List of Tables

List of Figures

Sales Analysis of Iowa Liquor Sales using Hadoop and Spark

A Performance Analysis with Spark Optimization

Seth Kofi Agbavitor
sethagbavitor23@gmail.com

ABSTRACT

This scientific report presents a pipeline for analyzing Iowa liquor sales data to answer specific use cases. The use cases include identifying the top 5 cities with the most liquor purchases, determining the month when liquor is sold the most, identifying the top 10 brands and liquor types, and exploring whether certain cities prefer specific liquors over others.

The pipeline involves using Hadoop to convert unstructured data into a structured dataset and saving it in CSV format. Spark is then used to create a data schema, read the CSV files, transform text variables into proper data types, and enhance the dataset using an algorithm.

To optimize Spark performance, the report implements three algorithms: Caching, Repartitioning and countDistinct(). Caching involves storing frequently used data in memory, reducing the time it takes to read data from disk and improving overall performance. Repartitioning involves partitioning data into smaller chunks to be processed in parallel, which can improve the efficiency of processing large datasets. countDistinct() uses the HyperLogLog algorithm to estimate the number of distinct values in the dataset, improving the efficiency of processing large datasets.

This pipeline provides a comprehensive approach to analyzing Iowa liquor sales data and answering specific use cases while optimizing Spark performance to improve efficiency and reduce processing time.

1 INTRODUCTION

Sales data analysis is essential for firms and organizations seeking to optimize their operations and maximize profitability. This scientific research outlines a method for analyzing Iowa liquor sales data in order to answer specific use cases. Among the use cases are determining the top 5 cities with the highest liquor purchases, calculating the month when the most liquor is sold, identifying the top 10 brands and liquor types, and investigating whether certain towns prefer certain liquors over others.

Big Data technologies have enabled businesses to collect, process, and analyze large volumes of data. However, processing such large datasets can be computationally intensive and time-consuming. Spark, a popular Big Data processing engine, can handle massive datasets by distributing the processing across a cluster of computers. Despite its efficiency, Spark can be further optimized to improve its performance.

Hadoop is a popular framework for processing and storing large datasets, including unstructured data. However, when dealing with large amounts of data, Hadoop can struggle to provide efficient processing. To address this issue, many organizations turn to Spark, a Big Data processing engine capable of handling massive datasets by distributing the processing across a cluster of computers. Spark can be further optimized to improve its performance, and this report focuses on three such Spark optimization techniques: Caching, Repartitioning, and countDistinct() function.

1.1 Caching

Data that is often accessed is cached in memory to increase efficiency by minimizing the time required to read data from disk. The caching optimization strategy involves keeping frequently used data in memory, which reduces the time required to read data from disk and improves overall efficiency. Caching is especially effective for frequently read datasets or processing queries that require repeated access to the same data. The cache optimization strategy can reduce the amount of disk reads required to access data by keeping frequently used data in memory, resulting in faster query processing times.

Caching works by temporarily storing frequently used data in memory so that it may be accessed fast when needed. The information is kept in memory as cache objects that are indexed with a key. The caching algorithm checks the cache for the requested data when a query is run. If the data is located in the cache, it is returned immediately rather than having to be retrieved from disk. If the material is not discovered in the cache, it is retrieved from disk and cached for later use.

1.2 Repartitioning

Data is divided into smaller parts for parallel processing. The repartitioning optimization algorithm divides data into smaller chunks that can be handled in parallel, improving the efficiency of processing huge datasets. Repartitioning is especially beneficial for huge datasets that cannot be processed in a single run or for queries that require concurrent processing.

Repartitioning divides a dataset into smaller portions, or partitions, that can be handled concurrently. The partitions are distributed among numerous nodes in a cluster, allowing the dataset to be processed by multiple nodes at the same time. The repartitioning optimization approach can minimize processing time and increase overall performance by processing the dataset in parallel. Furthermore, repartitioning can aid in ensuring that the workload is distributed evenly across the cluster's nodes, preventing any single node from becoming overloaded.

1.3 countDistinct()

The HyperLogLog technique is used to calculate the number of different values in a dataset. The countDistinct() optimization algorithm calculates the number of distinct values in a dataset. This optimization approach is very beneficial when dealing with enormous datasets that cannot be processed in a single run. Counting different values in huge datasets takes time and can be computationally expensive, especially when traditional counting methods are used. The HyperLogLog approach, on the other hand, can provide a faster and more accurate estimate of the number of different values in a dataset.

The HyperLogLog algorithm is a probabilistic algorithm that estimates the number of different values in a dataset using hash functions. The technique operates by mapping the dataset's values to a set of hash values. The number of different values in the dataset is then estimated using these hash values. Even when the dataset is quite huge, the HyperLogLog algorithm can produce an accurate estimate of the number of different values [4].

2 BACKGROUND

Iowa is among the few states in the United States that permits the sale of liquor in private establishments [5]. Iowa's liquor industry has experienced remarkable growth over the years, generating considerable revenue for the state. In the 2020 fiscal year, the state recorded over 100 million Dollars liquor revenue [3].

Analyzing Iowa liquor sales data can provide critical insights into the liquor industry and consumer behavior. Such insights can be useful for liquor stores and producers in optimizing their marketing and sales strategies. Also, policymakers and researchers can use the data to understand the socio-economic implications of the liquor industry in the state.

The data used in this study was obtained from the Iowa Alcoholic Beverages Division (ABD) and is publicly available on the data.gov website [2]. The dataset contains detailed information on every liquor purchase made in the state from 2016 to 2022, including the date of the purchase, location of the purchase, the type of liquor, the brand, and the price. This rich dataset provides a unique opportunity to analyze liquor sales trends in Iowa and gain insights into consumer behavior in the state.

3 METHOD

In this section, we describe how our application was implemented, including the configuration of our Hadoop cluster. We also provide information about our dataset and explain how we used Hadoop MapReduce to read unstructured Iowa liquor sales data. Finally, we provide a detailed description of our implementation of three Spark optimization techniques recommended in literature.

3.1 Data collection

The Iowa Alcoholic Beverage Division provided the dataset for this investigation and it was downloaded from data.gov website. The size of the dataset is 11.72 GB with 24 columns and 5,808,648 rows structured in table.The original dataset was unstructured into a text file and 19 columns were used. The columns used for this analysis are: Date; Date of which an order was made, City; City where the store that ordered the liquor is located, CategoryName; The type of liquor ordered, ItemDescription; Description of the specific alcoholic beverage ordered.

3.2 Hadoop cluster configuration

We created 1 namenode and 3 datanodes virtual machines respectively in openstack. The namenode was allocated a disk space of 40 GB and a ram of 4 GB while each of the 3 datanodes were allocated a disk space of 80 GB and a ram of 8 GB respectively to enhance the data processing and execution time. Hadoop version 3.2.1 was setup in a distributed mode along with spark version 3.3.2 installed. We also set the logging level of our Spark application to "OFF" for certain loggers, such as "org" and "akka", and for the Spark driver program. This disables logging output to the console or log files during run time, which can improve performance and reduce clutter.

3.3 Reading unstructured data from text dataset with Hadoop

MapReduce: MapReduce is a java-based processing technique and program architecture for distributed computing. The MapReduce method includes two key tasks: Map and Reduce. Map translates one collection of data into another, where individual pieces are split down into tuples (key/value pairs). Second, there is the reduction job, which takes the result of a map as an input and merges those data tuples into a smaller set of tuples. The reduction work is always executed after the map job [7].

3.3.1 Map stage .
In Hadoop, the mapper is responsible for handling input data, which is typically stored in a file or directory within the Hadoop file system (HDFS). The input file is processed by the mapper function line by line, with each line representing a unit of data. The mapper analyzes and transforms the data, breaking it down into smaller, more manageable pieces.

```
class LiqourSale(MRJob):
    def mapper(self, _, line):
```

From the code snippets above, the "LiqourSale" class is defining a MapReduce job that will process data related to liquor sales using MRjob library in Python. The job has a mapper function, which is defined with the "mapper" method of the class. The mapper function takes two arguments, _ and line. The underscore (_) is a placeholder argument that is not actually used in the function. The line argument represents a single line of input data that will be processed by the mapper. The mapper function will be called once for each line of input data. The job will read input data from a file and pass each line to the mapper function as the line argument. We then emit the extracted fields as key-value pairs as shown in the code snippets below:

```
yield None, (Date, Store, Address, City, Zip, CountyNumber,
County, Category, CategoryName, VendorNumber,
VendorName, ItemNumber, ItemDescription,
StateBottleCost, StateBottleRetail, BottlesSold,
SaleDollars, VolumeSoldLiteres, VolumeSoldGallons)
```

3.3.2 Reduce stage.
This stage combines the Shuffle and Reduce stages in Hadoop. The Reducer is responsible for handling data outputted by the mapper. Once the data is processed by the reducer, it generates a fresh set of output, which is then stored in the Hadoop file system (HDFS).

```
def reducer(self, _, values):
    for fields in values:
       yield None, ','.join(fields)
```

From the code above, The "reducer" function is responsible for processing and combining the output from the "mapper" functions. The "reducer" function takes two arguments: "self" and "values". The first argument, "self", refers to the instance of the class that the function is defined in, and is required for methods defined in a class. The second argument, "values", is an iterable object that contains a sequence of values that were emitted from the "mapper" function. Each value in "values" is a tuple that contains the key and value data emitted by the "mapper" function.

The "reducer" function then iterates through the "values" iterable using a for loop, and for each value, it joins the fields in the tuple using a comma as a separator. The " ','.join(fields)" function call creates a string that concatenates all the fields in the tuple together, separated by commas. Finally, the "reducer" function yields a key-value pair of "None" and the concatenated string of fields, using the yield statement. The output of the "reducer" function is then passed to the output handler for the MapReduce job.

3.4 Creating data schema and updating the results in a delta table with Spark

```
custom_schema = StructType([
    StructField("Date", StringType(), True),
    StructField("Store", StringType(), True),
    StructField("Address", StringType(), True),
    StructField("City", StringType(), True),
    StructField("Zip", IntegerType(), True),
    StructField("CountyNumber", IntegerType(), True),
    StructField("County", StringType(), True),
    StructField("Category", IntegerType(), True),
    StructField("CategoryName", StringType(), True),
    StructField("VendorNumber", IntegerType(), True),
    StructField("VendorName", StringType(), True),
    StructField("ItemNumber", IntegerType(), True),
   StructField("ItemDescription", StringType(), True),
   StructField("StateBottleCost", DoubleType(), True),
  StructField("StateBottleRetail", DoubleType(), True),
    StructField("BottleSold", IntegerType(), True),
    StructField("SaleDollars", DoubleType(), True),
  StructField("VolumeSoldLiters", DoubleType(), True),
  StructField("VolumeSoldGallons", DoubleType(), True)
])
df = spark.read.csv("/dis_materials/liqour1",
     header=False, schema=custom_schema).cache()
df.unpersist()
```

In the code snippet, we defined a custom schema to specify the structure of a liquor sales dataset. Date, Store, Address, CategoryName, VendorName, and SaleDollars information were among the fields contained in the schema. This was done to ensure that the data was correctly and consistently read during processing. The liquor sales dataset was saved in CSV format and read into a DataFrame using the read.csv() method in Apache Spark. To ensure that the data was read with the desired structure, the custom schema was passed as an input to the function. In addition, the DataFrame was cached for faster access during later processing steps.

We were able to execute numerous actions on the data using the Spark DataFrame API, such as filtering, selecting, aggregating, grouping, and sorting. Complex data types such as arrays, maps, and structs are also supported. The API includes a number of built-in data processing and transformation methods, as well as the ability to define custom functions. The Spark DataFrame API is highly optimized for distributed computing, allowing us to process our large datasets efficiently and quickly [6].

```
df.write.option("overwriteSchema", "true").mode
("overwrite").format("delta").saveAsTable
("liqourtable", path="hdfs:///dis_materials/l3")
```

In the code snippet, We wrote a Spark code to save a DataFrame as a Delta Lake table in Hadoop Distributed File System (HDFS). In the code, we used the "write" method of the DataFrame to specify the options for overwriting the schema of the table. We set the option "overwriteSchema" to "true" to ensure that any changes in the DataFrame's schema will be reflected in the Delta Lake table.

Next, we specified the write mode as "overwrite" to overwrite any existing data in the Delta Lake table with the new data from the DataFrame. We used the "format" method to specify the format of the output file, which is delta in this case. Finally, we called the "saveAsTable" method to save the DataFrame as a table in Delta Lake format in the specified path on HDFS, with the name "liqourtable".

3.5 Spark optimization

In this section, we will look at some optimization approaches that we used in our Spark application. Caching, repartitioning, and the "countDistinct()" function are examples of these strategies. These techniques were implemented to improve the performance of our application when processing large datasets. We were able to considerably reduce computation time and improve the performance of our program by employing these optimization strategies. In the sections that follow, we will go over each strategy in depth and explain how we implemented them in our Spark application.

3.5.1 Caching.

```
df = spark.read.csv("/dis_materials/liqour1",
header=False, schema=custom_schema).cache()
df.unpersist()
```

Two optimizations are made to the Spark DataFrame in this code. The first optimization is caching, which is done after the DataFrame has been loaded by invoking the "cache()" method. We utilized caching to keep the DataFrame in memory and reuse it across several processes. This considerably enhance performance, especially when we perform several computations on the same DataFrame. The second optimization is to execute the "unpersist()" method to unpersist the cached DataFrame. This method releases the cached DataFrame from memory, making room for future computations.

This is useful when we no longer need to use the cached DataFrame and wish to free up the memory it takes up.

```
df_sales_count = df.groupBy('City').count().cache()
top_5_most_sales = df_sales_count.orderBy
(desc('count')).limit(5)
```

Considering the code snippet, we applied the "groupBy()" function to our dataset, grouping the liquor sales data by "City". We then used the "count()" function to calculate the number of liquor sales for each city. we added the "cache()" function to our DataFrame to keep the frequently used data in memory. This allowed us to access the results faster, as we did not have to fetch them from disk repeatedly.

Next, we used the "orderBy()" function to sort the results in descending order based on the count of liquor sales. Finally, we limited the output to the top 5 cities with the highest liquor sales by using the "limit()" function. By using caching, we minimized the time required to read data from disk, which helped to improve the performance of our data analysis process.

3.5.2 *Repartioning.*

```
df = df.repartition(4)

df.select('Date', 'City', 'CategoryName',
'ItemDescription').describe().show()
```

In this code above, we first divide the DataFrame into four partitions for improved parallel processing. The "describe()" function is then used to obtain summary statistics for each column after selecting a selection of columns from the DataFrame. For each column, the "describe()" function computes the count, mean, standard deviation, minimum, and maximum values. During data cleaning, we used the summary statistics for data analysis and quality checking.

```
df_repartitioned = df.repartition
('CategoryName', 'ItemDescription')

top_brands = df_repartitioned.groupBy('ItemDescription')
.count().orderBy(desc('count')).limit(10)
```

In this code snippet, we first repartitioning the dataframe df by the columns CategoryName and ItemDescription using the repartition function. This operation redistributes the data across the partitions based on the specified columns, which improve the execution time of our program .

Next, we are grouping the repartitioned dataframe by the ItemDescription column and counting the number of occurrences of each unique value using the count function. We then order the resulting DataFrame in descending order by the count, and limit the output to the top 10 most frequent ItemDescription values using the limit function. This code finds the top ten most often occuring ItemDescription values in the DataFrame while taking advantage of repartitioning for greater performance.

3.5.3 *countDistinct().*

```
from pyspark.sql.functions import countDistinct, desc

df.select('City') \
     .groupBy('City') \
     .agg(countDistinct('City').alias('count')) \
     .orderBy(desc('City')) \
     .limit(10) \
     .show()
```

From the code, we begin by importing the required functions from the pyspark.sql.functions package. Then, using select, we select the City column from the DataFrame. Using "groupBy", we group the resulting DataFrame by "City".

The "countDistinct" function is then used to count the number of unique values in the "City" column, and the resulting column is aliased as "count" using "agg". This counts the number of unique cities in the dataset and renames the resulting column. Then, using "orderBy" with "desc", we sort the resulting DataFrame by the "City" column in descending order. We use "limit" to limit the output to 10 rows, and then present the resulting DataFrame.

4 IOWA LIQUOR SALES DATA ANALYSIS

In this section, we will explain the implementation of our four use cases and analyze the results and findings from our output. The use cases include: identifying the top 5 cities with the highest liquor purchases in Iowa, determining the month with the highest liquor sales, identifying the 10 most popular liquor brands and types, and investigating whether certain cities have a preference for specific types of liquor over others.

4.1 Which top 5 cities has the most liquor purchase from 2016 to 2022

Code and output:

```
df_sales_count = df.groupBy('City').count().cache()
top_5_most_sales = df_sales_count.orderBy(desc('count')).limit(5)
top_5_most_sales.show()
```

Table 1: Top 5 cities

City	count
DES MOINES	403379
CEDAR RAPIDS	337229
DAVENPORT	227799
WEST DES MOINES	202997
WATERLOO	156766

Table 1, shows the top 5 cities with the highest liquor sales. This table is quite fair. Des Moines, Cedar Rapid,and Davenport are the first, second and third popular cities by population in Iowa state while West Des Moines falls at 7th and Waterloo at 6th positions respectively according to Iowa Demographics by CUBIT [1]. Although Sioux City and Iowa City are the 4th and 5th cities by population West Des Moines and Waterloo appeared to have liquor purchase ahead of these two cities.

4.2 During Which month is liquor sold the most?

Code and output:

```
df = df.withColumn('Month', substring(split('Date', '/')
[0], 1, 2))
top_months=df.groupBy('Month').agg(count('*').alias
('total_sales')).orderBy(desc('total_sales'))
top_months.show()
```

Table 2: Monthly sales

Month	total_sales
1	1118
2	794212
3	201390
4	30
5	912448
6	1138656
7	202
8	119516
9	1276290
10	234
11	1132612
12	231939

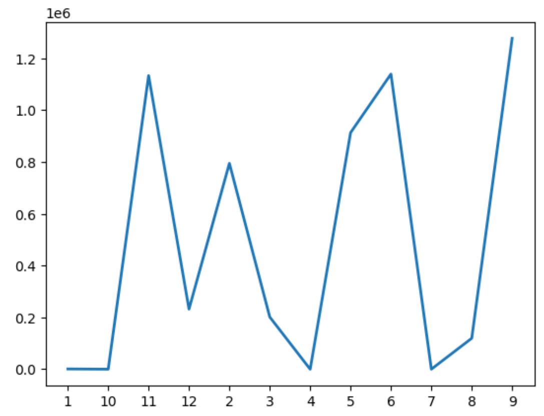

Figure 1: Monthly Sales

From the results, September recorded the highest liquor sale. One possible reason is that September marks the beginning of the fall season when college students return to school and football season starts. These events could lead to an increase in social gatherings and parties, which may result in higher liquor sales. Additionally, September also marks the end of summer when people may be more likely to have outdoor gatherings, barbecues, and other social events that could also contribute to higher liquor sales.

4.3 Which top 10 brands and top 10 liquor types are most popular in Iowa

Code and output:

```
df_repartitioned=df.repartition('CategoryName','ItemDescription')

top_brands=df_repartitioned.groupBy('ItemDescription')
.count().orderBy(desc('count')).limit(10)

top_types=df_repartitioned.groupBy('CategoryName')
.count().orderBy(desc('count')).limit(10)

print(top_brands)
print(top_types)
```

Table 3: To 10 by brands

ItemDescription	count
TITOS HANDMADE VODKA	133532
FIREBALL CINNAMON WHISKEY	120427
BLACK VELVET	115621
HAWKEYE VODKA	103157
CAPTAIN MORGAN ORIGINAL SPICED	77107
CROWN ROYAL	68596
CROWN ROYAL REGAL APPLE	67859
SMIRNOFF 80PRF	55944
SEAGRAMS 7 CROWN	53261
JIM BEAM	53155

Table 4: Top 10 liquor types

CategoryName	count
AMERICAN VODKAS	811788
CANADIAN WHISKIES	481656
STRAIGHT BOURBON WHISKIES	393641
WHISKEY LIQUEUR	299874
AMERICAN FLAVORED VODKA	252559
SPICED RUM	226778
BLENDED WHISKIES	222259
100% AGAVE TEQUILA	192765
AMERICAN SCHNAPPS	172338
COCKTAILS /RTD	171029

Top 10 brands and top 10 liquor types:

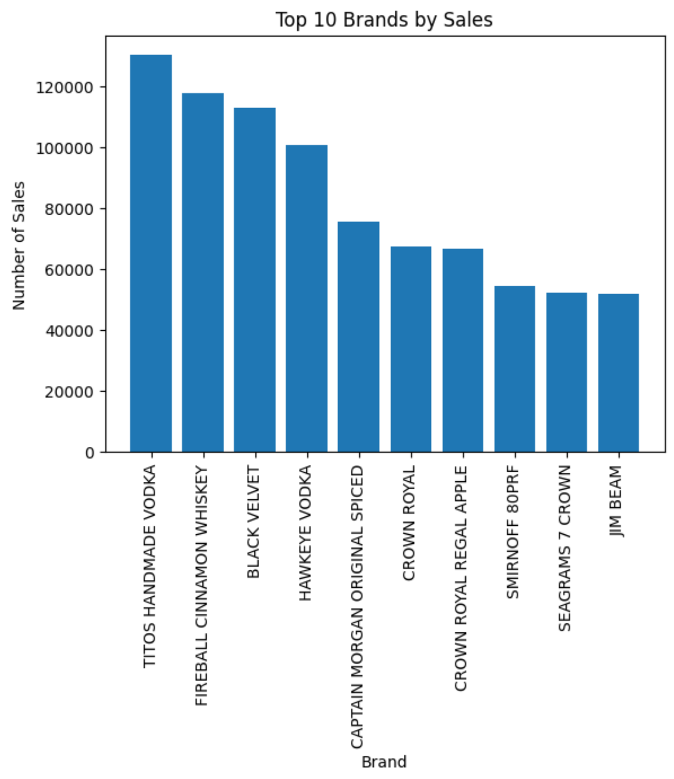

Figure 2: Top 10 brands

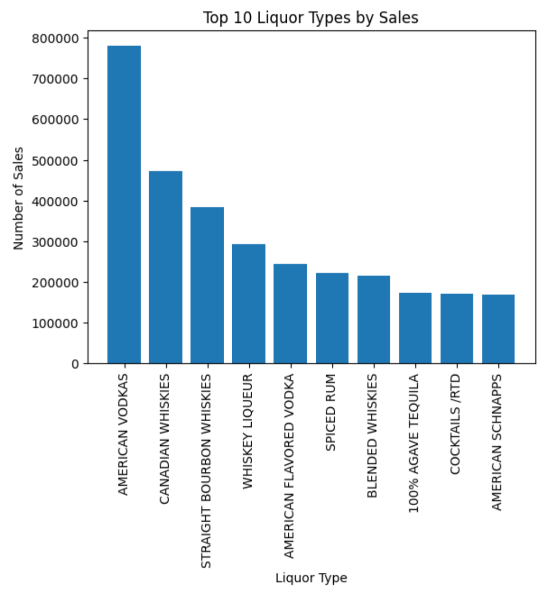

Figure 3: Top 10 liquor types

The results show that vodka and whiskey are the most popular types of alcoholic beverages. This result could indicate that the prices of vodka and whiskey are cheaper compared to other types of liquors.

4.4 Comparing liquor preference of two college cities; Ames and Iowa City

Code and output:

```
    top_type_ames = df.filter(df['City'] == 'AMES') \
            .groupBy('CategoryName') \
            .agg(count('*').alias('sales_count')) \
            .orderBy('sales_count', ascending=False) \
            .limit(4) \

  top_type_ia_city = df.filter(df['City'] == 'IOWA CITY') \
            .groupBy('CategoryName') \
            .agg(count('*').alias('sales_count')) \
            .orderBy('sales_count', ascending=False) \
            .limit(4) \
```

Table 5: Sales count by category in Ames

CategoryName	sales_count
AMERICAN VODKAS	19399
STRAIGHT BOURBON WHISKIES	13032
CANADIAN WHISKIES	10185
AMERICAN FLAVORED VODKA	9330

Table 6: Sales count by category in Iowa City

CategoryName	sales_count
AMERICAN VODKAS	22341
STRAIGHT BOURBON WHISKIES	11227
AMERICAN FLAVORED VODKA	7996
CANADIAN WHISKIES	7852

The tables indicate that the two cities have a preference for the same four popular types of liquor. However, the results suggest that vodka is more popular in both cities than whiskey.

5 RESULTS

This section presents a table with the execution times of our baseline code verses each optimization we implemented. We also present a screenshots from our Spark user interface.

5.1 Caching

Base line code:

```
 from pyspark.sql.functions import desc

df_sales_count = df.groupBy('City').count()
top_5_most_sales = df_sales_count.orderBy(desc('count')).limit(5)
top_5_most_sales.show()
```

Optimized code:

```
from pyspark.sql.functions import desc

df_sales_count = df.groupBy('City').count().cache()
top_5_most_sales=df_sales_count.orderBy(desc('count'))
.limit(5)
```

Table 7: Execution time for baseline code vs execution time for cache optimized code

Baseline Code (sec)	Optimized Code (sec)
17.7	9.4
16.9	7.3
14.3	5.1

5.2 Repartitioning

5.2.1 First repartition. Baseline code:

```
num_rows = df.count()
num_cols = len(df.columns)

df.select('Date', 'City', 'CategoryName',
'ItemDescription').describe().show()

print("There are {:,} rows and {} columns.\n"
.format(num_rows, num_cols))
```

Optimized code:

```
num_rows = df.count()
num_cols = len(df.columns)

df = df.repartition(4)

df.select('Date','City','CategoryName',
'ItemDescription').describe().show()

print("There are {:,} rows and {} columns.\n"
.format(num_rows, num_cols))
```

Table 8: Execution time for baseline code vs execution time for first repartition optimized code

Baseline Code (sec)	Optimized Code (sec)
318.5	162.5
316.6	158.3
317.6	160.2

5.2.2 Second repartition. Baseline code:

```
top_brands = df.groupBy('ItemDescription').count()
.orderBy(desc('count')).limit(10)

top_types = df.groupBy('CategoryName').count()
.orderBy(desc('count')).limit(10)

print(top_brands)
print(top_types)
```

Optimized code:

```
df_repartitioned = df.repartition('CategoryName', 'ItemDescription'

top_brands = df_repartitioned.groupBy('ItemDescription').count()
.orderBy(desc('count')).limit(10)


top_types = df_repartitioned.groupBy('CategoryName').count()
.orderBy(desc('count')).limit(10)

print(top_brands)
print(top_types)
```

Table 9: Execution time for baseline code vs execution time for second repartition optimized code

Baseline Code (sec)	Optimized Code (sec)
51.2	26.4
41.2	24.9
40.5	25.2

5.3 countDistinct()

Baselin code:

```
df.groupBy('City') \
     .pivot('City') \
     .count() \
     .orderBy(desc('City')) \
     .limit(10) \
     .show()

df.groupBy('CategoryName') \
     .pivot('CategoryName') \
     .count() \
     .orderBy(desc('CategoryName')) \
     .limit(10) \
     .show()
```

Optimized code:

```
df.select('City') \
     .groupBy('City') \
    .agg(countDistinct('City').alias('count')) \
    .orderBy(desc('City')) \
    .limit(10) \
    .show()

df.select('CategoryName') \
    .groupBy('CategoryName') \
   .agg(countDistinct('CategoryName').alias('count')) \
    .orderBy(desc('CategoryName')) \
    .limit(10) \
    .show()
```

Table 10: Execution time for baseline code (pivot function) vs execution time for (countDistinct function) optimized code

Baseline Code (sec)	Optimized Code (sec)
91	42.3
80.9	39.4
78.9	39.7

6 DISCUSSION

In this section, we will discuss the reasons why we obtained those results from our optimizations which include: caching, repartitioning, and the use of the countDistinct function.

6.1 Caching

In this analysis, we optimized the code to find the top 5 cities with the most liquor purchase in Iowa using Spark DataFrame API. In the baseline code, we grouped the data by the "City" column and counted the number of rows for each group. However, we did not add any optimization to the code to make it more efficient. This means that every time we needed to access the DataFrame, Spark would read it from disk again, which can be time-consuming for large datasets.

In the optimized code, we added a caching step after the groupBy operation. This means that we stored the DataFrame in memory after the first read, and every subsequent access to the DataFrame did not require reading from disk again. This optimization significantly reduced the execution time of the code, as Spark could retrieve the data much faster from memory than from disk.

Therefore, the main reason why the first code has a longer execution time compared to the second code is that it does not have the caching optimization. In contrast, the second code has a caching step that reduces the amount of disk reading, making it more efficient and faster.

6.2 Repartition

6.2.1 First repartition. In this data cleaning and preprocessing task, we optimized the code to improve the performance of our data analysis using Spark DataFrame API. In the baseline code, we counted the number of rows and columns in the DataFrame, and then selected specific columns to show descriptive statistics. However, we did not include any optimization step to improve the performance of the code.

In the optimized code, we added a repartition step to the DataFrame after counting the number of rows and columns. This step improved the performance of the code by reducing the number of partitions in the DataFrame. Fewer partitions allow Spark to process the data more efficiently, as it can shuffle data between nodes more quickly. As a result, the optimized code executed faster than the baseline code. The repartition step improved the performance of the code by reducing the number of partitions, and Spark could shuffle data more efficiently.

Therefore, the main reason why the first code has a longer execution time compared to the second code is that it does not have the repartition optimization. In contrast, the second code has a repartition step that improves the performance of the code by reducing the number of partitions, making it more efficient and faster.

6.2.2 Second repartition. In the first code, the DataFrame is not repartitioned before calling the groupBy function. This means that the data will not be evenly distributed among the partitions, leading to skewed performance. This can cause some partitions to take longer to process than others, resulting in slower overall execution time.

On the other hand, the second code optimizes the DataFrame by repartitioning it before calling the groupBy function. This ensures that the data is evenly distributed among the partitions, resulting in faster processing times. This optimization eliminates the potential bottleneck caused by uneven partitioning in the first code and significantly improves the overall execution time.

6.3 countDistinct()

The first code is using the "pivot" function, which creates a new DataFrame with columns for each unique value in the pivot column. In this case, it creates many new columns for each unique value in the "City" and "CategoryName" columns, which could be computationally expensive and lead to a larger DataFrame that takes more time to process.

On the other hand, the second code is using the "countDistinct" function to calculate the number of distinct values in each group of the "City" and "CategoryName" columns. This is a simpler operation that does not create new columns or significantly increase the size of the DataFrame. Therefore, it can be processed more efficiently and quickly than the first code.

7 CONCLUSION

Finally, we improved the performance of our data analysis by optimizing our Spark code in various ways. We employed caching to minimize disk reading, repartitioning to distribute data uniformly among partitions and eliminate potential bottlenecks, and the countDistinct function to simplify operations and reduce DataFrame size.

We were able to dramatically reduce the execution time of our code by adding these enhancements, making our data analysis faster and more efficient. When dealing with huge datasets, where even little efficiency improvements can have a substantial impact, these optimizations are critical. We learned the significance of Spark optimization techniques and how to use them to increase the performance of our data analysis. We will continue to investigate and implement new ideas.

REFERENCES

[1] Iowa Demographics by CUBIT. [n. d.]. Iowa Cities by Population. https://www.iowa-demographics.com/cities_by_population. Accessed May 2, 2023.

[2] Iowa Alcoholic Beverages Division. [n. d.]. Iowa Liquor Sales. https://catalog.data.gov/dataset/iowa-liquor-sales. Accessed March 20, 2023.

[3] Iowa Alcoholic Beverages Division. 2020. Annual Report. https://abd.iowa.gov/annual-reports. Accessed April 26, 2023.

[4] Stefan Heule, Michael Nunkesser, and Alex Hall. 2013. Algorithms for HyperLogLog Sketches. *Journal of Machine Learning Research* 14 (2013), 1–30. https://doi.org/10.5555/2998733.2998785

[5] National Institute on Alcohol Abuse and Alcoholism. 2018. State Alcohol Laws. https://www.niaaa.nih.gov/stateprofiles/state-alcohol-laws. Accessed April 26, 2023.

[6] Apache Spark. [n. d.]. DataFrames. https://spark.apache.org/docs/latest/sql-data-sources-dataframes.html. Accessed May 2, 2023.

[7] tutorialspoint. [n. d.]. Hadoop - MapReduce. https://www.tutorialspoint.com/hadoop/hadoop_mapreduce.htm. Accessed April 30, 2023.