INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

Dundigal - 500 043, Hyderabad, Telangana

COURSE CONTENT

BIG DATA MANAGEMENT LABORATORY								
VII Semester: CSE (DS)								
Course Code	**Category**	**Hours / Week**			**Credits**	**Maximum Marks**		
		L	**T**	**P**	**C**	**CIA**	**SEE**	**Total**
ACDD29	**Core**	3	0	0	3	40	60	100
Contact Classes: Nil	**Tutorial Classes: Nil**	**Practical Classes: 45**				**Total Classes: 45**		
Prerequisite: Basic knowledge of computer networks, operating systems, and parallel/distributed computing.								

I. COURSE OVERVIEW:

The Big Data Management Laboratory provides practical exposure to the tools and technologies used for handling and analyzing large-scale datasets. The course focuses on configuring big data environments using Hadoop and its ecosystem, performing data operations using Hive, Pig, and HBase, and developing distributed processing applications with MapReduce and Apache Spark. Through hands-on sessions, students will build complete big data workflows involving data ingestion, transformation, analysis, and visualization. This lab equips students with the technical skills required to solve real-world problems using big data tools and frameworks.

II. COURSE OBJECTIVES:

The students will try to learn:

I. The installation and configuration of big data platforms such as Hadoop and its ecosystem tools for managing large-scale data.

II. The techniques for ingesting, storing, and managing structured and unstructured data using Hive, Pig, and HBase.

III. The development of data processing and analysis workflows using MapReduce and Apache Spark frameworks.

IV. The practical use of big data tools through real-time experiments for data querying, transformation, and visualization.

III. COURSE OUTCOMES:

After successful completion of the course, students should be able to:

CO1 Identify the components of the Hadoop ecosystem and configure a big data processing environment.

CO2 Demonstrate data manipulation operations using HDFS, Hive, and Pig for structured and semi structured data.

CO3 Develop data processing applications using MapReduce and Spark for large-scale data analytics.

CO4 Analyze performance aspects of big data queries and transformations using tuning and benchmarking tools.

CO5 Design end-to-end big data workflows for practical problems using real-world datasets.

CO6 Interpret results from data analytics tasks and visualize outputs for informed decision-making.

Experiment 1: Introduction and Hadoop Setup

Aim

To study the architecture of Hadoop and its ecosystem, and to install and configure a single-node Hadoop cluster.

Software / Tools Required

- Ubuntu / Linux OS (or WSL)
- Java JDK 8/11
- Apache Hadoop (stable release)
- SSH

Theory

Hadoop is an open-source framework used for distributed storage and processing of very large data sets on clusters of commodity hardware. It consists of two core components: the Hadoop Distributed File System (HDFS) for storage, and MapReduce/YARN for distributed processing and resource management.

HDFS follows a master-slave architecture with a NameNode that manages the file system namespace and metadata, and multiple DataNodes that store the actual data blocks. YARN (Yet Another Resource Negotiator) manages cluster resources through a ResourceManager and NodeManagers, allowing multiple processing engines (MapReduce, Spark, etc.) to run on the same cluster.

Procedure

1. Update the system and install Java: sudo apt update && sudo apt install openjdk-11-jdk -y
2. Verify the Java installation using java -version.
3. Create a dedicated Hadoop user and configure passwordless SSH for localhost.
4. Download and extract the Apache Hadoop binary distribution.
5. Set the environment variables JAVA_HOME and HADOOP_HOME in ~/.bashrc.
6. Edit the configuration files core-site.xml, hdfs-site.xml, mapred-site.xml and yarn-site.xml.
7. Format the HDFS NameNode using the command hdfs namenode -format.
8. Start the HDFS and YARN daemons and verify all services are running.

Configuration Snippets

core-site.xml:

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

Starting the cluster:

```
hdfs namenode -format
start-dfs.sh
start-yarn.sh
jps
```

Sample Output

```
25412 NameNode
25631 DataNode
25890 SecondaryNameNode
26210 ResourceManager
26355 NodeManager
26489 Jps
```

Try It Yourself

Configure a multi-node cluster (one master and two slave nodes) and verify communication between nodes using the Hadoop web UI (port 9870).

Result

The Hadoop architecture was studied and a single-node Hadoop cluster was successfully installed and configured. All required daemons were verified to be running using the jps command.

Experiment 2: HDFS Data Management

Aim

To perform basic file management operations on the Hadoop Distributed File System (HDFS) using HDFS shell commands.

Software / Tools Required

- Configured Hadoop single-node cluster
- Terminal / Shell

Theory

HDFS stores files by splitting them into large blocks (default 128 MB) that are distributed and replicated (default replication factor 3) across DataNodes for fault tolerance. HDFS shell commands, prefixed with hdfs dfs or hadoop fs, are used to create directories, upload and download files, and manage permissions in a manner similar to standard Unix file commands.

Procedure

9. Start the Hadoop daemons using start-dfs.sh.
10. Create a new directory in HDFS.
11. Upload a local file into the created HDFS directory.
12. List the contents of the directory and view file details.
13. Display the contents of the uploaded file.
14. Download a file from HDFS to the local file system.
15. Copy, move, and delete files/directories within HDFS.
16. Check the disk usage and replication status of the file.

Program / Commands

HDFS commands:

```
hdfs dfs -mkdir /user/student/input
hdfs dfs -put localfile.txt /user/student/input/
hdfs dfs -ls /user/student/input
hdfs dfs -cat /user/student/input/localfile.txt
hdfs dfs -get /user/student/input/localfile.txt ./downloaded.txt
hdfs dfs -cp /user/student/input/localfile.txt /user/student/backup/
hdfs dfs -du -h /user/student/input
hdfs dfs -rm /user/student/input/localfile.txt
```

Sample Output

```
Found 1 items
-rw-r--r--  3 student supergroup  1024 2026-08-01 10:15
/user/student/input/localfile.txt

Deleted /user/student/input/localfile.txt
```

Try It Yourself

Change the replication factor of a file using hdfs dfs -setrep and observe the change on the NameNode web UI.

Result

Basic HDFS file management operations were successfully performed and verified using HDFS shell commands.

Experiment 3: MapReduce Programming – Word Count

Aim

To develop and execute a MapReduce program to count the occurrence of each word in a given text file.

Software / Tools Required

- Hadoop cluster
- Java JDK
- javac / Hadoop classpath

Theory

MapReduce is a programming model for processing large data sets in parallel across a cluster. It consists of a Map phase, which processes input key-value pairs and emits intermediate key-value pairs, and a Reduce phase, which aggregates the intermediate values associated with the same key. The Word Count program is the canonical example: the Mapper emits (word, 1) for every word, and the Reducer sums the counts for each unique word.

Procedure

17. Write the Mapper class that tokenizes each input line and emits (word, 1).
18. Write the Reducer class that sums the values for each word key.
19. Write the Driver class that configures the Job with the Mapper and Reducer.
20. Compile the Java classes and package them into a JAR file.
21. Upload the input text file to HDFS.
22. Run the MapReduce job using the hadoop jar command.
23. View the output stored in the HDFS output directory.

Program / Commands

WordCount.java:

```
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 java.io.IOException, InterruptedException {
```

```
      for (String token : value.toString().split("\\s+")) {
        word.set(token);
        context.write(word, one);
      }
    }
  }
  public static class IntSumReducer
      extends Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws java.io.IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) sum += val.get();
      context.write(key, new IntWritable(sum));
    }
  }
  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);
  }
}
```

Compile, package and run:

```
hadoop com.sun.tools.javac.Main WordCount.java
jar cf wordcount.jar WordCount*.class
hdfs dfs -put input.txt /user/student/input/
hadoop jar wordcount.jar WordCount /user/student/input /user/student/output
hdfs dfs -cat /user/student/output/part-r-00000
```

Sample Output

```
Hadoop 3
MapReduce    2
big    2
data   4
framework    1
is     2
```

Try It Yourself

Modify the program to also compute the average word length and output it alongside the word count.

Result

The MapReduce Word Count program was successfully developed and executed on the Hadoop cluster.

Experiment 4: MapReduce – Advanced Programs

Aim

To develop a MapReduce program to find the maximum temperature recorded for each year from a weather data set, demonstrating the use of a Combiner for optimization.

Software / Tools Required

- Hadoop cluster
- Java JDK

Theory

Beyond simple counting, MapReduce programs can perform aggregation such as finding minimum, maximum, or average values grouped by a key. A Combiner acts as a local reducer that runs on the Mapper's output before it is sent across the network to the Reducer, reducing data shuffle and improving performance. This experiment reads (year, temperature) records and outputs the maximum temperature per year.

Procedure

24. Prepare a sample weather data set with fields: year and temperature.
25. Write the Mapper to emit (year, temperature) pairs.
26. Write the Reducer to compute the maximum temperature for each year.
27. Configure the Combiner class to pre-aggregate results at the Mapper node.
28. Compile, package and run the job on the Hadoop cluster.
29. Verify the output for correctness against the input data set.

Program / Commands

MaxTemperatureMapper.java (excerpt):

```
public class MaxTemperatureMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {
  public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {
    String[] fields = value.toString().split(",");
    String year = fields[0];
    int temperature = Integer.parseInt(fields[1]);
    context.write(new Text(year), new IntWritable(temperature));
  }
}
```

MaxTemperatureReducer.java (excerpt):

```
public class MaxTemperatureReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {
  public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
    int max = Integer.MIN_VALUE;
    for (IntWritable val : values) max = Math.max(max, val.get());
```

```
    context.write(key, new IntWritable(max));
  }
}
```

Run:

```
hadoop jar maxtemp.jar MaxTemperature /user/student/weather
/user/student/maxtemp_out
hdfs dfs -cat /user/student/maxtemp_out/part-r-00000
```

Sample Output

```
2023   41
2024   39
2025   43
```

Try It Yourself

Extend the program to also compute the minimum temperature per year in the same job using a custom output format.

Result

The advanced MapReduce program to find the maximum temperature per year was implemented and verified successfully.

Experiment 5: Hive – Table Creation and Queries

Aim

To install/configure Apache Hive and perform table creation, data loading, and basic HiveQL queries.

Software / Tools Required

- Hadoop cluster
- Apache Hive
- MySQL/Derby metastore

Theory

Apache Hive is a data warehouse system built on top of Hadoop that provides an SQL-like interface, HiveQL, for querying data stored in HDFS. Hive translates queries into MapReduce, Tez, or Spark jobs internally. It stores table structure and schema information in a metastore, and supports both managed and external tables.

Procedure

30. Start the Hive services and open the Hive CLI/Beeline shell.
31. Create a database and switch to it using the USE command.
32. Create a managed table with an appropriate schema.
33. Load data into the table from a local file or HDFS.
34. Run SELECT queries with WHERE, ORDER BY and LIMIT clauses.
35. Verify the results and describe the table structure.

Program / Commands

HiveQL:

```
CREATE DATABASE bigdata_lab;
USE bigdata_lab;

CREATE TABLE employees (
  emp_id INT,
  emp_name STRING,
  department STRING,
  salary DOUBLE
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH '/home/student/employees.csv'
INTO TABLE employees;

SELECT * FROM employees WHERE salary > 50000 ORDER BY salary DESC;
DESCRIBE employees;
```

Sample Output

```
emp_id  emp_name   department   salary
104     Anitha     Data Science 78000.0
101     Ravi       Engineering  65000.0
102     Kavya      Engineering  55000.0
```

Try It Yourself

Create an external table pointing to an existing HDFS directory and compare its behaviour with the managed table when dropped.

Result

Hive tables were created and basic HiveQL queries were executed successfully to retrieve and filter data.

Experiment 6: Hive – Advanced Queries and Functions

Aim

To perform advanced HiveQL operations including joins, partitioning, and built-in functions.

Software / Tools Required

- Hadoop cluster
- Apache Hive

Theory

Hive supports advanced querying features such as partitioning (dividing a table into parts based on column values for faster query performance), bucketing, joins between multiple tables, and a rich set of built-in functions for string, date, and aggregate operations. Partitioning avoids full table scans by allowing Hive to prune irrelevant partitions at query time.

Procedure

36. Create a partitioned table on the department column.
37. Load data into specific partitions.
38. Create a second table (departments) to demonstrate joins.
39. Perform an INNER JOIN between the employees and departments tables.
40. Apply aggregate functions such as SUM, AVG, and COUNT with GROUP BY.
41. Use built-in string and date functions on selected columns.

Program / Commands

HiveQL:

```
CREATE TABLE employees_part (
  emp_id INT, emp_name STRING, salary DOUBLE
)
PARTITIONED BY (department STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';

SET hive.exec.dynamic.partition.mode=nonstrict;
INSERT INTO TABLE employees_part PARTITION (department)
SELECT emp_id, emp_name, salary, department FROM employees;

SELECT e.emp_name, d.dept_location, e.salary
FROM employees e
JOIN departments d ON e.department = d.dept_name;

SELECT department, COUNT(*) AS headcount, AVG(salary) AS avg_salary
FROM employees
GROUP BY department;
```

Sample Output

```
emp_name   dept_location   salary
Ravi       Block-A         65000.0
Anitha     Block-C         78000.0

department     headcount   avg_salary
Engineering    2           60000.0
Data Science   1           78000.0
```

Try It Yourself

Write a Hive query using window functions (RANK() OVER ...) to rank employees by salary within each department.

Result

Advanced HiveQL operations including partitioning, joins and aggregate functions were executed successfully.

Experiment 7: Pig – Data Flow Scripts

Aim

To install Apache Pig and write Pig Latin scripts to load, transform, and analyze data.

Software / Tools Required

- Hadoop cluster
- Apache Pig

Theory

Apache Pig is a high-level platform for processing large data sets using a scripting language called Pig Latin. Pig Latin expresses data analysis as a sequence of data-flow transformations such as LOAD, FILTER, FOREACH...GENERATE, GROUP, and STORE, which Pig converts internally into a series of MapReduce jobs.

Procedure

42. Start Pig in local or MapReduce mode using the pig command.
43. Load the input data set using the LOAD statement with an appropriate schema.
44. Filter records based on a condition using FILTER.
45. Project and transform required fields using FOREACH...GENERATE.
46. Group data by a key field using GROUP and compute aggregates.
47. Store or dump the final result.

Program / Commands

sales_analysis.pig:

```
records = LOAD '/user/student/sales.csv' USING PigStorage(',')
    AS (order_id:int, product:chararray, category:chararray, amount:double);

filtered = FILTER records BY amount > 500;

projected = FOREACH filtered GENERATE product, category, amount;

grouped = GROUP projected BY category;

summary = FOREACH grouped GENERATE group AS category,
    SUM(projected.amount) AS total_sales;

DUMP summary;
STORE summary INTO '/user/student/pig_output' USING PigStorage(',');
```

Sample Output

```
(Electronics,154500.0)
(Groceries,72300.0)
(Furniture,98750.0)
```

Try It Yourself

Modify the script to also compute the average order amount per category using the AVG() function.

Result

A Pig Latin script was developed to load, filter, group and summarize data successfully using Apache Pig.

Experiment 8: Pig – Joins and UDFs

Aim

To perform join operations in Pig Latin and to write and use a custom User Defined Function (UDF).

Software / Tools Required

- Hadoop cluster
- Apache Pig
- Java JDK (for UDF)

Theory

Pig supports joining two or more relations using the JOIN keyword, similar to SQL joins. When built-in functions are insufficient, Pig allows developers to write custom User Defined Functions (UDFs) in Java (or Python/Jython) by extending the EvalFunc class, which can then be registered and used inside a Pig script.

Procedure

48. Load two related data sets: orders and customers.
49. Perform a JOIN between the two relations on the common key.
50. Write a Java UDF that extends EvalFunc<T> to implement custom logic (e.g. discount calculation).
51. Compile the UDF into a JAR and register it in the Pig script using REGISTER.
52. Apply the UDF within a FOREACH...GENERATE statement.
53. Store and verify the final output.

Program / Commands

DiscountUDF.java:

```
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;

public class DiscountUDF extends EvalFunc<Double> {
  public Double exec(Tuple input) throws java.io.IOException {
    if (input == null || input.size() == 0) return null;
    double amount = (Double) input.get(0);
    return amount > 1000 ? amount * 0.9 : amount;
  }
}
```

join_udf.pig:

```
REGISTER '/home/student/discount-udf.jar';

orders = LOAD '/user/student/orders.csv' USING PigStorage(',')
    AS (order_id:int, cust_id:int, amount:double);
customers = LOAD '/user/student/customers.csv' USING PigStorage(',')
    AS (cust_id:int, cust_name:chararray);
```

```
joined = JOIN orders BY cust_id, customers BY cust_id;

result = FOREACH joined GENERATE cust_name,
    DiscountUDF(amount) AS final_amount;

DUMP result;
```

Sample Output

```
(Ravi,1350.0)
(Kavya,499.0)
(Anitha,2160.0)
```

Try It Yourself

Write a second UDF to categorize orders as 'High', 'Medium', or 'Low' value and apply it alongside the discount UDF.

Result

Join operations and a custom UDF were successfully implemented and applied in a Pig Latin script.

Experiment 9: HBase – NoSQL Database Setup and Operations

Aim

To install and configure Apache HBase and perform basic CRUD operations on an HBase table.

Software / Tools Required

- Hadoop cluster
- Apache HBase
- ZooKeeper

Theory

Apache HBase is a distributed, column-oriented NoSQL database built on top of HDFS, modeled after Google's Bigtable. Data in HBase is organized into tables containing rows identified by a row key, with columns grouped into column families. HBase is well-suited for random, real-time read/write access to large data sets.

Procedure

54. Start the HBase daemons (start-hbase.sh) after starting Hadoop.
55. Open the HBase shell using the hbase shell command.
56. Create a table with one or more column families.
57. Insert (put) records into the table.
58. Retrieve a record using get and scan the entire table.
59. Update an existing cell value and delete a record.
60. Disable and drop the table when finished.

Program / Commands

HBase shell commands:

```
create 'student', 'personal', 'academic'
put 'student', '1', 'personal:name', 'Ravi Kumar'
put 'student', '1', 'personal:age', '21'
put 'student', '1', 'academic:dept', 'CSE(DS)'
put 'student', '1', 'academic:cgpa', '8.7'

get 'student', '1'
scan 'student'

put 'student', '1', 'academic:cgpa', '8.9'
delete 'student', '1', 'personal:age'

disable 'student'
drop 'student'
```

Sample Output

```
COLUMN                    CELL
```

```
 academic:cgpa             timestamp=..., value=8.9
 academic:dept             timestamp=..., value=CSE(DS)
 personal:name             timestamp=..., value=Ravi Kumar
1 row(s) in 0.0250 seconds
```

Try It Yourself

Write a Java program using the HBase client API to perform the same put/get/scan operations programmatically.

Result

HBase was configured and CRUD operations were successfully performed on an HBase table using the HBase shell.

Experiment 10: Apache Spark – Basic RDD Operations

Aim

To install Apache Spark and perform basic transformations and actions on a Resilient Distributed Dataset (RDD).

Software / Tools Required

- Apache Spark
- Scala/PySpark shell

Theory

Apache Spark is a fast, in-memory distributed data processing engine. Its fundamental data structure is the Resilient Distributed Dataset (RDD), an immutable, partitioned collection of records that can be processed in parallel. RDDs support two types of operations: transformations (e.g. map, filter, flatMap), which are lazily evaluated, and actions (e.g. collect, count, reduce), which trigger actual computation.

Procedure

61. Start the PySpark or Spark-shell interactive shell.
62. Create an RDD from a local collection or a text file using sc.textFile / sc.parallelize.
63. Apply transformations such as map and filter to process the RDD.
64. Apply the flatMap and reduceByKey transformations to compute word counts.
65. Trigger computation using actions such as collect() and count().
66. Persist/cache the RDD and observe the performance difference on repeated actions.

Program / Commands

wordcount_rdd.py:

```
from pyspark import SparkContext

sc = SparkContext(appName='BasicRDDOps')
lines = sc.textFile('hdfs:///user/student/input.txt')

words = lines.flatMap(lambda line: line.split())
pairs = words.map(lambda word: (word, 1))
counts = pairs.reduceByKey(lambda a, b: a + b)

for word, count in counts.collect():
    print(f'{word}: {count}')

print('Total distinct words:', counts.count())
sc.stop()
```

Run:

```
spark-submit wordcount_rdd.py
```

Sample Output

```
Hadoop: 3
Spark: 5
distributed: 2
Total distinct words: 42
```

Try It Yourself

Rewrite the program using DataFrame's groupBy().count() instead of RDD transformations and compare execution time.

Result

Basic RDD transformations and actions were performed successfully using Apache Spark.

Experiment 11: Spark – DataFrame and SQL

Aim

To load structured data into a Spark DataFrame and perform data analysis using both the DataFrame API and Spark SQL.

Software / Tools Required

- Apache Spark
- PySpark
- Sample CSV data set

Theory

Spark SQL provides a programming abstraction called DataFrame, which is a distributed collection of data organized into named columns, similar to a table in a relational database. DataFrames can be manipulated using the DataFrame API or by registering them as temporary views and querying them with standard SQL through Spark SQL.

Procedure

67. Create a SparkSession, the entry point for DataFrame and SQL functionality.
68. Read a CSV file into a DataFrame with schema inference.
69. Explore the DataFrame using show(), printSchema(), and describe().
70. Apply DataFrame operations such as select, filter, and groupBy/agg.
71. Register the DataFrame as a temporary SQL view and run SQL queries on it.
72. Write the resulting DataFrame back to HDFS in Parquet format.

Program / Commands

dataframe_sql.py:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import avg, count

spark = SparkSession.builder.appName('DataFrameSQLDemo').getOrCreate()

df = spark.read.csv('hdfs:///user/student/employees.csv',
                    header=True, inferSchema=True)
df.printSchema()
df.show(5)

df.groupBy('department') \
  .agg(avg('salary').alias('avg_salary'), count('*').alias('headcount')) \
  .show()

df.createOrReplaceTempView('employees')
spark.sql("""
    SELECT department, AVG(salary) AS avg_salary
    FROM employees
    GROUP BY department
```

```
    ORDER BY avg_salary DESC
""").show()

df.write.mode('overwrite').parquet('hdfs:///user/student/employees_parquet')
```

Sample Output

```
+------------+----------+---------+
|  department|avg_salary|headcount|
+------------+----------+---------+
|Data Science|   78000.0|        1|
| Engineering|   60000.0|        2|
+------------+----------+---------+
```

Try It Yourself

Join the employees DataFrame with a departments DataFrame and find the department with the highest total salary expenditure.

Result

Structured data was analyzed successfully using Spark DataFrame operations and Spark SQL queries.

Experiment 12: Spark – Machine Learning with MLlib

Aim

To build and evaluate a simple machine learning model (Linear Regression) using Spark's MLlib library.

Software / Tools Required

- Apache Spark
- PySpark MLlib
- Sample numeric data set

Theory

MLlib is Spark's scalable machine learning library, providing implementations of common algorithms for classification, regression, clustering, and collaborative filtering, along with tools for feature extraction, pipelines, and model evaluation. It is designed to operate efficiently on distributed DataFrames, making it suitable for training models on very large data sets.

Procedure

73. Load the data set into a Spark DataFrame and inspect it for missing values.
74. Use VectorAssembler to combine feature columns into a single feature vector.
75. Split the data into training and test sets.
76. Train a LinearRegression model on the training set.
77. Generate predictions on the test set using the trained model.
78. Evaluate the model using RegressionEvaluator (e.g. RMSE, R-squared).

Program / Commands

spark_mllib_regression.py:

```
from pyspark.sql import SparkSession
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import LinearRegression
from pyspark.ml.evaluation import RegressionEvaluator

spark = SparkSession.builder.appName('MLlibRegressionDemo').getOrCreate()

data = spark.read.csv('hdfs:///user/student/housing.csv',
                      header=True, inferSchema=True)

assembler = VectorAssembler(
    inputCols=['area_sqft', 'bedrooms', 'age_years'],
    outputCol='features')
dataset = assembler.transform(data).select('features', 'price')

train_df, test_df = dataset.randomSplit([0.8, 0.2], seed=42)

lr = LinearRegression(featuresCol='features', labelCol='price')
model = lr.fit(train_df)
```

```
predictions = model.transform(test_df)
predictions.select('features', 'price', 'prediction').show(5)

evaluator = RegressionEvaluator(
    labelCol='price', predictionCol='prediction', metricName='rmse')
rmse = evaluator.evaluate(predictions)
print('Root Mean Squared Error (RMSE):', rmse)
```

Sample Output

```
+-------------------+--------+-----------------+
|           features|   price|       prediction|
+-------------------+--------+-----------------+
|[1200.0,2.0,5.0]   |4500000 | 4432110.5       |
|[1800.0,3.0,2.0]   |6800000 | 6721340.2       |
+-------------------+--------+-----------------+
Root Mean Squared Error (RMSE): 145230.7
```

Try It Yourself

Train a K-Means clustering model on the same data set to segment houses into price bands and visualize the clusters.

Result

A linear regression model was trained and evaluated successfully using Spark MLlib.

Experiment 13: Real-Time Big Data Project – I (Design and Implementation)

Aim

To identify a real-world problem, design an end-to-end big data workflow, and implement the data ingestion and processing modules.

Software / Tools Required

- Hadoop, Hive, Pig, HBase, and/or Spark as required
- A public real-world data set (e.g. retail, social media, or sensor data)

Theory

A real-time big data project integrates several tools studied in this laboratory into a single working pipeline: data ingestion (loading raw data into HDFS), storage and management (Hive/HBase), processing and transformation (MapReduce/Pig/Spark), and finally analysis and visualization. Careful architecture design ensures that each stage is modular and scalable.

Procedure

79. Select a real-world data set and clearly define the problem statement and expected insights.
80. Design the system architecture: identify data sources, storage layer, processing engine, and output/visualization layer.
81. Ingest the raw data set into HDFS, organizing it into a suitable directory structure.
82. Design and create the necessary Hive tables (or HBase schema) to store the structured/processed data.
83. Implement the core data processing logic (Spark job, MapReduce job, or Pig script) to clean and transform the raw data.
84. Run the processing job and validate the intermediate output for correctness.
85. Document the architecture diagram, module descriptions, and current progress.

Program / Commands

Sample project structure:

```
project/
  data/raw/                 # original data set
  ingestion/load_to_hdfs.sh # HDFS ingestion script
  processing/clean_transform.py  # Spark/Pig cleaning job
  hive/schema.sql           # Hive table DDL
  docs/architecture.png     # system design diagram
```

Try It Yourself

Prepare a one-page architecture diagram showing data flow from source to storage to processing, ready for review before the final demo.

Result

The problem statement and architecture for the real-time big data project were finalized, and the data ingestion and processing modules were implemented.

Experiment 14: Real-Time Big Data Project – II (Completion & Demo)

Aim

To complete the integration, testing, and analysis of the real-time big data project designed in Experiment 13, and to demonstrate the final working system.

Software / Tools Required

- Tools/scripts implemented in Experiment 13

Theory

The final stage of a big data project focuses on integrating all individual modules into a single working pipeline, testing it end-to-end with realistic data volumes, and presenting the derived insights through tables, charts, or dashboards. Performance tuning (e.g. partitioning, caching, indexing) is also carried out at this stage to ensure the pipeline runs efficiently.

Procedure

86. Integrate the ingestion, storage, and processing modules into a single automated pipeline (shell script or workflow).
87. Run the complete pipeline on the full data set and monitor job execution using the Hadoop/Spark web UI.
88. Query the final processed data using Hive/Spark SQL to generate the required analytical results.
89. Visualize the key insights using charts/tables (e.g. via a notebook, spreadsheet, or BI tool).
90. Perform basic performance tuning and record before/after execution times.
91. Prepare a final project report summarizing the objective, architecture, results, and conclusions.
92. Demonstrate the complete working pipeline and present the results in a viva-voce session.

Program / Commands

Sample pipeline driver script:

```
#!/bin/bash
set -e
echo 'Step 1: Ingesting data into HDFS'
bash ingestion/load_to_hdfs.sh

echo 'Step 2: Running Spark processing job'
spark-submit processing/clean_transform.py

echo 'Step 3: Loading processed data into Hive'
hive -f hive/load_processed.sql

echo 'Step 4: Running final analysis query'
hive -f hive/final_analysis.sql
echo 'Pipeline completed successfully.'
```

Sample Output

```
Step 1: Ingesting data into HDFS ... done
Step 2: Running Spark processing job ... done (42.3s)
Step 3: Loading processed data into Hive ... done
Step 4: Running final analysis query ... done
Pipeline completed successfully.
```

Try It Yourself

Schedule the pipeline script using cron or Apache Oozie to run automatically at fixed intervals.

Result

The real-time big data project was successfully completed, integrated end-to-end, and demonstrated with the derived analytical results and a final report.

VI. REFERENCE BOOKS

1. Behrouz A. Forouzan, **Cryptography and Network Security**, 3rd Edition, McGraw-Hill Education, 2015.

2. W. Richard Stevens, Bill Fenner & Andrew M. Rudoff, **UNIX Network Programming, Volume 1: The Sockets Networking API**, 3rd Edition,

3. Addison-Wesley, 2004. Kenneth C. Louden & Kenneth A. Lambert, **Programming in Python**, 2nd Edition, Cengage Learning, 2015.

VII. MATERIALS ONLINE

1. Course Outline Description

2. Lab Manul.