
Module 4

Introduction to Hive

History of Apache Hive

- ◆ Origin at Facebook
 - 2007–2008: Facebook was rapidly generating huge volumes of log data (hundreds of GBs per day, growing into TBs).
 - Traditional RDBMS systems couldn't handle such large-scale data efficiently.
 - While Hadoop + MapReduce could store and process this data, writing MapReduce programs in Java was complex and time-consuming for analysts.
 - To solve this, Facebook engineers (Joydeep Sen Sarma, Ashish Thusoo, and team) developed Hive, a SQL-like interface for Hadoop.

9.1 What is Hive

□ Definition

- Hive is a Data Warehousing tool built on top of Hadoop, used for processing structured data in Hadoop.

□ Main Tasks of Hive

- Summarization
- Querying
- Analysis

□ Hive Uses

- HDFS for storage.
- MapReduce for execution.
- RDBMS for storing metadata/schemas (Metastore).

□ HiveQL (HQL)

- Similar to SQL.
- Compiles queries into MapReduce jobs and runs them on Hadoop.
- Designed mainly for OLAP (Online Analytical Processing).

□ Suitable For

- Data warehousing applications.
- Batch jobs on large immutable data (e.g., web logs, application logs).

Note

- Hive is not an RDBMS.
- Not designed for OLTP (Online Transaction Processing).
- Not suitable for real-time queries.
- Does not support row-level updates.

9.1.1 History of Hive and its recent releases.

◆ Hive 0.10

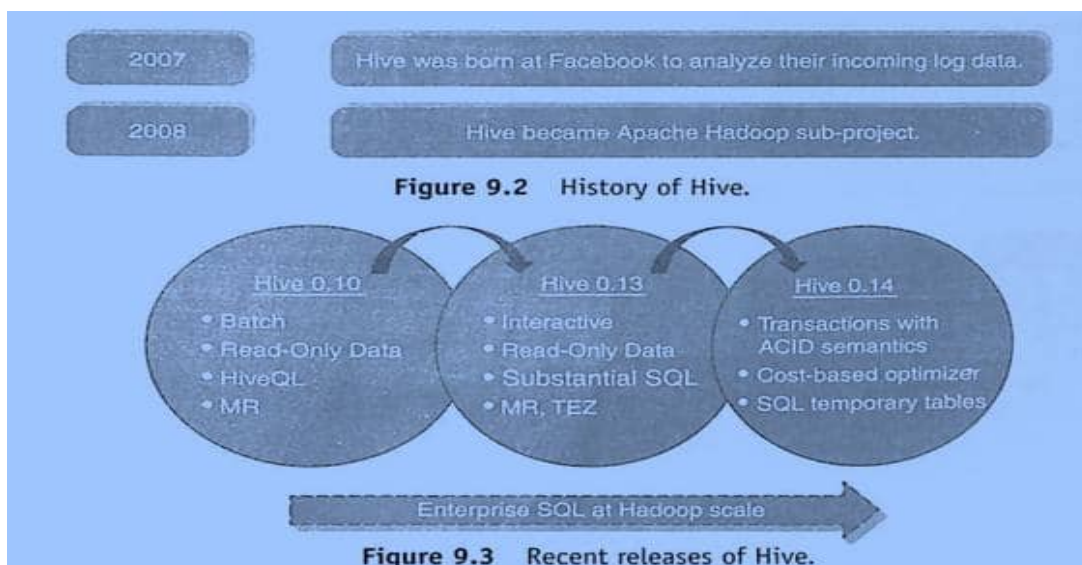
- Batch processing.
- Supports **read-only data**.
- Query language: **HiveQL**.
- Execution: **MapReduce (MR)**.

◆ Hive 0.13

- Introduced **interactive querying**.
- Still mainly **read-only data**.
- Added support for **substantial SQL features**.
- Execution engines: **MapReduce (MR)** and **Tez**.

◆ Hive 0.14

- Added **transaction support with ACID semantics**.
- Introduced **cost-based optimizer**.
- Support for **SQL temporary tables**.



9.1.2 Hive Features

1. Hive is **similar to SQL**.
2. **HiveQL (HQL)** is simple and easy to code.
3. Supports **rich data types** (structs, lists, maps).
4. Supports **SQL operations**: filters, GROUP BY, ORDER BY.
5. Allows defining **custom types** and **custom functions**.

9.1.3 Hive Integration and Work Flow

- **Explanation of workflow:**
 1. **Hourly log data** is collected.
 2. Data is stored in **HDFS**.
 3. **Log compression** is applied.
 4. Data is organized into **Hive tables** for querying and analysis.

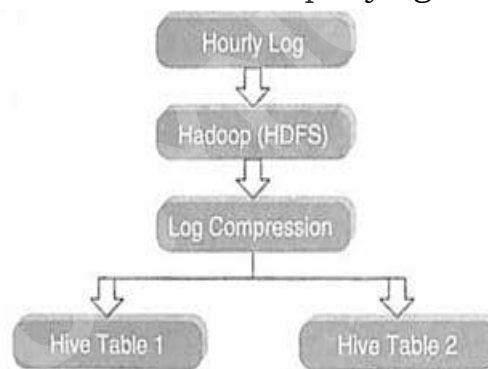


Figure 9.4 Flow of log analysis file.

9.1.4 Hive Data Units

1. **Databases** → Namespace for tables.
2. **Tables** → Set of records with similar schema.
3. **Partitions** → Logical division of data based on column values (e.g., country, year).
 - Each partition is stored in a **separate subdirectory** in HDFS.
 - Improves query performance by reading only relevant partitions.
4. **Buckets (or Clusters)** → Further division of partitions using **hashing**.
 - Records are distributed into fixed number of **buckets/files**.
 - Ensures even data distribution.

Partitioning Example

- A company (**XYZ Corp**) has **5M customer records across 190+ countries**.
- Without partitioning → Querying for a single country means scanning **all 5M records**.
- With partitioning → Data is divided by **country**, so queries only scan the relevant country's partition.

When to Use Partitioning/Bucketing?

- **Partitioning:**
 - Best for **low to medium cardinality columns** (e.g., country, year, month).
 - Too many partitions may cause performance issues.
 - Can partition on **multiple columns** (e.g., Year/Month/Day).
- **Bucketing:**
 - Best for **high cardinality columns** (e.g., customer_id).
 - Evenly distributes data into a fixed number of buckets using **hash function**.
 - Defined using:
 - CLUSTERED BY (customer_id) INTO XX BUCKETS;
 - Records with the same key always go into the same bucket.

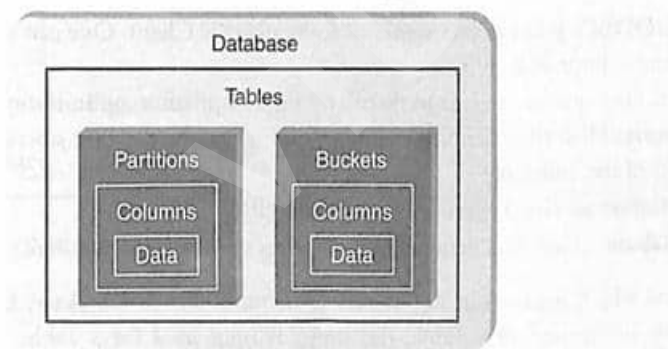


Figure 9.5 Data units as arranged in a Hive.

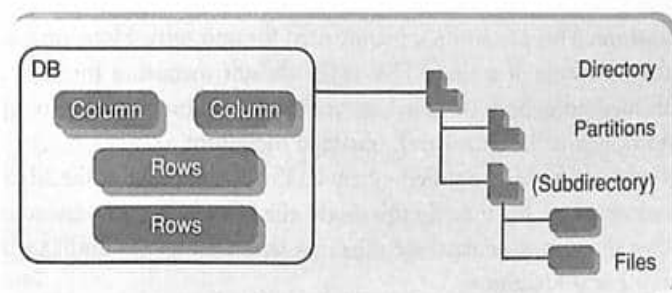


Figure 9.6 Semblance of Hive structure with database.

9.2 Hive Architecture

1. Interfaces to interact with Hive:

- **Hive Command-Line Interface (CLI):** Most commonly used interface.
- **Hive Web Interface:** GUI for executing queries and interacting with Hive.
- **Hive Server (Thrift Server):** Optional; allows submission of Hive jobs remotely.

2. Connectivity:

- **JDBC/ODBC support:** Enables jobs to be submitted via JDBC clients (e.g., Java programs).

3. Driver:

- Responsible for **query compilation, optimization, and execution.**

4. Metastore:

- Stores Hive **metadata** (table definitions, schema, partitions, mappings, etc.).
- Updated whenever tables are created or deleted.
- Includes IDs of databases, tables, indexes, input/output formats, etc.

Types of Metastore

1. Embedded Metastore:

- Default option (uses Apache Derby).
- Runs in the same JVM as Hive service.
- Only **one process/user** can connect at a time.
- Mainly used for **unit testing.**



Figure 9.8 Embedded Metastore.

2. Local Metastore:

- Metadata stored in **external RDBMS** (e.g., MySQL).
- Multiple processes/users can connect.
- Hive metastore service runs in Hive JVM, but **database runs in separate process/host.**

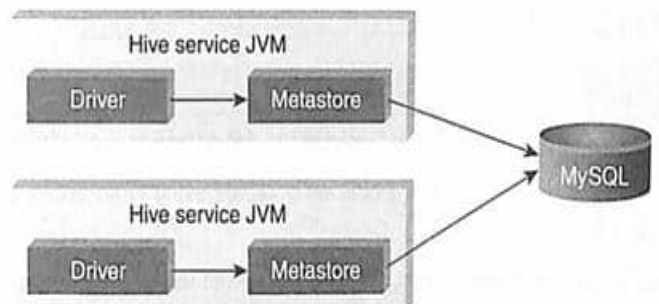


Figure 9.9 Local Metastore.

3. Remote Metastore:

- **Hive driver and metastore interface run in different JVMs** (possibly on different machines).
- Supports better isolation, firewalling, and multi-user access.
- Database credentials hidden from Hive users.

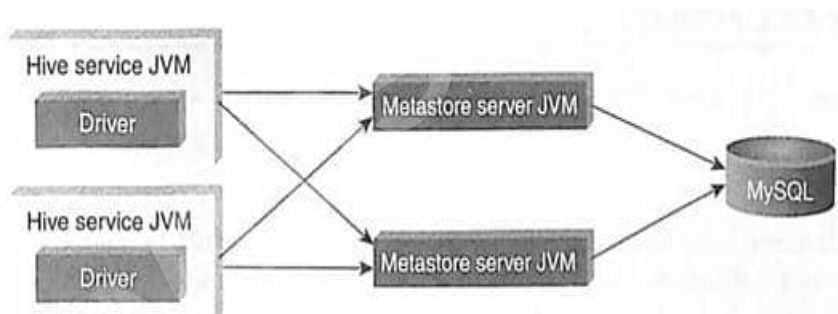


Figure 9.10 Remote Metastore.

9.3 Hive Data Types

9.3.1 Primitive Data Types

- Hive supports numeric types like **TINYINT (1-byte)**, **SMALLINT (2-byte)**, **INT (4-byte)**, **BIGINT (8-byte)**, **FLOAT (4-byte floating-point)**, and **DOUBLE (8-byte floating-point)** for storing numbers.
- For text, it provides **STRING** (default), **VARCHAR** (since Hive 0.12.0), and **CHAR** (since Hive 0.13.0). Strings can be written in either single (' ') or double (" ") quotes.
- Miscellaneous types include **BOOLEAN** for true/false values and **BINARY** for raw bytes (introduced in newer Hive versions).

◇ Numeric Data Types

- **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

◇ String Types

- **STRING** → Default string type
- **VARCHAR** → Available since Hive 0.12.0
- **CHAR** → Available since Hive 0.13.0
 - Strings can be expressed in **single quotes** (' ') or **double quotes** (" ")

◇ Miscellaneous Types

- **BOOLEAN** → True/False values
- **BINARY** → Available in Hive (binary data storage)

9.3.2 Collection Data Types

- **STRUCT** is like a C struct, meaning it can hold multiple fields of different types. You access each field using a dot notation (e.g., struct('John', 'Doe').first).
- **MAP** is a collection of key-value pairs. You retrieve values using square brackets with the key (e.g., map('first', 'John', 'last', 'Doe')['first']).
- **ARRAY** is an ordered list of elements of the same type. You access elements by their index (e.g., array('John', 'Doe')[0]).

9.4 Hive File Formats

1. Text File

- Default file format in Hive.
- Each record is stored as a line in the file.
- Different delimiters are used:
 - ^A → separates fields
 - ^B → separates elements in arrays/structs
 - ^C → separates key-value pairs
 - \n → new line for records
- Supports **CSV**, **TSV**, and also **JSON** or **XML** (when specified).

2. Sequential File

- A flat file that stores **binary key-value pairs**.
- Supports **compression** to reduce CPU and I/O requirements.

3. RCFile (Record Columnar File)

- Stores data in a **column-oriented format** instead of row-oriented.
- This makes **aggregation** faster and less expensive.
- Process:
 1. First, the table is **split into row groups** horizontally.
 2. Within each row group, data is further **partitioned vertically (by columns)**.
 3. This results in efficient storage and retrieval.

Example:

- Original table (Table 9.1) → split into row groups (Table 9.2).
- Then converted into RCFile format (Table 9.3), where data is stored column-wise within each row group.

Table 9.1 A table with four columns

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

Table 9.2 Table with two row groups

Row Group 1				Row Group 2			
C1	C2	C3	C4	C1	C2	C3	C4
11	12	13	14	41	42	43	44
21	22	23	24	51	52	53	54
31	32	33	34				

Table 9.3 Table in RCFile Format

Row Group 1	Row Group 2
11, 21, 31;	41, 51;
12, 22, 32;	42, 52;
13, 23, 33;	43, 53;
14, 24, 34;	44, 54;

9.5 Hive Query Language(HQL)

Hive Query Language (HQL) is similar to SQL and allows us to do tasks like:

1. Create and manage tables and partitions.
2. Use relational, arithmetic, and logical operators.
3. Run and evaluate functions.
4. Download query results to a local folder or HDFS.

9.5.1 DDL (Data Definition Language) Statements

DDL commands are used to **build and modify tables** or other objects. Common DDL commands are:

1. CREATE/DROP/ALTER DATABASE
2. CREATE/DROP/TRUNCATE TABLE
3. ALTER TABLE/PARTITION/COLUMN
4. CREATE/DROP/ALTER VIEW
5. CREATE/DROP/ALTER INDEX
6. SHOW (to display metadata)
7. DESCRIBE (to see table details)

9.5.2 DML (Data Manipulation Language) Statements

DML commands are used to **retrieve, store, modify, and update data** in Hive.

- Examples:
 1. Loading files into a table.
 2. Inserting data into Hive tables from queries.
- From Hive **0.14 onward**, UPDATE, DELETE, and TRANSACTION operations are supported.

9.5.3 Starting Hive Shell

To start Hive, go to the installation path and type:

```
$ hive
```

This opens the Hive shell, where you can write HiveQL commands.

Hive command structure follows 3 parts:

1. **Objective** → What are we trying to achieve?
2. **Input (optional)** → The data or query to work on.
3. **Act** → The HiveQL command we run.

4. **Outcome** → The result/output after execution.

Note:

- **DDL = Defines structure (create/alter/drop).**
- **DML = Works with data (insert/load/update/delete).**
- **Hive Shell = Place to run HiveQL commands.**

9.5.4 Hive Database

1. Creating a Database

- Syntax:
CREATE DATABASE IF NOT EXISTS STUDENTS
- COMMENT 'STUDENT Details'
- WITH DBPROPERTIES ('creator'='JOHN');
- **Clauses:**
 - IF NOT EXISTS: Prevents error if DB already exists.
 - COMMENT: Adds a short description.
 - WITH DBPROPERTIES: Stores key-value metadata (e.g., creator, owner).
- Note: We can use SCHEMA instead of DATABASE.

2. Viewing Databases

- Show all databases:
SHOW DATABASES;
- Filter by pattern:
SHOW DATABASES LIKE 'Stu*';
- Lists all databases in Hive metastore.

3. Describing a Database

- Basic info (name, comment, location):
DESCRIBE DATABASE STUDENTS;
- Extended info (properties too):
DESCRIBE DATABASE EXTENDED STUDENTS;

4. Altering a Database

- To update properties:
- ALTER DATABASE STUDENTS SET DBPROPERTIES ('edited-by' = 'JAMES');
- **Note:** You can add or modify DB properties but cannot remove them.

5. Using a Database

- To make a database the current working one:

USE STUDENTS;

- Optional setting:

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

(prints current DB in Hive prompt).

6. Dropping a Database

- Drop a DB:

DROP DATABASE STUDENTS;

- If DB contains tables → must use CASCADE:

DROP DATABASE STUDENTS CASCADE;

- Default is RESTRICT (prevents dropping DB with tables).

Note:

- ☐ **CREATE** makes a database with optional comments & properties.
- ☐ **SHOW** lists all databases (with filtering options).
- ☐ **DESCRIBE** shows metadata (basic or extended).
- ☐ **ALTER** updates DB properties.
- ☐ **USE** sets the working DB.
- ☐ **DROP** deletes a database (with CASCADE if it has tables).

9.5.6 Partitions

Partitioning in Hive

- Normally, Hive scans the entire dataset for a query, even when a filter is applied on a column (like StateName).
- This causes performance issues because the full dataset is read multiple times.
- **Partitioning** solves this by splitting data into smaller, logical parts (folders) based on a column value.
- Example: Instead of scanning all data for each state, Hive stores data in separate folders for each state. Queries only scan the relevant folder.

Types of Partitioning

1. Static Partitioning

- The user explicitly specifies the partition column value during data load.
- Example: Loading records into a folder where GPA = 4.0.

-
- Risk: If data is placed in the wrong partition folder, queries will return incorrect results.

2. Dynamic Partitioning

- Hive automatically creates partitions based on distinct column values found in the dataset.
- No need to manually define partition values.
- Requires enabling by setting:
 - SET hive.exec.dynamic.partition = true;
 - SET hive.exec.dynamic.partition.mode = nonstrict;
- This allows Hive to automatically generate folders for each unique partition value.

✦ Static Partition Example

- Create a table with partitioning on **GPA**.
- Load data into partition where GPA = 4.0.
- Hive automatically creates a folder for that partition.
- Additional partitions (like GPA = 3.5) can be added later using ALTER TABLE ... ADD PARTITION.

Dynamic Partition Example

- Create a table with partition on **GPA** (but do not specify values).
- Insert data, and Hive automatically creates folders for each unique GPA value found.
- This avoids manually specifying partitions and is useful for large, changing datasets.

Note:

- **Static Partition** → You decide and assign partitions manually.
- **Dynamic Partition** → Hive decides and creates partitions automatically based on data values.

9.5.7 Bucketing

□ Bucketing is a technique in Hive similar to partitioning, but instead of creating a partition for each unique column value, data is divided into a fixed number of buckets.

□ A bucket is a file, whereas a partition is a directory.

□ Bucketing helps avoid having too many partitions by limiting the number of buckets.

□ To enable bucketing in Hive, you need to set the property:

set hive.enforce.bucketing=true;

□ Example:

- A table STUDENT is created with fields (rollno, name, grade).
- Data is loaded into this table from a file (student.tsv).

□ A new table STUDENT_BUCKET is created, clustered by grade into 3 buckets.

CREATE TABLE IF NOT EXISTS STUDENT_BUCKET

(rollno INT, name STRING, grade FLOAT)

CLUSTERED BY (grade) INTO 3 BUCKETS;

- Data from the STUDENT table is inserted into the bucketed table:

INSERT OVERWRITE TABLE STUDENT_BUCKET

SELECT rollno, name, grade FROM STUDENT;

- The outcome shows that 3 bucket files are created under the bucketed table directory.
- To query a specific bucket, the TABLESAMPLE clause is used. Example:

SELECT DISTINCT grade

FROM STUDENT_BUCKET

TABLESAMPLE(BUCKET 1 OUT OF 3 ON grade);

- The above query retrieves data only from the first bucket. In the example shown, it fetched values 4.0 and 4.2.

Note: Bucketing distributes data into a fixed number of files (buckets), making queries faster and more efficient compared to creating many partitions.

9.5.8 Views in Hive

1. **Views in Hive** (introduced in version 0.6) are **purely logical objects**, meaning they don't store data physically but act as saved queries.

Creating a View

- **Objective:** Create a view named STUDENT_VIEW.
- **Command:**

```
CREATE VIEW STUDENT_VIEW AS
```

```
SELECT rollno, name FROM EXT_STUDENT;
```

- **Outcome:** View is created successfully.

Querying a View

- **Objective:** Query STUDENT_VIEW.
- **Command:**

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

Outcome: Displays the first 4 rows:

- 1001 John
- 1002 Jack
- 1003 Smith
- 1004 Scott

Dropping a View

- **Objective:** Delete the view.
- **Command:**
- DROP VIEW STUDENT_VIEW;
- **Outcome:** View is dropped successfully.

Key Points About Views in Hive

- A **view** is just a saved query definition (not a physical table).
- They are **read-only** in Hive (cannot be updated directly).
- Useful for simplifying queries and providing restricted data access.

9.5.9 Sub-Query in Hive

1. Sub-query Support in Hive

- Sub-queries are supported only in the **FROM clause** (from Hive version 0.12).
- Every table in a FROM clause must have a **name**.
- Columns in the sub-query can be used in the outer query just like normal table columns.
- From Hive **0.13**, sub-queries are also supported in the **WHERE clause**.

Objective

Count the occurrence of similar words in a file using a sub-query.

Steps

1. **Create a table docs to store lines**
2. CREATE TABLE docs (line STRING);
3. LOAD DATA LOCAL INPATH '/root/hivedemos/lines.txt' OVERWRITE INTO TABLE docs;
4. **Create a word count table using a sub-query**
5. CREATE TABLE word_count AS
6. SELECT word, COUNT(1) AS count
7. FROM (
8. SELECT explode(split(line, ' ')) AS word FROM docs
9.) w
10. GROUP BY word
11. ORDER BY word;
12. **Fetch results**
13. SELECT * FROM word_count;

Outcome

Words are split from lines using `explode(split(...))`.

Each word is counted.

Sample output:

Hadoop 2

Hive 2

Introducing 1

Introduction 1

Pig 1

Session 1

Welcome 2

Important Function

- `explode()` → Takes an array as input and outputs each element as a separate row.

9.5.10 Joins in Hive

1. What are Joins?

- Joins in Hive are similar to SQL joins.
- They are used to combine records from two tables based on a common column.

Objective

Create a JOIN between **Student** and **Department** tables using RollNo as the join key.

Steps

1. Create STUDENT table

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

Load data into student table:

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

2. Create DEPARTMENT table

```
CREATE TABLE IF NOT EXISTS DEPARTMENT(  
    rollno INT,  
    deptno INT,  
    name STRING  
)
```

ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';

Load data into department table:

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

3. Perform JOIN operation

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

Outcome (Sample Output)

```
101 John 4.0 101  
102 Jack 3.0 102  
103 Smith 3.5 103  
104 Scott 3.9 101  
105 Bob 3.6 102  
106 Alice 3.7 104  
107 Tom 3.8 103  
108 Harry 3.5 104
```

Key Points

- Hive **joins work like SQL joins**.
- Here we used an **INNER JOIN** (only matching roll numbers appear).
- Aliases a and b are used for simplicity (STUDENT a, DEPARTMENT b).
- Multiple join types exist in Hive:
 - **INNER JOIN** → Matches rows present in both tables.
 - **LEFT OUTER JOIN** → All rows from left + matching rows from right.
 - **RIGHT OUTER JOIN** → All rows from right + matching rows from left.
 - **FULL OUTER JOIN** → All rows from both tables.

9.5.10 Joins in Hive

1. What it is:

Joins in Hive work the same way as in SQL.

They allow combining rows from two or more tables based on a related column.

2. **Objective:**

Join **STUDENT** and **DEPARTMENT** tables using **RollNo** as the join key.

3. **Steps performed:**

- **Created STUDENT table** with columns:

```
rollno (INT),  
name (STRING),  
gpa (FLOAT).
```

- **Loaded data** into STUDENT from student.tsv.

- **Created DEPARTMENT table** with columns:

```
rollno (INT),  
deptno (INT),  
name (STRING).
```

- **Loaded data** into DEPARTMENT from department.tsv.

- **Executed Join Query:**

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

4. **Outcome (Result of Join):**

```
101  John   3.00  101  
102  Jack   4.00  102  
103  Smith  4.00  103  
104  Scott  3.90  101  
105  David  4.00  102  
106  James  4.20  103  
107  Tom    4.10  104  
108  Bob    3.85  104
```

✓ Key Points to Remember

- a and b are **aliases** for table names (used to simplify query writing).
- The query retrieves **RollNo, Name, GPA from Student table** and **DeptNo from Department table**.
- This is an **INNER JOIN** → it only shows rows where roll numbers exist in both tables.

9.5.11 Aggregation

- Hive supports **aggregation functions** such as avg, count, etc.
- **Objective:** Calculate the **average GPA** and **count of students**.
- **Query:**

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

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

- **Outcome:**
 - Average GPA = **3.83999965183027**
 - Count of students = **10**

9.5.12 Group By and Having

- GROUP BY is used to **group rows** based on column values.
- HAVING is used to **filter groups** after grouping (like WHERE but applied after aggregation).
- **Objective:** Get students with **GPA greater than 4.0**.

- **Query:**

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

- **Outcome (students with GPA > 4.0):**
 - 103 Smith 4.5
 - 104 Scott 4.2
 - 106 Alex 4.5
- 107 David 4.2

✓ Key Points:

- Use avg() to calculate average values.
- Use count(*) to count total rows.
- Use GROUP BY to organize data into groups.
- Use HAVING to filter based on aggregated/grouped results.

9.6 RCFile Implementation

- **RCFile (Record Columnar File)** is a **data placement structure** in Hive.
- It determines how to **store relational tables** on computer clusters.
- RCFile stores data in a **column-oriented format**, which improves performance for analytical queries.

Objective:

To work with the **RCFILE Format**.

Steps (Act):

1. **Create a table in RCFILE format:**

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

2. Insert data from the STUDENT table:

```
INSERT OVERWRITE TABLE STUDENT_RC SELECT * FROM STUDENT;
```

3. Query example (SUM of GPA):

```
SELECT SUM(gpa) FROM STUDENT_RC;
```

Outcome:

- Table STUDENT_RC created successfully in **RCFILE format**.
- Data from STUDENT table was inserted into it.
- Query returned **SUM(gpa) = 38.39999651853027**.

✅ Key Point:

RCFile stores data in **column-oriented manner**, making it more efficient for queries that process only a few columns of large datasets.

9.8 Use Defined Functions

- Hive allows users to create **custom functions** using **UDF**.
- These functions can be used when built-in Hive functions are not enough.

Objective: Write a Hive function to convert the values of a field to **uppercase**.

Steps (Act):

Java Code for UDF:

```
package com.example.hive.udf;

import org.apache.hadoop.hive.ql.exec.Description;
import org.apache.hadoop.hive.ql.exec.UDF;

@Description(name="SimpleUDFExample")
public final class MyLowerCase extends UDF {
    public String evaluate(final String word) {
        return word.toLowerCase();
    }
}
```

Here, the class defines a **UDF** method to convert text.

2. **Convert Java program into a JAR file.**
3. **Add JAR in Hive:**
ADD JAR /root/hivedemos/UpperCase.jar;
4. **Create Temporary Function:**
CREATE TEMPORARY FUNCTION touppercase
5. **AS 'com.example.hive.udf.MyUpperCase';**
6. **Use the Function in Query:**
SELECT touppercase(name) FROM STUDENT;

Outcome:

Names in the STUDENT table are converted to **uppercase**.

Example Result:

JACK
JOHN
SCOTT
ALEX
DAVID
JAMES
JOSH

(10 rows returned).

✓ Key Point:

- **UDFs extend Hive's functionality.**
- Custom logic (like uppercase, lowercase, formatting, etc.) can be written in **Java**, packaged as JAR, and then used in Hive queries.

Introduction to Pig

10.1 What is Pig?

- Pig is a platform for analyzing data.
- It is an alternative to **MapReduce Programming**.
- Pig was developed as a research project at **Yahoo**.

10.1.1 Key Features of Pig

1. Pig gives an **engine** to execute **data flows** (how data moves and is processed).
2. It uses a special language called **Pig Latin** to write data flows.
3. Pig Latin supports many operations like **join, filter, sort, group, etc.**
4. Pig also supports **User Defined Functions (UDFs)** for reading, processing, and writing data.

10.2 The Anatomy of Pig

Pig has 3 main parts:

1. **Pig Latin** → A scripting language used to write programs.
2. **Grunt Shell** → An interactive shell to type Pig commands.
3. **Pig Interpreter & Execution Engine** → Parses scripts, optimizes, and sends jobs to Hadoop.

👉 Example flow:

- Write a Pig Latin script → Interpreter converts it → Runs as **MapReduce jobs** in Hadoop.

10.3 Pig on Hadoop

- Pig works on top of **HDFS and MapReduce**.
- Pig reads input from **HDFS** (or local files, HBase, etc.), processes data, and writes results back.
- Pig supports:
 1. **HDFS commands**
 2. **UNIX shell commands**
 3. **Relational operators**
 4. **Positional parameters**
 5. **Mathematical functions**
 6. **Custom functions**
 - 7.

10.4 Pig Philosophy

1. **Pigs Eat Anything** → Can process structured & unstructured data.
2. **Pigs Live Anywhere** → Works with HDFS, local file system, or other sources.
3. **Pigs are Domestic Animals** → Developers can extend Pig with UDFs.
4. **Pigs Fly** → Pig processes data **quickly**.

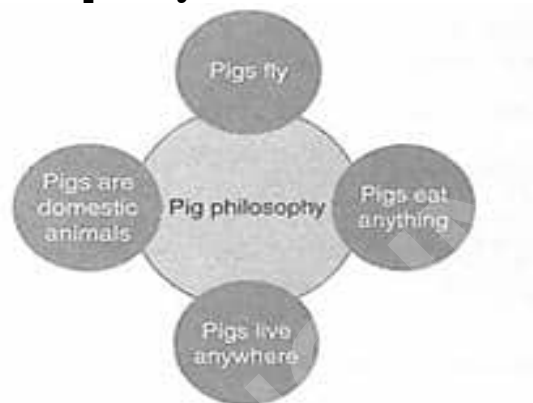


Figure 10.2 Pig philosophy.

10.5 Use Case of Pig ETL Processing (Extract, Transform, Load)

- Pig is widely used for **ETL processing**.
- It extracts data from sources like ERP, accounting systems, flat files, etc.
- It **validates, fixes errors, removes duplicates, encodes values**, and loads data into the warehouse.

◆ Pig Latin Overview

- Pig Latin is the language of Pig.
- Statements process data step by step:
 1. **LOAD** – Reads data from file system.
 2. **Transformations** – Apply filters, functions, etc.
 3. **DUMP/STORE** – Display or save results.

Example:

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

◇ Pig Latin Rules

- **Keywords** are reserved (cannot be used as names).
- **Identifiers** must start with a letter and can include letters, numbers, and underscores.
- **Comments:** -- for single-line, /* ... */ for multi-line.
- **Case Sensitivity:**
 - Keywords are **not case-sensitive**.
 - Relations and functions are **case-sensitive**.

◇ Operators in Pig Latin

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

Note:

Pig is a high-level data flow platform on Hadoop that uses Pig Latin to simplify ETL and data processing, making it easier than writing raw MapReduce code.

10.7 Data Types in Pig

1. Simple Data Types

(Fields of unspecified types default to bytearray in Pig.)

Name	Description
Int	Whole numbers
Long	Large whole numbers
Float	Decimals
Double	Very precise decimals
Chararray	Text strings
Bytearray	Raw bytes
Datetime	Date and time values
Boolean	True or False values

Note:

- NULL represents unknown or non-existent values in Pig.

2. Complex Data Types

Name	Description
Tuple	An ordered set of fields. Example: (2,3)
Bag	A collection of tuples. Example: {(2,3),(7,5)}
Map	Key-value pairs (like a dictionary). Example: ['name' #'Alex', 'age' #25]

So, Pig supports **primitive types** (like numbers, strings, boolean, date) and **nested/complex types** (like tuple, bag, map), making it flexible for semi-structured and unstructured data.

10.8 Running Pig

You can run Pig in two modes:

1. **Interactive Mode**
2. **Batch Mode**

1. Interactive Mode

- Run Pig interactively using the **grunt shell**.
- Command:
pig
- Once grunt shell is open, you can type Pig Latin statements directly

Example:

```
grunt> A = load '/pigdemo/student.tsv' as (rollno, name, gpa);
```

```
grunt> DUMP A;
```

- A loads student data from HDFS.
- DUMP A; displays the data in the console.

2. Batch Mode

- You write Pig Latin statements in a file with **.pig extension**.
- Run the script using:
pig scriptname.pig
- Useful for executing large jobs or automating tasks.

10.9 Execution Modes of Pig

Pig can run in **two modes**:

1. Local Mode

- Files are in the **local file system**.
- Syntax:
- `pig -x local filename`

2. MapReduce Mode (Default)

- Requires **Hadoop Cluster** for reading/writing files.
- Syntax:
- `pig filename`

◆ 10.10 HDFS Commands in Pig

You can run **HDFS commands** directly from the **grunt shell**.

Example – Create a directory:

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

- These commands allow you to **manage files & directories in HDFS** while working with Pig.

10.11 Relational Operators

◆ 10.11.1 FILTER

- **Purpose:** The FILTER operator is used to select tuples (rows) from a relation based on given conditions.
- **Objective:** Find the students whose GPA is greater than 4.0.
- **Input:**
- Student(rollno:int, name:chararray, gpa:float)
- **Pig Latin Script:**
- `A = LOAD '/pigdemo/student.tsv' AS (rollno:int, name:chararray, gpa:float);`
- `B = FILTER A BY gpa > 4.0;`
- `DUMP B;`
- **Output:**
- (1003, Smith, 4.5)

-
- (1004, Scott, 4.2)

◇ 10.11.2 FOREACH

- **Purpose:** The FOREACH operator is used for data transformation, especially column-wise operations.
- **Objective:** Display the names of all students in **uppercase**.
- **Input:**
 - Student(rollno:int, name:chararray, gpa:float)
- **Pig Latin Script:**
 - A = LOAD '/pigdemo/student.tsv' AS (rollno:int, name:chararray, gpa:float);
 - B = FOREACH A GENERATE UPPER(name);
 - DUMP B;
- **Output:**
 - (JOHN)
 - (JACK)
 - (SMITH)
 - (SCOTT)
 - (JOSH)

10.11.3 GROUP

The **GROUP** operator is used to group data.

Objective:

Group tuples of students based on their GPA.

Input:

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

Act:

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

B = GROUP A BY gpa;

DUMP B;

Output:

(3.0, {(1001, John, 3.0)})
(3.5, {(1005, Joshi, 3.5)})
(4.0, {(1002, Jack, 4.0), (1008, James, 4.0)})
(4.2, {(1007, David, 4.2), (1004, Scott, 4.2)})
(4.5, {(1006, Alex, 4.5), (1003, Smith, 4.5)})

10.11.4 DISTINCT

- The **DISTINCT** operator is used to remove duplicate tuples.
In Fig, **DISTINCT** works on the entire tuple and **not** on individual fields.

Objective:

Remove duplicate tuples of students.

Input:

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

Data:

1001	John	3.0
1002	Jack	4.0
1003	Smith	4.5
1004	Scott	4.2
1005	Joshi	3.5
1006	Alex	4.5
1007	David	4.2
1008	James	4.0
1001	John	3.0
1005	Joshi	3.5

Act:

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

B = DISTINCT A;

DUMP B;

Output:

(1001, John, 3.0)
(1002, Jack, 4.0)
(1003, Smith, 4.5)
(1004, Scott, 4.2)

(1005, Joshi, 3.5)
(1006, Alex, 4.5)
(1007, David, 4.2)
(1008, James, 4.0)

10.11.5 LIMIT

The **LIMIT** operator is used to restrict the number of output tuples.

Objective:

Display the first 3 tuples from the *student* relation.

Input:

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

Act:

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

B = LIMIT A 3;

DUMP B;

Output:

(1001, John, 3.0)
(1002, Jack, 4.0)
(1003, Smith, 4.5)

10.11.6 ORDER BY

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

Objective:

Display the names of the students in ascending order.

Input:

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

Act:

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

B = ORDER A BY name;

DUMP B;

Output:

(1006, Alex, 4.5)
(1007, David, 4.2)
(1002, Jack, 4.0)
(1008, James, 4.0)
(1001, John, 3.0)
(1001, John, 3.0)
(1005, Joshi, 3.5)
(1005, Joshi, 3.5)
(1004, Scott, 4.2)
(1003, Smith, 4.5)

10.11.7 JOIN

The **JOIN** operator is used to join two or more relations based on a common field. It always performs an **inner join**.

Objective:

To join two relations, namely *student* and *department*, based on the values 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;
```

Output:

```
(1001, John, 3.0, 1001, 10, B.E.)
(1002, Jack, 4.0, 1002, 11, M.Tech)
(1003, Smith, 4.5, 1003, 12, Ph.D)
(1004, Scott, 4.2, 1004, 13, M.Tech)
(1005, Joshi, 3.5, 1005, 20, MBA)
(1006, Alex, 4.5, 1006, 14, M.Tech)
(1007, David, 4.2, 1007, 15, MCA)
(1008, James, 4.0, 1008, 12, M.Tech)
```

10.11.8 UNION

The **UNION** operator is used to merge the contents of two relations.

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, gpa);
B = load '/pigdemo/department.tsv' as (rollno, deptno, deptname);
C = UNION A, B;
STORE C INTO '/pigdemo/uniondemo';
DUMP B;
```

Output Explanation:

- The STORE command saves the merged results into the given path.
- The output is divided into two parts:
 - part-m-00000 → contains *student* content.
 - part-m-00001 → contains *department* content.

Example Output (student part):

```
1001 John 3.0
1002 Jack 4.0
1003 Smith 4.5
1004 Scott 4.2
1005 Joshi 3.5
1006 Alex 4.5
1007 David 4.2
1008 James 4.0
```

Example Output (department part):

```
1001 10 B.E
1002 11 M.Tech
1003 12 Ph.D
1004 13 M.Tech
1005 20 MBA
1006 14 M.Tech
1007 15 MCA
1008 12 M.Tech
```

The UNION operator here simply stacks the contents of both relations together (like appending), but since schemas are different, the stored output is split into separate files.

10.11.9 SPLIT

The **SPLIT** operator is used to partition a relation into two or more relations based on conditions.

Objective:

Partition a relation based on the **GPA** of students.

- If GPA = 4.0 → place it into relation **X**
- If GPA ≤ 4.0 → place it into relation **Y**

Input:

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

Act:

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

Output:

- **Relation X (students with GPA exactly 4.0):**

```
(1002, Jack, 4.0)  
(1008, James, 4.0)
```

- **Relation Y (students with GPA $\leq$ 4.0):**

```
(1001, John, 3.0)  
(1002, Jack, 4.0)  
(1003, Smith, 4.0)  
(1004, Scott, 3.2)  
(1008, James, 4.0)  
(1005, Joshi, 3.5)
```

10.11.10 SAMPLE

The **SAMPLE** operator is used to select a random sample of data from a relation based on the specified sample size.

Objective:

To demonstrate 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;
```

Output:

- A random 1% sample of the data is returned.
- The actual output will vary each time since sampling is random.

10.12 EVAL FUNCTION

Eval functions are used to perform operations like **average, maximum, minimum, and sum** on numeric values in Pig.

10.12.1 AVG

AVG computes the **average of numeric values** in a single column bag.

Objective:

To calculate the average marks for each student.

Input:

Student (studname:chararray, marks:int)

Act:

```
A = load '/pigdemo/student.csv' USING PigStorage(',')
  as (studname:chararray, marks:int);
```

```
B = GROUP A BY studname;
```

```
C = FOREACH B GENERATE A.studname, AVG(A.marks);
```

```
DUMP C;
```

Output:

```
((Jack),(Jack),(Jack),(Jack)}, 39.75)
```

```
((John),(John),(John),(John)}, 39.0)
```

👉 This shows each student's **average marks**.

Example:

- Jack's average = 39.75
- John's average = 39.0

Note: You must use PigStorage for files other than .tsv.

10.12.2 MAX

MAX computes the **maximum value** of numeric values in a single column bag.

Objective:

To calculate the maximum marks for each student.

Input:

Student (studname:chararray, marks:int)

Act:

```
A = load '/pigdemo/student.csv' USING PigStorage(',')
  as (studname:chararray, marks:int);
B = GROUP A BY studname;
C = FOREACH B GENERATE A.studname, MAX(A.marks);
DUMP C;
```

Output:

```
{{(Jack),(Jack),(Jack),(Jack)}, 46}
```

```
{{(John),(John),(John),(John)}, 48}
```

This shows each student's **highest marks**.

Example:

- Jack's maximum marks = 46
- John's maximum marks = 48

Note: Similarly, you can also apply **MIN** (minimum) and **SUM** (total) functions.

AVG → calculates average marks of each student.

MAX → calculates maximum marks of each student.

10.13 COMPLEX DATA TYPES

Pig provides complex data types like **Tuple**, **Map**, and **Bag** to handle structured data.

10.13.1 TUPLE

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

Objective:

To use the complex data type *Tuple* to load data.

Input (studentdata.tsv):

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

Act:

```
A = LOAD '/root/pigdemo/studentdata.tsv'
  AS (t1:tuple(t1a:chararray, t1b:int),
      t2:tuple(t2a:chararray, t2b:int));

B = FOREACH A GENERATE t1.t1a, t1.t1b, t2.$0, t2.$1;

DUMP B;
```

Output:

(John,12,Jack,13)

(James,7,Joseph,5)

(Smith,8,Scott,12)

Note:

- t1 and t2 are **tuples**.
- Fields inside tuples are accessed as t1.t1a or by **positional notation** like \$0 (first field).

Note: Positional notation starts with \$0.

10.13.2 MAP

A **MAP** is a **key-value pair collection** in Pig.

Objective

To show how to use the complex data type *map*.

Input (studentcity.tsv):

John [city#Bangalore]

Jack [city#Pune]

James [city#Chennai]

Here,

- **Key** = city
- **Value** = City name (Bangalore, Pune, Chennai)

Pig Script

```
A = load '/root/pigdemo/studentcity.tsv'
```

```
Using PigStorage('\t')
```

```
as (studname:chararray, m:map(chararray));
```

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

```
DUMP B;
```

Output

(Bangalore)

(Pune)

(Chennai)

✅ Explanation

- The second column is stored as a **map** (m).
- In Pig, to access a value from a map, you use **m#'key'**.
Example: m#'city' → returns the value of the key "city".

So for:

- John → city = Bangalore
- Jack → city = Pune
- James → city = Chennai

10.14 PIGGY BANK

Piggy Bank is a collection of **user-contributed functions (UDFs)** for Pig. Users can use these functions in Pig Latin scripts or even add their own.

Objective

To use **Piggy Bank string UPPER function**.

Input (student.tsv):

rollno:int, name:chararray, gpa:float

Pig Script

```
-- Step 1: Register the Piggy Bank JAR
register '/root/pigdemos/piggybank-0.12.0.jar';

-- Step 2: Load the dataset
A = load '/pigdemo/student.tsv'
  as (rollno:int, name:chararray, gpa:float);

-- Step 3: Apply UPPER function from Piggy Bank
upper = foreach A generate
  org.apache.pig.piggybank.evaluation.string.UPPER(name);

-- Step 4: Display results
DUMP upper;
```

Output

```
(JACK)
(SMITH)
(SCOTT)
(JAMES)
(DAVID)
(JOHN)
(JOSH)
```

Explanation

- The **Piggy Bank JAR** is registered first with register.
- The **UPPER()** function from Piggy Bank converts all student names into **uppercase**.
- Output shows names in uppercase.

Note: Always register the piggybank.jar before using its functions in Pig scripts.

10.15 USER-DEFINED FUNCTIONS (UDFs)

Pig allows you to create **custom functions** (in Java, Python, etc.) for complex analysis.

Objective

To depict how a **user-defined function (UDF)** works.

Java Code (UPPER function)

This code converts names into uppercase:

```
package myudfs;

import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;
import org.apache.pig.impl.util.WrappedIOException;

public class UPPER extends EvalFunc<String> {
    public String exec(Tuple input) throws IOException {
        if (input == null || input.size() == 0)
            return null;
        try {
            String str = (String) input.get(0);
            return str.toUpperCase();
        } catch (Exception e) {
            throw WrappedIOException.wrap(
                "Caught exception processing input row ", e);
        }
    }
}
```

```
}  
}  
}
```

Steps

1. Save the above Java code, compile it, and package it into a **JAR file** (e.g., myudfs.jar).
2. Register the JAR in Pig using register.
3. Use the custom UDF in your Pig script.

Input (student.tsv)

rollno:int, name:chararray, gpa:float

Pig Script

```
-- Step 1: Register custom UDF JAR  
register /root/pigdemos/myudfs.jar;  
  
-- Step 2: Load dataset  
A = load '/pigdemo/student.tsv'  
    as (rollno:int, name:chararray, gpa:float);  
  
-- Step 3: Apply custom UDF  
B = FOREACH A GENERATE myudfs.UPPER(name);  
  
-- Step 4: Show results  
DUMP B;
```

Output

```
(JACK)  
(SMITH)  
(SCOTT)  
(JAMES)
```

(DAVID)

(JOHN)

(JOSH)

✅ Explanation

- This UDF behaves just like the Piggy Bank UPPER function, but here **you created it yourself**.
- The function extends EvalFunc and overrides the exec() method.
- It converts each student's name to **uppercase**.

📌 **Note:** Always remember to:

- Compile your Java class.
- Create a JAR.
- Register it before using in Pig.

10.22 Pig versus Hive

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

✓ Key Points

- **Pig** is more programming-oriented, designed for developers handling **semi-structured data**.
- **Hive** is more analyst/reporting-oriented, built for **structured data** using SQL-like queries.
- Pig requires **explicit schema definitions**, whereas Hive infers schema **implicitly**.
- Pig has **web interface support** and **partition support**, while Hive does not.
- Both support **UDFs, Joins, Sorting, and Shell access**.