Project Purpose

The Purpose of this project is to analyze customer behavior and market fluctuation in an online store using spark Data Frame API. The data used for this analysis comprises of customer behavioral data for the month of October and November of the year 2020, which would help in understanding the customers event type which could be “view”, ”purchase”, ”cart”. The reason for choosing these specific month related data is, the fact that here is a huge boom in shopping during the thanksgiving season and it is assumed that people view what they need in the previous month-October and make the purchase in November. Is this because of price reduction, offers or just a psychological pull, which makes a customer feel happy that he/she did a purchase during thanks-giving. The third data frame used for this analysis is the technology index for the year 2016.This dataset has price of most commonly bought electronic gadget price across many countries. Customers do buy gadgets from other countries rather than their own country due to the price variations .These variations may be due to currency value .This dataset added an interesting aspect of which country has the cheapest gadget for the year 2016 kind of data. It also helps to compare the gadget price in 2016 with that in 2019.

Data Frames

There data frames are use in this project which are derived from 3 csv files

Data Frame	CSV file	Details
octDF	2019-Oct.csv	This data frame contains customer behavioral data of an online store for the month of October 2019.
novDF	2019-Nov.csv	This data frame contains customer behavioral data of an online store for the month of November 2019.
tecinDF	TechnologyIndex.csv	This data frame contains cost of electronic gadgets in various countries for the year 2016.

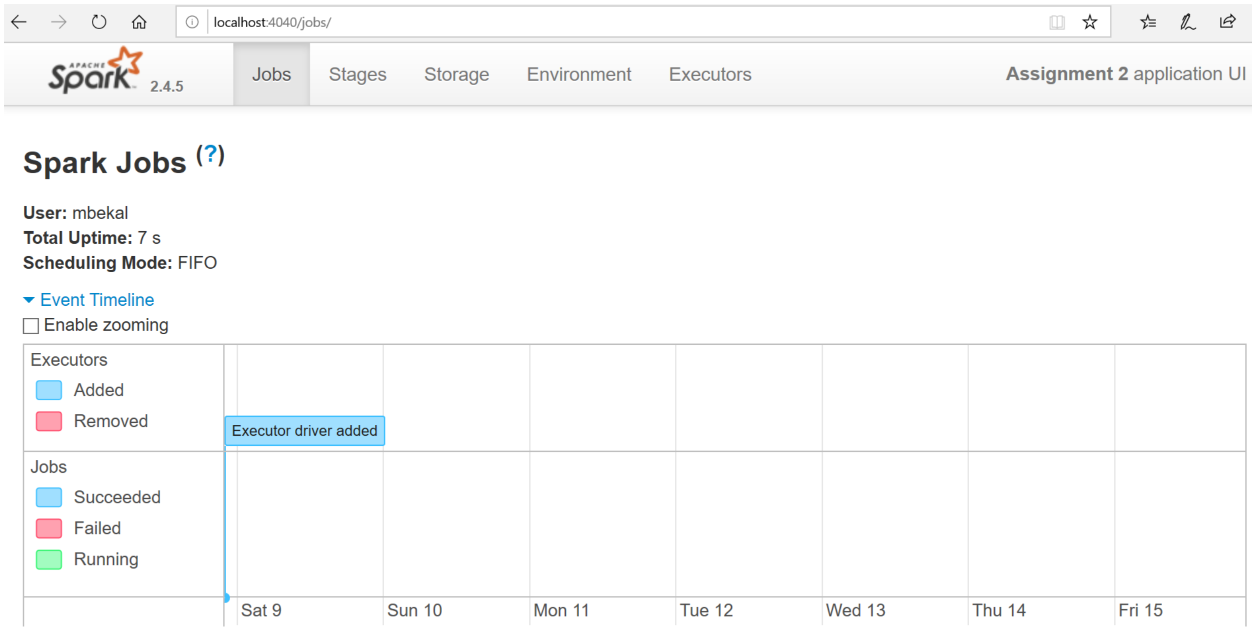
Reason for not using Parquet files

In this project we did not use parquet files for the below mentioned reasons.

1. The data in the three data frames were not having any kind of complex data formats.
2. The October data frame and November data frame had data pertaining to that particular month. It more seemed like enough partitioning is present for the data. However, I did considering partitioning based on event type or category but it did not make sense with a month set of data.
3. The entire project took close to 44seconds for completing. No complaints of performance degradation.

Spark UI

The local host is set to *http://localhost:4040* in the local mode. The UI is used for monitoring the job progress. Below is the screenshot of how my local UI looked when the executer driver was added.



Spark Jobs (?)

User: mbekal
Total Uptime: 7 s
Scheduling Mode: FIFO

Schema

The schemas for the three data frames used in this project are listed below. The purpose of putting this up is to give the column names and types of the data frame at a glance. **octDF** and **novDF** have the same schema, while the **tecinDF** has its own schema.

Oct 2019 DataFrame

```
root
 |-- event_time: string (nullable = true)
 |-- event_type: string (nullable = true)
 |-- category_code: string (nullable = true)
 |-- brand: string (nullable = true)
 |-- price: string (nullable = true)
 |-- userid: string (nullable = true)
```

Nov 2019 DataFrame

```
root
 |-- event_time: string (nullable = true)
 |-- event_type: string (nullable = true)
 |-- category_code: string (nullable = true)
 |-- brand: string (nullable = true)
 |-- price: string (nullable = true)
 |-- userid: string (nullable = true)
```

Technology Index 2016 DataFrame

```
root
 |-- country: string (nullable = true)
 |-- iPhone: string (nullable = true)
 |-- Android: string (nullable = true)
 |-- MacBook: string (nullable = true)
 |-- WindowsPowered: string (nullable = true)
 |-- PS4: string (nullable = true)
 |-- Xboxone: string (nullable = true)
 |-- iPadmini: string (nullable = true)
 |-- Samsungtablet: string (nullable = true)
 |-- fortyinchsmartTV: string (nullable = true)
 |-- AppleWatch: string (nullable = true)
 |-- Brandheadphone: string (nullable = true)
 |-- harddrive2TB: string (nullable = true)
 |-- Portablecharger: string (nullable = true)
 |-- Printer: string (nullable = true)
```

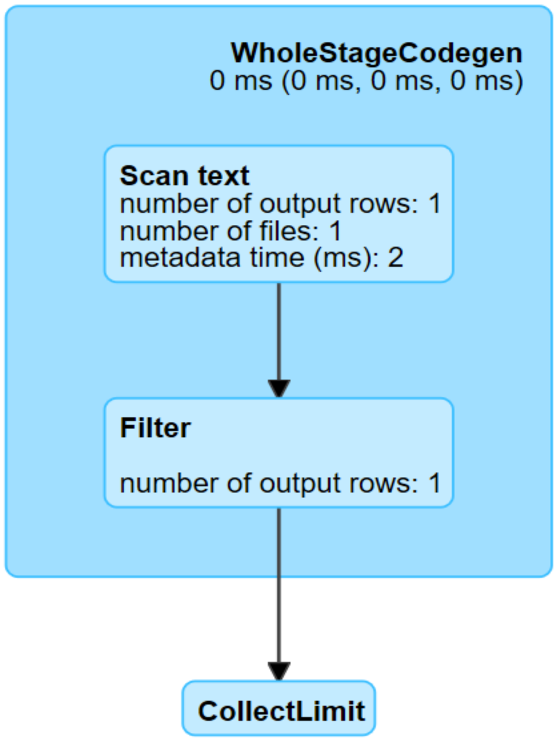
Business questions and execution plan

Problem 1: Count of users based on the behavior- viewing, adding to cart, removing from cart and purchasing in the month of OCT-2019?

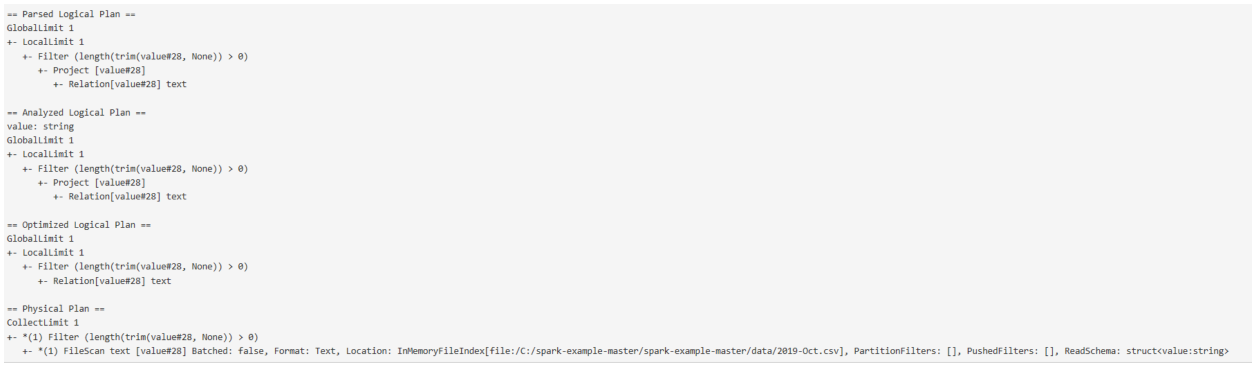
Data Movement Stats	1 data frame is returned
Partitioning /Predicating	Uncorrelated predicate subqueries.
Interesting aspects	It is uncorrelated, as it does not provide any information for the outer scope of the query. It’s a query that you can run on its own. In the sense, the subqueries do not have dependency on each other and could be run separately.

Spark computation details

Run duration: 7782 milliseconds

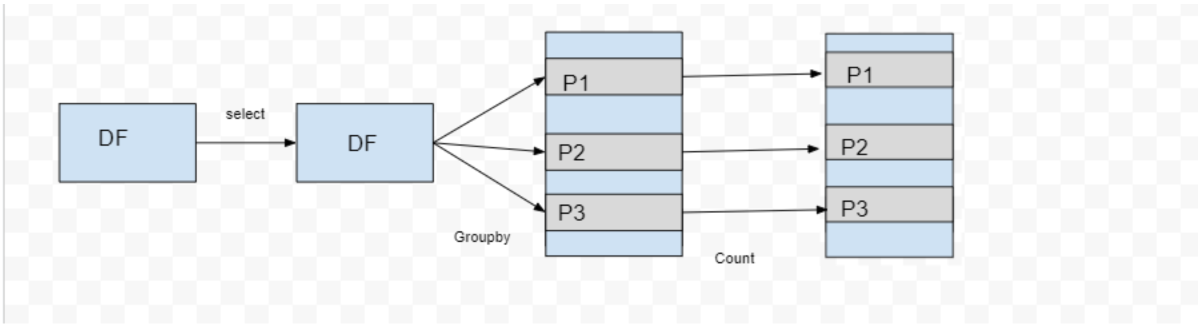


Details of execution



```
== Parsed Logical Plan ==
GlobalLimit 1
+- LocalLimit 1
   +- Filter (length(trim(value#28, None)) > 0)
      +- Project [value#28]
         +- Relation[value#28] text

== Analyzed Logical Plan ==
value: string
GlobalLimit 1
+- LocalLimit 1
   +- Filter (length(trim(value#28, None)) > 0)
      +- Project [value#28]
         +- Relation[value#28] text

== Optimized Logical Plan ==
GlobalLimit 1
+- LocalLimit 1
   +- Filter (length(trim(value#28, None)) > 0)
      +- Relation[value#28] text

== Physical Plan ==
CollectLimit 1
+- *(1) Filter (length(trim(value#28, None)) > 0)
   +- *(1) FileScan text [value#28] Batched: false, Format: Text, Location: InMemoryFileIndex[file:/C:/spark-example-master/spark-example-master/data/2019-Oct.csv], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<value:string>
```

Transformation type:



The image above depicts my understanding of narrow and wide dependencies of the transformations used in the problem one query.

Step1: Selecting the event type

The transformation type is narrow. This is because; each input partition will contribute to only one output partition.

Step2: Grouping based on event type.

The group by transformation has wide dependencies, as group by has to group records/rows based on the key, in this case being event type. The records are shuffled in order to make sure records with same event type are place in the respective partition.

Step 3: counting based on event type.

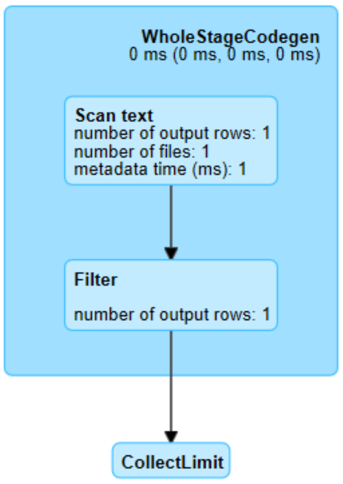
The counting happens based on co-partitioned inputs and hence it is has narrow dependency.

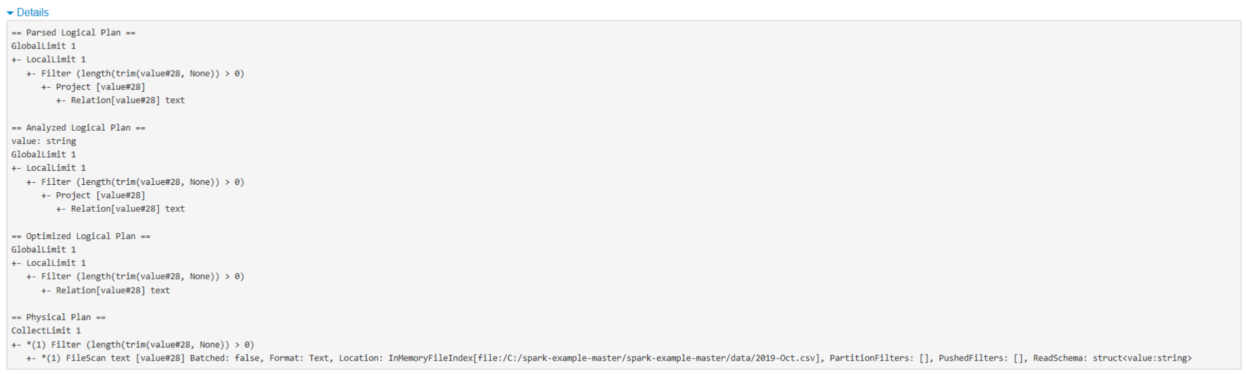
<u>Problem 2</u>: What are the unique product categories available in the online store?

Data Movement Stats	1 data frame is returned
Partitioning /Predicating	Uncorrelated predicate subqueries.
Interesting aspects	It is uncorrelated, as it does not provide any information for the outer scope of the query. It is a query that can run on its own. In the sense, the subqueries do not have dependency on each other.

<u>Spark computation details</u>

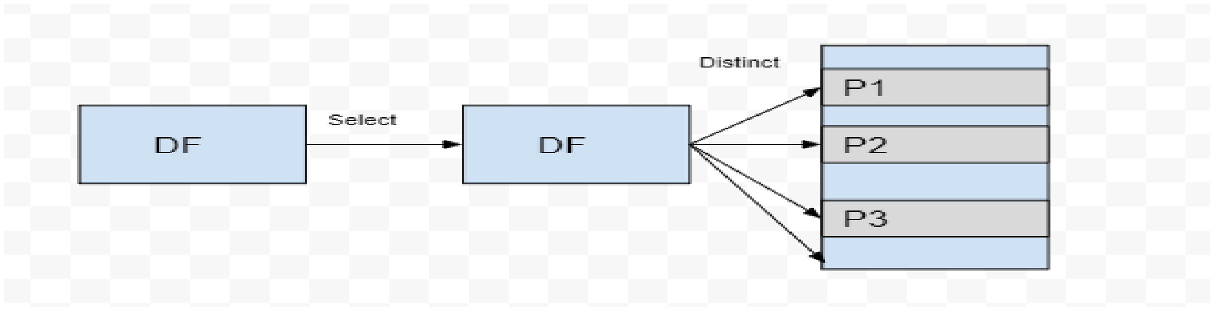
Run duration: 1839 milliseconds





```
== Parsed Logical Plan ==
GlobalLimit 1
+- LocalLimit 1
   +- Filter (length(trim(value#28, None)) > 0)
      +- Project [value#28]
         +- Relation[value#28] text

== Analyzed Logical Plan ==
value: string
GlobalLimit 1
+- LocalLimit 1
   +- Filter (length(trim(value#28, None)) > 0)
      +- Project [value#28]
         +- Relation[value#28] text

== Optimized Logical Plan ==
GlobalLimit 1
+- LocalLimit 1
   +- Filter (length(trim(value#28, None)) > 0)
      +- Relation[value#28] text

== Physical Plan ==
CollectLimit 1
+- *(1) Filter (length(trim(value#28, None)) > 0)
   +- *(1) FileScan text [value#28] Batched: false, Format: Text, Location: InMemoryFileIndex[file:/C:/spark-example-master/spark-example-master/data/2019-Oct.csv], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<value:string>
```

Transformation type:



Step1: Selecting based on category code.

This transformation is narrow as there is no shuffling required.

Step2: Distinct values of category code.

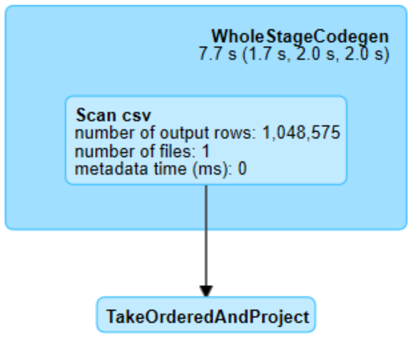
Distinct has wide dependencies as there is some shuffling to see if the category code of a row need to be considered as distinct value or need to be discarded as it has been included already.

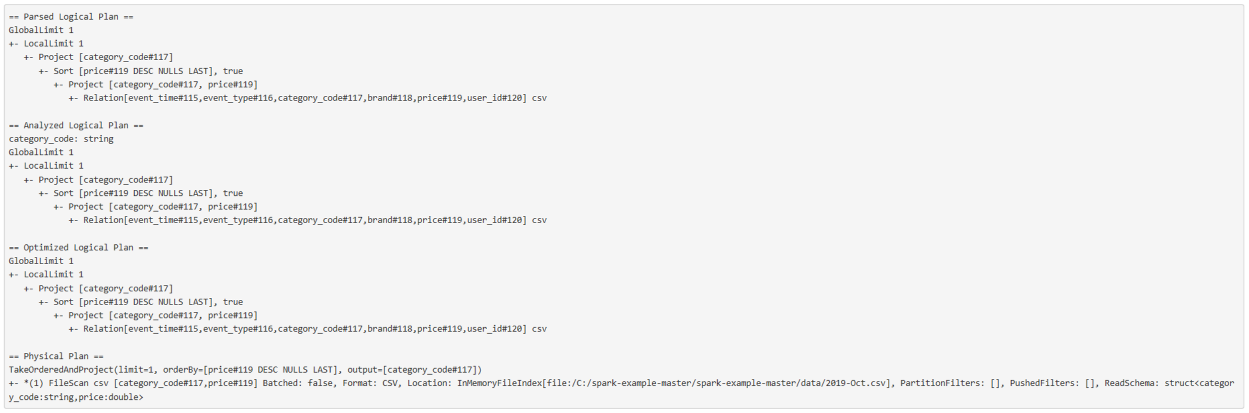
Problem 3: Which product is the most expensive in the entire online store?

Data Movement Stats	1 row is returned
Partitioning /Predicating	Correlated predicate subqueries.
Interesting aspects	The query is correlated as the second subquery changes the order of the first subquery.

Spark computation details:

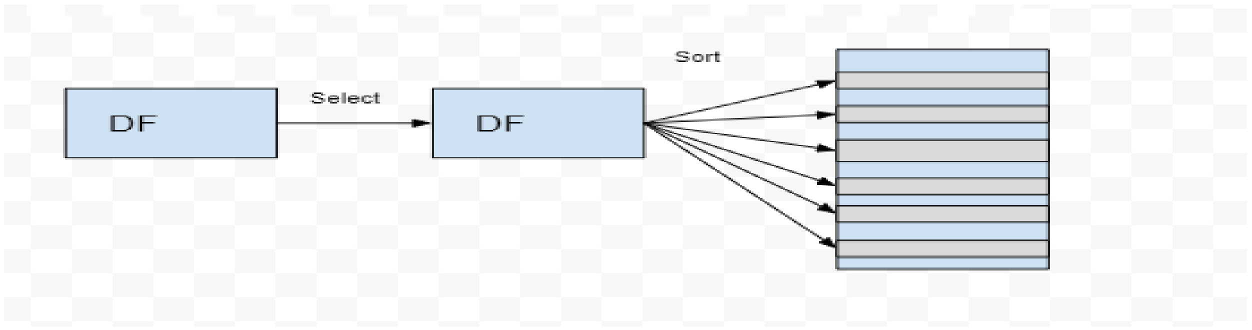
Run duration: 3972 milliseconds





```
== Parsed Logical Plan ==
GlobalLimit 1
+- LocalLimit 1
   +- Project [category_code#117]
      +- Sort [price#119 DESC NULLS LAST], true
         +- Project [category_code#117, price#119]
            +- Relation[event_time#115,event_type#116,category_code#117,brand#118,price#119,user_id#120] csv

== Analyzed Logical Plan ==
category_code: string
GlobalLimit 1
+- LocalLimit 1
   +- Project [category_code#117]
      +- Sort [price#119 DESC NULLS LAST], true
         +- Project [category_code#117, price#119]
            +- Relation[event_time#115,event_type#116,category_code#117,brand#118,price#119,user_id#120] csv

== Optimized Logical Plan ==
GlobalLimit 1
+- LocalLimit 1
   +- Project [category_code#117]
      +- Sort [price#119 DESC NULLS LAST], true
         +- Project [category_code#117, price#119]
            +- Relation[event_time#115,event_type#116,category_code#117,brand#118,price#119,user_id#120] csv

== Physical Plan ==
TakeOrderedAndProject(limit=1, orderBy=[price#119 DESC NULLS LAST], output=[category_code#117])
+- *(1) FileScan csv [category_code#117,price#119] Batched: false, Format: CSV, Location: InMemoryFileIndex[file:/C:/spark-example-master/spark-example-master/data/2019-Oct.csv], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<categor
y_code:string,price:double>
```

Transformation Type:



Step1: Selecting category code from the data frame.

Narrow, as no shuffling is required.

Step2: Sorting in descending order based on price.

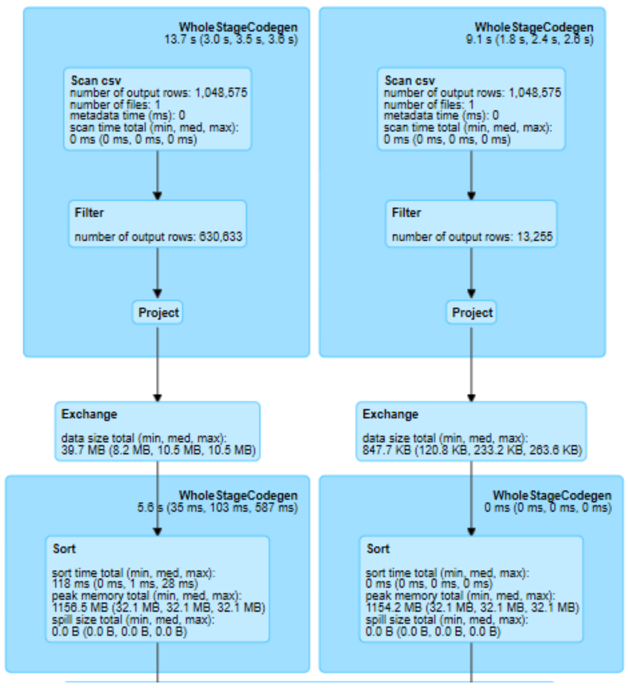
This transformation is wide as there is a lot of shuffling required in order to attain the rows in descending order of price.

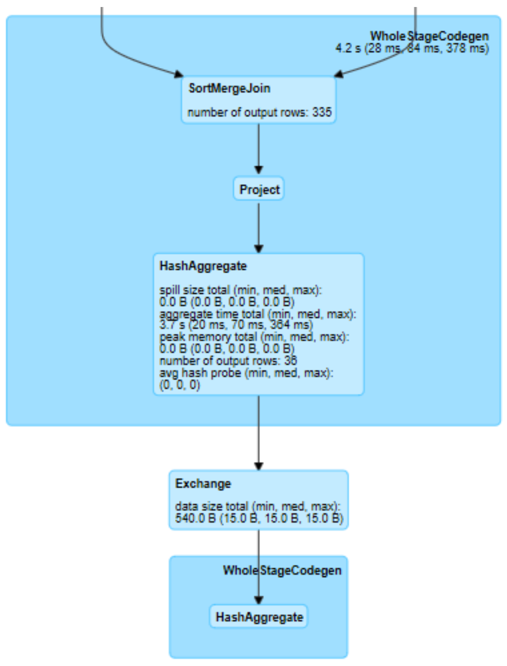
Problem 4: Count of customers who just viewed products in month of OCT-2019 and made purchases in NOV-2019.

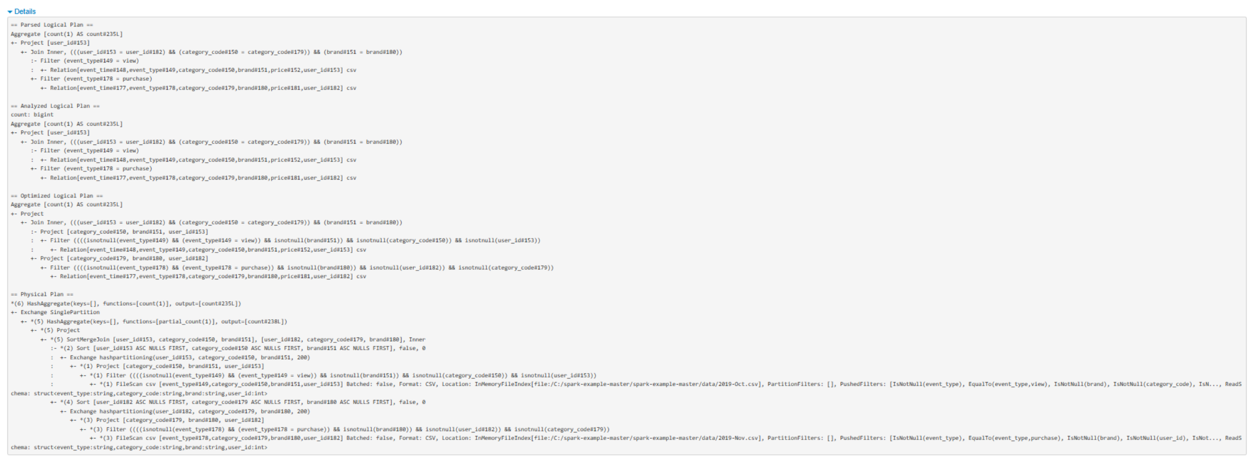
Data Movement Stats	1 row is returned
Partitioning /Predicating	Partitioning
Interesting aspects	Irrespective of so many transformations involved, the time taken to process is 18seconds.

<u>Spark computation details:</u>

Run duration: 18391 milliseconds

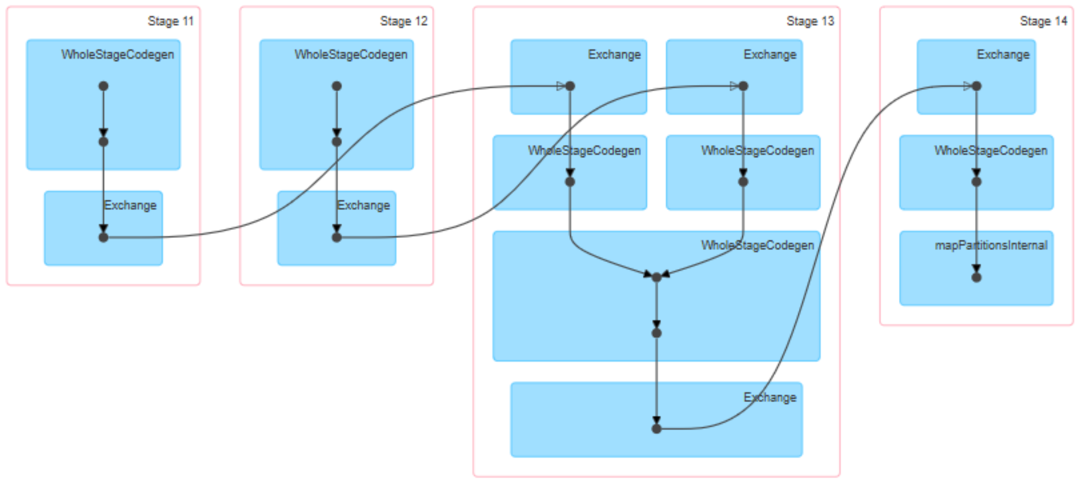

WholeStageCodegen
13.7 s (3.0 s, 3.5 s, 3.6 s)
Scan csv
number of output rows: 1,048,575
number of files: 1
metadata time (ms): 0
scan time total (min, med, max):
0 ms (0 ms, 0 ms, 0 ms)
Filter
number of output rows: 630,633
Project
WholeStageCodegen
9.1 s (1.8 s, 2.4 s, 2.6 s)
Scan csv
number of output rows: 1,048,575
number of files: 1
metadata time (ms): 0
scan time total (min, med, max):
0 ms (0 ms, 0 ms, 0 ms)
Filter
number of output rows: 13,255
Project
Exchange
data size total (min, med, max):
39.7 MB (8.2 MB, 10.5 MB, 10.5 MB)
Exchange
data size total (min, med, max):
847.7 KB (120.8 KB, 233.2 KB, 263.6 KB)
WholeStageCodegen
5.6 s (35 ms, 103 ms, 587 ms)
Sort
sort time total (min, med, max):
118 ms (0 ms, 1 ms, 28 ms)
peak memory total (min, med, max):
1156.5 MB (32.1 MB, 32.1 MB, 32.1 MB)
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
WholeStageCodegen
0 ms (0 ms, 0 ms, 0 ms)
Sort
sort time total (min, med, max):
0 ms (0 ms, 0 ms, 0 ms)
peak memory total (min, med, max):
1154.2 MB (32.1 MB, 32.1 MB, 32.1 MB)
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)


WholeStageCodegen
4.2 s (28 ms, 84 ms, 378 ms)
SortMergeJoin
number of output rows: 335
Project
HashAggregate
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
aggregate time total (min, med, max):
3.7 s (20 ms, 70 ms, 364 ms)
peak memory total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
number of output rows: 36
avg hash probe (min, med, max):
(0, 0, 0)
Exchange
data size total (min, med, max):
540.0 B (15.0 B, 15.0 B, 15.0 B)
WholeStageCodegen
HashAggregate

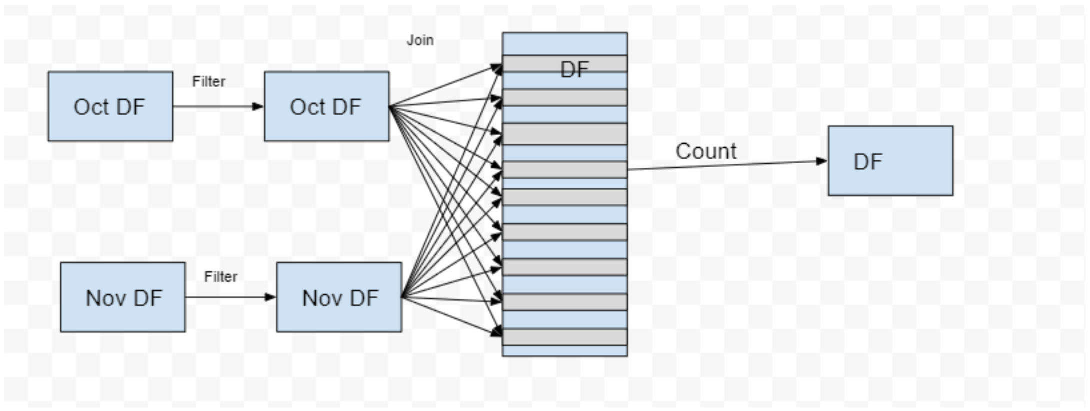

▾ Details

```
== Parsed Logical Plan ==
Aggregate [count(1) AS count#235L]
+- Project [user_id#153]
   +- Join Inner, (((user_id#153 = user_id#182) && (category_code#150 = category_code#179)) && (brand#151 = brand#180))
      :- Filter (event_type#149 = view)
      :  +- Relation[event_time#148,event_type#149,category_code#150,brand#151,price#152,user_id#153] csv
      +- Filter (event_type#178 = purchase)
         +- Relation[event_time#177,event_type#178,category_code#179,brand#180,price#181,user_id#182] csv

== Analyzed Logical Plan ==
count: bigint
Aggregate [count(1) AS count#235L]
+- Project [user_id#153]
   +- Join Inner, (((user_id#153 = user_id#182) && (category_code#150 = category_code#179)) && (brand#151 = brand#180))
      :- Filter (event_type#149 = view)
      :  +- Relation[event_time#148,event_type#149,category_code#150,brand#151,price#152,user_id#153] csv
      +- Filter (event_type#178 = purchase)
         +- Relation[event_time#177,event_type#178,category_code#179,brand#180,price#181,user_id#182] csv

== Optimized Logical Plan ==
Aggregate [count(1) AS count#235L]
+- Project
   +- Join Inner, (((user_id#153 = user_id#182) && (category_code#150 = category_code#179)) && (brand#151 = brand#180))
      :- Project [category_code#150, brand#151, user_id#153]
      :  +- Filter ((((isnotnull(event_type#149) && (event_type#149 = view)) && isnotnull(brand#151)) && isnotnull(category_code#150)) && isnotnull(user_id#153))
      :     +- Relation[event_time#148,event_type#149,category_code#150,brand#151,price#152,user_id#153] csv
      +- Project [category_code#179, brand#180, user_id#182]
         +- Filter ((((isnotnull(event_type#178) && (event_type#178 = purchase)) && isnotnull(brand#180)) && isnotnull(user_id#182)) && isnotnull(category_code#179))
            +- Relation[event_time#177,event_type#178,category_code#179,brand#180,price#181,user_id#182] csv

== Physical Plan ==
*(6) HashAggregate(keys=[], functions=[count(1)], output=[count#235L])
+- Exchange SinglePartition
   +- *(5) HashAggregate(keys=[], functions=[partial_count(1)], output=[count#238L])
      +- *(5) Project
         +- *(5) SortMergeJoin [user_id#153, category_code#150, brand#151], [user_id#182, category_code#179, brand#180], Inner
            :- *(2) Sort [user_id#153 ASC NULLS FIRST, category_code#150 ASC NULLS FIRST, brand#151 ASC NULLS FIRST], false, 0
            :  +- Exchange hashpartitioning(user_id#153, category_code#150, brand#151, 200)
            :     +- *(1) Project [category_code#150, brand#151, user_id#153]
            :        +- *(1) Filter ((((isnotnull(event_type#149) && (event_type#149 = view)) && isnotnull(brand#151)) && isnotnull(category_code#150)) && isnotnull(user_id#153))
            :           +- *(1) FileScan csv [event_type#149,category_code#150,brand#151,user_id#153] Batched: false, Format: CSV, Location: InMemoryFileIndex[file:/C:/spark-example-master/spark-example-master/data/2019-Oct.csv], PartitionFilters: [], PushedFilters: [IsNotNull(event_type), EqualTo(event_type,view), IsNotNull(brand), IsNotNull(category_code), IsN..., ReadS
chema: struct<event_type:string,category_code:string,brand:string,user_id:int>
            +- *(4) Sort [user_id#182 ASC NULLS FIRST, category_code#179 ASC NULLS FIRST, brand#180 ASC NULLS FIRST], false, 0
               +- Exchange hashpartitioning(user_id#182, category_code#179, brand#180, 200)
                  +- *(3) Project [category_code#179, brand#180, user_id#182]
                     +- *(3) Filter ((((isnotnull(event_type#178) && (event_type#178 = purchase)) && isnotnull(brand#180)) && isnotnull(user_id#182)) && isnotnull(category_code#179))
                        +- *(3) FileScan csv [event_type#178,category_code#179,brand#180,user_id#182] Batched: false, Format: CSV, Location: InMemoryFileIndex[file:/C:/spark-example-master/spark-example-master/data/2019-Nov.csv], PartitionFilters: [], PushedFilters: [IsNotNull(event_type), EqualTo(event_type,purchase), IsNotNull(brand), IsNotNull(user_id), IsNot..., ReadS
chema: struct<event_type:string,category_code:string,brand:string,user_id:int>
```

DAG visualization:



Transformation Types:



Step 1: Filtering oct data frame based on event type being view.

This does not require shuffling, hence it has narrow dependency.

Step2: Filtering nov data frame based on event type being purchase.

Narrow dependency.

Step3: joining based on userid, category code and brand.

Wide dependency as there is a lot of shuffling involved.

Step4: Count of rows.

Count has narrow dependency as there is no shuffling required and each input partition will contribute to only one output partition.

Join used:

Inner join – Since both the data frames have the same field names, the join was easier. By doing an inner join, the transformation evaluates the keys(userid, category code and brand)in both the data frames and include only the rows which satisfy the evaluation make it to the resultant data frame.

Problem 5: Is the cost of i-pad same in Oct 2019 and Nov 2019

Data Movement Stats	1 row is returned(Boolean)
Partitioning /Predicating	Partitioning-Data is partitioned based on brand and then sorted in descending order to get the cost of the most expensive i-pad.
Interesting aspects	Both the data frames had same fields, This helped in coming up with the solution easily.

Spark computation details:

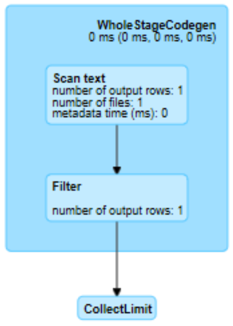
Run duration: 6310 milliseconds

Problem 6: Which country sold the costliest PS4 in 2016?

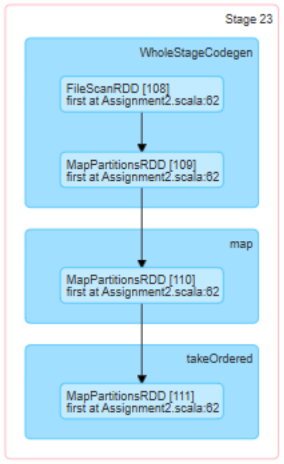
Data Movement Stats	1 row is returned
Partitioning /Predicating	Predicate
Interesting aspects	There is not much shuffling required.

Spark computation details:

Run duration: 486 milliseconds



DAG Visualization



Transformation type:

Transformation name	**Dependency type**
Select	Narrow
Sort	Wide

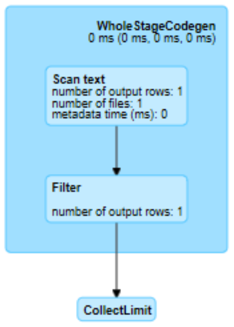
Problem 7: Is the cost of MacBook in 2019 more than how much it costed in 2016.

Data Movement Stats	1 row is returned(Boolean)
Partitioning /Predicating	Partitioning
Interesting aspects	It is interesting to see how the spark behave irrespective of the data type or the return type.

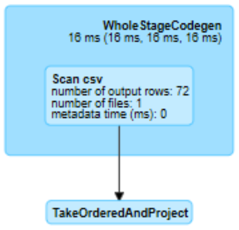
Spark computation details:

Run duration: 3121 milliseconds

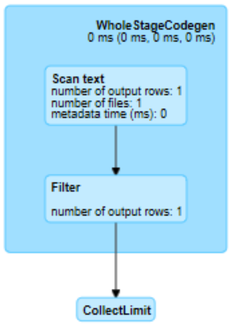
Query1:when the cost of the Mac book is found from Nov 2019 data frame



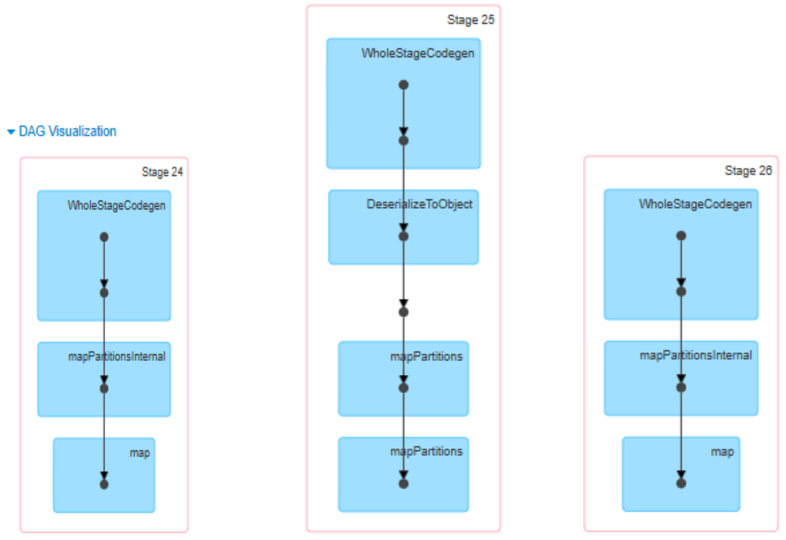
Query2: Cost of MacBook for The USA in the year 2016 is determined from technology index data frame.



Query 3: comparing the results from both the above queries and returning a Boolean.



DAG visualization:



Transformation types involved:

Transformation name	**Dependency type**
Select	Narrow
Filter	Narrow
Sort	Wide
First	Narrow

GitHub Link

https://github.com/monicabekal/spark-example-master/tree/master/spark-example-master