07 | NoSQL检索：为什么日志系统主要用LSM树而非B+树?

2020-04-10 陈东

检索技术核心20讲 进入课程 >

讲述：陈东
时长 15:13　大小 13.94M

你好，我是陈东。

B+ 树作为检索引擎中的核心技术得到了广泛的使用，尤其是在关系型数据库中。

但是，在关系型数据库之外，还有许多常见的大数据应用场景，比如，日志系统、监控系统。这些应用场景有一个共同的特点，那就是数据会持续地大量生成，而且相比于检索操作，它们的写入操作会非常频繁。另外，即使是检索操作，往往也不是全范围的随机检索，更多的是针对近期数据的检索。

那对于这些应用场景来说，使用关系型数据库中的 B+ 树是否合适呢?

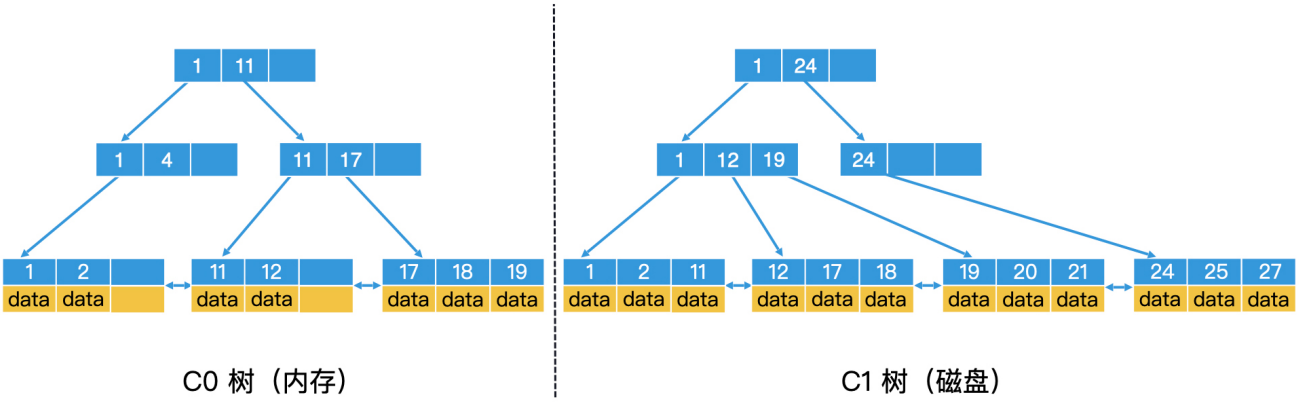
我们知道，B+ 树的数据都存储在叶子节点中，而叶子节点一般都存储在磁盘中。因此，每次插入的新数据都需要随机写入磁盘，而随机写入的性能非常慢。如果是一个日志系统，每秒钟要写入上千条甚至上万条数据，这样的磁盘操作代价会使得系统性能急剧下降，甚至无法使用。

那么，针对这种频繁写入的场景，是否有更合适的存储结构和检索技术呢？今天，我们就来聊一聊另一种常见的设计思路和检索技术：**LSM 树**（Log Structured Merge Trees）。LSM 树也是近年来许多火热的 NoSQL 数据库中使用的检索技术。

如何利用批量写入代替多次随机写入？

刚才我们提到 B+ 树随机写入慢的问题，对于这个问题，我们现在来思考一下优化想法。操作系统对磁盘的读写是以块为单位的，我们能否以块为单位写入，而不是每次插入一个数据都要随机写入磁盘呢？这样是不是就可以大幅度减少写入操作了呢？

LSM 树就是根据这个思路设计了这样一个机制：当数据写入时，延迟写磁盘，将数据先存放在内存中的树里，进行常规的存储和查询。当内存中的树持续变大达到阈值时，再批量地以块为单位写入磁盘的树中。因此，LSM 树至少需要由两棵树组成，一棵是存储在内存中较小的 C0 树，另一棵是存储在磁盘中较大的 C1 树。简单起见，接下来我们就假设只有 C0 树和 C1 树。



极客时间

LSM 树由至少 2 部分组成：内存的 C0 树和磁盘的 C1 树

C1 树存储在磁盘中，因此我们可以直接使用 B+ 树来生成。那对于全部存储在内存中的 C0 树，我们该如何生成呢？在上一讲的重点回顾中我们分析过，在数据都能加载在内存中的时候，B+ 树并不是最合适的选择，它的效率并不会更高。因此，C0 树我们可以选择其他的数据结构来实现，比如平衡二叉树甚至跳表等。但是为了让你更简单、清晰地理解 LSM 树的核心理念，我们可以假设 C0 树也是一棵 B+ 树。

那现在 C0 树和 C1 树就都是 B+ 树生成的了，但是相比于普通 B+ 树生成的 C0 树，C1 树有一个特点：所有的叶子节点都是满的。为什么会这样呢？原因就是，C1 树不需要支持随机写入了，我们完全可以等内存中的数据写满一个叶子节点之后，再批量写入磁盘。因此，每个叶子节点都是满的，不需要预留空位来支持新数据的随机写入。

如何保证批量写之前系统崩溃可以恢复？

B+ 树随机写入慢的问题，我们已经知道解决的方案了。现在第二个问题来了：如果机器断电或系统崩溃了，那内存中还未写入磁盘的数据岂不就永远丢失了？这种情况我们该如何解决呢？

为了保证内存中的数据在系统崩溃后能恢复，工业界会使用 **WAL 技术**（Write Ahead Log，预写日志技术）将数据第一时间高效写入磁盘进行备份。WAL 技术保存和恢复数据的具体步骤，我这里总结了一下。

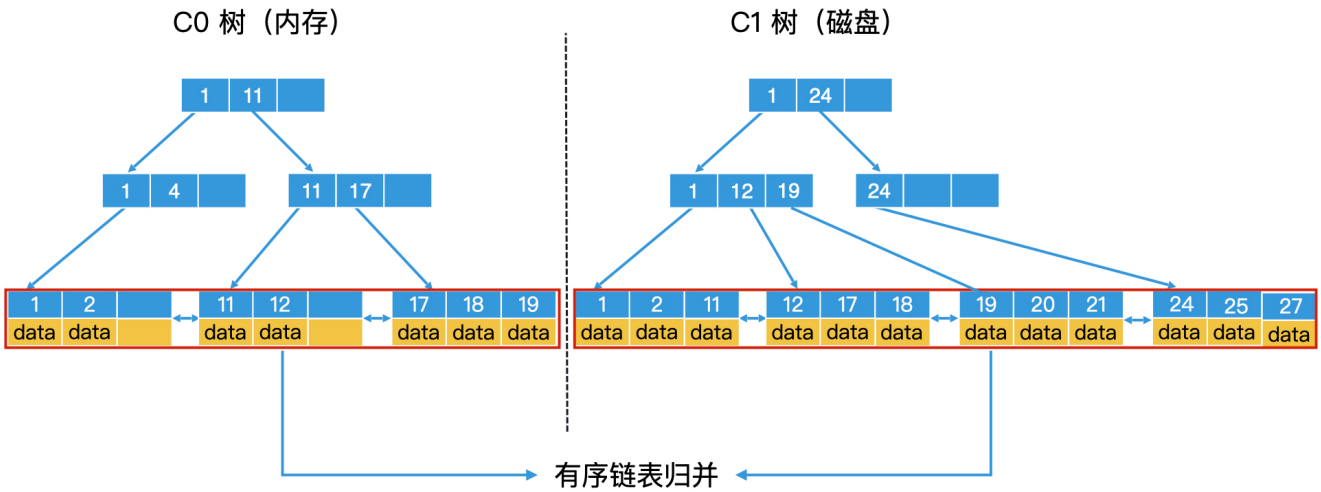
1. 内存中的程序在处理数据时，会先将对数据的修改作为一条记录，顺序写入磁盘的 log 文件作为备份。由于磁盘文件的顺序追加写入效率很高，因此许多应用场景都可以接受这种备份处理。
2. 在数据写入 log 文件后，备份就成功了。接下来，该数据就可以长期驻留在内存中了。
3. 系统会周期性地检查内存中的数据是否都被处理完了（比如，被删除或者写入磁盘），并且生成对应的检查点（Check Point）记录在磁盘中。然后，我们就可以随时删除被处理完的数据了。这样一来，log 文件就不会无限增长了。
4. 系统崩溃重启，我们只需要从磁盘中读取检查点，就能知道最后一次成功处理的数据在 log 文件中的位置。接下来，我们就可以把这个位置之后未被处理的数据，从 log 文件中读出，然后重新加载到内存中。

通过这种预先将数据写入 log 文件备份，并在处理完成后生成检查点的机制，我们就可以安心地使用内存来存储和检索数据了。

如何将内存数据与磁盘数据合并？

解决了内存中数据备份的问题，我们就可以接着写入数据了。内存中 C0 树的大小是有上限的，那当 C0 树被写满之后，我们要怎么把它转换到磁盘中的 C1 树上呢？这就涉及**滚动合并**（Rolling Merge）的过程了。

我们可以参考两个有序链表归并排序的过程，将 C0 树和 C1 树的所有叶子节点中存储的数据，看作是两个有序链表，那滚动合并问题就变成了我们熟悉的两个有序链表的归并问题。不过由于涉及磁盘操作，那为了提高写入效率和检索效率，我们还需要针对磁盘的特性，在一些归并细节上进行优化。



C0 树和 C1 树滚动合并可以视为有序链表归并

由于磁盘具有顺序读写效率高的特性，因此，为了提高 C1 树中节点的读写性能，除了根节点以外的节点都要尽可能地存放到连续的块中，让它们能作为一个整体单位来读写。这种包含多个节点的块就叫作**多页块**（Multi-Pages Block）。

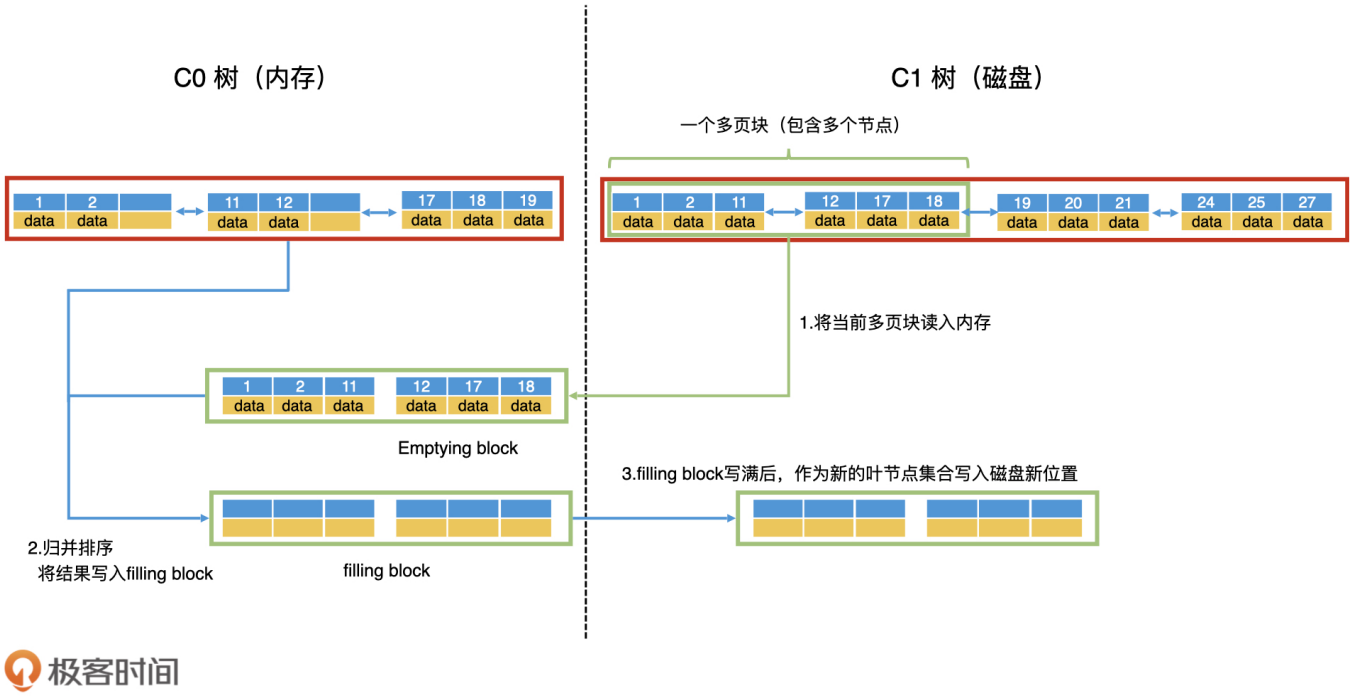
下面，我们来讲一下滚动归并的过程。在进行滚动归并的时候，系统会遵循以下几个步骤。

第一步，以多页块为单位，将 C1 树的当前叶子节点从前往后读入内存。读入内存的多页块，叫作清空块（Emptying Block），意思是处理完以后会被清空。

第二步，将 C0 树的叶子节点和清空块中的数据进行归并排序，把归并的结果写入内存的一个新块中，叫作填充块（Filling Block）。

第三步，如果填充块写满了，我们就要将填充块作为新的叶节点集合顺序写入磁盘。这个时候，如果 C0 树的叶子节点和清空块都没有遍历完，我们就继续遍历归并，将数据写入新的填充块。如果清空块遍历完了，我们就去 C1 树中顺序读取新的多页块，加载到清空块中。

第四步，重复第三步，直到遍历完 C0 树和 C1 树的所有叶子节点，并将所有的归并结果写入到磁盘。这个时候，我们就可以同时删除 C0 树和 C1 树中被处理过的叶子节点。这样就完成了滚动归并的过程。



使用清空块和填充块进行滚动归并

在 C0 树到 C1 树的滚动归并过程中，你会看到，几乎所有的读写操作都是以多页块为单位，将多个叶子节点进行顺序读写的。而且，因为磁盘的顺序读写性能和内存是一个数量级的，这使得 LSM 树的性能得到了大幅的提升。

LSM 树是如何检索的?

现在你已经知道 LSM 树的组织过程了，我们可以来看，LSM 树是如何完成检索的。

因为同时存在 C0 和 C1 树，所以要查询一个 key 时，我们会先到 C0 树中查询。如果查询到了则直接返回，不用再去查询 C1 树了。

而且，C0 树会存储最新的一批数据，所以 C0 树中的数据一定会比 C1 树中的新。因此，如果一个系统的检索主要是针对近期数据的，那么大部分数据我们都能在内存中查到，检索效率就会非常高。

那如果我们在 C0 树中没有查询到 key 呢？这个时候，系统就会去磁盘中的 C1 树查询。在 C1 树中查到了，我们能直接返回吗？如果没有特殊处理的话，其实并不能。你可以先想想，这是为什么。

我们先来考虑一种情况：一个数据已经被写入系统了，并且我们也把它写入 C1 树了。但是，在最新的操作中，这个数据被删除了，那我们自然不会在 C0 树中查询到这个数据。可是它依然存在于 C1 树之中。

这种情况下，我们在 C1 树中检索到的就是过期的数据。既然是过期的数据，那为了不影响检索结果，我们能否从 C1 树中将这个数据删除呢？删除的思路没有错，但是不要忘了，我们不希望对 C1 树进行随机访问。这个时候，我们又该怎么处理呢？

我们依然可以采取延迟写入和批量操作的思路。对于被删除的数据，我们会将这些数据的 key 插入到 C0 树中，并且存入删除标志。如果 C0 树中已经存有这些数据，我们就将 C0 树中这些数据对应的 key 都加上删除标志。

这样一来，当我们在 C0 树中查询时，如果查到了一个带着删除标志的 key，就直接返回查询失败，我们也就不用去查询 C1 树了。在滚动归并的时候，我们会查看数据在 C0 树中是否带有删除标志。如果有，滚动归并时就将它放弃。这样 C1 树就能批量完成“数据删除”的动作。

重点回顾

好了，今天的内容就先讲到这里。我们一起来回顾一下，你要掌握的重点内容。

在写大于读的应用场景下，尤其是在日志系统和监控系统这类应用中，我们可以选用基于 LSM 树的 NoSQL 数据库，这是比 B+ 树更合适的技术方案。

LSM 树具有以下 3 个特点：

1. 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（Merge Trees）；
2. 用批量写入代替随机写入，并且用预写日志 WAL 技术保证内存数据，在系统崩溃后可以被恢复；
3. 数据采取类似日志追加写的方式写入（Log Structured）磁盘，以顺序写的方式提高写入效率。

LSM 树的这些特点，使得它相对于 B+ 树，在写入性能上有大幅提升。所以，许多 NoSQL 系统都使用 LSM 树作为检索引擎，而且还对 LSM 树进行了优化以提升检索性能。在后面的章节中我们会介绍，工业界中实际使用的 LSM 树是如何实现的，帮助你对 LSM 树有更深入的认识。

课堂讨论

为了方便你理解，文章中我直接用 B+ 树实现的 C0 树。但是，对于纯内存操作，其他的类树结构会更合适。如果让你来设计的话，你会采用怎么样的结构作为 C0 树呢？

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，也欢迎把这篇文章分享给你的朋友。

上一篇 06 | 数据库检索：如何使用B+树对海量磁盘数据建立索引?

下一篇 08 | 索引构建：搜索引擎如何为万亿级别网站生成索引?

精选留言 (12)

写留言

innocent
2020-04-11

为了性能内存中的树至少有两棵吧

展开

作者回复: 你提到了性能问题，的确是这样的。在高性能的lsm树实现中，不仅仅是内存中要有两棵，磁盘上还要有多棵。
具体工业界是怎么真正实现一个高性能的lsm树，后面我会再具体介绍。

1 1

兰柯一梦
2020-04-10

感觉取决于系统需要提供什么样的功能，如果系统需要提供高效的查询不需要范围scan那

么C0用hashmap都可以，如果需要scan那么平衡树或者skiplist比较合适。leveldb是使用skiplist来实现的，这里的checkpoint主要目的是定期将数据落盘后用来对log文件进行清理的，使得系统重启时不需要重放过多的log影响性能

展开

作者回复: 你说的很对，取决于使用场景。有的系统的确是使用哈希表的。还有使用红黑树和跳表的都有。

1

xzy

2020-04-10

请问，如果wal所在的盘和数据在同一个盘，那怎么保证wal落盘是顺序写呢，我理解也得寻道寻址

作者回复: 你这个问题很好！我提炼出来有三个点:
1.wal的日志文件能否保证在物理空间上是顺序的?
这个是可以做到的。日志文件都是追加写模式，包括可以提前分配好连续的磁盘空间，不受其他文件干扰。因此是可以保证空间的连续性。
2.wal的日志文件和其他数据文件在一个磁盘，那么是否依然会面临磁头来回移动寻道寻址的问题?
这个问题的确存在，如果日志文件和数据文件在同一个盘上，的确可能面临一个磁头来回移动的情况。因此，尽量不要在一个磁盘上同时开太多进程太多文件进行随机写。包括你看lsm的写磁盘，也是采用了顺序写。
3.如果第二个问题存在，那么wal依然高效么?
wal依然是高效的。一方面，如果是wal连续写(没有其他进程和文件竞争磁头)，那么效率自然提升；另一方面，往磁盘的日志文件中简单地追加写，总比处理好数据，组织好b+树的索引结构再写磁盘快很多。

1 1

峰

2020-04-10

考虑的点
1 随机和顺序存取差距不大
2 什么样的有序结构适合高并发的写入
对于2，必须插入的时候只影响局部，这样上锁这样开销就只在局部细粒度上，如果是树可能存在需要调整树高的各种旋转或者分裂合并操作。对于1，在说不用像b树那样降低树...

展开

作者回复: 全内存数据库的确是一个研究热点。Redis的广泛流行也是这个潮流的一个例子。
b+树即使能全部缓存到内存中，但你思考一下它插入删除的效率(分裂合并节点)，它不会比红黑树这些结构效率高。因此，纯内存的环境下，红黑树和跳表这类结构更受欢迎。

1

pedro
2020-04-11

不知道为什么我昨天发布成功的评论没有被通过，可能 bug了，那我重说一次😭。

问题，前面的文章提到 B 树是为了解决磁盘 io 的问题而引入的，所以 B 树自然不适合做内存索引，适合的是红黑球和跳表。
...
展开

作者回复: 很好！你提到了bitcask，它是用哈希表作为内存索引的。其他的一些nosql实现，还有选择红黑树和跳表的。因此你会发现，对于内存中高效检索的技术选型，不同应用会根据自己的需求，选择我们在基础篇介绍过的适用技术。

那时刻
2020-04-11

请问两个关于检查点check point的问题。
1.检查点也是要落盘，和WAL一样的位置么?
2.在数据删除和同步到硬盘之后会生成检查点，还有其它情况会生成检查点么?

作者回复: 1.check point的信息是独立存在的，和wal的日志文件是不同文件。
2.check point的触发条件可以有多个，比如说间隔时长达到了预定时间，比如说wal文件增长到一定程度，甚至还可以主动调用check point相关命令强制执行。

xzy
2020-04-10

你好，这里还有个问题：如果是ssd，顺序写和随机写的差异不大，那么是否还有必要写wal， 毕竟写wal相当于double写了数据，那直接就写数据是否性能还会更好呢

作者回复: 这是一个好问题！对于SSD，这些理论和方法是否依然有效?答案是yes。
考虑这么两点:
1.SSD是以page作为读写单位，以block作为垃圾回收单位，因此，批量顺序写性能依然大幅高于随机写！
2.SSD的性能和内存相比依然有差距，因此，先在内存处理好，再批量写入SSD依然是高效的。


一步
2020-04-10

当内存的C0 树满时， 都要 把磁盘的 C1 树的全部数据 加载到内存中合并生成新树吗？ 我感觉这样性能不高啊。

还有就是 类型日志系统，都是天然按照时间排序；这样的话 ，就可以直接把 C0 树的叶子节点直接放到 C1 树的叶子节点后面啊，没有必要在进行合并生成新树了

展开

作者回复: 你提了一个非常好的问题！把c1树的全部叶子节点处理一次的确效率不高，因此实际上会有多棵不同大小的磁盘上的树。包括工业界还有其他的优化思路。后面会介绍。
此外，直接把c0树的叶子节点放在c1树后面，这样的话叶子节点就不是有序的了，就无法高效检索了。


一步
2020-04-10

填充块写满了，我们就要将填充块作为新的叶节点集合顺序写入磁盘，

这个时候 填充快写的磁盘位置会是之前C1 叶子节点 清空块的位置吗？ 还是另外开辟有个新的空间，当新的树生成后，在把旧的C1树 磁盘数据空间在标记为删除？

展开

作者回复: 是新的磁盘空间。因为c1树要保证在磁盘上的连续性，如果是利用原c1树的旧空间的话，可能会放不下(因为合并了c0树的数据)。


一步
2020-04-10

对于 对数据敏感的数据库，基本上都会采用 WAL 技术，来防止数据在内存中丢失， 比如 MySQL 的 redo log

展开

Mq

2020-04-10

B+树数据存储是以块为单位的，读到内存也是以块为单位，一个块里面如果数据多了顺序查找也不是太快。
内存存储我觉得应该选择跳表，他的查找、插入、删除都跟树一样，只是空间上会多一些消耗，他对范围查找的支持是其他内存数据结构一大优势。
另外，我感觉文章中有把c0写成c1的地方，不知道我理解对不‘和普通的 B+ 树相比，C...

展开

作者回复: 在内存数据库里面，Redis是一个典型的代表，它的确就是使用跳表的。
另外两点:
1.b+树块内的查找，可以使用二分查找，而不是顺序查找，这样能加快检索效率。
2.你说的是否“c0和c1树写错”的地方，这个的确没有写错哦，你可以仔细想一想，为什么c1树有这个特点“叶子节点是全满的”。

1

每天晒白牙

2020-04-10

思考题
对于c0树因为是存放在内存中的，可以用平衡二叉树或跳表来代替 B+ 树

展开

作者回复: 是的。如果你有关注一些内存数据库，你会发现，有好些是使用跳表和红黑树来实现的。比如Redis。
可见，b+树不是万能的。我们理解了存储介质的特性以后，能帮助我们在合适的场景做出正确的技术选型。