American International Journal of Research in Science, Technology, Engineering & Mathematics

Available online at http://www.iasir.net

ISSN (Print): 2328-3491, ISSN (Online): 2328-3580, ISSN (CD-ROM): 2328-3629

AIJRSTEM is a refereed, indexed, peer-reviewed, multidisciplinary and open access journal published by International Association of Scientific Innovation and Research (IASIR), USA

(An Association Unifying the Sciences, Engineering, and Applied Research)

Big Data Analytics using Hadoop Components like Pig and Hive

Sanjeev Dhawan[1], Sanjay Rathee[2]

[1]Faculty of Computer Science & Engineering, [2]Research Scholar

Department of Computer Science & Engineering, University Institute of Engineering and Technology, Kurukshetra University, Kurukshetra-136 119, Haryana, INDIA.

Abstract: *In big data world, Hadoop Distributed File System (HDFS) is very popular. It provides a framework for storing data in a distributed environment and also has set of tools to retrieve and process. These data set using map-reduce concept. In this paper, a thorough research has been carried to discuss that how big data analytics can be performed on data stored on Hadoop distributed file system using Pig and Hive. Apache Pig and Hive are two projects which are layered on top of Hadoop, and provide higher-level language to use Hadoop's MapReduce library. In this paper, first of all, the basic concepts of Pig and Hive are introduced. In part II, a map-reduce job using Pig has been explained. In part III, a map-reduce job using Hive is discussed. The final section of this paper compares and concludes both techniques.*

Keywords: *analysis; design; mapreduce; performance; query.*

I. INTRODUCTION

Apache's Pig and Hive are two major projects which are lying on top of Hadoop, and provide higher-level language to use Hadoop's MapReduce library. Pig provides the scripting language to describe operations like the reading, filtering and transforming, joining, and writing data which are exactly the same operations that MapReduce was originally designed for. Instead of expressing these operations in thousands of lines of Java code which uses MapReduce directly, Apache Pig lets the users express them in a language that is not unlike a bash or perl script. For prototyping and rapidly developing MapReduce-based jobs Pig is excellent, as opposed to coding MapReduce jobs in Java itself. If Apache Pig is "scripting for Hadoop", then Apache Hive is "SQL like queries for Hadoop". And Hive offers even more specific and higher-level language, to query data by running Hadoop jobs, instead of directly scripting step-by-step all operation of several MapReduce jobs on Hadoop. The language is, very much SQL-like, by design. Apache Hive is still intended as a tool for long-running batch-oriented queries over a massive data and it's not "real-time" in any sense. Apache Hive is an excellent tool for analysts and business development types who are accustomed to the SQL-like queries and Business Intelligence systems. Hive will let them easily leverage your shiny new Hadoop cluster to perform ad-hoc queries or generate report data across data stored in storage systems.

II. PIG

Pig was initially developed at Yahoo Research around 2006 but moved into the Apache Software Foundation in 2007. Pig consists of a language and an execution environment. Pig's language, called as PigLatin, is a data flow language - this is the kind of language in which you program by connecting things together. Pig can operate on complex data structures, even those that can have levels of nesting. Unlike SQL, Pig does not require that the data must have a schema, so it is well suited to process the unstructured data. But, Pig can still leverage the value of a schema if you want to supply one. PigLatin is relationally complete like SQL, which means it is at least as powerful as a relational algebra. Turing completeness requires conditional constructs, an infinite memory model, and looping constructs. PigLatin is not Turing complete on itself, but it can be Turing complete when extended with User-Defined Functions.

A. Execution environment

There are two choices of execution environment: a local environment and distributed environment. A local environment is good for testing when you do not have a full distributed Hadoop environment deployed. You tell Pig to run in the local environment when you start Pig's command line interpreter by passing it the *-x* local option. You tell Pig to run in a distributed environment by passing *-x* mapreduce instead. Alternatively, you can start the Pig command line interpreter without any arguments and it will start it in the distributed environment. There are three different ways to run Pig. You can run your PigLatin code as a script, just by passing the name of your script file to the *pig* command. You can run it interactively through the grunt command line launched using Pig with no script argument. Finally, you can call into Pig from within Java using Pig's embedded form.

B. Piglatin

A PigLatin program is a collection of statements. A statement can either be an operation or a command. For example, to load data from a file, issue the LOAD operation with a file name as an argument. A command could be an HDFS command used directly within PigLatin such as the *ls* command to list, say, all files with an extension of txt in the current directory. The execution of a statement does not necessarily immediately result in a job running on the Hadoop cluster. All commands and certain operators, such as DUMP will start up a job, but other operations simply get added to a logical plan. This plan gets compiled to a physical plan and executed once a data flow has been fully constructed, such as when a DUMP or STORE statement is issued. Here are some of the kinds of statements in PigLatin. There are UDF statements that can be used to REGISTER a user-defined function into PigLatin and DEFINE a short form for referring to it. It has been mentioned that HDFS commands are a form of PigLatin statement. Other commands include MapReduce commands and Utility commands. There are also diagnostic operators such as DESCRIBE that works much like an SQL DESCRIBE to show you the schema of the data. The largest number of operators falls under the relational operators' category. Here the operators are used to LOAD data into the program, to DUMP data to the screen, and to STORE data to disk. There are special operators for filtering, grouping, sorting, combining and splitting data. The relational operators produce relations as their output. A relation is a collection of tuples. Relations can be named using an alias. For example, the LOAD operator produces a relation based on what it reads from the file you pass it. You can assign this relation to an alias, say, *x*. If you DUMP the relation aliased by *x* by issuing "DUMP *x*", you get the collection of tuples, one for each row of input data. In this example, there is just the one tuple. When we ran LOAD on the file, we specified a schema with the AS argument. We can see this schema by running the DESCRIBE operator on the relation aliased by *x*. Statements which contain relational operators may also contain expressions. There are many different kinds of expressions. For example, a constant expression is simply a literal such as a number. A field expression lets you reference a field by its position. You specify it with a dollar sign and a number. PigLatin supports the standard set of simple data types like integers and character arrays. It also supports three complex types: the tuple as discussed previously, the bag, which is simply an unordered collection of tuples, and a map which is a set of key-value pairs with the requirement that the keys have a type of chararray. PigLatin also supports functions. These include eval functions, which take in multiple expressions and produce a single expression. For example, you can compute a maximum using the MAX function. A filter function takes in a bag or map and returns a Boolean. For example, the is Empty function can tell you if the bag or map is empty. A load function can be used with the LOAD operator to create a relation by reading from an input file. A store function does the reverse. It reads in a relation and writes it out to a file. You can write your own eval, filter, load, or store functions using PigLatin's UDF mechanism. You write the function in Java, package it into a jar, and register the jar using the REGISTER statement. You also have the option to give the function a shorter alias for referring to it by using the DEFINE statement.

Here it is shown how to use Apache Pig to process the 1988 subset of the Google Books 1-gram records to produce a histogram of the frequencies of words of each length. A subset of this database (0.5 million records) has been stored in the file googlebooks-1988.csv stored at /BigDataUniversity/input directory.

1. First of all, examine the format of the Google Books 1-gram records using head command:

```
$ head -10 /BigDataUniversity/input/googlebooks-1988.csv
```

2. Now copy the data file into Hadoop File System.

```
$ hadoop fs -put /BigDataUniversity/input/googlebooks-1988.csv googlebooks-1988.csv
```

3. Start the Apache Pig.

```
$  pig
grunt>
```

Now it will use a Pig UDF to compute the length of each word which is located inside the piggybank.jar file. It uses the *REGISTER* command to load this jar file:

```
grunt>  REGISTER /opt/ibm/biginsights/pig/piggybank/java/piggybank.jar;
```

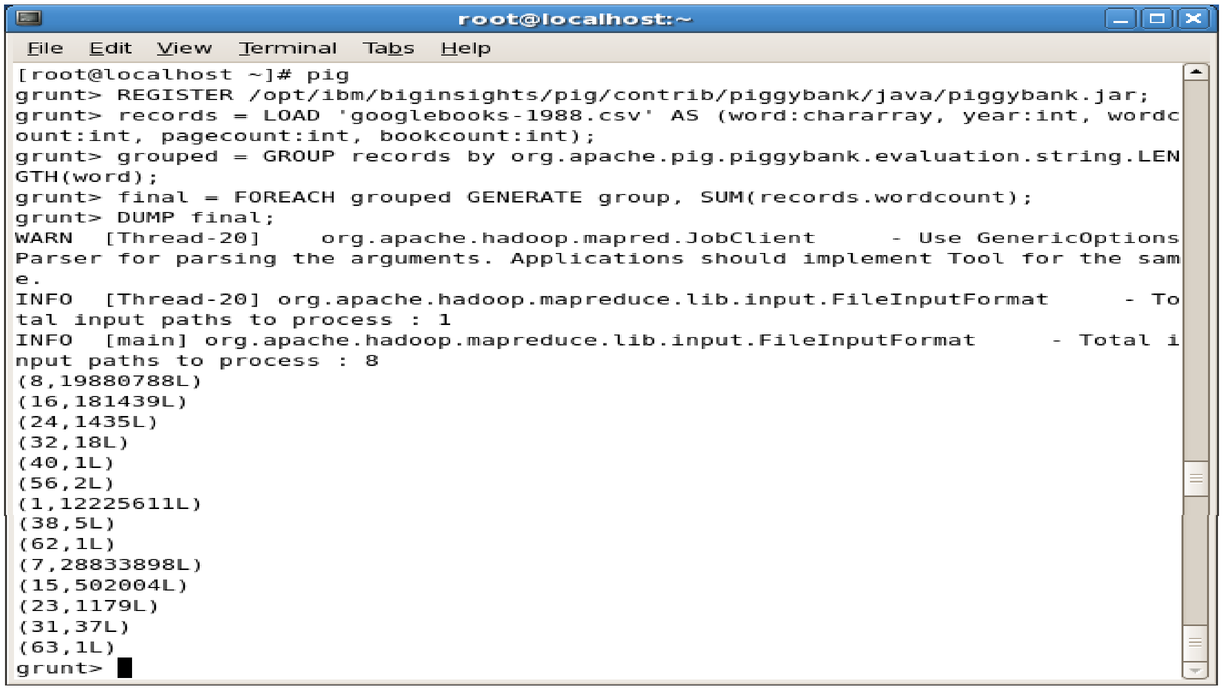
Now the first step to process the data is to LOAD it.

```
grunt> records = LOAD 'googlebooks-1988.csv' AS (word:chararray, wordcount:int, pagecount, bookcount:int);
```

It returns instantly. The processing may be delayed until the data needs to be reported.
To produce a histogram, we want to group by the length of the word:
grunt> grouped = GROUP records by org.apache.piggybank.evaluation.LENGTH(word);

At last, sum the word counts for each word length using SUM function with the FOREACH and GENERATE command.
grunt> final = FOREACH grouped GEN group, SUM(records.wordcount);
And use the DUMP command to print the result to the console. It will cause all the previous steps to be executed.
grunt> DUMP final;

Figure 1: Analysis using Pig



```
[root@localhost ~]# pig
grunt> REGISTER /opt/ibm/biginsights/pig/contrib/piggybank/java/piggybank.jar;
grunt> records = LOAD 'googlebooks-1988.csv' AS (word:chararray, year:int, wordc
ount:int, pagecount:int, bookcount:int);
grunt> grouped = GROUP records by org.apache.pig.piggybank.evaluation.string.LEN
GTH(word);
grunt> final = FOREACH grouped GENERATE group, SUM(records.wordcount);
grunt> DUMP final;
WARN  [Thread-20]    org.apache.hadoop.mapred.JobClient     - Use GenericOptions
Parser for parsing the arguments. Applications should implement Tool for the sam
e.
INFO  [Thread-20] org.apache.hadoop.mapreduce.lib.input.FileInputFormat     - To
tal input paths to process : 1
INFO  [main] org.apache.hadoop.mapreduce.lib.input.FileInputFormat     - Total i
nput paths to process : 8
(8,19880788L)
(16,181439L)
(24,1435L)
(32,18L)
(40,1L)
(56,2L)
(1,12225611L)
(38,5L)
(62,1L)
(7,28833898L)
(15,502004L)
(23,1179L)
(31,37L)
(63,1L)
grunt> 
```

III. HIVE

Hive is a technology developed at Facebook that turns Hadoop into a data warehouse complete with a dialect of SQL for querying. Being a SQL dialect, HiveQL is a declarative language. In PigLatin, you specify the data flow, but in Hive we describe the result we want and Hive figures out how to build a data flow to achieve that result. Unlike Pig, in Hive a schema is required, but you are not limited to only one schema. Like PigLatin and the SQL, HiveQL itself is a relationally complete language but it is not a Turing complete language. It can also be extended through UDFs just like Piglatin to be a Turing complete. Hive is a technology for turning the Hadoop into a data warehouse, complete with SQL dialect for querying it.

A. Configuring Hive

You can configure Hive using any one of three methods. You can edit a file called hive-site.xml. You can use this file to specify the location of your HDFS NameNode and your MapReduce JobTracker. You can also use it for specifying configuration settings for the metastore, a topic we will come to later. These same options can be specified when starting the Hive command shell by specifying a -hiveconf option. Finally, within the Hive shell, you can change any of these settings using the set command. There are three ways to run Hive. You can run it interactively by launching the hive shell using the hive command with no arguments. You can run a Hive script by passing the -f option to the hive command along with the path to your script file. Finally, you can execute a Hive program as one command by passing the -e option to the hive command followed by your Hive program in quotes.

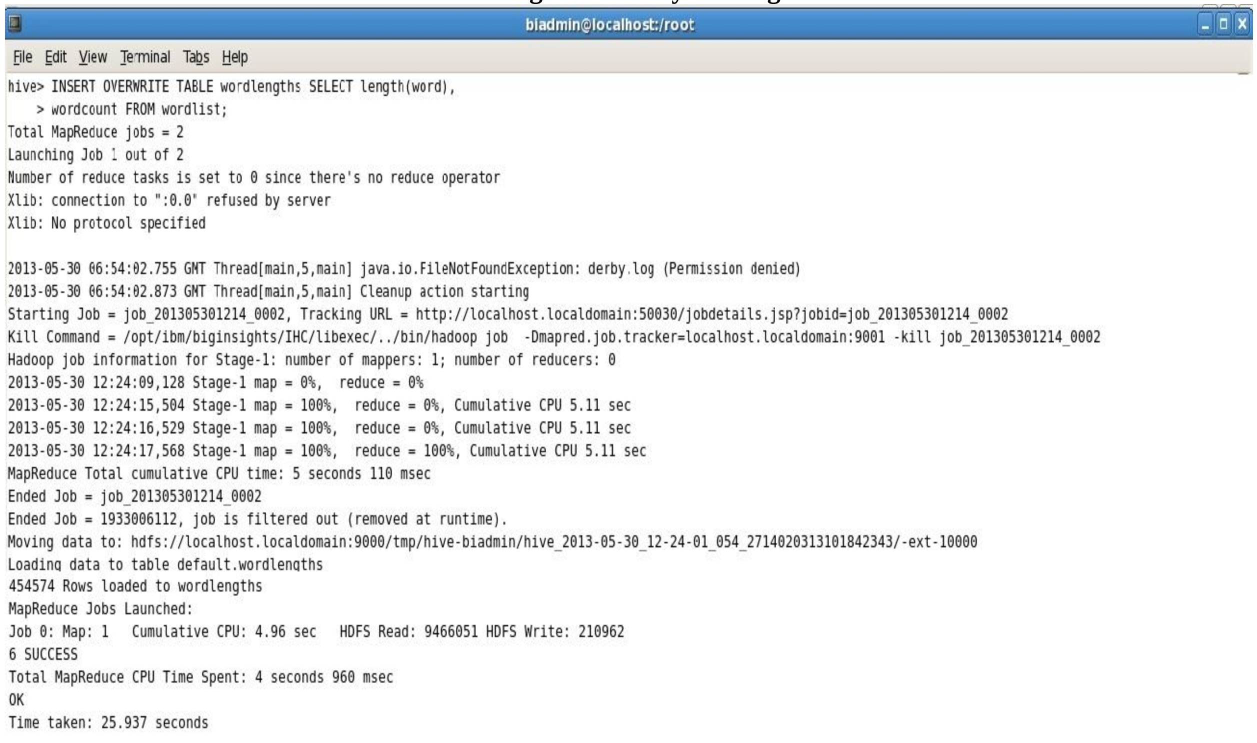
B. Services

Hive also supports launching services from the hive command. You can launch a service that lets you access Hive through Thrift, ODBC, or JDBC by passing service to the hive command followed by the word hive server. There is also a web interface to hive whose service is launched by following the service option with hive. You can also use a Hive service to run the hadoop command with the jar option the same as you could do directly, but with Hive jars on the classpath. Lastly, there is a service for an out of process metastore. The metastore stores the Hive metadata. There are three configurations you can choose for your metastore. First is embedded, which runs the metastore code in the same process with your Hive program and the database that backs the metastore is in the same process as well. The second option is to run it as local, which keeps the metastore code running in process, but moves the database into a separate process that the metastore code communicates with. The third option is to move the metastore itself out of process as well. This can be useful if you wish to share a metastore with other users.

Like other SQL databases, Hive works in terms of tables. There are two kinds of tables you can create: managed tables whose data is managed by Hive and external tables whose data is managed outside of Hive. When you load a file into a managed table, Hive moves the file into its data warehouse. When you drop such a table, both the data and metadata are deleted. When you load a file into an external table, no files are moved. Dropping the table only deletes the metadata. The data is left alone. External tables are useful for sharing data between Hive and other Hadoop applications or when you wish to use more than one schema on the same data. Hive offers a way to speed up queries of subsets of your data. You can partition your data based on the value of a column. When creating a table, you can specify a PARTITION BY clause to specify the column used to partition the data. Then, when loading the data, you specify a PARTITION clause to say what partition you are loading. You can then query individual partitions more efficiently than you could unpartitioned data. The SHOW PARTITIONS command will let you see a table's partitions.

Another option Hive provides for speeding up queries is bucketing. Like partitioning, bucketing splits up the data by a particular column, but in bucketing you do not specify the individual values for that column that correspond to buckets, you simply say how many buckets to split the table into and let Hive figure out how to do it. The benefit of bucketing is that imposes extra structure on your table that can be used to speed up certain queries, such as joins on bucketed columns. It also improves performance when sampling your data. You specify the column to bucket on and the number of buckets using the CLUSTERED BY clause. If you wanted the bucket to be sorted as well, you use the SORTED BY clause. If you wish to query a sample of your data rather than the whole data set, you can use the TABLESAMPLE command and it will take advantage of bucketing. Hive has multiple ways of reading and storing your data on disk. There is a delimited text format and two binary formats.

Figure 2: Analysis using Hive



```
hive> INSERT OVERWRITE TABLE wordlengths SELECT length(word),
    > wordcount FROM wordlist;
Total MapReduce jobs = 2
Launching Job 1 out of 2
Number of reduce tasks is set to 0 since there's no reduce operator
Xlib: connection to ":0.0" refused by server
Xlib: No protocol specified

2013-05-30 06:54:02.755 GMT Thread[main,5,main] java.io.FileNotFoundException: derby.log (Permission denied)
2013-05-30 06:54:02.873 GMT Thread[main,5,main] Cleanup action starting
Starting Job = job_201305301214_0002, Tracking URL = http://localhost.localdomain:50030/jobdetails.jsp?jobid=job_201305301214_0002
Kill Command = /opt/ibm/biginsights/IHC/libexec/../bin/hadoop job  -Dmapred.job.tracker=localhost.localdomain:9001 -kill job_201305301214_0002
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 0
2013-05-30 12:24:09,128 Stage-1 map = 0%,  reduce = 0%
2013-05-30 12:24:15,504 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 5.11 sec
2013-05-30 12:24:16,529 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 5.11 sec
2013-05-30 12:24:17,568 Stage-1 map = 100%,  reduce = 100%, Cumulative CPU 5.11 sec
MapReduce Total cumulative CPU time: 5 seconds 110 msec
Ended Job = job_201305301214_0002
Ended Job = 1933006112, job is filtered out (removed at runtime).
Moving data to: hdfs://localhost.localdomain:9000/tmp/hive-biadmin/hive_2013-05-30_12-24-01_054_2714020313101842343/-ext-10000
Loading data to table default.wordlengths
454574 Rows loaded to wordlengths
MapReduce Jobs Launched:
Job 0: Map: 1   Cumulative CPU: 4.96 sec   HDFS Read: 9466051 HDFS Write: 210962
6 SUCCESS
Total MapReduce CPU Time Spent: 4 seconds 960 msec
OK
Time taken: 25.937 seconds
```

Here it is shown how to use Hive to process the 1988 subset of the Google Books 1-gram records to produce a histogram of the frequencies of words of each length. A subset of this database (0.5 million records) has been stored in the file googlebooks-1988.csv stored at /BigDataUniversity/input directory.

1. First of all start hive.
***$** opt/ibm/biginsights/hive/bin/hive*
hive>

2. Create a new table called wordlist.
hive> CREATE TABLE wordlist (word STRING, year INT, pagecount INT, bookcount INT) ROW FORMAT DELIMITED TERMINATED BY '\t';

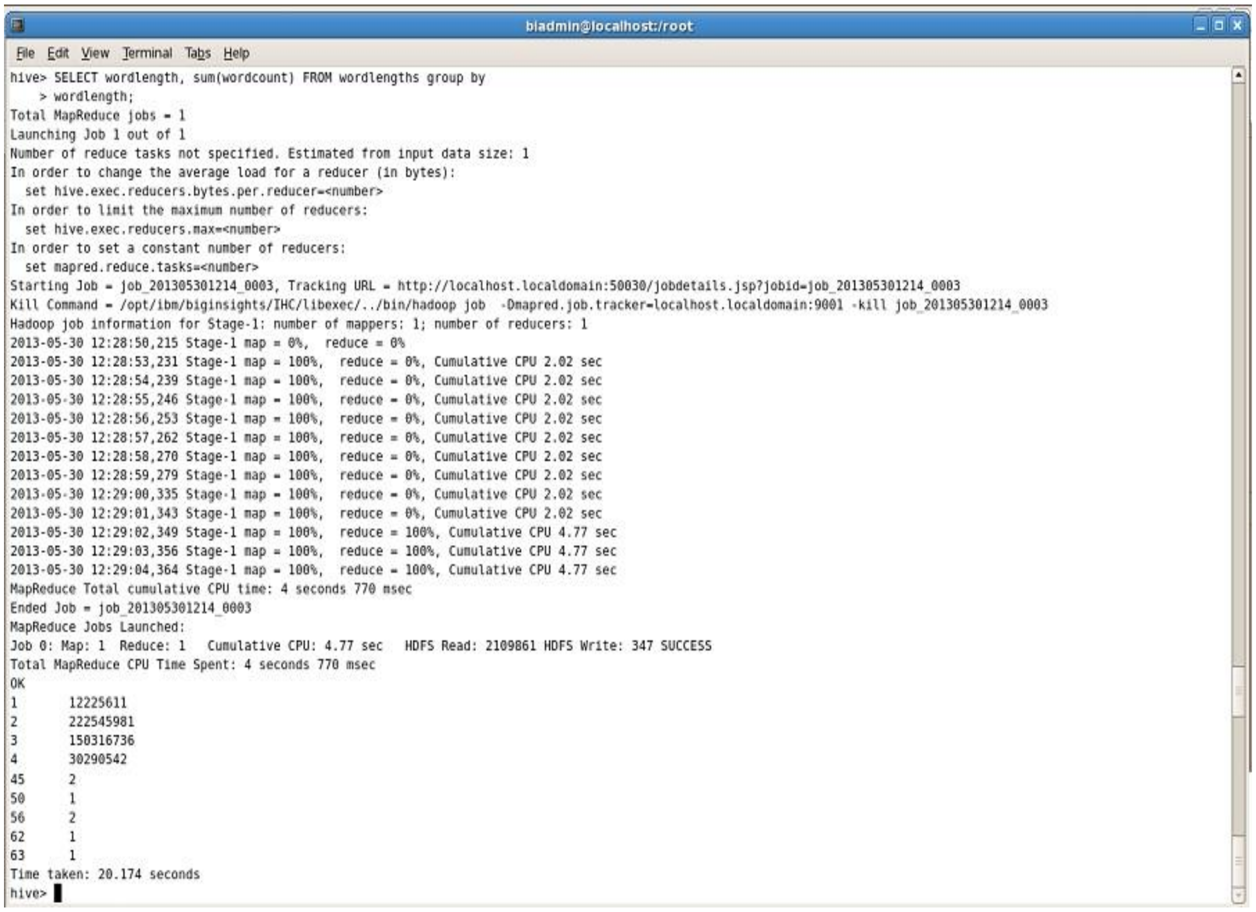
3. Now load the data from the googlebooks-1988.csv file into wordlist table.
hive> LOAD DATA LOCAL INPATH '/BigDataUniversity/input/googlebooks-1988.csv' OVERWRITE TABLE wordlist;

4. Create a new table named wordlengths to store the counts for each word length for our histogram.
hive> CREATE TABLE wordlengths (wordcount INT);

5. Fill out the wordlengths table with word length data from the wordlist table calculated with the length function.
hive> INSERT TABLE wordlengths SELECT length, wordcount FROM wordlist;

6. Finally produce a histogram by summing the word counts grouped by the word length.
*hive> SELECT wordlength, sum FROM wordlengths group by wordlength***;**

Figure 3: Histogram for Hive



```
hive> SELECT wordlength, sum(wordcount) FROM wordlengths group by
    > wordlength;
Total MapReduce jobs = 1
Launching Job 1 out of 1
Number of reduce tasks not specified. Estimated from input data size: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapred.reduce.tasks=<number>
Starting Job = job_201305301214_0003, Tracking URL = http://localhost.localdomain:50030/jobdetails.jsp?jobid=job_201305301214_0003
Kill Command = /opt/ibm/biginsights/IHC/libexec/../bin/hadoop job  -Dmapred.job.tracker=localhost.localdomain:9001 -kill job_201305301214_0003
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 1
2013-05-30 12:28:50,215 Stage-1 map = 0%,  reduce = 0%
2013-05-30 12:28:53,231 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 2.02 sec
2013-05-30 12:28:54,239 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 2.02 sec
2013-05-30 12:28:55,246 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 2.02 sec
2013-05-30 12:28:56,253 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 2.02 sec
2013-05-30 12:28:57,262 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 2.02 sec
2013-05-30 12:28:58,270 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 2.02 sec
2013-05-30 12:28:59,279 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 2.02 sec
2013-05-30 12:29:00,335 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 2.02 sec
2013-05-30 12:29:01,343 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 2.02 sec
2013-05-30 12:29:02,349 Stage-1 map = 100%,  reduce = 100%, Cumulative CPU 4.77 sec
2013-05-30 12:29:03,356 Stage-1 map = 100%,  reduce = 100%, Cumulative CPU 4.77 sec
2013-05-30 12:29:04,364 Stage-1 map = 100%,  reduce = 100%, Cumulative CPU 4.77 sec
MapReduce Total cumulative CPU time: 4 seconds 770 msec
Ended Job = job_201305301214_0003
MapReduce Jobs Launched:
Job 0: Map: 1  Reduce: 1   Cumulative CPU: 4.77 sec   HDFS Read: 2109861 HDFS Write: 347 SUCCESS
Total MapReduce CPU Time Spent: 4 seconds 770 msec
OK
1       12225611
2       222545981
3       150316736
4       30290542
45      2
50      1
56      2
62      1
63      1
Time taken: 20.174 seconds
hive> █
```

IV. CONCLUSION

From this study, it is concluded that in PigLatin, the data flow is specified. But in Hive, the results are described and it figures out how to build a data flow to achieve the desired results under distributed environment. Unlike Pig, in Hive a schema is required, but you are not limited to only one schema. Like PigLatin and the SQL, HiveQL itself is a relationally complete language but it is not a Turing complete language. It can also be extended through UDFs just like Piglatin to be a Turing complete. Hive is a technology for turning the Hadoop into a data warehouse, complete with SQL dialect for querying it. In this paper, initially a mapreduce job using Apache Pig is successfully created and then it is used to analyze a big database to get required results. Finally a mapreduce job is created using Hive and then exercised it to analyze a big database to get results. The final results show that the analysis performed by both of the mapreduce machines is successful. The performance of both was nearly same.

V. REFERENCES

[1] Alexandros Biliris, “An Efficient Database Storage Structure for Large Dynamic Objects”, IIEEE Data Engineering Conference, Phoenix, Arizona, pp. 301-308, February 1992.

[2] An Oracle White Paper, “Hadoop and NoSQL Technologies and the Oracle Database”, February 2011.

[3] Cattell, “Scalable sql and nosql data stores”, ACM SIGMOD Record, 39(4), pp. 12–27, 2010.

[4] Russom, “Big Data Analytics”, TDWI Research, 2011.

[5] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, “Google file system”, 2003.

[6] Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach,Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber , Fay Chang, Jeffrey Dean , “Bigtable: A Distributed Storage System for Structured Data”, OSDI 2006.

[7] Zhifeng YANG, Qichen TU, Kai FAN, Lei ZHU, Rishan CHEN, BoPENG, “Performance Gain with Variable Chunk Size in GFS-like File Systems”, Journal of Computational Information Systems4:3 pp- 1077-1084, 2008.

[8] Sam Madden, “From Databases to Big Data”, IEEE Computer Society, 2012.