**Session 7. Hive Operations**

**Assignment 3**

**Explain the below concepts with an example in brief.**

**● Hive Data Definitions**

**● Hive Data Manipulations**

**● HiveQL Manipulations**

HiveQL is the Hive query language. Like all SQL dialects in widespread use, it doesn’t fully conform to any particular revision of the ANSI SQL standard. It is perhaps closest to MySQL’s dialect, but with significant differences. Hive offers no support for row-level inserts, updates, and deletes. Hive doesn’t support transactions. Hive adds extensions to provide better performance in the context of Hadoop and to integrate with custom extensions and even external programs.

HiveQL is used for creating, altering, and dropping databases, tables, views, functions, and indexes.

The simplest syntax for creating a database is shown in the following example:

hive> **CREATE** **DATABASE** financials;

Hive will throw an error if financials already exists. You can suppress these warnings with this variation:

hive> **CREATE** **DATABASE** IF **NOT** **EXISTS** financials;

At any time, you can see the databases that already exist as follows:

hive> **SHOW** DATABASES;

**default**

financials

hive> **CREATE** **DATABASE** human\_resources;

hive> **SHOW** DATABASES;

**default**

financials

human\_resources

If you have a lot of databases, you can restrict the ones listed using a regular expression, a concept we’ll explain in [LIKE and RLIKE](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch06.html#LIKE-RLIKE), if it is new to you. The following example lists only those databases that start with the letter h and end with any other characters (the .\* part):

hive> **SHOW** DATABASES **LIKE** 'h.\*';

human\_resources

hive> ...

you can drop a database:

hive> **DROP** **DATABASE** IF **EXISTS** financials;

The IF EXISTS is optional and suppresses warnings if financials doesn’t exist.

By default, Hive won’t permit you to drop a database if it contains tables. You can either drop the tables first or append the CASCADE keyword to the command, which will cause the Hive to drop the tables in the database first:

hive> **DROP** **DATABASE** IF **EXISTS** financials **CASCADE**;

The USE command sets a database as your working database, analogous to changing working directories in a filesystem:

hive> USE financials;

You can set key-value pairs in the DBPROPERTIES associated with a database using the ALTER DATABASE command. No other metadata about the database can be changed, including its name and directory location:

hive> **ALTER** **DATABASE** financials **SET** DBPROPERTIES ('edited-by' = 'Joe Dba');

**Creating table**

**CREATE** **TABLE** IF **NOT** **EXISTS** mydb.employees (

name STRING **COMMENT** 'Employee name',

salary FLOAT **COMMENT** 'Employee salary',

subordinates ARRAY<STRING> **COMMENT** 'Names of subordinates',

deductions **MAP**<STRING, FLOAT>

**COMMENT** 'Keys are deductions names, values are percentages',

address STRUCT<street:STRING, city:STRING, **state**:STRING, zip:INT>

**COMMENT** 'Home address')

**COMMENT** 'Description of the table'

TBLPROPERTIES ('creator'='me', 'created\_at'='2012-01-02 10:00:00', ...)

**LOCATION** '/user/hive/warehouse/mydb.db/employees';

You can also copy the schema (but not the data) of an existing table:

**CREATE** **TABLE** IF **NOT** **EXISTS** mydb.employees2

**LIKE** mydb.employees;

The SHOW TABLES command lists the tables. With no additional arguments, it shows the tables in the current working database. Let’s assume we have already created a few other tables, table1 and table2, and we did so in the mydbdatabase:

hive> USE mydb;

hive> **SHOW** TABLES;

employees

table1

table2

Managed Tables

The tables we have created so far are called *managed* tables or sometimes called *internal* tables, because Hive controls the lifecycle of their data (more or less).

## External Tables

Suppose we are analyzing data from the stock markets. Periodically, we ingest the data for NASDAQ and the NYSE from a source like Infochimps (<http://infochimps.com/datasets>) and we want to study this data with many tools. (See the data sets named infochimps\_dataset\_4777\_download\_16185and infochimps\_dataset\_4778\_download\_16677, respectively, which are actually sourced from Yahoo! Finance.) The schema we’ll use next matches the schemas of both these data sources. Let’s assume the data files are in the distributed filesystem directory /data/stocks.

The following table declaration creates an external table that can read all the data files for this comma-delimited data in /data/stocks:

**CREATE** **EXTERNAL** **TABLE** IF **NOT** **EXISTS** stocks (

exchange STRING,

symbol STRING,

ymd STRING,

price\_open FLOAT,

price\_high FLOAT,

price\_low FLOAT,

price\_close FLOAT,

volume INT,

price\_adj\_close FLOAT)

**ROW** FORMAT DELIMITED FIELDS TERMINATED **BY** ','

**LOCATION** '/data/stocks';

The EXTERNAL keyword tells Hive this table is external and the LOCATION …clause is required to tell Hive where it’s located.

Because it’s external, Hive does not assume it owns the data. Therefore, dropping the table does not delete the data, although the metadata for the table will be deleted.

There are a few other small differences between managed and external tables, where some HiveQL constructs are not permitted for external tables. We’ll discuss those when we come to them.

However, it’s important to note that the differences between managed and external tables are smaller than they appear at first. Even for managed tables, you know where they are located, so you can use other tools, hadoop dfscommands, etc., to modify and even delete the files in the directories for managed tables. Hive may technically own these directories and files, but it doesn’t have full control over them! Recall, in [Schema on Read](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch03.html#SchemaOnRead), we said that Hive really has no control over the integrity of the files used for storage and whether or not their contents are consistent with the table schema. Even managed tables don’t give us this control.

# **Partitioned, Managed Tables**

The general notion of partitioning data is an old one. It can take many forms, but often it’s used for distributing load horizontally, moving data physically closer to its most frequent users, and other purposes.

Hive has the notion of partitioned tables. We’ll see that they have important performance benefits, and they can help organize data in a logical fashion, such as hierarchically.

We’ll discuss partitioned managed tables first. Let’s return to our employeestable and imagine that we work for a very large multinational corporation. Our HR people often run queries with WHERE clauses that restrict the results to a particular country or to a particular first-level subdivision (e.g., state in the United States or province in Canada). (First-level subdivision is an actual term, used here, for example: <http://www.commondatahub.com/state_source.jsp>.) We’ll just use the word state for simplicity. We have redundant state information in the address field. It is distinct from the state partition. We could remove the state element from address. There is no ambiguity in queries, since we have to use address.state to project the value inside theaddress. So, let’s partition the data first by country and then by state:

**CREATE** **TABLE** employees (

name STRING,

salary FLOAT,

subordinates ARRAY<STRING>,

deductions **MAP**<STRING, FLOAT>,

address STRUCT<street:STRING, city:STRING, **state**:STRING, zip:INT>

)

PARTITIONED **BY** (country STRING, **state** STRING);

Partitioning tables changes how Hive structures the data storage. If we create this table in the mydb database, there will still be an employees directory for the table:

hdfs://master\_server/user/hive/warehouse/mydb.db/employees

However, Hive will now create subdirectories reflecting the partitioning structure. For example:

...

.../employees/country=CA/state=AB

.../employees/country=CA/state=BC

...

.../employees/country=US/state=AL

.../employees/country=US/state=AK

...

Yes, those are the actual directory names. The state directories will contain zero or more files for the employees in those states.

Once created, the partition keys (country and state, in this case) behave like regular columns. There is one known exception, due to a bug (see [Aggregate functions](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch06.html#Aggregate-Functions)). In fact, users of the table don’t need to care if these “columns” are partitions or not, except when they want to optimize query performance.

For example, the following query selects all employees in the state of Illinois in the United States:

**SELECT** \* **FROM** employees

**WHERE** country = 'US' **AND** **state** = 'IL';

Note that because the country and state values are encoded in directory names, there is no reason to have this data in the data files themselves. In fact, the data just gets in the way in the files, since you have to account for it in the table schema, and this data wastes space.

Perhaps the most important reason to partition data is for faster queries. In the previous query, which limits the results to employees in Illinois, it is only necessary to scan the contents of one directory. Even if we have thousands of country and state directories, all but one can be ignored. For very large data sets, partitioning can dramatically improve query performance, but only if the partitioning scheme reflects common range filtering (e.g., by locations, timestamp ranges).

You can see the partitions that exist with the SHOW PARTITIONS command:

hive> **SHOW** PARTITIONS employees;

...

Country=CA/**state**=AB

country=CA/**state**=BC

...

country=US/**state**=AL

country=US/**state**=AK

...

## External Partitioned Tables

You can use partitioning with external tables. In fact, you may find that this is your most common scenario for managing large production data sets. The combination gives you a way to “share” data with other tools, while still optimizing query performance.

You also have more flexibility in the directory structure used, as you define it yourself. We’ll see a particularly useful example in a moment.

Let’s consider a new example that fits this scenario well: logfile analysis. Most organizations use a standard format for log messages, recording a timestamp, severity (e.g., ERROR, WARNING, INFO), perhaps a server name and process ID, and then an arbitrary text message. Suppose our Extract, Transform, and Load (ETL) process ingests and aggregates logfiles in our environment, converting each log message to a tab-delimited record and also decomposing the timestamp into separate year, month, and day fields, and a combined hms field for the remaining hour, minute, and second parts of the timestamp, for reasons that will become clear in a moment. You could do this parsing of log messages using the string parsing functions built into Hive or Pig, for example. Alternatively, we could use smaller integer types for some of the timestamp-related fields to conserve space. Here, we are ignoring subsequent resolution.

Here’s how we might define the corresponding Hive table:

**CREATE** **EXTERNAL** **TABLE** IF **NOT** **EXISTS** log\_messages (

hms INT,

severity STRING,

server STRING,

process\_id INT,

message STRING)

PARTITIONED **BY** (**year** INT, **month** INT, **day** INT)

**ROW** FORMAT DELIMITED FIELDS TERMINATED **BY** '\t';

ALTER TABLE statement is used to add each partition separately. It must specify a value for each partition key, the year, month, and day, in this case (see [Alter Table](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch04.html#AlterTable) for more details on this feature). Here is an example, where we add a partition for January 2nd, 2012:

**ALTER** **TABLE** log\_messages **ADD** PARTITION(**year** = 2012, **month** = 1, **day** = 2)

**LOCATION** 'hdfs://master\_server/data/log\_messages/2012/01/02';

As for managed partitioned tables, you can see an external table’s partitions with SHOW PARTITIONS:

hive> **SHOW** PARTITIONS log\_messages;

...

**year**=2011/**month**=12/**day**=31

**year**=2012/**month**=1/**day**=1

**year**=2012/**month**=1/**day**=2

...

## Customizing Table Storage Formats

In [Text File Encoding of Data Values](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch03.html#TextFileEncodingOfDataValues), we discussed that Hive defaults to a text file format, which is indicated by the optional clause STORED AS TEXTFILE, and you can overload the default values for the various delimiters when creating the table. Here we repeat the definition of the employees table we used in that discussion:

**CREATE** **TABLE** employees (

name STRING,

salary FLOAT,

subordinates ARRAY<STRING>,

deductions **MAP**<STRING, FLOAT>,

address STRUCT<street:STRING, city:STRING, **state**:STRING, zip:INT>

)

**ROW** FORMAT DELIMITED

FIELDS TERMINATED **BY** '\001'

COLLECTION ITEMS TERMINATED **BY** '\002'

**MAP** KEYS TERMINATED **BY** '\003'

LINES TERMINATED **BY** '\n'

STORED **AS** TEXTFILE;

TEXTFILE implies that all fields are encoded using alphanumeric characters, including those from international character sets, although we observed that Hive uses non-printing characters as “terminators” (delimiters), by default. When TEXTFILE is used, each line is considered a separate record.

You can replace TEXTFILE with one of the other built-in file formats supported by Hive, including SEQUENCEFILE and RCFILE, both of which optimize disk space usage and I/O bandwidth performance using binary encoding and optional compression.

The record parsing is handled by a serializer/deserializer or SerDe for short.

the SerDe Hive uses is another Java class called org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe.

For completeness, there is also an output format that Hive uses for writing the output of queries to files and to the console. For TEXTFILE, the Java class named org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat is used for output.

the SerDe Hive uses is another Java class called org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe.

For completeness, there is also an output format that Hive uses for writing the output of queries to files and to the console. For TEXTFILE, the Java class named org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat is used for output.

 DESCRIBE EXTENDED table command lists the input and output formats, the SerDe, and any SerDe properties in the DETAILED TABLE INFORMATION. For our example, we would see the following:

hive> **DESCRIBE** EXTENDED kst

...

inputFormat:com.linkedin.haivvreo.AvroContainerInputFormat,

outputFormat:com.linkedin.haivvreo.AvroContainerOutputFormat,

...

serdeInfo:SerDeInfo(name:**null**,

serializationLib:com.linkedin.haivvreo.AvroSerDe,

**parameters**:{**schema**.url=http://schema\_provider/kst.avsc})

...

# **Dropping Tables**

The familiar DROP TABLE command from SQL is supported:

**DROP** **TABLE** IF **EXISTS** employees;

The IF EXISTS keywords are optional. If not used and the table doesn’t exist, Hive returns an error.

For managed tables, the table metadata and data are deleted.

## Renaming a Table

Use this statement to rename the table log\_messages to logmsgs:

**ALTER** **TABLE** log\_messages **RENAME** **TO** logmsgs;

## Adding, Modifying, and Dropping a Table Partition

As we saw previously, ALTER TABLE table ADD PARTITION … is used to add a new partition to a table (usually an external table). Here we repeat the same command shown previously with the additional options available:

**ALTER** **TABLE** log\_messages **ADD** IF **NOT** **EXISTS**

PARTITION (**year** = 2011, **month** = 1, **day** = 1) **LOCATION** '/logs/2011/01/01'

PARTITION (**year** = 2011, **month** = 1, **day** = 2) **LOCATION** '/logs/2011/01/02'

PARTITION (**year** = 2011, **month** = 1, **day** = 3) **LOCATION** '/logs/2011/01/03'

...;

Multiple partitions can be added in the same query when using Hive v0.8.0 and later. As always, IF NOT EXISTS is optional and has the usual meaning.

## Changing Columns

You can rename a column, change its position, type, or comment:

**ALTER** **TABLE** log\_messages

CHANGE **COLUMN** hms hours\_minutes\_seconds INT

**COMMENT** 'The hours, minutes, and seconds part of the timestamp'

## AFTER severity; Adding Columns

You can add new columns to the end of the existing columns, before any partition columns.

**ALTER** **TABLE** log\_messages **ADD** COLUMNS (

app\_name STRING **COMMENT** 'Application name',

session\_id LONG **COMMENT** 'The current session id');

## Deleting or Replacing Columns

The following example removes all the existing columns and replaces them with the new columns specified:

**ALTER** **TABLE** log\_messages **REPLACE** COLUMNS (

hours\_mins\_secs INT **COMMENT** 'hour, minute, seconds from timestamp',

severity STRING **COMMENT** 'The message severity'

message STRING **COMMENT** 'The rest of the message');

This statement effectively renames the original hms column and removes the server and process\_id columns from the original schema definition. As for all ALTER statements, only the table metadata is changed.