Introduction to HIVE and PIG

Syllabus:

Introduction to HIVE: *Introduction, HIVE architecture, HIVE data types, HIVE file formats, HIVE query language, RCFile implementation, SerDe, User Defined Functions (UDF)* ***Introduction to PIG:*** *Anatomy of PIG, PIG on Hadoop, PIG philosophy, overview of PIG, Data types in PIG, Running and execution modes of PIG, HDFS commands, Relational operators, Eval function, Complex Data types.*

Resources: *https://data-flair.training/blogs/ , Textbook-1*

Hive is a data warehouse infrastructure tool to process structured data in Hadoop. It resides on top of Hadoop to summarize Big Data, and makes querying and analyzing easy. Initially Hive was developed by Facebook, later the Apache Software Foundation took it up and developed it further as an open source under the name Apache Hive. It is used by different companies. For example, Amazon uses it in Amazon Elastic MapReduce.

History

The Figure below outlines the evolution of Hive releases (Hive 0.10, Hive 0.13, Hive 0.14) and their key features.

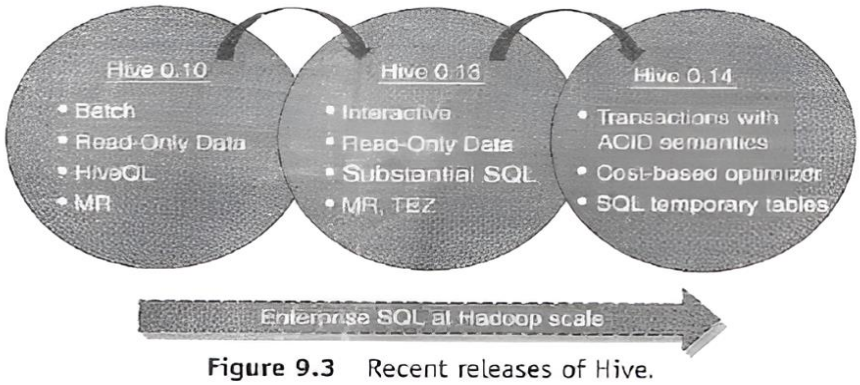


Figure 9.3 Recent releases of Hive.

Hive 0.10:

- **Batch Processing**: Primarily used for batch jobs with MapReduce (MR).
- **Read-Only Data**: Limited to querying existing datasets without supporting updates.
- **HiveQL**: Basic SQL-like query language for Hive.
- **MR (MapReduce)**: Relied solely on the MapReduce processing engine.

Hive 0.13:

- **Interactive Queries**: Enhanced query performance for interactive use cases.
- **Read-Only Data**: Continued support for querying static datasets.
- **Substantial SQL Support**: Added more advanced SQL features.
- **MR, TEZ**: Support for Tez, in addition to MapReduce, for faster query execution.

Hive 0.14:

- **Transactions with ACID Semantics**: Support for INSERT, UPDATE, DELETE operations.
- **Cost-Based Optimizer**: Improved query planning and execution efficiency.
- **SQL Temporary Tables**: Introduction of temporary tables for intermediate data.
-

Features of Hive

Apache Hive is a data warehouse software built on top of Hadoop that facilitates reading, writing, and managing large datasets stored in a distributed system using SQL-like queries. Below are the key features of Hive:

1. Similarity to SQL

- Hive uses **HiveQL (HQL)**, which is very similar to SQL, making it accessible to users familiar with traditional databases.
- SQL-like syntax allows seamless querying and analysis of big data without requiring knowledge of low-level programming.

2. Ease of Coding

- HiveQL is **simple and intuitive**, reducing the complexity of writing big data processing jobs.
- Simplifies the complexities of MapReduce programming, allowing users to concentrate on implementing business logic.

3. Rich Data Types and Collections

- Hive supports **primitive data types** like INT, FLOAT, STRING, etc., and **complex data types** such as:
  - **Structs**: For nested objects.
  - **Lists/Arrays**: For multiple values in a single column.
  - **Maps**: For key-value pairs.

4. Extensibility with Custom Functions

- Users can define:
  - **Custom Data Types** to handle specific types of data.
  - **Custom Functions** like UDFs (User-Defined Functions), for tailored data transformations.

5. Support for SQL Operations

- Hive supports standard SQL operations, including:
  - **Filters**: Apply conditions using WHERE clauses.
  - **Group By**: Aggregate data using GROUP BY.
  - **Order By**: Sort data with ORDER BY.

Hive integration and work flow

The diagram explains the integration of Hive for log file analysis workflow.
Here's a brief textual explanation of the workflow:

1. Hourly Log: Logs are collected on an hourly basis.

2. Hadoop (HDFS): These logs are stored in Hadoop Distributed File System (HDFS) for further processing.

3. Log Compression: The log data undergoes compression to save storage space and optimize performance.

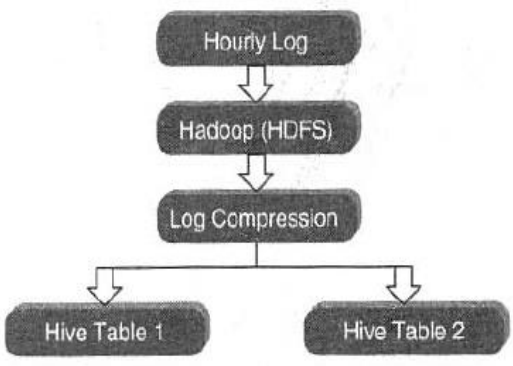


Figure 9.4 Flow of log analysis file.

4. Hive Table(s): The compressed data is stored in Hive tables (Hive Table 1 and Hive Table 2) to enable querying and analysis.

This workflow demonstrates the typical process of ingesting, compressing, and analyzing large-scale log data using Hadoop and Hive.

Hive Data Units

In Apache Hive, data units are organized as follows: **Database, Tables, Partitions, Buckets, Data.**

- **Databases:** The namespace for tables.
- **Tables:** Set of records that have similar schema.
- **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.
- **Buckets (Clusters):** Similar to partitions but uses hash functions to segregate data and determines the cluster or bucket into which the record should be placed.
- **Data:** The actual data resides at the lowest level, within the defined columns of tables, and is stored in the Hadoop Distributed File System (HDFS)

In Hive, tables are stored as a folder, partitions are stored as a sub-directory and buckets are stored as a file

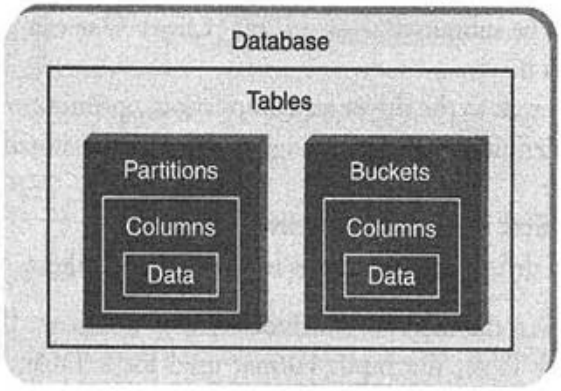


Figure 9.5 Data units as arranged in a Hive.

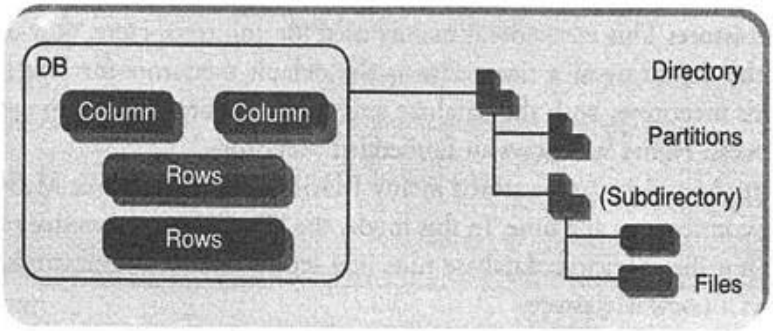


Figure 9.6 Semblance of Hive structure with database.

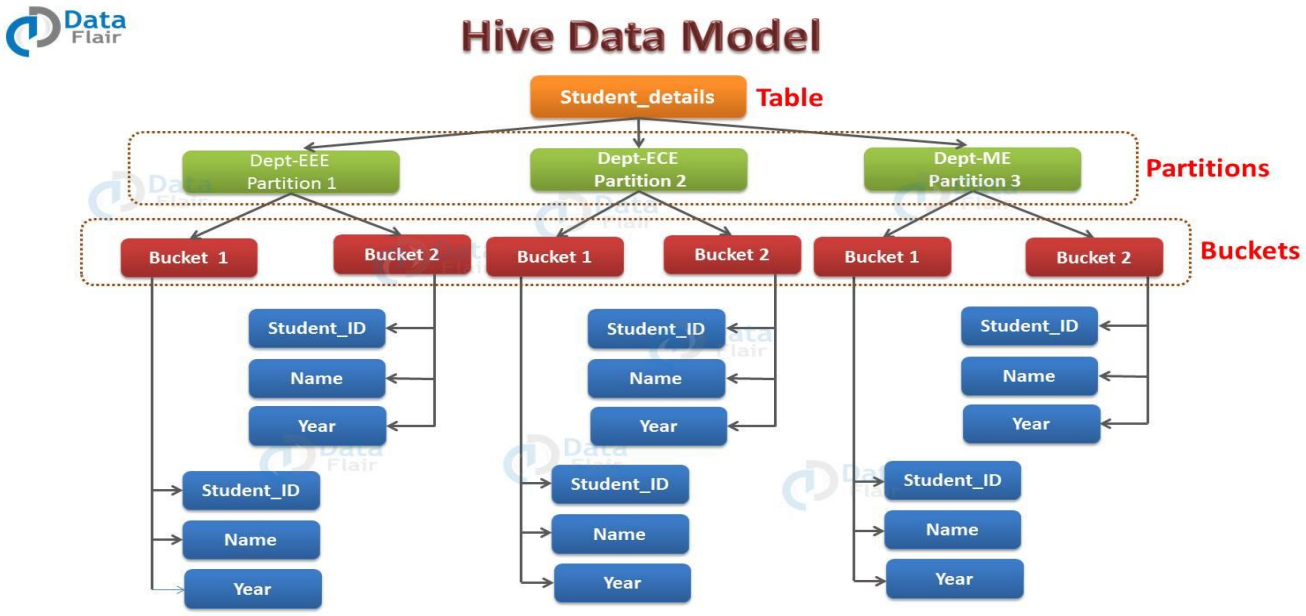
Figure 9.5 shows how data in Hive is organized:

- **Database**: Contains multiple tables.
- **Tables**: Divided into **Partitions** and **Buckets**.
  - **Partitions**: Logical divisions within the table based on a column, allowing efficient data retrieval.
  - **Buckets**: Further subdivisions of data within a partition, often based on a hash function applied to a column.
  - Both partitions and buckets are made up of **Columns** and **Data**.

Figure 9.6 compares Hive's structure with traditional database elements:

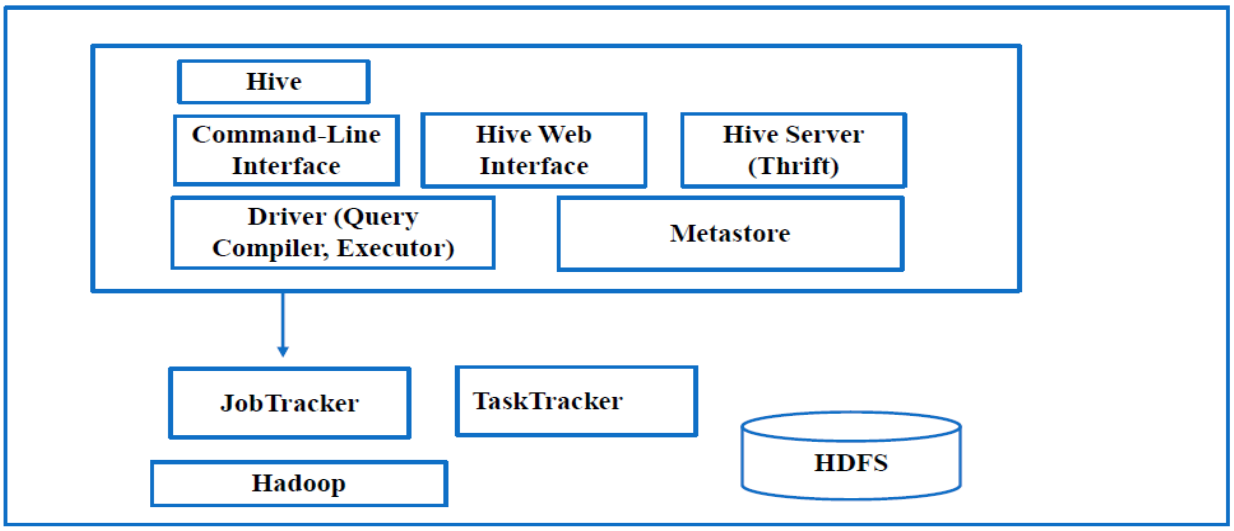
- **DB (Database)**:
  - Comprises **Columns** and **Rows** similar to relational database tables.
- **Directory Structure**:
  - Hive stores data on HDFS with a directory hierarchy.
  - **Partitions** map to subdirectories, organizing data based on specific keys or conditions.
  - **Files** within these subdirectories represent actual data blocks.

Example: With **Student** being the database-name, below shows the data units organization to store student details.



Hive Architecture

Apache Hive is a data warehouse framework built on top of Hadoop. It allows for querying and managing large datasets stored in HDFS (Hadoop Distributed File System) using a SQL-like language called HiveQL.



Components of Hive Architecture:

a. User Interfaces

i. **Command-line Interface (CLI):**
   - Users can execute HiveQL commands directly via the command line.

ii. **Hive Web Interface:**
   - A web-based UI for submitting and managing Hive queries.

iii. **Hive Server (Thrift Server):**
   - Acts as a middleware for communication between client applications and Hive. It supports JDBC/ODBC connections for external applications.

b. Driver- The **Driver** coordinates the execution of queries. It includes:

1. **Query Compiler**: Parses the query, validates it, and creates an execution plan.
2. **Executor**: Executes the execution plan and manages the query execution lifecycle.

c. Metastore:

- The **Metastore** is a central repository for Hive's metadata, including: **Table definitions, Partition information, Schema details**
- It interacts with external databases to store this metadata (e.g., MySQL or Derby).

d. Hadoop Integration:

- **JobTracker** and **TaskTracker**: Components of the Hadoop MapReduce framework that manage job scheduling and task execution.
- **HDFS (Hadoop Distributed File System)**: The underlying file storage system where data is stored and accessed.

e. Hive's Interaction with Hadoop:

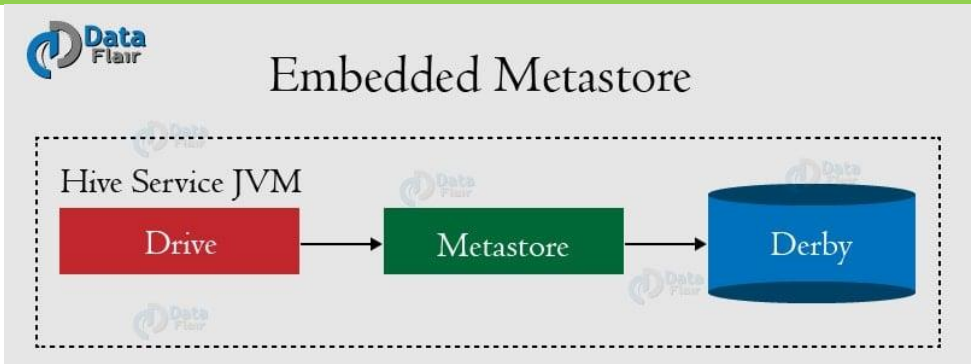
- Hive queries (written in HiveQL) are translated into MapReduce jobs, executed on Hadoop, and data is fetched from HDFS.

Metastore

- **Metastore** is the central repository of Apache Hive metadata. It stores metadata for Hive tables (like their schema and location) and **partitions** in a relational database. It provides client access to this information by using metastore service API.
- Hive metastore consists of two fundamental units:
  1. A service that provides metastore access to other Apache Hive services.
  2. Disk storage for the Hive metadata which is separate from **HDFS** storage.
- Hive Metastore Modes

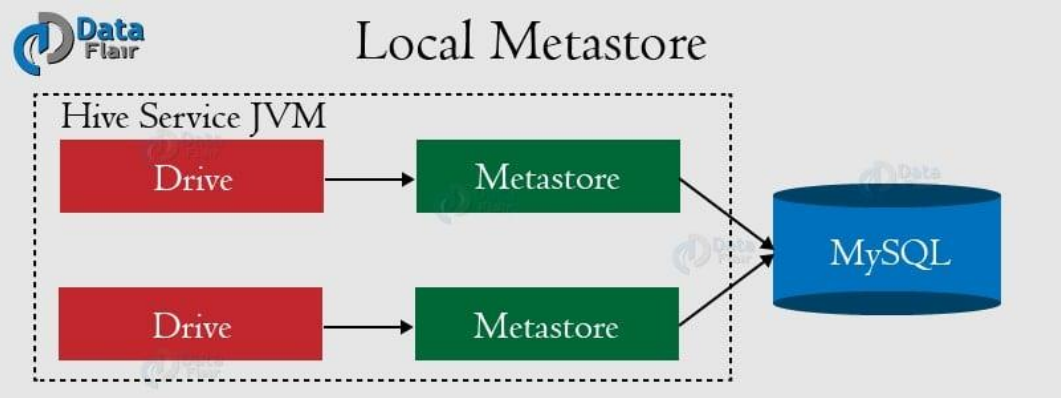
  There are three modes for Hive Metastore deployment:
  1. **Embedded Metastore**
  2. **Local Metastore**
  3. **Remote Metastore**

1. Embedded Metastore



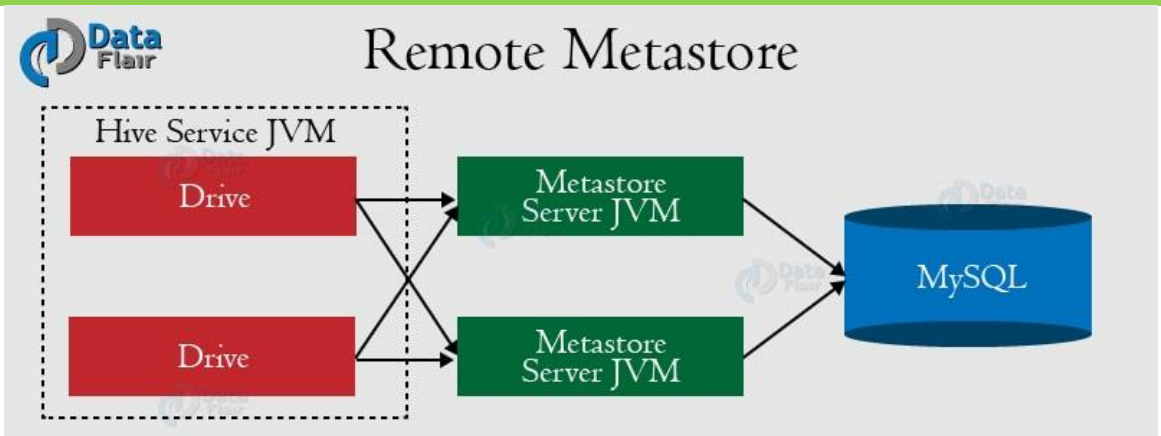
- **Description**: The Hive metastore runs in the same JVM (Java Virtual Machine) process as the Hive service.
- **Metadata Storage**: Metadata is stored in a **derby database** (by default).
- **Usage**: Typically used for **development or testing** purposes.
- **Advantages**:
  - Simple to set up.
  - No external database or service required.
- **Disadvantages**:
  - Limited scalability (not suitable for production environments).
  - Single-user access: Only one Hive session can access it at a time.

2. Local Metastore



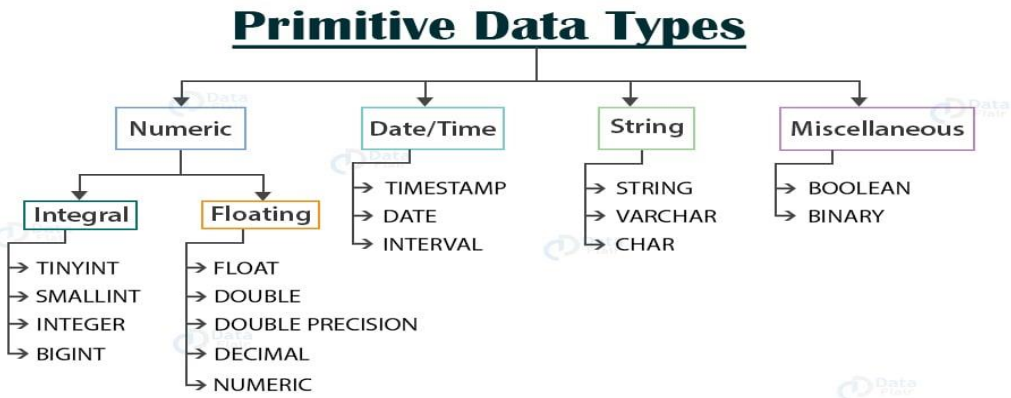
- **Description:** The Hive metastore service runs in the same process as the Hive service, but the metadata is stored in an external relational database (e.g., MySQL, PostgreSQL, etc.).
- **Metadata Storage:** Uses an external database for storing metadata, which allows for multi-session support.
- **Usage:** Suitable for small to medium-scale deployments.
- **Advantages:**
  - More robust than the embedded mode.
  - Allows multiple Hive sessions to share metadata.
- **Disadvantages:**
  - Metastore is still tightly coupled with Hive, which can limit scaling in larger deployments.

3. Remote Metastore



- **Description**: The Hive metastore is deployed as an **independent service**, running in its own JVM. Hive clients interact with the metastore over a network via the **Thrift protocol**.
- **Metadata Storage**: Metadata is stored in an external relational database, similar to the local metastore.
- **Usage**: Recommended for **production environments** with multiple Hive clients and heavy workloads.
- **Advantages**:
  - Decoupled architecture allows for scalability.
  - Multiple Hive services or clients can access the metastore concurrently.
  - Easier to manage and debug.
- **Disadvantages**:
  - Requires additional setup for the metastore service.

Hive 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

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')

Hive File Format

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

1. Text File

- In Hive, the **Text File format** is one of the simplest and most commonly used storage formats.
- The default file format is text file. Each record is a line in the file.
- In text file, different control characters are used as delimiters
  - **^A(octal 001, separates fields)**
  - **^B(octal 002, separates elements in array/struct)**
  - **^C(octal 003, separates key-value pairs).**
  - **\n record delimiter.**
- The term **field** is used when overriding the default delimiter. Formats supported csv, tsv, XML, JSON.

Example:

Table Schema:

```
name        STRING
salary      FLOAT
subordinates ARRAY<STRING>
deductions   MAP<STRING, FLOAT>
address      STRUCT<street: STRING, city: STRING, state: STRING, zip: INT>
```

Text File Content:

John Doe^A100000.0^AMary Smith^BTodd Jones^AFederal axes^C.2^BState Taxes^C.05^BInsurance^C.1^A1 Michigan Ave.^BChicago^BIL^B60600

Explanation:

- Field 1 (`name`): `"John Doe"`
- Field 2 (`salary`): `100000.0`
- Field 3 (`subordinates`): `"Mary Smith"` and `"Todd Jones"`
  - Separated by `^B` .
- Field 4 (`deductions`):
  - `"Federal Taxes": 0.2` , `"State Taxes": 0.05` , `"Insurance": 0.1`
  - Keys and values separated by `^C` , pairs separated by `^B` .
- Field 5 (`address`):
  - `"1 Michigan Ave."` , `"Chicago"` , `"IL"` , `60600`
  - Components separated by `^B` .

2. Sequential File

- In Hive, a **sequential file format** refers to a plain text file where data is stored line by line, with fields in each row separated by a delimiter (like commas, tabs, or pipes).
- Sequential files are often used as the default format for simple data storage and processing.
- Here's how a sequential file would look for the given schema.

Example: Given Schema:

```
CREATE TABLE employee (
   name STRING,
   salary FLOAT,
   subordinates ARRAY<STRING>,
   deductions MAP<STRING, FLOAT>,
   address STRUCT<street: STRING, city: STRING, state: STRING, zip: INT>
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
COLLECTION ITEMS TERMINATED BY ','
MAP KEYS TERMINATED BY ':'
LINES TERMINATED BY '\n';
```

Sequential File Content:

John Doe|75000.50|["Alice", "Bob"]|{"Tax": 5000.0, "Insurance": 2000.0}|{"street": "123 Elm St", "city": "Metropolis", "state": "NY", "zip": 10001}
Jane Smith|82000.75|["Eve", "Charlie"]|{"Tax": 6000.0, "Insurance": 3000.0}|{"street": "456 Oak St", "city": "Gotham", "state": "NJ", "zip": 90210}

Explanation:

1. Name: A string value (e.g., John Doe).
2. Salary: A floating-point number (e.g., 75000.50).
3. Subordinates: An array is serialized as a JSON-like list (e.g., ["Alice", "Bob"]).
4. Deductions: A map is serialized as a JSON-like object (e.g., {"Tax": 5000.0, "Insurance": 2000.0}).
5. Address: A struct is serialized as a nested JSON-like object (e.g., {"street": "123 Elm St", "city": "Metropolis", "state": "NY", "zip": 10001}).
6.

3. RCFile

- **RCFile (Record Columnar File)** is a columnar storage format used in Hadoop and Hive.
- It splits data into rows and stores them in a columnar manner, which improves read performance for analytical queries by allowing selective reading of only the required columns.
- In **RCFile,** the data is not directly human-readable, as it uses columnar encoding and compression for storage. However, we can describe how the data would look logically (conceptually) before it's encoded for RCFile storage.

Example:

Table Schema:

```
name        STRING
salary      FLOAT
subordinates ARRAY<STRING>
deductions  MAP<STRING, FLOAT>
address     STRUCT<street: STRING, city: STRING, state: STRING, zip: INT>
```

RCFile format:

Row-Wise Data (Raw Representation)

name	salary	subordinates	deductions	address
John Doe	75000.50	["Alice", "Bob"]	{"Tax": 5000, "Insurance": 2000}	{"street": "123 Elm St", "city": "Metropolis", "state": "NY", "zip": 10001}
Jane Smith	82000.75	["Eve", "Charlie"]	{"Tax": 6000, "Insurance": 3000}	{"street": "456 Oak St", "city": "Gotham", "state": "NJ", "zip": 90210}

Column 1 (name)
John Doe,
Jane Smith

Column 2 (salary)
75000.50,
82000.75

Column 3 (subordinates)
["Alice", "Bob"],
["Eve", "Charlie"]

Column 4 (deductions)
{"Tax": 5000, "Insurance": 2000},
{"Tax": 6000, "Insurance": 3000}

Column 5 (address)
{"street": "123 Elm St", "city": "Metropolis", "state": "NY", "zip": 10001},
{"street": "456 Oak St", "city": "Gotham", "state": "NJ", "zip": 90210}

Hive Query Language (HQL)

- **Hive Query Language (HiveQL)** is a query language used with Apache Hive, a SQL-like query system built on top of Hadoop.
- **HiveQL** provides a familiar SQL-like interface for querying and managing large datasets stored in Hadoop Distributed File System (HDFS) or other compatible storage systems.
- 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.

Hive DML 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	**2. Show**	**3. Describe**	**4. Use**
5. Drop	**6. Alter**	**7. Truncate**	

Hive DDL Statements

- Hive DML (Data Manipulation Language) statements are used to insert, update, retrieve, and delete data from the Hive table once the table and database schema has been defined using Hive DDL statements/commands.
- The various Hive DML commands are:

1. **LOAD**	2. **SELECT**	3. **INSERT**	4. **Use**
5. **UPDATE**	6. **EXPORT**	7. **IMPORT**	

Database-Create

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

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

- The **CREATE DATABASE** statement is used to create a database in the Hive.
- The DATABASE and SCHEMA are interchangeable.
- We can use either DATABASE or SCHEMA.

Syntax:

```
CREATE DATABASE [IF NOT EXISTS] database_name
[COMMENT 'database_comment']
[LOCATION 'hdfs_path']
[WITH DBPROPERTIES (property_name=property_value, ...)];
```

Description of Options

1. **IF NOT EXISTS**: Ensures the database is created only if it doesn't already exist.
2. **COMMENT**: Adds a description of the database.
3. **LOCATION**: Specifies the HDFS directory where the database data will be stored.
4. **WITH DBPROPERTIES**: Allows you to specify properties for the database.

Example:

```
CREATE DATABASE IF NOT EXISTS STUDENTS
COMMENT ' STUDENT Details'
LOCATION '/user/hive/warehouse/students.db'
WITH DBPROPERTIES ('creator'='admin', 'created_date'='2024-12-05');
```

Database-Show

Objective: To Display List of all Databases in Hive

- **SHOW** command is used to display all the databases in the metastore.

Syntax-1:

```
SHOW DATABSES;
```

Description:

- Displays all the databases in the metastore.

Syntax-2:

```
SHOW DATABASES LIKE 'pattern';
```

Description:

- **pattern**: A string or regular expression to match database names.
- **For example: 'my%'** matches all databases starting with **my.**
  '**test_db**' matches a specific database named **test_db.**

Suppose the following databases exist:
User_data
User_profile
Usage_stats
default

Example-1

```
SHOW DATABASES
```

Output:

User_data
User_profile
Usage_stats
default

Example-2

```
SHOW DATABASES LIKE 'user%';
```

Output:

User_data
User_profile

Database-Describe

The **DESCRIBE DATABASE** command in Hive is used to retrieve metadata about a specific database, such as its location, owner, and properties.

Syntax-1

```
DESCRIBE DATABASE [EXTENDED] database_name;
```

Description:

- **database_name**: The name of the database you want to describe.
- **EXTENDED**: Provides additional information about the database, including its properties.

Example-1

```
DESCRIBE DATABASE STUDENTS;
Output:
Database Name  : STUDENTS
Comment        : ' STUDENT Details'
Location       : hdfs://namenode/user/hive/warehouse/students.db
OwnerName      : admin
OwnerType      : USER
```

Syntax-2

```
DESCRIBE DATABASE [EXTENDED] database_name;
```

Description:

- **database_name**: The name of the database you want to describe.
- **EXTENDED**: Provides additional information about the database, including its properties.

Example-2

```
DESCRIBE DATABASE EXTENDED STUDENTS;
Output:
Database Name  : STUDENTS
Comment        : ' STUDENT Details'
Location       : hdfs://namenode/user/hive/warehouse/students.db
OwnerName      : admin
OwnerType      : USER
Properties : {created_by=admin, created_date=2024-12-05}
```

Note:

- Use the EXTENDED keyword to include properties specified during database creation (WITH DBPROPERTIES).

Database-Alter

The **ALTER DATABASE** command in Hive is used to modify the properties of an existing database. This is useful for updating metadata, such as ownership details, comments, or custom properties, without recreating the database.

Syntax

```
ALTER DATABASE database_name
SET DBPROPERTIES ('property_name'='property_value', ...);
```

Description:

- **database_name**: Name of the database you want to modify.
- **DBPROPERTIES**: A key-value pair to set or update metadata about the database.

Example

```
ALTER DATABASE STUDENTS
SET DBPROPERTIES ('owner'='new_admin', 'last_modified'='2024-12-05');
```

Note:

- This adds or updates properties for the STUDENT database
- You cannot directly alter the location of an existing database in Hive. Instead:
  - **Drop and recreate the database with a new location** (not ideal for databases with existing tables).
  - Or, manually update the location of individual tables within the database.
- To confirm the property changes, ***Describe The Database.***

Database-Use

- The **USE command** in Hive is used to switch the current working database to a specified one.
- Once the database is selected, all subsequent queries will operate within that database unless another database is specified.

Syntax

```
USE database_name;
```

Example

```
USE STUDENT;
```

Description:

- Here STUDENT database is selected, all subsequent queries will operate within STUDENT database unless another database is specified.

To confirm the current working database:

```
SELECT current_database();
Output:
STUDENT
```

Database-Drop

- The **DROP DATABASE** command in Hive is used to delete an existing database.
- Depending on the parameters you use, you can also remove all tables and data within the database.

Syntax

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

Description:

- **IF EXISTS:** Avoids an error if the database does not exist.
- **database_name:** The name of the database to drop.
- **RESTRICT (default):** Prevents the database from being dropped if it contains any tables.
- **CASCADE:** Deletes the database along with all its tables and associated data.

Example-1. Drop an Empty Database

```
DROP DATABASE my_database;
```

Description:
➢The command will succeed only if the database is empty.

Example- 2. Drop a Database (Avoid Error If Nonexistent)

```
DROP DATABASE IF EXISTS my_database;
```

Description:
➢Prevents an error if my_database does not exist.

Example- 3. Drop a Database and Its Tables

```
DROP DATABASE my_database CASCADE;
```

Description:
➢Deletes my_database along with all its tables and data.

Tables

- Hive provides 2 types of tables:
  1. Managed Table (Internal Table)
  2. External Table
- The difference is , when you drop a table,
  1. if it is managed table hive deletes both data and metadata,
  2. if it is external table Hive only deletes metadata

1. Managed Table (Internal Table)

- In Managed Tables, Hive manages both the table's metadata and the data stored in it.
- When the table is dropped, the data stored in HDFS is also deleted.
- Data is typically stored in the Hive warehouse directory **(/user/hive/warehouse**) by default, unless explicitly specified otherwise.

Key Characteristics:

- Hive has full control over the table and its data.
- Deleting the table removes the data permanently from HDFS.

Syntax To Create Managed Table

```
create table managedemp [IF NOT EXISTS]
(col1 datatype, col2 datatype, ....)
 row format delimited fields terminated by 'delimiter character'
location '/data/employee'
```

Example To Create Managed Table named STUDENT

```
CREATE TABLE IF NOT EXISTS STUDENT (
   rollno INT,
   name STRING,
   cgpa FLOAT
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
LOCATION '/user/hive/warehouse/students.db'
```

Example To show the structure of the STUDENT table.

```
DESCRIBE STUDENT;
Output:
rollno      int
name        string
cgpa        float
```

Example To check whether existing table is managed or external table

```
DESCRIBE FORMATTED STUDENT;
Output:
Table Type:        MANAGED_TABLE
```

2. External Table

- In External Tables, Hive manages only the table's metadata; the actual data resides at a specified location in HDFS or another file system.
- Dropping the table does not delete the data; only the table's metadata is removed from Hive.

Key Characteristics:

- Useful when the data is shared between multiple applications or needs to remain persistent.
- Allows you to work with data without moving it into Hive's default location.

Example To Create Managed Table named STUDENT

```
CREATE TABLE EXTERNAL IF NOT EXISTS STUDENT (
   rollno INT,
   name STRING,
   cgpa FLOAT
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
LOCATION '/user/hive/warehouse/students.db'
```

Example To check whether existing table is managed or external table

```
DESCRIBE FORMATTED STUDENT;
Output:
Table Type:        EXTERNAL_TABLE
```

Internal Vs External Table

Feature	Managed Table	External Table
Data Management	Hive manages data and metadata.	Hive manages only metadata.
Data Location	Stored in Hive warehouse directory by default.	Stored at user-specified location.
Data Deletion	Dropping the table deletes data and metadata.	Dropping the table deletes only metadata, not data.
Use Case	When Hive exclusively manages data.	When data is shared or pre-existing.

Load Data into Tables from file

- To load data from a local file into a Hive table using the LOAD DATA command.

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

- **Local** is used to load data from the local file system .
- To load the data from HDFS remove local key word.
- **LOCAL**: Indicates that the file is located on the local file system (as opposed to HDFS).
- **INPATH**: Specifies the path to the source file. In this case, the file is student.tsv located at /root/hivedemos/.
- **OVERWRITE**: This keyword will overwrite any existing data in the target table (EXT_STUDENT) with the new data. If omitted, the new data will be appended to the table.
- **INTO TABLE EXT_STUDENT**: Specifies the target table where the data should be loaded
- To retrieve the student details from "**EXT_STUDENT**" table.

```
SELECT * from EXT_STUDENT;
```

Querying Table

Objective: To retrieve student details from a table named EXT_STUDENT

Input Data:

The input consists of student records in a semicolon-separated format:

```
1001,John,Smith:Jones,Mark1!45:Mark2!46:Mark3!43
1002,Jack,Smith:Jones,Mark1!46:Mark2!47:Mark3!42
```

Create Table Statement:

```
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 '!';
```

Explanation:

- **rollno:** Integer field for roll number.
- **name:** String field for the student's name.
- **sub:** An array of subjects (ARRAY<STRING>).
- **marks:** A map where the key is a string (subject code) and the value is an integer (marks scored).
- **FIELDS TERMINATED BY ',':** Indicates that fields in the input are separated by commas.
- **COLLECTION ITEMS TERMINATED BY ':':** Specifies that elements within arrays or maps are separated by colons.
- **MAP KEYS TERMINATED BY '!':** Indicates that map key-value pairs are separated by an exclamation mark.

Load Data Statement:

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

This command loads the local file **studentinfo.csv** into the **STUDENT_INFO** table in Hive.

Collection Data Types

Objective:

1. To retrieve student details from a table named EXT_STUDENT

```
SELECT * FROM EXT_STUDENT;
```

Objective:

2. To query collection data types from a Hive table named STUDENT_INFO

```
// Retrieves all rows and columns from the STUDENT_INFO table.
SELECT * FROM STUDENT_INFO;

// Retrieves the NAME (student name) and SUB (subjects array) columns
SELECT NAME, SUB FROM STUDENT_INFO;

// Retrieves the NAME and the value of the Mark1 key from the MARKS map column
SELECT NAME, MARKS['Mark1'] FROM STUDENT_INFO;

// Retrieves the NAME and the first element of the SUB array
SELECT NAME, SUB[0] FROM STUDENT_INFO;
```

Views

A **view** in Hive is a logical structure that provides a way to simplify and reuse queries. Views do not store data themselves; instead, they store the SQL query definition.

Creating a View

Syntax

```
CREATE VIEW view_name AS SELECT ...;
```

Example

```
CREATE VIEW student_view AS
SELECT rollno, name, marks['Mark1'] AS mark1 FROM STUDENT_INFO;
```

Explanation:

- **student_view** is the view.
- The query selects **rollno, name, and Mark1** marks from the **STUDENT_INFO** table.

Querying a View

```
SELECT * FROM student_view; // Retrieves all rows and columns defined in the student_view.
```

Dropping a View

Syntax

```
DROP VIEW view_name;
```

Example

```
DROP VIEW student_view; //Deletes the
student_view.
```

Sub Query

A **sub query** in Hive is a query nested inside another query. It can be used in the SELECT, FROM, or WHERE clauses to perform operations like filtering, aggregation, and intermediate data.

Objective: Count Occurrence of Similar Words in a File

```
//Example Data
Hive is a data warehouse.
Hive supports SQL-like queries.
Hive is fast.
```

```
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(*) AS count FROM
(SELECT explode (split (line, ' ')) AS word FROM docs) w
GROUP BY word
ORDER BY word;
SELECT * FROM word_count;
```

Explanation:

- **split(line, ' '):** Splits each line into individual words based on spaces.
- **explode:** Converts the resulting array of words into multiple rows (one row per word).
- **GROUP BY word:** Groups the rows by each unique word and counts their occurrences.
- **ORDER BY count DESC:** Orders the result in descending order by word frequency.

Output

word	count
Hive	3
is	2
a	1
data	1
warehouse	1
supports	1
SQL-like	1
queries	1
fast	1

Joins

- Joins in Hive work similarly to SQL joins, allowing you to combine data from two or more tables based on a related column.
- The example provided demonstrates how to join STUDENT and DEPARTMENT tables using rollno as the join key.

Objective: To create a JOIN between the STUDENT and DEPARTMENT tables based on the rollno column.

Solution:

1. Create the Tables:

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

2. Load Data into Tables:

```
//Load into Student Table
LOAD DATA LOCAL INPATH '/root/hivedemos/student.tsv'
OVERWRITE INTO TABLE STUDENT;

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

3. Perform the Join:

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

Output

STUDENT Table (student.tsv):

```
101  John   3.8
102  Alice  3.5
103  Bob    3.7
```

DEPARTMENT Table (department.tsv):

```
101 10  ComputerScience
102 20  Mathematics
104 30  Physics
```

The result of the query would be:

```
rollno  name   gpa  deptno
-------------------------
101     John   3.8  10
102     Alice  3.5  20
```

Only matching rollno values (101 and 102) appear in the output.

Aggregation

Aggregation functions in Hive, like AVG and COUNT, allow you to compute summary statistics on datasets.

Objective: To write the average and count aggregation functions.

Solution:

1. Calculate Average GPA:

```
//avg(gpa): Computes the average of the gpa column in the STUDENT table.
SELECT avg (gpa) FROM STUDENT;
```

2. Count Total Records:

```
//count(*): Counts the total number of rows in the STUDENT table, including rows with NULL values.
SELECT count(*) FROM STUDENT;
```

Example Data (STUDENT Table):

rollno	name	gpa
101	John	3.8
102	Alice	3.5
103	Bob	3.7

Expected Output:

1. **Average GPA:** For the given data, the average GPA is:
   3.6667
2. **Total Count:** The count of rows in the STUDENT table:
   3

Group By and Having

- **The GROUP BY** clause in Hive allows you to group rows that have the same values in specified columns.
- The **HAVING** clause is used to filter the groups based on aggregate conditions.

Objective: To use GROUP BY to group rows and HAVING to filter groups based on a condition.

Query:

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

Explanation:

1. **GROUP BY**:
   - Groups rows with the same values in the **rollno, name,** and **gpa** columns.
   - It ensures that the output contains one row per unique combination of these three columns.
2. **HAVING**:
   - Filters out groups where the **gpa** value is **not greater than 4.0**.
   - Unlike the **WHERE** clause, which filters rows before grouping, **HAVING** filters after the grouping is complete.

Example Data (`STUDENT` Table):

rollno	name	gpa
101	John	3.8
102	Alice	4.5
103	Bob	4.2

Expected Output:

rollno	name	gpa
102	Alice	4.5
103	Bob	4.2

RCFile Implementation

- The **Record Columnar File (RCFile)** is a Hive storage format designed for efficient data storage and processing.
- It stores data in a columnar format, which makes reading specific columns faster and reduces I/O.
- **RCFile** is well-suited for analytical workloads.

Steps to Implement RCFile in Hive: 4 Steps

1. Create a Table with RCFile Format

```
CREATE TABLE student_rcfile (rollno INT, name STRING, gpa FLOAT)
STORED AS RCFILE;
```

Explanation:

- To use the RCFile format, specify **STORED AS RCFILE** when creating the table.
- The table student_rcfile will use the RCFile format for storage.
- **RCFile** splits the data into rows but organizes each row in column chunks to optimize read and write performance.

2. Load Data into the RCFile Table

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

Explanation:

- Data can be loaded into the RCFile table from a text file or another table.
- The **LOAD DATA** command adds data to the student_rcfile table.
- This step does not convert the format to RCFile; the input file should match the schema.

3. Insert Data into an RCFile Table

```
INSERT OVERWRITE TABLE student_rcfile
SELECT * FROM student;
```

Explanation:

- If you want to convert existing data into RCFile format, use the INSERT INTO or INSERT OVERWRITE command.
- Data is selected from a source table (student) and written into the RCFile table.
- The data is converted to RCFile format during the write process.

4. Query the RCFile Table

```
SELECT rollno, name, gpa FROM student_rcfile WHERE gpa > 4;
```

Explanation:

- Querying an RCFile table is identical to querying any other table in Hive.

Example Data (`STUDENT` Table):

rollno	name	gpa
101	John	3.8
102	Alice	4.5
103	Bob	4.2

Expected Output:

rollno	name	gpa
102	Alice	4.5
103	Bob	4.2

SerDe

- **SerDe** stands for Serializer/Deserializer in Hive.
- It is a framework that allows Hive to process data in various formats by providing the logic to read (deserialize) data from storage into Hive table rows and write (serialize) data from Hive into storage.
- **SerDe**
  1. Contains the logic to convert unstructured data into records.
  2. Implemented using Java.
  3. Serializers are used at the time of writing.
  4. Deserializers are used at query time (SELECT Statement).
- Deserializer interface takes a binary representation or string of a record, converts it into a java object that Hive can then manipulate.
- Serializer takes a java object that Hive has been working with and translates it into something that Hive can write to HDFS.

Steps to Work with XML Data Using SerDe in Hive

Objective: To manipulate XML Data

1. **Create a Hive Table for XML Data**: First, you need to create a Hive table to store XML data. You specify the column as a STRING type to store the raw XML data.

```
CREATE TABLE XMLSAMPLE (xmldata STRING);//This table will store the XML data in the xmldata column.
```

2. **Load Data into the Table**: After creating the table, you load the XML file into the table. The file should be in a path accessible to Hive.

```
LOAD DATA LOCAL INPATH '/root/hivedemos/input.xml' INTO TABLE XMLSAMPLE;
// This loads the XML file input.xml from the local filesystem into the XMLSAMPLE table.
//The XML data is stored as a string in the xmldata column.
```

3. **Extract Data Using XPath Functions**: You can use Hive's built-in xpath_* functions to extract values from XML data using XPath expressions. These functions allow you to query specific elements within the XML.

```
CREATE TABLE xpath_table AS
SELECT
  xpath_int(xmldata, 'employee/empid') AS empid,
  xpath_string(xmldata, 'employee/name') AS name,
  xpath_string(xmldata, 'employee/designation') AS designation
FROM xmlsample;
```

Explanation:

- The **xpath_int()** function extracts an integer value from the XML string using the **XPath** expression.
- The **xpath_string()** function extracts string values (such as the name and designation fields) from the XML.
- The result is stored in the xpath_table table with the empid, name, and designation columns.

4. **Query the Extracted Data**: Finally, you can query the newly created table to view the extracted data.

```
SELECT * FROM xpath_table;
// This query fetches the empid, name, and designation columns from the xpath_table.
```

Example XML Data (`input.xml`):

Assume you have an XML file like this:

```
<employee>
    <empid>101</empid>
    <name>John Doe</name>
    <designation>Software Engineer</designation>
</employee>
```

Expected Output for the Given Example XML:

empid	name	designation
101	John Doe	Software Engineer

Explanation:

In this case,

- t he **xpath_int(xmldata, 'employee/empid')** will return **101**, and
- **x path_string(xmldata, 'employee/name')** will **return 'John Doe'.**

Functions Used in the Query:

1. **xpath_int(xmldata, 'xpath_expression')**:
   - This function extracts an integer value from the XML data.
   - Example: xpath_int(xmldata, 'employee/empid') extracts the employee ID.
2. **xpath_string(xmldata, 'xpath_expression')**:
   - This function extracts a string value from the XML data.
   - Example: xpath_string(xmldata, 'employee/name') extracts the employee name.

User Defined Function-UDF

- In Apache Hive, **User Defined Functions (UDFs)** allow you to extend Hive's functionality by writing custom functions for processing and transforming data.
- UDFs can be written in Java, Python (using Hive's Streaming API), or other languages supported by Hive.

Steps to Create a UDF in Hive

Objective: UDF that converts a string to uppercase.

1. Create the UDF in Java

```
import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;

public class ToUpperCaseUDF extends UDF {
   public Text evaluate(Text input) {
      if (input == null) {
         return null;
      }
      return new Text(input.toString().toUpperCase());
   }
}
```

Explanation:

- **evaluate Method**: This is the main method that Hive calls to process data.
- **Input**: It accepts and processes the input (e.g., a string in this case).
- **Output**: Returns the transformed result.

2. Compile the Java Code

```
javac -cp "$(hive --auxpath)" ToUpperCaseUDF.java
jar -cf ToUpperCaseUDF.jar ToUpperCaseUDF.class
```

Explanation:
- Save the file as **ToUpperCaseUDF.java.**
- Compile it into **a .jar** file using the following commands

3. Add the UDF JAR to Hive- Upload the JAR file to a location accessible to Hive (e.g., HDFS or local path).

```
ADD JAR /path/to/ToUpperCaseUDF.jar;
```

4. Register the UDF-You must register the UDF in Hive before using it.

```
CREATE TEMPORARY FUNCTION to_upper_case AS 'ToUpperCaseUDF';
```

Explanation:
- **to_upper_case**: Name of the UDF you'll use in queries.
- **ToUpperCaseUDF**: Fully qualified class name of your Java UDF.

5. Use the UDF in Hive Queries-You can now use your UDF in HiveQL.

```
SELECT to_upper_case(column_name) AS uppercased_column
FROM your_table;
```

Example: Testing the UDF

Assume a table `employees` with the following structure:

name
alice
bob

Run the query:

```
SELECT to_upper_case(name) AS upper_name FROM employees;
```

Result:

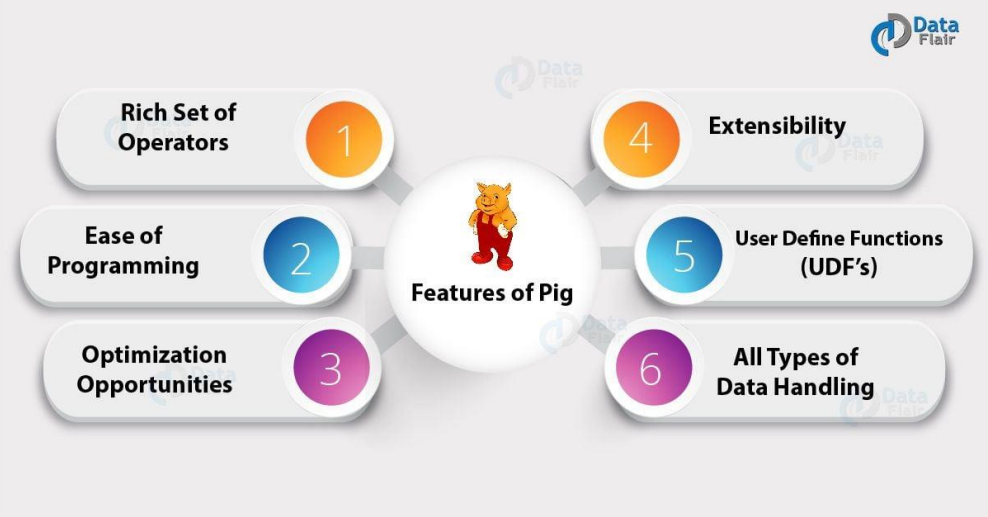
upper_name
ALICE
BOB

Introduction to Pig

- Apache Pig Represents Big Data as data flows. Pig is a high-level platform or tool which is used to process the large datasets. i.e., Apache Pig is a platform used for data analysis. It provides a high-level of abstraction for processing over the MapReduce. It provides a high-level scripting language, known as ***Pig Latin*** which is used to develop the data analysis codes.
- It was originally developed internally by Yahoo! in 2006, with the aim of creating and executing MapReduce jobs on all datasets and later integrated into the Apache Software Foundation.

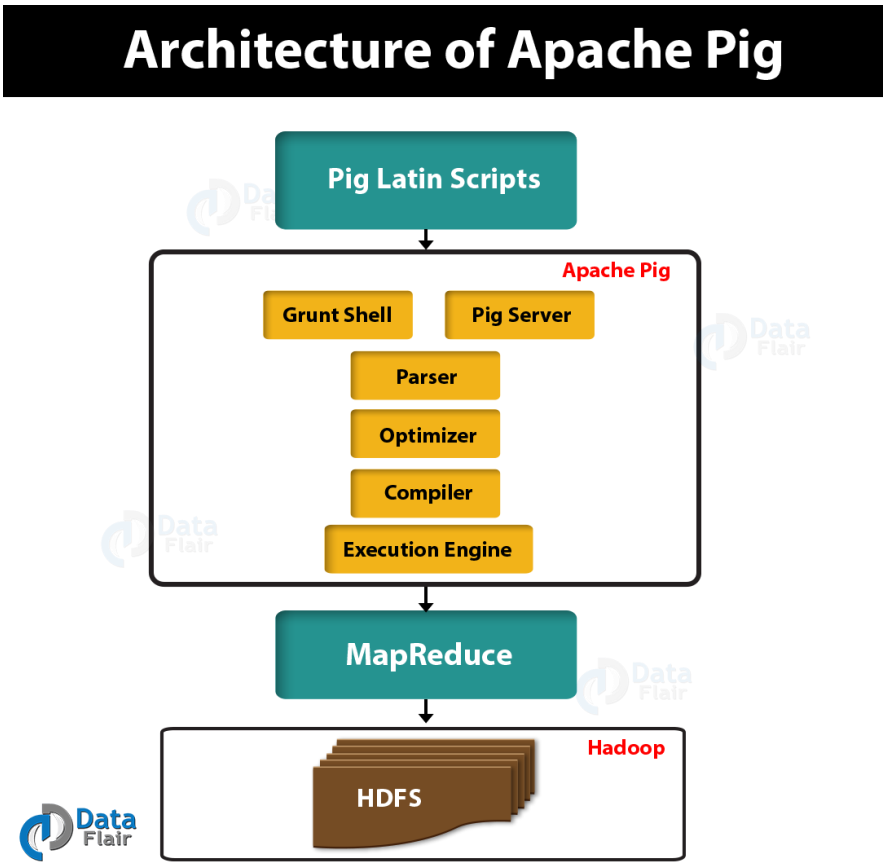
Features of PIG

Apache Pig comes with the below unique features:



- **Rich Set of Operators:** Pig consists of a collection of rich set of operators in order to perform operations such as join, filer, sort and many more.
- **Ease of Programming: Pig Latin** is similar to SQL and hence it becomes very easy for developers to write a Pig script. If you have knowledge of SQL language, then it is very easy to learn Pig Latin language as it is similar to SQL language.
- **Optimization opportunities:** The execution of the task in Apache Pig gets automatically optimized by the task itself, hence the programmers need to only focus on the semantics of the language.
- **Extensibility:** By using the existing operators, users can easily develop their own functions to read, process, and write data.
- **User Define Functions (UDF's)**: With the help of facility provided by Pig of creating UDF's, we can easily create User Defined Functions on a number of programming languages such as Java and invoke or embed them in Pig Scripts.
- **All types of data handling:** Analysis of all types of Data (i.e. both structured as well as unstructured) is provided by Apache Pig and the results are stored inside HDFS.

Anatomy of PIG



There are several components in the Apache Pig framework. Let's study these major components in detail:

1. Data Flow Language (Pig Latin): Pig Latin is the scripting language used in Apache Pig to express data transformations. It includes commands for loading, filtering, transforming, and storing data.

2. Apache Pig Core:

a) **Grunt Shell:** The interactive shell for writing and executing Pig Latin scripts in real-time.

b) **Pig Server:** Handles the backend processing, receiving scripts from the Grunt shell or other interfaces.

c) **Parser:** Converts the Pig Latin script into a logical plan and checks for syntax or semantic errors.

d) **Optimizer:** Refines the logical plan to create an optimized execution plan, improving efficiency.

e) **Compiler:** Translates the optimized execution plan into a physical plan of MapReduce jobs.

f) **Execution Engine:** Executes the compiled plan by launching MapReduce jobs.

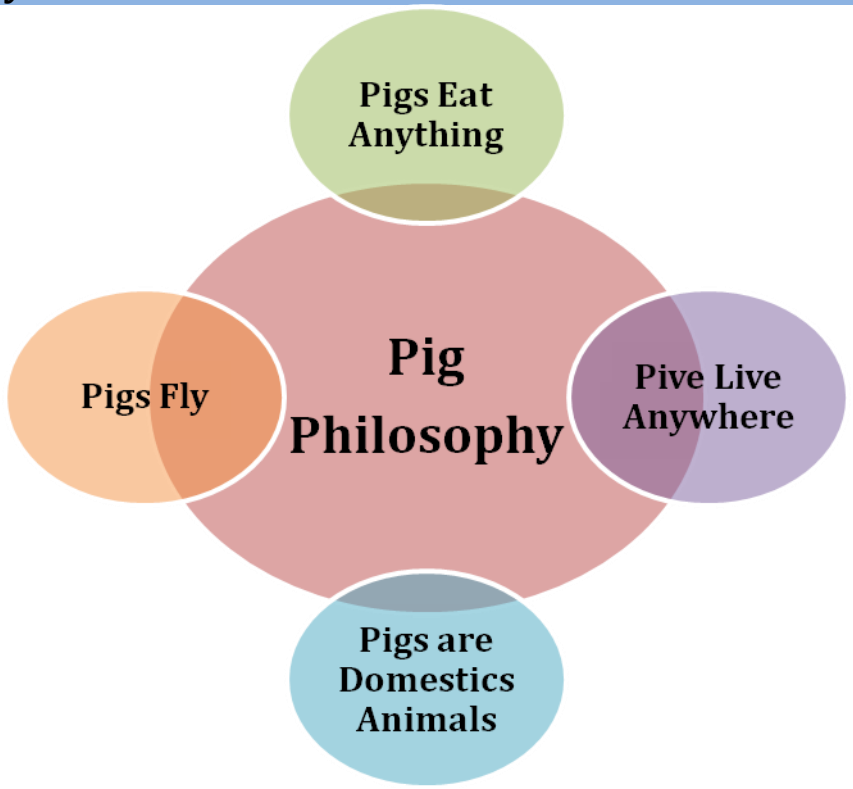
3. MapReduce Layer:

- Pig translates its execution plans into **MapReduce jobs** that run on Hadoop's distributed framework.

4. HDFS (Hadoop Distributed File System):

- Serves as the storage layer for input, intermediate, and output data during the execution of Pig jobs.

PIG Philosophy



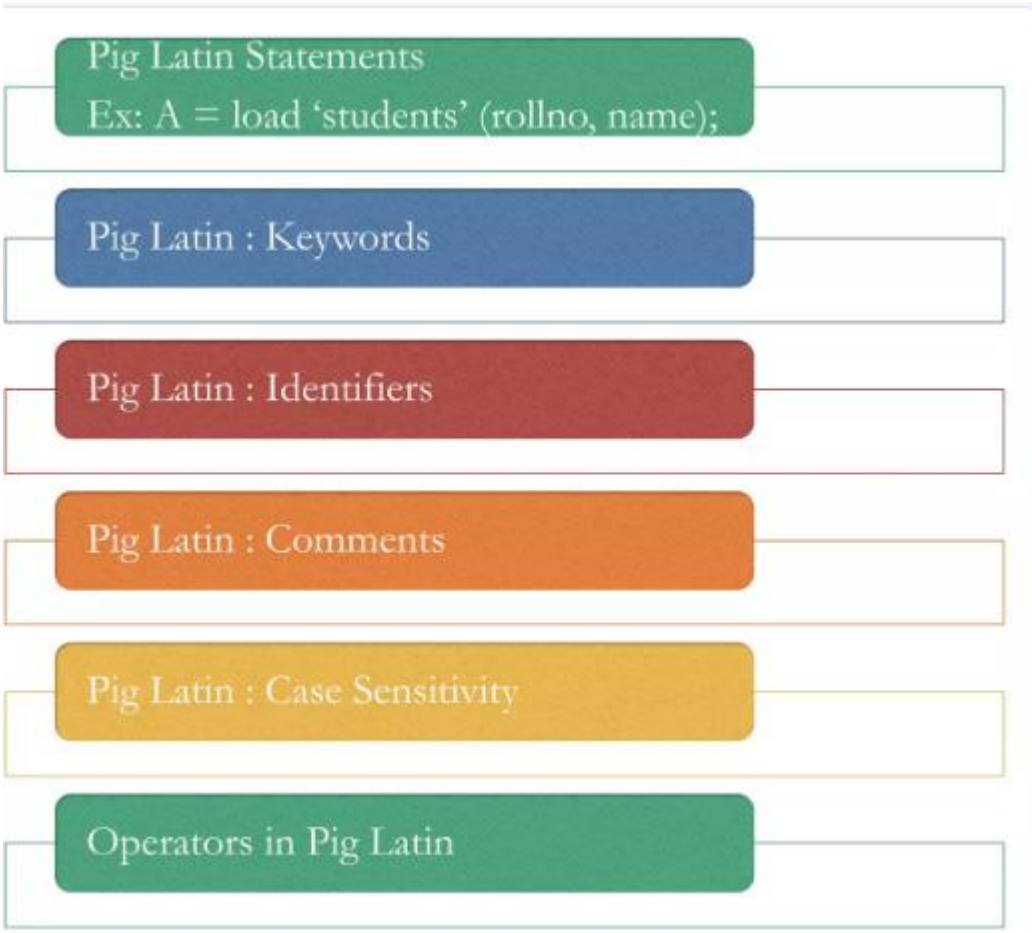
This diagram, illustrates the guiding philosophy behind Apache Pig by likening its principles to characteristics of actual pigs:

1. **"Pigs fly"**:
   - Suggests that Apache Pig is highly versatile and capable of solving seemingly impossible tasks in big data processing.
2. **"Pigs eat anything"**:
   - Highlights that Apache Pig can handle data in diverse formats (structured, semi-structured, or unstructured) and process it seamlessly.
3. **"Pigs are domestic animals"**:
   - Implies that Apache Pig is user-friendly, approachable, and easy to integrate within the Hadoop ecosystem.
4. **"Pigs live anywhere"**:
   - Refers to Apache Pig's flexibility to run in various environments, such as local mode for testing or MapReduce mode for large-scale distributed processing.

This metaphor encapsulates Pig's adaptability, broad applicability, and usability in big data scenarios.

Pig Latin Overview

The Pig Latin is a data flow language used by Apache Pig to analyze the data in Hadoop. The basics of Pig Latin are statements, keywords, identifiers, comments, case sensitivity, and operators.



1. Pig Latin: Statements

- Pig Latin statements are basic constructs to process data using Pig.
- Pig Latin statement is an operator.
- 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.
- 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.

Example: 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 = for each A generate UPPER (name);
STORE A INTO 'myreport'
```

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

2. Pig Latin: Keywords

- Reserved words in Pig Latin used for commands and operations.
- Examples include LOAD, FILTER, GROUP, FOREACH, STORE.

3. Pig Latin: Identifiers

- Identifiers are the names given to variables, relations, or aliases in a script.
- They start with a letter and can be followed by any number of letters, digits, or underscores.

Example:

Table 10.1 Valid and invalid identifiers

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

4. Pig Latin: Comments

- Pig Latin supports two types of comments:
  - **Single-line comments**: The single-line comments will begin with '--'.
  - **Multi-line comments:** The multi-line comments will begin with '/*', end them with '*/'.

Example-1:

A = load 'foo' ; ***--this is a single-line comment***

Example-2:

```
/*
This is a multi-line comment explaining
the purpose of the following Pig statement.

*/
```

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.

6. Operators in Pig Latin

Arithmetic Operators: The following table describes the arithmetic operators of Pig Latin. Suppose a = 10 and b = 20.

Operator	Description	Example
+	**Addition** – Adds values on either side of the operator	a + b will give 30
–	**Subtraction** – Subtracts right hand operand from left hand operand	a – b will give –10
*	**Multiplication** – Multiplies values on either side of the operator	a * b will give 200
/	**Division** – Divides left hand operand by right hand operand	b / a will give 2
%	**Modulus** – Divides left hand operand by right hand operand and returns remainder	b % a will give 0

Comparison Operators: The following table describes the comparison operators of Pig

Latin. Suppose a = 10 and b = 20.

Operator	Description	Example
==	**Equal** – Checks if the values of two operands are equal or not; if yes, then the condition becomes true.	(a = b) is not true
!=	**Not Equal** – Checks if the values of two operands are equal or not. If the values are not equal, then condition becomes true.	(a != b) is true.
>	**Greater than** – Checks if the value of the left operand is greater than the value of the right operand. If yes, then the condition becomes true.	(a > b) is not true.
<	**Less than** – Checks if the value of the left operand is less than the value of the right operand. If yes, then the condition becomes true.	(a < b) is true.
>=	**Greater than or equal to** – Checks if the value of the left operand is greater than or equal to the value of the right operand. If yes, then the condition becomes true.	(a >= b) is not true.
<=	**Less than or equal to** – Checks if the value of the left operand is less than or equal to the value of the right operand. If yes, then the condition becomes true.	(a <= b) is true.

Null Operators:

operator	Description
is null	**If the tested value is null, returns true; otherwise, returns false.**
is not null	**If the tested value is not null, returns true; otherwise, returns false.**

Boolean Operators:

Operator	Description
AND	Combines two conditions and evaluates to TRUE only if both conditions are true.
OR	Combines two conditions and evaluates to TRUE if at least one of the conditions is true.
IN	IN operator is equivalent to nested OR operators.
NOT	Negates a condition; evaluates to TRUE if the condition is false.

The result of a boolean expression (an expression that includes boolean and comparison operators) is always of type boolean (true or false).

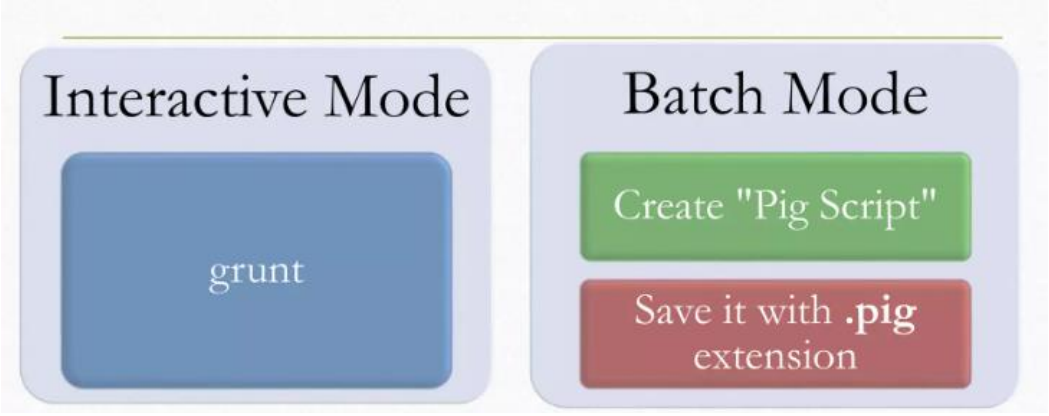
Data Types in Pig

Pig data type can be classified into two categories, and they are –**Simple and Complex.** Given below table describes the Pig Latin data types.

Simple Types		
S.N.	**Data Type**	**Description & Example**
1	int	Represents a signed 32-bit integer. **Example** : 8
2	long	Represents a signed 64-bit integer. **Example** : 5L
3	float	Represents a signed 32-bit floating point. **Example** : 5.5F
4	double	Represents a 64-bit floating point. **Example** : 10.5
5	chararray	Represents a character array (string) in Unicode UTF-8 format. **Example** : 'tutorials point'
6	Bytearray	Represents a Byte array (blob).
7	Boolean	Represents a Boolean value. **Example** : true/ false.
8	Datetime	Represents a date-time. **Example** : 1970-01-01T00:00:00.000+00:00
Complex Types		
11	Tuple	A tuple is an ordered set of fields. **Example** : (raja, 30)
12	Bag	A bag is a collection of tuples. **Example** : {(raju,30),(Mohhammad,45)}
13	Map	A Map is a set of key-value pairs. **Example** : ['name'#'Raju', 'age'#30]

Running Pig

The diagram illustrates two modes of running Apache Pig:



1. Interactive Mode

- **Description**: In this mode, commands are executed interactively through the **Grunt shell**, which is Pig's command-line interface.
- **How to Use**:
  1. Start the Grunt shell by typing pig in the terminal (if running in local or MapReduce mode).
  2. Execute Pig Latin commands one by one.

Example:

```
grunt> A = LOAD 'data.txt' USING PigStorage(',');
grunt> DUMP A;
```

2. Batch Mode

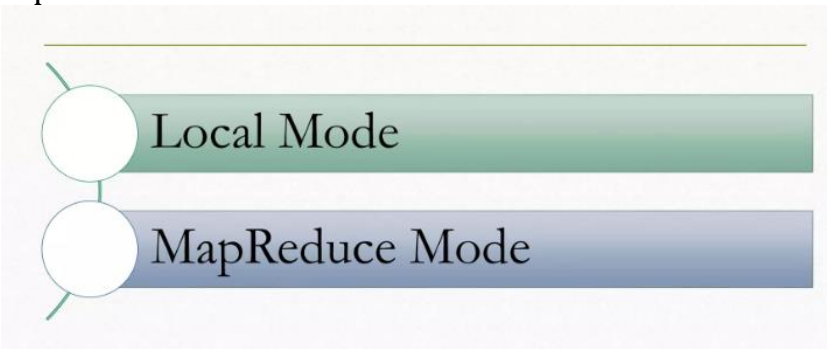
- **Description**: This mode allows you to write Pig scripts in a file and execute them as a batch process.
- **How to Use**:
  1. Write your Pig Latin commands in a script file with a .pig extension.
  2. Run the script using the pig command.
- **Example:**

```
bash

pig script.pig
```

```
pig

A = LOAD 'data.txt' USING PigStorage(',') AS (id:int, name:chararray);
B = FILTER A BY id > 100;
STORE B INTO 'output';
```

Execution modes of Pig

There are two modes in Apache Pig Execution, in which we can run Apache Pig such as, Local Mode and MapReduce mode.



1. Local Mode

- **Description**: Runs Pig in a local environment using the local file system (not Hadoop's HDFS).
- **Key Features**:
  - Suitable for small-scale data and testing.
  - Does not require a Hadoop cluster.
- **How to Execute**:
  - Start Pig in local mode by specifying the `-x local` option:

```bash
pig -x local
```

  - Example:

```
pig

A = LOAD 'data.txt' USING PigStorage(',') AS (field1:int, field2:int);
DUMP A;
```

2. MapReduce Mode

- **Description**: Executes Pig scripts on a Hadoop cluster using the MapReduce engine.
- **Key Features**:
  - Processes large-scale data stored in HDFS.
  - Requires a fully configured Hadoop environment.
- **How to Execute**:
  - Default mode when running Pig:

```bash
pig
```

- Example:

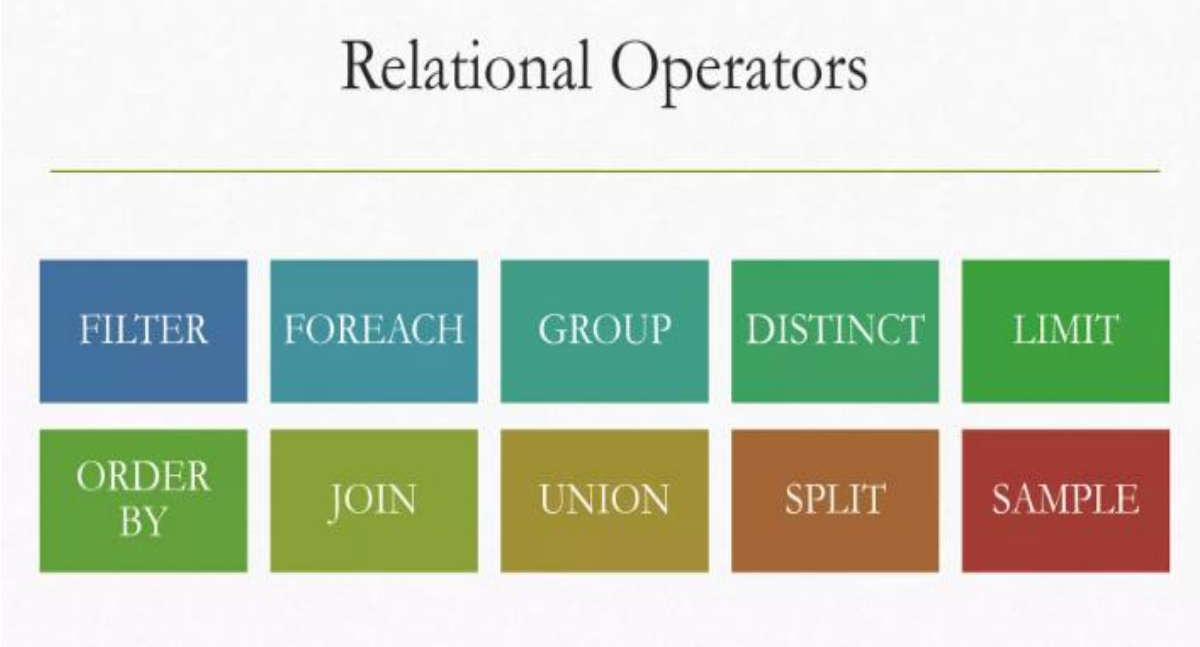
```
pig

A = LOAD 'hdfs://data/input.txt' USING PigStorage(',') AS (field1:int, field2:int);
STORE A INTO 'hdfs://data/output';
```

Local Mode Vs MapReduce Mode

Feature	Local Mode	MapReduce Mode
Data Source	Local file system	HDFS
Scale	Small-scale data	Large-scale data
Cluster Required	No	Yes
Execution Engine	Local execution	MapReduce
Execution Command	`pig -x local`	`pig` (default mode)

Relational Operations

In Apache Pig, **relational operators** are used to perform various data transformations, such as loading, filtering, grouping, joining, and sorting data. These operators work with relations (datasets) and form the backbone of Pig Latin scripts.



1. FILTER

The **FILTER** operator is used to select the required tuples from a relation based on a condition.

Syntax:

Relation2_name = FILTER Relation1_name BY (condition);

Example:

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;
```

Expected Output

Assume the content of `student.tsv` is as follows:

```
1    Alice    3.8
2    Bob      4.2
3    Charlie  4.5
4    David    3.9
```

After running the script, the output will be:

```
(2, Bob, 4.2)
(3, Charlie, 4.5)
```

2. FOREACH

The **FOREACH** operator is used to generate specified data transformations based on the column data.

Syntax:

Relation2_name = FOREACH Relatin_name1 GENERATE (required data);

Example:

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;
```

Expected Output

Assume the content of `student.tsv` is as follows:

```
1    Alice    3.8
2    Bob      4.2
3    Charlie  4.5
4    David    3.9
```

After running the script, the output will be:

```
(ALICE)
(BOB)
(CHARLIE)
(DAVID)
```

3. GROUP

The **GROUP** operator is used to group the data in one or more relations. It collects the data having the same key.

Syntax:

Group_data = GROUP Relation_name BY age;

Example:

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;
```

Expected Output

Assume the content of `student.tsv` is as follows:

```
1    Alice    3.8
2    Bob      4.2
3    Charlie  3.8
4    David    4.5
```

After running the script, the output will be:

```
(3.8,{(1, Alice, 3.8), (3, Charlie, 3.8)})
(4.2,{(2, Bob, 4.2)})
(4.5,{(4, David, 4.5)})
```

4. DISTINCT

The **DISTINCT** operator is used to remove redundant (duplicate) tuples from a relation.

Syntax:

Relation_name2 = DISTINCT Relatin_name1;

Example:

Objective: To remove duplicate tuples of students.

Input:

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

Act:

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

Expected Output

Assume the content of `student.tsv` is as follows:

```
1    Alice    3.8
2    Bob      4.2
1    Alice    3.8
3    Charlie  4.0
4    David    3.9
3    Charlie  4.0
```

After running the script, the output will be:

```
(1, Alice, 3.8)
(2, Bob, 4.2)
(3, Charlie, 4.0)
(4, David, 3.9)
```

5. LIMIT

The **LIMIT** operator is used to get a limited number of tuples from a relation.

Syntax:

Result = LIMIT Relation_name required number of tuples;

Example:

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;
```

Expected Output: First 3 rows will be displayed

6. ORDER BY

The **ORDER BY** operator is used to display the contents of a relation in a sorted order based on one or more fields.

Syntax:

Relation_name2 = ORDER Relatin_name1 BY (ASC|DESC);

Example:

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;
```

Expected Output:

let's assume the input file `student.tsv` contains the following data:

```
101    Alice    3.5
103    Bob      3.8
102    David    3.9
104    Charlie  3.6
```

After running the script, the **expected output** will display the records sorted by the `name` field in ascending order:

```
(101, Alice, 3.5)
(103, Bob, 3.8)
(102, Charlie, 3.6)
(104, David, 3.9)
```

7. JOIN

- The **JOIN** operator is used to combine records from two or more relations. While performing a join operation, we declare one (or a group of) tuple(s) from each relation, as keys. When these keys match, the two particular tuples are matched, else the records are dropped.
- Joins can be of the following types –
  - Self-join
  - Inner-join
  - Outer-join – left join, right join, and full join
- By default, it always performs **inner join**

Syntax:

result = JOIN relation1 BY columnname, relation2 BY columnname;

Example:

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;
```

Sample Input Data:

student.tsv:

```
101    Alice    3.5
102    Bob      3.8
103    Charlie  3.6
```

department.tsv:

```
101    10    ComputerScience
102    20    Mechanical
104    30    Electrical
```

Expected Output:

DUMP C (joined data):

```
(101, Alice, 3.5, 101, 10, ComputerScience)
(102, Bob, 3.8, 102, 20, Mechanical)
```

DUMP B (all records from department.tsv):

```
(101, 10, ComputerScience)
(102, 20, Mechanical)
(104, 30, Electrical)
```

8. UNION

The **UNION** operator of Pig Latin is used to merge the content of two relations. To perform UNION operation on two relations, their columns and domains must be identical.

Syntax:

Relation_name3 = UNION Relation_name1, Relation_name2;

Example:

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;
```

Sample Input Data:

student.tsv:

```
101    Alice    3.5
102    Bob      3.8
103    Charlie  3.6
```

department.tsv:

```
101    10    ComputerScience
102    20    Mechanical
104    30    Electrical
```

Expected Output:

The union operation would result in the following merged data if schemas were made compatible (e.g., filling missing values with `NULL`):

C (Merged Data):

```
(101, Alice, 3.5, NULL, NULL)
(102, Bob, 3.8, NULL, NULL)
(103, Charlie, 3.6, NULL, NULL)
(101, NULL, NULL, 10, ComputerScience)
(102, NULL, NULL, 20, Mechanical)
(104, NULL, NULL, 30, Electrical)
```

DUMP B (Department Data):

```
(101, 10, ComputerScience
(102, 20, Mechanical)
(104, 30, Electrical)
```

9. SPLIT

The **SPLIT** operator is used to split a relation into two or more relations.

Syntax:

SPLIT Relation1_name INTO Relation2_name
IF (condition1), Relation2_name (condition2),

Example:

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

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

Input:

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

Act:

```
A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);
SPLIT A INTO X IF gpa==4.0, Y IF gpa<=4.0;
DUMP X;
```

Sample Input Data:

student.tsv:

```
(1, Alice, 4.0)
(2, Bob, 3.8)
(3, Charlie, 4.0)
(4, Diana, 3.5)
```

Expected Output:

Output for `X`:

```
(1, Alice, 4.0)
(3, Charlie, 4.0)
```

10. SAMPLE

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

Syntax:

result = JOIN relation1 BY columnname, relation2 BY columnname;

Example:

Objective: To depict the use of *SAMPLE*.

Input:

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

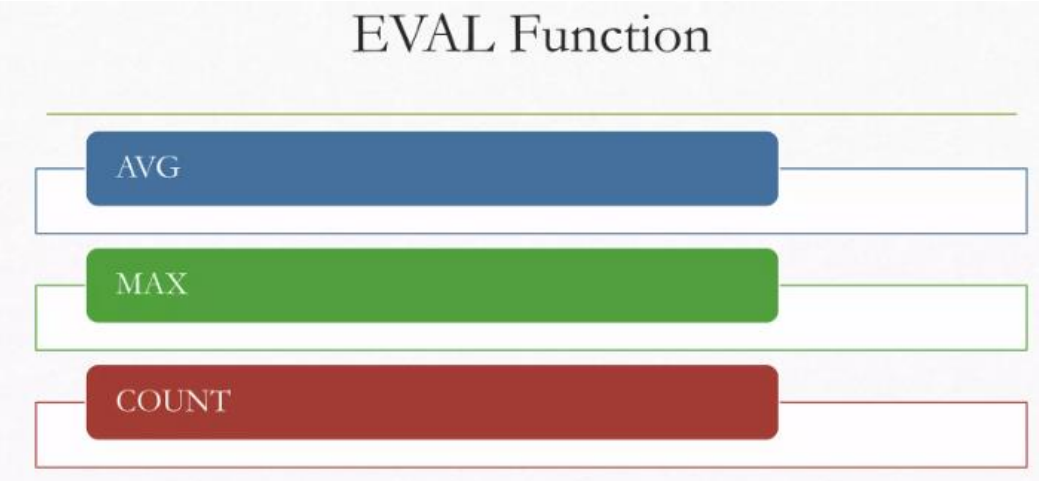
Act:

```
A = load '/pigdemo/student.tsv' as (rollno:int, name:chararray, gpa:float);
B = SAMPLE A 0.01;
DUMP B;
```

Expected Output: With a sample rate of 0.01, it's possible that no records or just one record (depending on the total dataset size) will be selected randomly.

EVAL Function

Eval Functions is the first types of Pig Built in Functions. Here are the Pig Eval functions, offered by Apache Pig.



1. AVG

The Pig-Latin **AVG()** function is used to compute the average of the numerical values within a bag. While calculating the average value, the **AVG()** function ignores the NULL values.

Syntax:

AVG(expression)

Example:

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;
```

Expected Output Format:

If the `student.csv` file contains data such as:

```
John,85
Jane,90
John,75
Jane,95
Doe,88
```

The output will be:

```
(John,80.0)
(Jane,92.5)
(Doe,88.0)
```

2. MAX

The Pig Latin **MAX()** function is used to calculate the highest value for a column (numeric values or chararrays) in a single-column bag. While calculating the maximum value, the **Max()** function ignores the NULL values..

Syntax:

MAX(expression)

Example:

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;
```

Expected Output Format:

If the `student.csv` file contains data such as:

```
John,85
Jane,90
John,75
Jane,95
Doe,88
```

The output will be:

```
(John,85)
(Jane,95)
(Doe,88)
```

3. COUNT

The **COUNT()** function of Pig Latin is used to get the number of elements in a bag. While counting the number of tuples in a bag, the **COUNT()** function ignores (will not count) the tuples having a NULL value in the FIRST FIELD.

Syntax:

COUNT(expression)

Example:

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;
```

Expected Output Format:

If the `student.csv` file contains data such as:

```
John,85
Jane,90
John,75
Jane,95
Doe,88
```

The output will be:

```
(John,2)
(Jane,2)
(Doe,1)
```

Complex Data Types

1. TUPLE

A ***TUPLE*** is and ordered collection of fields.

Example:

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;
```

Expected Output:

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

2. MAP

A ***TUPLE*** represents a key/value pair.

Example:

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
```

Expected Output:

```
(Bangalore)
(Pune)
(Chennai)
```

Questions on Unit-4

1. Explain the Hive architecture in detail. Also explain 3 types of Metastores.
2. Discuss the various data types supported in Hive.
3. Explain three file formats commonly used in Hive with examples.
4. Describe the features of the Hive Query Language (HQL). How can Hive be integrated into a data processing workflow? Explain.
5. Create a data file for below schemas:
   Order: CustomerId, ItemId, ItemName, OrderDate, DeliveryDate
   Customer: CustomerId, CustomerName, Address, City, State, Country
   i. Create a table for Order and Customer Data.
   ii. Write a HiveQL to find number of items bought by each customer.
6. What is RCFile in Hive? Explain its structure and advantages.
7. Explain the concept of SerDe in Hive. Explain the role of SerDe in processing XML data in Hive. Discuss the steps involved in configuring an XML SerDe for a Hive table and how XML data is transformed into a structured format for querying. Additionally, provide an example query on the resulting table and explain how it accesses XML elements.?
8. What are User-Defined Functions (UDFs) in Hive? Explain their purpose with an example of how to create and use a UDF.
9. Describe the anatomy of a Pig program and explain its components.
10. Write a Pig Latin script to perform the following tasks on a dataset sales_data (fields: product_id, category, amount, date):
    a. Filter the data for sales in the "Electronics" category.
    b. Calculate the total sales amount for each product_id in this category.
    c. Sort the results by total amount in descending order.

    Provide an explanation for each step in your script.
11. Explain User Defined Functions (UDFs) in Hive. Describe their purpose. Write a Hive function to convert the values of a field to uppercase.
12. Explain the RCFile format in Hive and its benefits. Create a table in Hive using the RCFile format and load sample data into it. Write a query to retrieve specific data from this table.
13. What is the philosophy behind Pig? How does it differ from traditional data processing systems?
14. Define the following key data processing operators in Pig Latin: LOAD, FILTER, FOREACH, GROUP, and JOIN. Explain each operator with an example.
15. Explain the following relational operators in Pig Latin with an example for each: a) FILTER b)GROUP BY c)JOIN d)UNION e)SPLIT
16. Describe how GROUP and JOIN operators work in Pig Latin and provide an example where both are used together to calculate the average score of students by course
17. What are the different execution modes of Pig? Compare their advantages and use cases.
18. Explain the following EVAL Functions with examples: AVG, MAX and COUNT.
19. Discuss the relational operators available in Pig. Provide examples of their usage.
20. What are Eval functions in Pig? Explain with examples of at least 2 commonly used Eval functions.
21. Explain the concept of complex data types in Pig. Illustrate the usage of tuples, and maps with examples.
22. Explain 2 types of table with example in Hive.