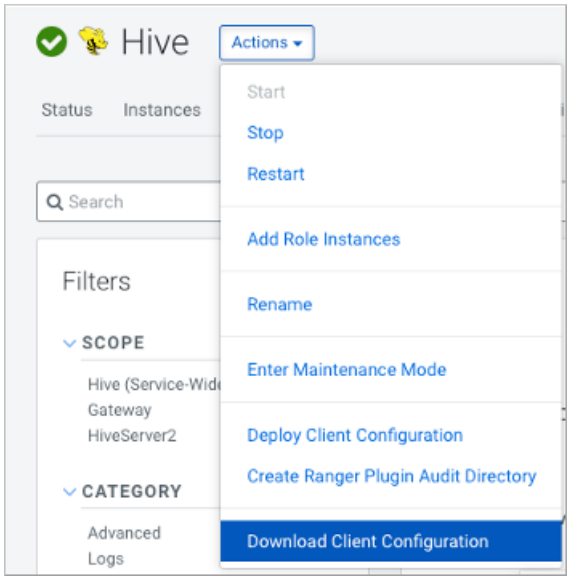
Integrate Hive and a BI tool

Before you begin

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

Procedure

1. Depending on the type of driver you obtain, proceed as follows:
   - ODBC driver: follow instructions on the ODBC driver download site, and skip the rest of the steps in this procedure.
   - JDBC driver: add the driver to the classpath of your JDBC client, such as Tableau. For example, check the client documentation about where to put the driver.
2. In Cloudera Manager (CM), click Clusters Hive click Actions, and select Download Client Configuration.



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

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

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

Specify the JDBC connection string

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

About this task

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

Procedure

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

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

   You need to specify httpPath when using the HTTP transport mode. <http_endpoint> has a corresponding HTTP endpoint configured in hive-site.xml.
3. Add parameters to the connection string for Kerberos authentication.
   jdbc:hive2://myserver:10000/default;principal=prin.dom.com@APRINCIPAL.DOM.COM

JDBC connection string syntax

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

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

Connection string parameters

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

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

JDBC Parameter	Description	Required
hiveConfs	Optional configuration parameters for Hive on the server in the following format: <key1>=<value1>;<key2>=<key2>; ... The configurations last for the duration of the user session.	no
hiveVars	Optional configuration parameters for Hive variables in the following format: <key1>=<value1>;<key2>=<key2>; ... The configurations last for the duration of the user session.	no

TCP and HTTP Transport

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

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

The syntax for using these parameters is:

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

User Authentication

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

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

The syntax for using these parameters is:

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

Transport Layer Security

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

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

The syntax for using the authentication parameters is:

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

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

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

The JDBC connection string for Sasl QOP uses these variables.

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

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

Using JdbcStorageHandler to query RDBMS

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

About this task

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

Procedure

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

```
CREATE EXTERNAL TABLE mytable_jdbc(
  col1 string,
  col2 int,
  col3 double
)
STORED BY 'org.apache.hive.storage.jdbc.JdbcStorageHandler'
TBLPROPERTIES (
```

```
  "hive.sql.database.type" = "MYSQL",
  "hive.sql.jdbc.driver" = "com.mysql.jdbc.Driver",
  "hive.sql.jdbc.url" = "jdbc:mysql://localhost/sample",
  "hive.sql.dbcp.username" = "hive",
  "hive.sql.dbcp.password" = "hive",
  "hive.sql.table" = "MYTABLE",
  "hive.sql.dbcp.maxActive" = "1"
);
```

3. Query the external table.

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