

LSM-DB

项目来源

CS265: Data Systems



<http://daslab.seas.harvard.edu/classes/cs265/project.html>

需求

实现一个基于 lsm tree 的 key-value database，支持 put, delete, get, range 命令。

一个简单的设计

考虑一个实现起来简单、又能体现 lsm tree 核心特性的 lsm db 应该是怎样的。

首先考虑 client server 交互方式。不考虑 RPC 或 plain TCP 等网络连接方式，直接让 server 暴露出一个 command line interface，即 client 直接像在 shell 中输入命令一样，与本地的 server 交互。server 收到 cmd 后，再将 cmd 转发给 db，由 db 进行 handle。

对于一个 key-value 数据库，某些数据库将 key, value 分离存储，有的甚至采取 column family 的方式将 value 分离存储。考虑到实现上的简洁性，本项目选择将 key, value 聚合在一起存储，之后将 key, value 的聚合物称为 key。暂时只考虑定长的 key, value，故 key 中不需要存储 key length 和 value length。lsm tree 是日志写，即不会真正地删除一个 key，而是写入一个新的 key，但是设置一个 marker 或称 tombstone，标识这个 key 被删除了，因此需要使用至少一个 bit 来存储这个 marker。同样是由于日志写，数据库中可能会同时存在同一个 key 的多个不同版本。为了标识不同的版本，需要引入一个 version number 或称 sequence number。对于每一个新的 key，数据库会分配一个新的、独特的、全局递增的 sequence number。总结，key 最少需要包含 key, value, delete marker 和 sequence number。

关于 memtable，不使用 double buffer，即像 leveldb 那样设置一个 active 和一个 immutable memtable，而只使用 single buffer，即 db 只维护一个 memtable。memtable 应该暴露出一些接口给 db，供 cmd handler 调用，例如 put, get, delete, range 等。至于 memtable 的 backing 数据结构，即 in-memory index，考虑使用最简单的 vector。当 memtable 中数据量达到设定的阈值以后，将所有 key 根据指定的比较器进行排序。排序后的 keys 被写入到一个 sstable 文件中。此过程即 minor compaction。当然，如果 memtable 的容量大于 sstable 的容量，一次 minor compaction 也可能生成多个 sstable 文件。

关于 sstable 文件，有很多设计点，提几个比较重要的。一个是 keys 的存储，即按照什么方式去组织 disk 中的数据。可以像 lsm tree 最初的论文那样，使用 B+ tree 的方式存储 keys，也可以像

leveldb 那样直接顺序存储 keys。另一个是，sstable 应该包含什么，如何去组织它所包含的这些东西。通常来说，memory 与 disk 之间是以 block（或称 page，当然还有其它叫法）为单位进行数据的传输。操作系统用 paging 的方式减小 memory internal fragmentation，且较小的 page 可以给 memory management 提供更高的自由度。对于数据库而言，以 block 为单位进行传输主要是为了分摊 disk io 的 overhead。因此，首先要明确的是，sstable 是以 block 为单位组织数据。

那么除了 keys 之外，sstable 还应该包含什么呢？经过多次 merge 之后，高层级的 sstables 可能会很大，则进行顺序查找会带来很大的 disk io overhead。故除了 keys 本身，sstable 通常会存储额外的 sparse index。sparse 意味着这个 index 是针对 block 而言的。根据 keys 在 sstable 中的组织方式的不同，sparse index 可以是简单的 fence pointers，即每个 block 中的最大 key 或最小 key 作为一个 fence pointer，所有的 blocks 所对应的所有的 fence pointers 被顺序组织起来，存储在 sstable 中特定的区域，通常称为 index segment。sparse index 也可以是比较复杂的形式，例如 b+ tree 或其他。由于 sparse index 的总大小通常小于一个 block 的大小，则 index segment 通常也称为 index block。

这里需要提一下，为什么 fence pointer 只需要包含最大 key 或最小 key，而不是两者都包含呢？除了两端的 fence pointers，对于中间的某两个相邻的 fence pointers，前者的最大值就是后者的最小值（当然，考虑到区间闭开，它们的大小关系可能与此处所述略有不同）。顺数第一的 fence pointer 所包含的最小值与倒数第一的 fence pointer 所包含的最大值，实际对应 sstable 所包含的 keys 的最小值和最大值。这两个值要么存储在 sstable 的 header 或 footer 中，要么当作 manifest 的一部分被持久化，因此没有必要作为 fence pointer 的一部分被存储。总结，fence pointer 只需要包含最小值或最大值。关于 header, footer, manifest 之后会提及。

sparse index 的一个主要缺陷是，它并不能直接确定某个 key 是否在 sstable 中，原因嘛，因为它是 sparse 的。数据库中通常会使用一些 filter 以确认某个 key 是否不在某个 sstable 中，例如 bloom filter, cuckoo filter 等。因此，sstable 中通过还会额外存储 filter。这个 filter 可以是关于整个 sstable 的 filter，也可以是关于某个 block 的 filter，当然也可以两种都有。

keys 被存储在 data blocks 中，sparse index 被存储在 index block 中，filter 被存储在 filter block 中。由于每个 sstable 的大小不同，即 data blocks 的数量不一样，因此需要一个 header 或 footer 来说明 index block 和 filter block 相对于 sstable 文件开头的 offsets。选择 header 或 footer 都行，如果是 footer，则在写入所有其它 blocks 后，再写入 footer，这是很自然的。如果是 header，则需要在文件开头预留出一个 block，以存储 header，这也是没问题的。我选择 footer。另一方面，由于是以 block 为单位进行存储，最后一个 data block 很有可能存不满 keys，因此 footer 还需要包含该 sstable 中所存储的 keys 的数量。除这些之外，footer 还需要存储一些东西，此处不赘述。

关于 sstable 的设计，就说到这，之后讲 compaction。

由于同一个 key 可能会被多次写入，则 minor compaction 得到的 sstables 之间可能存在 key overlapping，即一个 key 的不同版本存在不同的 sstables 中。key overlapping 使得 sstables 之间无序，这使得查询需要以 sstable 为单位进行线性查询，而无法进行二分查找。显然这会降低查询的效率。因此，需要执行 major compaction，将存在 key overlapping 的 sstables 进行 merge，去

除因为 update 带来的 dup keys、以及因为 delete 带来的 invalidated keys，以删除冗余的 keys，最重要的是可以维护 sstables 之间的有序性，为二分查找提供基础。

major compaction 应该包括层内 compaction（或称 horizontal compaction）和层间 compaction（或称 vertical compaction）。考虑只有 horizontal compaction，那么也就意味着除了 memtable 之外，lsm tree 只有一层 sstables。仅有的那一层的 sstables 会随着 horizontal compaction 变得很大。这带来的一个主要问题是写放大。考虑某一次写入引发了 minor compaction，进而引发了 horizontal compaction，则需要对这些很大的 sstables 中的所有 keys 执行多路归并式的 merge。这个过程涉及到大量 disk io，有很大的写放大，且如果该数据库不支持后台并发 compaction，还会 block 读写。即使支持后台并发 compaction，这个 merge 过程仍然会 block 对于这些 keys 的读。

对 lsm tree 进行分层，可以分摊写放大。为每一层设置一个 size capacity，随着层数的增加，size capacity 逐渐增大。minor compaction 生成的 sstables 被放置到最底层，即第 0 层。当某一层的 sstables 的总大小超过 size capacity 时，执行 vertical compaction，将这一层的某些 sstables 与下一层（即层数较高的相邻层）的某些 sstables 进行 merge。merge 得到的新 sstables 被放置在下一层，参与 merge 的旧 sstables 则被删除。对于分层后的 lsm tree，考虑一次写入引发了 minor compaction，进而引发了 major compaction（可能是 horizontal compaction，可能是 vertical compaction，也可能都引发了），由于低层级 size capacity 的限制，低层级的 sstables 比较小，则 compaction 带来的写放大问题得到缓解。

更进一步，即使低层级的 vertical compaction 引发了高层级的 vertical compaction，即 cascading compaction，由于不同层级的 compaction 存在并发的操作可能，因此写放大问题进一步得到缓解。不同层级的 compaction 是有较大的并发可能性的，这是因为数据库的写入操作通常是间歇性的。那么在两次大规模写入之间，数据库可以利用这一段空闲时间在后台进行 compaction，这既可以使得写放大被分摊，也可以充分 overlap cpu 和 disk io。

对于 compaction block 读的问题，分层也是可以缓解该问题的。通常来说，数据库中的大部分数据是冷数据，即不会被频繁读写，少部分数据是热数据，会被频繁读写。热数据会有更多的版本，且较新的版本会集中在 lsm tree 的低层级，而冷数据则会由于 vertical compaction 被逐渐地下移。如果不分层，由于写放大和 block 读，那么位于同一层的大量的冷数据会降低少部分热数据的读写效率。分层后，由于对低层级数据的读写更快，故对热数据的读写也更快，即 block 读热数据的问题得到缓解。

由于需要 vertical compaction，则 lsm tree 应该至少两层，每一层包含 0 个、1 个或多个 sorted runs，每个 sorted run 包含至少一个 sstable，一个 run 中的所有 sstables 之间是有序的，且不存在 key overlap。这样的 sorted runs 均由 merge 得到。之所以引入 run 的概念，而不是由每个 level 直接包裹 sstables，是为了给 compaction policy 提供更多的自由度。例如两个经常用的 compaction policies：tiered compaction 和 leveled compaction，它们两个的一个重要的区别就是每一层所允许的 sorted runs 的数量不同。

引入了 run 的概念之后，每一层有两个预设的 capacity 阈值：run capacity 与 size capacity。当某一层的 runs 的总数超过 run capacity 时，执行 horizontal compaction。当某一层的 sstables 的总

大小超过 size capacity 时，执行 vertical compaction。注意，这里的 horizontal compaction 与之前所述的只有单层时的 horizontal compaction 有所不同。只有单层时，horizontal compaction 可以是定期执行的一个 routine，其主要目的是为了维护 sstables 之间的有序性，以提高读的效率。有了 run capacity 的限制之后，horizontal compaction 通常作为一种被动的 trigger 机制，其主要目的是使得某一层的 runs 的总数不超过所设置的 run capacity。

关于 compaction，还有很多可设计点，且在实现上也有不少细节，此处不赘述。

以上所述的主要是关于 write，那么数据库如何 handle read 呢？对于 memtable 的 read，因为完全在内存中，因此不需要什么特殊处理。对于 sstable 的 read，由于 sstable 的大小可能过大，导致不能一次性完全读取到内存中，因此需要采取流式读取的方式。这样的流式读取可以完全直接有操作系统提供的文件 io 接口实现，也可以由 mmap 这样的方法实现。通常来说，会在操作系统的接口上添加一层 iterator，所有的读均通过统一的 iterator 接口执行。

具体而言，data block 有一个 block iterator；sstable 有一个 sstable iterator，与 block iterators 交互；run 有一个 run iterator，与 sstable iterators 交互；level 有一个 level iterator，与 run iterators 交互；lsm tree 有一个 tree iterator，与 level iterators 交互。memtable 也有一个 memtable iterator。对于点读，首先通过 memtable iterator 在 memtable 中读。如果没找到，再通过 tree iterator 在 lsm tree 中读。对于区间读，当然可以拆分成多个点读，但通常会使用 heap 将 memtable iterator 与 tree iterator wrap 到一起，再进行流式点读。

关于 iterator 的使用，除了以上这种情况要使用 heap 外，还有几个地方需要使用 heap，借助 heap 才能实现多路归并式的流式读取。runs 之间可能存在 key overlap、可能无序，则 level iterator 使用 heap 维护 run iterators。不同的 level 之间可能存在 key overlap、可能无序，则 tree iterator 使用 heap 维护。

到此为止，db 已经能支持简单的 put, delete, get, range 命令了。也就是说，db 的基本功能已经有了。但是还有一个很关键的点，如果 db crash 了怎么办？sstables 已经被 persist 到了 disk 中，因此不需要 recover，只有内存中的 state 需要 recover。内存中的 state 包含这些：

- 存在 memtable 中的、尚未被 flush 到 sstable 的 keys
- database state，例如要分配的下一个 sequence number、要分配的下一个 file number（用来标识 sstable 文件）等。
- cache 在内存中的一些 metadata。例如每个 level、每个 run、每个 sstable 所包含的 keys 的最小值、最大值，这些 metadata 的作用类似于 sparse index，其作用是为了提高读取的效率。
- lsm tree 的结构信息，例如有几个 level，每个 level 包含几个 runs，每个 run 包含哪几个 sstables。

不妨将后三种 state 统称为 database manifest，简称为 manifest。由于是 barebone 的设计，因此不考虑复杂的 crash 情况，只考虑两次 write 之间的 crash。具体而言，只考虑本次 write 完全成功之后、下一次 write 开始之前的 crash。

对于 memtable 中的 keys，考虑使用 write ahead logging (WAL)。对于每个 key，在每次写入 memtable 之前，先将其以 log record 的形式写入到 log file 中。在 recover 时，通过 replay log file 中的所有 log record，即可 recover memtable state。具体而言，对于每一条从 log file 中读取到的 log record，调用 db 的 write 接口，将其所描述的 key 重新写入到 memtable 中。当然，此次不需要再写入到 log file 中。


对于 manifest，每次更新 manifest 之前，也以 log record 的形式写入到 log file 中。当然，log record 的 format 以及 log file 与 memtable keys 所对应的 format 和 log file 是不同的。通常来说，有两种存储 log record 的方式。一种是完全替换，即每次都让最新的、完整的 manifest 去替换 log file 中旧的、完整的 manifest。另一种是增量式的更新，即每条 log record 存储的是 manifest 相较于上一条 log record 的 update。在 recover 时，通过 replay 的方式来 recover crash 前完整的 manifest。前者实现起来很简单，但是如果 manifest 较大，则 disk io overhead 较大，这会降低写的性能。后者实现起来较复杂，但写入 log record 时所需的时间较小。

参考的一些东西：

- [leveldb 源码与 doc](#)

leveldb/doc at main · google/leveldb

LevelDB is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values. - leveldb/doc at main · google/leveldb

 <https://github.com/google/leveldb/tree/main/doc>

google/leveldb

LevelDB is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string...

55

Contributors

2


Used by

32k

Stars


7k

Forks



- [某个人写的 leveldb handbook](#)


leveldb-handbook — leveldb-handbook 文档

 <https://leveldb-handbook.readthedocs.io/zh/latest/>

- [rocksdb wiki](#)

Home · facebook/rocksdb Wiki

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or window. Reload to refresh your session. Reload to refresh your session.

 <https://github.com/facebook/rocksdb/wiki>

facebook/rocksdb

A library that provides an embeddable, persistent key-value store for fast storage.

702

Contributors

1


Used by

25k

Stars

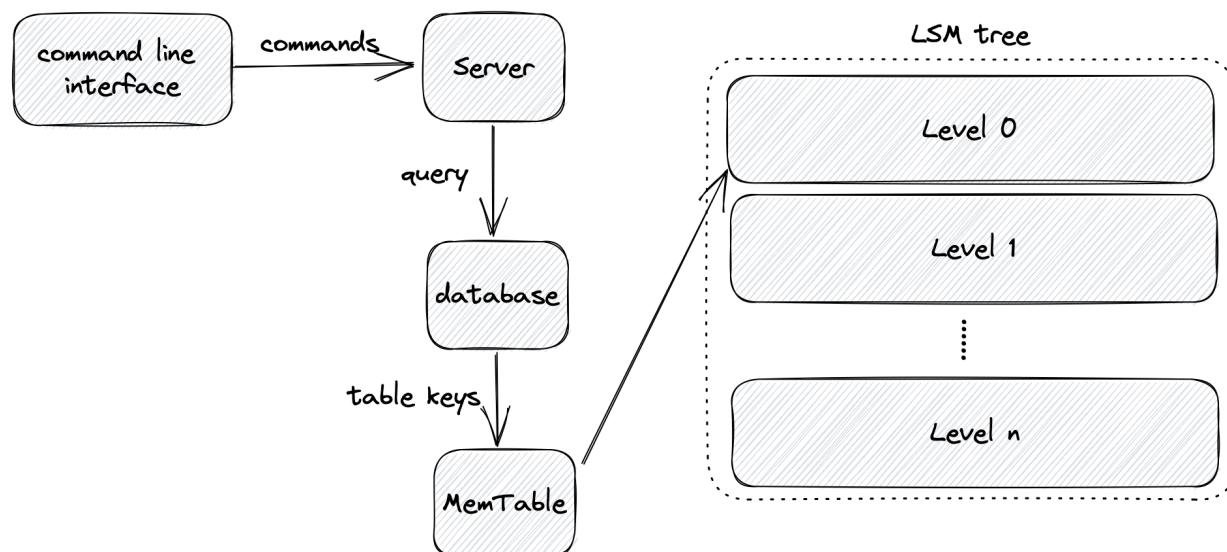
6k

Forks



我实现的 LSM-DB 是如何工作的

client-server 交互的设计



command line interface (cli) 作为一个简易的 client，接收用户的 commands，转发给 server。这些 commands 一部分是 database 的 domain specific language (dsl)，包括 put, delete, get, range，一部分是与 server 交互的 commands，包括 load, quit, help, print stats 等。

server 接收到 commands，先判断 commands 的合法性，再进行 handle。如果是 dsl，则交给 db 进行 handle。如果是与 server 交互的 commands，则自己 handle。

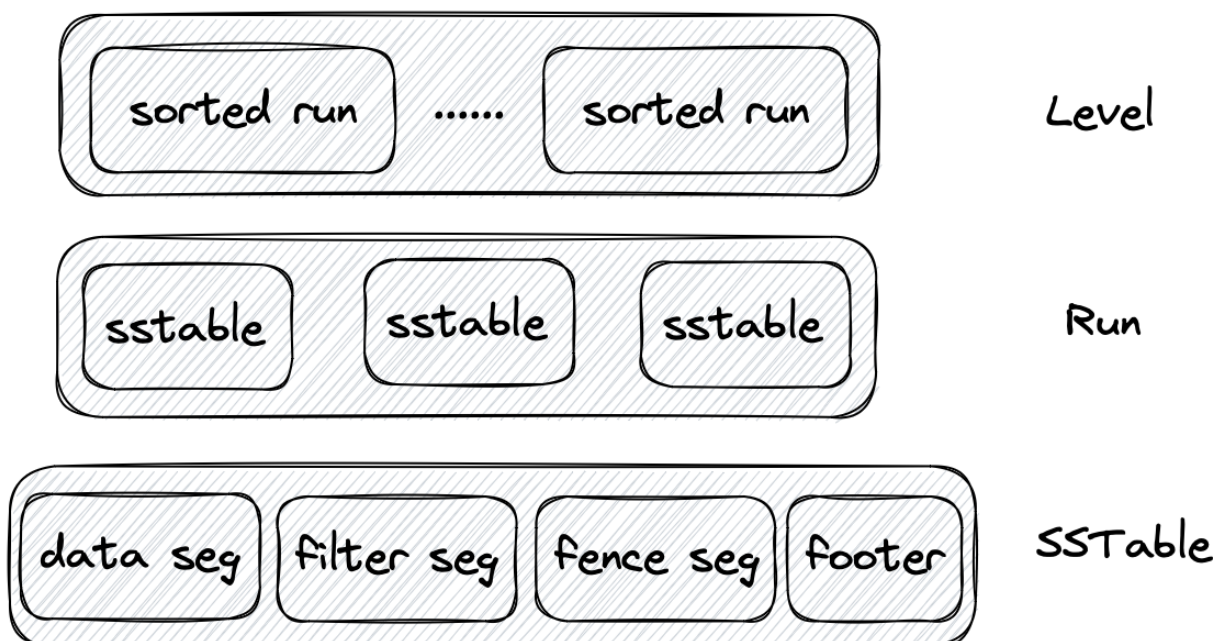
对于每一个 dsl commands，database 都暴露出了相应的接口。对于 put 和 delete，database 将 user key 封装成 table key，然后交给内部的 write 接口写入到 memtable。对于 get 和 range，database 将 user key 封装成 lookup key，然后调用对应的接口进行 point query 或 range query。详见之后关于读写请求的 handle 流程。

key 的设计

数据库中存在三种 key：user key, table key 和 lookup key。user key 是用户发送过来的最原始的 key。由于目前只考虑有符号 32 位的 user key，故 user key 的类型为 i32。table key 是聚合后的 key，包含 user key, sequence number（之后简称 seq num），write type, user val。其中 write type 是一个 enum，有 put, delete, empty 三个 variants。table key 中各部分的排列顺序并没有特别的讲究，但通常会将 user key 和 sequence number 放在一起。这是因为有些数据库为了能够统一处理多种 user key 类型，会在内存中就将 user key 转换为 bytes 进行存储。table key 中的其它部分也会被 bytes 化。在进行比较时，需要将 user key 和 seq num 聚合在一起，因此在存储时就将 user key 和 seq num 紧邻存放对于 slice 操作比较友好。另一方面，如果考虑到 cache，那肯定也是紧邻着存放比较好。

在 handle 读时，用户只提供了 user key，然而比较 table key 时需要用到 user key 和 seq num，故需要为每一次读分配一个 seq num。这样的 seq num 通常称为 snapshot seq num。数据库中所有 $\text{seq num} \leq \text{snapshot seq num}$ 的所有现存的 keys 组成一个版本号为 snapshot seq num 的 snapshot。对于单线程数据库，这样的 snapshot seq num 总为最新的 seq num。如果考虑到并发，这样的 snapshot seq num 不一定为最新的 seq num。由 user key 和 snapshot seq num 构成的 key 称为 lookup key。在比较时，lookup key 会被转化为对应 table key，其中 write type 为 empty, user val 为默认的 0 值。

LSM Tree 的设计



lsm tree 由多个 levels 构成，level 从 0 开始编号。每个 level 中包含 0 个、1 个或多个 sorted runs。lsm tree 是动态扩展的，初始时 lsm tree 只包含一个空的 level 0。有一个 max level config，用来限制 lsm tree 的最大层数。

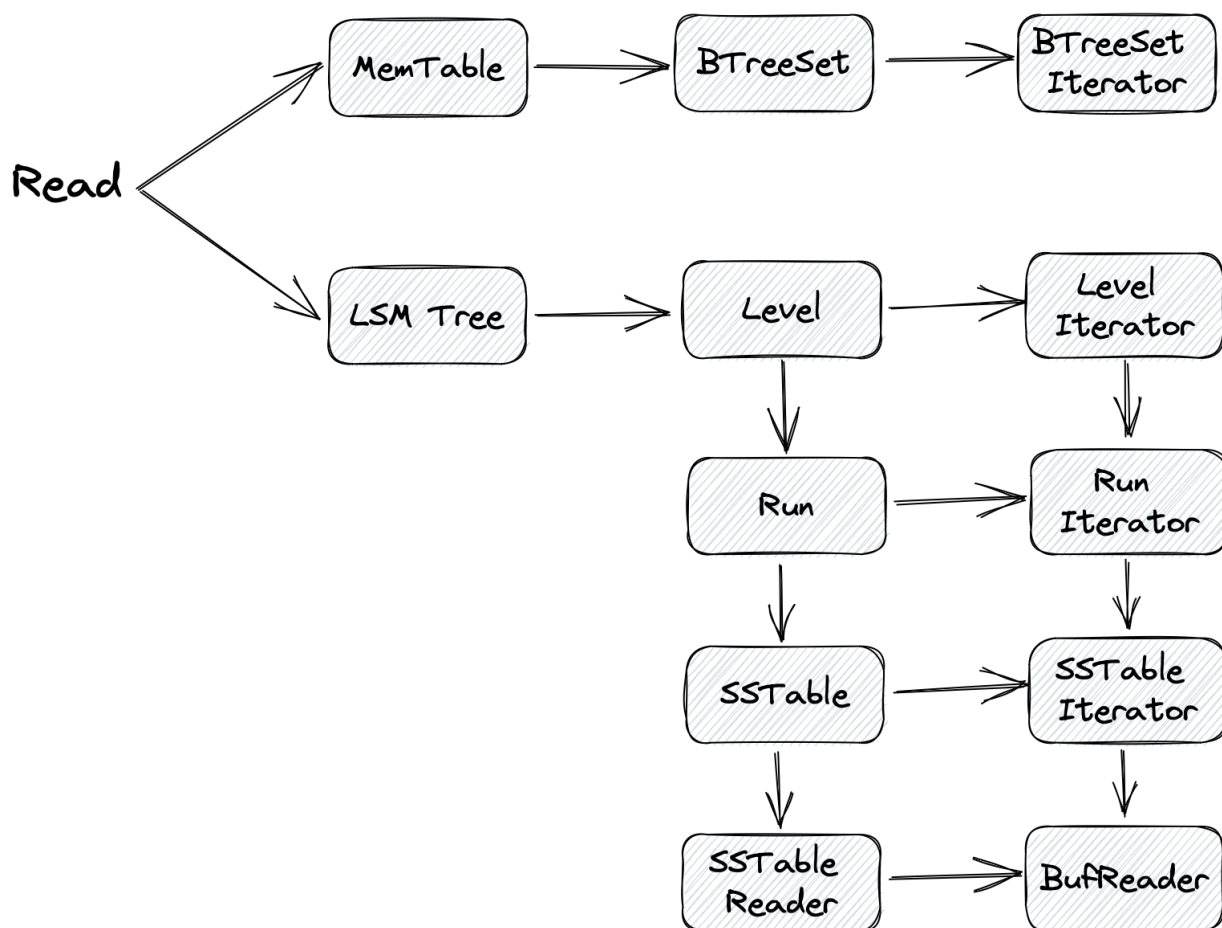
每个 sorted run 中包含 1 个或多个 sstables，同一个 sorted run 中的 sstables 相互之间不存在 user key overlap，且这些 sstables 按照它们所包含的 key 的最大值，即 max table key，从小到大顺序排列在 run 中。

table key 的顺序是这样定义的：user key 不等时，user key 越小，table key 越小。user key 相等时，seq num 越大，table key 越小。由于存储时按照从小到大进行存储，因此越小的 table key 排在越前面。只有这样的顺序定义，才能正确驱动基于 min heap 的多路归并式的流式读取。

一个 sstable 由多个 segments 组成，包括 data segment, index segment, filter segment 和 footer。data segment 中通常占用多个 blocks，其他 segment 通常只分别占用一个 block。每个 data block 中包含一系列的 table keys。每个 data block 有一个对应的 fence pointer，存储在 index block 中。filter block 中存储的是由该 sstable 中所有 table keys 生成的 bloom filter 序列化后的 bytes。

如何处理读请求

memtable, block, sstable, run, level, lsm tree 都实现了统一的 table key iterator trait，读请求均通过该接口执行。



Get

db 首先调用 memtable 的 get 接口，如果没有找到，则在 lsm tree 中从低往高进行查找，直到在某一层找到或最终都没有找到。

memtable 的 get 接口通过 memtable iterator 进行查找。memtable 的 backing 数据结构是 rust 标准库中的 btree set。memtable iterator 则是 btree set iterator 的 wrapper。由于 rust orphan rules

的存在，不能直接为 foreign btree set iterator 实现自定义的 trait，故按照 newtype pattern 将 btree set iterator wrap 在了 memtable iterator 中，然后对其实现 table key iterator trait。


level 中 runs 是无序的、有 key overlap 的，run 中的 sstables 是有序的、无 key overlap 的。基于此特性，level iterator 被设计为一个维护着所有 run iterators 的 heap，run iterator 则被设计为 sstable iterator 的 chain。sstable iterator 由一个 buffer reader 和一个 data block iterator 构成。在 sstable iterator 创建时，buffer reader 负责读取 footer, index block 和 filter block，其中 index block 和 filter block 被 cache 在内存中。之后，buffer reader 负责读取 data blocks，每次将一个 data block 读取到内存。sstable iterator 中的 block iterator 则始终对应当前 cache 在内存中的那个 data block。由于这个 data block 中的 keys 完全被 cache 在内存中，且为顺序存放，则 data block iterator 为一个简单的 cursor，指向下一个要读取的 table key。

table key iterator trait 有一个 seek 方法。对于 get 所指定的 table key，seek 会将 iterator 指向大于或等于这个 table key 的最小的 table key。对于大部分 iterator 的实现，都是通过直接调用 iterator 的 next 方法进行线性查找。对于 run iterator，由于 run 中的 sstables 是有序、无 key overlap 的，则可以利用每个 sstable metadata 中所保存的该 sstable 中所含 keys 的 max table key，进行二分查找。对于 sstable iterator，由于 sstable 中的 data blocks 是有序、无 key overlap 的，则可以利用 index block 中的 fence pointers 进行二分查找。定位到可能存在给定 key 的那个 data block 后，再利用 data block iterator 进行线性查找。

关于 orphan rule 和 newtype pattern，参考：

Generic Newtypes: A way to work around the orphan rule | Thomas Eizinger

Rust's orphan rule prevents us from implementing a foreign trait on a foreign type. While this may appear limiting at first, it is actually a good thing and one...

 <https://blog.eizinger.io/8593/generic-newtypes-a-way-to-work-around-the-orphan-rule>



Range

对于 range，使用一个 min heap 维护 memtable iterator 和 tree iterator，不妨称为 db iterator。通过不断地 pop 这个 binary heap 以 emit table keys，再 collect 在 range 内的所有未删除的 table keys 所对应的 value。

Bloom Filter 设计

bloom filter 是这样的一个东西：它维护了一个长度为 m 的 bit array。在 insert key 时，这个 key 会被 k 个 hash funcs 独立 hash，每个 hash value 通过 modulo m，被 delimit 到 [0, m) 这个 range，对应的这个 bit 则被置 1。这些 k 个 hash funcs 是 static 的，也就是说对于同一个 key，其多次 hash 的 hash value 是一致的。这就引出，每个 key 对应 bit array 中的一系列被置为 1 的 bits。

由于 hash func 本身并不能保证 unique 的 hash values，也因为 modulo 操作的存在，不可避免地会有 collision。也就是说，可能存在多个不同的 key 被 hash 到同一组 bits。这就产生了 false positive：即对于一个实际并没插入到 bloom filter 中的 key，bloom filter 通过 check 它所对应的那一组 bits，发现这些 bits 都为 1，则报告说这个 key 存在。实际上，这些 bits 可能是在 insert 其它 keys 的过程中被置 1 的。

在 bloom filter 的设计过程中，需要考虑四个参数：

- false positive rate: p
- #hash functions: k
- #bits in the array: m
- #keys expected: n

关于这四个参数的关系，有这样两条重要的结论：

So the optimal number of bits per element is

$$\frac{m}{n} = -\frac{\log_2 \varepsilon}{\ln 2} \approx -1.44 \log_2 \varepsilon$$

with the corresponding number of hash functions k (ignoring integrality):

$$k = -\frac{\ln \varepsilon}{\ln 2} = -\log_2 \varepsilon.$$

其中， ε 即 false positive rate p 。详细的推导，见：

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

这两条结论告诉我们，只要指定了 false positive rate p ，即可确定最优的 #hash functions k 以及最优的 bits per key，即 m/n 。

在实际应用中，通常是先指定 false positive rate p ，然后根据第二条结论求出 k ，此时即确定了四个参数中的两个。再根据第一条结论，求出 m/n 。知道了 m/n ，即可根据估计的 n ，即大概会有多少个 keys，求出 m 。

在 LSM-DB 中，sstable 是基础的存储单位，block 是基础的读写单位。可以考虑为每一个 block 都构建一个 bloom filter，这样在有些时候就可以避免对于整个 block 的查找。每个 block 会被先读

取到 memory，然后再进行二分查找，这样的在内存中的查找是很快，因此对每个 block 都构建一个 bloom filter，显得不是很必要。因此在 LSM-DB 中，每一个 sstable 在存储时会额外在 filter block 中存储一个序列化后的 bloom filter，里面存储着 insert 整个 sstable 中所有 keys 之后 bit array。

在查找时，将 filter block 读取到内存，反序列化重构出 bloom filter，然后在这个 bloom filter 中查找给定的 key。如果 filter 说肯定不在这个 sstable 中，那么不需要再将 data blocks 读取到内存。如果 filter 说可能在，则需要继续在 data blocks 二分查找。

在我的设计中，bloom filter 的四个参数是这样确定的：

设置 tolerant 的 false positive rate p 为 $1/100$ ，则根据第二条结论，得出 $k = 6.6$ 。根据公式知， k 越大，false positive rate 越小，则不妨取 $k = 7$ 。再根据第一条结论，得出 bits per key，即 $m/n = 9.57$ 。

假设 sstable 只保存 table keys，考虑到一个 table key 占用 17 bytes，其中 user key, user val 各占用 4 bytes，write type 占用 1 byte，sequence number 占用 8 bytes，则一个 sstable 能保存的 table keys 的数量范围大概为 240 ~ 963。由于 m/n 越大，false positive rate p 越小，则不妨取 $n = 1000$ ，再不妨取 $m = 10000$ 。这里主要考虑的是取整会比较方便。

至此，得出 bloom filter 的四个参数： $p = 1/100$, $k = 7$, $m = 10000$, $n = 1000$ 。当然，由于 sstable 的大小不一致，可以像 leveldb 那样，先给出大概的 m/n （给出 m/n ，即给出了 false positive rate p ），然后根据第二条结论算出 k 。然后根据第一条结论及每个 sstable 实际存储的 keys 的数量 n ，得出 m 。为了简单起见，我的设计中将这些量都固定了。在序列化时，由于其他参数都固定了，因此只需要将 m 个 bits 进行序列化。

关于 hash funcs，有两点需要考虑：怎么生成 k 个 hash funcs，以及选择什么 hash funcs。

对于这 k 个 hash functions，可以事先准备 k 个。也可以准备几个 hash func seeds，然后使用 double hashing 生成 k 个 hash funcs。一种 double hashing 的方法是：给出两个不相关的 hash func seeds $f(x)$ 和 $g(x)$ ，则生成的 hash funcs 具有一致的形式： $h(x) = f(x) + i * g(x)$ ，其中 i 为一个参数。如果把 double hashing 用在 probing-based hash func 中， i 即为步长 step。在这里，这个参数可以随意设置，有几个 unique 的 i ，则生成几个 hash funcs。

leveldb 使用的 double hashing 方法与上面描述的有一点不同，主要体现在第二个 hash func seed。leveldb 没有选择两个 hash func seed，而只选了一个 seed，不妨设为 $f(x)$ 。其第二个 hash func seed 通过对 $f(x)$ 的结果执行 bit shifting 操作得到。

hash funcs 有这样一种划分方式：加密型与非加密型。前者表示这个 hash 用于有加密需求的场景中，因此发生 collision 的概率更低，但 hash 的效率也更低。对于使用在 bloom filter 中的 hash funcs 而言，没有加密需求，因此 LSM-DB 使用了两个比较常见的非加密型 hash funcs：murmurhash 和 xxhash。这两个 hash funcs 作为 seeds 用来生成 k 个 hash funcs。

如何处理写请求

一个 table key 首先被写入到 memtable，这里直接调用了作为 memtable backing 数据结构的 btree set 的写接口。如果在本次写入后，发现下一次写入会超过 memtable 的 size capacity，则执行 minor compaction，将 memtable 中的 table keys 提取出来，写入到一个新创建的 sstable file 中。这个写入由 sstable writer 执行，写入的 dirty work 由其内部包含的一个 rust 标准库中的 buffer writer 完成。sstable writer 维护了一系列的 block cache，包括 data block cache, index block cache 和 filter block cache。当 data block cache 满了或 table keys 写入完毕时，将 data block cache flush 到 sstable file 中，同时将这个 data block 所对应的 fence pointer 写入到 index block cache 中。对于每个写入的 key，其还会被写入到 filter block cache 所对应的 bloom filter 中。当所有 data blocks 写入完毕后，将 index block cache 和 filter block cache flush 到 sstable 中。最后将 footer 写入到 sstable 中。

每次 minor compaction，一个叫做 `check_level_state` 的方法会被调用，其从低到高检测每个 level 的 state。level state 是一个 enum，其有 exceed run capacity, exceed size capacity 和 normal 三个 variants。每个 level 都有一个 run capacity 参数，目前统一设置为 4。每个 level 还有一个 size capacity。从 level 1 开始，每个 level 的 size capacity 是上一个 level 的 capacity 的 fanout 倍。fanout 也是一个可配置参数，默认为 10。level 0 比较特殊，因为 level 0 中存储的 sstables 都是从 memtable flush 而来，则 level 0 中的 sstables 的大小相当。memtable 也有一个 size capacity 参数，因此对于 level 0，只需规定 run capacity，其 size capacity 则为 run capacity * memtable size capacity。当然也可以为 level 0 设置一个独立的 size capacity 参数，但那样的话会多出一个参数。通常来说，应该遵循最少、最必要参数原则。

如果检测到某一层的 state 不为 normal，则会触发该层的 major compaction。major compaction 分为 vertical compaction 和 horizontal compaction 两种。当 size 超过 size capacity 时，触发 vertical compaction，当 run 的数量超过 run capacity 时，触发 horizontal compaction。由于可能同时触发两种 compaction，则每次 major compaction 完毕后，还会继续检测当前层的 state。如果有必要，则再次对该层执行 major compaction。

vertical compaction 是这样的：首先在当前 level 中随机选择一个 run，再从这个 run 中随机选择一个 sstable。以该 sstable 的 user key range 作为 base key range，在当前 level 的其他 runs 中搜寻所有存在 user key overlap 的 sstables。将这些 sstables 从它们所属的 runs 中抽离出来。这些 sstables 的 user key range 再作为当前 level 的 key range，去下一层的所有 runs 中搜寻所有存在 user key overlap 的 sstables。同样，将这些 sstables 从它们所属的 runs 抽离出来。如果这样的抽离使得某个 run 变空了，即其中没有 sstables 了，那么将该 run 从所属的 level 中删除。

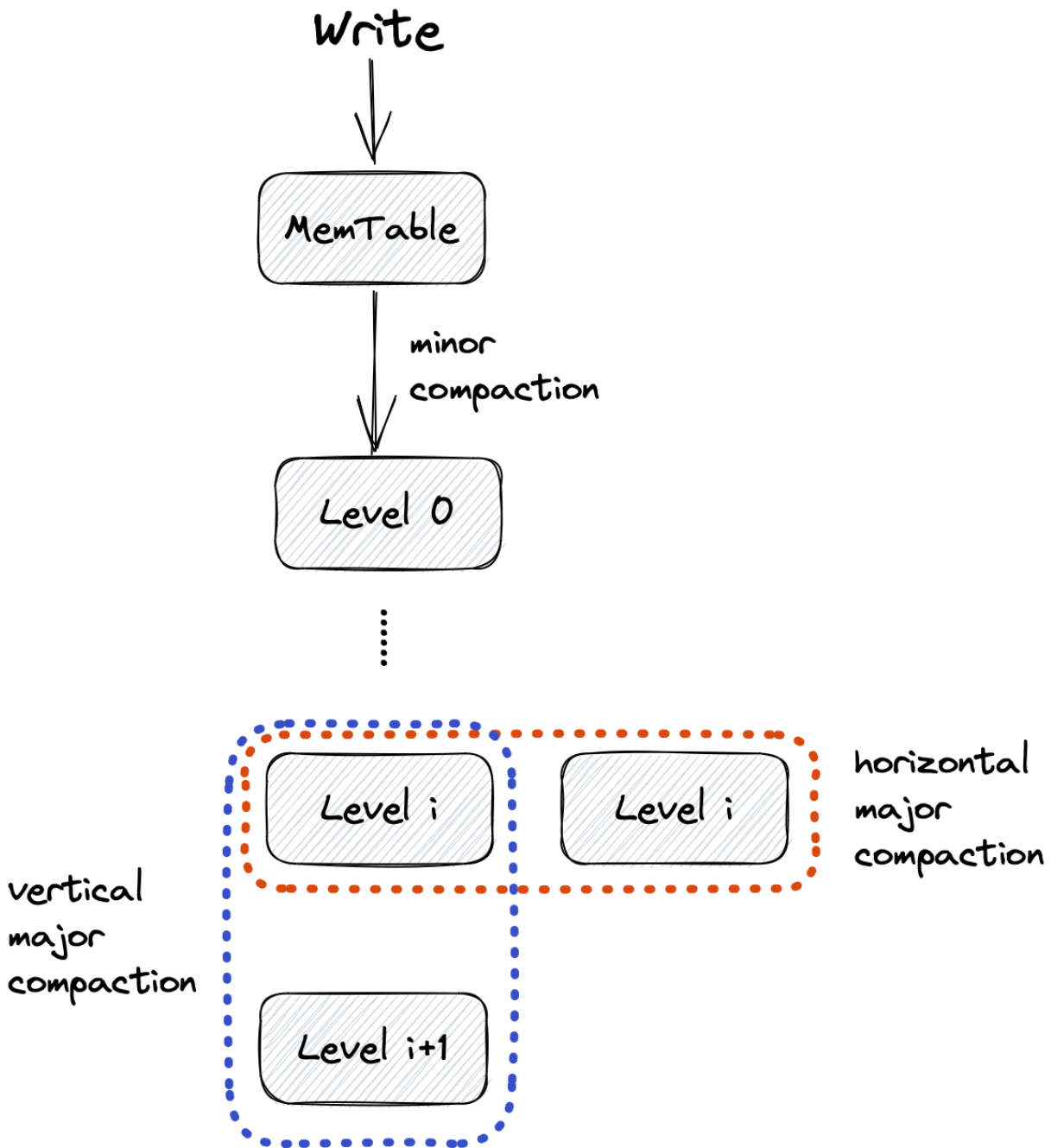
所有上述找到的 sstables 将作为 vertical compaction 的 input sstables。这些 input sstables 中的 table keys 会被 merge，即对于相同 user key 的所有 table keys，只有最新的那个 table key 才会被保留。merge 操作由一个 binary heap 以多路归并的方式完成。merge 保留的所有 table keys 会被写入到一个或多个新创建的 sstable files 中。由于保留的 table keys 可能很多，而 sstable file 有预设的容量限制，因此一个 sstable file 可能写不完。

在实现上，compaction 的相关信息，例如 key range, input sstables, output sstables 等信息，会由一个 compaction context 结构维护。对 merge 后的 table keys 的写入则通过一个 sstable writer batch 进行。其内部会维护一个 sstable writer，当这个 sstable writer 写满一个 sstable 后，会

harness 这个 sstable，存在内部维护的 output sstables 中。之后再刷新 sstable writer，准备将 table keys 写入到下一个 sstable file。

horizontal compaction 的流程是这样的：在当前层随机选出一个 base run，再从当前层选出与该 run 存在 user key overlap 的所有 runs。如果不存在这样的 run，则再从当前层随机选择一个 run。所选出的 runs 中的所有 sstables 作为 compaction input sstables，对这些 sstables 执行 merge 操作，生成一个新的 run，放置在当前层。注意，目前关于该部分的实现还有未找到的 bug，使得有些测试并不能稳定通过。

参与 compaction 的所有 sstables 将成为 obsolete sstables，major compaction 执行完毕之后，将这些 sstables 全部删除。目前，这样的删除是 eager 的，实际可以为 lazy。



如何执行 recovery

LSM-DB 在启动时，会调用 `recover` 函数，执行 recovery。这个函数首先读取已经被 persisted 的 manifest。这个 manifest 为整个 db 的 manifest，它包含了这么几层 manifest：db manifest, level manifest, run manifest, sstable manifest。这些 manifest 即为 db, level, run, sstable 的一些状态，或称 metadata。具体包含什么，在之前的章节中已有描述。每次这些 metadata 改动时，生成一份完整的 manifest，将其写入到特定的文件中。关于写入完整的 manifest，还是写入 manifest 的增

量改动，在之前的章节已有讨论。`recover` 函数即通过这些 metadata 重构出内存中的 db, level, run, sstable 的 struct instances。

关于 recover memtable 中的 keys，存在一个 log writer，负责在写入一个 key 到 memtable 之前，先将其写入到 log file 中。也有一个 log reader，负责在 recover 时将 log file 中的 keys 全部读取到内存中，然后按顺序插入到 memtable 中。这里是直接插入到 memtable 中，没有经过 db 的 write 接口。这是因为 log write 和 allocate seq num 都发生在 db 的 write 接口中，recover 显然不应该再次触发这些东西。

这里需要提一下 log file 与 memtable 的对应关系。任何时刻，disk 中只有一个 log file，其对应当前的这个 memtable。在 crash 之前，在写入一个 key 前后，log file 与 memtable 中存的 keys 完全相同。也就是说，即使在一次 minor compaction 将 memtable 清空之后，log file 与 memtable 中存的 keys 也是完全相同的，即都为空。使 log file 为空的方法是这样的：在 minor compaction 之后，将旧 log file 删除，再创建一个同名的新文件。

这里有一个可能引起 inconsistency 的点：如果在 minor compaction 后，在将旧 log file 删除之前，db crash 了，那么显然会有 inconsistency。但是由于 LSM-DB 暂时只考虑简单的 restart 时的 recovery，因此这样的设计是可行的。