lsmtree

牺牲了部分读性能，用来大幅提高写性能

论文原文

论文阅读-The Log-Structured Merge-Tree (LSM-Tree)

摘要

这两天看了下LSM树的内容，网上的博文大多讲的不太详细，但都有提到这篇论文。本着严谨的态度，就找时间啃了下此论文，在这里对核心内容做一些记录。

论文摘要

高性能的交易系统通常会为一次交易就插入若干条记录到历史表，使其可追溯。这就使得高效的实时索引十分重要□LSM树是一个基于磁盘的数据结构，主要用于为那些高概率插入和删除的记录提供低成本的索引。

LSM使用了一个算法来延迟批处理索引变更，然后类似归并排序的方式串联起一个基于内存的组件和若干基于磁盘的组件上面的所有变更信息。该算法相比于传统的B树访问方式大大减少磁盘臂的移动开销。

由于索引搜索需要立刻响应，这会在某些场景降低IO效率，所以LSM树在索引写入占比大大超过索引查找的场景中最适用，如历史表和日志文件。

介绍

3.1 五分钟法则

当磁盘页的访问频率超过每60秒1次，我们就可以通过购买（扩充）内存缓存空间来将磁盘页保存到内存，以减少磁盘IO带来的系统开销。

3.2 原文中例二

一个在具有高插入量的历史记录表上的索引，可以证明这样的一个索引将会使TPC应用的磁盘开销加倍：一个建立在 AccountId + Timestamp之上的联合索引。他对于支持类似以下的近期账户行为信息高效查询至关重要：

```
//搜索某个时间之后的账户Id的所有历史行为数据
(1.1) Select * from History
where History.Acct-ID = %custacctid
and History.Timestamp > %custdatetime;
```

这个例子中的场景，搜索远远少于插入(正常人不会像存取款那样频繁地查看自己的账户相关行为信息)。因为每次搜索Acct-ID基本都是随机的，基本每次都需要读取一个磁盘页。但根据五分钟法则又不能将这些磁盘页放入内存，因为这些磁盘页读取大约相隔2300秒，也就是说这些访问都是磁盘IO□原文中的数据展现了商业系统中最常用的数据结构B+树索引带来的巨大的IO和磁盘开销。

然后又提到LSM树可以在插入索引时减少磁盘动作，以致开销少一个量级□LSM树使用的算法，可以延迟和批处理索引变更（这一点十分重要），并且以一种特别高效的、类似归并排序的方式将这些变更迁移到磁盘□LSM树结构还支持其他索引操作，例如删除，更新，甚至是长延迟的查找动作（只不过哪些需要立刻返回的查询需要相对多的开销□□LSM特别适合前面例子那样搜索远远少于插入的场景。这种场景下，最重要的就是减少索引插入时的开销。当然，也需要维护一个索引，因为搜索行为不可能全是顺序的。

LSM树算法的两大组件

4.1 组件介绍

LSM树由两个或更多的类树组件组成。本章只讨论最简单的2个组件的情况。同时，这里会用LSM树来探讨前面的账户历史记录例子。

一棵拥有两个组件的LSM树拥有两部分：

- 一个较小的位于内存的组件，就是上图中的C0 tree
- 一个较大的位于磁盘的组件，就是上图中的C1 tree

尽管C1树常驻磁盘，但他的常被访问的磁盘页会被保留在内存中(未在图中展示)。

历史记录表的数据每生成一行新记录流程如下：

- 首先向顺序日志文件中写一条用于恢复这次插入行为的日志记录
- 该行数据的索引被插入到常驻内存的 C0 树中
- 会适时地将这些C0树上的数据迁移到磁盘上的C1树中
- 每个索引的搜索过程都是先C0后C1

上述C0->C1的数据迁移过程有一定时间时延，这就暗含了因为系统崩溃导致没刷到C1中的那部分索引数据可恢复的需求。

4.2 组件合并

4.2.1 C0合并到C1简述

上述C0树写入是无IO开销的，但是C0位于内存，成本远高于磁盘，这就需要有一种高效的刷盘方式到C1

LSM树采用的方法是当C0树上的插入的数据几乎达到指定的阈值时，有一个持续循环的合并进程服务会删除C0树上的一些连续segment段，将他们合并到磁盘中的C1树。下图展示了这个进程：

4.2.2 C1树结构

LSM中C1树具有与B树类似的目录结构，但C1树与B树不同的是C1树的所有节点都是满的。并且为了更有效的利用磁臂，做了以下优化：

单页块	4KB	根节点；每个层级上的单页节点	单页节点被用在匹配索引查找中，以最小化缓存需求
多页块	256KB	根目录下的每个层级上的单页节点序列会被打包，然后一起放入连续的多页磁盘块中（囊括了根节点以下的节点），利于磁盘顺序访问	多页块IO在滚动合并期间、大范围的范围搜索中被使用

4.2.3 C0滚动合并到C1简述

滚动合并的行为包括一系列的合并步骤：

- 先读取包含C1树的叶节点的多页块，这会使得C1中的一系列节点条目驻留到缓存
- 然后每次合并都会去读取已经被缓存的C1树的一个磁盘页大小的叶节点
- 接着将第二步中读取到的C1树叶节点上的条目与C0树的叶节点条目进行合并，并减少C0树的大小
- 合并完成后，会为C1树创建一个已合并的新叶节点（在缓存中，填满后被刷入磁盘）

4.2.4 empty block和filling block

- empty block 那些在合并前，已缓存、且包含旧的C1树节点的多页块被称为empty block，意味着他们会被清空、移除。
- filling block 而新的叶节点被写入与旧的多页块不同的已缓存的多页块，被称为filling block，意味着他们会被填满。当filling block被C1树新合并的叶节点填满时，该多页块会被写入一个磁盘上一个新的空闲区域。注意，新合并的块会被写入新的磁盘位置，这使得旧的块不会被覆盖。这样的好处是可在系统崩溃时快速恢复。 上面的图中下方圆圈内就是新的多页block，包含了merge结果。随后的合并步骤持续将C0和C1组件的增加的索引值段汇集在一起，直到达到最大值，此时滚动合并又从最小值再次开始。

C1的父目录节点也会被缓存到内存，被更新以反映新叶子节点的结构变动，但通常会在缓存中保留更长的时间以减少IO，C1上旧的叶节点会在合并完成后变为非法，然后被从C1目录中删除。为了缩短恢复时重建所需时长，会定期进行合并过程的checkpoint，这会将所有缓冲的信息强制刷到磁盘。

4.3 LSM树组件创建流程

4.3.1 C0树结构

前面说过磁盘上的C1树是类B树，但C0树不同，他不是类B树。因为C0树不会都是在内存不会放在磁盘，所以节点可以任意大小不必考虑磁盘页大小而设计。这样一来，就没有必要为了减少IO而牺牲CPU效率，从而刻意地将树高度压得很低。比如B树为了这么做，就让树每一层非常宽，每个节点内部的关键字特别多，需要顺序查找。这种情况下，2-3树或AVL树效率高于B树，可作为C0树的数据结构。

4.3.2 C0滚动合并到C1详解

首次从C0树到C1树的过程如下：

- 当增长中的C0树第一次达到阈值的时候
- 最靠左的一系列条目会以高效批量形式从C0树中删除
- 然后被重组到C1树(按key递增顺序)，将被完全填满
- 连续的C1树的叶节点会按从左到右的顺序，首先被放置到常驻内存的多页块内的若干初始页上
- 持续上一步直到该多页块被填满
- 然后该多页块被刷到磁盘，成为C1树叶节点层的第一部分，常驻磁盘
- 随着这些连续的叶节点添加的过程，一个C1树的目录节点结构会在内存缓存中被创建（这些上层目录节点被存在单独的多页块缓存或单独的页缓存中，为了更高效利用内存和磁盘。其中还包含分隔点索引，可以将访问精确匹配导向某个下一层级的单页节点而不是多页块，类似B树。这样一来，我们就可在滚动合并或长范围搜索时使用多页块，而在索引精确匹配访问时使用单页节点）
- 在发生如下情况时，C1的目录节点会被强制刷盘
    - 包含目录节点的多页块缓存满了 -> 只有该多页块会被刷盘
    - 根节点分裂，增加了C1树的深度(大于2) -> 所有多页块刷盘
    - checkpoint被执行-> 所有多页块刷盘

4.3.3 滚动合并与概念游标

可以把拥有两个组件的LSM树的滚动合并的过程，想象为拥有一个概念上的游标，他在C0树和C1树的等值key value间缓慢穿梭移动，将C0树的索引数据取出放入磁盘上的C1树中。

这个滚动合并游标在C1树的叶子节点和上层目录都有个位置点。每个层级上，所有C1树的正在合并的多

页块通常会被分割为两块，且为了并发访问他们分别拥有整数个页大小的C1树节点。这两个类型块如下：

- empty block 其条目已经耗尽，但保留了游标尚未到达的一些信息
- filling block 反映出此刻的合并结果

一旦需要将所有缓存的节点刷入磁盘，所有层级的缓存信息必须被写入新的磁盘位置，这些位置会被反映到上层的目录信息，还会生成一系列的日志条目可用来做恢复。

不久之后，当缓存中的filling block(存放了C1树某些层级)被填满时，需要再次被刷盘，此时会存入一个新的磁盘位置。在恢复期间需要的旧信息不会被覆写磁盘，只有当足够的新信息被写入后才会失效。原文第四部分详述了关于滚动合并的信息，提到了并发和可恢复的设计。（我其实觉得已经够详细够晦涩了，可能是我水平太低了- -|）

4.3.4 废弃块重用

LSM树关于C1树的某一特定层级节点的滚动合并过程，会达到相对较高的rate□所有读写都在多页块中，这是关于效率方面的重要考虑。通过消除磁盘寻道时间和旋转时延，我们希望效率远超过随机页IO(如B树插入时)。新的多页块写入使用新的磁盘空间，这就暗含了需要能重用废弃的块。这些使用记录可以放入到一个内存表中。旧的多页块被废弃然后被作为单独的单元被重用，而恢复是由checkpoint保证。

- 在Log-Structured文件系统中，对废旧块的重用包括了很大的IO开销，因为块通常仅为部分释放，因此重用时需要对块进行读/写。
- 而在LSM树中，块会在滚动合并中被完全释放，不存在额外IO□

4.4 LSM树索引查找

4.4.1 搜索原则

当一个需要立刻返回的精确匹配查询或是范围查询在LSM树的索引上执行时，会先在C0树执行搜索值，然后搜C1树。这暗含着少许额外的CPU开销(相对于B树来说)，因为分别去两棵树目录进行搜索。

如上图，考虑多个组件的LSM树，拥有C0, C1, C2 ... Ck□这样一个递增的索引树结构。其中C0是驻留在内存的，而且他组件都在磁盘。这种情况下，每当Ci-1 条目达到阈值时，每个(Ci-1,Ci)之间的异步滚动合并过程会从较小的组件中移动条目到较大的组件。这个结论很重要，务必牢记。

LSM有一个规则，即为了保证LSM中所有条目都被检查到，就必须让每个精确匹配或范围查找要访问每个Ci组件的索引结构。然而，有一些可能的优化方式，可使得此搜索范围限制在组件的初始子集。请看下一节。

4.4.2 搜索优化

- 如果记录的生成逻辑就能保证索引唯一性，比如时间戳，那么在较小的一个Ci组件内搜索到结果就可以直接停止搜索直接返回。
- 再比如，搜索条件中使用了最近的时间戳，我们可以限制搜索范围为那些还没有移动到最大组件的记录所处的序号小的组件上。当合并游标在□Ci□Ci + 1□对中循环徘徊时，我们通常有理由保留Ci中那些最近插入的条目（比如在最近τi秒内），而只允许那些较旧的条目移动到Ci+1中。
- 在最频繁的查询指向最近插入的值的这种场景中，许多搜索都可在C0树中就完成，这样就使得C0树完全发挥了了自己的内存缓存价值。这是十分重要的性能优化策略。举个例子，被用作短期事务UNDO日志索引，在中止事件中被访问，在创建这些索引后会有大比例是访问短期内的数据，所以我们可以预期大多数这样的索引会驻留在内存中。
- 通过跟踪每个事务的开始时间，我们可以保证在最后的τ0秒内启动事务的所有日志，例如，将在组件C0中找到，而无需搜索位于磁盘的其他LSM树组件。

4.5 LSM树删除、更新

4.5.1 删除

注意，删除也可以和插入一样享用延迟和批处理的好处。 具体来说，当一个索引行被删除时，如果一个key-value键值对没有在C0树中找到，那么可以把一个删除节点条目放到该位置，同样会被key索引，但该索引指向一个应该被删除的RowId条目。而真正的删除可在稍后的滚动合并过程中扫描到真正的该索引条目时完成：其实就是将删除节点条目合并到更大的组件时，遇到对应的真实条目是就之间被真正删除了，就跟正反物质湮灭一样，一遇到就嗖的一下双双归西了。

同时，查找请求必须被那些删除节点条目过滤掉，避免结果中返回要被删除的记录。这个过滤很容易实现，很容易想到那些删除节点条目肯定是在相较于真实条目较早的组件上，这样还能使得搜索尽量早的发现条目被删除而不必搜索到最后的组件才返回。

还有另一种高效索引修改的操作。称为谓词删除的过程提供了一种通过简单地断言谓词来执行批量删除的方法，例如，删除时间戳超过20天的所有索引值的谓词。当存在于最老（也是最大）的LSM组件中的受影响条目，在滚动合并的过程中变为内存驻留时，此断言会导致它们在合并过程中被简单地删除

4.5.2 更新

导致索引值更改的记录更新，这在任何类型的应用程序中都是不常见的。但如果我们将更新视为删除后紧跟着插入，则可以由LSM树以延迟方式处理此类更新。

4.5.3 长延时查找

长延迟查找的结果可以等待最慢的合并游标的循环周期□LSM提供了响应查询的有效手段。 在组件C0中插入查找注释条目，该查询会在迁移到后续组件时真正执行。 一旦该查找注释条目循环合并到达了LSM树的最大组件的适当区域，就完成了长延迟查找的过程，最终返回累积得到的RowID列表。

LSM并发

5.1 LSM树多组件原理回顾

回顾下前面的内容，考虑K+1个组件的LSM树，拥有C0, C1, C2 ... Ck□这样一个递增的索引树结构。其中C0是驻留在内存的，而且他组件都在磁盘。这种情况下，每当Ci-1 条目数量达到阈值时，每个(Ci-1,Ci)之间的异步滚动合并过程会从较小的组件中移动条目到较大的组件。每个驻留磁盘组件都是由磁盘页大小的节点按B树类型结构组成，此外根节点下的各层的节点都按照key的顺序排列并打包放置到多页块中□LSM树上层的目录信息可以指引单个页节点访问，还能指明哪些节点位于该多页块上，这样的好处是使得可以一次性执行对这样的块的读取或写入。

等值匹配时，一个基于此盘的Ci可被单独驻留在单页内存缓存中，也可被包含在多页块的缓存中。作为大范围搜索或是滚动合并的游标穿过块高频访问的结果，一个多页缓存会被缓存到内存。

5.2 LSM并发三大问题

无论如何□Ci组件的所有非锁定节点都可以随时进行访问目录查找，并且磁盘访问将执行旁路以查找内存中的任何节点，即使该节点是多页块一部分，参与滚动合并。总的来说□LSM树的并发访问必须解决以下三种物理冲突：

- 一个查找操作不应该访问一个基于磁盘组件的同时，另外一个进程正在滚动合并且正在修改该节点的内容。（读写不能并发）
- 当另外的进程正在更改C0某部分以执行滚动合并到C1的同时，在C0组件中查找或插入不应访问C0树的相同部分。（读和删除合并不能并发）
- 用来将数据从Ci-1移动到Ci的游标有时候需要穿越过用来将数据从Ci移动到Ci+1的游标，因为从Ci-1的数据导出速度总是至少与Ci数据导出数据相当，这就意味着Ci-1的游标运转速度更快。 无论采用何种并发方法，都必须允许上述情况发生，也就是说导出数据到Ci的进程不会因Ci从数据导出的进

程而阻塞。

5.3 基于磁盘组件合并时的并发

LSM树中用于并发控制访问基于磁盘的组件而导致冲突，所以加锁的单位是树的节点：

- 正在因滚动合并被修改的节点会被加上写锁
- 正在因搜索而读取的节点会被加上读锁
- 为了防止死锁，设计了目录锁相关方法

而C0树采用的锁实现方式具体依据是采用的数据结构。

- 例如，在 2-3树 的情况下，我们可以用写锁锁住2-3树的目录节点一个子树，该节点包含要合并到C1节点时受影响范围内的所有条目;
- 同时，查找操作会用读锁锁定那些处于搜索路径上的2-3树的所有节点，这是一个排他锁。锁的释放：

- 读锁 一旦叶节点上的条目被扫描完了，就会释放

- 写锁 滚动游标使用的写锁会在合并到更大的组件后背释放

为了提高并发，前面章节提到过的C1树的empty block和filling block都会包含整数个C1树中的页大小的节点，并驻留在内存。在合并重组节点时，这些节点会被加上写锁，以阻止对这些记录的其他类型并发访问。

5.4 C0到C1之时的并发

前面讨论的都是基于磁盘的组件间merger时的并发情况，现在说说C0到C1的合并时的并发情况。与其他合并步骤相同，CPU应该专注于合并任务，所以其他访问会被排他的写锁拒绝，当然这个时间会尽可能短。那些会被合并的C0条目应该被提前计算、提前加写锁。除此之外，CPU时间还会由于C0组件以批量的形式删除条目节省时间，而不是每次单独删除而尝试再平衡，C0树可以在整个合并步骤完成后被完全的平衡。

LSM恢复

注意，本节转自 日志结构的合并树 The Log-Structured Merge-Tree 作者:眺望海接天

6.1 检查点

在新条目插入到C0后，当C0与C1进行滚动合并时，某些条目将从C0转移到更大的组件中。由于滚动合并发生在内存缓存的多页块中，所以只有当条目真正写入硬盘时，滚动合并的成果才会真正生效。然而滚动合并时可能就会发生系统故障，进而使得内存数据丢失。为了能有效地进行系统恢复，在LSM树的日常使用中，需要记录一些用以恢复数据的日志。然而与以往数据库中的日志不同的是，日志中只需要要记录数据插入的事务。简单地说，这些日志只包含了被插入数据的行的号码及插入的域和值。

LSM树在记日志时设置检查点（checkpoint）以恢复某一时刻的LSM-tree，当需要在时刻T0设置检查点时：

- 完成所有组件的当前正在进行的合并，这样结点上的锁就会被释放；
- 将所有新条目的插入操作以及滚动合并推迟至检查点设置完成之后；
- 将C0写入硬盘中的一个已知的位置；此后对C0的插入操作可以开始，但是合并操作还要继续等待；
- 将硬盘中的所有部件（C1~CK）在内存中缓存的结点写入硬盘；
- 向日志中写入一条特殊的检查点日志。检查点日志的内容包括：
  - T0时刻最后一个插入的已索引的行的日志序列号（Log Sequence Number，LSN0）；
  - 硬盘中的所有部件的根在硬盘中的地址；
  - 各个部件的合并游标；
  - 新多页块动态分配的当前信息。在以后的恢复中，硬盘存储的动态分配算法将使用此信息判别哪些多页块是可用的。

一旦检查点的信息设置完毕，就可以开始执行被推迟的新条目的插入操作了。由于后续合并操作中向硬盘写入多页块时，会将信息写入硬盘中的新位置，所以检查点的信息不会被消除。只有当后续检查点使得过期的多页块作废时，检查点的信息才会被废弃。

6.2 恢复

当系统崩溃后重启进行恢复时，需要进行如下操作：

- 在日志中定位一个检查点；
- 将之前写入硬盘的C0和其它部件在内存中缓存的多页块加载到内存中；
- 将日志中在LSN0之后的部分读入内存，执行其中索引条目的插入操作；
- 读取检查点日志中硬盘部件（C1~CK）的根的位置和合并游标，启动滚动合并，覆盖检查点之后的多页块；
- 当检查点之后的所有新索引条目都已插入至LSM-tree且被索引后，恢复即完成。 这一恢复措施的唯一的一个缺点就是恢复的时间可能会比较长，但通常这并不严重。因为内存中的数据可以很快地写入硬盘。当两个相邻的部件进行滚动合并时，新产生的结点将会写入到硬盘中的新位置。这样在将合并产生的结点写入硬盘时，上层结点中指向该结点的指针需要更新为结点的新位置。当正在进行滚动合并，却临时需要设置检查点时，加载进内存的多页块和目录结点都会写入到硬盘中新的位置。这样，在高层的目录结点中指向这些结点的指针同样需要立即更新为硬盘中的新地址。在恢复的过程中需要注意的是目录结点的更新。

更进一步，当使用检查点进行恢复时，滚动合并所需的所有的多页块都会从硬盘重新读回内存，由于所有的多页块的新位置较之设置检查点时的旧位置都发生了改变，这样所有目录结点的指针都需要更新。这听起来似乎是一大笔性能开销，但这些多页块其实都已加载到内存里了，所以没有I/O开销。若要使得恢复的时间不超过几分钟，那么可以每隔几分钟的I/O操作就设置一次检查点。

结论

B树，因为它最常用的目录节点是可缓存在内存里的，所以实际上是一种混合数据结构：它结合了低成本磁盘和高成本内存，前者用来存放大多数的数据，后者为最热门的数据提供访问。 而LSM树将此层次结构扩展到多个层级，并在执行多页磁盘数据读取时结合了merge IO的优点。 下图展示了对于通过B树以及LSM树(仅包含内存中的C0和和磁盘上的C1树)的两种数据访问模式的数据热度，纵轴是访问开销/MB（横轴是插入速率/MB）

从上图可以得到以下结论：

- 最低的访问速率时（Cold Data的磁盘访问开销并太高
- 到Warm Data阶段（B树结构的数据访问成本开销急剧上升，此时磁盘臂会成为磁盘访问的主要限制因素；而LSM树结构的数据访问成本上升很缓慢
- 到了Hot Data阶段（B树结构的数据都应该缓存到内存中了，此时称之为沸点。使用内存缓存对B树来说效果显著，随着访问速率进入Hot Data区域而开销图形却变平缓，甚至更频繁的访问也不会导致更高的成本上升；而我们可以看出LSM树的作用是降低访问成本，对于任何实际访问速率的诸如插入和删除之类的可合并操作，特别是针对Cold Data（

此外，很多需要缓存B树的情况，如上图中的Hot Data阶段，其实可以用大部分驻留在磁盘的LSM树来代替。在这些场景中，由于LSM树的批处理效应，数据在逻辑访问速率方面是Hot的，但在磁盘物理访问的速率方面仅是Warm的。 对于具有大量可合并操作（写入、删除）的应用程序而言，这是一个非常重要的优势。

综合来看（LSM-tree的代价曲线在B-tree之下，可见LSM-tree对硬盘和内存的利用率都比B-tree要高。这主

要是因为两点：

- 滚动合并时批量写入新插入的数据，平摊单个条目插入的开销；
- C1的各层结点存储在多页块中，合并时顺序读写多页块，减小磁盘臂的移动。

好文推荐

网上有一些关于此论文的分析文章，在此推荐下：

- 日志结构的合并树 The Log-Structured Merge-Tree
- Paper笔记The Log structured Merge-TreeLSM-Tree

参考文档

英文原版论文地址：

The Log-Structured Merge-Tree (LSM-Tree)

因本人能力有限，还参考了这篇中文翻译版论文：

The Log-Structured Merge-Tree(译)

日志结构的合并树 The Log-Structured Merge-Tree