



2021 级

《大数据存储与管理》课程

实 验 报 告

姓 名 吴云鹏

学 号 U202115380

班 号 计算机 2103 班

日 期 2024.04.21

目录

一、 选题.....	3
二、 实验背景.....	3
三、 数据结构设计与实现.....	3
四、 操作流程分析.....	5
五、 理论分析.....	6
六、 实验测试的性能.....	10
七、 实验心得.....	11

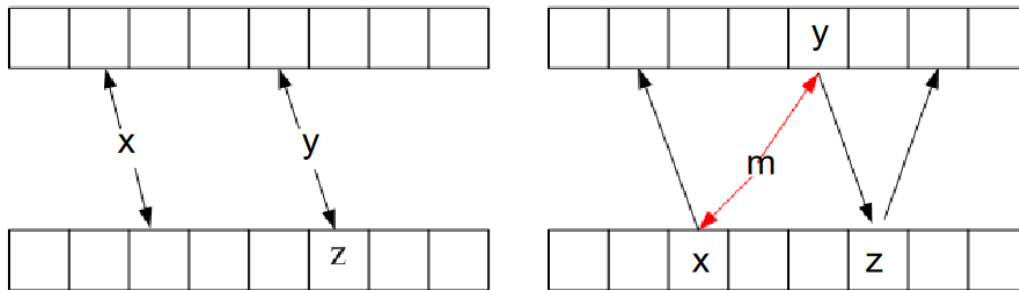
一、 选题

选题 3: Cuckoo-driven Way

如何确定循环，减少 cuckoo 操作中的无限循环的概率和有效存储。

二、 实验背景

Cuckoo Hash Table 使用了两个哈希函数来解决冲突。Cuckoo 查询操作的理论复杂度为最差 $O(1)$ ，而 Cuckoo 的插入复杂度为均摊 $O(1)$ 。我们引入 Cuckoo 是希望它在实际应用中，能够在较高的空间利用率下，仍然维持不错的查询性能。



Insert item x and y

Insert item m

循环产生：当要向哈希表中插入一个关键字时，首先通过两个不同的哈希函数计算出两个哈希位置。检查这两个位置，如果其中一个位置为空，则将关键字插入到该位置，插入完成。如果两个位置都已经被占据了，则选择其中一个位置插入关键字，并将原先位置的关键字挤出来。被挤出来的关键字会尝试插入到另一个哈希位置，如果那个位置也被占据了，就继续挤出去，如果每一个去到的位置都有了元素，则可能出现循环。

三、 数据结构设计与实现

按照 Cuckoo-driven 的特点设计数据如下：

```
class CuckooHashTable {
private:
    vector<int> table1;
    vector<int> table2;
public:
    CuckooHashTable() { //提供初始化函数
        table1.resize(TABLE_SIZE, -1);
```

```

        table2.resize(TABLE_SIZE, -1);
    }

    bool insert(int key) {
        return insertHelper(key, 0);
    }

    bool insertHelper(int key, int tableIdx) { //key 为放入的数据, tableIdx 为当前处理的 hash 表
        if (tableIdx == 0) { //对 hash 表 1 进行判断
            size_t hashValue = hash1(key) % TABLE_SIZE;
            if (table1[hashValue] == -1) { //hash 表 1 并未发生冲突, 放入数据
                table1[hashValue] = key;
                return true;
            }
            else { //hash 表 1 发生冲突, 将 1 的数据踢出
                int displacedKey = table1[hashValue];
                table1[hashValue] = key;
                return insertHelper(displacedKey, 1);
            }
        }
        else {
            size_t hashValue = hash2(key) % TABLE_SIZE;
            if (table2[hashValue] == -1) {
                table2[hashValue] = key;
                return true;
            }
            else {
                int displacedKey = table2[hashValue];
                table2[hashValue] = key;
            }
        }
    }

    bool search(int key) {
        size_t hashValue1 = hash1(key) % TABLE_SIZE;
        size_t hashValue2 = hash2(key) % TABLE_SIZE;
        if (table1[hashValue1] == key || table2[hashValue2] == key) { //检查 hash
            return true;
        }
        else {
            return false;
        }
    }

```

```
};
```

其中构造两个存储单元 table1, table2 用于存储实验数据,
insert() 函数用于插入数据,

insertHelper() 函数中使用 hash1(), 与 hash2() 函数对应 table1 与 table2 的插入, 用于
辅助 insert() 函数进行插入。

Hash1() 与 Hash2() 定义如下:

```
int hash1(int k) {  
    return k + 2;  
}
```

```
int hash2(int k) {  
    return k + 1;  
}
```

四、操作流程分析

在 main() 函数中检查操作是否完成:

```
int main() {  
    double average_time = 0; // 平均值  
    double average_false = 0; // 平均错误个数  
    double totaltime = 0;  
    double sum_false = 0;  
  
    int j;  
    for (j = 0; j < 10000; j++) {  
        CuckooHashTable hashTable;  
        // 创造随机数  
        vector<int> randoms;  
        randoms = random_creator();  
        // 插入随机数  
        auto start = chrono::high_resolution_clock::now();  
        for (int i : randoms) {  
            hashTable.insert(i);  
        }  
        auto end = chrono::high_resolution_clock::now();  
        auto duration = chrono::duration_cast<chrono::nanoseconds>(end - start);  
        totaltime += duration.count();  
        int false_cnt = 0;
```

```

        // 查询一些元素, 统计
        for (int i : randoms) {
            if (hashTable.search(i) == false) false_cnt++;
        }
        cout << false_cnt << endl;
        sum_false += false_cnt;
    }
    average_false = sum_false / 10000;
    average_time = totaltime / 10000;
    cout << "插入完成时间为: " << average_time << " nanoseconds" << endl;
    cout << "平均无法插入率为: " << average_false;
    return 0;
}

```

在 main 函数中, 调用 random_creator 函数生成[0, 10000]集合通过记录开始循环时的时间与循环结束的时间来统计获得运行循环所偶花费的总时间, 使用对于数据的查询, 记录未在存储单元中的数据从而获得有效存储率。

```

vector<int> random_creator() {
    random_device rd;
    mt19937 gen(rd());

    // 生成从 1 到 10000 的序列
    vector<int> sequence(10000);
    vector<int> randoms(0);
    iota(sequence.begin(), sequence.end(), 1);
    // 打乱序列
    shuffle(sequence.begin(), sequence.end(), gen);
    copy(sequence.begin(), sequence.begin() + 1000, back_inserter(randoms));
    return randoms;
}

```

五、理论分析

False Positive:

当我们进行查找操作时, 如果哈希表错误地将一个不存在的键标记为存在, 则发生了假阳性。换句话说, 哈希表错误地报告了一个元素存在于哈希表中, 但实际上该元素未被插入。

False Negative:

当我们进行查找操作时, 如果哈希表错误地将一个已存在的键标记为不存在, 则发生了假阴性。换句话说, 哈希表错误地报告了一个元素不存在于哈希表中, 但实际上该元素已经被插入。

出现可能：

哈希函数设计不合适：

如果哈希函数设计不合适，可能导致不同的键被映射到相同的哈希桶中，从而引发冲突。如果在冲突处理过程中没有正确处理这些冲突，就会导致插入操作失败，进而出现假阳性或假阴性。

冲突处理策略不足：

如果哈希表的冲突处理策略不够有效，比如简单地使用线性探测法或二次探测法，可能会导致冲突处理不彻底，使得一些元素无法正确插入到哈希表中，从而产生假阳性或假阴性。

负载因子过高：

当哈希表的负载因子过高时，即哈希表中存储的键值对数量与哈希表的大小之比超过了一个阈值，可能会导致哈希冲突的概率增加。如果在负载过高的情况下没有采取有效的哈希表调整策略，也会增加出现假阳性或假阴性的可能性。

哈希表大小不足：

如果哈希表的大小不足以容纳所有要插入的元素，可能会导致一些元素无法正确插入，从而出现假阳性。此外，如果哈希表的大小不足以提供足够的空间来减少冲突，也会增加冲突的发生率，从而导致假阳性或假阴性。

优化方案；

存储区：

使用 tables 替换 table1 与 table2

使用多个 hash 函数来使得其产生更好的分散率减少死循环的可能性。

对于 hash 函数的优化：

使用更好的 hash 函数提高分散率：本次使用 MurmurHash3 和 FNV-1a 两种哈希函数替代了原先的简单哈希函数。

两个函数的实现如下：

```
void MurmurHash3_x86_32(const void* key, int len, uint32_t seed, void* out) {
    const uint8_t* data = (const uint8_t*)key;
    const int nblocks = len / 4;

    uint32_t h1 = seed;

    const uint32_t c1 = 0xcc9e2d51;
    const uint32_t c2 = 0x1b873593;

    // Body
    const uint32_t* blocks = (const uint32_t*)(data + nblocks * 4);
```

```

for (int i = -nblocks; i; i++) {
    uint32_t k1 = blocks[i];

    k1 *= c1;
    k1 = (k1 << 15) | (k1 >> 17); // ROTL32(k1, 15);
    k1 *= c2;

    h1 ^= k1;
    h1 = (h1 << 13) | (h1 >> 19); // ROTL32(h1, 13);
    h1 = h1 * 5 + 0xe6546b64;
}

// Tail
const uint8_t* tail = (const uint8_t*)(data + nblocks * 4);
uint32_t k1 = 0;
switch (len & 3) {
case 3:
    k1 ^= tail[2] << 16;
    [[fallthrough]];
case 2:
    k1 ^= tail[1] << 8;
    [[fallthrough]];
case 1:
    k1 ^= tail[0];
    k1 *= c1;
    k1 = (k1 << 15) | (k1 >> 17); // ROTL32(k1, 15);
    k1 *= c2;
    h1 ^= k1;
};

// Finalization
h1 ^= len;
h1 ^= h1 >> 16;
h1 *= 0x85ebca6b;
h1 ^= h1 >> 13;
h1 *= 0xc2b2ae35;
h1 ^= h1 >> 16;

*(uint32_t*)out = h1;
}

// FNV-1a
size_t fnv1a(const void* key, size_t len) {

```

```

const unsigned char* data = (const unsigned char*)key;
const size_t prime = 0x100000001B3ULL;
size_t hash = 0xcbf29ce484222325ULL;
for (size_t i = 0; i < len; ++i) {
    hash ^= data[i];
    hash *= prime;
}
return hash;
}

```

动态调整哈希表大小:

因为插入成功率和 hash 表的大小挂钩, 数据较多时 hash 表小会导致死循环率高, 数据较少时给较大的 hash 表会导致空间浪费率高, 故改用动态 hash。

在 insert 函数中, 每次插入操作后会检查当前的负载因子是否超过了阈值。

如果负载因子超过了阈值, 则调用 resize 函数进行哈希表的动态调整。

resize 函数会将哈希表的大小扩大一定的倍数 (TABLE_RESIZE_FACTOR), 然后重新哈希并插入所有已有的键, 以减少负载因子。

// 动态调整哈希表大小

```

void resize() {
    int newSize = TABLE_RESIZE_FACTOR * size;
    vector<vector<int>> newTables(2, vector<int>(newSize, -1));

    // 重新哈希并插入所有键
    for (auto& table : tables) {
        for (int key : table) {
            int tableIdx = (&table - &tables[0]);
            size_t hashValue = hash(key, tableIdx);
            newTables[tableIdx][hashValue] = key;
        }
    }

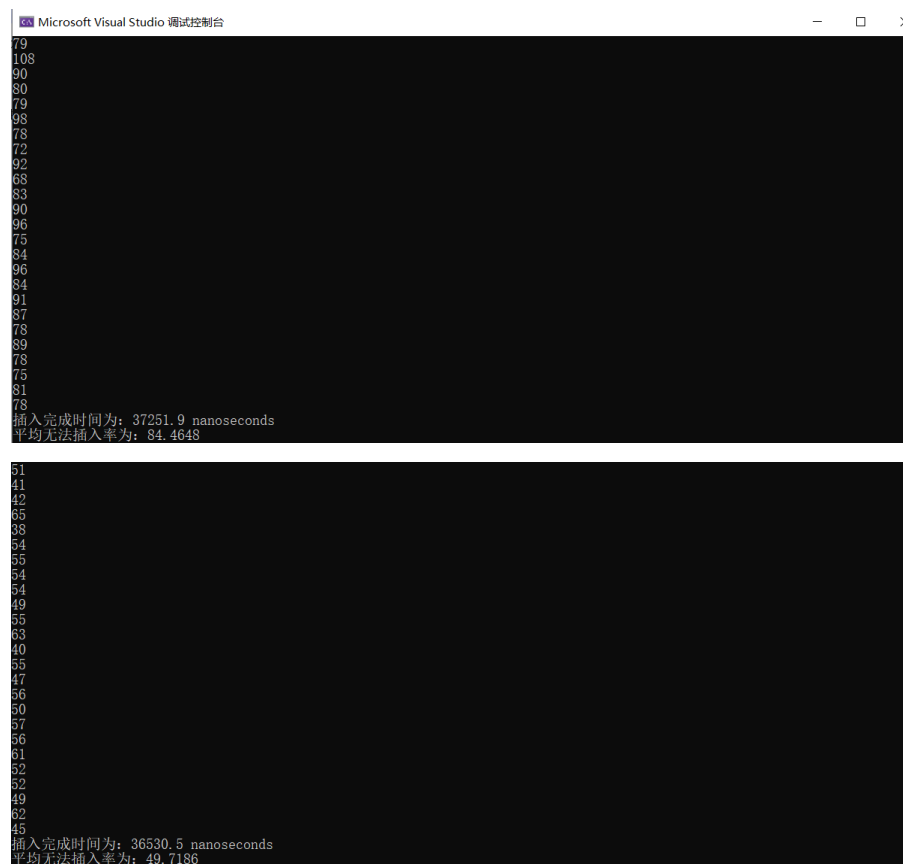
    size = newSize;
    tables = move(newTables);
}

```

六、实验测试的性能

设置处理数据大小为 1000

通过两次测试获得实验结果如下：



The image displays two screenshots of the Microsoft Visual Studio 调试控制台 (Debug Console) window. The window title is "Microsoft Visual Studio 调试控制台". The first screenshot shows a list of numbers (79, 108, 90, 80, 79, 98, 78, 72, 92, 68, 83, 90, 96, 75, 84, 96, 84, 91, 87, 78, 89, 78, 75, 81, 78) followed by the text "插入完成时间为: 37251.9 nanoseconds" and "平均无法插入率为: 84.4648". The second screenshot shows a list of numbers (51, 41, 42, 65, 38, 54, 55, 54, 54, 49, 55, 63, 40, 55, 47, 56, 50, 57, 56, 61, 52, 52, 49, 62, 45) followed by the text "插入完成时间为: 36530.5 nanoseconds" and "平均无法插入率为: 49.7186".

```
Microsoft Visual Studio 调试控制台
79
108
90
80
79
98
78
72
92
68
83
90
96
75
84
96
84
91
87
78
89
78
75
81
78
插入完成时间为: 37251.9 nanoseconds
平均无法插入率为: 84.4648

51
41
42
65
38
54
55
54
54
49
55
63
40
55
47
56
50
57
56
61
52
52
49
62
45
插入完成时间为: 36530.5 nanoseconds
平均无法插入率为: 49.7186
```

可知，存储成功率由 $(1000 - 84.4648) / 1000 = 91.3\%$

优化为 $(1000 - 49.7186) / 1000 = 95.03\%$

优化率为 5%。

同时存储时间性能提升 2.7%

七、实验心得

在本次实验课中学习了大数据的存储体系，学会搭建大数据存储系统，了解了经典的大数据存储数据结构，课程报告中选择了 Cukoo driven-hash，通过阅读论文，对其有了大致认识，成功优化了有效存储率等关键性能，并提高了自己论文阅读能力，

相关网页连接:

[Murmur 哈希 - 维基百科, 自由的百科全书 \(wikipedia.org\)](#)

[MurmurHash3 最详细的介绍-CSDN 博客](#)

[hash 算法 Fnv-1a-CSDN 博客](#)

参考文献:

- R. Pagh and F. Rodler, “Cuckoo hashing,” Proc. ESA, pp. 121–133, 2001.
- Yu Hua, Hong Jiang, Dan Feng, “FAST: Near Real-time Searchable Data Analytics for the Cloud”, Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), November 2014, Pages: 754–765.
- Yu Hua, Bin Xiao, Xue Liu, “NEST: Locality-aware Approximate Query Service for Cloud Computing”, Proceedings of the 32nd IEEE International Conference on Computer Communications (INFOCOM), April 2013, pages: 1327–1335.
- Qiuyu Li, Yu Hua, Wenbo He, Dan Feng, Zhenhua Nie, Yuanyuan Sun, “Necklace: An Efficient Cuckoo Hashing Scheme for Cloud Storage Services”, Proceedings of IEEE/ACM International Symposium on Quality of Service (IWQoS), 2014.
- B. Fan, D. G. Andersen, and M. Kaminsky, “MemC3: Compact and concurrent memcache with dumber caching and smarter hashing,” Proc. USENIX NSDI, 2013.
- B. Debnath, S. Sengupta, and J. Li, “ChunkStash: speeding up inline storage deduplication using flash memory,” Proc. USENIX ATC, 2010.