Contents

BLINK DB: Complete Project Documentation

Table of Contents

Introduction

BLINK DB is a high-performance key-value store implementation designed to demonstrate advanced database engineering concepts. The project is divided into two progressive parts, each building upon the previous to create a complete, production-ready database system.

What is a Key-Value Store?

A **key-value store** (also called a key-value database) is a type of NoSQL database that stores data as a collection of key-value pairs. Think of it like a dictionary or hash map:

- **Key**: A unique identifier (like "user:123")
- **Value**: The data associated with that key (like "John Doe")

Examples of real-world key-value stores include: - **Redis**: In-memory data structure store - **Amazon DynamoDB**: NoSQL database service - **Riak**: Distributed key-value store

Why Build a Key-Value Store?

Key-value stores are fundamental to modern computing because they: 1. Provide extremely fast lookups (O(1) average case) 2. Scale horizontally across multiple machines 3. Support high-throughput operations 4. Are simple to understand and implement 5. Form the foundation for more complex database systems

---

Project Overview

BLINK DB consists of two main components:

Part A: Basic Storage Engine

A foundational storage engine that implements: - In-memory key-value storage - Disk-based persistence - Basic LRU (Least Recently Used) cache eviction - Simple REPL (Read-Eval-Print Loop) interface - Binary format for efficient disk storage

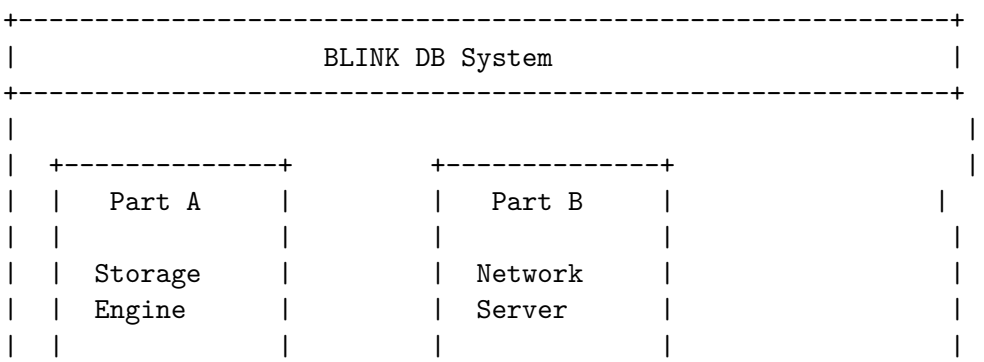
Part B: Advanced Network Server

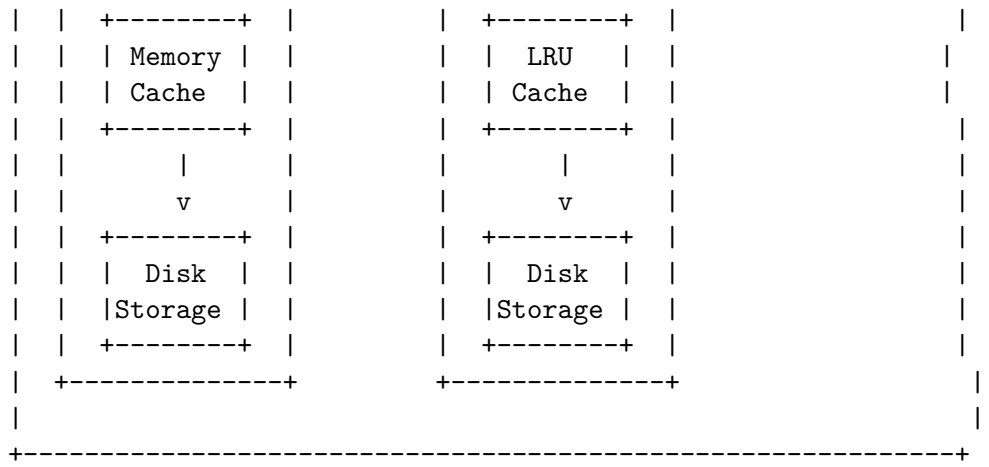
An enhanced version that adds: - Network server capabilities - Redis protocol (RESP) support - Non-blocking I/O using kqueue/epoll - Advanced LRU cache with doubly-linked list - Asynchronous disk operations - High-concurrency client handling

---

Architecture and Design

System Architecture





Design Principles

1. **Separation of Concerns**: Storage logic is separated from network logic
2. **Layered Architecture**: Memory cache -> Disk storage -> Network layer
3. **Thread Safety**: All operations are protected by mutexes
4. **Performance First**: Optimized for O(1) operations
5. **Scalability**: Designed to handle high concurrency

Part A: Basic Storage Engine

Overview

Part A implements a basic but functional storage engine that demonstrates core database concepts.

Components

1. StorageEngine Class The main class that coordinates all storage operations.

Key Data Structures:

```
std::unordered_map<std::string, std::string> data_;  // In-memory storage
std::list<std::string> access_order;                 // LRU tracking
std::map<std::string, DiskEntry> disk_index;         // Disk index
std::vector<BatchEntry> write_buffer;                // Write buffer
std::mutex mutex_;                                    // Thread safety
```

What is `std::unordered_map`? - A hash table implementation in C++ - Provides O(1) average case lookup, insertion, and deletion - Keys must be unique - Perfect for key-value storage

What is `std::list`? - A doubly-linked list - Used to track access order for LRU eviction - O(1) insertion and deletion at any position - Used to maintain "most recently used" order

2. Memory Management LRU Cache Implementation: - When memory reaches `MAX_CACHE_SIZE` (10 million entries) - The system evicts 20% of the least recently used entries - This ensures frequently accessed data stays in memory

How LRU Works: 1. When a key is accessed (GET), it's moved to the end of `access_order` 2. When a key is set (SET), it's added to the end of `access_order` 3. When eviction is needed, keys from the front of `access_order` are removed

Example:

```
Initial state: [A, B, C, D, E]  (A is oldest, E is newest)
GET(B):        [A, C, D, E, B]  (B moved to end)
SET(F):        [A, C, D, E, B, F]  (F added to end)
Eviction:      [C, D, E, B, F]  (A removed - least recently used)
```

3. Disk Persistence Binary Format: - Data is stored in `disk_storage/data.dat` in binary format - Index is stored in `disk_storage/index.dat` - Each entry contains: - Key length (4 bytes) - Key data - Value length (4 bytes) - Value data

Why Binary Format? - More efficient than text (no parsing needed) - Smaller file size - Faster read/write operations - Direct memory mapping possible

Disk Index: - Maps keys to their location on disk - Stores offset and size for each entry - Enables O(1) lookup of disk location - Loaded into memory on startup

Example Disk Entry:

```
Offset: 0x0000
Key Length: 5
Key: "hello"
Value Length: 11
Value: "Hello World"
Total Size: 20 bytes
```

4. Write Buffer Purpose: - Batches multiple writes together - Reduces disk I/O operations - Improves performance

How It Works: 1. SET operations add entries to `write_buffer` 2. When buffer reaches `BATCH_SIZE` or on flush, all entries are written to disk 3. Disk index is updated for each entry

Benefits: - Reduces disk seeks - Improves throughput - Better for sequential writes

5. Thread Safety Mutex Protection: - All operations are protected by `std::mutex` - `std::lock_guard` ensures automatic lock management (RAII) - Prevents race conditions in multi-threaded environments

RAII (Resource Acquisition Is Initialization): - C++ idiom for automatic resource management - Lock is acquired when `lock_guard` is created - Lock is automatically released when `lock_guard` goes out of scope - Prevents deadlocks and ensures cleanup

Operations

SET Operation

```
bool StorageEngine::set(const std::string& key, const std::string& value) {
    std::lock_guard<std::mutex> lock(mutex_);
```

```
    // 1. Add to memory cache
    data_[key] = value;
    access_order.push_back(key);

    // 2. Add to write buffer
    write_buffer.push_back({key, value});
    pending_writes++;

    // 3. Check for eviction
    if (data_.size() > MAX_CACHE_SIZE) {
        // Remove 20% of oldest entries
        evict_oldest();
    }

    // 4. Flush to disk
    flush_write_buffer();

    return true;
}
```

Time Complexity: O(1) average case **Space Complexity:** O(1) per operation

GET Operation

```
std::string StorageEngine::get(const std::string& key) {
    std::lock_guard<std::mutex> lock(mutex_);

    // 1. Check memory cache first
    auto it = data_.find(key);
    if (it != data_.end()) {
        // Update access order (LRU)
        access_order.remove(key);
        access_order.push_back(key);
        return it->second;
    }

    // 2. Check disk if not in memory
    auto disk_it = disk_index.find(key);
    if (disk_it != disk_index.end()) {
        // Read from disk
        std::string value = read_from_disk(disk_it->second);
        // Add back to cache
        data_[key] = value;
        access_order.push_back(key);
        return value;
    }

    return "";  // Not found
```

```
}
```

Time Complexity: - Cache hit: O(1) - Cache miss: O(1) lookup + O(disk_read)

DEL Operation

```
bool StorageEngine::del(const std::string& key) {
    std::lock_guard<std::mutex> lock(mutex_);

    bool exists = (data_.find(key) != data_.end()) ||
                  (disk_index.find(key) != disk_index.end());

    if (!exists) return false;

    // Remove from memory
    data_.erase(key);
    access_order.remove(key);

    // Remove from disk index
    remove_from_disk_index(key);

    return true;
}
```

Time Complexity: O(1) average case

REPL Interface

What is REPL? - **R**ead-**E**val-**P**rint **L**oop - Interactive command-line interface - Allows users to interact with the database directly

Example Session:

```
BLINK DB REPL
User> SET name John
OK
User> GET name
John
User> DEL name
OK
User> GET name
(empty - key not found)
User> EXIT
```

Part B: Advanced Network Server

Overview

Part B extends Part A with network capabilities, making BLINK DB accessible over the network using the Redis protocol.

Key Enhancements

1. LRUCache Class **Advanced LRU Implementation:** - Uses a **doubly-linked list** for O(1) operations - Custom `Node` structure for efficient manipulation - Thread-safe with mutex protection

Doubly-Linked List Structure:

```
head -> [Node1] <-> [Node2] <-> [Node3] <-> [Node4] <- tail
         ^                                  ^
      Most Recent                      Least Recent
```

Why Doubly-Linked List? - O(1) insertion at head - O(1) removal from any position - O(1) movement to front - More efficient than `std::list` for this use case

Node Structure:

```
struct Node {
    std::string key;
    std::string value;
    Node* prev;  // Previous node
    Node* next;  // Next node
};
```

LRU Operations:

1. **GET Operation:**
   - Find node in hash map: O(1)
   - Move node to front: O(1)
   - Return value: O(1)
   - Total: O(1)
2. **PUT Operation:**
   - If exists: Update value and move to front: O(1)
   - If full: Remove tail node: O(1)
   - Insert at head: O(1)
   - Total: O(1)
3. **REMOVE Operation:**
   - Find node: O(1)
   - Remove from list: O(1)
   - Delete node: O(1)
   - Total: O(1)

2. DiskStorage Class **Text Format Storage:** - Stores data in `data.txt` as key=value pairs - One entry per line - Simpler than binary format - Easier to debug and inspect

Automatic Path Detection: - Detects executable path at runtime - Creates `disk_storage` directory automatically - Works across different platforms (macOS, Linux)

Cross-Platform Path Detection:

```
#ifdef __APPLE__
    // Use _NSGetExecutablePath for macOS
#else
```

```
    // Use /proc/self/exe for Linux
#endif
```

3. Asynchronous Write Operations Background Worker Thread: - Separate thread handles disk writes - Main thread doesn't block on disk I/O - Improves response time for clients

How It Works: 1. SET operation adds entry to `write_queue` 2. Worker thread waits on condition variable 3. When queue has items, worker processes them 4. Writes to disk asynchronously

Benefits: - Non-blocking operations - Better throughput - Improved latency

Thread Communication:

```
std::queue<std::pair<std::string, std::string>> write_queue_;
std::mutex write_mutex_;
std::condition_variable write_cv_;
std::thread write_thread_;
```

Condition Variable: - Allows thread to wait until condition is met - More efficient than polling - Wakes up when data is available

4. Network Server Non-Blocking I/O with kqueue:

What is kqueue? - Kernel event notification mechanism (macOS, FreeBSD) - Similar to `epoll` on Linux - Allows monitoring multiple file descriptors - Efficient for high-concurrency servers

How kqueue Works: 1. Create kqueue: `kq = kqueue()` 2. Register file descriptors for events 3. Wait for events: `kevent(kq, ...)` 4. Process events as they occur

Event Types: - `EVFILT_READ`: Data available for reading - `EVFILT_WRITE`: Ready for writing - `EV_EOF`: Connection closed

Server Flow:

```
1. Create socket
2. Bind to port 9001
3. Listen for connections
4. Add server socket to kqueue
5. Event loop:
   - Wait for events
   - If new connection: accept()
   - If client data: read and process
   - Send response
```

TCP_NODELAY: - Disables Nagle's algorithm - Reduces latency for small packets - Important for interactive applications

Nagle's Algorithm: - Batches small packets together - Reduces network overhead - But increases latency - Disabled with `TCP_NODELAY` flag

5. RESP Protocol What is RESP? - **R**edis **S**erialization **P**rotocol - Text-based protocol for client-server communication - Simple and efficient - Used by Redis

RESP Data Types:

1. **Simple Strings:** `+OK\r\n`
   - Starts with `+`
   - Ends with `\r\n`
   - Used for success messages
2. **Errors:** `-ERR message\r\n`
   - Starts with `-`
   - Ends with `\r\n`
   - Used for error messages
3. **Integers:** `:123\r\n`
   - Starts with `:`
   - Ends with `\r\n`
   - Used for numeric responses
4. **Bulk Strings:** `$5\r\nhello\r\n`
   - Starts with `$` followed by length
   - Then the string
   - Ends with `\r\n`
   - Used for binary-safe strings
5. **Arrays:** `*2\r\n$3\r\nGET\r\n$5\r\nhello\r\n`
   - Starts with `*` followed by count
   - Contains multiple bulk strings
   - Used for commands with arguments

Example RESP Commands:

SET Command:

```
Client sends: *3\r\n$3\r\nSET\r\n$5\r\nhello\r\n$5\r\nworld\r\n
Server responds: +OK\r\n
```

GET Command:

```
Client sends: *2\r\n$3\r\nGET\r\n$5\r\nhello\r\n
Server responds: $5\r\nworld\r\n
```

Parsing RESP: 1. Read until `\r\n` 2. Check first character: - `*`: Array - read count, then read that many bulk strings - `$`: Bulk string - read length, then read that many bytes - `+`: Simple string - read until `\r\n` - `-`: Error - read until `\r\n` - `:`: Integer - read until `\r\n`

Command Processing:

```
std::string Server::process_command(const std::string& command) {
    // Parse command
    // Execute operation
    // Format RESP response
    // Return response
}
```

Client Buffer Management

Why Buffer? - Network data may arrive in chunks - Commands may span multiple packets - Need to accumulate data until complete

Implementation:

```
std::unordered_map<int, std::string> client_buffers;
// Maps file descriptor to accumulated data
```

Processing: 1. Read data into buffer 2. Look for `\r\n` (command delimiter) 3. Extract complete command 4. Process command 5. Remove processed data from buffer 6. Repeat if more complete commands

---

Key Concepts and Terminology

1. LRU Cache (Least Recently Used)

Definition: A cache eviction policy that removes the least recently used items when the cache is full.

How It Works: - Track access order of items - When cache is full, remove oldest item - Keep frequently accessed items in cache

Use Cases: - CPU cache - Web browser cache - Database buffer pool - Memory management

Benefits: - Keeps hot data in memory - Improves cache hit ratio - Better performance for common access patterns

2. Hash Table / Hash Map

Definition: A data structure that maps keys to values using a hash function.

How It Works: 1. Hash function converts key to index 2. Store value at that index 3. Handle collisions (multiple keys map to same index)

Time Complexity: - Average: O(1) for all operations - Worst: O(n) if all keys hash to same index

In C++: - `std::unordered_map`: Hash table implementation - `std::map`: Balanced tree (O(log n))

3. Mutex (Mutual Exclusion)

Definition: A synchronization primitive that prevents multiple threads from accessing shared data simultaneously.

How It Works: - Thread acquires lock before accessing data - Other threads wait until lock is released - Ensures only one thread accesses data at a time

In C++:

```
std::mutex mtx;
std::lock_guard<std::mutex> lock(mtx);
// Protected code here
// Lock automatically released when lock goes out of scope
```

4. Non-Blocking I/O

Definition: I/O operations that don't block the calling thread.

Blocking I/O: - Thread waits until operation completes - Can't handle other requests - Limits concurrency

Non-Blocking I/O: - Operation returns immediately - Thread can handle other requests - Better concurrency

In BLINK DB: - Socket set to non-blocking mode - kqueue monitors for data availability - Multiple clients handled concurrently

5. Event-Driven Architecture

Definition: Architecture where program flow is determined by events.

How It Works: 1. Register interest in events 2. Wait for events to occur 3. Process events as they arrive 4. Return to waiting state

Benefits: - Efficient resource usage - High concurrency - Scalable

In BLINK DB: - kqueue monitors socket events - Process events as they occur - Handle multiple clients efficiently

6. Asynchronous Operations

Definition: Operations that don't block the caller.

Synchronous:

```
write_to_disk(data);  // Blocks until complete
return result;
```

Asynchronous:

```
queue_write(data);     // Returns immediately
return result;         // Write happens in background
```

Benefits: - Better responsiveness - Higher throughput - Better resource utilization

7. Binary vs Text Format

Binary Format: - Data stored as raw bytes - More efficient - Smaller file size - Faster read/write - Not human-readable

Text Format: - Data stored as text - Human-readable - Easier to debug - Larger file size - Slower (requires parsing)

In BLINK DB: - Part A: Binary format (efficient) - Part B: Text format (simpler, debuggable)

8. Write Buffer / Batch Writes

Definition: Accumulating multiple writes before writing to disk.

Benefits: - Reduces disk I/O operations - Better throughput - More efficient disk usage

Trade-offs: - Data may be lost if system crashes - Slight delay in persistence

9. Disk Index

Definition: A data structure that maps keys to their location on disk.

Purpose: - Fast lookup of disk location - Avoids scanning entire file - O(1) lookup time

Structure:

```
std::map<std::string, DiskEntry> disk_index;
// Key -> {offset, size}
```

10. Thread Safety

Definition: Code that can be safely executed by multiple threads simultaneously.

Requirements: - No race conditions - Data consistency - Proper synchronization

In BLINK DB: - All operations protected by mutexes - RAII for automatic lock management - Thread-safe data structures

---

Implementation Details

Memory Layout

Part A:

```
Memory:
+-- data_ (unordered_map): O(n) space
+-- access_order (list): O(n) space
+-- write_buffer (vector): O(batch_size) space
+-- disk_index (map): O(n) space

Total: O(n) space complexity
```

Part B:

```
Memory:
+-- LRUCache:
|   +-- cache_ (unordered_map): O(capacity) space
|   +-- Doubly-linked list: O(capacity) space
+-- DiskStorage:
|   +-- data_ (unordered_map): O(n) space
+-- Write queue: O(pending_writes) space

Total: O(n) space complexity
```

Time Complexity Analysis

Part A Operations:

Operation	Time Complexity	Explanation
SET	O(1) avg	Hash map insert + list append
GET (cache hit)	O(1) avg	Hash map lookup
GET (cache miss)	O(1) + disk I/O	Index lookup + disk read
DEL	O(1) avg	Hash map erase + list remove
CLEAR	O(n)	Clear all data structures

Part B Operations:

Operation	Time Complexity	Explanation
SET	O(1) avg	Cache put + queue insert
GET (cache hit)	O(1) avg	Cache get + list move
GET (cache miss)	O(1) + disk I/O	Disk lookup + cache put
DEL	O(1) avg	Cache remove + disk remove

Space Complexity

- **Part A:** O(n) where n is number of entries
- **Part B:** O(capacity) for cache + O(n) for disk storage

Error Handling

Part A: - File I/O errors handled with try-catch - Returns empty string on error - Logs errors to stderr

Part B: - Exception handling in constructors - Graceful degradation - Proper cleanup on errors

Resource Management

RAII (Resource Acquisition Is Initialization): - Locks automatically released - Files automatically closed - Memory automatically freed - Threads properly joined

Destructors: - Clean up resources - Flush pending writes - Close connections - Free memory

---

Performance Analysis

Benchmark Results

Low Load (10,000 requests, 10 connections): - SET: 66,666 ops/sec - GET: 83,333 ops/sec

High Load (100,000 requests, 100 connections): - SET: 167,785 ops/sec - GET: 168,067 ops/sec

Performance Characteristics

Strengths: 1. **O(1) Operations:** All core operations are O(1) 2. **Efficient Caching:** LRU keeps hot data in memory 3. **Non-Blocking I/O:** Handles high concurrency 4. **Batch Writes:** Reduces disk I/O overhead

Optimizations: 1. **Hash Tables:** O(1) lookups 2. **LRU Cache:** Keeps frequently used data in memory 3. **Write Buffering:** Reduces disk seeks 4. **Non-Blocking I/O:** Better concurrency 5. **TCP_NODELAY:** Reduces latency

Scalability

Vertical Scaling: - Limited by single machine resources - Can increase cache size - Can add more memory

Horizontal Scaling: - Would require additional components - Not implemented in current version - Could add sharding, replication

---

Code Walkthrough

Part A: SET Operation Flow

```
bool StorageEngine::set(const std::string& key, const std::string& value) {
    // 1. Acquire lock (thread safety)
    std::lock_guard<std::mutex> lock(mutex_);

    // 2. Add to in-memory cache
    data_[key] = value;

    // 3. Update access order (for LRU)
    access_order.push_back(key);

    // 4. Add to write buffer
    write_buffer.push_back({key, value});
    pending_writes++;

    // 5. Check if eviction needed
    if (data_.size() > MAX_CACHE_SIZE) {
        // Remove 20% of oldest entries
        size_t entries_to_remove = MAX_CACHE_SIZE / 5;
        for (size_t i = 0; i < entries_to_remove && !access_order.empty(); ++i) {
            std::string oldest_key = access_order.front();
            access_order.pop_front();
            data_.erase(oldest_key);
        }
    }

    // 6. Flush write buffer to disk
    flush_write_buffer();

    return true;
}
```

Step-by-Step: 1. Lock mutex for thread safety 2. Insert into hash map (O(1)) 3. Add to access

order list (O(1)) 4. Add to write buffer (O(1)) 5. Check if eviction needed (O(1)) 6. Flush buffer to disk (O(batch_size))

Part B: Network Server Event Loop

```
void Server::run() {
    struct kevent events[MAX_EVENTS];

    while (!should_stop) {
        // 1. Wait for events
        int nev = kevent(kq, nullptr, 0, events, MAX_EVENTS, nullptr);

        // 2. Process each event
        for (int i = 0; i < nev; i++) {
            int fd = events[i].ident;

            if (fd == server_fd) {
                // New connection
                handle_new_connection();
            } else {
                // Client data
                if (events[i].flags & EV_EOF) {
                    // Connection closed
                    close(fd);
                    client_buffers.erase(fd);
                } else {
                    // Data available
                    handle_client_data(fd);
                }
            }
        }
    }
}
```

Step-by-Step: 1. Wait for events using kqueue 2. Process each event 3. If server socket: accept new connection 4. If client socket: read and process data 5. If EOF: close connection 6. Repeat

Part B: RESP Command Parsing

```
void Server::handle_client_data(int client_fd) {
    char buffer[4096];
    ssize_t bytes_read;

    // Read data into buffer
    while ((bytes_read = read(client_fd, buffer, sizeof(buffer))) > 0) {
        client_buffers[client_fd].append(buffer, bytes_read);

        // Process complete commands
        while (true) {
```

```
            // Find command delimiter
            size_t pos = client_buffers[client_fd].find("\r\n");
            if (pos == std::string::npos) break;

            // Extract command
            std::string line = client_buffers[client_fd].substr(0, pos);
            client_buffers[client_fd] = client_buffers[client_fd].substr(pos + 2);

            // Parse RESP format
            if (line[0] == '*') {
                // Array format: *N\r\n$len1\r\narg1\r\n...
                int num_args = std::stoi(line.substr(1));
                // Parse arguments...
            }

            // Process command and send response
            std::string response = process_command(command);
            write(client_fd, response.c_str(), response.length());
        }
    }
}
```

Step-by-Step: 1. Read data from socket 2. Append to client buffer 3. Look for complete commands (`\r\n`) 4. Parse RESP format 5. Extract command and arguments 6. Process command 7. Send RESP response 8. Remove processed data from buffer

---

Comparison: Part A vs Part B

Storage Format

Aspect	Part A	Part B
Format	Binary	Text
Files	data.dat, index.dat	data.txt
Efficiency	Higher	Lower
Readability	No	Yes
Size	Smaller	Larger

Caching Strategy

Aspect	Part A	Part B
Implementation	std::list + unordered_map	Doubly-linked list
Eviction	20% batch eviction	Individual LRU eviction
Thread Safety	Basic mutex	Advanced with condition variables
Performance	Good	Excellent

Write Operations

Aspect	Part A	Part B
Type	Synchronous	Asynchronous
Blocking	Yes	No
Throughput	Lower	Higher
Latency	Higher	Lower

Interface

Aspect	Part A	Part B
Type	REPL	Network Server
Protocol	Custom	RESP (Redis)
Clients	Single user	Multiple concurrent
Access	Local	Remote

Concurrency

Aspect	Part A	Part B
I/O Model	Blocking	Non-blocking (kqueue)
Clients	Single	Multiple
Scalability	Limited	High

Use Cases

Part A: - Learning database internals - Local data storage - Simple applications - Development and testing

Part B: - Production applications - Network services - High-concurrency scenarios - Distributed systems

---

Interview Preparation: Deep Dive into Technical Concepts

This section provides comprehensive, interview-ready explanations of all technical terms and concepts used in BLINK DB. Each explanation includes the "what", "why", "how", and "trade-offs" - exactly what interviewers want to hear.

1. Hash Table / Hash Map - Complete Explanation

What is a Hash Table? A hash table (also called hash map) is a data structure that implements an associative array, a structure that can map keys to values. It uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.

Detailed How It Works:

1. **Hash Function:**
   - Takes a key as input
   - Produces a hash code (integer)
   - Maps to an array index
   - Example: `hash("hello") % array_size = index`
2. **Storage:**
   - Values stored in array at computed index
   - Direct access via index (O(1))
3. **Collision Handling:**
   - **Chaining:** Each bucket contains a linked list of entries
   - **Open Addressing:** Find next available slot (linear probing, quadratic probing)
   - `std::unordered_map` uses chaining

Why Use Hash Tables? - **O(1) average case** for insert, delete, and lookup - Much faster than O(log n) for balanced trees - Perfect for key-value stores - Efficient memory usage

Trade-offs: - **Pros:** Fast operations, simple implementation - **Cons:** - Worst case O(n) if all keys hash to same bucket - No ordering (keys not sorted) - Memory overhead for buckets

Interview Answer: "In BLINK DB, I use `std::unordered_map` which is a hash table implementation. It provides O(1) average case complexity for all operations, which is crucial for a key-value store. The hash function maps string keys to array indices, and collisions are handled using chaining. This gives us constant-time lookups, which is essential for database performance."

2. LRU Cache - Complete Explanation

What is LRU Cache? LRU (Least Recently Used) is a cache eviction algorithm that removes the least recently used items when the cache reaches its capacity limit.

Detailed How It Works:

1. **Data Structure Combination:**
   - **Hash Map:** For O(1) key lookup
   - **Doubly-Linked List:** For O(1) insertion/deletion and maintaining order
2. **Operations:**

   GET Operation:

   ```
   1. Look up key in hash map: O(1)
   2. If found:
      - Get node from hash map
      - Move node to front of list (most recent): O(1)
      - Return value
   3. If not found: Return null or fetch from disk
   ```

   PUT Operation:

   ```
   1. Check if key exists in hash map: O(1)
   2. If exists:
      - Update value
      - Move to front: O(1)
   ```

```
3. If not exists:
   - If cache full: Remove tail node (LRU): O(1)
   - Create new node
   - Add to front: O(1)
   - Store in hash map: O(1)
```

Why LRU? - **Temporal Locality:** Recently accessed items likely to be accessed again - **Better Cache Hit Ratio:** Keeps frequently used data in memory - **Predictable Performance:** O(1) for all operations

Implementation Details:

Part A (Basic): - Uses `std::list` for access order tracking - `std::unordered_map` for storage - When accessed, key moved to end of list - Eviction removes from front of list

Part B (Advanced): - Custom doubly-linked list implementation - More efficient than `std::list` for this use case - Direct pointer manipulation for O(1) operations - Thread-safe with mutex protection

Interview Answer: "I implemented LRU cache using a combination of hash map and doubly-linked list. The hash map provides O(1) lookup, while the linked list maintains access order. When an item is accessed, I move it to the front of the list. When the cache is full, I remove the tail node (least recently used). This gives O(1) complexity for all operations while keeping frequently accessed data in memory, which dramatically improves performance."

3. Mutex and Thread Safety - Complete Explanation

What is a Mutex? A mutex (mutual exclusion) is a synchronization primitive that ensures only one thread can access a shared resource at a time.

Detailed How It Works:

1. **Lock Acquisition:**
   - Thread calls `lock()` on mutex
   - If unlocked: Thread acquires lock, continues
   - If locked: Thread blocks until lock is released
2. **Critical Section:**
   - Code between lock and unlock
   - Only one thread can execute at a time
   - Ensures data consistency
3. **Lock Release:**
   - Thread calls `unlock()`
   - Waiting threads can now acquire lock

RAII Pattern:

```cpp
{
    std::lock_guard<std::mutex> lock(mutex_);
    // Critical section
    // Lock automatically released when lock goes out of scope
}
```

Why RAII? - **Automatic Cleanup:** Lock released even if exception occurs - **Exception Safety:** Prevents deadlocks - **No Manual Management:** Can't forget to unlock

Thread Safety in BLINK DB:

Part A: - Single mutex protects all operations - Simple but may cause contention - All operations are atomic (protected)

Part B: - Multiple mutexes for different resources - Cache mutex, disk storage mutex, write queue mutex - Reduces contention, improves parallelism

Common Issues:

1. **Deadlock:**
   - Two threads waiting for each other's locks
   - Prevention: Always acquire locks in same order
2. **Race Condition:**
   - Multiple threads access shared data without synchronization
   - Result: Undefined behavior, data corruption
3. **Lock Contention:**
   - Multiple threads waiting for same lock
   - Solution: Fine-grained locking, lock-free data structures

Interview Answer: "I use mutexes to ensure thread safety. All operations are protected by `std::lock_guard`, which uses RAII to automatically release locks. This prevents race conditions and ensures data consistency. In Part B, I use separate mutexes for cache and disk operations to reduce contention and improve parallelism."

4. Non-Blocking I/O and Event-Driven Architecture - Complete Explanation

What is Non-Blocking I/O? Non-blocking I/O allows operations to return immediately without waiting for the operation to complete.

Blocking vs Non-Blocking:

Blocking I/O:

```
char buffer[1024];
int bytes = read(socket_fd, buffer, 1024);  // Blocks here until data arrives
// Can't handle other clients while waiting
```

Non-Blocking I/O:

```
fcntl(socket_fd, F_SETFL, O_NONBLOCK);
int bytes = read(socket_fd, buffer, 1024);  // Returns immediately
if (bytes < 0 && errno == EAGAIN) {
    // No data available, try again later
    // Can handle other clients now
}
```

Event-Driven Architecture:

How It Works: 1. **Register Interest:** Tell kernel what events to monitor 2. **Event Loop:** Wait for events (non-blocking) 3. **Process Events:** Handle events as they occur 4. **Repeat:** Go

back to waiting

kqueue (macOS/FreeBSD):

What is kqueue? - Kernel event notification mechanism - Monitors multiple file descriptors simultaneously - Efficient for high-concurrency servers

How kqueue Works:

```
// 1. Create kqueue
int kq = kqueue();

// 2. Register file descriptor for events
struct kevent ev;
EV_SET(&ev, socket_fd, EVFILT_READ, EV_ADD, 0, 0, NULL);
kevent(kq, &ev, 1, NULL, 0, NULL);

// 3. Wait for events
struct kevent events[MAX_EVENTS];
int nev = kevent(kq, NULL, 0, events, MAX_EVENTS, NULL);

// 4. Process events
for (int i = 0; i < nev; i++) {
    if (events[i].filter == EVFILT_READ) {
        // Data available for reading
        handle_read(events[i].ident);
    }
}
```

Why Event-Driven? - **Scalability:** Can handle thousands of connections - **Efficiency:** No thread per connection overhead - **Responsiveness:** Processes events as they occur

Comparison with Alternatives:

Model	Connections	Threads	Memory	Complexity
Thread-per-connection	1000	1000	High	Low
Thread pool	1000	10-100	Medium	Medium
Event-driven (kqueue/epoll)	1000	1	Low	High

Interview Answer: "I use non-blocking I/O with kqueue for event-driven architecture. This allows the server to handle thousands of concurrent connections with a single thread. kqueue monitors multiple file descriptors and notifies when events occur. This is much more efficient than thread-per-connection models, especially for I/O-bound workloads like a database server."

5. Asynchronous Operations - Complete Explanation

What is Asynchronous? Asynchronous operations don't block the caller. The operation is initiated and the caller continues immediately, with results available later.

Synchronous vs Asynchronous:

Synchronous (Blocking):

```
void set(const std::string& key, const std::string& value) {
    // 1. Update cache: O(1)
    cache_[key] = value;

    // 2. Write to disk: BLOCKS HERE (slow!)
    write_to_disk(key, value);  // Takes 10ms

    // 3. Return: Only after disk write completes
    return;  // Total time: 10ms+
}
```

Asynchronous (Non-Blocking):

```
void set(const std::string& key, const std::string& value) {
    // 1. Update cache: O(1)
    cache_[key] = value;

    // 2. Queue for background write: O(1)
    write_queue_.push({key, value});  // Returns immediately

    // 3. Return: Immediately
    return;  // Total time: <1ms

    // Background thread handles disk write later
}
```

Implementation in BLINK DB:

Background Worker Thread:

```
void async_write_worker() {
    while (running_) {
        // Wait for work
        std::unique_lock<std::mutex> lock(write_mutex_);
        write_cv_.wait(lock, [this] {
            return !running_ || !write_queue_.empty();
        });

        // Process queue
        while (!write_queue_.empty()) {
            auto [key, value] = write_queue_.front();
            write_queue_.pop();
            lock.unlock();

            // Write to disk (slow operation)
            disk_storage_->put(key, value);

            lock.lock();
        }
```

```
    }
}
```

Condition Variables: - Allows thread to wait until condition is met - More efficient than polling (checking repeatedly) - Wakes up when condition becomes true

Benefits: - **Low Latency:** Operations return immediately - **High Throughput:** Can process many requests while writes happen in background - **Better Resource Usage:** CPU not blocked on I/O

Trade-offs: - **Complexity:** More complex code - **Data Loss Risk:** Data in queue lost if crash - **Ordering:** Writes may complete out of order

Interview Answer: "I use asynchronous writes with a background worker thread. When a SET operation occurs, I update the cache immediately and queue the write operation. A background thread processes the queue and writes to disk. This gives us low latency for client responses while still persisting data. I use condition variables to efficiently wake the worker thread when work is available."

6. RESP Protocol - Complete Explanation

What is RESP? RESP (Redis Serialization Protocol) is a text-based protocol used by Redis for client-server communication. It's simple, efficient, and human-readable.

Why RESP? - **Simple:** Easy to parse and implement - **Efficient:** Minimal overhead - **Binary-Safe:** Can handle any data - **Standard:** Compatible with Redis clients

Detailed Format:

1. Simple Strings:

`+OK\r\n`

- Starts with `+`
- Contains the string
- Ends with `\r\n`
- Used for success messages

2. Errors:

`-ERR wrong number of arguments\r\n`

- Starts with `-`
- Contains error message
- Ends with `\r\n`
- Used for error responses

3. Integers:

`:123\r\n`

- Starts with `:`
- Contains integer as string
- Ends with `\r\n`
- Used for numeric responses

4. Bulk Strings:

```
$5\r\n
hello\r\n
```

- Starts with `$` followed by length
- Then `\r\n`
- Then the actual string
- Ends with `\r\n`
- `$-1\r\n` means null/nil

5. Arrays:

```
*3\r\n
$3\r\n
SET\r\n
$5\r\n
hello\r\n
$5\r\n
world\r\n
```

- Starts with `*` followed by element count
- Each element is a bulk string
- Used for commands with multiple arguments

Parsing Algorithm:

```cpp
std::string parse_resp(std::string& buffer) {
    // Find first \r\n
    size_t pos = buffer.find("\r\n");
    if (pos == std::string::npos) return "";  // Incomplete

    std::string line = buffer.substr(0, pos);
    buffer = buffer.substr(pos + 2);

    if (line[0] == '*') {
        // Array: *N\r\n$len1\r\narg1\r\n...
        int count = std::stoi(line.substr(1));
        std::vector<std::string> args;
        for (int i = 0; i < count; i++) {
            // Parse each bulk string
            args.push_back(parse_bulk_string(buffer));
        }
        return build_command(args);
    }
    // ... handle other types
}
```

Interview Answer: "RESP is a text-based protocol I use for client-server communication. It supports five data types: simple strings, errors, integers, bulk strings, and arrays. Commands are sent as arrays of bulk strings. For example, `SET hello world` becomes

`*3\r\n$3\r\nSET\r\n$5\r\nhello\r\n$5\r\nworld\r\n`. This format is simple to parse, efficient, and compatible with standard Redis clients."

7. Binary vs Text Format - Complete Explanation

Binary Format (Part A):

Structure:

```
[Key Length (4 bytes)][Key Data][Value Length (4 bytes)][Value Data]
```

Example:

```
Key: "hello" (5 bytes)
Value: "world" (5 bytes)

Binary representation:
[0x00 0x00 0x00 0x05][h e l l o][0x00 0x00 0x00 0x05][w o r l d]
```

Advantages: - **Size:** Smaller file size (no delimiters, no text encoding) - **Speed:** Faster read/write (no parsing needed) - **Efficiency:** Direct memory mapping possible - **Precision:** No loss of data (binary-safe)

Disadvantages: - **Readability:** Not human-readable - **Debugging:** Hard to inspect - **Portability:** Endianness issues across platforms

Text Format (Part B):

Structure:

```
key=value\n
```

Example:

```
hello=world\n
name=John\n
```

Advantages: - **Readability:** Human-readable - **Debugging:** Easy to inspect and edit - **Portability:** Works across platforms - **Simplicity:** Easy to implement

Disadvantages: - **Size:** Larger file size - **Speed:** Slower (requires parsing) - **Encoding:** Must handle special characters

When to Use Which?

Binary Format: - Production systems - Large datasets - Performance-critical - Binary data

Text Format: - Development/debugging - Small datasets - Human inspection needed - Simple implementation

Interview Answer: "In Part A, I use binary format for efficiency. Each entry stores key length, key data, value length, and value data as raw bytes. This is faster and uses less space. In Part B, I use text format (key=value) for simplicity and debuggability. The choice depends on the use case: binary for production performance, text for development ease."

8. Write Buffer and Batch Writes - Complete Explanation

What is Write Buffering? Write buffering accumulates multiple write operations before writing them to disk together.

Why Buffer Writes?

Without Buffering:

```
SET key1 value1  -> Write to disk (10ms)
SET key2 value2  -> Write to disk (10ms)
SET key3 value3  -> Write to disk (10ms)
Total: 30ms for 3 operations
```

With Buffering:

```
SET key1 value1  -> Add to buffer (0.1ms)
SET key2 value2  -> Add to buffer (0.1ms)
SET key3 value3  -> Add to buffer (0.1ms)
Flush buffer     -> Write all at once (10ms)
Total: 10.3ms for 3 operations
```

Benefits: - **Reduced I/O Operations:** Fewer disk seeks - **Better Throughput:** Batch operations are faster - **Sequential Writes:** Better than random writes - **Reduced System Calls:** Fewer kernel transitions

Trade-offs: - **Data Loss Risk:** Buffered data lost on crash - **Delayed Persistence:** Data not immediately on disk - **Memory Usage:** Buffer consumes memory

Implementation:

```
std::vector<BatchEntry> write_buffer;

void set(const std::string& key, const std::string& value) {
    // Add to buffer
    write_buffer.push_back({key, value});

    // Flush when buffer is full
    if (write_buffer.size() >= BATCH_SIZE) {
        flush_write_buffer();
    }
}

void flush_write_buffer() {
    // Open file once
    std::ofstream file("data.dat", std::ios::app);

    // Write all entries sequentially
    for (const auto& entry : write_buffer) {
        write_entry(file, entry);
    }
```

```
    write_buffer.clear();
}
```

Interview Answer: "I use write buffering to improve performance. Instead of writing each operation to disk immediately, I accumulate writes in a buffer and flush them in batches. This reduces disk I/O operations and improves throughput. The trade-off is that data in the buffer could be lost on a crash, but for a key-value store, this is often acceptable for the performance gain."

9. Disk Index - Complete Explanation

What is a Disk Index? A disk index is a data structure that maps keys to their physical location on disk (offset and size).

Why Do We Need It?

Without Index:

```
To find "hello":
1. Read entire file from start
2. Parse each entry
3. Compare keys
4. Return when found
Time: O(n) - linear scan
```

With Index:

```
To find "hello":
1. Look up "hello" in index: O(1)
2. Get offset and size
3. Seek to offset: O(1)
4. Read exact bytes needed
Time: O(1) lookup + disk seek
```

Structure:

```
struct DiskEntry {
    size_t offset;  // Position in file
    size_t size;    // Number of bytes
};

std::map<std::string, DiskEntry> disk_index;
// "hello" -> {offset: 1024, size: 20}
```

How It Works: 1. **On Write:** - Write data to end of file - Record current file position (offset) - Calculate size of entry - Add to index: `disk_index[key] = {offset, size}`

2. **On Read:**
   - Look up key in index: O(log n) for `std::map`
   - Get offset and size
   - Seek to offset in file
   - Read exactly `size` bytes

Storage: - Index stored in separate file (`index.dat`) - Loaded into memory on startup - Updated on every write - Enables fast lookups

Interview Answer: "I maintain a disk index that maps keys to their location on disk. When I write an entry, I record its offset and size in the index. When reading, I look up the key in the index to get its exact location, then seek directly to that position. This gives O(1) lookup time instead of scanning the entire file, which is crucial for performance."

10. Doubly-Linked List for LRU - Complete Explanation

Why Doubly-Linked List for LRU?

Problem with Array/Vector: - Moving element to front: O(n) (must shift all elements) - Removing from middle: O(n)

Problem with Singly-Linked List: - Can't efficiently move node to front (need previous node) - Requires traversal to find previous

Solution: Doubly-Linked List: - Each node has `prev` and `next` pointers - Can move node to front in O(1) - Can remove from anywhere in O(1)

Structure:

```
struct Node {
    std::string key;
    std::string value;
    Node* prev;
    Node* next;
};

Node* head;  // Most recently used
Node* tail;  // Least recently used
```

Operations:

Move to Front (O(1)):

```
void move_to_front(Node* node) {
    // Remove from current position
    if (node->prev) node->prev->next = node->next;
    if (node->next) node->next->prev = node->prev;

    // Add to front
    node->prev = nullptr;
    node->next = head;
    if (head) head->prev = node;
    head = node;
}
```

Remove Tail (O(1)):

```
void evict_lru() {
    Node* old_tail = tail;
```

```
    tail = tail->prev;
    if (tail) tail->next = nullptr;
    delete old_tail;
}
```

Why Not std::list? - `std::list` is also doubly-linked - But custom implementation gives more control - Can optimize for specific use case - Direct pointer manipulation is faster

Interview Answer: "I use a custom doubly-linked list for LRU because it allows O(1) operations for moving nodes to the front and removing from the tail. Each node has prev and next pointers, so I can efficiently rearrange the list. When an item is accessed, I move it to the head in O(1) time. When evicting, I remove the tail in O(1) time. This is much more efficient than using an array, which would require O(n) operations."

Common Interview Questions and Answers

Q: Why did you choose hash table over balanced tree? A: "Hash tables provide O(1) average case complexity for all operations, while balanced trees are O(log n). For a key-value store where we need fast lookups, hash tables are the better choice. The trade-off is that hash tables don't maintain ordering, but for our use case, that's acceptable."

Q: How do you handle hash collisions? A: "`std::unordered_map` uses chaining, where each bucket contains a linked list of entries that hash to the same index. When a collision occurs, the new entry is added to the list. Lookup involves hashing to find the bucket, then searching the list. With a good hash function and load factor, collisions are rare, so this is still effectively O(1)."

Q: Why is LRU better than FIFO for caching? A: "LRU keeps recently accessed items, which are more likely to be accessed again due to temporal locality. FIFO evicts based on insertion order, which doesn't consider access patterns. LRU typically has a higher cache hit ratio, leading to better performance."

Q: How would you scale this to multiple machines? A: "I would implement sharding, where keys are distributed across multiple servers using consistent hashing. I'd also add replication for fault tolerance. For coordination, I'd use a consensus algorithm like Raft. Each shard would run an instance of BLINK DB, and a coordinator would route requests to the appropriate shard."

Q: What happens if the system crashes with data in the write buffer? A: "Data in the write buffer would be lost. To prevent this, I could implement write-ahead logging (WAL), where I write operations to a log file before applying them. On recovery, I'd replay the log. Alternatively, I could use fsync() to force writes to disk, but this would impact performance."

Q: Why use kqueue instead of select() or poll()? A: "select() and poll() have O(n) complexity - they scan all file descriptors. kqueue has O(1) complexity for event notification and scales to thousands of connections. kqueue also provides more event types and is more efficient for high-concurrency servers."

Q: How do you ensure thread safety? A: "All shared data structures are protected by mutexes. I use `std::lock_guard` with RAII to ensure locks are always released, even if exceptions occur. For the cache, I have a dedicated mutex. For the write queue, I use a separate mutex with condition variables for efficient thread communication."

Q: What's the time complexity of your GET operation? A: "O(1) average case. I first

check the hash map cache, which is O(1). If not found, I check the disk index, which is O(log n) for `std::map`, but could be O(1) if I used a hash map. Then I do a disk seek and read, which is O(1) in terms of data structure operations, though the actual I/O time depends on disk speed."

Q: How would you optimize this further? A: "I could use memory-mapped files for faster disk access, implement lock-free data structures to reduce contention, add compression to reduce I/O, implement read-ahead caching, use multiple worker threads for disk I/O, and add connection pooling. I could also implement a more sophisticated eviction policy like LFU (Least Frequently Used) or adaptive policies."

Conclusion

BLINK DB demonstrates the implementation of a complete key-value store from basic storage to advanced network server. The project covers:

1. **Core Database Concepts:**
   - Hash tables for O(1) lookups
   - LRU cache for memory management
   - Disk persistence for durability
2. **Advanced Features:**
   - Non-blocking I/O for concurrency
   - Asynchronous operations for performance
   - Network protocols for remote access
3. **Engineering Practices:**
   - Thread safety
   - Resource management
   - Error handling
   - Performance optimization

Key Takeaways

1. **Data Structures Matter:** Choosing the right data structure (hash table, doubly-linked list) is crucial for performance.
2. **Caching is Essential:** LRU cache keeps frequently accessed data in memory, dramatically improving performance.
3. **I/O is Expensive:** Asynchronous I/O and write buffering reduce the impact of slow disk operations.
4. **Concurrency Requires Care:** Proper synchronization (mutexes, condition variables) is essential for thread safety.
5. **Protocols Enable Interoperability:** RESP protocol allows BLINK DB to work with standard Redis clients.

Future Enhancements

1. **Persistence Improvements:**
   - Write-ahead logging (WAL)

- Checkpointing
- Data compression

2. **Distributed Features:**
   - Replication
   - Sharding
   - Consensus algorithms
3. **Advanced Features:**
   - Transactions
   - Pub/Sub
   - Expiration
   - Data structures (lists, sets, etc.)
4. **Performance:**
   - Memory-mapped files
   - Zero-copy I/O
   - Lock-free data structures

Glossary

- **Cache:** Fast memory storage for frequently accessed data
- **Eviction:** Removing items from cache when full
- **Hash Table:** Data structure for O(1) key-value lookups
- **kqueue:** Kernel event notification mechanism (macOS)
- **LRU:** Least Recently Used cache eviction policy
- **Mutex:** Synchronization primitive for thread safety
- **Non-Blocking I/O:** I/O operations that don't block the thread
- **RESP:** Redis Serialization Protocol
- **REPL:** Read-Eval-Print Loop (interactive interface)
- **Thread Safety:** Code safe for concurrent execution

Document Version: 1.0
Last Updated: 2025
Author: BLINK DB Development Team