a) List two primary differences between Hadoop version-2 and Hadoop version-3

Solution:

Any two difference among these is correct-
Major difference :

In Hadoop 2 Fault tolerance is ensured using **Replication** whereas in Hadoop-3 it is ensured using **Erasure Coding**.
(Due to replication, overhead is 200 percent in Hadoop-2 and
due to erasure coding, Hadoop-3 overhead is 50% only).

Hadoop 2 supports one standby Name-Node whereas Hadoop-3 supports **multiple standby Name-Nodes** .

For Haddop-2 , Java-7 is the minimum compatible version whereas for Hadoop-3 it is **Java - 8**.

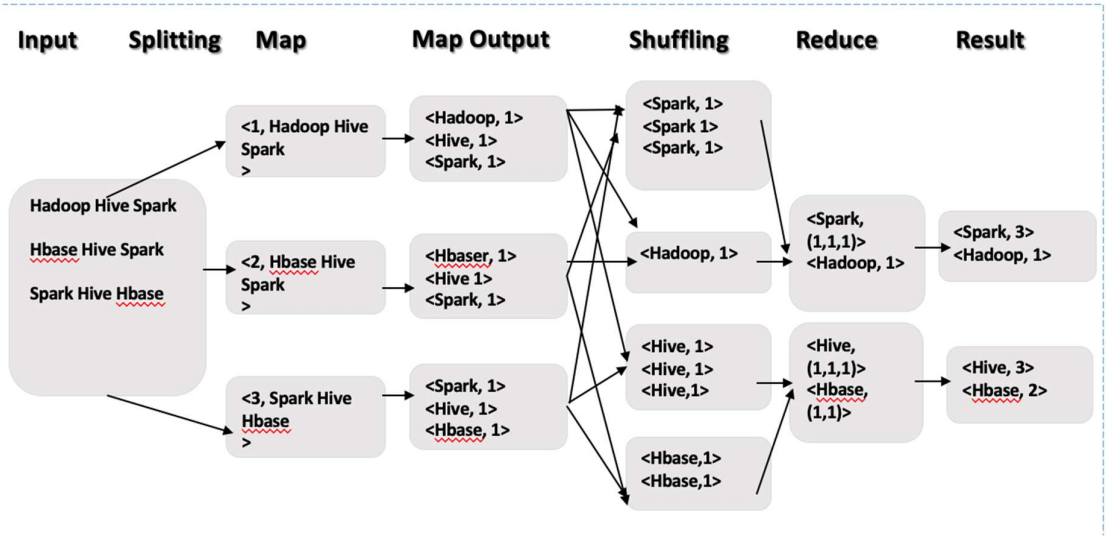
Improved Scalability :Hadoop-2 can scale up to 10 thousands cluster nodes where Hadoop-3 can have more than 10 thousands nodes in the cluster

b) What is Hive metastore? Can NoSQL Database- HBase can be configured as hive metastore?

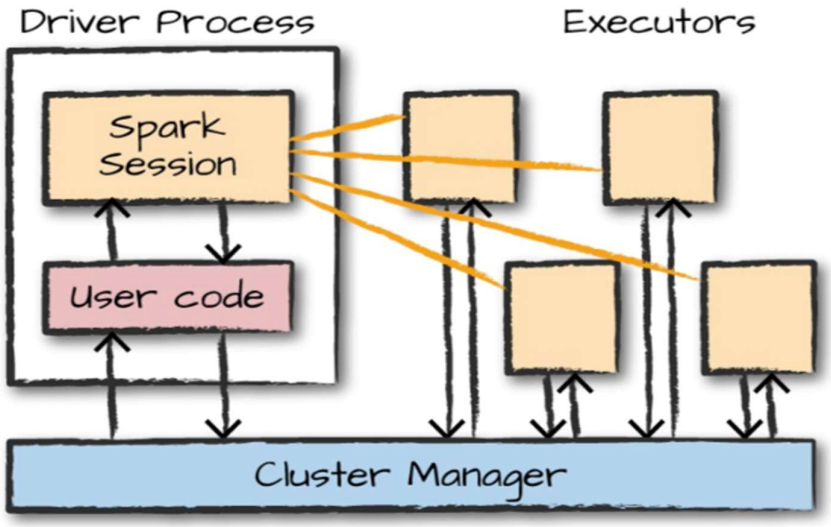
Hive-Metastore is the central repository of Hive Metadata. It is service that mainly **stores the meta data for Hive tables** and relations. For example, Schema and Locations etc.

Yes. HBases can be configured as Hive -Metastore. In fact, HBase is preferred metastore service for highly scalable production systems.

c) Using an example, depict how MapReduce computes word count.



d) Draw and explain various components of Spark architecture.



The components of Apache Spark architecture are essential for its functionality in processing large datasets. Here's a brief overview:

1. **Spark Driver**: The master node of a Spark application. It's responsible for translating the user's program into tasks, scheduling them, and managing their execution with the cluster manager[1].

2. **Executors**: These are worker nodes' processes in charge of running the tasks assigned by the driver and returning the results to the driver. They also provide in-memory storage for RDDs (Resilient Distributed Datasets) that are cached by user programs[1].
3. **Cluster Manager**: An external service for acquiring resources on the cluster (e.g., Standalone Manager, Mesos, YARN, or Kubernetes). It allocates resources to Spark applications based on the requirements[1].
4. **Worker Nodes**: These nodes are the actual working machines of the cluster that run the application code. They host the executors that perform the tasks assigned by the driver[1].

These components work together to optimize the processing of big data computations and utilize datasets and data frames as fundamental data storage mechanisms[1].

---

e) What is cap theorem ? Where does MongoDB stands in cap theorem

The CAP Theorem, also known as Brewer's Theorem, is a fundamental principle in distributed system design that outlines the limitations and trade-offs between three key properties: Consistency, Availability, and Partition Tolerance. Here's a brief explanation of each:

- **Consistency**: Every read operation receives the most recent write or an error[1].
- **Availability**: Every request receives a response, without guaranteeing that it contains the most recent write[1].
- **Partition Tolerance**: The system continues to operate despite an arbitrary number of messages being dropped or delayed by the network between nodes[1].

The theorem posits that it's impossible for a distributed data store to simultaneously provide all three of these guarantees. At most, only two can be achieved at the same time, leading to different system designs prioritizing different properties based on their specific requirements[1]. For example, some systems might prioritize consistency and partition tolerance (CP), sacrificing availability, while others might prioritize availability and partition tolerance (AP), sacrificing consistency.

This theorem is crucial for understanding the trade-offs when designing distributed systems, especially in deciding which two properties are most critical to the system's success and user experience[2].

CAP theorem says, A distributed database system can only have 2 of the below 3 properties:

1. **Consistency** : All nodes/clients see the same data at the same time. *read* operation will return the value of the most recent *write.* entire transaction gets rolled back if there is an error during any stage in the process.
2. **Availability** : Every request gets a response on success/failure. the system remains operational 100% of the time.
3. **Partition Tolerance** : A system that is partition-tolerant can sustain any amount of network failure that doesn't result in a failure of the entire network. Data records are sufficiently replicated across combinations of nodes and networks to keep the system up through intermittent outages.

MongoDB is a CP data store—it resolves network partitions by maintaining consistency, while compromising on availability

Section B

a) Write HDFC shell commands for the following

To Print Version of installed Hadoop

- **hadoop version**

To Copy 'file1.txt' from 'InputDir' to 'OutputDir' as file2.txt

- **hadoop fs -cp InputDir /file1.txt OutputDir/file2.txt**

To Delete an empty directory named as **XYZ.**

- **hadoop fs -rmdir XYZ**

To list the contents of folder named **SampleDir.**

- **hadoop fs -ls SampleDir**

To fetch the usage instructions of **mkdir** command

- **hadoop fs -help mkdir**

b) Write a Spark program psudocode to load a textile named as text.txt as spark RDD and compute its wordcounts.

```
sc = SparkContext("local)

# read data from text file and split each line into words
words = sc.textFile("test.txt".flatMap(lambda line: line.split(" "))
```

```
# count the occurrence of each word
wordCounts = words.map(lambda word: (word, 1).reduceByKey(lambda a,b: a +b)

# save the counts to output
wordCounts.saveAsTextFile("output")
```

c) Two hive tables are shown below. Write a hive query to perform an inner join on the Table1 and Table 2 on 'id' column

```
SELECT t1.Id, t1.NAME, t2.Name
FROM Table1 t1 JOIN Table2 t2
ON (t1.Id = t2.Id);
```

Or

```
SELECT Table1.*, Table1.*
FROM Table1 JOIN Table2 ON (Table1.Id = Table2.id);
```

Output order will be based on the query though should have these records

Joe,2,Tie
Hank,4, Coat

d) Write commands/query in MongoDB to

i. Create a collection named **orders**.
- db.createCollection(' **orders** ')

ii. Insert below record in orders.
{"order_id": 1,
"order_date": '2013-07-25 00:00:00.0',
"order_customer_id": 11599,
"order_status": "CLOSED" }

- **db.orders.insertOne(**
  {
  "order_id" : 1,
  "order_date" : '2013-07-25 00:00:00.0',
  "order_customer_id" : 11599,
  "order_status" : "CLOSED"

}

iii. Fetch orders with **order_status** as COMPLETE.

- db.orders.aggregate(
  [
  {$match: {order_status: "COMPLETE"}}
  ]

iv. Compute count of orders with status COMPLETE and CLOSED.

- db.orders.aggregate(
  [
  {$match : {$or: [{order_status: {$eq: "COMPLETE"}}, {order_status: {$eq: "CLOSED"}}]} },
  {$group : {_id: {"status": "$order_status"}, "count": {$sum: 1}}}
  ]
)