Here’s a comparison of **Apache Pig, Apache Hive, and SQL** based on key aspects:

Feature	Apache Pig	Apache Hive	SQL
Purpose	Designed for processing large-scale data in Hadoop using scripts.	Used for querying and managing structured data in Hadoop using SQL-like queries.	Used for querying and managing structured data in traditional relational databases.
Language	Pig Latin (procedural scripting language).	HiveQL (SQL-like query language).	Standard SQL.
Ease of Use	Requires learning Pig Latin, which is simpler than Java but not as familiar as SQL.	Easy for users familiar with SQL.	Standardized and widely known.
Processing Type	Data flow-based, procedural.	Query-based, declarative.	Query-based, declarative.
Performance	Faster for ETL tasks and data transformations.	Slower than Pig for complex transformations but optimized for queries.	Optimized for transactions and quick responses in traditional databases.
Use Cases	Data preprocessing, ETL, and unstructured/semi-structured data handling.	Interactive querying and data analysis on large datasets.	Transactional and analytical queries on structured data.
Underlying Execution	Converts Pig Latin scripts to MapReduce, Tez, or Spark jobs.	Converts HiveQL to MapReduce, Tez, or Spark queries.	Runs directly on relational database engines.
Schema Enforcement	Schema-on-read (can handle unstructured data).	Schema-on-read (structured but flexible).	Schema-on-write (strict structure).
Support for Joins	Supports joins but not as optimized as Hive.	Optimized for SQL-style joins.	Strong support for joins.
Integration with Hadoop	Runs on top of Hadoop and supports HDFS.	Runs on top of Hadoop and supports HDFS.	Does not natively support Hadoop (though some databases integrate with it).

Feature	Apache Pig	Apache Hive	SQL
Best For	Complex data transformations.	Analytical querying on big data.	Structured transactional databases.

Summary:

- **Use Pig** for **ETL and data transformation tasks** in Hadoop.
- **Use Hive** when dealing with **structured data and SQL-like queries** on Hadoop.
- **Use SQL** for **traditional relational databases**.

Would you like a deeper dive into any specific aspect? 🚀

Apache Pig provides multiple execution methods, allowing flexibility depending on the environment and use case. Here are the main execution methods:

1. Interactive Mode (Grunt Shell)

- The **Grunt Shell** is Pig's interactive command-line interface.
- You can enter Pig Latin commands line by line and execute them on the fly.
- Useful for testing and debugging scripts.

Command to start Grunt Shell

- **Local Mode** (Runs on a single machine without Hadoop):
- pig -x local
- **MapReduce Mode** (Runs on Hadoop cluster):
- pig

2. Script Mode

- Run Pig Latin scripts stored in a .pig file.
- Useful for executing multiple commands in a batch process.

Example Command

pig -x local myscript.pig

or

pig myscript.pig

(if running on a Hadoop cluster)

3. Embedded Mode (Using Java API)

- Allows embedding Pig scripts inside Java programs.
- Useful when integrating Pig with other applications.

Example Java Code

```
import org.apache.pig.PigServer;

import java.util.*;


public class PigEmbeddedExample {

   public static void main(String[] args) throws Exception {

      PigServer pigServer = new PigServer("local");

      pigServer.registerQuery("A = LOAD 'data.txt' USING PigStorage(',') AS (id:int,
name:chararray);");

      pigServer.registerQuery("B = FILTER A BY id > 100;");

      pigServer.store("B", "output");

   }

}
```

4. Batch Mode (Running Pig with a Parameter File)

- You can use **parameter substitution** and pass arguments dynamically using a parameter file.

Example Command

```
pig -param_file params.txt myscript.pig
```

where params.txt contains:

```
INPUT=data.txt

OUTPUT=result
```

Inside myscript.pig, reference them as:

```
A = LOAD '$INPUT' USING PigStorage(',');

STORE A INTO '$OUTPUT';
```

5. Tez and Spark Execution Mode (Optional)

- Pig can run on **Tez** or **Apache Spark** for better performance instead of the default MapReduce mode.

Running with Tez:

pig -x tez

Running with Spark:

pig -x spark

Summary

Execution Method	Use Case
Grunt Shell	Interactive execution and debugging
Script Mode	Running full Pig scripts in batch mode
Embedded Mode	Running Pig within a Java program
Batch Mode	Running Pig with dynamic parameters
Tez/Spark Mode	Faster execution compared to MapReduce

Would you like details on any specific execution mode? 🚀

Apache Pig provides **relational operators** to process and transform data efficiently. These operators work on datasets stored in Hadoop Distributed File System (HDFS) or other storage sources.

---

1. Load & Store Operators

Operator	Description	Example
LOAD	Reads data from HDFS or local storage.	A = LOAD 'data.txt' USING PigStorage(',') AS (id:int, name:chararray, age:int);
STORE	Writes data to HDFS or local storage.	STORE A INTO 'output' USING PigStorage(',');

---

2. Filtering & Transformation Operators

Operator	Description	Example
FILTER	Filters data based on a condition.	B = FILTER A BY age > 25;
FOREACH … GENERATE	Applies transformations to each row and extracts specific fields.	C = FOREACH A GENERATE name, age * 2;
DISTINCT	Removes duplicate rows from a dataset.	D = DISTINCT A;
MAPREDUCE	Runs a custom MapReduce function.	E = MAPREDUCE 'com.example.MyMapperReducer' STORE output;

---

3. Grouping & Aggregation Operators

Operator	Description	Example
GROUP	Groups data by a specific column.	G = GROUP A BY age;
COGROUP	Similar to GROUP, but used for multiple datasets.	G = COGROUP A BY id, B BY id;
JOIN	Performs SQL-style joins on datasets.	H = JOIN A BY id, B BY id;
CROSS	Computes the Cartesian product of two datasets.	I = CROSS A, B;

---

4. Sorting & Ordering Operators

Operator	Description	Example
ORDER BY	Sorts data based on a column.	J = ORDER A BY age DESC;
LIMIT	Retrieves a specific number of rows.	K = LIMIT A 10;

---

5. Combining & Splitting Data

Operator	Description	Example
UNION	Combines two datasets (similar to SQL UNION).	L = UNION A, B;

Operator	Description	Example
SPLIT	Splits data into multiple datasets based on conditions.	SPLIT A INTO M IF age < 30, N IF age >= 30;

---

6. Debugging Operators

Operator	Description	Example
DUMP	Displays data on the terminal (for debugging).	DUMP A;
DESCRIBE	Shows the schema of a relation.	DESCRIBE A;
EXPLAIN	Displays the logical execution plan.	EXPLAIN A;
ILLUSTRATE	Shows step-by-step transformations.	ILLUSTRATE A;

Summary

Category	Operators
Loading & Storing	LOAD, STORE
Filtering & Transformations	FILTER, FOREACH, DISTINCT, MAPREDUCE
Grouping & Aggregation	GROUP, COGROUP, JOIN, CROSS
Sorting & Ordering	ORDER BY, LIMIT
Combining & Splitting	UNION, SPLIT
Debugging	DUMP, DESCRIBE, EXPLAIN, ILLUSTRATE

Would you like an example script using these operators? 🚀

Here's an **Apache Pig Latin script** that demonstrates **sorting, filtering, and joining** datasets.

Scenario:

We have two datasets:

1. **employees.txt** → Contains employee details (ID, Name, Age, Department ID).
2. **departments.txt** → Contains department details (Department ID, Department Name).

We will:
✅ **Filter** employees older than 25
✅ **Join** employees with departments
✅ **Sort** employees by name

Sample Data Files

employees.txt

1,John,30,101

2,Alice,24,102

3,Bob,28,101

4,Charlie,27,103

5,David,23,102

departments.txt

101,HR

102,Finance

103,Engineering

Pig Latin Script (sort_filter_join.pig)

```
-- Load Employee Data
employees = LOAD 'employees.txt' USING PigStorage(',')
        AS (id:int, name:chararray, age:int, dept_id:int);


-- Load Department Data
departments = LOAD 'departments.txt' USING PigStorage(',')
          AS (dept_id:int, dept_name:chararray);


-- Filter employees older than 25
filtered_employees = FILTER employees BY age > 25;
```

-- Join employees with department data

joined_data = JOIN filtered_employees BY dept_id, departments BY dept_id;

-- Sort by employee name

sorted_data = ORDER joined_data BY name ASC;

-- Store output

STORE sorted_data INTO 'output' USING PigStorage(',');

Expected Output (output directory)

3,Bob,28,101,HR

4,Charlie,27,103,Engineering

1,John,30,101,HR

Command to Run the Script

pig -x local sort_filter_join.pig

or on Hadoop cluster:

pig sort_filter_join.pig

Breakdown of the Script

1. **LOAD** → Loads employee and department data.
2. **FILTER** → Keeps employees older than 25.
3. **JOIN** → Combines employees with department details using dept_id.
4. **ORDER BY** → Sorts employees alphabetically.
5. **STORE** → Saves the result in the output directory.

Would you like modifications or an explanation of any step? 🚀

Hive: External and Managed Tables

In Hive, tables can be classified into two types: **Managed (Internal) Tables** and **External Tables**. The choice between these depends on how the data is stored and managed.

---

1. Managed (Internal) Tables

Managed tables are controlled entirely by Hive, meaning that both the data and metadata are managed within Hive's warehouse directory.

Characteristics of Managed Tables

- Data is **moved** to Hive’s warehouse directory (/user/hive/warehouse).
- Hive **manages** the table's lifecycle, meaning if you **drop** the table, both metadata and data are **deleted**.
- Suitable for cases where Hive should **own and manage** the data.

Creating a Managed Table

```
CREATE TABLE managed_table (
  id INT,
  name STRING
);
```

- By default, this is a **managed table**.
- When data is loaded, it is stored in the Hive warehouse directory.

Loading Data into a Managed Table

```
LOAD DATA INPATH '/user/data.txt' INTO TABLE managed_table;
```

- The data file is **moved** from its location into the Hive warehouse.

Dropping a Managed Table

```
DROP TABLE managed_table;
```

- The **table and data** are **both deleted** from Hive's storage.

---

2. External Tables

External tables allow Hive to access data stored **outside** the warehouse directory, meaning Hive only manages the metadata, but the actual data remains at its original location.

Characteristics of External Tables

- The data **remains in its original location**.
- Dropping an external table only **removes the metadata**, but the actual data remains untouched.
- Useful when multiple applications or users need access to the same dataset.

Creating an External Table

```
CREATE EXTERNAL TABLE external_table (
   id INT,
   name STRING
)
LOCATION '/user/external_data';
```

- The LOCATION clause specifies where the data resides.
- Hive does **not move** the data but simply **references** it.

Loading Data into an External Table

```
LOAD DATA INPATH '/user/data.txt' INTO TABLE external_table;
```

- The data is **not moved** to the warehouse directory.

Dropping an External Table

```
DROP TABLE external_table;
```

- **Metadata is deleted**, but the actual data remains in its original location.

---

3. When to Use Managed vs. External Tables

Feature	**Managed Table**	**External Table**
Data Storage	Moved to Hive Warehouse	Remains in the specified location
Data Management	Hive fully manages	User/application manages
Dropping Table	Deletes **both data & metadata**	Deletes **only metadata**
Use Case	When Hive **owns the data**	When data is **shared** or used by other applications

4. Best Practices

- **Use managed tables** when Hive should **fully control** the data.
- **Use external tables** when working with **existing datasets** that other applications also use.
- **Backup important data** before dropping a table, especially for managed tables, as the data will be permanently deleted.

Would you like more details on specific commands or configurations?

HiveQL Operations: Syntax & Examples

HiveQL (Hive Query Language) is a SQL-like language used in Apache Hive for querying and managing large datasets stored in Hadoop. Below are the key HiveQL operations along with syntax and examples.

---

1. Database Operations

Create a Database

CREATE DATABASE my_database;

Use a Database

USE my_database;

Drop a Database

DROP DATABASE my_database CASCADE;

- CASCADE removes all tables inside the database.

---

2. Table Operations

Create a Table

```
CREATE TABLE employees (
   id INT,
   name STRING,
   age INT,
   department STRING
```

)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ',';

- Defines a **managed table** with comma-separated values.

Create an External Table

CREATE EXTERNAL TABLE external_employees (

id INT,

name STRING,

age INT,

department STRING

)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ','

LOCATION '/user/hive/external/employees';

- The data remains at the specified **external location**.

Describe a Table

DESCRIBE employees;

Alter Table (Rename)

ALTER TABLE employees RENAME TO staff;

Alter Table (Add Column)

ALTER TABLE employees ADD COLUMNS (salary FLOAT);

Drop a Table

DROP TABLE employees;

- **Managed Table** → Data & Metadata deleted.
- **External Table** → Only Metadata deleted.

---

3. Data Manipulation Operations

Insert Data into a Table

INSERT INTO TABLE employees VALUES (1, 'Alice', 30, 'HR');

Insert Data from Another Table

INSERT INTO TABLE new_employees

SELECT * FROM employees WHERE age > 25;

Load Data from File

LOAD DATA INPATH '/user/data/employees.csv'

INTO TABLE employees;

- Moves the file to Hive's warehouse **(Managed Table).**
- Leaves the file in its location for **External Tables**.

Update Data (Only in ACID Tables)

UPDATE employees SET department = 'Finance' WHERE id = 1;

Delete Data (Only in ACID Tables)

DELETE FROM employees WHERE age < 25;

---

4. Querying Data

Select All Data

SELECT * FROM employees;

Select Specific Columns

SELECT name, age FROM employees;

Filter Data (WHERE)

SELECT * FROM employees WHERE department = 'HR';

Sort Data (ORDER BY)

SELECT * FROM employees ORDER BY age DESC;

Group Data (GROUP BY)

SELECT department, COUNT(*) FROM employees GROUP BY department;

Filter Grouped Data (HAVING)

SELECT department, COUNT(*) FROM employees

GROUP BY department

HAVING COUNT(*) > 5;

---

5. Joins in HiveQL

Inner Join

SELECT e.id, e.name, d.dept_name

FROM employees e

JOIN departments d

ON e.department_id = d.id;

Left Join

SELECT e.id, e.name, d.dept_name

FROM employees e

LEFT JOIN departments d

ON e.department_id = d.id;

Right Join

SELECT e.id, e.name, d.dept_name

FROM employees e

RIGHT JOIN departments d

ON e.department_id = d.id;

Full Outer Join

SELECT e.id, e.name, d.dept_name

FROM employees e

FULL OUTER JOIN departments d

ON e.department_id = d.id;

---

6. Advanced Queries

Using CASE Statement

SELECT name, age,

CASE

```sql
        WHEN age < 30 THEN 'Young'
        WHEN age BETWEEN 30 AND 50 THEN 'Middle Age'
        ELSE 'Senior'
    END AS age_group
FROM employees;
```

Subqueries

```sql
SELECT name FROM employees
WHERE id IN (SELECT id FROM new_employees WHERE age > 30);
```

---

7. Partitioning and Bucketing

Create a Partitioned Table

```sql
CREATE TABLE employees_partitioned (
    id INT,
    name STRING,
    age INT
)
PARTITIONED BY (department STRING);
```

Insert into Partitioned Table

```sql
INSERT INTO TABLE employees_partitioned
PARTITION (department='HR')
VALUES (1, 'Alice', 30);
```

Create a Bucketed Table

```sql
CREATE TABLE employees_bucketed (
    id INT,
    name STRING
)
CLUSTERED BY (id) INTO 4 BUCKETS;
```

---

8. Views

Create a View

CREATE VIEW high_paid_employees AS

SELECT * FROM employees WHERE salary > 50000;

Query a View

SELECT * FROM high_paid_employees;

Drop a View

DROP VIEW high_paid_employees;

Conclusion

HiveQL provides a powerful way to process and analyze large datasets in Hadoop. Mastering these commands will help you efficiently work with structured data stored in Hive.

Would you like any explanations on specific topics? 🚀

Partitioning and Bucketing in Hive - Example Programs

Hive **Partitioning** and **Bucketing** help improve query performance by reducing the amount of data scanned. Below are practical programs demonstrating both.

---

1. Partitioning in Hive

Partitioning divides a table into separate parts based on column values. This reduces query execution time by limiting data scanning.

Step 1: Create a Partitioned Table

CREATE TABLE employees_partitioned (

  id INT,

  name STRING,

  age INT,

  salary FLOAT

)

PARTITIONED BY (department STRING)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ','

STORED AS TEXTFILE;

- The department column is used as a **partition key**.
- Each department will have its own folder in HDFS.

Step 2: Load Data into Partitions

INSERT INTO TABLE employees_partitioned PARTITION (department='HR')

VALUES (1, 'Alice', 30, 60000);

INSERT INTO TABLE employees_partitioned PARTITION (department='IT')

VALUES (2, 'Bob', 28, 70000);

- Data is stored in HDFS as:
- /warehouse/employees_partitioned/department=HR/
- /warehouse/employees_partitioned/department=IT/

Step 3: Query Data from a Specific Partition

SELECT * FROM employees_partitioned WHERE department='HR';

- Only scans the HR partition, improving performance.

Step 4: Show Partitions

SHOW PARTITIONS employees_partitioned;

- Displays available partitions.

---

2. Dynamic Partitioning in Hive

Instead of manually specifying partitions, enable **dynamic partitioning**.

Step 1: Enable Dynamic Partitioning

SET hive.exec.dynamic.partition = true;

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

Step 2: Insert Data Dynamically

INSERT INTO TABLE employees_partitioned PARTITION (department)

SELECT id, name, age, salary, department FROM employees;

- Hive automatically creates partitions based on department.

---

3. Bucketing in Hive

Bucketing divides data into a fixed number of files based on **hashing** a column.

Step 1: Create a Bucketed Table

```
CREATE TABLE employees_bucketed (
  id INT,
  name STRING,
  age INT,
  salary FLOAT
)
CLUSTERED BY (id) INTO 4 BUCKETS
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
```

- CLUSTERED BY (id) INTO 4 BUCKETS → Data is divided into 4 buckets based on id.

Step 2: Enable Bucketing

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

Step 3: Load Data into Buckets

```
INSERT INTO TABLE employees_bucketed
SELECT * FROM employees;
```

- The id column is used to distribute data across 4 buckets.

Step 4: Verify Buckets

```
SHOW CREATE TABLE employees_bucketed;
```

- Confirms bucket creation.

---

4. Querying Partitioned and Bucketed Tables

Query Specific Partition

```
SELECT * FROM employees_partitioned WHERE department = 'IT';
```

Query Specific Bucket (Sampling)

SELECT * FROM employees_bucketed TABLESAMPLE(BUCKET 2 OUT OF 4);

- Retrieves data from **bucket 2** out of 4.

Key Differences

Feature	Partitioning	Bucketing
Concept	Divides table into separate directories	Divides table into fixed buckets (files)
Performance	Reduces scanning of irrelevant data	Ensures even distribution across buckets
Storage	Directory-based organization in HDFS	Data stored in bucketed files
Use Case	Ideal for categorical data (e.g., departments)	Useful for better join performance

---

Conclusion

- **Partitioning** is useful when filtering data by **high-cardinality columns** (e.g., department).
- **Bucketing** is useful for **optimized joins** by ensuring even distribution.
- **Combining both** (Partition + Bucketing) can further enhance query efficiency.

Would you like help with any specific queries? 🚀

HBase: Features and Architecture

Introduction to HBase

HBase is a **distributed, scalable, and NoSQL database** built on top of Hadoop's HDFS. It is designed for **real-time read/write access** to large datasets. Unlike relational databases, HBase follows a **column-oriented** storage model and supports structured and semi-structured data.

Key Features of HBase

🔹 **1. Schema-less (Flexible Data Model)**

- HBase stores data in a **key-value pair format**.
- It follows a **column-family-based** structure instead of tables and rows like RDBMS.

🔹 **2. High Scalability**

- Horizontally scalable by adding more nodes to the cluster.
- Suitable for **big data applications** with petabytes of data.

🔹 **3. Strong Consistency**

- Ensures **atomic read and write operations** within a row.
- Uses **HDFS for storage** and **WAL (Write-Ahead Logs)** for fault tolerance.

🔹 **4. Auto-Sharding (Automatic Partitioning)**

- Data is automatically split into **regions** (shards) and distributed across nodes.
- Regions are dynamically split and moved to balance the load.

🔹 **5. High Availability & Fault Tolerance**

- Data replication using **HDFS** ensures high availability.
- **HMaster** and **RegionServers** provide fault tolerance and distributed processing.

🔹 **6. Integration with Hadoop Ecosystem**

- Works with **MapReduce, Apache Spark, Hive, and Pig**.
- Can process real-time and batch data efficiently.

🔹 **7. Support for ACID Properties (at row level)**

- Ensures **Atomicity, Consistency, Isolation, and Durability (ACID)** at the **row level**.
- No support for multi-row transactions.

🔹 **8. Random & Sequential Reads/Writes**

- Unlike HDFS (which is optimized for batch processing), HBase supports **low-latency random reads/writes**.

🔹 **9. Column-Oriented Storage**

- Stores data in **columns (not rows)**, making it efficient for **analytics and sparse datasets**.

🔹 **10. Row-Level Security**

- Provides access control at the **row and column-family level** for security.

HBase Architecture

🔷 1. HBase Components

HBase follows a **master-slave architecture** with key components:

🔹 **(A) HMaster (Master Node)**

- **Manages metadata and region assignment** across RegionServers.
- **Handles schema changes** and administrative operations.
- **Load balances** by moving regions between servers.

🔹 **(B) RegionServers (Slave Nodes)**

- Stores actual **data** and handles **read/write requests**.
- Each RegionServer manages **multiple regions** (partitions of a table).
- Uses **MemStore (for writes) and HFiles (for permanent storage)**.

🔹 **(C) Regions**

- **Smallest unit of data storage** in HBase.
- Each table is divided into **multiple regions**, stored in **RegionServers**.

🔹 **(D) ZooKeeper**

- Ensures **coordination** and **failure detection** in HBase.
- Stores metadata about **RegionServers and HMaster**.

🔹 **(E) HDFS (Hadoop Distributed File System)**

- Stores all HBase data **persistently**.
- Ensures **replication and fault tolerance**.

🔷 2. HBase Data Model

HBase follows a **Key-Value Store Model** with **Column-Family** structure:

Row Key	Column Family 1	Column Family 2
ID123	Name: "Alice"	Salary: 60000
ID124	Name: "Bob"	Salary: 70000

Data Model Components:

- **Row Key** → Unique identifier for each row.
- **Column Family** → Groups related columns together.
- **Columns** → Store individual attributes.
- **Timestamp** → Each cell versioned by a timestamp.

HBase Read & Write Operations

🔹 **1. Write Operation**

1. **Client sends data** to HBase.
2. **Data is first written to WAL (Write-Ahead Log)** for durability.
3. **Data is stored in MemStore** (in-memory).
4. When MemStore is full, data is **flushed to HFiles** in HDFS.

🔹 **2. Read Operation**

1. **Client sends a request** to read data.
2. HBase **first checks MemStore** (for recent data).
3. If data isn't in MemStore, it looks in **HFiles** (HDFS).
4. **Data is returned to the client**.

HBase vs. RDBMS

Feature	**HBase**	**RDBMS**
Data Model	NoSQL (Column-oriented)	Relational (Row-based)
Schema	Flexible (Schema-less)	Fixed Schema
Scaling	Horizontally Scalable	Vertically Scalable
Joins	No support for Joins	Supports Joins
Transactions	Row-level ACID only	Full ACID Transactions
Query Language	HBase Shell, Java API	SQL

Feature	HBase	RDBMS
Read/Write Performance	Optimized for real-time access	Optimized for structured queries

---

Use Cases of HBase

✅ **Real-time Big Data Applications** – Log processing, fraud detection, IoT.
✅ **Social Media & Messaging** – Facebook uses HBase for storing messages.
✅ **Time-Series Data Storage** – Sensor data, stock market transactions.
✅ **Search Engines** – Indexing and storing web pages (e.g., Apache Nutch).

Conclusion

- HBase is a **highly scalable, distributed, column-oriented NoSQL database**.
- It provides **real-time read/write access** to large datasets.
- It is **not a replacement for RDBMS** but is ideal for **Big Data applications** where scalability and fast access are needed.

Would you like a **hands-on example** of using HBase commands? 🚀