2021 级

《大数据存储与管理》课程

实 验 报 告

姓　名　艾筠舜

学　号　U202115388

班　号　计算机 2103 班

日　期　2024.04.21

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-driven Way 是一种基于 Cuckoo Hashing 算法的散列表技术。用于解决表中散列函数值的散列冲突，在最坏情况下具有恒定的查找时间。这个名字来源于某些布谷鸟的行为，布谷鸟雏鸟在孵化时会将其他蛋或雏鸟推出巢外，这是一种被称为“巢寄生”行为的变体。类似地，将新密钥插入到布谷鸟哈希表中可能会将旧密钥推送到表中的不同位置。

Cuckoo hash 是开放寻址的一种形式，其中哈希表的每个非空单元格都包含一个键或键值对。哈希函数用于确定每个键的位置，并且可以通过检查表的该单元格来找到它在表中的存在（或与其关联的值）。然而，开放寻址会遇到冲突，当多个键映射到同一单元时就会发生这种情况，如图 1 所示。Cuckoo 哈希的基本思想是通过使用两个哈希函数而不是仅一个哈希函数来解决冲突。这为每个键在哈希表中提供了两个可能的位置。在该算法的一种常用变体中，哈希表被分成两个大小相等的较小表，每个哈希函数提供这两个表之一的索引。两个哈希函数也可以为单个表提供索引。[1]

$$h(k) = k \bmod 11$$
$$h'(k) = \left\lfloor \frac{k}{11} \right\rfloor \bmod 11$$

1. table for h(k)

		Key inserted									
k		20	50	53	75	100	67	105	3	36	39
h(k)		9	6	9	9	1	1	6	3	3	6
Hash table entries	0										
	1					100	67	67	67	67	100
	2										
	3								3	36	36
	4										
	5										
	6		50	50	50	50	50	105	105	105	39
	7										
	8										
	9	20	20	20	75	75	75	53	53	53	75
	10										

2. table for h′(k)

		Key inserted									
k		20	50	53	75	100	67	105	3	36	39
h′(k)		1	4	4	6	9	6	9	0	3	3
Hash table entries	0									3	3
	1				20	20	20	20	20	20	20
	2										
	3										
	4			53	53	53	53	50	50	50	53
	5										
	6							75	75	75	67
	7										
	8										
	9						100	100	100	100	105
	10										

图 1 Cuckoo hash 插入

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，如果备用位置上 还有人，再把它踢走，如此往复如图 2 所示。但是，哈希表中的元素在各个哈希表中反复移动可能会发生无限循环。例如，我在 test.py 中尝试实现的 cuckoo 操作中的无限循环。

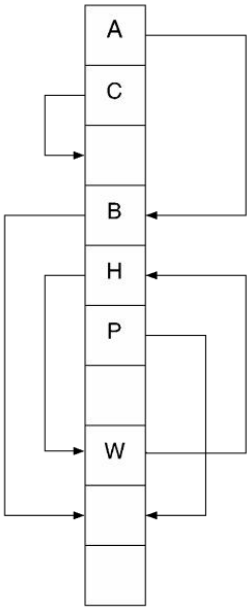

图 2 Cuckoo hash 示意图

在本次实验中，我尝试去解决这个无限循环带来的问题。

三、 数据结构设计与实现

Cuckoo hash 中最有可能触发无限循环的部分就是 insert 的部分，因此这是着重关注的部分。

为了实现 cuckoo hash 的 hash table，我需要准备一些 classes 与相关函数。

1. 首先是与 cuckoo 中 hash 函数相关的数据结构与相关方法实现的代码

```
# setup a list of random 64-bit values to be used by BitHash
__bits = [0] * (64*1024)
__rnd = random.Random()

# seed the generator to produce repeatable results
__rnd.seed("BitHash random numbers")

# fill the list
for i in range(64*1024):
    __bits[i] = __rnd.getrandbits(64)


def BitHash(s, h=0):
    """ BitHash(s, h=0) -> int """
    s = str(s)
    for c in s:
        h = (((h << 1) | (h >> 63)) ^ __bits[ord(c)])
        h &= 0xffffffffffffffff
    return h

```

```
    # this function causes subsequent calls to BitHash to be
    # based on a new set of random numbers. This is useful
    # in the event that client code needs a new hash function,
    # for example, for Cuckoo Hashing.


    def ResetBitHash():
        """ ResetBitHash() -> None """
        global __bits
        for i in range(64*1024):
            __bits[i] = __rnd.getrandbits(64)
```

2. 为了实现 cuckoo hash，需要一个类代表哈希表中的一个 node

```
    class Node(object):
        """ Node class for hash table """

        def __init__(self, k, d):
            self.key = k
            self.data = d

        def __str__(self):
            return "(" + str(self.key) + ", " + str(self.data) + ")"
```

3. 接下来是最重要的实现 cuckoo hash 所需要的 hash table 类，这个大类里面有一些重要的方法需要实现，由于代码很多，因此我只展示一下 insert 函数，其余的函数略，详情可以看 cuckoohash.py 文件

   (1) def insert(self, k, d):

   这个函数是 cuckoo hash 中最重要的插入部分，这个函数中有对发生冲突时的简单处理:

   - 使用哈希函数计算键 k 在两个哈希表中的位置，分别为 position1 和 position2。
   - 尝试将节点 n 插入到第一个哈希表的 position1 位置。如果这个位置为空，那么插入成功，增加记录数，返回 True。
   - 如果 position1 位置已经被其他节点占用，那么将这个节点移出，并将节点 n 插入到这个位置。
   - 随机选择一个哈希表，并计算节点 n 在这个哈希表中的位置。如果选择的是第一个哈希表，那么位置为 position1，如果选择的是第二个哈希表，那么位置为 position2。

```
    def insert(self, k, d):
            if self.find(k) is not None:
                return False

            n = Node(k, d)

            if self.__numRecords >= (self.__size // 2):
                self.__growHash()
```

```
9.
10.             position1, position2 = self.hashFunc(n.key)
11.
12.             pos = position1
13.             table = self.__hashArray1
14.
15.             while True:
16.                 self.loop_count += 1
17.                 print(f"loop count: {self.loop_count}")
18.                 if table[pos] is None:
19.                     table[pos] = n
20.                     self.__numRecords += 1
21.                     return True
22.
23.                 n, table[pos] = table[pos], n
24.
25.                 if random.choice([True, False]):
26.                     position1, position2 = self.hashFunc(n.key)
27.                     pos = position2
28.                     table = self.__hashArray2
29.                     # print(f"键值 {n.key} 被插入到第二个哈希表的位置 {pos}")
30.                 else:
31.                     position1, position2 = self.hashFunc(n.key)
32.                     pos = position1
33.                     table = self.__hashArray1
34.                     # print(f"键值 {n.key} 被插入到第二个哈希表的位置 {pos}")
```

但是，在上面的这个 insert 方法中，我并没有检测无限循环，也没有做出相应的处理，只做到了最基本 cuckoo hash 的功能，可以向 hashArray1 与 hashArray2 中插入元素，以及踢出元素。很明显，这样子的 insert 会导致无限循环。

(2) def rehash(self, size)

(3) def hashFunc(self, s)

(4) def __growHash(self)

(5) def find(self, k)

(6) def delete(self, k)

4. test_hashtab()与 test_hashtab_without_check()

为了方便测试处理无限循环与优化的效果，需要额外的测试函数:

```
1.     def test_hashtab():
2.         """ Test the Cuckoo Hash table """
3.         size = SIZE
4.         missing = 0
5.         found = 0
6.
7.         # create a hash table with an initially small number of bukets
8.         c = HashTab(HASHTAB_SIZE)
```

```
9.
10.         # Now insert size key/data pairs, where the key is a string consisting
11.         # of the concatenation of "foobarbaz" and i, and the data is i
12.         inserted = 0
13.         for i in range(size):
14.             if c.insert(str(i)+"foobarbaz", i):
15.                 inserted += 1
16.         print("There were", inserted, "nodes successfully inserted")
17.
18.         # Make sure that all key data pairs that we inserted can be found in the
19.         # hash table. This ensures that resizing the number of buckets didn't
20.         # cause some key/data pairs to be lost.
21.         for i in range(size):
22.             ans = c.find(str(i)+"foobarbaz")
23.             if ans is None or ans != i:
24.                 print(i, "Couldn't find key", str(i)+"foobarbaz")
25.                 missing += 1
26.
27.         print("There were", missing, "records missing from Cuckoo")
28.
29.         # Makes sure that all key data pairs were successfully deleted
30.         for i in range(size):
31.             c.delete(str(i)+"foobarbaz")
32.
33.         for i in range(size):
34.             ans = c.find(str(i)+"foobarbaz")
35.             if ans is not None or ans == i:
36.                 print(i, "Couldn't delete key", str(i)+"foobarbaz")
37.                 found += 1
38.         print("There were", found, "records not deleted from Cuckoo")
39.         print("loop count: ", c.loop_count)
40.         return c
```

为了观察结果明显，我把循环的次数输出了出来。

四、操作流程分析

1．创建一个 Cuckoo 哈希表

2．插入元素

Cuckoo hash 的操作流程正如上一节分析的一样，最复杂最重要的就是 insert 函数所执行的插入操作。当我们向 hash table 中插入一个新的元素的时候发生了冲突，在这个函数中就需要处理对应的冲突，并且要一定程度上避免死循环。例如，如果我在图 1 的基础上，继续插入元素 6，就会发生如图 3 所示的问题:

table 1	table 2
6 replaces 50 in cell 6	50 replaces 53 in cell 4
53 replaces 75 in cell 9	75 replaces 67 in cell 6
67 replaces 100 in cell 1	100 replaces 105 in cell 9
105 replaces 6 in cell 6	6 replaces 3 in cell 0
3 replaces 36 in cell 3	36 replaces 39 in cell 3
39 replaces 105 in cell 6	105 replaces 100 in cell 9
100 replaces 67 in cell 1	67 replaces 75 in cell 6
75 replaces 53 in cell 9	53 replaces 50 in cell 4
50 replaces 39 in cell 6	39 replaces 36 in cell 3
36 replaces 3 in cell 3	3 replaces 6 in cell 0
6 replaces 50 in cell 6	50 replaces 53 in cell 4

图 3 发生循环

因此我在上一节的 insert 函数的基础上，做出了部分更改，得到了下面这个 insert 函数。

```
    # insert and return true, return False if the key/data is already there,
      # grow the table if necessary, and rehash if necessary
      def insert(self, k, d):
          """ Insert a key/data pair into the hash table """
          if self.find(k) is not None:
              return False   # if already there, return False (no duplicates)


          # create a new node with key/data
          n = Node(k, d)

          # increase size of table if necessary
          if self.__numRecords >= (self.__size // 2):
              self.__growHash()

          position1, position2 = self.hashFunc(n.key)  # hash

          # start the loop checking the 1st position in table 1
          pos = position1
          table = self.__hashArray1

          # dynamically adjust loop times based on load factor
          load_factor = self.__numRecords / self.__size
          max_loop = max(5, int(load_factor * 10))
          for i in range(max_loop):
              self.loop_count += 1
```

```
                if table[pos] is None:                # if the position in the cu
    rrent table is empty
                    # insert the node there and return True
                    table[pos] = n
                    self.__numRecords += 1
                    return True

                # else, evict item in pos and insert the item
                n, table[pos] = table[pos], n
                # then deal with the displaced node.

                # randomly choose which position to check next
                if random.choice([True, False]):
                    position1, position2 = self.hashFunc(
                        n.key)  # hash the displaced node,
                    pos = position2                               # and check its
    2nd position
                    # in the 2nd table (next time through loop)
                    table = self.__hashArray2
                    # print(f"键值 {n.key} 被插入到第二个哈希表的位置 {pos}")
                else:
                    # otherwise, hash the displaced node,
                    position1, position2 = self.hashFunc(n.key)
                    # and check the 1st table position.
                    pos = position1
                    table = self.__hashArray1
                    # print(f"键值 {n.key} 被插入到第二个哈希表的位置 {pos}")

            self.__growHash()               # grow and rehash if we make it here

            self.rehash(self.__size)
            self.insert(n.key, n.data)      # deal with evicted item

            return True
```

在这个函数中，我会根据散列表的负载因子动态调整循环次数，并且随机选择哈希函数来帮助打破可能的循环模式。这样可以有效的减少 insert 所花费的循环次数，从而提高插入效率，同时也可以很大程度上避免无限循环的发生。

3．查找操作

这个操作就直接通过哈希函数获得索引，然后在哈希表中访问索引，对比结果即可。

4．删除操作

与上面的查找操作类似，只不过时把对应的元素进行删除，同时修改表中总共的元素数量即可

5．growHash

如果在插入元素的时候，发现哈希表满了，就需要扩大哈希表，进行 growhash 操作。

6．rehash

当哈希函数或哈希表大小改变时，需要将所有元素重新插入到新的哈希表中。这通常发生在哈希函数改变或哈希表大小改变时。重新哈希可以帮助分散元素，减少冲突，但是重新哈希的过程需要消耗一定的时间和计算资源。

五、理论分析

1. 为了减少 Cuckoo-driven Way 中无限循环的概率，可以采取以下措施:
   - 设定最大重试次数：在进行 Cuckoo 操作时，可以设置一个最大重试次数，如果超过该次数则认为当前操作失败。这可以防止出现死循环的情况。但这倾向于一个启发式的设计，往往需要经验去判断，然后才可以在大部分情况下得到不错的结果
   - 增加哈希表容量：扩大哈希表，从而降低 Cuckoo 操作失败的概率。当我我发现负载因子超过 0.5 的时候，我会调用 growHash 来扩大我的哈希表，防止出现无限循环。
   - 使用随机化:在 Cuckoo-driven Way 中，可以使用随机化来选择哈希函数，从而降低哈希冲突的概率，减少 Cuckoo 操作失败的概率。也像我上一节在 insert 下面说的，随机选择哈希函数可以帮助打破可能的循环模式。
   - 使用 necklace[4]：这篇论文里面写到了一种解决无限循环问题的方法。通过设置一种 concise data structure 去记录各个 position，然后利用 BFS 来找通往空闲 bucket 的路径，从而一定程度上避免无限循环

2. 为了有效存储，在 Cuckoo-driven Way 中，可以采取以下措施:
   - 压缩哈希表：可以使用一些压缩技术来减少哈希表的存储空间，如哈希表压缩。
   - 合并哈希表项：可以将一些相邻的哈希表项合并成一个大的哈希表项，从而减少哈希表的存储空间。
   - 使用紧凑的哈希表:可以使用紧凑的哈希表来减少哈希表的存储空间，如线性探测哈希表。
   - 使用 FAST[2]：引入一个扁平结构（flat structure）来改进 Cuckoo 哈希。在这个结构中，每个哈希表的位置都可以存储多个元素，而不仅仅是一个。这样，当插入一个新元素时，如果发现目标位置已经被占用，那么可以直接将新元素添加到这个位置的列表中，而不需要移动已存在的元素。这可以大大减少哈希冲突和重新哈希的概率，从而提高插入操作的成功率。

六、实验测试的性能

我把实验结果都保存到了 result 文件夹。文件名形式是 experiment_results_{SIZE}。如图 4 所示。其中第一个数是采用无限循环处理方法的 loop 数目，后面是未对其处理的 loop 数目，Infinite 代表着发生了无限循环。

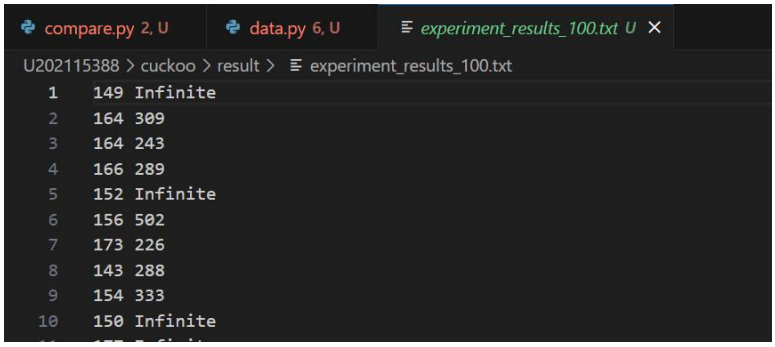

图 4 实验结果

测试的 epoch 我选择了 100，size 分别是 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000。Hash table 的初始化的大小是 size 的十分之一。

本次实验我主要从以下四个方向分析了 cuckoo hash 以及我的对于无限循环处理方法的性能。

1. 如果不对无限循环进行处理导致发生无限循环的概率

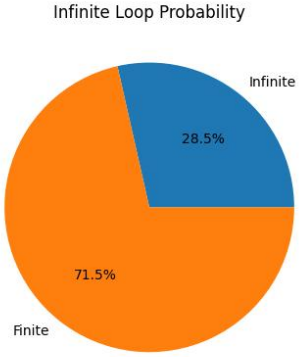

图 5 Cuckoo hash 发生无限循环的概率

2. SIZE 大小对于发生无限循环的影响

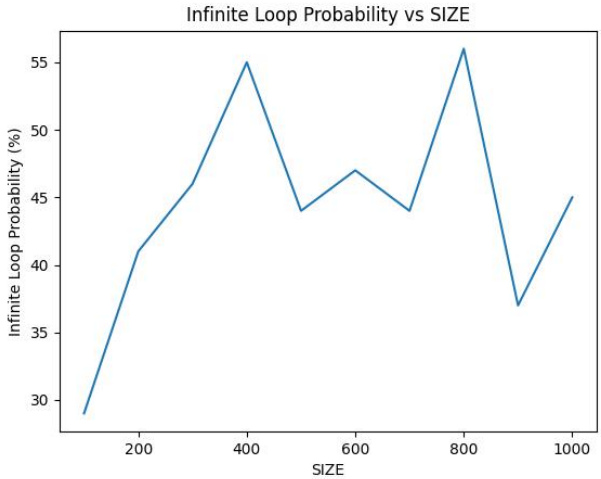

图 6 SIZE 大小对于发生无限循环的影响

我推测的规律不是很明显的原因是我的 hash table 的大小不是固定的。

3. 当不发生无限循环时，不同策略对于 loop count 的影响

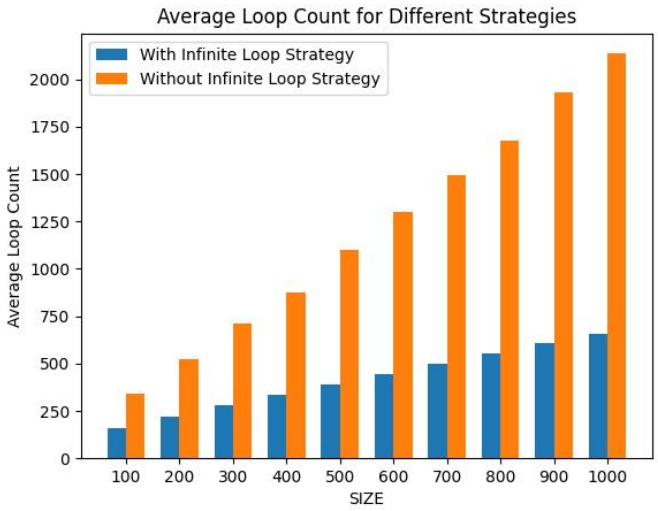

图 7 不同策略对于 loop count 的影响

可以明显看到，无论 SIZE 的大小是多少，我对无限循环做出处理之后，所需要的循环数均小于不做处理的循环数。(在这里计算平均数的时候，如果发生了无限循环，那么这个数据我是不会加进去的)。

4. SIZE 大小对于提升性能的影响

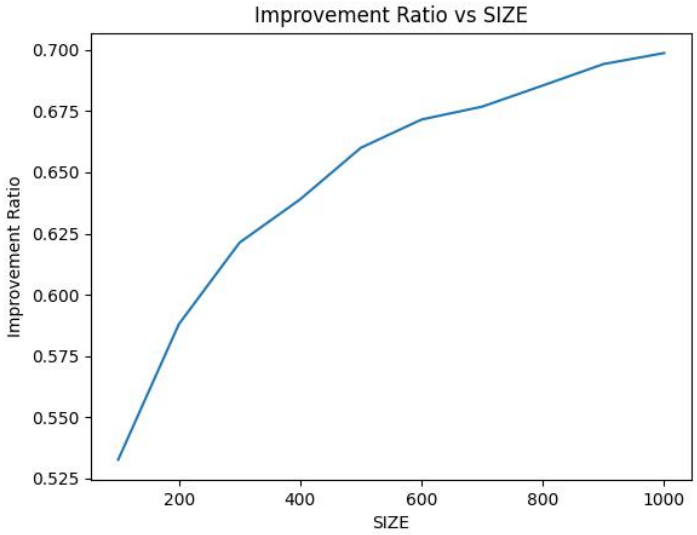

图 8 SIZE 大小对于提升性能的影响

这个结果对应着图 7 也可以看出来，当 SIZE 越大的时候，减少循环的次数就越多，提升越大。

七、实验心得

在本课程中深入学习了大数据存储有关的知识，对于大数据方面的前沿知识有了更深刻的了解。

在本实验中，我选择了 Cuckoo-driven Way，深入学习了这种哈希的方法，并了解了其中的补足(发生无限循环)，分析了如何减少死循环的方法，并对此做出了尝试，取得了不错的提升。此外，我还阅读了相关的文献，了解了更加有效的方法。

参考文献

[1] R. Pagh and F. Rodler, “Cuckoo hashing,” Proc. ESA, pp. 121–133, 2001.

[2] 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.

[3] 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.

[4] 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.

[5] B. Fan, D. G. Andersen, and M. Kaminsky, “MemC3: Compact and concurrent memcache with dumber caching and smarter hashing,” Proc. USENIX NSDI, 2013.

[6] B. Debnath, S. Sengupta, and J. Li, “ChunkStash: speeding up inline storage deduplication using flash memory,” Proc. USENIX ATC, 2010

[7] Libcuckoo library. https://github.com/efficient/libcuckoo.