|  |  |
| --- | --- |
| **分 数：** |  |
| **评卷人：** |  |

****

**研究生（数据中心技术）课程论文（报告）**

**题 目：数据中心范围查询优化技术**

**学 号 M202173794**

**姓 名 杨劲帆**

**专 业 电子信息**

**课程指导教师 施展 童薇**

**院（系、所） 计算机科学与技术学院**

**2022年 1月6日**

数据中心范围查询优化技术

杨劲帆1)

1)(华中科技大学, 武汉 430074)

摘 要 随着范围查询需求的提升，范围查询的性能、成本与准确度显得愈发重要，在近几年的学术会议中，针对范围查询进行的优化也逐渐被重视起来。本文对提高LSM树的范围查询效率和ART的范围查询效率展开论述，就LSM树而言，对传统LSM树的范围查询必须动态地从多个表文件中查找和排序合并数据，这是昂贵的，而且通常会导致平庸的读取性能。为了提高LSM树的范围查询效率，引入了一种空间高效的KV索引数据结构，名为remix，它记录了跨越多个表文件的KV数据的全局排序视图。对多个混音索引数据文件的范围查询可以使用二进制搜索快速定位目标键，并按排序顺序检索后续键，而无需进行键比较。我们构建了RemixDB，一个基于LSM-tree的kv存储，它采用了写高效的压缩策略，并使用REMIX进行快速点和范围查询。就ART而言，本文提出了新的持久性索引ROART（基于自适应半径树），ROART(i)提出了一种叶压缩方法来减少对范围查询的指针追逐，(ii)通过三种优化减少持久性开销，即入口压缩、选择性元数据持久化和最小有序分割，(iii)设计了快速内存管理以防止内存泄漏，并通过提出即时重启策略消除了较长时间的恢复时间。

关键词 数据中心 范围查询 LSM树 ROART 压缩技术

Range query optimization techniques in data center

Jinfan Yang1)

1)( Huazhong University of Science and Technology, Wuhan 430074)

**Abstract** With the improvement of the demand for range query, the performance, cost and accuracy of range query become more and more important. In recent academic conferences, the optimization of range query has been paid more and more attention. This paper discusses how to improve the range query efficiency of LSM tree and art. As for LSM tree, the range query of traditional LSM tree must dynamically find and sort merged data from multiple table files, which is expensive and usually leads to mediocre reading performance. In order to improve the range query efficiency of LSM tree, a spatially efficient KV index data structure named remix is introduced, which records the global sorted view of KV data across multiple table files. For the range query of multiple mixed index data files, binary search can be used to quickly locate the target key and retrieve the subsequent keys in sorted order without key comparison. We build remixDB, a KV storage based on LSM tree, which adopts a write efficient compression strategy and uses remix for fast point and range queries. As far as art is concerned, this paper proposes a new persistent index row (based on adaptive radius tree). Row (I) proposes a leaf compression method to reduce pointer chasing for range queries, (II) reduces persistence overhead through three optimizations, namely entry compression, selective metadata persistence and minimum ordered segmentation, (III) designs fast memory management to prevent memory leakage, By proposing an immediate restart strategy, the long recovery time is eliminated.

**Key words** Data center Range query LSM-tree ROART Compaction technology

# 引言

数据中心是全球协作的特定设备网络，用来在因特[网络基础设施](https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E5%9F%BA%E7%A1%80%E8%AE%BE%E6%96%BD/5183560" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E4%B8%AD%E5%BF%83/_blank)上传递、加速、展示、计算、存储数据信息。随着数据中心行业在全球的蓬勃发展，随着社会经济的快速增长，数据中心的发展建设将处于高速时期，再加上各地政府部门给予新兴产业的大力扶持，都为数据中心行业的发展带来了很大的优势。随着数据中心行业的大力发展，将来在很多城市中都会有很大的发展空间，一些大型的数据中心也会越来越多。2017年全球经历了前所未有的自然灾害之后，很多数据中心管理人员都在积极制定灾难恢复计划。例如就可以通过云计算工具对电力使用的功率进行限制，在遭遇停电时间时将允许以降低的功率继续运行，可以为电力企业的正常运行提供有效的保障。还可以利用数据中心指定备份计划，对服务器的操作进行拓展，就不需要通过关闭和重启服务器操作。

数据中心是与人力资源、自然资源一样重要的战略资源，在信息时代下的数据中心行业中，只有对数据进行大规模和灵活性的运用，才能更好的去理解数据，运用数据，才能促使我国数据中心行业快速高效发展，体现出国家发展的大智慧。海量数据的产生，也促使信息数据的收集与处理发生了重要的转变，企业也从实体服务走向了数据服务。产业界需求与关注点也发生了转变，企业关注的重点转向了数据，计算机行业从追求的计算能力转变为数据处理能力，软件业也将从编程为主向数据为主转变，云计算的主导权也将从分析向服务转变。

随着数据量日益增大，查询搜索功能的重要性日益增强。范围查询所要消耗的成本、范围查询系统的性能、查询结果的正确性等问题都需要纳入重点考虑范围之内。

对于ROART而言，(i)功能：可变大小的密钥和范围查询。可变大小的键在现实世界的系统中很重要，比如RDBMS和键值存储在本文中，我们将重点关注同时支持可变大小的键和范围查询的索引结构。(ii)性能：持久性的开销。持久性开销在针对NVM的索引性能中起着至关重要的作用。为了保证索引的崩溃一致性，一旦操作完成，就必须通过缓存线刷新和内存围栏指令将修改持久化到NVM中。由于硬件的设计，NVM写操作的吞吐量低于读取，带宽的可伸缩性较差。此外，持久性操作所产生的开销比正常的写操作要大得多。(iii)正确性：异常分辨率和内存安全。首先，如果持久性索引没有对并发操作提供保护，则可能会出现异常，如丢失更新和脏读取。这些异常将导致成功操作的影响在系统崩溃和重启后消失。其次，NVM中的内存分配需要处理崩溃一致性，这在DRAM中不是问题。

对于LSM树而言，LSM 代表的是更新开销和读开销的一种权衡，相比于 B+tree 保证了更小的写开销以及更大的读开销。关于读开销的优化就有，驻留在内存中的为每个 Table 维护的 BloomFilters，来减小不必要的 Table 访问。但是 BloomFilter 不能处理范围查询，所以出现了 Range Filter，如SIGMOD18-SuRF、SIGMOD20-Rosetta 来在范围查询时过滤掉 Tables。但是当范围查询内的 Key 位于很多个候选的 Tables 中时，filtering 就很难提升查询性能了，特别是大范围查询，而且当查询请求可以在缓存中处理时，访问 Filters 的计算开销可能导致性能表现一般，这是真实负载中比较普遍的情况。LSM 本身是有 Compaction 来减少查询时检索的 SSTable 数量的，选择 Table 的策略又分为了 leveled 和 tiering。问题的关键其实在于限制 sorted runs 的数量以及 KV 存储不得不归并排序且重写现有的数据。如今的硬件技术使得随机访问的效率也很高了，因此 KV 存储不再说必须保证物理上的有序，而可以只保证逻辑有序同时避免大量的重写。为此，我们设计了REMIX，现有的范围查询解决方案很难在物理重写数据和动态执行昂贵的排序合并之间进行改进，与此不同的是，REMIX 使用了一个空间效率高的数据结构来记录跨多个表文件的 KV 数据的全局排序视图。使用 REMIX，基于 LSM 树的 KV-store 可以利用高效写压缩策略，而不会牺牲搜索性能。基于此我们还构建了 RemixDB，和高效的 Tiered 压缩策略以及分区的布局集成，同时实现了较低的写放大和快速的查询。

# 原理

## ART

ART是一种空间效率高的基树，它的基数是256，每个节点代表Key的一个1字节字符。

Path Compression：根树的高度可以通过Path Compression来降低，如图6所示。只有一个子节点的节点被合并到它的子节点中，它所代表的字符被合并到它的子节点的前缀中。

Node Split：使用Path Compression，节点分割可能会在插入期间发生。节点拆分分为内部节点拆分和叶节点拆分两类，如图7所示。当新的插入(例如，L3)与内部节点(旧)的前缀不匹配时，就会发生内部节点分裂。新节点被创建并插入到树中，它指向L3和old。old前缀将被更新。当新的插入（L3）不匹配叶节点（L2）中的键时发生叶节点分割。将创建指向L2和L3的新节点并插入树中。

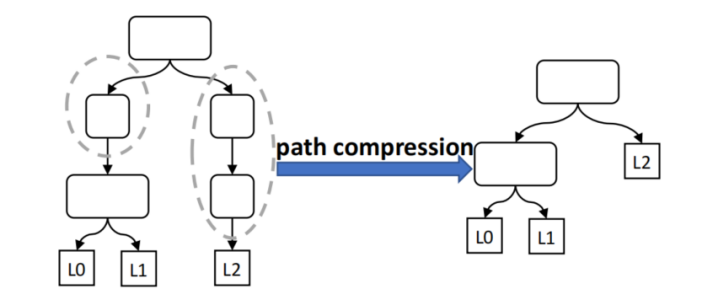


图1：在radix树中的路径压缩

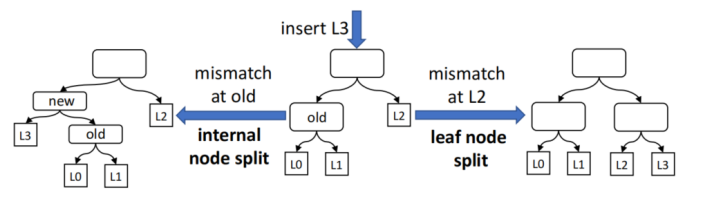


图2：有两种节点分裂的方式

与B+-Trees相比，基数树在范围查询中表现较差，因为每个叶节点只存储一对键和值，而且叶节点可能位于树中的许多不同级别。范围扫描除了需要访问叶节点外，还需要访问不同层次上的大量非叶节点，这将带来巨大的开销。ROART提出叶压缩(LC)，将叶节点的指针压缩到根树中的叶数组中。一个叶数组最多可以包含m个叶指针。如果根树的子树（PACTree只是将所有叶子节点放在B+Tree-like的叶子节点，并没有压缩Trie子树结构,但PACTree只索引Anchor Key，可以减少索引的大小）有小于或等于m个叶子节点，则将该子树压缩为一个叶子数组。(在实现中设置m = 64)

图3(a)和图3(b)显示了具有相同叶节点的ART和ROART之间的结构差异。

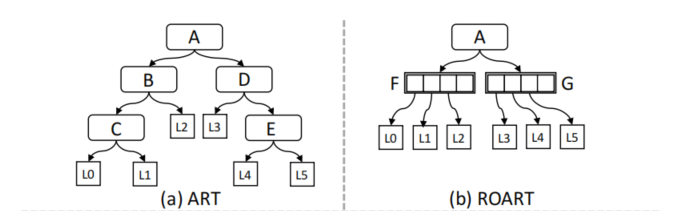


图3

## LSM-tree

LSM-tree被设计用于在持久性存储设备上的高写效率。它通过缓冲内存中一个称为MemTable的结构中的所有更新来实现高速写操作。当MemTable填充时，缓冲键将被排序，并作为一个称为小压缩的进程排序运行。小的压缩是写效率的，因为更新是按批编写的，而不会与存储中的现有数据合并。由于排序的运行可能有重叠的键范围，点查询必须检查所有可能的运行，从而导致较高的搜索成本。为了限制重叠运行的数量，LSM-tree使用一个主要的压缩过程将几个重叠的运行排序合并为更少的运行。

LSM 代表的是更新开销和读开销的一种权衡，相比于 B+tree 保证了更小的写开销以及更大的读开销。

## Minor compaction

一个minor compaction将新的KV数据从不可变的MemTable写入到一个分区中，而无需重写现有的表文件，并重建分区的重新混合。根据新数据的大小，一个较小的压缩会创建一个或几个新的表文件。当压缩后的表文件的预期数量（现有表文件的数量加上新表文件的估计数量）低于阈值T，该阈值为10时，将使用小压缩。图4显示了一个创建一个新表文件的minor compaction示例。

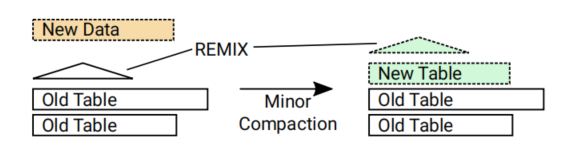


图4：Minor compaction

## Major compaction

当分区中预期的表文件数量超过阈值t时，需要进行major compaction。Major compaction排序将现有的表文件合并到更少的表文件中。通过减少表文件的数量，将来可以执行较小的压缩。主要压缩的效率可以通过输入表文件的数量与输出表文件的数量的比值来估计。图5显示了major compaction示例。在本示例中，新数据与三个小表文件合并，并且在压缩后只创建一个新表文件（比率为=3/1）。如果整个分区是排序合并的，压缩需要重写更多的数据，但由于表文件的大小限制，仍然会产生三个表（比率=5/3）。因此，major compaction选择可以产生最高比率的输入文件的数量。

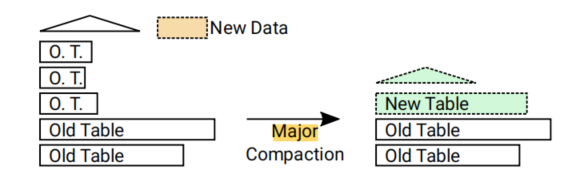


图5：Major compaction

## REMIX

对多个排序运行的范围查询操作可以动态地构建底层表的排序视图，以便可以按排序顺序检索键。实际上，排序视图继承了表文件的不变性，并在删除或替换任何表之前保持有效。然而，现有的基于LSM-tree的KV存储并不能利用这种遗传的不变性。相反，在搜索时重复重构排序后的视图，然后立即丢弃，导致过度计算和I/O，导致搜索性能不佳。混音的动机是利用表文件的不变性，通过保留底层表的排序视图来重复利用表文件的不变性。

为了提高I/O效率，基于LSM-tree的KV存储采用了内存高效的元数据格式，包括稀疏索引和Bloom过滤器。如果我们记录每个键及其位置以保留存储中排序的视图，存储的元数据可能会大大膨胀，导致读写的性能下降。为了避免这个问题，重新混合的数据结构必须具有空间效率。

# 研究进展

## ROART结构

对于查找，Reader搜索树，直到它到达叶子数组。它必须检查叶子数组点的每个叶节点，为了最大限度地减少这个开销，ROART将每个叶键的16位指纹（哈希值）嵌入到叶数组中的指针中 （即|fingerprint:16-bit|address:48-bit| ）.

对于插入，当它到达叶数组时，Writer检查Key是否已经存在。如果不存在，Writer选择一个空槽插入，如图6所示。复杂的情况是，当叶数组满时，叶数组分裂，如图7所示。

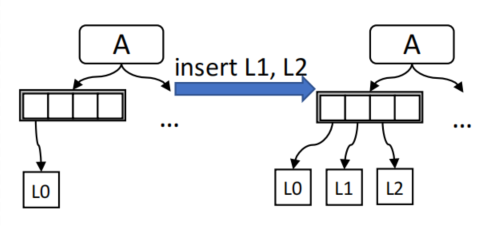


图6：ROART中的插入操作

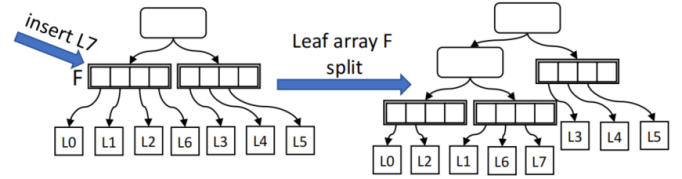


图7：ROART中的叶压缩

这里，Writer想要将一个新的叶子L7插入到已经满了的叶子数组F中。注意，F中的所有键对应于原始基数树中的一个子树，因此共享一个公共前缀。要进行分割，需要找到第一个字节的位置，表示为P，也就是键分叉的位置，称为标识字节。根据Key的标识字节将其划分为子集。为每个子集（这里与PACTree不一样，PACTree将DataNode根据Key拆分成大小相同的两部分，因此两者的查找也略有不同，PACTree会首先检查叶节点的anchor Key，Anchor Key是创建数据节点时数据节点的最小键。在创建数据节点之后，它不会改变。根据比较结果去查找当前叶节点的左右兄弟）构建一个叶数组，并创建一个新的内部节点，其中包含每个标识字节和指向相关叶数组的指针。例如，在图8中，一个包含四个叶指针的叶数组被拆分为三个新的叶数组，Key在第5字节发散，即P = 5。先划分键，然后创建新的叶数，注意，拆分叶数组的代价很高，但幸运的是，这是一种罕见的操作。

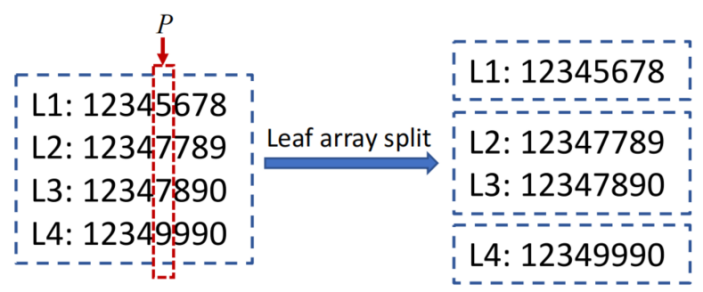


图8：叶数组分裂的一个例子

对于更新和删除，过程类似于查找。在找到匹配的键后，将执行相应的修改或删除。

对于范围查询，与ART的唯一区别是Key在叶数组中不是排序的，但键在叶数组之间排序。因此，只需要检查/排序开始和最后一个叶数组，以确保返回值在请求的范围内。如果值在返回之前需要完全排序，那么通过一些优化，例如跳过键的前缀，只比较键的不同部分，只会导致8.9%的性能下降。

有趣的是，叶压缩（Leaf Compression）不仅可以改进范围查询，还可以改进遍历和插入。遍历是所有操作(查找/插入/更新/删除/扫描)中必不可少的步骤。LC会降低根到叶片的路径长度，如图3(a)和(b)所示。较短的路径长度有利于遍历。此外，当新的插入与叶子节点中的Key不匹配时，ART会导致叶子节点分裂(图2)，而ROART只是将新的叶子插入到叶子数组中，减少了持久指令的数量。(具体统计见表1)

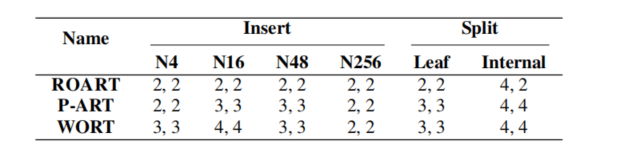


表1：持久性分析，两个值分别是clwb和sfence的数量（越低即好）

总之，LC有几个好处:(i)减少范围查询的指针追逐（point chasing）次数，(ii)减少根到叶路径长度和遍历开销，(iii)减少插入的持久性开销。

## 减小持续开销

图9是ART中可以存储4，16，64，256个entry的四种类型的节点。每个entry包含一个字节和一个子指针。在N4和N16中，字节和指针分别存储在字节数组和指针阵列中。在N48中，Child Index有256个slot，因此可以直接使用byte作为索引（即child\_index[byte]，不用在进行比较）以查找child point所在pointer array的位置，N256直接有一个带有256个指针的数组。当节点满时将扩展为更大的节点类型，并插入新条目，当条目数量低于阈值时将收缩为更小的节点类型。

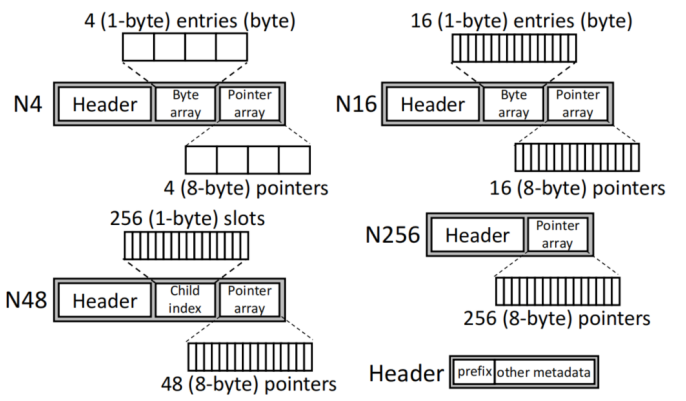


图9：ART中的节点类型

图10是ROART的节点结构，提出了Entry Compression（EC）将Key byte打包到N4，N16和N48中的指针（其中48位用于指针）因此ROART结构位｜empty: 8-bit | key: 8-bit | pointer: 48-bit｜，成为Zentry。由于Zentry是8字节的，与ART相比，它可以进行原子更新，因此EC减少了用于持久保存ROART中的每个entry的一条持久指令（不用持久化Child Index/Byte Array）。

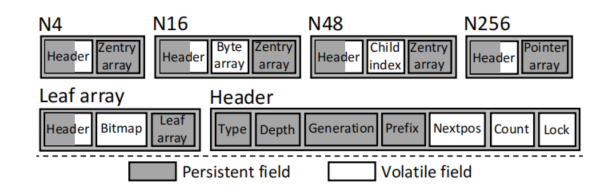


图10：ROART中的节点结构

## 选择性元数据持久性

并非所有元数据都需要持久化: 例如，Header中的Nextpos和Count表示下一个空slot和已使用slot的数量。叶数组中的位图显示了哪个叶数组条目正在使用中。它们都可以通过扫描Zentry或指针数组来计算，其中slot被设置为0。此外，N16的byte array被用来加速SIMD指令的搜索。可以通过从每个Zentry检索嵌入（packed）的Key byte来重新构建它。N48中的Child索引也可以用同样的方法恢复，最后，Lock用于并发控制，可以在崩溃恢复时清除。

基于这一观察提出了选择性元数据持久化(SMP)，以选择性地持久化元数据的一个子集，并在恢复后重新计算其余的元数据。如图11所示，volatile元数据以白色背景突出显示。背景为灰色的字段将被持久化。

传统的恢复需要暂停处理请求并扫描整个索引，随着数据量的增加，这会导致恢复时间变长。受NV-Heaps中的generation lock的启发，使用generation来实现选择性的持久元数据，以隐藏恢复开销。ROART在NVM中维护一个全局代数(GGN)。GGN在每次重启时增加。ROART中的每个节点都有自己的持久节点生成编号(NGN)。当访问时，如果NGN = GGN，则表示元数据在节点中是最新的。否则，恢复节点内的元数据后，将GGN赋值给NGN。用于恢复的Per-node Latch(通过在内存和CAS指令中使用标志实现)保护对同一节点的并发访问不受多个线程的影响。这样，在重新启动后，ROART不会暂停恢复整个丢失的元数据的正常操作。相反，它会按需恢复元数据，同时执行正常操作。

## 最低限度地次序分割

内部节点分割代价很高，如图7所示。它有四个步骤:(i)分配一个新的叶子L3， (ii)分配一个内部节点，标记为new，有两个子节点(L3和节点old)， (iii)改变父节点的指针(从旧的到新的)，(iv)更新旧的前缀。如果没有优化，这四个步骤需要四个sfence指令。

我们观察到这四个步骤的顺序可以放宽。Step (i)和Step (ii)对其他线程是不可见的，在初始化两个节点后可以使用一个sfence。步骤(iii)和(iv)不能原子执行，在并发执行下，Reader可能会看到不完整的分割与不一致的前缀。注意，节点的深度(包括前缀)保持不变，可以利用这个属性来检测不一致的前缀，一旦检测到这种不一致，就很容易通过重新计算前缀来修复不一致。因此，步骤(iii)和步骤(iv)的顺序并不重要。

ROART按照如下方式执行内部分割。它执行步骤(i)， (ii)和(iv)，刷新修改的缓存行，然后调用单个sfence。之后，它执行步骤(iii)，刷新修改后的缓存行，并调用第二个sfence。通过这种方式，ROART将内部拆分的sfence指令数量从4条减少到2条。

## Remix数据结构

图11很好理解，简单解释一下概念。下图所示例子包含了三个 sorted runs，顺序对应地由箭头表示，共计 15 个 Keys，为了构建 REMIX，首先进行了个划分，划分为一定数量的 segments，每个分段包含的 Keys 数量相等。每个 Segment 对应包含一个起始 Key，也就是 Anchor Key，包含一组游标偏移，对应就是记录每个 Sorted Runs 现在的指针位置，也就是大于等于起始锚点的指针位置，还有一个 Run Selectors，包含了真正的顺序，即下一个 Key 所在的 runs 编号。

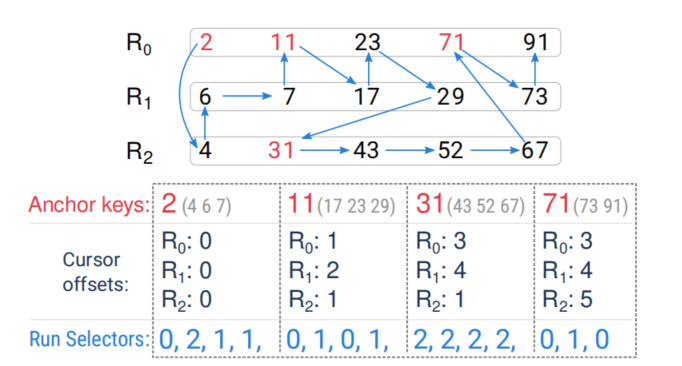


图11：使用REMIX的三个排序运行的排序视图

范围查询的过程就变成了：

(1)首先检索稀疏索引，也就是 Anchor Key，二分查找找到所属的 segment。

(2)迭代器对应被 seek 到该 segment 对应的 Anchor Key，然后使用该 segment 的游标偏移来移动迭代器指针，根据 selectors 中的顺序来进行 seek。

(3)最后通过在全局有序的视图上找到了对应的目标 Key。

例子：图示中找到 Key 17，首先二分找到 Segment2。然后游标从 11 开始移动，根据 Seletors 发现 11 在 R0 上，以及 Offset 的结果，即在 R0 的索引为 1 的地方开始，因为 11 < 17，那么要继续移动游标，直到找到大于等于 17 的 Key，也就是图中的 17，这时候 offset 变成了 2 2 1，根据 Selectors 那么找到了 R1，根据 offset 也就找到了 17。

# 折半查找

为了在一个段中执行二分搜索，我们必须能够随机访问段中的每个键。段中的键属于 run，由相应的 run 选择器指示。要访问一个键，我们需要将 run 的游标放在正确的位置。这可以通过计算相同的 run 选择器在键之前的段中出现的次数，并将相应的游标向前移动相同的次数来实现。可以使用现代 CPU 上的 SIMD 指令快速计算出现次数。搜索范围可以通过对段的一些随机访问快速缩小，直到识别出目标键。为了结束查找操作，我们使用每个 run 选择器在目标键之前出现的次数初始化所有游标。

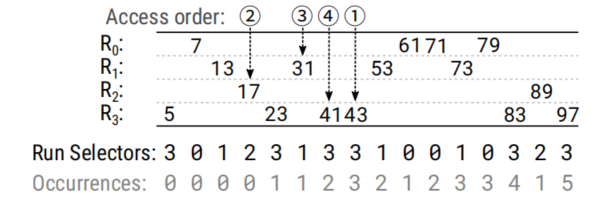


图12：在段中的二进制搜索的一个例子，圈出的数字表示键的访问顺序

图12显示了一个包含16个运行选择器的段的示例。每个运行选择器下面显示的数字表示同一运行选择器在其位置之前出现的次数。例如，41是这个段中R3中的第三个键，因此相应的出现次数是2次（在第三个“3”下）。为了访问键41，我们初始化R3的光标，并推进两次以跳过5和23。

为了在图4的段中寻找键41，在二进制搜索过程中依次访问键43、17、31和41，如箭头、圈出的数字所示。键43是段中的第八个键，也是段中R3的第四个键。为了访问键43，我们初始化R3的光标，并推进它三次，以跳过键5、23和41。然后，可以通过读取该段中R2上的第一个键来访问键17。类似地，31和41分别是R1和R3上的第二和第三键。最后，迭代器的所有游标都被初始化为指向正确的键。在这个示例中，游标最终将出现在键61、53、89和41处，其中41是当前键。

# I/O优化

执行段内二分查找当然是为了减少比较的次数，但是，搜索路径上的键可能驻留在不同的 runs 中，如果各自的数据块没有被缓存，则必须通过单独的 I/O 请求来检索。上图所示就比较了四次 Key，对应访问了 3 个 runs，但是像 41 43 这两个 Key 其实属于一个 Run，甚至可能属于一个数据块。因此，在进行键比较之后，搜索可以利用相同数据块中的其余键，在必须访问不同的 Run 之前进一步缩小搜索范围。这样，R3 中的每个键都可以在不访问任何其他 Run 的情况下找到。比如为了查找 79，访问 R3 可以将搜索范围缩小到键 43 和键 83 之间，这时候二分查找找到了 71，也就是 R0，然后在 R0 中找到了 79。

## 搜索效率

REMIXes在以下三个方面改进了范围查询。

**REMIXes使用一次二分查找找到目标密钥：**一个REMIX提供了多个排序运行的排序视图。在多次运行中，只需要进行一次二分查找就可以将游标定位在目标键上。而在传统的基于LSM树的KV-store中，查找操作对每次运行都需要进行大量的二分查找。例如，假设具有四个相同大小运行的存储在每次运行中有N个键。没有混音的搜索操作需要4×log2N键比较，而它只需要log24N，或2+log2N键比较。

**REMIXes移动迭代器而不进行键比较：**通过使用预先录制的运行选择器更新游标和当前指针，混合音上的迭代器直接切换到下一个（或前一个）kv对。这个过程不需要任何关键的比较。如果迭代器跳过该密钥，也可以避免读取KV对。相比之下，传统的基于LSM树的KV-store中的迭代器维护一个最小堆，以从多个重叠的排序运行中的键进行排序。在此场景中，下一个操作需要从多个运行中读取键以进行比较。

**REMIXes跳过不在搜索路径上的运行：**具有混音的搜索操作需要在目标段中进行二进制搜索。只有那些在搜索路径上包含键的排序运行将在搜索时被访问。在最好的情况下，如果一个目标键范围驻留在一次运行中，例如图11中的段（31,43,52,67），则只访问一次运行(示例中的R2)。但是，合并迭代器必须访问查找操作中的每一次运行。

此外，大幅降低的寻路成本允许在不使用Bloom过滤器的情况下，对多个已排序的运行进行有效的点查询(例如，GET)。

## REMIX存储开销

Remix元数据由三个组件组成：锚定键、光标偏移量和运行选择器。我们将D定义为段中键的最大数。remix为每个D键存储一个锚键，平均需要总键大小的1/D。假设光标偏移量的大小为S字节，混合需要S×H字节来存储每个D键的光标偏移量，其中H表示由混合索引的运行次数。运行选择器需要位。将所有这三个部分加到一起，一个REMIX将存储字节/键，其中是锚键的平均大小。

我们使用在脸书的生产KV工作负载中公开报告的平均KV尺寸来估计一个REMIX的存储成本。在实践中，S是由实现定义的，而H取决于被建立索引的表的数量。在估计中，我们使用4字节的光标偏移量(S=4)，这样每次排序运行的光标偏移量就可以解决4GB的空间。我们将排序运行的次数设置为8(H=8)。使用这些实际的配置，一个REMIX存储字节/密钥。

表2显示了具有不同D(D=16、32、64)的每个工作负载的重新混合存储成本。为了进行比较，它还显示了LevelDB和RocksDB中SSTable格式的块索引(BI)和Bloom过滤器(BF)的存储成本。请注意，由REMIXes索引的表文件不使用块索引或Bloom过滤器。SSTable为每个4KB的数据块存储一个密钥和一个块句柄。块指数存储成本是通过将平均KV大小和一个近似的块手柄大小(4B)的总和除以一个4KB块中KV对的估计数量来估计的。Bloom滤波器估计为10位/键。对于不同的D和值，重新混合的存储成本从1.0到5.4字节/键不等。对于每一个键的大小，增加D可以大大降低混音存储成本。最后一列显示了一个混音与其索引KV数据的大小比。在最坏的情况下(USR商店)，混音器的大小仍然小于KV数据大小的10%。

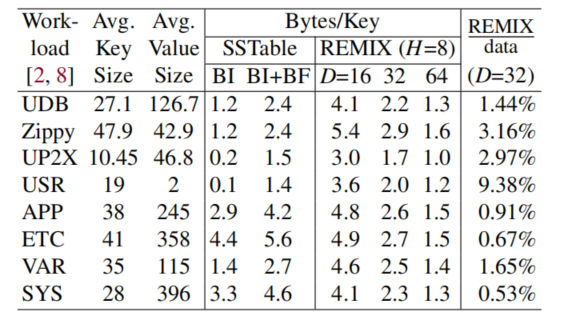


表2：REMIX存储成本与真实世界的KV尺寸，BI代表方块指数，BF代表了Bloom Filter最后一列显示了混音与KV数据的尺寸比

## RemixDB

RemixDB 使用了 Tiered Compaction 加一个分区的布局。因为有很多研究表明真实负载大多有很强的空间局部性，然后分区存储的话可以很好地降低压缩开销。所以 RemixDB 将键空间划分为不重叠键范围的分区。REMIX 对每个分区中的表文件进行索引，提供分区的排序视图。通过这种方式，RemixDB 本质上是一个使用分层压缩的单层 LSM 树。RemixDB 不仅继承了 Tiered 压缩的写效率，而且在 REMIX 的帮助下实现了高效的读取。RemixDB 的点查询操作(GET)执行一个 seek 操作，如果它与目标键匹配，则返回迭代器下的键。RemixDB 不使用Bloom过滤器。

下图展示了 RemixDB 的结构，内存组件和 LevelDB 这些是一样的。分区中的压缩创建分区的新版本，其中包含新旧表文件的混合以及一个新的 REMIX 文件。旧版本在压缩后进行垃圾收集。

在多级 LSM-tree 设计中，MemTable 的大小通常只有几十 MB，接近于默认的 SSTable 大小。在分区存储布局中，较大的 Memtable 在触发压缩之前可以积累更多的更新，这有助于减少 WA。MemTables 和 WAL 的空间成本几乎不变，考虑到当今数据中心的大内存和存储容量，这是合理的。在 RemixDB 中，MemTable 的最大大小被设置为 4GB。

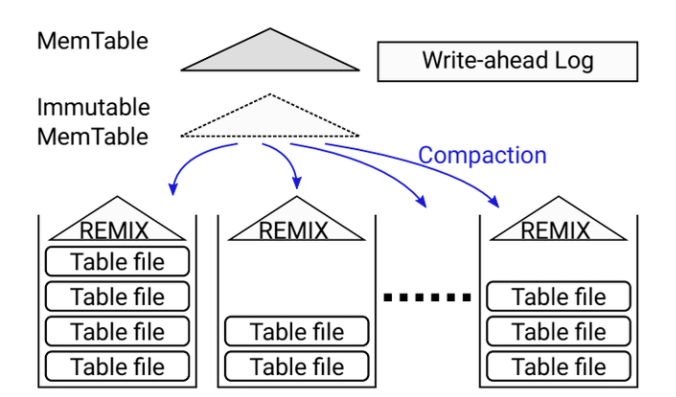


图13：RemixDB概述

# 总结与展望

本次主要调研了三篇近三年CCF/A类文献。第一篇《ROART: Range-query Optimized Persistent ART》是FAST’2021的一篇文章。Ma等[1]人首先分析了功能、性能和正确性等三个实践方面。然后提出了几种优化的ROART，即(i)叶压缩、(ii)入口压缩、(iii)选择性元数据持久性、(iv)最小有序分割和(v)即时重启。最后，评估表明，在各种工作负载下，ROART可以优于其他最先进的索引。

第二篇《REMIX: Efficient Range Query for LSM-trees》是FAST’2021的一篇文章。Zhong[2]

等人介绍了remix，这是一种紧凑的多表索引数据结构，用于LSM树中的快速范围查询。其核心思想是记录多个表文件的全局排序视图，以便有效地进行搜索和扫描。基于Remixes，RemixDB使用分层压缩有效地提高了范围查询性能，同时保持了低写放大。最后，他们还注意到，他们的经济方法可以应用于除分析数据库查询处理工作负载之外的其他领域。例如，机器学习模型的训练可以使用类似的方法，使用一个关注浮点计算的模型进行优化。在其他领域探索我们的基于成本的方法将是未来工作的一个有趣的途径。

第三篇《Towards Cost-Optimal Query Processing in the Cloud》是VLDB’2021的一篇文章，Leis等[3]人提出了一种直观的云中分析查询处理的模型，该模型估计了特定硬件实例上的工作负载的运行时和成本。该模型最重要的应用程序是做出数据驱动的架构设计决策，发现性能瓶颈，以及朝着云中成本最优查询处理的目标发展数据管理系统。

这三篇文章对于范围查询优化都做了或多或少的贡献，范围查询的性能、成本、准确度等方面经过学者们的研究得到了不小的提升，相信未来定会有更多好的途径来进一步优化范围查询功能。

# 参考文献

1. Shaonan Ma , Kang Chen1, Shimin Chen , Mengxing Liu, Jianglang Zhu , Hongbo Kang , and Yongwei Wu. ROART: Range-query Optimized Persistent ART[C]// 19th USENIX Conference on File and Storage Technologies (FAST ’21).2021:1-16.
2. Wenshao Zhong，Chen Chen，Xingbo Wu，Song Jiang. REMIX: Efficient Range Query for LSM-trees[C]//19th USENIX Conference on File and Storage Technologies (FAST’21).2021:51-64.
3. Viktor Leis, Maximilian Kuschewski. Towards Cost-Optimal Query Processing in the Cloud[C]// PROCEEDINGS OF THE VLDB ENDOWMENT, no. 9 (2021): 1606-1612