



华中科技大学

大数据存储系统与管理报告

姓 名：李明锦
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：BD2102
学 号：U202115396
指导教师：施展

分数	
教师签名	

2024 年 4 月 21 日

目 录

1 数据结构的设计	1
1.1 Cuckoo 哈希表概述	1
1.2 哈希桶设计	1
1.3 图模型设计	1
1.4 Cuckoo 哈希表实现	2
1.5 测试和验证函数设计	3
2 操作流程分析	4
2.1 插入操作流程分析	4
2.2 查找操作流程	5
2.3 驱逐操作流程	6
2.4 重构操作流程	7
2.5 性能测试流程	7
3 理论分析	8
3.1 False positive 和 false negative	8
3.2 有效存储	8
3.3 减少循环次数	8
4 性能测试	9

1 数据结构的设计

1.1 Cuckoo 哈希表概述

Cuckoo hash 是哈希表的一种实现方式，它使用多个哈希函数来计算键的多个哈希值，一个键有多个候选位置，当发生哈希冲突时，被淘汰的元素将寻找它的下一个候选位置。

1.2 哈希桶设计

Bucket 类：定义了一个哈希桶，包含 key 和 value 属性，用于存储哈希表中的键值对。

```
class Bucket:
    def __init__(self, key=None, value=None):
        self.key = key
        self.value = value
```

1.3 图模型设计

Graph 类：用于表示哈希表的图模型，包含以下属性和方法：

构造函数：

```
def __init__(self, max_id):
    self.max_id = max_id
    self.graph = defaultdict(set)
    self.free_list = set(range(max_id + 1))
```

max_id: 表示节点索引值的最大范围。

graph: 使用 defaultdict(set)表示邻接表，存储节点间的驱逐路径。

free_list: 存储所有有空闲位置的节点集合。

add_free_node(id): 向图中添加一个空闲节点。

remove_free_node(id): 移除一个不再空闲的节点。

add_edge(id_x, id_y): 在图中添加一条驱逐路径。

shortest_path_from_src(src_node_id): 使用广度优先搜索算法寻找从源节点到空闲节点的最短驱逐路径。

reconstruct(): 重构图，用于在找不到可行驱逐路径时重构哈希表。

1.4 Cuckoo 哈希表实现

CuckooHash 类：实现 Cuckoo 哈希表的核心功能，包含以下属性：

capacity: 哈希表的容量。

entry_cnt: 每个哈希桶的槽位数。

max_depth: 允许的最大驱逐深度。

table1 和 table2: 两个哈希表，用于 Cuckoo 哈希的两个哈希函数。Table 声明如下：

```
self.table1 = [[Bucket() for _ in range(entry_cnt)]
for _ in range(capacity//2)]
self.table2 = [[Bucket() for _ in range(entry_cnt)]
for _ in range(capacity//2)]
```

seed1 和 seed2: 为 mmh3 哈希函数提供的种子，用于生成不同的哈希值。

方法：

hash1(key) 和 hash2(key): 定义两个哈希函数，返回键的哈希位置。代码实现如下：

```
def hash1(self, key):
    return mmh3.hash(str(key), self.seed1) % self.capacity // 2

def hash2(self, key):
    return mmh3.hash(str(key), self.seed2) %
self.capacity // 2
```

insert(key, value): 插入操作，处理哈希冲突和驱逐过程。

kick(key, value, hash_func, table, other_table, other_hash_func, depth): 尝试将一个键值对从一个哈希桶“踢”到另一个哈希桶。

lookup(key, table, hash_func): 查找操作，检查键是否存在于哈希表中。

copy_to_buffer(): 将哈希表中的所有键值对复制到缓冲区，为重构做准备。

reconstruction(): 重构哈希表，更新哈希种子并重新插入缓冲区中的所有键值对。

get_value(key): 获取与特定键关联的值。

print_info(): 打印哈希表的状态信息，包括占用率、平均驱逐次数和平均插入时间。

1.5 测试和验证函数设计

`init_test_array(array, size)`: 初始化测试数组，填充随机值。

`check(test_array, cuckoo_hash_instance, test_size)`: 验证 Cuckoo 哈希表的插入操作是否正确。

`main()`: 程序入口点，解析命令行参数，初始化 Cuckoo 哈希表实例，执行插入操作，验证结果并打印性能信息。

2 操作流程分析

2.1 插入操作流程分析

函数遍历两个哈希表，对于每个哈希表，它首先检查键是否存在。如果键已经存在，那么函数直接返回 `True`。然后它计算键的哈希值，然后检查哈希表中的对应位置。如果这个位置是空的，那么他就将新的键值对插入到这个位置，同时添加一条驱逐路径——该哈希值到另外一个哈希函数映射的哈希值，然后返回 `True`。

如果在两个哈希表中都没有找到空的位置，那么函数就会尝试进行提出操作。在踢出操作之前，如果索引项仍然在空闲节点中，那么首先将其从空闲列表中拿出（因为进行到这一步的时候，说明该索引项对应的哈希桶的所有槽位都已经满了）。如果提出操作成功，那么函数返回 `True`。否则，就将原来的键值对重新插入到哈希表中。

如果所有的踢出操作都失败，那么函数就会重构哈希表，然后尝试再次插入新的键值对。插入成功就返回 `True`，否则返回 `False`。

```
def insert(self, key, value):
    start_time = time.time()
    # print("insert:: key: ", key, " value: ", value)
    for table, hash_func, other_table, other_hash_func in ((self.table1,
self.hash1, self.table2, self.hash2), (self.table2, self.hash2, self.table1,
self.hash1)):
        if self.lookup(key, table, hash_func)[0]:
            # 如果 key 已经存在，则直接返回 true
            end_time = time.time()
            self.total_insert_time += end_time - start_time
            self.total_insert_count += 1
            return True
        index = hash_func(key)
        for i in range(self.entry_cnt):
            # print("index: ", index, " i: ", i, " table[index][i]: ",
table[index][i])
            if table[index][i] is None or table[index][i].key is None:
                table[index][i] = Bucket(key, value)
                end_time = time.time()
                # 添加一条驱逐路径的边
                self.graph.add_edge(index, other_hash_func(key))
```

```

        self.total_insert_time += end_time - start_time
        self.total_insert_count += 1
        return True

# 如果插入失败，进行踢出操作
for table, hash_func, other_table, other_hash_func in ((self.table1,
self.hash1, self.table2, self.hash2), (self.table2, self.hash2, self.table1,
self.hash1)):
    index = hash_func(key)
    if index in self.graph.free_list:
        # 当一个哈希桶的所有 entry 都被占用时，将该哈希桶从 free_list 中移除
        self.graph.remove_free_node(index)
    for i in range(self.entry_cnt):
        old_key, old_value = table[index][i].key, table[index][i].value
        # 将当前的 key-value 插入到哈希桶中
        table[index][i] = Bucket(key, value)
        # 添加一条驱逐路径的边
        self.graph.add_edge(index, other_hash_func(old_key))
        print(f"old_key: {old_key}, old_value: {old_value}")

    if self.kick(old_key, old_value, hash_func, table, other_table,
other_hash_func, 1):
        return True
    else :
        # 如果踢出失败，将原来的 key-value 重新插入到哈希桶中
        table[index][i] = Bucket(old_key, old_value)
        continue

```

```

# 如果所有的踢出操作都失败，进行重构操作
self.reconstruction()
if self.insert(key, value):
    end_time = time.time()
    self.total_insert_time += end_time - start_time
    self.total_insert_count += 1
    return True
else:
    end_time = time.time()
    self.total_insert_time += end_time - start_time
    self.total_insert_count += 1
    return False

```

2.2 查找操作流程

函数遍历传入的哈希表，使用哈希函数计算键(key)的可能哈希位置，对于

哈希位置，遍历桶内的所有槽位以查找键。

如果找到键，则返回对应的值；否则，返回 None。

```
def lookup(self, key, table, hash_func):
    """
    用来查询 key 在当前 table 和 hash 中是否存在,如果存在同时返回 key 所在的位置
    """

    index = hash_func(key)
    for j in range(self.entry_cnt):
        if table[index][j] is not None and table[index][j].key == key:
            return True, j
    return False, None
```

2.3 驱逐操作流程

首先确定被驱逐键值对的备用哈希位置，尝试将被驱逐的键值对插入备用位置的空闲槽位中。如果备用位置被占用，将该位置的键值对踢出，并递归执行驱逐操作。

每次成功插入或驱逐操作后，更新图模型中的驱逐路径。在必要时，使用图模型检测是否存在可行的驱逐路径以避免无限循环，也可以使用最大递归深度避免无线递归。

```
def kick(self, kick_key, kick_value, hash_func, table, other_table,
other_hash_func, depth):
    if depth > self.max_depth:
        return False
    self.total_kick_count += 1
    kick_pos = other_hash_func(kick_key) % (self.capacity // 2) # Ensure the
index is within the correct range
    for i in range(self.entry_cnt):
        if other_table[kick_pos][i] is None:
            other_table[kick_pos][i] = Bucket(kick_key, kick_value)
            self.graph.add_edge(kick_pos, hash_func(kick_key) % (self.capacity
// 2)) # Add edge between the two hash positions
            return True
        elif other_table[kick_pos][i].key == kick_key:
            return True
        else:
            old_key, old_value = other_table[kick_pos][i].key,
other_table[kick_pos][i].value
            other_table[kick_pos][i] = Bucket(kick_key, kick_value)
```



```

        # Recursively kick the old key, using the other hash function
        if self.kick(old_key, old_value, other_hash_func, other_table,
table, hash_func, depth+1):
            return True
        else:
            # If the kick fails, put the old key-value back
            other_table[kick_pos][i] = Bucket(old_key, old_value)
        return False

```

2.4 重构操作流程

首先将当前哈希表中的所有键值对复制到缓冲区；然后改变哈希函数所使用的种子，以获得新的哈希位置。清空当前哈希表，同时重构图模型；然后从缓冲区中取出键值对，并使用更新后的哈希函数重新插入到哈希表中。

```

def reconstruction(self):
    # 重构哈希表同时将缓冲区中的 key-value 对重新插入到哈希表中
    self.copy_to_buffer()
    self.table1 = [[Bucket() for _ in range(self.entry_cnt)] for _ in
range(self.capacity//2)]
    self.table2 = [[Bucket() for _ in range(self.entry_cnt)] for _ in
range(self.capacity//2)]
    self.seed1 = (self.seed1*2)%2**32
    self.seed2 = (self.seed2*2)%2**32
    # 重构图
    self.graph.reconstruct()
    for key, value in self.buffer:
        if not self.insert(key, value):
            return False
    return True

```

2.5 性能测试流程

首先生成一个包含随机整数的数组作为测试数据；然后将测试数据中的每个键值对插入 CUckoo 哈希表中。记录插入操作的开始和结束时间，检查每个键是否成功插入，并获取其对应的值。最后打印总插入时间，每项插入的平均时间以及哈希表的其他统计信息。

同时为了多次测量求平均值，添加了一个运行 10 次主函数的循环，最后求统计信息的平均值。

3 理论分析

3.1 False positive 和 false negative

首先，明确 false positive 和 false negative 在哈希表上下文中的含义：

False Positive (FP)：错误地报告某个键存在于哈希表中，而实际上该键并不存在。

False Negative (FN)：错误地报告某个键不存在于哈希表中，而实际上该键存在。

而 Cuckoo 不存在误判的情况，因此 false positive 和 false negative 均为 0，同时代码中的 check 函数也实现了检测这两项指标的功能，所以当函数正常运行时结果的时候，这两项指标均为 0。

3.2 有效存储

为了进行有效存储，哈希函数选用 Python 的 MMH 哈希函数；

使用 MMH3 作为 Cuckoo Hash 的哈希函数，理论上可以带来以下效果：

- 快速的哈希计算：MMH3 非常快，这对于 Cuckoo Hash 来说是一个重要特性，因为它依赖于快速的哈希计算来决定元素的放置。
- 均匀的哈希分布：MMH3 设计用于减少哈希碰撞，提供均匀的哈希分布，这对于 Cuckoo Hash 的性能至关重要。
- 减少踢出操作：由于 MMH3 可以提供较好的哈希分布，使用它作为 Cuckoo Hash 的哈希函数可能会减少踢出操作的数量，因为元素更有可能被分配到正确的桶中。
- 高吞吐量：在高并发场景下，MMH3 的快速哈希计算和较低的碰撞率可以提高 Cuckoo Hash 表的整体吞吐量。

3.3 减少循环次数

为了避免插入过程中的驱逐操作出现无限循环，我们在插入前，检测一下哈希表中是否存在一条可行的驱逐路径，驱逐路径的起始节点是待放置到哈希表中的元素，终点如果是一个哈希表中还存在空闲位置的节点，那么就存在这样一条驱逐路径，使得在执行驱逐的时候不会出现无限循环。

该函数的具体实现是在 Graph 类中的 `shortest_path_from_src(src_node_id)` 函数。

该函数使用广度优先搜索算法寻找从源节点到空闲节点的最短驱逐路径。其中对于驱逐路径的边的解释为：当插入一个节点的时候，将该节点所在哈希桶的位置，与在另一个哈希桶的该节点的哈希位置建立连接。

空闲节点：当哈希桶的某个位置的槽位还没有占满的时候，该节点仍然是空闲节点；当该位置的槽位占满的时候，从空闲节点删除。

同时当插入的元素接近哈希表的容量限制的时候，插入操作的耗时可能会显著增加，从而导致性能瓶颈——从而将占有率限制在一定范围内。

4 性能测试

保持哈希表的总大小不变，即哈希桶的数量和桶的相联度的乘积不变，一次设置哈希桶的相联度为 1,2,4,8.分别对这四个 CuckooHash 执行一系列的插入操作，并记录插入过程中发生驱逐的次数，以及插入操作进行的时间。

运行结果如下表所示。可见，随着相联度的提高，哈希表的占用率也随之提高。同时，由于每次即使运行相同的测试命令，得到的数据仍然存在差异，所以为了获得更加准确的数据，对于每次测试，我都加入一个运行十次的循环，最终取平均值为结果。同时在运行过程中发现，当哈希表无法插入一个大容量数据的时候，运行时间会变得无限长，所以这里凭借多次运行的经验，判断是否能过运行成功，（所以这可能会导致我的测试结果存在一定误差）。

表 4-1 不同相联度下的 cuckoo hash 的运行结果

相联度	1	2	4	8
占用率/%	57.45	87.45	96.95	98.45
平均驱逐次数	0.52	25.82	33.13	131.80
每元素插入平均用时/us	12	69	90	856