Data Access 3

Using Apache HiveQL

Date of Publish: 2018-07-12

http://docs.hortonworks.com

Contents

Apache Hive 3 tables

Using Hive, you can create managed tables or external tables.

In Hive 3, Hive has full control over managed tables. Only through Hive can you access and change the data in managed tables. Managed tables, except temporary tables, are transactional tables having ACID (atomicity, consistency, isolation, and durability) properties. Because Hive has full control of managed tables, Hive can optimize these tables extensively. If you need to bypass Hive to access data directly on the file system, you use external tables or a storage handler, such as Druid or HBase.

The following matrix lists the types of tables you can create using Hive, whether or not ACID properties are supported, required storage format, and key operations.

Table Type	ACID	File Format	Insert	Update/Delete
Managed: CRUD transactional	Yes	ORC	Yes	Yes
Managed: Insert-only transactional	Yes	Any	Yes	No
Managed: Temporary	No	Any	Yes	No
External	No	Any	Yes	Yes

The managed table storage type is Optimized Row Column (ORC) by default. If you accept the default by not specifying any storage during table creation, or if you specify ORC storage, the result is an ACID table with insert, update, and delete (CRUD) capabilities. If you specify any other storage type, such as text, CSV, AVRO, or JSON, the result is an insert-only ACID table. You cannot update or delete columns in the table.

The following table and subsequent sections cover other differences between managed (transactional) and external tables:

Table type	Security	Spark access	Optimizations
Managed (transactional)	Ranger authorization only, no SBA	Yes, using Hive Warehouse Connector	Statistics and others
External	Ranger or SBA, which requires an ACL in HDFS	Yes, direct file access	Limited

Transactional tables

Transactional (ACID) tables reside in the Hive warehouse. To achieve ACID compliance, Hive has to manage the table, including access to the table data. The data in CRUD (create, retrieve, update, and delete) tables must be in ORC file format. Insert-only tables support all file formats. Hive is designed to support a relatively low rate of transactions, as opposed to serving as an online analytical processing (OLAP) system. You can use the SHOW TRANSACTIONS command to list open and aborted transactions.

Transactional tables in Hive 3 are on a par with non-ACID tables. No bucketing or sorting is required in Hive 3 transactional tables. These tables are compatible with native cloud storage.

Hive supports one statement per transaction, which can include any number of rows, partitions, or tables.

External tables

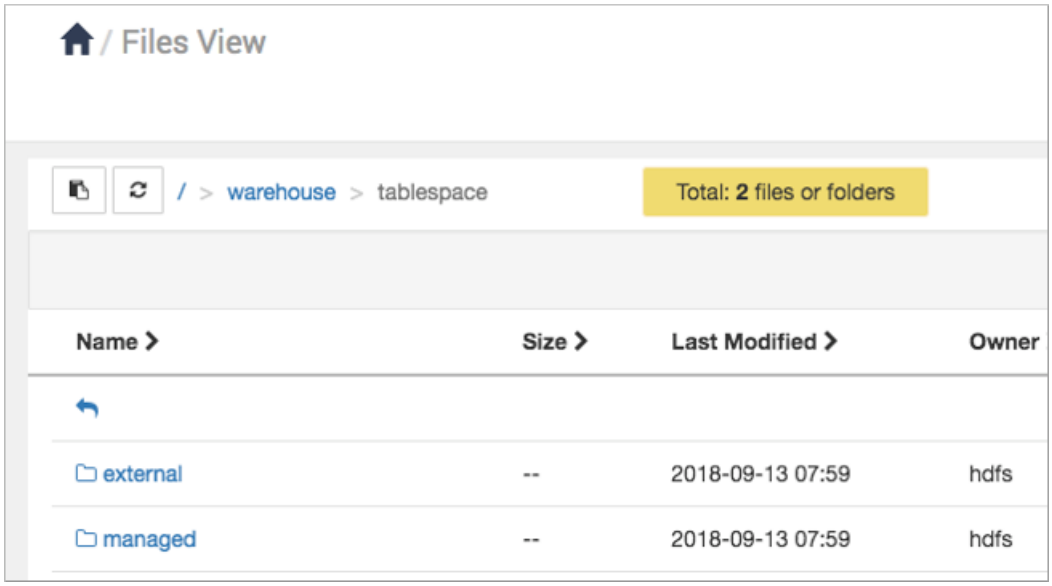
External table data is not owned or controlled by Hive. You typically use an external table when you want to access data directly at the file level, using a tool other than Hive. Hive 3 does not support the following capabilities for external tables:

- Query cache
- Materialized views, except in a limited way

- Default statistics gathering
- Compute queries using statistics
- Automatic runtime filtering
- File merging after insert

Location of tables in HDP 3.0

Managed tables reside in the managed tablespace, which only Hive can access. By default, Hive assumes external tables reside in the external tablespace. The warehouse tablespaces are shown in the Files view in Ambari:



To determine the managed or external table type, you can run the DESCRIBE EXTENDED table_name command.

Hive limitations and prerequisites

Hive is not designed to replace systems such as MySQL or HBase. If upgrading from an earlier version to Hive 3, you must run a major compaction to use your transactional tables.

Create a CRUD transactional table

You create a CRUD transactional table when you need a managed table that you can update, delete, and merge.

About this task

In this task, you create a CRUD transactional table on the command line. You cannot sort this type of table. Bucketing is optional in Hive 3 and does not affect performance. You must use STORED AS ORC in the schema definition of a CRUD transactional table as shown in the task example. Implementing a storage handler that supports AcidInputFormat and AcidOutputFormat is equivalent to specifying ORC storage used in this task.

Procedure

1. Start Hive from the Beeline shell:
   # hive
2. Enter your user name and password.
   The Hive 3 connection message, followed by the Hive prompt for entering HiveQL queries on the command line, appears.

3. Create a CRUD transactional table named T having two integer columns, a and b:

```
CREATE TABLE T(a int, b int)
  STORED AS ORC;
```

Create an insert-only transactional table

You can create a transactional table using any storage format if you do not require update and delete capability.

About this task

In this task, you create an insert-only transactional table for storing text.

Procedure

1. Start the Hive shell:
   [vagrant@c7401]# hive
2. Enter your user name and password.
   The Hive 3 connection message appears, followed by the Hive prompt for entering queries on the command line.
3. Create a insert-only transactional table named T2 having two integer columns, a and b:

```
CREATE TABLE T2(a int, b int)
 TBLPROPERTIES ('transactional'='true',
 'transactional_properties'='insert_only');
```

Create, use, and drop an external table

You use an external table, which is a table that Hive does not manage, to import data from a file on HDFS, or another file system, into Hive.

Before you begin

In this task, you need access to HDFS to put a comma-separated values (CSV) file on HDFS. If you do not use Ranger and an ACL is not in place that allows you to access HDFS, you need to log in to a node on your cluster as the hdfs user. Alternatively, when using Ranger, you need to be authorized by a policy, such as the default HDFS all-path policy (shown below) to access HDFS.

About this task

In this task, you create an external table, store the data in Hive using a managed table, and drop the external table. You create an external table and load data from a file into the table. You then use a Hive managed table to store the data in Hive. This task demonstrates the following Hive principles:

- A major difference between an external and a managed (internal) table: the persistence of table data on the files system after a DROP TABLE statement.
  - External table drop: Hive drops only the metadata, which consists mainly of the schema definition.
  - Managed table drop: Hive deletes the data and the metadata stored in the Hive warehouse.
- You can make the external table data available after dropping it by issuing another CREATE EXTERNAL TABLE statement to load the data from the file system.
- The LOCATION clause in the CREATE TABLE specifies the location of external table data.

Procedure

1. Create a text file named students.csv that contains the following lines.

```
1,jane,doe,senior,mathematics
2,john,smith,junior,engineering
```

2. As root, move the file to /home/hdfs on a node in your cluster, create a directory on HDFS in the users directory called andrena that allows access by all, and put students.csv in the directory.
   - On the command-line of a node on your cluster, enter the following commands:

```
sudo su -
mv students.csv /home/hdfs
sudo su - hdfs
hdfs dfs -mkdir /user/andrena
hdfs dfs -chmod 777 /user/andrena
hdfs dfs -put /home/hdfs/students.csv /user/andrena
hdfs dfs -chmod 777 /user/andrena/students.csv
```

   - Having authorization to HDFS through a Ranger policy, use the command line or Ambari to create the directory and put the students.csv file in the directory.
3. Start the Hive shell.
   For example: hive -n user1 -p mypassword
4. Create an external table schema definition that specifies the text format, loads data from students.csv located in /user/andrena.

```
CREATE EXTERNAL TABLE IF NOT EXISTS names_text(
  student_ID INT, FirstName STRING, LastName STRING,
  year STRING, Major STRING)
  COMMENT 'Student Names'
  ROW FORMAT DELIMITED
  FIELDS TERMINATED BY ','
  STORED AS TEXTFILE
  LOCATION '/user/andrena';
```

5. Verify that the Hive warehouse stores the student names in the external table.
   SELECT * FROM names_text;
6. Create the schema for a managed table.

```
CREATE TABLE IF NOT EXISTS Names(
  student_ID INT, FirstName STRING, LastName STRING,
  year STRING, Major STRING)
  COMMENT 'Student Names'
  STORED AS ORC;
```

7. Move the external table data to the managed table.
   INSERT OVERWRITE TABLE Names SELECT * FROM names_text;

8. Verify that the data from the external table resides in the managed table, and drop the external table, and verify that the data still resides in the managed table.

```
SELECT * from Names;
DROP TABLE names_text;
SELECT * from Names;
```

   The results from the managed table Names appears.

9. Verify that the external table schema definition is lost.
   SELECT * from names_text;

   Selecting all from names_text returns no results because the external table schema is lost. The students.csv file on HDFS containing student names data remains intact.

Determine the table type

You can determine the type of a Hive table, whether it has ACID properties, the storage format, such as ORC, and other information. Knowing the table type is important for a number of reasons, such as understanding how to store data in the table or to complete remove data from the cluster.

Procedure

1. In the Hive shell, get an extended description of the table.
   For example: DESCRIBE EXTENDED mydatabase.mytable;
2. Scroll to the bottom of the command output to see the table type.
   The following output includes that the table type is managed and transaction=true indicates that the table has ACID properties:

```
...
| Detailed Table Information  | Table(tableName:t2, dbName:mydatabase,
 owner:hdfs, createTime:1538152187, lastAccessTime:0, retention:0,
 sd:StorageDescriptor(cols:[FieldSchema(name:a, type:int, comment:null),
 FieldSchema(name:b, type:int, comment:null)], ...
```

Using constraints

You can use DEFAULT, PRIMARY KEY, FOREIGN KEY, and NOT NULL constraints in Hive ACID table definitions to improve the performance, accuracy, and reliability of data.

The Hive engine and BI tools can simplify queries if data is predictable and easily located. Hive enforces constraints as follows:

DEFAULT	Ensures a value exists, which is useful in EDW offload cases.
PRIMARY KEY	Identifies each row in a table using a unique identifier.
FOREIGN KEY	Identifies a row in another table using a unique identifier.
NOT NULL	Checks that a column value is not set to NULL.

The optimizer uses the information to make smart decisions. For example, if the engine knows that a value is a primary key, it does not look for duplicates. The following examples show the use of constraints:

```
CREATE TABLE Persons (
    ID INT NOT NULL,
    Name STRING NOT NULL,
```

```
    Age INT,
   Creator STRING DEFAULT CURRENT_USER(),
   CreateDate DATE DEFAULT CURRENT_DATE(),
   PRIMARY KEY (ID) DISABLE NOVALIDATE);

   CREATE TABLE BusinessUnit (
   ID INT NOT NULL,
   Head INT NOT NULL,
   Creator STRING DEFAULT CURRENT_USER(),
   CreateDate DATE DEFAULT CURRENT_DATE(),
   PRIMARY KEY (ID) DISABLE NOVALIDATE,
   CONSTRAINT fk FOREIGN KEY (Head) REFERENCES Persons(ID) DISABLE
NOVALIDATE
   );
```

Altering tables from flat to transactional

Knowing how Hive converts tables from flat to transactional, and being aware of the operations that are supported by the conversion, helps you transition pre-existing tables to Hive 3.

If you have a flat table (a managed, non-transactional table) that you created in release earlier than HDP 3.0, you can convert the table to transactional using an ALTER TABLE statement. Hive changes only metadata, so this operation executes very quickly. Compaction eventually rewrites the table to convert it to ACID format, but it occurs in the background, so you can run update and delete operations on the table immediately after altering it.

Wide feature parity exists between flat and transactional tables as shown in the following list of features supported in transactional tables:

- Add Partition...
- Alter Table
- Alter Table T Concatenate
- Alter Table T Rename To...
- Create Table As...
- Export/Import Table
- Fully Vectorized
- Insert Overwrite
- Into Table...
- LLAP Cache
- Load Data...
- Non-bucketed tables
- Predicate Push Down

Alter a table from flat to transactional

You might have a flat table, which is a non-transactional table in the Hive warehouse, present from earlier releases. You can use an ALTER TABLE statement to change a table from flat to transactional.

About this task

Upon completion of the task, you can immediately run update and delete operations on the table.

Procedure

1. Start the Hive shell:
   From the command line:hive
2. Enter your user name and password.

The Hive 3 connection message appears, followed by the Hive prompt for entering HiveQL queries on the command line:

```
Connected to: Apache Hive (version 3.0.0.3.0.0.0-1361)
Driver: Hive JDBC (version 3.0.0.3.0.0.0-1361)
Transaction isolation: TRANSACTION_REPEATABLE_READ
Beeline version 3.0.0.3.0.0.0-1361 by Apache Hive

0: jdbc:hive2://c7402.ambari.apache.org:2181,>
```

3. Alter the flat table to make it transactional.

```
ALTER TABLE T3 SET TBLPROPERTIES ('transactional'='true');
```

Hive 3 ACID transactions

Hive 3 achieves atomicity and isolation of operations on transactional tables by using techniques in write, read, insert, create, delete, and update operations that involve delta files, which can provide query status information and help you troubleshoot query problems.

Write and read operations

Hive 3 write and read operations improve the ACID properties and performance of transactional tables. Transactional tables perform as well as other tables. Hive supports all TPC Benchmark DS (TPC-DS) queries.

Hive 3 and later extends atomic operations from simple writes and inserts to support the following operations:

- Writing to multiple partitions
- Using multiple insert clauses in a single SELECT statement

A single statement can write to multiple partitions or multiple tables. If the operation fails, partial writes or inserts are not visible to users. Operations remain performant even if data changes often, such as one percent per hour. Hive 3 and later does not overwrite the entire partition to perform update or delete operations.

Read semantics consist of snapshot isolation. Hive logically locks in the state of the warehouse when a read operation starts. A read operation is not affected by changes that occur during the operation.

Atomicity and isolation in insert-only tables

When an insert-only transaction begins, the transaction manager gets a transaction ID. For every write, the transaction manager allocates a write ID. This ID determines a path to which data is actually written. The following code shows the schema of an insert-only transactional table:

```
CREATE TABLE tm (a int, b int) TBLPROPERTIES
        ('transactional'='true',
        'transactional_properties'='insert_only');
```

Assume that three insert operations occur, and the second one fails:

```
INSERT INTO tm VALUES(1,1);
        INSERT INTO tm VALUES(2,2); // Fails
        INSERT INTO tm VALUES(3,3);
```

For every write operation, Hive creates a delta directory to which the transaction manager writes data files. Hive writes all data to delta files, designated by write IDs, and mapped to a transaction ID that represents an atomic operation. If a failure occurs, the transaction is marked aborted, but it is atomic:

```
tm
```

```
        ___ delta_0000001_0000001_0000
        ### 000000_0
        ___ delta_0000002_0000002_0000       //Fails
        ### 000000_0
        ___ delta_0000003_0000003_0000
        ### 000000_0
```

During the read process, the transaction manager maintains the state of every transaction. When the reader starts, it asks for the snapshot information, represented by a high watermark. The watermark identifies the highest transaction ID in the system followed by a list of exceptions that represent transactions that are still running or are aborted.

The reader looks at deltas and filters out, or skips, any IDs of transactions that are aborted or still running. The reader uses this technique with any number of partitions or tables that participate in the transaction to achieve atomicity and isolation of operations on transactional tables.

Atomicity and isolation in CRUD tables

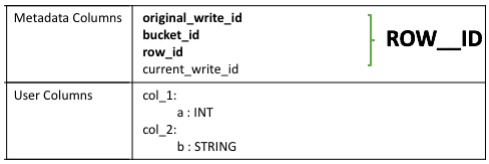
Tables that support updates and deletions require a slightly different technique to achieve atomicity and isolation. Hive runs on top of an append-only file system, which means Hive does not perform in-place updates or deletions. Isolation of readers and writers cannot occur in the presence of in-place updates or deletions. In this situation, a lock manager or some other mechanism, is required for isolation. These mechanisms create a problem for long-running queries.

Instead of in-place updates, Hive decorates every row with a row ID. The row ID is a struct that consists of the following information:

- The write ID that maps to the transaction that created the row
- The bucket ID, a bit-backed integer with several bits of information, of the physical writer that created the row
- The row ID, which numbers rows as they were written to a data file

The following example shows the schema of a full CRUD (create, retrieve, update, delete) transactional table:

```
CREATE TABLE acidtbl (a INT, b STRING)
        STORED AS ORC TBLPROPERTIES
        ('transactional'='true');
```



Instead of in-place deletions, Hive appends changes to the table when a deletion occurs. The deleted data becomes unavailable and the compaction process takes care of the garbage collection later.

Create operation

The following example inserts several rows of data into a full CRUD transactional table, creates a delta file, and adds row IDs to a data file.

```
INSERT INTO acidtbl (a,b) VALUES (100, "oranges"), (200, "apples"), (300,
 "bananas");
```

This operation generates a directory and file, delta_00001_00001/bucket_0000, that have the following data:

ROW_ID	a	b
{1,0,0}	100	"oranges"
{1,0.1}	200	"apples"
{1,0,2}	300	"bananas"

Delete operation

A delete statement that matches a single row also creates a delta file, called the delete-delta. The file stores a set of row IDs for the rows that match your query. At read time, the reader looks at this information. When it finds a delete event that matches a row, it skips the row and that row is not included in the operator pipeline. The following example deletes data from a transactional table:

```
DELETE FROM acidTbl where a = 200;
```

This operation generates a directory and file, delete_delta_00002_00002/bucket_0000 that have the following data:

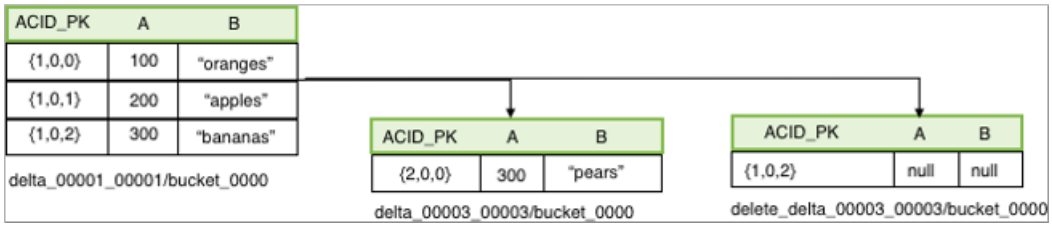
ROW_ID	a	b
{1,0,1}	null	null

Update operation

An update combines the deletion and insertion of new data. The following example updates a transactional table:

```
UPDATE acidTbl SET b = "pears" where a = 300;
```

One delta file contains the delete event, and the other, the insert event:



The reader, which requires the AcidInputFormat, applies all the insert events and encapsulates all the logic to handle delete events. A read operation first gets snapshot information from the transaction manager based on which it selects files that are relevant to that read operation. Next, the process splits each data file into the number of pieces that each process has to work on. Relevant delete events are localized to each processing task. Delete events are stored in sorted ORC file. The compressed, stored data is minimal, which is a significant advantage of Hive 3. You no longer need to worry about saturating the network with insert events in delta files.

Apache Hive Query Language basics

Using Apache Hive you can query distributed data storage including Hadoop data.

Hive supports ANSI SQL and atomic, consistent, isolated, and durable (ACID) transactions. For updating data, you can use the Hive Query Language (HiveQL) MERGE statement, which now also meets ACID standards. Materialized views optimize queries based on access patterns. Hive supports tables up to 300PB in Optimized Row Columnar (ORC) format. Other file formats are also supported. You can create tables that resemble those in a traditional relational database. You use familiar insert, update, delete, and merge SQL statements to query table data. The insert statement writes data to tables. Update and delete statements modify and delete values already written to Hive. The merge statement streamlines updates, deletes, and changes data capture operations by drawing on co-existing tables. These statements support auto-commit that treats each statement as a separate transaction and commits it after the SQL statement is executed.

Related Information

ORC Language Manual on the Apache wiki

Query the information_schema database

Hive supports the ANSI-standard information_schema database, which you can query for information about tables, views, columns, and your Hive privileges. The information_schema data reveals the state of the system, similar to sys database data, but in a user-friendly, read-only way. You can use joins, aggregates, filters, and projections in information_schema queries.

Before you begin

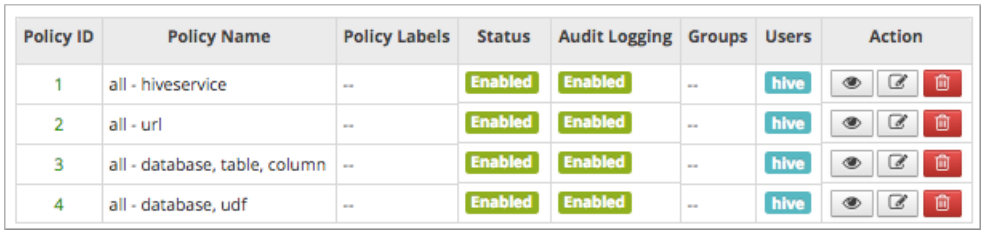
- You used Ambari to install HDP 3.0 or later.
- In Ambari, you added, configured, and started the Ranger service, which makes the information_schema database accessible and sets up an access policy for the Hive user.

About this task

One of the steps in this task involves changing the time interval for synchronization between HiveServer and the policy. HiveServer responds to any policy changes within this time interval. You can query the information_schema database for only your own privilege information.

Procedure

1. In Ambari, open the Ranger Access Manager at <node URI>:6080, and check that access policies exist for the hive user.



Policy ID	Policy Name	Policy Labels	Status	Audit Logging	Groups	Users	Action
1	all - hiveservice	--	Enabled	Enabled	--	hive	
2	all - url	--	Enabled	Enabled	--	hive	
3	all - database, table, column	--	Enabled	Enabled	--	hive	
4	all - database, udf	--	Enabled	Enabled	--	hive	

2. Navigate to **Services** > **Hive** > **Configs** > **Advanced** > **Custom hive-site**.
3. Add the hive.privilege.synchronizer.interval key and set the value to 1.

   This setting changes the synchronization from the default one-half hour to one minute.
4. From the Beeline shell, start Hive, and check that Ambari installed the information_schema database:

```
SHOW DATABASES;
...
+----------------------+
|    database_name     |
+----------------------+
| default              |
| information_schema   |
| sys                  |
+----------------------+
```

5. Use the information_schema database to list tables in the database.

```
USE information_schema;
...
SHOW TABLES;
...
+--------------------+
|      tab_name      |
+--------------------+
| column_privileges  |
| columns            |
| schemata           |
```

```
| table_privileges   |
| tables             |
| views              |
+--------------------+
```

6. Query the information_schema database to see, for example, information about tables into which you can insert values.

```
SELECT * FROM information_schema.tables WHERE is_insertable_into='YES'
 limit 2;
...
+--------------------+-------------------+-----------------
|tables.table_catalog|tables.table_schema|tables.table_name
+--------------------+-------------------+-----------------
|default             |default            |students2
|default             |default            |t3
```

Insert data into an ACID table

You can insert data into an Optimized Row Columnar (ORC) table that resides in the Hive warehouse.

About this task

You assign null values to columns you do not want to assign a value. You can specify partitioning as shown in the following syntax:

INSERT INTO TABLE tablename [PARTITION (partcol1=val1, partcol2=val2 ...)] VALUES values_row [, values_row...]

where

values_row is (value [, value]) :

Procedure

1. Create a table to contain student information.
   CREATE TABLE students (name VARCHAR(64), age INT, gpa DECIMAL(3,2)) STORED AS ORC;
2. Insert name, age, and gpa values for a few students into the table.
   INSERT INTO TABLE students VALUES ('fred flintstone', 35, 1.28), ('barney rubble', 32, 2.32);
3. Create a table called pageviews and assign null values to columns you do not want to assign a value.

```
CREATE TABLE pageviews (userid VARCHAR(64), link STRING, from STRING)
 PARTITIONED BY (datestamp STRING) CLUSTERED BY (userid) INTO 256 BUCKETS
 STORED AS ORC;
INSERT INTO TABLE pageviews PARTITION (datestamp = '2014-09-23') VALUES
 ('jsmith', 'mail.com', 'sports.com'), ('jdoe', 'mail.com', null);
INSERT INTO TABLE pageviews PARTITION (datestamp) VALUES ('tjohnson',
 'sports.com', 'finance.com', '2014-09-23'), ('tlee', 'finance.com', null,
 '2014-09-21');
```

Using materialized views

Apache Hive works with Apache Calcite to optimize your queries automatically using materialized views you create.

Using a materialized view, the optimizer can compare old and new tables, rewrite queries to accelerate processing, and manage maintenance of the materialized view when data updates occur. The optimizer can use a materialized view to fully or partially rewrite projections, filters, joins, and aggregations. Hive stores materialized views in the Hive warehouse or Druid. You can perform the following operations related to materialized views:

- Create a materialized view of queries or subqueries
- Drop a materialized view
- Show materialized views
- Describe a materialized view
- Enable or disable query rewriting based on a materialized view
- Globally enable or disable rewriting based on any materialized view

Related Information

Materialized view commands

Create and use a materialized view

You can create a materialized view of a query to calculate and store results of an expensive operation, such as join.

About this task

In this task, you create and populate example tables. You create a materialized view of a join of the tables. Subsequently, when you run a query to join the tables, the query plan takes advantage of the precomputed join to accelerate processing. This task is over-simplified and is intended to show the syntax and output of a materialized view, not to demonstrate accelerated processing that results in a real-world task, which would process a large amount of data.

Procedure

1. In the Hive shell or other Hive UI, create two tables:

```
CREATE TABLE emps (
   empid INT,
   deptno INT,
   name VARCHAR(256),
   salary FLOAT,
   hire_date TIMESTAMP)
   STORED AS ORC;

CREATE TABLE depts (
   deptno INT,
   deptname VARCHAR(256),
   locationid INT)
   STORED AS ORC;
```

2. Insert some data into the tables for example purposes:

```
INSERT INTO TABLE emps VALUES (10001,101,'jane doe',250000,'2018-01-10');
INSERT INTO TABLE emps VALUES (10002,100,'somporn
 klailee',210000,'2017-12-25');
INSERT INTO TABLE emps VALUES (10003,200,'jeiranan
 thongnopneua',175000,'2018-05-05');

INSERT INTO TABLE depts VALUES (100,'HR',10);
INSERT INTO TABLE depts VALUES (101,'Eng',11);
INSERT INTO TABLE depts VALUES (200,'Sup',20);
```

3. Create a materialized view to join the tables:

```
CREATE MATERIALIZED VIEW mv1
  AS SELECT empid, deptname, hire_date
  FROM emps JOIN depts
  ON (emps.deptno = depts.deptno)
  WHERE hire_date >= '2017-01-01';
```

4. Execute a query that takes advantage of the precomputation performed by the materialized view:

```
SELECT empid, deptname
  FROM emps
  JOIN depts
  ON (emps.deptno = depts.deptno)
  WHERE hire_date >= '2017-01-01'
  AND hire_date <= '2019-01-01';
```

Output is:

```
+--------+-----------+
| empid  | deptname  |
+--------+-----------+
| 10003  | Sup       |
| 10002  | HR        |
| 10001  | Eng       |
+--------+-----------+
```

5. Verify that the query rewrite used the materialized view by running an extended EXPLAIN statement:

```
EXPLAIN EXTENDED SELECT empid, deptname
  FROM emps
  JOIN depts
  ON (emps.deptno = depts.deptno)
  WHERE hire_date >= '2017-01-01'
  AND hire_date <= '2019-01-01';
```

The output shows the alias default.mv1 for the materialized view in the TableScan section of the plan.

```
OPTIMIZED SQL: SELECT `empid`, `deptname`
FROM `default`.`mv1`
WHERE TIMESTAMP '2019-01-01 00:00:00.000000000' >= `hire_date`
STAGE DEPENDENCIES:
  Stage-0 is a root stage


STAGE PLANS:
  Stage: Stage-0
    Fetch Operator
      limit: -1
      Processor Tree:
        TableScan
          alias: default.mv1
          filterExpr: (hire_date <= TIMESTAMP'2019-01-01
            00:00:00') (type: boolean) |
          GatherStats: false
          Filter Operator
            isSamplingPred: false
            predicate: (hire_date <= TIMESTAMP'2019-01-01
              00:00:00') (type: boolean)
            Select Operator
              expressions: empid (type: int), deptname (type:
 varchar(256))
              outputColumnNames: _col0, _col1
              ListSink
```

Related Information

Materialized view commands

Use a materialized view in a subquery

You can create a materialized view for optimizing a subquery.

About this task

In this task, you create a materialized view and use it in a subquery to return the number of destination-origin pairs. Suppose the data resides in a table named flights_hdfs that has the following data:

c_id	dest	origin
1	Chicago	Hyderabad
2	London	Moscow
...		

Procedure

1. Create a table schema definition named flights_hdfs for destination and origin data.

```
CREATE TABLE flights_hdfs(
  c_id INT,
  dest VARCHAR(256),
  origin VARCHAR(256))
  STORED AS ORC;
```

2. Create a materialized view that counts destinations and origins.

```
CREATE MATERIALIZED VIEW mv1
AS
  SELECT dest, origin, count(*)
  FROM flights_hdfs
  GROUP BY dest, origin;
```

3. Use the materialized view in a subquery to return the number of destination-origin pairs.

```
SELECT count(*)/2
FROM(
  SELECT dest, origin, count(*)
  FROM flights_hdfs
  GROUP BY dest, origin
) AS t;
```

Related Information

Materialized view commands

Drop a materialized view

You must understand when to drop a materialized view to successfully drop related tables.

About this task

Drop a materialized view before performing a DROP TABLE operation on a related table. Hive does not support dropping a table that has a relationship with a materialized view.

In this task, you drop a materialized view named mv1 from the my_database database.

Procedure

Drop a materialized view in my_database named mv1 .
DROP MATERIALIZED VIEW my_database.mv1;

Related Information

Materialized view commands

Show materialized views

You can list all materialized views in the current database or in another database.

Procedure

1. List materialized views in the current database.
   SHOW MATERIALIZED VIEWS;
2. List materialized views in a particular database.
   SHOW MATERIALIZED VIEWS IN my_database;

Related Information

Materialized view commands

Describe a materialized view

You can get summary, detailed, and formatted information about a materialized view.

About this task

This task builds on the task that creates a materialized view named mv1.

Procedure

1. Get summary information about the materialized view named mv1.

```
DESCRIBE mv1;
```

```
+------------+---------------+----------+
|  col_name  |   data_type   | comment  |
+------------+---------------+----------+
| empid      | int           |          |
| deptname   | varchar(256)  |          |
| hire_date  | timestamp     |          |
+------------+---------------+----------+
```

2. Get detailed information about the materialized view named mv1.

```
DESCRIBE EXTENDED mv1;
```

```
+----------------------------+----------------------------------...
|          col_name          |                     data_type   ...
+----------------------------+----------------------------------...
| empid                      | int                               ...
| deptname                   | varchar(256)                      ...
| hire_date                  | timestamp                         ...
|                            | NULL                              ...
| Detailed Table Information |Table(tableName:mv1, dbName:default,
 owner:hive, createTime:1532466307, lastAccessTime:0,
 retention:0, sd:StorageDescriptor(cols:[FieldSchema(name:empid,
 type:int, comment:null), FieldSchema(name:deptname,
 type:varchar(256), comment:null), FieldSchema(name:hire_date,
 type:timestamp, comment:null)], location:hdfs://krishahn-
hdp3-1.field.hortonworks.com:8020/warehouse/tablespace/managed/hive/
mv1, inputFormat:org.apache.hadoop.hive.ql.io.orc.OrcInputFormat,
 outputFormat:org.apache.hadoop.hive.ql.io.orc.OrcOutputFormat,
 compressed:false, numBuckets:-1, serdeInfo:SerDeInfo(name:null,
```

```
 serializationLib:org.apache.hadoop.hive.ql.io.orc.OrcSerde,
 parameters:{}), bucketCols:[], sortCols:[], parameters:{},
 skewedInfo:SkewedInfo(skewedColNames:[], skewedColValues:[],
 skewedColValueLocationMaps:{}), storedAsSubDirectories:false),
 partitionKeys:[], parameters:{totalSize=488, numRows=4,
 rawDataSize=520, COLUMN_STATS_ACCURATE={\"BASIC_STATS\":\"true\"},
 numFiles=1, transient_lastDdlTime=1532466307, bucketing_version=2},
 viewOriginalText:SELECT empid, deptname, hire_date\nFROM emps2
 JOIN depts\nON (emps2.deptno = depts.deptno)\nWHERE hire_date
 >= '2017-01-17', viewExpandedText:SELECT `emps2`.`empid`,
 `depts`.`deptname`, `emps2`.`hire_date`\nFROM `default`.`emps2` JOIN
 `default`.`depts`\nON (`emps2`.`deptno` = `depts`.`deptno`)\nWHERE
 `emps2`.`hire_date` >= '2017-01-17', tableType:MATERIALIZED_VIEW,
 rewriteEnabled:true, creationMetadata:CreationMetadata(catName:hive,
 dbName:default, tblName:mv1, tablesUsed:[default.depts,
 default.emps2], validTxnList:53$default.depts:2:9223372036854775807::
$default.emps2:4:9223372036854775807::,
 materializationTime:1532466307861), catName:hive, ownerType:USER)
```

3. Get formatting details about the materialized view named mv1.

```
DESCRIBE FORMATTED mv1;
```

```
+-------------------------------+--------------------------------...
|           col_name            |                     data_type  ...
+-------------------------------+--------------------------------...
| # col_name                    | data_type                      ...
| empid                         | int                            ...
| deptname                      | varchar(256)                   ...
| hire_date                     | timestamp                      ...
|                               | NULL                           ...
| # Detailed Table Information  | NULL                           ...
| Database:                     | default                        ...
| OwnerType:                    | USER                           ...
| Owner:                        | hive                           ...
| CreateTime:                   | Tue Jul 24 21:05:07 UTC 2018   ...
| LastAccessTime:               | UNKNOWN                        ...
| Retention:                    | 0                              ...
| Location:                     | hdfs://mycluster-hdp3-1.field. ...
| Table Type:                   | MATERIALIZED_VIEW              ...
| Table Parameters:             | NULL                           ...

|                               | COLUMN_STATS_ACCURATE          ...

|                               | bucketing_version              ...

|                               | numFiles                       ...

|                               | numRows                        ...

|                               | rawDataSize                    ...

|                               | totalSize                      ...

|                               | transient_lastDdlTime          ...
|                               | NULL                           ...

| # Storage Information         | NULL                           ...

| SerDe Library:                | org.apache.hadoop.hive.ql.io.or...

| InputFormat:                  | org.apache.hadoop.hive.ql.io.or...
| OutputFormat:                 | org.apache.hadoop.hive.ql.io.or...
```

```
| Compressed:                   | No                                 ...
| Num Buckets:                  | -1                                 ...
| Bucket Columns:               | []                                 ...
| Sort Columns:                 | []                                 ...
| # View Information            | NULL                               ...
| View Original Text:           | SELECT empid, deptname, hire_da...
| View Expanded Text:           | SELECT `emps2`.`empid`, `depts`...
| View Rewrite Enabled:         | Yes                                ...
```

Related Information

Materialized view commands

Manage rewriting of a query

You can use a Hive query to stop or start the optimizer from rewriting a query based on a materialized view, and as administrator, you can globally enable or disable rewriting of all queries based on materialized views.

About this task

By default, the optimizer can rewrite a query based on a materialized view. If you want a query executed without regard to a materialized view, for example to measure the execution time difference, you can disable rewriting and then enable it again.

Procedure

1. Disable rewriting of a query based on a materialized view named mv1 in the default database.

```
ALTER MATERIALIZED VIEW default.mv1 DISABLE REWRITE;
```

2. Enable rewriting of a query based on materialized view mv1.

```
ALTER MATERIALIZED VIEW default.mv1 ENABLE REWRITE;
```

3. Globally disable rewriting of queries based on materialized views by setting a global property.

```
SET hive.materializedview.rewriting=true;
```

Related Information

Materialized view commands

Create a materialized view and store it in Druid

You can create a materialized view and store it in an external system, such as Druid, which supports JSON queries, very efficient timeseries queries, and groupBy queries.

Before you begin

- Hive is running as a service in the cluster.
- Druid is running as a service in the cluster.

About this task

In this task, you include the STORED BY clause followed by the Druid storage handler. The storage handler integrates Hive and Druid for saving the materialized view in Druid.

Procedure

1. Execute a Hive query to set the location of the Druid broker using a DNS name or IP address and port 8082, the default broker text listening port.

```
SET hive.druid.broker.address.default=10.10.20.30:8082;
```

2. Create a materialized view store the view in Druid.

```
CREATE MATERIALIZED VIEW druid_mv
STORED AS 'org.apache.hadoop.hive.druid.DruidStorageHandler'
AS SELECT __time, page, user, c_added, c_removed
FROM src;
```

Related Information

Materialized view commands

Update data in a table

You use the UPDATE statement to modify data already stored in an Apache Hive table.

About this task

You construct an UPDATE statement using the following syntax:

UPDATE tablename SET column = value [, column = value ...] [WHERE expression];

Depending on the condition specified in the optional WHERE clause, an UPDATE statement might affect every row in a table. The expression in the WHERE clause must be an expression supported by a Hive SELECT clause. Subqueries are not allowed on the right side of the SET statement. Partition and bucket columns cannot be updated.

Before you begin

You must have SELECT and UPDATE privileges to use the UPDATE statement.

Procedure

Create a statement that changes the values in the name column of all rows where the gpa column has the value of 1.0.
UPDATE students SET name = null WHERE gpa <= 1.0;

Merge data in tables

You can conditionally insert, update, or delete existing data in Hive tables using the ACID MERGE statement.

About this task

The MERGE statement is based on ANSI-standard SQL.

Procedure

1. Construct a query to update the customers' names and states in customer table to match the names and states of customers having the same IDs in the new_customer_stage table.
2. Enhance the query to insert data from new_customer_stage table into the customer table if none already exists.

```
MERGE INTO customer USING (SELECT * FROM new_customer_stage) sub ON sub.id
 = customer.id
WHEN MATCHED THEN UPDATE SET name = sub.name, state = sub.new_state
WHEN NOT MATCHED THEN INSERT VALUES (sub.id, sub.name, sub.state);
```

Related Information

Merge documentation on the Apache wiki

Delete data from a table

You use the DELETE statement to delete data already written to Hive.

About this task

Use the following syntax to delete data from a Hive table. DELETE FROM tablename [WHERE expression];

Procedure

Delete any rows of data from the students table if the gpa column has a value of 1 or 0.
DELETE FROM students WHERE gpa <= 1,0;

Create a temporary table

Create a temporary table to improve performance by storing data outside HDFS for intermediate use, or reuse, by a complex query.

About this task

Temporary table data persists only during the current Apache Hive session. Hive drops the table at the end of the session. If you use the name of a permanent table to create the temporary table, the permanent table is inaccessible during the session unless you drop or rename the temporary table. You can create a temporary table having the same name as another user's temporary table because user sessions are independent. Temporary tables do not support partitioned columns and indexes.

Procedure

1. Create a temporary table having one string column.
   CREATE TEMPORARY TABLE tmp1 (tname varchar(64));
2. Create a temporary table using the CREATE TABLE AS SELECT (CTAS) statement.

   ```
   CREATE TEMPORARY TABLE tmp2 AS SELECT c2, c3, c4 FROM mytable;
   ```

3. Create a temporary table using the CREATE TEMPORARY TABLE LIKE statement.

   ```
   CREATE TEMPORARY TABLE tmp3 LIKE tmp1;
   ```

Related Information

Create/Drop/Truncate Table on the Apache wiki

Configure temporary table storage

You can change the storage of temporary table data to meet your system requirements.

About this task

By default, Apache Hive stores temporary table data in the default user scratch directory /tmp/hive-<username>. Often, this location is not set up by default to accommodate a large amount of data such as that resulting from temporary tables.

Procedure

1. Configure Hive to store temporary table data in memory or on SSD by setting hive.exec.temporary.table.storage.

- Store data in memory. hive.exec.temporary.table.storage=memory
- Store data on SSD. hive.exec.temporary.table.storage=ssd

2. Create and use temporary tables.

   Hive drops temporary tables at the end of the session.

Use a subquery

Hive supports subqueries in FROM clauses and WHERE clauses that you can use for many Hive operations, such as filtering data from one table based on contents of another table.

About this task

A subquery is a SQL expression in an inner query that returns a result set to the outer query. From the result set, the outer query is evaluated. The outer query is the main query that contains the inner subquery. A subquery in a WHERE clause includes a query predicate and predicate operator. A predicate is a condition that evaluates to a Boolean value. The predicate in a subquery must also contain a predicate operator. The predicate operator specifies the relationship tested in a predicate query.

Procedure

Select all the state and net_payments values from the transfer_payments table if the value of the year column in the table matches a year in the us_census table.

```
SELECT state, net_payments
FROM transfer_payments
WHERE transfer_payments.year IN (SELECT year FROM us_census);
```

The predicate starts with the first WHERE keyword. The predicate operator is the IN keyword.

The predicate returns true for a row in the transfer_payments table if the year value in at least one row of the us_census table matches a year value in the transfer_payments table.

Subquery restrictions

To construct queries efficiently, you must understand the restrictions of subqueries in WHERE clauses.

- Subqueries must appear on the right side of an expression.
- Nested subqueries are not supported.
- A single query can have only one subquery expression.
- Subquery predicates must appear as top-level conjuncts.
- Subqueries support four logical operators in query predicates: IN, NOT IN, EXISTS, and NOT EXISTS.
- The IN and NOT IN logical operators may select only one column in a WHERE clause subquery.
- The EXISTS and NOT EXISTS operators must have at least one correlated predicate.
- The left side of a subquery must qualify all references to table columns.
- References to columns in the parent query are allowed only in the WHERE clause of the subquery.
- Subquery predicates that reference a column in a parent query must use the equals (=) predicate operator.
- Subquery predicates may not refer only to columns in the parent query.
- Correlated subqueries with an implied GROUP BY statement may return only one row.
- All unqualified references to columns in a subquery must resolve to tables in the subquery.
- Correlated subqueries cannot contain windowing clauses.

Aggregate and group data

You use AVG, SUM, or MAX functions to aggregate data, and the GROUP BY clause to group data query results in one or more table columns..

About this task

The GROUP BY clause explicitly groups data. Hive supports implicit grouping, which occurs when aggregating the table in full.

Procedure

1. Construct a query that returns the average salary of all employees in the engineering department grouped by year.

```
SELECT year, AVG(salary)
FROM Employees
WHERE Department = 'engineering' GROUP BY year;
```

2. Construct an implicit grouping query to get the highest paid employee.

```
SELECT MAX(salary) as highest_pay,
AVG(salary) as average_pay
FROM Employees
WHERE Department = 'engineering';
```

Query correlated data

You can query one table relative to the data in another table.

About this task

A correlated query contains a query predicate with the equals (=) operator. One side of the operator must reference at least one column from the parent query and the other side must reference at least one column from the subquery. An uncorrelated query does not reference any columns in the parent query.

Procedure

Select all state and net_payments values from the transfer_payments table for years during which the value of the state column in the transfer_payments table matches the value of the state column in the us_census table.

```
SELECT state, net_payments
FROM transfer_payments
WHERE EXISTS
   (SELECT year
   FROM us_census
   WHERE transfer_payments.state = us_census.state);
```

This query is correlated because one side of the equals predicate operator in the subquery references the state column in the transfer_payments table in the parent query and the other side of the operator references the state column in the us_census table.

This statement includes a conjunct in the WHERE clause.

A conjunct is equivalent to the AND condition, while a disjunct is the equivalent of the OR condition The following subquery contains a conjunct:

... WHERE transfer_payments.year = "2018" AND us_census.state = "california"

The following subquery contains a disjunct:

... WHERE transfer_payments.year = "2018" OR us_census.state = "california"

Using common table expressions

Using common table expression (CTE), you can create a temporary view that repeatedly references a subquery.

A CTE is a set of query results obtained from a simple query specified within a WITH clause that immediately precedes a SELECT or INSERT keyword. A CTE exists only within the scope of a single SQL statement and not stored in the metastore. You can include one or more CTEs in the following SQL statements:

- SELECT
- INSERT
- CREATE TABLE AS SELECT
- CREATE VIEW AS SELECT

Recursive queries are not supported and the WITH clause is not supported within subquery blocks.

Use a CTE in a query

You can use a common table expression (CTE) to simplify creating a view or table, selecting data, or inserting data.

Procedure

1. Use a CTE to create a table based on another table that you select using the CREATE TABLE AS SELECT (CTAS) clause.

```
CREATE TABLE s2 AS WITH q1 AS (SELECT key FROM src WHERE key = '4') SELECT
 * FROM q1;
```

2. Use a CTE to create a view.

```
CREATE VIEW v1 AS WITH q1 AS (SELECT key FROM src WHERE key='5') SELECT *
 from q1;
```

3. Use a CTE to select data.

```
WITH q1 AS (SELECT key from src where key = '5')
              SELECT * from q1;
```

4. Use a CTE to insert data.
CREATE TABLE s1 LIKE src; WITH q1 AS (SELECT key, value FROM src WHERE key = '5') FROM q1
INSERT OVERWRITE TABLE s1 SELECT *;

Escape an illegal identifier

When you need to use reserved words, special characters, or a space in a column or partition name, enclose it in backticks (`).

About this task

An identifier in SQL is a sequence of alphanumeric and underscore (_) characters enclosed in backtick characters. In Hive, these identifiers are called quoted identifiers and are case-insensitive. You can use the identifier instead of a column or table partition name.

Before you begin

You have set the following parameter to column in the hive-site.xml file to enable quoted identifiers:

Set the hive.support.quoted.identifiers configuration parameter to column in the hive-site.xml file to enable quoted identifiers in column names. Valid values are none and column. For example, hive.support.quoted.identifiers = column.

Procedure

1. Create a table named test that has two columns of strings specified by quoted identifiers:
CREATE TABLE test (`x+y` String, `a?b` String);

2. Create a table that defines a partition using a quoted identifier and a region number:
   CREATE TABLE partition_date-1 (key string, value string) PARTITIONED BY (`dt+x` date, region int);
3. Create a table that defines clustering using a quoted identifier:
   CREATE TABLE bucket_test(`key?1` string, value string) CLUSTERED BY (`key?1`) into 5 buckets;

CHAR data type support

Knowing how Hive supports the CHAR data type compared to other databases is critical during migration.

Table 1: Trailing Whitespace Characters on Various Databases

Data Type	Hive	Oracle	SQL Server	MySQL	Teradata
CHAR	Ignore	Ignore	Ignore	Ignore	Ignore
VARCHAR	Compare	Compare	Configurable	Ignore	Ignore
STRING	Compare	N/A	N/A	N/A	N/A