CMPE 275 Grand Challenge-1 Report

Build a distributed system for large scale data transfer and retrieval by incorporating toplogies

Akhil, 016592238

priyatha, 016421418

pranav,016587064

lokesh

gowtham

May 2023

Table of Contents

1. Introduction

The main objective of the Grand challenge is to create a distributed system that uses multiple servers for data injection and querying .The project endeavors to develop a cohesive system

capable of efficiently managing vast streams of data for input and output, dynamically redistributing data, implementing caching, and implementing various optimizations.

The main goal of the project:

The project's primary aims revolve around several key objectives. Firstly, it involves providing multiple servers to each team, enabling them to engage in distributed processing and achieve enhanced scalability. Collaboration and integration among teams are actively encouraged to create a unified system that collectively attains the project's goals. An essential aspect is efficiently managing the substantial inflow and outflow of data, with multiple clients connected to multiple servers. The system should dynamically redistribute data, optimizing storage, redundancy, and resource utilization. Emphasizing strategies like caching frequently accessed data and optimizing data movement, such as relocating it closer to areas of high demand, contributes to improved performance and reduced latency. The system is designed to be adaptive, seamlessly accommodating the addition of new servers and the removal of non-responsive ones. Furthermore, the project highlights the implementation of use cases where each team's servers act as self-contained systems aligned with the overall project objectives. These mini-systems can be integrated into a larger framework, enabling collaboration and resource sharing. Effective redistribution of data to other teams' servers promotes efficient collaboration and maximizes resource utilization.

2. System Design and Architecture

2.1 Introduction

Our design incorporates a central server that acts as an intermediary between clients and the data-holding servers. To facilitate communication between the clients and servers, we utilized the grpc framework. For interconnecting the servers and facilitating efficient querying of requests, we implemented a ring topology. Additionally, we incorporated the hub and spoke method for injecting data among the servers based on predefined logic. We made improvements to the data processing and retrieval processes, ensuring enhanced efficiency. Furthermore, we optimized the query processing to enable swift retrieval of data from the servers.

2.2 gRPC

The decision to use gRPC for client and server communication was based on several advantages it offers.

Firstly, gRPC is a high-performance, open-source framework developed by Google. It utilizes the efficient and lightweight protocol buffers (protobuf) for message serialization, making it more efficient in terms of network bandwidth and processing speed compared to other alternatives like REST or SOAP.

Secondly, gRPC supports bi-directional streaming and asynchronous communication, allowing for efficient and real-time interaction between clients and servers. This is particularly useful in distributed systems where multiple clients need to send and receive data concurrently.

Thirdly, gRPC provides language-agnostic support, making it easier to build and integrate services using different programming languages. This flexibility allows for more diverse development teams and simplifies interoperability between different components of the distributed system.

Additionally, gRPC offers built-in support for features like authentication, load balancing, and error handling, which are essential in distributed systems. These features help improve the security, reliability, and scalability of the system.

Overall, gRPC was chosen for its efficiency, flexibility, and comprehensive feature set, making it a suitable choice for the client-server communication needs of the distributed system.

2.2 System Architecture

the data uploading process involves the client streaming the data to the main server. The main server then utilizes a predefined logic, along with a hub and spoke method, to split and distribute the data among the sub-servers.

When it comes to querying the data, the client sends the request to the main server. The main server, in turn, forwards the request to the appropriate sub-servers. The sub-servers process the request and retrieve the corresponding results, which are then sent back to the main server. Finally, the main server delivers the retrieved data to the client.

2.2 Data Injection

In our system, the client streams data to the main server in chunks. The main server receives the data and partitions it into sub-CSV files, organizing the data based on the year. Each year's data is stored in its respective CSV file.

Following the partitioning process, the data is streamed from the main server to the sub-servers based on the year and server assignments. The sub-servers receive the incoming data and convert it into CSV files for each day. The files are named using the format "yyyy-mm-dd," representing the specific day of the data.

Overall, this process ensures that the data is efficiently organized and distributed among sub-servers, with each server responsible for maintaining CSV files corresponding to specific days and years.

2.3 Data Query

The client initiates a request by sending a string in the format "yyyy-mm-dd:yyyy-mm-dd" to specify the desired start and end dates. This request

is then forwarded to the main server. The main server, in turn, distributes the request to the appropriate sub-servers.

The sub-servers search for CSV files with names in the "yyyy-mm-dd" format that fall within the specified date range. If the files are present on the current sub-server, they are retrieved. However, if the files are not found, the search continues on the next sub-server until all the relevant files are obtained. The retrieved files are then sent back to the main server.

Finally, the main server consolidates the results and sends them back to the client. This process ensures that the client's request is effectively handled, with the sub-servers collaborating to retrieve the necessary CSV files based on the specified date range.

3. Implementation

3.1 System implementation

3.1.1 grpc

We implemented 2 grpc services in our project .they are mentioned in the proto file:

```
syntax = "proto3";

package myapp;

service RouteService {
    rpc upload(stream Route) returns (Route) {}
    rpc query(QueryRequest) returns (Route) {}
}

message Route {
    string id = 1;
    bytes payload = 2;
}

message QueryRequest {
    string issue_date = 1;
}
```

rpc upload(stream Route) returns(Route) {}:
this is used to upload the data to the servers and then to the sub servers.
rpc query(QueryRequest) returns (Route) {}:
this service is used to retrive the record based on the client request

3.1.3 ETL : Upload function

The data is sent to the server using the upload function using streaming of chunks of data

```
stub = route_pb2_grpc.RouteServiceStub(channel)
print("sending the CSV file to sub server")
def generate_routes():
    with open(f"./csv/{int(i)}.csv", "rb") as csv_file:
        while True:
            data = csv_file.read(4096)  # Adjust the chunk size as needed
            if not data:
                break
            route = route_pb2.Route(payload=data)
            yield route

routes = generate_routes()
response = stub.upload(routes)
print("File uploaded successfully to the sub server1")
```

```
def upload(self, request_iterator, context):
    file = "./sub_server/client_file.csv"
    print("got the csv file")
    with open(file, "wb") as csv_file:
        for route in request_iterator:
            payload = route.payload
            csv_file.write(payload)
    df = pd.read_csv(file, low_memory=False)
    cols = df.columns

    # Create the 'csv_files' folder if it doesn't exist
    folder_name = "./csv_files_20"
    if not os.path.exists(folder_name):
        os.makedirs(folder_name)

    for i in set(df['Issue Date']):
        j = i.split("/")
        year = int(j[2])
        filename = f"{j[2]}-{j[0]}-{j[1]}.csv"
        file_path = os.path.join(folder_name, filename)
        print(file_path)
        df.loc[df['Issue Date'] == i].to_csv(file_path, index=False, columns=cols)
    return route_pb2.Route()
```

The files are stored for each day. The records for each day are stored as a csv file

3.1.3 ETL : Query function

```
filename = f"{self.csv_directory}/{current_date.strftime('%Y-%m-%d')}.csv"
if os.path.exists(filename):
    while (current_date<=end_date):
        print("hello")
        print(current_date.strftime('%Y-%m-%d'))
        filename = f"{self.csv_directory}/{current_date.strftime('%Y-%m-%d')}.csv"
        print(filename)
        if os.path.exists(filename):
            print("found")
            with open(filename, "rb") as file:
                payload = file.read()
                if not self.header_written:
                    results.append(payload)  # Include the header for the first file
                    self.header_written = True
                    print("found1")
                else:
                    results.append(payload[bytes(payload).index(b"\n")+1:])
        print("appended")
        current_date +=timedelta(days=1)
        print(current_date)
```

The query function searches for the csv file in the date range and returns it to the main server .if not found the request is sent to the next server until all the servers are reached or the files are retrived.

3.2 Test results

Test conditions: 5 servers and 1 main server are being setup in the local.

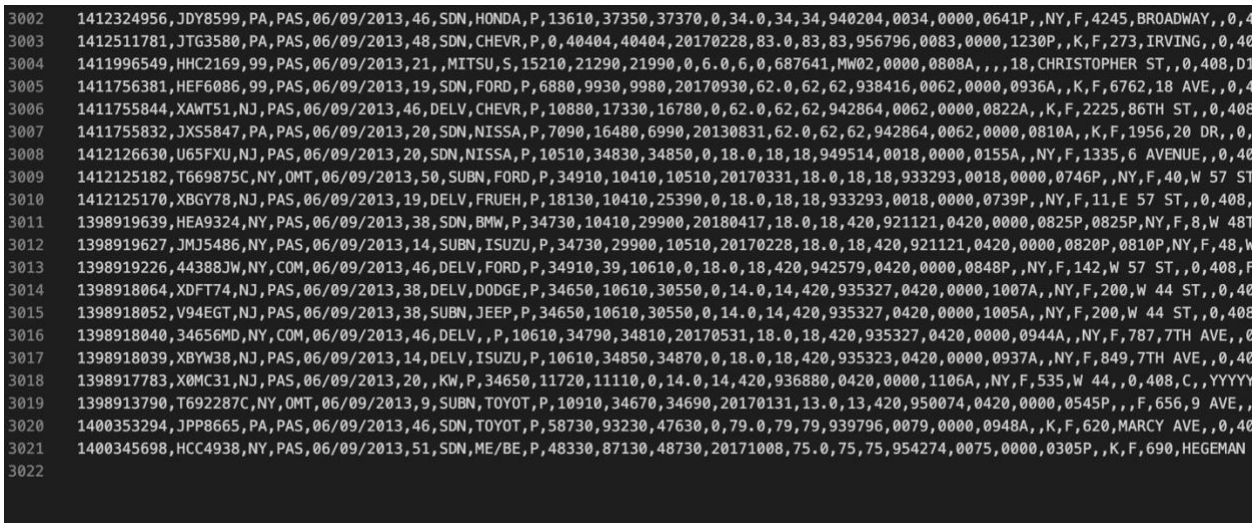
3.2.2 Upload files

The parking data is uploaded to the server from the client in chunks of 1024 bytes until the. File is completely transfered .

3.2.3 Query files

Sent a query. In form of yyyy-mm-dd:yyyy-mm-dd format

(1) **Query results**: After a query is submitted from the client, the main server collects the data from the sub servers and sends back to the client.



4. test cases

our group set up 5 computers (4 servers + 1 client). We successfully passed the test cases with another group listed below.

4.1 Load Data into System

We loaded the parking violation data into the servers using our predefined architecture

4.2 Query Data for One Day

We gave a query for a single day and we were able to get the results successfully

4.3 Query Data for multiple Days

We provided the range for multiple days and we were able to retrive the results

4.7 Query Data Not Present in Local Cluster

We provided the date that is not present and we got the empty result

5. Conclusion

We have developed a large scale distributed system that shares heavy data and enhances the quick querying of the record from the request from the clients

Appendix

Each team member contribution

Akhil, 016592238 :

- Client side coding
- ETL

priyatha, 016421418 :

-ETL

- upload function

pranav, 016587064 :

- Query function
- Server logic

gowtham, :

- Documentation
- Server logic

lokesh:

- Documentation
- Architecture design

Reference

https://grpc.io/

https://medium.com/codex/distributed-services-using-grpc-100743363c6b

https://understandingdistributed.systems/