KV 存储优化研究

谢晨

1) (武汉光电国家研究中心 武汉 国家名 430000)

摘 要 以闪存作为存储介质的固态硬盘，读写速度快，寻道时间短，功耗低，无噪音并且工作温度范围大，因此，其在存储系统中占据越来越重要的地位。

KV 是现在常见的存储方式，它在普通硬盘组成的存储系统经常受到 I/O 性能的限制。最近出现的 SSD 的超低延迟和高带宽，使得这一限制被解除成为了可能，但是切换到全新的数据布局并将整个数据库扩展到高端 SSD 需要相当大的投资。

同时，LSM 本身是为高速写操作优化的，而范围查询在传统的 LSM-tree 上需要 seek 且归并排序来自多个 Table 的数据，开销是很大的且经常导致较差的读性能。

于是，第一篇论文提出了一种名为 SpanDB 的存储方式，以便于解决将整个数据库全部扩展到高端 SSD 时的成本问题它适应了流行的 RocksDB 系统，以利用高速 SSD 的选择性部署。SpanDB 允许用户在更便宜和更大的 SSD 上托管大部分数据，同时将预写日志（WAL）和 LSM 树的顶层重新定位到更小，更快的 NVMe SSD。

第二篇论文提出了一个空间高效的 KV 索引数据结构 REMIX，记录跨多个表文件的 KV 数据的全局排序视图，以便于提升范围查询的性能。对多个 REMIX 索引数据文件的范围查询可以使用二分搜索快速定位目标键，并在不进行键比较的情况下按排序顺序检索后续键。基于此构建了 RemixDB，一个采用了一个更高效的压缩策略并使用 REMIX 来进行快速的点查询和范围查询的 KV 存储。

第三篇论文提及了闪存的特性对 RocksDB 的设计产生的影响，读写性能的不对称性和有限的持久性给数据结构和系统架构的设计带来了挑战和机遇。于是 RocksDB 采用了闪存友好的数据结构，并对现代硬件进行了优化。

关键词 KV 存储系统，SpanDB,RocksDB,RemixDB

Research on KV storage optimization

Chen-Xie[1)]

[1)](Wuhan National Laboratory for Optoelectronics, Wuhan, 430000, China)

摘要 The solid-state disk with flash memory as the storage medium has the advantages of fast reading and writing speed, short seek time, low power consumption, no noise and large working temperature range. Therefore, it plays a more and more important role in the storage system.

KV is a common storage mode. It is often limited by I / O performance in the storage system composed of ordinary hard disk. Recently, the ultra-low latency and high bandwidth of SSDs make it possible to lift this limitation, but switching to a new data layout and expanding the entire database to high-end SSDs require considerable investment.

Therefore, the first paper proposes a storage method called spandb to solve the cost problem of expanding the whole database to high-end SSD. It adapts to the popular rocksdb system to take advantage of the

selective deployment of high-speed SSD. Spandb allows users to host most data on cheaper and larger SSDs, while relocating the top level of the pre write log (wal) and LSM tree to smaller, faster nvme SSDs.

The second paper proposes a spatially efficient kV index data structure remix, which records the global sorted view of kV data across multiple table files, so as to improve the performance of range query. For the range query of multiple Remix index data files, you can use binary search to quickly locate the target key and retrieve the subsequent keys in sorted order without key comparison. Based on this, remixdb is built, which adopts a more efficient compression strategy and uses remix for fast point query and range query.

The third paper mentions the impact of the characteristics of flash memory on the design of rocksdb. The asymmetry of read-write performance and limited persistence bring challenges and opportunities to the design of data structure and system architecture. Therefore, rocksdb adopts flash memory friendly data structure and optimizes modern hardware.

Key words Key value storage,SpanDB,RocksDB,RemixDB

1. 引言

以闪存作为介质的固态硬盘，读写速度非常快，功耗低，可以使得存储系统的性能不受 I/O 速度的制约。于是，人们试图设计对闪存友好的存储系统，因此，闪存的特性深刻地影响着存储系统的设计。

1.1 SpanDB 的产生背景

由于 KV 存储系统的应用环境通常是写密集型的，因此，对写入友好的日志结构合并树（LSM）被广泛地采用为底层存储引擎。

随着 SSD 的发展，基于 SSD 的 KV 存储系统的性能也得到了提升。有些系统为了获得更高的吞吐量和可扩展性，放弃了 LSM 树或者将 KV 数据管理迁移至特别的硬件。这样的做法提升了 KV 存储系统的性能，但却带来了成本的激增。

因此，第一篇文章试图使主流的基于 LSM 树的 KV 存储系统设计适应 SSD 以及 SSD 带来的高速 I/O，并且着重于降低成本。

在传统的 KV 系统中，基于 LSM 树的 KV 存储未能充分利用 NVMe SSD 的潜力。例如，在 Optane P4800X 上部署 RocksDB 与 SATA SSD 相比，在 50% 写入工作负载上部署 RocksDB 只能将吞吐量提高 23.58%。同时，常见 KV 存储设计的 I/O 路径严重未充分利用 SSD 的超低延迟，尤其是对于小写入，这对预写日志记录的损害尤其之甚。此外，现有的 KV 系统的工作流设计会引入高软件开销。同时，NVMe 的接口访问限制使得 KV 设计复杂程度变高。

因此，SpanDB 应运而生，其对流行的 KV 存储移植到 USEPDK I/O 的瓶颈的全面分析，并且通过合并相对较小但速度较快的磁盘 （SD） 来扩展对较新数据的所有写入和读取的处理，同时在一个或多个更大、更便宜的容量磁盘 （CD） 上横向扩展数据存储。

同时支持通过 SPDK 进行快速、并行的访问，以更好地利用 SD，绕过 Linux I/O 堆栈，特别是允许高速 WAL 写入。并设计了一个适用于基于轮询的 I/O 的异步请求处理管道，该管道消除了不必要的同步，使 I/O 等待与内存中处理严重重叠，并自适应地协调前台/后台 I/O。

根据实际的 KV 工作负载对数据进行战略性和适应性分区，主动让 CD 参与其 I/O 资源（尤其是带宽），以帮助共享当代 KV 系统中常见的写入放大。

评估表明，在所有测试用例中，SpanDB 在所有类别（吞吐量，平均延迟和尾部延迟）中的表现都明显优于 RocksDB，尤其是写入密集型用例。

1.2 RemixDB 的产生背景

LSM 本身是有 Compaction 来减少查询时检索的 SSTable 数量的，选择 SSTable 的策略又分为了 leveled 和 tiering：

LevelDB、RocksDB 采用的 Leveled 的策略就是把小的 sorted run 合并一个更大的 sorted run 来保证重叠的 Tables 数量小于阈值，该策略确实保证了较好的读性能，但是因为归并排序的方式导致写放大问题比较严峻。

Tiered 则是等待多个相近大小的 sorted runs 达到阈值后合并到一个更大的 runs，从而提供更小的写放大以及更高的写吞吐。Cassandra、ScyllaDB 就采用了这样的方式。但是没有限制重叠的 Tables 的数量从而导致较大的查询开销。

SIGMOD18 Dostoevsky 和 SIGMOD19 The Log-Structured Merge-Bush & the Wacky Continuum 提出的 compaction 策略虽然做了一定程度上的读写的平衡，但是还是没能同时实现最好的读和写。

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。

1.3 RocksDB 的产生背景

在过去的十年里，基于 flash 的 SSD 的数量激增。低延迟和高吞吐量的设备不仅挑战了软件充分利用它的能力，而且改变了实现有状态服务的数量。SSD 每秒提供成千上万的输入/输出操作(IOPS)，这比旋转硬盘快数千倍。它还可以支持数百个 mbs 的带宽。

然而，由于程序/擦除周期的数量有限，SSD 的高写带宽无法持续下去。SSD 的特性为重新考虑存储引擎的数据结构以优化这些硬件提供了机会。在许多情况下，SSD 的高性能也将性能瓶颈从设备 I/O 转移到网络的延迟和吞吐量。对于应用程序来说，在本地 ssd 上存储数据而不是使用远程数据存储服务变得更具吸引力。这增加了对嵌入在应用程序中的键值存储引擎的需求。

RocksDB 是一个由 facebook 创建的高性能的持久性键值存储引擎，它针对固态驱动器(SSDs)的特定特性进行了优化，针对大规模（分布式）应用程序，并被设计为嵌入到高级应用程序中的库组件。因此，每个 RocksDB 实例只管理单个服务器节点的存储设备上的数据；它不处理任何主机间操作，如复制和负载平衡，也不执行高级操作，如检查点——它将这些操作的实现留给应用程序，但提供适当的支持，以便它们能够有效地进行这些操作。

RocksDB 及其各种组件具有高度可定制性，允许存储引擎根据广泛的需求和工作负载进行定制；定制可以包括预写日志(WAL)处理、压缩策略和压缩策略。RocksDB 可以调整为高写吞吐量或高读吞吐量、空间效率或介于两者之间。由于其可配置性，RocksDB 被许多应用程序使用，代表了广泛的用例。除了用作数据库的存储引擎外，RocksDB 还用于流处理、日志记录、派对服务、索引服务以及 SSD 缓存。

RocksDB 使用对数结构化合并(LSM)树作为其主要数据结构来存储数据。

2. 原理与优势

2.1 SpanDB 的原理及优势

SpanDB，这是一种使用异构存储设备的高性能，经济高效的基于 LSM 树的 KV 存储。SpanDB 主张使用小型，快速且通常更昂贵的 NVMe SSD 作为速度磁盘（SD），同时部署更大，更慢，更便宜的 SSD（或此类设备的阵列）作为容量磁盘（CD）。SpanDB 将 SD 用于两个目的：WAL 写入和存储 RocksDB LSM 树的顶层。

由于 WAL 处理成本对用户可见且直接影响延迟，因此 SPanDB 预留了足够的资源（核心和并发 SPDK 请求，以及足够的 SPDK 队列），以最大限度地提高其性能。同时，WAL 数据只需要维护到相应的冲洗操作，并且通常需要 GB 的空间，而今天的"小型"高端 SSD，如 Optane，提供超过 300GB。这促使 SpanDB 将 RocksDB LSM 树的顶层移动到 SD。 这还会从 CD 卸载大量刷新/压缩流量，而 CD 中大量较冷的数据驻留在 CD 中。

改进的方法：

1.采用 SPDK 访问的小而快速的 SD，而 SPDK 通过快速、并行的 WAL 写入来加快 WAL 的速度。

2.将 SD 也用于数据存储，它优化了这种快速 SSD 的带宽利用率。

3.通过启用工作负载自适应 SD-CD 数据分发，它可以主动聚合跨设备可用的 I/O 资源（而不是仅将 CD 用作"溢出层"）。

4.虽然主要优化写入，但通过将 I/O 卸载到

SD，它减少了读取密集型工作负载的尾部延迟。

5.通过修剪同步和主动平衡前台/后台 I/O 需求，它利用了快速轮询 I/O，同时节省了 CPU 资源。

SpanDB 重新设计了 RocksDB 的 KV 请求处理、存储管理和组 WAL 写，利用快速 SPDK 接口，保留了 RocksDB 的 lsm 树组织、后台 I/O 机制、事务支持等数据结构和算法.因此，它的设计与 RocksDB 的许多其他优化方案是互补的。添加 SD 后，可以将现有的 RocksDB 数据库迁移到 SpanDB。

2.2 RemixDB 的原理及优势

范围查询依赖排序视图，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 树的 KV 存储无法利用这种继承的不变性，而是在每次范围查询中重复构建该全局排序视图。

为了提高 I/O 效率，基于 LSM 树的 KV 存储采用内存效率高的元数据格式，包括稀疏索引和 Bloom 过滤器[4]。如果我们记录每个键及其位置以保留存储中的排序视图，则存储的元数据可能会显著膨胀，从而导致读写性能下降。为了避免这个问题，REMIX 数据结构必须节省空间。

因此，RemixDB 选择将排序视图高效地存储下来，排序视图天然就是多个搜索的索引，能够有效地加速查询。

在记录排序视图的同时，还需要保证内存高效，不能因为存储更多元数据导致读写性能丢失。

这样 RemixDB 除了使 I/O 更加高效，还加速了查询。

2.3 RocksDB 的原理及优势

2.3.1 RocksDB 的写

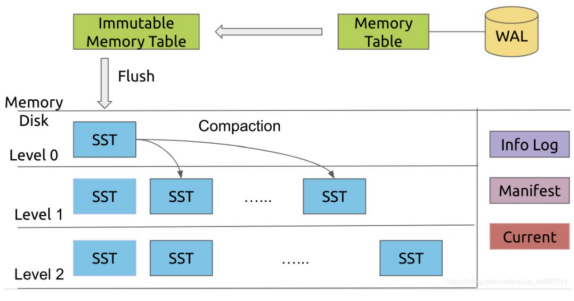

图 1 RocksDB 写操作

首先写入到名为 MemTable 的内存 Buffer 和磁盘上的 WAL（硬盘中的 WAL 写并不是强制的）中。

当 MemTable 达到一定的大小（设定的阈值），当前 MemTable 和 WAL 就不可修改了。

此时分配新的 MemTable 和 WAL 用于接受后续的写入，将不可修改的 Memtable 刷新到磁盘上的 SSTable 文件中，这就是 RocksDB 写入的全过程。

2.3.2 RocksDB 的压缩

为了要完成 RocksDB 的 Compaction,首先要确定各个级别 SSTable 的大小。

最新一级的 sstable 由 MemTable 创建，并置为 0 级，高于 0 级的级别由 Compaction 创建。

当 L 级的目标大小被超过的时候，将它与 L+1 级的重叠 sstable 合并。接着删除已经被覆盖或删除的数据。

同时，RocksDB 支持多种不同类型的压缩，通过使用不同的压缩策略，RocksDB 可以配置为写友好/读友好/对某些特殊缓存友好等等，灵活性非常高。

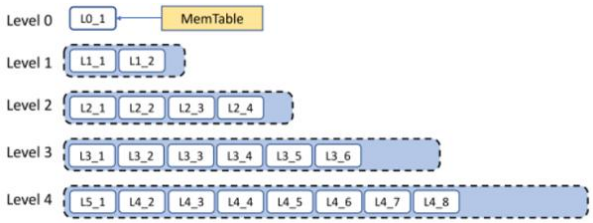

图 2 RocksDB 压缩操作

2.3.3 RocksDB 的读

在读取路径中，在每个后续级别都进行键查找，直到找到键为止。它首先搜索所有 memtable，然后搜索所有级别为 0 的 sstable，然后依次搜索更高级别的 sstable。在每一个级别上，都使用二分查找。Bloom 过滤器用于消除 SSTable 文件中不必要的搜索。扫描要求搜索所有级别

2.3.4 RocksDB 的优势

RocksDB 起源于 LevelDB，针对 SSD 的一些特性进行了优化，以分布式应用程序为目标，同时也被设计成可以被嵌入到一些高级应用程序（如 Ceph）的 KV 库。每个 RocksDB 实例只管理单个节点上的数据，没有处理任何节点之间的操作（比如 replication、 loadbalancing），也不执行高级操作，比如 checkpoints，让上层应用来实现，底层只是提供一些支持。

RocksDB 可以根据负载以及性能需求进行定制和调优，主要包括对 WAL 的处理、压缩策略的选择。

RocksDB 应用广泛，可以应用在流处理，SSD 缓存，索引服务，日志/队列服务等服务之中。

3. 研究进展

3.1 SpanDB 的研究进展

3.1.1 SpanDB 的结构

SpanDB 提倡使用小型、快速且通常更昂贵的 NVMe SSD 作为快速磁盘(SD)，同时部署更大、更慢、更便宜的 SSD 作为容量磁盘(CD)。SpanDB 使用 SD 来进行（1）WAL 写（2）存储 RocksDB 的 LSM-tree 的顶层。

SpanDB 提倡使用小型、快速且通常更昂贵的 NVMe SSD 作为快速磁盘(SD)，同时部署更大、更慢、更便宜的 SSD 作为容量磁盘(CD)。SpanDB 使用 SD 来进行（1）WAL 写（2）存储 RocksDB 的 LSM-tree 的顶层。

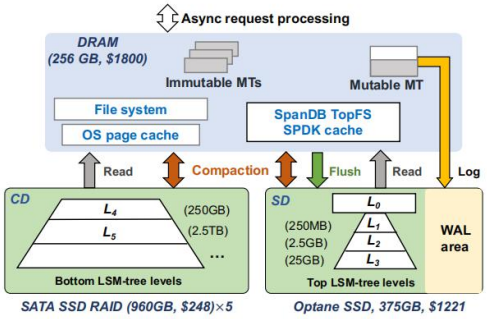

图 3 SpanDB 结构示意图

SpanDB 在保留 RocksDB 的 MemTable 设计，具有一个或多个不可改变的 MemTable,而且并没有修改修改 RocksDB 的 KV 数据结构，算法或操作语义。

磁盘上的数据分布在 CD 和 SD 上，两个物理存储分区。SD 进一步分区，有一个小的 WAL 区域作为数据区域使用的其余空间。

CD 分区同时存储了 tree stump，经常包含较冷的大部分数据。SpanDB 的管理还是与 RocksDB 一样，通过文件系统访问，并借助操作系统页面缓存进行辅助。

3.1.2 SpanDB 的设计与实现

SpanDB 采用异步请求处理。在 n 核机器上，用户将客户端线程的数量配置为 N client，每个线程占用一个核。剩下的 n-N client 个核心处理 SpanDB 内部服务线程，相应地被分为两个角色：loggers and workers。这些线程都被绑定到他们分配的核心上。Loggers 主要执行 WAL 写入，Workers 处理后台任务处理（flush/compaction）以及一些非 I/O 任务，例如 Memtables 的写入和更新，WAL 日志项的准备，相关事务的加锁/同步。根据观察到的写强度，head-server 线程自动自适应地决定 logger 的数量，这些 logger 被绑定到分配了 SPDK 队列的核上，从而保证 WAL 的写带宽。

SpanDB 提供简单，直观的异步 API。 对于存在 RocksDB 同步获取和输出操作，它增加了它们的异步对等对象 A_get 和 A_put， 加上 A_check 来检查请求状态。

SpanDB 将异步写分为了三个部分：首先将请求 dump 到 prolog 队列中，让 worker 进行处理，然后将 worker 生成的 WAL 日志传递到 log 队列中，最后，当一个组写入到 SD 之后，logger 把相应的请求添加到 EpiLog 之中，以便于 worker 完成最终的处理。

SpanDB 保留了 group logging 机制，但也允许多个 WAL 流并发写入。它不是让一个客户端作为 leader (并迫使 follower 等待)，而是使用专用的 logger，并发地发出批处理写操作。每个 logger 抓取它在 QLog 中看到的所有请求，并将这些 WAL 条目聚合为尽可能少的 4KB 块。它通过为一个请求窃取 SPDK 繁忙等待时间准备/检查其他请求来执行 pipeline。

SpanDB 首先在 SD 上分配可配置数量的 Pages 给相应的 WAL 区域，具有唯一逻辑页编号 LPN。这些页中的一个会被设置为元数据页，在任何时候只会有一个可变的 Memtable，其对应的日志文件会不断增大。SpanDB 会分配固定数量的逻辑页 groups，每个组会包含连续的页，而且足够大可以装下一个 Memtable 的日志数据。被使用了的日志页和对应的 Memtable 给组织在一起：SpanDB 重用了 RocksDB Memtable 的日志文件号 LFN 作为日志标签号 LTN，嵌入在所有日志页的开头以便进行故障恢复。

为了 KV 服务器的持续均衡的运行，SpanDB 将 RocksDB 的 lsm 树的顶层迁移到 SD。

3.1. 3 SpanDB 的性能测试

测试表明，SD-CD 的组合基于表 1 中展示的 4 种数据中心 SSD 设备。Workload 采用了广泛使用的 YCSB 和 LinkBench。选取 RocksDB, KVell 和 RocksDB-BlobFS (基于 BlobFS 的 RocksDB, BlobFS 是基于 SPDK 开发的文件系统)作为 baseline。实验表明，与 RocksDB 相比（图 2）, SpanDB 最高可以实现 8.8 倍的吞吐率，同时能够降低延迟 9.5-58.3%。

ID	Model	Interface	Capacity	Price	Seq. write bandwidth	Write latency	Endurance (DWPD)
S	Intel SSD DC S4510	SATA	960 GB	$248 $0.26/GB	510 MB/s	37 us	1.03
N1	Intel SSD DC P4510	NVMe	4.0 TB	$978 $0.25/GB	2900 MB/s	18 us	1.03
N2	Intel SSD DC P4610	NVMe	1.6 TB	$634 $0.40/GB	2080 MB/s	18 us	1.03
O	Intel Optane SSD P4800X	NVMe	375 GB	$1221 $3.25/GB	2000 MB/s	10 us	30

表 1 SSD 相关参数

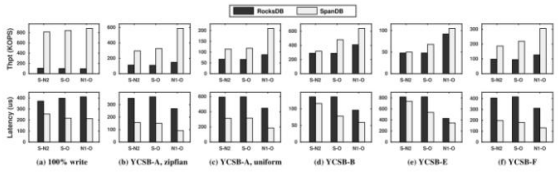

Figure 11: Throughput and latency of various YCSB workloads, 20M requests on 512GB database. (YCSB-A: 50% update and 50% read, YCSB-B: 5% update and 95% read, YCSB-E: 95% scan and 5% insert, YCSB-F: 50% read and 50% read-modify-write)

图 4 与 RocksDB 对比结果

综上所述，SpanDB 兼顾了性价比、数据中心现状、工业界应用等重要因素，具备无缝替代原生 RocksDB 的能力，可以节省数据中心硬件部署成本。

3.2 RemixDB 的研究进展

3.2.1 RemixDB 的查询过程

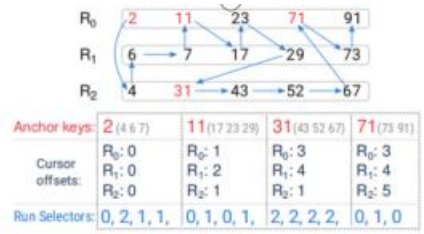

图 5 使用 REMIX 的三个排序运行的排序视图

REMIX 将排序视图涉及到的 key range 按照顺序划分为多个 segment，每个 segment 内保存一个 anchor key，记录该 segment 内最小的 key。

此外，每个 segment 内还维护了 cursor offsets 和 run selectors 两个信息。Cursor offsets 记录了每个 run 中首个大于等于 anchor key 的 key offset，比如 11 这个 segment，run 1,2,3 首个大于等于该值的 key 是 11, 17, 31（offset 1, 2, 1）。Run selectors 顺序记录了 segment 内每个 key 所在的 run 序号，以 71 这个 segment 为例，71, 73, 79 分别在 run 0, 1, 0。

例如找到 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。5. 查询结束。

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

3.2.2 RemixDB 的 segment 的优化

但从上面的查询过程可知，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。Size 增大时，Anchor Keys 就会减少，与之相应的，二分查找的速度就会增快，但 Segment 的平均访问次数也会增多，针对这一现象，SpanDB 针对 segment 内的查询做了额外的优化：

Segment 是对 sorted view 的切分，所以在每个 segment 内，它所维护的 key 之间也是有序的，所以对它也进行二分查找也是很自然的想法。但与 segment 间基于 anchor key 进行查询不同的是，每个 segment 内仅维护了一个 key，所以无法借助 key 比较来查询。

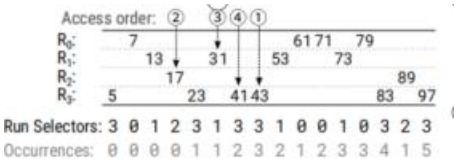

图 6 分段二进制搜索的示例。圆圈数字表示键的访问顺序、

本文借助了 run selectors 来进行二分查询，但由于 selector 仅记录了该 key 归属的 run，并不确定是该 run 中的哪一个 key，所以此处还计算了一份 occurrences 来辅助定位。该值表示的是当前 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。

基于 selectors 的二分以及基于 occurrences 快速跳过 run 内元素，两个方法都能有效地减小 key 之间的比较次数，降低计算的开销。

3.2.3 RemixDB 的 I/O 优化

I/O 优化的方法是进行键比较之后，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。

3.2.4 RemixDB 的性能评估

为了评估 REMIX 的性能，我们实现了一个名为 RemixDB 的基于 LSM 树的 KV 存储。RemixDB 采用分层压缩策略以实现最佳写入效率现实世界的工作负载通常表现出高度的空间局部性。最近的研究表明，分区存储布局可以有效降低现实工作负载下的压缩成本。RemixDB 采用这种方法，将密钥空间划分为非重叠密钥范围的分区。每个分区中的表文件由 REMIX 索引，提供分区的排序视图。这样，RemixDB 本质上是一个使用分层压缩的单级 LSM 树。RemixDB 不仅继承了分层压缩的写入效率，而且在 REMIXes 的帮助下实现了高效读取。RemixDB 的点查询操作（GET）执行查找操作，如果与目标键匹配，则返回迭代器下的键。RemixDB 不使用 Bloom 过滤器。

在每个实验中，我们首先创建一组 H 个表文件(1≤H≤16)，它们类似于 RemixDB 中的一个分区或 LSM-tree 中的一个级别，使用 tiered compaction。每个表文件包含 64MB 的 KV-pairs，其中键和值的大小分别为 16B 和 100B。当 H≥2 时，KV 对可以用两种不同的模式分配到表中:

Weak locality：每个键被分配给一个随机选择的表。

Strong locality：每 64 个逻辑上连续的键被分配给一个随机选择的表。

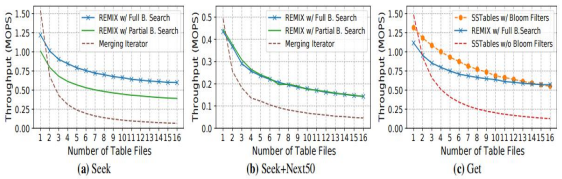

图 7 点和范围查询性能，其中密钥被随机分配（弱位置）

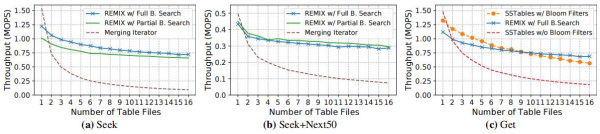

图 8 表上的点和范围查询性能，其中 64 个键分配给表（强位置）

Seek：Merge Iterator 在只有一个 Table 的时候表现更好，因为和 REMIX 需要执行相同次数的二分查找，但是 REMIX 需要动态地计算出现次数，并将迭代器从段的开头移动到一个键进行比较，开销相对更大。随着 Tables 数量增加，REMIX 的性能优势渐渐体现出来了。

Seek+Next50 整体性能低于 Seek ，因为这个操作需要拷贝数据到 Buffer。随着 Tables 数量增加，REMIX 的性能优势渐渐体现出来了。

Get：当少于 14 个 tables 的时候，REMIX 不如带 bloom filters 的 Get.随着 table 数量的增加，REMIX 的性能将逐渐逼近并超越 bloom filters

强局部性的时候在 range query 的结果上差距不大，通常，改进的局部性允许更快的二分搜索，因为在这种情况下，最后几个键比较通常可以使用相同数据块中的键。但是，合并迭代器的吞吐量仍然很低，因为密集的键比较操作占据了搜索时间。带有部分二分搜索的 REMIX 比完全二分搜索的改进更多，是因为局部性的提升减少了在目标 segment 里 scan 的开销，导致更少的缓存缺失。

REMIX 点查询性能也得到了改善，因为强大的局域性加快了底层查找操作的速度。同时，Bloom 过滤器的结果保持不变，因为搜索代价主要由假阳性率和对单个表的搜索代价决定。因此，当包含超过 9 个表时，remix 的性能可以超过 Bloom 过滤器。

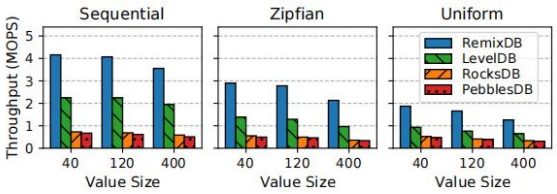

图 9 具有不同值大小的范围查询

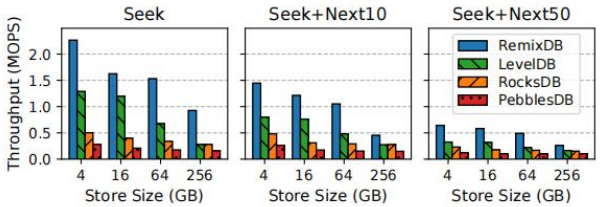

图 10 具有不同存储大小的范围查询

分别在具有不同值大小的范围查询与具有不同存储大小的范围查询中，将其与 LevelDB,RocksDB 以及 PebblesDB 进行对比。

可以发现，RemixDB 都展现出了较高的吞吐率，也就是提供了较高的性能。

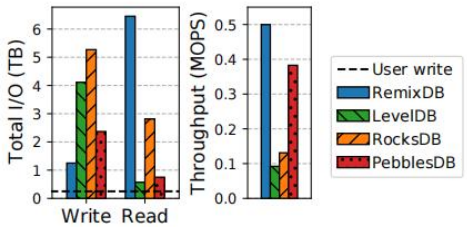

图 11 以随机顺序加载 256 GB 数据集

在用随机顺序加载 256GB 命令集时，RemixDB 也展现出了更高的吞吐率和总 IO。

3.3 RocksDB 的研究进展

3.3.1 RocksDB 的优化方向的转变

随着时间推移，作者及其团队设计的 RocksDB 资源优化目标从写放大到空间放大最后到达了 CPU 利用。

当作者团队开始开发 RocksDB 时，最初遵循当时社区的一般观点，关注的是保存闪存删除周期，从而实现写放大。这对于许多应用程序来说都是一个重要的目标，特别是对于那些具有重写工作负载的应用程序来说。

RocksDB 的写放大出现在两个层次上。首先是 ssd 本身引入了写放大，其次是存储和数据库软件也生成写放大；有时这样的写放大可能高达 100(例如，当小于 100 字节的更改被写入整个 4KB/8KB/16KB 页面时)。

RocksDB 中的 leveled-compaction 通常表现出 10 到 30 之间的写放大，在许多情况下比使用 b-trees 时大好几倍。例如，当在 MySQL 上运行 LinkBench 时，RocksDB 每个事务发出的写数数仅为 InnoDB 基于 b-trees 的存储引擎的 5%。尽管如此，在 10-30 范围内的写放大对于重写的应用程序来说太高了。因此，我们添加了 tiered-compaction，它将写放大降低到 4-10 范围，但却降低了读的性能。

下图描述了 RocksDB 在不同数据摄入率下的写放大情况。

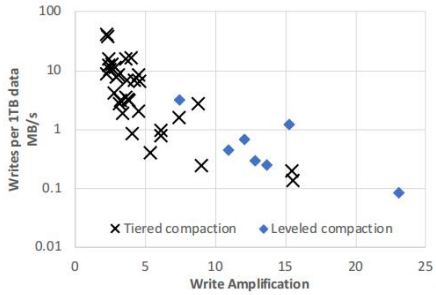

图 12 在 42 个随机抽样的 ZippyDB 和 MyRocks 应用程序中对写入放大和写入速率进行调查。

在实际的开发中，SSD 能提供的 IOPS 远高于实际上用到的 IOPS,因此，磁盘空间的优化比写放大的优化更重要。

lsm -tree 的非碎片数据布局，使得它在优化磁盘空间时也可以很好地工作。于是，作者通过减少 LSM 树中的老数据(即删除和覆盖的数据)的数量来改进 Leveled Compaction。

因此，第三篇文章开发了 Dynamic Leveled Compaction，树中每个 Level 的大小会根据最后一个 Level 的实际大小自动调整，获得了相比于 Leveled Compaction 更好以及更稳定的空间效率。

人们的普遍认知为 SSD 的性能已经高到大部分软件都无法充分利用其潜力了，换而言之，对于 SSD 来说，瓶颈已经从存储设备转移到 CPU 了。

然而基于作者及其团队的经验，这种担忧应当是不被接受的，这种担忧也不应当成为未来基于 NAND 闪存的 ssd 的问题，原因有二。

首先，只有少数应用程序受到 ssd 提供的 IOPS 的限制，大多数应用程序都受到空间的限制。

其次，任何具有高端 CPU 的服务器都有足够的计算能力来饱和一个高端 SSD。RocksDB 在我们的环境中充分利用 SSD 性能而从没有遇到过 CPU 计算能力够的问题。当然，也可以配置一个系统，使 CPU 成为一个瓶颈；例如，一个具有一个 CPU 和多个 SSD 的系统。然而，有效的系统通常是那些配置为良好平衡的系统，这是今天的技术所允许的。密集的以写为主的工作负载也可能导致 CPU 成为一个瓶颈。对于一些人来说，这可以通过配置 RocksDB 以使用更轻量级的压缩选项来缓解。对于其他情况，工作负载可能根本不适合 SSD，因为它将超过典型的闪存耐力预算。

为了证实这一的观点，作者及其团队调查了 ZippyDB 和 MyRocks 的 42 种不同部署，每个部署都服务于不同的应用程序。下图显示了结果。大多数工作负载都受到空间的限制。有些确实是 CPU 负担重，但主机通常没有被充分利用来为增长和处理数据中心或区域级故障（或由于配置错误）留下领先空间。大多数这些部署都包括数百台主机，因此考虑到在这些主机之间自由（重新）平衡工作负载，所以用平均值给出了这些用例的资源需求。

图 13 跨四个指标的资源利用率。每行表示具有不同工作负载的不同部署。在一个月内进行了测量。所有数字是部署中所有主机的平均数。CPU 和读取带宽是一个月中最高的小时数。闪存耐久性和空间利用率是整个月的平均值。

上图跨四个指标的资源利用率。每一行都表示具有不同工作负载的不同部署。共进行了一个月的测量结果。所有的数字都是部署中所有主机的平均值。CPU 和读取带宽是这个月中最高的一个小时。闪存耐力和空间利用率是平均的。

然而，由于减少空间放大的成果，减少 CPU 开销已经成为一个重要的优化目标。减少 CPU 开销可以提高 CPU 确实受到限制的少数应用程序的性能。更重要的是，减少 CPU 开销的优化允许更划算的硬件配置——直到几年前，CPU 和内存的价格相对于 ssd 的价格还相当低，但 CPU 和内存的价格已经大幅增加，因此减少 CPU 开销和内存使用的重要性增加了。在降低 CPU 开销方面，目前还有进一步改进的空间。

3.3.2 RocksDB 的优化过程中的经验教训

大规模分布式数据服务通常将数据划分为分散在多个服务器节点上进行存储。这意味着存储主机将在其上运行许多 RocksDB 实例。这些实例都可以在一个地址空间中运行，也可以在自己的地址空间中运行。

主机可能运行多个 RocksDB，这个现象对资源管理有一定的影响，需要在主机和每个实例之中进行妥善的管理，最重要的是要支持 RocksDB 的优先级化，确保资源优先给最需要它的实例。

其次，大量生成未合并的线程可能存在问题，特别是在线程长寿的情况下。有太多的线程会增加 CPU 的概率，导致过多的上下文切换开销，并使调试极其困难。

同时，大规模分布式存储系统通常为了性能和可用性而复制数据，而且它们具有各种一致性保证。对于这样的系统，RocksDB 的 WAL 写数不那么重要。此外，分布式系统通常有它们自己的复制日志，在这种情况下，根本不需要 RocksDB 的 WAL。

此外，RocksDB 通常通过文件系统与底层设备进行交互。当文件被删除时，会发送一个 TRIM 命令到 SSD。TRIM 命令会更新地址映射，同时还要求 SSD 固件更改写入 FLASH 的 FTL 日志，多个文件的同时删除可能触发 SSD 的内部垃圾搜集机制，导致相当大的数据迁移。因此，需要限制文件删除速率，以便于避免这一现象。

RocksDB 每个月都需要进行更新，或者是当问题出现时，RocksDB 的版本将被回滚。因此，新的 RocksDB 必须理解前一个实例在硬盘上存储的数据。

为了解决不同配置的 RocksDB 无法打开数据文件的问题，RocksDB 首先引入了 RocksDB 实例使用包含配置选项的字符串参数打开数据库的能力。后来，RocksDB 引入了对可选存储选项文件和数据库的支持。

关于故障处理有三个主要教训。首先，需要尽早检测数据损坏，以将数据不可用或丢失的风险降到最低，并据此查明错误的来源。其次，完整性保护必须覆盖整个系统，以防止无声的损坏暴露于 RocksDB 客户端或扩散到其他副本.第三，最好只有在错误无法在本地恢复的情况下才中断 RocksDB 操作，在遇到瞬态错误时定期重试恢复操作。

版本信息应该成为 RocksDB 中的一流公民，以便正确地支持特性，如多版本并发控制(MVCC)和时间点读取。为了实现这一点，RocksDB 需要能够有效地访问不同的版本。

4. 总结与展望（500 字）

4.1 SpanDB 的总结与展望

SpanDB 探索了一种“穷人的设计”，即在 KV 存储最需要的位置部署一个小型且昂贵的高速 SSD，同时将大部分数据留在更大、更便宜和更慢的设备上。我们的结果表明，主流的基于 LSM 树的设计可以显著改进，以利用这种部分硬件升级（同时保留主要的数据结构和算法，以及许多正交优化）。

SpanDB 具有无缝替代 RocksDB 的能力，在更

多的 LSM 树的设计改进被应用后，SpanDB 将在数据中心硬件部署中发挥重要的作用。

4.2 RemixDB 的总结与展望

我们介绍了 REMIX，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，以实现高效的搜索和扫描。基于 Remixes，RemixDB 有效地提高了范围查询性能，同时使用分层压缩保持了较低的写入放大率。

未来，RemixDB 将与其他有关的工作，例如使用过滤器以改进搜索，使用有效索引改进搜索等工作进行结合，以增加 RemixDB 的应用范围，并一定程度上提升 RemixDB 的性能

4.3 RocksDB 的总结与展望

作为主要数据结构的 LSM 树很好地服务于 RocksDB，因为它具有良好的写入和空间放大能力。然而，我们对性能的看法随着时间的推移而演变。虽然写和空间放大仍然是主要关注点，但更多的关注点已经转移到 CPU 和 DRAM 效率以及远程存储上。

同时，未来的 RocksDB 将会优化分散存储、键值分离、多级校验和以及应用程序指定的时间戳。并统一 leveled compaction 和 tiered compaction ，并提高适应性以提升性能。

参 考 文 献

[1] Dong S, Kryczka A, Jin Y, et al. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience[C]//19th {USENIX} Conference on File and Storage Technologies ({FAST} 21). 2021: 33-49.

[2] Dong S, Kryczka A, Jin Y, et al. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience[C]//19th {USENIX} Conference on File and Storage Technologies ({FAST} 21). 2021: 33-49.

[3] Zhong W, Chen C, Wu X, et al. {REMIX}: Efficient Range Query for LSM-trees[C]//19th {USENIX} Conference on File and Storage Technologies ({FAST} 21). 2021: 51-64.