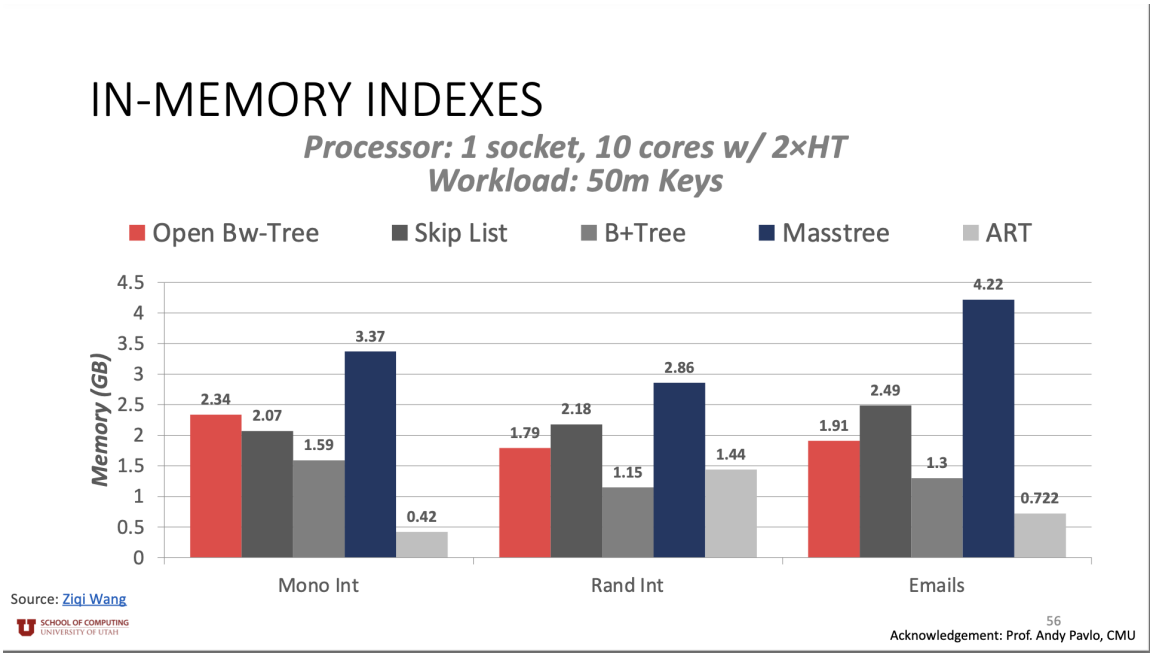
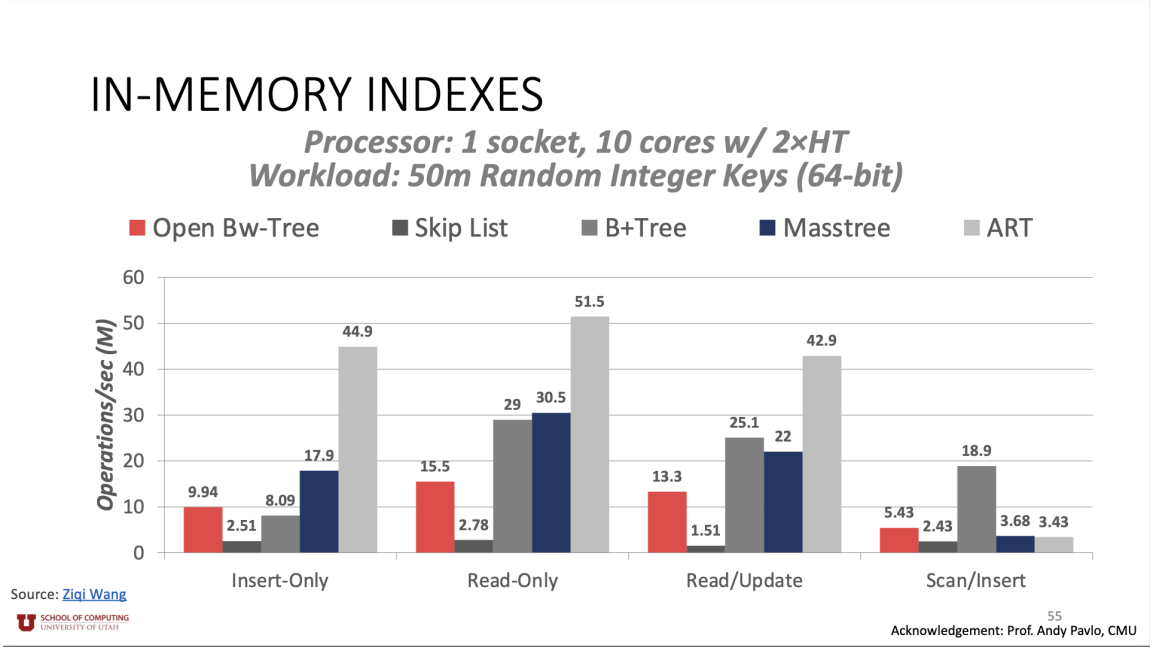
LSM-DB 的设计点

- client-server 交互方式
  - 本地 command line interface
  - 网络连接 tcp-rpc。
- 是否支持并发读写
  - 如何支持
- 是否支持批量写，即 write batch
  - 如何做 write batch
- 是否有 cache 层
  - 哪些地方加入 cache 层
  - 如何保证 cache consistency，即 in-memory cache 中的数据与 on-disk sstable file 中的数据的 consistency
  - 使用什么 cache eviction policy，LRU-K 等
- key 的设计
  - user key 和 user val 是否绑定在一起
    - 聚合为一个 table key
    - 还是 key 和 val 分开存储
  - 需要哪些 key
    - table key
      - 应该包含什么：user key, user val, delete marker, seq num
    - lookup key
      - 应该包含什么：user key, seq num
  - 支持哪些 user key 类型
    - 固定支持一种类型
    - 支持多种类型，全部都转换为 bytes string，然后由用户指定 comparator
- memtable
  - single buffer vs. double buffer。double buffer 即：使用一个 mutable buffer 和一个 immutable buffer。

- backing 数据结构，即 in-memory index
  - vector
  - b tree
  - skiplist
  - radix tree





- sstable
  - 是否有大小限制
  - 是否 approx equal size
  - table key 怎么存储，即 sstable 的 backing 数据结构
    - vector
    - b tree
  - 是否额外存储稀疏索引（sparse index）
    - 什么样的稀疏索引
      - fence pointers
      - b+ tree
  - 是否额外存储过滤器（filter）
    - 什么过滤器
      - bloom filter
      - cukcoo filter

Cuckoo Filter | Brilliant Math & Science Wiki

The Cuckoo Filter is a probabilistic data structure that supports fast set membership testing. It is very similar to a bloom filter in that they both are very fast and space efficient. Both the bloom filter and

https://brilliant.org/wiki/cuckoo-filter/

Probabilistic Filters By Example: Cuckoo Filter and Bloom Filters

https://bdupras.github.io/filter-tutorial/

https://www.eecs.harvard.edu/~michaelm/postscripts/cuckoo-conext2014.pdf

  - header vs. footer
    - header 或 footer 应该存储哪些东西
- lsm tree
  - 是否分层？

- 首先要说明的是，不分层不代表没有 compaction。虽然没有 vertical compaction，但是可以有 horizontal compaction。也就是说，唯一的这一层中的 sstables 虽然均经由 minor compaction 得到，即它们分属于不同的 sorted runs，但是经过 merge 之后，这些 sstables 之间还是可以维持有序性。

  由于 merge 的存在，对于分层或不分层，stale keys 和 invalidated keys 所带来的 space amp 其实都是一样的。虽然，分层肯定需要一些额外的信息，例如 lsm tree 层级 metadata，但这些 metadata 比较小，可以忽略。

  对于 write amp，分层也是不能解决的。考虑一次写入引发了一次 minor compaction，生成了一个 sstable。如果不分层，那么进行 merge 时，假设这个 sstable 的 key range 刚好覆盖了其它所有 sstables 的 key range，那么这次 merge 会 involve 所有的 sstables。如果分层，虽然这些 sstables 中的一部分可能会被存储在 lsm tree 的更高层，但是最终还是会被卷入到 merge 当中，可能是 involve 到此次 minor compaction 所引发的 major compaction，也可能是后续的 major compactions。

  对于 read amp，分层也是不能解决的。如果分层，考虑对于一个存在的 key 的 point query，最坏情况下，它会被存储在最高层，则本次 read 会 involve 所有 sstables。考虑对于一个不存在的 key 的 point query，在低层级找不到，就会继续在高层级找，则本次 read 也会 involve 所有 sstables。如果不分层，则需要对所有的 sstables 进行一次二分查找，则也会 involve 所有 sstables。也就说，分层或不分层，read amp 在最坏情况下是一样的。

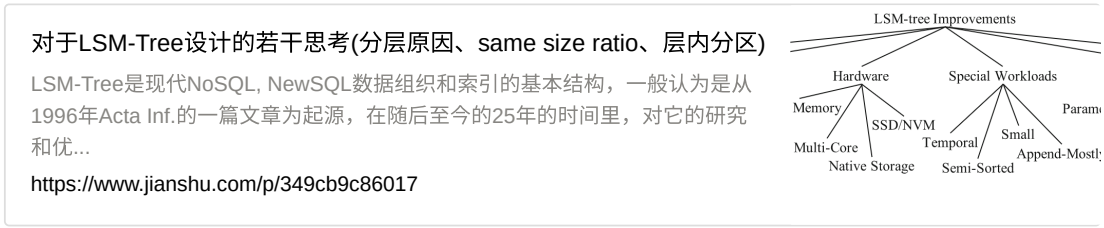
  这里还要提一下，分层实际上会降低读的性能。如果不分层，binary search on all sstables 的 step 相比较分层情况下的 step 是更大的，因此所有的 sstables 被集中在一起，进行 binary search。牺牲一点读的性能是值得的，这是由 lsm tree 的预期使用场景决定的。lsm tree 通常作为一个较底层的 storage engine 的 backing data structure。对于一个数据库而言，会有很多 cache 层。对于一个存在的 key 的 query，大部分会被 cache 层所 handle。对于一个不存在的 key 的 query，大部分会被 filter 层所 handle，例如 bloom filter。因此 lsm tree based storage engine 实际 handle 读的机会是比较少的。而对于写，都需要经过 lsm tree based storage engine。因此，对于这种写多读少的场景，牺牲一点 read performance 以 optimize write performance 是值得的。

  那么分层有什么用呢？有两个主要的作用：分摊 write 所引发的 write amp，减少热数据的 read amp。

  分摊 write 所引发的 write amp：对于 write amp，虽然一次 write 所引发的 write amp 的总量并不能减少，但是这些 write amp 可以被分摊。这是因为，不同层级的 compactions 可以异步进行。如果分层，考虑一次写入引发了一次 minor compaction，生成了一个 sstable。这个 sstable 被插入到第 0 层时，触发了 major compaction。对于最坏情况，即这次 major compaction involves lsm tree 中所有的 sstables，由于分层，这些 sstables 只有一小部分位于第 0 层，则第 0 层的 compaction 就会很快。即使这次 compaction 引发了 cascading compaction，高层级的 compactions 可以异步进行。尤其对于数据库的读写可能是间歇突发的，也就是说数据库可以在两次大量读写之间的空闲时间段，schedule 这些 compactions。

减少热数据的 read amp：如果不考虑数据的冷热，那么分层对于 read amp 在最坏情况下是没啥用的。但是通常来说，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。冷热数据比大概是 80 / 20。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，热数据的旧版本和冷数据则会由于 vertical compaction 被逐渐地下移。如果不分层，由于写放大和 major compaction 的 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。如果分层，则对热数据的读通常只会 involve 低层级的 sstables，则对于热数据的 read amp 会降低。且分层后，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。

  - 分几层
  - 第 0 层的 size capacity 怎么设置？
    - 由于第 0 层的每个 sorted run 只包含一个由 memtable flush 而来的 sstable，而 memtable 的 size capacity 是一个参数。不妨将第 0 层的 size capacity 定义为 memtable size capacity 的某个倍数。这样的话，只需要调整 memtable 的容量大小，就可以控制第 0 层的容量大小，避免新增一个参数。
  - 相邻层之间的 size capacity ratio，即 fanout，是否应相等？应设置为多少？
    - fanout 设置为相等，可以最优地减小写放大。原因，参考：



- compaction policy，见下面的详细说明

Compaction

首先需要提一点，compaction policy 是关于 level 的，而不是关于整个 lsm tree 的。因此，lsm tree 的不同 levels 可以使用不同的 compaction policies。

Read, Write and Space Amplification

讨论 compaction policies，首先要介绍 read, write, space amp。类似分布式领域中的 CAP, FLT 理论，对于一个 backing storage，其也只能最多 optimize read, write, space amp 三者中的两个。

关于 amplification 的量化，有很多需要讨论的点：

- 如何计算 amp：
  - total #ops：一次 read/write 触发了几次 ops

    - #ops diff：实际触发的 #ops - 预期的 #ops
    - #ops ratio：实际触发的 #ops / 预期的 #ops
- 需要计算哪些地方的 ops：in-memory ops, on-disk ops, or both
- 如何量化一个 op：例如一次 memory access, 一次 disk seek，一次 data transfer between disk and memory 等等

关于 amplification 的定义，有很多种不同的定义。下面给出我所使用的定义：

- read amp：对于一次 point query，read amp = #bytes reads / kv pair size。假设数据库使用一个 unsorted array 作为 backing storage，那么一次 point query 最坏情况下需要查找整个 array。在这种情况下，read amp = array length * kv pair size / kv pair size = array length。
- write amp: 对于一次 put 或 delete，write amp = #bytes written / kv pair size。例如在 LSM-DB 中，一次 write 可能触发一次 compaction，则实际写入的 bytes 远大于预期要求写入的一个 kv pair 的 size。
- space amp: 数据库实际占用的 size in bytes / 有效的 kv pairs 占用的 size in bytes。例如，在 LSM-DB 中，为了存储一个 kv pair，除了 key 和 value 之外，还需要存储 delete marker, seq num 等。又例如，有些数据库使用的 backing data structure，例如 tree，也会存储一些 metadata，例如 children 指针 等。至于“有效的”如何定义，考虑一些无效的 keys：由于 update 产生的 stale keys，由于 delete 产生的 invalidated keys。

Compaction characteristics

为了 define 一个 compaction policy，需要考虑这些 characteristics：

- compaction trigger：什么东西会触发一次 compaction
- compaction granularity：每次 compaction 选择的东西是以 level 为单位，还是以 sorted run 为单位，还是以 sstable 为单位。
- file picking strategy：这里的 file 的定义当然是宽泛的，对应上面的“单位”。

Compaction trigger

通用的说法是，当 size 超过了预设的 capacity，则会触发 compaction。这里的 size 是一个宽泛的概念，可以指代很多东西。我总结的 trigger 方式有这么几种：

- size based：
    - bytes：当一个 level 所包含的所有 sstables 的 total size in bytes 超过了阈值后，触发该层的 compaction。
- count based：
    - #sorted runs：当一个 level 所包含的 sorted runs 的数量超过了阈值后，触发该层的 compaction。

    - #sstables：当一个 level 所包含的 sstables 的数量超过了阈值后，触发该层的 compaction。
- time based：
    - ttl：每个 sstable 在创建时会被 attach 一个 ttl。存在一个后台进程会周期性地检查这些 ttl，当然是 cache 在 memory 中的 ttl。如果发现一个 sstable 的 ttl 过期了，则会 schedule 一个 involve 这个 sstable 的 compaction。
    - period：存在一个后台进程周期性地尝试 schedule 一个 compaction。

大概来说，这些 trigger 方式有着不同的侧重点。

size based 主要是考虑到 write amp，通过限定低层级的 sstables 的 total size in bytes，可以减小 involve 到 major compaction 的 bytes，则达到减小该层的 write amp 的目的。

count based 主要是考虑到提高读的性能。因为不同 sorted runs 之间可能存在 key range overlap，则无法进行二分查找。通过减小 sorted runs 的数量，可以更频繁地触发 horizontal compaction，则可以 merge stale 和 invalidated keys，提高读的性能。要注意的是，如果是 count based，通常来说 sstables 的大小应该是近似的，且会有一个 sstable size capacity，否则 count 就失去了意义。

time based 主要是考虑到冷热数据。在给 sstable assign ttl 时，不会考虑数据的冷热。因此不管一个 sstable 中存的冷数据多一些，还是热数据多一些，它们的 ttl 都是一样的。对于热数据而言，它会有更大的概率有新的版本，这些版本存在更新的 sstable 中。当存在新版本时，旧的版本实际就成为了冷数据。因此在 ttl 过期时，把包含这些旧版本热数据的 sstable compact 到更高层，对于热数据的读写更有利。对于包含冷数据更多的 sstable，在 ttl 过期时将其 compact 到更高层，则更有必要。

Compaction granularity

之前说过了，此处略。

File picking strategy

我总结这些 file picking strategies：

- random
- round-robin：假设 granularity 为 sstable。则会在 sstable view 下，即一个 level 中的所有 sorted runs 的 sstable flatten 之后的 view，维护一个 cursor。每次 compaction 选择这个 cursor 指向的这个 sstable。这个 sstable 可以作为该层唯一参与 compaction 的那个 sstable，也可以作为 seed，这个无所谓。
- 考虑到数据冷热：
    - sstable seek 次数：数据库会存储每个 sstable 的 seek 次数，即一次 query 查到这个 sstable 的次数，或最终在这个 sstable 查找到 target key 的次数。显然，seek 次数越大，说明这个 sstable 所包含的热数据越多，则它更应该被保留在低层级。相对地，那些 seek 次数少的就更应该被 compact 到更高层。
    - sstable timestamp：每个 sstable 有一个 timestamp。这个 timestamp 的分配是递增的，即较新的 sstable 的 timestamp 越大。在 file pick 时，选择 timestamp 最小的，即最老的

sstable。即使这个 sstable 中包含了很多热数据，但热数据之所以为热数据，就是因为它的版本更新更快，所以即使这个 sstable 被 compact 到了更高层，包含热数据新版本的 sstable 也依然在底层，所以这样的策略是可行的。

- sstable ttl：类似于 sstable timestamp，即 ttl 过期、且过期最久的那个 sstable 会被选中。
- tombstone density：tombstone 即 delete marker，tombstone density 即表示一个 sstable 中所包含的 delete marker 的比例。因为被 deleted keys 后续再被 query 的概率较小，所以 tombstone density 越大，更应该被 compact。