CS60002: Distributed Systems

Assignment 3: Implementing a Write-Ahead Logging for consistency in Replicated Database with Sharding

Date: March 25th, 2024

Target Deadline (For sharing the git repo): April 14, 2024

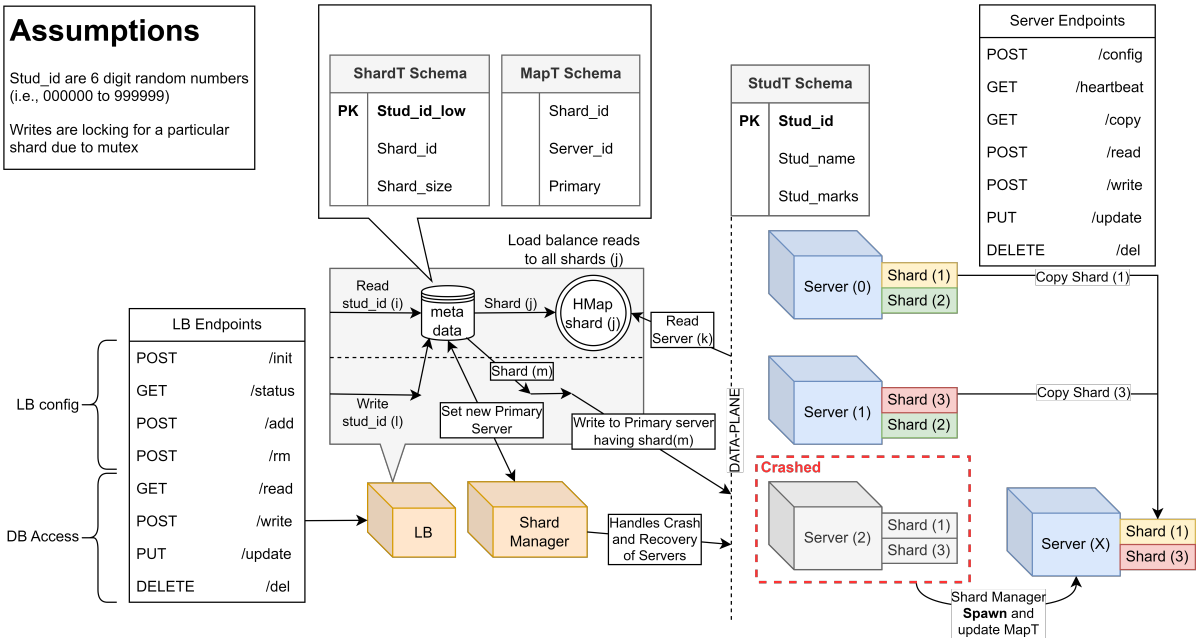

Fig. 1: System Diagram of a Sharded Database

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 you can reuse the codebase from the first and the second assignments. 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 have to implement a Write-Ahead Logging (WAL) mechanism for maintaining consistency among the replicas of the shards that are distributed among various servers. WAL is employed to guarantee that no data is lost and that the database can recover to a consistent state after an unexpected shutdown.

How WAL Works

1) `Logging Changes Before Writing to Disk`: With WAL, any changes to the database are first recorded in a log file before they are applied to the database files on disk. This log file is stored in a durable storage medium (like SSDs, RAID-configured HDDs or SSDs, and cloud storage).
2) `Sequential Writes`: The changes are written sequentially to the log, which is more efficient than random disk access, especially for large transactions or when multiple transactions are occurring simultaneously.
3) `Recovery Process`: In the event of a crash or restart, the system reads the WAL file(s) to redo operations that were not fully committed to the database files, ensuring that no committed transactions are lost. It can also undo any changes from transactions that were not completed, maintaining data integrity.

WAL for Replication and Consistency

1) `Replication Log Shipping`: A primary database server is chosen for each shard when writing to that shard is going to happen. The primary database server writes changes to its WAL as part of normal operations. These WAL records can then be shipped to replica servers, where they are replayed to apply the same changes. This ensures that all replicas process the same set of changes in the same order, maintaining consistency across the system.
2) `Synchronous Replication`: The primary waits for the majority replica to commit the data before committing its copy of data and acknowledging transactions, which ensures strong consistency. If the primary shard fails, then a new primary is chosen from the replicated shards having the most updated log entries. As soon as the downed database server is up, it copies all the shards from the primary shards. It ensures the system can be recovered from crash failure with consistent data.

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):** Keep same as in Assignment 2
2) **Endpoint (/copy, method=GET):** Keep same as in Assignment 2
3) **Endpoint (/read, method=POST):** Keep same as in Assignment 2
4) **Endpoint (/write, method=POST):** For secondary servers, this endpoint makes changes in the log and then writes data entries in a shard in a particular server container and for Primary server it first makes changes to its log, then send write request to other servers where same shard is present, once it gets confirmation from other secondary servers then it writes the data in its database. 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",
    "data": [{"Stud_id":2255,"Stud_name":GHI,"Stud_marks":27}, ...] /* 5 entries */
}
Response Json ={
    "message": "Data entries added",
    "status": "success"
},
Response Code = 200
```

5) **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*. Here also you need to follow the WAL mechanism. 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
```

6) **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. Here you need to add another component that is **Shard Manager**, which should run in a separate container. All the previous assumptions and implementation details still hold. Shard Manager is responsible for detecting if any server is down then making one server as Primary server and updating that in metadata. However, you must modify and integrate a few features to the load balancer

and shard manager to make it work for assignment 3 like, **/heartbeat** endpoint will be now called by Shard Manager, not by the load balancer. In the system diagram shown in Fig. 1, you can observe that the loadbalancer manages *Stud_id* $\rightarrow$ *Shard_id* $\rightarrow$ *Server_id* mapping with two data tables in the **metadata**. No **valid_idx** is required as we are using the WAL mechanism. The table schemas are as follows:

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

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 the WAL mechanism. The endpoints of the load balancer are as follows.

1) **Endpoint (/init, method=POST):** Same as Assignment 2
2) **Endpoint (/status, method=GET):** Same as Assignment 2
3) **Endpoint (/add, method=POST):** Same as Assignment 2
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. If Primary server for a shard is removed then also the Shard Manager should trigger the Primary selection from available servers for that shard. Please make a suitable endpoint for Shard Manager (like, **/primary_elect, method=GET**) to handle the request from Load Balancer once **/rm** endpoint is triggered. 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):** Same as Assignment 2
6) **Endpoint (/write, method=POST):** Same as Assignment 2
7) **Endpoint (/update, method=PUT):** Same as Assignment 2
8) **Endpoint (/del, method=DELETE):** Same as Assignment 2

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.