Introduction to Hive: What is Hive, Hive Architecture, Hive data types, Hive file formats, Hive Query Language (HQL), RC File implementation, User Defined Function (UDF). Introduction to Pig: What is Pig, Anatomy of Pig, Pig on Hadoop, Pig Philosophy, Use case for Pig, Pig Latin Overview, Data types in Pig, Running Pig, Execution Modes of Pig, HDFS Commands, Relational Operators, Eval Function, Complex Data Types, Piggy Bank, User Defined Function, Pig Vs Hive.

4.1 What is HIVE?

Hive is a Data Warehousing tool that sits on top of Hadoop.

Hive Suitable For		
Data warehousing applications	**Processes batch jobs on huge data that is immutable (data whose structure cannot be changed after it is created is called immutable data)**	**Examples: Web Logs, Application Logs**

Hive is used to process structured data in Hadoop.

The three main tasks performed by Apache Hive are:

1. Summarization
2. Querying
3. Analysis

Facebook initially created Hive component to manage their ever-growing volumes of log data. Later Apache software foundation developed it as open-source and it came to be known as Apache Hive.

Hive makes use of the following:

1. HDFS for Storage.
2. MapReduce for execution.

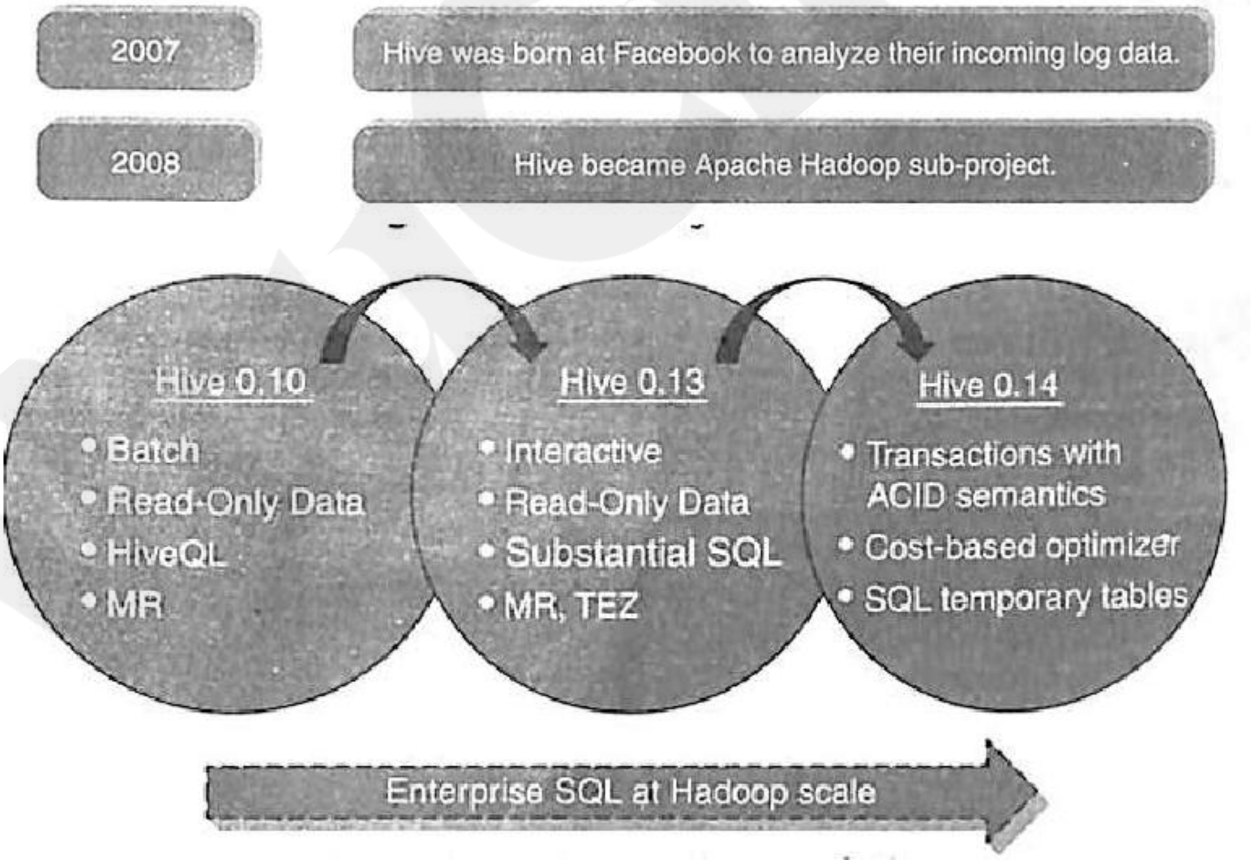
3. Stores metadata/schemas in an RDBMS.

Hive provides HQL (Hive Query Language) or HiveQL which is similar to SQL. Hive compiles SQL queries into MapReduce jobs and then runs the job in the Hadoop Cluster. It is designed to support OLAP (Online Analytical Processing). Hive provides extensive data type functions and formats for data summarization and analysis.

Note:

1. Hive is not RDBMS.
2. It is not designed to support OLTP (Online Transaction Processing).
3. It is not designed for real-time queries.
4. It is not designed to support row-level updates.

4.1.1 History of Hive and recent releases of Hive

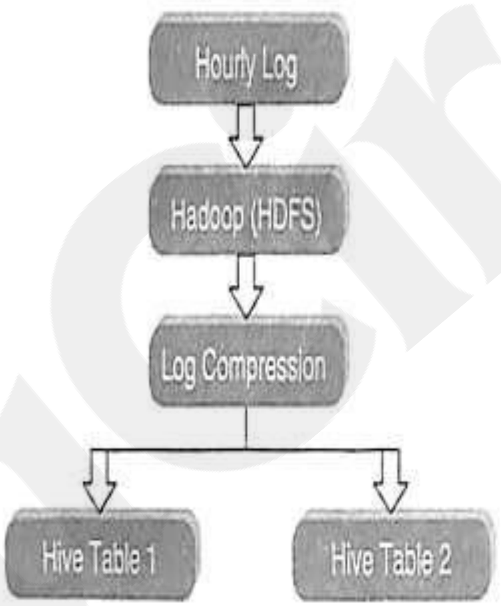


4.1.2 Hive Features

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.

4.1.3 Hive Integration and Work Flow

Figure shows 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 file. Finally, Hive table(s) can be created to query the log file.

4.1.4 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 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.

- Partitioning tables changes how Hive structures the data storage.
- Hive will create subdirectories reflecting the partitioning structure.
- Although partitioning helps in enhancing performance and is recommended, having too many partitions may prove detrimental for few queries.
- Bucketing is another technique of managing large datasets. If we partition the dataset based on customer_ID, we would end up with far too many partitions. Instead, if we bucket the customer table and use customer_id as the bucketing column, the value of this column will be hashed by a user-defined number into buckets.

When to Use Partitioning/Bucketing?

Bucketing works well when the field has high cardinality (cardinality is the number of values a column or field can have) and data is evenly distributed among buckets. Partitioning works best when the cardinality of the partitioning field is not too high. Partitioning can be done on multiple fields with an order (Year/Month/ Day) whereas bucketing can be done on only one field.

Figure shows how these data units are arranged in a Hive Cluster.

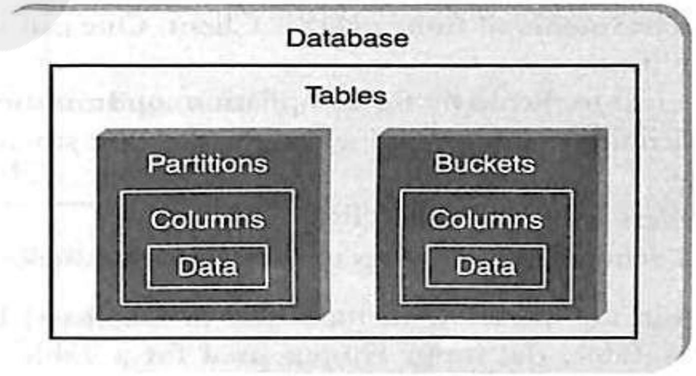
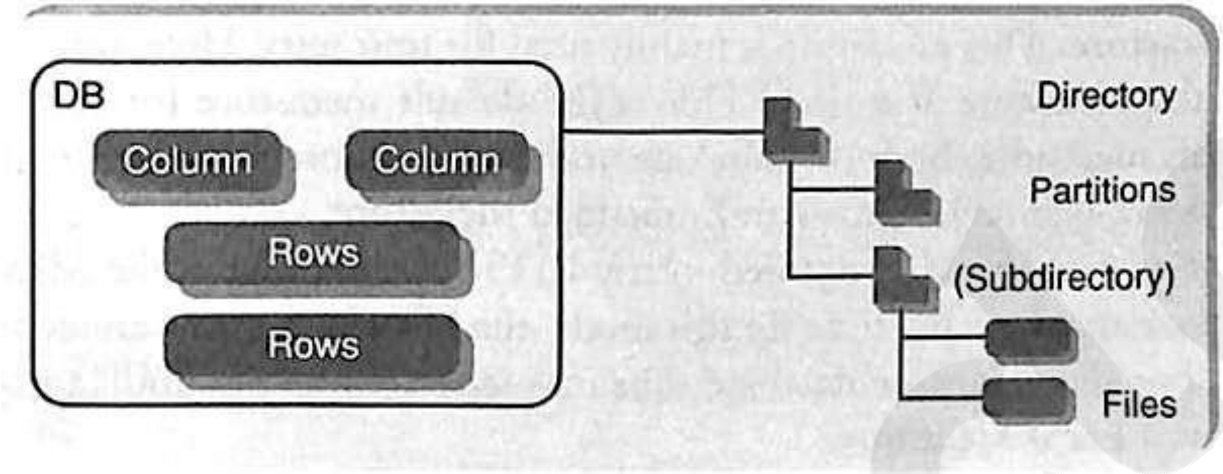


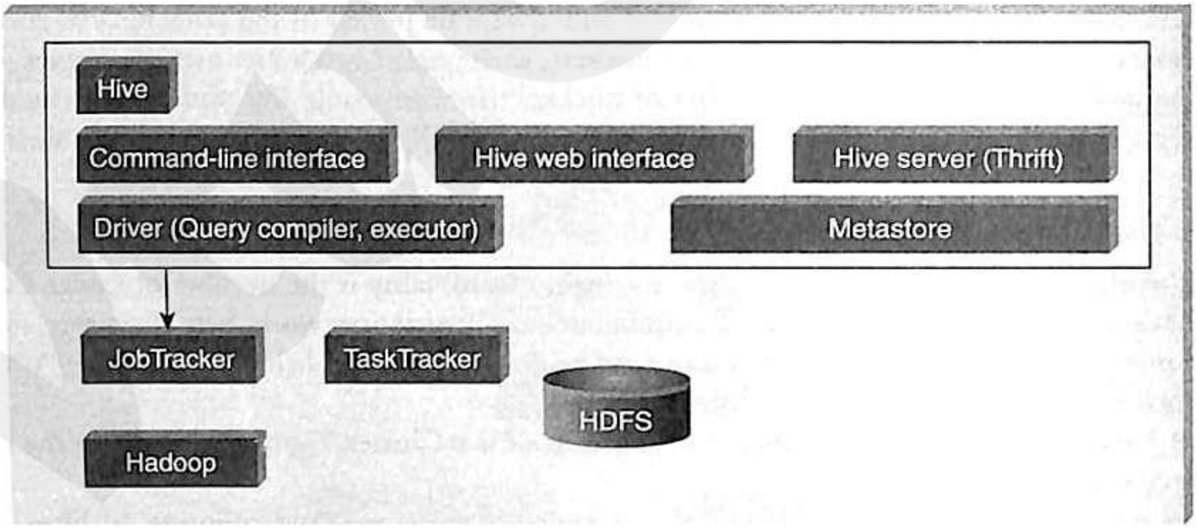
Figure below describes the semblance of Hive structure with database.



A database contains several tables. Each table is constituted of rows and columns. In Hive, tables are stored as a folder and partition tables are stored as a sub-directory. Bucketed tables are stored as a file.

4.2 HIVE ARCHITECTURE

The below figure shows Hive Architecture.



The various parts are as follows:

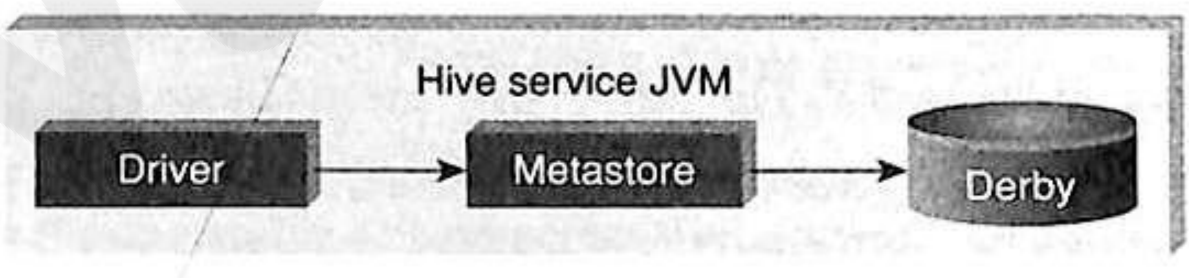
1. **Hive Command-Line Interface (Hive CLI):** The most commonly used interface to interact with Hive.

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.

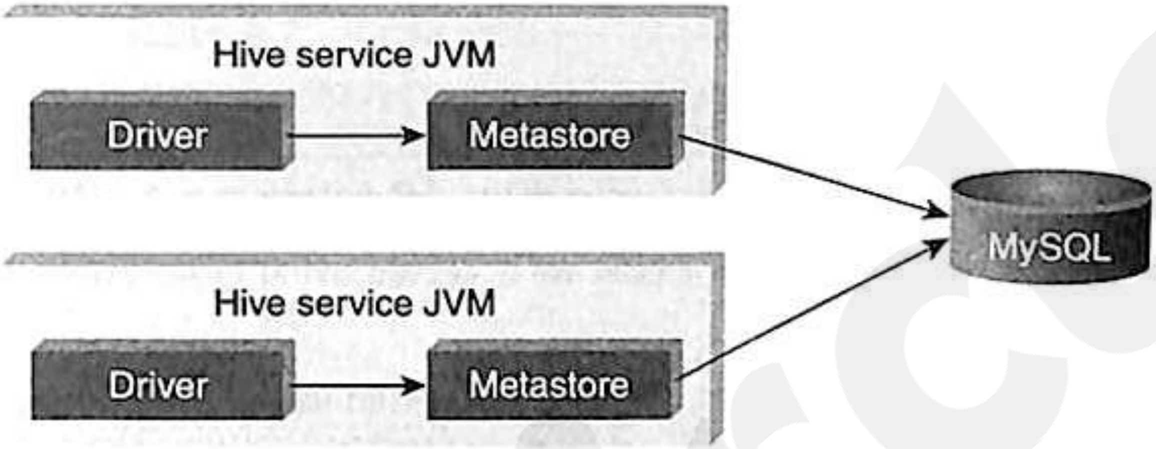
The metadata which is stored in the metastore includes IDs of Database, IDs of Tables, IDs of Indexes, etc., the time of creation of a Table, the Input Format used for a Table, the Output Format used for a Table, etc. The metastore is updated whenever a table is created or deleted from Hive.

There are three kinds of metastore.

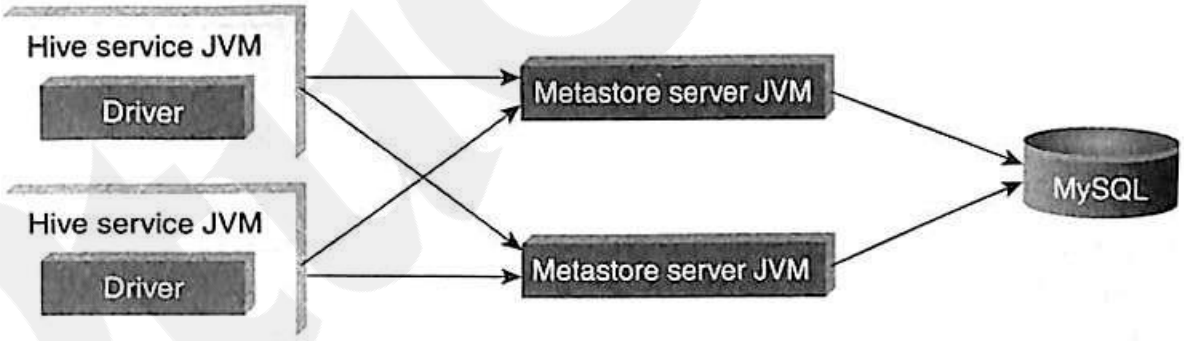
1. Embedded Metastore: This metastore is mainly used for unit tests. Here, only one process is allowed to connect to the metastore at a time. This is used for default setup in Hive. It is Apache Derby Database. In this metastore, both the database and the metastore service run embedded in the main Hive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

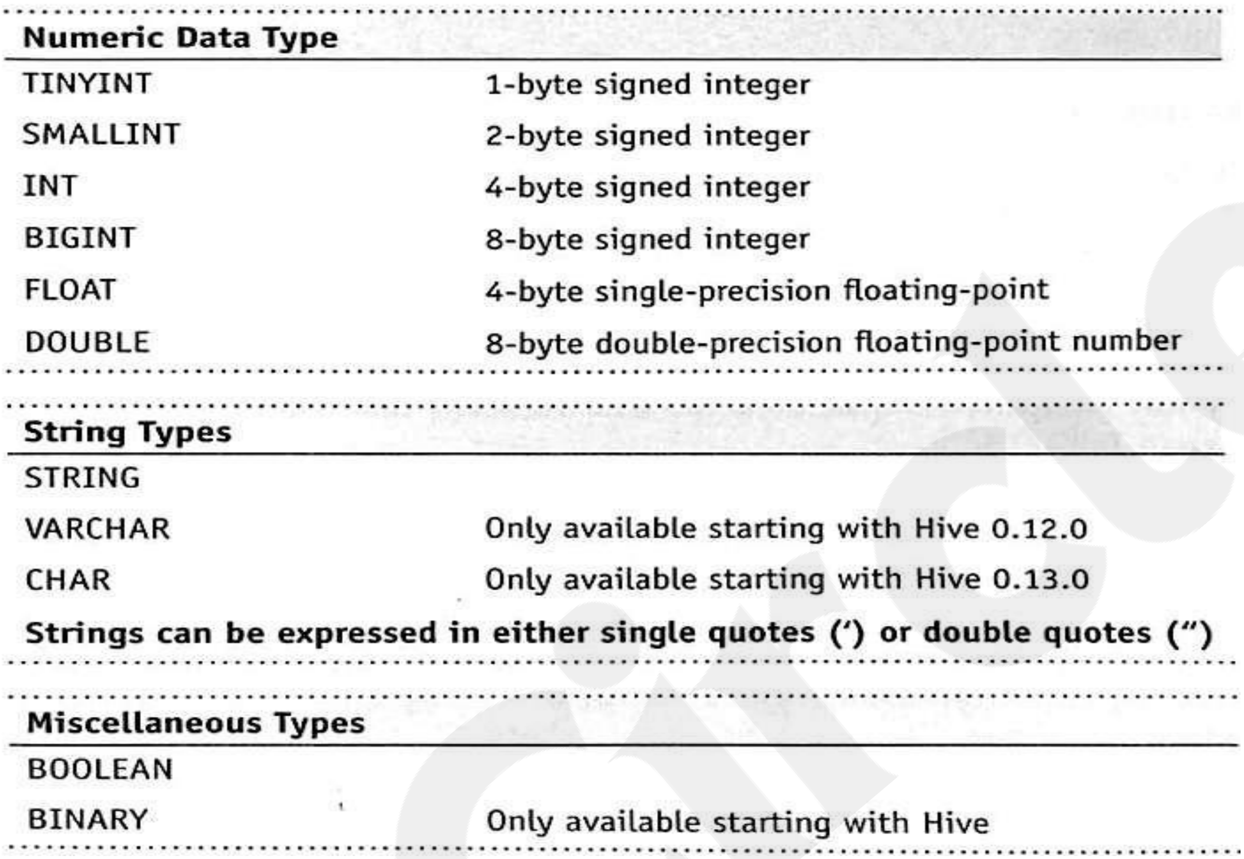


3. Remote Metastore: In this, the Hive driver and the metastore interface run on different JVMs (which can run on different machines as well) as in Figure below. This way the database can be fire-walled from the Hive user and also database credentials are completely isolated from the users of Hive.



4.3 HIVE Datatypes

4.3.1 PRIMITIVE DATATYPES



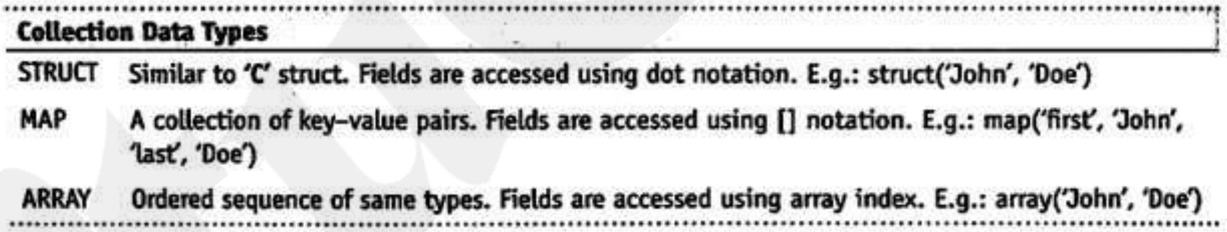
Numeric Data Type	
TINYINT	1-byte signed integer
SMALLINT	2-byte signed integer
INT	4-byte signed integer
BIGINT	8-byte signed integer
FLOAT	4-byte single-precision floating-point
DOUBLE	8-byte double-precision floating-point number

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

4.3.2 COLLECTION DATATYPES



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')

4.4 HIVE FILE FORMAT

4.4.1. 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.

Types of Supported Text File Formats

Even though the default format is a plain text file, several structured formats are also treated as text files:

CSV (Comma-Separated Values)

Fields are separated by commas

Example: Alice,22,Delhi

TSV (Tab-Separated Values)

Fields are separated by a tab (\t)

Example: Bob\t23\tMumbai

JSON (JavaScript Object Notation)

Structured text format often used to store objects and arrays.

Example: {"name": "Charlie", "age": 30, "city": "Chennai"}

XML (eXtensible Markup Language)

Uses tags to structure data

Example:<person><name>Diana</name><age>28</age><city>Pune</city></person>

Even though JSON and XML look very different from CSV/TSV, they are still text files because they store data in readable text form.

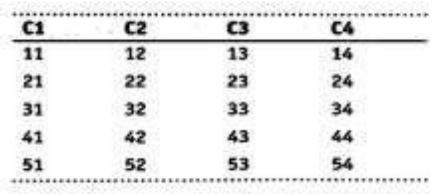
4.4.2. Sequential File

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

4.4.3. RCFile (Record Columnar File)

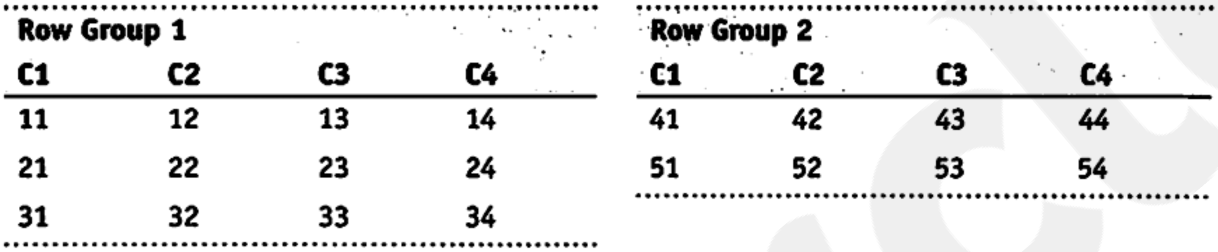
RCFile stores the data in Column Oriented Manner which ensures that Aggregation operation is not an expensive operation.

For example, consider a table which contains four columns as shown in Table 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 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-2, Table-1 is partitioned into two row groups by considering three rows as the size of each row group.



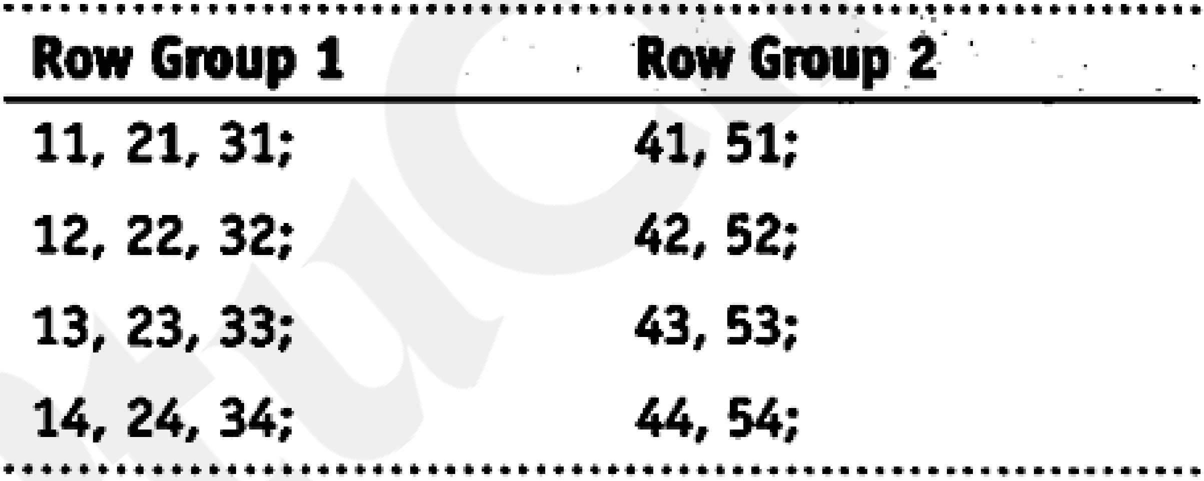
Row Group 1

C1	C2	C3	C4
11	12	13	14
21	22	23	24
31	32	33	34

Row Group 2

C1	C2	C3	C4
41	42	43	44
51	52	53	54

Next, in every row group RCFile partitions the data vertically like column-store. So the table will be serialized as shown in Table-3.



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

Hive query language provides basic SQL like operations. Below 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.

4.5.1. 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

4.5.2. 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.

4.5.3. Starting Hive Shell

Steps to Start Hive Shell:

1. Navigate to the installation path of Hive.
2. Open the terminal and type the command: **hive**
3. When executed, the terminal initializes Hive and you might see log messages related to SLF4J bindings and configurations.
4. After successful initialization, the hive> prompt appears, meaning Hive Shell is ready to accept commands.

Understanding the Log Output:

SLF4J Binding Messages:

These messages indicate that Hive is loading its logging configuration files and choosing the appropriate logging framework.

If multiple bindings are found, SLF4J selects one to use for logging output.

4.5.4. Database

A **Database** in Hive is a logical container used to organize tables. It helps manage and group related data sets within the Hive ecosystem.

Objective:

To create a database named **STUDENTS** with a comment for clarity and custom properties for identification.

Command (Act):

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

IF NOT EXISTS: Prevents errors if the database already exists.

COMMENT: Describes the purpose of the database.

WITH DBPROPERTIES: Allows setting key-value properties for the database (metadata).

Outcome:

Hive creates the database STUDENTS if it doesn't already exist.

Displays:

```
OK
Time taken: 0.536 seconds
hive>
```

The hive> prompt returns, ready for the next command.

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.

Note: We have not specified the location where the Hive database will be created. By default all the Hive databases will be created under default warehouse directory (set by the property hive.metastore.warehouse. dir) as /user/hive/warehouse/database_name.db. But if we want to specify our own location, then the LOCATION clause can be specified. This clause is optional.

Objective:

To display a list of all databases available in Hive.

Command (Act):

SHOW DATABASES;

Outcome:

Lists all databases present in the Hive metastore.

Example output:

students

Time taken: 0.082 seconds, Fetched: 22 row(s)

The hive> prompt returns, ready for the next command.

Additional Notes:

The SHOW DATABASES command lists all databases.

The word **SCHEMAS** can be used instead of **DATABASES**:

SHOW SCHEMAS;

The command supports the LIKE clause for filtering results based on patterns:

* matches multiple characters.

? matches a single character.

Examples with LIKE Clause:

SHOW DATABASES LIKE 'Stu*'; Lists all databases whose names start with "Stu".

SHOW DATABASES LIKE 'Stud???'; Lists all databases whose names start with "Stud" followed by exactly 3 characters.

Objective:

To view details about an existing database in Hive.

Command (Act):

DESCRIBE DATABASE STUDENTS;

This command displays the **database name**, **comment (description)**, and the **directory location** in HDFS where the database is stored.

Outcome (Sample Output):

students STUDENT Details

hdfs://volgalnx010.ad.infosys.com:9000/user/hive/warehouse/students.

db root USER

Time taken: 0.03 seconds, Fetched: 1 row(s)

students: The name of the database.

STUDENT Details: The comment provided during creation.

hdfs://.../students.db: The HDFS directory path where the database is located.

root USER: Owner information.

This command is useful for confirming database metadata and verifying storage paths.

Objective:

To describe the extended properties of a database.

Command (Act):

DESCRIBE DATABASE EXTENDED STUDENTS;

This command displays additional information such as **DB properties** (defined using DBPROPERTIES during creation), along with standard database metadata.

Outcome (Sample Output):

```
students STUDENT Details
hdfs://volgalnx010.ad.infosys.com:9000/user/hive/warehouse/students.
db root USER {creator=JOHN}
Time taken: 0.027 seconds, Fetched: 1 row(s)
```

Name: students

Comment: STUDENT Details

Location: HDFS path where the database is stored

Owner: root, role: USER

Properties: {creator=JOHN} (user-defined property)

DESCRIBE DATABASE EXTENDED reveals **DBPROPERTIES**, unlike the basic DESCRIBE DATABASE command.

SCHEMA can be used instead of DATABASE:

```
DESCRIBE SCHEMA EXTENDED STUDENTS;
```

DESC is a shorthand for DESCRIBE:

```
DESC DATABASE EXTENDED STUDENTS;
```

This command is useful when you need a complete overview of database metadata and user-defined properties.

Objective:

To alter the database properties.

Command (Act):

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

The ALTER DATABASE command is used to:

1. Add new **key-value** pairs into DBPROPERTIES.
2. Set the **owner** or **role** for the database.

Important: Hive does **not** allow unsetting (removing) existing DB properties.

Outcome (Sample Output):

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

This confirms that the property was added successfully.

Verification:

You can verify the updated properties using the command:

```
DESCRIBE DATABASE EXTENDED STUDENTS;
```

This will now include:
{creator=JOHN, edited-by=JAMES}

Objective:

To make a database the current working database.

Command (Act):

```
USE STUDENTS;
```

Outcome (Example):

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

There is **no direct command** to display the **current active database** in Hive. However, to make the command prompt display the current database name as a suffix, use the following setting:

```
set hive.cli.print.current.db=true;
```

This will help you keep track of which database you're working in during a Hive session.

Objective:

To drop a database.

Command (Act):

```
DROP DATABASE STUDENTS;
```

Hive stores databases in the warehouse directory (e.g., /user/hive/warehouse). Deleting a database only works if **no tables** exist inside it — this is the **default RESTRICT mode**.

To drop a database along with all its tables:

```
DROP DATABASE STUDENTS CASCADE;
```

The **CASCADE** option deletes the database **along with all contained tables**. **RESTRICT** (default) prevents deletion if any tables are present.

Complete Syntax:

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

4.5.5 Tables

Hive provides two kinds of table:

1. **Internal or Managed Table**
2. **External Table**

4.5.5.1 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 a managed table named STUDENT.

Command (Act):

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

Explanation:

CREATE TABLE IF NOT EXISTS ensures the table is only created if it doesn't already exist.

This is a **managed table** – Hive manages both the data and metadata.

ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' specifies that fields are separated by **tab characters** in the input data.

Outcome Example:

Table creation completes successfully.

Execution time may vary (e.g., 0.355 seconds in the sample output).

Objective:

To describe the structure and metadata of the STUDENT table.

Command (Act):

```
DESCRIBE STUDENT;
```

Outcome:

Lists the columns in the STUDENT table with their respective data types:

```
rollno int
name string
gpa float
```

Note:

Hive stores managed tables in the warehouse directory (e.g., /user/hive/warehouse/students.db).

The table data resides inside a subdirectory (like student inside students.db).

To Check if a Table is Managed or External

Command:

```
DESCRIBE FORMATTED STUDENT;
```

Details:

Displays **complete metadata** of the table.

One of the fields will be:

Table Type: MANAGED_TABLE – if Hive manages both data and metadata.

Table Type: EXTERNAL_TABLE – if data is managed outside Hive.

4.5.5.2 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.

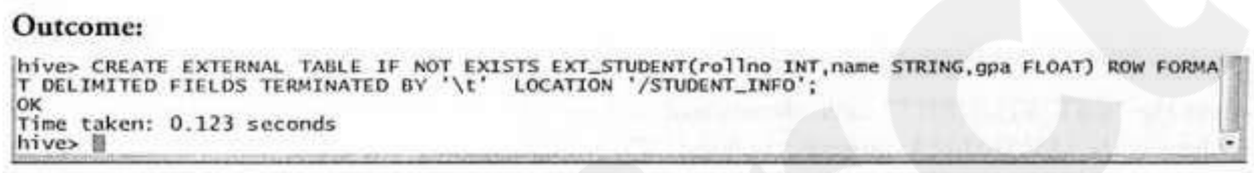
3. Location needs to be specified to store the dataset in that particular location**.**

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:



Outcome:

```
hive> CREATE EXTERNAL TABLE IF NOT EXISTS EXT_STUDENT(rollno INT,name STRING,gpa FLOAT) ROW FORMA
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.

4.5.5.3 Loading Data into Table from File

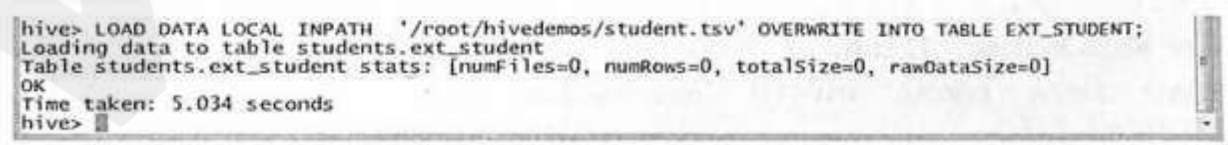
Objective: To load data into the table from file named student.tsv.

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>
```

4.5.5.4 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>)

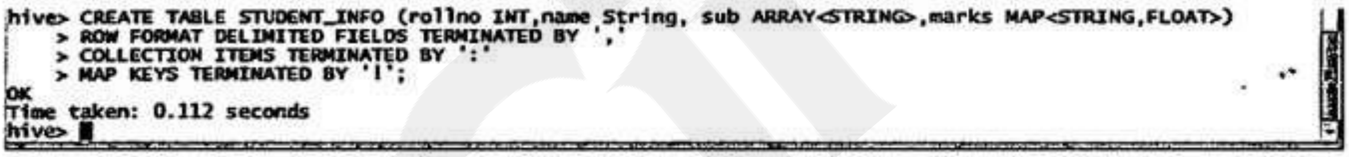
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','

COLLECTION ITEMS TERMINATED BY ':'

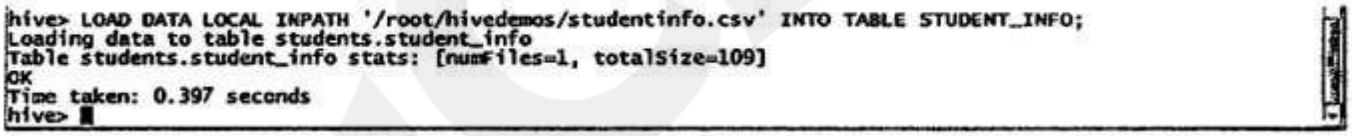
MAP KEYS TERMINATED BY '!';

LOAD DATA LOCAL INPATH '/root/hivedemos/studentinfo.csv' INTO 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> 
```

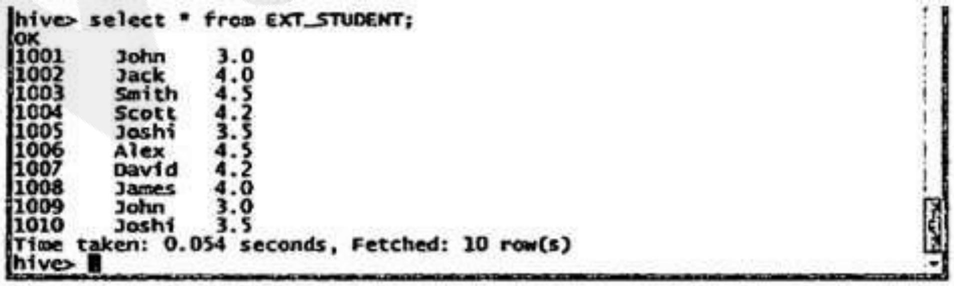
4.5.5.5 Querying Table

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;

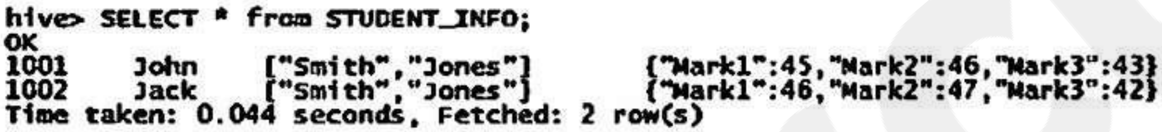
// To retrieve value of Mark1

SELECT NAME, MARKS['Mark1'] from STUDENT_INFO;-

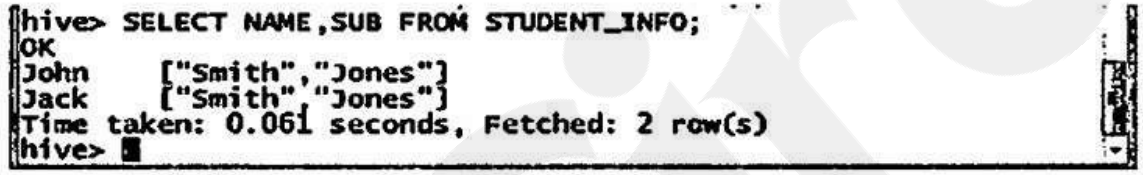
// To retrieve subordinate (array) value

SELECT NAME,SUB[0] FROM STUDENT_INFO;

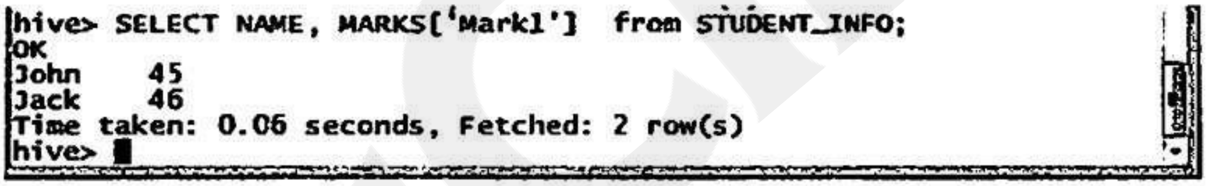
Outcome:



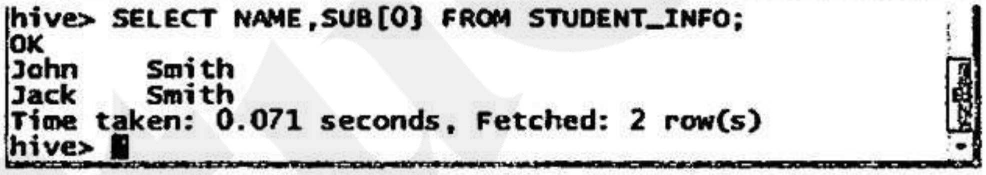
```
hive> SELECT * from STUDENT_INFO;
OK
1001    John    ["Smith","Jones"]       {"Mark1":45,"Mark2":46,"Mark3":43}
1002    Jack    ["Smith","Jones"]       {"Mark1":46,"Mark2":47,"Mark3":42}
Time taken: 0.044 seconds, Fetched: 2 row(s)
```



```
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>
```

4.5.6 Partitions

In Hive, without partitioning, queries scan the entire dataset, leading to high I/O and slow performance. Partitioning divides data into subdirectories (like by year or region), so Hive reads only relevant parts. This reduces I/O, speeds up queries, and improves MapReduce job efficiency.

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.

Note:

1. STATIC PARTITIONING implies that the user controls everything from defining the PARTITION column to loading data into the various partitioned folders.

2. 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.

3. 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
```

4.5.6.1 Static Partition

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

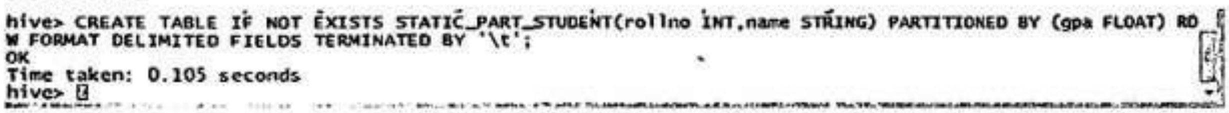
Objective: To create static partition based on "gpa" column.

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:

Outcome:



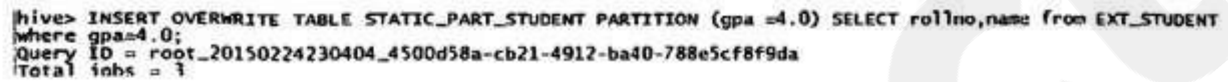
```
hive> CREATE TABLE IF NOT EXISTS STATIC_PART_STUDENT(rollno INT,name STRING) PARTITIONED BY (gpa FLOAT) RO
W FORMAT DELIMITED FIELDS TERMINATED BY '\t';
OK
Time taken: 0.105 seconds
hive>
```

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 = 1
```

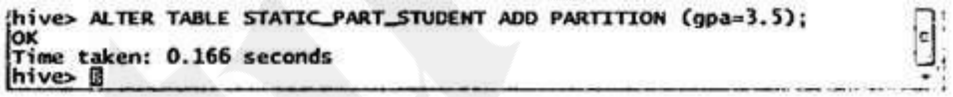
Hive creates the folder for the value specified in the partition.

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

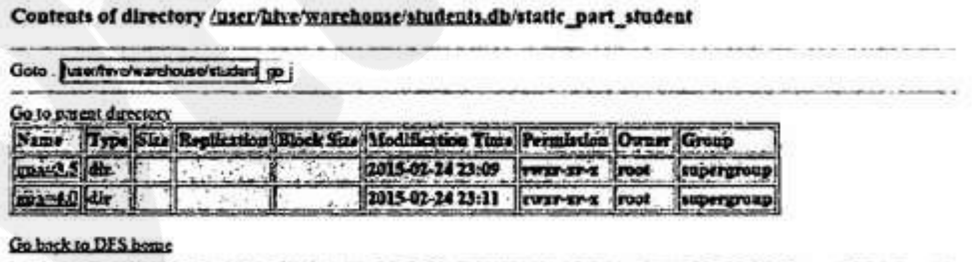
Act:

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 : /user/hive/warehouse/student go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
gpa=3.5	dir				2015-02-24 23:09	rwxr-xr-x	root	supergroup
gpa=4.0	dir				2015-02-24 23:11	rwxr-xr-x	root	supergroup

Go back to DFS home

4.5.6.2 Dynamic Partition

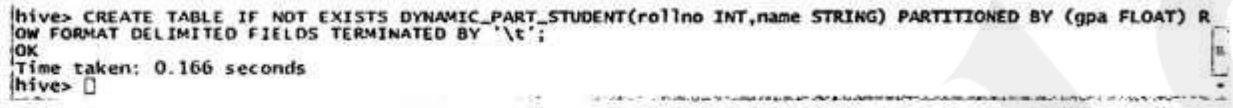
Dynamic partition have columns whose values are known only at Execution Time.

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:

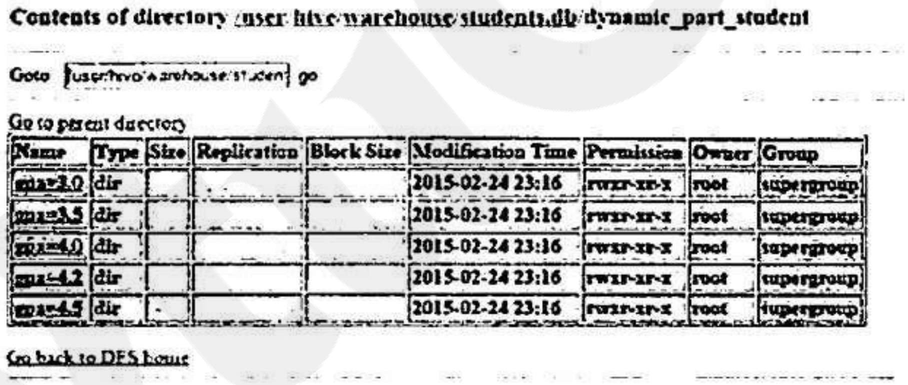
SET hive.exec.dynamic.partition = true;

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 /user/hive/warehouse/students.db/dynamic_part_student

Goto /user/hive/warehouse/studen go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
gpa=3.0	dir				2015-02-24 23:16	rwxr-xr-x	root	supergroup
gpa=3.5	dir				2015-02-24 23:16	rwxr-xr-x	root	supergroup
gpa=4.0	dir				2015-02-24 23:16	rwxr-xr-x	root	supergroup
gpa=4.2	dir				2015-02-24 23:16	rwxr-xr-x	root	supergroup
gpa=4.5	dir				2015-02-24 23:16	rwxr-xr-x	root	supergroup

Go back to DFS home

Note: Create partition for all values.

4.5.7 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);
```

4.5.8 Views

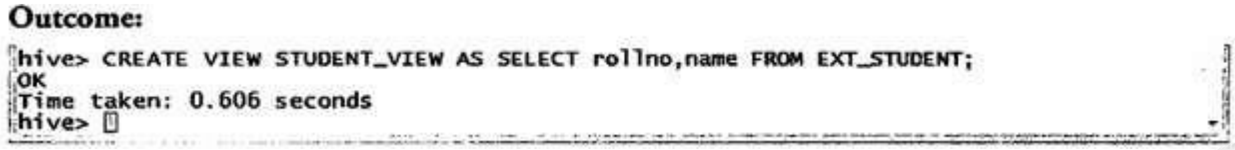
In Hive, view support is available only in version starting from 0.6. Views are purely logical object.

Objective: To create a view table named “STUDENT_VIEW”.

Act:

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

Outcome:



Outcome:

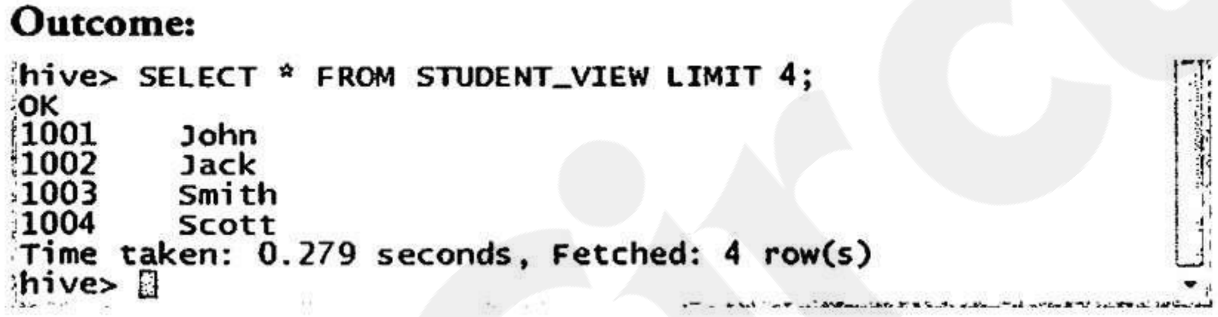
```
hive> CREATE VIEW STUDENT_VIEW AS SELECT rollno,name FROM EXT_STUDENT;
OK
Time taken: 0.606 seconds
hive> ▯
```

Objective: Querying the view "STUDENT_VIEW”.

Act:

SELECT * FROM STUDENT_VIEW LIMIT 4;

Outcome:



Outcome:

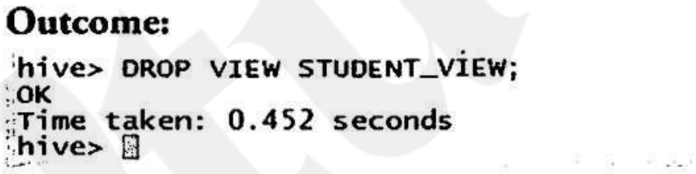
```
hive> SELECT * FROM STUDENT_VIEW LIMIT 4;
OK
1001    John
1002    Jack
1003    Smith
1004    Scott
Time taken: 0.279 seconds, Fetched: 4 row(s)
hive> ▯
```

Objective: To drop the view "STUDENT_VIEW”.

Act:

DROP VIEW STUDENT_VIEW;

Outcome:



Outcome:

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

4.5.9 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;
```

4.5.10 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.tsy'
OVERWRITE INTO TABLE DEPARTMENT;
SELECT a.rollno, a.name, a.gpa, b.deptno FROM STUDENT a JOIN
DEPARTMENT ь ON a.rollno = b.rollno;
```

4.5.11 Aggregation

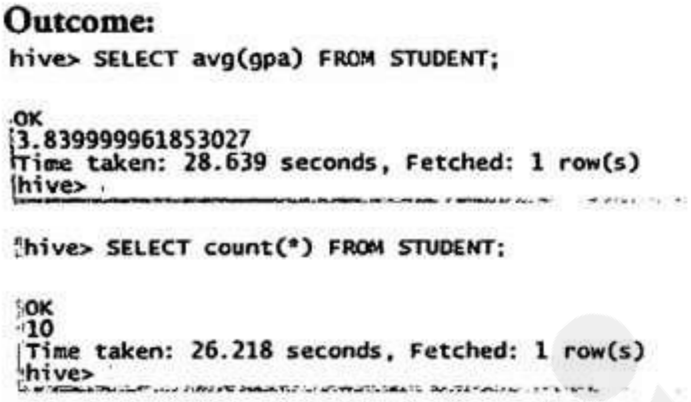
Hive supports aggregation functions like avg, count, etc.

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
3.839999961853027
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>
```

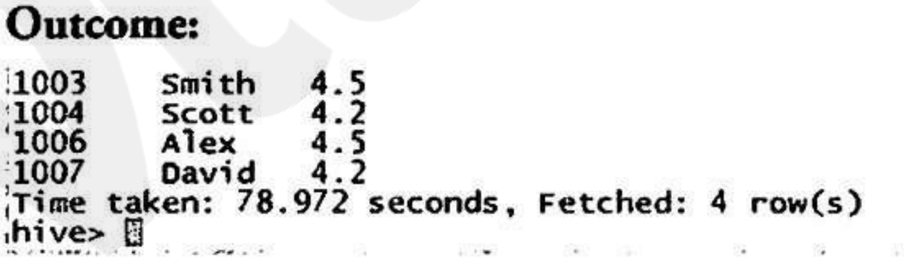
4.5.12 Group By and Having

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>
```

4.6 RCFILE IMPLEMENTATION

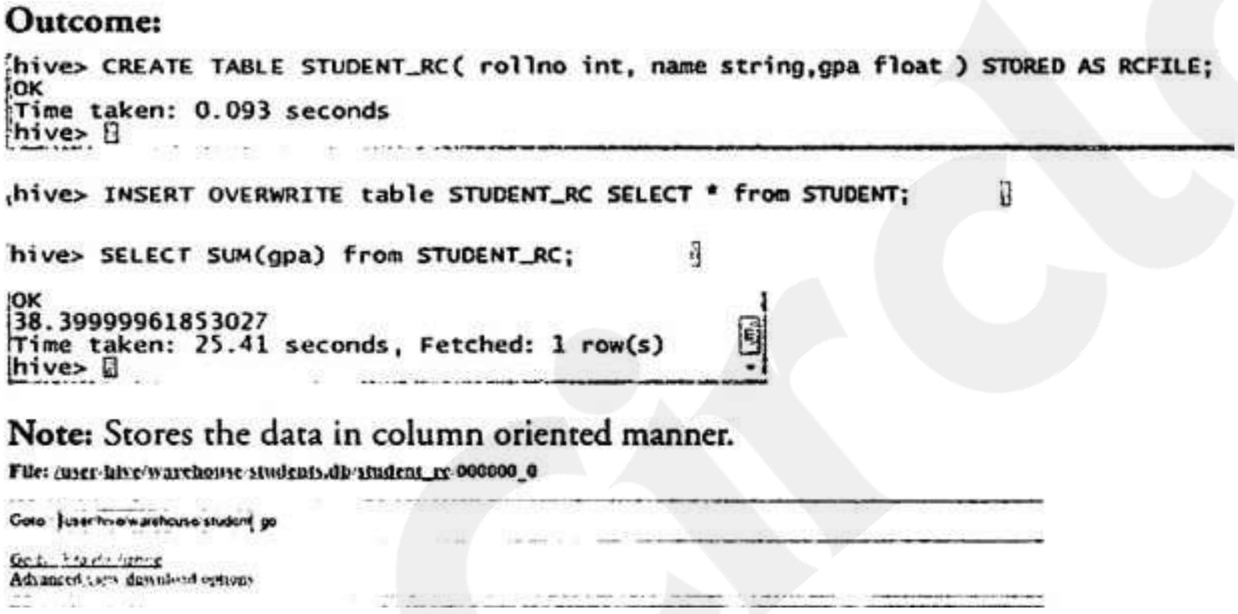
RCFile (Record Columnar File) is a data placement structure that determines how to store relational tables on computer clusters.

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:



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>
```

Note: Stores the data in column oriented manner.

File: /user/hive/warehouse/students.db/student_rc/000000_0

Goto /user/hive/warehouse/student go

Go t... [illegible]
Advanced ... download options

4.8 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;
```

INTRODUCTION TO PIG

4.9 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.

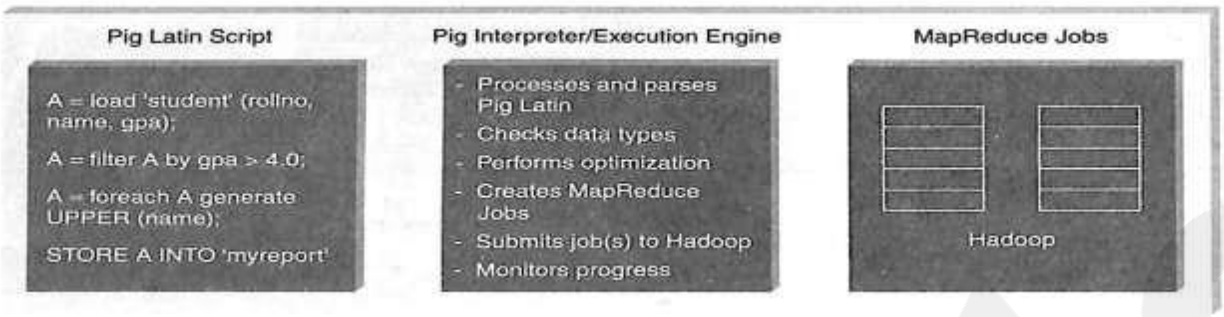
Key Features of Pig

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.

4.10 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.



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

4.11 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:

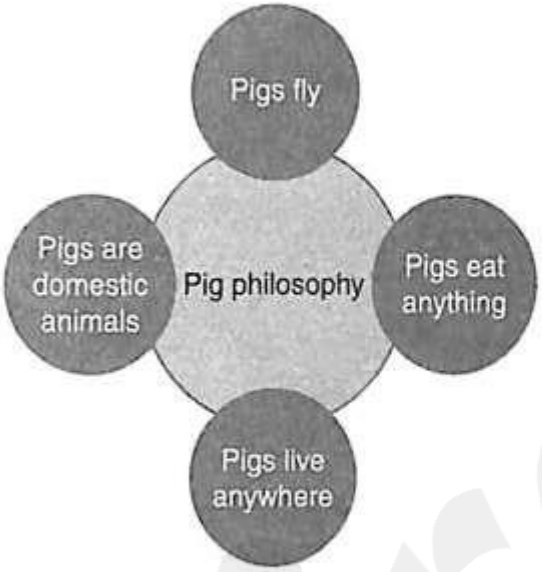
1. HDFS commands.
2. UNIX shell commands.
3. Relational operators.
4. Positional parameters.
5. Common mathematical functions.
6. Custom functions.
7. Complex data structures.

4.12 PIG Philosophy

Figure below describes the Pig philosophy.

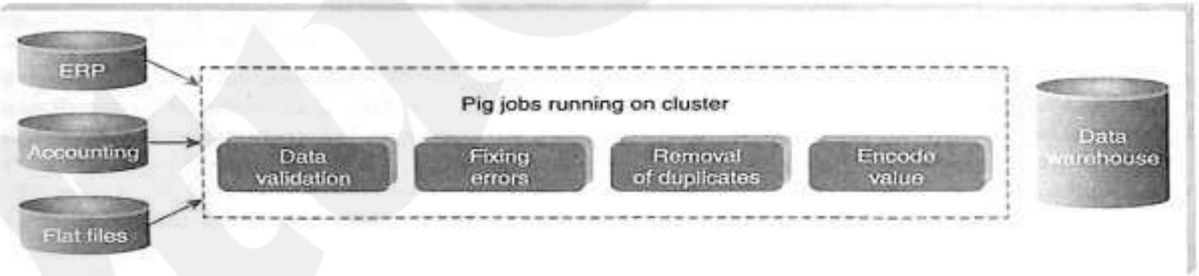
1. **Pigs Eat Anything:** Pig can process different kinds of data such as structured and unstructured data.
2. **Pigs Live Anywhere:** Pig not only processes files in HDFS, it also processes files in other sources such as files in the local file system.
3. **Pigs are Domestic Animals:** Pig allows you to develop user-defined functions and the same can be included in the script for complex operations.

4. **Pigs Fly:** Pig processes data quickly.



4.13 USE CASE FOR PIG: ETL PROCESSING

Pig is widely used for "ETL" (Extract, Transform, and Load). Pig 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. Refer Figure below.



4.14 PIG Latin Overview

4.14.1 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'
```

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

4.14.2 Pig Latin Keywords

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

LOAD: Loads data from a file or data source into a relation.

STORE: Stores a relation to a file or data source.

GROUP: Groups rows in a relation based on certain fields.

JOIN: Combines two relations based on a common field.

FILTER: Selects rows based on a condition.

FOREACH: Applies a transformation to each row in a relation.

DISTINCT: Removes duplicate rows from a relation.

DEFINE: Defines user-defined functions (UDFs).

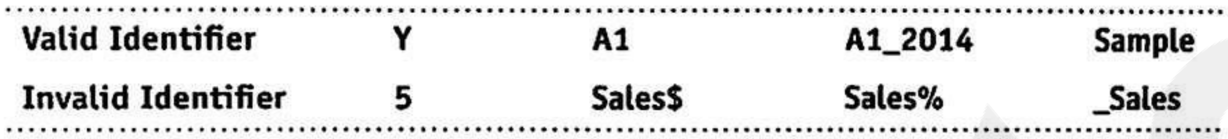
AS: Assigns an alias to a relation or field.

4.14.3 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.

The below table shows the list of valid and invalid identifiers.



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

4.14.4 Pig Latin: Comments

In Pig Latin two types of comments are supported:

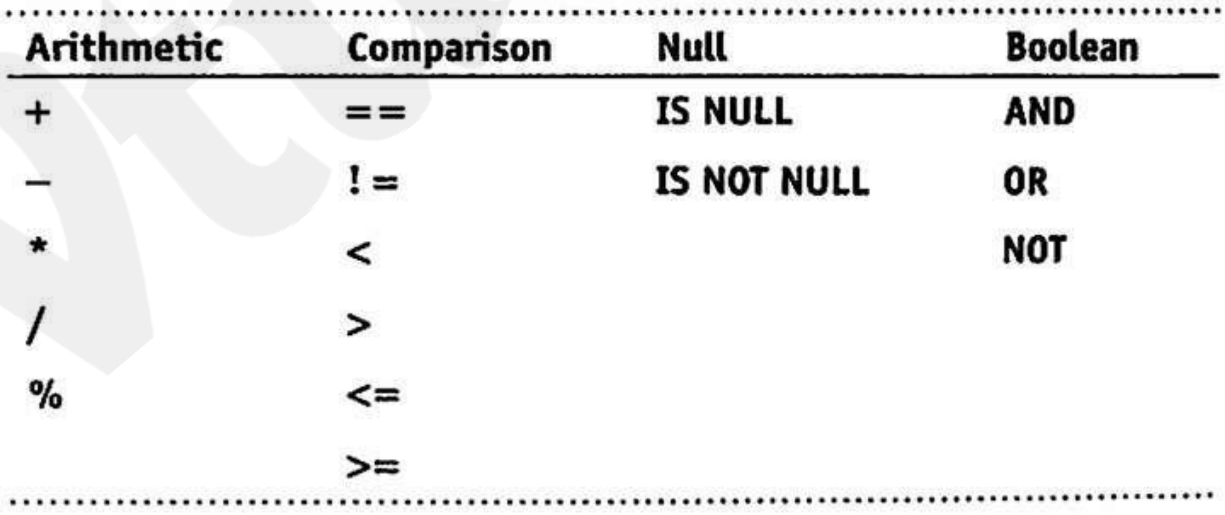
1. Single line comments that begin with "--".
2. Multiline comments that begin with "/* and end with */".

4.14.5 Pig Latin: Case Sensitivity

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.

4.14.6 Operators in Pig Latin

Table below describes operators in Pig Latin.

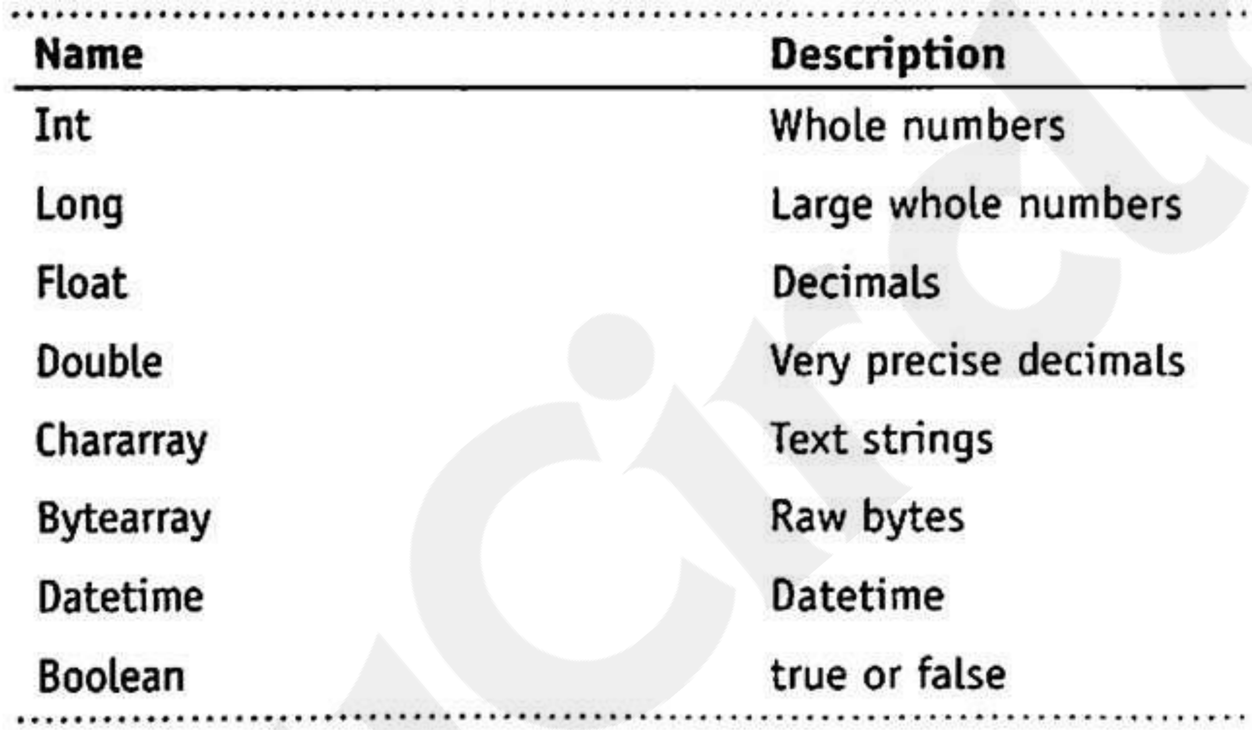


Arithmetic	Comparison	Null	Boolean
+	==	IS NULL	AND
–	!=	IS NOT NULL	OR
*	<		NOT
/	>		
%	<=		
	>=		

4.15 Simple Data Types

Table below describes simple data types supported in Pig. In Pig, fields of unspecified types are considered as an array of bytes which is known as bytearray.

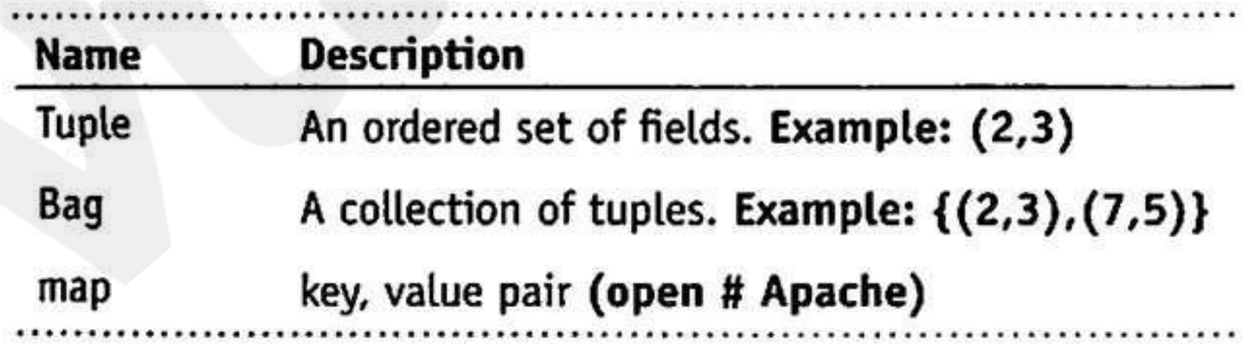
Null: In Pig Latin, NULL denotes a value that is unknown or is non-existent.



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

Complex Data Types

Table below 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)**

4.16 Running PIG

We can run Pig in 2 ways:

1. Interactive Mode
2. Batch Mode

1. **Interactive Mode:**

Command to start Pig shell:

pig

- Pig runs in Grunt shell, an interactive environment for executing Pig Latin scripts.
- Warnings show deprecated configuration parameters (e.g., mapred.job.tracker, fs.default.name) and suggest alternatives like mapreduce.jobtracker.address, fs.defaultFS.

Loading and Viewing Data in Pig

Pig Latin Command to load data:

```
A = LOAD '/pigdemo/student.tsv' AS (rollno, name, gpa);
```

Command to display the data:

```
DUMP A;
```

Output (Sample Data):

```
(1001, John, 3.0)
(1002, Jack, 4.0)
(1003, Smith, 4.5)
(1004, Scott, 4.2)
(1005, Joshi, 3.5)
```

1. The path /pigdemo/student.tsv refers to an HDFS location.
2. DUMP outputs the dataset to the console.
3. Each tuple contains: rollno (int), name (string), gpa (float).

2. **Batch Mode:**

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

4.17 EXECUTION MODES OF PIG

We can execute pig in two modes:

1. Local Mode.

2. MapReduce Mode.

Local Mode

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

Syntax:

```
pig -x local filename
```

MapReduce Mode

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

Syntax:

```
pig filename
```

4.18 HDFS COMMANDS in Pig (Grunt Shell)

Objective: Learn how to perform HDFS operations directly from the Pig Grunt shell.

Command Example:

```
grunt> fs -mkdir /piglatindemos;
```

Explanation:

The fs keyword allows HDFS commands within the Grunt shell.

-mkdir creates a new directory in HDFS at the specified path.

Outcome: A directory named /piglatindemos is created in HDFS.

4.19 Relational Operators

1. FILTER Operator in Pig

Purpose:

The FILTER operator is used to select tuples (records) from a relation based on specified conditions.

Objective:

Select students whose GPA is greater than 4.0.

Input:

Relation Student with schema:

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

Commands (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)
```

FILTER is used to apply conditions and reduce the dataset.

It helps in extracting meaningful subsets of data for further processing.

2. FOREACH Operator in Pig

Purpose:

The FOREACH operator is used for data transformation based on columns of a relation.

Objective:

Display the names of all students in **uppercase**.

Input:

Relation Student with schema:

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

Commands (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)
```

FOREACH ... GENERATE is used to apply transformations to each row.

Functions like UPPER() can be used to modify specific fields.

3. GROUP Operator in Pig

Purpose:

The GROUP operator is used to group data based on a column.

Objective:

Group tuples of students based on their GPA.

Input:

Relation Student with schema:

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

Commands (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)})
```

GROUP A BY <column> creates groups where all tuples with the same value in the specified column are grouped together.

Each group is represented as (group_key, bag_of_tuples).

4. DISTINCT Operator in Pig

Purpose:

The DISTINCT operator is used to remove duplicate tuples from a relation.

Important Note:

It operates on entire tuples, not on individual fields.

Objective:

Remove duplicate tuples of students.

Input:

Relation Student with schema:

(rollno:int, name:chararray, gpa:float) and data:

```
(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) <-- Duplicate
(1005, Joshi, 3.5) <-- Duplicate
```

Commands (Act):

```
A = LOAD '/pigdemo/student.tsv' AS (rollno:int, name:chararray,
gpa:float);
B = DISTINCT A;
DUMP B;
```

Output (unique tuples only):

```
(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)
```

1. The DISTINCT operator helps clean data by removing rows that are completely identical.
2. Useful before performing groupings or joins to reduce data redundancy.

5. LIMIT Operator in Pig

Purpose:

The LIMIT operator is used to restrict the number of output tuples from a relation.

Objective:

Display the first 3 tuples from the student relation.

Input Schema:

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

Pig Latin Script (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)
```

The LIMIT operator is useful for:

- Previewing data.
- Debugging scripts with large datasets.
- Controlling the volume of intermediate or final results.

6. ORDER BY Operator in Pig

Purpose:

The ORDER BY operator is used to sort a relation based on a specific column value.

Objective:

Display the names of students in ascending order.

Input Schema:

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

Pig Latin Script (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)
```

1. ORDER BY helps sort data globally, unlike GROUP which organizes data into bins.
2. The default is ascending order. To sort in descending order, use:

```
B = ORDER A BY name DESC;
```

7. JOIN Operator in Pig

Purpose:

The JOIN operator is used to combine two or more relations based on values in a common field (typically a key like rollno).

Pig always performs an inner join by default.

Objective:

Join two relations—student and department—based on the rollno column.

Input Schemas:

Student (rollno:int, name:chararray, gpa:float) Department (rollno:int, deptno:int, deptname:chararray)

Pig Latin Script (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;
```

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, Alex, 4.5, 1006, 101, B.E.)
(1007, David, 4.2, 1007, 104, MCA)
(1008, James, 4.0, 1008, 102, B.Tech)
```

1. The JOIN operator merges rows from two datasets where the join keys match.

2. Duplicate entries in either relation will result in multiple joined records (Cartesian effect for those keys).
3. Default join is inner join — only records with matching keys in both relations are included.

Syntax:

```
C = JOIN A BY key1, B BY key2;
```

8. UNION in Pig Latin

Objective:

To merge the contents of two relations student and department.

Input Data:

Student relation:

student.tsv Format: rollno:int, name:chararray, gpa:float

Department relation:

department.tsv Format: rollno:int, deptno:int, deptname:chararray

Pig Latin Script:

```
A = LOAD '/pigdemo/student.tsv' AS (rollno:int, name:chararray,
gpa:float);
B = LOAD '/pigdemo/department.tsv' AS (rollno:int, deptno:int,
deptname:chararray);
A_proj = FOREACH A GENERATE rollno, name AS info1, gpa AS info2;
B_proj = FOREACH B GENERATE rollno, deptname AS info1, deptno AS
info2;
C = UNION A_proj, B_proj
STORE C INTO '/pigdemo/uniondemo';
DUMP C;
```

- UNION in Pig is used to combine rows from two or more relations having the same schema.

- Since student and department have different schemas, we first project both into a common structure with rollno, a chararray field, and a numeric field.

Output Note:

- STORE saves the merged data into HDFS.
- The files inside /pigdemo/uniondemo represent distributed parts of the output.

9. SPLIT

It is used to partition a relation into two or more relations.

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

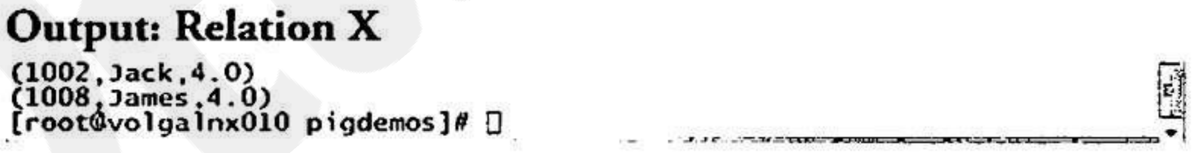
- GPA = 4.0, place it into relation X.
- GPA is < 4.0, place it into relation Y.

Input:

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

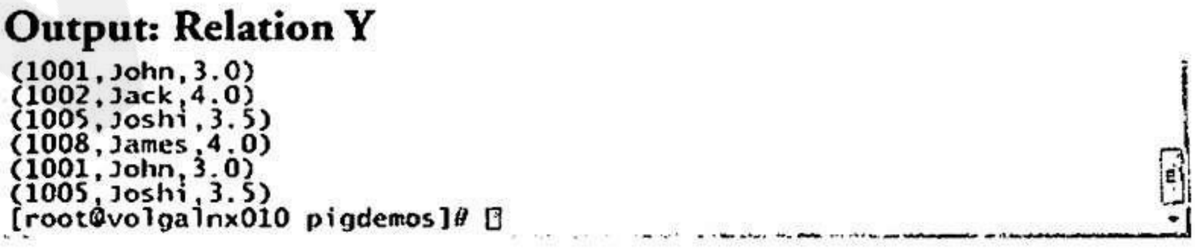
Act:

```
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;
```



Output: Relation X

```
(1002,Jack,4.0)
(1008,James,4.0)
[root@volgalnx010 pigdemos]# 
```



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@volgalnx010 pigdemos]# 
```

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;
```

4.20 EVAL FUNCTION

4.20.1 AVG

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

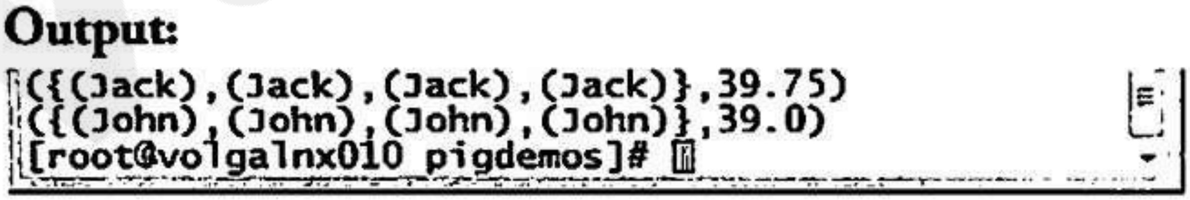
Objective: To calculate the average marks for each student.

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 A.studname, AVG(A.marks);
DUMP C;
```



Output:

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

4.20.2 MAX

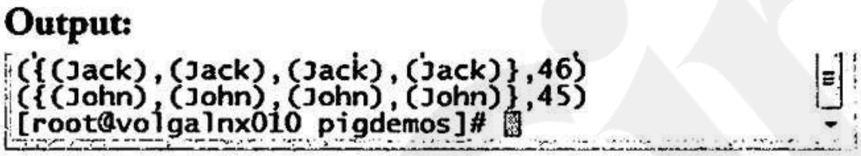
MAX is used to compute the maximum of numeric values in a single column bag.

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]#
```

4.20.3 COUNT

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

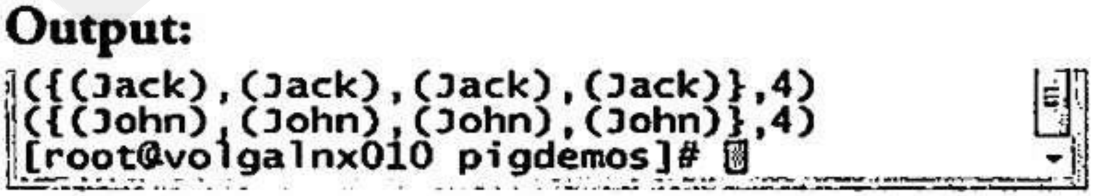
Objective: To count the number of tuples in a bag.

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 pigdemos]#
```

10.21 COMPLEX DATA TYPES

10.21.1 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)

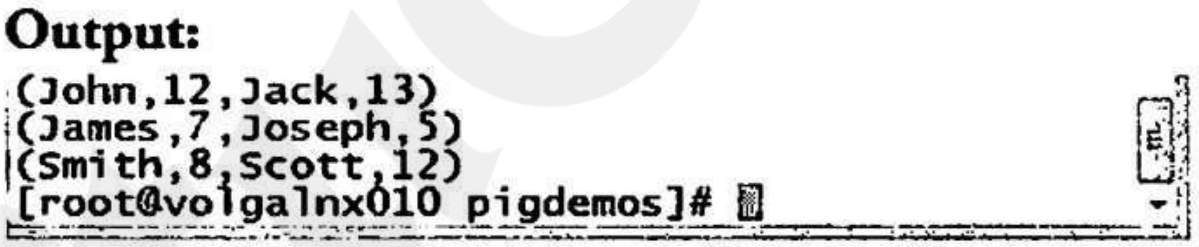
(Joseph,5)

(Smith,8)

(Scott,12)

Act:

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



Output:

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

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

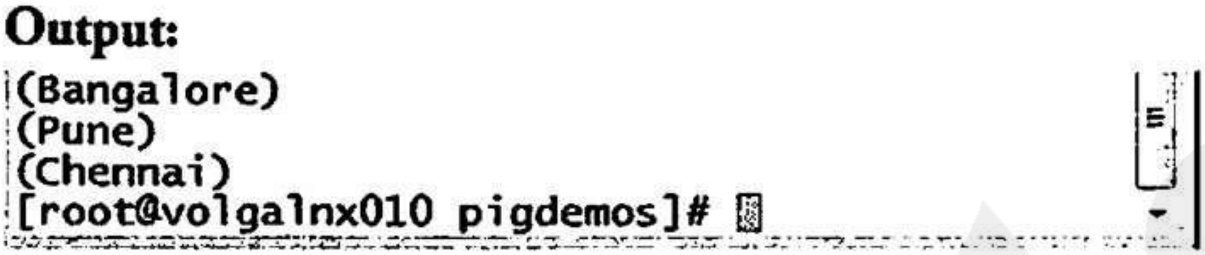
Input:

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]#
```

10.22 PIGGY BANK

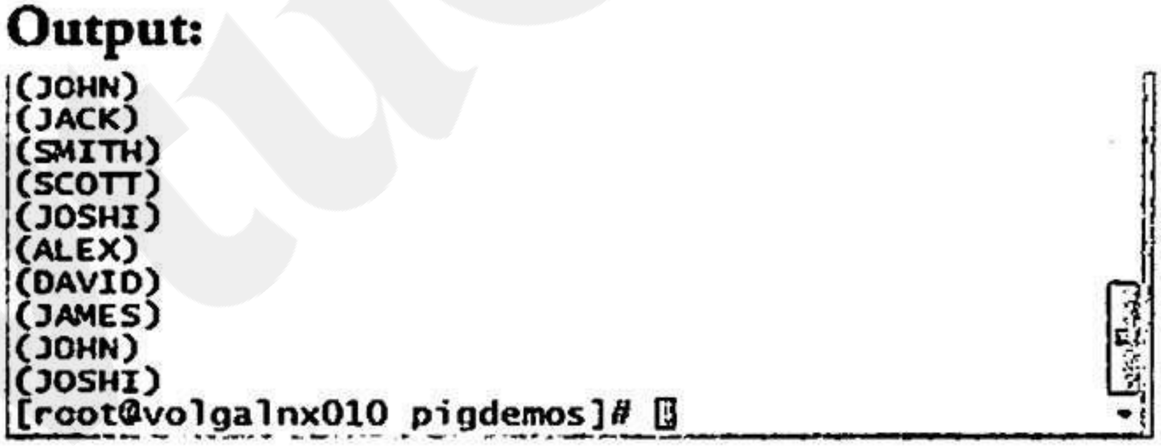
Pig user can use Piggy Bank functions in Pig Latin script and they can also share their functions in Pigg Bank.

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@volgalnx010 pigdemos]#
```

10.23 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.

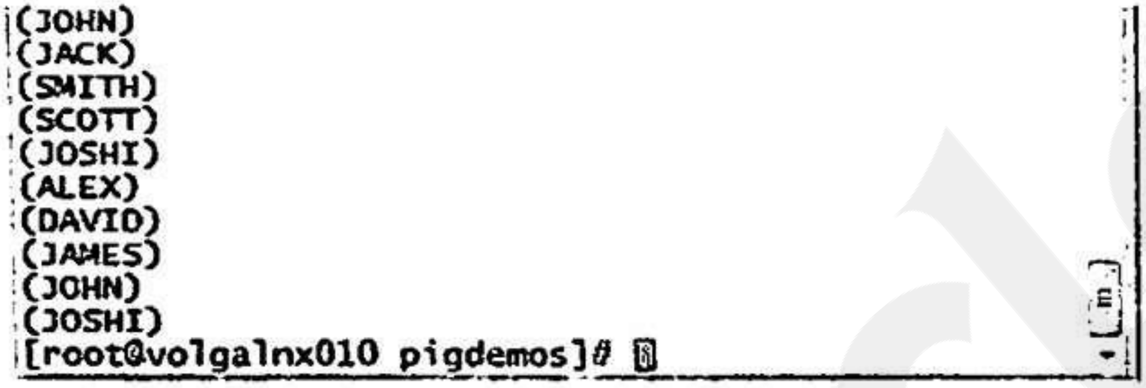
Input:

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

Act:

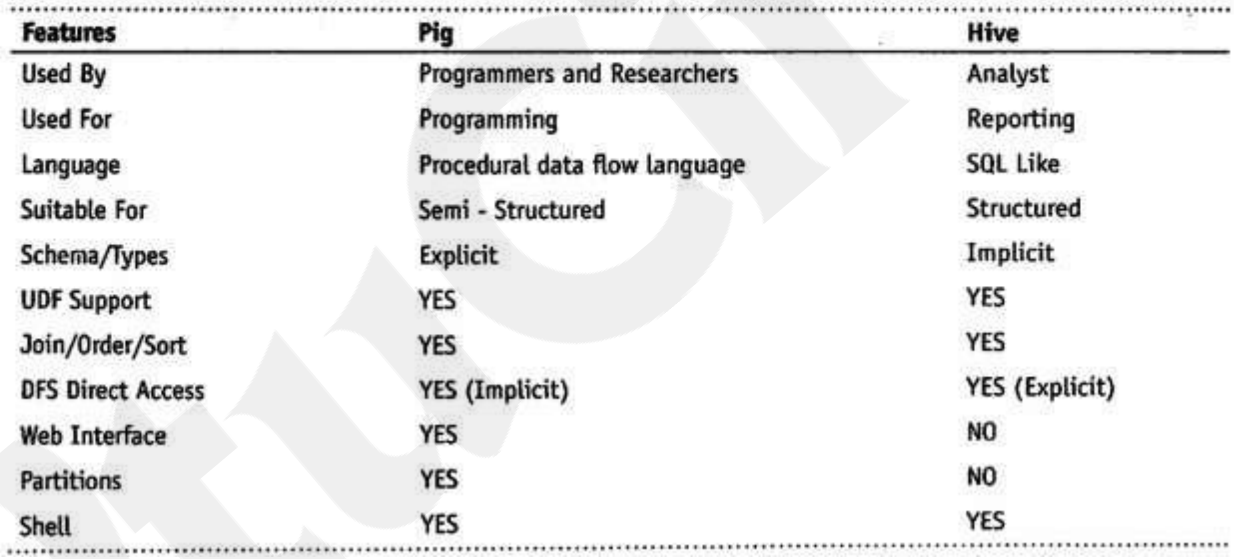
```
register /root/pigdemos/myudfs.jar;
A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);
B = FOREACH A GENERATE myudfs.UPPER(name);
DUMP B;
```

Output:



```
(JOHN)
(JACK)
(SMITH)
(SCOTT)
(JOSHI)
(ALEX)
(DAVID)
(JAMES)
(JOHN)
(JOSHI)
[root@volgalnx010 pigdemos]#
```

10.24 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

********END********