17 | 存储系统：从检索技术角度剖析LevelDB的架构设计思想

2020-05-08 陈东

检索技术核心20讲 进入课程 >

讲述：陈东
时长 20:25　大小 18.71M

你好，我是陈东。

LevelDB 是由 Google 开源的存储系统的代表，在工业界中被广泛地使用。它的性能非常突出，官方公布的 LevelDB 的随机读性能可以达到 6 万条记录 / 秒。那这是怎么做到的呢？这就和 LevelDB 的具体设计和实现有关了。

LevelDB 是基于 LSM 树优化而来的存储系统。都做了哪些优化呢？我们知道，LSM 树会将索引分为内存和磁盘两部分，并在内存达到阈值时启动树合并。但是，这里面存在着一些的细节问题。比如说，数据在内存中如何高效检索？数据是如何高效地从内存转移到磁盘的？以及我们如何在磁盘中对数据进行组织管理？还有数据是如何从磁盘中高效地检索出来的？

其实，这些问题也是很有代表性的工业级系统的实现问题。LevelDB 针对这些问题，使用了大量的检索技术进行优化设计。今天，我们就一起来看看，LevelDB 究竟是怎么优化检索系统，提高效率的。

如何利用读写分离设计将内存数据高效存储到磁盘？

首先，对内存中索引的高效检索，我们可以用很多检索技术，如红黑树、跳表等，这些数据结构会比 B+ 树更高效。因此，LevelDB 对于 LSM 树的第一个改进，就是使用跳表代替 B+ 树来实现内存中的 C0 树。

好，解决了第一个问题。那接下来的问题就是，内存数据要如何高效存储到磁盘。在第 7 讲中我们说过，我们是将内存中的 C0 树和磁盘上的 C1 树归并来存储的。但如果内存中的数据一边被写入修改，一边被写入磁盘，我们在归并的时候就会遇到数据的一致性管理问题。一般来说，这种情况是需要进行“加锁”处理的，但“加锁”处理又会大幅度降低检索效率。

为此，LevelDB 做了读写分离的设计。它将内存中的数据分为两块，一块叫作 **MemTable**，它是可读可写的。另一块叫作 **Immutable MemTable**，它是只读的。这两块数据的数据结构完全一样，都是跳表。那它们是怎么应用的呢？

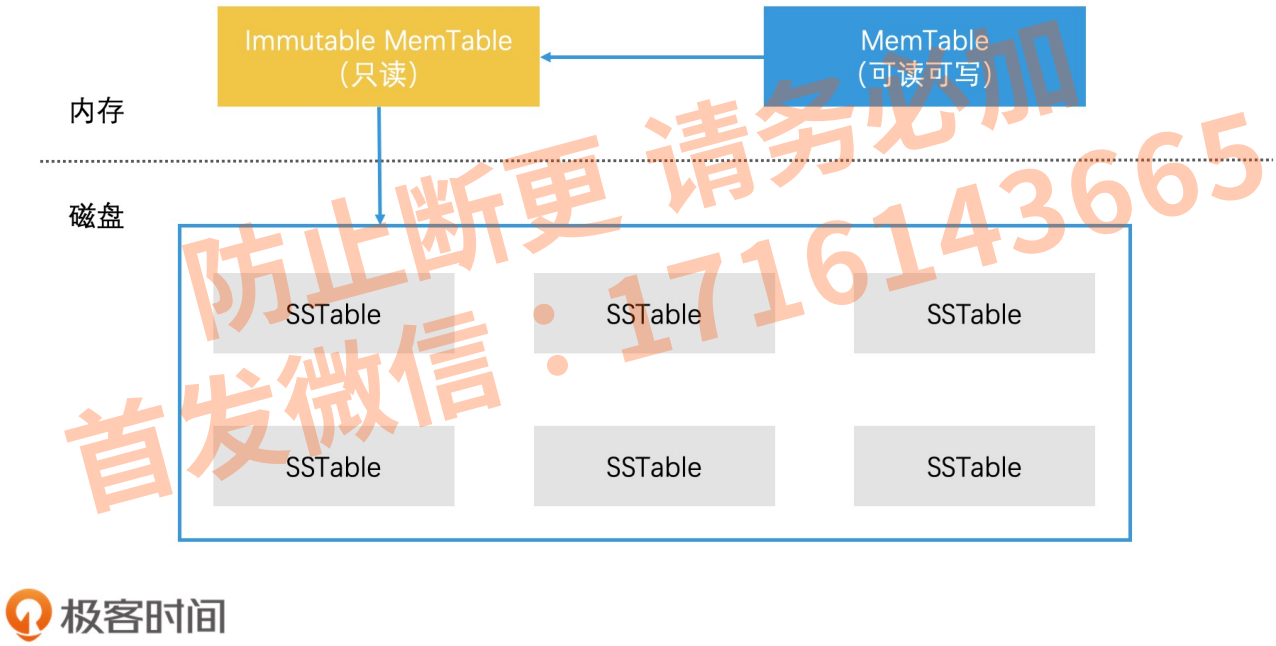
具体来说就是，当 MemTable 的存储数据达到上限时，我们直接将它切换为只读的 Immutable MemTable，然后重新生成一个新的 MemTable，来支持新数据的写入和查询。这时，将内存索引存储到磁盘的问题，就变成了将 Immutable MemTable 写入磁盘的问题。而且，由于 Immutable MemTable 是只读的，因此，它不需要加锁就可以高效地写入磁盘中。

好了，数据的一致性管理问题解决了，我们接着看 C0 树和 C1 树的归并。在原始 LSM 树的设计中，内存索引写入磁盘时是直接和磁盘中的 C1 树进行归并的。但如果工程中也这么实现的话，会有两个很严重的问题：

1. 合并代价很高，因为 C1 树很大，而 C0 树很小，这会导致它们在合并时产生大量的磁盘 IO；
2. 合并频率会很频繁，由于 C0 树很小，很容易被写满，因此系统会频繁进行 C0 树和 C1 树的合并，这样频繁合并会带来的大量磁盘 IO，这更是系统无法承受的。

那针对这两个问题，LevelDB 采用了延迟合并的设计来优化。具体来说就是，先将 Immutable MemTable 顺序快速写入磁盘，直接变成一个个 **SSTable**（Sorted String Table）文件，之后再对这些 SSTable 文件进行合并。这样就避免了 C0 树和 C1 树昂贵的合并代价。至于 SSTable 文件是什么，以及多个 SSTable 文件怎么合并，我们一会儿再详细分析。

好了，现在你已经知道了，内存数据高效存储到磁盘上的具体方案了。那在这种方案下，数据又是如何检索的呢？在检索一个数据的时候，我们会先在 MemTable 中查找，如果查找不到再去 Immutable MemTable 中查找。如果 Immutable MemTable 也查询不到，我们才会到磁盘中去查找。



极客时间

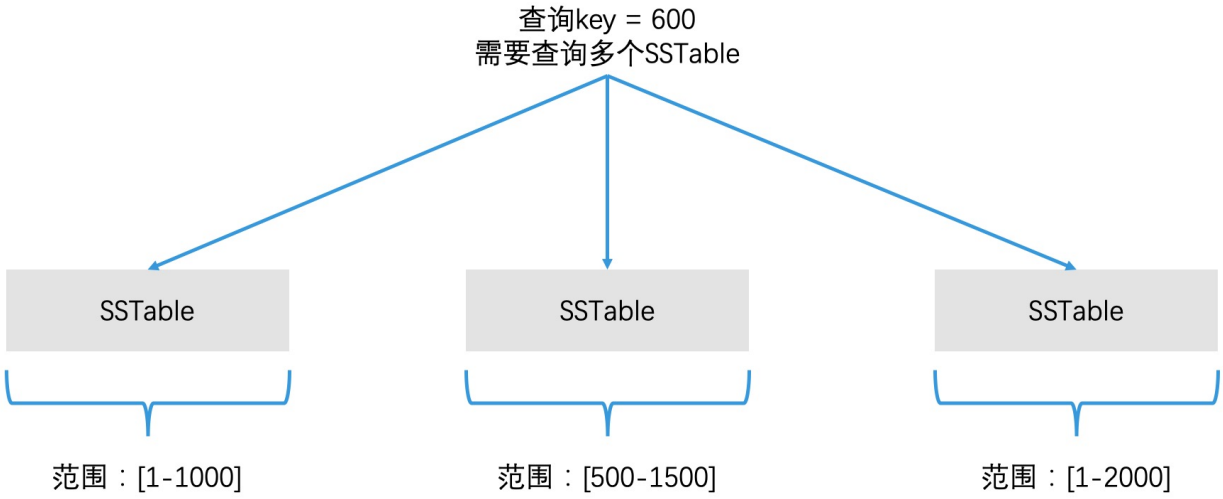
增加Immutable MemTable设计的示意图

因为磁盘中原有的 C1 树被多个较小的 SSTable 文件代替了。那现在我们要解决的问题就变成了，如何快速提高磁盘中多个 SSTable 文件的检索效率。

SSTable 的分层管理设计

我们知道，SSTable 文件是由 Immutable MemTable 将数据顺序导入生成的。尽管 SSTable 中的数据是有序的，但是每个 SSTable 覆盖的数据范围都是没有规律的，所以 SSTable 之间的数据很可能有重叠。

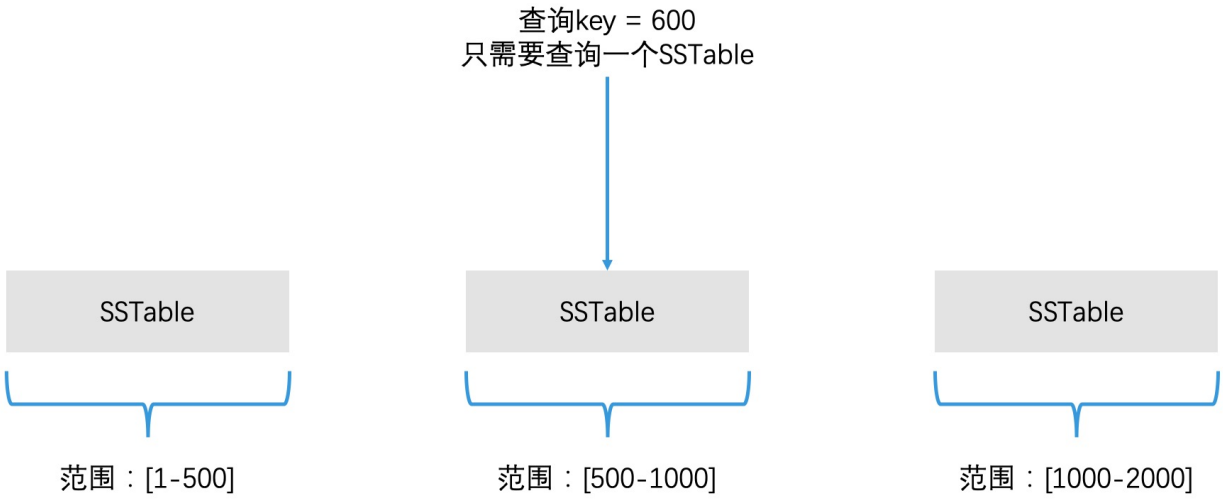
比如说，第一个 SSTable 中的数据从 1 到 1000，第二个 SSTable 中的数据从 500 到 1500。那么当我们要查询 600 这个数据时，我们并不清楚应该在第一个 SSTable 中查找，还是在第二个 SSTable 中查找。最差的情况是，我们需要查询每一个 SSTable，这会带来非常巨大的磁盘访问开销。



极客时间

范围重叠时，查询多个SSTable的示意图

因此，对于 SSTable 文件，我们需要将它整理一下，将 SSTable 文件中存的数据进行重新划分，让每个 SSTable 的覆盖范围不重叠。这样我们就能将 SSTable 按照覆盖范围来排序了。并且，由于每个 SSTable 覆盖范围不重叠，当我们需要查找数据的时候，我们只需要通过二分查找的方式，找到对应的一个 SSTable 文件，就可以在这个 SSTable 中完成查询了。



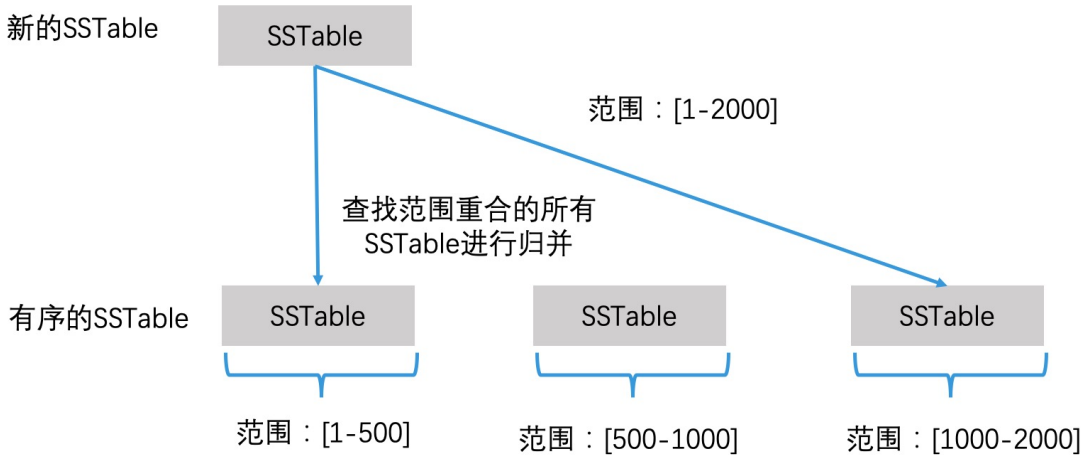
极客时间

范围不重叠时，只需查询一个SSTable的示意图

但是要让所有 SSTable 文件的覆盖范围不重叠，不是一个很简单的事情。为什么这么说呢？我们看一下这个处理过程。系统在最开始时，只会生成一个 SSTable 文件，这时候我们不需要进行任何处理，当系统生成第二个 SSTable 的时候，为了保证覆盖范围不重合，我们需要将这两个 SSTable 用多路归并的方式处理，生成新的 SSTable 文件。

那为了方便查询，我们要保证每个 SSTable 文件不要太大。因此，LevelDB 还控制了每个 SSTable 文件的容量上限（不超过 2M）。这样一来，两个 SSTable 合并就会生成 1 个到 2 个新的 SSTable。

这时，新的 SSTable 文件之间的覆盖范围就不重合了。当系统再新增一个 SSTable 时，我们还用之前的处理方式，来计算这个新的 SSTable 的覆盖范围，然后和已经排好序的 SSTable 比较，找出覆盖范围有重合的所有 SSTable 进行多路归并。这种多个 SSTable 进行多路归并，生成新的多个 SSTable 的过程，也叫作 Compaction。



极客时间

SSTable保持有序的多路归并过程

随着 SSTable 文件的增多，多路归并的对象也会增多。那么，最差的情况会是什么呢？最差的情况是所有的 SSTable 都要进行多路归并。这几乎是一个不可能被接受的时间消耗，系统的读写性能都会受到很严重的影响。

那我们该怎么降低多路归并涉及的 SSTable 个数呢？在第 9 讲中，我们提到过，对于少量索引数据和大规模索引数据的合并，我们可以采用滚动合并法来避免大量数据的无效复制。因此，LevelDB 也采用了这个方法，将 SSTable 进行分层管理，然后逐层滚动合并。这就是 LevelDB 的分层思想，也是 LevelDB 的命名原因。接下来，我们就一起来看看 LevelDB 具体是怎么设计的。

首先，**从 Immutable MemTable 转成的 SSTable 会被放在 Level 0 层。**Level 0 层最多可以放 4 个 SSTable 文件。当 Level 0 层满了以后，我们就要将它们进行多路归并，生成新的有序的多个 SSTable 文件，这一层有序的 SSTable 文件就是 Level 1 层。

接下来，如果 Level 0 层又存入了新的 4 个 SSTable 文件，那么就需要和 Level 1 层中相关的 SSTable 进行多路归并了。但前面我们也分析过，如果 Level 1 中的 SSTable 数量很多，那么在大规模的文件合并时，磁盘 IO 代价会非常大。因此，LevelDB 的解决方案就是，**给 Level 1 中的 SSTable 文件的总容量设定一个上限**（默认设置为 10M），这样多路归并时就有了一个代价上限。

当 Level 1 层的 SSTable 文件总容量达到了上限之后，我们就需要选择一个 SSTable 的文件，将它并入下一层（为保证一层中每个 SSTable 文件都有机会并入下一层，我们选择 SSTable 文件的逻辑是轮流选择。也就是说第一次我们选择了文件 A，下一次就选择文件 A 后的一个文件）。**下一层会将容量上限翻 10 倍**，这样就能容纳更多的 SSTable 了。依此类推，如果下一层也存满了，我们就在该层中选择一个 SSTable，继续并入下一层。这就是 LevelDB 的分层设计了。

极客时间

LevelDB的层次结构示意图

尽管 LevelDB 通过限制每层的文件总容量大小，能保证做多路归并时，会有一个开销上限。但是层数越大，容量上限就越大，那发生在下层的多路归并依然会造成大量的磁盘 IO 开销。这该怎么办呢？

对于这个问题，LevelDB 是通过加入一个限制条件解决的。在多路归并生成第 n 层的 SSTable 文件时，LevelDB 会判断生成的 SSTable 和第 n+1 层的重合覆盖度，如果重合覆盖度超过了 10 个文件，就结束这个 SSTable 的生成，继续生成下一个 SSTable 文件。

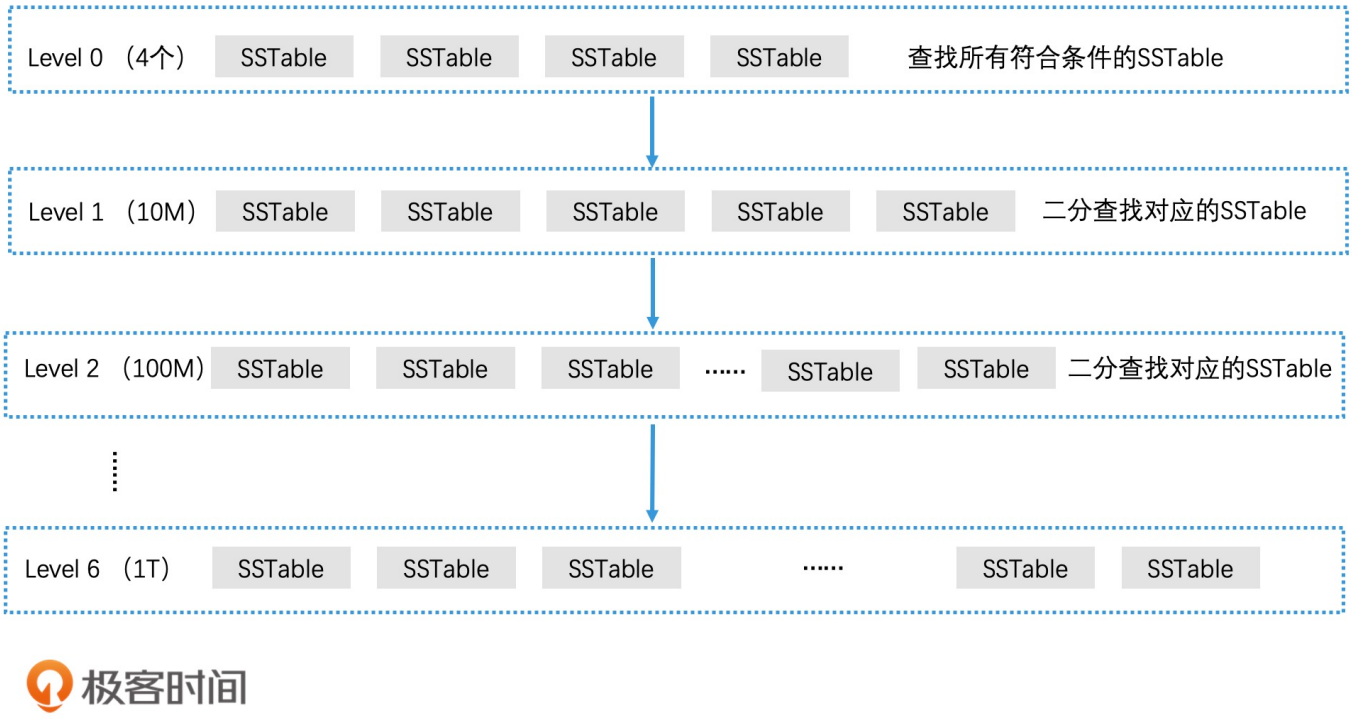
通过这个限制，**LevelDB 就保证了第 n 层的任何一个 SSTable 要和第 n+1 层做多路归并时，最多不会有超过 10 个 SSTable 参与**，从而保证了归并性能。

如何查找对应的 SSTable 文件

在理解了这样的架构之后，我们再来看看当我们想在磁盘中查找一个元素时，具体是怎么操作的。

首先，我们会在 Level 0 层中进行查找。由于 Level 0 层的 SSTable 没有做过多路归并处理，它们的覆盖范围是有重合的。因此，我们需要检查 Level 0 层中所有符合条件的 SSTable，在其中查找对应的元素。如果 Level 0 没有查到，那么就下沉一层继续查找。

而从 Level 1 开始，每一层的 SSTable 都做过了处理，这能保证覆盖范围不重合的。因此，对于同一层中的 SSTable，我们可以使用二分查找算法快速定位唯一的一个 SSTable 文件。如果查到了，就返回对应的 SSTable 文件；如果没有查到，就继续沉入下一层，直到查到了或查询结束。



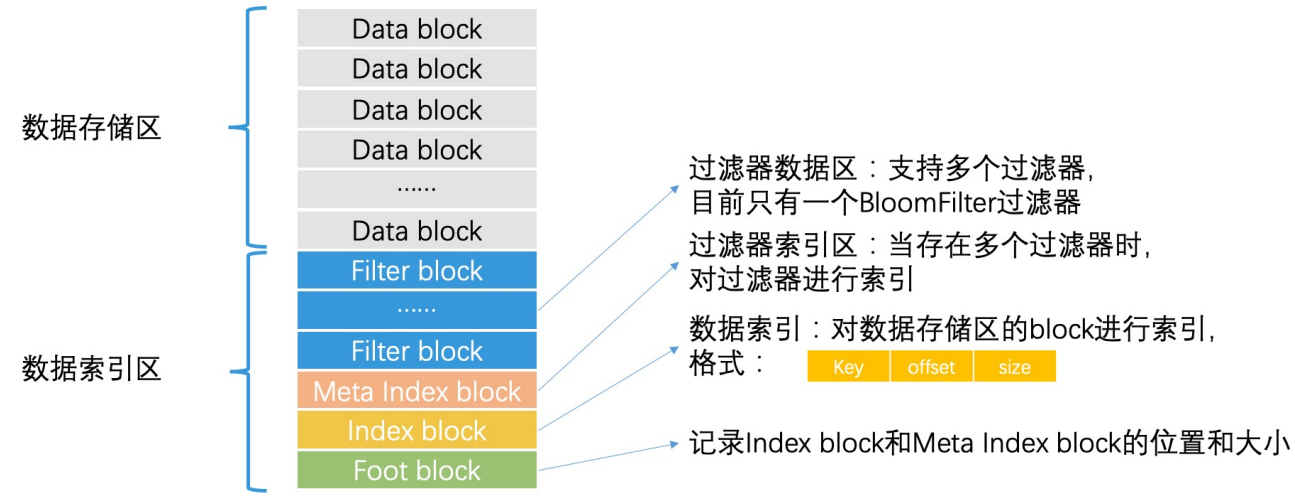
LevelDB分层检索过程示意图

可以看到，通过这样的一种架构设计，我们就将 SSTable 进行了有序的管理，使得查询操作可以快速被限定在有限的 SSTable 中，从而达到了加速检索的目的。

SSTable 文件中的检索加速

那在定位到了对应的 SSTable 文件后，接下来我们该怎么查询指定的元素呢？这个时候，前面我们学过的一些检索技术，现在就可以派上用场了。

首先，LevelDB 使用索引与数据分离的设计思想，将 SSTable 分为数据存储区和数据索引区两大部分。



SSTable文件格式

我们在读取 SSTable 文件时，不需要将整个 SSTable 文件全部读入内存，只需要先将数据索引区中的相关数据读入内存就可以了。这样就能大幅减少磁盘 IO 次数。

然后，我们需要快速确定这个 SSTable 是否包含查询的元素。对于这种是否存在的状态查询，我们可以使用前面讲过的 BloomFilter 技术进行高效检索。也就是说，我们可以从数据索引区中读出 BloomFilter 的数据。这样，我们就可以使用 O(1) 的时间代价在 BloomFilter 中查询。如果查询结果是不存在，我们就跳过这个 SSTable 文件。而如果 BloomFilter 中查询的结果是存在，我们就继续进行精确查找。

在进行精确查找时，我们将数据索引区中的 Index Block 读出，Index Block 中的每条记录都记录了每个 Data Block 的最小分隔 key、起始位置，还有 block 的大小。由于所有的记录都是根据 Key 排好序的，因此，我们可以使用二分查找算法，在 Index Block 中找到我们想查询的 Key。

那最后一步，就是将这个 Key 对应的 Data block 从 SSTable 文件中读出来，这样我们就完成了数据的查找和读取。

利用缓存加速检索 SSTable 文件的过程

在加速检索 SSTable 文件的过程中，你会发现，每次对 SSTable 进行二分查找时，我们都需要将 Index Block 和相应的 Data Block 分别从磁盘读入内存，这样就会造成两次磁盘 I/O 操作。我们知道磁盘 I/O 操作在性能上，和内存相比是非常慢的，这也会影响数据的检索速度。那这个环节我们该如何优化呢？常见的一种解决方案就是使用缓存。LevelDB 具体是怎么做的呢？

针对这两次读磁盘操作，LevelDB 分别设计了 table cache 和 block cache 两个缓存。其中，block cache 是配置可选的，它是将最近使用的 Data Block 加载在内存中。而 table cache 则是将最近使用的 SSTable 的 Index Block 加载在内存中。这两个缓存都使用 LRU 机制进行替换管理。

那么，当我们想读取一个 SSTable 的 Index Block 时，首先要去 table cache 中查找。如果查到了，就可以避免一次磁盘操作，从而提高检索效率。同理，如果接下来要读取对应的 Data Block 数据，那么我们也先去 block cache 中查找。如果未命中，我们才会去真正读磁盘。

这样一来，我们就可以省去非常耗时的 I/O 操作，从而加速相关的检索操作了。

重点回顾

好了，今天我们学习了 LevelDB 提升检索效率的优化方案。下面，我带你总结回顾一下今天的重点内容。

首先，在内存中检索数据的环节，LevelDB 使用跳表代替 B+ 树，提高了内存检索效率。

其次，在将数据从内存写入磁盘的环节，LevelDB 先是使用了**读写分离**的设计，增加了一个只读的 Immutable MemTable 结构，避免了给内存索引加锁。然后，LevelDB 又采用了**延迟合并**设计来优化归并。具体来说就是，它先快速将 C0 树落盘生成 SSTable 文件，再使用其他异步进程对这些 SSTable 文件合并处理。

而在管理多个 SSTable 文件的环节，LevelDB 使用**分层和滚动合并**的设计来组织多个 SSTable 文件，避免了 C0 树和 C1 树的合并带来的大量数据被复制的问题。

最后，在磁盘中检索数据的环节，因为 SSTable 文件是有序的，所以我们通过**多层二分查找**的方式，就能快速定位到需要查询的 SSTable 文件。接着，在 SSTable 文件内查找元素时，LevelDB 先是使用**索引与数据分离**的设计，减少磁盘 IO，又使用 **BloomFilter 和二分查找**来完成检索加速。加速检索的过程中，LevelDB 又使用**缓存技术**，将会被反复读取的数据缓存在内存中，从而避免了磁盘开销。

总的来说，一个高性能的系统会综合使用多种检索技术。而 LevelDB 的实现，就可以看作是我们之前学过的各种检索技术的落地实践。因此，这一节的内容，我建议你多看几遍，这对我们之后的学习也会有非常大的帮助。

课堂讨论

1. 当我们查询一个 key 时，为什么在某一层的 SSTable 中查到了以后，就可以直接返回，不用再去下一层查找了呢？如果下一层也有 SSTable 存储了这个 key 呢？
2. 为什么从 Level 1 层开始，我们是限制 SSTable 的总容量大小，而不是像在 Level 0 层一样限制 SSTable 的数量？ （提示：SSTable 的生成过程会受到约束，无法保证每一个 SSTable 文件的大小）

欢迎在留言区畅所欲言，说出你的思考过程和最终答案。如果有收获，也欢迎把这一讲分享给你的朋友。

上一篇 测一测 | 高性能检索系统的实战知识，你掌握了多少?

下一篇 18 | 搜索引擎：输入搜索词以后，搜索引擎是怎么工作的?

精选留言 (13)

写留言

一步

2020-05-08

当 MemTable 的存储数据达到上限时，我们直接将它切换为只读的 Immutable MemTable，然后重新生成一个新的 MemTable

---

这样的一个机制，内存中会出现多个Immutable MemTable 吗？ 上一个Immutable MemTable 没有及时写入到磁盘

展开

作者回复: 这是一个好问题！实际上，这也是levelDB的一个瓶颈。当immutable memtable还没有完全写入磁盘时，memtable如果写满了，就会被阻塞住。

因此，Facebook基于Google的levelDB，开源了一个rocksDB，rocksDB允许创建多个memtable，这样就解决了由于写入磁盘速度太慢导致memtable阻塞的问题。

3

吴小智
2020-05-08

之前看过基于 lsm 的存储系统的代码，能很好理解这篇文章。不过，还是不太理解基于 B + 树与基于 lsm 的存储系统，两者的优缺点和使用场景有何不同，老师有时间可以解答一下。

作者回复: lsm树和b+树会有许多不同的特点。但是如果从使用场景来看，最大的区别就是看读和写的需求。
在随机读很多，但是写入很少的场合，适合使用b+树。因为b+树能快速二分找到任何数据，并且磁盘io很少；但如果是使用lsm树，对于大量的随机读，它无法在内存中命中，因此会去读磁盘，并且是一层一层地多次读磁盘，会带来很严重的读放大效应。
但如果是大量的写操作的场景的话，lsm树进行了大量的批量写操作优化，因此效率会比b+树高许多。b+树每次写入都要去修改叶子节点，这会带来大量的磁盘io，使得效率急剧下降。这也是为什么日志系统，监控系统这类大量生成写入数据的应用会采用lsm树的原因。

2 2

一步
2020-05-09

在评论下回复老师看不到啊，那就在评论问一下

第一问还有点疑问:level0层的每个sstable可能会有范围重叠，需要把重叠的部分提取到合并列表，这个这个合并列表是什么？还有就是提取之后呢，还是要遍及level0层的每个sstable与level1层的sstable进行归并吗？...
展开

作者回复: 1.合并列表其实就是记录需要合并的sstable的列表。实际上，每次合并时，系统都会生成两个合并列表。
以你提问的level 0层的情况为例，先选定一个要合并的sstable，然后将level 0层中和它范围重叠的sstable都加入到这个列表中;这就是合并列表1。
然后，对于合并列表1中所有的sstable，我们能找到整体的范围begin和end。那么我们在下一层中，将和begin和end范围重叠的所有sstable文件加入合并列表2。
那么，对于合并列表1和合并列表2中的所有的sstable，我们将它们一起做一次多路归并就可以了。
2.如果下层空间满了，没关系，先合并完，这时候，下层空间就超容量了。那么，我们再针对这一层，按之前介绍的规则，选择一个sstable再和下层合并即可。

PS:再补充一下知识点:合并的触发条件。
系统会统计每个level的文件容量是否超过限制。超过上限比例最大的，将会被触发合并操作。

1 1

一步
2020-05-09

LevelDB 分层的逻辑没有理解
当 Level0 层 有四个 SSTable 的时候，这时候把这个四个进行归并，然后放到 Level1 层，这时候 Level0 层清空，这个有个问题是 当进行归并后 后生成几个 SSTable ,这里是有什么规则吗?
...
展开

作者回复: 你的问题我重新整理一下，尤其是level 0层怎么处理，这其实是一个很好的问题:
问题1:level 0层到level 1层合并的时候，level 0层是有多少个sstable参与合并?
回答:按道理来说，我们应该是根据轮流选择的策略，选择一个level 0层的sstable进行和下层的合并，但是由于level 0层中的sstable可能范围是重叠的，因此我们需要检查每一个sstable，将有重叠部分的都加入到合并列表中。
问题2:level n层中的一个sstable要和level n+1层中的所有sstable进行合并么?
回答:不需要。如果level n层的sstable的最大最小值是begin和end，我们只需要在level n+1层中，找到可能包含begin到end之间的sstable即可。这个数量不会超过10个。因此不会带来太大的io。
问题3:为什么level n层的sstable和level n+1层的合并，个数不会超过10个?
回答:在level n层的sstable生成的时候，我们会开始判断这个sstable和level n+1层的哪些sstable有重叠。如果发现重叠个数达到十个，就要结束这个sstable文件的生成。
举个例子，如果level n+1层的11个sstable的第一个元素分别是[100，200，300，400，……，1000，1100]，即开头都是100的整数倍。那么，如果level n层的sstable文件生成时，准备写入的数据就是[100，200，300，400，……，1000，1100]，那么在要写入1100的时候，系统会发现，如果写入1100，那么这个sstable文件就会和下一层的11个sstable文件有重叠了，会违反规则，因此，这时候会结束这个sstable，也就是说，这个sstable文件中只有100到1000十个数。然后1100会被写入到一个新的sstable中。

1 1

牛牛
2020-05-09

老师、我想请教下、levelDB是怎么处理`脏缓存`(eg. 有用户突然访问了别人很久不访问的数据(假设还比较大)、导致本来应该在缓存中的数据被驱逐, Data Block的优化效果就会打折扣)的 ?

------...

展开

作者回复: 首先，levelDB并没有“脏缓存”的问题。因为lsm树和b+树不一样。
b+树的缓存对应着磁盘上的叶子节点，叶子节点是可以被修改的，因此会出现缓存在内存中的数据被修改，但是磁盘对应的叶子节点还未修改的“脏缓存”问题。
而levelDB中，data block存的是sstable的数据，而每个sstable文件是只读的，不可修改的，因此不会出现“脏缓存”问题。
另一点，如果缓存数据被大片读入的新数据驱除，是否会有优化方案？这其实就依赖于lru的具体实现了(比如分为old和young区)，levelDB本身并没有做特殊处理。

1

投入另外一个陌生
2020-05-08

老师，不好意思哈，再追问一下😀那为啥用change buffer + WAL优化后的MySQL的写性能还是不如LSM类的存储系统啊？原因是啥啊

展开

作者回复: 因为对于b+树，当内存中的change buffer写满的时候，会去更新多个叶子节点，这会带来多次磁盘IO;但lsm当内存中的memtable写满时，只会去写一次sstable文件。因此它们的主要差异，还是在怎么将数据写入磁盘上。
当然所有的系统设计都是有利有弊，要做权衡。b+树写入磁盘后，随机读性能比较好;而lsm树写磁盘一时爽，但要随机读的时候就不爽了，它可能得在多层去寻找sstable文件，因此随机读性能比b+树差。

投入另外一个陌生
2020-05-08

老师晚上好，请教个问题哈。

MySQL在写数据的时候，是先写到change buffer内存中的，不会立刻写磁盘的，达到一定量再将change buffer落盘。这个和Memtable的设计理念类似，按理说，速度也不会太慢吧?

展开

作者回复: 你说得对，在MySQL的b+树的具体实现中，其实借鉴了许多lsm树的设计思想来提升性能，比如使用wal技术+change buffer，然后批量写叶子节点，而不是每次都随机写。这样就

能减少磁盘IO。的确比原始的b+树快。
不过在大批量写的应用场景中，这样优化后的b+树性能还是没有lsm树更好。因此日志系统这类场景还是使用lsm树更合适。

1

一步
2020-05-08

课后思考：
1: 因为 LevelDB 天然的具有缓存的特性，最经常使用的最新的数据离用户最近，所有在上层找到数据就不会在向下找了
2: 如果规定生成文件的个数，那么有可能当前层和下一层的存储大小相近了，起不到分层的作用了

展开

作者回复: 1.没错，最新的数据在上层，所以上层能找到，就不需要去下层读旧数据了。
2.是的，由于在生成sstable文件时，有这么一个限制:新生成的sstable文件不能和下层的sstable覆盖度超过十个，因此可能会生成多个小的sstable文件。那如果只看文件数的话，多个小的sstable文件可能容量和下一层差不多，这样就没有了分层的作用了。

那时刻
2020-05-08

有两个问题，请教下老师。
1。在多路归并生成第 n 层的 SSTable 文件时，如何控制当前层最大容量呢？如果超过当前层的容量是停止计算还是把多余的量挪到下一层?
2。数据索引区里meta index block，当存在多个过滤器时，对过滤器进行索引。这是涉及到filter block过滤么?

展开

作者回复: 1.不用停止计算，而是算完后，判断容量是否达到上限，如果超过，就根据文中介绍的选择文件的方式，将多余的文件和下一层进行合并。
2.如果存在多个filter block，而且每个filter都很大的话(比如说bloomfilter就有许多数据)，将所有的filter都读入内存会造成多次磁盘IO,因此需要有metaphor index block，帮助我们只读取我们需要的filter即可。

那时刻
2020-05-08

讨论问题1：在某一层找到了key，不需要再去下一层查找的原因是，这一层是最新的数据。即使下一层有的话，是旧数据。这引申出另外一个问题，数据删除的时候是怎么处理的呢？是另外一个删除列表来保存删除的key吗？
问题2：如老师提示的这样，SSTable 的生成过程会受到约束，SSTable在归并的过程中，可能由于数据倾斜，导致某个分区里的数据量比较大，所以没有办法保证每个SSTable的...
展开

作者回复: 问题1:你进一步去思考删除问题。这一点很好。数据删除时，是会将记录打上一个删除标记，然后写入sstable中。
sstable和下一层合并时，对于带着删除标记的记录，levelDB会判断下层到最后一层是否还有这个key记录，有两种结果:
1.如果还有，那么这个删除标记就不能去掉，要一直保留到最后一层遇到相同key的时候才能删除;
2.如果没有，那么就可以删除掉这个带删除标记的记录。
问题2:因为在进行合并时，新生成的sstable会受到一个约束:如果和下一层的sstable重叠数超过了十个，就要停止生成这个sstable，要再继续生成一个新的sstable。这个机制会导致我们不好控制文件的个数。不如限定容量更合适。


投入另外一个陌生
2020-05-08

老师辛苦了，经常都是凌晨更新，终于等到这篇文章了😁

作者回复: 哈哈，凌晨更新是系统默认上线操作，是由极客时间的工作人员们负责的，他们辛苦了。


ykkk88
2020-05-08

老师，如果是memtable有删除key的情况下，skiplist是不是设置墓碑标志，刷level 0的时候 sstable也还是有这个删除标记，只有在最下层的sstable合并时候再真的物理删除key啊，感觉不这么做，可能get时候会读出来已经被删除的key
展开

作者回复: 你思考得很仔细，的确是的。sstable中，每一条记录都有一个标志位，表示是否是删除。这样就能避免误查询。

对于有删除标志的记录，其实查询流程是一致的，就是查到数据就返回，不再往下查。然后看这个数据的状态位是有效还是删除，决定是否使用。

1

峰

2020-05-08

问题1： 因为只是get(key), 所以上层的sstabe的数据是最新的，所以没必要再往下面查，但如果有hbase这样的scan(startkey,endkey) 那还是得全局的多路归并（当然可以通过文件元数据迅速排除掉一些hfile）

问题2：SSTable 的生成过程会受到约束，无法保证每一个 SSTable 文件的大小。哈哈哈，我在抄答案，我其实有疑问，就算限定文件数量，那么在层次合并的时候，假设我...

展开

作者回复: 1.你提到的这个问题非常好，lsm的范围查询比较弱，需要遍历。一种优化思路是先在level 1层中找到range，然后基于start和end的位置，去下一层再找start和end的位置。这样能提高范围查询的性能。

2.我说一下我的理解，由于有“生成的每个sstable和下一层的sstable重合度不能超过十个”这个约束，所以sstable生成过程中可能随时被截断，因此不好控制sstable的数量。不如控制容量简单。