# Apache Pig & Hive

## Data Analytics on Hadoop

Dr Fahimeh Jafari

f.jafari@uel.ac.uk

https://youtu.be/S_h0xhb8cvA


CN7031 – Lecture 4

University of East London

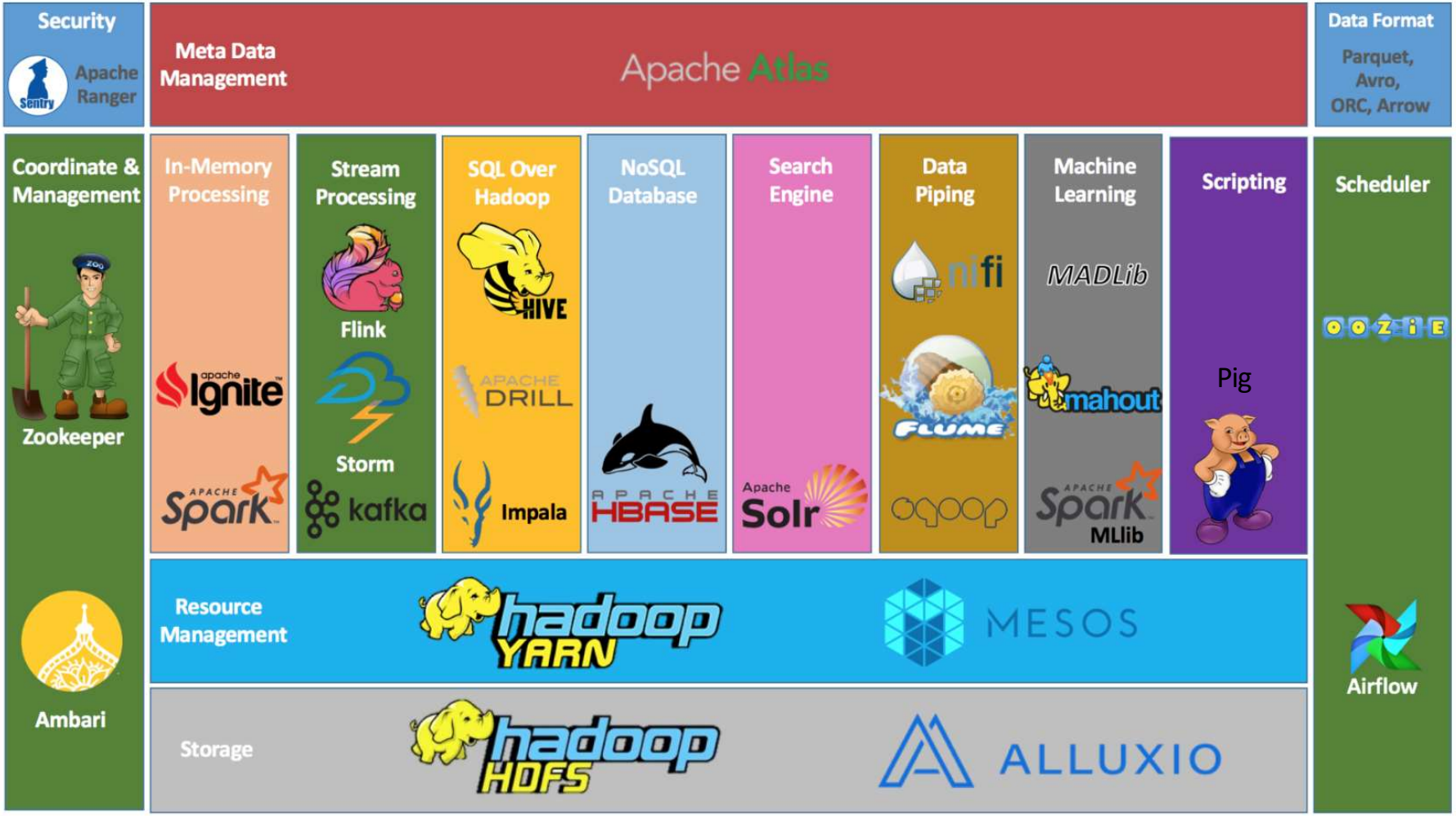
# Outline

- ZooKeeper and Oozie
- What is Apache Pig?
- Pig Operations
- What is Apache Hive?
- Partitioning in Hive for querying massive data
- Hive Queries

# Learning Outcomes

- Be able to explain Pig and Hive
- Understand the differences between Pig and Hive
- Understand Pig operations
- Be able to write HIVE queries
- Understand optimization technique for HIVE Querying
- Understand the functionalities of Zookeeper and Oozie

# Hadoop Ecosystem




University of East London

# Distributed Systems - Requirements

The requirements in the distributed systems

1. **Networking**: Workstations can be either physically closed (LAN) or geographically distant (MAN or WAN)
2. **Scalability**: Systems can be easily expanded by adding more machines
3. **Redundancy & Reliability**: if a node is unavailable due to the system failure, the copy version of the machine should work
4. **Availability**: the system must be up 100%

# What is ZooKeeper?


## Zookeeper = Coordination Service

- It is a coordinator (written in Java) for your Hadoop cluster to enable highly reliable distributed coordination. (https://zookeeper.apache.org/)
- It is basically keeping track of information that must be synchronised across your cluster.
- **Purpose:** Zookeeper is a centralised service for maintaining configuration information, naming, providing distributed synchronisation, and offering group services.
- **Key Features:**
  - Provides distributed synchronisation.
  - Acts as a registry for distributed services.
  - Helps in managing metadata.
  - Facilitates fault tolerance in distributed systems.

# ZooKeeper Roles

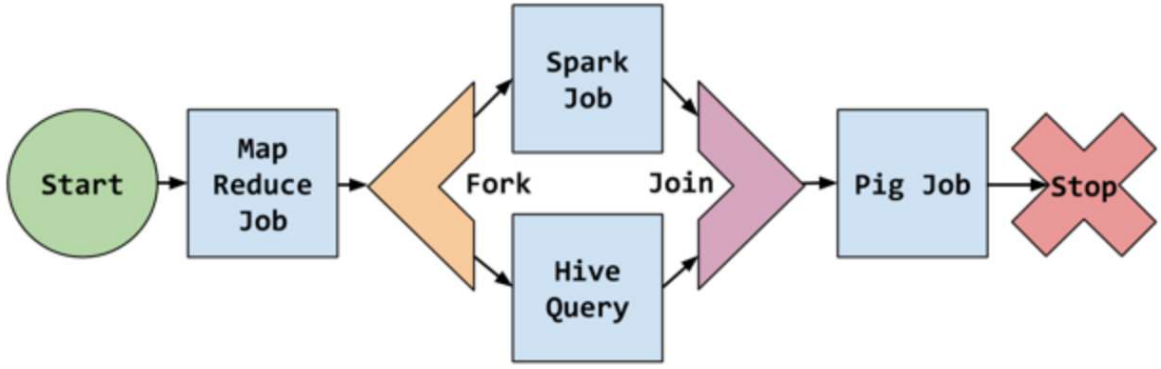

- **Distributed Coordination:** Provides mechanisms for synchronizing processes across distributed nodes, such as leader election and distributed locking. This coordination ensures that tasks are managed correctly across multiple nodes.
- **Configuration Management:** Centralizes configuration information for distributed applications, making it easy for different nodes to access and update configuration data in a consistent manner. This helps maintain uniform settings across the cluster.
- **Naming Service:** Acts as a directory of services where each service is identified by a unique name. It allows clients to look up information about services, similar to a DNS for distributed systems.
- **Leader Election:** Manages the election of leader nodes in a distributed cluster, ensuring that there is always a single leader coordinating tasks. This role is crucial for master-slave architectures where a leader node is needed for task delegation.

# ZooKeeper Roles


- **Failure Detection and Recovery:** Monitors the health of nodes in the cluster and helps detect failures. When a node fails, Zookeeper can trigger recovery actions, such as reassigning tasks or electing a new leader.
- **Metadata Management:** Stores metadata required for operating distributed systems, such as configuration data, state information, and other coordination-related data. This metadata is kept up-to-date and consistent across all nodes.
- **Distributed Locking:** Implements distributed locks to control access to shared resources, preventing conflicts or data corruption when multiple nodes try to access the same resource simultaneously.
- **High Availability and Fault Tolerance:** Ensures continuous operation of distributed systems by providing mechanisms to handle node failures and recover from faults without significant downtime.

# What is Oozie?


- Usually, we run several jobs for processing massive amount of data in the cluster. How do we can schedule them?
- **Oozie** is a workflow scheduler system to manage Apache Hadoop jobs (such as MapReduce, Streaming, Pig, Hive, Sqoop, HDFS operations).
- Oozie is a scalable, reliable and extensible system.
- It executes and monitors the workflows in Hadoop and performs a periodic scheduling of workflows.



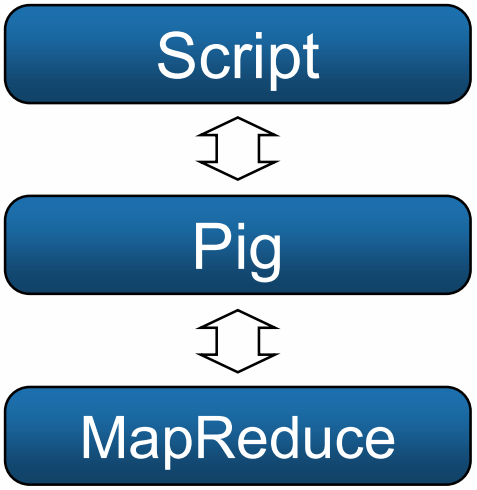
# Why Hive and Pig?


- Although MapReduce is very powerful, it can also be complex to master
- Many organizations have business or data analysts who are skilled at writing SQL queries/ scripting languages, but not at writing Java code
- Hive and Pig are two projects which evolved separately to help such people analyze huge amounts of data via MapReduce
- Hive was initially developed at Facebook, Pig at Yahoo!

# Pig – Initiated by YAHOO!

- Framework for analyzing large un-structured and semi-structured data on top of Hadoop.
- Yahoo worked on Pig to facilitate application deployment on Hadoop, mainly on unstructured data.

  - A high-level scripting language (Pig Latin)
  - Process data one step at a time
  - Simple to write MapReduce program
  - Easy understand
  - Easy debug



# Pig Operations

- Loading data
  - `LOAD` loads input data
  - `Lines=LOAD 'input/access.log' AS (line: chararray);`
- Projection
  - `FOREACH … GENERTE …` (similar to `SELECT`)
  - takes a set of expressions and applies them to every record.
- Grouping
  - GROUP collects together records with the same key
- Dump/Store
  - DUMP displays results to screen, STORE save results to file system
- Aggregation
  - `AVG, COUNT, MAX, MIN, SUM`

# Pig Operations Examples

1) `PigStorage` (field_delimiter): loads/stores relations using field-delimited text format

```
(1811 John 18  4.0F)
(1823 Mary  19  3.8F)
(1845 Bill  20  3.9F)
```

```
students = load 'student.txt' using PigStorage('\t')
       as (studentid: int, name:chararray, age:int, gpa:double);
```

2) `Foreach…Generate`

This statement iterates over the members of a bag

```
studentid = FOREACH  students  GENERATE  studentid, name;
```

- The result of a Foreach is another bag
- Elements are named as in the input bag

# Pig Operations Examples

3) `Group…By` groups the data in one or more relations

```
A = GROUP students BY age;
```

4) `DUMP` displays output results, will always trigger execution

```
DUMP A;
```

5) `STORE` Pig will parse entire script prior to writing for efficiency purposes

```
STORE A INTO “output/c”
```

# Pig Operations Examples

6) COUNT computes the number of elements in a bag. This function requires a preceding GROUP BY statement for group counts.

```
A = GROUP students BY age;
```

```
X = FOREACH A GENERATE COUNT(students);
```

7) ORDER

- sorts a relation based on one or more fields.
- In Pig, relations are unordered. If you order relation A to produce relation X relations A and X still contain the same elements.

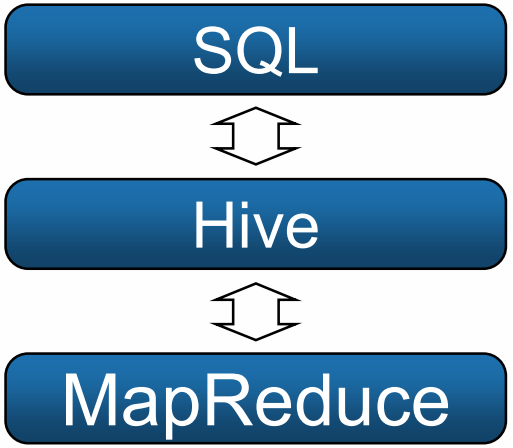
```
Student-Order = ORDER students BY gpa DESC;
```

# What is Hive?

Facebook started working on deploying warehouse solutions on Hadoop that resulted in Hive.

A data warehouse infrastructure tool to process structured data in Hadoop. It resides on top of Hadoop to summarize Big Data and makes querying and analyzing easy.

- Uses Map-Reduce for execution
- HDFS for storage




Hive is **not**

- A relational database
- A design for OnLine Transaction Processing (OLTP)
- A language for real-time queries and row-level updates

# What is Hive?

- Hive is not a relational database: Hive does not function as a traditional RDBMS like MySQL, PostgreSQL, or Oracle. Although it uses a SQL-like language (HiveQL), it is not built for the same type of structured, transactional operations that relational databases support. Hive is primarily used for querying and analyzing large datasets stored in distributed storage systems like HDFS (Hadoop Distributed File System
- Hive is not designed for Online Transaction Processing (OLTP): Hive is not suitable for OLTP, which involves handling a large number of short online transactions like inserting, updating, or deleting individual records. Hive is optimized for Online Analytical Processing (OLAP), which focuses on complex queries and data analysis over large datasets, rather than the quick and frequent data manipulation typical in OLTP
- Hive is not a language for real-time queries and row-level updates: Hive is not designed to support real-time query execution or fine-grained row-level updates. Queries in Hive are generally batch-oriented and may take longer to process compared to systems optimized for real-time analytics. Hive is more suitable for batch processing of large datasets, where queries can take minutes or even hours. Additionally, Hive works with large datasets in a read-only manner rather than updating individual rows in real time.

# Sqoop vs Pig vs hive

| | Hive | Pig |
|---|---|---|
| Language | HiveQL (SQL-like) | Pig Latin, a scripting language |
| Schema | Table definitions that are stored in a metastore | A schema is optionally defined at runtime |
| Programmait Access | JDBC, ODBC | PigServer |

- Sqoop: It is used to import and export data to and from between HDFS and RDBMS.
- Pig: It is a procedural language platform used to develop a script for MapReduce operations.
- Hive: It is a platform used to develop SQL type scripts to do MapReduce operations.

# Hive Syntax: Create a Database

HIVE Syntax for creating databases:

```
CREATE DATABASE|SCHEMA [IF NOT EXISTS] <database name>
```

A database in Hive is a **namespace** or a collection of tables.

```
hive> CREATE DATABASE [IF NOT EXISTS] userdb;
```

```
hive> CREATE SCHEMA userdb;
```

# Hive Syntax: Create a Table

Syntax:

```
CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.] table_name

[(col_name data_type [COMMENT col_comment], ...)]
[COMMENT table_comment]
[ROW FORMAT row_format]
[STORED AS file_format]
```

Example:

```
hive> CREATE TABLE IF NOT EXISTS employee ( eid int, name String,
salary String, destination String)
COMMENT ‘Employee details’
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ‘\t’
LINES TERMINATED BY ‘\n’
STORED AS TEXTFILE;
```

# Hive Syntax: Load Data into a Table

Generally, after creating a table in SQL, we can insert data using the Insert statement. But in Hive, we can insert data using the LOAD DATA statement.

While inserting data into Hive, it is better to use LOAD DATA to store bulk records.

There are two ways to load data:

- From local file system
- From Hadoop file system

# Hive Syntax: Load Data into a Table

Syntax:

```
LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO TABLE tablename
[PARTITION (partcol1=val1, partcol2=val2 ...)]
```

- *LOCAL* is identifier to specify the local path. It is optional.
- *OVERWRITE* is optional to overwrite the data in the table.
- *PARTITION* is optional.

Example:

```
hive> LOAD DATA LOCAL INPATH '/home/user/sample.txt'
OVERWRITE INTO TABLE employee;
```

# Hive Syntax: Alter a Table

Syntax:

```
ALTER TABLE name RENAME TO new_name
ALTER TABLE name ADD COLUMNS (col_spec[, col_spec ...])
ALTER TABLE name DROP [COLUMN] column_name
ALTER TABLE name CHANGE column_name new_name new_type
ALTER TABLE name REPLACE COLUMNS (col_spec[, col_spec ...])
```

Example:

```
hive> ALTER TABLE employee RENAME TO emp;
```

```
hive> ALTER TABLE employee CHANGE name ename String;
hive> ALTER TABLE employee CHANGE salary salary Double;
```

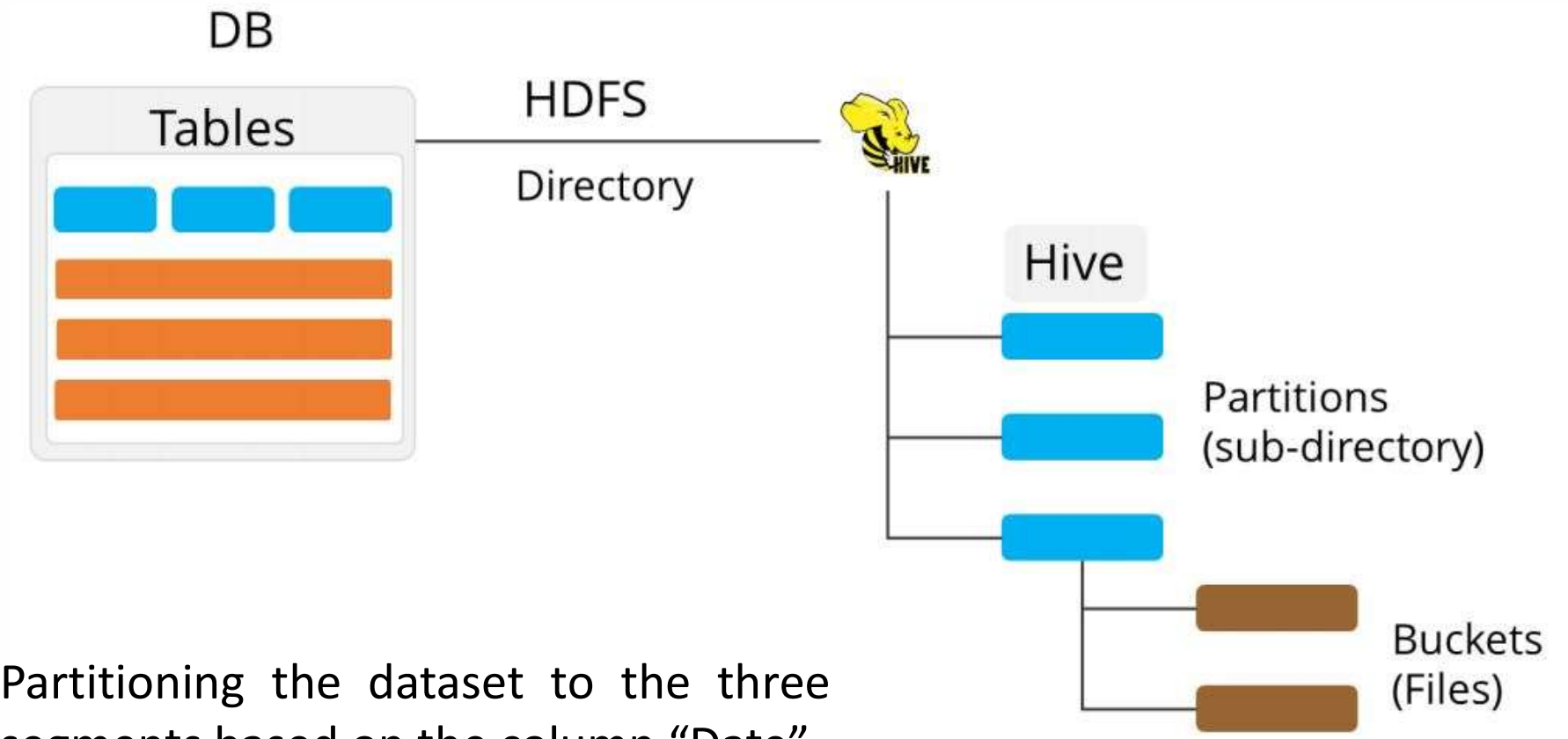
```
hive> ALTER TABLE employee ADD COLUMNS (
dept STRING COMMENT 'Department name');
```

# Partitioning in Hive

- Apache Hive converts the SQL queries into MapReduce jobs. When we submit a SQL query, Hive read the entire dataset. So, it becomes inefficient to run MapReduce jobs over a large table. Thus, this is resolved by creating partitions in tables.
- Partitioning is a way of dividing a table into related parts based on the values of particular columns, such as date and city. Each table in the hive can have one or more partition keys to identify a particular partition.
- Using partition, it is easy to query a portion of the data.
- Tables or partitions are sub-divided into buckets, to provide extra structure to the data that may be used for more efficient querying.

```
CREATE TABLE table_name (column1 type, column2 type, …)
PARTITIONED BY (partition1 type, …);
```

# Partitioning in Hive




Partitioning the dataset to the three segments based on the column “Date”

# Partitioning in Hive: example

| salesperson_id | product_id | date_of_sale |
|---|---|---|
| 12 | 101 | 10-27-2017 |
| 10 | 10010 | 10-27-2017 |
| 111 | 2010 | 10-27-2017 |
| 13 | 222 | 10-28-2017 |
| 15 | 235 | 10-28-2017 |

```
CREATE TABLE Sale (personID int, ProductID int, DateofSale string)
PARTITIONED BY (DateofSale string);
```

**Further studies on partitioning:**

- ✓ Static/Dynamic Partitioning
- ✓ External Partitioned tables
- ✓ Alter Partitions

# Partitioning in Hive

**Advantages:**

Partitioning in Hive distributes execution load horizontally.

In partition faster execution of queries with the low volume of data takes place. For example, search population from Vatican City returns very fast instead of searching entire world population.

**Disadvantages:**

There is the possibility of too many small partition creations- too many directories.

But there some queries like group by on high volume of data take a long time to execute. For example, grouping population of China will take a long time as compared to a grouping of the population in Vatican City.

# Hive Syntax: SELECT … WHERE

Syntax:

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list]
[HAVING having_condition]
[CLUSTER BY col_list | [DISTRIBUTE BY col_list] [SORT BY col_list]]
[LIMIT number];
```

Example:

```
hive> SELECT * FROM employee WHERE salary>30000;
```

# Hive Syntax: Select … Order By

Syntax:

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list]
[HAVING having_condition]
[CLUSTER BY col_list | [DISTRIBUTE BY col_list] [SORT BY col_list]]
[LIMIT number];
```

Example:

```
hive> SELECT Id, Name, Dept FROM employee ORDER BY DEPT;
```

# Hive Syntax: SELECT … GROUP BY

Syntax:

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list]
[HAVING having_condition]
[CLUSTER BY col_list | [DISTRIBUTE BY col_list] [SORT BY col_list]]
[LIMIT number];
```

Example:

```
hive> SELECT Dept,count(*) FROM employee GROUP BY DEPT;
```

# Hive Syntax: Retrieving Information

| Function | Hive |
|---|---|
| Retrieving Information (General) | `SELECT from_columns FROM table WHERE conditions;` |
| Retrieving All Values | `SELECT * FROM table;` |
| Retrieving Some Values | `SELECT * FROM table WHERE rec_name = "value";` |
| Retrieving With Multiple Criteria | `SELECT * FROM TABLE WHERE rec1 = "value1" AND rec2 = "value2";` |
| Retrieving Specific Columns | `SELECT column_name FROM table;` |
| Retrieving Unique Output | `SELECT DISTINCT column_name FROM table;` |
| Sorting | `SELECT col1, col2 FROM table ORDER BY col2;` |
| Sorting Reverse | `SELECT col1, col2 FROM table ORDER BY col2 DESC;` |
| Counting Rows | `SELECT COUNT(*) FROM table;` |
| Grouping With Counting | `SELECT owner, COUNT(*) FROM table GROUP BY owner;` |
| Maximum Value | `SELECT MAX(col_name) AS label FROM table;` |

# Built-in Functions with Hive

| Data type | Function | Description |
|---|---|---|
| BIGINT | round(double a) | Returns the rounded BIGINT value of the double. |
| BIGINT | floor(double a) | Returns the maximum BIGINT value that is equal or less than the double. |
| BIGINT | ceil(double a) | Returns the minimum BIGINT value that is equal or greater than the double. |
| double | rand(), rand(int seed) | Returns a random number (that changes from row to row). Specifying the seed will make sure the generated random number sequence is deterministic. |
| string | concat(string A, string B,...) | Returns the string resulting from concatenating B after A. For example, concat('foo', 'bar') results in 'foobar'. This function accepts an arbitrary number of arguments and returns the concatenation of all of them. |
| string | substr(string A, int start) | Returns the substring of A starting from start position till the end of string A. For example, substr('foobar', 4) results in 'bar'. |
| string | substr(string A, int start, int length) | Returns the substring of A starting from start position with the given length, for example, substr('foobar', 4, 2) results in 'ba'. |
| string | upper(string A) | Returns the string resulting from converting all characters of A to uppercase, for example, upper('fOoBaR') results in 'FOOBAR'. |
| string | ucase(string A) | Same as upper. |
| string | lower(string A) | Returns the string resulting from converting all characters of B to lowercase, for example, lower('fOoBaR') results in 'foobar'. |
| string | lcase(string A) | Same as lower. |
| string | trim(string A) | Returns the string resulting from trimming spaces from both ends of A, for example, trim('foobar ') results in 'foobar'. |
| string | ltrim(string A) | Returns the string resulting from trimming spaces from the beginning (left hand side) of A. For example, ltrim(' foobar ') results in 'foobar '. |
| string | rtrim(string A) | Returns the string resulting from trimming spaces from the end (right hand side) of A. For example, rtrim(' foobar') results in 'foobar'. |
| string | regexp_replace(string A, string B, string C) | Returns the string resulting from replacing all substrings in B that match the Java regular expression syntax (See Java regular expressions syntax) with C. For example, regexp_replace('foobar', 'oo\|ar', ) returns 'fb'. |
| int | size(Map<K.V>) | Returns the number of elements in the map type. |
| int | size(Array<T>) | Returns the number of elements in the array type. |

# Hive Syntax: Create a VIEW

- Views are in the virtual tables and display only selected data.
- Views are generated based on user requirements.
- They are good for security purposes.
- You can save any result set data as a view.
- The usage of view in Hive is same as that of the view in SQL.
- It is a standard RDBMS concept. We can execute all DML operations on a view.

Syntax

```
CREATE VIEW [IF NOT EXISTS] view_name [(column_name [COMMENT column_comment], ...) ]
[COMMENT table_comment]
AS SELECT ...
```

Example

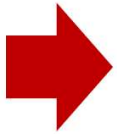
```
hive> CREATE VIEW emp_30000 AS
SELECT * FROM employee
WHERE salary>30000;
```

# Hive Syntax: DROP Database/Table/View

```
DROP DATABASE StatementDROP (DATABASE|SCHEMA) [IF EXISTS] database_name
```

```
hive> DROP DATABASE IF EXISTS userdb;
```

```
DROP TABLE [IF EXISTS] table_name;
```



```
hive> DROP TABLE IF EXISTS employee;
```

```
DROP VIEW view_name
```

```
hive> DROP VIEW emp_30000;
```

# Hive Syntax: JOIN

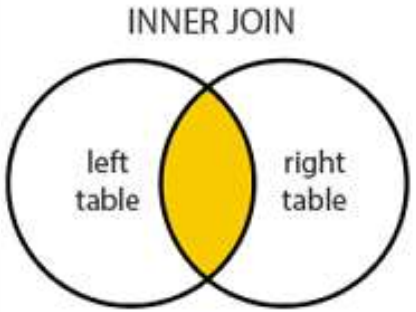
- JOIN is a clause that is used for combining records from two or more tables by using values common to each one.
- A JOIN condition is to be raised using the primary keys and foreign keys of the tables.
- There are different types of joins given as follows:
  - (INNER) JOIN
  - LEFT (OUTER) JOIN
  - RIGHT (OUTER) JOIN
  - FULL (OUTER) JOIN

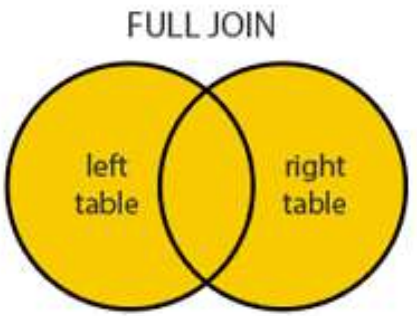
Syntax

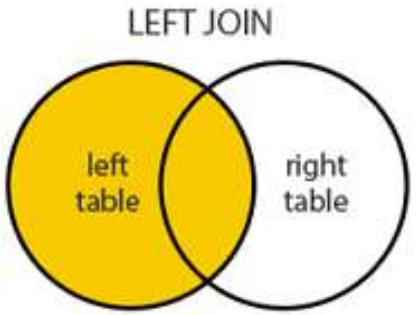
```
join_table:

   table_reference JOIN table_factor [join_condition]
   | table_reference {LEFT|RIGHT|FULL} [OUTER] JOIN table_reference
   join_condition
   | table_reference LEFT SEMI JOIN table_reference join_condition
   | table_reference CROSS JOIN table_reference [join_condition]
```

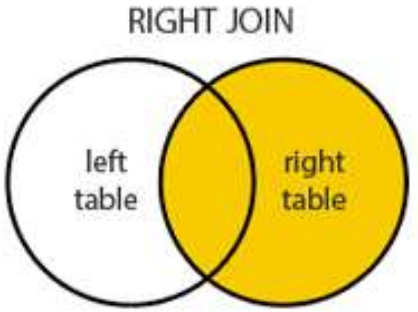
# Hive Syntax: JOIN

- (INNER) JOIN returns all records that have matching values in both tables.
- LEFT (OUTER) JOIN returns all records from the first (left-most) table plus the matching records from the right table, or NULL in case of no matching JOIN predicate.
- RIGHT (OUTER) JOIN returns all records from the second (right-most) table, plus the matched records from the left table, or NULL in case of no matching join predicate.
- FULL OUTER JOIN combines the records of both the left and the right tables that fulfil the JOIN condition. The joined table contains either all the records from both the tables, or fills in NULL values for missing matches on either side.




Source: https://www.dofactory.com/sql/join
Source: https://www.w3schools.com/sql/sql_join.asp

University of East London

# Hive: JOIN Example

CUSTOMERS

| ID | NAME | AGE | ADDRESS | SALARY |
|---|---|---|---|---|
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |

ORDERS

| OID | DATE | CUSTOMER_ID | AMOUNT |
|---|---|---|---|
| 102 | 2009-10-08 00:00:00 | 3 | 3000 |
| 100 | 2009-10-08 00:00:00 | 3 | 1500 |
| 101 | 2009-11-20 00:00:00 | 2 | 1560 |
| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

```
hive> SELECT c.ID, c.NAME, c.AGE, o.AMOUNT
FROM CUSTOMERS c JOIN ORDERS o
ON (c.ID = o.CUSTOMER_ID);
```

| ID | NAME | AGE | AMOUNT |
|---|---|---|---|
| 3 | kaushik | 23 | 3000 |
| 3 | kaushik | 23 | 1500 |
| 2 | Khilan | 25 | 1560 |
| 4 | Chaitali | 25 | 2060 |

# Hive: LEFT (OUTER) JOIN Example

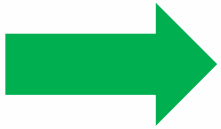
CUSTOMERS

| ID | NAME | AGE | ADDRESS | SALARY |
|---|---|---|---|---|
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |

ORDERS

| OID | DATE | CUSTOMER_ID | AMOUNT |
|---|---|---|---|
| 102 | 2009-10-08 00:00:00 | 3 | 3000 |
| 100 | 2009-10-08 00:00:00 | 3 | 1500 |
| 101 | 2009-11-20 00:00:00 | 2 | 1560 |
| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

```
hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATE
FROM CUSTOMERS c
LEFT OUTER JOIN ORDERS o
ON (c.ID = o.CUSTOMER_ID);
```



| ID | NAME | AMOUNT | DATE |
|---|---|---|---|
| 1 | Ramesh | NULL | NULL |
| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |
| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |
| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |
| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |
| 5 | Hardik | NULL | NULL |
| 6 | Komal | NULL | NULL |
| 7 | Muffy | NULL | NULL |

# Hive: RIGHT (OUTER) JOIN Example

CUSTOMERS

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1  | Ramesh   | 32  | Ahmedabad | 2000.00  |
| 2  | Khilan   | 25  | Delhi     | 1500.00  |
| 3  | kaushik  | 23  | Kota      | 2000.00  |
| 4  | Chaitali | 25  | Mumbai    | 6500.00  |
| 5  | Hardik   | 27  | Bhopal    | 8500.00  |
| 6  | Komal    | 22  | MP        | 4500.00  |
| 7  | Muffy    | 24  | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

ORDERS

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 | 3000   |
| 100 | 2009-10-08 00:00:00 |           3 | 1500   |
| 101 | 2009-11-20 00:00:00 |           2 | 1560   |
| 103 | 2008-05-20 00:00:00 |           4 | 2060   |
+-----+---------------------+-------------+--------+
```

```
notranslate"> hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATE FROM CUSTOMERS c
RIGHT OUTER JOIN ORDERS o ON (c.ID = o.CUSTOMER_ID);
```

```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
| 3    | kaushik  | 3000   | 2009-10-08 00:00:00 |
| 3    | kaushik  | 1500   | 2009-10-08 00:00:00 |
| 2    | Khilan   | 1560   | 2009-11-20 00:00:00 |
| 4    | Chaitali | 2060   | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

# Hive: FULL JOIN Example

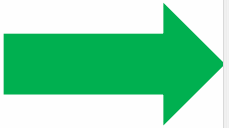
CUSTOMERS

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1  | Ramesh   | 32  | Ahmedabad | 2000.00  |
| 2  | Khilan   | 25  | Delhi     | 1500.00  |
| 3  | kaushik  | 23  | Kota      | 2000.00  |
| 4  | Chaitali | 25  | Mumbai    | 6500.00  |
| 5  | Hardik   | 27  | Bhopal    | 8500.00  |
| 6  | Komal    | 22  | MP        | 4500.00  |
| 7  | Muffy    | 24  | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

ORDERS

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 | 3000   |
| 100 | 2009-10-08 00:00:00 |           3 | 1500   |
| 101 | 2009-11-20 00:00:00 |           2 | 1560   |
| 103 | 2008-05-20 00:00:00 |           4 | 2060   |
+-----+---------------------+-------------+--------+
```

```
hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATE
FROM CUSTOMERS c
FULL OUTER JOIN ORDERS o
ON (c.ID = o.CUSTOMER_ID);
```



```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
| 1    | Ramesh   | NULL   | NULL                |
| 2    | Khilan   | 1560   | 2009-11-20 00:00:00 |
| 3    | kaushik  | 3000   | 2009-10-08 00:00:00 |
| 3    | kaushik  | 1500   | 2009-10-08 00:00:00 |
| 4    | Chaitali | 2060   | 2008-05-20 00:00:00 |
| 5    | Hardik   | NULL   | NULL                |
| 6    | Komal    | NULL   | NULL                |
| 7    | Muffy    | NULL   | NULL                |
| 3    | kaushik  | 3000   | 2009-10-08 00:00:00 |
| 3    | kaushik  | 1500   | 2009-10-08 00:00:00 |
| 2    | Khilan   | 1560   | 2009-11-20 00:00:00 |
| 4    | Chaitali | 2060   | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

# Optimising Hive Queries for Performance and Efficiency

In Apache Hive, query optimization techniques are crucial for improving performance and reducing execution time, especially when dealing with large datasets. Three common strategies to optimise Hive queries include:

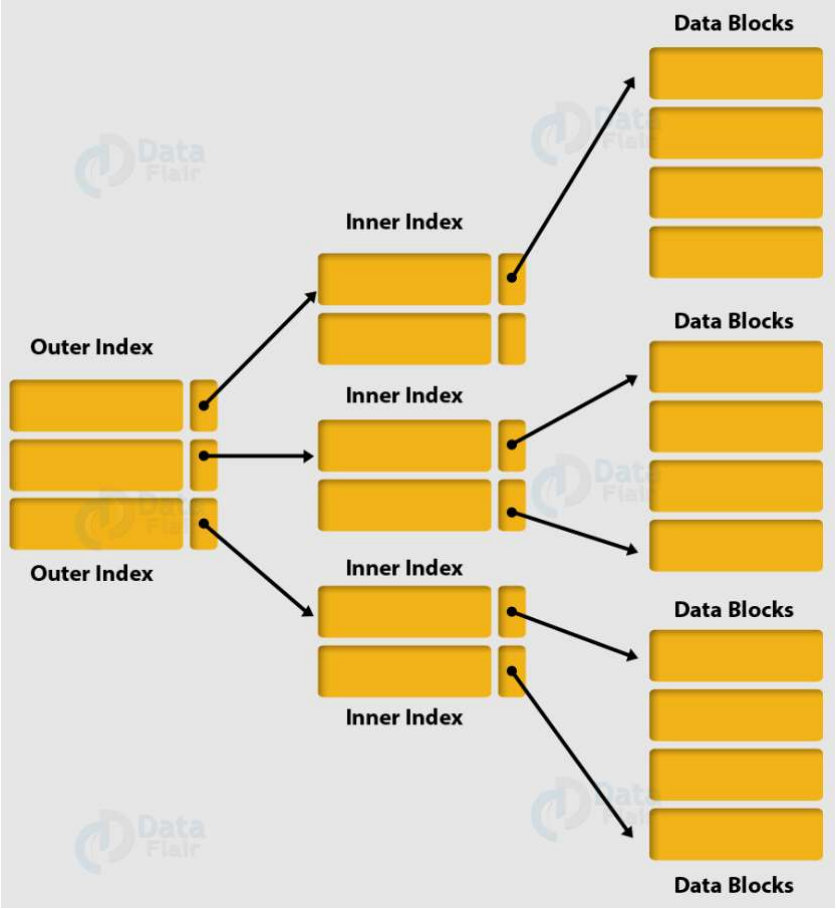
- Indexing
- Partitioning
- Bucketing

# Optimising Hive Queries: Indexing

- **Indexing:** Like traditional databases, indexing in Hive creates a data structure to speed up data retrieval for specific queries. Basically, we are creating the pointer to a particular column name of the table wherever we are creating the Hive index.
- An index can be created on columns frequently used in the `WHERE` clause, such as `salary` or `employee_id`. Indexing helps reduce the amount of data scanned during query execution, making the process faster.

Indexing creates data structures for quick data retrieval based on column values.

**Benefit:** Reduces data scanning and speeds up queries.




University of East London

# Optimising Hive Queries: Partitioning

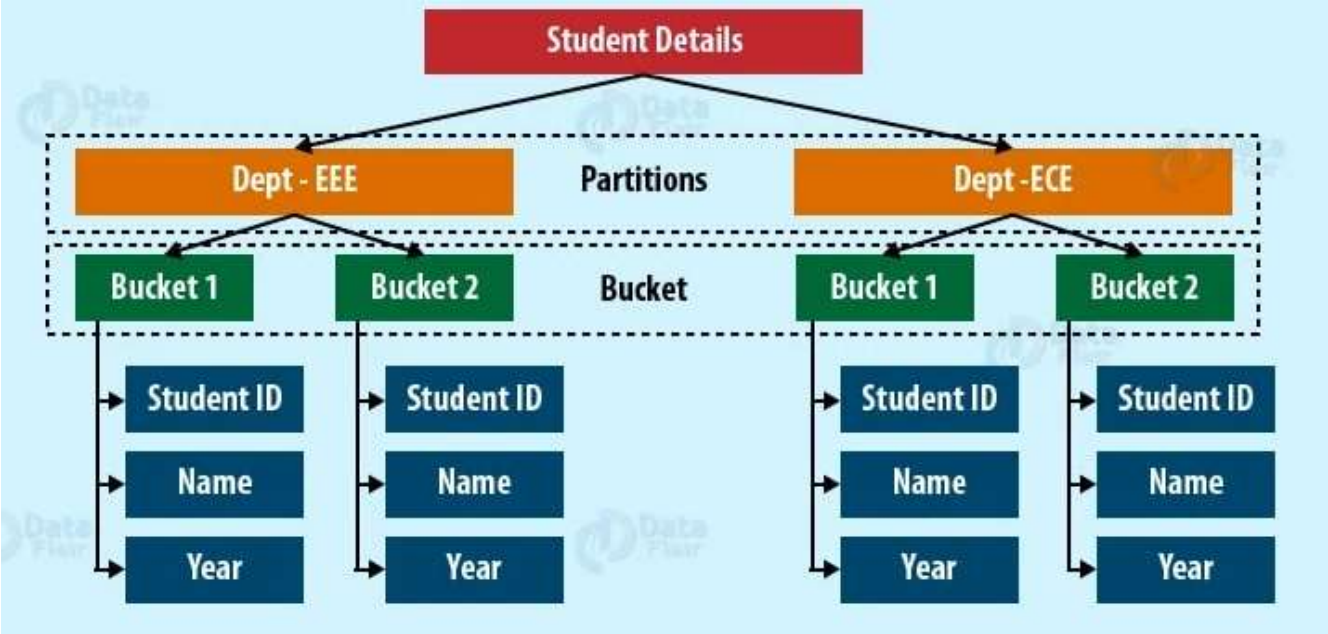
- **Partitioning:** Partitioning involves dividing a table into smaller, more manageable segments based on the values of a specific column, such as `department` or `date`. When a query is executed, Hive can skip entire partitions that are not relevant, significantly reducing the amount of data scanned. This approach is particularly effective for large datasets where queries filter data based on partition columns.

Partitioning divides tables into segments based on a column (`e.g., department`).

**Benefit:** Skips irrelevant data, improving performance.

# Optimising Hive Queries: Bucketing

- **Bucketing:** Bucketing further subdivides the data within each partition into fixed-size clusters, or "buckets," based on the hash of a column's value, such as employee_id. This allows Hive to perform more efficient joins and aggregations, as data with the same bucketed value is stored together. Bucketing works well in combination with partitioning to optimize complex queries.




Bucketing further divides partitions into clusters based on a hash of column values (e.g., `StudentID`).
**Benefit**: Optimises joins and aggregations, enhancing efficiency.

# Summary

- Discussed Zookeeper and Oozie
- Introduced Apache Pig and Hive and their differences
- Practiced Pig Operations
- Practiced HIVE Queries
- Learned Partitioning in HIVE for Querying massive data