

INTRODUCTION TO HIVE

4.1 What is Hive?

Hive is a Data Warehousing tool. Hive is used to query structured data built on top of Hadoop. Facebook created Hive component to manage their ever-growing volumes of log data. Hive makes use of the following:

1. HDFS for Storage.
2. MapReduce for execution.
3. Stores metadata in an RDBMS.

Hive provides HQL (Hive Query Language) which is similar to SQL. Hive compiles SQL queries into Reduce jobs and then runs the job in the Hadoop Cluster. Hive provides extensive data type functions and formats for data summarization and analysis. Hive suitable for Data warehousing applications. Process batch jobs on huge data that is immutable. Examples: Weblogs, Applications logs.

History of Hive and Recent Releases of Hive

In 2007, Hive was born at Facebook to analyze their incoming log data. In 2008 Hive became Apache Hadoop sub-project. Hive recent releases are shown in Figure 4.1.

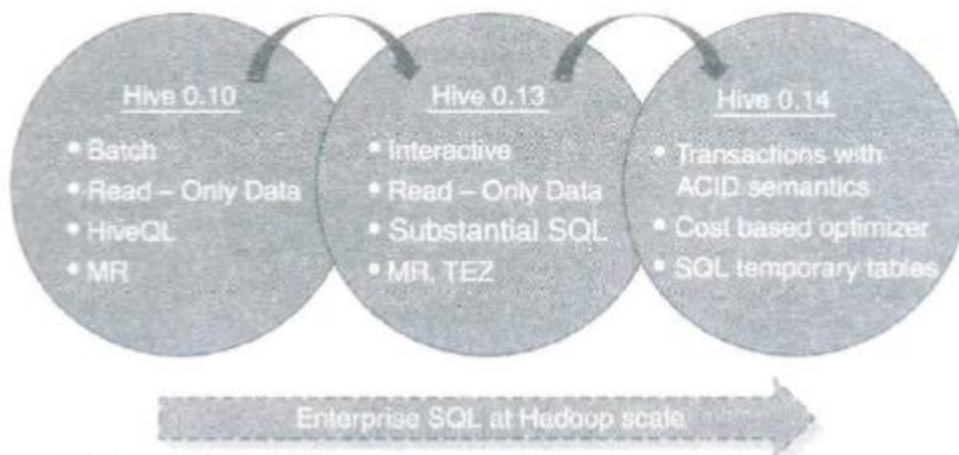


Figure 4.1: Recent releases of Hive

Hive features

- It is similar to SQL.
- HQL is easy to code.
- Hive supports rich data types such as structs, lists, and maps.
- Hive supports SQL filters, group-by and order-by clauses.
- CustomTypes, Custom Functions can be defined.

Hive Integration and Work Flow: Figure 4.2 depicts the flow of log file analysis. Hourly Log Data can be stored directly into HDFS and then data cleansing is performed on the log. Finally Hive table(s) can be created to query the log file. A database contains several tables. Each table is constituted of rows and columns. In Hive, tables stored as a folder and partition tables are stored as a sub-directory. Bucketed tables are stored as a file.

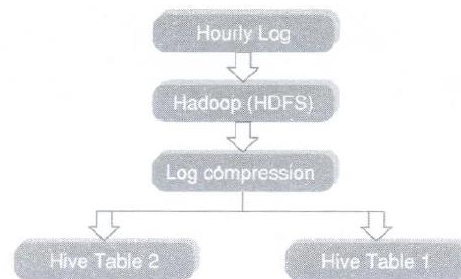


Figure 4.2: Flow of Log analysis file

Hive Data Units

1. Databases: The namespace for tables.
2. Tables: Set of records that have similar schema.
3. Partitions: Logical separations of data based on classification of given information as per specific attributes. Once hive has partitioned the data based on a specified key, it starts to assemble the records into specific folders as and when the records are inserted.
4. Buckets (or Clusters): Similar to partitions but uses hash function to segregate data and determines the cluster or bucket into which the record should be placed.

Figure 4.3 shows how these data units are arranged in a Hive Cluster. Figure 4.4 describes the resemblance of Hive structure with database.

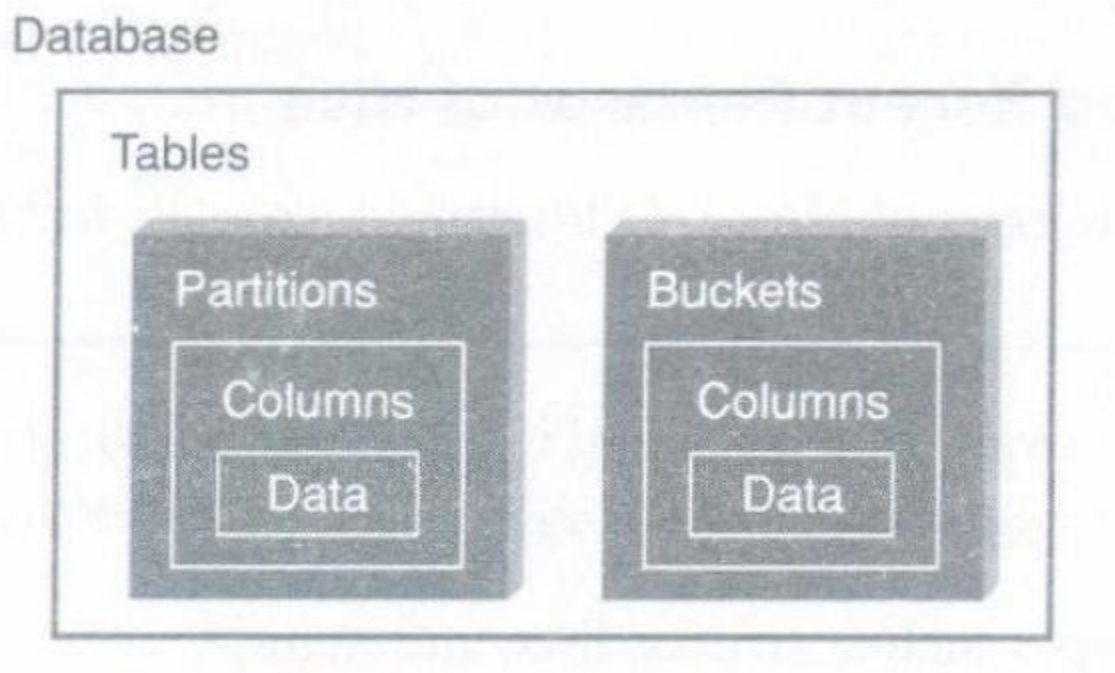


Figure 4.3: Data units as arranged in HIVE

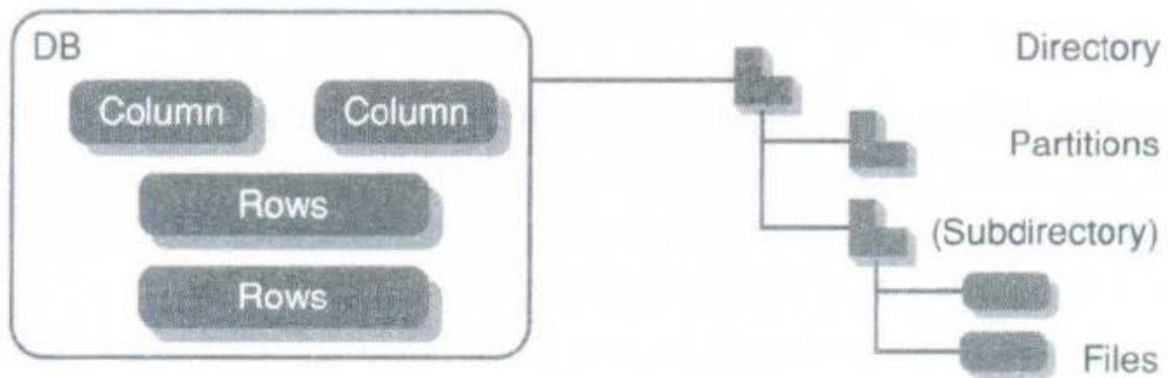


Figure 4.4: Semblance of Hive structure with database

4.2 Hive Architecture

Hive Architecture is depicted in Figure 4.5. The various parts are as follows:

1. Hive Command-Line Interface (Hive CLI): The most commonly used interface to interact with Hive.
2. Hive Web Interface: It is a simple Graphic User Interface to interact with Hive and to execute query.
3. Hive Server: This is an optional server. This can be used to submit Hive Jobs from a remote client.
4. JDBC / ODBC: Jobs can be submitted from a JDBC Client. One can write a Java code to connect to Hive and submit jobs on it.
5. Driver: Hive queries are sent to the driver for compilation, optimization and execution.
6. Metastore: Hive table definitions and mappings to the data are stored in a Metastore. A Metastore consists of the following:
 - Metastore service: Offers interface to the Hive.
 - Database: Stores data definitions, mappings to the data and others.

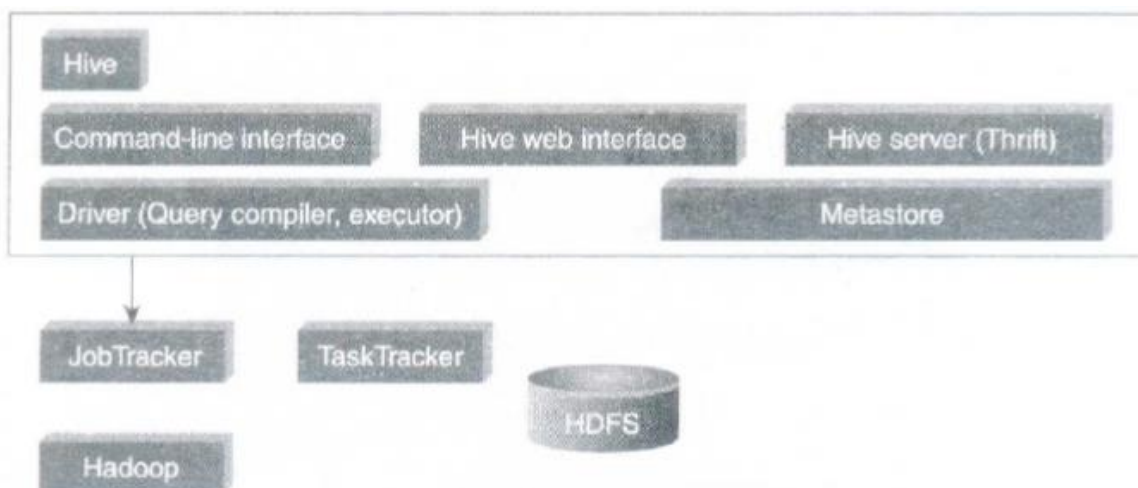


Figure 4.5: Hive Architecture

The metadata which is stored in the metastore includes IDs of Database, IDs of Tables, IDs of Indexes, etc. The time of creation of a Table, the Input Format used for a Table, the Output

Format used for a table, etc. The metastore is updated whenever a table is created or deleted from Hive. There are three kind of metastore.

1. Embedded Metastore: This metastore is mainly used for unit tests. Here, only one process is allowed to connect to the metastore at a time. This is the default metastore for Hive. It is Apache Derby Database. In this metastore, both the database and the metastore service runs, embedded in the main Hive Server process. Figure 4.6 shows an Embedded Metastore.

2. Local Metastore: Metadata can be stored in any RDBMS component like MySQL. Local metastore allows multiple connections at a time. In this mode, the Hive metastore service runs in the main Hive Server process, but the metastore database runs in a separate process, and can be on a separate host. Figure 4.7 shows a Local Metastore.

3. Remote Metastore: In this, the Hive driver and the metastore interface run on different JVMs (which can run on different machines as well) as in Figure 4.7. This way the database can be fire-walled from the Hive user and also database credentials are completely isolated from the users of Hive.

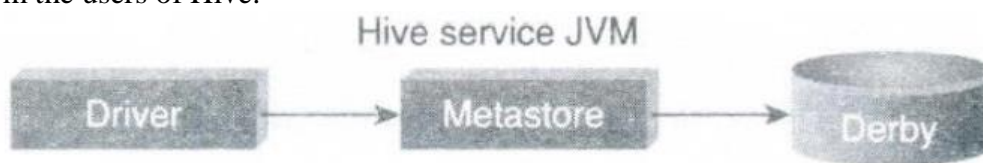


Figure 4.6: Embedded Metastore

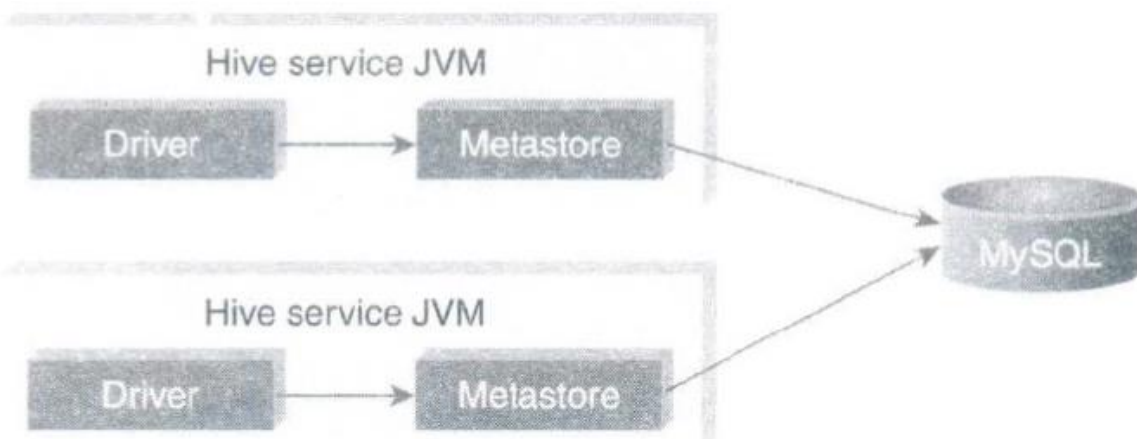


Figure 4.7: Local Metastore

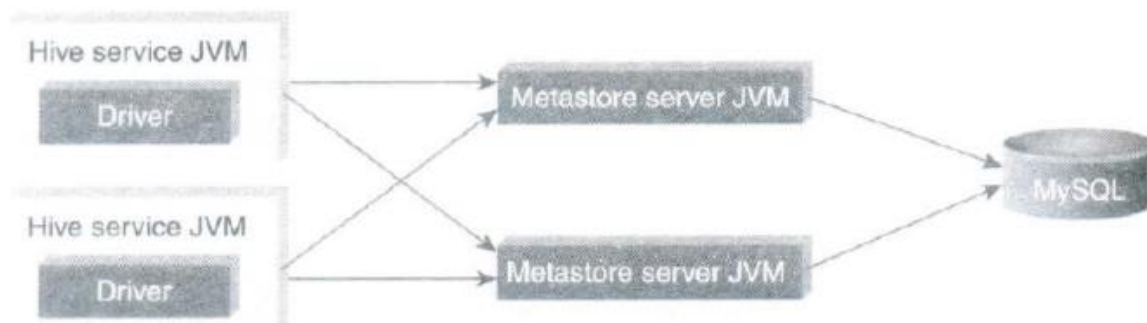


Figure 4.8: Remote Metastore

4.3 Hive Data Types

Hive Data types are classified as Primitive data types and Collection data types.

Primitive Data Types-Can be classified as Numeric, String, and Miscellaneous types

Numeric Data Type

TINYINT 1 - byte signed integer

SMALLINT 2 - byte signed integer

INT 4 - byte signed integer

BIGINT 8 - byte signed integer

FLOAT 4 - byte single-precision floating-point

DOUBLE 8 - byte double-precision floating-point number

String Types

STRING

VARCHAR Only available starting with Hive 0.12.0

CHAR Only available starting with Hive 0.13.0

Strings can be expressed in either single quotes (') or double quotes (")

Miscellaneous Types

BOOLEAN

BINARY only available starting with Hive

Data Analytics Using Hadoop Page 111

Collection Data Types

Collection Data Types

STRUCT Similar to T struct. Fields are accessed using dot notation. E.g.:
struct('John', 'Doe')

MAP A collection of key - value pairs. Fields are accessed using []
notation. E.g.: map('first', 'John', 'last', 'Doe')

ARRAY Ordered sequence of same types. Fields are accessed using
array index. E.g.: array('John', 'Doe')

4.4 Hive File Format

The file formats in Hive specify how records are encoded in a file.

Text File: The default file format is text file. In this format, each record is a line in the file. In text file, different control characters are used as delimiters. The delimiters are ^A (octal 001, separates all fields), ^B (octal 002, separates the elements in the array or struct), ^C (octal 003, separates key-value pair), and \n. The term field is used when overriding the default delimiter. The supported text files are CSV and TSV. JSON or XML documents too can be specified as text file.

Sequential File: Sequential files are Bat files that store binary key-value pairs. It includes compression support which reduces the CPU, I/O requirement.

RCFILE (Record Columnar File)

RCFile stores the data in Column Oriented Manner which ensures that Aggregation operation is not an expensive operation. For example, consider a table which contains four columns as shown below:

C1	C2	C3	C4
11	12	13	14
21	22	23	24
31	32	33	34
41	42	43	44
51	52	53	54

Instead of only partitioning the table horizontally like the row-oriented DBM5 (row-store), RCFile partitions this table first horizontally and then vertically to serialize the data. Based on the user-specified value, first the table is partitioned into multiple row groups horizontally as shown below.

Row Group 1				Row Group 2			
C1	C2	C3	C4	C1	C2	C3	C4
11	12	13	14	41	42	43	44
21	22	23	24	51	52	53	54
31	32	33	34				

The first table is partitioned into two row groups by considering three rows as the size of each row group. Next, in every row group RCFile partitions the data vertically like column-store. So the table will be serialized as shown below:

Row Group 1	Row Group 2
11, 21, 31;	41, 51;
12, 22, 32;	42, 52;
13, 23, 33;	43, 53;
14, 24, 34;	44, 54;

4.5 Hive Query Language

The query language provides basic SQL like operations. Here are few of the tasks which HQL can do easily.

1. Create and manage tables and partitions.
2. Supports various Relational, Arithmetic, and Logical Operators.
3. Evaluate functions.
4. Download the contents of a table to a local directory or result of queries to HDFS directory.

DDL (Data Definition Language) Statements

These statements are used to build and modify the tables and other objects in the database.

The DDL commands are as follows:

1. Create/Drop/Alter Database
2. Create/Drop/Truncate Table
3. Alter Table/Partition/Column
4. Create/Drop/Alter View
5. Create/Drop/Alter Index
6. Show
7. Describe

DML (Data Manipulation Language) Statements

These statements are used to retrieve, store, modify, delete, and update data In database. The DML commands are as follows:

1. Loading files into table.
2. Inserting data into Hive Tables from queries.

Note: Hive0.14 supports update, delete, and transaction operations.

Starting Hive Shell

To start Hive, go to the installation path of hive and type as below;

```
root@volgalnx005 ~] # hive
```

The Section has been designed as follows

Objective: What is it that we are trying to achieve here?

Input (optional): What is the input that has been given to us to act upon?

Act: The actual statement/command to accomplish the task at hand.

Outcome: The result/output as a consequence of executing the statement.

Database: A database is like a container for data. It has collection of tables which houses data.

Objective: To create a database named "STUDENTS" with comments and database properties.

Act:

```
CREATE DATABASE IF NOT EXISTS STUDENTS COMMENT 'STUDENT Details'
WITH DBPROPERTIES ('creator' = 'JOHN');
```

Outcome:

Objective: To display a list of all databases.

Act:

```
SHOW DATABASES;
```

Outcome:

Objective: To describe a database.

Act:

```
DESCRIBE DATABASE STUDENTS;
```

Note: Shows only DB name, comment, and DB directory.

Objective: To describe the extended database.

Act:

```
DESCRIBE DATABASE EXTENDED STUDENTS;
```

Objective: To alter the database properties.

Act:

```
ALTER DATABASE STUDENTS SET DBPROPERTIES ('edited_by' = 'JAMES');
```

Note: In Hive, it is not possible to unset the DB properties.

Objective: To make the database as current working database.

Act:

```
USE STUDENTS;
```

Objective: to drop Database.

Act:

```
DROP DATABASE STUDENTS;
```

Note: Hive creates database in the warehouse directory of Hive as shown below:

Tables: Hive provide two kind of table: Managed and External

Managed Table

1. Hive stores the Managed tables under the warehouse folder under Hive.
2. The complete life cycle of table and data is managed by Hive,
3. When the internal table is dropped, it drops the data as well as the metadata.

Objective: To create managed table named 'STUDENT'.

Act:

```
CREATE TABLE IF NOT EXISTS STUDENT(rollno INT,name STRING,gpa FLOAT)
ROW
```


FORMAT DELIMITED FIELDS TERMINATED BY '\t';

Objective: To describe the "STUDENT" table,

Act:

DESCRIBE STUDENT;

Note: Hive creates managed table in the warehouse directory of Hive

External or Self-Managed Table

1. When the table is dropped, it retains the data in the underlying location.

2. External keyword is used to create an external table.

3. Location needs to be specified to store the dataset in that particular location.

Objective: To create external table named 'EXT_STUDENT'.

Act:

CREATE EXTERNAL TABLE IF NOT EXISTS EXT_STUDENT(rollno INT, name STRING, gpa FLOAT) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' LOCATION '/STUDENT_INFO';

Objective: To load data into the table from file named student.tsv.

Act:

LOAD DATA LOCAL INPATH '/root/hivedemos/student.tsv' OVERWRITE INTO TABLE EXT_STUDENT;

Note: Local keyword is used to load the data from the local file system. To load the data from HDFS, remove local keyword from the statement.

Collection Data Types

Objective: To work with collection data types.

Input:

1001 , John, Smith:Jones, Mark1!45:Mark2!46:Mark3!43

1002, Jack, Smith:Jones, Mark1!46:Mark2!47:Mark3!42

Act:

CREATE TABLE STUDENT_INFO (rollno INT, name String, sub ARRAY<STRING>, marks MAP<STRING,INT>)

ROW FORMAT DELIMITED FIELDS TERMINATED BY ','

COLLECTION ITEMS TERMINATED BY ':'

MAP KEY TERMINATED BY '!';

LOAD DATA LOCAL INPATH '/root/hivedemos/studentinfo.csv' INTO TABLE STUDENT_INFO;

Querying Table:

Objective: To retrieve the student details from "EXT_STUDENT" table.

Act:

SELECT * FROM EXT_STUDENT;

Objective: Querying Collection Data Types.

Act:

SELECT * FROM STUDENT_INFO;

SELECT NAME, SUB FROM STUDENT_INFO;

// To retrieve value of Mark1

SELECT NAME, MARKS ['Mark1'] FROM STUDENT_INFO;

// To retrieve subordinate (array) value

SELECT NAME, SUB[0] FROM STUDENT_INFO;

4.6 Partitions

In Hive, the query reads the entire dataset even though a where clause filter is specified on a particular column. This becomes a bottleneck in most of the MapReduce jobs as it involves

huge degree of I/O. So it is necessary to reduce I/O required by the MapReduce job to improve the performance of the query. A very common method to reduce I/O is data partitioning.

Partitions split the larger dataset into more meaningful chunks.

Hive provides two kinds of partitions: Static Partition and Dynamic Partition.

Objective: To load data into a dynamic partition table from table.

Act:

```
SET hive.exec.dynamic.partition = true;
```

```
SET hive.exec.dynamic.partition.mode = nonstrict;
```

Note: The dynamic partition strict mode requires at least one static partition column. To turn this off,

```
set hive.exec.dynamic.partition.mode=nonstrict
```

```
INSERT OVERWRITE TABLE DYNAMIC_PART_STUDENT PARTITION (gpa)
```

```
SELECT rollno, name, gpa from EXT_STUDENT;
```

Bucketing

Bucketing is similar to partition. However, there is a subtle difference between partition and bucketing. In a partition, you need to create partition for each unique value of the column.

This may lead to situations where you may end up with thousands of partition. This can be avoided by using Bucketing in which you can limit the number of buckets to create. A bucket is a file whereas a partition is a directory.

Objective: To learn about bucket in hive.

Act:

```
CREATE TABLE IF NOT EXISTS STUDENT (rollno INT,name STRING,grade FLOAT)
```

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

```
LOAD DATA LOCAL INPATH '/root/hivedemos/student.tsv' INTO TABLE STUDENT;
```

Set below property to enable bucketing.

```
set hive.enforce.bucketing=true;
```

//To create a bucketed table having 3 buckets

```
CREATE TABLE IF NOT EXISTS STUDENT_BUCKET (rollno INT,name STRING,grade FLOAT)
```

```
CLUSTERED BY (grade) into 3 buckets;
```

// Load data to bucketed table

```
FROM STUDENT
```

```
INSERT OVER WRITE TABLE STUDENT_BUCKET
```

```
SELECT rollno,name,grade;
```

//To display content of first bucket

```
SELECT DISTINCT GRADE FROM STUDENT_BUCKET
```

```
TABLE SAMPLE(BUCKET 1 OUT OF 3 ON GRADE);
```

Objective: Querying the view "STUDENT_VIEW".

Act:

```
SELECT * FROM STUDENT_VIEW LIMIT 4;
```

Objective: To drop the view "STUDENT_VIEW".

Act:

```
DROP VIEW STUDENT_VIEW;
```

Sub-Query

In Hive, sub-queries are supported only in the FROM clause (Hive 0.12). You need to specify name for sub-query because every table in a FROM clause has a name. The columns in the sub-query select list should have unique names. The columns in the sub-query select list are available to the outer query just like columns of a table.

Objective: Write a sub-query to count occurrence of similar words in the file.

Act:

```
CREATE TABLE docs (line STRING);
LOAD DATA LOCAL INPATH '/root/hivedemos/lines.txt' OVERWRITE INTO TABLE
docs;
CREATE TABLE word_count AS
SELECT word, count(1) AS count FROM
(SELECT explode (split (line, ' ')) AS word FROM docs) w
GROUP BY word
ORDER BY word;
SELECT * FROM word_count;
```

Joins: Joins in hive is similar to SQL Joins

Objective: To create JOIN between Student and Department tables where we use RollNo from both the tables as the join key.

Act:

```
CREATE TABLE IF NOT EXISTS STUDENT(rollno INT,narne STRING,gpa FLOAT)
ROW FORMAT DELIMITED FIELDS TERMINATED BY'\t';
LOAD DATA LOCAL INPATH '/root/hivedemos/student.tsv' OVERWRITE INTO TABLE
STUDENT;
CREATE TABLE IF NOT EXISTS DEPARTMENT(rollno INT,deptno int,narne STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY'\t';
```

```
LOAD DATA LOCAL INPATH '/root/hivedemos/department.tsv' OVERWRITE INTO
TABLE DEPARTMENT;
SELECT a.rollno, a.narne, a.gpa, b.deptno FROM STUDENT a JOIN DEPARTMENT b ON
a.rollno = b.rollno;
```

Aggregation

Hive supports aggregate functions like avg, count, etc.

Objective: To write the average and count aggregation function.

Act:

```
SELECT avg(gpa) FROM STUDENT;
SELECT count(*) FROM STUDENT;
```

Group By and Having

Data in a column or columns can be grouped on the basis of values contained therein by using "Group By". "Having" clause is used to filter our groups NOT meeting the specified condition.

Objective: To write group by and having function.

Act:

```
SELECT rollno, name,gpa FROM STUDENT GROUP BY rollno,name,gpa HAVING gpa >
4.0;
```

4.7 RCFILE IMPLEMENTATION

RCFile (Record Columnar File) is a data placement structure that determines how to store relational tables on computer clusters.

Objective: To work with RC file implementation

```
CREATE TABLE STUDENT_RC( collno int, name string,gpa float) STORED AS RCFILE;  
INSERT OVERWRITE table STUDENT_RC SELECT * FROM STUDENT;  
SELECT SUM(gpa) FROM STUDENT_RC;
```

SERDE

SerDe stands for Serializer / Deserializer

1. Contains the logic to convert unstructured data into records.
2. Implemented using Java.
3. Serializers are used at the time of writing.
4. Deserializers

Deserializer interface takes a binary representation or string of a record, converts it into a java object that Hive can then manipulate. Serializer takes a java object that Hive has been working with and translates it into something that Hive can write to HDFS.

Objective: To manipulate the XML data.

Input:

```
<employee> <empid> 1001</empid> <name> John</name> <designation> Team  
Lead</designation>  
</employee>  
<employee> <empid> 1002</empid> <name>Smith</name>  
<designation>Analyst</designation>  
</employee>
```

Act:

```
CREATETABLE XMLSAMPLE(xmldata string);  
LOAD DATA LOCAL INPATH '/root/hivedemos/input.xml' INTO TABLE XMLSAMPLE;  
CREATE TABLE xpath_table AS  
SELECT xpath_int(xmldata,' employee/empid'),  
xpath_string(xmldata, 'employee/name'),  
xpath_string(xmldata,' employee/designation')  
FROM xmlsample;  
SELECT* FROM xpath_table;
```

User Defined Functions (UDF)

In Hive, you can use custom functions by defining the User-Defined Function (UDF)

Objective: Write a Hive function to convert the values of a field to uppercase,

Act:

```
package com.example.hive. udf;  
import org.apache. hadoop.hive.ql.exec.Description;  
import org.apache.hadoop.hive.ql.exec.UDF;  
@Description(  
name= "SimpleUDFExample")  
public final class MyLowerCase extends UDF {  
public String evaluate(final String word) {  
return word.toLowerCase();  
}  
}
```

Short Answer Questions

1. List out any two phases of map task.
2. List out any two phases of reduce task.
3. Which node acts as the Slave and is responsible for executing a Task assigned to it?
4. Which function is responsible for consolidating the results produced by each of the Map() functions/tasks?
5. Which maps input key/value pairs to a set of intermediate key/value pairs?
6. On what basis number of map is usually driven?
7. Which is the default partitioner for partitioning key space?
8. Which is the intermediate phase between map phase and reduce phase?
9. Which command sets the value of a particular configuration variable (key)?
10. Which operator executes a shell command from the Hive shell?
11. Which shell utility can be used to run Hive queries in either interactive or batch mode?
12. What is the command used in Hive for logging?
13. Mention the two benefits of compression in MapReduce programming.
14. Which can be used as warehousing tool in Hadoop?
15. Who created Hive initially?
16. Which is used for storage purpose in Hive?
17. Where metadata is stored in Hive?
18. Which is used by the Hive for execution purpose?
19. In Hive how tables are stored?
20. In Hive how partition tables are stored?
21. In hive which is similar to partitions but uses hash function to segregate data?
22. Which is used to submit Hive Jobs from a remote client?
23. Where Hive table definitions and mappings to the data are stored?
24. Which metastore is mainly used for unit tests?
25. What are the two major classifications of Hive Data types?

Long Answer Questions

1. Explain MapReduce programming with its different phases
2. Explain Partitioner phase of MapReduce with an example.
3. Explain searching in MapReduce with an example.
4. Write a short note on Hive features and Hive integration and workflow.
5. Explain Hive Architecture.
6. Explain Hive Data types.
7. Write a short note on Hive File format.
8. Explain any three DDL command of Hive with syntax and example.
9. Explain partitioning and Bucketing in Hive with an example.
10. Write a short note on RC file implementation and SERDE.

Answers for Short Answer Questions

1. Record Reader , Mapper/Partitioner
2. Shuffle, Reducer, Sort, Output Format

3. Task Tracker 4. Reduce 5. Mapper 6. Total size of input 7. HashPartitioner
8. Partitioner 9. set <key>=<value> 10. ! 11. \$HIVE_HOME/bin/hive
12. log4j
13. Reduces the space to store files, Speed up data transfer across network
14. Hive 15. Facebook 16. HDFS 17. In RDBMS 18. MapReduce 19. As Folders
20. As sub-directory 21. Buckets 22. Hive Server 23. Metastore
24. Embedded Metastore 25. Primitive data types and Collection data types

UNIT-V

INTRODUCTION TO PIG

5.1 What is Pig?

Apache Pig is a platform for data analysis. It is an alternative to Map Reduce Programming. Pig was developed as a research project at Yahoo.

Key Features of PIG

1. It provides an engine for executing data flows (how your data should flow). Pig processes data in parallel on the Hadoop cluster.
2. It provides a language called "Pig Latin" to express data flows.
3. Pig Latin contains operators for many of the traditional data operations such as join, filter, sort, etc.
4. It allows users to develop their own functions (User Defined Functions) for reading, processing, and writing data.

The Anatomy of Pig

The main components of Pig are as follows:

1. Data flow language (Pig Latin).
2. Interactive shell where you can type Pig Latin statements (Grunt).
3. Pig interpreter and execution engine.

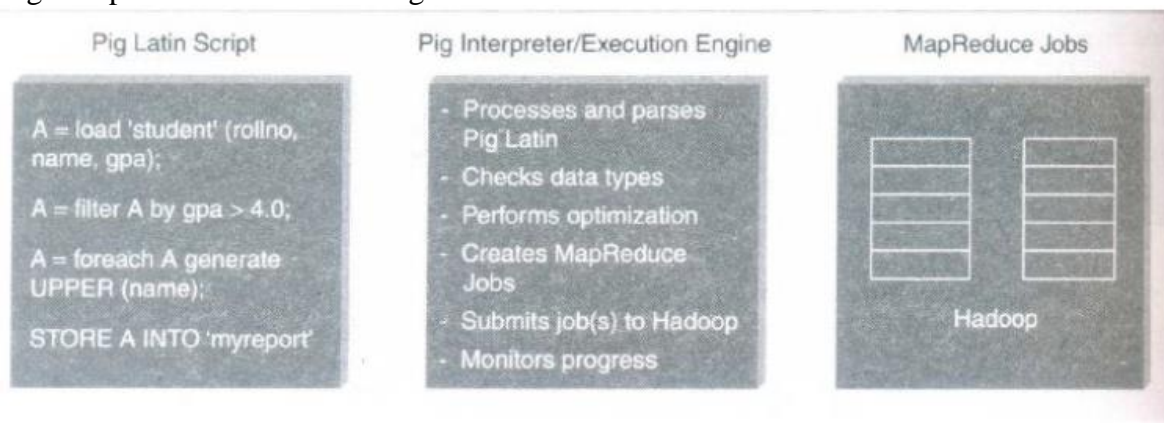


Figure 5.1: Anatomy of Pig

PIG on Hadoop

Pig runs on Hadoop. Pig uses both Hadoop Distributed File System and MapReduce Programming. By default, Pig reads input files from HDFS. Pig stores the intermediate data (data produced by MapReduce jobs) and the output in HDFS. However, Pig can also read input from and place output to other sources.

Pig supports the following:

1. HDFS commands.
2. UNIX shell commands.
3. Relational operators.
4. Positional parameters.
5. Common mathematical functions.
6. Custom functions.
7. Complex data structures.

5.2 Pig Philosophy: Figure 5.2 describes the Pig Philosophy

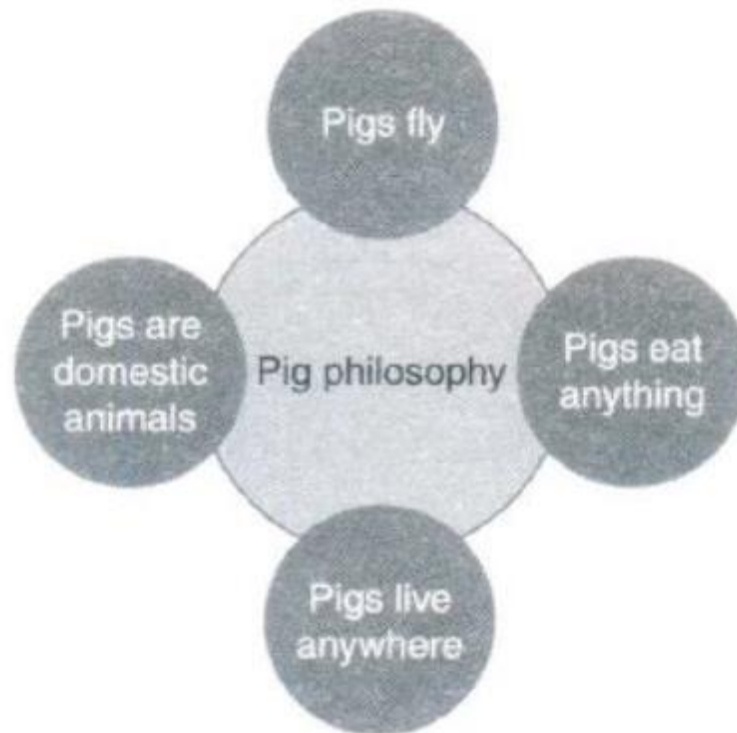


Figure 5.2: Pig philosophy

Pigs Eat Anything: Pig can process different kinds of data such as structured and unstructured data.

Pigs Live Anywhere: Pig not only processes files in HDFS, it also processes files in other sources such as files in the local file system.

Pigs are Domestic Animals: Pig allows you to develop user-defined functions and the same can be included in the script for complex operations.

Pigs Fly: Pig processes data quickly.

5.3 Pig Latin Overview

Pig Latin Statements:

1. Pig Latin statements are basic constructs to process data using Pig.
2. Pig Latin statement is an operator.
3. An operator in Pig Latin takes a relation as input and yields another relation as output.
4. Pig Latin statements include schemas and expressions to process data.
5. Pig Latin statements should end with a semi-colon.

Pig Latin Statements are generally ordered as follows:

1. LOAD statement that reads data from the file system.
2. Series of statements to perform transformations.
3. DUMP or STORE to display/store result.

The following is a simple Pig Latin script to load, filter, and store "student" data.

```
A = load 'student' (rollno, name, gpa);  
A = filter A by gpa > 4.0;  
A = foreach A generate UPPER (name);  
STORE A INTO 'myreport'
```

Note: In the above example A is a relation and NOT a Variable.

Pig Latin Keywords: Keywords are reserved. It cannot be used to name identifiers.

Pig Latin Identifiers

The following are the rules for constructing identifiers in Pig Latin.

1. Identifiers are names assigned to fields or other data structures.
2. It should begin with a letter and should be followed only by letters, numbers, and underscores.

Example:

Valid identifier - Y A1 A1_2014 Sample

Invalid identifier - 5 Sales\$ Sales% _Sales

Pig Latin Comments: In Pig Latin two types of comments are supported:

1. Single line comments that begin with " ____".
2. Multiline comments that begin with "/*" and end with */".

Case Sensitivity in Pig Latin:

1. Keywords are *not* case sensitive such as LOAD, STORE, GROUP, FOREACH, DUMP, etc.
2. Relations and paths are case-sensitive.
3. Function names are case sensitive such as PigStorage, COUNT.

Operators in Pig Latin

Table 5.1: Operators in Pig Latin

Arithmetic	Comparison	Null	Boolean
+	= =	IS NULL	AND
-	! =	IS NOT NULL	OR
*	<		NOT
/	>		
%	< =		
	> =		

DATA TYPES IN PIG

Table 5.2 describes simple data types supported in Pig. In Pig, fields of unspecified types are considered as array of bytes which is known as byte array.
Null: In Pig Latin, NULL denotes a value that is unknown or is non-existent.

Table 5.2: Simple Data Types supported in Pig

Name	Description
Int	Whole numbers
Long	Large whole numbers
Float	Decimals
Double	Very precise decimals
Chararray	Text string
Bytearray	Raw bytes
Datetime	Datetime
Boolean	true and false

Complex Data Type: Table 5.3 describes complex data types in Pig.

Name	Description
Tuple	An ordered set of fields. Example: (2,3)
Bag	A collection of tuples. Example{ (2, 3), (7, 5)}
Map	key, value pair (open # Apache)

Executing Pig

You can run Pig in two ways:

- 1. Interactive Mode.
- 2. Batch Mode.

Interactive Mode: You can run pig in interactive mode by invoking grunt shell. Type pig to get grunt shell.

Once you get the grunt prompt, you can type the Pig Latin statement as shown below. `grunt> A = load '/pigdemo/student.tsv' as (rollno, name, gpa);`

`grunt> DUMP A;`

Here, the path refers to HDFS path and DUMP displays the result on the console as shown below. (1001, John, 3.0)

(1002, Jack, 4.0)

(1003, Smith, 4.5)

(1004, Scott, 4.2)

(1005, Joshi, 3.5)

`grunt>`

Batch Mode

You need to create "Pig Script" to run pig in batch mode. Write Pig Latin statements in a file and save it with .pig extension.

EXECUTION MODES OF PIG

You can execute pig in two modes

- 1. Local mode
- 2. MapReduce mode

Local Mode: To run Pig in local mode, you need to have your files in the local file system. Syntax: `-x local filename`

MapReduce Mode: To Run Pig in MapReduce, you need to have access to a Hadoop Cluster to read /write file. This is the default mode of Pig.

Syntax: `pig filename`

HDFS COMMANDS: You can work with all HDFS commands in Grunt shell. For example, you can create a directory as shown below.

`grunt> fs -mkdir`

/piglatindemos; grunt>

The sections have been designed as follows:

Objective: What is it that we are trying to achieve here?

Input: What is the input that has been given to us to act upon?

Act: The actual statement/command to accomplish the task at hand.

Outcome: The result/output as a consequence of executing the statement.

RELATIONAL OPERATORS

Filter: The FILTER operator is used to select tuples from a relation based on specified conditions. **Objective:** Find the tuples of those student where the GPA is greater than 4.0.

Input:

Student (

rollno:int,name:chararray,gpa:float) **Act:**

A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float); B = filter A by gpa > 4.0;

DUMP B;

Output:

```
(1003,Smith,4.5)
(1004,Scott,4.2)
[root@volga1nx010 pigdemos]#
```

FOREACH: Use *FOREACH* when you want to do data transformation based on columns of data. **Objective:** Display the name of all students in uppercase.

Input:

Student (roll no: in t, name:chararray,gpa:

float) **Act:**

A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float); B = foreach A generate UPPER (name);

DUMP B;

Output

```
(JOHN)
(JACK)
(SMITH)
(SCOTT)
(JOSHI)
[root@volga1nx010 pigdemos]#
```

GROUP: GROUP operator is used to group data.

Objective: Group tuples of students based on their GPA.

Input:

Student(rollno: int, name:chararray,gpa:float)

let:

B=GROUP A BY gpa;

DUMP B;

DISTINCT

DISTINCT operator is used to remove duplicate tuples. In Pig, DISTINCT operator works on the entire and NOT on individual fields.

Objective: To remove duplicate tuples of students.

Input:

Student (rollno: int,name:chararray,gpa: float)

Input:

```
1001  John  3.0
1002  Jack  4.0
1003  Smith 4.5
1004  Scott 4.2
1005  Joshi 3.5
1006  Alex  4.5
1007  David 4.2
1008  James 4.0
1001  John  3.0
1005  Joshi 3.5
```

Act:

A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);

B= DISTINCT A;

DUMP B;

Output:

```
(1001,John,3.0)
(1002,Jack,4.0)
(1003,Smith,4.5)
(1004,Scott,4.2)
(1005,Joshi,3.5)
(1006,Alex,4.5)
(1007,David,4.2)
(1008,James,4.0)
[root@volga1nx010 pigdemos]#
```

LIMIT

LIMIT operator is used to limit the number of output tuples.

Objective: Display the first 3 tuples from the "student" relation.

Input: Student (rollno: int, name:chararray,gpa:float)

Act:

A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);

B = LIMIT A 3;

DUMP B;

ORDER BY: Order by is used to sort relation based on the specific value.

Objective: Display the names of the students in Ascending Order.

Input:

Student (rollno: int, name:chararray,gpa:float)

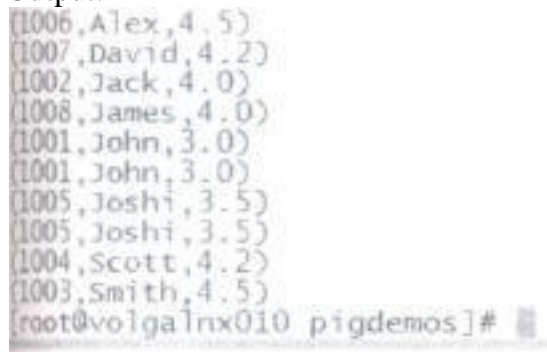
Act:

A= load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);

B = ORDER A BY name;

DUMP B;

Output:



```
(1006,Alex,4.5)
(1007,David,4.2)
(1002,Jack,4.0)
(1008,James,4.0)
(1001,John,3.0)
(1001,John,3.0)
(1005,Joshi,3.5)
(1005,Joshi,3.5)
(1004,Scott,4.2)
(1003,Smith,4.5)
root@volga1nx010 pigdemos]#
```

JOIN

Join is used to two or more relations based on values in the common filed. It always performs inner join.

Objective: To join two relations namely, "student" and "department" based on the values contained in the "rollno" column.

Input:

Student (rollno: int,name:chararray,gpa:float)

Department(rollno: int,deptno: int,deptname: chararray)

Act:

A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);

B = load '/pigdemo/department.tsv' as (rollno:int, deptno:int,deptname:chararray);

C = JOIN A BY rollno, B BY rollno;

DUMP C;

DUMP B;

Output:

(1001, John, 3.0 , 1001, 101, B. E

(1001, John, 3.0, 1001, 101, B. E.)
(1002, Jack, 4.0, 1002, 102, B. Tech)
(1001, Smith, 4.5, 1003, 103, M. Tech)
(1004, Scott, 4.2, 1004, 104, MCA)
(1005, Joshi, 3.5, 1005, 105, MBA)
(1005, Joshi, 3.5, 1005, 105, MBA)
(1006, Alex, 4.5, 1006, 101, B. E)
(1007, David, 4.2, 1007, 104, MCA)
(1008, James, 4.0, 1008, 102, B. Tech)
[root@volglx010 pigdemos]#

UNION: Union used to merge the contents of two relations

Objective: To merge the contents of two relations "student" and "department".

Input:

Student (rollno: int, name:chararray,gpa:float)

Department (rollno: int,deptno: int,deptname: chararray)

Act:

A = load '/pigdemo/student.tsv' as (rollno, name, gp);

B = load '/pigdemo/department.tsv' as (rollno, deptno,deptname);

C = UNION A, B;

STORE C INTO '/pigdemo/uniondemo';

DUMP B;

Output:

"Store" is used to save the output to a specified path. The output is stored in two files: part-m-00000 contains "student" content and part-m-00001 contains "department" content.

SPLIT: It is used to partition two or more relations

Objective: To partition a relation based on the GPAs acquired by the students.

GPA = 4.0, place it into relation X.

GPA is < 4.0, place it into relation Y.

Input:

Student (rollno: int, name:chararray,gpa:float)

Act:

A= load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);

SPLIT A INTO X IF gpa==4.0, Y IF gpa <=4.0;

DUMP X;

SAMPLE: It is used to select random sample of data based on the specified sample size.

Objective:To depict the use of SAMPLE.

Input:

Student (rollno: int,name:chararray,gpa: float)

Act:

A= load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);

B = SAMPLE A 0.01;

DUMP B;

Eval Function

AVG: AVG is used to compute the average of numeric values in a single column bag.

Objective: To calculate the average marks for each student.

Input:

Student (studname:chararray,marks:int)

Act:

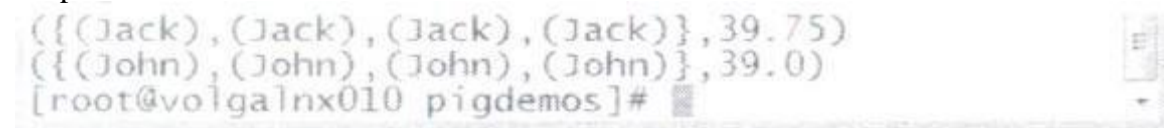
```
A= load '/pigdemo/student.csv' USING PigStorage(',') as (studname:chararray,marks:int);
```

```
B = GROUP A BY studname;
```

```
C = FOREACH B GENERATE Astudname, AVG(Amarks);
```

```
DUMPC;
```

Output:



```
(({Jack}), (Jack), (Jack), (Jack)), 39.75)
(({John}), (John), (John), (John)), 39.0)
[root@volga1nx010 pigdemos]#
```

You need to use pigStorage function if you wish to manipulate files other than .tsv

Max: Max is used to compute the maximum marks for each student.

Objective: To calculate the maximum marks for each student.

Input:

Student (studname:chararray,marks:int)

Act:

```
A= load '/pigdemo/student.csv' USING PigStorage(',') as (studname:chararray, marks:int);
```

```
B = GROUP A BY studname;
```

```
C = FOREACH B GENERATE A.studname, MAX(A.marks);
```

```
DUMP C;
```

Output:

```
(({Jack}), (Jack), (Jack), (Jack)), 46)
```

```
(({John}), (John), (John), (John)), 45)
```

Similarly you can try the MIN and the SUM function as well.

COUNT: It is used to count the number of elements in a bag.

Objective: To count the number of tuples in a bag.

Input:

Student (studname:chararray,marks:int)

Act:

```
A=load '/pigdemo/student.csv' USING PigStorage(',') as (studname:chararray, marks:int);
```

```
B = GROUP A BY studname;
```

```
C = FOREACH B GENERATE A. studname,COUNT(A);
```

```
DUMP C;
```

The default file format of Pig is .tsv file. Use PigStorage() to to manipulate files other than .tsv file.