Autumn：一种可扩展、读取优化的基于LSM树的键值存储，具有快速的点读取和范围读取速度

Fuheng Zhao 加州大学圣塔芭芭拉分校 f uheng_zhao@ucsb.edu

Zach Miller 加州大学圣塔芭芭拉分校 zmiller@ucsb.edu

Leron Reznikov 加州大学圣塔芭芭拉分校 lreznikov@ucsb.edu

Divyakant Agrawal 加州大学圣塔芭芭拉分校 agrawal@cs.ucsb.edu

Amr El Abbadi 加州大学圣塔芭芭拉分校 amr@cs.ucsb.edu

arXiv:2305.05074v2 [cs.DB] 1 Jul 2024

摘要

基于日志结构合并树（LSM 树）的键值存储广泛应用于许多存储系统，以支持各种操作，例如更新、点读取和范围读取。传统上，LSM 树的合并策略会将数据组织到多个容量呈指数级增长的层级中，以支持高速写入。然而，我们认为，传统合并策略并未针对读取进行优化。在这项工作中，我们提出了 Autumn，这是一种可扩展且针对读取优化的基于 LSM 树的键值存储，其最坏情况下的点读取和范围读取开销接近最优。提升读取性能的关键思路是，随着存储的数据不断增加，动态调整两个相邻层级之间的容量比。因此，较低层级的容量会逐渐增大，并更频繁地进行合并。具体而言，通过采用新颖的 *Garnering* 合并策略，Autumn 将点读取和范围读取开销从此前已知的最佳复杂度 $O(logN)$ 降至 $O(\sqrt{logN})$。*Garnering*合并策略在优化点读取和范围读取的同时，也能保持较高的写入性能，因为它会优先安排较低层级的合并；Garnering 为较低层级安排了更多合并操作。我们在 RocksDB 和 LevelDB 之上实现了 Autumn，并通过实验展示了它在真实工作负载下的性能提升。

PVLDB 引用格式：
Fuheng Zhao、Zach Miller、Leron Reznikov、Divyakant Agrawal 和 Amr El Abbadi。Autumn：一种可扩展、读优化的 LSM 树键值存储，支持快速点读取和范围读取。PVLDB，14(1)：XXX-XXX，2020。doi:XX.XX/XXX.XX

PVLDB 工件获取：
源代码、数据和/或其他工件已在 https://github.com/ZhaoFuheng/rocksdb 上提供。

1 引言

日志结构合并树（LSM 树）是现代键值存储和大规模数据系统存储层中广泛使用的一种结构。LSM 树通过设计实现了始终如一的高写入吞吐量，其方式是采用异地更新。它会将所有更新缓存在主存中，将更新写入预写日志（WAL），并在缓冲区达到容量上限时，将所有缓冲的更新作为一个有序运行段刷新到磁盘。此过程将随机磁盘写入转变为顺序磁盘写入，显著提升了写入性能。因此，LSM 树已成为工业级键值存储引擎的基石 [11]，包括 BigTable [10]、Dynamo [23]、HBase [28]、Cassandra [38]、RocksDB [25]和 LevelDB [30]。此外，基于 LSM 树的存储也常用于图处理 [54]、流处理 [13]、在线事务处理（OLTP）[47]，以及混合事务分析处理（HTAP）中对实时新鲜数据分析的支持。ByteHTAP [14]和 TiDB [31]等常见 HTAP 架构使用两个独立的存储引擎，分别处理在线事务处理（OLTP）和在线分析处理（OLAP）工作负载。在 ByteHTAP [14]中，数据操作语言（DML）、数据定义语言（DDL）和简单查询会发送至其 OLTP 引擎。同样，在 TiDB [31]中，小型范围查询和点查询可发送至 TiKV 存储引擎 [51]；该引擎是一种采用面向行布局、基于 LSM 树的键值存储引擎，专门用于处理此类工作负载。

LSM 树将数据组织到多个容量递增的层级中，最高层存储了大部分数据。LSM 树中的合并策略明确规定了数据何时应在两个相邻层级之间移动。许多现有研究通过引入新的合并策略，致力于最大限度地降低空间放大 [24]、减少写放大 [19]，并优化点查询 [18]。LSM 树中的合并策略隐含地定义了存储效率（即，采用异地更新时，过时的数据条目可能存储在不同层级中）、更新的输入/输出成本以及读取的输入/输出成本之间的权衡。主流的 LSM树存储引擎通常采用分级合并或*Leveling* 合并策略 [25, 30, 38, 51]。一方面，分级合并策略逐步对某一层中的有序段进行排序合并，侧重于优化写入。另一方面，Leveling 策略会主动对某一层中的有序段进行排序合并，侧重于优化读取。不过，我们认为现有设计对范围读取的优化并不理想，因此提出采用 Garnering 策略的 Autumn，以提供优于现有方案的读取性能。

除了合并策略之外，现有优化读查询的解决方案主要采用更好的缓存策略 [58, 61] 以及概率数据草图/过滤器 [37, 46, 66, 68, 70]，这些过滤器存储在主存中，通过跳过搜索键范围内不含记录的有序段来减少磁盘 I/O。这些方法与我们的工作相互正交。然而，当每个有序运行段都可能包含部分范围查询结果、因而必须访问所有有序段时，范围过滤器并不能提升

本作品采用知识共享署名-非商业性使用-禁止演绎 4.0 国际许可协议授权。访问 https://creativecommons.org/licenses/by-nc-nd/4.0/ 查看该许可协议副本。如需进行该许可协议未涵盖的任何使用，请发送电子邮件至 info@vldb.org 获取许可。版权所有者为作者。出版权已授权给 VLDB 基金会。VLDB 基金会会刊，第 14 卷，第 1 期，ISSN 2150-8097。doi:XX.XX/XXX.XX

范围查询性能。此外，范围过滤器无法同时提供良好的范围过滤器查询功能，以及点过滤器查询所需的低假阳性率。在过滤器内存预算固定时，与传统点过滤器 [7]相比，范围过滤器在点过滤器查询上的性能较差。因此，当工作负载包含大量点查询时，范围过滤器无法直接取代许多 LSM 树存储引擎中的点过滤器。

与以往的工作不同，Autumn 致力于同时提升范围查询和点查询的性能，在保证强有力的最坏情况理论复杂度的同时，保持更新的可扩展性。Autumn 采用新的 *Garnering* 策略来优化范围查询，并在现有合并策略中实现最佳的范围查询复杂度。Autumn 还采用受 Monkey [17] 启发的布隆过滤器优化，实现了最先进的点查询性能。凭借低延迟的点查询和短范围查询，以及可扩展的更新成本，Autumn 非常适用于在线事务处理和混合事务与分析处理工作负载。具体而言，我们提出 Autumn，这是一种基于 LSM 的存储引擎，旨在优化键值存储中的读取，并具有以下目标：

- **高性能范围查询**。范围读取对分析和在线事务处理工作负载十分重要。Autumn 致力于从根本上改善范围读取的最坏情况复杂度，尽可能减少磁盘访问次数。• **快速点查询**。点查询十分常见。当点查询未使用概率型点过滤器来跳过有序段时，Autumn 也应改善点查询的最坏情况性能。此外，概率型数据结构在提升点查询性能方面发挥着重要作用。与以往的工作一样，Autumn 利用并优化存储在主存中的点过滤器，以提高点查询性能。• **可扩展性**。随着存储的键值对不断增加，Autumn 应能够扩展，并提供良好的空间放大率和更新性能，同时在读取操作方面表现出色。

本文提出了新颖的键值存储 Autumn。它采用 Garnering 策略重新设计了传统的 LSM 树数据组织，并着重优化读取，同时确保更新具备可扩展性。总而言之，本文的主要贡献如下：

(1) 我们介绍了采用全新 Garnering 合并策略的 Autumn，以重新设计 LSM 树中的数据组织。在所有现有合并策略中，Garnering 的范围查询成本最低；在不使用布隆过滤器时，它能提升点查询性能；在使用布隆过滤器时，则能保持业界领先的点查询成本。(2) 我们应用磁盘 I/O 成本模型，对范围查询、点查询、写放大和空间放大的复杂度进行全面的理论分析。(3) 我们发现，通过在 LSM 树的第一层维持固定数量的有序运行段，可以避免第一层的所有合并，同时不影响读取复杂度，从而进一步优化 Autumn 的写入和更新性能。事实上，LevelDB 和 RocksDB 已实现了这种方法，并且这种方法能显著降低 Autumn 的写放大。由于随着存储的数据条目增多，Autumn 会逐步提高较低层级的容量，因此，由第一层引发的压实占总写放大的很大一部分。

(4) 我们进行了全面的实验，使用默认基准测试 [30] 以及雅虎！的 YCSB 基准测试来模拟真实世界的工作负载 [15]，以展示 Autumn 在读取方面的高性能，以及其写入和读取的可扩展性，并与 RocksDB 和 LevelDB 进行了比较。

本文的组织结构如下：第 2 节介绍 LSM 树的背景信息，并概述现有合并策略及其特性。第 3 节介绍我们提出的、可扩展且针对读取进行优化的键值存储 Autumn，以及新颖的 Garnering 合并策略，并详细分析 Autumn 在空间放大、读取和更新开销方面的复杂度界限。第 4 节介绍使用 YCSB 基准测试进行评估的实验结果。第 5 节讨论与 Autumn 正交的相关工作，范围涵盖范围过滤器到新型硬件。最后，第 6 节总结本文的贡献并得出结论。

2 LSM 树背景

本节将介绍有关 LSM 树的背景信息。O'Neil 等人 [50] 于 1996 年提出 LSM 树，通过异地更新减少数据库索引结构在更新过程中的随机磁盘访问。在 LSM 树出现之前，差分文件、Postgres 项目和日志结构文件系统 [52, 53, 55] 已成功支持异地更新：先收集大量数据，再通过一次大型输入/输出将数据写入文件。然而，这些将数据顺序写入仅追加日志的方法会损害查询性能，而且缺乏对写入、读取和空间利用率之间权衡的系统性成本分析。表 1 列出了本文用于分析 LSM 树的重要符号。

LSM 树提供了一种系统的方法来分析写入、读取和空间利用率之间的权衡。LSM 树将数据物理存储在 L 层中，其中第 0 层位于内存中，其余层位于持久性存储中。每一层的容量（即每层可存储的数据量）呈指数级增长。如公式 1 所示，第 i 层的容量是第 $i-1$层容量的 T 倍。

$$C_i/C_{i-1} = T \tag{1}$$

在原始设计中， [50], 每一层都采用 a B^+-Tree 结构；当第 i 层达到其容量时，滚动合并策略会将第 i 层的一段 B^+-Tree叶页移至第 $i+1$层。在现代 LSM 树中，新的更新始终存储在第0层内存表（内存表）中，该内存表通常采用跳表实现，以支持点读取和范围读取。在更新过程中，如果已存储具有相同键的数据条目，新值将替换旧值；如果键被删除，LSM 树会为该键关联一个墓碑标记。内存表已满时会变为不可变，存储在内存表中的数据将被刷新为一个有序的**运行段** ，并存储在位于磁盘上的第1层静态有序表（SST）中。从逻辑上看，运行段是一个按键排序的数组，

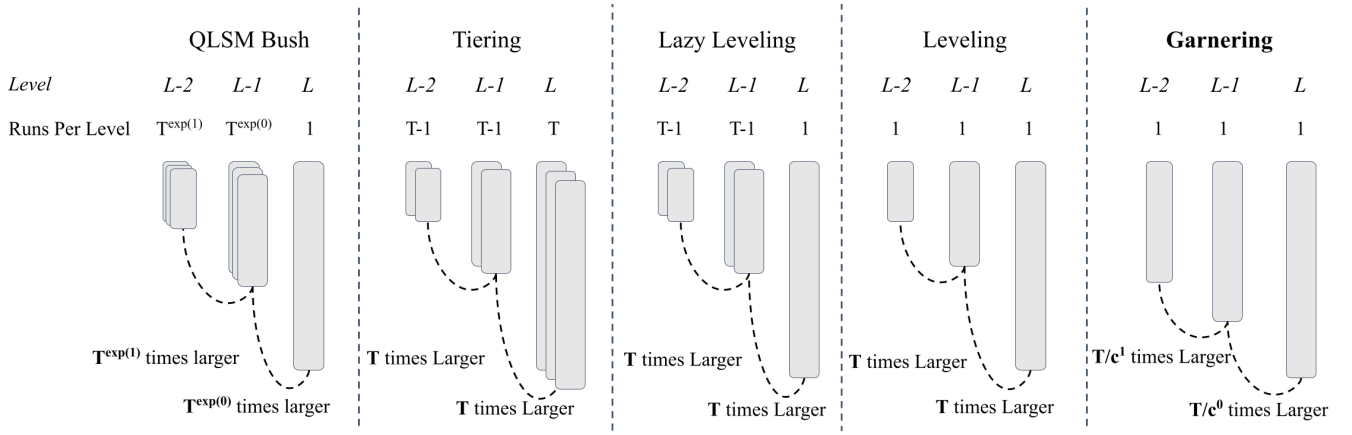


图 1：QLSM-Bush、分级合并、Lazy-Leveling、Leveling 和我们提出的 Garnering 中最后三个最大层级的示意图。左侧更针对写入优化，右侧更针对读取优化。

从物理上看，它由一个或多个SST文件组成。当存储在第 i层的数据达到第 i层的容量时，第 i层的数据文件将被合并到容量更大的第 $i+1$层。在合并过程中，对于键重复的条目，将保留较新的值并丢弃较旧的值。

N	数据大小
B	内存表大小
L	LSM 树中的层级数
N_i	层 i的数据大小
C_i	i层的容量
T	合并策略中的大小比例
c	*Garnering* 策略中使用的缩放比例。
M_{filter}	布隆过滤器的内存预算

表 1：符号表

2.1 并发控制与恢复

可用性的一项严格要求是，确保存储引擎在处理并发请求（读取和更新）以及机器崩溃时的正确性。LSM 树通常采用多版本并发控制协议，而非加锁，以避免争用 [45]。刷新操作或压缩完成后，相关文件会生成新版本，而旧版本的过时文件则可进行垃圾回收。点查询和范围查询会先获取最新版本的一组文件，再利用这组文件返回所需且一致的结果。由于所有更新都会先缓冲在主存中，因此 LSM 树自然会使用预写日志（WAL），以确保机器发生故障时数据仍具持久性。此外，还有一个元数据日志用于存储 LSM 树结构的状态。在恢复阶段，键值存储随后会根据事务日志重做所有已提交的事务，并可利用元数据日志恢复磁盘上存储的所有文件。

2.2 LSM树放大

写放大 写放大表示写入成本，衡量向 LSM 树中添加一个数据条目所需的摊销磁盘写入次数。LSM 树将数据组织在多个层级中。较小的层级收集数据，并通过一次较大规模的顺序 I/O，将收集到的数据写入较大的层级。因此，随着 LSM 树不断增长，数据条目会被多次写入存储设备。反复写入会对现代固态存储设备（SSD）的使用寿命产生负面影响 [39, 49]。因此，高写放大可能会损耗设备。为减少写放大的程度，研究人员研究并成功开发了多种机制，例如分离键和值 [44]，以及在主存和磁盘之间引入非易失性内存层 [33]。为了避免较大层级中耗时的合并操作导致写入停顿，基于LSM树的键值存储（如 LevelDB [30] 和 RocksDB [25],）会执行部分合并，即在两个相邻层级之间移动静态有序表文件（2–64 MB），而不是移动整个有序运行段。这种机制试图将合并开销分摊到一段时间内，但可能会对合并操作的最坏情况下更新I/O成本造成负面影响 [22]。采用完整合并时，相邻层级中的两个有序段会一次性完全合并；Cassandra [38] 和 HBase [28] 默认使用完整合并。本文讨论合并操作的复杂度时，均以有序段为粒度。

点查询放大点查询通过遍历 LSM 树的所有层级，找到某个数据条目的最新版本。由于最新的信息存储在较低层级，因此查询一旦找到键匹配的条目便会终止。加速点查询的一种常见做法是为每个有序运行段配备一个**布隆过滤器** [7],，并将其存储在主存中。布隆过滤器是一种节省空间的数据摘要，可用于回答成员查询。此外，它能保证不会出现假阴性，即如果成员查询返回 false，则该数据条目不在数据集中；它还具有较低且可调节的假阳性率（FPR），即如果成员查询返回 true，该数据条目仍可能不在数据集中。FPR 取决于每个数据条目所使用的比特数，每个数据条目使用 10 比特时，

数据条目对应的 FPR 约为 1% [57], , 如公式 2 所示，其中 M_{filter} 是过滤器内存大小， N 是数据条目的数量。

$$FPR = e^{-ln(2)^2 M_{filter}/N} \tag{2}$$

范围查询放大 范围查询用于查找从起始目标键开始，直至满足特定条件（例如迭代器到达结束键边界，或迭代器返回的键达到指定数量）为止的所有数据条目的最新版本。与找到目标键后即可返回的点查询不同，范围查询需要检查所有层，以确保目标键范围内没有遗漏任何键。合并迭代器会从与各个有序运行段关联的迭代器中收集目标键范围内相关数据条目的最新版本，而每个有序运行段的迭代器负责读取数据条目，直到满足终止条件。为了加快迭代器的查找速度，一种常见做法是在主存中存储**栅栏指针** 。栅栏指针在主存中存储每个运行段中每个数据块的第一个键，从而能够通过一次磁盘 I/O 定位某个运行段中的相关键范围。

空间放大由于 LSM 树采用异地更新，过时的数据条目仍存储在磁盘上，从而增加了总存储量。因此，LSM 树所需的存储空间大于逻辑数据大小。空间放大表示为物理存储大小与逻辑存储大小之比。传统上，由于存储设备价格低廉，人们很少关注空间放大。在云原生数据库时代，空间放大可能会增加运营成本。为了全面呈现设计空间中的各项指标，我们也力求将空间放大控制在较小水平。

2.3 合并策略

合并策略决定何时应在 LSM 树中的两个相邻层级之间合并数据。此外，合并策略会直接影响点查询开销、范围查询成本、写入成本和空间放大。这些指标之间存在固有的权衡。

2.3.1 Leveling 和分级合并。 图 1 展示了最先进的合并策略。Leveling 和分级合并合并策略以指数方式增加层级容量，从小层级到大层级，其中两个相邻层级的容量相差一个固定因子 T （一个可调参数）和 $T > 1$。

层级数。 在传统策略中：由于层的容量呈指数增长，最后一层的容量为 BT^L，其中 B 是内存表大小，L 是层级总数。除最后一层之外的所有层中存储的数据总量约等于最后一层中存储的数据量，而最后一层中存储的数据量为 $N\frac{T-1}{T}$，其中 N 是 LSM 树中存储的~~数~~据大小。因此，通过求解方程 $BT^L = N\frac{T-1}{T}$ 以得到 L，层级数可以计算为 $O(log_T(N/B))$，如 3 所示。

$$L = O(log_T(\frac{N}{B} \cdot \frac{T-1}{T})) \tag{3}$$

在 **Leveling** 合并策略中，每个层最多有一个有序运行段。新有序运行段到达后会立即触发合并。当层 l 达到其容量 C_l 时，层 l 的有序运行段将进行压缩，使层 l 的运行段与层 $l+1$ 的运行段按序合并。因此，在达到层容量之前，这种设计要求数据在每个层内平均合并 $O(T)$ 次。写放大可推导为 $O(T \cdot L)$。对于范围查询，需要访问所有有序运行段。由于每层只有一个有序运行段，范围查询成本为 $O(L)$ 次磁盘 I/O。最坏情况下的点读查询成本发生在目标键未存储时。因此，最坏情况下需要访问所有有序运行段，成本为 $O(L)$ 次磁盘 I/O。每个有序运行段附带一个布隆过滤器时，磁盘 I/O 次数等于所有布隆过滤器的假阳性率之和。假设所有布隆过滤器对每个数据条目使用相同数量的位，那么最坏情况下的点读查询成本为 $O(L \cdot e^{-M_{filter}/N})$。

在**分级合并**合并策略中， [38], 每层最多可存储 T 个有序运行段。当第 l 层满时， T 个有序段将进行排序合并，然后作为一个新的有序运行段存储在第 $l+1$ 层。因此，在达到层容量之前，此设计在每层只进行一次合并，写放大为 O（L）。范围查询需要访问所有有序段，因此开销为 O（$T \cdot L$）次磁盘 I/O。对于最坏情况下的点读查询开销，假设不使用布隆过滤器，点读取开销为 O（$T \cdot L$）次磁盘 I/O。如果所有布隆过滤器使用相同的位数，最坏情况下的点读取开销将变为O（$T \cdot L \cdot e^{-M_{filter}/N}$），因为每层有 T 个布隆过滤器。一方面，Leveling 策略会主动对有序段进行排序合并，以减少有序段总数，因此更注重读取性能。另一方面，分级合并策略会被动地对有序段进行排序合并，以降低更新成本。

布隆过滤器优化。 业界的 LSM 树实现传统上会在所有层级统一设置假阳性率，即每个层级的每个数据条目分配 $O(ln(1/\epsilon))$ 位。结果是，为了节省一次磁盘 I/O，较大的有序运行段需要指数级更多的过滤器内存预算，因为栅栏指针可以定位给定的数据块。对于点查询，最坏情况下的磁盘 I/O 次数，是目标查询键未存储在数据库中的所有有序段的假阳性率之和。Monkey [17] 提出了一种更好的方法，即降低较低层级的假阳性率。其核心直觉是，每一层访问磁盘的成本相同。因此，降低较低层级的假阳性率在内存预算方面效率更高，因为这些层级的数据条目数量呈指数级减少；与较大的层级相比，它们只需更少的过滤器内存，就能实现较低的假阳性率。因此，这种设计将最坏情况下的点读取查询复杂度提升了 L 倍。采用布隆过滤器优化后，Leveling 和分级合并的最坏情况点读取查询成本分别为 $O(e^{-\frac{M_{filter}}{N}})$ 和 $O(T \cdot e^{-\frac{M_{filter}}{N}})$，与层级数无关，仅取决于过滤器内存预算 ($M_{filter}$)。

2.3.2 其他合并策略。 一系列工作 [18, 19] 致力于提出新的合并策略，以在保持可扩展写入性能的同时，降低点查询开销。然而，这些策略并未实现完全的读取优化，因为它们只改善了点读取，而没有改善范围读取。具体而言，Lazy-Leveling [18] 同时借鉴了 Leveling 和分级合并的思想，将最后一层保留为一个有序运行段，而其他层均保留为 T 有序运行段。因此，写放大变为 $O(T + \frac{log_T(N/B)}{B})$，范围读取开销为 $O(1+$

策略	范围查询	无过滤器点查询	带过滤条件的点查询	写入	空间
QLSM Bush	$O\left(\sqrt{T\cdot\frac{N}{B}}\right)$	$O\left(\sqrt{T\cdot\frac{N}{B}}\right)$	$O\left(T\cdot e^{-\frac{M_{filter}}{N}}\right)$	$O(1+log_2(log_T(\frac{N}{B})))$	$O(1)$
分级合并	$O(T\cdot log_T(\frac{N}{B}))$	$O(T\cdot log_T(\frac{N}{B}))$	$O\left(T\cdot e^{-\frac{M_{filter}}{N}}\right)$	$O(\frac{log_T(\frac{N}{B})B}{})$	$O(T)$
Lazy-Leveling	$O(1+T\cdot log_T(\frac{N}{B}))$	$O(1+T\cdot log_T(\frac{N}{B}))$	$O(e^{-\frac{M_{filter}}{N}})$	$O(T+\frac{log_T(\frac{N}{B})B}{})$	$O(\frac{T+1}{T})$
Leveling	$O(log_T(\frac{N}{B}))$	$O(log_T(\frac{N}{B}))$	$O(e^{-\frac{M_{filter}}{N}})$	$O(T\cdot\frac{log_T(\frac{N}{B})B}{})$	$O(\frac{T+1}{T})$
Garnering（我们的方法）	$O(\sqrt{\frac{log_c(\frac{N}{B}\cdot T)}{}}),\ c<1)$	$O(\sqrt{\frac{log_c(\frac{N}{B}\cdot T)}{}})$	$O(e^{-\frac{M_{filter}}{N}})$	$O(T/c^{\sqrt{log_c(\frac{N}{B}\cdot T)}})$	$O(\frac{T+1}{T})$

表 2：现有策略与 Garnering 之间的复杂度分析。

$T\cdot\ log_T(N/B)$)，其范围读取性能劣于 Leveling。QLSM Bush [19] 以牺牲范围读取为代价，将写放大降至最佳水平 $O(1+log_2(log_T(N/B)))$，范围读取开销则为 $O(\sqrt{TN/B})$。如图 1 所示，QLSM Bush 在较低层级组织了数量呈双指数增长的有序段，从而实现极低的写放大。因此，范围读取开销变为 $O(\sqrt{T\cdot N/B})$，其中最小层大约包含 $O(\sqrt{T\cdot N/B})$个有序运行段。通过布隆过滤器优化，Lazy-Leveling 和 QLSM Bush 降低了点查询开销（$O(e^{-\frac{M_{filter}}{N}})$），分别如此。显然，现有的 LSM 策略并未针对范围查询性能进行优化。相比之下，Autumn 专注于同时优化点查询和范围查询。通过引入**Garnering** 合并策略，该策略在层级之间使用额外的缩放因子c，Autumn 在保持空间放大和写放大可扩展性的同时，实现了有无布隆过滤器两种情况下的最优点查询开销，并提供了最优的最坏情况下的范围读取性能。在下一节中，我们将更详细地介绍新的 Garnering 策略以及 Autumn 的架构。表 2 展示了不同策略之间的权衡。

3 AUTUMN

本节将介绍采用 Garnering 合并策略的 Autumn 新设计，该设计针对点查询和范围查询进行了优化，并在写入性能和空间放大方面表现出色。

3.1 一种新设计

在 Autumn 中，我们重新思考了 LSM 树的数据组织，并提出了一种新的合并策略 Garnering。我们的目标是在保持高写入性能的同时，从根本上改善点读取和范围读取的最坏情况复杂度。应用布隆过滤器时，先前的研究通过调整各层的布隆过滤器内存预算，将最坏情况假阳性率降至最低，从而提高了点读取效率 [17]。此外，大量研究致力于减少写放大 [19, 44, 63]。然而，优化点读取和范围读取的成本仍然是一项挑战。我们的设计着重于同时优化范围查询和点查询，无论是否使用布隆过滤器，并为更新提供高吞吐量。我们的主要直觉是，要提高范围查询效率，就需要在数据库大小增长时更积极地减少层级数。此外，为了确保该设计具有可扩展性，写放大需要关于 N 具有次线性复杂度。

Garnering 高层设计。 Garnering 不再以常数因子 T 指数级地增加每一层的容量，而是通过将最后一层 L 与第 $L-1$ 层之间的容量比差异固定为常数因子 T，并以缩放因子 c 增加较低层级之间的容量比差异来展平 LSM 树，其中 c 小于 1，如公式 4 所示。将 $c<1$ 设为该值的原因，在分析写放大时便会显而易见：不同层级的写放大之和构成一个几何数列，因此小层级的压实开销占主导地位。从本质上说，随着存储的数据增多，相邻两层之间的容量比差异会动态变化。

$$C_i/C_{i-1}=T/c^{L-i} \tag{4}$$

第 i 层的容量遵循方程 5。当第 i 层存储的数据达到其容量时，第 i 层的文件将合并到第 $i+1$ 层。因此，随着数据存储量增长，与传统的数据组织方式相比，较低层级将拥有更大的容量。

$$C_i=T^i/c^{(2L-1-i)i/2}\cdot B \tag{5}$$

延迟末级压缩。传统上，随着存储的数据增多，每层的容量不会发生变化，也不依赖于层级数。然而，在我们新的Garnering策略中，随着存储的数据增多，我们会逐步增大较低层级的容量。因此，每层的容量取决于总层级数。通常，当最后一层 l 已满时，应创建新的一层 $l+1$。由于在 Garnering 中，创建新层时各层的容量会增大，因此无需压缩第 l 层，因为创建新层后，相对于总层级数 $l+1$，第 l 层的新容量严格大于已存储的数据量，即 $C_l>N_l$。因此，当最后一层已满且需要创建新层时，我们只将总层级数增加一，然后将压缩延迟到下一个周期。通过这种方法，我们避免了不必要的末级压缩，并提升了写入性能。**空间放大**。回顾一下，空间放大定义为物理存储大小与逻辑存储大小的比值；空间放大越小，磁盘存储效率越高。由于每层只包含一个有序运行段，在最坏情况下，存储在第 1 层到第 $L-1$ 层中的所有条目都是对第 L 层中已有数据条目的更新。为分析空间放大的最坏情况复杂度上界，

我们首先观察到，从第 1 层到第 $L-1$ 层存储的所有数据总和受第 $L-1$ 层的容量主导，即 C_{L-1}。我们还知道，最后一层 L 是第 $L-1$ 层的 T 倍。因此，在最坏情况下，最后一层中有 $O(1/T)$ 的条目已失效。因此，空间放大为 $O(\frac{T+1}{T})$。

在分析中，我们假设所有层级都已填满至其容量。实际上，当最后一层未填满至其容量时，空间放大可能会更大。因此，为了尽量减少空间放大，基于LSM树的键值存储通常会根据最后一层所存储的数据量，动态调整较低层级的容量（强制填满最后一层），使得第 1 层至第 $L-1$ 层所存储的数据仅占最后一层所存储数据的一小部分 [22, 24]。

Garnering 中的层数。 由于容量由公式 4 建模，最后一层的容量为 $T^L/c^{(L(L-1)/2)}\cdot B$。由于最后一层的容量 L 是第 $L-1$ 层容量的 T 倍，最后一层的数据大小约为 $N\cdot\frac{T-1}{T}$。因此，解方程 $T^L/c^{(L(L-1)/2)}\cdot B = N\cdot\frac{T-1}{T}$（其中未知数为 L）后，可得层数为 $O(\sqrt{-log_c(\frac{N}{B}\cdot T)})$，如公式 6 所示。~~由于 c 小于 1~~，正实数的 log_c 会得到负值，因此平方根内出现负号。 [1]

$$L = O(\sqrt{-log_c(\frac{N}{B}\cdot\frac{T-1}{T})} - log_{1/c}(T)) \tag{6}$$

$$L = O(\sqrt{-log_c(\frac{N}{B\cdot T})})$$

点查询。 在 LSM 树中，最坏情况下的点读取开销为 $O(L)$，且查找次数为零。当目标键未存储在存储引擎中时，必须检查所有层级才能返回答案（即未找到该键）。如公式 3.1 所示，~~，~~由于 Autumn 的层级数为 $O(\sqrt{logN})$，配合 Garnering 的 Autumn 将点查询的性能提升至 $O(\sqrt{logN})$ 复杂度。此外，布隆过滤器可以加快点查询速度，让系统跳过不包含目标键的有序段。过滤器每个数据条目约需 2 到 10 位主存空间，以此节省磁盘I/O，但会产生一定的主存开销。

布隆过滤器优化。 我们展示了 Garnering 如何通过在不同层级之间精细调整布隆过滤器内存预算，实现与 Leveling 相同的最坏情况点查询复杂度。推导最优布隆过滤器内存分配的方法受 Monkey [17] 启发。

零结果查找所浪费的预期探测次数，等于所有布隆过滤器的假阳性率之和。在 Garnering 中，每层包含一个有序运行段。点读取开销 R 可使用公式 7 计算，即对从第 1 层到第 L 层的所有假阳性率求和。

$$R = \sum_{i=1}^{L} p_i \tag{7}$$

其中，p_i 是第 i 层布隆过滤器的假阳性率，p_i 介于 0 和 1 之间。

然后，我们可以通过对每一层的过滤器内存求和来计算过滤器内存总量。根据公式 2，我们可以重新排列各项，使得层 i 所需的过滤器内存由以下因素决定。

[1]注意，$-log_c(x) = ln(x)/-ln(c) = ln(x)/ln(1/c) = log_{1/c}(x)$。

p_i和 N_i。层 i所需的过滤器内存为 $M_i = -N_i\,^{ln}{}_{(p_i)ln(2)^2}$。每层的过滤器内存大小与该层的容量及相应的假阳性率成正比，因此我们可以应用公式 4 来计算较低层级与最高层之间的比率因子。由此，过滤器内存总大小如公式 8 所示。

$$M_{filter} = -\frac{N}{ln(2)^2}\cdot\frac{T-1}{T}\sum_{i=1}^{L} ln(p_i)\frac{c^{(L-i)(L-i-1)/2}}{T^{L-i}} \tag{8}$$

然后，我们可以将布隆过滤器的最优内存分配表述为一个优化问题。目标是在给定总内存预算 M_{filter}（如公式 8 所示）的情况下，最小化读取开销 R；该开销是所有层的假阳性率之和，如公式 7 所示。对于等式约束，我们采用常用的拉格朗日方法 [6] 来最小化开销。对拉格朗日表达式关于 $\{p_1, p_2, ..., p_L\}$ 求偏导，并令导数为零，即可推导出各层的最佳假阳性率与 p_L之间的函数关系，如公式 9 所示。

$$p_{L-i} = p_L\cdot\frac{c^{i(i-1)/2}}{T^i} \tag{9}$$

将公式 9 代入读取开销 R 中的每个假阳性率，即可改写为

$$R = p_L + p_L\cdot\frac{c^0}{T} + p_L\cdot\frac{c^{0+1}}{T^2} + p_L\cdot\frac{c^{0+1+2}}{T^3} + ...$$

仔细考察上面读取开销的新表达式，我们发现，如果分子始终为 1，那么这就是一个几何级数。这里，分子是 c 的某个幂。由于 c 是小于 1 的较小常数，这个级数实际上比几何数列收敛得快得多；随着 L 增大，读取开销会迅速收敛到 $R = O(p_L)$。

因此，给定期望的读取开销，我们可以推导出最优假阳性率，如公式 10 所示。

$$p_i = O(R\cdot\frac{c^{(L-i)(L-i-1)/2}}{T^{L-i}}) \tag{10}$$

过滤器内存预算。 假设过滤器内存预算固定，那么随着数据量增长，假阳性率会增大，因为过滤器的假阳性率与 M_{filter}/N 呈线性关系。最终，第 L 层的假阳性率将收敛至 1。因此，分析每个数据条目的比特数之比 M_{filter}/N 很有用，这样便可将读取开销 $R = O(1)$ 以及最后一层的假阳性率设为 1。将公式 9 中经过优化分配的 p_i 代入公式 8，并取 $\sum_{i=1}^{L} ln(p_i)\frac{c^{(L-i)(L-i-1)/2}}{T^{L-i}}$ 求和中的主导项，即可得到闭式解

$$\frac{M_{filter}}{N} \approx \frac{-1}{ln(2)^2}(\frac{1}{T}ln(\frac{1}{T}p_l) + \frac{c}{T^2}ln(\frac{c}{T^2}p_l)$$

令 p_l等于 1，并令偏导数等于零，我们可以发现上述函数在每条目 1.52 比特处取得最大值。通过布隆过滤器优化，实现 $O(1)$ 零查找复杂度所需的内存预算非常可观。在业界，标准的 LSM 树键值存储引擎通常为每个数据条目使用 10 比特，因此假阳性率均低于 1。当内存预算受限时，例如在移动设备或物联网设备上，读取

与在所有层级均匀分配固定数量的每条目比特数相比，采用布隆过滤器优化后，点查询会快得多。

CPU 优化。 每个有序段都关联一个存储在主存（动态随机存取存储器）中的布隆过滤器，以减少磁盘 I/O。研究人员观察到，计算能力的提升速度已经放缓 [59],，因此过滤器的中央处理器成本未来可能成为新的瓶颈 [20, 21, 74]。Autumn 并未设计新的过滤器来降低中央处理器成本，而是通过在内存中存储更少的布隆过滤器来直接降低中央处理器成本，因为有序段数量从 $O(logN)$ 减少到 $O(\sqrt{logN})$（请回想每层有一个有序段）。因此，虽然 Leveling 和 Garnering 在磁盘 I/O 次数方面的最坏情况点读复杂度相同，但 Garnering 的中央处理器成本更低，性能优于 Leveling；实验也验证了这一点。

范围读取成本。 Autumn 致力于同时优化点读取和范围读取。范围读取可分为两类：i）短范围读取；ii）长范围读取。短范围读取意味着，对于每个有序段，目标键范围内的数据条目平均存储在磁盘上的一个数据块中。因此，每个有序段都存在一个可能需要检查的数据块，以执行短范围读取。因此，Garnering 将短范围读取的复杂度从 Leveling 中的 $O(logN)$ 降低到 $O(\sqrt{logN})$，因为 Garnering 的层级数少于 Leveling。

范围读取较长，意味着目标键范围内的数据条目平均分布在每个运行中的多个数据块中。因此，每个运行中的长范围读取有两个磁盘 I/O 来源：一是查找初始位置，这需要一次磁盘 I/O；二是读取连续的数据块，直到到达结束边界。连续数据块的数量取决于用户的查询，而 Garnering 会减少需要执行的查找次数。

写放大。 回顾一下，写放大是 LSM 树中每个数据条目的平均磁盘写入次数。我们观察到，由于第 L 层的大小是第 $L-1$ 层的 T 倍，第 L 层的数据条目平均需要压缩 T 次。同样，第 L_{L-i} 层平均需要压缩 T/c^i 次。因此，第 1 层的合并操作数量主导了写放大，因而每个数据条目的总体压缩次数为 $O(T/c^{\sqrt{log_{1/c}(\frac{N}{B}\cdot T)}})$。

写放大复杂度分析。写放大对于扩展存储引擎至关重要，在本小节中，我们将说明 Garnering 为何具有可扩展的写入性能。乍一看，$O(T/c^{\sqrt{log_{1/c}(\frac{N}{B}\cdot T)}})$，其中的 $O(1/c^{\sqrt{log_{1/c}(N/B}\cdot T)})$ 项似乎很大。然而，若仔细考察这一复杂度界限，就会发现 $O(1/c^{\sqrt{log_{1/c}(\frac{N}{B}\cdot T)}})$ 实际上相对于 $\frac{N}{B}\cdot T$ 是次线性的。我们观察到，对于任意常数 x，$O((\frac{N}{B}\cdot T)^{1/x})$ 等价于 $O(1/c^{\frac{1}{x}\overline{log}_{1/c}(\frac{N}{B}\cdot T)})$。例如，$O(\sqrt{\frac{N}{B}\cdot T})$ 等价于 $O(1/c^{\frac{1}{2}log_{1/c}(\frac{N}{B}\cdot T)})$。由于这些复杂度的底数相同（$1/c$），我们只需比较指数。Garnering 的 $O(T/c^{\sqrt{log_{1/c}(\frac{N}{B}\cdot T)}})$ 相对于底数 $1/c$ 的指数为 $\sqrt{log_{1/c}(\frac{N}{B}\cdot T)}$，而对于 $O((\frac{N}{B}\cdot T)^{1/x})$，其指数相对于底数 $1/c$ 为 $\frac{1}{x}log_{1/c}(\frac{N}{B}\cdot T)$。显然，$O(T/c^{\sqrt{log_{1/c}(\frac{N}{B}\cdot T)}})$ 的复杂度是次线性的，并且对于任意常数 x，它都小于 $O((\frac{N}{B}\cdot T)^{1/x})$。因此，我们得出结论：采用我们提出的 Garnering策略的 Autumn 具有可扩展的更新性能。

Autumn 会隐式地优先压缩较低层级。随着数据条目数量增长、Autumn 中的层级不断扩展，每一层的容量也会相应增加。值得注意的是，较高层级的容量增长速度快于较低层级。这种增长速度的差异使较高层级较少触发压缩，而较低层级则更频繁地进行压缩。尽管这可能导致更高的写放大，但也能提升整体写吞吐量。由于内存预算固定，且基于 LSM 树的键值存储必须为尚未刷新的内存表保留预写日志，这类系统通常会实现速率限制器。当有过多文件尚未作为有序运行段刷新到磁盘时，速率限制器便会启动，要么限制写入速率，要么暂停所有写入，以便较低层级进行压缩，从而避免潜在的数据丢失。先前研究表明，优先执行刷新可以显著提高写吞吐量 [4]。因此，Autumn 通过隐式地在较低层级进行更多压缩，有效减少了写入停顿，并提高了写吞吐量。我们将在 YCSB 宏观实验部分进一步探讨这一优势。

3.2 性能的进一步提升

Autumn 可以通过在第0层保留多个有序运行段，进一步降低写入成本。不可变内存表刷新到第0层时，不会触发第0层内部压缩（排序合并）。此次刷新的主要目的是确保数据持久性。采用这种方法后，第一层会有多个有序运行段。事实上，LevelDB 和 RocksDB 就是这样实现的：它们在第0层保留固定数量的有序运行段，以加快从内存刷新到磁盘的速度。由于第0层有序运行段的数量固定，因此不会影响读取复杂度。

Autumn 受益于这一设计选择。主要观察结果是，Autumn 将更多压实操作分布在较低层级，并且随着数据量的增加，Autumn 的第一层会频繁进行压实，而第一层的压实开销在整体写入成本中占据主导地位。例如，假设 Garnering 采用 $c = 0.8$、$T = 2$ 和六个层级，第一层仅占总数据量的 0.18%，而第一层产生的写放大占总写入量的 30%。相比之下，传统合并策略（如 Leveling 和分级合并）在每个层级上产生相同程度的写放大，因此写入成本均匀分布在所有层级中。因此，在第0层保留多个有序运行段，可以进一步提升采用 Garnering 策略的 Autumn 的整体性能，因为刷新操作不会执行任何压实，从而降低总写放大（第一层实际上变成了分级结构）。此外，像 RocksDB 这样的 LSM 树存储引擎提供了将第0层元数据固定在块缓存中的选项。启用此功能可以加快从第0层读取文件的速度。鉴于第0层的容量相对于总存储容量而言较小，将其元数据固定在主存中只会占用块缓存极少的容量。

先前的研究 [33, 63] 提出将最初几个层级固定在非易失性内存中，以提高写入性能。这些工作将存储在非易失性内存（NVM）中的大量数据固定下来。例如，Pink [33] 将 LSM 树的前三个层级固定在 NVM 中，同时在动态随机存取存储器（DRAM）中存储数百兆字节的数据，用于保存层列表，以帮助定位元数据块。相比之下，LevelDB、RocksDB 和 Autumn 为第一个 LSM 树层级中的文件分配的空间最多为 256 MB（由 *max_bytes_for_level_base* 参数决定）。实验部分介绍了实现细节，其中我们修改了从 LevelDB 和 RocksDB 分叉的代码库（第 4 节）。因此，我们不将 Autumn 与采用专用硬件的相关工作进行比较，但我们认为，由于存储在非易失性内存中的较低层级具有较高的写带宽，这些设计也可能受益于不均匀的压缩分布。

4 实验

本节介绍实验结果，并展示 Autumn 如何针对读取进行优化，同时在空间放大和写入方面保持可扩展性。

4.1 实现细节

我们基于 RocksDB [47] 11, 2023 1 23（于 10 月分叉）和 LevelDB [30]（基于最新版本分叉）构建了 Autumn。为纳入 Garnering 随着存储数据增加而调整层容量的机制，Autumn 修改了 *MaxBytesForLevel* 函数，存储 最大_层_中_使用_的信息，并在每个文件上附加层级信息，以便根据方程 5 计算层容量。当 c 设为 1 时，Garnering 与 Leveling 的容量比相同。此外，为了观察跨层改变容量比的完整效果，我们禁用了查找压缩优化；由于其会增加关键代码路径的复杂性，RocksDB 默认将其禁用 [2]。我们还在 LevelDB 上实现 Autumn，主要是为了与 Monkey [17] 公平比较点读取；Monkey 引入了布隆过滤器优化，且基于 LevelDB 实现。默认情况下，RocksDB 和 LevelDB 都不会对第一个层级执行内部压缩；而且，从第一层到第二层的压缩触发条件不是容量，而是所存储有序段的数量。此外，还有一个速率限制器，用于限制有序运行段的数量。例如，最大_写入_缓冲区_数量 参数限制主存中的 memetable 数量，而层 *0*_ 停止 _ 写入 _ 触发器 则限制第 0 层中的有序段数量。

配置 为确保所有实验之间的比较公平，除 c 参数外，每个存储实例都将采用完全相同的配置。该参数用于区分我们提出的新型容量比配置方法与现有方法。此外，我们禁用了块缓存和压缩，并关闭了 RocksDB 中旨在尽量减少空间放大的动态分层策略。我们为 RocksDB 启用了直接 I/O，以绕过操作系统页缓存；不过，由于 LevelDB 不支持此功能，因此使用缓冲 I/O。当 c 设为 1.0 时，实验直接在截至 2023 年 10 月 11 日基于 RocksDB 源代码编译的存储引擎上进行。

[2]https://github.com/facebook/rocksdb/blob/main/HISTORY.md

4.2 微基准测试

为比较 Autumn 与 RocksDB，我们首先使用默认基准测试工具*db_bench*进行小规模微基准测试。对于*db_bench*，我们禁用了压缩，以便更好地了解和分析性能。此外，我们将存储引擎的选项设为 *OptimizeForSmallDb* 配置（RocksDB 推荐），该配置将目标文件大小设为 2MB，最大基础容量设为 10 MB。

此外，我们在微基准测试中使用了六种不同的操作：

- *FillSeq*：按顺序写入数据条目。
- *FillRandom*：按随机顺序写入数据条目。
- *ReadRandom*：执行随机点查询。
- *SeekRandom*：执行随机查找。
- *SeekRandomNext10*：执行随机的小范围查询。
- *SeekRandomNext100*：执行随机的大范围查询。

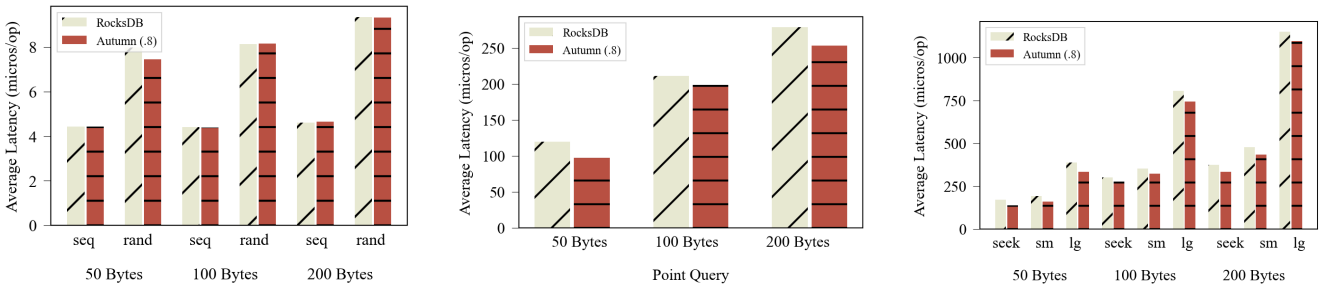
FillSeq 和 FillRandom 写入两个独立的数据库实例。读取操作在由 FillRandom 写入数据条目的数据库实例上执行。对于 *SeekRandomNextN*，迭代器首先随机查找到目标键，然后在迭代器有效时执行后续操作 N 次。

微基准测试运行环境。微基准测试实验在一台 AWS i3en.large EC2 实例上运行，该实例采用 Ubuntu 22.04 操作系统，配备 2.5 GHz Intel Xeon Platinum 8259CL 中央处理器和 1.14 TiB Amazon EC2 NVMe 实例存储。

4.2.1 不同键值大小下的不同操作。 我们为 Autumn 和 RocksDB 设置默认策略参数 $T = 2$，并使用 $c = .8$。我们将值大小分别设为 50 字节（Zippy、UP2X）、100 字节（UDB、VAR）和 200 字节（APP、ETC），因为这些值大小反映了 Facebook 生产系统中不同数据工作负载的平均值大小 [3, 8]。请回想，存储选项设置为 *OptimizeForSmallDb* 配置，并且我们禁用了数据压缩和块缓存，这与下文的宏基准测试保持一致。为了了解点查询的最差性能，我们在此微基准测试中省略布隆过滤器，并在后续章节的大规模实验中展示布隆过滤器优化（第 3 节）的优势。

写入性能。 写入性能通过两种操作进行测量：FillSeq 和 FillRandom，每种操作均包含两百万条按顺序或均匀随机排列的数据条目。我们采用每次操作的平均延迟作为指标，延迟越低，表示性能越好。如图 2 (a) 所示，Autumn 和 RocksDB 顺序写入时的延迟低于均匀随机写入时。随着值大小增加，两者的写入延迟也都会升高，因为需要写入更多字节。总体而言，在不同值大小和写入模式下，Autumn 和 RocksDB 的延迟表现相近。

点查询。 点查询性能通过 ReadRandom 操作进行测量，这些操作会读取超过一百万个随机键。我们使用每次操作的平均延迟作为指标，延迟越低越理想。此外，Autumn 和 RocksDB 都未附加布隆过滤器。如图 2(b) 所示，对于所有值大小，Autumn 的性能始终优于 RocksDB。例如，当值大小为 50 字节时，Autumn 的随机读取平均延迟为每次操作 97.526 微秒，而

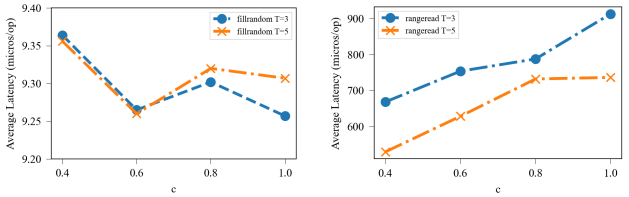


(a) 写入性能　(b) 点查询　(c) 范围查询

图 2：使用 db_bench 微基准测试，对 RocksDB 和 Autumn（$c = .8$）在六种不同操作下的评估结果。

RocksDB 的随机读取平均延迟为每次操作 119.937 微秒，性能提升约 19%。当有序段没有布隆过滤器时，点读取延迟与层级数成正比。由于 Autumn 的设计能够减少层级总数，因此其性能优于 RocksDB。

范围查询。 范围查询性能通过三种操作进行测量，即：Seek、SeekAndNext10 和 SeekAndNext100。这些操作各执行一百万次。与之前的指标类似，我们使用每次操作的平均延迟来衡量性能，纵轴数值越低越好。如图 2(c) 所示，在所有值大小和全部三种操作下，Autumn 的性能均优于 RocksDB。在这三种操作中，查找操作的性能提升最为显著。例如，当值大小为 50 字节时，Autumn 的查找延迟为每次操作 138.864 微秒，而 RocksDB 的查找延迟为每次操作 170.742 微秒，性能提升约 19%。随着下一步操作数量的增加，性能提升略有下降，这是由于需要获取更多数据。此外，随着值大小增加，性能提升也会下降。更大的键值大小意味着需要获取更多数据块，才能找到所有目标数据条目。例如，当值大小设置为 200 字节时，短范围读取性能提升 9%，长范围读取性能提升 5%。



(a) FillRandom　(b) 小范围读取

图 3：c 和 T 对小范围查询的影响。

4.2.2 写入和读取对 c 和 T 的敏感性。 Garnering策略中有两个参数：c 和 T。参数 c 控制随着存储的数据增多，小层级扩展的渐进程度，而 T 则决定最后两层之间的容量比。本节探讨在保持另一个参数不变的情况下，改变 c 或 T 对随机写入和小范围读取的影响。为进行此项分析，我们首先使用 FillRandom 写入 2 GB 键值对，每个键值对的大小均为 db_bench 的默认值 100 字节。图 3 中，x 轴为 c，取值范围为 0.4 至 1.0；y 轴表示每次操作的平均延迟。平均延迟越低，性能越好。图 3 (a) 展示了随机写入性能的两条曲线，分别对应 $T = 3$ 或 $T = 5$；同样，图 3 (b) 展示了小范围读取性能的两条曲线，分别对应 $T = 3$ 或 $T = 5$。

我们可以观察到，通常在 T 固定时，较低的 c 会带来更好的读取性能和更差的写入性能。回顾一下，Garnering中的层数为 $O(\sqrt{-log_c(\frac{N}{B} \cdot k)})$，而小范围查询性能与层数成正比。正如预期的那样，当 c 减小时，层数也会减少，因此小范围查询性能会提升。此外，当 T 增大时，层数会减少，因此小范围查询性能会提升。

4.3 宏基准测试

在本节中，我们使用广泛用于研究键值存储性能的宏基准测试套件 YCSB 基准测试 [15] (雅虎云服务基准测试)，比较 Autumn 和 RocksDB 在大规模数据下的性能。该宏基准测试进一步比较 Autumn 和 RocksDB 在真实世界工作负载流量下的性能。

<escape>**宏基准测试运行环境。** 宏基准测试实验在与上述微基准测试相同的环境中运行。</escape>

YCSBCore 工作负载 A-F。 为进一步比较 Autumn 与 RocksDB，我们使用 YCSB 的默认设置和工作负载进行实验。我们为 Autumn 和 RocksDB 设置 $T = 5$，并禁用数据压缩和块缓存，以便更好地了解性能。为更好地展示 Autumn 提升读取性能的能力，我们为 Autumn 设置 $c = 0.8$ 和 $c = 0.4$。我们预计，随着 c 减小，读取性能会提升。首先，我们写入一个 80 GB 的数据集，其中包含 1 KB 的值（默认值大小），以便加载

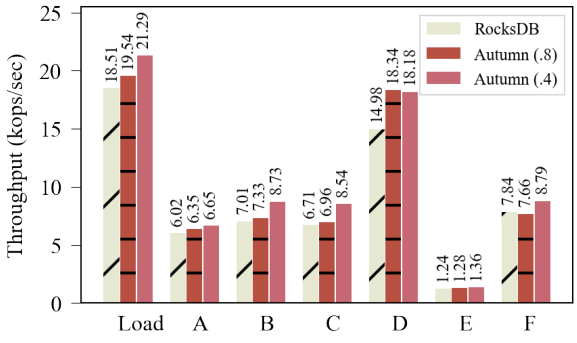


图4：YCSB 工作负载性能。该图展示了 Autumn 和 RocksDB 在各种 YCSB 核心工作负载上的性能对比。x 轴列出了不同的工作负载名称，y 轴显示吞吐量，单位为每秒千次操作（kops/sec）。每个柱状图顶部标注了实际达到的吞吐量（kops/sec）。

然后使用一百万次操作评估工作负载 A-F。不同工作负载的特征如下：

- 加载：向相应的键值存储中写入 80 GB 的数据条目，每个数据条目的大小为 1KB，包含一个 24 字节的键和十个字段，每个字段为 100 字节。• *A*：工作负载 A 以更新操作为主。50% 为点读取，50% 为更新。键遵循 Zipf [75] 分布。• *B*：工作负载 B 以读取操作为主。95% 为点读取，5% 为更新，键遵循 Zipf 分布。• *C*：工作负载 C 仅执行读取操作。100% 为遵循 Zipf 分布的点读取。

- *D*：工作负载 D 为读取最新数据。95% 点读取，5% 插入。请求按时间加权。• *E*：工作负载 E 为范围读取。95% 随机范围查询，5% 插入。每次范围读取平均读取约 100 个键。

- *F*：工作负载 F 是读-修改-写工作负载，其中包含 50% 的读取和 50% 的读-修改-写操作。

如图4所示，我们的初步观察表明，在所有默认的 YCSB 工作负载下，Autumn 的表现始终不逊于 RocksDB，甚至更胜一筹。具体而言，在加载阶段，Autumn 的吞吐量显著高于 RocksDB；将 Autumn 与 $c = 0.4$ 和 RocksDB 进行比较时，Autumn 的写吞吐量提升了 15%。查看数据库统计信息后可以发现，使用 $c = 0.4$ 的 Autumn 出现了 1265 次写入停顿，而使用 $c = 0.8$ 的 Autumn 出现了 2592 次写入停顿，RocksDB 则出现了 3403 次写入停顿。这些写入停顿主要是由内存表数量过多以及待压缩字节数过高造成的，后者指计划在近期进行压缩的字节数。这些因素会延迟写操作的处理，从而影响整体性能。通过有策略地优先处理较低层级的压缩，并逐步扩展各层的容量，使其在压缩前能够缓冲更多条目，Autumn 出人意料地提升了写吞吐量。此外，较小的数据加载量不太可能触发写入停顿，因此我们在微基准实验中没有观察到这一现象。

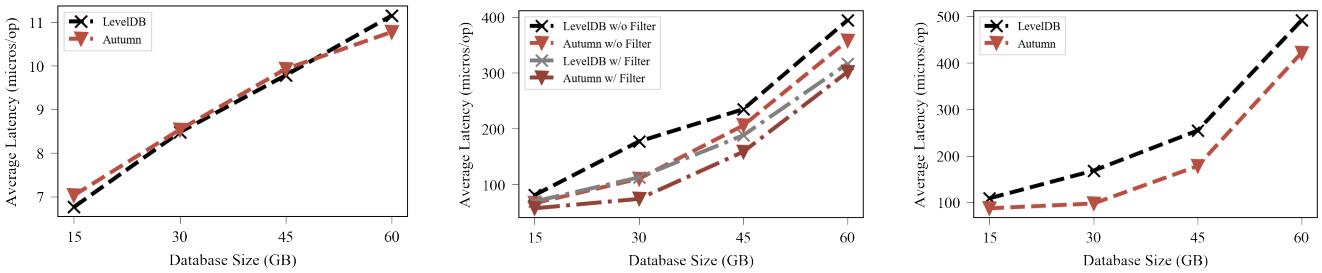
在只读工作负载（工作负载 C）中，我们确实发现 Autumn 带来的性能提升低于预期。查看层级摘要后，我们发现 RocksDB 有 5 层，Autumn (.8) 有 4 层，而 Autumn (.4) 几乎有 3 层，其中第 4 层有一个大小为 64.3 MB 的小文件。根据我们的分析，读取性能应与层级数成正比。然而，我们观察到，与 RocksDB 相比，Autumn 的 $c = .8$ 吞吐量整体仅略有提升（4%），而 Autumn 的 $c = .4$ 吞吐量整体提升了 24%。这一现象可归因于最坏情况分析和 YCSB 工作负载所采用设置的差异。在最坏情况场景中，当键值存储中不存在目标键时，读取性能最差，因为必须搜索所有层。相反，在模拟真实流量偏斜分布的 YCSB 工作负载中，键值存储中几乎包含所有键。因此，一旦找到某个键，就不需要额外的磁盘 I/O，从而使 Autumn 相对于 RocksDB 的性能提升有限。

在专注于范围读取的工作负载 E 中，我们也观察到性能有所提升，但提升幅度没有预期那么大。造成这一有限改进的主要因素有两个。首先，范围读取涉及的几乎所有键都存在于数据库中，这与最坏情况分析的假设相反，后者假设目标键范围可能未被存储。此外，在 YCSB 中，数据条目的大小设为 1KB，磁盘块大小为 4KB，扫描长度为 100，这意味着平均需要 25 个磁盘块。因此，磁盘 I/O 操作的数量主要受范围读取迭代器调用“next”操作的影响，而不是查找操作。数据条目较大的尺寸自然会提高数据检索成本，导致 Autumn 和 RocksDB 的吞吐量均低于其点读性能。

此外，在涉及读取最近写入的键的工作负载 D 中，Autumn 展现出超过 20% 的性能提升。这一改进主要得益于该工作负载中写入与读取之间的时间局部性。由于 Autumn 会策略性地提高每一层的容量，Autumn with $c = 0.4$ 在第 1 层的命中率达到 40%，而 Autumn with $c = 0.8$ 在同一层的命中率为 33%；相比之下，RocksDB 在第 1 层的命中率仅为 9%。由于点读取查询在找到键后会立即停止，因此 Autumn 的吞吐量显著提升。

我们还分析了这三种键值存储在加载阶段之后的空间放大。虽然预期的逻辑存储大小为 80 GB，但物理存储测量值略高：RocksDB 存储了 87.42 GB，使用 $c = .8$ 的 Autumn 存储了 87.49 GB，使用 $c = .4$ 的 Autumn 存储了 87.61 GB。虽然 Autumn 的空间放大略有增加，但各系统之间的整体差异仍然很小。RocksDB 是众多工业数据库中部署的生产级存储引擎，[27, 56, 73]，树立了很高的标杆；我们的设计在各种工作负载下始终能够可靠地匹配甚至超越其性能。

YCSB平均延迟和尾延迟。 部署在生产环境中的基于LSM树的键值存储应为客户提供高质量的服务。点查询和



(a) 写入性能 **（b）点查询** **(c) 小范围查询**

图 5：在数据库大小和不同操作各异的宏基准测试中，数据库_基准 的性能。性能以平均延迟衡量。

延迟（μs/op）	Avg.	95%	99%
工作负载 A 中的点读延迟			
RocksDB	157	513	640
Autumn .8	148	475	612
Aurumn .4	140	443	536
工作负载 C 中的点读延迟			
RocksDB	149	488	595
Autumn .8	143	454	579
Aurumn .4	117	332	459
工作负载 E 中的范围读取延迟			
RocksDB	846	1907	2229
Autumn .8	822	1851	2141
Autumn .4	773	1759	2006

表 3：平均延迟和尾延迟

范围查询是提供优质服务时至关重要的统计指标，而尾延迟对于延迟关键型应用可能尤为重要。更低的延迟更有利于用户体验。在表 3 中，我们列出了工作负载 A、C 和 E 中点读取或范围读取操作的平均延迟和尾延迟数值（每次操作的微秒数）。这些工作负载代表了一组不同的操作：工作负载 A 以更新为主，工作负载 C 仅执行读取，工作负载 E 则包含范围读取和一些插入。我们发现，在所有报告的分位数值以及这些不同的工作负载中，Autumn 都实现了更低的延迟数值。例如，将 RocksDB 与 Autumn 进行比较（$c = .4$），在工作负载 A、C 和 E 中，RocksDB 的 99% 尾延迟分别高出 1.2 倍、1.3 倍和 1.1 倍。这些改进能够切实提升用户体验的质量。

4.4 布隆过滤器优化

在本节中，我们将从 LevelDB 分叉而来的 Autumn 与其原始版本进行比较。最初提出布隆过滤器优化的 Monkey [17], which originally proposed 在 LevelDB 中实现了布隆过滤器优化。值得注意的是，LevelDB 不提供启用直接 I/O 的选项，因此使用缓冲 I/O。如第 3 节所示，采用布隆过滤器优化时，预计 Autumn 的性能可与 LevelDB 相当。在以下实验 $T = 2$ 中，LevelDB 和 Autumn 均使用该方案。

LevelDB 数据库_基准测试 基准测试。 与之前的基准测试步骤类似，Autumn 和 LevelDB 分别使用 *Fillrandom* 操作创建四个数据存储实例，大小分别为 15、30、45 和 60 GB（键均匀随机分布）。键长为 16 字节，值长为 100 字节。写入所有数据条目后，我们收集写入、使用和不使用布隆过滤器的点读取（将每个数据条目的位数设为 2 并应用布隆过滤器优化）以及小范围读取的平均延迟。如图 5 所示，所有子图的横轴均为数据库大小，纵轴为每次操作的平均延迟，单位为微秒。平均延迟越低，性能越好。图 5 (a) 展示了写入性能；采用缓冲 I/O 时，Autumn 的写入性能与 LevelDB 相当。例如，在 60 GB 时，Autumn 的性能略有提升，平均延迟从 LevelDB 的每次操作 11.153 微秒降至 Autumn 的 10.776 微秒。与 RocksDB 类似，当有过多内存表尚未刷新到磁盘时，LevelDB 也会启用速率限制器。速率限制器引入的写入停顿旨在确保数据持久化到磁盘，并避免机器故障时可能发生的数据丢失。随着数据量增大，Autumn 通过隐式地优先处理较低层级的压缩，逐步缓解这些写入停顿。如图 5 (b) 所示，无论是否使用布隆过滤器，或为每个有序运行段应用布隆过滤器（并进行布隆过滤器优化），Autumn 的表现都优于 LevelDB。使用布隆过滤器后，Autumn 在点读性能上的提升幅度有所下降，但性能仍然超过 LevelDB。这一现象符合预期，因为在累加假阳性率时，Autumn 收敛得更快；随着数据量和层级数增加，两者的性能最终会趋于一致。此外，我们还观察到范围读取性能稳步提升，如图 5 (c) 所示；该图比较了使用 *SeekAndNext10* 操作进行的小范围查询。当数据库大小为 30 GB 时，Autumn 的小范围

与 LevelDB 相比，查询性能提升了 42%。这些实验表明，在数据量较大时，Autumn 能够在保持较低写入延迟的同时，提供快速的点读取和范围读取。

5 相关工作

许多现有研究工作已经探索了其他提升 LSM 树性能的途径。在本节中，我们将讨论这一领域的相关工作，其中许多方法实际上与 Autumn 正交，Autumn 也可以结合这些方法进一步提升性能。

降低写放大。 研究人员主要致力于优化 LSM 树键值存储的写入成本。WiscKey [44] 通过将键存储在仅追加日志中，实现了键值分离。因此，WiscKey 降低了写放大；而且，由于现代固态硬盘 [12], 具有快速随机读取的特性，读取查询性能依然稳定（仅有轻微下降）。其他研究 [9, 62] 也采用这种设计，使用外部日志存储值。Autumn 可以采用键值分离方法，进一步提升写入性能。此外，MatrixKV [63], 、Pink [33], 和 Nov-eLSM [35],等现有研究探索了混合动态随机存取存储器-非易失性内存-固态硬盘分层存储系统的应用，以便将相当一部分数据存储在非易失性内存中，从而降低写入成本。在这类场景中，Autumn 的 Garnering 合并策略可能带来更大的收益，因为 Autumn 将较高的写入成本分散到较小的层级中。当写入成本较高的较小层级能够固定在处理能力更强的设备上时，整体写入成本会显著降低。

减少写停顿。一类研究致力于降低为客户端请求提供服务时写停顿所带来的影响。SILK [4] 通过将刷新和压缩推迟到低加载时段，并允许较小层级的压缩抢占较高层级的压缩，从而缓解写入停顿。Kvell [40] 通过最大限度地降低中央处理器的计算开销来减少写入停顿，其方法是在磁盘上存储未排序的键值对，以降低中央处理器开销。因此，它牺牲了读取性能。

降低空间放大RocksDB [24] 提出根据最后一层的大小来限制较小层级的容量，从而通过实证方法降低 LSM 树中的存储空间放大。由于最后一层的近似大小存储在元数据中，LevelDB 和 Autumn 经过一些修改后都可以采用这一优化。

面向 LSM 树的数据摘要。 数据概要能够为各类问题提供近似答案，并具有理论证明的误差界 [16, 36, 42, 43, 48, 69, 71, 72]。例如，可以使用布隆过滤器 [7]解决成员查询。如今，过滤器已广泛用于基于 LSM 树的键值存储，以提升点读取性能。为在固定内存预算下提高准确率，研究人员开发了新的点过滤器，例如布谷鸟过滤器 [26] 和商过滤器 [5], 。为提升范围读取性能，研究人员设计了范围过滤器，最初由 ARF [1]提出，之后又有 SuRF [66] 和 Rosetta [46], , 用于在存储介质或文件不包含目标键范围时跳过磁盘访问。然而，与点过滤器相比，范围过滤器需要更多空间预算才能支持任意键范围。在固定内存预算下，范围过滤器会牺牲点过滤器查询准确率。

硬件加速与工作负载优化。 键值存储可以利用专用硬件的新进展来提升性能，例如现场可编程门阵列 [32, 34], 智能网卡 [41], 图形处理器 [2], 持久内存 [60]和异构存储 [64]。此外，多核 CPU [65] 和无锁同步 [29] 技术也可以提升 LSM 树的性能。此外，当读取分布偏斜时，张等人 [67] 提出通过将热点数据条目移回较小的层来提升读取性能。这些硬件加速和依赖工作负载的优化方法与 Autumn 正交，因为 Garnering 合并策略可以构建在这些不同的底层硬件架构之上。

未来，我们还希望研究 Garnering 如何提升 LSM 树在多层存储（例如动态随机存取存储器、非易失性内存、固态硬盘）中的性能，或如何帮助降低流式数据库 [3] 的成本（例如，数据存储在本地驱动器、分布式文件系统和 Amazon Simple Storage 中）。由于 Autumn 会自动在较低层级引入更多压缩操作，而数据量较少的较低层级通常位于速度更快的设备上，我们认为 Autumn 也可能提升此类系统的性能。

6 结论

本文介绍了 Autumn，这是一种新型键值存储，采用创新的 Garnering 压实策略，重新定义了 LSM 树中的数据组织方式，从而改善点读取和范围读取的最坏情况复杂度。此外，延迟压实的引入，以及 Garnering 策略优先对较小层级进行压实这一特点，显著提升了 Autumn 的读写效率。我们使用 *db_bench* 和 *YCSB* 基准测试，对 Autumn 与谷歌的 LevelDB 和 Meta 的 RocksDB 进行了全面评估。结果表明，Autumn 不仅优化了读取操作，还保持了稳健的写入性能。因此，Garnering 压实策略使 Autumn 能够显著提升键值存储性能，使其非常适用于在线事务处理和混合事务与分析处理工作负载。

参考文献

[1] Karolina Alexiou、Donald Kossmann 和 Per-Åke Larson。2013年。面向冷数据的自适应范围过滤器：避免访问西伯利亚。*VLDB* 基金会会刊 6，14（2013），1714–1725。[2] Saman Ashkiani、Shengren Li、Martin Farach-Colton、Nina Amenta 和 John D Owens。2018年。GPU LSM：一种面向 GPU 的动态字典数据结构。载于*2018 IEEE* 国际并行与分布式处理研讨会（*IPDPS*）。IEEE，430–440。[3] Berk Atikoglu、Yuehai Xu、Eitan Frachtenberg、Song Jiang 和 Mike Paleczny。2012年。大规模键值存储的工作负载分析。载于第*12*届 *ACM SIGMETRICS/PERFORMANCE* 联合国际会议：计算机系统测量与建模。53–64。[4] Oana Balmau、Florin Dinu、Willy Zwaenepoel、Karan Gupta、Ravishankar Chand- hiramoorthi 和 Diego Didona。2019年。SILK：防止日志结构合并键值存储中的延迟尖峰。载于 *USENIX* 年度技术会议。753–766。[5] Michael A Bender、Martin Farach-Colton、Rob Johnson、Bradley C Kuszmaul、Dzejla Medjedovic、Pablo Montes、Pradeep Shetty、Richard P Spillane 和 Erez Zadok。2011年。别让缓存失效：如何在闪存上缓存哈希。载于第*3*届存储与文件系统热点主题研讨会（*HotStorage 11*）。[6] Dimitri P Bertsekas。2014年。约束优化与拉格朗日乘子法。Academic Press。[7] Burton H Bloom。1970年。允许错误的哈希编码中的时空权衡。*Commun. ACM* 13，7（1970），422–426。

[3] https://www.risingwave.dev/docs/current/data-persistence/

[8] Zhichao Cao 和 Siying Dong。2020年。Facebook 上 RocksDB 键值工作负载的特征分析、建模与基准测试。载于第18届 USENIX 文件与存储技术会议（FAST'20）。[9] Helen HW Chan、Yongkun Li、Patrick PC Lee 和 Yinlong Xu。2018年。Hashkv：通过哈希实现 KV 存储中的高效更新。载于2018年 USENIX 年度技术会议（USENIX ATC 18）。1007–1019。[10] Fay Chang、Jeffrey Dean、Sanjay Ghemawat、Wilson C Hsieh、Deborah A Wallach、Mike Burrows、Tushar Chandra、Andrew Fikes 和 Robert E Gruber。2008年。Bigtable：一种用于结构化数据的分布式存储系统。ACM 计算机系统汇刊（TOCS）26，2（2008），1–26。[11] Subarna Chatterjee、Meena Jagadeesan、Wilson Qin 和 Stratos Idreos。2021年。Cosine：一种云成本优化的自设计键值存储引擎。VLDB 基金会会刊15，1（2021），112–126。[12] Feng Chen、Rubao Lee 和 Xiaodong Zhang。2011年。充分利用基于闪存的固态硬盘内部并行性，在高速数据处理中发挥关键作用。载于2011年 IEEE 第17届高性能计算机体系结构国际研讨会。IEEE，266–277。[13] Guoqiang Jerry Chen、Janet L Wiener、Shridhar Iyer、Anshul Jaiswal、Ran Lei、Nikhil Simha、Wei Wang、Kevin Wilfong、Tim Williamson 和 Serhat Yilmaz。2016年。Facebook 的实时数据处理。载于2016年数据管理国际会议论文集。1087–1098。[14] Jianjun Chen、Yonghua Ding、Ye Liu、Fangshi Li、Li Zhang、Mingyi Zhang、Kui Wei、Lixun Cao、Dan Zou、Yang Liu 等。2022年。ByteHTAP：字节跳动的 HTAP 系统，具备高数据新鲜度和强数据一致性。VLDB 基金会会刊15，12（2022），3411–3424。[15] Brian F Cooper、Adam Silberstein、Erwin Tam、Raghu Ramakrishnan 和 Russell Sears。2010年。使用 YCSB 对云服务系统进行基准测试。载于第1届 ACM 云计算研讨会论文集。143–154。[16] Graham Cormode 和 Shan Muthukrishnan。2005年。一种改进的数据流摘要：Count-Min Sketch 及其应用。《算法杂志》55，1（2005），58–75。[17] Niv Dayan、Manos Athanassoulis 和 Stratos Idreos。2017年。Monkey：最优可导航键值存储。载于2017年 ACM 数据管理国际会议论文集。79–94。[18] Niv Dayan 和 Stratos Idreos。2018年。Dostoevsky：通过自适应移除冗余合并，实现基于 LSM 树的键值存储更优的时空权衡。载于2018年数据管理国际会议论文集。505–520。[19] Niv Dayan 和 Stratos Idreos。2019年。日志结构合并灌木与奇异连续体。载于2019年数据管理国际会议论文集。449–466。[20] Niv Dayan 和 Moshe Twitto。2021年。Chucky：一种用于 LSM 树的精简布谷鸟过滤器。载于2021年数据管理国际会议论文集。365–378。[21] Niv Dayan、Moshe Twitto、Yuval Rochman、Uri Beitler、Itai Ben Zion、Edward Bortnikov、Shmuel Dashevsky、Ofer Frishman、Evgeni Ginzburg、Igal Maly 等。2021年。摩尔定律的终结与数据处理器的崛起。VLDB 基金会会刊14，12（2021），2932–2944。[22] Niv Dayan、Tamar Weiss、Shmuel Dashevsky、Michael Pan、Edward Bortnikov 和 Moshe Twitto。2022年。Spooky：正确地细分 LSM 树压实。VLDB 基金会会刊15，11（2022），3071–3084。[23] Giuseppe DeCandia、Deniz Hastorun、Madan Jampani、Gunavardhan Kakulapati、Avinash Lakshman、Alex Pilchin、Swaminathan Sivasubramanian、Peter Vosshall 和 Werner Vogels。2007年。Dynamo：亚马逊的高可用键值存储。ACM SIGOPS 操作系统评论41，6（2007），205–220。[24] Siying Dong、Mark Callaghan、Leonidas Galanis、Dhruba Borthakur、Tony Savor 和 Michael Strum。2017年。在 RocksDB 中优化空间放大。载于 CIDR，第3卷。3。[25] Siying Dong、Andrew Kryczka、Yanqin Jin 和 Michael Stumm。2021年。RocksDB：服务于大规模应用的键值存储开发优先事项的演变。ACM 存储汇刊（TOS）17，4（2021），1–32。[26] Bin Fan、Dave G Andersen、Michael Kaminsky 和 Michael D Mitzenmacher。2014年。布谷鸟过滤器：实际表现优于 Bloom 过滤器。载于第10届 ACM 新兴网络实验与技术国际会议论文集。75–88。[27] Tao Feng。2015年。Apache Samza 基准测试：单节点每秒处理120万条消息。网址：https://engineering. linkedin. com/performance/benchmarking-apache-samza-12-million-messagessecond-single-node（2015）。[28] Lars George。2011年。《HBase 权威指南：随机访问行星级数据》。"O'Reilly Media, Inc."。[29] Guy Golan-Gueta、Edward Bortnikov、Eshcar Hillel 和 Idit Keidar。2015年。扩展并发日志结构化数据存储。载于第十届欧洲计算机系统会议论文集。1–14。[30] 谷歌。2011年。LevelDB。https://github.com/google/leveldb[31] Dongxu Huang、Qi Liu、Qiu Cui、Zhuhe Fang、Xiaoyu Ma、Fei Xu、Li Shen、Liu Tang、Yuxing Zhou、Menglong Huang 等。2020年。TiDB：一种基于 Raft 的 HTAP 数据库。VLDB 基金会会刊13，12（2020），3072–3084。[32] Gui Huang、Xuntao Cheng、Jianying Wang、Yujie Wang、Dengcheng He、Tieying Zhang、Feifei Li、Sheng Wang、Wei Cao 和 Qiang Li。2019年。X-Engine：一种面向大规模电子商务事务处理的优化存储引擎。在《2019 年数据管理国际会议论文集》中。651–665。[33] Junsu Im、Jinwook Bae、Chanwoo Chung、Arvind Arvind 和 Sungjin Lee。2020。PinK：具有受限尾延迟的高速存储内键值存储。在 USENIX 年度技术会议上。173–187。[34] Wenqi Jiang、Dario Korolija 和 Gustavo Alonso。2023。现代架构上的现场可编程门阵列数据处理。arXiv 预印本 arXiv:2304.03044（2023）。[35] Sudarsun Kannan、Nitish Bhat、Ada Gavrilovska、Andrea Arpaci-Dusseau 和 Remzi Arpaci-Dusseau。2018。使用 NoveLSM 为非易失性内存重新设计日志结构合并树。在 2018 年 {USENIX} 年度技术会议（{USENIX}{ATC} 18）上。993–1005。[36] Zohar Karnin、Kevin Lang 和 Edo Liberty。2016。数据流中的最优分位数近似。在 2016 年 IEEE 第 57 届计算机科学基础年度研讨会（计算机科学基础研讨会）上。IEEE，71–78。[37] Eric R Knorr、Baptiste Lemaire、Andrew Lim、Siqiang Luo、Huanchen Zhang、Stratos Idreos 和 Michael Mitzenmacher。2022。Proteus：一种自设计范围过滤器。在《2022 年数据管理国际会议论文集》中。1670–1684。[38] Avinash Lakshman 和 Prashant Malik。2010。Cassandra：一种去中心化的结构化存储系统。《ACM SIGOPS 操作系统评论》44，2（2010），35–40。[39] Changman Lee、Dongho Sim、Joo Young Hwang 和 Sangyeun Cho。2015。F2fs：一种面向闪存存储的新型文件系统。在 FAST，第 15 卷。273–286。[40] Baptiste Lepers、Oana Balmau、Karan Gupta 和 Willy Zwaenepoel。2019。Kvell：快速持久化键值存储的设计与实现。在第 27 届 ACM 操作系统原理研讨会论文集上。447–461。[41] Bojie Li、Zhenyuan Ruan、Wencong Xiao、Yuanwei Lu、Yongqiang Xiong、Andrew Putnam、Enhong Chen 和 Lintao Zhang。2017。KV-Direct：具有可编程网络接口卡的高性能内存键值存储。在第 26 届操作系统原理研讨会论文集上。137–152。[42] Zirui Liu、Chaozhe Kong、Kaicheng Yang、Tong Yang、Ruijie Miao、Qizhi Chen、Yikai Zhao、Yaofeng Tu 和 Bin Cui。2023。HyperCalm Sketch：单遍挖掘数据流中的周期性批次。在 2023 年 IEEE 第 39 届国际数据工程会议（国际数据工程会议）上。IEEE。[43] Zirui Liu、Yixin Zhang、Yifan Zhu、Ruwen Zhang、Tong Yang、Kun Xie、Sha Wang、Tao Li 和 Bin Cui。 [不详。]TreeSensing：灵活压缩草图。([不详。])。[44] Lanyue Lu、Thanumalayan Sankaranarayana Pillai、Hariharan Gopalakrishnan、Andrea C Arpaci-Dusseau 和 Remzi H Arpaci-Dusseau。2017。WiscKey：在面向固态硬盘的存储中将键与值分离。《ACM 存储汇刊》13，1（2017），1–28。[45] Chen Luo 和 Michael J Carey。2020。基于日志结构合并树的存储技术：综述。《VLDB 期刊》29，1（2020），393–418。[46] Siqiang Luo、Subarna Chatterjee、Rafael Ketsetsidis、Niv Dayan、Wilson Qin 和 Stratos Idreos。2020。Rosetta：一种适用于键值存储的稳健时空优化范围过滤器。在《2020 年 ACM SIGMOD 数据管理国际会议论文集》中。2071–2086。[47] Yoshinori Matsunobu、Siying Dong 和 Herman Lee。2020。MyRocks：为 Facebook 社交图提供服务的 LSM 树数据库存储引擎。《VLDB 基金会会刊》13，12（2020），3217–3230。[48] Ahmed Metwally、Divyakant Agrawal 和 Amr El Abbadi。2005。高效计算数据流中的高频元素和前 k 个元素。在《数据库理论国际会议 2005：第 10 届国际会议，爱丁堡，英国，2005 年 1 月 5 日至 7 日，论文集 10》中。施普林格，398–412。[49] Changwoo Min、Kangnyeon Kim、Hyunjin Cho、Sang-Won Lee 和 Young Ik Eom。2012。SFS：随机写入对固态硬盘有害。在 FAST，第 12 卷。1–16。[50] Patrick O'Neil、Edward Cheng、Dieter Gawlick 和 Elizabeth O'Neil。1996。日志结构合并树（LSM 树）。《信息学学报》33（1996），351–385。[51] PingCAP。2016。TiKV。https://github.com/tikv/tikv [52] Mendel Rosenblum 和 John K Ousterhout。1992。日志结构文件系统的设计与实现。《ACM 计算机系统汇刊》10，1（1992），26–52。[53] Dennis G Severance 和 Guy M Lohman。1976。差分文件：其在大型数据库维护中的应用。《ACM 数据库系统汇刊》1，3（1976），256–267。[54] Arun Sharma。2016。Dragon：分布式图查询引擎。Facebook 工程博客（2016）。[55] Michael Stonebraker。1987。Postgres 存储系统的设计。技术报告。加州大学伯克利分校电子研究实验室。[56] Rebecca Taft、Irfan Sharif、Andrei Matei、Nathan VanBenschoten、Jordan Lewis、Tobias Grieger、Kai Niemi、Andy Woods、Anne Birzin、Raphael Poss 等。2020。CockroachDB：弹性地理分布式 SQL 数据库。在《2020 年 ACM SIGMOD 数据管理国际会议论文集》中。1493–1509。[57] Sasu Tarkoma、Christian Esteve Rothenberg 和 Eemil Lagerspetz。2011。分布式系统中布隆过滤器的理论与实践。《IEEE 通信综述与教程》14，1（2011），131–155。[58] Dejun Teng、Lei Guo、Rubao Lee、Feng Chen、Yanfeng Zhang、Siyuan Ma 和 Xiaodong Zhang。2018。一种低成本磁盘解决方案，使 LSM 树能够实现 乐彩

混合读写工作负载下的高性能。*ACM 存储汇刊（TOS）*14, 2 (2018), 1–26。[59] Thomas N Theis 和 H-S Philip Wong。2017。摩尔定律的终结：信息技术的新开端。计算机科学与工程19, 2 (2017), 41–50。[60] Lukas Vogel、Alexander van Renen、Satoshi Imamura、Jana Giceva、Thomas Neumann 和 Alfons Kemper。2022。Plush：一种写入优化的持久化日志结构哈希表。*VLDB 基金会会刊*15, 11 (2022), 2895– 2907。[61] Fenggang Wu、Ming-Hong Yang、Baoquan Zhang 和 David HC Du。2020。AC-key：面向基于 LSM 的键值存储的自适应缓存。载于*2020 年 USENIX 年度技术会议论文集*。603–615。[62] Giorgos Xanthakis、Giorgos Saloustros、Nikos Batsaras、Anastasios Papagiannis 和 Angelos Bilas。2021。Parallax：基于 LSM 的键值存储中的混合键值放置。载于*ACM 云计算研讨会论文集*。305–318。[63] Ting Yao、Yiwen Zhang、Jiguang Wan、Qiu Cui、Liu Tang、Hong Jiang、Chang-sheng Xie 和 Xubin He。2020。MatrixKV：通过在非易失性内存中使用矩阵容器，减少基于 LSM 树的键值存储中的写停顿和写放大。载于*2020 年 USENIX 年度技术会议论文集*。17–31。[64] Hobin Yoon、Juncheng Yang、Sveinn Fannar Kristjansson、Steinn E Sigurdarson、Ymir Vigfusson 和 Ada Gavrilovska。2018。Mutant：平衡 LSM 树数据存储中的存储成本与延迟。载于*ACM 云计算研讨会论文集*。162–173。[65] Dongping Zhang、Nuwan Jayasena、Alexander Lyashevsky、Joseph L Greathouse、Lifan Xu 和 Michael Ignatowski。2014。TOP-PIM：面向吞吐量的内存可编程处理。载于第 *23* 届高性能并行与分布式计算国际研讨会论文集。85–98。[66] Huanchen Zhang、Hyeontaek Lim、Viktor Leis、David G Andersen、Michael Kaminsky、Kimberly Keeton 和 Andrew Pavlo。2018。Surf：使用快速紧凑字典树实现实用的范围查询过滤。载于*2018 年国际数据管理会议*. 323–336. [67] Xin Zhang、Qizhong Mao、Ahmed Eldawy、Vagelis Hristidis 和 Yihan Sun。2022年。双向日志结构合并树。载于第*34*届科学与统计数据库管理国际会议论文集。1–4. [68] Yueming Zhang、Yongkun Li、Fan Guo、Cheng Li 和 YinlongXu。2018年。ElasticBF：面向基于 LSM 树的键值存储的细粒度弹性布隆过滤器，助力高效读取。载于*HotStorage*。[69] Fuheng Zhao、Divyakant Agrawal、Amr El Abbadi 和 Ahmed Metwally。2021年。SpaceSaving±：有界删除模型中频率估计和高频项估计的最优算法。*arXiv 预印本 arXiv:2112.03462* (2021).[70] Fuheng Zhao 、 Punnal Ismail Khan 、Divyakant Agrawal、Amr El Abbadi、Arpit Gupta 和 Zaoxing Liu。2023年。Panakos：追踪多维数据流的尾部。*VLDB 基金会会刊*16, 6 (2023), 1291–1304. [71] Fuheng Zhao、Sujaya Maiyya、Ryan Wiener、Divyakant Agrawal 和 Amr El Abbadi。2021年。动态数据集上的 Kll± 近似分位数草图。*VLDB 基金会会刊*14, 7 (2021), 1215–1227.[72] Fuheng Zhao、Dan Qiao、Rachel Redberg、Divyakant Agrawal、Amr El Abbadi 和 Yu-Xiang Wang。2022年。差分隐私线性草图：高效实现与应用。神经信息处理系统进展35 (2022), 12691–12704.[73] Jingyu Zhou、Meng Xu、Alexander Shraer、Bala Namasivayam、Alex Miller、Evan Tschannen、Steve Atherton、Andrew J Beamon、Rusty Sears、John Leach 等。2021年。Foundationdb：一种分布式、解耦式事务键值存储。载于*2021*年数据管理国际会议论文集。2653–2666。[74] Zichen Zhu、Ju Hyoung Mun、Aneesh Raman 和 Manos Athanassoulis。2021年。降低现代存储设备上 LSM 树的布隆过滤器中央处理器开销。载于第*17*届新型硬件数据管理国际研讨会（*DaMoN 2021*）论文集。1–10. [75] George Kingsley Zipf。2016年。人类行为与最省力原则：人类生态学导论。Ravenio Books。