****

**研究生（数据中心技术）课程论文（报告）**

**题 目：ElasticBF: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large Key-Value Stores**

**学 号 M201973011**

**姓 名 孙卓航**

**专 业 计算机技术**

**课程指导教师 曾令仿、施展**

**院（系、所） 光电国家研究中心**

**2019年11月8日**

# elasticbf：用于提高大键值存储中的读取性能，具有热感知的动态布隆过滤器

国光硕1905班 M201973011 孙卓航

## 1论文背景

随着数据量的指数增长，传统关系数据库在处理超大规模数据时面临着可扩展性的挑战。替代方法是，在云系统、广告、社交网络、搜索索引、在线游戏等应用中，键值存储被广泛用作基本的存储基础设施。根据使用的索引结构，键值存储可以分为基于散列索引的设计，基于b树的设计和基于lsm树的设计。因为基于散列索引的设计需要很大的内存，不能很好地支持范围查询，而基于b树的设计涉及大量的随机写入，所以大多数现代的键值存储都使用lsm树。

基于lsm树的键值存储通常由两个组件组成，并以leveldb为例进行说明。一个驻留在内存中以缓存键值对，它包括一个内存表和一个不可变的内存表。另一个存储在辅助存储器中，辅助存储器分为多个级别，由多个sstable组成。当一个级别达到其大小限制时，它的sstable将通过合并压缩到下一个级别，压缩会导致严重的写入放大。

另一方面，基于lsm树的键值存储也会遭受严重的读取放大。这是因为当查找一个键值对时，键值存储需要检查从最低级别到最高级别的多个sstable，直到找到密钥或检查了所有级别。此外，需要读取多个元数据块才能真正检查一个sstable中是否存在键值对。

为了减少检查多个sstable导致的额外I/O，现代设计在键值存储中使用布隆过滤器来快速检查单个sstable中是否存在键值对。然而布隆过滤器会出现假阳性，因此即使sstable中不存在键值对，它们也可能返回阳性结果，这会导致不必要的I/O。在实际应用中，数据访问通常还会出现偏差，因此，本文提出了一种细粒度动态布隆过滤器管理方案elasticbf，通过动态调度来减少布隆过滤器的假阳性，从而减少额外I/O。

## 2 研究现状

近年来，许多研究提出了基于lsm树的新设计。WiscKey [1]通过使用键值分离技术来管理lsm树中的键和元数据，同时将值存储到附加的日志中，减少了压缩I/O。 HashKV [2]通过使用基于散列的数据组织进一步优化了基于键值分离设计的值管理。 LSM-trie [3]专注于较小的键值对，并将数据组织为基于散列的字典树结构以减少写放大。 bLSM [4]使用新的合并调度程序来减少压缩对前端写入性能的影响，并且还使用Bloom过滤器来帮助有效查找。TRIAD [5]通过利用偏斜的数据热度和延迟的压缩策略来减少写入I/O。 PebblesDB [6]根据跳表的启发重新组织存储布局，从而避免在同一级别重写数据，以减少压缩开销。

其他一些研究旨在更好地利用新兴存储设备的功能来提高键值存储的性能。 例如，RocksDB [7]通过同时调度多个压缩操作来利用SSD的并行性。 LOCS [8]利用SSD的多通道来利用丰富的并行性来进行有效的压缩和数据访问。 NVMKV [9]通过在物理地址空间中映射KV对来与FTL合作，以减少存储层和设备层之间的冗余工作。 HiKV [10]还通过使用混合索引来利用NVRAM。

最后，还有一些考虑布隆过滤器优化的工作。尤其是，RocksDB [11]使用前缀布隆过滤器来减少范围查询的读取放大。 SuRF [12]基于简洁的数据结构，通过过滤点查询和范围查询的请求来减少I/O。异构布隆过滤器设计也被认为可以针对不同级别或文件配置不同的布隆过滤器。但是，Monkey [13]采用了一种粗粒度方案，该方案为同一级别内的SSTable分配了相同数量的过滤器，并且也无法根据热度进行动态调整。

## 3 系统设计

图1描述了elasticbf的架构，它主要包含三个组件：细粒度布隆过滤器分配、热识别和继承以及内存中的布隆过滤器管理。在细粒度布隆过滤器分配的设计中，我们主要面对的问题是每个sstable应该分配多少个布隆过滤器，每个过滤器应该分配多少位，以实现低误报率和低内存使用率。我们还需要仔细设计低I/O开销的数据结构和管理方案。对于热度识别，我们的目标是以较低的开销实现相对准确的热度估计。最后，热继承的目的是避免压缩后热识别的冷启动，内存中的布隆过滤器管理是根据热度有效地调整布隆过滤器。

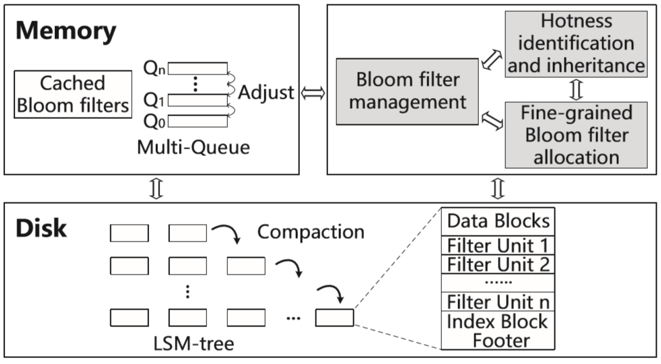


图1 elasticbf的结构

### 3.1 细粒度布隆过滤器

**（1）多数量布隆过滤器**

elasticbf使用不同的独立散列函数为每个sstable生成多个布隆过滤器。每一个过滤器分配给每个密钥的比特数更少，我们称之为过滤器单元（ﬁlter unit）。分配给SSTable的所有过滤器单元都被命名为过滤器组，如图 2所示。如果启用了多个过滤单元，那么只有当所有启用的过滤单元都指示存在密钥时，我们才需要读取sstable来搜索密钥。

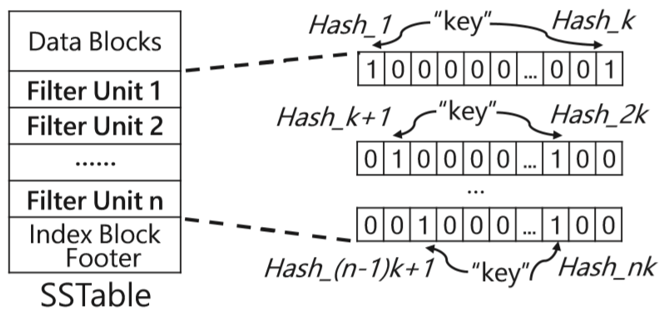


图 2 多数量布隆过滤器的结构

由于每个过滤器单元是独立的，所以整个过滤器组的假阳性率和相同大小的单个布隆过滤器完全相同。因此，可以对此进行动态调度，例如需要对最热的SSTable启用一组中的所有单元，使得假阳性率接近于0。与静态设置布隆过滤器相比，这种细粒度的方式能减少55.9%-89.7%的由不同工作负载下的误报引起的I/O数量。

**（2）分段的细粒度设计**

在sstable中，访问不均匀仍然很严重。因此，我们可以通过区分同一sstable中密钥的热度来进一步降低假阳性率。为了平衡测量热度的准确性和对键值存储的额外开销，我们进一步将每个sstable划分为多个称为段的区域，并在段的粒度处记录热度。然后为每个段分配一组过滤单元。

此方法仍是通过细粒度划分的方式，为不同的访问热度区域提供不同大小的布隆过滤器。挑战性的问题是优化段的尺寸，因为大段不能准确反映SStable中不同的KV对的热度，而小段会给KV商店带来很大的开销。我们确定段大小的规则是使每个bloom filter的大小接近设备块大小，例如4KB，以便在加载bloom filter时减少I/O开销。

### 3.2热识别和遗传

**（1）热识别**

段的热度由其访问频率和自上次访问以来的持续时间决定。具体来说，我们提出了一个过期政策，以区分热/冷段。我们维护一个名为currenttime的全局变量，它被定义为到目前为止向整个键值存储发出的get请求的总数，并且我们还将一个名为expiredtime的变量与每个段相关联，以表示该段将“过期”的时间点。确切地说，expiredtime被定义为lastaccessedtime+lifetime，其中lastaccessedtime表示最近访问段的时间，lifetime是一个固定的常量。注意，这里的“时间”概念是指逻辑时间，它实际上由访问次数表示。每次访问一个段时，我们都会将currentTime增加一个，并通过将lastaccessedTime设置为currentTime的更新值来更新该段的expiredTime。如果currentTime已经大于expiredTime，我们将段定义为“expired”。

上述策略的含义是，如果在按生存期定义的固定数量的GET请求期间未访问某个段，则该段将过期并视为冷段。我们可以知道该操作的时间复杂度仅为O(1)，且内存开销也很小，因为只需要记录每个段的expiredTime。

**（2）热度遗传**

压缩将触发sstables之间的合并排序以生成新的sstables。因此，新的sstables中的段也是新生成的，它们的热度应该改变。如果将新段的热度设置为0，则elasticbf将面临热度的冷启动，这可能会降低将来从新段读取数据的性能。为了继承热度，理想情况下，我们可以根据其中所有键的热度准确估计新段的热度，但这会给键值存储带来太大的开销。

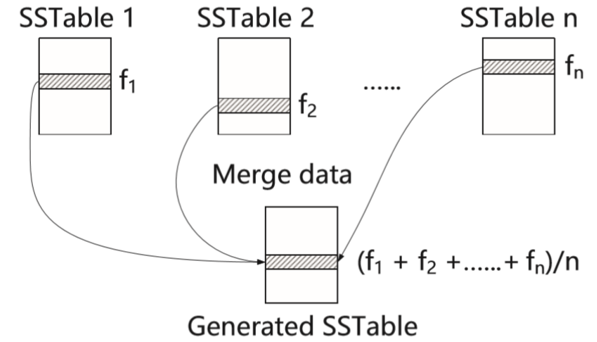


图 3 压缩后的热度遗传

elasticbf使用旧段的热度来估计新段的热度。具体来说，如图 3所示，当生成一个新段时，我们首先找出参与生成新段的过程的旧段，并且具有重叠的关键范围，然后简单地使用所有旧段的热度平均值来估计新段的热度值。最后，我们相应地为新段启用了一些过滤单元。这种简单的方案不会有太大的开销。

### 3.3多队列动态调度

**（1）布隆过滤器调度规则**

最后还需要确定每个段应该启用多少个过滤单元。使用一个定义为误报导致的预期I/O数量的指标来指导调整，我们将这个额外I/O数量表示为E[Extra IO]，可以表示为：



其中m表示键值存储中的段总数，fi表示段i的访问频率，ri表示误报率，由段i的存储器中加载的过滤器单元数计算得出。每次访问一个段时，我们都会更新它的访问频率和E[Extra IO]，然后检查如果我们为这个段启用一个以上的过滤单元，并为其他段禁用一个单元以保证相同的内存使用量，是否可以减少E[Extra IO]。如果E[Extra IO]可以减少，那么我们应用调整，否则，我们什么也不做。

**（2）多队列动态调度**

上述调度策略的关键问题是确定应该禁用哪个过滤单元，我们通过维护基于多队列的内存索引来解决这个问题。

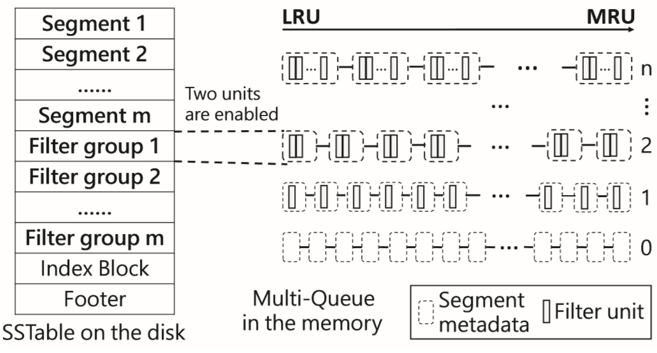


图 4 内存中elasticbf的多队列

elasticbf维护多个最近使用最少（LRU）的内存队列来管理每个段的元数据，如图 4所示。我们将这些队列表示为q0，…，qn，其中n等于分配给每个段的过滤器单元的最大数量。队列的每个元素对应于一个段，它管理该段启用的过滤单元。准确地说，队列qi中的每个元素都表示为相应的段启用了过滤单元，即只有这些过滤单元用于检查密钥的存在。为了保持每个队列的LRU特性，每次访问一个段时，我们都将相应的元素移动到同一队列中的MRU端（最远端）。

为了确定哪些过滤单元应该被禁用，然后从内存中删除，我们使用上述的冷热段信息。具体来说，我们搜索从qn到q1的冷段，对于每个队列，我们从LRU端搜索到MRU端，因为冷段必须是最近使用最少的一个。当我们发现一个冷段并且禁用该片段的一个过滤单元时，如果E[Extra IO]可以减少，那么我们将其降级到下一个较低级别队列以释放一个过滤单元。冷段的访问频率没有变化，而通过调整相应段中的过滤器单元，由于假阳性率的变化，E[Extra IO]可能会降低。如果没有冷段，我们跳过这次调整，这是一个保守的策略，由于调整频率的限制，调整开销是有限的。另一方面，我们将生存期设置为与段总数相同的数量级。理由是，如果没有冷段，则意味着几乎所有的段都是最近访问过的，因此在这段时间内它们可能具有相似的热度，我们不需要进行调整。

## 4评估

实验在SSD上运行，使用ycsb测试，在leveldb、rocksdb和pebblesdb之上构建elasticbf，SSTable大小都设置为64M，只对bloom filters的管理策略进行了相应的更改，而内存使用限制为相同。每个kv对的大小设置为1kb，并使用随机生成的不同密钥加载100gb数据库。生成的负载遵循zipf分布， zipfian常数为0.99。默认情况下，假设一半的get操作请求不存在的项（即零查找）。实验禁用块缓存以最小化缓存的影响。

### 4.1各种工作负载的比较

**（1）只读工作负载**

图 5（a）-（c）显示了读取吞吐量、平均读取延迟和I/O总数的结果。在leveldb、rocksdb和pebblesdb下，与不使用elasticbf的结果相比，使用elasticbf的读取吞吐量分别增加到2.08、2.15、和2.17倍。对于平均读取延迟，elasticbf可以将leveldb、rocksdb和pebblesdb的延迟分别降低51.9%、54.0%和55.8%。elasticbf的改进主要是由于布隆过滤器误报导致额外I/O的减少。如图 5（c）所示。我们可以看到，elasticbf将在不同的键值存储下的I/O数量减少了59.1%-63.8%。

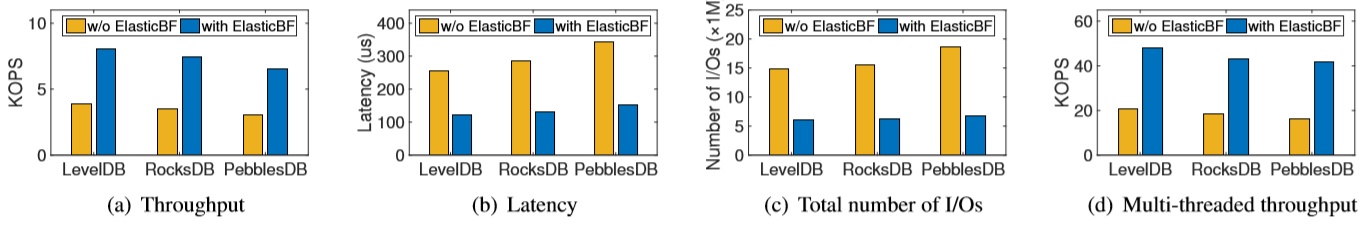


图 5 elasticbf在只读负载下的表现

实验进一步研究了elasticbf的并发读取性能，使用16个线程运行ycsb基准测试，每个线程执行1M get请求。结果如图 5（d）所示，elasticbf将这些键值存储中的读吞吐量增加到2.34、2.58倍。

**（2）混合工作负载**

图 6（a）-（b）显示了工作负载下50%读取和50%写入的结果，图 6（c）-（d）显示了工作负载下90%读取和10%写入的结果。工作负载中的请求总数为1000万，可以看到，在读取率为50%的工作负载下，elasticbf可以帮助leveldb、rocksdb和pebblesdb分别减少48.2%、28.4%和54.8%的读取延迟，在读取率为90%的工作负载下，相应的减少率分别为51.8%、38.9%和48.8%。

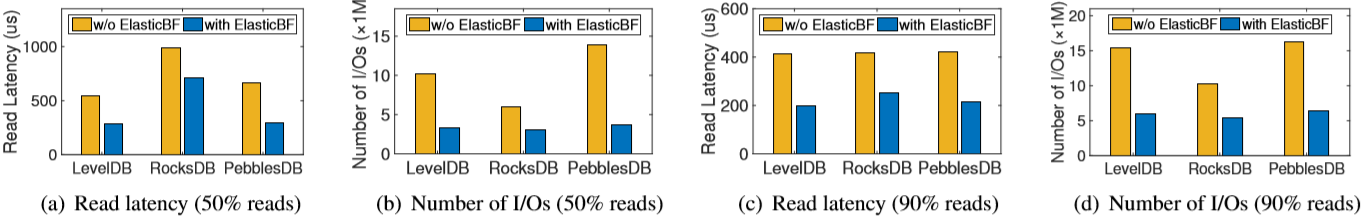


图 6 elasticbf在混合负载下的表现

**（3）写入和范围查询负载**

对于不同的键值存储，首先随机加载100GB数据库，然后发出10M扫描请求。比较加载数据库和执行扫描请求的时间，以评估写入和范围查询性能，结果如图 7所示。可以看到，写入和范围查询的性能也几乎保持不变。主要原因是在sstables中，布隆过滤器被组织成块，elasticbf还使用多线程来加速布隆过滤器的生成。对于范围查询，由于它需要获取与给定范围重叠的所有块，因此布隆过滤器不参与此过程。因此，elasticbf对写入和范围查询性能的影响可以忽略不计。

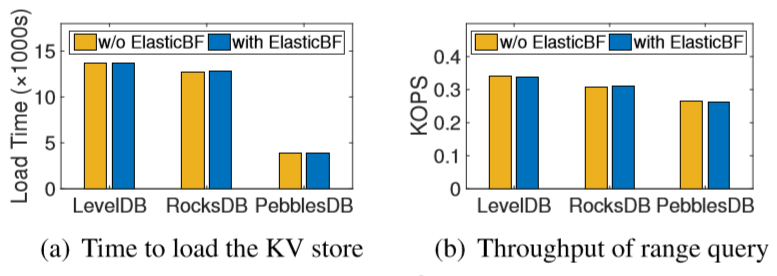


图 7 写入和范围查询表现

### 4.2不同参数设置的影响

论文只显示了leveldb的性能改进，而rocksdb和pebblesdb的性能改进也是类似的。

在机械硬盘上elasticbf的吞吐量提高了2.31-2.38倍，也就是说，elasticbf可以提高所有主流存储设备（包括ssd和hdd）的键值存储的读取性能。

当零查找比从0%增加到100%时，elasticbf将读取吞吐量增加到1.65-3.08倍。此外，我们可以看到，随着零查找比的增加，elasticbf得到了较大的改进。这是因为查找不存在的密钥会导致在键值存储中搜索所有级别，因此需要检查更多的sstables。

块缓存越大，读取性能越好。此外，elasticbf在启用cache时仍能提高性能。

当我们将键值对大小从1KB更改为128B时，elasticbf将吞吐量增加到1.99-2.08倍。当我们将数据库大小从100GB更改为400GB时，elasticbf将吞吐量提高到2.00-2.08倍。

实验还考虑了段大小的影响，SSTable大小为64MB，如果段大小也设置为64MB，则意味着我们以SSTable为单位测量热度并调整布隆过滤器。结果表明，在4MB设置下，改进幅度最大，4MB设置下的吞吐量比64MB设置下高17.8%。原因在于，随着段尺寸的减小，elasticbf可以进行更细粒度的热识别，因此它可以通过调整布隆过滤器获得更多的好处。

总结评估发现，elasticbf可以有效地提高不同存储介质和数据库规模下的不同键值存储的读取性能，但它主要集中在内存受限的环境中。也就是说，如果内存容量不是瓶颈，那么可以简单地为每个布隆过滤器分配更多的位并将它们全部保存在内存中，在这种情况下，假阳性率可能非常小，elasticbf的好处是有限的。

## 5总结

在本文中，作者通过利用工作负载中的访问不均匀，开发了一种名为elasticbf的细粒度异构布隆过滤器管理方案。 elasticbf用一种轻量级的方法来测量热度信息，并且还支持以精细的粒度动态调整布隆过滤器，同时可以和其他优化lsm树型键值存储结构的工作相结合来进一步提高性能。 结果，elasticbf可以大大降低预期的总体误报率，而不会增加布隆过滤器的数量和内存开销，从而提高了键值存储中的读取性能。 最后，作者还进行了广泛的实验，通过在各种键值存储的顶部构建elasticbf来证明其效率。

## 6参考文献

1. Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-Conscious Storage. In FAST, pages 133–148, 2016.
2. Helen HW Chan, Yongkun Li, Patrick PC Lee, and Yinlong Xu. HashKV: Enabling Efﬁcient Updates in KV Storage via Hashing. In 2018 USENIX Annual Technical Conference (USENIX ATC 18). USENIX Association, 2018.
3. Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data. In USENIX ATC 15, pages 71–82. USENIX Association, 2015.
4. Russell Sears and Raghu Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. ACM, 2012.
5. Oana Maria Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In USENIX ATC 17, 2017.
6. Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In Proceedings of the 26th Symposium on Operating Systems Principles, pages 497–514. ACM, 2017.
7. Facebook. RocksDB. <http://rocksdb.org/>.
8. Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An Efﬁcient Design and Implementation of LSM-tree based Key-Value Store on Open-Channel SSD. In Proceedings of the Ninth European Conference on Computer Systems, page 16. ACM, 2014.
9. Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. NVMKV: A Scalable and Lightweight Flash Aware Key-Value Store. In HotStorage, 2014.
10. Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A Hybrid Index Key-Value Store for DRAMNVM Memory Systems. In 2017 USENIX Annual Technical Conference (USENIX ATC 17), pages 349–362, 2017.
11. Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, and Tony Savor. Optimizing Space Ampliﬁcation in RocksDB.
12. Huanchen Zhang, Hyeontaek Lim,Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In Proceedings of the 2018 International Conference on Management of Data, pages 323–336. ACM, 2018.
13. Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal Navigable Key-Value Store. In Proceedings of the 2017 ACM International Conference on Management of Data. ACM, 2017.