Database Optimization in Python

Jimmy Gray-Jones

Table of Contents

The Problem

Databases and their importance (1.1)

Databases are found in every facet of modern life. Social media, banking, universities. . . the list goes on. Modern databases, especially when it comes to more of the popular applications, contain nearly inconceivable amounts of information.

They are the backbone for all modern technological infrastructure because they hold all forms of information we rely upon every day. And as more and more people join the technological sphere, the size and amount of databases needed to hold people's information grows larger

Issues with large scale databases (1.2)

As more and more people join the never-ending amounts of websites, applications, and other digital market-fronts, some issues begin to develop as a database grows larger. Some of these issues include:

- Scalability as size of data increases
- Performance & Maintaining Performance
- Processing Speed
- Data Safety
- Data Management / Distribution
- Data Integration

All of these are very important to not only recognize and understand, but to also adapt to as well. Because if these issues are not addressed as a database continues to take in more and more information, then eventually it can lead to things like a database being corrupted, slower times in receiving information, or even the total inability to access a database

A proposed solution (1.3)

My project was focused on trying to optimize retrieval of information from databases, specifically focusing on the issues of scalability as size of data increases, performance and maintaining performance (as size of data increases), and processing speed. I chose these issues specifically because they seemed like the most important issues out of the ones listed, and they were the most achievable with what tools I had.

To accomplish this, I decided to use python's sqlite3 module to create a database on the jupyterhub server where I focused on optimizing queries using python's multiprocessing module to speed up data retrieval times from my database.

In this project, I specifically chose to focus on data retrieval times for all the data in a given table as my metric so as to not confuse and/or taint my results due to the timing differences that different queries might create.

The Project

Multiprocessing in Python (2.1)

Multiprocessing in Python3 comes in the form of a module that can be imported on one's IDE of choice. By default, this module uses all available cores to divide up any given multiprocessing task by a set number of workers. In the case of the MSU Jupyterhub server, there are 256 available cores for any given task to be divided amongst.

For this project, there was a focus on two specific functions that come from the Multiprocessing module; Multiprocessing.Pool() and Multiprocessing.Threadpool()

Multiprocessing.Pool() uses a process based approach to achieving parallelism; It takes the parameter of a function and a list of objects (or some form of iterable) and applies the given function to all of the objects in the list / iterable in parallel by dividing the task among "workers".

Multiprocessing.Threadpool() uses the same process as its Pool counterpart, except that instead of using "workers", it uses "Threads". All threads are placed in the same shared memory space, and can be useful for I/O (input/output) tasks that the CPU cannot do on its own.

Figure 2.1.1

```
# define the queries to be executed
queries = ["SELECT * FROM cancer_data"]

# execute the queries in parallel using a thread pool
start_time = time.time()
results = ThreadPool(4).map(query_db, queries)
end_time = time.time()
timing_data['Cancer Data'] = (end_time - start_time)
```

Defining a SQL query within a list that is fed into Multiprocessing.ThreadPool to run in parallel, given the query_db function

Figure 2.1.2

```
# define the queries to be executed
queries = ["SELECT * FROM cancer_data"]

# execute the queries in parallel using pool
start_time = time.time()
results = multiprocessing.Pool(4).map(query_db, queries)
end_time = time.time()
timing_data['Cancer Data'] = (end_time - start_time)
```

Defining a SQLi query within a list that is fed into Multiprocessing.Pool to run in parallel, given the query_db function

Setting up the base environment (2.2)

To begin setting up the environment for this project, I created a connection object using Python3's SQLite3 module to initialize the database in the MSU Jupyterhub server. And from this connection object, a cursor object was created to interact with the database.

Figure 2.2.1

```
conn = sqlite3.connect('cmse_401_project_database')
cur = conn.cursor()
```

Figure 2.3.1 shows the creation of connection and cursor objects, dubbed conn and cur. Below this, displays the database file created named

Figure 2.2.2

```
def query_db(query):
    conn = sqlite3.connect('cmse_401_project_database')
    cur = conn.cursor()
    cur.execute(query)
    if 'SELECT' in query:
        result = cur.fetchall()
        conn.close()
        return result
    conn.close()
    return
```

Figure 2.3.2 shows the creation of the "query_db" function,which takes in a given query and creates a connection and cursor object, then executes said query. If SELECT is present, then the results are shown, promptly by closing the connections at the end of it.

After the creation of the database, dubbed cmse_401_project_database, four tables were created from four datasets that were first converted into pandas dataframes then subsequently converted into sql tables using the pd.to_sql command. These four datasets of varying shapes and sizes function as my base datasets until I use artificially created data down the line.

Figure 2.2.3

```
print(cancer_data.shape)
print('')
print(us_school_data.shape)
print('')
print(best_buy_data.shape)
print('')
print(manga_data.shape)
```

```
(569, 33)

(1715, 266)

(15210, 6)

(67273, 31)
```

Figure 2.3.3 shows the rows and columns of the pandas tables that are later turned into sql tables

Figure 2.2.4

```
start_time = time.time()
cancer_data.to_sql(name='cancer_data', con=conn, if_exists='replace', index=False)
end_time = time.time()
print("Total run time:",end_time - start_time)
timing_data['Cancer Data'] = (end_time - start_time)
```

```
query_db("""SELECT id, diagnosis, radius_mean FROM cancer_data LIMIT 1""")
```

```
[(842302, 'M', 17.99)]
```

Figure 2.3.4 shows the conversion of the Pandas dataframe "cancer_data" being converted into a sqlite3 table. Below this, is a query sent into the query_db function that displays the result of sending in a query into the database, demonstrating that the Pandas dataframe was converted into a SQL table

Results

For each section of the results, there are three different categories that I used to benchmark the speed of data retrieval from the given tables;

Unoptimized Query Times, which are simply the queries being run without the involvement of multiprocessing. Optimized Query Times w/ ThreadPool (OQWT), which are the queries that were run with Multiprocessing.Threadpool(), and Optimized Query Times w/ Pool (OQWP), which are the queries that were run with Multiprocessing.Pool()

Base Datasets (3.1)

For the base datasets, the queries that were executed the fastest overall were a mixture between the OQWT and the serially run queries. This was a bit of a confusing result, as I had assumed that both versions of my optimized queries would execute faster than the serially run queries.

Instead however, the OQWP times were double the times of the serially run queries and the OQWT times. These results however gave me some insight into the fact that data retrieval from the database is most likely an I/O task, and the CPU has no impact on how quickly this can get done.

Even with some of the OQWT times being faster than the serially run queries, the erraticness of the runs were something that I decided to investigate further using the artificial datasets.

Artificial Datasets (3.2)

I started off by benchmarking just a single run of the higher data, including the manga_data data frame as a good reference in these results as well (see figures 5.2). I displayed the serially run queries, the OQWT times, and the OQWP times using histograms and for all three I got a staircase style visual from smallest to largest of the data.

This was both expected and something I wanted as a means to both ensure that the results for my base datasets were replicable and carried on into greater sizes of data, with OQWT and Serial Query Times having mixed results of execution times while OQWP having the worst.

In an attempt to get a better visual of the data however, I decided to run all three of my query categories twenty times, starting with zero threads/worker and adding a thread/worker with each iteration of executing the queries.

Figure 3.2.1

```
for i in range(1,21):
    start_time = time.time()
    results = ThreadPool(i).map(query_db, queries)
    end_time = time.time()
    parallel_query_times.append(end_time - start_time)
    print(end_time - start_time)
```

Figure 3.2.1 shows a simple for loop where i is increasing the number of threads for Multiprocessing.ThreadPool() to use

I did this for each of my five tables in succession, and got the graphs displayed in section 5.3. Observing the results of the twenty runs of the higher data was fascinating to me, because I

noticed that for both OQWT and the serially run queries that the execution times became more erratic the greater the data became.

For the OQWT however, the variance in speeds was far greater in comparison to the serially run queries and a majority of the OQWP (with the exception of table four), with the greatest amount of data having the greatest variance in speeds.

I was shocked to learn that the OWQT execution times had a nearly exponential increase as the amount of workers increased for table four. The other tables for the OQWP had surprisingly very little variation, so I am unsure of what caused the results to be that way but it was a recurring thing.

To further visualize the data, I decided to condense all of the times for each of my query categories into three scatterplots that allow me to do the following:

(1) Gain a comparison estimation of serially running all of the queries vs. actually parallelizing all of the queries in a single list

(2) Provide a hypothetical time for the parallelization of the unoptimized queries. This is extremely useful and important, because sqlite3 does have built-in means for parallelization of queries. So I am comparing figure 5.4.1 to figures 5.5.1 and 5.5.2 as a means to prove whether parallelization of queries (with either workers or threading) is faster than a hypothetical parallelization of my unoptimized queries.

Artificial Datasets w/ Query Parallelization (3.3)

Finally, after benchmarking and visualizing all of the artificial data after twenty serial runs as well as combining them to obtain an assumed average of what each of my three query categories should be if I were to run them in parallel, I decided to benchmark running all of the retrieval queries in parallel with my OQWT and my OQWP using the same method in figure 3.2.1.

Figure 3.3.1

```
queries = ["SELECT * FROM Manga_data",
           "SELECT * FROM table_one",
           "SELECT * FROM table_two",
           "SELECT * FROM table_three",
           "SELECT * FROM table_four"]
```

Figure 3.3.1 shows all 5 queries written into a single list to be run all at once, instead of sequentially

Because sqlite3, again, doesn't have a built-in method for parallelizing queries, I used the combined average times of the twenty runs from the serial runs as an assumed "serial" parallelized time.

The results for the OQWT and my OQWP were both extremely surprising and interesting for me. All of the benchmarking within the project so far has ended up with my OQWT yielding the fastest times, followed by the queries run in serial, then the OQWP.

However when running the queries in parallel, the OQWP were actually faster than both the assumed parallelized time of the serial queries and the OQWT. The times for the OQWT were growing nearly exponentially as threading with each iteration. For the OQWP, the timings started out high, but got lower with each iteration until a rough limit was reached that was nearly two seconds faster than the combined serial times. You can see this by comparing both figures in 5.5 and 5.4.1.

Conclusions and Future Work

From all of the data visualizations and execution times I have gathered throughout this project, I have come to a few definitive conclusions:

For non-parallelized queries, serially running queries is best. While Multiprocessing.Threadpool() does oftentimes yield faster results for data retrieval, the erraticness of it might not make it very practical for use.

For parallelization of queries (running x queries at once), Multiprocessing.Threadpool() is the best way to go about it as opposed to serially running all of your desired queries.

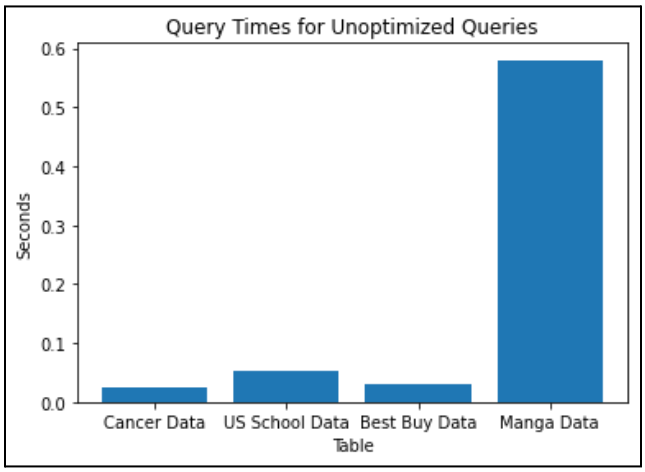
While Multiprocessing.Threadpool() and Multiprocessing.Pool() both speed up the programs, the average improvement in time is not by much. Likely by a few seconds at most, though this difference is likely to increase with the greater the data used. But for practical applications on databases, it is probably not necessary.

In the future, I'd like the ability to work with databases that are outside of the scope of Sqlite3 and the MSU Jupyterhub server. There are several softwares that utilize sql to connect to local and online databases alike, that I could use as a more "realistic" scenario for trying to optimize a database

Figures

Base Dataset Times (5.1)

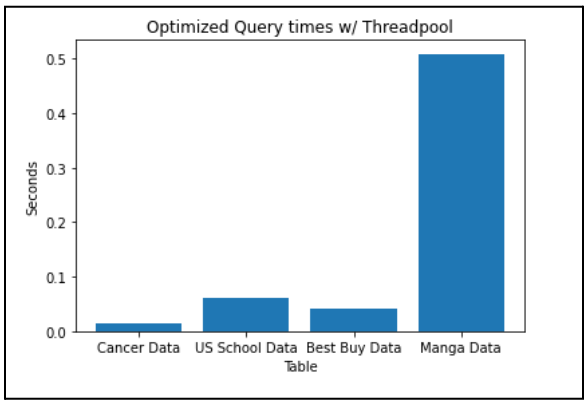
Figure 5.1.1




```
{'Cancer Data': 0.024936199188232422,
 'US School Data': 0.05270814895629883,
 'Best Buy Data': 0.02920699119567871,
 'Manga Data': 0.5801727771759033}
```

Figure 5.1.1 shows the unoptimized query times for the four base datasets

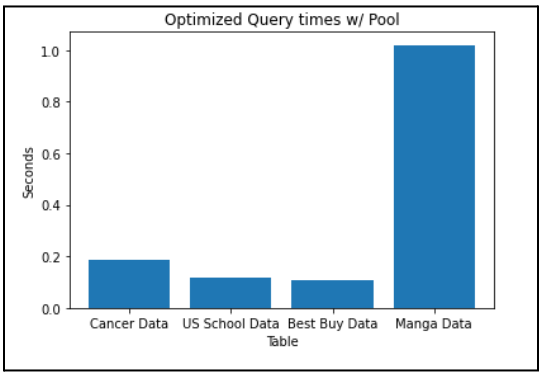
Figure 5.1.2




```
{'Cancer Data': 0.01537466049194336,
 'US School Data': 0.059957265853881836,
 'Best Buy Data': 0.04099750518798828,
 'Manga Data': 0.5090718269348145}
```

Figure 5.1.2 shows the optimized query times using Threadpool for the four base datasets

Figure 5.1.3

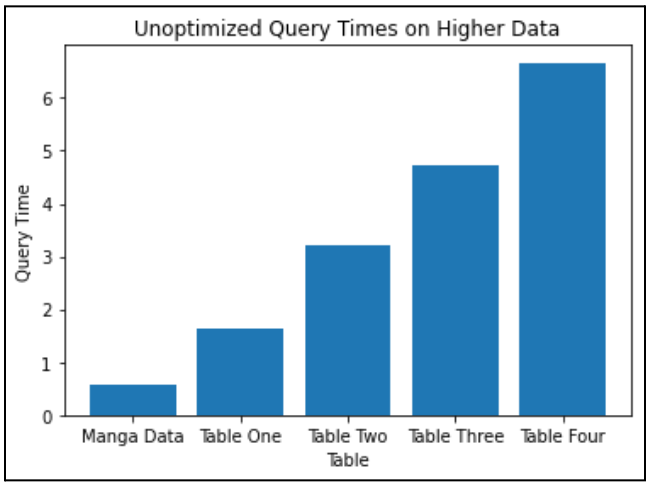



```
{'Cancer Data': 0.18683791160583496,
 'US School Data': 0.11629295349121094,
 'Best Buy Data': 0.10711908340454102,
 'Manga Data': 1.0195527076721191}
```

Figure 5.1.2 shows the optimized query times using Pool for the four base datasets

Artificial Dataset Times [Single Runs] (5.2)

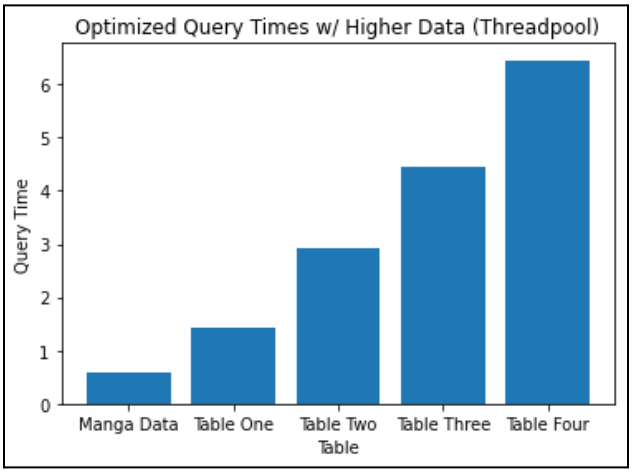
Figure 5.2.1



```
{'Manga Data': 0.5808172225952148,
 'Table One': 1.634765625,
 'Table Two': 3.227726459503174,
 'Table Three': 4.718907833099365,
 'Table Four': 6.664623498916626}
```

Figure 5.2.1 shows the unoptimized query times for the artificial data

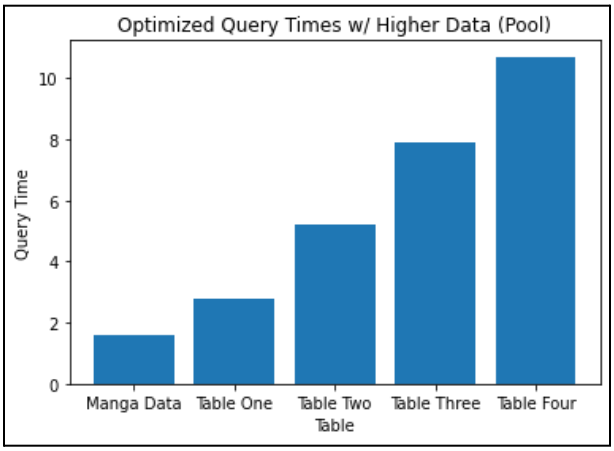
Figure 5.2.2




```
{'Manga Data': 0.5775856971740723,
 'Table One': 1.421968936920166,
 'Table Two': 2.918689012527466,
 'Table Three': 4.460422039031982,
 'Table Four': 6.4522058963775635}
```

Figure 5.2.2 shows the optimized query times using Threadpool with the artificial data

Figure 5.2.3



```
{'Manga Data': 1.5788512229919434,
 'Table One': 2.7909162044525146,
 'Table Two': 5.197488307952881,
 'Table Three': 7.879606246948242,
 'Table Four': 10.710070848464966}
```

Figure 5.2.3 shows the optimized query times using Pool with the artificial data

Artificial Dataset Times [Twenty Runs] (5.3)

Figure 5.3.1

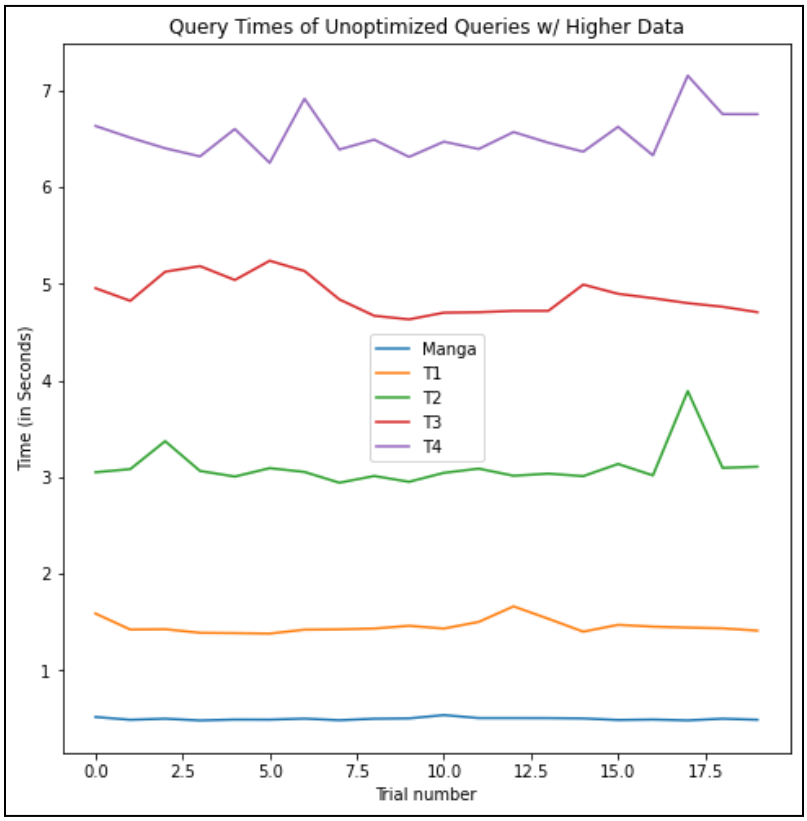


Figure 5.3.1 shows the unoptimized query times over 20 iterations with the artificial datasets

Figure 5.3.2

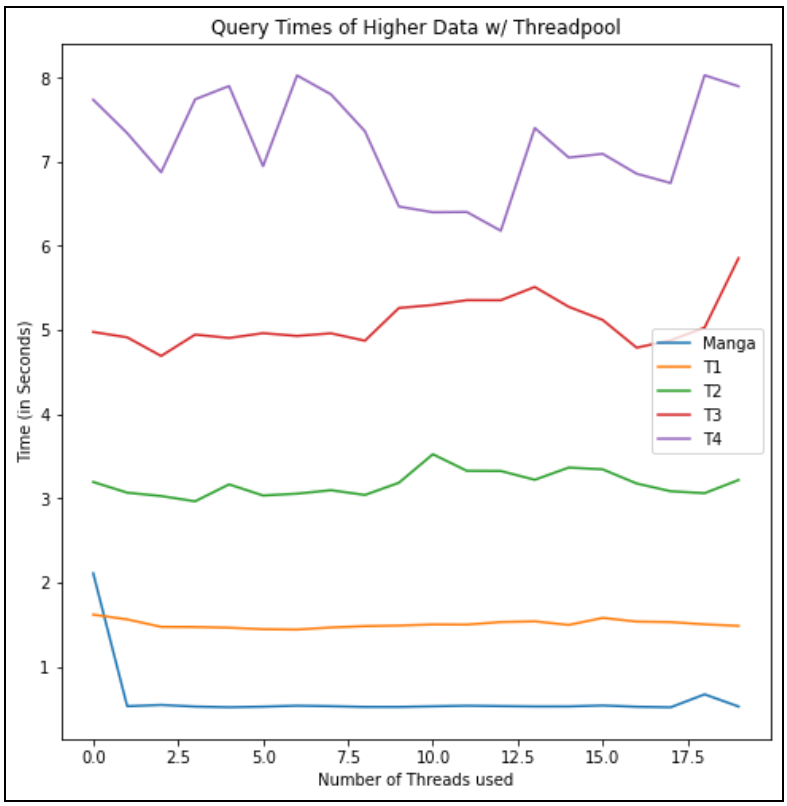


Figure 5.3.1 shows the optimized query times using Threadpool over 20 iterations with the artificial datasets

Figure 5.3.3

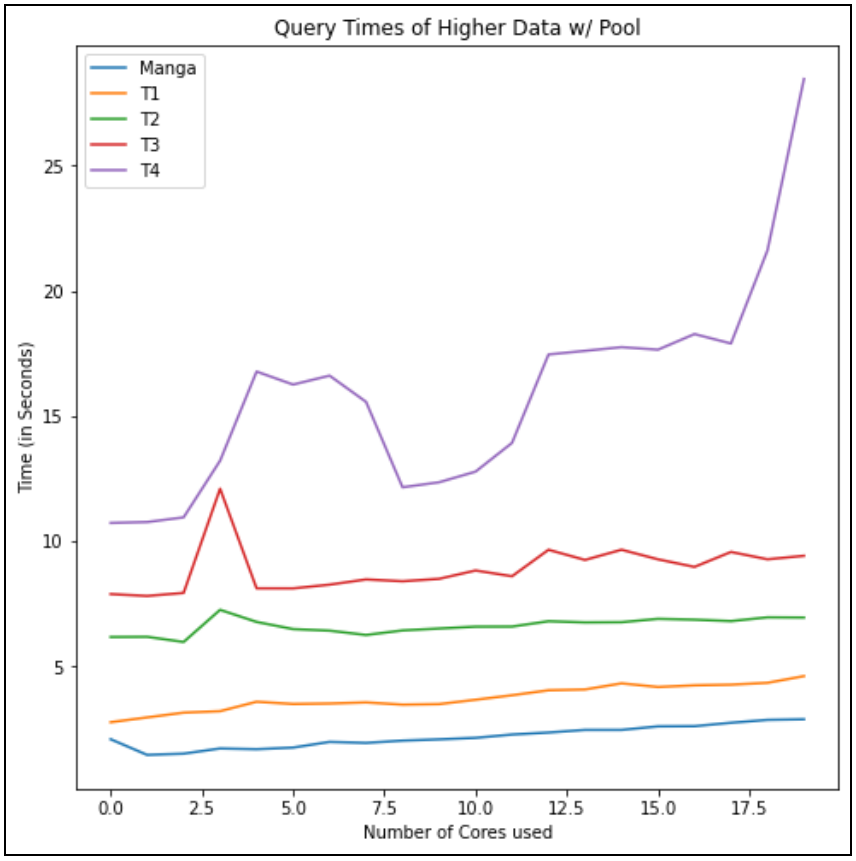


Figure 5.3.3 shows the optimized query times using Pool over 20 iterations with the artificial datasets

Artificial Dataset Times [Combined Twenty Runs] (5.4)

Figure 5.4.1

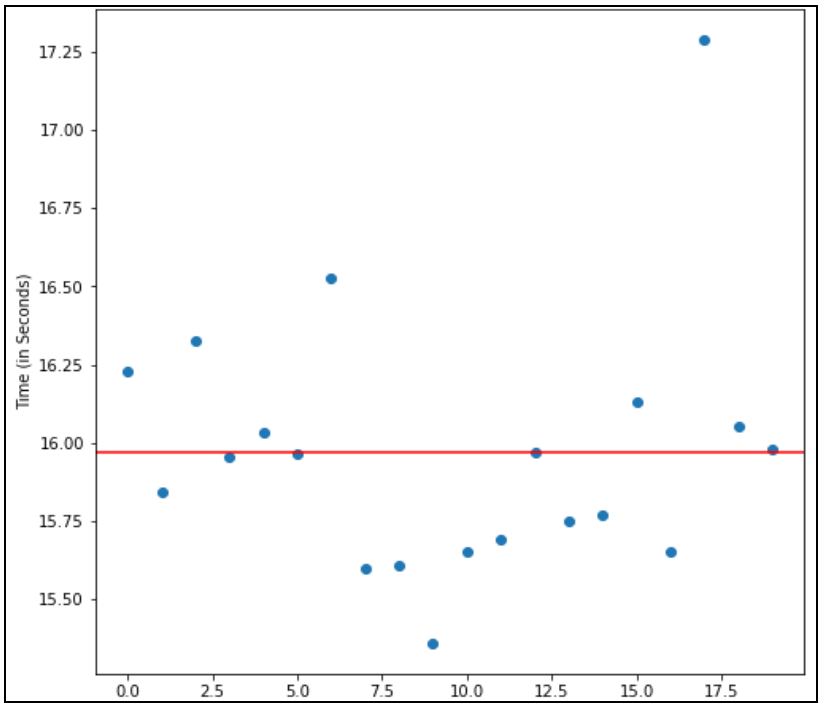


Figure 5.4.1 shows the combined unoptimized query times over 20 iterations with the artificial datasets

Figure 5.4.2

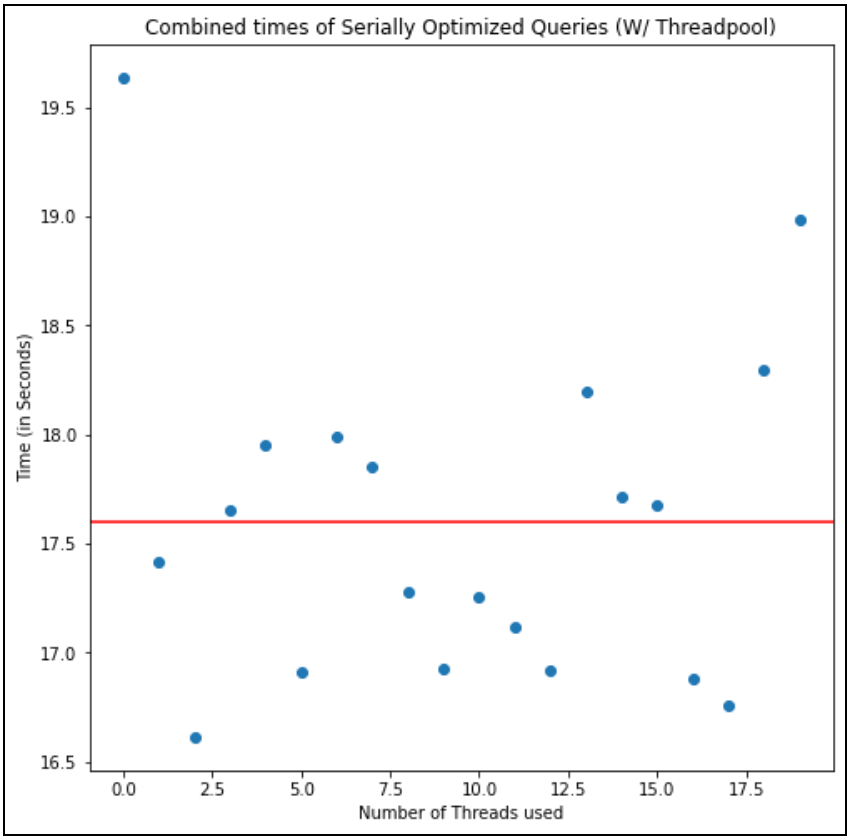


Figure 5.4.2 shows the combined optimized query times using Threadpool over 20 iterations with the artificial datasets

Figure 5.4.3

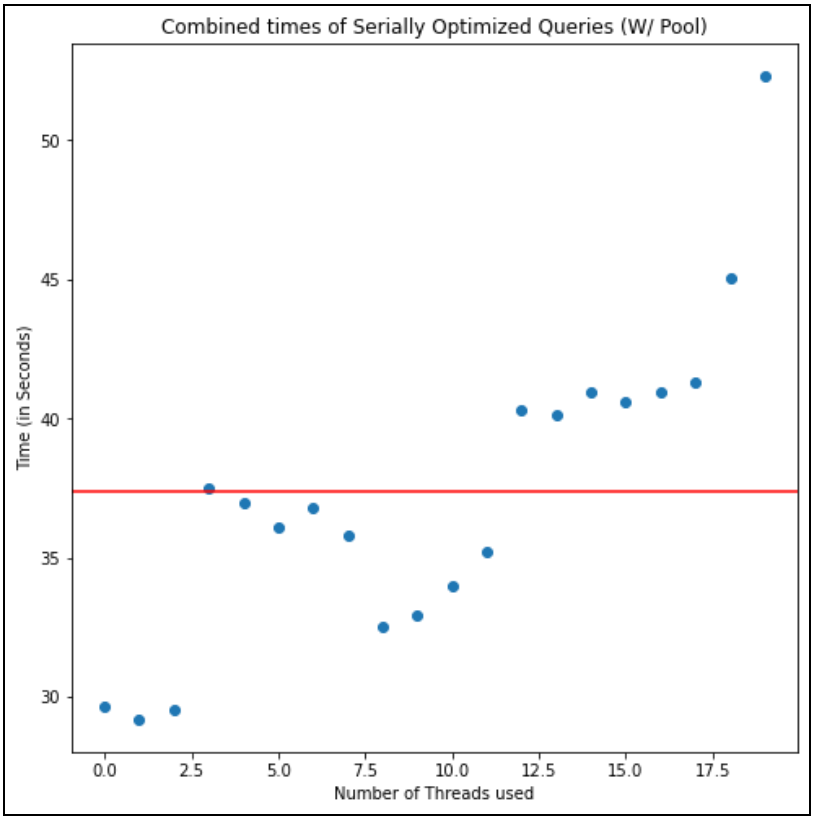


Figure 5.4.3 shows the combined optimized query times using Pool over 20 iterations with the artificial datasets

Artificial Dataset Times w/ Parallelization of Queries (5.5)

Figure 5.5.1

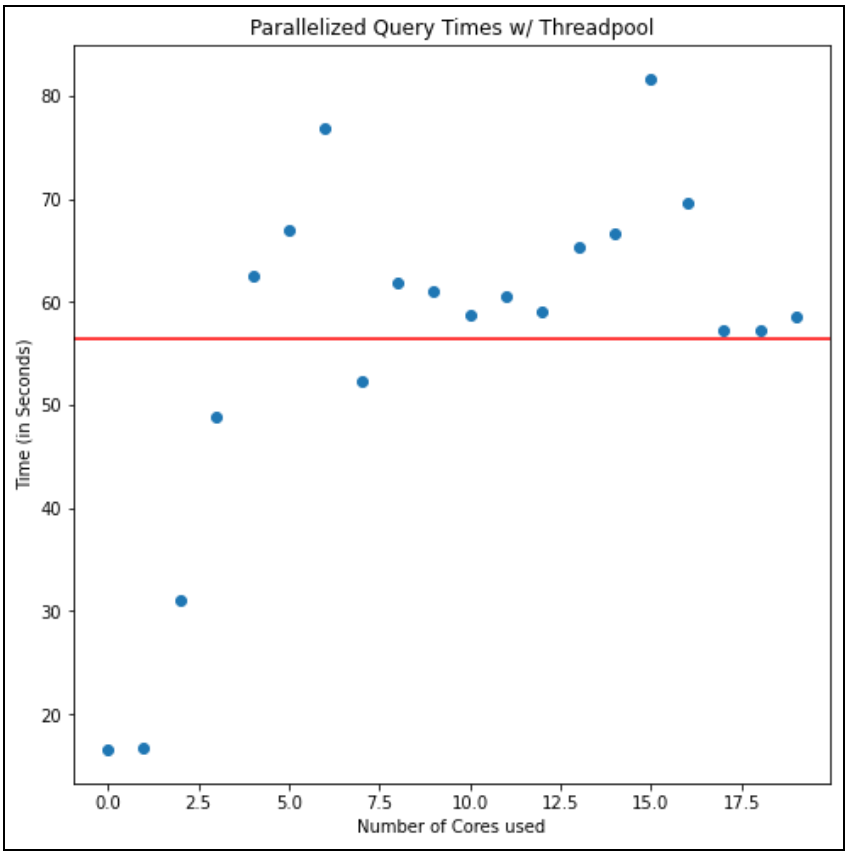


Figure 5.5.1 shows the combined optimized query times in parallel using Threadpool over 20 iterations with the artificial datasets

Figure 5.5.2

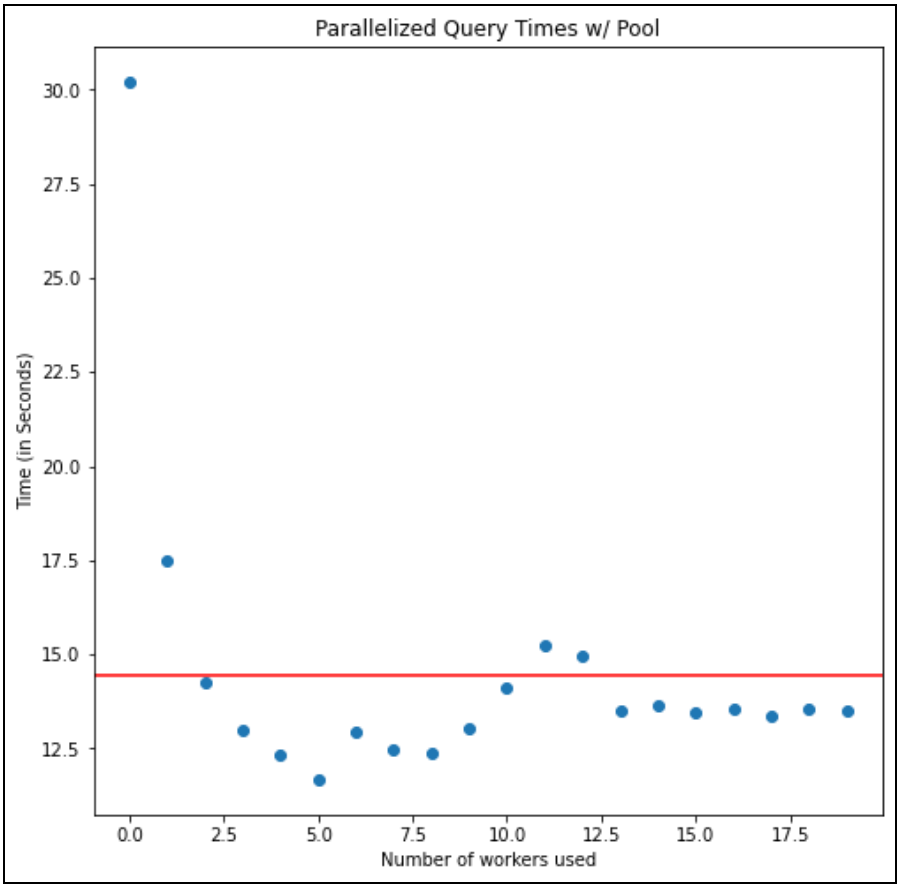


Figure 5.5.2 shows the combined optimized query times in parallel using Pool over 20 iterations with the artificial datasets

References and Specs

https://tudip.com/blog-post/9-common-database-management-challenges-and-how-to-fix-them/

https://www.buchanan.com/improve-database-performance/

https://medium.com/geoblinktech/parallelizing-queries-in-postgresql-with-python-572995ae340

https://www.geeksforgeeks.org/running-queries-in-python-using-multiprocessing/

https://cs.stackexchange.com/questions/86893/what-does-an-i-o-task-mean

Computer Specs:

Device: Dell Laptop

System Manufacturer: Dell Inc.

OS Manufacturer: Microsoft Inc.

Processor: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz, 1992 Mhz, 4 Core(s), 8 Logical Processor(s)