软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(7):3339−3357 [doi: 10.13328/j.cnki.jos.007214] [CSTR: 32375.14.jos.007214]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

无锁并发布谷鸟过滤器*

王瀚橙, 陈志鹏, 戴海鹏, 顾 荣, KIM Chaewon, 陈贵海

(南京大学 计算机学院, 江苏 南京 210023)

通信作者: 戴海鹏, E-mail: haipengdai@nju.edu.cn; 陈贵海, E-mail: gchen@nju.edu.cn

摘 要: 布谷鸟过滤器是一种高效的概率型数据结构, 该数据结构可以快速判断某个元素是否存在于给定集合中, 被广泛应用于计算机网络、物联网应用以及数据库系统中. 在实践中, 上述系统通常需要处理海量数据以及大量并发请求. 实现支持高并发的布谷鸟过滤器可以显著提升系统吞吐以及数据处理能力, 对提升系统性能至关重要. 为此, 设计一个支持无锁并发的布谷鸟过滤器. 该过滤器通过所提出的两阶段查询、路径探查与元素迁移分离, 以及基于多机器字比较并交换的原子迁移技术实现高性能的查询、插入和删除操作. 理论分析和实验验证结果均表明, 无锁并发布谷鸟过滤器显著提升现有最先进算法的并发性能. 无锁并发布谷鸟过滤器的查询吞吐量, 平均为使用细粒度锁的布谷鸟过滤器的查询吞吐量的 1.94 倍.

关键词: 布谷鸟过滤器; 并发; 近似成员资格查询; 概率数据结构; 计算机网络

中图法分类号: TP393

中文引用格式: 王瀚橙, 陈志鹏, 戴海鹏, 顾荣, Kim C, 陈贵海. 无锁并发布谷鸟过滤器. 软件学报, 2025, 36(7): 3339–3357. http://www.jos.org.cn/1000-9825/7214.htm

英文引用格式: Wang HC, Chen ZP, Dai HP, Gu R, Kim C, Chen GH. Lock-free Concurrent Cuckoo Filter. Ruan Jian Xue Bao/Journal of Software, 2025, 36(7): 3339–3357 (in Chinese). http://www.jos.org.cn/1000-9825/7214.htm

Lock-free Concurrent Cuckoo Filter

WANG Han-Cheng, CHEN Zhi-Peng, DAI Hai-Peng, GU Rong, KIM Chaewon, CHEN Gui-Hai

(School of Computer Science, Nanjing University, Nanjing 210023, China)

Abstract: The cuckoo filter is an efficient probabilistic data structure that can quickly determine whether an element exists in a given set. The cuckoo filter is widely used in computer networks, IoT applications, and database systems. These systems usually involve the handling of massive amounts of data and numerous concurrent requests in practice. A cuckoo filter that supports high concurrency can significantly improve system throughput and data processing capabilities, which is crucial to system performance enhancement. Therefore, a cuckoo filter that supports lock-free concurrency is designed. The filter achieves high-performance lookup, insertion, and deletion through the two-stage query, separation of path exploration and element migration, and atomic migration based on multi-word compare-and-swap. Theoretical analysis and experimental results indicate that the lock-free concurrent cuckoo filter significantly improves the concurrent performance of the most cutting-edge algorithms in current times. The lookup throughput of a lock-free concurrent cuckoo filter is on average 1.94 times that of a cuckoo filter using fine-grained locks.

Key words: cuckoo filter; concurrency; approximate membership query; probabilistic data structure; computer network

布谷鸟过滤器是一种高性能的近似成员资格查询数据结构. 该数据结构可以快速判断某个元素是否存在于给定集合中, 因而被广泛应用于流数据挖掘[1,2]、查询加速[3,4]、隐私保护[5,6]、智慧城市[7]、网络监测[8]、源代码漏洞检测[9]、区块链[10]、无线传感器网络安全[11]、数据库系统[12−16]、缓存性能优化[17]等应用中. 在实践中, 上述应用通常需要高效地处理海量数据以及大量并发请求. 设计并实现支持高并发的布谷鸟过滤器可以显著提升系统吞

* 基金项目: 国家自然科学基金 (62272223); 江苏省高层次双创计划; 软件新技术与产业化协同创新中心 (南京大学) 支持项目
收稿时间: 2023-10-13; 修改时间: 2024-02-12; 采用时间: 2024-04-11; jos 在线出版时间: 2024-06-20
CNKI 网络首发时间: 2024-06-21

吐以及数据处理能力, 对提升系统性能意义重大[3,18−21].

但是, 设计支持高并发的布谷鸟过滤器十分困难. 其难点主要包括以下两个方面: (1) 正确性. 设计正确的并发数据结构充满挑战. 具体来说, 当多个线程并发地修改同一块数据时, 若不采取合适的同步方案保障操作的原子性, 一个线程可能会读取到另一线程修改操作的中间状态, 使得多线程程序的运行结果完全不可预测. 此外, 编译器指令重排、处理器乱序执行以及数据缓存产生的线程间可见性问题使得多线程程序难以分析. 因此, 为保证正确性, 多线程程序需要仔细设计并分析同步机制以确保数据一致和线程协调. (2) 扩展性. 设计具有高扩展性 (即吞吐随并发线程数量增加而线性或亚线性增加) 的并发数据结构充满挑战. 具体来说, 如果使用全局锁来保证数据结构在多线程并发场景下的正确性, 同一时刻只有一个线程能获得全局锁. 这造成所有线程只能依次执行. 因此, 使用全局锁的数据结构的吞吐无法随线程数量增长而提升[22,23]. 如果使用细粒度锁来保证数据结构在多线程并发场景下的正确性, 多个线程可以同时执行. 但是由于不同线程之间存在锁竞争以及上下文切换开销, 当并发执行的线程数量较多时, 使用细粒度锁的数据结构的吞吐难以进一步提升[24−26].

为解决上述问题, 本文提出了支持无锁并发的布谷鸟过滤器. 此过滤器通过本文提出的两阶段查询协议、路径探查与元素迁移分离、基于多机器字比较并交换的原子迁移实现线程安全的查询、插入和删除操作. 理论分析和实验结果表明, 无锁并发的布谷鸟过滤器显著提升了现有最新算法的并发性能.

本文第 1 节介绍并发过滤器的研究现状. 第 2 节介绍本文所需的基础知识, 包括布谷鸟过滤器以及无锁并发技术. 第 3 节介绍本文提出的无锁并发布谷鸟过滤器. 第 4 节通过实验验证所提方法的有效性. 最后总结全文.

1 相关工作

1.1 过滤器数据结构概述

过滤器数据结构可以近似地判断一个元素是否存在于给定集合中. 过滤器数据结构主要可分为两类: 基于位图的过滤器和基于指纹的过滤器.

基于位图的过滤器主要包括布隆过滤器及其变体[27,28]. 布隆过滤器由一个长度为 m 比特的位图数组以及 k 个哈希函数组成. 布隆过滤器的插入操作根据 k 个哈希函数的计算结果, 将位图中 k 个比特位设置为 1. 布隆过滤器的查询操作根据 k 个哈希函数的计算结果, 查询位图中的 k 个比特位. 若 k 个比特位均为 1, 则返回该元素存在于已插入元素的集合中. 此外, 布隆过滤器不支持删除操作.

基于指纹的过滤器主要包括布谷鸟过滤器及其变体[29,30]. 不同于布隆过滤器通过将元素映射到位图数组中的 k 个比特位来保存元素, 布谷鸟过滤器通过保存元素的 f 位指纹 (即元素哈希值的 f 位片段) 来保存元素. 这一设计使得基于指纹的过滤器可以支持删除操作. 例如, 布谷鸟过滤器由一个具有若干哈希桶的指纹哈希表组成. 布谷鸟过滤器的插入操作会将元素的指纹保存到指纹哈希表中. 查询操作通过查询指纹哈希表中是否存在待查询元素的指纹近似地判断待查询元素是否存在. 布谷鸟过滤器的删除操作会将待删除元素的指纹从指纹哈希表中移除. 注意, 本文将在第 2.1 节中详细介绍布谷鸟过滤器的插入、查询和删除操作.

1.2 并发过滤器及其他相关并发数据结构

现有的过滤器主要使用锁支持并发. 多线程布隆过滤器[31]将细粒度锁应用于布隆过滤器中, 以提升其多核可扩展性. 向量商过滤器[32]将布谷鸟哈希替换为 power-of-two-choices 哈希, 并将哈希桶替换为商过滤器[33]. 之后, 向量商过滤器使用桶级别的细粒度锁实现了较高的并发扩展性. 计数商过滤器[34]在商过滤器基础上通过增加让多个指纹槽共享一个互斥锁的设计提升了多线程性能. 据我们所知, 目前支持并发的过滤器均使用锁实现并发. 由于不同线程之间存在锁竞争, 这不可避免地造成当并发线程数较多时产生性能瓶颈. 本文提出的无锁并发布谷鸟过滤器可以避免锁竞争产生的开销, 从而缓解当并发线程数较多时性能难以继续提升的问题.

并发技术也广泛应用于其他非过滤器数据结构中, 如哈希表[22]、B-树[35,36]、自适应前缀树[37]等. 其中, 与过滤器数据结构最相关的是哈希表数据结构. 例如, 细粒度锁布谷鸟哈希表[22]为若干个连续的哈希桶设置一个锁, 通过细粒度锁实现并发. 无锁并发布谷鸟哈希表[38]针对哈希桶内仅存在一个插槽的特殊情况提出基于 CAS 指令的

无锁并发方案. 但是, 该方案无法应用于哈希桶内存在多个插槽的布谷鸟过滤器中. 无锁罗宾汉哈希表[39]使用 K-CAS 操作[40,41]实现原子地写入多个元素, 并结合时间戳无锁地读取元素. 无锁跳房子哈希[42]使用 K-CAS 操作进行元素迁移操作中的状态设置和时间戳更新. 无等待可扩展哈希[43]通过将桶和目录的数据与引用拆分, 结合写时复制机制与线程间帮助机制, 实现无等待的读写操作. 需要注意, 哈希表数据结构与布谷鸟过滤器虽然结构相近, 但两种数据结构语义上具有本质区别. 例如, 哈希表数据结构会保存元素的完整值, 但布谷鸟过滤器只保存元素的部分值 (指纹). 这一区别造成该领域的相关工作难以应用于本文研究的过滤器数据结构中.

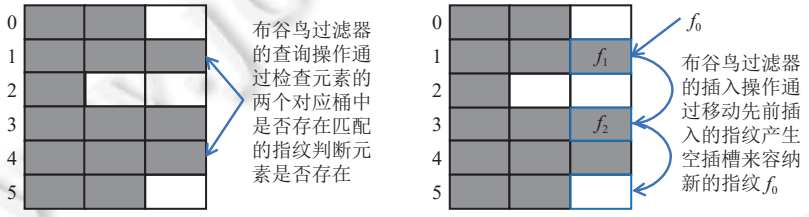
综上, 现有的并发过滤器数据结构主要使用锁提升多线程性能, 当并发线程数量较多时, 锁竞争会影响数据结构的性能, 继而导致并发性能瓶颈. 据我们所知, 本文提出的无锁并发布谷鸟过滤器是首个可以支持无锁并发的布谷鸟过滤器数据结构.

2 背景知识

本文所提出的方法主要基于布谷鸟过滤器和多机器字比较并交换 (K-CAS) 技术, 接下来将对上述两点背景知识进行介绍.

2.1 布谷鸟过滤器

布谷鸟过滤器[30]的数据结构由哈希表、哈希桶、插槽三级存储结构组成. 具体来说, 布谷鸟过滤器的哈希表由若干哈希桶组成, 每个哈希桶中包含若干插槽. 图 1 展示了一个具有 6 个哈希桶的布谷鸟过滤器, 其中每个哈希桶中具有 3 个插槽.



(a) 布谷鸟过滤器的查询操作 (b) 布谷鸟过滤器的插入操作

图 1 布谷鸟过滤器执行查询操作和插入操作的示例

查询操作: 布谷鸟过滤器通过保存元素的指纹来近似地存储元素, 并且每个元素均有两个哈希桶与该元素对应. 具体来说, 对于待查询元素 e, 布谷鸟过滤器首先使用哈希函数 h_f 计算元素的指纹 $f = h_f(e)$. 之后, 布谷鸟过滤器使用哈希函数 h_i 计算元素对应的两个候选桶地址 $i_1 = h_i(e)$ 和 $i_2 = h_i(e) \oplus h_i(f)$. 最后, 布谷鸟过滤器逐个比较两个候选桶中的插槽, 判断是否存在和待查询元素 e 的指纹 f 相同的指纹. 只要某个候选桶中存在一个相同的指纹则返回待查询元素 e 存在. 反之, 则返回待查询元素 e 不存在. 图 1(a) 给出了查询元素是否存在的一个示例. 待查询元素的两个候选桶为桶 1 和桶 4, 只要桶 1 和桶 4 所包含的 6 个插槽中的任意一个插槽内具有匹配的指纹, 则返回待查询元素 e 存在.

插入操作: 对于待插入元素 e, 布谷鸟过滤器首先计算元素的指纹 f 以及该元素的两个候选桶的地址 i_1 和 i_2. 若两个候选桶中存在空插槽, 则直接将元素 e 的指纹插入任意一个空插槽中并返回插入成功. 若两个候选桶中均没有空插槽, 则从两个候选桶中随机踢出一个指纹 f', 进而产生一个空插槽来存放待插入元素 e 的指纹 f. 之后, 若被踢出的指纹 f' 的另一个候选桶中有空插槽, 则将被踢出的指纹保存到另一个候选桶中. 否则, 从另一个候选桶中再随机踢出一个指纹 f'', 产生一个空插槽存放指纹 f'. 重复上述过程直到寻找到一个空插槽存放被踢出的指纹. 图 1(b) 给出了向布谷鸟过滤器中插入指纹为 f_0 的元素的示例. 由于该元素的两个候选桶均已满, 因此需要从候选桶中随机踢出一个指纹 (例如 f_1), 并将 f_0 保存到 f_1 所在的插槽中. 由于指纹 f_1 的另一个候选桶中也不存在空插槽, 则继续随机踢出指纹 f_2 来存放指纹 f_1. 最后, 由于指纹 f_2 的另一个候选桶中有空插槽, 因此指纹 f_2 可以直接

存放在另一个候选桶中, 并返回插入成功.

删除操作: 删除操作首先计算待删除元素的指纹以及两个候选桶的地址, 之后从候选桶中删除一个和待删除元素的指纹相同的指纹.

2.2 多机器字比较并交换

比较并交换操作 (compare and swap) 是一种底层的原子操作, 该操作常被用于无锁并发算法的设计. 该操作的功能为: 首先将内存中存储的值与待比较数据进行对比, 若内存中存储的值和待比较数据 d_1 相同, 则触发交换操作将内存中存储的值更新为 d_2; 若内存中存储的值和待比较数据 d_1 不同, 则不触发交换操作并保持内存中存储的值不变. 该操作可以原子地实现比较并交换功能, 执行时不会被拆分为子指令. 因而任何线程不会观测到该操作的中间状态, 而只会观测到该操作尚未执行, 或已经执行完成. 相比于使用加锁的方式实现并发, 比较并交换操作更加底层且高效. 因此, 当使用锁的多线程程序遇到高并发性能瓶颈时, 将锁替换为比较并交换操作有望进一步提升并发性能.

多机器字比较并交换操作 (K-CAS)[40,41]是比较并交换操作的扩展. 具体来说, 不同于比较并交换操作只能原子地修改单个内存地址中的数据, 多机器字比较并交换操作能同时对多个不同内存地址中的数据执行比较并交换操作. 换句话说, 使用多机器字比较并交换技术读写数据的线程不会观测到多机器字比较并交换操作的中间状态, 只会观测到所有 K 个比较并交换操作均未执行或均已全部执行完毕.

3 无锁并发布谷鸟过滤器设计

本节从查询、插入和删除这 3 个操作出发, 详细介绍了无锁并发布谷鸟过滤器的设计. 此外, 本节还证明了上述 3 个操作均是可线性化的.

3.1 查询操作

传统布谷鸟过滤器的查询操作在并发场景下会产生查询漏报问题. 具体来说, 对于单线程场景, 布谷鸟过滤器的查询操作只需要依次检查待查询元素的两个候选桶 i_1 和 i_2 中是否存在匹配的指纹即可. 但是, 对于多线程并发场景, 直接沿用这一设计会导致并发插入的元素指纹无法被查询到. 例如, 对于某个执行查询操作的线程, 若该线程首先访问待查询元素的第 1 个候选桶 i_1, 再访问待查询元素的第 2 个候选桶 i_2. 由于两次访问操作不是原子的, 若在两次访问操作执行之间存在某个执行插入操作的线程将待查询元素的指纹从候选桶 i_2 迁移到候选桶 i_1 中, 则会造成即使待查询元素的指纹存在于布谷鸟过滤器中, 执行查询操作的线程依然无法查询到该指纹, 并返回“待查询元素不存在”的错误信息, 继而造成查询漏报, 这是使用过滤器数据结构的应用[1−17]所无法接受的.

为解决并发引起的查询漏报问题, 本文提出了两阶段查询技术. 该技术的第 1 阶段将检查两个候选桶中是否存在匹配的指纹, 若不存在, 则进入第 2 阶段, 再次检查两个候选桶中是否存在匹配的指纹. 若两个阶段均未发现匹配的指纹, 并且在两阶段查询过程中没有指纹在两个候选桶之间反复迁移, 则待查询元素必不存在, 从而避免了查询漏报.

两阶段查询技术解决查询漏报问题的核心设计在于通过迁移计数器判断指纹是否在两个候选桶之间反复迁移. 具体来说, 无锁并发布谷鸟过滤器为每个候选桶添加一个记录元素迁移情况的迁移计数器. 当某个指纹被执行插入操作的线程从候选桶 i_1 迁移到候选桶 i_2 后, 若迁移前两个候选桶的迁移计数器值分别为 t_1 和 t_2, 则迁移后无锁并发布谷鸟过滤器会将两个候选桶的迁移计数器的值分别更新为 t_1+1 和 $\max(t_1,t_2)+1$. 基于这一设计, 两阶段查询技术可以通过迁移计数器数值变化判断候选桶中是否有指纹迁移发生.

基于上述核心设计, 两阶段查询技术解决查询漏报问题的方案如下. (1) 两阶段查询首先检查待查询元素对应的两个候选桶中是否存在匹配的指纹, 并记录此时两个候选桶的迁移计数器取值 t_1 和 t_2. (2) 若未查询到匹配的指纹, 则重新检查两个候选桶中是否存在匹配的指纹, 并再次记录两个候选桶的迁移计数器取值 t_1' 和 t_2'. (3) 若上述两个阶段均未查询到匹配的指纹, 并不能直接说明待查询元素不存在. 具体来说, 若在第 1 阶段查询候选桶 i_1 和候

选桶 i_2 之间的某一时刻, 存在其他执行插入操作的线程将待查询元素的指纹从候选桶 i_2 迁移到候选桶 i_1 中. 之后在第 2 阶段查询开始前, 又将该指纹从候选桶 i_1 迁移到候选桶 i_2 中. 最后在第 2 阶段查询候选桶 i_1 和候选桶 i_2 之间的某一时刻, 将该指纹从候选桶 i_2 再次迁移到候选桶 i_1 中, 则一次两阶段查询无法发现该指纹. 本文将上述情况称为: 该指纹在两个候选桶之间反复迁移. 上述问题可通过检查两阶段查询过程中记录的迁移计数器的取值发现并加以解决. 具体来说, 当迁移操作发生后, 参与迁移的两个桶的计数器取值都会大于等于迁移前取值加一. 利用这一性质, 当第 2 阶段查询没有发现指纹时, 无锁并发布谷鸟过滤器会检查两个阶段记录的迁移计数器值. 若两个候选桶在第 2 阶段中记录的迁移计数器值均大于等于第 1 阶段中记录的迁移计数器值加 2, 且第 2 个候选桶第 2 阶段迁移计数器值大于等于第 1 个候选桶第 1 阶段迁移计数器值加 3, 则说明指纹发生了反复迁移, 并可能会造成两阶段查询无法发现该指纹. 此时需要重试两阶段查询, 直到某次两阶段查询没有发生指纹反复迁移为止. 本文第 3.5 节证明, 任意时刻至少有一个线程不会无限重试. 当两阶段查询结束时, 若两个阶段均未发现匹配指纹, 且查询过程中没有指纹在两个候选桶之间反复迁移, 说明待查询元素必不存在. 因此, 两阶段查询技术可以避免查询漏报发生.

无锁并发布谷鸟过滤器的查询操作如算法 1 所示. 无锁并发布谷鸟过滤器首先计算待查询元素的指纹以及两个候选桶的下标 (第 1, 2 行). 之后执行两阶段查询流程 (第 4–18 行). 具体来说, 其中第 1 阶段查询会分别读取两个候选桶并查询其中是否存在匹配的指纹 (第 5–8 行). 若第 1 阶段查询未找到匹配的指纹, 则会发起第 2 阶段查询 (第 10–13 行). 若两阶段查询均未找到匹配的指纹则需检查迁移计数器并判断是否需要再次重试两阶段查询流程. 若不需要重试则直接判定元素不是集合成员 (第 15–18 行).

算法 1. 无锁并发布谷鸟过滤器的查询操作.

输入: 一个字符串类型的变量 e, 其中 e 表示待查询的元素;
输出: 一个布尔类型的变量 TRUE 或 FALSE. 其中, 输出 TRUE 表示查询成功, 即待查询元素存在于布谷鸟过滤器中; 输出 FALSE 表示查询失败, 即待查询元素不存在于布谷鸟过滤器中.

```
1. 计算元素指纹 f = h_f(e);
2. 计算元素对应的两个候选桶地址 i_1 = h_i(e) 和 i_2 = h_i(e)⊕h_i(f);
3. /*注释: 性质 7 证明, 不存在某一时刻, 所有线程均陷入死循环*/
4. while (TRUE)
5.    读取桶 i_1 中的所有指纹和迁移计数器 t_1;
6.    读取桶 i_2 中的所有指纹和迁移计数器 t_2;
7.    if (桶 i_1 或者桶 i_2 中有和待查询元素 e 的指纹 f 相同的指纹)
8.       return TRUE;
9.    /*注释: 执行第 2 轮查询*/
10.   读取桶 i_1 中的所有指纹和迁移计数器 t'_1;
11.   读取桶 i_2 中的所有指纹和迁移计数器 t'_2;
12.   if (桶 i_1 或者桶 i_2 中有和待查询元素 e 的指纹 f 相同的指纹)
13.      return TRUE;
14.   /*注释: 判断是否需要重试查询操作*/
15.   if (t'_1 ⩾ t_1 + 2 & t'_2 ⩾ t_2 + 2 & t'_2 ⩾ t_1 + 3)
16.      continue;
17.   else
18.      return FALSE;
```

两阶段查询技术可以实现无锁并发查询. 具体来说, 对于某个存在于无锁并发布谷鸟过滤器中的元素 e, 对该

元素的查询操作可分为 4 种情况, 且每种情况均可查询到该元素的指纹.

(1) 若查询过程中不存在其他执行插入操作的线程将元素 e 的指纹从候选桶 i_2 迁移到候选桶 i_1 中, 则无锁并发布谷鸟过滤器会在第 1 阶段查询中返回元素存在.

(2) 若查询过程中存在其他执行插入操作的线程将指纹从候选桶 i_2 迁移到候选桶 i_1, 且后续没有任何线程继续迁移该指纹. 则无锁并发布谷鸟过滤器会在第 2 阶段查询中发现该指纹存在于候选桶 i_1 中, 并返回存在.

(3) 若查询过程中存在其他执行插入操作的线程首先将待查询元素的指纹从候选桶 i_2 迁移到候选桶 i_1 中, 之后在第 2 阶段查询开始前, 将该指纹从候选桶 i_1 迁移到候选桶 i_2 中, 且后续没有任何线程继续迁移该指纹. 则无锁并发布谷鸟过滤器会在第 2 阶段查询中发现该指纹存在于候选桶 i_2 中, 并返回元素存在.

(4) 若查询过程中存在其他执行插入操作的线程首先将待查询元素的指纹从候选桶 i_2 迁移到候选桶 i_1 中. 之后在第 2 阶段查询开始前, 将该指纹从候选桶 i_1 迁移到候选桶 i_2 中. 最后在第 2 阶段访问候选桶 i_1 和候选桶 i_2 的间隙将该指纹从候选桶 i_2 再次迁移到候选桶 i_1 中, 则一次两阶段查询无法发现该指纹. 此时, 无锁并发的布谷鸟过滤器可根据两阶段查询过程中记录的迁移计数器取值发现指纹在查询期间发生了反复迁移, 并将重试两阶段查询过程.

查询操作时间复杂度分析如下. 当两阶段查询未发生重试时, 无锁并发布谷鸟过滤器只需检查两次两个候选桶即可返回元素是否存在. 因此, 当未发生重试时, 无锁并发布谷鸟过滤器查询操作的时间复杂度与标准布谷鸟过滤器相同, 均为 $O(1)$. 若两阶段查询发生重试, 无锁并发布谷鸟过滤器查询操作的时间复杂度为 $O(r)$, 其中 r 为重试次数. 本文通过第 3.5 节的理论分析表明查询操作的进展性为无锁, 即任何时刻至少有一个线程不会无限重试 (详见第 3.5 节).

3.2 插入操作

无锁并发布谷鸟过滤器的插入操作如算法 2 所示. 无锁并发布谷鸟过滤器首先计算待插入元素的指纹以及两个候选桶的索引 (第 1, 2 行). 之后读取两个候选桶 (第 5, 6 行), 并判断候选桶中是否存在空插槽. 若某个候选桶中存在空插槽, 则使用比较并交换操作 (compare and swap) 将指纹原子地写入到无锁并发布谷鸟过滤器中 (第 8–16 行). 若两个候选桶中均没有空插槽, 则需要移动已插入元素的指纹来产生空插槽以容纳待插入元素的指纹 (第 17 行). 下文将详细描述移动已插入指纹的方法.

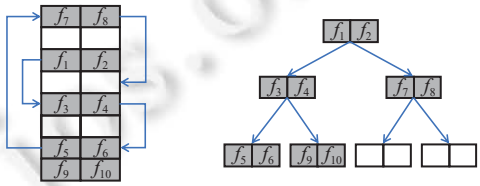
算法 2. 无锁并发布谷鸟过滤器的插入操作.

输入: 一个字符串类型的变量 e, 其中 e 表示待插入的元素;
输出: 一个布尔类型的变量 TRUE, 表示插入成功. /*注: 由于过滤器可扩容, 插入必能成功.*/

1. 计算元素指纹 $f = h_f(e)$;
2. 计算元素对应的两个候选桶地址 $i_1 = h_i(e)$ 和 $i_2 = h_i(e) \oplus h_i(f)$;
3. /*注释: 循环可在第 11 或 16 行跳出. 性质 8 证明, 不存在某一时刻, 所有线程均陷入死循环*/
4. **while** (TRUE)
5. 读取桶 i_1 中的所有插槽;
6. 读取桶 i_2 中的所有插槽;
7. /*注释: 查询桶 i_1 中是否有空插槽 */
8. **if** (桶 i_1 中有空插槽)
9. **if** (使用比较并交换操作写入失败)
10. **continue**;
11. **return** TRUE;
12. /*注释: 查询桶 i_2 中是否有空插槽*/
13. **if** (桶 i_2 中有空插槽)

14. **if** (使用比较并交换操作写入失败)
15. **continue**;
16. **return** TRUE;
17. 根据桶 i_1 和 i_2 开始执行路径探查和元素迁移操作;
18. /*注释: 如果无法找到可行路径, 则对过滤器进行扩容, 直到插入成功*/

对于单线程场景, 标准布谷鸟过滤器指纹移动所涉及的路径探查和指纹迁移过程是紧密耦合的. 具体来说, 若某个指纹 f 从一个候选桶 (例如, 桶 i_1) 中被踢出后, 在其另一个候选桶 i_2 中没有发现空插槽, 此时插入操作会随机踢出候选桶 i_2 中的某个插槽中的指纹 (例如, 桶 f') 以产生一个空插槽来存储被踢出的指纹. 被踢出的指纹 f' 将重复上述流程, 直到在另一个候选桶中找到能够直接存储该指纹的空插槽为止. 图 2(a) 给出了上述过程的一个示例. 标准布谷鸟过滤器以桶 $< f_1, f_2 >$ 为起点, 首先执行路径探查操作发现指纹 f_1 可迁移到桶 $< f_3, f_4 >$ 中. 之后执行元素迁移操作, 将指纹 f_3 踢出用于存放指纹 f_1. 指纹 f_3 被踢出后, 标准布谷鸟过滤器再次执行路径探查操作发现被踢出的指纹 f_3 可迁移到桶 $< f_5, f_6 >$ 中并再次执行元素迁移操作. 简言之, 标准布谷鸟过滤器的插入操作需要交替执行路径探查和指纹迁移过程, 造成路径探查和指纹迁移过程紧密耦合.



(a) 单线程标准布谷鸟过滤器使用深度优先方法以桶$<f_1, f_2>$为起点探查布谷鸟逐出路径　(b) 无锁并发布谷过滤器使用广度优先方法以桶$<f_1, f_2>$为起点探查布谷鸟逐出路径

图 2　单线程标准布谷鸟过滤器和无锁并发布谷鸟过滤器路径探查过程对比

但是, 对于多线程并发场景, 直接使用上述方案存在两点问题: (1) 路径探查和指纹迁移紧密耦合会导致插入过程中存在某些指纹被暂时踢出数据结构, 当一个插入线程将指纹从哈希表中踢出且尚未将其写入到另一个候选桶中时, 此刻指纹存储在线程本地内存空间中, 无法被其他线程获取. 若此时有其他执行查询操作的线程正在寻找此指纹, 显然无法查询到该元素, 继而产生假阴性. (2) 路径探查和指纹迁移紧密耦合使得标准布谷鸟过滤器的路径探查操作不得不基于深度优先方法进行搜索. 然而深度优先方法相比于广度优先方法探查到的逐出路径更长, 造成多线程插入时不同写入线程冲突加剧, 继而难以获得性能提升.

为解决上述问题, 本文提出了路径探查与元素迁移分离技术. 该技术首先使用广度优先方法进行路径探查, 当探查到可行逐出路径后再执行元素迁移. 如图 2(b) 所示, 无锁并发布谷鸟过滤器以桶 $< f_1, f_2 >$ 为起点, 通过广度优先搜索首先计算出指纹 f_1 和指纹 f_2 的候选桶分别为桶 $< f_3, f_4 >$ 和桶 $< f_7, f_8 >$. 由于上述两桶中均没有空插槽, 无锁并发布谷鸟过滤器继续使用广度优先搜索计算指纹 f_3, f_4, f_7, f_8 的候选桶. 并可探查到一条长度为 3 的指纹逐出路径 $f_2 \Rightarrow f_7 \Rightarrow < empty >$. 由于使用广度优先搜索可获得最短的逐出路径, 因此这一设计缓解了标准布谷鸟过滤器深度优先搜索造成迁移路径过长的问题 (简言之, 这一设计可缓解问题 (2)).

基于这一设计, 无锁并发布谷鸟过滤器实现了在不迁移元素的前提下探查路径. 当找到一条可行的指纹迁移路径后, 无锁并发布谷鸟过滤器开始执行元素迁移操作: 无锁并发布谷鸟过滤器将从可行迁移路径的最后一个桶开始, 反向遍历路径, 逐步将空插槽移动到待插入元素的候选桶中. 此设计解决了其他执行查询操作的线程因无法访问到迁移过程中被暂时踢出数据结构的指纹从而产生假阴性的问题 (简言之, 这一设计可解决问题 (1)).

路径探查与元素迁移分离技术的伪代码如算法 3 所示. 具体来说, 路径探查操作首先向队列中插入广度优

先搜索起点 (第 3–5 行). 该队列记录了每个元素的哈希桶索引、已确定的迁移路径以及当前探测深度. 需要注意, 对于已确定的迁移路径, 若使用一个变长数组记录已确定路径的哈希桶索引和指纹槽下标会占用较多空间. 本文提出一种压缩编码方案, 该方案仅使用一个定长整数即可记录已确定的迁移路径. 如图 3 所示, 由于指纹槽下标仅需两个比特位即可表示, 因此可以将已确定路径中的所有指纹槽下标编码到一个整数中. 在此基础上, 额外使用一个比特位区分迁移路径的起始哈希桶. 根据起始哈希桶索引和迁移路径上的指纹槽下标, 可解码出完整的迁移路径.

算法 3. 无锁并发布谷鸟过滤器的路径探查和元素迁移操作.

输入: 两个整数类型的变量 i_1 和 i_2. 其中, i_1 和 i_2 表示两个路径探查起点桶地址;
输出: 一个布尔类型的变量 TRUE 或 FALSE. 其中, 输出 TRUE 表示操作成功, 即成功探查到可行迁移路径, 并成功完成指纹迁移; 输出 FALSE 表示操作失败, 即无法探查到可行迁移路径.

```
1. /*注释: 开始路径探查过程*/
2. path_discovery:
3.    初始化广度优先队列 q 为空;
4.    插入元素 (bucket = i1, path = 0, depth = 0) 到队列 q 中;
5.    插入元素 (bucket = i2, path = 1, depth = 0) 到队列 q 中;
6.    while (q 不为空)
7.       取队列 q 队首元素 x;
8.       读取桶 [x.bucket] 的值;
9.       for (桶 [x.bucket] 中所有的插槽)
10.         if (该插槽中没有指纹)
11.            成功获得一条可行迁移路径;
12.            跳出第 6 行的 while 循环;
13.         if (当前搜索深度未达到阈值)
14.            将插槽中存储的指纹的另一候选桶的信息 (bucket, path, depth) 入队;
15. 若无法成功探查到可行路径, 则返回 FALSE, 否则开始指纹迁移流程;
16. 解码第 11 行获得的可迁移路径;
17. 记录可迁移路径长度;
18. /*注释: 反向遍历可迁移路径并迁移元素*/
19. for (反向遍历可行迁移路径)
20.      获得源哈希桶索引;
21.      获得待迁移指纹的指纹槽下标;
22.      计算目标哈希桶索引;
23.      if (源哈希桶索引和目标哈希桶索引相同)
24.         continue;
25.      读取目标哈希桶中每个插槽中的指纹;
26.      if (目标哈希桶中没有空插槽)
27.         goto path_discovery;
28.      将待迁移指纹原子地从源哈希桶迁移到目标哈希桶中;
29. return TRUE;
```

路径探查与元素迁移分离技术初始化后不断取出队首元素, 读取队首元素对应哈希桶 (第 7, 8 行), 并检查

该哈希桶中是否存在空插槽. 若哈希桶中存在空插槽, 则成功搜索到可行路径并结束路径探查 (第 10–12 行). 否则, 将该哈希桶中每个指纹对应的另一候选桶的相关信息插入到队列中 (第 13, 14 行), 并继续寻找可行迁移路径. 找到可行迁移路径后, 首先使用图 3 中存储的信息解码可行迁移路径 (第 16, 17 行). 之后反向遍历可行迁移路径, 对于可行迁移路径中的每个元素, 首先寻找该元素的源哈希桶对应的目标哈希桶 (第 20–27 行). 若目标桶中无空槽则需要重试搜索路径. 这是因为在执行元素迁移操作过程中, 已探查到的可行迁移路径可能被其他并发执行插入操作的线程破坏. 此时, 元素迁移操作无法继续执行, 需要重新执行路径探查操作以获得新的可行迁移路径. 若目标桶中有空槽, 则可直接进行一次指纹原子迁移 (第 28 行). 下文将详细介绍指纹原子迁移过程.

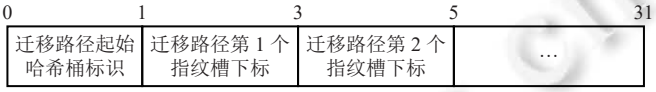


图 3 使用一个 32 位定长整数编码迁移路径

在执行元素迁移操作过程中 (算法 3 第 28 行), 每个指纹从源哈希桶迁移到目标哈希桶的过程需要保证是原子的[44]. 如果简单地将迁移步骤拆分为两个 CAS 操作 (先在目标哈希桶中插入指纹, 再将源哈希桶中指纹删除), 则在某一时刻会出现源哈希桶和目标哈希桶中指纹都存在的中间状态. 若有并发的删除操作在此中间状态执行, 此时本应被删除的指纹将仍然存在于过滤器中, 导致后续查询操作仍然认为该元素是集合成员, 与预期的删除操作语义不符.

为解决这一问题, 本文提出了基于多机器字比较并交换 (K-CAS) 的原子迁移技术, 该技术使用 K-CAS 操作原子地更新两个哈希桶, 使得无锁并发布谷鸟过滤器能在目标哈希桶与源哈希桶间原子地迁移指纹. 具体来说, 单次指纹迁移操作如算法 4 所示. 该操作首先读取源哈希桶索引并判断待迁移指纹是否仍然存在, 以防止待迁移指纹在迁移前被其他执行删除操作的线程删除 (第 2–4 行). 若待迁移指纹存在, 则继续读取目标哈希桶, 并判断目标哈希桶中是否有空指纹槽 (第 6–11 行). 若有空指纹槽, 则计算更新后的源哈希桶和目标哈希桶值, 并使用 K-CAS 操作原子地更新两个哈希桶 (第 12 行).

算法 4. 无锁并发布谷鸟过滤器的单次指纹迁移操作.

输入: 两个整数类型变量: 源哈希桶索引以及待迁移指纹所在指纹槽下标;
输出: 一个布尔类型的变量 TRUE 或 FALSE. 其中, 输出 TRUE 表示操作成功, 即成功完成单次指纹迁移; 输出 FALSE 表示操作失败, 即未成功完成单次指纹迁移.

```
1. while (TRUE)
2.     读取源哈希桶中的所有指纹;
3.     if (待迁移指纹已不存在)
4.         return FALSE;
5.     /*注释: 读取目标哈希桶*/
6.     计算目标哈希桶的索引;
7.     读取目标哈希桶中的所有指纹;
8.     /*注释: 若目标桶可以容纳待迁移指纹则开始迁移指纹*/
9.     if (源哈希桶和目标哈希桶索引不相等并且目标哈希桶中有空插槽)
10.        if (源哈希桶中的数据发生了改变)
11.            continue;
12.        使用 K-CAS 更新源哈希桶和目标哈希桶中的指纹和迁移计数器.
```

```
13.      if (K-CAS 成功更新源哈希桶和目标哈希桶中的指纹和迁移计数器)
14.          return TRUE;
15.      else
16.          return FALSE;
17.  else
18.      return FALSE;
```

K-CAS 操作可原子地将一个指纹从哈希桶 i_1 迁移到哈希桶 i_2 中. 具体来说, 若线程 T_1 正在执行指纹迁移操作, 则该线程需要完成如下 3 个步骤: (1) 线程 T_1 首先原子地将哈希桶 i_1 的值标记为特殊值 V_{T1}. 若标记成功, 则其他需要读写哈希桶 i_1 的线程将停止读写哈希桶 i_1 并协助线程 T_1 完成线程 T_1 的后续步骤 (步骤 (2) 和步骤 (3)). 如果标记失败, 则线程 T_1 不会对哈希桶 i_1 进行任何修改, 从而保证原子性. (2) 当哈希桶 i_1 被线程 T_1 标记成功后, 线程 T_1 将会再次原子地将哈希桶 i_2 的值标记为特殊值 V_{T1}. 如果再次标记成功, 此时, 其他任何读写哈希桶 i_1 和 i_2 的线程均只能协助 T_1 线程完成 T_1 线程的第 3 步. 如果标记失败, 线程 T_1 会还原哈希桶 i_1 的值. 还原后线程 T_1 没有对哈希桶 i_1 和哈希桶 i_2 进行任何修改, 从而保证原子性. (3) 当哈希桶 i_1 和哈希桶 i_2 均被标记为特殊值 V_{T1} 后, 此时, 其他所有需要读写哈希桶 i_1 或 i_2 的线程均只能协助线程 T_1 修改这两个哈希桶. 即, 这些线程只能执行线程 T_1 将要执行的修改操作, 并且只有在 T_1 所需修改全部完成后, 才能切换回先前程序运行的上下文继续读写这两个哈希桶. 基于上述机制, 其他需要读写这两个哈希桶的线程在其程序运行上下文中只能读取到 T_1 线程开始修改两个哈希桶前或修改两个哈希桶后的哈希桶值. 因此, 从其他线程视角看, 这两个哈希桶的修改是原子的. 综上所述, 基于多机器字比较并交换的原子迁移技术可以保证指纹迁移操作只存在两种结果: 其一为指纹迁移完全成功 (步骤 (3)), 其二为不对哈希桶执行任何修改 (步骤 (1)、步骤 (2)). 即, 其他线程在他们的程序运行上下文中只能读取到 T_1 开始修改前或修改后的值, 因此可保证操作的原子性.

上述说明的一个简单示例如下, 假设线程 T_1 需要将指纹从哈希桶 i_1 迁移到哈希桶 i_2, 线程 T_2 需要读取两个哈希桶的值. 使用 K-CAS 操作后, 实际执行过程为: (1) 假设线程 T_1 抢先将哈希桶 i_1 的值标记为特殊值. (2) 当线程 T_2 读取到被标记的哈希桶 i_1 时, 线程 T_2 将暂停对哈希桶 i_1 的读取, 并转而代替线程 T_1 完成对哈希桶 i_2 的标记以及对哈希桶 i_1 和 i_2 的写入操作. (3) 当线程 T_2 协助线程 T_1 完成写入后, 线程 T_2 才能继续读取哈希桶 i_1 和 i_2 的值. 即, 线程 T_2 最终读取到的是写入完成后的值, 而非修改期间可能出现的值, 因此保证了原子性.

插入操作时间复杂度分析如下. 相较于标准布谷鸟过滤器, 无锁并发布谷鸟过滤器的插入操作主要引入了 K-CAS 多字节比较并交换操作. 当插入操作不产生重试时, 由于 K-CAS 操作本身并不增加插入操作的时间复杂度, 无锁并发布谷鸟过滤器的插入时间复杂度和标准布谷鸟过滤器的插入时间复杂度同为 $O(1)$. 若插入操作发生重试, 无锁并发布谷鸟过滤器插入操作的时间复杂度为 $O(r)$, 其中 r 为重试次数. 本文通过第 3.5 节的理论分析表明插入操作的进展性为无锁, 即任何时刻至少有一个线程不会无限重试 (详见第 3.5 节).

3.3 删除操作

无锁并发布谷鸟过滤器的删除操作如算法 5 所示. 无锁并发布谷鸟过滤器的删除操作在查询操作的基础上增加了找到待删除指纹后使用 CAS 操作删除指纹的过程 (第 8, 12, 19, 23 行).

算法 5. 无锁并发布谷鸟过滤器的删除操作.

输入: 一个字符串类型的变量 e, 其中 e 表示待删除的元素;
输出: 一个布尔类型的变量 TRUE 或 FALSE. 其中, 输出 TRUE 表示删除成功, 即成功从布谷鸟过滤器中删除给定元素 e; 输出 FALSE 表示删除失败, 即未成功从布谷鸟过滤器中删除给定元素 e.

1. 计算元素指纹 $f = h_f(e)$;
2. 计算元素对应的两个候选桶地址 $i_1 = h_i(e)$ 和 $i_2 = h_i(e) \oplus h_i(f)$;

```
3. /*注释: 开始删除操作, 性质 9 证明, 不存在某一时刻, 所有线程均陷入死循环*/
4. while (TRUE)
5.    读取桶 i1 中的所有指纹和迁移计数器 t1;
6.    读取桶 i2 中的所有指纹和迁移计数器 t2;
7.    if (桶 i1 中有指纹 f)
8.       if (使用比较并交换操作写入删除指纹后的桶失败)
9.          continue;
10.      return TRUE;
11.   if (桶 i2 中有指纹 f)
12.      if (使用比较并交换操作写入删除指纹后的桶失败)
13.         continue;
14.      return TRUE;
15.   /*注释: 执行第 2 轮查询 */
16.   读取桶 i1 中的所有指纹和迁移计数器 t1';
17.   读取桶 i2 中的所有指纹和迁移计数器 t2';
18.   if (桶 i1 中有指纹 f)
19.      if (使用比较并交换操作写入删除指纹后的桶失败)
20.         continue;
21.      return TRUE;
22.   if (桶 i2 中有指纹 f)
23.      if (使用比较并交换操作写入删除指纹后的桶失败)
24.         continue;
25.      return TRUE;
26.   /*注释: 判断是否需要重试*/
27.   if (t1' ≥ t1 + 2 & t2' ≥ t2 + 2 & t2' ≥ t1 + 3)
28.         continue;
29.   else
30.         return FALSE;
```

删除操作时间复杂度分析如下. 当删除操作未发生重试时, 无锁并发布谷鸟过滤器只需读写两次两个候选桶即可. 因此, 当未发生重试时, 无锁并发布谷鸟过滤器删除操作的时间复杂度与标准布谷鸟过滤器相同, 均为 $O(1)$. 若删除操作发生重试, 和查询操作的分析方法相同, 无锁并发布谷鸟过滤器删除操作的时间复杂度也为 $O(r)$, 其中 r 为重试次数. 通过第 3.5 节的理论分析表明删除操作的进展性也为无锁, 即任何时刻至少有一个线程不会无限重试 (详见第 3.5 节).

3.4 正确性分析

性质 1. 查询操作是可线性化的.

若查询操作在两阶段查询过程中发现与待查询元素匹配的指纹, 则读对应的哈希桶指令可作为查询操作的可线性化点. 除此之外, 当且仅当待查询元素的指纹在访问哈希桶的间隙中被多次迁移时, 两阶段查询过程无法发现该指纹. 但是由于每次指纹迁移成功后, 参与迁移的两个哈希桶的迁移计数器值至少增加 1. 因此当两阶段查询没有发现元素对应的指纹时, 通过检查迁移计数器即可识别此类情况. 之后, 触发重试直到找到待查询元素的指纹为止. 需要注意, 通过迁移计数器判断是否需要重试, 可能会将不需要重试的情况误报为需要重试. 例如在第 1 和第 2

阶段查询之间迁移两次哈希桶中某个其他指纹, 将会产生需要重试两阶段查询的误判. 但是, 由于使用迁移计数器不会将需要重试的场景判断为无需重试. 因此, 当查询操作返回元素不存在时, 说明元素对应的指纹在两阶段查询流程中并未找到, 并且两阶段查询过程中的计数器变化也不足以引发重试. 在这种情况下, 最后一次读哈希桶的指令可作为可线性化点.

但是, 在部分特殊情况下, 线性化点会发生变化. 在算法 1 第 10 行的读哈希桶操作完成后指纹被插入到此哈希桶的情况下, 若将最后一次读哈希桶的操作视为线性化点, 则预期能够查询到元素对应的指纹, 而实际无法查询到, 因此可线性化点应调整为第 10 行. 同理, 在算法 1 第 11 行的读哈希桶操作完成后指纹被插入到此哈希桶的场景下, 可线性化点应调整为第 11 行. 在第 10 行和第 11 行的读哈希桶操作之间, 指纹被从第 11 行对应的哈希桶中移除, 再被插入到第 10 行对应的哈希桶的场景下, 第 10 行和第 11 行都不能作为线性化点, 因为这样预期能够查询元素对应的指纹, 而实际上无法查询. 但是从指纹删除到指纹再次插入的时间段中, 指纹在过滤器中并不存在, 此时间段在从第 10 行到第 11 行的执行时间范围内, 因此可将此时间段中任一时间点作为可线性化点.

性质 2. 插入操作是可线性化的.

插入时最终将在两个候选哈希桶中找到空槽, 故通过 CAS 操作将指纹写入到过滤器的指令为可线性化点.

性质 3. 删除操作是可线性化的.

如果删除操作找到了待删除的指纹, 则通过 CAS 操作将指纹删除的指令为可线性化点. 若删除操作未找到指纹, 则线性化点的分析与查询操作返回元素不是集合成员的情况相同, 此处暂不赘述.

性质 4. 无锁并发布谷鸟过滤器是可线性化的.

由于无锁并发布谷鸟过滤器的各项操作均为可线性化的, 所以无锁并发布谷鸟过滤器是可线性化的.

3.5 进展性分析

性质 5. 执行一次指纹原子迁移操作的算法 4 将在有限步骤后完成, 否则其他操作将在有限步骤后完成.

当算法 4 不发生重试时, 算法 4 将在有限步骤后完成. 若算法 4 发生重试, 则说明源哈希桶内的元素发生了变化, 这说明其他线程执行的指纹插入、迁移或删除操作取得了进展.

性质 6. 若算法 3 执行完毕后没有迁移源哈希桶指纹, 则算法 3 执行的过程中一定有其他线程取得了进展.

若算法 3 因为目标哈希桶中没有空指纹槽而无法完成迁移, 则说明其他线程通过执行插入或迁移操作占用了目标哈希桶中原有的空指纹槽, 即其他线程取得了进展. 若算法 3 发现目标哈希桶中有空指纹槽, 但是通过 K-CAS 操作原子更新源哈希桶和目标哈希桶时失败, 则说明从读取目标哈希桶到原子更新过滤器期间, 有其他线程通过执行插入、删除、迁移等操作修改了源哈希桶或目标哈希桶, 即其他线程取得了进展.

性质 7. 查询操作将在有限步骤后完成, 否则其他操作将在有限步骤后完成.

查询操作只有当两个候选哈希桶的迁移计数器满足条件时才会发起重试. 如第 3.1 节所述, 查询操作的重试说明此时有其他迁移操作正在取得进展.

性质 8. 插入操作将在有限步骤后完成, 否则其他操作将在有限步骤后完成.

插入操作只有在 CAS 操作修改哈希桶失败后 (算法 2 第 9、14 行) 或指纹迁移成功完成后 (算法 2 第 17 行) 发生重试. 对于第 1 种情况, 之所以 CAS 操作修改哈希桶失败, 是因为有其他的插入、删除或迁移操作对哈希桶进行了修改, 即其他操作取得了进展. 对于第 2 种情况, 指纹迁移成功完成后一定会在一个候选桶中会产生空指纹槽, 此时插入操作会重试并在此空指纹槽中插入指纹, 即插入操作取得了进展.

性质 9. 删除操作将在有限步骤后完成, 否则其他操作将在有限步骤后完成.

删除操作只有在 CAS 操作修改哈希桶失败后 (算法 5 第 8、12、19 或 23 行) 或迁移计数器满足两阶段查询重试条件后将发起重试 (算法 5 第 27 行). 对于第 1 种情况, CAS 修改哈希桶失败说明有其他的插入、删除或迁移操作能够取得进展. 而第 2 种重试情况与查询操作的重试情况相同, 此处暂不赘述.

性质 10. 无锁并发布谷鸟过滤器为无锁过滤器.

本文所提过滤器的各项操作将在有限步骤后完成, 否则其他操作将在有限步骤后完成. 这符合无锁的进展性

要求. 因此无锁并发布谷鸟过滤器为无锁过滤器.

4 实验评估

本节首先介绍实验设置, 包括实验平台、对比算法、评估指标、数据集和参数设置. 之后本节从以下 4 个方面评估无锁并发布谷鸟过滤器的性能: (1) 评估线程数对算法吞吐的影响, 本节通过分别测量插入、查询和删除操作的吞吐随线程数量的变化实现这一目的; (2) 评估数据集大小对算法吞吐的影响, 本节通过测量插入、查询和删除操作的吞吐随数据集大小的变化实现这一目的; (3) 评估工作负载对算法执行时间的影响, 本节通过测量不同线程数量下执行时间随工作负载的变化实现这一目的; (4) 评估数据偏斜对算法执行时间和吞吐的影响, 本节通过分别在并发写入和读取场景下测量执行时间和吞吐随数据偏度的变化实现这一目的.

4.1 实验设置

实验平台: 本文使用一台配备了两颗 20 核 40 线程的 Intel Xeon Gold 5218R CPU 以及 64 GB 内存的标准服务器评估了不同算法的性能. 本文使用 C++17 实现无锁并发布谷鸟过滤器, 并将其开源 (https://github.com/wanghanchengchn/lock-free-concurrent-cuckoo-filter). 所有实验均使用 GCC 8.5.0 编译器编译并运行在 CentOS Linux release 8.2.2004 操作系统上. 此外, 编译器的优化等级为 O3.

对比算法: 本文使用 4 个最新的并发算法和无锁并发布谷鸟过滤器进行对比. 上述 4 个最新的对比算法分别为: 向量商过滤器 (VQF)[32]、单哈希块布隆过滤器 (OHBBF)[45]、基于细粒度锁的布谷鸟过滤器 (SLCF)[17]和基于全局锁的布谷鸟过滤器 (GLCF)[17].

评估指标: 本文在 3 个指标上评估了无锁并发布谷鸟过滤器的性能. (1) 吞吐量. 本文将其定义为单位时间可以成功执行的操作数量, 单位为百万次操作每秒 (MOPS). (2) 执行时间. 本文将其定义为执行一组操作所需的时间, 其中不同类别的操作数量按照比例设定, 例如插入操作数量:查询操作数量=3:1. 单位为毫秒 (ms). (3) 最大负载因子. 本文将其定义为指定过滤器的容量后, 对一个未插入任何元素的过滤器不断插入元素, 直到各线程均返回插入失败时, 实际插入的元素数量占过滤器容量的比例.

数据集: 本文在 4 个数据集上评估了算法的性能. (1) 模拟数据集. 本文随机生成了 5 个具有不同元素数量的数据集, 每个数据集中的元素数量分别为 2^{23}、2^{24}、2^{25}、2^{26}、2^{27}. 本文将具有 2^{26} 个不同元素数量的数据集作为默认数据集. (2) CAIDA 数据集[46]. CAIDA 数据集是一个收集自骨干网络流量的真实数据集. 本文使用的数据集中具有 483 481 个不同的元素. (3) YCSB 数据集[47]. 该数据集包含了使用 YCSB 生成器生成的 2 千万个不同的元素. (4) Wiki 数据集[48]. 该数据集是维基百科编辑提交的时间戳, 具有 90 437 011 个不同的元素.

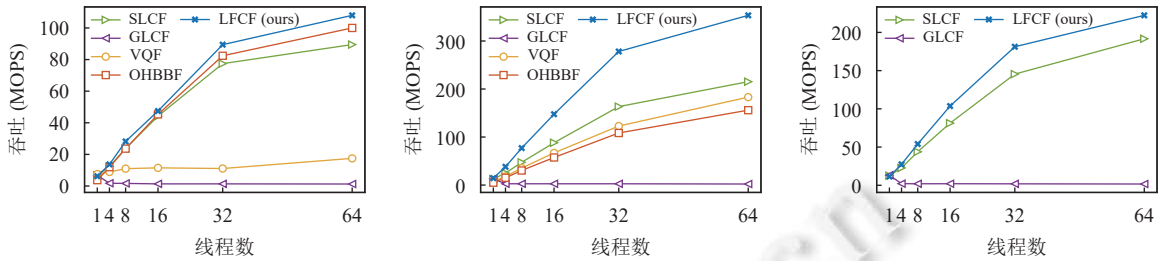
参数设置: 为了公平地比较不同算法的吞吐, 本文的参数设置需要保证不同 AMQ 数据结构的内存空间消耗、误报率相同. 在保证上述要求的前提下, 本文依照算法的默认参数设置设定了不同算法的参数. 所有算法的参数设置也已开源 (https://github.com/wanghanchengchn/lock-free-concurrent-cuckoo-filter).

4.2 评估线程数对算法吞吐的影响

本节评估了不同过滤器的插入、查询和删除操作的吞吐随线程数量的变化. 具体来说, 本节将不同过滤器的容量设置为 2^{26}. 之后使用不同数量的线程向不同过滤器中插入其容量 90% 数量的元素, 并测量插入过程中的吞吐. 此外, 本节还测量了使用不同数量的线程执行查询和删除操作的吞吐. 需要注意, 在插入过程中, 所有的过滤器均可填充到其容量的 90% 以上 (即所有过滤器的最大负载因子均可到达 90% 以上). 如图 4 所示, 无锁并发布谷鸟过滤器插入、查询和删除操作的吞吐平均是基于细粒度锁的布谷鸟过滤器 (SLCF) 插入、查询和删除操作吞吐的 1.15 倍、1.65 倍以及 1.21 倍. 并且当线程数量小于 32 时, 无锁并发布谷鸟过滤器实现了几乎线性的并发扩展性. 由于本文使用的机器每颗 CPU 只有 20 核 40 线程, 因此当线程数为 64 时, 由于跨 CPU 访问数据的问题, 并发可扩展性略微受到影响, 但依然展现出了亚线性的并发扩展性.

无锁并发布谷鸟过滤器之所以能实现上述亚线性扩展性能, 是因为本文提出的路径探查与元素迁移分离技术可让原子操作更快地完成. 这一设计降低了不同线程独占某一内存空间的时间, 继而提升了并发可扩展性. 需要注

意, 基于细粒度锁的布谷鸟过滤器也可以降低不同线程独占某一内存空间的时间, 因此也实现了较高的性能. 但是由于基于细粒度锁的布谷鸟过滤器每次加锁会产生一次缓存失效, 因此具有较高的加锁开销, 造成不同操作的吞吐低于本文提出无锁并发布谷鸟过滤器.

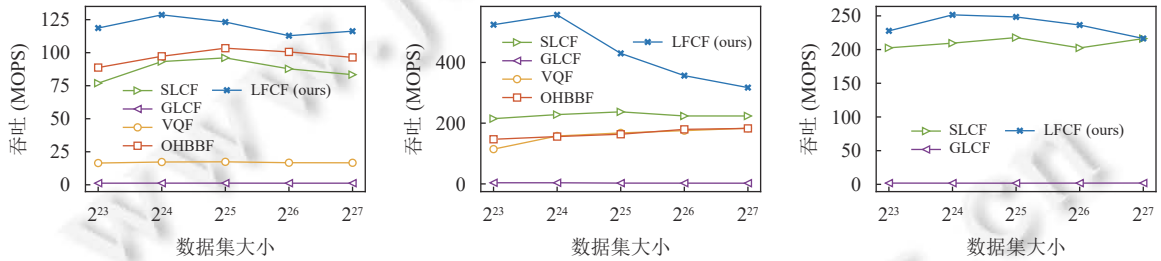


(a) 插入操作吞吐 (b) 查询操作吞吐 (c) 删除操作吞吐

图 4 插入、查询和删除操作的吞吐量随线程数量的变化

4.3 评估数据集大小对算法吞吐的影响

本节评估了不同数据集大小对插入、查询和删除操作吞吐的影响. 具体来说, 本节将不同过滤器的容量设置为 2^{23}、2^{24}、2^{25}、2^{26}、2^{27}, 之后使用 64 个线程插入不同数量的元素, 并测量插入过程的吞吐. 之后使用 64 个线程测量查询操作和删除操作的吞吐. 如图 5 所示, 无锁并发布谷鸟过滤器插入、查询和删除操作的吞吐平均是基于细粒度锁的布谷鸟过滤器 (SLCF) 插入、查询和删除操作吞吐的 1.37 倍、1.94 倍以及 1.13 倍. 本节的结果表明无锁并发布谷鸟过滤器针对具有不同数量元素的数据集均具有较高的吞吐.



(a) 插入操作吞吐 (b) 查询操作吞吐 (c) 删除操作吞吐

图 5 插入、查询和删除操作的吞吐随数据集中元素数量的变化

相比于其他基于锁的算法, 无锁并发布谷鸟过滤器具有更高的吞吐. 这是因为无锁并发布谷鸟过滤器具有可以显著降低锁竞争开销. 具体来说, 以基于细粒度锁的布谷鸟过滤器 (SLCF) 为例, 加锁操作占基于细粒度锁的布谷鸟过滤器 (SLCF) 查询操作总执行时间的 59% 以上. 而无锁并发布谷鸟过滤器可以消除高并发时锁竞争对过滤器性能的影响. 因此, 无锁并发布谷鸟过滤器可以显著提升吞吐.

4.4 评估工作负载对算法执行时间的影响

本节评估了工作负载对算法执行时间的影响. 具体来说, 本节测量了当插入操作和查询操作的比例分别为 3:1 和 1:3 时, 不同算法执行所有操作所需时间. 表 1 展示了不同算法在 CAIDA、YCSB、Wiki 数据集上的实验结果. 如表 1 所示, 相比于基于细粒度锁的布谷鸟过滤器 (SLCF), 无锁并发布谷鸟过滤器将两种工作负载的平均执行时间分别缩短了 13.56% 和 32.86%.

之所以能获得如此提升是因为本文提出的两阶段查询算法显著降低了无锁并发布谷鸟过滤器的查询开销. 因此, 无锁并发布谷鸟过滤的查询操作相比于插入操作具有更大的提升. 继而在查询操作比例较大的工作负载中无锁并发布谷鸟过滤器具有更高的性能. 注意, 在两类工作负载下, 基于全局锁的布谷鸟过滤器 (GLCF) 的执行时间随着线程数量的增加而提升. 这是因为基于全局锁的布谷鸟过滤器 (GLCF) 的全局锁机制使得其所有操作只能依

次串行执行, 无法随线程数量增加获得性能提升. 加之随线程数量增加, 不同线程对锁的竞争增加, 导致执行时间随线程数量增多反而不断增加. 此外, 在单线程场景下, 无锁并发布谷鸟过滤器为支持并发操作而引入的设计会产生无谓的开销, 从而导致单线程性能较低. 针对上述问题, 一种可行的解决方案为: 若过滤器使用场景仅需要单线程运行, 则可将无锁并发布谷鸟过滤器直接退化成传统单线程布谷鸟过滤器, 从而避免为支持并发而引入额外的开销.

表 1 执行时间随线程数量的变化

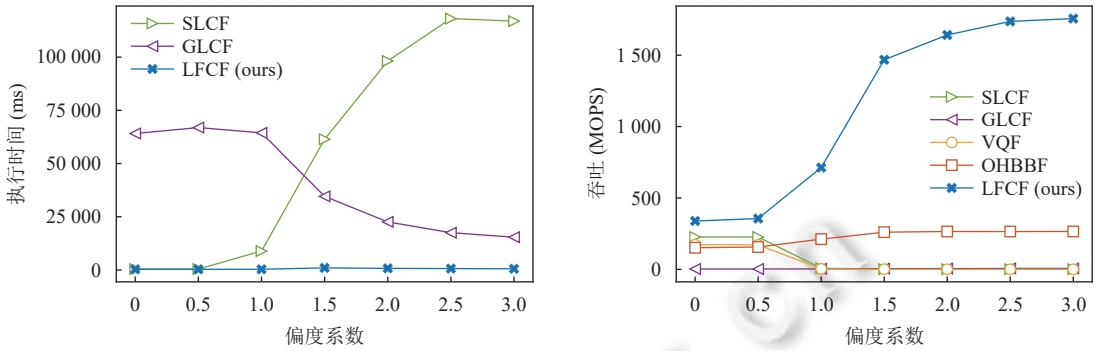
数据集名称	插入与查询操作数量比	算法名称	执行时间 (ms)					
			1 线程	4 线程	8 线程	16 线程	32 线程	64 线程
YCSB	3:1	LFCF (ours)	2222.85	**783.38**	**411.86**	**213.30**	**121.59**	**100.60**
		SLCF	1719.84	1045.97	555.62	299.17	163.48	128.75
		GLCF	**1281.81**	8930.98	11575.49	12266.68	13730.83	14853.98
		VQF	2047.23	2677.12	2145.90	1938.97	1918.55	1200.95
		OHBBF	6315.07	1851.64	940.04	518.28	290.82	244.42
	1:3	LFCF (ours)	5243.55	**1759.79**	**901.83**	**463.87**	**253.34**	**215.69**
		SLCF	4379.73	3274.39	1729.05	911.48	484.79	352.25
		GLCF	**3391.24**	24152.45	24548.34	26302.97	29656.75	33278.21
		VQF	6554.31	5789.88	3731.29	2760.81	2340.23	1581.44
		OHBBF	15317.49	4543.70	2328.67	1235.04	678.20	504.16
CAIDA	3:1	LFCF (ours)	36.50	**28.89**	**19.89**	**15.49**	**14.51**	**18.52**
		SLCF	34.52	39.51	21.28	18.07	18.78	19.97
		GLCF	**24.73**	236.32	267.32	282.00	296.52	348.96
		VQF	29.07	70.95	57.20	49.34	48.96	42.39
		OHBBF	105.71	46.73	24.53	15.62	15.30	19.29
	1:3	LFCF (ours)	70.72	**35.17**	**19.34**	**17.26**	**17.02**	**21.11**
		SLCF	83.38	81.29	44.51	25.92	23.16	26.89
		GLCF	**51.56**	421.04	454.87	476.90	530.59	620.48
		VQF	91.78	148.88	96.62	72.75	62.77	50.52
		OHBBF	273.29	95.57	54.21	30.56	17.57	26.13
Wiki	3:1	LFCF (ours)	9543.69	**3840.26**	**2367.05**	**966.47**	**599.98**	**527.76**
		SLCF	8847.99	4862.55	2511.50	1351.26	732.60	578.39
		GLCF	**7160.31**	38767.69	45332.15	46286.37	47670.45	60647.45
		VQF	14841.70	11957.73	9167.24	8136.14	8217.88	5506.31
		OHBBF	29781.57	9398.44	4674.94	2492.28	1770.57	1124.41
	1:3	LFCF (ours)	24449.55	**9159.73**	**4591.13**	**2408.55**	**1839.64**	**1272.58**
		SLCF	24325.10	14308.21	7515.03	3990.51	2625.32	1758.12
		GLCF	**21652.63**	109624.30	116815.23	119939.11	120100.52	155377.62
		VQF	45731.39	23477.13	15684.93	11778.93	10375.25	6863.91
		OHBBF	76435.16	24763.37	12127.59	6457.38	3597.11	2591.57

4.5 评估并发冲突对算法执行时间和吞吐的影响

本节评估了并发冲突对算法执行时间和吞吐的影响. 具体来说, 本节首先引入并发冲突比的概念. 并发冲突比被定义为多个线程同时读写相同数据的概率. 本节通过调整数据集的偏斜程度控制并发冲突比的大小. 具体来说, 本节生成一系列服从 Zipf 分布[49]且元素数量为 2^{26} 的数据集, 之后通过改变 Zipf 分布的偏度系数改变数据集的偏斜程度. 当偏度系数为 0 时, 不同数据出现频率满足均匀分布, 此时并发冲突比最小. 偏度系数越大, 不同数据出现频率差异越大, 数据集越偏斜, 并发冲突比越高.

本节将不同过滤器的容量设置为 2^{26}, 将实验线程数量设置为 64. 之后, 对于并发写入场景, 本节测量了不同算法执行混合写入工作负载 (插入操作数量:删除操作数量=1:1) 所需的执行时间随偏度系数的变化. 对于并发读取场景, 本节测量了不同算法的查询吞吐随偏度系数的变化. 如图 6 所示, 相比于基于全局锁的布谷鸟过滤器 (GLCF),

无锁并发布谷鸟过滤器将执行时间平均降低了 98.61%. 相比于基于细粒度锁的布谷鸟过滤器 (SLCF), 无锁并发布谷鸟过滤器将查询吞吐平均提升 16.24 倍.



(a) 并发写入执行时间 (b) 并发读取吞吐

图 6 执行时间和吞吐随偏度系数的变化

对于基于细粒度锁的布谷鸟过滤器 (SLCF), 随着数据集偏斜程度的增加, 线程间的锁争用加剧, 从而导致性能下降. 对于基于全局锁的布谷鸟过滤器 (GLCF), 由于任何数据访问均需要获取全局锁, 因此数据偏斜程度的变化并不会加剧锁争用程度. 反之, 随着数据偏斜程度的提高, 数据访问呈现出更好的局部性特征, 继而造成缓存命中率提高, 故性能有所提升. 本文提出的无锁并发布谷鸟过滤器可以避免锁争用带来的性能开销, 并可充分利用缓存局部性特征获得性能提升. 因此, 在并发读写场景下均表现出了最佳的执行时间和吞吐性能.

5 总结与展望

布谷鸟过滤器是一种应用广泛的近似成员资格查询数据结构. 设计并实现支持高并发的布谷鸟过滤器可以显著提升现有系统吞吐以及数据处理能力, 对提升系统性能意义重大. 本文提出了一个支持无锁并发的布谷鸟过滤器. 该过滤器通过两阶段查询协议、路径探查与元素迁移分离、基于多机器字比较并交换的原子迁移实现线程安全的查询、插入和删除操作. 理论分析和实验结果表明, 无锁并发的布谷鸟过滤器显著提升了最新算法的并发性能. 例如, 无锁并发布谷鸟过滤器的查询吞吐平均为使用细粒度锁的布谷鸟过滤器的查询吞吐的 1.94 倍. 后续工作将使用无锁并发技术优化更多数据结构的性能[50,51]. 此外, 还将考虑从存储角度出发, 利用非易失内存等新型存储硬件特性[52], 进一步提升过滤器数据结构性能.

References:

[1] Cohen E. All-distances sketches, revisited: HIP estimators for massive graphs analysis. In: Proc. of the 33rd ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems. Snowbird: ACM, 2014. 88–99. [doi: 10.1145/2594538.2594546]

[2] Dai HP, Li M, Liu A. Finding persistent items in distributed datasets. In: Proc. of the 2018 IEEE Conf. on Computer Communications. Honolulu: IEEE, 2018. 1403–1411. [doi: 10.1109/INFOCOM.2018.8486425]

[3] Mosharraf SIM, Adnan MA. Improving lookup and query execution performance in distributed big data systems using cuckoo filter. Journal of Big Data, 2022, 9(1): 12. [doi: 10.1186/S40537-022-00563-W]

[4] Vora AV, Hegde S. Keyword-based private searching on cloud data along with keyword association and dissociation using cuckoo filter. Int'l Journal of Information Security, 2019, 18(3): 305–319. [doi: 10.1007/s10207-018-0418-0]

[5] Zhao YK, Dai WC, Wang SR, Xi L, Wang SQ, Zhang F. A review of cuckoo filters for privacy protection and their applications. Electronics, 2023, 12(13): 2809. [doi: 10.3390/ELECTRONICS12132809]

[6] Morales D, Agudo I, Lopez J. Private set intersection: A systematic literature review. Computer Science Review, 2023, 49: 100567. [doi: 10.1016/j.cosrev.2023.100567]

[7] Kumar M, Singh A. Probabilistic data structures in smart city: Survey, applications, challenges, and research directions. Journal of Ambient Intelligence and Smart Environments, 2022, 14(4): 229–284. [doi: 10.3233/AIS-220101]

[8] Dai HP, Li M, Liu AX, Zheng JQ, Chen GH. Finding persistent items in distributed datasets. IEEE/ACM Trans. on Networking, 2020, 28(1): 1–14. [doi: 10.1109/TNET.2019.2946417]

[9] Zhao QC, Huang C, Dai LH. VULDEFF: Vulnerability detection method based on function fingerprints and code differences. Knowledge-based Systems, 2023, 260: 110139. [doi: 10.1016/J.KNOSYS.2022.110139]

[10] Yang JS, Jia WC, Gao Z, Guo ZH, Zhou Y, Pan Z. Cuckoo-store engine: A Reed-Solomon code-based ledger storage optimization scheme for blockchain-enabled IoT. Electronics, 2023, 12(15): 3328. [doi: 10.3390/electronics12153328]

[11] Sajitha M, Kavitha D, Reddy PC. An optimized clone node detection in WSN using cuckoo filter. SN Computer Science, 2023, 4(2): 167. [doi: 10.1007/S42979-022-01586-Z]

[12] The Apache software foundation. Bloom filter index. 2022. https://doris.apache.org/docs/1.2/data-table/index/bloomfilter/

[13] Honarkhah M, Talebzadeh A. HyperLogLog in presto: A significantly faster way to handle cardinality estimation. 2018. https://engineering.fb.com/2018/12/13/data-infrastructure/hyperloglog/

[14] Ray N, Yang FJ. How we scaled HyperLogLog: Three real-world optimizations. 2014. https://metamarkets.com/2014/engineering-hyperloglog-optimizations-for-real-world-systems/

[15] Ghemawat S, Dean J. LevelDB. 2023. https://github.com/google/leveldb

[16] Meta Inc. Rocksdb. 2023. https://github.com/facebook/rocksdb

[17] Gu R, Li SM, Dai HP, Wang HC, Luo YL, Fan B, Basat RB, Wang K, Song ZY, Chen SW, Wang BN, Huang YH, Chen GH. Adaptive online cache capacity optimization via lightweight working set size estimation at scale. In: Proc. of the 2023 USENIX Annual Technical Conf. Boston: USENIX Association, 2023. 467–484.

[18] Sutter H. The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb's Journal, 2005, 30(3): 202–210.

[19] Lü TG, Hong RC, He J, Hu SJ. Multimodal-guided local feature selection for few-shot learning. Ruan Jian Xue Bao/Journal of Software, 2023, 34(5): 2068–2082 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6771.htm [doi: 10.13328/j.cnki.jos.006771]

[20] Liu RC, Zhang JC, Luo YP, Jin PQ. Heterogeneous index for non-volatile memory. Ruan Jian Xue Bao/Journal of Software, 2022, 33(3): 832–848 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6456.htm [doi: 10.13328/j.cnki.jos.006456]

[21] Fu PT, Luo LL, Guo DK, Zhao X, Li SS, Wang HM. Jump filter: Dynamic sketch design for big data governance. Ruan Jian Xue Bao/Journal of Software, 2023, 34(3): 1193–1212 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6782.htm [doi: 10.13328/j.cnki.jos.006782]

[22] Li XZ, Andersen DG, Kaminsky M, Freedman MJ. Algorithmic improvements for fast concurrent cuckoo hashing. In: Proc. of the 9th European Conf. on Computer Systems. Amsterdam: ACM, 2014. 27. [doi: 10.1145/2592798.2592820]

[23] Rinberg A, Spiegelman A, Bortnikov E, Hillel E, Keidar I, Rhodes L, Serviansky H. Fast concurrent data sketches. In: Proc. of the 25th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. San Diego: ACM, 2020. 117–129. [doi: 10.1145/3332466.3374512]

[24] Arpaci-Dusseau RH, Arpaci-Dusseau AC. Operating Systems: Three Easy Pieces. North Charleston: CreateSpace Independent Publishing Platform, 2018.

[25] Abraham S, Peter BG, Greg G. Operating System Concepts. 10th ed., Hoboken: John Wiley & Sons, 2018.

[26] The Apache Software Foundation. Atomic instructions. 2023. https://brpc.incubator.apache.org/docs/rpc-in-depth/atomic-instructions/

[27] Bloom BH. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 1970, 13(7): 422–426. [doi: 10.1145/362686.362692]

[28] Xie K, Wen JG, Zhang DF, Xie GG. Bloom filter query algorithm. Ruan Jian Xue Bao/Journal of Software, 2009, 20(1): 96–108 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3458.htm [doi: 10.3724/SP.J.1001.2009.03458]

[29] Wang HC, Dai HP, Li M, Yu J, Gu R, Zheng JQ, Chen GH. Bamboo filters: Make resizing smooth. In: Proc. of the 38th IEEE Int'l Conf. on Data Engineering. Kuala Lumpur: IEEE, 2022. 979–991. [doi: 10.1109/ICDE53745.2022.00078]

[30] Fan B, Andersen DG, Kaminsky M, Mitzenmacher MD. Cuckoo filter: Practically better than bloom. In: Proc. of the 10th ACM Int'l on Conf. on Emerging Networking Experiments and Technologies. Sydney: ACM, 2014. 75–88. [doi: 10.1145/2674005.2674994]

[31] Voras I, Žagar M. Adapting the bloom filter to multithreaded environments. In: Proc. of the 15th IEEE Mediterranean Electrotechnical Conf. Valletta: IEEE, 2010. 1488–1493. [doi: 10.1109/MELCON.2010.5476244]

[32] Pandey P, Conway A, Durie J, Bender MA, Farach-Colton M, Johnson R. Vector quotient filters: Overcoming the time/space trade-off in filter design. In: Proc. of the 2021 Int'l Conf. on Management of Data. ACM, 2021. 1386–1399. [doi: 10.1145/3448016.3452841]

[33] Bender MA, Farach-Colton M, Johnson R, Karner R, Kuszmaul BC, Medjedovic D, Montes P, Shetty P, Spillane RP, Zadok E. Don't thrash: How to cache your hash on flash. Proc. of the VLDB Endowment, 2012, 5(11): 1627–1637. [doi: 10.14778/2350229.2350275]

[34] Pandey P, Bender MA, Johnson R, Patro R. A general-purpose counting filter: Making every bit count. In: Proc. of the 2017 ACM Int'l

Conf. on Management of Data. Chicago: ACM, 2017. 775–787. [doi: 10.1145/3035918.3035963]

[35] Wang ZQ, Pavlo A, Lim H, Leis V, Zhang HC, Kaminsky M, Anderson DG. Building a BW-tree takes more than just buzz words. In: Proc. of the 2018 Int'l Conf. on Management of Data. Houston: ACM, 2018. 473–488. [doi: 10.1145/3183713.3196895]

[36] Levandoski JJ, Lomet DB, Sengupta S. The BW-tree: A B-tree for new hardware platforms. In: Proc. of the 29th IEEE Int'l Conf. on Data Engineering. Brisbane: IEEE, 2013. 302–313. [doi: 10.1109/ICDE.2013.6544834]

[37] Leis V, Scheibner F, Kemper A, Neumann T. The ART of practical synchronization. In: Proc. of the 12th Int'l Workshop on Data Management on New Hardware. San Francisco: ACM, 2016. 3. [doi: 10.1145/2933349.2933352]

[38] Nguyen N, Tsigas P. Lock-free cuckoo hashing. In: Proc. of the 34th IEEE Int'l Conf. on Distributed Computing Systems. Madrid: IEEE, 2014. 627–636. [doi: 10.1109/ICDCS.2014.70]

[39] Kelly R, Pearlmutter BA, Maguire P. Concurrent robin hood hashing. arXiv:1809.04339, 2018.

[40] Harris TL, Fraser K, Pratt IA. A practical multi-word compare-and-swap operation. In: Proc. of the 16th Int'l Conf. on Distributed Computing. Toulouse: Springer, 2002. 265–279. [doi: 10.1007/3-540-36108-1_18]

[41] Arbel-Raviv M, Brown T. Reuse, Don't recycle: Transforming lock-free algorithms that throw away descriptors. arXiv:1708.01797, 2017.

[42] Kelly R, Pearlmutter BA, Maguire P. Lock-free hopscotch hashing. In: Proc. of the 2020 Symp. on Algorithmic Principles of Computer Systems. Philadelphia: SIAM, 2020. 45–59. [doi: 10.1137/1.9781611976021.4]

[43] Fatourou P, Kallimanis ND, Ropars T. An efficient wait-free resizable hash table. In: Proc. of the 30th on Symp. on Parallelism in Algorithms and Architectures. Vienna: ACM, 2018. 111–120. [doi: 10.1145/3210377.3210408]

[44] Williams A. C++ Concurrency in Action. 2nd ed., New York: Manning Publications, 2017.

[45] Prakasam E, Manoharan A. A cache efficient one hashing blocked bloom filter (OHBB) for random strings and the K-mer strings in DNA sequence. Symmetry, 2022, 14(9): 1911. [doi: 10.3390/SYM14091911]

[46] CAIDA. The CAIDA anonymized Internet traces dataset (April 2008–January 2019). 2019. https://www.caida.org/data/passive/passive_dataset.xml

[47] Copper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking cloud serving systems with YCSB. In: Proc. of the 1st ACM Symp. on Cloud Computing. Indianapolis: ACM, 2010. 143–154. [doi: 10.1145/1807128.1807152]

[48] Marcus R, Kipf A, van Renen A, Stoian M, Misra S, Kemper A, Neumann T, Kraska T. Benchmarking learned indexes. Proc. of the VLDB Endowment, 2020, 14(1): 1–13. [doi: 10.14778/3421424.3421425]

[49] Powers DMW. Applications and explanations of Zipf's law. In: Proc. of the 1998 Joint Conf. on New Methods in Language Processing and Computational Natural Language Learning. Sydney: ACM, 1998. 151–160.

[50] Yu JP, Chen HH, Qian JB, Dong YH. A heterogeneous bloom filter scheme in LSM tree based on hotness prediction. Acta Electronica Sinica, 2021, 49(11): 2090–2095 (in Chinese with English abstract). [doi: 10.12263/DZXB.20200945]

[51] Zhou Z, Fu WL, Song T, Liu QY. Fast URL lookup using parallel bloom filters. Acta Electronica Sinica, 2015, 43(9): 1833–1840 (in Chinese with English abstract). [doi: 10.3969/j.issn.0372-2112.2015.09.023]

[52] Li Q, Zhong J, Li X, Li Q. Memory management mechanism for hybrid memory architecture based on new non-volatile memory. Acta Electronica Sinica, 2019, 47(3): 664–670 (in Chinese with English abstract). [doi: 10.3969/j.issn.0372-2112.2019.03.021]

附中文参考文献:

[19] 吕天根, 洪日昌, 何军, 胡社教. 多模态引导的局部特征选择小样本学习方法. 软件学报, 2023, 34(5): 2068–2082. http://www.jos.org.cn/1000-9825/6771.htm [doi: 10.13328/j.cnki.jos.006771]

[20] 刘睿诚, 张俊晨, 罗永平, 金培权. 面向非易失内存的异构索引. 软件学报, 2022, 33(3): 832–848. http://www.jos.org.cn/1000-9825/6456.htm [doi: 10.13328/j.cnki.jos.006456]

[21] 符鹏涛, 罗来龙, 郭得科, 赵翔, 李尚森, 王怀民. 跳跃滤波: 一种面向大数据治理的动态数据摘要设计. 软件学报, 2023, 34(3): 1193–1212. http://www.jos.org.cn/1000-9825/6782.htm [doi: 10.13328/j.cnki.jos.006782]

[28] 谢鲲, 文吉刚, 张大方, 谢高岗. 布鲁姆过滤器查询算法. 软件学报, 2009, 20(1): 96–108. http://www.jos.org.cn/1000-9825/3458.htm [doi: 10.3724/SP.J.1001.2009.03458]

[50] 俞加平, 陈华辉, 钱江波, 董一鸿. LSM 树中基于热度预测的异构布隆过滤器方案. 电子学报, 2021, 49(11): 2090–2095. [doi: 10.12263/DZXB.20200945]

[51] 周舟, 付文亮, 嵩天, 刘庆云. 一种基于并行 Bloom Filter 的高速 URL 查找算法. 电子学报, 2015, 43(9): 1833–1840. [doi: 10.3969/j.issn.0372-2112.2015.09.023]

[52] 李琪, 钟将, 李雪, 李青. 基于新型非易失存储器的混合内存架构的内存管理机制. 电子学报, 2019, 47(3): 664–670. [doi: 10.3969/j.issn.0372-2112.2019.03.021]

王瀚橙(1998－), 男, 博士生, CCF 学生会员, 主要研究领域为数据索引和查询优化.

顾荣(1988－), 男, 博士, 研究员, 博士生导师, CCF 高级会员, 主要研究领域为大数据并行处理系统.

陈志鹏(2001－), 男, 硕士生, 主要研究领域为并发数据索引设计.

KIM Chaewon(1995－), 女, 硕士生, 主要研究领域为数据挖掘, 并发算法.

戴海鹏(1985－), 男, 博士, 副教授, 博士生导师, CCF 杰出会员, 主要研究领域为物联网, 数据挖掘, 边缘计算, 移动计算.

陈贵海(1963－), 男, 博士, 教授, 博士生导师, CCF 会士, 主要研究领域为无线网络, 并行计算, 算法设计.