Seminar Report

Department:	Mathematical and Computational Sciences
Specialization:	Computational and Data Science
Subject:	Big Data and Analytics
Topic:	MapReduce
Course Instructor:	Dr. Pushpraj Shetty.
Date:	13/4/2021

Team Number 3

- Gavali Deshabhakt Naganath - 202CD005
- Mohammad Ahsan - 202CD016
- Shimpi Mayur Anil - 202CD027

INDEX

Chapter - I

Introduction to Hadoop

Hadoop is an Apache open source framework written in java that allows distributed processing of large datasets across clusters of computers using simple programming models. The Hadoop framework application works in an environment that provides distributed *storage* and *computation* across clusters of computers. Hadoop is designed to scale up from single server to thousands of machines, each offering local computation and storage. Hadoop can manage SQL as well as No SQL databases efficiently. However, it is not preferred for smaller databases.

Key Aspects of Hadoop:

1. **Open source software**
2. **Framework:**

   Hadoop includes everything that we need to develop, analyse and store data.
3. **Distributed:**

   Hadoop divides and stores data across multiple computers. Computation/ Processing is done in parallel across multiple connected nodes/computers.
4. **Massive Storage:**

   Hadoop stores vast amount of data across nodes of low-cost commodity hardware.
5. **Fault Tolerance is Available:**

   In Hadoop data is replicated on various DataNodes in a Hadoop cluster which ensures the availability of data if somehow any of your systems got crashed. You can read all of the data from a single machine if this machine faces a technical issue data can also be read from other nodes in a Hadoop cluster because the data is copied or replicated by default.

History of Hadoop:

Apache Software Foundation is the developers of Hadoop, and it's co-founders are **Doug Cutting** and **Mike Cafarella**. It's co-founder Doug Cutting named it on his son's toy elephant. In October 2003 the first paper release was Google File System. In January 2006, MapReduce development started on the Apache Nutch which consisted of around 6000 lines coding for it and around 5000 lines coding for HDFS. In April 2006 Hadoop 0.1.0 was released.

Core Components of Hadoop Eco-system:

Over the years different generations of hadoop were released and additional components were included in each release. Here, we will take a brief loop at different components in hadoop eco-system

1. HDFS:

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. It is highly fault-tolerant and is designed to be deployed on low-cost hardware. It provides high throughput access to application data and is suitable for applications that have large data sets. HDFS relaxes a few POSIX requirements to enable streaming access to file system data. HDFS was originally built as infrastructure for the Apache Nutch web search engine project

2. YARN: It was introduced in Hadoop 2.0. It allows processing and running batch processing on data, stream processing, interactive processing and graph processing which are stored in HDFS. In this way, it helps to run different types of distributed applications other than MapReduce.

3. MapReduce: MapReduce is a framework using which we can write applications to process huge amounts of data, in parallel, on large clusters of commodity hardware in a reliable manner.

Other Components are,

- Spark: It allows In-Memory data processing
- PIG, HIVE: It used for query-based processing of data services
- HBase: It is used for NoSQL Database
- Mahout, Spark MLLib: These contain Machine Learning algorithm libraries which can be used for data analysis
- Solar, Lucene: These are used for Searching and Indexing
- Zookeeper: Managing cluster
- Oozie: Job Scheduling

Following figure summarizes all the essential components of Hadoop 2.0,

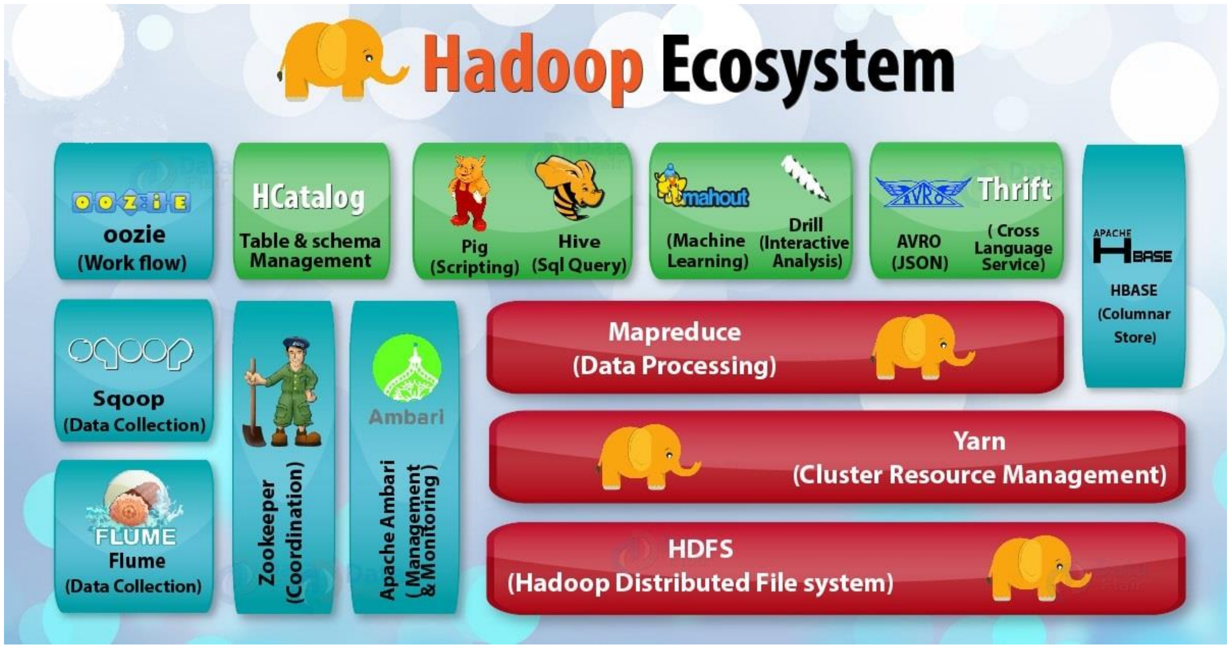


Figure 1: Hadoop Ecosystem

Chapter - II

Hadoop Installation

Now let's discuss the installation process of Hadoop. We will use latest version of both java and Hadoop for installation.

1. **Prerequisites:**

   - Supported Platforms: Hadoop supports both Unix/Linux and Windows OS, but here we will discuss installation process for Unix/Linux OS only. (Follow this for Windows setup)
   - Required software for Linux include:
     i. Java™ must be installed. Recommended Java versions are described at HadoopJavaVersions.
     ii. ssh (secure shell) must be installed and sshd (secure shell daemon) must be running to use the Hadoop scripts that manage remote Hadoop daemons if the optional start and stop scripts are to be used.

2. **Creating a User:**

   At the beginning, it is recommended to create a separate user for Hadoop to isolate Hadoop file system from Unix file system.

   Follow the steps given below to create a user
   i. Open terminal and type 'su' to get root access
   ii. Then type 'useradd <username>' and hit enter (<username> means name by which you want to create new user (preferably hadoop))
   iii. Then to add password to new user account use command 'passwd <username>' and then enter password which you want to set once and hit enter and then again and hit enter.

```
$ su
  password:
# useradd hadoop
# passwd hadoop
  New passwd:
  Retype new passwd
```

3. **SSH Setup and Key Generation:**

   SSH setup is required to do different operations on a cluster such as starting, stopping, distributed daemon shell operations. To authenticate different users of Hadoop, it is required to provide public/private key pair for a Hadoop user and share it with different users. The following commands are used for generating a key value pair using SSH. Copy the public keys form id_rsa.pub to authorized_keys, and provide the owner with read and write permissions to authorized_keys file respectively. **We have to do these steps in hadoop installation directory.**

```
$ ssh-keygen -t rsa
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
$ chmod 0600 ~/.ssh/authorized_keys
```

4. **Java Installation:**

Java is the main prerequisite for Hadoop. First of all, you should verify the existence of java in your system using the command "java -version". The syntax of java version command is given below.

```
$ java -version
```

If java is already on your system then it will give you the output like following stating which version of java is installed in your system.
If java is not installed in your system, then follow the steps given below for installing java.

i. Download java (JDK <latest version> - X64.tar.gz) by visiting the following link and download source code package (.tar.gz file).

```
openjdk version "15.0.2" 2021-01-19
OpenJDK Runtime Environment (build 15.0.2+7)
OpenJDK 64-Bit Server VM (build 15.0.2+7, mixed mode)
```

ii. Generally, you will find the downloaded java file in Downloads folder. Verify it and extract the jdk-8u281-linux-x64.tar.gz file using the following commands.
(Note your package name may differ from one used here.)

```
$ cd Downloads/
$ ls
jdk-8u281-linux-x64.tar.gz
$ tar xvf jdk-8u281-linux-x64.tar.gz
$ ls
jdk-8u281-linux-x64   jdk-7u71-linux-x64.gz
```

iii. Now we'll move the extracted folder to home directory of our new user

```
$ mv /home/<username>/Downloads/jdk-8u281-linux-x6  /home/<username>/
```

iv. For setting up PATH and JAVA_HOME variables, add the following commands to ~/.bashrc file.

```
export JAVA_HOME=/home/hadoop/jdk-8u281-linux-x64.tar.gz
export PATH=$PATH:$JAVA_HOME/bin
```

Now apply all the changes into the current running system and verify that java is properly installed by

```
$ source ~/.bashrc
$ java -version
```

using following commands

5. Downloading and Installing Hadoop:

i. Download and extract Hadoop 3.2.2 from Apache software foundation using the following commands.

```
$ su
password:
# cd /home/hadoop/
# wget http://apache.claz.org/hadoop/common/hadoop-3.2.2/
hadoop-3.2.2.tar.gz
# tar xzf hadoop-3.2.2.tar.gz
# ls
hadoop-3.2.2 hadoop-3.2.2.tar.gz
# exit
```

ii. **Hadoop Operation Modes:**

Once you have downloaded Hadoop, you can operate your Hadoop cluster in one of the three supported modes –

a) Local/Standalone Mode − After downloading Hadoop in your system, by default, it is configured in a standalone mode and can be run as a single java process.

b) Pseudo Distributed Mode − It is a distributed simulation on single machine. Each Hadoop daemon such as hdfs, yarn, MapReduce etc., will run as a separate java process. This mode is useful for development.

c) Fully Distributed Mode − This mode is fully distributed with minimum two or more machines as a cluster.

Here we'll install hadoop in pseudo distributed mode (recommended for single pc users). For installation in standalone mode and fully distributed mode follow this link.

iii. Installing Hadoop in Pseudo Distributed Mode:

Installing hadoop is way different from installing any other operating system or software. For installing hadoop we just need configure hadoop files properly. This is a tricky task because even if you do one step wrong or mis-spell something on the way then your hadoop environment will not work properly. So, follow steps carefully and avoid mistakes.

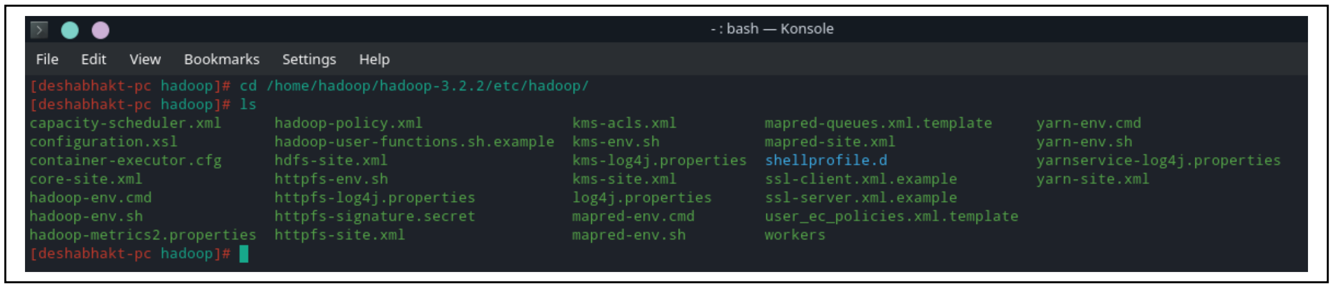
a) Setting up environment variables for Hadoop: You can set Hadoop environment variables by appending the following commands to ~/.bashrc file.

```
export HADOOP_HOME=/home/hadoop/hadoop-3.2.2
export HADOOP_CONF_DIR=/home/hadoop/hadoop-3.2.2/etc/hadoop
export HADOOP_MAPRED_HOME=/home/hadoop/hadoop-3.2.2
export HADOOP_COMMON_HOME=/home/hadoop/hadoop-3.2.2
export HADOOP_HDFS_HOME=/home/hadoop/hadoop-3.2.2
export YARN_HOME=$HADOOP_HOME
export HADOOP_COMMON_LIB_NATIVE_DIR=/home/hadoop/hadoop-
3.2.2/lib/native
export HADOOP_OPTS="-Djava.library.path=/home/hadoop/hadoop-
3.2.2/lib"
export PATH=$PATH:/home/hadoop/hadoop-3.2.2/bin
export HADOOP_PID_DIR=/home/hadoop/hadoop-3.2.2/hadoop_data/hdfs/pid
```

Save the file and exit. Then use ‘source ~/.bashrc’ command to let terminal know the changes.

b) Hadoop Configuration:

This step involves configuring hadoop files. For this open terminal and use following command to go to hadoop configuration directory ‘cd /home/hadoop/hadoop-3.2.2/etc/hadoop/’. The



```
[deshabhakt-pc hadoop]# cd /home/hadoop/hadoop-3.2.2/etc/hadoop/
[deshabhakt-pc hadoop]# ls
capacity-scheduler.xml      hadoop-policy.xml                 kms-acls.xml            mapred-queues.xml.template     yarn-env.cmd
configuration.xsl           hadoop-user-functions.sh.example  kms-env.sh              mapred-site.xml                yarn-env.sh
container-executor.cfg      hdfs-site.xml                     kms-log4j.properties    shellprofile.d                 yarnservice-log4j.properties
core-site.xml               httpfs-env.sh                     kms-site.xml            ssl-client.xml.example         yarn-site.xml
hadoop-env.cmd              httpfs-log4j.properties           log4j.properties        ssl-server.xml.example
hadoop-env.sh               httpfs-signature.secret           mapred-env.cmd          user_ec_policies.xml.template
hadoop-metrics2.properties  httpfs-site.xml                   mapred-env.sh           workers
[deshabhakt-pc hadoop]# 
```

configuration file contains following files.

Now we will use text editor to open the necessary files and add required configuration in those.

i. core-site.xml:

Open the core-site.xml and add the following properties in between <configuration>,

```
<property>
      <name>fs.default.name</name>
      <value>hdfs://localhost:9000</value>
</property>
```

</configuration> tags and save and exit.

ii. hdfs-site.xml:

The hdfs-site.xml file contains information such as the value of replication data, namenode path, and datanode paths of your local file systems. It means the place where you want to store the Hadoop infrastructure. Add the following in hdfs-site.xml and save and exit.

```
<configuration>
   <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
    <property>
        <name>dfs.permission</name>
        <value>false</value>
    </property>
    <property>
        <name>dfs.namenode.name.dir</name>
        <value>/home/hadoop/hadoop-3.2.2/hadoop_data/hdfs/namenode</value>
    </property>
    <property>
        <name>dfs.datanode.data.dir</name>
        <value>/home/hadoop/hadoop-3.2.2/hadoop_data/hdfs/datanode</value>
    </property>
</configuration>
```

iii. yarn-site.xml

This file is used to configure yarn into Hadoop. Open the yarn-site.xml file and add the following properties in between the <configuration>, </configuration> tags in this file.

```
<configuration>
   <property>
      <name>yarn.nodemanager.aux-services</name>
      <value>mapreduce_shuffle</value>
   </property>
</configuration>
```

iv. mapred-site.xml

This file is used to specify which MapReduce framework we are using. Open mapred-site.xml file and add the following properties in between the <configuration>, </configuration>tags in this file.

```
<configuration>
   <property>
      <name>mapreduce.framework.name</name>
      <value>yarn</value>
   </property>
</configuration>
```

v. Adding following Java installation directory path to **hadoop-env.sh**
`‘export JAVA_HOME=/home/hadoop/jdk1.8.0_281’`

d) Verifying Hadoop Installation:

The following steps are used to verify the Hadoop installation.

i. Name Node Setup: Set up the namenode using the command “hdfs namenode -format” as follows and output should be something like this.

```
$ cd ~
$ hdfs namenode -format
4/10/21 21:30:55 INFO namenode.NameNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = localhost/192.168.1.11
STARTUP_MSG:   args = [-format]
STARTUP_MSG:   version = 2.4.1
...
4/10/21 21:30:56 INFO common.Storage: Storage directory
/home/hadoop/hadoopinfra/hdfs/namenode has been successfully format-
ted.
4/10/21 21:30:56 INFO namenode.NNStorageRetentionManager: Going to
retain 1 images with txid >= 0
4/10/21 21:30:56 INFO util.ExitUtil: Exiting with status 0
4/10/21 21:30:56 INFO namenode.NameNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at localhost/192.168.1.11
************************************************************/
```

ii. **Verifying Hadoop dfs:**

The following command is used to start dfs. Executing this command will start your Hadoop file system.

```
$ /home/hadoop/hadoop-3.2.2/sbin/start-dfs.sh
$ hdfs namenode -format
4/10/21 21:37:56
Starting namenodes on [localhost]
localhost: starting namenode, logging to /home/hadoop/hadoop-
3.2.2/logs/hadoop-hadoop-namenode-localhost.out
localhost: starting datanode, logging to /home/hadoop/hadoop-
3.2.2/logs/hadoop-hadoop-datanode-localhost.out
Starting secondary namenodes [0.0.0.0]
```

iii. **Starting hadoop daemons:**

Go to etc/hadoop directory located in hadoop installation folder and enter './start-all.sh'. This command will start all hadoop daemons. Then you can also skip step iv and directly access Web UI.

iv. **Verifying Yarn Script**

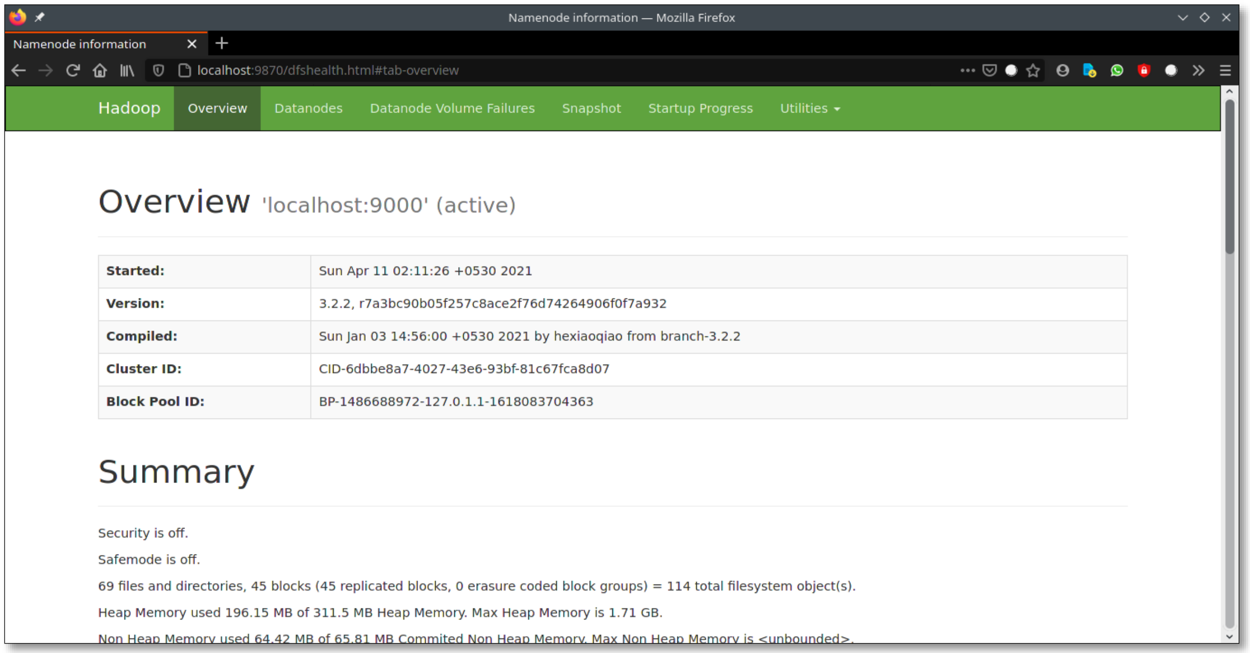
The following command is used to start the yarn script. Executing this command will start your yarn daemons.

```
$ /home/hadoop/hadoop-3.2.2/sbin/start-yarn.sh h
starting yarn daemons
starting resourcemanager, logging to /home/hadoop/hadoop-
3.2.2/logs/yarn-hadoop-resourcemanager-localhost.out
localhost: starting nodemanager, logging to /home/hadoop/hadoop-
3.2.2/logs/yarn-hadoop-nodemanager-localhost.out
```

v. **Accessing Hadoop Web UI**

The default port number to access Hadoop is 9870 (thought it depends on versions of hadoop but for hadoop 3.x.x it's 9870 and all previous versions use 50070). Use the following url to get Hadoop services on browser.

```
http://localhost:9870/
```



Overview 'localhost:9000' (active)

Started:	Sun Apr 11 02:11:26 +0530 2021
Version:	3.2.2, r7a3bc90b05f257c8ace2f76d74264906f0f7a932
Compiled:	Sun Jan 03 14:56:00 +0530 2021 by hexiaoqiao from branch-3.2.2
Cluster ID:	CID-6dbbe8a7-4027-43e6-93bf-81c67fca8d07
Block Pool ID:	BP-1486688972-127.0.1.1-1618083704363

Summary

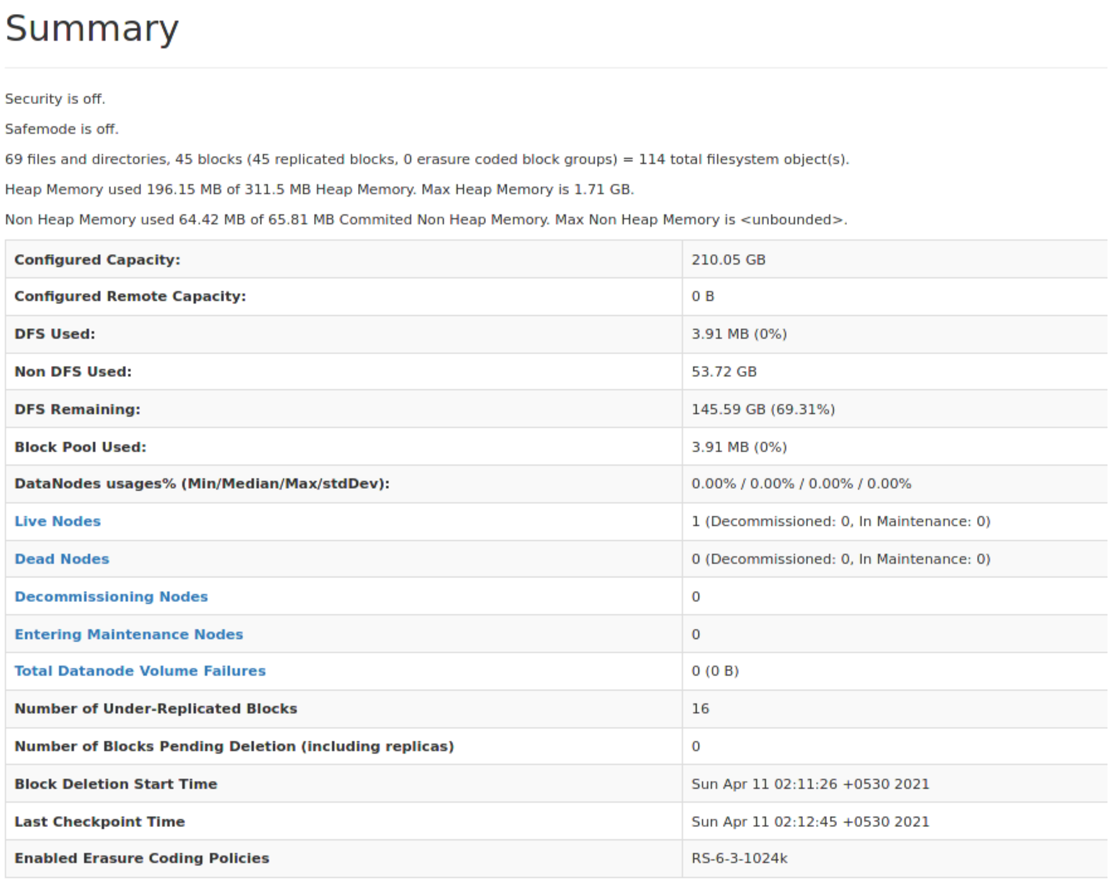
Security is off.

Safemode is off.

69 files and directories, 45 blocks (45 replicated blocks, 0 erasure coded block groups) = 114 total filesystem object(s).

Heap Memory used 196.15 MB of 311.5 MB Heap Memory. Max Heap Memory is 1.71 GB.

Non Heap Memory used 64.42 MB of 65.81 MB Commited Non Heap Memory. Max Non Heap Memory is <unbounded>.



Summary

Security is off.

Safemode is off.

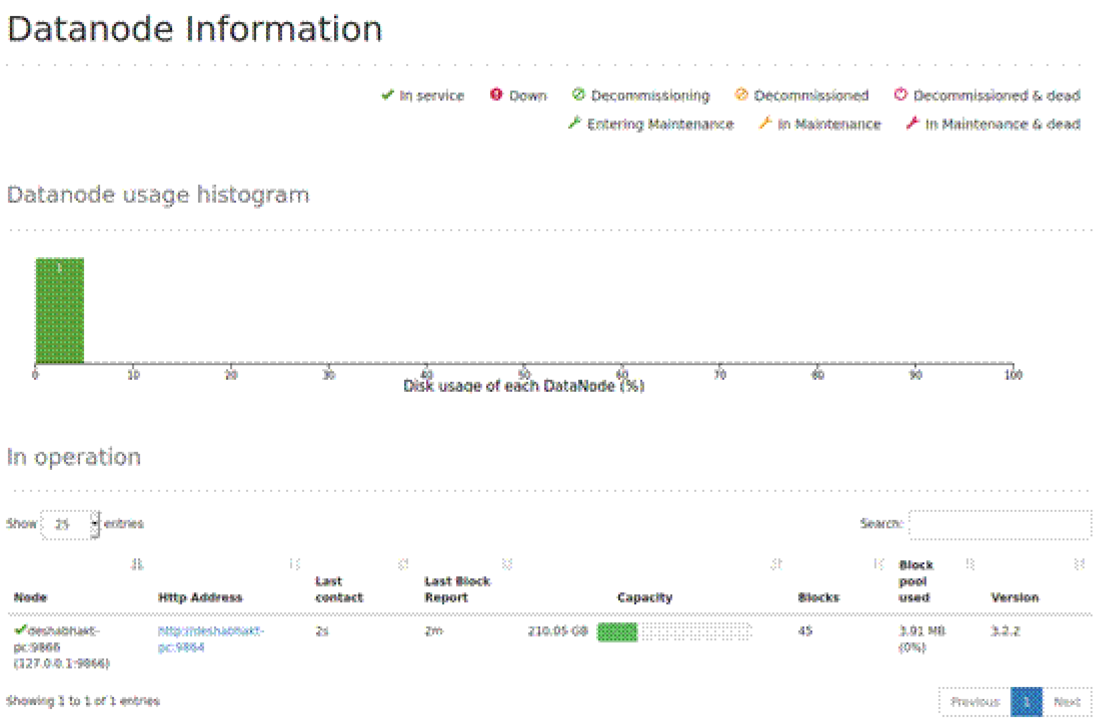
69 files and directories, 45 blocks (45 replicated blocks, 0 erasure coded block groups) = 114 total filesystem object(s).

Heap Memory used 196.15 MB of 311.5 MB Heap Memory. Max Heap Memory is 1.71 GB.

Non Heap Memory used 64.42 MB of 65.81 MB Commited Non Heap Memory. Max Non Heap Memory is <unbounded>.

Configured Capacity:	210.05 GB
Configured Remote Capacity:	0 B
DFS Used:	3.91 MB (0%)
Non DFS Used:	53.72 GB
DFS Remaining:	145.59 GB (69.31%)
Block Pool Used:	3.91 MB (0%)
DataNodes usages% (Min/Median/Max/stdDev):	0.00% / 0.00% / 0.00% / 0.00%
Live Nodes	1 (Decommissioned: 0, In Maintenance: 0)
Dead Nodes	0 (Decommissioned: 0, In Maintenance: 0)
Decommissioning Nodes	0
Entering Maintenance Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	16
Number of Blocks Pending Deletion (including replicas)	0
Block Deletion Start Time	Sun Apr 11 02:11:26 +0530 2021
Last Checkpoint Time	Sun Apr 11 02:12:45 +0530 2021
Enabled Erasure Coding Policies	RS-6-3-1024k

Fig 2 & 3: Overview of Hadoop Environment


Datanode Information

✓ In service ● Down ⊘ Decommissioning ⊘ Decommissioned ⊘ Decommissioned & dead
Entering Maintenance In Maintenance In Maintenance & dead

Datanode usage histogram

In operation

Show 25 entries Search:

Node	Http Address	Last contact	Last Block Report	Capacity	Blocks	Block pool used	Version
✓deshabhakt-pc:9866 (127.0.0.1:9866)	http://deshabhakt-pc:9864	2s	2m	210.05 GB	45	3.01 MB (0%)	3.2.2

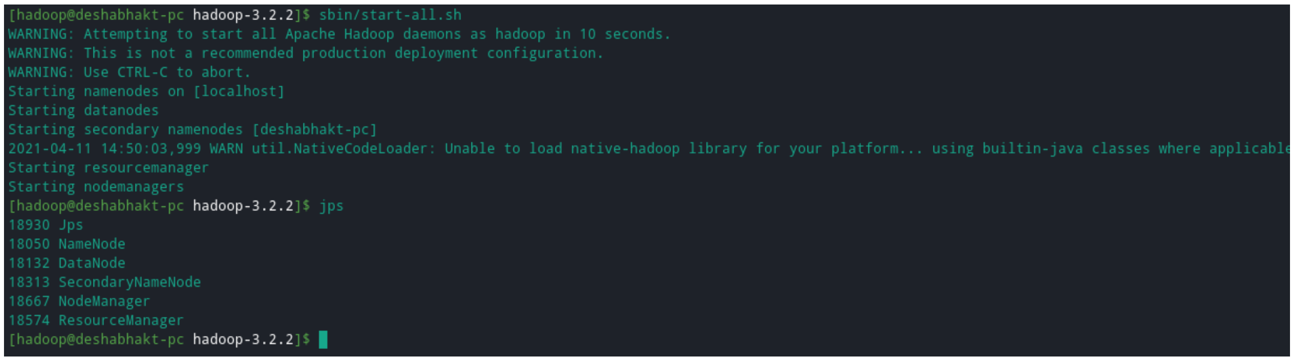
Showing 1 to 1 of 1 entries

Previous 1 Next

Fig 4: DataNode information

Additionally,

1. ./start-all.sh command can be used for starting all hadoop daemons in one go.
2. ‘jps’ command: Used to find out which hadoop daemons are running/active.


```
[hadoop@deshabhakt-pc hadoop-3.2.2]$ sbin/start-all.sh
WARNING: Attempting to start all Apache Hadoop daemons as hadoop in 10 seconds.
WARNING: This is not a recommended production deployment configuration.
WARNING: Use CTRL-C to abort.
Starting namenodes on [localhost]
Starting datanodes
Starting secondary namenodes [deshabhakt-pc]
2021-04-11 14:50:03,999 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Starting resourcemanager
Starting nodemanagers
[hadoop@deshabhakt-pc hadoop-3.2.2]$ jps
18930 Jps
18050 NameNode
18132 DataNode
18313 SecondaryNameNode
18667 NodeManager
18574 ResourceManager
[hadoop@deshabhakt-pc hadoop-3.2.2]$
```
Fig 5: jps and start-all.sh demo

3. Also, we can stop all hadoop daemons with ./stop-all.sh command.
4. All the hadoop executables are located in sbin folder of hadoop installation directory.
5. At any point we can check if hadoop is installed or not by ‘hadoop version’ command

Chapter – III

Introduction to MapReduce

MapReduce is a processing technique and a program model for distributed computing based on java. The MapReduce algorithm contains two important tasks, namely Map and Reduce. Map takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). Secondly, reduce task, which takes the output from a map as an input and combines those data tuples into a smaller set of tuples. As the sequence of the name MapReduce implies, the reduce task is always performed after the map job.

Map task takes care of loading, parsing, transforming, and filtering. The responsibility of reduce task is grouping and aggregating data that is produced by map tasks to generate final output. Each map task is broken into the following phases:

1. Record Reader
2. Mapper
3. Combiner
4. Partitioner

The output produced by map task is known as intermediate keys and values. These intermediate keys and values are sent to reducer. The reduce tasks are broken into the following phases:

1. Shuffle
2. Sort
3. Reducer
4. Output Format

Hadoop assigns map tasks to the DataNode where the actual data to be processed resides. This way, Hadoop ensures data locality. Data locality means that data is not moved over network; only computational code is moved to process data which saves network bandwidth.

MapReduce Workflow:

Following diagram gives a brief idea about MapReduce working schema.

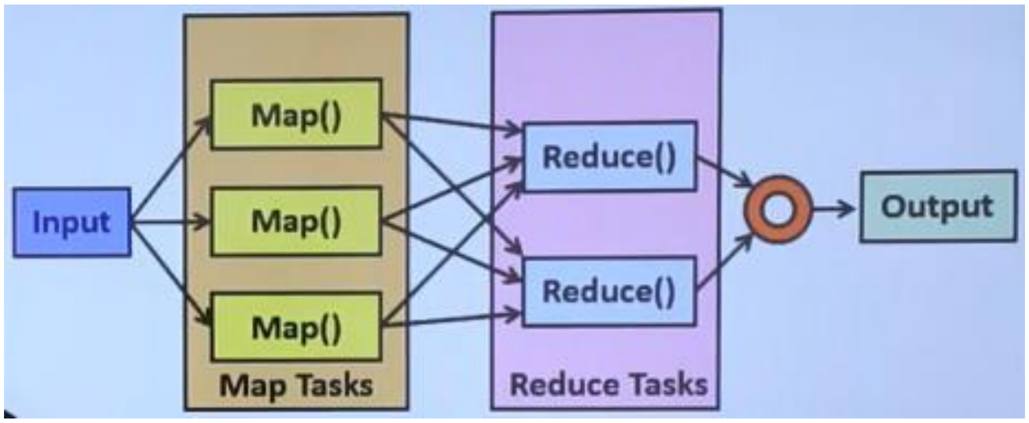


Fig 5: MapReduce Workflow

MapReduce Architecture:

The following diagram shows a MapReduce architecture.

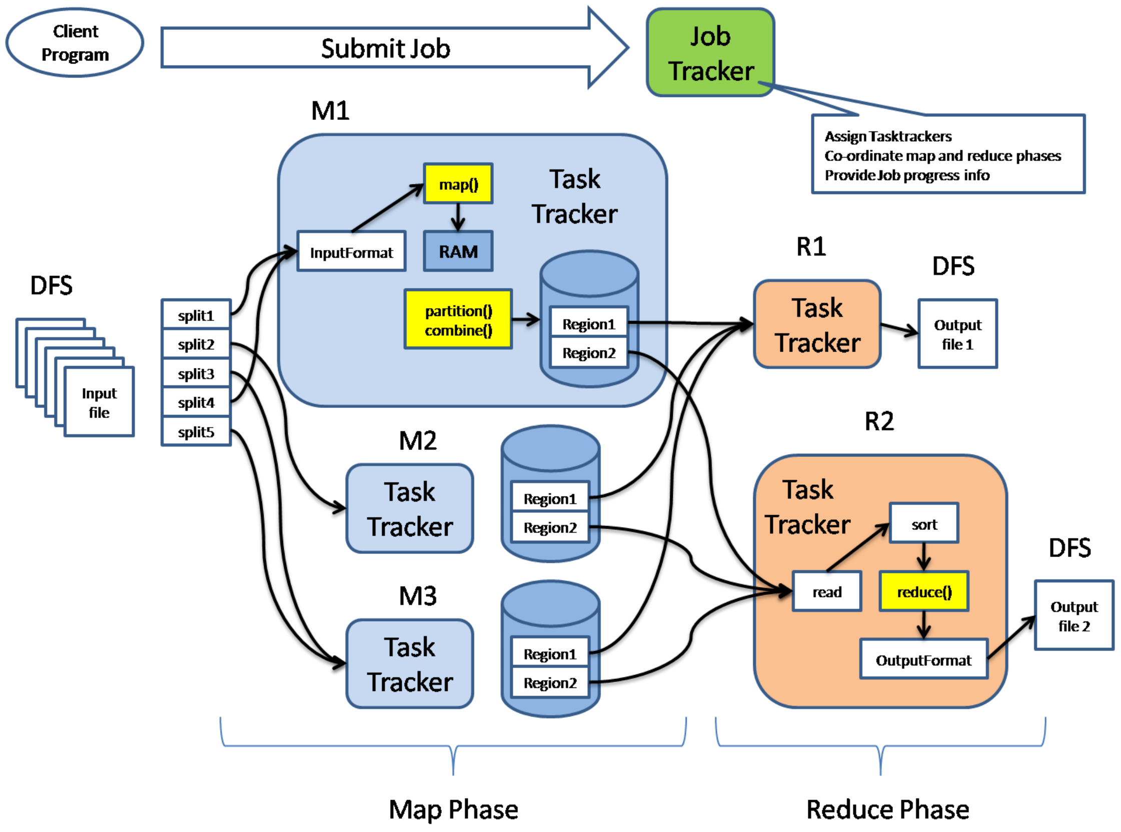


Fig 6: MapReduce Architecture

MapReduce architecture consists of various components. A brief description of these components can improve our understanding on how MapReduce works.

- **Job:** This is the actual work that needs to be executed or processed
- **Task:** This is a piece of the actual work that needs to be executed or processed. A MapReduce job comprises many small tasks that need to be executed.
- **Job Tracker:** This tracker plays the role of scheduling jobs and tracking all jobs assigned to the task tracker.
- **Task Tracker:** This tracker plays the role of tracking tasks and reporting the status of tasks to the job tracker.
- **Input data:** This is the data used to process in the mapping phase.
- **Output data:** This is the result of mapping and reducing.
- **Client:** This is a program or Application Programming Interface (API) that submits jobs to the MapReduce. MapReduce can accept jobs from many clients.
- **Hadoop MapReduce Master:** This plays the role of dividing jobs into job-parts.
- **Job-parts:** These are sub-jobs that result from the division of the main job.

In the MapReduce architecture, clients submit jobs to the MapReduce Master. This master will then sub-divide the job into equal sub-parts. The job-parts will be used for the two main tasks in MapReduce: mapping and reducing. The developer will write logic that satisfies the requirements of the organization or company. The input data will be split and mapped. The intermediate data will then be sorted and merged. The reducer that will generate a final output stored in the HDFS will process the resulting output. The following diagram shows a simplified flow diagram for the MapReduce program.

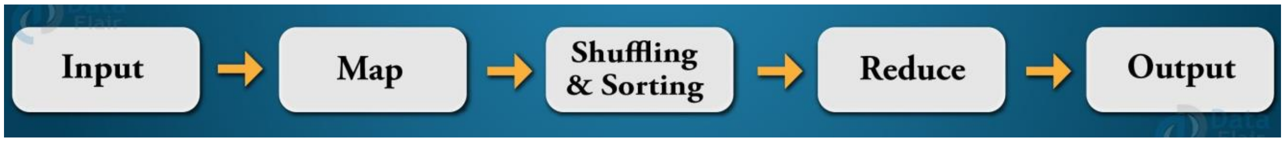
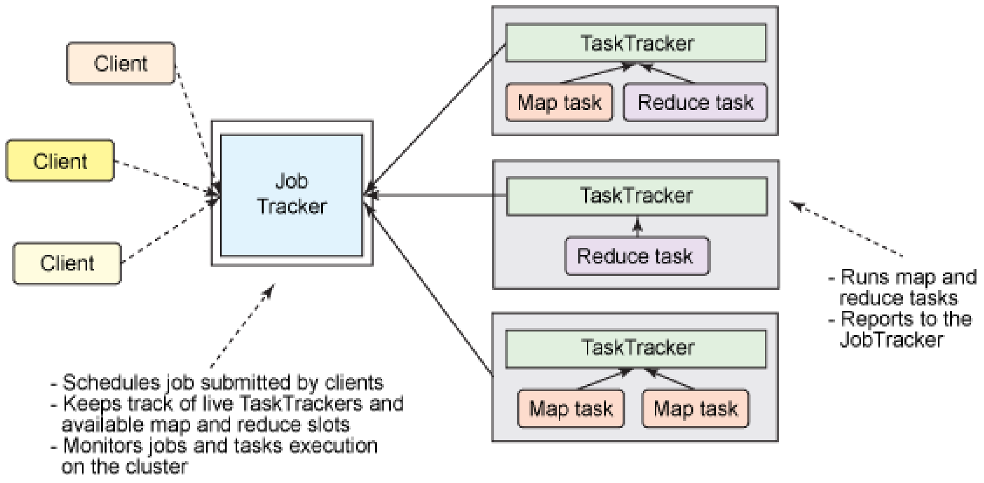


Fig 7: MapReduce Flowchart

Job Trackers and Task Trackers:

Every job consists of two key components: mapping task and reducing task. The map task plays the role of splitting jobs into job-parts and mapping intermediate data. The reduce task plays the role of shuffling and reducing intermediate data into smaller units.

The job tracker acts as a master. It ensures that we execute all jobs. The job tracker schedules jobs that have been submitted by clients. It will assign jobs to task trackers. Each task tracker consists of a map task and reduces the task. Task trackers report the status of each assigned job to the job tracker. The following diagram summarizes how job trackers and task trackers work.



Client
Client
Client
Job Tracker
TaskTracker
Map task
Reduce task
TaskTracker
Reduce task
TaskTracker
Map task
Map task
- Schedules job submitted by clients
- Keeps track of live TaskTrackers and available map and reduce slots
- Monitors jobs and tasks execution on the cluster
- Runs map and reduce tasks
- Reports to the JobTracker

Fig 8: Job Tracker and Task Trackers workflow

Chapter – IV

Mapper & Reducer

A mapper maps the input key−value pairs into a set of intermediate key–value pairs. Maps are individual tasks that have the responsibility of transforming input records into intermediate key–value pairs. We'll first see stages of Mapper in detail and then go to reducer part.

A. Stages/Phases in Mapper

1. **RecordReader:**

   RecordReader converts a byte-oriented view of the input (as generated by the InputSplit) into a record-oriented view and presents it to the Mapper tasks. It presents the tasks with keys and values. Generally, the key is the positional information and value is a chunk of data that constitutes the record.

2. **Map**:

   Map function works on the key–value pair produced by RecordReader and generates zero or more intermediate key–value pairs. The MapReduce decides the key–value pair based on the context.

3. **Combiner:**

   It is an optional function but provides high performance in terms of network bandwidth and disk space. It takes intermediate key–value pair provided by mapper and applies user-specific aggregate function to only that mapper. It is also known as local reducer.

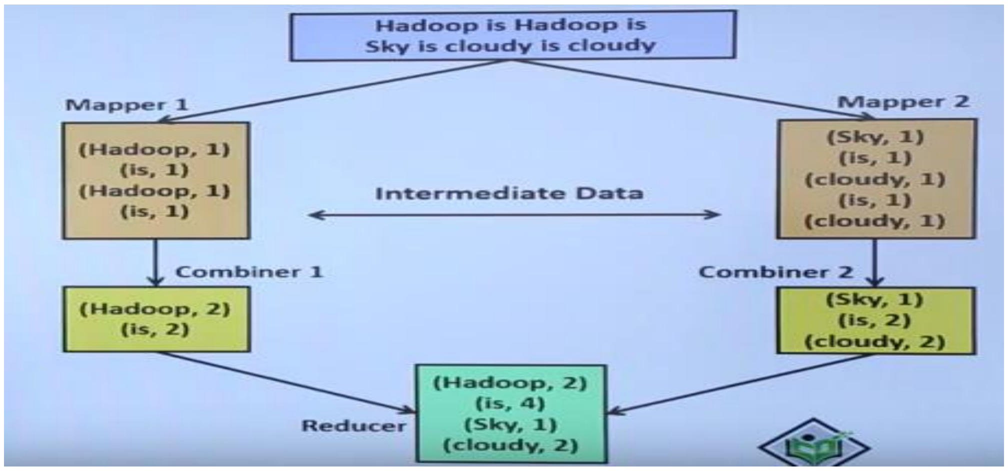


Fig 9: Combiner in action

4. **Partitioner:**

   The partitioner takes the intermediate key–value pairs produced by the mapper, splits them into shard, and sends the shard to the particular reducer as per the user-specific code. Usually, the key with same values goes to the same reducer. The partitioned data of each map task is written to the local disk of that machine and pulled by the respective reducer.

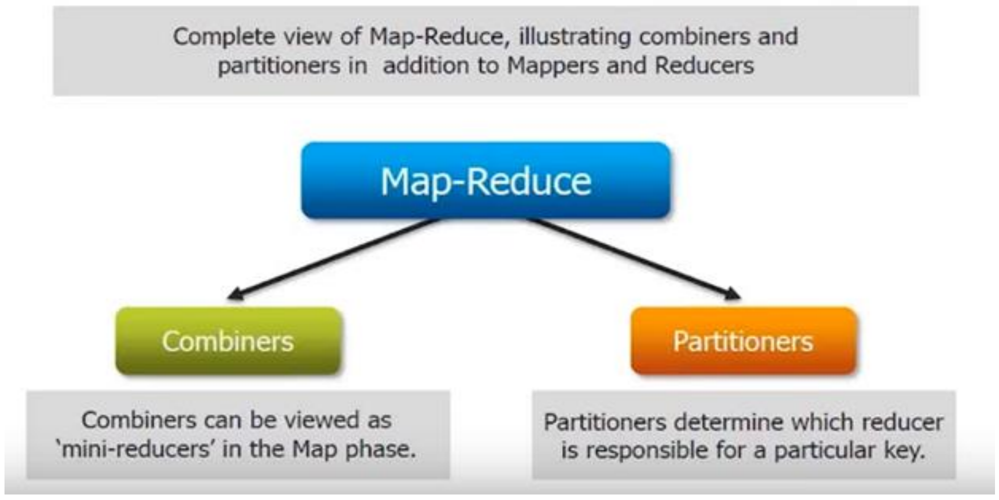

Complete view of Map-Reduce, illustrating combiners and partitioners in addition to Mappers and Reducers
Map-Reduce
Combiners
Partitioners
Combiners can be viewed as 'mini-reducers' in the Map phase.
Partitioners determine which reducer is responsible for a particular key.

Fig 10: Partitioner

B. Stages/Phases in Reducer

The primary chore of the Reducer is to reduce a set of intermediate values (the ones that share a common key) to a smaller set of values. The Reducer has three primary phases: Shuffle and Sort, Reduce, and Output Format.

1. **Shuffle and Sort:**

   This phase takes the output of all the partitioners and downloads them into the local machine where the reducer is running. Then these individual data pipes are sorted by keys which produce larger data list. The main purpose of this sort is grouping similar words so that their values can be easily iterated over by the reduce task.

2. **Reduce:**

   The reducer takes the grouped data produced by the shuffle and sort phase, applies reduce function, and processes one group at a time. The reduce function iterates all the values associated with that key. Reducer function provides various operations such as aggregation, filtering, and combining data. Once it is done, the output (zero or more key–value pairs) of reducer is sent to the output format.

3. **Output Format:**

   The output format separates key–value pair with tab (default) and writes it out to a file using record writer.

Chapter – V

Workflow of MapReduce

The whole process goes through various MapReduce phases of execution, namely, splitting, mapping, sorting and shuffling, and reducing. Let us explore each phase in detail.

1. **InputFiles**

   The data that is to be processed by the MapReduce task is stored in input files. These input files are stored in the Hadoop Distributed File System. The file format is arbitrary, while the line-based log files and the binary format can also be used.

2. **InputFormat**

   It specifies the input-specification for the job. InputFormat validates the MapReduce job input-specification and splits-up the input files into logical InputSplit instances. Each InputSplit is then assigned to the individual Mapper. TextInputFormat is the default InputFormat.

3. **InputSplit**

   It represents the data for processing by the individual Mapper. InputSplit typically presents the byte-oriented view of the input. It is the RecordReader responsibility to process and present the record-oriented view. The default InputSplit is the FileSplit.

4. **RecordReader**

   RecordReader reads the <key, value> pairs from the InputSplit. It converts a byte-oriented view of the input and presents a record-oriented view to the Mapper implementations for processing. It is responsible for processing record boundaries and presenting the Map tasks with keys and values. The record reader breaks the data into the <key, value> pairs for input to the Mapper.

5. **Mapper**

   Mapper maps the input <key, value> pairs to a set of intermediate <key, value> pairs. It processes the input records from the RecordReader and generates the new <key, value> pairs. The <key, value> pairs generated by Mapper are different from the input <key, value> pairs. The generated <key, value> pairs is the output of Mapper known as intermediate output. These intermediate outputs of the Mappers are written to the local disk. The Mappers output is not stored on the Hadoop Distributed File System because this is the temporary data, and writing this data on HDFS will create unnecessary copies. The output of the Mappers is then passed to the Combiner for further processing.

6. **Combiner**

   It is also known as the 'Mini-reducer'. Combiner performs local aggregation on the output of the Mappers. This helps in minimizing data transfer between the Mapper and the Reducer. After the execution of the Combiner function, the output is passed to the Partitioner for further processing.

7. **Partitioner**

   When we are working on the MapReduce program with more than one Reducer then only the Partitioner comes into the picture. For only one reducer, we do not use Partitioner. It partitions the keyspace. It controls the partitioning of keys of the Mapper intermediate outputs. Partitioner takes the output from the Combiner and performs partitioning. Key is for deriving the partition typically through the hash function. The number of partitions is similar to the number of reduce tasks. HashPartitioner is the default Partitioner.

8. Shuffling and Sorting

The input to the Reducer is always the sorted intermediate output of the mappers. After combining and partitioning, the framework via HTTP fetches all the relevant partitions of the output of all the mappers. Once the output of all the mappers is shuffled, the framework groups the Reducer inputs on the basis of the keys. This is then provided as an input to the Reducer.

9. Reducer

Reducer then reduces the set of intermediate values who shares a key to the smaller set of values. The output of reducer is the final output. This output is stored in the Hadoop Distributed File System.

10. RecordWriter

RecordWriter writes the output (key, value pairs) of Reducer to an output file. It writes the MapReduce job outputs to the FileSystem.

11. OutputFormat

The OutputFormat specifies the way in which these output key-value pairs are written to the output files. It validates the output specification for a MapReduce job. OutputFormat basically provides the RecordWriter implementation used for writing the output files of the MapReduce job. The output files are stored in a FileSystem.

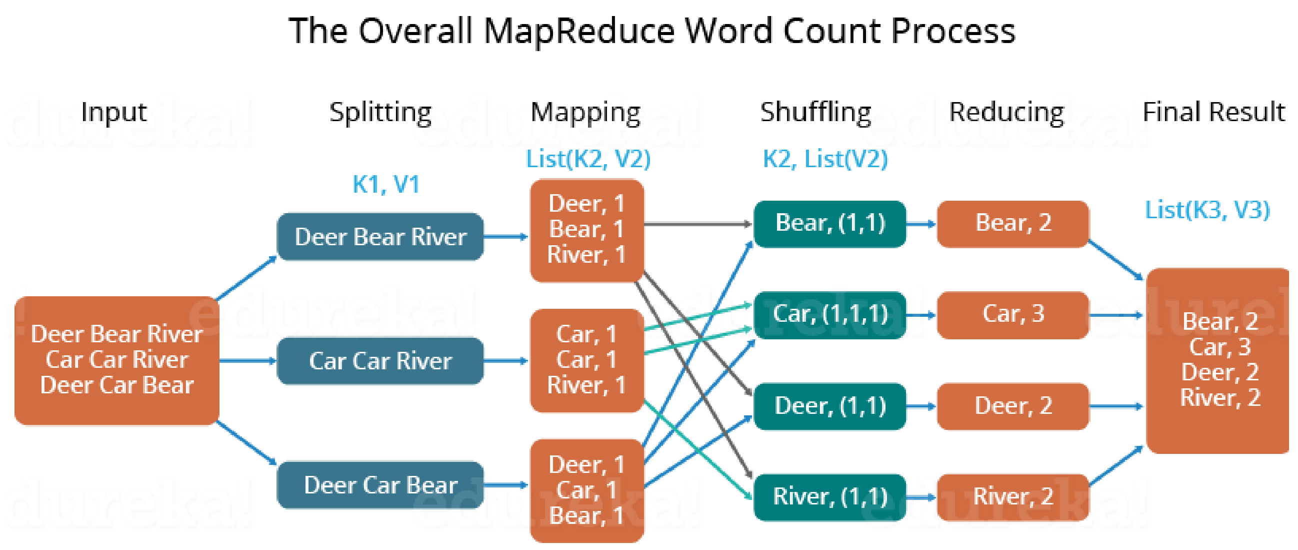


Fig 11. Word Count process Demonstration

In this manner, MapReduce works over the Hadoop cluster in different phases.

Chapter – VI

MapReduce Use Case

K-Means Clustering Algorithm

Clustering-

Clustering is one of the most common exploratory data analysis technique used to get an intuition about the structure of the data. It can be defined as the task of identifying subgroups in the data such that data points in the same subgroup (cluster) are very similar while data points in different clusters are very different. In other words, we try to find homogeneous subgroups within the data such that data points in each cluster are as similar as possible according to a similarity measure such as Euclidean-based distance or correlation-based distance. The decision of which similarity measure to use is application-specific.

Clustering analysis can be done on the basis of features where we try to find subgroups of samples based on features or on the basis of samples where we try to find subgroups of features based on samples. We'll cover here clustering based on features. Clustering is used in market segmentation; where we try to find customers that are similar to each other whether in terms of behaviours or attributes, image segmentation/compression; where we try to group similar regions together, document clustering based on topics, etc.

Unlike supervised learning, clustering is considered an unsupervised learning method since we don't have the ground truth to compare the output of the clustering algorithm to the true labels to evaluate its performance. We only want to try to investigate the structure of the data by grouping the data points into distinct subgroups.

K-Means Algorithm-

K-Means algorithm is an iterative algorithm that tries to partition the dataset into *K* pre-defined distinct non-overlapping subgroups (clusters) where each data point belongs to **only one group**. It tries to make the intra-cluster data points as similar as possible while also keeping the clusters as different (far) as possible. It assigns data points to a cluster such that the sum of the squared distance between the data points and the cluster's centroid (arithmetic mean of all the data points that belong to that cluster) is at the minimum. The less variation we have within clusters, the more homogeneous (similar) the data points are within the same cluster.

Let X = {x1, . . . , xn} be a set of n data points, each with dimension d. The k-means problem seeks to find a set of k means M = {µ1, . . . , µk} which minimizes the function

$$f(M) = \sum_{x \in X} \min_{\mu \in M} ||x - \mu||_2^2$$

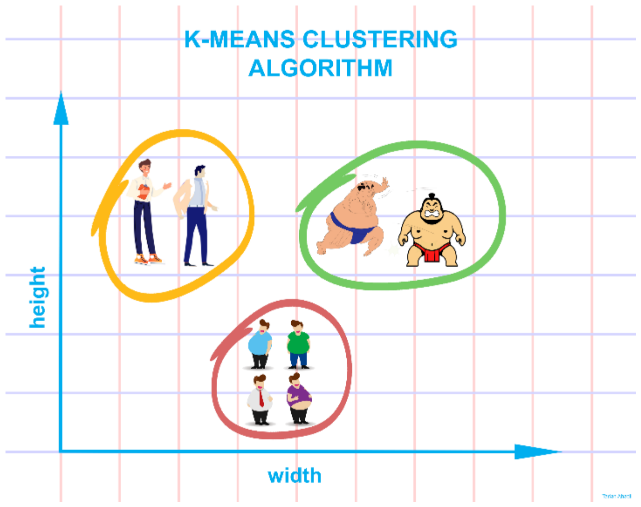
In other words, we wish to choose k means so as to minimize the sum of the squared distances between each point in the dataset and the mean closest to that point.

The standard algorithm for solving k-means uses an iterative process which guarantees a decrease in total error value of the objective function f(M)) on each step [1]. The algorithm is as follows:
Standard k-means

1. Choose k initial means µ1, . . ., µk uniformly at random from the set X.
2. For each point x ∈ X, find the closest mean µi and add x to a set Si
3. For i = 1, . . ., k, set µi to be the centroid of the points in Si

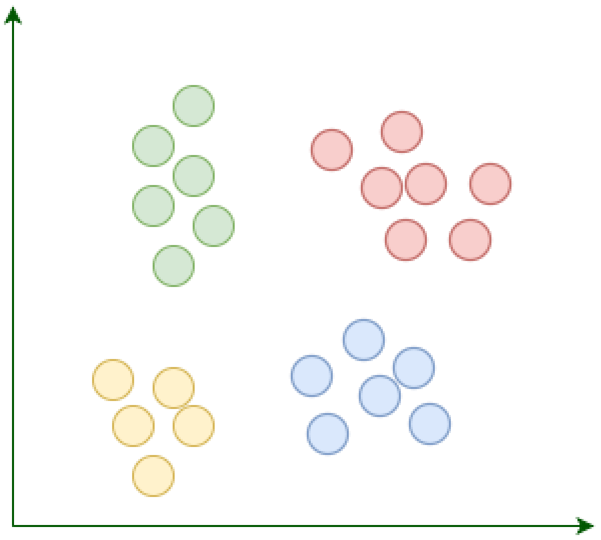
4. Repeat steps 2 and 3 until the means have converged.

The convergence criterion for step 4 is typically when the total error stops changing between steps, in which case a local optimum of the objective function has been reached. However, some implementations terminate the search when the change in error between iterations drops below a certain threshold. Each iteration of this standard algorithm takes time O(nkd). In principle, the number of iterations required for the algorithm to fully converge can be very large, but on real datasets the algorithm typically converges in at most a few dozen iterations

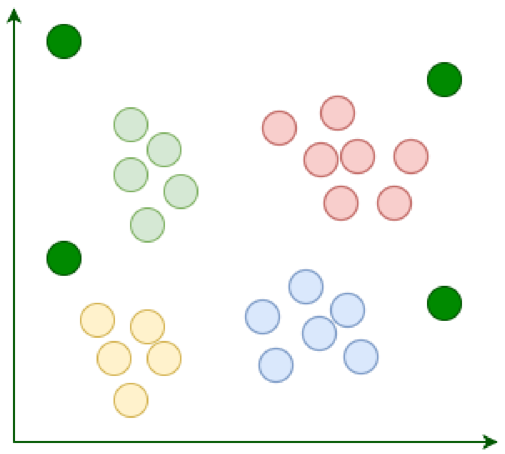


How K-Means works:

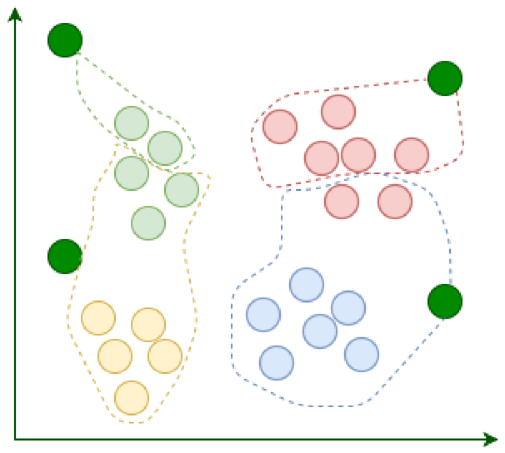
Suppose our goal is to find a few similar groups in a dataset like:



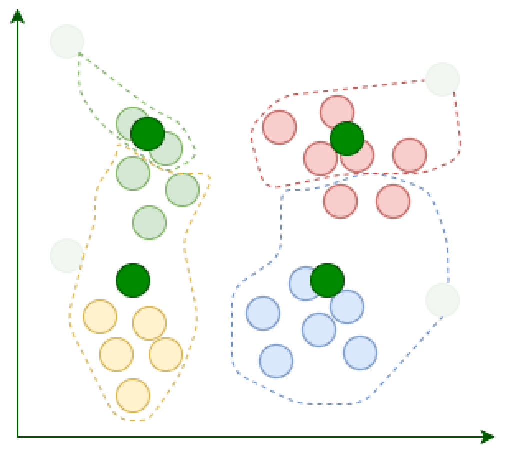
K-Means begins with k randomly placed centroids. **Centroids, as their name suggests, are the center points of the clusters**. For example, here we're adding four random centroids:



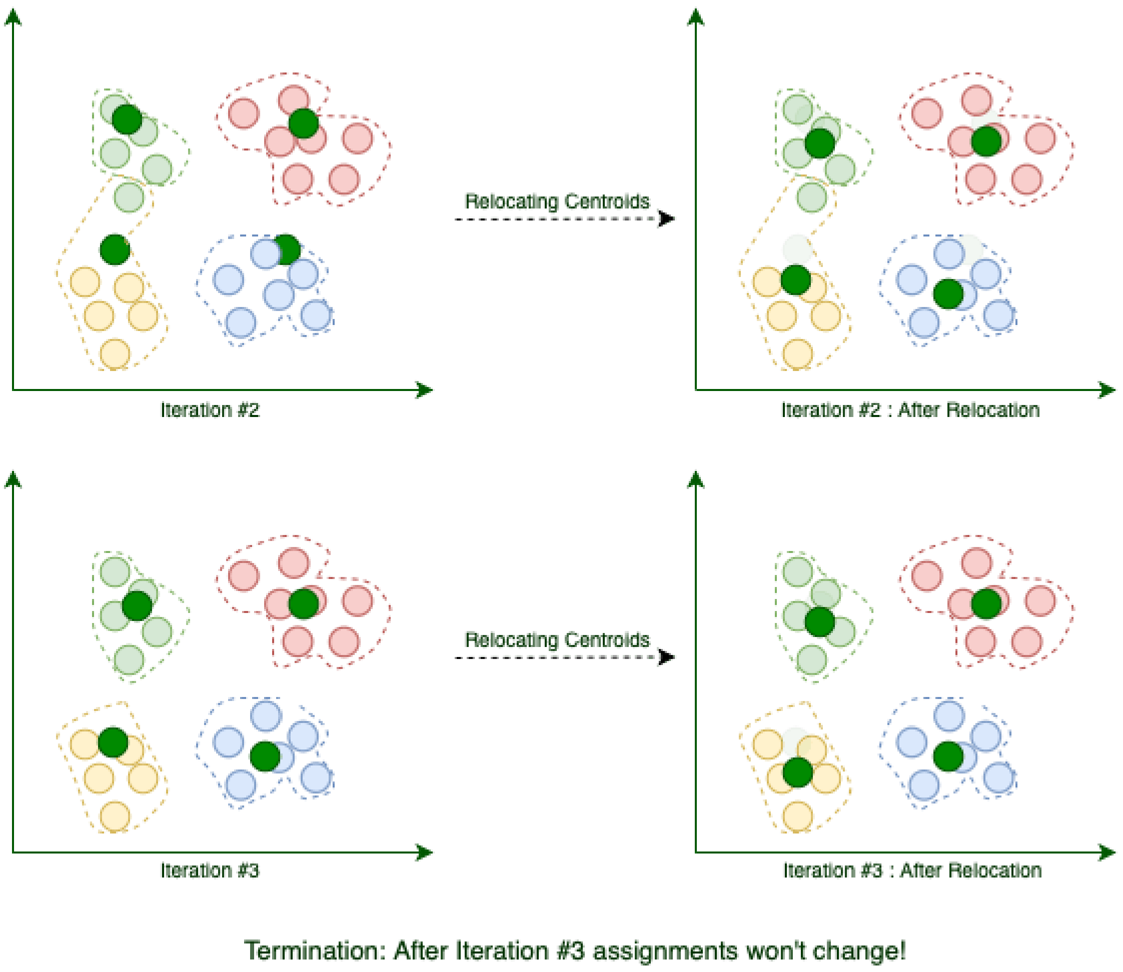
Then we assign each existing data point to its nearest centroid:



After the assignment, we move the centroids to the average location of points assigned to it. Remember, centroids are supposed to be the center points of clusters:



The current iteration concludes each time we're done relocating the centroids. **We repeat these iterations until the assignment between multiple consecutive iterations stops changing:**


Relocating Centroids
Iteration #2
Iteration #2 : After Relocation
Relocating Centroids
Iteration #3
Iteration #3 : After Relocation
Termination: After Iteration #3 assignments won't change!

When the algorithm terminates, those four clusters are found as expected. Now we know how K-Means works.

K-Means via MapReduce:

Recall that each iteration of standard k-means can be divided into two phases, the first of which computes the sets Si of points closest to mean µi, and the second of which computes new means as the centroids of these sets. These two phases correspond to the Map and Reduce phases of our MapReduce algorithm.

The Map phase operates on each point x in the dataset. For a given x, we compute the squared distance between x and each mean and find the mean µi which minimizes this distance. We then emit a key-value pair with this mean's index i as key and the value (x, 1). So our function is

```
k-meansMap(x):
emit (argmini$\|x - \mu i\|_2^{2}$,(x, 1))
```

The Reduce phase is just a straightforward pairwise summation over the values for each key. That is, given two value pairs for a particular key, we combine them by adding each corresponding element in the pairs. So our function is:

```
k-meansReduce (i, [(x, s),(y, t)]):
```

return (i,(x + y, s + t))

The MapReduce characterized by these two functions produces a set of k values of the form

$$\left(i, \left(\sum_{x \in S_i} x,\ |S_i|\right)\right)$$

Where Si denotes the set of points closest to mean µi. We can then compute the new means (the centroids of the sets Si) as

$$\mu_i \leftarrow \frac{1}{|S_i|} \sum_{x \in S_i} x$$

Note that in order for the Map function to compute the distance between a point x and each of the means, each machine in our distributed cluster must have the current set of means. We must therefore broadcast the new means across the cluster at the end of each iteration.

We can write the entire algorithm (henceforth referred to as K-MEANS) as follows:

K-MEANS

1. Choose k initial means µ1, . . ., µk uniformly at random from the set X.
2. Apply the MapReduce given by k-meansMap and k-meansReduce to X.
3. Compute the new means µ1, s. . ., µk from the results of the MapReduce.
4. Broadcast the new means to each machine on the cluster.
5. Repeat steps 2 through 4 until the means have converged.

In the Map phase of this algorithm, we must do O(knd) total work. The total communication cost is O(nd), and largest number of elements associated with a key in the Reduce phase is O(n). However, since our Reduce function is commutative and associative, we can use combiners and bring down the communication cost to O(kd) from each machine.

Also, after the Map phase is completed, we must perform a one-to-all communication to broadcast the new set of means with size O(kd) out to all the machines in the cluster. So in total, each iteration of K-MEANS does O(knd) work (the same as the serial algorithm) and has communication cost O(kd) when combiners are used, which can be broadcast in all-to-one and one-to-all communication patterns. As noted in the introduction, the number of iterations required for convergence can theoretically be quite large, but in practice it is typically a few dozen even for large datasets. So in this way we can use MapReduce for K-Means Clustering Algorithm.

Chapter – VI

Advantages and Disadvantages of Hadoop

After studying almost everything about MapReduce, now we'll discuss the advantages of Hadoop. They are as follows

1. **Scalability**

   Hadoop is a platform that is highly scalable. This is largely because of its ability to store as well as distribute large data sets across plenty of servers. These servers can be inexpensive and can operate in parallel. And with each addition of servers one adds more processing power.

   Contrary to the traditional relational database management systems (RDMS) that cannot scale in order to process huge amounts of data, Hadoop MapReduce programming enables business organizations to run applications from a huge number of nodes that could involve the usage of thousands of terabytes of data.

2. **Cost-effective solution**

   Hadoop's highly scalable structure also implies that it comes across as a very cost-effective solution for businesses that need to store ever growing data dictated by today's requirements.

   In the case of traditional relational database management systems, it becomes massively cost prohibitive to scale to the degrees possible with Hadoop, just to process data. As such, many of the businesses would have to downsize data and further implement classifications based on assumptions of how certain data could be more valuable that the other. In the process, raw data would have to be deleted. This basically serves short term priorities, and if a business happens to change its plans somewhere down the line, the complete set of raw data would be unavailable for later usage.

   Hadoop's scale-out architecture with MapReduce programming, allows the storage and processing of data in a very affordable manner. It can also be used in later times. In fact, the cost savings are massive and costs can reduce from thousands and figures to hundred figures for every terabyte of data.

3. **Flexibility**

   Business organizations can make use of Hadoop MapReduce programming to have access to various new sources of data and also operate on different types of data, whether they are structured or unstructured. This allows them to generate value from all of the data that can be accessed by them.

   Along such lines, Hadoop offers support for numerous languages that can be used for data processing and storage. Whether the data source is social media, email, or clickstream, MapReduce can work on all of them. Also, Hadoop MapReduce programming allows for many applications, such as recommendation systems, processing of logs, marketing analysis, warehousing of data and fraud detection.

4. **Speed**

   Hadoop uses a storage method known as distributed file system, which basically implements a mapping system to locate data in a cluster. The tools used for data processing, such as MapReduce programming, are also generally located in the very same servers, which allows for faster processing of data.

   Even if you happen to be dealing with large volumes of data that is unstructured, Hadoop MapReduce takes minutes to process terabytes of data, and hours for petabytes of data.

5. **Security and Authentication**

   Security is a vital aspect of any application. If any unlawful person or organization had access to multiple petabytes of your organization's data, it can do you massive harm in terms of business dealings and operations.

   In this regard, MapReduce works with HDFS and HBase security that allows only approved users to operate on data stored in the system.

6. **Parallel processing**

   One of the primary aspects of the working of MapReduce programming is that it divides tasks in a manner that allows their execution in parallel.

   Parallel processing allows multiple processors to take on these divided tasks, such that they run entire programs in less time.

7. **Availability and resilient nature**

   When data is sent to an individual node in the entire network, the very same set of data is also forwarded to the other numerous nodes that make up the network. Thus, if there is any failure that affects a particular node, there are always other copies that can still be accessed whenever the need may arise. This always assures the availability of data.

   One of the biggest advantages offered by Hadoop is that of its fault tolerance. Hadoop MapReduce has the ability to quickly recognize faults that occur and then apply a quick and automatic recovery solution. This makes it a game changer when it comes to big data processing.

8. **Simple model of programming**

   Among the various advantages that Hadoop MapReduce offers, one of the most important ones is that it is based on a simple programming model. This basically allows programmers to develop MapReduce programs that can handle tasks with more ease and efficiency.

   The programs for MapReduce can be written using Java, which is a language that isn't very hard to pick up and is also used widespread. Thus, it is easy for people to learn and write programs that meets their data processing needs sufficiently.

Now we’ll discuss disadvantages of Hadoop.

1. **Not beginner friendly**

   From installing hadoop to using MapReduce programming in it is a big task for beginners. Hadoop being open-source, there are many resources on internet and one might get confused while using those. Although from release of first version of hadoop lot has changed now, the apache hadoop documentation is well organised and neatly prepared. So we emphasis more on use it at beginning.

2. **Not Fit for Small Data**

   While big data is not exclusively made for big businesses, not all big data platforms are suited for small data needs. Unfortunately, Hadoop happens to be one of them. Due to its high capacity design, the Hadoop Distributed File System, lacks the ability to efficiently support the random reading of small files. As a result, it is not recommended for organizations with small quantities of data.

3. **Potential Stability Issues**

   Like all open source software, Hadoop has had its fair share of stability issues. To avoid these issues, organizations are strongly recommended to make sure they are running the latest stable version, or run it under a third-party vendor equipped to handle such problems.

References:

1. **Acharya, Seema; Subhashini Chellappan. Big Data and Analytics, 2ed (p. 253). Kindle Edition.**
2. **https://hadoop.apache.org/docs/stable/index.html**
3. **Tutorialspoint.com**
4. **Javatpoint.com**
5. **GeeksForGeeks**
6. **https://stanford.edu/~rezab/classes/cme323/S16/projects_reports/bodoia.pdf**
7. **https://www.coursera.org/lecture/ml-clustering-and-retrieval/mapreduce-for-k-means-EhCYk**
8. **https://medium.com/@tarlanahad/a-friendly-introduction-to-k-means-clustering-algorithm-b31ff7df7ef1**
9. **https://www.baeldung.com/java-k-means-clustering-algorithm**

COVID-19 Database Analysis

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Importing Database

In [2]:
```
df = pd.read_csv('./covid 19 data.csv')
```

In [3]:
```
df.head(5)
```

Out[3]:

	Sno	Date	Time	State/UnionTerritory	ConfirmedIndianNational	ConfirmedForeignNational	Cured	Death
0	1	30/01/20	6:00 PM	Kerala	1	0	0	
1	2	31/01/20	6:00 PM	Kerala	1	0	0	
2	3	01/02/20	6:00 PM	Kerala	2	0	0	
3	4	02/02/20	6:00 PM	Kerala	3	0	0	
4	5	03/02/20	6:00 PM	Kerala	3	0	0	

In [4]:
```
df.tail(5)
```

Out[4]:

	Sno	Date	Time	State/UnionTerritory	ConfirmedIndianNational	ConfirmedForeignNational	Cured
9286	9287	09/12/20	8:00 AM	Telengana	-	-	266120
9287	9288	09/12/20	8:00 AM	Tripura	-	-	32169
9288	9289	09/12/20	8:00 AM	Uttarakhand	-	-	72435
9289	9290	09/12/20	8:00 AM	Uttar Pradesh	-	-	528832
9290	9291	09/12/20	8:00 AM	West Bengal	-	-	475425

Dropping Un-neccesary Columns

In [5]:
```
df = df.drop(['Sno','Time','ConfirmedIndianNational','ConfirmedForeignNational
'],axis=1)
```

In [6]:
```
df = df.rename(columns={'State/UnionTerritory':'State'})
```

In [7]:
```
df
```
Out[7]:

	Date	State	Cured	Deaths	Confirmed
0	30/01/20	Kerala	0	0	1
1	31/01/20	Kerala	0	0	1
2	01/02/20	Kerala	0	0	2
3	02/02/20	Kerala	0	0	3
4	03/02/20	Kerala	0	0	3
...	...	...	...	...	...
9286	09/12/20	Telengana	266120	1480	275261
9287	09/12/20	Tripura	32169	373	32945
9288	09/12/20	Uttarakhand	72435	1307	79141
9289	09/12/20	Uttar Pradesh	528832	7967	558173
9290	09/12/20	West Bengal	475425	8820	507995

9291 rows × 5 columns

In [8]:
```
df['Date'] = pd.to_datetime(df['Date'], dayfirst = True)
```

In [9]:
```
df
```
Out[9]:

	Date	State	Cured	Deaths	Confirmed
0	2020-01-30	Kerala	0	0	1
1	2020-01-31	Kerala	0	0	1
2	2020-02-01	Kerala	0	0	2
3	2020-02-02	Kerala	0	0	3
4	2020-02-03	Kerala	0	0	3
...	...	...	...	...	...
9286	2020-12-09	Telengana	266120	1480	275261
9287	2020-12-09	Tripura	32169	373	32945
9288	2020-12-09	Uttarakhand	72435	1307	79141
9289	2020-12-09	Uttar Pradesh	528832	7967	558173
9290	2020-12-09	West Bengal	475425	8820	507995

9291 rows × 5 columns

Extracing States Names from database

In [10]:
```
states = df['State'].unique()
```

In [11]:
```
states
```

Out[11]:
```
array(['Kerala', 'Telengana', 'Delhi', 'Rajasthan', 'Uttar Pradesh',
       'Haryana', 'Ladakh', 'Tamil Nadu', 'Karnataka', 'Maharashtra',
       'Punjab', 'Jammu and Kashmir', 'Andhra Pradesh', 'Uttarakhand',
       'Odisha', 'Puducherry', 'West Bengal', 'Chhattisgarh',
       'Chandigarh', 'Gujarat', 'Himachal Pradesh', 'Madhya Pradesh',
       'Bihar', 'Manipur', 'Mizoram', 'Andaman and Nicobar Islands',
       'Goa', 'Unassigned', 'Assam', 'Jharkhand', 'Arunachal Pradesh',
       'Tripura', 'Nagaland', 'Meghalaya', 'Dadar Nagar Haveli',
       'Cases being reassigned to states', 'Sikkim', 'Daman & Diu',
       'Dadra and Nagar Haveli and Daman and Diu', 'Telangana',
       'Telangana***', 'Telengana***', 'Maharashtra***', 'Chandigarh***',
       'Punjab***'], dtype=object)
```

Renaming Mis-spelled State names

In [12]:
```
df=df.replace('Telengana','Telangana')
df=df.replace('Telengana***','Telangana')
df=df.replace('Telangana***','Telangana')
df=df.replace('Maharashtra***','Maharashtra')
df=df.replace('Chandigarh***','Chandigarh')
df=df.replace('Punjab***','Punjab')
```

In [13]:
```
states = df['State'].unique()
```

In [14]:
```
states = list(states)   # Numpy array to python list
```

Removing Un-necessary entries from states list

Daman & Diu is removed because of lack of availability of data ---> database contains only 1 entry coresponding to Daman & Diu

In [15]:
```
df[df.State == 'Daman & Diu']
```

Out[15]:

	Date	State	Cured	Deaths	Confirmed
2890	2020-06-11	Daman & Diu	0	0	2

In [16]:
```
states.remove('Cases being reassigned to states')
states.remove('Unassigned')
states.remove('Daman & Diu')
```

```
In [17]: states
```

```
Out[17]: ['Kerala',
          'Telangana',
          'Delhi',
          'Rajasthan',
          'Uttar Pradesh',
          'Haryana',
          'Ladakh',
          'Tamil Nadu',
          'Karnataka',
          'Maharashtra',
          'Punjab',
          'Jammu and Kashmir',
          'Andhra Pradesh',
          'Uttarakhand',
          'Odisha',
          'Puducherry',
          'West Bengal',
          'Chhattisgarh',
          'Chandigarh',
          'Gujarat',
          'Himachal Pradesh',
          'Madhya Pradesh',
          'Bihar',
          'Manipur',
          'Mizoram',
          'Andaman and Nicobar Islands',
          'Goa',
          'Assam',
          'Jharkhand',
          'Arunachal Pradesh',
          'Tripura',
          'Nagaland',
          'Meghalaya',
          'Dadar Nagar Haveli',
          'Sikkim',
          'Dadra and Nagar Haveli and Daman and Diu']
```

```
In [18]: df.isnull().sum()
```

```
Out[18]: Date         0
         State        0
         Cured        0
         Deaths       0
         Confirmed    0
         dtype: int64
```

Adding Months column into database and removing Dates column

```
In [19]: df['Month'] = pd.DatetimeIndex(df['Date']).month
```

In [20]:
```
df
```

Out[20]:

	Date	State	Cured	Deaths	Confirmed	Month
0	2020-01-30	Kerala	0	0	1	1
1	2020-01-31	Kerala	0	0	1	1
2	2020-02-01	Kerala	0	0	2	2
3	2020-02-02	Kerala	0	0	3	2
4	2020-02-03	Kerala	0	0	3	2
...	...	...	...	...	...	...
9286	2020-12-09	Telangana	266120	1480	275261	12
9287	2020-12-09	Tripura	32169	373	32945	12
9288	2020-12-09	Uttarakhand	72435	1307	79141	12
9289	2020-12-09	Uttar Pradesh	528832	7967	558173	12
9290	2020-12-09	West Bengal	475425	8820	507995	12

9291 rows × 6 columns

In [21]:
```
df.drop('Date',axis=1,inplace=True)
```

In [22]:
```
df
```

Out[22]:

	State	Cured	Deaths	Confirmed	Month
0	Kerala	0	0	1	1
1	Kerala	0	0	1	1
2	Kerala	0	0	2	2
3	Kerala	0	0	3	2
4	Kerala	0	0	3	2
...	...	...	...	...	...
9286	Telangana	266120	1480	275261	12
9287	Tripura	32169	373	32945	12
9288	Uttarakhand	72435	1307	79141	12
9289	Uttar Pradesh	528832	7967	558173	12
9290	West Bengal	475425	8820	507995	12

9291 rows × 5 columns

Creating a dictonary with keys as state names and their corresponding data

In [23]:
```
state_dct_df ={}
for i in states:
    state_dct_df[i] = df[df.State == i].copy().reset_index().drop(['index'],axis
=1)
```

```
In [24]: state_dct_df['Maharashtra']
```

Out[24]:

	State	Cured	Deaths	Confirmed	Month
0	Maharashtra	0	0	2	3
1	Maharashtra	0	0	5	3
2	Maharashtra	0	0	2	3
3	Maharashtra	0	0	11	3
4	Maharashtra	0	0	14	3
...	...	...	...	...	...
271	Maharashtra	1710050	47599	1842587	12
272	Maharashtra	1715884	47694	1847509	12
273	Maharashtra	1723370	47734	1852266	12
274	Maharashtra	1730715	47774	1855341	12
275	Maharashtra	1737080	47827	1859367	12

276 rows × 5 columns

Creating a Dictonary to Keep count of number of entries i.e. number of days for which data is available in database corresponding to each state

```
In [25]: row_count_dct_or_day_count_dct = {}
         for i in states:
             rows, columns = state_dct_df[i].shape
             row_count_dct_or_day_count_dct[i] = rows
```

In [26]:
```
row_count_dct_or_day_count_dct
```

Out[26]:
```
{'Kerala': 315,
 'Telangana': 283,
 'Delhi': 283,
 'Rajasthan': 282,
 'Uttar Pradesh': 281,
 'Haryana': 281,
 'Ladakh': 278,
 'Tamil Nadu': 278,
 'Karnataka': 276,
 'Maharashtra': 276,
 'Punjab': 276,
 'Jammu and Kashmir': 276,
 'Andhra Pradesh': 273,
 'Uttarakhand': 270,
 'Odisha': 269,
 'Puducherry': 267,
 'West Bengal': 267,
 'Chhattisgarh': 266,
 'Chandigarh': 266,
 'Gujarat': 265,
 'Himachal Pradesh': 264,
 'Madhya Pradesh': 264,
 'Bihar': 263,
 'Manipur': 261,
 'Mizoram': 260,
 'Andaman and Nicobar Islands': 259,
 'Goa': 259,
 'Assam': 253,
 'Jharkhand': 253,
 'Arunachal Pradesh': 251,
 'Tripura': 247,
 'Nagaland': 207,
 'Meghalaya': 240,
 'Dadar Nagar Haveli': 37,
 'Sikkim': 200,
 'Dadra and Nagar Haveli and Daman and Diu': 181}
```

In [27]:
```
state_dct_df['Maharashtra']
```

Out[27]:

	State	Cured	Deaths	Confirmed	Month
0	Maharashtra	0	0	2	3
1	Maharashtra	0	0	5	3
2	Maharashtra	0	0	2	3
3	Maharashtra	0	0	11	3
4	Maharashtra	0	0	14	3
...	...	...	...	...	...
271	Maharashtra	1710050	47599	1842587	12
272	Maharashtra	1715884	47694	1847509	12
273	Maharashtra	1723370	47734	1852266	12
274	Maharashtra	1730715	47774	1855341	12
275	Maharashtra	1737080	47827	1859367	12

276 rows × 5 columns

Getting Back per day count of cured, deaths and confirmed cases from cumulative sum

In [28]:
```
# Here we're getting back per day cases from cumulative sum

for s in states:
    for i in range(row_count_dct_or_day_count_dct[s]-1,0,-1):
            state_dct_df[s].iloc[i,1] -= state_dct_df[s].iloc[i-1,1]
            state_dct_df[s].iloc[i,2] -= state_dct_df[s].iloc[i-1,2]
            state_dct_df[s].iloc[i,3] -= state_dct_df[s].iloc[i-1,3]
```

In [29]:
```
state_dct_df['Maharashtra']
```

Out[29]:

	State	Cured	Deaths	Confirmed	Month
0	Maharashtra	0	0	2	3
1	Maharashtra	0	0	3	3
2	Maharashtra	0	0	-3	3
3	Maharashtra	0	0	9	3
4	Maharashtra	0	0	3	3
...	...	...	...	...	...
271	Maharashtra	6776	127	5229	12
272	Maharashtra	5834	95	4922	12
273	Maharashtra	7486	40	4757	12
274	Maharashtra	7345	40	3075	12
275	Maharashtra	6365	53	4026	12

276 rows × 5 columns

As some of the entries in database turns out to be negative, now we will get back there indices and aftwerwards remove those entries

Negative entries can occur in cured, deaths and confirmed columns

In [30]:
```
check_neg_cured = {}
check_neg_deaths = {}
check_neg_confirmed = {}
for s in states:
    index_cured = []
    index_deaths = []
    index_confirmed = []

    for i in range(row_count_dct_or_day_count_dct[s]):
        if(state_dct_df[s].iloc[i,1]<0):
            index_cured.append(i)
        if(state_dct_df[s].iloc[i,2]<0):
            index_deaths.append(i)
        if(state_dct_df[s].iloc[i,3]<0):
            index_confirmed.append(i)

    check_neg_cured[s] = index_cured
    check_neg_deaths[s] = index_deaths
    check_neg_confirmed[s] = index_confirmed
```

In [31]:
```
check_neg_cured
```

Out[31]:
```
{'Kerala': [],
 'Telangana': [],
 'Delhi': [],
 'Rajasthan': [],
 'Uttar Pradesh': [11],
 'Haryana': [],
 'Ladakh': [],
 'Tamil Nadu': [],
 'Karnataka': [],
 'Maharashtra': [],
 'Punjab': [],
 'Jammu and Kashmir': [],
 'Andhra Pradesh': [],
 'Uttarakhand': [],
 'Odisha': [],
 'Puducherry': [],
 'West Bengal': [16],
 'Chhattisgarh': [],
 'Chandigarh': [],
 'Gujarat': [],
 'Himachal Pradesh': [],
 'Madhya Pradesh': [],
 'Bihar': [],
 'Manipur': [],
 'Mizoram': [],
 'Andaman and Nicobar Islands': [],
 'Goa': [],
 'Assam': [],
 'Jharkhand': [],
 'Arunachal Pradesh': [],
 'Tripura': [],
 'Nagaland': [],
 'Meghalaya': [],
 'Dadar Nagar Haveli': [],
 'Sikkim': [],
 'Dadra and Nagar Haveli and Daman and Diu': []}
```

Removing all the entries which are invalid/negative for calculation from Statewise database dictonary

In [32]:
```
for i in states:
    state_dct_df[i] = df[df.State == i].copy().reset_index().drop(['index'],axis
=1)
```

In [33]:
```
for s in states:
    state_dct_df[s].drop(check_neg_cured[s],axis=0,inplace=True)
    state_dct_df[s].drop(check_neg_deaths[s],axis=0,inplace=True)
    state_dct_df[s].drop(check_neg_confirmed[s],axis=0,inplace=True)
```

In [34]:
```
state_dct_df['Maharashtra']
```
Out[34]:

	State	Cured	Deaths	Confirmed	Month
0	Maharashtra	0	0	2	3
1	Maharashtra	0	0	5	3
3	Maharashtra	0	0	11	3
4	Maharashtra	0	0	14	3
5	Maharashtra	0	0	14	3
...	...	...	...	...	...
271	Maharashtra	1710050	47599	1842587	12
272	Maharashtra	1715884	47694	1847509	12
273	Maharashtra	1723370	47734	1852266	12
274	Maharashtra	1730715	47774	1855341	12
275	Maharashtra	1737080	47827	1859367	12

274 rows × 5 columns

Calculating Updated row count for each state and putting it in dictonary

In [35]:
```
for i in states:
    rows, columns = state_dct_df[i].shape
    row_count_dct_or_day_count_dct[i] = rows
```

```
In [36]: row_count_dct_or_day_count_dct
```

```
Out[36]: {'Kerala': 315,
          'Telangana': 283,
          'Delhi': 283,
          'Rajasthan': 281,
          'Uttar Pradesh': 280,
          'Haryana': 281,
          'Ladakh': 278,
          'Tamil Nadu': 278,
          'Karnataka': 276,
          'Maharashtra': 274,
          'Punjab': 276,
          'Jammu and Kashmir': 276,
          'Andhra Pradesh': 273,
          'Uttarakhand': 270,
          'Odisha': 269,
          'Puducherry': 266,
          'West Bengal': 266,
          'Chhattisgarh': 266,
          'Chandigarh': 266,
          'Gujarat': 265,
          'Himachal Pradesh': 264,
          'Madhya Pradesh': 264,
          'Bihar': 263,
          'Manipur': 261,
          'Mizoram': 260,
          'Andaman and Nicobar Islands': 259,
          'Goa': 259,
          'Assam': 253,
          'Jharkhand': 252,
          'Arunachal Pradesh': 251,
          'Tripura': 247,
          'Nagaland': 206,
          'Meghalaya': 240,
          'Dadar Nagar Haveli': 37,
          'Sikkim': 200,
          'Dadra and Nagar Haveli and Daman and Diu': 181}
```

Again Calculating Absolute values from cumulative sum

```
In [37]: for s in states:
             for i in range(row_count_dct_or_day_count_dct[s]-1,0,-1):
                     state_dct_df[s].iloc[i,1] -= state_dct_df[s].iloc[i-1,1]
                     state_dct_df[s].iloc[i,2] -= state_dct_df[s].iloc[i-1,2]
                     state_dct_df[s].iloc[i,3] -= state_dct_df[s].iloc[i-1,3]
```

Summing confirmed cases monthwise

```
In [38]: months = np.arange(1,13,1)
         months
```

```
Out[38]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

In [39]:
```
state_dct_df['Maharashtra'].head()
```
Out[39]:

	State	Cured	Deaths	Confirmed	Month
0	Maharashtra	0	0	2	3
1	Maharashtra	0	0	3	3
3	Maharashtra	0	0	6	3
4	Maharashtra	0	0	3	3
5	Maharashtra	0	0	0	3

In [40]:
```
month_wise_sum_of_confirmed_cases_dct={}

# Initializing Dictonary
for i in months:
    month_wise_sum_of_confirmed_cases_dct[i] = 0

# Summing cases month-wise
for s in states:
    for i in range(row_count_dct_or_day_count_dct[s]):
        month_wise_sum_of_confirmed_cases_dct[state_dct_df[s].iloc[i,4]] +=  sta
te_dct_df[s].iloc[i,3]
```
In [41]:
```
month_wise_sum_of_confirmed_cases_dct
```
Out[41]:
```
{1: 1,
 2: 2,
 3: 1356,
 4: 31971,
 5: 143322,
 6: 383210,
 7: 1079034,
 8: 1982375,
 9: 2604518,
 10: 1911356,
 11: 1294572,
 12: 304159}
```

Q1.Filter the month in which heighest people are get infected to Covid-19 virus?

In [42]:
```
peak_month = 0
peak_val = 0
for i in months:
    if month_wise_sum_of_confirmed_cases_dct[i]>peak_val:
        peak_val = month_wise_sum_of_confirmed_cases_dct[i]
        peak_month = i
```
In [43]:
```
print(f"Peak Month = {peak_month}\nPeak Cases = {peak_val}")
```
```
Peak Month = 9
Peak Cases = 2604518
```

Survival Rate = Total Cured/(Total Confirmed*Total number of months for which data is available)

Q2.Obtain state in which survival rate is high

```
In [44]: survival_rate ={}
         for s in states:
             total_confirmed = 0
             total_cured = 0
             for i in range(row_count_dct_or_day_count_dct[s]):
                total_confirmed += state_dct_df[s].iloc[i,3]
                total_cured += state_dct_df[s].iloc[i,1]
             factor = (row_count_dct_or_day_count_dct[s]/365)*12
             survival_rate[s] = (total_cured/(total_confirmed*factor))
```

```
In [45]: survival_rate
```

```
Out[45]: {'Kerala': 0.08722297882343857,
          'Telangana': 0.10391016019493547,
          'Delhi': 0.1017062890076429,
          'Rajasthan': 0.09935099708743428,
          'Uttar Pradesh': 0.1029206425418711,
          'Haryana': 0.10185051486100315,
          'Ladakh': 0.09827481976956065,
          'Tamil Nadu': 0.10631967162991153,
          'Karnataka': 0.1056613715720078,
          'Maharashtra': 0.10370883525849273,
          'Punjab': 0.10163298794009995,
          'Jammu and Kashmir': 0.1036763753975389,
          'Andhra Pradesh': 0.10982446022585154,
          'Uttarakhand': 0.10310857508422613,
          'Odisha': 0.11133686358437969,
          'Puducherry': 0.11127444055718348,
          'West Bengal': 0.10701694750562563,
          'Chhattisgarh': 0.10402583607907026,
          'Chandigarh': 0.10646141162984768,
          'Gujarat': 0.10525413909896267,
          'Himachal Pradesh': 0.09444154620598312,
          'Madhya Pradesh': 0.10639309264609538,
          'Bihar': 0.11238642781192737,
          'Manipur': 0.10227842776596245,
          'Mizoram': 0.11095691250330426,
          'Andaman and Nicobar Islands': 0.11421900732281647,
          'Goa': 0.1126126783631637,
          'Assam': 0.11765568343362025,
          'Jharkhand': 0.11771077809613047,
          'Arunachal Pradesh': 0.11567467430272826,
          'Tripura': 0.1202438059260834,
          'Nagaland': 0.13867538753051514,
          'Meghalaya': 0.11934232026143791,
          'Dadar Nagar Haveli': 0.06323631323631324,
          'Sikkim': 0.13808525087887502,
          'Dadra and Nagar Haveli and Daman and Diu': 0.1669947620154617}
```

```
In [46]: max_survival_rate_state = 0
         max_survival_rate = 0
         for i in states:
             if survival_rate[i]>max_survival_rate:
                 max_survival_rate = survival_rate[i]
                 max_survival_rate_state = i

         print(f'State with maximum Survival Rate (per month) = "{max_survival_rate_stat
         e}"\nSurvival Rate (per month) = {max_survival_rate}')

         State with maximum Survival Rate (per month) = "Dadra and Nagar Haveli and Daman
         and Diu"
         Survival Rate (per month) = 0.1669947620154617
```

In [47]:
```
state_dct_df['Dadra and Nagar Haveli and Daman and Diu']
```

Out[47]:

	State	Cured	Deaths	Confirmed	Month
0	Dadra and Nagar Haveli and Daman and Diu	2	0	30	6
1	Dadra and Nagar Haveli and Daman and Diu	0	0	0	6
2	Dadra and Nagar Haveli and Daman and Diu	0	0	5	6
3	Dadra and Nagar Haveli and Daman and Diu	0	0	1	6
4	Dadra and Nagar Haveli and Daman and Diu	3	0	0	6
...	...	...	...	...	...
176	Dadra and Nagar Haveli and Daman and Diu	2	0	0	12
177	Dadra and Nagar Haveli and Daman and Diu	0	0	1	12
178	Dadra and Nagar Haveli and Daman and Diu	0	0	1	12
179	Dadra and Nagar Haveli and Daman and Diu	2	0	4	12
180	Dadra and Nagar Haveli and Daman and Diu	2	0	5	12

181 rows × 5 columns

Death Rate = Total Deaths/(Total Confirmed*Total number of months for which data is available)

Q3.Check for state in which death rate is more than 1%

In [48]:
```
death_rate ={}
for s in states:
    total_confirmed = 0
    total_deaths = 0
    for i in range(row_count_dct_or_day_count_dct[s]):
       total_confirmed += state_dct_df[s].iloc[i,3]
       total_deaths += state_dct_df[s].iloc[i,2]
    factor = (row_count_dct_or_day_count_dct[s]/365)*12
    death_rate[s] = (total_deaths/(total_confirmed*factor))
```

In [49]:
```
death_rate
```

Out[49]:
```
{'Kerala': 0.0003702495636678569,
 'Telangana': 0.0005778860554956558,
 'Delhi': 0.001757327369582662,
 'Rajasthan': 0.0009402747248058188,
 'Uttar Pradesh': 0.0015505278786667354,
 'Haryana': 0.0011514284341568246,
 'Ladakh': 0.0014882730898568024,
 'Tamil Nadu': 0.0016315512099369584,
 'Karnataka': 0.0014623729793392742,
 'Maharashtra': 0.0028554139497938675,
 'Punjab': 0.003477122618835203,
 'Jammu and Kashmir': 0.0017018148158597529,
 'Andhra Pradesh': 0.0008988989001339504,
 'Uttarakhand': 0.0018604667306562237,
 'Odisha': 0.0006266364786400396,
 'Puducherry': 0.00188481273941467,
 'West Bengal': 0.00198535936688146,
 'Chhattisgarh': 0.0013852831691562153,
 'Chandigarh': 0.0018557551288166136,
 'Gujarat': 0.0021298428529067186,
 'Himachal Pradesh': 0.0018778084627579226,
 'Madhya Pradesh': 0.001780429001243812,
 'Bihar': 0.0006282270015243421,
 'Manipur': 0.0013730722194256375,
 'Mizoram': 0.00017649561904025068,
 'Andaman and Nicobar Islands': 0.001499324176176416,
 'Goa': 0.0016823264754193537,
 'Assam': 0.0005600591862538943,
 'Jharkhand': 0.0010778536095106203,
 'Arunachal Pradesh': 0.00040548802336839095,
 'Tripura': 0.0013942285930687653,
 'Nagaland': 0.0008618171750806525,
 'Meghalaya': 0.0012459150326797385,
 'Dadar Nagar Haveli': 0.0,
 'Sikkim': 0.003412032598274209,
 'Dadra and Nagar Haveli and Daman and Diu': 0.00010029715436364066}
```

In [50]:
```
states_with_death_rate_more_than_one_percent = []
for i in states:
    if death_rate[i]>0.01:
        states_with_death_rate_more_than_one_percent.append(i)
```

In [51]:
```
print('States with death rate of more than 1%  (per month) :\n')
if(not bool(states_with_death_rate_more_than_one_percent)):    # checking if di
ctonary is empty
    print("No states have death rate of more than 1%")
else:
    for i in states_with_death_rate_more_than_one_percent:
        print(i)
```

```
States with death rate of more than 1%  (per month) :

No states have death rate of more than 1%
```

Q1. Filter the month in which heighest people are get infected to Covid-19 virus?

In [52]:
```
print(f"Peak Month = {peak_month}\nPeak Cases = {peak_val}")
```

```
Peak Month = 9
Peak Cases = 2604518
```

Q2. Obtain state in which survival rate is high.

In [53]:
```
print(f'State with maximum Survival Rate (per month) = "{max_survival_rate_stat
e}"\nSurvival Rate (per month) = {max_survival_rate}')
```

```
State with maximum Survival Rate (per month) = "Dadra and Nagar Haveli and Daman
and Diu"
Survival Rate (per month) = 0.1669947620154617
```

Q3. Check for state in which death rate is more than 1%

In [54]:
```
if(not bool(states_with_death_rate_more_than_one_percent)):     # checking if di
ctonary is empty
    print("No state has death rate of more than 1% (per month).")
else:
    print('States with death rate of more than 1% (per month):\n')
    for i in states_with_death_rate_more_than_one_percent:
        print(i)
```

```
No state has death rate of more than 1% (per month).
```