Final Report

Team members: Hantian Liu, Junchen Liu, Joseph Mitchell
Project: Implementation of an LSM-Tree-based key-value store

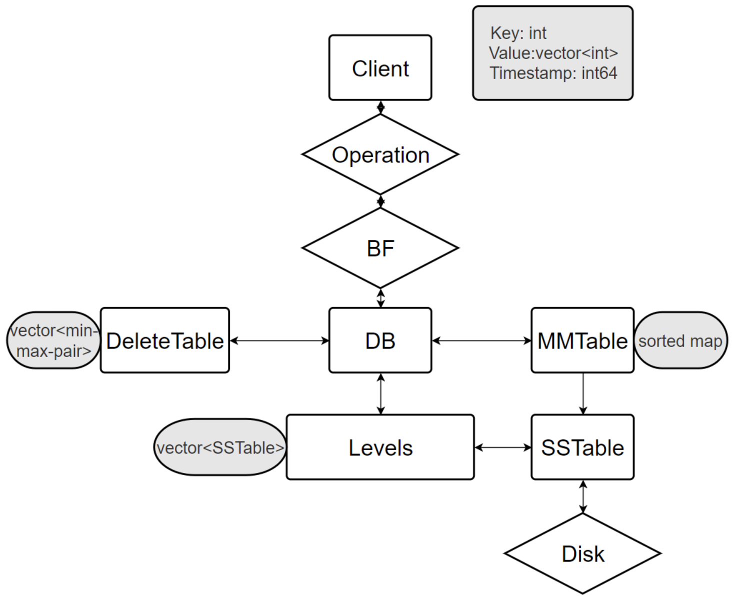
Project Progress

We implemented the main sections in the template database project, including the APIs such as PUT, GET, (range)DELETE. (range)Scan, as well as the basic functions of a database such as Bloomfilter, persistency, and tuning parameters.
We use std::map as the component in memory and SSTable as the data structure on disk. In general, we maintain the c1 component as DiskStorage in memory with some metadata of the SSTables. The c1 component has 4 levels, each level is 4 times the size of the previous one and level 0 is 4 times the size of the c0 component.
After the mid-term, we implemented a leveling compact policy and added a Bloomfilter level. Also, we designed some basic unit tests and passed the basic&persistent test in the template. With regards to work to be completed after the midterm, further optimizations will be made. For example, we designed a delete table based on the timestamp of every range delete intervals. Another improvement is on the project structure, we delete the SearchResult class and use the Value class instead.

Further Adjustments or Improvements

1. Adjusted our implementation of tiering strategies. When tiring, we first extract and merge sort the current level(***extract*** and ***compact method***) and append to the next level(***merge method***).
2. Improved ***compact method***, using a merge sort algorithm and a heap data structure.
3. Add Bloomfilters for each level and persistent them when db.close() is called.

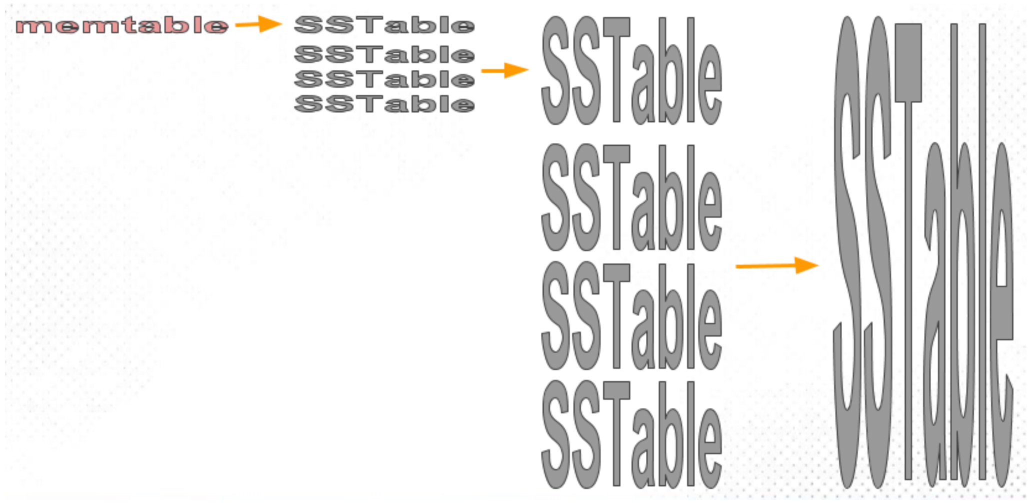
System Architecture



We use a map that is sorted by key as the MemTable. When you want to put/get/del a k-v pair, you first check the map(in memory), if not hit, then use the operations in DiskStorage Class(see the next section for detail).
When getting a value by a key, our database first asks the bloomfilter and then fetches it from the mem or disk. After getting the value, the database will compare its timestamp with the one in DeleteTable before outputting.
DiskStorage is responsible for managing the levels including the CRUD operation and merging the SSTable. DiskStorage keeps a vector of Level objects. The level with the larger index in the vector represents the deeper level in the LSM tree.
Level Class implements the merge(extract, sort, then save) method. Each level keeps one or many SSTables. SSTable class implements the loading and saving from disk.

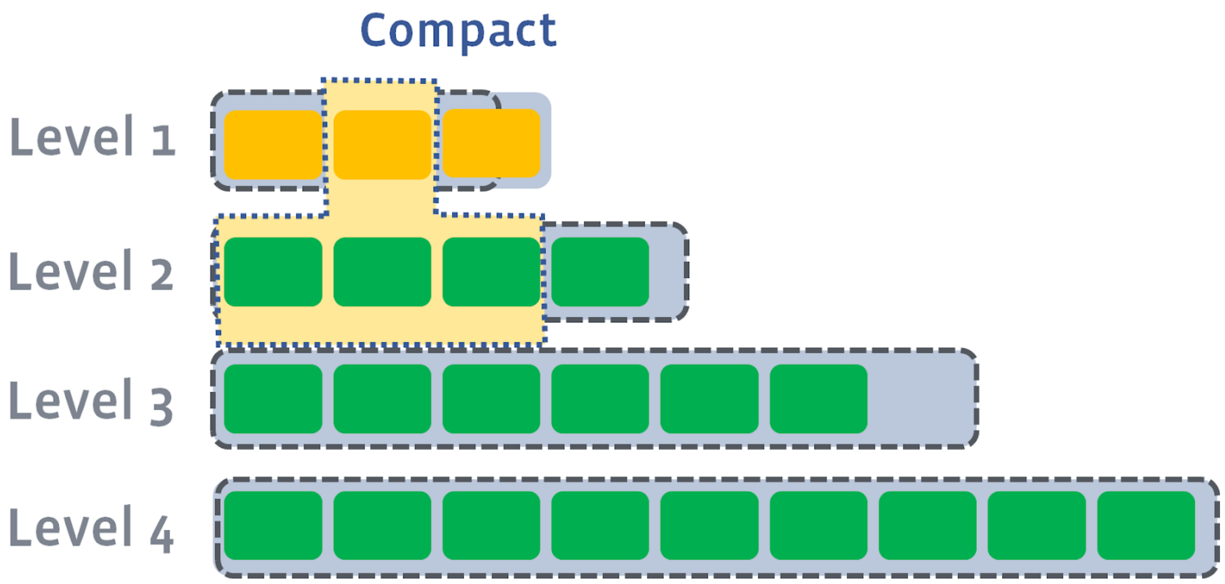
Files on disk are organized in multiple levels. We call them level-1, level-2, etc, or L1, L2, etc, for short. A special level-0 (or L0 for short) contains files just flushed from an in-memory write buffer (memtable). Each level (except level 0) is one data sorted run. As one level is full, the data are pushed to the next level and compact with original data in the level. We implemented two different compacting strategies, one is tiered compaction and the other is leveled compaction.

Size-Tiered compaction strategy: As usual, memtables are periodically flushed to new sstables. These are pretty small, and soon their number grows. As soon as we have enough of these small sstables, we compact them into one medium sstable. When we have collected enough medium tables, we compact them into one large table. And so on, with compacted sstables growing increasingly large. We implement this strategy in a preliment way just as the figure describes.

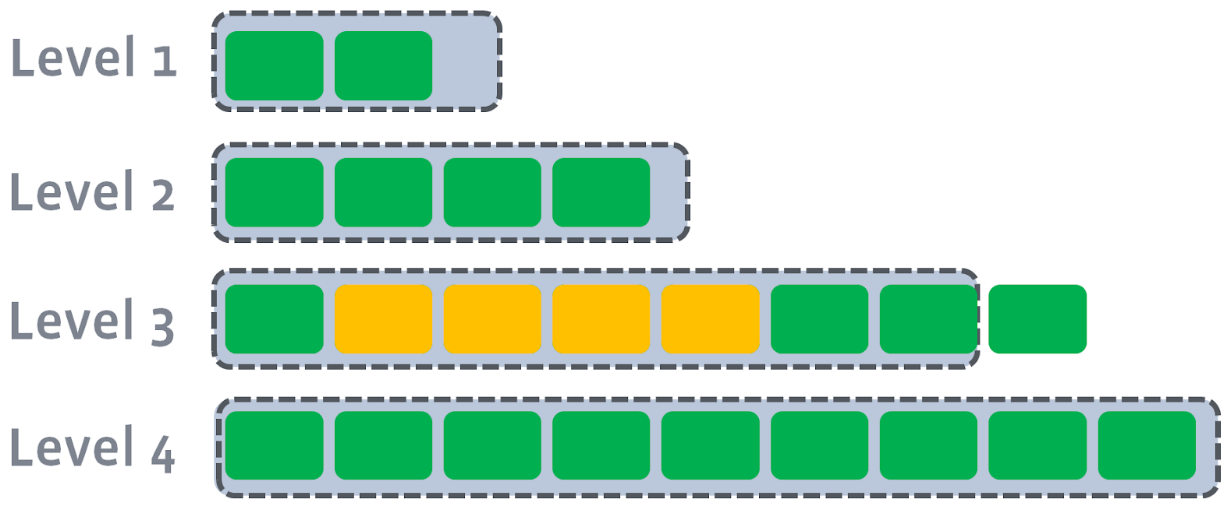


Size-Tiered compaction strategy

Leveled Compaction: Compaction triggers when number of L0 files reaches space limitation, files of L0 will be merged into L1 as before. However, as soon as the files are pushed to the next level, we merge the data to the original run. In this way, a single level always maintains a single run. In our preliminary implementation, we keep a single sstable as a run and expand the size of it as new data are merged into the run.



Before Compaction



After compaction

Classes Introduction and Implementation

DB

Core API / Fields	Description
fstream file	Save the memory table as the database
map<int, Value> table	Memory table
DiskStorage disk	Disk storage component
db_status status	Status of database (Open, Closed, Error)
int min_key	The minimum key value in the database
int max_key	The maximum key value in the database
size_t size()	Returns size of table
bool close()	Closes database
db_status open(string & fname)	Opens database
bool load_data_file(string & fname)	Takes data file and adds data to database

bool write_to_file()	Write items to file
vector<Value> execute_op(Operation op)	Executes a workload
Value get(int key)	Retrieves item from database
void put(int key, Value val)	Inserts item into database
vector<Value> scan()	Get all the entries
vector<Value> scan(int min_key, int max_key)	Get all entries in a specified range
void del(int key)	Delete an item
void del(int min_key, int max_key)	Delete a range of items
DeleteTable deleteTable	Table used to keep track of deleted items

DiskStorage

Core API / Fields	Description
string dir	Directory to save metadata of the whole component
uint64_t no	Id number of SSTables
LevelZero level0	The first level of the component
vector<LevelNonZero> levels	Other level objects saved in an array
void add(map<int, Value>)	Add the memory table into the disk as it's full and merge down if levels are full after inserting operation
Value search(int key)	Search a specific key in all levels
map<int, Value> search(int min_key, int max_key)	Search for range of keys in all levels
void clear()	Remove all items in the component
void save()	Save metadata to disk

DeleteTable

Core API / Fields	Description
vector<MinMaxPair> vec	An identifier of the SSTable
string fileName;	Filename used when persistenting
load()	Used for persistence
save()	Used for persistence
int64_t getTimeInt(int key);	Return the latest timestamp of a key in delete table

LevelNonZero

Core API / Fields	Description
string dir	Directory to save contents of the tables in this level
uint64_t size	Number of SSTables in the level
uint64_t byteCnt	Number of items in the level
int lastKey	Largest key in the level
list<SSTable> ssts	Save the metadata of the SSTables in the level
SearchResult search(int)	Search for a specific key in this level
void merge(map<int, Value>, uint64_t)	Merge content of tables from other levels into this level
void clear()	Remove all contents in this level
void save()	Save the metadata of the level on disk

LevelZero

Core API / Fields	Description

string dir	Directory to save contents of the tables in this level
uint64_t size	Number of SSTables in the level
uint64_t byteCnt	Number of items in the level
list<SSTable> ssts	Save the metadata of the SSTables in the level
SearchResult search(int)	Search for a specific key in this level
void clear()	Remove all contents in this level
void save()	Save the metadata of the level on disk

Value

Core API / Fields	Description
vector<int> items	Values that store in the map.
Value visible	Become false when this represents a tombstone

SSTable

Core API / Fields	Description
int space	The size of the table(nums of keys, can be improved); Update when load() or save()
SSTableId sstid	An identifier of the SSTable
save(map entries)	Save the given map to disk
load()	Read from disk and output a map correspond to the SSTable
Value search(int key)	Load the map from the disk and search for the key. Return a SearchResult object.
map<int, Value> search(int min_key, int max_key)	Search within range of keys

Implementation of Major APIs

Put

1. Insert the key in the BloomFilter
2. Insert into the Memory Table(if memTable's size reaches the threshold, we move this table to the disk)

Get

1. Check the BloomFilter
2. Fetch key from the database with newest insertion
3. Compare the timestamps

Delete

1. Insert a tombstone into the Memory Table which has the specific key and value.visible = false.
2. When getting a key from the database, if the value is invisible, it is deleted.

Range Delete

1. Push a <Min Key, Max Key, TimeStamp> tuple to the vector
2. If you want to get an entry, the database will check the timestamp(the time when this entry was put) of this entry, and compare it with its timestamp in DeleteTable(if not in DeleteTable, return 0)

Range Scan

1. Scan the whole database table by table. Use binary search to get the lower bound of the min key and the higher bound of the max key.
2. Merge all result maps and remove items whose value is invisible.
3. Specifically, if no parameter is passed to the method, it scans from minimum key to maximum key of the whole database.

Experiments

Performance Test

We generate 10k entries with value dimension 5 and workload with 10k operations. With leveled compaction, simple_benchmark runtime is 410 seconds, with tiered compaction, simple_benchmark runtime is 419 seconds.

Link to Repo

https://github.com/junchen-liu/cs561_templatedb