

Preface

MINI-LSM

Build a simple key-value storage engine in a week.
And extend your LSM engine in the second + third week.

This course teaches you how to build a simple LSM-Tree storage engine in Rust.

What is LSM, and Why LSM?

Log-structured merge trees are data structures that maintain key-value pairs. This data structure is widely used in distributed database systems like [TiDB](#) and [CockroachDB](#) as their underlying storage

engine. [RocksDB](#), based on [LevelDB](#), is an implementation of LSM-Tree storage engines. It provides many key-value access functionalities and is used in many production systems.

Generally speaking, LSM Tree is an append-friendly data structure. It is more intuitive to compare LSM to other key-value data structures like RB-Tree and B-Tree. For RB-Tree and B-Tree, all data operations are in place. That is to say, when you want to update the value corresponding to the key, the engine will overwrite its original memory or disk space with the new value. But in an LSM Tree, all write operations, i.e., insertions, updates, deletions, are lazily applied to the storage. The engine batches these operations into SST (sorted string table) files and writes them to the disk. Once written to the disk, the engine will not directly modify them. In a particular background task called compaction, the engine will merge these files to apply the updates and deletions.

This architectural design makes LSM trees easy to work with.

1. Data are immutable on persistent storage. Concurrency control is more straightforward. Offloading the background tasks (compaction) to remote servers is possible. Storing and serving data directly from cloud-native storage systems like S3 is also feasible.
2. Changing the compaction algorithm allows the storage engine to balance between read, write, and space amplification. The data structure is versatile, and by adjusting the compaction parameters, we can optimize the LSM structure for different workloads.

This course will teach you how to build an LSM-tree-based storage engine in the Rust programming language.

Prerequisites

- You should know the basics of the Rust programming language. Reading [the Rust book](#) is enough.
- You should know the basic concepts of key-value storage engines, i.e., why we need a complex design to achieve persistence. If you have no experience with database systems and storage

systems before, you can implement Bitcask in [PingCAP Talent Plan](#).

- Knowing the basics of an LSM tree is not a requirement, but we recommend you read something about it, e.g., the overall idea of LevelDB. Knowing them beforehand would familiarize you with concepts like mutable and immutable mem-tables, SST, compaction, WAL, etc.

What should you expect from this course

After taking this course, you should deeply understand how an LSM-based storage system works, gain hands-on experience in designing such systems, and apply what you have learned in your study and career. You will understand the design tradeoffs in such storage systems and find optimal ways to design an LSM-based storage system to meet your workload requirements/goals. This very in-depth course covers all the essential implementation details and design choices of modern storage systems (i.e., RocksDB) based on the author's experience in several LSM-like storage systems, and you will be able to directly apply what you have learned in both industry and academia.

Structure

The course is an extensive course with several parts (weeks). Each week has seven chapters; you can finish each within 2 to 3 hours. The first six chapters of each part will instruct you to build a working system, and the last chapter of each week will be a *snack time* chapter that implements some easy things over what you have built in the previous six days. Each chapter will have required tasks, *check your understanding* questions, and bonus tasks.

Testing

We provide a full test suite and some CLI tools for you to validate if your solution is correct. Note that the test suite is not exhaustive, and your solution might not be 100% correct after passing all test cases. You might need to fix earlier bugs when implementing later parts of the system. We recommend you think thoroughly about your implementation, especially when there are multi-thread operations and race conditions.

Solution

We have a solution that implements all the functionalities as required in the course in the mini-lsm main repo. At the same time, we also have a mini-lsm solution checkpoint repo where each commit corresponds to a chapter in the course.

Keeping such a checkpoint repo up-to-date with the mini-lsm course is challenging because each bug fix or new feature must go through all commits (or checkpoints). Therefore, this repo might not use the latest starter code or incorporate the latest features from the mini-lsm course.

TL;DR: We do not guarantee the solution checkpoint repo contains a correct solution, passes all tests, or has the correct doc comments. For a correct implementation and the solution after implementing everything, please look at the solution in the main repo instead.

<https://github.com/skyzh/mini-lsm/tree/main/mini-lsm>.

If you are stuck at some part of the course or need help determining where to implement functionality, you can refer to this repo for help. You may compare the diff between commits to know what has been changed. You might need to modify some functions in the mini-lsm course multiple times throughout the chapters, and you can understand what exactly is expected to be implemented for each chapter in this repo.

You may access the solution checkpoint repo at <https://github.com/skyzh/mini-lsm-solution-checkpoint>.

Feedbacks

Your feedback is greatly appreciated. We have rewritten the whole course from scratch in 2024 based on the feedback from the students. Please share your learning experience and help us continuously improve the course. Welcome to the [Discord community](#) and share your experience.

The long story of why we rewrote it: The course was originally planned as a general guidance that students start from an empty directory and implement whatever they want based on the specifications we had. We had minimal tests that checked if the behavior was correct. However, the original course was too open-ended, which caused huge obstacles to the learning experience. As students do not have an overview of the whole system beforehand and the instructions are vague, sometimes it is hard for them to know why a design decision is made and what they need to achieve a goal. Some parts of the course were so compact that delivering the expected contents within just one chapter was impossible. Therefore, we completely redesigned the course for an easier learning curve and clearer learning goals. The original one-week course is now split into two weeks (the first week on storage format and the second week on deep-dive compaction), with an extra part on MVCC. We hope you find this course interesting and helpful in your study and career. We want to thank everyone who commented in [Feedback after coding day 1](#) and [Hello, when is the next update plan for the course?](#) -- Your feedback greatly helped us improve the course.

License

The source code of this course is licensed under Apache 2.0, while the book is licensed under CC BY-NC-SA 4.0.

Will this course be free forever?

Yes! Everything publicly available now will be free forever and receive lifetime updates and bug fixes. Meanwhile, we might provide paid code review and office hour services. For the DLC part (*rest of*

your life chapters), we do not have plans to finish them as of 2024 and have yet to decide whether they will be publicly available.

Community

You may join skyzh's Discord server and study with the mini-lsm community.



Get Started

Now, you can get an overview of the LSM structure in [Mini-LSM Course Overview](#).

About the Author

As of writing (at the beginning of 2024), Chi obtained his master's degree in Computer Science from Carnegie Mellon University and his bachelor's degree from Shanghai Jiao Tong University. He has been working on a variety of database systems, including [TiKV](#), [AgateDB](#), [TerarkDB](#), [RisingWave](#), and [Neon](#). Since 2022, he has worked as a teaching assistant for [CMU's Database Systems course](#) for three semesters on the BusTub educational system, where he added a lot of new features and more challenges to the course (check out the redesigned [query execution](#) project and the super challenging [multi-version concurrency control](#) project). Besides working on the BusTub educational system, he also maintains the [RisingLight](#) educational database system. Chi is interested in exploring how the Rust programming language can fit into the database world. Check out his previous course

on building a vectorized expression framework [type-exercise-in-rust](#) and on building a vector database [write-you-a-vector-db](#) if you are also interested in that topic.

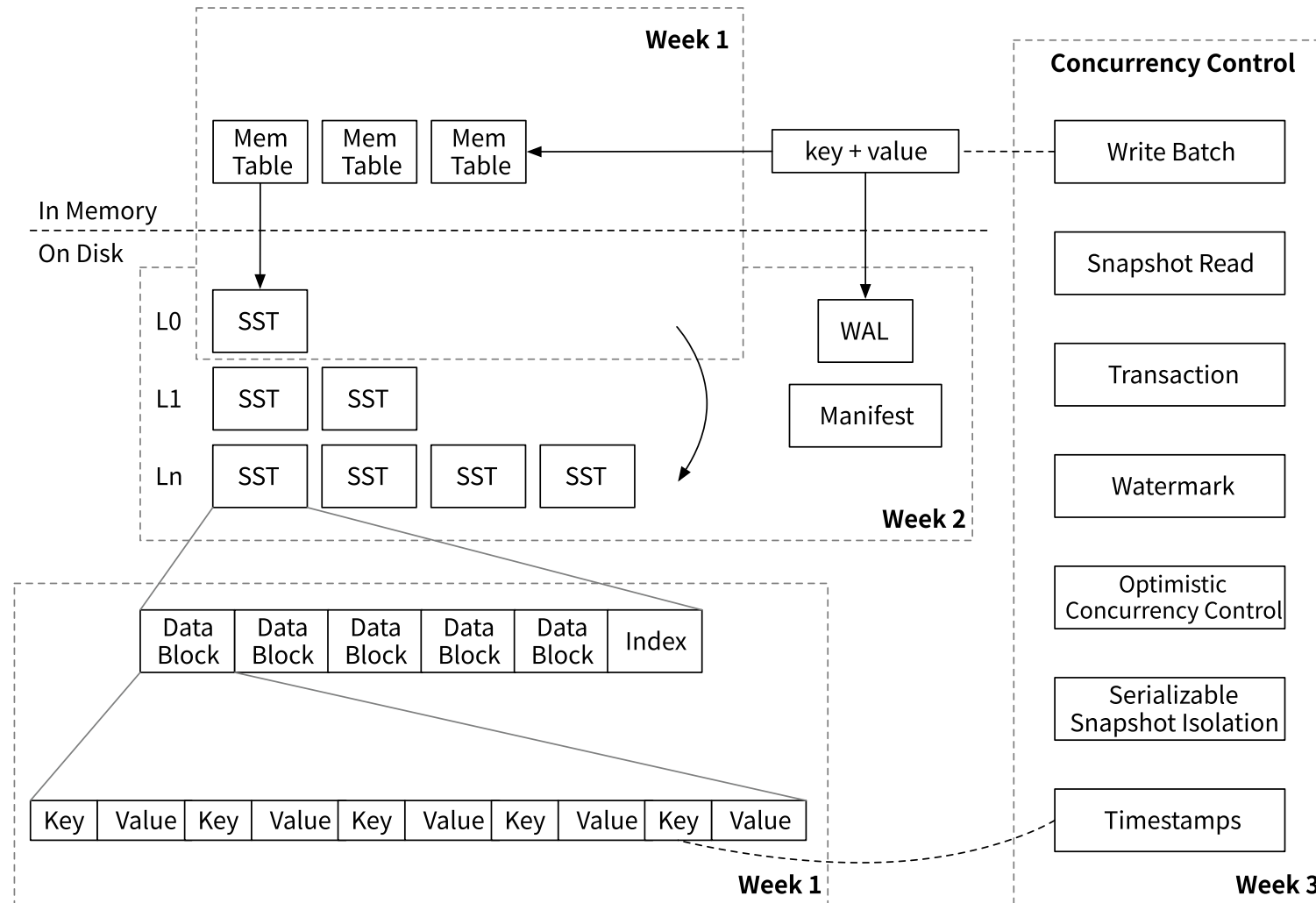
Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Mini-LSM Course Overview

Course Structure



We have three parts (weeks) for this course. In the first week, we will focus on the storage structure and the storage format of an LSM storage engine. In the second week, we will deeply dive into compactions and implement persistence support for the storage engine. In the third week, we will implement multi-version concurrency control.

- [The First Week: Mini-LSM](#)
- [The Second Week: Compaction and Persistence](#)
- [The Third Week: Multi-Version Concurrency Control](#)

Please look at [Environment Setup](#) to set up the environment.

Overview of LSM

An LSM storage engine generally contains three parts:

1. Write-ahead log to persist temporary data for recovery.
2. SSTs on the disk to maintain an LSM-tree structure.
3. Mem-tables in memory for batching small writes.

The storage engine generally provides the following interfaces:

- `Put(key, value)` : store a key-value pair in the LSM tree.
- `Delete(key)` : remove a key and its corresponding value.
- `Get(key)` : get the value corresponding to a key.
- `Scan(range)` : get a range of key-value pairs.

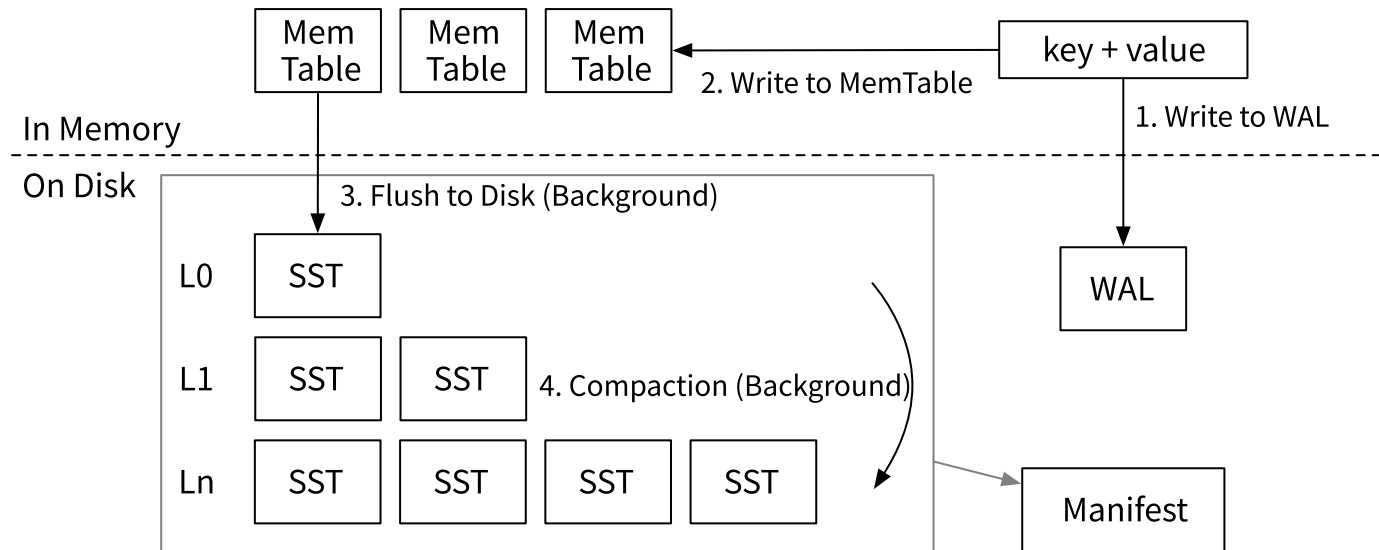
To ensure persistence,

- `Sync()` : ensure all the operations before `sync` are persisted to the disk.

Some engines choose to combine `Put` and `Delete` into a single operation called `WriteBatch`, which accepts a batch of key-value pairs.

In this course, we assume the LSM tree is using a leveled compaction algorithm, which is commonly used in real-world systems.

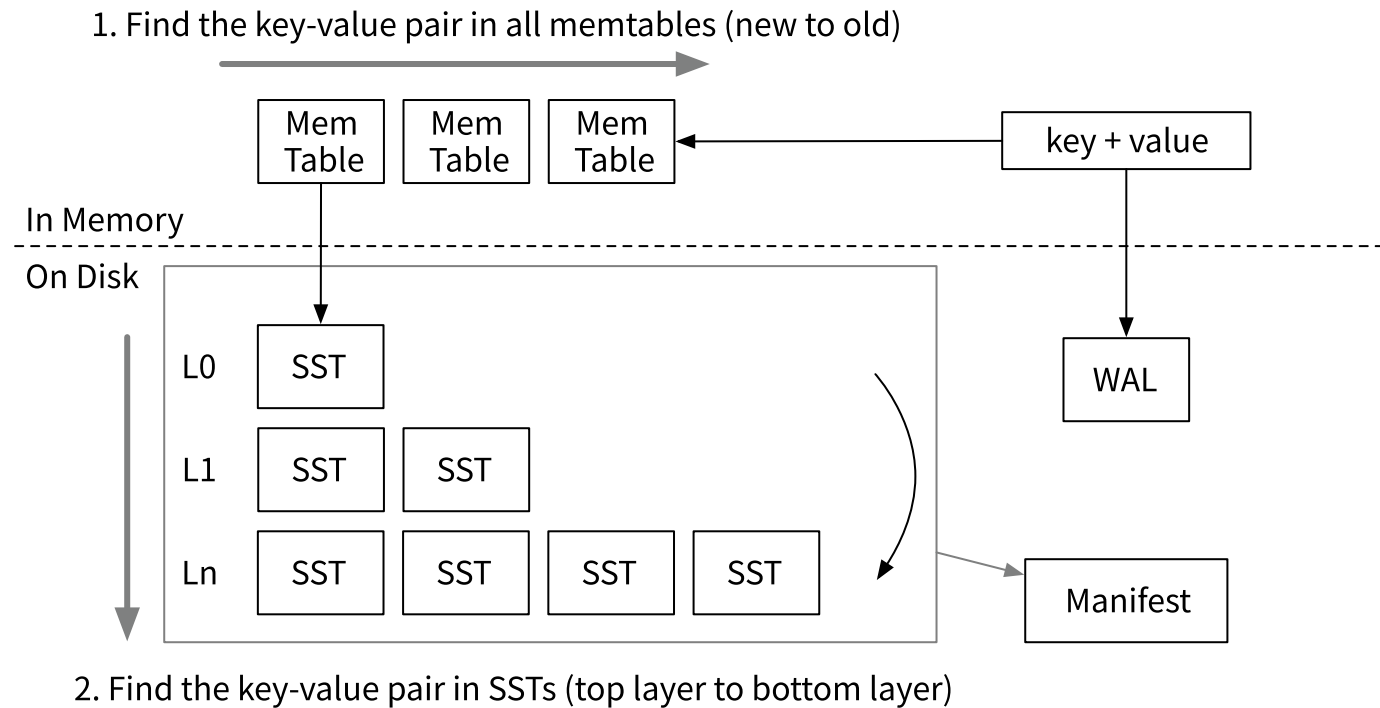
Write Path



The write path of LSM contains four steps:

1. Write the key-value pair to the write-ahead log so that it can be recovered after the storage engine crashes.
2. Write the key-value pair to memtable. After (1) and (2) are completed, we can notify the user that the write operation is completed.
3. (In the background) When a mem-table is full, we will freeze them into immutable mem-tables and flush them to the disk as SST files in the background.
4. (In the background) The engine will compact some files in some levels into lower levels to maintain a good shape for the LSM tree so that the read amplification is low.

Read Path



When we want to read a key,

1. We will first probe all the mem-tables from the latest to the oldest.
2. If the key is not found, we will then search the entire LSM tree containing SSTs to find the data.

There are two types of read: lookup and scan. Lookup finds one key in the LSM tree, while scan iterates all keys within a range in the storage engine. We will cover both of them throughout the course.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Environment Setup

The starter code and reference solution is available at <https://github.com/skyzh/mini-lsm>.

Install Rust

See <https://rustup.rs> for more information.

Clone the repo

```
git clone https://github.com/skyzh/mini-lsm
```

Starter code

```
cd mini-lsm/mini-lsm-starter  
code .
```

Install Tools

You will need the latest stable Rust to compile this project. The minimum requirement is 1.74 .

```
cargo x install-tools
```

Run tests

```
cargo x copy-test --week 1 --day 1  
cargo x scheck
```

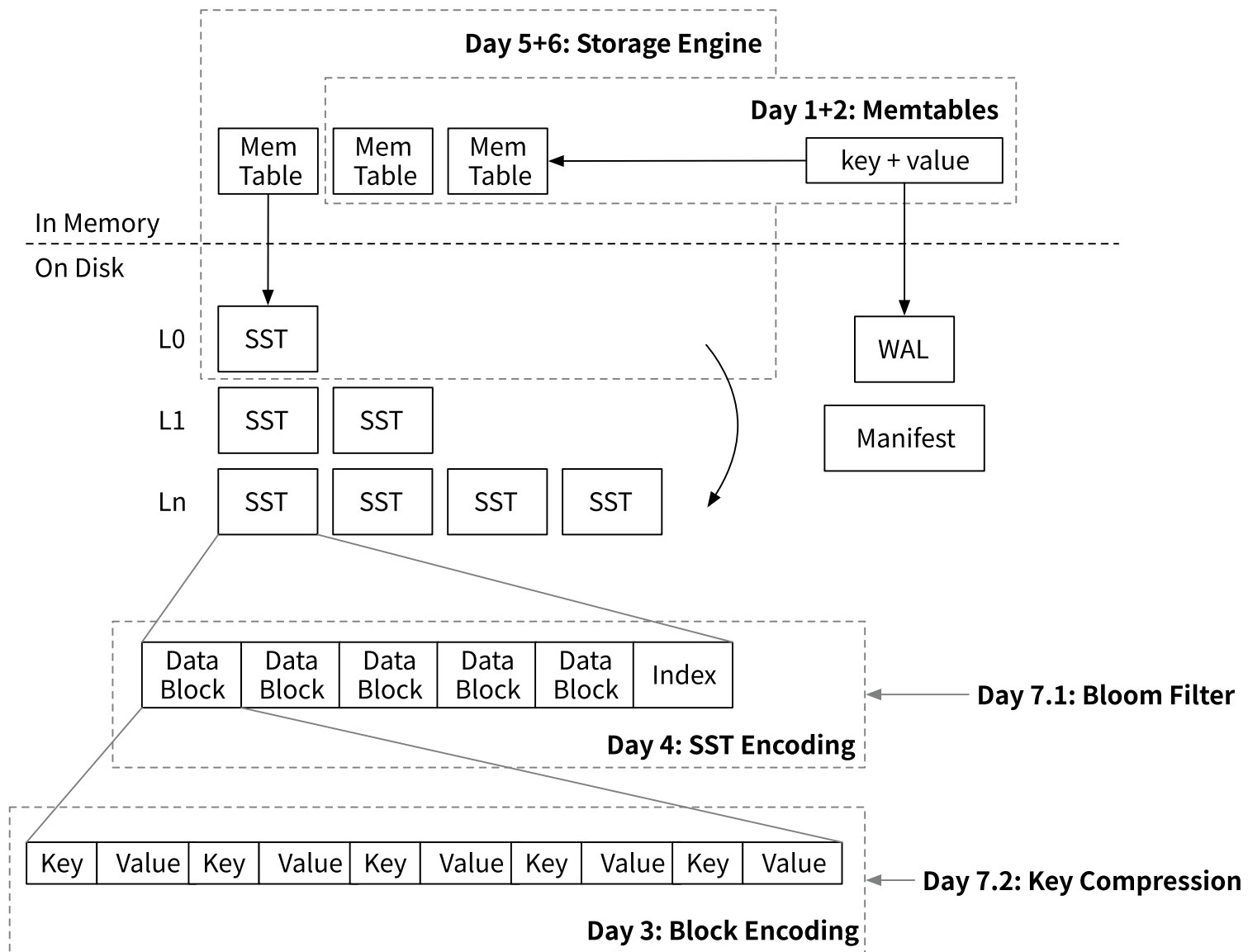
Now, you can go ahead and start [Week 1: Mini-LSM](#).

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Week 1 Overview: Mini-LSM



In the first week of the course, you will build necessary storage formats for the storage engine, the read path and the write path of the system, and have a working implementation of an LSM-based key-value store. There are 7 chapters (days) for this part.

- [Day 1: Memtable](#). You will implement the in-memory read and write path of the system.
- [Day 2: Merge Iterator](#). You will extend what you have built in day 1 and implement a `scan` interface for your system.
- [Day 3: Block Encoding](#). Now we start the first step of the on-disk structure and build the encoding/decoding of the blocks.
- [Day 4: SST Encoding](#). SSTs are composed of blocks and at the end of the day, you will have the basic building blocks of the LSM on-disk structure.
- [Day 5: Read Path](#). Now that we have both in-memory and on-disk structures, we can combine them together and have a fully-working read path for the storage engine.
- [Day 6: Write Path](#). In day 5, the test harness generates the structures, and in day 6, you will control the SST flushes by yourself. You will implement flush to level-0 SST and the storage engine is complete.
- [Day 7: SST Optimizations](#). We will implement several SST format optimizations and improve the performance of the system.

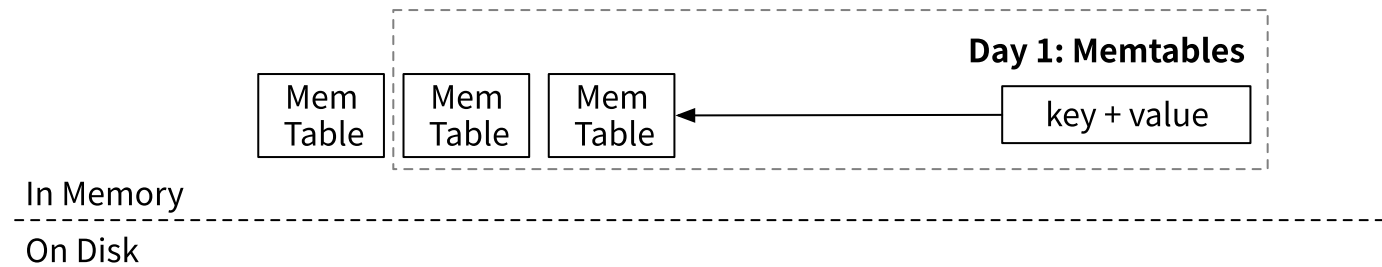
At the end of the week, your storage engine should be able to handle all get/scan/put requests. The only missing parts are persisting the LSM state to disk and a more efficient way of organizing the SSTs on the disk. You will have a working **Mini-LSM** storage engine.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Memtables



In this chapter, you will:

- Implement memtables based on skiplists.
- Implement freezing memtable logic.
- Implement LSM read path `get` for memtables.

To copy the test cases into the starter code and run them,

```
cargo x copy-test --week 1 --day 1
cargo x scheck
```

Task 1: SkipList Memtable

In this task, you will need to modify:

```
src/mem_table.rs
```

Firstly, let us implement the in-memory structure of an LSM storage engine -- the memtable. We choose [crossbeam's skiplist implementation](#) as the data structure of the memtable as it supports

lock-free concurrent read and write. We will not cover in-depth how a skiplist works, and in a nutshell, it is an ordered key-value map that easily allows concurrent read and write.

crossbeam-skiplist provides similar interfaces to the Rust std's `BTreeMap`: `insert`, `get`, and `iter`. The only difference is that the modification interfaces (i.e., `insert`) only require an immutable reference to the skiplist, instead of a mutable one. Therefore, in your implementation, you should not take any mutex when implementing the memtable structure.

You will also notice that the `MemTable` structure does not have a `delete` interface. In the mini-lsm implementation, deletion is represented as a key corresponding to an empty value.

In this task, you will need to implement `MemTable::get` and `MemTable::put` to enable modifications of the memtable. Note that `put` should always overwrite a key if it already exists. You won't have multiple entries of the same key in a single memtable.

We use the `bytes` crate for storing the data in the memtable. `bytes::Byte` is similar to `Arc<[u8]>`. When you clone the `Bytes`, or get a slice of `Bytes`, the underlying data will not be copied, and therefore cloning it is cheap. Instead, it simply creates a new reference to the storage area and the storage area will be freed when there are no reference to that area.

Task 2: A Single Memtable in the Engine

In this task, you will need to modify:

```
src/lsm_storage.rs
```

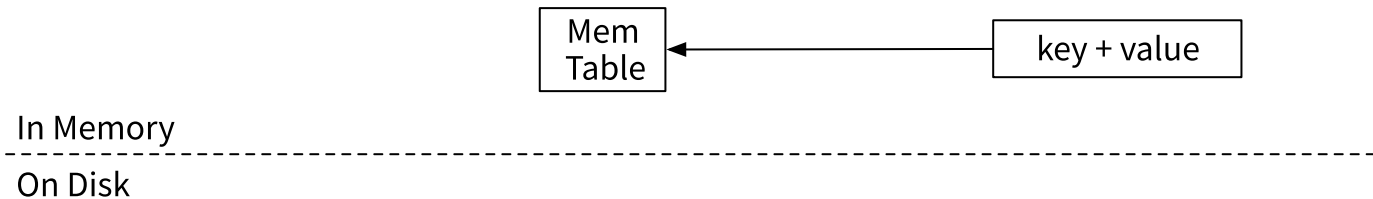
Now, we will add our first data structure, the memtable, to the LSM state. In

`LsmStorageState::create`, you will find that when a LSM structure is created, we will initialize a memtable of id 0. This is the **mutable memtable** in the initial state. At any point of the time, the

engine will have only one single mutable memtable. A memtable usually has a size limit (i.e., 256MB), and it will be frozen to an immutable memtable when it reaches the size limit.

Taking a look at `lsm_storage.rs`, you will find there are two structures that represents a storage engine: `MiniLSM` and `LsmStorageInner`. `MiniLSM` is a thin wrapper for `LsmStorageInner`. You will implement most of the functionalities in `LsmStorageInner`, until week 2 compaction.

`LsmStorageState` stores the current structure of the LSM storage engine. For now, we will only use the `memtable` field, which stores the current mutable memtable. In this task, you will need to implement `LsmStorageInner::get`, `LsmStorageInner::put`, and `LsmStorageInner::delete`. All of them should directly dispatch the request to the current memtable.



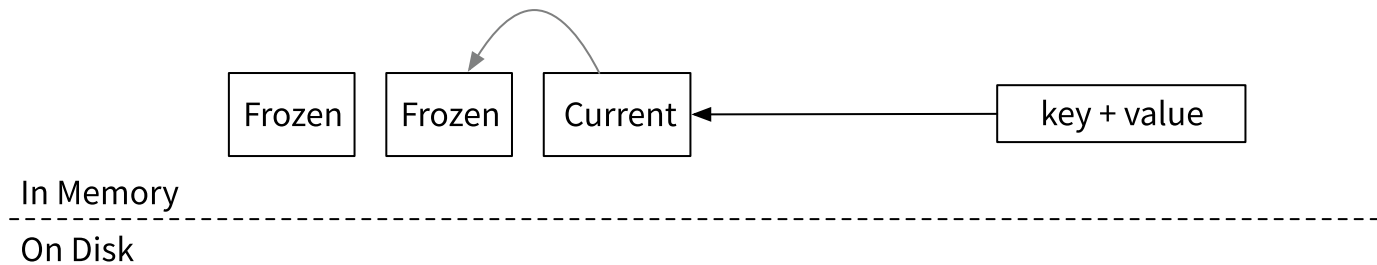
Your `delete` implementation should simply put an empty slice for that key, and we call it a *delete tombstone*. Your `get` implementation should handle this case correspondingly.

To access the memtable, you will need to take the `state` lock. As our memtable implementation only requires an immutable reference for `put`, you ONLY need to take the read lock on `state` in order to modify the memtable. This allows concurrent access to the memtable from multiple threads.

Task 3: Write Path - Freezing a Memtable

In this task, you will need to modify:

src/lsm_storage.rs
src/mem_table.rs



A memtable cannot continuously grow in size, and we will need to freeze them (and later flush to the disk) when it reaches the size limit. You may find the memtable size limit, which is **equal to the SST size limit** (not `num_memtables_limit`), in the `LsmStorageOptions`. This is not a hard limit and you should freeze the memtable at best effort.

In this task, you will need to compute the approximate memtable size when put/delete a key in the memtable. This can be computed by simply adding the total number of bytes of keys and values when `put` is called. If a key is put twice, though the skiplist only contains the latest value, you may count it twice in the approximate memtable size. Once a memtable reaches the limit, you should call `force_freeze_memtable` to freeze the memtable and create a new one.

The `state: Arc<RwLock<Arc<LsmStorageState>>>` field in `LsmStorageInner` is structured this way to manage the LSM tree's overall state concurrently and safely, primarily using a Copy-on-Write (CoW) strategy:

1. Inner `Arc<LsmStorageState>`: This holds an **immutable snapshot** of the actual `LsmStorageState` (which contains memtable lists, SST references, etc.). Cloning this `Arc` is very cheap (just an atomic reference count increment) and gives any reader a consistent, unchanging view of the state for the duration of their operation.
2. `RwLock<Arc<LsmStorageState>>`: This read-write lock protects the *pointer* to the current `Arc<LsmStorageState>` (the active snapshot).

- **Readers** acquire a read lock, clone the `Arc<LsmStorageState>` (getting their own reference to the current snapshot), and then quickly release the read lock. They can then work with their snapshot without further locking.
- **Writers** (when modifying the state, e.g., freezing a memtable) will:
 - Create a *new* `LsmStorageState` instance, often by cloning the data from the current snapshot and then applying modifications.
 - Wrap this new state in a new `Arc<LsmStorageState>`.
 - Acquire the write lock on the `RwLock`.
 - Replace the old `Arc<LsmStorageState>` with the new one.
 - Release the write lock.

3. Outer `Arc<RwLock<...>>` : This allows the `RwLock` itself (and thus the mechanism for accessing and updating the state) to be shared safely across multiple threads or parts of your application that might need to interact with `LsmStorageInner`.

This CoW approach ensures that readers always see a valid, consistent state snapshot and experience minimal blocking. Writers update the state atomically by swapping out the entire state snapshot, reducing the time critical locks are held and thus improving concurrency.

Because there could be multiple threads getting data into the storage engine, `force_freeze_memtable` might be called concurrently from multiple threads. You will need to think about how to avoid race conditions in this case.

There are multiple places where you may want to modify the LSM state: freeze a mutable memtable, flush memtable to SST, and GC/compaction. During all of these modifications, there could be I/O operations. An intuitive way to structure the locking strategy is to:

```
fn freeze_memtable(&self) {
    let state = self.state.write();
    state.immutable_memtable.push(/* something */);
    state.memtable = MemTable::create();
}
```

...that you modify everything in LSM state's write lock.

This works fine for now. However, consider the case where you want to create a write-ahead log file for every memtables you have created.

```
fn freeze_memtable(&self) {  
    let state = self.state.write();  
    state.immutable_memtable.push(/* something */);  
    state.memtable = MemTable::create_with_wal()?; // <- could take several  
    milliseconds  
}
```

Now when we freeze the memtable, no other threads could have access to the LSM state for several milliseconds, which creates a spike of latency.

To solve this problem, we can put I/O operations outside of the lock region.

```
fn freeze_memtable(&self) {  
    let memtable = MemTable::create_with_wal()?; // <- could take several milliseconds  
    {  
        let state = self.state.write();  
        state.immutable_memtable.push(/* something */);  
        state.memtable = memtable;  
    }  
}
```

Then, we do not have costly operations within the state write lock region. Now, consider the case that the memtable is about to reach the capacity limit and two threads successfully put two keys into the memtable, both of them discovering the memtable reaches capacity limit after putting the two keys. They will both do a size check on the memtable and decide to freeze it. In this case, we might create one empty memtable which is then immediately frozen.

To solve the problem, all state modification should be synchronized through the state lock.

```

fn put(&self, key: &[u8], value: &[u8]) {
    // put things into the memtable, checks capacity, and drop the read lock on LSM
    state
    if memtable_reaches_capacity_on_put {
        let state_lock = self.state_lock.lock();
        if /* check again current memtable reaches capacity */ {
            self.freeze_memtable(&state_lock)?;
        }
    }
}

```

You will notice this kind of pattern very often in future chapters. For example, for L0 flush,

```

fn force_flush_next_imm_memtable(&self) {
    let state_lock = self.state_lock.lock();
    // get the oldest memtable and drop the read lock on LSM state
    // write the contents to the disk
    // get the write lock on LSM state and update the state
}

```

This ensures only one thread will be able to modify the LSM state while still allowing concurrent access to the LSM storage.

In this task, you will need to modify `put` and `delete` to respect the soft capacity limit on the memtable. When it reaches the limit, call `force_freeze_memtable` to freeze the memtable. Note that we do not have test cases over this concurrent scenario, and you will need to think about all possible race conditions on your own. Also, remember to check lock regions to ensure the critical sections are the minimum required.

You can simply assign the next memtable id as `self.next_sst_id()`. Note that the `imm_memtables` stores the memtables from the latest one to the earliest one. That is to say, `imm_memtables.first()` should be the last frozen memtable.

Task 4: Read Path - Get

In this task, you will need to modify:

```
src/lsm_storage.rs
```

Now that you have multiple memtables, you may modify your read path `get` function to get the latest version of a key. Ensure that you probe the memtables from the latest one to the earliest one.

Test Your Understanding

- Why doesn't the memtable provide a `delete` API?
- Does it make sense for the memtable to store all write operations instead of only the latest version of a key? For example, the user puts `a->1`, `a->2`, and `a->3` into the same memtable.
- Is it possible to use other data structures as the memtable in LSM? What are the pros/cons of using the skiplist?
- Why do we need a combination of `state` and `state_lock`? Can we only use `state.read()` and `state.write()`?
- Why does the order to store and to probe the memtables matter? If a key appears in multiple memtables, which version should you return to the user?
- Is the memory layout of the memtable efficient / does it have good data locality? (Think of how `Byte` is implemented and stored in the skiplist...) What are the possible optimizations to make the memtable more efficient?
- So we are using `parking_lot` locks in this course. Is its read-write lock a fair lock? What might happen to the readers trying to acquire the lock if there is one writer waiting for existing readers to stop?
- After freezing the memtable, is it possible that some threads still hold the old LSM state and wrote into these immutable memtables? How does your solution prevent it from happening?

- There are several places that you might first acquire a read lock on state, then drop it and acquire a write lock (these two operations might be in different functions but they happened sequentially due to one function calls the other). How does it differ from directly upgrading the read lock to a write lock? Is it necessary to upgrade instead of acquiring and dropping and what is the cost of doing the upgrade?

We do not provide reference answers to the questions, and feel free to discuss about them in the Discord community.

Bonus Tasks

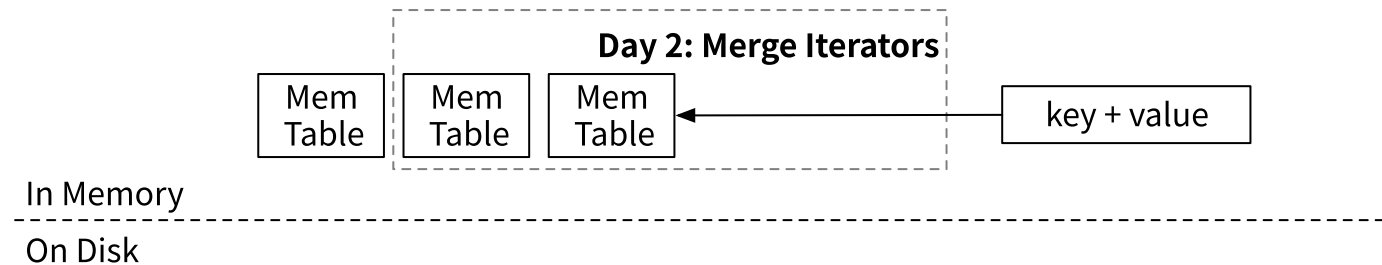
- **More Memtable Formats.** You may implement other memtable formats. For example, BTree memtable, vector memtable, and ART memtable.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Merge Iterator



In this chapter, you will:

- Implement memtable iterator.
- Implement merge iterator.
- Implement LSM read path `scan` for memtables.

To copy the test cases into the starter code and run them,

```
cargo x copy-test --week 1 --day 2
cargo x scheck
```

Task 1: Memtable Iterator

In this chapter, we will implement the LSM `scan` interface. `scan` returns a range of key-value pairs in order using an iterator API. In the previous chapter, you have implemented the `get` API and the logic to create immutable memtables, and your LSM state should now have multiple memtables. You will need to first create iterators on a single memtable, then create a merge iterator on all memtables, and finally implement the range limit for the iterators.

In this task, you will need to modify:

```
src/mem_table.rs
```

All LSM iterators implement the `StorageIterator` trait. It has 4 functions: `key`, `value`, `next`, and `is_valid`. If you're familiar with Rust's standard library `Iterator` trait, you might find `StorageIterator` a bit different. Instead, `StorageIterator` employs a cursor-based API, a design pattern common in database systems and notably inspired by RocksDB's iterators (see [iterator_base.h](#) and [iterator.h](#) for reference).

When the iterator is created, its cursor will stop on some element, and `key` / `value` will return the first key in the memtable/block/SST satisfying the start condition (i.e., start key). These two interfaces will return a `&[u8]` to avoid copy.

From the caller's perspective, the typical usage pattern is:

```
let mut iter: impl StorageIterator = ...;
while iter.is_valid() {
    let key = iter.key();
    let value = iter.value();
    // Process key and value
    iter.next()?; // Advance to the next item, handling potential errors
}
```

The semantics of `StorageIterator` are distinct for its core methods:

- `next()` : This method is solely responsible for attempting to move the cursor to the next element. It returns a `Result` to report any errors encountered during this advancement (e.g., I/O issues). It does *not* inherently guarantee that the new position is valid, only that the attempt to move was made.
- `is_valid()` : This method indicates whether the iterator's current cursor points to a valid data element. It does *not* advance the iterator.

Therefore, as an implementer of `StorageIterator`, after each call to `next()` (even if it succeeds without an error from the `next()` operation itself), you are responsible for updating the internal state so that `is_valid()` correctly reflects whether the new cursor position actually points to a valid item.

In summary, `next` moves the cursor to the next place. `is_valid` returns if the iterator has reached the end or errored. You can assume `next` will only be called when `is_valid` returns true. There will be a `FusedIterator` wrapper for iterators that block calls to `next` when the iterator is not valid to avoid users from misusing the iterators.

Back to the memtable iterator. You should have found out that the iterator does not have any lifetime associated with that. Imagine that you create a `Vec<u64>` and call `vec.iter()`, the iterator type will be something like `VecIterator<'a>`, where `'a` is the lifetime of the `vec` object. The same applies to `SkipMap`, where its `iter` API returns an iterator with a lifetime. However, in our case, we do not want to have such lifetimes on our iterators to avoid making the system overcomplicated (and hard to compile...).

If the iterator does not have a lifetime generics parameter, we should ensure that *whenever the iterator is being used, the underlying skiplist object is not freed*. The only way to achieve that is to put the `Arc<SkipMap>` object into the iterator itself. To define such a structure,

```
pub struct MemtableIterator {
    map: Arc<SkipMap<Bytes, Bytes>>,
    iter: SkipMapRangeIter<'???'>,
}
```

Okay, here is the problem: we want to express that the lifetime of the iterator is the same as the `map` in the structure. How can we do that?

This is the first and most tricky Rust language thing that you will ever meet in this course -- self-referential structure. If it is possible to write something like:

```
pub struct MemtableIterator { // <- with lifetime 'this
    map: Arc<SkipMap<Bytes, Bytes>>,
    iter: SkipMapRangeIter<'this>,
}
```

Then the problem is solved! You can do this with the help of some third-party libraries like `ouroboros`. It provides an easy way to define self-referential structure. It is also possible to do this with unsafe Rust (and indeed, `ouroboros` itself uses unsafe Rust internally...)

We have leveraged `ouroboros` to define the self-referential `MemtableIterator` fields for you. You will need to implement the `MemtableIterator` logic and the `Memtable::scan` API based on this provided structure.

Task 2: Merge Iterator

In this task, you will need to modify:

```
src/iterators/merge_iterator.rs
```

Now that you have multiple memtables and you will create multiple memtable iterators. You will need to merge the results from the memtables and return the latest version of each key to the user.

`MergeIterator` maintains a binary heap internally. Consider the challenge of merging n sorted sequences (our iterators) into a single sorted output; a binary heap is a natural fit here, as it efficiently helps identify which sequence currently holds the overall smallest element. You'll see that the ordering of the binary heap is such that the iterator with the lowest head key value is first. When multiple iterators have the same head key value, the newest one is first. Note that you will need to handle errors (i.e., when an iterator is not valid) and ensure that the latest version of a key-value pair comes out.

For example, if we have the following data:

```
iter1: b->del, c->4, d->5  
iter2: a->1, b->2, c->3  
iter3: e->4
```

The sequence that the merge iterator outputs should be:

```
a->1, b->del, c->4, d->5, e->4
```

The constructor of the merge iterator takes a vector of iterators. We assume the one with a lower index (i.e., the first one) has the latest data.

When using the Rust binary heap, you may find the `peek_mut` function useful.

```
let Some(mut inner) = heap.peek_mut() {  
    *inner += 1; // <- do some modifications to the inner item  
}  
// When the PeekMut reference gets dropped, the binary heap gets reordered  
// automatically.  
  
let Some(mut inner) = heap.peek_mut() {  
    PeekMut::pop(inner) // <- pop it out from the heap  
}
```

One common pitfall is on error handling. For example,

```
let Some(mut inner_iter) = self.iters.peek_mut() {  
    inner_iter.next()?; // <- will cause problem  
}
```

If `next` returns an error (i.e., due to disk failure, network failure, checksum error, etc.), it is no longer valid. However, when we go out of the if condition and return the error to the caller, `PeekMut`'s drop

will try move the element within the heap, which causes an access to an invalid iterator. Therefore, you will need to do all error handling by yourself instead of using `?` within the scope of `PeekMut`.

We want to avoid dynamic dispatch as much as possible, and therefore we do not use `Box<dyn StorageIterator>` in the system. Instead, we prefer static dispatch using generics. Also note that `StorageIterator` uses generic associated type (GAT), so that it can support both `KeySlice` and `&[u8]` as the key type. We will change `KeySlice` to include the timestamp in week 3 and using a separate type for it now can make the transition more smooth.

Starting this section, we will use `Key<T>` to represent LSM key types and distinguish them from values in the type system. You should use provided APIs of `Key<T>` instead of directly accessing the inner value. We will add timestamp to this key type in part 3, and using the key abstraction will make the transition more smooth. For now, `KeySlice` is equivalent to `&[u8]`, `KeyVec` is equivalent to `Vec<u8>`, and `KeyBytes` is equivalent to `Bytes`.

Task 3: LSM Iterator + Fused Iterator

In this task, you will need to modify:

```
src/lsm_iterator.rs
```

We use the `LsmIterator` structure to represent the internal LSM iterators. You will need to modify this structure multiple times throughout the course when more iterators are added into the system. For now, because we only have multiple memtables, it should be defined as:

```
type LsmIteratorInner = MergeIterator<MemTableIterator>;
```

You may go ahead and implement the `LsmIterator` structure, which calls the corresponding inner iterator, and also skip deleted keys.

We do not test `LsmIterator` in this task. There will be an integration test in task 4.

Then, we want to provide extra safety on the iterator to avoid users from misusing them. Users should not call `key`, `value`, or `next` when the iterator is not valid. At the same time, they should not use the iterator anymore if `next` returns an error. `FusedIterator` is a wrapper around an iterator to normalize the behaviors across all iterators. You can go ahead and implement it by yourself.

Task 4: Read Path - Scan

In this task, you will need to modify:

```
src/lsm_storage.rs
```

We are finally there -- with all iterators you have implemented, you can finally implement the `scan` interface of the LSM engine. You can simply construct an LSM iterator with the memtable iterators (remember to put the latest memtable at the front of the merge iterator), and your storage engine will be able to handle the scan request.

Test Your Understanding

- What is the time/space complexity of using your merge iterator?
- Why do we need a self-referential structure for memtable iterator?
- If a key is removed (there is a delete tombstone), do you need to return it to the user? Where did you handle this logic?
- If a key has multiple versions, will the user see all of them? Where did you handle this logic?

- If we want to get rid of self-referential structure and have a lifetime on the memtable iterator (i.e., `MemtableIterator<'a>`, where `'a` = memtable or `LsmStorageInner` lifetime), is it still possible to implement the `scan` functionality?
- What happens if (1) we create an iterator on the skiplist memtable (2) someone inserts new keys into the memtable (3) will the iterator see the new key?
- What happens if your key comparator cannot give the binary heap implementation a stable order?
- Why do we need to ensure the merge iterator returns data in the iterator construction order?
- Is it possible to implement a Rust-style iterator (i.e., `next(&self) -> (Key, Value)`) for LSM iterators? What are the pros/cons?
- The scan interface is like `fn scan(&self, lower: Bound<&[u8]>, upper: Bound<&[u8]>)`. How to make this API compatible with Rust-style range (i.e., `key_a..key_b`)? If you implement this, try to pass a full range `..` to the interface and see what will happen.
- The starter code provides the merge iterator interface to store `Box<I>` instead of `I`. What might be the reason behind that?

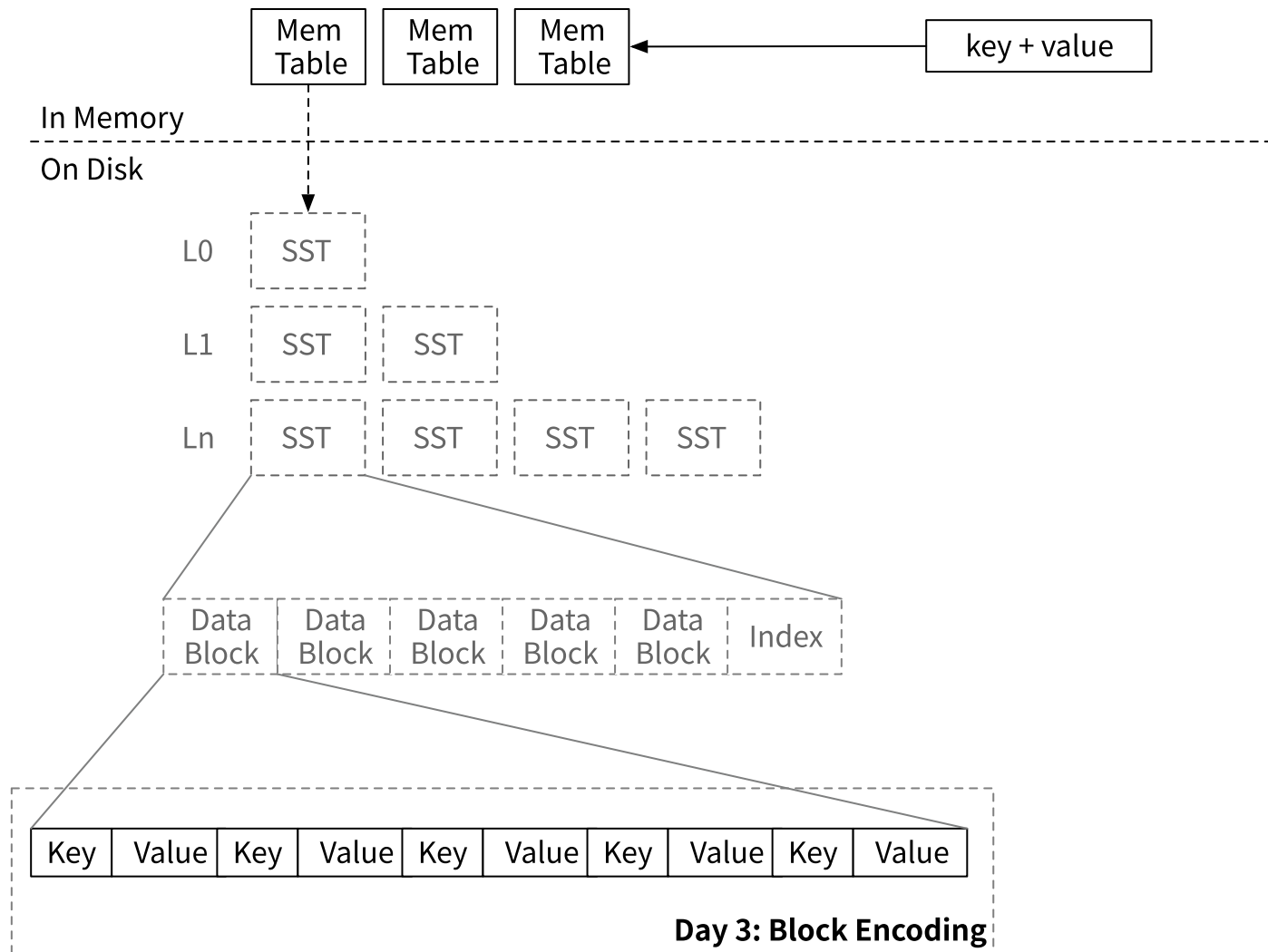
We do not provide reference answers to the questions, and feel free to discuss about them in the Discord community.

Bonus Tasks

- **Foreground Iterator.** In this course we assumed that all operations are short, so that we can hold reference to mem-table in the iterator. If an iterator is held by users for a long time, the whole mem-table (which might be 256MB) will stay in the memory even if it has been flushed to disk. To solve this, we can provide a `ForegroundIterator` / `LongIterator` to our user. The iterator will periodically create new underlying storage iterator so as to allow garbage collection of the resources.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).
Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.
mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Block



In this chapter, you will:

- Implement SST block encoding.
- Implement SST block decoding and block iterator.

To copy the test cases into the starter code and run them,

```
cargo x copy-test --week 1 --day 3  
cargo x scheck
```

Task 1: Block Builder

You have already implemented all in-memory structures for an LSM storage engine in the previous two chapters. Now it's time to build the on-disk structures. The basic unit of the on-disk structure is blocks. Blocks are usually of 4-KB size (the size may vary depending on the storage medium), which is equivalent to the page size in the operating system and the page size on an SSD. A block stores ordered key-value pairs. An SST is composed of multiple blocks. When the number of memtables exceed the system limit, it will flush the memtable as an SST. In this chapter, you will implement the encoding and decoding of a block.

In this task, you will need to modify:

```
src/block/builder.rs  
src/block.rs
```

The block encoding format in our course is as follows:

	Data Section		Offset Section
Extra			

	Entry #1		Entry #2
	...		Entry #N
	num_of_elements		Offset #1
			Offset #2
			...
			Offset #N

Each entry is a key-value pair.

	Entry #1			...

	key_len (2B)	key (keylen)	value_len (2B)	value (varlen) ...

Key length and value length are both 2 bytes, which means their maximum lengths are 65535.
(Internally stored as `u16`)

We assume that keys will never be empty, and values can be empty. An empty value means that the corresponding key has been deleted in the view of other parts of the system. For the `BlockBuilder` and `BlockIterator` , we just treat the empty value as-is.

At the end of each block, we will store the offsets of each entry and the total number of entries. For example, if the first entry is at 0th position of the block, and the second entry is at 12th position of the block.

offset		offset	num_of_elements

	0		12
			2

The footer of the block will be as above. Each of the number is stored as `u16` .

The block has a size limit, which is `target_size` . Unless the first key-value pair exceeds the target block size, you should ensure that the encoded block size is always less than or equal to `target_size` . (In the provided code, the `target_size` here is essentially the `block_size`)

The `BlockBuilder` will produce the data part and unencoded entry offsets when `build` is called. The information will be stored in the `Block` structure. As key-value entries are stored in raw format and offsets are stored in a separate vector, this reduces unnecessary memory allocations and processing overhead when decoding data — what you need to do is to simply copy the raw block data to the `data` vector and decode the entry offsets every 2 bytes, *instead of* creating something like `Vec<(Vec<u8>, Vec<u8>)>` to store all the key-value pairs in one block in memory. This compact memory layout is very efficient.

In `Block::encode` and `Block::decode` , you will need to encode/decode the block in the format as indicated above.

Task 2: Block Iterator

In this task, you will need to modify:

`src/block/iterator.rs`

Now that we have an encoded block, we will need to implement the `BlockIterator` interface, so that the user can lookup/scan keys in the block.

`BlockIterator` can be created with an `Arc<Block>`. If `create_and_seek_to_first` is called, it will be positioned at the first key in the block. If `create_and_seek_to_key` is called, the iterator will be positioned at the first key that is `>=` the provided key. For example, if 1, 3, 5 is in a block.

```
let mut iter = BlockIterator::create_and_seek_to_key(block, b"2");
assert_eq!(iter.key(), b"3");
```

The above `seek 2` will make the iterator to be positioned at the next available key of 2, which in this case is 3.

The iterator should copy `key` from the block and store them inside the iterator (we will have key compression in the future and you will have to do so). For the value, you should only store the begin/end offset in the iterator without copying them.

When `next` is called, the iterator will move to the next position. If we reach the end of the block, we can set `key` to empty and return `false` from `is_valid`, so that the caller can switch to another block if possible.

Test Your Understanding

- What is the time complexity of seeking a key in the block?
- Where does the cursor stop when you seek a non-existent key in your implementation?
- So `Block` is simply a vector of raw data and a vector of offsets. Can we change them to `Byte` and `Arc<[u16]>`, and change all the iterator interfaces to return `Byte` instead of `&[u8]`? (Assume that we use `Byte::slice` to return a slice of the block without copying.) What are the pros/cons?

- What is the endian of the numbers written into the blocks in your implementation?
- Is your implementation prone to a maliciously-built block? Will there be invalid memory access, or OOMs, if a user deliberately construct an invalid block?
- Can a block contain duplicated keys?
- What happens if the user adds a key larger than the target block size?
- Consider the case that the LSM engine is built on object store services (S3). How would you optimize/change the block format and parameters to make it suitable for such services?
- Do you love bubble tea? Why or why not?

We do not provide reference answers to the questions, and feel free to discuss about them in the Discord community.

Bonus Tasks

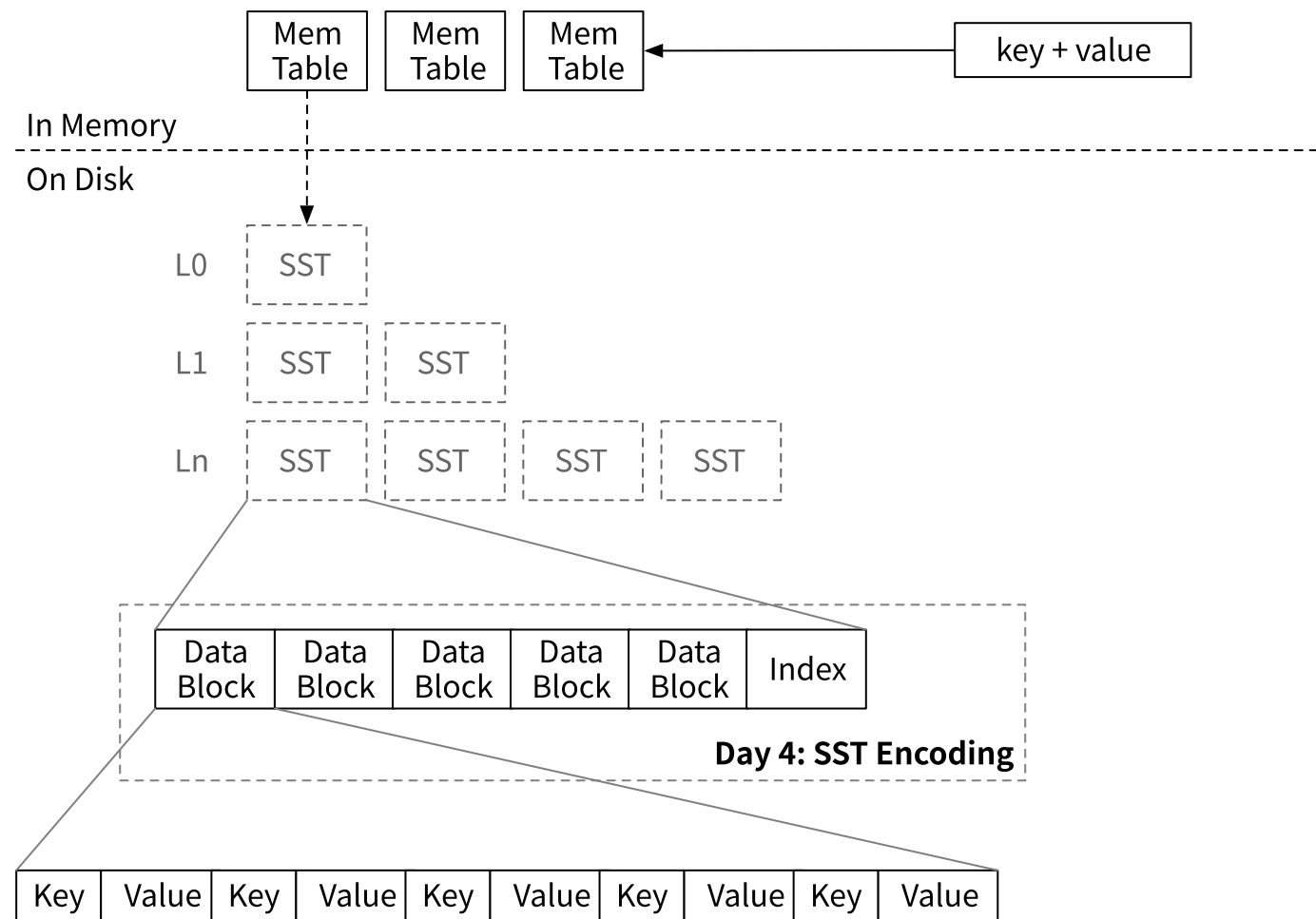
- **Backward Iterators.** You may implement `prev` for your `BlockIterator` so that you will be able to iterate the key-value pairs reversely. You may also have a variant of backward merge iterator and backward SST iterator (in the next chapter) so that your storage engine can do a reverse scan.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Sorted String Table (SST)



In this chapter, you will:

- Implement SST encoding and metadata encoding.
- Implement SST decoding and iterator.

To copy the test cases into the starter code and run them,


```
cargo x copy-test --week 1 --day 4
cargo x scheck
```

Task 1: SST Builder

In this task, you will need to modify:

```
src/table/builder.rs
src/table.rs
```

SSTs are composed of data blocks and index blocks stored on the disk. Usually, data blocks are lazily loaded -- they will not be loaded into the memory until a user requests it. Index blocks can also be loaded on-demand, but in this course, we make simple assumptions that all SST index blocks (meta blocks) can fit in memory (actually we do not have a designated index block implementation.) Generally, an SST file is of 256MB size.

The SST builder is similar to block builder -- users will call `add` on the builder. You should maintain a `BlockBuilder` inside SST builder and split blocks when necessary. Also, you will need to maintain block metadata `BlockMeta`, which includes the first/last keys in each block and the offsets of each block. The `build` function will encode the SST, write everything to disk using `FileObject::create`, and return an `SsTable` object.

The encoding of SST is like:

Block Section	Meta Section	Extra

data block ... data block	metadata	meta block offset
(u32)		

You also need to implement `estimated_size` function of `SsTableBuilder`, so that the caller can know when can it start a new SST to write data. The function doesn't need to be very accurate. Given the assumption that data blocks contain much more data than meta block, we can simply return the size of data blocks for `estimated_size`.

Besides SST builder, you will also need to complete the encoding/decoding of block metadata, so that `SsTableBuilder::build` can produce a valid SST file.

Task 2: SST Iterator

In this task, you will need to modify:

```
src/table/iterator.rs
src/table.rs
```

Like `BlockIterator`, you will need to implement an iterator over an SST. Note that you should load data on demand. For example, if your iterator is at block 1, it should not hold any other block content in memory until it reaches the next block.

`SsTableIterator` should implement the `StorageIterator` trait, so that it can be composed with other iterators in the future.

One thing to note is `seek_to_key` function. Basically, you will need to do binary search on block metadata to find which block might possibly contain the key. It is possible that the key does not exist in the LSM tree so that the block iterator will be invalid immediately after a seek. For example,

```
-----  
| block 1 | block 2 | block meta |  
-----  
| a, b, c | e, f, g | 1: a/c, 2: e/g |  
-----
```

We recommend only using the first key of each block to do the binary search so as to reduce the complexity of your implementation. If we do `seek(b)` in this SST, it is quite simple -- using binary search, we can know block 1 contains keys $a \leq \text{keys} < e$. Therefore, we load block 1 and seek the block iterator to the corresponding position.

But if we do `seek(d)`, we will position to block 1, if we only use first key as the binary search criteria, but seeking `d` in block 1 will reach the end of the block. Therefore, we should check if the iterator is invalid after the seek, and switch to the next block if necessary. Or you can leverage the last key metadata to directly position to a correct block, it is up to you.

Task 3: Block Cache

In this task, you will need to modify:

```
src/table/iterator.rs  
src/table.rs
```

You can implement a new `read_block_cached` function on `SsTable` .

We use `moka-rs` as our block cache implementation. Blocks are cached by `(sst_id, block_id)` as the cache key. You may use `try_get_with` to get the block from cache if it hits the cache / populate the cache if it misses the cache. If there are multiple requests reading the same block and cache misses, `try_get_with` will only issue a single read request to the disk and broadcast the result to all requests.

At this point, you may change your table iterator to use `read_block_cached` instead of `read_block` to leverage the block cache.

Test Your Understanding

- What is the time complexity of seeking a key in the SST?
- Where does the cursor stop when you seek a non-existent key in your implementation?
- Is it possible (or necessary) to do in-place updates of SST files?
- An SST is usually large (i.e., 256MB). In this case, the cost of copying/expanding the `Vec` would be significant. Does your implementation allocate enough space for your SST builder in advance? How did you implement it?
- Looking at the `moka` block cache, why does it return `Arc<Error>` instead of the original `Error`?
- Does the usage of a block cache guarantee that there will be at most a fixed number of blocks in memory? For example, if you have a `moka` block cache of 4GB and block size of 4KB, will there be more than 4GB/4KB number of blocks in memory at the same time?
- Is it possible to store columnar data (i.e., a table of 100 integer columns) in an LSM engine? Is the current SST format still a good choice?
- Consider the case that the LSM engine is built on object store services (i.e., S3). How would you optimize/change the SST format/parameters and the block cache to make it suitable for such services?

- For now, we load the index of all SSTs into the memory. Assume you have a 16GB memory reserved for the indexes, can you estimate the maximum size of the database your LSM system can support? (That's why you need an index cache!)

We do not provide reference answers to the questions, and feel free to discuss about them in the Discord community.

Bonus Tasks

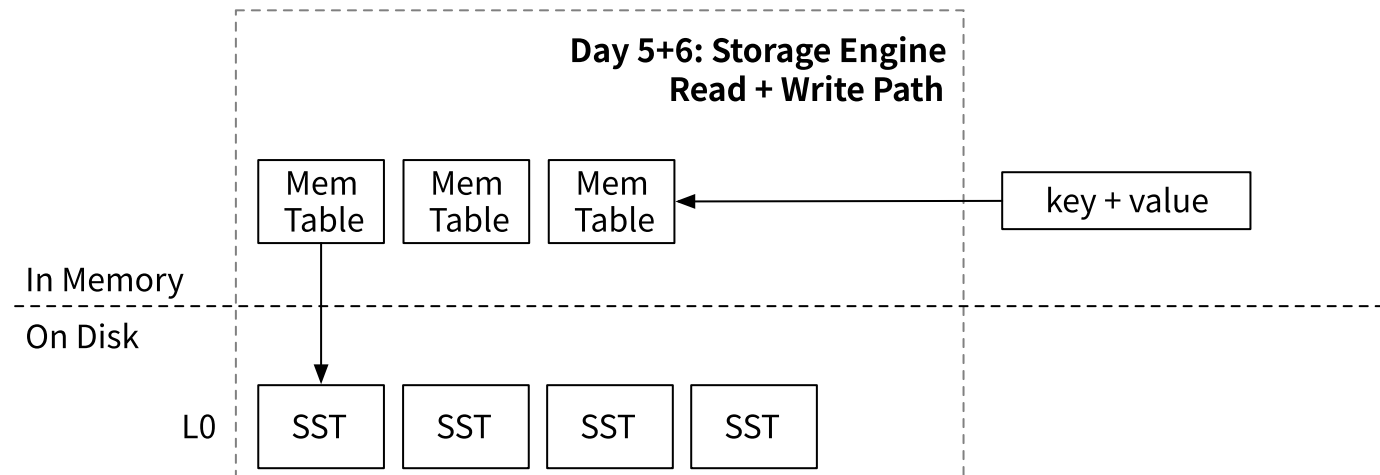
- **Explore different SST encoding and layout.** For example, in the [Lethe: Enabling Efficient Deletes in LSMs](#) paper, where the author adds secondary key support to SST.
 - Or you can use B+ Tree as the SST format instead of sorted blocks.
- **Index Blocks.** Split block indexes and block metadata into index blocks, and load them on-demand.
- **Index Cache.** Use a separate cache for indexes apart from the data block cache.
- **I/O Optimizations.** Align blocks to 4KB boundary and use direct I/O to bypass the system page cache.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Read Path



In this chapter, you will:

- Integrate SST into the LSM read path.
- Implement LSM read path `get` with SSTs.
- Implement LSM read path `scan` with SSTs.

To copy the test cases into the starter code and run them,

```
cargo x copy-test --week 1 --day 5
cargo x scheck
```

Task 1: Two Merge Iterator

In this task, you will need to modify:

```
src/iterators/two_merge_iterator.rs
```

You have already implemented a merge iterator that merges iterators of the same type (i.e., memtable iterators). Now that we have implemented the SST formats, we have both on-disk SST structures and in-memory memtables. When we scan from the storage engine, we will need to merge data from both memtable iterators and SST iterators into a single one. In this case, we need a `TwoMergeIterator<X, Y>` that merges two different types of iterators.

You can implement `TwoMergeIterator` in `two_merge_iterator.rs`. As we only have two iterators here, we do not need to maintain a binary heap. Instead, we can simply use a flag to indicate which iterator to read. Similar to `MergeIterator`, if the same key is found in both of the iterator, the first iterator takes the precedence.

Task 2: Read Path - Scan

In this task, you will need to modify:

```
src/lsm_iterator.rs  
src/lsm_storage.rs
```

After implementing `TwoMergeIterator`, we can change the `LsmIteratorInner` to have the following type:

```
type LsmIteratorInner =  
    TwoMergeIterator<MergeIterator<MemTableIterator>, MergeIterator<SsTableIterator>>;
```

So that our internal iterator of the LSM storage engine will be an iterator combining both data from the memtables and the SSTs.

Currently, our SST iterator doesn't support an end bound for scans. To address this, you'll need to implement this boundary check within the `LsmIterator` itself. This involves updating the `LsmIterator::new` constructor to accept an `end_bound` parameter:

```
pub(crate) fn new(iter: LsmIteratorInner, end_bound: Bound<Bytes>) -> Result<Self> {}
```

You will then need to modify the `LsmIterator`'s iteration logic to ensure it stops when the keys from the inner iterator reach or exceed this specified `end_bound`.

Our test cases will generate some memtables and SSTs in `l0_sstables`, and you will need to scan all of these data out correctly in this task. You do not need to flush SSTs until next chapter. Therefore, you can go ahead and modify your `LsmStorageInner::scan` interface to create a merge iterator over all memtables and SSTs, so as to finish the read path of your storage engine.

Because `SsTableIterator::create` involves I/O operations and might be slow, we do not want to do this in the `state` critical section. Therefore, you should firstly take read the `state` and clone the `Arc` of the LSM state snapshot. Then, you should drop the lock. After that, you can go through all L0 SSTs and create iterators for each of them, then create a merge iterator to retrieve the data.

```
fn scan(&self) {  
    let snapshot = {  
        let guard = self.state.read();  
        Arc::clone(&guard)  
    };  
    // create iterators and seek them  
}
```

In the LSM storage state, we only store the SST ids in the `l0_sstables` vector. You will need to retrieve the actual SST object from the `sstables` hash map.

Task 3: Read Path - Get

In this task, you will need to modify:

```
src/lsm_storage.rs
```

For get requests, it will be processed as lookups in the memtables, and then scans on the SSTs. You can create a merge iterator over all SSTs after probing all memtables. You can seek to the key that the user wants to lookup. There are two possibilities of the seek: the key is the same as what the user probes, and the key is not the same / does not exist. You should only return the value to the user when the key exists and is the same as probed. You should also reduce the critical section of the state lock as in the previous section. Also remember to handle deleted keys.

Test Your Understanding

- Consider the case that a user has an iterator that iterates the whole storage engine, and the storage engine is 1TB large, so that it takes ~1 hour to scan all the data. What would be the problems if the user does so? (This is a good question and we will ask it several times at different points of the course...)
- Another popular interface provided by some LSM-tree storage engines is multi-get (or vectored get). The user can pass a list of keys that they want to retrieve. The interface returns the value of each of the key. For example, `multi_get(vec!["a", "b", "c", "d"]) -> a=1,b=2,c=3,d=4`. Obviously, an easy implementation is to simply doing a single get for each of the key. How will you implement the multi-get interface, and what optimizations you can do to make it more efficient? (Hint: some operations during the get process will only need to be done once for all keys, and besides that, you can think of an improved disk I/O interface to better support this multi-get interface).

We do not provide reference answers to the questions, and feel free to discuss about them in the Discord community.

Bonus Tasks

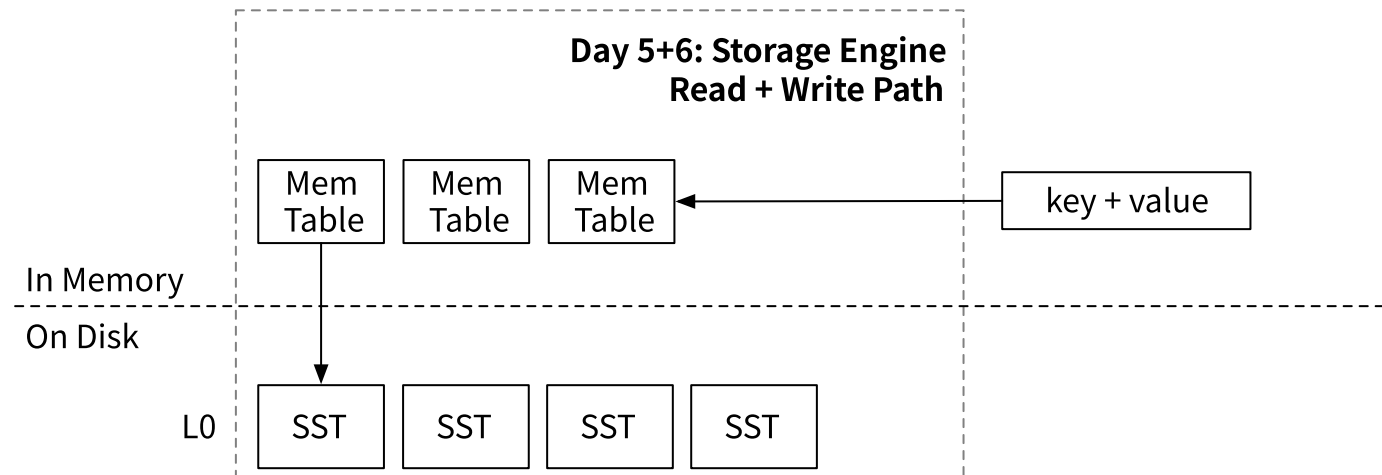
- **The Cost of Dynamic Dispatch.** Implement a `Box<dyn StorageIterator>` version of merge iterators and benchmark to see the performance differences.
- **Parallel Seek.** Creating a merge iterator requires loading the first block of all underlying SSTs (when you create `SSTIterator`). You may parallelize the process of creating iterators.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Write Path



In this chapter, you will:

- Implement the LSM write path with L0 flush.
- Implement the logic to correctly update the LSM state.

To copy the test cases into the starter code and run them,

```
cargo x copy-test --week 1 --day 6
cargo x scheck
```

Task 1: Flush Memtable to SST

At this point, we have all in-memory things and on-disk files ready, and the storage engine is able to read and merge the data from all these structures. Now, we are going to implement the logic to move things from memory to the disk (so-called flush), and complete the Mini-LSM week 1 course.

In this task, you will need to modify:

```
src/lsm_storage.rs  
src/mem_table.rs
```

You will need to modify `LSMStorageInner::force_flush_next_imm_memtable` and `MemTable::flush`. In `LSMStorageInner::open`, you will need to create the LSM database directory if it does not exist. To flush a memtable to the disk, we will need to do three things:

- Select a memtable to flush.
- Create an SST file corresponding to a memtable.
- Remove the memtable from the immutable memtable list and add the SST file to L0 SSTs.

We have not explained what is L0 (level-0) SSTs for now. In general, they are the set of SSTs files directly created as a result of memtable flush. In week 1 of this course, we will only have L0 SSTs on the disk. We will dive into how to organize them efficiently using leveled or tiered structure on the disk in week 2.

Note that creating an SST file is a compute-heavy and a costly operation. Again, we do not want to hold the `state` read/write lock for a long time, as it might block other operations and create huge latency spikes in the LSM operations. Also, we use the `state_lock` mutex to serialize state modification operations in the LSM tree. In this task, you will need to think carefully how to use these locks to make the LSM state modification race-condition free while minimizing critical sections.

We do not have concurrent test cases and you will need to think carefully about your implementation. Also, remember that the last memtable in the immutable memtable list is the earliest one, and is the one that you should flush.

► Spoilers: Flush L0 Pseudo Code

Task 2: Flush Trigger

In this task, you will need to modify:

```
src/lsm_storage.rs  
src/compact.rs
```

When the number of memtables (immutable + mutable) in memory exceeds the `num_memtable_limit` in LSM storage options, you should flush the earliest memtable to the disk. This is done by a flush thread in the background. The flush thread will be started with the `MiniLSM` structure. We have already implemented necessary code to start the thread and properly stop the thread.

In this task, you will need to implement `LsmStorageInner::trigger_flush` in `compact.rs`, and `MiniLsm::close` in `lsm_storage.rs`. `trigger_flush` will be executed every 50 milliseconds. If the number of memtables exceed the limit, you should call `force_flush_next_imm_memtable` to flush a memtable. When the user calls the `close` function, you should wait until the flush thread (and the compaction thread in week 2) to finish.

Task 3: Filter the SSTs

Now that you have a fully working storage engine, and you can use the `mini-lsm-cli` to interact with your storage engine.

```
cargo run --bin mini-lsm-cli -- --compaction none
```

And then,

```
fill 1000 3000
get 2333
flush
fill 1000 3000
get 2333
flush
get 2333
scan 2000 2333
```

If you fill more data, you can see your flush thread working and automatically flushing the L0 SSTs without using the `flush` command.

And lastly, let us implement a simple optimization on filtering the SSTs before we end this week. Based on the key range that the user provides, we can easily filter out some SSTs that do not contain the key range, so that we do not need to read them in the merge iterator.

In this task, you will need to modify:

```
src/lsm_storage.rs
src/iterators/*
src/lsm_iterator.rs
```

You will need to change your read path functions to skip the SSTs that is impossible to contain the key/key range. You will need to implement `num_active_iterators` for your iterators so that the test cases can do the check on whether your implementation is correct or not. For `MergeIterator` and `TwoMergeIterator`, it is the sum of `num_active_iterators` of all children iterators. Note that if you did not modify the fields in the starter code of `MergeIterator`, remember to also take `MergeIterator::current` into account. For `LsmIterator` and `FusedIterator`, simply return the number of active iterators from the inner iterator.

You can implement helper functions like `range_overlap` and `key_within` to simplify your code.

Test Your Understanding

- What happens if a user requests to delete a key twice?
- How much memory (or number of blocks) will be loaded into memory at the same time when the iterator is initialized?
- Some crazy users want to *fork* their LSM tree. They want to start the engine to ingest some data, and then fork it, so that they get two identical dataset and then operate on them separately. An easy but not efficient way to implement is to simply copy all SSTs and the in-memory structures to a new directory and start the engine. However, note that we never modify the on-disk files, and we can actually reuse the SST files from the parent engine. How do you think you can implement this fork functionality efficiently without copying data? (Check out [Neon Branching](#)).
- Imagine you are building a multi-tenant LSM system where you host 10k databases on a single 128GB memory machine. The memtable size limit is set to 256MB. How much memory for memtable do you need for this setup?
 - Obviously, you don't have enough memory for all these memtables. Assume each user still has their own memtable, how can you design the memtable flush policy to make it work? Does it make sense to make all these users share the same memtable (i.e., by encoding a tenant ID as the key prefix)?

We do not provide reference answers to the questions, and feel free to discuss about them in the Discord community.

Bonus Tasks

- **Implement Write/L0 Stall.** When the number of memtables exceed the maximum number too much, you can stop users from writing to the storage engine. You may also implement write stall for L0 tables in week 2 after you have implemented compactions.

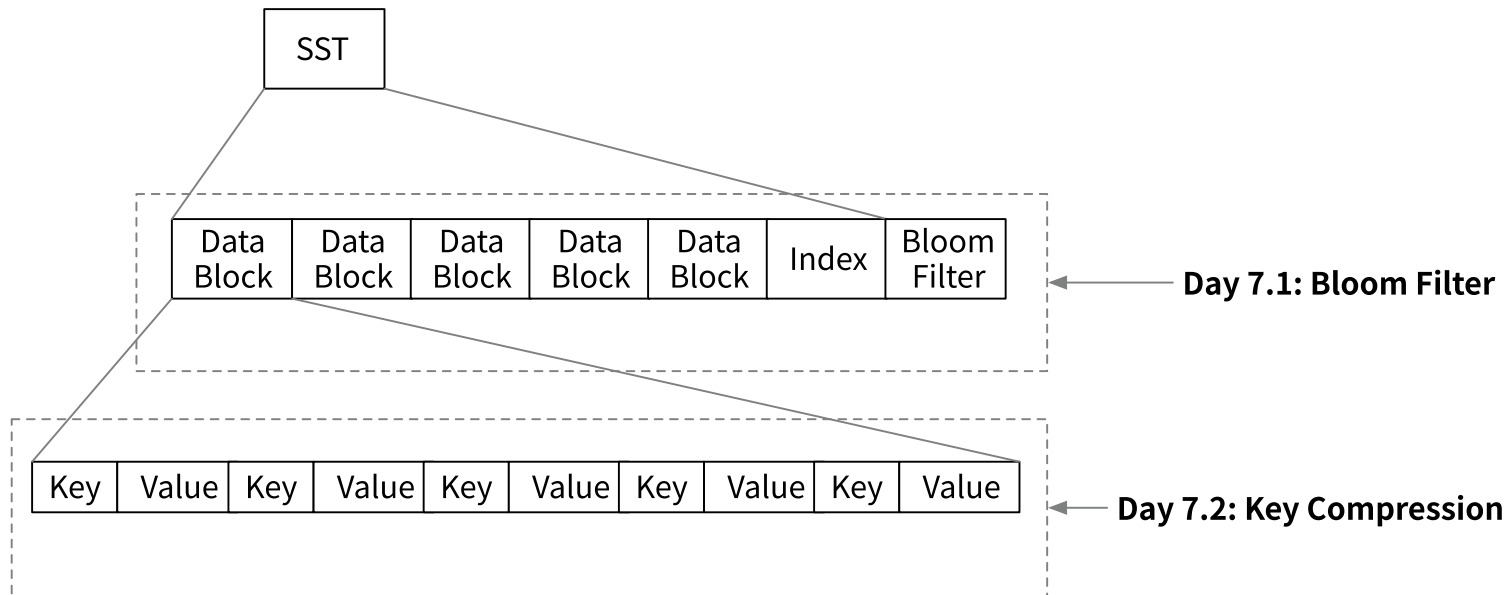
- **Prefix Scan.** You may filter more SSTs by implementing the prefix scan interface and using the prefix information.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Snack Time: SST Optimizations



In the previous chapter, you already built a storage engine with get/scan/put support. At the end of this week, we will implement some easy but important optimizations of SST formats. Welcome to Mini-LSM's week 1 snack time!

In this chapter, you will:

- Implement bloom filter on SSTs and integrate into the LSM read path `get`.
- Implement key compression in SST block format.

To copy the test cases into the starter code and run them,

```
cargo x copy-test --week 1 --day 7
cargo x scheck
```

Task 1: Bloom Filters

Bloom filters are probabilistic data structures that maintains a set of keys. You can add keys to a bloom filter, and you can know what key may exist / must not exist in the set of keys being added to the bloom filter.

You usually need to have a hash function in order to construct a bloom filter, and a key can have multiple hashes. Let us take a look at the below example. Assume that we already have hashes of some keys and the bloom filter has 7 bits.

[Note: If you want to understand bloom filters better, look [here](#)]

```
hash1 = ((character - a) * 13) % 7
hash2 = ((character - a) * 11) % 7
b -> 6 4
c -> 5 1
d -> 4 5
e -> 3 2
g -> 1 3
h -> 0 0
```

If we insert b, c, d into the 7-bit bloom filter, we will get:

```
bit 0123456
insert b    1 1
insert c    1 1
insert d    11
result 0101111
```

When probing the bloom filter, we generate the hashes for a key, and see if the corresponding bit has been set. If all of them are set to true, then the key may exist in the bloom filter. Otherwise, the key must NOT exist in the bloom filter.

For `e -> 3 2`, as the bit 2 is not set, it should not be in the original set. For `g -> 1 3`, because two bits are all set, it may or may not exist in the set. For `h -> 0 0`, both of the bits (actually it's one bit) are not set, and therefore it should not be in the original set.

```
b -> maybe (actual: yes)
c -> maybe (actual: yes)
d -> maybe (actual: yes)
e -> MUST not (actual: no)
g -> maybe (actual: no)
h -> MUST not (actual: no)
```

Remember that at the end of last chapter, we implemented SST filtering based on key range. Now, on the `get` read path, we can also use the bloom filter to ignore SSTs that do not contain the key that the user wants to lookup, therefore reducing the number of files to be read from the disk.

In this task, you will need to modify:

```
src/table/bloom.rs
```

In the implementation, you will build a bloom filter from key hashes (which are u32 numbers). For each of the hash, you will need to set `k` bits. The bits are computed by:

```
let delta = (h >> 17) | (h << 15); // h is the key hash
for _ in 0..k {
    // TODO: use the hash to set the corresponding bit
    h = h.wrapping_add(delta);
}
```

We provide all the skeleton code for doing the magic mathematics. You only need to implement the procedure of building a bloom filter and probing a bloom filter.

Task 2: Integrate Bloom Filter on the Read Path

In this task, you will need to modify:

```
src/table/builder.rs
src/table.rs
src/lsm_storage.rs
```

For the bloom filter encoding, you can append the bloom filter to the end of your SST file. You will need to store the bloom filter offset at the end of the file, and compute meta offsets accordingly.



We use the `farmhash` crate to compute the hashes of the keys. When building the SST, you will need also to build the bloom filter by computing the key hash using `farmhash::fingerprint32`. You will need to encode/decode the bloom filters with the block meta. You can choose false positive rate 0.01 for your bloom filter. You may need to add new fields to the structures apart from the ones provided in the starter code as necessary.

After that, you can modify the `get` read path to filter SSTs based on bloom filters.

We do not have integration test for this part and you will need to ensure that your implementation still pass all previous chapter tests.

Task 3: Key Prefix Encoding + Decoding

In this task, you will need to modify:

```
src/block/builder.rs  
src/block/iterator.rs
```

As the SST file stores keys in order, it is possible that the user stores keys of the same prefix, and we can compress the prefix in the SST encoding so as to save space.

We compare the current key with the first key in the block. We store the key as follows:

```
key_overlap_len (u16) | rest_key_len (u16) | key (rest_key_len)
```

The `key_overlap_len` indicates how many bytes are the same as the first key in the block. For example, if we see a record: `5|3|LSM`, where the first key in the block is `mini-something`, we can recover the current key to `mini-LSM`.

After you finish the encoding, you will also need to implement decoding in the block iterator. You may need to add new fields to the structures apart from the ones provided in the starter code as necessary.

Test Your Understanding

- How does the bloom filter help with the SST filtering process? What kind of information can it tell you about a key? (may not exist/may exist/must exist/must not exist)
- Consider the case that we need a backward iterator. Does our key compression affect backward iterators?
- Can you use bloom filters on scan?

- What might be the pros/cons of doing key-prefix encoding over adjacent keys instead of with the first key in the block?

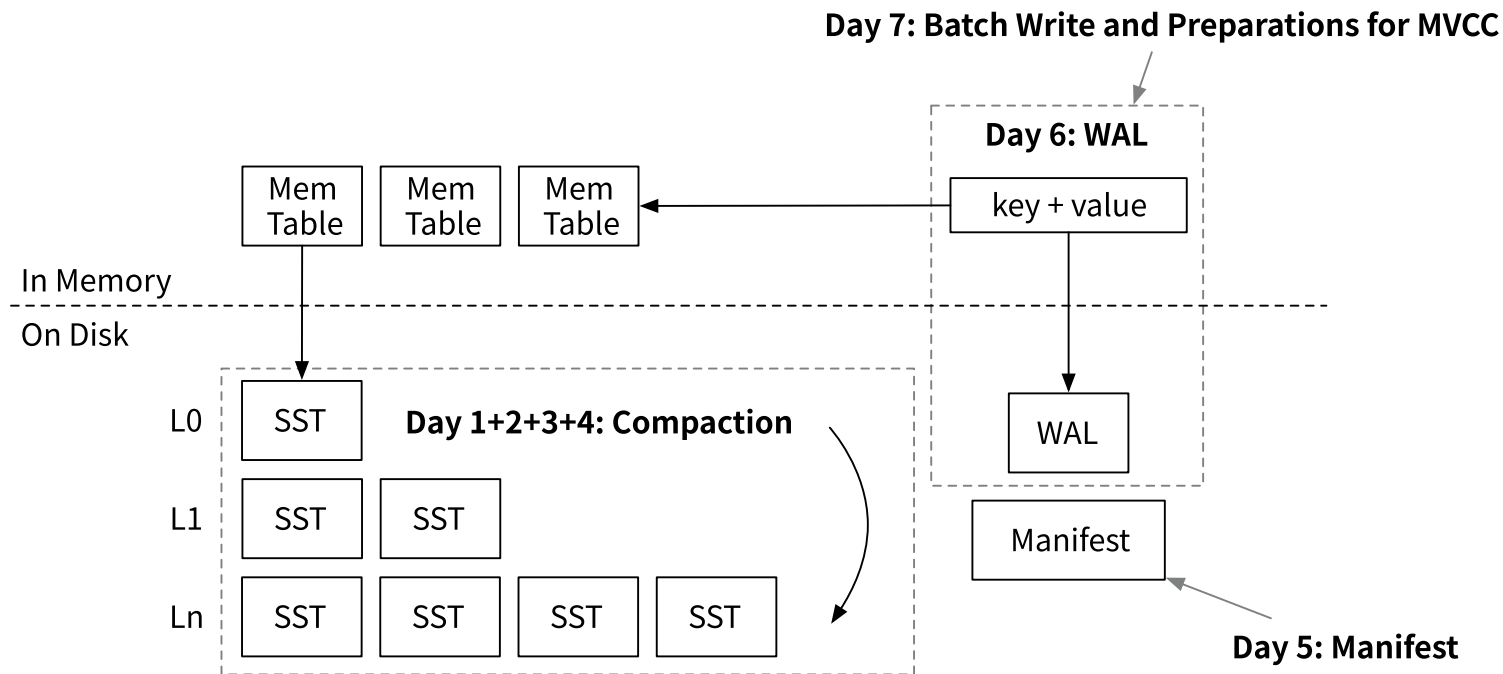
We do not provide reference answers to the questions, and feel free to discuss about them in the Discord community.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Week 2 Overview: Compaction and Persistence



In the last week, you have implemented all necessary structures for an LSM storage engine, and your storage engine already supports read and write interfaces. In this week, we will deep dive into the disk organization of the SST files and investigate an optimal way to achieve both performance and cost efficiency in the system. We will spend 4 days learning different compaction strategies, from the easiest to the most complex ones, and then implement the remaining parts for the storage engine persistence. At the end of this week, you will have a fully functional and efficient LSM storage engine.

We have 7 chapters (days) in this part:

- [Day 1: Compaction Implementation](#). You will merge all L0 SSTs into a sorted run.
- [Day 2: Simple Leveled Compaction](#). You will implement a classic leveled compaction algorithm and use compaction simulator to see how well it works.

- [Day 3: Tiered/Universal Compaction](#). You will implement the RocksDB universal compaction algorithm and understand the pros/cons.
- [Day 4: Leveled Compaction](#). You will implement the RocksDB leveled compaction algorithm. This compaction algorithm also supports partial compaction, so as to reduce peak space usage.
- [Day 5: Manifest](#). You will store the LSM state on the disk and recover from the state.
- [Day 6: Write-Ahead Log \(WAL\)](#). User requests will be routed to both memtable and WAL so that all operations will be persisted.
- [Day 7: Write Batch and Checksums](#). You will implement write batch API (for preparation for week 3 MVCC) and checksums for all of your storage formats.

Compaction and Read Amplification

Let us talk about compaction first. In the previous part, you simply flush the memtable to an L0 SST. Imagine that you have written gigabytes of data and now you have 100 SSTs. Every read request (without filtering) will need to read 100 blocks from these SSTs. This amplification is read amplification -- the number of I/O requests you will need to send to the disk for one get operation.

To reduce read amplification, we can merge all the L0 SSTs into a larger structure, so that it would be possible to only read one SST and one block to retrieve the requested data. Say that we still have these 100 SSTs, and now, we do a merge sort of these 100 SSTs to produce another 100 SSTs, each of them contains non-overlapping key ranges. This process is **compaction**, and these 100 non-overlapping SSTs is a **sorted run**.

To make this process clearer, let us take a look at this concrete example:

```
SST 1: key range 00000 - key 10000, 1000 keys
SST 2: key range 00005 - key 10005, 1000 keys
SST 3: key range 00010 - key 10010, 1000 keys
```


We have 3 SSTs in the LSM structure. If we need to access key 02333, we will need to probe all of these 3 SSTs. If we can do a compaction, we might get the following 3 new SSTs:

SST 4: key range 00000 – key 03000, 1000 keys

SST 5: key range 03001 – key 06000, 1000 keys

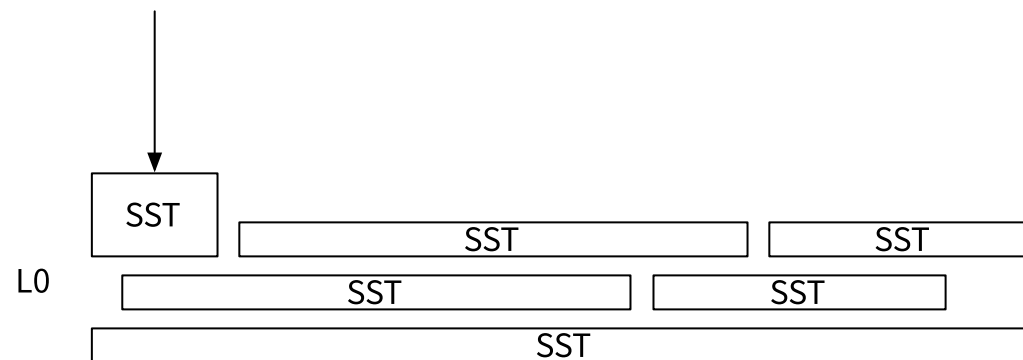
SST 6: key range 06000 – key 10010, 1000 keys

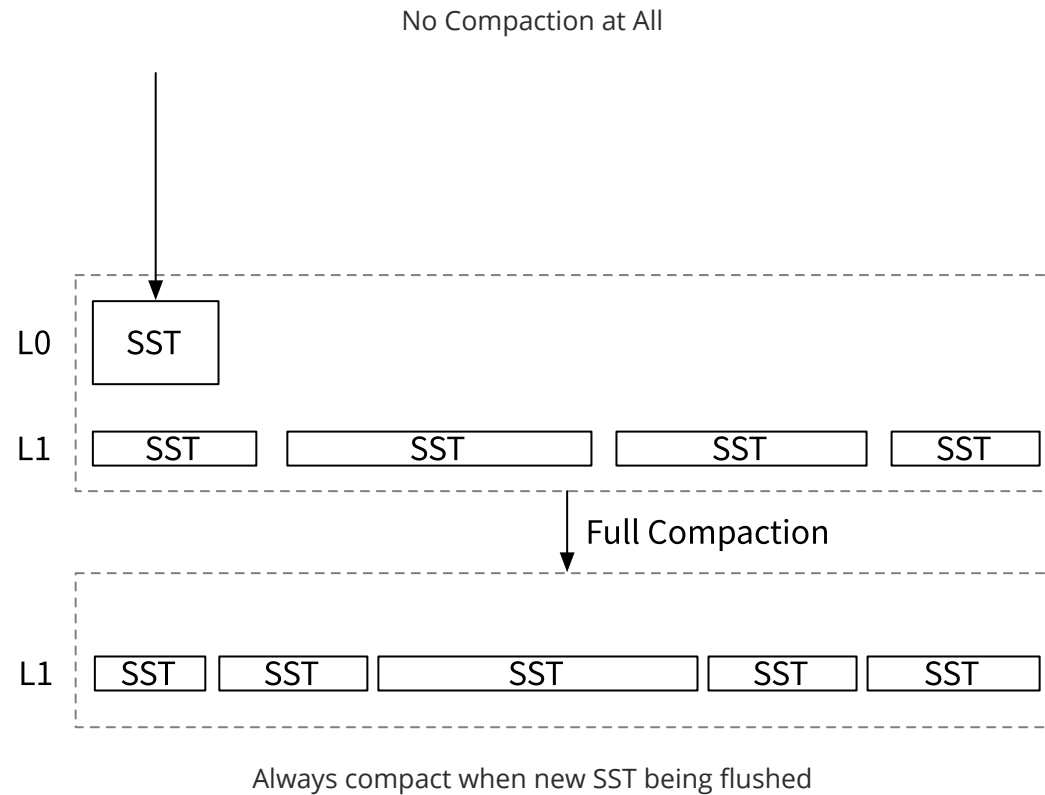
The 3 new SSTs are created by merging SST 1, 2, and 3. We can get a sorted 3000 keys and then split them into 3 files, so as to avoid having a super large SST file. Now our LSM state has 3 non-overlapping SSTs, and we only need to access SST 4 to find key 02333.

Two Extremes of Compaction and Write Amplification

So from the above example, we have 2 naive ways of handling the LSM structure -- not doing compactions at all, and always do full compaction when new SSTs are flushed.

Compaction is a time-consuming operation. It will need to read all data from some files, and write the same amount of files to the disk. This operation takes a lot of CPU resources and I/O resources. Not doing compactions at all leads to high read amplification, but it does not need to write new files. Always doing full compaction reduces the read amplification, but it will need to constantly rewrite the files on the disk.

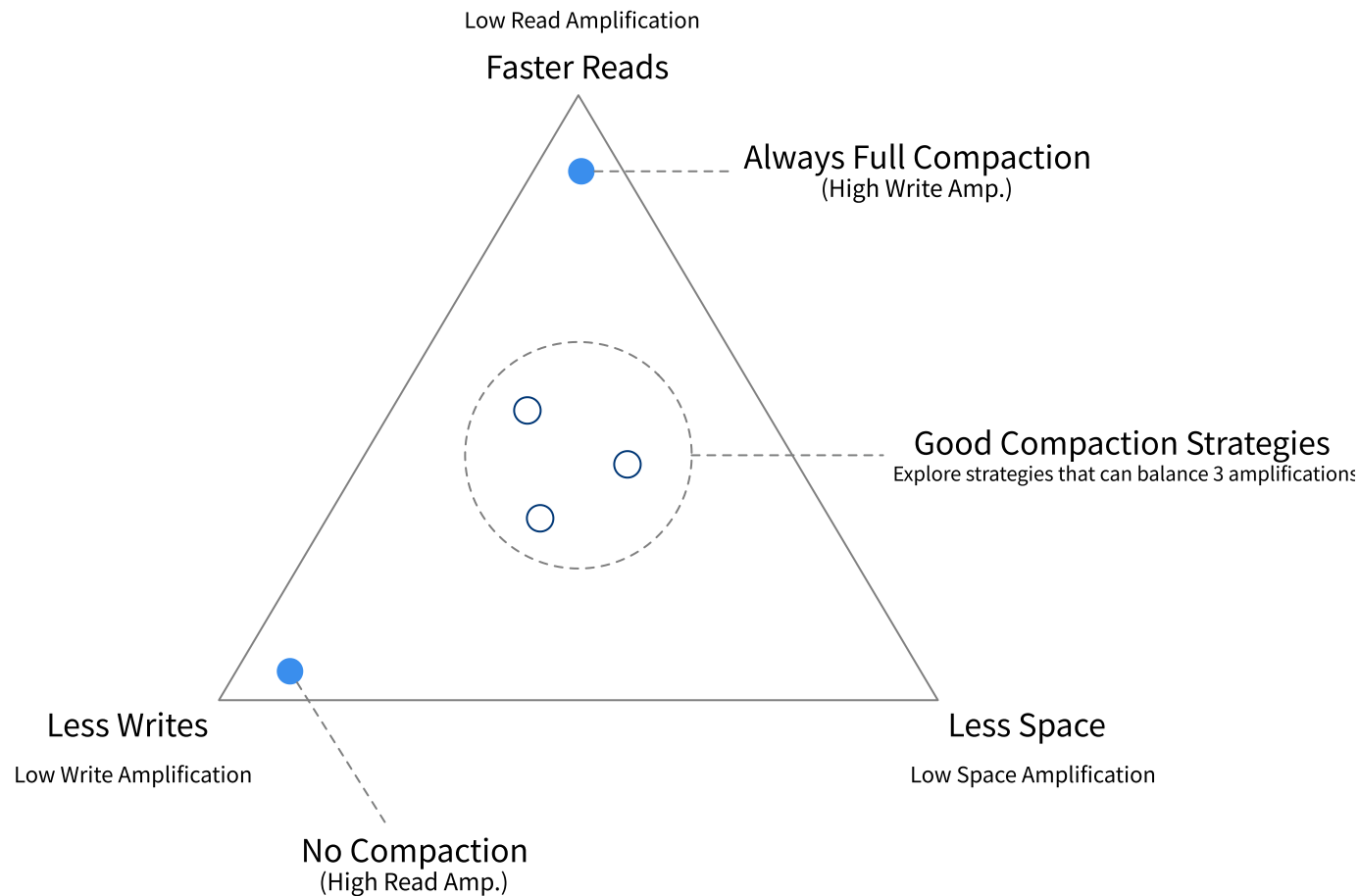




The ratio of memtables flushed to the disk versus total data written to the disk is write amplification. That is to say, no compaction has a write amplification ratio of 1x, because once the SSTs are flushed to the disk, they will stay there. Always doing compaction has a very high write amplification. If we do a full compaction every time we get an SST, the data written to the disk will be quadratic to the number of SSTs flushed. For example, if we flushed 100 SSTs to the disk, we will do compactations of 2 files, 3 files, ..., 100 files, where the actual total amount of data we wrote to the disk is about 5000 SSTs. The write amplification after writing 100 SSTs in this cause would be 50x.

A good compaction strategy can balance read amplification, write amplification, and space amplification (we will talk about it soon). In a general-purpose LSM storage engine, it is generally impossible to find a strategy that can achieve the lowest amplification in all 3 of these factors, unless there are some specific data pattern that the engine could use. The good thing about LSM is that we can theoretically analyze the amplifications of a compaction strategy and all these things happen in

the background. We can choose compaction strategies and dynamically change some parameters of them to adjust our storage engine to the optimal state. Compaction strategies are all about tradeoffs, and LSM-based storage engine enables us to select what to be traded at runtime.



One typical workload in the industry is like: the user first batch ingests data into the storage engine, usually gigabytes per second, when they start a product. Then, the system goes live and users start doing small transactions over the system. In the first phase, the engine should be able to quickly ingest data, and therefore we can use a compaction strategy that minimize write amplification to accelerate this process. Then, we adjust the parameters of the compaction algorithm to make it

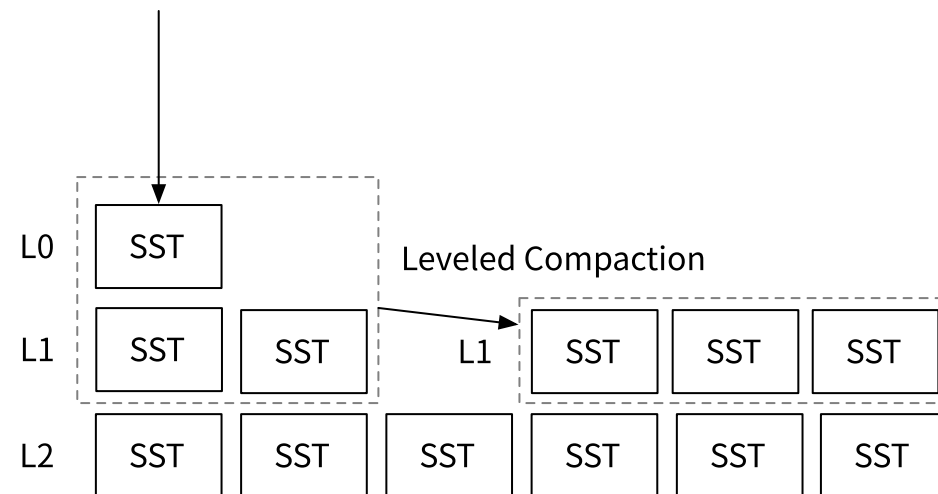
optimized for read amplification, and do a full compaction to reorder existing data, so that the system can run stably when it goes live.

If the workload is like a time-series database, it is possible that the user always populate and truncate data by time. Therefore, even if there is no compaction, these append-only data can still have low amplification on the disk. Therefore, in real life, you should watch for patterns or specific requirements from the users, and use these information to optimize your system.

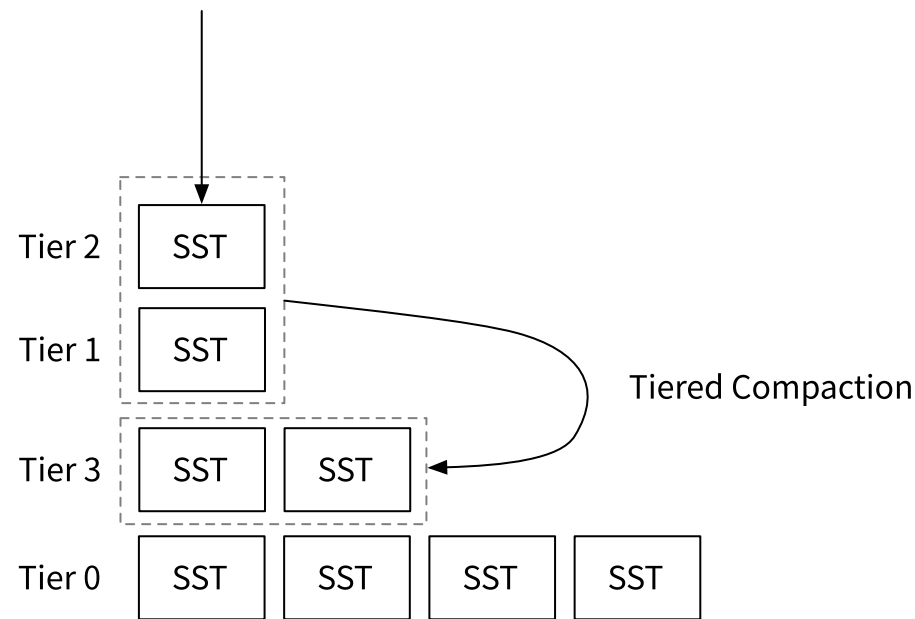
Compaction Strategies Overview

Compaction strategies usually aim to control the number of sorted runs, so as to keep read amplification in a reasonable amount of number. There are generally two categories of compaction strategies: leveled and tiered.

In leveled compaction, the user can specify a maximum number of levels, which is the number of sorted runs in the system (except L0). For example, RocksDB usually keeps 6 levels (sorted runs) in leveled compaction mode. During the compaction process, SSTs from two adjacent levels will be merged and then the produced SSTs will be put to the lower level of the two levels. Therefore, you will usually see a small sorted run merged with a large sorted run in leveled compaction. The sorted runs (levels) grow exponentially in size -- the lower level will be `<some number>` of the upper level in size.



In tiered compaction, the engine will dynamically adjust the number of sorted runs by merging them or letting new SSTs flushed as new sorted run (a tier) to minimize write amplification. In this strategy, you will usually see the engine merge two equally-sized sorted runs. The number of tiers can be high if the compaction strategy does not choose to merge tiers, therefore making read amplification high. In this course, we will implement RocksDB's universal compaction, which is a kind of tiered compaction strategy.



Space Amplification

The most intuitive way to compute space amplification is to divide the actual space used by the LSM engine by the user space usage (i.e., database size, number of rows in the database, etc.). The engine will need to store delete tombstones, and sometimes multiple version of the same key if compaction is not happening frequently enough, therefore causing space amplification.

On the engine side, it is usually hard to know the exact amount of data the user is storing, unless we scan the whole database and see how many dead versions are there in the engine. Therefore, one way of estimating the space amplification is to divide the full storage file size by the last level size. The assumption behind this estimation method is that the insertion and deletion rate of a workload should be the same after the user fills the initial data. We assume the user-side data size does not change, and therefore the last level contains the snapshot of the user data at some point, and the upper levels contain new changes. When compaction merges everything to the last level, we can get a space amplification factor of 1x using this estimation method.

Note that compaction also takes space -- you cannot remove files being compacted before the compaction is complete. If you do a full compaction of the database, you will need free storage space as much as the current engine file size.

In this part, we will have a compaction simulator to help you visualize the compaction process and the decision of your compaction algorithm. We provide minimal test cases to check the properties of your compaction algorithm, and you should watch closely on the statistics and the output of the compaction simulator to know how well your compaction algorithm works.

Persistence

After implementing the compaction algorithms, we will implement two key components in the system: manifest, which is a file that stores the LSM state, and WAL, which persists memtable data to the disk before it is flushed as an SST. After finishing these two components, the storage engine will have full persistence support and can be used in your products.

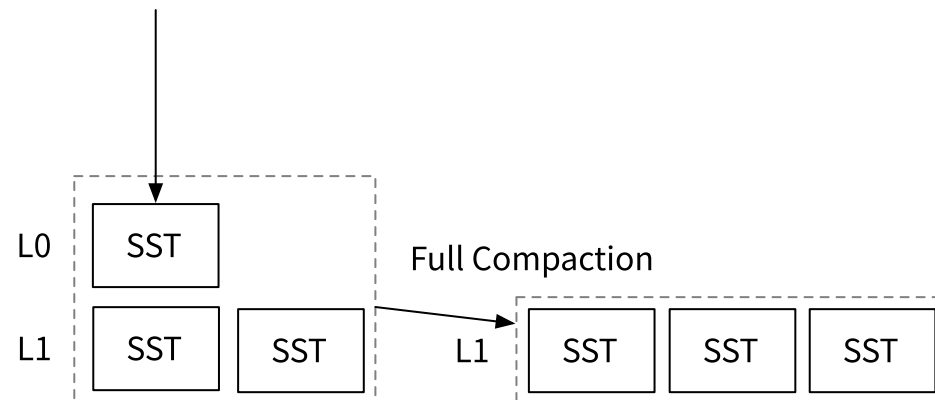
If you do not want to dive too deep into compactions, you can also finish chapter 2.1 and 2.2 to implement a very simple leveled compaction algorithm, and directly go for the persistence part. Implementing full leveled compaction and universal compaction are not required to build a working storage engine in week 2.

Snack Time

After implementing compaction and persistence, we will have a short chapter on implementing the batch write interface and checksums.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).
Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.
mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Compaction Implementation




In this chapter, you will:

- Implement the compaction logic that combines some files and produces new files.
- Implement the logic to update the LSM states and manage SST files on the filesystem.
- Update LSM read path to incorporate the LSM levels.

To copy the test cases into the starter code and run them,

```
cargo x copy-test --week 2 --day 1  
cargo x scheck
```

 It might be helpful to take a look at [week 2 overview](#) before reading this chapter to have a general overview of compactions.

Task 1: Compaction Implementation

In this task, you will implement the core logic of doing a compaction -- merge sort a set of SST files into a sorted run. You will need to modify:

`src/compact.rs`

Specifically, the `force_full_compaction` and `compact` function. `force_full_compaction` is the compaction trigger that decides which files to compact and update the LSM state. `compact` does the actual compaction job that merges some SST files and return a set of new SST files.

Your compaction implementation should take all SSTs in the storage engine, do a merge over them by using `MergeIterator`, and then use the SST builder to write the result into new files. You will need to split the SST files if the file is too large. After compaction completes, you can update the LSM state to add all the new sorted run to the first level of the LSM tree. And, you will need to remove unused files in the LSM tree. In your implementation, your SSTs should only be stored in two places: the L0 SSTs and the L1 SSTs. That is to say, the `levels` structure in the LSM state should only have one vector. In `LsmStorageState`, we have already initialized the LSM to have L1 in `levels` field.

Compaction should not block L0 flush, and therefore you should not take the state lock when merging the files. You should only take the state lock at the end of the compaction process when you update the LSM state, and release the lock right after finishing modifying the states.

You can assume that the user will ensure there is only one compaction going on.

`force_full_compaction` will be called in only one thread at any time. The SSTs being put in the level 1 should be sorted by their first key and should not have overlapping key ranges.

► Spoilers: Compaction Pseudo Code

In your compaction implementation, you only need to handle `FullCompaction` for now, where the task information contains the SSTs that you will need to compact. You will also need to ensure the

order of the SSTs are correct so that the latest version of a key will be put into the new SST.

Because we always compact all SSTs, if we find multiple version of a key, we can simply retain the latest one. If the latest version is a delete marker, we do not need to keep it in the produced SST files. This does not apply for the compaction strategies in the next few chapters.

There are some things that you might need to think about.

- How does your implementation handle L0 flush in parallel with compaction? (Not taking the state lock when doing the compaction, and also need to consider new L0 files produced when compaction is going on.)
- If your implementation removes the original SST files immediately after the compaction completes, will it cause problems in your system? (Generally no on macOS/Linux because the OS will not actually remove the file until no file handle is being held.)

Task 2: Concat Iterator

In this task, you will need to modify,

```
src/iterators/concat_iterator.rs
```

Now that you have created sorted runs in your system, it is possible to do a simple optimization over the read path. You do not always need to create merge iterators for your SSTs. If SSTs belong to one sorted run, you can create a concat iterator that simply iterates the keys in each SST in order, because SSTs in one sorted run do not contain overlapping key ranges and they are sorted by their first key. We do not want to create all SST iterators in advance (because it will lead to one block read), and therefore we only store SST objects in this iterator.

Task 3: Integrate with the Read Path

In this task, you will need to modify,

```
src/lsm_iterator.rs  
src/lsm_storage.rs  
src/compact.rs
```

Now that we have the two-level structure for your LSM tree, and you can change your read path to use the new concat iterator to optimize the read path.

You will need to change the inner iterator type of the `LsmStorageIterator`. After that, you can construct a two merge iterator that merges memtables and L0 SSTs, and another merge iterator that merges that iterator with the L1 concat iterator.

You can also change your compaction implementation to leverage the concat iterator.

You will need to implement `num_active_iterators` for concat iterator so that the test case can test if concat iterators are being used by your implementation, and it should always be 1.

To test your implementation interactively,

```
cargo run --bin mini-lsm-cli-ref -- --compaction none # reference solution  
cargo run --bin mini-lsm-cli -- --compaction none # your solution
```

And then,

```
fill 1000 3000
flush
fill 1000 3000
flush
full_compaction
fill 1000 3000
flush
full_compaction
get 2333
scan 2000 2333
```

Test Your Understanding

- What are the definitions of read/write/space amplifications? (This is covered in the overview chapter)
- What are the ways to accurately compute the read/write/space amplifications, and what are the ways to estimate them?
- Is it correct that a key will take some storage space even if a user requests to delete it?
- Given that compaction takes a lot of write bandwidth and read bandwidth and may interfere with foreground operations, it is a good idea to postpone compaction when there are large write flow. It is even beneficial to stop/pause existing compaction tasks in this situation. What do you think of this idea? (Read the [SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores](#) paper!)
- Is it a good idea to use/fill the block cache for compactions? Or is it better to fully bypass the block cache when compaction?
- Does it make sense to have a `struct ConcatIterator<I: StorageIterator>` in the system?
- Some researchers/engineers propose to offload compaction to a remote server or a serverless lambda function. What are the benefits, and what might be the potential challenges and performance impacts of doing remote compaction? (Think of the point when a compaction completes and what happens to the block cache on the next read request...)

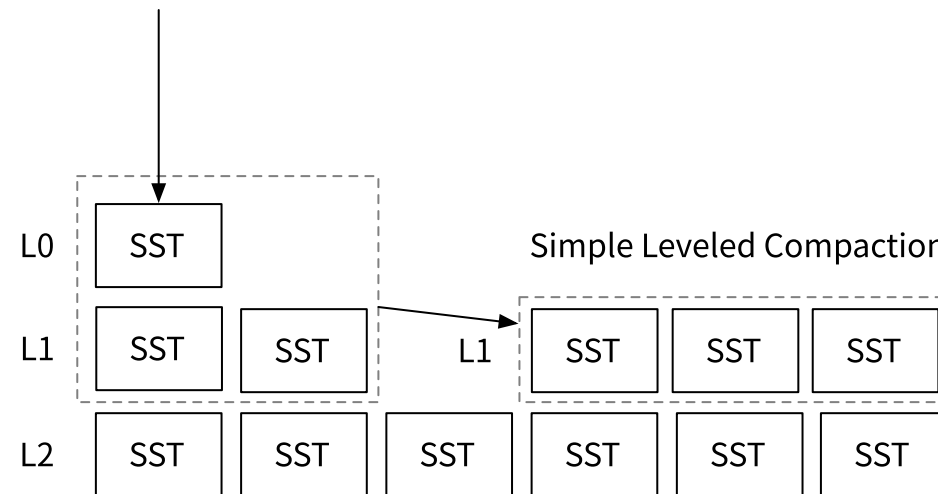
We do not provide reference answers to the questions, and feel free to discuss about them in the Discord community.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Simple Compaction Strategy




In this chapter, you will:

- Implement a simple leveled compaction strategy and simulate it on the compaction simulator.
- Start compaction as a background task and implement a compaction trigger in the system.

To copy the test cases into the starter code and run them,

```
cargo x copy-test --week 2 --day 2
cargo x scheck
```

 It might be helpful to take a look at [week 2 overview](#) before reading this chapter to have a general overview of compactions.

Task 1: Simple Levelled Compaction

In this chapter, we are going to implement our first compaction strategy -- simple leveled compaction. In this task, you will need to modify:

`src/compact/simple_levelled.rs`

Simple leveled compaction is similar the original LSM paper's compaction strategy. It maintains a number of levels for the LSM tree. When a level ($\geq L1$) is too large, it will merge all of this level's SSTs with next level. The compaction strategy is controlled by 3 parameters as defined in `SimpleLevelledCompactionOptions`.

- `size_ratio_percent`: lower level number of files / upper level number of files. In reality, we should compute the actual size of the files. However, we simplified the equation to use number of files to make it easier to do the simulation. When the ratio is too low (upper level has too many files), we should trigger a compaction.
- `level0_file_num_compaction_trigger`: when the number of SSTs in L0 is larger than or equal to this number, trigger a compaction of L0 and L1.
- `max_levels`: the number of levels (excluding L0) in the LSM tree.

Assume `size_ratio_percent=200` (Lower level should have 2x number of files as the upper level), `max_levels=3`, `level0_file_num_compaction_trigger=2`, let us take a look at the below example.

Assume the engine flushes two L0 SSTs. This reaches the `level0_file_num_compaction_trigger`, and your controller should trigger an L0->L1 compaction.


```
--- After Flush ---
L0 (2): [1, 2]
L1 (0): []
L2 (0): []
L3 (0): []
--- After Compaction ---
L0 (0): []
L1 (2): [3, 4]
L2 (0): []
L3 (0): []
```

Now, L2 is empty while L1 has two files. The size ratio percent for L1 and L2 is $(L2/L1) * 100 = (0/2) * 100 = 0 < \text{size_ratio_percent} (200)$. Therefore, we will trigger a L1+L2 compaction to push the data lower to L2. The same applies to L2 and these two SSTs will be placed at the bottom-most level after 2 compactions.

```
--- After Compaction ---
L0 (0): []
L1 (0): []
L2 (2): [5, 6]
L3 (0): []
--- After Compaction ---
L0 (0): []
L1 (0): []
L2 (0): []
L3 (2): [7, 8]
```

Continue flushing SSTs, we will find:

```
L0 (0): []
L1 (0): []
L2 (2): [13, 14]
L3 (2): [7, 8]
```

At this point, $L3/L2 = (1 / 1) * 100 = 100 < \text{size_ratio_percent} (200)$. Therefore, we need to trigger a compaction between L2 and L3.

```
--- After Compaction ---  
L0 (0): []  
L1 (0): []  
L2 (0): []  
L3 (4): [15, 16, 17, 18]
```

As we flush more SSTs, we will possibly end up at a state as follows:

```
--- After Flush ---  
L0 (2): [19, 20]  
L1 (0): []  
L2 (0): []  
L3 (4): [15, 16, 17, 18]  
--- After Compaction ---  
L0 (0): []  
L1 (0): []  
L2 (2): [23, 24]  
L3 (4): [15, 16, 17, 18]
```

Because $L3/L2 = (4 / 2) * 100 = 200 \geq \text{size_ratio_percent} (200)$, we do not need to merge L2 and L3 and will end up with the above state. Simple leveled compaction strategy always compact a full level, and keep a fanout size between levels, so that the lower level is always some multiplier times larger than the upper level.

We have already initialized the LSM state to have `max_level` levels. You should first implement `generate_compaction_task` that generates a compaction task based on the above 3 criteria. After that, implement `apply_compaction_result` . We recommend you implement L0 trigger first, run a compaction simulation, and then implement the size ratio trigger, and then run a compaction simulation. To run the compaction simulation,

```
cargo run --bin compaction-simulator-ref simple # Reference solution
cargo run --bin compaction-simulator simple # Your solution
```

The simulator will flush an L0 SST into the LSM state, run your compaction controller to generate a compaction task, and then apply the compaction result. Each time a new SST gets flushed, it will repetitively call the controller until no compaction needs to be scheduled, and therefore you should ensure your compaction task generator will converge.

In your compaction implementation, you should reduce the number of active iterators (i.e., use concat iterator) as much as possible. Also, remember that merge order matters, and you will need to ensure that the iterators you create produces key-value pairs in the correct order, when multiple versions of a key appear.

Also, note that some parameters in the implementation is 0-based, and some of them are 1-based. Be careful when you use the `level` as an index in a vector.

Note: we do not provide fine-grained unit tests for this part. You can run the compaction simulator and compare with the output of the reference solution to see if your implementation is correct.

Task 2: Compaction Thread

In this task, you will need to modify:

```
src/compact.rs
```

Now that you have implemented your compaction strategy, you will need to run it in a background thread, so as to compact the files in the background. In `compact.rs`, `trigger_compaction` will be called every 50ms, and you will need to:

1. generate a compaction task, if no task needs to be scheduled, return ok.
2. run the compaction and get a list of new SSTs.
3. Similar to `force_full_compaction` you have implemented in the previous chapter, update the LSM state.

Task 3: Integrate with the Read Path

In this task, you will need to modify:

```
src/lsm_storage.rs
```

Now that you have multiple levels of SSTs, you can modify your read path to include the SSTs from the new levels. You will need to update the scan/get function to include all levels below L1. Also, you might need to change the `LsmStorageIterator` inner type again.

To test your implementation interactively,

```
cargo run --bin mini-lsm-cli-ref -- --compaction simple # reference solution
cargo run --bin mini-lsm-cli -- --compaction simple # your solution
```

And then,

```
fill 1000 3000
flush
fill 1000 3000
flush
fill 1000 3000
flush
get 2333
scan 2000 2333
```

You may print something, for example, the compaction task information, when the compactor triggers a compaction.

Test Your Understanding

- What is the estimated write amplification of leveled compaction?
- What is the estimated read amplification of leveled compaction?
- Is it correct that a key will only be purged from the LSM tree if the user requests to delete it and it has been compacted in the bottom-most level?
- Is it a good strategy to periodically do a full compaction on the LSM tree? Why or why not?
- Actively choosing some old files/levels to compact even if they do not violate the level amplifier would be a good choice, is it true? (Look at the [Lethe](#) paper!)
- If the storage device can achieve a sustainable 1GB/s write throughput and the write amplification of the LSM tree is 10x, how much throughput can the user get from the LSM key-value interfaces?
- Can you merge L1 and L3 directly if there are SST files in L2? Does it still produce correct result?
- So far, we have assumed that our SST files use a monotonically increasing id as the file name. Is it okay to use `<level>_<begin_key>_<end_key>.sst` as the SST file name? What might be the potential problems with that? (You can ask yourself the same question in week 3...)
- What is your favorite boba shop in your city? (If you answered yes in week 1 day 3...)

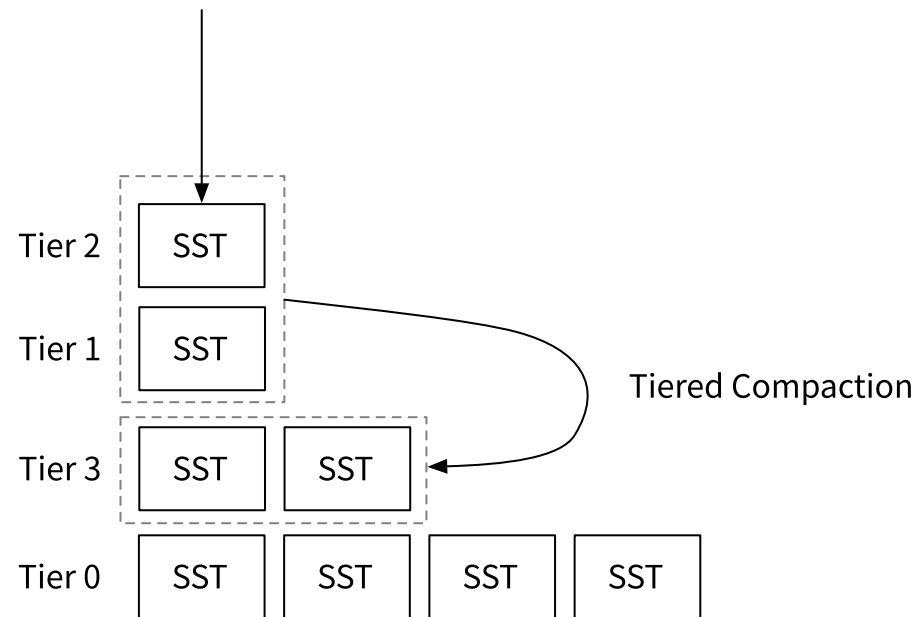
We do not provide reference answers to the questions, and feel free to discuss about them in the Discord community.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Tiered Compaction Strategy




In this chapter, you will:

- Implement a tiered compaction strategy and simulate it on the compaction simulator.
- Incorporate tiered compaction strategy into the system.

The tiered compaction we talk about in this chapter is the same as RocksDB's universal compaction. We will use these two terminologies interchangeably.

To copy the test cases into the starter code and run them,

```
cargo x copy-test --week 2 --day 3
cargo x scheck
```

 It might be helpful to take a look at [week 2 overview](#) before reading this chapter to have a general overview of compactions.

Task 1: Universal Compaction

In this chapter, you will implement RocksDB's universal compaction, which is of the tiered compaction family compaction strategies. Similar to the simple leveled compaction strategy, we only use number of files as the indicator in this compaction strategy. And when we trigger the compaction jobs, we always include a full sorted run (tier) in the compaction job.

Task 1.0: Precondition

In this task, you will need to modify:

```
src/compact/tiered.rs
```

In universal compaction, we do not use L0 SSTs in the LSM state. Instead, we directly flush new SSTs to a single sorted run (called tier). In the LSM state, `levels` will now include all tiers, where **the lowest index is the latest SST flushed**. Each element in the `levels` vector stores a tuple: level ID (used as tier ID) and the SSTs in that level. Every time you flush L0 SSTs, you should flush the SST into a tier placed at the front of the vector. The compaction simulator generates tier id based on the first SST id, and you should do the same in your implementation.

Universal compaction will only trigger tasks when the number of tiers (sorted runs) reaches `num_tiers`. Otherwise, it does not trigger any compaction.

Task 1.1: Triggered by Space Amplification Ratio

The first trigger of universal compaction is by space amplification ratio. As we discussed in the overview chapter, space amplification can be estimated by `engine_size / last_level_size`. In our implementation, we compute the space amplification ratio by `all levels except last level size / last level size`, so that the ratio can be scaled to `[0, +inf)` instead of `[1, +inf]`. This is also consistent with the RocksDB implementation.

The reason why we compute the space amplification ratio like this is because we model the engine in a way that it stores a fixed amount of user data (i.e., assume it's 100GB), and the user keeps updating the values by writing to the engine. Therefore, eventually, all keys get pushed down to the bottom-most tier, the bottom-most tier size should be equivalent to the amount of data (100GB), the upper tiers contain updates to the data that are not yet compacted to the bottom-most tier.

When `all levels except last level size / last level size >= max_size_amplification_percent * 1%`, we will need to trigger a full compaction. For example, if we have a LSM state like:

```
Tier 3: 1
Tier 2: 1 ; all levels except last level size = 2
Tier 1: 1 ; last level size = 1, 2/1=2
```

Assume `max_size_amplification_percent = 200`, we should trigger a full compaction now.

After you implement this trigger, you can run the compaction simulator. You will see:

```
cargo run --bin compaction-simulator tiered --iterations 10
```



```
=== Iteration 2 ===  
--- After Flush ---  
L3 (1): [3]  
L2 (1): [2]  
L1 (1): [1]  
--- Compaction Task ---  
compaction triggered by space amplification ratio: 200  
L3 [3] L2 [2] L1 [1] -> [4, 5, 6]  
--- After Compaction ---  
L4 (3): [3, 2, 1]
```

With this trigger, we will only trigger full compaction when it reaches the space amplification ratio.
And at the end of the simulation, you will see:

```
cargo run --bin compaction-simulator tiered
```

=== Iteration 7 ===

--- After Flush ---

L8 (1): [8]

L7 (1): [7]

L6 (1): [6]

L5 (1): [5]

L4 (1): [4]

L3 (1): [3]

L2 (1): [2]

L1 (1): [1]

--- Compaction Task ---

--- Compaction Task ---

compaction triggered by space amplification ratio: 700

L8 [8] L7 [7] L6 [6] L5 [5] L4 [4] L3 [3] L2 [2] L1 [1] -> [9, 10, 11, 12, 13, 14, 15, 16]

--- After Compaction ---

L9 (8): [8, 7, 6, 5, 4, 3, 2, 1]

--- Compaction Task ---

1 compaction triggered in this iteration

--- Statistics ---

Write Amplification: $16/8=2.000x$

Maximum Space Usage: $16/8=2.000x$

Read Amplification: 1x

=== Iteration 49 ===

--- After Flush ---

L82 (1): [82]

L81 (1): [81]

L80 (1): [80]

L79 (1): [79]

L78 (1): [78]

L77 (1): [77]

L76 (1): [76]

L75 (1): [75]

L74 (1): [74]

L73 (1): [73]

L72 (1): [72]

L71 (1): [71]

L70 (1): [70]

```

L69 (1): [69]
L68 (1): [68]
L67 (1): [67]
L66 (1): [66]
L65 (1): [65]
L64 (1): [64]
L63 (1): [63]
L62 (1): [62]
L61 (1): [61]
L60 (1): [60]
L59 (1): [59]
L58 (1): [58]
L57 (1): [57]
L33 (24): [32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 9, 10, 11,
12, 13, 14, 15, 16]
--- Compaction Task ---
--- Compaction Task ---
no compaction triggered
--- Statistics ---
Write Amplification: 82/50=1.640x
Maximum Space Usage: 50/50=1.000x
Read Amplification: 27x

```

The `num_tiers` in the compaction simulator is set to 8. However, there are far more than 8 tiers in the LSM state, which incurs large read amplification.

The current trigger only reduces space amplification. We will need to add new triggers to the compaction algorithm to reduce read amplification.

Task 1.2: Triggered by Size Ratio

The next trigger is the size ratio trigger. The trigger maintains the size ratio between the tiers. From the first tier, we compute the size of `this tier / sum of all previous tiers`. For the first

encountered tier where this value $> (100 + \text{size_ratio}) * 1\%$, we will compact all previous tiers excluding the current tier. We only do this compaction with there are more than `min_merge_width` tiers to be merged.

For example, given the following LSM state, and assume `size_ratio = 1`, and `min_merge_width = 2`. We should compact when the ratio value $> 101\%$:

```
Tier 3: 1
Tier 2: 1 ; 1 / 1 = 1
Tier 1: 1 ; 1 / (1 + 1) = 0.5, no compaction triggered
```

Example 2:

```
Tier 3: 1
Tier 2: 1 ; 1 / 1 = 1
Tier 1: 3 ; 3 / (1 + 1) = 1.5, compact tier 2+3
```

```
Tier 4: 2
Tier 1: 3
```

Example 3:

```
Tier 3: 1
Tier 2: 2 ; 2 / 1 = 2, however, it does not make sense to compact only one tier; also
note that min_merge_width=2
Tier 1: 4 ; 4 / 3 = 1.33, compact tier 2+3
```

```
Tier 4: 3
Tier 1: 4
```

With this trigger, you will observe the following in the compaction simulator:

```
cargo run --bin compaction-simulator tiered
```

```
=== Iteration 49 ===
--- After Flush ---
L119 (1): [119]
L118 (1): [118]
L114 (4): [113, 112, 111, 110]
L105 (5): [104, 103, 102, 101, 100]
L94 (6): [93, 92, 91, 90, 89, 88]
L81 (7): [80, 79, 78, 77, 76, 75, 74]
L48 (26): [47, 46, 45, 44, 43, 37, 38, 39, 40, 41, 42, 24, 25, 26, 27, 28, 29, 30, 9,
10, 11, 12, 13, 14, 15, 16]
--- Compaction Task ---
--- Compaction Task ---
no compaction triggered
--- Statistics ---
Write Amplification: 119/50=2.380x
Maximum Space Usage: 52/50=1.040x
Read Amplification: 7x
```

```
cargo run --bin compaction-simulator tiered --iterations 200 --size-only
```

```
=== Iteration 199 ===
--- After Flush ---
Levels: 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 3 4 5 6 10 15 21
28 78
no compaction triggered
--- Statistics ---
Write Amplification: 537/200=2.685x
Maximum Space Usage: 200/200=1.000x
Read Amplification: 38x
```

There will be fewer 1-SST tiers and the compaction algorithm will maintain the tiers to have smaller to larger sizes by size ratio. However, when there are more SSTs in the LSM state, there will still be

cases that we have more than `num_tiers` tiers. To limit the number of tiers, we will need another trigger.

Task 1.3: Reduce Sorted Runs

If none of the previous triggers produce compaction tasks, we will do a major compaction that merges SST files from the first up to `max_merge_tiers` tiers into one tier to reduce the number of tiers.

With this compaction trigger enabled, you will see:

```
cargo run --bin compaction-simulator-ref tiered --iterations 200 --size-only
```

```
=== Iteration 199 ===  
--- After Flush ---  
Levels: 0 1 1 4 5 21 28 140  
no compaction triggered  
--- Statistics ---  
Write Amplification: 742/200=3.710x  
Maximum Space Usage: 280/200=1.400x  
Read Amplification: 7x
```

You can also try tiered compaction with more number of tiers:

```
cargo run --bin compaction-simulator tiered --iterations 200 --size-only --num-tiers  
16
```

```
=== Iteration 199 ===  
--- After Flush ---  
Levels: 0 1 1 1 1 1 1 1 1 1 1 15 175  
no compaction triggered  
--- Statistics ---  
Write Amplification: 607/200=3.035x  
Maximum Space Usage: 350/200=1.750x  
Read Amplification: 12x
```

Note: we do not provide fine-grained unit tests for this part. You can run the compaction simulator and compare with the output of the reference solution to see if your implementation is correct.

Task 2: Integrate with the Read Path

In this task, you will need to modify:

```
src/compact.rs  
src/lsm_storage.rs
```

As tiered compaction does not use the L0 level of the LSM state, you should directly flush your memtables to a new tier instead of as an L0 SST. You can use

`self.compaction_controller.flush_to_l0()` to know whether to flush to L0. You may use the first output SST id as the level/tier id for your new sorted run. You will also need to modify your compaction process to construct merge iterators for tiered compaction jobs.

Related Readings

[Universal Compaction - RocksDB Wiki](#)

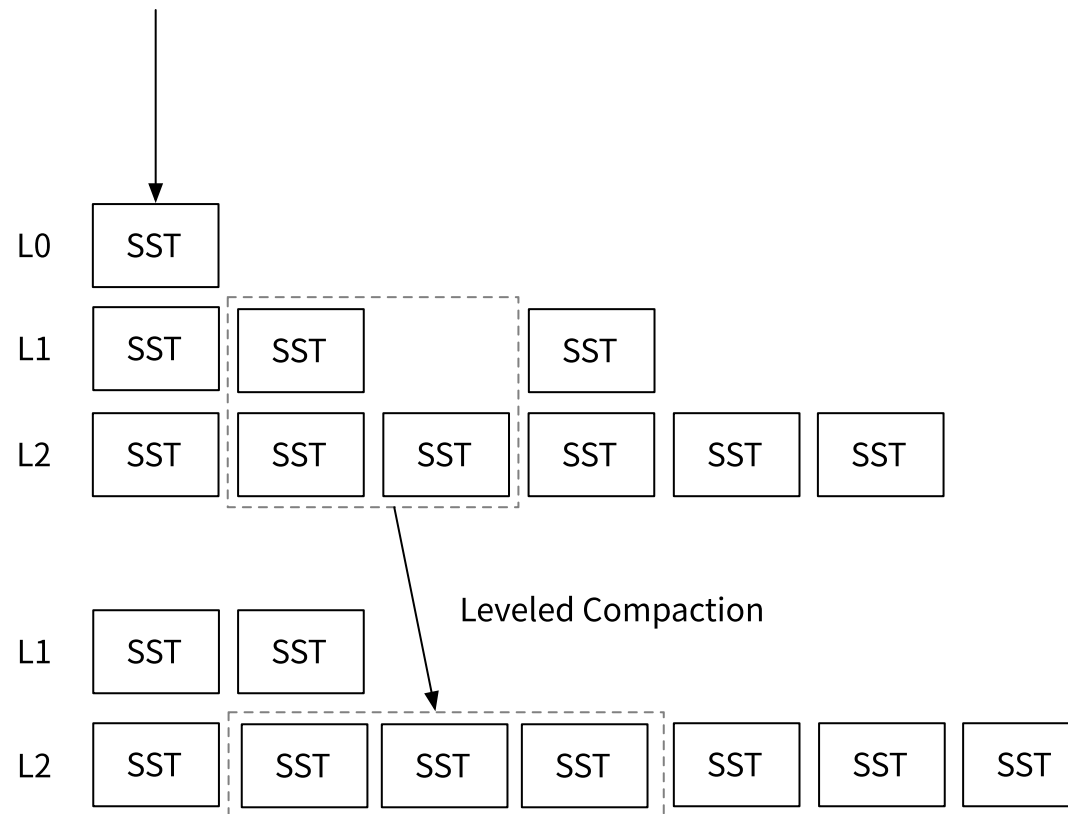
Test Your Understanding

- What is the estimated write amplification of leveled compaction? (Okay this is hard to estimate... But what if without the last *reduce sorted run* trigger?)
- What is the estimated read amplification of leveled compaction?
- What are the pros/cons of universal compaction compared with simple leveled/tiered compaction?
- How much storage space is it required (compared with user data size) to run universal compaction?
- Can we merge two tiers that are not adjacent in the LSM state?
- What happens if compaction speed cannot keep up with the SST flushes for tiered compaction?
- What might needs to be considered if the system schedules multiple compaction tasks in parallel?
- SSDs also write its own logs (basically it is a log-structured storage). If the SSD has a write amplification of 2x, what is the end-to-end write amplification of the whole system? Related: [ZNS: Avoiding the Block Interface Tax for Flash-based SSDs](#).
- Consider the case that the user chooses to keep a large number of sorted runs (i.e., 300) for tiered compaction. To make the read path faster, is it a good idea to keep some data structure that helps reduce the time complexity (i.e., to $O(\log n)$) of finding SSTs to read in each layer for some key ranges? Note that normally, you will need to do a binary search in each sorted run to find the key ranges that you will need to read. (Check out Neon's [layer map](#) implementation!)

We do not provide reference answers to the questions, and feel free to discuss about them in the Discord community.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).
Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.
mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Leveled Compaction Strategy




In this chapter, you will:

- Implement a leveled compaction strategy and simulate it on the compaction simulator.
- Incorporate leveled compaction strategy into the system.

To copy the test cases into the starter code and run them,

```
cargo x copy-test --week 2 --day 4  
cargo x scheck
```

 It might be helpful to take a look at [week 2 overview](#) before reading this chapter to have a general overview of compactions.

Task 1: Leveled Compaction

In chapter 2 day 2, you have implemented the simple leveled compaction strategies. However, the implementation has a few problems:

- Compaction always include a full level. Note that you cannot remove the old files until you finish the compaction, and therefore, your storage engine might use 2x storage space while the compaction is going on (if it is a full compaction). Tiered compaction has the same problem. In this chapter, we will implement partial compaction that we select one SST from the upper level for compaction, instead of the full level.
- SSTs may be compacted across empty levels. As you have seen in the compaction simulator, when the LSM state is empty, and the engine flushes some L0 SSTs, these SSTs will be first compacted to L1, then from L1 to L2, etc. An optimal strategy is to directly place the SST from L0 to the lowest level possible, so as to avoid unnecessary write amplification.

In this chapter, you will implement a production-ready leveled compaction strategy. The strategy is the same as RocksDB's leveled compaction. You will need to modify:

```
src/compact/leveled.rs
```

To run the compaction simulator,

```
cargo run --bin compaction-simulator leveled
```

Task 1.1: Compute Target Sizes

In this compaction strategy, you will need to know the first/last key of each SST and the size of the SSTs. The compaction simulator will set up some mock SSTs for you to access.

You will need to compute the target sizes of the levels. Assume `base_level_size_mb` is 200MB and the number of levels (except L0) is 6. When the LSM state is empty, the target sizes will be:

```
[0 0 0 0 0 200MB]
```

Before the bottom level exceeds `base_level_size_mb`, all other intermediate levels will have target sizes of 0. The idea is that when the total amount of data is small, it's wasteful to create intermediate levels.

When the bottom level reaches or exceeds `base_level_size_mb`, we will compute the target size of the other levels by dividing the `level_size_multiplier` from the size. Assume the bottom level contains 300MB of data, and `level_size_multiplier=10`.

```
0 0 0 0 30MB 300MB
```

In addition, at most *one* level can have a positive target size below `base_level_size_mb`. Assume we now have 30GB files in the last level, the target sizes will be,

```
0 0 30MB 300MB 3GB 30GB
```

Notice in this case L1 and L2 have target size of 0, and L3 is the only level with a positive target size below `base_level_size_mb`.

Task 1.2: Decide Base Level

Now, let us solve the problem that SSTs may be compacted across empty levels in the simple leveled compaction strategy. When we compact L0 SSTs with lower levels, we do not directly put it to L1. Instead, we compact it with the first level with `target size > 0`. For example, when the target level sizes are:

```
0 0 0 0 30MB 300MB
```

We will compact L0 SSTs with L5 SSTs if the number of L0 SSTs reaches the `level0_file_num_compaction_trigger` threshold.

Now, you can generate L0 compaction tasks and run the compaction simulator.

```
--- After Flush ---
```

```
L0 (1): [23]
```

```
L1 (0): []
```

```
L2 (0): []
```

```
L3 (2): [19, 20]
```

```
L4 (6): [11, 12, 7, 8, 9, 10]
```

```
...
```

```
--- After Flush ---
```

```
L0 (2): [102, 103]
```

```
L1 (0): []
```

```
L2 (0): []
```

```
L3 (18): [42, 65, 86, 87, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 61, 62, 52, 34]
```

```
L4 (6): [11, 12, 7, 8, 9, 10]
```

The number of levels in the compaction simulator is 4. Therefore, the SSTs should be directly flushed to L3/L4.

Task 1.3: Decide Level Priorities

Now that we will need to handle compactions below L0. L0 compaction always has the top priority, thus you should compact L0 with other levels first if it reaches the threshold. After that, we can compute the compaction priorities of each level by $\text{current_size} / \text{target_size}$. We only compact levels with this ratio > 1.0 . The one with the largest ratio will be chosen for compaction with the lower level. For example, if we have:

```
L3: 200MB, target_size=20MB  
L4: 202MB, target_size=200MB  
L5: 1.9GB, target_size=2GB  
L6: 20GB, target_size=20GB
```

The priority of compaction will be:

```
L3: 200MB/20MB = 10.0  
L4: 202MB/200MB = 1.01  
L5: 1.9GB/2GB = 0.95
```

L3 and L4 needs to be compacted with their lower level respectively, while L5 does not. And L3 has a larger ratio, and therefore we will produce a compaction task of L3 and L4. After the compaction is done, it is likely that we will schedule compactions of L4 and L5.

Task 1.4: Select SST to Compact

Now, let us solve the problem that compaction always include a full level from the simple leveled compaction strategy. When we decide to compact two levels, we always select the oldest SST from the upper level. You can know the time that the SST is produced by comparing the SST id.

There are other ways of choosing the compacting SST, for example, by looking into the number of delete tombstones. You can implement this as part of the bonus task.

After you choose the upper level SST, you will need to find all SSTs in the lower level with overlapping keys of the upper level SST. Then, you can generate a compaction task that contain exactly one SST in the upper level and overlapping SSTs in the lower level.

When the compaction completes, you will need to remove the SSTs from the state and insert new SSTs into the correct place. Note that you should keep SST ids ordered by first keys in all levels except L0.

Running the compaction simulator, you should see:

```
--- After Compaction ---  
L0 (0): []  
L1 (4): [222, 223, 208, 209]  
L2 (5): [206, 196, 207, 212, 165]  
L3 (11): [166, 120, 143, 144, 179, 148, 167, 140, 189, 180, 190]  
L4 (22): [113, 85, 86, 36, 46, 37, 146, 100, 147, 203, 102, 103, 65, 81, 105, 75, 82,  
95, 96, 97, 152, 153]
```

The sizes of the levels should be kept under the level multiplier ratio. And the compaction task:

```
Upper L1 [224.sst 7cd080e..=33d79d04]  
Lower L2 [210.sst 1c657df4..=31a00e1b, 211.sst 31a00e1c..=46da9e43] -> [228.sst  
7cd080e..=1cd18f74, 229.sst 1cd18f75..=31d616db, 230.sst 31d616dc..=46da9e43]
```

...should only have one SST from the upper layer.

Note: we do not provide fine-grained unit tests for this part. You can run the compaction simulator and compare with the output of the reference solution to see if your implementation is correct.

Task 2: Integrate with the Read Path

In this task, you will need to modify:

```
src/compact.rs  
src/lsm_storage.rs
```

The implementation should be similar to simple leveled compaction. Remember to change both get/scan read path and the compaction iterators.

Related Readings

[Leveled Compaction - RocksDB Wiki](#)

Test Your Understanding

- What is the estimated write amplification of leveled compaction?
- What is the estimated read amplification of leveled compaction?
- Finding a good key split point for compaction may potentially reduce the write amplification, or it does not matter at all? (Consider that case that the user write keys beginning with some prefixes, `00` and `01`. The number of keys under these two prefixes are different and their write patterns are different. If we can always split `00` and `01` into different SSTs...)
- Imagine that a user was using tiered (universal) compaction before and wants to migrate to leveled compaction. What might be the challenges of this migration? And how to do the migration?
- And if we do it reversely, what if the user wants to migrate from leveled compaction to tiered compaction?

- What happens if compaction speed cannot keep up with the SST flushes for leveled compaction?
- What might need to be considered if the system schedules multiple compaction tasks in parallel?
- What is the peak storage usage for leveled compaction? Compared with universal compaction?
- Is it true that with a lower `level_size_multiplier`, you can always get a lower write amplification?
- What needs to be done if a user not using compaction at all decides to migrate to leveled compaction?
- Some people propose to do intra-L0 compaction (compact L0 tables and still put them in L0) before pushing them to lower layers. What might be the benefits of doing so? (Might be related: [PebblesDB SOSP'17](#))
- Consider the case that the upper level has two tables of `[100, 200]`, `[201, 300]` and the lower level has `[50, 150]`, `[151, 250]`, `[251, 350]`. In this case, do you still want to compact one file in the upper level at a time? Why?

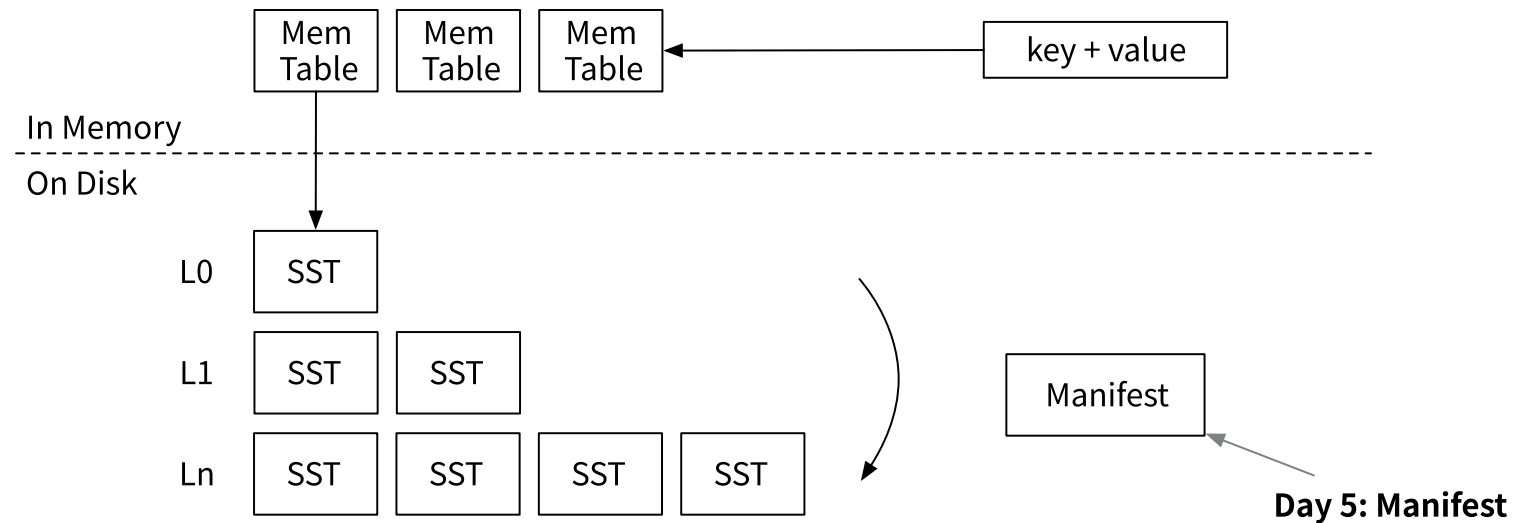
We do not provide reference answers to the questions, and feel free to discuss about them in the Discord community.

Bonus Tasks

- **SST Ingestion.** A common optimization of data migration / batch import in LSM trees is to ask the upstream to generate SST files of their data, and directly place these files in the LSM state without going through the write path.
- **SST Selection.** Instead of selecting the oldest SST, you may think of other heuristics to choose the SST to compact.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).
Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.
mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Manifest



In this chapter, you will:

- Implement encoding and decoding of the manifest file.
- Recover from the manifest when the system restarts.

To copy the test cases into the starter code and run them,

```
cargo x copy-test --week 2 --day 5
cargo x scheck
```

Task 1: Manifest Encoding

The system uses a manifest file to record all operations happened in the engine. Currently, there are only two types of them: compaction and SST flush. When the engine restarts, it will read the manifest file, reconstruct the state, and load the SST files on the disk.

There are many approaches to storing the LSM state. One of the easiest way is to simply store the full state into a JSON file. Every time we do a compaction or flush a new SST, we can serialize the entire LSM state into a file. The problem with this approach is that when the database gets super large (i.e., 10k SSTs), writing the manifest to the disk would be super slow. Therefore, we designed the manifest to be a append-only file.

In this task, you will need to modify:

```
src/manifest.rs
```

We encode the manifest records using JSON. You may use `serde_json::to_vec` to encode a manifest record to a json, write it to the manifest file, and do a `fsync`. When you read from the manifest file, you may use `serde_json::Deserializer::from_slice` and it will return a stream of records. You do not need to store the record length or so, as `serde_json` can automatically find the split of the records.

The manifest format is like:

```
| JSON record | JSON record | JSON record | JSON record |
```

Again, note that we do not record the information of how many bytes each record has.

After the engine runs for several hours, the manifest file might get very large. At that time, you may periodically compact the manifest file to store the current snapshot and truncate the logs. This is an optimization you may implement as part of bonus tasks.

Task 2: Write Manifests

You can now go ahead and modify your LSM engine to write manifests when necessary. In this task, you will need to modify:

```
src/lsm_storage.rs  
src/compact.rs
```

For now, we only use two types of the manifest records: SST flush and compaction. SST flush record stores the SST id that gets flushed to the disk. Compaction record stores the compaction task and the produced SST ids. Every time you write some new files to the disk, first sync the files and the storage directory, and then write to the manifest and sync the manifest. The manifest file should be written to `<path>/MANIFEST`.

To sync the directory, you may implement the `sync_dir` function, where you can use `File::open(dir).sync_all()? to sync it. On Linux, directory is a file that contains the list of files in the directory. By doing fsync on the directory, you will ensure that the newly-written (or removed) files can be visible to the user if the power goes off.`

Remember to write a compaction manifest record for both the background compaction trigger (leveled/simple/universal) and when the user requests to do a force compaction.

Task 3: Flush on Close

In this task, you will need to modify:

```
src/lsm_storage.rs
```

You will need to implement the `close` function. If `self.options.enable_wal = false` (we will cover WAL in the next chapter), you should flush all memtables to the disk before stopping the storage engine, so that all user changes will be persisted.

Task 4: Recover from the State

In this task, you will need to modify:

```
src/lsm_storage.rs
```

Now, you may modify the `open` function to recover the engine state from the manifest file. To recover it, you will need to first generate the list of SSTs you will need to load. You can do this by calling `apply_compaction_result` and recover SST ids in the LSM state. After that, you may iterate the state and load all SSTs (update the sstables hash map). During the process, you will need to compute the maximum SST id and update the `next_sst_id` field. After that, you may create a new memtable using that id and increment the id by one.

If you have implemented leveled compaction, you might have sorted the SSTs every time you apply the compaction result. However, with manifest recover, your sorting logic will be broken, because during the recovery process, you cannot know the start key and the end key of each of the SST. To resolve this, you will need to read the `in_recovery` flag of the `apply_compaction_result` function. During the recovery process, you should not attempt to retrieve the first key of the SST. After the LSM state is recovered and all SSTs are opened, you can do a sorting at the end of the recovery process.

Optionally, you may include the start key and the end key of each of the SSTs in the manifest. This strategy is used in RocksDB/BadgerDB, so that you do not need to distinguish the recovery mode and the normal mode during the compaction apply process.

You may use the mini-lsm-cli to test your implementation.

```
cargo run --bin mini-lsm-cli  
fill 1000 2000  
close  
cargo run --bin mini-lsm-cli  
get 1500
```

Test Your Understanding

- When do you need to call `fsync`? Why do you need to `fsync` the directory?
- What are the places you will need to write to the manifest?
- Consider an alternative implementation of an LSM engine that does not use a manifest file. Instead, it records the level/tier information in the header of each file, scans the storage directory every time it restarts, and recover the LSM state solely from the files present in the directory. Is it possible to correctly maintain the LSM state in this implementation and what might be the problems/challenges with that?
- Currently, we create all SST/concat iterators before creating the merge iterator, which means that we have to load the first block of the first SST in all levels into memory before starting the scanning process. We have start/end key in the manifest, and is it possible to leverage this information to delay the loading of the data blocks and make the time to return the first key-value pair faster?
- Is it possible not to store the tier/level information in the manifest? i.e., we only store the list of SSTs we have in the manifest without the level information, and rebuild the tier/level using the key range and timestamp information (SST metadata).

Bonus Tasks

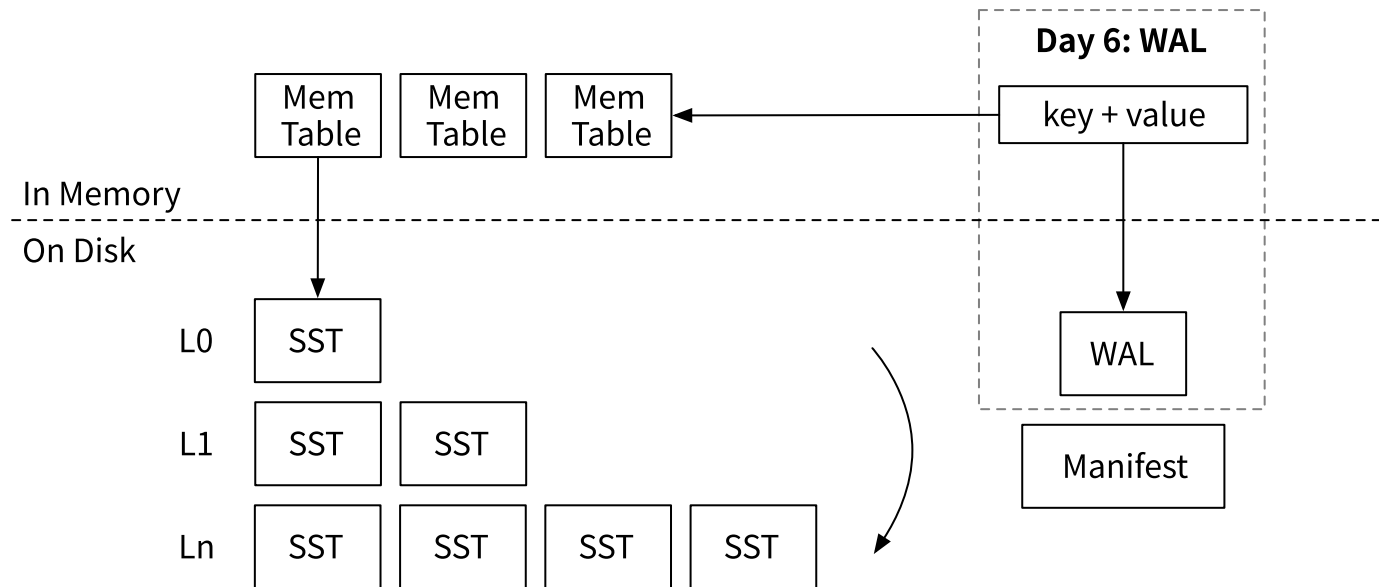
- **Manifest Compaction.** When the number of logs in the manifest file gets too large, you can rewrite the manifest file to only store the current snapshot and append new logs to that file.
- **Parallel Open.** After you collect the list of SSTs to open, you can open and decode them in parallel, instead of doing it one by one, therefore accelerating the recovery process.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Write-Ahead Log (WAL)



In this chapter, you will:

- Implement encoding and decoding of the write-ahead log file.
- Recover memtables from the WALs when the system restarts.

To copy the test cases into the starter code and run them,

```
cargo x copy-test --week 2 --day 6
cargo x scheck
```

Task 1: WAL Encoding

In this task, you will need to modify:

```
src/wal.rs
```

In the previous chapter, we have implemented the manifest file, so that the LSM state can be persisted. And we implemented the `close` function to flush all memtables to SSTs before stopping the engine. Now, what if the system crashes (i.e., powered off)? We can log memtable modifications to WAL (write-ahead log), and recover WALs when restarting the database. WAL is only enabled when `self.options.enable_wal = true`.

The WAL encoding is simply a list of key-value pairs.

```
| key_len | key | value_len | value |
```

You will also need to implement the `recover` function to read the WAL and recover the state of a memtable.

Note that we are using a `BufWriter` for writing the WAL. Using a `BufWriter` can reduce the number of syscalls into the OS, so as to reduce the latency of the write path. The data is not guaranteed to be written to the disk when the user modifies a key. Instead, the engine only guarantee that the data is persisted when `sync` is called. To correctly persist the data to the disk, you will need to first flush the data from the buffer writer to the file object by calling `flush()`, and then do a `fsync` on the file by using `get_mut().sync_all()`. Note that you *only* need to `fsync` when the engine's `sync` gets called. You *do not* need to `fsync` every time on writing data.

Task 2: Integrate WALs

In this task, you will need to modify:

```
src/mem_table.rs  
src/wal.rs  
src/lsm_storage.rs
```

`MemTable` has a WAL field. If the `wal` field is set to `Some(wal)`, you will need to append to the WAL when updating the memtable. In your LSM engine, you will need to create WALs if `enable_wal = true`. You will also need update the manifest using the `ManifestRecord::NewMemtable` record when new memtable is created.

You can create a memtable with WAL by using the `create_with_wal` function. WAL should be written to `<memtable_id>.wal` in the storage directory. The memtable id should be the same as the SST id if this memtable gets flushed as an L0 SST.

Task 3: Recover from the WALs

In this task, you will need to modify:

```
src/lsm_storage.rs
```

If WAL is enabled, you will need to recover the memtables based on WALs when loading the database. You will also need to implement the `sync` function of the database. The basic guarantee of `sync` is that the engine is sure that the data is persisted to the disk (and will be recovered when it restarts). To achieve this, you can simply sync the WAL corresponding to the current memtable.

```
cargo run --bin mini-lsm-cli -- --enable-wal
```

Remember to recover the correct `next_sst_id` from the state, which should be `max{memtable id, sst id} + 1`. In your `close` function, you should not flush memtables to SSTs if `enable_wal` is set to true, as WAL itself provides persistency. You should wait until all compaction and flush threads to exit before closing the database.

Test Your Understanding

- When should you call `fsync` in your engine? What happens if you call `fsync` too often (i.e., on every put key request)?
- How costly is the `fsync` operation in general on an SSD (solid state drive)?
- When can you tell the user that their modifications (put/delete) have been persisted?
- How can you handle corrupted data in WAL?
- Is it possible to design an LSM engine without WAL (i.e., use L0 as WAL)? What will be the implications of this design?

We do not provide reference answers to the questions, and feel free to discuss about them in the Discord community.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Batch Write and Checksums

In the previous chapter, you already built a full LSM-based storage engine. At the end of this week, we will implement some easy but important optimizations of the storage engine. Welcome to Mini-LSM's week 2 snack time!

In this chapter, you will:

- Implement the batch write interface.
- Add checksums to the blocks, SST metadata, manifest, and WALs.

Note: We do not have unit tests for this chapter. As long as you pass all previous tests and ensure checksums are properly encoded in your file format, it would be fine.

Task 1: Write Batch Interface

In this task, we will prepare for week 3 of this course by adding a write batch API. You will need to modify:

```
src/lsm_storage.rs
```

The user provides `write_batch` with a batch of records to be written to the database. The records are `WriteBatchRecord<T: AsRef<[u8]>>`, and therefore it can be either `Bytes`, `&[u8]` or `Vec<u8>`. There are two types of records: delete and put. You may handle them in the same way as your `put` and `delete` function.

After that, you may refactor your original `put` and `delete` function to call `write_batch`.

You should pass all test cases in previous chapters after implementing this functionality.

Task 2: Block Checksum

In this task, you will need to add a block checksum at the end of each block when encoding the SST. You will need to modify:

```
src/table/builder.rs
src/table.rs
```

The format of the SST will be changed to:

	Block Section						
Meta Section							

data block	checksum	...	data block	checksum	metadata	meta block offset	
bloom filter	bloom filter offset						
varlen		u32		varlen		u32	
varlen		u32					

We use `crc32` as our checksum algorithm. You can use `crc32fast::hash` to generate the checksum for the block after building a block.

Usually, when user specify the target block size in the storage options, the size should include both block content and checksum. For example, if the target block size is 4096, and the checksum takes 4 bytes, the actual block content target size should be 4092. However, to avoid breaking previous test

cases and for simplicity, in our course, we will **still** use the target block size as the target content size, and simply append the checksum at the end of the block.

When you read the block, you should verify the checksum in `read_block` correctly generate the slices for the block content. You should pass all test cases in previous chapters after implementing this functionality.

Task 3: SST Meta Checksum

In this task, you will need to add a block checksum for bloom filters and block metadata:

```
src/table.rs
src/table/bloom.rs
src/table/builder.rs
```

	Meta Section					

no. of block metadata checksum meta block offset bloom filter checksum						
bloom filter offset						
u32 varlen u32 u32 varlen u32						
u32						

You will need to add a checksum at the end of the bloom filter in `Bloom::encode` and `Bloom::decode`. Note that most of our APIs take an existing buffer that the implementation will write into, for example, `Bloom::encode`. Therefore, you should record the offset of the beginning of

the bloom filter before writing the encoded content, and only checksum the bloom filter itself instead of the whole buffer.

After that, you can add a checksum at the end of block metadata. You might find it helpful to also add a length of metadata at the beginning of the section, so that it will be easier to know where to stop when decoding the block metadata.

Task 4: WAL Checksum

In this task, you will need to modify:

```
src/wal.rs
```

We will do a per-record checksum in the write-ahead log. To do this, you have two choices:

- Generate a buffer of the key-value record, and use `crc32fast::hash` to compute the checksum at once.
- Write one field at a time (e.g., key length, key slice), and use a `crc32fast::Hasher` to compute the checksum incrementally on each field.

This is up to your choice and you will need to *choose your own adventure*. Both method should produce exactly the same result, as long as you handle little endian / big endian correctly. The new WAL encoding should be like:

```
| key_len | key | value_len | value | checksum |
```


Task 5: Manifest Checksum

Lastly, let us add a checksum on the manifest file. Manifest is similar to a WAL, except that previously, we do not store the length of each record. To make the implementation easier, we now add a header of record length at the beginning of a record, and add a checksum at the end of the record.

The new manifest format is like:

```
| len | JSON record | checksum | len | JSON record | checksum | len | JSON record |  
checksum |
```

After implementing everything, you should pass all previous test cases. We do not provide new test cases in this chapter.

Test Your Understanding

- Consider the case that an LSM storage engine only provides `write_batch` as the write interface (instead of single put + delete). Is it possible to implement it as follows: there is a single write thread with an mpsc channel receiver to get the changes, and all threads send write batches to the write thread. The write thread is the single point to write to the database. What are the pros/cons of this implementation? (Congrats if you do so you get BadgerDB!)
- Is it okay to put all block checksums altogether at the end of the SST file instead of store it along with the block? Why?

We do not provide reference answers to the questions, and feel free to discuss about them in the Discord community.

Bonus Tasks

- **Recovering when Corruption.** If there is a checksum error, open the database in a safe mode so that no writes can be performed and non-corrupted data can still be retrieved.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Week 3 Overview: Multi-Version Concurrency Control

In this part, you will implement MVCC over the LSM engine that you have built in the previous two weeks. We will add timestamp encoding in the keys to maintain multiple versions of a key, and change some part of the engine to ensure old data are either retained or garbage-collected based on whether there are users reading an old version.

The general approach of the MVCC part in this course is inspired and partially based on [BadgerDB](#).

The key of MVCC is to store and access multiple versions of a key in the storage engine. Therefore, we will need to change the key format to `user_key + timestamp (u64)`. And on the user interface side, we will need to have new APIs to help users to gain access to a history version. In summary, we will add a monotonically-increasing timestamp to the key.

In previous parts, we assumed that newer keys are in the upper level of the LSM tree, and older keys are in the lower level of the LSM tree. During compaction, we only keep the latest version of a key if multiple versions are found in multiple levels, and the compaction process will ensure that newer keys will be kept on the upper level by only merging adjacent levels/tiers. In the MVCC implementation, the key with a larger timestamp is the newest key. During compaction, we can only remove the key if no user is accessing an older version of the database. Though not keeping the latest version of key in the upper level may still yield a correct result for the MVCC LSM implementation, in our course, we choose to keep the invariant, and if there are multiple versions of a key, a later version will always appear in a upper level.

Generally, there are two ways of utilizing a storage engine with MVCC support. If the user uses the engine as a standalone component and do not want to manually assign the timestamps of the keys, they will use transaction APIs to store and retrieve data from the storage engine. Timestamps are transparent to the users. The other way is to integrate the storage engine into the system, where the

user manages the timestamps by themselves. To compare these two approaches, we can look at the APIs they provide. We use the terminologies of BadgerDB to describe these two usages: the one that hides the timestamp is *un-managed mode*, and the one that gives the user full control is *managed mode*.

Managed Mode APIs

```
get(key, read_timestamp) -> (value, write_timestamp)
scan(key_range, read_timestamp) -> iterator<key, value, write_timestamp>
put/delete/write_batch(key, timestamp)
set_watermark(timestamp) # we will talk about watermarks soon!
```

Un-managed/Normal Mode APIs

```
get(key) -> value
scan(key_range) -> iterator<key, value>
start_transaction() -> txn
txn.put/delete/write_batch(key, timestamp)
```

As you can see, the managed mode APIs requires the user to provide a timestamp when doing the operations. The timestamp may come from some centralized timestamp systems, or from the logs of other systems (i.e., Postgres logical replication log). The user will need to specify a watermark, which is the versions below which the engine can remove.

And for the un-managed APIs, it is the same as what we have implemented before, except that the user will need to write and read data by creating a transaction. When the user creates a transaction, they can gain a consistent state of the database (which is a snapshot). Even if other threads/transactions write data into the database, these data will be invisible to the ongoing transaction. The storage engine manages the timestamps internally and do not expose them to the user.

In this week, we will first spend 3 days doing a refactor on table format and memtables. We will change the key format to key slice and a timestamp. After that, we will implement necessary APIs to

provide consistent snapshots and transactions.

We have 7 chapters (days) in this part:

- [Day 1: Timestamp Key Refactor](#). You will change the `key` module to the MVCC one and refactor your system to use key with timestamp.
- [Day 2: Snapshot Read - Memtables and Timestamps](#). You will refactor the memtable and the write path to support multiple version reads/writes.
- [Day 3: Snapshot Read - Transaction API](#). You will implement the transaction API and finish the rest part of read/write path so as to support snapshot reads.
- [Day 4: Watermark and Garbage Collection](#). You will implement the watermark computation algorithm and implement garbage collection at compaction time to remove old versions.
- [Day 5: Transaction and Optimistic Concurrency Control](#). You will create a private workspace for all transactions and commit them in batch so that the modifications of a transaction will not be visible to other transactions.
- [Day 6: Serializable Snapshot Isolation](#). You will implement the OCC serializable checks to ensure the modifications to the database is serializable and abort transactions that violates serializability.
- [Day 7: Compaction Filter](#). At the end of the week, we will generalize the compaction-time garbage collection logic to a compaction filter, that removes data at compaction time as user's requirement.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Timestamp Key Encoding + Refactor

In this chapter, you will:

- Refactor your implementation to use key+ts representation.
- Make your code compile with the new key representation.

To run test cases,

```
cargo x copy-test --week 3 --day 1  
cargo x scheck
```

Note: The MVCC subsystem is not fully implemented until week 3 day 2. You only need to pass week 3 day 1 tests and all week 1 tests at the end of this day. Week 2 tests won't work because of compaction.

Task 0: Use MVCC Key Encoding

You will need to replace the key encoding module to the MVCC one. We have removed some interfaces from the original key module and implemented new comparators for the keys. If you followed the instructions in the previous chapters and did not use `into_inner` on the key, you should pass all test cases on day 3 after all the refactors. Otherwise, you will need to look carefully on the places where you only compare the keys without looking at the timestamps.

Specifically, the key type definition has been changed from:

```
pub struct Key<T: AsRef<[u8]>>>(T);
```

...to:

```
pub struct Key<T: AsRef<[u8]>>(T /* user key */, u64 /* timestamp */);
```

...where we have a timestamp associated with the keys. We only use this key representation internally in the system. On the user interface side, we do not ask users to provide a timestamp, and therefore some structures still use `&[u8]` instead of `KeySlice` in the engine. We will cover the places where we need to change the signature of the functions later. For now, you only need to run,

```
cp mini-lsm-mvcc/src/key.rs mini-lsm-starter/src/
```

There are other ways of storing the timestamp. For example, we can still use the `pub struct Key<T: AsRef<[u8]>>(T);` representation, but assume the last 8 bytes of the key is the timestamp. You can also implement this as part of the bonus tasks.

```
Alternative key representation: | user_key (varlen) | ts (8 bytes) | in a single slice
Our key representation: | user_key slice | ts (u64) |
```

In the key+ts encoding, the key with a smallest user key and a largest timestamp will be ordered first. For example,

```
("a", 233) < ("a", 0) < ("b", 233) < ("b", 0)
```

Task 1: Encode Timestamps in Blocks

The first thing you will notice is that your code might not compile after replacing the key module. In this chapter, all you need to do is to make it compile. In this task, you will need to modify:

```
src/block.rs
src/block/builder.rs
src/block/iterator.rs
```

You will notice that `raw_ref()` and `len()` are removed from the key API. Instead, we have `key_ref` to retrieve the slice of the user key, and `key_len` to retrieve the length of the user key. You will need to refactor your block builder and decoding implementation to use the new APIs. Also, you will need to change your block encoding to encode the timestamps. In `BlockBuilder::add`, you should do that. The new block entry record will be like:

```
key_overlap_len (u16) | remaining_key_len (u16) | key (remaining_key_len) | timestamp
(u64)
```

You may use `raw_len` to estimate the space required by a key, and store the timestamp after the user key.

After you change the block encoding, you will need to change the decoding in both `block.rs` and `iterator.rs` accordingly.

Task 2: Encoding Timestamps in SSTs

Then, you can go ahead and modify the table format,

```
src/table.rs
src/table/builder.rs
src/table/iterator.rs
```

Specifically, you will need to change your block meta encoding to include the timestamps of the keys. All other code remains the same. As we use `keySlice` in the signature of all functions (i.e.,

seek, add), the new key comparator should automatically order the keys by user key and timestamps.

In your table builder, you may directly use the `key_ref()` to build the bloom filter. This naturally creates a prefix bloom filter for your SSTs.

Task 3: LSM Iterators

As we use associated generic type to make most of our iterators work for different key types (i.e., `&[u8]` and `KeySlice<'_>`), we do not need to modify merge iterators and concat iterators if they are implemented correctly. The `LsmIterator` is the place where we strip the timestamp from the internal key representation and return the latest version of a key to the user. In this task, you will need to modify:

```
src/lsm_iterator.rs
```

For now, we do not modify the logic of `LsmIterator` to only keep the latest version of a key. We simply make it compile by appending a timestamp to the user key when passing the key to the inner iterator, and stripping the timestamp from a key when returning to the user. The behavior of your LSM iterator for now should be returning multiple versions of the same key to the user.

Task 4: Memtable

For now, we keep the logic of the memtable. We return a key slice to the user and flush SSTs with `TS_DEFAULT`. We will change the memtable to be MVCC in the next chapter. In this task, you will need to modify:

`src/mem_table.rs`

Task 5: Engine Read Path

In this task, you will need to modify,

`src/lsm_storage.rs`

Now that we have a timestamp in the key, and when creating the iterators, we will need to seek a key with a timestamp instead of only the user key. You can create a key slice with `TS_RANGE_BEGIN`, which is the largest ts.

When you check if a user key is in a table, you can simply compare the user key without comparing the timestamp.

At this point, you should build your implementation and pass all week 1 test cases. All keys stored in the system will use `TS_DEFAULT` (which is timestamp 0). We will make the engine fully multi-version and pass all test cases in the next two chapters.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Snapshot Read - Memtables and Timestamps

In this chapter, you will:

- Refactor your memtable/WAL to store multiple versions of a key.
- Implement the new engine write path to assign each key a timestamp.
- Make your compaction process aware of multi-version keys.
- Implement the new engine read path to return the latest version of a key.

During the refactor, you might need to change the signature of some functions from `&self` to `self: &Arc<Self>` as necessary.

To run test cases,

```
cargo x copy-test --week 3 --day 2
cargo x scheck
```

Note: You will also need to pass everything `<= 2.4` after finishing this chapter.

Task 1: MemTable, Write-Ahead Log, and Read Path

In this task, you will need to modify:

```
src/wal.rs
src/mem_table.rs
src/lsm_storage.rs
```

We have already made most of the keys in the engine to be a `KeySlice`, which contains a bytes key and a timestamp. However, some part of our system still did not consider the timestamps. In our first task, you will need to modify your memtable and WAL implementation to take timestamps into account.

You will need to first change the type of the `SkipMap` stored in your memtable.

```
pub struct MemTable {  
    // map: Arc<SkipMap<Bytes, Bytes>>,  
    map: Arc<SkipMap<KeyBytes, Bytes>>, // Bytes -> KeyBytes  
    // ...  
}
```

After that, you can continue to fix all compiler errors so as to complete this task.

MemTable::get

We keep the `get` interface so that the test cases can still probe a specific version of a key in the memtable. This interface should not be used in your read path after finishing this task. Given that we store `KeyBytes`, which is `(Bytes, u64)` in the skiplist, while the user probe the `KeySlice`, which is `(&[u8], u64)`. We have to find a way to convert the latter to a reference of the former, so that we can retrieve the data in the skiplist.

To do this, you may use unsafe code to force cast the `&[u8]` to be static and use

`Bytes::from_static` to create a bytes object from a static slice. This is sound because `Bytes` will not try to free the memory of the slice as it is assumed static.

► Spoilers: Convert u8 slice to Bytes

This was not a problem because what we had before is `Bytes` and `&[u8]`, where `Bytes` implements `Borrow<[u8]>`.

MemTable::put

The signature should be changed to `fn put(&self, key: KeySlice, value: &[u8])` and You will need to convert a key slice to a `KeyBytes` in your implementation.

MemTable::scan

The signature should be changed to `fn scan(&self, lower: Bound<KeySlice>, upper: Bound<KeySlice>) -> MemTableIterator`. You will need to convert `KeySlice` to `KeyBytes` and use these as `SkipMap::range` parameters.

MemTable::flush

Instead of using the default timestamp, you should now use the key timestamp when flushing the memtable to the SST.

MemTableIterator

It should now store `(KeyBytes, Bytes)` and the return key type should be `KeySlice`.

Wal::recover and **Wal::put**

Write-ahead log should now accept a key slice instead of a user key slice. When serializing and deserializing the WAL record, you should put timestamp into the WAL file and do checksum over the timestamp and all other fields you had before.

The WAL format is as follows:

```
| key_len (exclude ts len) (u16) | key | ts (u64) | value_len (u16) | value | checksum (u32) |
```

LsmStorageInner::get

Previously, we implement `get` as first probe the memtables and then scan the SSTs. Now that we change the memtable to use the new key-ts APIs, we will need to re-implement the `get` interface.

The easiest way to do this is to create a merge iterator over everything we have -- memtables, immutable memtables, L0 SSTs, and other level SSTs, the same as what you have done in `scan`, except that we do a bloom filter filtering over the SSTs.

LsmStorageInner::scan

You will need to incorporate the new memtable APIs, and you should set the scan range to be `(user_key_begin, TS_RANGE_BEGIN)` and `(user_key_end, TS_RANGE_END)`. Note that when you handle the exclude boundary, you will need to correctly position the iterator to the next key (instead of the current key of the same timestamp).

Task 2: Write Path

In this task, you will need to modify:

```
src/lsm_storage.rs
```

We have an `mvcc` field in `LsmStorageInner` that includes all data structures we need to use for multi-version concurrency control in this week. When you open a directory and initialize the storage engine, you will need to create that structure.

In your `write_batch` implementation, you will need to obtain a commit timestamp for all keys in a write batch. You can get the timestamp by using `self.mvcc().latest_commit_ts() + 1` at the beginning of the logic, and `self.mvcc().update_commit_ts(ts)` at the end of the logic to increment the next commit timestamp. To ensure all write batches have different timestamps and new keys are placed on top of old keys, you will need to hold a write lock `self.mvcc().write_lock.lock()` at the beginning of the function, so that only one thread can write to the storage engine at the same time.

Task 3: MVCC Compaction

In this task, you will need to modify:

```
src/compact.rs
```

What we had done in previous chapters is to only keep the latest version of a key and remove a key when we compact the key to the bottom level if the key is removed. With MVCC, we now have timestamps associated with the keys, and we cannot use the same logic for compaction.

In this chapter, you may simply remove the logic to remove the keys. You may ignore `compact_to_bottom_level` for now, and you should keep ALL versions of a key during the compaction.

Also, you will need to implement the compaction algorithm in a way that the same key with different timestamps are put in the same SST file, *even if* it exceeds the SST size limit. This ensures that if a key is found in an SST in a level, it will not be in other SST files in that level, and therefore simplifying the implementation of many parts of the system.

Task 4: LSM Iterator

In this task, you will need to modify:

```
src/lsm_iterator.rs
```

In the previous chapter, we implemented the LSM iterator to act as viewing the same key with different timestamps as different keys. Now, we will need to refactor the LSM iterator to only return the latest version of a key if multiple versions of the keys are retrieved from the child iterator.

You will need to record `prev_key` in the iterator. If we already returned the latest version of a key to the user, we can skip all old versions and proceed to the next key.

At this point, you should pass all tests in previous chapters except persistence tests (2.5 and 2.6).

Test Your Understanding

- What is the difference of `get` in the MVCC engine and the engine you built in week 2?
- In week 2, you stop at the first memtable/level where a key is found when `get`. Can you do the same in the MVCC version?
- How do you convert `keySlice` to `&KeyBytes`? Is it a safe/sound operation?
- Why do we need to take a write lock in the write path?

We do not provide reference answers to the questions, and feel free to discuss about them in the Discord community.

Bonus Tasks

- **Early Stop for Memtable Gets.** Instead of creating a merge iterator over all memtables and SSTs, we can implement `get` as follows: If we find a version of a key in the memtable, we can stop searching. The same applies to SSTs.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Snapshot Read - Engine Read Path and Transaction API

In this chapter, you will:

- Finish the read path based on previous chapter to support snapshot read.
- Implement the transaction API to support snapshot read.
- Implement the engine recovery process to correctly recover the commit timestamp.

At the end of the day, your engine will be able to give the user a consistent view of the storage key space.

During the refactor, you might need to change the signature of some functions from `&self` to `self: &Arc<Self>` as necessary.

To run test cases,

```
cargo x copy-test --week 3 --day 3
cargo x scheck
```

Note: You will also need to pass test cases for 2.5 and 2.6 after finishing this chapter.

Task 1: LSM Iterator with Read Timestamp

The goal of this chapter is to have something like:

```

let snapshot1 = engine.new_txn();
// write something to the engine
let snapshot2 = engine.new_txn();
// write something to the engine
snapshot1.get(/* ... */); // we can retrieve a consistent snapshot of a previous state
of the engine

```

To achieve this, we can record the read timestamp (which is the latest committed timestamp) when creating the transaction. When we do a read operation over the transaction, we will only read all versions of the keys below or equal to the read timestamp.

In this task, you will need to modify:

```
src/lsm_iterator.rs
```

To do this, you will need to record a read timestamp in `LsmIterator`.

```

impl LsmIterator {
    pub(crate) fn new(
        iter: LsmIteratorInner,
        end_bound: Bound<Bytes>,
        read_ts: u64,
    ) -> Result<Self> {
        // ...
    }
}

```

And you will need to change your LSM iterator `next` logic to find the correct key.

Task 2: Multi-Version Scan and Get

In this task, you will need to modify:

```
src/mvcc.rs  
src/mvcc/txn.rs  
src/lsm_storage.rs
```

Now that we have `read_ts` in the LSM iterator, we can implement `scan` and `get` on the transaction structure, so that we can read data at a given point in the storage engine.

We recommend you to create helper functions like `scan_with_ts(/* original parameters */, read_ts: u64)` and `get_with_ts` if necessary in your `LsmStorageInner` structure. The original `get/scan` on the storage engine should be implemented as creating a transaction (snapshot) and do a `get/scan` over that transaction. The call path would be like:

```
LsmStorageInner::scan -> new_txn and Transaction::scan ->  
LsmStorageInner::scan_with_ts
```

To create a transaction in `LsmStorageInner::scan`, we will need to provide a `Arc<LsmStorageInner>` to the transaction constructor. Therefore, we can change the signature of `scan` to take `self: &Arc<Self>` instead of simply `&self`, so that we can create a transaction with `let txn = self.mvcc().new_txn(self.clone(), /* ... */)`.

You will also need to change your `scan` function to return a `TxnIterator`. We must ensure the snapshot is live when the user iterates the engine, and therefore, `TxnIterator` stores the snapshot object. Inside `TxnIterator`, we can store a `FusedIterator<LsmIterator>` for now. We will change it to something else later when we implement OCC.

You do not need to implement `Transaction::put/delete` for now, and all modifications will still go through the engine.

Task 3: Store Largest Timestamp in SST

In this task, you will need to modify:

```
src/table.rs  
src/table/builder.rs
```

In your SST encoding, you should store the largest timestamp after the block metadata, and recover it when loading the SST. This would help the system decide the latest commit timestamp when recovering the system.

Task 4: Recover Commit Timestamp

Now that we have largest timestamp information in the SSTs and timestamp information in the WAL, we can obtain the largest timestamp committed before the engine starts, and use that timestamp as the latest committed timestamp when creating the `mvcc` object.

If WAL is not enabled, you can simply compute the latest committed timestamp by finding the largest timestamp among SSTs. If WAL is enabled, you should further iterate all recovered memtables and find the largest timestamp.

In this task, you will need to modify:

```
src/lsm_storage.rs
```

We do not have test cases for this section. You should pass all persistence tests from previous chapters (including 2.5 and 2.6) after finishing this section.

Test Your Understanding

- So far, we have assumed that our SST files use a monotonically increasing id as the file name. Is it okay to use `<level>_<begin_key>_<end_key>_<max_ts>.sst` as the SST file name? What might be the potential problems with that?
- Consider an alternative implementation of transaction/snapshot. In our implementation, we have `read_ts` in our iterators and transaction context, so that the user can always access a consistent view of one version of the database based on the timestamp. Is it viable to store the current LSM state directly in the transaction context in order to gain a consistent snapshot? (i.e., all SST ids, their level information, and all memtables + ts) What are the pros/cons with that? What if the engine does not have memtables? What if the engine is running on a distributed storage system like S3 object store?
- Consider that you are implementing a backup utility of the MVCC Mini-LSM engine. Is it enough to simply copy all SST files out without backing up the LSM state? Why or why not?

We do not provide reference answers to the questions, and feel free to discuss about them in the Discord community.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Watermark and Garbage Collection

In this chapter, you will implement necessary structures to track the lowest read timestamp being used by the user, and collect unused versions from SSTs when doing the compaction.

To run test cases,

```
cargo x copy-test --week 3 --day 4  
cargo x scheck
```

Task 1: Implement Watermark

In this task, you will need to modify:

```
src/mvcc/watermark.rs
```

Watermark is the structure to track the lowest `read_ts` in the system. When a new transaction is created, it should call `add_reader` to add its read timestamp for tracking. When a transaction aborts or commits, it should remove itself from the watermark. The watermark structures returns the lowest `read_ts` in the system when `watermark()` is called. If there are no ongoing transactions, it simply returns `None`.

You may implement watermark using a `BTreeMap`. It maintains a counter that how many snapshots are using this read timestamp for each `read_ts`. You should not have entries with 0 readers in the b-tree map.

Task 2: Maintain Watermark in Transactions

In this task, you will need to modify:

```
src/mvcc/txn.rs  
src/mvcc.rs
```

You will need to add the `read_ts` to the watermark when a transaction starts, and remove it when `drop` is called for the transaction.

Task 3: Garbage Collection in Compaction

In this task, you will need to modify:

```
src/compact.rs
```

Now that we have a watermark for the system, we can clean up unused versions during the compaction process.

- If a version of a key is above watermark, keep it.
- For all versions of a key below or equal to the watermark, keep the latest version.

For example, if we have watermark=3 and the following data:

```
a@4=del <- above watermark
a@3=3    <- latest version below or equal to watermark
a@2=2    <- can be removed, no one will read it
a@1=1    <- can be removed, no one will read it
b@1=1    <- latest version below or equal to watermark
c@4=4    <- above watermark
d@3=del  <- can be removed if compacting to bottom-most level
d@2=2    <- can be removed
```

If we do a compaction over these keys, we will get:

```
a@4=del
a@3=3
b@1=1
c@4=4
d@3=del (can be removed if compacting to bottom-most level)
```

Assume these are all keys in the engine. If we do a scan at $ts=3$, we will get $a=3, b=1, c=4$ before/after compaction. If we do a scan at $ts=4$, we will get $b=1, c=4$ before/after compaction. Compaction *will not* and *should not* affect transactions with read timestamp \geq watermark.

Test Your Understanding

- In our implementation, we manage watermarks by ourselves with the lifecycle of `Transaction` (so-called un-managed mode). If the user intends to manage key timestamps and the watermarks by themselves (i.e., when they have their own timestamp generator), what do you need to do in the `write_batch/get/scan` API to validate their requests? Is there any architectural assumption we had that might be hard to maintain in this case?
- Why do we need to store an `Arc` of `Transaction` inside a transaction iterator?
- What is the condition to fully remove a key from the SST file?

- For now, we only remove a key when compacting to the bottom-most level. Is there any other prior time that we can remove the key? (Hint: you know the start/end key of each SST in all levels.)
- Consider the case that the user creates a long-running transaction and we could not garbage collect anything. The user keeps updating a single key. Eventually, there could be a key with thousands of versions in a single SST file. How would it affect performance, and how would you deal with it?

Bonus Tasks

- **$O(1)$ Watermark.** You may implement an amortized $O(1)$ watermark structure by using a hash map or a cyclic queue.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Transaction and Optimistic Concurrency Control

In this chapter, you will implement all interfaces of `Transaction`. Your implementation will maintain a private workspace for modifications inside a transaction, and commit them in batch, so that all modifications within the transaction will only be visible to the transaction itself until commit. We only check for conflicts (i.e., serializable conflicts) when commit, and this is optimistic concurrency control.

To run test cases,

```
cargo x copy-test --week 3 --day 5
cargo x scheck
```

Task 1: Local Workspace + Put and Delete

In this task, you will need to modify:

```
src/mvcc/txn.rs
```

You can now implement `put` and `delete` by inserting the corresponding key/value to the `local_storage`, which is a skiplist memtable without key timestamp. Note that for deletes, you will still need to implement it as inserting an empty value, instead of removing a value from the skiplist.

Task 2: Get and Scan

In this task, you will need to modify:

```
src/mvcc/txn.rs
```

For `get`, you should first probe the local storage. If a value is found, return the value or `None` depending on whether it is a deletion marker. For `scan`, you will need to implement a `TxnLocalIterator` for the skiplist as in chapter 1.1 when you implement the iterator for a memtable without key timestamp. You will need to store a `TwoMergeIterator<TxnLocalIterator, FusedIterator<LsmIterator>>` in the `TxnIterator`. And, lastly, given that the `TwoMergeIterator` will retain the deletion markers in the child iterators, you will need to modify your `TxnIterator` implementation to correctly handle deletions.

Task 3: Commit

In this task, you will need to modify:

```
src/mvcc/txn.rs
```

We assume that a transaction will only be used on a single thread. Once your transaction enters the commit phase, you should set `self.committed` to true, so that users cannot do any other operations on the transaction. Your `put`, `delete`, `scan`, and `get` implementation should error if the transaction is already committed.

Your commit implementation should simply collect all key-value pairs from the local storage and submit a write batch to the storage engine.

Task 4: Atomic WAL

In this task, you will need to modify:

```
src/wal.rs
src/mem_table.rs
src/lsm_storage.rs
```

Note that `commit` involves producing a write batch, and for now, the write batch does not guarantee atomicity. You will need to change the WAL implementation to produce a header and a footer for the write batch.

The new WAL encoding is as follows:

	HEADER		BODY												
FOOTER															
	u32		u16		var		u64		u16		var		...		
u32															
	batch_size		key_len		key		ts		value_len		value		more key-value pairs ...		
checksum															

`batch_size` is the size of the `BODY` section. `checksum` is the checksum for the `BODY` section.

There are no test cases to verify your implementation. As long as you pass all existing test cases and implement the above WAL format, everything should be fine.

You should implement `Wal::put_batch` and `MemTable::put_batch`. The original `put` function should treat the single key-value pair as a batch. That is to say, at this point, your `put` function should call `put_batch`.

A batch should be handled in the same mem table and the same WAL, even if it exceeds the mem table size limit.

Test Your Understanding

- With all the things we have implemented up to this point, does the system satisfy snapshot isolation? If not, what else do we need to do to support snapshot isolation? (Note: snapshot isolation is different from serializable snapshot isolation we will talk about in the next chapter)
- What if the user wants to batch import data (i.e., 1TB?) If they use the transaction API to do that, will you give them some advice? Is there any opportunity to optimize for this case?
- What is optimistic concurrency control? What would the system be like if we implement pessimistic concurrency control instead in Mini-LSM?
- What happens if your system crashes and leave a corrupted WAL on the disk? How do you handle this situation?
- When you commit the txn, is it necessary to put everything into the memtable in batch, or you can simply put it key by key? Why?

Bonus Tasks

- **Spill to Disk.** If the private workspace of a transaction gets too large, you may flush some of the data to the disk.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

(A Partial) Serializable Snapshot Isolation

Now, we are going to add a conflict detection algorithm at the transaction commit time, so as to make the engine to have some level of serializable.

To run test cases,

```
cargo x copy-test --week 3 --day 6
cargo x scheck
```

Let us go through an example of serializable. Consider that we have two transactions in the engine that:

```
txn1: put("key1", get("key2"))
txn2: put("key2", get("key1"))
```

The initial state of the database is `key1=1, key2=2`. Serializable means that the outcome of the execution has the same result of executing the transactions one by one in serial in some order. If we execute `txn1` then `txn2`, we will get `key1=2, key2=2`. If we execute `txn2` then `txn1`, we will get `key1=1, key2=1`.

However, with our current implementation, if the execution of these two transactions overlaps:

```
txn1: get key2 <- 2
txn2: get key1 <- 1
txn1: put key1=2, commit
txn2: put key2=1, commit
```

We will get `key1=2, key2=1`. This cannot be produced with a serial execution of these two transactions. This phenomenon is called write skew.

With serializable validation, we can ensure the modifications to the database corresponds to a serial execution order, and therefore, users may run some critical workloads over the system that requires serializable execution. For example, if a user runs bank transfer workloads on Mini-LSM, they would expect the sum of money at any point of time is the same. We cannot guarantee this invariant without serializable checks.

One technique of serializable validation is to record read set and write set of each transaction in the system. We do the validation before committing a transaction (optimistic concurrency control). If the read set of the transaction overlaps with any transaction committed after its read timestamp, then we fail the validation, and abort the transaction.

Back to the above example, if we have txn1 and txn2 both started at timestamp = 1.

```
txn1: get key2 <- 2
txn2: get key1 <- 1
txn1: put key1=2, commit ts = 2
txn2: put key2=1, start serializable verification
```

When we validate txn2, we will go through all transactions started before the expected commit timestamp of itself and after its read timestamp (in this case, $1 < ts < 3$). The only transaction satisfying the criteria is txn1. The write set of txn1 is `key1`, and the read set of txn2 is `key1`. As they overlap, we should abort txn2.

Task 1: Track Read Set in Get and Write Set

In this task, you will need to modify:

```
src/mvcc/txn.rs
src/mvcc.rs
```

When `get` is called, you should add the key to the read set of the transaction. In our implementation, we store the hashes of the keys, so as to reduce memory usage and make probing the read set faster, though this might cause false positives when two keys have the same hash. You can use `farmhash::hash32` to generate the hash for a key. Note that even if `get` returns a key is not found, this key should still be tracked in the read set.

In `LsmMvccInner::new_txn`, you should create an empty read/write set for the transaction if `serializable=true`.

Task 2: Track Read Set in Scan

In this task, you will need to modify:

```
src/mvcc/txn.rs
```

In this course, we only guarantee full serializability for `get` requests. You still need to track the read set for scans, but in some specific cases, you might still get non-serializable result.

To understand why this is hard, let us go through the following example.

```
txn1: put("key1", len(scan(..)))  
txn2: put("key2", len(scan(..)))
```

If the database starts with an initial state of `a=1,b=2`, we should get either `a=1,b=2,key1=2,key2=3` or `a=1,b=2,key1=3,key2=2`. However, if the transaction execution is as follows:


```
txn1: len(scan(..)) = 2
txn2: len(scan(..)) = 2
txn1: put key1 = 2, commit, read set = {a, b}, write set = {key1}
txn2: put key2 = 2, commit, read set = {a, b}, write set = {key2}
```

This passes our serializable validation and does not correspond to any serial order of execution! Therefore, a fully-working serializable validation will need to track key ranges, and using key hashes can accelerate the serializable check if only `get` is called. Please refer to the bonus tasks on how you can implement serializable checks correctly.

Task 3: Engine Interface and Serializable Validation

In this task, you will need to modify:

```
src/mvcc/txn.rs
src/lsm_storage.rs
```

Now, we can go ahead and implement the validation in the commit phase. You should take the `commit_lock` every time we process a transaction commit. This ensures only one transaction goes into the transaction verification and commit phase.

You will need to go through all transactions with commit timestamp within range (`read_ts`, `expected_commit_ts`) (both excluded bounds), and see if the read set of the current transaction overlaps with the write set of any transaction satisfying the criteria. If we can commit the transaction, submit a write batch, and insert the write set of this transaction into `self.inner.mvcc().committed_txns`, where the key is the commit timestamp.

You can skip the check if `write_set` is empty. A read-only transaction can always be committed.

You should also modify the `put`, `delete`, and `write_batch` interface in `LsmStorageInner`. We recommend you define a helper function `write_batch_inner` that processes a write batch. If `options.serializable = true`, `put`, `delete`, and the user-facing `write_batch` should create a transaction instead of directly creating a write batch. Your write batch helper function should also return a `u64` commit timestamp so that `Transaction::Commit` can correctly store the committed transaction data into the MVCC structure.

Task 4: Garbage Collection

In this task, you will need to modify:

```
src/mvcc/txn.rs
```

When you commit a transaction, you can also clean up the committed txn map to remove all transactions below the watermark, as they will not be involved in any future serializable validations.

Test Your Understanding

- If you have some experience with building a relational database, you may think about the following question: assume that we build a database based on Mini-LSM where we store each row in the relation table as a key-value pair (key: primary key, value: serialized row) and enable serializable verification, does the database system directly gain ANSI serializable isolation level capability? Why or why not?
- The thing we implement here is actually write snapshot-isolation (see [A critique of snapshot isolation](#)) that guarantees serializable. Is there any cases where the execution is serializable, but will be rejected by the write snapshot-isolation validation?

- There are databases that claim they have serializable snapshot isolation support by only tracking the keys accessed in gets and scans (instead of key range). Do they really prevent write skew caused by phantoms? (Okay... Actually, I'm talking about [BadgerDB](#).)

We do not provide reference answers to the questions, and feel free to discuss about them in the Discord community.

Bonus Tasks

- **Read-Only Transactions.** With serializable enabled, we will need to keep track of the read set for a transaction.
- **Precision/Predicate Locking.** The read set can be maintained using a range instead of a single key. This would be useful when a user scans the full key space. This will also enable serializable verification for scan.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Snack Time: Compaction Filters

Congratulations! You made it there! In the previous chapter, you made your LSM engine multi-version capable, and the users can use transaction APIs to interact with your storage engine. At the end of this week, we will implement some easy but important features of the storage engine. Welcome to Mini-LSM's week 3 snack time!

In this chapter, we will generalize our compaction garbage collection logic to become compaction filters.

For now, our compaction will simply retain the keys above the watermark and the latest version of the keys below the watermark. We can add some magic to the compaction process to help the user collect some unused data automatically as a background job.

Consider a case that the user uses Mini-LSM to store database tables. Each row in the table are prefixed with the table name. For example,

```
table1_key1 -> row
table1_key2 -> row
table1_key3 -> row
table2_key1 -> row
table2_key2 -> row
```

Now the user executes `DROP TABLE table1`. The engine will need to clean up all the data beginning with `table1`.

There are a lot of ways to achieve the goal. The user of Mini-LSM can scan all the keys beginning with `table1` and requests the engine to delete it. However, scanning a very large database might be slow, and it will generate the same number of delete tombstones as the existing keys. Therefore, scan-and-delete will not free up the space occupied by the dropped table -- instead, it will add more

data to the engine and the space can only be reclaimed when the tombstones reach the bottom level of the engine.

Or, they can create column families (we will talk about this in *rest of your life* chapter). They store each table in a column family, which is a standalone LSM state, and directly remove the SST files corresponding to the column family when the user drop the table.

In this course, we will implement the third approach: compaction filters. Compaction filters can be dynamically added to the engine at runtime. During the compaction, if a key matching the compaction filter is found, we can silently remove it in the background. Therefore, the user can attach a compaction filter of `prefix=table1` to the engine, and all these keys will be removed during compaction.

Task 1: Compaction Filter

In this task, you will need to modify:

```
src/compact.rs
```

You can iterate all compaction filters in `LsmStorageInner::compaction_filters`. If the first version of the key below watermark matches the compaction filter, simply remove it instead of keeping it in the SST file.

To run test cases,

```
cargo x copy-test --week 3 --day 7  
cargo x scheck
```

You can assume that the user will not get the keys within the prefix filter range. And, they will not scan the keys in the prefix range. Therefore, it is okay to return a wrong value when a user requests

the keys in the prefix filter range (i.e., undefined behavior).

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

The Rest of Your Life (TBD)

This is an advanced part that deep dives into optimizations and applications of the LSM storage engine and will make your implementation more production-ready. We are still planning on the content, and this part will not be publicly available in near future.

Week + Chapter	Topic	Solution	Starter Code	Writeup
4.1	Benchmarking			
4.2	Block Compression			
4.3	Trivial Move and Parallel Compaction			
4.4	Alternative Block Encodings			
4.5	Rate Limiter and I/O Optimizations			
4.6	Build Your Own Block Cache			
4.7	Build Your Own SkipList			
4.8	Async Engine			
4.9	IO-uring-based I/O engine			
4.10	Prefetching			
4.11	Key-Value Separation			
4.12	Column Families			
4.13	Sharding			
4.14	Compaction Optimizations			
4.15	SQL over Mini-LSM			

Mini-LSM v1

This is a legacy version of the Mini-LSM course and we will not maintain it anymore. We now have a new version of this course. We keep the legacy version in this book so that the search engine can keep the pages in the index and users can follow the links to the new version of the course.

V1 Course Overview



In this course, we will build the LSM tree structure in 7 days:

- Day 1: Block encoding. SSTs are composed of multiple data blocks. We will implement the block encoding.
- Day 2: SST encoding.
- Day 3: MemTable and Merge Iterators.
- Day 4: Block cache and Engine. To reduce disk I/O and maximize performance, we will use moka-rs to build a block cache for the LSM tree. In this day we will get a functional (but not persistent) key-value engine with `get`, `put`, `scan`, `delete` API.
- Day 5: Compaction. Now it's time to maintain a leveled structure for SSTs.
- Day 6: Recovery. We will implement WAL and manifest so that the engine can recover after restart.
- Day 7: Bloom filter and key compression. They are widely-used optimizations in LSM tree structures.

Development Guide

We provide you starter code (see `mini-lsm-starter` crate), where we simply replace all function body with `unimplemented!()`. You can start your project based on this starter code. We provide test cases, but they are very simple. We recommend you to think carefully about your implementation and write test cases by yourself.


- You can use `cargo x scheck` to run all test cases and do style check in your codebase.
- You can use `cargo x copy-test dayX` to copy test cases to the starter code.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Block Builder and Block Iterator

 This is a legacy version of the Mini-LSM course and we will not maintain it anymore. We now have a better version of this course and this chapter is now part of [Mini-LSM Week 1 Day 3: Blocks](#).

In this part, you will need to modify:

- `src/block/builder.rs`
- `src/block/iterator.rs`
- `src/block.rs`

You can use `cargo x copy-test day1` to copy our provided test cases to the starter code directory. After you have finished this part, use `cargo x scheck` to check the style and run all test cases. If you want to write your own test cases, write a new module `#[cfg(test)] mod user_tests { /* your test cases */ }` in `block.rs`. Remember to remove `#![allow(...)]` at the top of the modules you modified so that cargo clippy can actually check the styles.

Task 1 - Block Builder

Block is the minimum read unit in LSM. It is of 4KB size in general, similar to database pages. In each block, we will store a sequence of sorted key-value pairs.

You will need to modify `BlockBuilder` in `src/block/builder.rs` to build the encoded data and the offset array. The block contains two parts: data and offsets.

data	offsets	meta					
-----	-----	-----					
entry	entry	entry	entry	offset	offset	offset	offset
num_of_elements							

When user adds a key-value pair to a block (which is an entry), we will need to serialize it into the following format:

	Entry #1			...

key_len (2B)	key (keylen)	value_len (2B)	value (varlen)	...

Key length and value length are both 2 bytes, which means their maximum lengths are 65535. (Internally stored as `u16`)

We assume that keys will never be empty, and values can be empty. An empty value means that the corresponding key has been deleted in the view of other parts of the system. For the `BlockBuilder` and `BlockIterator` , we just treat the empty value as-is.

At the end of each block, we will store the offsets of each entry and the total number of entries. For example, if the first entry is at 0th position of the block, and the second entry is at 12th position of the block.

offset	offset	num_of_elements	

0	12	2	

The footer of the block will be as above. Each of the number is stored as `u16`.

The block has a size limit, which is `target_size`. Unless the first key-value pair exceeds the target block size, you should ensure that the encoded block size is always less than or equal to `target_size`. (In the provided code, the `target_size` here is essentially the `block_size`)

The `BlockBuilder` will produce the data part and unencoded entry offsets when `build` is called. The information will be stored in the `Block` struct. As key-value entries are stored in raw format and offsets are stored in a separate vector, this reduces unnecessary memory allocations and processing overhead when decoding data — what you need to do is to simply copy the raw block data to the `data` vector and decode the entry offsets every 2 bytes, *instead of* creating something like `Vec<(Vec<u8>, Vec<u8>)>` to store all the key-value pairs in one block in memory. This compact memory layout is very efficient.

For the encoding and decoding part, you'll need to modify `Block` in `src/block.rs`. Specifically, you are required to implement `Block::encode` and `Block::decode`, which will encode to / decode from the data layout illustrated in the above figures.

Task 2 - Block Iterator

Given a `Block` object, we will need to extract the key-value pairs. To do this, we create an iterator over a block and find the information we want.

`BlockIterator` can be created with an `Arc<Block>`. If `create_and_seek_to_first` is called, it will be positioned at the first key in the block. If `create_and_seek_to_key` is called, the iterator will be positioned at the first key that is `>=` the provided key. For example, if `1, 3, 5` is in a block.

```
let mut iter = BlockIterator::create_and_seek_to_key(block, b"2");
assert_eq!(iter.key(), b"3");
```

The above `seek 2` will make the iterator to be positioned at the next available key of `2`, which in this case is `3`.

The iterator should copy `key` and `value` from the block and store them inside the iterator, so that users can access the key and the value without any extra copy with `fn key(&self) -> &[u8]`, which directly returns the reference of the locally-stored key and value.

When `next` is called, the iterator will move to the next position. If we reach the end of the block, we can set `key` to empty and return `false` from `is_valid`, so that the caller can switch to another block if possible.

After implementing this part, you should be able to pass all tests in `block/tests.rs`.

Extra Tasks

Here is a list of extra tasks you can do to make the block encoding more robust and efficient.

Note: Some test cases might not pass after implementing this part. You might need to write your own test cases.


- Implement block checksum. Verify checksum when decoding the block.
- Compress / Decompress block. Compress on `build` and decompress on decoding.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

SST Builder and SST Iterator

 This is a legacy version of the Mini-LSM course and we will not maintain it anymore. We now have a better version of this course and this chapter is now part of [Mini-LSM Week 1 Day 4: Sorted String Table \(SST\)](#).

In this part, you will need to modify:

- `src/table/builder.rs`
- `src/table/iterator.rs`
- `src/table.rs`

You can use `cargo x copy-test day2` to copy our provided test cases to the starter code directory. After you have finished this part, use `cargo x scheck` to check the style and run all test cases. If you want to write your own test cases, write a new module `#[cfg(test)] mod user_tests { /* your test cases */ }` in `table.rs`. Remember to remove `#![allow(...)]` at the top of the modules you modified so that cargo clippy can actually check the styles.

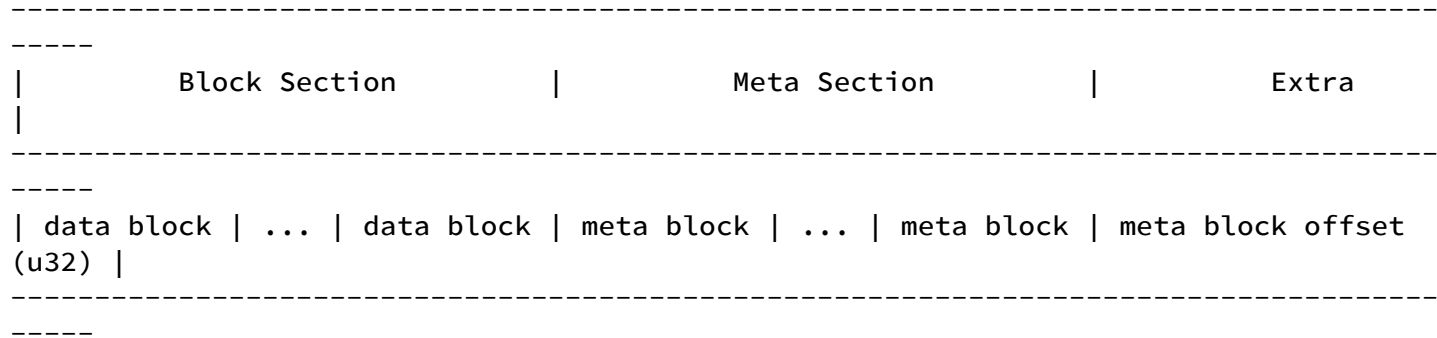
Task 1 - SST Builder

SST is composed of data blocks and index blocks stored on the disk. Usually, data blocks are lazily loaded -- they will not be loaded into the memory until a user requests it. Index blocks can also be loaded on-demand, but in this course, we make simple assumptions that all SST index blocks (meta blocks) can fit in memory. Generally, an SST file is of 256MB size.

The SST builder is similar to block builder -- users will call `add` on the builder. You should maintain a `BlockBuilder` inside SST builder and split block when necessary. Also, you will need to maintain block metadata `BlockMeta`, which includes the first key in each block and the offset of each block.

The `build` function will encode the SST, write everything to disk using `FileObject::create`, and return an `SsTable` object. Note that in part 2, you don't need to actually write the data to the disk. Just store everything in memory as a vector until we implement a block cache (Day 4, Task 5).

The encoding of SST is like:



You also need to implement `estimated_size` function of `SsTableBuilder`, so that the caller can know when can it start a new SST to write data. The function don't need to be very accurate. Given the assumption that data blocks contain much more data than meta block, we can simply return the size of data blocks for `estimated_size`.

You can also align blocks to 4KB boundary so as to make it possible to do direct I/O in the future. This is an optional optimization.

The recommend sequence to finish **Task 1** is as below:

- Implement `SsTableBuilder` in `src/table/builder.rs`
 - Before implementing `SsTableBuilder`, you may want to take a look in `src/table.rs`, for `FileObject` & `BlockMeta`.
 - For `FileObject`, you should at least implement `read`, `size` and `create` (No need for Disk I/O) before day 4.

- For `BlockMeta`, you may want to add some extra fields when encoding / decoding the `BlockMeta` to / from a buffer.
- Implement `SsTable` in `src/table.rs`
 - Same as above, you do not need to worry about `BlockCache` until day 4.

After finishing **Task 1**, you should be able to pass all the current tests except two iterator tests.

Task 2 - SST Iterator

Like `BlockIterator`, you will need to implement an iterator over an SST. Note that you should load data on demand. For example, if your iterator is at block 1, it should not hold any other block content in memory until it reaches the next block.

`SsTableIterator` should implement the `StorageIterator` trait, so that it can be composed with other iterators in the future.

One thing to note is `seek_to_key` function. Basically, you will need to do binary search on block metadata to find which block might possibly contain the key. It is possible that the key doesn't exist in the LSM tree so that the block iterator will be invalid immediately after a seek. For example,

```

-----
| block 1 | block 2 | block meta |
-----
| a, b, c | e, f, g | 1: a, 2: e |
-----

```

If we do `seek(b)` in this SST, it is quite simple -- using binary search, we can know block 1 contains keys `a <= keys < e`. Therefore, we load block 1 and seek the block iterator to the corresponding position.

But if we do `seek(d)`, we will position to block 1, but seeking `d` in block 1 will reach the end of the block. Therefore, we should check if the iterator is invalid after the seek, and switch to the next block if necessary.

Extra Tasks

Here is a list of extra tasks you can do to make the block encoding more robust and efficient.

Note: Some test cases might not pass after implementing this part. You might need to write your own test cases.

- Implement index checksum. Verify checksum when decoding.
- Explore different SST encoding and layout. For example, in the [Lethe](#) paper, the author adds secondary key support to SST.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Mem Table and Merge Iterators



This is a legacy version of the Mini-LSM course and we will not maintain it anymore. We now have a better version of this course and this chapter is now part of [Mini-LSM Week 1 Day 1: Memtable](#) and [Mini-LSM Week 1 Day 2: Merge Iterator](#)

In this part, you will need to modify:

- `src/iterators/merge_iterator.rs`
- `src/iterators/two_merge_iterator.rs`
- `src/mem_table.rs`

You can use `cargo x copy-test day3` to copy our provided test cases to the starter code directory. After you have finished this part, use `cargo x scheck` to check the style and run all test cases. If you want to write your own test cases, write a new module `#[cfg(test)] mod user_tests { /* your test cases */ }` in `table.rs`. Remember to remove `#![allow(...)]` at the top of the modules you modified so that cargo clippy can actually check the styles.

This is the last part for the basic building blocks of an LSM tree. After implementing the merge iterators, we can easily merge data from different part of the data structure (mem table + SST) and get an iterator over all data. And in part 4, we will compose all these things together to make a real storage engine.

Task 1 - Mem Table

In this course, we use [crossbeam-skiplist](#) as the implementation of memtable. Skiplist is like linked-list, where data is stored in a list node and will not be moved in memory. Instead of using a single

pointer for the next element, the nodes in skiplists contain multiple pointers and allow user to "skip some elements", so that we can achieve $O(\log n)$ search, insertion, and deletion.

In storage engine, users will create iterators over the data structure. Generally, once user modifies the data structure, the iterator will become invalid (which is the case for C++ STL and Rust containers). However, skiplists allow us to access and modify the data structure at the same time, therefore potentially improving the performance when there is concurrent access. There are some papers argue that skiplists are bad, but the good property that data stays in its place in memory can make the implementation easier for us.

In `mem_table.rs`, you will need to implement a mem-table based on crossbeam-skiplist. Note that the memtable only supports `get`, `scan`, and `put` without `delete`. The deletion is represented as a tombstone `key -> empty value`, and the actual data will be deleted during the compaction process (day 5). Note that all `get`, `scan`, `put` functions only need `&self`, which means that we can concurrently call these operations.

Task 2 - Mem Table Iterator

You can now implement an iterator `MemTableIterator` for `MemTable`. `memtable.iter(start, end)` will create an iterator that returns all elements within the range `start`, `end`. Here, `start` is `std::ops::Bound`, which contains 3 variants: `Unbounded`, `Included(key)`, `Excluded(key)`. The expresiveness of `std::ops::Bound` eliminates the need to memorizing whether an API has a closed range or open range.

Note that `crossbeam-skiplist`'s iterator has the same lifetime as the skiplist itself, which means that we will always need to provide a lifetime when using the iterator. This is very hard to use. You can use the `ouroboros` crate to create a self-referential struct that erases the lifetime. You will find the [ouroboros examples](#) helpful.

```
pub struct MemTableIterator {
    /// hold the reference to the skiplist so that the iterator will be valid.
    map: Arc<Skiplist>
    /// then the lifetime of the iterator should be the same as the `MemTableIterator`
    struct itself
    iter: Skiplist::Iter<'this>
}
```

You will also need to convert the Rust-style iterator API to our storage iterator. In Rust, we use `next() -> Data`. But in this course, `next` doesn't have a return value, and the data should be fetched by `key()` and `value()`. You will need to think a way to implement this.

► Spoiler: the MemTableIterator struct

In this design, you might have noticed that as long as we have the iterator object, the mem-table cannot be freed from the memory. In this course, we assume user operations are short, so that this will not cause big problems. See extra task for possible improvements.

You can also consider using [AgateDB's skiplist](#) implementation, which avoids the problem of creating a self-referential struct.

Task 3 - Merge Iterator

Now that you have a lot of mem-tables and SSTs, you might want to merge them to get the latest occurrence of a key. In `merge_iterator.rs`, we have `MergeIterator`, which is an iterator that merges all iterators *of the same type*. The iterator at the lower index position of the `new` function has higher priority, that is to say, if we have:

```
iter1: 1->a, 2->b, 3->c
iter2: 1->d
iter: MergeIterator::create(vec![iter1, iter2])
```

The final iterator will produce 1->a, 2->b, 3->c . The data in iter1 will overwrite the data in other iterators.

You can use a `BinaryHeap` to implement this merge iterator. Note that you should never put any invalid iterator inside the binary heap. One common pitfall is on error handling. For example,

```
let Some(mut inner_iter) = self.iters.peek_mut() {  
    inner_iter.next()?; // <- will cause problem  
}
```

If `next` returns an error (i.e., due to disk failure, network failure, checksum error, etc.), it is no longer valid. However, when we go out of the if condition and return the error to the caller, `PeekMut`'s drop will try move the element within the heap, which causes an access to an invalid iterator. Therefore, you will need to do all error handling by yourself instead of using `?` within the scope of `PeekMut` .

You will also need to define a wrapper for the storage iterator so that `BinaryHeap` can compare across all iterators.

Task 4 - Two Merge Iterator

The LSM has two structures for storing data: the mem-tables in memory, and the SSTs on disk. After we constructed the iterator for all SSTs and all mem-tables respectively, we will need a new iterator to merge iterators of two different types. That is `TwoMergeIterator` .

You can implement `TwoMergeIterator` in `two_merge_iter.rs` . Similar to `MergeIterator` , if the same key is found in both of the iterator, the first iterator takes precedence.

In this course, we explicitly did not use something like `Box<dyn StorageIter>` to avoid dynamic dispatch. This is a common optimization in LSM storage engines.

Extra Tasks


- Implement different mem-table and see how it differs from skiplist. i.e., BTree mem-table. You will notice that it is hard to get an iterator over the B+ tree without holding a lock of the same timespan as the iterator. You might need to think of smart ways of solving this.
- Async iterator. One interesting thing to explore is to see if it is possible to asynchronize everything in the storage engine. You might find some lifetime related problems and need to workaround them.
- Foreground iterator. In this course we assumed that all operations are short, so that we can hold reference to mem-table in the iterator. If an iterator is held by users for a long time, the whole mem-table (which might be 256MB) will stay in the memory even if it has been flushed to disk. To solve this, we can provide a `ForegroundIterator` / `LongIterator` to our user. The iterator will periodically create new underlying storage iterator so as to allow garbage collection of the resources.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Storage Engine and Block Cache

 This is a legacy version of the Mini-LSM course and we will not maintain it anymore. We now have a better version of this course and this chapter is now part of [Mini-LSM Week 1 Day 5: Read Path](#) and [Mini-LSM Week 1 Day 6: Write Path](#)

In this part, you will need to modify:

- `src/lsm_iterator.rs`
- `src/lsm_storage.rs`
- `src/table.rs`
- Other parts that use `SsTable::read_block`

You can use `cargo x copy-test day4` to copy our provided test cases to the starter code directory. After you have finished this part, use `cargo x scheck` to check the style and run all test cases. If you want to write your own test cases, write a new module `#[cfg(test)] mod user_tests { /* your test cases */ }` in `table.rs`. Remember to remove `#![allow(...)]` at the top of the modules you modified so that cargo clippy can actually check the styles.

Task 1 - Put and Delete

Before implementing put and delete, let's revisit how LSM tree works. The structure of LSM includes:

- Mem-table: one active mutable mem-table and multiple immutable mem-tables.
- Write-ahead log: each mem-table corresponds to a WAL.
- SSTs: mem-table can be flushed to the disk in SST format. SSTs are organized in multiple levels.

In this part, we only need to take the lock, write the entry (or tombstone) into the active mem-table. You can modify `lsm_storage.rs`.

Task 2 - Get

To get a value from the LSM, we can simply probe from active memtable, immutable memtables (from latest to earliest), and all the SSTs. To reduce the critical section, we can hold the read lock to copy all the pointers to mem-tables and SSTs out of the `LsmStorageInner` structure, and create iterators out of the critical section. Be careful about the order when creating iterators and probing.

Task 3 - Scan

To create a scan iterator `LsmIterator`, you will need to use `TwoMergeIterator` to merge `MergeIterator` on mem-table and `MergeIterator` on SST. You can implement this in `lsm_iterator.rs`. Optionally, you can implement `FusedIterator` so that if a user accidentally calls `next` after the iterator becomes invalid, the underlying iterator won't panic.

The sequence of key-value pairs produced by `TwoMergeIterator` may contain empty value, which means that the value is deleted. `LsmIterator` should filter these empty values. Also it needs to correctly handle the start and end bounds.

Task 4 - Sync

In this part, we will implement mem-tables and flush to L0 SSTs in `lsm_storage.rs`. As in task 1, write operations go directly into the active mutable mem-table. Once `sync` is called, we flush SSTs to

the disk in two steps:

- Firstly, move the current mutable mem-table to immutable mem-table list, so that no future requests will go into the current mem-table. Create a new mem-table. All of these should happen in one single critical section and stall all reads.
- Then, we can flush the mem-table to disk as an SST file without holding any lock.
- Finally, in one critical section, remove the mem-table and put the SST into `l0_tables`.

Only one thread can sync at a time, and therefore you should use a mutex to ensure this requirement.

Task 5 - Block Cache

Now that we have implemented the LSM structure, we can start writing something to the disk! Previously in `table.rs`, we implemented a `FileObject` struct, without writing anything to disk. In this task, we will change the implementation so that:

- `read` will read from the disk without any caching using `read_exact_at` in `std::os::unix::fs::FileExt`.
- The size of the file should be stored inside the struct, and `size` function directly returns it.
- `create` should write the file to the disk. Generally you should call `fsync` on that file. But this would slow down unit tests a lot. Therefore, we don't do `fsync` until day 6 recovery.
- `open` remains unimplemented until day 6 recovery.

After that, we can implement a new `read_block_cached` function on `SsTable` so that we can leverage block cache to serve read requests. Upon initializing the `LsmStorage` struct, you should create a block cache of 4GB size using `moka-rs`. Blocks are cached by SST id + block id. Use `try_get_with` to get the block from cache / populate the cache if cache miss. If there are multiple

requests reading the same block and cache misses, `try_get_with` will only issue a single read request to the disk and broadcast the result to all requests.

Remember to change `SsTableIterator` to use the block cache.

Extra Tasks

- As you might have seen, each time we do a get, put or deletion, we will need to take a read lock protecting the LSM structure; and if we want to flush, we will need to take a write lock. This can cause a lot of problems. Some lock implementations are fair, which means as long as there is a writer waiting on the lock, no reader can take the lock. Therefore, the writer will wait until the slowest reader finishes its operation before it can actually do some work. One possible optimization is to implement `WriteBatch`. We don't need to immediately write users' requests into mem-table + WAL. We can allow users to do a batch of writes.
- Align blocks to 4K and use direct I/O.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Leveled Compaction



This is a legacy version of the Mini-LSM course and we will not maintain it anymore. We now have a better version of this course and this chapter is now part of:

- [Mini-LSM Week 2 Day 1: Compaction Implementation](#)
- [Mini-LSM Week 2 Day 2: Simple Compaction Strategy](#)
- [Mini-LSM Week 2 Day 3: Tiered Compaction Strategy](#)
- [Mini-LSM Week 2 Day 4: Leveled Compaction Strategy](#)

We did not finish this chapter as part of Mini-LSM v1.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Write-Ahead Log for Recovery



This is a legacy version of the Mini-LSM course and we will not maintain it anymore. We now have a better version of this course and this chapter is now part of:

- [Mini-LSM Week 2 Day 5: Manifest](#)
- [Mini-LSM Week 2 Day 6: Write-Ahead Log \(WAL\)](#)

We did not finish this chapter as part of Mini-LSM v1.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Bloom Filters



This is a legacy version of the Mini-LSM course and we will not maintain it anymore. We now have a better version of this course and this chapter is now part of [Mini LSM Week 1 Day 7: SST Optimizations](#).

We did not finish this chapter as part of Mini-LSM v1.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

Key Compression



This is a legacy version of the Mini-LSM course and we will not maintain it anymore. We now have a better version of this course and this chapter is now part of [Mini LSM Week 1 Day 7: SST Optimizations](#).

We did not finish this chapter as part of Mini-LSM v1.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.

What's Next

We did not finish this chapter as part of Mini-LSM v1.

Your feedback is greatly appreciated. Welcome to join our [Discord Community](#).

Found an issue? Create an issue / pull request on github.com/skyzh/mini-lsm.

mini-lsm-book © 2022-2025 by Alex Chi Z is licensed under CC BY-NC-SA 4.0.