# Apache Hive

Apache Hive is a data warehouse infrastructure built on top of Hadoop for providing:

- Data summarization
- Querying
- Analysis of large datasets stored in Hadoop-compatible file systems

It is a distributed, fault-tolerant data warehouse system that enables analytics at a massive scale. Hive Metastore(HMS) provides a central repository of metadata that can easily be analyzed to make informed, data driven decisions, and therefore it is a critical component of many data lake architectures. Hive is built on top of Apache Hadoop and supports storage on S3, adls, gs etc though hdfs. Hive allows users to read, write, and manage petabytes of data using SQL. It provides a SQL-like interface (HiveQL) to query data stored in various databases and file systems that integrate with Hadoop.

Thus, Apache Hive is a data warehousing software built on top of Hadoop for providing data summarization, query, and analysis of large datasets stored in Hadoop compatible file systems (like HDFS) or other data stores (like HBase). It essentially brings the familiarity of SQL to the world of big data, allowing users to interact with massive datasets without needing to write complex MapReduce jobs. Hive is highly scalable, leveraging Hadoop's distributed architecture to handle petabytes of data, and it's extensible, allowing users to customize functionality through user-defined functions and storage handlers. It inherits Hadoop's fault-tolerance, ensuring data reliability, and operates on commodity hardware, making it a cost-effective solution for data warehousing. While Hive excels at batch processing and analytical queries, newer versions have incorporated features to improve interactive querying performance. Common use cases for Hive include data warehousing, ETL processes, ad-hoc querying, reporting, data mining, and historical data analysis.

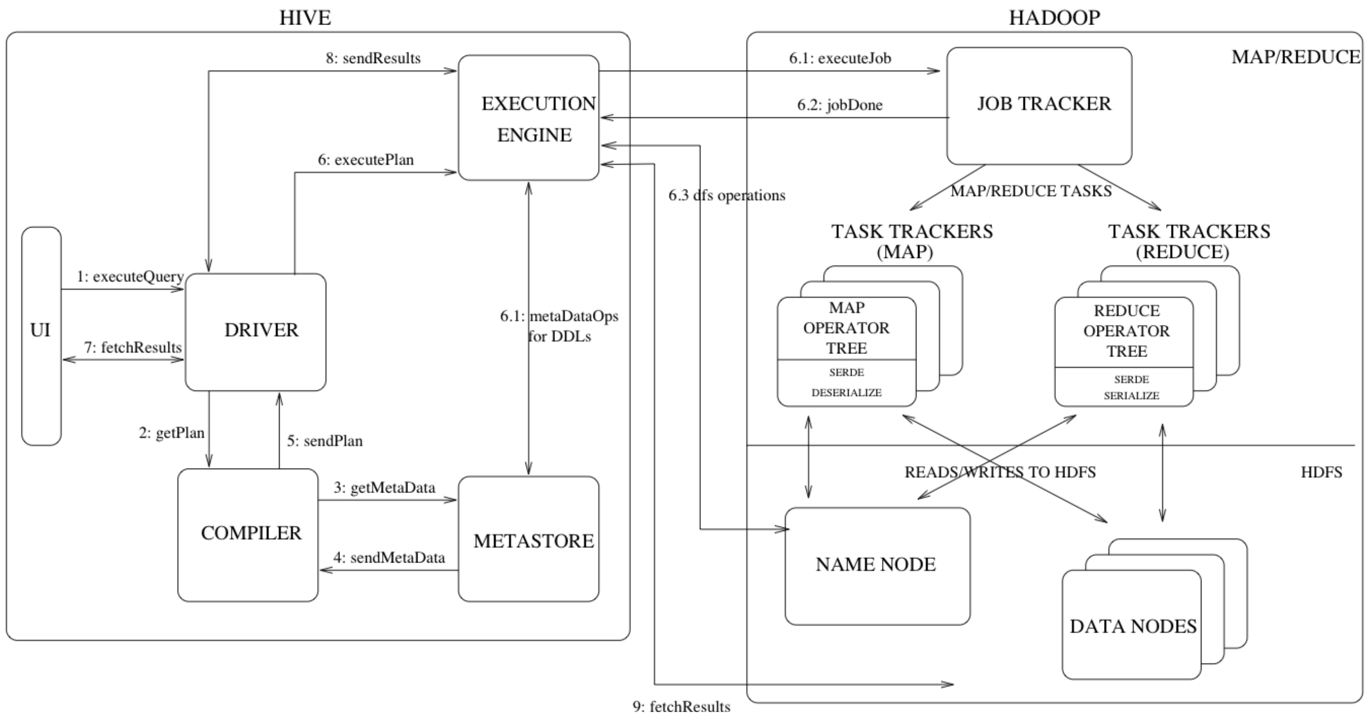
Despite its strengths, Hive has limitations. It's not designed for OLTP, and query latency can be higher than in traditional databases, though improvements have been made. Support for updates and deletes is limited, and schema enforcement is less strict than in traditional RDBMS. Recent developments in Hive include ACID transactions, performance enhancements, better integration with data lakes, and improved cloud integration.

## Key Features of Hive

- Supports SQL-like queries (HiveQL)
- Provides schema flexibility (supports schema-on-read)
- Enables ETL operations (Extract, Transform, Load)
- Offers partitioning and bucketing for performance optimization
- Supports custom MapReduce scripts
- Includes Hive Metastore for schema management

## Architecture of Hive

The architecture of Hive comprises several key components working in concert. The Hive Client serves as the interface for submitting HiveQL queries, with options including the Hive CLI, Beeline, web UIs, and programmatic interfaces. The Hive Metastore is a central repository storing metadata about Hive tables, including schema information, data types, and storage locations; it's crucial for Hive to understand data structure. The Metastore can be configured in embedded, local, or remote modes, with the remote mode being recommended for production environments to support concurrency. HiveServer (or HiveServer2) is the service that allows clients to submit HiveQL queries over JDBC/ODBC, providing concurrency control and authentication. The Hive Driver processes the HiveQL query, parsing, optimizing, and translating it into a physical execution plan. This plan, a DAG of MapReduce or Tez tasks, is then executed by the Execution Engine on the Hadoop cluster, interacting with the NameNode for metadata and DataNodes for the actual data. Hadoop, with HDFS for storage and MapReduce/Tez/Spark for processing, forms the foundation upon which Hive operates.



- **Hive Client**: This is the interface through which users submit their HiveQL queries. It could be the command-line interface (Hive CLI), Beeline (a JDBC/ODBC client), a web-based UI (like Hue), or even a program using a Java API. Essentially, it's how you interact with Hive.

- **Hive Metastore**: This is a central repository that holds all the metadata about the tables in Hive. This includes the schema (the structure of the data), data types, and where the data is located in HDFS. The Metastore is crucial because it allows Hive to understand the structure of your data. It can be configured in different ways, with the 'remote' mode being the most suitable for production environments.

- **Hive Server (HiveServer2)**: This service allows clients to submit HiveQL queries over JDBC/ODBC. Think of it as the component that allows other applications to connect to Hive. HiveServer2 also handles things like concurrency (multiple users running queries at the same time) and authentication (making sure users have the correct permissions).

- **Hive Driver**: When you submit a HiveQL query, the Driver is the component that processes it. It takes your query, checks that it's written correctly, figures out the best way to execute it, and then translates it into a series of tasks that Hadoop can perform. It essentially manages the lifecycle of a query.

- **Execution Engine**: This component takes the execution plan generated by the Driver and submits it to the Hadoop cluster. It then manages the execution of those tasks. It interacts with Hadoop's

NameNode (which manages the file system) and DataNodes (where the data is actually stored) to get the job done.

- **Hadoop (HDFS and MapReduce/Tez/Spark)**:
    - HDFS (Hadoop Distributed File System): This is the storage layer where Hive data resides. It's a distributed file system designed to store very large amounts of data reliably. Data in Hive is primarily stored in HDFS, and Hive supports various file formats, each with different performance and storage characteristics. Common formats include TextFile, SequenceFile, RCFile, ORC, Parquet, and Avro. ORC and Parquet are generally preferred for their efficiency. Storage Handlers extend Hive's capabilities to access data in other systems like HBase and Amazon S3.
    - MapReduce/Tez/Spark: These are the processing engines used by Hive. Hive translates HiveQL queries into jobs that these engines can execute on the Hadoop cluster. MapReduce was the original engine, but Tez and Spark are more modern and efficient alternatives.

Figure above illustrates the typical workflow of a query in the system. The process begins when the UI invokes the Driver's execute interface (Step 1). The Driver then initializes a session handle for the query and forwards it to the compiler for execution plan generation (Step 2). During compilation, the system retrieves required metadata from the metastore (Steps 3 and 4), which is used for expression type-checking in the query tree and partition pruning based on query predicates.

The compiler produces an execution plan (Step 5) structured as a Directed Acyclic Graph (DAG) of stages, where each stage may be a map/reduce job, a metadata operation, or an HDFS operation. For map/reduce stages, the plan includes map operator trees (executed on mappers) and, if needed, a reduce operator tree. The execution engine then dispatches these stages to the appropriate system components (Steps 6, 6.1, 6.2, and 6.3).

During task execution (whether in mappers or reducers), the system uses the relevant deserializer to read rows from HDFS files, processes them through the corresponding operator tree, and writes the output to a temporary HDFS file via a serializer. If no reduce phase is required, this happens entirely in the mapper. These temporary files serve as input for subsequent stages in the execution plan.

For Data Manipulation Language (DML) operations, the final temporary file is relocated to the table's designated HDFS location, leveraging HDFS's atomic rename operation to prevent dirty reads. In the case of queries, the execution engine retrieves the temporary file's contents directly from HDFS when the Driver initiates a fetch call (Steps 7, 8, and 9).

This approach ensures data consistency and efficient query execution across distributed stages.

## Hive Data Model

Hive organizes data into the following structures:

### 1. Tables

- Similar to tables in relational databases, Hive tables support operations like filtering, projection, joins, and unions.
- Each table’s data is stored in an HDFS directory.

- **External tables** can be created over existing HDFS files or directories by specifying the location in the table creation DDL.
- Tables consist of rows with strongly typed columns, much like traditional relational databases.

### 2. Partitions

- Tables can be partitioned using one or more partition keys, which determine how data is physically stored.
  - Example: A table `T` with a partition column `ds` (date) stores data for `ds='2008-09-01'` in `/ds=2008-09-01/` in HDFS.
- **Partition pruning** optimizes queries by scanning only relevant partitions.
  - Example: A query filtering on `T.ds = '2008-09-01'` only reads files under `/ds=2008-09-01/`.

### 3. Buckets

- Data within a partition can be further divided into **buckets** based on a column's hash value.
- Each bucket is stored as a separate file in the partition directory.
- **Bucketing improves efficiency** for:
  - Sampling queries (using the `SAMPLE` clause).
  - Join optimizations by reducing shuffling.

## Data Types in Hive

Hive supports:

- **Primitive types**: Integers, floats, strings, dates, booleans.
- **Complex types**: Arrays, maps (associative arrays).
- **User-defined types (UDTs)**: Users can define custom types by composing primitives, collections, or other UDTs.

### Type System & Extensibility

- The type system integrates with **SerDe (Serialization/Deserialization)** and **Object Inspector** interfaces.
- Users can:
  - Implement custom **Object Inspectors** to define how data is interpreted.
  - Create **custom SerDes** to read/write data in specialized HDFS formats.
- **Built-in Object Inspectors** (e.g., `ListObjectInspector`, `StructObjectInspector`, `MapObjectInspector`) enable flexible type composition.

### Operations on Complex Types

- **Maps & Arrays**: Supported by built-in functions like `size()` and index operators.
- **Nested types**: Accessed using dot notation (e.g., `a.b.c = 1` checks field `c` inside struct `b`, which is part of `a`).

This model allows Hive to efficiently manage structured and semi-structured data while supporting extensibility for custom formats.

## Hive Query Language (HiveQL)

HiveQL is Hive's query language, designed to resemble SQL in syntax and functionality. It supports standard database operations like table creation, data loading, and querying. A key differentiator of HiveQL is its ability to incorporate custom MapReduce scripts. Users can write these scripts in any programming language, with Hive handling the data flow through a simple streaming interface—reading input rows from standard input and writing processed results to standard output. While this string-based processing incurs some performance overhead, the flexibility to use familiar programming languages makes it a popular choice for many users.

Another distinctive feature of HiveQL is its support for multi-table inserts. This allows multiple queries to run against the same input data in a single statement. Hive optimizes these operations by scanning the input data just once, significantly improving query throughput compared to executing separate queries. While additional details could be provided, they are omitted here for brevity.

### 1. Basic HiveQL Commands

#### Creating a Table

```sql
-- Internal table (managed by Hive, data deleted when table is dropped)
CREATE TABLE employees (
    id INT,
    name STRING,
    salary FLOAT,
    department STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;

-- External table (references existing HDFS files, data remains if table is
dropped)
CREATE EXTERNAL TABLE employees_external (
    id INT,
    name STRING,
    salary FLOAT,
    department STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LOCATION '/user/hive/warehouse/employees_data/';
```

#### Loading Data into a Table

```sql
-- Load data from a local file
LOAD DATA LOCAL INPATH '/home/user/employees.csv' INTO TABLE employees;
```

```sql
-- Load data from HDFS
LOAD DATA INPATH '/hdfs/path/employees.csv' INTO TABLE employees_external;
```

#### Querying Data

```sql
-- Basic SELECT
SELECT * FROM employees WHERE department = 'Engineering';

-- Aggregation (GROUP BY)
SELECT department, AVG(salary) as avg_salary
FROM employees
GROUP BY department;

-- JOIN operations
SELECT e.name, d.department_name
FROM employees e
JOIN departments d ON e.department = d.dept_id;
```

---

## 2. Advanced Features

#### Partitioned Tables

```sql
-- Create a partitioned table (by date)
CREATE TABLE logs (
    log_id STRING,
    message STRING
)
PARTITIONED BY (dt STRING)
STORED AS ORC;

-- Load data into a specific partition
LOAD DATA INPATH '/logs/2023-10-01.csv' INTO TABLE logs PARTITION
(dt='2023-10-01');

-- Query with partition pruning (only scans relevant partitions)
SELECT * FROM logs WHERE dt = '2023-10-01';
```

#### Bucketed Tables

```sql
-- Create a bucketed table (for efficient sampling & joins)
CREATE TABLE user_activity (
    user_id INT,
    activity STRING,
    timestamp BIGINT
)
```

```
CLUSTERED BY (user_id) INTO 32 BUCKETS
STORED AS ORC;

-- Sample 10% of the data
SELECT * FROM user_activity TABLESAMPLE(BUCKET 1 OUT OF 32 ON user_id);
```

---

### 3. Custom MapReduce Scripts

HiveQL allows embedding **custom scripts** (Python, Bash, etc.) for advanced processing.

**Example: Using a Python Script**

```
-- Define the script
ADD FILE /scripts/filter_salary.py;

-- Use TRANSFORM to apply the script
SELECT TRANSFORM (id, name, salary)
USING 'python filter_salary.py'
AS (id INT, name STRING, salary FLOAT)
FROM employees;
```

*(`filter_salary.py` reads input rows from `stdin` and writes results to `stdout`.)*

---

### 4. Multi-Table Insert (Optimized Querying)

Instead of scanning the same data multiple times, Hive allows **a single scan for multiple outputs**.

```
FROM source_table
INSERT OVERWRITE TABLE high_salary_employees
    SELECT * WHERE salary > 100000
INSERT OVERWRITE TABLE low_salary_employees
    SELECT * WHERE salary <= 100000;
```

*(Hive optimizes this to read `source_table` only once.)*

---

## HiveQL vs. SQL**

| Feature | HiveQL | Traditional SQL |
|---|---|---|
| **Schema-on-read** | Yes (flexible) | No (schema-on-write) |
| **Transactions** | Limited (ACID in newer versions) | Full support |
| **Custom scripts** | Yes (Python, Perl, etc.) | No (requires UDFs) |

| Feature | HiveQL | Traditional SQL |
|---|---|---|
| **Optimization** | MapReduce-based | Query planner-based |

---

### User-Defined Functions (UDFs) in Hive

Hive allows users to extend its functionality by creating **User-Defined Functions (UDFs)** when built-in functions are insufficient. UDFs enable custom data processing in HiveQL queries and can be written in **Java, Python, or other scripting languages**.

---

## Types of UDFs in Hive

Hive supports three main types of UDFs:

| Type | Description | Example Use Case |
|---|---|---|
| **UDF (Regular)** | Processes **one input row** → **one output row** (like `LOWER()`, `UPPER()`) | String manipulation, math operations |
| **UDAF (Aggregate)** | Processes **multiple rows** → **single output** (like `SUM()`, `AVG()`) | Custom aggregations, statistics |
| **UDTF (Table-Generating)** | Processes **one input row** → **multiple output rows** (like `EXPLODE()`) | Flattening arrays/maps |

---

## How to Create & Use UDFs

### 1. Java UDF (Most Common)

**Step 1:** Write a Java class extending Hive's `UDF` class.

```java
package com.hive.udf;
import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;

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

**Step 2:** Compile & package as a JAR:

```
javac -cp $(hadoop classpath) ReverseStringUDF.java
jar cf rev_udf.jar com/hive/udf/ReverseStringUDF.class
```

**Step 3:** Register & use in Hive:

```
ADD JAR /path/to/rev_udf.jar;
CREATE TEMPORARY FUNCTION reverse_str AS 'com.hive.udf.ReverseStringUDF';
SELECT reverse_str(name) FROM employees;
```

**Output:**
`"John"` → `"nhoJ"`

---

## 2. Python UDF (Using TRANSFORM)

Hive can execute Python scripts via `TRANSFORM` (slower but flexible).
**Example:**

```
-- Define script (uppercase names)
ADD FILE /scripts/upper.py;
SELECT TRANSFORM(name)
USING 'python upper.py'
AS (upper_name STRING)
FROM employees;
```

**`upper.py`**:

```
import sys
for line in sys.stdin:
    print(line.strip().upper())
```

---

## 3. UDAF (Aggregate Function)

For custom aggregations (e.g., geometric mean):

```
package com.hive.udaf;
import org.apache.hadoop.hive.ql.exec.UDAF;
import org.apache.hadoop.hive.ql.exec.UDAFEvaluator;

public class GeometricMean extends UDAF {
    public static class GeoMeanEvaluator implements UDAFEvaluator {
        // Logic here
    }
}
```

**Usage:**

```
CREATE TEMPORARY FUNCTION geo_mean AS 'com.hive.udaf.GeometricMean';
SELECT geo_mean(salary) FROM employees;
```

---

## When to Use UDFs?

- **Custom string/number processing** (e.g., regex extraction)
- **Complex aggregations** (e.g., weighted averages)
- **Integration with external libraries** (e.g., ML models)
- **Avoid for simple operations** (use built-ins like `CONCAT()` instead)
- **Performance overhead** (Python UDFs are slower than Java)

---

# Big Data - Apache Pig

Apache Pig is an open-source platform developed by the Apache Software Foundation that sits on top of the Hadoop ecosystem. It is designed to facilitate the analysis of large data sets by providing a high-level scripting language known as **Pig Latin**. This language abstracts away the complexities of writing low-level MapReduce programs, making it easier for developers and analysts to transform and analyze data stored in Hadoop.

One of Pig's main strengths lies in its ability to work with both **structured** and **unstructured** data. Pig Latin scripts are compiled into sequences of **MapReduce** jobs or **Tez DAGs**, which are then executed in a parallel and distributed manner on Hadoop clusters. This parallelism allows Pig to efficiently handle massive data volumes.

Pig is composed of two main layers:

- **Language Layer**, which includes Pig Latin – a SQL-like language used to express data analysis tasks.
- **Infrastructure Layer**, which compiles Pig Latin scripts into executable tasks using MapReduce or Tez.

The three core properties of Pig Latin are:

1. **Ease of programming**: Data transformations are easier to express in Pig Latin compared to Java-based MapReduce.
2. **Optimization opportunities**: The Pig engine can optimize the execution of Pig scripts, enabling faster processing.
3. **Extensibility**: Developers can write their own User Defined Functions (UDFs) in languages like Java or Python to perform specialized data operations.

---

## Execution Modes

Apache Pig supports multiple execution modes that determine how and where a Pig script will run. The choice of execution mode typically depends on the environment and the nature of the data processing task.

### 1. Local Mode

In this mode, Pig runs on a single machine and accesses the local file system. It is useful for small datasets or testing purposes.

```
pig -x local
```

### 2. Tez Local Mode

This is an experimental mode that allows Pig to run on a local machine using the Tez engine. Tez is a more efficient execution framework compared to MapReduce.

```
pig -x tez_local
```

### 3. Spark Local Mode

Another experimental mode, it allows execution of Pig scripts using the Spark engine locally.

```
pig -x spark_local
```

### 4. MapReduce Mode

This is the default execution mode. Here, Pig scripts are converted into MapReduce jobs and run on a Hadoop cluster.

```
pig
# or
pig -x mapreduce
```

5. Tez Mode If Tez is configured on the Hadoop cluster, Pig can use it as the execution engine. Tez often provides better performance due to its DAG-based model.

```
pig -x tez
```

### 6. Spark Mode

Similar to Tez, Spark can also be used as an execution engine for Pig scripts. However, it requires a Spark cluster setup.

```
pig -x spark
```

## Comparison of Pig and SQL Databases

Apache Pig and traditional SQL-based databases are designed for different use cases, although they share some similarities in how they manipulate data.

- Purpose: Pig is mainly used for Extract, Transform, and Load (ETL) processes and large-scale data analysis in big data environments. On the other hand, SQL databases are used for managing structured data and supporting transactions.
- Data Model: Pig supports complex, nested data models, including bags (collections), tuples, and maps. SQL databases typically use flat, table-based schemas.
- Execution: Pig scripts are compiled into MapReduce or Tez jobs for execution across a Hadoop cluster. SQL queries are executed using optimized query planners and executors within relational database management systems.
- Data Types: Pig is flexible with semi-structured or unstructured data, whereas SQL databases expect structured data formats.
- Language: Pig Latin is a procedural data flow language, while SQL is declarative, focusing on what data to retrieve rather than how to retrieve it.
- Learning Curve: SQL has a lower learning curve for people familiar with databases, whereas Pig Latin might take more time to master, especially for users unfamiliar with distributed systems.
- Performance: SQL databases offer fast query performance for well-structured datasets. Pig, while optimized for large-scale data processing, may appear slower due to its reliance on batch processing via MapReduce.

## Apache Pig Architecture

Apache Pig architecture is divided into two primary components:

1. Pig Latin Language This is the user-facing component. Users write Pig scripts in Pig Latin, which is a high-level scripting language that expresses data transformation and processing logic.
2. Execution Environment The Pig runtime environment parses, validates, compiles, and optimizes Pig Latin scripts into executable tasks. These are either MapReduce jobs or DAGs (Directed Acyclic Graphs) in the case of Tez or Spark.

Pig scripts consist of a series of transformation steps applied to the input data. These steps form a pipeline, where each operation (like filtering or grouping) produces a new relation from an existing one. Internally, the Pig engine translates this pipeline into a sequence of jobs that are then run on Hadoop or another execution engine.

Pig abstracts the complexities of job scheduling and data partitioning, allowing users to focus solely on data logic.

## Pig Grunt (Interactive Shell)

Pig provides an interactive shell known as Grunt, where users can execute Pig Latin commands interactively. It is particularly useful for learning, testing, and debugging scripts.

To start the shell, users simply type:

```
pig
```

### Features of Grunt

- Interactive Execution: Users can test Pig commands one at a time and immediately view results.
- Testing & Debugging: It's easier to test segments of a larger script and inspect intermediate data.
- Data Loading & Storage: Data can be loaded from HDFS or the local file system and stored likewise.
- File System Access: Grunt allows access to HDFS or local directories using shell-like commands.
- Help Commands: Typing help within the shell shows available commands and syntax guidance.

## Example Pig Script

The following example demonstrates loading a data file and extracting a specific field.

```
A = load 'passwd' using PigStorage(':'); -- Loads the passwd file delimited
by ":"
B = foreach A generate $0 as id;         -- Extracts the first field
(username)
store B into 'id.out';                   -- Stores the result in 'id.out'
```

## Data Types in Pig Latin

Apache Pig supports various data types, which are categorized into two groups: **simple** (scalar) types and **complex** types.

### Simple Data Types

- `int`: 32-bit signed integer.
- `long`: 64-bit signed integer.
- `float`: 32-bit floating-point number.
- `double`: 64-bit floating-point number.
- `chararray`: A sequence of characters (string).
- `bytearray`: Used for untyped data, typically raw input.

### Complex Data Types

- **Tuple**: An ordered set of fields. Example: `(John, 25)`

- **Bag**: A collection of tuples. Example: `{(John,25),(Alice,30)}`
- **Map**: A set of key-value pairs. Keys must be `chararray`. Example: `[name#John, age#25]`

Pig's flexibility in handling these types makes it a suitable tool for semi-structured data.

---

## Relational Operators in Pig

Pig Latin provides a variety of relational operators that are used to process and transform data.

- LOAD: Loads data from the file system into Pig.

  ```
  A = LOAD 'input/data.txt' USING PigStorage(',') AS (name:chararray,
  age:int);
  ```

- STORE: Writes results to the file system.

  ```
  STORE A INTO 'output/result' USING PigStorage(',');
  ```

- FILTER Selects tuples based on a condition.

  ```
  B = FILTER A BY age > 30;
  ```

- FOREACH Applies transformations to each tuple.

  ```
  C = FOREACH B GENERATE name, age + 1;
  ```

- GROUP Groups data based on a key.

  ```
  D = GROUP A BY age;
  ```

- JOIN Joins two or more relations.

  ```
  E = JOIN A BY name, B BY name;
  ```

- ORDER Sorts tuples.

  ```
  F = ORDER A BY age DESC;
  ```

- DISTINCT Removes duplicate tuples.

  ```
  G = DISTINCT A;
  ```

- LIMIT Restricts the number of output tuples.

  ```
  H = LIMIT A 10;
  ```

### Pig vs Hive

Apache Pig and Hive are both built on top of Hadoop and help users process large datasets using simpler abstractions over raw MapReduce code. However, they serve different audiences and have different philosophies:

1. Language Pig uses Pig Latin, a data flow language that is procedural in nature.

Hive uses HiveQL, a declarative language similar to SQL.

2. Paradigm Pig follows a procedural data flow model.

Hive follows a declarative model like traditional SQL.

3. Best Use Case Pig is better for ETL (Extract, Transform, Load) processes where you are cleaning, transforming, and preparing data.

Hive is better for analytical queries and reporting on large datasets.

4. Users Pig is more suited for programmers and developers who prefer scripting.

Hive is designed for data analysts familiar with SQL.

5. Extensibility Pig allows easy creation of User Defined Functions (UDFs) in Java, Python, etc.

Hive also supports UDFs, but Pig’s model offers more flexibility in custom data transformations.

6. Schema Pig allows optional schema definitions. You can process data without specifying column names or types.

Hive requires a strict schema, similar to traditional databases.

7. Execution Pig runs on MapReduce.

Hive initially used MapReduce but now also supports Tez and Spark, making it faster for some workloads.

## Use Cases of Apache Pig

Apache Pig is widely used in big data environments where data pipelines involve large-scale ETL tasks.

- Log Processing: Analyzing web server logs, clickstream data, etc.
- Data Cleansing: Filtering out corrupt or malformed records.

- ETL Workflows: Transforming raw data before loading it into a warehouse.
- Machine Learning Preprocessing: Preparing large datasets for model training.
- Social Media Analytics: Aggregating data for sentiment analysis or trend tracking.
- Pig’s simple syntax and MapReduce abstraction make it effective for quick development of big data jobs.