2018 级

《物联网数据存储与管理》课程

实 验 报 告

姓　名　杨清帆

学　号　U201814758

班　号　物联网 1801 班

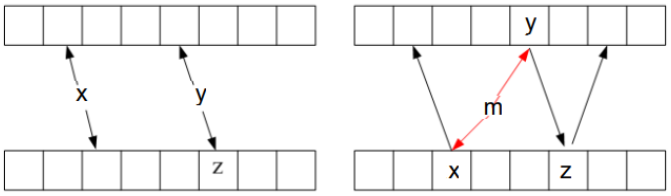
日　期　2021.06.21

目　　录

一、选题.....1
二、基本介绍.....1
2.1 技术背景.....1
2.2 Cuckoo Filter.....1
三、Cuckoo Filter 设计与实现.....3
参考文献.....9

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。



Insert item x and y　　Insert item m

二、基本介绍

2.1 技术背景

对于海量数据处理业务，我们通常需要一个索引数据结构用来帮助查询，快速判断数据记录是否存在，这种数据结构通常又叫过滤器(filter)。

索引的存储又分为有序和无序，前者使用关联式容器，比如 B 树，后者使用哈希算法。这两类算法各有优劣：关联式容器时间复杂度稳定 O(logN)，且支持范围查询；又比如哈希算法的查询、增删都比较快 O(1)，但这是在理想状态下的情形，遇到碰撞严重的情况，哈希算法的时间复杂度会退化到 O(n)。因此，选择一个好的哈希算法是很重要的。

bloom filter 的位图模式存在两个问题：一个是误报，在查询时能提供“一定不存在”，但只能提供“可能存在”，因为存在其它元素被映射到部分相同 bit 位上，导致该位置 1，那么一个不存在的元素可能会被误报成存在；另一个是漏报，如果删除了某个元素，导致该映射 bit 位被置 0，那么本来存在的元素会被漏报成不存在。由于后者问题严重得多，所以 bloom filter 必须确保“definitely no”从而容忍“probably yes”，不允许元素的删除。

为了解决这一问题，引入了一种新的哈希算法——cuckoo filter，它既可以确保元素存在的必然性，又可以在不违背此前提下删除任意元素，仅仅比 bitmap 牺牲了微量空间效率。

2.2 Cuckoo Filter

Cuckoo Hash（布谷鸟散列）是为了解决哈希冲突问题而提出，利用较少的计算换取较大的空间。

特点：占用空间少，查询速度快。

算法描述:使用 hashA、hashB 计算对应的 key 位置：

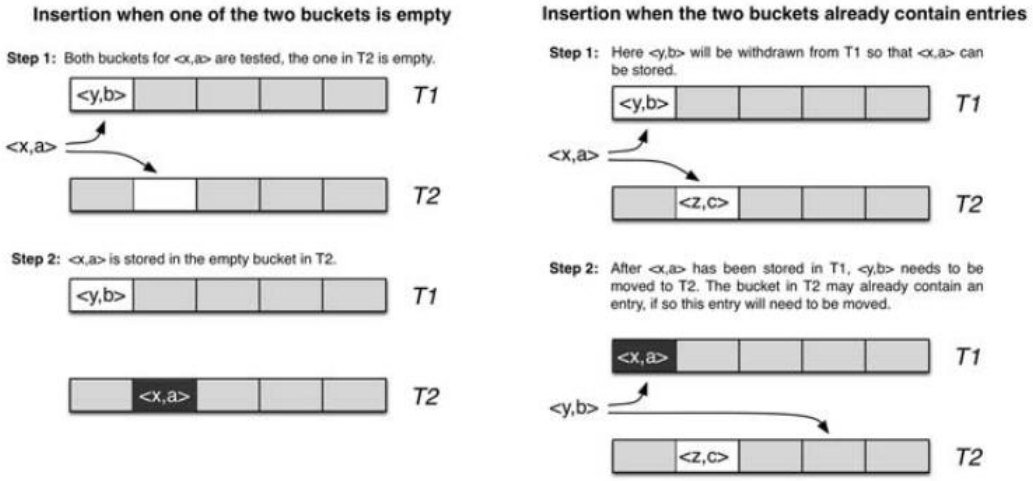
①两个位置均为空，则任选一个插入；

②两个位置中一个为空，则插入到空的那个位置；

③两个位置均不为空，则踢出一个位置后插入，被踢出的对调用该算法，再执行该算法找其另一个位置，循环直到插入成功。

④如果被踢出的次数达到一定的阈值，则认为 hash 表已满，并进行重新哈希。

cuckoo hashing 的哈希函数是成对的（具体的实现可以根据需求设计），每一个元素都是两个，分别映射到两个位置，一个是记录的位置，另一个是备用位置。这个备用位置是处理碰撞时用的，cuckoo hashing 处理碰撞的方法，就是把原来占用位置的这个元素踢走，不过被踢出去的元素还有一个备用位置可以安置，如果备用位置上还有人，再把它踢走，如此往复。直到被踢的次数达到一个上限，才确认哈希表已满，并执行 rehash 操作。



优化方式：

①将一维改成多维，使用桶（bucket）的 4 路槽位（slot）；

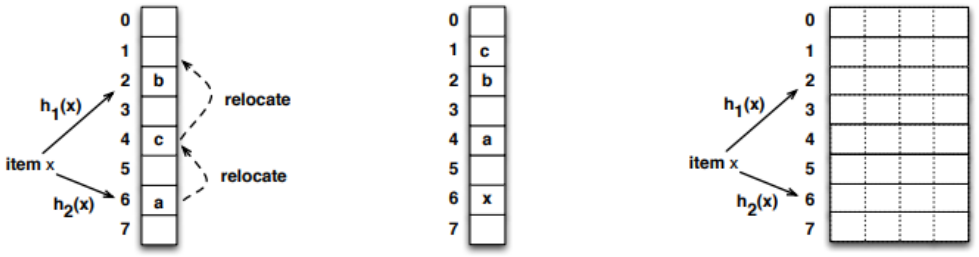
②一个 key 对应多个 value；

③增加哈希函数，从两个增加到多个；

④增加哈希表。

在发生哈希碰撞之前，一维数组的哈希表跟其它哈希函数没什么区别，空间利用率差不多为 50%。

一个改进的哈希表如下图所示，每个桶（bucket）有 4 路槽位（slot）。当哈希函数映射到同一个 bucket 中，在其它三路 slot 未被填满之前，是不会有元素被踢的，这大大缓冲了碰撞的几率。采用二维哈希表（4 路 slot）大约 80%的占用率（CMU 论文数据据说达到 90%以上，应该是扩大了 slot 关联数目所致）。



(a) before inserting item *x* (b) after item *x* inserted (c) A cuckoo filter, two hash per item and functions and four entries per bucket

三、Cuckoo Filter 设计与实现

假设有一段文本数据，我们把它通过 cuckoo filter 导入到一个虚拟的 flash 中，再把它导出到另一个文本文件中。flash 存储的单元页面是一个 log_entry，里面包含了一对 key/value，value 就是文本数据，key 就是这段大小的数据的 SHA1 值。

```
#define SECTOR_SIZE      (1 << 10)
#define DAT_LEN          (SECTOR_SIZE - 20)   /* minus sha1 size */
struct log_entry {
        uint8_t sha1[20];
        uint8_t data[DAT_LEN];
};
```

对于 DAT_LEN 设置，由于 flash 的单位是按页大小 SECTOR_SIZE 读写，这里假设每个 log_entry 正好一个页大小，也可以根据实际情况调整。

哈希表里的 slot 有三个成员 tag、status 和 offset，分别是哈希值、状态值和在 flash 的偏移位置。其中 status 有三个枚举值：AVAILIBLE、OCCUPIED、DELETED，分别表示这个 slot 是空闲的，占用的还是被删除的。对于 tag，因其中一个哈希值已经对应于 bucket 的位置上了，所以只要保存另一个备用 bucket 的位置就行，这样万一被踢，只要用这个 tag 就可以找到它的另一个的位置。

```
enum { AVAILIBLE, OCCUPIED, DELETED, };
struct hash_slot_cache {
        uint32_t tag : 30;   /* summary of key */
        uint32_t status : 2;   /* FSM */
        uint32_t offset;   /* offset on flash memory */
};
```

buckets 是一个二级指针，每个 bucket 指向 4 个 slot 大小的缓存，即 4 路 slot，

那么 bucket_num 也就是 slot_num 的 1/4。这里把 slot_num 调小了点，为的是测试 rehash 的发生。

```
#define ASSOC_WAY   (4)   /* 4-way association */
struct hash_table {
      struct hash_slot_cache **buckets;
      struct hash_slot_cache *slots;
      uint32_t slot_num;
      uint32_t bucket_num;
};
int cuckoo_filter_init(size_t size)
{
      /* Allocate hash slots */
      hash_table.slot_num = nvrom_size / SECTOR_SIZE;
      /* Make rehashing happen */
      hash_table.slot_num /= 4;
      hash_table.slots = calloc(hash_table.slot_num, sizeof(struct hash_slot_cache));
      if (hash_table.slots == NULL) {
            return -1;
      }
      /* Allocate hash buckets associated with slots */
      hash_table.bucket_num = hash_table.slot_num / ASSOC_WAY;
      hash_table.buckets = malloc(hash_table.bucket_num * sizeof(struct hash_slot_cache *));
      if (hash_table.buckets == NULL) {
            free(hash_table.slots);
            return -1;
      }
      for (i = 0; i < hash_table.bucket_num; i++) {
            hash_table.buckets[i] = &hash_table.slots[i * ASSOC_WAY];
      }
}
```

下面是哈希函数的设计，这里有两个，前面提到既然 key 是 20 字节的 SHA1 值，我们就可以分别是对 key 的低 32 位和高 32 位进行位运算，只要 bucket_num 满足 2 的幂次方，我们就可以将 key 的一部分同 bucket_num – 1 相与，就可以定位到相应的 bucket 位置上，注意 bucket_num 随着 rehash 而增大，哈希函数简单的好处是求哈希值很快。

```
#define cuckoo_hash_lsb(key, count)  (((size_t *)(key))[0] & (count - 1))
#define cuckoo_hash_msb(key, count)  (((size_t *)(key))[1] & (count - 1))
```

cuckoo filter 最重要的三个操作——查询、插入还有删除。

查询操作对传进来的参数 key 进行两次哈希求值 tag[0]和 tag[1]，并先用 tag[0]定位到 bucket 的位置，从 4 路 slot 中再去对比 tag[1]。只有比中了 tag 后，由于只是 key 的一部分，再去从 flash 中验证完整的 key，并把数据在 flash 中的偏移值 read_addr 输出返回。相应的，如果 bucket[tag[0]]的 4 路 slot 都没有比中，再去 bucket[tag[1]]中比对，如果还比不中，可以肯定这个 key 不存在。这种设计的好处就是减少了不必要的 flash 读操作，每次比对的是内存中的 tag 而不需要完整的 key。

```
static int cuckoo_hash_get(struct hash_table *table, uint8_t *key, uint8_t **read_addr)
{
    int i, j;
    uint8_t *addr;
    uint32_t tag[2], offset;
    struct hash_slot_cache *slot;
    tag[0] = cuckoo_hash_lsb(key, table->bucket_num);
    tag[1] = cuckoo_hash_msb(key, table->bucket_num);
    /* Filter the key and verify if it exists. */
    slot = table-&gt;buckets[tag[0]];
    for (i = 0; i bucket_num) == slot[i].tag) {
        if (slot[i].status == OCCUPIED) {
            offset = slot[i].offset;
            addr = key_verify(key, offset);
            if (addr != NULL) {
                if (read_addr != NULL) {
                    *read_addr = addr;
```

```
                    }
                    break;
                }
            } else if (slot[i].status == DELETED) {
                return DELETED;
            }
        }
}
```

删除操作中 delete 只需将相应 slot 的状态值设置一下即可，也就是说它不会真正到 flash 里面去把数据清除掉。因为 flash 的写操作之前需要擦除整个页面，这种擦除是会折寿的，所以很多 flash 支持随机读，但必须保持顺序写。

```
static void cuckoo_hash_delete(struct hash_table *table, uint8_t *key)
{
    uint32_t i, j, tag[2];
    struct hash_slot_cache *slot;
    tag[0] = cuckoo_hash_lsb(key, table->bucket_num);
    tag[1] = cuckoo_hash_msb(key, table->bucket_num);
    slot = table->buckets[tag[0]];
    for (i = 0; i bucket_num) == slot[i].tag) {
        slot[i].status = DELETED;
        return;
    }
}
```

哈希表层面的插入逻辑其实跟查询差不多，不过多说明。这里主要说明如何判断并处理碰撞，用 old_tag 和 old_offset 保存临时变量，以便一个元素被踢出去之后还能找到备用的位置。这里会有一个判断，每次踢人都会计数，当 alt_cnt 大于 512 时候表示哈希表真的快满了，这时候需要 rehash 了。

```
static int cuckoo_hash_collide(struct hash_table *table, uint32_t *tag, uint32_t *p_offset)
{
    int i, j, k, alt_cnt;
```

```
    uint32_t old_tag[2], offset, old_offset;
    struct hash_slot_cache *slot;
    /* Kick out the old bucket and move it to the alternative bucket. */
    offset = *p_offset;
    slot = table->buckets[tag[0]];
    old_tag[0] = tag[0];
    old_tag[1] = slot[0].tag;
    old_offset = slot[0].offset;
    slot[0].tag = tag[1];
    slot[0].offset = offset;
    i = 0 ^ 1;
    k = 0;
    alt_cnt = 0;
KICK_OUT:
    slot = table->buckets[old_tag[i]];
    for (j = 0; j < ASSOC_WAY; j++) {
        if (offset == INVALID_OFFSET && slot[j].status == DELETED) {
            slot[j].status = OCCUPIED;
            slot[j].tag = old_tag[i ^ 1];
            *p_offset = offset = slot[j].offset;
            break;
        } else if (slot[j].status == AVAILIBLE) {
            slot[j].status = OCCUPIED;
            slot[j].tag = old_tag[i ^ 1];
            slot[j].offset = old_offset;
            break;
        }
    }
    if (j == ASSOC_WAY) {
        if (++alt_cnt > 512) {
```

```
                if (k == ASSOC_WAY - 1) {
                    /* Hash table is almost full and needs to be resized */
                    return 1;
                } else {
                    k++;
                }
            }
            uint32_t tmp_tag = slot[k].tag;
            uint32_t tmp_offset = slot[k].offset;
            slot[k].tag = old_tag[i ^ 1];
            slot[k].offset = old_offset;
            old_tag[i ^ 1] = tmp_tag;
            old_offset = tmp_offset;
            i ^= 1;
            goto KICK_OUT;
        }
        return 0;
}
```

Rehash 为将 buckets 和 slots 重新 realloc，空间扩展一倍，然后再从 flash 中的 key 重新插入到新的哈希表里去。需要注意的是，不能有相同的 key。虽然 cuckoo hashing 不像拉链法那样会退化成 O(n)，但由于每个元素有两个哈希值，而且每次计算的哈希值随着哈希表 rehash 的规模而不同，相同的 key 并不能立即检测到冲突，但当相同的 key 达到一定规模后，由于 rehash 里面有插入操作，一旦在这里触发碰撞，又会触发 rehash，这时就是一个 rehash 不断递归的过程，由于其中老的内存没释放，新的内存不断重新分配，整个程序就会瘫痪。所以每次插入操作前一定要判断一下 key 是否已经存在过，并且对 rehash 里的插入使用碰撞断言防止此类情况发生。

```
static void cuckoo_rehash(struct hash_table *table)
{
    uint8_t *read_addr = nvrom_base_addr;
    uint32_t entries = log_entries;
```

```
    while (entries--) {
        uint8_t key[20];
        uint32_t offset = read_addr - nvrom_base_addr;
        for (i = 0; i &lt; 20; i++) {
            key[i] = flash_read(read_addr);
            read_addr++;
        }
        assert(!cuckoo_hash_put(table, key, &offset));
        if (cuckoo_hash_get(&old_table, key, NULL) == DELETED) {
            cuckoo_hash_delete(table, key);
        }
        read_addr += DAT_LEN;
    }
}
```

测试效果：网上找到一个大文件 unqlite.c 进行测试，这是一个嵌入式数据库源代码，共 59959 行代码。作为需要导入的文件，编译我们的 cuckoo filter，然后执行：./cuckoo_db unqlite.c output.c

发现生成 output.c 正好也是 59959 行代码，同时也可以感受到 cuckoo filter 真的比较快。

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.