大数据存储管理课程报告

姓　　名：　　杨尚君

学　　院：　　计算机科学与技术

专　　业：　　计算机科学与技术

班　　级：　　CS2110

学　　号：　　U202115648

指导教师：　　华宇

分数	
教师签名	

2023 年 12 月 20 日

利用 Necklace 模型改进 Cuckoo Hashing

1. 实验背景

Cuckoo Hashing 是一种高效的哈希算法，其核心思想在于使用两个或更多的哈希函数和对应的哈希表来解决冲突，从而实现高效的查询、插入和删除操作。

Cuckoo Hashing 的核心是使用两个哈希函数 h1(x)和 h2(x)，每个元素 x 可以放在两个位置 h1(x)或 h2(x)中的任意一个。如果一个新插入的元素在两个位置都已被占用，则会踢出一个已有元素，将其放到另一个位置，可能导致连锁反应。

虽然 Cuckoo Hashing 能够带来复杂度为 O(1)查询操作，但是在写入发生冲突，从而产生踢出操作时，就很可能带来无限循环踢出的问题。

传统的解决方法是设置一定的踢出次数上限，一旦发现踢出次数达到上限，就立刻停止，并返回插入失败。然而这样的作法存在缺陷，会让数据反复的在 bucket 之间移动，会带来写放大问题，并且在对插入延迟敏感的场景下，这种反复踢出是不能容忍的。

因此，可以使用 Necklace 模型改进 Cuckoo Hashing，通过添加一个辅助表的形式，在插入操作发生冲突时，利用辅助表反向搜索冲突的 bucket 到空位的最佳路径，从而避免盲目的踢出操作带来的无限循环问题。

2. 数据结构的设计

我们要基于 Necklace 模型在 Java 中实现一个 HashMap，从而实现 put 和 get 操作，为此需要定义以下数据结构，包括哈希表、辅助表。

2.1 哈希表的设计

首先要定义哈希表的表项，我们以 KV 对的 key 的哈希值作为索引来访问 table，因此哈希表的表项存放的是 KV 对。

定义了哈希表的表项之后，哈希表以数组的形式存储即可。

2.2 哈希函数的设计

Cuckoo Hashing 的核心是多个哈希函数。需要注意的是，各个哈希函数之间一定要满足独立不相关的条件，因此我们设计三个哈希函数如下所示：

```
private int hash1(Object key) {
    return (key == null) ? 0 : key.hashCode() % CAPACITY;
}

private int hash2(Object key) {
    if (key == null) return 0;
```

```
    int result = 17;
    result = 31 * result + key.getClass().getName().hashCode(); // 类名的哈希
    result = 31 * result + System.identityHashCode(key); // 系统身份哈希
    return Math.abs(result) % CAPACITY;
}

private int hash3(Object key) {
    if (key == null) return 0;
    long constant = 0x9e3779b97f4a7c15L; // 一个较大的质数的黄金分割数
    long keyHash = key.hashCode();
    long mix = (keyHash ^ (keyHash >>> 16)) * constant;
    mix ^= (mix >>> 30) + (System.nanoTime() & 0xFF); // 添加纳秒级时间的低 8 位以引入更多随机性
    return Math.abs((int) (mix % CAPACITY));
}
```

2.3 辅助表的设计

为了能够在发生冲突时，找出哪个 bucket 是空位并且可能存在路径到达这个空位，需要两个 boolean 型的数组 isHashed 和 isOccupied。isHashed 数组用于表示各个 bucket 是否有元素能够哈希到此位置；isOccupied 数组用于表示各个 bucket 当前是否以及被占用。在寻找可能的空位时，只需要线性扫描这两个数组，isHashed 为 true 且 isOccupied 为 false 的位置即为我们的目标(vacancy).

然后需要定义一个能够动态插入的表（使用 ArrayList 实现）作为辅助表，其中的表项包括：Key，hash1(key),hash2(key),hash3(key)，同时需要用三个 boolean 型的变量标记该元素具体存放在哪个哈希函数得到的索引中。

3 操作流程分析

3.1 get 操作

在 Cuckoo Hashing 中，get 操作比较简单，只需要求出通过三个哈希函数得到的索引，再在哈希表中访问这三个 bucket，找到 key 相同的 bucket 后返回存放的 value 即可。

3.2 put 操作

put 操作分为冲突和非冲突两种情况，两种情况都需要先通过三个哈希函数求出三个索引。

求出索引之后，如果 bucket 中的三个位置有其中一个为空，那么将数据直接放入空位并更新辅助表即可完成此次 put 操作。

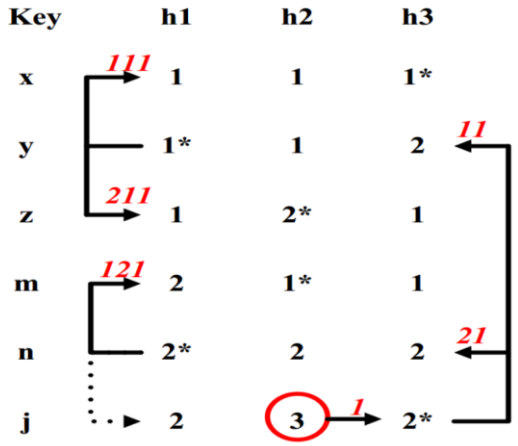
如果 bucket 中的三个位置都非空，那么此次 put 操作则发生了冲突，需要对冲突进行处理。

3.3 冲突处理

一旦检测到发生冲突，就需要利用辅助表找到从冲突位置到某个空位的路径，然后根据此条路径逐一进行踢出操作。

要找到这条路径，首先要确定一个可能到达的空位的位置。如果一个 bucket 的 isHashed 值为 false，表明没有哪个元素能够放到这个位置，那么就可能存在合适的路径；如果一个 bucket 的 isOccupied 值为 true，表面这个 bucket 已被占用，不是我们要找的空位。因此我们需要找到一个 isHashed 值为 true 且 isOccupied 值为 false 的位置。如果不存在这样的位置，意味着不存在能够用于踢出的空位，那么这次插入操作即可被认为失败。

找到所有可能到达的空位后，对每个空位依次进行下面的反向搜索操作：



例如，对于空位 3，先找到该行存放元素的位置 2，表明位置 2 可以通过一次移动操作到达 3，使得位置 2 变得空闲。然后找到同一列的所有 2，表明可以将某个元素移动到位置 2，从而产生新的空闲。通过这种类似于 BFS 的方式，即可反向扩展出所有能够到达空位 3 的元素，同时在反向扩展的过程中需要保存路径，以便于后续的踢出操作。

利用上述方法找到从冲突位置到空闲位置的路径后，便可以根据路径依次进行踢出操作，从而完成发生冲突时的插入。

冲突处理函数的核心代码如下：

```
private void handleConflict(K key, V value) {
    int index1 = hash1(key);
    int index2 = hash2(key);
    int index3 = hash3(key);

    // Try to relocate existing entries
    if (!relocate(index1) && !relocate(index2) && !relocate(index3))
{
        System.out.println("Rehash needed: no available slots found
through relocation");
        // Rehash or resize logic could be implemented here
    }
}
```

```
    // Attempts to relocate the item at the specified index
    private boolean relocate(int index) {
        AuxiliaryEntry<K, V> entry = findAuxiliaryEntryByIndex(index);
        if (entry != null) {
            // Try the other two indices for this entry
            int newIndex1 = entry.h1_valid && entry.h1_item != index ?
entry.h1_item : -1;
            int newIndex2 = entry.h2_valid && entry.h2_item != index ?
entry.h2_item : -1;
            int newIndex3 = entry.h3_valid && entry.h3_item != index ?
entry.h3_item : -1;

            // Check if new indices are available
            if (newIndex1 != -1 && !idOccupied[newIndex1]) {
                moveEntry(index, newIndex1);
                return true;
            }
            if (newIndex2 != -1 && !idOccupied[newIndex2]) {
                moveEntry(index, newIndex2);
                return true;
            }
            if (newIndex3 != -1 && !idOccupied[newIndex3]) {
                moveEntry(index, newIndex3);
                return true;
            }
        }
        return false;
    }

    // Move entry from one index to another
    private void moveEntry(int fromIndex, int toIndex) {
        table[toIndex] = table[fromIndex];
        table[fromIndex] = null;
        idOccupied[fromIndex] = false;
        idOccupied[toIndex] = true;
    }

    // Find auxiliary entry by table index
    private AuxiliaryEntry<K, V> findAuxiliaryEntryByIndex(int index) {
        for (AuxiliaryEntry<K, V> entry : auxiliary) {
            if (entry.h1_item == index || entry.h2_item == index ||
entry.h3_item == index) {
                return entry;
            }
```

```
        }
        return null;
    }
```

3. 理论分析

3.1无限循环问题

当插入发生冲突时，冲突处理逻辑会试图找到一条从冲突位置到空闲位置的路径，如果存在路径，则会根据路径依次进行踢出操作。由于每次踢出操作都是向着空位的方向前进，因此 Necklace 模型完全避免了无限循环的问题。

3.2插入失败

如果在冲突处理逻辑中没有找到一条从冲突位置到空位的路径，那么就会立即停止插入并且返回插入失败。从理论上分析，这种插入失败是无法避免的。即便不使用 Necklace 模型，这种情况下也会导致无限循环而无法插入。因此 Necklace 模型优化了插入的效率，减少了由于路径不合理导致的插入失败，但是理论上无法解决不存在路径时的插入失败。

4. 测试性能

4.1延迟测试

在实现了 NecklaceHashMap 后，初始化一个容量为 1000 的 HashMap，对其进行插入、读取延迟测试。

平均插入延迟为：792 ns

平均读取延迟为：105 ns

在 750 次插入操作中有 102 次发生了哈希冲突，2 次插入失败。

4.2空间开销

假设一个使用Necklace模型的HashMap,Key为Integer类型,Value为String类型，大小均为 4B，最大容量为 1000，插入了 750 个表项。

对于数组形式的哈希表，大小为 8000B

对于辅助表，大小为(4+3*4+3)*750=14250B

可见，在这种实现下，辅助表占用的空间较大，因此可以判断 Necklace 是一种用空间换时间的方案。