

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:

BlinkServer	Implements a TCP server for BLINK DB	??
BloomFilter	Implements a bloom filter for efficient negative lookups	??
LRUCache< K, V >	Implements an LRU (Least Recently Used) cache	??
StorageEngine	Implements the LSM-based storage engine	??
ThreadPool	Implements a thread pool for background operations	??
resp::Value	Class representing a RESP-2 value	??

Chapter 2

File Index

2.1 File List

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

client.cpp	Simple client for the BLINK DB server	??
main.cpp	Entry point for the BLINK DB server application	??
resp.cpp	Implementation of Redis RESP-2 protocol	??
resp.h	Implementation of Redis RESP-2 protocol	??
server.cpp	Implementation of the BLINK DB server	??
server.h	Header file for the BLINK DB server	??
storage_engine.cpp	Implementation of the StorageEngine class	??
storage_engine.h	Header file for the LSM-based storage engine	??

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

<i>port</i>	Port to listen on.
<i>storage_engine</i>	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

<i>expected_items</i>	The expected number of items to store.
<i>false_positive_rate</i>	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

<i>key</i>	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

<i>key</i>	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

<i>K</i>	The type of the cache keys.
<i>V</i>	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

<i>capacity</i>	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

<i>key</i>	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

<i>key</i>	The key to insert.
<i>value</i>	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

<i>max_memory_size</i>	The maximum memory size allowed for the storage engine.
<i>memtable_size</i>	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

<i>key</i>	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

<i>key</i>	The key to retrieve.
<i>value</i>	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

<i>keys</i>	The keys to retrieve.
<i>results</i>	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

<i>kvs</i>	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

<i>key</i>	The key to insert or update.
<i>value</i>	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

<code>num_threads</code>	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

<i>task</i>	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

<i>values</i>	The array values.
---------------	-------------------

Returns

A new [Value](#) object representing an Array.

Parameters

<i>values</i>	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

<i>str</i>	The string value.
------------	-------------------

Returns

A new [Value](#) object representing a Bulk String.

Parameters

<i>str</i>	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

<i>message</i>	The error message.
----------------	--------------------

Returns

A new [Value](#) object representing an Error.

Parameters

<i>message</i>	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

<i>value</i>	The integer value.
--------------	--------------------

Returns

A new [Value](#) object representing an Integer.

Parameters

<i>value</i>	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

<i>str</i>	The string value.
------------	-------------------

Returns

A new [Value](#) object representing a Simple String.

Parameters

<i>str</i>	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

	<i>data</i>	The serialized RESP-2 message.
<i>out</i>	<i>consumed</i>	The number of bytes consumed during deserialization.

Returns

The deserialized [Value](#) object, or std::nullopt if deserialization fails.

Parameters

<i>data</i>	The serialized RESP string.
<i>consumed</i>	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

<code>argc</code>	The number of command-line arguments.
<code>argv</code>	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

<i>signal</i>	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

<code>SimpleString</code>	Simple string prefixed with '+'.
<code>Error</code>	Error prefixed with '-'.
<code>Integer</code>	Integer prefixed with ':'.
<code>BulkString</code>	Bulk string prefixed with '\$'.
<code>Array</code>	Array prefixed with '*'.

4.5 resp.h

[Go to the documentation of this file.](#)

```
00001
00009 #ifndef RESP_H
00010 #define RESP_H
00011
00012 #include <string>
00013 #include <vector>
00014 #include <optional>
00015
00016 namespace resp {
00017
00022 enum class Type {
00023     SimpleString,
```

```

00024     Error,
00025     Integer,
00026     BulkString,
00027     Array
00028 };
00029
00038 class Value {
00039 public:
00045     static Value createSimpleString(const std::string& str);
00046
00052     static Value createError(const std::string& message);
00053
00059     static Value createInteger(int64_t value);
00060
00066     static Value createBulkString(const std::string& str);
00067
00072     static Value createNullBulkString();
00073
00079     static Value createArray(const std::vector<Value>& values);
00080
00085     static Value createNullArray();
00086
00091     Type getType() const;
00092
00097     std::optional<std::string> getString() const;
00098
00103     std::optional<int64_t> getInteger() const;
00104
00109     std::optional<std::vector<Value>> getArray() const;
00110
00115     bool isNull() const;
00116
00121     std::string serialize() const;
00122
00129     static std::optional<Value> deserialize(const std::string& data, size_t& consumed);
00130
00131 private:
00132     Type type_;
00133     bool null_ = false;
00134     std::string string_value_;
00135     int64_t integer_value_ = 0;
00136     std::vector<Value> array_values_;
00137
00142     explicit Value(Type type);
00143 };
00144
00145 } // namespace resp
00146
00147 #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.](#)

```
00001
00009 #ifndef SERVER_H
00010 #define SERVER_H
00011
00012 #include "storage_engine.h"
00013 #include "resp.h"
00014 #include <string>
00015 #include <unordered_map>
00016 #include <vector>
00017 #include <atomic>
00018 #include <thread>
00019 #include <functional>
00020
00028 class BlinkServer {
00029 public:
00035     BlinkServer(int port, StorageEngine& storage_engine);
00036
00040     ~BlinkServer();
00041
00046     bool start();
00047
00051     void stop();
00052
00057     bool isRunning() const;
```

```

00058
00059 private:
00064     struct ClientConnection {
00065         int fd;
00066         std::string buffer;
00067         std::string output_buffer;
00068     };
00069
00074     bool initEpoll();
00075
00080     bool startListening();
00081
00085     void acceptClient();
00086
00091     void handleClient(int client_fd);
00092
00098     resp::Value processCommand(const resp::Value& command);
00099
00105     void sendResponse(int client_fd, const resp::Value& response);
00106
00111     void closeClient(int client_fd);
00112
00116     void serverLoop();
00117
00118     int port_;
00119     StorageEngine& storage_engine_;
00120     int server_fd_ = -1;
00121     int epoll_fd_ = -1;
00122     std::atomic<bool> running_ = {false};
00123     std::thread server_thread_;
00124     std::unordered_map<int, ClientConnection> clients_;
00125 };
00126
00127 #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.](#)

```
00001
00009 #ifndef STORAGE_ENGINE_H
00010 #define STORAGE_ENGINE_H
00011
00012 #include <string>
00013 #include <map>
00014 #include <vector>
00015 #include <memory>
00016 #include <mutex>
00017 #include <shared_mutex>
00018 #include <atomic>
00019 #include <chrono>
```



```

00020 #include <array>
00021 #include <deque>
00022 #include <functional>
00023 #include <optional>
00024 #include <unordered_map>
00025 #include <list>
00026 #include <thread>
00027 #include <condition_variable>
00028 #include <queue>
00029 #include <stdexcept>
00030 #include <cmath>
00031
00033 #define NUM_SHARDS 16
00034
00036 #define NUM_CACHES 4
00037
00042 class BloomFilter {
00043 public:
00049     BloomFilter(size_t expected_items = 10000, double false_positive_rate = 0.01) {
00050         bit_array_size_ = static_cast<size_t>(-expected_items * std::log(false_positive_rate) /
00051 (std::log(2) * std::log(2)));
00052         num_hashes_ = static_cast<size_t>(bit_array_size_ * std::log(2) / expected_items);
00053
00054         bit_array_size_ = std::max(bit_array_size_, size_t(1024));
00055         num_hashes_ = std::max(num_hashes_, size_t(2));
00056         num_hashes_ = std::min(num_hashes_, size_t(20));
00057
00058         bits_.resize(bit_array_size_, false);
00059     }
00064
00065     void add(const std::string& key) {
00066         size_t h1 = hash1(key);
00067         size_t h2 = hash2(key);
00068
00069         for (size_t i = 0; i < num_hashes_; ++i) {
00070             size_t hash = (h1 + i * h2) % bit_array_size_;
00071             bits_[hash] = true;
00072         }
00073
00079     bool mightContain(const std::string& key) const {
00080         size_t h1 = hash1(key);
00081         size_t h2 = hash2(key);
00082
00083         for (size_t i = 0; i < num_hashes_; ++i) {
00084             size_t hash = (h1 + i * h2) % bit_array_size_;
00085             if (!bits_[hash]) {
00086                 return false;
00087             }
00088         }
00089
00090         return true;
00091     }
00092
00093 private:
00094     size_t num_hashes_;
00095     size_t bit_array_size_;
00096     std::vector<bool> bits_;
00097
00098     size_t hash1(const std::string& key) const {
00099         size_t hash = 14695981039346656037ULL;
00100         for (char c : key) {
00101             hash ^= static_cast<size_t>(c);
00102             hash *= 1099511628211ULL;
00103         }
00104         return hash;
00105     }
00106
00107     size_t hash2(const std::string& key) const {
00108         size_t hash = 5381;
00109         for (char c : key) {
00110             hash = ((hash << 5) + hash) + static_cast<size_t>(c);
00111         }
00112         return hash;
00113     }
00114 };
00115
00122 template <typename K, typename V>
00123 class LRUCache {
00124 public:
00129     explicit LRUCache(size_t capacity = 1000) : capacity_(capacity) {}
00130
00131     // Delete copy constructor/assignment - mutex can't be copied
00132     LRUCache(const LRUCache&) = delete;
00133     LRUCache& operator=(const LRUCache&) = delete;
00134
00135     // We don't need move operations as we'll initialize the cache in-place

```

```

00136     LRUCache(LRUCache&&) = delete;
00137     LRUCache& operator=(LRUCache&&) = delete;
00138
00144     std::optional<V> get(const K& key) {
00145         std::lock_guard<std::mutex> lock(mutex_);
00146
00147         auto it = cache_map_.find(key);
00148         if (it == cache_map_.end()) {
00149             return std::nullopt;
00150         }
00151
00152         // Move to front (most recently used)
00153         cache_list_.splice(cache_list_.begin(), cache_list_, it->second);
00154         return it->second->second;
00155     }
00156
00162     void put(const K& key, const V& value) {
00163         std::lock_guard<std::mutex> lock(mutex_);
00164
00165         auto it = cache_map_.find(key);
00166         if (it != cache_map_.end()) {
00167             // Update existing item and move to front
00168             it->second->second = value;
00169             cache_list_.splice(cache_list_.begin(), cache_list_, it->second);
00170             return;
00171         }
00172
00173         // Remove least recently used item if full
00174         if (cache_list_.size() >= capacity_) {
00175             auto last = cache_list_.back();
00176             cache_map_.erase(last.first);
00177             cache_list_.pop_back();
00178         }
00179
00180         // Insert new item at front
00181         cache_list_.emplace_front(key, value);
00182         cache_map_[key] = cache_list_.begin();
00183     }
00184
00185 private:
00186     using ListItem = std::pair<K, V>;
00187     using ListIterator = typename std::list<ListItem>::iterator;
00188
00189     std::list<ListItem> cache_list_;
00190     std::unordered_map<K, ListIterator> cache_map_;
00191     size_t capacity_;
00192     std::mutex mutex_;
00193 };
00194
00199 class ThreadPool {
00200 public:
00205     ThreadPool(size_t num_threads) : stop_(false) {
00206         for (size_t i = 0; i < num_threads; ++i) {
00207             workers_.emplace_back([this] {
00208                 while (true) {
00209                     std::function<void()> task;
00210                     {
00211                         std::unique_lock<std::mutex> lock(queue_mutex_);
00212                         condition_.wait(lock, [this] {
00213                             return stop_ || !tasks_.empty();
00214                         });
00215
00216                         if (stop_ && tasks_.empty()) {
00217                             return;
00218                         }
00219
00220                         task = std::move(tasks_.front());
00221                         tasks_.pop();
00222                     }
00223                     task();
00224                 }
00225             });
00226         }
00227     }
00228
00233     void enqueue(std::function<void()> task) {
00234         {
00235             std::unique_lock<std::mutex> lock(queue_mutex_);
00236             if (stop_) {
00237                 throw std::runtime_error("enqueue on stopped ThreadPool");
00238             }
00239             tasks_.emplace(std::move(task));
00240         }
00241         condition_.notify_one();
00242     }
00243
00247     ~ThreadPool() {

```

```

00248     {
00249         std::unique_lock<std::mutex> lock(queue_mutex_);
00250         stop_ = true;
00251     }
00252     condition_.notify_all();
00253     for(std::thread &worker: workers_) {
00254         if(worker.joinable()) {
00255             worker.join();
00256         }
00257     }
00258 }
00259
00260 private:
00261     std::vector<std::thread> workers_;
00262     std::queue<std::function<void()>> tasks_;
00263     std::mutex queue_mutex_;
00264     std::condition_variable condition_;
00265     std::atomic<bool> stop_;
00266 };
00267
00272 class StorageEngine {
00273 public:
00274     StorageEngine(size_t max_memory_size = 1024 * 1024 * 100,
00275                   size_t memtable_size = 1024 * 1024 * 10);
00276
00285     ~StorageEngine();
00286
00293     bool set(const std::string& key, const std::string& value);
00294
00301     bool get(const std::string& key, std::string& value);
00302
00308     bool del(const std::string& key);
00309
00315     bool multiSet(const std::vector<std::pair<std::string, std::string>& kvs);
00316
00323     bool multiGet(const std::vector<std::string>& keys, std::vector<std::pair<std::string,
std::optional<std::string>>& results);
00324
00329     size_t getMemoryUsage() const;
00330
00331 private:
00332     // Token bucket for flow control
00333     class TokenBucket {
00334     public:
00335         TokenBucket(size_t rate, size_t capacity)
00336             : tokens_(capacity), rate_(rate), capacity_(capacity),
00337               last_refill_(std::chrono::steady_clock::now()) {}
00338
00339         bool consumeToken() {
00340             std::lock_guard<std::mutex> lock(mutex_);
00341
00342             refillTokens();
00343
00344             if (tokens_ == 0) {
00345                 return false;
00346             }
00347
00348             tokens_--;
00349             return true;
00350         }
00351
00352         void refillTokens() {
00353             auto now = std::chrono::steady_clock::now();
00354             auto elapsed = std::chrono::duration_cast<std::chrono::seconds>(now -
last_refill_).count();
00355
00356             if (elapsed > 0) {
00357                 tokens_ = std::min(capacity_, tokens_ + (elapsed * rate_));
00358                 last_refill_ = now;
00359             }
00360         }
00361
00362     private:
00363         size_t tokens_;
00364         size_t rate_;
00365         size_t capacity_;
00366         std::chrono::steady_clock::time_point last_refill_;
00367         std::mutex mutex_;
00368     };
00369
00370     // Segmented bloom filter
00371     class SegmentedBloomFilter {
00372     public:
00373         SegmentedBloomFilter(size_t expected_items = 10000, size_t num_segments = 4) {
00374             size_t items_per_segment = (expected_items + num_segments - 1) / num_segments;
00375
00376             for (size_t i = 0; i < num_segments; i++) {

```

```

00377         segments_.emplace_back(items_per_segment);
00378     }
00379 }
00380
00381 void add(const std::string& key) {
00382     size_t segment = getSegment(key);
00383     segments_[segment].add(key);
00384 }
00385
00386 bool mightContain(const std::string& key) const {
00387     size_t segment = getSegment(key);
00388     return segments_[segment].mightContain(key);
00389 }
00390
00391 private:
00392     std::vector<BloomFilter> segments_;
00393
00394     size_t getSegment(const std::string& key) const {
00395         if (segments_.empty()) return 0;
00396
00397         uint32_t hash = 0;
00398         for (char c : key) {
00399             hash = hash * 31 + c;
00400         }
00401
00402         return hash % segments_.size();
00403     }
00404 };
00405
00406 // Fence pointers for SSTable indexing
00407 class FencePointers {
00408 public:
00409     std::vector<std::string> keys;
00410     std::vector<size_t> offsets;
00411
00412     // Binary search to find position
00413     size_t findPosition(const std::string& key) const {
00414         if (keys.empty()) return 0;
00415
00416         // Binary search to find the position
00417         auto it = std::upper_bound(keys.begin(), keys.end(), key);
00418         if (it == keys.begin()) return 0;
00419
00420         size_t index = std::distance(keys.begin(), it) - 1;
00421         return offsets[index];
00422     }
00423 };
00424
00425 // Key prefix optimization
00426 class KeyPrefix {
00427 public:
00428     std::string prefix;
00429     std::unordered_map<std::string, size_t> suffix_to_index;
00430
00431     void addKey(const std::string& key, size_t& common_prefix_len) {
00432         if (prefix.empty()) {
00433             prefix = key;
00434             common_prefix_len = key.length();
00435             return;
00436         }
00437
00438         // Find common prefix length
00439         size_t min_len = std::min(prefix.length(), key.length());
00440         common_prefix_len = 0;
00441
00442         while (common_prefix_len < min_len && prefix[common_prefix_len] == key[common_prefix_len])
00443         {
00444             common_prefix_len++;
00445         }
00446
00447         // Update prefix
00448         if (common_prefix_len < prefix.length()) {
00449             prefix = prefix.substr(0, common_prefix_len);
00450         }
00451
00452         // Add suffix
00453         std::string suffix = key.substr(common_prefix_len);
00454         suffix_to_index[suffix] = suffix_to_index.size();
00455     }
00456
00457     std::string getSuffix(const std::string& key) const {
00458         if (prefix.empty() || key.length() < prefix.length()) {
00459             return key;
00460         }
00461
00462         // Check if key starts with prefix
00463         for (size_t i = 0; i < prefix.length(); i++) {

```

```

00463         if (key[i] != prefix[i]) {
00464             return key;
00465         }
00466     }
00467
00468     return key.substr(prefix.length());
00469 }
00470 };
00471
00472 // Memtable shard
00473 struct MemTableShard {
00474     std::map<std::string, std::string> data;
00475     size_t size = 0;
00476     std::chrono::steady_clock::time_point creation_time;
00477     mutable std::shared_mutex mutex; // Changed to mutable to allow locking in const methods
00478
00479     MemTableShard() : creation_time(std::chrono::steady_clock::now()) {}
00480
00481     void insert(const std::string& key, const std::string& value) {
00482         std::unique_lock<std::shared_mutex> lock(mutex);
00483
00484         size_t old_size = 0;
00485         auto it = data.find(key);
00486         if (it != data.end()) {
00487             old_size = key.size() + it->second.size() + sizeof(size_t) * 2;
00488         }
00489
00490         data[key] = value;
00491
00492         size_t new_size = key.size() + value.size() + sizeof(size_t) * 2;
00493         size += (new_size - old_size);
00494     }
00495
00496     bool get(const std::string& key, std::string& value) const {
00497         std::shared_lock<std::shared_mutex> lock(mutex); // Now works with mutable mutex
00498
00499         auto it = data.find(key);
00500         if (it != data.end()) {
00501             if (it->second.empty()) {
00502                 return false;
00503             }
00504             value = it->second;
00505             return true;
00506         }
00507         return false;
00508     }
00509
00510     bool remove(const std::string& key) {
00511         std::unique_lock<std::shared_mutex> lock(mutex);
00512
00513         auto it = data.find(key);
00514         if (it != data.end()) {
00515             size_t entry_size = key.size() + it->second.size() + sizeof(size_t) * 2;
00516             size -= entry_size;
00517             data.erase(it);
00518             return true;
00519         }
00520         return false;
00521     }
00522
00523     size_t memoryUsage() const {
00524         std::shared_lock<std::shared_mutex> lock(mutex); // Now works with mutable mutex
00525         return size;
00526     }
00527
00528     bool empty() const {
00529         std::shared_lock<std::shared_mutex> lock(mutex); // Now works with mutable mutex
00530         return data.empty();
00531     }
00532 };
00533
00534 // Partitioned memtable
00535 struct PartitionedMemTable {
00536     std::array<MemTableShard, NUM_SHARDS> shards;
00537
00538     size_t getShard(const std::string& key) const {
00539         uint32_t hash = 0;
00540         for (char c : key) {
00541             hash = hash * 31 + c;
00542         }
00543         return hash % shards.size();
00544     }
00545
00546     void insert(const std::string& key, const std::string& value) {
00547         size_t shard_idx = getShard(key);
00548         shards[shard_idx].insert(key, value);
00549     }

```

```

00550
00551     bool get(const std::string& key, std::string& value) const {
00552         size_t shard_idx = getShard(key);
00553         return shards[shard_idx].get(key, value);
00554     }
00555
00556     bool remove(const std::string& key) {
00557         size_t shard_idx = getShard(key);
00558         return shards[shard_idx].remove(key);
00559     }
00560
00561     size_t memoryUsage() const {
00562         size_t total = 0;
00563         for (const auto& shard : shards) {
00564             total += shard.memoryUsage();
00565         }
00566         return total;
00567     }
00568
00569     bool empty() const {
00570         for (const auto& shard : shards) {
00571             if (!shard.empty()) {
00572                 return false;
00573             }
00574         }
00575         return true;
00576     }
00577 };
00578
00579 // SSTable with optimization
00580 struct SSTable {
00581     std::map<std::string, std::string> data;
00582     SegmentedBloomFilter bloom_filter;
00583     FencePointers fence_pointers;
00584     KeyPrefix key_prefix;
00585     size_t level = 0;
00586     std::string min_key;
00587     std::string max_key;
00588     mutable std::atomic<size_t> access_count = {0}; // Changed to mutable
00589
00590     SSTable() = default;
00591
00592     explicit SSTable(const std::map<std::string, std::string>& source_data, size_t expected_items
= 10000)
00593         : bloom_filter(expected_items) {
00594
00595         if (!source_data.empty()) {
00596             data = source_data;
00597             min_key = source_data.begin()->first;
00598             max_key = source_data.rbegin()->first;
00599
00600             // Build fence pointers and optimize key prefixes
00601             buildFencePointers();
00602
00603             // Add all keys to bloom filter
00604             for (const auto& [key, _] : source_data) {
00605                 bloom_filter.add(key);
00606             }
00607         }
00608     }
00609
00610     bool get(const std::string& key, std::string& value) const {
00611         // Increment access count for this SSTable - now works with mutable atomic
00612         access_count++;
00613
00614         // Use bloom filter for fast negative lookups
00615         if (!bloom_filter.mightContain(key)) {
00616             return false;
00617         }
00618
00619         // Use key range for quick rejection
00620         if (!min_key.empty() && !max_key.empty()) {
00621             if (key < min_key || key > max_key) {
00622                 return false;
00623             }
00624         }
00625
00626         // Actual lookup
00627         auto it = data.find(key);
00628         if (it == data.end()) {
00629             return false;
00630         }
00631
00632         if (it->second.empty()) {
00633             // Tombstone value
00634             return false;
00635         }

```

```

00636         value = it->second;
00637         return true;
00638     }
00639 }
00640
00641 bool mightContain(const std::string& key) const {
00642     // Key range check for quick rejection
00643     if (!min_key.empty() && !max_key.empty()) {
00644         if (key < min_key || key > max_key) {
00645             return false;
00646         }
00647     }
00648     return bloom_filter.mightContain(key);
00649 }
00650
00651 size_t memoryUsage() const {
00652     size_t usage = 0;
00653
00654     // Size of keys and values
00655     for (const auto& [key, value] : data) {
00656         usage += key.size() + value.size() + sizeof(key) + sizeof(value);
00657     }
00658
00659     // Size of the map structure itself
00660     usage += sizeof(data) + (data.size() * (sizeof(std::map<std::string,
00661 std::string>::node_type)));
00662
00663     return usage;
00664 }
00665
00666 void buildFencePointers() {
00667     if (data.empty()) return;
00668
00669     const size_t FENCE_INTERVAL = 16; // Every 16 keys
00670
00671     size_t count = 0;
00672     size_t offset = 0;
00673
00674     for (const auto& [key, _] : data) {
00675         if (count % FENCE_INTERVAL == 0) {
00676             fence_pointers.keys.push_back(key);
00677             fence_pointers.offsets.push_back(offset);
00678         }
00679         count++;
00680         offset++;
00681     }
00682 }
00683 };
00684
00685 // Core methods
00686 void writeLogEntry(const std::string& operation, const std::string& key, const std::string& value
= "");
00687 void flushMemTable(size_t shard_index = NUM_SHARDS);
00688 void compactLevel(size_t level);
00689 void checkAndScheduleCompaction();
00690 void recoverFromLog();
00691 void monitorAndAdjustCompaction();
00692
00693 // Calculate shard index for a key
00694 size_t getShardIndex(const std::string& key) const;
00695
00696 // Fast hash function for partitioning
00697 uint32_t murmurHash(const std::string& key) const {
00698     const uint32_t m = 0x5bd1e995;
00699     const int r = 24;
00700     uint32_t h = 0; // Seed
00701
00702     // Mix 4 bytes at a time into the hash
00703     const unsigned char* data = (const unsigned char*)key.data();
00704     size_t len = key.size();
00705
00706     while (len >= 4) {
00707         uint32_t k = *(uint32_t*)data;
00708
00709         k *= m;
00710         k ^= k >> r;
00711         k *= m;
00712
00713         h *= m;
00714         h ^= k;
00715
00716         data += 4;
00717         len -= 4;
00718     }
00719
00720     // Handle the last few bytes

```

```

00721         switch (len) {
00722             case 3: h ^= data[2] << 16; // FALLTHROUGH
00723             case 2: h ^= data[1] << 8;  // FALLTHROUGH
00724             case 1: h ^= data[0];
00725                     h *= m;
00726         };
00727
00728         // Do a few final mixes
00729         h ^= h >> 13;
00730         h *= m;
00731         h ^= h >> 15;
00732
00733         return h;
00734     }
00735
00736     // Data members
00737     std::unique_ptr<PartitionedMemTable> active_memtable_;
00738     std::vector<std::unique_ptr<PartitionedMemTable>> immutable_memtables_;
00739     std::vector<std::vector<std::unique_ptr<SSTable>>> sstable_levels_;
00740
00741     // LRU cache for hot items with multiple segments for less contention
00742     std::array<LRUCache<std::string, std::string>, NUM_CACHES> read_caches_;
00743
00744     // Thread pool for background operations
00745     ThreadPool thread_pool_;
00746
00747     // Token bucket for limiting background operations
00748     TokenBucket token_bucket_;
00749
00750     // Configuration parameters
00751     size_t max_memory_size_;
00752     size_t memtable_size_threshold_;
00753     std::atomic<size_t> current_memory_usage_ = {0};
00754
00755     // Adaptive compaction parameters
00756     std::atomic<size_t> reads_since_compaction_ = {0};
00757     std::atomic<size_t> writes_since_compaction_ = {0};
00758     std::chrono::steady_clock::time_point last_compaction_time_;
00759     std::atomic<float> compaction_frequency_ = {1.0};
00760
00761     std::string log_file_path_ = "wal.log";
00762     mutable std::mutex log_mutex_;
00763
00764     std::atomic<bool> flush_in_progress_ = {false};
00765     std::atomic<bool> compaction_in_progress_ = {false};
00766     std::atomic<bool> shutdown_requested_ = {false};
00767
00768     mutable std::shared_mutex rw_mutex_;
00769     std::mutex flush_mutex_;
00770     std::mutex compaction_mutex_;
00771 };
00772
00773 #endif // STORAGE_ENGINE_H

```