华中科技大学

大数据存储系统与管理报告

姓　名：　王彬

学　院：　计算机科学与技术学院

专　业：　计算机科学与技术

班　级：　计卓 2101 班

学　号：　U202112071

指导教师：　施展

分数	
教师签名	

2024 年 4 月 22 日

目 录

1 选题描述

考虑 Cuckoo-Hash 操作时，每次添加入新元素如果恰巧与旧元素所占用的位置冲突，需要将旧元素逐出，并让旧元素重新寻找新的哈希地址。因此我们的 Cuckoo-Hash 操作可以表示如下：

Cuckoo-Hash 具有 insert(T Value)和 query(T Value)两种操作，也就是插入和查询操作。Cuckoo-Hash 具有若干个桶，每个桶对应一个哈希函数，保证每个哈希函数不具备相关性。

- 查询操作的流程为：

1. 遍历所有桶，如果该桶内本对象所对应的哈希键值的对应槽位有预期元素，则目标存在；

2. 如果未找到，则失败退出。

- 插入操作的流程为：

1. 遍历所有桶，如果其中存在某个桶，该对象对应的哈希键值位上没有元素占据，则将对象插入至其中；

2. 如果所有桶的对应哈希键值位上均有元素，那么则需要根据一定的策略逐出某个对象，并对该对象重新分配桶空间；

3. 重复步骤 2，直到被逐出的对象找到存储空间。若陷入死循环，则需要按照一定的策略重新分配空间。

特别地，在本次选题中我们使用**多槽位（k-slot）重分配策略**优化 Cuckoo 哈希方法，可见效率有显著提升。

2 数据结构设计

2.1 HashF

我们首先自定义一个哈希函数族。HashF 函数通过对字符串中的每个字符执行按位异或操作，并左移一个与哈希选项对应的质数位数（hash_key_primes[num]）来计算哈希值，然后对结果取模以确保它在哈希表大小范围内。这里的 BUCKET_SIZE 是哈希表的大小。哈希代码如下：

```
/*
 @func:HashF
```

```
 @params: 待哈希对象 str，哈希选项 num
 @Hash 函数，满足 0<=num<10，使用质数进行字符串哈希
 */
unsigned int HashF(const std::string& str, int num){
    if (num < 0 || num >= 10){
        throw "Number of Hash Map cannot satisfy needs.";
    }
    unsigned int ret = 0;
    for (auto i:str){
        ret ^= i;
        ret <<= hash_key_primes[num];
    }
    return ret % BUCKET_SIZE;
}

/*
 @func:fingerPrint
 @params: 待哈希对象 str
 @输出元素的哈希指纹
 */
unsigned int fingerPrint(const std::string& str){
    return (unsigned)szHash(str) % P + 1;
}
```

2.2 槽位设计

我们使用多槽的设计减少哈希冲突的可能性。多槽设计，即每一个哈希桶内含有若干个槽位，以存放哈希表到值表（ValueTable）的指针。通过对于值表统一使用 std::vector<std::string>存放的方式，将哈希表与对象存储分开来，缩减哈希表的空间开销。

2.2.1 超参数

我们对于多槽CuckooFilter的建立使用动态扩容的方式构建，也就是拟定好一系列哈希函数，对多桶操作进行支持。其中具体的超参数如下：

```
int hash_key_primes[10] = {2,3,5,7,11,13,17,19,23,29}; // 质数，哈希使用
const unsigned int ITER_ROUNDS_MAX = 15;    // Cuckoo Insert 循环次数最大值
const unsigned int P = 1e9+7;
const double conflict_rate = 0.5; // 扩容阈值比例
const int NUM_SLOTS = 8; // 槽位个数
```

这里的超参数可以再编译之前进行更改，例如在实际调试的时候发现，如果将槽位个数 NUM_SLOTS 增大的同时，减小桶容量 BUCKET_SIZE，尽管桶哈希占用的地址空间大小没有发生变化，却可以减少 67%的时间使用。这是因为适当的槽位个数可以减少冲突的产生。

2.2.2 成员变量及函数定义声明

相关的成员变量如下所示。NUM_BUCKETS 位当前桶的数量，如果检测到容量不够（容量超过阈值或发生了失败的插入）将会重新构建哈希表；这可以有多种方式进行扩容，例如增加桶数量、增加桶大小、增加槽个数等等。这里我们采取增加桶数量的方式进行扩容。

```
int NUM_BUCKETS = 2; // 桶的数量，即当前哈希函数的个数
int BUCKET_SIZE = 250; // 每个桶的大小

// 结果输出
int insert_failure = 0;
int insert_total = 0;
float time_consumed_for_reconstruction = 0.0;

unsigned int *T; // 哈希表
std::vector<std::string> ValueTable;    // 值表，0 作空
std::hash<std::string> szHash;  // hash 指纹函数
```

```
std::mt19937 rng; // 用于生成随机数
std::uniform_int_distribution<int> distribution_alloc; // 均匀分布
```

这里使用的随机数生成器，是为了在哈希冲突产生且无法利用多槽设计规避时，使用 Cuckoo 策略判断下一个可能适合的地址。当然，除了随机分配外，可以通过图论的方式寻找较优的最小化冲突次数的解，我们基于桶内元素个数，期望选择较优的下一节点。

2.2.3 成员函数

我们定义下列函数，实现布谷鸟插入、查询等基本操作。

```
Private:
    unsigned int HashF(const std::string& str, int num)；
    unsigned int fingerPrint(const std::string& str)；
    bool Realloc_insert(const std::string& item, unsigned pos)；
    void resize()；
    // 触发重做 Hash 的策略
    void TriggerResize(int op);
    // 随机选择一个哈希表进行替换
    int chooseTable(int mod);
    // 插入元素到指定的哈希表，如果成功插入返回 TRUE，插入失败返回 FALSE
    bool insertToTable(const std::string& item, int table_index);
    // 插入指定元素到指定的哈希表，如果成功插入返回 TRUE，插入失败返回 FALSE
    bool insertSpecificItemToTable(const std::string& item, unsigned ptr, int table_index)

Public:
    // 构造函数
    CuckooMap() : rng(std::random_device()()), distribution_alloc(0, 1)
    // 析构函数
```

```
    ~CuckooMap(){}
    // 进行 T 元素的插入
    bool insert(const std::string& item);
    // 检查对象是否存在
    bool query(const std::string& item);
```

以及查询字段值函数若干，此处不再赘述。

3 操作流程分析

我们的操作过程主要分为下列几个调用模块：

1. 哈希键值的取得和解析
2. 插入和查询功能实现
3. 重分配大小，对哈希桶进行重做
4. 选择跳转策略

3.1 哈希键值计算

这部分以及在 2.1 节说明，此处不再赘述。

3.2 插入和查询操作支持

对于插入操作，首先在所有哈希桶中寻找目标对象 hash 地址。查询所有槽位，如果所有槽位中均以有占用元素，则按照一定的策略逐出原先的元素，“鸠占鹊巢”让原有元素进行寻址。

```
        /*
         @func:insert
         @params: 重新插入字符串 item
         @进行 T 元素的插入
         */
        bool insert(const std::string& item) {
           insert_total++;
           // insert to ValueTable
           ValueTable.push_back(item);

           // CuckooMap：只要找到一个桶的 key 对应槽位有空缺，则成功插入，否则执行随机踢出操作
           for (int i=0;i<NUM_BUCKETS;++i){
              if (insertToTable(item, i)) return true;
           }
```

```
            // pointer of Item x
            unsigned int ptr = (unsigned int)ValueTable.size() - 1;
            unsigned int current_bucket = chooseTable(NUM_BUCKETS);
            unsigned int slot_index = chooseTable(NUM_SLOTS);
            std::string str(item);

            // kick-out policy
            for (int i = 0;i < ITER_ROUNDS_MAX;++i){
               // pick a random table to kick out
               // pick a random slot to kick out
               unsigned key = HashF(ValueTable[ptr], current_bucket);
               unsigned array_index = NUM_SLOTS * (BUCKET_SIZE *
current_bucket + key) + slot_index;
               // 将目标移出的对象尝试插入
               if (insertSpecificItemToTable(ValueTable[ptr], ptr, current_bucket))
                  return true;

               std::swap(ptr, T[array_index]);
               current_bucket = chooseTable(NUM_BUCKETS);
               slot_index = chooseTable(NUM_SLOTS);
            }
            // 如果失败，则考虑对桶进行扩容策略或增加 hash 函数桶
            insert_failure++;
            TriggerResize(1);
            return false;
   }
```

若仍然发生死循环，说明原有哈希策略的效率已经较低，也就是说哈希表中占有元素对于原哈希表总容量的占比较大，可能已经超过阈值。这里我们发现我们给出的阈值（0.5）较大，应做尝试再调整到合适的值。

查询操作可以在 O(1)的时间中求得。具体的操作是对于每个哈希桶找到其哈希键，对于每个槽位进行检查，查看其中存放的指针是否指向目的对象。其代码如下：

```
        /*
         @func:query
         @params: 字符串 item
         @检查对象是否存在
         */
        bool query(const std::string& item) {
           for (int table_index = 0;table_index < NUM_BUCKETS; ++table_index){
              unsigned key = HashF(item, table_index);
              for (int offset = 0; offset < NUM_SLOTS; ++offset){
                 unsigned array_index = NUM_SLOTS * (BUCKET_SIZE *
table_index + key) + offset;
                 if (ValueTable[T[array_index]]==item) return true;
              }
           }
           return false;
   }
```

在执行具体的插入操作时，可调用下列函数：

```
        // 插入指定元素到指定的哈希表，如果成功插入返回 TRUE，插入失败返
回 FALSE
        bool insertSpecificItemToTable(const std::string& item, unsigned ptr, int
table_index) {
           // 取得元素指向的指针
           unsigned int get_vector_pointer = ptr;
           unsigned key = HashF(item, table_index);
           // T[NUM_BUCKETS][BUCKET_SIZE][NUM_SLOTS]
           unsigned array_index = NUM_SLOTS * (BUCKET_SIZE * table_index +
key);
```

```
        // 尝试所有的槽位，如果都满则失败退出
        for (int offset = 0; offset < NUM_SLOTS; ++offset){
            if (!T[array_index + offset]) {
                T[array_index + offset] = get_vector_pointer;
                return true;
            }
        }
        return false;
    }
```

3.3 扩容与哈希桶重分配

如果发生错误插入，如死循环等情形发生时，应采取哈希扩容的方式加以解决。我们这里采取了增加哈希桶的方式进行扩容，事实上还可以采取其它的方式进行哈希扩容，比如增加槽个数等。扩容的时候，需要对哈希链表进行空间重分配，对每个已有元素重新插入至哈希桶中，需要消耗的时间较多。

```
        /*
         @func:resize
         @重分配哈希函数
         */
        void resize(){
            // 增加哈希函数个数，同时改变桶的个数
            auto start = std::chrono::high_resolution_clock::now();
            ++NUM_BUCKETS;
            if (NUM_BUCKETS>=10) throw "Exception: NUM_PRIMES not enough.";
            // 删除旧序列
            delete[] T;
            // 大小等于桶个数*桶大小*槽个数
            T = new unsigned int[(NUM_BUCKETS*BUCKET_SIZE)*NUM_SLOTS];

            printf("====================New Hash==================\n");
```

```
            printf("Params:  Num_Buckets  =  %d;  Bucket_Size  =  %d\n",
NUM_BUCKETS,BUCKET_SIZE);

printf("================================================\n\n");

            // Todo: Write Insert Algorithm
            for (int i=0;i<ValueTable.size();++i){
               // 注意：不可以简单地添加
               Realloc_insert(ValueTable[i], i);
            }
            auto end = std::chrono::high_resolution_clock::now();
            std::chrono::duration<float> duration = end - start;
            std::cout << "Time consumed by reconstruction : " << duration.count() *
1000.0f <<" ms"<< std::endl;
            time_consumed_for_reconstruction += duration.count() * 1000.0f;

         }

         /*
          @func:TriggerResize
          @触发重做 Hash 的策略
          */
         void TriggerResize(int op){
            if (op == 1) resize();
   }
```

其中，扩容函数 Resize()中调用 Realloc_insert()函数，该函数和普通插入的区别在于，它不再在值表 ValueTable 中再次加入对象，而只对哈希表进行操作。

3.4 跳转策略选择

我们这里编写了两种跳转策略：一种是完全随机地选择下一跳转地址，事实上，随机游走本身就可以达到哈希桶均衡分配的效果；另一种是以一定的概率选择对象数最少的桶进行游走，再以一定的概率随机选择下一地址。事实上可以对每个元素建立图论模型，选择出边总数最小的哈希桶，可以有更大的概率减小哈希冲突。

```
        /*
         @func:chooseTable
         @随机选择一个哈希表进行替换
         */
        int chooseTable(int mod) {
            return distribution_alloc(rng) % mod;
    }
        // 踢出策略，选择目标 PT 表内值最小的元素
        unsigned int chooseWithDegree(unsigned ptr){
            // 有 1/3 概率选择随机跳出
            if (chooseTable(3)==0) return chooseTable(NUM_BUCKETS);
            // 剩下 2/3 概率选择当前最优值
            unsigned int min_degree = PT[0];
            unsigned int min_index = 0;
            for (int i=1;i<NUM_BUCKETS;++i){
                if (PT[i] < min_degree){
                    min_degree = PT[i];
                    min_index = i;
                }
            }
            return min_index;
        }
```

这里的 PT 表存放的是每个哈希桶的出边个数总和。我们根据测试发现，使用后一种方案比随机游走策略速度提高了 3%，但同时扩容时间也略有增加。

为了测试我们的 Cuckoo-Hash 的性能，我们还写了一些工具函数，如字符串数据生成器等，这里不再赘述。

4 理论分析

Cuckoo-Hash 不会发生误判，不存在原本不存在却误判为正确（False Negative），也不会存在被判定为负样本，实际上却为正样本的情形（False Positive）。也就是 FP=FN=0。

我们实现的 Cuckoo-Hash 的插入时间复杂度为 O(1+a)，查询时间复杂度则为 O(1)，这部分只和超参数及哈希效率有关。如果插入的元素过多，由于我们的哈希函数给定得有限，如果超过门限会抛出异常。同时，如果哈希桶空间开得过大，超过主存容量时，会引起磁盘 I/O，严重影响 Cuckoo 性能。

5 实验性能测试

首先我们编写功能程序如下：

```
CuckooMap filter;
filter.insert("apple");
filter.insert("orange");
filter.insert("banana");
filter.insert("grape");
std::cout << std::boolalpha;
std::cout << "Have apple? " << filter.query("apple") << std::endl;
std::cout << "Have grape? " << filter.query("grape") << std::endl;
std::cout << "Have pineapple? " << filter.query("pineapple") << std::endl;
```

测试结果如下：

```
wangbin@CHINAMI-TV5O8C1:/mnt/d/Administrator/Desktop/U202112071/
t$ ./"CuckooHash"
Have apple? true
Have grape? true
Have pineapple? false
```

我们再进行性能测试，对于 500、1000、1500、2000 个长度为 32 的字符串进行插入。

测试结果如下：当桶大小为 250，槽位数为 8，循环次数最大值为 15 时，有

Time consumed Total : 0.396 ms

#Insert_Total: 500 #Insert_Failure: 0 #Time_Consumed_by_Reconstruction: 0 ms

Time consumed Total : 0.4033 ms

#Insert_Total: 1000 #Insert_Failure: 2 #Time_Consumed_by_Reconstruction: 2.5454 ms

Time consumed Total : 1.5942 ms

#Insert_Total: 1500 #Insert_Failure: 0 #Time_Consumed_by_Reconstruction: 0 ms

Time consumed Total : 19.2958 ms

#Insert_Total: 2000 #Insert_Failure: 6 #Time_Consumed_by_Reconstruction: 50.2311 ms

```
wangbin@CHINAMI-TV5O8C1:/mnt/d/Administrator/Desktop/U202112071/bigdata-storage-experiment-assi
t$ ./"CuckooHash"
Time consumed Total : 0.396 ms
#Insert_Total: 500 #Insert_Failure: 0 #Time_Consumed_by_Reconstruction: 0 ms
Time consumed Total : 0.4033 ms
#Insert_Total: 1000 #Insert_Failure: 2 #Time_Consumed_by_Reconstruction: 2.5454 ms
```

可见随着插入对象个数的增加，由于插入失败的次数增加，重构哈希表的次数急剧增加，导致时间消耗的上涨。但是在小数据时，插入一个数据的时间在 0.792 纳秒，而在 1000 数据下测试，如果只计算插入时间（不包括重构时间），

平均插入时间可以达到 0.403 纳秒。

而在保持桶容量不变的时候，也就是槽位数乘单个桶哈希表项个数之积（我们暂取 2000）保持不变时，我们有下列测试结果。

Slot_Num = 4; Bucket_Size = 500;

Time consumed Total : 7.8519 ms

#Insert_Total: 2000 #Insert_Failure: 4 #Time_Consumed_by_Reconstruction: 6.701 ms

Slot_Num = 8; Bucket_Size = 250;

Time consumed Total : 5.5155 ms

#Insert_Total: 2000 #Insert_Failure: 2 #Time_Consumed_by_Reconstruction: 4.4959 ms

Slot_Num = 16; Bucket_Size = 125;

Time consumed Total : 2.0238 ms

#Insert_Total: 2000 #Insert_Failure: 0 #Time_Consumed_by_Reconstruction: 0 ms

可见在一定范围内增加槽个数可以有效减小哈希冲突的可能性，从而降低时间消耗。

而对于跳转策略的选择，我们对于 1. 随机跳转；2. 一定概率最少出边哈希表+一定概率随机跳转；这两种策略的测试分别如下：

```
wangbin@CHINAMI-TV5O8C1:/mnt/d/Administrator/Desktop/U202112071/bigdata-storage-experiment-assignment-2024
t$ ./"CuckooHash"
====================New Hash=================
Params: Num_Buckets = 3; Bucket_Size = 250
=============================================

Time consumed by reconstruction : 3.7162 ms
Time consumed Total : 15.3743 ms
#Insert_Total: 2000
#Insert_Failure: 1
#Time_Consumed_by_Reconstruction: 3.7162 ms
```

```
wangbin@CHINAMI-TV5O8C1:/mnt/d/Administrator/Desktop/U202112071/bigdata-storage-experiment-ass
t$ ./"CuckooHash"
====================New Hash=================
Params: Num_Buckets = 3; Bucket_Size = 250
=============================================

Time consumed by reconstruction : 4.5446 ms
Time consumed Total : 14.9719 ms
#Insert_Total: 2000
#Insert_Failure: 1
#Time_Consumed_by_Reconstruction: 4.5446 ms
```

可见策略 2 具有较小的优化，但在扩容时也提高了时间消耗，这可能因为计算策略的时间消耗。