Interacting with the Hadoop Ecosystem

Big Data Analysis

Pig and Pig Latin

A common complaint with MapReduce is that it's often "difficult to program." Rather than thinking in terms of data flow operations, you need to think in terms of mapper and reducer functions along with job chaining. Operations such as joining and filtering, which are relatively easy to implement with SQL in higher-level languages, become complicated using MapReduce because the programmer needs to write code to carry them out.

Fortunately, programming is made easier by a Hadoop extension called *Pig*. This extension provides a high-level data processing language called *Pig Latin*, an easy-to-understand data flow language that offers common operations such as *join*, *group*, *filter*, *sort*, etc. This significantly simplifies the amount of coding that is required and effectively democratizes MapReduce for the benefit of non-Java developers.

A Pig Latin program is a sequence of steps where each step is a single data transformation. When a Pig Latin program is submitted to a Hadoop cluster, the Pig engine transforms/converts the Pig Latin script into a MapReduce program.

Let's look at an example. The code below is what you would write for a MapReduce program:

```
A     =     LOAD 'myfile'
            AS (x, y, z);
B     =     FILTER A by x > 0;
D     =     FOREACH A GENERATE
            x, COUNT(B);
STORE D INTO 'output';
```

The Pig engine does the following with the MapReduce code:

- parses
- checks
- optimizes
- plans execution
- submits jar to hadoop

© Copyright UC Regents

- monitors job progress

Using Pig Latin, the MapReduce statement `B = FILTER A by x > 0;` becomes "Map: Filter" and the statement `x, COUNT(B);` becomes "Reduce: Count."

In other words, filtering occurs in the map function and the counting happens in the reduce function. The analyst/user who authors the Pig Latin program does not need to worry about transforming the Pig Latin script into a MapReduce program. It all happens behind the scenes and is taken care of by the Pig engine. The analyst just needs to focus on authoring the Pig Latin script.

SInce Pig Latin is a data processing language, it supports both *scalar* and *complex* data types. The supported scalar data types are: *int*, *long*, *double*, *chararray*, and *bytearray*.

The supported complex data types are:

- **map**: an associative array that is composed of a collection of key → value pairs such that each possible key appears at most once in the collection
- **tuple**: an ordered list of data elements that constitute the tuple can be of any scalar or complex type
- **bag**: an unordered collection of tuples

Anatomy of a Basic Pig Latin Program

As highlighted earlier, a Pig Latin program is a sequence of steps where each step is a single high-level data transformation. The transformations support relational-style operations such as filter, union, group, and join. Below is an example of some of the steps used when authoring a Pig Latin script.

Step 1

```
students = LOAD 'students.txt' as (first:chararray, last:chararray,
age:int, dept:chararray);
```

In the above step, a file is loaded into a *relation* which has a defined schema.

Step 2

```
students_filtered  = FILTER students BY age >= 18;
```

In this step, all rows that do not fulfill the predicate are filtered out.

© Copyright UC Regents

Step 3

```
dept_info = LOAD 'dept_info.txt' as (dept:chararray, credits:int)
```

Load another file into a relation called 'dept_info' which has a defined scheme.

Step 4

```
students_dept = JOIN students BY dept, dept_info BY dept;
```

Equi-join two relations on the 'dept' column.

Step 5

```
students_proj = FOREACH students_dept GENERATE
students::first as first, students::last as last,
students::dept as dept, dept_info::credits as credits;
```

In the above transformation, certain columns are chosen and renamed.

Invoking Pig

Pig can be executed in either local mode or Hadoop mode. When executed in local mode, a local *Java Virtual Machine* (JVM) is leveraged to execute the Pig Latin script. This option does not utilize the Hadoop cluster and is often used to iterate quickly under the development cycle. On the other hand, when executed in Hadoop mode, the compiled Pig Latin script will execute in a Hadoop cluster.

Option 1: Local Mode

```
pig -x local
```

Option 2: Hadoop Mode

```
pig -x hadoop
```

Hive Query Language (Hive QL)

Hive is a SQL-like data warehouse infrastructure built on top of Hadoop that compiles SQL queries into MapReduce jobs and runs the job in the cluster. The target users are data analysts who are conversant with SQL and have a need to perform ad-hoc queries, summarizations, and data manipulations on large volumes of data.

© Copyright UC Regents

Components of Hive

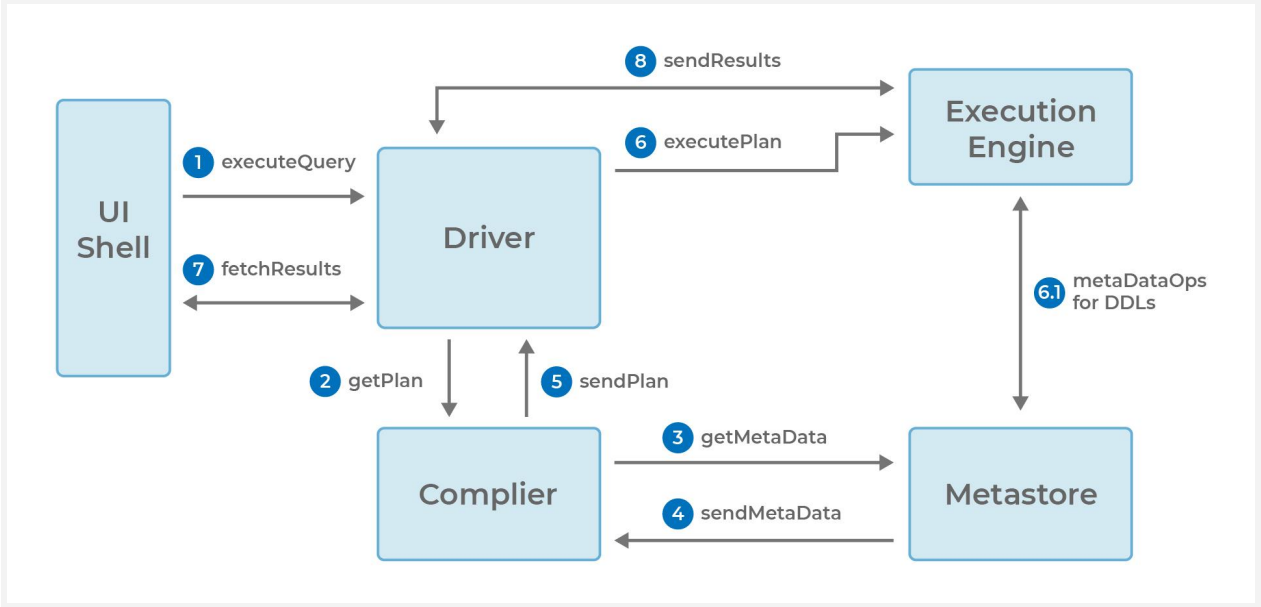

Figure 2: The Hive architecture

The following components make up the Hive architecture:

- **Shell**: allows for interactive queries and acts as the interface to Hive. The shell supports Command Line Interface (CLI), Web and JDBC (Java Database Connectivity) clients.
- **Driver**: establishes a communication path to the Hive data warehouse.
- **Compiler**: parses the queries and ascertains an optimal query execution plan.
- **Execution Engine**: executes the queries.
- **Metastore**: stores the schema (i.e., table definitions) and information on all the data file locations in HDFS.

Hive Data Model

The data model for Hive is decomposed into the following:

- **Tables** represent the high-level structure. Tables are created with *typed* columns where each column has a specific data type associated with it.
- **Partitions** equate to sections of tables which foster an optimized table structure for data access. For example, a partition can be constructed based on date ranges

© Copyright UC Regents

which would enable more performant data access since partitions effectively help you divide and organize the data.

- **Buckets** are hashed partitions, i.e., smaller segments of a partition. When analyzing terabytes of data, there is often a need to ensure that the Hive job being executed is running successfully. In such cases, the Hive job should be executed against a bucket that contains a sample of the data.

Hive Physical Layout

- Hive tables are stored in Hive's warehouse directory in HDFS; the default path is `/user/hive/warehouse`
- Partitions and buckets form sub-directories of tables
- Actual data is stored in flat files

Query Data with Hive QL

Launching Hive

To launch Hive, type in 'hive' on the Cloudera VM Command Line Interface (CLI).

```
→ [cloudera@localhost examples] $hive
```

Loading Data

Data can be loaded from a *HDFS-resident folder* OR from the *local file system*.

Loading Data from a HDFS-resident Folder:

```
→ LOAD DATA INPATH 'mydata' [OVERWRITE] INTO TABLE FOO;
```

Loading Data from a HDFS-resident Folder:

```
→ LOAD DATA LOCAL 'mydata' [OVERWRITE] INTO TABLE FOO;
```

Partitioning Data

```
→ LOAD DATA INPATH 'new_logs' INTO TABLE mytable PARTITION
(dt=2020-12-22);
```


© Copyright UC Regents

Executing the above command will create a data partition that includes records with a date of 2020-2022.

As highlighted earlier, a data 'PARTITION' provides the means to divide and organize the data - it optimizes the table structure for data access.

Bucketing Data

Bucketing enables the ability to execute on data sampling strategies. As highlighted earlier, when many terabytes of data are being analyzed, there is often a need to ensure that the Hive job being executed is running successfully. In such cases, the Hive job should be executed against a bucket that contains a sample of the data. Effectively, the HIVE query is executed on only a fraction of the data.

```
→ CREATE TABLE PURCHASERS (id INT,cost DOUBLE, msg STRING) CLUSTERED BY Id
INTO 32 BUCKETS;
```

In the above case, the 'id' values are being bucketed into 32 buckets.

The case below indicates *sampling* data from a specific bucket, i.e., Bucket 5:

```
→ SELECT avg(cost) FROM PURCHASERS TABLESAMPLE (BUCKET 5 OUT OF 32);
```

Manipulating Tables in HIVE

The following DDL (data definition language) commands can be utilized to manipulate tables in HIVE:

```
SHOW TABLES
CREATE TABLE
DESCRIBE TABLE
ALTER TABLE
DROP TABLE
```

Create Table

```
→ CREATE TABLE MOVIE
→ (id INT, name STRING, year INT)
→ ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
→ LOCATION '/user/training/movie';
```


© Copyright UC Regents

Describe Table

```
→ DESCRIBE MOVIE;
```

Selecting Data in HIVE

```
→ SELECT * from MOVIE LIMIT 10;
```

Storing Output Results in HDFS

To store results in HDFS, include the following command before the “select” statement.

```
→ INSERT OVERWRITE TABLE newTable
```

The output results are stored in the ‘*newTable’* table; results are just files within the newTable directory. Once the data is stored in HDFS, it can be used in other MapReduce jobs.

References

Lam, C. (2011). Chapter 10. Programming with Pig. *Hadoop in Action.*

© Copyright UC Regents