Log-Structured Merge Tree based Write-optimized Key Value Store

Jiecheng Shi and Lianke Qin

Motiv

- Ver
- Mo
- RD
- Sim

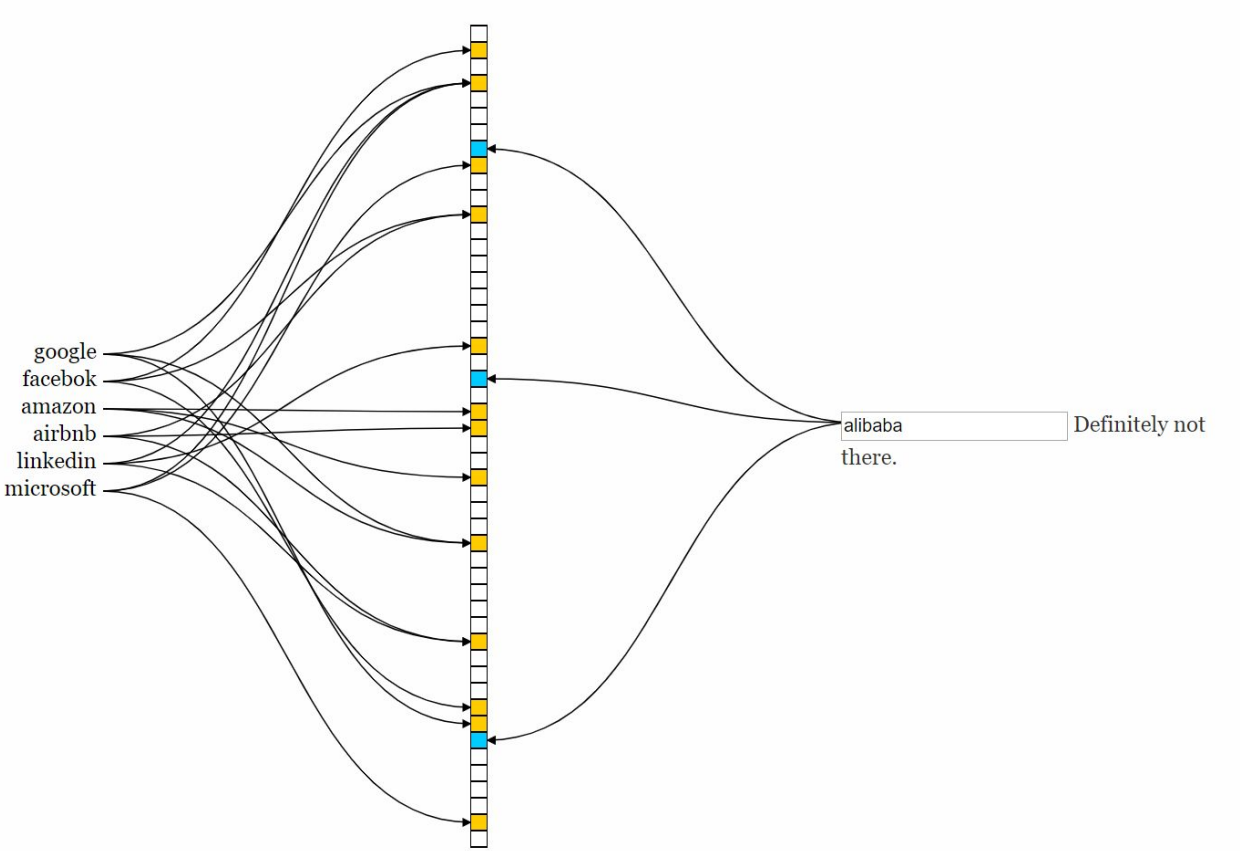
Similar projects

- LevelDB(https://github.com/google/leveldb)
- HyperLevelDB(https://github.com/rescrv/HyperLevelDB)
- RocksDB(https://github.com/facebook/rocksdb)
- PebblesDB(https://github.com/utsaslab/pebblesdb)
- SlimDB(https://github.com/aminubakori/SlimDB)

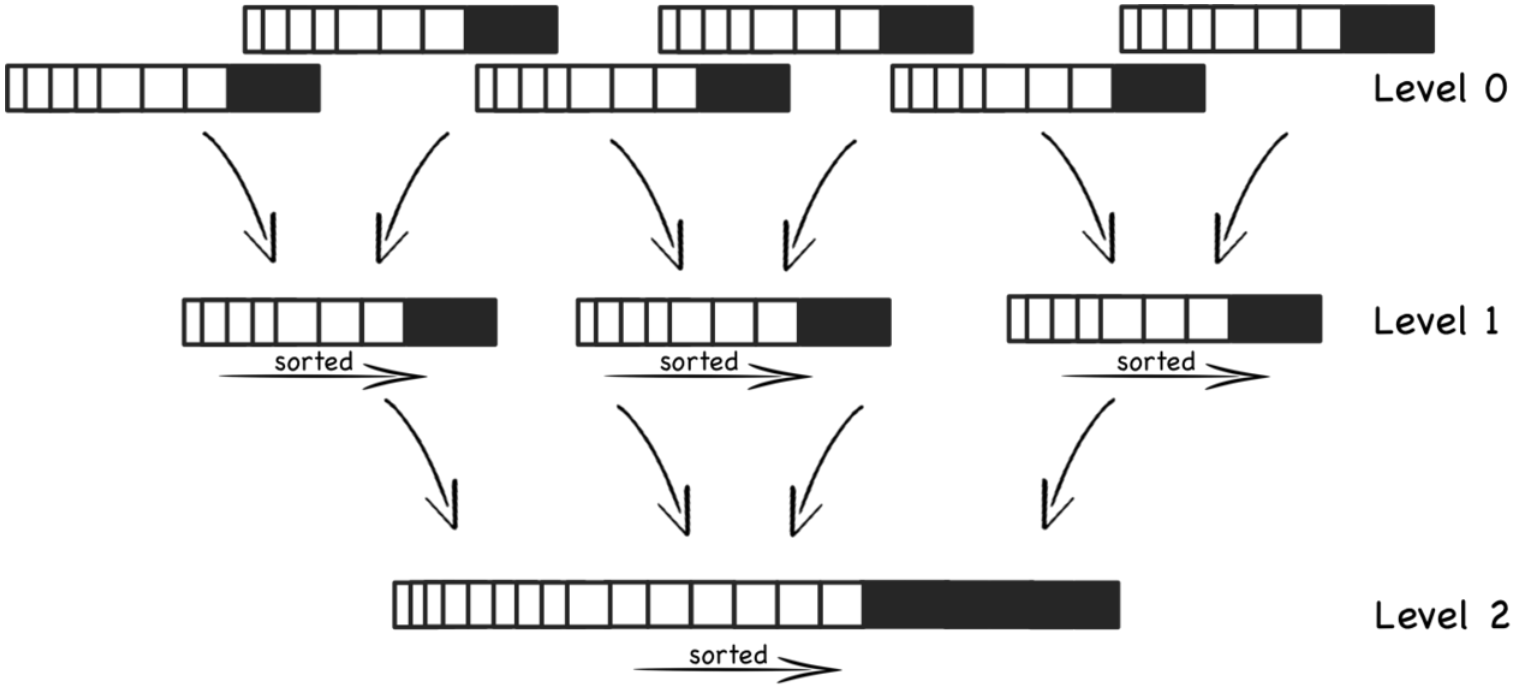
LSM tree

- In-memory buffer unsorted.
- Write operation to buffer is append-only for high throughput
- On-disk Sorted String Table, binary search in SSTable
- Multiple levels, each level consists of many SSTables
- SSTable Compaction to deeper level
- Bloom Filter to reduce unnecessary file I/O

Bloom filter visualization



Compaction visualization



Recent cool optimizations in KVS

- GPU/FPGA accelarated compaction[1][2]
- Fragmented LSM tree to reduce write amplification[3]
- Adapt for block-addressable Non-Volatile Memory SSD[4]
- LSM-Trie to reduce metadata of item locations[5]

LSMTree API design

pub fn *new*(buf_max_entries: **u64**, dep: **u64**, fa
LSMTree

pub fn put(&**mut self**, key_str: &**str**, value_str: &

pub fn get(&**self**, key_str: &**str**) -> Option<Strin

pub fn range(&**self**, start_str: &**str**, end_str: &**st**

pub fn del(&**mut self**, key_str: &**str**)

pub fn close(&mut self)(TODO)

pub fn open(&mut self, filename: &str)(TODO)

```
///
/// # Arguments
///
/// * `buf_max_entries` - Max number of entries in memory buffer
/// * `dep` - depth of LSM tree
/// * `fanout` - A factor that determines how to scale Run size for deeper levels
/// * `bf_bits_per_entry` - Used for bloom filter size initialization
/// * `num_threads` - Used for thread pool initialization
///
/// # Example
///
/// ```
///
/// use lsm_kv::lsm;
/// let mut lsm = lsm::LSMTree::new(100, 5, 10, 0.5, 4);
/// lsm.set("hello", "world");
/// lsm.set("facebook", "google");
/// lsm.set("amazon", "linkedin");
/// assert_eq!(lsm.get("hello"), Some("world"));
/// assert_eq!(lsm.get("facebook"), Some("google"));
/// lsm.del("hello");
/// assert_eq!(lsm.get("hello"), None);
/// lsm.range("amazon", "facebook");
///
/// ```
```

Client code

```rust
let mut buffer_num_pages : u64 = lsm::DEFAULT_BUFFER_NUM_PAGES;
let mut depth : u64 = lsm::DEFAULT_TREE_DEPTH;
let mut fanout : u64 = lsm::DEFAULT_TREE_FANOUT;
let mut num_threads : u64 = lsm::DEFAULT_THREAD_COUNT;
let mut bf_bits_per_entry : f32 = lsm::DEFAULT_BF_BITS_PER_ENTRY;

let mut opts : Options = Options::new();
opts.optopt( short_name: "b", long_name: "", desc: "number of pages in buffer", hint: "PAGE_NUM");
opts.optopt( short_name: "d", long_name: "", desc: "number of levels", hint: "LEVEL_NUM");
opts.optopt( short_name: "f", long_name: "", desc: "level fanout", hint: "FANOUT");
opts.optopt( short_name: "t", long_name: "", desc: "number of threads", hint: "THREADS_NUM");
opts.optopt( short_name: "r", long_name: "", desc: "bloom filter bits per entry", hint: "BLOOM_BITS");
let matches : Matches = match opts.parse( args: &args[1..]) {
    Ok(m : Matches ) => m,
    Err(f : Fail ) => panic!(f.to_string()),
};
if matches.opt_str( nm: "b").is_some() {
    buffer_num_pages = matches.opt_str( nm: "b").unwrap().parse().unwrap()
}
if matches.opt_str( nm: "d").is_some() {
    depth = matches.opt_str( nm: "d").unwrap().parse().unwrap()
}
if matches.opt_str( nm: "f").is_some() {
    fanout = matches.opt_str( nm: "f").unwrap().parse().unwrap()
}
if matches.opt_str( nm: "t").is_some() {
    num_threads = matches.opt_str( nm: "t").unwrap().parse().unwrap()
}
if matches.opt_str( nm: "r").is_some() {
    bf_bits_per_entry = matches.opt_str( nm: "r").unwrap().parse().unwrap()
}

let buffer_max_entries : u64 =
    buffer_num_pages * page_size::get() as u64 / size_of::<EntryT>() as u64;

let mut lsm_tree : LSMTree = LSMTree::new(
    buf_max_entries: buffer_max_entries,
    dep: depth,
    fanout,
    bf_bits_per_entry,
    num_threads,
);
```

```rust
fn command_loop(lsm_tree: &mut LSMTree, input: impl BufRead) {
    for line : Result<String, Error> in input.lines() {
        match line {
            Ok(line : String ) => {
                let tokens: Vec<&str> = line.split_whitespace().collect();
                if tokens.is_empty() {
                    continue;
                } else {
                    match tokens[0] {
                        "p" => {
                            lsm_tree.put( key_str: tokens[1], value_str: tokens[2]);
                            println!("The k-v ({}, {}) has been inserted!", tokens[1], tokens[2]);
                        }
                        "g" => {
                            let val : Option<String> = lsm_tree.get( key_str: tokens[1]);
                            if val.is_some() {
                                println!("The value of key {} is {}", tokens[1], val.unwrap());
                            } else {
                                println!("No value with key {} in the DB!", tokens[1]);
                            }
                        }
                        "r" => {
                            let vals : Vec<String> = lsm_tree.range( start_str: tokens[1], end_str: tokens[2]);
                            if vals.is_empty() {
                                println!(
                                    "No value with key between {} and {} in the DB!",
                                    tokens[1], tokens[2]
                                );
                            } else {
                                println!(
                                    "Values with keys between {} and {} are:",
                                    tokens[1], tokens[2]
                                );
                                for val : String in vals {
                                    print!("{} ", val);
                                }
                                println!();
                            }
                        }
                        "d" => {
                            lsm_tree.del( key_str: tokens[1]);
                            println!("The k-v with key {} has been deleted!", tokens[1]);
                        }
                        "l" => {}
```

Client usage example

./lsm-kv [-b PAGE_NUM] [-d LEVEL_NUM] [-f FANOUT] [-t THREADS_NUM] [-r BLOOM_BITS]

PUT -> “p KEY VAL”

GET -> “g KEY”

RANGE -> “r START END”

DELETE -> “d KEY”

```
p facebook 1
The k-v (facebook, 1) has been inserted!
p google 22
The k-v (google, 22) has been inserted!
p amazon 333
The k-v (amazon, 333) has been inserted!
p linkedin 444
The k-v (linkedin, 444) has been inserted!
g google
The value of key google is 22
g airbnb
No value with key airbnb in the DB!
d google
The k-v with key google has been deleted!
g google
No value with key google in the DB!
r airbnb facebook
Values with keys between airbnb and facebook are:
333 1
```

Interesting implementation tidbits

Data Persistence On Disk: **Memory Map**

```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

Unsafe?

Inevitable and Sound!

- Shared memory
- Raw pointer

```
let len :usize = size_of::<EntryT>() * self.max_size as usize;

unsafe {
    assert!(libc::lseek(self.mapping_fd, offset: (len - 1) as i64, whence: 0) != -1);
    assert!(libc::write(self.mapping_fd, buf: "".as_ptr() as *const c_void, count: 1) != -1);
}

match MemoryMap::new(
    min_len: len,
    options: &[
        MapOption::MapWritable,
        MapOption::MapFd(self.mapping_fd),
        MapOption::MapOffset(0),
        MapOption::MapNonStandardFlags(0x01),
    ],
) {
    Ok(map :MemoryMap ) => {
        self.mapping = Some(map);
    }
    Err(_) => panic!("Mapping failed!"),
}
```

```
its_per_entry),
as u64) as usize),

(),
```

```
pub struct MemoryMap {
    data: *mut u8,
    len: usize,
    kind: MemoryMapKind,
}
```

```
Ok(map :MemoryMap ) => unsafe {
    assert_eq!(map.len(), KEY_SIZE + VALUE_SIZE);
    self.mapping = Some(map);
    let mut res: Vec<EntryT> = Vec::new();
    for i :u64 in 0..self.size {
        res.push( value: EntryT {
            key: std::slice::from_raw_parts(
                data: self.mapping
                    .as_ref()
                    .unwrap()
                    .data()
                    .add( count: size_of::<EntryT>() * i as usize),
                len: KEY_SIZE,
            )
            .to_vec(),
            value: std::slice::from_raw_parts(
                data: self.mapping
                    .as_ref()
                    .unwrap()
                    .data()
                    .add( count: size_of::<EntryT>() * i as usize + KEY_SIZE),
                len: VALUE_SIZE,
            )
            .to_vec(),
        })
    }
    res
},
Err(_) => panic!("Mapping failed!"),
```

Interesting implementation tidbits

Should we use unsafe in Rust?

Should we use pointer & file descriptor in idiomatic Rust?

Think carefully but don't be afraid of unsafe!

- Feature
- Special requirement
- Foreign Function Interface

```rust
match File::open( path: self.tmp_file.as_path()) {
    Ok(fd :File ) => {
        self.mapping_fd = fd.as_raw_fd();
    }
    Err(_) => panic!("Open temp file failed!"),
};
```

```rust
unsafe {
    assert!(libc::lseek(self.mapping_fd, offset: (len - 1) as i64, whence: 0) != -1);
    assert!(libc::write(self.mapping_fd, buf: "".as_ptr() as *const c_void, count: 1) != -1);
}
```

```rust
unsafe {
    libc::close(self.mapping_fd);
}
```

Possible improvement

- Safe parallel operations
- Better documentation
- Support more data structure in Key/Value
- Going distributed?

References

[1] Zhang, Teng, et al. "FPGA-Accelerated Compactions for LSM-based Key-Value Store." 18th USENIX Conference on File and Storage Technologies. 2020.
[2] Huang, Gui, et al. "X-Engine: An optimized storage engine for large-scale E-commerce transaction processing." Proceedings of the 2019 International Conference on Management of Data. 2019.
[3] Raju, Pandian, et al. "Pebblesdb: Building key-value stores using fragmented log-structured merge trees." Proceedings of the 26th Symposium on Operating Systems Principles. 2017.
[4] Lepers, Baptiste, et al. "KVell: the design and implementation of a fast persistent key-value store." Proceedings of the 27th ACM Symposium on Operating Systems Principles. 2019.
[5] Wu, Xingbo, et al. "LSM-trie: An LSM-tree-based ultra-large key-value store for small data items." 2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15). 2015.

Thanks! Any Question?