

Pig:

I installed Pig from the provided Apache link. Path is set in bashrc. To work in local mode 'pig -x local' command is given, and coding is to be done within that pig environment with grunt prompt.

```
soumya@soumya-VirtualBox:~/Desktop$ pig -x local
17/10/18 13:44:04 INFO pig.ExecTypeProvider: Trying ExecType : LOCAL
17/10/18 13:44:04 INFO pig.ExecTypeProvider: Picked LOCAL as the ExecType
2017-10-18 13:44:05,529 [main] INFO org.apache.pig.Main - Apache Pig version 0.16.0 (r1746530) compiled Jun 01 2016, 23:09:59
2017-10-18 13:44:05,529 [main] INFO org.apache.pig.Main - Logging error message s to: /home/soumya/Desktop/pig_1508348645526.log
2017-10-18 13:44:05,650 [main] INFO org.apache.pig.impl.util.Utils - Default bootstrap file /home/soumya/.pigbootstrap not found
2017-10-18 13:44:06,621 [main] INFO org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to hadoop file system at: file:///
2017-10-18 13:44:06,810 [main] INFO org.apache.pig.PigServer - Pig Script ID for the session:
2017-10-18 13:44:06,812 [main] WARN org.apache.pig.PigServer - ATS is disabled since yarn.timeline-service.enabled set to false
grunt>
```

Spark:

Spark is installed, and configuration is setup in spark-env.sh. Start the master and slaves.

'./bin/pyspark' command takes us to the spark shell.

```
soumya@soumya-VirtualBox: /usr/local/spark
soumya@soumya-VirtualBox: /usr/local/spark$ ./bin/pyspark
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
17/10/21 01:53:38 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
17/10/21 01:53:38 WARN Utils: Your hostname, soumya-VirtualBox resolves to a loopback address: 127.0.1.1; using 10.0.2.15 instead (on interface enp0s3)
17/10/21 01:53:38 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
17/10/21 01:53:52 WARN ObjectStore: Failed to get database global_temp, returning NoSuchObjectException
Welcome to

  ____      _
 / ___|  _ \| | | |
 \___ \ |_) | |_| |
  ___) |  __/| | | |
 |_____|_|_||_| |_|

 version 2.2.0

Using Python version 2.7.12 (default, Nov 19 2016 06:48:10)
SparkSession available as 'spark'.
>>>
```

PART-I

(1.1)

```
A = load 'input.txt';
```

Input file is loaded into the grunt shell, working in the same directory where the input file is present. If working in a different directory, the path for the input file has to be given. It is stored in A.

```
B = foreach A generate flatten(TOKENIZE((chararray)$0)) as token;
```

The output of A is de-nested by FLATTEN operator and tokenized into multiple elements from one tuple and returned them as tokens. They are stored in B.

```
C = group B by token;
```

It makes same tokens as a group. They are stored in C.

```
D = foreach C generate group, COUNT(B);
```

For each group generated by the previous command, number of tokens per group are counted and is stored in D.

```
store D into './output';
```

The table generated for the operation D is stored as a text file in output directory.

(1.2)

```
lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])
```

It reads the data from the argument, argument can be a path of the input file or the input file name itself if working in the present directory, turns it into lines with rdd and stores the first element in it.

```
A = lines.flatMap(lambda x: x.split(' '))
```

It splits the data and returns a list of values which are stored in x. That bag of elements is stored in A.

```
B = A.map(lambda x: (int(x), 1))
```

Each element is given an int with count 1 with map. Map can be used to implement key count. It can be completely implemented in this case if reduceByKey is used.

```
C = B.sortByKey()
```

Keys are sorted in any specified order. In this case, if nothing is mentioned it sorts in ascending order.

PART-II

(2.1) Pig

The input files are present on my desktop, so I worked in the local mode of Pig in the desktop directory. I loaded the purchase.json file as an input for the program. Label the columns of the table with a name and data type how you want them to be referred. Load this into *Records* as following.

```
Records = LOAD 'purchase.json' AS (year:chararray, cid:chararray, isbn:chararray, seller: chararray, price:int);
```

```
grunt> Records = LOAD 'purchase.txt' AS (year:chararray, cid:chararray, isbn:chararray, seller:chararray, price:int);  
grunt> DUMP Records
```

Dump Records prints the data stored in the Records as following.

```
(1999,C1,B1,Amazon,90)  
(2001,C1,B2,Amazon,20)  
(2008,C2,B2,Barnes Noble,30)  
(2008,C3,B3,Amazon,28)  
(2009,C2,B1,Borders,90)  
(2010,C4,B3,Barnes Noble,26)  
grunt>
```

Grouping is done on basis of the same seller.

```
Grouped = GROUP Records BY seller;
```

```
soumya@soumya-VirtualBox: ~/Desktop
Total records proactively spilled: 0

Job DAG:
job_local_0006

2017-10-14 21:48:42,190 [main] INFO org.apache.pig.backend.hadoop.executionengi
ne.mapReduceLayer.MapReduceLauncher - Success!
2017-10-14 21:48:42,191 [main] WARN org.apache.pig.data.SchemaTupleBackend - Sc
hemaTupleBackend has already been initialized
2017-10-14 21:48:42,193 [main] INFO org.apache.hadoop.mapreduce.lib.input.FileI
nputFormat - Total input paths to process : 1
2017-10-14 21:48:42,193 [main] INFO org.apache.pig.backend.hadoop.executionengi
ne.util.MapRedUtil - Total input paths to process : 1
(1999,C1,B1,Amazon,90)
(2001,C1,B2,Amazon,20)
(2008,C2,B2,Barnes Noble,30)
(2008,C3,B3,Amazon,28)
(2009,C2,B1,Borders,90)
(2010,C4,B3,Barnes Noble,26)
grunt> Grouped = GROUP Records BY seller;
grunt> Results = FOREACH Grouped GENERATE group, SUM(Records.price);
grunt> DUMP Results
```

And for each such group, their corresponding prices are to be added.

Results = FOREACH Grouped GENERATE group, SUM(Records.price);

Dump Results gives the following output.

```
Counters:
Total records written : 3
Total bytes written : 0
Spillable Memory Manager spill count : 0
Total bags proactively spilled: 0
Total records proactively spilled: 0

Job DAG:
job_local_0004

2017-10-14 21:24:51,728 [main] INFO org.apache.pig.backend.hadoop.executionengi
ne.mapReduceLayer.MapReduceLauncher - Success!
2017-10-14 21:24:51,729 [main] WARN org.apache.pig.data.SchemaTupleBackend - Sc
hemaTupleBackend has already been initialized
2017-10-14 21:24:51,731 [main] INFO org.apache.hadoop.mapreduce.lib.input.FileI
nputFormat - Total input paths to process : 1
2017-10-14 21:24:51,731 [main] INFO org.apache.pig.backend.hadoop.executionengi
ne.util.MapRedUtil - Total input paths to process : 1
(Amazon,138)
(Borders,90)
(Barnes Noble,56)
grunt>
```

(2.1) Spark RDD

The input file for the Spark is stored in the Spark environment itself. The input file name path is read and loaded into the *readFile* with rdd, turns them into lines and stores the first element.

readFile = *spark.read.text("path").rdd.map(lambda x: x[0])* should be the form.

readFile = *spark.read.text("/usr/local/spark/pr2/purchase.json").rdd.map(lambda x: x[0])*

Split function splits the elements separated by tabs, and stores each row in a bag. All such bags are stored in *divided*.

```
divided = readFile.map(lambda x: x.split('\t'))
```

divided.take(50) prints the content of *divided*.

dividedAgain again maps only the fourth and fifth columns of every row i.e., seller and the price. This makes a bag of (seller,price) tuples

```
dividedAgain = divided.map(lambda x: (x[3], int(x[4])))
```

result reduces the above bag by each key and adds their corresponding key values.

```
result = dividedAgain.reduceByKey(lambda x, y: x + y)
```

result will be the added prices for every seller as shown below.

result.take(50) dumps the result.

```
>>> readFile = spark.read.text("/usr/local/spark/pr2/purchase.json").rdd.map(lambda x: x[0])
>>> divided = readFile.map(lambda x: x.split('\t'))
>>> divided.take(50)
[[u'1999', u'C1', u'B1', u'Amazon', u'90'], [u'2001', u'C1', u'B2', u'Amazon', u'20'], [u'2008', u'C2', u'B2', u'Barnes Noble', u'30'], [u'2008', u'C3', u'B3', u'Amazon', u'28'], [u'2009', u'C2', u'B1', u'Borders', u'90'], [u'2010', u'C4', u'B3', u'Barnes Noble', u'26']]
>>> dividedAgain = divided.map(lambda x: (x[3], int(x[4])))>>> dividedAgain.take(50)
[(u'Amazon', 90), (u'Amazon', 20), (u'Barnes Noble', 30), (u'Amazon', 28), (u'Borders', 90), (u'Barnes Noble', 26)]
>>> result = dividedAgain.reduceByKey(lambda x, y: x + y)
>>> result.take(50)
[(u'Amazon', 138), (u'Barnes Noble', 56), (u'Borders', 90)]
>>>
```

(2.2) Pig

Records loads the data from the text file purchase.json with data types for each column.

```
Records = LOAD 'purchase.json' AS (year:chararray, cid:chararray, isbn:chararray, seller: chararray, price:int);
```

BookRecords loads data from the text file 'book.json' with their corresponding datatypes.

```
BookRecords = LOAD 'book.json' AS (isbn: chararray, name: chararray);
```

Combined joins both the purchase and book tables taking *isbn* as the common column.

```
Combined = JOIN Records BY (isbn), BookRecords BY (isbn);
```

CombinedPars gives the *combined* columns names. It relates to the *Records* column names.

CombinedPars = FOREACH Combined GENERATE Records::year, Records::cid, Records::isbn, Records::Seller, Records::price, BookRecords::name;

DUMP CombinedPars gives the following output.

```
(1999,C1,B1,Amazon,90,Novel)
(2009,C2,B1,Borders,90,Novel)
(2001,C1,B2,Amazon,20,Drama)
(2008,C2,B2,Barnes Noble,30,Drama)
(2008,C3,B3,Amazon,28,Poem)
(2010,C4,B3,Barnes Noble,26,Poem)
grunt>
```

Grouped groups the *combinedPars* by book id, Therefore, for each book id it creates a group.

Grouped = GROUP CombinedPars BY isbn;

Grouped1 finds the minimum price for each *Grouped* groups.

Grouped1 = FOREACH Grouped GENERATE group, MIN(CombinedPars.price);

DUMP Grouped1 gives the following output.

```
ne.util.MapRedUtil - Total input paths to process : 1
(B1,90)
(B2,20)
(B3,26)
grunt>
```

Filtered filters the data in *Records*. It is filtered only the data related to Amazon.

Filtered = FILTER Records BY seller == 'Amazon';

DUMP Filtered gives the following output.

```
ne.util.MapRedUtil - Total input paths to process : 1
(1999,C1,B1,Amazon,90)
(2001,C1,B2,Amazon,20)
(2008,C3,B3,Amazon,28)
grunt>
```

Joined joins the *filtered* and *Grouped1* by price.

Joined = JOIN Filtered by price, Grouped1 BY \$1;

Joined1 joins the output of *Joined* and book names.

Joined1 = JOIN Joined by Filtered::isbn, BookRecords BY isbn;

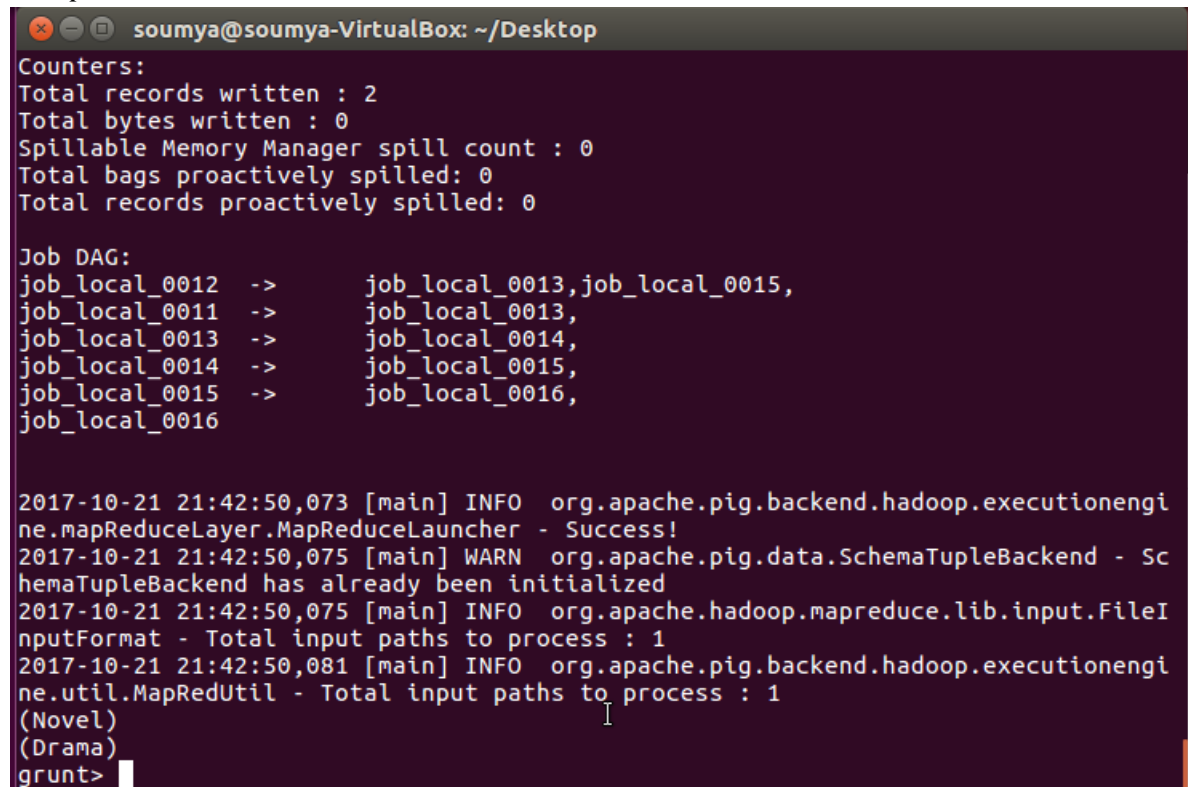
DUMP Joined1 dumps the following.

```
ne.util.MapRedUtil - Total input paths to process : 1
(1999,C1,B1,Amazon,90,Novel,90,B1,Novel)
(2001,C1,B2,Amazon,20,Drama,20,B2,Drama)
grunt>
```

Result prints the book names, which Amazon provides for the least price when compared to other sellers.

Result = Foreach Joined1 GENERATE name;

Dump Result shows the result.

A screenshot of a terminal window titled 'soumya@soumya-VirtualBox: ~/Desktop'. The terminal displays the output of a Pig script execution. It starts with 'Counters:' followed by statistics like 'Total records written : 2', 'Total bytes written : 0', and 'Spillable Memory Manager spill count : 0'. Then it shows the 'Job DAG:' with a dependency graph of job_local_0012 through 0016. Finally, it shows log messages from the Pig backend and Hadoop MapReduce, including a success message and input path counts. The prompt 'grunt>' is visible at the bottom.

```
soumya@soumya-VirtualBox: ~/Desktop
Counters:
Total records written : 2
Total bytes written : 0
Spillable Memory Manager spill count : 0
Total bags proactively spilled: 0
Total records proactively spilled: 0

Job DAG:
job_local_0012 ->      job_local_0013,job_local_0015,
job_local_0011 ->      job_local_0013,
job_local_0013 ->      job_local_0014,
job_local_0014 ->      job_local_0015,
job_local_0015 ->      job_local_0016,
job_local_0016

2017-10-21 21:42:50,073 [main] INFO  org.apache.pig.backend.hadoop.executionengi
ne.mapReduceLayer.MapReduceLauncher - Success!
2017-10-21 21:42:50,075 [main] WARN  org.apache.pig.data.SchemaTupleBackend - Sc
hemaTupleBackend has already been initialized
2017-10-21 21:42:50,075 [main] INFO  org.apache.hadoop.mapreduce.lib.input.FileI
nputFormat - Total input paths to process : 1
2017-10-21 21:42:50,081 [main] INFO  org.apache.pig.backend.hadoop.executionengi
ne.util.MapRedUtil - Total input paths to process : 1
(Novel)
(Drama)
grunt>
```

(2.2) Spark RDD

purchase.json is read into *records* by rdd just like the previous task.

records = spark.read.text("/usr/local/spark/pr2/purchase.json").rdd.map(lambda x:x[0])

records is split by tabs and made a bag of bags with elements of each row, just like the previous task.

recordsDivide = records.map(lambda x: x.split('\t'))

recordsDivide.take(50) shows the split data.

As the prices are to be compared among the sellers for each book id, I concatenated the seller and price columns for every bag and made a tuple of book id and concatenated element.

RecordsMap = recordsDivide.map(lambda x: (x[2], x[3] + " " + x[4]))

book.json is read and stored into *bookrecords* by rdd.

```
bookRecords = spark.read.text("/usr/local/spark/pr2/book.json").rdd.map(lambda x: x[0])
```

bookRecords is split and is stored into *bookRecordsDivide*.

```
bookRecordsDivide = records.map(lambda x: x.split('\t'))
```

bookRecordsMap stores the tuples of book id and book name.

```
bookRecordsMap = bookRecordsDivide.map(lambda x: (x[0], x[1]))
```

```
soumya@soumya-VirtualBox: /usr/local/spark
NameError: global name 'b' is not defined
>>> records = spark.read.text("/usr/local/spark/pr2/purchase.json").rdd.map(lambda x: x[0])
>>> recordsDivide = records.map(lambda x: x.split('\t'))
>>> recordsDivide.take(50)
[[u'1999', u'C1', u'B1', u'Amazon', u'90'], [u'2001', u'C1', u'B2', u'Amazon', u'20'], [u'2008', u'C2', u'B2', u'Barnes Noble', u'30'], [u'2008', u'C3', u'B3', u'Amazon', u'28'], [u'2009', u'C2', u'B1', u'Borders', u'90'], [u'2010', u'C4', u'B3', u'Barnes Noble', u'26']]
>>> recordsMap = recordsDivide.map(lambda x: (x[2], x[3] + " " + x[4]))
>>> recordsMap.take(50)
[(u'B1', u'Amazon 90'), (u'B2', u'Amazon 20'), (u'B2', u'Barnes Noble 30'), (u'B3', u'Amazon 28'), (u'B1', u'Borders 90'), (u'B3', u'Barnes Noble 26')]
>>> bookRecords = spark.read.text("/usr/local/spark/pr2/book.json").rdd.map(lambda x: x[0])
>>> bookRecordsDivide = bookRecords.map(lambda x: x.split('\t'))
>>> bookRecordsDivide.take(50)
[[u'B1', u'Novel'], [u'B2', u'Drama'], [u'B3', u'Poem']]
>>> bookRecordsMap = bookRecordsDivide.map(lambda x: (x[0], x[1]))
>>> bookRecordsMap.take(50)
[(u'B1', u'Novel'), (u'B2', u'Drama'), (u'B3', u'Poem')]
>>> combined = bookRecordsMap.join(recordsMap)
>>> combinedPars = combined.map(lambda x: x[1])
>>>
```

I made a table by joining the second columns of *recordsMap* and *bookRecordsMap*, which has the required data to find the minimum prices for every book name.

```
Combined = bookRecordsMap.join(recordsMap)
```

Tuple form is (book name, seller and price) in *combinedPars*. Second element is chosen from *combined*.

```
combinedPars = combined.map(lambda x: x[1])
```

```
>>> combined = bookRecordsMap.join(recordsMap)
>>> combinedPars = combined.map(lambda x: x[1])
>>> combinedPars.take(50)
[(u'Drama', u'Amazon 20'), (u'Drama', u'Barnes Noble 30'), (u'Novel', u'Amazon 90'), (u'Novel', u'Borders 90'), (u'Poem', u'Amazon 28'), (u'Poem', u'Barnes Noble 26')]
>>>
```


I defined a function to find the minimum price with its corresponding seller for each book name. It takes the input like `minimumFunc("Amazon 90", "Barens Noble 26")`, splits each element and compares the price. Returns the whole element with lowest price.

```
def minimumFunc(x, y):
    splitx = int (x.split(" ") [len(x.split(" "))-1])
    splity = int (y.split(" ") [len(y.split(" "))-1])
    if (splitx <= splity):
        return x
    else:
        return y
```

`minimumFunc` should give the following output

```
>>> minimumFunc("Amazon 90", "Barens Noble 26")
'Barens Noble 26'
>>>
```

`allminimums` reduces the `combinedPars` by `minimumFunc`. It gives the output as all the least priced tuples for each book name.

```
allMinimums = combinedPars.reduceByKey(minimumFunc)
```

Now, the least prices offered only by amazon compared to others are to be filtered. For this, `allMinimums` is filtered such that the first element after splitting the `x[1]` in the tuple has to be amazon as following.

```
filteredResult = allMinimums.filter(lambda x: x[1].split(" ") [0] == "Amazon")
```

`filteredResult.take(50)` prints the output as following.

```
>>> allMinimums = combinedPars.reduceByKey(minimumFunc)
>>> allMinimums.take(50)
[(u'Drama', u'Amazon 20'), (u'Poem', u'Barnes Noble 26'), (u'Novel', u'Amazon 90')]
>>> filteredResult = allMinimums.filter(lambda x: x[1].split(" ") [0] == "Amazon")
>>> filteredResult.take(50)
[(u'Drama', u'Amazon 20'), (u'Novel', u'Amazon 90')]
>>>
```

(2.3) Spark SQL

Row feature is imported from the `pyspark.sql`. It is used to name the columns of the tables loaded from the input files.

```
from pyspark.sql import Row
```

I used *customerRecords* to store the data from customer.json by sparkContext sc. Splitting the elements is similar to previous task.

```
customerRecords = sc.textFile("/usr/local/spark/pr2/customer.json")
```

```
>>> from pyspark.sql import Row
>>> customerRecords = sc.textFile("/usr/local/spark/pr2/customer.json")
>>> customerRecordsDivided = customerRecords.map(lambda x: x.split('\t'))
>>> customerRecordsDivided.take(50)
[[u'C1', u'Jackie Chan', u'50', u'Dayton', u'M'], [u'C2', u'Harry Smith', u'30',
u'Beavercreek', u'M'], [u'C3', u'Ellen Smith', u'28', u'Beavercreek', u'F'], [u'C
4', u'John Chan', u'20', u'Dayton', u'M']]
```

Data in *customerRecords* is split by using the following command.

```
customerRecordsDivided = customerRecords.map(lambda x: x.split('\t'))
```

customerRecordsDF uses row to name the columns of the table. From the following figure we can see that each and every element is named now.

```
customerRecordsDF = customerRecordsDivide.map(lambda x: Row(cid = x[0], name =
x[1], age = x[2], city = x[3], sex = x[4]))
```

```
>>> customerRecordsDF = customerRecordsDivide.map(lambda x: Row(cid = x[0], name
= x[1], age = x[2], city = x[3], sex = x[4]))
>>> customerRecordsDF.take(50)
[Row(age=u'50', cid=u'C1', city=u'Dayton', name=u'Jackie Chan', sex=u'M'), Row(ag
e=u'30', cid=u'C2', city=u'Beavercreek', name=u'Harry Smith', sex=u'M'), Row(age=
u'28', cid=u'C3', city=u'Beavercreek', name=u'Ellen Smith', sex=u'F'), Row(age=u'
20', cid=u'C4', city=u'Dayton', name=u'John Chan', sex=u'M')]
```

customerDFSQl stores the content of *customerRecordsDF* as data frame which is used for creating SQL table.

```
customerDFSQl = sqlContext.createDataFrame(customerRecordsDF)
```

```
>>> customerDFSQl = sqlContext.createDataFrame(customerRecordsDF)
>>> customerDFSQl.take(50)
[Row(age=u'50', cid=u'C1', city=u'Dayton', name=u'Jackie Chan', sex=u'M'), Row(ag
e=u'30', cid=u'C2', city=u'Beavercreek', name=u'Harry Smith', sex=u'M'), Row(age=
u'28', cid=u'C3', city=u'Beavercreek', name=u'Ellen Smith', sex=u'F'), Row(age=u'
20', cid=u'C4', city=u'Dayton', name=u'John Chan', sex=u'M')]
>>> customerDFSQl.createOrReplaceTempView("CustomerRecordsTable")
>>>
```

sqlContext is used to bring the functionalities of Spark SQL into scope. For that Spark Context object is initialized at first. createDataFrame is used to create a dataframe.

Name the table by *createOrReplacetempview*.

```
CustomerDFSQl.createOrReplaceTempView("CustomerRecordsTable")
```

To view the outcome of the data frame as table, *.show()* is used.

```
CustomerDFSQl.show()
```

```
soumya@soumya-VirtualBox: /usr/local/spark
>>> customerDFSQL.show()
+----+-----+-----+-----+
|age|cid|      city|      name|sex|
+----+-----+-----+-----+
| 50| C1|    Dayton|Jackie Chan| M|
| 30| C2|Beavercreek|Harry Smith| M|
| 28| C3|Beavercreek|Ellen Smith| F|
| 20| C4|    Dayton|  John Chan| M|
+----+-----+-----+-----+
```

Similarly, the purchase is loaded by spark context object.

```
purchaseRecords = sc.textFile("usr/local/spark/pr2/purchase.json")
```

Data in *purchaseRecords* is split by using the following command.

```
purchaseRecordsDivided = purchaseRecords.map(lambda x: x.split('t'))
```

purchaseRecordsDF uses row to name the columns of the table. From the following figure we can see that every element is named now.

```
purchaseRecordsDF = purchaseRecordsDivide.map(lambda x: Row(year = x[0], cid = x[1], isbn = x[2], seller = x[3], price = x[4]))
```

purchaseDFSQL stores the content of *purchaseRecordsDF* as data frame which is used for creating SQL table.

```
purchaseDFSQL = sqlContext.createDataFrame(purchaseRecordsDF)
```

```
soumya@soumya-VirtualBox: /usr/local/spark
>>> purchaseRecordsDivided = purchaseRecords.map(lambda x: x.split('t'))
>>> purchaseRecordsDivided.take(50)
[[u'1999', u'C1', u'B1', u'Amazon', u'90'], [u'2001', u'C1', u'B2', u'Amazon', u'20'], [u'2008', u'C2', u'B2', u'Barnes Noble', u'30'], [u'2008', u'C3', u'B3', u'Amazon', u'28'], [u'2009', u'C2', u'B1', u'Borders', u'90'], [u'2010', u'C4', u'B3', u'Barnes Noble', u'26']]
>>> purchaseRecordsDF = purchaseRecordsDivide.map(lambda x: Row(year = x[0], cid = x[1], isbn = x[2], seller = x[3], price = x[4]))
>>> purchaseRecordsDF.take(50)
[Row(cid=u'C1', isbn=u'B1', price=u'90', seller=u'Amazon', year=u'1999'), Row(cid=u'C1', isbn=u'B2', price=u'20', seller=u'Amazon', year=u'2001'), Row(cid=u'C2', isbn=u'B2', price=u'30', seller=u'Barnes Noble', year=u'2008'), Row(cid=u'C3', isbn=u'B3', price=u'28', seller=u'Amazon', year=u'2008'), Row(cid=u'C2', isbn=u'B1', price=u'90', seller=u'Borders', year=u'2009'), Row(cid=u'C4', isbn=u'B3', price=u'26', seller=u'Barnes Noble', year=u'2010')]
>>> purchaseDFSQL = sqlContext.createDataFrame(purchaseRecordsDF)
>>> purchaseDFSQL.take(50)
[Row(cid=u'C1', isbn=u'B1', price=u'90', seller=u'Amazon', year=u'1999'), Row(cid=u'C1', isbn=u'B2', price=u'20', seller=u'Amazon', year=u'2001'), Row(cid=u'C2', isbn=u'B2', price=u'30', seller=u'Barnes Noble', year=u'2008'), Row(cid=u'C3', isbn=u'B3', price=u'28', seller=u'Amazon', year=u'2008'), Row(cid=u'C2', isbn=u'B1', price=u'90', seller=u'Borders', year=u'2009'), Row(cid=u'C4', isbn=u'B3', price=u'26', seller=u'Barnes Noble', year=u'2010')]
>>>
```

Name the table by *createOrReplacetempview*.

```
purchaseDFSQL.createOrReplaceTempView("PurchaseRecordsTable")
```

To view the outcome of the data frame as table, `.show()` is used.

```
purchaseDFSQL.show()
```

```
>>> purchaseDFSQL.createOrReplaceTempView("PurchaseRecordsTable")
>>> purchaseDFSQL.show()
+-----+-----+-----+-----+
|cid|isbn|price|seller|year|
+-----+-----+-----+-----+
| C1| B1| 90| Amazon|1999|
| C1| B2| 20| Amazon|2001|
| C2| B2| 30| Barnes Noble|2008|
| C3| B3| 28| Amazon|2008|
| C2| B1| 90| Borders|2009|
| C4| B3| 26| Barnes Noble|2010|
+-----+-----+-----+-----+
```

Both the purchase and customer tables are joined by `cid`, as `cid` is the common column.

‘Select *’ selects all the columns, name *purchaseRecordsTable* as *p* and *customerRecordsTable* as *c* and join *p*, *c* by `cid` as following.

```
combinedTable = spark.sql("select * from purchaseRecordsTable p JOIN  
customerRecordsTable c ON p.cid = c.cid")
```

Name the *combinedTable* .

```
CombinedTable.createOrReplaceTempView("CombineRcordsTable")
```

Use `.show()` to print the table.

```
combinedTable.show()
```

```
>>> combinedTable = spark.sql("select * from purchaseRecordsTable p JOIN customer
RecordsTable c ON p.cid = c.cid")
>>> combinedTable.createOrReplaceTempView("CombinedRecordsTable")
>>> combinedTable.show()
+-----+-----+-----+-----+-----+-----+-----+-----+
|cid|isbn|price|seller|year|age|cid|city|name|sex|
+-----+-----+-----+-----+-----+-----+-----+-----+
| C3| B3| 28| Amazon|2008| 28| C3|Beavercreek|Ellen Smith| F|
| C4| B3| 26| Barnes Noble|2010| 20| C4| Dayton| John Chan| M|
| C1| B1| 90| Amazon|1999| 50| C1| Dayton| Jackie Chan| M|
| C1| B2| 20| Amazon|2001| 50| C1| Dayton| Jackie Chan| M|
| C2| B2| 30| Barnes Noble|2008| 30| C2|Beavercreek|Harry Smith| M|
| C2| B1| 90| Borders|2009| 30| C2|Beavercreek|Harry Smith| M|
+-----+-----+-----+-----+-----+-----+-----+-----+
>>> 
```

Now select only *name* and *isbn* from the *combinedRecordsTable* and make it as a new table.

```
Grouped = spark.sql("select name, isbn from combinedRecordsTable")
```

I named it as *groupedTable*.

```
Grouped.createOrReplaceTempView("GroupedTable")
```

It is displayed as follows.

Grouped.show()

```
>>> grouped = spark.sql("select name, isbn from combinedRecordsTable")
>>> grouped.createOrReplaceTempView("GroupedTable")
>>> grouped.show()
+-----+-----+
|      name|isbn|
+-----+-----+
|Ellen Smith|  B3|
|  John Chan|  B3|
|Jackie Chan|  B1|
|Jackie Chan|  B2|
|Harry Smith|  B2|
|Harry Smith|  B1|
+-----+-----+
```

The customer with name Harry Smith, whatever books he has purchased, that set of book *isbns* are selected and checked in the *combinedRecordsTable* if there is any other customer with same set of *isbns* and make them as a group.

result = spark.sql("select name from combinedRecordsTable where isbn in (select isbn from GroupedTable where name = \"Harry Smith\") group by name")

Final result is displayed by

result.show()

```
>>> result = spark.sql (" select name from combinedRecordsTable where isbn in (se
lect isbn from GroupedTable where name = \"Harry Smith\") group by name")
>>> result.show()
+-----+
|      name|
+-----+
|Harry Smith|
|Jackie Chan|
+-----+
>>> 
```

(2.3) Spark RDD

Records loads the data from purchase.json

Records = spark.read.text("/usr/local/spark/pr2/purchase.json").rdd.map(lambda x: x[0])

customerRecords loads the data from customer.json.

customerRecords = spark.read.text("/usr/local/spark/pr2/customer.json").rdd.map(lambda x: x[0])

The elements in records are split and stored them as tuples in a bag.

```
recordsDivide = records.map(lambda x: x.split('\t'))
```

The bag recordsMap has cid and isbn as tuple elements.

```
recordsMap = recordsDivide.map(lambda x: x[1], x[2]))
```

The elements in customerRecords are split and stored them as tuples in a bag.

```
customerRecordsDivide = customerRecords.map(lambda x: x.split('\t'))
```

The bag customerRecordsMap has cid and name as tuple elements.

```
customerRecordsMap = recordsDivide.map(lambda x: x[0], x[1]))
```

```
>>> records = spark.read.text("/usr/local/spark/pr2/purchase.json").rdd.map(lambda x: x[0])
>>> recordsDivide = records.map(lambda x: x.split('\t'))
>>> recordsMap = recordsDivide.map(lambda x: (x[1], x[2]))
>>> recordsMap.take(50)
[(u'C1', u'B1'), (u'C1', u'B2'), (u'C2', u'B2'), (u'C3', u'B3'), (u'C2', u'B1'),
(u'C4', u'B3')]
>>> customerRecords = spark.read.text("/usr/local/spark/pr2/customer.json").rdd.map(lambda x: x[0])
>>> customerRecordsDivide = customerRecords.map(lambda x: x.split('\t'))
>>> customerRecordsMap = customerRecordsDivide.map(lambda x: (x[0], x[1]))
>>> customerRecordsMap.take(50)
[(u'C1', u'Jackie Chan'), (u'C2', u'Harry Smith'), (u'C3', u'Ellen Smith'), (u'C4', u'John Chan')]
>>>
```

customerRecordsMap and recordsMap are joined.

```
Combined = customerRecordsMap.join(recordsMap)
```

Only first column of combined is stored in combinedPars

```
combinedPars = combined.map(lambda x: x[1])
```

combinedPars reduces combined as set of books bought for each customer name. First element in the tuple would be the customer name, second one is the concatenation of set of isbn.

```
combinedPars1 = combinedPars.reduceByKey(lambda, y: x + " " + y)
```

Mapped makes the concatenated element as a set of book ids by splitting them as elements.

```
Mapped = combinedPars1 = combinedPars.map(lambda x: (x[0], set(x[1].split(" "))))
```



```

>>> combined = customerRecordsMap.join(recordsMap)
>>> combined.take(50)
[(u'C3', (u'Ellen Smith', u'B3')), (u'C1', (u'Jackie Chan', u'B1')), (u'C1', (u'Jackie Chan', u'B2')), (u'C2', (u'Harry Smith', u'B2')), (u'C2', (u'Harry Smith', u'B1')), (u'C4', (u'John Chan', u'B3'))]
>>> combinedPars = combined.map(lambda x: x[1])
>>> combinedPars.take(50)
[(u'Ellen Smith', u'B3'), (u'Jackie Chan', u'B1'), (u'Jackie Chan', u'B2'), (u'Harry Smith', u'B2'), (u'Harry Smith', u'B1'), (u'John Chan', u'B3')]
>>> combinedPars1 = combinedPars.reduceByKey(lambda x, y: x + " " + y)
>>> combinedPars1.take(50)
[(u'John Chan', u'B3'), (u'Ellen Smith', u'B3'), (u'Harry Smith', u'B2 B1'), (u'Jackie Chan', u'B1 B2')]
>>> Mapped = combinedPars1.map(lambda x: (x[0], set(x[1].split(" "))))
>>> Mapped.take()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: take() takes exactly 2 arguments (1 given)
>>> Mapped.take(50)
[(u'John Chan', set([u'B3'])), (u'Ellen Smith', set([u'B3'])), (u'Harry Smith', set([u'B1', u'B2'])), (u'Jackie Chan', set([u'B1', u'B2']))]
>>>

```

Harry stores the tuples only if the first element is Harry Smith and flat maps the set of *book ids*.

```

Harry = set(Mapped.filter(lambda x: x[0] == "Harry Smith")
.flatMap(lambda x: x[1]).collect())

```

I defined a function *sameSetOfBooks*, which returns the same tuples with same set of books that harry had. *X[1]* is compared.

```

Def sameSetOfBooks(x):
    If(harry.issubset(x[1])):
        Return x

```

Check maps the *Mapped* bag with the *sameSetOfBooks* function, which filters only the tuples with same set of book ids that harry bought.

```

Check = Mapped.map(sameSetOfBooks)

```

Result filters the check, deletes the elements that don't match with harry.

```

Result = check.filter(lambda x: x != None)

```

```

>>> harry = set(Mapped.filter(lambda x: x[0] == "Harry Smith")
... .flatMap(lambda x: x[1]).collect())
>>> def sameSetOfBooks(x):
...     if(harry.issubset(x[1])):
...         return x
...
>>> check = Mapped.map(sameSetOfBooks)
>>> check.take(50)
[None, None, (u'Harry Smith', set([u'B1', u'B2'])), (u'Jackie Chan', set([u'B1', u'B2']))]
>>> result = check.filter(lambda x: x != None)
>>> result.take(50)
[(u'Harry Smith', set([u'B1', u'B2'])), (u'Jackie Chan', set([u'B1', u'B2']))]
>>>

```