

OUTLINE



1. Introduction
2. Major Components
3. Other things

OUTLINE

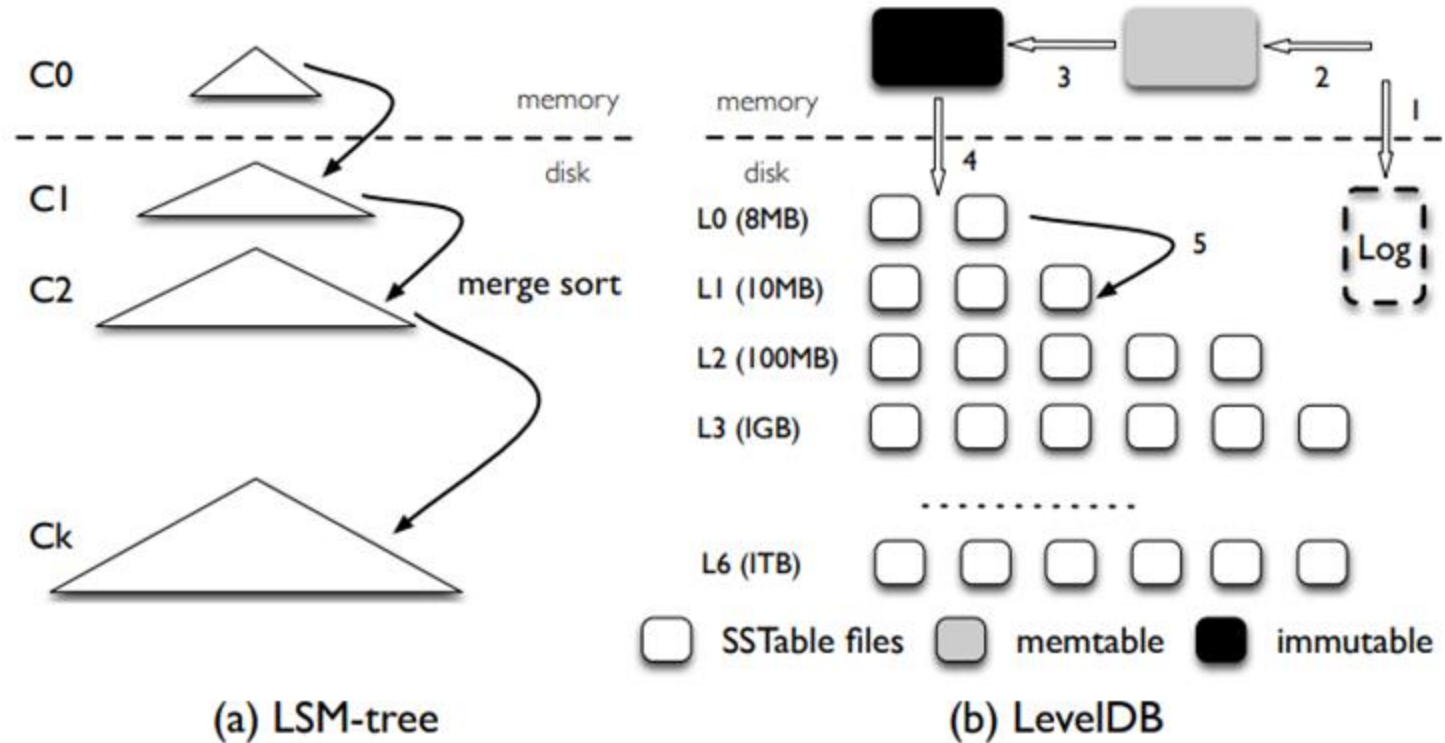


1. Introduction

2. Major Components

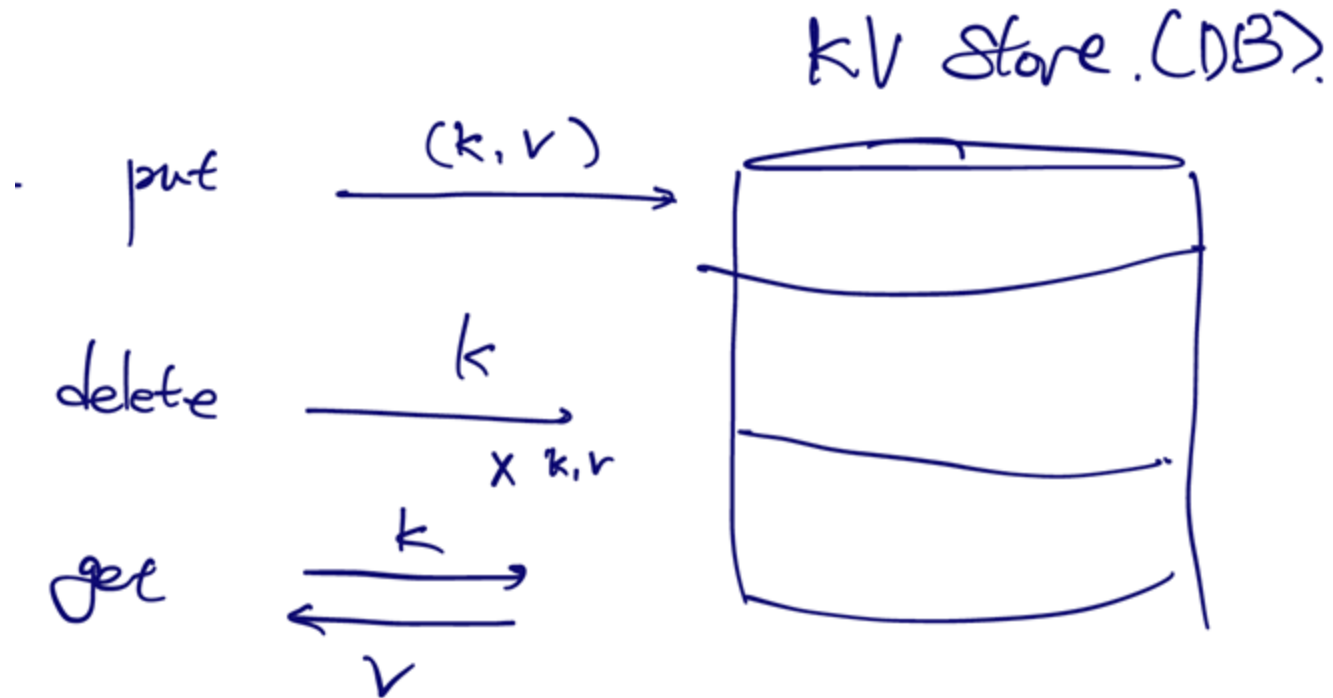
3. Other things

High-Level Architecture



Lanyue Lu. et al, WiscKey (Fast '16)

Basic Operations



Basic Operations



Basic Operations

Features

- * Keys and values are arbitrary byte arrays.
- * Data is stored sorted by key.
- * Callers can provide a custom comparison function to override the sort order.
- * The basic operations are `Put(key,value)`, `Get(key)`, `Delete(key)`.
- * Multiple changes can be made in one atomic batch.
- * Users can create a transient snapshot to get a consistent view of data.
- * Forward and backward iteration is supported over the data.

Basic Operations

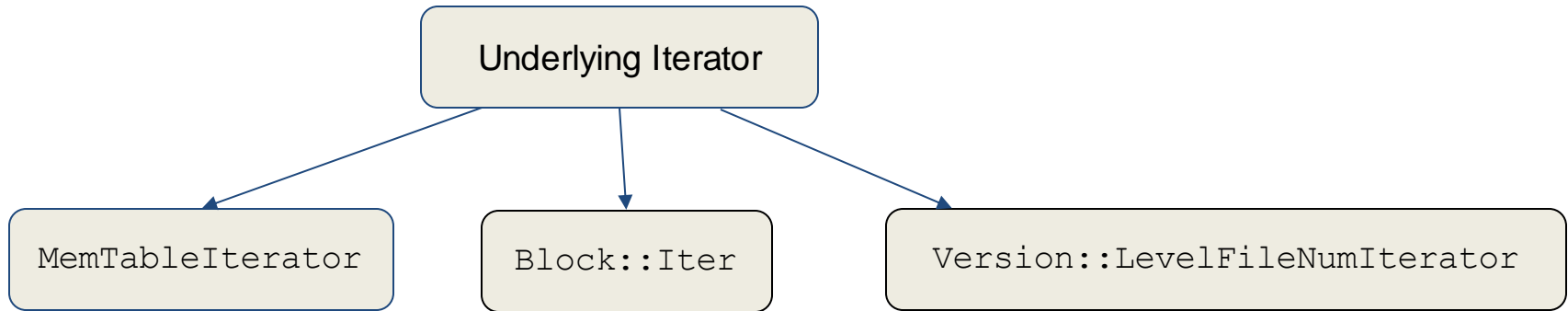
Features

- * Keys and values are arbitrary byte arrays.
- * Data is stored sorted by key.
- * Callers can provide a custom comparison function to override the sort order.
- * The basic operations are `Put(key,value)`, `Get(key)`, `Delete(key)`.
- * Multiple changes can be made in one atomic batch.
- * Users can create a transient snapshot to get a consistent view of data.
- * Forward and backward iteration is supported over the data.

```
```C++
leveldb::Iterator* it = db->NewIterator(leveldb::ReadOptions());
for (it->SeekToFirst(); it->Valid(); it->Next()) {
 cout << it->key().ToString() << ": " << it->value().ToString() << endl;
}
assert(it->status().ok()); // Check for any errors found during the scan
delete it;
```
```

Iterator

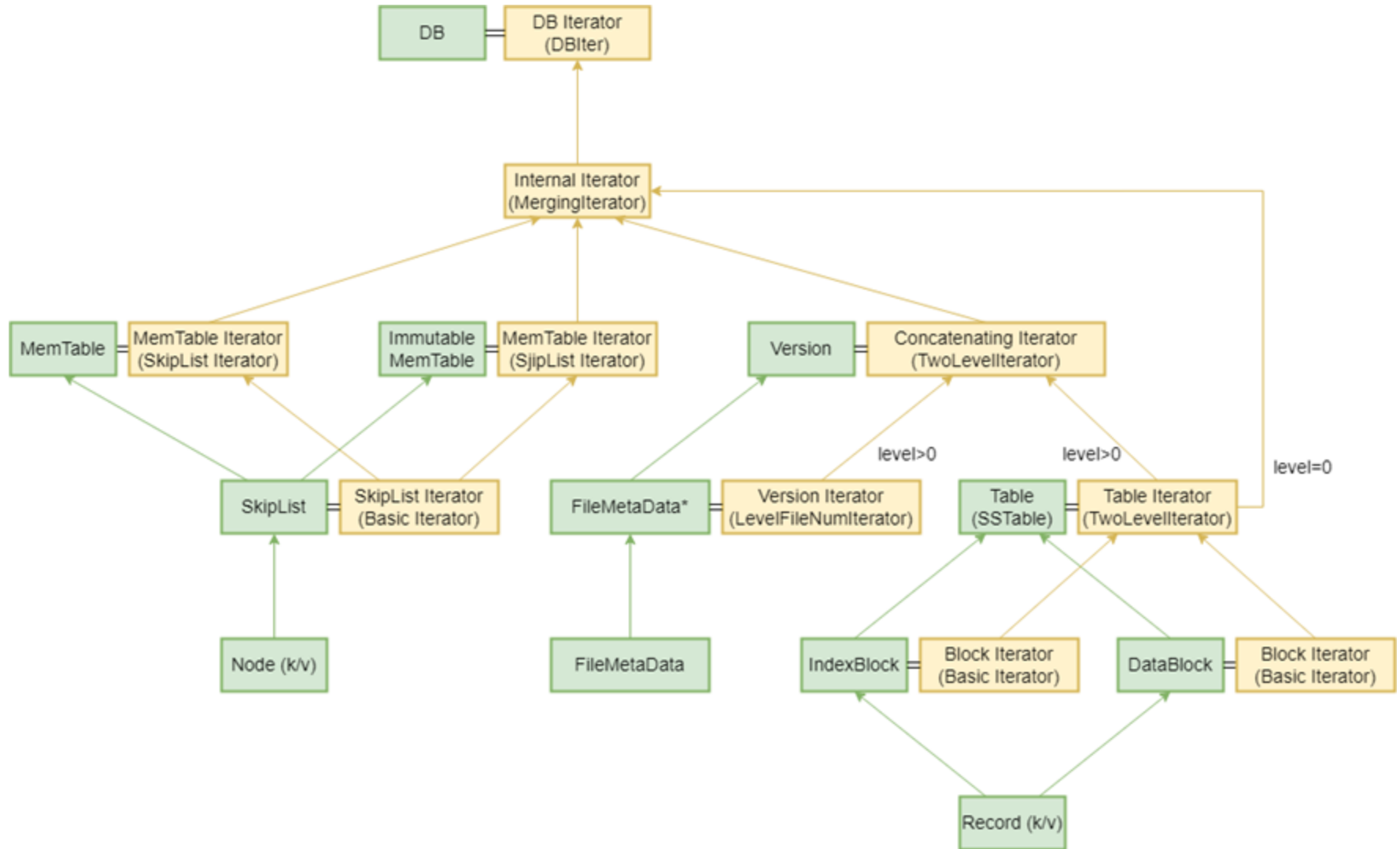
Most Beautiful thing in LevelDB! for MemTable, SSTable, Compaction
Underlying Iterator:



Combination Iterator: Combine underlying iterator













1. TwoLevelIterator 2. MergingIterator

Iterator



Overview of LevelDB Code

30,000+ LoC written in C++

| |
|---|
|  .github/workflows |
|  benchmarks |
|  cmake |
|  db |
|  doc |
|  helpers/memenv |
|  include/leveldb |
|  issues |
|  port |
|  table |
|  third_party |
|  util |

OUTLINE



1. Introduction

2. Major Components

3. Other things

Basic Data Structures (include/leveldb/, /util/)

./util/

- arena
- bloom
- cache
- coding
- comparator
- crc32c
- env/env_posix/env_windows
- testharness
- logging
- ...

- *String Slice*: construct, clear, compare, acquire
- *Hash*: For creating Hash Table, Bloom Filter.
- *Cache* (LRU Cache, mutex, double linked list + Hash Table)
LevelDB divides the LRU Cache into 16 blocks. Why? for efficiency!
- *Bloom Filter*: create a small filter based on a set of keys, store filter and kv pairs.
`virtual bool KeyMayMatch()`
- *Arena*: memory pool, 4KB, an optimization for the limited space.
- *Skiplist*: for MemTable `db/skiplist.h`

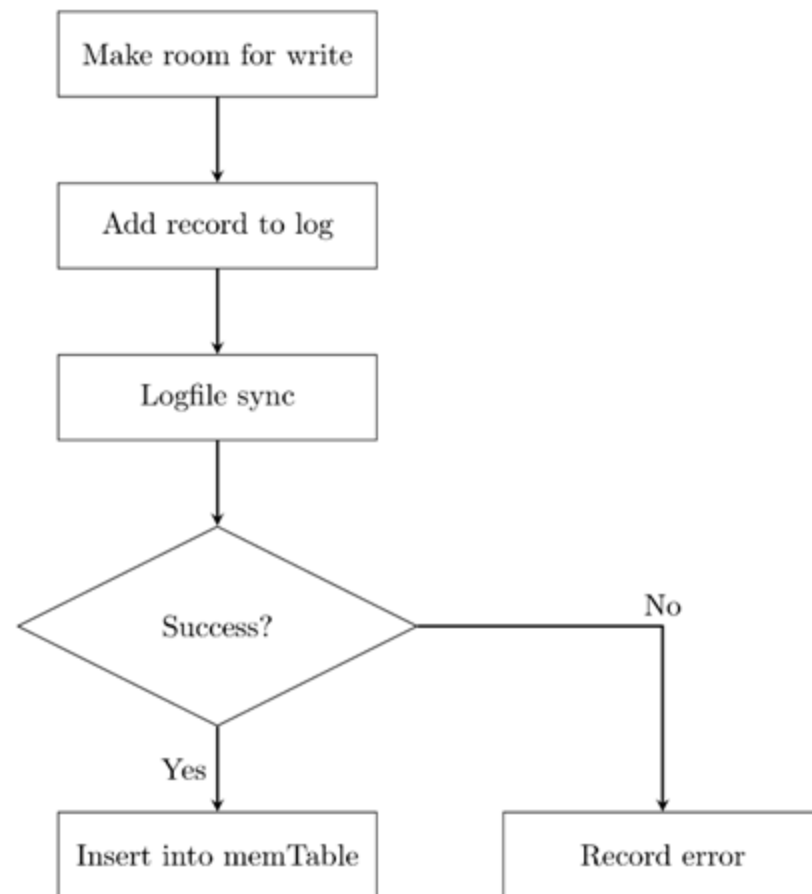
Write in LevelDB

Write to DB:

1. Append data into log,

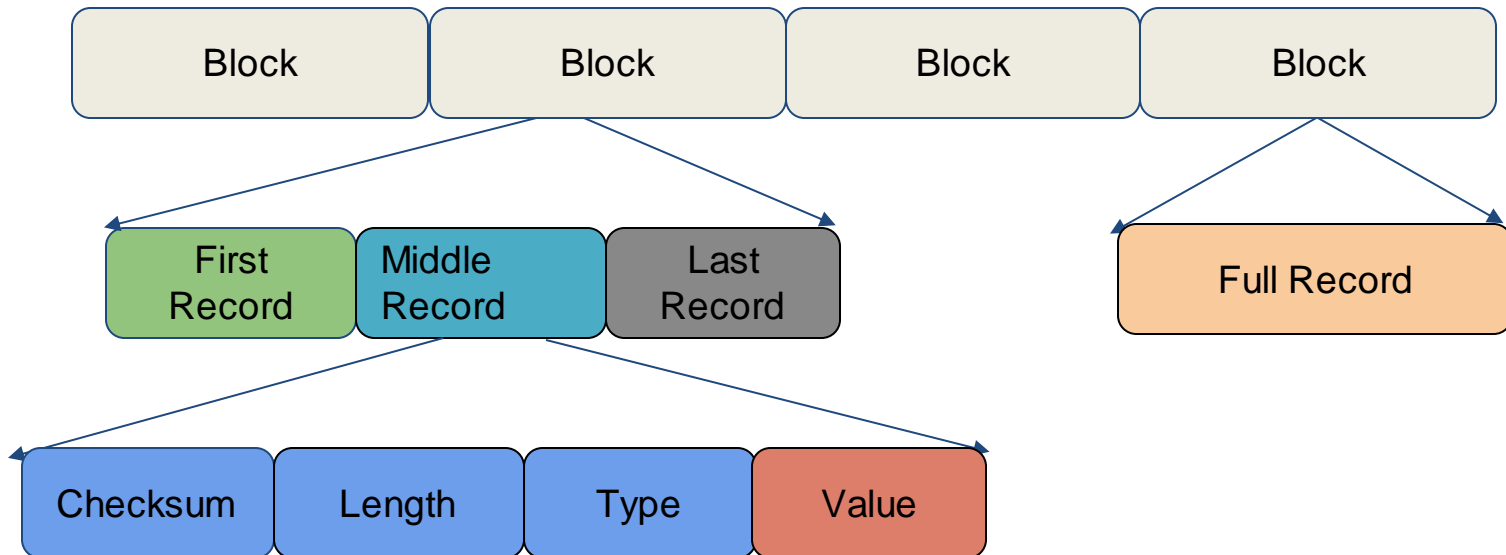
2. Insert data to MemTable

- guarantee write speed.
- write to disk, easily recovery based on log.
- MemTable reaches a certain size, set new MemTable

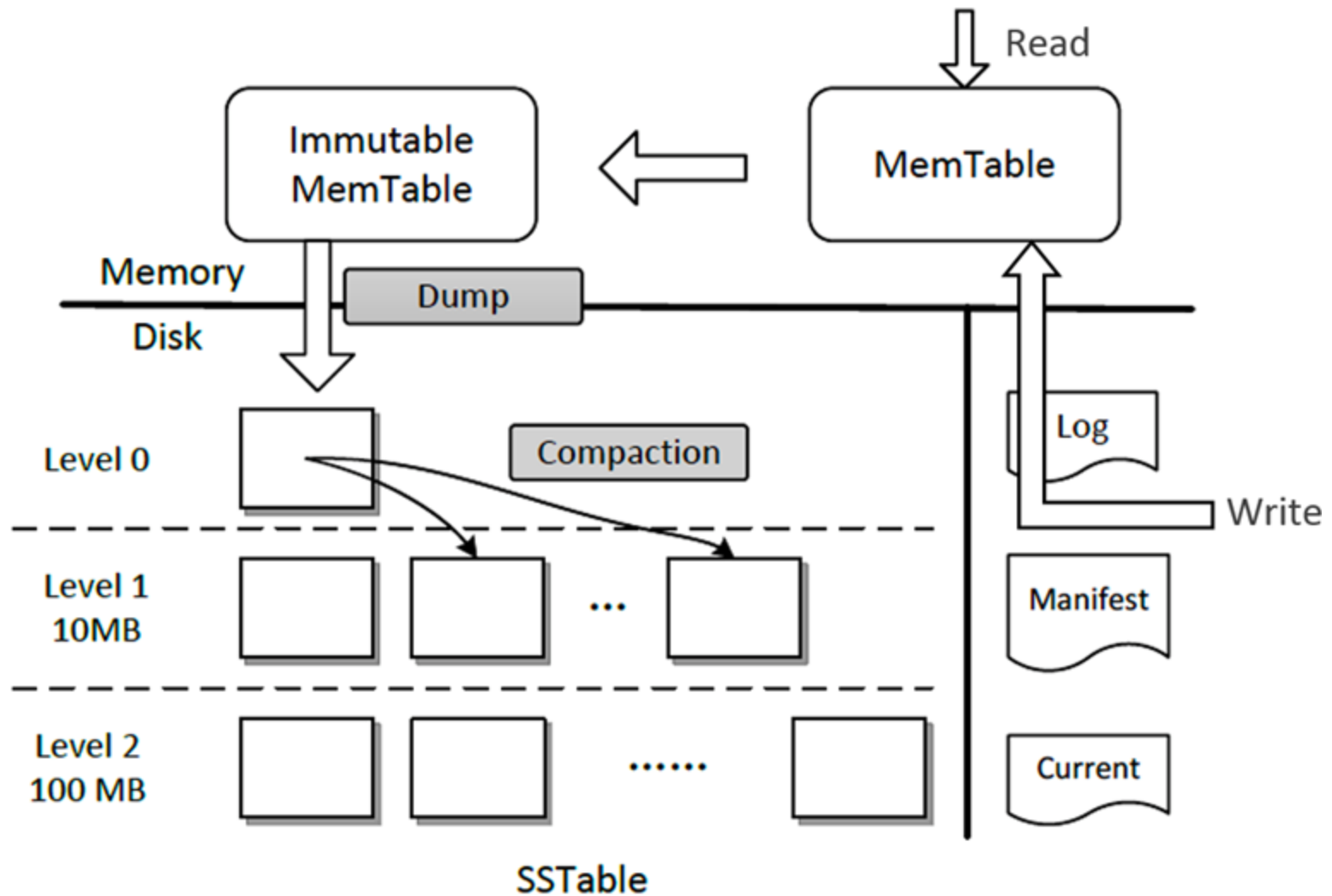


Write in LevelDB

1. Write Batch: `Write(opt, &batch)` put, delete, append
put, delete -> string, provide interface for MemTable
1. Log in LevelDB: WAL Sync, WAL crop

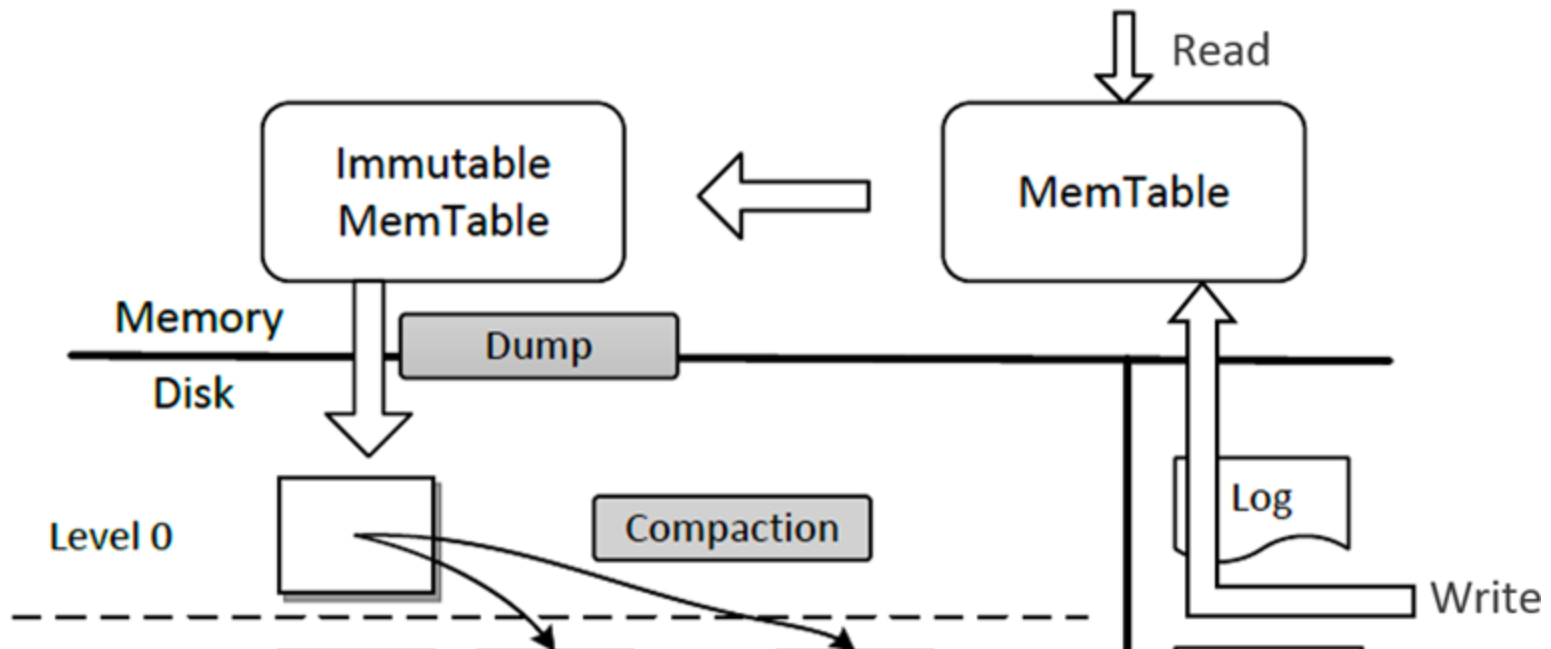


MemTable - Operation



<https://microsoft.github.io/MLOS/notebooks/LevelDbTuning/>

MemTable - Operation



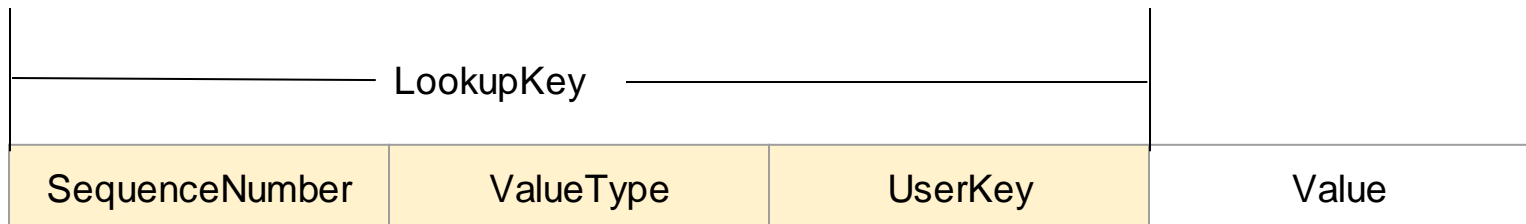
```
// Amount of data to build up in memory (backed by an unsorted log
10M // on disk) before converting to a sorted on-disk file.
//
// Larger values increase performance, especially during bulk loads.
Level // Up to two write buffers may be held in memory at the same time,
100M // so you may wish to adjust this parameter to control memory usage.
// Also, a larger write buffer will result in a longer recovery time
// the next time the database is opened.
size_t write_buffer_size = 4 * 1024 * 1024;
```

<https://microsoft.github.io/MLOS/notebooks/LevelDbTuning/>

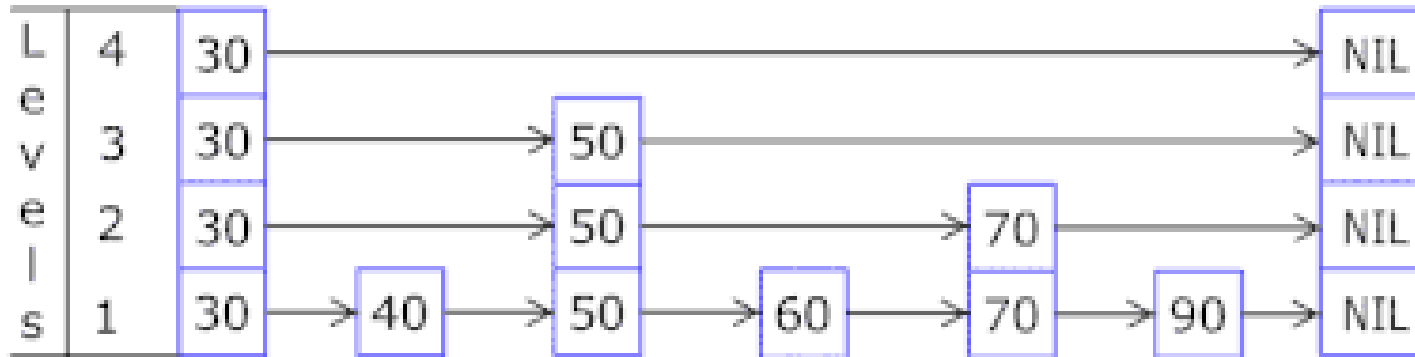
MemTable - Structure

```
// Add an entry into memtable that maps key to value at the
// specified sequence number and with the specified type.
// Typically value will be empty if type==kTypeDeletion.
void Add(SequenceNumber seq, ValueType type, const Slice& key,
         const Slice& value);

// If memtable contains a value for key, store it in *value and return true.
// If memtable contains a deletion for key, store a NotFound() error
// in *status and return true.
// Else, return false.
bool Get(const LookupKey& key, std::string* value, Status* s);
```



MemTable - Skiplist



https://en.wikipedia.org/wiki/Skip_list

- Structure / Implementation is simpler than balanced binary trees
- Time Complexity : $O(\log n)$, Also space saving
- Better concurrence ability than binary tree

SSTable - Structure

| |
|---------------------|
| Data Block 1 |
| Data Block 2 |
| ... |
| Data Block N |
| Filter Meta Block 1 |
| ... |
| Filter Meta Block K |
| Meta Index Block |
| Index Block |
| Footer |

SSTable - Read / DBImpl::Get

```
MemTable* mem = mem_;
MemTable* imm = imm_;
Version* current = versions_>current();
mem->Ref();
if (imm != nullptr) imm->Ref();
current->Ref();

bool have_stat_update = false;
Version::GetStats stats;

// Unlock while reading from files and memtables
{
    mutex_.Unlock();
    // First look in the memtable, then in the immutable memtable (if any).
    LookupKey lkey(key, snapshot);
    1. if (mem->Get(lkey, value, &s)) {
        // Done
    2. } else if (imm != nullptr && imm->Get(lkey, value, &s)) {
        // Done
    3. } else {
        s = current->Get(options, lkey, value, &stats);
        have_stat_update = true;
    }
    mutex_.Lock();
}
```

SSTable - Read / Version::Get

```
ForEachOverlapping(state.saver.user_key, state.ikey, &state, &State::Match);  
  
return state.found ? state.s : Status::NotFound(Slice());
```

```
static bool Match(void* arg, int level, FileMetaData* f) {  
    State* state = reinterpret_cast<State*>(arg);  
  
    if (state->stats->seek_file == nullptr &&  
        state->last_file_read != nullptr) {  
        // We have had more than one seek for this read. Charge the 1st file.  
        state->stats->seek_file = state->last_file_read;  
        state->stats->seek_file_level = state->last_file_read_level;  
    }  
}
```

SSTable - Read / TableCache::Get

```
Status TableCache::Get(const ReadOptions& options, uint64_t file_number,
                      uint64_t file_size, const Slice& k, void* arg,
                      void (*handle_result)(void*, const Slice&,
                      const Slice&)) {
    Cache::Handle* handle = nullptr;
    Status s = FindTable(file_number, file_size, &handle);
    if (s.ok()) {
        Table* t = reinterpret_cast<TableAndFile*>(cache_>Value(handle))>table;
        s = t->InternalGet(options, k, arg, handle_result);
        cache_>Release(handle);
    }
    return s;
}
```

SSTable - Read / Table::InternalGet

```
Status Table::InternalGet(const ReadOptions& options, const Slice& k, void* arg,
                          void (*handle_result)(void*, const Slice&,
                          const Slice&)) {
    Status s;
    Iterator* iiter = rep_>index_block->NewIterator(rep_>options.comparator);
1. iiter->Seek(k);
    if (iiter->Valid()) {
        Slice handle_value = iiter->value();
        FilterBlockReader* filter = rep_>filter;
        BlockHandle handle;
2. if (filter != nullptr && handle.DecodeFrom(&handle_value).ok() &&
        !filter->KeyMayMatch(handle.offset(), k)) {
            // Not found
        } else {
3. Iterator* block_iter = BlockReader(this, options, iiter->value());
4. block_iter->Seek(k);
5. if (block_iter->Valid()) {
        (*handle_result)(arg, block_iter->key(), block_iter->value());
    }
    s = block_iter->status();
    delete block_iter;
}
}
if (s.ok()) {
    s = iiter->status();
}
delete iiter;
return s;
}
```

SSTable - Write

- Flush from Memtable (Background Compaction)
- Compaction occurs at storage

SSTable - Write

- Flush from Memtable (Background Compaction)
- Compaction occurs at storage

SSTable - Write / Flush from MemTable

```
void DBImpl::BackgroundCompaction() {  
    mutex_.AssertHeld();  
  
    if (imm_ != nullptr) {  
        CompactMemTable();  
        return;  
    }  
}
```

```
Status s = WriteLevel0Table(imm_, &edit, base);
```

```
Status s;  
{  
    mutex_.Unlock();  
    s = BuildTable(doname_, env_, options_, table_cache_, iter, &meta);  
    mutex_.Lock();  
}
```

SSTable - Write / BuildTable

```
Status BuildTable(const std::string& dbname, Env* env, const Options& options,
                  TableCache* table_cache, Iterator* iter, FileMetaData* meta) {
    Status s;
    meta->file_size = 0;
    iter->SeekToFirst();
```

```
1. std::string fname = TableFileName(dbname, meta->number);
   if (iter->Valid()) {
       WritableFile* file;
       s = env->NewWritableFile(fname, &file);
       if (!s.ok()) {
           return s;
       }

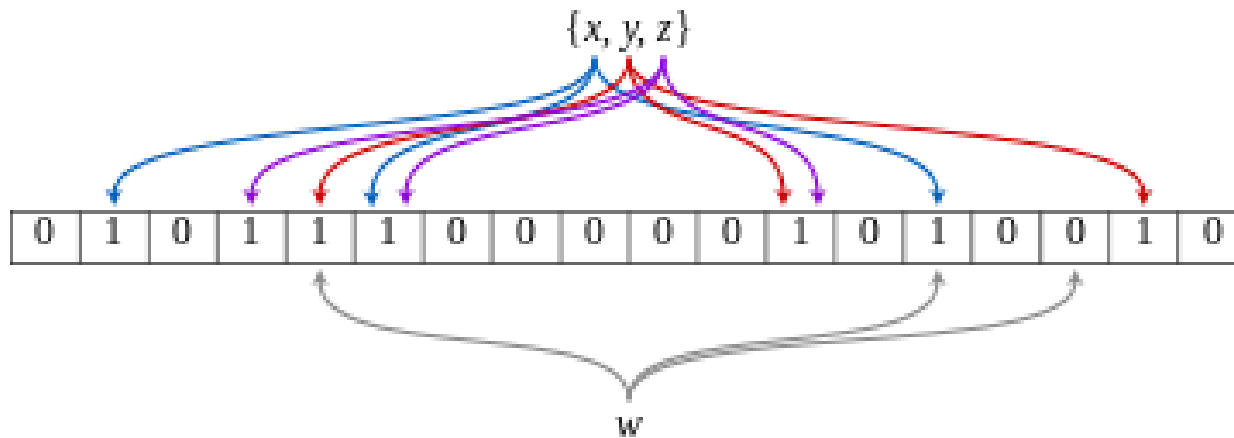
       TableBuilder* builder = new TableBuilder(options, file);
       meta->smallest.DecodeFrom(iter->key());
       Slice key;
2. for (; iter->Valid(); iter->Next()) {
       key = iter->key();
       builder->Add(key, iter->value());
   }
   if (!key.empty()) {
       meta->largest.DecodeFrom(key);
   }
```

```
       // Finish and check for builder errors
3. s = builder->Finish();
   if (s.ok()) {
       meta->file_size = builder->FileSize();
       assert(meta->file_size > 0);
   }
   delete builder;

   // Finish and check for file errors
4. if (s.ok()) {
       s = file->Sync();
   }
   if (s.ok()) {
       s = file->Close();
   }
   delete file;
   file = nullptr;

   if (s.ok()) {
       // Verify that the table is usable
5. Iterator* it = table_cache->NewIterator(ReadOptions(), meta->number,
                                          meta->file_size);
       s = it->status();
       delete it;
   }
}
```

Bloom Filter



https://en.wikipedia.org/wiki/Bloom_filter

| |
|---------------------|
| Data Block 1 |
| Data Block 2 |
| ... |
| Data Block N |
| Filter Meta Block 1 |
| ... |
| Filter Meta Block K |
| Meta Index Block |
| Index Block |
| Footer |

Bloom Filter

```
class BloomFilterPolicy : public FilterPolicy {
public:
    explicit BloomFilterPolicy(int bits_per_key) : bits_per_key_(bits_per_key) {
        // We intentionally round down to reduce probing cost a little bit
        k_ = static_cast<size_t>(bits_per_key * 0.69); // 0.69  $\approx \ln(2)$ 
        if (k_ < 1) k_ = 1;
        if (k_ > 30) k_ = 30;
    }
}
```

Compaction

Minor Compaction > Manual Compaction > Size Compaction > Seek Compaction

immutable memtable -> level 0 SSTable DBImpl::MakeRoomForWrite()

When MemTable -> SSTable, trigger BackgroundCompaction()

1. Score of a certain level ≥ 1 .
2. The number of invalid queries of a certain file exceeds the threshold.

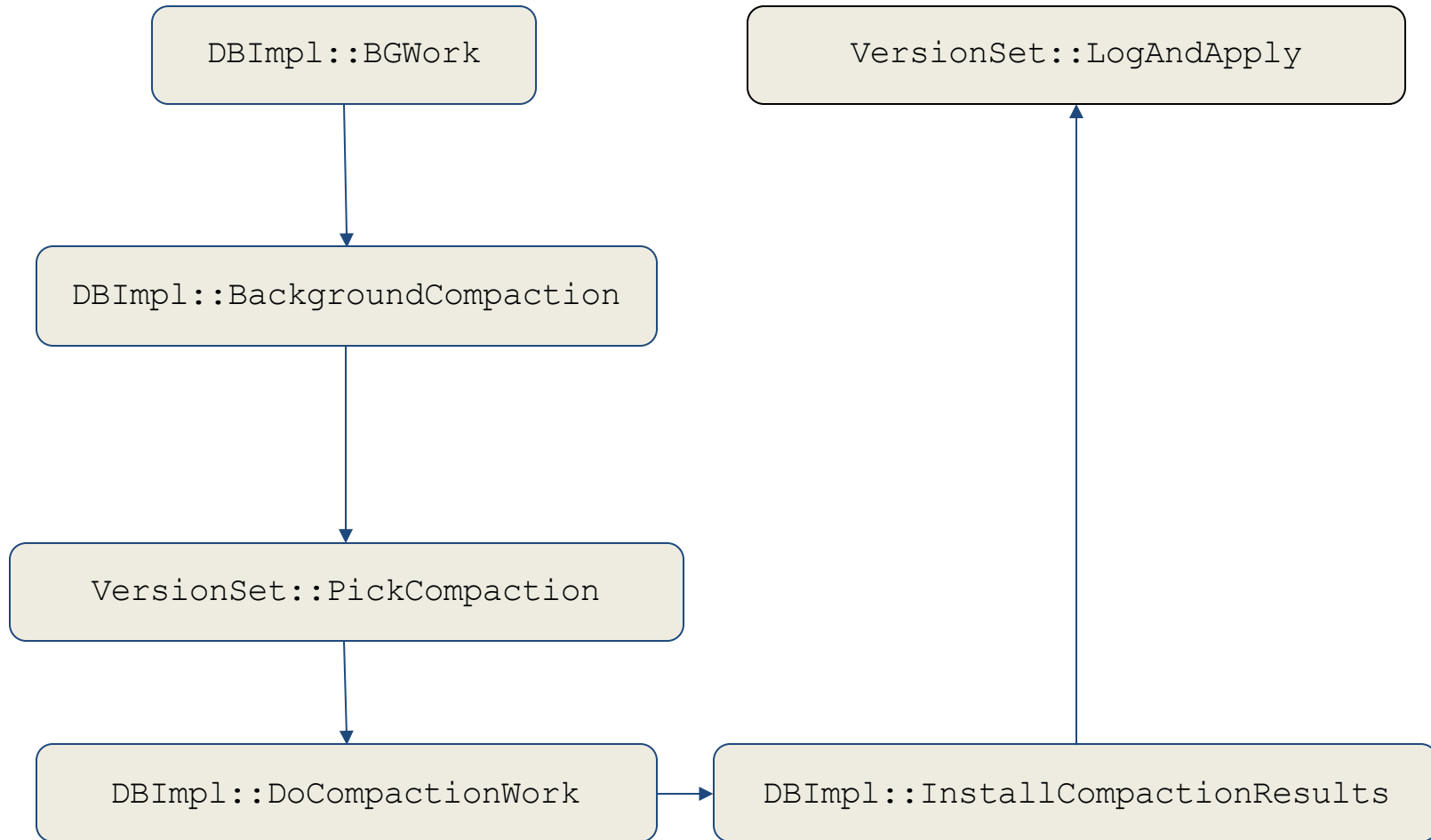
```
const bool size_compaction = (current->compaction_score_ >= 1);  
const bool seek_compaction = (current->file_to_compact_ != nullptr);
```

Fig . Part of VersionSet::PickCompaction()

More detail for compaction?

1. For level 0, if the number of files > 4, compaction.
2. Other level, the size of all files > 10^4 , compaction
3. The threshold for the number of queries for a file is defined in VersionSet::Builder::Apply

Compaction



Version Control

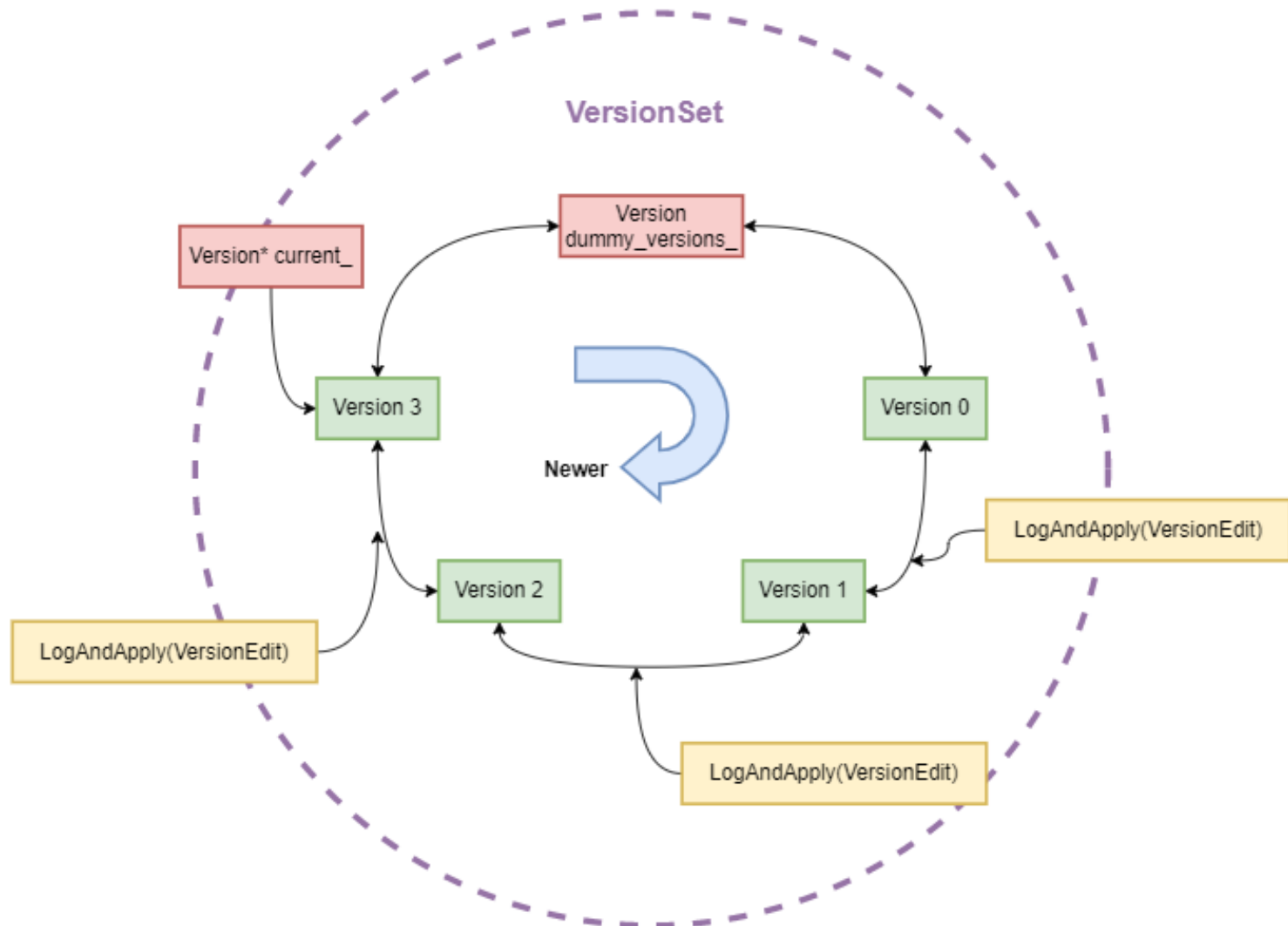
Changes of SSTable -> version control: 1. MemTable becomes SSTable. 2. Compaction. Just like git!

1. init, empty repo (No SSTable files)
2. add files or delete files, commit (VersionEdit)
3. based on current commit and previous version, can get current version (Version)
4. based on initial state and all commit log, can get all version (VersionSet)
5. HEAD point to current version (CURRENT file)

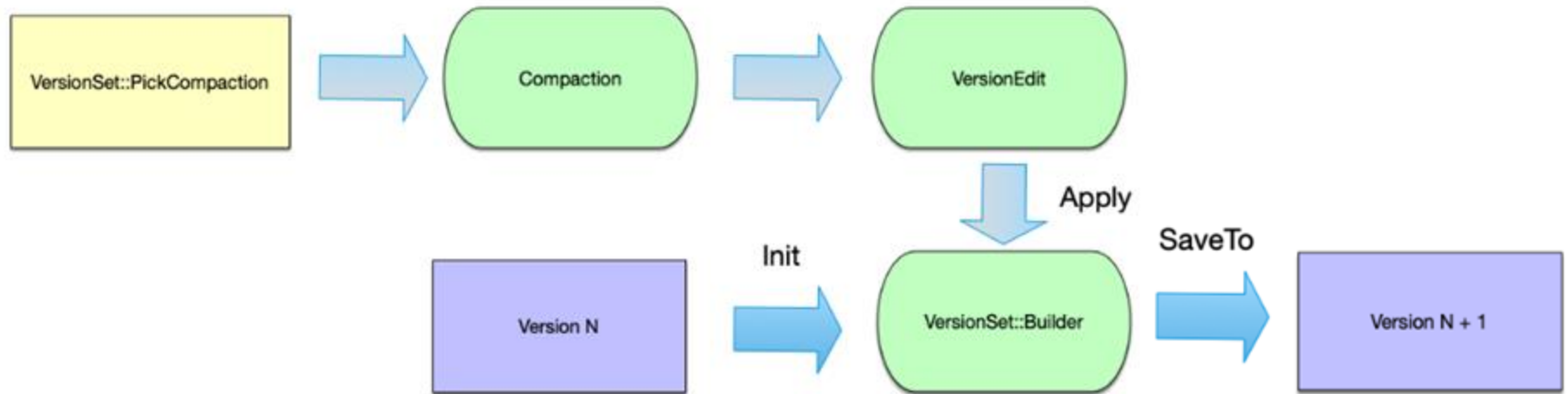
Data info for one version:

1. `std::vector<FileMetaData*>` : store metadata of every SSTable.
2. `FileMetaData* file_to_compact_, int file_to_compact_level_ :`
level and file for next compaction.
3. `double compaction_score_ and int compaction_level_.`

Version Control



Version Control



OUTLINE



1. Introduction

2. Major Components

3. Other things

Other things

- **Synchronization:**

There are a large number of concurrent access scenarios in LevelDB, which requires synchronization support.

Package C++ standard lib(mutex and ConVar)

to port/port_stdcxx.h

DBImpl::Write()

- **Atomic:**

Relaxed Ordering: atomicity, counting function (eg. arena)

Release-Acquire Ordering: atomicity also execution order

Synchronization and atomic

- Synchronization:

```
size_t MemoryUsage() const {  
    return memory_usage_.load(std::memory_order_relaxed);  
}  
  
char* Arena::AllocateNewBlock(size_t block_bytes) {  
    char* result = new char[block_bytes];  
    blocks_.push_back(result);  
    memory_usage_.fetch_add(block_bytes + sizeof(char*),  
                             std::memory_order_relaxed);  
    return result;  
}
```

Synchronization and atomic

- ```
template <typename Key, class Comparator>
class SkipList {
 ...
 Node* Next(int n) {
 assert(n >= 0);
 // Use an 'acquire load' so that we observe a fully initialized
 // version of the returned Node.
 return next_[n].load(std::memory_order_acquire);
 }
 void SetNext(int n, Node* x) {
 assert(n >= 0);
 // Use a 'release store' so that anybody who reads through this
 // pointer observes a fully initialized version of the inserted node.
 next_[n].store(x, std::memory_order_release);
 }
}
```
- 

B,

## Other things

- **Synchronization:**

There are a large number of concurrent access scenarios in LevelDB, which requires synchronization support.

Package C++ standard lib(mutex and ConVar)

to port/port\_stdcxx.h

DBImpl::Write()

- **Atomic:**

Relaxed Ordering: atomicity, counting function (eg. arena)

Release-Acquire Ordering: atomicity also execution order

- **Snapshots:**

Snapshots provide consistent read-only views over the entire state of the key-value store.

- **Unit Test:** util/testharness.cc

- **Make**

# Thanks!