Design Laboratory (CS69202)
Project: BLINK DB - A High Performance Key-Value DB

Deadline: March 31, 2025 (11.59 PM) **Maximum Marks: 200**

General Instructions:

1. This project is supposed to be done individually.
2. The project is to be implemented in C/C++.
3. Implementation **modularity, and code quality** will be considered in evaluation.

BLINK DB is a key-value based, high-performance in-memory database inspired by Redis. It allows you to efficiently store and retrieve data, based on unique keys. **BLINK DB** operates as a standalone system rather than a distributed one. It runs on a single thread but handles multiple client requests using I/O multiplexing.

While it does not offer the benefits of distributed architectures, it provides advantages such as simpler deployment and maintenance, making it easier for developers to manage their data. **BLINK DB** is suitable for applications such as content management systems, agile development systems, etc.

In this project, you are going to implement **BLINK DB** step by step. In the first part, you will be designing and testing the **BLINK DB** storage engine, which can be run locally. Storage Engine is the underlying software component that a database management system uses to create, read, update and delete (CRUD) data from a database. In the second part, you will be creating a network infrastructure to handle multiple parallel connections. Finally, you will integrate both parts to create the complete **BLINK DB**. More details below.

This project is supposed to give you an idea of the following things:

1. How a database is designed from scratch, to be specific, what data structures are most suitable, data storage techniques, caching heuristics, etc.
2. How to integrate a database with network stack and make the most out of it.

Now, we present the project in 2 parts below:

Part 1: Implementing a Key-Value based Storage Engine

This part of the project involves developing a key-value based storage engine.

1. You will implement the storage engine by defining a class that encapsulates its functionality. It provides the following operations:

```
set(const char* key, const char* value); // Set the value for a given key
get(const char* key); // Get the value associated with a given key
del(const char* key); // Delete a key-value pair
```

2. You will implement a simple REPL (Read-Eval-Print Loop) to demonstrate the functionality of the storage engine in a C/C++ file.
    - The REPL only supports the following commands-

1. `SET <key> “<value>”`
2. `GET <key>`
3. `DEL <key>`

Your implementation should include an interface that allows users to input their queries. A sample interaction is provided below for reference.

```
User> GET mykey
NULL
User> SET mykey myvalue
User> GET mykey
myvalue
User> DEL mykey2
Does not exist.
User> SET mykey2 myvalue2
User> DEL mykey2
User> GET mykey2
```

Note: Since the database operates in memory, it may eventually reach capacity. To manage this, you must periodically flush outdated data to prevent overflow. However, when older data is accessed, it should be restored efficiently. Clearly outline the design decisions for this mechanism, including data eviction policies, retrieval strategies, and any trade-offs involved.

The goal of this project is to evaluate your design on its performance for `READs (GET)`, `INSERTs (SET)`, and `DELETEs (DEL)`.

1. The evaluation will involve running a set of benchmark files of queries (synchronously), containing a mixed workload. Hence it is advisable to choose your design such that no query type is too slow. You NEED to optimize for a particular workload among the following:
    - **read heavy** (less writes),
    - **balanced** (reads, and writes are comparable),
    - **write heavy** (reads are infrequent)

2. You will be required to submit a design document describing your engineering decisions, and optimizations. This document will specify:
    - **which among the above 3 workloads** are you optimizing for,
    - **the data structures** (BTree/LSM/anything else),
    - **any other specific optimizations** (optional)

Note 1, as mentioned above, you have to make all the design decisions for the storage engine yourself.

Note 2, in the next part we will add a network infrastructure to the engine, transforming the system into a client-server architecture using the Redis protocol**.** It is advisable to separate the implementation of the REPL, and the storage engine, for simpler reuse later.

Part 2: Putting it all together, BLINK DB

Now we will extend the storage engine to create a full-fledged database called **BLINK DB**. To accomplish this, we'll add a TCP server and a connection management layer. The core idea is to integrate a TCP server onto the **Storage Engine** from part 1 that manages client connections while communicating using the Redis wire protocol (RESP-2).

1. Implement a TCP server which handles multiple client requests asynchronously while communicating using the Redis wire protocol (RESP-2).

    a. Implement a TCP server that listens on port (say 9001) (default redis port: 6379).
    b. For efficient multi-connection handling, use **`epoll()`** (for Linux) or **`kqueue()`** (for Mac/BSD). Threads are also an alternative for concurrent processing. Choose based on scalability and performance needs, and document your decision with justification.
    c. Communication between server and client should be using RESP-2.
    d. Server should handle requests and update the database on the storage engine.
    e. Client encodes user commands `(SET, GET, DEL)` into RESP format, sends them to the server, and then decodes and displays the server's response
    f. Explain the design decisions in the connection management layer, in a design document.

2. We will evaluate you using the `redis-benchmark` utility from the official Redis project, and the rest on your implementation and design decisions, and you will be required to submit the results for GET, and SET queries, with different concurrency, and RPS.
    a. You need to submit redis-benchmark results for **10,000, 100,000, and 1,000,000** concurrent (`SET, GET`) requests over **10, 100, and 1000 parallel connections** respectively.
    b. Hence 3 ✕3 benchmark reports (concurrent request x parallel connections)

Note, given that `redis-benchmark` will be used as a part of the evaluation, make sure to correctly implement the RESP2 specification.

Documentation

For documentation, you will use doxygen to generate the documentation (in pdf as well as html) of your software. Write doxygen style comments and create a Doxyfile(doxygen config file for your project).
Note: Make sure that doxygen produces proper documentation of your project.

Deliverables

You are required to submit the following –

Note 1, Make a sub-directory for each part (`part-a`, `part-b`),
Note 2, The design document, and documentation of each part should be in the `docs` subdirectory, and source code should be in the `src` subdirectory.

1) **Part-A:**
    a. The code, and a shell script/Makefile for building and running the code.
    b. A design document as detailed in the problem statement.
    c. Documentation PDF generated using Doxygen.

2) **Part-B:**
    a. The code, and a shell script/Makefile for building and running the code.
    b. A design document as detailed in the problem statement.
    c. Documentation PDF generated using Doxygen.
    d. Redis-benchmark results in a `` `result` `` directory, 3 × 3 files, named as `result_{concurrent_requests}_{parallel_connections}.txt`, for example `result_1000000_100.txt` for the case with 1,000,000 concurrent requests, 100 parallel connections

Submit a single, well-structured ZIP archive named ***<rollno>_deslab_project.zip***, containing all required files in an organized directory structure, as stated above.

Evaluation Guidelines

The total marks for the project = 200. The marks breakup is given below:

Parts	**Marks**
Part 1	100
Part 2	100
Total	200