日志结构存储

DB存储结构有两大流派，一是以B树为代表的可变数据结构。另外一种是以日志结构合并树为代表的不可变数据结构。

前者对读性能进行了优化：

当在磁盘上定位数据之后，就可以把记录返回给客户端。

但是牺牲了写性能：

更新数据时候，需要在**磁盘上定位数据记录，并且在文件的原始偏移量上更新页。**

后者对写性能进行了优化：

更新或者插入数据则采用append-only，写操作是不需要在磁盘上找到记录来覆盖他们。

但是牺牲了读性能：

读取必须检索多个数据记录版本并对其进行协调。

B树在读写和维护期间大多数IO操作都是随机的，每个写操作首先要找到保存数据记录的页才能做修改，而且即使是修改了页中的一个单元格，也必须要重写整个页。LSM树的思想就是让IO操作顺序化，并且避免修改期间的页重写。

LSM 树

直觉告诉我们，使用Cache可以帮助我们改善空间开销和写入放大问题。通常在不同的存储结构中有两种应用cache的方式:

- 推迟对磁盘驻留页的写入，可以理解为先写入到内存，再积攒成batch落盘
- 使得写操作顺序化

所以你可以把LSM树理解为B树的一种变体，但是其节点是完全被填满的，并且使用缓冲和append-only存储来实现顺序写操作，优化写性能。

LSM树写入的是不可变文件，并会随着时间推移将其合并在一起。这些文件通常包含自己的索引，用于帮助读取者高效定位数据。

而LSM的M表示merge，意思是在回收冗余副本所占空间的维护期或者是向用户返回内容之前的读取期，都会用merge sort来合并树的内容。

LSM树会把数据文件写入推迟，并且把更改缓冲在一个内存驻留表中。表中内容随后会被写到不可变磁盘中。在文件持久化之前，所有记录都可以在内存中访问。

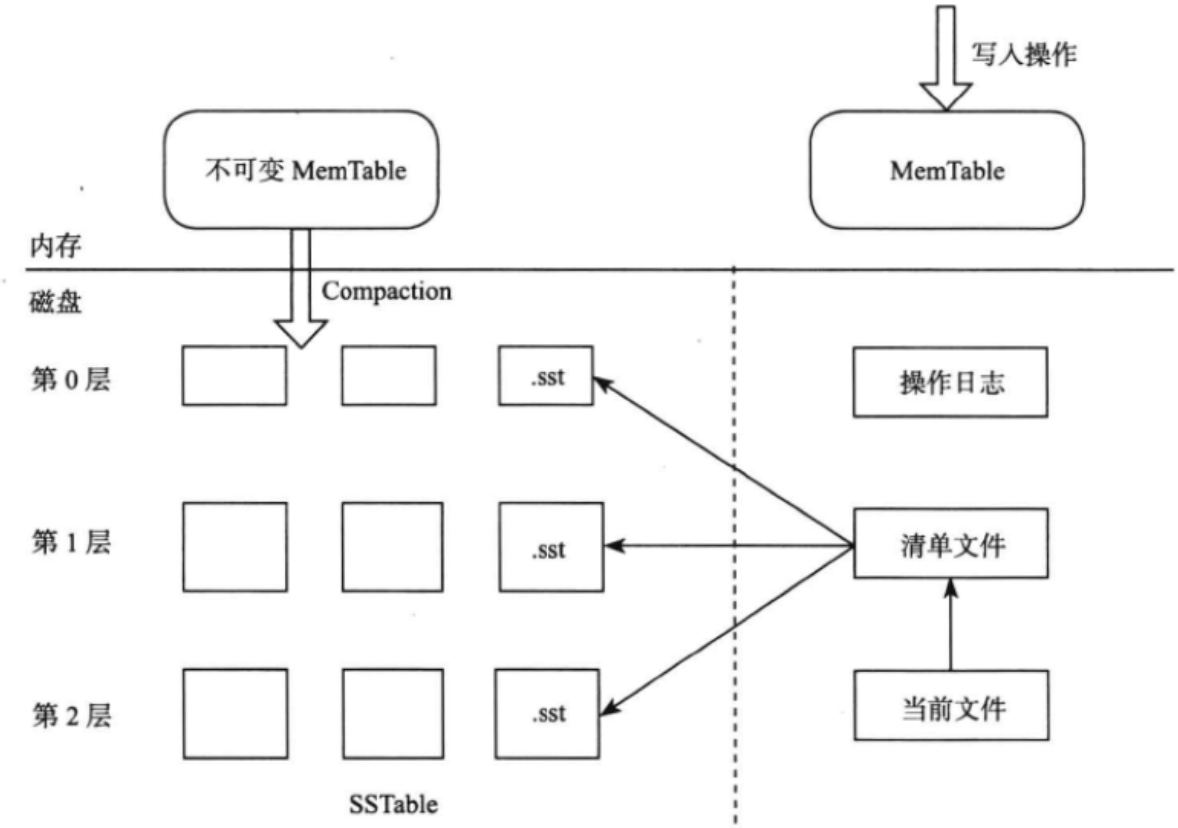
为什么说不可变数据结构利于顺序写入呢?

- 数据一次性写入磁盘，文件是append-only的。可变结构也可以一次性预分配数据块，但后续的访问还是随机读写。

另外顺序append-only写入可以防止产生碎片，文件数据密度更高，不需要为写入的记录保留额外空间，也不会因为更新后的记录可能比最初写入的记录需要更多空间而为其保留额外的空间

而且由于文件不可变，CRUD操作不需要在磁盘上定位数据记录，显著提高了写入的性能和吞吐量。另一方面由于重复内容是允许的，而且要在读取的时候解决冲突。

所以LSM树适用于**写远大于读的场景**



LSM树的结构

memtable是可变的，其缓冲数据记录，并且充当写操作的目标。当大小到达一个阈值时候会转换成不可变Memtable，然后会被持久化到磁盘上。

可变memtable的更新不需要磁盘访问，也没有IO开销，需要一个单独的pre-write log用来保证数据记录的持久性。

LSM的缓冲是在内存中完成的所有读写操作都会应用于一个内存驻留表，这个表维护一个允许并发访问的有序数据结构，其通常是某种形式的内存排序树。

sstable则是通过把内存中的缓冲内容flush到磁盘来构建的，sstable仅用于读取：缓存中的内容会被持久化成文件，并且这些文件永远不会被修改。

也就是说：对内存中的表进行写操作，对磁盘和基于内存的表进行读和compaction，delete操作

memtable flush 到磁盘的操作可以周期性触发，也可以通过大小阈值触发。在flush之前，必须进行memtable的切换：**分配一个新的memtable称为写入操作的新目标，同时旧的memtable变成刷写状态** 这两个步骤必须原子进行。

在内容完全刷写之前，被刷写的memtable仍然可用于读取。然后旧的memtable会被丢弃，取而代之的是一个新的sstable

当memtable完全落盘后，其pre-write log是可以被修建的。与落盘相关的memtable的操作都可以丢弃掉。

更新和删除

删除操作必须显式记录下来，常见实现是可以插入一个特殊的删除条目，也叫作墓碑(tomestone)

因为可能磁盘和内存中分别维护了k1:v1,k2:v2，如果你只是删掉了memtable的k2:v2（磁盘的sstable不可变）这就复活了k1:v1，因为k1:v1现在是k1唯一的关联。

compaction的两个策略：

1.分层compaction

2.按大小compaction

读写放大和空间放大

- 回收重复记录占用的空间->增加了写放大，因为要不断重写表
- 避免连续重写数据->增加了读放大，读取期间协调关联多个key-value，同时也会有空间放大，因为冗余记录会被保存更长时间

  所以以不可变方式在磁盘上存储会遇到三个问题：

  1.读放大

  为了检索数据而需要读取多个表

  2.写放大

  compaction过程中不断重写

  3.空间放大

  存储关联到同一个key的多个value记录所引起

实现细节

SSTable

Sorted String Table

SSTtable数据是按照key顺序进行排序和布局的，SSTable由两个组件组成：索引文件和数据文件。

索引文件通常以对数时间复杂度或者常量时间复杂度结构实现。

Bloom Filter

LSM的读放大来源：必须寻址多个SSTable，因为我们不知道一个SSTable是否一定包含要搜索的key所指向的记录

Bloom Filter使用一个很大的bit array和多个hash function构建，详细原理就google好了，我这里不赘述。

总的来说，对于每个sstable维护一个bloom filter，那么在读取sstable之前先通过bloom filter来快速判断你要搜索的key-value是否存在于这个sstable中。

假阳性的概率取决于**size of bit array**以及**number of hash function**

当然如果bit array越大，占用内存也会越大(bit array是存在内存里的)

而hash function越多，也会对性能造成负面影响，所以这就是一个trade-off的过程

skipList

多用于memtable中二级索引实现，具体skipList原理自己google

磁盘访问

LSM把Page Cache用于磁盘访问和中间缓存。

而且由于读取不可变memtable时候，是不需要额外的锁来控制并发控制。引用计数可以确保当前访问的Page不会被内存中换出，并且保证在compaction过程中发出的请求在底层文件被移除之前均已经完成。