BIG DATA COMPUTING LABORATORY

A19PC2CS49

LABORATORY RECORD

B.TECH

(III YEAR – II SEM)

(2023-2024)

DEPARTMENT OF CSE - (CyS, DS) and AI&DS

(ARTIFICIAL INTELLIGENCE AND DATA SCIENCE)

VALLURUPALLI NAGESWARA RAO VIGNANA JYOTHI INSTITUTE OF ENGINEERING AND TECHNOLOGY

VALLURUPALLI NAGESWARA RAO VIGNANA JYOTHI INSTITUTE OF ENGINEERING AND TECHNOLOGY

TAMASOMA JYOTIRGAMAYA
Estd. 1995

CERTIFICATE

Certified that this is the Bonafide record of the practical work done during the Academic Year 2023-24 by the Student Name G. Vinay Kalyan bearing Hall Ticket Number 21071A7219 Class III-II BTECH - AI&DS in the BIG DATA COMPUTING LABORATORY (A19PC2CS49) Department of CSE - (CyS, DS) and AI&DS.

Dr. D. MANJU
Signature of
Staff Member

Dr. M. RAJA SEKAR
Signature of HOD

Signature of
External Examiner

Scan for
Online Manual

INDEX

INDEX

EXPERIMENT NO: Week 1	DATE:

Hadoop Storage File system (HDFS) (Storage)

Hadoop Distributed File System (HDFS) is a cornerstone of the Apache Hadoop ecosystem, purpose-built to handle vast amounts of data efficiently and reliably. It is designed to scale horizontally, allowing the system to expand by adding more nodes to manage growing data volumes and computational demands. One of the key features of HDFS is its fault tolerance; it achieves this by distributing data across multiple nodes and replicating it to ensure that data remains accessible even if one or more nodes fail. This distributed architecture is optimized for high throughput, making HDFS particularly well-suited for applications that involve large-scale batch processing, such as big data analytics, where the system's ability to process large datasets in parallel is a significant advantage.

The architecture of HDFS is composed of several critical components that work together to manage and store data efficiently. At the core is the NameNode, which functions as the master server, managing the filesystem namespace and regulating client access to files. It keeps a comprehensive directory tree of all files and directories and records the locations of data blocks across the DataNodes. DataNodes, on the other hand, are the workhorses that store the actual data. They handle read and write requests from clients and perform block creation, deletion, and replication under the NameNode's direction. Additionally, the Secondary NameNode plays a vital role in maintaining system efficiency by periodically merging the namespace image with the edit logs, thus preventing the logs from becoming excessively large and ensuring the NameNode can restart quickly after a failure. In HDFS, files are split into large blocks (typically 128MB or 256MB), and each block is replicated across several DataNodes to guarantee data reliability and availability.

HDFS's robust features make it suitable for a variety of use cases, particularly those involving large datasets. It is commonly used as the storage layer for big data analytics platforms like Apache Hadoop and Apache Spark, where its ability to handle vast amounts of data efficiently is crucial. HDFS is also effective in data warehousing environments, capable of storing and managing both structured and unstructured data. In the realm of machine learning, HDFS provides the storage necessary for large datasets used in training models. Moreover, it is ideal for storing substantial volumes of log and event data generated by various applications and systems. Overall, HDFS stands out as a highly scalable and fault-tolerant storage solution, essential for big data applications requiring high throughput. Mastery of HDFS commands, such as hdfs dfs -mkdir for creating directories and hdfs dfs -put for moving files, is fundamental for efficiently managing data within this powerful file system.

A. Hadoop Storage File system

I. Write a command to create the directory structure in HDFS.

hdfs dfs -mkdir -p /user/yourusername/directory/subdirectory

Description:

- hdfs dfs : The command to interact with HDFS.
- -mkdir : The option to create directories.
- -p : This option ensures that parent directories are created if they do not exist.
- /user/yourusername/directory/subdirectory : The path of the directory structure to be created in HDFS.

Example :

hdfs dfs -mkdir -p /user/alice/data/input

This command creates the /user/alice/data/input directory structure in HDFS.

II. Write a Command to move file from local unix/linux machine to HDFS.

hdfs dfs -put /local/path/to/yourfile.txt /hdfs/path/to/destination/

Description:

- hdfs dfs : The command to interact with HDFS.
- -put : The option to copy files from the local filesystem to HDFS.
- /local/path/to/yourfile.txt : The path to the local file you want to copy to HDFS.
- /hdfs/path/to/destination/ : The HDFS directory where you want to place the file.

Example :

hdfs dfs -put /home/alice/documents/sample.txt /user/alice/data/input/

This command copies the sample.txt file from the local directory /home/alice/documents/ to the HDFS directory /user/alice/data/input/.

B. Viewing Data Contents, Files and Directory

a) Write HDFS command to see the contents of files in HDFS.

hdfs dfs -cat /path/to/hdfs/file

Description:

- hdfs dfs: The command used to interact with the HDFS.
- -cat: The option to display the contents of a file.
- /path/to/hdfs/file: The path to the file in HDFS whose contents you want to view.

Example :

hdfs dfs -cat /user/alice/data/input/sample.txt

This command will display the contents of the file sample.txt located at /user/alice/data/input/ in HDFS.

Output - Hello, this is a sample file.

B. Viewing Data Contents, Files and Directory

b) Write HDFS command to see contents of files which are present in HDFS.

hdfs dfs -ls /path/to/hdfs/directory

Description:

- hdfs dfs: The command used to interact with the HDFS.
- -ls: The option to list the files and directories within a specified directory.
- /path/to/hdfs/directory: The path to the directory in HDFS where you want to list the contents.

Example :

hdfs dfs -ls /user/alice/data/input

This command lists all files and directories within the /user/alice/data/input directory in HDFS, showing their permissions, ownership, size, and modification date.

Output -

Found 2 items

-rw-r--r-- 1 alice supergroup 1234 2024-06-23 12:34 /user/alice/data/input/sample.txt

drwxr-xr-x - alice supergroup 0 2024-06-23 12:34 /user/alice/data/input/subdir

C. Getting Files data from the HDFS to Local Disk

I. Write a HDFS command to copy the file from HDFS to local file system.

hdfs dfs -get /path/to/hdfs/file /local/path/to/destination/

Description:

- hdfs dfs: The command to interact with the HDFS.
- -get: The option to copy files from HDFS to the local filesystem.
- /path/to/hdfs/file: The path to the file in HDFS that you want to copy.
- /local/path/to/destination/: The local directory where you want to place the copied file.

Example :

hdfs dfs -get /user/alice/data/input/sample.txt /home/alice/documents/

This command copies the file sample.txt from the HDFS directory /user/alice/data/input/ to the local directory /home/alice/documents/.

EXPERIMENT NO: Week 2	DATE:

Map Reduce Programming (Processing data)

MapReduce is a programming model and processing technique developed by Google for distributed computing on large data sets across a cluster of computers. The core idea behind MapReduce is to divide a task into smaller sub-tasks (mapping) and then combine the results of these sub-tasks (reducing) to produce the final output. This model is particularly well-suited for processing vast amounts of data because it allows tasks to be distributed across many nodes, enabling parallel processing and thus significantly speeding up data analysis and computation tasks.

In the MapReduce model, there are two primary phases: the Map phase and the Reduce phase. During the Map phase, input data is split into independent chunks, which are then processed by the Map function to produce a set of intermediate key-value pairs. These intermediate pairs are then shuffled and sorted by the framework, which groups all values associated with the same key together. The Reduce phase takes over from here, processing each group of key-value pairs to generate the final output. This two-step process simplifies the complexity of data processing, allowing developers to focus on writing the Map and Reduce functions without worrying about the underlying details of data distribution and fault tolerance.

MapReduce has several advantages, making it a popular choice for big data processing. It is highly scalable, capable of handling petabytes of data by leveraging the distributed nature of the framework. Its fault-tolerant design ensures that tasks are automatically retried upon failure, making the system robust and reliable. Additionally, the simplicity of the MapReduce programming model abstracts the intricacies of parallel processing, making it accessible for developers. Common use cases for MapReduce include data analytics, log analysis, search indexing, and machine learning, where it efficiently processes and analyzes large datasets to extract valuable insights.

A. Word Count

i. Develop MapReduce example program in a MapReduce environment to find out the number of occurrences of each word in a text file.

Steps :

1. Create a new java project .Give name as WordCount. Then click next
2. Then go to libraries there click on add external jar files then File system->usr->lib->hadoop, copy all the jar files and click ok.
3. Again click on add external jar files, goto File system->usr->lib->hadoop->client Select all the files and then click ok, and then click finish.
4. Now click on wordCount->src create a new class WordCount.java.
5. From apache documentation of hadoop goto MapReduce tutorials copy the java code

Java Code :

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {
  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);}}}
```

```
public static class IntSumReducer
extends Reducer<Text,IntWritable,Text,IntWritable> {
private IntWritable result = new IntWritable();
public void reduce(Text key, Iterable<IntWritable> values,
Context context
) throws IOException, InterruptedException {
int sum = 0;
for (IntWritable val : values) {
sum += val.get();}
result.set(sum);
context.write(key, result);
}}
public static void main(String[] args) throws Exception {
Configuration conf = new Configuration();
Job job = Job.getInstance(conf, "word count");
job.setJarByClass(WordCount.class);
job.setMapperClass(TokenizerMapper.class);
job.setCombinerClass(IntSumReducer.class);
job.setReducerClass(IntSumReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1); }}
```

Steps :

6. Save the java code and run it.

7. Right click on WordCount folder->export->select jar file->click on next.

8. Name the jar file and select the required location the jar file to be exported - >ok->finish.

9. Go to the location where we exported the jar file and verify it.

Terminal :

i/p - pwd

o/p - /home/cloudera

i/p - ls

o/p -

cloudera-manager	Documents	kerberos	parcels	Templates	workspace
cm_api.py	Downloads	lib	Pictures	Videos	
Desktop	eclipse	Music	Public	Wordcound.jar	

Terminal :

i/p - hdfs dfs mkdir /input

i**/p -** hadoop dfs -put /home/cloudera/Desktop/processfile.txt /input/processfile.txt

i/p - hdfs dfs -cat /input/processfile1.txt

hello sravanthi keerthana hi hi

hi sravanthi hello

i/p - hadoop jar /home/cloudera/Wordcount.jar WordCount /input/processfile1.txt /out

i/p - hdfs dfs -ls /out

o/p -

Found 2 items

-rw-r--r-- 1 cloudera supergroup 0 2023-04-12 22:05 /out/_SUCCESS

-rw-r--r-- 1 cloudera supergroup 37 2023-04-12 22:05 /out/part-r-00000

i/p - hdfs dfs -cat /out/part-r-00000

o/p -

hello 2

hi 3

keerthana 1

sravanthi 2

EXPERIMENT NO: Week 3	DATE:

Data Processing Tool – Hive (NOSQL query based language)

Apache Hive is a data warehouse infrastructure developed on top of Hadoop that enables users to manage and query large datasets stored in a distributed environment using a SQL-like language known as HiveQL. Hive simplifies the process of data analysis by providing an interface similar to SQL, making it accessible to users familiar with traditional database query languages. This abstraction allows users to write complex queries without needing to delve into the underlying MapReduce code, streamlining the process of data processing and analysis.

One of the key features of Hive is its scalability and flexibility. Leveraging Hadoop's distributed storage and processing capabilities, Hive can efficiently manage and process petabytes of data by distributing the workload across multiple nodes in a cluster. This scalability ensures that as the volume of data grows, Hive can continue to perform efficiently. Additionally, Hive supports a variety of data formats, including text files, sequence files, ORC, Parquet, and AVRO, providing the flexibility to handle diverse types of data seamlessly.

Another significant aspect of Hive is its "schema on read" approach. Unlike traditional databases that enforce a schema at the time of data ingestion ("schema on write"), Hive applies the schema when the data is read during query execution. This method allows for greater flexibility in dealing with unstructured and semi-structured data, as users can define the schema as needed at query time. This feature is particularly useful in big data environments where data formats and structures can vary widely.

The architecture of Hive includes a central repository known as the Metastore, which stores metadata about tables, columns, data types, and the location of the data. The Metastore is critical for query planning and execution, as it provides the necessary metadata information to the Hive query engine. Operations such as creating, altering, and dropping tables are managed through the Metastore, ensuring that the data is efficiently organized and accessible. By combining the powerful querying capabilities of HiveQL with Hadoop's robust storage and processing infrastructure, Hive provides a comprehensive solution for big data management and analysis.

Note - To execute HQL Commands in the terminal first type hive and press enter

i. Create the table in HIVE using hive nosql based query.

CREATE TABLE user_data (data_date STRING, user_id STRING, age INT, state STRING, gender STRING) ROW FORMAT DELIMITED FIELDS TERMINATED BY LINES TERMINATED BY ‘,’ stored as textfile;

Description:

This HiveQL statement sets up the user_data table to handle data stored in a delimited text file format where fields are separated by commas and each record is on a new line. Adjust the FIELDS TERMINATED BY and LINES TERMINATED BY ‘,’clauses based on the actual format of your data files to ensure correct parsing and loading into Hive tables.

ii. Fill the table with sample data by using some sample data bases.

Create a sample data file named user_data.txt with the following content:
"2023-06-01","user1",21,"CA","M"
"2023-06-02","user2",25,"NY","F"
"2023-06-03","user3",30,"TX","M"
"2023-06-04","user4",22,"CA","F"
"2023-06-05","user5",28,"NY","M"

Upload this file to HDFS:
hdfs dfs -put user_data.txt /path/to/hdfs/user_data/

Then, load the data into the Hive table:
LOAD DATA INPATH '/path/to/hdfs/user_data/user_data.txt' INTO TABLE user_data;

iii. Write a program that produces a list of properties with minimum value(min_value), largest value(max_value) and number of unique values. Before you start, execute the prepare step to load the data into HDFS.

SELECT MIN(age) AS min_value, MAX(age) AS max_value, COUNT(DISTINCT User) AS unique_values_count FROM user_property;

o/p -

min_value max_value unique_values_count
21 30 5

iv. Generate a count per state.

SELECT State, COUNT(*) AS user_count_per_state FROM user_property GROUP BY State;

o/p -

State	user_count_per_state
CA	2
NY	2
TX	1

v. Write a program that lists the states and their count from the data input.

Java Code -

```
import java.util.*;

public class StateCount {
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      System.out.println("Enter number of data entries:");
      int numEntries = scanner.nextInt();
      scanner.nextLine();
      String[] data = new String[numEntries];
      System.out.println("Enter data in the format 'Date User Age State Gender':");
      for (int i = 0; i < numEntries; i++) {
         data[i] = scanner.nextLine();}
      scanner.close();
      Map<String, Integer> stateCount = new HashMap<>();
      for (String line : data) {
         String[] parts = line.trim().split("\\s+");
         String state = parts[3];
         if (stateCount.containsKey(state)) {
            stateCount.put(state, stateCount.get(state) + 1);
         } else {
            stateCount.put(state, 1);}}
      for (Map.Entry<String, Integer> entry : stateCount.entrySet()) {
         System.out.println(entry.getKey() + ": " + entry.getValue());}}}
```

i/p -

```
2023-06-01   user1   21   CA    M
2023-06-02   user2   25   NY    F
2023-06-03   user3   30   TX    M
2023-06-04   user4   22   CA    F
2023-06-05   user5   28   NY    M
```

o/p -

```
CA: 2
NY: 2
TX: 1
```

EXPERIMENT NO: Week 4	DATE:

Data Processing Tool – Pig (Latin based scripting lang)

Apache Pig is a data processing tool within the Hadoop ecosystem that offers a high-level scripting language called Pig Latin. This language simplifies the complexities associated with MapReduce programming, providing a more intuitive way to process and analyze large datasets. Pig Latin resembles SQL syntax, making it accessible to developers and data engineers familiar with relational databases. This abstraction allows users to focus more on the logic of data transformations rather than the intricacies of distributed computing frameworks like MapReduce.

One of the key strengths of Apache Pig is its extensive set of built-in operators and functions. These operators include relational operations like JOIN, GROUP, FILTER, FOREACH, and others that facilitate common data processing tasks. Users can chain these operators together to create complex data pipelines, enabling comprehensive transformations and analytics within a single Pig Latin script. Additionally, Pig supports user-defined functions (UDFs) written in Java, Python, and other languages, offering flexibility to extend its capabilities with custom logic tailored to specific use cases.

Integration with various components of the Hadoop ecosystem is another significant feature of Apache Pig. It seamlessly interacts with Hadoop's distributed file system (HDFS), allowing data to be loaded from and stored back into HDFS. Pig also integrates with HBase for NoSQL data processing and supports data ingestion from other storage systems. This interoperability makes it a versatile tool for processing diverse data sources in big data environments.

Apache Pig optimizes data processing jobs through internal optimizations such as query plan optimization and execution planning. It automatically optimizes data flows to minimize data movement and maximize resource utilization across the Hadoop cluster. By executing jobs in a data flow pipeline manner, Pig enhances performance and efficiency, making it suitable for handling large-scale batch processing tasks effectively.

Common use cases for Apache Pig include data preparation for analytics, ETL (Extract, Transform, Load) operations, data cleansing, and exploratory data analysis. It is particularly valuable in scenarios where data requires significant transformation before analysis or where complex data workflows need to be orchestrated efficiently. Overall, Apache Pig simplifies big data processing with its intuitive scripting language and powerful processing capabilities, catering to both developers and data analysts aiming to derive insights from large datasets within the Hadoop ecosystem.

Note - To execute Grunt Commands in the terminal first type pig and press enter

Create a sample data file named input_data.txt with the following content:

"site1",1,10,"2023-04-27"
"site2",1,5,"2023-04-27"
"site1",2,15,"2023-04-27"
"site2",2,20,"2023-04-27"
"site1",1,5,"2023-04-28"
"site2",1,10,"2023-04-28"
"site1",2,25,"2023-04-28"
"site2",2,30,"2023-04-28"

Load Data and Define Schema

logs = LOAD 'input_data.txt' USING PigStorage(',') AS (site: chararray, hour_of_day: int, page_views: int, data_date: chararray);

i. Calculate the total views per hour per day.

views_per_hour_per_day = FOREACH (GROUP logs BY (data_date, hour_of_day)) { GENERATE FLATTEN(group) AS (data_date, hour_of_day), SUM(logs.page_views) AS total_views; };

ii. Calculate the total views per day.

views_per_day = FOREACH (GROUP logs BY data_date) { GENERATE group AS data_date, SUM(logs.page_views) AS total_views; };

iii. Calculate the total counts of each hour across all days.

counts_per_hour = FOREACH (GROUP logs BY hour_of_day) { GENERATE group AS hour_of_day, COUNT(logs) AS count_per_hour; };

Storing all the values in files :

Storing total views per hour per day -

STORE views_per_hour_per_day INTO 'output/views_per_hour_per_day' USING PigStorage(',');

Values Stored -

2023-04-27,1,15
2023-04-27,2,35
2023-04-28,1,15
2023-04-28,2,55

Storing total views per day -

STORE views_per_day INTO 'output/views_per_day' USING PigStorage(',');

Values Stored -

2023-04-27,50
2023-04-28,85

Storing counts for each hour in all days -

STORE counts_per_hour INTO 'output/counts_per_hour' USING PigStorage(',');

Values Stored -

2023-04-27,1,15

2023-04-27,2,35

2023-04-28,1,15

2023-04-28,2,55

iv. We can write word count script by passing text file as input

Create a sample data file named input_data.txt with the following content:

hello world

hello pig world

pig pig

```
data = LOAD 'input_text.txt' AS (line: chararray);
words = FOREACH data GENERATE FLATTEN(TOKENIZE(line)) AS word;
word_counts = FOREACH (GROUP words BY word) {
   GENERATE group AS word, COUNT(words) AS count;
};
STORE word_counts INTO 'output/word_count' USING PigStorage(',');
```

Values Stored -

hello,2

world,2

pig,3

EXPERIMENT NO: Week 5	DATE:

SQOOP

Apache Sqoop is a robust and widely used tool designed for efficiently transferring bulk data between Apache Hadoop and structured data stores such as relational databases. It addresses the need to move data in and out of Hadoop in an automated and streamlined manner, facilitating seamless integration of Hadoop with existing data processing workflows.

At its core, Sqoop provides a command-line interface that allows users to specify various parameters to initiate data transfer operations. These operations typically involve importing data from a relational database into Hadoop (HDFS) or exporting data from Hadoop back into a relational database. Sqoop supports a variety of relational databases including MySQL, PostgreSQL, Oracle, SQL Server, and others, ensuring flexibility in data integration across different systems.

One of Sqoop's primary strengths lies in its ability to parallelize data transfer operations, thereby optimizing performance and reducing the time taken to move large datasets. By breaking down the data transfer process into parallel tasks, Sqoop leverages the scalability of Hadoop clusters to handle substantial volumes of data efficiently. This parallelization is achieved by splitting data into chunks based on specified criteria (e.g., primary key ranges or number of mappers), allowing multiple mappers to work concurrently to import or export data.

Moreover, Sqoop provides mechanisms for data serialization and deserialization, ensuring that data is transferred in a format compatible with Hadoop's storage and processing frameworks. It supports various file formats such as text files, Avro, SequenceFiles, and others, enabling users to choose the format that best suits their data processing needs within the Hadoop ecosystem.

Another key feature of Sqoop is its support for incremental data imports, which allows users to efficiently import only new or updated data from a relational database into Hadoop. This incremental import capability is crucial for maintaining consistency and minimizing data transfer overhead, particularly in scenarios where datasets are regularly updated or appended.

In summary, Apache Sqoop plays a vital role in facilitating seamless data integration between Hadoop and relational databases. Its command-line interface, support for parallel data transfer, compatibility with various databases and file formats, and incremental import capabilities make it a versatile tool for handling large-scale data movement tasks in enterprise environments leveraging Apache Hadoop.

I. Create table in HIVE using hive query language.

CREATE TABLE users (id INT, name STRING);

INSERT INTO TABLE users VALUES (1, 'Alice'), (2, 'Bob'), (3, 'Charlie');

Description:

- CREATE TABLE: This HQL command initializes the users table in Hive with two columns: id of type INT and name of type STRING.
- INSERT INTO TABLE: This command inserts sample rows directly into the users table in Hive.
- Hive will store these rows as part of its managed table storage in HDFS.

Note - New Table must have primary key

II. Import the sql table data into hive using sqoop too.

sqoop import --connect jdbc:mysql://localhost/retail_db --username root --password cloudera --table categories --hive-import --hive-table demo30;

Description:

- sqoop import: Initiates Sqoop to perform data import operations.
- --connect: Specifies the connection string to the MySQL database where mysql_host is the hostname or IP address of your MySQL server, database_name is the name of the database you're connecting to.
- --username and --password: Provide the credentials (mysql_username and mysql_password) to authenticate and access the MySQL database.
- --table: Specifies the name of the table in MySQL (sql_users) from which data will be imported.
- --hive-import: Indicates that data should be imported into Hive.
- --hive-table: Specifies the name of the Hive table (users) where data will be imported.
- --create-hive-table: Instructs Sqoop to create the Hive table (users) if it doesn't already exist. This is useful if you haven't created the table manually in Hive.
- --fields-terminated-by '\t': Specifies the delimiter used in the input data (\t for tab-delimited data). Ensure this matches the actual delimiter used in your SQL table.

III. Export hive table data into local machine and into SQL.

```
sqoop export \
--connect jdbc:mysql://mysql_host:3306/database_name \
--username mysql_username \
--password mysql_password \
--table sql_users_exported \
--export-dir /user/hive/warehouse/users \
--input-fields-terminated-by '\t' \
--input-null-string '\\N' \
```

Description:

- sqoop export: Initiates Sqoop to perform data export operations.
- --connect: Specifies the connection string to the MySQL database where mysql_host is the hostname or IP address of your MySQL server, database_name is the name of the database you're connecting to.
- --username and --password: Provide the credentials (mysql_username and mysql_password) to authenticate and access the MySQL database.
- --table: Specifies the name of the table in MySQL (sql_users_exported) where data will be exported.
- --export-dir: Specifies the HDFS directory path (/user/hive/warehouse/users) where the Hive table data (users) is stored. Sqoop will read data from this location and export it to MySQL.
- --input-fields-terminated-by '\t': Specifies the delimiter used in the input data (\t for tab-delimited data). Ensure this matches the actual delimiter used in your Hive table.
- --input-null-string '\\N' and --input-null-non-string '\\N': Specifies how null values are represented in the input data. Sqoop will replace these values appropriately when exporting to MySQL.

EXPERIMENT NO: Week 5	DATE:

Exploring ARIMA model

The Autoregressive Integrated Moving Average (ARIMA) model is a widely used method for time series forecasting, offering a structured approach to capturing and predicting patterns in sequential data. It combines the concepts of autoregression (AR), differencing (I), and moving average (MA) to handle different types of time series behaviors such as trends, seasonality, and autocorrelation.

Components of ARIMA Model:

1. Autoregressive (AR) Component:
The AR component of ARIMA reflects the dependency between an observation and a number of lagged observations of the series itself. It models the linear relationship where the current value of the series depends on its own previous values. This component is particularly useful for capturing trends and cyclic patterns in the data. By specifying the order of AR (denoted as p), analysts determine how many lagged terms to include in the model, thereby controlling the influence of past observations on the current prediction.

2. Integrated (I) Component:
The I component involves differencing the raw observations to make the series stationary, meaning its statistical properties remain constant over time. Stationarity is crucial in time series analysis because it simplifies the modeling process and enhances the reliability of forecasts. The order of differencing (denoted as d) indicates how many times differencing is applied to achieve stationarity. This step is essential when the data exhibits trends or seasonal patterns that need to be removed for accurate modeling.
Differencing involves subtracting the previous observation from the current observation. It is denoted as ddd in ARIMA(p, d, q), where ddd is the order of differencing needed to make the series stationary.

3. Moving Average (MA) Component:
The MA component of ARIMA accounts for the dependency between an observation and a residual error from a moving average model applied to lagged observations. It captures short-term fluctuations or random shocks in the series that are not explained by the autoregressive component. Similar to AR, the order of MA (denoted as q) specifies the number of lagged forecast errors included in the model, helping to smooth out irregularities and noise in the data.

ARIMA Model Order: ARIMA(p,d,q)

- p: The number of lag observations included in the model (AR order).
- d: The number of times that the raw observations are differenced (I order).
- q: The size of the moving average window (MA order).

Steps in Exploring ARIMA Models:

1. Data Exploration and Preprocessing:

- Data Understanding: Understand the nature of the time series data, its trends, seasonality, and any patterns.
- Stationarity Check: Ensure the time series is stationary through statistical tests (like ADF test) or visual inspection (plotting rolling statistics).
- Transformation: Apply transformations such as differencing to achieve stationarity if necessary.

2. Model Identification:

- ACF and PACF Plots: Use Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) plots to identify potential values of p (AR order) and q (MA order).
- Parameter Selection: Choose appropriate values of p, d, and q based on the ACF and PACF plots, and the stationarity achieved.

3. Model Estimation and Diagnostic Checking:

- Fit ARIMA Model: Use the chosen parameters to fit the ARIMA model to the data.
- Diagnostic Checks: Evaluate the model residuals using statistical tests and visual checks (e.g., histogram of residuals, Q-Q plot) to ensure they are normally distributed and independent.

4. Forecasting and Model Evaluation:

- Forecasting: Generate forecasts using the fitted ARIMA model.
- Model Evaluation: Evaluate forecast accuracy using metrics such as Mean Absolute Error (MAE), Mean Squared Error (MSE), or others suitable for your application.

5. Model Refinement and Validation:

- Parameter Tuning: Refine model parameters if necessary based on model performance metrics.
- Cross-validation: Validate the model on out-of-sample data to assess its generalizability and robustness.

Applications and Benefits:

ARIMA models find extensive applications in forecasting stock prices, predicting economic indicators, analyzing weather patterns, and planning resource allocation. Their ability to handle both short-term fluctuations and long-term trends makes them valuable tools for businesses, researchers, and policymakers seeking to make data-driven decisions.

Practical Example:

Consider a scenario where historical sales data is analyzed using ARIMA to forecast future sales trends. The steps involve preprocessing the data to ensure stationarity, identifying appropriate ARIMA parameters through ACF and PACF analysis, fitting the ARIMA model, generating forecasts, and evaluating the forecast accuracy.

Python Code for ARIMA Modelling:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.stattools import adfuller

data = { 'date': pd.date_range('2023-01-01', periods=100), 'value': np.random.randn(100).cumsum() }
df = pd.DataFrame(data)
df['value'] = df['value'] + 100
result = adfuller(df['value'])
print(f'ADF Statistic: {result[0]}')
print(f'p-value: {result[1]}')
plt.figure(figsize=(12, 6))
plt.subplot(211)
plot_acf(df['value'], lags=20, title='Autocorrelation Function (ACF)')
plt.subplot(212)
plot_pacf(df['value'], lags=20, title='Partial Autocorrelation Function (PACF)')
plt.tight_layout()
plt.show()
model = ARIMA(df['value'], order=(1, 1, 1))  # Example order (p, d, q)
results = model.fit()
print(results.summary())
forecast = results.get_forecast(steps=10)  # Example forecast 10 steps ahead
forecast_mean = forecast.predicted_mean
plt.figure(figsize=(12, 6))
plt.plot(df.index, df['value'], label='Actual')
plt.plot(forecast_mean.index, forecast_mean.values, label='Forecast')
plt.fill_between(forecast_mean.index, forecast.conf_int()[:, 0], forecast.conf_int()[:, 1], color='gray',
alpha=0.2)
plt.legend()
plt.title('ARIMA Forecast')
plt.xlabel('Time')
plt.ylabel('Value')
plt.show()
```

o/p -

```
ADF Statistic: -1.1839750714409927
p-value: 0.6804509660806153
```

Output Figures -