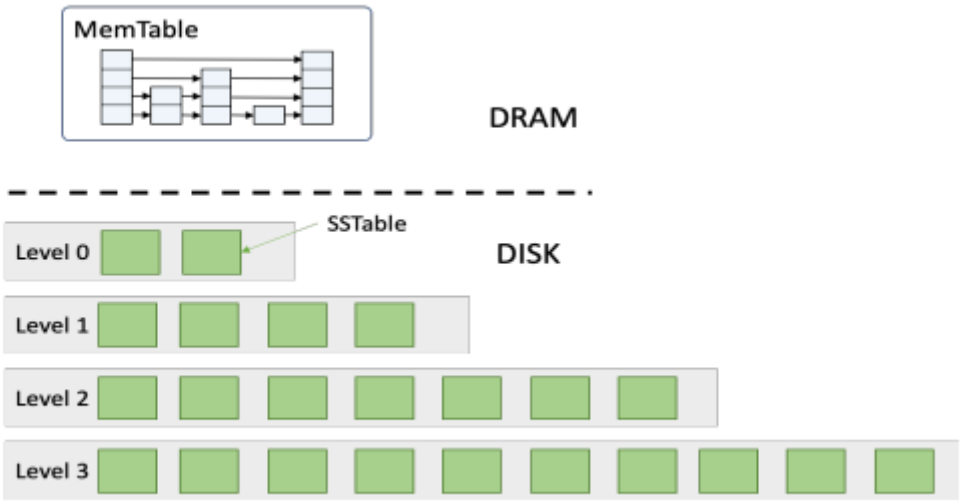
LSMTree（Log-Structured Merge-Tree）

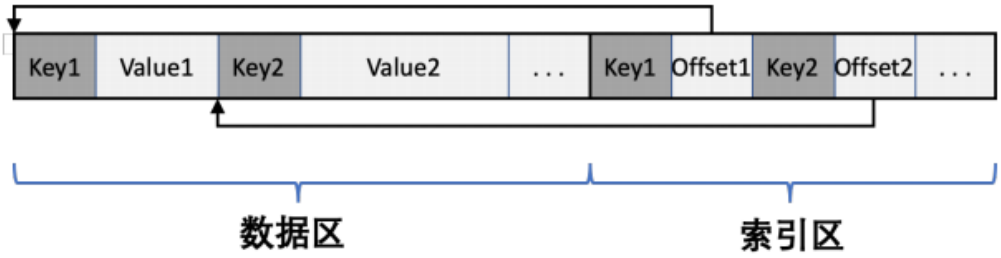
一、LSMTree 概要

1、结构介绍



LSMTree 是我在算法课上做的一个课程项目，这是一种专门为键值存储系统设计的数据结构，相较于 B+树，LSMTree 写效率更高。它由两个部分组成，一个是位于内存中的 Memory table，一个是位于硬盘中的分层存储。

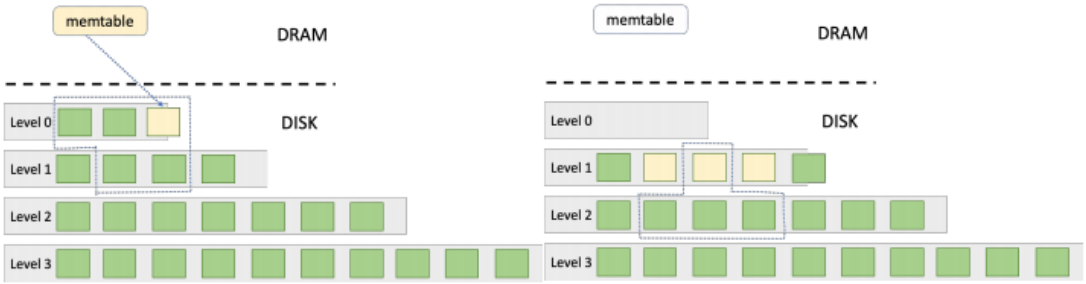
内存中的 Memory table 我是通过一个跳表来进行实现的，每次有新的数据插入或者删除的时候，我在 Memory table 中插入对应的键值对。当 Memory table 满了之后，我们进行一次对硬盘的写入操作，一个 Memory table 可以被写成一个 SSTable。



硬盘的存储分为很多层，每一层中包括多个文件，每个文件叫 SSTable（Sorted Strings Table），用于有序存储多个键值对。这里我把一个 SSTable 文件设置为了 2MB。我的 SSTable 在文件结构上分为数据区和索引区两部分。每个 SSTable 文件的前部分为数据区，用于存储有序的键值对数据。文件的后端部分为索引区，用来存储索引数据，索引区相对文件开头的偏移量和时间戳。索引区存储 key 和这个 key 相对于文件开头的偏移量，方便快速对键值对进行查找。

硬盘中的分层存储，比如说我们从 L0 开始，第 Ln 层的上限是 2 的 n+1 次方，也就是说 L0 最多有两个 SSTable，L1 最多有四个 SSTable 这样。

2、合并操作



如果 L0 中的文件数量超出限制，则开始进行合并操作。对于 L0 的合并操作来说，需要将所有的 L0 层中的所有 SSTable 与 L1 的中部分 SSTable 进行合并，随后将产生的新 SSTable 文件写入到 L1 中。

我先统计 L0 层中所有 SSTable 所覆盖的键的区间。然后在 L1 中找到与此区间有交集的所有 SSTable 。下面我用归并排序，把所有涉及到的 SSTable 进行合并，并将结果每 2MB 分成一个新的 SSTable，写入到 L1 中。合并时，我们还需要处理冗余数据、处理数据的删除操作。我这里的键值保留的是时间戳最新的记录，SSTable 最后的时间戳我取的是这个 SSTable 生成后的时间戳。

如果在 L0 跟 L1 合并之后，L1 也超过了文件上限，那么我把 L1 最后溢出的那个文件拿出来，用同样的方法跟下一层合并，如果没有下一层，我们就新建一层，直到文件数满足层数要求。

3、增删改查操作

增加：

对于 PUT 操作，我们只需要直接向 Memory Table 中直接插入就好了。如果在插入后 MemTable 数据大小超出 MemTable 限制，就把 MemTable 中的数据转换成 SSTable 保存在 Level 0 层中，如果 Level 0 层的文件数量超出限制，就开始进行合并操作。

删除：

这里为了提升性能，我使用了延迟（Lazy）的方法会处理删除操作。当进行删除时，我插入一条新的记录，表示键 K 被删除。在执行合并操作时，根据时间戳将相同键 K 的多个记录进行合并。

更改：

我们直接向 Memory Table 中增加一个新的键值对就好了，这里也比较像延迟（Lazy）的方法。由于后续的合并操作会把多个 key 相同的合并成一个。我们的查询也总是去查时间戳最新的文件中，最靠后的键值对，所以这么做是效率高且正确的。

查询：

对于查询 GET 操作，首先从 MemTable 中进行查找，当查找到键 K 所对应的记录之后结束。如果 MemTable 中不存在键 K，则从 Level 0 开始逐层进行查找，直到找到键值对记录，或找遍了所有层。在查找每个 SSTable 时，可直接使用内存中缓存的索引。如果要查询的键 K 在 SSTable 的键区间内，那么我们使用二分法进行查找，查到记录后，我们就可以从磁盘文件中读取相应的键值对内容。

二、LSMTree 和其他数据结构的比较（LSMTree 的优缺点）

1、优点

写效率高

LSMTree 是一种分层，有序，面向磁盘的数据结构，它充分了利用了磁盘批量的顺序写的效率，远比随机写效率高的这个性质，所以会有更好的性能（可认为是O(1)）。我们可以看到，数据的增删改全是对数据的追加，其实是不存在删除和修改的，真正的删除和修改是在合并的时候做的。

锁操作简单

因为我们不会去修改 SSTable，只有合并操作会修改 SSTable。在当有多用户对 LSMTree 进行操作的时候，相应的锁操作会更加简单。一般来说，唯一的需要竞争的资源就是 MemTable。

吞吐高

基于 LSMTree 的分层存储能够做到写的高吞吐，因为我们的增删改操作永远都是向 MemTable 中插入一个键值对，操作简单高效。

2、缺点

读性能差

LSMTree 虽然大大提升了数据的写入能力，但它的读取性能是比较差的（可认为是O(N)），所以 LSMTree 通常应用在读写多读少的场景。

为什么读性能差呢，因为当一个读操作开始时，我们会首先检查 MemTable，如果没有找到相应的 key，就会逐个检查 SSTable 文件，直到 key 被找到。虽然每个 SSTable 是有序的，查找会相对比较高效（O(logN)），但是随着文件个数增加，读操作会越来越慢，因为每一个 SSTable 都要被检查。（O(Klog N), K 为 SSTable个数， N 为 SSTable 平均大小）

合并操作消耗大

高吞吐的代价是，整个系统必须频繁的进行合并操作。写入量越大，合并操作越频繁。而合并操作是非常消耗性能的，在高吞吐的写入情形下，整个系统的性能会发生巨大下降。

3、比较

在数据的更新和删除方面，B+ 树可以做到原地更新和删除，且可以支持高效读操作（稳定的O(logN)），但是在大规模的写请求下（复杂度O(LogN)），效率会变得比较低，因为随着插入的操作，为了维护 B+ 树结构，节点会不断的分裂和合并。操作磁盘的随机读写概率会变大，故导致性能降低。

三、对于 LSMTree 的优化

1、增加布隆过滤器，优化读操作（没有实现）

谷歌当年发布的论文中有提到，正常情况下，一个读操作是需要读取所有的 SSTable 的。但如果我们对每一个 SSTable 在内存中增加一个布隆过滤器，那么就能很快地对数据进行判断。

由于布隆过滤器存在假阳性，如果判断一个 SSTable 存在某个 key，实际上是不一定存在的。但是如果布隆过滤器判断出不存在某个 key，由于布隆过滤器不存在假阴性，那么该数据就一定不会存在。利用这个特性，如果我们通过布隆过滤器判断出该数据不存在于这个 SSTable 中，我们就可以减少不必要的磁盘扫描，直接返回不存在就好了。

2、合并的时候采用多路归并算法

compact:小树合并为大树:因为小树他性能有问题，所以要有个进程不断地将小树合并到大树上，这样大部分的老数据查询也可以直接使用log2N的方式找到，不需要再进行(N/m)*log2n的查询了

3、可以不用立刻合并 SSTable，我们定期合并 SSTable（没有是实现且不确定，但是很好奇，阿里据说有采用这种策略）

通过定期合并， 可以有效的清除无效数据，缩短读取路径，提高磁盘利用空间。但合并操作是非常消耗 CPU 和磁盘 IO 的，在业务高峰期，会降低整个系统的吞吐量，这可能是我们很难接受的。我们可以暂时禁用合并功能，并在业务低峰期的时候再进行合并。

四、可能会问的问题

1、LSMTree 的优势在哪里，为什么使用 LSMTree，而不是 B+ 树?

LSMTree 和 B+ 树各有优劣。

在数据的更新和删除方面，B+ 树可以做到原地更新和删除，且可以支持高效读操作（稳定的O(logN)），但是在大规模的写请求下（复杂度O(LogN)），效率会变得比较低，因为随着插入的操作，为了维护 B+ 树结构，节点会不断的分裂和合并。操作磁盘的随机读写概率会变大，故导致性能降低。

而对于 LSMTree 来说，它充分了利用了磁盘批量的顺序写的效率，远比随机写效率高的这个性质，所以会有更好的性能（可认为是O(1)）。我们可以看到，数据的增删改全是对数据的追加，其实是不存在删除和修改的，真正的删除和修改是在合并的时候做的。

总的来说，LSMTree 虽然大大提升了数据的写入能力，但它的读取性能是比较差的，所以 LSMTree 通常应用在读写多读少的场景。

而在数据的更新和删除方面，B+ 树可以做到原地更新和删除，且支持高效读操作（稳定的O(logN)），所以通常应用在读多写少的场景。

2、平衡二叉树（AVL树）、B 树（B- 树）、B+ 树、B* 树简介

平衡二叉树

B 树（B- 树）

B+ 树

B* 树

3、为什么 MemTable 使用跳表，而不是其他数据结构?

当然我们可以在 MemTable 中使用其他数据结构，比如说一个链表、一个vector、一个平衡树。这里选择跳表，是因为跳表是一种可以取代平衡树的数据结构。跳表使用概率均衡，不是严格均衡策略，从而比起平衡树来说，简化插入和删除，提高了效率，当然代价是跳表会占用更多的空间，空间复杂度为O（n）。

4、为什么一个 SSTable 是 2MB

这其实是一个可以调整的参数。由于磁盘每次读写都是对一个磁盘块进行操作的，为了保证我们每次读写的效率，SSTable 的大小最好是磁盘块大小 4KB 的整数倍。SSTable 不应过大，至少应当小于内存大小的一半（因为内存里维护的跳表是 O(n) 的），过大的话也会导致读一个文件的时间开销过大。但是 SSTable 也不应当太小，否则会导致 LevelDB 层数很深，从而要进行更多的合并操作。所以这里我选择了 2MB 这样一个合适的数值。

5、为什么文件的后半部分是索引区，而不是前面

6、LSMTree 的合并过程中如何处理多个时间戳记录的

7、简单介绍一下布隆过滤器

8、简单介绍一下多路归并算法