

## **EXPERIMENT-1**

**Aim :** Write a program Install, configure, and run Python, NumPy and Pandas.

### **Procedure : -**

- Install python
- Verify installation
- Install numpy and pandas
- Run a simple test program
- Done!

### **Input :**

#### **1. Install Python**

Go to: <https://www.python.org/downloads>

Download the latest version (e.g., Python 3.11 or newer)

Run the installer

On Windows: make sure you check “Add Python to PATH” before clicking "Install Now"

#### **2. Verify Installation**

Open your terminal or command prompt and run:

```
python --version
```

#### **3. Install NumPy and Pandas**

```
pip install numpy pandas
```

#### **4. Run a Simple Test Program**

```
import numpy as np
```

```
import pandas as pd
```

```
# NumPy Example
```

```
arr = np.array([1, 2, 3, 4, 5])
```

```
print("NumPy Array:", arr)
```

```
# Pandas Example
```

```
df = pd.DataFrame({
```

```
    "Name": ["Alice", "Bob", "Charlie"],
```

```
    "Age": [25, 30, 35]
```

```
})
```

```
print("\nPandas DataFrame:\n", df)
```

## Output :

```
import numpy as np
import pandas as pd

# NumPy array
arr = np.array([1, 2, 3, 4])
print("NumPy array:", arr)

# Pandas DataFrame
data = {'Name': ['Alice', 'Bob'], 'Age': [25, 30]}
df = pd.DataFrame(data)
print("Pandas DataFrame:\n", df)
```

```
NumPy array: [1 2 3 4]
Pandas DataFrame:
   Name  Age
0  Alice   25
1   Bob   30
```

**Result :** The program executed successfully.

## **EXPERIMENT-2**

**Aim :** Write steps to install Apache Hadoop and HDFS.

**Procedure :-**

- Install java
- Download hadoop
- Configure environment variables
- Configure SSH
- Configure Hadoop files
- Format hdfs
- Start hdfs
- Done

**Steps :**

### **1. Install Java**

```
apt update  
apt install openjdk-11-jdk -y
```

### **2. Download & Extract Hadoop**

```
wget https://downloads.apache.org/hadoop/common/hadoop-3.3.6/hadoop-3.3.6.tar.gz  
tar -xzf hadoop-3.3.6.tar.gz  
mv hadoop-3.3.6 ~/hadoop
```

### **3. Set Environment Variables**

```
export HADOOP_HOME=~/hadoop  
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64  
export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin
```

### **4. Setup SSH**

```
sudo apt install ssh -y  
ssh-keygen -t rsa -P ""  
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys  
ssh localhost
```

## 5. Configure Hadoop

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

## 6. Format HDFS

```
hdfs namenode -format
```

## 7. Start Hadoop

```
start-dfs.sh
```

## 8. Test HDFS

```
echo "hello" > test.txt  
hdfs dfs -mkdir /user  
hdfs dfs -put test.txt /user/  
hdfs dfs -ls /user/
```

**Result :** You have successfully write the steps to install apache hadoop.

### **EXPERIMENT-3**

**Aim :** Develop a MapReduce program to calculate the frequency of a given word in each file.

**Procedure : -**

- Write the mapper
- Write the reducer
- Run with hadoop streaming
- Check the output

**Input :**

```
from collections import defaultdict
#calling of libraries.

# Sample data (like lines from a file)
data = [
    "jai baba ki",
    "jai mata ki",
    "mata ji"
    "Map Reduce program written in SRM",
    "Hello BCA students again",
]
# Mapper function
#-----
def mapper(line):
    for word in line.strip().lower().split():
        yield (word, 1)
# -----
# Reducer function
def reducer(pairs):
    counts = defaultdict(int)
    for word, count in pairs:
        counts[word] += count
    return counts

# -----
# Simulated MapReduce
# -----
def run_mapreduce(data):
    # Map phase
    mapped = []
    for line in data:
        mapped.extend(mapper(line))
```

```

# Shuffle and sort (group by key)
shuffled = defaultdict(list)
for word, count in mapped:
    shuffled[word].append(count)
# Reduce phase
reduced = {}
for word, counts in shuffled.items():
    reduced[word] = sum(counts)
return reduced
# Run the program
if __name__ == "__main__":
    result = run_mapreduce(data)
    for word in sorted(result):
        print(f"{word}\t{result[word]}")

```

### Output :

```

again 1
baba 1
bca 1
hello 1
in 1
jai 2
jimap 1
ki 2
mata 2
program 1
reduce 1
srm 1
students 1
written 1

```

---

**Result :** The program executed successfully.

## **EXPERIMENT-4**

**Aim :** Develop a MapReduce program to find the maximum temperature in each year.

**Procedure : -**

- Write the mapper
- Write the reducer
- Run with hadoop streaming
- Check the output

**Input :**

```
from pyspark.sql import SparkSession
# Initialize Spark session
spark = SparkSession.builder.appName("MaxTempPerYear").getOrCreate()

# Your data as list of strings "year,temp"
data = [
    "2018,18",
    "2019,35",
    "2019,18",
    "2020,35",
    "2020,48",
    "2021,40",
    "2021,42",
    "2021,45",
    "2023,36",
    "2023,45",
    "2024,42",
    "2025,39",
    "2025,44"
]
# Create RDD from the list
rdd = spark.sparkContext.parallelize(data)

# Parse each record into (year, temperature) tuple
def parse_record(record):
    year, temp = record.strip().split(",")
    return (year, int(temp))
parsed_rdd = rdd.map(parse_record)
# Reduce by key (year) to find max temperature
max_temp_per_year = parsed_rdd.reduceByKey(lambda a, b: max(a, b))
# Collect and print results sorted by year
results = max_temp_per_year.collect()
```

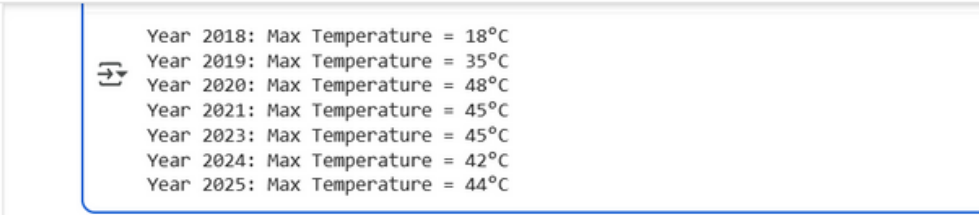
or year, temp in sorted(results):

```
print(f"Year {year}: Max Temperature = {temp}°C")
```

```
# Stop Spark session
```

```
spark.stop()
```

### Output :



```
Year 2018: Max Temperature = 18°C  
Year 2019: Max Temperature = 35°C  
Year 2020: Max Temperature = 48°C  
Year 2021: Max Temperature = 45°C  
Year 2023: Max Temperature = 45°C  
Year 2024: Max Temperature = 42°C  
Year 2025: Max Temperature = 44°C
```

**Result :** The program executed successfully.



## **EXPERIMENT-5**

**Aim :** Develop a MapReduce program to find the grades of student's.

**Procedure : -**

- Write the mapper
- Write the reducer
- Run with hadoop streaming
- Check the output

**Input :**

```
from pyspark.sql import SparkSession
def assign_grade(mark):
    if mark >= 90:
        return 'A'
    elif mark >= 80:
        return 'B'
    elif mark >= 70:
        return 'C'
    elif mark >= 60:
        return 'D'
    else:
        return 'F'
def main():
    # Initialize a SparkSession
    spark = SparkSession.builder \
        .appName("StudentGrades") \
        .getOrCreate()
    # Get the SparkContext from the SparkSession
    sc = spark.sparkContext
```

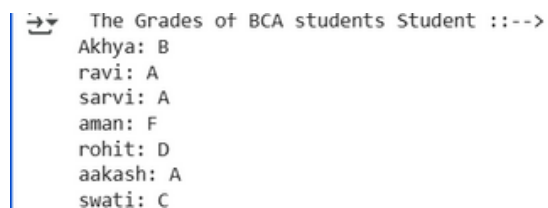
```

# Create a sample dataset of BCA students
student_data = [
    ("Akhyia", 85),
    ("ravi", 92),
    ("sarvi", 98),
    ("aman", 55),
    ("rohit", 68),
    ("aakash", 99),
    ("swati", 71)
]

student_rdd = sc.parallelize(student_data)
grades_rdd = student_rdd.map(lambda record: (record[0], assign_grade(record[1])))
# Collect the results
results = grades_rdd.collect()
# Print the results
print(" The Grades of BCA students Student ::-->")
for name, grade in results:
    print(f"{name}: {grade}")
# Stop the SparkSession
spark.stop()
if __name__ == "__main__":
    main()

```

## Output :



```

→ The Grades of BCA students Student ::-->
Akhyia: B
ravi: A
sarvi: A
aman: F
rohit: D
aakash: A
swati: C

```

**Result :** The program executed successfully.

## **EXPERIMENT-6**

**Aim :** Develop a MapReduce program to implement Matrix Multiplication.

**Procedure : -**

- Write the mapper
- Write the reducer
- Run with hadoop streaming
- Check the output

**Input :**

```
from pyspark import SparkContext
```

```
def main():
```

```
    sc = SparkContext("local", "Matrix Multiplication")
```

```
    A = [
```

```
        (0, [1, 2, 3]), # Row 0
```

```
        (1, [4, 5, 6]), # Row 1
```

```
    ] # 2x3 matrix
```

```
    B = [
```

```
        (0, [7, 8]), # Row 0
```

```
        (1, [9, 10]), # Row 1
```

```
        (2, [11, 12]), # Row 2
```

```
    ] # 3x2 matrix
```

```
    rddA = sc.parallelize(A)
```

```
    rddB = sc.parallelize(B)
```

```
    rddA_mapped = rddA.flatMap(lambda x: [(x[0], (k, v)) for k, v in enumerate(x[1])])
```

```
    # Emit (k, (j, B[k][j])) for matrix B
```

```
    rddB_mapped = rddB.flatMap(lambda x: [(k, (x[0], v)) for k, v in enumerate(x[1])])
```

```
    rdd_joined = rddA_mapped.join(rddB_mapped)
```

```
rdd_result = rdd_joined.map(lambda x: ((x[1][0][0], x[1][1][0]), x[1][0][1] * x[1][1][1]))  
.reduceByKey(lambda x, y: x + y)
```

```
# Collect and print the result
```

```
result = rdd_result.collect()
```

```
print("Resulting Matrix C:")
```

```
for ((i, j), value) in result:
```

```
    print(f"C[{i}][{j}] = {value}")
```

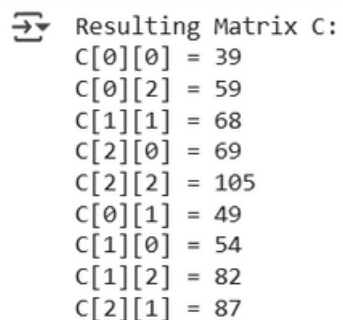
```
#Gracefully shuts down the Spark context to free up resources.
```

```
sc.stop()
```

```
if __name__ == "__main__":
```

```
    main()
```

## Output :



```
Resulting Matrix C:  
C[0][0] = 39  
C[0][2] = 59  
C[1][1] = 68  
C[2][0] = 69  
C[2][2] = 105  
C[0][1] = 49  
C[1][0] = 54  
C[1][2] = 82  
C[2][1] = 87
```

**Result :** The program executed successfully.

## **EXPERIMENT-7**

**Aim :** Develop a MapReduce to find the maximum electrical consumption in each year given the electrical consumption for each month in each year

**Procedure : -**

- Write the mapper
- Write the reducer
- Run with hadoop streaming
- Check the output

**Input :**

```
from pyspark import SparkContext
```

```
# Example SRM consumption electricity yearwise input
```

```
data = [  
    (2020, "Jan", 120),  
    (2020, "Feb", 135),  
    (2020, "Mar", 110),  
    (2021, "Jan", 140),  
    (2021, "Feb", 160),  
    (2021, "Mar", 155),  
    (2022, "Jan", 145),  
    (2022, "Feb", 150),  
    (2022, "Mar", 138)  
]  
  
max_consumption = {}
```

```
for year, month, consumption in data:
```

```
    if year not in max_consumption:
```

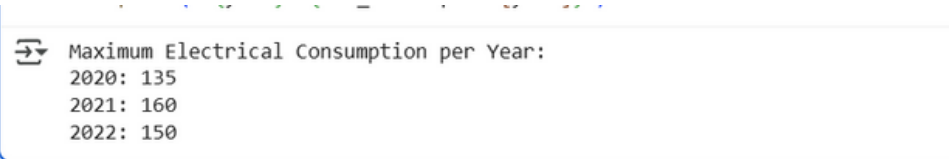
```
        max_consumption[year] = consumption
```

```
    else:
```

```
        max_consumption[year] = max(max_consumption[year], consumption)
```

```
print("Maximum Electrical Consumption per Year:")  
for year in sorted(max_consumption):  
    print(f"{year}: {max_consumption[year]}")
```

### Output :

A screenshot of a terminal window with a light blue border. The terminal shows the output of a Python program. The first line is "Maximum Electrical Consumption per Year:". The subsequent lines are "2020: 135", "2021: 160", and "2022: 150".

```
Maximum Electrical Consumption per Year:  
2020: 135  
2021: 160  
2022: 150
```

**Result :** The program executed successfully.

## **EXPERIMENT-8**

**Aim :** Develop a MapReduce to analyse weather data set and print whether the day is shinny or cool day.

**Procedure : -**

- Write the mapper
- Write the reducer
- Run with hadoop streaming
- Check the output

**Input :**

```
from pyspark import SparkContext
```

```
# Step 1: Initialize SparkContext
```

```
sc = SparkContext("local", "WeatherAnalysis")
```

```
# Step 2: Create dataset in the program
```

```
weather_data = [  
    ("2023-06-01", 29),  
    ("2023-06-02", 22),  
    ("2023-06-03", 25),  
    ("2023-06-04", 19),  
    ("2023-06-05", 30),  
    ("2023-06-06", 21)  
]
```

```
# Step 3: Parallelize the data (convert to RDD)
```

```
rdd = sc.parallelize(weather_data)
```

```
# Step 4: Map function to classify days
```

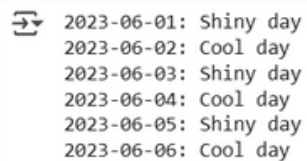
```
def classify_day(record):
```

```
    date, temp = record
```

```
    label = "Shiny day" if temp >= 25 else "Cool day"
```

```
    return (date, label)
result = rdd.map(classify_day)
# Step 5: Collect and print results
for date, label in result.collect():
    print(f'{date}: {label}')
# Step 6: Stop SparkContext
sc.stop()
```

### Output :

A terminal window showing the output of the Spark program. The output consists of six lines, each representing a date and its corresponding label. The dates are 2023-06-01 through 2023-06-06, and the labels are 'Shiny day' and 'Cool day' alternating.

```
⇒ 2023-06-01: Shiny day
   2023-06-02: Cool day
   2023-06-03: Shiny day
   2023-06-04: Cool day
   2023-06-05: Shiny day
   2023-06-06: Cool day
```

**Result :** The program executed successfully.



## **EXPERIMENT-9**

**Aim :** Develop a MapReduce program to find the number of products sold in each country by considering sales data containing fields

**Procedure : -**

- Write the mapper
- Write the reducer
- Run with hadoop streaming
- Check the output

**Input :**

```
from pyspark import SparkContext
```

```
# Step 1: Initialize SparkContext
```

```
sc = SparkContext("local", "SalesByCountry")
```

```
# Step 2: Create sample sales data within the program
```

```
# Format: (Country, Product, Quantity)
```

```
sales_data = [  
    ("USA", "Laptop", 2),  
    ("India", "Phone", 1),  
    ("USA", "Tablet", 3),  
    ("UK", "Laptop", 1),  
    ("India", "Laptop", 2),  
    ("UK", "Phone", 1),  
    ("USA", "Phone", 1),  
    ("India", "Tablet", 1)
```

```
]
```

```
# Step 3: Parallelize the data
```

```
rdd = sc.parallelize(sales_data)
```

```
country_sales = rdd.map(lambda record: (record[0], record[2])) \
    .reduceByKey(lambda a, b: a + b)
```

# Step 5: Collect and print the result

```
print("Number of products sold per country:")
```

```
or country, total in country_sales.collect():
```

```
    print(f'{country}: {total}')
```

# Step 6: Stop SparkContext

```
sc.stop()
```

## Output :

```
➦ Number of products sold per country:
USA: 6
India: 4
UK: 2
```

---

**Result :** The program executed successfully.

## **EXPERIMENT-10**

**Aim :** Develop a MapReduce program to find the tags associated with each movie by analysing movie lens data.

**Procedure : -**

- Write the mapper
- Write the reducer
- Run with hadoop streaming
- Check the output

**Input :**

```
# movie_tags.py
```

```
from pyspark import SparkContext
```

```
# Step 1: Initialize SparkContext
```

```
sc = SparkContext("local", "MovieTags")
```

```
# Step 2: Create sample MovieLens tag data
```

```
# Format: (movie_id, user_id, tag)
```

```
tag_data = [
```

```
    (1, 101, "funny"),
```

```
    (1, 102, "romantic"),
```

```
    (2, 103, "thriller"),
```

```
    (1, 104, "comedy"),
```

```
    (3, 105, "drama"),
```

```
    (2, 106, "action")
```

```
]
```

```
# Step 3: Parallelize the dataset
```

```
rdd = sc.parallelize(tag_data)
```

```
# Step 4: Map to (movie_id, tag)
```

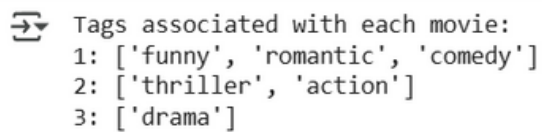
```
movie_tags = rdd.map(lambda x: (x[0], x[2]))
```

```
# Step 5: Reduce by movie_id to get list of tags
tags_per_movie = movie_tags.groupByKey().mapValues(list)

# Step 6: Print results
print("Tags associated with each movie:")
for movie_id, tags in tags_per_movie.collect():
    print(f"{movie_id}: {tags}")

# Step 7: Stop SparkContext
sc.stop()
```

### Output :

A terminal window with a light blue border. It contains the output of the Spark program. The first line is "Tags associated with each movie:". The next three lines are numbered 1, 2, and 3, each followed by a list of tags in single quotes.

```
➤ Tags associated with each movie:
1: ['funny', 'romantic', 'comedy']
2: ['thriller', 'action']
3: ['drama']
```

**Result :** The program executed successfully.

## **EXPERIMENT-11**

**Aim :** XYZ.com is an online music website where users listen to various tracks.

the data gets collected which is given,

**Procedure : -**

- Write the mapper
- Write the reducer
- Run with hadoop streaming
- Check the output

**Input :**

```
from pyspark import SparkContext, SparkConf
```

```
def main():
```

```
    """
```

```
    Main function to run the PySpark MapReduce job for counting music track plays.
```

```
    """
```

```
    # 1. Setup: Initialize SparkContext
```

```
    # This sets up the connection to a Spark cluster. "local" means it runs on your machine.
```

```
    conf = SparkConf().setAppName("MusicTrackCounter").setMaster("local")
```

```
    sc = SparkContext(conf=conf)
```

```
    # 2. Input Data: A list representing user track plays on XYZ.com
```

```
    # In a real-world scenario, you would load this from a file:
```

```
    sc.textFile("path/to/your/log.txt")
```

```
    track_plays_data = [
```

```
        "SummerVibes", "MidnightMelody", "RetroFunk", "SummerVibes",
```

```
        "OceanBreeze", "MidnightMelody", "SummerVibes", "RetroFunk",
```

```
        "CityLights", "MidnightMelody", "OceanBreeze", "UrbanGroove"
```

```
    ]
```

```
    # Create an RDD (Resilient Distributed Dataset) from the data
```

```
    # RDDs are the fundamental data structure in Spark.
```

```
    tracks_rdd = sc.parallelize(track_plays_data)
```

```
mapped_rdd = tracks_rdd.map(lambda track: (track, 1))
```

#### # 4. Reduce Phase

```
# The reduceByKey operation groups all pairs with the same key (track_name)
```

```
# and applies the specified function to their values.
```

```
# lambda a, b: a + b simply adds the counts together.
```

```
# For "SummerVibes", it will compute (1 + 1 + 1) = 3.
```

```
counts_rdd = mapped_rdd.reduceByKey(lambda a, b: a + b)
```

#### # 5. Action: Collect the results

```
# The .collect() action brings the final computed data from the distributed workers
```

```
# back to the main driver program.
```

```
results = counts_rdd.collect()
```

#### # 6. Output: Display the results

```
print("--- Track Play Counts for XYZ.com ---")
```

```
for track, count in sorted(results, key=lambda item: item[1], reverse=True):
```

```
    print(f"🎵 {track}: {count} plays")
```

```
print("-----")
```

#### # 7. Cleanup: Stop the SparkContext

```
sc.stop()
```

```
if __name__ == "__main__":
```

```
    main()
```

### Output :

```
--- Track Play Counts for XYZ.com ---
🎵 SummerVibes: 3 plays
🎵 MidnightMelody: 3 plays
🎵 RetroFunk: 2 plays
🎵 OceanBreeze: 2 plays
🎵 CityLights: 1 plays
🎵 UrbanGroove: 1 plays
-----
```

**Result :** The program executed successfully.

## **EXPERIMENT-12**

**Aim :** Develop a MapReduce program to find the frequency of books published each year and find in which year maximum number of books were published using the given data.

**Procedure : -**

- Write the mapper
- Write the reducer
- Run with hadoop streaming
- Check the output

**Input :**

```
from pyspark import SparkConf, SparkContext

# configuration spark setup
#conf = SparkConf().setAppName("BooksPerYear").setMaster("local[*]")
#sc = SparkContext(conf=conf)

# -----
# Inbuilt data (id, title, author, year)
# -----

books = [
    "1,The Great Gatsby,F. Scott Fitzgerald,1925",
    "2,To Kill a Mockingbird,Harper Lee,1960",
    "3,1984,George Orwell,1949",
    "4,Pride and Prejudice,Jane Austen,1813",
    "5,The Catcher in the Rye,J.D. Salinger,1951",
    "6,Animal Farm,George Orwell,1945",
    "7,The Fellowship of the Ring,J.R.R. Tolkien,1954",
    "8,The Two Towers,J.R.R. Tolkien,1954",
    "9,The Return of the King,J.R.R. Tolkien,1955"
]
```

```

data = sc.parallelize(books)

#this is map reduce function
# 1. Extract year (last column)
year_counts = data.map(lambda line: line.split(",")[-1]) \
    .map(lambda year: (year, 1)) \
    .reduceByKey(lambda a, b: a + b)

# 2. Collect frequency of books per year (counting per year)
books_per_year = year_counts.collect()

print("number of books published each year:")
for year, count in sorted(books_per_year, key=lambda x: x[0]):
    print(year, ":", count)

# 3. Find year with maximum number of books
max_year = year_counts.reduce(lambda a, b: a if a[1] > b[1] else b)

print("\nYear with maximum books published:", max_year[0], "with", max_year[1], "books")

# Stop Spark

sc.stop()

```

## Output :

```

. . .
number of books published each year:
1813 : 1
1925 : 1
1945 : 1
1949 : 1
1951 : 1
1954 : 2
1955 : 1
1960 : 1

Year with maximum books published: 1954 with 2 books

```

**Result :** The program executed successfully.



## **EXPERIMENT-13**

**Aim :** Develop a MapReduce program to analyse Titanic ship data and to find the average age of the people (both male and female) who died in the tragedy. How many people survive in each class.

**Procedure : -**

- Write the mapper
- Write the reducer
- Run with hadoop streaming
- Check the output

**Input :**

```
from pyspark.sql import SparkSession
```

```
# Initialize Spark
```

```
spark = SparkSession.builder.appName("TitanicMapReduce").getOrCreate()
```

```
sc = spark.sparkContext
```

```
# Sample Titanic data (as Python dictionaries)
```

```
titanic_data = [  
    {"PassengerId": 1, "Survived": 0, "Pclass": 3, "Name": "John Smith", "Sex": "male", "Age": 22},  
    {"PassengerId": 2, "Survived": 1, "Pclass": 1, "Name": "Mary Johnson", "Sex": "female", "Age": 38},  
    {"PassengerId": 3, "Survived": 1, "Pclass": 3, "Name": "William Brown", "Sex": "male", "Age": 26},  
    {"PassengerId": 4, "Survived": 1, "Pclass": 1, "Name": "Elizabeth Davis", "Sex": "female", "Age": 35},  
    {"PassengerId": 5, "Survived": 0, "Pclass": 3, "Name": "Thomas Miller", "Sex": "male", "Age": 35},  
    {"PassengerId": 6, "Survived": 0, "Pclass": 3, "Name": "Anna Wilson", "Sex": "female", "Age": None},  
    {"PassengerId": 7, "Survived": 0, "Pclass": 1, "Name": "James Anderson", "Sex": "male", "Age": 54},  
    {"PassengerId": 8, "Survived": 1, "Pclass": 2, "Name": "Emily Clark", "Sex": "female", "Age": 28},  
    {"PassengerId": 9, "Survived": 0, "Pclass": 3, "Name": "Robert Moore", "Sex": "male", "Age": 19},  
    {"PassengerId": 10, "Survived": 1, "Pclass": 2, "Name": "Susan Taylor", "Sex": "female", "Age": 14}  
]
```

```
# Create an RDD from the list of dictionaries
```

```
rdd = sc.parallelize(titanic_data)
```

```
died_rdd = rdd.filter(lambda x: x["Survived"] == 0 and x["Age"] is not None)
```

```
# Map to (gender, (age, 1))
```

```
age_pairs = died_rdd.map(lambda x: (x["Sex"], (x["Age"], 1)))
```

```
# Reduce to (sum of ages, count)
```

```
age_totals = age_pairs.reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1]))
```

```
# Compute average age
```

```
average_age = age_totals.mapValues(lambda x: x[0] / x[1])
```

```
print("Average Age of People Who Died (by Gender):")
```

```
for gender, avg_age in average_age.collect():
```

```
    print(f'{gender}: {avg_age:.2f}')
```

```
# -----
```

```
# Part 2: Number of Survivors in Each Class
```

```
# -----
```

```
# Filter people who survived
```

```
survivors_rdd = rdd.filter(lambda x: x["Survived"] == 1)
```

```
# Map to (Pclass, 1)
```

```
class_pairs = survivors_rdd.map(lambda x: (x["Pclass"], 1))
```

```
# Reduce to count
```

```
survivor_counts = class_pairs.reduceByKey(lambda a, b: a + b)
```

```
print("\nNumber of Survivors by Passenger Class:")
```

```
for pclass, count in survivor_counts.collect():
```

```
    print(f'Class {pclass}: {count}')
```

## Output :

```
➡ Average Age of People who Died (by Gender):  
male: 32.50
```

```
Number of Survivors by Passenger Class:  
Class 2: 2  
Class 1: 2  
Class 3: 1
```

**Result :** The program executed successfully.

## **EXPERIMENT-14**

**Aim :** Develop a MapReduce program to analyse Uber dataset to find the days on which each basement has more trips using the given dataset.

### **Procedure : -**

- Write the mapper
- Write the reducer
- Run with hadoop streaming
- Check the output

### **Input :**

```
from pyspark.sql import SparkSession
```

```
# Initialize Spark
```

```
spark = SparkSession.builder.appName("UberBaseAnalysis").getOrCreate()
```

```
sc = spark.sparkContext
```

```
# Sample Uber dataset as list of dictionaries
```

```
uber_data = [  
    {"Date": "1/1/2015", "Base": "B02512", "Trips": 190},  
    {"Date": "1/1/2015", "Base": "B02598", "Trips": 225},  
    {"Date": "1/2/2015", "Base": "B02512", "Trips": 215},  
    {"Date": "1/2/2015", "Base": "B02598", "Trips": 197},  
    {"Date": "1/3/2015", "Base": "B02512", "Trips": 300},  
    {"Date": "1/3/2015", "Base": "B02598", "Trips": 225},  
    {"Date": "1/4/2015", "Base": "B02512", "Trips": 150},  
    {"Date": "1/4/2015", "Base": "B02598", "Trips": 225},  
    {"Date": "1/5/2015", "Base": "B02512", "Trips": 300},  
    {"Date": "1/5/2015", "Base": "B02598", "Trips": 180}  
]
```

```
# Create RDD
```

```
rdd = sc.parallelize(uber_data)
```

```

base_date_trips = rdd.map(lambda x: ((x["Base"], x["Date"]), x["Trips"]))

# Step 2: Sum trips per (Base, Date) in case data has duplicates
base_date_trip_sums = base_date_trips.reduceByKey(lambda a, b: a + b) # ((Base, Date), TotalTrips)

# Step 3: Map to (Base, (Date, TotalTrips))
base_to_date_trip = base_date_trip_sums.map(lambda x: (x[0][0], (x[0][1], x[1])))

# Step 4: Group by Base
grouped_by_base = base_to_date_trip.groupByKey()

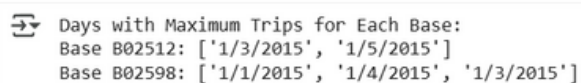
# Step 5: For each base, find date(s) with max trips
def get_max_days(trip_list):
    trip_list = list(trip_list)
    max_trips = max(trip_list, key=lambda x: x[1])[1]
    return [date for date, trips in trip_list if trips == max_trips]

result = grouped_by_base.mapValues(get_max_days)

# Print results
print("Days with Maximum Trips for Each Base:")
for base, days in result.collect():
    print(f'Base {base}: {days}')

```

## Output :



```

Days with Maximum Trips for Each Base:
Base B02512: ['1/3/2015', '1/5/2015']
Base B02598: ['1/1/2015', '1/4/2015', '1/3/2015']

```

**Result :** The program executed successfully.

## **EXPERIMENT-15**

**Aim :** Develop a program to calculate the maximum recorded temperature by year-wise for the weather dataset in Pig Latin

**Procedure : -**

- Write the mapper
- Write the reducer
- Run with hadoop streaming
- Check the output

**Input :**

```
def parse_line(line):
    """
    Extract year and temperature from a fixed-width line.
    """
    year = line[0:4]
    temp_str = line[8:11]

    try:
        temperature = int(temp_str)
    except ValueError:
        temperature = None

    return year, temperature


def max_temperature_by_year(dataset):
    """
    Compute the max temperature for each year from the dataset.
    """
    max_temps = {}
```

for line in dataset:

```
    year, temperature = parse_line(line.strip())
```

```
    if temperature is None:
```

```
        continue # Skip bad data
```

```
    if year not in max_temps or temperature > max_temps[year]:
```

```
        max_temps[year] = temperature
```

```
return max_temps
```

# === Inline Dataset (Simulated Fixed-Width Lines) ===

```
weather_data = [
```

```
    "19490101123", # Jan 1, 1949, temp = 123
```

```
    "19490102135", # Jan 2, 1949, temp = 135
```

```
    "19490103130", # Jan 3, 1949, temp = 130
```

```
    "19500101350", # Jan 1, 1950, temp = 350
```

```
    "19500102345", # Jan 2, 1950, temp = 345
```

```
    "19500103300", # Jan 3, 1950, temp = 300
```

```
    "19510101280", # Jan 1, 1951, temp = 280
```

```
    "19510102310", # Jan 2, 1951, temp = 310
```

```
    "19510103090" # Jan 3, 1951, temp = 90 (lowest)
```

```
]
```

# === Main Execution ===

```
if __name__ == "__main__":
```

```
    results = max_temperature_by_year(weather_data)
```

```
    print("Max temperature by year:")
```

```
    for year in sorted(results):
```

```
        print(f"{year}: {results[year]}")
```

## Output :

```
↔ Max temperature by year:  
1949: 135  
1950: 350  
1951: 310
```

**Result :** The program executed successfully.



## EXPERIMENT-16

**Aim :** What are the aggregate functions in HiveQL? Write queries to sort and aggregate the data in a table using HiveQL.

**Answer:**

In HiveQL, aggregate functions are used to perform calculations on a set of values and return a single summarized result. These functions are typically used with the GROUP BY clause to aggregate data across rows. Below is an explanation of common aggregate functions in HiveQL, followed by example queries to sort and aggregate data in a table.

### Common Aggregate Functions in HiveQL

1. COUNT(expr): Returns the number of rows where expr is not NULL. Use COUNT(\*) to count all rows, including those with NULL values.
2. SUM(expr): Computes the sum of expr for all non-NULL values in the group.
3. AVG(expr): Calculates the average of expr for all non-NULL values in the group.
4. MIN(expr): Returns the minimum value of expr in the group.
5. MAX(expr): Returns the maximum value of expr in the group.
6. VARIANCE(expr) / VAR\_POP(expr): Computes the population variance of expr.
7. VAR\_SAMP(expr): Computes the sample variance of expr.
8. STDDEV(expr) / STDDEV\_POP(expr): Computes the population standard deviation of expr.
9. STDDEV\_SAMP(expr): Computes the sample standard deviation of expr.
10. COLLECT\_SET(expr): Returns a set of unique values for expr (removes duplicates).
11. COLLECT\_LIST(expr): Returns a list of all values for expr, including duplicates.
12. APPROX\_COUNT\_DISTINCT(expr): Estimates the number of distinct values for expr (useful for large datasets).
13. PERCENTILE(expr, percentile): Computes the percentile value for expr at the specified percentile.
14. SUM(DISTINCT expr): Computes the sum of distinct non-NULL values of expr.

I'll demonstrate HiveQL aggregate functions and sorting queries using a **student** table. I'll first define a sample **student** table schema, provide sample data, and then write queries to aggregate and sort the data using HiveQL.

#### Student Table Schema

Assume the **student** table has the following columns:

```
student(  
  student_id  INT,  name  
  STRING,    department  
  STRING,    marks  
  DOUBLE,   age    INT,  
  enrollment_date  
  STRING  
)
```

Sample Data

student_id	name	department	marks	age	enrollment_date
1	Alice	CS	85.0	20	2024-09-01
2	Bob	ECE	78.0	21	2024-09-01
3	Charlie	CS	92.0	19	2024-09-02
4	Diana	ME	65.0	22	2024-09-03
5	Eve	CS	88.0	20	2024-09-03

HiveQL Queries for Aggregation and Sorting

1. Aggregate Query: Average Marks by Department

This query calculates the average marks for each department.

```
SELECT
    department, AVG(marks) AS
    avg_marks
FROM student GROUP BY
    department;
```

	avg_marks

2. Aggregate Query: Count of Students by Department

This query counts the number of students in each department.

```
SELECT
    department, COUNT(*) AS
    student_count
FROM student GROUP
    BY department;
```

Output:

department	student_count
CS	3
ECE	1
ME	1

### 3. Sorting Query: Sort Students by Marks (Descending)

This query lists all students sorted by their marks in descending order.

```
SELECT
  name,
  department,
  marks
FROM student
ORDER BY marks
DESC;
```

Output:

name	department	marks
Charlie	CS	92.0
Eve	CS	88.0
Alice	CS	85.0
Bob	ECE	78.0
Diana	ME	65.0

### 4. Aggregate Query with Filtering: Departments with Average Marks > 80

This query finds departments where the average marks are greater than 80, **HAVING** to using filter groups.

```
SELECT
  department, AVG(marks)
  AS avg_marks
FROM student
GROUP BY department
HAVING avg_marks > 80;
```

Output:

department	avg_marks
CS	88.33

### 5. Aggregate Query with Multiple Aggregations: Department Statistics

This query computes multiple metrics for each department: total marks, minimum marks, maximum marks, and unique student names.

*SELECT*

*department, SUM(marks) AS total\_marks,  
MIN(marks) AS min\_marks, MAX(marks) AS  
max\_marks, COLLECT\_SET(name) AS  
student\_names*

*FROM student*

*GROUP BY*

*department;*

Output:

department	total_marks	min_marks	max_marks	student_name
CS	265.0	85.0	92.0	[Alice, Charlie, Eve] [Bob] [Diana]
ECE	78.0	78.0	78.0	
ME	65.0	65.0	65.0	

## 6. Sorting and Aggregation: Top 2 Departments by Student Count

This query counts students per department, sorts by count in descending order, and limits to the top 2 departments.

*SELECT*

*department, COUNT(\*) AS  
student\_count  
FROM student GROUP BY  
department ORDER BY  
student\_count DESC LIMIT 2;*

Output:

department	student_count
CS	3
ECE	1

## 7. Aggregate Query: Standard Deviation of Marks by Department

This query calculates the sample standard deviation of marks for each department.

*SELECT*

*department,  
STDDEV\_SAMP(marks) AS  
marks\_stddev  
FROM student GROUP BY  
department;*

Output:

department	marks_stddev
CS	3.51
ECE	NULL
ME	NULL

### 8. Aggregate Query: Students by Age Group

This query groups students by age and calculates the count and average marks for each age group, sorted by age.

```
SELECT
age,
COUNT(*) AS student_count,
AVG(marks) AS avg_marks
FROM student
GROUP BY age
ORDER BY age
ASC;
```

Output:

age	student_count	avg_marks
19	1	92.0
20	2	86.5
21	1	78.0
22	1	65.0

### Notes

Aggregate Functions Used: **AVG**, **COUNT**, **SUM**, **MIN**, **MAX**, **STDDEV\_SAMP**, **COLLECT\_SET**. Sorting: The **ORDER BY** clause is used to sort results (**ASC** for ascending, **DESC** for descending). Filtering Groups: The **HAVING** clause filters aggregated results (e.g., departments with avg\_marks > 80).

NULL Handling: Aggregate functions ignore NULL values, and functions like **STDDEV\_SAMP** return NULL for single-value groups.