2020 级

《大数据存储系统与管理》课程

课 程 报 告

选　题	**Cuckoo-driven Way**
姓　名	**付景洋**
学　号	**U202015424**
班　号	**计算机 2005 班**
日　期	**2023.5.29**

目 录

一. 选题背景

字典（dictionary，也被称为关联数组 associative array）的数据结构在计算机科学中有着广泛的应用，它能够维护一个元素键值（和相关属性）的集合，支持键值的插入、删除、查找和判定键值是否在集合中。哈希表被认为是一种能够高效地实现字典功能的数据结构，它通过计算哈希函数将所需查询的键映射到表中一个位置，可以从该位置直接获得元素的值，它的各种操作的平均时间复杂度为 O(1)，最差时间复杂度为 O(n)，空间复杂度为 O(n)。不同的哈希表实现方法在时间复杂度上有着常数量级的差异，然而在当下的各种云存储和数据库系统中，为实现更高效的操作减小该常数的值至关重要。

哈希表的设计方式分为哈希函数和哈希模式两个方面[2]。哈希函数决定了如何将较大的键值域映射到哈希表存储的有限空间中，当key1≠key2时 hash(key1)=hash(key2)被称为哈希冲突，实现一个较低哈希冲突率的哈希函数往往需要更多的时间，存在一个时间和冲突率的 trade-off，目前较为优秀的哈希算法有 Facebook XXHash。哈希模式决定了在发生哈希冲突后哈希表处理键值的行为，静态的哈希模式如开放寻址往往直接分配一个大的哈希表，动态的哈希模式如可扩展哈希(extendible hashing)在插入删除的过程中对哈希表进行了修改，这里存在时间和空间的 trade-off。

传统的哈希冲突解决在最坏情况下查找时间为 Ω(n), 而 Cuckoo hashing 的最坏情况下查找时间为 O(1)。Cuckoo Hashing 使用多个哈希函数和多个哈希表。对于插入操作，如果在某一个哈希表中的对应位置为空，则直接插入，如果所有哈希表对应位置都不为空，则选取一个哈希表中对应位置的元素进行驱逐，对被驱逐的元素重新进行哈希操作，就像杜鹃鸟占据别的鸟巢并把鸟巢中原有的蛋踢掉的行为。对于查找和删除操作，Cuckoo Hashing 的时间复杂度永远为 O(1)，因为对每个哈希表只会检查其中的一个位置。

Cuckoo Hashing 插入元素的操作与查询操作相比需要更大的时间开销，是改进和优化的关键。在插入踢除的过程中，如果一个元素在某个位置被踢除，但是在后续的操作中又重新占据该位置，那么就会产生踢除的无限循环，在实现中需要选择合理的踢除策略并及时检测是否有发生无限循环的可能。

二. 选题分析

2.1 Cuckoo 图

传统的 Cuckoo Hashing 方法使用 2 个哈希函数和哈希表，每个元素有两个候选位置。哈希表中的每一个位置看为一个点，将元素看为一条无向边连接两个候选位置对应的点，就可以直接生成一个二分图，如图 1 所示[3]，记 Tij(i=1,2)表示第 i 个表的从上到下第 j 个位置，例如 T12 的值为 16。

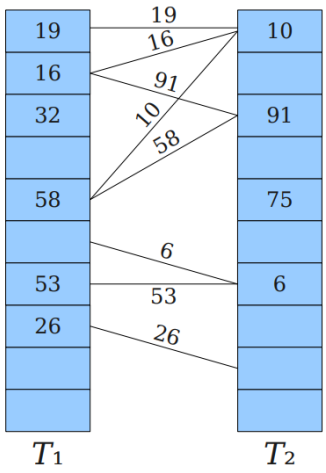

图 1

分析插入操作，假设需要插入元素 88，其候选位置为 T15 和 T27，如图 2 所示，向二分图中增加一条 T15 和 T27 相连的边。

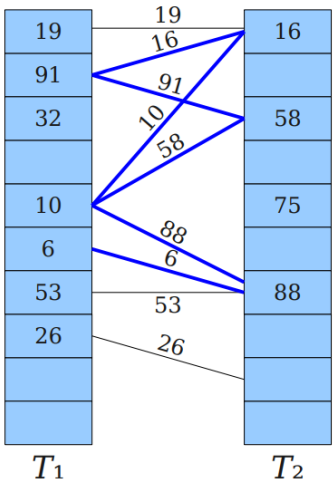

图 2

在图 1 中可以看到 88 的候选位置已经分别被 58 和 6 占据，第一次尝试踢除 58，58 踢除 91，91 踢除 16，16 踢除 10，10 又重新踢除 88，本次尝试失败，第二次尝试踢除 6，6 有空闲位置，尝试成功，得到图 2。一次踢除相当于走过二分图的一条边。在第一次尝试失败的模拟过程中可以发现数据的迁移过程组成了一个环，我们可以猜测在 Cuckoo 图中组成环的情况尝试必然失败。实际上，如果插入元素所在的连通分量有大于 1 个环，则插入必然失败，即能直接预见会发生无限循环。上述结论是直观的，如果连通分量中存在大于 1 个环，则连通分量的边数一定大于点数，因为没有环的连通分量有边数=点数-1，而增加第一条边只能产生一个环，因此产生大于 1 个环需要至少增加两条边，总待插入元素数量大于总候选位置数量，插入必然失败。上述结论说明了插入失败的充分性，而其必要性也成立，即如果插入元素所在的连通分量存在不超过 1 个环，插入必然成功，因为该连通分量的边数≤点数。

我们通过判断 Cuckoo 图中的连通分量的环的个数，在未插入前直接判定是否会发生无限循环。在实际的很多情况中，插入最终可以成功但是会走很多“冤枉路”，例如图 2 的插入中的第一次尝试，如果能够直接选择 6 踢除，可以把踢除的次数从 6 减少到 1，因此确定踢除的策略而非随机踢除能够减少数据迁移的次数从而提高效率。

在插入一个元素前，Cuckoo 图中该边（元素）对应的两个点（候选位置）有四种情况。情况 1：两个点不连通，且某个点所在的连通分量存在环，则选择另一个点踢除。情况 2：两个点不连通，且所在连通分量都存在环，则连接该边后连通分量存在大于一个环，无法插入。情况 3：两个点连通但不存在环，选择连通分量中点数少的踢除。情况 4：两个点连通且存在环，则连接改变后存在大于一个环，无法插入。

上述情况 3 是唯一的两个点都可以踢除的情况，其选择策略值得进一步考虑。因为最优踢除次数等于连通分量中所有空闲点到新加边对应点的最短路径距离的最小值，而选择连通分量中点数少的踢除只是考虑了连通分量中的最长路，依据的是一个很松的上界。更精确的解决方法当然是求出最短路径，然而任意一种最短路径算法都意味着每一次插入的较大的时间开销，违背了哈希函数的初衷，我认为这是不可接受的。

另外，我们需要判断连通分量的环的个数与 1 的大小，而仅仅是维护动态图的连通性问题，考虑到有插入删除的操作，并且需要在线回答查询，使用 LCT(Link Cut Tree)等数据结构都有至少 O(lgn)的时间复杂度[4]，因此，即使上述所有方法能够较为精确判定无限循环的情况，由于其较大的时间开销，只能在特定情况下使用。

2.2 另一种 Cuckoo 图

另一种自然的将 Cuckoo Hashing 转化为图论问题解决的方式为将元素和哈希表的位置都看成点，将元素对应点和待选位置的几个点连边[5]，这实际上也是一个二分图，所有元素对应点在左侧，哈希表位置对应点在右侧，而元素实际填入哈希表位置就是二分图的一个匹配。这种二分图的构建方式不局限于只有 2 个哈希函数和哈希表的情况。

要避免无限循环使插入成功，则要使左侧每个点都有匹配边，左侧完美匹配实际上是二分图最大匹配的特殊情况。在求二分图的最大匹配的增广路算法中，从左侧的未匹配点（要插入的元素）开始，经过一条由匹配边和非匹配边构成的交错路径，找到右侧的空闲位置（最后一个被踢除点最终所在位置）。我们可以给二分图定向，将未匹配边设为由左指向右，匹配边设为由右指向左，则问题转化为有向图从给定起点找一条简单路径走到某个未匹配的点。二分图增广路算法给图定向的过程，结合 Cuckoo Hashing 的实际意义，得到的增广路径的一条有向边恰好为由元素代表的点指向由该元素占据的位置代表的点。在插入过程中，Cuckoo Hashing 的踢除模式完美转化为求二分图的最大匹配，或者说，Cuckoo Hashing 的踢除和二分图求增广路径一定程度上本质是同一个问题。

二分图是否判断左边的每个点都能被匹配的方式是赫尔婚姻(Hall's Marriage)定理，对左侧点的任意子集的点个数小于等于与之相邻的右侧点个数，符合条件时 Cuckoo Hashing 能够成功插入。二分图求最大匹配方案有匈牙利算法和转化为网络流模型求最大流，具体转化方式为由源点向左侧点连容量为 1 的边，由左侧点连向右侧的边容量设为 1，右侧点向汇点连容量为 1 的边。网络流求完最大流后得到的残留网络就是上文中二分图定向后的图，也能给出 Cuckoo Hashing 的踢除方案。

2.3 传统 Cuckoo Hashing 的拓展

在[1]中的传统 Cuckoo Hashing 中只使用了 2 个哈希函数，每个位置只存储一个元素，对传统 Cuckoo Hashing 的拓展有增加哈希函数个数和增加一个位置中存储元素个数的方式。通过增加哈希函数个数，每个元素的候选位置增加，用上文网络流的模型思考，就是中间的边数增加，由源点向汇点的流量自然更加通畅。增加一个位置中元素个数，类似 cache 中设置组相联的做法，cache 中的 miss 率减少对应这里的 hash 冲突减

少。

如图 3 所示[6]，两个维度的拓展对有效存储率都有显著的提升，在我看来，有效存储率较高意味着发生无限循环的概率较小，否则需要进一步增大存储容量解决无限循环的问题。

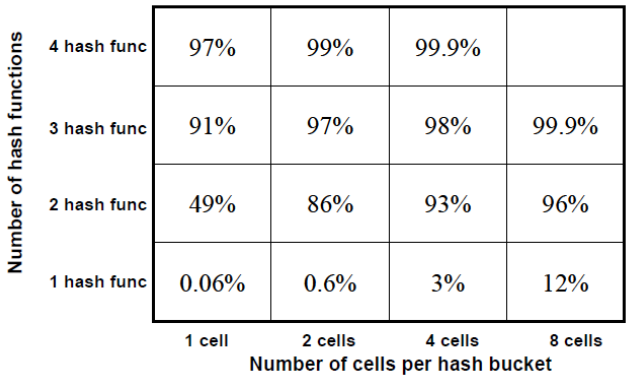

图 3

三. 总结

本次课程报告分析了 Cuckoo Hashing 判定 cuckoo 踢除操作中的无限循环的方案，简单思考了更优的选择策略，将 Cuckoo Hashing 的行为方式与二分图、网络流进行了类比，对于提高有效存储率，思考了增加哈希函数个数和增加哈希表宽两个维度的方案。

Cuckoo Hashing 的在表宽较大的情况下空间利用率较高，读操作的效率高，但插入操作复杂，特别时空间利用率较高时需要多次 cuckoo 完成一次插入，如果考虑到并发设计则加锁的方式将更为复杂。

在阅读查找资料的过程中，我的体会是 Cuckoo Hashing 已经有了广泛的研究和应用，我对 Cuckoo Hashing 算法的思考还很不成熟，另外较为遗憾的是我在较晚的时候才发现 libcuckoo 的开源实现[7]，没有进一步探索源代码并自己动手进行实践。

参考文献

[1] R. Pagh, F. Rodler, "Cuckoo hashing", Proc. ESA, 2001, pages: 121–133.

[2] A. Pavlo, "Hash Tables", https://15445.courses.cs.cmu.edu/fall2022/slides/07-hashtables.pdf, 2021

[3] K. Schwarz. "Cuckoo Hashing", http://web.stanford.edu/class/archive/cs/cs166/cs166.1216/lectures/07/Slides07.pdf, 2021.

[4] E. Demaine, "Dynamic graphs", https://courses.csail.mit.edu/6.851/spring21/lectures/L20.pdf，2012

[5] Fotakis, D., Pagh, R., Sanders, P., Spirakis, P. (2003). Space Efficient Hash Tables with Worst Case Constant Access Time. In: Alt, H., Habib, M. (eds) STACS 2003. STACS 2003. Lecture Notes in Computer Science, vol 2607. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-36494-3_25

[6] Erlingsson, Úlfar et al. “A cool and practical alternative to traditional hash tables.”, 2006

[7] Libcuckoo library. https://github.com/efficient/libcuckoo