BlinkDB - Part A

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 StorageEngine Class Reference

A storage engine implementation using an LSM Tree with LRU caching.

`#include <storage_engine.h>`

Public Member Functions

- StorageEngine (const string &db_dir="blinkdb_data")
  Constructor for StorageEngine.
- ~StorageEngine ()
  Destructor for StorageEngine.
- bool set (const char *key, const char *value)
  Set a key-value pair in the storage engine.
- const char * get (const char *key)
  Get the value associated with a key.
- bool del (const char *key)
  Delete a key-value pair from the storage engine.
- void sync ()
  Synchronize the in-memory data with disk.
- void debug_print_tree () const
  Print debug information about the LSM tree structure.

3.1.1 Detailed Description

A storage engine implementation using an LSM Tree with LRU caching.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 StorageEngine()

```
StorageEngine::StorageEngine (
            const string & db_dir = "blinkdb_data" )
```

Constructor for StorageEngine.

Constructs a StorageEngine instance.

Parameters

db_dir	Directory for database files
db_dir	Directory for the database files

3.1.2.2 ~StorageEngine()

```
StorageEngine::~StorageEngine ( )
```

Destructor for StorageEngine.

Destructor - cleans up resources and ensures data is saved.

3.1.3 Member Function Documentation

3.1.3.1 debug_print_tree()

```
void StorageEngine::debug_print_tree ( ) const
```

Print debug information about the LSM tree structure.

Prints the LSM Tree structure for debugging purposes.

3.1.3.2 del()

```
bool StorageEngine::del (
            const char * key )
```

Delete a key-value pair from the storage engine.

Deletes a key-value pair from the storage engine.

Parameters

key	The key to delete

Returns

True if the operation was successful, false otherwise

Parameters

key	The key to delete

Returns

true if operation succeeded

3.1.3.3 get()

```
const char * StorageEngine::get (
            const char * key )
```

Get the value associated with a key.

Retrieves the value associated with a key.

Parameters

key	The key to retrieve

Returns

The value associated with the key, or nullptr if not found

Parameters

key	The key to look up

Returns

The value as a C string, or nullptr if key not found

3.1.3.4 set()

```
bool StorageEngine::set (
            const char * key,
            const char * value )
```

Set a key-value pair in the storage engine.

Sets a key-value pair in the storage engine.

Parameters

key	The key to set
value	The value to associate with the key

Returns

True if the operation was successful, false otherwise

Parameters

key	The key to set
value	The value to associate with the key

Returns

true if operation succeeded

3.1.3.5 sync()

```
void StorageEngine::sync ( )
```

Synchronize the in-memory data with disk.

Forces synchronization of data to disk.

This method ensures that all pending operations are applied to the storage system and saved to disk immediately.

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

- storage_engine.h
- storage_engine.cpp

Chapter 4

File Documentation

4.1 main.cpp File Reference

Entry point for the BLINK DB application.

```
#include "storage_engine.h"
#include <bits/stdc++.h>
```
Include dependency graph for main.cpp:

4.2 storage_engine.cpp File Reference

Implementation of the LSM Tree based key-value storage engine.

```
#include "storage_engine.h"
#include <bits/stdc++.h>
#include <filesystem>
#include <sys/stat.h>
```
Include dependency graph for storage_engine.cpp:

4.2.1 Detailed Description

Implementation of the LSM Tree based key-value storage engine.

This file contains the implementation of the StorageEngine class, which provides a persistent key-value storage system using an LSM (Log-Structured Merge) Tree. The engine features:

- LRU caching for frequently accessed keys
- In-memory MemTable for recent writes
- Disk-based SSTables organized in levels
- Background compaction to maintain performance
- Persistence via sorted disk files

4.3 storage_engine.h

```
00001 // storage_engine.h
00002 #ifndef STORAGE_ENGINE_H
00003 #define STORAGE_ENGINE_H
00004
00005 #include <bits/stdc++.h>
00006 using namespace std;
00007
00012 class StorageEngine {
00013 private:
00014     // Forward declarations
00015     struct MemTable;
00016     struct SSTable;
00017
00022     struct KeyValue {
00023         string key;
00024         string value;
00025         uint64_t timestamp;
00026         bool is_deleted;
00027
00031         KeyValue() : timestamp(0), is_deleted(false) {}
00032
00040         KeyValue(const string& k, const string& v, uint64_t ts = 0, bool deleted = false)
00041             : key(k), value(v), timestamp(ts), is_deleted(deleted) {}
00042
00046         bool operator<(const KeyValue& other) const {
00047             return key < other.key;
00048         }
00049     };
00050
00055     struct MemTable {
00056         vector<KeyValue> entries;
00057         size_t size_bytes;
00058
00062         MemTable() : size_bytes(0) {}
00063
00069         size_t put(const KeyValue& kv) {
00070             size_t entry_size = kv.key.size() + kv.value.size() + sizeof(uint64_t) + sizeof(bool);
00071
00072             // Check if key already exists to avoid double-counting
00073             for (size_t i = 0; i < entries.size(); i++) {
00074                 if (entries[i].key == kv.key) {
00075                     // Subtract the size of the existing entry
00076                     size_bytes -= entries[i].key.size() + entries[i].value.size() +
00077                                   sizeof(uint64_t) + sizeof(bool);
00078                     // Replace the entry
00079                     entries[i] = kv;
00080                     size_bytes += entry_size;
00081                     return entry_size;
00082                 }
00083             }
00084
00085             // Key doesn't exist, add it
00086             entries.push_back(kv);
00087             size_bytes += entry_size;
00088
00089             // Keep entries sorted by key
00090             sort(entries.begin(), entries.end(),
00091                  [](const KeyValue& a, const KeyValue& b) { return a.key < b.key; });
00092
00093             return entry_size;
00094         }
00095
00101         const KeyValue* get(const string& key) const {
00102             // Binary search since entries are sorted
00103             auto it = lower_bound(entries.begin(), entries.end(), key,
00104                                   [](const KeyValue& kv, const string& k) { return kv.key < k; });
00105
00106             if (it != entries.end() && it->key == key) {
00107                 return &(*it);
00108             }
00109             return nullptr;
00110         }
00111
00116         size_t size() const {
00117             return entries.size();
00118         }
00119
00124         bool empty() const {
00125             return entries.empty();
00126         }
00127     };
00128
00133     struct SSTable {
00134         string file_path;
00135         size_t level;
```

```
00136         map<string, size_t> index;
00137         string min_key;
00138         string max_key;
00139 
00145         SSTable(const string& path, size_t lvl)
00146             : file_path(path), level(lvl) {}
00147     };
00148 
00153     struct CacheEntry {
00154         string key;
00155         string value;
00156         time_t timestamp;
00157 
00163         CacheEntry(const string& k, const string& v)
00164             : key(k), value(v), timestamp(time(nullptr)) {}
00165     };
00166 
00167     // LSM Tree properties
00168     MemTable* active_memtable;
00169     MemTable* immutable_memtable;
00170     vector<vector<SSTable*>> levels;
00171     size_t level_count;
00172     uint64_t next_timestamp;
00173     atomic<bool> compaction_running;
00174     string db_directory;
00175 
00176     // Cache properties
00177     static const size_t CACHE_SIZE = 1024;
00178     list<CacheEntry> cache_list;
00179     unordered_map<string, list<CacheEntry>::iterator> cache_map;
00180 
00181     // Configuration
00182     static const size_t MEMTABLE_MAX_SIZE = 4 * 1024 * 1024;
00183     static const size_t LEVEL_SIZE_RATIO = 10;
00184 
00185     // Multithreading support
00186     mutex memtable_mutex;
00187     mutex compaction_mutex;
00188     mutex cache_mutex;
00189     thread compaction_thread;
00190 
00191     // Cache operations
00192     void update_cache(const string& key, const string& value);
00193     const string* get_from_cache(const string& key);
00194     void evict_cache_if_needed();
00195 
00196     // LSM Tree operations
00197     void flush_memtable();
00198     void maybe_compact(size_t level);
00199     void compact_level(size_t level);
00200     void merge_sstables(const vector<SSTable*>& input_tables, SSTable* output_table);
00201     SSTable* create_sstable_from_memtable(MemTable* memtable, size_t level);
00202     const KeyValue* get_from_sstable(const SSTable* sstable, const string& key);
00203     void load_sstables();
00204     void start_background_compaction();
00205     void compaction_worker();
00206     uint64_t get_timestamp();
00207 
00208     // File operations
00209     void write_sstable_index(const string& path, const map<string, size_t>& index);
00210     bool read_sstable_index(const string& path, map<string, size_t>& index, string& min_key, string&
      max_key);
00211     string get_sstable_path(size_t level, size_t table_id);
00212 
00213 public:
00218     StorageEngine(const string& db_dir = "blinkdb_data");
00219 
00223     ~StorageEngine();
00224 
00231     bool set(const char* key, const char* value);
00232 
00238     const char* get(const char* key);
00239 
00245     bool del(const char* key);
00246 
00250     void sync();
00251 
00255     void debug_print_tree() const;
00256 };
00257 
00258 #endif
```