CMPE 275: Mini 2 Report

Team NexaBytes

Team Members: Ankit Ojha, Chayan Shah, Sarvesh Borkar, Yash Kumar

Introduction

In Mini 1, our primary focus was optimizing the loading & querying processes of the NYC Motor Vehicle Collision dataset through parallelization and memory optimization. We designed and benchmarked various strategies involving Array of Objects (AoO) and Object of Arrays (OoA) layouts and applied OpenMP to accelerate performance. The goal was to fully utilize the computational potential of a single node, maximizing CPU and memory efficiency for large-scale data analysis.

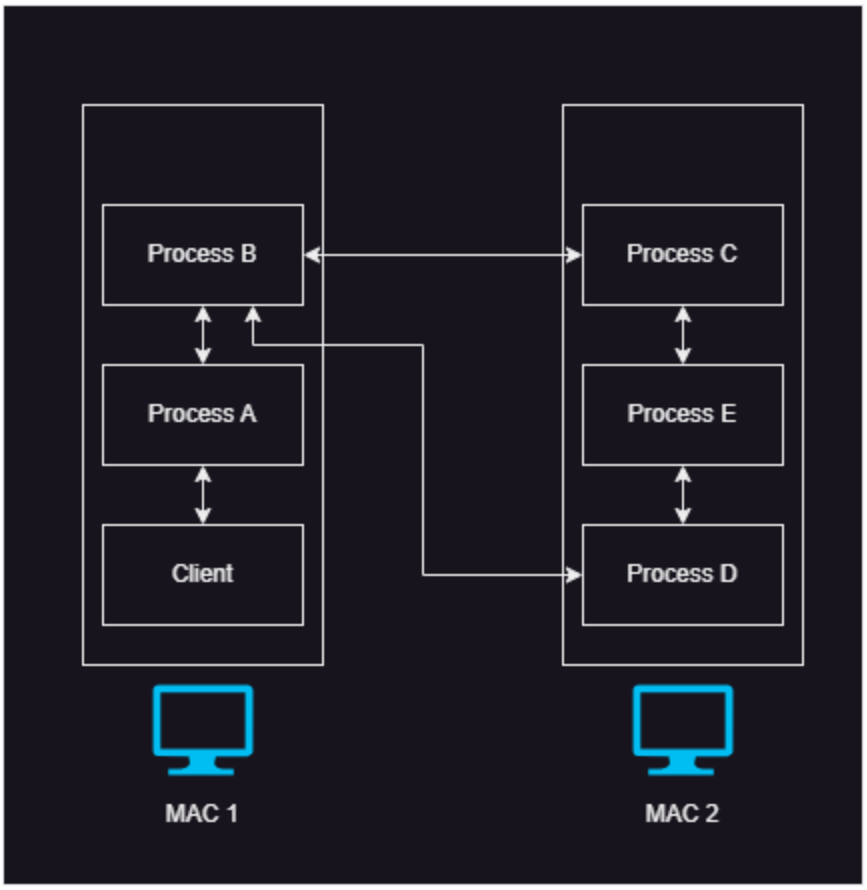
In Mini 2, we build upon those foundations by shifting from intra-process to inter-process communication and coordination across multiple machines. Our objective was to explore how distributed systems can scale performance by dividing computational work across processes, each responsible for a distinct subset of the dataset, using the Scatter-Gather pattern. Given the nature of the data, we partitioned it by boroughs, assigning each to a process, and designed a structured overlay network for routing and coordinating between processes.

To facilitate this coordination efficiently, our system integrates two primary components: gRPC and Shared Memory. gRPC enables remote procedure calls across machines, while shared memory allows co-located processes to cache query results without the overhead of serialization or locking. Client queries are routed to a central portal node, which forwards them through the overlay. Each node performs local computation, aggregates downstream results, and returns the final response. This architecture not only supports scalable, distributed querying but also significantly reduces response time for repeated queries through caching.

Overall Design & Implementation

Our distributed crash data analysis system consists of five processes (A–E) deployed across two Mac computers. The setup follows a structured overlay network - Mac 1 hosts Process A (the portal) and Process B (the first chunk of data) and Mac 2 hosts Process C (the second chunk of data), Process D (the third chunk of data), and Process E (the fourth chunk of data). The processes form a Scatter-Gather topology, where queries flow from the client portal to borough-specific processes and results are recursively aggregated back.

The connections are shown in the diagram below. The overlay is designed to allow efficient parallel query handling, minimizing redundant data transfer and enabling partial aggregation as queries propagate.

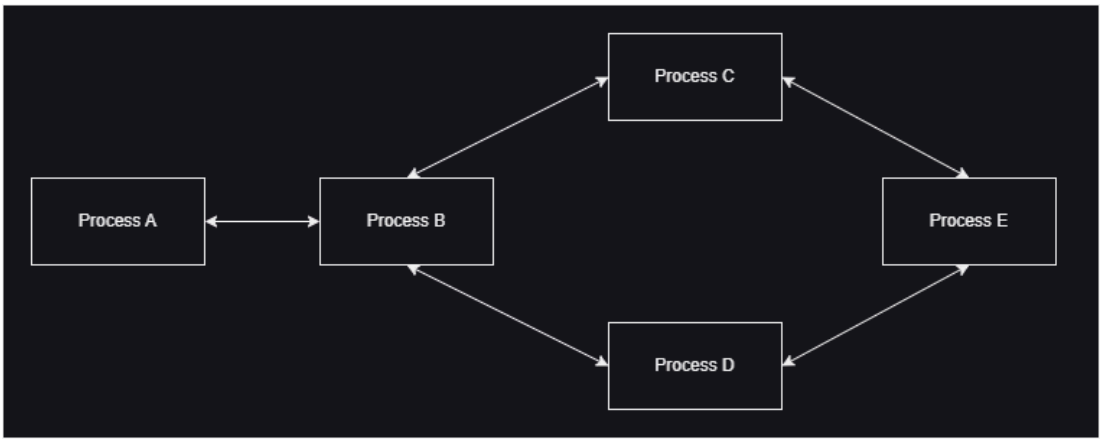


Each process has a specific role and connections it needs to manage.

Process	Machine	Data Handled	Forwards To	Receives From	Role
A	Mac 1	N/A	B	Client	The entry point for Clients
B	Mac 1	Chunk1	C, D	A	Relay node & aggregator for A
C	Mac 2	Chunk2	E	B	Mid node forwards to E
D	Mac 2	Chunk3	E	B	Mid node forwards to E
E	Mac 2	Chunk4	N/A	C, D	The leaf node returns data to C & D

Each process manages its own data subset while propagating (scattering) or collecting (gathering) results from its child processes to forward the requests to it.

The overlay of processes looks like a diamond.



After designing the flow, we designed a compact CrashData struct, similar to Mini 1, to represent each record of the data, consisting of various primitive data types for the columns of the NYC MVC dataset.

Since the dataset is limited by the number of columns to query on, we can utilize caching and avoid redundant computation for repeated queries on the same machine. For this, we implemented a transactionless shared memory cache using POSIX APIs with components like SharedMemoryManager (manages memory segments), SharedMemorySegment (region of shared memory), and SharedCache (high-level cache built on top of shared memory for storing query results). Moreover, for cross-machine messaging, we used three core gRPCs - QueryData (standard request-response), SendData (one-way dispatch), and StreamData (streaming large responses).

For our use case, we chose to compare our changes over two queries:

1. Query Crashes by Time: get crashes that occurred at a particular hour
2. Query Crashes with Fatalities: get crashes with at least N fatalities

At each process, queries are processed with their local subset of data and returned according to the overlay back to the client.

Moreover, for quicker responses, we implemented a caching mechanism that stores query results in shared memory in a key-value store with time-to-live (TTL) capabilities. Keys are generated based on the query string & parameters, while the values are the serialized query results. Cache entries automatically expire after a configurable TTL period.

Challenges & Fixes

Throughout the development process, we encountered several challenges with the code that required fixes to make it all work.

Shared Memory Limitations for macOS

Challenge: Due to stricter limits on shared memory usage on macOS, our initial attempts failed with “Invalid argument” errors when trying to allocate shared memory using shm_open and map frequently.
Solution: We designed and implemented a fallback mechanism that automatically switches to regular heap-allocated memory when shared memory is unavailable, ensuring functional caching across co-located processes, albeit at a reduced performance.

Data Serialization for Custom Types

Challenge: Our custom data type CrashData struct is transmitted between processes via gRPC, which requires careful serialization for transporting it across a network.
Solution: We implemented a string-based serialization of CrashData objects to pass in gRPC messages, along with robust parsing logic on the receiving end to reconstruct the original data in the appropriate data structure. This enabled seamless communication between machines.

Cross-Machine Communication

Challenge: gRPC communication across different machines led to some complexities involving IP addresses, port conflicts, and consistent overlay configuration.
Solution: We introduced a JSON-based configuration system that defines the overlay topology, IP addresses, and ports. This abstraction made it easier to modify deployment without changing the application code.

Query Forwarding & Result Aggregation

Challenge: With a multi-hop overlay network, we needed to ensure that queries flowed correctly through the hierarchy and results were aggregated without duplication or loss.
Solution: Each process was assigned a defined role in the forwarding hierarchy. Queries are passed downstream according to the overlay, and results are aggregated on the return path, ensuring accuracy and avoiding redundant forwarding.

Cache Size Management

Challenge: Large query results occasionally exceeded the capacity of our shared memory segments, resulting in incomplete caching or failures.
Solution: We implemented logic to estimate cache size before insertion and gracefully fall back to in-memory caching when limits are exceeded. We also added detailed logging to monitor cache health and diagnose bottlenecks.

Data Duplication in Overlay Network

Challenge: Our overlay network design created redundant paths to Process E (through both C and D), causing duplicate data in query results when the full network was traversed.
Solution: We implemented deduplication logic when aggregating results, ensuring each process's data appeared only once in the final response to clients.

Results & Metrics

We experimented with this code over 4 different settings

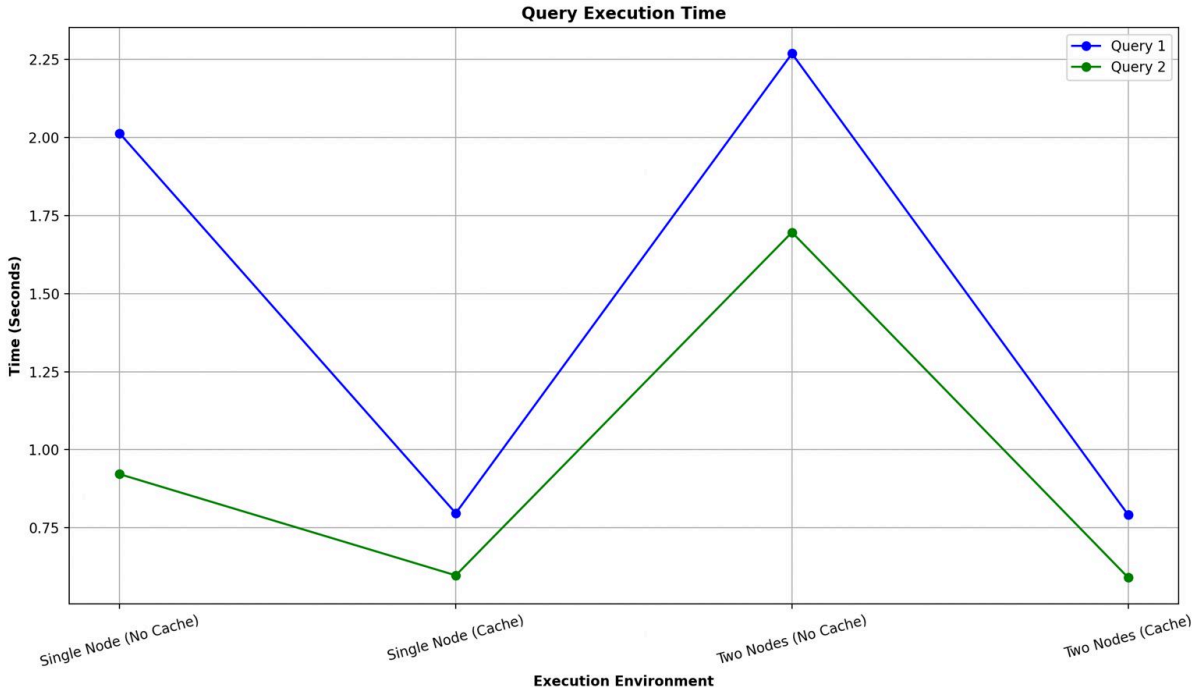
- Single node, no caching
- Single node with caching
- Two nodes, no caching
- Two nodes with caching

The first configuration gives us a benchmark like mini 1 to see our performance using multiple systems. We compare that with the caching mechanism as well as our multi-system, multi-process setup to observe if there are any potential improvements observed.

After performing experiments over multiple runs and taking an average of each run, we observed the following query times:

Query	Single Node, No Cache	Single Node w/ Cache	Two Nodes, No Cache	Two Nodes w/ Cache
Query Crashes by Time	2.103	0.797	2.268	0.772
Query Crashes with Fatalities	0.922	0.598	1.696	0.591

The metrics are visually summarized in the graph below:



From the graph, we can observe the following:

1. Caching cuts down query time massively. For Query 1, it drops from ~2.0s to ~0.8s on a single node (~60% faster).
2. Without caching, a two-node setup is slower than a single-node setup due to network overhead. For Query 1, we observed a 7% increase in runtime.
3. Query 2 always runs faster than Query 1, showing some queries are naturally lighter due to the nature of the data that we are searching.
4. Caching boosts performance in both setups, with two-node cached queries also dropping to ~0.8s. It narrows the performance gap between different query types, making things more consistent.

We have also attached some screenshots to display the output of different processes:

```
Connecting to E at 169.254.106.40:50055
Process D started. Press Enter to exit.
Server started at 169.254.106.40:50054
Process D received query: get_by_time with parameters: 14:00
```

```
Process E started. Press Enter to exit.
Server started at 169.254.106.40:50055
Process E received query: get_by_time with parameters: 14:00
Process E received query: get_by_time with parameters: 14:00
```

```
Connecting to E at 169.254.106.40:50055
Process C started. Press Enter to exit.
Server started at 169.254.106.40:50053
Process C received query: get_by_time with parameters: 14:00
```

```
Connecting to C at 169.254.106.40:50053
Connecting to D at 169.254.106.40:50054
Process B started. Press Enter to exit.
Server started at 169.254.181.125:50052
Process B received query: get_by_time with parameters: 14:00
```

```
Connecting to B at 169.254.181.125:50052
Process A started. Press Enter to exit.
Server started at 169.254.181.125:50051
Received query: get_by_time with parameters: 14:00
```


```
{
  "query_id": "1744358641213",
  "success": true,
  "message": "Combined results (37652 total entries)",
  "execution_time": "1.414 seconds",
  "result_count": 37652,
  "results": [
    {
      "key": "crash_537505",
      "value": "Date: 10/28/2018, Time: 4:45, Borough: , Killed: 0",
      "type": "crash_data",
      "crash_time": "4:45"
    },
    {
      "key": "crash_537279",
      "value": "Date: 11/04/2018, Time: 4:15, Borough: , Killed: 0",
      "type": "crash_data",
      "crash_time": "4:15"
    },
    {
      "key": "crash_537185",
      "value": "Date: 10/20/2018, Time: 4:20, Borough: QUEENS, Killed: 0",
      "type": "crash_data",
      "crash_time": "4:20"
    },
    {
      "key": "crash_537131",
      "value": "Date: 11/11/2018, Time: 4:42, Borough: BROOKLYN, Killed: 0",
      "type": "crash_data",
      "crash_time": "4:42"
    },
```

Future Scope

While our current system achieves the core goals of distributed querying, caching, and overlay coordination, several improvements could further enhance its robustness and performance:

Smarter Shared Memory Management

Although our fallback mechanism ensures functional caching on macOS, the current setup is rigid. We could improve this through dynamically sized shared memory segments based on the availability of system resources. A more efficient cache size estimation as well as a more sophisticated memory management scheme would also be useful for the same.

Fault Tolerance Mechanisms

Currently, if a process fails or disconnects, we need to restart it manually. A more resilient system, like the ones in production, is capable of handling this. By gracefully handling node failures or implementing retries with backoff, we can counter this failure through classic solutions in distributed systems.

Adaptive Overlay Network

In its current state, we are statically defining the overlay network. Although it is effective for our use case, this becomes tedious to manage as increasing scale and resources increase. By enabling some form of dynamic updates to the network, which will automatically adjust it, we can ensure that the network is capable of handling any updates to itself.

Query Efficiency Enhancements

Some queries, especially ones that span all nodes, could be optimized further through better filtering at source nodes, ultimately reducing data passed upstream. We could also explore support for asynchronous query forwarding, improving the responsiveness.

Conclusion

Through mini 2, we evolved our approach and code from a single-node setup in mini 1 to a fully distributed, multi-process architecture. Simulating real-world solutions, we designed and implemented an overlay network with data partitioning over processes and inter-process communication and coordination. The integration of gRPC for cross-machine communication and POSIX shared memory for intra-machine caching enabled our system to efficiently propagate queries and return results with minimal redundancy and delay.

Our experimental results validated the architectural decisions made. Caching dramatically improved performance across both single-node and two-node setups, while the distributed structure allowed for more scalable query handling. At the same time, challenges like shared memory limitations, serialization, and result deduplication highlighted the complexity of distributed systems and the value of robust design decisions.

Overall, this mini-project gave us practical exposure to system-level programming, performance tuning, and designing communication strategies between cooperating services. With further enhancements around fault tolerance and dynamic overlays, this system could evolve into a solid foundation for scalable data analytics in a distributed environment.

References

1. gRPC Documentation. (n.d.). Retrieved from https://grpc.io/docs/
2. Protocol Buffers Documentation. (n.d.). Retrieved from https://developers.google.com/protocol-buffers
3. POSIX Shared Memory. (n.d.). Retrieved from https://man7.org/linux/man-pages/man7/shm_overview.7.html
4. Tanenbaum, A. S., & Van Steen, M. (2007). Distributed systems: principles and paradigms. Pearson Prentice Hall.
5. NYC Open Data. (n.d.). Motor Vehicle Collisions - Crashes. Retrieved from https://data.cityofnewyork.us/Public-Safety/Motor-Vehicle-Collisions-Crashes/h9gi-nx95
6. Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2011). Distributed Systems: Concepts and Design (5th ed.). Addison-Wesley.
7. Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7), 558-565.
8. Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1), 107-113.
9. ChatGPT