Design Document for BLINK DB

1. Overview

BLINK DB is a lightweight, in-memory database server with a persistent storage component that communicates using the RESP-2 protocol. The server is designed to efficiently handle thousands of concurrent client connections, making it compatible with Redis clients and benchmarking tools such as redis-benchmark.

2. Architecture

BLINK DB is composed of two primary layers:

- **Storage Engine:**
  This layer maintains an in-memory key-value store with basic operations (`SET`, `GET`, and `DEL`). It employs a thread-safe design using mutexes to protect concurrent access. It also flushes data to disk as needed, preserving a log of all operations.

- **Connection Management Layer:**
  This layer provides a TCP server that accepts client connections on a designated port (e.g., 9001). It uses epoll for efficient I/O multiplexing and a thread pool for concurrent processing of client requests. Commands are communicated using the RESP-2 protocol, ensuring compatibility with Redis clients.

3. Key Design Decisions

Epoll for Scalability

- **why??:**
  Epoll is used for efficient I/O multiplexing, allowing the server to manage thousands of concurrent connections with minimal overhead.
- **Advantages:**
  - More scalable than traditional mechanisms like `select()` or `poll()` when dealing with large numbers of file descriptors.
  - Provides edge-triggered event notifications to minimize redundant wakeups.

- **Implementation:**
  The server creates an epoll instance that monitors the listening socket and all active client sockets. When a client connection is ready for I/O, the server processes the corresponding event without blocking.

Multithreading for Concurrency

- **why??:**
  To ensure that one client's long-running operations do not block the processing of other clients' requests.
- **Advantages:**
  - Utilizes the hardware's multi-core capabilities to improve throughput.
  - Increases responsiveness under high concurrent load.
- **Implementation:**
  - A thread pool is used to limit the overhead of creating and destroying threads.
  - Each client request is dispatched to a worker thread from the pool.
  - Threads are managed via standard C++11 threads and are detached (or joined via the pool) to avoid explicit thread management in the main event loop.

RESP-2 Protocol

- **why??:**
  RESP-2 (Redis Serialization Protocol) is the communication standard for Redis. Implementing it ensures compatibility with Redis clients and benchmarking tools.
- **Advantages:**
  - Simplifies integration with existing Redis tools (e.g., `redis-cli`, `redis-benchmark`).
  - Standardizes message formatting for commands and responses.
- **Implementation:**
  - The server includes a parser that decodes incoming RESP-2 messages into command tokens.
  - Supported commands include `SET`, `GET`, `DEL`, and stubs for commands like `CONFIG GET` to suppress warnings.
  - Responses are encoded in the appropriate RESP-2 format before being sent back to the client.

Thread Safety

- **why??:**
  Since multiple threads access the in-memory key-value store concurrently, thread safety is crucial.
- **Advantages:**
  - Prevents race conditions and data corruption.
  - Ensures consistency of operations across concurrent accesses.
- **Implementation:**
  - The `StorageEngine` class uses `std::mutex` to guard the underlying data structures.
  - Logging operations and connection state management are also protected with mutexes.

Non-blocking I/O

- **why??:**
  Blocking I/O can cause delays when a client connection is slow or unresponsive. Non-blocking I/O ensures that the server can continue processing other events without waiting.
- **Advantages:**
  - Maximizes throughput by reducing idle waiting time.
  - Improves scalability and responsiveness.
- **Implementation:**
  - Client sockets are set to non-blocking mode using the `fcntl()` function.
  - Combined with epoll in edge-triggered mode, this approach ensures that the server reads available data without stalling.

4. Additional Considerations

- **Logging:**
  All operational logs are written to a dedicated log file (`blink_db.log`). The server avoids printing verbose logs on the terminal to prevent interference with client output.

- **Benchmarking:**
  To accurately measure performance and command processing, pipelining can be disabled during benchmarks (using the `-P 1` option with `redis-benchmark`).

5. Implementation Details

- **Server Operation:**
  The TCP server listens on port 9001. New client connections are accepted and added to the epoll instance. Each event from a client is processed by reading available data, parsing complete RESP-2 commands, executing the corresponding storage engine operations, and sending back the appropriate RESP-2 responses.

- **Logging and CONFIG GET Handling:**
  In addition to standard command logging, the server provides a stub for `CONFIG GET` to satisfy clients like `redis-benchmark` and suppress warnings.

6. Compilation and Execution Instructions

On the Server Side

Compile the Server Code:
Use the following command to compile with C++11 and pthread support:
```
g++ -std=c++11 -pthread part2.cpp -o part2
```
Run the Server:
Start the server by running:
```
./part2
```

On the Client Side

Using redis-cli:
Connect to the server using:`redis-cli -p 9001`

- You can then issue commands such as `SET`, `GET`, and `DEL`.

Benchmarking

Prepare the Benchmark Script:
Ensure your benchmark script (e.g., `benchmark_part2.sh`) is executable:
```
chmod +x benchmark_part2.sh
```
Run Benchmarking Tests:
Execute the script to run the benchmark tests:
```
./benchmark_part2.sh
```

7. Summary

BLINK DB – Part 2 builds upon the Part 1 storage engine by adding a network layer with a TCP server that uses epoll for scalable I/O multiplexing and a thread pool for efficient concurrent processing. By implementing the RESP-2 protocol, the system is fully compatible with Redis clients and benchmarking tools, making it a viable solution for real-world workloads. The design decisions—epoll, multithreading, non-blocking I/O, and thread safety—ensure that the server can handle thousands of concurrent connections efficiently. Detailed instructions for compiling, running, and benchmarking the server are provided to streamline deployment and performance evaluation.