华中科技大学

大数据存储系统与管理课程报告

姓　　名：　　胡潇阳

学　　院：　　计算机科学与技术学院

专　　业：　　计算机科学与技术

班　　级：　　计科 2105 班

学　　号：　　U202115484

指导教师：　　华宇

分数	
教师签名	

2024 年　4 月　18 日

目　录

背景

在大数据存储中，我们常用到一个名叫哈希表的数据结构。哈希表也叫散列表，它提供了快速的插入操作和查找操作，使得无论哈希表总共有多少条数据，插入和查找的时间复杂度都是 O(1)，从而实现快速检索数据的目的。然而哈希的本质是从一个较大的空间映射到一个较小的空间，因此在插入足够多数据之后，根据鸽巢原理，一定会存在位置冲突。常见的哈希表会通过链表、开放地址检测等方式来处理冲突，布谷鸟哈希（Cuckoo Hashing）便是用来处理冲突的一种方法。

布谷鸟哈希的名字来源于自然中的布谷鸟，即大杜鹃。该鸟会在其他鸟窝里产蛋，而布谷鸟幼鸟出生后则会将其他蛋踢出，来借此独享母鸟的喂养和生存环境。而布谷鸟哈希的设计关键便在于“踢出”（kicks out）这个动作。

布谷鸟哈希使用多个哈希函数来计算键的多个哈希值，从而使得一个键具有多个候选位置，当一个键的位置与已存在的键冲突时，新的键会选择踢出之前的数据，被踢出的数据会寻找自己的其他候选位置，若其他候选位置也被占用则重复踢出这个过程直到被踢出的数据找到了空位置。

布谷鸟哈希在避免哈希冲突的同时兼具可动态调整大小、支持高效地插入、删除和查找操作的特点，因此受到广泛应用。

数据结构的设计

基本 cuckoo hash 的设计

基本的布谷鸟哈希使用两个哈希函数 h1(x)、h2(x)和两个哈希桶 T1、T2。对每个对象，都有两个存放位置，这两个位置分别在 T1 的 h1(x)和 T2 的 h2(x)上。

当进行插入操作时，假设插入 x，首先检测是否已经存在，若存在，则直接返回，不存在则进行具体插入操作。如果 T1[h1(x)]、T2[h2(x)]中有一个为空，则插入；若两者都为空，则随机选择其中一个插入。若两者都满，则说明发生了哈希冲突，为了将插入对象放入哈希桶中，需要将两个位置中的随机一个对象驱逐出去并将 x 插入到对应的位置。被驱逐的对象则需要去寻找自己的另一个候选位置，重复之前的插入操作，直到找到一个空的位置为止。

当进行查询操作时，对于查询的数据 x，仅需要查询 T1[h1(x)]和 T2[h2(x)]两个位置的数据，将其与 x 对比获得结果。

当进行删除操作时，对于要删除的数据 x，同样仅需要查询 T1[h1(x)]和 T2[h2(x)]两个位置，若两者有数据是 x，则将其置为空。若两者都不是 x，则返回删除错误的信息。

如图 1 所示，这是一个插入示意图。当插入 x 时，x 的两个候选位置都被占用，此时随机选择其中一个位置，即 y 所在位置，随后 y 被驱逐，选择自己的备用位置，即 u 所在位置；u 被驱逐，选择自己的备用位置，即 v 所在位置；v 被驱逐，选择自己的备用位置，即当前 x 所在的位置。以上驱逐过程表示出来即为 x—>y—>z—>u—>v—>x，这样的一条路径便被称为驱逐路径。

对于上述的一条驱逐路径，可以看到并没有结束，被驱逐的 x 同样还会继续选择备用位置 t；t 被驱逐后又会选择自己的备用位置 s；s 被驱逐后会选择自己的备用位置，也就是现在 x 的位置。x 在被驱逐后又会重复上述的过程，可以发现，永远有数据无法插入进去，驱逐路径形成了无限循环，这样会导致死循环。为此，要设置一个 MaxLoop 变量，当驱逐次数超过 MaxLoop 时便停止插入，此时说明可能陷入了死循环，即使没有陷入死循环，也说明哈希表冲突过多了。此时应将所有数据存入缓冲区，使用新的哈希函数进行重新插入。当重新哈希的次数过多，也说明哈希表的容量不足以容纳当前的所有数据，应该进行扩容操作。

由于重新哈希操作和扩容操作都是较为耗时的，因此对死循环的判断及 MaxLoop 值的设置都是影响哈希效率的关键。

对哈希表的扩容，一个比较简单的实现方式是，每次都将内存扩大正好一倍，这样做的优势在于实现简单，能够从较小的空间迅速进行扩容以应对大规模数据。缺点在于在后期扩容所需的内存将出现指数级增长，在某些极端情况会大幅减少空间利用率。

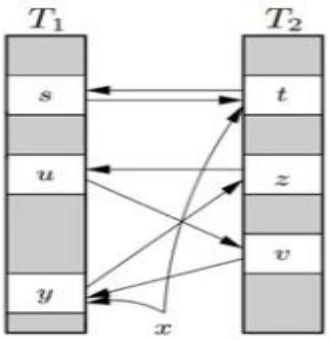

图 1 驱逐路径示意

对 cuckoo hash 的优化

根据上文的叙述可见，对 cuckoo hash 的优化将主要侧重在增加哈希表的空间利用率和减小无限循环的概率上，下面将通过哈希桶多路相连和添加额外缓存结构的方法实现该目的。

哈希桶多路相连

在初始的布谷鸟哈希设计中，对每个桶每个位置只能放一个数据，这样的结果是，布谷鸟哈希的空间利用率在 50%左右。为了提高空间利用率同时减少对其他数据的驱逐行为，我们将每个桶同一个位置拓展为一个数组，可以存放多个数据，即多路相连。

对多路相连而言，进行插入操作时，对于所在的两个位置，只要数组中具有空的位置就可以插入，而非被占用就必须进行驱逐操作，在一定程度上增加了容错率，减小了缓冲的碰撞率，而碰撞有可能会引起更多的驱逐操作，无疑是较为消耗性能的。如果数组中确实没有空闲的数据，则随机选取一个位置进行驱逐操作，其他与基本 cuckoo hash 操作一致。

将哈希桶转化为多路相连后，在整体存储空间不变的情况下，能够有效提升哈希表的空间利用率，经程序测试，当哈希表为四路相连时，最高空间利用率可达到 95%，平均空间利用率也远高于普通布谷鸟哈希，此外得益于低碰撞率，对同样规模大小数据的插入操作的耗时也要低于普通布谷鸟哈希。

但多路相连并非是相联路数越多越好，路数过多也会影响插入和查询的效率。在比较极端的情况下，哈希桶会退化为普通的无序列表，经实验验证将相联路数设为 4 或 8 比较合理。

添加额外缓存结构

当一个数据插入失败时，经典布谷鸟哈希的做法是进行重构，而这无疑是对性能有较大影响的。因此我们可以增加一个临时缓存结构（stash），当数据插入失败（即驱逐次数超过设定的阈值）时，不立刻进行重新哈希，而是存储到这个新增的临时缓存结构中，仅当此缓存结构也填满后再进行重新哈希与扩容操作，这样可以减少因碰撞导致的过多的重新哈希的性能开销。

使用 stash 后，由上，插入操作在插入失败后会先检测 stash 结构是否满，若满，则选择重新哈希，同时，由于 stash 结构满说明已经插入失败较多次了，故无需再对重新哈希进行计数，直接扩容即可。

具有 stash 的查询操作与普通布谷鸟哈希有所不同，因为对数据而言，除了由两个哈希函数计算出的位置外，还可能在 stash 中存储着，因此对每次查询操作，若候选位置都没有所查数据，还应在 stash 中遍历查询。删除操作也一样，候选位置找不到待删除数据时还应在 stash 中遍历来尝试删除。

由于查询操作偶尔需要对 stash 进行遍历，因此虽然查询操作的时间复杂度仍为 O(1)，但终究是比普通布谷鸟哈希消耗要大。因此，作为无序列表，stash 的大小不能太大，不然会因为频繁的查询操作而严重影响性能。事实上，实验可得，当 stash 为存储三到四个的数据元素的大小，就已经能对系统性能产生较大的优化了。在本次的具体实现中，我们也将使用这个优化。

Cuckoo Hash 的具体实现

在本次实验中，笔者使用 Python 来具体实现布谷鸟哈希的结构，首先定义一个具体哈希表的类。如图 2 所示。

```
class HashTable:
    def __init__(self,size,bucketnum=8,hashnum=2,loadfactor=0.7) -> None:
        self.bucketnum=bucketnum
        self.size=size
        self.hasharray0=([[None for i in range(self.bucketnum)]for j in range(self.size//self.bucketnum//2)])
        self.hasharray1=([[None for i in range(self.bucketnum)]for j in range(self.size//self.bucketnum//2)])
        self.hashFuncNum=[0,0]
        self.hashFuncNum[0]=random.randint(0,10000)
        self.hashFuncNum[1]=random.randint(0,10000)
        while self.hashFuncNum[1]==self.hashFuncNum[0]:
            self.hashFuncNum[1]=random.randint(0,10000)
        self.objnums=0
        self.maxloop=8
        self.rehashcnt=0
        self.stash=[]
        self.stashMaxSize=4
        self.hashnum=hashnum
        self.loadfactor=loadfactor
```

图 2 哈希表类定义

其中，size 表示哈希表整体的大小，即可容纳数据的多少；hasharray0 和 hasharray1 则是两个哈希桶；bucketnum 表示每个哈希桶的槽位数。objnums 表示现在已经容纳的数据多少；maxloop 为驱逐路径最大长度的阈值；rehashcnt 表示当前已进行过的重新哈希的次数，当达到一定阈值时会进行扩容；stash 表示新增的缓冲部分结构，stashMaxSize 表示这个缓冲部分结构的最大大小，当缓冲部分元素数超过这个数值时便进行扩容和重新哈希。loadfactor 表示哈希表的负载参数，当哈希表需要扩容时，便按照这个参数进行扩容。

本次实现主要使用的哈希函数来自 mmh3 库，该库支持通过种子确定哈希函数，因此本次哈希的实现主要为随机生成两个不同的数来代表两个不同的哈希函数。之所以要不同的数是为了防止两个哈希函数计算出的位置一致出现自己驱逐自己的死循环现象。另外，由于该哈希函数是对字符串进行哈希，因此在使用该哈希表时，仅需实现对数据对象的字符串化即可。

插入函数的实现如图 3 所示。

```
def insert(self,data,rehashing=False):
    if self.search(data=data):
        return True
    index0=mmh3.hash(str(data),self.hashFuncNum[0])%(self.size//self.bucketnum//2)
    index1=mmh3.hash(str(data),self.hashFuncNum[1])%(self.size//self.bucketnum//2)
    if random.randint(0,1)==0:
        for i in range(self.bucketnum):
            if self.hasharray0[index0][i]==None:
                self.hasharray0[index0][i]=data
                self.objnums+=1
                return True
        for i in range(self.bucketnum):
            if self.hasharray1[index1][i]==None:
                self.hasharray1[index1][i]=data
                self.objnums+=1
                return True
    else:...
    bnum=random.randint(0,self.bucketnum-1)
    if random.randint(0,1)==0:
        newdata=self.hasharray0[index0][bnum]
        self.hasharray0[index0][bnum]=data
        self.objnums+=1
        return self.drive(newdata=newdata,array=0,loop=0,rehashing=rehashing)
    else:
        newdata=self.hasharray1[index1][bnum]
        self.hasharray1[index1][bnum]=data
        self.objnums+=1
        return self.drive(newdata=newdata,array=1,loop=0,rehashing=rehashing)
```

图 3 插入函数

在具体函数中，为了增添随机性，减少碰撞概率，我们在检测空位和进行驱逐放置时都进行随机，以使数据分布更加均衡。其余与结构设计部分所述一致。

查询函数具体实现如图 4 所示。

```
def search(self,data):
    index0=mmh3.hash(str(data),seed=self.hashFuncNum[0])%(self.size//self.bucketnum//2)
    index1=mmh3.hash(str(data),seed=self.hashFuncNum[1])%(self.size//self.bucketnum//2)
    if data in self.hasharray0[index0]:
        return True
    if data in self.hasharray1[index1]:
        return True
    if data in self.stash:
        return True
    return False
```

图 4 查询函数

我们首先根据两个哈希函数计算出位置，随后逐个判断是否存在，若都不存在，则去 stash 结构中判断是否存在，最后返回结果。

当发生冲突时，驱逐函数如图 5 所示。

```python
def drive(self,newdata,array,loop,rehashing):
    if loop>self.maxloop:
        if len(self.stash)<self.stashMaxSize:
            self.stash.append(newdata)
            return True
        if not rehashing:
            self.rehash(newdata=newdata,havebuf=None)
        return False
    if array==0:
        index=mmh3.hash(str(newdata),self.hashFuncNum[1])%(self.size//self.bucketnum//2)
        for i in range(self.bucketnum):
            if self.hasharray1[index][i]==None:
                self.hasharray1[index][i]=newdata
                return True
        else:
            bnum=random.randint(0,self.bucketnum-1)
            data=self.hasharray1[index][bnum]
            self.hasharray1[index][bnum]=newdata
            return self.drive(newdata=data,array=1-array,loop=loop+1,rehashing=rehashing)
    else:
        index=mmh3.hash(str(newdata),self.hashFuncNum[0])%(self.size//self.bucketnum//2)
        for i in range(self.bucketnum):
            if self.hasharray0[index][i]==None:
                self.hasharray0[index][i]=newdata
                return True
        else:
            bnum=random.randint(0,self.bucketnum-1)
            data=self.hasharray0[index][bnum]
            self.hasharray0[index][bnum]=newdata
            return self.drive(newdata=data,array=1-array,loop=loop+1,rehashing=rehashing)
```

图 5 驱逐函数

可见，驱逐函数含有几个参数，newdata 表示当前待插入的数据，array 表示该数据之前所在的哈希桶编号，loop 表示当前驱逐行为的已进行次数，rehashing 表示是否正在进行重新哈希。

首先判断 loop 参数是否已经超过设定阈值，若超过，则判断 stash 是否已满，未满则直接插入返回，满则返回失败。当前若未处于重新哈希过程，则将所有数据转存到缓冲区，重新生成哈希函数的种子进行重新哈希；若处于重新哈希过程，说明已有缓冲区存储了所有数据，仅需返回失败信息即可。

性能分析

对于上述的哈希表结构，笔者编写了一个测试程序，该编写程序会按照给定的数据规模生成一定范围内的 int 数据，我们将这些 int 数据插入到哈希表中，根据插入总体的时间计算插入延迟。之后根据当前哈希表的尺寸来计算空间利用率。最后，我们将之前插入过的 int 数据逐个查询，看是否能够查询成功，同时根据总体时间查询延迟。

笔者设置数据规模为 1000000 个，对普通布谷鸟哈希表、带有多路相连的哈希表、带有额外缓冲的哈希表、同时带有多路相联和额外缓冲的哈希表分别测试如图 6 图 7 图 8 图 9 所示：

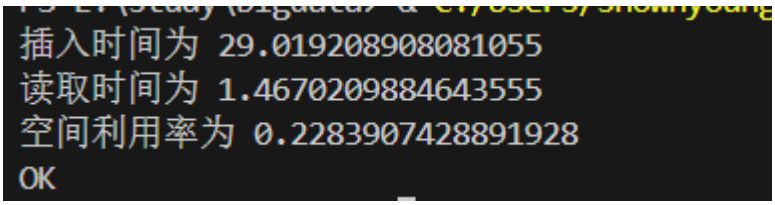

图 6 普通哈希性能测试

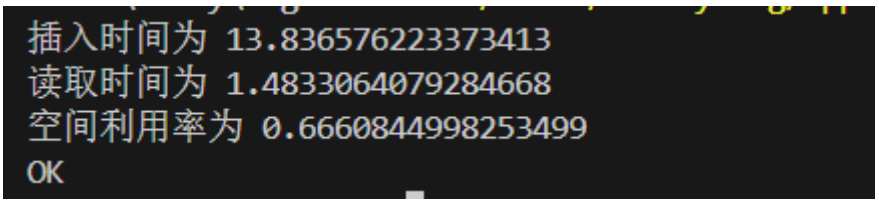

图 7 多路相联哈希性能测试

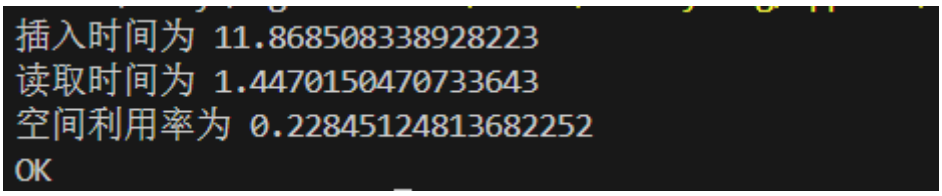

图 8 额外缓冲哈希性能测试

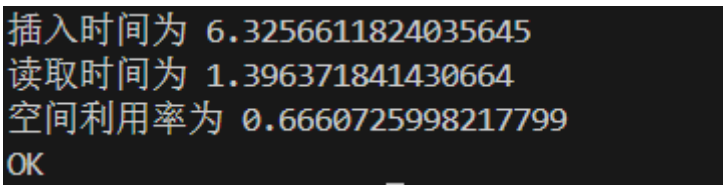

图 9 多路相联和额外缓冲哈希性能测试

结果分析：可以看到，对普通的布谷鸟哈希，插入是较为耗时的，同时空间利用率也并不高；而添加上多路相连后，在读取时间差别不大的情况下，对插入时间和空间利用率都有着不小的优化；添加有额外缓冲的哈希表对插入时间有优化但空间利用率仍然较低；同时使用多路相连和额外缓冲的哈希表不仅插入时间

有较大优化，空间利用率也得到了较大提升。由此可见，上文所提到的两种优化方法都能对无限循环和有效存储有较大优化提升。

需要说明，在本次简易实现中，数据表中存储的是原数据，故不会出现误判，同时也没有测定错误率。而在有些实现中哈希表中存储的是原数据的指纹，可能会出现一定程度的假阳率，程度随使用的哈希算法浮动，但基本不影响使用。

总结

在本次实验中，笔者实现了基本的布谷鸟哈希结构并对其进行了基本的优化，在解决无限循环问题和提升有效存储方面进行了尝试并取得了一定的成果。

通过本次的实践，我对于布谷鸟哈希算法的原理和实现细节有了更加深刻的理解，同时，这也促使我不断在网上寻找相关的资料并进行自己的实践，这培养了我对文献资料的阅读能力和归纳能力，对将来的工作和生活也大有裨益。