Apache Hive
Data Warehouse Framework

Bin Jiang

02/25/2017

Objective

This section introduces the Apache Hive data warehouse framework where participants will learn about:

- Main Hive features
- Hive Architecture
- HiveQL Basics

What is Hive

- Hive is a Hadoop-based data warehouse framework/infrastructure written in Java
- Works with large datasets stored in Hadoop's HDFS and compatible file systems such as Amazon S3 file system
- Originally developed by Facebook to cope with the growing volumes of data in their data warehouse that they were not able to solve using traditional technologies

Hive's Value Proposition

- Hive shields non-programmers (business analysts, etc.) from programming MapReduce jobs with its own SQL-like query language called HiveQL
- Internally HiveQL queries are converted into actual MapReduce jobs submitted to Hadoop for execution against files stored on HDFS
- In YARN, in addition to MapReduce, Hive also supports Apache Tex and Apache Spark execution engines (MR is used by Hive by default)
- HiveQL is extendable via new types, user functions and scripts
- Custom map and reduce modules can be plugged in HiveQL for fine-tuned processing logic

Hive's Value Proposition

In addition to MapReduce, the newer versions of Hive can leverage alternative execution engines, such as Apache Spark and Apache Tez. Switching between execution engines is as simple as setting Hive's hive.execution.engine system property to the target engine string value in your Hive shell session. For example:

```
set hive.execution.engine = tez;
set hive.execution.engine = spark;
```

The default value for this configuration is "mr"

Who uses Hive

- Facebook
- Netflix
- AWS

Hive's Main Sub-Systems

- Hive consists of three main sub-systems:
  - Metadata store (metastore) for schema information (table column data types, related files location etc.) is stored in an embedded or external database
  - Another Hive's sub-system called HCatalog sits on top of the metastore and provides access to the metastore by other Hadoop-centric products, such as Pig and MapReduce jobs
  - HCatalog provides RESTful interface to the metastore through its web server called WebHCat

Hive's Main Sub-Systems

- Serization/deserialization framework for reading and writing data
- Query processor that translates HiveQL statements into MapReduce instructions

Hive Features

- Support for four file formats: TEXTFILE, SEQUENCEFILE, RCFILE, PARQUET and ORC
- Bitmap indexing with other index types
- Metadata storage in an internal (embedded) or external RDBMS
- Ability to work on data compressed with gzip, bzip2 and other algorithm
- Support for User-Defined Functions
- SQL-like query language (HiveQL)

Hive Features

- Support for JDBC, ODBC and Thrift clients
- Hive lends itself to integration with business intelligence and visualization tools such as Microstrategy, Tableau, Revolutions Analytics etc. In many cases, integration is done using Hive's high performance ODBC driver which supports all SQL-92 interfaces

The "Classic" Hive Architecture

- CLI -> Driver -> HADOOP
- JDBC/ODBC -> Thrift Server -> Driver -> HADOOP
- Web GUI -> Driver -> HADOOP

The New Hive Architecture

- Hive architecture was redesigned as a client/server system to enable various remote clients to execute queries against the Hive Server, referred to as HiveServer2
- The new architecture provides better support for multi-client concurrency and authentication
- In addition to the original Hive CLI, Hive 2 also supports the new CLI, called Beeline and JDBC, ODBC clients

HiveQL

- SQL-like query language developed to hide complexities of manually coding MapReduce jobs
- HiveQL does not fully implement SQL-92 standard
- It has Hive-specific syntactic extensions
- Hive is suited for batch processing over large sets of immutable data (e.g. web logs)
- UPDATE or DELETE operations are not supported; but INSERT INTO is acceptable
- HiveQL does not support transactions
- Full ACID transactional support is planned for future releases
- Hive often works more efficiently on denormalized data
- Generally, HiveQL statements are case-intensiive

Where are the Hive Tables Located?

- Location of the tables that you create in Hive is specified in the hive.metastore.warehouse.dir property of the hive-site.xml configuration file:

```
<property>
        <name>hive.metastore.warehouse.dir</name>
        <value>/user/hive/warehouse</value>
</property>
```

- The value of the property points to the directory on HDFS
- For the above value, you can verify the existence of the warehouse directory by running the following command:

```
hadoop fs -ls /user/hive/warehouse
```

Where are the Hive Tables Located?

Setting up Hive, among other steps, requires the following steps to be performed:

- Have Hadoop in your path or issue the following command:
- export HADOOP_HOME=<hadoop-install-dir>
- You need to create the /user/hive/warehouse and /tmp folders
- Then, you need set apply the chmod g+w HDFS command to give write permissions to the user in the working groups so that that user can create tables in Hive.

Command to perform this setup are:

- $HADOOP_HOME/bin/hadoop fs -mkdir /tmp
- $HADOOP_HOME/bin/hadoop fs -mkdir /user/hive/warehouse
- $HADOOP_HOME/bin/hadoop fs -chmod g+w /tmp
- $HADOOP_HOME/bin/hadoop fs -chmod g+w /user/hive/warehouse

Add Hive Env to the system

Add Hive Env to the system path by opening /etc/profile or ~/.bashrc

Enable the settings immediately:
$source /etc/profile

Change temporary data file path

Modify the configuration file at $HIVE_HOME/conf/hive-site.xml

hive.exec.scratchdir: This is the temporary data file path. By default it is /tmp/hive-${user.name}.

Change Hive Metadata Store

By default, Hive uses the MySQL database as the metadatastore. Hive can also use other databases, such as PostgreSQL as the metadata store.

To configure Hive to use other databases, the following parameters should be configured:

- javax.jdo.option.ConnectionURL // the database URL
- javax.jdo.option.ConnectionDriverName // the JDBC driver name
- javax.jdo.option.ConnectionUserName // database username
- javax.jdo.option.ConnectionPassword // database password

Change Hive Metadata Store

The following is an example setting using MySQL as the metastore database:

```
<property>
        <name>javax.jdo.option.ConnectionURL</name>
        <value>jdbc:mysql://sandbox.hortonworks.com/hive?createDatabaseIfNotExist=true</value>
</property>
<property>
        <name>javax.jdo.option.ConnectionDriverName</name>
        <value>com.mysql.jdbc.Driver</value>
</property>
<property>
        <name>javax.jdo.option.ConnectionUserName</name>
        <value>hive</value>
</property>
<property>
        <name>javax.jdo.option.ConnectionPassword</name>
        <value>hive</value>
</property>
```

"Legacy" Hive Command-line Interface (CLI)

Hive offers command-line interface (CLI) through its hive utility that can be used in two modes of operations:

- An interactive mode, where the user enters HiveQL-based queries manually
- In unattended (batch) mode, where the hive utility takes commands on the command-line as arguments

The Beeline Command Shell

- HiveServer2 comes with its own CLI called Beeline
- Beeline is a JDBC client that is based on the SQLLine Client
- The Beeline shell supports two operational modes: embedded and remote. In embedded mode, Beeline runs an embedded Hive (similar to Hive CLI).In remote mode, it connects to a separate HiveServer2 process over Thrift
- You can run Hive CLI commands from Beeline

The Comparison of Beeline and CLI Command-line Syntax

Purpose	HiveServer2 Beeline	HiveServer1 CLI
Server connection	beeline -u <jdbcurl> -n <username> -p <password>	hive -h <hostname> -p <port>
Help	beeline -h or beeline --help	hive -H
Run query	beeline -e <query in quotes>	hive -e <query in quotes>
	beeline -f <query file name>	hive -f <query file name>
Define variable	beeline --hivevar key=value.	hive --hivevar key=value
Enter mode beeline hive	Connect !connect <jdbcurl>	n/a
List tables	!table	show tables;
List columns	!column <table_name>	desc <table_name>;
Run query	<HQL query>;	<HQL query>;
Save result set	!record <file_name>	N/A
	!record	
Run shell CMD	!sh ls	!ls;
Run dfs CMD	dfs -ls	dfs -ls;
Run file of SQL	!run <file_name>	source <file_name>;
Check Hive version	!dbinfo !hive	--version;
Quit mode	!quit	quit;

The Hive-integrated Development Environment

Besides the command-line interface, there are a few integrated development environment (IDE) tools available for Hive development. One of the best is Oracle SQL Developer, which leverages the powerful functionalities of Oracle IDE and is totally free to use. If we have to use Oracle along with Hive in a project, it is quite convenient to switch between them only from the same IDE.

Oracle SQL developer has supported Hive since version 4.0.3. Configuring it to work with Hive is quite straightforward.

The Hive-integrated Development Environment

The following are a few steps to configure the IDE to connect to Hive:

- Download Hive JDBC drivers from the vendor website, such as Cloudera.
- Unzip the JDBC version 4 driver to a local directory.
- Start Oracle SQL Developer and navigate to Preferences | Database | Third Party JDBC Drivers.
- Add all of the JAR files contained in the unzipped directory to the Third-party JDBC Driver Path setting as follows:
- Click on the OK button and restart Oracle SQL Developer.
- Create new connections in the Hive tab giving a proper Connection Name, Username, Password, Host name (Hive server hostname), Port, and Database.
- Click on the Add and Connect buttons to connect to Hive.

Summary

- Hive is a data warehouse framework/infrastructure written in Java that runs on top of Hadoop
- Hive offers users an SQL-like query language called HiveQL which shields users from complexities of programming MapReduce jobs directly
- Hive also offers a CLI through its hive utility that can be used in two modes of operations (Interactive and batch)
- Beeline, the CLI for Hive 2, provides support for Hive Commands
- We also looked into a few of the Hive interactive commands and queries in Hive CLI, Beeline, and IDEs. After going through this chapter, we should be able to set up our own Hive environment locally and use Hive from CLI or IDE tools.

Objective

This section introduces the Hive command-line interface that allows users to execute HiveQL scripts in two modes:

- Interactive mode
- Unattended (batch) mode

Hive Command-line Interface (CLI)

- Hive CLI is based on the command-line hive shell utility which can be used to execute HiveQL commands in either interactive or unattended (batch) mode:
  - In interactive mode, the user enters HiveQL-based queries manually, submitting commands sequentially
  - In unattended (batch) mode, the hive shell takes command parameters on the command-line
- Hive CLI also supports variable substitution that helps create dynamic scripts

The Hive Interactive Shell

- You start the interactive shell by running the hive command to suppress information messages printed while you are in the shell, start the shell with the -S (silent) flag:

  hive –S

- After successful initialization, you will get the hive> command prompt

- To end the shell session, enter quit; or exit; at the command prompt. Don't forget a semi-colon; at the end of each command

- Command syntax of the Hive shell was influenced by MySQL command-line, e.g.

  SHOW tables; DESCRIBE sample_07;

Running Host OS Commands from the Hive Shell

- 17.1 The Hive shell supports execution of Host OS (operation system)
- 17.2 OS commands must be prefixed with "!" and terminated with ";"e.g. to print the current working directory, issue the following command:

```
hive>!pwd;
```

Interfacing with HDFS from the Hive Shell

- Use the dfs command while in the shell to get access to the HDFS APU
- For example,
  - To get the listing of files in the user home directory in HDFS, issue this command:

    hive>dfs -ls;

  - To remove a file HDFS skipping HDFS's Trash bin;

    hive>dfs -rm -skipTrash myDS;
- Note:

  The dfs command returns a reference to the HDFS command-line interface so to get help on dfs commands, run the following command from the host OS terminal (not from the Hive shell):

  hadoop fs -help

The Hive in Unattended Mode

- In additional to the interactive shell interface, Hive supports invocation of commands in unattended mode
- Commands to execute on command-line are preceded by the '-e' flag
- This mode is suitable for executing short commands that can be issued directly against Hive, e.g.

```
$hive -e 'SHOW TABLES;'
```

- You can submit more than one command, just use the ";" command separator, e.g.:

```
$hive -e 'SHOW TABLES;DESCRIBE sample_07;'
```

- To suppress information messages, use the -S (silent) flag
- To get the CLI help, run the hive -H command

The Hive CLI Integration with the OS Shell

- The Hive CLI can be easily integrated with the underlying OS shell from which the hive utility is launched, e.g.

  ```
  $hive -S -e 'SELECT * FROM sample_07 LIMIT 3;' > /tmp/sample_07.dat
  ```

  The output of the above command is redirected into the

  /tmp/sample_07.dat file on the file system

- More sophisticated command chains can be built using Unix command pipelines

Executing HiveQL Scripts

- Hive can take scripts that contain one or more HiveQL commands for execution in non-interactive (a.k.a unattended or batch) mode, e.g.: Use any text edit tools to add the following statement to myscript.hql: select * from sample_07;

  e.g. vi myscript.hql
  $hive -f myscript.hql

- The HiveQL scripts may have any extensions; by convention, the .hql extension is used

- To execute a script file from inside the Hive shell, use the source command
  hive>source /root/myscript.hql;

Comments in Hive Scripts

- When you execute a file script using the command line, e.g. hive -f myscript.hql, the script may have comments which are denoted as '--' in front of the comment line, e.g. -- Monthly report data generator

---

- Comments don't work inside the interactive Hive shell

Variables and Properties in Hive CLI

- Using command-line, you can define custom variable (a.k.a properties) that you can later use in your Hive scripts using the ${varname} syntax Hive replaces references to variables with their values and then submits the query
- This facility aids in creating dynamic scripts

Setting Properties in CLI

- For setting properties when executing commands in unattended mode, you have two functionally equivalent operations:

  --define key1=value1
  --hivevar ley1=value1

- The above assignments add the key1=value1 key-value pair in the hivevar namespace for the duration of the script execution session

Example of Setting Properties in CLI

The command below shows how to set a property (variable) on command-line:

```
hive -S -e 'DESCRIBE ${tblName};' --define tblName=sample_07
```

The above command will execute the command in silent mode (-S) We define the variable tblName as having the value of foo The Hive shell will substitute the ${tbName} parameter with its value (sample_07) and then execute the command as 'DESCRIBE sample_07;' which will show the structure of the table sample_07

Hive Namespaces

- Hive namespaces help group variables into buckets of functionally similar properties
- Hive supports four namespaces (access types are: R for read-only and W for Write-enabled)

Namespaces	Access Type	Description
hivevar	R/W	User-defined variables (properties)
hiveconf	R/W	Hve configuration properties
system	R/W	Java system properties
env	R/W	Shell environment variables

Using the SET Command

- In the Hive shell, you read/write variables using the SET command
- Except for the user-defined hivevar and hiveconf namespaces, you need to prefix the variable with namespace it belongs to
- For example:
  - To read the OS user's home directory (which is a property of the env namespace)

    hive> SET env:HOME;
  - Using the SET command without the argument, will print all available variables in all the namespaces:env, hivevar, hiveconf and system

    hive>SET;
- Note: To get extensive information on the Hadoop configuration, use the SET -v; command

Setting Properties in the Shell

- To set the value of a writable (settable) property in the interactive shell, use the following command:

  SET [namespace:]myVar=myVarValue;

  The above command is also used to create the variable if it does not existence

- Inside the interactive Hive shell, the hivevar and hiveconf namespaces can be omitted

- You can verify the new property's value by running the following command:

  SET [namespace:]myVar;

- You can reset the configuration to the default values by using the reset command

Setting Properties for the New Shell Session

One of the useful features of the property facility is the ability to set properties before invoking the interactive shell

```
$hive --define var1=val1 --define var2=val2
```

The above command will set two session-side properties var1 and var2 that can be used in the launched Hive interactive shell session

```
hive>set var1;
hive>set var2;
```

Setting Alternative Hive Execution Engines

- By default, Hive uses the MR engine
- You can configure to run Hive on alternative execution engines:
  Apache Tez: set hive.execution.engine=tex;
  Apache Spark: set hive.execution.engine=spark;
- The default value for this configuration is set as follows:
  set hive.execution.engine = mr;

Summary

The Hive CLI is build around the Hive shell utility which allow users to run commands in two modes:

- Interactive mode, where users enter HiveQL-based queries manually inside the shell
- Unattended (batch) mode, where the Hive shell is invoked from the host OS command-line and executes commands passed on as arguments

Objective

This section provides an extended overview Hive's Data Definition Language (DDL), participants will learn about:

- Creating and Dropping Hive Databases
- Creating and Dropping Tables in Hive
- Supported Data Type Categories
- Table Partitioning
- The EXTERNAL Keyword
- Hive Views

Hive Data Definition Language

The following commands are supported by the Hive Data Definition Language (DDL)

- Create/Drop/Alter Database
- Create/Drop/Truncate Table
- Alter Table/Partition/Column
- Create/Drop/Alter View
- Create/Drop Index
- Show
- Describe

Creating Databases in Hive

- When you start working with Hive, you are provided with the default database which is sufficient in most cases
- To create a new database in Hive, issue the following command:

```
CREATE (DATABASE|SCHEMA) [IF NOT EXIST] database_name [COMMENT database_comment]
[LOCATION hdfs_path]
```

- Note: The SCHEMA and DATABASE terms are used interchangeably

Using Databases

- Create the database without checking whether the database already exists:

```
CREATE DATABASE myhivedatabase;
```

- Create the database and check whether the database already exists:

```
CREATE DATABASE IF NOT EXISTS myhivedatabase;
```

- Create the database with location, comments, and metadata information:

```
CREATE DATABASE IF NOT EXISTS myhivedatabase
COMMENT 'my hive database'
LOCATION '/user/root/hivedatabase'
WITH DBPROPERTIES ('creator'='tester','date'='2016-10-05');
```

- To show available databases on the system, issue this command in the Hive shell:

```
hive>SHOW DATABASES;
```

Using Databases

- The USE command sets the current working database, e.g:

```
hive>USE default;
```

- To drop a database, issue this command:

```
DROP (DATABASE|SCHEMA)[IF EXISTS] database_name
```

- Note that Hive keeps the database and the table in directory mode. In order to remove the parent directory, we need to remove the subdirectories first. By default, the database cannot be dropped if it is not empty, unless CASCADE is specified. CASCADE drops the tables in the database automatically before dropping the database.

```
DROP DATABASE IF EXISTS myhivedatabase CASCADE;
```

Creating Tables in Hive

- 36.1 To create a table, use the following (simplified) table creation statement:

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name [(col_name data_type [commnt col_comment], ...)]
[COMMENT table_comment]
[PARTITIONED BY (col_name data_type [COMMENT col_comment],...)]
[ROW FORMAT row_format]
DELIMITED [FIELDS TERMINATED BY char] [COLLECTION ITEMS TERMINATED BY char] [MAP KEYS TERMINATED BY char]
[STORED AS file_format]
[LOATION hdfs_path]
```

Note: The CREATE TABLE statement supports inserting string comments for documenting purposes

Creating Tables in Hive

- Before creating a new table, you can check the list of existing tables by issuing the following command:

```
SHOW TABLES;
```

- Hive stores the table data in a directory of the directory defined by hive.metastore.warehouse.dir property of the hive-site.xml configuration file (the default value is /user/hive/warehouse)

- Check the table

```
Describe employee;
Describe extended employee;
```

Supported Data Type Categories

- The following data type categories are supported in Hive DDL:
  - Primitive types
  - Array type (holds elements of the same data type)
  - Map type (holds maps of data types as key-values pairs)
  - Struct (holds a C-like structure of grouped elements)
  - Union (C-like union types)

Common Numeric Types

The following numeric types are supported:

TINYINT	1 byte signed integer
SMALLINT	2 byte signed integer
INT	4 byte signed integer
BIGINT	8 byte signed integer
BOOLEAN	{true|false}
FLOAT	4-byte single precision floating point number
DOUBLE	8-byte double precision floating point number
DECIMAL	

String and Date/Time Types

- 39.1 STRING: String literals can be used with either single quotes (') or double quotes ("). C-style escaping within the strings (e.g. '\t') is supported
- 39.2 VARCHAR: Created with a length specifier (between 1 and 65355)
- 39.3 CHAR: Similar to VARCHAR but fixed-length (1 to 255); values shorter than the specified length value are padded with spaces
- 39.4 TIMESTAMP: Traditional Unix timestamp with optional nanosecond precision timestamp text should be in the YYYY-MM-DD HH:MM:SS[.fffffffff] format
- 39.5 Date: In the YYYY-MM-DD format without the time of day participants
- Note: TIMESTAMP supported conversion:
  - Integer numeric types: Interpreted as UNIX timestamp in seconds
  - Floating point numeric types: Interpreted as UNIX timestamp in seconds with decimal precision
  - Strings: JDBC compliant java.sql.Timestamp format YYYY-MM-DD HH:MM:SS[.fffffffff]

Miscellaneous Types

- BOOLEAN
- BINARY

Example of the CREATE TABLE Statement

- Prepare the data as follows:
    - Use any text editor to create the file employee.txt and add the following data into employee.txt file:

Michael|Montreal,Toronto|Male,30|DB:80|Product:Developer^DLead
Will|Montreal|Male,35|Perl:85|Product:Lead,Test:Lead
Shelley|New York|Female,27|Python:80|Test:Lead,COE:Architect
Lucy|Vancouver|Female,57|Sales:89,HR:94|Sales:Lead

    - Note:The ^D should be input lie the following: Press ALT and Hold ALT, then press 004

Example of the CREATE TABLE Statement

```
CREATE TABLE employee(
name string                              COMMENT 'name',
work_place ARRAY<string>                 COMMENT 'Work Place',
gender_age STRUCT<gender:string,age:int> COMMENT 'Gender and Age',
skills_score MAP<string,int>             COMMENT 'Skill Score',
depart_title MAP<string,ARRAY<string>>   COMMENT 'Department and Title')
ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'
COLLECTION ITEMS TERMINATED BY ','
MAP KEYS TERMINATED BY ':';
```

Example of the CREATE TABLE Statement

- DESCRIBE employee;
- LOAD DATA LOCAL INPATH '/training/apps/hive/dataset/employee.txt' OVERWRITE INTO TABLE employee;
-
- SELECT * FROM employee;
-
- Hive uses Java generics' syntax (<DATA_TYPE> for specifying the type of data held in ARRAYs and MAPs)

Example of the CREATE TABLE Statement

```
CREATE TABLE employee_id (
        name string,
        employee_id int,
        work_place ARRAY<string>,
        sex_age STRUCT<sex:string,age:int>,
        skills_score MAP<string,int>,
        depart_title MAP<STRING,ARRAY<STRING>>)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
COLLECTION ITEMS TERMINATED BY ','
MAP KEYS TERMINATED BY ':';
```

Example of the CREATE TABLE Statement

- LOAD DATA LOCAL INPATH '/training/apps/hive/dataset/employee_id.txt' OVERWRITE INTO TABLE employee_id;

- CREATE TABLE IF NOT EXISTS employee_hr(
  name string,
  employee_id int,
  sin_number string,
  start_date date)
  ROW FORMAT DELIMITED
  FIELDS TERMINATED BY '|'
  STORED AS TEXTFILE;

- LOAD DATA LOCAL INPATH '/training/apps/hive/dataset/employee_hr.txt' OVERWRITE INTO TABLE employee_hr;

Working with Complex Types

- The COLLECTION ITEMS TERMINATED BY char clause defines the delimiter character, e.g. ','
- The collection items can be struct members, array elements, and map key-value pairs
- For example, with a ',' used as the delimiting char:

  gender_age STRUCT<gender:string,age:int> will require that the underlying rows will have its elements grouped as follows:

  Male,30
  Female,27

Working with Complex Types

- You access struct elements as follows:
  SELECT gender_age.gender, gender_age.age from employee;

- Query the whole struct and each struct column in the table:
  SELECT gender_age FROM employee;

- work_place ARRAY<INT> will require elements of the array be packed as follows:
  Montreal,Toronto

- To read the first element (Montreal) of the commands array in all rows:
  SELECT work_place[0] FROM employee;

- Query the whole array and each array column in the table:
  SELECT work_place FROM employee;

  SELECT work_place[0] AS col_1, work_place[1] AS col_2, work_place[2] AS col_3 FROM employee;

Working with Complex Types

- MAP KEYS TERMINATED BY ':' would require you separate keys from values with a ':'
- The COLLECTION ITEMS TERMINATED BY char clause controls the key-value pair separator
- The DELIMITED FIELDS TERMINATED BY char clause controls the field separator
- The decisions MAP<STRING, INT> definition would require the input file elements representing the map be grouped as follows:

  Python:80
  Test:Lead,COE:Architect

Working with Complex Types

- In order to read values keyed, for example, by "Python" from the decision map:

  SELECT skills_score["Python"] FROM employee;

- Query the whole map and each map column in the table:

  SELECT skills_score FROM employee;

  SELECT name, skills_score['DB'] AS DB, skills_score['Perl'] AS Perl, skills_score['Python'] AS Python, skills_score['Sales'] as Sales, skills_score['HR'] as HR FROM employee;

Working with Complex Types

- HIVE nested ARRAY in MAP data type

Query the composite type in the table:

```
SELECT depart_title FROM employee;

SELECT name, depart_title['Product'] AS Product,
depart_title['Test'] AS Test, depart_title['COE'] AS COE,
depart_title['Sales'] AS Sales FROM employee;

SELECT name, depart_title['Product'][0] AS product_col0,
depart_title['Test'][0] AS test_col0 FROM employee;
```

Data type conversions

- Hive supports both implicit type conversion and explicit type conversion
- Primitive type conversion from a narrow to a wider type is known as implicit conversion. However, the reverse conversion is not allowed. All the integral numeric types, FLOAT, and STRING can be implicitly converted to DOUBLE, and TINYINT, SMALLINT, and INT can all be converted to FLOAT. BOOLEAN types cannot be converted to any other type.
- Explicit type conversion is using the CAST function with the CAST(value AS TYPE) syntax.
- For example, CAST('100' AS INT) will convert the string 100 to the integer value 100. If the cast fails, such as CAST('INT' AS INT), the function returns NULL. In addition, the BINARY type can only cast to STRING, then cast from STRING to other types, if needed.

```
SELECT CAST(gender_age.age AS SMALLINT) AS age FROM employee;
```

Table Partitioning

- Table partitioning significantly improves query performance by having Hive store table data in physically isolated directories and files
- Partitioning is done on partitioning pseudo-column(s) that you specify in the PARTITIONED BY clause PARTITIONED BY (make STRING, year SMALLINT);
- Partitioning columns usually used in the WHERE clause of your HiveQL queries, e.g.

  SELECT * FROM CARS WHERE make='Ford' AND year=2014;
- The partitioning columns must not be repeated in the table definition itself, If they do, you will get this error: "FAILED:Error in semantic analysis:Column repeated in partitioning columns"

Table Partitioning

- 45.1 Table partitioning changes Hive table table directory structure on the data storage
- 45.2 The table directory will have new sub-directories that reflect the sequence of columns in the PARTITIONED BY clause as well as discrete values in the partitioning columns, e.g.:

```
CREATE TABLE CAR (
PARTITIONED BY (make STRING));
```

clause will create a bunch of .../CAR/make=<car_make> directories, e.g.

.../CAR/make='Acura'
.../CAR/make='Ford'
.../CAR/make='Nissan'

Table Partitioning

- Note: The specific make=<CAR_MAKE> directory now must have all the data that belongs to the particular CAR_MAKE; The make column is not repeated in the table definition
- Partitioning makes data storage more efficient

Table Partitioning on Multiple Columns

- Partitioning a table on more than once column will create a hierarchy of sub-directories that follows the sequence of columns in the PARTITION BY clause

```
CREATE TABLE CAR(
...
PARTITIONED BY (make STRING, year SMALLINT);
clause will create a bunch of
        .../CAR/make=<CAR)MAKE_VALUE>/year=<YEAR>
sub-directories of the CAR table, e.g.
        .../CAR/make='Acura'/year=2012/
        .../CAR/make='Acura'/year=2013/
        .../CAR/make='Acura'/year=2014/
        .../CAR/make='Ford'/year=2012/
        .../CAR/make='Ford'/year=2013/
        .../CAR/make='Ford'/year=2014/
        .../CAR/make='Nissan'/year=2012/
```

Viewing Table Partitions

47.1 The following command lists the partitions defined on a table:

SHOW PARTITIONS <table_name>

47.2 You can narrow down a list of returned partitions defined on a table, e.g.

SHOW PARTITIONS car PARTITION (make='Ford');

make='Ford'/year=2012
make='Ford'/year=2013
make='Ford'/year=2014

Bucket Table

Similar to partitioning, a bucket table organizes data into separate files in the HDFS. Bucketing can speed up the data sampling in Hive with sampling on buckets. Bucketing can also improve the join performance if the join keys are also bucket keys because bucketing ensures that the key is present in a certain bucket.

Examples of Bucket Table

- Prepare data for backet tables

```
CREATE TABLE employee_id(
  name string,
  employee_id int,
  work_place ARRAY<string>,
  sex_age STRUCT<sex:string,age:int>,
  skills_score MAP<string,int>,
  depart_title MAP<STRING,ARRAY<STRING>>
)ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
COLLECTION ITEMS TERMINATED BY ','
MAP KEYS TERMINATED BY ':';

LOAD DATA LOCAL INPATH '/training/apps/hive/dataset/employee_id.txt'
OVERWRITE INTO TABLE employee_id;
```

Examples of Bucket Table

- Create the bucket table

```
CREATE TABLE employee_id_buckets(
  name string,
  employee_id int,
  work_place ARRAY<string>,
  sex_age STRUCT<sex:string,age:int>,
  skills_score MAP<string,int>,
  depart_title MAP<STRING,ARRAY<STRING>>
)CLUSTERED BY (employee_id) INTO 2 BUCKETS
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
COLLECTION ITEMS TERMINATED BY ','
MAP KEYS TERMINATED BY ':';

INSERT OVERWRITE TABLE employee_id_buckets SELECT * FROM employee_id;
```

Row Format

- Declared by the [ROW FORMAT row_format] DDL clause
- If the ROW FORMAT section not specified, the default ROW FORMAT DELIMITED .. format is used by Hive
- The following most common formats are supported:
  - DELIMITED FIELDS TERMINATED BY <char>, e.g.
    ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
  - SERDE serde_name [WITH SERDEPROPERTIES (property_name=property_value, property_name=property_value,...)]

Data Serializers/Deserializers

- Hive supports reading/writing data using specific data serializers/deserializers (The SERDE ... DDL fragment)

- Reading/Writing of data using Serde occurs as follows
  HDFS files --> Deserializer --> Row Object in Hive
  Row object in Hive --> Serializer --> HDFS files

- Hive has the following built-in SerDe protocols
  Avro
  ORC
  RegEx
  Thrift
- There exist external Serdes libraries, e.g. for handling JSON files

- Users can write custom Serdes for their own data formats

- Note: AvroSerde takes care of creating the appropriate Avro schema from the Hive table schema

File Format Storage

- Declared with the [STORED AS file_format] DDL clause
- The following file_format options are supported:
  - SEQUENCEFILE
  - TEXTFILE
  - RCFILE
  - ORC
  - PARQUET
  - AVRO

File Compression

- Text: 585 GB
- RCFile: 505 GB
- Parquet: 221GB
- ORCFile: 131GB

The Text File Format

This is the default file format for Hive. Data is not compressed in the text file. It can be compressed with compression tools, such as GZip, Bzip2, and Snappy. However, these compressed files are not splittable as input during processing. As a result, it leads to running a single, huge map job to process one big file.

The Sequence File Format

This is a binary storage format for key/value pairs. The benefit of a sequence file is that it is more compact than a text file and fits well with the MapReduce output format. Sequence files can be compressed on record or block level where block level has a better compression ratio. To enable block level compression, we need to do the following settings:

```
SET hive.exec.compress.output=true;
SET io.seqfile.compression.type=BLOCK;
```

Unfortunately, both text and sequence files as a row level storage file format are not an optimal solution since Hive has to read a full row even if only one column is being requested. For instance, a hybrid row-columnar storage file format, such as RCFILE, ORC, and PARQUET implementation, is created to resolve this problem.

The RCFile Data Format

This is short for Record Columnar File. It is a flat file consisting of binary key/value pairs that shares much similarity with a sequence file. The RCFile splits data horizontally into row groups. One or several groups are stored in an HDFS file. Then, RCFile saves the row group data in a columnar format by saving the first column across all rows, then the second column across all rows, and so on. This format is splittable and allows Hive to skip irrelevant parts of data and get the results faster and cheaper.

The ORC Data Format

The ORC (Optimized Row Columnar) file format provides a highly efficient way to store Hive data. ORC has the following advantages over the RCFile format:

- Reduced load on the NameNode
- Support for an extended range of types: datetime, decimal, and the complex types (struct, list, map and union)
- Light-weight indexes stored within the file
- Seeking for a given row in the data set
- Data compression
- Efficient and flexible metadata storage using Google's Protocol Buffers

The PARQUET Data Format

This is another row columnar file format that has a similar design to that of ORC. What's more, Parquet has a wider range of support for the majority projects in the Hadoop ecosystem compared to ORC that only supports Hive and Pig. Parquet leverages the design best practices of Google's Dremel to support the nested structure of data. Parquet is supported by a plugin since Hive 0.10.0 and has got native support since Hive 0.13.0.

Converting Text to ORC Data Format

- Let's say you have a text-based Hive table employee
- Create a table that has the same schema as employee, but has the STORED AS ORC qualifier:

```
CREATE TABLE IF NOT EXISTS employee_orc(
name string,
work_place ARRAY<string>,
sex_age STRUCT<sex:string,age:int>,
skills_score MAP<string,int>,
depart_title MAP<STRING,ARRAY<STRING>>
)ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
COLLECTION ITEMS TERMINATED BY ','
MAP KEYS TERMINATED BY ':'
STORED AS ORC;
```

Converting Text to ORC Data Format

- Insert data into the ORC table from the TEXT table:

INSERT INTO TABLE employee_orc SELECT * FROM employee;

The TEXT-to-ORC conversion will happen automatically

- Note: By default, ORC uses the ZLIB compression codec. You can configure compression (or switch it off completely) by using the orc.compress Hive table property, for example:

STORED AS ORC tblproperties ("orc.compress"="SNAPPY")

will set the compression algorithm to SNAPPY; using NONE for this configuration value will disable

Converting ORC to PARQUET Data Format

- Create a table that has the same schema as employee, but has the STORED AS PARQUET qualifier:

```
CREATE TABLE IF NOT EXISTS employee_parquet(
name string,
work_place ARRAY<string>,
sex_age STRUCT<sex:string,age:int>,
skills_score MAP<string,int>,
depart_title MAP<STRING,ARRAY<STRING>>
)ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
COLLECTION ITEMS TERMINATED BY ','
MAP KEYS TERMINATED BY ':'
STORED AS PARQUET;
```

Converting ORC to PARQUET Data Format

- Insert data into the PARQUET table from the ORC table:

```
INSERT INTO TABLE employee_parquet SELECT * FROM employee_orc;
```

The ORC-to-PARQUET conversion will happen automatically

More on File Formats

- The STORED BY clause is used to support non-native artifacts, e.g. HBase tables
- STORED AS INPUTFORMAT ... OUTPUTFORMAT allows users to specify the file format managing Java Class names, e.g.

```
org.apache.hadoop.hive.contrib.fileformat.base64.Base64TextInputFormat
com.hadoop.mapred.DeprecatedLzoTextInputFormat
org.apache.hadoop.hive.q1.io.HiveIgnoreKeyTextOutputFormat
```

Example of Creating Managed Table

- Create the internal table and load the data

```
CREATE TABLE IF NOT EXISTS employee_internal(
        name string,
        work_place ARRAY<string>,
        gender_age STRUCT<gender:string,age:int>,
        skills_score MAP<string,int>,
        depart_title MAP<STRING,ARRAY<STRING>>
)
COMMENT 'This is an internal table'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
COLLECTION ITEMS TERMINATED BY ','
MAP KEYS TERMINATED BY ':'
STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH '/root/employee.txt' OVERWRITE INTO TABLE
employee_internal;
```

The EXTERNAL DDL Parameter

- The EXTERNAL parameter allows users to create a table in location different from the specified by the hive.metastore.warehouse.dir property in the hive-site.xml configuration file
- The directory for the table is specified by the LOCATION DDL parameter of the CREATE EXTERNAL TABLE statement
- This option gives an advantage of re-using data already generated in that location and which has data structure that corresponds to the field data types declared by the CREATE statement
- Note: When you drop a table created with EXTERNAL parameter, data in the table is not deleted in HDFS (since Hive does not own the data)
- Tables created without the EXTERNAL parameter are referred to as as Hive-managed table

Example of Using EXTERNAL

- If you have an HDFS folder named /user/me/myexternal_folder/ which contains a text file with two tab-delimited columns, you can use the following CREATE EXTERNAL TABLE statement against this location:

```
CREATE EXTERNAL TABLE tblExternal (col1 STRING, col2 STRING)
COMMENT 'Cteating a table at a specific location'
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE
LOCATION '/user/me/myexternal_folder/';
```

Creating an Empty Table

- In some cases, you may need to have a table with the structure similar to that of an already existing table

- Use the following command to copy the table definition:

```
CREATE TABLE T2 LIKE T1;
```

In this case, the T2 table will be created with the same table definition used in creating the T1 table No records will be copied and T2 will be empty

Dropping a Table

- Use the following command to drop a table
  DROP TABLE [IF EXISTS] table_name

- The DROP TABLE statement removes table's metadata as well as the data in the table

- Note: The data is moved to the ./Trash/Current directory in the user's home HDFS directory (if trashing is configured);the table metadata in the metadstore can no longer be recovered. When dropping a table created with the EXTERNAL parameter, the original data fille will not be deleted from the file system

Table/Partitions Truncation

- Table/Partitions truncation removes all rows from a table or partitions
- The command has the following syntax:

  TRUNICATE TABLE table_name [PARTITION partition_spec];

- User can specify partial partition_spec for trunicating specific partitions

Alter Table/Partition/Column

- These commands allow user to have a fine-grained control over the table structure, e.g.
  - Changing table name:

    ALTER TABLE table_name RENAME TO new_table_name;
  - ADDING a partition:

    ALTER TABLE table_name ADD PARTITION [partCol = 'pc_name']
    location 'path_to_data_location';
  - Dropping a partition:

    ALTER TABLE table_name DROP [IF EXISTS] PARTITION partition_spec, PARTITION partition_spec, ...
  - Changing column:

    ALTER TABLE test_name CHANGE orgi_col_name new_col_name
- This command will allow users to change a column's name, and (optionally) its data type

More Examples of Managed Table

```
CREATE TABLE IF NOT EXISTS employee_internal (
  name string,
  work_place ARRAY<string>,
  sex_age STRUCT<sex:string,age:int>,
  skills_score MAP<string,int>,
  depart_title MAP<STRING,ARRAY<STRING>>
) COMMENT 'This is an internal table'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
COLLECTION ITEMS TERMINATED BY ','
MAP KEYS TERMINATED BY ':'
STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH '/training/apps/hive/dataset/employee.txt'
OVERWRITE INTO TABLE employee_internal;
```

More Examples of Managed Table

```
CREATE TABLE employee_test(
            name string,
            age int
);

drop table employee_test;

CREATE TABLE employee_test_partition(
            name string,
            age int)
partitioned by (edate string);

insert into table employee_test_partition partition (edate) values("peter", 30,
"20161001") ;

insert into table employee_test_partition partition (edate) values("john", 50,
"20161003") ;
```

More Examples of Managed Table

ALTER TABLE employee_test_partition PARTITION(edate="20161001") RENAME TO PARTITION(edate="20161011");

ALTER TABLE employee_test_partition DROP IF EXISTS PARTITION(edate="20161011");

LOAD DATA INPATH '/user/root/employee_test_partition.txt' INTO TABLE employee_test_partition PARTITION (edate="20161101");

ALTER TABLE employee_test_partition ADD PARTITION (edate="20171101") location "/user/root/20171101";

(the files are not moved into /apps/user/warehouse for managed table, change the file will immediately reflect in the table);

More Examples of Managed Table

ALTER TABLE employee_test_partition PARTITION(edate="20161001") RENAME TO PARTITION(edate="20161011");

ALTER TABLE employee_test_partition DROP IF EXISTS PARTITION(edate="20161011");

LOAD DATA INPATH '/user/root/employee_test_partition.txt' INTO TABLE employee_test_partition PARTITION (edate="20161101");

ALTER TABLE employee_test_partition ADD PARTITION (edate="20171101") location "/user/root/20171101";

(the files are not moved into /apps/user/warehouse for managed table, change the file will immediately reflect in the table);

More Examples of Managed Table

```
hadoop fs -mv
/apps/hive/warehouse/employee_test_partition/edate=20161001
/apps/hive/warehouse/employee_test_partition/emdate=20161001

hadoop fs -mv
/apps/hive/warehouse/employee_test_partition/edate=20161003
/apps/hive/warehouse/employee_test_partition/emdate=20161003

ALTER TABLE employee_test_partition DROP IF EXISTS
PARTITION(edate="20161001");

ALTER TABLE employee_test_partition DROP IF EXISTS
PARTITION(edate="20161003");

update hive.PARTITION_KEYS set PKEY_NAME = "emdate" where TBL_ID =
18;
```

More Examples of Managed Table

```
ALTER TABLE employee_test_partition ADD PARTITION (emdate="20161001")
location
"/apps/hive/warehouse/employee_test_partition/emdate=20161001";


ALTER TABLE employee_test_partition ADD PARTITION (emdate="20161003")
location
"/apps/hive/warehouse/employee_test_partition/emdate=20161003";

CREATE TABLE employee_test(
            name string,
            age int
)
location "/user/root/20171101";
```

(the files are not moved into /apps/user/warehouse for managed table, change the file will immediately reflect in the table);

Examples of Dynamic Partition

SET hive.exec.dynamic.partition.mode;
SET hive.exec.dynamic.partition=true;

Dynamic partition insert could potentially be a resource hog in that it could generate a large number of partitions in a short time. To get yourself buckled, we define three parameters:

- hive.exec.max.dynamic.partitions.pernode (default value being 2000) is the maximum dynamic partitions that can be created by each mapper or reducer. If one mapper or reducer created more than that the threshold, a fatal error will be raised from the mapper/reducer (through counter) and the whole job will be killed.

Examples of Dynamic Partition

- hive.exec.max.dynamic.partitions (default value being 5000) is the total number of dynamic partitions could be created by one DML. If each mapper/reducer did not exceed the limit but the total number of dynamic partitions does, then an exception is raised at the end of the job before the intermediate data are moved to the final destination.

- hive.exec.max.created.files (default value being 5000) is the maximum total number of files created by all mappers and reducers. This is implemented by updating a Hadoop counter by each mapper/reducer whenever a new file is created. If the total number is exceeding hive.exec.max.created.files, a fatal error will be thrown and the job will be killed.

More Examples of External Table

```
CREATE EXTERNAL TABLE IF NOT EXISTS employee_external(
  name string,
  work_place ARRAY<string>,
  sex_age STRUCT<sex:string,age:int>,
  skills_score MAP<string,int>,
  depart_title MAP<STRING,ARRAY<STRING>>
 )
COMMENT 'This is an external table'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
COLLECTION ITEMS TERMINATED BY ','
MAP KEYS TERMINATED BY ':'
STORED AS TEXTFILE
LOCATION '/user/root/employee_external';

LOAD DATA LOCAL INPATH '/training/apps/hive/dataset/employee.txt' OVERWRITE INTO TABLE
employee_external;
```

More Examples of External Table

```
CREATE EXTERNAL TABLE employee_external_test(
            name string,
            age int
)
partitioned by (edate string);
```

insert into table employee_external_test partition (edate) values("peter", 30, "20161001") ;

data is written to /apps/user/warehouse since external table doesn't define location

More Examples of External Table

insert into table employee_external_test partition (edate) select name, age, emdate from employee_test_partition;

insert into table employee_external_test partition (edate) select * from employee_test_partition;

hadoop fs -put '/root/employee_test_partition.txt' /user/root/employee_test_partition.txt

LOAD DATA INPATH '/user/root/employee_test_partition.txt' INTO TABLE employee_external_test PARTITION (edate="20181101");

(the files are removed even ig it is external table)

More Examples of External Table

hadoop fs -mkdir /apps/hive/warehouse/employee_external_test/edate=20191001

hadoop fs -put '/root/employee_test_partition.txt'

/apps/hive/warehouse/employee_external_test/edate=20191001

MSCK REPAIR TABLE employee_external_test;

hadoop fs -mkdir /user/root/data

hadoop fs -mkdir /user/root/data/edate=20160101

hadoop fs -put '/root/employee_test_partition.txt'
/user/root/data/20160101/employee_test_partition.txt

More Examples of External Table

```
CREATE EXTERNAL TABLE employee_external_test1(
            name string,
            age int
)
partitioned by (edate string)
location "/user/root/data";

MSCK REPAIR TABLE employee_external_test1;

insert into table employee_external_test1 partition (edate) values("peter", 30,
"20160109");

hadoop fs -put '/root/employee_test.txt' /user/root/data/edate=20160101
```

Example of the Internal and External TABLE Statement

- Create Table With Data

CREATE TABLE AS SELECT (CTAS)

CREATE TABLE ctas_employee AS SELECT * FROM employee_external;

- Create Table As SELECT (CTAS) with Common Table Expression (CTE)

```
CREATE TABLE cte_employee AS
WITH r1 AS (SELECT name FROM r2 WHERE name = 'Michael'),
r2 AS (SELECT name FROM employee WHERE sex_age.sex= 'Male'),
r3 AS (SELECT name FROM employee WHERE sex_age.sex= 'Female')
SELECT * FROM r1 UNION ALL select * FROM r3;
```

Example of the Internal and External Statement

- Create Table Without Data

With CTAS

```
CREATE TABLE empty_ctas_employee AS SELECT * FROM employee_internal
WHERE 1=2;
```

With LIKE

```
CREATE TABLE empty_like_employee LIKE employee_internal;
```

Example of the Internal and External Statement

- Alter table statements, alter table name

ALTER TABLE cte_employee RENAME TO c_employee;

- Alter table properties, such as comments

ALTER TABLE c_employee SET TBLPROPERTIES ('comment' = 'New name with new comments');

- Alter Table File Format

ALTER TABLE c_employee SET FILEFORMAT ORC;
ALTER TABLE c_employee CONCATENATE;
ALTER TABLE c_employee SET FILEFORMAT TEXTFILE;

- Alter Table Protection

ALTER TABLE c_employee ENABLE NO_DROP;
ALTER TABLE c_employee DISABLE NO_DROP;
ALTER TABLE c_employee ENABLE OFFLINE;
ALTER TABLE c_employee DISABLE OFFLINE;

Example of the Internal and External Statement

- Add columns to the table

ALTER TABLE c_employee ADD COLUMNS (work string);

Views

- Hive views are virtual tables with no associated physical storage
- Views are built as a logical construct on top of physical tables
- You can run regular queries against views:

  SELECT name FROM myView WHERE phone LIKE '416%‘
- A view's schema is immutable and is defined when the view is created; subsequent changes to underlying tables (e.g. renaming or adding a column) will not be propagated to the view's schema. Any subsequent invalid reference to the underlying changed parameters will raise an exception

Create View Statement

- A view is created with the following statement:

CREATE VIEW [IF NOT EXISTS} view_name [(column_name [COMMENT column_comment], ...)]
[COMMENT view_comment]
AS SELECT ...

- For example:

CREATE VIEW vNames (firstName, lastName) AS SELECT fn, ln FROM User;

- Note: If no column names are supplied (they are optional) in the view definition, the names of the view's columns will be derived automatically from the defining AS SELECT expression. Column names may be redefined in the view definition

Why Use Views?

- Views are useful when you need:

Restrict viewable data based on some conditions (limiting columns and rows for security and other considerations) Wrap up complex queries

Restricting Amount of Viewable Data

- Views help restrict the amount of viewable data by the following techniques:
  - Providing a view of the subset of columns in the source table
  - Providing a subset of rows matching the WHERE clause
  - Using a combination of the above techniques
  - Using these techniques, you can hide sensitive information by not declaring them in the view

Examples of Restricting Amount of Viewable Data

- You can restrict the viewing of the client income (which may be regarded as confidential information) in the client table defined as follows:

  CREATE TABLE client (fn STRING, ln STRING, income FLOAT)...;

  by omitting the income column in the view based on the client table:

  CREATE VIEW publicClientView AS SELECT fn, ln FROM client;

- Views can also be used to limit the number of returned records by specifying the WHERE matching condition in the AS SELECT clause:

  CREATE VIEW shortView AS
  SELECT * FROM bigTable WHERE <limiting conditions>;

Creating and Dropping Indexes

- Hive supports column indexing to speed up data querying
  - Hive indexing capabilities are limited
- Hive supports index creation on tables/partitions since Hive 0.7.0.
- The index in Hive provides key-based data view and better data access for certain operations, such as WHERE, GROUP BY, and JOIN
- Indexes are created with the CREATE INDEX... Statement
- The indexed data for a table is stored in another table defined in the CREATE INDEX... Statement
- You can drop a created index by using the DROP INDEX...statement
  - DROP INDEX drops the index and deletes the index table

Examples of Creating Index

- The command to create an index in Hive is straightforward as follows:

```
CREATE INDEX idx_id_employee_id ON TABLE employee_id (employee_id) AS 'COMPACT' WITH DEFERRED REBUILD;
```

- In addition to the COMPACT keyword used in the preceding example, Hive also supports BITMAP indexes since HIVE 0.8.0 for columns with less different values, as shown in the following example:

```
CREATE INDEX idx_sex_employee_id ON TABLE employee_id (sex_age) AS 'BITMAP' WITH DEFERRED REBUILD;
```

- The WITH DEFERRED REBUILD keyword in the preceding example prevents the index from immediately being built

- When data in the base table changes, the ALTER…REBUILD command must be used to bring the index up to date

```
ALTER INDEX idx_id_employee_id ON employee_id REBUILD;
```

Describing Data

- 67.1 Hive offers the DESCRIBE statement that can be applied to a number of objects: databases, tables, partitions, views, and columns
- 67.2 The DESCRIBE statement shows metadata associated with the target object
- 67.3 The general syntax of the statement:

  DESCRIBE some_object;

  For example:

  DESCRIBE my Table;

Summary

Hive offers an extensive Data Definition Language (DDL) that addresses the most practical user needs, such as:

- Creating and dropping Hive databases
- Creating and dropping tables
- Performing table partitioning for faster data querying
- Creating views to minimize the amount of viewable data either for performance of security considerations

Objective

In this section, participants will learn about Hive's Data Manipulation Language (DML) and its two primary ways of data loading:

Using the LOAD DATA statement
Using the INSERT statement
Using the Import statement
Using the Export statement

Hive Data Manipulation Language (DML)

- The Hive Data Manipulation Language (DML) deals with loading data in a table or partition

- There are three primary ways to load data in Hive:
  - Using the LOAD statement
  - Using the INSERT statement
  - Using the INSERT...VALUES statement

- Note: (Hive apparently supports INSERT...VALUES starting in Hive 0.14)

Using the LOAD DATA statement

- The LOAD DATA statement performs the bulk load operation and has the following syntax:

LOAD DATA [LOCAL] INPATH 'file path' [OVERWRITE] INTO TABLE table name [PARTITION (partcol1=val1,partcol2=val2 ...)]

- If the keyword LOCAL is specified, then the contents of the file specified by the file path parameter is loaded (copied over from the OS file system)into the target file system, The file path parameter can be a relative or an absolute path to the file

Using the LOAD DATA statement

- If the keyword LOCAL is not specified, then Hive will apply some file location algorithm and move the file specified by the file path parameter into Hive
- If the OVERWRITE keyword is present, then the contents of the target table (or partition) will be overwritten by the file referred to by file path
- If the OVERWRITE keyword is not present, the source file's content is added to the target table

Example of Loading Data into a Hive Table

- Load local data to the Hive table:

```
LOAD DATA LOCAL INPATH '/root/employee_hr.txt' OVERWRITE INTO TABLE employee_hr;
```

- Load local data to the Hive partition table:

```
LOAD DATA LOCAL INPATH '/root/employee.txt' OVERWRITE INTO TABLE employee_partitioned
PARTITION (year=2014,month=12);
```

- Load HDFS data to the Hive table using the default system path:

```
LOAD DATA INPATH '/user/root/employee.txt' OVERWRITE INTO TABLE employee;
```

- Load HDFS data to the Hive table with full URI:

```
LOAD DATA INPATH 'hdfs://127.0.0.1:8020/user/root/employee.txt' OVERWRITE INTO TABLE
employee;
```

Loading Data with the INSERT Statement

- The INSERT statement allows inserting data from SELECT-based queries

- The INSERT statement has two variants:

INSERT OVERWRITE TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2...)] select_statements1 FROM from_statement;

INSERT TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2...)] select_statements1 FROM from_statement;

Appending and Replacing Data with the INSERT Statement

- The INSERT OVERWRITE variant will overwrite any existing data in the table or partition
- The INSERT INTO variant will append to the table or partition
- Data inserts with the INSERT statement can be done to a table or a partition

Examples of Using the INSERT Statement

INSERT INTO TABLE Q1 SELECT sales FROM JanSales;

INSERT INTO TABLE Q1 SELECT sales FROM FebSales;

INSERT INTO TABLE Q1 SELECT sales FROM MarSales;

The above three INSERT statements will append data from three different tables to the Q1 table

Multi Table Inserts

- Multi Table Inserts is an optimization technique that helps minimize the number of data scans on the source table

The source data is scanned only once and it is then re-used as input for building different queries which result sets for multiple INSERT statements

Multi Table Inserts Syntax

- Multiple Table Inserts work for both data appending and data overwriting

FROM from_statement

INSERT OVERWRITE TABLE tablename1 [PARTITION(partcol1=val1, partcol2=val2 ...)] select_statement1

[INSERT OVERWRITE TABLE tablename2 [PARTITION(partcol1=val1, partcol2=val2 ...)] select_statement2] ...;

- Note: The Multi Table statement starts with the FROM keyword and ends with the ";"

Multi Table Inserts Syntax

- The FROM from_statement clause a full scan of the source data. The from_statement can be the name of a Hive table or a JOIN statement. The scanned result returned by the FROM from_statement clause can now be re-used for multiple table inserts

Multi Table Inserts Example

```
FROM sourceTable sales
INSERT OVERWRITE TABLE destTable1 SELECT s.col1 WHERE s.col1 != 999
INSERT OVERWRITE TABLE destTable2 SELECT s.col1 WHERE s.col2 < 0
```

Note: There is no FROM clause in the INSERT statements that follow the FROM clause

More Inserts Examples

- Populate data from SELECT

```
INSERT OVERWRITE TABLE employee SELECT * FROM ctas_employee;
```

- INSERT from CTE

```
WITH a as (SELECT * FROM ctas_employee )
FROM a INSERT OVERWRITE TABLE employee SELECT *;
```

- Multiple INSERTS by only scanning the source table once

```
FROM ctas_employee
INSERT OVERWRITE TABLE employee
SELECT *
INSERT OVERWRITE TABLE employee_internal
SELECT * ;
```

More Inserts Examples

- Insert to local files with default row separators

```
INSERT OVERWRITE LOCAL DIRECTORY '/root/output1' SELECT * FROM employee;
```

- Insert to local files with specified row separators

```
INSERT OVERWRITE LOCAL DIRECTORY '/root/output2'
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
SELECT * FROM employee;
```

- Multiple INSERTFROM employee

```
FROM employee
INSERT OVERWRITE DIRECTORY '/user/root/output'
SELECT *
INSERT OVERWRITE DIRECTORY '/user/root/output1'
SELECT * ;
```

Data Exchange

When working with Hive, sometimes we need to migrate data among different environments. Or we may need to back up some data. Since Hive 0.8.0, EXPORT and IMPORT statements are available to support the import and export of data in HDFS for data migration or backup/restore purposes.

Data Exchange - Export

The EXPORT statement will export both data and metadata from a table or partition. Metadata is exported in a file called _metadata. Data is exported in a subdirectory called data:

```
EXPORT TABLE employee TO '/root/output';
```

Data Exchange - Import

After EXPORT, we can manually copy the exported files to other Hive instances or use Hadoop distcp commands to copy to other HDFS clusters. Then, we can import the data in the following manner:

- Import data to a table with the same name. It throws an error if the table exists

- Import data to a new table:

```
IMPORT TABLE empolyee_imported FROM '/user/root/output';
```

- Import data to an external table, where the LOCATION property is optional:

```
IMPORT EXTERNAL TABLE empolyee_imported_external FROM '/user/root/output'
LOCATION '/user/root/output1' ;
```

Data Exchange - Import

Export and import partitions:

```
EXPORT TABLE employee_partitioned partition (year=2014, month=11) TO '/user/root/output';
```

```
IMPORT TABLE employee_partitioned_imported FROM '/user/root/output';
```

Summary

Hive supports data loading into tables and partitions using four statements:

- LOAD DTAA
- INSERT
- EXPORT
- IMPORT

Objective

In this section, participants will learn about:

- Hive Query Language (HiveQL)
- SELECT and related statements
- HiveQL built-in functions

HiveQL

- For data query, Hive uses an SQL-like language called HiveQL
- While many of the language structs resemble SQL-92, HiveQL does not claim full standard compliance
- HiveQL offers Hive-specific extensions that help leverage Hive internal architecture
- Hive offers only basic support for indexes, so, in most cases, a full table scan is performed when a query is run against a table

The SELECT Statement Syntax

The center-piece of HiveQL is the SELECT statement which has the following syntax:

```
SELECT [ALL|DISTINCT] select_expr1, select_expr2,...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list]
[HAVING having_condition]
[LIMIT number]
```

The WHERE Clause

- Hive closely models the SQL's WHERE clause syntax
- As of Hive 0.13, some types of subqueries are supported in the WHERE clause
- The WHERE clause supports a number of Hive operator and user-defined functions that evaluate to a Boolean result

Examples of the WHERE Statement

For example:

```
SELECT ... FROM ... WHERE X <= Y AND X != Z;
SELECT ... FROM ... WHERE round(X) = Y;
```

Using Hive's round() built-in functionally

```
SELECT ... FROM ... WHERE X IS NULL;
SELECT ... FROM ... WHERE X IS NOT NULL;
SELECT ... FROM ... WHERE X LIKE 'A%';
```

Examples of the WHERE Statement

Finding all rows where values in the X column start with 'A'

SELECT ... FROM ... WHERE X LIKE '%Z‘

Finding all rows where values in the X column end with 'Z'

Notes: Through its WHERE extension, the RLIKE clause, Hive also allow users to use more powerful java regular expressions for finding matches

For example, SELECT ... FROM ... WHERE name RLIKE '.*(Bill|William).*'

Partition-based Queries

- Generally, HiveQL's SELECT query performs a full table scan (a highly inefficient process) when finding the matching rows
- Hive offers a query optimization technique bases on table partitioning
- Partitions are column-based and are created using the PARTITIONED BY clause of the CREATE TABLE statement
- SELECT statements that take advantage of existing table partitions help reduce amount of data to be scanned

Example of an Efficient SELECT Statement

- Provided that the stats table is partitioned by the date field, the following select statement will speed up queries by limiting the amount of data to be scanned (it is assumed that there are data partitioning directories named 2014-01-01, 2014-01-02,...,2014-02-01, 2014-02-02, ...2014-12-31):

```
SELECT * FROM stats WHERE date > '2014-05-01' AND date <= '2014-09-01'
```

More Examples of Statement

- Query all columns in the table

```
SELECT * FROM employee;
```

- Select only one column

```
SELECT name FROM employee;
```

- Select unique rows

```
SELECT DISTINCT name FROM employee LIMIT 2;
```

- Enable fetch

```
SET hive.fetch.task.conversion=more;
```

More Examples of Statement

- Verify the improvement, which is less than one second

```
SELECT name FROM employee;
```

- Use LIMIT and WHERE keywords

```
SELECT name, work_place FROM employee WHERE name = 'Michael';
```

- Nest SELECT using CTE

```
WITH t1 AS (SELECT * FROM employeeWHERE sex_age.sex = 'Male')
SELECT name, sex_age.sex AS sex FROM t1;
```

- Nest SELECT after the FROM

```
SELECT name, sex_age.sex AS sex FROM
( SELECT * FROM employee  WHERE sex_age.sex = 'Male') t1;
```

More Examples of Statement

- Subquery

```
SELECT name, sex_age.sex AS sex
FROM employee a
WHERE a.name IN(SELECT name FROM employee WHERE sex_age.sex = 'Male');

SELECT name, sex_age.sex AS sex
FROM employee a
WHERE EXISTS(SELECT * FROM employee b WHERE a.sex_age.sex = b.sex_age.sex
AND b.sex_age.sex = 'Male');
```

The DISTINCT Clause

- The DISTINCT clause removes duplicate rows in the output

The default is to return all the matching records (the ALL clause)

SELECT f1, f2 FROM Features

101 303

101 303

101 404

202 505

Note: The default ALL clause is applied

The DISTINCT Clause

SELECT DISTINCT f1, f2 FROM Features

101, 303

101, 404

202, 505

Note: One 101, 303 row is dropped as the row duplication filter is applied across all columns

SELECT DISTINCT f1 FROM Features

101

202

Supported Numeric Operators

- HiveQL supports the standard set of arithmetic operations on numeric types in the SELECT statement

Operators	Description
A+B	Addition of A and B
A-B	Subtraction of B from A
A*B	Multiplication of A and B`
A/B	Division of A with B`
A%B	The remainder of dividing A with B`
A&B	Bitwise AND of A and B
A|B	Bitwise OR of A and B
A^B	Bitwise XOR of A and B`
~A	Bitwise NOT of A

- Arithmetic operators take any numeric type
- No type coercion (casting) is performed if both operands are of the same numeric type Otherwise, the value of the smaller of the two types is promoted to the wider type (with more allocated bytes) of the other operand

Built-in Mathematical Functions

- HiveQL supports the usual set of mathematical functions in its SELECT statement round(), floow(), ceil(), exp(), log(), sqrt(), sin(), cos() etc.

- Most mathematical functions return a DOUBLE or a NULL if a NULL parameter is passed to the functionally

Built-in Aggregate Functions

AVG(col)	Return the average of the values in column col
AVG(DISTINCT col)	Return the average of the distinct values in column col
COUNT(*)	Return the total number of retrieved rows, including rows containing NULL values
SUM(col)	Return the sum of the values in column col
SUM(DISTINCT col)	Return the sum of the distinct values in column col
MAX(col)	Return the maximum value in column col
MIN(col)	Return the minimum value in column col

Note: All functions return their results as a DOUBLE except for COUNT() which returns a BIGINT

Built-in Statistical Functions

Function	Description
CORR(col1,col2)	Return the correlation of two sets of numbers
COVAR_POP(col1,col2)	Return the covariance of a set of numbers
STDDEV_POP(col)	Return the standard deviation of a set of numbers

Note: All statistical functions return result as a DOUBLE For finding the mean of a population, use the AVG() function

Other Useful Built-in Functions

Function	Return Type	Description
INSTR(str,substr)	INT	Return the index of str where substr is found
LENGTH(s)	INT	Return the length of the string
REPEAT(str,n)	STRING	Repeat str n times
SPACE(n)	STRING	Returns n spaces
YEAR(timestamp) 00:00:00") return 2016	INT	Return the year part as an INT of a timestamp string, e.g., year("2016-10-03
MONTH(timestamp) 03 00:00:00") return 10	INT	Return the month part as an INT of a timestamp string, e.g., year("2016-10-
DAY(timestamp) 00:00:00") return 3	INT	Return the day part as an INT of a timestamp string, e.g., year("2016-10-03
TO_DATE(tinmestamp) 00:00:00") returns "2016-10-03"	STRING	Return the date part of a timestamp string, e.g. to_date("2016-10-03
col IN (val1, val2, ...)) otherwise	BOOLEAN	Return true if col equals one of the values in the list (val1, val2, ...) false

The GROUP BY Clause

- The GROUP BY statement is normally used in conjunction with the aggregate functions to group the result-set by one or more columns

- The SELECT statement performs an aggregation over each group

```
SELECT sales_month, SUM(sales) FROM sales_2014
WHERE city = 'Toronto' AND price >= '1000'
GROUP BY sales_month;
```

The HAVING Clause

- The HAVING clauses was added to HiveQL in ver 0.7 because the WHERE keyword can not be used with aggregate functions
- The following SELECT statement will list all employee' age where the number of employee per age group were not greater than 1

```
SELECT sex_age.age FROM employee GROUP BY sex_age.age HAVING count(*)<=1;
```

The LIMIT Clause

- The LIMIT clause sets the number of rows to be returned
- Note: The rows that will be returned are randomly chosen, Repeating this command may yield different results (different rows may be returned)

```
SELECT * FROM employee LIMIT 3
```

The ORDER BY Clause

- The ORDER BY syntax in HiveQL is similar to that of ORDER BY in SQL language

- The ORDER BY supports ascending and descending ordering represented by the ASC (default) and DESC keywords respectively, e.g.

```
SELECT * FROM myTable ORDER BY col1 DESC;
```

- A sorted order is maintained across all of the output from every reducer. It performs the global sort using only one reducer, so it takes a longer time to return the result. Usage with LIMIT is strongly recommended for ORDER BY. When hive.mapred.mode = strict (by default, hive.mapred.mode = nonstrict) is set and we do not specify LIMIT, there are exceptions. This can be used as follows:

The SORT BY Clause

This indicates which columns to sort when ordering the reducer input records. This means it completes sorting before sending data to the reducer. The SORT BY statement does not perform a global sort and only makes sure data is locally sorted in each reducer unless we set mapred.reduce.tasks=1. In this case, it is equal to the result of ORDER BY. It can be used as follows:

The DISTRIBUTE BY Clause

Rows with matching column values will be partitioned to the same reducer. When used alone, it does not guarantee sorted input to the reducer. The DISTRIBUTE BY statement is similar to GROUP BY in RDBMS in terms of deciding which reducer to distribute the mapper output to. When using with SORT BY, DISTRIBUTE BY must be specified before the SORT BY statement. And, the column used to distribute must appear in the select column list. It can be used as follows:

```
SELECT name, employee_id FROM employee_hr DISTRIBUTE BY employee_id;
```

```
SELECT name, employee_id FROM employee_hr DISTRIBUTE BY employee_id SORT BY name;
```

The CLUSTER BY Clause

This is a shorthand operator to perform DISTRIBUTE BY and SORT BY operations on the same group of columns. And, it is sorted locally in each reducer. The CLUSTER BY statement does not support ASC or DESC yet. Compared to ORDER BY, which is globally sorted, the CLUSTER BY operation is sorted in each distributed group. To fully utilize all the available reducers when doing a global sort, we can do CLUSTER BY first and then ORDER BY. This can be used as follows:

```
SELECT name, employee_id FROM employee_hr CLUSTER BY name;
```

The JOIN Clause

- HiveQL supports the SQL-like JOIN clause which is used to combine rows from two or more tables on values from a common column

- Only equality joins are allowed where the equak condition is used (t1.col1 = t2.col2)

Examples:

```
SELECT A.* FROM A JOIN B ON (A.col1 = B.col2)
```

More than 1 tables can be joined in the same query, e.g.:

```
SELECT A.col1, B.col2, C.col3
FROM A JOIN B ON (A.id1 = B.id2)
JOIN C ON (C.id3 = B.id4)
```

The CASE ... Clause

- HiveQL supports the if-like CASE ... WHEN ... ELSE combined statement which has the following syntax:

```
SELECT col1, ...,
CASE WHEN <condition A> THEN <label 1>
[WHEN <condition B> THEN <label 2>, WHEN ...]
ELSE <label N>
END AS <pseudo_column_name>
FROM table_name;
```

Example of CASE ... Clause

```
SELECT location, datetime,
        CASE
        WHEN spedMPH > 40 AND sppedMPH < 73 THEN
'Light'
        WHEN spedMPH >= 73 AND sppedMPH < 113 THEN
'Moderate'
        WHEN spedMPH >= 113 AND sppedMPH < 158
THEN 'Considerable'
        WHEN spedMPH >=158 THEN 'Severe'
        ELSE 'Unknown' END AS category FROM Tornadoes;
```

Summary

- 100.1 HiveQL offers SQL-like SELECT statement with the related WHERE, GROUP BY and HAVING clauses

- 100.2 HiveQL supports the common set of

  Numeric Operators (+/-,%,etc.)

  Built-in Aggregate Functions (AVG(),SUM(),COUNT() etc.)

  Assorted functions (INSTR(),LENGTH(),YEAR() etc.)

- 100.3 In addition, HiveQL has a number of built-in statistical functions (CORR(), STDDEV_POP() etc)

- 100.4 The CASE ... WHEN ... THEN ... ELSE combined statement is also supported

Join

Hive JOIN is used to combine rows from two or more tables together. Hive supports common JOIN operations such as what's in the RDBMS, for example, JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN, and CROSS JOIN. However, Hive only supports equal JOIN instead of unequal JOIN, because unequal JOIN is difficult to be converted to MapReduce jobs.

The common join is also called reduce side join. It is a basic join in Hive and works for most of the time. For common joins, we need to make sure the big table is on the rightmost side or specified by hit, as follows:

/*+ STREAMTABLE(stream_table_name) */.

Inner Join

The INNER JOIN in Hive uses JOIN keywords, which return rows meeting the JOIN conditions from both left and right tables. The INNER JOIN keyword can also be omitted by comma-separated table names since Hive 0.13.0.

- Prepare another table to join and load data:

```
CREATE TABLE IF NOT EXISTS employee_hr (
                name string,
                employee_id int,
                sin_number string,
                start_date date
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH '/root/employee_hr.txt' OVERWRITE INTO TABLE employee_hr;
```

Inner Join

- Perform inner JOIN between two tables with equal JOIN conditions:

```
SELECT emp.name, emph.sin_number
FROM employee emp
JOIN employee_hr emph ON emp.name = emph.name;
```

- The JOIN operation can be performed among more tables (three tables in this case), as follows:

```
SELECT emp.name, empi.employee_id, emph.sin_number
FROM employee emp
JOIN employee_hr emph ON emp.name = emph.name
JOIN employee_id empi ON emp.name = empi.name;
```

- Self-join is a special JOIN where one table joins itself. When doing such joins, a different alias should be given to distinguish the same table

```
SELECT emp.name
FROM employee emp
JOIN employee emp_b
ON emp.name = emp_b.name;
```

Inner Join

- Implicit join is a JOIN operation without using the JOIN keyword:

```
SELECT emp.name, emph.sin_number
FROM employee emp, employee_hr emph
WHERE emp.name = emph.name;
```

- The JOIN operation uses different columns in join conditions and will create an additional MapReduce:

```
SELECT emp.name, empi.employee_id, emph.sin_number
FROM employee emp
JOIN employee_hr emph ON emp.name = emph.name
JOIN employee_id empi ON emph.employee_id = empi.employee_id;
```

Inner Join

- When JOIN is performed between multiple tables, the MapReduce jobs are created to process the data in the HDFS. Each of the jobs is called a stage. Usually, it is suggested for JOIN statements to put the big table right at the end for better performance as well as avoiding **Out Of Memory** (**OOM**) exceptions, because the last table in the sequence is streamed through the reducers where the others are buffered in the reducer by default. Also, a hint, such as /*+STREAMTABLE (table_name)*/, can be specified to tell which table is streamed as follows:

```
SELECT /*+ STREAMTABLE(employee_hr) */
emp.name, empi.employee_id, emph.sin_number
FROM employee emp
JOIN employee_hr emph ON emp.name = emph.name
JOIN employee_id empi ON emph.employee_id = empi.employee_id;
```

Outer Join

Besides INNER JOIN, Hive also supports regular OUTER JOIN and FULL JOIN. The logic of such JOIN is the same to what's in the RDBMS.

```
SELECT emp.name, emph.sin_number
FROM employee emp
LEFT JOIN employee_hr emph ON emp.name = emph.name;
```

```
SELECT emp.name, emph.sin_number
FROM employee emp
RIGHT JOIN employee_hr emph ON emp.name = emph.name;
```

```
SELECT emp.name, emph.sin_number
FROM employee emp
FULL JOIN employee_hr emph ON emp.name = emph.name;
```

Cross Join

The CROSS JOIN statement, which is available since Hive 0.10.0, does not have the JOIN condition. The CROSS JOIN statement can also be written using JOIN without condition or with the *always true* condition, such as 1 = 1. The following three ways of writing CROSS JOIN produce the same result set:

```
SELECT emp.name, emph.sin_number
FROM employee emp
CROSS JOIN employee_hr emph;

SELECT emp.name, emph.sin_number
FROM employee emp
JOIN employee_hr emph;

SELECT emp.name, emph.sin_number
FROM employee emp
JOIN employee_hr emph on 1=1;
```

Cross Join

Although Hive does not support unequal JOIN explicitly, there are workarounds using CROSS JOIN and WHERE conditions mentioned in the following example:

```
SELECT emp.name, emph.sin_number
FROM employee emp
CROSS JOIN employee_hr emph WHERE emp.name <> emph.name;
```

Map Join

The MAPJOIN statement means doing the JOIN operation only by map without the reduce job. The MAPJOIN statement reads all the data from the small table to memory and broadcasts to all maps. During the map phase, the JOIN operation is performed by comparing each row of data in the big table with small tables against the join conditions. Because there is no reduce needed, the JOIN performance is improved.

When the hive.auto.convert.join setting is set to true, Hive automatically converts the JOIN to MAPJOIN at runtime if possible instead of checking the map join hint.

Map Join

Once autoconvert is enabled, Hive will automatically check if the smaller table file size is bigger than the value specified by hive.mapjoin.smalltable.filesize, and then Hive will convert the join to a common join. If the file size is smaller than this threshold, it will try to convert the common join into a map join.

SET hive.auto.convert.join=true; --default false

SET hive.mapjoin.smalltable.filesize=600000000; --default 25M

SET hive.auto.convert.join.noconditionaltask=true;

--default false. Set to true so that map join hint is not needed

SET hive.auto.convert.join.noconditionaltask.size=10000000;

--The default value controls the size of table to fit in memory

Map Join

The following is an example of MAPJOIN that is enabled by query hint:

```
SELECT /*+ MAPJOIN(employee) */ emp.name, emph.sin_number
FROM employee emp
CROSS JOIN employee_hr emph WHERE emp.name <> emph.name;
```

The MAPJOIN operation does not support the following:

- The use of MAPJOIN after UNION ALL, LATERAL VIEW, GROUP BY/JOIN/SORT BY/CLUSTER BY/DISTRIBUTE BY
- The use of MAPJOIN before UNION, JOIN, and another MAPJOIN

Bucket Map Join

The bucket map join is a special type of MAPJOIN that uses bucket columns (the column specified by CLUSTERED BY in the CREATE table statement) as the join condition. Instead of fetching the whole table as done by the regular map join, bucket map join only fetches the required bucket data.

To enable bucket map join, we need to set hive.optimize.bucketmapjoin = true and make sure the buckets number is a multiple of each other.

The following additional settings are needed to enable this behavior:

```
SET hive.optimize.bucketmapjoin = true;
SET hive.optimize.bucketmapjoin.sortedmerge = true;
SET
hive.input.format=org.apache.hadoop.hive.ql.io.BucketizedHiveInputFormat
;
```

Bucket Map Join

In bucket map join, all the join tables must be bucket tables and join on buckets columns. In addition, the buckets number in bigger tables must be a multiple of the bucket number in the small tables.

Left Semi Join

The LEFT SEMI JOIN statement is also a type of MAPJOIN. Before Hive supports IN/EXIST, LEFT SEMI JOIN is used to implement such a request as shown in the following example. The restriction of using LEFT SEMI JOIN is that the right-hand side table should only be referenced in the join condition, but not in WHERE or SELECT clauses.

```
SELECT a.name FROM employee a
WHERE EXISTS (SELECT * FROM employee_id b
WHERE a.name = b.name);
```

```
SELECT a.name FROM employee a
LEFT SEMI JOIN employee_id b
ON a.name = b.name;
```

Sort Merge Bucket (SMB)

SMB is the join performed on the bucket tables that have the same sorted, bucket, and join condition columns. It reads data from both bucket tables and performs common joins (map and reduce triggered) on the bucket tables. We need to enable the following properties to use SMB:

```
SET
hive.input.format=org.apache.hadoop.hive.ql.io.BucketizedHiveInputFormat
;
SET hive.auto.convert.sortmerge.join=true;
SET hive.optimize.bucketmapjoin=true;
SET hive.optimize.bucketmapjoin.sortedmerge=true;
SET hive.auto.convert.sortmerge.join.noconditionaltask=true;
```

Sort Merge Bucket Map (SMBM)

SMBM join is a special bucket join but triggers map-side join only. It can avoid caching all rows in the memory like map join does. To perform SMBM joins, the join tables must have the same bucket, sort, and join condition columns. To enable such joins, we need to enable the following settings:

SET hive.auto.convert.join=true;

SET hive.auto.convert.sortmerge.join=true

SET hive.optimize.bucketmapjoin=true;

SET hive.optimize.bucketmapjoin.sortedmerge=true;

SET hive.auto.convert.sortmerge.join.noconditionaltask=true;

SET hive.auto.convert.sortmerge.join.bigtable.selection.policy=org.apache.hadoop.hive.ql.optimizer.TableSizeBasedBigTableSelectorForAutoSM

Skew Join

When working with data that has a highly uneven distribution, the data skew could happen in such a way that a small number of compute nodes must handle the bulk of the computation. The following setting informs Hive to optimize properly if data skew happens:

```
SET hive.optimize.skewjoin=true;
--If there is data skew in join, set it to true. Default is false.

SET hive.skewjoin.key=100000;
--This is the default value. If the number of key is bigger than
--this, the new keys will send to the other unused reducers.
```

Skew Join

Skew data could happen on the GROUP BY data too. To optimize it, we need to do the following settings to enable skew data optimization in the GROUP BY result:

```
SET hive.groupby.skewindata=true;
```

Union All

To operate the result set vertically, Hive only supports UNION ALL right now. And, the result set of UNION ALL keeps duplicates if any.

- Check the name column in the employee_hr and employee table:

```
SELECT name FROM employee_hr;
SELECT name FROM employee;
```

- Use UNION on the name column from both tables, including duplications:

```
SELECT a.name FROM employee a
UNION ALL
SELECT b.name
FROM employee_hr b;
```

Union All

- Implement UNION between two tables without duplications:

```
SELECT DISTINCT name FROM (
        SELECT a.name AS name
        FROM employee a
        UNION ALL
        SELECT b.name AS name
        FROM employee_hr b
) union_set;
```

- The employee table implements INTERCEPT on employee_hr using JOIN:

```
SELECT a.name FROM employee a JOIN employee_hr b ON a.name = b.name;
```

- The employee table implements MINUS on employee_hr using OUTER JOIN:

```
SELECT a.name FROM employee a LEFT JOIN employee_hr b ON a.name = b.name
WHERE b.name IS NULL;
```

Basic Aggregation – GROUP BY

Data aggregation is any process to gather and express data in a summary form to get more information about particular groups based on specific conditions. Hive offers several built in aggregate functions, such as MAX, MIN, AVG, and so on.

The Hive basic built-in aggregate functions are usually used with the GROUP BY clause.

The following are a few examples using the built-in aggregate functions:

- Aggregation without GROUP BY columns:

```
SELECT count(*) AS row_cnt FROM employee;
```

Basic Aggregation – GROUP BY

- Aggregation with GROUP BY columns:

```
SELECT sex_age.sex, count(*) AS row_cnt FROM employee GROUP BY sex_age.sex;
```

If we have to select the columns that are not GROUP BY columns, one way is to use analytic functions, which are introduced later, to completely avoid using the GROUP BY clause. The other way is to use the collect_set function, which returns a set of objects with duplicate elements eliminated as follows:

```
SELECT sex_age.sex, collect_set(sex_age.age)[0] AS random_age, count(*) AS row_cnt
FROM employee GROUP BY sex_age.sex;
```

Basic Aggregation – GROUP BY

- Multiple aggregate functions are called in the same SELECT statement, as follows:

```
SELECT sex_age.sex, AVG(sex_age.age) AS avg_age, count(*) AS row_cnt
FROM employee GROUP BY sex_age.sex;
```

- These aggregate functions are used with CASE WHEN, as follows

```
SELECT sum(CASE WHEN sex_age.sex = 'Male' THEN sex_age.age ELSE 0 END)/
count(CASE WHEN sex_age.sex = 'Male' THEN 1 ELSE NULL END) AS male_age_avg FROM
employee;
```

Basic Aggregation – GROUP BY

- These aggregate functions are used with COALESCE and IF, as follows:

```
SELECT sum(coalesce(sex_age.age,0)) AS age_sum, sum(if(sex_age.sex =
'Female',sex_age.age,0)) AS female_age_sum FROM employee;
```

- Aggregate functions can also be used with the DISTINCT keyword:

```
SELECT count(DISTINCT sex_age.sex) AS sex_uni_cnt, count(DISTINCT name) AS name_uni_cnt
FROM employee;
```

Basic Aggregation – GROUP BY

When we use COUNT and DISTINCT together, Hive always ignores the setting (such as mapred.reduce.tasks = 20) for the number of reducers used and uses only one reducer.

The workaround is to use the subquery as follows:

```
--Trigger single reducer during the whole processing
SELECT count(distinct sex_age.sex) AS sex_uni_cnt FROM employee;

--Use subquery to select unique value before aggregations for better performance
SELECT count(*) AS sex_uni_cnt FROM (SELECT distinct sex_age.sex FROM employee) a;
```

Advanced Aggregation – GOUPING SETS

Hive has offered the GROUPING SETS keywords to implement advanced multiple GROUP BY operations against the same set of data. Actually, GROUPING SETS is a shorthand way of connecting several GROUP BY result sets with UNION ALL

The GROUPING SETS keyword completes all processes in one stage of jobs, which is more efficient than GROUP BY and UNION ALL having multiple stages.

A blank set () in the GROUPING SETS clause calculates the over aggregation

Advanced Aggregation – GOUPING SETS

Here are some examples, from there, you can see the difference between GROUP BY + UNION ALL and GROUP SETS

```
SELECT name, work_place[0] AS main_place,
count(employee_id) AS emp_id_cnt
FROM employee_id
GROUP BY name, work_place[0] GROUPING SETS(name, work_place[0]);
```

```
SELECT name, NULL AS main_place, count(employee_id) AS emp_id_cnt
FROM employee_id
GROUP BY name
UNION ALL
SELECT NULL AS name, work_place[0] AS main_place,
count(employee_id) AS emp_id_cnt
FROM employee_id
GROUP BY work_place[0];
```

Advanced Aggregation – GOUPING SETS

Here are some examples, from there, you can see the difference between GROUP BY + UNION ALL and GROUP SETS

```
SELECT name, work_place[0] AS main_place,
count(employee_id) AS emp_id_cnt
FROM employee_id
GROUP BY name, work_place[0]
GROUPING SETS((name, work_place[0]), name);
```

```
SELECT name, work_place[0] AS main_place,
count(employee_id) AS emp_id_cnt
FROM employee_id
GROUP BY name, work_place[0]
UNION ALL
SELECT name, NULL AS main_place, count(employee_id) AS emp_id_cnt
FROM employee_id
GROUP BY name;
```

Advanced Aggregation – GOUPING SETS

Here are some examples, from there, you can see the difference between GROUP BY + UNION ALL and GROUP SETS

```
SELECT name, work_place[0] AS main_place,
count(employee_id) AS emp_id_cnt
FROM employee_id
GROUP BY name, work_place[0]
GROUPING SETS((name, work_place[0]), name, work_place[0], ());
```

```
SELECT name, work_place[0] AS main_place,
count(employee_id) AS emp_id_cnt
FROM employee_id
GROUP BY name, work_place[0]
UNION ALL
SELECT name, NULL AS main_place, count(employee_id) AS emp_id_cnt
FROM employee_id
GROUP BY name
UNION ALL
SELECT NULL AS name, work_place[0] AS main_place,
count(employee_id) AS emp_id_cnt
FROM employee_id
GROUP BY work_place[0]
UNION ALL
SELECT NULL AS name, NULL AS main_place,
count(employee_id) AS emp_id_cnt
FROM employee_id;
```

Advanced Aggregation – ROLLUP

The ROLLUP statement enables a SELECT statement to calculate multiple levels of aggregations across a specified group of dimensions. The ROLLUP statement is a simple extension to the GROUP BY clause with high efficiency and minimal overhead to a query. Compared to GROUPING SETS that creates specified levels of aggregations, ROLLUP creates *n+1* levels of aggregations, where *n* is the number of grouping columns.

Advanced Aggregation – ROLLUP

First, it calculates the standard aggregate values specified in the GROUP BY clause. Then, it creates higher level subtotals, moving from right to left through the list of combinations of grouping columns, as shown in the following example:

GROUP BY a,b,c WITH ROLLUP

This is equivalent to the following:

GROUP BY a,b,c GROUPING SETS ((a,b,c),(a,b),(a),())

Advanced Aggregation – CUBE

The CUBE statement takes a specified set of grouping columns and creates aggregations for all of their possible combinations. If *n* columns are specified for CUBE, there will be *2n* combinations of aggregations returned, as shown in the following example:

GROUP BY a,b,c WITH CUBE

This is equivalent to the following:

GROUP BY a,b,c GROUPING SETS ((a,b,c),(a,b),(b,c),(a,c),(a),(b),(c),())

Advanced Aggregation – HAVING

Since Hive 0.7.0, HAVING is added to support the conditional filtering of GROUP BY results. By using HAVING, we can avoid using a subquery after GROUP BY. The following is an example:

SELECT sex_age.age FROM employee GROUP BY sex_age.age HAVING count(*)<=1;

Analytic Function

Analytic functions, available since Hive 0.11.0, are a special group of functions that scan the multiple input rows to compute each output value. Analytic functions are usually used with OVER, PARTITION BY, ORDER BY, and the windowing specification. Different from the regular aggregate functions used with the GROUP BY clause that is limited to one result value per group, analytic functions operate on windows where the input rows are ordered and grouped using flexible conditions expressed through an OVER PARTITION clause.

The syntax for the analyze function is as follows:

Function (arg1,..., argn) OVER ([PARTITION BY <...>] [ORDER BY <....>] [<window_clause>])

Analytic Function

Prepare the table and data for demonstration:

```
CREATE TABLE IF NOT EXISTS employee_contract (
name string,  dept_num int,
employee_id int, salary int,
type string, start_date date)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH '/root/employee_contract.txt' OVERWRITE INTO TABLE
employee_contract;
```

Analytic Function - Standard

This can be either COUNT(), SUM(), MIN(), MAX(), or AVG().

```
SELECT name, dept_num, salary,
COUNT(*) OVER (PARTITION BY dept_num) AS row_cnt,
SUM(salary) OVER(PARTITION BY dept_num ORDER BY dept_num) AS deptTotal,
SUM(salary) OVER(ORDER BY dept_num) AS runningTotal1,
SUM(salary) OVER(ORDER BY dept_num, name rows unbounded
preceding) AS runningTotal2
FROM employee_contract
ORDER BY dept_num, name;
```

Analytic Function - RANK

It ranks items in a group, such as finding the top *N* rows for specific conditions.

```
SELECT name, dept_num, salary,
RANK() OVER (PARTITION BY dept_num ORDER BY salary) AS rank,
DENSE_RANK() OVER (PARTITION BY dept_num ORDER BY salary)
AS dense_rank,
ROW_NUMBER() OVER () AS row_num,
ROUND((CUME_DIST() OVER (PARTITION BY dept_num
ORDER BY salary)), 1) AS cume_dist,
PERCENT_RANK() OVER(PARTITION BY dept_num
ORDER BY salary) AS percent_rank,
NTILE(4) OVER(PARTITION BY dept_num ORDER BY salary) AS ntile
FROM employee_contract
ORDER BY dept_num;
```

Analytic Function – DENSE_RANK

It is similar to RANK, but leaves no gaps in the ranking sequence when there are ties.

```
SELECT name, dept_num, salary,
RANK() OVER (PARTITION BY dept_num ORDER BY salary) AS rank,
DENSE_RANK() OVER (PARTITION BY dept_num ORDER BY salary)
AS dense_rank,
ROW_NUMBER() OVER () AS row_num,
ROUND((CUME_DIST() OVER (PARTITION BY dept_num
ORDER BY salary)), 1) AS cume_dist,
PERCENT_RANK() OVER(PARTITION BY dept_num
ORDER BY salary) AS percent_rank,
NTILE(4) OVER(PARTITION BY dept_num ORDER BY salary) AS ntile
FROM employee_contract
ORDER BY dept_num;
```

Analytic Function – CUME_DIST

It computes the number of rows whose value is smaller or equal to the value of the total number of rows divided by the current row

```
SELECT name, dept_num, salary,
RANK() OVER (PARTITION BY dept_num ORDER BY salary) AS rank,
DENSE_RANK() OVER (PARTITION BY dept_num ORDER BY salary)
AS dense_rank,
ROW_NUMBER() OVER () AS row_num,
ROUND((CUME_DIST() OVER (PARTITION BY dept_num
ORDER BY salary)), 1) AS cume_dist,
PERCENT_RANK() OVER(PARTITION BY dept_num
ORDER BY salary) AS percent_rank,
NTILE(4) OVER(PARTITION BY dept_num ORDER BY salary) AS ntile
FROM employee_contract
ORDER BY dept_num;
```

Analytic Function – PERCENT_RANK

It is similar to CUME_DIST, but it uses rank values rather than row counts in its numerator as *total number of rows - 1* divided by *current rank - 1*.

```
SELECT name, dept_num, salary,
RANK() OVER (PARTITION BY dept_num ORDER BY salary) AS rank,
DENSE_RANK() OVER (PARTITION BY dept_num ORDER BY salary)
AS dense_rank,
ROW_NUMBER() OVER () AS row_num,
ROUND((CUME_DIST() OVER (PARTITION BY dept_num
ORDER BY salary)), 1) AS cume_dist,
PERCENT_RANK() OVER(PARTITION BY dept_num
ORDER BY salary) AS percent_rank,
NTILE(4) OVER(PARTITION BY dept_num ORDER BY salary) AS ntile
FROM employee_contract
ORDER BY dept_num;
```

Analytic Function – ROW_NUMBER

It assigns a unique sequence number starting from 1 to each row according to the partition and order specification

```
SELECT name, dept_num, salary,
RANK() OVER (PARTITION BY dept_num ORDER BY salary) AS rank,
DENSE_RANK() OVER (PARTITION BY dept_num ORDER BY salary)
AS dense_rank,
ROW_NUMBER() OVER () AS row_num,
ROUND((CUME_DIST() OVER (PARTITION BY dept_num
ORDER BY salary)), 1) AS cume_dist,
PERCENT_RANK() OVER(PARTITION BY dept_num
ORDER BY salary) AS percent_rank,
NTILE(4) OVER(PARTITION BY dept_num ORDER BY salary) AS ntile
FROM employee_contract
ORDER BY dept_num;
```

Analytic Function - NTILE

It divides an ordered dataset into number of buckets and assigns an appropriate bucket number to each row

```
SELECT name, dept_num, salary,
RANK() OVER (PARTITION BY dept_num ORDER BY salary) AS rank,
DENSE_RANK() OVER (PARTITION BY dept_num ORDER BY salary)
AS dense_rank,
ROW_NUMBER() OVER () AS row_num,
ROUND((CUME_DIST() OVER (PARTITION BY dept_num
ORDER BY salary)), 1) AS cume_dist,
PERCENT_RANK() OVER(PARTITION BY dept_num
ORDER BY salary) AS percent_rank,
NTILE(4) OVER(PARTITION BY dept_num ORDER BY salary) AS ntile
FROM employee_contract
ORDER BY dept_num;
```

Analytic Function - LEAD

The LEAD function, lead(value_expr[,offset[,default]]), is used to return data from the next row.

```
SELECT name, dept_num, salary,
LEAD(salary, 2) OVER(PARTITION BY dept_num
ORDER BY salary) AS lead,
LAG(salary, 2, 0) OVER(PARTITION BY dept_num
ORDER BY salary) AS lag,
FIRST_VALUE(salary) OVER (PARTITION BY dept_num
ORDER BY salary) AS first_value,
LAST_VALUE(salary) OVER (PARTITION BY dept_num
ORDER BY salary) AS last_value_default,
LAST_VALUE(salary) OVER (PARTITION BY dept_num
ORDER BY salary
RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
AS last_value
FROM employee_contract ORDER BY dept_num;
```

Analytic Function - LAG

The LAG function, lag(value_expr[,offset[,default]]), is used to access data from a previous row.

```
SELECT name, dept_num, salary,
LEAD(salary, 2) OVER(PARTITION BY dept_num
ORDER BY salary) AS lead,
LAG(salary, 2, 0) OVER(PARTITION BY dept_num
ORDER BY salary) AS lag,
FIRST_VALUE(salary) OVER (PARTITION BY dept_num
ORDER BY salary) AS first_value,
LAST_VALUE(salary) OVER (PARTITION BY dept_num
ORDER BY salary) AS last_value_default,
LAST_VALUE(salary) OVER (PARTITION BY dept_num
ORDER BY salary
RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
AS last_value
FROM employee_contract ORDER BY dept_num;
```

Analytic Function – FIRST_VALUE

It returns the first result from an ordered set.

```
SELECT name, dept_num, salary,
LEAD(salary, 2) OVER(PARTITION BY dept_num
ORDER BY salary) AS lead,
LAG(salary, 2, 0) OVER(PARTITION BY dept_num
ORDER BY salary) AS lag,
FIRST_VALUE(salary) OVER (PARTITION BY dept_num
ORDER BY salary) AS first_value,
LAST_VALUE(salary) OVER (PARTITION BY dept_num
ORDER BY salary) AS last_value_default,
LAST_VALUE(salary) OVER (PARTITION BY dept_num
ORDER BY salary
RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
AS last_value
FROM employee_contract ORDER BY dept_num;
```

Analytic Function – LAST_VALUE

It returns the last result from an ordered set.

default windowing clause is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW, which in this example means the current row will always be the last value. Changing the windowing clause to RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING gives us the result we probably expected

Analytic Function - Windows

The [<window_clause>] clause is used to further subpartition the result and apply the analytic functions. There are two types of windows: row type window and range type window.

The RANK, NTILE, DENSE_RANK, CUME_DIST, PERCENT_RANK, LEAD, LAG, and ROW_NUMBER functions do not support being used with a window clause yet.

For row type windows, the definition is in terms of row numbers before or after the current row. The general syntax of the row window clause is as follows:

ROWS BETWEEN <start_expr> AND <end_expr>

Analytic Function - Windows

```
SELECT name, dept_num, salary AS sal,
MAX(salary) OVER (PARTITION BY dept_num ORDER BY
name ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) win1,
MAX(salary) OVER (PARTITION BY dept_num ORDER BY
name ROWS BETWEEN 2 PRECEDING AND UNBOUNDED FOLLOWING) win2,
MAX(salary) OVER (PARTITION BY dept_num ORDER BY
name ROWS BETWEEN 1 PRECEDING AND 2 FOLLOWING) win3,
MAX(salary) OVER (PARTITION BY dept_num ORDER BY
name ROWS BETWEEN 1 FOLLOWING AND 2 FOLLOWING) win5,
MAX(salary) OVER (PARTITION BY dept_num ORDER BY
name ROWS BETWEEN CURRENT ROW AND CURRENT ROW) win7,
MAX(salary) OVER (PARTITION BY dept_num ORDER BY
name ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING) win8,
MAX(salary) OVER (PARTITION BY dept_num ORDER BY
name ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) win9,
MAX(salary) OVER (PARTITION BY dept_num ORDER BY
name ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) win10,
MAX(salary) OVER (PARTITION BY dept_num ORDER BY
name ROWS BETWEEN UNBOUNDED PRECEDING AND 1 FOLLOWING) win11,
MAX(salary) OVER (PARTITION BY dept_num ORDER BY
name ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED
FOLLOWING) win12,
MAX(salary) OVER (PARTITION BY dept_num ORDER BY
name ROWS 2 PRECEDING) win13
FROM employee_contract
ORDER BY dept_num, name;
```

Analytic Function - Windows

```
SELECT name, dept_num, salary,
MAX(salary) OVER w1 AS win1,
MAX(salary) OVER w1 AS win2,
MAX(salary) OVER w1 AS win3
FROM employee_contract
ORDER BY dept_num, name
WINDOW
w1 AS (PARTITION BY dept_num ORDER BY name ROWS BETWEEN
2 PRECEDING AND CURRENT ROW),
w2 AS w3,
w3 AS (PARTITION BY dept_num ORDER BY name ROWS BETWEEN
1 PRECEDING AND 2 FOLLOWING);
```

Analytic Function - Windows

```
SELECT name, salary, start_year,
MAX(salary) OVER (PARTITION BY dept_num ORDER BY
start_year RANGE BETWEEN 2 PRECEDING AND CURRENT ROW) win1
FROM
(
  SELECT name, salary, dept_num,
  YEAR(start_date) AS start_year
  FROM employee_contract
) a;
```

Sampling

When data volume is extra large, we may need to find a subset of data to speed up data analysis. Here it comes to a technique used to select and analyze a subset of data in order to identify patterns and trends. In Hive, there are three ways of sampling data: random sampling, bucket table sampling, and block sampling.

Random Sampling

Uses the RAND() function and LIMIT keyword to get the sampling of data as shown in the following example. The DISTRIBUTE and SORT keywords are used here to make sure the data is also randomly distributed among mappers and reducers efficiently. The ORDER BY RAND() statement can also achieve the same purpose, but the performance is not good.

SELECT * FROM <Table_Name> DISTRIBUTE BY RAND() SORT BY RAND()
LIMIT <N rows to sample>;

Bucket Table Sampling

A special sampling optimized for bucket tables as shown in the following syntax and example. The col name value specifies the column where to sample the data. The RAND() function can also be used when sampling is on the entire rows. If the sample column is also the CLUSTERED BY column, the TABLESAMPLE statement will be more efficient.

Bucket Table Sampling

```
CREATE TABLE employee_id_buckets  (
  name string,   employee_id int,   work_place ARRAY<string>,
  sex_age STRUCT<sex:string,age:int>,   skills_score MAP<string,int>,
  depart_title MAP<STRING,ARRAY<STRING>>
) CLUSTERED BY (employee_id) INTO 2 BUCKETS
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
COLLECTION ITEMS TERMINATED BY ','
MAP KEYS TERMINATED BY ':';

set hive.enforce.bucketing = true;

INSERT OVERWRITE TABLE employee_id_buckets SELECT * FROM employee_id;
```

If Run into OutOfMemory error since using Tez engine, then SET hive.tez.java.opts=-Xmx1024m

```
SELECT name FROM employee_id_buckets TABLESAMPLE(BUCKET 1 OUT OF 2 ON rand()) a;
```

Block Sampling

Allows Hive to randomly pick up *N* rows of data, percentage (*n* percentage) of data size, or *N* byte size of data. The sampling granularity is the HDFS block size. Its syntax and examples are as follows:

- Block sampling - Sample by rows

```
SELECT name FROM employee_id_buckets TABLESAMPLE(4 ROWS) a;
```

- Sample by percentage of data size

```
SELECT name FROM employee_id_buckets TABLESAMPLE(10 PERCENT) a;
```

- Sample by data size

```
SELECT name FROM employee_id_buckets TABLESAMPLE(3M) a;
```

Hive Integration with Hcatalog

HCatalog is a metadata management system for Hadoop data. It stores consistent schema information for Hadoop ecosystem tools, such as Pig, Hive, and MapReduce.

HCatalog is built on top of the Hive metastore and incorporates support for Hive DDL. It provides read and write interfaces and HCatLoader and HCatStorer, for Pig, by implementing Pig's load and store interfaces, respectively

HCatalog provides a REST API from a component called WebHCat so that HTTP requests can be made to access the metadata of Hadoop MapReduce/Yarn, Pig, Hive, and HCatalog DDL from other applications

Hive Integration with Hcatalog

select a.driverId,a.driverName,a.eventType,b.certified from truck_events a join drivers b ON (a.driverId = b.driverId);

Go to PIG Shell:

```
a = LOAD 'drivers' using org.apache.hive.hcatalog.pig.HCatLoader();
b = LOAD 'truck_events' using org.apache.hive.hcatalog.pig.HCatLoader();
c = join b by driverid, a by driverid;
dump c;
```

WebHCat

http://127.0.0.1:50111/templeton/v1/ddl/database/default/table/employee?user.name=hive

Hive Integration with Oozie

Oozie (see http://oozie.apache.org/) is an open source workflow coordination and schedule service to manage data processing jobs. Oozie workflow jobs are defined in a series of nodes in a **Directed Acyclic Graph** (**DAG**). Acyclic here means that there are no loops in the graph and all nodes in the graph flow in one direction without going back.

Oozie workflows contain either the control flow node or action node:

- **Control flow node**: This either defines the start, end, and failed node in a workflow or controls the workflow execution path such as decision, fork, and join nodes.

- **Action node**: This defines the core data processing action job such as MapReduce, Hadoop filesystem, Hive, Pig, Java, Shell, e-mail, and Oozie subworkflows. Additional types of actions are also supported by developing extensions.

Hive Integration with HBase

```
CREATE TABLE IF NOT EXISTS pagecounts (projectcode STRING, pagename STRING, pageviews STRING, bytes STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ' '
LINES TERMINATED BY '\n'
STORED AS TEXTFILE
LOCATION '/user/root/pagecounts';

select INPUT__FILE__NAME from pagecounts limit 10;

CREATE VIEW IF NOT EXISTS pgc (rowkey, pageviews, bytes) AS
SELECT concat_ws('/',
                projectcode,
                concat_ws('/',
                pagename,
                regexp_extract(INPUT__FILE__NAME, 'pagecounts-(\\d{8}-\\d{6})', 1))),
                                                        pageviews, bytes
FROM pagecounts;
```

Hive Integration with HBase

```
CREATE TABLE IF NOT EXISTS pagecounts_hbase (rowkey STRING, pageviews STRING, bytes STRING)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ('hbase.columns.mapping' = ':key,0:PAGEVIEWS,0:BYTES')
TBLPROPERTIES ('hbase.table.name' = 'PAGECOUNTS');

FROM pgc INSERT INTO TABLE pagecounts_hbase SELECT pgc.* WHERE rowkey LIKE 'en/q%' LIMIT 10;
```

Go to HBASE SHELL, run the following command

```
scan 'PAGECOUNTS'
```

Go to Phoenix Zeepline, and create new note:

```
@phoenix
CREATE VIEW "PAGECOUNTS" (pk VARCHAR PRIMARY KEY,
"0".PAGEVIEWS VARCHAR,
"0".BYTES VARCHAR)
```

Hive Integration with Zeppline

Go to Zeppline http://127.0.0.1:9995/

Create new note and add the following:

```
%hive(default)
select * from employee
```

Hive Integration with Sqoop

- sqoop import --hive-import --hive-overwrite --connect jdbc:mysql://localhost/sqoop_test --table stocks --fetch-size 10 --username hip_sqoop_user -P
- sqoop import --hive-import --hive-overwrite --hive-partition-key edate --hive-partition-value "20160101" --target-dir /user/root/stocks/edate=20160101 --connect jdbc:mysql://localhost/sqoop_test --table stocks --fetch-size 10 --username hip_sqoop_user -P
- sqoop import --hive-import --hive-partition-key edate --hive-partition-value "20160102" --target-dir /user/root/stocks/edate=20160102 --connect jdbc:mysql://localhost/sqoop_test --table stocks --fetch-size 10 --username hip_sqoop_user -P
- sqoop import --hive-import --hive-partition-key edate --hive-partition-value "20160103" --target-dir /user/root/stocks/edate=20160103 --connect jdbc:mysql://localhost/sqoop_test --table stocks --fetch-size 10 --username hip_sqoop_user -P

Hive Integration with Zookeeper

ZooKeeper is a centralized service for configuration management and the synchronization of various aspects of naming and coordination. It manages a naming registry and effectively implements a system for managing the various statically and dynamically named objects in a hierarchical system. It also enables coordination and control to the shared resources, such as files and data, which are manipulated by multiple concurrent processes.

Hive does not natively support concurrency access and locking mechanisms. Hive relies on ZooKeeper for locking the shared resources since Hive 0.7.0.

There are two types of locks provided by Hive through Zookeeper and they are as follows:

- Shared lock
- Exclusive lock

Hive Integration with Zookeeper

To enable locking in Hive, we need to make sure ZooKeeper is installed and configured.
Then, configure the following properties in Hive's hive-site.xml file:

<property>
<name>hive.support.concurrency</name>
<description>Enable Hive's Table Lock Manager Service</description>
<value>true</value>
</property>

- LOCK TABLE employee shared;
- SHOW LOCKS employee EXTENDED;
- UNLOCK TABLE employee;
- SHOW LOCKS;
- LOCK TABLE employee exclusive;
- SELECT * FROM employee;

Hive Integration with Spark

```
val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)

sqlContext.sql("FROM employee SELECT *").collect().foreach(println)

sqlContext.sql("FROM employee SELECT count(*)").collect().foreach(println)

sqlContext.sql("FROM employee_external_test1 SELECT
*").collect().foreach(println)

sqlContext.sql("MSCK REPAIR TABLE employee_external_test1")
```

Hive Integration with RDBMS

```
add jar /training/apps/hive/lib/hive-jdbc-handler-0.8.1-wso2v7.jar;

CREATE EXTERNAL TABLE business
ROW FORMAT SERDE 'org.wso2.carbon.hadoop.hive.jdbc.storage.JDBCDataSerDe'
with serdeproperties (
        "escaped" = "true"
)
STORED BY 'org.wso2.carbon.hadoop.hive.jdbc.storage.JDBCStorageHandler'
TBLPROPERTIES (
        "mapred.jdbc.driver.class"="com.mysql.jdbc.Driver",
        "mapred.jdbc.url"="jdbc:mysql://localhost:3306/test",
        "mapred.jdbc.username"="root",
        "mapred.jdbc.input.table.name"="business",
        "mapred.jdbc.output.table.name"="business"
);
```

Hive Integration with RDBMS

```
CREATE EXTERNAL TABLE PhonebrandTable(brand STRING,totalOrders INT, totalQuantity INT)
STORED BY 'org.wso2.carbon.hadoop.hive.jdbc.storage.JDBCStorageHandler'
TBLPROPERTIES (
        "mapred.jdbc.driver.class"="com.mysql.jdbc.Driver",
        "mapred.jdbc.url"="jdbc:mysql://localhost:3306/test",
        "mapred.jdbc.username"="root",
        "mapred.jdbc.password"="",
        "hive.jdbc.update.on.duplicate" = "true",
        "hive.jdbc.primary.key.fields" = "brand",
        "hive.jdbc.table.create.query" = "CREATE TABLE brandSummary (brand VARCHAR(100) NOT NULL PRIMARY KEY, totalOrders INT, totalQuantity INT)"
);
```

Hive Integration with Tableau

Hive Integration with Talend Open Studio

Hive Integration with Datameer

Hive Integration with Qlikview

Hive Integration with Excel

Hive Integration with Kylin

Hive LLAP

Hive Security Authentication

Authentication is the process of verifying the identity of a user by obtaining the user's credentials. Hive has offered authentication since HiveServer2

Kerberos

Kerberos is a network authentication protocol developed by MIT as part of Project Athena. It uses time-sensitive tickets that are generated using symmetric key cryptography to securely authenticate a user in an unsecured network environment. Kerberos is derived from Greek mythology, where Kerberos was the three-headed dog that guarded the gates of Hades. The three-headed part refers to the three parties involved in the Kerberos authentication process: client, server, and **Key Distribution Center** (**KDC**). All clients and servers registered to KDC are known as a realm, which is typically the domain's DNS name in all caps.

Metadata Store Authentication

To force clients to authenticate with the Hive Metastore server using Kerberos, we can set the following properties in the hive-site.xml file:

- Enable the **Simple Authentication and Security Layer** (**SASL**) framework to enforce client Kerberos authentication via setting true to hive.metastore.sasl.enabled
- Specify the Kerberos keytab that is generated via setting keytab file to hive.metastore.kerberos.keytab.file
- Specify the Kerberos principal pattern string via setting principal to hive.metastore.kerberos.principal

HiveServer2 Authentication

HiveServer2 supports the following authentications:

- **None authentication**
- **Kerberos authentication**
- **LDAP authentication**

HiveServer2 Authentication

To configure HiveServer2 to use one of these authentication modes, we can set the proper properties in hive_site.xml

None authentication

hive.server2.authentication: none

Kerberos authentication

hive.server2.authentication:KERBEROS
hive.server2.authentication.kerberos.keytab
hive.server2.authentication.kerberos.principal

HiveServer2 Authentication

To configure HiveServer2 to use one of these authentication modes, we can set the proper properties in hive_site.xml

LDAP authentication

hive.server2.authentication:LDAP

hive.server2.authentication.ldap.url

hive.server2.authentication.ldap.Domain

ive.server2.authentication.ldap.baseDN

Hive Authorization

Authorization is verifying if a user has permission to perform a certain action, and not about Authentication. Three modes of Hive authorization are available to satisfy different use cases.

- Default Hive Authorization (Legacy Mode)
- Storage Based Authorization in the Metastore Server
- SQL Standards Based Authorization in HiveServer2

Hive Legacy Authorization

Hive Default Authorization is the authorization mode that has been available in earlier versions of Hive. However, this mode does not have a complete access control model, leaving many security gaps unaddressed. For example, the permissions needed to grant privileges for a user are not defined, and any user can grant themselves access to a table or database.

It can be mainly used to prevent good users from accidentally doing bad things rather than preventing malicious users' operations.

Hive Legacy Authorization

- In order to use Hive authorization, there are two parameters that should be set in hive-site.xml:

  hive.security.authorization.enabled: true
  hive.security.authorization.createtable.owner.grants: ALL

- At the core of Hive's authorization system are users, groups, and roles. Roles allow administrators to give a name to a set of grants which can be easily reused. A role may be assigned to users, groups, and other roles.

Hive Legacy Authorization

- **Create/Drop Role**

CREATE ROLE role_name DROP ROLE role_name

- **Grant/Revoke Roles**

GRANT ROLE role_name [, role_name] ... TO principal_specification [, principal_specification] ... [WITH ADMIN OPTION]

REVOKE [ADMIN OPTION FOR] ROLE role_name [, role_name] ... FROM principal_specification [, principal_specification] ...

- **Viewing Granted Roles**

SHOW ROLE GRANT principal_specification

Hive Legacy Authorization

- **Privileges**

- ALL - Gives users all privileges
- ALTER - Allows users to modify the metadata of an object
- UPDATE - Allows users to modify the physical data of an object
- CREATE - Allows users to create objects. For a database, this means users can create tables, and for a table, this means users can create partitions
- DROP - Allows users to drop objects
- INDEX - Allows users to create indexes on an object (Note: this is not currently implemented)
- LOCK - Allows users to lock or unlock tables when concurrency is enabled
- SELECT - Allows users to access data for objects
- SHOW_DATABASE - Allows users to view available databases

Hive Legacy Authorization

- **Grant/Revoke Privileges**

GRANT priv_type [(column_list)] [, priv_type [(column_list)]] ... [ON object_specification] TO principal_specification [, principal_specification] ... [WITH GRANT OPTION]

REVOKE [GRANT OPTION FOR] priv_type [(column_list)] [, priv_type [(column_list)]] ... [ON object_specification] FROM principal_specification [, principal_specification] ...

REVOKE ALL PRIVILEGES, GRANT OPTION FROM user [, user] ...

Hive Storage-based Authorization

- By enabling this mode, you can use this single source for truth and have a consistent data and metadata authorization policy. To control metadata access on the metadata objects such as Databases, Tables and Partitions, it checks if you have permission on corresponding directories on the file system.
- It relies on the authorization provided by the storage layer HDFS.
- This mode checks Hive user permissions against the POSIX permissions on the corresponding file directories in HDFS.
- Considering its implementation, the storage-based authorization mode only offers authorization at the level of Hive databases, tables, and partitions rather than column and row level. With dependency on the HDFS permissions, it lacks the flexibility to manage the authorization through HQL statements.

Hive Storage-based Authorization

To enable Hive metastore server security, set these parameters in hive-site.xml:

- hive.metastore.pre.event.listeners - Set to org.apache.hadoop.hive.ql.security.authorization.AuthorizationPreEventListener. This turns on metastore-side security.
- hive.security.metastore.authorization.manager - Set to org.apache.hadoop.hive.ql.security.authorization.StorageBasedAuthorizationProvider. This tells Hive which metastore-side authorization provider to use.
- hive.security.metastore.authenticator.manager - Set to org.apache.hadoop.hive.ql.security.HadoopDefaultMetastoreAuthenticator
- hive.security.metastore.authorization.auth.reads - When this is set to true, Hive metastore authorization also checks for read access

Hive SQL-based Authorization

Although Storage Based Authorization can provide access control at the level of Databases, Tables and Partitions, it can not control authorization at finer levels such as columns and views because the access control provided by the file system is at the level of directory and files.

A prerequisite for fine grained access control is a data server that is able to provide just the columns and rows that a user needs (or has) access to. In the case of file system access, the whole file is served to the user. HiveServer2 satisfies this condition, as it has an API that understands rows and columns (through the use of SQL), and is able to serve just the columns and rows that your SQL query asked for.

Hive SQL-based Authorization

To enable SQL standard-based mode authorization, we can set the following properties in the hive-site.xml file:

- hive.server2.enable.doAs: false
- hive.users.in.admin.role: administrator
- hive.security.authorization.enabled: true
- hive.security.authorization.manager:org.apache.hadoop.hive.ql.security.authorization.plugin.sql
- hive.security.authenticator.manager: org.apache.hadoop.hive.ql.security.SessionStateUserAuthenticator

Apache Ranger and Hive

- Apache Ranger provides centralized security administration for Hadoop, and it enables fine grain access control and deep auditing for Apache components such as Hive, HBase, HDFS, Storm and Knox, which are installed with an Ranger plugin used to intercept authorization requests for that component, as shown in the following illustration.

- Administrators who are responsible for managing access to multiple components are strongly encouraged to use the Ranger Policy Manager to configure authentication for Hive rather than using storage-based or SQL standard-based authorization to take advantage of the ease-of-use provided by the Policy Manager.

Apache Ranger and Hive

However, there are two primary use cases where administrators might choose to integrate Ranger with SQL standard-based authorization provided by Hive:

- An administrator is responsible for Hive authentication but not authentication for other components
- An administrator wants row-level authentication for one or more table views

Apache Ranger and Hive

There are two notable differences between Ranger authorization and SQL standard-based authorization:

- Ranger does not have the concept of a role. Instead, Ranger translates roles into users and groups.

- The ADMIN permission in Ranger is the equivalent to the WITH GRANT OPTION in SQL standard-based authorization. However, the ADMIN permission gives the grantee the ability to grant all permissions rather than just the permissions possessed by the grantor. With SQL standard-based authorization, the WITH GRANT OPTION applies only to permissions possessed by the grantor

Apache Sentry and Hive

Sentry is a highly modular system for providing centralized, fine-grained, role-based authorization to both data and metadata stored on an Apache Hadoop cluster. It can be integrated with Hive to deliver advanced authorization controls.

Hive Encryption

- For sensitive and legally protected data, it is required to store the data in encrypted format in the file system. However, Hive does not natively support encryption and decryption yet.
- The new HDFS encryption offers great transparent encryption and decryption of data on HDFS.
- The best solution for now is to use Hive UDF to plug in encryption and decryption implementations on selected columns or partial data in the Hive tables.

Extending Hive

Although Hive has many built-in functions, users sometimes will need power beyond that provided by built-in functions. For these instances, Hive offers the following main areas where its functionalities can be extended

Extending Hive with UDF

These are regular user-defined functions that operate row-wise and output one result for one row, such as most built-in mathematic and string functions.

```
public class udf_name extends UDF {
public String evaluate(){
/*
* Do something here
*/
return "return the udf result";
}
//override is supported
public String evaluate(<Type_arg1> arg1,..., <Type_argN> argN){
/*
* Do something here
*/
return "return the udf result";
}
}
```

Extending Hive with UDF

Add the JAR to the Hive environment using one of the following options (option 3 or 4 is recommended):

Option 1: Run ADD JAR /home/dayongd/hive/lib/toupper.jar in the Hive CLI. This is only valid for the current session, but does not work for ODBC connections.

Option 2: Add ADD JAR /home/dayongd/hive/lib/toupper.jar in /home/$USER/.hiverc (we can create the file if it is not there). In this case, the file needs to be deployed to every node from where we might launch the Hive shell. This is only valid for the current session, but does not work for ODBC connections.

Option 3: Add the following configuration in the hive-site.xml file:

<property>

<name>hive.aux.jars.path</name>

<value>file:///home/dayongd/hive/lib/toupper.jar</value>

</property>

Option 4: Copy and paste the JAR file to the /${HIVE_HOME}/auxlib/ folder (create it if it does not exist).

Extending Hive with UDF

- **Create the function. We can create a temporary function that is only valid in the current Hive session as follows:**

```
CREATE TEMPORARY FUNCTION toUpper AS 'com.packtpub.hive.essentials.hiveudf.toupper';
```

- **Create permanent function:**

```
CREATE FUNCTION toUpper AS 'com.packtpub.hive.essentials.hiveudf.ToUpper' USING JAR
'hdfs:///path/to/jar';
```

- **Verify the function:**

```
SHOW FUNCTIONS ToUpper;
DESCRIBE FUNCTION ToUpper;
DESCRIBE FUNCTION EXTENDED ToUpper;
```

Extending Hive with UDF

- **Use the UDF in HQL:**

SELECT toUpper(name) FROM employee LIMIT 1000;

- **Drop the function when needed:**

DROP TEMPORARY FUNCTION IF EXISTS toUpper;

Extending Hive with UDAF

These are user-defined aggregating functions that operate row-wise or group-wise and output one row or one row for each group as a result, such as the MAX and COUNT built-in functions.

```
public final class udaf_name extends UDAF {
public static class UDAFExampleAvgEvaluator implements UDAFEvaluator {
        public void init() {}
        public boolean iterate(){}
        public UDAFState terminatePartial() {}
        public boolean merge(UDAFState o) {}
        public long terminate() {}
}
}
```

Extending Hive with UDTF

These are user-defined table-generating functions that also operate rowwise, but they produce multiple rows/tables as a result, such as the EXPLODE function. UDTF can be used either after SELECT or after the LATERAL VIEW statement.

```
public class udtf_name extends GenericUDTF {
}
```

Extending Hive with SerDe

SerDe stands for Serializer and Deserializer. It is the technology that Hive uses to process records and map them to column data types in Hive tables. To explain the scenario of using SerDe, we need to understand how Hive reads and writes data. The process to read data is as follows:

- Data is read from HDFS.
- Data is processed by the INPUTFORMAT implementation, which defines the input data split and key/value records. In Hive, we can use CREATE TABLE… STORED AS <FILE_FORMAT> (see Chapter 7, Performance Considerations, for available file formats) to specify which INPUTFORMAT it reads from.
- The Java Deserializer class defined in SerDe is called to format the data into a record that maps to column and data types in a table.

Extending Hive with SerDe

The following are examples of SerDe:

- LazySimpleSerDe: CREATE TABLE test_serde_lz STORED AS TEXTFILE AS SELECT name from employee;

- ColumnarSerDe:CREATE TABLE test_serde_cs ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.columnar.ColumnarSerDe' STORED AS RCFile AS SELECT name from employee;

- RegexSerDe:CREATE TABLE test_serde_rex(name string,sex string,age string) ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'
- WITH SERDEPROPERTIES('input.regex' = '([^,]*),([^,]*),([^,]*)','output.format.string' = '%1$s %2$s %3$s') STORED AS TEXTFILE;

Extending Hive with SerDe

- HBaseSerDe: CREATE TABLE test_serde_hb(id string, name string, sex string, age string) ROW FORMAT SERDE 'org.apache.hadoop.hive.hbase.HBaseSerDe' STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler' WITH SERDEPROPERTIES ("hbase.columns.mapping"=":key,info:name,info:sex,info:age")
  TBLPROPERTIES("hbase.table.name" = "test_serde");

- AvroSerDe: CREATE TABLE test_serde_avro(name string, sex string, age string) ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe' STORED AS INPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat' OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'

Extending Hive with SerDe

- ParquetHiveSerDe: CREATE TABLE test_serde_parquet STORED AS PARQUET AS SELECT name from employee;
- OpenCSVSerDe: CREATE TABLE test_serde_csv(name string, sex string, age string) ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde' STORED AS TEXTFILE;
- JSONSerDe: CREATE TABLE test_serde_js(name string, sex string,age string) ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe' STORED AS TEXTFILE;

Extending Hive with InputFormat

HDFS files-->InputFileFormat--> --> Deserializer --> Row object

Extending Hive with OutputFormat

Row object -->Serializer --> --> OutputFileFormat --> HDFS files

Extending Hive with Storage Handler

The motivation is to make it possible to allow Hive to access data stored and managed by other systems in a modular, extensible fashion.

Hive storage handler support builds on existing extensibility features in both Hadoop and Hive:

- input formats
- output formats
- serialization/deserialization libraries

Supported Storage Handlers:

- HBase, Cassandra, JDBC, MongoDB, and Google Spreadsheets.

Extending Hive with Storage Handler

Example:

```
CREATE TABLE hbase_table_1(key int, value string)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES (
"hbase.columns.mapping" = "cf:string",
"hbase.table.name" = "hbase_table_0"
);
```

DROP TABLE works as usual, but ALTER TABLE is not yet supported for non-native tables.

Extending Hive with Storage Handler

Storage Handler Interface

```
public interface HiveStorageHandler extends Configurable {
  public Class<? extends InputFormat> getInputFormatClass();
  public Class<? extends OutputFormat> getOutputFormatClass();
  public Class<? extends SerDe> getSerDeClass();
  public HiveMetaHook getMetaHook();
  public void configureTableJobProperties(
    TableDesc tableDesc,
    Map<String, String> jobProperties);
}
```

Extending Hive with Streaming

Hive can also leverage the streaming feature in Hadoop to transform data in an alternative way. The streaming API opens an I/O pipe to an external process (script). Then, the process reads data from the standard input and writes the results out through the standard output. In Hive, we can use TRANSFORM clauses in HQL directly, and embed the mapper and the reducer scripts written in commands, shell scripts, Java, or other programming languages.

```
FROM (
FROM src
SELECT TRANSFORM '(' expression (',' expression)* ')'
(inRowFormat)?
USING 'map_user_script'
(AS colName (',' colName)*)?
(outRowFormat)? (outRecordReader)?
(CLUSTER BY?|DISTRIBUTE BY? SORT BY?) src_alias
)
SELECT TRANSFORM '(' expression (',' expression)* ')'
(inRowFormat)?
USING 'reduce_user_script'
(AS colName (',' colName)*)?
(outRowFormat)? (outRecordReader)?
```

Hive and Machine Learning

Hive Performance

Since Hive 0.13.0, Hive includes the following new features for performance optimizations:

- Tez:

Tez is an application framework built on Yarn that can execute complex directed acyclic graphs (DAGs) for general data-processing tasks. Tez further splits map and reduce jobs into smaller tasks and combines them in a flexible and efficient way for execution. Tez is considered a flexible and powerful successor to the MapReduce framework. To configure Hive to use Tez, we need to overwrite the following settings from the default MapReduce:

```
SET hive.execution.engine=tez;
```

Hive Performance

- Vectorization:

Vectorization optimization processes a larger batch of data at the same time rather than one row at a time, thus significantly reducing computing overhead.

Each batch consists of a column vector that is usually an array of primitive types.

Operations are performed on the entire column vector, which improves the instruction pipelines and cache usage. Files must be stored in the Optimized Row Columnar (ORC) format in order to use vectorization. To enable vectorization, we need to do the following setting:

```
SET hive.vectorized.execution.enabled=true;
```

Hive Performance

- Spark:

Hive on Spark provides Hive with the ability to utilize Apache Spark as its execution engine. Configuration for hive on spark as following:

  - set spark.home=/usr/hdp/2.4.0.0-169/spark;
  - SET hive.execution.engine=spark;
  - set spark.master=yarn-client;
  - set spark.eventLog.enabled=true;
  - set spark.executor.memory=512m;

Added spark-assembly jar file as shown below:

ADD jar /usr/hdp/2.4.0.0-169/spark/lib/spark-assembly-1.6.0.2.4.0.0-169-hadoop2.7.1.2.4.0.0-169.jar to Hive Lib;

Hive Performance

- LLAP:

Hive has become significantly faster thanks to various features and improvements that were built by the community in recent years,

including Tez and Cost-based-optimization. The following were needed to take Hive to the next level:

  - Asynchronous spindle-aware IO
  - Pre-fetching and caching of column chunks
  - Multi-threaded JIT-friendly operator pipelines

LLAP provides a hybrid execution model which consists of a long-lived daemon replacing direct interactions with the HDFS DataNode and a tightly integrated DAG-based framework. Functionality such as caching, pre-fetching, some query processing and access control are moved into the daemon. Small/short queries are largely processed by this daemon directly, while any heavy lifting will be performed in standard YARN containers.

Performance Utility - Explain

Hive provides an EXPLAIN command to return a query execution plan without running the query. We can use an EXPLAIN command for queries if we have a doubt or a concern about performance. The EXPLAIN command will help to see the difference between two or more queries for the same purpose. The syntax for EXPLAIN is as follows:

EXPLAIN [EXTENDED|DEPENDENCY|AUTHORIZATION] hive_query

EXPLAIN SELECT sex_age.sex, count(*) FROM employee_partitioned WHERE year=2014 GROUP BY sex_age.sex LIMIT 2;

Performance Utility - Analyze

Hive statistics are a collection of data that describe more details, such as the number of rows, number of files, and raw data size, on the objects in the Hive database. Statistics is a metadata of Hive data. Hive supports statistics at the table, partition, and column level. These statistics serve as an input to the Hive Cost-Based Optimizer (CBO), which is an optimizer to pick the query plan with the lowest cost in terms of system resources required to complete the query.

The statistics are gathered through the ANALYZE statement since Hive 0.10.0 on tables, partitions, and columns as given in the following examples:

Performance Utility - Analyze

ANALYZE TABLE employee COMPUTE STATISTICS;

ANALYZE TABLE employee_partitioned PARTITION(year=2014, month=12) COMPUTE STATISTICS;

ANALYZE TABLE employee_id COMPUTE STATISTICS FOR COLUMNS employee_id;

Performance Consideration using Partition

Hive partitioning is one of the most effective methods to improve the query performance on larger tables. The query with partition filtering will only load the data in the specified partitions (subdirectories), so it can execute much faster than a normal query that filters by a non-partitioning field. The selection of partition key is always an important factor for performance. It should always be a low cardinal attribute to avoid many subdirectories overhead. The following are some commonly used dimensions as partition keys:

- **Partitions by date and time**: Use date and time, such as year, month, and day (even hours), as partition keys when data is associated with the time dimension
- **Partitions by locations**: Use country, territory, state, and city as partition keys when data is location related
- **Partitions by business logics**: Use department, sales region, applications, customers, and so on as partitioned keys when data can be separated evenly by some business logic

Performance Consideration using Bucket

One thing buckets are used for is to increase load performance

- SELECT performance (predicate pushdown)

Buckets can help with the predicate pushdown since every value belonging to one value will end up in one bucket. So if you bucket by 31 days and filter for one day Hive will be able to more or less disregard 30 buckets. Obviously this doesn't need to be good since you often WANT parallel execution like aggregations. So it depends on your query if this is good. It might be better to sort by day and bucket by something like customer id if you have to have buckets for some of the other reasons.

Performance Consideration using Bucket

- Join Performance (bucket join)

Buckets can lead to efficient joins if both joined tables are bucketed on the join key since he only needs to join bucket with bucket. This was big in the old times but is not that applicable anymore with cost based optimization in newer Hive versions (since the optimizer already is very good at choosing map side vs shuffle join and a bucket join can actually stop him from using the better one.

- Sampling performance

Some sample operations can get faster with buckets.

Performance Consideration using Bucket

So to summarize buckets are a bit of an older concept and I wouldn't use them unless I have a clear case for it. The join argument is not that applicable anymore, the increased load performance also is not always relevant since you normally load single partitions where a map only load is often best. Select pushdown can be enhanced but also hindered depending how you do it and a SORT by is normally better during load (see document).
And I think sampling is a bit niche.

Performance Consideration using Index

Index is very common with RDBMS when we want to speed access to a column or set of columns. Hive supports index creation on tables/partitions since Hive 0.7.0. The index in Hive provides key-based data view and better data access for certain operations, such as WHERE, GROUP BY, and JOIN. We can use index is a cheaper alternative than full table scans.

Performance Consideration choosing File Format

Hive supports TEXTFILE, SEQUENCEFILE, RCFILE, ORC, and PARQUET file formats. The three ways to specify the file format are as follows:

- CREATE TABLE… STORE AS <File_Format>
- ALTER TABLE… [PARTITION partition_spec] SET FILEFORMAT <File_Format>
- SET hive.default.fileformat=<File_Format> --default fileformat for table

Here, <File_Type> is TEXTFILE, SEQUENCEFILE, RCFILE, ORC, and PARQUET.

We can load a text file directly to a table with the TEXTFILE format. To load data to the table with other file formats, we need to load the data to a TEXTFILE format table first. Then, use INSERT OVERWRITE TABLE <target_file_format_table> SELECT * FROM <text_format_source_table> to convert and insert the data to the file format as expected.

Performance Consideration using Compression

Compression techniques in Hive can significantly reduce the amount of data transferring between mappers and reducers by proper intermediate output compression as well as output data size in HDFS by output compression. As a result, the overall Hive query will have better performance. To compress intermediate files produced by Hive between multiple MapReduce jobs, we need to set the following property (false by default) in the Hive CLI or the hive-site.xml file:

- SET hive.exec.compress.intermediate=true
- SET hive.exec.compress.output=true
- SET hive.intermediate.compression.codec=org.apache.hadoop.io.compress.SnappyCodec

Performance Consideration using Compression

Compression	Codec	Extension	Splittable
Deflate	DefaultCodec	.deflate	N
GZip	GzipCodec	.gz	N
Bzip2	BZip2Codec	.gz	Y
LZO	LzopCodec	.lzo	N
LZ4	Lz4Codec	.lz4	N
Snappy	SnappyCodec	.snappy	N

Performance Consideration with Merging Small Files

For Hive, we can do the following configurations for merging files of query results to avoid recreating small files:

- hive.merge.mapfiles: This merges small files at the end of a map-only job. By default, it is true.
- hive.merge.mapredfiles: This merges small files at the end of a MapReduce job. Set it to true since its default is false.
- hive.merge.size.per.task: This defines the size of merged files at the end of the job. The default value is 256,000,000.
- hive.merge.smallfiles.avgsize: This is the threshold for triggering file merge. The default value is 16,000,000.

When the average output file size of a job is less than the value specified by hive.merge.smallfiles.avgsize, and both hive.merge.mapfiles (for map-only jobs) and hive.merge.mapredfiles (for MapReduce jobs) are set to true, Hive will start an additional MapReduce job to merge the output files into big files.

Performance Consideration using Local mode

Hadoop can run in standalone, pseudo-distributed, and fully distributed mode. Most of the time, we need to configure Hadoop to run in fully distributed mode. When the data to process is small, it is an overhead to start distributed data processing since the launching time of the fully distributed mode takes more time than the job processing time. Since Hive 0.7.0, Hive supports automatic conversion of a job to run in local mode with the following settings:

- SET hive.exec.mode.local.auto=true; --default false
- SET hive.exec.mode.local.auto.inputbytes.max=50000000;
- SET hive.exec.mode.local.auto.input.files.max=5; --default 4

A job must satisfy the following conditions to run in the local mode:

- The total input size of the job is lower than hive.exec.mode.local.auto.inputbytes.max
- The total number of map tasks is less than hive.exec.mode.local.auto.input.files.max

Performance Consideration using JVM reuse

By default, Hadoop launches a new JVM for each map or reduce job and runs the map or reduce task in parallel. When the map or reduce job is a lightweight job running only for a few seconds, the JVM startup process could be a significant overhead. The MapReduce framework (version 1 only, not Yarn) has an option to reuse JVM by sharing the JVM to run mapper/reducer serially instead of parallel. JVM reuse applies to map or reduce tasks in the same job. Tasks from different jobs will always run in a separate JVM. To enable the reuse, we can set the maximum number of tasks for a single job for JVM reuse using the mapred.job.reuse.jvm.num.tasks property. Its default value is 1:

```
SET mapred.job.reuse.jvm.num.tasks=5;
```

We can also set the value to –1 to indicate that all the tasks for a job will run in the same JVM.

Performance Consideration using Parallel execution

Hive queries commonly are translated into a number of stages that are executed by the default sequence. These stages are not always dependent on each other. Instead, they can run in parallel to save the overall job running time. We can enable this feature with the following settings:

- SET hive.exec.parallel=true;—default false
- SET hive.exec.parallel.thread.number=16; -- default 8, it defines the max number for running in parallel

Parallel execution will increase the cluster utilization. If the utilization of a cluster is already very high, parallel execution will not help much in terms of overall performance.

Hive Transaction

Since Hive version 0.13.0, Hive fully supports row-level transactions by offering full Atomicity, Consistency, Isolation and Durability (ACID) to Hive. For now, all the transactions are autocommuted and only support data in the Optimized Row Columnar (ORC) file (available since Hive 0.11.0) format and in bucketed tables.

The following configuration parameters must be set appropriately to turn on transaction support in Hive:

```
SET hive.support.concurrency = true;
SET hive.enforce.bucketing = true;
SET hive.exec.dynamic.partition.mode = nonstrict;
SET hive.txn.manager = org.apache.hadoop.hive.ql.lockmgr.DbTxnManager;
SET hive.compactor.initiator.on = true;
SET hive.compactor.worker.threads = 1;
```

The SHOW TRANSACTIONS command is added since Hive 0.13.0 to show currently open and aborted transactions in the system:

```
SHOW TRANSACTIONS;
```

Hive Log

Logs provide useful information to find out how a Hive query/job runs. By checking the Hive logs, we can identify runtime problems and issues that may cause bad performance. There are two types of logs available in Hive: system log and job log. The system log contains the Hive running status and issues. It is configured in {HIVE_HOME}/conf/hive-log4j.properties. The following three lines for Hive log can be found:

- hive.root.logger=WARN,DRFA
- hive.log.dir=/tmp/${user.name}
- hive.log.file=hive.log

To modify the status, we can either modify the preceding lines in hive-log4j.properties (applies to all users) or set from the Hive CLI (only applies to the current user and current session) as follows:

- hive --hiveconf hive.root.logger=DEBUG,console

Hive Log

The job log contains Hive query information and is saved at the same place,/tmp/${user.name}, by default as one file for each Hive user session. We can override it in hive-site.xml with the hive.querylog.location property. If a Hive query generates MapReduce jobs, those logs can also be viewed through the Hadoop JobTracker Web UI.

Efficient Hive queries

- Data Layout (Partitions and Buckets)
- Data Sampling (Bucket and Block sampling)
- Data Processing (Bucket Map Join and Parallel execution)

What is .hiverc file?

It is a file that is executed when you launch the hive shell - making it an ideal place for adding any hive configuration/customization you want set,

on start of the hive shell. This could be:

- Setting column headers to be visible in query results
- Making the current database name part of the hive prompt
- Adding any jars or files
- Registering UDFs

.hiverc file location

The file is loaded from the hive conf directory.

If using CDH4.2 distribution and the location is: /etc/hive/conf.cloudera.hive1
If the file does not exist, you can create it. It needs to be deployed to every node from where you might launch the Hive shell.

You also can create this file under your login user home folder.

[Note: I had to create the file; The distribution did not come with it.]

Sample .hiverc

```
add jar /home/airawat/hadoop-lib/hive-contrib-0.10.0-cdh4.2.0.jar;
set hive.exec.mode.local.auto=true;
set hive.cli.print.header=true;
set hive.cli.print.current.db=true;
set hive.auto.convert.join=true;
set hive.mapjoin.smalltable.filesize=30000000;
```

What is a hook?

As you know, this is about computer programming technique, but ..

- Hooking - Techniques for intercepting function calls or messages or events in an operating system, applications, and other software components.

- Hook - Code that handles intercepted function calls, events or messages

Hive Some hooking points

- pre-execution
- post-execution
- execution-failure
- pre- and post-driver-run
- pre- and post-semantic-analyze
- metastore-initialize

How to set up hooks in Hive

hive-site.xml

<property>
<name>hive.exec.pre.hooks</name>
<value></value>
<description> Comma-separated list of pre-execution hooks to be invoked for each statement. A pre-execution hook is specified as the name of a Java class which implements the org.apache.hadoop.hive.ql.hooks.ExecuteWithHookContext interface. </description>
</property>

<property>
<name>hive.exec.post.hooks</name>
<value>org.apache.hadoop.hive.ql.hooks.ATSHook, org.apache.atlas.hive.hook.HiveHook</value>
</property>

How to set up hooks in Hive

hive-site.xml

```
<property>
        <name>hive.exec.failure.hooks</name>
        <value>org.apache.hadoop.hive.ql.hooks.ATSHook</value>
</property>
<property>
        <name>hive.aux.jars.path</name>
        <value></value>
</property>
```

Setting hook property Setting path of jars contains implementations of hook interfaces or abstract class You can use hive.added.jars.path instead of hive.aux.jars.path

Best Practices in Hive

- Use of Partitions

  Earlier database experts used to create a table per day means they used to create a complete new table for every single day. Though this practice needs lots of maintenance, people still use this in real time scenarios. But in case of hive, we have the concept of partitions which helps in maintaining a new partition for every new day without putting many efforts. As we have already seen, in case of partitions hive creates new folder and store the records accordingly which helps in targeting only required area and to avoid scanning the complete table.

Hive and Data Governance

Best Practices in Hive

- Over Partitioning

  Even though there are various benefits of creating partitions in table, like we say, anything in access in dangerous. When hive creates a partition for a table, it has to maintain the extra metadata to redirect query as per partition. So if in any case we get too many partitions a table, it would get difficult for hive to cater to this situation. So it is very important to understand the data growth and the kind of data we are going to get so that we can plan our schemas. Also it's very important to select correct columns for partitioning after completely understanding the kind of queries we want to on that data. As a partition for some queries would be beneficial at the same time it could affect the performance of other queries badly. As we know HDFS is beneficial when we have smaller set of large files instead of larger set of smaller files.

Best Practices in Hive

- Normalization

  Like any other SQL engines, we don't have any primary keys and foreign keys in Hive as hive is not meant to run complex relational queries. It's used to get data in easy and efficient manner. So while designing hive schema, we don't need to bother about selecting unique key etc. Also we don't need to bother about normalizing the set of data for efficiency.

  By keeping denormalized data, we avoid multiple disk seeks which is generally the case when we have foreign key relations. Here by not doing this, we avoid multiple I/O operation which ultimately helps in performance benefits.

Best Practices in Hive

- Efficient Use of Single Scan

  As we all know, hive does complete table scan in order to process a certain query. So it's recommend to use that single scan to perform multiple operations. Take a look at the following queries

  ```
  INSERT INTO page_views_20140101
  SELECT * FROM page_views WHERE date='20140101';
  And
  INSERT INTO page_views_20140102
  SELECT * FROM page_views
  WHERE date='20140102';
  ```

Best Practices in Hive

- Use of Bucketing

  Bucketing is a similar optimization technique as partitioning but looking at the concerns of over partitioning; we can always go for system defined data segregation. Buckets distribute the data load into user defined set of clusters by calculating hash code of key mentioned in query. Bucketing is useful when it is difficult to create partition on a column as it would be having huge variety of data in that column on which we want to run queries.

  One such example would be page_views table where we want to run queries on user_id but looking at the no. users it would get difficult to create separate partition for each and every user. So in this case we can create buckets on user_id.

Create Hive Dual

- Workaround 1:

We can use existing table to achieve dual functionality by following query

select 1, 'name', array('str1','str2'), map('key',array(1,2)) from employee limit 1;

- Workaround 2:

create table dual (dummy int);

INSERT INTO TABLE dual SELECT count(*)+1 FROM dual;

Nested ARRAY in MAP data type

Hive's default delimiters are:

- Row Delimiter => Control-A ('\001')
- Collection Item Delimiter => Control-B ('\002')
- Map Key Delimiter => Control-C ('\003')

If you override these delimiters then overridden delimiters are used during parsing. The preceding description of delimiters is correct for the usual case of flat data structures, where the complex types only contain primitive types. For nested types the level of the nesting determines the delimiter.

Nested ARRAY in MAP data type

For an array of arrays, for example, the delimiters for the outer array are Control-B ('\002') characters, as expected, but for the inner array they are Control-C ('\003') characters, the next delimiter in the list.

For nested types, for example, the depart_title column in the preceding tables, the level of nesting determines the delimiter. Using ARRAY of ARRAY as an example, the delimiters for the outer ARRAY are Ctrl + B (\002) characters, as expected, but for the inner ARRAY they are Ctrl + C (\003) characters, the next delimiter in the list. For our example of using MAP of ARRAY, the MAP key delimiter is \003, and the ARRAY delimiter is Ctrl + D or ^D (\004).

Nested ARRAY in MAP data type

Hive actually supports eight levels of delimiters, corresponding to ASCII codes 1, 2, ... 8, but you can only override the first three.

So you can write your input file as following format:

1|JOHN|abu1/abu2|key1:1'\004'2'\004'3/key2:6'\004'7'\004'8

Output of SELECT * FROM test_stg; will be:

1 JOHN ["abu1","abu2"] {"key1":[1,2,3],"key2":[6,7,8]}

Nested ARRAY in MAP data type

Quick workaround:

create table test_table as select 1, 'name', array('str1','str2'), map('key',array(1,2)) from employee limit 1;

$ hdfs dfs -copyToLocal /apps/hive/warehouse/test_table/000000_0 test_table

$ vi test_table

1^Aname^Astr1^Bstr2^Akey^C1^D2

How to access Hive from JavaScript or Apps

- With WebHCat REST API a REST API, Web-based applications can make HTTP requests to access the Hive metastore or to create and queue Hive queries and commands, Pig jobs, and MapReduce or YARN jobs (either standard or streaming).

- URL Format - HCatalog's REST resources are accessed using the following URL format: http://yourserver/templeton/v1/resource where "yourserver" is replaced with your server name, and "resource" is replaced with the HCatalog resource name. For example, to check if the server is running you could access the following URL:

  - http://127.0.0.1:50111/templeton/v1/ddl/database/default/table/employee?user.name=hive

How to access Hive from JavaScript or Apps

- Query Hive
  - Go to VM via SSH client, login as root.
  - Run the following command:

```
curl -s -d execute="select+*+from+employee;" -d
statusdir="/user/root/output"
"http://127.0.0.1:50111/templeton/v1/hive?user.name=hive"
```