计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.191200109

基于智能放置策略的 Cuckoo 哈希表

金 琪 王俊昌 付 雄
南京邮电大学计算机学院 南京 210003
(524983744@qq.com)

摘 要 由于查询时间复杂度为 O(1),Cuckoo 哈希表在大数据、云计算等领域得到了广泛应用。然而,现有 Cuckoo 哈希表的写入操作在遇到写冲突时普遍采用随机替换策略来替换已有表项。一方面,写入操作容易出现高迟插入和无限循环,尤其是当哈希表负载率较高时,甚至有重构整个哈希表的风险;另一方面,由于现有随机替换策略将数据项尽量散布在哈希表的各个桶中,哈希表项间缺乏良好的空间局部性,降低了数据正向查询的效率。为解决以上问题,提出了一种基于智能放置策略的 Cuckoo 哈希表。具体地,为提升写入操作的效率,提出了一种基于负载均衡的 Cuckoo 哈希表(Load-Balance Cuckoo Hash Table,LBCHT),实时限制每个桶的负载,并使用广度优先搜索寻找最佳 Cuckoo 路径,实验结果表明 LBCHT 能有效减少高负载率下写入操作可能出现的长尾效应;为提升查询操作的效率,提出了一种充分利用局部性原理的 Cuckoo 哈希表(Locality Principle Cuckoo Hash Table,LPCHT),通过充分发掘哈希表项间的空间局部性,来有效减小查询操作引起的 CPU 高速缓存缺失率,提高正向查询的效率。实验结果证明,在高负载率的压力测试环境中,与 libcuckoo 相比,LBCHT 的写入效率提升了 50%,LPCHT 的正向查询效率提升了 7%。

关键词:负载均衡;广度优先搜索;长尾效应;局部性原理;正向查询
中图法分类号 TN393

Cuckoo Hash Table Based on Smart Placement Strategy

JIN Qi,WANG Jun-chang and FU Xiong
School of Computer,Nanjing University of Posts and Telecommunications,Nanjing 210003,China

Abstract Because time overhead of query is O(1),Cuckoo hash table has been widely used in big data,cloud computing and other fields. However,insertion of the existing Cuckoo hash tables generally uses random replacement strategy to replace existing entries when encountering conflicts. On the one hand,writers are prone to high-latency insertion and infinite loops,especially when the load rate of hash table is relatively high,and even have the risk of reconstructing entire hash table;on the other hand,due to existing random replacement strategies,entries are scattered as much as possible in the buckets of hash table. The lack of good spatial locality among entries reduces the efficiency of positive query. To solve the above problems,this paper proposes a Cuckoo hash table based on smart placement strategy. Specifically,in order to improve efficiency of insertion,a load-balance Cuckoo hash table (LBCHT) is proposed,which limits the load of each bucket in real time,using breadth-first search to find the best Cuckoo path to ensures load balancing among buckets. Experimental results show that LBCHT can effectively reduce long tail effect of insertion under high load rate. And in order to improve efficiency of lookup,a Cuckoo hash table that makes full use of the locality principle is proposed (LPCHT). By fully exploring the spatial Locality among entries,CPU cache miss rate caused by lookup is reduced and efficiency of positive query is improved. Experiments show that in a high load rate stress test environment,compared with libcuckoo,LBCHT throughput of insertion can be increased by 50%,and LPCHT improves positive query efficiency by 7%.

Keywords Load balance,Breadth first search,Long tail effect,Locality principle,Positive query

1 引言

大数据时代,云计算服务器需要及时、准确地处理并分析大量数据,因此高效的查询模块对于存储系统非常重要。2018 年 9 月,Facebook 手机日活跃用户达 14.9 亿,同比增长 9%,每分钟就有 300 个新的个人资料被创建,超过 20.8 万张

到稿日期:2019-12-17 返修日期:2020-05-03 本文已加入开放科学计划(OSID),请扫描上方二维码获取补充信息。
基金项目:国家自然科学基金(61602264);江苏省重点研发计划(BE2017743)
This work was supported by the National Natural Science Foundation of China (61602264) and Key Research & Development Plan of Jiangsu Province (BE2017743).
通信作者:王俊昌(wangjc@njupt.edu.cn)

照片被上传到 Facebook[1]。数据量的激增给存储系统带来了很大的挑战，尤其是查询服务的支持[2]。

为了实现高效的查询，基于哈希的数据结构由于具有恒定的查询时间复杂度而被广泛地应用于索引的构建。传统处理哈希冲突的技术包括开放寻址[3]、链接[4]和合并哈希[5]。与传统哈希表不同，Cuckoo 哈希通过简单的“踢出”操作来解决哈希冲突，数据项插入期间在哈希表中移动数据项，而不是搜索链表。哈希的体系结构[6]已经证明，随着负载率的增加，Cuckoo 哈希表的吞吐量比链接哈希表高得多。

在实践中，由于哈希函数的本质属性，Cuckoo 哈希表并不能完全避免哈希冲突，现有处理冲突的工作主要是利用随机替换策略[7]，其本质上是深度优先搜索。由于一般的 Cuckoo 哈希使用的是不可预知的随机替换，为了提升哈希表的负载率，Cuckoo 路径可能会非常长，例如在 MemC3 中“踢出”操作的阈值为 500[8]。由于读取操作必须等待同一存储桶上写入操作完成，因此长 Cuckoo 路径导致哈希表的效率低下。为了提供高吞吐量和并发友好的 Cuckoo 哈希表，我们面临以下两个主要问题：

(1)写入性能差。数据项写入 Cuckoo 哈希表时，如果发生哈希冲突，则将候选桶中任意一个数据项迁移到该数据项的其他候选桶中，并且可能迁移另一个数据项，直到找到一个空槽或者达到最大搜索阈值[9-11]。本质上，深度优先搜索的随机替换策略导致 Cuckoo 哈希的 Cuckoo 路径很长，频繁的“踢出”操作导致其在多个桶之间进行密集的数据迁移，使得哈希表写入性能低下，同时细粒度锁的大量使用存在死锁和活锁的风险。基于随机替换策略的 Cuckoo 哈希表的桶之间的负载不均衡，导致在高负载率下易出现高延迟插入和无限循环。因此，我们需要避免大量数据迁移，减少甚至消除高负载率下写入操作可能出现的长尾效应。

(2)正向查询性能差。Cuckoo 哈希表通常使用 $D(D>1)$ 个散列函数[8,12-14]，查询时最少只需要访问 1 个桶就能得到查询结果，最多时也只需要访问 D 个桶。在读负载远大于写负载的情况下，哈希表的吞吐量取决于读负载的吞吐量。因此，通过减少读取时访问缓存行的数量，尤其是正向查询时访问缓存行的数量，可以提升查询效率。

本文用到的参数如下：

(1)负载率 LF，其表示 Cuckoo 哈希表当前存储的数据量 $Counter$ 与哈希表长 L、桶的大小 B 的比率。

$$LF=\frac{Counter}{L*B} \tag{1}$$

(2)额外负载因子 a。根据负载率 LF 和桶的大小 B 可以限制当前每个桶可存储的数据量 $Limit$，额外负载因子 a 表示负载率的参数，用于调整每个桶负载的限制。

$$Limit=(LF+\alpha)*B+1 \tag{2}$$

为了应对写负载大于读负载的情况，并提升写入效率，本文提出了一种基于负载均衡的 Cuckoo 哈希表(LBCHT)，用于缓解甚至消除写入过程中出现的高延迟插入和无限循环问题。LBCHT 提供两个散列函数，每个桶有 $B(B=2,4,8,16)$ 个可用于存储数据项的槽。哈希表分配一个计数器，用于实时计算哈希表的负载率，并根据负载率限制当前每个桶的负载，同时额外负载因子 a 可用于调整每个桶的限制负载。LBCHT 使用广度优先搜索代替随机替换策略，以找到符合限制的桶，减少数据项的迁移次数，同时使得各个桶负载均衡，抑制甚至消除长尾效应，提升写入效率。

为了应对读负载大于写负载的情况，并提升查询效率，本文提出了一种充分利用局部性原理的 Cuckoo 哈希表(LPCHT)，利用 Cuckoo 哈希表具有多个散列函数的特性，将哈希表中的数据项集中存储到编号较小的桶中，充分发掘哈希表项间的空间局部性，以有效减少查询引起的 CPU 高速缓存缺失率，提高正向查询的效率。为了避免桶负载不均衡可能引起的高延迟插入和无限循环，使用广度优先搜索找到符合条件的桶，利用局部性原理的同时使得各个桶的负载相对均衡。

实验证明，当额外负载因子 a 为 15%～30%时，相比 libcuckoo[15]，LBCHT 的写入效率提升了 28%，哈希表的负载率为 90%时，LBCHT 的写入效率最多可提升 50%，有效地减弱了长尾效应的影响。当哈希表的负载率为 20%～50%时，LPCHT 正向查询的效率提升了 7%。

本文第 2 节介绍了哈希表的相关工作；第 3 节描述了 LBCHT 的总体设计和实现细节；第 4 节描述了 LPCHT 的总体设计和实现细节；第 5 节对实验进行评估；最后总结全文。

2　相关工作

2.1　Cuckoo 哈希表及其改进

Cuckoo 哈希[14]是一种开放寻址技术，其查询、插入和删除操作的时间复杂度均为常数。为了减少哈希冲突，通常将数据项存储到哈希表两个候选桶中的任意一个，如果从其中一个候选桶中“踢出”数据项，那么将该项迁移到另一个候选桶。频繁地进行“踢出”操作可以帮助找到插入位置，但会带来额外的延迟。另外，查询数据时，读操作最多只需要读取两个缓存行，因此获得了恒定规模的查询时间复杂度。

假设 D 是散列函数的个数，S 是一组数据项。对于 $D=2$，常规 Cuckoo 哈希表 H 使用两个独立且均匀分布的散列函数 $h_1,h_2:S->\{0,1,\cdots,L-1\}$，其中 L 为哈希表的长度。如果数据项 x 写入成功，则 x 可以存储到 $h_1(x)$ 和 $h_2(x)$ 任意一个候选桶中，且只能存储在其中一个候选桶中。

实际应用中，通常对 Cuckoo 哈希表进行适当的改进，第一种方法是使用 $D(D>2)$ 个散列函数，第二种方法是使得每个桶的大小 B 大于 1(通常 $B=2,4,8,16$)。

大多数 Cuckoo 哈希表使用随机替换策略搜索空槽，如果数据项 x 的候选桶已满，则随机选择桶中的一个数据项 y 并将其迁移到候选桶中，这可能导致循环迁移数据项，直到找到一个空槽或者达到最大搜索阈值。这个过程中所有相关的桶都是 Cuckoo 路径的一部分。随着哈希表负载率的增加，Cuckoo 路径的长度也会增加，大量数据项的迁移导致哈希表的性能低下。Cuckoo 哈希表可看成一个无向图，每个桶为一个顶点，每个数据项为一条边，用以连接两个候选桶(顶点)。使用广度优先方法搜索空槽，桶中的每个数据项都被视为可能的路径，并以相同的方式将路径拓展到其他候选桶。广度优先搜索能够以对数的形式缩短深度优先搜索 Cuckoo

路径的长度，在缩短临界区长度的同时保证了哈希表的空间利用率。

2.2 高性能哈希表

GoogleSparseHash 库提供的 dense_hash_map[16] 支持快速查询服务。该哈希表使用二次内部探测的开放式寻址，具有极高的查询速度。dense_hash_map 将数据项存储到一个数组中，默认情况下最大的负载率为 50%。该哈希表以空间效率为代价实现了极高的吞吐量。C++11 中实现了 unordered_map，该哈希表是链接哈希表，具有非常快的查询速度，但其以使用更多的内存为代价。这些哈希表由于最多只允许一个读线程或者写线程，其性能并不随着机器中内核数量的增加而拓展。

Horton[17] 表以桶化 Cuckoo 哈希表(Bucketize Cuckoo Hash Table，BCHT)为基础，减少每次查询中读取 CPU 缓存行的数量，从而实现更高的吞吐量。哈希表中的大多数数据项使用主哈希函数，发生冲突的数据项使用辅助函数散列到其他位置，因此可以通过访问单个缓存行来进行检索。对于否定查找，数据项的重新映射可以降低访问的成本。

SmartCuckoo[13] 可以准确地预测“踢出”操作的状态和是否发生无限循环，其使用有向伪森林表示数据项的哈希关系，进一步避免了无限循环的发生。由于允许执行的线程数有限，哈希表的性能不会随着处理器中内核数的增加而拓展，这成为写入效率的瓶颈。

与上述哈希表不同的是，LBCHT 和 LPCHT 的设计目标是通过并发友好的哈希表来实现高效的读写操作。

Liu 等提出动态调整大小的非阻塞哈希表[18]，使用 Java 语言实现，并使用 java.util.concurrent 包进行优化。具有相同散列地址的数据项使用顺序存储，减少了读取数据时缓存的缺失率，提升了读取效率。其在执行数据项的插入或删除操作时，替换相同散列地址的数据集合，降低了内存的利用率。相对哈希表[19] 是支持拓展和缩小的，同时允许并发、无等待和线性查找的数据结构，Triplett 提出的调整哈希表大小的算法会随着数据项的减少而回收内存，并且随着数据项的增加使用读取-复制-更新(RCU)[20] 来维持恒定的性能，不会影响查询操作的效率。

MemC3[8] 实现了乐观的 Cuckoo 哈希表，该哈希表利用分离锁(lock striping)和原语实现多个读线程和单个写线程的并行。MemC3 应用的 Cuckoo 哈希不是沿着 Cuckoo 路径向前移动“数据项”，而是沿着 Cuckoo 路径向后移动“空槽”，从而保证数据永远存在于哈希表中。Li 等[21] 重新设计了 MemC3，以最大限度地减小临界区，从而显著地提高并行度，并使用两种并行控制机制、细粒度锁和硬件事务内存实现了 libcuckoo[15]。

这些方法提升了哈希表的吞吐量，当写负载大于读负载时，由于高延迟插入和无限循环的发生，降低了哈希表写入操作的效率；当读负载大于写负载时，散列函数将数据项尽量散布在哈希表的各个桶中，增加了正向查询时 CPU 高速缓存的缺失率，降低了读负载的效率。本文提出的 LBCHT 应用于写负载大于读负载的场景时，通过使每个桶负载均衡来缓解甚至消除写入过程中可能出现的长尾效应。LPCHT 应用于读负载大于写负载的场景时，充分发掘了哈希表项间的空间局部性，减小了正向查询时缓存的缺失率，提升了读负载的效率。

3 LBCHT 的设计与实现

我们观察到，在桶化 Cuckoo 哈希表(BCHT)中，通常每个桶实际的负载并不一致，并且随着哈希表负载率的增加，发生哈希冲突的概率越来越高。我们将频繁发生哈希冲突的桶称为“热桶”，较少发生哈希冲突的桶称为“冷桶”。基于这一特征，我们提出了一种基于负载均衡的 Cuckoo 哈希表(LBCHT)，通过哈希表负载率和额外负载因子 a 来实时限制每个桶可存储的数据量，使得各个桶的负载保持均衡，以此来减弱长尾效应对写负载性能的影响。

3.1 LBCHT 的核心思想

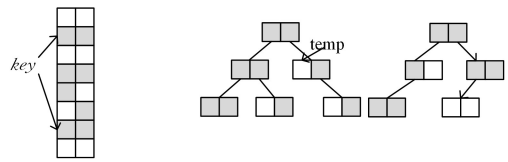
LBCHT 实时统计表中已存储的数据量并计算哈希表的负载率，函数 $Limit()$ 的值等于负载率和额外负载因子 a 均摊到每个桶的负载。插入数据项时，根据数据项的 key 和散列函数 h_1 和 h_2 映射哈希表的两个候选桶，如果其中一个候选桶的负载小于 $Limit()$，则将数据项写入到该桶中。如果两个候选桶都不满足要求，则使用广度优先搜索，使桶中的每个数据项连接两个候选桶，并以相同的方式拓展，直到找到符合要求的桶，生成 Cuckoo 路径并进行数据项的迁移和插入。

3.2 实际操作

本文提出的 LBCHT 支持数据项的查询、插入和删除这 3 种基本操作。

3.2.1 插入

LBCHT 提供了两个散列函数，假设桶的大小 $B=2$，即每个桶最多可存储两个数据项。插入数据项时，根据数据项的 key 和散列函数计算两个候选桶的编号，假设桶的编号为 i 和 j，如图 1(a)所示。首先检查待插入的数据项是否已经存储到哈希表中，如果存在，则插入失败。



(a)key 的散列　　(b)BFSCuckooSearch 过程

图 1 LBCHT 的 BFSCuckooSearch

Fig. 1 LBCHT's BFSCuckooSearch

LBCHT 的 CuckooSerach 算法如算法 1 所示。

算法 1 LBCHT 的 CuckooSearch 算法

```
Bucket LBCHTCuSearch(Bucket b1,Bucket b2)
begin
InitQueue q;
q.push(b1,b2);
Bucket temp;
temp.SetLoad(BucketSize);
int head=0;
while (head<SearchThreshold)
  bucket b=q.GetBucket(head);
  for (int i=0;i<BucketSize;i++)
    bucket new_b=b.RelatedBucket(i);
    if(new_b.Load()<Limit())
```

```
        return new_b;
      else
        if(new_b. Load() < temp. Load())
          temp=new_b;
        end if
        q. push(new_b);
      end if
    end for
    head++;
end while
return temp;
end
```

LBCHT 实时统计表中已存储的数据量并计算哈希表的负载率，额外负载因子 a 可视为哈希表中假设存在的负载，函数 *Limit*()的值等于实际负载率和额外负载因子 a 均摊到每个桶的负载，a 值越大，*Limit*()的返回值也越大，*Limit*()对每个桶的负载限制就越小。如果编号为 i 的桶的负载小于或等于编号 j 的桶，并且小于 *Limit*()的值，则直接将数据项写入编号为 i 的桶中，数据项的计数器 *Counter* 加 1，插入成功。同理，如果编号为 j 的桶的负载小于编号为 i 的桶，且小于 *Limit*()，那么直接将数据项插入到编号为 j 的桶中，*Counter* 加 1，插入成功。

如果上述情况都不满足，则使用广度优先搜索进行 Cuckoo 搜索，编号为 i 和 j 的桶中每个数据项连接两个候选桶，并以相同的方式进行拓展，如图 1(b)所示。依次访问每个桶并计算该桶的负载是否小于函数 *Limit*()的值，如果找到满足条件的桶，则返回生成 Cuckoo 路径，执行数据项的迁移和插入，并将 *Counter* 加 1，插入成功。如果不满足限制但是该桶中存在空槽，则使用 temp 标记该桶，避免在搜索阈值内找不到符合条件的桶从而导致插入失败。继续进行搜索，如果当前访问的桶的负载不满足函数 *Limit*()，但是小于 temp 标记的桶，则 temp 标记当前访问的桶，继续进行搜索，直到找到满足 *Limit*()限制的桶或者达到最大搜索阈值。当达到最大搜索阈值并且存在 temp 标记的桶，则生成 Cuckoo 路径，执行数据项的迁移和插入，并将 *Counter* 加 1，插入成功；否则插入失败。

图 1(a)显示了 *key* 映射到哈希表编号为 i 和 j 的桶，图 1(b)显示了以编号 i 和 j 的桶为根节点的 LBCHT 的 Cuckoo 搜索过程，此时 *Limit*()的值为 1，“－＞”表示最终生成的 Cuckoo 路径，temp 标记最接近条件的桶。

3.2.2　查询和删除

LBCHT 和 libcuckoo 的查询和删除操作基本一致。在查询和删除数据项之前，首先将该数据项的两个候选桶加锁，然后执行查询或者删除操作，使用细粒度锁可以一次完成查询或删除操作，避免桶中的数据在查询或删除期间发生变化而需要重试。删除数据项之后将计数器 *Counter* 减 1。

4　LPCHT 的设计与实现

我们观察到，在桶大小为 $B(B>1)$ 的 Cuckoo 哈希表的负载率不高的情况下，哈希表存储的数据项会尽量散布在表的各个桶中。实际应用中，Cuckoo 哈希表的读负载通常远大于写负载，这时哈希表的吞吐量取决于读负载的效率。然而，由于数据项散布在哈希表的各个桶中，使得正向查询时缓存行的缺失率较高。基于这一特性，本文提出了充分利用局部性原理的 Cuckoo 哈希表(LPCHT)，其利用多个散列函数，充分发掘哈希表项间的空间局部性，以提升正向查询的效率。

4.1　LPCHT 的核心思想

为了减小读操作引起的 CPU 高速缓存的缺失率，提升正向查询的效率，利用多个散列函数将数据项存储在编号较小的桶内，充分发掘哈希表项间的空间局部性。同时，为了减少整体上各个桶负载不均衡可能带来的问题，待插入数据项的两个候选桶保持负载均衡。如果直接将数据项写入到编号较小的桶导致负载失衡，则使用广度优先搜索，找到编号小于候选桶中较大的编号的桶(该桶的负载小于候选桶中较大的负载)进行数据项的插入和迁移。

4.2　实际操作

本文提出的 LPCHT 支持数据项的查询、插入和删除这 3 种基本操作。

4.2.1　插入

LPCHT 提供了两个散列函数，假设桶的大小 $B=2$，即每个桶最多可存储两个数据项。插入数据项时，根据数据项的 *key* 和散列函数计算出两个候选桶的编号，假设桶的编号为 i 和 j，且 $i<j$。首先检查待插入的数据项是否已经存在于哈希表中，如果已经存在，则插入失败。

计算两个候选桶的负载，如果编号为 i 的桶负载小于或等于编号为 j 的桶负载，那么直接将数据项写入到编号为 i 的桶中，否则使用广度优先搜索进行 Cuckoo 搜索，找到存在空槽的桶，如果该桶的负载小于待插入数据项候选桶中较大的负载，并且该桶的编号小于候选桶中较大的编号，则返回生成的 Cuckoo 路径，执行数据项的迁移和插入。

LPCHT 的 CuckooSearch 算法如算法 2 所示。

算法 2　LPCHT 的 CuckooSearch 算法

```
Bucket LPCHTCuSearch(Bucket b1,Bucket b2)
begin
InitQueue q;
q. push(b1,b2);
int head=0;
int maxLoad=MaxLoad (b1,b2);
int maxID=MaxID(b1,b2);
Bucket temp;
temp. SetID(maxID+1);
while(head < SearchThreshold)
  bucket b=q. GetBucket(head);
  for(int i=0;i < BucketSize;i++)
    bucket new_b=b. RelatedBucket(i);
    if(new_b. Load() < maxLoad && new_b. ID() < maxID)
        return new_b;
    else
    if(new_b. ID() < temp. ID() && new_b. Load() < Bucket-
    Size)
        temp=new_b;
    end if
      q. push(new_b);
    end if
```

```
    end for
    head++;
end while
return temp.
end
```

如果存在空槽的桶不能同时满足上述条件，则使用 temp 标记该桶，避免因为找不到符合要求的桶导致插入失败。继续进行 Cuckoo 搜索，如果找到存在空槽的桶不能同时满足上述条件，但是当前桶的编号小于当前 temp 标记的桶编号，则更换 temp 所标记的桶，继续进行 Cuckoo 搜索，直到找到符合条件的桶，或者达到最大搜索阈值。当达到最大搜索阈值时，如果 temp 标记了桶，则生成 Cuckoo 路径，并执行数据项的迁移和插入，否则插入失败。

4.2.2 查询和删除

LPCHT 和 libcuckoo 中的查询和删除操作一致。在查询和删除数据项 *key* 前，首先将该数据项的两个候选桶加锁，然后执行查询或者是删除操作，使用细粒度锁可以一次完成查询和删除操作，避免了桶中的数据在查询或删除期间发生变化而需要重试。

5 性能评价

我们在服务器上实现了基于负载均衡的 Cuckoo 哈希表和充分利用局部性原理的 Cuckoo 哈希表，该服务器 CPU 是 2 个 Intel(R) Xeon(R) CPU E5-2630 处理器，共有 12 个 CPU 核，频率为 2.60 GHz，支持超线程，LLC 的大小为 15 360 kB。

本次实验中，使用随机数生成数据项，进行哈希表的查询、插入和删除操作。根据 Li 等[21]的证明，广度优先搜索的深度设置为 5，哈希表桶的数量为 2^{20}，每个桶中槽的数量为 4，哈希表最多可存储 2^{22} 个数据项。

5.1 额外负载因子的影响

哈希表中当前负载率和额外负载因子 a 共同作用，实时限制每个桶的负载，a 越小限制就越强，当 $a=0$ 时，限制最强。随着 a 值增大，对高负载下长尾效应的抑制减弱；当 a 接近 0 时，虽然能够有效缓解高负载率下的长尾效应，但是增加了中低负载率下出现长尾效应的频率，整体上降低了写入的效率。使用 8 线程，统计在不同额外负载因子 a 的影响下，哈希表的负载率分别为 24%，49%和 90%时写入效率以及整个表的写入效率，结果如图 2 所示。

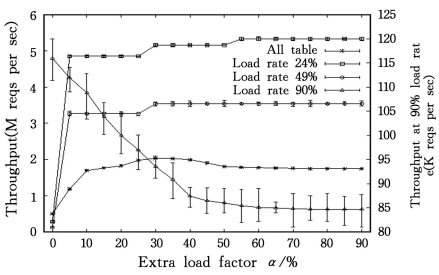


图 2 额外负载因子 a 对吞吐量的影响

Fig. 2 Impacts of extra load factor a on throughput

随着额外负载因子 a 逐渐增大，高负载率下写负载的效率逐渐降低，a 的增大导致 *Limit*() 对桶负载的限制降低，桶之间的负载均衡性也降低，减弱了对长尾效应的抑制。a 的增大导致整个表的写负载效率先上升后降低，当 $a=0$ 时，桶之间的负载均衡性达到最佳，但是会极大地增加中低负载率下插入数据项出现长尾效应的频率。例如，当哈希表的负载率为 24%时，额外负载因子 $a=0$，函数 *Limit*() 的值为 1，为了找到已存储数据量小于 *Limit*() 的桶，会出现长 Cuckoo 路径，急剧降低了写入效率，由于 a 对 *Limit*() 的影响是阶梯式的，因此写负载效率也是阶梯式地变化。随着 a 的增加，哈希表的写入效率增加，*Limit*() 对桶负载的限制逐渐降低，使得中低负载率下写入效率激增；当 a 超过 30%之后，写入效率逐渐降低，*Limit*() 对桶负载限制的降低导致在高负载率下的写入过程中易出现长尾效应，降低了写负载的效率。当 a 超过 50%之后，无论是高负载率下的写入效率还是整体上的写入效率不变，*Limit*() 对哈希表写入操作基本没有影响。

5.2 长尾效应的比较

为了验证 LBCHT 能够有效抑制高负载率下 Cuckoo 哈希写操作出现的长尾效应，在负载率为 90%的哈希表中，使用 8 线程，总共插入 1 000 组数据，统计插入每组数据所需的时间，同时对比 libcuckoo 中的算法，累计概率分布结果如图 3 所示，横坐标为时间，纵坐标为比率。

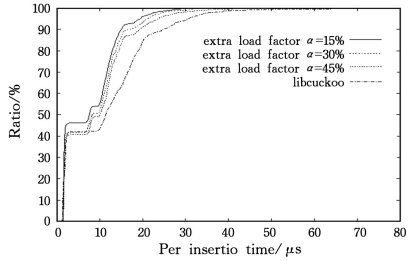


图 3 累积概率分布

Fig. 3 Cumulative probability distribution

如图 3 所示，在 $a=15\%$ 的 LBCHT 中，几乎 100%的数据项可以在 3 μs 内完成插入，而在 libcuckoo 中，大约 7%的数据项的写入时间超过 3 μs。综合而言，在 90%的负载率下，$a=15\%$ 的 LBCHT 写负载效率相比 libcuckoo 提升了 50%。随着 a 的增大，高负载率下的写入效率逐渐降低，由于数据项的两个候选桶负载均衡，LBCHT 仍然能有效缓解长尾效应。

5.3 libcuckoo 与不同额外负载因子的 LBCHT 的比较

如表 1 所列，额外负载因子 a 对哈希表最大负载率和吞吐量产生影响。随着 a 的增大，哈希表最大负载率逐渐降低。当 $a=30\%$ 时，LBCHT 获得最高的写入效率，相比 libcuckoo，写入效率提升了 28%。高负载率下写入效率随着 a 的增加而降低，当 $a=30\%$ 时，LBCHT 的写入效率提升了 35%。

表 1 算法的性能比较

Table 1 Comparison of algorithms performance

Algorithms	LBCHT $a=15\%$	LBCHT $a=20\%$	LBCHT $a=25\%$	LBCHT $a=30\%$	libcuckoo
Max load/%	96.53	96.52	96.50	96.45	95.96
Throughput (M/s)	1.77	1.83	1.98	2.04	1.59
Throughput with 90% load (K/s)	103.92	100.00	96.96	93.60	69.28

5.4 不同线程数之间写负载效率的对比

统计在不同写线程数下，向 $a=20\%$ 的 LBCHT 和 lib-

cuckoo 写入数量是其大小 95%的数据项时的写入效率，结果如图 4 所示。LBCHT 的写入效率相比 libcuckoo 有不同程度的提升，当线程数量为 8 时，吞吐量提升了 15%。

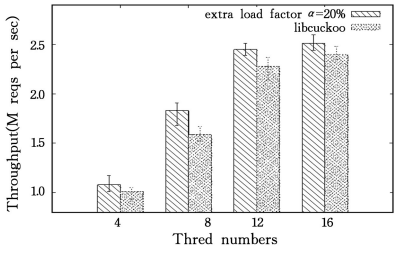


图 4　不同写线程数的写入性能

Fig. 4　Writers' performance under different writer threads

图 5 显示了不同写线程数下，负载率为 90%的 libcuckoo 和 $a=15\%$的 LBCHT 的写入效率。在不同线程数下，相比 libcuckoo，LBCHT 提升了写入效率，当线程数为 8 时，写入效率提升 50%。

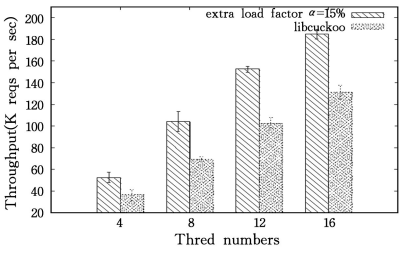


图 5　90%负载率下不同写线程数的写入性能

Fig. 5　Writes' performance under different writer threads with 90% load

5.5　正向查询效率的比较

为了验证本文提出的 LPCHT 在正向查询方面性能的提升，我们将 Cuckoo 哈希表桶的数量设置为 2^{17}，桶的大小 $B=4$，哈希表最多可以存储 2^{19} 组数据。测试程序使用了 10 线程，90%用于查询，5%用于插入和 5%用于删除。

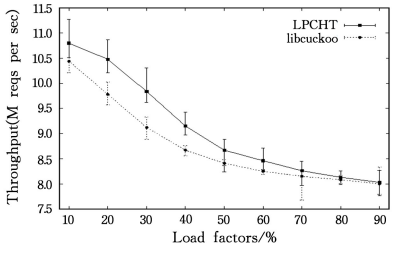


图 6　正向查询的吞吐量

Fig. 6　Throughput for positive queries

统计不同负载率下 LPCHT 和 libcuckoo 正向查询吞吐量的变化，结果如图 6 所示。可以看出，吞吐量随着负载率的上升有下降的趋势。假设数据项使用两个哈希函数的概率是相等的，每个哈希函数映射到每个桶的概率也是相等的，则读命中平均需要 1.5 次读缓存行，不命中需要 2 次。本文算法的实现使用了细粒度锁，当负载率增加时发生数据项迁移的概率也会提升，锁数量的增加导致了查询性能的下降。当负载率在 20%～50%之间时查询性能提升得最为显著，吞吐量提升了 7%，之后查询性能会逐渐降低。这主要有两个方面的原因：一方面，数据量的增加导致成功查询的数据并不在相同区域内，成功读取数据时 CPU 高速缓存缺失率增加；另一方面，负载率的上升增加了读写时加锁的时间消耗，降低了读取效率。

5.6　LBCHT 和 LPCHT 的应用

将 LBCHT 和 LPCHT 分别应用到 SILT[22-23]（Small Index Large Table）中，SILT 是内存占用率低的高性能键值存储系统，可广泛应用于大数据、云计算领域。我们将 *key-value* 的大小设置为 32B，*key* 的大小设置为 20B，*value* 的大小设置为 12B，哈希表桶的数量设置为 2^{15}，桶的大小 $B=4$，测试程序使用 8 线程。通过替换 SILT 原有的 Cuckoo 哈希表，SILT 的性能有了较大的提升。

实验结果如表 2 所列，LPCHT 和 LBCHT 能够有效提升 SILT 的读写效率，SILT 整体的读写效率提升了 7%左右。在 50%用于查询、50%用于写的情况下，LBCHT 对 SILT 读写效率的提升优于 LPCHT，性能的提升高出 5%左右；同时，在 90%用于查询、10%用于写的情况下，LPCHT 对 SILT 读写性能的提升略高于 LBCHT，符合 LPCHT 通过提升正向查询性能来提升读写效率的期望。

表 2　LBCHT 和 LPCHT 的应用性能

Table 2　Application performance of LBCHT and LPCHT

（单位：k/s）

Algorithms	SILT	SILT LBCHT	SILT LPCHT
50%用于读，50%用于写	62.85	67.24	63.87
90%用于读，10%用于写	98.26	103.45	105.03

通过实验发现，SILT 的性能无法进一步提升的主要原因是硬盘的读写速率成为系统性能提升的瓶颈，可以使用 SSD 或者新型 NVM 作为 SILT 的主存储器件，以进一步提升 LBCHT 和 LPCHT 的性能潜力。

结束语　LBCHT 使用计数器和额外负载因子 a 来动态限制每个桶当前可存储的数据量，使得每个桶的负载保持均衡，但是这种限制是弱限制，在搜索阈值内，如果没有找到符合要求的桶就选择最接近条件的桶，本质上是提前进行数据项迁移，将高负载率下长 Cuckoo 路径转换成多个短 Cuckoo 路径。使用广度优先搜索进行 Cuckoo 搜索，随着搜索深度的增加，读取的数据量以指数级的速度增长，因此能够提升写入效率。计数器使用 CAS 原语，资源消耗少。与其他的 Cuckoo 哈希表相比，LBCHT 主要缓解甚至消除了高延迟插入和无限循环，桶之间的负载保持均衡，提升了高负载率下数据项的写入效率。

LPCHT 发掘数据表项间的空间局部性，将数据优先存储在哈希表的某一区域内，在读负载远大于写负载的情况下，由于减小了正向查询 CPU 高速缓存的缺失率，提升了中低负载率下查询成功的效率，但是高负载率下正向查询效率的提升并不明显。

参 考 文 献

[1]　ZEPHORIA. The Top 20 Valuable Facebook Statistics [EB/OL]. (2018-05-01) [2019-12-01]. https://zephoria.com/top-

15-valuable-facebook-statistics.
[2] WU S, LI F, MEHROTRA S, et al. Query optimization for massively parallel data processing[C]// Proceedings of the 2nd ACM Symposium on Cloud Computing. ACM, 2011: 12.
[3] PETERSON W. Addressing for random-access storage[J]. IBM journal of Research and Development, 1957, 1(2): 130-146.
[4] WANG X, YU H, YIN Y L. Efficient collision search attacks on SHA-0[C]// Annual International Cryptology Conference. Springer, Berlin, Heidelberg, 2005: 1-16.
[5] VITTER J S. Analysis of the search performance of coalesced hashing[J]. Journal of the ACM (JACM), 1983, 30(2): 231-258.
[6] ZUKOWSKI M, HÉMAN S, BONCZ P. Architecture-conscious hashing[C]// Proceedings of the 2nd International Workshop on Data Management on New Hardware. ACM, 2006: 6.
[7] Pagh R, Rodler F F. Cuckoo hashing[J]. Journal of Algorithms, 2004, 51(2): 122-144.
[8] FAN B, ANDERSEN D G, KAMINSKY M. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing[C]// 10th Symposium on Networked Systems Design and Implementation. 2013: 371-384.
[9] SUN Y, HUA Y, FENG D, et al. A Collision-mitigation cuckoo hashing scheme for large-scale storage systems[J]. IEEE Transactions on Parallel and Distributed Systems, 2016, 28(3): 619-632.
[10] KUSZMAUL W. Fast concurrent cuckoo kick-out eviction schemes for high-density tables[J]. arXiv: 1605.05236, 2016.
[11] SAHNI K. CuckooQ Hashing[J]. Rochester Institute of Technology, 2018, 8(5): 1-5.
[12] SUN Y, HUA Y, CHEN Z, et al. Mitigating asymmetric read and write costs in cuckoo hashing for storage systems[C]// 2019 Annual Technical Conference. 2019: 329-344.
[13] SUN Y, HUA Y, JIANG S, et al. SmartCuckoo: A fast and cost-efficient hashing index scheme for cloud storage systems[C]// 2017 Annual Technical Conference. 2017: 553-565.
[14] PAGH R, RODLER F F. Cuckoo hashing[J]. Journal of Algorithms, 2004, 51(2): 122-144.
[15] Github. efficient/libcuckoo[EB/OL]. (2019-12-1)[2019-10-2]. https://github.com/efficient/libcuckoo.
[16] GitHub. sparsehash/sparsehash[EB/OL]. (2019-07-18)[2019-10-18]. https://github.com/sparsehash/sparsehash.
[17] BRESLOW A D, ZHANG D P, GREATHOUSE J L, et al. Horton tables: Fast hash tables for in-memory data-intensive computing[C]// 2016 Annual Technical Conference. 2016: 281-294.
[18] LIU Y, ZHANG K, SPEAR M. Dynamic-sized nonblocking hash tables[C]// Proceedings of the 2014 ACM symposium on Principles of distributed computing. ACM, 2014: 242-251.
[19] TRIPLETT J, MCKENNEY P E, WALPOLE J. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming [C]// USENIX Annual Technical Conference. 2011: 1-14.
[20] MCKENNEY P E, APPAVOO J, KLEEN A, et al. Read-copy update[C]// AUUG Conference Proceedings. 2001: 175.
[21] LI X, ANDERSEN D G, KAMINSKY M, et al. Algorithmic improvements for fast concurrent cuckoo hashing[C]// Proceedings of the Ninth European Conference on Computer Systems. ACM, 2014: 27.
[22] LIM H, FAN B, ANDERSEN D G, et al. SILT: A memory-efficient, high-performance key-value store[C]// Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. ACM, 2011: 1-13.
[23] GitHub. silt/silt [EB/OL]. (2017-9-7) [2020-3-1]. https://github.com/silt/silt.

JIN Qi, born in 1995, postgraduate. His main research interests include high performance computing and data storage.

WANG Jun-chang, born in 1982, Ph.D, lecturer. His main research interests include parallel and distributed computing systems.