华中科技大学

大数据存储系统与管理课程报告

姓　　名：　　杨烨

学　　院：　　计算机科学与技术学院

专　　业：　　数据科学与大数据技术

班　　级：　　大数据 2102 班

学　　号：　　U202115566

指导教师：　　施展

分数	
教师签名	

2024 年 4 月 21 日

目　录

1 Cuckoo-driven Way

1.1 背景

Cuckoo Hashing 是一种用于解决哈希表中键冲突的技术。它最初由 Pagh 和 Rodler 于 2001 年提出，并在 2003 年的一篇论文中进行了更全面的研究。Cuckoo Hashing 之所以出现，是因为传统的哈希表解决冲突的方法（如链地址法和开放寻址法）在某些情况下可能会导致性能下降。

Cuckoo Hashing 的提出旨在解决这些问题。它通过强制要求每个键值对只能存储在两个位置中的一个，从而避免了链表过长的问题，并通过重新哈希操作解决了开放寻址法中出现的簇问题。这使得 Cuckoo Hashing 在某些情况下比传统方法更具效率，并被广泛用于高性能的哈希表实现中。

其名称源自布谷鸟孵化它们的幼鸟的方式。在 Cuckoo Hashing 中，哈希表被分成两个数组（通常称为桶），每个数组都有自己的哈希函数。当插入键值对时，首先通过第一个哈希函数计算出在第一个桶中的位置，如果该位置已经被占据，则将新的键值对插入到该位置，同时将原先的键值对“踢出”到第二个桶中，并继续尝试在第二个桶中插入。如果第二个桶中的位置也已经被占据，同样的操作会在第一个桶中进行，直到找到空位或达到最大尝试次数。如果超过了最大尝试次数仍然找不到空位，则会进行重新哈希操作，扩大哈希表的大小，并重新插入所有键值对。Cuckoo Hashing 的优点是插入和查找的时间复杂度都是常数时间，而不是与哈希表中元素数量成比例的时间复杂度，因此在某些情况下比传统的哈希表更快。

1.2 数据结构的设计

定义了一个 CuckooHashing 类用于实现 Cuckoo 哈希的基本功能（插入和搜索），并定义了一个计时装饰器用于测试插入和搜索的性能

1.2.1 CuckooHashing 综述

1. 维护两个表 table1 和 table2，每个表大小都为 size
2. 选择两个 hash 函数应用至两个表上，如下所示。

```
def hash_func1(self, key):
    return hash(key) % self.size
def hash_func2(self, key):
    return (hash(key) // self.size) % self.size
```

3. 对于任意元素 x，x 要么存储在 table1 的 h1(x)处，要么存储在 table2 的 h2(x)处

1.2.2 CuckooHashing 类包含的方法

1. __init__：用于初始化 CuckooHashing 类，该方法初始化两个大小为 size 的列表
2. hash_fun1 和 hash_fun2：用于将 key 通过 hash 映射到对应范围
3. insert：将 key 插入至表中，成功返回 true，失败返回 false
4. search：从表中搜索 key，成功返回 true，失败返回 false
5. rehash：动态扩容，使哈希表的存储更加高效

1.3 操作流程分析

其大致流程如图 1 所示。

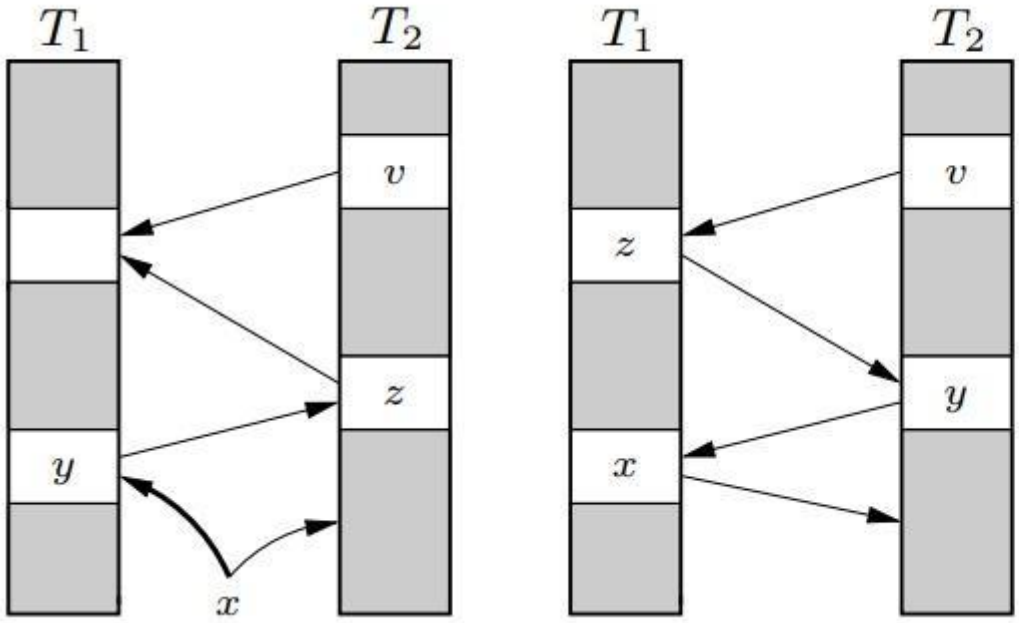

图 1

1.3.1 Insert 操作

1. 计算哈希值：首先，根据给定的键（key），通过两个哈希函数 hash_func1 和 hash_func2 计算出两个哈希值 i1 和 i2。
2. 尝试插入：然后，使用一个循环来尝试将键插入到哈希表中。循环的次数由 max_try 控制，避免死循环。
3. 检查空槽位：在循环中，首先检查两个哈希值对应的位置是否为空。如果其中一个位置为空，则将键插入到该位置，并返回 True。
4. 处理冲突：如果两个位置都已经被占用，那么需要进行冲突处理。这里采用了随机选择的方法，随机选择要从中踢出键的表（table1 或 table2）。然后，将原来位置的键踢出，并将新的键插入到该位置。然后，更新新的键和哈希

值，继续循环。

5. 达到最大尝试次数：如果达到了最大尝试次数（max_try），则会调用 self.rehash() 方法进行重新哈希。然后，再次尝试将键插入到哈希表中。
6. 递归插入：重新哈希后，调用 self.insert(key) 进行递归插入，以尝试将键插入到新的哈希表中。

1.3.2 Search 操作

1. 计算哈希值：首先，根据给定的键（key），通过两个哈希函数 hash_func1 和 hash_func2 计算出两个哈希值 i1 和 i2。
2. 检查直接位置：然后，检查哈希表中与这两个哈希值对应的位置是否包含要搜索的键。如果任何一个位置包含要搜索的键，则返回 True，表示找到了键。
3. 检查备用位置：如果直接位置中没有找到键，那么尝试查找备用位置。备用位置是指另一个表中与当前位置对应的位置。首先，根据当前位置中的键，通过相应的哈希函数计算出备用位置的索引。然后，检查备用位置是否包含要搜索的键。
4. 返回结果：如果备用位置中包含要搜索的键，则返回 True，表示找到了键。否则，返回 False，表示没有找到键。

1.3.3 Rehash 操作

1. 增加表大小：首先，将哈希表的大小加倍，以便存储更多的键值对。这通过将 self.size 属性乘以 2 来实现。
2. 创建新的表：然后，创建两个新的哈希表 new_table1 和 new_table2，它们的大小都是原来的两倍。初始时，所有的槽位都设置为 None。
3. 重新哈希键：接下来，对原来的哈希表中的每个键进行重新哈希。对于哈希表 1（self.table1），对每个键使用 hash_func1 函数计算出新的索引，并将键存储到新的哈希表 1 中的相应位置。对于哈希表 2（self.table2），对每个键使用 hash_func2 函数计算出新的索引，并将键存储到新的哈希表 2 中的相应位置。
4. 更新引用：最后，将原来的哈希表引用指向新的哈希表，以便后续的插入和查找操作可以使用新的哈希表。

1.4 理论分析

1.4.1 false positive 和 false negative

在理想情况下，我们认为布谷鸟哈希表不存在 false positive 和 false negative。但是布谷鸟哈希表在处理极端情况下可能会出现错误，例如哈希表过度填充、哈希函数选择不当等情况下。因此，在一些特殊情况下，仍然可能出现 false positive 和 false negative 的情况。

1.4.2 解决无限循环的措施

1. 设定了最大重试次数：在 insert 方法中，使用了一个 max_try 变量来限制尝试插入的最大次数。如果尝试次数超过了这个限制，就会触发重新哈希的操作，从而避免了无限循环的发生
2. 动态扩容：动态增加哈希表的容量，减少哈希冲突的概率，从而降低 Cuckoo 操作失败的概率

1.4.3 有效存储的措施

在发生冲突时，随机选择要从哪个表中踢出键值，可以使得两个表的负载更加均衡，避免了一个表过载而另一个表空闲的情况。这样做有助于提高整个哈希表的存储效率，使得键值能够更均匀地分布在两个表中，从而最大程度地利用了哈希表的存储空间。

1.5 实验测试的性能

1.5.1 测试流程

1. 创建布谷鸟哈希表实例：创建了四个不同大小的布谷鸟哈希表实例，分别是 hash_table、hash_table1、hash_table2 和 hash_table3，它们的大小分别是 1000、10000、50000 和 100000。
2. 生成一组随机键值：生成了一个包含了一组随机键值的列表 keys，范围从 0 到 99999。
3. 测试函数定义：定义了一个测试函数 test_insert_search，该函数接受三个参数：size（哈希表的大小）、hash_table（布谷鸟哈希表实例）和 keys（随机键值列表）。该函数的作用是将所有键插入到哈希表中，然后再进行搜索操作。
4. 测试函数执行：使用 test_insert_search 函数测试了四个不同大小的布谷鸟哈希表的性能。对于每个哈希表，先插入所有的随机键值，然后再对每个键值进行搜索操作。
5. @timer 装饰器：@timer 装饰器用于测量函数的执行时间。它会在函数执行前记录开始时间，在函数执行后记录结束时间，并计算出函数执行的时间差

1.5.2 测试结果与分析

测试结果如图 2 所示。在向 table 大小小于 key 数量的哈希表插入数据时，性能显著小于向较大的表中插入数据。原因应该是需要进行动态扩容，且小表冲突率较高。但同时表越大所需要的空间也会更大。所以表的大小应该设定在一个合理的区间，平衡对时间和空间的需求。


HashTable大小:1000 执行时间: 70.987000 毫秒
HashTable大小:10000 执行时间: 75.009300 毫秒
HashTable大小:50000 执行时间: 61.992500 毫秒
HashTable大小:100000 执行时间: 54.659600 毫秒

图 2