Data Access 3

Using Apache HiveQL

Date of Publish: 2018-07-12

Contents

Apache Hive 3 tables

You can create a Hive-managed table or an external table. Hive does not fully control an external table, and you can keep the data source for this type of table outside the Hive metastore.

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.

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 SQL 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	No

Although you cannot use the SQL UPDATE or DELETE statements to delete data in some types of tables, you can use DROP PARTITION on any table type to delete the data.

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 simple authentication	Yes, using Hive Warehouse Connector	Statistics and others
External	Ranger or simple authentication	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 through 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

When you run DROP TABLE on an external table, by default Hive drops only the metadata (schema). If you want the DROP TABLE command to also remove the actual data in the external table, as DROP TABLE does on a managed table, you need to set the external.table.purge property to true as described later.

Location of tables

In Cloudera Data Platform (CDP), you specify the location of managed and external tables in the Hive warehouse:

- hive.metastore.warehouse.external.dir = s3a://bucketName/warehouse/tablespace/external/hive
- hive.metastore.warehouse.dir=s3a://bucketName/warehouse/tablespace/managed/hive

In Cloudera Manager (CM), when you launch your cluster, you accept default or specify Hive metastore variables hive.metastore.warehouse.dir and hive.metastore.warehouse.external.dir that determine storage locations for Hive tables. Managed tables reside in the managed tablespace, which only Hive can access. By default, Hive assumes external tables reside in the external tablespace.

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

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. By default, table data is stored in the Optimized Row Columnar (ORC) file format. Implementing a storage handler that supports AcidInputFormat and AcidOutputFormat is equivalent to specifying ORC storage.

Procedure

1. Launch Beeline to start Hive.
   For example:

   ```
   beeline -u jdbc:hive2://myhiveserver.com:10000 -n hive -p
   ```
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);
   ```

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. Use Data Analytics Studio, or start Hive from the command line using your user name and substituting the name or IP address of your HiveServer host as follows.
   beeline -u jdbc:hive2://myhiveserver.com:10000 -n <your user name> -p
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_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 S3, or another file system, into Hive.

Before you begin

- Obtain access to S3 to store a comma-separated values (CSV) file that will serve as the data source for the external table.
- Set up S3Guard to prevent data inconsistency caused by the S3 eventual consistency model.
- Set up Hive policies in Ranger to include S3 URLs.

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. Move the file to S3 in a bucket called andrena, and put students.csv in the directory.
3. Start the Hive shell, use Data Analytics Studio (DAS), or use another Hive UI.
   For example, substitute the URI of your HiveServer: beeline -u jdbc:hive2://myhiveserver.com:10000 -n hive -p
4. Create an external table schema definition that specifies the text format, loads data from students.csv in s3a://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 's3a://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';
   ```

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 S3 containing student names data remains intact.

Drop an external table along with data

When you run DROP TABLE on an external table, by default Hive drops only the metadata (schema). If you want the DROP TABLE command to also remove the actual data in the external table, as DROP TABLE does on a managed table, you need to configure the table properties accordingly.

Procedure

1. **Create a CSV file of data you want to query in Hive.**
2. **Launch Beeline to start Hive.**
   For example:

   ```
   beeline -u jdbc:hive2://myhiveserver.com:10000 -n hive -p
   ```

3. **Create an external table to store the CSV data, configuring the table so you can drop it along with the data.**

   ```
   CREATE EXTERNAL TABLE IF NOT EXISTS names_text(
     a INT, b STRING)
     ROW FORMAT DELIMITED
     FIELDS TERMINATED BY ','
     STORED AS TEXTFILE
     LOCATION 's3a://andrena'
     TBLPROPERTIES ('external.table.purge'='true');
   ```

4. **Run DROP TABLE on the external table.**

   ```
   DROP TABLE names_text;
   ```

The table is removed from Hive Metastore and the data stored externally. For example, names_text is removed from the Hive Metastore and the CSV file that stored the data is also deleted from HDFS.

5. Prevent data in external table from being deleted by a DROP TABLE statement.

```
ALTER TABLE addresses_text SET TBLPROPERTIES
 ('external.table.purge'='false');
```

Create an S3-based table

In Cloudera Data Platform sets up S3Guard for you when you create a table. In CDP Data Center, you need to set up S3Guard before creating an external table having a data source located on S3. This action prevents data inconsistency caused by the S3 eventual consistency model.

About this task

This task assumes you work in CDP Data Center. In this task, you create a partitioned, external table and load data from the source on S3. You can use the LOCATION clause in the CREATE TABLE to specify the location of external table data. The metadata is stored in the Hive warehouse.

Before you begin

- Set up S3Guard .
- Set up Hive policies in Ranger to include S3 URLs.

Procedure

1. Put data source files on S3.
2. Create an external table based on the data source files.

```
CREATE EXTERNAL TABLE `inventory`(
  `inv_item_sk` int,
  `inv_warehouse_sk` int,
  `inv_quantity_on_hand` int)
  PARTITIONED BY (
  `inv_date_sk` int) STORED AS ORC
  LOCATION
  's3a://BUCKET_NAME/tpcds_bin_partitioned_orc_200.db/inventory';
```

Related Information

Introducing S3Guard: S3 Consistency for Apache Hadoop

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
   );
```

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)], ...
```

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 an example of a statement that creates insert-only transactional table:

```
CREATE TABLE tm (a int, b int) TBLPROPERTIES
('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

You create a full CRUD (create, retrieve, update, delete) transactional table using the following SQL statement:

```
CREATE TABLE acidtbl (a INT, b STRING);
```

Running SHOW CREATE TABLE acidtbl provides information about the defaults: transactional (ACID) and the ORC data storage format:

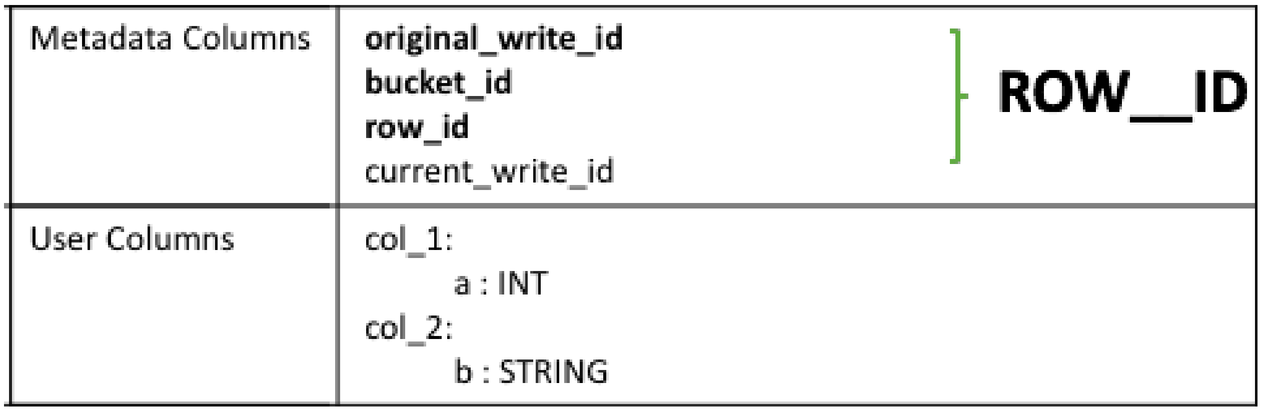
```

        +----------------------------------------------------+
        |                    createtab_stmt                  |
        +----------------------------------------------------+
        | CREATE TABLE `acidtbl`(                            |
        |   `a` int,                                         |
        |   `b` string)                                      |
        | ROW FORMAT SERDE                                   |
        |   'org.apache.hadoop.hive.ql.io.orc.OrcSerde'      |
        | STORED AS INPUTFORMAT                              |
        |   'org.apache.hadoop.hive.ql.io.orc.OrcInputFormat'  |
        | OUTPUTFORMAT                                       |
        |   'org.apache.hadoop.hive.ql.io.orc.OrcOutputFormat' |
        | LOCATION                                           |
        |   's3://myserver.com:8020/warehouse/tablespace/managed/hive/
acidtbl' |
        | TBLPROPERTIES (                                    |
        |   'bucketing_version'='2',                         |
        |   'transactional'='true',                          |
        |   'transactional_properties'='default',            |
        |   'transient_lastDdlTime'='1555090610')            |
        +----------------------------------------------------+
```

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



Metadata Columns	**original_write_id** **bucket_id** **row_id** current_write_id
User Columns	col_1: a : INT col_2: b : STRING

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:

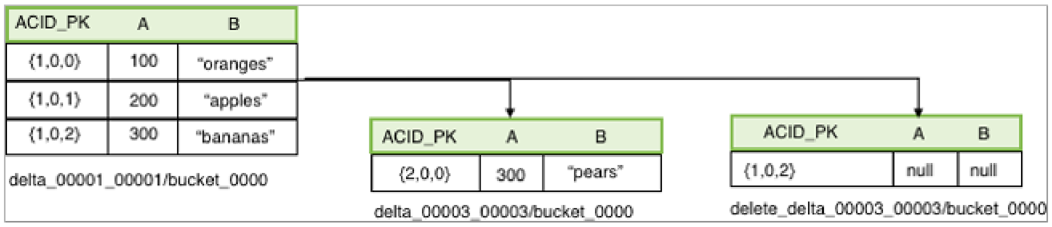
ROW_ID	a	b
{1,0,1}	null	null

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 a 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.

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
- Use partitioning to improve the performance of materialized views.

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 Data Analytics Studio (DAS), 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);

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

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));
   ```

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;

Show materialized views

You can list all materialized views in the current database or in another database. You can filter a list of materialized views in a specified database using regular expression wildcards.

About this task

You can use regular expression wildcards to filter the list of materialized views you want to see. The following wildcards are supported:

- Asterisk (*)

  Represents one or more characters.
- Pipe symbol (|)

  Represents a choice.

For example, mv_q* and *mv|q1* match the materialized view mv_q1. Finding no match does not cause an error.

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;
3. Show materialized views having names that begin with mv.
   SHOW MATERIALIZED VIEWS mv*;

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://
myserver.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://myserver...
| 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.
- You created a transactional table named src that has timestamp, dimension, and metric columns: __time TIMESTAMP, page STRING, user STRING, c_added INT, and c_removed INT columns.

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 BY 'org.apache.hadoop.hive.druid.DruidStorageHandler'
   AS SELECT __time, page, user, c_added, c_removed
   FROM src;
   ```

Related Information

Materialized view commands

Create and use a partitioned materialized view

When creating a materialized view, you can partition selected columns to improve performance. Partitioning separates the view of a table into parts, which often improves query rewrites of partition-wise joins of materialized views with tables or other materialized views.

About this task

This task assumes you created a materialized view of the emps and depts tables and assumes you created these tables. The emps table contains the following data:

empid	deptno	name	salary	hire_date
10001	101	jane doe	250000	2018-01-10
10005	100	somporn klailee	210000	2017-12-25
10006	200	jeiranan thongnopneua	175000	2018-05-05

The depts table contains the following data:

deptno	deptname	locationid
100	HR	10

deptno	deptname	locationid
101	Eng	11
200	Sup	20

In this task, you create two materialized views: one partitions data on department; the another partitions data on hire date. You select data, filtered by department,from the original table, not from either one of the materialized views. The explain plan shows that Hive rewrites your query for efficiency to select data from the materialized view that partitions data by department. In this task, you also see the effects of rebuilding a materialized view.

Procedure

1. Create a materialized view of the emps table that partitions data into departments.

```
CREATE MATERIALIZED VIEW partition_mv_1 PARTITIONED ON (deptno)
AS SELECT hire_date, deptno FROM emps WHERE deptno > 100 AND deptno < 200;
```

2. Create a second materialized view that partitions the data on the hire date instead of the department number.

```
CREATE MATERIALIZED VIEW partition_mv_2 PARTITIONED ON (hire_date)
  AS SELECT deptno, hire_date FROM emps where deptno > 100 AND deptno <
 200;
```

3. Generate an extended explain plan by selecting data for department 101 directly from the emps table without using the materialized view.

```
EXPLAIN EXTENDED SELECT deptno, hire_date FROM emps_a where deptno = 101;
```

The explain plan shows that Hive rewrites your query for efficiency, using the better of the two materialized views for the job: partition_mv_1.

```
+----------------------------------------------------+
|                       Explain                      |
+----------------------------------------------------+
| OPTIMIZED SQL: SELECT CAST(101 AS INTEGER) AS `deptno`, `hire_date` |
| FROM `default`.`partition_mv_1`                    |
| WHERE 101 = `deptno`                               |
| STAGE DEPENDENCIES:                                |
|   Stage-0 is a root stage
...
```

4. Correct Jane Doe's hire date to February 12, 2018, rebuild one of the materialized views, but not the other, and compare contents of both materialized views.

```
INSERT INTO emps VALUES (10001,101,'jane doe',250000,'2018-02-12');
ALTER MATERIALIZED VIEW partition_mv_1 REBUILD;
SELECT * FROM partition_mv_1 where deptno = 101;
SELECT * FROM partition_mv_2 where deptno = 101;
```

The output of selecting the rebuilt partition_mv_1 includes the original row and newly inserted row because INSERT does not perform in-place updates (overwrites).

```
+---------------------------+------------------------+
| partition_mv_1.hire_date  | partition_mv_1.deptno  |
+---------------------------+------------------------+
| 2018-01-10 00:00:00.0     | 101                    |
| 2018-02-12 00:00:00.0     | 101                    |
+---------------------------+------------------------+
```

The output from the other partition is stale because you did not rebuild it:

```
+------------------------+--------------------------+
| partition_mv_2.deptno  | partition_mv_2.hire_date  |
+------------------------+--------------------------+
| 101                    | 2018-01-10 00:00:00.0     |
+------------------------+--------------------------+
```

5. Create a second employees table and a materialized view of the tables joined on the department number.

```
CREATE TABLE emps2 TBLPROPERTIES AS SELECT * FROM emps;

CREATE MATERIALIZED VIEW partition_mv_3 PARTITIONED ON (deptno) AS
  SELECT emps.hire_date, emps.deptno FROM emps, emps2
  WHERE emps.deptno = emps2.deptno
  AND emps.deptno > 100 AND emps.deptno < 200;
```

6. Generate an explain plan that joins tables emps and emps2 on department number using a query that omits the partitioned materialized view.

```
EXPLAIN EXTENDED SELECT emps.hire_date, emps.deptno FROM emps, emps2
  WHERE emps.deptno = emps2.deptno
  AND emps.deptno > 100 AND emps.deptno < 200;
```

The output shows that Hive rewrites the query to use the partitioned materialized view partition_mv_3 even though your query omitted the materialized view.

7. Verify that the partition_mv_3 sets up the partition for deptno=101 for partition_mv_3.

```
SHOW PARTITIONS partition_mv_3;
```

Output is:

```
+-------------+
|  partition  |
+-------------+
| deptno=101  |
+-------------+
```

Related Information

Create and use a materialized view

Materialized view commands

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

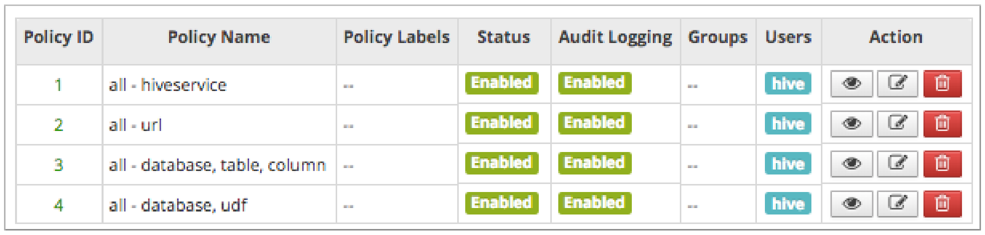
Hive supports the ANSI-standard information_schema database, which you can query for information about tables, views, columns, and your Hive privileges. The information_schema data reveals the state of the system, similar to sys database data, but in a user-friendly, read-only way. You can use joins,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

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. 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));
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;
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');
```

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 an ACID table.

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;

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

Create partitions dynamically

You can configure Hive to create partitions dynamically and then run a query that creates the related directories on the file system, such as S3. Hive then separates the data into the directories.

About this task

This example assumes you have the following CSV file named employees.csv to use as the data source:

```
1,jane doe,engineer,service
2,john smith,sales rep,sales
3,naoko murai,service rep,service
4,somporn thong,ceo,sales
5,xi singh,cfo,finance
```

Procedure

1. Upload the CSV file to a file system, for example S3.
2. Use Data Analytics Studio (DAS) or launch Beeline, and in the Hive shell, create an unpartitioned table that holds all the data.

```
CREATE EXTERNAL TABLE employees (eid int, name string, position string,
 dept string)
  ROW FORMAT DELIMITED
  FIELDS TERMINATED BY ','
  STORED AS TEXTFILE
  LOCATION 's3://user/hive/dataload/employee';
```

3. Check that the data loaded into the employees table.

```
SELECT * FROM employees;
```

The output, formatted to fit this publication, appears:

```

+------+---------------+-------------+-------+---------+
| eid  |    name       | position    | dept  |         |
+------+---------------+-------------+-------+---------|
| 1    | jane doe      | engineer    | service         |
| 2    | john smith    | sales rep   | sales           |
| 3    | naoko murai   | service rep | service         |
| 4    | somporn thong | ceo         | sales           |
| 5    | xi singh      | cfo         | finance         |
+------+---------------+-------------+-----------------+
```

4. Create a partition table.

```
CREATE EXTERNAL TABLE EMP_PART (eid int, name string, position string)
```

```
  PARTITIONED BY (dept string);
```

5. Set the dynamic partition mode to create partitioned directories of data dynamically when data is inserted.

```
SET hive.exec.dynamic.partition.mode=nonstrict;
```

6. Insert data from the unpartitioned table (all the data) into the partitioned table , dynamically creating the partitions.

```
INSERT INTO TABLE EMP_PART PARTITION (DEPT)
  SELECT eid,name,position,dept FROM employees;
```

   Partitions are created dynamically.

7. Check that the partitions were created.

```
SHOW PARTITIONS emp_part;
```

```
+----------------+
|   partition    |
+----------------+
| dept=finance   |
| dept=sales     |
| dept=service   |
+----------------+
```

Repair partitions using MSCK repair

The MSCK REPAIR TABLE command was designed to manually add partitions that are added to or removed from the file system, such S3, but are not present in the Hive metastore.

About this task

This task assumes you created a partitioned external table named emp_part that stores partitions outside the warehouse. You remove one of the partition directories on the file system. This action renders the metastore inconsistent with the file system. You repair the discrepancy manually to synchronize the metastore with the file system, HDFS for example.

Procedure

1. Remove the dept=sales object from S3.
2. In Data Analytics Studio (DAS) or in the Hive shell, look at the emp_part table partitions.

```
SHOW PARTITIONS emp_part;
```

   The list of partitions is stale; it still includes the dept=sales directory.

```
+----------------+
|   partition    |
+----------------+
| dept=finance   |
| dept=sales     |
| dept=service   |
+----------------+
```

3. Repair the partition manually.

```
MSCK REPAIR TABLE emp_part DROP PARTITIONS;
```

Manage partitions automatically

You can discover partition changes and synchronize Hive metadata automatically. Performing synchronization automatically as opposed to manually can save substantial time, especially when partitioned data, such as logs, changes frequently. You can also configure how long to retain partition data and metadata.

About this task

After creating a partitioned table, Hive does not update metadata about corresponding objects or directories on the file system that you add or drop. The partition metadata in the Hive metastore becomes stale after corresponding objects/ directories are added or deleted. You need to synchronize the metastore and the file system.

You can refresh Hive metastore partition information manually or automatically.

- Manually

  You run the MSCK (metastore consistency check) Hive command: MSCK REPAIR TABLE table_name SYNC PARTITIONS every time you need to synchronize a partition with the file system.
- Automatically

  You set up partition discovery to occur periodically.

The discover.partitions table property is automatically created and enabled for external partitioned tables. When discover.partitions is enabled for a table, Hive performs an automatic refresh as follows:

- Adds corresponding partitions that are in the file system, but not in metastore, to the metastore.
- Removes partition schema information from metastore if you removed the corresponding partitions from the file system.

Partition retention

You can configure how long to keep partition metadata and data, and remove it after the retention period elapses.

Limitations

Generally, partition discovery and retention is not recommended for use on managed tables. The Hive metastore acquires an exclusive lock on a table that enables partition discovery that can slow down other queries.

Related Information

Create partitions dynamically

Apache Wiki: Discover Partitions and Partition Retention

Automate partition discovery and repair

Hive automatically and periodically discovers discrepancies in partition metadata in the Hive metastore and corresponding directories or objects on the file system, and then performs synchronization. This operation is often automated for processing log data or data in Spark and Hive catalogs.

About this task

The discover.partitions table property enables and disables synchronization of the file system with partitions. In external partitioned tables, this property is enabled (true) by default when you create the table. To a legacy external table (created using a version of Hive that does not support this feature), you need to add discover.partitions to the table properties to enable partition discovery.

By default, the discovery and synchronization of partitions occurs every 5 minutes, but you can configure the frequency as shown in this task.

Procedure

1. Assuming you have an external table created using a version of Hive that does not support partition discovery, enable partition discovery for the table.

   ```
   ALTER TABLE exttbl SET TBLPROPERTIES ('discover.partitions' = 'true');
   ```

2. Set synchronization of partitions to occur every 10 minutes expressed in seconds: Set metastore.partition.management.task.frequency to 600.

Related Information

Create partitions dynamically

Apache Wiki: Discover Partitions and Partition Retention

Manage partition retention time

You can keep the size of the Hive metadata and data you accumulate for log processing, and other activities, to a manageable size by setting a retention period for the data.

Before you begin

The table must be configured to automatically synchronize partition metadata with directories or objects on a file system, such as S3.

About this task

If you specify a partition metadata retention period, Hive drops the metadata and corresponding data in any partition created after the retention period. You express the retention time using a numeral and the following character or characters:

- ms (milliseconds)
- s (seconds)
- m (minutes)
- d (days)

In this task, you configure automatic synchronization of the file system partitions with the metastore and a partition retention period. Assume you already created a partitioned, external table named employees as described earlier (see link below).

Procedure

1. If necessary, enable automatic discovery of partitions for the table employees.

   ```
   ALTER TABLE employees SET TBLPROPERTIES ('discover.partitions'='true');
   ```

   By default, external partitioned tables already set this table property to true.

2. Configure a partition retention period of one week.

   ```
   ALTER TABLE employees SET TBLPROPERTIES
    ('partition.retention.period'='7d');
   ```

   The partition metadata as well as the actual data for employees in Hive is automatically dropped after a week.

Related Information

Create partitions dynamically

Apache Wiki: Discover Partitions and Partition Retention

Generate surrogate keys

You can use the built-in SURROGATE_KEY user-defined function (UDF) to automatically generate numerical Ids for rows as you enter data into a table. The generated surrogate keys can replace wide, multiple composite keys.

Before you begin

Hive supports the surrogate keys on ACID tables only, as described in the following matrix of table types:

Table Type	ACID	Surrogate Keys	File Format	INSERT	UPDATE/DELETE
Managed: CRUD transactional	Yes	Yes	ORC	Yes	Yes
Managed: Insert-only transactional	Yes	Yes	Any	Yes	No
Managed: Temporary	No	No	Any	Yes	No
External	No	No	Any	Yes	No

The table you want to join using surrogate keys cannot have column types that need casting. These data types must be primitives, such as INT or STRING.

About this task

Joins using the generated keys are faster than joins using strings. Using generated keys does not force data into a single node by a row number. You can generate keys as abstractions of natural keys. Surrogate keys have an advantage over UUIDs, which are slower and probabilistic.

The SURROGATE_KEY UDF generates a unique Id for every row that you insert into a table. It generates keys based on the execution environment in a distributed system, which includes a number of factors, such as internal data structures, the state of a table, and the last transaction id. Surrogate key generation does not require any coordination between compute tasks.

The UDF takes either no arguments or two arguments:

- Write Id bits
- Task Id bits

Procedure

1. Create a students table in the default ORC format that has ACID properties.

```
CREATE TABLE students (row_id INT, name VARCHAR(64), dorm INT);
```

2. Insert data into the table. For example:

```
INSERT INTO TABLE students VALUES (1, 'fred flintstone', 100), (2, 'barney
 rubble', 200);
```

3. Create a version of the students table using the SURROGATE_KEY UDF.

```
CREATE TABLE students_v2
(`ID` BIGINT DEFAULT SURROGATE_KEY(),
 row_id INT,
 name VARCHAR(64),
 dorm INT,
 PRIMARY KEY (ID) DISABLE NOVALIDATE);
```

4. Insert data, which automatically generates surrogate keys for the primary keys.

```
INSERT INTO students_v2 (row_id, name, dorm) SELECT * FROM students;
```

5. Take a look at the surrogate keys.

```
SELECT * FROM students_v2;
```

```
+-----------------+---------------------+-------------------
+--------------------+
| students_v2.id  | students_v2.row_id  | students_v2.name  |
 students_v2.dorm  |
+-----------------+---------------------+-------------------
+--------------------+
| 1099511627776   | 1                   | fred flintstone   | 100
      |
| 1099511627777   | 2                   | barney rubble     | 200
      |
+-----------------+---------------------+-------------------
+--------------------+
```

6. Add the surrogate keys as a foreign key to another table, such as a student_grades table, to speed up subsequent joins of the tables.

```
ALTER TABLE student_grades ADD COLUMNS (gen_id BIGINT);

MERGE INTO student_grades g USING students_v2 s ON g.row_id = s.row_id
WHEN MATCHED THEN UPDATE SET gen_id = s.id;
```

7. Perform fast joins on the surrogate keys.

Query a SQL data source using the JdbcStorageHandler

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

About this task

In this task you create an external table that uses the JdbcStorageHandler to connect to and read a local JDBC data source.

Procedure

1. Load data into a supported SQL database, such as MySQL, on a node in your cluster, or familiarize yourself with existing data in the your database.

2. Create an external table using the JdbcStorageHandler and table properties that specify the minimum information: database type, driver, database connection string, user name and password for querying hive, table name, and number of active connections to Hive.

```
CREATE EXTERNAL TABLE mytable_jdbc(
  col1 string,
  col2 int,
  col3 double
)
STORED BY 'org.apache.hive.storage.jdbc.JdbcStorageHandler'
TBLPROPERTIES (
  "hive.sql.database.type" = "MYSQL",
  "hive.sql.jdbc.driver" = "com.mysql.jdbc.Driver",
  "hive.sql.jdbc.url" = "jdbc:mysql://localhost/sample",
  "hive.sql.dbcp.username" = "hive",
  "hive.sql.dbcp.password" = "hive",
  "hive.sql.table" = "MYTABLE",
  "hive.sql.dbcp.maxActive" = "1"
);
```

3. Query the external table.

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

Using functions

You can call a built-in Hive function to perform one of a wide-range of corresponding multi-step operations. You use SHOW FUNCTIONS to search for or list available functions. You can create a user-defined function (UDF) when a built-in is not available to do what you need. You might need to reload functions to update the availability of functions created in another session.

Reload, view, and filter functions

To determine which Hive functions and operators are available, you reload functions, and then use the SHOW FUNCTIONS statement. An optional pattern in the statement filters the list of functions returned by the statement.

About this task

In this task, you first reload functions to make available any user-defined functions that were registered in Hive session that started later than your session. The syntax is:

```
RELOAD (FUNCTION|FUNCTIONS);
```

Next, you use the SHOW FUNCTIONS statement. The syntax of this statement is:

```
SHOW FUNCTIONS [LIKE "<pattern>"];
```

<pattern> represents search characters that can include regular expression wildcards.

Finally, you get more information about use by issuing the DESCRIBE FUNCTION statement.

Procedure

1. Open DAS or the Hive shell.
2. Reload functions to ensure all registered UDFs are available in your session.

   ```
   RELOAD FUNCTIONS;
   ```

   Use the plural form of the command. RELOAD FUNCTION is for backward compatibility.
3. Generate a list of available built-in and user-defined functions (UDFs).

   ```
   SHOW FUNCTIONS;
   ```

   The list of built-in functions, operators, and UDFs appear.

   ```
   +------------------------------+
   |           tab_name           |
   +------------------------------+
   ```

```
| !                              |
| !=                             |
| $sum0                          |
| %                              |
...
```

4. Generate a filtered list of functions using the regular expression wildcard %.

   ```
   SHOW FUNCTIONS LIKE "a%";
   ```

   All available functions that begin with the character a appear.

   ```
   +-------------------------------+
   |           tab_name            |
   +-------------------------------+
   | abs                           |
   | acos                          |
   | add_months                    |
   ...
   ```

5. Get more information about a particular function.

   ```
   DESCRIBE FUNCTION abs;
   ```

   ```
   +-------------------------------------------+
   |                 tab_name                  |
   +-------------------------------------------+
   | ABS(x) - returns the absolute value of x  |
   +-------------------------------------------+
   ```

6. Get more information about the function.

   ```
   DESCRIBE FUNCTION EXTENDED abs;
   ```

   ```
   +----------------------------------------------------+
   |                      tab_name                      |
   +----------------------------------------------------+
   | ABS(x) - returns the absolute value of x           |
   | Synonyms: abs                                      |
   | Example:                                           |
   |   > SELECT ABS(0) FROM src LIMIT 1;                |
   |   0                                                |
   |   > SELECT ABS(-5) FROM src LIMIT 1;               |
   |   5                                                |
   | Function class:org.apache.hadoop.hive.ql.udf.generic.GenericUDFAbs |
   | Function type:BUILTIN                              |
   +----------------------------------------------------+
   ```

Create a user-defined function

You export user-defined functionality (UDF) to a JAR from a Hadoop- and Hive-compatible Java project and store the JAR on your cluster. Using Hive commands, you register the UDF based on the JAR, and call the UDF from a Hive query.

Before you begin

- You have access rights to upload the JAR to TBD.
- HiveServer or Hive Interactive Server, or both, are running on the cluster.

- You have installed Java and a Java integrated development environment (IDE) tool on the machine where you will create the UDF.

Set up the development environment

You can create a Hive UDF in a development environment using IntelliJ, for example, and build the UDF with Hive and Hadoop JARS that you download from your Cloudera cluster.

Procedure

1. On your cluster, locate the hadoop-common-<version>.jar and hive-exec-<version>.jar.
2. Download the JARs to your development computer to add to your IntelliJ project later.
3. Open IntelliJ and create a new Maven-based project. Click Create New Project. Select Maven and the supported Java version as the Project SDK. Click Next.
4. Add archetype information.
   For example:
   - GroupId: com.mycompany.hiveudf
   - ArtifactId: hiveudf
5. Click Next and Finish.
   The generated pom.xml appears in sample-hiveudf.
6. To the pom.xml, add properties to facilitate versioning.
   For example:

   ```
   <properties>
      <hadoop.version>3.1.1.3.1.0.0-78</hadoop.version>
      <hive.version>3.1.0.3.1.0.0-78</hive.version>
   </properties>
   ```
7. In the pom.xml, define the repositories.

   Use internal repositories if you do not have internet access.
8. Define dependencies.
   For example:

   ```
   <dependencies>
     <dependency>
          <groupId>org.apache.hive</groupId>
          <artifactId>hive-exec</artifactId>
          <version>${hive.version}</version>
       </dependency>
       <dependency>
          <groupId>org.apache.hadoop</groupId>
          <artifactId>hadoop-common</artifactId>
          <version>${hadoop.version}</version>
       </dependency>
   </dependencies>
   ```
9. Select File > Project Structure. Click Modules. On the Dependencies tab, click + to add JARS or directories. Browse to and select the JARs you downloaded in step 1.

Create the UDF class

You define the UDF logic in a new class that returns the data type of a selected column in a table.

Procedure

1. In IntelliJ, click the vertical project tab, and expand hiveudf: hiveudf > src > main. Select the java directory, and on the context menu, select New > Java Class and name the class, for example, TypeOf.
2. Extend the GenericUDF class to include the logic that identifies the data type of a column.

For example:

```
package com.mycompany.hiveudf;

import org.apache.hadoop.hive.ql.exec.UDFArgumentException;
import org.apache.hadoop.hive.ql.metadata.HiveException;
import org.apache.hadoop.hive.ql.udf.generic.GenericUDF;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector;
import
 org.apache.hadoop.hive.serde2.objectinspector.primitive.PrimitiveObjectInspectorFact
import org.apache.hadoop.io.Text;

public class TypeOf extends GenericUDF {
  private final Text output = new Text();

@Override
  public ObjectInspector initialize(ObjectInspector[] arguments) throws
 UDFArgumentException {
    checkArgsSize(arguments, 1, 1);
    checkArgPrimitive(arguments, 0);
    ObjectInspector outputOI =
 PrimitiveObjectInspectorFactory.writableStringObjectInspector;
    return outputOI;
  }

@Override
  public Object evaluate(DeferredObject[] arguments) throws HiveException
 {
    Object obj;
    if ((obj = arguments[0].get()) == null) {
      String res = "Type: NULL";
      output.set(res);
      } else {
      String res = "Type: " + obj.getClass().getName();
      output.set(res);
    }
    return output;
  }

@Override
  public String getDisplayString(String[] children) {
    return getStandardDisplayString("TYPEOF", children, ",");
  }
}
```

Build the project and upload the JAR

You compile the UDF code into a JAR and place the JAR on the cluster. You choose one of several methods of configuring the cluster so Hive can find the JAR.

About this task

In this task, you choose one of several methods for configuring the cluster to find the JAR:

- Direct reference

  Straight-forward, but recommended for development only.
- Hive aux library directory method

  Prevents accidental overwriting of files or functions. Recommended for tested, stable UDFs to prevent accidental overwriting of files or functions.
- Reloadable aux JAR Avoids HiveServer restarts.

  Recommended if you anticipate making frequent changes to the UDF logic.

Procedure

1. Build the IntelliJ project.

```
...
[INFO] Building jar: /Users/max/IdeaProjects/hiveudf/target/TypeOf-1.0-
SNAPSHOT.jar
[INFO]
 ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
 ------------------------------------------------------------------------
[INFO] Total time: 14.820 s
[INFO] Finished at: 2019-04-03T16:53:04-07:00
[INFO] Final Memory: 26M/397M
[INFO]
 ------------------------------------------------------------------------

Process finished with exit code 0
```

2. Navigate to the JAR in the /target directory of the project.
3. Configure the cluster so Hive can find the JAR using one of the following methods.
   - Direct JAR reference
     a. Upload the JAR to the cluster.
     b. Move the JAR into the Hive warehouse.
   - Reloadable Aux JAR
     a. Upload the JAR to the /hadoop/hive-udf-dyn directory on all HiveServer instances (and all Metastore instances, if separate).
     b. In hive-site.xml, set the following property: hive.reloadable.aux.jars.path=/hadoop/hive-udf-dyn.
   -

Register the UDF

In the cluster, you log into Hive, and run a command from Beeline to make the UDF functional in Hive queries. The UDF persists between HiveServer restarts.

Before you begin

You need to set up UDF access, using a Ranger policy for example.

About this task

In this task, the registration command differs depending on the method you choose to configure the cluster for finding the JAR. If you use another method, you do not need to restart HiveServer. You must recreate the symbolic link after any patch or maintenance upgrades that deploy a new version of Hive.

Procedure

1. Using Beeline, login to HiveServer or HiveServer Interactive as a user who has UDF access.
   - HiveServer, for example:

     ```
     beeline -u jdbc:hive2://mycluster.com:10000 -n hive -p
     ```

   - HiveServer Interactive, for example:

     ```
     beeline -n hive -u jdbc:hive2://
     mycluster.com:10500/;transportMode=binary
     ```

2. At the Hive prompt, select a database for use.

USE default;

3. Run the registration command that corresponds to the way you configured the cluster to find the JAR.

   In the case of the direct JAR reference configuration method, you include the JAR location in the command. If you use another method, you do not include the JAR location. The classloader can find the JAR.

   - Direct JAR reference:

     ```
     CREATE FUNCTION udftypeof AS 'com.mycompany.hiveudf.TypeOf01' USING JAR
      'S3:///warehouse/tablespace/managed/TypeOf01-1.0-SNAPSHOT.jar';
     ```

   - Reloadable aux JAR:

     ```
     RELOAD;
     CREATE FUNCTION udftypeof AS 'com.mycompany.hiveudf.Typeof01';
     ```

   -

4. Check that the UDF is registered.

   ```
   SHOW FUNCTIONS;
   ```

   You scroll through the output and find default.typeof.

Call the UDF in a query

After registration of a UDF, you do not need to restart Hive before using the UDF in a query. In this example, you call the UDF you created in a SELECT statement, and Hive returns the data type of a column you specify.

Before you begin

- For the example query in this task, you need to create a table in Hive and insert some data.
- As a user, you need to have permission to call a UDF, which a Ranger policy can provide.

About this task

This task assumes you have the following example table in Hive:

```

+------------------+----------------+---------------+
|  students.name   | students.age  | students.gpa  |
+------------------+----------------+---------------+
| fred flintstone  | 35             | 1.28          |
| barney rubble    | 32             | 2.32          |
+------------------+----------------+---------------+
```

Procedure

1. Use the database in which you registered the UDF.
   For example:

   ```
   USE default;
   ```

2. Query Hive depending on how you configured the cluster for Hive to find the JAR.

   - Direct JAR reference

     For example:

     ```
     SELECT students.name, udftypeof(students.name) AS type FROM students
      WHERE age=35;
     ```

   - Reloadable aux JAR

For example:

```
RELOAD;
SELECT students.name, udftypeof(students.name) AS type FROM students
 WHERE age=35;
```

You get the data type of the name column in the students table.

```
+-------------------+-----------------------------------------------------+
|  students.name    |                        type                         |
+-------------------+-----------------------------------------------------+
| fred flintstone   | Type:
 org.apache.hadoop.hive.serde2.io.HiveVarcharWritable |
+-------------------+-----------------------------------------------------+
```