Chapter 12

Hive

Learning Objectives

- Defining and Running Hive
- Hive vs. Traditional Databases
- HiveQL
- Tables
- Querying Data
- User-Defined Functions

What Is Hive

- Hive is a framework for data warehousing on top of Hadoop.
- Hive grew from a need to manage and learn from the huge volumes of data that Facebook was producing every day.
- Hive allows analysts with strong SQL skills to run queries on the huge volumes of data.
- Today, Hive is a successful Apache project used by many organizations as a general-purpose, scalable data processing platform.

The Hive Shell

- The shell is the primary way that we will interact with Hive, by issuing commands in HiveQL.
- HiveQL is Hive's query language, a dialect of SQL.
- Like SQL, HiveQL is generally case-insensitive (except for string comparisons).
- When starting Hive for the first time, we can check that it is working by listing its tables:

```
hive> SHOW TABLES;
OK
Time taken: 10.425 seconds
```

Hive –f and –e Options

You can also run the Hive shell in noninteractive mode. The `-f` option runs the commands in the specified file, which is *script.q* in this example:

```
% hive -f script.q
```

For short scripts, you can use the `-e` option to specify the commands inline, in which case the final semicolon is not required:

```
% hive -e 'SELECT * FROM dummy'
Hive history file=/tmp/tom/hive_job_log_tom_201005042112_1906486281.txt
OK
X
Time taken: 4.734 seconds
```

A Hive Example – 1

Just like an RDBMS, Hive organizes its data into tables. We create a table to hold the weather data using the `CREATE TABLE` statement:

```
CREATE TABLE records (year STRING, temperature INT, quality INT)
ROW FORMAT DELIMITED
  FIELDS TERMINATED BY '\t';
```

Next, we can populate Hive with the data. This is just a small sample, for exploratory purposes:

```
LOAD DATA LOCAL INPATH 'input/ncdc/micro-tab/sample.txt'
OVERWRITE INTO TABLE records;
```

A Hive Example – 2

Thus, the files for the `records` table are found in the */user/hive/warehouse/records* directory on the local filesystem:

```
% ls /user/hive/warehouse/records/
sample.txt
```

Now that the data is in Hive, we can run a query against it:

```
hive> SELECT year, MAX(temperature)
    > FROM records
    > WHERE temperature != 9999
    >   AND (quality = 0 OR quality = 1 OR quality = 4 OR quality = 5 OR quality = 9)
    > GROUP BY year;
1949    111
1950    22
```

Configuring Hive – 1

You can override the configuration directory that Hive looks for in *hive-site.xml* by passing the `--config` option to the `hive` command:

```
% hive --config /Users/tom/dev/hive-conf
```

Hive also permits you to set properties on a per-session basis, by passing the `-hiveconf` option to the `hive` command. For example, the following command sets the cluster (in this case, to a pseudodistributed cluster) for the duration of the session:

```
% hive -hiveconf fs.default.name=localhost -hiveconf mapred.job.tracker=localhost:8021
```

Configuring Hive – 2

You can change settings from within a session, too, using the SET command. This is useful for changing Hive or MapReduce job settings for a particular query. For example, the following command ensures buckets are populated according to the table definition (see "Buckets" on page 433):

```
hive> SET hive.enforce.bucketing=true;
```

To see the current value of any property, use SET with just the property name:

```
hive> SET hive.enforce.bucketing;
hive.enforce.bucketing=true
```

Schema on Read vs. Schema on Write

- Schema on read – doesn't verify the data when it is loaded, but rather when a query is issued.
  - Makes for a very fast initial load, since the data does not have to be read, parsed, and serialized to disk in the database's internal format.
- Schema on write – data is checked against the schema when it is written into the database.
  - Makes query time performance faster because the database can index columns and perform compression on the data.
  - Takes longer to load data into the database.

Hive Data Types – 1

Table 12-3. Hive data types

Category	Type	Description	Literal examples
Primitive	TINYINT	1-byte (8-bit) signed integer, from -128 to 127	1
	SMALLINT	2-byte (16-bit) signed integer, from -32,768 to 32,767	1
	INT	4-byte (32-bit) signed integer, from -2,147,483,648 to 2,147,483,647	1
	BIGINT	8-byte (64-bit) signed integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	1
	FLOAT	4-byte (32-bit) single-precision floating-point number	1.0
	DOUBLE	8-byte (64-bit) double-precision floating-point number	1.0

Hive Data Types – 2

Category	Type	Description	Literal examples
	BOOLEAN	true/false value	TRUE
	STRING	Character string	'a', "a"
	BINARY	Byte array	Not supported
	TIMESTAMP	Timestamp with nanosecond precision	1325502245000, '2012-01-02 03:04:05.123456789'
Complex	ARRAY	An ordered collection of fields. The fields must all be of the same type.	array(1, 2)[a]
	MAP	An unordered collection of key-value pairs. Keys must be primitives; values may be any type. For a particular map, the keys must be the same type, and the values must be the same type.	map('a', 1, 'b', 2)
	STRUCT	A collection of named fields. The fields may be of different types.	struct('a', 1, 1.0)[b]

[a] The literal forms for arrays, maps, and structs are provided as functions. That is, `array()`, `map()`, and `struct()` are built-in Hive functions.

[b] The columns are named `col1`, `col2`, `col3`, etc.

Functions

- Hive comes with a large number of built-in functions.
  - Mathematical and statistical functions
  - String functions
  - Date functions
  - Conditional functions
  - Aggregate functions
  - Functions for working with XML.

You can retrieve a list of functions from the Hive shell by typing `SHOW FUNCTIONS`.[5] To get brief usage instructions for a particular function, use the `DESCRIBE` command:

```
hive> DESCRIBE FUNCTION length;
length(str | binary) - Returns the length of str or number of bytes in binary data
```

Managed Table vs. External Table

- Managed table is in Hive's warehouse directory, whereas external table is at an existing location outside the warehouse directory.

```
CREATE TABLE managed_table (dummy STRING);
LOAD DATA INPATH '/user/tom/data.txt' INTO table managed_table;
```

will *move* the file *hdfs://user/tom/data.txt* into Hive's warehouse directory for the `managed_table` table, which is *hdfs://user/hive/warehouse/managed_table*.[6]

An external table behaves differently. You control the creation and deletion of the data. The location of the external data is specified at table creation time:

```
CREATE EXTERNAL TABLE external_table (dummy STRING)
  LOCATION '/user/tom/external_table';
LOAD DATA INPATH '/user/tom/data.txt' INTO TABLE external_table;
```

Partitions – 1

- Hive organizes tables into partitions, a way of dividing a table into parts based on the value of a partition column, such as a date.
- A table may be partitioned in multiple dimensions.
- Tables or partitions may be subdivided further into buckets to give extra structure to the data for more efficient queries.
- Partitions are defined at table creation time using the PARTITIONED BY clause.

```
CREATE TABLE logs (ts BIGINT, line STRING)
PARTITIONED BY (dt STRING, country STRING);
```

When we load data into a partitioned table, the partition values are specified explicitly:

```
LOAD DATA LOCAL INPATH 'input/hive/partitions/file1'
INTO TABLE logs
PARTITION (dt='2001-01-01', country='GB');
```

Partitions – 2

After loading a few more files into the logs table, the directory structure might look like this:

```
/user/hive/warehouse/logs
├── dt=2001-01-01/
│   ├── country=GB/
│   │   ├── file1
│   │   └── file2
│   └── country=US/
│       └── file3
└── dt=2001-01-02/
    ├── country=GB/
    │   └── file4
    └── country=US/
        ├── file5
        └── file6
```

Partitions – 3

We can ask Hive for the partitions in a table using SHOW PARTITIONS:

```
hive> SHOW PARTITIONS logs;
dt=2001-01-01/country=GB
dt=2001-01-01/country=US
dt=2001-01-02/country=GB
dt=2001-01-02/country=US
```

You can use partition columns in SELECT statements in the usual way. Hive performs *input pruning* to scan only the relevant partitions. For example:

```
SELECT ts, dt, line
FROM logs
WHERE country='GB';
```

Bucket – 1

First, let's see how to tell Hive that a table should be bucketed. We use the CLUSTERED BY clause to specify the columns to bucket on and the number of buckets:

```
CREATE TABLE bucketed_users (id INT, name STRING)
CLUSTERED BY (id) INTO 4 BUCKETS;
```

efficient merge-sort. The syntax for declaring that a table has sorted buckets is:

```
CREATE TABLE bucketed_users (id INT, name STRING)
CLUSTERED BY (id) SORTED BY (id ASC) INTO 4 BUCKETS;
```

Bucket – 2

Take an unbucketed users table:

```
hive> SELECT * FROM users;
0    Nat
2    Joe
3    Kay
4    Ann
```

To populate the bucketed table, we need to set the `hive.enforce.bucketing` property to `true` so that Hive knows to create the number of buckets declared in the table definition. Then it is a matter of just using the `INSERT` command:

```
INSERT OVERWRITE TABLE bucketed_users
SELECT * FROM users;
```

Bucket – 3

Physically, each bucket is just a file in the table (or partition) directory. The filename is not important, but bucket *n* is the *n*th file when arranged in lexicographic order. In fact, buckets correspond to MapReduce output file partitions: a job will produce as many buckets (output files) as reduce tasks. We can see this by looking at the layout of the `bucketed_users` table we just created. Running this command:

```
hive> dfs -ls /user/hive/warehouse/bucketed_users;
```

shows that four files were created, with the following names (the name is generated by Hive):

```
000000_0
000001_0
000002_0
000003_0
```

Bucket – 4

The first bucket contains the users with IDs 0 and 4, since for an `INT` the hash is the integer itself, and the value is reduced modulo the number of buckets—4 in this case:[9]

```
hive> dfs -cat /user/hive/warehouse/bucketed_users/000000_0;
0Nat
4Ann
```

We can see the same thing by sampling the table using the `TABLESAMPLE` clause, which restricts the query to a fraction of the buckets in the table rather than the whole table:

```
hive> SELECT * FROM bucketed_users
    > TABLESAMPLE(BUCKET 1 OUT OF 4 ON id);
0    Nat
4    Ann
```

rows would be returned. It's possible to sample a number of buckets by specifying a different proportion (which need not be an exact multiple of the number of buckets, as sampling is not intended to be a precise operation). For example, this query returns half of the buckets:

```
hive> SELECT * FROM bucketed_users
    > TABLESAMPLE(BUCKET 1 OUT OF 2 ON id);
0    Nat
4    Ann
2    Joe
```

Sampling a bucketed table is very efficient because the query only has to read the buckets that match the TABLESAMPLE clause. Contrast this with sampling a nonbucketed table using the rand() function, where the whole input dataset is scanned, even if only a very small sample is needed:

```
hive> SELECT * FROM users
    > TABLESAMPLE(BUCKET 1 OUT OF 4 ON rand());
2    Joe
```

Storage Formats – 1

- There are two dimensions that govern table storage in Hive: the row format and the file format.
- The row format dictates how rows, and the fields in a particular row, are stored.
- In Hive parlance, the row format is defined by a SerDe, a portmanteau word for a Serializer-Deserializer.
- When you create a table with no ROW FORMAT or STORED AS clauses, the default format is delimited text with one row per line.

Storage Formats – 2

- When acting as a **deserializer**, which is the case when querying a table, a SerDe will deserialize a row of data from the bytes in the file to objects used internally by Hive to operate on that row of data.
- When used as a **serializer**, which is the case when performing an INSERT or CTAS (see "Importing Data" on page 441), the table's SerDe will serialize Hive's internal representation of a row of data into the bytes that are written to the table.

Sequence Files

- Hadoop's sequence file format (see "SequenceFile" on page 130) is a general-purpose binary format for sequences of records (key-value pairs).
- Sequence files in Hive can be declared using STORED AS SEQUENCEFILE in the CREATE TABLE statement.
- Sequence files support for splittable compression.
- Sequence files are row-oriented, that is, the fields in each row are stored together as the contents of a single sequence-file record.

Avro Files and RCFile

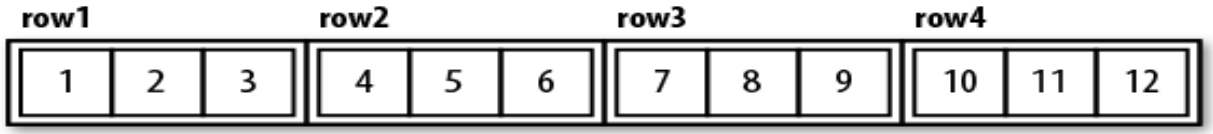
- **Avro** datafiles are like sequence files (splittable, compressible, row-oriented), except they have support for schema evolution and bindings in multiple languages (see “Avro” on page 110).
- Hive provides another binary storage format called ***RCFile***, short for *Record Columnar File*.
- RCFiles are similar to sequence files, except that they store data in a column-oriented fashion.
- RCFile breaks up the table into row splits, then within each split stores the values for each row in the first column, followed by the values for each row in the second column, and so on.

Row-Oriented vs. Column-Oriented Storage

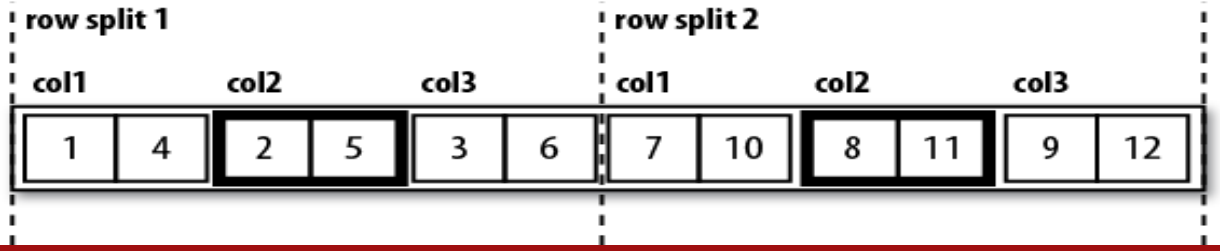
Logical table

	col1	col2	col3
row1	1	2	3
row2	4	5	6
row3	7	8	9
row4	10	11	12

Row-oriented layout (SequenceFile)



Column-oriented layout (RCFile)



RegexSerDe – 1

Let's see how to use another SerDe for storage. We'll use a contrib SerDe that uses a regular expression for reading the fixed-width station metadata from a text file:

```
CREATE TABLE stations (usaf STRING, wban STRING, name STRING)
ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
  "input.regex" = "(\\d{6}) (\\d{5}) (.{29}) .*"
);
```

SerDes can be configured with extra properties using the `WITH SERDEPROPERTIES` clause. Here we set the `input.regex` property, which is specific to `RegexSerDe`.

RegexSerDe – 2

To populate the table, we use a `LOAD DATA` statement as before:

```
LOAD DATA LOCAL INPATH "input/ncdc/metadata/stations-fixed-width.txt"
INTO TABLE stations;
```

When we retrieve data from the table, the SerDe is invoked for deserialization, as we can see from this simple query, which correctly parses the fields for each row:

```
hive> SELECT * FROM stations LIMIT 4;
010000    99999    BOGUS NORWAY
010003    99999    BOGUS NORWAY
010010    99999    JAN MAYEN
010013    99999    ROST
```

Insert – 1

Here's an example of an INSERT statement:

```
INSERT OVERWRITE TABLE target
SELECT col1, col2
  FROM source;
```

For partitioned tables, you can specify the partition to insert into by supplying a PARTITION clause:

```
INSERT OVERWRITE TABLE target
PARTITION (dt='2001-01-01')
SELECT col1, col2
  FROM source;
```

Insert – 2

You can specify the partition dynamically by determining the partition value from the SELECT statement:

```
INSERT OVERWRITE TABLE target
PARTITION (dt)
SELECT col1, col2, dt
  FROM source;
```

This is known as a *dynamic-partition insert*. This feature is off by default, so you need to enable it by setting `hive.exec.dynamic.partition` to `true` first.

Insert Statement Turn-Around

In HiveQL, you can turn the INSERT statement around and start with the FROM clause for the same effect:

```
FROM source
INSERT OVERWRITE TABLE target
  SELECT col1, col2;
```

The reason for this syntax becomes clear when you see that it's possible to have multiple INSERT clauses in the same query. This so-called *multitable insert* is more efficient than multiple INSERT statements because the source table needs to be scanned only once to produce the multiple, disjoint outputs.

Multitable Insert

Here's an example that computes various statistics over the weather dataset:

```
FROM records2
INSERT OVERWRITE TABLE stations_by_year
  SELECT year, COUNT(DISTINCT station)
  GROUP BY year
INSERT OVERWRITE TABLE records_by_year
  SELECT year, COUNT(1)
  GROUP BY year
INSERT OVERWRITE TABLE good_records_by_year
  SELECT year, COUNT(1)
  WHERE temperature != 9999
    AND (quality = 0 OR quality = 1 OR quality = 4 OR quality = 5 OR quality = 9)
  GROUP BY year;
```

SELECT clause. In the following query, the target table has two columns named col1 and col2 whose types are the same as the ones in the source table:

```
CREATE TABLE target
AS
SELECT col1, col2
FROM source;
```

You can rename a table using the ALTER TABLE statement:

```
ALTER TABLE source RENAME TO target;
```

For example, consider adding a new column:

```
ALTER TABLE target ADD COLUMNS (col3 STRING);
```

Drop and Select

If you want to delete all the data in a table but keep the table definition (like `DELETE` or `TRUNCATE` in MySQL), you can simply delete the datafiles. For example:

```
hive>
dfs -rmr /user/hive/warehouse/my_table;
```

In some cases, you want to control which reducer a particular row goes to, typically so you can perform some subsequent aggregation. This is what Hive's `DISTRIBUTE BY` clause does. Here's an example to sort the weather dataset by year and temperature, in such a way to ensure that all the rows for a given year end up in the same reducer partition:[11]

```
hive> FROM records2
    > SELECT year, temperature
    > DISTRIBUTE BY year
    > SORT BY year ASC, temperature DESC;
1949    111
1949    78
1950    22
1950    0
1950    -11
```

Inner Join

```
hive> SELECT * FROM sales;
Joe     2
Hank    4
Ali     0
Eve     3
Hank    2
```

```
hive> SELECT * FROM things;
2       Tie
4       Coat
3       Hat
1       Scarf
```

We can perform an inner join on the two tables as follows:

```
hive> SELECT sales.*, things.*
    > FROM sales JOIN things ON (sales.id = things.id);
Joe     2       2       Tie
Hank    2       2       Tie
Eve     3       3       Hat
Hank    4       4       Coat
```

Outer Join – 1

```
hive> SELECT sales.*, things.*
    > FROM sales LEFT OUTER JOIN things ON (sales.id = things.id);
Ali     0    NULL NULL
Joe     2    2    Tie
Hank    2    2    Tie
Eve     3    3    Hat
Hank    4    4    Coat

hive> SELECT sales.*, things.*
    > FROM sales RIGHT OUTER JOIN things ON (sales.id = things.id);
NULL    NULL 1    Scarf
Joe     2    2    Tie
Hank    2    2    Tie
Eve     3    3    Hat
Hank    4    4    Coat
```

Outer Join – 2

Finally, there is a full outer join, where the output has a row for each row from both tables in the join:

```
hive> SELECT sales.*, things.*
    > FROM sales FULL OUTER JOIN things ON (sales.id = things.id);
Ali     0    NULL NULL
NULL    NULL 1    Scarf
Joe     2    2    Tie
Hank    2    2    Tie
Eve     3    3    Hat
Hank    4    4    Coat
```

Subqueries

The following query finds the mean maximum temperature for every year and weather station:

```
SELECT station, year, AVG(max_temperature)
FROM (
  SELECT station, year, MAX(temperature) AS max_temperature
  FROM records2
  WHERE temperature != 9999
    AND (quality = 0 OR quality = 1 OR quality = 4 OR quality = 5 OR quality = 9)
  GROUP BY station, year
) mt
GROUP BY station, year;
```

Views – 1

We can use views to rework the query from the previous section for finding the mean maximum temperature for every year and weather station. First, let's create a view for valid records, that is, records that have a particular `quality` value:

```
CREATE VIEW valid_records
AS
SELECT *
FROM records2
WHERE temperature != 9999
  AND (quality = 0 OR quality = 1 OR quality = 4 OR quality = 5 OR quality = 9);
```

Next, let's create a second view of maximum temperatures for each station and year. It is based on the `valid_records` view:

```
CREATE VIEW max_temperatures (station, year, max_temperature)
AS
SELECT station, year, MAX(temperature)
FROM valid_records
GROUP BY station, year;
```

Views – 2

With the views in place, we can now use them by running a query:

```
SELECT station, year, AVG(max_temperature)
FROM max_temperatures
GROUP BY station, year;
```

The result of the query is the same as running the one that uses a subquery, and in particular, Hive creates the same number of MapReduce jobs for both: two in each case, one for each `GROUP BY`. This example shows that Hive can combine a query on a view into a sequence of jobs that is equivalent to writing the query without using a view. In other words, Hive won't needlessly materialize a view, even at execution time.

at an example. Consider a table with a single column, x, which contains arrays of strings. It's instructive to take a slight detour to see how the table is defined and populated:

```
CREATE TABLE arrays (x ARRAY<STRING>)
ROW FORMAT DELIMITED
  FIELDS TERMINATED BY '\001'
  COLLECTION ITEMS TERMINATED BY '\002';
```

Notice that the `ROW FORMAT` clause specifies that the entries in the array are delimited by Control-B characters. The example file that we are going to load has the following contents, where `^B` is a representation of the Control-B character to make it suitable for printing:

```
a^Bb
c^Bd^Be
```

User-Defined Functions – 2

After running a `LOAD DATA` command, the following query confirms that the data was loaded correctly:

```
hive> SELECT * FROM arrays;
["a","b"]
["c","d","e"]
```

Next, we can use the `explode` UDTF to transform this table. This function emits a row for each entry in the array, so in this case the type of the output column `y` is `STRING`. The result is that the table is flattened into five rows:

```
hive> SELECT explode(x) AS y FROM arrays;
a
b
c
d
e
```

User-Defined Functions – 3

Example 12-3. A UDF for stripping characters from the ends of strings

```
package com.hadoopbook.hive;

import org.apache.commons.lang.StringUtils;
import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;

public class Strip extends UDF {
  private Text result = new Text();

  public Text evaluate(Text str) {
    if (str == null) {
      return null;
    }
    result.set(StringUtils.strip(str.toString()));
    return result;
  }
  public Text evaluate(Text str, String stripChars) {
    if (str == null) {
      return null;
    }
    result.set(StringUtils.strip(str.toString(), stripChars));
    return result;
  }
}
```

User-Defined Functions – 4

- Two User-Defined Aggregate Functions: P. 454 and P. 457