Vivekanand Education Society's
Institute of Technology
Approved by AICTE & Affiliated to University of Mumbai

Artificial Intelligence and Data Science Department

Big Data Analytics/Odd Sem 2023-23/Experiment 2B

Name : Prasad Jawale	**Class/Roll No.:** D16AD 20	**Grade :**

Title of Experiment :
To implement the following programs using pyspark.
1. program to find no of words starting specific letter (e.g. 'h'/'a')
2. RDBMS operations:
   - Selection
   - Projection
   - Union
   - Aggregates and grouping
   - Joins
   - Intersection

Objective of Experiment :

The objective of the project is to leverage PySpark, a powerful data processing framework, to implement two distinct tasks. The first task involves developing a program to analyze text data and determine the number of words starting with specific letters. The second task focuses on performing fundamental relational database management system operations, including selection, projection, union, aggregates with grouping , Joins and Intersections using Pyspark.

Outcome of Experiment :

Thus we implemented both programs using Pyspark

Problem Statement :

a. Create a program to count words starting with specific letters ina given test dataset.
b. Implemented key Relational Database management System operations using Pyspark, including selections, projection, Union, Aggregates with grouping Joins, and Intersections.

Description / Theory :

1. **Spark:**
   Apache Spark is a critical player in the realm of big data processing. This open-source, distributed computing framework excels at handling vast volumes of data with remarkable speed and efficiency. Unlike traditional batch processing systems, Spark leverages in-memory computing, enabling it to store and manipulate data in RAM, resulting in significantly faster data processing. It is inherently designed for distributed computing, allowing it to distribute data across a cluster of machines and perform parallel processing, thus scaling gracefully to tackle massive datasets and computationally intensive tasks. Its user-friendly APIs in languages like Java, Scala, Python, and R, along with libraries for machine learning, graph processing, and SQL querying, make it accessible to a broad range of developers. Spark's adaptability extends to various data processing workloads, including batch processing, real-time streaming, interactive queries, and iterative machine learning. Its resilience is upheld by lineage information, ensuring fault tolerance in case of node failures. Moreover, Spark seamlessly integrates with other prominent big data tools and maintains a vibrant open-source community, further enhancing its capabilities and solidifying its place as a fundamental component in the big data landscape, benefiting industries such as finance, healthcare, e-commerce, and more.

2. **PySpark**
   PySpark is a crucial component in the realm of big data analytics. It's an open-source Python library that facilitates seamless integration with Apache Spark, a powerful distributed data processing framework. PySpark allows data engineers, data scientists, and analysts to harness the capabilities of

Spark while using the Python programming language, renowned for its simplicity and extensive ecosystem of data science and machine learning libraries.

With PySpark, users can efficiently process massive datasets, leverage distributed computing, and perform various data operations, including ETL (Extract, Transform, Load), data wrangling, machine learning, graph processing, and real-time streaming analytics. Its Pythonic interface simplifies the development of complex data workflows, making it accessible to a broader audience.

One of PySpark's key advantages is its ability to distribute data processing tasks across a cluster of machines. This distributed computing capability ensures scalability and high performance, making it suitable for handling large-scale data processing workloads in industries like finance, healthcare, e-commerce, and more. Additionally, PySpark seamlessly integrates with popular Python libraries like Pandas, NumPy, Matplotlib, and scikit-learn, enabling users to combine the strengths of Spark's distributed processing with Python's rich data science ecosystem.

In summary, PySpark is a valuable tool in the world of big data, bridging the gap between the simplicity of Python and the processing power of Apache Spark. It empowers data professionals to tackle complex data challenges and unlock insights from vast datasets efficiently and effectively.

3. RDD

RDD stands for "Resilient Distributed Dataset." RDD is a fundamental abstraction in Spark and is central to its data processing model.
RDD is a distributed collection of data elements that can be processed in parallel across a cluster of machines. These data elements can be anything from simple numbers and strings to more complex objects. RDDs are designed to be fault-tolerant, which means they can recover from node failures in a distributed computing environment.

Key characteristics and features of RDDs in big data include:

Immutability: RDDs are immutable, which means their data cannot be changed once they are created. Instead, you create new RDDs through transformations on existing ones.

Partitioning: RDDs are divided into partitions, which are the basic units of parallelism. Each partition can be processed on a separate node in the cluster, allowing for efficient distributed processing.

Resilience: RDDs are resilient to node failures. Spark keeps track of the transformations applied to the base data to rebuild lost partitions in case of a node failure. This lineage information enables fault tolerance.

In-Memory Processing: RDDs can be cached in memory, allowing for faster data access and iterative processing. This in-memory capability significantly speeds up data processing compared to disk-based processing.

Laziness: RDDs follow a lazy evaluation model. Transformations on RDDs are not executed immediately but are recorded and executed only when an action is called. This optimization helps minimize unnecessary data processing.

RDDs provide a level of abstraction that simplifies distributed data processing and makes it more accessible to developers. They serve as the foundation for Spark's high-level APIs, such as DataFrames and Datasets, which provide even more structured and optimized ways to work with data in Spark.

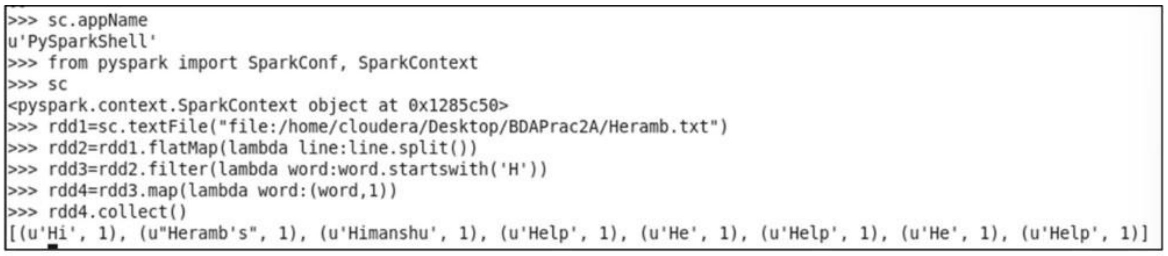
In summary, RDDs in big data, especially in the context of Apache Spark, are distributed and fault-tolerant collections of data that can be processed in parallel across a cluster. They are key to Spark's ability to efficiently handle large-scale data processing tasks while ensuring reliability and fault tolerance.

Program:

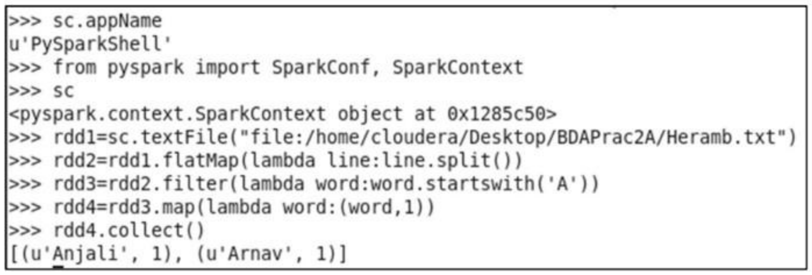
First Type 'pyspark' in the terminal then type the below commands.

```
>>> sc.appName

u'PySparkShell'

>>>from pyspark import SparkConf, SparkContext

>>> sc

<pyspark.context.SparkContext object at 0x2918c50>

>>> rdd1=sc.textFile("file:/home/cloudera/RT/data1.txt")

>>> rdd2=rdd1.flatMap(lambda line:line.split())

>>> rdd3=rdd2.filter(lambda word:word.startswith('h'))

>>> rdd4=rdd3.map(lambda word:(word,1))

>>> rdd4.collect
```

Output:



```
>>> sc.appName
u'PySparkShell'
>>> from pyspark import SparkConf, SparkContext
>>> sc
<pyspark.context.SparkContext object at 0x1285c50>
>>> rdd1=sc.textFile("file:/home/cloudera/Desktop/BDAPrac2A/Heramb.txt")
>>> rdd2=rdd1.flatMap(lambda line:line.split())
>>> rdd3=rdd2.filter(lambda word:word.startswith('H'))
>>> rdd4=rdd3.map(lambda word:(word,1))
>>> rdd4.collect()
[(u'Hi', 1), (u"Heramb's", 1), (u'Himanshu', 1), (u'Help', 1), (u'He', 1), (u'Help', 1), (u'He', 1), (u'Help', 1)]
```

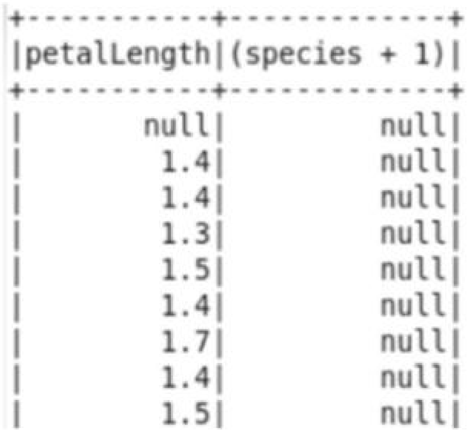


```
>>> sc.appName
u'PySparkShell'
>>> from pyspark import SparkConf, SparkContext
>>> sc
<pyspark.context.SparkContext object at 0x1285c50>
>>> rdd1=sc.textFile("file:/home/cloudera/Desktop/BDAPrac2A/Heramb.txt")
>>> rdd2=rdd1.flatMap(lambda line:line.split())
>>> rdd3=rdd2.filter(lambda word:word.startswith('A'))
>>> rdd4=rdd3.map(lambda word:(word,1))
>>> rdd4.collect()
[(u'Anjali', 1), (u'Arnav', 1)]
```

Program + Output RDD Programs

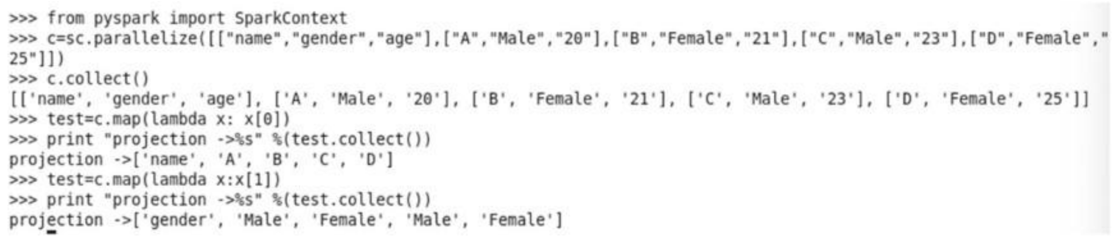
A. Selection

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
df = sqlContext.read.json("/user/cloudera/iris.json")
df.show()
df.select("species").show()
df.select(df['petalLength'], df['species'] + 1).show()
```



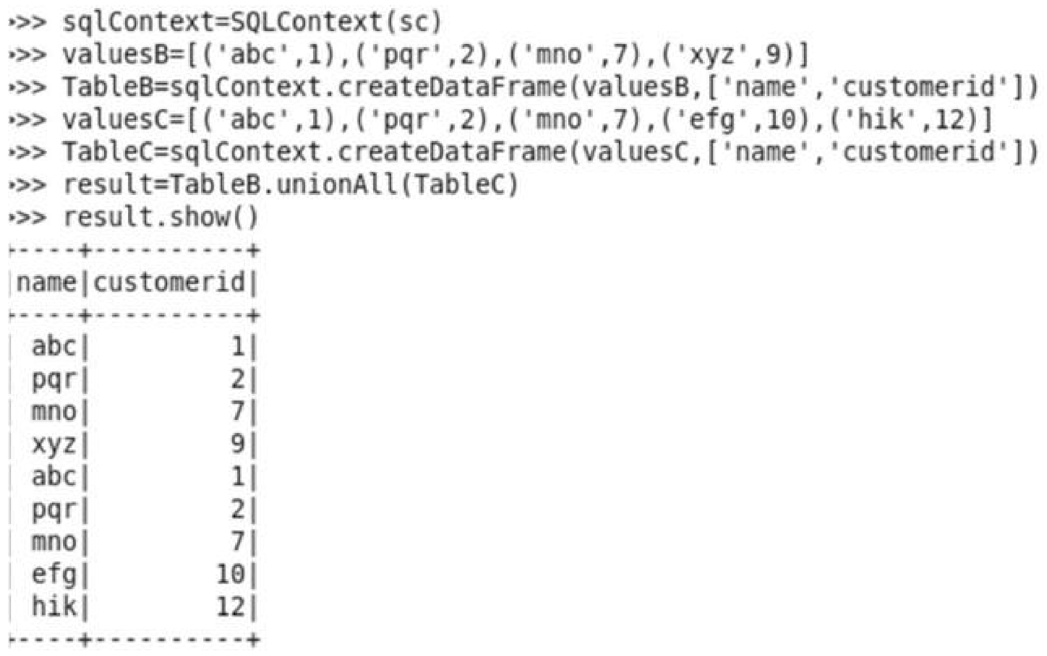
```
+-----------+-------------+
|petalLength|(species + 1)|
+-----------+-------------+
|       null|         null|
|        1.4|         null|
|        1.4|         null|
|        1.3|         null|
|        1.5|         null|
|        1.4|         null|
|        1.7|         null|
|        1.4|         null|
|        1.5|         null|
```

B. Projection



```
>>> from pyspark import SparkContext
>>> c=sc.parallelize([["name","gender","age"],["A","Male","20"],["B","Female","21"],["C","Male","23"],["D","Female","
25"]])
>>> c.collect()
[['name', 'gender', 'age'], ['A', 'Male', '20'], ['B', 'Female', '21'], ['C', 'Male', '23'], ['D', 'Female', '25']]
>>> test=c.map(lambda x: x[0])
>>> print "projection ->%s" %(test.collect())
projection ->['name', 'A', 'B', 'C', 'D']
>>> test=c.map(lambda x:x[1])
>>> print "projection ->%s" %(test.collect())
projection ->['gender', 'Male', 'Female', 'Male', 'Female']
```

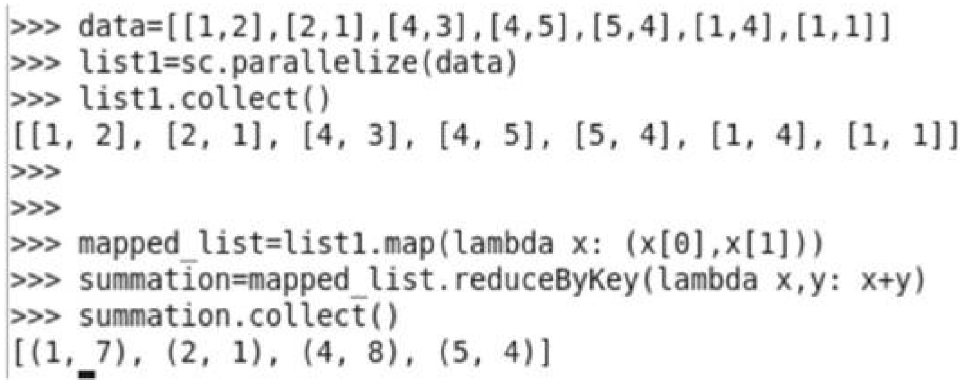
C. Union



```
>>> sqlContext=SQLContext(sc)
>>> valuesB=[('abc',1),('pqr',2),('mno',7),('xyz',9)]
>>> TableB=sqlContext.createDataFrame(valuesB,['name','customerid'])
>>> valuesC=[('abc',1),('pqr',2),('mno',7),('efg',10),('hik',12)]
>>> TableC=sqlContext.createDataFrame(valuesC,['name','customerid'])
>>> result=TableB.unionAll(TableC)
>>> result.show()
+----+----------+
|name|customerid|
+----+----------+
| abc|         1|
| pqr|         2|
| mno|         7|
| xyz|         9|
| abc|         1|
| pqr|         2|
| mno|         7|
| efg|        10|
| hik|        12|
+----+----------+
```

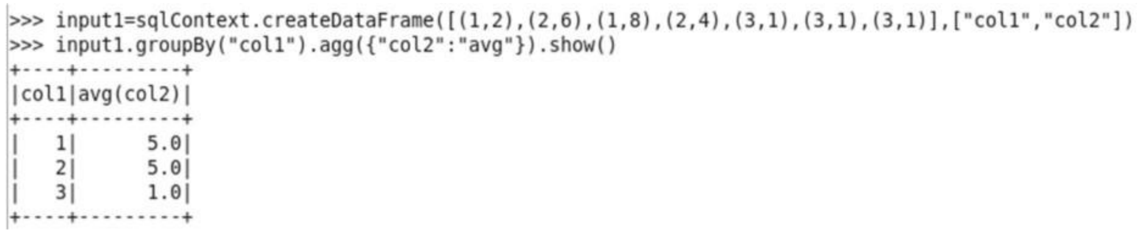
D. Aggregate and Grouping

Sum:

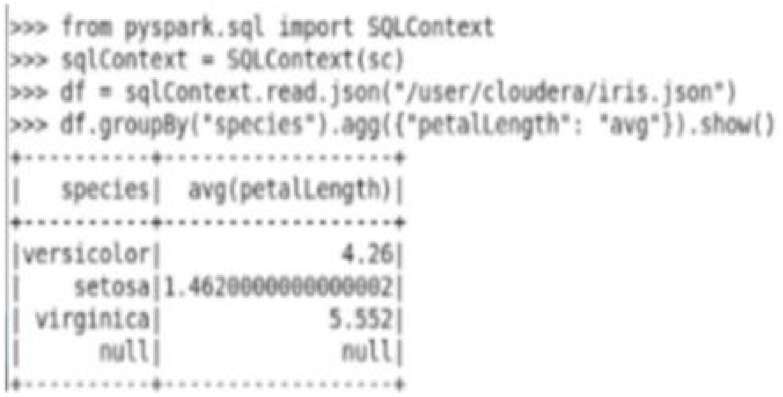


```
>>> data=[[1,2],[2,1],[4,3],[4,5],[5,4],[1,4],[1,1]]
>>> list1=sc.parallelize(data)
>>> list1.collect()
[[1, 2], [2, 1], [4, 3], [4, 5], [5, 4], [1, 4], [1, 1]]
>>>
>>>
>>> mapped_list=list1.map(lambda x: (x[0],x[1]))
>>> summation=mapped_list.reduceByKey(lambda x,y: x+y)
>>> summation.collect()
[(1, 7), (2, 1), (4, 8), (5, 4)]
```

Average:

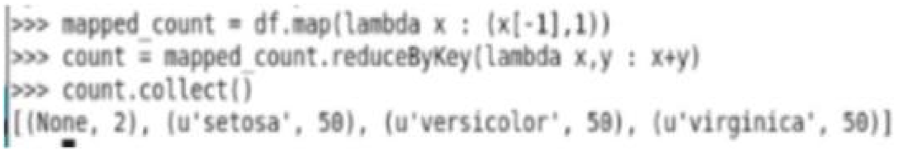


```
>>> input1=sqlContext.createDataFrame([(1,2),(2,6),(1,8),(2,4),(3,1),(3,1),(3,1)],["col1","col2"])
>>> input1.groupBy("col1").agg({"col2":"avg"}).show()
+----+---------+
|col1|avg(col2)|
+----+---------+
|   1|      5.0|
|   2|      5.0|
|   3|      1.0|
+----+---------+
```



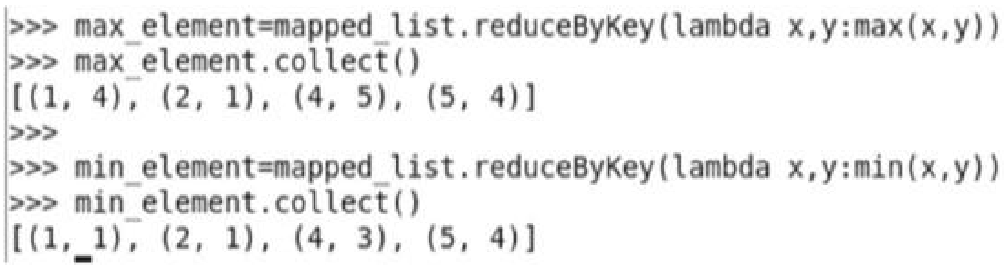
```
>>> from pyspark.sql import SQLContext
>>> sqlContext = SQLContext(sc)
>>> df = sqlContext.read.json("/user/cloudera/iris.json")
>>> df.groupBy("species").agg({"petalLength": "avg"}).show()
+----------+------------------+
|   species|  avg(petalLength)|
+----------+------------------+
|versicolor|              4.26|
|    setosa|1.4620000000000002|
| virginica|             5.552|
|      null|              null|
+----------+------------------+
```

Count:



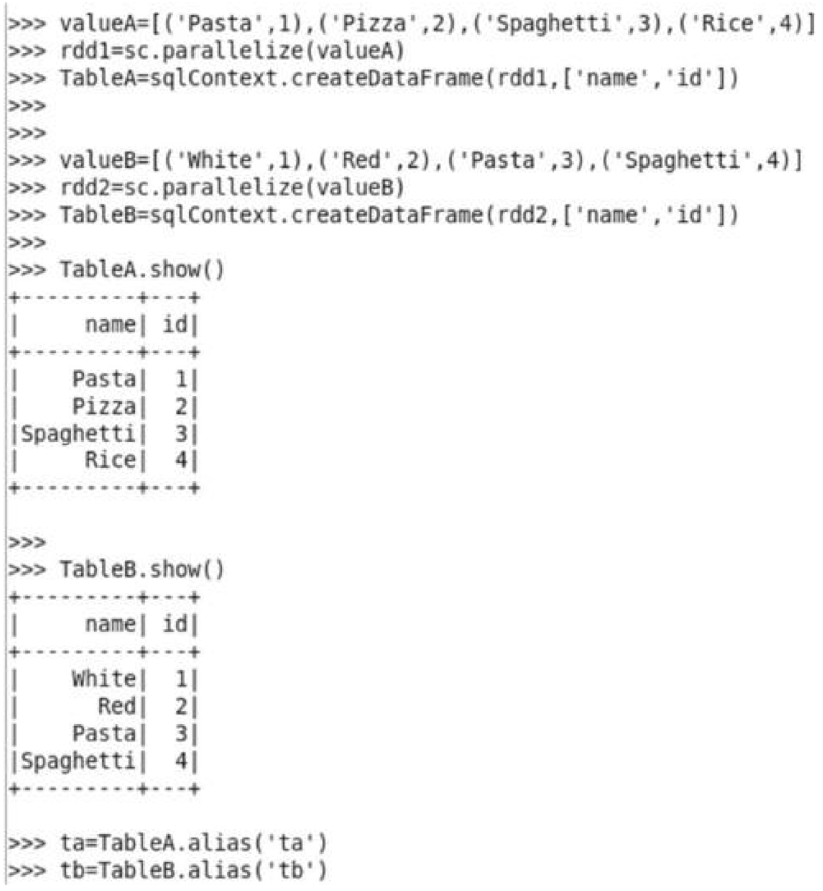
```
>>> mapped_count = df.map(lambda x : (x[-1],1))
>>> count = mapped_count.reduceByKey(lambda x,y : x+y)
>>> count.collect()
[(None, 2), (u'setosa', 50), (u'versicolor', 50), (u'virginica', 50)]
```

Max & Min element

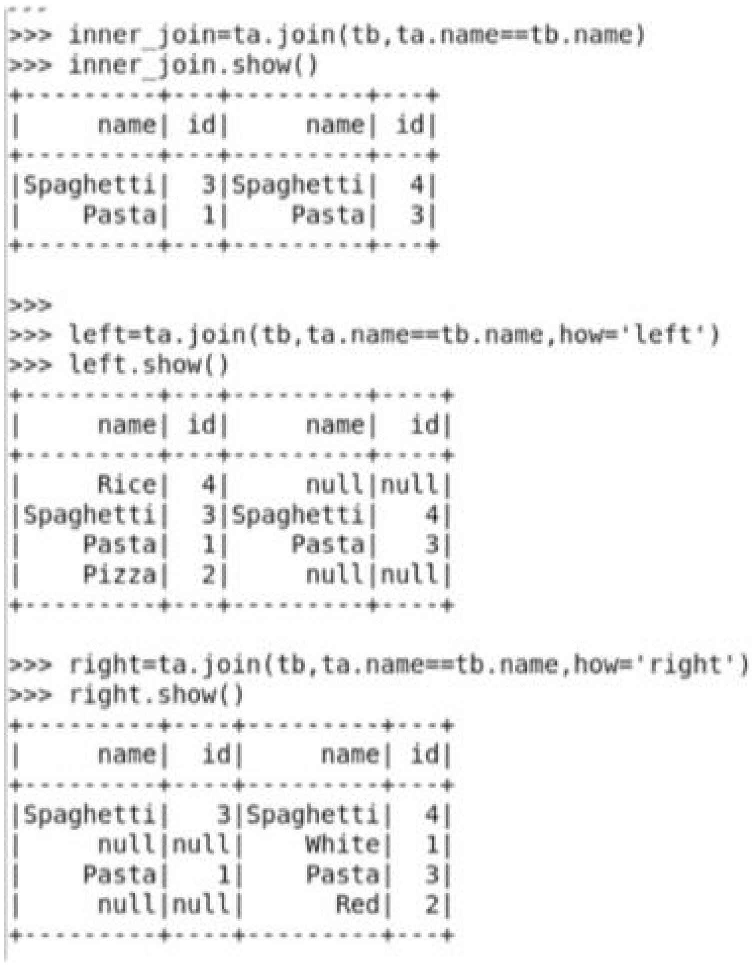


```
>>> max_element=mapped_list.reduceByKey(lambda x,y:max(x,y))
>>> max_element.collect()
[(1, 4), (2, 1), (4, 5), (5, 4)]
>>>
>>> min_element=mapped_list.reduceByKey(lambda x,y:min(x,y))
>>> min_element.collect()
[(1, 1), (2, 1), (4, 3), (5, 4)]
```

E. Join

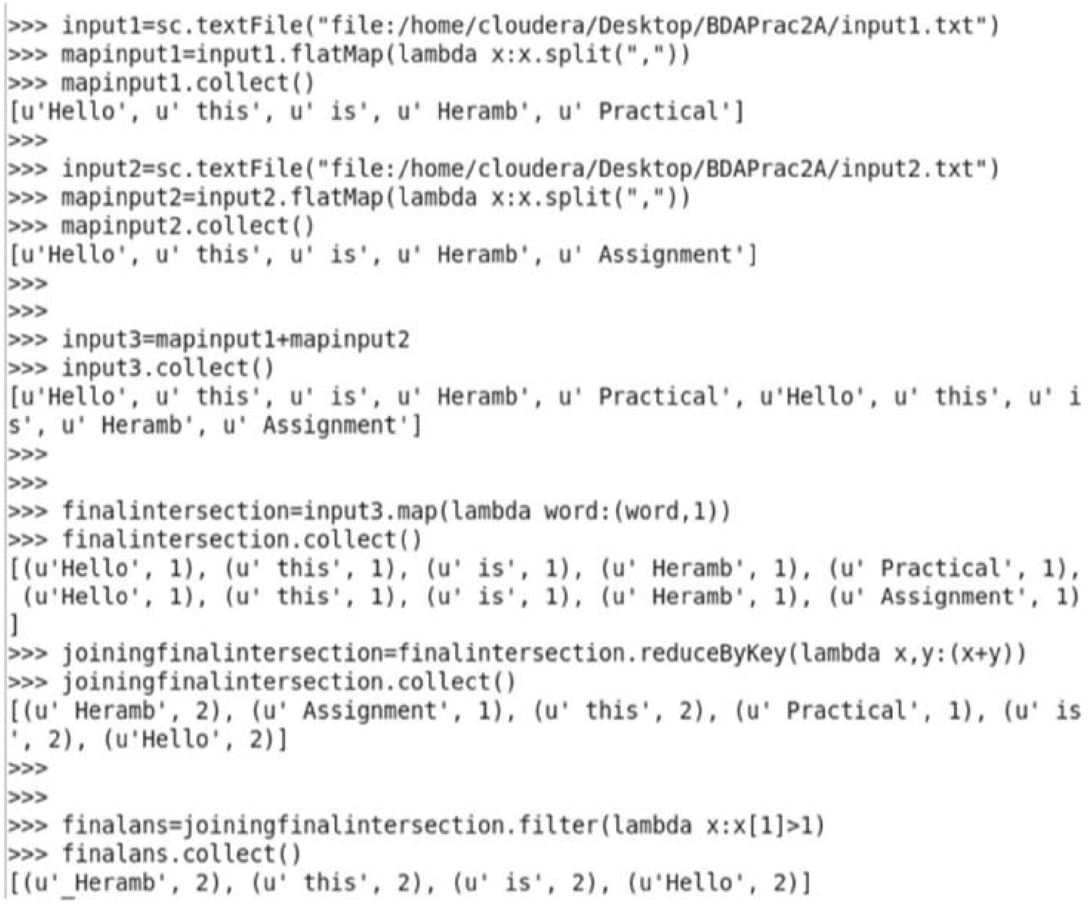


```
>>> valueA=[('Pasta',1),('Pizza',2),('Spaghetti',3),('Rice',4)]
>>> rdd1=sc.parallelize(valueA)
>>> TableA=sqlContext.createDataFrame(rdd1,['name','id'])
>>>
>>>
>>> valueB=[('White',1),('Red',2),('Pasta',3),('Spaghetti',4)]
>>> rdd2=sc.parallelize(valueB)
>>> TableB=sqlContext.createDataFrame(rdd2,['name','id'])
>>>
>>> TableA.show()
+---------+---+
|     name| id|
+---------+---+
|    Pasta|  1|
|    Pizza|  2|
|Spaghetti|  3|
|     Rice|  4|
+---------+---+

>>>
>>> TableB.show()
+---------+---+
|     name| id|
+---------+---+
|    White|  1|
|      Red|  2|
|    Pasta|  3|
|Spaghetti|  4|
+---------+---+

>>> ta=TableA.alias('ta')
>>> tb=TableB.alias('tb')
```



```
...
>>> inner_join=ta.join(tb,ta.name==tb.name)
>>> inner_join.show()
+---------+---+---------+---+
|     name| id|     name| id|
+---------+---+---------+---+
|Spaghetti|  3|Spaghetti|  4|
|    Pasta|  1|    Pasta|  3|
+---------+---+---------+---+

>>>
>>> left=ta.join(tb,ta.name==tb.name,how='left')
>>> left.show()
+---------+---+---------+----+
|     name| id|     name|  id|
+---------+---+---------+----+
|     Rice|  4|     null|null|
|Spaghetti|  3|Spaghetti|   4|
|    Pasta|  1|    Pasta|   3|
|    Pizza|  2|     null|null|
+---------+---+---------+----+

>>> right=ta.join(tb,ta.name==tb.name,how='right')
>>> right.show()
+---------+----+---------+---+
|     name|  id|     name| id|
+---------+----+---------+---+
|Spaghetti|   3|Spaghetti|  4|
|     null|null|    White|  1|
|    Pasta|   1|    Pasta|  3|
|     null|null|      Red|  2|
+---------+----+---------+---+
```

F. Intersection



```
>>> input1=sc.textFile("file:/home/cloudera/Desktop/BDAPrac2A/input1.txt")
>>> mapinput1=input1.flatMap(lambda x:x.split(","))
>>> mapinput1.collect()
[u'Hello', u' this', u' is', u' Heramb', u' Practical']
>>>
>>> input2=sc.textFile("file:/home/cloudera/Desktop/BDAPrac2A/input2.txt")
>>> mapinput2=input2.flatMap(lambda x:x.split(","))
>>> mapinput2.collect()
[u'Hello', u' this', u' is', u' Heramb', u' Assignment']
>>>
>>>
>>> input3=mapinput1+mapinput2
>>> input3.collect()
[u'Hello', u' this', u' is', u' Heramb', u' Practical', u'Hello', u' this', u' i
s', u' Heramb', u' Assignment']
>>>
>>>
>>> finalintersection=input3.map(lambda word:(word,1))
>>> finalintersection.collect()
[(u'Hello', 1), (u' this', 1), (u' is', 1), (u' Heramb', 1), (u' Practical', 1),
 (u'Hello', 1), (u' this', 1), (u' is', 1), (u' Heramb', 1), (u' Assignment', 1)
]
>>> joiningfinalintersection=finalintersection.reduceByKey(lambda x,y:(x+y))
>>> joiningfinalintersection.collect()
[(u' Heramb', 2), (u' Assignment', 1), (u' this', 2), (u' Practical', 1), (u' is
', 2), (u'Hello', 2)]
>>>
>>>
>>> finalans=joiningfinalintersection.filter(lambda x:x[1]>1)
>>> finalans.collect()
[(u' Heramb', 2), (u' this', 2), (u' is', 2), (u'Hello', 2)]
```

Results and Discussions:

In the realm of text analysis, the Pyspark program for counting words starting with a specific letter is a useful asset for discerning textual patterns. It yields the count of such words, pivotal for applications like sentiment analysis and content categorization. This tool simplifies the analysis of extensive text data offering objective insights.

RDBMS operations emulated through Pyspark each, operation serves a unique purpose in data manipulation, Selection extracts pertinent data, projection does the analysis by retaining chosen attributes, Union merges dataset, Intersection identifies shared elements e.t.c. These operations showcase pyspark's prowess in data management and analysis

To conclude, Pyspark programs cater to text analysis and data manipulation needs effectively. They mirror essential RDBMS actions and harness Pyspark's distributed computing. Real World applications span sentiment analysis to integration, with scalability and optimization as important considerations.