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: Akshat Tiwari	**Class/Roll No.:** D16AD/62	**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 this 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 (RDBMS) operations, including Selection, Projection, Union, Aggregates with grouping, Joins, and Intersection, 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 (e.g., 'h' or 'a') in a given text dataset.

b. Implement key Relational Database Management System (RDBMS) operations using PySpark, including Selection, Projection, Union, Aggregates with grouping, Joins, and Intersection.

Description / Theory:

1. **Spark:**
   Apache Spark (Software) - A Data Processing Framework: Apache Spark is an open-source, distributed computing system that provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. It's designed to handle big data processing and analytics workloads. Spark supports various programming languages (Java, Scala, Python, R) and provides libraries for data processing, machine learning, graph processing, and more.

   Spark's key features include in-memory processing, which makes it much faster than traditional MapReduce-based systems for certain types of workloads, and a wide range of built-in libraries for machine learning (MLlib), graph processing (GraphX), SQL queries (Spark SQL), and stream processing (Spark Streaming).

   Spark can be used for tasks like data transformation, batch processing, real-time stream processing, iterative machine learning algorithms, and more. It's often employed in big data environments for handling large-scale data analytics tasks efficiently.

2. **PySpark:**

   PySpark is the Python library for Apache Spark, an open-source distributed computing system for big data processing and analytics. PySpark allows you to use Spark's capabilities and features through the Python programming language. It provides a Python API that enables developers to write Spark applications using familiar Python syntax, which can be especially beneficial for those who are more comfortable with Python programming.

   Key features of PySpark include:

   - **Python API:** PySpark allows you to interact with Spark using Python code. This is useful for data engineers, data scientists, and analysts who are proficient in Python and want to leverage Spark's capabilities without having to learn Scala or Java.

- **Data Processing:** You can use PySpark to perform various data processing tasks, such as transforming and cleaning large datasets, aggregating data, filtering data, and more.

- **Spark Libraries:** PySpark provides access to Spark's built-in libraries, such as MLlib for machine learning, GraphX for graph processing, and Spark SQL for querying structured data using SQL-like syntax.

- **Parallel Processing:** PySpark takes advantage of Spark's distributed computing capabilities, allowing you to process data in parallel across a cluster of machines.

- **In-Memory Processing:** Like Spark, PySpark also supports in-memory processing, which speeds up data processing by keeping frequently accessed data in memory.

- **Interactive Analysis:** PySpark can be used interactively, similar to working with the Python interpreter, allowing you to explore and analyze data in real time.

3. RDD:

RDD stands for Resilient Distributed Dataset. It's a fundamental data structure in Apache Spark, designed to handle and process data in a distributed and fault-tolerant manner. RDDs provide a high-level abstraction over distributed data and enable efficient parallel processing across a cluster of machines.

Here are the key characteristics and concepts related to RDDs:

- **Resilient:** RDDs are designed to be fault-tolerant. They automatically recover from node failures by recomputing lost partitions of data using the lineage information (transformations that were applied to the original data).

- **Distributed:** RDDs represent data that is distributed across multiple nodes in a cluster. This enables parallel processing, where operations can be performed on different parts of the data simultaneously.

- **Dataset Abstraction:** RDDs provide a logical representation of distributed data, abstracting away the complexities of managing data distribution and parallel processing. This makes it easier for developers to work with large-scale datasets.

- **Immutable:** RDDs are immutable, meaning their contents cannot be changed once they are created. Instead, transformations on RDDs create new RDDs with the desired changes.

- **Lazy Evaluation:** Transformations on RDDs are evaluated lazily. This means that operations are not executed immediately when called, but rather they build up a computation plan (or lineage). The actual computation is triggered when an action is performed.

- **Lineage:** RDDs keep track of the sequence of transformations that were applied to create them. This lineage information is crucial for fault recovery. If a partition of an RDD is lost, Spark can recompute it using the original data and the sequence of transformations.

- **Wide and Narrow Transformations:** Transformations on RDDs are categorized as narrow or wide. Narrow transformations (e.g., map, filter) do not require shuffling or data movement between partitions, while wide transformations (e.g., groupByKey, join) involve shuffling and redistribution of data.

- **Actions:** Actions are operations that trigger the execution of transformations and return results to the driver program or write data to external storage. Examples of actions include count, collect, saveAsTextFile, etc.

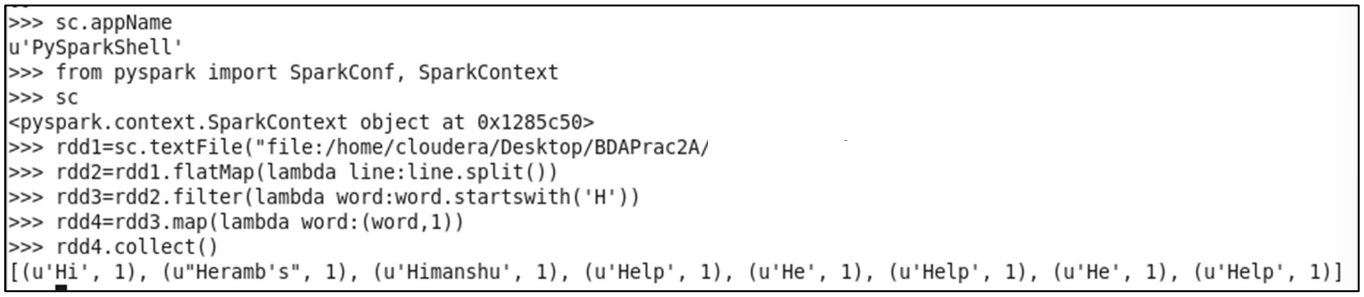
- **Caching:** RDDs can be cached in memory to improve performance for iterative algorithms or frequently used data. This reduces the need to recompute transformations when the same data is accessed multiple time

Program:

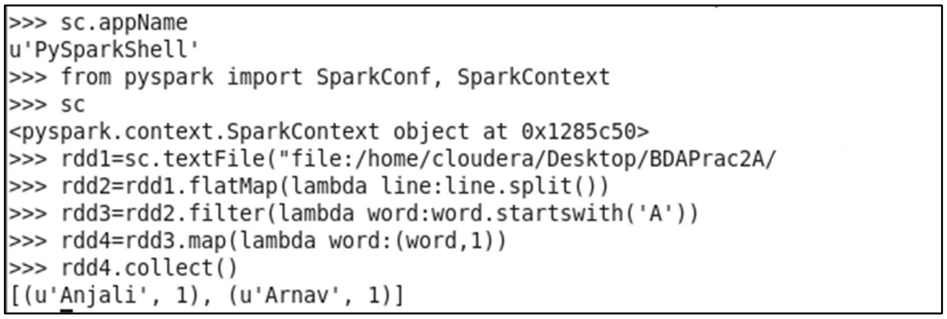
First Type 'pyspark' in the terminal then type the below commands.

```
>>> 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/
>>> 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)]
```

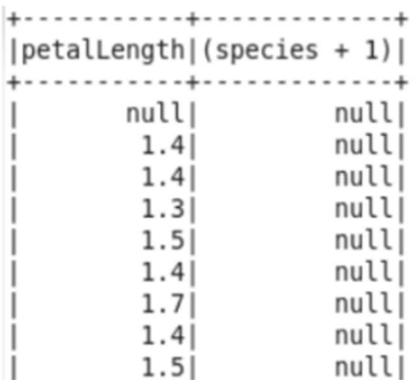


```
>>> sc.appName
u'PySparkShell'
>>> from pyspark import SparkConf, SparkContext
>>> sc
<pyspark.context.SparkContext object at 0x1285c50>
>>> rdd1=sc.textFile("file:/home/cloudera/Desktop/BDAPrac2A/
>>> rdd2=rdd1.flatMap(lambda line:line.split())
>>> rdd3=rdd2.filter(lambda word:word.startswith('A'))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

Program + Output: RDD Programs

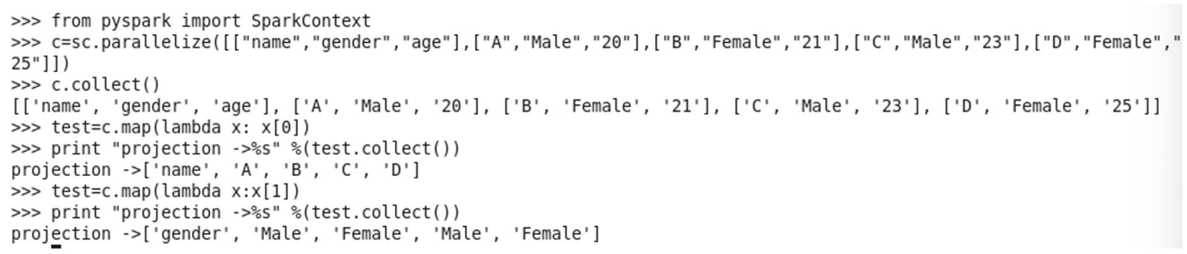
A. Selection

```
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()
```



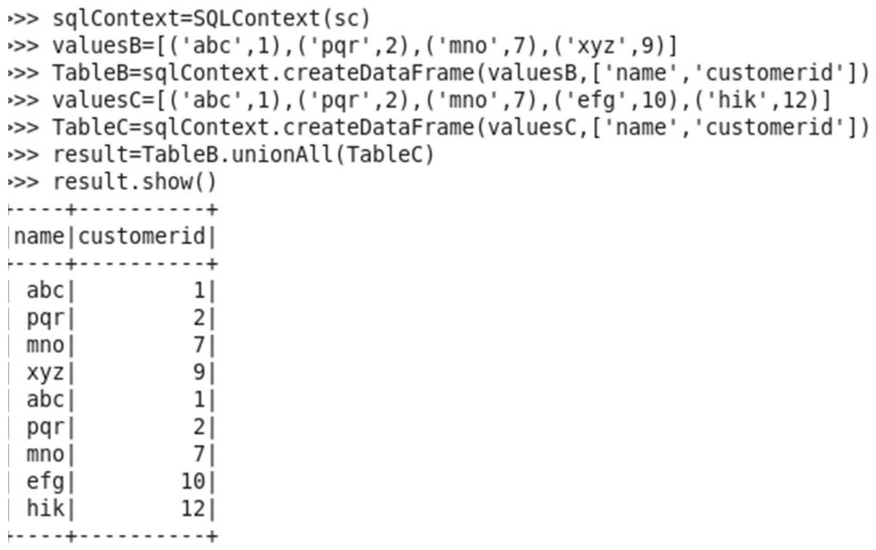
```
+-----------+-------------+
|petalLength|(species + 1)|
+-----------+-------------+
|       null|         null|
|        1.4|         null|
|        1.4|         null|
|        1.3|         null|
|        1.5|         null|
|        1.4|         null|
|        1.7|         null|
|        1.4|   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

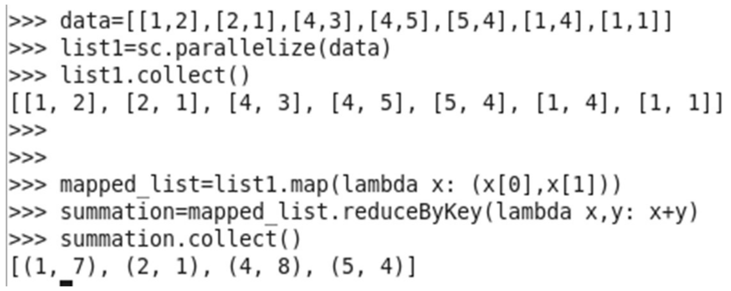
C. Union



```
>>> 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|
+----+----------+
```

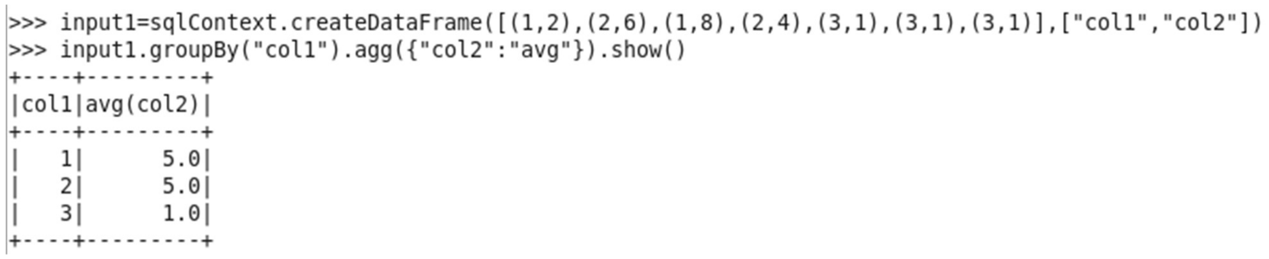
D. Aggregate And Grouping

Sum:

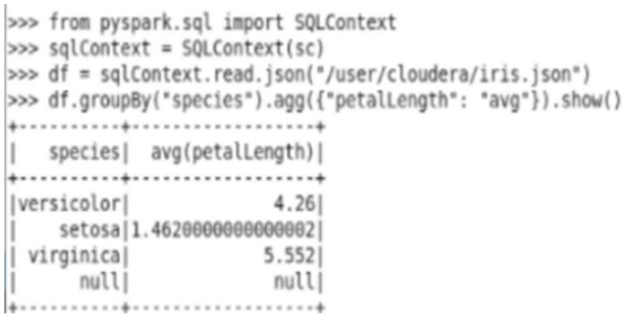


```
>>> data=[[1,2],[2,1],[4,3],[4,5],[5,4],[1,4],[1,1]]
>>> list1=sc.parallelize(data)
>>> list1.collect()
[[1, 2], [2, 1], [4, 3], [4, 5], [5, 4], [1, 4], [1, 1]]
>>>
>>>
>>> mapped_list=list1.map(lambda x: (x[0],x[1]))
>>> summation=mapped_list.reduceByKey(lambda x,y: x+y)
>>> summation.collect()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

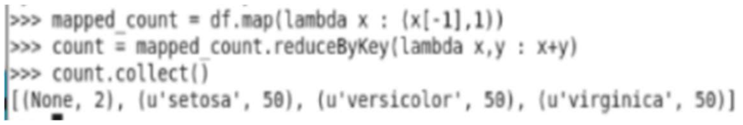
Average:



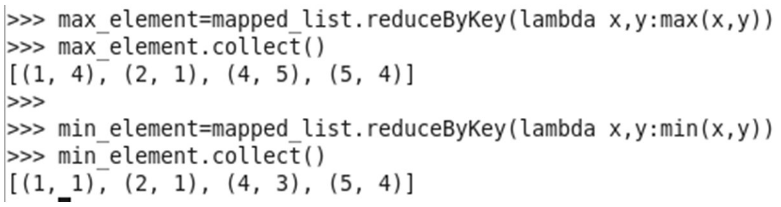
```
>>> input1=sqlContext.createDataFrame([(1,2),(2,6),(1,8),(2,4),(3,1),(3,1),(3,1)],["col1","col2"])
>>> input1.groupBy("col1").agg({"col2":"avg"}).show()
+----+---------+
|col1|avg(col2)|
+----+---------+
|   1|      5.0|
|   2|      5.0|
|   3|      1.0|
+----+---------+
```


```
>>> 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|
+----------+-----------------+
```

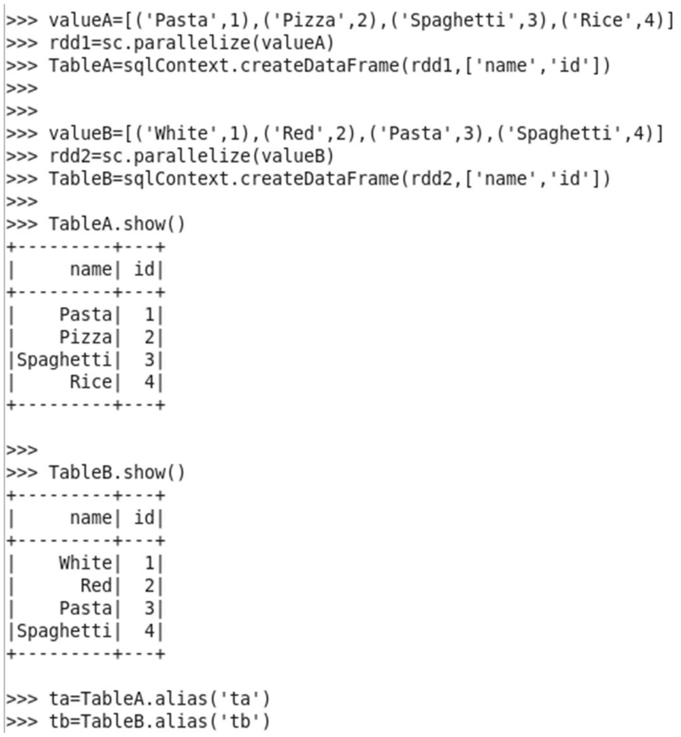
Count

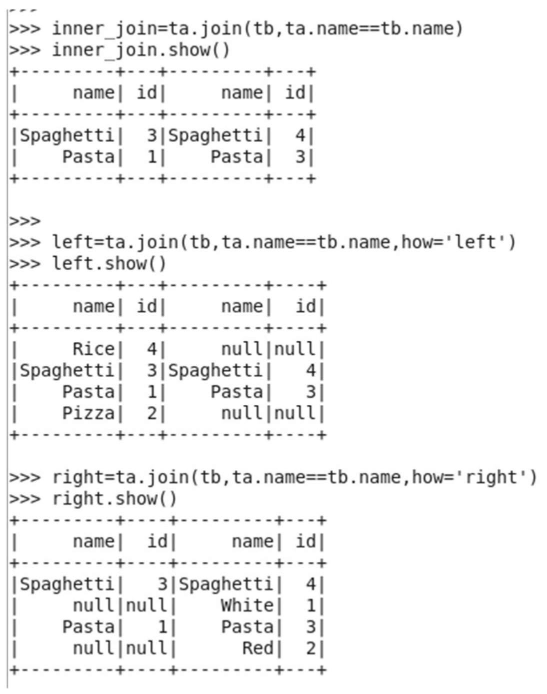

```
>>> mapped_count = df.map(lambda x : (x[-1],1))
>>> count = mapped_count.reduceByKey(lambda x,y : x+y)
>>> count.collect()
[(None, 2), (u'setosa', 50), (u'versicolor', 50), (u'virginica', 50)]
```

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)]
```

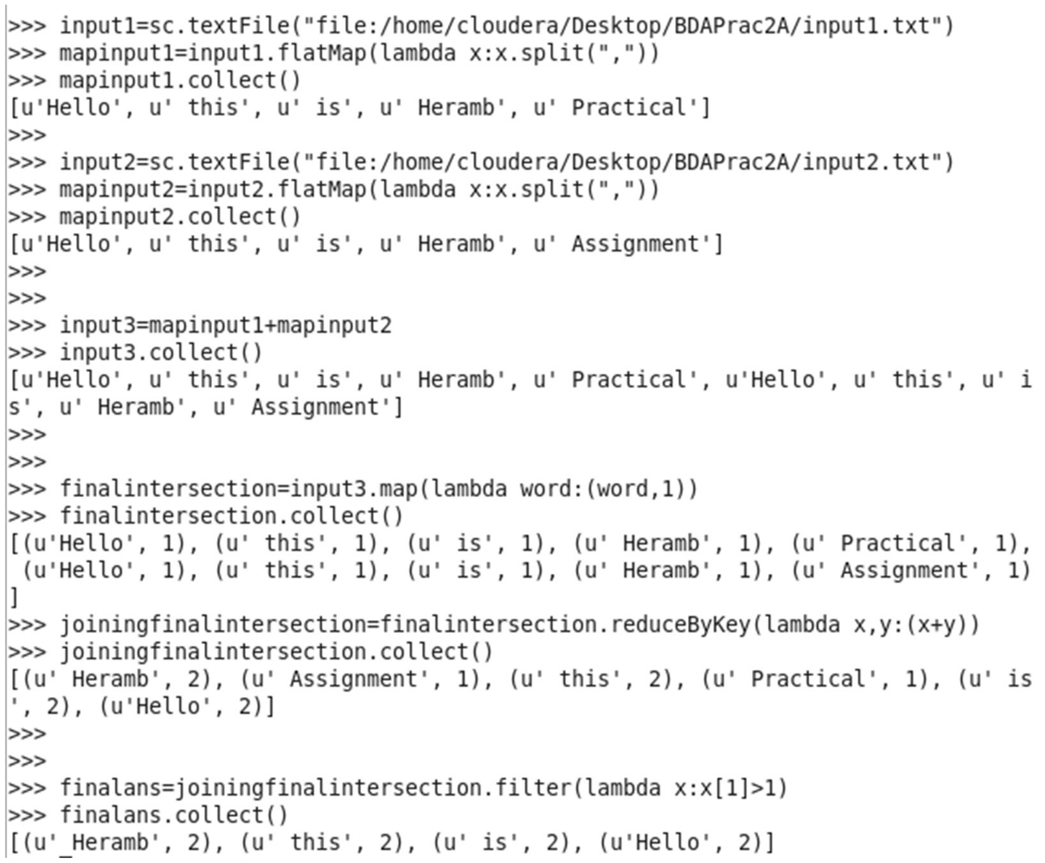
E. Join


```
>>> 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.

Shifting focus to the RDBMS operations emulated through PySpark, each operation serves a unique purpose in data manipulation. Selection extracts pertinent data, Projection simplifies the analysis by retaining chosen attributes, and Union merges datasets seamlessly. Aggregates and grouping unveil trends like average age, while Joins combine datasets based on shared attributes. Lastly, Intersection identifies shared elements. These operations showcase PySpark's prowess in data management and analysis.

To conclude, the 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.