

Module 4

Chapter 9 – Introduction to Hive

Prepared By:

Mr. Rajesh Nayak

Department of Artificial Intelligence and Data Science

- **9.1 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 data.
- Three main tasks performed by Hive are:
 - Summarization.
 - Querying.
 - Analysis.
- Hive makes use of following:
 - HDFS for storage.
 - MapReduce for execution.
 - Stores metadata in an RDBMS.
- Hive provides HQL or HiveQL which is similar to SQL. Hive compiles SQL queries into MapReduce jobs and then runs the job in the Hadoop cluster. It supports OLAP.

• 9.1.1 History of Hive and Recent Releases of Hive



Figure 9.2 History of Hive.

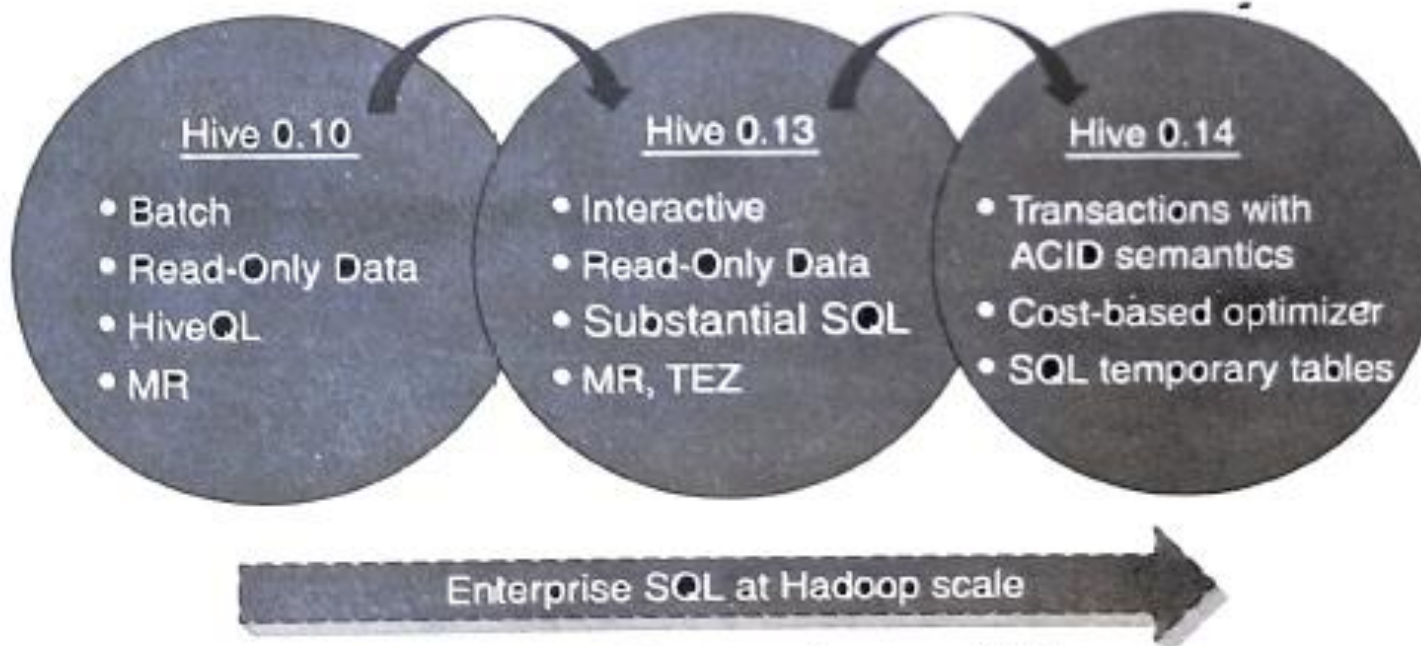


Figure 9.3 Recent releases of Hive.

• 9.1.2 Hive Features

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

• 9.1.3 Hive Integration and Work Flow

Figure 9.4 depicts the flow of log file analysis.

Explanation of the workflow. Hourly Log Data can be stored directly into HDFS and then data cleaning is performed on the log file. Finally, Hive table(s) can be created to query the log file.

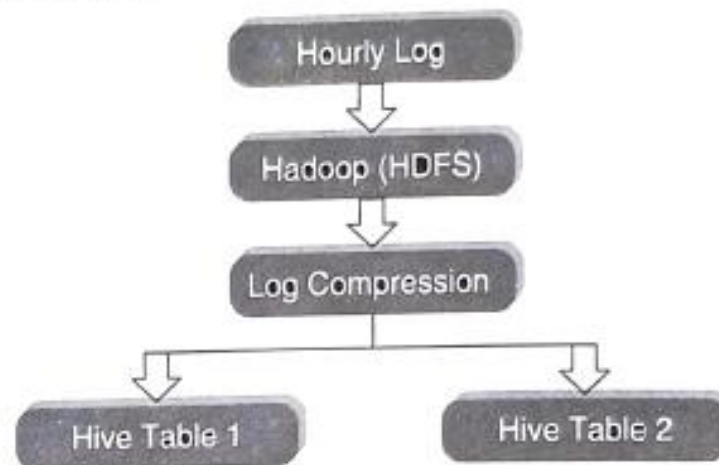


Figure 9.4 Flow of log analysis file.

• 9.1.4 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 specific 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.
- Bucketing is used when cardinality of the field is high. When cardinality is less, partitioning works well.
 - Partitioning can be done on multiple fields whereas bucketing can be done on only one field.

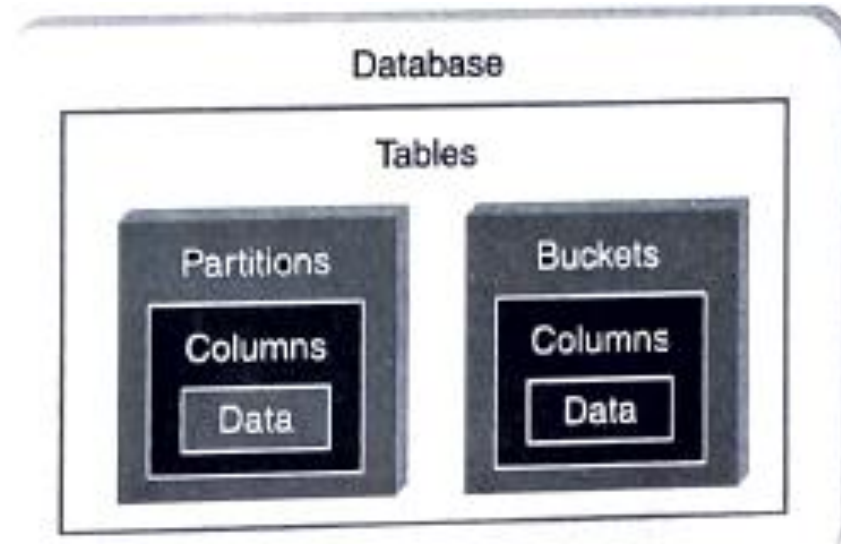


Figure 9.5 Data units as arranged in a Hive.

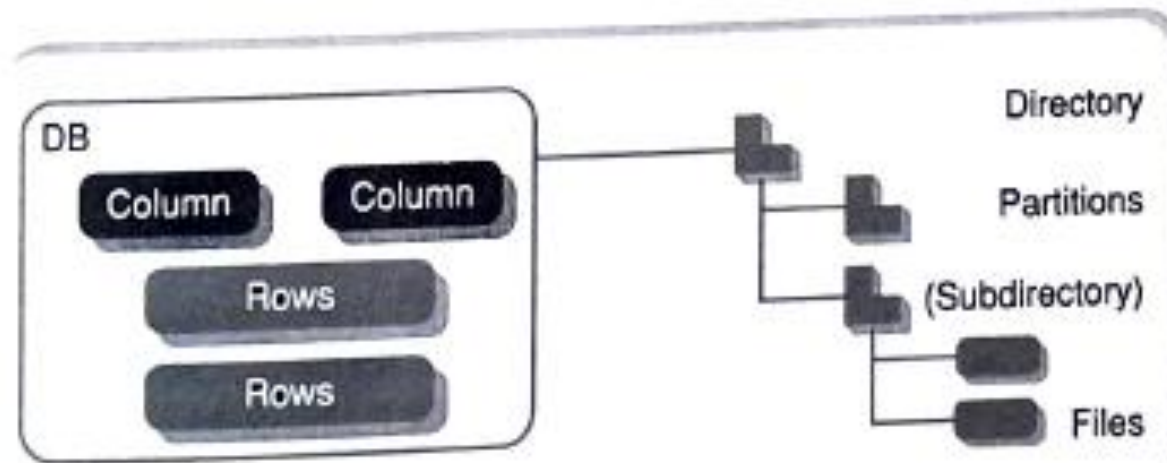
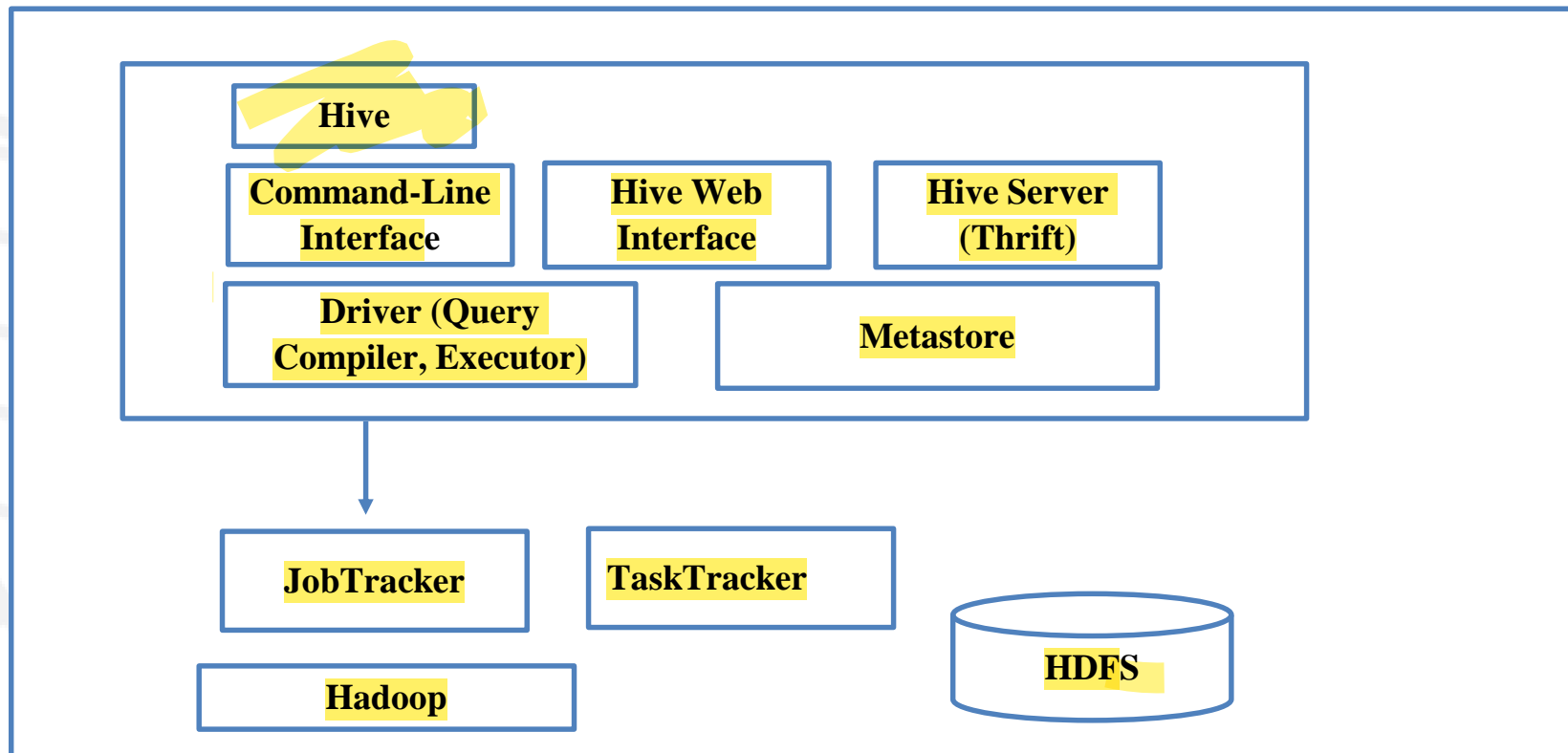


Figure 9.6 Semblance of Hive structure with database.

- 9.2. Hive Architecture



• 9.2. Hive Architecture

- Various parts of Hive architecture 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 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. It consists of metastore services and database

• 9.2. Hive Architecture

- Metadata which is stored in the metastore includes IDs of Tables, IDs of Indexes, the time of creation on table, the input format used for table, output format used for table, etc. Metastore is updated whenever a table is created or deleted from Hive.
- There are 3 types of metastores:

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 run embedded in the main Hive Server process. Figure 9.3 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 9.9 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 9.10. This way the database can be fire-walled from the Hive user and also database credentials are completely isolated from the users of Hive.

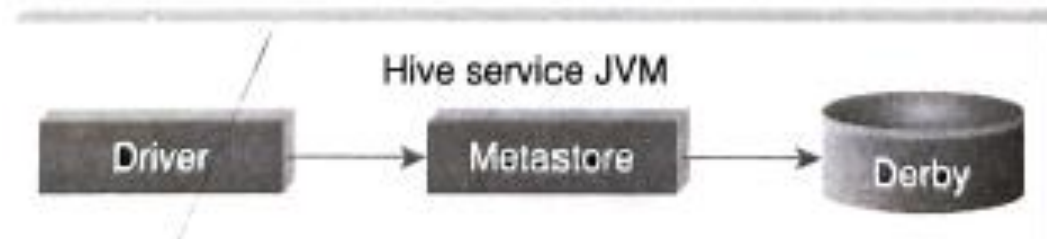


Figure 9.8 Embedded Metastore.

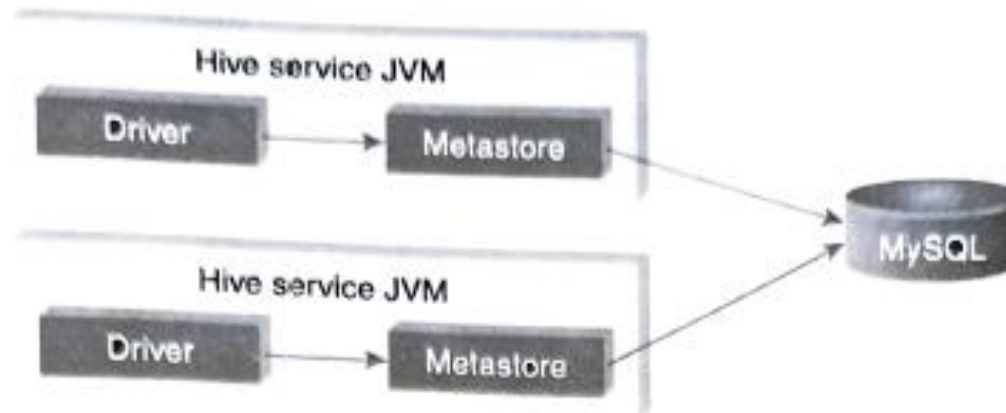


Figure 9.9 Local Metastore.

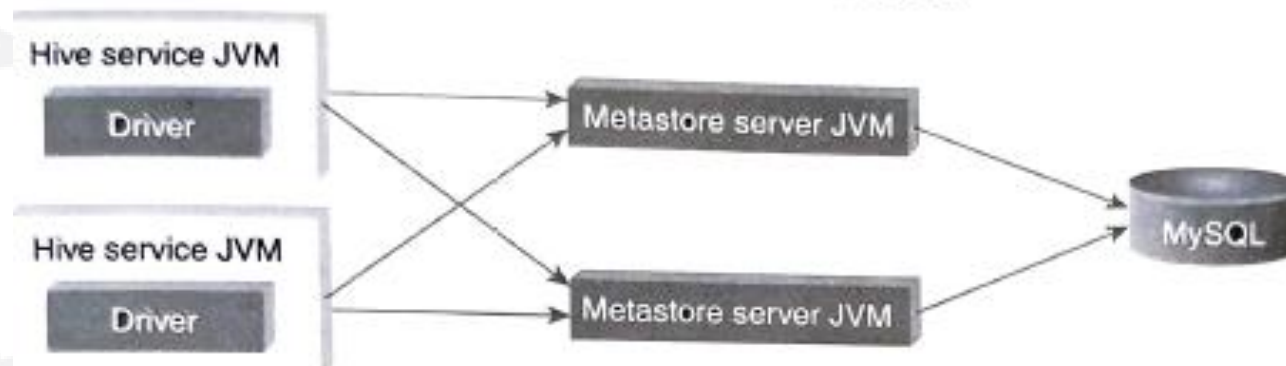


Figure 9.10 Remote Metastore.

- **9.3. Hive Data Types**
- **9.3.1 Primitive Data 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

- **9.3. Hive Data Types**
- **9.3.2 Collection Data Types**

Collection Data Types

STRUCT	Similar to 'C' 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')

• 9.4. Hive File Format

9.4.1 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.

9.4.2 Sequential File

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

• 9.4. Hive File Format

9.4.3 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 in Table 9.1.

Instead of only partitioning the table horizontally like the row-oriented DBMS (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. Depicted in Table 9.2, Table 9.1 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 in Table 9.3.

Table 9.1 A table with four columns

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

• 9.4. Hive File Format

Table 9.2 Table with two row groups

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				

Table 9.3 Table in RCFile Format

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;

Chapter 10 – Introduction to Pig

- **10.1. What is Pig**
 - Apache Pig is a platform for data analysis. It is an alternative to MapReduce programming.
 - Pig was developed as a research project at *Yahoo*.
-
- **10.1.1 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.

- **10.2. 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.

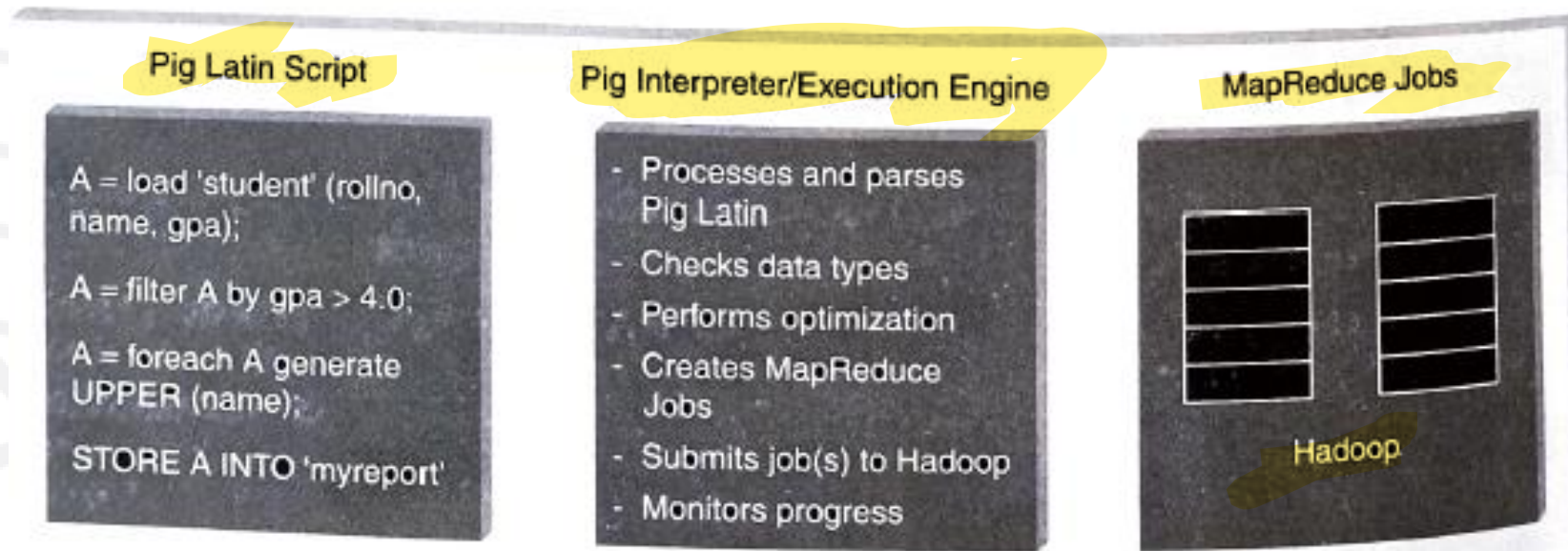


Figure 10.1 The anatomy of Fig.

- **10.3. Pig on Hadoop**
- Pig runs on Hadoop. It uses both HDFS 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 following:
 1. HDFS commands.
 2. UNIX shell commands.
 3. Relational operators.
 4. Positional operators.
 5. Common mathematical functions.
 6. Custom functions.
 7. Complex data structures.

• 10.4. Pig Philosophy

Figure 10.2 describes the Pig philosophy.

1. **Pigs Eat Anything:** Pig can process different kinds of data such as structured and unstructured data.
2. **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.
3. **Pigs are Domestic Animals:** Pig allows you to develop user-defined functions and the same can be included in the script for complex operations.
4. **Pigs Fly:** Pig processes data quickly.

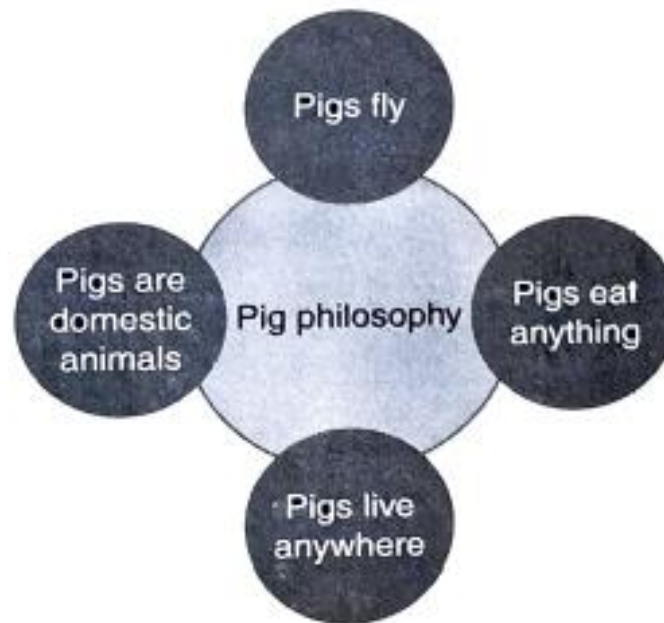


Figure 10.2 Pig philosophy.

- **10.5. Use Case for Pig : ETL Processing**
- Pig is widely used for ETL (Extract, Transform, and Load).
- It can extract data from different sources such as ERP, Accounting, Flat Files, etc.
- Pig then makes use of various operators to perform transformation on the data and subsequently loads it into the data warehouse.

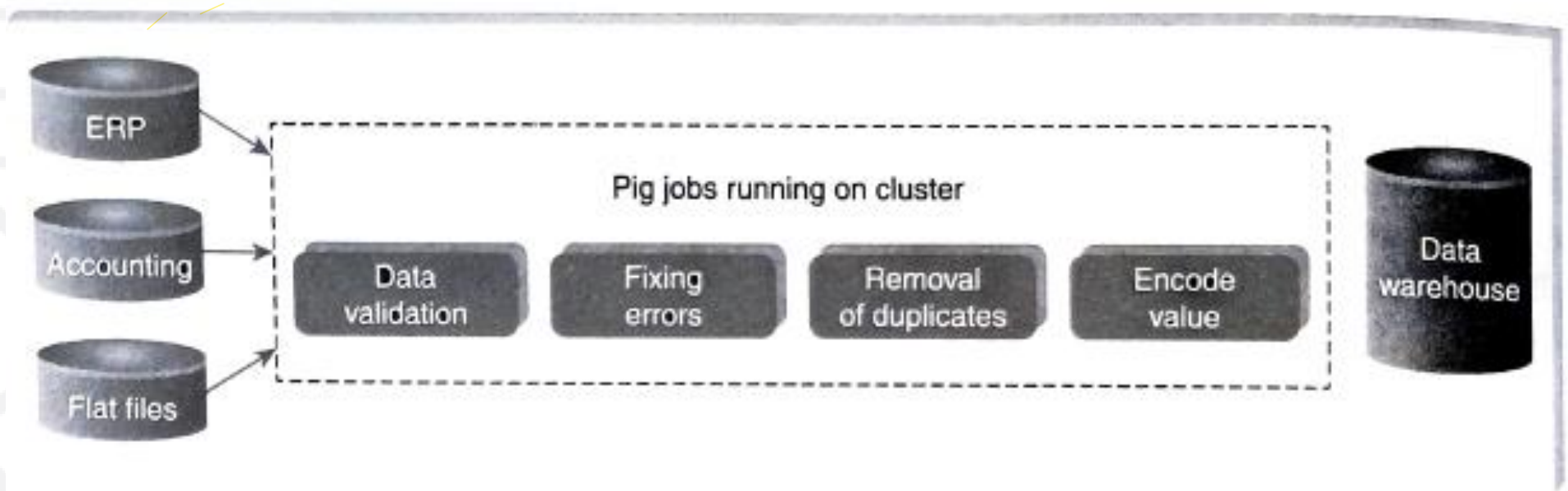


Figure 10.3 Pig: ETL Processing.

- **10.6. Pig Latin Overview**

- **10.6.1 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'
```


- **10.6. Pig Latin Overview**

- **10.6.2 Pig Latin: Keywords**

- Keywords are reserved. It cannot be used to name things.

- **10.6.3 Pig Latin: Identifiers**

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.

Valid Identifier	Y	A1	A1_2014	Sample
Invalid Identifier	5	Sales\$	Sales%	_Sales

- **10.6.4 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 */".

- **10.6. Pig Latin Overview**

- **10.6.5 Pig Latin: Case Sensitivity**

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

- **10.6.6 Operators in Pig Latin**

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

- **10.7 Data Types in Pig**
- **10.7.1 Simple Data Types**

Name	Description
int	Whole numbers
long	Large whole numbers
float	Decimals
double	Very precise decimals
chararray	Text strings
bytearray	Raw bytes
datetime	Datetime
boolean	true or false

- **10.7.2 Complex Data Types**

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)

• 9.5 Hive Query Language (HQL/HiveQL)

Hive 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. Support 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.

9.5.1 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

9.5.2 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: Hive 0.14 supports update, delete, and transaction operations.

9.5.4 Database

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

```
hive> CREATE DATABASE IF NOT EXISTS STUDENTS COMMENT 'STUDENT Details' WITH DBPROPERTIES ('creator' = 'JOHN');  
OK  
Time taken: 0.536 seconds  
hive> □
```

Explanation of the syntax:

IF NOT EXIST: It is an optional clause. The create database statement with “IF Not EXISTS” clause creates a database if it does not exist. However, if the database already exists then it will notify the user that a database with the same name already exists and will not show any error message.

COMMENT: This is to provide short description about the database.

WITH DBPROPERTIES: It is an optional clause. It is used to specify any properties of database in the form of (key, value) separated pairs. In the above example, “Creator” is the “Key” and “JOHN” is the value.

We can use “SCHEMA” in place of “DATABASE” in this command.

Objective: To display a list of all databases.

Act:

SHOW DATABASES;

Outcome:

```
hive> SHOW DATABASES;  
OK
```

```
students  
Time taken: 0.082 seconds, Fetched: 22 row(s)  
hive> █
```

By default, **SHOW DATABASES** lists all the databases available in the metastore. We can use "**SCHEMAS**" in place of "**DATABASES**" in this command. The command has an optional "Like" clause. It can be used to filter the database names using regular expressions such as "*", "?", etc.

SHOW DATABASES LIKE "Stu*"

SHOW DATABASES like "Stud???s"

Objective: To describe a database.

Act:

DESCRIBE DATABASE STUDENTS;

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

Outcome:

```
hive> DESCRIBE DATABASE STUDENTS;  
OK  
students      STUDENT Details hdfs://volga1nx010.ad.infosys.com:9000/user/hive/warehouse/studen  
ts.db root     USER  
Time taken: 0.03 seconds, Fetched: 1 row(s)  
hive> █
```

Objective: To describe the extended database.

Act:

DESCRIBE DATABASE EXTENDED STUDENTS;

Note: Shows DB properties also.

Objective: To make the database as current working database.

Act:

USE STUDENTS;

Outcome:

```
hive> USE STUDENTS;  
OK  
Time taken: 0.02 seconds  
hive> █
```

Objective: To alter the database properties.

Act:

ALTER DATABASE STUDENTS SET DBPROPERTIES ('edited-by' = 'JAMES');

Note: We can use the “ALTER DATABASE” command to

- Assign any new (key, value) pairs into DBPROPERTIES.
- Set owner user or role to the Database.

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

Outcome:

```
hive> ALTER DATABASE STUDENTS SET DBPROPERTIES ('edited-by' = 'JAMES');  
OK  
Time taken: 0.086 seconds  
hive> █
```

In the above example, the ALTER DATABASE command is used to assign new ('edited-by' = 'JAMES') pair into DBPROPERTIES. This can be verified by using the 'describe *extended*'.

Hive> DESCRIBE DATABASE Student EXTENDED

Objective: To drop database.

Act:

DROP DATABASE STUDENTS;

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

Contents of directory /usr/hive/warehouse

ls -l /usr/hive/warehouse

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
students.db	dir				2015-02-24 21:50	rw-r-xr-x	root	supergroup

Now assume that the database “STUDENTS” has 10 tables within it. How do we delete the complete database along with the tables contained therein?

Use the command:

DROP DATABASE STUDENTS CASCADE;

By default the mode is RESTRICT which implies that the database will NOT be dropped if it contains tables.

Note: The complete syntax is as follows:

DROP DATABASE [IF EXISTS] database_name [RESTRICT | CASCADE]

9.5.5 Tables

Hive provides two kinds of table:

1. Internal or Managed Table
2. External Table

9.5.5.1 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.

When you create a table in Hive, by default it is internal or managed table. If one needs to create an external table, one will have to use the keyword "EXTERNAL".

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';
```

Outcome:

```
hive> CREATE TABLE IF NOT EXISTS STUDENT(rollno INT,name STRING,gpa FLOAT) ROW FORMAT DELIMITED F  
IELDS TERMINATED BY '\t';  
OK  
Time taken: 0.355 seconds  
hive>
```

Objective: To describe the “STUDENT” table.

Act:

DESCRIBE STUDENT;

Outcome:

```
hive> DESCRIBE STUDENT;
OK
rollno          int
name            string
gpa             float
Time taken: 0.163 seconds, Fetched: 3 row(s)
hive>
```

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

Contents of directory `/user/hive/warehouse/students.db`:

Go to `/user/hive/warehouse/students.db` go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permissions	Owner	Group
student	dir				2015-02-24 22:03	rw-r-xr-x	root	supergroup

Go back to DFS home

Local logs

Log directory

Hadoop: 2015

9.5.5.2 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';
```

Outcome:

```
hive> CREATE EXTERNAL TABLE IF NOT EXISTS EXT_STUDENT(rollno INT,name STRING,gpa FLOAT) ROW FORM  
T DELIMITED FIELDS TERMINATED BY '\t' LOCATION '/STUDENT_INFO';  
OK  
Time taken: 0.123 seconds  
hive>
```

Note: Hive creates the external table in the specified location.

9.5.5.3 Loading Data into Table from File

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. In this case, the file is copied. To load the data from HDFS, remove local key word from the statement. In this case, the file is moved from the original location.

Outcome:

```
hive> LOAD DATA LOCAL INPATH '/root/hivedemos/student.tsv' OVERWRITE INTO TABLE EXT_STUDENT;  
Loading data to table students.ext_student  
Table students.ext_student stats: [numFiles=0, numRows=0, totalSize=0, rawDataSize=0]  
OK  
Time taken: 5.034 seconds  
hive>
```

9.5.5.4 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 KEYS TERMINATED BY '!';
```

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

Outcome:

```
hive> CREATE TABLE STUDENT_INFO (rollno INT,name String, sub ARRAY<STRING>,marks MAP<STRING,FLOAT>)  
> ROW FORMAT DELIMITED FIELDS TERMINATED BY ','  
> COLLECTION ITEMS TERMINATED BY ':'  
> MAP KEYS TERMINATED BY '!';
```

```
OK  
Time taken: 0.112 seconds  
hive>
```

```
hive> LOAD DATA LOCAL INPATH '/root/hivedemos/studentinfo.csv' INTO TABLE STUDENT_INFO;  
Loading data to table students.student_info  
Table students.student_info stats: [numFiles=1, totalSize=109]  
OK  
Time taken: 0.397 seconds  
hive>
```




9.5.5.5 Querying Table

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

Act:

```
SELECT * from EXT_STUDENT;
```

Outcome:

```
hive> select * from EXT_STUDENT;
OK
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
1009  John     3.0
1010  Joshi    3.5
Time taken: 0.054 seconds, Fetched: 10 row(s)
hive>
```

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;
```

Outcome:

```
hive> SELECT * from STUDENT_INFO;
OK
001  John    ["Smith","Jones"]  {"Mark1":45,"Mark2":46,"Mark3":43}
1002 Jack    ["Smith","Jones"]  {"Mark1":46,"Mark2":47,"Mark3":42}
Time taken: 0.044 seconds, Fetched: 2 row(s)
```

```
hive> SELECT NAME,SUB FROM STUDENT_INFO;
OK
John    ["Smith","Jones"]
Jack    ["Smith","Jones"]
Time taken: 0.061 seconds, Fetched: 2 row(s)
hive>
```

```
hive> SELECT NAME, MARKS['Mark1'] from STUDENT_INFO;
OK
John    45
Jack    46
Time taken: 0.06 seconds, Fetched: 2 row(s)
hive>
```

```
hive> SELECT NAME,SUB[0] FROM STUDENT_INFO;
OK
John    Smith
Jack    Smith
Time taken: 0.071 seconds, Fetched: 2 row(s)
hive>
```



PICTURE THIS...

"XYZ enterprise" has a wide customer base spread across several states in the US. The data has been fed to the central system. The senior leadership team would like to get a report providing the statewise Sales %.

The IT team has proposed two options to help service this request:

1. Run the query with the where clause of each state name (such as where StateName = "A"). This will mean that the entire dataset is scanned/read through for each and every state. If we have data for 25 states, the dataset is read 25 times (once for each state). Assuming we have 5 million records, it would mean that 5 million records are read 25 times. This can be a major performance bottle neck and will worsen as the dataset grows.
2. The second option stated here can help alleviate this problem. [No Title] team can start off with creating 25 folders (one for each state) and

instruct to have the data pertaining to states place into the folder of the respective states. This arrangement will greatly help at the time of querying the data. To get the Sales % of a particular state, the folder belonging to that state only has to be scanned/read through.

This intelligent way of grouping data during data load is termed as PARTITIONING in Hive.

This brings us to the next question.

If you are looking through the folder for state = "A", is there any possibility of you coming across data for state = "B" or state = "C"? Think through! I am sure your answer will be No! And we know the reason.

A point to note here is that as we create the partitions, there is no need to include the partitioned column along with the other columns of the dataset as this is something that is automatically taken care of "BY PARTITIONED" clause.

9.5.6 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.

We will try to understand partitioning with the help of a simple example.

In our example above, we will refrain from adding the partitioned column along with other columns of the dataset and trust Hive to automatically manage this.

Partition is of two types:

1. **STATIC PARTITION:** It is upon the user to mention the partition (the segregation unit) where the data from the file is to be loaded.
2. **DYNAMIC PARTITION:** The user is required to simply state the column, basis which the partitioning will take place. Hive will then create partitions basis the unique values in the column on which partition is to be carried out.

Points to consider as you create partitions:

1. **STATIC PARTITIONING** implies that the user controls everything from defining the **PARTITION** column to loading data into the various partitioned folders.
 2. As in our example above, if **STATIC** partition is done over the **STATE** column and assume by mistake the data for state "B" is placed inside the partition for state "A", our query for data for state "B" is bound to return zilch records. The reason is obvious. A **Select** fired on **STATIC** partition just takes into consideration the partition name, and does not consider the data held inside the partition.
- DYNAMIC PARTITIONING** means Hive will intelligently get the distinct values for partitioned column and segregate data into respective partitions. There is no manual intervention.

By default, dynamic partitioning is enabled in Hive. Also by default it is strictly implying that one is required to do one level of **STATIC** partitioning before Hive can perform **DYNAMIC** partitioning inside this **STATIC** segregation unit.

In order to go with full dynamic partitioning, we have to set below property to non-strict in Hive.

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

5.6.1 Static Partition

Static partitions comprise columns whose values are known at compile time.

Objective: To create static partition based on "gpa" column.

Act:

```
CREATE TABLE IF NOT EXISTS STATIC_PART_STUDENT (rollno INT, name STRING)
PARTITIONED BY (gpa FLOAT) ROW FORMAT DELIMITED FIELDS TERMINATED
BY '\t';
```

Outcome:

```
hive> CREATE TABLE IF NOT EXISTS STATIC_PART_STUDENT(rollno INT,name STRING) PARTITIONED BY (gpa FLOAT) RO
W FORMAT DELIMITED FIELDS TERMINATED BY '\t';
OK
Time taken: 0.105 seconds
hive> █
```

Objective: Load data into partition table from table.

Act:

```
INSERT OVERWRITE TABLE STATIC_PART_STUDENT PARTITION (gpa =4.0)
SELECT rollno, name from EXT_STUDENT where gpa=4.0;
```

Outcome:

```
hive> INSERT OVERWRITE TABLE STATIC_PART_STUDENT PARTITION (gpa =4.0) SELECT rollno,name from EXT_STUDENT
where gpa=4.0;
Query ID = root_20150224230404_4500d58a-cb21-4912-ba40-788e5cf8f9da
Total jobs = 3
```


Objective: To add one more static partition based on “gpa” column using the “alter” statement.

Act:

```
ALTER TABLE STATIC_PART_STUDENT ADD PARTITION (gpa=3.5);
INSERT OVERWRITE TABLE STATIC_PART_STUDENT PARTITION (gpa =4.0) SELECT
rollno,name from EXT_STUDENT where gpa=4.0;
```

Outcome:

```
hive> ALTER TABLE STATIC_PART_STUDENT ADD PARTITION (gpa=3.5);
OK
Time taken: 0.166 seconds
hive>
```

Contents of directory /user/hive/warehouse/students.db/static_part_student

Goto go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
gpa=3.5	dir				2015-02-24 13:09	rw-r-xr-x	root	supergroup
gpa=4.0	dir				2015-02-24 13:11	rw-r-xr-x	root	supergroup

Go back to DFB home

9.5.6.2 Dynamic Partition

Dynamic partition have columns whose values are known only at Execution Time.

Objective: To create dynamic partition on column date.

Act:

```
CREATE TABLE IF NOT EXISTS DYNAMIC_PART_STUDENT(rollno INT,name STRING)
PARTITIONED BY (gpa FLOAT) ROW FORMAT DELIMITED FIELDS TERMINATED
BY '\t';
```

Outcome:

```
hive> CREATE TABLE IF NOT EXISTS DYNAMIC_PART_STUDENT(rollno INT,name STRING) PARTITIONED BY (gpa FLOAT) R
OW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
OK
Time taken: 0.166 seconds
hive>
```

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;
```

Outcome:

Contents of directory `/usr/hive/warehouse/students.db/dynamic_part_student`

Use `ls` to view contents of the directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
part-1.0	dir				2015-02-24 23:16	rw-rw-r--	root	supergroup
part-1.1	dir				2015-02-24 23:16	rw-rw-r--	root	supergroup
part-1.2	dir				2015-02-24 23:16	rw-rw-r--	root	supergroup
part-1.3	dir				2015-02-24 23:16	rw-rw-r--	root	supergroup
part-1.4	dir				2015-02-24 23:16	rw-rw-r--	root	supergroup

Go back to DFS browser

Note: Create partition for all values.

9.5.7 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 partitions. This can be avoided by using Bucketing in which you can limit the number of buckets that will be created. A bucket is a file whereas a partition is a directory.

Objective: To learn the concept of 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 OVERWRITE TABLE STUDENT_BUCKET
```

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

// To display content of first bucket

```
SELECT DISTINCT GRADE FROM STUDENT_BUCKET
```

```
TABLESAMPLE(BUCKET 1 OUT OF 3 ON GRADE);
```


Outcome:

```
hive> CREATE TABLE IF NOT EXISTS STUDENT (rollno INT,name STRING,grade FLOAT)  
> ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
```

OK

Time taken: 0.101 seconds

```
hive>
```

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

Loading data to table book.student

Table book.student stats: [numFiles=1, totalSize=145]

OK

Time taken: 0.536 seconds

```
hive>
```

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

```
hive>
```

```
hive> CREATE TABLE IF NOT EXISTS STUDENT_BUCKET (rollno INT,name STRING,grade FLOAT)  
> CLUSTERED BY (grade) into 3 buckets;
```

OK

Time taken: 0.101 seconds

```
hive>
```

```
hive> FROM STUDENT  
> INSERT OVERWRITE TABLE STUDENT_BUCKET  
> SELECT rollno,name,grade;
```

9.5.8 Views

5.0 Views

Hive, view support is available only in version starting from 0.6. Views are purely logical object.

Objective: To create a view table named "STUDENT_VIEW".

Act:

```
CREATE VIEW STUDENT_VIEW AS SELECT rollno, name FROM EXT_STUDENT;
```

Outcome:

```
hive> CREATE VIEW STUDENT_VIEW AS SELECT rollno,name FROM EXT_STUDENT;
OK
Time taken: 0.606 seconds
hive>
```

Objective: Querying the view "STUDENT_VIEW".

Act:

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

Outcome:

```
hive> SELECT * FROM STUDENT_VIEW LIMIT 4;
OK
1001    John
1002    Jack
1003    Smith
1004    Scott
Time taken: 0.279 seconds, Fetched: 4 row(s)
hive>
```

Objective: To drop the view "STUDENT_VIEW".

Act:

```
DROP VIEW STUDENT_VIEW;
```

Outcome:

```
hive> DROP VIEW STUDENT_VIEW;
OK
Time taken: 0.452 seconds
hive>
```

9.5.9 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 subquery 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;
```

7.5.10 Joins

Joins in Hive is similar to the SQL Join.

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,name 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,name STRING)  
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
```

```
LOAD DATA LOCAL INPATH '/root/hivedemos/department.tsv' OVERWRITE INTO  
TABLE DEPARTMENT;
```

```
SELECT a.rollno, a.name, a.gpa, b.deptno FROM STUDENT a JOIN DEPARTMENT b ON  
a.rollno = b.rollno;
```

9.5.11 Aggregation

Hive supports aggregation functions like avg, count, etc.

Objective: To write the average and count aggregation functions.

Act:

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

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

Outcome:

```
hive> SELECT avg(gpa) FROM STUDENT;
```

```
OK
3.8399999961853027
Time taken: 28.639 seconds, Fetched: 1 row(s)
hive>
```

```
hive> SELECT count(*) FROM STUDENT;
```

```
OK
10
Time taken: 26.218 seconds, Fetched: 1 row(s)
hive>
```

9.5.12 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 out 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;
```

Outcome:

```
1003    Smith    4.5
1004    Scott    4.2
1006    Alex     4.5
1007    David    4.2
Time taken: 78.972 seconds, Fetched: 4 row(s)
hive>
```


9.6 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 RCFILE Format.

Act:

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

Outcome:

```
hive> CREATE TABLE STUDENT_RC( rollno int, name string,gpa float ) STORED AS RCFILE;  
OK  
Time taken: 0.093 seconds  
hive>
```

```
hive> INSERT OVERWRITE table STUDENT_RC SELECT * from STUDENT;
```

```
hive> SELECT SUM(gpa) from STUDENT_RC;
```

```
OK  
38.39999961853027  
Time taken: 25.41 seconds, Fetched: 1 row(s)  
hive>
```


9.8 USER-DEFINED FUNCTION (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();
    }
}
```

Note: Convert this Java Program into Jar.

```
ADD JAR /root/hivedemos/UpperCase.jar;  
CREATE TEMPORARY FUNCTION touppercase AS 'com.example.hive.udf.MyUpperCase';  
SELECT TOUPPERCASE(name) FROM STUDENT;
```

Outcome:

```
hive> ADD JAR /root/hivedemos/UpperCase.jar;  
Added [/root/hivedemos/UpperCase.jar] to class path  
Added resources: [/root/hivedemos/UpperCase.jar]  
hive> CREATE TEMPORARY FUNCTION touppercase AS 'com.example.hive.udf.MyUpperCase';  
OK  
Time taken: 0.014 seconds  
hive> 
```

```
hive> Select touppercase (name) from STUDENT;  
OK  
JOHN  
JACK  
SMITH  
SCOTT  
JOSHI  
ALEX  
DAVID  
JAMES  
JOHN  
JOSHI  
Time taken: 0.061 seconds, Fetched: 10 row(s)  
hive> 
```

10.9 EXECUTION MODES OF PIG

You can execute pig in two modes:

1. Local Mode.
2. MapReduce Mode.

10.9.1 Local Mode

To run pig in local mode, you need to have your files in the local file system.

Syntax:

```
pig -x local filename
```

10.9.2 MapReduce Mode

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

Syntax:

```
pig filename
```

10.11 RELATIONAL OPERATORS

10.11.1 FILTER

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]#
```

10.11.2 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 (rollno:int,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@volgalnx010 pigdemos]#

10.11.3 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)

Act:

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

```
B = GROUP A BY gpa;
```

```
DUMP B;
```

Output:

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


10.11.4 DISTINCT

DISTINCT operator is used to remove duplicate tuples. In Fig, *DISTINCT* operator works on the entire tuple 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]#
```


10.11.5 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;
```

Output:

```
(1001,John,3.0)
(1002,Jack,4.0)
(1003,Smith,4.5)
[root@volga1nx010 pigdemos]#
```

10.11.6 ORDER BY

ORDER BY is used to sort a relation based on 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;
```

[No Title]

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]#
```

10.11.7 JOIN

It is used to join two or more relations based on values in the common field. 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)
(1003,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,ATex,4.5,1006,101,B.E)
(1007,David,4.2,1007,104,MCA)
(1008,James,4.0,1008,102,B.Tech)
[root@volga1nx010 pigdemo]#
```

10.11.8 UNION – Used to merge contents of two relationships

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.

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
<u>SUCCESS</u>	file	0 B	3	128 MB	2015-02-24 17:23	rw-r--r--	root	supergroup
<u>part-m-00000</u>	file	146 B	3	128 MB	2015-02-24 17:23	rw-r--r--	root	supergroup
<u>part-m-00001</u>	file	114 B	3	128 MB	2015-02-24 17:23	rw-r--r--	root	supergroup

10.11.9 SPLIT – Used to partition a relation into two or more relationship

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;
```

Output: Relation X

```
(1002,Jack,4.0)
(1008,James,4.0)
[root@volga1nx010 pigdemo]#
```

Output: Relation Y

```
(1001,John,3.0)
(1002,Jack,4.0)
(1005,Joshi,3.5)
(1008,James,4.0)
(1001,John,3.0)
(1005,Joshi,3.5)
[root@volga1nx010 pigdemo]#
```


10.11.10 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;
```


10.12 EVAL FUNCTION

10.12.1 AVG

AVG is used to compare 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 A.studname, AVG(A.marks);  
DUMP C;
```

Output:

```
{(Jack),(Jack),(Jack),(Jack)},39.75)  
{(John),(John),(John),(John)},39.0)  
[root@volgalnx010 pigdemos]#
```

Note: You need to use PigStorage function if you wish to manipulate files other than .tsv.

10.12.2 MAX

MAX is used to compute the maximum of numeric values in a single column bag.

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)  
[root@volgalnx010 pigdemos]#
```

Note: Similarly, you can try the MIN and the SUM functions as well.

10.12.3 COUNT

COUNT 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;
```

Output:

```
{{(Jack),(Jack),(Jack),(Jack)},4}  
{{(John),(John),(John),(John)},4}  
[root@volgalnx010 pigdemo]#
```



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

10.13 COMPLEX DATA TYPES

10.13.1 TUPLE

A *TUPLE* is an ordered collection of fields.

Objective: To use the complex data type "Tuple" to load data.

Input:

(John,12)	(Jack,13)
(James,7)	(Joseph,5)
(Smith,8)	(Scott,12)

Act:

```
A = LOAD '/root/pigdemos/studentdata.tsv' AS (t1:tuple(t1a:chararray,
t1b:int),t2:tuple(t2a:chararray,t2b:int));
B = FOREACH A GENERATE t1.t1a, t1.t1b,t2.$0,t2.$1;
DUMP B;
```

Output:

```
(John,12,Jack,13)
(James,7,Joseph,5)
(Smith,8,Scott,12)
[root@volgalnx010 pigdemos]#
```

10.13.2 MAP

MAP represents a key/value pair.

Objective: To depict the complex data type “map”.

Input:

John [city#Bangalore]

Jack [city#Pune]

James [city#Chennai]

Act:

```
A = load '/root/pigdemos/studentcity.tsv' Using PigStorage as  
(studname:chararray,m:map[chararray]);
```

```
B = foreach A generate m#'city' as CityName:chararray;
```

```
DUMP B
```

Output:

```
(Bangalore)  
(Pune)  
(Chennai)  
[root@volgalnx010 pigdemos]#
```


10.14 PIGGY BANK

Pig user can use Piggy Bank functions in Pig Latin script and they can also share their functions in Piggy Bank.

Objective: To use Piggy Bank string UPPER function.


Input:

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

Act:

```
register '/root/pigdemos/piggybank-0.12.0.jar';  
A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);  
upper = foreach A generate  
org.apache.pig.piggybank.evaluation.string.UPPER(name);  
DUMP upper;
```

Output:



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

Note: You need to use the “**register**” keyword to use Piggy Bank jar function in your pig script.

10.15 USER-DEFINED FUNCTIONS (UDF)

Pig allows you to create your own function for complex analysis.

Objective: To depict user-defined function.

Java Code to convert name into uppercase:

```
package myudfs;
import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;
import org.apache.pig.impl.util.WrappedIOException;
public class UPPER extends EvalFunc<String>
{
    public String exec(Tuple input) throws IOException {
        if (input == null || input.size() == 0)
            return null;
        try{
            String str = (String)input.get(0);
            return str.toUpperCase();
        }catch(Exception e){
            throw WrappedIOException.wrap("Caught exception processing input row ", e);
        }
    }
}
```

Note: Convert above java class into jar to include this function into your code.

Input:

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

Act:

```
register /root/pigdemos/myudfs.jar;
A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);
B = FOREACH A GENERATE myudfs.UPPER(name);
DUMP B;
```

Output:

```
(JOHN)
(JACK)
(SMITH)
(SCOTT)
(NOSHI)
(ALEX)
(DAVID)
(JAMES)
(JOHN)
(NOSHI)
[root@volga1nx010 pigdemos]#
```

10.22 PIG versus HIVE

Features	Pig	Hive
Used By	Programmers and Researchers	Analyst
Used For	Programming	Reporting
Language	Procedural data flow language	SQL Like
Suitable For	Semi - Structured	Structured
Schema/Types	Explicit	Implicit
UDF Support	YES	YES
Join/Order/Sort	YES	YES
DFS Direct Access	YES (Implicit)	YES (Explicit)
Web Interface	YES	NO
Partitions	YES	NO
Shell	YES	YES