**Section A (20 Marks)**

1. List two primary differences between Hadoop version-2 and Hadoop version-3 – 4 Marks

|  |  |  |
| --- | --- | --- |
| **Feature** | **Hadoop 2** | **Hadoop 3** |
| Resource Management | Introduced YARN for resource management, allowing multiple applications to share cluster resources. | Improved YARN with Opportunistic Containers and Distributed Scheduling for better resource utilization. |
| Storage Efficiency | Data stored with a replication factor (default is 3). | Introduced Erasure Coding for reduced storage overhead, requiring less disk space than replication. |
| Scaling | Supports scaling up to thousands of nodes but has limitations in large-scale deployments. | Improved scalability, with better resource management and distributed scheduling, supporting even larger clusters |
| HDFS (Hadoop Distributed File System) | Single NameNode; provided High Availability (HA) by allowing a standby NameNode. | Supports Multiple NameNodes (Federation), improving scalability and fault tolerance. |
| MapReduce | Traditional MapReduce framework; supports batch processing. | Improved support for different processing models, including enhancements in MapReduce. |
| Minimum Java Version | Java 6 or Java 7. | Requires Java 8 as the minimum version. |

1. What is Hive metastore? Can NoSQL Database -HBase can be configured as hive metastore? (3+1)

The Hive Metastore is a central repository in Apache Hive that stores metadata about the Hive tables, partitions, and databases. It stores only schema, location, partition details etc.

**HBase is not used as a typical Hive Metastore.**

The Hive Metastore is designed to store metadata related to table schema, partitions, and file locations. This metadata is usually structured

**Relational Data Model:** Hive Metastore uses a **relational database model** to store metadata,which works well with SQL-like querying. HBase, being a **NoSQL database.**

Relational databases (like MySQL or PostgreSQL) used as Hive Metastore offer ACID (Atomicity, Consistency, Isolation, Durability) transactional properties, which are essential for managing the metadata

HBase does not support a fixed schema in the same way that relational databases do.

1. Using an example, depict how MapReduce computes word count. – 4 Marks

Refer Day 1 PPT

1. Draw Spark architecture and explain its various components 4 Marks

Refer day 4 PPT

1. What is cap theorem ? Where does MongoDB stands in cap theorem? – 4 Marks

The **CAP Theorem** (also known as **Brewer's Theorem**) is a fundamental concept in distributed systems. It states that a distributed database system can provide only **two out of three** guarantees simultaneously

**Consistency (C)**,

**Availability (A),**

**Partition Tolerance (P)**

MongoDB is a NoSQL database, and it is designed to be **Highly Available and Partition-Tolerant**, meaning it falls under the AP category in the CAP Theorem.

MongoDB in CAP Theorem (AP)

**Section B – 30 Marks**

**Write HDFC shell commands for the following**

i. To print version of installed Hadoop. (1 mark)

ii. To Copy file1.txt from folder InputDir to OutputDir as file2.txt. (2 marks)

iii. To Delete an empty directory named as XYZ. (2 marks)

iv. To list the contents of folder named SampleDir. (2 marks)

v. To fetch the usage instructions/details of mkdir command. (2 marks)

hadoop version

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

hadoop fs -rmdir /XYZ

hadoop fs -ls /SampleDir

hadoop fs -help mkdir

**Write a Spark program pseudo-code to load a text file named as text.txt into spark RDD and compute its wordcounts.**

textRDD = sc.textFile("text.txt")

wordsRDD = textRDD.flatMap(lambda line: line.split(" "))

wordPairsRDD = wordsRDD.map(lambda word: (word, 1))

wordCountsRDD = wordPairsRDD.reduceByKey(lambda a, b: a + b)

wordCounts = wordCountsRDD.collect()

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

db.createCollection("orders")

db.orders.insertOne({ "order\_id": 1, "order\_date": "2013-07-25 00:00:00.0", "order\_customer\_id": 11599, "order\_status": "CLOSED" })

db.orders.find({ "order\_status": "COMPLETE" })

db.orders.aggregate([ { $match: { "order\_status": { $in: ["COMPLETE", "CLOSED"] } } }, { $group: { \_id: "$order\_status", count: { $sum: 1 } } } ])

**Load data set**

file\_path = "/FileStore/tables/housing\_data.csv"

housing\_data = spark.read.csv(file\_path, header=True, inferSchema=True)

housing\_data.show()

**Count the total number of housing-properties listed from 'HSR Layout' location. (2 marks)**

hsr\_count = housing\_data.filter(housing\_data["location"] == "HSR Layout").count() print(f"Total properties in HSR Layout: {hsr\_count}")

**# Register the DataFrame as a SQL table**

housing\_data.createOrReplaceTempView("housing") # SQL query hsr\_count\_sql = spark.sql(""" SELECT COUNT(\*) AS total\_properties FROM housing WHERE location = 'HSR Layout' """) hsr\_count\_sql.show()

**# Count 2 BHK properties in 'Whitefield'**

whitefield\_2bhk\_count = housing\_data.filter( (housing\_data["location"] == "Whitefield") & (housing\_data["size"] == "2 BHK") ).count()

print(f"Total 2 BHK properties in Whitefield: {whitefield\_2bhk\_count}")

**# Average price of 2 BHK in 'HSR Layout'**

hsr\_2bhk\_avg\_price = housing\_data.filter( (housing\_data["location"] == "HSR Layout") & (housing\_data["size"] == "2 BHK") ).groupBy().avg("price").collect()[0][0]

print(f"Average price of 2 BHK in HSR Layout: {hsr\_2bhk\_avg\_price}")

Using Spark ML execute the steps, as questioned below.

1. Remove the features, having more than one third of their entries as missing/null. For the remaining missing values- remove the corresponding row entry from the DataFrame. (3 marks)

**Import required libraries**

from pyspark.sql.functions import col, isnan, when, count

from pyspark.sql import functions as F

**# Calculate the total number of rows**

total\_rows = df.count()

**# Calculate the percentage of missing values in each column**

missing\_percentage = df.select([ (count(when(col(c).isNull() | isnan(col(c)), c)) / total\_rows).alias(c) for c in df.columns ])

**# Show missing percentages**

missing\_percentage.show()

**# Get the list of columns to keep**

columns\_to\_keep = [ c for c in df.columns

if df.select((count(when(col(c).isNull() | isnan(col(c)), c)) / total\_rows).alias(c)).first()[0] <= 1/3 ]

**# Select only the columns to keep**

filtered\_df = df.select(columns\_to\_keep)

Convert all string columns into numeric values using StringIndexer transformer and make sure now DataFrame does not have any string columns anymore. (5 marks)

from pyspark.ml.feature

import StringIndexer from pyspark.ml

import Pipeline from pyspark.sql.types import StringType

**# Step 1: Identify string columns**

string\_columns = [col\_name for col\_name, dtype in df.dtypes if dtype == "string"] print(f"String columns to index: {string\_columns}")

**# Step 2: Create a StringIndexer for each string column**

indexers = [ StringIndexer(inputCol=col\_name, outputCol=f"{col\_name}\_indexed").setHandleInvalid("skip") for col\_name in string\_columns ]

**# Step 3: Create a Pipeline with the StringIndexers**

pipeline = Pipeline(stages=indexers)

**# Step 4: Fit the pipeline and transform the DataFrame**

indexed\_df = pipeline.fit(df).transform(df)

**# Step 5: Drop the original string columns**

final\_df = indexed\_df.drop(\*string\_columns)

**Using vectorAssembler combines all columns (except target column i.e., 'price') of spark DataFrame into single column (name as features). Make sure DataFrame now contains only two columns features and price. (5 marks)**

from pyspark.ml.feature import VectorAssembler

target\_column = "price"

feature\_columns = [col for col in df.columns if col != target\_column]

vector\_assembler = VectorAssembler(inputCols=feature\_columns, outputCol="features")

assembled\_df = vector\_assembler.transform(df) – Trnasform DataFrame

final\_df = assembled\_df.select("features", target\_column) – select only features and price

**Split the vectorized dataframe into training and test sets with one fourth records being held for testing (2 marks)**

train\_df, test\_df = final\_df.randomSplit([0.75, 0.25], seed=42)

print(f"Training set count: {train\_df.count()}")

print(f"Test set count: {test\_df.count()}")

Train default LinearRegression model with features as 'featuresCol' and ‘price’ as label. (5 marks)

from pyspark.ml.regression import LinearRegression

lr = LinearRegression(featuresCol="features", labelCol="price") – initialize model

lr\_model = lr.fit(train\_df) – Train the model

lr\_summary = lr\_model.summary – summary of train model

predictions = lr\_model.transform(test\_df) -- trained model to make predictions on the test set

predictions.select("features", "price", "prediction").show(truncate=False) -- Show predictions