摘要

现代键值存储通常依赖于存储中的LSM树（SSD）来处理写操作，并依赖于内存中的Bloom过滤器（DRAM）来优化读操作。随着SSD技术的不断进步缩小了存储设备和内存设备之间的性能差距，Bloom过滤器现在正成为一个性能瓶颈。

我们提出了Chucky，一种新的设计，用一个Chucky过滤器代替多个Bloom过滤器，该过滤器将每个数据条目映射到LSM树中其位置的辅助地址。我们表明，虽然这种设计需要的内存访问比使用Bloom过滤器更少，但其即时误报率更高。原因是辅助地址占用了一些位，否则这些位将被用作布谷鸟过滤器指纹的一部分。为了解决这个问题，我们利用信息论中的技术对辅助地址进行简洁的编码，这样指纹就可以保持较大。因此，Chucky实现了两全其美：适中的访问成本和较低的误报率。

ACM参考格式：

Niv Dayan, Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSMTree . In Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457273

1. 介绍

现代KV存储依赖于LSM树在存储中持久化数据。LSM树在内存中缓冲新数据，在缓冲区填满时以排序运行的方式将缓冲区刷新到存储器中，并在对数Level上压缩运行[79]。为了优化应用程序点查询，每个运行都有一个内存中的Bloom筛选器[11]，用于排除不包含目标条目的运行。此类设计用于OLTP[45]、HTAP[71]、社交图[74]、FTL设计[12,27]、数据系列[61-63]、区块链[34]、流处理[21]等。

问题1：更改存储介质. LSM树最初是为HDD设计的，HDD比DRAM内存芯片慢5-6个数量级。然而，SSD的出现将存储和内存之间的性能差距缩小到了2-3个数量级[10,30,37]。今天，内存I/O需要 ≈ 100纳秒，而Intel Optane SSD上的读取I/O需要≈ 10微秒。因此，相对于存储访问，内存访问不再是可忽略的。对于LSM树，尤其是具有数十到数百次运行的现代设计[59、76、85、86、98]，查询 ≈ 每次运行100纳秒可能接近甚至超过从存储器中获取目标条目的SSD I/O的延迟。

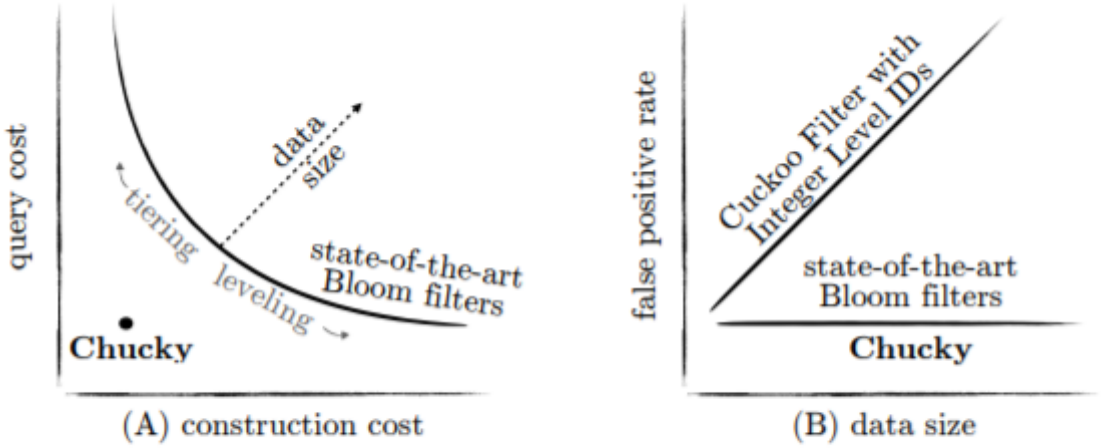


Figure 1: Existing filters for LSM-tree can not scale query cost, construction cost and the false positive rate all at once.

问题2：工作负载倾斜。 KV存储通常维护内存块缓存，以存储频繁访问的数据块，从而优化倾斜，这通常很常见[41]。查询块缓存仍然需要首先遍历所有Bloom筛选器，以标识包含目标项的运行。由于在这种情况下没有存储I/O，因此遍历Bloom过滤器的成本变得更加重要。

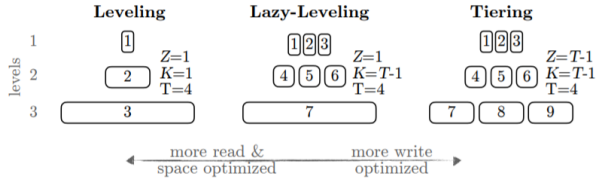
问题3：读与写争用。 为了降低Bloom过滤器的查询成本，可以调整LSM树以更频繁地合并，从而减少运行次数，从而减少访问Bloom过滤器的次数。然而，这增加了Bloom过滤器的建造成本。Bloom过滤器是不可变的，必须在每次压缩过程中从头开始重建。因此，更频繁地压缩需要更频繁地重建Bloom过滤器。最近有报道称，Bloom过滤器的构造可能占写入路径性能开销的70%以上[58]。因此，Bloom过滤器的访问和构建成本在LSM树上下文中相互竞争，如图1第（A）部分中的概念性描述

问题4：数据大小的可伸缩性。 随着数据大小的增长，需要在更多的LSM树Level上查询和构造更多的Bloom过滤器。因此，查询和构造Bloom过滤器的开销也会增加。如图1第（A）部分所示，这会导致读写折衷曲线向外移动，让应用程序在两者之间做出更糟糕的折衷选择。随着数据在许多现代应用程序中呈指数级增长，其结果是性能迅速恶化。

研究问题。 我们是否可以设计一个Bloom过滤器的替代品，使LSM树在（1）存储介质、（2）工作负载偏差、（3）LSM树调整和（4）数据大小方面具有更强健和可扩展的性能?

Cuckoo Filter: 承诺。 在过去十年中，出现了一个新的数据结构家族，作为Bloom过滤器的替代品。这些结构通过在紧凑的哈希表中为每个条目的密钥存储一个称为指纹的小哈希摘要来运行。它们包括Quotient filter[9,81]，Cuckoo Filter[39]和其他[15,44,80,96]。

在本文中，我们将LSM树的多个Bloom过滤器替换为布谷鸟过滤器变体，该变体将每个数据条目映射到指纹和辅助Level ID（LID）。LID是一小串位，用于标识给定条目在LSM树中的位置。它指向在点读取期间需要在布谷鸟过滤器内搜索匹配指纹的Level（或Level的一部分）。其承诺是将过滤内存I/O减少到一个较小且恒定的数量，这是对Bloom过滤器的改进。



Term	Definition
N	largest level size (entries)
P	buffer size (entries)
T	LSM-tree size ratio
L	number of LSM-tree levels
Z	number of sub-levels at largest level
K	number of sub-levels at each of Levels 1 to L-1
Ai	number of sub-levels at Level i
A	number of sub-levels in the whole LSM-tree
M	filtering memory budget (bits/entry)

Figure 2: LSM-tree variants and terms used to describe them throughout the paper. Each rectangle represents a sub-level.

调整False Positive Rate(假阳性). 尽管这种方法很有前途，但我们也发现了一个挑战：保持低且稳定的假阳性率（FPR）。FPR与指纹大小成反比。 在有限的内存预算下，LIDs占据了原本用作指纹部分的位。更糟糕的是，随着数据的增长，LSM树中的Level数也在增长。这需要增加LID的大小（以位为单位），以便仍然能够识别每个Level（或其中的一部分）。结果，随着数据的增长，LIDS会从指纹中"窃取"更多的比特。这会导致FPR随时间增加。结果是更多的存储I/O，这会损害性能。我们在图1第（B）部分中使用带整数级ID的曲线标记的布谷鸟过滤器从概念上说明了这一挑战。

洞察：LevelID可压缩性。 我们在这篇论文中的核心观点是，幸运的是，LIDs是非常可压缩的。原因是LSM树的指数增长；大多数条目位于更高的Level。因此，布谷鸟过滤器中的LIDs分布严重倾斜：大多数LIDs对应于最大Level，而指数级更少的LIDs对应于较小Level。因此，我们可以使用比较小Level的LID更少的比特对较大Level的LID进行编码，以最小化平均LID大小。可以将保存的位分配给指纹，使其保持较大。

Chucky. 我们介绍了Chucky:Huffman编码键值存储，这是一种新的设计，可以同时扩展内存I/O、内存占用和误报率。它通过用带有简洁编码（即压缩）LIDs的布谷鸟过滤器取代布卢姆过滤器来实现这一点。我们详细探索了LIDs 压缩的设计空间，并确定并解决了由此带来的挑战：（1）如何在布谷鸟过滤器的桶中对齐指纹和压缩LIDs，以及（2）如何高效地对LIDs进行编码/解码。我们使用保存的位来保持指纹较大，从而随着数据的增长保持较低的FPR和稳定。

贡献. 我们的贡献如下。

1. 我们发现，随着SSD的发展和速度的加快，LSM树的Bloom过滤器正在成为内存I/O瓶颈。
2. 我们用布谷鸟过滤器替换Bloom过滤器，布谷鸟过滤器将数据条目映射到它们在LSM树中的位置。
3. 我们证明了Level IDs是可压缩的。我们研究如何使用哈夫曼编码有效地对它们进行编码/解码。
4. 我们将展示如何在布谷鸟过滤器桶中对齐压缩Level IDs和指纹，以确保良好的空间利用率。
5. 我们通过实验证明，Chucky可以同时扩展内存带宽、内存占用和FPR。

2. 最新技术：LSM&BLOOM

LSM树由多级指数增长的容量组成。level 0是内存缓冲区（也称为memtable），通常作为SkipList或HashTable实现。所有其他level 都在磁盘中。应用程序将键值条目插入memtable。当memtable填满时，它会被刷新到存储器中。

合并策略. LSM树的合并策略指定在存储中合并哪些数据以及何时合并。虽然合并策略可以以不同的方式形式化，但我们采用了陀思妥耶夫斯基框架[28]，因为它概括了几个著名的策略。level L的数量是$[\log_T (N/p)]$，其中N是最大Level的条目数，p是缓冲区大小，T是任意两个相邻level 之间的容量比（T ≥ 2). 每个Level由一个或多个sub-Levels组成，其中sub-Level是一个已排序数据块的占位符。在最大level 上，有Z个sub-Level（1≤ Z<T）。在每个较小的Level上，都有K个sub-Level（1≤ K<T）。图2显示了如何为参数Z和K的不同配置对sub-Level进行编号。level i的容量为$P \cdot T^i$条目，并且在Level i的sub-Level之间平均分配。方程式1表示A_i为i层级的子层级数量，表示A_i为子层级总数。level L和sub-level A的数量都随着数据大小的增加而增加。

$$A_i = \begin{cases} K & \text{for } 1 \le i < L \\ Z & \text{else} \end{cases} \qquad A = \sum_{i=1}^{L} A_i = (L-1) \cdot K + Z \quad (1)$$

每个sub-level 都有零次或一次run。run由按键排序的键值项组成。不同的梯段可能具有重叠的关键帧范围。每个run可以进一步划分为更小的文件，这些文件具有非重叠的键范围，称为排序字符串表（SST_s），尽管我们将在本文中使用run作为数据单位。

当i级的所有sub-level 达到容量时，它们的组成runs将合并到i+1级低于容量的最高sub-level 。如果此目标sub-level 上已经有run，则该run将包含在合并中。因此，在一级run的j^{th}最年轻的run始终在sub-level 编号（i − 1) · K + j。

参数K和Z可以共同调整以假设不同的权衡。图2显示了如何对它们进行优化，以假设三种合并策略：（1）level 调整，最适合范围读取，（2）分层调整，最适合写入，以及（3）延迟level 调整，最适合点读取。虽然图2将其修正为4，但可以调整大小比T来微调这些权衡。当大小比T设置为2（其可能的最低设置）时，三个合并策略的行为相同。随着我们每项政策的增加，他们的行为也会有所不同。基于对可导航系统的愿景，该系统可以在广阔的设计空间中学习和适应，以优化不同的工作负载[5、6、47、52–57]，我们设计了Chucky，以跨越整个广阔的设计空间。

更新和删除。更新和删除是通过在缓冲区中插入带有更新值的键值条目来执行的（对于删除，该值是一个墓碑）。每当合并包含具有相同密钥的条目的run时，旧版本将被丢弃，只保留最新版本。要始终查找条目的最新版本，查询将从最年轻到最老（从较小level 到较大level ，以及从较低level 到较高level ）遍历run。当它找到具有匹配键的第一个条目时终止。如果该条目的值是墓碑，则查询返回负数结果。

	Leveling	Lazy-Leveling	Tiering
application query	$O(L)$	$O(L \cdot T)$	$O(L \cdot T)$
application update	$O(L \cdot T)$	$O(L + T)$	$O(L)$

Table 1: Blocked Bloom filters' memory I/O complexities.

Fence Pointers. 对于每次run，内存中都有Fence Pointers.，这些指针在每个数据块上都包含最小/最大键。它们允许查询以二进制方式搜索包含给定密钥的相关数据块 ≈ log(N)内存I/O，以便使用一个存储I/O即可廉价检索此块。

Bloom Filters. 对于LSM树中的每次run，都有一个内存Bloom filter（BF），这是一种节省空间的概率数据结构，用于测试密钥是否是集合的成员[11]。BF是具有h哈希函数的位数组。使用这些散列函数将每个键映射到h个随机位，将它们从0设置为1或将它们设置为1。检查密钥是否存在需要检查其h位。如果其中任何一个被设置为0，我们就有一个负数。如果all设置为1，则为真或假阳性。假阳性概率（FPP）是$2^{-M \cdot ln(2)}$，其中M是每个条目的位数。随着M的增加，散列冲突的概率降低，因此FPP下降。BF不支持范围读取或删除。缺少delete支持意味着，由于压缩，每一次新的运行都必须从头开始构建一个新的BF。

BF需要h内存I/O来插入或查询现有密钥。对于对不存在的键的查询，它平均需要两个内存I/O≈ 50%的位设置为零，因此在产生零之前检查的预期位数为2。

Blocked Bloom Filters. 为了优化内存I/O，已提出将Blocked Bloom Filter作为连续BF的数组，每个BF的大小与缓存线相同[66,84]。通过首先将密钥散列到其中一个组成BF，然后将密钥插入其中来插入密钥。对于任何插入或查询，这只需要一个内存I/O。折衷是FPP略有增加。RocksDB采用了Blocked BFs。在本文中，我们使用Standard和Blocked BFs作为基线，我们更关注Blocked BFs，因为它们是更为激烈的竞争。

Memory I/O Analysis. 对于Blocked BFs，查询的总成本为(L− 1)·K+Z内存I/O，LSM树的每个子级一个。另一方面，插入/更新/删除的成本与LSM树的写入放大相同: $\approx \frac{T-1}{K+1} \cdot (L-1) + \frac{T-1}{Z+1}$ 和Dostoevsky. 原因是条目参与的每一次压缩都会导致一次BF插入，这需要一次内存I/O。表1总结了每个合并策略的这些成本。这表明，BFs上的查询开销可能很大，尤其是在分层和延迟均衡的情况下。此外，BF的查询和构建成本都随着level L的数量以及数据大小的增加而增加。最后，BFs的查询与构建成本之间存在反向关系：我们将LSM树的合并策略设置得越贪婪（即，通过微调参数T、K和Z），当BFs越少，查询成本越低，而构建成本则随着BFs的重建越频繁而增加。我们能否更好地根据数据大小调整这些成本，同时减轻读/写争用?

False Positive Rate Analysis. 我们将假阳性率（FPR）定义为所有过滤器的FPP之和。FPR表示由于整个LSM树上每个点查询出现误报而导致的平均I/O数。等式2表示大多数KV存储的FPR，它们为每个条目分配相同数量的位给所有BF。然而，这种方法最近被认为是次优的。最佳方法是重新分配 ≈ 从最大level开始，每个条目1位，并使用它将每个条目的更多位线性分配给较小level 的过滤器[25,26]。虽然这增加了最大水平的FPP，但它在较小水平上以指数方式降低FPP，从而使总体FPR更低，如等式3所示[28]。

$$FPR_{uniform} = 2^{-M \cdot \ln(2)} \cdot (K \cdot (L-1) + Z) \tag{2}$$

$$FPR_{optimal} = 2^{-M \cdot \ln(2)} \cdot Z^{\frac{T-1}{T}} \cdot K^{\frac{1}{T}} \cdot \frac{T^{\frac{T}{T-1}}}{T-1} \tag{3}$$

方程3表明，与方程2不同，采用最优方法时，内存和FPR之间的关系与level 数无关，因此与数据大小无关。原因是，随着LSM树的增长，较小的level 被分配成指数级的较低FPR，从而导致FPR的总和收敛。我们为LSM树的Bloom过滤器设计的任何替代品必须匹配或改进方程式3中表示的FPR。

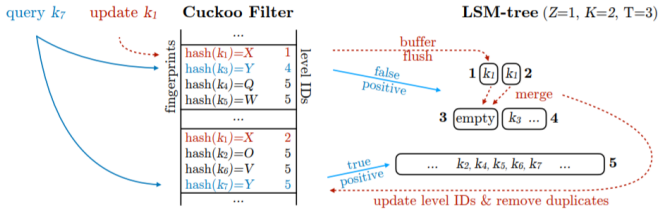
3. 承诺：LSM-TREE和布谷鸟过滤器

布谷鸟过滤器[39]（CF）是最近出现的几种数据结构[9、15、44、81、96]中的一种，它们是Bloom过滤器的替代品。在它们的核心中，这些结构都使用了一个紧凑的哈希表来存储密钥的指纹，其中指纹是通过对密钥进行哈希运算得到的F位串。CF包含一组桶，每个桶都有用于指纹的S槽。在插入过程中，使用等式4将带有键k的条目散列到两个bucket地址b_1和b_2。密钥k的指纹被插入到任何有空间的存储桶中。但是，如果两个存储桶都已满，则随机选择其中一个存储桶的某些指纹，并将其交换到另一个存储桶以清除空间。通过使用xor运算符，等式4的右侧允许始终使用指纹和当前存储桶地址计算条目的备用存储桶，而不使用原始密钥。交换过程以递归方式继续，直到找到所有指纹的可用插槽，或者直到达到交换阈值，此时插入失败。

$$b_1 = hash(k) \qquad b_2 = b_1 \oplus hash(k's\ fingerprint) \quad (4)$$

我们通过CF论文将S设置为每个桶四个槽。这样的调优可以达到95%的空间利用率，并且很有可能不会导致插入失败，并且每次插入只需1-2次摊销交换。假阳性率为$\approx 2 \cdot S \cdot 2^{-F}$，其中F是以位为单位的指纹大小。查询最多需要两个内存I/O，因为每个条目位于两个存储桶中的一个。

许诺. 布谷鸟过滤器支持为每个条目及其指纹存储可更新的辅助数据。我们建议将LSM树的多个Bloom过滤器替换为一个CF，该CF不仅将每个数据条目映射到指纹，还映射到一个level ID（LID），映射到包含该条目的sub-level 。其承诺是允许查找包含最多两个内存I/O的给定项的run，这比使用Blocked Bloom筛选器要便宜得多。



Term	Definition
B	CF bucket size (bits)
F	fingerprint size (bits)
D	level ID size (bits)
S	slots per CF bucket
p_i	fraction of data at Level i
f_j	probability of Level ID j
H	entropy of level ID probability distribution
H_{comb}	entropy of combination probability distribution
C	the set of all level ID combinations
C_{freq}	the set of most probable level ID combinations

Figure 3: Chucky uses a Cuckoo filter to map entries in the LSM-tree, and it keeps this mapping up-to-date during compactions.

同时，方程式5通过从内存预算M中减去LIDs 大小D给出了FPR。挑战在于保持D小，以便FPR也保持小。

$$FPR \approx 2 \cdot S \cdot 2^{-F} = 2 \cdot S \cdot 2^{-M+D} \quad (5)$$

案例研究. . SlimDB[86]是第一个用带LIDs 的布谷鸟过滤器取代LSMtree Bloom过滤器的系统。因此，它提供了一个有趣的案例研究。SlimDB将LID编码为固定长度整数。因此，每个LID至少包含$\log_2(A)$位，以唯一地标识所有run，其中A是等式1中的子级总数。通过在方程式5中插入$\log_2(A)$作为D并替换A，FPR简化为方程式6，这表明FPR随着level L的数量增加而增加。原因是随着数据的增长，LIDs会从指纹中删除位，以便能够唯一地识别更多的sub-level 。随着数据量的增长，FPR是否可能保持较低和稳定?

$$FPR_{binary} \approx 2 \cdot S \cdot 2^{-M} \cdot (K \cdot (L-1) + Z) \quad (6)$$

SlimDB的另一个问题是，它在每次应用程序写入之前访问存储，以检查有问题的条目是否存在。如果是这样，它会在CF中修改条目的LIDs ，以反映更新条目的位置。这会在写入路径上产生大量开销。我们可以在不进行先读后写操作的情况下使LIDs 保持最新吗?

4. CHUCKY

Chucky是一种新型的LSM树过滤器，可同时扩展内存带宽、内存占用和误报率（FPR）。它通过使用布谷鸟过滤器（CF）变体替换Bloom过滤器来实现这一点，该变体使用level ID将每个条目映射到LSM树中的sub-level 。它沿着两个方向进一步创新。

Opportunistic Maintenance. Chucky在合并操作期间，在不增加存储I/O成本的情况下，随时更新CF中的level ID。我们将在第4.1节中对此进行讨论。

Level ID 压缩. Chucky压缩level ID，以防止其大小随着数据的增长而增加并从指纹中窃取位。因此，Chucky将FPR保持在较低水平。我们将在第4.2节中展示如何压缩level ID。我们分别在第4.3节和第4.4节中确定并讨论了压缩level ID对bucket对齐和计算效率的影响。第4.5节涵盖了其他设计考虑事项。

4.1与LSM树的集成

图3展示了Chucky的架构。对于LSM树中的每个数据条目，都有一个CF条目，由一个指纹和一个level ID（LID）组成，该ID映射条目的当前sub-level 编号。图3还分别用蓝色实心箭头和红色虚线箭头说明了Chucky的查询和更新工作流。

Querying. Chucky通过访问键映射到的两个CF buckets来处理查询（使用等式4）。对于这两个buckets 中的所有匹配指纹，它从最年轻到最老搜索相应的run。例如，在图3中，应用程序查询键k_7。Chucky将此密钥映射到两个bucket，这两个bucket都有一个具有匹配指纹Y的条目。一个映射到sub-level 4，一个映射到sub-level 5。查询首先搜索sub-level 4，因为它包含较年轻的数据，但会导致误报。然后它搜索sub-level 5，在那里它成功地找到了目标条目。由于查询访问两个CF存储桶，因此无论数据大小或合并策略如何，开销都是两个内存I/O。

Inserting New Data. 每当memtable被刷新到存储器时，Chucky就会为批处理中的每个键（包括墓碑）添加一个CF条目。开销大约是每个条目两个内存I/O，因为条目可以跨存储桶交换以清除空间。例如，考虑图3中的条目k_1，它最初在Sub-Level 2中有一个版本。然后将此条目的新版本刷新到sub-level 1。Chucky为新版本添加了一个新的映射条目，同时仍将旧版本的映射保留在CF中。这与SlimDB不同，SlimDB将发出一个存储I/O，以检查是否存在键为k_1的条目，如果存在，则更新CF中现有映射条目的LID。因此，Chucky将删除SlimDB的先读后写操作。结果是更好的表现。取舍是，Chucky必须在压缩之前在其CF中映射过时的条目，这与SlimDB不同，但与Bloom Filters类似，对于Bloom Filters，相同条目的不同版本在压缩之前也会占用多个过滤器的空间。Chucky可能出现的一个问题是，如果同一条目的过时版本太多，CF bucket可能会溢出，因为它们的映射条目都放在同一对CF bucket中。我们在第4.5节中通过扩展桶处理此类溢出。

维护. 当一个条目在压缩过程中移动到一个新的sub-level 时，Chucky会在将其放入内存合并时更新CF中的LIDs 。另一方面，对于压缩过程中识别的每个过时条目，Chucky会从CF中查找并删除相应的映射条目。例如，在图3中，会触发压缩，将sub-level 1和2的两次run合并为sub-level 3的一次run。在此操作过程中，sub-level 2中较旧版本的密钥k_1将从CF中删除。对于较新版本的k_1，LID将从1更新为3以反映新的sub-level 。这种方法不需要在已经发布以进行压缩的存储I/O之上添加任何额外的存储I/O。在可能包含目标项的两个可能的CF存储桶中查找目标项，平均内存访问成本为1.5 I/O。

有趣的是，当条目在压缩后保持在同一sub-level 时，条目的LID不需要更新。例如，假设图3中的sub-level 3和4现在与sub-level 5合并，结果run保持在sub-level 5，因为它没有超过其容量。Y。在本例中，条目k_2、k_4、k_5、k_6和k_7的LIDs 保持不变。相比之下，使用Bloom过滤器，我们必须在sub-level 5重建过滤器，并以显著的内存I/O成本重新插入每个条目。因此，LID最多更新L次，每次更新一次。因此，总体开销最多≈ 1.5·L每个应用程序插入/更新/删除的摊销内存I/O，与合并策略无关。

	Leveling	Lazy-Leveling	Tiering
application query	$O(1)$	$O(1)$	$O(1)$
application update	$O(L)$	$O(L)$	$O(L)$

Table 2: Chucky's memory I/O complexities.

Memory I/O 复杂性. 表2总结了Chucky的内存I/O复杂性。相对于表1中的BFs，Chucky的核心优势是将查询成本降低到一个小常量，该常量与数据大小和合并策略无关。Chucky还降低了升级和延迟升级的更新成本复杂性，从而消除了更新成本对合并策略的依赖。实际上，对于分层和延迟升级，Chucky的更新成本为 ≈ 每个条目1.5·L内存I/O可能比Blocked BFs稍微贵一些。然而，这被大量的点读取成本降低所抵消。

与CPU缓存的相互作用. 对于具有点倾斜的工作负载，即应用程序重复读取相同的数据条目，Chucky可以在CPU缓存中容纳更大的工作集。原因是对于任何频繁访问的条目，只需要缓存两个CF bucket。然而，对于被阻止的BFs，对于这样的条目，最多需要缓存一个过滤器。另一方面，对于具有区域倾斜的工作负载，相同时间或空间区域中的条目更有可能被读取，BFs可能更适合CPU缓存，因为它们在时间上（跨sub-level ）和空间上（跨sub-level 内的SST）更细粒度。就更新成本而言，LSM树的较小run的

BFs实际上可能适合CPU缓存，因此需要比表1中预测的更少的内存I/O。CPU缓存的效果很微妙。虽然可以预见到有利于BFs的工作负载，但Chucky提供了更好的最坏情况保证，从而使性能在所有情况下都更加健壮，尤其是随着数据的增长。

4.1 压缩 Level IDs

本节建立了LIDs 可压缩性的理论界限，并详细探讨了其编码设计空间。

LID 概率分布。 方程式7将p_i表示为LSM树在i级的总容量的分数。左侧的表达式更精确，但随着level 数的增加，会快速收敛到右侧的表达式。正如预期的那样，较小level 的容量正呈指数级下降。

$$p_i = \frac{T-1}{T^{L-i}} \cdot \frac{T^{L-1}}{T^L - 1} \qquad \lim_{L\to\infty} p_i = \frac{T-1}{T} \cdot \frac{1}{T^{L-i}}, \tag{7}$$

等式8表示f_j为LSM树在子级j的容量分数，它是Level [j/K]的一部分。 Equation 8 is derived by dividing the level's capacity p [j/K] (from Eq. 7) by the number of sub-levels $A_{\lceil j/K \rceil}$ at that level (from Eq. 1). For example, in Figure 3 Sub-Level 5 is at Level ⌈5/2⌉ = 3, and so it comprises a fraction of $f_3/A_3 \approx 0.62$ of the overall LSM-tree's capacity.

$$f_j = \frac{p_{\lceil j/K \rceil}}{A_{\lceil j/K \rceil}} \tag{8}$$

让我们假设LSM树的所有sub-level 都已满负荷。我们还假设在不同的run中，平均数据条目大小是相同的。在这两个假设下，等式8给出了从布谷鸟过滤器中随机选择的LIDs 对应于子级j的概率。换句话说，等式8成为CF内LIDs 的概率分布。

所有sub-level 都已满的假设反映了内存压力最高的情况。为了在最坏的情况下优化内存占用，我们在第4节的其余部分中保持这一假设。

熵. 等式9导出LID概率分布的香农熵，它表示在最大压缩后表示LID所需的平均比特数。我们通过在左边陈述熵的定义，插入方程8中的f_j，将sub-level A的数量和数据大小取为无穷大，并进行简化来推导。有趣的是，熵收敛于sub-level A的数量，因此也收敛于数据大小。直觉是，较小level 的LID概率指数下降胜过较小level 的LID需要更多位来唯一表示的事实。

$$H = \lim_{A\to\infty} \sum_{j=1}^{A} -f_j \cdot \log_2(f_j) = \log_2\left(Z^{\frac{T-1}{T}} \cdot K^{\frac{1}{T}} \cdot \frac{T^{\frac{T}{T-1}}}{T-1}\right) \tag{9}$$

Chucky的FPR下限。通过将等式9中的熵H作为等式5中的LIDs 尺寸D，我们得到了等式10中Chucky的乐观FPR近似值。通过尽可能多地压缩每个LID并将所有剩余位分配给指纹，这是我们期望的最低FPR。我们观察到，对于等式6中具有整数编码LID的CF，相对于数据大小，该界渐近低于FPR。它也渐近低于等式2中均匀分配的Bloom过滤器的FPR上界。F最后，与方程3中具有最佳分配Bloom过滤器的FPR上界相比，我们观察到，当方程10具有更高的乘法常数2·S时，FPR相对于内存的下降速度更快(i.e., $\propto 2^{-M}$ 相对于 $\propto 2^{-M\cdot ln(2)}$). 这意味着对于足够高的内存预算（M≥ 在FPR方面，Chucky应该能够击败最先进的Bloom过滤器。这一理论发现重申了我们的方法。

$$FPR_{chucky} \gtrsim 2 \cdot S \cdot 2^{-M} \cdot Z^{\frac{T-1}{T}} \cdot K^{\frac{1}{T}} \cdot \frac{T^{\frac{T}{T-1}}}{T-1} \tag{10}$$

哈夫曼编码。 Chucky在实践中使用Huffman编码[46]来压缩LIDs 。作为输入，哈夫曼编码器获取特定LSMtree配置（即T、K、Z和L）的LIDs及其概率分布（即，等式8）。作为输出，它返回一个代码来表示每个LID，其中概率较高的LID被分配较短的代码。它通过首先将最不可能的LID连接为子树，从LID创建二叉树来实现这一点。LID的最终代码长度对应于其在结果树中的深度。图4展示了一个LSM树的示

例，该树带有标记的LIDs，其概率来自等式8。例如，LIDs 6包含5/124的分数≈ LSM树容量的4%，因此，当所有sub-level 都满时，≈ 布谷鸟过滤器中4%的入口具有6的LIDs 。Huffman编码器为这个LSM树实例创建旁边显示的树。给定LID的代码是通过在从树的根到给定LID的叶节点的路径上连接树的边缘标签而派生的。例如，LIDs 4和9的编码分别为011011和1。

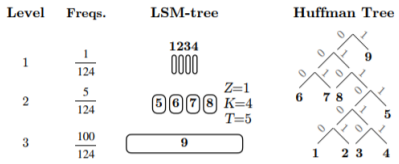


Figure 4: A Huffman tree encodes each LID uniquely such that the average code length is minimized.

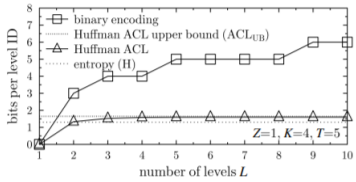


Figure 5: Compression causes the LIDs' average code length to converge with respect to data size.

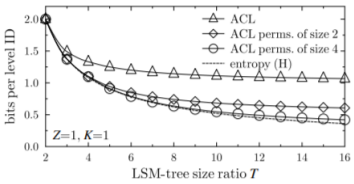


Figure 6: The average code length approaches the entropy as larger permutations of LIDs are used.

可剥性. 对于哈夫曼编码，没有代码是另一个代码的前缀[46]。此属性允许对输入比特流进行唯一解码，方法是从根开始遍历哈夫曼树，直到到达一个叶，在给定叶输出LID，然后在根重新开始。例如，输入比特流11001根据图4中的哈夫曼树被唯一地解码为lid9、9和7。此属性允许我们唯一地解码一个桶中的所有LIDs，而无需分隔符号。

Average Code Length. 我们使用编码LID的平均代码长度（ACL）测量编码LID的大小，定义为 $\sum_{j=1}^{A} l_j \cdot f_j$，其中$l_j$是分配给LID j的代码长度。 例如，该方程为图4中的哈夫曼树计算1.52位。与整数编码相比，这节省了62%，整数编码需要四位来唯一地表示九个LID中的每一个。

内存占用分析. 信息论中众所周知，哈夫曼码ACL的上界是熵加一[46]。增加一个的直觉是，每个代码长度必须向上舍入到整数位数。我们将其表示为ACL ≤ H + 1，其中H是等式9中的熵。因此，我们期望ACL也收敛并独立于数据大小，类似于等式9。我们在图5中通过增加图4中示例的level 数并绘制哈夫曼ACL来验证这一点，哈夫曼ACL确实收敛（与整数编码的LID相反）。原因是，虽然较小level 的run分配了较长的代码，但它们的概率呈指数级降低，因此较大level 的较小run代码支配ACL。

Tight ACL Upper Bound. 众所周知，哈夫曼编码是最优的，因为它最小化了ACL[46]。然而，精确的ACL很难分析，因为哈夫曼树结构很难从一开始就预测。相反，我们可以通过假设一种不太通用的编码方法，并观察到哈夫曼ACL至少会同样短，从而得出比H+1更严格的ACL上界。让我们使用（1）长度为L的一元编码前缀来表示每个LID − i + 1位表示level i，后跟（2）长度的截断二进制编码后缀≈ $log_2(Ai)$ 唯一地表示level i的每个A_i sub-level 。这实际上是一种Golomb编码[43]。我们在等式11中推导出该编码的平均长度为ACL_{UB}，并在图5中作为哈夫曼ACL的合理紧上界加以说明。

$$ACL_{UB} = \lim_{L\to\infty} \sum_{i=1}^{L} p_i \cdot \left(L - i + 1 + \log_2(A_i)\right) = \frac{T}{T-1} + \log_2\left(K^{\frac{1}{T}} \cdot Z^{\frac{T-1}{T}}\right) \quad (11)$$

接近熵。 图5还绘制了等式9中LID概率分布的熵。如图所示，哈夫曼ACL和熵之间存在差距。图6显示，随着LSM树的大小比T的增加，ACL和熵之间的差距增大；ACL接近1，而熵趋于零。原因是，较大的大小比率会将较高比例的数据推到较大的level ，从而增加LID概率分布的倾斜。倾斜越大，每个LIDs 携带的信息越少，导致熵越低，因此压缩性越高。但是，每个LID至少需要一位代码来表示，因此ACL不能降到一位以下。因此，我们无法控制可压缩性的增加。

Level ID 排列. 由于每个CF存储桶都有多个LID，我们可以通过对多个LID进行集体编码来推动每个LID一位的压缩屏障。哈夫曼树的标签是烫发。图7给出了一个玩具示例，展示了如何一次将两个LIDs 编码为排列，通过将大小为S的每个可能排列（在本例中为两个）及其概率（组成LIDs 概率的乘积）输入哈夫曼编码器获得。如图所示，ACL现在下降到一位以下，这是因为它用少于其中LID数的位表示最可能的排列。有趣的是，图6显示，当我们增加排列中集体编码的LID数量时，ACL接近熵。

Level ID 组合. 为了进一步推动压缩，我们可以对组合进行编码，而不是对LIDs 进行排列。与排列不同，组合忽略了有关条目顺序的信息。因为可能的组合比CF bucket内的LIDs 排列少($\binom{S+A-1}{S}$ 相比于A^S), 我们平均需要更少的位来表示它们。

LID组合的概率分布是多项式的。对于n个独立试验，每个试验都导致k个类别中的一个成功，每个类别都有固定的成功概率，多项式分布给出了各个类别中任何特定成功组合的概率。在我们的案例中，试验次数是每个CF桶的槽数，不同类别是A LIDs，成功概率由等式中的LID概率分布给出。

现在，让我们将c（j）表示为组合c中lidj的出现次数。等式12给出了使用多项式分布的组合c的概率cprob。通过将所有组合及其概率输入图7中示例的哈夫曼编码器，我们获得了名为Combs的哈夫曼树，其中组合12替换了两个先前的排列12和21。对于这个组合，我们有S=2，c（1）=1和c（2）=1，所以它的概率是2！·(1/11) · (10/11) = 20/121.

$$c_{prob} = S! \cdot \prod_{j=1}^{A} \frac{f_j^{c(j)}}{c(j)!} \tag{12}$$

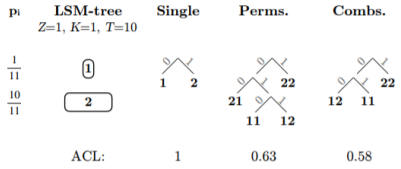


Figure 7: Encoding level IDs as permutations or combinations allows reducing the average code length below one.

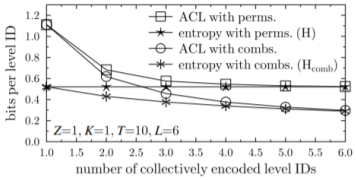


Figure 8: Encoding level IDs as large combinations maximizes compressibility.

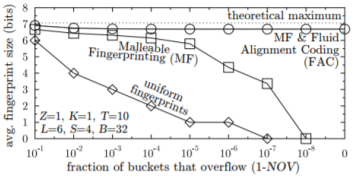


Figure 9: Chucky resolves the contention between fingerprint size and bucket overflows via FM and FAC.

组合分析。 组合排除了有关排序的信息这一事实导致了熵的减少。方程13通过使用多项式分布的熵函数并插入s、A和方程9，导出H_{comb}作为LID组合分布的熵。图8显示，当我们增加组合大小时，H_{comb}相对于H下降，因为它消除了更多的排序信息。这会导致压缩性增加。

$$H_{comb} = H - \frac{1}{S} \cdot \left(\log_2(S!) - \sum_{i=1}^{A} \sum_{j=0}^{S} \binom{S}{j} \cdot f_i^j \cdot (1 - f_i)^{S-j} \cdot \log_2(j!) \right) \tag{13}$$

带有组合的ACL是$\sum_{c \in c}$（$l_c \cdot c_{propb}$）$/S$，其中c是所有组合的集合，lc是组合c的代码长度（我们除以S表示每个LID而不是每个bucket的ACL）。我们观察到，组合ACL严格低于图8中的置换ACL，并且随着我们增加集体编码LID的数量，组合ACL与组合熵收敛。在本文的其余部分中，我们继续使用编码组合，因为它们实现了最佳压缩。

Bucket 结构. Chucky中的每个CF桶都以一个组合码开始，后跟S指纹。由于组合代码排除了有关排序的信息，因此桶中的指纹根据其LIDs 进行排序，以便能够推断哪个指纹对应于哪个LIDs 。

4.3 将level ID代码与指纹对齐

由于LIDs 因压缩而变长，因此将其与CF桶内的指纹对齐成为一项挑战。我们在图10第（A）部分用16位CF存储桶描述了这一挑战，这些存储桶需要为两个条目存储一个组合码以及两个五位指纹（FPs）。这个例子基于图4中的LSM树实例，只是我们现在对组合进行编码，而不是对每个LID进行单独编码。图中的术语$l_{x,\ y}$是分配给具有LIDs x和y的桶的代码长度。我们观察到，一些代码和指纹在一个桶内完全对齐（第一行）。然而，另一些则出现下溢（第二行），这意味着存储桶末端的一些位未使用。还有一些存储桶出现溢出（第三行和第四行），这意味着代码和指纹的累积长度超过了存储桶的大小。由于组合代码较短，底流发生在具有更多可能LIDs （属于较大液位）的铲斗上。它们是不受欢迎的，因为它们浪费了原本可以用来产生更大指纹的碎片。另一方面，由于组合代码较长，溢流发生在LIDs 可能性较小（属于较小level ）的桶中。它们是不可取的，因为它们需要将桶中的其余内容存储在其他地方。这可能导致较差的内存利用率和较高的访问成本。

图10第（A）部分暗示了存储桶溢出倾向和指纹大小之间存在争用。虽然减小指纹大小可以减少某些存储桶中的溢出，但总体上会导致过滤器的误报率更高。我们在图9中用标记为均匀指纹的曲线证实了这一论点。x轴测量溢出CF桶的比例，而y轴测量指纹大小。是否有可能消除此争用，以同时保证很少的溢出和较大的指纹？

可塑性指纹（MF）. 为了更好地对齐代码和指纹，我们引入了MF。我们的目标是平衡这样一个事实，即来自更高level 的条目往往具有更小的组合代码长度，并使用存储桶中的空闲空间来拥有更长的指纹。因此，MF在LSM树的较小level 分配条目，使指纹更短。然而，当它们被合并到更大的level 时，它们会被分配更长的指纹。

MF的问题是如何为每个level 选择指纹长度，以便仔细控制指纹长度和桶溢出之间的平衡。我们将其视为一个约束优化问题，目标是最大化平均指纹长度，$\sum_{i=1}^{L} FP_i \cdot p_i$，其中，$FP_i$是一个整数，表示i级条目的指纹长度。 这个问题是为$2^B > (^{S+A-1}uS)$定义的，这意味着桶大小B必须至少足够大，以唯一地标识所有组合。 We我们使用一个参数NOV来约束问题，该参数用于我们希望保证的非溢出桶的分数（理想情况下至少为0.9999）。我们使用此参数将C_{freq}定义为C的子集，该子集只包含C中最可能的LID组合，因此它们的累积概率刚好高于NOV^1。 将等式14定义为一个约束，要求所有 $c \in C_{freq}$, q, 代码长度（表示为l_c）加上累积指纹长度（表示为c_{FP}）不超过$bucket^2$ 中的位数B。

$$\forall \in C_{freq} : c_{FP} + l_C \leq B \qquad (14)$$

虽然已知涉及整数的优化问题很难解决，但我们使用算法1中所示的有效山岭限制方法来利用问题的特殊结构。该算法将所有指纹长度初始化为零。然后，它尽可能增大较大level 的指纹大小，如果违反等式14中的溢出约束，则移动到下一个较小level 。

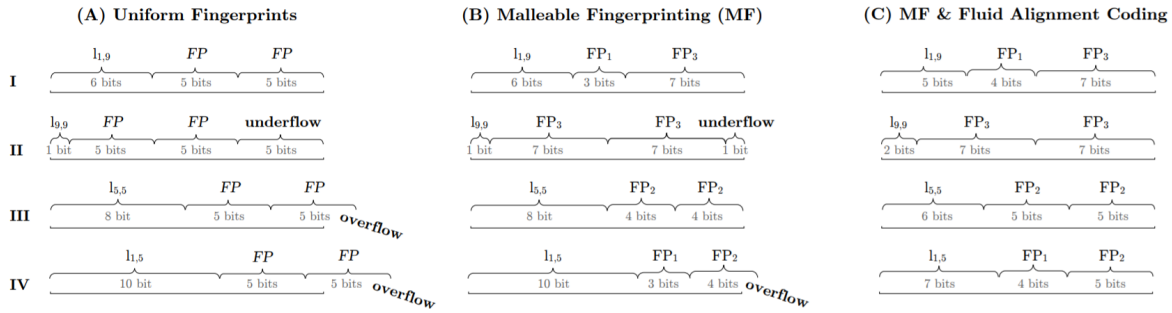


Figure 10: Storing compressed level ID codes with uniformly sized fingerprints leads to poor bucket alignment (Part A). We solve this problem using Malleable Fingerprinting (Part B) and Fluid Alignment Coding (Part C).

1 **for** $i \leftarrow 1$ **to** L **by** 1 **do** $FP_i \leftarrow FP_{min}$ **end**
2 $FP_{max} = M - 1$
3 **for** $i \leftarrow L$ **to** 1 **by** -1 **do**
4 **for** $b \leftarrow FP_{min} + 1$ **to** FP_{max} **by** 1 **do**
5 temp $\leftarrow FP_i$
6 $FP_i \leftarrow b$
7 **if** *overflow constraint is violated* **then**
8 $FP_i \leftarrow$ temp
9 FP_{max} = temp

Algorithm 1: Maximizing the average fingerprint size by hill-climbing.

首先延长较大level 的指纹的基本原理是，它们的条目在CF中更常见。因此，该算法遵循最大化目标函数的最陡上升。图10第（B）部分给出了MF如何允许更大level 的条目具有更长的指纹（第二行），同时消除一些溢出（第三行）的示例。结果是溢出和平均指纹大小之间有了更好的平衡，如图9所示。

由于MF在不同level 为同一条目的不同版本分配不同的指纹长度，因此出现了一个问题，即布谷鸟过滤器可以将同一条目的这些不同版本映射到两个以上的CF桶。原因是等式4依赖于条目的指纹来计算备用存储桶位置，因此不同的指纹长度将导致不同的存储桶地址。我们通过确保所有指纹至少包含$FPmin$位来解决这一问题，并根据条目的第一个$FPmin$位调整CF以确定条目的备用存储桶。这将强制

同一条目的所有版本驻留在同一对CF存储桶中。虽然该约束略微减小了算法1给出的平均指纹大小，但由于没有为任何条目分配非常小的指纹，因此它提供了较低的FPR方差。根据最初的CF论文[39]，我们将FP_{min}设置为五位，以确保条目的两个存储桶足够独立，从而实现95%的空间利用率。

流体校准编码（FAC）. 图10第（B）部分说明，即使使用MF，仍会发生下溢和溢流（分别为第II行和第IV行）。为了进一步缓解这些问题，我们引入了FAC。FAC利用了一个众所周知的折衷方法，即在哈夫曼码中分配的某些代码越小，其他代码必须越长，所有代码才能保持唯一可解码。这种权衡体现在卡夫-麦克米兰不等式[64,75]中，该不等式指出，对于给定的一组代码长度L，所有代码都可以唯一解码，如果$1 \geq \sum_{l \in L} 2^{-l}$. 直觉是，代码长度是从总计为1的预算中设置的，较小的代码消耗的预算比例较高。

为了利用这种折衷，FAC为最可能的桶组合分配更长的代码来占用下溢位。因此，不太可能的铲斗组合代码可以缩短。这会在可能性较小的存储桶中创建更多空间，利用这些存储桶可以减少溢出并增加较小level 的指纹大小。图10第（C）部分说明了这一想法。第二行中的组合是系统中最常见的组合，现在被分配一位更长的代码以消除下溢。这允许减少所有其他组合的代码长度，从而允许为level 1和2的条目设置更长的指纹，并消除第IV行中的桶溢出。

我们在MF之上实现FAC，如下所示。首先，我们将等式14中先前的约束替换为等式15中给出的新约束。用Kraft-McMillan不等式表示，它确保指纹大小保持足够短，以便仍然可以为C_{freq}中的所有组合构造具有唯一可解码代码的非溢出桶。它还确保所有不在C_{freq}中的bucket组合都可以使用最大为bucket B大小的唯一代码进行唯一标识。

$$1 \geq \sum_{c \in C} \begin{cases} 2^{-(B-c_{FP})}, & \text{for } c \in C_{freq} \\ 2^{-B}, & \text{else} \end{cases} \tag{15}$$

方程15不依赖于预先知道哈夫曼码（即，如方程14所示）。因此，我们在使用算法1查找指纹长度之后而不是之前运行哈夫曼编码器。 此外，我们仅在$cfreq$中的组合上运行哈夫曼编码器，同时为组合2^{-B-cFP} 设置概率输入，而不是像以前那样使用其多项式概率（在等式12中）。这会导致哈夫曼编码器生成完全填充剩余位的代码B − c_{FP} . 对于不在c_{freq}中的所有组合，我们设置大小为B位的统一大小的二进制代码，它由哈夫曼树中的公共前缀和唯一后缀组成。因此，我们可以唯一地解码这两个集合。

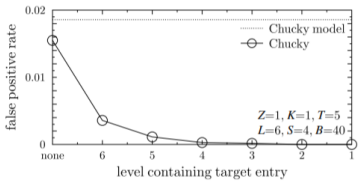


Figure 11: The FPR decreases exponentially as newer entries are accessed by queries.

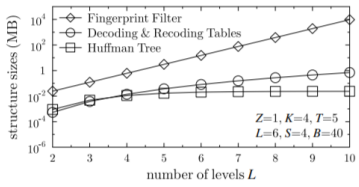


Figure 12: The Huffman tree size converges while the de/recoding table sizes grow slowly with data size.

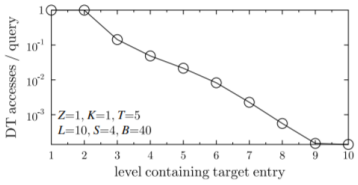


Figure 13: The decoding table access cost increases slowly with data size until flattening at one memory I/O.

图9中的水平曲线显示，MF和FAC在一起应用时消除了溢出和指纹大小之间的争用；指纹保留时间长，同时溢出量少。取舍是平均代码长度比以前稍长。原因是，通过占用最可能的组合码的下溢位，FAC使得ACL至少有S位长(≥ 每个条目1位）。这意味着实现良好的铲斗对准需要牺牲一些空间。图9以MF和FAC曲线与标记为理论最大值的曲线之间的间隙来衡量这种牺牲，该曲线是通过从内存预算M中减去熵（从等式13中）得到的。它表示为≈ 每个条目1/2位对于我们的示例，价格适中。在论文的其余部分，我们默认使用MF和FAC。

施工时间. 算法1的运行时复杂性为 $O((L+M-M_{min}) \cdot |C|)$, $L+M-M_{min}$ 是迭代次数，| C |是计算等式15中约束的成本。此外，哈夫曼编码器的时间复杂度是$O(|C_{freq}| \cdot log_2(|C_{freq}|))$. 要以封闭形式更松散地表达这些边界，请注意$|C_{freq}| \leq |C| = \binom{A+S-1}{S} < (A+S-1)^S \cdot (e/S)^S < A^S$. 只有当LSM树level 的数量发生变化时，才很少调用此工作流，并且可以脱机执行。因此，它的运行时是实用的。图9中的每个点都需要几分之一秒的时间来生成。

False Positive Rate (FPR). Chucky的FPR很难精确分析，因为指纹大小不一，事先不知道。相反，我们保守地近似FPR，以允许对系统行为进行推理。我们使用等式11中的ACL上限ACL_{UB}稍微高估了FAC每个条目的平均组合代码长度。通过在等式5中为D插入ACL_{UB}，我们得到等式16，对于该等式，解释是对不存在的密钥的查询的预期误报数。

$$FPR_{chucky} \approx 2 \cdot S \cdot 2^{-M} \cdot 2^{\frac{T}{T-1}} \cdot K^{\frac{1}{T}} \cdot Z^{\frac{T-1}{T}} \tag{16}$$

图11将等式16与Chucky的实际FPR进行了比较。x轴改变目标条目所在的level ，其中6是最大level 的ID，“none”表示目标条目不存在。y轴测量每个查询产生的误报平均数。当目标条目处于较小level 时，FPR呈指数下降。原因是点读取从最小到最大访问相关level （以便能够找到条目的最新版本），并在找到第一个匹配条目后终止。因此，误报发生率呈指数级下降，因为平均而言，目标两个存储桶中的条目数量呈指数级下降，对应的level 甚至比目标条目所在的level 更小。该图表明，等式16在合理范围内预测了目标条目不存在的情况，并且它在条目存在的所有情况下提供了可靠的上限。

4.4 优化解码和重编码

我们现在讨论如何在应用程序读写期间有效地解码和重新编码组合代码。

Cached Huffman Tree. 哈夫曼码通常通过从根节点到叶节点遍历哈夫曼树一次解码一位。一个可能的问题是，遍历它可能需要每个访问的节点多达一个内存I/O。当有更多level 时，随着哈夫曼树变得更深，此成本会随着数据大小的增加而增加。为了限制这一成本，我们观察到等式12中的桶组合分布是重尾分布。因此，在CPU缓存中保留一个小的哈夫曼树是可行的，以允许仅快速解码最常见的组合码。因此，我们只为集合$Cfreq$中最常见的LID组合构造一个哈夫曼树，并将参数NOV设置为0.9999，以便集合$Cfreq$包含CF中最常见组合的99.99%。图12显示了相应的哈夫曼树的大小相对于数据大小收敛。原因是，给定桶组合的概率（在等式12中）相对于层数收敛，因此，根据其组成成员的累积概率定义其大小的任何集合，其大小也相对于层数收敛。此属性确保随着数据的增长，Huffman树不会超过CPU缓存大小。

解码表 (DT).除了哈夫曼树之外，主存中还有一个解码表，允许解码C_{freq}以外的组合码。为了确保DT的快速解码速度，我们利用了上一小节中的属性，即所有不在C_{freq}中的bucket组合都具有统一大小的代码。因此，我们将DT构造为一个数组，其中索引i包含代码i对应的LIDs 。这保证了在一个内存I/O中的解码速度。图12在我们增加xaxis上的level 数时测量DT大小（每个DT条目为8个字节）。 因为DT包含 $\approx |C| = \binom{A+S-1}{S}$ 条目, 它的大小随着数据大小的增加而缓慢增长，即使对于具有十个level 的大型LSM树实例，它的大小也保持在1MB以下。

当point以较小的level 查询目标数据时，DT更容易被访问。图13改变了目标条目所在的level ，并在日志刻度y轴上测量了每个查询的平均DT访问次数。较小level 的条目访问成本增加的原因是，具有至少一个对应于较小level 的LID的bucket不太可能位于集合C_{freq}中，因此也不太可能位于缓存的Huffman树中。然而，这种开销最终会变平，因此即使在最坏的情况下也会保持适度。总的来说，与普通布谷鸟相比，Chucky在找到匹配指纹之前平均访问1.5个存储桶，而Chucky总是访问两个存储桶，还可能访问解码表，导致三个内存I/O。

溢出哈希表. 为了处理bucket溢出，我们使用一个小哈希表将溢出bucket的ID映射到相应的指纹。它的大小是 $\approx (1 - NOV) = 10^{-4}$ CF的大小. 它很少被访问，也就是说，仅针对不频繁的存储桶组合，并且它支持在O（1）内存i/O中进行访问。

记录表 (RT). 为了在处理应用程序写入时为给定的LID组合找到正确的代码，我们使用了一个重编码表，该表被实现为一个快速静态哈希表。访问它最多需要O（1）个内存I/O，其规模与图12中的解码表相同。请注意，在运行时，最频繁的RT条目位于CPU缓存中，因此访问不需要内存I/O。

空间摘要. 图12显示了我们增加LSM树level 数量时的CF大小。所有辅助数据结构都相对较小，因此不存在空间瓶颈。

4.5 其他设计注意事项

本节讨论其他设计注意事项。

Sizing & Resizing. 当Chucky达到容量时，需要调整其大小以适应新数据。由于CF需要访问基础数据才能调整大小，因此我们利用了将操作合并到LSM树的最大level 会通过整个数据集这一事实。我们利用这个机会还构建了一个新的更大的Chucky实例。

Partitioning. 由于布谷鸟过滤器依赖于xor运算符来定位条目的备用存储桶，因此存储桶的数量必须是2的幂。这可能会浪费高达50%的分配内存，尤其是当LSM树的容量刚刚超过二的幂时。为了更好地利用内存，真空过滤器[96]建议将一个CF划分为多个独立的CFs，每个CFs是2的幂，但CFs的总数是灵活的。使用哈希模运算（类似于Blocked 的Bloom过滤器）将每个键映射到一个组成的CFs。这样，通过改变CFs的数量，容量变得可调。虽然Chucky还不支持这一点，但对于内存敏感的应用程序来说，这是一个重要的未来步骤。

Empty CF Slots. 我们使用保留的全零指纹和最频繁的LID来表示空指纹槽，以最小化相应的组合码长度。

Entry Overflows. 由于CF将来自不同LSM树的同一条目的多个版本映射到同一对CF存储桶中，因此如果给定条目的版本超过2·S，则可能发生存储桶溢出。一些过滤器使用嵌入式指纹计数器解决此问题（例如，计数商过滤器[81]）。然而，Chucky使用了一个额外的哈希表（AHT），它将桶ID映射到溢出的条目。在工作负载繁重的情况下，AHT保持为空。即使更新工作量很大，AHT仍然很小，因为LSM树的设计限制了空间放大，从而限制了每个条目的平均版本数(e.g. 大约 $\frac{T}{T-1} \leq 2$ 用平层还是惰性平层). 在查询或更新过程中，我们会检查AHT中遇到的每一个完整的CF bucket，从而最多增加0（1）个额外的内存访问。

维持. 对于每次运行，Chucky都会保留存储中所有条目的指纹。在恢复过程中，它只从存储器中读取指纹，从而避免对数据进行全面扫描。它将每个指纹及其LIDs 插入一个全新的CF中，每个条目的摊销内存I/O成本几乎不变。这样，恢复在存储和内存I/O方面都是高效的。

Range Reads. 与主流KV存储[3,4,38]类似，Chucky通过在每次运行时访问相关密钥范围来处理读取的范围，而不使用布谷鸟过滤器。因此，范围读取不会直接受到此工作的影响。然而，请注意，具有多个范围读取的应用程序通常选择水平LSM树，因此Bloom过滤器构成了较高的构建开销。Chucky可以通过提高写入吞吐量，从而提高整个系统的性能，间接地提高此类应用程序的性能。

Batch Updates. Chucky可以通过以下方式支持批量更新：（1）以原子方式将批插入WAL和memtable中，（2）将批中的所有条目插入CF中，（3）在memtable已满时将其异步刷新到存储器中，最后（4）以原子方式从读取路径中删除memtable。

5. 评价

我们现在通过实验证明，Chucky使内存和存储带宽比现有设计更稳定。

Baselines. 我们使用我们自己的LSM树实现，基于陀思妥耶夫斯基[28]设计。我们分别在RocksDB[38]和Cassandra[3]中添加了基线阻断[84]和非阻断BFs，并使用均匀假阳性率（FPR）表示设计决策。我们还支持最佳FPR[25]。我们实现了第4节中描述的Chucky。我们支持Chucky的一个版本，该版本具有未压缩的level ID，以松散地表示SlimDB[86]。

Setup. 默认设置包括一个具有1MB缓冲区的延迟level LSM树，大小比为5，六个level 总计为≈ 16GB的数据。每个条目为64B。有1GB的块缓存，数据库块大小为4KB。Chucky每个条目使用10位，并超出5%的配置空间。因此，为所有BF基线分配了1/0.95倍的内存，以均衡基线上的内存。图中的每一点都是三次试验的平均值。我们使用统一的工作负载分布来表示最坏情况下的性能，使用Zipfian分布来创建倾

斜，并在最频繁访问的数据位于块缓存中时显示性能属性。为了考虑调整过滤器大小的开销，任何测量写入成本的实验都从LSM树状态开始，其中除最大的level 外，所有level 都为空。然后，我们用写操作填充它们，直到发生到最大level 的主要压缩，从而导致调整过滤器大小。在图14的（A）至（D）部分中，我们评估了与系统其他部分（如memtable、存储I/O、块缓存、块索引）隔离的过滤器性能。我们主要关注（E）至（H）部分中的端到端性能。

Platform. 我们的机器有32GB DDR内存，Xeon E3-1505M v5，有四个2.8 GHz内核和8MB L3缓存。它运行Ubuntu 18.04 LTS，并连接到750GB Intel Optane SSD DC P4800X。

Memory I/O Scalability. 图14第（A）部分将Chucky的读/写延迟与Blocked 和非Blocked BFs（均具有最佳FPR）进行了比较，随着数据的增长，具有统一的工作负载。

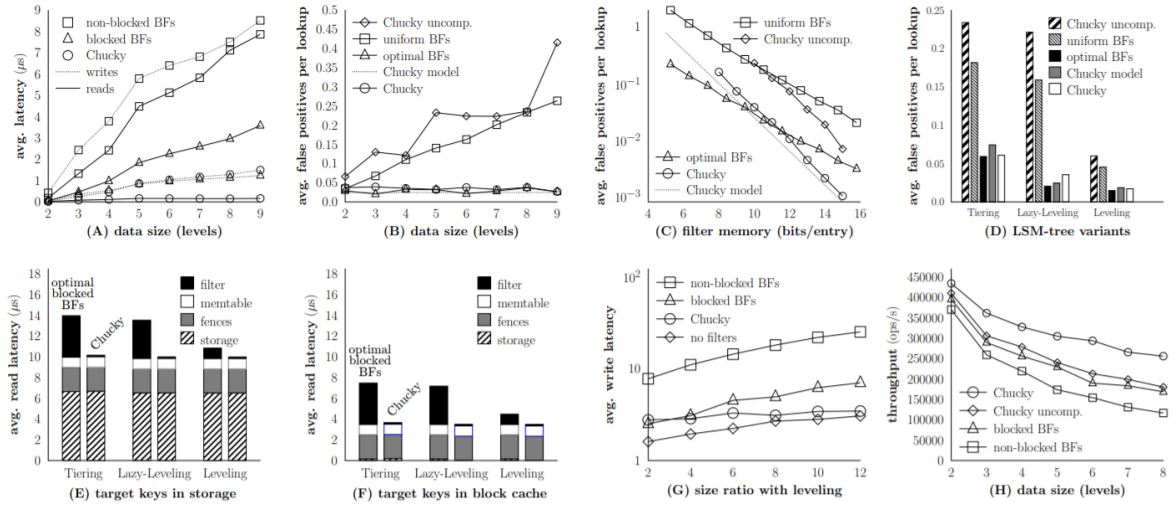


Figure 14: Chucky scales memory I/Os with data size (A) and for any LSM-tree variant (B). At the same time, Chucky's false positive rate scales with data sizes (C) with memory footprint (D), and for any LSM-tree variant (E). Hence, it improves read latency when target data is in fast storage (F) or cached (G), resulting in more scalable throughput (H).

写延迟是用过滤器维护所花费的总时间除以应用程序发出的写操作数来衡量的。读取延迟是在完全合并操作之前测量的（当系统中运行次数最多时），以突出显示最坏情况下的性能。非Blocked BF的延迟增长最快，因为它们需要在越来越多的过滤器中为每个过滤器提供多个内存I/O。对于Blocked BF，读/写延迟增长更慢，因为每次读或写最多需要一个内存I/O。Chucky的写入延迟也随着数据的增长而缓慢增长，因为更新LIDs的level 越来越多。关键的是，Chucky是唯一一个保持低数据量的读取延迟的基线，因为每次读取都需要恒定数量的内存I/O。

FPR 可伸缩性. 图14第（B）部分比较了具有压缩和未压缩LIDs 的Chucky FPR与具有均匀和最佳空间分配的Blocked BFs。随着数据大小的增加，未压缩LIDs 的Chucky的FPR会增加，因为LIDs 会增长并从指纹中窃取位。使用统一的BFs时，FPR也会随着数据大小的增加而增加，因为有更多的过滤器会发生误报。相比之下，对于最佳BFs，较小的level 分配的FPR指数更低，因此FPR的总和收敛到一个与level 数无关的常数。类似地，Chucky的FPR随着数据的增长而保持不变，因为平均LID代码长度收敛，因此允许大多数指纹保持较大。该图还包括方程式16中楚基的FPR模型，以表明它在实践中也给出了FPR的合理近似值。

图14第（B）部分比较了具有压缩和未压缩LIDs 的Chucky FPR与具有均匀和最佳空间分配的Blocked BFs。随着数据大小的增加，未压缩LIDs 的Chucky的FPR会增加，因为LIDs 会增长并从指纹中窃取位。使用统一的BFs时，FPR也会随着数据大小的增加而增加，因为有更多的过滤器会发生误报。相比之下，对于最佳BFs，较小的level 分配的FPR指数更低，因此FPR的总和收敛到一个与level 数无关的常数。类似地，Chucky的FPR随着数据的增长而保持不变，因为平均LID代码长度收敛，因此允许大多数指纹保持较大。该图还包括方程式16中楚基的FPR模型，以表明它在实践中也给出了FPR的合理近似值。

Data in Storage vs. Memory. 图14第（E）部分和第（F）部分分别测量了uniform和Zipfian（参数s=1）工作负载下的端到端读取延迟。读取延迟分为四个部分，包括存储I/O、围栏指针、memtable和筛选器搜索。在第（F）部分中，相关数据通常在存储器中，因此存储器I/O在读取成本中占主导地位。然而，由于我们的SSD速度很快，BFs探测器仍然会带来很大的延迟开销，Chucky能够消除这一开销。另一方面，在（F）部分中，工作负载是倾斜的，这意味着目标数据通常在块缓存中。在这种情况下，

BF会成为瓶颈，因为必须先搜索BF，然后才能识别缓存中的相关块。Chucky缓解了这一瓶颈，从而显著提高了读取延迟。

端到端写入成本. . 图14第（G）部分突出显示了Chucky在增加合并贪婪度时保持低过滤器结构开销的能力（例如，优化范围读取）。我们从左侧的大小比为2的水平LSM树开始，并沿x轴增加它。y轴测量端到端写入成本，其计算方法是将处理这些更新所花费的总时间除以应用程序发布的更新数量。随着大小比率的增加，所有基线的写入成本都会增加，因为相邻level 的run之间有更多的重叠，因此在每个合并操作期间平均需要重写更多的数据。由于合并贪婪度相对较低（即图的左侧），Chucky和Bloom Bloom过滤器的结构开销相似。然而，随着我们增加大小比，Blocked Bloom过滤器的端到端写入成本增加得更快。原因是Bloom过滤器必须在每次合并操作期间从头构造，因此它们的构造开销与LSM树的合并开销成正比。另一方面，在禁用过滤器的情况下，Chucky的性能更接近曲线。原因是，它只会在条目从一个level 移动到下一个level 时更新条目的LID，并且随着大小比率的增大，LSM树中的level 会减少。因此，在贪婪合并策略（通常用于优化范围读取）（例如RocksDB中的默认设置）下，Chucky明显改善了端到端写入。

吞吐量可扩展性. 图14第（H）部分显示了当我们增加由95%Zipfian读取和5%Zipfian写入组成的工作负载的数据大小时，吞吐量是如何扩展的（以YCSB[23]中的工作负载B为模型）。由于写操作是倾斜的，在较小level 的压缩过程中，较新的更新会快速替换较旧的条目，因此不会进行主要的压缩和过滤器大小调整。BF基线不能很好地扩展，因为它们跨越来越多的BF发出内存I/O。随着FPR的增长和存储I/O的增加，未压缩LIDs 的Chucky的性能也在不断恶化。压缩LIDs 的Chucky也表现出性能恶化，主要是因为跨过栅栏指针的二进制搜索成本不断增长。但是，Chucky在数据大小方面比所有基线提供更好的吞吐量，因为它同时扩展了过滤器的FPR和内存I/O。

6. 相关工作

LSM-Tree Performance. 随着LSM树被改编为跨多个系统的存储引擎（例如Cassandra[3]、HBase[4]、AsterixDB[2]、RocksDB[35、38、74]），人们对优化LSM树性能有着极大的兴趣[48、70]。迄今为止，大多数设计都侧重于管理压缩开销，例如，通过将值与键分离[18,68]，将run划分到文件中，并根据最大文件交叉点进行合并[7,92,94]，将热条目保留在缓冲区[7]，使缓冲区更密集[14]或更并发[42]，谨慎地进行调度以防止尾部延迟[8,69,91]，使用定制或专用硬件[1,45,95,97,101]，并通过控制删除持久性[90]。

另一种工作是使用更懒惰的压缩策略[76、85、86、98、99]，这会导致更多的run，从而导致更多的BF发生误报和内存I/O。有几项工作展示了如何在保持FPR适度的同时实现更懒惰的合并策略，但它们仍然会在许多BF中产生许多内存I/O[25、28、29、49–51]。SlimDB[86]展示了如何使用布谷鸟过滤器来减少内存I/O，但是它的内存占用并不像第3节中讨论的那样可扩展。相反，我们展示了如何通过使用压缩level ID替换Bloom过滤器来同时扩展FPR、内存I/O和内存占用（对于任何合并策略，包括惰性策略）。

指纹过滤器. 虽然我们在布谷鸟过滤器[39]的基础上构建了Chucky，使其设计简单，但还有许多其他指纹过滤器设计具有细微的特性。许多人通过使用线性探测[9,81]，将布谷鸟插入偏向一个存储桶[15]，或确保两个候选存储桶在物理上接近[96]，来争取更好的缓存位置。真空过滤器允许过滤器大小不为二的幂，从而提供更好的内存利用率[96]。一些设计允许通过链接溢出过滤器[22,96]或牺牲指纹位[81]来延迟调整过滤器的大小。Xor过滤器支持更好的FPR，以换取更高的构建时间[44,80]。其他设计可防止由于使用内部计数器重复插入而导致的溢出[81]。Morton filter[15]将条目映射到较大的固定大小“块”中的可变大小“插槽”，因此可以更优雅地容纳可变大小的条目。将Chucky与这些过滤器集成，以利用它们的特性，将有助于未来有趣的工作。

Range Filters. 最近的LSM树设计在每次运行时都使用范围过滤器[73100]，这可以为范围读取节省存储I/O，但需要更多的内存I/O来访问和构造。应用Chucky的设计元素在整个LSM树上创建一个统一的范围过滤器，以减少内存I/O，这是一个有趣的未来方向。

Learned Fence Pointers. 最近的工作试图通过学习索引[24]来减少围栏指针的内存I/O开销，方法是根据数据的密钥分布推断一个条目在运行中的位置。这样的工作可以通过寻址围栏指针来补充Chucky，一旦Chucky被应用，围栏指针将成为下一个内存I/O瓶颈。

Learning from Negative Queries. 最近设计的过滤方法是从通常发出的否定查询（到不存在的密钥）中学习，以降低误报率[32,65,78]。将这些技术与Chucky相结合是一个有趣的方向。

Bloom Filters (BF). 已经提出了许多BF变体[16,72,93]，它们支持计数[13,40,89]、压缩性[77]、向量化[83]、删除部分但非全部条目[88]、高效哈希[33,60]和缓存局部性[17,31,66,67,84]。Bloomier过滤器允许将值与密钥关联，但无法压缩值，并且比指纹过滤器更复杂[19,20]。

Entropy Coding. 除了哈夫曼编码，还有其他基于组成符号概率分布的字母压缩方法：算术编码[82,87]和非对称数字系统[36]。这些方法不需要使用辅助结构来编码或解码符号。利用这些技术消除Chucky的辅助结构（即哈夫曼树、解码表和重编码表）是一个有趣的未来方向。

7. 结论

本文表明，随着SSD和内存设备之间性能差距的缩小，LSM树的Bloom过滤器正在成为内存访问瓶颈。因此，我们提出了Chucky，一个用于LSM树的过滤器，它比Bloom过滤器需要更少的内存I/O来查询和维护。Chucky使用内存中的布谷鸟过滤器将所有条目映射到它们在LSMtree中的位置，并压缩这些位置信息以保持低误报率和稳定。因此，Chucky实现了世界上最好的：更少的内存I/O和低且稳定的误报率，所有这些都是为了相同的内存预算。

8. 致谢

We thank the anonymous Reviewers for their invaluable feedback.

We also thank the entire Pliops team for facilitating this work.