CMPE 275: Mini 1 Report

Team NexaBytes

Introduction

For the given dataset of NYC Motor Vehicles Collisions containing approximately 2.15 million rows, we want to load the dataset and query it using different search criteria optimally. The work is divided into three phases, each focusing on different aspects, utilizing parallelization strategies and memory layouts to optimize the loading and querying performance.

Phase 1: Base Design & Benchmark

In the first phase, we implemented a basic working Object Oriented design to read all the rows from the csv file and store them as a vector of Collision type objects. This acts as a data store for us, and on subsequent queries, Collisions which match the search criteria are returned successfully.

For the base design, we used primitive data types such as int and float for numeric values and string class for textual data. We defined these class members inside the Collision class. Each value is read from the CSV file and converted to the appropriate format, like number of persons injured being stored as an int and borough being stored as string. It is using custom methods to handle the data formatting before storing.

After loading the data, we are defining 5 key queries for the benchmark.

- Search by Zip Code (integer equality)
- Search by Injury Ranges (integer min-max range)
- Search by Latitude (float equality)
- Search by Date Range (integer min-max range special case)
- Search by Borough (string equality)

One specific area we want to highlight is that we are storing Date and Time as int, and not string. It is stored in the format YYYYMMDD, which makes it easier to perform comparison. This could have been a potential optimization in the future, but we chose to do so here itself in order to enable date range search queries using a simple date comparison step.

We ran the code 15 times using a bash script to test benchmarks over a large sample size to get a higher accuracy of average performance. These metrics values act as our benchmark representative values for phase 1, and will be used to compare with the performance metrics of the following phases.

Metric	Benchmark Time (ms)
Load Time	18707.75
Zip Code Query	550.02
Injury Range Query	461.02
Latitude Query	435.16
Date Range Query	465.38
Borough Query	744.11

As mentioned previously, we ran the code over 15 runs to obtain higher accuracy for our observed values. The graph showing the same is shown below. We have done the same for all phases.

Phase 2: Parallel Processing

In this phase, we took the code from phase 1 and used OpenMP to implement the same code using parallel processing. We did this in 3 steps and observed the improvements in each. The code can be largely divided into three steps - load, parse, and query. We improved them one by one and observed the jump in each step for their particular task. All code changes were run 15 times in succession in similar conditions on the same system to maintain consistency and calculate the time measurements as accurately as possible, averaging over the 15 runs.

2a) Parallel Parsing

After loading the data from the csv file into a shared variable line-by-line, we started parsing the data into the Collisions vector using OpenMP. We observed an improvement of around 25% in the loading time when compared to phase 1. This was achieved using the default number of threads and the inbuilt ‘pragma omp for’ directive. The other sections, i.e., querying and loading were not done using parallel processing.

Metric	**New Time (ms)**	**Benchmark Time (ms)**	**%Improvement**
Load Time	14019.63	18707.75	25.06%

As mentioned previously, adding parallelization to parsing led to an increase of around 25% to the benchmark value. We also observed some improved performances to the querying phase despite not implementing parallel processing for that phase yet.

2b) Parallel Querying

Now, for this one, we added parallelization to the querying steps using OpenMP, and observed the changes in the data. The loading time remained similar as that section is the same as 2a, but the querying times improved drastically, as high as 97% improvement over the benchmark value from phase 1.

Metric	**New Time (ms)**	**Benchmark Time (ms)**	**%Improvement**
Zip Code Query	137.09	550.02	75.08%
Injury Range Query	57.62	461.02	87.50%
Latitude Query	12.23	435.16	17.66%
Date Range Query	20.86	465.38	97.19%
Borough Query	309.14	744.11	58.46%

This was a drastic improvement over the previous query measurements.

2c) Parallel Loading

With this step, we started dividing the data into chunks of equal size during the read step of the code. Instead of loading the data to a variable and parsing it separately in threads using ‘pragma omp for’ directive (like we did previously), we manually split the data into chunks and parsed it at the same time, combining the two steps at once. We observed an improvement of around 42% from the 2a step and around 56% from the original benchmark.

Metric	**New Time (ms)**	**Benchmark Time (ms)**	**%Improvement**
Load Time	8118.81	18707.75	56.60%

Phase 2 Summary

We observed a strong improvement again through creating chunks, and hence, this will act as the benchmark representative for phase 2, along with the query performance metrics from 2c. The values are shown in the table below.

Metric	**Benchmark Time (ms)**
Load Time	8118.81
Zip Code Query	119.66
Injury Range Query	52.47
Latitude Query	11.83
Date Range Query	19.72
Borough Query	312.07

Phase 3: Optimizations

3a) Object of Arrays

In this phase, our entire goal is to optimize the code using different approaches. The primary change required is to rewrite the class to enable Object-of-Arrays (OoA) instead of the traditional Array-of-Objects (AoO). This is akin to columnar databases where consistent data is stored together.

Instead of storing a vector of Collision objects, we create a single Collisions object containing vectors of all class members. For instance, Borough is now stored as a vector<string> with the index representing a particular row from the csv.

As the code is still parsed row-by-row from the csv, we observed a similar performance in the loading time. However, the major performance improvements were found in the query times. Previously, the queries would run over the objects one-by-one and check the particular field, whereas now it only needs to access the same vector each time, which we can iterate over directly. It also only stores the matching indices for retrieval instead of the whole object in a vector<int> format.

In this process, we again saw major improvements in the query times with a minor penalty for the loading time. This can be seen in the table below.

Metric	**New Time (ms)**	**Benchmark Time P1 (ms)**	**%Improvement P1**	**Benchmark Time P2 (ms)**	**%Improvement P2**
Load Time	8508.43	18707.75	54.52%	8118.81	-4.80%
Zip Code Query	1.70	550.02	99.69%	119.66	98.58%
Injury Range Query	1.32	461.02	99.71%	52.47	97.48%
Latitude Query	1.24	435.16	99.71%	11.83	89.52%
Date Range Query	1.27	465.38	99.73%	19.72	93.56%
Borough Query	7.89	744.11	98.94%	312.07	97.47%

With these optimizations made to the code, the queries started responding in a fraction of previous benchmark times, showing extremely high optimization values. We have also plotted the same.

3b) Borough Encoding Optimization

Since there are a limited number of boroughs, we decided to attempt optimization using encoding through an unordered map. Each value out of the given 4-5 distinct values for the field corresponds to a fixed index in the map. As we iterate and find new values, the map grows dynamically and stores an integer for a given borough string.

This helps with optimizing memory usage as an int is much better compared to a string due to its smaller compact size. This is especially useful for the Borough matching query because now we are comparing integers instead of strings each time, which makes the processing simpler.

As we can also see, by changing the data type to int and using a map given the limited values, we observed a jump of 64.51% in query performance from 3a to 3b. This is a huge jump given the fact that this was our worst-performing query consistently due to string matching.

Metric	**New Time (ms)**	**Benchmark Time (ms)**	**%Improvement**
Borough Query	2.80	7.89	64.51%

Moreover, it is worth mentioning that we did something similar for the Crash Date variable in phase 1 itself, which optimized its query off the bat. The reason we did it was the same, comparing int values is much simpler for the system as compared to string values, and through data type conversion, we are able to speed up the process of querying at some expense of loading and parsing.

Miscellaneous Attempts & Ideas

Vector Shrinking Optimization

As part of our optimization efforts, we attempted to reduce the memory footprint of our program by applying shrink_to_fit() to vectors used for storing query results. Since our dataset is large and query operations generate dynamic vectors, we wanted to see if shrink_to_fit() could help free up excess memory.

The assumption was that vectors allocate extra capacity to avoid frequent memory reallocations, and calling shrink_to_fit() should allow us to trim the unused capacity and lower memory consumption. However, despite applying this method, we observed no significant reduction in overall memory usage.

Since query functions return vectors containing search results, we decided to apply shrink_to_fit() right before returning the result to ensure minimal memory overhead.

In this version, the query_result vector might be holding extra allocated memory beyond what is necessary to store the results.

We expected shrink_to_fit() to:
1. Reduce memory usage by trimming down vector capacity.
2. Improve overall memory efficiency, particularly in cases where queries returned large results.
3. Prevent memory waste when working with multiple queries in a short timeframe.

Even after applying shrink_to_fit(), total memory consumption remained the same before and after query execution. The function did shrink vector capacity, but the memory was not actually freed at the system level. Memory profiling showed that memory was still being held by the program, even when shrink_to_fit() was applied.

Optimized Memory Usage by Using Smaller Data Types

Replaced int with int16_t and int32_t where appropriate to reduce memory consumption. Converted categorical fields like borough into integer encodings (int16_t) to minimize string storage. Used float instead of double for latitude and longitude to optimize space. These changes improved memory efficiency, CPU cache utilization, and search performance while maintaining data precision.

Space Optimization Using reserve()

To optimize memory allocation and improve performance, we used the reserve method for vectors in the CollisionDataset::load_CSV function. By estimating the number of records based on the file size and an average record size, we pre-allocated memory for the vectors. This reduces the number of reallocations needed as elements are added, leading to better performance and efficient memory usage.

This approach ensures that the vectors have enough capacity to store the data without frequent reallocations, enhancing the overall efficiency of the data processing pipeline.

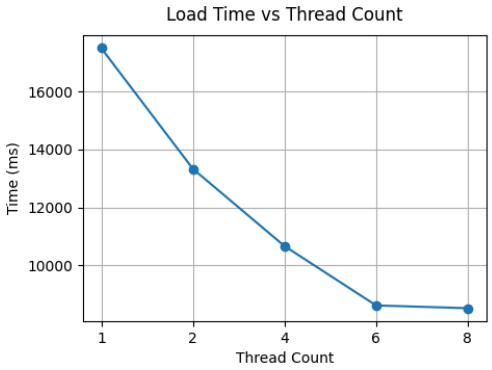
Notable Observations - Thread Count Optimization

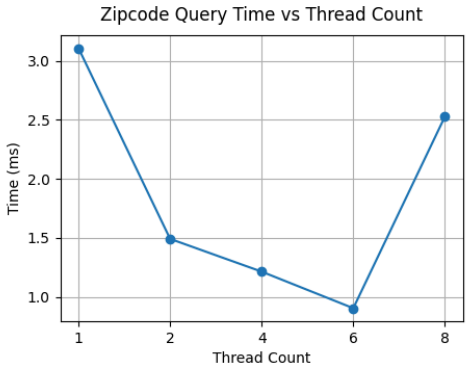
We explored thread count optimization during phase 3. The system we tested on was a Dell G3 15, which has 4 Intel cores. Through hyperthreading, we were able to attain a maximum of 8 threads for parallel processing. We measured the performance metrics for thread counts 1, 2, 4, 6, and 8 to observe the trends.

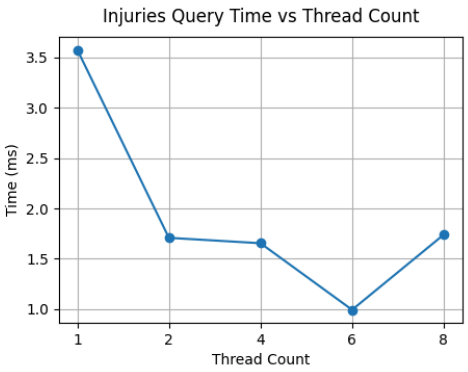
Thread Count ——— **Metric**	**1 Thread**	**2 Threads**	**4 Threads**	**6 Threads**	**8 Threads**
Load Time (ms)	17508.84	13332.86	10660.01	8605.88	8509.55
Zip Code Query (ms)	3.10	1.49	1.21	0.90	2.53
Injury Range Query (ms)	3.58	1.71	1.65	0.99	1.74
Latitude Query (ms)	5.76	1.31	1.03	0.96	1.72
Date Range Query (ms)	2.94	1.23	1.06	0.76	1.75
Borough Query (ms)	39.86	19.52	12.67	11.42	10.35

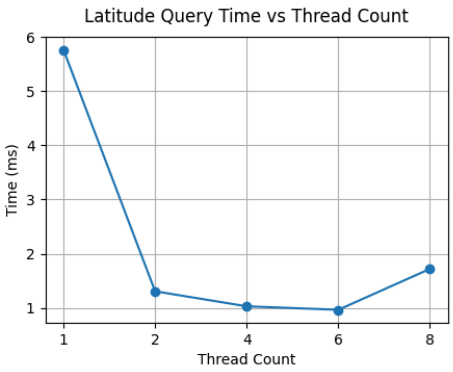
As observed, choosing the highest thread count (default value) is not always the optimal choice. In our case, we observed the best overall performance at 6 threads and not the maximum value of 8 threads. We also plotted these data points on a graph.

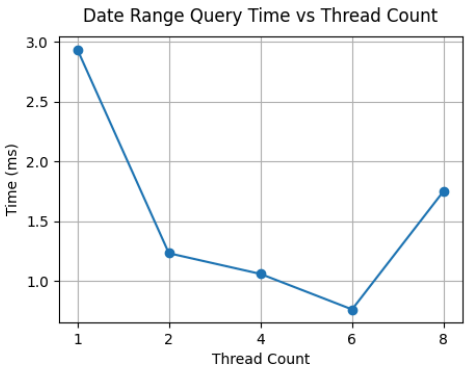
Performance Metrics Across Thread Counts

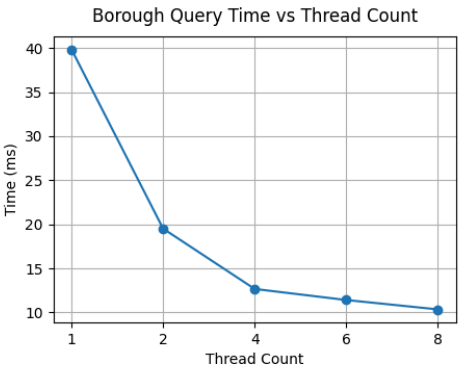




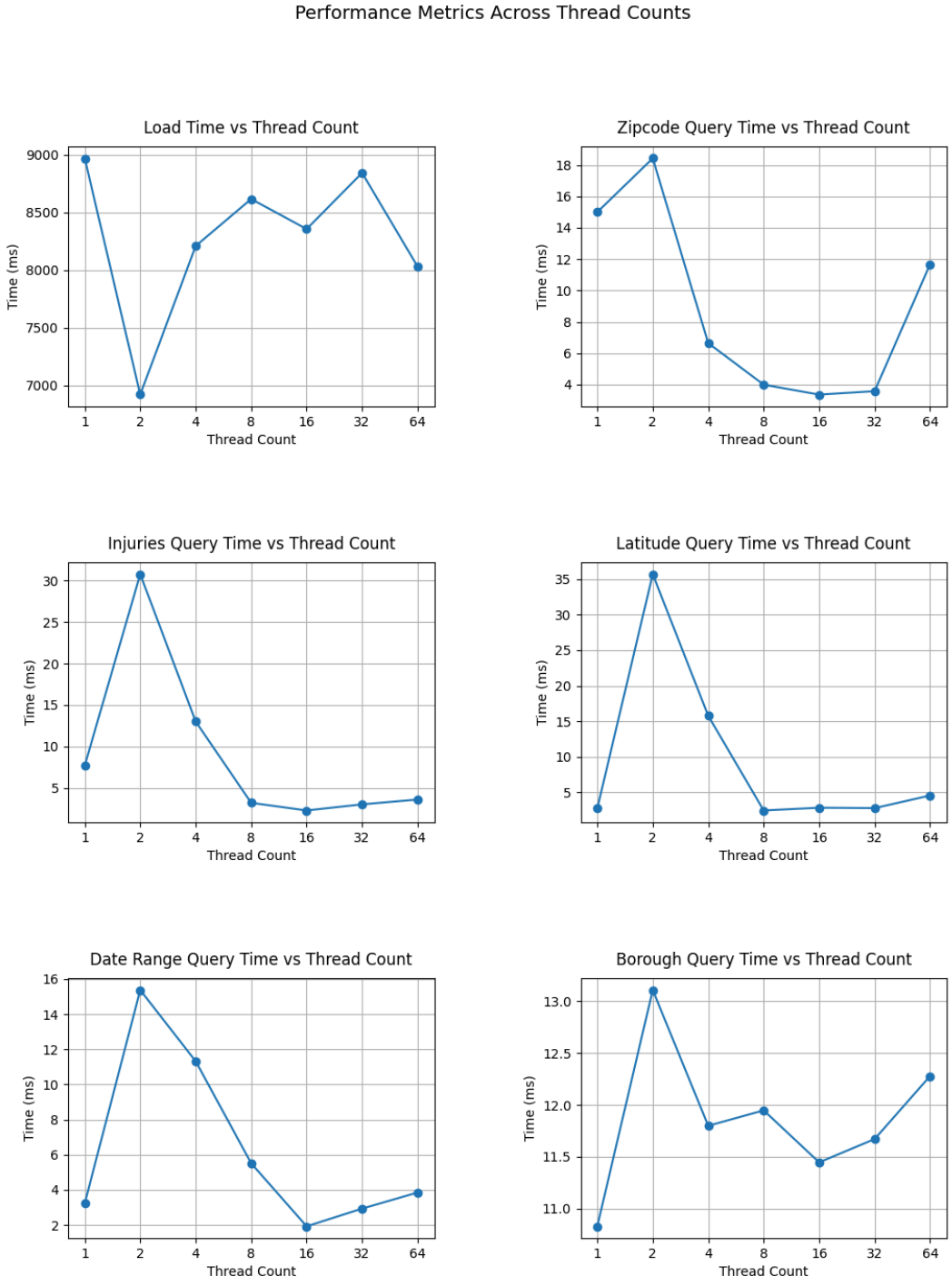






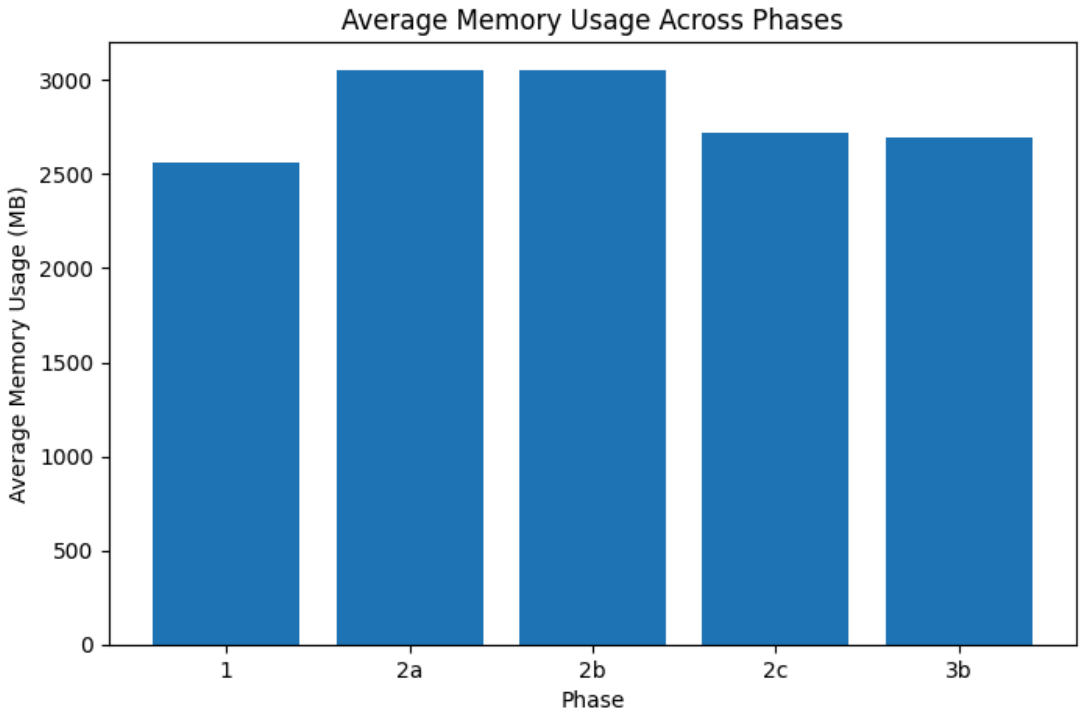


Similarly, we also tried running the same experiment on a more powerful system that enabled running upto 64 threads and plotted the data for the same.



Notable Observations - Memory Usage

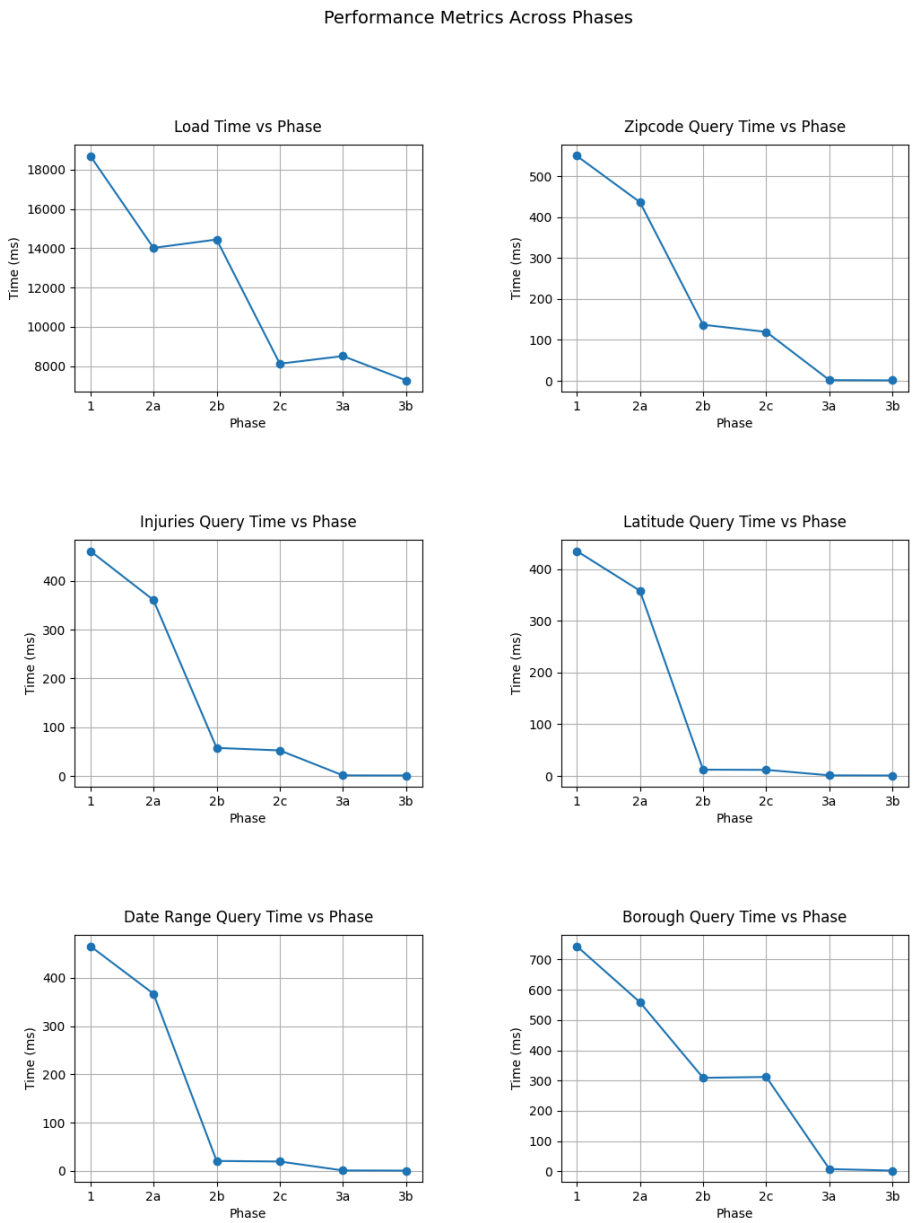
We also calculated the memory usage for all the phases and compared the data across phases.



Using OpenMP used the most amount of data, compared to phase 1 as well, but we were able to reduce it to some extent through chunking.

Conclusions

Throughout the phases, we set a benchmark for comparison and utilized different techniques such as parallelization, chunking, object-of-arrays design, and data compression (string to int) wherever logical. We explored various approaches and found some to be useful more so than others. To represent our conclusions, we have put it all in a graph.



The best version of our application came by using a combination of these optimization techniques represented in 3b.

The techniques used can be listed as follows -

1. OpenMP Parallelization over parsing and queries
2. Chunking data during loading and parsing
3. Object of Arrays design
4. String to int optimizations
5. Map for limited attribute value sets

Through a combination of these five, we were able to achieve drastically quicker responses to loading the data and querying over different attributes.

We can also optimize this even further by choosing primitive data types like uint8 where possible instead of the current version, to speed up the querying process even more.

Team Contributions

The team collectively worked on getting the best out of the dataset through our application by improving the code phase-to-phase. Each member had their own unique footprint on this.

Ankit Ojha

- Implemented the initial working code for phase 1.
- Explored and implemented chunking in phase 2.

Chayan Shah

- Implemented the parallel processing for phase 2 using OpenMP.
- Explored primitive data type usage and pointer references in phase 3.

Yash Kumar

- Refactored all phases of code with a cleaner object-oriented design.
- Designed different query types to challenge different aspects of each data type.
- Designed and implemented the date integer conversion and comparison.

Sarvesh Borkar

- Implemented the initial working code for phase 3 using OoA.
- Explored and implemented borough encoding optimization in phase 3.

Moreover, each team member contributed to discussions over potential optimizations as well as documented their work for the report and presentation, simultaneously testing their work and others' through a common Github repository.

Citations

1. https://stackoverflow.com/questions/4763455/data-type-size-impact-on-performance
2. https://medium.com/%40r.siddhesh96/small-object-optimization-soo-a-clever-c-trick-you-should-know-32edcb1035de
3. https://codeinterstellar.medium.com/optimizing-memory-usage-in-c-a-deep-dive-into-data-alignment-and-padding-fd0ea3999aed
4. https://johnnysswlab.com/make-your-programs-run-faster-by-better-using-the-data-cache/
5. https://stackoverflow.com/questions/43195927/openmp-file-reading-in-c