2020 级

《大数据存储系统与管理》课程

课 程 报 告

姓　名 倪志兴

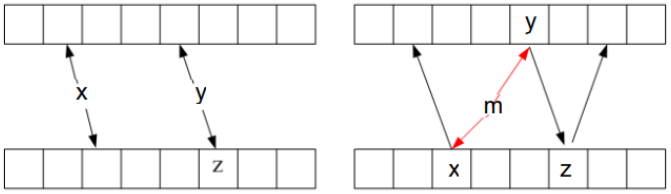
学　号 U202015625

班　号 CS 本硕博 2001 班

日　期 2023.05.25

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

定义：

CuckooHash（布谷鸟散列）是为了解决哈希冲突问题而提出，其目的是使用简单的 hash 函数来提高 hash table 的利用率，同时保证 O(1)的查询时间。基本思想是使用 2 个 hash 函数来处理碰撞，从而每个 key 都对应到 2 个位置。利用较少的计算换取较大的空间。特点是占用空间少，查询速度快。

之所以起这个名字是因为布谷鸟生性贪婪，不自己筑巢，而是在别的鸟巢里面鸟蛋孵化，先成长的幼鸟会将别的鸟蛋挤出，这样独享“母爱”，类似于哈希冲突处理过程。

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 rehash

插入时的循环踢出 rehash。由于标准 Cuckoo hashing 不能预测哪个 item 有空的备用 buckets，只能通过 BFS 或者是随机的方式去决策。会浪费时间不说，还可能造成死循环。

如何减少死循环

1) 尝试预先判断出循环次数，从一开始就不允许进入死循环。
2) 减少踢出过程中的重复部分，使得更多的 buckets 被搜索到，搜索到空桶的概率也会增加
3) 每个桶分配多个槽

三、实现过程：

我们知道实现布谷鸟散列是需要一个散列函数的集合。因此，我们要定义一个接口来获取到这样的一个集合。

```
public interface HashFamily <AnyType>{
    //根据 which 来选择散列函数，并返回 hash 值
    int hash(AnyType x, int which);
    //返回集合中散列函数的个数
    int getNumberOfFunctions();
    //获取到新的散列函数
    void generateNewFunctions();
}
```

定义变量：

```
    private static final double MAX_LOAD = 0.4;
    private static final int ALLOWED_REHASHES = 1;
    private static final int DEFAULT_TABLE_SIZE = 101;
    private final HashFamily<? super AnyType> hashFunctions;
    private final int numHashFunctions;
    private AnyType[] array;
    private int currentSize;
    private int rehashes = 0;
    private Random r = new Random();
```

初始化操作：

```
    public CuckooHashTable(HashFamily<? super AnyType> hf){
        this(hf, DEFAULT_TABLE_SIZE);
    }
    //初始化操作
    public CuckooHashTable(HashFamily<? super AnyType> hf, int size){
        allocateArray(nextPrime(size));
        doClear();
        hashFunctions = hf;
```

```
        numHashFunctions = hf.getNumberOfFunctions();
    }

    public void makeEmpty(){
        doClear();
    }
    //清空操作
    private void doClear(){
        currentSize = 0;
        for (int i = 0; i < array.length; i ++){
            array[i] = null;
        }
    }
    //初始化表
    private void allocateArray(int arraySize){
        array = (AnyType[]) new Object[arraySize];
    }
```

定义 hash 函数：

```
    private int myHash(AnyType x, int which){
        //调用散列函数集合中的 hash 方法获取到 hash 值
        int hashVal = hashFunctions.hash(x, which);
        //再做一定的处理
        hashVal %= array.length;
        if (hashVal < 0){
            hashVal += array.length;
        }
        return hashVal;
    }
```

查询元素是否存在：

```
    /**
     * 查询元素的位置，若找到元素，则返回其当前位置，否则返回-1
     */
    private int findPos(AnyType x){
        //遍历散列函数集合，因为不确定元素所用的散列函数为哪个
        for (int i = 0; i < numHashFunctions; i ++){
            //获取到当前 hash 值
            int pos = myHash(x, i);
            //判断表中是否存在当前元素
            if (array[pos] != null && array[pos].equals(x)){
                return pos;
            }
```

```
        }
        return -1;
    }
public boolean contains(AnyType x){
        return findPos(x) != -1;
    }
```

- 删除元素：

```
    /**
     * 删除元素：先查询表中是否存在该元素，若存在，则进行删除该元素
*/
    public boolean remove(AnyType x){
        int pos = findPos(x);
        if (pos != -1){
            array[pos] = null;
            currentSize --;
        }
        return pos != -1;
    }
```

插入元素：

```
    /**
     * 插入：先判断该元素是否存在，若存在，在判断表的大小是否达到最大
负载，
     * 若达到，则进行扩展，最后调用 insertHelper 方法进行插入元素
     */
    public boolean insert(AnyType x){
        if (contains(x)){
            return false;
        }
        if (currentSize >= array.length * MAX_LOAD){
            expand();
        }
        return insertHelper(x);
    }
```

具体的插入过程：

```
 * a. 先遍历散列函数集合，找出元素所有的可存放的位置，若找到的位置为
空，则放入即可，完成插入
 * b. 若没有找到空闲位置，随机产生一个位置
 * c. 将插入的元素替换随机产生的位置，并将要插入的元素更新为被替换的
元素
```

* d. 替换后，回到步骤 a.
* e. 若超过查找次数，还是没有找到空闲位置，那么根据 rehash 的次数，判断是否需要进行扩展表，若超过 rehash 的最大次数，则进行扩展表，否则进行 rehash 操作，并更新散列函数集合

```
    private boolean insertHelper(AnyType x) {
        //记录循环的最大次数
        final int COUNT_LIMIT = 100;
        while (true){
            //记录上一个元素位置
            int lastPos = -1;
            int pos;
            //进行查找插入
            for (int count = 0; count < COUNT_LIMIT; count ++){
                for (int i = 0; i < numHashFunctions; i ++){
                    pos = myHash(x, i);
                    //查找成功，直接返回
                    if (array[pos] == null){
                        array[pos] = x;
                        currentSize ++;
                        return true;
                    }
                }
                //查找失败，进行替换操作，产生随机数位置，当产生的位置不能与原来的位置相同
                int i = 0;
                do {
                    pos = myHash(x, r.nextInt(numHashFunctions));
                } while (pos == lastPos && i ++ < 5);
                //进行替换操作
                AnyType temp = array[lastPos = pos];
                array[pos] = x;
                x = temp;
            }
            //超过次数，还是插入失败，则进行扩表或 rehash 操作
            if (++ rehashes > ALLOWED_REHASHES){
                expand();
                rehashes = 0;
            } else {
                rehash();
            }
        }
    }
```

扩表和 rehash 操作：

```
 private void expand(){
        rehash((int) (array.length / MAX_LOAD));
    }

    private void rehash(){
        hashFunctions.generateNewFunctions();
        rehash(array.length);
    }

    private void rehash(int newLength){
        AnyType [] oldArray = array;
        allocateArray(nextPrime(newLength));
        currentSize = 0;
        for (AnyType str : oldArray){
            if (str != null){
                insert(str);
            }
        }
    }
```

进行测试：

```
public class CuckooHashTableTest {
    //定义散列函数集合
    private static HashFamily<String> hashFamily = new
HashFamily<String>() {
        //根据 which 选取不同的散列函数
        @Override
        public int hash(String x, int which) {
            int hashVal = 0;
            switch (which){
                case 0:{
                    for (int i = 0; i < x.length(); i ++){
                        hashVal += x.charAt(i);
                    }
                    break;
                }
                case 1:
                    for (int i = 0; i < x.length(); i ++){
                        hashVal = 37 * hashVal + x.charAt(i);
                    }
                    break;
            }
            return hashVal;
        }
```

```
        //返回散列函数集合的个数
        @Override
        public int getNumberOfFunctions() {
            return 2;
        }

        @Override
        public void generateNewFunctions() {

        }
    };

    public static void main(String[] args){
        //定义布谷鸟散列
        CuckooHashTable<String> cuckooHashTable = new
CuckooHashTable<String>(hashFamily, 5);
        String[] strs = {"abc","aba","abcc","abca"};
        //插入
        for (int i = 0; i < strs.length; i ++){
            cuckooHashTable.insert(strs[i]);
        }
        //打印表
        cuckooHashTable.printArray();
    }
}
```

运行结果：

```
当前散列表如下：
表的大小为：13
current pos: 1 current value: abca
current pos: 3 current value: abcc
current pos: 6 current value: aba
current pos: 8 current value: abc
```

四 总结

通过此次课程任务我对 Cuckoo Hashing 有了更多了解，也初步测试了一下，学习了很多东西。Cuckoo hash 利用较少的计算换取较大的空间,但是可能会产生死循环，需要进行一定的优化，相比于 bloom filter 有各自的优点。

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.