TOWARDS INTELLIGENT DATA PROFILING AND AGGREGATION

Nidhi Menon, Sneha Venkatachalam

1. Introduction

Data Scientists continuously rely on modern data systems to provide support for data profiling and exploratory statistical analysis. This process often involves recomputation of certain repeated measures and statistics, without a comprehensive mechanism to organize and reuse them.

We address this problem by proposing a system that is capable of intelligently aggregating preliminary statistics to speed-up the computation of dependent metrics. Basic statistical metrics are pre-computed on data instances and stored in memory. These pre-computed values are then used to accelerate query processing of metrics that make use of these basic statistics for computation. We aim to find aggregation patterns in various data profiling operations to speed-up exploratory data analytics tasks, and thereby resulting in faster querying of data.

2. Motivation

In most data science pipelines, it is very often a necessity for data scientists to conduct several rounds of exploratory data analysis in order to get familiar with the data, and to gather key insights from it. Statistics serve as building blocks for core machine learning algorithms, and hence using statistics to summarize the dataset is an approach that data scientists generally adopt. This task of exploratory statistical analysis often involves repeated calculation of either different statistics on the same data, or same statistics on different data ranges. If we identify such repetitive access patterns, we can reduce computational overhead and increase the efficiency and accuracy of our system. But existing systems always calculate statistics from scratch, instead of building frameworks that opportunistically compute statistics to speed up the process of exploratory statistical analysis. This is a relevant problem today considering that datasets continue to grow, and the process of interactive exploratory data analysis is becoming increasingly more complex because of the amount of information required to be extracted from large data sets.

Data profiling is an important task of reviewing data from an existing source to understand its structure, content and relationships that will aid us in computing statistics or in collective informative summaries about the data. While there are many different data profiling tasks, in this project we intend to use the concept of data aggregation to handle single-column data profiling tasks that fall under the categories of ‘cardinalities’ and ‘value distributions’. This will enable faster and easier summarization of the data for data scientists and statisticians without incurring huge cost, or generating overhead with excessive hardware usage.

An important motivation comes from the paper by Jain et. al. [3] where they explored two workloads namely SQLShare and SDSS which are two database services with a large corpus of database queries. The authors studied three different types of repetitions- column repetitions (where the same column is queried for different statistics which might share the same basic aggregates), statistics repetitions (where the same statistics are calculated on different data columns using the same basic aggregates), and exact repetitions (hybrid of column repetitions and statistics repetitions). The results of the study are shown in the figure below, from which we can see that repetition is everywhere between 50% to 99%.

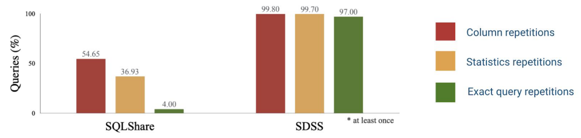


Figure 1: Exploratory workloads exhibiting repetition

3. Background and Related work

Wasay et. al. [1] proposed a system called Data Canopy where descriptive statistics were synthesized from a library of basic aggregates stored using segment trees. Their proposed system computes and caches the basic primitives of statistical measures to reduce redundant data movement. Results are synthesized for future queries to avoid having to repeatedly go back and trigger full pass on base data. They ensure that the data canopy could be built and updated regularly. Based on hardware properties, data size, and latency requirements, Data Canopy is capable of operating in one of three modes: offline, online, and speculative. Studies show that Data Canopy accelerates core machine learning classification and filtering algorithms like linear regression, bayesian classification, and collaborative filtering. Data Canopy scales with the number of columns and rows in the data set. Experimental evaluations showed an average speed-up of at least 10× after just 100 exploratory queries when Data Canopy was compared with state-of-the-art systems used for exploratory statistical analysis. The concepts put forth in this paper form the baseline of our project wherein we build the data aggregator for efficient memory and time usage.

In [2] the authors have introduced the readers to the domain of data profiling which goes hand-in-hand with exploratory data analysis. Data profiling also called data archeology is the statistical analysis and assessment of the quality of data values within a data set. The end goal of this process is to compute statistics or collecting informative summaries about the data. The paper focuses on relational data model to give a brief overview on the different profiling techniques that data scientists and statisticians often use. This being an ongoing area of research, the authors highlight some commercial tool available for the task, before proposing future work possible in the domain as per the current trends. Since the task of data profiling also revolves around dealing with huge parts of a dataset, we try adapting the idea of Data Canopy to the task of data profiling by initially only focusing on single-column profiling tasks.

4. Technical Approach

4.1 Data Aggregation

We have implemented the segment tree for caching, as proposed in the Data Canopy [1] paper. A segment tree is a tree data structure that stores data about intervals, or segments. It is represented as a binary tree whose leave represent data instances and internal nodes represent a union of elementary intervals. From the implementation point of view, for N data instances, a segment tree implementation will use a 2*N sized array.

A segment tree design for sum and sum of squares is shown in the figure below:

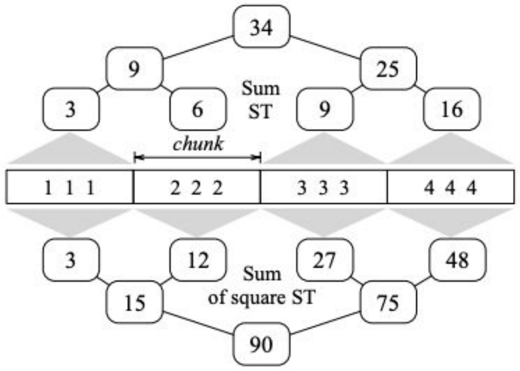
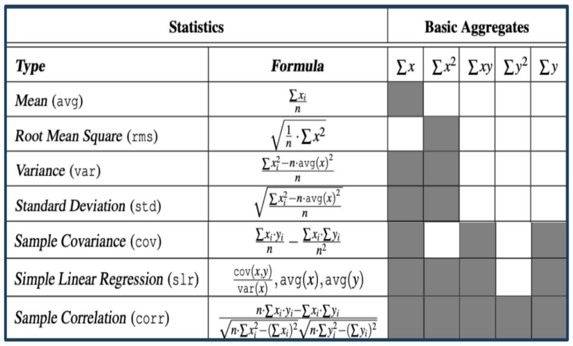


Figure 2: Segment tree representation for sum and sum of square

A segment tree is built for each basic aggregate for the entire dataset. A hashmap is implemented to map the incoming query to its corresponding segment tree. The unit as a whole is currently capable of handling queries over continuous data ranges, and updates to the data. This design helps in efficient computation of statistics like mean, variance, etc. by using the precomputed aggregates.

This idea of data aggregation using a segment tree was implemented for some of the basic aggregates that were used to compute some common statistics, as mentioned in the Data Canopy paper [1]. A table that clearly outlines the basic aggregates and corresponding statistics computed is shown below:



Statistics		Basic Aggregates				
Type	*Formula*	$\sum x$	$\sum x^2$	$\sum xy$	$\sum y^2$	$\sum y$
Mean (avg)	$\frac{\sum x_i}{n}$	■				
Root Mean Square (rms)	$\sqrt{\frac{1}{n} \cdot \sum x^2}$		■			
Variance (var)	$\frac{\sum x_i^2 - n \cdot \text{avg}(x)^2}{n}$	■	■			
Standard Deviation (std)	$\sqrt{\frac{\sum x_i^2 - n \cdot \text{avg}(x)^2}{n}}$	■	■			
Sample Covariance (cov)	$\frac{\sum x_i \cdot y_i}{n} - \frac{\sum x_i \cdot \sum y_i}{n^2}$	■		■		■
Simple Linear Regression (slr)	$\frac{\text{cov}(x,y)}{\text{var}(x)}, \text{avg}(x), \text{avg}(y)$	■	■	■		■
Sample Correlation (corr)	$\frac{n \cdot \sum x_i \cdot y_i - \sum x_i \cdot \sum y_i}{\sqrt{n \cdot \sum x_i^2 - (\sum x_i)^2} \sqrt{n \cdot \sum y_i^2 - (\sum y_i)^2}}$	■	■	■	■	■

Figure 3 :Table of statistics from the paper titled 'Data Canopy' by Wasay et. al.

4.2 Data Profiling

We concentrate on the analysis of individual columns in a given table. Metadata such as various counts: the number of values, the number of unique values and the number of non-null values are often part of the basic statistics gathered by a DBMS. From [2], it is seen that more advanced techniques use histograms of value distributions, functional dependencies and unique column combinations to optimize range queries.

We focus on implementing the following tasks, by concentrating on aggregating the metrics over a range of values in a single column:

1) Null values:

In this profiling task, we count the number of null values in a data column. For this, the data column which is the original array is converted into an intermediate representation which is yet another array. The rule followed to generate this array is that every NULL value in the original array is replaced by a '1' in the intermediate array while every other value is replaced by a '0' as shown in the figure below.

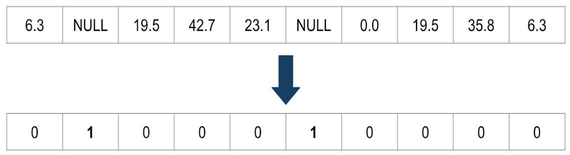

6.3 | NULL | 19.5 | 42.7 | 23.1 | NULL | 0.0 | 19.5 | 35.8 | 6.3
0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0

Figure 4: Counting the number of null values in a data column

2) Cardinality:

In this profiling task, we count the number of distinct values in a single column over a range of rows. For this, the data column which is the original array is converted into an intermediate representation which is yet another array. The rule followed to generate this array is that every NULL value in the original array is replaced by a '1' in the intermediate array while every other value is replaced by a '0' as shown in the figure below.

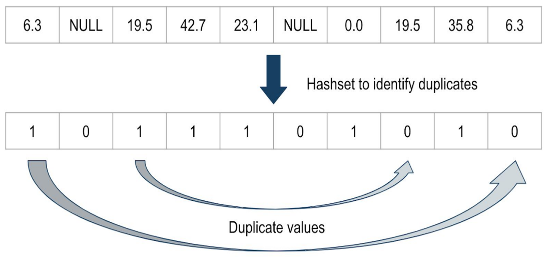

6.3 | NULL | 19.5 | 42.7 | 23.1 | NULL | 0.0 | 19.5 | 35.8 | 6.3
Hashset to identify duplicates
1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0
Duplicate values

Figure 5: Counting the cardinality (number of distinct values) in a data column

3) Uniqueness:

In this profiling task, we measure uniqueness of rows in the dataset, which can be computed using the formula given below:

$$\textit{Uniqueness} = \frac{\textit{No. of distinct values}}{\textit{No. of rows}}$$

Hence, the underlying task here is similar to that of cardinality as shown above.

4) Histogram:

Implement binning; depict frequency of occurrence of values. The implementation of histograms is divided into two sub-categories:

A] Equal-width histogram:

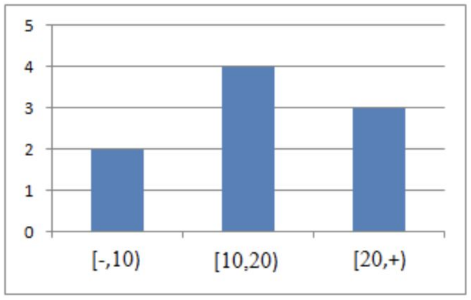


Figure 6: Example of an equal-width histogram

This is a single-column profiling task under the category of value distribution. The equal-width histogram handles querying over equal-sized range of values, and updates to the data. For this project, we have experimented with two different implementations which are explained below.

Method 1:
This method of histogram implementation is based on the concept of binning wherein it aggregates for base width 'w' and multiples of 'w'. This method is supported for data types int and float. In this method, every cell of the intermediate array represents a bin of the histogram for width 'w'. Hence the value within each cell denotes the count of numbers in each bin.

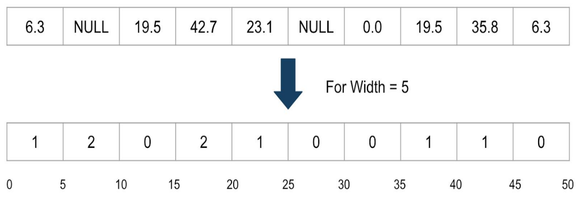


Figure 7: Binning intermediate array to generate equal-width histogram

Method 2:
This method of histogram implementation is based on the concept of inverted index wherein it aggregates for any bin-size i.e. width ‘w’. This method is supported for data type int only. In this method, every cell of the intermediate array represents a number while the value within each cell denotes the count of occurrence of that number in the original array i.e. data column.

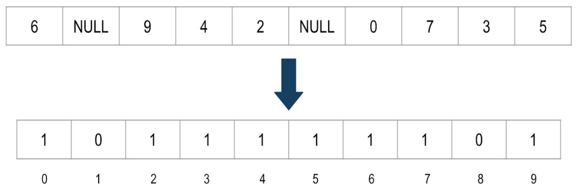


Figure 8: Counting occurrence of elements in a data column to generate equal-width histogram

B] Equal-height histogram

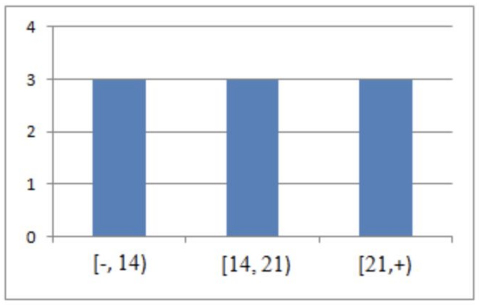


Figure 9: Example of an equal-height histogram

This is a single-column profiling task under the category of value distribution. The equal-height histogram handles querying equal count of values over dynamic-sized ranges, and updates to the data.

4.3 Persistence

Persistence is a method of efficiently storing data structures in such a way that they can continue to be accessed using memory instructions or via memory APIs even after the process that created or last modified them has ended. A way of including persistence to the system is by implementing the LRU cache.

LRU (Least Recently Used) cache helps us discard the least recently used items first, in order to make room for new data without having to keep all of the data in the main memory for the program to access.

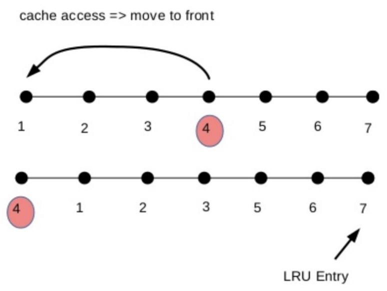


Figure 10: Working of an LRU cache

To implement the LRU cache we need to have some priori information such as the possible blocks that can be referred, and the cache size. We implement the LRU cache using two data structures: a queue, and a hash. The queue is implemented using a doubly linked list such that the maximum queue size is equal to the cache size. The most recently used blocks will be at the front end while the least recently used ones will be at the rear end. This approach is depicted in the figure shown above. The hash has the page number as the key and the address of the corresponding queue node as value.

When a page is referenced, the required page may be in the memory. If it is in the memory, we detach the node from its current position in the list, and bring it to the front of the queue. If the required page is not in the memory, we bring it in to memory. In other words, we add a new node to the front of the queue and update the corresponding node address in the hash. If the queue is full, i.e. all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.

4.4 B-Trees

B-Tree is a self-balancing multi-way search tree which is useful when all of the data being dealt with cannot be fit into the main memory. Generally when the number of keys is high, data is read as blocks from the disk. However the disk access time is very high when compared to the memory access time. The use of B-Trees will help us reduce the number of disk accesses by reducing the height of the tree, and instead making it a fat tree. The height of the B-tree is kept low by putting maximum possible keys in a tree node, the size of which is often kept equal to the size of the disk block.

Some common operations for the B-tree that we implemented are as listed below:

(1) Traversal:

The traversal of a B-tree is similar to the inorder traversal in case of a binary tree, wherein we start from the leftmost child, recursively print the leftmost child, then repeat the same process for remaining children and keys before recursively printing the rightmost child.

(2) Search:

The search operation for a B-tree is similar to the search operation in binary search tree.The search starts at the root and recursively traverses down. If we find the node, we return it. If we reach the leaf node and still don't find it, we return NULL.

(3) Insertion:

The insertion of a key always happens at the leaf node after ensuring that the node has extra space. This is a proactive inversion algorithm where we split the node if it is full before going down to a node. The advantage is that we never traverse a node twice. However, the disadvantage is that we might create unnecessary splits.

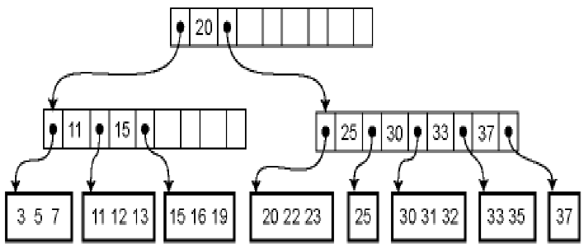


Figure 11: Example of a B-Tree

We have implemented a B-Tree for this project, and aim to incorporate it with our data-aggregation approach in the future to aid efficient retrieval of stored data in a block-oriented storage context.

5. Performance Evaluation and Analysis

1) Segment tree construction (array size = 10k)

The table below shows the time in microseconds that it takes for constructing segment trees for the three main basic aggregates, as mentioned in the Data Canopy paper.

Base aggregates	Construction time (micro-seconds)
Σx	170
Σsquare(x)	192
Σxy	82

Table 1: Time taken for segment tree construction for basic aggregates

2) Query time analysis for histograms

A] Equal-width histogram

We study the effect on query time as we increase the width of histogram being queried, and we observe the following trend.

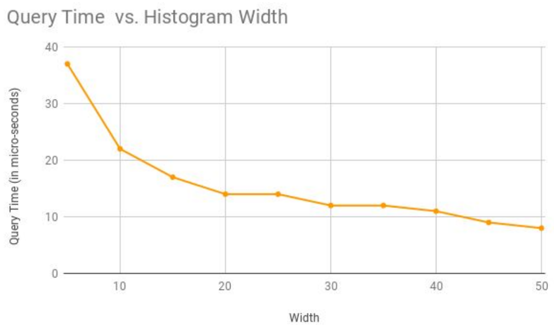


Figure 10: Query time vs. histogram width

We see that the query time decreases as we increase the width of the equal-width histogram being queried. This is totally justified because for a larger query-width, the segment tree is queried at higher levels closer to the root of the tree. Hence it doesn't take time to traverse the segment tree till the leaf level to find values. This is a good example of why data aggregation must be used to reduce redundancy and to speed up query retrieval in data exploration tasks.

B] Equal-height histogram

We study the effect on query time as we increase the height of histogram being queried, and we observe the following trend.

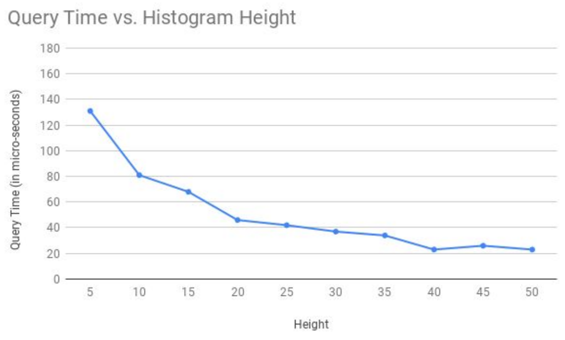


Figure 11: Query time vs. histogram height

We see that the query time decreases as we increase the height of the equal-height histogram being queried. This is totally justified because for a larger query-height, the segment tree is queried at higher levels closer to the root of the tree. Hence it doesn't take time to traverse the segment tree till the leaf level to find values. This is yet another good example of why data aggregation must be used to reduce redundancy and to speed up query retrieval in data exploration tasks.

3) Comparison: Traditional vs. Segment tree implementation

A] Time taken for querying segment tree (array size = 10k)

The table below compares the time taken for different profiling tasks when implemented in the traditional manner wherein the certain statistics are repetitively calculated vs. the time taken for the same tasks when implemented with data aggregation using segment trees.

	Query Time (in microseconds)	
	Traditional approach	Data Aggregation using Segment Tree

Equal width histogram	200281	111
Equal height histogram	681	69
Null values	20	8
Distinct values	7546	74

Table 2: Time taken for querying segment tree for different profiling tasks

From the above table we can see that there is a significant time improvement in runtime for our implementation as compared to the traditional approach. This shows that our approach is indeed very valuable since it will help data scientists save time when conducting exploratory data analytics on the datasets before embarking upon any project.

B] Speedup for computation of different statistics

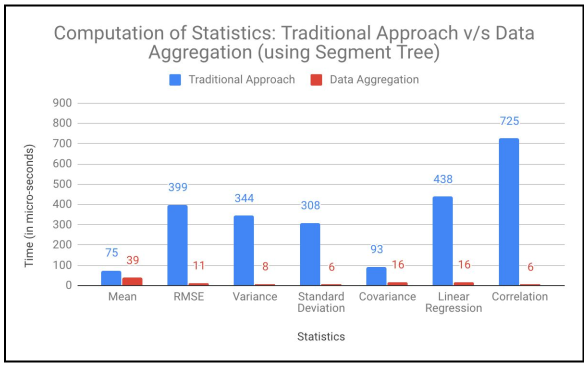


Figure 13: Speedup in computations of different statistics using segment tree

This chart shows the speedup observed for our approach vs. the traditional approach while calculating different statistics that are very common in data exploration tasks. The significant difference in time taken again supports our claim that the segment-tree based approach is more effective.

C] Correlation operation: Query time for varying input size

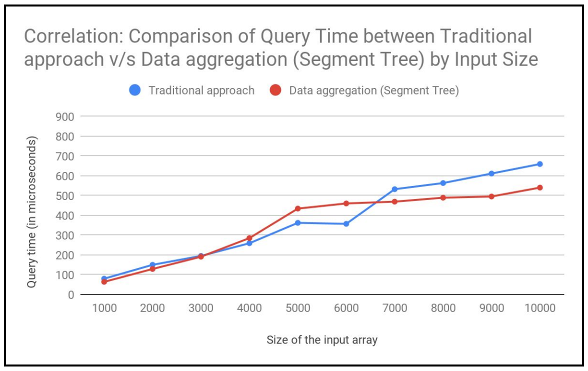


Figure 14: Query time comparison for computation of correlation

One of the most complex operations of the lot mentioned in the Data Canopy paper, which makes use of all basic aggregates. Hence, we thought of evaluating the performance of our system and the traditional approach for this particular statistic. Our observations are depicted in the above chart. We can see from the overall trend observed that the data aggregation based approach is better than the traditional approach most of the times, and the improvement in query time keeps increasing gradually as the size of input array increases.

D] Mean operation: Differences in speed for varying input size

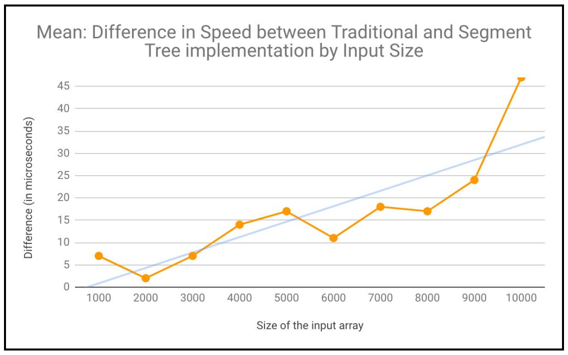


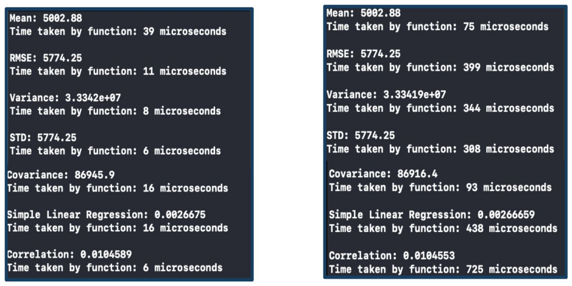
Figure 15: Differences in execution speed for varying input size

Another statistical calculation of interest to us is the mean of a data column. In the above figure we plot the differences in speed for varying size of input array to the segment tree. We see a linearly increasing overall trend, and we can also see that the differences between the consecutive two data points indicating a difference of about 1k data instances keeps increasing as the input array size increases.

6. Evaluating correctness

We believe that another way to evaluate our approach and the system we built, is to compare the values of the different statistics being computed to check if the values are same. The results observed can be viewed in the figure below where the screenshot to the left indicate results obtained for our approach, while the screenshot to the right has results for the traditional approach.

We can see in the figure below that the values are mostly the same for the different statistics computed. We see a significant difference in covariance, but since the values for underlying aggregates were consistent with the traditional approach, we believe that the difference in the values are due to the float values in the dataset.



```
Mean: 5002.88
Time taken by function: 39 microseconds

RMSE: 5774.25
Time taken by function: 11 microseconds

Variance: 3.3342e+07
Time taken by function: 8 microseconds

STD: 5774.25
Time taken by function: 6 microseconds

Covariance: 86945.9
Time taken by function: 16 microseconds

Simple Linear Regression: 0.0026675
Time taken by function: 16 microseconds

Correlation: 0.0104589
Time taken by function: 6 microseconds
```

```
Mean: 5002.88
Time taken by function: 75 microseconds

RMSE: 5774.25
Time taken by function: 399 microseconds

Variance: 3.33419e+07
Time taken by function: 344 microseconds

STD: 5774.25
Time taken by function: 308 microseconds

Covariance: 86916.4
Time taken by function: 93 microseconds

Simple Linear Regression: 0.00266659
Time taken by function: 438 microseconds

Correlation: 0.0104553
Time taken by function: 725 microseconds
```

Figure 16: Evaluating correctness of our approach

7. Conclusion and Future Work

In this project, we started out with the hypothesis that the segment-tree based data aggregation approach will help us reduce query-time for the different data profiling tasks. We were inspired by the Data Canopy paper by Wasay et. al. and we tried to replicate their results with our implementation. We extended the project to then include other profiling tasks as mentioned in the paper by Abedjan et. al.

From the results we observed, we can conclude that our approach is indeed more efficient in terms of time taken for the repetitive operations that data scientists often perform. However, a downside to our approach is that segment trees take up a lot of storage space, depending on the size of the segment tree. To address this issue, we plan to try and persist the segment trees for efficient management. We believe that we could also incorporate the concept of multithreading later on, to facilitate multiple user access to the same database and segment trees. We are also interested in trying to extend this approach for other interesting statistics where the concept of data aggregation might prove useful, including approximate queries. We are also keen on experimenting with inverted or adaptive indexing of trees for implementing the histograms, as future work for this project.

References

[1] Wasay, A., Wei, X., Dayan, N., & Idreos, S. (2017, May). Data canopy: Accelerating exploratory statistical analysis. In Proceedings of the 2017 ACM International Conference on Management of Data (pp. 557-572). ACM

[2] Abedjan, Z., Golab, L., & Naumann, F. (2015). Profiling relational data: a survey. The VLDB Journal—The International Journal on Very Large Data Bases, 24(4), 557-581

[3] Jain, S., Moritz, D., Halperin, D., Howe, B., & Lazowska, E. (2016, June). Sqlshare: Results from a multi-year sql-as-a-service experiment. In Proceedings of the 2016 International Conference on Management of Data (pp. 281-293). ACM

[4] Segment Tree Implementation https://www.geeksforgeeks.org/segment-tree-efficient-implementation

[5] LRU Cache Implementation https://www.geeksforgeeks.org/lru-cache-implementation

[6] B-Tree Implementation https://www.geeksforgeeks.org/b-tree-set-1-insert-2