2019 级

《物联网数据存储与管理》课程

报 告

姓　名　李启鑫

学　号　U201915165

班　号　计算机 1908 班

日　期　2022.04.12

目　　录

一、理论分析

哈希表的原理是通过 hash 函数将元素映射到一个空间中，因此如果插入数据过多，根据鸽巢原理，一定会存在位置冲突。

常见的哈希表（Hash Table 或者字典，dictionary）会通过链表、开放地址探测等方式来处理冲突。单桶多函数的布谷鸟哈希，便是开放地址法处理冲突的一种哈希表，只不过有冲突后，不是通过线性寻找新的位置，而是通过额外哈希函数来寻找。

1.1 普通 Hash Table

为了保证插入和查找的平均复杂度为 O(1)，hash table 底层一般都是使用数组来实现。对于给定的 key，一般先进行 hash 操作，然后相对哈希表的长度取模，将 key 映射到指定的地方。

冲突处理

冲突，也叫做碰撞，意思是两个或者多个 key 映射到了哈希表的同一个位置。冲突处理一般有两种方法：开放寻址（open addressing）和链表法（separate chaining）。

开放寻址在解决当前冲突的情况下同时可能会导致新的冲突，而链表法不会有这种问题。同时链表相比于开放寻址局部性较差，在程序运行过程中可能引起操作系统的缺页中断，从而导致系统颠簸。

1. 开放寻址

开放寻址的意思是当插入 key，value 发生冲突时，从当前位置向后按某种策略遍历哈希表。当发现可用的空间的时候，则插入元素。开放地址有一次探测、二次探测和双重哈希。一次探测是指我们的遍历策略是一个线性函数，比如依次遍历冲突位置之后的第 1，2，3…N 位置。如果直接遍历 1，4（=2^2），9 (=3^2)，这就是二次探测的一个例子。双重哈希就是遍历策略间隔由另一个哈希函数来确定。

开放寻址法会导致之后的 key 被占用。如图 1.1 所示，key “John Smith” 和 “Sandra Dee” 在 index = 152 位置出现冲突，使用开放地址的方法将 “Sandra Dee” 存放在 index = 153 的位置。之后 key “Ted Baker” 的映射位置为 index = 153，又出现冲突，则将其存放在 index = 154 的位置。

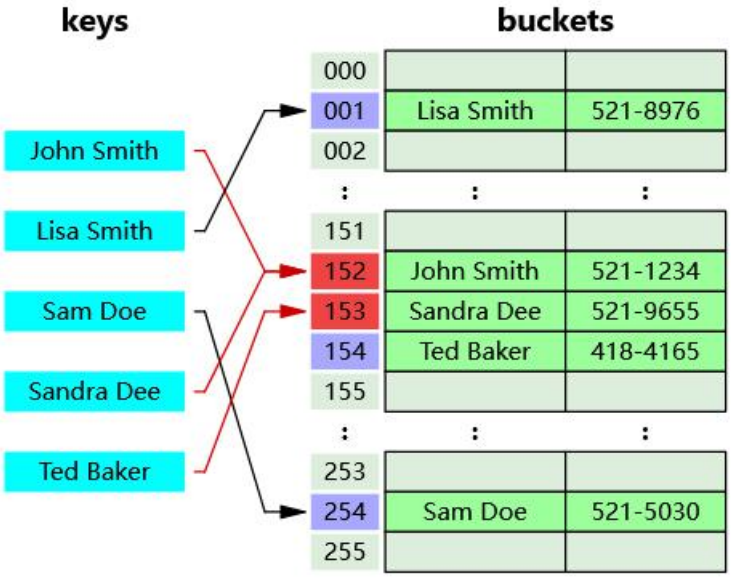

图 1.1 开放寻址法

2. 链表法

链表法的意思是哈希表中的每个元素都是一个类似链表或者其他数据结构的 head。当出现冲突时，在链表后面添加元素。这也就意味着，如果某一个位置冲突过多的话，插入的时间复杂段将退化为 O(N)。

1.2 Cuckoo Hash

Cuckoo Hash 的就是采自“鸠占鹊巢”的意思。Cuckoo 是布谷鸟的意思。“鸠占鹊巢”里面的“鸠”的就是类似布谷鸟这种鸟类。和普通的哈希表不同，在 Cuckoo Hash 表中，每个元素在 bucket list 中有两个位置可以存放。如果两个位置都被其他元素占了，则随机选择一个位置将其踢走。被踢走的元素被移动到它的备用位置，如果也被其他元素占了则也将其踢走，如此反复。下图是 Cuckoo Hash 的示例。

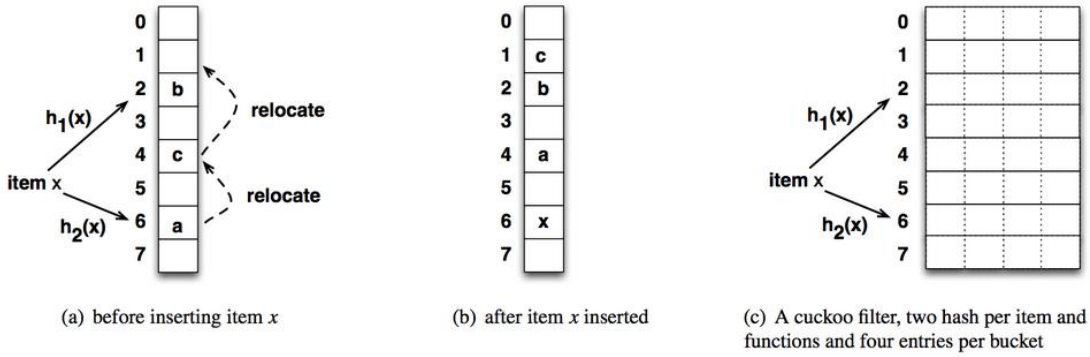

图 1.2 cuckoo hash

(a) 表示插入元素 x 之前已经存在的元素 a，b，c。通过计算 x 的两个可用位置分别被 a，b 占用了，随机选择一个位置，也就是元素 a 存放的位置。然后元素 a 被迁移到它的备用位置，也就是元素 c 现在的位置。类似的 c 再做迁移。

(c) 表示 Cuckoo Hash 的一种实现，每个 bucket 有四个 slot。对于 (a) 的情形，则直接把 x 插入到元素 a 或 b 所在的 bucket 的空的 slot 里面即可。

其基本思想为：

1. 使用两个哈希函数 h1 (x) 、 h2 (x) 和两个哈希桶 T1、T2 。

2. 插入元素 x：

如果 T1 [h1 (x)] 、T2 [h2 (x)] 有一个为空，则插入；两者都空，随便选一个插入。

如果 T1 [h1 (x)] 、T2 [h2 (x)] 都满，则随便选择其中一个（设为 y ），将其踢出，插入 x。

重复上述过程，插入元素 y。

如果插入时，踢出次数过多，则说明哈希桶满了。则进行扩容、ReHash 后，再次插入。

3. 查询元素 x：

读取 T1 [h1 (x)] 、T2 [h2 (x)] 和 x 比对即可

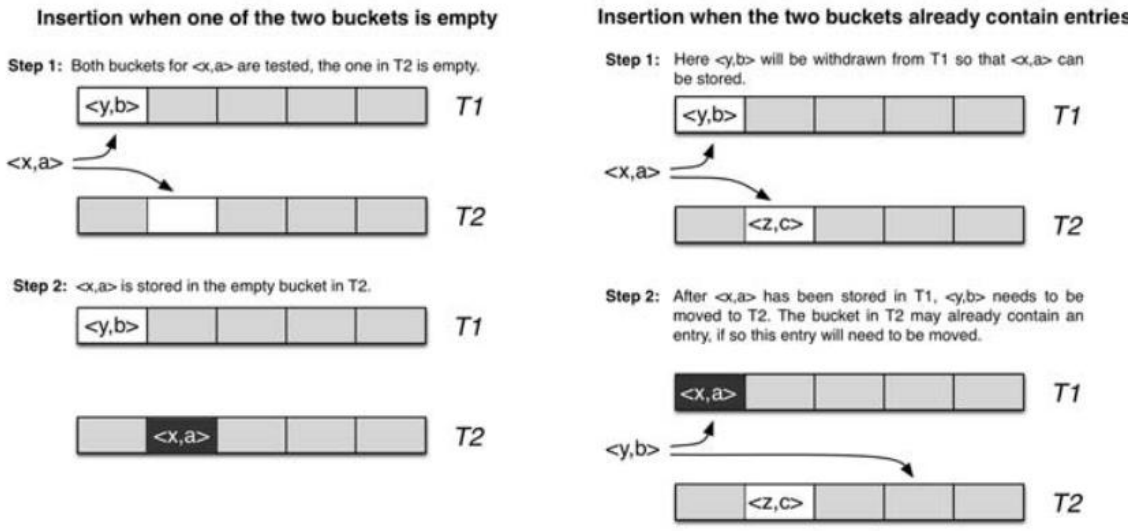

图 1.3 cuckoo hash 操作

1.3 Cuckoo Hash 问题

Cuckoo Hash 主要有以下两个问题：

1. 写入性能差.数据项写入 Cuckoo 哈希表时,如果发生哈希冲突,则将候选桶中任意一个数据项迁移到该数据项的其他候选桶中,并且可能迁移另一个数据项,直到找到一个空槽或者达到最大搜索阈值。本质上,深度优先搜索的随机替换策略导致 Cuckoo 哈希的 Cuckoo 路径很长,频繁的“踢出”操作导致其在多个桶之间进行密集的数据迁移,使得哈希表写入性能低下,同时细粒度锁的大量使用存在死锁和活锁的风险.基于随机替换策略的 Cuckoo 哈希表的桶之间的负载不均衡,导致在高负载率下易出现高延迟插入和无限循环.

2. 正向查询性能差.Cuckoo 哈希表通常使用 D(D＞1)个散列函数,查询时最少只需要访问 1 个桶就能得到查询结果,最多时也只需要访问 D 个桶.在读负载远大于写负载的情况下,哈希表的吞吐量取决于读负载的吞吐量.

1.4 Cuckoo Hash 优化

1. 使用多路哈希桶。
2. 使用多个 Hash 函数
3. 使用广度优先搜索缩短深度优先搜索路径长度

大多数 Cuckoo 哈希表使用随机替换策略搜索空槽,如果数据项 x 的候选桶已满,则随机选择桶中的一个数据项 y 并将其迁移到候选桶中,这可能导致循环迁移数据项,直到找到一个空槽或者达到最大搜索阈值.这个过程中所有相关的桶都是 Cuckoo 路径的一部分.随着哈希表负载率的增加,Cuckoo 路径的长度也会增加,大量数据项的迁移导致哈希表的性能低下.Cuckoo 哈希表可看成一个无向图,每个桶为一个顶点,每个数据项为一条边,用以连接两个候选桶(顶点).用广度优先方法搜索空槽,桶中的每个数据项都被视为可能的路径,并以相同的方式将路径拓 展到其他候选桶.广度优先搜索能够以对数的形式缩短深度优先搜索 Cuckoo

本次课题代码将采用 1、3 对 cuckoo Hash 进行优化。

二、代码实现

cuckoo hash 采用 Go 语言实现，下面将对代码层面进行说明。

2.1 数据结构说明

```
type hashSlotCache struct {
  tag  int
  data string
}
type block struct {
  bucket [4]hashSlotCache
  num    int8
}

type hashTable struct {
  blockCache []block
  blockNum   int
  bucketNum  int
}
```

图 2.1 数据结构定义

如上图所示，hashSlotCache 存储一个键值对，<tag,data>，block 表示一块中有四个桶，num 记录多少个桶已经存储了数据，hashTable 中 blockCache 作为一个切片存储块，blockNum 记录存在着多少个块，buketNum 记录已经存储的键值对个数。

2.2 get 方法

```
func (t hashTable) get(key int) string {
  h1, h2 := t.hash(key)
  value1 := h1.get(key)
  if value1 != nil {
    return value1.data
  }
  value2 := h2.get(key)
  if value2 != nil {
    return value2.data
  }
  return ""
}
```

图 2.2 get 方法

cuckoo hash get 方法通过 key 值得到可能存在的两个块，block 上面存在一个 get 方法可以获取到 tag 等于 key 的桶，如果没有，则返回控制住 nil，通过两次获取，最终得到 value。

2.3 set 方法

```go
func (t hashTable) set(key int, value string) int {
  h1, h2 := t.hash(key)
  result := t.search(h1, h2)
  if result == nil {
    t.refresh()
    return t.set(key, value)
  }
  result.bucket[result.num] = hashSlotCache{tag: key, data: value}
  result.num++
  t.bucketNum++
  return 1
}
```

图 2.3 cuckoo hash set 方法

cuckoo hash set 方法需要考虑到如果两个 block 都已经满了，需要将某个 key 踢出去，如果是随机踢出去，那么可能会连续踢出多个值，导致性能下降。这里使用了 search 方法查找踢出最短路径。如果 search 方法返回的是空指针，表明

```go
func (t hashTable) search(b1 *block, b2 *block) *block {
  if b1.num < ASSOC_WAY {
    return b1
  }
  if b2.num < ASSOC_WAY {
    return b2
  }
  queue := make([]node, 20)
  queue = append(queue, node{cur: b1, parent: nil}, node{cur: b2, parent: nil})

  for i := 0; i < 5 && len(queue) > 0; i++ {
    blocks := queue[:]
    queue = queue[len(blocks):]
    for _, curNode := range blocks {
      if curNode.cur.num != ASSOC_WAY {
        return curNode.move().cur
      }
      for i, block := range curNode.cur.bucket {
        h1, h2 := t.hash(block.tag)
        queue = append(queue, node{cur: h1, parent: &curNode, idx: int8(i)}, node{cur: h2, parent: &curNode, idx: int8(i)})
      }
    }
  }
  return nil
}
```

图 2.4 cuckoo search 方法

search 方法需要减少 cuckoo hash 踢出数据的次数，此处我采用广度优先搜索，使用一个队列搜索每层的 hashSlotCache，如果发现有空位则调用 move 方法，将父元素往下层移动，否则搜索下一层。这样保证了踢出时的路径是最少的，同时，为了防止循环现象，最多只允许搜索 5 层。

三、测试

本次实验性能分析将与 Go 语言本身的 map 数据结构进行比较。实验中使用如图 3.1 测试函数，该测试函数返回运行耗时。

```
func test(slots []hashSlotCache, con consumer) int64 {
  start := time.Now().UnixNano()
  for _, slot := range slots {
    con(slot)
  }
  end := time.Now().UnixNano()
  return end - start
}
```

图 3.1 测试框架

3.1 set 测试

```
t := cuckooInit(80000)
slots := make([]hashSlotCache, 100000)
for i := 0; i < cap(slots); i++ {
  slots[i].data = fmt.Sprintf("%v", rand.Int())
  slots[i].tag = rand.Int()
}
duration1 := setTest(slots, func(hsc hashSlotCache) {
  t.set(hsc.tag, hsc.data)
})
s := map[int]string{}
duration2 := setTest(slots, func(hsc hashSlotCache) {
  s[hsc.tag] = hsc.data
})

fmt.Printf("cuckoo hash spend:%vns\n", duration1)
fmt.Printf("map spend:%vns\n", duration2)
```

图 3.2 set 测试运行程序

图 3.2 为测试 set 方法代码，该测试将插入 100000 条数据到 cuckoo Hash 和 map 中，使用的 time 进行计时，获取到以纳秒为单位的插入耗时。程序输出结果见图 3.3。

```
cuckoo hash set spend:8661100ns
map set spend:12502800ns
```

图 3.3 set 测试输出结果

可以看到，通过 cuckoo hash 实现的数据结构插入 100000 条数据需要 8.66 毫秒，map 需要 12.5 毫秒，性能提升近 50%。

3.2 get 测试

get 测试代码见图 3.4，通过图 3.1 的测试函数获取到 map 和 cuckoo Hash 的获取 100000 条数据的时间。

```
duration1 = test(slots, func(hsc hashSlotCache) {
  t.get(hsc.tag)
})
res := ""
duration2 = test(slots, func(hsc hashSlotCache) {
  res = s[hsc.tag]
})
res = ""
fmt.Printf("cuckoo hash get spend:%vns\n", duration1)
fmt.Printf("map get spend:%vns\n", duration2)
println(res)
```

图 3.4 get 测试代码

get 测试结果见图 3.5，其中 cuckoo hash 耗时 12 毫秒，map 耗时 3 毫秒，与 map 相比，cuckoo hash 获取性能严重下降。其中比较重要的原因是 cuckoo hash 使用了四路相联块，两个 hash 函数，每次获取都需要从两个块中共 8 条记录中比较，因此性能下降。

```
cuckoo hash get spend:12605000ns
map get spend:2995100ns
```

图 3.5 get 测试输出

四、总结

本次报告探讨了 cuckoo Hash 的原理以及解决 cuckoo Hash 两个问题的方案，最后通过 Go 语言实现了一个简单的 cuckoo Hash，并与 Go 语言原生的 map 进行比较。

这次对课题的研究加深了我对于大数据存储实际方案的理解，增强了我代码实现的能力，让我了解到在现代数据规模翻倍增长时面临的困难，拓宽了我知识的广度。

在实验时也发现了一个难题，cuckoo Hash 通过增加桶的方式优化了存储数据的方式，增加了效率，但与此同时，增加了查询的耗时。这种情况在计算机领域可谓屡见不鲜，时间与空间的权衡、网络与计算的权衡。

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.