ANSWERING A LARGE NUMBER OF AGGREGATE QUERIES EFFICIENTLY

An Honors Thesis

Presented by

Caroline KyuMin Kim

Completion Date:

May 2019

Approved By:

Gerome Miklau, College of Information and Computer Sciences

Marco Serafini, College of Information and Computer Sciences

ABSTRACT

Title: **Answering a Large Number of Aggregate Queries Efficiently**

Author: **Caroline KyuMin Kim**

Thesis/Project Type: **Independent Honors Thesis**

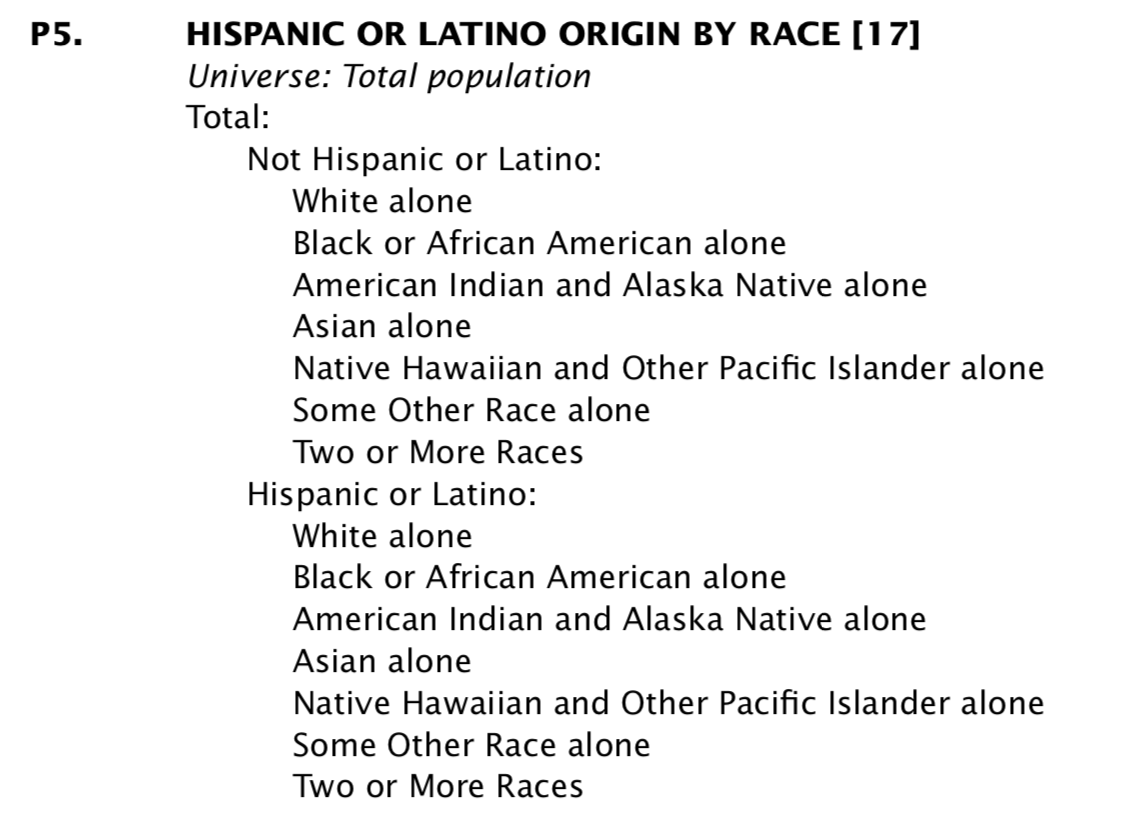
Approved By: **Gerome Miklau, College of Information and Computer Sciences**

Approved By: **Marco Serafini, College of Information and Computer Sciences**

Aggregate queries are frequently used over databases to perform calculations over the dataset. Because this function is being used more and more, methods that would optimize the time of this were explored. By using the census data from the year of 1940 and counting queries generated by the tables from Census Summary File of 2010, 4 different methods were explored. The counting queries were evaluated by running one query at a time in a relational database system, creating one CUBE query that will answer all the workload queries, vectorizing the dataset and the workload queries, and using MapReduce. These 4 methods were used with different sizes of the databases and number of workload queries and were compared to determine which method optimizes the time the most in different conditions.

**1. Introduction**

Through this research, different methods that answer a set of large number of aggregate queries were explored to find which method would be the most effective depending on the size of the dataset. An aggregate query is a query that performs a calculation on the dataset and returns a single value. This is a method that is used in daily life such as computing average value, counting number of certain elements, finding maximum or minimum value, or computing sum or standard deviation. But for this project, the focus was only on the predicate counting queries. Predicate counting queries are used in many different areas to give a count of objects that satisfy certain conditions. For example, if we were to create a histogram for the height of orange trees, we would need the count of orange trees that have height between 100 cm to 150 cm, 150 cm to 200 cm, 200 cm to 250 cm, and so on. These would be found by running predicate counting queries with these restrictions in the dataset of orange trees. Aggregate queries like these are common in databases, and there are ways to speed up the process up for a single query such as by using indexes. But answering many aggregate queries simultaneously is challenging.

To solve this problem, this research was conducted to find how to efficiently run a large collection of aggregate queries on a database. The goal of this research is to efficiently run all queries on a table when given the input of a large collection of aggregate queries Q and a database which will be a single table R. For this research, Person table from the Census Database from 1940 was used as the dataset. This table is 2159 MB big, and holds record of 15 different attributes including sex, age, race, relation, household, and etc. for 19,999,999 people. Out of the 15 attributes that this table has, only 4 columns were used for this research. The sex attribute stores smallint of 1 for male and 2 for female. The age attribute stores smallint ranging from 0 to 114 that corresponds to a person’s age. The race attribute stores smallint from 1 to 6 where 1 means “White alone”, 2 means “Black or African American alone”, 3 means “American Indian and Alaska Native alone”, 4 means “Asian alone”, 5 means “Native Hawaiian and Other Pacific Islander alone”, and 6 means “Some Other Race alone”. And hispan attribute store smallint from 0 to 4 where 0 means that they are not Hispanic or Latino, and 1-4 means that they are Hispanic or Latino. The workload queries were generated from the 2010 Census Summary File. This file has Population Tables that show possible combinations of different conditions. For example, table P5 from the file breaks the total population down by “not Hispanic or Latino” and “Hispanic or Latino” and then each of those by possible races and a category of “two or more races”, creating 17 different possible combinations of the attributes. 

From these Population Tables, the total of 3460 counting queries were generated to be used as the workload of this research. Because the 1940 Census data did not put into account for possibility of a person having more than one race, the “two or more races” queries were ignored from the Population Tables.

With this dataset and workload, 4 different methods were evaluated: Query-at-a-Time, CUBE Query, Vectorization, and MapReduce. For the Query-at-a-Time method, all the workload queries were executed one at a time in relational database management system with and without an index. For the CUBE Query method, one CUBE query was generated. And from the result of that CUBE query, all the workload queries were answered. For the Vectorization method, the dataset and the workload has been vectorized. The dataset vector contains 2760 rows which corresponds to all possible combinations of the 4 attributes, and 1 column that contains the counts of each combinations. The query vector contains 1 column that contains 1 for True and 0 for False, and 2760 rows that corresponds to the same combinations of the 4 attributes as the dataset vector. All the workload queries are converted to query vector and put into one matrix. Then the dot product of this matrix and the dataset vector is taken to return answers for all the workload queries. For the MapReduce method, each row of the dataset was mapped by the 4 attributes. Then it was reduced by matching the same attribute conditions and adding the count. From the reduced list, the workload queries were answered.

These 4 methods were evaluated with the full dataset, then on the same dataset with the first 100 rows, then with the first 10,000 rows. The time of running these methods on the different sizes of the dataset were recorded and compared to find the most effective method of answering these aggregate queries.

2. Significance

The aggregate queries are frequently used for many different purposes. Even though running one aggregate query is a very simple process, to run a large set of aggregate queries simultaneously, there are a lot of complications. Depending on the size of the set of aggregate queries and on the size of the dataset, the process can take a really long time to answer all the workload queries. So through this research, the different methods were explored to see which method will optimize the time of the process the most. This research can be extended to run the same methods with aggregate queries other than the counting queries and with different database systems. And the process of answering a large set of aggregate queries will be optimized despite the different environment.

**3. Background**

In the paper *Efficient Implementation of Data Cubes via Materialized Views*, Online Analytical Processing (OLAP) is described. OLAP is an approach for conducting multidimensional analysis of data. It is a relational database system, and it lets the users directly query the raw data. The multidimensional databases retain significant performance advantage. And performance in relational database systems can be improved dramatically by materializing the data cube into summary tables. Data cubes are special-purpose DBMS for storing multidimensional data and handling queries that aggregate over some dimensions. And views are projections of the cube onto some of its dimensions. Each query have a natural view, but with views that group by more attributes, more can be answered. However, those views are larger and require additional cost. This paper explains the greedy algorithm for selecting optimal view is explained. This algorithm assumes that the top view is materialized. Then it selects additional views to materialize one at a time, until some total cost of selected views is reached. And at each step, it selects the view that most reduces the average cost of answering a query per unit space.

In the paper *The Matrix Mechanism: Optimizing Linear Counting Queries Under Differential Privacy,* the matrix mechanism, linear query, and the query matrix are explained. The matrix mechanism is an algorithm for answering a workload of linear counting queries that adapts the noise distribution to properties of the provided queries. This mechanism uses a different set of queries which are answered using a standard Laplace or the Gaussian mechanism. Linear query is an aggregation query over a single relation that can be expressed as a linear combination of a set of database counts. And query matrix is a collection of m linear queries. Other details of privacy mechanisms in this paper are not relevant. But the vector representation of the table used in this research is important since it will be used in my research. This paper uses single query evaluation as dot product of query vector and data vector, and multiple query evaluation as matrix multiplication of query matrix and data vector.

The use of continuous queries is discussed in the *NiagaraCQ* paper. Continuous queries allow users to obtain new results from a database without having to issue the same query repeatedly. It can be useful to work with database that is frequently changing. It also introduces incremental grouping methodology. In this method, dynamic regrouping is used in cases of reduction in the overall performance of the system to re-establish their effectiveness because the quality of the group can deteriorate over time. Also, new queries are added to existing groups can optimize without having to regroup already installed queries.

A paper on Spark SQL was reviewed because the usage of Spark is included in my research. This paper explained about the relational processing, declarative queries, and optimized storage. And it also showed how SQL users can call complex analytics libraries in Spark. Spark SQL supports relational processing both within Spark programs and on external data sources, provides high performance using established DBMS techniques, supports new data sources, and enables extension with advanced algorithms. It also explains the use of catalyst optimizer which makes it easier to add new optimization techniques and features to Spark SQL and enables external developers to extend the optimizer.

**4. Methodology**

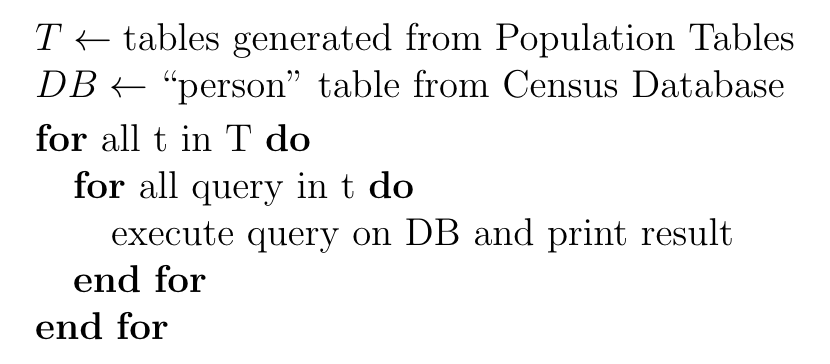
For this research, Python was used to code along with PostgreSQL and Spark. Everything was tested on a MacBook Pro with 2.3GHz Intel Core i5 processor and 8GB of memory. A tunnel is created to Yeeha to login remotely to PostgreSQL and connect to the Census Database. Then, Psycopg2 library is used to connect the Python code to the database.

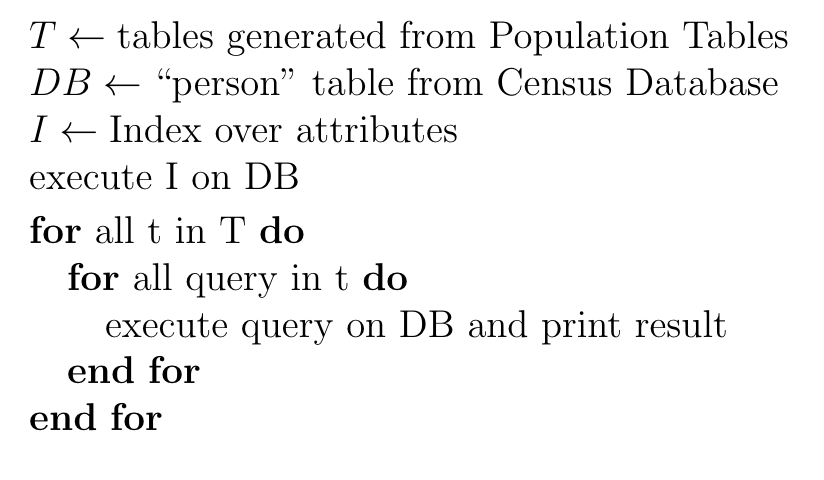
There is a CensusTable class that represents a Census Summary File Population Table of queries. This class has methods for creating format of the Population Table and uses Sqlalchemy and Anytree libraries. In this class, orthogonal level of a predicate can be added on to the table, the root of this table can be selected, and the table can be rendered into a String of description or of SQL in a tree formation to match the design of each Population Table.

Then, using the CensusTable class, classes for each Population Table used were created. For example, table P5 from the 2010 Census Summary File was used. This table has a root of “total population”. Then it branches to “Not Hispanic or Latino” and “Hispanic or Latino”. From these two branches, it branches once more into the 6 different race categories. So in the class for P5, the root of this CensusTable object was set to the “total population”, and orthogonal level of “Hispanic” and “not Hispanic” predicates were added where these predicates had the predicates of each of the 6 race categories as their children. Then, this class was built and rendered into SQL. Such classes were created for P1, P3, P4, P5, P8, P9, P10, P11, P12 (A-I), and PCT 12 (A-O). And with these tables, 3460 SQL queries for the workload were generated.

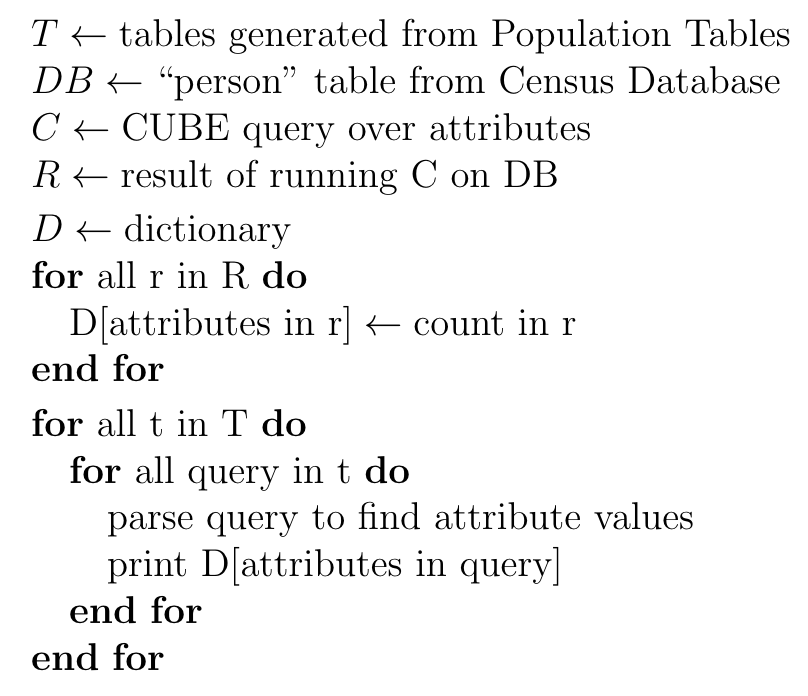
With this workload, 4 methods were tested for their efficiency according to time. And for all 4 of the methods, each were evaluated with the dataset of 100 rows, 10,000 rows, and 19,999,999 rows to compare the effect of dataset size as well.

**4.1 Query-at-a-Time**

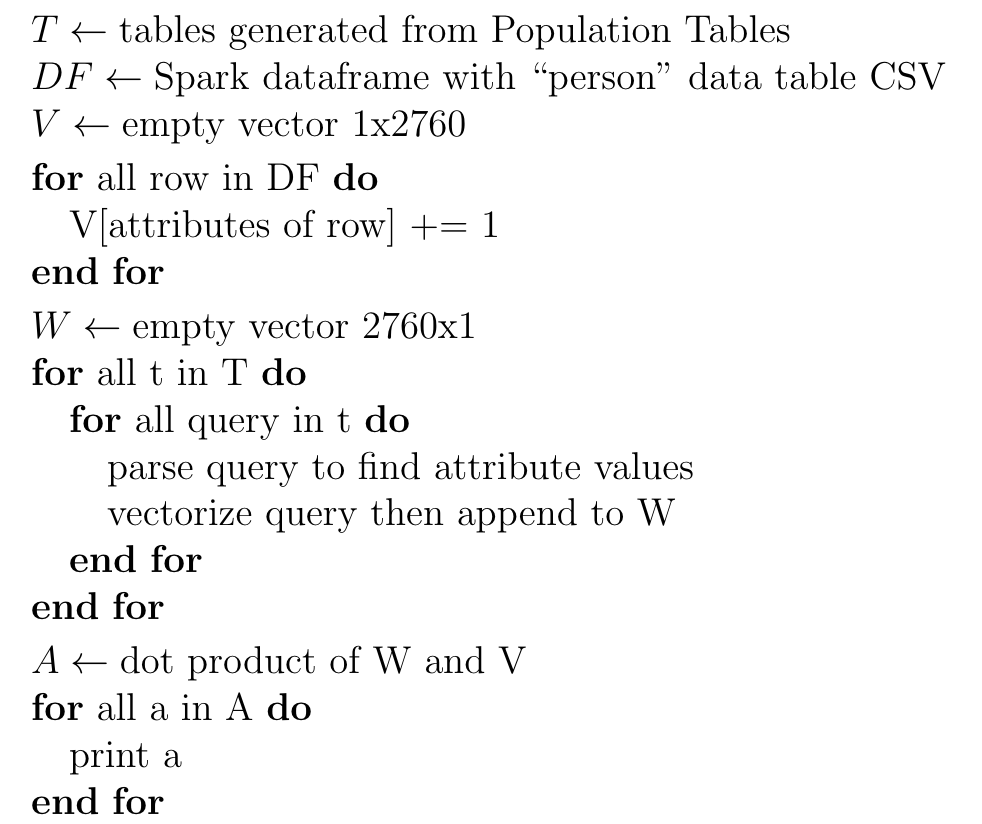
The most simple way of running the set of aggregate queries is to execute each of the queries one at a time. This method was tested in a class called RDBMS. In this class, the database was connected through Psycopg2. Then each of the Population Tables were built. For each of the Population Tables, all the queries generated from the table were executed on the database. For each of the results the queries got, it printed the count number to the console. Starting from the point of iterating through each table that was built to the point of finishing printing out the results, the time was measured and was recorded.

Continuing with the first method, another class named IndexRDBMS was created. In this class, it did the exact same thing as in the RDBMS class except for the fact that an index was created for this database table. An index was created on the “person” table over the attributes of “sex”, “race”, “hispan”, and “age” to speed up the process of this method a little bit. In this class, the time was measured starting from the point of creating the index to the end of printing out all the results.

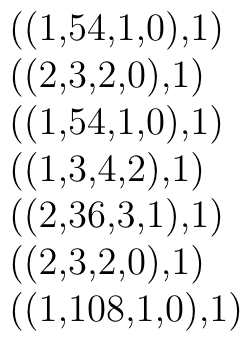
**4.2 CUBE Query**

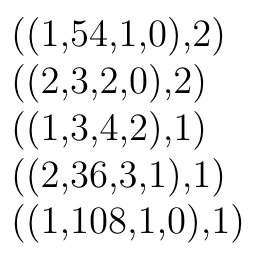
This method of creating a grouped query with the CUBE function was tested in a class named GoupedRDB. After connecting to the database and building each Population Tables, one SQL statement was executed for this class. “SELECT sex, race, hispan, age, count(\*) FROM people GROUP BY CUBE(sex, race, hispan, age)”. This CUBE query was executed to retrieve the count of all possible combinations of the 4 attributes. A Python dictionary was created, and for each row of the result from the CUBE query, the key was created in the dictionary to match the condition of each attributes, and the value was created to store the count retrieved by the CUBE query. Then, for every Population Table built, the SQL String of each query was parsed to find the conditions of each attribute in the query. If the attribute was not mentioned in the WHERE clause, it was just set as “None”. For example, for a SQL query such as “SELECT count(\*) FROM person WHERE sex = 1 AND race = 3”, this String was parsed to retrieve “sex = 1”, “race = 3”, “age = None”, “hispan = None”. With this information, the dictionary was searched and the corresponding number of count was returned. For this method, two different time measurements were taken. The first time measurement was measured from the point of the CUBE query being executed to the point of printing out all the results for each workload query in the console. The second measurement started from the same point as the first measurement, but the time taken to parse the SQL String was taken out.

**4.3 Vectorization**

The method of vectorizing the dataset and the workload was tested in a class named Vector. In this class, instead of connecting to the database through Psycopg2, the CSV file for the “person” table in the Census Database was used in order to use Apache Spark. By using Findspark and Pyspark libraries, SparkSession was built, and the Spark Dataframe was created using the “person.csv” file saved locally. Using the Numpy library, an array was created with the size of 2760 x 1. For each combination of the attributes, the count was retrieved and stored in this array in the corresponding row. Again, using the Numpy library, another array was created to for the workload queries. Each workload queries were stored into an array size of 1 x 2760 by storing 1 if the workload query corresponded with the combination of the attributes associated in that column, and 0 if it didn’t. These Numpy arrays were appended to one another for all the queries in the workload to create a matrix with the size of 3460 x 2760. Then, the dot product of these two are taken to produce an array that is the answer to all the queries in the workload with the size of 1 x 3460. For this method, two different time measurements were taken. The first time measurement was taken from the point of vectorizing the dataset to the end of printing out all the results of query answers to the console, and the second measurement from the point of retrieving the dot product until the point of printing out all the results of query answers to the console.

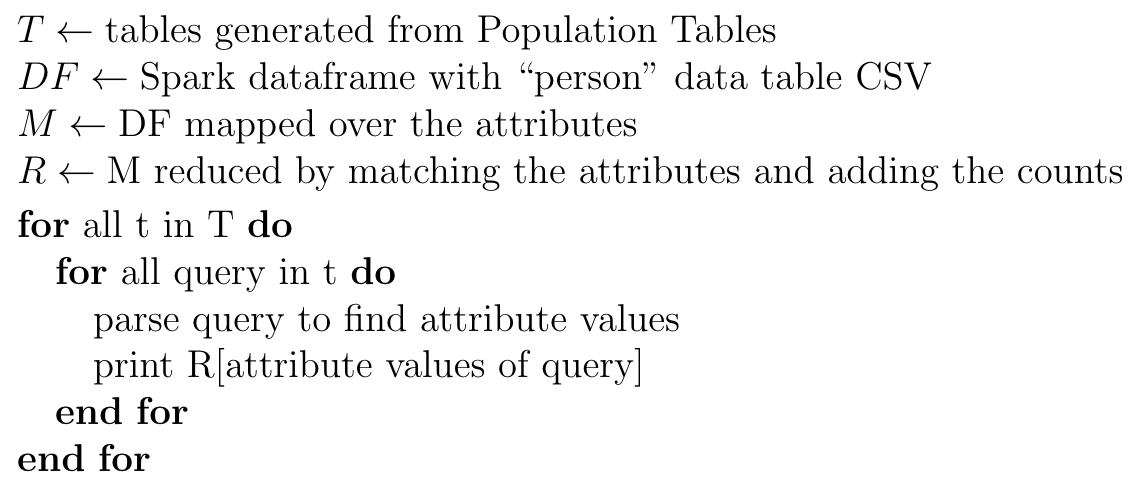
**4.4 MapReduce**

For the last method MapReduce, two different approaches were taken. For both of the approaches, Apache Spark and the local CSV file for the “person” table was used as in the Vector class. And the same MapReduce functions were used for the both approaches. The list created by the Map function has keys being tuples of numbers corresponding to the attribute values in the order of “sex”, “age”, “race”, “hispan”. The values of the map are all 1. 

Then this result goes through a Reduce function where it matches the keys together and adds the value up together for each matching key. 

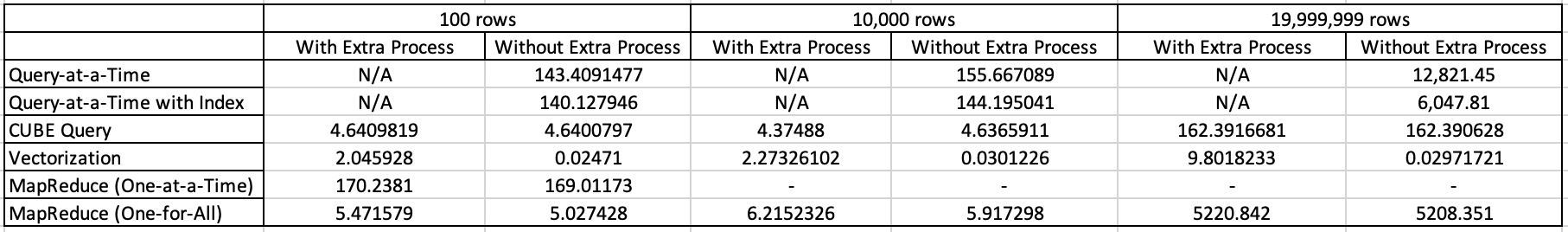
This result was searched through to find the match for the query in the workload. For any of the keys that matched the condition of the query, the values were added together to produce an answer for the query.

The first approach was to do MapReduce per query in the workload. This was done within the MapReduce class. In this class, per query in the workload, the SQL String was parsed to find attributes are specified in the WHERE clause. Then, according to those attributes specified, a map was created using those attributes in tuples as keys over the dataset. So instead of mapping over all four of the attributes, it only maps over the attributes that are explicitly given a value in the WHERE clause. The values of the map were all set to 1. Then, it was reduced by matching the same keys and adding up the values. From this reduced list, the answer for the query was found and printed onto the console.

The second approach was to do one MapReduce for the whole class, and this was done in the MapReduce2 class. In this class, one map was created with tuples of 4 attribute conditions as key over the dataset. This was then reduced by matching the keys and adding up the counts. Then, by iterating through the queries in the workload, each SQL query was parsed to find the conditions of attributes in the WHERE clause. Then correct value from the reduced list was printed out in the console. For both of the approaches, two different time measurements were taken. Both of the time measurements started with the map being created. But the second measurement took out the time taken to parse the SQL String.

**5. Results**

For each of the 4 methods, time of retrieving the answer of all workload queries was used as the main metric. For all the methods except for the Query-at-a-Time method, it needed extra process of handling the dataset and the workload to be used for the desired methods. For example, the SQL String had to be parsed to find the conditions of each attributes. And for the Vectorization method, the dataset and the workload had to be vectorized into the correct design that will optimize time and space. So for these, two different sets of time measurement was taken, with and without the time of the extra process. And all of the methods were evaluated with the same dataset but with 100 rows, 10,000 rows, and 19,999,999 rows.

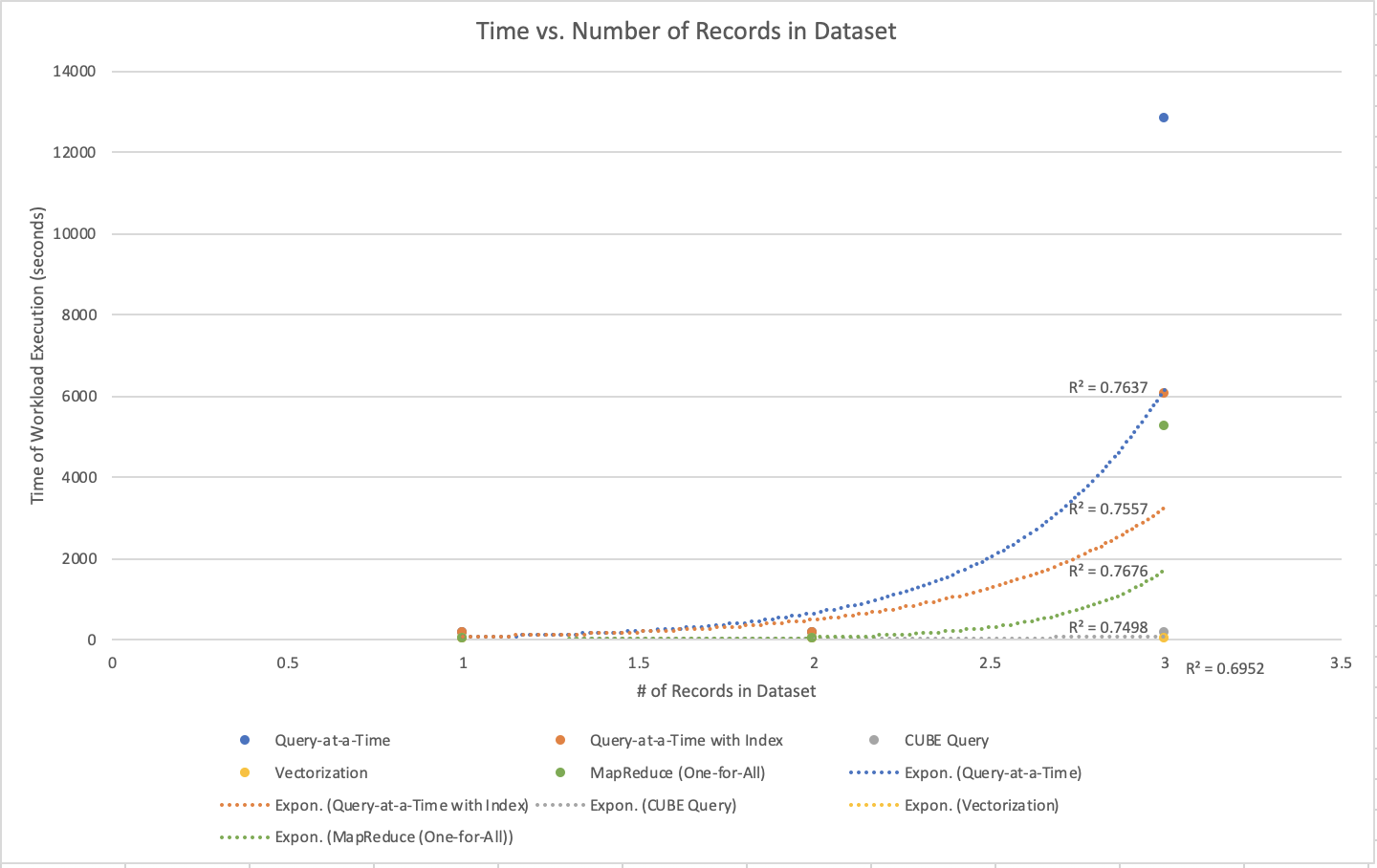
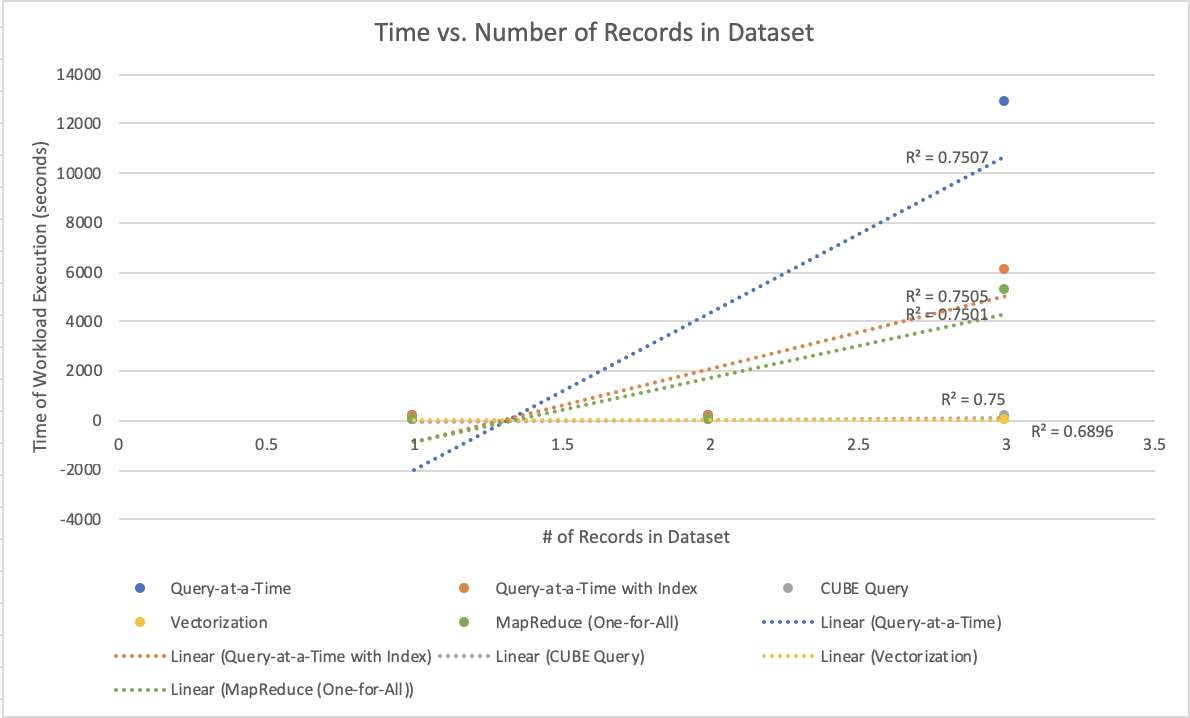


Vectorization came out to be the fastest method of all with or without the extra process. From the results, we can see that the extra process of parsing the SQL strings and storing data into a list or dictionary to perform different methods did not hugely impact the time. But for the Vectorization method, it added significant amount of time to for it to vectorize the dataset and the workload queries to the time of just taking a dot product of the two queries. Also for the Vectorization method, we can see that despite the size of the dataset, the time measurement for the dot product of the vectors remain pretty much constant. So for running a set of aggregate queries with similar sizes of workload and the dataset as this, as long as the design of the vectorization is well chosen, this method will perform the fastest.

**5.1 Index Creation**

Comparing the two Query-at-a-Time methods, we see that when the dataset only has 100 rows, the difference between the time is not too big. But when the number of rows increases to 20 million, we can see that the method that uses an index is twice as fast than the one without the index. From this, we can conclude that if we are running a large set of aggregate queries on a smaller dataset, creating the index will not be in effective method to optimize time. But as the dataset grows, using an index is one way to optimize time.

**5.2 Size of the Dataset**

In this research, only the dataset with 100 records, 10,000 records, and 19,999,999 records were explored. But from the results we have collected, assumptions for dataset with greater number of records can be made. The results collected have been put on a scatter plot and was given a line of regression in linear form and in exponential form. 

Even though it is a slight difference, R2 value for the exponential form is higher than for the linear form for all the methods except for the CUBE Query method. So as the number of records in the dataset grows, we can conclude that the Query-at-a-Time method with and without index, and the MapReduce method will be very inefficient to use to run a large set of aggregate queries on. But the CUBE Query and the Vectorization methods can be explored more as the CUBE Query method fits the linear regression better and as the Vectorization method does not show any noticeable change in the slope compared to the other methods.

Also, in some of the methods, the dataset was locally saved as a different type of data structure. For example, in the MapReduce method, each row of the dataset has been put in a list from the Map and Reduce functions. So if the number of records in the database was to grow significantly, it would require significantly larger memory space for these methods.

**5.3 Size of the Workload**

To look at the trend in how the number of the queries in the workload might affect the runtime, we can look at the number of iterations made on the workload in all the methods. In each of the methods, the workload was only iterated through once. But during the iteration of the workload, the process taken for each methods are a little different. For the Query-at-a-Time method, the workload was iterated through to execute that query. For the CUBE Query method and the MapReduce method, the queries in the workload had to be searched through another data structure to match attribute conditions and receive a value. And for the Vectorization method, the queries in the workload had to be vectorized during the iteration. Out of these different processes, executing the query takes the longest, then searching through another data structure, then the vectorization of the query. So even though all the methods only use one iteration through the workload, because of the process that takes in place during the iteration, Vectorization will be the fastest method as the number of the queries in the workload grows.

**5.4 Size of the Domain**

Another variable that this research can be evaluated for is the size of the domain. Depending on the number of attributes the workload is associated with, and the domains of those attributes, the time and the memory each method takes can vary. In this research, only 4 attributes “sex”, “age”, “race”, “hispan” were used. The size of the domain of these attributes were 2, 115, 6, and 5 respectively. But if more attributes were associated with the workload, and if the size of the domains were greater, the results of each method would be affected by it. For example, the Vectorization method uses all the possible combinations of the attributes. The dataset vector came to have the size of 2760 x 1 because 2760 is the number generated by the size of the domains in each attribute multiplied together. So if the size of the domain were greater, more time and space would be needed for the Vectorization method. This is the same for the CUBE Query method. When a CUBE query is executed, it gathers the answer to the aggregate function for all the possible combinations of the attributes. So if the size of the domain were greater, the result of the CUBE query would be longer. And because these are stored in a local dictionary, it would take more space for this larger dictionary and take more time to search through this dictionary.

**6. Conclusion**

From this research, it was shown that Vectorization method out of the 4 methods tested optimized the time the most to answer a large set of counting queries with Census Database from 1940 and the 3460 queries generated from the 2010 Census Summary File Population Tables. It was found that creating an index and running one query at a time is more effective as the number of records in the dataset grows when only that method is compared with the usage of index. But overall, it was found that the Query-at-a-Time method and the MapReduce method will not be effective for the dataset with larger number of records. Also, it was found that as the number of queries in the workload grows, there is not much difference in ranking of the methods. The number of iteration through the workload is the same across all the methods. So the effectiveness of each method will not be different with the change in the size of the workload. Lastly, it was found that the size of the domain affects the time and space that the CUBE Query method and the Vectorization method takes. As the size of the domain grows, these two methods perform slower and use more space for memory.

For each method, the boundary between the process of retrieving the answers for the workload queries itself and the process of doing the initial work according to the dataset and the workload queries to use certain methods was not clear. So, if I were to start this research over from scratch, I would define that boundary clearly and code separately for the preparation and the answer retrieval so that the time measurement would come out to be two separate measurements. This way, the results will more accurately output the time it took for main part of the method used, and this result will be more relatable when the methods are applied to different datasets and workloads.

During this research, only one dataset and workload was used to collect data from. If more set of datasets and more set of workloads were used, we would have been able to make the comparison between different sizes of the dataset, workloads, and domains. And from those results, the optimization over many variables could have been analyzed.

With the results from this research, more research can be done in this area. The similar approach can be used with a database other than the Relational Database Management System to find the most effective method for answering a large set of aggregate queries. Also, the 4 different methods could be tested out with other aggregate queries other than the predicate counting queries. And lastly, some system can be developed so that these methods can be applied to any different types of dataset and workload. With these results, this method of answering a large set of aggregate queries that optimize time and memory space for different environment the database system is set up in and for different dataset and workload inputs.

**7. References**

*The Matrix Mechanism: Optimizing Linear Counting Queries Under Differential Privacy*

*Li, Chao, et al. “The Matrix Mechanism: Optimizing Linear Counting Queries under Differential Privacy.” The VLDB Journal, vol. 24, no. 6, 16 July 2015, pp. 757–781., doi:10.1007/s00778-015-0398-x.*

*Efficient Implementation of Data Cubes via Materialized Views*

*Harinarayan, Venky, et al. “Implementing Data Cubes Efficiently.” Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data - SIGMOD '96, 1996, doi:10.1145/233269.233333.*

*NiagaraCQ: A Scalable Continuous Query System for Internet Databases*

*Chen, Jianjun, et al. “NiagaraCQ.” ACM SIGMOD Record, vol. 29, no. 2, Jan. 2000, pp. 379–390., doi:10.1145/335191.335432.*

*Introduction to Spark and Spark SQL Paper*

*Armbrust, Michael, et al. “Spark SQL.” Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15, 2015, doi:10.1145/2723372.2742797.*

*U.S. Census Bureau, 2010 Census of Population and Housing, Demographic Profile*

*Summary File: Technical Documentation, 2011.*