2019 级

《物联网数据存储与管理》课程

课 程 报 告

姓　名　胡嘉慧

学　号　U201990067

班　号　计算机 1908 班

日　期　2022.04.14

目　　录

一、摘要

Cuckoo hashing 用于解决表中散列函数值的散列冲突，具有最坏情况下的 O(1) 查找时间。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。空间利用率不高，当 load factor 低於 50%时，才能顺利插入数据。

本文基于 Cuckoo hashing 结构进行了研究，提出了一种以增加哈希桶的维度来减少 cuckoo 操作中的无限循环的概率和有效存储。

关键词：Cuckoo hashing ， hash

二、选题背景与意义

Cuckoo Hashing 是一种解决哈希表碰撞的技术，它产生的字典具有恒定时间的最坏情况下的查找和删除操作，以及摊销的恒定时间的插入操作。最早由 Rasmus Pagh 和 Flemming Friche Rodler 在 2001 年提出，作为以前的静态字典数据结构的扩展。它的基本思想是使用两个哈希函数 h1 (x)、h2 (x) 和两个哈希桶 T1、T2，当要插入元素 x 时，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，此过程如图 2.1 所示。

如果插入时，踢出次数过多，则说明哈希桶满了需要行扩容、重哈希后再次插入。

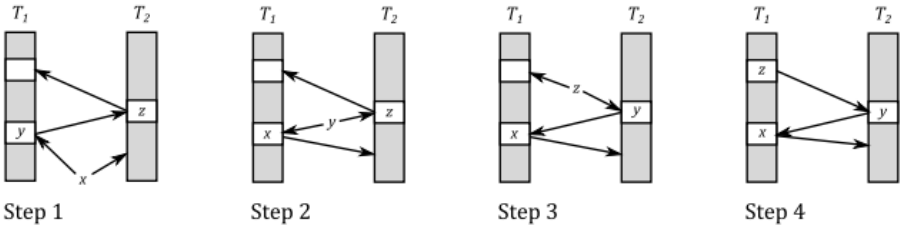

图 2.1 插入 x 的过程实例

Cuckoo Hashing 还有很多变种，例如:

1. 使用两个哈希函数和一个哈希桶，如图 2.2 所示。

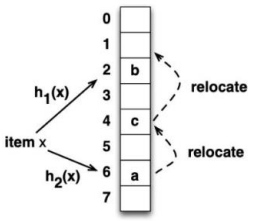

图 2.2 变种 1

2. 使用两个哈希函数和四路哈希桶。如图 2.3 所示。

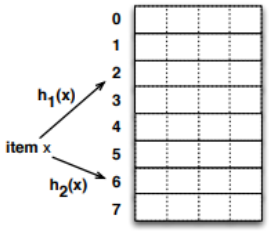

图 2.3 变种 2

本文尝试改善 Cuckoo Hashing 中的一些缺点，例如提高空间利用率，减少无限循环和重哈希的概率。

三、总体设计

3.1 基本的 Cuckoo Hashing

基本的 Cuckoo Hashing 使用两个哈希函数 h1 (x)、h2 (x) 和两个哈希桶，如图 3.1 所示。

插入元素可能会引致无限循环问题，此时需要使用重哈希解决此问题，设置一个计数器，当踢出次数超过 200 次就进行重哈希。

一共有三种操作:

1. 插入元素 x

   两个哈希桶的对应位置有一个为空位，则插入，若两者都空，随便选一个插入。如果两个哈希桶的对应位置都满，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，此过程如图 2.1 所示。

2. 查询元素 x

   读取两个哈希桶的对应位置和 x 比对即可。

3. 删除元素 x

   读取两个哈希桶的对应位置和 x 比对，若成功查询到 x，则在对应位置删除该元素。

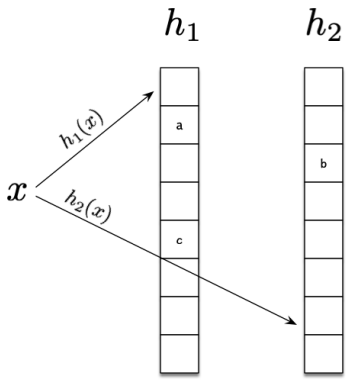

图 3.1 基本的 Cuckoo Hashing 结构图

3.2 改进后的 Cuckoo Hashing

改进后的 Cuckoo Hashing 使用两个哈希函数 h1 (x)、h2 (x)，采用两个二维哈希表，如图 3.2 所示。这是参照图 2.3 的变种 Cuckoo Hashing 和基本的 Cuckoo Hashing 的结构得出来的，同时结合这两种 Cuckoo Hashing 的特性。

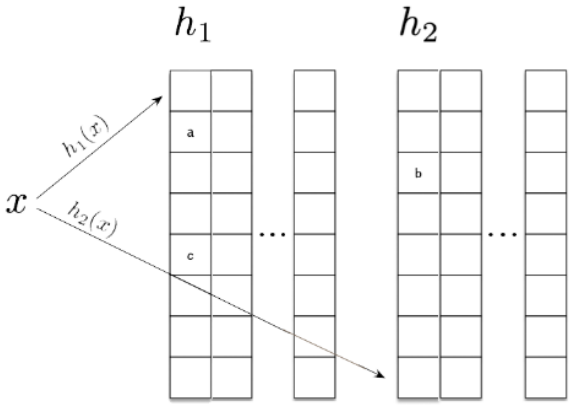

图 3.2 改进后的 Cuckoo Hashing 结构图

一共有三种操作:

1. 插入元素 x

   两个哈希桶的对应位置的所有 slot 有一个为空位，则插入，若两者都空，随便选一个插入。如果两个哈希桶的对应位置都满，则随便选择其中一个哈希桶的 slot(设为 y)，将其踢出，插入 x。重复上述过程，插入元素 y。

2. 查询元素 x

   读取两个哈希桶的全部 slot，直到找到元素 x 的对应位置。

3. 删除元素 x

   读取两个哈希桶的全部 slot，直到找到元素 x 的对应位置。若成功查询到 x，则在对应位置删除该元素。

改进后的 Cuckoo Hashing 会减少引致无限循环问题的概率，因为有 slot 可以

提供更多的位置来存放元素，但当遇到无限循环问题时，也需要使用重哈希解决问题。

四、理论分析

基本的 Cuckoo Hashing 的插入操作时间复杂度、查询操作时间复杂度和删除操作时间复杂度均为 O(1)。改进后的 Cuckoo Hashing 的除了查询操作时间复杂度和删除操作时间复杂度的最坏情况下会变为 O(s)外(s 为 slot 数)，其余与基本的 Cuckoo Hashing 无异。

虽然改进后的 Cuckoo Hashing 的查询操作时间复杂度和删除操作时间复杂度变大了，但还是常量增大，差别并不大，但却能大大增加空间利用率和减无限循环次数。

五、实验测试

5.1 实验设计

设 m 是哈希桶的长度，n 是元素的数量，s 是哈希桶的维度，α 是 Load Factor。

Load Factor: The *load factor* of a hash table T with n keys is

$$\text{load factor} := \frac{n}{|T|}.$$

图 5.1 Load Factor 公式

根据图 5.1 可得，$\alpha = n/2ms (0 \le \alpha \le 1)$。

测试 Load Factor 过程从 α = 30%开始，每次增加 5%，直到发生错误时将上次成功的 α 每次加 0.5%测试，直到成功。测试数据是一个不重复序列，每一次测试都分为四个步骤:

（1）插入不重复序列中的所有元素。

（2）查询不重复序列中的所有元素，此时每次查询都一定会成功。

（3）查询不重复序列中的所有元素。

（4）查询不重复序列中的所有元素，此时每次查询都一定会失败。

最后分别记录每个步骤运行的时间，以及每次测试发生的重哈希次数。

5.2 实验结果与分析

实验程序输出结果如图 5.2 所示。

```
m: 1000000 n: 600000 s: 1
insert cost = 89.94ms
search cost = 38.34ms
delete cost = 80.75ms
search cost = 32.42ms
total crash = 0
```

图 5.2 程序输出结果

实验测试结果如图 5.3，纪录在命名为 CuckooHashing data 的 excel 表格中。

哈希桶的长度 m	元素的数量 n	维度数 s	Load Factor α	插入花费时间(ms)	查询成功花费时间(ms)	删除花费时间(ms)	查询失败花费时间(ms)	重哈希次数
1000000	600000	1	30%	89.98	38.34	80.75	32.42	0
1000000	700000	1	35%	141.75	40.6	69.24	37.86	0
1000000	800000	1	40%	188.47	49.22	79.52	52.93	0
1000000	900000	1	45%	181.78	55.94	109.34	58.3	0
1000000	1000000	1	50%	1635.61	52.65	80.42	54.64	6
1000000	1010000	1	50.5%	3777.66	54.33	88.88	52.2	34
250000	600000	4	30%	89.62	51.09	108.24	52.13	0
250000	700000	4	35%	147.13	57.33	74.06	43.06	0
250000	800000	4	40%	156.65	54.27	72.48	60.18	0
250000	900000	4	45%	279.66	109.32	151.23	84.15	0
250000	1000000	4	50%	207.88	97.01	155.89	98.4	0
250000	1100000	4	55%	235.5	119.8	186.55	106.26	0
250000	1200000	4	60%	271.02	124.68	202.14	118.12	0
250000	1300000	4	65%	305.24	149.64	291.37	111.99	0
250000	1400000	4	70%	1127.09	96.08	149.76	103	0
250000	1500000	4	75%	1866.17	103.43	153.62	90.29	0
250000	1520000	4	76%	6260.02	117.54	172.47	102.67	36
125000	600000	8	30%	131.08	72.66	128.15	60.94	0
125000	700000	8	35%	194.54	74.86	95.83	62.56	0
125000	800000	8	40%	202.54	86.03	115.55	92.59	0
125000	900000	8	45%	221.83	997.31	131.14	106.47	0
125000	1000000	8	50%	189.45	107.19	144.56	98.66	0
125000	1100000	8	55%	253.1	157.48	184.44	149.96	0
125000	1200000	8	60%	278.33	159.18	198.95	137.11	0
125000	1300000	8	65%	291.66	194.45	299.15	127.35	0
125000	1400000	8	70%	1977.55	100.71	129.29	108.2	0
125000	1500000	8	75%	1103.64	109.72	151.14	103.77	0
125000	1600000	8	80%	1143.55	144.58	186.24	129.94	0
125000	1630000	8	81.5%	1422.54	146.86	268.53	135.79	2

图 5.3 测试结果

根据图 5.3 的数据可得基本的 Cuckoo Hashing 和改进后的 Cuckoo Hashing 的 α 对比。当 slot 数为 1 时，即是基本的 Cuckoo Hashing，最大空间占用率大概是 50.5%，当 slot 数为 4 时，最大空间占用率大概是 76%，当 slot 数为 8 时，最大空间占用率大概是 81.5%。显然，增加 slot 的增量可以有效地增加最大空间占用率。

根据图 5.3 的数据可得出图 5.4，从图 5.4 可看出四种操作中除了失败查询操作，1 个、4 个和 8 个 slot 的 Cuckoo Hashing 的时间成本都相差不大，但 Load Factor 却大大提高了。

并且可观察到接近最大空间占用率时，时间成本就会突增，其它时候都接近线性增长。

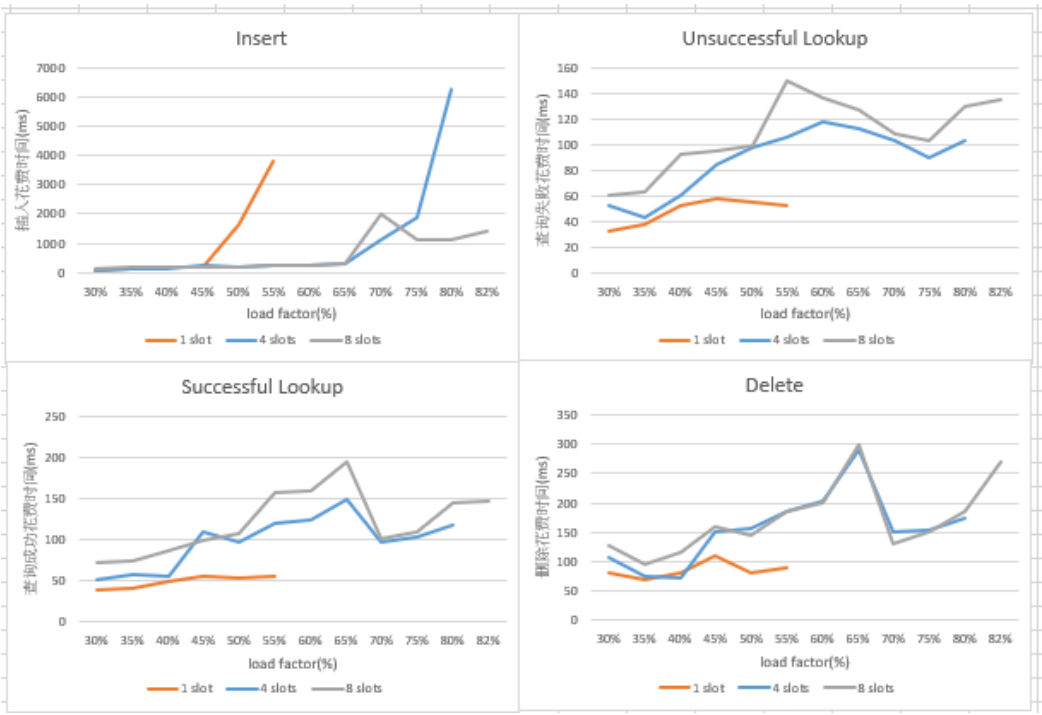

图 5.4 四种操作的时间成本对比

六、总结

这次的实验使得基本的 Cuckoo Hashing 的 Load Factor 有不错的提升，但也增加时间成本，在查阅了相关论文后，发现还能使用 fingerprint 来进一步改进算法，但因为时间关系，没能实现，希望以后会有更多的时间研究。

为了完成这次的课程报告，我查阅了不少的论文，从一开始看到上面密密麻麻的英文和变量名而感到不知所措，到逐渐熟悉并理解，花不了少时间，通过这个过程我发现我现在看论文已经没有这么痛苦了。

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] Charles Chen, "An Overview of Cuckoo Hashing", https://cs.stanford.edu/~rishig/courses/ref/l13a.pdf
[8] Noah Fleming,"Cuckoo Hashing and Cuckoo Filters", https://www.cs.toronto.edu/~noahfleming/CuckooHashing.pdf

代码附录

1. AdvancedCuckooHashing.java(改进后的 CuckooHashing,slot=1 时即基本的 CuckooHashing)

```
import java.util.Arrays;
import java.util.HashSet;

public class AdvancedCuckooHashing {
    private int unitsNum = 2, unitsLen, slotsNum;
    private Hash h = new Hash();
    private HashSet<Integer> elements = new HashSet<Integer>();
    private int[] units;
    public int crash = 0;

    public AdvancedCuckooHashing(int unitsLen, int slotsNum) {
        this.unitsLen = unitsLen;
        this.slotsNum = slotsNum;
        this.units = new int[unitsNum * unitsLen * slotsNum];
    }

    boolean find_s(int unit, int pos, int value) {
        for (int i = 0; i < slotsNum; i++) {
            if (units[unit * unitsLen * slotsNum + pos * slotsNum + i]==
 value)
                return true;
        }
        return false;
    }

    boolean insert_s(int unit, int pos, int value) {
        for(int i = 0; i < slotsNum; i++) {
            if (units[unit * unitsLen * slotsNum + pos * slotsNum + i] =
= 0) {
                units[unit * unitsLen * slotsNum + pos * slotsNum + i]
= value;
                return true;
            }
        }
        return false;
    }

    boolean delete_s(int unit, int pos, int value) {
        for (int i = 0; i < slotsNum; i++) {
```

```
            if (units[unit * unitsLen * slotsNum + pos * slotsNum + i] =
= value){
                units[unit * unitsLen * slotsNum + pos * slotsNum + i]
= 0;
                return true;
            }
        }
        return false;
    }

    int seize_s(int unit, int pos, int value) {
        int v = units[unit * unitsLen * slotsNum + pos * slotsNum + 0];
        units[unit * unitsLen * slotsNum + pos * slotsNum + 0] = value;
        return v;
    }

    boolean insert_e(int value) {
        int index1 = h.hash1(value) % unitsLen, index2 = h.hash2(value)
 % unitsLen;
        if(insert_s(0, index1, value) || insert_s(1, index2, value)){
            elements.add(value);
            return true;
        }

        int cnt = 0, lastElement = value, index = index1;

        while(cnt < new Config().kickMax){
            int kick = seize_s(cnt % 2, index, lastElement);
            int nextPos;
            if(cnt % 2 == 1){
                nextPos = (h.hash1(kick) % unitsLen);
            }else{
                nextPos = (h.hash2(kick) % unitsLen);
            }
            cnt++;
            if(insert_s(cnt % 2, nextPos, kick)){
                elements.add(value);
                return true;
            }
            lastElement = kick;
            index = nextPos;
        }
        return false;
    }
```

```
    boolean insert(int value) {
        if(elements.size() > unitsNum * unitsLen * slotsNum){
            System.out.println("insert failed because over capacity!");
            return false;
        }

        boolean ok = insert_e(value);
        if(ok)
            return true;
        else{
            int t = 0;
            while(t < new Config().rehashMax){
                h.rehash();
                if(rebuild(value)){
                    System.out.println( "have rebuilt "+(t+1) +" times");
                    System.out.println("rebuild successful!");
                    elements.add(value);
                    return true;
                }
                t++;
            }
            System.out.println("insert failed because "+ new Config().rehash
Max +" times of rebuild don't help!");
            return false;
        }
    }

    boolean rebuild(int value) {
        Arrays.fill(units, 0);
        crash++;

        for(int elem : elements)
            if(!(insert_e(elem))){
                System.out.println(elem +" have crashed!");
                return false;
            }
        return insert_e(value);
    }

    boolean delete(int value) {
        int index1 = h.hash1(value) % unitsLen, index2 = h.hash2(value)
 % unitsLen;
        if(delete_s(0, index1, value)){
```

```
                elements.remove(value);
                return true;
            }

            if(delete_s(1, index2, value)){
                elements.remove(value);
                return true;
            }
            return false;
        }

        boolean search(int value) {
            int index1 = h.hash1(value) % unitsLen, index2 = h.hash2(value)
 % unitsLen;
            return find_s(0, index1, value) || find_s(1, index2, value);
        }

}
```

2. AdvancedCuckooHashingTest.java(测试)

```
    import java.math.RoundingMode;
    import java.text.DecimalFormat;
    import java.util.HashSet;

    import static java.lang.System.exit;

    public class AdvancedCuckooHashingTest {

        public static void main(String[] args) {
            int n = new Config().n, m = new Config().m, s = new Config
    ().s;
            HashSet<Integer> hashSet = new HashSet<Integer>();
            for(int i=0; i<n; i++){
                int value = new Config().r.nextInt(1000000000) + 1;
                hashSet.add(value);
            }
            DecimalFormat df = new DecimalFormat("0.00");
            df.setRoundingMode(RoundingMode.HALF_UP);
            AdvancedCuckooHashing cuckoo = new AdvancedCuckooHashin
    g(m, s);

            long startTime = System.nanoTime(); //程序开始记录时间
            for(int i:hashSet){
                if(!(cuckoo.insert(i))){
```

```
            System.out.println("insert "+ i +" error!");
            exit(-1);
        }
}
long endTime = System.nanoTime(); //程序结束记录时间
float TotalTime = (endTime - startTime);
float t1 = TotalTime/1000000;

startTime = System.nanoTime(); //程序开始记录时间
for(int i:hashSet){
    if(!(cuckoo.search(i))){
            System.out.println("search "+ i +" error!");
            exit(-1);
    }
}
endTime = System.nanoTime(); //程序结束记录时间
TotalTime = (endTime - startTime);
float t2 = TotalTime/1000000;

startTime = System.nanoTime();
for(int i:hashSet){
    if(!(cuckoo.delete(i))){
            System.out.println("delete "+ i +" error!");
            exit(-1);
    }
}
endTime = System.nanoTime(); //程序结束记录时间
TotalTime = (endTime - startTime);
float t3 = TotalTime/1000000;

startTime = System.nanoTime(); //程序开始记录时间
for(int i:hashSet){
    if(!(cuckoo.search(i))){
            //System.out.println(" error!");
            //exit(-1);
    }
}
endTime = System.nanoTime(); //程序结束记录时间
TotalTime = (endTime - startTime);
float t4 = TotalTime/1000000;
System.out.println("m: "+ m +" n: "+n+" s: "+s);
System.out.println("insert cost = " + df.format(t1) + "ms");
System.out.println("search cost = " + df.format(t2) + "ms");
System.out.println("delete cost = " + df.format(t3) + "ms");
```

```
            System.out.println("search cost = " + df.format(t4) + "ms");
            System.out.println("total crash = " + cuckoo.crash);
        }

    }
```

3. Config.java

```
    import java.util.Random;

    public class Config {
        final int n = 1300000; // n
        final int m = 125000; //m
        final int s = 8; // s
        final int kickMax = 200;
        final int rehashMax = 100;

        Random r = new Random();
        int random = r.nextInt(19) + 1;//1~20

    }
```

4. Hash.java

```
    import java.util.*;
    public class Hash {
        private int h1,h2,h3,h4;

        public Hash() {
            this.h1 = new Config().random;
            this.h2 = new Config().random;
            this.h3 = new Config().random;
            this.h4 = new Config().random;
        }

        void rehash(){
            this.h1 = new Config().random;
            this.h2 = new Config().random;
            this.h3 = new Config().random;
            this.h4 = new Config().random;
        }
        int hash1(int h){
            h ^= (h << h1) ^ (h << h2);
            return (h ^ (h >> h3) ^ (h >> h4)) & 0x7fffffff;
        }
```

```
    int hash2(int h) {
        h ^= (h << h4) ^ (h << h3);
        return (h ^ (h >> h2) ^ (h >> h1)) & 0x7fffffff;
    }
}
```