****

FACULTY OF ENGINEERING AND TECHNOLOGY

BACHELOR OF TECHNOLOGY

**BIG DATA ANALYSIS**

**(203105348)**

7th SEMESTER

7A13

**NAME**: **VISHAL SURESH UMAVANE**

**ROLL NO.: 32**

**ENROLLMENT NUMBER**: **210306105192**

**CERTIFICATE**

*This is to certify that**Mr.* ***Vishal Suresh Umavane*** *with Enrolment no* ***210306105192*** *has successfully completed his laboratory experiments in the Big data analytics (203105348) from the department of* ***B.Tech CSE*** *during the academic year* ***2024-25***

**

Date of Submission:.........................

Staff In charge:.............................

Head Of Department:...........................................

# INDEX

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Sr. No | Experiment Title | Page No | | Date of Start | Date of Completion | Sign | Remarks |
| From | To |
| 1. | To understand the overall programming architecture using Map Reduce API. |  |  |  |  |  |  |
| 2. | Write a program of Word Count in Map Reduce over HDFS. |  |  |  |  |  |  |
| 3. | Basic CRUD operations in MongoDB. |  |  |  |  |  |  |
| 4. | Store the basic information about students such as roll no, name, date of birth, and address of student using various collection types such as List, Set and Map. |  |  |  |  |  |  |
| 5. | Basic commands available for the Hadoop Distributed File System. |  |  |  |  |  |  |
| 6. | Basic commands available for HIVE Query Language. |  |  |  |  |  |  |
| 7. | Basic commands of HBASE Shell. |  |  |  |  |  |  |
| 8. | Creating the HDFS tables and loading them in Hive and learn joining of tables in Hive. |  |  |  |  |  |  |

**PRACTICAL 1**

**Aim:** To Understand the overall programing architecture using Map Reduce API.

**Solution:**

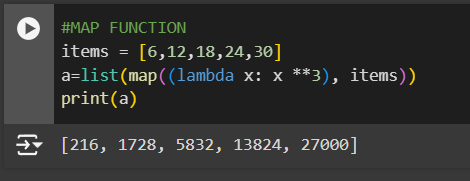
* **MapReduce** and **HDFS** are the two major components of Hadoop which makes it so powerful and efficient to use.
* Reduce is a programming model used for efficient processing in parallel over large data-sets in a distributed manner.
* The data is first split and then combined to produce the final result.
* The libraries for MapReduce is written in so many programming languages with various different-different optimizations.
* The purpose of MapReduce in Hadoop is to Map each of the jobs and then it will reduce it to equivalent tasks for providing ess overhead over the cluster network and to reduce the processing power.
* MapReduce task is mainly divided into two phases Map Phase and Reduce Phase.
* map ( ), filter(), and reduce() in Python.
* These functions are most commonly used with Lambda function.

### Map() Function:

A map function executes certain instructions or functionality provided to it on every item of an iterable. "The iterable could be a list, tuple, set, etc.

Syntax:

map(function, iterable)



The Map() function passes each element in the list to a lambda function and return the mapped object.

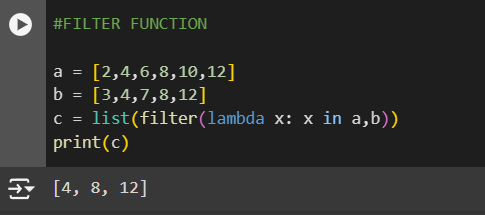
## Filter() Function:

A Filter function in Python tests a specific user defined condition for a function and returns an iterable for the elements and values that satisfy the condition or, in other words, return True.

Syntax:

Filter(function, iterable)

Example:



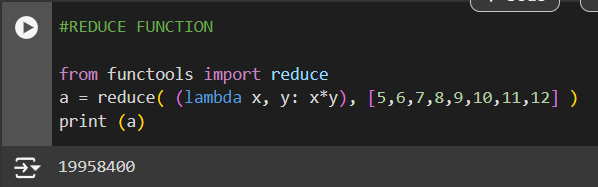
## Reduce() Function:

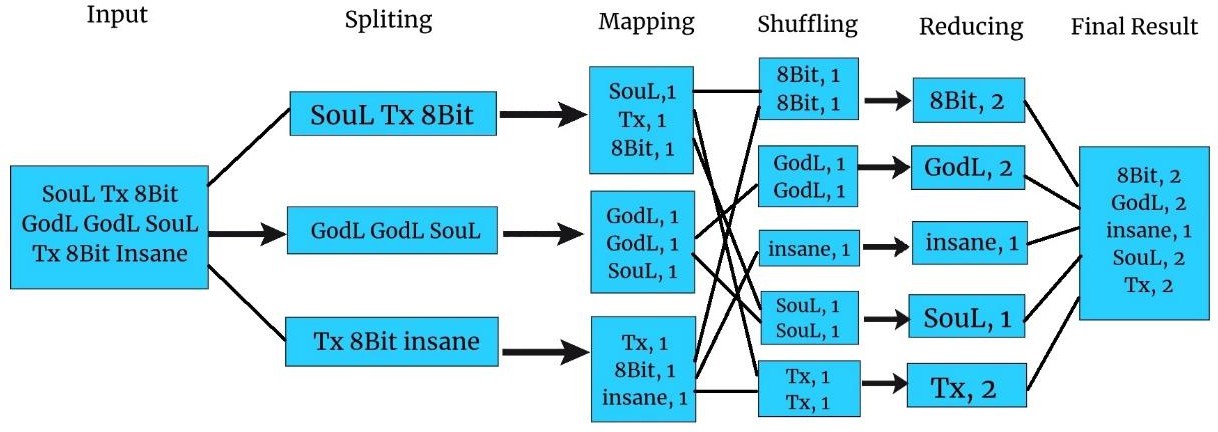
Reduce functions apply a function to every item of an iterable and gives back a single value as a resultant.

We have to import to reduce function from Functools module using the statement. Syntax:

Reduce(function, iterable)

Example:





**PRACTICAL 2**

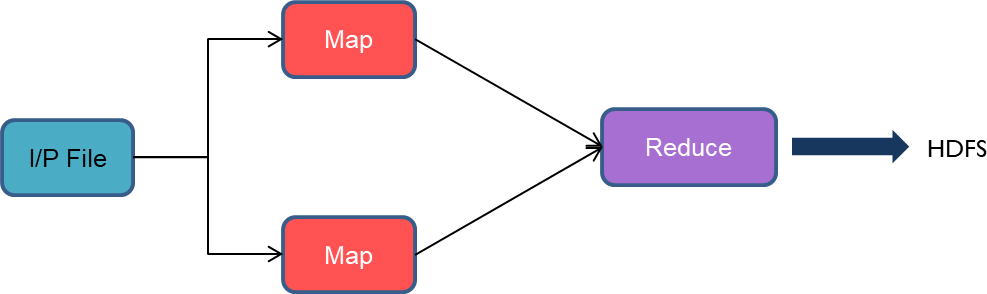
**Aim:** Write a program of Word Count in Map Reduce over HDFS.

### Description:

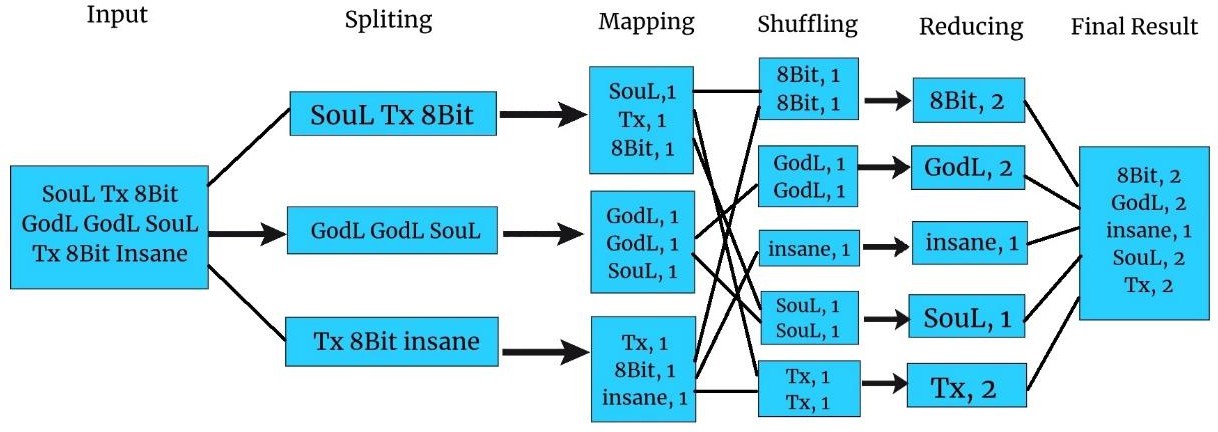
MapReduce is a framework for processing large datasets using a large number of computers (nodes), collectively referred to as a cluster. Processing can occur on data stored in a file system (HDFS).A method for distributing computation across multiple nodes.Each node processes the data that is stored at that node.

Consists of two main phases Mapper Phase

Reduce phase



Input data set is split into independent blocks – processed in parallel. Each input split is converted in Key Value pairs. Mapper logic processes each key value pair and produces and intermediate key value pairs based on the implementation logic. Resultant key value pairs can be of different type from that of input key value pairs. The output of Mapper is passed to the reducer. Output of Mapper function is the input for Reducer. Reducer sorts the intermediate key value pairs. Applies reducer logic upon the key value pairs and produces the output in desired format.Output is stored in HDFS



## Code:

import urllib.request import random

from operator import itemgetter

current\_word = {} current\_count = 0

story = ['http://sixty](http://sixty-north.com/c/t.txt%27)-[north.com/c/t.txt'](http://sixty-north.com/c/t.txt%27) request = urllib.request.Request(story)

def map\_function(line): words = line.split()

return [(word.lower(), 1) for word in words]

def reduce\_function(mapped\_values): word\_counts = {}

for word, count in mapped\_values: if word in word\_counts:

word\_counts[word] += count else:

word\_counts[word] = count return word\_counts

def map\_reduce():

global current\_word, current\_count

with urllib.request.urlopen(request) as response: text = response.read().decode('utf-8')

mapped = []

for line in text.split('\n'): mapped.extend(map\_function(line))

random.shuffle(mapped)

reduced = reduce\_function(mapped)

sorted\_counts = sorted(reduced.items(), key=itemgetter(1), reverse=True) if sorted\_counts:

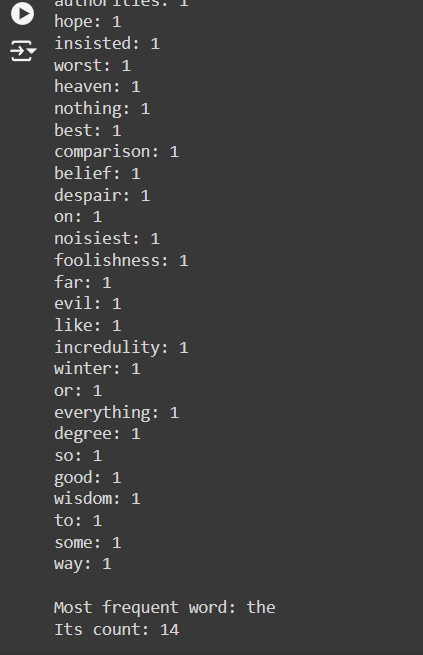
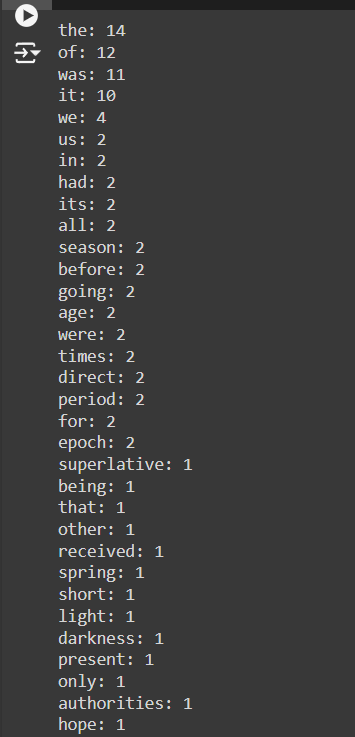
current\_word, current\_count = sorted\_counts[0] return sorted\_counts

result = map\_reduce()

for word, count in result: print(f"{word}: {count}")

print(f"\nMost frequent word: {current\_word}") print(f"Its count: {current\_count}")

**Output:**



# PRACTICAL 3

**Aim:** To study Basic CRUD operations in MongoDB.

### Description: CRUD operation does:

* **Create**
  + Adds new records or data to the database.
  + Example: Registering a new user account.
  + SQL Command: INSERT INTO table\_name (columns) VALUES (values);
  + HTTP Method: POST
* **Read**
  + Retrieves existing records or data from the database.
  + Example: Fetching user details or listing all users.
  + SQL Command: SELECT columns FROM table\_name WHERE condition;
  + HTTP Method: GET
* **Update**
  + Modifies existing records or data in the database.
  + Example: Updating user profile information.
  + SQL Command: UPDATE table\_name SET column1 = value1, column2 = value2 WHERE condition;
  + HTTP Method: PUT or PATCH
* **Delete**
  + Removes records or data from the database.
  + Example: Deleting a user account.
  + SQL Command: DELETE FROM table\_name WHERE condition;
  + HTTP Method: DELETE

**Program:** db.createCollection("employees"); db.employees.insertMany([

]);

{empId: 1, name: 'Clark', dept: 'Sales' },

{empId: 2, name: 'Dave', dept: 'Accounting' },

{empId: 3, name: 'Ava', dept: 'Sales' }

db.employees.find({dept: 'Sales'});

db.employees.insert({empId: 4, name: 'Raja', dept: 'marketing' }) db.employees.update({name:'Raja'},{$set:{name:'Alise'}}); db.employees.find({dept: 'marketing'}); db.employees.remove({dept:'Accounting'});

db.createCollection("orders") db.orders.insertMany( [

{ id: 0, name: "Pepperoni", size: "small", price: 19, quantity: 10, date: ISODate( "2021-03-13T08:14:30Z" ) },

{ id: 1, name: "Pepperoni", size: "medium", price: 20, quantity: 20, date : ISODate( "2021-03-13T09:13:24Z" ) },

{ id: 2, name: "Pepperoni", size: "large", price: 21,

quantity: 30, date : ISODate( "2021-03-17T09:22:12Z" ) },

{ id: 3, name: "Cheese", size: "small", price: 12,

quantity: 15, date : ISODate( "2021-03-13T11:21:39.736Z" ) },

{ id: 4, name: "Cheese", size: "medium", price: 13,

quantity:50, date : ISODate( "2022-01-12T21:23:13.331Z" ) },

{ id: 5, name: "Cheese", size: "large", price: 14,

quantity: 10, date : ISODate( "2022-01-12T05:08:13Z" ) },

{ id: 6, name: "Vegan", size: "small", price: 17,

quantity: 10, date : ISODate( "2021-01-13T05:08:13Z" ) },

{ id: 7, name: "Vegan", size: "medium", price: 18,

quantity: 10, date : ISODate( "2021-01-13T05:10:13Z" ) }

] )

db.orders.find({size: "medium"});

db.orders.insert({id: 9, name: "Vegan", size: "medium", price: 8, quantity: 5, date : ISODate( "2021-01-22T05:10:13Z" )})

db.orders.updateMany({name:'Vegan'},{$set:{name:'Veg'}}) db.orders.find({name: 'Veg'}); db.orders.remove({name:'Pepperoni'})

db.orders.find({ $and: [ {name: 'Veg'}, { size: "small"} ] }) db.orders.find()

**Output:**



**Practical 4**

**Aim:** Store the basic information about students such as roll no, name, date of birth and address of student using various collection types such as List, Set and Map.

**Description :**

**Using List**

* **Definition:** An ordered collection that allows duplicates and maintains the order of insertion.
* **Use Case:** When the order of students matters or you need to access them by their position.
* **Example:** Storing students in the order they enrolled or taking attendance.

**Using Set**

* **Definition:** An unordered collection that does not allow duplicates.
* **Use Case:** Ensuring each student is unique and preventing duplicate entries.
* **Example:** Storing unique student records to avoid duplicate roll numbers.

**Using Map**

* **Definition:** A collection of key-value pairs where each key is unique.
* **Use Case:** Associating student roll numbers (keys) with their details (values) for quick look-up.
* **Example:** Using roll numbers as keys to retrieve student information efficiently.

**Program 1 :**

use studentdb;

db.createCollection("student");

// Insert multiple documents using insertMany with an array

db.student.insertMany([

{

no: 1,

name: "Vishal",

dob: "19-11-2002",

e\_mail: "vumavane@gmail.com",

phone: "7039144205",

address: {

building: "1234",

street: "main road",

zipcode: "421306"

},

Branch: "CSE",

marks: [50, 70, 60]

},

{

no: 2,

name: "Harsh",

dob: "17-07-2003",

e\_mail: "harshsingh@gmail.com",

phone: "8208629356",

address: {

building: "5678",

street: "main road",

zipcode: "431511"

},

Branch: "CSE",

marks: [45, 20, 80]

},

{

no: 3,

name: "Akash",

dob: "30-09-2003",

e\_mail: "akashpanchal@gmail.com",

phone: "9890898563",

address: {

building: "1256",

street: "main road",

zipcode: "431511"

},

Branch: "CSE",

marks: [59, 89, 68]

},

{

no: 4,

name: "Vidya",

dob: "15-04-2003",

e\_mail: "vidya85@gmail.com",

phone: "9698741235",

address: {

building: "3478",

street: "main road",

zipcode: "431511"

},

Branch: "CSE",

marks: [53, 29, 33]

},

{

no: 5,

name: "Sahil",

dob: "13-05-2003",

e\_mail: "sahil168@gmail.com",

phone: "9642317893",

address: {

building: "2525",

street: "ain road",

zipcode: "431511"

},

Branch: "CSE",

marks: [61, 20, 66]

}

]);

db.student.find()

db.student.update({no:1},{$set:{name:"Rishikesh"}})

db.student.find({name:/^R/})

db.student.find({name:/l$/})

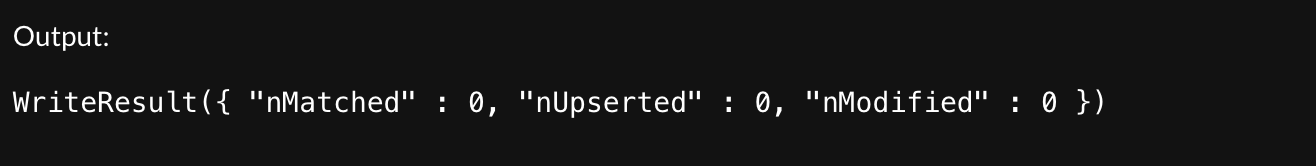
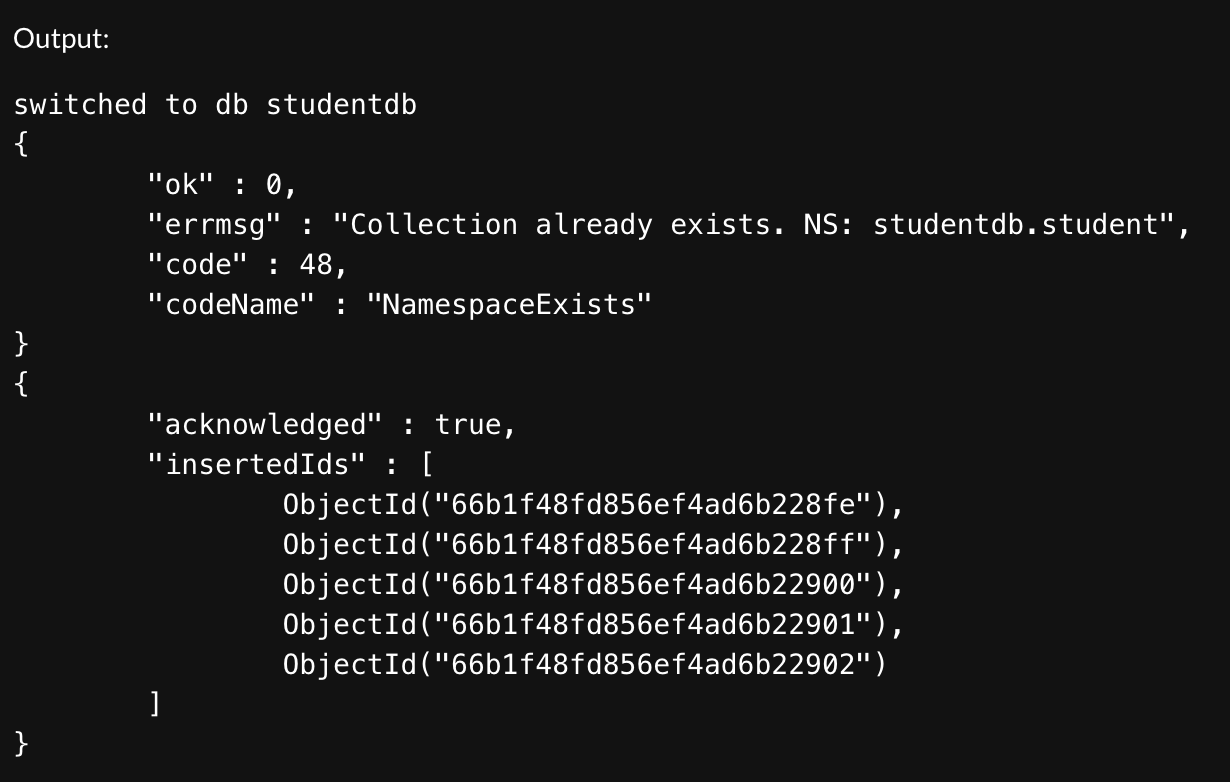
db.student.find({name:/S/})

db.student.count()

db.studnet.find().forEach(function(myDoc) {print("name: Vidya "+myDoc.name);})

db.student.find().limit(2);

db.student.find().sort({"name": -1})

Output :

**Program-2**

db.grades.insertMany([

{\_id: 1, quizzes: [5,6,7]},

{\_id: 2, quizzes: []},

{\_id: 3, quizzes: [3,8,9]}

])

db.grades.aggregate(

[

{

$project:

{

adjustedGrades:

{

$map:

{

input: "$quizzes",

as: "grade",

in: {$add: ["$$grade",2]}

}

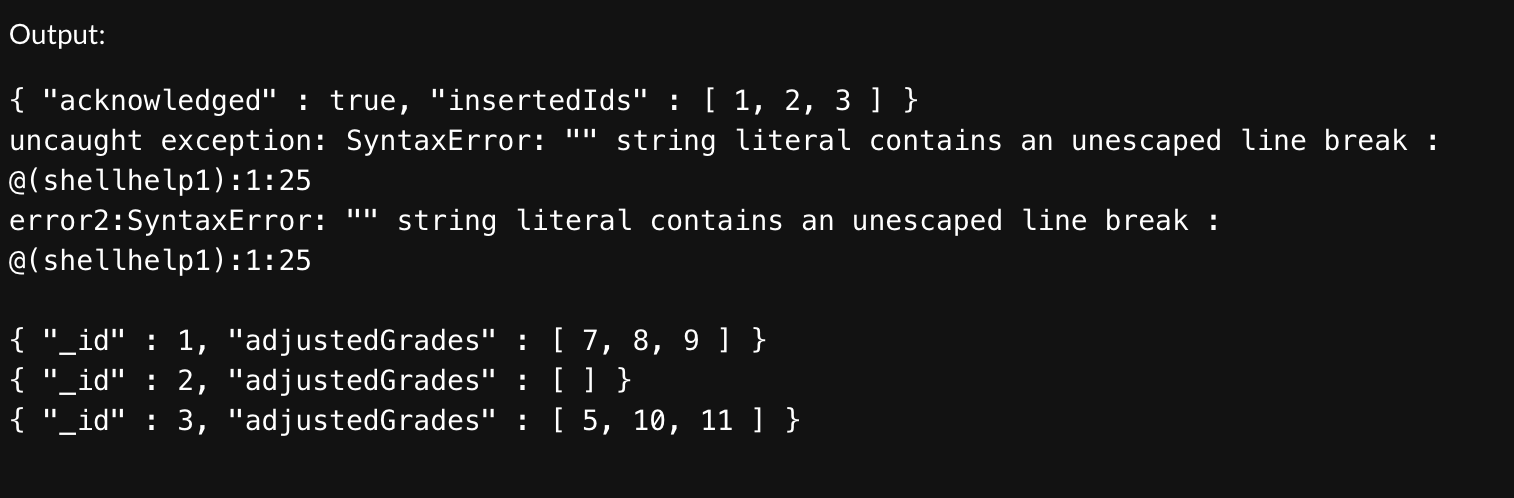
}

}

}

]

)



**Program-3**

db.scores.insertMany([

{\_id: 1, student: "Maya", homework: [10,5,10], quiz: [10,8],extraCredit: 0},

{\_id: 2, student: "Ryan", homework: [5,6,5], quiz: [8,8],extraCredit: 8}

])

db.scores.aggregate([

{

$set:{

totalHomework:{$sum: "$homework"},

totalQuiz:{$sum:"$quiz"}

}

},

{

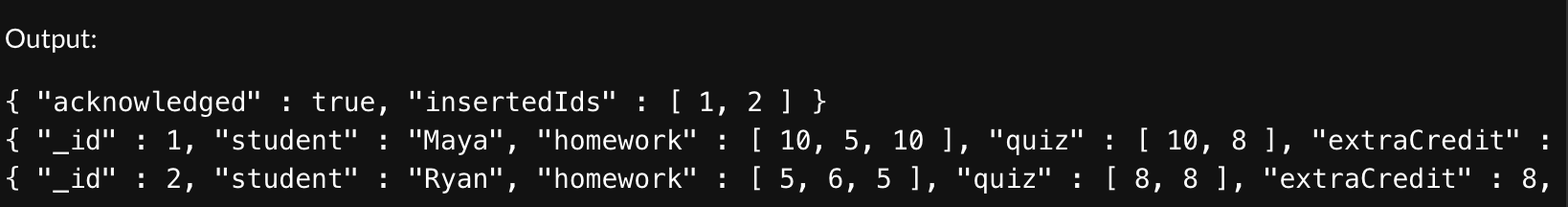
$set:{

totalScore:{$add:["$totalHomework","$totalQuiz","$extraCredit"]}

}

}

])



**Practical-5**

**Aim: To study Basic commands available for the Hadoop Distributed File System**

**HDFS Commands**

HDFS is the primary or major component of the Hadoop ecosystem which is responsible for storing large data sets of structured or unstructured data across various nodes and thereby maintaining the metadata in the form of log files. To use the HDFS commands, first you need to start the Hadoop services using the following command:

**start-all.sh**

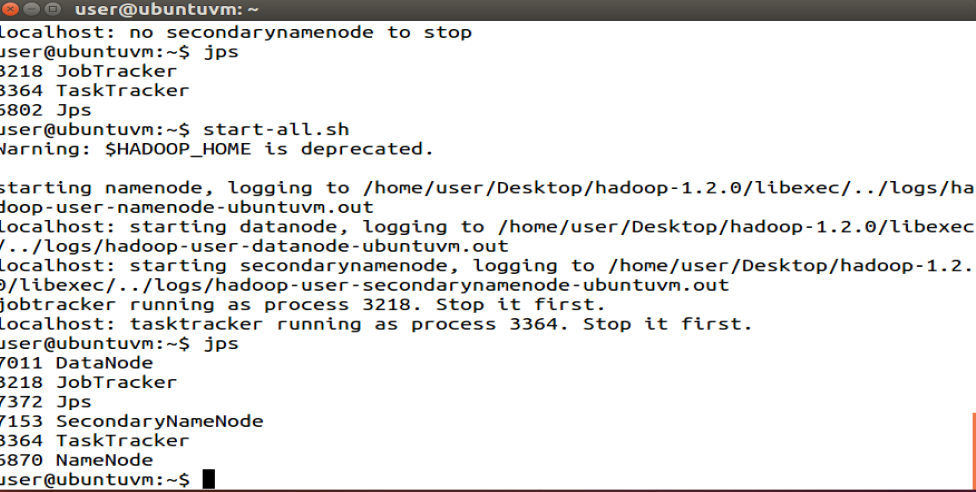
**stop-all.sh**

**hadoop version**

The Hadoop fs shell command version prints the Hadoop version.

**Jps**

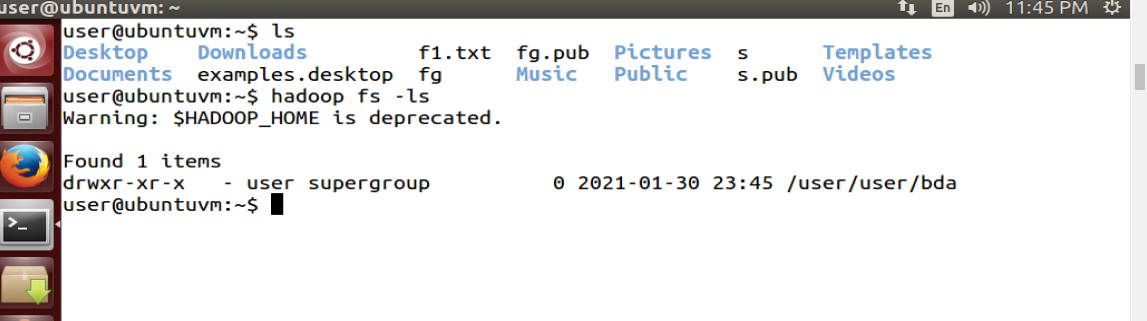
To check the Hadoop services are up and running use the following command:



l**s:** This command is used to list all the files.

**hadoop fs -ls**

It will print all the directories present in HDFS. bin directory contains executables so,



**mkdir:**

To create a directory. In Hadoop dfs there is no home directory by default. So let’s first create it.

hadoop dfs -mkdir bdalab

**vi lab.txt**

**cat lab.txt**

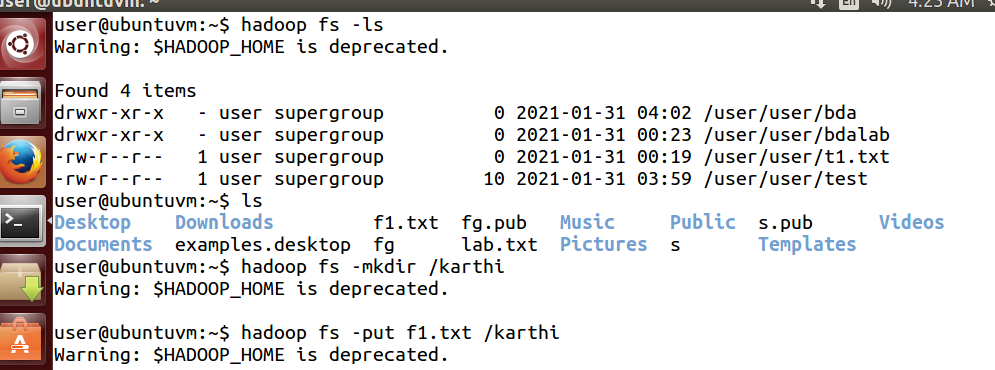
creating local file and viewing the content.

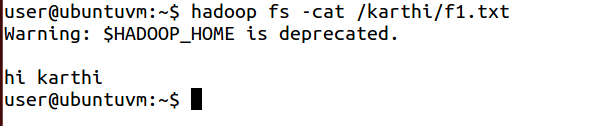
**put**

To copy files/folders from local file system to hdfs store. This is the most important command. Local filesystem means the files present on the OS.

**syntax**

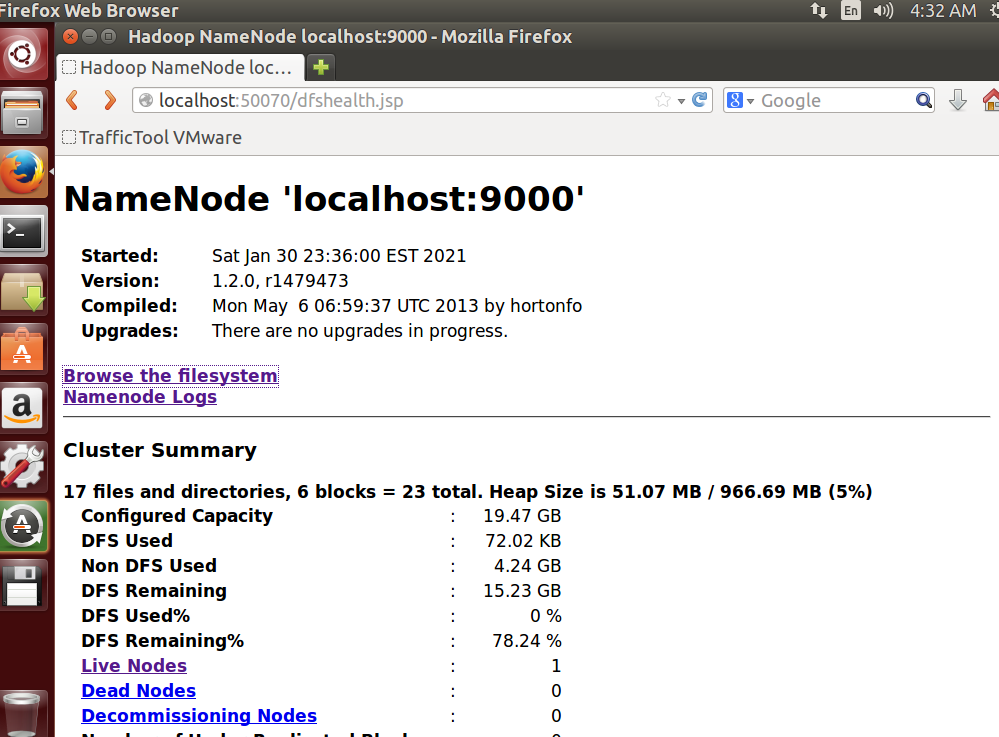
haoop fs -put <localsrc> <dest>

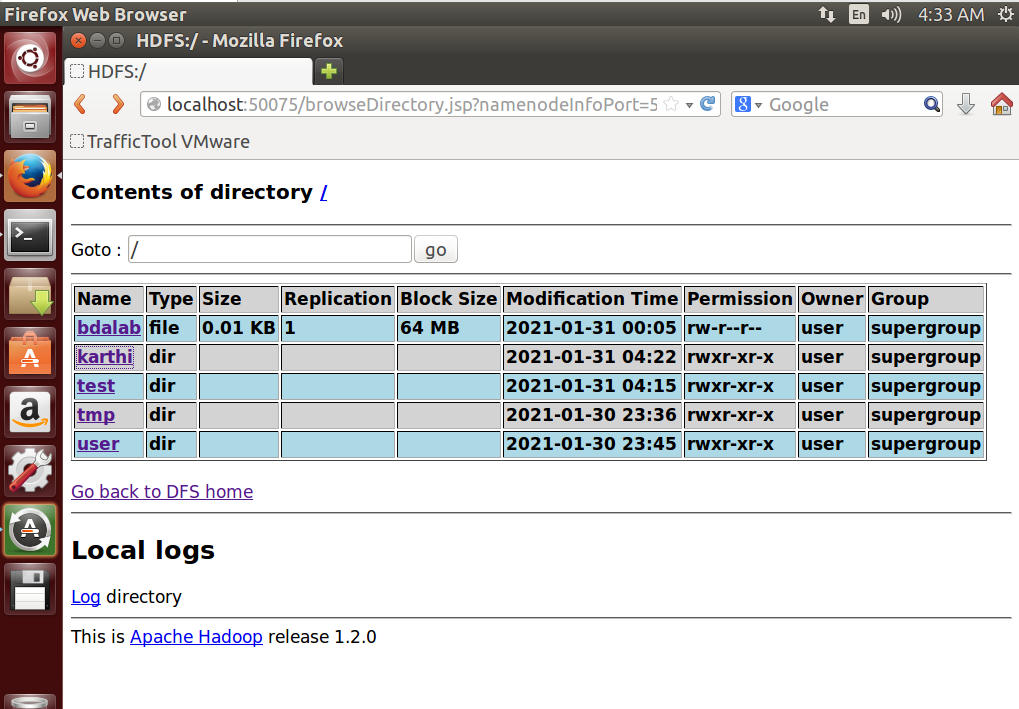




**http://localhost:50070/**

to check the file copied to Hadoop file system or not in the graphical user interface.



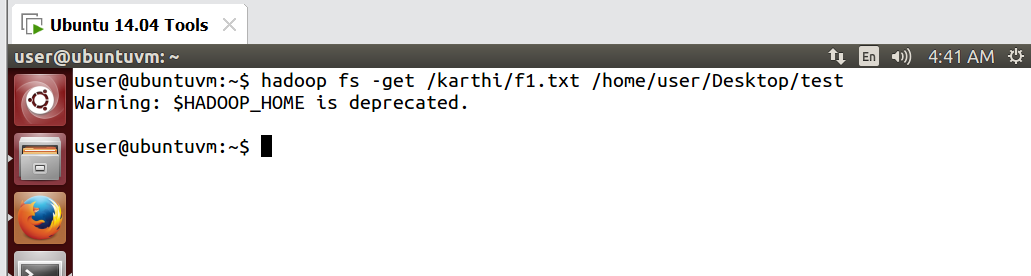


copyToLocal (or) get: To copy files/folders from hdfs store to local file system.

Syntax:

Hadoop fs -get <<srcfile(on hdfs)> <local file dest>

Example:



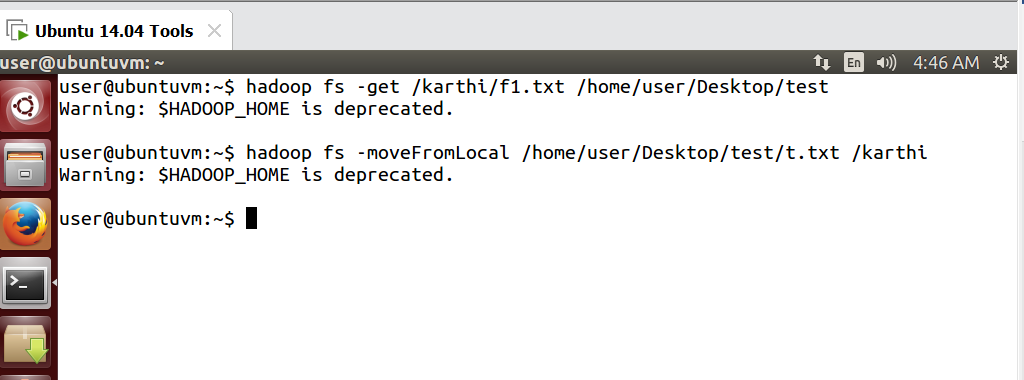
**moveFromLocal:** This command will move file from local to hdfs.

Syntax:

Hadoop fs -moveFromLocal <local src> <dest(on hdfs)>

Example:

hadoop fs -moveFromLocal /home/user/Desktop/test/t.txt /karthi

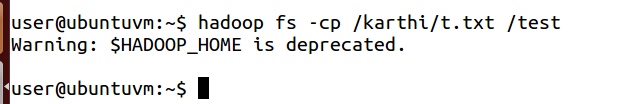


**cp:** This command is used to copy files within hdfs. Lets copy folder geeks to geeks\_copied.

Syntax:

Hadoop -fs -cp <src(on hdfs)> <dest(on hdfs)>

Example:

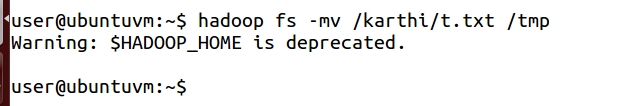


**mv:** This command is used to move files within hdfs.

Syntax:

Hadoop fs -mv <src(on hdfs)> <src(on hdfs)>

Example:

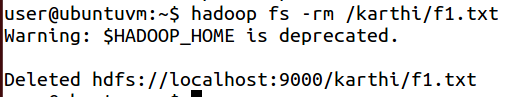


**rm:** This command deletes a file from HDFS.

Syntax:

Hadoop fs -rm <filename/directoryName>

Example:



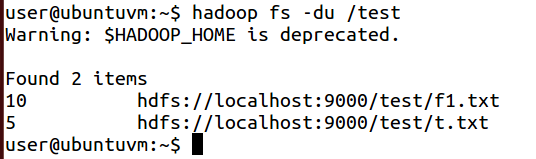
Hadoop fs -rmr /directory -> It will delete all the content inside the directory then the directory itself.

**du:** It will give the size of each file in directory.

Syntax:

Hadoop fs -du <dirName>

Example:

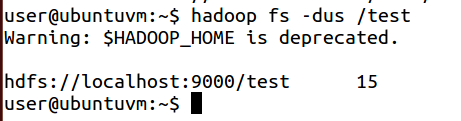


dus:: This command will give the total size of directory/file.

Syntax:

Hadoop fs -dus <dirName>

Example:

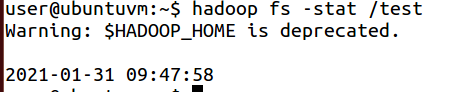


**stat:** It will give the last modified time of directory or path. In short it will give stats of the directory or file.

Syntax:

Hadoop fs -stat <hdfs file>

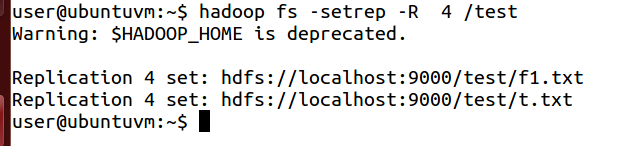
Example:



**setrep**: This command is used to change the replication factor of a file/directory in HDFS. By default, it is 3 for anything which is stored in HDFS (as set in hdfs core-site.xml).

Example 1: To change the replication factor to 6 for geeks.txt stored in HDFS.

Hadoop fs -setrep -R -w 6 test



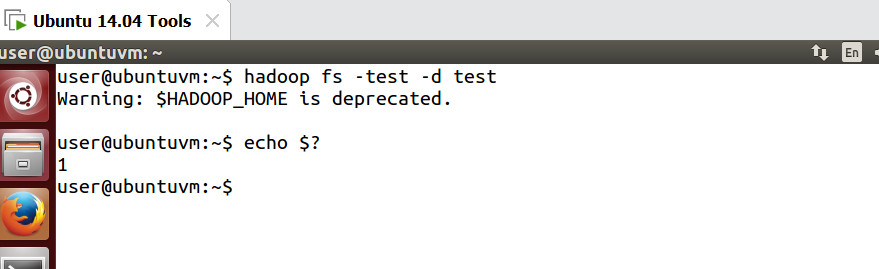
Note: -R means recursively, we use it for directories as they may also contain many files and folders inside them.

**test**

The test command is used for file test operations.

|  |  |
| --- | --- |
| **Options** | **Description** |
| **-d** | Check whether the path given by the user is a directory or not, return 0 if it is a directory. |
| **-e** | Check whether the path given by the user exists or not, return 0 if the path exists. |
| **-f** | Check whether the path given by the user is a file or not, return 0 if it is a file. |
| **-s** | Check if the path is not empty, return 0 if a path is not empty. |
| **-r** | return 0 if the path exists and read permission is granted |
| **-w** | return 0 if the path exists and write permission is granted |
| **-z** | Checks whether the file size is 0 byte or not, return 0 if the file is of 0 bytes. |

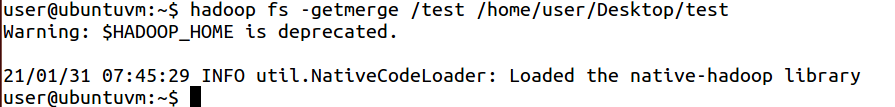
Example



**getmerge**

getmerge command merges a list of files in a directory on the HDFS filesystem into a single local file on the local filesystem.

Example



**stat** prints the statistics about the file or directory in the specified format.

Formats:

%b – file size in bytes

%g – group name of owner

%n – file name

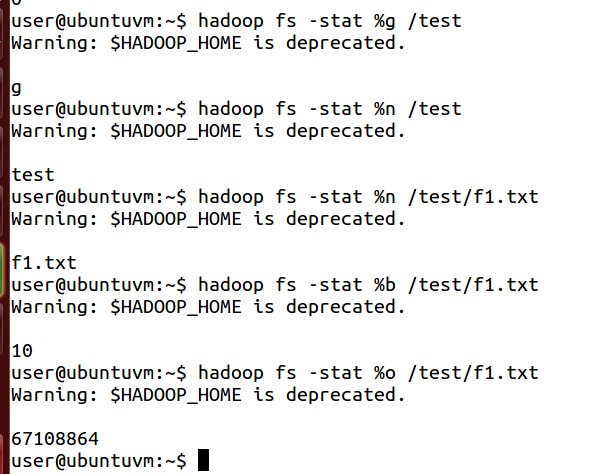
%o – block size

%r – replication

%u – user name of owner

%y – modification date

**Example**

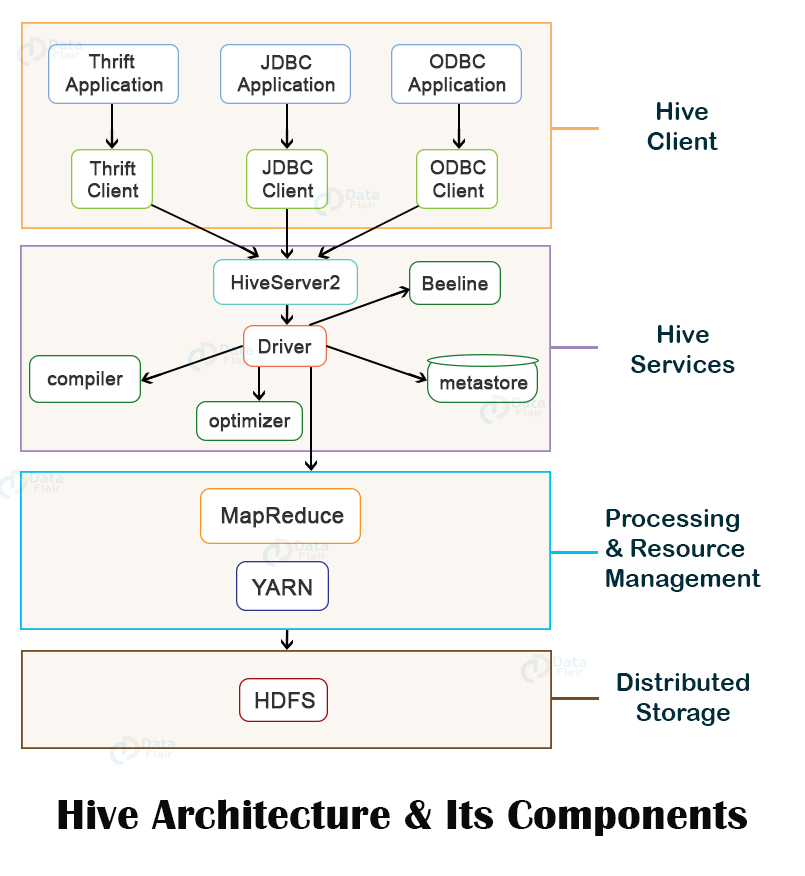


**Practical-6**

**Aim: To study basic commands available for HIVE Query Language.**

**Description:**

Apache Hive is an open-source data warehousing tool for performing distributed processing and data analysis. It was developed by Facebook to reduce the work of writing the Java MapReduce program. Apache Hive uses a Hive Query language, which is a declarative language similar to SQL. Hive translates the hive queries into MapReduce programs. It supports developers to perform processing and analyses on structured and semi-structured data by replacing complex java MapReduce programs with hive queries. One who is familiar with SQL commands can easily write the hive queries.



Hive supports applications written in any language like Python, Java, C++, Ruby, etc. using JDBC, ODBC, and Thrift drivers, for performing queries on the Hive. Hence, one can easily write a hive client application in any language of its own choice.

Hive clients are categorized into three types:

1. Thrift Clients

The Hive server is based on Apache Thrift so that it can serve the request from a thrift client.

2. JDBC client

Hive allows for the Java applications to connect to it using the JDBC driver. JDBC driver uses Thrift to communicate with the Hive Server.

3. ODBC client

Hive ODBC driver allows applications based on the ODBC protocol to connect to Hive. Similar to the JDBC driver, the ODBC driver uses Thrift to communicate with the Hive Server.

**Hive - Create Database**

In Hive, the database is considered as a catalog or namespace of tables. So, we can maintain multiple tables within a database where a unique name is assigned to each table. Hive also provides a default database with a name default.

Initially, we check the default database provided by Hive. So, to check the list of existing databases, follow the below command: -

hive> show databases;

hive> create database demo;

hive> show databases;

hive> describe database extended demo;

**Hive - Create Table**

In Hive, we can create a table by using the conventions similar to the SQL. It supports a wide range of flexibility where the data files for tables are stored. It provides two types of table: -

**Internal table**

The internal tables are also called managed tables as the lifecycle of their data is controlled by the Hive. By default, these tables are stored in a subdirectory under the directory defined by hive.metastore.warehouse.dir (i.e. /user/hive/warehouse). The internal tables are not flexible enough to share with other tools like Pig. If we try to drop the internal table, Hive deletes both table schema and data

hive> create table demo.employee (Id int, Name string , Salary float)

row format delimited

fields terminated by ',' ;

**External Table**

The external table allows us to create and access a table and a data externally. The external keyword is used to specify the external table, whereas the location keyword is used to determine the location of loaded data. As the table is external, the data is not present in the Hive directory. Therefore, if we try to drop the table, the metadata of the table will be deleted, but the data still exists.

Let's create a directory on HDFS by using the following command: -

hadoop dfs -mkdir /HiveDirectory

Now, store the file on the created directory.

Hadoop dfs -put hive/emp\_details /HiveDirectory

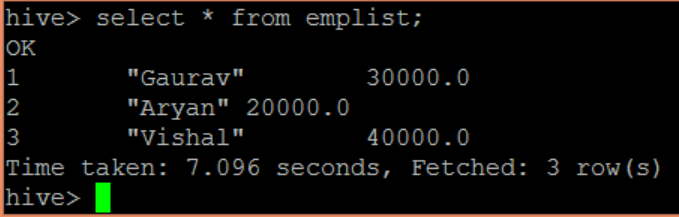
hive> create external table emplist (Id int, Name string , Salary float)

row format delimited

fields terminated by ','

location '/HiveDirectory';

select \* from emplist;



**Hive - Load Data**

Once the internal table has been created, the next step is to load the data into it. So, in Hive, we can easily load data from any file to the database.

load data local inpath '/home/codegyani/hive/emp\_details' into table demo.employee;

select \* from demo.employee;

**Hive - Drop Table**

Hive facilitates us to drop a table by using the SQL drop table command. Let's follow the below steps to drop the table from the database.

show databases;

use demo;

show tables;

drop table new\_employee;

Alter table emp rename to employee\_data;

**Practical-7**

**Aim: Basic commands of HBASE Shell**

**Description:**

HBase is a distributed column-oriented database built on top of the Hadoop file system. It is an open-source project and is horizontally scalable. HBase is a data model that is similar to Google’s big table designed to provide quick random access to huge amounts of structured data. It leverages the fault tolerance provided by the Hadoop File System (HDFS).It is a part of the Hadoop ecosystem that provides random real-time read/write access to data in the Hadoop File System. One can store the data in HDFS either directly or through HBase. Data consumer reads/accesses the data in HDFS randomly using HBase. HBase sits on top of the Hadoop File System and provides read and write access.

Data Definition Language :

1.create

create 'emp', 'personal data', 'professional data'

2.list

list

3.disable

disable 'emp'

4.is\_disabled

is\_disabled 'emp'

5.enable

enable 'emp'

6.is\_enabled

is\_enabled 'emp'

7.describe

describe 'emp'

8.drop

drop 'emp'

Data Manipulation Language :

9.put :

put 'emp','1','personal data:name','raju'

put 'emp','1','personal data:city','hyderabad'

put 'emp','1','professional data:designation','manager'

put 'emp','1','professional data:salary','50000'

put 'emp','1','professional data:vechiv','50000'

put 'emp','2','personal data:name','sathish'

put 'emp','2','personal data:city','bangalore'

put 'emp','2','professional data:designation','professor'

put 'emp','2','professional data:salary','60000'

put 'emp','3','personal data:name','muthu'

put 'emp','3','personal data:city','chennai'

put 'emp','3','professional data:designation','analyst'

put 'emp','3','professional data:salary','20000'

10.get

get 'emp', '1'

11.delete

delete 'emp', '1', 'personal data:city',1417521848375

12.deleteall

deleteall 'emp','1'

13.scan

scan 'emp'

14.count

count 'emp'

15.truncate

truncate 'emp'

**Practical-8**

**Aim: Creating the HDFS tables and loading them in Hive and learn join, partition of tables in Hive.**

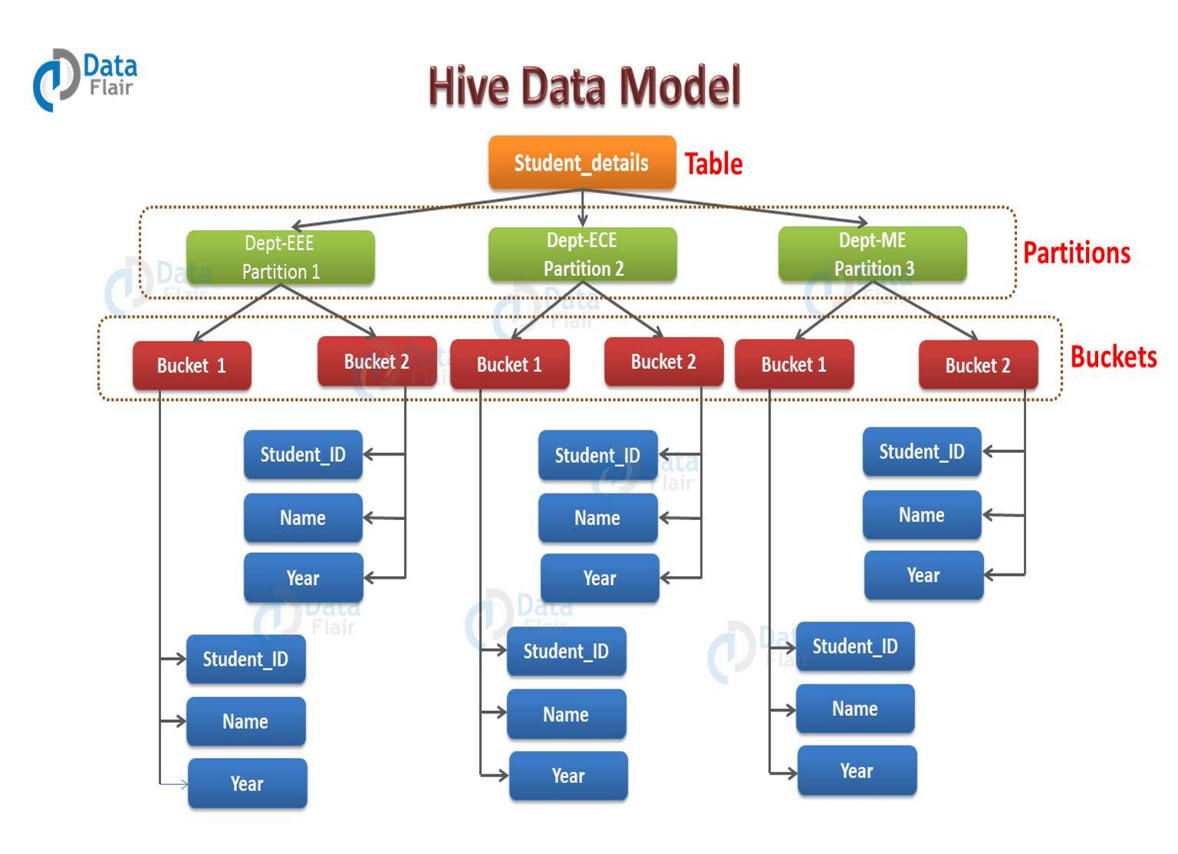
**Description:**

Partitions

Each table can be broken into partitions, Partitions determine distribution of data within subdirectories. In the current century, we know that the huge amount of data which is in the range of petabytes is getting stored in HDFS. So due to this, it becomes very difficult for Hadoop users to query this huge amount of data.

The Hive was introduced to lower down this burden of data querying. Apache Hive converts the SQL queries into MapReduce jobs and then submits it to the Hadoop cluster. When we submit a SQL query, Hive read the entire data-set. So, it becomes inefficient to run MapReduce jobs over a large table. Thus this is resolved by creating partitions in tables. Apache Hive makes this job of implementing partitions very easy by creating partitions by its automatic partition scheme at the time of table creation.

In Partitioning method, all the table data is divided into multiple partitions. Each partition corresponds to a specific value(s) of partition column(s). It is kept as a sub-record inside the table’s record present in the HDFS. Therefore on querying a particular table, appropriate partition of the table is queried which contains the query value. Thus this decreases the I/O time required by the query. Hence increases the performance speed.



**Static partitions**

Insert input data files individually into a partition table is Static Partition. Usually when loading files (big files) into Hive tables static partitions are preferred. Static Partition saves your time in loading data compared to dynamic partition. You “statically” add a partition in the table and move the file into the partition of the table. We can alter the partition in the static partition. You can get the partition column value from the filename, day of date etc without reading the whole big file. If you want to use the Static partition in the hive you should set property set hive.mapred.mode = strict This property set by default in hive-site.xml.Static partition is in Strict Mode. You should use where clause to use limit in the static partition. You can perform Static partition on Hive Manage table or external table.

**Dynamic partitions**

Single insert to partition table is known as a dynamic partition. Usually, dynamic partition loads the data from the non-partitioned table. Dynamic Partition takes more time in loading data compared to static partition. When you have large data stored in a table then the Dynamic partition is suitable. If you want to partition a number of columns but you don’t know how many columns then also dynamic partition is suitable. Dynamic partition there is no required where clause to use limit. We can’t perform alter on the Dynamic partition. You can perform dynamic partition on hive external table and managed table. If you want to use the Dynamic partition in the hive then the mode is in non-strict mode.Here are Hive dynamic partition properties you should allow

1 create database test;

use test;

drop database test

show tables;

drop table student;

show databases;

2 create table student(name string,rollno int,percentage float)partitioned by(state string,city string)row format delimited fields terminated by ',';

3 load data local inpath '/home/training/Desktop/maharastra'

into table student partition(state='maharastra',city='mumbai');

4 load data local inpath '/home/training/Desktop/karnataka'

into table student partition(state='karnataka',city='bangalore');

5select \* from student;

6 select \* from student where state='maharastra';

**Dynamic partitioning**

Note: By default dynamic partioning will be disabled. We need to enable it using the followng command:

7. set hive.exec.dynamic.partition=true;

8. set hive.exec.dynamic.partition.mode=nonstrict;

9. create table stu(name string, rollno int, percentage float, state string, city string) row format delimited fields terminated by ',';

10. load data local inpath '/home/training/Desktop/Result1' into table stu;

11. create table stud\_part (name string, rollno int, percentage float)

partitioned by (state string, city string)

row format delimited

fields terminated by ',';

12. insert overwrite table stud\_part

partition (state, city)

select name,rollno, percentage

,state,

city

from stu;

13. select \* from stud\_part where city='bangalore';

**Karnataka.txt**

Rajesh,100,78

Abhishek,95,76

Manish,102,89

siva,203,66

sania,204,77

**Maharastra.txt**

ravi,100,56

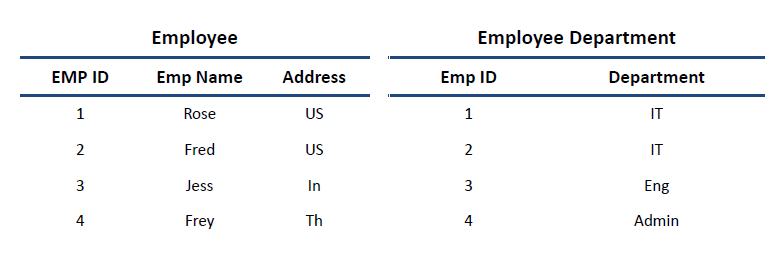
mohan,95,89

mahesh,102,67

janvi,103,66

**Hive Join**

Let's see two tables Employee and Employee Department that are going to be joined.



Employee department table hive DML operation

**Inner joins**

Select \* from employee join employeedepartment ON (employee.empid=employeedepartment.empId)

Next →← Prev

Hive Join

Let's see two tables Employee and EmployeeDepartment that are going to be joined.

Employee department table hive DML operation

**Inner joins**

Select \* from employee join employeedepartment ON (employee.empid=employeedepartment.empId)

**Left outer joins**

Select e.empId, empName, department from employee e Left outer join employeedepartment ed on(e.empId=ed.empId);

**Right outer joins**

Select e.empId, empName, department from employee e Right outer join employeedepartment ed on(e.empId=ed.empId);