2019 级

《物联网数据存储与管理》课程

实 验 报 告

姓　名　张睿

学　号　U201912633

班　号　计算机校交 1902 班

日　期　2022.04.14

目　　录

一、选题背景

Cuckoo Hashing 是一种用于解决表中散列函数值的散列冲突的方案，具有最坏情况下 O(1) 的查找时间，O(1)的删除时间。同时对于插入，使用均摊分析也有 O(1)的插入时间。

Cuckoo Hashing 的结构如下图所示。

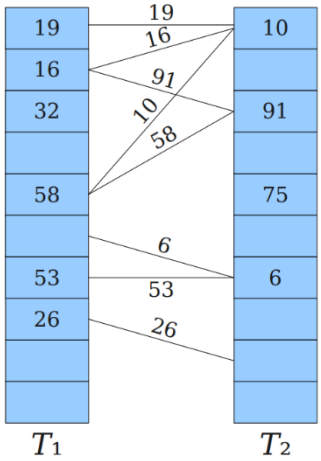


Figure 1 Cuckoo Hashing **结构** (Small13.pdf (stanford.edu))

其中，每条边为一个要插入的元素，对于一个要插入的元素，其插入位置有两个选择，分别为两个相互独立的哈希函数散列，如果两个位置都被占据，则任意踢出一个元素，该踢出的元素插入到另外一个散列的对应位置，若该位置被占据，则踢出该位置的元素并插入，如此迭代。

尽管 Cuckoo Hashing 具有很好的性能，但是也存在着问题。根据上面的描述，可以知道，当插入时，存在着死循环导致一直反复踢的情况使得 Cuckoo Hashing 性能损失。另一方面，由于反复踢会触发重哈希过程，而这个过程是非常费时间的，也导致了 Cuckoo Hashing 性能下降。

二、选题分析

根据图理论的推导，原本基础版的 Cuckoo Hashing 的理论最高负载因子为 50%左右，主要是由于环的出现，导致无法进一步插入元素。另一方面，由于环的出现而带来的重哈希过程是一个费时费力的操作，也是应该尽力避免的，所以优化的方向主要有两个：环与重哈希

对于环，可以考虑：

1. 是否可以提前检测到环的出现，以及时避免会带来环出现的操作。

2. 有没有一种比较好的数据结构或者算法，可以减少出现环的概率。

对于重哈希，可以考虑：

1. 重哈希是否是必需的，当出现环的时候，有没有另一种消耗更小方法去解决出现环的现象。

2. 对于重哈希，可否进行改进提升，使得其复杂度降低。

由于时间限制，本次实验主要聚焦于关于环的第二点，尽力去减少出现环的概率。

三、算法设计

直观想法是在原本 Cuckoo Hashing 的基础上进行改进，借鉴 chained hashing 的方法，对每一个位置增加额外的空间。如下图所示：

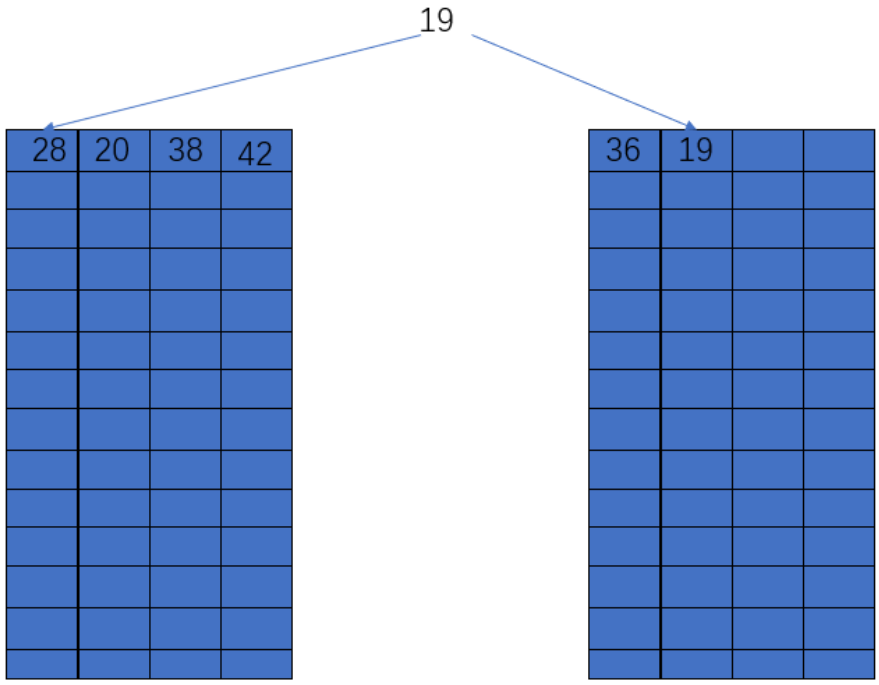


Figure 2 **改进后的** Cuckoo Hashing

针对每一个元素，相当于增加待选的位置，比如对于 19 这个元素来说，他会在两个散列共 8 个位置上进行查看，如果都满才选择一个元素去踢，否则直接插入到空位置。

在这种设定下，不管是插入，查找还是删除，都只有常数项复杂度的增加，但是经过测试，重哈希的时间，即反映出现循环的概率大大降低了。

四、实验与分析

实验部分主要从两个方面对改进后的 Cuckoo Hashing 进行测评：负载因子和时间性能，接下来分别叙述。测评环境如下表所示。

表 1 测评环境

操作系统	Ubuntu 20.04 LTS （WSL）
CPU 版本	Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80 GHz
Go 版本	1.18

需要注意的是，在所有实验中，对于成环的判断，采用踢出次数大于某一个常数便判断成环，实验中这个常数为 1000。

4.1 负载因子

对于负载因子，其定义为使用空间占总空间的比例，对于基础版 Cuckoo Hashing，其计算公式如下：

$$LF = \frac{OccupiedNum_1 + OccupiedNum_2}{2 * HashTableSize}$$

其中，OccupiedNum1 代表散列 1 中的占用的位置数量，OccupiedNum2 同理，HashTableSize 为散列的长度

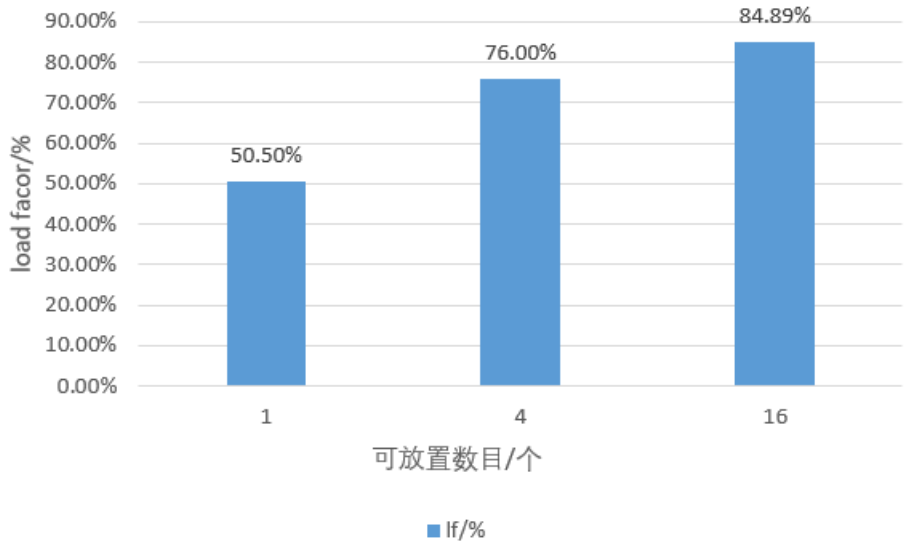
对于 Cuckoo Hashing Plus，其负载因子的计算公式如下：

$$LF = \frac{OccupiedNum_1 + OccupiedNum_2}{2 * HashTableSize * PlaceNum}$$

其中，PlaceNum 为某一个位置所拥有的可放置位置的数量。

数据规模为 1000000，最大负载因子随可放置位置的数量的变化情况如下图所示：

表 2 最大负载因子随可放置位置的数量的变化情况



其中，可放置数目为 1 时为基础版 cuckoo hashing。可以看到，随着可放置数目的增加，最大负载因子上升，这说明增加可放置位置确实可以提升 cuckoo hashing 的空间利用率。

4.2 时间性能

时间性能主要从两个角度测评：

1. 达到最大负载因子时的插入时间、重哈希时间、查询时间。
2. 插入时间、重哈希时间、查询时间随着负载因子的变化情况。

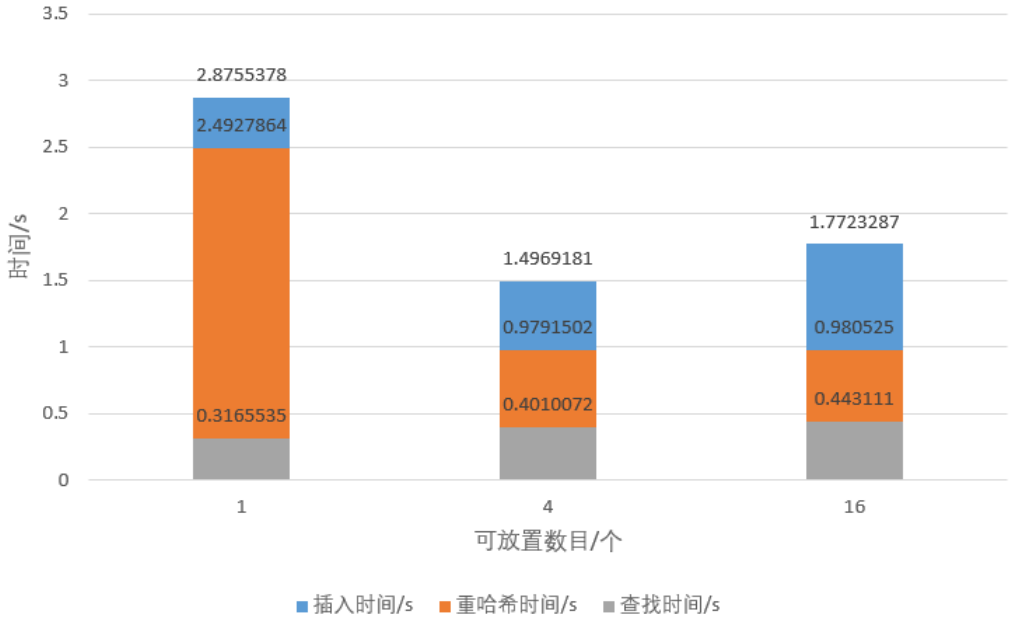
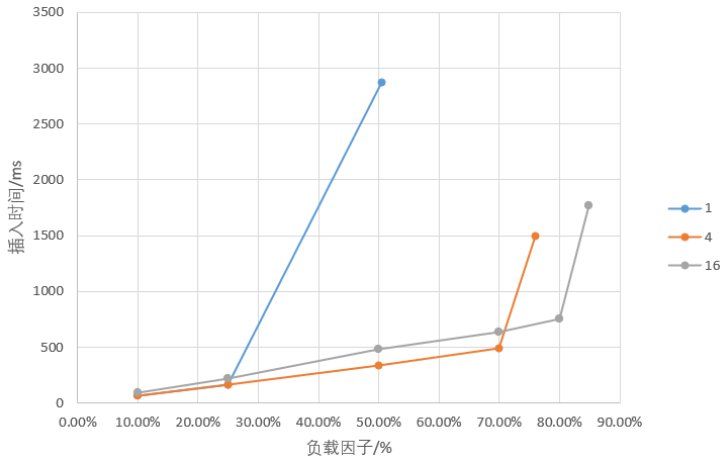


Figure 3 **达到最大负载因子时的各操作时间（设定同上）**

可以看到，三种操作中，最耗时的是插入操作，而对于插入操作，其大部分时间都用于了重哈希，这也是比较符合预期的。

值得注意的是，对于可放置数目增加的情况，由之前的的实验可知，其负载因子更高，也就是空间利用率更高了，而这些的总的数据规模都相同，所以也就是说插入到哈希表中的数据也就更多了，根据结果可以发现，尽管数据多了，但是插入时间和重哈希时间都有所降低。这进一步说明了可放置数目的增加大大减少了重哈希的概率，也就是成环的概率大大减少了。



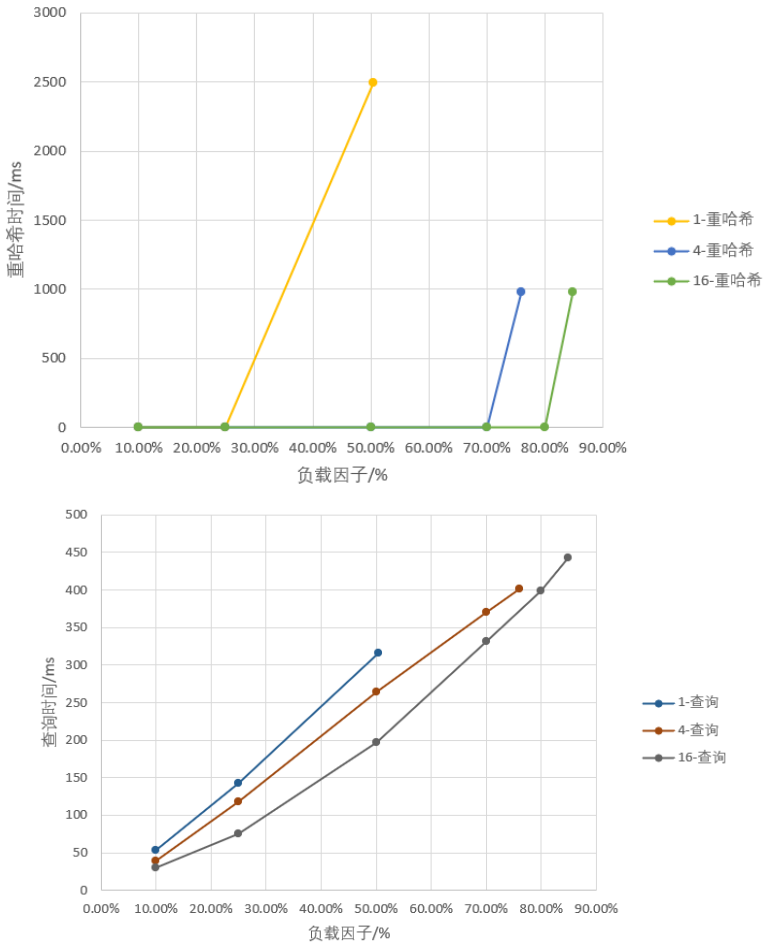


Figure 4 **插入时间、重哈希时间、查询时间随着负载因子的变化情况**

可以看到，对于插入时间，其实可放置数目的增加并不会显著增加插入的时间，可以说是几乎没有影响。

另一方面，比较有趣的是，结合插入时间和重哈希时间两张图我们可以发现，重哈希往往发生在负载因子快要达到最大的时候才发生。这可以给我们一个使用上的建议，即只要我们在最大负载因子的阈值内，进行插入查询删除等操作，就并不会受到重哈希的困扰，可以非常高效的使用 cuckoo hashing。

五、结论

本次实验对 Cuckoo Hashing 进行了优化，通过增加可放置位置数目来提高空间利用率，经过实验发现，在这种优化下，各种操作都不会显著增加时间复杂度，与此同时，却极大地提升了空间的利用率。

此外，在实验中还发现，重哈希过程只在快要接近最大负载因子时才会发生，即在实验设置下，快要接近最大负载因子时才有较大成环的概率。这个发现可以给我们使用 cuckoo hashing 带来一定的指导意义，即在某个阈值范围内进行使用。

当然，对于 cuckoo hashing 还有许多可以改进的方向，比如思考如何预测成环，是否可以通过增加哈希函数来增加位置等等。由于时间的关系，没有进一步地去探究，也是稍有遗憾。

除此之外，借这个机会，也学习了 go 的使用，由于时间的限制，没有完成最开始使用并发的期望，不过也算是熟悉了 go 的语法，还是有所收获的。

最后，感谢华宇老师一个学期以来教导！

参考文献

[1] R. Pagh and F. Rodler, “Cuckoo hashing,” Proc. ESA, pp. 121–133, 2001.

[2] Yu Hua, Hong Jiang, Dan Feng, "FAST: Near Real-time Searchable Data Analytics for the Cloud", Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), November 2014, Pages: 754-765.

[3] Yu Hua, Bin Xiao, Xue Liu, "NEST: Locality-aware Approximate Query Service for Cloud Computing", Proceedings of the 32nd IEEE International Conference on Computer Communications (INFOCOM), April 2013, pages: 1327-1335.

[4] Qiuyu Li, Yu Hua, Wenbo He, Dan Feng, Zhenhua Nie, Yuanyuan Sun, "Necklace: An Efficient Cuckoo Hashing Scheme for Cloud Storage Services", Proceedings of IEEE/ACM International Symposium on Quality of Service (IWQoS), 2014.

[5] B. Fan, D. G. Andersen, and M. Kaminsky, “MemC3: Compact and concurrent memcache with dumber caching and smarter hashing,” Proc. USENIX NSDI, 2013.

[6] B. Debnath, S. Sengupta, and J. Li, “ChunkStash: speeding up inline storage deduplication using flash memory,” Proc. USENIX ATC, 2010.