Cloudera Runtime 7.3.3

Using Apache Hive

Date published: 2019-08-21
Date modified: 2026-11-13



https://docs.cloudera.com/

Legal Notice

© Cloudera Inc. 2026. All rights reserved.

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

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

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

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

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

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

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

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

Contents

Apache Hive 3 tables **5**
Locating Hive tables and changing the location 6
Refer to a table using dot notation 7
Understanding CREATE TABLE behavior 7
Creating a CRUD transactional table 9
Creating an insert-only transactional table 9
Creating, using, and dropping an external table 10
Creating an Ozone-based external table 11
Accessing Hive files in Ozone 12
Recommended Hive configurations when using Ozone 14
Dropping an external table along with data 15
Converting a managed non-transactional table to external 15
Using constraints 16
Determining the table type 18
Apache Hive 3 ACID transactions **18**
Apache Hive query basics **21**
Querying the information_schema database 21
Inserting data into a table 23
Updating data in a table 24
Merging data in tables 24
Deleting data from a table 25
Creating a temporary table 25
Configuring temporary table storage 25
Using a subquery 26
Subquery restrictions 26
Use wildcards with SHOW DATABASES 27
Aggregating and grouping data 27
Querying correlated data 27
Using common table expressions 28
Use a CTE in a query 28
Comparing tables using ANY/SOME/ALL 29
Escaping an invalid identifier 29
CHAR data type support 30
ORC vs Parquet formats 30
Hive reserved words 31
Creating a default directory for managed tables 31
Configuring a table location outside the warehouse root directory 32
Generating surrogate keys 32
Partitions and performance 34
Creating partitions dynamically 35
Partition refresh and configuration 36
Repairing partitions manually using MSCK repair 38
Query scheduling 39
Enabling scheduled queries 40
Enabling all scheduled queries 40
Periodically rebuilding a materialized view 40
Getting scheduled query information and monitor the query 41
Lateral View 43
Materialized views 43
Creating and using a materialized view 43
HPL/SQL stored procedures 52

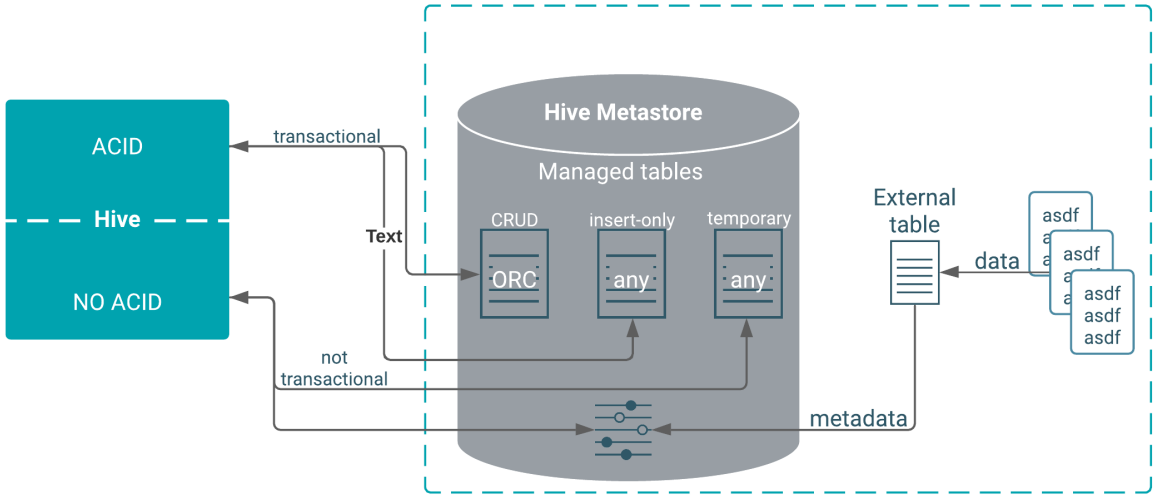
Creating a function.....54
Setting up a Hive client.....55
Using the cursor to return record sets.....55
Stored procedure examples.....56
Using JdbcStorageHandler to query RDBMS.....58
Using functions.....59
Reloading, viewing, and filtering functions.....59
Create a user-defined function.....60

Apache Hive 3 tables

Table type definitions and a diagram of the relationship of table types to ACID properties clarifies Hive tables. The location of a table depends on the table type. You might choose a table type based on its supported storage format.

You can create ACID (atomic, consistent, isolated, and durable) tables for unlimited transactions or for insert-only transactions. These tables are Hive managed tables. Alternatively, you can create an external table for non-transactional use. Because Hive control of the external table is weak, the table is not ACID compliant.

The following diagram depicts the Hive table types.



The following matrix includes 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.

Table storage formats

The data in CRUD tables must be in ORC format. Implementing a storage handler that supports AcidInputFormat and AcidOutputFormat is equivalent to specifying ORC storage.

Insert-only tables support all file formats.

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, you get an ACID table with insert, update, and delete (CRUD) capabilities. If you specify any other storage type, such as text, CSV, AVRO, or JSON, you get an insert-only ACID table. You cannot update or delete columns in the insert-only table.

Transactional tables

Transactional tables are ACID tables that reside in the Hive warehouse. To achieve ACID compliance, Hive has to manage the table, including access to the table data. Only through Hive can you access and change the data in managed tables. Because Hive has full control of managed tables, Hive can optimize these tables extensively.

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. Bucketing does not affect performance. 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
- Automatic runtime filtering
- File merging after insert
- ARCHIVE, UNARCHIVE, TRUNCATE, MERGE, and CONCATENATE. These statements only work for Hive Managed tables.

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.

Locating Hive tables and changing the location

You need to know where Hive stores tables on your object store or file system. Also, right after installing Hive-on-Tez, you might also want to change the warehouse location.

About this task

New tables that you create in Cloudera are stored in either the Hive warehouse for managed tables or the Hive warehouse for external tables. The following default warehouse locations are in the HDFS file system:

- /warehouse/tablespace/managed/hive
- /warehouse/tablespace/external/hive

Managed tables reside in the managed tablespace, which only Hive can access. By default, Hive assumes external tables reside in the external tablespace.

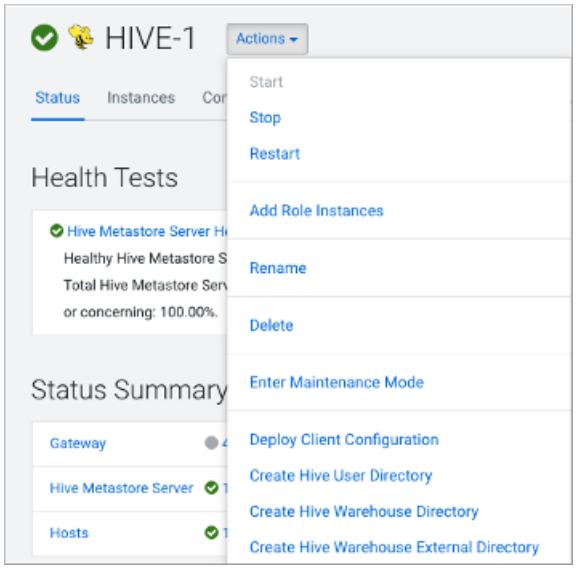
To determine the managed or external table type, you can run the DESCRIBE EXTENDED table_name command.

You need to set Ranger policies to access external tables in the object store or file system, such as HDFS.

The capability to change the location of the Hive warehouse is intended for use immediately after installing the service. You can change the location of the warehouse using the Hive Metastore Action menu in Cloudera Manager as described in the following steps:

Procedure

1. In Cloudera Manager, click Clusters Hive Action Menu Create Hive Warehouse Directory .



2. In Cloudera Manager, click Clusters Hive (the Hive Metastore service) Configuration , and change the hive.met astore.warehouse.dir property value to the path for the new Hive warehouse directory.
3. Click Hive Hive Action Menu Create Hive Warehouse External Directory .
4. Change the hive.metastore.warehouse.external.dir property value to the path for the Hive warehouse external directory.
5. Configure Ranger Hadoop SQL policy to access the URL of the directory on the object store, such as S3 or Ozone, or file system, such as HDFS.

Refer to a table using dot notation

Hive 3.1 changes to table references using dot notation might require changes to your Hive scripts.

About this task

Hive 3.1 in Cloudera includes SQL compatibility (Hive-16907), which rejects `db.table` in SQL queries. The dot (.) is not allowed in table names. To reference the database and table in a table name, enclosed both in backticks as follows:

```
`db`.`table`
```

Understanding CREATE TABLE behavior

Hive table creation has changed significantly since Hive 3 to improve useability and functionality. If you are upgrading from CDH or HDP, you must understand the changes affecting legacy table creation behavior.

Hive has changed table creation in the following ways:

- Creates ACID-compliant table, which is the default in Cloudera
- Supports simple writes and inserts
- Writes to multiple partitions
- Inserts multiple data updates in a single SELECT statement
- Eliminates the need for bucketing.

If you have an ETL pipeline that creates tables in Hive, the tables will be created as ACID. Hive now tightly controls access and performs compaction periodically on the tables. Using ACID-compliant, transactional tables causes no

performance or operational overload. The way you access managed Hive tables from Spark and other clients changes. In Cloudera, access to external tables requires you to set up security access permissions.

You must understand the behavior of the CREATE TABLE statement in legacy platforms like CDH or HDP and how the behavior changes after you upgrade to Cloudera.

Before upgrading to Cloudera Base on premises

In CDH 5, CDH 6, and HDP 2, by default CREATE TABLE creates a non-ACID managed table in plain text format.

In HDP 3 and CDP 7.1.0 through 7.1.7.x, by default CREATE TABLE creates either a full ACID transactional table in ORC format or insert-only ACID transactional tables for all other table formats.

After upgrading to Cloudera Base on premises

- If you are upgrading from HDP 2, CDH 5, or CDH 6 to CDP 7.1.0 through CDP 7.1.8, by default CREATE TABLE creates a full ACID transactional table in ORC format or insert-only ACID transactional tables for all other table formats.
- If you are upgrading from HDP 3 or CDP 7.1.0 through 7.1.7.x to CDP 7.1.8, the existing behavior persists and CREATE TABLE creates either a full ACID transactional table in ORC format or insert-only ACID transactional tables for all other table formats.

Now that you understand the behavior of the CREATE TABLE statement, you can choose to modify the default table behavior by configuring certain properties. The order of preference for configuration is as follows:

Modify the default CREATE TABLE behavior

Override default behavior when creating the table

Irrespective of the database, session, or site-level settings, you can override the default table behavior by using the MANAGED or EXTERNAL keyword in the CREATE TABLE statement.

```
CREATE [MANAGED][EXTERNAL] TABLE foo (id INT);
```

Set the default table type at a database level

You can use the database property, defaultTableType=EXTERNAL or ACID to specify the default table type to be created using the CREATE TABLE statement. You can specify this property when creating the database or at a later point using the ALTER DATABASE statement. For example:

```
CREATE DATABASE test_db WITH DBPROPERTIES ('defaultTableType'='E
XTERNAL');
```

In this example, tables created under the test_db database using the CREATE TABLE statement creates external tables with the purge fucntionality enabled (external.table.purge = 'true').

You can also choose to configure a database to allow only external tables to be created and prevent creation of ACID tables. While creating a database, you can set the database property, EXTER NAL_TABLES_ONLY=true to ensure that only external tables are created in the database. For example:

```
CREATE DATABASE test_db WITH DBPROPERTIES ('EXTERNAL_TABLES_ONLY
'='true');
```

Set the default table type at a session level

You can configure the CREATE TABLE behavior within an existing beeline session by setting hive.create.as.external.legacy to true or false. Setting the value to true results in configuring the CREATE TABLE statement to create external tables by default.

When the session ends, the default CREATE TABLE behavior also ends.

Set the default table type at a site level

You can configure the CREATE TABLE behavior at the site level by configuring the hive.create.as.insert.only and hive.create.as.acid properties in Cloudera Manager under Hive configuration. When configured at the site level, the behavior persists from session to session. For more information, see Configuring CREATE TABLE behavior.

If you are a Spark user, switching to legacy behavior is unnecessary. Calling 'create table' from SparkSQL, for example, creates an external table after upgrading to Cloudera as it did before the upgrade. You can connect to Hive using the Hive Warehouse Connector (HWC) to read Hive ACID tables from Spark. To write ACID tables to Hive from Spark, you use the HWC and HWC API. Spark creates an external table with the purge property when you do not use the HWC API. For more information, see Hive Warehouse Connector for accessing Spark data.

Related Information

Configuring legacy CREATE TABLE behavior

Creating a CRUD transactional table

You create a CRUD transactional table having ACID (atomic, consistent, isolated, and durable) properties when you need a managed table that you can update, delete, and merge. You learn by example how to determine the table type.

About this task

In this task, you create a CRUD transactional table. You cannot sort this type of table. To create a CRUD transactional table, you must accept the default ORC format by not specifying any storage during table creation, or by specifying ORC storage explicitly.

Procedure

1. Start Hive.
2. Enter your user name and password.
   The Hive 3 connection message, followed by the Hive prompt for entering SQL 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);
```

4. Confirm that you created a managed, ACID table.

```
DESCRIBE FORMATTED T;
```

   The table type says MANAGED_TABLE and transactional = true.

Related Information

HMS storage

Creating an insert-only transactional table

You can create a transactional table using any storage format if you do not require update and delete capability. This type of table has ACID properties, is a managed table, and accepts insert operations only. The storage format of an insert-only table is not restricted to ORC.

About this task

In this task, you create an insert-only transactional table for storing text. In the CREATE TABLE statement, specifying a storage type other than ORC, such as text, CSV, AVRO, or JSON, results in an insert-only ACID table. You can explicitly specify insert-only in the table properties clause.

Procedure

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

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

   The 'transactional_properties'='insert_only' is required; otherwise, a CRUD table results. The STORED AS ORC clause is optional (default = ORC).
4. Create an insert-only transactional table for text data.

```
CREATE TABLE T3(a int, b int)
STORED AS TEXTFILE;
```

   The 'transactional_properties'='insert_only' is not required because the storage format is other than ORC.

Related Information

HMS storage

Creating, using, and dropping an external table

You use an external table, which is a table that Hive does not manage, to import data from a file on a file system into Hive. In contrast to the Hive managed table, an external table keeps its data outside the Hive metastore. Hive metastore stores only the schema metadata of the external table. Hive does not manage, or restrict access, to the actual external data.

Before you begin

You need to set up access to external tables in the file system or object store using Ranger.

About this task

In this task, you create an external table from CSV (comma-separated values) data stored on the file system. Next, you want Hive to manage and store the actual data in the metastore. You create a managed table and insert the external table data into the managed table.

This task demonstrates the following Hive principles:

- The LOCATION clause in the CREATE TABLE specifies the location of external table data.
- 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, consisting mainly of the schema.
  - Managed table drop: Hive deletes the data and the metadata stored in the Hive warehouse.

After dropping an external table, the data is not gone. To retrieve it, you issue another CREATE EXTERNAL TABLE statement to load the data from the file system.

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 HDFS in a directory called andrena, and put students.csv in the directory.
3. Start the Hive shell.
   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 /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';
```

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.
10. Check that the students.csv file on the file system or object store remains intact.

Related Information

HMS storage
HDFS ACLS

Creating an Ozone-based external table

You use the LOCATION clause in the CREATE EXTERNAL TABLE statement to create an external table having source data in Ozone.

About this task

In this task, you create an external table and load data from a .csv file that is stored in Ozone. You can use the LOCATION clause in the CREATE EXTERNAL TABLE statement to specify the location of the external table data. The metadata is stored in the Hive warehouse.

Before you begin

- If you are running an insecure cluster, ensure that you set up the necessary policies and permissions in Ranger to give user access to Hive external files in Ozone.
- You must have created a volume and bucket in Ozone. For example:

```
ozone sh volume create vol1
ozone sh bucket create vol1/bucket1
```

- It is recommended that you set certain Hive configurations before querying Hive tables in Ozone.
- For the Hive Table creation, the warehouse directory must be set at bucket level or directory level under the hive.metastore.warehouse.dir or hive.metastore.warehouse.external.dir parameters. For more information, see Changing the Hive warehouse location.

Procedure

1. Create a text file named employee.csv that contains the following records.

```
1,Andrew,45000,Technical Manager
2,Sam,35000,Proof Reader
```

2. Move the employee.csv file to a directory called employee_hive in the Ozone filesystem.
3. Connect to the gateway node of the cluster and on the command line of the cluster, launch Beeline to start the Hive shell.
   beeline -u jdbc:hive2://myhiveserver.com:10000 -n hive -p
   The Hive connection message appears, followed by the Hive prompt for entering queries on the command line.
4. Create an external table schema definition that specifies the text format and loads data from employee.csv in ofs://ozone1/vol1/bucket1/employee_hive.

```
CREATE EXTERNAL TABLE IF NOT EXISTS employee(
  employee_ID INT,
  name STRING,
  salary DOUBLE,
  designation STRING)
  ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
  STORED AS TEXTFILE
  LOCATION 'ofs://ozone1/vol1/bucket1/employee_hive';
```

5. Verify that the Hive warehouse stores the employee.csv records in the external table.
   SELECT * FROM employee;

Related Information

Setting up Ranger policies to access Hive files in Ozone
Commands for managing Ozone volumes and buckets
Recommended Hive configurations when using Ozone

Accessing Hive files in Ozone

Learn how to set up policies to give users access to Hive external files in Ozone. For example, if Ozone users are running SparkSQL statements that query Hive tables, you must set up an Ozone access policy and Ozone file system access policy.

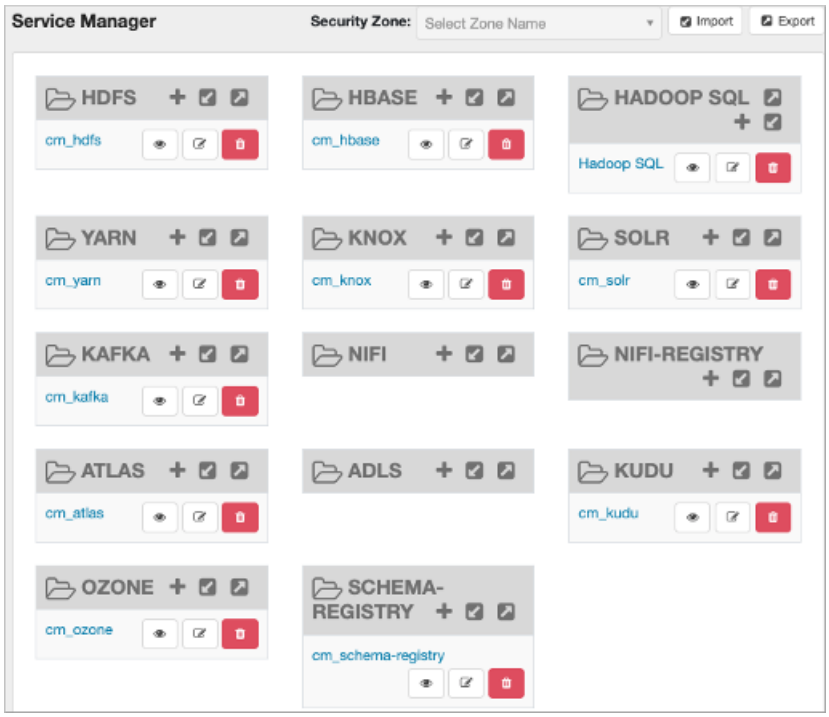
About this task

When Ranger is enabled in the cluster, any user other than the default admin user, "om" requires the necessary Ranger permissions and policy updates to access the Ozone filesystem. To create a Hive external table that points to the Ozone filesystem, the "hive" user should have the required permissions in Ranger.

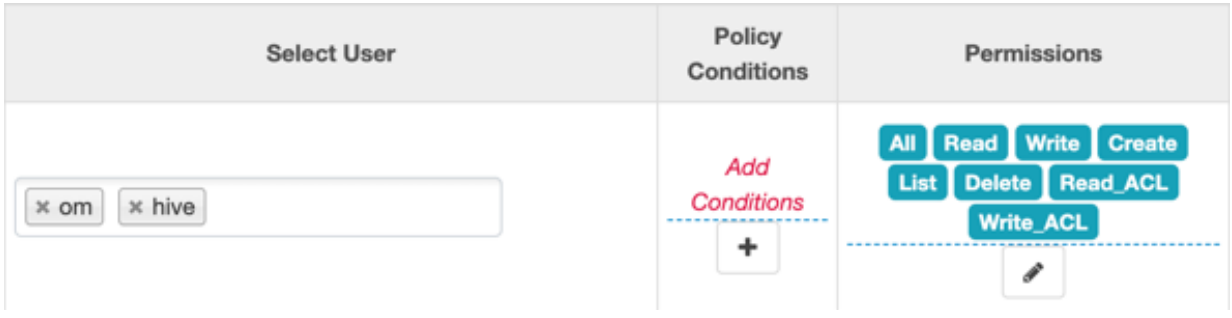
In this task, you first enable Ozone in the Ranger service, and then set up the required policies.

Procedure

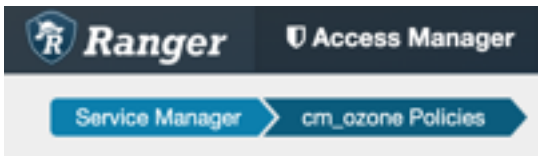
1. In Cloudera Manager, click Clusters Ozone Configuration to navigate to the configuration page for Ozone.
2. Search for ranger_service, and enable the property.
3. Click Clusters Ranger Ranger Admin Web UI , enter your user name and password, then click Sign In.
   The **Service Manager for Resource Based Policies** page is displayed in the Ranger console.



4. Click the cm_ozone preloaded resource-based service to modify an Ozone policy.
5. In the cm_ozone policies page, click the Policy ID or click  Edit against the "all - volume, bucket, key" policy to modify the policy details.
6. In the Allow Conditions pane, add the "hive" user, choose the necessary permissions, and then click Save.

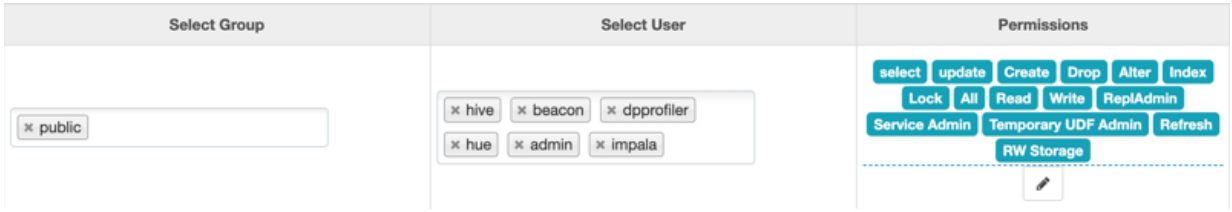


7. Click the Service Manager link in the breadcrumb trail and then click the Hadoop SQL preloaded resource-based service to update the Hadoop SQL URL policy.



8. In the Hadoop SQL policies page, click the Policy ID or click  Edit against the "all - url" policy to modify the policy details.

   By default, "hive", "Data Explorer", "impala", "admin" and a few other users are provided access to all the Ozone URLs. You can select users and groups in addition to the default. To grant everyone access, add the "public" group to the group list. Every user is then subject to your allow conditions.



What to do next

Create a Hive external table having source data in Ozone.

Also, it is recommended that you set certain Hive configurations before querying Hive tables in Ozone.

Related Information

Using Ranger with Ozone

Creating an Ozone-based Hive external table

Creating partitions dynamically

Recommended Hive configurations when using Ozone

Recommended Hive configurations when using Ozone

It is recommended that you configure certain Hive properties if you are querying Hive tables in Ozone.

Configurations

The following configurations can be specified through Beeline during runtime using the SET command. For example, SET key=value;. The configuration persists only for the particular session or query. If you want to set it permanently, then specify the properties in hive-site.xml using the Cloudera Manager Safety Valve:

Table 1:

Configuration	Value
hive.optimize.sort.dynamic.partition	true
hive.optimize.sort.dynamic.partition.threshold	0
hive.query.results.cache.enabled	false
hive.acid.direct.insert.enabled	true
hive.orc.splits.include.fileid	false

Important: If you notice that some queries are taking a longer time to complete or failing entirely (usually noticed in large clusters), you can choose to revert the value of hive.optimize.sort.dynamic.partition.threshold to "-1". The performance issue is related to HIVE-26283.

Related Information

Creating an Ozone-based Hive external table

Setting up Ranger policies to access Hive files in Ozone

Dropping 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. Start Hive.
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 '<file system>://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 the file system.

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

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

Converting a managed non-transactional table to external

You can easily convert a managed table, if it is not an ACID (transactional) table, to external using the ALTER TABLE statement. You might have a non-ACID, managed table after an upgrade from Hive 1 or 2.

About this task

The following pseudo-code changes a managed table, if it is not transactional, to external. The data and metadata is dropped when the table is dropped.

```
ALTER TABLE ... SET TBLPROPERTIES('EXTERNAL'='TRUE','external.table.purge'='
true')
```

Related Information

Before and After Upgrading Table Type Comparison

Using constraints

You can use SQL constraints to enforce data integrity and improve performance. Using constraints, the optimizer can simplify queries. Constraints can make data predictable and easy to locate. Using constraints and supported modifiers, you can follow examples to constrain queries to unique or not null values, for example.

Hive enforces DEFAULT, NOT NULL and CHECK only, not PRIMARY KEY, FOREIGN KEY, and UNIQUE.

You can use the constraints listed below in your queries. Hive enforces DEFAULT, NOT NULL and CHECK only, not PRIMARY KEY, FOREIGN KEY, and UNIQUE. DEFAULT even if enforced, does not support complex types (array,map,struct). Constraint enforcement is limited to the metadata level. This limitation aids integration with third party tools and optimization of constraints declarations, such as materialized view rewriting.

CHECK

Limits the range of values you can place in a column.

DEFAULT

Ensures a value exists, which is useful in offloading data from a data warehouse.

PRIMARY KEY

Identifies each row in a table using a unique identifier.

FOREIGN KEY

Identifies a row in another table using a unique identifier.

UNIQUE KEY

Checks that values stored in a column are different.

NOT NULL

Ensures that a column cannot be set to NULL.

Supported modifiers

You can use the following optional modifiers:

ENABLE

Ensures that all incoming data conforms to the constraint.

DISABLE

Does not ensure that all incoming data conforms to the constraint.

VALIDATE

Checks that all existing data in the table conforms to the constraint.

NOVALIDATE

Does not check that all existing data in the table conforms to the constraint.

ENFORCED

Maps to ENABLE NOVALIDATE.

NOT ENFORCED

Maps to DISABLE NOVALIDATE.

RELY

Specifies abiding by a constraint; used by the optimizer to apply further optimizations.

NORELY

Specifies not abiding by a constraint.

You use modifiers as shown in the following syntax:

```
( ( ( (ENABLE | DISABLE) (VALIDATE | NOVALIDATE) ) | ( ENFORCED | NOT ENFORC
ED ) ) (RELY | NORELY) )
```

Note: For external tables, the RELY constraint is the only supported constraint.

Default modfiers

The following default modifiers are in place:

- The default modifier for ENABLE is NOVALIDATE RELY.
- The default modifier for DISABLE is NOVALIDATE NORELY.
- If you do not specify a modifier when you declare a constraint, the default is ENABLE NOVALIDATE RELY. The following constraints do not support ENABLE:
  - PRIMARY KEY
  - FOREIGN KEY
  - UNIQUE KEY

  To prevent an error, specify a modfier when using these constraints to override the default.

Constraints examples

The optimizer uses the constraint information to make smart decisions. The following examples show the use of constraints.

The following example shows how to create a table that declares the NOT NULL in-line constraint to constrain a column.

```
CREATE TABLE t(a TINYINT, b SMALLINT NOT NULL ENABLE, c INT);
```

The constrained column b accepts a SMALLINT value as shown in the first INSERT statement.

```
INSERT INTO t values(2,45,5667);
    ...
    ------------------------------------------------------------
    1 row affected ...
```

The constrained column b will not accept a NULL value.

```
INSERT INTO t values(2,NULL,5667);
    Error: Error running query: org.apache.hadoop.hive.ql.exec.errors.Dat
aConstraintViolationError: /
    Either CHECK or NOT NULL constraint violated! (state=,code=0)
```

The following examples shows how to declare the FOREIGN KEY constraint out of line. You can specify a constraint name, in this case fk, in an out-of-line constraint

```
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 NOVA
LIDATE
     );
```

Determining 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 completely 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 says the table type is managed. transaction=true indicates that the table has ACID properties:

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

Related Information

HMS storage

Apache 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. You can obtain query status information from these files and use the files to troubleshoot query problems.

Write and read operations

Hive 3 write and read operations improve the ACID qualities 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 fast 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.

Hive compacts ACID transaction files automatically without impacting concurrent queries. Automatic compaction improves query performance and the metadata footprint when you query many small, partitioned files.

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'='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

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:

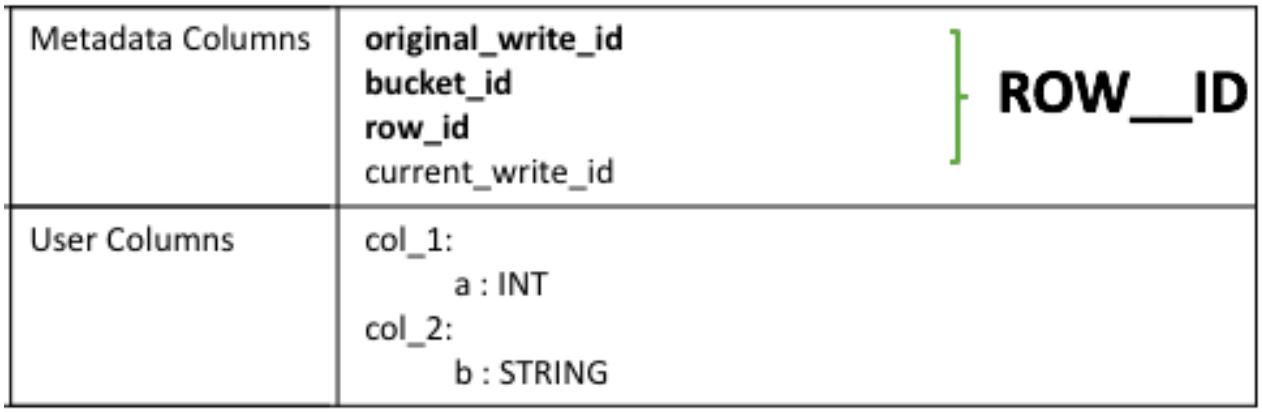
```
        +----------------------------------------------------+
        |                  createtab_stmt                    |
        +----------------------------------------------------+
        | CREATE TABLE `acidtbl`(                            |
        |   `a` int,                                         |
        |   `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/acidtb
l' |
        | 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 in append-only mode, 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



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, "b
ananas");
```

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:

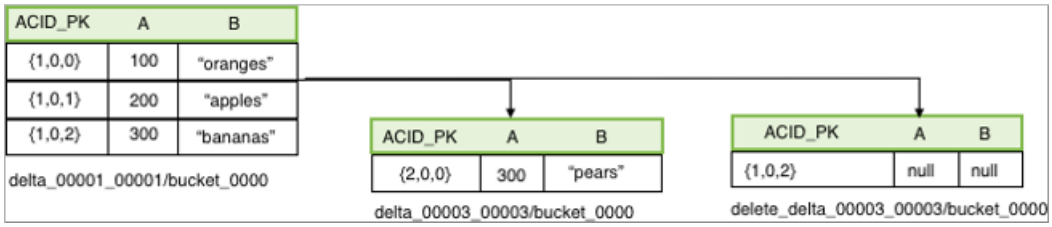
ROW_ID	a	b
{1,0,1}	null	null

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.

Apache Hive query basics

Using Apache Hive, you can query distributed data storage. You need to know ANSI SQL to view, maintain, or analyze Hive data. Examples of the basics, such as how to insert, update, and delete data from a table, helps you get started with Hive.

Hive supports ANSI SQL and atomic, consistent, isolated, and durable (ACID) transactions. For updating data, you can use the MERGE statement, which 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

Querying 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 following steps 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 Ranger Access Manager, and check that the preloaded default database tables columns and information_schema database policies are enabled for group public.

Policy ID	Policy Name	Policy Labels	Status	Audit Logging	Roles	Groups	Users
7	all - global	--	Enabled	Enabled	--	--	hive beacon dpprofiler hue + More..
8	all - database, table, column	--	Enabled	Enabled	--	--	hive beacon dpprofiler hue + More..
9	all - database, table	--	Enabled	Enabled	--	--	hive beacon dpprofiler hue + More..
10	all - database	--	Enabled	Enabled	--	public	hive beacon dpprofiler hue + More..
11	all - hiveservice	--	Enabled	Enabled	--	--	hive beacon dpprofiler hue + More..
12	all - database, udf	--	Enabled	Enabled	--	--	hive beacon dpprofiler hue + More..
13	all - url	--	Enabled	Enabled	--	--	hive beacon dpprofiler hue + More..
14	default database tables columns	--	Enabled	Enabled	--	public	--
15	Information_schema database ...	--	Enabled	Enabled	--	public	--

   The information schema database is synchronized every half hour by default.

2. Start Hive, and check for the information_schema database:

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

3. 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              |
+--------------------+
```

4. 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' l
imit 2;
...
+--------------------+-------------------+-----------------
|tables.table_catalog|tables.table_schema|tables.table_name
+--------------------+-------------------+-----------------
|default             |default            |students2
|default             |default            |t3
```

Inserting data into a table

To insert data into a table you use a familiar ANSI SQL statement. A simple example shows you have to accomplish this basic task.

About this task

To insert data into an ACID table, use the Optimized Row Columnar (ORC) storage format. To insert data into a non-ACID table, you can use other supported formats. 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]) .

A value can be NULL or any SQL literal.

Procedure

1. Create an ACID 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, origin 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');
```

   The ACID data resides in the warehouse.

Updating data in a table

The syntax describes the UPDATE statement you use to modify data already stored in a table. An example shows how to apply the syntax.

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 SELECT clause. Subqueries are not allowed on the right side of the SET statement. Partition 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;

Merging data in tables

A sample statement shows how you can conditionally insert existing data in Hive tables using the ACID MERGE statement. Additional merge operations are mentioned.

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 target table to match the names and states of customers having the same IDs in the new_customer_stage source table.
2. Enhance the query to insert data from new_customer_stage table into the customer table if none already exists.

   Update or delete data using MERGE in a similar manner.

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



Note: You can map specific columns in the INSERT clause of the query instead of passing values (including null) for columns in the target table that do not have any data to insert. The unspecified columns in the INSERT clause are either mapped to null or use default constraints, if any.

For example, you can construct the INSERT clause as WHEN NOT MATCHED THEN INSERT VALUES (customer.id=sub.id, customer.name=sub.name, customer.state=sub.state) instead of WHEN NOT MATCHED THEN INSERT VALUES (sub.id, sub.name, 'null', sub.state).

Related Information

Merge documentation on the Apache wiki

Deleting 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;

Creating a temporary table

In Cloudera Base on premises, you can create a temporary table to improve performance by storing data temporarily 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.

Cloudera Base on premises only

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

Configuring temporary table storage

In Cloudera Base on premises, 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.

Cloudera Base on premises only

Procedure

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

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

2. Create and use temporary tables.

Results

Hive drops temporary tables at the end of the session.

Using a subquery

Hive supports subqueries in FROM clauses and WHERE clauses that you can use for many Apache 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.
- 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 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.

Use wildcards with SHOW DATABASES

In a SHOW DATABASES LIKE statement, you can use wildcards and specify any character or a single character.

About this task

SHOW DATABASES or SHOW SCHEMAS lists all of the databases defined in Hive metastore. You can use the following wildcards:

%

Matches any single character or multiple characters.

_ (underscore)

Matches any single character.

|

Matches either the part of the pattern on the left or the right side of the pipe.

For example, 'students', 'stu%', 'stu_ents' match the database named students.

Aggregating and grouping 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';
```

Querying 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 OV
ERWRITE TABLE s1 SELECT *;
```

Comparing tables using ANY/SOME/ALL

You learn how to use quantified comparison predicates (ANY/SOME/ALL) in non-correlated subqueries according to the SQL standard. SOME is any alias for ANY.

About this task

You can use one of the following operators with a comparison predicate:

- >
- <
- >=
- <=
- <>
- =

ALL:

- If the table is empty, or the comparison is true for every row in subquery table, the predicate is true for that predicand.
- If the comparison is false for at least one row, the predicate is false.

SOME or ANY:

- If the comparison is true for at least one row in the subquery table, the predicate is true for that predicand.
- If the table is empty or the comparison is false for each row in subquery table, the predicate is false.

If the comparison is neither true nor false, the result is undefined.

For example, you run the following query to match any value in c2 of tbl equal to any value in c1 from the same tbl:

```
select c1 from tbl where c1 = ANY (select c2 from tbl);
```

You run the following query to match all values in c1 of tbl not equal to any value in c2 from the same tbl.

```
select c1 from tbl where c1 <> ALL (select c2 from tbl);
```

Escaping an invalid 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, in Hive execute the following command: SET 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 2: 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

ORC vs Parquet formats

The differences between Optimized Row Columnar (ORC) file format for storing data in SQL engines are important to understand. Query performance improves when you use the appropriate format for your application.

ORC and Parquet capabilities comparison

The following table compares SQL engine support for ORC and Parquet.

Table 3:

Capability	Cloudera Data Warehouse	ORC	Parquet	SQL Engine
Read non-transactional data	Apache Hive	#	#	Hive
Read non-transactional data	Apache Impala	#	#	Impala
Read/Write Full ACID tables	Apache Hive	#		Hive
Read Full ACID tables	Apache Impala	#		Impala
Read Insert-only managed tables	Apache Impala	#	#	Impala
Column index	Apache Hive	#	#	Hive
Column index	Apache Impala		#	Impala
CBO uses column metadata	Apache Hive	#		Hive

Capability	Cloudera Data Warehouse	ORC	Parquet	SQL Engine
Recommended format	Apache Hive	#		Hive
Recommended format	Apache Impala		#	Impala
Vectorized reader	Apache Hive	#	#	Hive
Read complex types	Apache Impala	#	#	Impala
Read/write complex types	Apache Hive	#	#	Hive

Hive reserved words

Refer to this documentation to identify reserved and non-reserved keywords in Hive.

Hive defines a set of reserved keywords that cannot be used directly as identifiers in SQL statements. To avoid conflicts, ensure that table names, column names, and other object names do not match any reserved keywords.

For the latest list of reserved and non-reserved keywords, see *Apache Wiki: Keywords, Non-reserved Keywords and Reserved Keywords*.

Related Information

Apache Wiki: Keywords, Non-reserved Keywords and Reserved Keywords

Creating a default directory for managed tables

You can specify a top-level directory for managed tables when creating a Hive database.

About this task

Create a default directory for managed tables only after limiting CREATE DATABASE and ALTER DATABASE statements to users having the Admin role, which has hive service user permissions. Permissions to the managed directory must be limited to the hive service user. In addition to restricting permissions to the hive user, you can further secure managed tables using Ranger fine-grained permissions, such as row-level filtering and column masking.

As Admin, you specify a managed location within the default location specified by the hive.metastore.warehouse.dir configuration property to give managed tables a common location for governance policies. The managed location designates a single root directory for all tenant tables, managed and external.

Setting the metastore.warehouse.tenant.colocation property to true allows a common location for managed tables outside the warehouse root directory, providing a tenant-based common root for setting quotas and other policies. To set this property, in Cloudera Manager use the Hive on Tez safety valve for hive-site.xml as described in the next topic.

Use the following syntax to create a database that specifies a location for managed tables:

```
CREATE (DATABASE|SCHEMA) [IF NOT EXISTS] database_name
    [COMMENT database_comment]
    [LOCATION external_table_path]
    [MANAGEDLOCATION managed_table_directory_path]
    [WITH DBPROPERTIES (property_name=property_value, ...)];
```

Do not set LOCATION and MANAGEDLOCATION to the same file system path.

Use the following syntax to set or change a location for managed tables.

```
ALTER (DATABASE|SCHEMA) database_name SET MANAGEDLOCATION [managed_table_dir
ectory_path];
```

Procedure

1. Create a database mydatabase that specifies a top level directory named sales for managed tables.

   ```
   CREATE DATABASE mydatabase MANAGEDLOCATION '/warehouse/tablespace/managed/
   hive/sales';
   ```

2. Change the abc_sales database location to the same location as mydatabase.

   ```
   ALTER DATABASE abc_sales SET MANAGEDLOCATION '/warehouse/tablespace/mana
   ged/hive/sales';
   ```

Configuring a table location outside the warehouse root directory

You need to know how to set the property that allows you to establish a common location for tables.

Procedure

1. In Cloudera Manager, click Clusters Hive on Tez Configuration .
2. In scope, click Hive on Tez (Service-Wide).
3. In Hive Service Advanced Configuration Snippet (Safety Valve) for hive-site.xml, click +.
4. In Name, enter metastore.warehouse.tenant.colocation.
5. In Value, enter true.



6. Save changes.
7. In Cloudera Manager Home, restart Hive on Tez.

Generating 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;
```

Now you can achieve fast joins on the surrogate keys.

Partitions and performance

A brief description of partitions and the performance benefits includes characters you must avoid when creating a partition. Examples of creating a partition and inserting data in a partition introduce basic partition syntax. Best practices for partitioning are mentioned.

A table you create without partitioning puts the data in a single directory. Partitioning divides the data into multiple directories. Queries of one or more columns based on the directories can run faster. Lengthy full table scans are avoided. Only data in the relevant directory is scanned. For example, a school_records table partitioned on a year column, segregates values by year into separate directories. A WHERE condiition such as YEAR=2020, YEAR IN (2020,2019), or YEAR BETWEEN 2001 AND 2010 scans only the data in the appropriate directory to resolve the query. Using partitions typically improves query performance.

In a SQL query, you define the partition as shown in the following example:

```
CREATE TABLE sale(id in, amount decimal) PARTITIONED BY (xdate string, state
 string);
```

To insert data into this table, you specify the partition key for fast loading:

```
INSERT INTO sale (xdate='2016-03-08', state='CA') SELECT * FROM staging_tabl
e WHERE xdate='2016-03-08' AND state='CA';
```

You do not need to specify dynamic partition columns. Hive generates a partition specification if you enable dynamic partitions.

Examples of a query on partitioned data

```
INSERT INTO sale (xdate, state)
SELECT * FROM staging_table;
```

Follow these best practices when you partition tables and query partitioned tables:

- Never partition on a unique ID.
- Size partitions to greater than or equal to 1 GB on average.
- Design queries to process not more than 1000 partitions.

Invalid Characters in a Partition Name

When you create a partition, do not use the following characters in a partition name:

- colon
- question mark
- percent

If you use these characters in a partition name, your directories will be named using the URL encoding of these characters, as described in "Why some special characters should not be used in a partition name in Hive/Impala."

Related Information

Why some special characters should not be used in a partition name in Hive/Impala

Creating partitions dynamically

You can configure Hive to create partitions dynamically and then run a query that creates the related directories on the file system or object store. 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.
2. 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 '<object store or file system path>';
   ```

   For example, specify an Ozone location 'ofs://ozone1/vol1/bucket1/employees'.
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. Accept the default dynamic partition mode (nonstrict) to create partitioned directories of data dynamically when data is inserted, or if you changed the default, reset the mode as follows:

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

Partition refresh and configuration

You can discover partition changes and synchronize Hive metadata automatically. Performing synchronization automatically, instead of 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. The time it takes to refresh the partition information is proportional to the number of partitions involved.

- Manually

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

  This command triggers a full repair which might take a longer time if the number of partitions are more. Instead, you can use operators in the command, such as EQUAL, NOTEQUAL, LESSTHANOREQUALTO, LESSTHAN, GREATERTHANOREQUALTO, GREATERTHAN, or LIKE in the partition column so that a larger subset of partitions can be recovered (added/removed) without triggering a full repair.

  Syntax: MSCK REPAIR TABLE <table_name> ADD/DROP/SYNC PARTITIONS (<pcol1><operator><value>,<pcol2><operator><value>);
- 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 the 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

Creating partitions dynamically

Apache Wiki: Discover Partitions and Partition Retention

Automating partition discovery and repair

Automated partition discovery and repair is useful for processing log data, and other data, in Spark and Hive catalogs. You learn how to set the partition discovery parameter to suit your use case. An aggressive partition discovery and repair configuration can delay the upgrade process.

About this task

Apache Hive can automatically and periodically discover discrepancies in partition metadata in the Hive metastore and in corresponding directories, or objects, on the file system. After discovering discrepancies, Hive performs synchronization.

The discover.partitions table property enables and disables synchronization of the file system with partitions. In external partitioned tables, this property is disabled (false) 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, discovery and synchronization of partitions occur every 5 minutes. If you are upgrading, this frequency causes performance degradation. Adjust the frequency of these batch routines to hourly or daily, depending on your use case.



Important: For partition discovery to work correctly, the HDFS or S3 (in cloud environment) directory structure must follow the format:/warehouse/tablespace/external/hive/my.db/mytable/p1=<value>/p2=<value>/.../pN=<value>. If the directory structure does not follow this convention, automatic partition discovery will not work. In such cases, you must manually recreate missing partitions in HMS. For example:

```
ALTER TABLE mytable ADD PARTITION (dt='2024-01-01')
LOCATION 's3://custom-bucket/external/my.db/mytable/2024-01-01';
```

For more details, see *Apache Wiki: Discover Partitions and Partition Retention*.

Procedure

1. For external partitioned tables and for legacy external tables that are 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. In Cloudera Manager, click Clusters Hive Configuration , search for Hive Server Advanced Configuration Snippet (Safety Valve) for hive-site.xml.
3. Add the following property and value to hive-site.xml: Property: metastore.partition.management.task.frequency Value: 600.

   The default value of metastore.partition.management.task.frequencyis 300. Changing this to 600 sets synchronization of partitions to occur every 10 minutes expressed in seconds. If you are upgrading to a cloud environment (AWS, AZURE, GCP), consider running discovery and synchonization once every 24 hours by setting the value to 86,400 seconds. Decreasing the value of metastore.partition.management.task.frequency incur higher cloud costs.

   Important: In a multi-Hive Metastore setup the Hive metastore instances will elect a single leader among themselves. There is no need to override the metastore.housekeeping.threads.on on the Hive Metastore instance level. This ensures efficient partition management by one instance while the other handles different tasks, minimizing locking issues and enhancing performance. For more information, see *Hive Metastore leader election*.

Related Information

Creating partitions dynamically

Apache Wiki: Discover Partitions and Partition Retention

Hive Metastore leader election

Managing partition retention time

You can keep the size of the Apache 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.

About this task

When a partition metadata retention period is specified, Hive will drop the metadata and corresponding data for partitions created after the retention period, and this action will also apply to existing partitions. 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.

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'='7
   d');
   ```

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

Related Information

Creating partitions dynamically

Apache Wiki: Discover Partitions and Partition Retention

Repairing partitions manually using MSCK repair

The MSCK REPAIR TABLE command was designed to manually add partitions that are added to or removed from the file system, 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.

Procedure

1. Remove the dept=sales object from the file system.
2. From the Hive command line, 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;
```

Query scheduling

Apache Hive scheduled queries is a simple, secure way to create, manage, and monitor scheduled jobs. For applications that require OS-level schedulers like cron, Apache Oozie, or Apache Airflow, you can use scheduled queries.

Using SQL statements, you can schedule Hive queries to run on a recurring basis, monitor query progress, and optionally disable a query schedule. You can run queries to ingest data periodically, refresh materialized views, replicate data, and perform other repetitive tasks. For example, you can insert data from a stream into a transactional table every 10 minutes, refresh a materialized view used for BI reporting every hour, and replicate data from one cluster to another on a daily basis.

A Hive scheduled query consists of the following parts:

- A unique name for the schedule
- The SQL statement to be executed
- The execution schedule defined by a Quartz cron expression.

Quartz cron expressions are expressive and flexible. For instance, expressions can describe simple schedules such as every 10 minutes, but also an execution happening at 10 AM on the first Sunday of the month in January, February in 2021, 2022. You can describe common schedules in an easily comprehensible format, for example every 20 minutes or every day at ‘3:25:00’.

Operation

A scheduled query belongs to a namespace, which is a collection of HiveServer (HS2) instances that are responsible for executing the query. Scheduled queries are stored in the Hive metastore. The metastore stores scheduled queries, the status of ongoing and previously executed statements, and other information. HiveServer periodically polls the metastore to retrieve scheduled queries that are due to be executed. If you run multiple HiveServer roles, the metastore guarantees that only one of them executes a certain scheduled query at any given time.

You create, alter, and drop scheduled queries using dedicated SQL statements.

Related Information

Apache Hive Language Manual--Scheduled Queries

Enabling scheduled queries

You need to know how to enable and disable scheduled queries and understand how the default state can prevent you from running a query unintentionally.

About this task

Scheduled queries are created in disabled mode by default in Cloudera. This default helps prevent you from running new scheduled queries inadvertantly. You must explicitly enable new scheduled queries. A scheduled query can keep the cluster awake at the wrong time. To enable a particular schedule, for example schedule1, you run the ALTER SCHEDULED QUERY statement:

```
ALTER SCHEDULED QUERY schedule1 ENABLE;
```

To disable this schedule: ALTER SCHEDULED QUERY schedule1 DISABLE;

Related Information

Apache Hive Language Manual--Scheduled Queries

Enabling all scheduled queries

You can enable a single scheduled query using a SQL command, but to enable multiple scheduled queries you need perform a few steps in Cloudera Manager.

About this task

To enable all newly created schedules in Cloudera Base on premises, follow these steps:

Procedure

1. In Cloudera Manager, click Clusters Hive on TEZ Configuration
2. In Search, enter safety.
3. In Hive Service Advanced Configuration Snippet (Safety Valve) for hive-site.xml HIVE_ON_TEZ-1 (Service-Wide), click + and add the following property: hive.scheduled.queries.create.as.enabled
4. Set the value to true.
5. Save and restart Hive on Tez.

Periodically rebuilding a materialized view

Using materialized views can enhance query performance. You need to update materialized view contents when new data is added to the underlying table. Instead of rebuilding the materialized view manually, you can schedule this task. Automatic rebuilding then occurs periodically and transparently to users.

About this task

This task assumes you created the following schemas for storing employee and departmental information:

```
CREATE TABLE emps (
empid INTEGER,
deptno INTEGER,
name VARCHAR(256),
salary FLOAT,
hire_date TIMESTAMP);

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

Imagine that you add data for a number of employees to the table. Assume many users of your database issue queries to access to data about the employees hired during last year including the department they belong to.

You perform the steps below to create a materialized view of the table to address these queries. Imagine new employees are hired and you add their records to the table. These changes render the materialized view contents outdated. You need to update its contents. You create a scheduled query to perform this task. The scheduled rebuilding will not occur unless there are changes to the input tables. You test the scheduled query by bypassing the schedule and executing the schedule immediately. Finally, you change the schedule to rebuild less often.

Procedure

1. To handle many queries to access recently hired employee and departmental data, create a materialized view.

```
CREATE MATERIALIZED VIEW mv_recently_hired AS
  SELECT empid, name, deptname, hire_date FROM emps
  JOIN depts ON (emps.deptno = depts.deptno)
  WHERE hire_date >= '2020-01-01 00:00:00';
```

2. Use the materialized view by querying the employee data.

```
SELECT empid, name FROM emps
JOIN depts  ON (emps.deptno = depts.deptno)
WHERE hire_date >= '2020-03-01 00:00:00' AND deptname = 'finance';
```

3. Assuming new hiring occurred and you added new records to the emps table, rebuild the materialized view.

```
ALTER MATERIALIZED VIEW mv_recently_hired REBUILD;
```

   The rebuilding updates the contents of the materialized view.

4. Create a scheduled query to invoke the rebuild statement every 10 minutes.

```
CREATE SCHEDULED QUERY scheduled_rebuild
EVERY 10 MINUTES AS
ALTER MATERIALIZED VIEW mv_recently_hired REBUILD;
```

   A rebuild executes every 10 minutes, assuming changes to the emp table occur within that period. If a materialized view can be rebuilt incrementally, the scheduled rebuild does not occur unless there are changes to the input tables.

5. To test the schedule, run a scheduled query immediately.

```
ALTER SCHEDULED QUERY scheduled_rebuild EXECUTE;
```

6. Change the frequency of the rebuilding.

```
ALTER SCHEDULED QUERY scheduled_rebuild EVERY 20 MINUTES;
```

Related Information

Apache Hive Language Manual--Scheduled Queries

Getting scheduled query information and monitor the query

After you create a scheduled query you can access information about it in the scheduled_queries table of the Hive information schema. You can also use the information schema to monitor scheduled query execution.

Procedure

1. Query the information schema to get information about a schedule.

```
SELECT *
FROM information_schema.scheduled_queries
```

```
WHERE schedule_name = 'scheduled_rebuild';
```

The following information appears about the scheduled query:

scheduled_query_id

Unique numeric identifier for a scheduled query.

schedule_name

Name of the scheduled query.

enabled

Whether the scheduled query is currently enabled or not.

cluster_namespace

Namespace that the scheduled query belongs to.

schedule

Schedule described as a Quartz cron expression.

user

Owner of the scheduled query.

query

SQL query to be executed.

next_execution

When the next execution of this scheduled query is due.

2. Monitor the most recent scheduled query execution.

```
SELECT *
FROM information_schema.scheduled_executions;
```

You can configure the retention period for this information in the Hive metastore.

scheduled_execution_id

Unique numeric identifier for a scheduled query execution.

schedule_name

Name of the scheduled query associated with this execution.

executor_query_id

Query ID assigned to the execution by HiveServer (HS2).

state

One of the following phases of execution.

- STARTED. A scheduled query is due and a HiveServer instance has retrieved its information.
- EXECUTING. HiveServer is executing the query and reporting progress in configurable intervals.
- FAILED. The query execution was stopped due to an error or exception.
- FINISHED. The query execution was successful.
- TIMED_OUT. HiveServer did not provide an update on the query status for more than a configurable timeout.

start_time

Start time of execution.

end_time

End time of execution.

elapsed

Difference between start and end time.

error_message

If the scheduled query failed, it contains the error message associated with its failure.

last_update_time

Time of the last update of the query status by HiveServer.

Related Information

Apache Hive Language Manual--Scheduled Queries

Lateral View in Hive

Learn how to use lateral views in Hive to apply table-generating functions and handle empty results.

You can use lateral views in Hive to work with user-defined table-generating functions (UDTFs), such as explode(). Lateral views generate output rows for each input row by applying the UDTF and joining the results with the original input row. Multiple lateral views can be used in a single query, referencing columns from preceding tables. Additionally, you can use the OUTER keyword to ensure rows are generated with NULL values when no output rows are produced.

For more information, see *Hive Lateral View Documentation*.

Related Information

Hive Lateral View Documentation

Materialized views

A materialized view is a Hive-managed database object that holds a query result you can use to speed up the execution of a query workload. If your queries are repetitive, you can reduce latency and resource consumption by using materialized views. You create materialized views to optimize your queries automatically.

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.

You can perform the following materialized view operations:

- 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

Creating and using a materialized view

You can create a materialized view of a query to calculate and store results of an expensive operation, such as a particular join, on a managed, ACID table that you repeatedly run. When you issue queries specified by that materialized view, the optimizer rewrites the query based on it. This action saves reprocessing. Query performance improves.

About this task

In the tasks that follow, you create and populate example tables. The tables are managed tables. You cannot create a materialized view of an external table. You create a materialized view of a join of the tables. Subsequently, you

run a query to join the tables, and the query plan takes advantage of the precomputed join to accelerate processing. These over-simplified tasks show the syntax and output of a materialized view, and do not demonstrate accelerated processing that occurs in a real-world task, processing a large amount of data.

Related Information

Materialized view commands

Creating the tables and view

You see how to create simple tables, insert the data, and join the tables using a materialized view. You run the query, and the optimizer takes advantage of the precomputation performed by the materialized view to speed response time.

Procedure

1. Create two ACID 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);
```

   Tables must be ACID (managed) tables.

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. Run 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       |
+--------+-----------+
```

Verifing use of a query rewrite

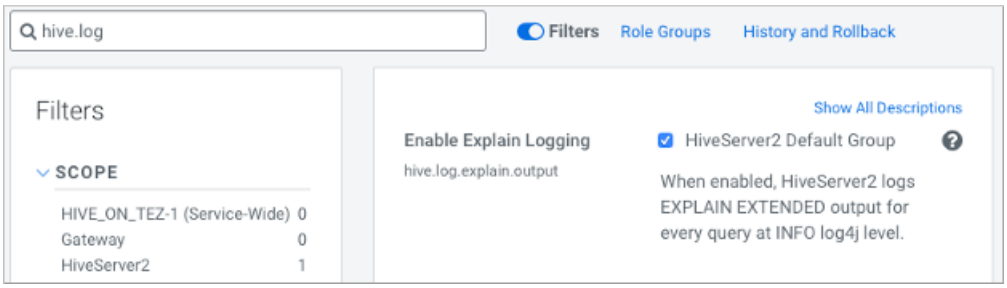
You can use Apache Hive explain logging to check that a materialized view used a query rewrite.

About this task

By default explain logging is set to false in Cloudera.

Procedure

1. In Cloudera Manager, enable explain logging: Navigate to Clusters HIVE_ON_TEZ-1 Configuration , search for hive.log.explain.output, check HiveServer2 Default Group, and click Save Changes.



2. 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: v
archar(256))
                        outputColumnNames: _col0, _col1
                        ListSink
```

Using optimizations from a subquery

You can create a query having a subquery that the optimizer rewrites based on a materialized view. You create a materialized view, and then run a query that uses the materialized view.

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_data that has the following columns:

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

Procedure

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

```
CREATE TABLE flights_data(
  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_data
  GROUP BY dest, origin;
```

3. Take advantage of the materialized view to speed your queries when you have to count destinations and origins again.

   For example, use a subquery to select the number of destination-origin pairs like the materialized view.

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

   Transparently, the SQL engine uses the work already in place since creation of the materialized view instead of reprocessing.

Related Information

Materialized view commands

Dropping 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. You cannot drop a table that has a relationship with a materialized view.

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

Procedure

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

Related Information

Materialized view commands

Showing 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 another_database;

Related Information

Materialized view commands

Describing 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, own
er:hive, createTime:1532466307, lastAccessTime:0, retention:0, sd:Storag
eDescriptor(cols:[FieldSchema(name:empid, type:int, comment:null), Field
Schema(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.hado
op.hive.ql.io.orc.OrcInputFormat, outputFormat:org.apache.hadoop.hive.ql
.io.orc.OrcOutputFormat, compressed:false, numBuckets:-1, serdeInfo:SerD
eInfo(name:null, serializationLib:org.apache.hadoop.hive.ql.io.orc.OrcSe
rde, parameters:{}), bucketCols:[], sortCols:[], parameters:{}, skewedIn
fo:SkewedInfo(skewedColNames:[], skewedColValues:[], skewedColValueLocat
ionMaps:{}), storedAsSubDirectories:false), partitionKeys:[], parameters
:{totalSize=488, numRows=4, rawDataSize=520, COLUMN_STATS_ACCURATE={\"BA
SIC_STATS\":\"true\"}, numFiles=1, transient_lastDdlTime=1532466307, buc
keting_version=2}, viewOriginalText:SELECT empid, deptname, hire_date\nF
ROM 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` >= '20
17-01-17', tableType:MATERIALIZED_VIEW, rewriteEnabled:true, creationMet
adata:CreationMetadata(catName:hive, dbName:default, tblName:mv1, tables
Used:[default.depts, default.emps2], validTxnList:53$default.depts:2:922
3372036854775807::$default.emps2:4:9223372036854775807::, materializatio
nTime: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 2019     ...
| 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

Managing query rewrites

After changes to base tables, the data in a materialized view is stale. You need to know how to prevent the SQL optimizer from rewriting queries in this situation. 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.

About this task

As administrator, you can globally enable or disable all query rewrites based on materialized views. By default, the optimizer rewrites a query based on a materialized view.

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 enable rewriting of queries based on materialized views by setting a global property.

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

Related Information

Materialized view commands

Purposely using a stale materialized view

A rewrite of a query based on a stale materialized view does not occur automatically. If you want a rewrite of a stale or possibly stale materialized view, you can force a rewrite.

About this task

For example, you might want to use the contents of a materialized view of a non-transactional table because the freshness of such a table is unknown. The optimizer cannot determine the data freshness if you use external tables. You can purposely rewrite a query based on a stale materialized views using these techniques:

- Schedule the materialized view for rebuilding. For example, schedule a rebuild to occur every x minutes.
- Adjust the rewriting time window to use stale or possibly stale data for a period of time. For example, schedule the window within which to use stale data for x + y minutes.

Procedure

1. Create a scheduled query to invoke the rebuild statement every 10 minutes.

```
CREATE SCHEDULED QUERY scheduled_rebuild
EVERY 10 MINUTES AS
ALTER MATERIALIZED VIEW mv_recently_hired REBUILD;
```

2. Define the window of time for using stale data.

```
SET hive.materializedview.rewriting.time.window=10min;
```

Creating and using 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
101	Eng	11
200	Sup	20

In this task, you create two materialized views: one partitions data on department; the other 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 < 2
00;
```

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 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 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

Creating and using a materialized view

Materialized view commands

HPL/SQL stored procedures

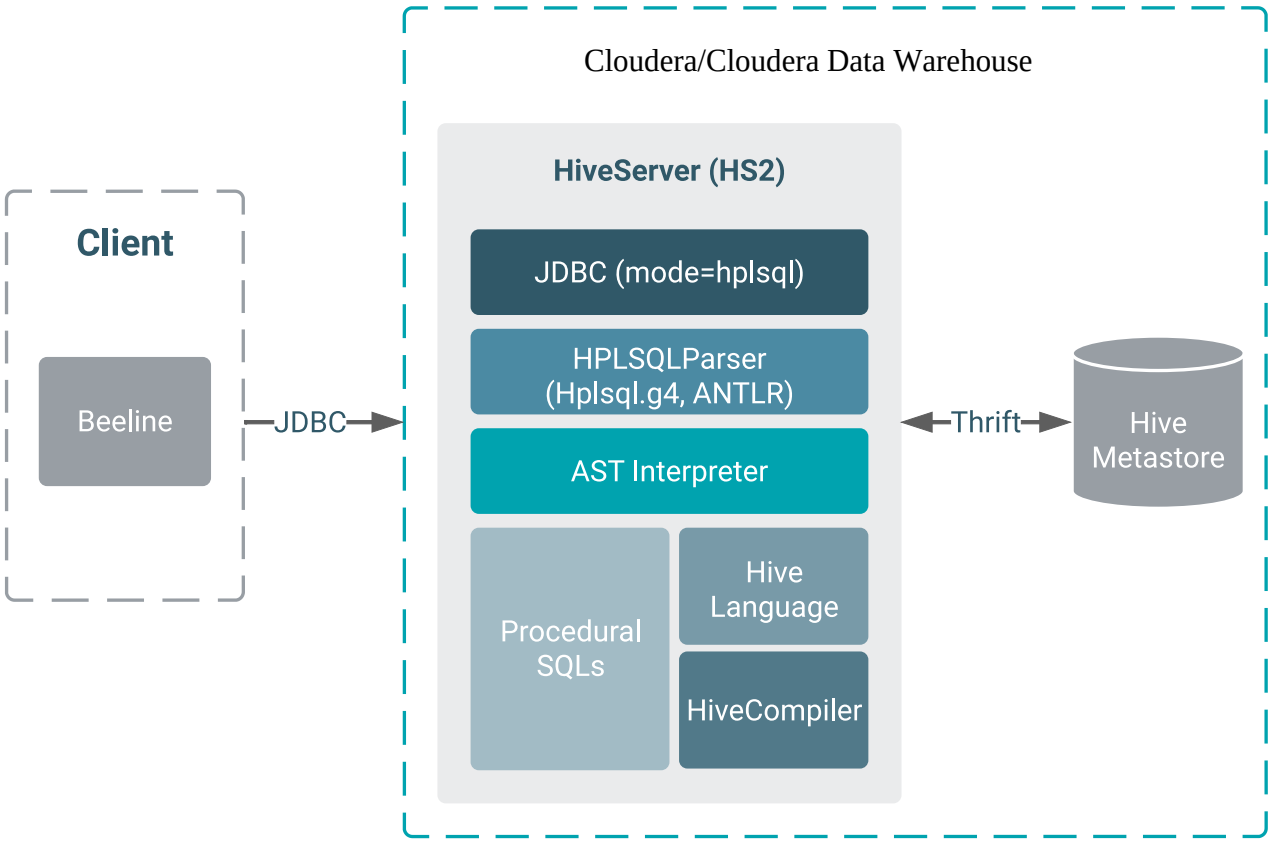
Cloudera supports Hive Hybrid Procedural SQL (HPL/SQL). HPL/SQL is an Apache open source procedural extension for SQL for Hive users.

HPL/SQL includes imperative programming structures (variables, procedures, control flow, and exceptions), and is typically used for ETL. The HPL/SQL language understands the syntax and semantics of most procedural SQL dialects, such as Oracle PL/SQL.

HPL/SQL has its own grammar. For more information about the HPL/SQL language, see the HPL/SQL Reference.

HPL/SQL architecture

HPL/SQL has been re-architected from a command line tool to an integrated part of HiveServer (HS2). From a JDBC client, such as Beeline, you connect to HiveServer in Cloudera on cloud or Cloudera Base on premises. The interpreter executes the abstract syntax tree (AST) from the parser. Hive metastore securely stores the function and procedure code permanently. The procedure is loaded and cached on demand to the interpreter's memory when needed. You can close the session or restart Hive without losing the definitions.



You can enable and use HPL/SQL from any host or third-party tool that can make a JDBC connection to HiveServer. Beeline is a popular client for use with HPL/SQL because other third-party tools do not show you some of the error messages about syntax mistakes.

Enabling HPL/SQL in the beeline connection string

After setting up a client to connect to Hive, you append mode=hplsql to the JDBC URL that connects the client to Hive:

```
beeline -u "jdbc:hive2://<HiveServer host>:10000/default;mode=hplsql"
```

When the client connects to HiveServer configuring this mode, HPL/SQL is enabled; otherwise, HPL/SQL is disabled.

HPL/SQL key features

- Flow of Control Statements (FOR, WHILE, IF, CASE, LOOP, LEAVE, RETURN)
- Functions, procedures, and packages
- Built-in functions (string manipulations, datetime functions, conversions)
- Exception handling and conditions
- Constants and variable, assignment (DECLARE count INT := 1)
- Processing results using a CURSOR
- UDF to run HPL/SQL scripts from Hive queries

  ```
  (SELECT hplsql('mycustomfunc(:1)', name) FROM users;)
  ```
- Bulk data processing with BULK COLLECT statement

HPL/SQL limitations

- Some of the Hive specific CREATE TABLE parameters are missing.
- No colon syntax to parametrize SQL strings.
- No quoted string literals.
- No GOTO and Label.
- EXECUTE does not have output parameters.
- Some complex data types, such as Arrays and Records are not supported.
- No object-oriented extension.
- Data Analytics Studio (DAS) does not support HPL/SQL.

Creating a function

You need to know the syntax of HPL/SQL, which closely resembles Oracle's PL/SQL. An example of creating a function and calling it in a Hive SELECT statement demonstrates the HPL/SQL basics.

Procedure

1. Create a table and populate it with some numbers.

```
create table numbers (n int);

for i in 1..10 loop
  insert into numbers values(i);
end loop;
```

2. Create a function called fizzbuzz to return numbers.

```
create function fizzbuzz(n int) returns string
begin
  if mod(n, 15) == 0 then
     return 'FIZZBUZZ';
  elseif mod(n, 5) == 0 then
     return 'BUZZ';
  elseif mod(n, 3) == 0 then
     return 'FIZZ';
  else
     return n;
  end if;
end;
```

3. Call the function from a Hive select statement.

```
select fizzbuzz(n) from numbers;
```

Output looks something like this:

```
...
1
2
FIZZ
4
BUZZ
1
2
FIZZ
4
BUZZ
FIZZ
7
8
```

```
FIZZ
BUZZ
```

Setting up a Hive client

As a Hive user who connects to HiveServer2 from a client, you need to know how to set up the client for using HPL/SQL.

Procedure

1. On the client end, download the latest version of the Hive JDBC driver from the Cloudera Downloads page.
2. Install the driver on the client end.

   Typically, you add the JAR file to the Libraries folder.
3. In , in Cloudera Manager, click Clusters Hive on Tez to go to the **Hive on Tez** service page.
4. From the **Hive on Tez** service page, click Actions and select Download Client Configuration.
5. Unpack the hive_on_tez-1-clientconfig.zip, open the beeline-site.xml file, and copy the value of beeline.hs2.jdbc .url.HIVE_ON_TEZ-1. This value is the JDBC URL.

   ```
   jdbc:hive2://[***HOST***]:[***PORT***]/default;serviceDiscoveryMode=zooK
   eeper;zooKeeperNamespace=hiveserver2;sslTrustStore=[***TRUSTSTORE-PATH**
   *];trustStorePassword=[***TRUSTSTORE-PASSWORD***]
   ```
6. Paste the copied JDBC URL into a text file.
7. Append mode=hplsql to the end of the JDBC URL for connecting to Hive from the client.

   ```
   jdbc:hive2://[***HOST***]:[***PORT***]/default;serviceDiscoveryMode=zooK
   eeper;zooKeeperNamespace=hiveserver2;sslTrustStore=[***TRUSTSTORE-PATH**
   *];trustStorePassword=[***TRUSTSTORE-PASSWORD***];mode=hplsql
   ```
8. In the BI tool, configure the JDBC connection using the JDBC URL and driver class name, com.cloudera.hive.jd bc.HS2Driver, and use the URL in Beeline to connect to Hive and enable HPL/SQL.

   ```
   beeline -n <csso_username> -p <password> -u
   "jdbc:hive2://[***HOST***]:[***PORT***]/default;serviceDiscoveryMode=zo
   oKeeper;zooKeeperNamespace=hiveserver2;sslTrustStore=[***TRUSTSTORE-PATH
   ***];trustStorePassword=[***TRUSTSTORE-PASSWORD***];mode=hplsql"
   ```

   At the Hive prompt, you can run High-Performance Language for SQL (HPL/SQL) at the Hive prompt. Use the forward slash (/) as a statement delimiter to group multiple HPL/SQL statements across multiple rows. HiveServer2 (HS2) receives all statements present before the forward slash (/) for processing. If any statement raises an error during processing, the processing of remaining statements stops.

   Note: When you submit HPL/SQL statements by using a file, use a forward slash (/) to group the statements. If all statements present in the file must group as one, do not add a forward slash (/)

What to do next

Set hive.security.authorization.sqlstd.confwhitelist.append=QUERY_EXECUTOR in Hive Service Advanced Configuration Snippet (Safety Valve) for hive-site.xml before running HPL/SQL queries.

Using the cursor to return record sets

You use a cursor to return recordsets from stored procedures and functions. A cursor defines a set of rows and row-by-row operations on the set. You need to know how to open a cursor, fetch data into local variables row-by-row, and close a cursor.

About this task

A function called fizzbuzz2 calls another function fizzbuzz, created in the previous topic. The new function declares a cursor of type SYS_REFCURSOR to fetch the contents of the numbers table, also created in the previous topic.

Before you begin

- You created the fizzbuzz function.
- You created the numbers table.

Procedure

Use an OPEN-FOR statement to select the number in each row of the numbers table, iteratively FETCH the numerical content of the each row INTO cursor variable num, close the cursor, and print the results.

```
create function fizzbuzz2() returns string
begin
 declare num int = 0;
 declare result string = '';
 declare curs SYS_REFCURSOR;
 open curs for select n from numbers;
 fetch curs into num;
 while (SQLCODE = 0) do
    result = result || fizzbuzz(num) || ' ';
    fetch curs INTO num;
 end while;
 close curs;
 return result;
end;
print fizzbuzz2();
```

The output looks something like this:

```
...
1 2 FIZZ 4 BUZZ 1 2 FIZZ 4 BUZZ FIZZ 7 8 FIZZ BUZZ
No rows affected
```

Stored procedure examples

You see by example some of the frequently used and useful HPL/SQL code. HPL/SQL is an Apache open source procedural extension for SQL for Hive users.

Greeting function

The following example creates a function that takes the input of your name and returns "hello <name>":

```
CREATE PROCEDURE greet(name STRING)
BEGIN
  PRINT 'Hello ' || name;
END;
```

Pass a cursor to return records

This example of a procedure, called from another procedure, takes a cursor parameter declared OUT; an OPEN-FOR statement opens the cursor and executes the SELECT query. The query returns a subset of the records to the test_even procedure from the numbers table, created in a previous topic.

```
CREATE PROCEDURE even(cur OUT SYS_REFCURSOR)
BEGIN
OPEN cur FOR
```

```
SELECT n FROM numbers
WHERE MOD(n, 2) == 0;
END;
```

The test_even procedure below calls the even procedure above, passing the cursor of type SYS_REFCURSOR to fetch each row containing an even number.

```
CREATE PROCEDURE test_even()
BEGIN
 DECLARE curs SYS_REFCURSOR;
 DECLARE n INT = 0;
 DECLARE result STRING = 'Even numbers are: ';
 even(curs);
 FETCH curs INTO n;
 WHILE (SQLCODE = 0) DO
    result = result || n || ' ';
    FETCH curs INTO n;
 END WHILE;
 CLOSE curs;
 PRINT result;
END;
```

Using table types to index fields in records

You can set up table types and reference fields in records using indexing. This example assumes you created the emp table.

```
-- CREATE TABLE emp (name string, age int);
TYPE emp_type IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
TYPE emp_age_type IS TABLE OF emp.age%TYPE INDEX BY BINARY_INTEGER;
TYPE emp_name_type IS TABLE OF STRING INDEX BY BINARY_INTEGER;

DECLARE rows emp_type;
DECLARE ages emp_age_type;

SELECT * INTO rows(1) FROM emp WHERE name = 'alice';
PRINT 'name=' || rows(1).name || ' age=' || rows(1).age;

SELECT age INTO ages(1) FROM emp WHERE name = 'alice';
PRINT 'age=' || ages(1);
```

BULK COLLECT

Using BULK COLLECT, you can retrieve multiple rows in a single fetch quickly.

```
TYPE emp_type IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
    DECLARE rows emp_type;

    SELECT * BULK COLLECT INTO rows FROM emp;


    DECLARE idx INT = rows.FIRST;
    WHILE idx IS NOT NULL LOOP
    PRINT rows(idx).name || ' = ' || rows(idx).age;
    idx = rows.NEXT(idx);
    END LOOP;
```

Using JdbcStorageHandler to query RDBMS

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

About this task

This task assumes you are a Cloudera Base on premises user. You can 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 on a cluster node or review the existing data in your database.
2. Create an external table using the JdbcStorageHandler and table properties that specify the minimum information which includes database type, driver, database connection string, user name and password for querying Hive, table name, and number of active connections to Hive.
   - Creating an external JDBC table to map a table in a remote MySQL database.

     ```
     CREATE EXTERNAL TABLE mytable_example1(
       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_EXAMPLE1",
       "hive.sql.dbcp.maxActive" = "1"
     );
     ```
   - Creating an external JDBC table to map a table in another Hive cluster.

     ```
     create external table mytable2(
     colm1 string,
     colm2 int,
     colm3 double
     )
     STORED BY 'org.apache.hive.storage.jdbc.JdbcStorageHandler'
     TBLPROPERTIES (
     "hive.sql.database.type" = "HIVE",
     "hive.sql.jdbc.driver" = "org.apache.hive.jdbc.HiveDriver",
     "hive.sal.jdbc.url" = "jdbc:hive2://localhost/sample",
     "hive.sql.dbcp.username" = "hive"
     "hive.sql.dbcp.password" = "hive",
     "hive.sql.table" = "MYTABLE_EXAMPLE2",
     "hive.sql.dbcp.maxActive" = "1"
     );
     ```
3. Query the external table.

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

Related Information

Apache Wiki: JdbcStorageHandler

Using functions

You can call a built-in Hive function to run one of a wide-range of operations instead of performing multiple steps. 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.

Reloading, viewing, and filtering 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 after your session started. 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 the Hive shell.
   Open a client connection, for example:

   ```
   beeline -u jdbc:hive2://mycloudhost-3.com:10000 -n <your user name> -p
   ```

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 or object store. 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 the cluster to your cluster or object store.

  Cloudera Manager, Minimum Required Role: Configurator (also provided by Cluster Administrator, Full Administrator).
- Hive on Tez or Hive LLAP is running on the cluster.
- You have installed Java and a Java integrated development environment (IDE) tool on the machine, or virtual machine, where you will create the UDF.

Setting up the development environment

You can create a Hive UDF in a development environment using IntelliJ, for example, and build the UDF. You define the Cloudera Maven Repository in your POM, which accesses necessary JARS hadoop-common-<version>.jar and hive-exec-<version>.jar.

Procedure

1. 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.
2. Add archetype information.
   For example:
   - GroupId: com.mycompany.hiveudf
   - ArtifactId: hiveudf
3. Click Next and Finish.
   The generated pom.xml appears in sample-hiveudf.
4. To the pom.xml, add properties to facilitate versioning.
   For example:

```
<properties>
   <hadoop.version>TBD</hadoop.version>
   <hive.version>TBD</hive.version>
</properties>
```

5. In the pom.xml, define the repositories.

   Use internal repositories if you do not have internet access.

```
<repositories>
   <repository>
      <releases>
         <enabled>true</enabled>
         <updatePolicy>always</updatePolicy>
         <checksumPolicy>warn</checksumPolicy>
      </releases>
      <snapshots>
         <enabled>false</enabled>
         <updatePolicy>never</updatePolicy>
         <checksumPolicy>fail</checksumPolicy>
      </snapshots>
      <id>HDPReleases</id>
      <name>HDP Releases</name>
      <url>http://repo.hortonworks.com/content/repositories/releases/</u
rl>
      <layout>default</layout>
   </repository>
   <repository>
      <id>public.repo.hortonworks.com</id>
      <name>Public Hortonworks Maven Repo</name>
      <url>http://repo.hortonworks.com/content/groups/public/</url>
      <snapshots>
         <enabled>false</enabled>
      </snapshots>
   </repository>
   <repository>
         <id>repository.cloudera.com</id>
         <url>https://repository.cloudera.com/artifactory/cloudera-repos/<
/url>
   </repository>
 </repositories>
```

6. 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>
```

Creating 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.\
PrimitiveObjectInspectorFactory;
import org.apache.hadoop.io.Text;
public class TypeOf extends GenericUDF {
  private final Text output = new Text();
@Override
  public ObjectInspector initialize(ObjectInspector[] arguments) throws U
DFArgumentException {
    checkArgsSize(arguments, 1, 1);
    checkArgPrimitive(arguments, 0);
    ObjectInspector outputOI = PrimitiveObjectInspectorFactory.writableSt
ringObjectInspector;
    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, ",");
  }
}
```

Building the project and upload the JAR

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

About this task

Cloudera Base on premises

Use one of these methods to configure the cluster to find the JAR:

- Direct reference

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

  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.

Cloudera on cloud

Use the Direct reference method only.

Procedure

1. Build the IntelliJ project.

```
...
[INFO] Building jar: /Users/max/IdeaProjects/hiveudf/target/TypeOf-1.0-S
NAPSHOT.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. In IntelliJ, 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 HDFS (Cloudera Base on premises) or S3 (Public Cloud).
     b. Move the JAR into the Hive warehouse. For example, in Cloudera Base on premises:

```
$ hdfs dfs -put TypeOf-1.0-SNAPSHOT.jar /warehouse/tablespace/manage
d/hiveudf-1.0-SNAPSHOT.jar
```

   - Hive aux JARs path (Cloudera Base on premises only)
     a. In Cloudera Base on premises, click Clusters and select the Hive service, for example, HIVE. Click Configuration and search for Hive Auxiliary JARs Directory.
     b. Specify a directory value for the Hive aux JARs property if necessary, or make a note of the path.
     c. Upload the JAR to the specified directory on all HiveServer instances (and all Metastore instances, if separate).

- Reloadable aux JAR (Cloudera Base on premises only)
  a. Upload the JAR to the /hadoop/hive-udf-dyn directory on all HiveServer instances (and all Metastore instances, if separate). An HDFS location is not supported.
  b. In hive-site.xml, set the following property: hive.reloadable.aux.jars.path=/hadoop/hive-udf-dyn.

4. In IntelliJ, click Save.
5. Click Actions Deploy Client Configuration .
6. Restart the Hive service. For example, restart HIVE.

Registering 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 the Hive aux library directory method that involves a symbolic link, you need to restart HiveServer after registration. If you use the Direct JAR reference or Reloadable aux JAR methods, 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 as a user who has UDF access.

   HiveServer, for example:

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

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';
     ```

   - Hive aux library directory (Cloudera Base on premises only):
     a. Set up a symbolic link on the command line of the local file system.

        ```
        ln -s /local-apps/hive-udf-aux <path to hive parcel>/hive/auxlib
        ```

     b. In Beeline, run the command to register the UDF.

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

     c. Restart HiveServer.
   - Reloadable aux JAR (Cloudera Base on premises only):

     ```
     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.

Calling 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 or Hive aux library directory

     For example:

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

   - Reloadable aux JAR

     For example:

```
RELOAD;
SELECT students.name, udftypeof(students.name) AS type FROM students WHE
RE 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.HiveVarcharWri
table |
```

```
+------------------+-----------------------------------------------------
--------+
```