Accelerating Aggregation Efficiency: Using Postgres as a Cache with MongoDB

Senior Project Submitted to
The Division of Science, Math, and Computing
of Bard College

by

Mason Porter-Brown

Annandale-on-Hudson, New York
December 2021

Acknowledgments

I would like to express my gratitude to my Senior Thesis advisor, Professor Robert W. McGrail, for the guidance they provided me throughout my senior year. Their time and advice supported the progression of my project and enabled me to focus my studies and successfully complete this project.

I also thank my professors, friends, and family for the immense support they've provided throughout my time at Bard College as an undergraduate student.

Table of Contents

1. Introduction

For my senior project, I will be exploring the optimization of query times for a MongoDB database by using PostgreSQL, or Postgres, as a local cache. I originally developed an interest in MongoDB while exploring the possibility of building and optimizing a database for an e-commerce company. After researching NoSQL database systems, which MongoDB employs, I became aware of the strengths of NoSQL databases and where they fall short. NoSQL databases management systems were created as an alternative to the industry standard; SQL database systems. The two differ a fair amount in their structure and dependencies and therefore thrive in different environments.

The idea to cache MongoDB queries and data within Postgres developed from considering ways in which the two database systems could work together harmoniously, providing a system that enables users to take advantage of the flexibility of MongoDB without sacrificing speed (specifically when conducting more complex aggregate queries within MongoDB that require the referencing of multiple data collections). As will be discussed in the paper, these queries can be costly under MongoDB's NoSQL database system, and these "JOIN" operations display an instance of where NoSQL database systems fall short of traditional SQL systems.

This paper will include brief background into SQL and NoSQL database systems, and an analysis of the technical differences between the two. The paper will then describe a method of using Postgres as a cache, and how MongoDB aggregate queries can be stored within Postgres tables to retain the data that the queries retrieve, as well

as key information that foreshadows what the query will return purely based on the query itself. This cached data can later be used to satisfy queries without the need to query the MongoDB database itself, thus reducing the query time and accelerating the rate at which data can be returned to the user. Finally, benchmark testing will provide data that verifies the improvements that this environment provides over querying the MongoDB database through conventional means.

2. Background

SQL

As discussed, my project will utilize a NoSQL data system facilitated by MongoDB's interface for the main database, and an SQL database through Postgres for caching query data. "NoSQL" stands for "non-SQL," or "non-relational SQL." To understand this distinction, it is important to contextualize what an SQL database is.

After its conception in 1974, the SQL language, based on the "relational data model," would become the industry standard for managing databases. The SQL language came to be after recent PhD graduates, Ray Boyce and Donald D. Chamberlin, were introduced to Ted Codd's new "relational data model," which was being developed at IBM's San Jose Research Laboratory. The "relational data model" was inspiring to Boyce and Chamberlin, as they appreciated its ability to allow a query of a database to be reduced to only a few lines. Compared to other programs at the

time, which required much more complex queries, they saw an opportunity to create a language, SQL, that would allow everyday users to query a data system using simple, human-language-like commands.

Boyce and Chamberlin believed that the biggest barrier between relational data models and public use was the notation of the queries to access data in the data system. Codd developed query languages that were based on Relational Algebra and Relational Calculus. His languages were effective in querying the relational model, however, they required knowledge of mathematical notation that was used in formal logic and lacked familiarity with common human language.[1] Boyce and Chamberlin envisioned that the queries, or the questions that you "ask" a database in order to retrieve information, could be represented by a notation more closely related to human-like languages, hence the creation of SQL: "Sequel: A structured English Query Language."[2]

Table 1. Employee.

Name	Salary	Manager
Smith	45,000	Harker
Jones	40,000	Smith
Baker	50,000	Smith
Nelson	55,000	Baker

Figure 1. Example of a Relational Database[3]

1. Chamberlin, Donald D. "Early History of SQL." *IEEE Annals of the History of Computing* 34, no. 4 (2012): 78.

2. Chamberlin, 79.

3. See note 1 above.

$\pi_{e.name}$ ($\sigma_{e.salary > m.salary}$ (ρ_e(employee) $\bowtie_{e.manager = m.name}$ ρ_m(employee)))

(a) Relational Algebra version

RANGE employee e;
RANGE employee m;
GET w (e.name): ∃m((e.manager = m.name) ∧ (e.salary > m.salary))

(b) Relational Calculus version

```
select e.name
from employee e, employee m
where e.manager = m.name and e.salary > m.salary
```

(c) Sequel (SQL) version

Figure 2. Example of three different query languages[4]

NoSQL

Although SQL relational databases are incredibly effective for managing data systems, other data models exist that are more effective in certain situations. SQL databases require a certain rigidity of the data. All of the data present in the SQL database must adhere to the same overall structure. Figure 1 displays this in a simple database, where every member of the “Employee” database must contain the same attributes, “Name,” “Salary,” and “Manager.” NoSQL (Not Only SQL) databases provide database solutions for large volumes of data that are not necessarily all structured the same way.

4. Chamberlin, 79.

The relational model usually represents data with a database schema. Data in the database are required to follow the same schema, therefore requiring that each entry have the same type, format, and number of characteristics. The data is stored in columns and rows, where each row has the same number of columns. NoSQL databases, on the other hand, support data that is semi-structured or not structured at all, allowing data to be grouped together, even though the attributes of the data points may have different characteristics. There are over 150 different NoSQL databases, all of which are based on the same principles but with slightly different implementations. They can typically be defined into one of four categories: Key-Value Store, Document Store, Column-family, and Graph databases.[5] MongoDB organizes data using Key-Value pairs, where the key identifies the particular point of data and the value can be a variety of types, such as numerical values, words, or even another complex structure with its own unique attributes.[6]

Although the SQL relational model is efficient and well established, the use of a NoSQL database model is perhaps more conducive to the flexibility and ease of use required for many modern companies whose data may change over time, and whose data needs to be able to scale well through many phases of company growth. Additionally, if a strict database schema is necessary, there are still methods to enforce a schema with NoSQL databases.

5. Abramova, Veronika, and Jorge Bernardino. “NoSQL Databases.” *Proceedings of the International C* Conference on Computer Science and Software Engineering - C3S2E '13*, (2013): 16.

6. Parker, Zachary, Scott Poe, and Susan V. Vrbsky. “Comparing NoSQL Mongodb to an SQL DB.” *Proceedings of the 51st ACM Southeast Conference on - ACMSE '13*, (2013): 1.

```
Customer
{
     “_id”: UniqueID,
     “name”: String,
     “email”: String,
     “birth _year”: Int,
     “address”: {
          “line_1”: String,
          “line_2”: String,
          “state”: String,
          “zip_code”: String
     }
}
```

Figure 3. Example of a Document-Based Database Schema

3. SQL vs NoSQL Technical Analysis

There are many SQL database management systems and a growing number of NoSQL systems. Although the systems share the same qualities with other database management systems of their respective type (SQL/ NoSQL), SQL and NoSQL database management systems, while maintaining the key attributes of their respective Query language types, can also differ a fair amount between other database management systems. Each SQL and NoSQL database focuses on its own unique qualities based on its niches. There are many SQL database management systems, and the NoSQL array of database programs is also expanding.

For the purpose of this paper, I will be using Postgres to analyze the differences between SQL databases and NoSQL databases. Postgres will be

particularly convenient to use because it is free and reliable, supports foreign keys without needing advanced configuration, and also supports JSON data (MongoDB's documents are also stored in JSON).[7]

ACID vs BASE

Both SQL and NoSQL databases are based on a set of principles to ensure the integrity and performance of data transactions. Both use principles derived from the CAP theorem. This theorem ensures:

- Consistency: all nodes have the same data at the same time
- Availability: all requests have a response
- Partition tolerance: if one part of the system fails, the rest of the system will be maintained

While generally adhering to these principles, SQL and NoSQL databases differ slightly in their implementation. SQL databases follow ACID principles, while NoSQL databases follow BASE principles.

ACID:

- Atomic: a transaction is completed when ALL operations are completed, otherwise previous state is restored
- Consistent: transaction cannot collapse database, as if an error occurs, the previous state is restored
- Isolated: transactions are independent and cannot affect each other
- Durable: when an operation is committed, the transaction cannot be undone

7. Obe, Regina O., and Leo S. Hsu, *PostgreSQL: Up and Running* (Sebastopol, CA: O'Reilly, 2015), 107-158.

BASE:

- Basically Available: all data is distributed, even if there is an error, the system continues to function
- Soft state: consistency is not guaranteed at every moment of database usage
- Eventually consistent: system guarantees that data will eventually be consistent

Overall, ACID databases are more robust and reliable, however as the amount of data grows, adhering to ACID principles is far more difficult. NoSQL databases rely on easy horizontal scaling, and therefore adhering to BASE principles allows the database to be more flexible.[8]

Process of Initializing Database

The process of creating a new database and adding your first piece of data in MongoDB is very similar to Postgres. As a note, both of these programs support creating a database natively on your computer. Thus, the server is your computer's localhost. This is what I will be using for the following example.

The first major difference between SQL and NoSQL becomes apparent immediately when creating a database. With MongoDB, there is no defined "create" function. When you want to create a new database, you simply run the command to "use" a database that doesn't exist. The following example will use **mongosh**, the command line MongoDB shell program:

Shell Example> show dbs

8. Abramova, Veronika, and Jorge Bernardino, "NoSQL Databases," 16.

sample_cars

sample_boats

Shell Example> use sample_newExampleDatabase

switched to db sample_newExampleDatabase

Shell Example> db.createCollection("ExampleCollection")

{ok : 1}

Shell Example> show dbs

sample_newExampleDatabase

Sample_cars

Sample_boats

Figure 4. Creating databases in MongoDB using the shell command line

In this example, we first use the "show dbs" command to show all of the currently created databases. The next "use" command is used to select a database to work with. In the example, we run "use" on a database that does not currently exist, as displayed by the absence of the database name from the initial "show dbs" command. MongoDB then automatically assumes that you want to create a new database, confirming that you are within the new database. After adding a collection and, we show that the new database has been added after the second "show dbs" command is run. It is important to note that at this point, MongoDB does not care about the Schema of the new collection, which will house all of the data made up of JSON format information. At this point, you would be able to insert any JSON data into the collection without error.[9]

9. Chodorow, Kristina, *MongoDB: The Definitive Guide* (Sebastopol, CA: O'Reilly, 2013), 13-14.

When creating a database with SQL, the process is very similar. The only notable difference is that when creating a “table” within a database (similar to creating a “collection” in Mongo) you must provide a schema before any data is inputted. Additionally, SQL is less flexible than MongoDB’s NoSQL environment, so commands are more explicit and if you try to insert or alter a table that doesn't exist, an error will be thrown. The following example will use Postgres specific syntax:

```
SELECT datname FROM pg_database;
```

Result:

datname
sample_cars
sample_boats

```
CREATE DATABASE sample_newExampleDatabase
CREATE TABLE ExampleTable(
        COLUMN1 INT,
        COLUMN2 CHAR,
        COLUMN3 TEXT
);
SELECT datname FROM pg_database;
```

Result:

datname
sample_newExampleDatabase
sample_cars
sample_boats

Figure 5. Creating databases in Postgres[10]

As shown, this example showcases the same process as the previous MongoDB example. The notable difference between the two is that the process in SQL is more rigid and requires the explicit creation of tables. SQL requires that a table has a predefined schema when it is created within the database. The user must specify the exact column names and data types, and also has the option to add other requirements to the rows, such as setting a row to be a primary key, requiring that a field be NOT NULL, specifying limits of characters or numbers in a field, and many others.[11] MongoDB, although it does not enforce a schema on documents within a collection, does still give you the ability to enforce document validation rules. You can enforce document schemas in MongoDB, however, the flexibility of the documents within a collection is one of the drawing qualities of MongoDB's environment.

10. Obe, Regina O., and Leo S. Hsu, *PostgreSQL: Up and Running*, 26-30.
11. Obe, Regina O., and Leo S. Hsu, 107-111.

“JOIN” Operations

It would be mundane to compare every clause relation between Postgres and MongoDB, however, there are a few key differences between the two that offer additional insight into how they differ and how they achieve different goals. At the end of the day, both SQL and NoSQL databases are just that, databases. They both have the ability to read, write, update, and delete data. Where they differ is how exactly they achieve these goals, especially when spanning multiple tables or collections.

The most important difference between the two is that SQL uses “JOIN” operations frequently, whereas in MongoDB, joining multiple collections can be extremely expensive and require complex aggregate queries that become algorithmic in nature.[12] SQL gives the user the ability to “JOIN” tables in a variety of ways: “INNER JOIN,” “FULL OUTER JOIN,” “LEFT JOIN,” “RIGHT JOIN,” and others.

SQL “JOIN” Operations

For the following “JOIN” operations, two simple tables will be used. Cart_a will be the “left” table and cart_b the “right.” Joins are called on the tables where item_a matches item_b:[13]

12. Copeland, Rick, *MongoDB Applied Design Patterns* (Beijing, China: O'Reilly, 2013), 3-14.

13. Ullman, Jeffrey D., and Jennifer Widom, *A First Course in Database Systems*. Seconded (Upper Saddle River, New Jersey: Prentice-Hall, 2002), 270.

Tables (cart_a, and cart_b)

cart_a

a	item_a
1	Pencil
2	Pen
3	Eraser
4	Highlighter

cart_b

b	item_b
1	Pen
2	Pencil
3	Sharpener
4	Calculator

INNER JOIN: Compares rows from cart_a to cart_b and returns rows that are equal.

a	item_a	b	item_b
1	Pencil	2	Pencil
2	Pen	1	Pen

LEFT JOIN: Compares rows from cart_a to cart_b and returns all rows from cart_a. If cart_b has a row that matches,

a	item_a	b	item_b
1	Pencil	2	Pencil
2	Pen	1	Pen
3	Eraser	NULL	NULL
4	Highlighter	NULL	NULL

RIGHT JOIN:

a	item_a	b	item_b
1	Pencil	2	Pencil
2	Pen	1	Pen
NULL	NULL	3	Sharpener
NULL	NULL	4	Calculator

FULL OUTER JOIN:

a	item_a	b	item_b
1	Pencil	2	Pencil
2	Pen	1	Pen
3	Eraser	NULL	NULL
4	Highlighter	NULL	NULL
NULL	NULL	4	Calculator
NULL	NULL	3	Sharpener

Figure 6. Tables produced by "JOIN" functions in Postgres

MongoDB "JOIN" Operations

In MongoDB, documents are meant to be organized into collections that should ideally have no connection to one another. Therefore, the necessity for "JOIN" operations would be diminished. MongoDB's object-oriented approach to storing data promotes denormalized data. This means that redundancy is almost encouraged to increase readability and ease of use at the expense of memory. As such, MongoDB's API does not include equivalents to SQL's "JOIN" functions. They are still achievable, however, they require writing complex code and are far more complex queries than their SQL counterparts.

In order to get around using “JOIN” functions in MongoDB, one might decide to simply use a nested object within a document.[14] For example, to continue the shopping cart example, one could create a “carts” collection in MongoDB and then create two different documents, “cart_a” and “cart_b.” The following JSON documents represent the two different carts and the items they hold.

```
{       _id: ObjectId(“0”),
        name: “cart_a”,
        contents: [‘Pencil’, ‘Pen’, ‘Eraser’, ‘Highlighter’ ]
}
{       _id: ObjectId(“1”),
        name: “cart_b”,
        contents: [‘Pen’, ‘Pencil’, ‘Sharpener’, ‘Calculator’ ]
}
```

Figure 7. Two documents that make up the “carts” collection

Using built-in MongoDB functions, one could then get similar results to an SQL “INNER JOIN” by running the following aggregate query:

```
db.carts.aggregate(
  {$group:{_id:null, first:{$first:"$contents"}, second:{$last:"$contents"}}},
  {$project: {commonToBoth: {$setIntersection: ["$first", "$second"]}, _id: 0 }}
)
```

14. Parker, Zachary, Scott Poe, and Susan V. Vrbsky, “Comparing NoSQL Mongodb to an SQL DB,” 2.

This query utilizes the “$group” and “$project” aggregate functions. Inside of the “$project” function, the “$setIntersection” function is used. In the aggregate function, the “$group” function is used to first convert the contents array from both documents into a usable variable. The values of the two contents arrays are stored into the “first” and “second” variables. Next, the “$project” function is used to return a new document with our desired properties, which in this case is simply the “commonItems” array. The contents of the “commonItems” array are provided by calling the “$setIntersection” function on the “first” and “second” arrays that were created. This is the step in which the common items are actually calculated. Thus, this aggregate function resembles an “INNER JOIN” in that it only returns common values.[15]

However, this is not entirely the same as an “INNER JOIN” in SQL. When comparing MongoDB to Postgres, Mongo’s collections are what tables are to Postgres, and each document in a collection is equivalent to a row in a table. SQL’s “JOIN” functions, however, are specifically for joining multiple tables, and therefore the previous example only serves to show that MongoDB’s document-based database is more useful for data that does not require “JOIN” functions. If redundant data is acceptable, then it is far more convenient to work within a MongoDB database. To properly emulate an SQL “JOIN” function in MongoDB, we must pull data from two separate collections.

In order to pull and compare data from two different collections in a MongoDB database, one must use the “$lookup” aggregate function. The “$lookup” function is essentially the same as a “LEFT JOIN” from SQL. Although “$lookup” can be used to

15. “Aggregation,” Aggregation - MongoDB Manual, accessed December 2, 2021, https://docs.mongodb.com/manual/aggregation/.

replicate a “LEFT JOIN,” SQL’s “JOIN” functions offer higher performance and greater variety in how two tables are joined.[16]

The following example will use two separate collections, “cart_a,” and “cart_b,” that are both under the same MongoDB database. The collections are the NoSQL equivalents of the SQL tables that were used in the previous example. Both collections, which resemble the SQL tables, will have a separate document for each cart item, which resembles the rows in the SQL tables for each of the two carts.

16. “Aggregation.” Aggregation - MongoDB Manual. Accessed December 2, 2021. https://docs.mongodb.com/manual/aggregation/.

cart_a collection	cart_b collection
{	{
_id: ObjectId(“0”),	_id: ObjectId(“0”),
a: 1,	b: 1,
item_a: “Pencil”	item_b: “Pen”
}	}
{	{
_id: ObjectId(“1”),	_id: ObjectId(“1”),
a: 2,	b: 2,
item_a: “Pen”	item_b: “Pencil”
}	}
{	{
_id: ObjectId(“2”),	_id: ObjectId(“2”),
a: 3,	b: 3,
item_a: “Eraser”	item_b: “Sharpener”
}	}
{	{
_id: ObjectId(“3”),	_id: ObjectId(“3”),
a: 4,	b: 4,
item_a: “Highlighter”	item_b: “Calculator”
}	}

	“LEFT OUTER JOIN”	“INNER JOIN”

“$lookup” function	db.cart_a.aggregate([{
	$lookup: {
	from: 'cart_b',
	localField: 'item_a',
	foreignField: 'item_b',
	as: 'common_items'
	}
	}]
		db.cart_a.aggregate([
		{
		'$lookup': {
		'from': 'cart_b',
		'localField': 'item_a',
		'foreignField': 'item_b',
		'as': 'common_items'
		}
		}, {
		'$match': {
		'common_items': {
		'$ne': []
		}
		}
		}
		])

Results	{ _id: ObjectId("612300b33e905 ef047d451da"), a: 1, item_a: 'Pencil', common_items: [{ _id: ObjectId("612300f53e905e f047d451df"), b: 2, item_b: 'Pencil'}] } { _id: ObjectId("612300b33e905 ef047d451db"), a: 2, item_a: 'Pen', common_items: [{ _id: ObjectId("612300f53e905e f047d451de"), b: 1, item_b: 'Pen' }]	{ _id: ObjectId("612300b33e905 ef047d451da"), a: 1, item_a: 'Pencil', common_items: [{ _id: ObjectId("612300f53e905e f047d451df"), b: 2, item_b: 'Pencil' }]} { _id: ObjectId("612300b33e905 ef047d451db"), a: 2, item_a: 'Pen', common_items: [{ _id: ObjectId("612300f53e905e f047d451de"), b: 1, item_b: 'Pen' }]}

	} { _id: ObjectId("612300b33e905 ef047d451dc"), a: 3, item_a: 'Eraser', common_items: [] } { _id: ObjectId("612300b33e905 ef047d451dd"), a: 4, item_a: 'Highlighter', common_items: [] }	

Figure 8. Replicating SQL "JOIN" functions with "$lookup" function

MongoDB "$lookup"	Postgres "JOIN"
db.cart_a.aggregate()	FROM cart_a
'from': 'cart_b'	'%' JOIN cart_b
'localField': 'item_a' 'foreignField': 'item_b'	ON item_a = item_b

Figure 9. "$lookup" function notation compared to SQL "JOIN" notation

After running the queries with “$lookup,” objects are returned with a new value called “common_items.” If there is a match between local and foreign fields as specified within the “$lookup” function, the “common_items” array will contain the matched object. Otherwise, it will remain empty or NULL, as is the case with SQL’s “LEFT JOIN” and “RIGHT JOIN.” In order to replicate SQL’s “INNER JOIN,” a “$match” function is added to the aggregate pipeline, filtering out all of the comparisons that yielded a NULL result within the “common_items” array.

Although more complex joins are possible using “$lookup,” as previously discussed, oftentimes the queries become very algorithmic in nature and require the use of a complex aggregate pipeline. Complex queries with the aggregate pipeline are inefficient compared to SQL’s “JOIN” functions. Additionally, they are not very adaptable because of the various stages of filtering that are required. Because MongoDB “JOIN” equivalents are complicated to achieve, users will often format their database in such a way that documents within the collection can use nesting or references to other documents in order to achieve a similar goal to a “JOIN” operation.[17]

Aggregation

Both Postgres and MongoDB support aggregation on databases and can both be used to analyze and compute statistical data. SQL has been used for statistical analysis for a long time, and thus it is well established and efficient. The two database management systems differ greatly in how the user creates aggregate functions, with MongoDB offering expanded complex aggregation pipelines that give the user more control over

17. Copeland, Rick, *MongoDB Applied Design Patterns* (Beijing, China: O'Reilly, 2013), 3-14.

how they interpret and present data from collections. SQL's aggregation functions are few; however, given the nature of how SQL stores data, all of the functions can produce results with extreme efficiency.[18]

Once again, the priorities of each database system, NoSQL and SQL, are displayed. SQL aggregation is extremely efficient and straightforward but lacks some of the customizability that MongoDB offers with its aggregation functions.

Postgres (SQL) Aggregation Functions	MongoDB (NoSQL) Aggregation Functions
AVG(), COUNT(), MAX(), MIN(), SUM()	$addFields, $count, $group, $limit, $lookup, $match, $merge, $project, $redact, $sample, $search, $set, $unwind, $unset, $unionWith, $sortByCount, $skip, $replaceWith, $replaceRoot, $redact, $planCacheState, $out, $listSessions, $indexStats, $graphLookup, $geoNear, $facets, $collStats, $bucketAuto, $bucket, $setWindowFields

Figure 10. Postgres aggregate functions vs MongoDB aggregate functions [19, 20]

18. Ullman, Jeffrey D., and Jennifer Widom, *A First Course in Database Systems*. Seconded (Upper Saddle River, New Jersey: Prentice-Hall, 2002), 270.

19. See note 18 above.

20. "Aggregation." Aggregation - MongoDB Manual. Accessed December 2, 2021. https://docs.mongodb.com/manual/aggregation/.

Postgres Aggregate Functions and Terms	MongoDB Aggregate Methods
WHERE / HAVING	$match
GROUP BY	$group
SELECT	$project
LIMIT	$limit
OFFSET	$skip
ORDER BY	$sort
SUM() / COUNT()	$sum
JOIN	$lookup
SELECT INTO NEW_TABLE	$out
MERGE INTO TABLE	$merge
UNION ALL	$unionWith

Figure 11. Direct comparison of SQL to NoSQL functions

SQL's aggregation ability is empowered by the fact that "JOIN" functions can be carried out with ease. This enables a user to carry out statistical analyses across multiple tables within a database and draw insightful statistics. Thus, Postgres has the upper hand when it comes to drawing statistical conclusions across different datasets. It is important to note that, once again, a user has to have the foresight of exactly how to select and analyze data before the complex query can be written. Although aggregation functions can be used within complex queries, the order of aggregation functions and "JOIN" functions, along with all of the other clauses such as SELECT, WHERE, and GROUP

BY, can result in busy code that is difficult to write and read. SQL is a declarative language, and thus the programmer needs to know exactly what they want to see before they write the query. When writing SQL queries, the user also has to think in terms of sets which can be trivial for those who are not familiar with this way of programming.

MongoDB's aggregation pipeline places a greater emphasis on enabling the user to interpret and present data in a customizable way. MongoDB provides users with a more intuitive format of using aggregate functions in an "aggregation pipeline." Each separate aggregation function makes up its own "stage" of the aggregation pipeline. The aggregation pipeline is composed of an array, with each element of the array being a different aggregate function. The data of concern is then altered and filtered based on the order of the various "stages."[21] The ability to intuitively create a pipeline of aggregation functions, the plethora of functions in MongoDB, and the ability to mix the aggregation with every other query clause available allows for data within a collection (or between multiple collections) to be transformed and analyzed with extreme customizability. Although running statistical analyses on collections that require "$lookup" (SQL "JOIN" equivalent) functions is possible, it is far less quick and efficient than SQL and requires the use of more complex aggregation queries.

MongoDB also provides an aggregation framework within their software, MongoDB Compass, that provides the user with a real-time preview of the data as the user provides aggregation stages. This is very useful for writing complex queries, as it enables the user to think of their complex query in stepwise fashion, as opposed to

21. Chodorow, Kristina, *MongoDB: The Definitive Guide*, 127-129.

having to write the entire query in SQL before you can see the outputted data. It is also possible to turn off aggregate stages with ease, and even drag and drop aggregate stages into different places in the aggregate pipeline.

After each stage is added to the pipeline, MongoDB Compass also provides a template for the syntax of the aggregation function. Also, if there are any errors present, or if the query stage is not properly formatted, MongoDB will make it apparent and point you towards the quickest fix.

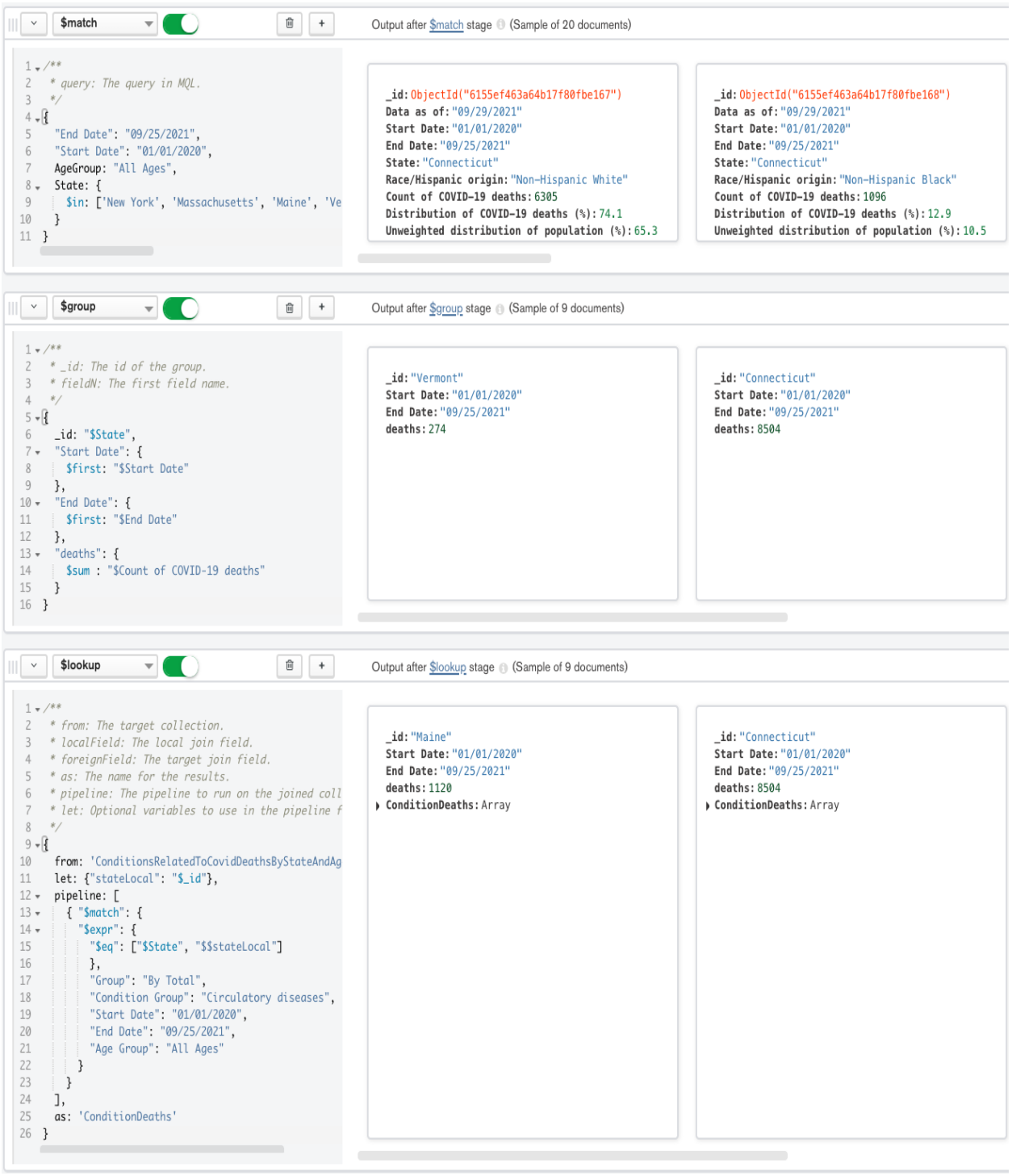

Figure 12. Three stages of aggregate pipeline in MongoDB Compass UI

MongoDB's aggregation pipelines are represented in JSON format, making them very readable and easy to interpret. Here is an example of an aggregate query that reduces a collection with 3710 documents to 9 documents with useful statistics. In this query, two MongoDB collections, 'ConditionsRelatedToCovidDeathsByStageAndAge', and 'CovidDeathsByStateAgeAndRace', are joined in order to draw statistics on Covid deaths with the related condition 'Circulatory Diseases' in a given array of states within the United States.

MongoDB aggregate function stages

```
{
   '$match': {
     'End Date': '09/25/2021',
     'Start Date': '01/01/2020',
     'AgeGroup': 'All Ages',
     'State': {
       '$in': [
         'New York', 'Massachusetts', 'Maine', 'Vermont', 'Connecticut', 'New Hampshire', 'Rhode Island', 'New Jersey', 'Pennsylvania'
       ]
     }
   }
 }
```

In this first stage of the query, the "$match" stage, the documents that are returned are filtered based on key-value pairs within the document. In this case, only documents with an 'End Date' of '09/25/2021', a 'Start Date' of '01/01/2020', etc. will be returned. The '$in' array within the 'State' key filters out every document that contains a value for 'State' that is not present in the '$in' array. Sometimes, the queries in this project will contain the '$nin' operator, which filters out all documents with values that are contained within the '$nin' array.

This stage is equivalent to an SQL WHERE or

Sample document:

1. _id:6155ef463a64b17f80fbe167
2. Data as of:"09/29/2021"
3. Start Date:"01/01/2020"
4. End Date:"09/25/2021"
5. State:"Connecticut"
6. Race/Hispanic origin:"Non-Hispanic White"
7. Count of COVID-19 deaths:6305
8. Distribution of COVID-19 deaths (%):74.1
9. Unweighted distribution of population (%):65.3
10. Weighted distribution of population (%):60.8
11. Difference between COVID-19 and unweighted population %:8.8
12. Difference between COVID-19 and weighted population %:13.3
13. AgeGroup:"All Ages"

	HAVING clause.	

'$group': {
'_id': '$State',
'Start Date': {
'$first': '$Start Date'
},
'End Date': {
'$first': '$End Date'
},
'deaths': {
'$sum': '$Count of COVID-19 deaths'
}
}
}
	In the “$group” stage, documents are grouped together by the value given to the ‘_id’ stage. In this example, documents are being grouped by the ‘$State’ value. This then allows us to compute the sum of Covid deaths in each state within the remaining documents. This is achieved by the ‘$sum’ operator on ‘$Count of COVID-19 deaths’.
	
	This stage is equivalent to an SQL GROUP BY clause.
		Sample document:
		
		1. **_id:**"Connecticut"
		2. **Start Date:**"01/01/2020"
		3. **End Date:**"09/25/2021"
		4. **deaths:**8504

'$lookup': {
'from': 'ConditionsRelatedToCovidDeathsByStateAndAge',
'let': {
'stateLocal': '$_id'
},
'pipeline': [
{
'$match': {
'$expr': {
'$eq': [
'$State',
'$$stateLocal'
]
},
'Group': 'By Total',
'Condition Group': 'Circulatory diseases',
'Start Date': '01/01/2020',
	In the “$lookup” stage, two MongoDB collections are joined. The ‘let’ keyword allows the user to store values from the local collection in a dictated variable name; in this case, the ‘_id’ value (which represents a state) from the group stage is stored in order to join the documents from the two collections on their ‘State’ value.
	
	In the ‘pipeline’, the user can filter the foreign collections documents, and match the local variables to a field in the foreign collection. In this stage, the documents
		Sample document:
		
		1. **_id:**"Connecticut"
		2. **Start Date:**"01/01/2020"
		3. **End Date:**"09/25/2021"
		4. **deaths:**8504
		5. **ConditionDeaths:**Array
		1. **0:**Object
		2. **1:**Object
		3. **2:**Object
		4. **3:**Object
		5. **4:**Object
		6. **5:**Object
		7. **6:**Object
		
		Nested document:

'End Date': '09/25/2021', 'Age Group': 'All Ages' } }], 'as': 'ConditionDeaths' } }	from 'ConditionsRelatedToCovidDeathsByStateAndAge', are filtered so that the remaining documents align with the local collection, including sharing the same Start and End date, and the same Age Group. The two collections are also joined on their '$State' values, as displayed by the '$eq' operator found nested within the "$match" phase of the 'pipeline'. The "$lookup" stage returns an array under each document from the local collection that includes matching documents from the foreign collection. This stage is similar to a "LEFT JOIN" ON '$State' in SQL.	1:Object 1. _id:6155ec8a3a64b17f80f75b6d 2. Data As Of:"09/26/2021" 3. Start Date:"01/01/2020" 4. End Date:"09/25/2021" 5. Group:"By Total" 6. State:"Connecticut" 7. Condition Group:"Circulatory diseases" 8. Condition:"Ischemic heart disease" 9. ICD10_codes:"I20-I25" 10. Age Group:"All Ages" 11. COVID-19 Deaths:750 12. Number of Mentions:776

'$addFields': {
'Condition Deaths': {
'$sum': '$ConditionDeaths.COVID-19 Deaths'
}
}
}, {
'$addFields': {
'Ratio': {
'$divide': [
'$Condition Deaths', '$deaths'
]
}
}
}, {
	I've included three "$addFields" stages in the same section. They are all used to simply add a key-value pair to the remaining documents.
	
	First, all of the 'COVID-19 deaths' are summed from the ConditionDeaths array that was returned from the "$lookup" stage. Then, due to the fact that added fields cannot be referenced within the same aggregate stage, two additional
		Sample document:
		1. _id:"Connecticut"
		2. Start Date:"01/01/2020"
		3. End Date:"09/25/2021"
		4. deaths:8504
		5. ConditionDeaths:Array
		6. Condition Deaths:5062
		7. Ratio:0.5952492944496708
		8. Percent:59.52492944496708

'$addFields': {
'Percent': {
'$multiply': [
'$Ratio', 100
]
}
}
}
	"$addFields" stages are used for the final steps of computing the final statistic.	

'$project': {
'_id': 1,
'state': 1,
'Start Date': 1,
'End Date': 1,
'Percent of deaths with a CC': {
'$round': [
'$Percent', 2
]
}
}
}
	The "$project" stage is the final stage, and it is used to dictate what information will be present in the final documents.
	
	This is similar to SQL's SELECT clause.
		Sample document:
		
		1. _id:"Connecticut"
		2. Start Date:"01/01/2020"
		3. End Date:"09/25/2021"
		4. Percent of deaths with a CC:59.52

Figure 13. Aggregate pipeline stages

4. Caching MongoDB Queries in Postgres

The goal of this project is to optimize MongoDB queries by caching them (specifically queries that require "JOIN" operations) in Postgres. This allows new queries to then be analyzed and satisfied using the cached data, avoiding the need to query the MongoDB database. Queries that require "JOIN" operations (through the "$lookup" function) in MongoDB are far less efficient than "JOIN" queries in SQL as previously discussed.

Caching query results in Postgres eliminates the need for any cached query to be retrieved from the MongoDB database more than once. Thus, MongoDB's expansive version of an SQL "JOIN" function will never have to be run more than once. Additionally, the program includes methods that parse and analyze queries and, using these methods, can determine if a new query is a subset or union of previously cached queries in Postgres, as well as simply retrieve the cached data if the exact query has been run before.

Semantic query information can also be entered via text by the user to describe relationships between queries in "query1 is union/subset of query2" form. Having this additional semantic layer can save even more time by allowing data to be pulled from Postgres without ever having to parse the MongoDB aggregate query in the first place.

The project uses a main MongoDB database as a test example that contains statistics and information about Covid-19. The data was pulled from the federal government's open data site,[22] and contains statistics about Covid-19 deaths and related conditions based on criteria such as Age, State, and Race. Finally, for the purpose of this project, the program assumes that there are a set number of pre-made aggregate queries, each with a relatively similar structure. Additionally, assume that each of the queries will result in less than or equal to 10 final statistics. This structure is necessary as Postgres tables are strict, and therefore a predetermined limit on returned statistics must be imposed to ensure that the Postgres table that stores the data will always suffice.

22. "Data Catalog." Datasets - CKAN. Accessed September 12, 2021. https://catalog.data.gov/dataset?groups=older-adults-health-data&res_format=JSON&page=1.

The first step in this process is to parse the aggregate queries from MongoDB. The goal of the parser is to draw out useful information from the aggregate query so that it can later be used to identify the query, as well as find relationships with other future queries. Because MongoDB aggregate queries are written in JSON format, javascript is able to easily sift through the various stages in the aggregate pipeline and pull key identifiers out of the query. The results from parsed aggregate queries then get stored in Postgres with the following columns:

id	Primary identifier for the cached query
data_ref_id	Reference ID used as a foreign key to another Postgres table that stores the stats pulled after running the query
pipeline	Copy of the aggregate query (JSON query)
native_collection	Local collection that query is called on
foreign_collection	Joined query as part of the “$lookup” stage of the aggregate pipeline
content_filter	Stage of query where most documents are filtered out. Queries revolve around State, Age, Race, and Related Conditions to Covid-19, and this is the stage where the final documents are dictated. Ex/ Filtering out all States that are not equal to Florida, New York, Connecticut, New Jersey, and Washington. In this case, each value in the “aggregate_stats” table would relate to statistics gathered in each of the 5 states. Thus, there would be 5 statistical data points retrieved.
group_id	How documents are grouped together. The queries used for this project all have a group_id that matches the category of categorical variables dictated in the content_filter stage. Ex/ Grouping documents by their State.
start_date	Because all of the documents from the Covid-19 database contain statistics from the US, they each have a start and end date dictating where the data spans.
end_date	See above

common_attr	Contains the data point that collections are joined on in the "$lookup" state (Similar to the 'ON' in an SQL "JOIN" clause. Ex/ "JOIN" collection1 ON local_State = foreign_state
final_stat_name	The description of the final statistics that are found from the aggregate query. Ex/ "Percent of Covid-19 Deaths with CLRD"
stages	The number of stages in the aggregate query pipeline. This is useful for comparing queries as one can easily see how many stages of filtering a query went through

Figure 14. "aggregate_obj" column descriptions

If a query must be run for the first time, it will not exist within the cached data in Postgres. Therefore, the aggregate query will be parsed, and the query information will first be cached within the "aggregate_obj" table within Postgres. Then, the query data will be retrieved from MongoDB, and the data itself then parsed and cached into another Postgres table. This second Postgres table, called "aggregate_stats", contains the following columns:

id	This is the primary key for this table, and a reference to this id is included in the "aggregate_obj" row that represents the query that yielded the statistics
statname	The statname is a general description of the found statistics and is identical to the final_stat_name of the related row in the "aggregate_obj" table
stat1…. statN (where N is the number of statistical data points retrieved)	Each stat contains a datapoint retrieved from the aggregate function. Ex/ with statname "Percent of Deaths with a CC by State," a stat will look like "Maine: 46.79"

Figure 15. "aggregate_stats" column descriptions

```
{
  stages: 7,
  end_date: '09/25/2021',
  extras: [],
  start_date: '01/01/2020',
  '$in': [
    'New York',
    'Massachusetts',
    'Maine',
    'Vermont',
    'Connecticut',
    'New Hampshire',
    'Rhode Island',
    'New Jersey',
    'Pennsylvania'
  ],
  'in/nin': 9,
  group_id: '$State',
  foreign: 'ConditionsRelatedToCovidDeathsByStateAndAge',
  aggrStat: 'Percent of deaths with a CC by State',
  '$eq': [ '$State', '$State' ]
}
```

Figure 16. Parsed query data

As shown in the figure above, parsing the query returns a new object with important identifier attributes for the query. From this new object, attributes are pulled and stored within the “aggregate_obj” table in Postgres. It is important to restate that the “content_filter” attribute is represented as an array of categorical variables: in this case, states in the North East. Additionally, the “group_id” attribute will always relate to the contents of the “content_filter” array. Thus, in this case, the “group_id” attribute would be “$State.”

Overall Structure

Whenever a query is made in the program, it goes through a series of checkpoints in order to identify whether or not the query can be fulfilled using cached data. If the query has not been cached and fails all of the checkpoints, the MongoDB database is then queried, and the query attributes and resulting data are cached in Postgres into their respective tables, “aggregate_obj” and “aggregate_stats.”

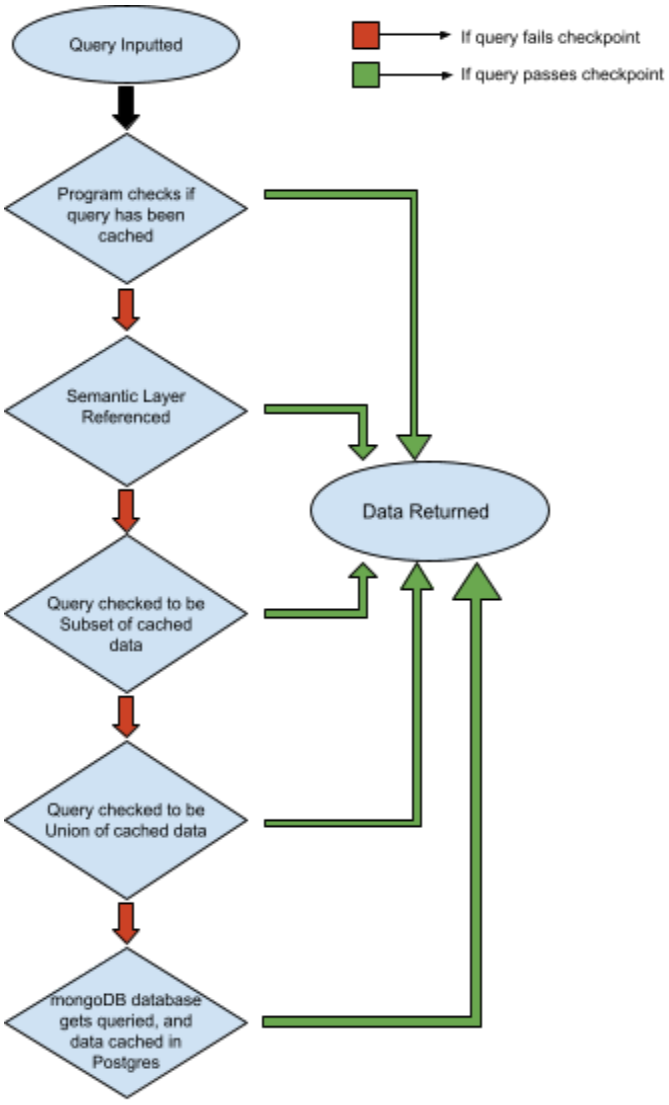

Figure 17. Program checkpoints

Query Checkpoints

At the first checkpoint, the program checks if the query is already cached within Postgres. This is accomplished by connecting to the Postgres server and checking the “aggregate_obj” table for any rows of data where the “pipeline” column matches the input query. This is the most straightforward checkpoint, as only one comparison needs to be made. If the input query matches the “pipeline” column of any row in the Postgres table, the row is returned, and the “data_ref_id” attribute is used to pull the cached statistics from the “aggregate_stats” table. The data is then returned to the user and the query terminated. If the input query has no matches, the query is sent to the proceeding checkpoint.

At the second checkpoint, the “Semantic Layer” is referenced. The program allows users to store semantic information if they are certain of relationships between various queries. The user is able to store Union and Subset relationships in the semantic layer, following the form “query1 is union/subset of query1,query2…queryN.” These relationships are then referenced when a query reaches the second checkpoint. If, for example, “query1” is the input query, and there exists a semantic pointer for “query1” that states “query1 is union of query2,query3,” the data will be retrieved from Postgres and returned without ever having to parse the aggregate function. The data of concern will simply be returned by using the query labels (in this case “query2” and “query3”) to reference the data stored in the two Postgres tables. This is able to work due to the fact that the program depends on a set amount of queries, each of which has a generic label such as “query1” or “query2.”

At the next checkpoint, the query is parsed and the query information is compared to other queries that exist in the cache. The parsed query is then compared to cached queries to determine if it is a possible subset of an already cached query. If the input query is confirmed to be a subset of another query, the data points of concern will be pulled from the “aggregate_stats" table within Postgres. In this case, since the input query pipeline is slightly different from the cached query pipeline, more than one comparison will need to be made. The two queries cannot be matched simply based on the JSON pipeline. However, if a query is a subset of another query, they will have many attributes in common. Once the subset query is parsed, elements of the parsed data can be compared to the rows within the “aggregate_obj” table. If the two queries share a determined set of attributes, we can then assume that the two queries are concerned with the same data.

```
//Retrieve data_ref_id from aggregate_obj table
 const query = `SELECT * FROM aggregate_obj
       WHERE (native_collection = $1 and foreign_collection = $2 and group_id = $3
       and
       start_date = $4 and end_date = $5 and common_attr = $6 and stages = $7 and
       final_stat_name = $8)`;
 const values = parsedQuery;
 let data_test = await postgresClient.query(query, values);
```

Figure 18. Code snippet from Subset Test method

As shown in the code snippet above, the following attributes are compared between the two parsed queries: the native collection of the aggregate function, foreign or joined

collection in the query, group_id, start and end date, common_atr, stages, and final_stat_name. If the two queries share these attributes (refer to figure 17), we can be highly confident that they return similar data. After the two queries are confirmed as a match, the content_filter attributes of the two queries are compared. Since we are checking that the input query is a subset of the cached query, we need to determine if the cached query contains the correct final statistics corresponding to the input query's content_filter. Since the content_filter attribute dictates what the final documents will be, and how many there will be, it can be used to finally determine if the input query is a subset of a matched cached query.

For example, if an input query has a content_filter attribute that consists of 5 states, we can first check that the matched cached query has a content_filter that consists of at least 5 members. If there are fewer members than the input query, then we know that the cached data cannot satisfy the query, as the input query cannot be a subset of a query that returns fewer statistics than itself. After the sizes are compared, the program will then loop through the two content_filter arrays and determine if the cached query contains all of the members that the input query seeks to gather statistics on. Once it is determined that the cached query contains the necessary statistics, the program will then return the data to satisfy the input query.

The following checkpoint is the last attempt by the program to satisfy the input query using cached data. At this stage, the query is parsed, and it is then determined if the query can be satisfied by performing a union of already cached data. The input query can be determined as a possible candidate of a union operation by first checking

the final_stat_name attribute. If this attribute consists of more than one element, it means that the query intends to return more than one statistic for each group of documents. For example, if a query seeks to find the percentages of three different conditions related to COVID-19 across 7 states, it may be a candidate for a union of cached data. In this scenario, if there existed 3 rows of cached statistics for each of the three conditions relating to the 7 states, the program would recognize that the input query can be satisfied from the cached data, and return it without querying the MongoDB database.

```
for (var statistic in queryArrayFinalStats){
    let statName = queryArrayFinalStats[statistic];
    const query = `SELECT * FROM aggregate_obj
        WHERE (native_collection = $1 and foreign_collection = $2 and group_id = $3
        and start_date = $4 and end_date = $5 and common_attr = $6 and
        final_stat_name = $7)`;
    const values = parsedQuery;
    let data_test = await postgresClient.query(query, values);
    // If query returns any results
    if (data_test.rows.length > 0) {
      idArray.push(data_ref_id);
    } else {
      dataAsUnion = false;
    }
  }
```

Figure 19. Code snippet from Union Test Method

In the code snippet above, the code loops through all of the statistical names found in the “final_stat_name” array of the input query. For each element of this array, the Postgres table “aggregate_obj” is queried in order to see if any rows of currently cached data match. When the Postgres client is queried, if there are any matching rows, the “data_ref_id” from the row will be saved in another array that will eventually be used to retrieve all of the data from the “aggregate_stats” table. If for any of the statistical names in the input queries “final_stat_name” array, there does not exist a match in currently cached data, the input query is correctly deemed unable to be satisfied by a Union operation. If this final checkpoint fails, the query is sent to the MongoDB database, and then the results are cached in Postgres.

5. Methods

Sample Groups

In order to validate the improvements that the checkpoints provide in regards to accelerated query times, five benchmark tests were carried out. The benchmark tests were run using a set of 60 total queries. Each test consisted of two groups of 30 queries. Additionally, in four of the tests, half of the total queries were cached, and half uncached. Of the 60 total queries, half of them were “base queries,” where the queries represent a collection of data that is not a possible union or subset of any other queries. The other half are Union or Subset queries of the base queries. Having the 60 queries organized in this manner allowed the tests to always consist of two groups, each with a

sample size of 30 queries, regardless of whether the test is a comparison between uncached, cached, union, subset, or semantic queries. In every test, the queries in Group One had a corresponding query in Group Two that was relatively identical (or identical) in content and length. This ensured that the independent variable of concern was always the method of retrieving the data.

Measures

Independent Variable: Query algorithm (Retrieving data from MongoDB servers and caching data, only retrieving data from MongoDB servers, cached data in Postgres, Union/Subset of cached data, or pulling from cached data using Semantic information).

Dependent variable: Time (ms) to satisfy the query

Test 1 Procedures:

This test used the following sets of queries: Queries retrieved from MongoDB servers (uncached data), and queries retrieved from cached data. First, the Postgres tables were cleared to ensure the integrity of the test. Next, the 30 queries in Group Two were parsed and cached within the two Postgres tables. The uncached queries in Group One were then run, retrieving data from MongoDB servers and recording the time of each query. Finally, the queries in Group Two were run, retrieving information from the data cached within the Postgres tables, recording the time of each query.

Test 2 Procedures:

This test used the following sets of queries: Queries retrieved from MongoDB servers (uncached data), and parsed queries satisfied by checking already cached data for Unions and Subsets. First, the Postgres tables were wiped to ensure the integrity of the test. Next, the 30 base queries were cached within the Postgres tables. Then, a set of 30 Union/Subset queries were retrieved from MongoDB servers and were not checked against cached data. Finally, the same queries were run, but this time they were parsed and then checked to be Unions or Subsets of already cached data, and thus were retrieved from already cached data.

Test 3 Procedures:

This test used the following sets of queries: Queries retrieved from MongoDB servers (uncached data), and queries satisfied by checking already cached data using Semantic information, detailing Union/Subset relationships between the queries and already cached data. First, the Postgres tables were wiped to ensure the integrity of the test. Next, the 30 base queries were cached within the Postgres tables. Then, a set of 30 Union/Subset queries were retrieved from MongoDB servers and were not checked against cached data. Finally, the same queries were run, but this time they were checked to be Unions or Subsets of already cached data using Semantic information, and thus were retrieved from already cached data.

Test 4 Procedures:

This test used the following sets of queries: Parsed queries satisfied by checking already cached data for Unions and Subsets, and queries satisfied by checking already cached data using Semantic information, detailing Union/Subset relationships between the queries and already cached data. First, the Postgres tables were wiped to ensure the integrity of the test. Next, the 30 base queries were cached within the Postgres tables. Then, a set of 30 Union/Subset queries were parsed and checked to be Unions or Subsets of already cached data. Finally, the same queries were run, but this time they were checked to be Unions or Subsets of already cached data using Semantic information, and thus were also retrieved from already cached data.

Test 5 Procedures:

This test used the following set of queries: Uncached queries retrieved from MongoDB that were then cached into Postgres, and uncached queries retrieved from MongoDB that were NOT cached. First, the 30 uncached queries were sent to MongoDB servers and then cached within Postgres. Next, the Postgres tables were wiped. Finally, the same 30 queries were sent to MongoDB servers but were NOT cached into Postgres. This test intends to show the overall additional time that it takes to cache the queries in Postgres, as compared to simply running the query against the MongoDB database.

```
await postgresClientMain.connect();
await mongoClientMain.connect();

//Postgres tables wiped to ensure integrity of tests
clearTables(postgresClientMain);

var groupOneQueryTimes = [ ];
var groupTwoQueryTimes = [ ];

//Initial group of queries Cached in Postgres
for (let query in cachedQueries) {
      //Queries run and data cached within two Postgres tables
      await query(collection, queryLabel, aggregateQuery, true, false, false, postgresClientMain,
      mongoClientMain, masterQueryArray);
}

//Time logged for running 30 uncached queries
for (let query in uncachedQueries) {
      let start = process.hrtime();
      await query(collection, queryLabel, aggregateQuery, false, false, false, postgresClientMain,
      mongoClientMain, masterQueryArray);
      let end = process.hrtime(start);
      //Query run time formatted and saved into respective data set
      groupOneQueryTimes.push((end[1]/1000000).toFixed(3) + 'ms');
}

//Time logged for running 30 cached queries
for (let query in cachedQueries) {
      let start = process.hrtime();
      await query(collection, queryLabel, aggregateQuery, false, false, false, postgresClientMain,
      mongoClientMain, masterQueryArray);
      let end = process.hrtime(start);
      //Query run time formatted and saved into respective data set
      groupTwoQueryTimes.push((end[1]/1000000).toFixed(3) + 'ms');
```

Figure 20. Sample testing script (Uncached vs Cached Data)

6. Results

After collecting data through the various query tests, Paired Samples t-tests were conducted to determine the statistical difference between the two samples in each test. Paired t-tests, or the t-test for Dependent Means, are used when the means of two groups are compared in order to draw conclusions about the differences between the groups. This method of testing is also called “repeated measures design”, as the two sample groups are often composed of the same group of subjects (in this case, queries). The Paired t-tests yield a “p-value,” which represents the probability of the results occurring under the assumption that there is no statistically significant difference between the two groups. Therefore, the lower the p-value, the more likely it is that the two groups are significantly different. Generally, if the p-value is less than the conventional 0.05 alpha value, the two sample groups are found to be significantly different.[23]

The sample groups used within the following tests all came from the same collective group of queries such that for each test, the two samples of queries are near (or completely) identical. The tests are therefore concerned with the performance of these queries under a variety of conditions: the queries before and after the implementation of a checkpoint algorithm, the queries under two different checkpoint algorithms, and the queries run with and without being cached into Postgres.

23. Aron, Arthur, Elaine N. Aron, and Elliot J. Coups, *Statistics for Psychology* (Upper Saddle River, New Jersey: Pearson Education, 2013), 239-247.

Test 1: Uncached vs Cached Data

Paired Samples T-Test

			statistic	df	p
Group 1	Group 2	Student's t	20.9	29.0	< .001

Descriptives

	N	Mean	Median	SD	SE
Group 1	30	544.5	585.6	140.46	25.64
Group 2	30	12.3	12.8	7.18	1.31

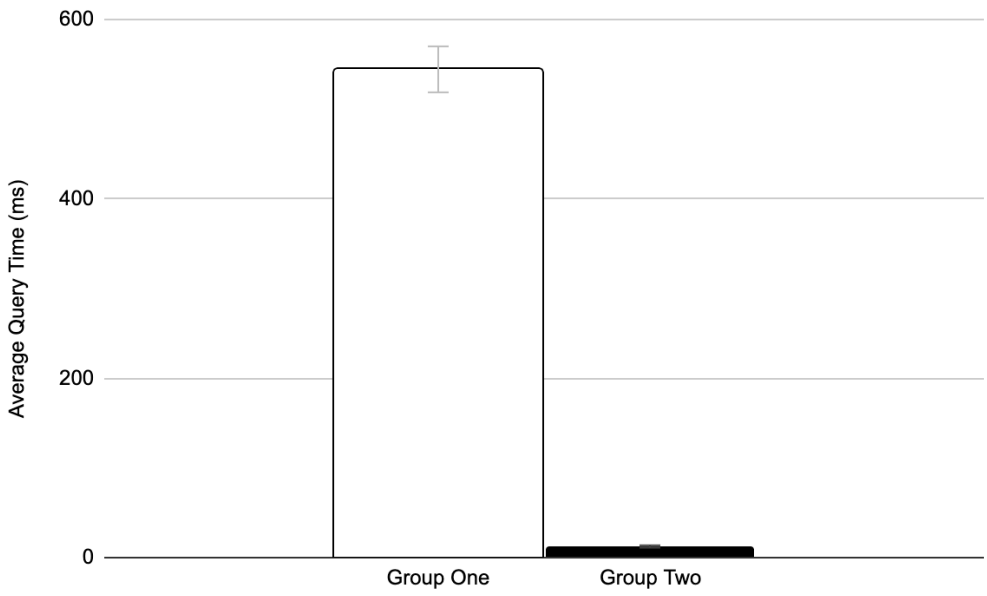

Figure 21. Test 1 Avg. Query Times with Standard Error Bar

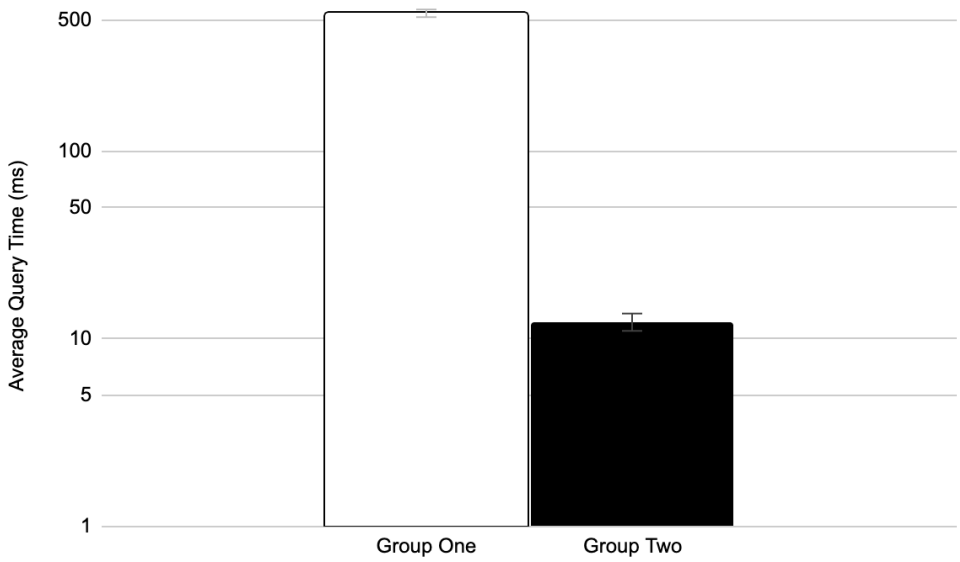

Figure 22. Test 1 Avg. Query Times with Standard Error Bar

(Log scale for better readability)

The results for test 1 found that there was a statistically significant difference between the query times for the two groups, $t(29) = 20.9$, $p = < 0.001$. Referencing the query in Postgres and retrieving the data took a fraction of the time in comparison to querying the MongoDB database (Group One Mean = 544.5ms, Group Two Mean = 12.3ms) and also had far less variation in the query times (Group One SD = 140.46, Group Two SD = 7.18).

Test 2: Uncached Queries vs Queries with Union/Subset Checking

Paired Samples T-Test

			statistic	df	p
Group 1	Group 2	Student's t	14.9	29.0	< .001

Descriptives

	N	Mean	Median	SD	SE
Group 1	3	561.3	509.0	200.4	36.5
Group 2	3	9.0	9.4	9.3	1.7

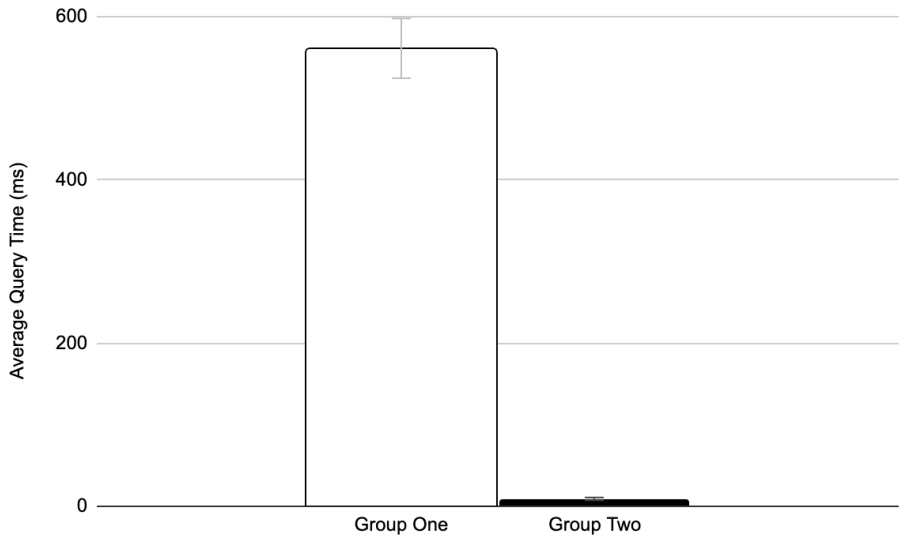

Figure 23. Test 2 Avg. Query Times with Standard Error Bar

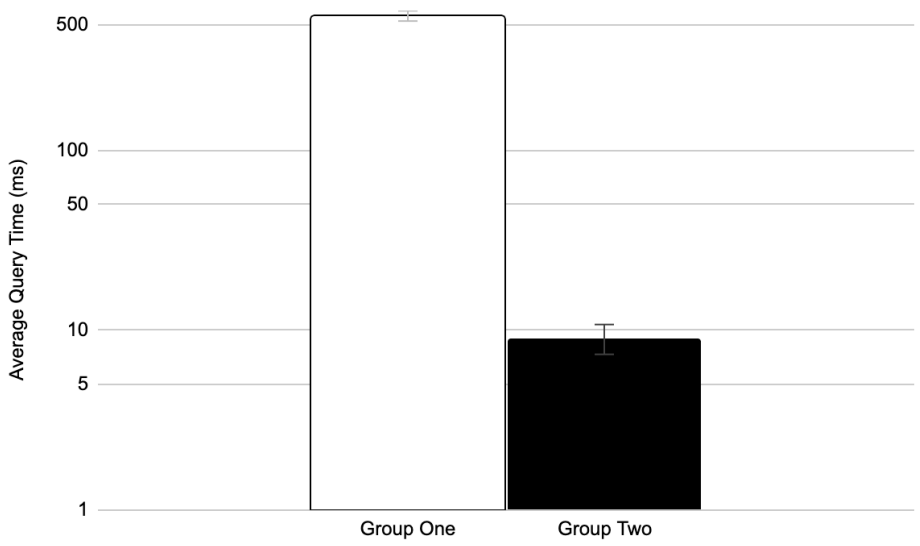

Figure 24. Test 2 Avg. Query Times with Standard Error Bar

(Log scale for better readability)

The results for test 2 found that there was a statistically significant difference between the query times for the two groups, $t(29) = 14.9$, $p = < 0.001$. Parsing the query and retrieving the data from already cached data using Union and Subset checking was far quicker than querying the MongoDB database (Group One Mean = 561.3ms, Group Two Mean = 9.0ms) and again had far less variation in the query times (Group One SD = 200.4, Group Two SD = 9.3).

Test 3: Uncached Queries vs Queries with Semantic Info Help

Paired Samples T-Test

			statistic	df	p
Group 1	Group 2	Student's t	17.1	29.0	< .001

Descriptives

	N	Mean	Median	SD	SE
Group 1	30	530.52	485.43	169.58	30.960
Group 2	30	3.32	2.85	2.22	0.405

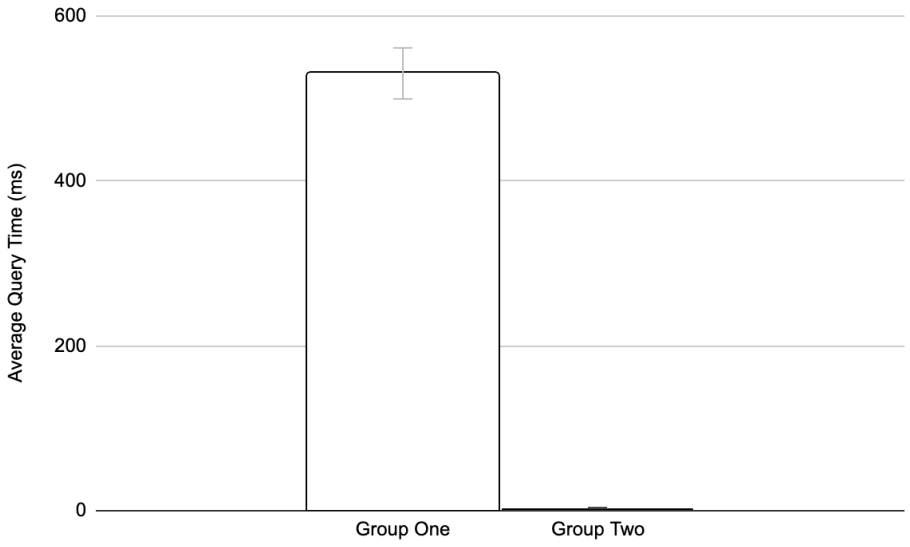

Figure 25. Test 3 Avg. Query Times with Standard Error Bar

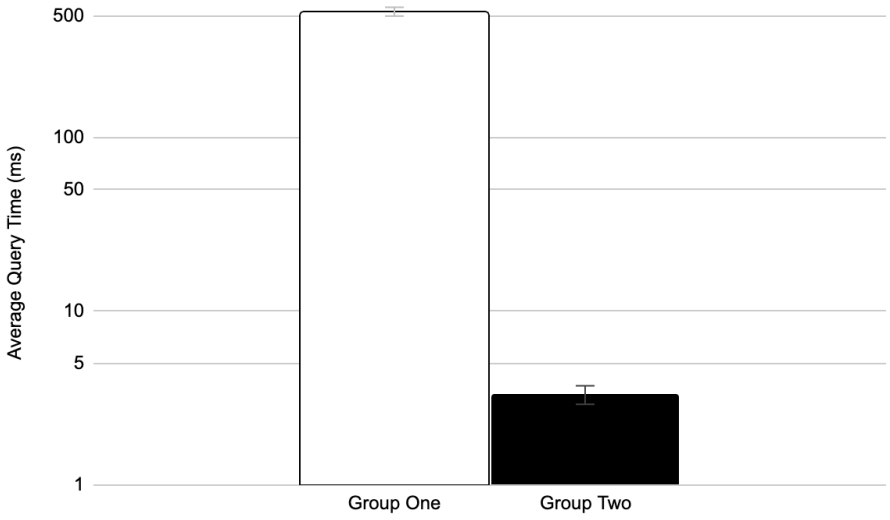

Figure 26. Test 3 Avg. Query Times with Standard Error Bar

(Log scale for better readability)

The results for test 3 found that there was a statistically significant difference between the query times for the two groups, $t(29) = 17.1$, $p = < 0.001$. Assembling the data needed to satisfy the query using semantic information from cached data saw vast improvements over querying the MongoDB database (Group One Mean = 530.52ms, Group Two Mean = 3.32ms) and had far less variation in the query times (Group One SD = 169.58, Group Two SD = 2.22).

Test 4: Queries with Union/Subset Checking vs Queries with Semantic Info

Paired Samples T-Test

			statistic	df	p
Group 1	Group 2	Student's t	4.25	29.0	< .001

Descriptives

	N	Mean	Median	SD	SE
Group 1	30	10.62	8.50	9.12	1.666
Group 2	30	2.98	2.20	2.25	0.410

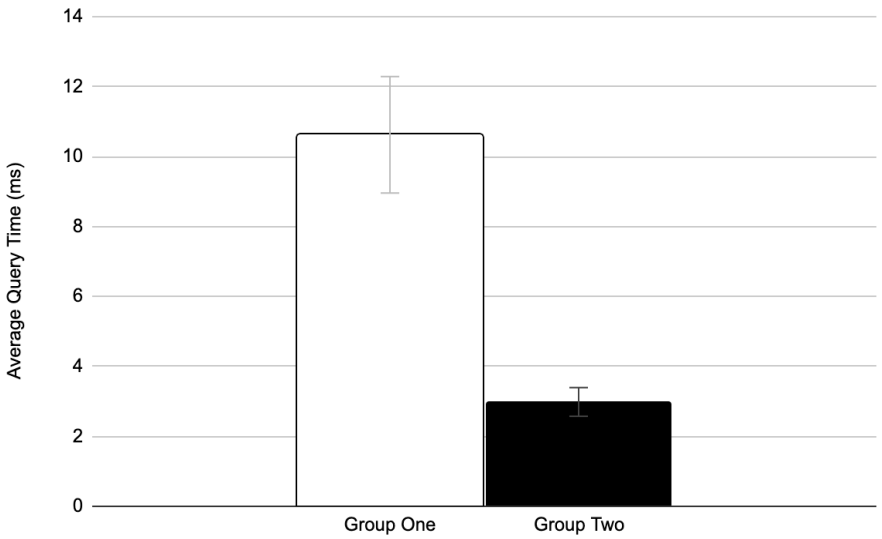

Figure 27. Test 4 Avg. Query Times with Standard Error Bar

The results for test 4 found that there was a statistically significant difference between the query times for the two groups, t(29) = 4.25, p = < 0.001. Pulling the data from already cached data using semantic information saw further improvements over parsing the query and pulling data from Postgres using Union/Subset checking (Group One Mean = 10.62ms, Group Two Mean = 2.98ms).

Test 5: Running and Caching Queries in Postgres vs Running Queries and Not Caching

Paired Samples T-Test

			statistic	df	p
Group 1	Group 2	Student's t	0.176	29.0	0.862

Descriptives

	N	Mean	Median	SD	SE
Group 1	30	582.29	538.17	148.77	27.161
Group 2	30	578.39	569.90	108.08	19.732

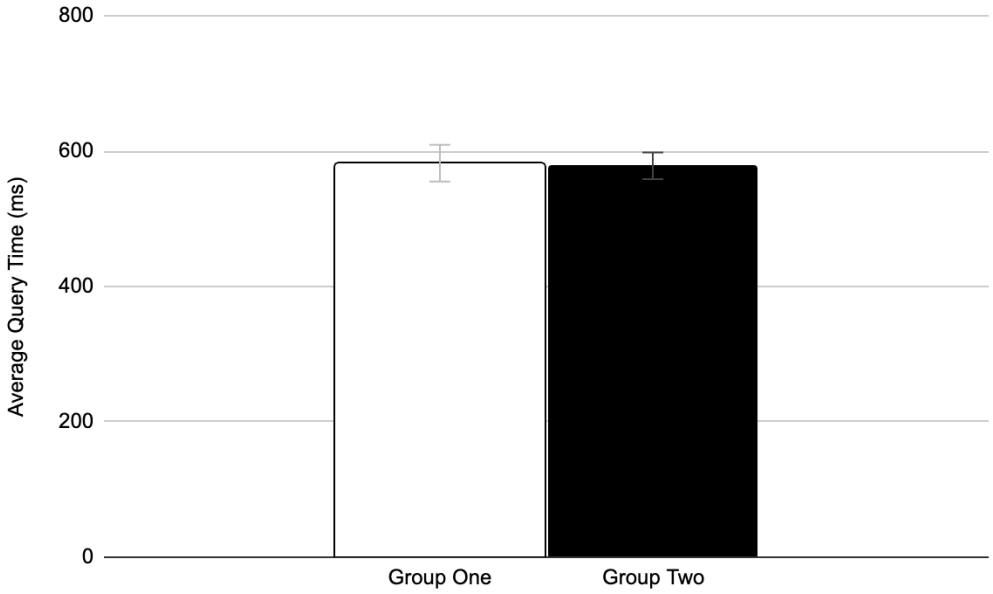

Figure 28. Avg. Query Times with Standard Error bar

The results for test 5 found that there was not a statistically significant difference between the query times for the two groups, t(29) = 0.176, p = 0.862. The additional time it takes to cache MongoDB aggregate queries into Postgres, as opposed to simply running the query, is negligible and therefore only provides the user with dramatic increases in query time performance (Group One Mean = 582.29ms, Group Two Mean = 578.39ms).

7. Conclusion

The results found that caching MongoDB aggregate query information into Postgres dramatically accelerated query time performance. Each of the four checkpoints were found to greatly reduce query times. Additionally, test 5 found that the additional time that it takes to cache the query data and parsed query information into Postgres, as opposed to simply querying the MongoDB database, is negligible and therefore only provides the user with an immense boost to query time performance. Therefore, initially implementing this system into the backend of an application (part of a program concerned with database functionality) wouldn't create any noticeable deceleration in overall database performance.

Of course, these improvements assume that the user intends to revisit cached data, uses a reasonably strict aggregate query pipeline structure, and follows mindful naming practices in regards to the statistical names. However, as is the case with most database usage, users will frequent the same data or slight variations of that data.
It is also important to note that the improvements to query times assume that the user has access to the cached data on a local server. Throughout the duration of this project, when the MongoDB database was queried, the query was sent to MongoDB servers which are located in Virginia and hosted on the AWS cloud. On the other hand, when cached data in Postgres was referenced, the query was sent to a localhost on the same computer, as the data was stored locally.

The future of this project would best be served by further developing ways in which queries can be analyzed and parsed, such that the program would be better able to recognize relationships between more generic queries. This would enable the user to use the program more casually and allow for the writing of aggregate queries to be less constraining. Additionally, further research could be done into how well query time performance would be improved if the cached data were not hosted locally, but on a server elsewhere. This might make the project more suitable for situations in which the user has an amount of data that is large enough to require storage in a database that cannot be hosted locally.

All in all, this project was successful in optimizing query run times by caching query data and information within Postgres. This was made possible by developing algorithms that utilize parsed query information and semantic information in order to make judgments about relationships between cached data and new queries. The ability to not only retrieve data from cached queries but also satisfy new uncached queries by parsing and comparing the query to cached query information results in dramatic decreases in query run times. Additionally, the latter provides this acceleration of query times without increasing the amount of storage needed. Finally, the additional layer of allowing the user to enter semantic information further improves query run times and provides the user with the option of greater flexibility and control over the data.

Bibliography

Abramova, Veronika, and Jorge Bernardino. “NoSQL Databases.” *Proceedings of the International C* Conference on Computer Science and Software Engineering - C3S2E '13*, (2013): 14–22. https://doi.org/10.1145/2494444.2494447.

“Aggregation.” Aggregation - MongoDB Manual. Accessed December 2, 2021. https://docs.mongodb.com/manual/aggregation/.

Aron, Arthur, Elaine N. Aron, and Elliot J. Coups. *Statistics for Psychology*. Upper Saddle River, New Jersey: Pearson Education, 2013.

Chamberlin, Donald D. “Early History of SQL.” *IEEE Annals of the History of Computing* 34, no. 4 (2012): 78–82. https://doi.org/10.1109/mahc.2012.61.

Chodorow, Kristina. *MongoDB: The Definitive Guide*. Sebastopol, CA: O'Reilly, 2013.

Copeland, Rick. *MongoDB Applied Design Patterns*. Beijing, China: O'Reilly, 2013.

“Data Catalog.” Datasets - CKAN. Accessed September 12, 2021. https://catalog.data.gov/dataset?groups=older-adults-health-data&res_format=JSON&page=1.

Obe, Regina O., and Leo S. Hsu. *PostgreSQL: Up and Running*. Sebastopol, CA: O'Reilly, 2015.

Parker, Zachary, Scott Poe, and Susan V. Vrbsky. “Comparing NoSQL Mongodb to an SQL DB.” *Proceedings of the 51st ACM Southeast Conference on - ACMSE '13*, (2013). https://doi.org/10.1145/2498328.2500047.

Ullman, Jeffrey D., and Jennifer Widom. *A First Course in Database Systems*. Seconded. Upper Saddle River, New Jersey: Prentice-Hall, 2002.