CREATING AND POPULATING HIVE TABLES

Creating and Populating Hive Tables

- **Introduction**
- Managed and External Tables
- Partitioned Tables
- Bucketed Sorted Tables
- Skewed Tables
- Storage and Row Formats
- CREATE TABLE as SELECT
- Populating Tables and Partitions

Hive Tables

- Recall that creating a Hive table is just putting a structure over HDFS data
- For Hive to be able to know where data is (or will be) stored, Hive tables have a **LOCATION** property
- For Hive to be able to construct appropriate Input/Output Formats while running MapReduce jobs, Hive tables have a **STORED AS** property
- For Hive to extract columns from a record, Hive tables have a **ROW FORMAT** property

Creating a Hive Table

- A Hive table is created by a **CREATE TABLE** statement
- The **LOCATION** for the table's data, if not specified, is by default set in the **`hive.metastore.warehouse.dir`** configuration property, which is by default the HDFS path /user/hive/warehouse

Creating and Populating Hive Tables

- Introduction
- **Managed and External Tables**
- Partitioned Tables
- Bucketed Sorted Tables
- Skewed Tables
- Storage and Row Formats
- CREATE TABLE as SELECT
- Populating Tables and Partitions

Managed/External Tables

- A Hive table is external if it is created by the **CREATE EXTERNAL TABLE** statement
- If a table is not external, *when it is dropped*, *the directory* denoted in its LOCATION property *is deleted*
- This is the only difference between an external and a managed table

Creating and Populating Hive Tables

- Introduction
- Managed and External Tables
- **Partitioned Tables**
- Bucketed Sorted Tables
- Skewed Tables
- Storage and Row Formats
- CREATE TABLE as SELECT
- Populating Tables and Partitions

Partitioned Tables

- In Hive, a table can be partitioned by the different values of a given column
- We call such tables partitioned
- Data belonging to a partition should be stored in a separate directory, with the name `<LOCATION>/c1=v1`
- Partition column is a pseudocolumn, and its data should not be included in the data files, since this value is unique among all rows in the directory denoting this partitions
- Partitioning can be nested

Partitioned Tables

- The column(s) by which the table is partitioned are stated at creation time

```
$ hive

hive> CREATE TABLE users (
        id INT,
        name STRING,
        email STRING)
        PARTITIONED BY(city)
        ROW FORMAT DELIMITED
        FIELDS TERMINATED BY ‘,’;

hive> OK
```

Partitioned Tables

- Notice that we did not define an extra column for city

```
$ hive

hive> CREATE TABLE users (
        id INT,
        name STRING,
        email STRING)
        PARTITIONED BY(city)
        ROW FORMAT DELIMITED
        FIELDS TERMINATED BY ‘,’;

hive> OK
```

Partitioned Tables

- The directory structure would look like this:

```
/user
  /hive
    /warehouse
     /users
       /city=34
         <file(s)>
       /city=06
         <file(s)>
       /city=01
         <file(s)>
       ...
```

Partitioned Tables

- Users can still query on the partitioned column (in fact, it is efficient since Hive can skip the non-relevant partitions)

```
$ hive

hive> SELECT * FROM users
      WHERE city = '34'
```

Creating and Populating Hive Tables

- Introduction
- Managed and External Tables
- Partitioned Tables
- **Bucketed Sorted Tables**
- Skewed Tables
- Storage and Row Formats
- CREATE TABLE as SELECT
- Populating Tables and Partitions

Bucketed Tables

- Instead of partitioning, a table can be bucketed into a predefined number of buckets based on a column
- All rows with the same value for that column goes to the same bucket (like partitioning), but a bucket may contain more than one unique values for that column (unlike partitioning)
- Data can additionally be sorted within each bucket
  - Hive takes advantage of such structure to optimize certain properties (such as performing a Map-side join)

Bucketed Tables

- The column to be bucketed, and number of buckets are stated at creation time

```
$ hive

hive> CREATE TABLE users (
        id INT,
        name STRING,
        email STRING)

        PARTITIONED BY(city)
        CLUSTERED BY(id) SORTED BY(name) INTO 32 BUCKETS

        ROW FORMAT DELIMITED
        FIELDS TERMINATED BY ‘,’;

hive> OK
```

Creating and Populating Hive Tables

- Introduction
- Managed and External Tables
- Partitioned Tables
- Bucketed Sorted Tables
- **Skewed Tables**
- Storage and Row Formats
- CREATE TABLE as SELECT
- Populating Tables and Partitions

Skewed Tables

- A skewed table can be created when a column has a high skew, by stating the values that appear very often
- This is similar to partitioning, but table is partitioned based on the specified values
- When inserting data into such tables, Hive would create separate files (or directories) for each skewed value and one for the rest
- The skew information is used by Hive to optimize queries (it can skip the whole file, for example)

Skewed Tables

- Skew is specified at creation time

```
$ hive

hive> CREATE TABLE users (
        id INT,
        name STRING,
        email STRING,
        city INT)

        SKEWED BY(city) ON (34) [STORED AS DIRECTORIES]

        ROW FORMAT DELIMITED
        FIELDS TERMINATED BY ‘,’;

hive> OK
```

Creating and Populating Hive Tables

- Introduction
- Managed and External Tables
- Partitioned Tables
- Bucketed Sorted Tables
- Skewed Tables
- **Storage and Row Formats**
- CREATE TABLE as SELECT
- Populating Tables and Partitions

Storage and Row Formats

- When creating a table, we need to tell Hive
  - **The storage format** of the underlying data (*Data is in text files, each line representing a database row*, for example)
  - **The row format** from which the columns can be extracted and written (*In each line, columns are seperated with commas*, for example)

Storage and Row Formats

- At creation time:
  - A table's strorage format is specified with the STORED AS clause
  - A table's row format is specified with the ROW FORMAT clause

Storage Formats

- The default storage format is defined in the configuration parameter 'hive.default.fileformat', and its default value is TEXTFILE
- Some example storage formats:
  - SEQUENCEFILE
  - AVRO
  - PARQUET
  - ORC
  - ...

Storage Formats

- StorageFormat can also be specified by directly passing the InputFormat and OutputFormat to the STORED AS clause
- The following clause is equivalent to STORED AS TEXTFILE, for example:

```
STORED AS

    INPUTFORMAT
    'org.apache.hadoop.mapred.TextInputFormat'

    OUTPUTFORMAT

    'org.apache.hadoop.hive.ql.io.IgnoreKeyTextOutputF
    ormat'
```

Storage Formats

- There is also support for non-native underlying storage, such as HBase, and such storage backends are specified by StorageHandlers
- A StorageHandler is specified at creation time with 'STORED BY' clause,
  - e.g. `STORED BY 'org.apache.hadoop.hive.io.HBaseStorageHandler'`

Row Formats

- To create a row from a <key, value> record coming from the InputFormat, and to write a row object as a <key, value> record into the underlying storage, Hive uses a mechanism called SerDe
- SerDe stands for 'Serializer and Deserializer'
- Tables can be created by either a native SerDe or a custom SerDe
- A native SerDe is used when the ROW FORMAT clause is not specified, or ROW FORMAT DELIMITED is used
  - DELIMITED is used to read delimited files
  - Other optional properties such as ESCAPED BY, NULL DEFINED AS can also be specified

Row Formats

- Sometimes it is not necessary to specify both StorageFormat and RowFormat, since one implies the other (example: Avro storage)
- Built-in SerDes are:
  - Avro
  - ORC
  - RegEx
  - Thrift
  - Parquet
  - CSV

Row Formats

- There are also 3rd party SerDes avaliable, such as JSON SerDe provided by Amazon
- It is also possible to implement a custom SerDe
- RowFormat with a custom SerDe is specified with:

```
ROW FORMAT SERDE
    'org.apache.hadoop.hive.serde2.RegexSerDe'
```

- Additional properties are specified by WITH SERDEPROPERTIES clause

Creating and Populating Hive Tables

- Introduction
- Managed and External Tables
- Partitioned Tables
- Bucketed Sorted Tables
- Skewed Tables
- Storage and Row Formats
- **CREATE TABLE as SELECT**
- Populating Tables and Partitions

Create Table as Select

- To create a Hive table with results of a select statement can be performed in one step
- This comes handy when creating a Hive table with some custom Storage Format when the original data is not stored in this format
- CTaS is allowed when
  - The target table is not a partitioned table
  - The target table is a Hive-managed table
  - The target table is not a list-bucketing (a skewed table stored as directories) table
- Any select statement that Hive supports is possible

Create Table as Select

```
$ hive

hive> CREATE TABLE t1
        ROW FORMAT SERDE
        "org.apache.hadoop.hive.serde2.columnar.Columnar
        SerDe"
        STORED AS RCFile
        AS
        SELECT ...

hive> OK
```

Creating and Populating Hive Tables

- Introduction
- Managed and External Tables
- Partitioned Tables
- Bucketed Sorted Tables
- Skewed Tables
- Storage and Row Formats
- CREATE TABLE as SELECT
- **Populating Tables and Partitions**

Populating Tables and Partitions

- A Hive table can be populated
  - By directly copying/moving data into the directory specified in the table's LOCATION property
  - By moving (possibly local) data with the **`LOAD DATA [LOCAL] INPATH <path> [OVERWRITE] INTO TABLE <table_name> PARTITION`** (partcol=val1) statement
  - With an **`INSERT INTO/OWERWRITE TABLE <table_name> [PARTITION] (partcol1=val1)`** statement to insert data from queries
    - This is also handy for populating tables with a different format than the underlying data

Populating Tables and Partitions

- Note that Hive is schema-on-read
- That is, Hive would not complain if you LOAD or manually move/ copy wrong-formatted data into a table's LOCATION
- Such errors can only be captured at query-time

Dynamic Partition Inserts

- When the table is partitioned, one must always insert into the appropriate partition
- That is, in the INSERT INTO/OVERWRITE statement, both the partition column and the partition value should be specified
- If the **`hive.exec.dynamic.partition`** configuration parameter is set **`true`**, the value for the partition column is not need to be specified
  - Hive will automatically load data into correct partitions in this case

Dynamic Partition Inserts

```
// Dynamic partition insert example

  FROM t1
  INSERT OVERWRITE TABLE t2
  PARTITION(country)
    SELECT t1.c1, t1.c2, t1.cnt
```

```
// Static partition insert example

  FROM t1
  INSERT OVERWRITE TABLE t2
  PARTITION(country=“TR”)
    SELECT t1.c1, t1.c2, t1.cnt
    WHERE t1.cnt = “TR”
```

Populating Bucketed Tables

- Recall that Hive statements (including INSERT) are converted into MapReduce jobs
- If the target table is bucketed, the number of reducers created by the INSERT statement should be equal to the number of buckets of the target table

Populating Bucketed Tables

- In Hive, number of reducers created for a MapReduce job can be controlled by setting `mapreduce.reduce.tasks` configuration property and specifying CLUSTER BY clause for controlling partitioning behavior

```
// If t2 is bucketed by c1 (into 10 buckets)

  set mapreduce.reduce.tasks=10;
  FROM t1
  INSERT OVERWRITE TABLE t2
    SELECT t1.c1, t1.c2
  CLUSTER BY t1.c1;
```

Populating Bucketed Tables

- Alternatively, the **`hive.enforce.bucketing`** configuration property can be set **`true`**, and Hive would set the correct number of reducers and the cluster by column automatically

```
// If t2 is bucketed by c1 (into 10 buckets)

  set hive.enforce.bucketing=true;
  FROM t1
  INSERT OVERWRITE TABLE t2
    SELECT t1.c1, t1.c2
```

Populating Skewed Tables

- If a skewed table is list bucketed (STORED AS DIRECTORIES), to load data into it:
  - **`hive.mapred.supports.subdirectories`**
  - **`hive.optimize.listbucketing`**

  must be set **`true`**

```
// If t2 is a skewed table stored as directories

  set hive.mapred.supports.subdirectories=true;
  set hive.optimize.listbucketing=true;

  FROM t1
  INSERT OVERWRITE TABLE t2
    SELECT t1.c1, t1.c2
```

Multi-Insert Statements

- Writing multi-insert statements at once is possible with Hive
- This is simply done by writing multiple insert statements without using a semicolon separating them
- The FROM can be common among all inserts
- Hive can avoid doing multiple passes over the same data in multi-insert

```
FROM t1
INSERT OVERWRITE TABLE t2
  SELECT t1.c1, t1.c2
INSERT OVERWRITE TABLE t3
  SELECT t1.c2
INSERT OVERWRITE TABLE t4
  SELECT t1.c2, t1.c1;
```

Exporting Data into FileSystem from Queries

- The result of a query can also be written into (possibly local) filesystem
- This is done by **`INSERT OVERWRITE [LOCAL] DIRECTORY <path> [ROW FORMAT`** ***`row_format`*****`] [STORED AS`** ***`storage_format`*****`] SELECT ... FROM ...`**
- Writing multi-insert statements, possibly as a combination of written into directories and inserting into tables, is possible

```
INSERT OVERWRITE DIRECTORY '/user/hdfs/data/t1_data'
SELECT c1, c2 FROM t1
```

Demo

Creating and Populating Partitioned, Skewed, and Bucketed Tables

Demo

Working with different Storage Formats

Creating and Populating Hive Tables

End of Chapter