CS60002: Distributed Systems

Assignment 2: Implementing a Scalable Database with Sharding

Date: Feb 06th, 2024

Target Deadline (For sharing the git repo): March 15, 2024

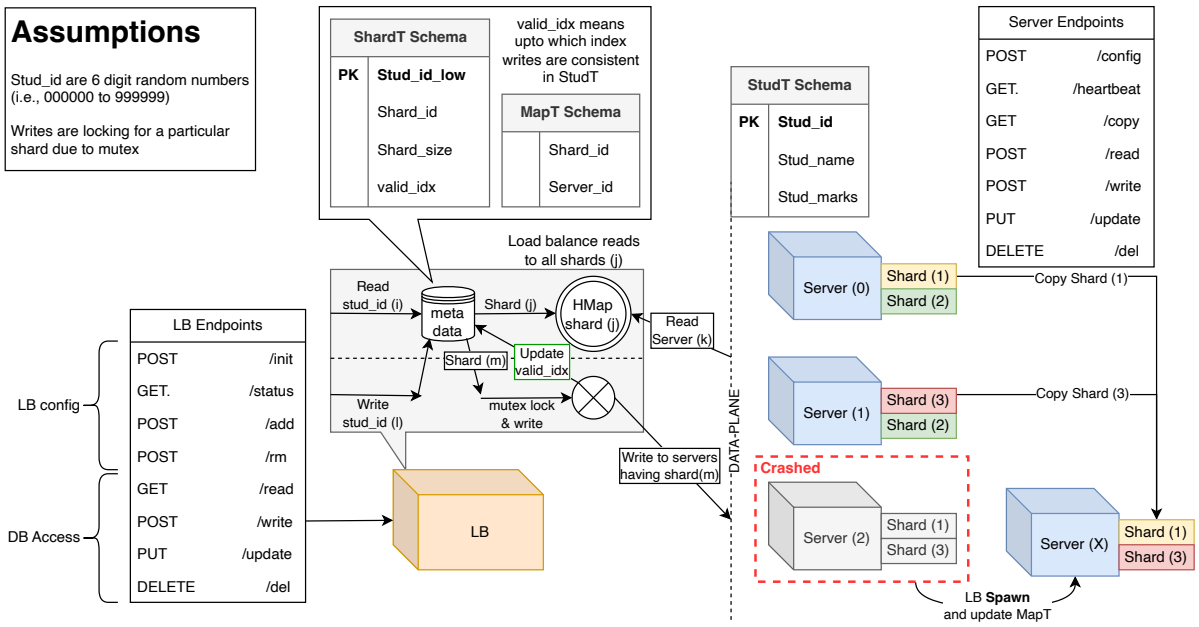

Fig. 1: System Diagram of a Sharded Database

I. Overview

In this assignment, you have to implement a sharded database that stores only one table *StudT* in multiple shards distributed across several server containers. This is an incremental project so that you can reuse the codebase from the first assignment. A system diagram of the sharded database is shown in Fig. 1. Here, shards are subparts of the database that only manage a limited number of entries (i.e., *shard_size* as shown in the diagram). Shards can be replicated across multiple server containers to enable parallel read capabilities. For this assignment, we assume that write requests are blocking for a particular shard. Thus, if two write requests are scheduled simultaneously on shard (i), one of them will wait for the other to complete. However, Parallel writing to different shards, for instance, shard (i) and shard (j), is possible. The system's current design provides scaling in two ways: (i) Read speed with more shard replicas and (ii) Database size with more shards and servers.

A. Coding Environment

- **OS:** Ubuntu 20.04 LTS or above
- **Docker:** Version 20.10.23 or above
- **Languages:** C++, Python (preferable), Java, or any other language of your choice

B. Submission Details

- Write clean and well-documented code.
- Add `README` file and mention the design choices, assumptions, testing, and performance analysis.
- Add `Makefile` to deploy and run your code.

Your Code should be version-controlled using Git, and the GitHub repository link must be shared before the deadline. Please note that the contribution of each group member is essential to learn from these assignments. Thus, we will inspect the commit logs to award marks to individual group members. An example submission from previous course session can be found at https://github.com/prasenjit52282/shardQ for your reference.

II. Task1: Server

The server containers handles shards of the ***StudT*** **(Stud_id: Number, Stud_name: String, Stud_marks:Number)** table. Each server container can manage a different set of shard replicas of the distributed database, as shown in Fig. 1. The endpoints to handle requests to a specific server are described as follows:

1) **Endpoint (/config, method=POST):** This endpoint initializes the shard tables in the server database after the container is loaded. The shards are configured according to the request payload. An example request-response pair is shown below.

```
Payload Json= {
    "schema":{"columns":["Stud_id","Stud_name","Stud_marks"],
              "dtypes":["Number","String","String"]}
    "shards":["sh1","sh2"]
}
```

```
Response Json ={
    "message" : "Server0:sh1, Server0:sh2 configured",
    "status" : "success"
},
Response Code = 200
```

2) **Endpoint (/heartbeat, method=GET):** This endpoint sends heartbeat responses upon request. The heartbeat endpoint is further used to identify failures in the set of server containers maintained in the distributed database. Therefore, you could send an empty response with a valid response code.

```
Response [EMPTY],
Response Code = 200
```

3) **Endpoint (/copy, method=GET):** This endpoint returns all data entries corresponding to one shard table in the server container. Copy endpoint is further used to populate shard tables from replicas in case a particular server container fails, as shown in Fig. 1. An example request-response pair is shown below.

```
Payload Json= {
    "shards":["sh1","sh2"]
}
Response Json ={
    "sh1" : [{"Stud_id":1232,"Stud_name":ABC,"Stud_marks":25},
             {"Stud_id":1234,"Stud_name":DEF,"Stud_marks":28},
             ....],
    "sh2" : [{"Stud_id":2255,"Stud_name":GHI,"Stud_marks":27},
             {"Stud_id":2535,"Stud_name":JKL,"Stud_marks":23},
             ....],
    "status" : "success"
},
Response Code = 200
```

4) **Endpoint (/read, method=POST):** This endpoint reads data entries from a shard in a particular server container. The endpoint expects a range of *Stud_ids* (low, high) and *Shard_id* that has to be read from the server container.

```
Payload Json= {
    "shard":"sh2",
    "Stud_id": {"low":2235, "high":2555}
}
Response Json ={
    "data" : [{"Stud_id":2535,"Stud_name":JKL,"Stud_marks":23},
              {"Stud_id":2536,"Stud_name":MNO,"Stud_marks":22},
              .....,
              {"Stud_id":2554,"Stud_name":XYZ,"Stud_marks":25},
              {"Stud_id":2255,"Stud_name":GHI,"Stud_marks":27}],
    "status" : "success"
},
Response Code = 200
```

5) **Endpoint (/write, method=POST):** This endpoint writes data entries in a shard in a particular server container. The endpoint expects multiple entries to be written in the server container along with *Shard_id* and the current index for the shard. An example request-response pair is shown below.

```
Payload Json= {
    "shard":"sh2",
    "curr_idx": 507
    "data": [{"Stud_id":2255,"Stud_name":GHI,"Stud_marks":27}, ...] /* 5 entries */
}
Response Json ={
    "message": "Data entries added",
    "current_idx": 512, /* 5 entries added */
    "status" : "success"
},
Response Code = 200
```

6) **Endpoint (/update, method=PUT):** This endpoint updates a particular data entry in a shard in a particular server container. The endpoint expects only one entry to be updated in the server container along with *Shard_id*. An example request-response pair is shown below.

```
Payload Json= {
    "shard":"sh2",
    "Stud_id":2255,
    "data": {"Stud_id":2255,"Stud_name":GHI,"Stud_marks":28} /* see marks got updated */
}
Response Json ={
    "message": "Data entry for Stud_id:2255 updated",
    "status" : "success"
},
Response Code = 200
```

7) **Endpoint (/del, method=DELETE):** This endpoint deletes a particular data entry (based on *Stud_id*) in a shard in a particular server container. The endpoint expects only one entry to be deleted in the server container along with *Shard_id*. An example request-response pair is shown below.

```
Payload Json= {
    "shard":"sh1",
    "Stud_id":2255
}
Response Json ={
    "message": "Data entry with Stud_id:2255 removed",
    "status" : "success"
},
Response Code = 200
```

Finally, write a `Dockerfile` to containerize the server as an image and make it deployable for the subsequent tasks. Note that two containers can communicate via hostnames in a docker network. Docker provides a built-in DNS service that allows containers to resolve hostnames to the correct IP addresses within the same Docker network.

III. Task2: Improve The Load Balancer

You have implemented the load balancer with consistent hashing in the previous assignment. All the previous assumptions and implementation details still hold. However, you must modify and integrate a few features to the load balancer to make it work for assignment 2. In the system diagram shown in Fig. 1, you can observe that the load balancer manages *Stud_id* $\rightarrow$ *Shard_id* $\rightarrow$ *Server_id* mapping with two data tables in the **metadata**. The table schemas are as follows:

1) ***ShardT* (Stud_id_low: Number, Shard_id: Number, Shard_size:Number, valid_idx:Number)**
2) ***MapT* (Shard_id: Number, Server_id: Number)**

A. Assumptions

To connect the distributed database assignment with the earlier load balancer assignment, you should realize that you need to maintain consistent hashmaps for each of the shards. The consistent hashmaps can be identified with the *Shard_id*, and the hashmap entries will be populated by the replicas of the shard.

- There are only 4 (sh1, sh2, sh3, sh4) shards in the database.
- Each shard has 3 replicas across the servers.
- There are 6 servers having shards in configuration:

```
{
    "N":6
    "schema":{"columns":["Stud_id","Stud_name","Stud_marks"],
              "dtypes":["Number","String","String"]}
    "shards":[{"Stud_id_low":0, "Shard_id": "sh1", "Shard_size":4096},
              {"Stud_id_low":4096, "Shard_id": "sh2", "Shard_size":4096},
              {"Stud_id_low":8192, "Shard_id": "sh3", "Shard_size":4096},
              {"Stud_id_low":12288, "Shard_id": "sh4", "Shard_size":4096}]
    "servers":{"Server0":["sh1","sh2"],
               "Server1":["sh3","sh4"],
               "Server3":["sh1","sh3"],
               "Server4":["sh4","sh2"],
               "Server5":["sh1","sh4"],
               "Server6":["sh3","sh2"]}
}
```

- Thus consistent hashmap for shard sh1 will contain entries for servers {Server0:sh1, Server3:sh1, Server5:sh1}. Similarly each shard would have corresponding hashmap. Other constants for hashmap are as follows:
  1) Total number of slots in the consistent hash map ($\#slots$) = 512
  2) Number of virtual servers for each server container (K) = $\log(512) = 9$
  3) Function for request mapping H(i), & virtual server mapping Phi(i, j) is what you found works best for your load balancer implementation.

B. Endpoints

The read requests to a particular shard (i) will be load balanced with the consistent hash among all shard (i) replicas across all the server containers. Therefore, parallel read requests can be serviced. In case of write requests, the load balancer ensures consistent write to all the appropriate shard replicas with a mutex lock. Hence, writes are blocking for a particular shard and can only be parallelized across different shard instances (e.g., not replicas). The endpoints of the load balancer are as follows.

1) **Endpoint (/init, method=POST):** This endpoint initializes the distributed database across different shards and replicas in the server containers. As administrators, we should provide the configuration of shards and their placements in the set of server containers (place randomly if not specified). An example request-response pair is shown below.

```
Payload Json= {
    "N":3
    "schema":{"columns":["Stud_id","Stud_name","Stud_marks"],
              "dtypes":["Number","String","String"]}
    "shards":[{"Stud_id_low":0, "Shard_id": "sh1", "Shard_size":4096},
              {"Stud_id_low":4096, "Shard_id": "sh2", "Shard_size":4096},
              {"Stud_id_low":8192, "Shard_id": "sh3", "Shard_size":4096},]
    "servers":{"Server0":["sh1","sh2"],
               "Server1":["sh2","sh3"],
```

```
                "Server2":["sh1","sh3"]}
}
Response Json ={
    "message" : "Configured Database",
    "status" : "success"
},
Response Code = 200
```

2) **Endpoint (/status, method=GET):** This endpoint sends the database configurations upon request. The configuration is set via the */init* endpoint; thus, a valid response according to the above */init* example is shown below.

```
Response Json= {
    "N":3
    "schema":{"columns":["Stud_id","Stud_name","Stud_marks"],
              "dtypes":["Number","String","String"]}
    "shards":[{"Stud_id_low":0, "Shard_id": "sh1", "Shard_size":4096},
              {"Stud_id_low":4096, "Shard_id": "sh2", "Shard_size":4096},
              {"Stud_id_low":8192, "Shard_id": "sh3", "Shard_size":4096},]
    "servers":{"Server0":["sh1","sh2"],
               "Server1":["sh2","sh3"],
               "Server2":["sh1","sh3"]}
}
Response Code = 200
```

3) **Endpoint (/add, method=POST):** This endpoint adds new server instances in the load balancer to scale up with increasing client numbers in the system. The endpoint expects a JSON payload that mentions the number of new instances, their server names, and the shard placements. An example request and response is below.

```
Payload Json= {
    "n" : 2,
    new_shards:[{"Stud_id_low":12288, "Shard_id": "sh5", "Shard_size":4096}]
    "servers" : {"Server4":["sh3","sh5"], /* new shards must be defined */
                 "Server[5]":["sh2","sh5"]}
}
Response Json ={
    "N":5,
    "message" : "Add Server:4 and Server:58127", /* server id is randomly set in case of
    Server[5] */
    "status" : "successful"
},
Response Code = 200
```

Perform simple sanity checks on the request payload and ensure that servers mentioned in the Payload are less than or equal to newly added instances unless through an error. Server IDs are preferably set randomly to make the consistent hashing algorithm work efficiently. Therefore, you can also define a template for servernames (shown above) where ids are randomly set by the load balancer.

```
Payload Json= {
    "n" : 3,    /* wrong input n > len(servers) */
    new_shards:[{"Stud_id_low":12288, "Shard_id": "sh5", "Shard_size":4096}]
    "servers" : {"Server4":["sh3","sh5"],
                 "Server[5]":["sh2","sh5"]}
}
Response Json ={
    "message" : "<Error> Number of new servers (n) is greater than newly added instances",
    "status" : "failure"
},
Response Code = 400
```

4) **Endpoint (/rm, method=DELETE):** This endpoint removes server instances in the load balancer to scale down with decreasing client or system maintenance. The endpoint expects a JSON payload that mentions the number of instances to be removed and their preferred server names in a list. An example request and response is below.

```
Payload Json= {
    "n" : 2,
    "servers" : ["Server4"]
}
Response Json ={
    "message" : {
        "N" : 3,
        "servers" : ["Server1", "Server4"] /*See "Server1" is choosen randomly to be
    deleted along with mentioned "Server4" */
    },
    "status" : "successful"
},
Response Code = 200
```

Perform simple sanity checks on the request payload and ensure that server names mentioned in the Payload are less than or equal to the number of instances to be removed. Note that the server names are preferably mentioned with the delete request. One can never set the server names. In that case, servers are randomly selected for removal. However, sending a server list with a greater length than the number of removable instances will result in an error.

```
Payload Json= {
    "n" : 2,      /* wrong input n < len(servers) */
    "servers" : ["Server1", "Server4", "Server2"]
}
Response Json ={
    "message" : "<Error> Length of server list is more than removable instances",
    "status" : "failure"
},
Response Code = 400
```

5) **Endpoint (/read, method=POST):** Based on the consistent hashing algorithm, this endpoint reads data entries from the shard replicas across all server containers. The endpoint expects a range of *Stud_ids* (low, high) to be read from the distributed database. The load balancer derives all the required shards to be read from the provided *Stud_id* range and accumulates all read results before sending them to the client.
Hint: Due to our indexing (*Stud_id_low*, *Stud_id_low+Shard_size*) rule for each shard, we can easily find the shard entry from the Stud_id. (Calculation: *Stud_id_low+ Stud_id % Shard_size* entry will have the corresponding *Shard_id*).

```
Payload Json= {
    "Stud_id": {"low":1000, "high":8889}
}
Response Json ={
    "shards_queried": ["sh1",sh2","sh3"]
    "data" : [{"Stud_id":1000,"Stud_name":PQR,"Stud_marks":23},
              {"Stud_id":1001,"Stud_name":STV,"Stud_marks":22},
              .....,
              {"Stud_id":8888,"Stud_name":ZQN,"Stud_marks":65},
              {"Stud_id":8889,"Stud_name":BYHS,"Stud_marks":76}],
    "status" : "success"
},
Response Code = 200
```

6) **Endpoint (/write, method=POST):** This endpoint writes data entries in the distributed database. The endpoint expects multiple entries that are to be written in the server containers. The load balancer schedules each write to its corresponding shard replicas and ensures data consistency using mutex locks for a particular shard and its replicas. The general write work flow will be like: **(1) Get Shard_ids from Stud_ids and group writes for each shard → For each shard Do: (2a)Take mutex lock for Shard_id (m) → (2b) Get all servers (set S) having replicas of Shard_id (m) → (2c) Write entries in all servers (set S) in Shard_id (m) → (2d) Update the valid_idx of Shard_id (m) in the metadata if writes are successful → (2e) Release the mutex lock for Shard_id (m)**. An example request-response pair is shown.

```
Payload Json= {
    "data": [{"Stud_id":2255,"Stud_name":GHI,"Stud_marks":27},
             {"Stud_id":3524,"Stud_name":JKBFSFS,"Stud_marks":56}
             ...,
             {"Stud_id":1005,"Stud_name":YUBAAD,"Stud_marks":100}] /* 100 entries */
}
Response Json ={
    "message": "100 Data entries added",
    "status" : "success"
},
Response Code = 200
```

7) **Endpoint (/update, method=PUT):** This endpoint updates a particular data entry (based on *Stud_id*) in the distributed database. The load balancer retrieves all the shard replicas and their corresponding server instances where the entry has to be updated. The endpoint expects only one entry to be updated. An example request-response pair is shown. *Hint:* Note that the updates are blocking for a particular shard and its replicas to maintain data consistency.

```
Payload Json= {
    "Stud_id":"2255",
    "data": {"Stud_id":2255,"Stud_name":GHI,"Stud_marks":30} /* see marks got updated */
}
Response Json ={
    "message": "Data entry for Stud_id: 2255 updated",
    "status" : "success"
},
Response Code = 200
```

8) **Endpoint (/del, method=DELETE):** This endpoint deletes a particular data entry (based on *Stud_id*) in the distributed database. The load balancer retrieves all the shard replicas and their corresponding server instances where the entry has to be deleted. The endpoint expects only one entry that is to be deleted. An example request-response pair is shown. *Hint:* Note that the deletes are blocking for a particular shard and its replicas to maintain data consistency.

```
Payload Json= {
    "Stud_id":2255
}
Response Json ={
    "message": "Data entry with Stud_id:2255 removed from all replicas",
    "status" : "success"
},
Response Code = 200
```

IV. TASK3: ANALYSIS

To analyze the performance of the developed distributed database. You need to perform four subtasks. As mentioned earlier, the design provides scaling in two ways: (i) Read speed with more shard replicas and (ii) Size with more shards and servers.

A-1 Report the read and write speed for 10000 writes and 10000 reads in the default configuration given in task 2.

A-2 Increase the number of shard replicas (to 7) from the configuration (init endpoint). Report the write speed down for 10000 writes and read speed up for 10000 reads.

A-3 Increase the number of Servers (to 10) by adding new servers and increase the number of shards (shard to 6, shard replicas to 8). Define the (init endpoint) configurations according to your choice. Report the write speed up for 10000 writes and read speed up for 10000 reads.

A-4 Finally, check all the endpoints and ensure their correctness. Manually drop a server container and show that the load balancer spawns a new container and copies the shard entries from other replicas.

APPENDIX

We encourage you to use MYSQL:8.0 to implement the data tables in this assignment. However, you can use any database you want. To run the database and a flask app in the same container, you can follow the method below.

```
#Dockerfile
#-----------------------
FROM mysql:8.0-debian

COPY deploy.sh /always-initdb.d/        #here the flask app deploy script is copied
COPY . /bkr
WORKDIR /bkr

RUN apt-get update
RUN apt-get install -y python3
RUN apt-get install -y python3-pip

RUN pip install --upgrade pip
RUN pip install -r requirements.txt

ENV MYSQL_ROOT_PASSWORD="abc"   #host='localhost', user='root',password='abc'

EXPOSE 5000
```

Whatever is copied in the '/always-initdb.d/' folder will be run after the database is initialized. Therefore, the Flask app can connect to the local database with root credentials (mysql.connector.connect(host='localhost',user='root',password='abc').

```
#deploy.sh file
#---------------------------
#!/bin/bash
python3 flaskApp.py &
```

For reference, you can observe how the Dockerfile and deploy.sh file is coded for https://github.com/prasenjit52282/shardQ/tree/main/broker. For other databases (i.e., MongoDB, Cassandra, etc.), similar always-init scripts can be run when the database initializes in every startup. Feel free to choose any database of your choice.

A. Grading Scheme

- TASK1: Server - 40 %
- TASK2: Improve the load balancer - 30 %
- TASK3: Analysis - 20 %
- Documentation & Code Optimizations - 10 %

REFERENCES

[1] Docker, "What is a container?." https://docs.docker.com/guides/get-started/, 2024.

[2] Docker, "Use docker compose." https://docs.docker.com/get-started/08_using_compose/, 2024.

[3] makefiletutorial, "Learn makefiles with the tastiest examples." https://makefiletutorial.com/, 2024.

[4] J. Li, Y. Nie, and S. Zhou, "A dynamic load balancing algorithm based on consistent hash," in *2018 2nd IEEE Advanced Information Management,Communicates,Electronic and Automation Control Conference (IMCEC)*, pp. 2387–2391, 2018.

[5] T. Roughgarden and G. Valiant, "Cs168: The modern algorithmic toolbox lecture 1: Introduction and consistent hashing." https://web.stanford.edu/class/cs168/l/l1.pdf, 2022.

[6] DockerHub, "Mysql:8.0 docker image source in dockerhub." https://hub.docker.com/_/mysql, 2024.