Mini-2 report

Team member:
Thinh Bui thinh.bui@sjsu.edu
Aditya Rao aditya.rao@sjsu.edu

Introduction:

In this report, a comparison between gRPC and REST HTTP is presented, together with the instruction to build and run the benchmark. From the experiments, we found that the REST implementation has better performance than the gRPC implementation in our use case. The document is organized as follows. First, the experiment setup will be explained. Next, an experiment results and comparison will be presented. Furthermore, we also compared the implementation overhead between gRPC and REST. Finally, we conclude with the instruction to build and run the benchmark.

Experiment setup

To measure the performance of gRPC and REST, we built a service on top of query-engine from mini-1. The services are capable of returning all rows having the field's value matched with the request. To further the performance comparison, for each service, we built 2 API returning the same data but in different formats. While the search API in HTTP Service or GetRecord in gRPC service return an array of rows matching results, the search_string API and the GetRecordString RPC return a concatenated string of rows to the client. Below is the API definition of REST service and proto definition of the gRPC service.

```
Unset
HTTP API:
       GET /search?headerKey=[header-key]&headerValue=[header-value]

       header-key is the csv column name
       header-value is the value we want to match
```

```
       Response: a JSON object contains all the row data that matches the
       header-key and header-value.

HTTP API:
       GET /search_string?headerKey=[header-key]&headerValue=[header-value]

       header-key is the csv column name
       header-value is the value we want to match

       Response: a string contains all the row data that matches the header-key
       and header-value.
```

Unset

```
GRPC proto:
service ReaderService {
   rpc GetRecord(ReaderRequest) returns (ReaderResponse);
   rpc GetRecordString(ReaderRequest) returns (ReaderResponseString);
}

message ReaderRequest {
    string key = 1;
    string value = 2;
}

message ReaderResponse {
    repeated Record records = 1;
    int64 num_record = 2;
}

message ReaderResponseString {
    string data = 1;
}

message Record {
    repeated Field fields = 1;
}

message Field {
    string key = 1;
    string value = 2;
}
```

Regarding the server implementation, both are implemented by C++. While the gRPC server is implemented using a synchronous method and gRPC library, the HTTP server is implemented using Crow HTTP library. To run the benchmark, we used the wrk to generate workload for the HTTP server, and ghz to generate workload for the gRPC server. In this experiment, as our server can not handle concurrent requests well enough, we focus on the performance to process one request. Thus we will limit the benchmark tool to generate one request at a time.

Experiment results

This section presented the results between REST and gRPC. The table [1] below compare the performance results between /search API and GetRecord RPC for the same request header-key=”locations1” and header-value=”Jacobs”

	p50	p90	p99
GetRecord (gRPC)	721.79 ms	724.82 ms	1.11 s
/search (REST HTTP)	726 ms	768 ms	1137 ms

Table 1. Comparison between GetRecord and /search

From table 1, we found the gRPC service is faster than the REST service. However, the performance difference between them is not too much. To dig deeper into the performance, we tried to reduce the size of response from the server by introducing a flag called result_size. Result_size is a double value from 0 to 1, indicating the percentage of matched rows we returned to the client. Result_size equal 1 indicates that all the matched rows will be returned, while result_size equals 0.5 indicates that half of the matched rows will be returned.

		p50	p90	p99
Result_size = 0.5	GetRecord (gRPC)	716.84 ms	725.78 ms	1.12 s
	/search (REST HTTP)	703 ms	706 ms	1137 ms
Result_size = 0.25	GetRecord (gRPC)	702.99 ms	711.57 ms	1.08 s

	/search (REST HTTP)	699 ms	703 ms	1095 ms

Table 2. Comparison between GetRecord and /search, with reduced response's size

Table 2 recorded the performance of gRPC and REST when we reduced the response's size. From the results, we can easily see that the latency is decreased as we reduced the result_size. However, an interesting finding here is that the performance of REST is better than the gRPC. There is a significant gap between p90 of gRPC and REST when result_size was 0.5. To further confirm the performance gap between REST and gRPC, we tried to optimize the query-engine code by turning on the parallel processing using the openMP library. Since the round trip time is equal to the gRPC/REST processing time plus the query-engine, reducing the query-engine time can boost the difference between REST and gRPC.

	p50	p90	p99
GetRecord (gRPC)	149.41 ms	158.50 ms	253.71 ms
/search (REST HTTP)	141 ms	156 ms	204 ms
GetRecordString (gRPC)	138.50 ms	171.42 ms	329.67 ms
/search_string (REST HTTP)	127 ms	146 ms	233 ms

Table 3. Comparison between GetRecord and /search, with optimized query-engine

From table 3, we can see that there is a big gap between the p99 of GetRecord and /search. Furthermore, when benchmarking the GetRecordString gRPC and /search_string REST API, the /search_string REST API is much better than the gRPC API. It is suggested that there could be performance issues when serializing a big string in protobuf.

In conclusion, given our use case, the gRPC did not outperform the REST API. In some case, such as when returning result is a big string, the gRPC performance is worse than the REST API.

Instruction to build and run the benchmark

Setup and build the server

```
Unset
# Generate proto code

mkdir generated-src
./generate-proto.sh

# Build the project

mkdir build
cd build
cmake ..
make all
```

Run the benchmark

```
Unset
# GRPC benchmark
# Start the GRPC server

cd build
./reader-server

# Install the ghz tool from https://ghz.sh/
# Run the benchmark
./benchmark/grpc.sh
```

```
Unset
# HTTP benchmark
# Start the HTTP server

cd build
./http-server

# Install the wrk tool from https://github.com/wg/wrk
# Run the benchmark
./benchmark/http.sh
```

Issues and Implementation

Initially we were trying to build the c++ program using the g++ command. Multiple errors were encountered while importing the crow library which was the library used for running the server. To bypass this issue we tried to change the library and we tried Pistache, Boost & Drogon. We also decided to switch to CMake as it is recommended to run bigger projects which have a lot of external dependencies. Now we were running into multiple OS library linking issues and gcc version mismatch issues which were mostly caused by the Windows OS. Eventually we fixed all the issues and were able to run the program by linking the correct libraries and versions. There was no direct support to run ‘wrk’ for benchmarking performance in Windows. For this we tried running the entire project in WSL(Windows Subsystem For Linux). We were able to compile the project and run the benchmark very smoothly.

Reference

[1] ghz https://ghz.sh/
[2] wrk https://github.com/wg/wrk