Blink DB - Part B

Generated by Doxygen 1.9.8

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Class Documentation

3.1 BlinkServer Class Reference

Implements a TCP server for BLINK DB.

```
#include <server.h>
```

Public Member Functions

- BlinkServer (int port, StorageEngine &storage_engine)

  Constructor.
- ∼**BlinkServer** ()

  Destructor.
- bool start ()

  Starts the server.
- void stop ()

  Stops the server.
- bool isRunning () const

  Checks if the server is running.

3.1.1 Detailed Description

Implements a TCP server for BLINK DB.

This class provides server functionality for the BLINK DB system, handling client connections using epoll for I/O multiplexing.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 BlinkServer()

```
BlinkServer::BlinkServer (
          int port,
          StorageEngine & storage_engine )
```

Constructor.

Parameters

port	Port to listen on.
storage_engine	Reference to the storage engine.

3.1.3 Member Function Documentation

3.1.3.1 isRunning()

```
bool BlinkServer::isRunning ( ) const
```

Checks if the server is running.

Returns

True if the server is running, false otherwise.

3.1.3.2 start()

```
bool BlinkServer::start ( )
```

Starts the server.

Returns

True if the server started successfully, false otherwise.

True if the server starts successfully, false otherwise.

3.1.3.3 stop()

```
void BlinkServer::stop ( )
```

Stops the server.

Stops the server and cleans up resources.

The documentation for this class was generated from the following files:

- server.h
- server.cpp

3.2 BloomFilter Class Reference

Implements a bloom filter for efficient negative lookups.

```
#include <storage_engine.h>
```

Public Member Functions

- BloomFilter (size_t expected_items=10000, double false_positive_rate=0.01)

  Constructs a BloomFilter.
- void add (const std::string &key)

  Adds a key to the bloom filter.
- bool mightContain (const std::string &key) const

  Checks if a key might be in the bloom filter.

3.2.1 Detailed Description

Implements a bloom filter for efficient negative lookups.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 BloomFilter()

```
BloomFilter::BloomFilter (
            size_t expected_items = 10000,
            double false_positive_rate = 0.01 )  [inline]
```

Constructs a BloomFilter.

Parameters

expected_items	The expected number of items to store.
false_positive_rate	The desired false positive rate.

3.2.3 Member Function Documentation

3.2.3.1 add()

```
void BloomFilter::add (
            const std::string & key )  [inline]
```

Adds a key to the bloom filter.

Parameters

key	The key to add.

3.2.3.2 mightContain()

```
bool BloomFilter::mightContain (
            const std::string & key ) const  [inline]
```

Checks if a key might be in the bloom filter.

Parameters

key	The key to check.

Returns

True if the key might be in the filter, false otherwise.

The documentation for this class was generated from the following file:

- storage_engine.h

3.3 LRUCache< K, V > Class Template Reference

Implements an LRU (Least Recently Used) cache.

```
#include <storage_engine.h>
```

Public Member Functions

- LRUCache (size_t capacity=1000)

  Constructs an LRUCache.
- **LRUCache** (const LRUCache &)=delete
- LRUCache & **operator=** (const LRUCache &)=delete
- **LRUCache** (LRUCache &&)=delete
- LRUCache & **operator=** (LRUCache &&)=delete
- std::optional< V > get (const K &key)

  Retrieves a value from the cache.
- void put (const K &key, const V &value)

  Inserts a key-value pair into the cache.

3.3.1 Detailed Description

template<typename K, typename V>
class LRUCache< K, V >

Implements an LRU (Least Recently Used) cache.

Template Parameters

K	The type of the cache keys.
V	The type of the cache values.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 LRUCache()

```
template<typename K , typename V >
LRUCache< K, V >::LRUCache (
             size_t capacity = 1000 )  [inline], [explicit]
```

Constructs an LRUCache.

Parameters

capacity	The maximum number of items the cache can hold.

3.3.3 Member Function Documentation

3.3.3.1 get()

```
template<typename K , typename V >
std::optional< V > LRUCache< K, V >::get (
             const K & key )  [inline]
```

Retrieves a value from the cache.

Parameters

key	The key to retrieve.

Returns

The value if found, or std::nullopt if not found.

3.3.3.2 put()

```
template<typename K , typename V >
void LRUCache< K, V >::put (
             const K & key,
             const V & value )  [inline]
```

Inserts a key-value pair into the cache.

Parameters

key	The key to insert.
value	The value to associate with the key.

The documentation for this class was generated from the following file:

- storage_engine.h

3.4 StorageEngine Class Reference

Implements the LSM-based storage engine.

```
#include <storage_engine.h>
```

Public Member Functions

- StorageEngine (size_t max_memory_size=1024 ∗1024 ∗100, size_t memtable_size=1024 ∗1024 ∗10)

  Constructs a StorageEngine.
- ∼StorageEngine ()

  Destructor for the StorageEngine.
- bool set (const std::string &key, const std::string &value)

  Inserts or updates a key-value pair in the storage engine.
- bool get (const std::string &key, std::string &value)

  Retrieves the value associated with a key.
- bool del (const std::string &key)

  Deletes a key from the storage engine.
- bool multiSet (const std::vector< std::pair< std::string, std::string > > &kvs)

  Inserts multiple key-value pairs in the storage engine.
- bool multiGet (const std::vector< std::string > &keys, std::vector< std::pair< std::string, std::optional< std::string > > > &results)

  Retrieves multiple values associated with keys.
- size_t getMemoryUsage () const

  Gets the current memory usage of the storage engine.

3.4.1 Detailed Description

Implements the LSM-based storage engine.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 StorageEngine()

```
StorageEngine::StorageEngine (
            size_t max_memory_size = 1024 * 1024 * 100,
            size_t memtable_size = 1024 * 1024 * 10 )
```

Constructs a StorageEngine.

Constructs a StorageEngine object.

Parameters

max_memory_size	The maximum memory size allowed for the storage engine.
memtable_size	The size threshold for the memtable before flushing.

3.4.2.2 ∼StorageEngine()

```
StorageEngine::∼StorageEngine ( )
```

Destructor for the StorageEngine.

Destructor for the StorageEngine class.

3.4.3 Member Function Documentation

3.4.3.1 del()

```
bool StorageEngine::del (
             const std::string & key )
```

Deletes a key from the storage engine.

Parameters

key	The key to delete.

Returns

True if the operation is successful, false otherwise.

3.4.3.2 get()

```
bool StorageEngine::get (
             const std::string & key,
             std::string & value )
```

Retrieves the value associated with a key.

Parameters

key	The key to retrieve.
value	The retrieved value.

Returns

True if the key exists, false otherwise.

3.4.3.3 getMemoryUsage()

```
size_t StorageEngine::getMemoryUsage ( ) const
```

Gets the current memory usage of the storage engine.

Returns

The current memory usage.

3.4.3.4 multiGet()

```
bool StorageEngine::multiGet (
            const std::vector< std::string > & keys,
            std::vector< std::pair< std::string, std::optional< std::string > > > & results
)
```

Retrieves multiple values associated with keys.

Parameters

keys	The keys to retrieve.
results	The retrieved key-value pairs.

Returns

True if the operation is successful, false otherwise.

3.4.3.5 multiSet()

```
bool StorageEngine::multiSet (
            const std::vector< std::pair< std::string, std::string > > & kvs )
```

Inserts multiple key-value pairs in the storage engine.

Parameters

kvs	The key-value pairs to insert.

Returns

True if the operation is successful, false otherwise.

3.4.3.6 set()

```
bool StorageEngine::set (
            const std::string & key,
            const std::string & value )
```

Inserts or updates a key-value pair in the storage engine.

Parameters

key	The key to insert or update.
value	The value to associate with the key.

Returns

True if the operation is successful, false otherwise.

The documentation for this class was generated from the following files:

- storage_engine.h
- storage_engine.cpp

3.5 ThreadPool Class Reference

Implements a thread pool for background operations.

```
#include <storage_engine.h>
```

Public Member Functions

- ThreadPool (size_t num_threads)

  Constructs a ThreadPool.
- void enqueue (std::function< void()> task)

  Enqueues a task into the thread pool.
- ∼**ThreadPool** ()

  Destructor for the ThreadPool.

3.5.1 Detailed Description

Implements a thread pool for background operations.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 ThreadPool()

```
ThreadPool::ThreadPool (
             size_t num_threads )  [inline]
```

Constructs a ThreadPool.

Parameters

num_threads	The number of threads in the pool.

3.5.3 Member Function Documentation

3.5.3.1 enqueue()

```
void ThreadPool::enqueue (
            std::function< void()> task )  [inline]
```

Enqueues a task into the thread pool.

Parameters

task	The task to enqueue.

The documentation for this class was generated from the following file:

- storage_engine.h

3.6 resp::Value Class Reference

Class representing a RESP-2 value.

```
#include <resp.h>
```

Public Member Functions

- Type getType () const

  Get the type of this value.
- std::optional< std::string > getString () const

  Get the string value.
- std::optional< int64_t > getInteger () const

  Get the integer value.
- std::optional< std::vector< Value > > getArray () const

  Get the array values.
- bool isNull () const

  Check if this is a null value.
- std::string serialize () const

  Serialize this value to a RESP-2 message.

Static Public Member Functions

- static Value createSimpleString (const std::string &str)

  Create a RESP Simple String.
- static Value createError (const std::string &message)

  Create a RESP Error.
- static Value createInteger (int64_t value)

  Create a RESP Integer.
- static Value createBulkString (const std::string &str)

Create a RESP Bulk String.
- static Value createNullBulkString ()

  Create a RESP Null Bulk String.
- static Value createArray (const std::vector< Value > &values)

  Create a RESP Array.
- static Value createNullArray ()

  Create a RESP Null Array.
- static std::optional< Value > deserialize (const std::string &data, size_t &consumed)

  Deserialize a RESP-2 message.

3.6.1 Detailed Description

Class representing a RESP-2 value.

This class provides methods to create, serialize, and deserialize RESP-2 values. It supports all RESP-2 data types, including Simple Strings, Errors, Integers, Bulk Strings, and Arrays.

3.6.2 Member Function Documentation

3.6.2.1 createArray()

```
Value resp::Value::createArray (
            const std::vector< Value > & values )  [static]
```

Create a RESP Array.

Creates a RESP Array value.

Parameters

values	The array values.

Returns

A new Value object representing an Array.

Parameters

values	The array elements.

Returns

A Value object representing an Array.

3.6.2.2 createBulkString()

```
Value resp::Value::createBulkString (
            const std::string & str )  [static]
```

Create a RESP Bulk String.

Creates a RESP Bulk String value.

Parameters

str	The string value.

Returns

A new Value object representing a Bulk String.

Parameters

str	The string content.

Returns

A Value object representing a Bulk String.

3.6.2.3 createError()

```
Value resp::Value::createError (
            const std::string & message )  [static]
```

Create a RESP Error.

Creates a RESP Error value.

Parameters

message	The error message.

Returns

A new Value object representing an Error.

Parameters

message	The error message.

Returns

A Value object representing an Error.

3.6.2.4 createInteger()

```
Value resp::Value::createInteger (
            int64_t value )  [static]
```

Create a RESP Integer.

Creates a RESP Integer value.

Parameters

value	The integer value.

Returns

A new Value object representing an Integer.

Parameters

value	The integer value.

Returns

A Value object representing an Integer.

3.6.2.5 createNullArray()

```
Value resp::Value::createNullArray ( )  [static]
```

Create a RESP Null Array.

Creates a RESP Null Array value.

Returns

A new Value object representing a Null Array.

A Value object representing a Null Array.

3.6.2.6 createNullBulkString()

```
Value resp::Value::createNullBulkString ( )  [static]
```

Create a RESP Null Bulk String.

Creates a RESP Null Bulk String value.

Returns

A new Value object representing a Null Bulk String.

A Value object representing a Null Bulk String.

3.6.2.7 createSimpleString()

```
Value resp::Value::createSimpleString (
             const std::string & str )  [static]
```

Create a RESP Simple String.

Creates a RESP Simple String value.

Parameters

str	The string value.

Returns

A new Value object representing a Simple String.

Parameters

str	The string content.

Returns

A Value object representing a Simple String.

3.6.2.8 deserialize()

```
std::optional< Value > resp::Value::deserialize (
            const std::string & data,
            size_t & consumed )  [static]
```

Deserialize a RESP-2 message.

Deserializes a RESP value from a string.

Parameters

	data	The serialized RESP-2 message.
out	*consumed*	The number of bytes consumed during deserialization.

Returns

The deserialized Value object, or std::nullopt if deserialization fails.

Parameters

data	The serialized RESP string.
consumed	The number of characters consumed during deserialization.

Returns

The deserialized Value object or std::nullopt if deserialization fails.

3.6.2.9 getArray()

```
std::optional< std::vector< Value > > resp::Value::getArray ( ) const
```

Get the array values.

Gets the array content of the RESP value, if applicable.

Returns

The array values if this is an array type, or std::nullopt otherwise.

The array content or std::nullopt if not applicable.

3.6.2.10 getInteger()

```
std::optional< int64_t > resp::Value::getInteger ( ) const
```

Get the integer value.

Gets the integer content of the RESP value, if applicable.

Returns

The integer value if this is an integer type, or std::nullopt otherwise.

The integer content or std::nullopt if not applicable.

3.6.2.11 getString()

```
std::optional< std::string > resp::Value::getString ( ) const
```

Get the string value.

Gets the string content of the RESP value, if applicable.

Returns

The string value if this is a string type, or std::nullopt otherwise.

The string content or std::nullopt if not applicable.

3.6.2.12 getType()

```
Type resp::Value::getType ( ) const
```

Get the type of this value.

Gets the type of the RESP value.

Returns

The type of the value.

3.6.2.13 isNull()

```
bool resp::Value::isNull ( ) const
```

Check if this is a null value.

Checks if the RESP value is null.

Returns

True if this is a null value, false otherwise.

True if the value is null, false otherwise.

3.6.2.14 serialize()

```
std::string resp::Value::serialize ( ) const
```

Serialize this value to a RESP-2 message.

Serializes the RESP value into a string.

Returns

The serialized RESP-2 message as a string.

The serialized RESP string.

The documentation for this class was generated from the following files:

- resp.h
- resp.cpp

Chapter 4

File Documentation

4.1 client.cpp File Reference

Simple client for the BLINK DB server.

```
#include "resp.h"
#include <iostream>
#include <string>
#include <sstream>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <vector>
#include <algorithm>
```

Include dependency graph for client.cpp:

4.2 main.cpp File Reference

Entry point for the BLINK DB server application.

```
#include "storage_engine.h"
#include "server.h"
#include <iostream>
#include <signal.h>
#include <unistd.h>
```

Include dependency graph for main.cpp:

Functions

- void signalHandler (int signal)

  Signal handler for gracefully shutting down the server.

- int main (int argc, char ∗argv[ ])

  Main function for the BLINK DB server application.

Variables

- BlinkServer ∗ **g_server** = nullptr

4.2.1 Detailed Description

Entry point for the BLINK DB server application.

This file initializes the BLINK DB server, sets up signal handling, and starts the server to handle client requests. It uses the LSM-based storage engine for efficient data management.

4.2.2 Function Documentation

4.2.2.1 main()

```
int main (
            int argc,
            char * argv[] )
```

Main function for the BLINK DB server application.

This function parses command-line arguments, initializes the storage engine and server, sets up signal handlers, and starts the server to handle client requests.

Supported command-line options:

- `--port PORT`: Set the server port (default: 9001).
- `--memory SIZE`: Set the maximum memory size in MB for the storage engine (default: 100MB).
- `--help`: Display usage information.

Parameters

argc	The number of command-line arguments.
argv	The array of command-line arguments.

Returns

0 on successful execution, or a non-zero value on error.

4.2.2.2 signalHandler()

```
void signalHandler (
            int signal )
```

Signal handler for gracefully shutting down the server.

This function is called when the server receives a termination signal (e.g., SIGINT or SIGTERM). It stops the server and performs cleanup.

Parameters

signal	The signal number received.

4.3 resp.cpp File Reference

Implementation of Redis RESP-2 protocol.

```
#include "resp.h"
#include <sstream>
#include <string>
```

Include dependency graph for resp.cpp:

Variables

- const char **resp::CRLF** [ ] = "\r\n"

  Carriage return and line feed sequence.
- const char **resp::SIMPLE_STRING_PREFIX** = '+'

  Prefix for RESP Simple Strings.
- const char **resp::ERROR_PREFIX** = '-'

  Prefix for RESP Errors.
- const char **resp::INTEGER_PREFIX** = ':'

  Prefix for RESP Integers.
- const char **resp::BULK_STRING_PREFIX** = '$'

  Prefix for RESP Bulk Strings.
- const char **resp::ARRAY_PREFIX** = '*'

  Prefix for RESP Arrays.

4.3.1 Detailed Description

Implementation of Redis RESP-2 protocol.

This file provides the implementation of the RESP (REdis Serialization Protocol) used for communication between the client and server. It includes serialization and deserialization of RESP data types such as Simple Strings, Errors, Integers, Bulk Strings, and Arrays.

4.4 resp.h File Reference

Implementation of Redis RESP-2 protocol.

```
#include <string>
#include <vector>
#include <optional>
```

Include dependency graph for resp.h: This graph shows which files directly or indirectly include this file:

Classes

- class resp::Value

  Class representing a RESP-2 value.

Enumerations

- enum class resp::Type {
  SimpleString , Error , Integer , BulkString ,
  Array }

  Enumeration for RESP data types.

4.4.1 Detailed Description

Implementation of Redis RESP-2 protocol.

This file contains the definitions for serializing and deserializing RESP-2 protocol messages. RESP is the Redis Serialization Protocol used for client-server communication.

4.4.2 Enumeration Type Documentation

4.4.2.1 Type

```
enum class resp::Type  [strong]
```

Enumeration for RESP data types.

Enumerator

SimpleString	Simple string prefixed with '+'.
Error	Error prefixed with '-'.
Integer	Integer prefixed with ':'.
BulkString	Bulk string prefixed with '$'.
Array	Array prefixed with '∗'.

4.5 resp.h

Go to the documentation of this file.

```

#ifndef RESP_H
#define RESP_H

#include <string>
#include <vector>
#include <optional>

namespace resp {

enum class Type {
    SimpleString,
    Error,
    Integer,
    BulkString,
    Array
};

class Value {
public:
    static Value createSimpleString(const std::string& str);

    static Value createError(const std::string& message);

    static Value createInteger(int64_t value);

    static Value createBulkString(const std::string& str);

    static Value createNullBulkString();

    static Value createArray(const std::vector<Value>& values);

    static Value createNullArray();

    Type getType() const;

    std::optional<std::string> getString() const;

    std::optional<int64_t> getInteger() const;

    std::optional<std::vector<Value» getArray() const;

    bool isNull() const;

    std::string serialize() const;

    static std::optional<Value> deserialize(const std::string& data, size_t& consumed);

private:
    Type type_;
    bool null_ = false;
    std::string string_value_;
    int64_t integer_value_ = 0;
    std::vector<Value> array_values_;

    explicit Value(Type type);
};

} // namespace resp

#endif // RESP_H
```

4.6 server.cpp File Reference

Implementation of the BLINK DB server.

```
#include "server.h"
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/epoll.h>
#include <errno.h>
#include <iostream>
#include <string.h>
```

Include dependency graph for server.cpp:

Variables

- const int **MAX_EVENTS** = 64

  Maximum number of events to process at once.
- const int **BACKLOG** = 128

  Maximum number of pending connections in the listen queue.
- const int **BUFFER_SIZE** = 4096

  Buffer size for reading client data.

4.6.1 Detailed Description

Implementation of the BLINK DB server.

4.7 server.h File Reference

Header file for the BLINK DB server.

```
#include "storage_engine.h"
#include "resp.h"
#include <string>
#include <unordered_map>
#include <vector>
#include <atomic>
#include <thread>
#include <functional>
```

Include dependency graph for server.h: This graph shows which files directly or indirectly include this file:

Classes

- class BlinkServer

  Implements a TCP server for BLINK DB.

4.7.1 Detailed Description

Header file for the BLINK DB server.

This file contains the declaration of the BlinkServer class, which handles network connections using epoll for efficient I/O multiplexing.

4.8 server.h

Go to the documentation of this file.

```

#ifndef SERVER_H
#define SERVER_H

#include "storage_engine.h"
#include "resp.h"
#include <string>
#include <unordered_map>
#include <vector>
#include <atomic>
#include <thread>
#include <functional>

class BlinkServer {
public:
    BlinkServer(int port, StorageEngine& storage_engine);

    ~BlinkServer();

    bool start();

    void stop();

    bool isRunning() const;

private:
    struct ClientConnection {
        int fd;
        std::string buffer;
        std::string output_buffer;
    };

    bool initEpoll();

    bool startListening();

    void acceptClient();

    void handleClient(int client_fd);

    resp::Value processCommand(const resp::Value& command);

    void sendResponse(int client_fd, const resp::Value& response);

    void closeClient(int client_fd);

    void serverLoop();

    int port_;
    StorageEngine& storage_engine_;
    int server_fd_ = -1;
    int epoll_fd_ = -1;
    std::atomic<bool> running_ = {false};
    std::thread server_thread_;
    std::unordered_map<int, ClientConnection> clients_;
};

#endif // SERVER_H
```

4.9 storage_engine.cpp File Reference

Implementation of the StorageEngine class.

```
#include "storage_engine.h"
#include <iostream>
#include <algorithm>
```

Include dependency graph for storage_engine.cpp:

4.9.1 Detailed Description

Implementation of the StorageEngine class.

This file provides the implementation of the LSM-based storage engine, including methods for data insertion, retrieval, deletion, and compaction.

4.10 storage_engine.h File Reference

Header file for the LSM-based storage engine.

```
#include <string>
#include <map>
#include <vector>
#include <memory>
#include <mutex>
#include <shared_mutex>
#include <atomic>
```

```
#include <chrono>
#include <array>
#include <deque>
#include <functional>
#include <optional>
#include <unordered_map>
#include <list>
#include <thread>
#include <condition_variable>
#include <queue>
#include <stdexcept>
#include <cmath>
```

Include dependency graph for storage_engine.h: This graph shows which files directly or indirectly include this file:

Classes

- class BloomFilter

  Implements a bloom filter for efficient negative lookups.
- class LRUCache< K, V >

  Implements an LRU (Least Recently Used) cache.
- class ThreadPool

  Implements a thread pool for background operations.
- class StorageEngine

  Implements the LSM-based storage engine.

Macros

- #define **NUM_SHARDS** 16

  Number of shards for partitioned memtable.
- #define **NUM_CACHES** 4

  Number of read cache segments.

4.10.1 Detailed Description

Header file for the LSM-based storage engine.

This file contains the declaration of the StorageEngine class and its supporting components, including memtables, SSTables, bloom filters, LRU caches, and thread pools.

4.11 storage_engine.h

Go to the documentation of this file.

```

#ifndef STORAGE_ENGINE_H
#define STORAGE_ENGINE_H

#include <string>
#include <map>
#include <vector>
#include <memory>
#include <mutex>
#include <shared_mutex>
#include <atomic>
#include <chrono>
```

```
#include <array>
#include <deque>
#include <functional>
#include <optional>
#include <unordered_map>
#include <list>
#include <thread>
#include <condition_variable>
#include <queue>
#include <stdexcept>
#include <cmath>

#define NUM_SHARDS 16

#define NUM_CACHES 4

class BloomFilter {
public:
    BloomFilter(size_t expected_items = 10000, double false_positive_rate = 0.01) {
        bit_array_size_ = static_cast<size_t>(-expected_items * std::log(false_positive_rate) / (std::log(2) * std::log(2)));
        num_hashes_ = static_cast<size_t>(bit_array_size_ * std::log(2) / expected_items);

        bit_array_size_ = std::max(bit_array_size_, size_t(1024));
        num_hashes_ = std::max(num_hashes_, size_t(2));
        num_hashes_ = std::min(num_hashes_, size_t(20));

        bits_.resize(bit_array_size_, false);
    }

    void add(const std::string& key) {
        size_t h1 = hash1(key);
        size_t h2 = hash2(key);

        for (size_t i = 0; i < num_hashes_; ++i) {
            size_t hash = (h1 + i * h2) % bit_array_size_;
            bits_[hash] = true;
        }
    }

    bool mightContain(const std::string& key) const {
        size_t h1 = hash1(key);
        size_t h2 = hash2(key);

        for (size_t i = 0; i < num_hashes_; ++i) {
            size_t hash = (h1 + i * h2) % bit_array_size_;
            if (!bits_[hash]) {
                return false;
            }
        }

        return true;
    }

private:
    size_t num_hashes_;
    size_t bit_array_size_;
    std::vector<bool> bits_;

    size_t hash1(const std::string& key) const {
        size_t hash = 14695981039346656037ULL;
        for (char c : key) {
            hash ^= static_cast<size_t>(c);
            hash *= 1099511628211ULL;
        }
        return hash;
    }

    size_t hash2(const std::string& key) const {
        size_t hash = 5381;
        for (char c : key) {
            hash = ((hash « 5) + hash) + static_cast<size_t>(c);
        }
        return hash;
    }
};

template <typename K, typename V>
class LRUCache {
public:
    explicit LRUCache(size_t capacity = 1000) : capacity_(capacity) {}

    // Delete copy constructor/assignment - mutex can't be copied
    LRUCache(const LRUCache&) = delete;
    LRUCache& operator=(const LRUCache&) = delete;

    // We don't need move operations as we'll initialize the cache in-place
    LRUCache(LRUCache&&) = delete;
    LRUCache& operator=(LRUCache&&) = delete;

    std::optional<V> get(const K& key) {
        std::lock_guard<std::mutex> lock(mutex_);

        auto it = cache_map_.find(key);
        if (it == cache_map_.end()) {
            return std::nullopt;
        }

        // Move to front (most recently used)
        cache_list_.splice(cache_list_.begin(), cache_list_, it->second);
        return it->second->second;
    }

    void put(const K& key, const V& value) {
        std::lock_guard<std::mutex> lock(mutex_);

        auto it = cache_map_.find(key);
        if (it != cache_map_.end()) {
            // Update existing item and move to front
            it->second->second = value;
            cache_list_.splice(cache_list_.begin(), cache_list_, it->second);
            return;
        }

        // Remove least recently used item if full
        if (cache_list_.size() >= capacity_) {
            auto last = cache_list_.back();
            cache_map_.erase(last.first);
            cache_list_.pop_back();
        }

        // Insert new item at front
        cache_list_.emplace_front(key, value);
        cache_map_[key] = cache_list_.begin();
    }

private:
    using ListItem = std::pair<K, V>;
    using ListIterator = typename std::list<ListItem>::iterator;

    std::list<ListItem> cache_list_;
    std::unordered_map<K, ListIterator> cache_map_;
    size_t capacity_;
    std::mutex mutex_;
};

class ThreadPool {
public:
    ThreadPool(size_t num_threads) : stop_(false) {
        for(size_t i = 0; i < num_threads; ++i) {
            workers_.emplace_back([this] {
                while(true) {
                    std::function<void()> task;
                    {
                        std::unique_lock<std::mutex> lock(queue_mutex_);
                        condition_.wait(lock, [this] {
                            return stop_ || !tasks_.empty();
                        });

                        if(stop_ && tasks_.empty()) {
                            return;
                        }

                        task = std::move(tasks_.front());
                        tasks_.pop();
                    }
                    task();
                }
            });
        }
    }

    void enqueue(std::function<void()> task) {
        {
            std::unique_lock<std::mutex> lock(queue_mutex_);
            if(stop_) {
                throw std::runtime_error("enqueue on stopped ThreadPool");
            }
            tasks_.emplace(std::move(task));
        }
        condition_.notify_one();
    }

    ~ThreadPool() {
        {
            std::unique_lock<std::mutex> lock(queue_mutex_);
            stop_ = true;
        }
        condition_.notify_all();
        for(std::thread &worker: workers_) {
            if(worker.joinable()) {
                worker.join();
            }
        }
    }

private:
    std::vector<std::thread> workers_;
    std::queue<std::function<void()» tasks_;
    std::mutex queue_mutex_;
    std::condition_variable condition_;
    std::atomic<bool> stop_;
};

class StorageEngine {
public:
    StorageEngine(size_t max_memory_size = 1024 * 1024 * 100,
                 size_t memtable_size = 1024 * 1024 * 10);

    ~StorageEngine();

    bool set(const std::string& key, const std::string& value);

    bool get(const std::string& key, std::string& value);

    bool del(const std::string& key);

    bool multiSet(const std::vector<std::pair<std::string, std::string»& kvs);

    bool multiGet(const std::vector<std::string>& keys, std::vector<std::pair<std::string, std::optional<std::string»>& results);

    size_t getMemoryUsage() const;

private:
    // Token bucket for flow control
    class TokenBucket {
    public:
        TokenBucket(size_t rate, size_t capacity)
            : tokens_(capacity), rate_(rate), capacity_(capacity),
              last_refill_(std::chrono::steady_clock::now()) {}

        bool consumeToken() {
            std::lock_guard<std::mutex> lock(mutex_);

            refillTokens();

            if (tokens_ == 0) {
                return false;
            }

            tokens_--;
            return true;
        }

        void refillTokens() {
            auto now = std::chrono::steady_clock::now();
            auto elapsed = std::chrono::duration_cast<std::chrono::seconds>(now - last_refill_).count();

            if (elapsed > 0) {
                tokens_ = std::min(capacity_, tokens_ + (elapsed * rate_));
                last_refill_ = now;
            }
        }

    private:
        size_t tokens_;
        size_t rate_;
        size_t capacity_;
        std::chrono::steady_clock::time_point last_refill_;
        std::mutex mutex_;
    };

    // Segmented bloom filter
    class SegmentedBloomFilter {
    public:
        SegmentedBloomFilter(size_t expected_items = 10000, size_t num_segments = 4) {
            size_t items_per_segment = (expected_items + num_segments - 1) / num_segments;

            for (size_t i = 0; i < num_segments; i++) {
                segments_.emplace_back(items_per_segment);
            }
        }

        void add(const std::string& key) {
            size_t segment = getSegment(key);
            segments_[segment].add(key);
        }

        bool mightContain(const std::string& key) const {
            size_t segment = getSegment(key);
            return segments_[segment].mightContain(key);
        }

    private:
        std::vector<BloomFilter> segments_;

        size_t getSegment(const std::string& key) const {
            if (segments_.empty()) return 0;

            uint32_t hash = 0;
            for (char c : key) {
                hash = hash * 31 + c;
            }

            return hash % segments_.size();
        }
    };

    // Fence pointers for SSTable indexing
    class FencePointers {
    public:
        std::vector<std::string> keys;
        std::vector<size_t> offsets;

        // Binary search to find position
        size_t findPosition(const std::string& key) const {
            if (keys.empty()) return 0;

            // Binary search to find the position
            auto it = std::upper_bound(keys.begin(), keys.end(), key);
            if (it == keys.begin()) return 0;

            size_t index = std::distance(keys.begin(), it) - 1;
            return offsets[index];
        }
    };

    // Key prefix optimization
    class KeyPrefix {
    public:
        std::string prefix;
        std::unordered_map<std::string, size_t> suffix_to_index;

        void addKey(const std::string& key, size_t& common_prefix_len) {
            if (prefix.empty()) {
                prefix = key;
                common_prefix_len = key.length();
                return;
            }

            // Find common prefix length
            size_t min_len = std::min(prefix.length(), key.length());
            common_prefix_len = 0;

            while (common_prefix_len < min_len && prefix[common_prefix_len] == key[common_prefix_len]) {
                common_prefix_len++;
            }

            // Update prefix
            if (common_prefix_len < prefix.length()) {
                prefix = prefix.substr(0, common_prefix_len);
            }

            // Add suffix
            std::string suffix = key.substr(common_prefix_len);
            suffix_to_index[suffix] = suffix_to_index.size();
        }

        std::string getSuffix(const std::string& key) const {
            if (prefix.empty() || key.length() < prefix.length()) {
                return key;
            }

            // Check if key starts with prefix
            for (size_t i = 0; i < prefix.length(); i++) {
```

```
            if (key[i] != prefix[i]) {
                return key;
            }
        }

        return key.substr(prefix.length());
    }
};

// Memtable shard
struct MemTableShard {
    std::map<std::string, std::string> data;
    size_t size = 0;
    std::chrono::steady_clock::time_point creation_time;
    mutable std::shared_mutex mutex;  // Changed to mutable to allow locking in const methods

    MemTableShard() : creation_time(std::chrono::steady_clock::now()) {}

    void insert(const std::string& key, const std::string& value) {
        std::unique_lock<std::shared_mutex> lock(mutex);

        size_t old_size = 0;
        auto it = data.find(key);
        if (it != data.end()) {
            old_size = key.size() + it->second.size() + sizeof(size_t) * 2;
        }

        data[key] = value;

        size_t new_size = key.size() + value.size() + sizeof(size_t) * 2;
        size += (new_size - old_size);
    }

    bool get(const std::string& key, std::string& value) const {
        std::shared_lock<std::shared_mutex> lock(mutex);  // Now works with mutable mutex

        auto it = data.find(key);
        if (it != data.end()) {
            if (it->second.empty()) {
                return false;
            }
            value = it->second;
            return true;
        }
        return false;
    }

    bool remove(const std::string& key) {
        std::unique_lock<std::shared_mutex> lock(mutex);

        auto it = data.find(key);
        if (it != data.end()) {
            size_t entry_size = key.size() + it->second.size() + sizeof(size_t) * 2;
            size -= entry_size;
            data.erase(it);
            return true;
        }
        return false;
    }

    size_t memoryUsage() const {
        std::shared_lock<std::shared_mutex> lock(mutex);  // Now works with mutable mutex
        return size;
    }

    bool empty() const {
        std::shared_lock<std::shared_mutex> lock(mutex);  // Now works with mutable mutex
        return data.empty();
    }
};

// Partitioned memtable
struct PartitionedMemTable {
    std::array<MemTableShard, NUM_SHARDS> shards;

    size_t getShard(const std::string& key) const {
        uint32_t hash = 0;
        for (char c : key) {
            hash = hash * 31 + c;
        }
        return hash % shards.size();
    }

    void insert(const std::string& key, const std::string& value) {
        size_t shard_idx = getShard(key);
        shards[shard_idx].insert(key, value);
    }
```

```
        bool get(const std::string& key, std::string& value) const {
            size_t shard_idx = getShard(key);
            return shards[shard_idx].get(key, value);
        }

        bool remove(const std::string& key) {
            size_t shard_idx = getShard(key);
            return shards[shard_idx].remove(key);
        }

        size_t memoryUsage() const {
            size_t total = 0;
            for (const auto& shard : shards) {
                total += shard.memoryUsage();
            }
            return total;
        }

        bool empty() const {
            for (const auto& shard : shards) {
                if (!shard.empty()) {
                    return false;
                }
            }
            return true;
        }
    };

    // SSTable with optimization
    struct SSTable {
        std::map<std::string, std::string> data;
        SegmentedBloomFilter bloom_filter;
        FencePointers fence_pointers;
        KeyPrefix key_prefix;
        size_t level = 0;
        std::string min_key;
        std::string max_key;
        mutable std::atomic<size_t> access_count = {0};  // Changed to mutable

        SSTable() = default;

        explicit SSTable(const std::map<std::string, std::string>& source_data, size_t expected_items = 10000)
            : bloom_filter(expected_items) {

            if (!source_data.empty()) {
                data = source_data;
                min_key = source_data.begin()->first;
                max_key = source_data.rbegin()->first;

                // Build fence pointers and optimize key prefixes
                buildFencePointers();

                // Add all keys to bloom filter
                for (const auto& [key, _] : source_data) {
                    bloom_filter.add(key);
                }
            }
        }

        bool get(const std::string& key, std::string& value) const {
            // Increment access count for this SSTable - now works with mutable atomic
            access_count++;

            // Use bloom filter for fast negative lookups
            if (!bloom_filter.mightContain(key)) {
                return false;
            }

            // Use key range for quick rejection
            if (!min_key.empty() && !max_key.empty()) {
                if (key < min_key || key > max_key) {
                    return false;
                }
            }

            // Actual lookup
            auto it = data.find(key);
            if (it == data.end()) {
                return false;
            }

            if (it->second.empty()) {
                // Tombstone value
                return false;
            }

            value = it->second;
            return true;
        }

        bool mightContain(const std::string& key) const {
            // Key range check for quick rejection
            if (!min_key.empty() && !max_key.empty()) {
                if (key < min_key || key > max_key) {
                    return false;
                }
            }

            return bloom_filter.mightContain(key);
        }

        size_t memoryUsage() const {
            size_t usage = 0;

            // Size of keys and values
            for (const auto& [key, value] : data) {
                usage += key.size() + value.size() + sizeof(key) + sizeof(value);
            }

            // Size of the map structure itself
            usage += sizeof(data) + (data.size() * (sizeof(std::map<std::string,
std::string>::node_type)));

            return usage;
        }

        void buildFencePointers() {
            if (data.empty()) return;

            const size_t FENCE_INTERVAL = 16; // Every 16 keys

            size_t count = 0;
            size_t offset = 0;

            for (const auto& [key, _] : data) {
                if (count % FENCE_INTERVAL == 0) {
                    fence_pointers.keys.push_back(key);
                    fence_pointers.offsets.push_back(offset);
                }
                count++;
                offset++;
            }
        }
    };

    // Core methods
    void writeLogEntry(const std::string& operation, const std::string& key, const std::string& value
= "");
    void flushMemTable(size_t shard_index = NUM_SHARDS);
    void compactLevel(size_t level);
    void checkAndScheduleCompaction();
    void recoverFromLog();
    void monitorAndAdjustCompaction();

    // Calculate shard index for a key
    size_t getShardIndex(const std::string& key) const;

    // Fast hash function for partitioning
    uint32_t murmurHash(const std::string& key) const {
        const uint32_t m = 0x5bd1e995;
        const int r = 24;
        uint32_t h = 0; // Seed

        // Mix 4 bytes at a time into the hash
        const unsigned char* data = (const unsigned char*)key.data();
        size_t len = key.size();

        while (len >= 4) {
            uint32_t k = *(uint32_t*)data;

            k *= m;
            k ^= k » r;
            k *= m;

            h *= m;
            h ^= k;

            data += 4;
            len -= 4;
        }

        // Handle the last few bytes
        switch (len) {
            case 3: h ^= data[2] « 16; // FALLTHROUGH
            case 2: h ^= data[1] « 8;  // FALLTHROUGH
            case 1: h ^= data[0];
                    h *= m;
        };

        // Do a few final mixes
        h ^= h » 13;
        h *= m;
        h ^= h » 15;

        return h;
    }

    // Data members
    std::unique_ptr<PartitionedMemTable> active_memtable_;
    std::vector<std::unique_ptr<PartitionedMemTable» immutable_memtables_;
    std::vector<std::vector<std::unique_ptr<SSTable»> sstable_levels_;

    // LRU cache for hot items with multiple segments for less contention
    std::array<LRUCache<std::string, std::string>, NUM_CACHES> read_caches_;

    // Thread pool for background operations
    ThreadPool thread_pool_;

    // Token bucket for limiting background operations
    TokenBucket token_bucket_;

    // Configuration parameters
    size_t max_memory_size_;
    size_t memtable_size_threshold_;
    std::atomic<size_t> current_memory_usage_ = {0};

    // Adaptive compaction parameters
    std::atomic<size_t> reads_since_compaction_ = {0};
    std::atomic<size_t> writes_since_compaction_ = {0};
    std::chrono::steady_clock::time_point last_compaction_time_;
    std::atomic<float> compaction_frequency_ = {1.0};

    std::string log_file_path_ = "wal.log";
    mutable std::mutex log_mutex_;

    std::atomic<bool> flush_in_progress_ = {false};
    std::atomic<bool> compaction_in_progress_ = {false};
    std::atomic<bool> shutdown_requested_ = {false};

    mutable std::shared_mutex rw_mutex_;
    std::mutex flush_mutex_;
    std::mutex compaction_mutex_;
};

#endif // STORAGE_ENGINE_H
```