大数据存储系统与管理课程报告

Cuckoo-driven Way

院系	计算机科学与技术
专业班级	大数据 2102
姓名	朱鑫材
学号	U202115636
指导老师	华宇

目录

1 背景

Cuckoo Hashing 是一种计算机科学中用于实现哈希表的算法。它由 Rasmus Pagh 和 Flemming Friche Rodler 在 2001 年提出。这种算法的名字来源于杜鹃鸟的习性，杜鹃鸟会将其他鸟的蛋从巢中推出，然后在巢中产下自己的蛋。 Cuckoo Hashing 的基本思想是使用两个哈希函数，而不是一个。

当插入一个新的元素时，首先会使用第一个哈希函数计算位置，如果该位置已经被占用，那么就会使用第二个哈希函数计算位置。如果第二个位置也被占用，那么就会将原来的元素”驱逐”，并插入新的元素。这个过程会一直重复，直到找到一个空的位置，或者达到预设的最大尝试次数.

Cuckoo Hashing 的优点是查找时间是常数的，不会随着哈希表的大小或者元素的数量而变化。此外，由于使用了两个哈希函数，Cuckoo Hashing 的负载因子可以达到约 50%，这意味着哈希表的一半以上的位置都被元素占用，这比传统的哈希表要高。

然而，Cuckoo Hashing 也有一些缺点。首先，插入操作可能需要多次尝试，特别是当哈希表接近满载时。其次，在进行 “驱逐” 操作时, 容易陷入无限循环的处境, 使得插入效率大幅下降. 另一方面, 在面临数据爆炸的挑战时，我们也迫切需要提高 Cuckoo Hashing 的负载因子, 以提高存储效率.

在这片报告中, 笔者将从 **hash 函数数目**, **block cuckoo hashing**, **stash cuckoo hashing** 三个可行策略进行研究, 探索提高负载因子以及提高插入效率的策略. 并尝试通过短路径环检测方法来减少无限循环的概率

2 Cuckoo Hashing

2.1 Cuckoo Hashing 的基本思想

hashs: hash 函数集合, 通常包含两个 hash 函数, 用于计算元素的位置, 函数选择为 **Jekins Hash** 和 **MurmurHash**

2.1.1 lookup

Cuckoo Hashing 的查找操作非常简单，只需要使用每个 hash 函数计算出的位置, 比较是否等于目标值即可.

伪代码如下:

```
function lookup(x) -> bool
    for h in hashs
        if T[h(x)] == x
            return true
    return false
```

2.1.2 delete

Cuckoo Hashing 的删除操作也非常简单，只需要使用每个 hash 函数计算出的位置, 比较是否等于目标值, 如果相等则将该位置置为 nil 即可.

伪代码如下:

```
function delete(x)
    for h in hashs
        if T[h(x)] == x
            T[h(x)] = nil
            return
```

2.1.3 kick

驱逐操作是 Cuckoo Hashing 的核心操作, 其目的是将一个以及存在于哈希表中的元素移动到另一个位置, 以腾出空位置来插入新的元素.

伪代码如下:

```
function kick(x) -> bool
       for h in hashs
        if T[h(x)] == nil
             T[h(x)] = x
             return true
       for h in hashs
          y = T[h(x)]
          T[h(x)] = x
           if kick(y)
             return true
         return false
```

2.1.4 insert

插入操作是 Cuckoo Hashing 的核心操作, 其目的是将一个元素插入哈希表中.

插入操作会出现四种情况:

1. 该元素已经存在于哈希表中, 则直接返回
2. 该元素不存在于哈希表中且存在空位置, 则将该元素插入空位置
3. 该元素不存在于哈希表中且不存在空位置, 则执行驱逐操作, 以腾出空位置来插入该元素
4. 无法腾出空位置, 则对哈希表进行 rehash 操作

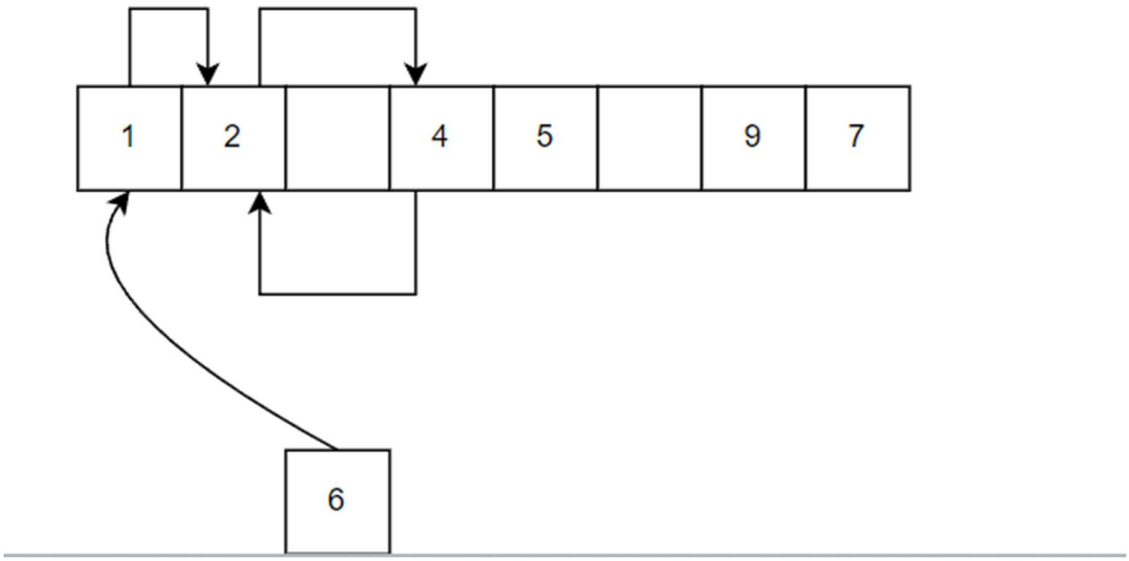
伪代码如下:

```
function insert(x)
    if lookup(x)
        return
    loop max_loop times
        for h in hashs
            if T[h(x)] == nil
                T[h(x)] = x
                return
        for h in hashs
            y = T[h(x)]
            T[h(x)] = x
            if kick(y)
                return
    end
    rehash
```

2.2 Cuckoo Hashing 的问题

Cuckoo Hashing 的主要问题是插入操作可能需要多次尝试, 特别是当哈希表接近满载时.

并且在驱逐操作时, 由于没有检测环的机制, 仅通过迭代深度检测是否处于无限循环, 陷入无限循环处境概率较大, 一个无限循环的示意图如下: 其中的驱逐路径 1 -> 2 -> 4 -> 2 -> 4 -> 2 -> 4 -> …陷入无限循环



图表 1

下面我们将从 **hash 函数数目**, **block cuckoo hashing**, **stash cuckoo hashing** 三个可行策略进行研究, 观测每个插入策略的优劣势并尝试添加简单的短路径环检测方法来减少无限循环的概率

2.3 Cuckoo Hashing 的改进策略

2.3.1 hash 函数数目

Cuckoo Hashing 的基本实现中, 通常使用两个 hash 函数, 用于计算元素的位置, 现在我们将尝试使用更多的 hash 函数, 以减少冲突的发生.

2.3.2 block cuckoo hashing

Blocked Cuckoo Hashing 是一种改进的 Cuckoo Hashing 算法, 其核心思想是将哈希表分为多个 block, 每个 block 包含多个位置, 每个哈希函数对应的内容不再是一个位置, 而是一个 block, 我们将在 block 中选择一个位置插入元素.

2.3.3 stash cuckoo hashing

Stash Cuckoo Hashing 也是一种改进的 Cuckoo Hashing 算法, 其核心思想是将哈希表分为两部分, 一部分是主表, 另一部分是 stash, 当插入操作无法腾出空位置时, 将元素插入 stash 中.

2.3.4 短路径环检测

考虑到完备的环检测算法会增加插入操作的时间复杂度, 我们采用简单的短路径环检测方法, 在驱逐操作时, 检测是否存在短路径环, 如果存在则不选择该位置, 以减少无限循环的概率.

在设计上, 我们在驱逐的过程中记录当前元素以及上一个元素, 在选择下一个位置时, 我们会检测是否存在短路径环, 如果存在则不选择该位置, 以减少无限循环的概率.

短路径环检测的一个简单例子:

不采用短路径环检测: 驱逐路径 1 -> 2 -> 4 -> 2 -> 4 -> 2 -> 4 -> …陷入无限循环

采用短路径环: 驱逐路径 1 -> 2 -> 4, 在选择下一个位置时, 会检测是否存在短路径环, 由于 2 -> 4 -> 2 存在短路径环, 则不选择该位置, 选择其他位置, 此时避免了一条无限循环的路径

2.4 Cuckoo Hashing 的实现

2.4.1 基础介绍

我们将每个插入策略实现为一个类, 并分别实现 lookup, delete, insert, kick 等操作.

实现基础内容:

- **DataGenerator** 用于生成随机数据, 用于测试插入操作
- **Hash** 用于计算元素的位置, 可设置随机种子用于生成不同的 hash 函数
  - **JenkinsHash** Jenkins Hash
  - **MurmurHash** Murmur Hash
- **build_cuckoo** 用于构建 Cuckoo Hashing, 可设置 hash 函数数目
  - **Cuckoo** Cuckoo Hashing
  - **BlockedCuckoo** Block Cuckoo Hashing
  - **StashCuckoo** Stash Cuckoo Hashing
  - **StashBlockCuckoo** Stash Block Cuckoo Hashing
- **argparse** 用于解析命令行参数, 用于设置插入策略, hash 函数数目, 数据量等参数

2.4.2 基本实现

基本函数:.

- **lookup(key)** 查找操作
- **remove(key)** 删除操作
- **insert(key)** 插入操作
- **kick(key,loop)** 驱逐操作
- **rehash()** 重新哈希操作

2.4.3 Cuckoo

经典的 Cuckoo Hashing 实现, 可设置 hash 函数数目, 我们也将探索 hash 函数数目负载因子的影响.

2.4.4 BlockedCuckoo

block cuckoo hashing 实现, 可设置 hash 函数数目, block 大小.

实现思路:

哈希表被分为多个 block, 每个 block 包含多个位置, 每个哈希函数对应的内容不再是一个位置, 而是一个 block, 可以发现当 BLOCK_SIZE=1 时, BlockedCuckoo 等价于 Cuckoo, 而当 BLOCK_SIZE 等于哈希表大小时, BlockedCuckoo 等价于链表. 所以 block cuckoo 的基本思想是利用 block 减少冲突, 从而提高负载因子, 但相应的查询效率会降低.

2.4.5 StashCuckoo

stash cuckoo hashing 实现, 可设置 hash 函数数目, stash 大小.

实现思路:

哈希表被分为两部分, 一部分是主表, 另一部分是 stash, 当插入操作无法腾出空位置时, 将元素插入 stash 中. stash cuckoo 的基本思想是利用 stash 存储无法插入的元素, 期望在未来的插入操作中能找到空位置, 从而减少哈希表重构的次数.

我们采用 unordered_set 来存储 stash, 将 stash 的效率也维持在常数级别. 可以发现当 STASH_SIZE=0 时, StashCuckoo 等价于 Cuckoo, 而当 STASH_SIZE 等于哈希表大小时, StashCuckoo 等价于普通的哈希表.

2.4.6 StashBlockCuckoo

stash block cuckoo hashing 实现, 可设置 hash 函数数目, block 大小, stash 大小.

同时吸收了 StashCuckoo 和 BlockedCuckoo 的优点, 一方面利用 block 减少冲突的发生提高存储效率, 另一方面利用 stash 存储无法插入的元素,减少哈希表重构的次数从而提高插入效率.

2.4.7 短路径环检测

实现思路:

kick(key,loop,prev_key)函数中, 我们增加了 prev_key 参数, 用于记录上一个元素, 在选择下一个位置时, 我们会检测是否存在短路径环, 如果存在则不选择该位置.

2.5 测试

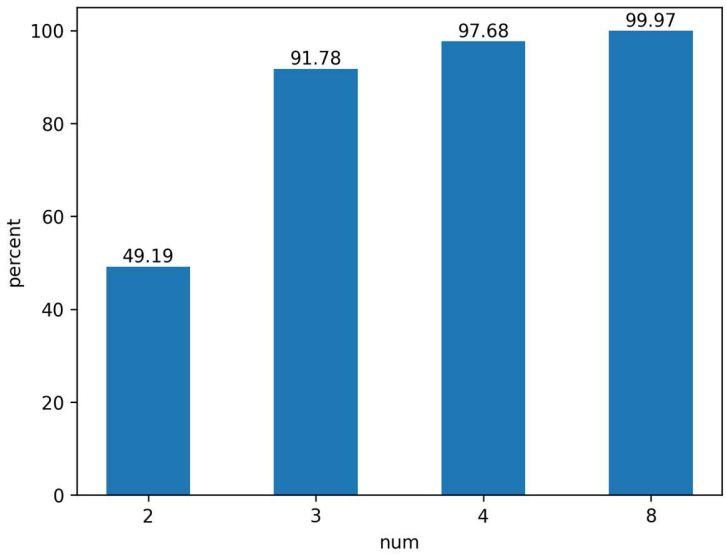
在测试中, 我将随机种子设置为我的学号 202115636, 以控制随机数生成的一致性.

2.5.1 hash 函数数目

基本参数介绍:

- `Cuckoo Type`: Cuckoo
- `Hash Function Number`: 2,3,4,8
- `Data Size`: 500000
- `Max Loop`: 20

经过测试, 我们得到了如下结果:



图表 2

可以发现, 在 hash 函数数目为 3 时, 负载因子就已达到了 90%以上, 而在 hash 函数数目为 8 时, 负载因子达到了 99.97%, 说明 hash 函数数目对于提高负载因子是有效的.

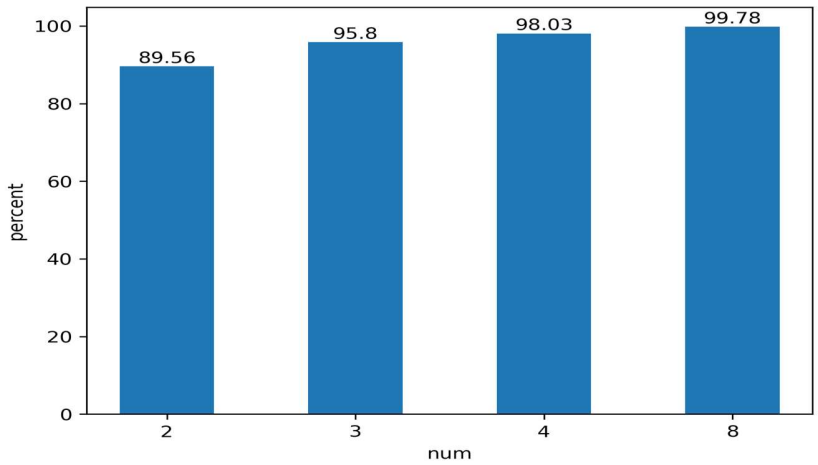
2.5.2 Block Cuckoo Hashing

block cuckoo hashing 用于减少冲突的发生, 从而提高负载因子, 故我们将测试 block cuckoo hashing 的负载因子变化.

基本参数介绍:

- `Cuckoo Type`: BlockedCuckoo
- `Hash Function Number`: 2
- `Data Size`: 500000
- `Max Loop`: 20
- `BLOCK_SIZE`: 2,3,4,8

经过测试, 我们得到了如下结果:



图表 3

可以发现, BLOCK_SIZE=2 时, 负载因子就达到了 89.56%,相较于没有 block 的 Cuckoo Hashing, 负载因子有了明显的提高, 而当 BLOCK_SIZE=8 时, 负载因子也达到了 99.78%, 说明 block cuckoo hashing 对于提高负载因子是有效的.

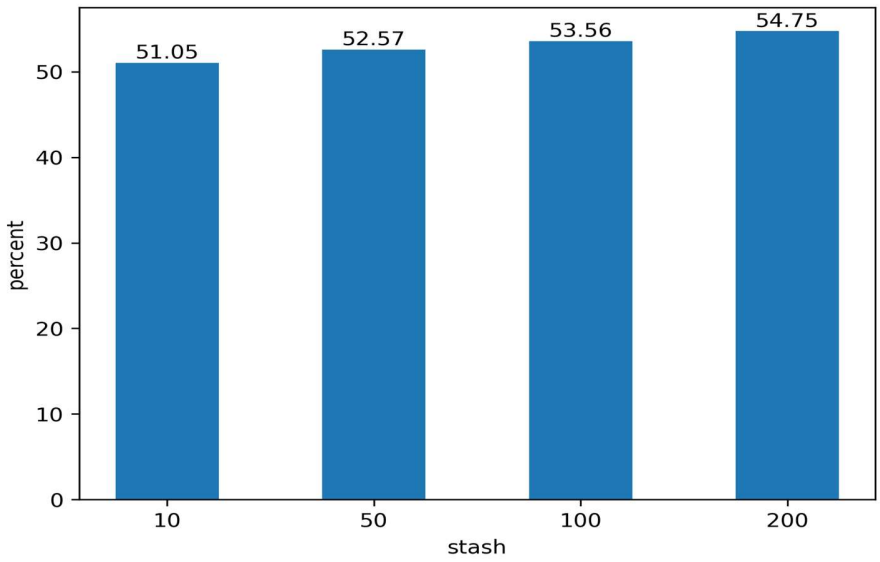
2.5.3 Stash Cuckoo Hashing

stash cuckoo hashing 用于存储无法插入的元素, 从而减少哈希表重构的次数, 使得在相近的时间内能插入更多的元素, 从而提高负载因子.

基本参数介绍:

- `Cuckoo Type`: StashCuckoo
- `Hash Function Number`: 2
- `Data Size`: 500000
- `Max Loop`: 20
- `STASH_SIZE`: 10,50,100,200

经过测试, 我们得到了如下结果:



图表 4

在 stash size = 10 的时候, 负载因子变化为 51.05%, 相较于没有 stash 的 Cuckoo Hashing(49.19%)提高了 1.86%, 实际元素个数为 500000*1.86% = 9300, 可以发现通过存储暂时无法插入的元素, 期望在未来的插入操作中能找到空位置, 可以在一定程度上提高负载因子.

通过调整 stash size, 我们还可以发现, 负载因子随着 stash size 的增加而增加, 但是增加的幅度逐渐减小, 说明 stash size 对于提高负载因子是有效的, 但是增加 stash size 的效果逐渐减小.

这是因为 stash cuckoo hashing 避免的是短期内的重构, 在冲突较多的情况下, stash 总是会被很快填满, 此时就会退化为普通的 Cuckoo Hashing.

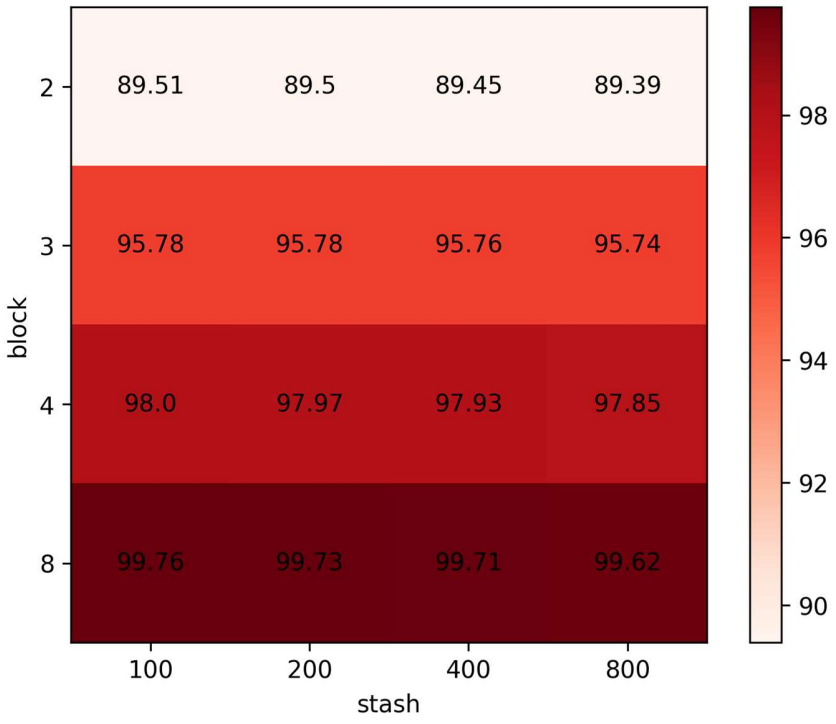
2.5.4 Stash Block Cuckoo Hashing

尝试结合 stash cuckoo hashing 和 block cuckoo hashing 的优点, 我们设计了 stash block cuckoo hashing, 以此观测其负载因子变化.

基本参数介绍:

- `Cuckoo Type`: StashBlockCuckoo
- `Hash Function Number`: 2
- `Data Size`: 500000
- `Max Loop`: 20
- `BLOCK_SIZE`: 2,3,4,8
- `STASH_SIZE`: 100,200,400,800

经过测试, 我们得到了如下结果:

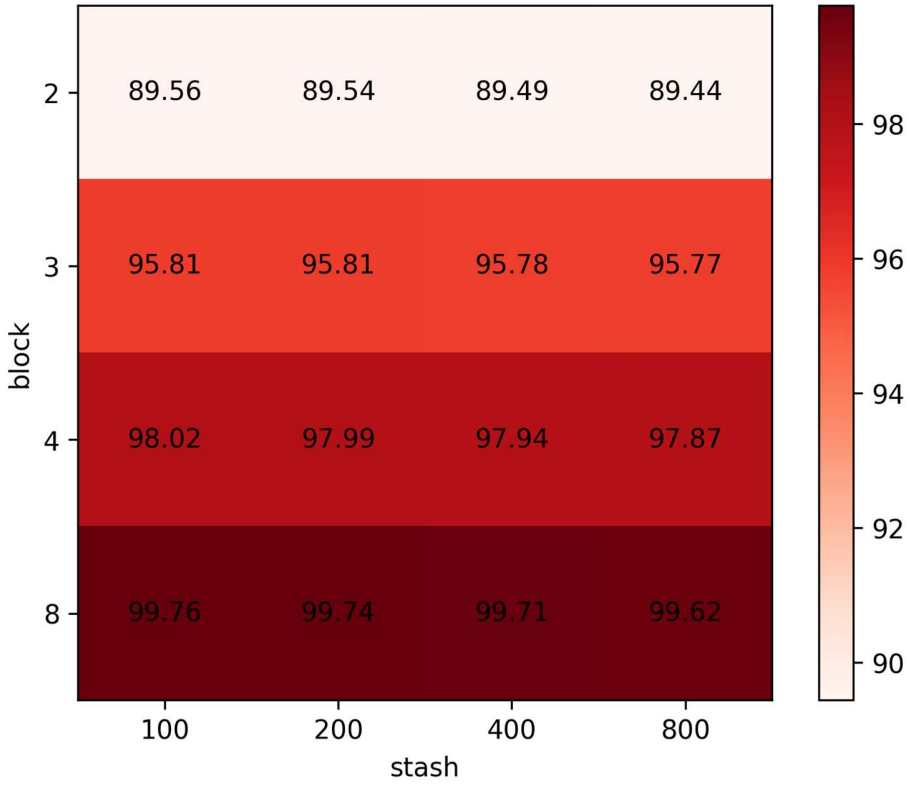


图表 5

可以发现, 负载因子仍会随着 block size 的增加而增加, 与此同时负载因子却在随着 stash size 的增加而减少. 经过调试, 发现一个可能的原因为由于 stash 的存在, 进行重构时, 暂时无法插入 hash 表的元素, 也会被放入 stash 中, 从而使得重构的时间相较于没有 stash 的 cuckoo hashing 时间更长, 但我们无法保证该次重构一定能为所有元素找到空位置, 从而使得在相同的时间内, 能插入的元素数量减少, 从而使得负载因子减少.

我们尝试通过调高程序运行时间限制为 20s, 在此之前设置为 10s, 来观测是否猜测正确.

于是我们得到了如下结果:



图表 6

可以发现在冲突相对较少时, 负载因子得到了微量的提升,而冲突较多时, 负载因子没有变化, 说明我们的猜测是正确的. 由此我们也可以得出结论, stash cuckoo hashing 在冲突较少时, 能够提高负载因子,

但在冲突较多时, 由于 stash 的存在, 使得重构时间变长, 从而使得负载因子减少. 所以我们需要根据实际情况来选择是否使用 stash cuckoo hashing.

2.5.5 短路径环检测

我们将测试短路径环检测的效果, 以此观测是否能够减少无限循环的概率. 测试方法为统计短路径环检测成功的次数, 以及尝试驱逐的次数, 以此来计算短路径环检测的效果.

- `Cuckoo Type`: Cuckoo
- `Hash Function Number`: 4
- `Data Size`: 500000
- `Max Loop`: 20

最终我们得到了如下结果:

驱逐次数: 537922520

短路径环检测成功次数: 179409571

短路径环检测成功率: 33.36%

可以发现, 我们的短路径环检测成功率为 33.36%, 说明短路径环检测能够有效减少无限循环的概率, 从而提高插入效率.

3 总结

在这篇报告中, 我们基于 Cuckoo Hashing 的基本思想, 从 **hash 函数数目**, **block cuckoo hashing**, **stash cuckoo hashing** 三个可行策略进行研究, 探索提高负载因子以及提高插入效率的策略. 并尝试通过短路径环检测方法来减少无限循环的概率.

从实验结果来看, 我们发现 hash 函数数目对于提高负载因子是有效的, 它为每个元素提供了更多的位置选择, 从而减少冲突的发生.

block cuckoo hashing 通过将哈希表分为多个 block, 每个 block 包含多个位置, 通过牺牲一定的查询效率, 提高了负载因子.

stash cuckoo hashing 为暂时无法插入的元素提供了存储空间, 期望在未来的插入操作中能找到空位置, 从而减少哈希表重构的次数, 使得在相近的时间内能插入更多的元素, 从而提高负载因子.

同时我们也要注意在冲突较多时, 由于 stash 的存在, 使得重构时间变长, 也会使得插入效率降低, 所以我们需要根据实际情况来选择是否使用 stash cuckoo hashing.

短路径环检测能够有效减少无限循环的概率, 它通过简单的短路径环检测方法, 来检测无限循环的发生, 从而提高插入效率.

当然, 我们也可以拓展更多点数的短路径环检测方法, 以提高短路径环检测的成功率, 如四个点的环, 五个点的环等. 点数的选择需要综合考虑插入效率的需求和环检测的开销, 以满足实际需求.

4 参考文献

- Cuckoo Hashing