目录

1. 背景研究

在网络普及的大数据时代，愈发庞大的数据规模以及各式各样的内容结构，数据云存储系统实现高效的服务迎来个巨大的挑战。哈希表可以说是为解决这一挑战而诞生的数据结构。

哈希表对大部分领域都受用的基础数据结构，其中就包括数据库、网络、存储、与安全领域等。它通过计算哈希函数，将所要查询的键映射到一个位置来访问记录，且将查询操作的时间复杂度控制在 O(1)级别，也是因此受到了各个领域的广泛使用。

哈希函数在操作中可能会把不同的键映射到相同的位置，从而产生了哈希冲突。解决哈希冲突的方法有很多。比如，较为传统的方法有：

1. 开链法：
   把所有的被映射到同个位置的键以链表的方式链接并存放，哈希表中的每个位置则存放链表的表头
2. 开放寻址法：
   当映射位置不为空时，按规则循序检查下一个空位，找到后插入元素。进行查询操作时也是如此。

以上这些传统解决方法只适用于单个哈希函数，且进行操作时的时间复杂度也会变得不稳定。

Cuckoo hashing 是改良后的哈希表的一种。Cuckoo hashing 与传统的哈希表的不同在于，使用两个或以上个哈希函数，且给每个元素提供多个候选位置，根据规则挑选其一。具体流程如下：

1. 通过对应的哈希函数分别进行计算待插入元素 x 可以映射的位置 h1(x)，h2(x)
2. 根据规则选择适合的位置，进行插入操作：
   - 两个映射位置同时为空，任选其一；
   - 一个位置为空，一个位置不为空，选择为空的位置；
   - 两个映射位置同时不为空，任选其一，踢掉原本的元素后进行插入操作；
   - 被踢掉的元素须在之前计算好的映射位置中再次根据规则选择适合的位置重新插入。

3. 直至所有元素都找到位置存放。

根据位置选择规则的随机性，不难推测出在位置选择这一环节，若有大批数据需要理的情况下，很可能会在多个元素会陷入无限踢出和插入的死循环，即插入失败，则反复进行 rehash 操作，知道所有元素插入成功为止。

综上所说，可以见得 Cuckoo hashing 带来高效查询的前提是牺牲了插入的效率的。为此，Cuckoo hashing 需要得到一定的改良及优化，来减少插入的开销。其中，选择位置插入这一步骤尤为关键，我们需要尽可能避免死循环的出现。

会影响 Cuckoo hashing 性能的因素可以大致分为以下三点：

1. 插入操作中的循环的踢出；
2. 插入或查询失败造成的开销；
3. 多个桶的查询开销；

本次报告以第一点为重点，主要考虑如何降 Cuckoo hashing 操作中死循环出现的概率。为此，改进方式如下：

1. 设置 maxloop，防止死循环
   - 在 Smart Cuckoo 和 Necklace 中，他们尝试预先判断出循环的次数且设定 maxloop 值，在循环次数达到 maxloop 值后判断为无效存储，进行 rehash。这样可以从源头上阻止无限循环的出现。
   - 在 MinCounter 中，他们尝试减少重复的踢出过程，也就是说 A 桶中的内容在前一次循环中被选择踢出后，A 桶中的内容在被选中的概率就会降低。这样会使得更多的桶被查询，一次提高查询到空桶的概率。
   - 在 Blocked Cuckoo Hash Tables(BCHT)中，他们给每个桶分配了多个槽，槽中的关联性会提高灵活度，以此减少哈希冲突的出现
   -
2. Multi-copy Cuckoo（MC Cuckoo）

   MC Cuckoo 与传统的 Cuckoo hashing 不同，它不再选择插入哪一个桶，而是插入全部候选桶。此外，MC Cuckoo 会给每个桶匹配一个计数器，用来追踪元素的实时备份数量。用冗余数据与特定来规避插入操作中的哈希冲突所带来的死循环。

2. Cuckoo Hashing 的设计

2.1 基本 Cuckoo Hashing 的设计

Cuckoo hashing 是由[1]第一个提出的动态变化的多选哈希字典。基本的 Cuckoo Hash 是由一个哈希桶和至少两个哈希函数组成的。一般情况下，Cuckoo hashing 会由一个哈希桶和两个哈希函数组成，这意味着经过计算后，每个元素会拥有两个可插入位置。也就是说，在插入操作中元素只会插入到这两个位置中的其中一个；在查询操作中，要查询该元素时也只需要查询这两个位置即可。

2.1.1 插入

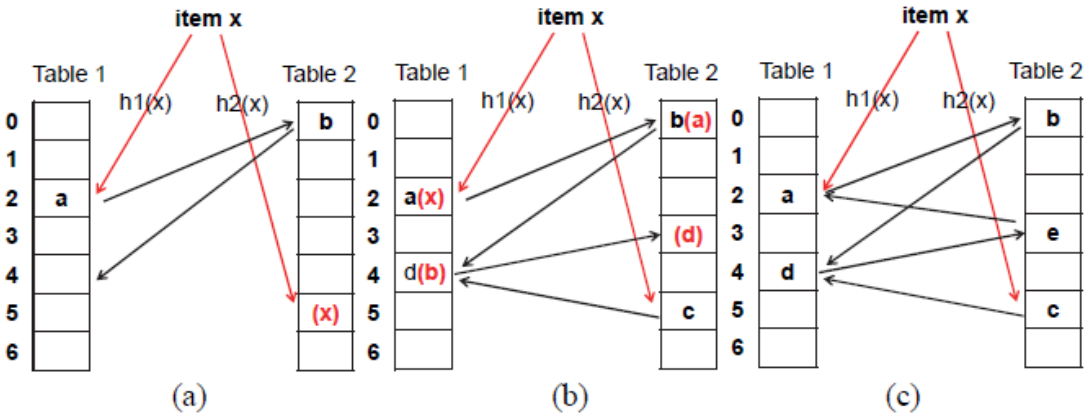


图 1 基本的 Cuckoo Hashing

如上图所示，常见的基本 Cuckoo Hashing 的数据结构有两个哈希表，每个元素 x 经过哈希函数的计算可在各哈希表中得一个位置（例子中共两个）。选择规则如下表：

H1	H2	选择
空	空	H1 | H2
空	非空	H1
非空	空	H2
非空	非空	H1 | H2

表 1 基本 Cuckoo Hashing 的插入选择

需要注意的是当两个哈希表的候选位置皆为非空时，任意选择一个位置后，当前元素会将原本存放在该位置的元素（之后都称原元素）踢出后再插入当前元素。而原元素则需要在自己其他的候补位置中再次遵循规则选择位置插入。图 1（b）所示中，元素 x 选择了踢掉元素 a，而元素 a 则需要到自己另一个候补位置去，因此踢掉了元素 b，同样的元素 b 也因此踢掉了元素 d，而元素 d 的另一个候补位置为空，则刚好存放元素 d，至此所有元素都找到了属于自己的位置。可是，如果一开始元素 x 选择踢掉的原元素不是元素 a 而是元素 c 存储效率将会有所提高。但因为选择的算法是随机的，因此这部分的开销是无法避免的。

在元素数量少且存储位置足够多的情况下，哈希冲突都可以被解决。但如果在相同的存储位置数量相同但元素数量急剧增加的情况下，哈希冲突无法被解决的概率将大大提高。元素踢出循环将一直无限循环下去，如图 1（c）所示，元素 x 无论选择替换哪个原元素，结果都是陷入无限循环中。解决这一问题的方法之一就是设置循环次数阈值 maxloop。其伪代码如下算法 1 所示：

算法 1 插入

输入：待存储元素 item，键 key，循环次数阈值 maxloop，循环计数器 count

1：$\boldsymbol{i_1 = hash(key)}$

2：$\boldsymbol{i_2 = hash(key)}$

3：if 在 bucket[$\boldsymbol{i_1}$] == null ||在 bucket[$\boldsymbol{i_2}$] == null

4：　在 null bucket 里 insert item

5：else if 在 bucket[$\boldsymbol{i_1}$] == ！ null ||在 bucket[$\boldsymbol{i_2}$] == ！ null

6：　随机选择一个 bucket 踢掉原元素后 insert item

7：　while（count<maxloop）

8：　　为被踢的原元素找其他位置

9：　end while

10：end if

设置 maxloop 限制每次重复循环的次数，从而主动结束无法停下的无限循环后，重新进行整个插入流程。每一次的插入失败和 rehash 都会造成很多开销。

2.1.2 查询

Cuckoo hashing 是一个查询开销相对稳定的数据结构，这是因为特殊的插入规则让其的时间复杂度维持在 O(1)。当需要查询一个元素是否存在在哈希表中，或者在某个哈希表中查询一个元素对应的值时，都需要将改元素的键分别进行哈希转换，找到各个表中的位置后进行查询。其伪代码如下算法 1 所示：

算法 2 查询
输入：查询对象的键（key）
1: $\boldsymbol{i_1 = hash(key)}$
2: $\boldsymbol{i_2 = hash(key)}$
3: if 在 bucket[$\boldsymbol{i_1}$]中找到 key ‖在 bucket[$\boldsymbol{i_2}$]中找到 key
4：　命中
5：else
6：　不命中
7：end if

2.2 Multi-copy Cuckoo Hashing 的设计

Multi-copy Cuckoo Hashing（后续称做 MC Cuckoo）是由计数器与哈希表组成的。与传统 Cuckoo hashing 最大的不同在于，MC Cuckoo 的算法下元素不做选择，而是插入到所有候选位置中，并在一一对应的计数器表中存入数量。如下图 2 所示：

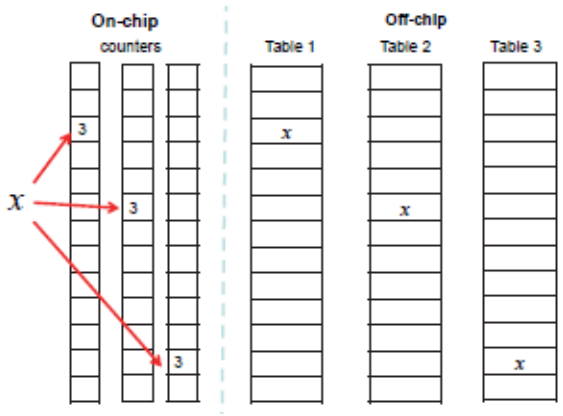


图 2 MC Cuckoo 的数据结构

哈希表和基本的 Cuckoo hashing 表所提供的功能基本一致。计数器表记录了与之对应的哈希表中元素在全局哈希表中的数量。如图 2 中所示 item x 在全局哈希表中有三个，因此与之对应的计数器表中都存入计数器值为 3。

2.2.1 插入

为了将冗余数据带来的灵活度最大化，每个元素都需要尽可能多的冗余数据。这也意味着，我们需要一个算法来平衡元素之间的冗余数量，使得数据之间的碰撞数量变少。当且仅当所有的计数器值都不为空且为 1 时，才会真正的出现哈希冲突。因此，我们需要尽可能的降低各个元素的数量降到 1 的速度。

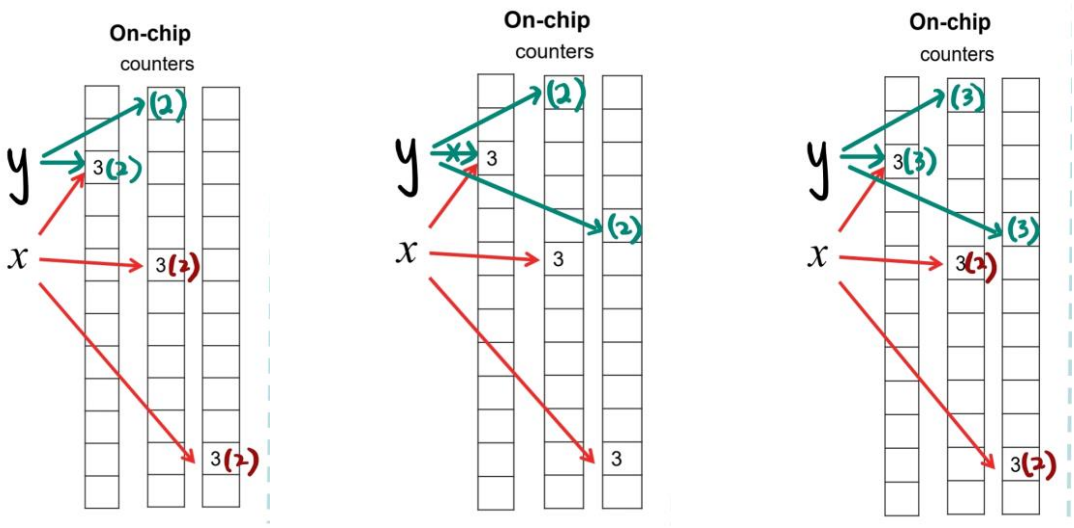
如图 2 所示，经过哈希函数的计算可得元素 x 在三个哈希表中分别可存的位置。与基本的 Cuckoo hashing 不同，MC Cuckoo 不进行选择，而是直接将元素插入所有位置，且在与哈希表一一对应的计数器表中各自存入 x 的数量。

当第二个元素 y 需要进行插入操作时，也同样需要经过哈希函数的计算得到元素 y 在各个哈希表中的位置，然后识别对应位置的计数器值，根据规则进行插入，插入操作结束后，再更新计数器表上的值。

插入规则如下，对应位置的计数器值：

1. 若为空，直接插入
2. 若不为空，且大于 1，根据特定算法决定是否踢出原函数存入该位置
3. 若不为空，且等于 1，放弃插入该位置

规则 2 可以这么解释，经过哈希函数计算得元素 y 有两个候选位置，其中 h1 对应的计数器值为 3（原元素数量）时，元素 y 就会踢掉该原元素，如图 3（a)。这是因为原元素被踢掉后依旧还有冗余的元素，并且元素 y 亦可避免元素数量（计数器值）为 1 的情况。但，如果经过哈希计算的元素 y 可得位置有三个，且其中一个位置计数器值为 3，元素 y 就不会踢掉该原元素。因为不管踢掉与否，对两个元素都没有影响。比例只是 2：3 和 3：2 的差别，对全部完全没有影响，如图 3（b）没踢掉原元素，图 3（c）踢掉原元素。



（a） （b） （c）

图 3 MC Cuckoo 规则 2 解释

以上插入规则的实现成功避免了误删数据的可能性，提高了数据的安全性。其伪代码如下算法 3 所示：

算法 1 插入
输入：待存储元素 item，键 key，计数器值 countervalue
1：$\boldsymbol{i_1 = hash(key)}$
2：$\boldsymbol{i_2 = hash(key)}$
3：$\boldsymbol{i_3 = hash(key)}$
4：if 在 counter[$\boldsymbol{i_1}$]==null ||在 counter[$\boldsymbol{i_2}$]==null ||在 counter[$\boldsymbol{i_3}$]==null
5：　在 null bucket 里 insert item
6：　在 counter 里 insert counter value
7：else if 在 bucket[$\boldsymbol{i_1}$]!=null ||在 bucket[$\boldsymbol{i_2}$]!=null ||在 bucket[$\boldsymbol{i_3}$]!=null
8：　比较 counter[i]value 和自己的 counter value 后选择是否踢掉。
9：　updated countertable 里的 countervalue
9：end if

2.2.2 查询

在 MC Cuckoo 的查询操作中，计数器表可以起到一个非常好的作用，不仅可以提高查询的效率，还降低了访问主存的概率缩短了查询的时间。

其一，在查询过程中可以通过计数器表快速判定查找元素是否存在表内。由于位置只要被元素插入后，不论是否被踢与否，其计数器值都不会是 0。因此，只要改位置的计数器值为 0，即可判定该元素还未插入，就可以不用到主哈希表中查询了。虽然存在假阳性错误的概率，但只要到主表里查了也会发现元素不存在，也许还未插入。

其二，减少了对存在的 item 查询时的内存访问。由于只要有存入元素的位置对应的计数器值都大于 0，因此在进行查询时，我们可以直接筛掉计数器值为 0 的位置。除此之外，由于同一个元素占有的所有位置对应的计数器值都会是相同的，因此只要将候选位置按照计算器值分组，组内位置数量为 s，本次查询目标的位置的计数器值为 c，若 s<c,则可以直接不查改组。对于其他计数器值相同的位置，只需要查询一个即可返回正确答案。

由此可见，简单有效的逻辑可以减少搜索的范围的同时还可以减少不必要的主存访问。在实际操作中，可以在大量数据中查询中控制访问主存次数在 0 到 1 次之间内找到目标元素，尤其适用于完全加载后的哈希表。

综上所述，MC Cuckoo 的查询操作规则如下：

1. 如果任何候选位置的计数器值为 0，则跳过所有位置并返回负值
2. 根据非 0 的候选位置的计数器值进行分组，跳过所有大小小于查询目标计数器值的组
3. 对于剩下的组，组内大小为 S，目标查询的计数器值为 V，检查 S-V+1 个位置，查询成功则返回该元素值，否则返回负值。

2.2.3 Stash

在 MC Cuckoo 中使用了 stash 作为插入失败的元素的存放地，以此避免需要更大开销的 rehash。MC Cuckoo 的方式是在主哈希表中增加大小为 1bit 的 flag，其工作原理类似于 Bloom Filter，在元素插入主哈希表失败后，就会把该元素在主哈希表中所有候选位置的 flag 更新为 1，而后将元素存入 stash 中。

这也意味着，当在进行查询工作中在主表中查询不到目标元素时，可以先检查该候补元素的 flag 值。当所有 flag 值为 1 时，则表示 stash 被启动，目标元素在主表中插入失败后存放在 stash 中，从而跳转到 stash 表中查询。如若 flag 值不全为 1，则表示 stash 并未被启动，因此也不需要在 stash 中查询，可认定该元素可能从未被插入，或插入后被删除。

并且，在 MC Cuckoo 中，一旦启动 stash 后，计数器表将不会再更改。这是因为触发 stash 意味着主表所有位置都已被占用且不可以被替换（计数器值都为 1），也就是说主哈希表已经没有空位支持该元素存放，同时意味着主哈希表中的所有元素都是唯一且不可被替代的。

3. Cuckoo Hashing 的实现

3.1 基本 Cuckoo Hashing 的实现

3.1.1 数据结构

CuckooHash 类使用两个 Python 列表来模拟两个哈希表：self.__table1：第一个哈希表、self.__table2：第二个哈希表。每个哈希表的元素是一个键值对，其中键是插入的键，值是与键相关联的数据列表。

哈希函数：

__h1(key)：使用 BitHash 计算键的第一个哈希值。

__h2(key)：使用 BitHash 和第一个哈希值作为种子来计算键的第二个哈希值。

插入操作：

1. 检查键是否已存在于其中一个表中。
2. 如果键存在，则将新数据追加到相关的数据列表中。
3. 如果键不存在，则根据哈希值确定插入到哪个表。
4. 如果任一表的装填因子超过 50%，则调用__growTables 方法进行表扩容。
5. 如果插入时遇到死循环，将重新哈希所有键。

查找操作：

1. 使用__h1 和__h2 方法计算键的哈希值。
2. 在两个表中查找键
3. 如果找到，返回表索引的和相关数据；否则，返回“None”

删除操作：

1. 查找键的位置
2. 将该位置设置为“None”
3. 更新哈希表键的数量

维护：

__rehash：当插入操作遇到死循环是，会重新哈希所有键

__growTables：扩大表的大小并重新插入所有键

3.1.2 性能和扩展性

冲突解决：

Cuckoo Hashing 通过多个哈希函数和双表结构有效地解决了哈希冲突，提高了检索效率。

哈希函数质量：

BitHash 函数提供了 64 位的随机哈希值，有助于减少冲突。但在某些情况下，可能需要考虑使用更复杂或更高质量的哈希函数。

测试：

测试用例在 test_cuckoo.py 中实现，包括插入操作、查找操作、删除操作、表扩容、数据追加、随机操作

- 插入操作：检查是否正确插入所有键。
- 查找操作：检查所有插入的键是否都能被正确查找。
- 删除操作：检查删除后表是否为空。
- 表扩容：在插入大量数据时检查是否成功扩容。
- 数据追加：检查相同键的数据是否被正确追加。
- 随机操作：随机插入和删除数据，并检查表的大小是否正确。

运行结果：

```
Microsoft Windows [Version 10.0.19045.4291]
(c) Microsoft Corporation. All rights reserved.

C:\Users\pings\Desktop\code1>py -u C:\Users\pings\Desktop\code1\test_cuckoo.py
============================= test session starts =============================
platform win32 -- Python 3.12.2, pytest-8.1.1, pluggy-1.5.0 -- C:\Users\pings\AppData\Local\Programs\Python\Python312\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\pings\Desktop\code1
plugins: anyio-4.3.0
collected 6 items

test_cuckoo.py::test_insert PASSED                                       [ 16%]
test_cuckoo.py::test_find PASSED                                         [ 33%]
test_cuckoo.py::test_delete PASSED                                       [ 50%]
test_cuckoo.py::test_growHash PASSED                                     [ 66%]
test_cuckoo.py::test_appendData PASSED                                   [ 83%]
test_cuckoo.py::test_random PASSED                                       [100%]

============================== 6 passed in 1.00s ==============================
```

图 4　Cuckoo hashing 的运行结果

4. 总结

在本次课程中，我学习到了有关很多关于哈希的不同使用方式及其优缺点。本次的课程报告也让我对 Cuckoo Hashing 有了更多的了解。通过学习和实现 Cuckoo Hashing，让我自己对数据结构有了更深的理解与认识。

在这篇报告中出了基本的 Cuckoo hashing，我还学习到了一个算是比较小众的 multi-copy Cuckoo。其实在课上老师公布课题的时候，对于 Cuckoo 无限循环的解决办法，我想到的都是一些很普遍的，比如通过 BFS 去搜索、设置时间限制判定循环、扩容等等。后来也突发奇想，如果利用计数器来标记位置会不会是一个更好的办法。再后来看到了 Mc Cuckoo 的文献，更加肯定了自己的想法不是异想天开。Mc Cuckoo 的冗余做法其实更让我醍醐灌顶，其实我们可以 think out of the box，不用只关注在现有条件下怎么优化，而是往前退一步，他是否有更好的做法。

总而言之，在这次的课程报告的完成过程中，我学到了很多专业知识的同时，也接触了文献阅读，更是学会了站在更远的地方看待问题可能会有不一样的答案的态度。Cuckoo hashing 是一个很有趣的数据结构，但是，受限于我目前的能力有限且在有限的时间内需要兼顾的东西太多，没有办法完成 Mc Cuckoo 的实现以及其与基本 Cuckoo 的性能比较。本篇报告虽然有很多瑕疵，但这也提醒着我在学有余力的时候，可以回过头来更深入的学习及实现更完整的内容实践。非常感谢老师们的教导，是一堂会一直记得的课。

参考文献

[1] Dagang Li Rong Du. Ziheng Liu, Tong Yang, Bin Cui Multi-copy Cuckoo Hashing [Report]. - Macao, China : IEEE, 2019.

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