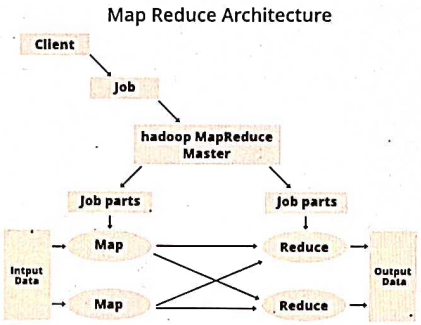
1

Compression techniques in Hadoop are used to reduce the size of data stored on disk and transferred over the network, thereby improving storage efficiency and reducing processing time. Here are some commonly used compression techniques in Hadoop:

1. **Gzip**: Gzip is a widely used compression algorithm that provides good compression ratios. It is suitable for compressing individual files but not optimal for MapReduce tasks in Hadoop due to its lack of splittability. However, it can be used with Hadoop streaming for non-splittable file formats.
2. **Bzip2**: Bzip2 offers better compression ratios compared to Gzip and is suitable for compressing larger files. Bzip2 compressed files are not splittable, but Hadoop can process them efficiently using the `TextInputFormat` class, which reads the compressed files as a whole.
3. **Snappy**: Snappy is a fast compression and decompression algorithm developed by Google. It provides moderate compression ratios but excels in speed, making it suitable for real-time processing and data interchange between Hadoop jobs. Snappy-compressed files are splittable, enabling parallel processing in Hadoop.
4. **LZO**: Lempel-Ziv-Oberhumer (LZO) is another fast compression algorithm optimized for speed and suitable for Hadoop processing. LZO-compressed files are splittable and can be processed efficiently in Hadoop. However, LZO requires native libraries to be installed on Hadoop nodes for compression and decompression, which may require additional setup.
5. **Deflate**: Deflate is the compression algorithm used in the popular ZIP file format. It provides good compression ratios and is supported by Hadoop. However, Deflate-compressed files are not splittable, which can affect the performance of MapReduce jobs.
6. **Zstandard (Zstd)**: Zstandard is a relatively newer compression algorithm that offers a balance between compression ratio and speed. It provides competitive compression ratios with faster compression and decompression speeds compared to some other algorithms. Zstd-compressed files are not splittable, but tools like Hadoop-S3 allow reading and writing Zstd-compressed files in Hadoop.

When choosing a compression technique in Hadoop, it's essential to consider factors such as compression ratio, speed, splittability, and compatibility with existing tools and libraries. Experimentation with different algorithms and configurations can help determine the most suitable compression technique for specific use cases and requirements.

MapReduce Architecture: 2



Components of MapReduce Architecture:

1. **Client:** The MapReduce client is the one who brings the Job to the MapReduce for processing. There can be multiple clients available that continuously send jobs for processing to the Hadoop MapReduce Manager.
2. **Job:** The MapReduce Job is the actual work that the client wanted to do which is comprised of so many smaller tasks that the client wants to process or execute.
3. **Hadoop MapReduce Master:** It divides the particular job into subsequent job-parts.
4. **Job-Parts:** The task or sub-jobs that are obtained after dividing the main job. The result of all the job-parts combined to produce the final output.
5. **Input Data:** The data set that is fed to the MapReduce for processing.
6. **Output Data:** The final result is obtained after the processing.

In **MapReduce**, we have a client. The client will submit the job of a particular size to the Hadoop MapReduce Master. Now, the MapReduce master will divide this job into further equivalent job-parts. These job-parts are then made available for the Map and Reduce Task. This Map and Reduce task will contain the program as per the requirement of the use-case that the particular company is solving. The developer writes their logic to fulfill the requirement that the industry requires. The input data which we are using is then fed to the Map Task and the Map will generate intermediate key-value pair as its output. The output of Map i.e. these key-value pairs are then fed to the Reducer and the final output is stored on the HDFS. There can be n number of Map and Reduce tasks made available for processing the data as per the requirement. The algorithm for Map and Reduce is made with a very optimized way such that the time complexity or space complexity is minimum.

Let's discuss the MapReduce phases to get a better understanding of its architecture:

The MapReduce task is mainly divided into **2 phases** i.e. Map phase and Reduce phase.

1. **Map:** As the name suggests its main use is to map the input data in key-value pairs. The input to the map may be a key-value pair where the key can be the id of some kind of address and value is the actual value that it keeps. The *Map()* function will be executed in its memory repository on each of these input key-value pairs and generates the intermediate key-value pair which works as input for the Reducer or *Reduce()* function.

2. **Reduce:** The intermediate key-value pairs that work as input for Reducer are shuffled and sort and send to the *Reduce()* function. Reducer aggregate or group the data based on its key-value pair as per the reducer algorithm written by the developer.

How Job tracker and the task tracker deal with MapReduce:

1. **Job Tracker:** The work of Job tracker is to manage all the resources and all the jobs across the cluster and also to schedule each map on the Task Tracker running on the same data node since there can be hundreds of data nodes available in the cluster.

2. **Task Tracker:** The Task Tracker can be considered as the actual slaves that are working on the instruction given by the Job Tracker. This Task Tracker is deployed on each of the nodes available in the cluster that executes the Map and Reduce task as instructed by Job Tracker.

There is also one important component of MapReduce Architecture known as **Job History Server**. The Job History Server is a daemon process that saves and stores historical information about the task or application, like the logs which are generated during or after the job execution are stored on Job History Server.

3a

Cassandra collections are used to handle tasks. You can store multiple elements in collection. There are three types of collection supported by Cassandra:

- Set
- List
- Map

Set Collection

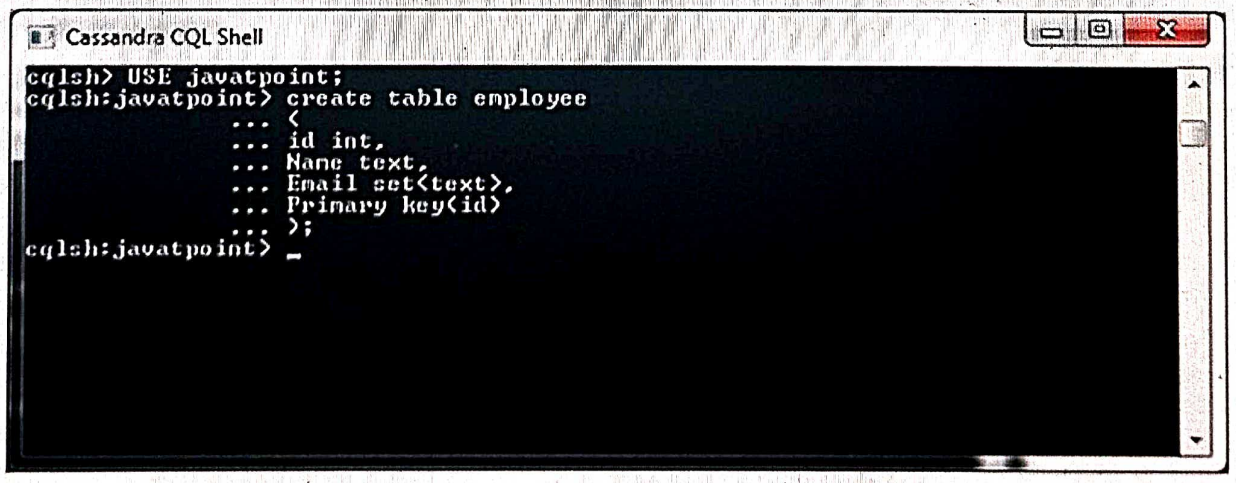
A set collection stores group of elements that returns sorted elements when querying.

Syntax:

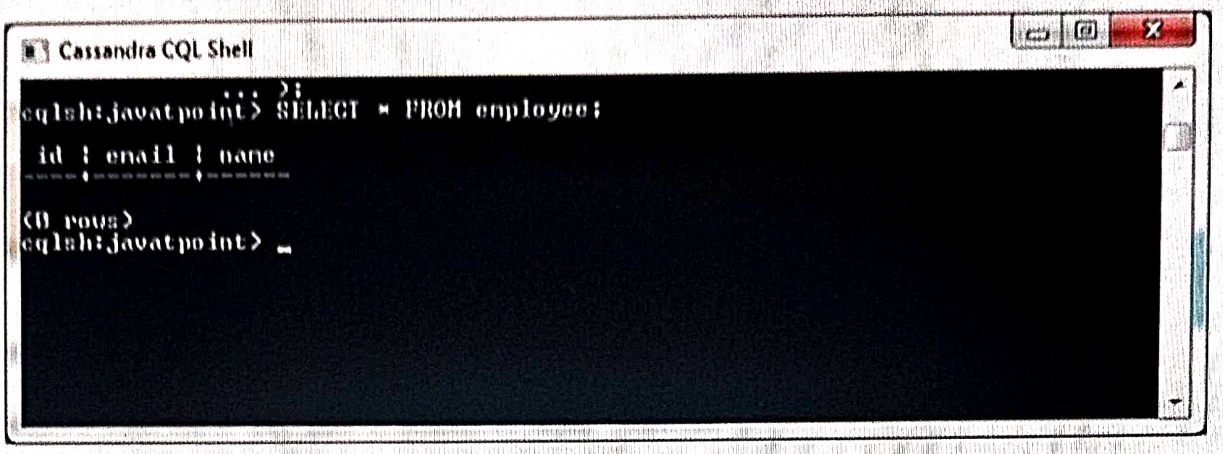
```
1. Create table table_name
2. (
3. id int,
4. Name text,
5. Email set<text>,
6. Primary key(id)
7. );
```

Example:

Let's take an example to demonstrate set collection. Create a table "employee" having the three columns id, name and email.



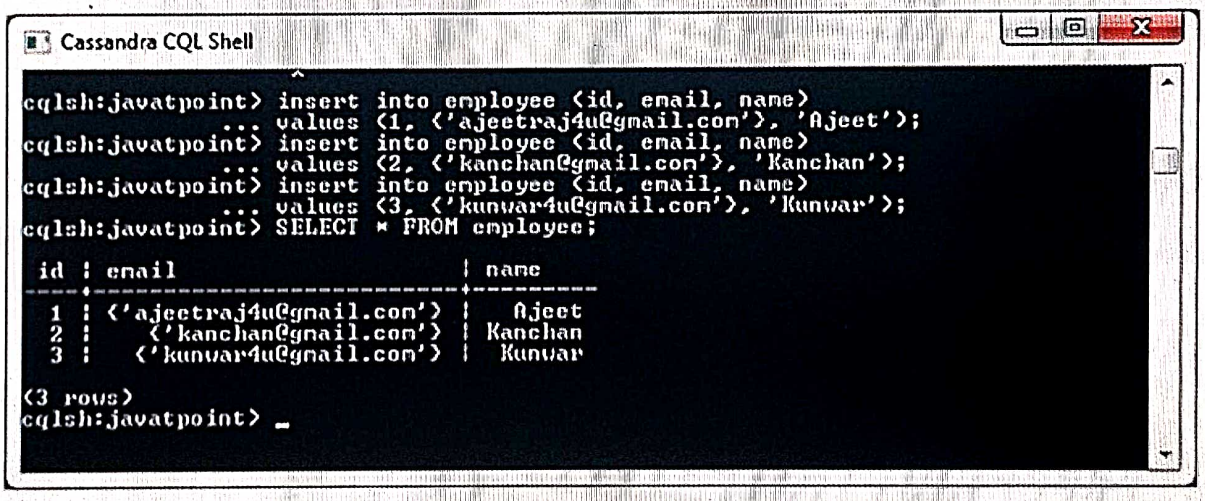
The table is created like this:



Insert values in the table:

1. **INSERT INTO** employee (id, email, **name**)
2. **VALUES**(1, {'ajeetraj4u@gmail.com'}, 'Ajeet');
3. **INSERT INTO** employee (id, email, **name**)
4. **VALUES**(2,{'kanchan@gmail.com'}, 'Kanchan');
5. **INSERT INTO** employee (id, email, **name**)
6. **VALUES**(3, {'kunwar4u@gmail.com'}, 'Kunwar');

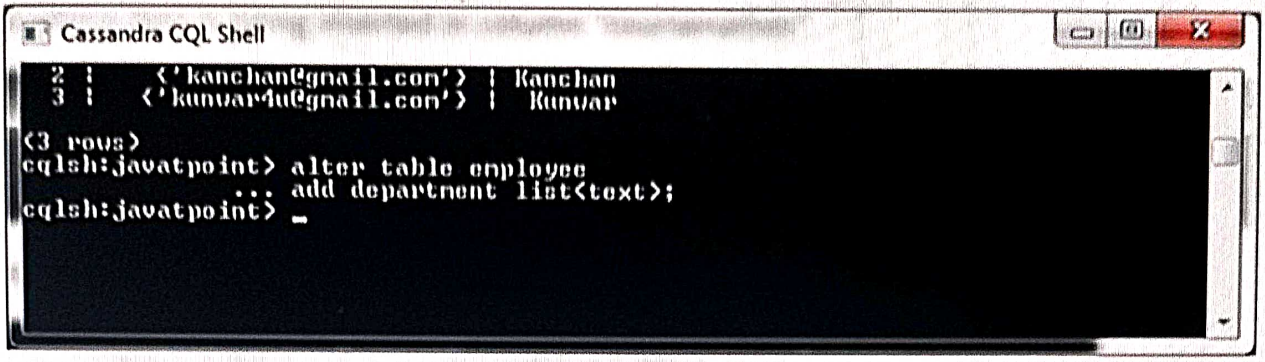
Output:



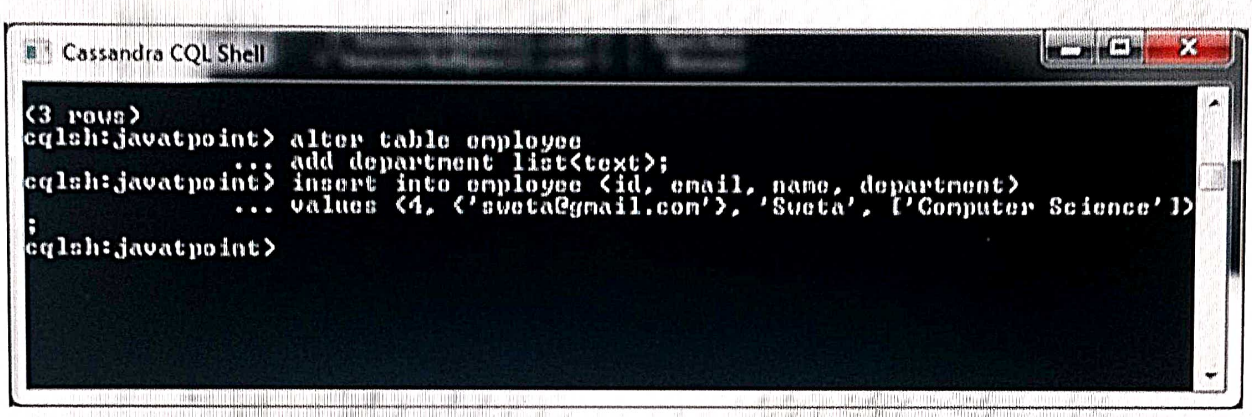
List Collection

The list collection is used when the order of elements matters.

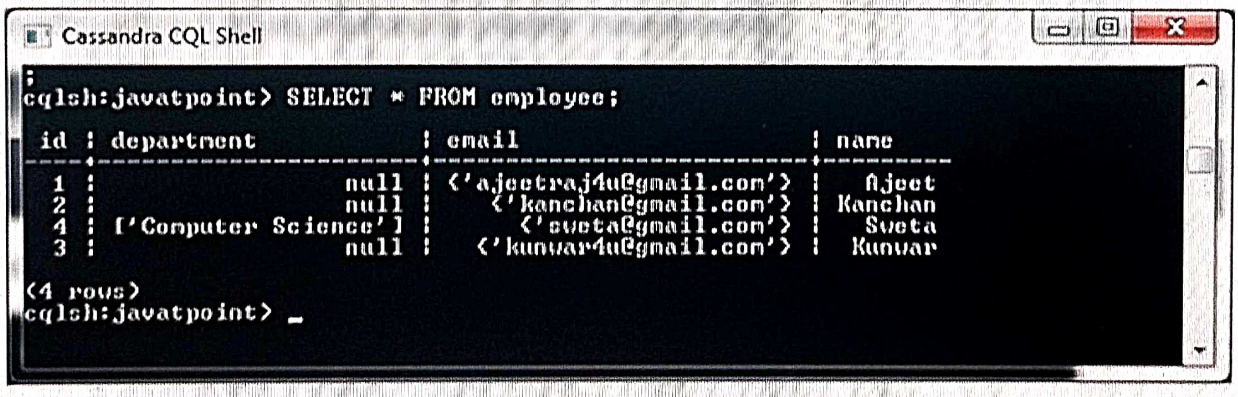
Let's take the above example of "employee" table and a new column name "department" in the table employee.



Now the new column is added. Insert some value in the new column "department".



Output:

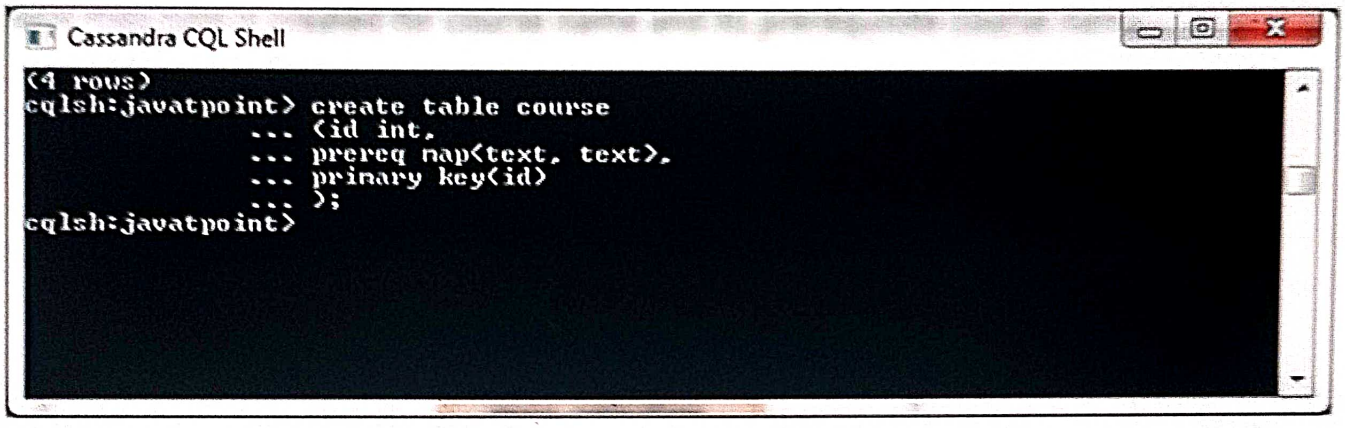


Map Collection

The map collection is used to store key value pairs. It maps one thing to another. For example, if you want to save course name with its prerequisite course name, you can use map collection.

See this example:

Create a table named "course".



Now table is created. Insert some data in map collection type.

Output:

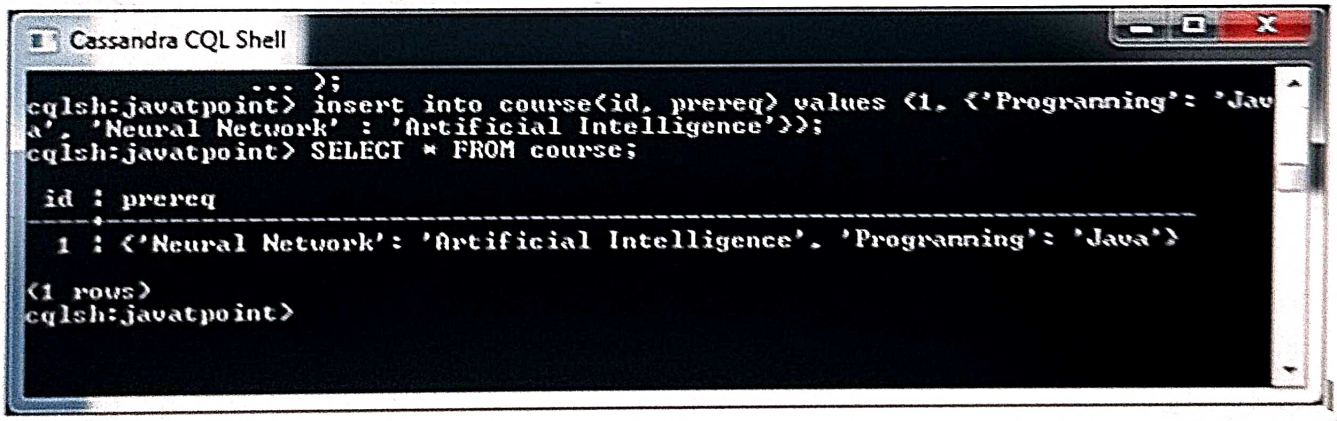


Table Operations:

1. **Insert**:

   Insert operation is used to add new rows or update existing rows in a Cassandra table.

   Example:

```cql
INSERT INTO user_profiles (user_id, profile)
VALUES (uuid(), {'name': 'Alice', 'email': 'alice@example.com', 'phone': '555-123-45
```

2. **Select**:

   Select operation is used to retrieve data from a Cassandra table.

   Example:

```cql
SELECT * FROM user_profiles;
```

3. **Update**:

   Update operation is used to modify existing data in a Cassandra table.

   Example:

```cql
UPDATE user_profiles SET profile['phone'] = '555-987-6543' WHERE user_id = some_uuid
```

4. **Delete**:

   Delete operation is used to remove data from a Cassandra table.

   Example:

```cql
DELETE FROM user_interests WHERE user_id = some_uuid;
```

5. **Truncate**:

   Truncate operation is used to remove all data from a Cassandra table, effectively clearing it.

   Example:

```cql
TRUNCATE user_profiles;
```

3b

In Cassandra, the data model revolves around keyspaces, tables, column families (deprecated in newer versions), and columns. Here's an explanation of each with suitable examples:

1. Keyspaces:

Keyspaces in Cassandra are similar to databases in relational databases. They serve as a container for tables and replication settings. Each keyspace can contain multiple tables.

Example:

```cql
CREATE KEYSPACE my_keyspace
WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 3};
```

2. Tables:

Tables in Cassandra store data in rows and columns. Each table belongs to a keyspace and has a primary key that uniquely identifies rows.

Example:

```cql
CREATE TABLE my_keyspace.users (
    user_id UUID PRIMARY KEY,
    name TEXT,
    email TEXT
);
```

3. Column Families (Deprecated):

Column families were used in earlier versions of Cassandra and are conceptually similar to tables. However, in newer versions, tables are used exclusively.

Example (deprecated):

```cql
CREATE COLUMNFAMILY my_keyspace.users_cf (
    user_id UUID PRIMARY KEY,
    name TEXT,
    email TEXT
);
```

4. Columns:

Columns in Cassandra represent individual fields within a row. Each row can have a different set of columns, and columns are not required to be defined upfront.

Example:
Assuming we have the table `users` in the keyspace `my_keyspace`, we can insert data into it:

```cql
INSERT INTO my_keyspace.users (user_id, name, email)
VALUES (uuid(), 'John Doe', 'john@example.com');
```

In this example, `user_id`, `name`, and `email` are columns within the `users` table.

Key Differences:

- Keyspaces serve as a container for tables, while tables hold rows and columns of data.
- Column families are deprecated in favor of tables.
- Columns in Cassandra are not rigidly defined and can vary between rows within a table.

Features of MongoDB – 4a

- **Schema-less Database:** It is the great feature provided by the MongoDB. A Schema-less database means one collection can hold different types of documents in it. Or in other words, in the MongoDB database, a single collection can hold multiple documents and these documents may consist of the different numbers of fields, content, and size. It is not necessary that the one document is similar to another document like in the relational databases. Due to this cool feature, MongoDB provides great flexibility to databases.
- **Document Oriented:** In MongoDB, all the data stored in the documents instead of tables like in RDBMS. In these documents, the data is stored in fields(key-value pair) instead of rows and columns which make the data much more flexible in comparison to RDBMS. And each document contains its unique object id.
- **Indexing:** In MongoDB database, every field in the documents is indexed with primary and secondary indices this makes easier and takes less time to get or search data from the pool of the data. If the data is not indexed, then database search each document with the specified query which takes lots of time and not so efficient.
- **Scalability:** MongoDB provides horizontal scalability with the help of sharding. Sharding means to distribute data on multiple servers, here a large amount of data is partitioned into data chunks using the shard key, and these data chunks are evenly distributed across shards that reside across many physical servers. It will also add new machines to a running database.
- **Replication:** MongoDB provides high availability and redundancy with the help of replication, it creates multiple copies of the data and sends these copies to a different server so that if one server fails, then the data is retrieved from another server.
- **Aggregation:** It allows to perform operations on the grouped data and get a single result or computed result. It is similar to the SQL GROUPBY clause. It provides three different aggregations i.e, aggregation pipeline, map-reduce function, and single-purpose aggregation methods
- **High Performance:** The performance of MongoDB is very high and data persistence as compared to another database due to its features like scalability, indexing, replication, etc.

4b

❖ **MongoDB datatypes**

- ➢ **String** − This is the most commonly used datatype to store the data. The string in MongoDB must be UTF-8 valid.
- ➢ **Integer** − This type is used to store a numerical value. Integer can be 32-bit or 64-bit, depending upon your server.
- ➢ **Boolean** − This type is used to store a boolean (true/ false) value.
- ➢ **Double** − This type is used to store floating point values.
- ➢ **Min/ Max keys** − This type is used to compare a value against the lowest and highest BSON elements.
- ➢ **Arrays** − This type is used to store arrays or lists or multiple values into one key.
- ➢ **Timestamp** − timestamp. This can be handy for recording when a document has been modified or added.
- ➢ **Object** − This datatype is used for embedded documents.
- ➢ **Null** − This type is used to store a Null value.
- ➢ **Symbol** − This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.
- ➢ **Date** − This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating an object of Date and passing a day, month, or year into it.
- ➢ **Object ID** − This datatype is used to store the document's ID.
- ➢ **Binary data** − This datatype is used to store binary data.
- ➢ **Code** − This datatype is used to store JavaScript code in the document.
- ➢ **Regular expression** − This datatype is used to store regular expressions.

5a **What is Apache Pig**

Apache Pig is a high-level data flow platform for executing MapReduce programs of Hadoop. The language used for Pig is Pig Latin.

The Pig scripts get internally converted to Map Reduce jobs and get executed on data stored in HDFS. Apart from that, Pig can also execute its job in Apache Tez or Apache Spark.

Pig can handle any type of data, i.e., structured, semi-structured or unstructured and stores the corresponding results into Hadoop Data File System. Every task which can be achieved using PIG can also be achieved using java used in MapReduce.

Features of Apache Pig

Let's see the various uses of Pig technology.

1) Ease of programming

Writing complex java programs for map reduce is quite tough for non-programmers. Pig makes this process easy. In the Pig, the queries are converted to MapReduce internally.

2) Optimization opportunities

It is how tasks are encoded permits the system to optimize their execution automatically, allowing the user to focus on semantics rather than efficiency.

3) Extensibility

A user-defined function is written in which the user can write their logic to execute over the data set.

4) Flexible

It can easily handle structured as well as unstructured data.

5) In-built operators

It contains various type of operators such as sort, filter and joins.

Differences between Apache MapReduce and PIG

Advantages of Apache Pig

- Less code - The Pig consumes less line of code to perform any operation.
- Reusability - The Pig code is flexible enough to reuse again.

- Nested data types - The Pig provides a useful concept of nested data types like tuple, bag, and map.

Pig Data Types

Apache Pig supports many data types. A list of Apache Pig Data Types with description and examples are given below.

Type	Description	Example
Int	Signed 32 bit integer	2
Long	Signed 64 bit integer	15L or 15l
Float	32 bit floating point	2.5f or 2.5F
Double	32 bit floating point	1.5 or 1.5e2 or 1.5E2
charArray	Character array	hello javatpoint
byteArray	BLOB(Byte array)	
tuple	Ordered set of fields	(12,43)
bag	Collection f tuples	{(12,43),(54,28)}
map	collection of tuples	[open#apache]

5b

INPUT DATA:

Wordcount.txt

Hello world

This is an example of input data for word count using Hadoop

Hadoop is a distributed computing framework

It is widely used for big data processing

COMMANDS TO RUN:

-- Load the input file from HDFS

input_data = LOAD 'hdfs://<hdfs_host>:<hdfs_port>/<input_file>' USING PigStorage(',') AS (line:chararray);

-- Tokenize each line into words

words = FOREACH input_data GENERATE FLATTEN(TOKENIZE(line)) AS word;

-- Group the words

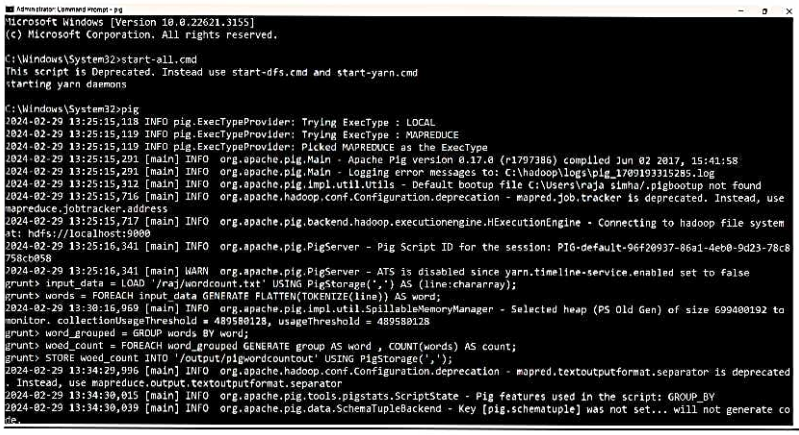
word_grouped = GROUP words BY word;

-- Count the occurrences of each word

word_count = FOREACH word_grouped GENERATE group AS word, COUNT(words) AS count;

-- Store the result in HDFS

STORE word_count INTO 'hdfs://<hdfs_host>:<hdfs_port>/<output_directory>' USING PigStorage(',');



OUTPUT DATA:

a,1

It,1

an,1

is,3

of,1

big,1

for,2

This,1

data,2

used,1

word,1

Hello,1

count,1

input,1

using,1

world,1

Hadoop,2

widely,1

example,1

6a HIVE Data Types

Hive data types are categorized in numeric types, string types, misc types, and complex types. A list of Hive data types is given below.

Integer Types

Type	Size	Range
TINYINT	1-byte signed integer	-128 to 127
SMALLINT	2-byte signed integer	32,768 to 32,767
INT	4-byte signed integer	2,147,483,648 to 2,147,483,647
BIGINT	8-byte signed integer	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Decimal Type

Type	Size	Range
FLOAT	4-byte	Single precision floating point number
DOUBLE	8-byte	Double precision floating point number

Date/Time Types

TIMESTAMP

- It supports traditional UNIX timestamp with optional nanosecond precision.
- As Integer numeric type, it is interpreted as UNIX timestamp in seconds.
- As Floating point numeric type, it is interpreted as UNIX timestamp in seconds with decimal precision.
- As string, it follows java.sql.Timestamp format "YYYY-MM-DD HH:MM:SS.fffffffff" (9 decimal place precision)

DATES

The Date value is used to specify a particular year, month and day, in the form YYYY--MM--DD. However, it didn't provide the time of the day. The range of Date type lies between 0000--01--01 to 9999--12--31.

String Types

STRING

The string is a sequence of characters. It values can be enclosed within single quotes (') or double quotes (").

Varchar

The varchar is a variable length type whose range lies between 1 and 65535, which specifies that the maximum number of characters allowed in the character string.

CHAR

The char is a fixed-length type whose maximum length is fixed at 255.

Complex Type

Type	Size	Range
Struct	It is similar to C struct or an object where fields are accessed using the "dot" notation.	struct('James','Roy')
Map	It contains the key-value tuples where the fields are accessed using array notation.	map('first','James','last','Roy')
Array	It is a collection of similar type of values that indexable using zero-based integers.	array('James','Roy')

6b Hive DDL Commands

create database

drop database

create table

drop table

alter table

create index

create view

Hive DML Commands

Select

Where

Group By

Order By

Load Data

Join:

- Inner Join
- Left Outer Join
- Right Outer Join
- Full Outer Join

Hive DDL Commands

Create Database Statement

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

1. hive> CREATE SCHEMA userdb;
2. hive> SHOW DATABASES;

Drop database

1. ive> DROP DATABASE IF EXISTS userdb;

Creating Hive Tables

Create a table called Sonoo with two columns, the first being an integer and the other a string.

1. hive> CREATE TABLE Sonoo(foo INT, bar STRING);

Create a table called HIVE_TABLE with two columns and a partition column called ds. The partition column is a virtual column. It is not part of the data itself but is derived from the partition that a particular dataset is loaded into.By default, tables are assumed to be of text input format and the delimiters are assumed to be ^A(ctrl-a).

1. hive> CREATE TABLE HIVE_TABLE (foo INT, bar STRING) PARTITIONED BY (ds STRI NG);

Browse the table

1. hive> Show tables;

Altering and Dropping Tables

1. hive> ALTER TABLE Sonoo RENAME TO Kafka;
2. hive> ALTER TABLE Kafka ADD COLUMNS (col INT);
3. hive> ALTER TABLE HIVE_TABLE ADD COLUMNS (col1 INT COMMENT 'a comment');
4. hive> ALTER TABLE HIVE_TABLE REPLACE COLUMNS (col2 INT, weight STRING, baz INT COMMENT 'baz replaces new_col1');

Hive DML Commands

To understand the Hive DML commands, let's see the employee and employee_department table first.

Employee			Employee Department	
EMP ID	Emp Name	Address	Emp ID	Department
1	Rose	US	1	IT
2	Fred	US	2	IT
3	Jess	In	3	Eng
4	Frey	Th	4	Admin

LOAD DATA

1. hive> LOAD DATA LOCAL INPATH './usr/Desktop/kv1.txt' OVERWRITE INTO TABLE Employe e;

SELECTS and FILTERS

1. hive> SELECT E.EMP_ID FROM Employee E WHERE E.Address='US';

GROUP BY

1. hive> hive> SELECT E.EMP_ID FROM Employee E GROUP BY E.Addresss;

Adding a Partition

We can add partitions to a table by altering the table. Let us assume we have a table called **employee** with fields such as Id, Name, Salary, Designation, Dept, and yoj.

Syntax:

```
ALTER TABLE table_name ADD [IF NOT EXISTS] PARTITION partition_spec
[LOCATION 'location1'] partition_spec [LOCATION 'location2'] ...;

partition_spec:
: (p_column = p_col_value, p_column = p_col_value, ...)
```

The following query is used to add a partition to the employee table.

```
hive> ALTER TABLE employee
> ADD PARTITION (year='2012')
> location '/2012/part2012';
```

Renaming a Partition

The syntax of this command is as follows.


```
ALTER TABLE table_name PARTITION partition_spec RENAME TO PARTITION
partition_spec;
```

The following query is used to rename a partition:

```
hive> ALTER TABLE employee PARTITION (year='1203')
  > RENAME TO PARTITION (Yoj='1203');
```

Dropping a Partition

The following syntax is used to drop a partition:

```
ALTER TABLE table_name DROP [IF EXISTS] PARTITION partition_spec, PARTITION
partition_spec,...;
```

The following query is used to drop a partition:

```
hive> ALTER TABLE employee DROP [IF EXISTS]
  > PARTITION (year='1203');
```