华中科技大学

大数据存储管理课程报告

姓　　名：　　武文博

学　　院：　　计算机科学与技术

专　　业：　　计算机科学与技术

班　　级：　　CS2109

学　　号：　　U202110020

指导教师：　　华宇

分数	
教师签名	

2024 年 4 月 20 日

1. 背景

哈希表，也称散列表，是一种常见的数据结构，它可以快速插入、删除和查找元素，通过将键值通过哈希函数映射到表中的某一位置来实现数据的快速检索。然而，哈希冲突的存在可能会对检索和存储效率产生影响。Cuckoo hash就是这个问题的一种可能的解决方案，Cuckoo 哈希是一种查找开销十分稳定的哈希算法，这是其相较于开放地址法、链表法的显著特征，它利用多个哈希函数生成多个键的哈希值，为每个键创建多个候选位置。当出现哈希冲突时，被替换的元素会去寻找自己的下一个候选位置存储。

Cuckoo 哈希可以视作是开放地址法的一种进阶，其基础算法描述为：在一个由一维数组构成的哈希表中，采用两个哈希函数 h1,h2，使得任意一个元素在哈希表中存在 2 个存储位，最基础的 Cuckoo 哈希实现思路如下：

1. 查询元素

当查询哈希表中元素 x 时，我们分别通过两个哈希函数$h1,h2$计算出其 2 个可能存在的位置，然后进行(并行)查询，如果在两个位置中存在元素 x，则存在。否则，不存在。

2. 插入元素

当在哈希表中插入元素 x 时，我们先进行查询，如果 x 已经存在，直接返回。否则，我们按照 2 个哈希函数顺序，依次计算其位置$i = h1(x), j = h2(x)$，如果$T[i], T[j]$ 任一位置空闲，则将 x 放置在其中一个位置上；如果 2 个位置都不空闲，则可以将$T[i]$位置的元素 y 拿出，然后将 x 放置在$T[i]$的位置上，然后将 y 放置在$h1(y), h2(y)$中的另一个位置上去，以此递归，如果 y 所属的另一个位置仍被占用，则再将其拿出，再次放置，当递归踢元素的此处到达一定次数，则我们可以判定当前哈希表已经满了，需要重构哈希表大小。细节如下图 1-1 所示；

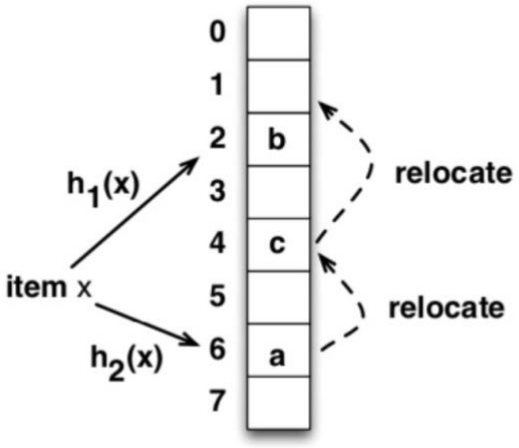

图 1-1

3. 删除元素

当在哈希表中删除元素 x 时，按照查询的逻辑找到所属位置，将其直接删除即可。

同时 Cuckoo 哈希也有一些问题，例如可能会出现无限循环和冲突等。因此，对该算法进行优化与改进对于提升数据处理性能有着非常重要的意义。在本次实验中，我考虑应用多路相联和路径检测等策略来解决这些问题，来进一步提升 Cuckoo 哈希的性能，减少冲突和无限循环的出现。

2. Cuckoo Hash 设计

2.1 哈希表设计

Cuckoo 哈希是一种独特的哈希表设计，它利用了多个哈希函数来决定一个键的可能存在位置。也就是说，在 Cuckoo 哈希表中，每个键并非仅有一个固定的位置，相反，它有多个备选位置供我们选择。在哈希冲突出现时，这个设计发挥出了巨大的优势。此时被挤出来的元素不会在原地等待，而会去寻找它的备选位置，以实现其再次存入哈希表。在本次实验中我使用 C++ 实现 Cuckoo Hash。

```
template <typename K, typename V>
class CuckooHashTable {
   private:
    std::shared_ptr<spdlog::logger> logger =
        spdlog::stderr_color_mt("cuckoo_hash_table");
    std::vector<std::pair<K, V>> table1;
    std::vector<std::pair<K, V>> table2;
    std::vector<bool> null_bits1;
    std::vector<bool> null_bits2;
    size_t table_size;
    uint32_t max_retry;
    uint32_t entry_cnt;
}
```

table1 和 table2 是存储哈希表键值对的向量，在这里使用 std::pair<K, V> 类型的向量来存储键值对，其中 K 是键的类型，V 是值的类型，null_bits1 和 null_bits2 用于标记哈希表对应位置是否为空的向量。如果 null_bits1[i]为 true，则表示 table1[i]位置为空；如果为 false，则表示该位置已被占用。table_size 用于存储哈希表大小的变量，max_retry 存储最大重试次数的变量。如果在尝试插入一个元素时发生冲突，哈希表会尝试把已有的元素移到不同的位置以腾出空间，entry_cnt 用来表示哈希桶的槽位数。

在本次实验中，我实现的是 2-array Cuckoo Hash，其采用两个数组分别关联一个哈希函数，然后放置元素、踢出元素在这两个数组之间进行，元素 x 在第一个数组中的位置是$h1(x)$，在第二个数组中的位置是$h2(x)$。如下图 2-1 所示：

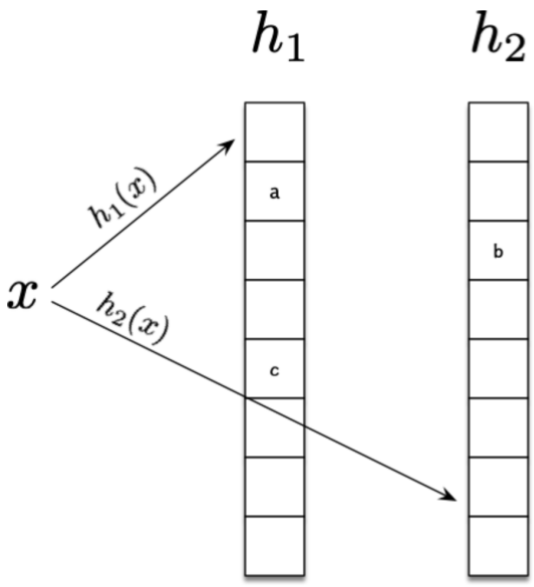

图 2-1

2.2 哈希表方法

1. 插入元素

```
1. bool insert(const K& key_, const V& val_)
```

首先进行一个随机次数（最多为 max_retry 次）的循环，试图将键值对插入到哈希表中。在这个过程中，首先检查哈希值 hash1 和 hash2 在两个哈希表中对应的位置是否为空。如果 table1 的 hash1 位置为空，那么将键值对插入到该位置并将该位置的 null 值标志置为 false，然后返回 true 表示成功插入；同样的，如果在 table2 的 hash2 位置为空，将键值对插入该位置并返回 true。

如果 hash1 和 hash2 在哈希表对应的位置都已被占用，代码使用一个均匀分布的随机数选择器 dist 选择一个表（table1 或 table2），然后将所选表的相应位置的键值对（即应被置换项 evict item）改为新的键值对。这是 Cuckoo Hash 的 evict 驱逐操作。

```
1.                 std::uniform_int_distribution<int> dist(1, 2);
2.                 int table_index = dist(generator);
3.                 if (table_index == 1) {
4.                     K evict_key = table1[hash1].first;
5.                     V evict_val = table1[hash1].second;
6.                     table1[hash1] = make_pair(key, val);
7.                     key = evict_key;
8.                     val = evict_val;
```

```
                hash1 = get_first_hash(key);
                std::cout << "evicted from table1: hash" << hash1
                          << ", key: " << key << ", val: " << val
                          << std::endl;
            } else {
                K evict_key = table2[hash2].first;
                V evict_val = table2[hash2].second;
                table2[hash2] = make_pair(key, val);
                key = evict_key;
                val = evict_val;
                hash2 = get_second_hash(key);
                std::cout << "evicted from table2: hash" << hash2
                          << ", key: " << key << ", val: " << val
                          << std::endl;
            }
```

如果在 max_retry 次循环后仍然没有成功插入键值对，则说明需要扩大哈希表，于是调用 resize_hash_table()方法进行扩容，然后重新插入键值对。

2. 查找元素

```
    V find(const K& key) {
```

在我的实现中，find 函数中调用了 lookup 函数。

```
    bool lookup(const K& key, V& val) {
        size_t hash1 = get_first_hash(key);
        size_t hash2 = get_second_hash(key);
        if (table1[hash1].first == key) {
            val = table1[hash1].second;
            return true;
        } else if (table2[hash2].first == key) {
            val = table2[hash2].second;
            return true;
        }
        return false;
    }

```

可以看到 lookup 函数的目的是在哈希表中查找一个键，并返回是否找到。如果在 table1 的 hash1 位置或在 table2 的 hash2 位置找到了匹配的键，就将相应位置的值分别复制到 val，并返回 true 表示找到。如果在两个哈希表中都没有找到，那么返回 false。

3. 删除元素

```
    bool remove(const K& key)
```

采用软删除的方式，例如对于 table1，如果 table1 的 hash1 位置的键和给定键相等，就将 null_bits1[hash1]设置为 true,表 table1 的 hash1 位置现在是空的，同时返回 true 表示成功删除。

4. 哈希表扩容

```
1.    void resize_hash_table() {
```

在插入元素时，当驱逐次数超过指定次数时，需要对哈希表进行扩容处理，创建新的更大的表并将旧表的所有键值对复制到新表中。这里需要注意的是，由于旧表的 table_size 发生变化，所以需要对旧哈希表已经存在的 KV 重新进行 hash 计算并插入，为了避免大量数据拷贝，可以考虑使用一致性哈希算法优化原来的哈希算法。

2.3 Cuckoo hash 优化

1. 多路相连优化

在一个哈希桶中放置多个槽位，即设置的 entry_cnt 大小，也就是说哈希表中的每个位置不再只存储一个元素，而是存储一个元素列表。这样做的好处是可以减小哈希冲突的可能性，因为即使多个元素的哈希值相同，它们也可以存储在同一个桶的不同槽位中。但是，这也会增加哈希表的空间复杂。

```
1.    std::vector<std::vector<std::pair<K, V>>> table1;
2.    std::vector<std::vector<std::pair<K, V>>> table2;
```

2. D-Cuckoo Hash 优化

参考 Rocksdb 对 cuckoo hash 的优化，将常见的 2 数组实现扩展到多层实现，同时搭配多种哈希函数，如下图 2-2 所示，扩展到 4 层数组，这样能一定程度提高整体的填充率，减少 rehash 的次数。RocksDB 实现了基于多层 Cuckoo Hashing 的 SST 文件结构，最多支持 64 层 hash 表。充分利用其高效的查找优势。当 Cuckoo Hash 表层数增多时，哈希冲突时，RocksDB 采用广度优先搜索来搜索最合适踢出的元素。同时 RocksDB 支持将 Cuckoo hash 中的数据持久化到磁盘，这对于需要处理非常大的数据集并且需要在系统重启后能够恢复数据的应用非常有用。

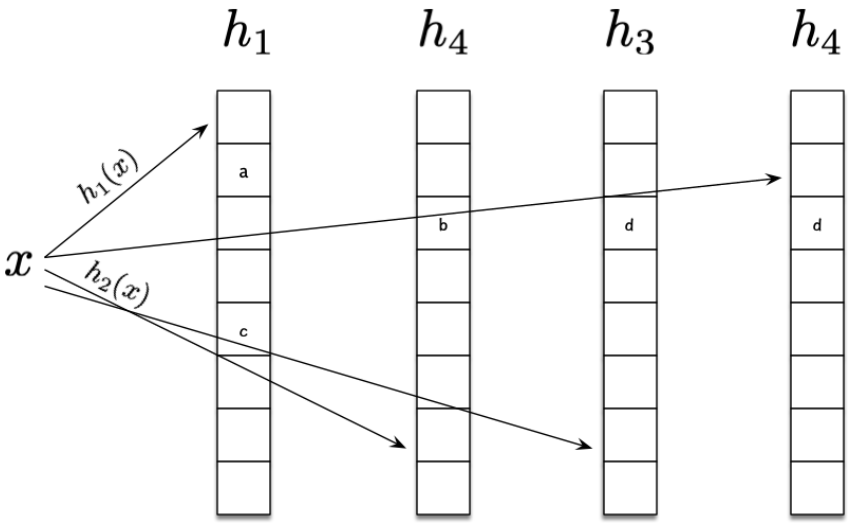

图 2-2

3. 存储优化

为了确保哈希表更加高效地使用存储空间。通过只存储元素的“指纹”(经过哈希运算生成的一种小型标识)来替代元素本身，这样可以显著减少哈希表所需的存储空间。

在这个方案中，元素的“指纹”与元素的键会经过一个异或运算。这是一个位操作，当两位相同返回 0，两位不同返回 1。由于异或运算的性质，我们可以通过对键和指纹再次进行异或运算来找到该元素在哈希表中的位置，即键 XOR 指纹 = 位置，这就使得可以在不需存储元素本身的情况下，依然能够迅速地找到元素的存储位置。

```
    std::hash<K> hash_fn;

    uint32_t fingerprint(const K& key) {
        return static_cast<uint32_t>(hash_fn(key));
    }

    uint32_t get_hash(uint32_t fingerprint, uint32_t i) {
        return (fingerprint + i) % buckets.size();
    }

```

4. 路径检测

首先需要有一种方式作为哈希表的图形表示，可以使用图数据结构，其中每个节点表示一个哈希桶。当插入一个元素时，如果其两个候选位置分别为 i1

和 i2，那么就在节点 i1 和 i2 之间添加一条路径。这条路径表示元素可以从节点 i1 被驱逐到 i2，或者从 i2 被驱逐到 i1。

然后，通过 dfs/bfs 检测是否存在一个可行的驱逐路径。需要记录哪些节点是有空闲位置的，因为这些节点是希望通过驱逐路径能够到达的目标节点，当需要插入一个元素并且触发驱逐操作时，就开始搜索一个可行的驱逐路径。如果找到了一条路径，就顺着这条路径将元素驱逐，直到元素被驱逐到一个有空闲位置的节点。

经过测试发现，寻找可行驱逐路径这个步骤可能会带来比较大的性能损失，特别是当哈希表变得非常大时。为了解决这个问题，需要设置一个驱逐次数阈值，当插入一个元素的过程中驱逐的次数大于这个阈值时，才进行驱逐路径的检测。同时也要根据数据量评估是否使用路径检测或者如何设置阀值。

3. Cuckoo Hash 测试分析

3.1 功能测试

在本次实验中，我使用 google test 框架对我的 Cuckoo Hash 进行测试，使用 spdlog 作为日志库，同时使用 cmake 构建程序。首先我对 Cuckoo Hash 的基本功能进行测试。

如下代码所示，我对插入元素进行测试，哈希表的大小为 10，同时插入了 21 个元素，Cuckoo Hash 具有两个哈希表一共可以存储 20 个元素，因此会出发哈希表扩容。

```
1. TEST(CuckooHashTest, InsertAndFindTest) {
2.     auto logger = spdlog::stderr_color_mt("insert_and_find_test");
3.     logger->set_level(spdlog::level::debug);
4.
5.     CuckooHashTable<int, std::string> hash_table(10);
6.
7.     for (int i = 0; i < 21; ++i) {
8.         ASSERT_TRUE(hash_table.insert(i, std::to_string(i)));
9.     }
10.    logger->debug("Inserted 21 elements");
11.
12.    for (int i = 0; i < 21; ++i) {
13.        EXPECT_EQ(hash_table.find(i), std::to_string(i));
14.    }
15.    logger->debug("Found 21 elements");
16. }
```

如下图 3-1 所示，测试插入功能通过。

```
evicted from table1: hash0, key: 0, val: 0
evicted from table1: hash0, key: 10, val: 10
evicted from table2: hash0, key: 20, val: 20
evicted from table2: hash0, key: 10, val: 10
evicted from table1: hash0, key: 0, val: 0
[2024-04-22 20:44:28.844] [cuckoo_hash_table] [info] Resizing hash table to 20
inserted into table1: hash: 0, key: 0, val: 0
[2024-04-22 20:44:28.845] [insert_and_find_test] [debug] Inserted 21 elements
[2024-04-22 20:44:28.845] [insert_and_find_test] [debug] Found 21 elements
[       OK ] CuckooHashTest.InsertAndFindTest (1 ms)
[----------] 1 test from CuckooHashTest (1 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (1 ms total)
[  PASSED  ] 1 test.
```

图 3-1

同理可以进行其他测试，测试如下图 3-2 所示：

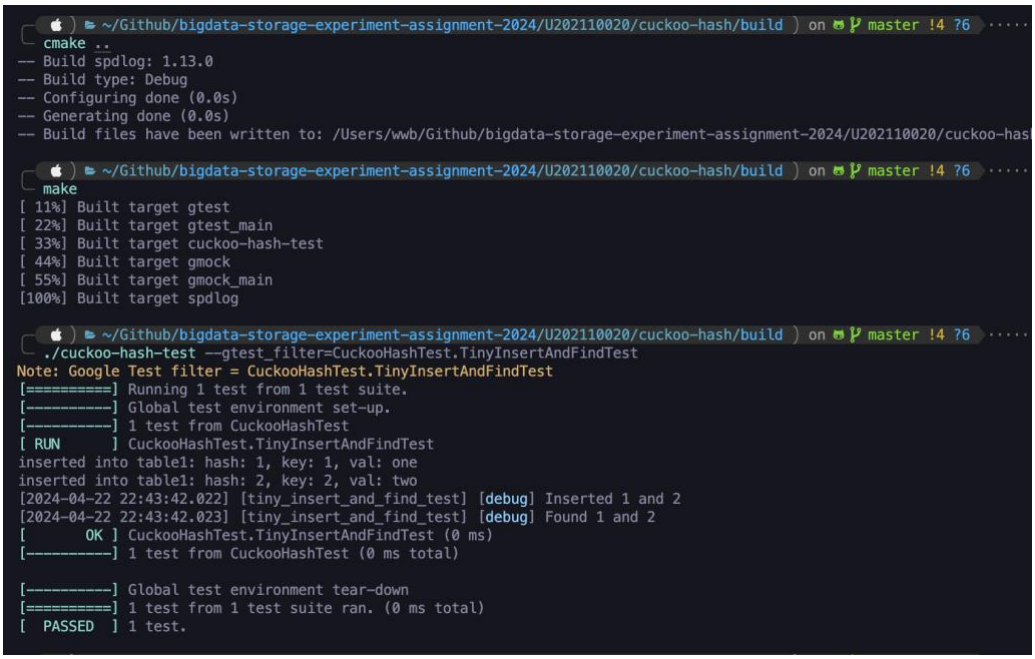

图 3-2

3.2 性能测试

在性能测试中，根据预设的数据规模生成指定范围内的 KV 数据，并将这些数据插入哈希表中。通过测量插入这些数据所需的总时间，可以计算出插入操作的延迟。进一步地，通过查看哈希表的当前大小，可以计算出空间利用率。而在插入数据之后，测试程序会逐一查询之前插入过的 int 数据，以验证它们是否可以成功地被查询出来，同时也可以通过测量查询这些数据所需的总时间来计算出查询操作的延迟。如下图所示，测试插入 1000000 个 KV 数据，哈希表的初始大小为 300000，在插入过程中会触发自动扩容，如下图 3-3 所示，可以看到空间利用率为 0.4166。

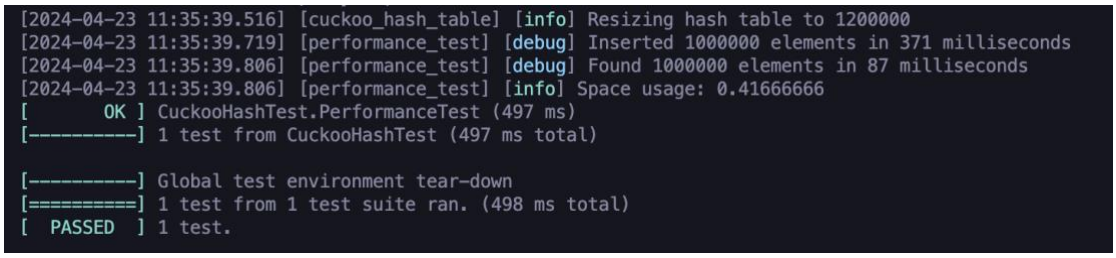

图 3-3

之后测试经过多路相连和存储优化后的 Cuckoo Hash，可以看到空间利用率提升，同时插入元素的总延迟也在下降，如下图 3-4 所示。

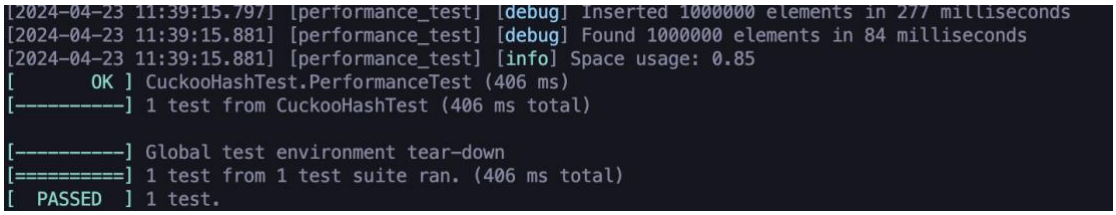

图 3-4

为了进行对比观察，我又分别测试了 Chained Hashing（桶链式哈希，哈希冲突时形成长长的一串链表），Linear Probing（当哈希冲突时，顺序找下一个位置的桶，看是否有元素...直到找到为空的位置），Cuckoo Hash 以及经过多路相连和存储优化后的 Cuckoo Hash。可以观察到 Cuckoo Hash 查找比 Chained Hashing 快，但平均性能低于 Liner Probe；插入，Cuckoo 插入性能是几种算法中最慢的，需要不断的 kick out；删除，Cuckoo 删除是几种算法中最快的。

4. 实验总结

在大数据存储的课程实验中，我选择学习和分析 Cuckoo Hash，深入理解其工作原理，对这一数据结构有了进一步的认识。Cuckoo Hash 是一种查找开销十分稳定的哈希算法，这是其相较于开放地址法、链表法的显著特征。在本次实验中，我也学习到了优化 Cuckoo Hash 的若干方法，例如多路相联、路径检测等，在实际测试过程中，我发现路径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，但有时也会增大插入元素的延时，可能因为我的路径检测的算法还存在一定问题，需要继续改进。同时我发现当相联度提升时，哈希表的占用率也随之提高，插入速率也会一定程度提升。

同时我也了解了 Cuckoo Hash 在实际工程项目中的应用，比如 RocksDB 中对 Cuckoo Hash 的进一步优化，以及用在过滤器方向，替代 bloom filter。在以后我会更加深入学习了解 Cuckoo Hash 的优化以及实际应用。