LSM-tree查询优化的研究

徐子清[1)]

[1)]华中科技大学计算机科学与技术学院, 武汉市430074

摘 要 在信息化进程中，“大数据”的局限性一直是业界普遍关注的问题，键值存储是许多云和数据中心服务的支柱，其中日志结构合并树（LSM-tree）是键值存储中的最先进的技术，它被广泛应用于现代NoSQL系统的存储层。因此，从数据库社区和操作系统社区都进行了大量的研究工作，试图改进日志结构合并树的各个方面。日志结构合并树是许多KV 存储的核心数据结构，针对写入繁重的工作负载进行了优化，可以通过消除随机I/O来提高写入性能，但它会降低读取性能，这就使得日志结构合并树的查询效率较低。大多数现有的索引解决方案都注重提高写入性能，而牺牲读性能。为了充分利用带宽并应对随机I/O，需要适当的外部存储器数据结构，比如索引来实现高效查询。围绕优化LSM-tree 中的查询效率，本文介绍了LSM-tree以及在LSM-tree中查询优化策略，重点介绍了Monkey、REMIX和BOURBON这三种优化策略，以及其他类似或相关的优化策略，并进行了比较。其中Monkey使用布鲁姆过滤器加速点查询，并为了减少布鲁姆过滤器的误报率而做了改进；REMIX通过建立一个全局有序视图，重点优化了LSM-tree的范围查询，对点查询也有优化；BOURBON 利用机器学习，在LSM-tree中引入学习索引，提升查询效率。

关键词 日志结构合并树；查询优化；点查询；范围查询；布鲁姆过滤器；学习索引

中图法分类号 TP DOI号:

Research of Query Optimization on LSM-Trees: A Survey

Ziqing Xu[1)]

[1)]Huazhong University of Science and Technology, College of Computer Science and Technology, Wuhan, 430074

Abstract

In the process of informatization, the limitation of ”big data” has always been a common concern in the industry. Key-value storage is the backbone of many cloud and data center services, among which log-structured merge tree (LSM-tree) is a key-value storage. State-of-the-art, it is widely used in the storage layer of modern NoSQL systems. As a result, there has been a lot of research work from both the database community and the operating system community in an attempt to improve various aspects of log-structured merge trees. The log-structured merge tree is the core data structure of many KV stores, optimized for write-heavy workloads, and can improve write performance by eliminating random I/O, but it reduces read performance, which makes log-structured Merge tree queries are less efficient. Most existing indexing solutions focus on improving write performance at the expense of read performance. To fully utilize bandwidth and cope with random I/O, appropriate external memory data structures, such as indexes, are required for efficient querying. Focusing on optimizing the query efficiency in LSM-tree, this paper introduces LSM-tree and query optimization strategies in LSM-tree, focusing on three optimization strategies of Monkey, REMIX and BOURBON, as well as other similar or related optimization strategies, and comparisons were made. Among them, Monkey uses the Bloom filter to speed up point queries, and has made improvements to reduce the false positive rate of the Bloom filter; REMIX focuses on optimizing the range query of LSM-tree by establishing a global ordered view, and there are also some point queries. Optimization; BOURBON uses machine learning to introduce learning indexes into LSM-tree to improve query efficiency.

Keywords Log-Structured Merge Tree; Query Optimization; Point Query; Range Query; Bloom Filter; Learned Index

1 引言

在信息化进程中，无论是个人还是组织，数据量都在高速增长。数据在各个行业和业务功能领域都发挥了重要作用，将我们带入了数据时代。因此，“大数据”存在的局限性一直是业界普遍关注的问题。大数据管理和分析的要求与传统数据管理有很大不同，海量数据需要保持在线进行查询和分析，此外，需要立即响应查询，以实现实时分析和决策。

键值存储是一个数据库，它可以有效地将搜索键映射到它们对应的数据值，KV存储是许多云和数据中心服务的支柱，包括社交网络[1]、实时分析[2]、电子商务[3] 和加密货币[4]。对于写入密集型工作负载，基于Log-Structured Merge-Trees (LSM-trees)[5]的键值存储已成为最先进的技术。LSM-tree 是许多KV 存储的核心数据结构。与需要在磁盘上进行就地更新的传统存储结构（例如B+- 树）相比，LSM- 树遵循支持高速顺序写入I/O 的就地更新方案。基于LSM 树构建的各种分布式和本地存储广泛部署在大规模生产环境中，例 如Google 的BigTable[6]和LevelDB，Facebook的Cassandra [7]、HBase[8]和RocksDB等 。LSM-tree相对于其他索引结构（例如B 树）的主要优势在于它们维护写入的顺序访问模式。B 树上的小更新可能涉及许多随机写入，因此在固态存储设备或硬盘驱动器上效率不高。LSM-tree在内存中缓冲更新并定期将它们刷新到持久存储以生成不可变的表文件。这会降低搜索效率，因为范围内的键可能驻留在不同的表中，由于计算和I/O 成本高，可能会减慢查询速度。基于LSM-tree 的设计代表了更新成本和搜索成本之间的权衡[9]，与B+-tree 相比，更新成本较低，但搜索成本要高得多。

对于LSM-tree，有两种查询方式，分别为点查找和范围查找。点查找通过从最小到最大遍历级别来查找条目的最新版本，并在从最年轻到最旧的级别内分层run。当它找到具有匹配键的第一个条目时终止。范围查找。范围查找必须找到目标键范围内所有条目的最新版本。它通过对所有级别的所有run的相关键范围进行排序合并来实现这一点。在排序合并时，它会在不同的run中识别具有相同键的条目并丢弃旧版本。

LSM-tree 在主内存中缓冲插入/更新的条目，并在填满时将缓冲区作为有序的run刷新到辅助存储。然后LSM-tree 对这些run进行排序合并，以限制查找必须探测run的次数并删除过时的条目，即存在具有相同键的更新条目。LSM-tree 将run组织成指数增加的容量级别，其中较大的级别包含较旧的run。当条目被异地更新时，点查找通过从最小到最大探测级别并在找到目标键时终止来找到条目的最新版本。另一方面，范围查找必须从所有级别的所有run 中访问相关键范围，并从结果集中消除过时的条目。为了加快单个run的查找速度，现代设计在主存储器中保留了两个额外的结构。首先，对于所有run，都有一组栅栏指针，其中包含run的ÿ个块的第一个键；这允许查找在一个run中通过一个I/O 访问特定键。其次，ÿ个run都有一个Bloom过滤器；这允许点查找跳过不包含目标键的run。

LSM-tree 针对写入繁重的工作负载进行了优化。这是当今系统的一个重要性能目标，因为应用程序写入的比例不断增加。为了优化写入，LSM-tree 最初在主内存中缓冲所有更新、插入和删除，如图1所示。当缓冲区填满时，LSM-tree 刷新缓冲到二级存储作为有序的run。LSM-tree 归并排序run是为了限制查找必须在辅助存储中访问的run的次数，以及删除过时的条目以回收空间。它将run组织成L 个大小呈指数增长的概念级别。级别0 是主内存中的缓冲区，属于所有其他级别的run在辅助存储中。

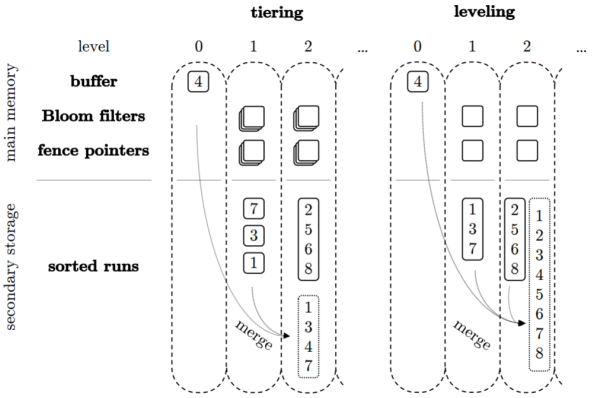

图1 LSM-tree结构

基于LSM-tree的键值存储中的合并控制了更新、点查找、范围查找和空间放大成本之间的内在权衡。可以通过调节两个参数调整合并的I/O 成本与查找和空间放大的I/O 成本之间的平衡。第一个是相邻level容量之间的大小比T；T 控制LSM-tree的级别数，从而控制条目跨级别合并的总次数。第二个是合并策略，它控制条目在一个级别内合并的次数。今天的所有设计都使用以下两种合并策略之一：分层（tiered）或分级（leveled）。通过分层，仅在级别达到容量时才在级别内合并run。只要有新的run进入，就会在一个级别内合并run。在这两种情况下，合并都是由缓冲区刷新触发的，并导致级别1 达到容量。使用分层时，第1 级的所有run都合并到放置在第2 级的新run中，合并还包括第2 级

的预先存在的run。

本文主要介绍了日志结构合并树的特征以及在LSM-tree中进行查询的优化策略，重点介绍了Monkey[10]、REMIX[11]和BOURBON[12]这三种优化策略，以及其他类似或相关的优化策略。本文安排如下，第二章介绍了索引优化策略的原理和优势，第三章对几种优化策略进行了介绍和对比，最后一章总结了全文并对未来的研究方向进行了介绍。

2 原理和优势

LSM-tree是实现写入优化的通用模型，但是LSM-tree 牺牲读取性能以提高写入吞吐量。基于LSM 的技术的成功与其在经典硬盘驱动器(HDD)上的使用密切相关。在HDD 中，随机I/O 比顺序I/O 慢100 倍以上；因此，执行额外的顺序读取和写入以连续排序键并启用高效查找是一个很好的替代方案。

然而，存储环境正在迅速变化，现代固态存储设备(SSD) 在许多重要用例中正在取代HDD。与HDD 相比，SSD 在性能和可靠性方面有着根本的不同。在考虑键值存储系统设计时，以下三个差异至关重要。首先，随机和顺序性能之间的差异并不像HDD 那样大；因此，执行大量顺序I/O 以减少后续随机I/O 的LSM-tree可能会不必要地浪费带宽。第二，SSD内部并行度大；构建在SSD 之上的LSM-tree 必须经过精心设计以利用所述并行性。第三，SSD 会因重复写入而磨损；LSM-tree 中的高写入放大可以显着降低设备寿命。这些因素的组合极大地影响了SSD 上的LSM-tree 性能，吞吐量降低了90%，写入负载增加了10 倍以上[13]。同时用SSD 替换HDD在LSM-tree 之下确实提高了性能，但使用当前的LSM-tree 技术，SSD 的真正潜力在很大程度上并未发挥。

过时条目的存在放大存储空间的因素称为空间放大。由于磁盘的可负担性，空间放大传统上并不是数据结构设计的主要关注点。然而，SSD 的出现使空间放大成为一个重要的成本问题。

面对查询这一应用场景时，所有主要的基于LSM-tree的键值存储索引主内存中各个run的各个块的第一个键。称为围栏指针。栅栏指针在主内存中占用$O(N/B)$空间，并且它们可以在各个run通过一个I/O 查找相关的键范围。

为提高查询性能付出了很多努力。为了加快点查询，所有表通常都与内存驻留的Bloom过滤器相关联，以便查询可以跳过不包含目标键的表。但是，Bloom过滤器无法处理范围查询。范围过滤器如SuRF[14]和Rosetta[15] 被提议通过过滤掉不包含请求范围内任何键的表来加速范围查询。但是，当请求范围内的键位于大多数候选表中时，过滤方法很难提高查询性能，尤其是对于大范围查询。此外，当查询可以由缓存回答时，访问过滤器的计算成本可能会导致性能平庸。

最坏情况下的查找成本与LSM树所有级别上Bloom过滤器的误报率之和成正比。与将各个元素的固定位数分配给所有Bloom过滤器的最先进的键值存储相反，Monkey将内存分配给不同级别的过滤器，以最小化其误报率之和。

在做范围查询的过程中，实际上已经在多个run 之间构造了一个全局的有序视图。但在大部分LSM-tree 实现中，这个视图都是用时构造，查询结束后即进行释放，重复构造过程中产生了很多计算/IO 开销，导致查询性能很差。如果能够将全局有序视图高效地存储下来，天然就是多个run 的索引，能够有效地加速查询。

REMIX为LSM-tree 设计了一种高效的索引，在保证写性能不下降的情况下，通过利用较少的额外存储开销记录下数据的全局有序视图（globally sorted view），有效地提升了查询性能，尤其是范围查询。

机器学习对构建计算机应用程序和系统有深刻的影响，与其以传统的算法思维方式编写代码，不如收集适当的数据，训练模型，从而为手头的任务实现稳健且通用的解决方案。在过去十年中，通过机器学习的方法，图像处理、自然语言处理都得到了飞速的发展。使用机器学习改进核心系统的一个概念是“学习索引”[16]。这种方法应用机器学习来取代在数据库系统中发现的传统索引结构，即B-Tree。为了查找一个键，系统使用一个预测键（和值）位置的学习函数；这种方法可以提高查找性能，并且还可能减少空间开销。自从机器学习引入到核心系统中后，已经有许多工作，使用更好的模型或者树结构，普遍改进学习如何减少基于树的访问时间和开销。

LSM-tree 非常适合用于学习索引，虽然写操作修改了LSM-tree，但树的大多数部分是不可变的；因此，学习一个预测键/值位置的函数只需要完成一次，并且只要不可变数据存在就可以使用它。BOURBON提供了类似于积极学习的高收益，同时显著降低了总成本。

3 研究进展

3.1 布鲁姆过滤器

布鲁姆过滤器是一种节省空间的概率数据结构，旨在查询集合中是否包含一个元素。它支持两种操作，即插入和测试。为了插入一个键，它应

用多个散列函数将键映射到位向量中的多个位置，并将这些位置的位设置为1。为了检查给定键的存在，该键再次被散列到多个位置。如果所有位都为1，则布鲁姆过滤器报告密钥可能存在。根据设计，布鲁姆过滤器可以报告假阳性，但不能报告假阴性。

布鲁姆过滤器可以构建在磁盘组件之上，以极大地提高点查找性能。要搜索磁盘组件，点查找查询可以首先检查其布鲁姆过滤器，然后仅在其关联的布鲁姆过滤器报告肯定答案时才继续搜索其B+-树。也可以为磁盘组件的ÿ个叶页构建一个布鲁姆过滤器。在这种设计中，点查找查询可以首先搜索B+-树的非叶子页面来定位叶子页面，假设非叶子页面足够小可以被缓存，然后检查相关联的布鲁姆过滤器在获取叶子页面以减少磁盘I/O。布鲁姆过滤器报告的误报不会影响查询的正确性，但查询可能会浪费一些I/O 搜索不存在的键。布鲁姆过滤器的误报率可以计算为

$$(1-e^{-kn/m})^k$$

，其中k 是哈希函数的数量，n 是键的数量，m 是总比特数[17]。此外，最小化误报率的哈希函数的最佳数量是$k=m/n\times ln2$。由于Bloom 过滤器非常小，并且通常可以缓存在内存中，因此用于点查找的磁盘I/O 数量由于使用它们而大大减少。

然而，现有设计为各个布鲁姆过滤器分配相同的误报率（即各个元素的位数），而不管它对应的run规模如何。Monkey的作者认为，整个LSM-tree的最坏情况点查询成本与所有过滤器的误报率之和成正比。然而，为所有过滤器分配相等的误报率并不能最小化这个总和，因为在所有run中保持相同的误报率意味着较大的run具有成比例地较大的布鲁姆过滤器，而无论其大小如何，探测任何运行的I/O 成本都是相同的。结果，最大级别的布鲁姆过滤器占用了大部分内存预算，而不会显着降低查询成本。

查询的I/O 成本取决于LSM 树中组件的数量。如果不使用Bloom 过滤器，点查找的I/O 成本将是$O(L)$（leveled）和$O(T\times L)$（tiered）。但是，布鲁姆过滤器可以大大提高点查找效率。对于零结果点查找，即搜索不存在的键，所有磁盘I/O 都是由布鲁姆过滤器误报引起的。假设所有Bloom 过滤器总共有M 位，并且在所有级别上具有相同的误报率。总共有N 个键，各个Bloom 过滤器的误报率为$O(e^{-M/N})$。因此，零结果点查找的I/O 成本将是$O(L\times e^{-M/N})$（leveled）和$O(T\times L\times e^{-M/N})$用于（tiered）。要搜索现有的唯一键，必须执行至少一个I/O 来获取条目。鉴于在实践中Bloom 过滤器误报率远小于1，因此leveled 和tiered的成功点查找I/O 成本将为$O(1)$。

Monkey 调整内存组件和Bloom 过滤器之间的合并策略、大小比率和内存分配，以找到给定工作负载的最佳LSM-tree 设计。Monkey 的第一个贡献是展示了通常的Bloom 过滤器内存分配方案，它为所有Bloom 过滤器的ÿ个键分配相同数量的位，导致次优性能。直觉是最后一层的T个组件，包含大部分数据，消耗了大部分Bloom filter 内存，但他们的Bloom filter最多只能节省T个磁盘I/O用于点查找。为了最小化所有Bloom 过滤器的总体误报率，Monkey 分析表明应该将更多位分配给较低级别的组件，以便Bloom 过滤器误报率呈指数增长。

Monkey在数据维度、软硬件等多个指标都有较好表现，这里我们重点关注它的查询优化表现。Monkey 更适合混合点查找工作负载。在图2中，我们展示了Monkey 在零结果与非零结果点查找的任何比率方面都占主导地位。同时，该实验表明，Monkey 显着改善了查询工作负载中跨广泛时间局部性的非零结果查找的查找延迟。

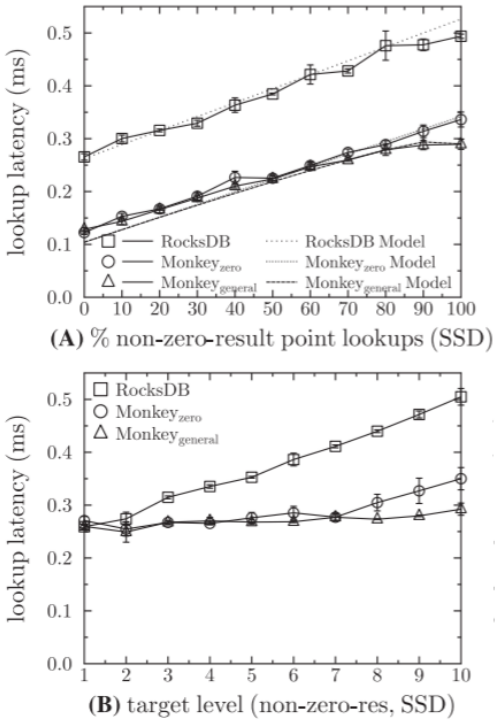

图2 Monkey查询效率

在图3中，作者改变了小范围查找和工作负载更新之间的比例。由于本实验中未设置点查找，布鲁姆过滤器不会产生任何性能优势，因此将布鲁姆过滤器原本会消耗的空间分配给缓冲区以提高更新成本。此外，随着小范围查找比例的

增加，Monkey切换到越来越多的读取优化合并。注意到Monkey的曲线是钟形的。原因是当工作负载趋向于单一操作类型时，实现更高吞吐量的可能性就越大，因为单一配置可以很好地处理大多数操作。总体而言，随着短距离查找比例的增加，Monkey 在RocksDB 中的优势高达约50%。

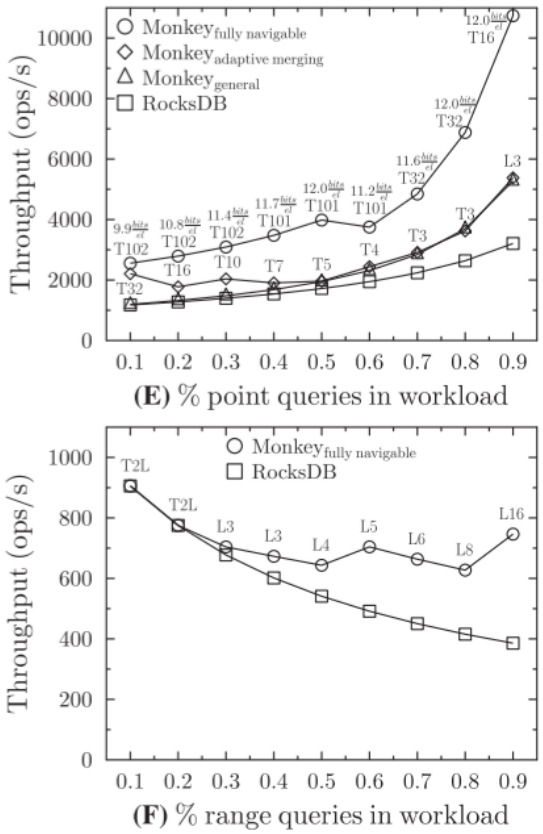

图3　Monkey范围查询吞吐量

3.2　范围查询索引

LevelDB/RocksDB 中的范围查询是通过使用迭代器结构来实现跨多个表的导航，就好像所有键都在一个有序run中一样。范围查询首先使用带有查找键的SeekTo() 调用查找操作初始化迭代器，查找键是目标键范围的下边界。查找操作定位迭代器，使其指向存储中等于或大于查找键的最小键表示为范围查询的目标键。next操作使迭代器前进，使其指向有序视图中的下一个键。一系列next操作可用于检索目标范围中的后续键，直到满足终止条件。由于有序的run是按时间顺序生成的，因此目标键可以驻留在任何运行中。因此，迭代器必须跟踪所有已排序的run。合并迭代器将从这些迭代器中的ÿ一个收集键并按排序顺序提供数据。

对多个有序run的范围查询，动态构建基础表的排序视图，可以按排序顺序检索键。但是，现有的基于LSM-tree 的KV-store 排序的视图在搜索时反复重建，然后在查询后立即丢弃，这导致了过多的计算和I/O ，搜索性能不佳。REMIX 的思想是通过保留基础表的排序视图并在将来的搜索中反复使用来提高范围查询效率。

REMIX 将有序视图涉及到的键范围按照顺序划分为多个segment，各个segment 内保存一个anchor key，记录该segment 内最小的key。此外，segment 内还维护了cursor offsets 和run selectors 两个信息。Cursor offsets 记录了各个run 中首个大于等于anchor key 的offset，run selectors 顺序记录了segment 内各个key 所在的run 序号。

范围查询过程就是不断地根据run selectors 调整访问的cursor offset 及run selector，并结合与end key 的比较结果，将点查过程扩展为范围查询。

然而，segment 内查询是从anchor key 开始，这个过程还是可能会产生不必要的比较和run 访问。这和segment 内的key 数量有关，如果数量多，顺序查询性能差，但存储效率高，反之anchor key 会变多，存储开销大。

因此，本文针对segment 内的查询做了额外的优化。作者借助run selector来进行二分查询，但由于run selector 仅记录了该key 归属的run，并不确定是该run 中的哪一个key，所以还需要额外计算一份occurrences 来辅助定位。该值表示的是当前run selector 所在的run 在segment 内是第一次出现，通过该run 的cursor offset 加上selector 对应的occurrence 值，即可得到它在该run 中对应key 的下标。

以 图4中16个selector、 查 询41为 例 （ 假设cursor offset 都是0）：

(1) 访问第8个selector，通过selector 和occurrence 确认该key 为run3 的第4个key（43），大于41，所以向左查询。

(2) 访问第4 个selector，run2 第一个key（17），小于41，所以向右查询。

(3) 不断二分，最终查询到41 为run3 的第3个key。

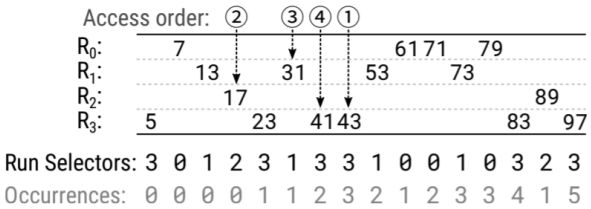

图4　segment中的二分查找示例

显然，二分搜索和occurrences 跳过run 内元素

都能有效地减小key 之间的比较次数，降低计算的开销。

通过全局的二分查找，有效地减少了定位一个key 时的复杂度。传统方式为$M \times logN$，REMIX为$logMN$。Run selectors 可以帮助REMIX 在查询时快速地在不同run 的key 间顺序推进，不用维护最小堆，也就避免了在pop 时通过key 比较进行重排，降低了计算开销。基于anchor key 的二分也使得查询过程有机会跳过一些无需访问的run。

表1显示了不同工作负载的REMIX 存储成本和LevelDB和RocksDB中SSTable格式的块索引（BI）和布鲁姆过滤器（BF）的存储成本。在最坏的情况下，REMIX 的大小仍然不到KV 数据大小的10%。

Work-load	Avg. Key Size	Avg. Value Size	Bytes/Key SSTable BI	SSTable BI+BF	REMIX (H=8) D=16	D=32	D=64	REMIX data (D=32)
UDB	27.1	126.7	1.2	2.4	4.1	2.2	1.3	1.44%
Zippy	47.9	42.9	1.2	2.4	5.4	2.9	1.6	3.16%
UP2X	10.45	46.8	0.2	1.5	3.0	1.7	1.0	2.97%
USR	19	2	0.1	1.4	3.6	2.0	1.2	9.38%
APP	38	245	2.9	4.2	4.8	2.6	1.5	0.91%
ETC	41	358	4.4	5.6	4.9	2.7	1.5	0.67%
VAR	35	115	1.4	2.7	4.6	2.5	1.4	1.65%
SYS	28	396	3.3	4.6	4.1	2.3	1.3	0.53%

表1 REMIX 存储成本与实际KV 大小。BI 代表块索引。BF 代表Bloom过滤器。最后一列显示了REMIX 与KV 数据的大小比例

作者使用不同表集来测量三个范围和点查询操作的单线程吞吐量，即Seek、Seek+Next50和Get。图5和6分别显示了具有弱访问局部性和强访问局部性的表的吞吐量结果。结果显示，Seek中不使用segment内二分搜索的表文件为8和16个的REMIX比合并迭代器的性能好3.5倍和6.1倍，Seek+Next50中，对于2个、8个和16个表文件，加速比分别为1.4倍、2.3倍和3.1倍。点查询中REMIX的效率略低，当表少于14 个时，使用Bloom 过滤器搜索SSTable 的性能优于搜索REMIX 索引表文件。因为Bloom 过滤器的效率高且在SSTable 中搜索比在管理更多键的REMIX 上更快。在最坏的情况下，REMIX 的吞吐量比布鲁姆过滤器低20%。REMIX 点查询性能也由于加速底层寻道操作的强局部性而提高，如图5(c)所示。同时，布鲁姆过滤器的结果保持不变，因为搜索成本主要取决于误报率和单个表的搜索成本。因此，当表超过9 个时，REMIX 能够胜过Bloom 过滤器。

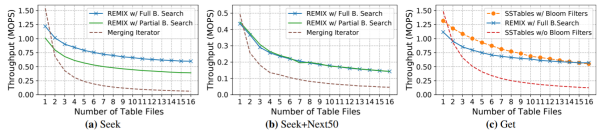

图5 随机分配键的点和范围查询性能（弱局部性）

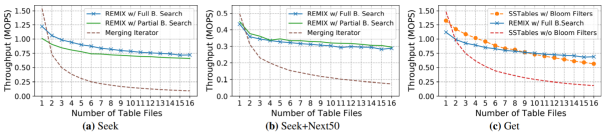

图6 随机分配键的点和范围查询性能（强局部性）

3.3 学习索引

BOURBON在Wisckey的基础上加入了学习索引。学习索引就是指当查询一个key 的时候，系统使用该索引预测出要查询的key 所对应的位置，相比于传统的数据结构中的查找性能有比较大的提升，同时一定程度上因为不直接构建具体的数据结构节省了空间开销。

现有的学习索引大多是基于数据库场景中的B树，而LSM-tree相关的优化很少，因为学习索引是针对读优化的，而LSM-tree主要针对写进行了优化。然而，虽然较多的写操作使LSM-tree的数据发生了变化，但树的大多数部分是不可变的；因此，学习一个预测键/值位置的函数只需要完成一次，并且只要不可变数据还存在就可以使用。

BOURBON 使用分段线性回归，能够以很少的空间开销实现快速训练和查找。BOURBON 采用文件学习，模型是基于文件构建的，因为LSM 文件一旦创建，就永远不会就地修改。BOURBON 实施成本效益分析器，动态决定是否学习文件，减少不必要的学习，同时最大化收益。BOURBON 将学习集成到已经高度优化的生产质量系统中，BOURBON 的实现为WiscKey （大约20K 行代码）增加了大约5K 行。

对于LSM-tree 的查询可能需要对多个level 进行查询，学习索引的核心思想是针对输入训练一个模型从而预测出对应的记录的具体地址。模型可能有误差，因此预测有一个相关的误差界。在查找过程中，如果模型预测的键的位置是正确的，则返回记录;如果不正确，则在错误范围内执行本地搜索。

LSM-tree 中的查询操作包含索引和数据访问两个方面，学习索引可以减少索引的一些步骤的开销，但是对于数据访问的开销几乎无影响。因此如果索引在总查找延迟中占相当大的比例，学习索引就可以显著提升查询的性能。当数据集或它的一部分缓存在内存中时，数据访问成本很低，因此索引

成本就变得很重要。学习索引并不局限于数据缓存在内存中的场景。它们在快速存储设备中也可以发挥优势，并且可以在正在出现的更快的设备上发挥更大的作用，延迟降低，索引结构的操作所占的延迟比重越来越大，随着存储器件的发展，学习索引能够发挥的效果也越来越显著。

对于学习索引的查询方式，步骤如下：

(1) FindFiles：找到候选的SSTables 文件

(2) LoadIB+FB：加载filter 和index block，这些块可能已经被缓存在内存中

(3) Model Lookup：BOURBON 在候选的SSTables 文件中对要查询的key 进行检索，模型相应地输出键对应的文件内偏移pos 和误差边界δ。然后BOURBON 计算包含记录$pos-\delta$ 到$pos+\delta$ 的数据块

(4) SearchFB：检查该数据块对应的filter block来判断key 是否存在，若存在，则BOURBON 计算要加载的块对应的字节范围

(5) LoadChunk：加载对应的字节范围

(6) LocateKey：键位于加载的块中，那ô该key 将位于预测的位置上（加载的chunk 的中点）；如果不在，BOURBON 将会对该chunk 执行二分查找

(7) ReadValue：使用相应的Value Pointer从ValueLog 中读取对应的Value

步骤1是因为BOURBON使用了文件学习，步骤4中键和指针的大小固定，因此计算相对简单。经过实验得到，BOURBON减少了索引花费的时间。图7中标记为Search 的部分对应基线中的SearchIB 和SearchDB。还降低了数据访问成本，因为BOURBON 加载的字节范围比基线加载的整个块要小。

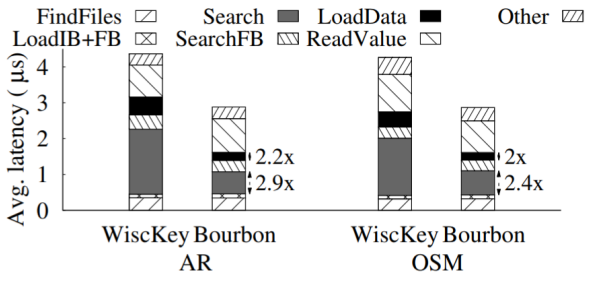

图7　WiscKey 和BOURBON 的延迟细分

BOURBON 范围查询也有一定的优化，如图8展示了BOURBON 的吞吐量标准化到了WiscKey 之后的结果。对于较短的范围，索引开销(即查找范围的第一个键的开销)占主导地位，BOURBON 优化效果比较明显，但随着范围的增大，其效果就略不明显，这是因为BOURBON 可以加速索引部分，但它遵循与WiscKey 类似的·径来扫描后续键。因此，在大范围查询时，索引查询占较少的总性能，性能提升就不明显了。

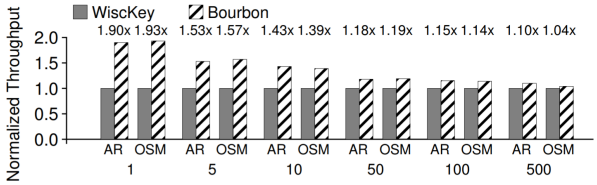

图8　不同情况下的范围查询

3.4　本章小结

Monkey、REMIX和BOURBON分别使用布鲁姆过滤器、全局有序视图、学习索引的方法来提高LSM-tree的查询效率。

其中，Monkey采用的布鲁姆过滤器主要针对点查询，实验结果表明Monkey 更适合混合点查找工作负载。随着范围查询比例的增加，Monkey的效率降低又升高，原因是当工作负载趋向于单一操作类型时，实现更高吞吐量的可能性就越大，因为单一配置可以很好地处理大多数操作。REMIX主要针对范围查询，利用了LSM-tree的不变性构建全局有序视图，LSM-tree中的范围查询通常采用合并迭代器，效率较低，REMIX的本质是索引，也就是用一块额外存储空间换时间的思想，实验表明，它在点查询中也有一定的优化。BOURBON将机器学习的思想引入到LSM-tree中，加入学习索引，显著降低了查询的成本，同时也不会产生额外的空间开销，结果表明，BOURBON 对范围查询有一定的优化。

4　总结与展望

近年来，LSM-tree由于其优越的写入性能、高空间利用率等特性，在现代NoSQL系统中越来越流行。这些因素使LSM-tree在大数据领域得以广泛采用和部署，以服务于各种工作负载。然而，LSM-tree的主要问题是，记录可能存储在多个位置中的任何位置，因此会牺牲读取性能。此外，这些结构通常需要单独的数据重组过程，以不断提高存储和查询效率。LSM-tree上的查询必须搜索多个run以执行协调，即查找各个key的最新版本。获取特定键值的点查询可以简单地从最新到最旧逐个搜索所有run，并在找到第一个匹配项后立即停止。范围查询可以同时搜索所有run，将搜索结果输入优先

级队列排序。

本文主要介绍了三种在LSM-tree中进行查询的优化策略，分别是采用布鲁姆过滤器的Monkey、全局有序视图的REMIX 和学习索引的BOURBON这三种优化策略，以及其他类似或相关的优化策略。

其中，Monkey和BOURBON在范围查询中的优化不是特别明显，REMIX很大程度上优化了范围查询，但在最差的条件下需要额外10%的空间开销。BOURBON是一个基于学习索引的原型，未来会有更多利用学习索引的LSM-tree在实际生产环境中。

随着LSM-tree逐渐广泛应用于端DBMS存储引擎中，应针对这种多索引的设置开发新的查询处理和数据摄取技术。可能发展的方向包括辅助结构的自适应维护，用于LSM-tree的查询优化，以及LSM树维护任务与查询执行的协同。

致 谢 感谢施展老师和童薇老师讲授的课程，受益匪浅。同时感谢这门课带来的锻炼机会。

参 考 文 献

[1] Armstrong, T., Ponnekanti, V., Borthakur, D. & Callaghan, M. LinkBench: a database benchmark based on the Facebook social graph. *Proceedings Of The 2013 ACM SIGMOD International Conference On Management Of Data*. pp. 1185-1196 (2013)

[2] Goel, A., Pound, J., Auch, N., Bumbulis, P., MacLean, S., F?rber, F., Gropengiesser, F., Mathis, C., Bodner, T. & Lehner, W. Towards scalable real-time analytics: An architecture for scale-out of OLxP workloads. *Proceedings Of The VLDB Endowment*. **8**, 1716-1727 (2015)

[3] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. & Vogels, W. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*. **41**, 205-220 (2007)

[4] Raju, P., Ponnapalli, S., Kaminsky, E., Oved, G., Keener, Z., Chidambaram, V. & Abraham, I. mlsm: Making authenticated storage faster in ethereum. *10th USENIX Workshop On Hot Topics In Storage And File Systems (HotStorage 18)*. (2018)

[5] O'Neil, P., Cheng, E., Gawlick, D. & O'Neil, E. The log-structured merge-tree (LSM-tree). *Acta Informatica*. **33**, 351-385 (1996)

[6] Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A. & Gruber, R. Bigtable: A distributed storage system for structured data. *ACM Transactions On Computer Systems (TOCS)*. **26**, 1-26 (2008)

[7] Lakshman, A. & Malik, P. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*. **44**, 35-40 (2010)

[8] Harter, T., Borthakur, D., Dong, S., Aiyer, A., Tang, L., Arpaci-Dusseau, A. & Arpaci-Dusseau, R. Analysis of HDFS under HBase: a Facebook messages case study. *12th USENIX Conference On File And Storage Technologies (FAST 14)*. pp. 199-212 (2014)

[9] Dayan, N. & Idreos, S. The log-structured merge-bush & the wacky continuum. *Proceedings Of The 2019 International Conference On Management Of Data*. pp. 449-466 (2019)

[10] Dayan, N., Athanassoulis, M. & Idreos, S. Optimal bloom filters and adaptive merging for LSM-trees. *ACM Transactions On Database Systems (TODS)*. **43**, 1-48 (2018)

[11] Zhong, W., Chen, C., Wu, X. & Jiang, S. REMIX: Efficient Range Query for LSM-trees. *19th USENIX Conference On File And Storage Technologies (FAST 21)*. pp. 51-64 (2021)

[12] Dai, Y., Xu, Y., Ganesan, A., Alagappan, R., Kroth, B., Arpaci-Dusseau, A. & Arpaci-Dusseau, R. From wisckey to bourbon: A learned index for log-structured merge trees. *14th USENIX Symposium On Operating Systems Design And Implementation (OSDI 20)*. pp. 155-171 (2020)

[13] Lu, L., Pillai, T., Gopalakrishnan, H., Arpaci-Dusseau, A. & Arpaci-Dusseau, R. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions On Storage (TOS)*. **13**, 1-28 (2017)

[14] Zhang, H., Lim, H., Leis, V., Andersen, D., Kaminsky, M., Keeton, K. & Pavlo, A. Surf: Practical range query filtering with fast succinct tries. *Proceedings Of The 2018 International Conference On Management Of Data*. pp. 323-336 (2018)

[15] Luo, S., Chatterjee, S., Ketsetsidis, R., Dayan, N., Qin, W. & Idreos, S. Rosetta: A robust space-time optimized range filter for key-value stores. *Proceedings Of The 2020 ACM SIGMOD International Conference On Management Of Data*. pp. 2071-2086 (2020)

[16] Kraska, T., Beutel, A., Chi, E., Dean, J. & Polyzotis, N. The case for learned index structures. *Proceedings Of The 2018 International Conference On Management Of Data*. pp. 489-504 (2018)

[17] Bloom, B. Space/time trade-offs in hash coding with allowable errors. *Communications Of The ACM*. **13**, 422-426 (1970)