2019 级

# 《物联网数据存储与管理》课程

# 课 程 报 告

姓　名　余逸飞

学　号　U202015329

班　号　计算机 2002 班

日　期　2023.05.27

# 目　　录

## 一、选题背景

哈希函数(Hash Function)，又称散列函数，是一类特定的函数，它将任意长度的输入映射为固定长度的输出，从而形成一种数字“指纹”(Fingerprint)。哈希函数被广泛应用在计算机行业的各个领域，具有时间复杂度低，不可逆的特点。

设计哈希函数时的一个非常重要的顾虑就是如何减小哈希冲突(Hash Collide)带来的影响，哈希冲突是指不同的输入经相同哈希函数后被映射到同一值，从而破坏数字“指纹”的唯一性，带来错误的结果。解决哈希冲突的方式有多种，例如链表法、线性探查法、再散列法等等。它们在解决哈希冲突的同时，都会在不同程度上影响哈希函数的理想性能，带来时间和空间上的损耗。如何最大程度减小这些损耗，也成为我们研究的重点。

布谷鸟哈希(Cuckoo Hashing)是一种哈希方法，它设置有多个哈希函数。当存在输入时，先看其通过第一个哈希函数是否产生唯一映射，如果出现哈希冲突，那么将原本的数据挤出并将新的数据存入该处，接着将被挤出数据通过别的哈希函数插入哈希表，如此往复，直到不发生哈希冲突为止。这种哈希冲突的解决方式与布谷鸟的行为类似，故取名布谷鸟哈希。下图是布谷鸟哈希遇到哈希冲突时的处理示意图：

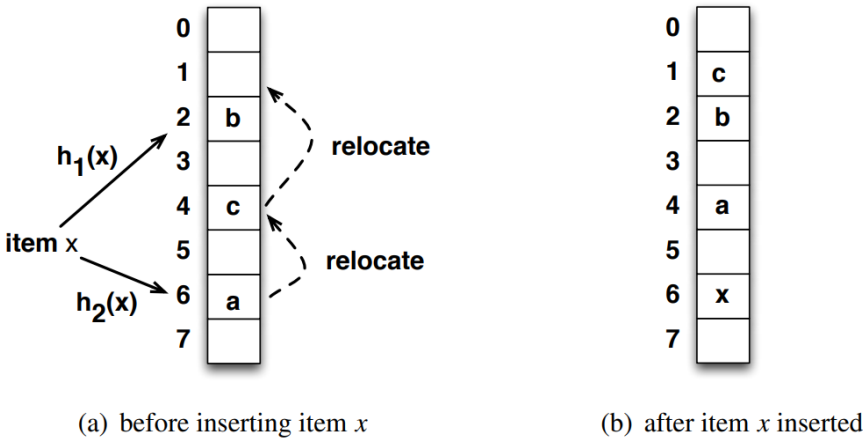


图 1 布谷鸟哈希处理哈希冲突示意图

可以证明，布谷鸟哈希插入操作的均摊时间复杂度是 O(1)。相较于其他哈希方法，布谷鸟哈希具有空间利用率高、查找速度快等优势。

## 二、选题分析

布谷鸟哈希主要的时间开销集中在插入操作中对哈希冲突的处理上：如果出现反复的 kick out，不仅会带来巨大的时间开销，甚至有可能陷入无限循环而无法退出的场景。针对无限循环的情况，简单的处理方式是设置一个循环计数器：一旦循环次数过多，那么退出循环，执行重哈希(Rehash)操作，扩展哈希表空间，最后再完成数据的插入。

优化布谷鸟哈希的主要方向也是去减小哈希冲突的时间消耗。优化思路有两种：

其一是优化哈希表数据结构以减少哈希冲突的出现，其二是优化 kick out 算法来减小处理哈希冲突的时间开销。

本文主要探讨第一种优化方向，也就是优化哈希表数据结构的优化策略。具体思路是将哈希表的每个单元扩展成一个桶，桶内可以存放多个数据，从而在增加一定空间开销的前提下，减少哈希冲突出现的概率。

## 三、算法设计与实现

本节主要介绍数据结构、哈希函数的设计，查找、删除和插入的实现。

首先介绍哈希函数的设计，也即不同映射的设计。我认为哈希函数的设计首先需要确保数据经不同的映射后能够被限制到同一区间，如此方便设置哈希表的大小；其次需要保证数据经不同的映射后的结果尽量不同，否则处理哈希冲突时反复循环的概率将会大大增加。

针对这两点，哈希函数的设计如下：

$$h_i(key) = (key \gg (i * HASH\_BITS)) \,\&\, (1 \ll HASH\_BITS - 1)$$

公式中的 i 代表第 i 个哈希函数。哈希函数本质上是一个按位与，不同的哈希函数选择原数据中不同范围的位进行按位与，从而保证完全随机情况下，不同哈希函数的映射结果基本不同；同时按位与的位数不变，从而保证了数据被映射到同一范围区间。

接着是数据结构的设计，在具体实现时为简化代码框架，我选择使用多个哈希表而非单个哈希表的结构，整体设计思路与上一小节不变。这里使用静态数组，实际实现中建议采用动态申请空间的方法。具体哈希表结构如下：

```
int bucket[HASH_CNT][HASH_SIZE][BUCKET_SIZE];
```

其中，HASH_CNT 表示哈希表的个数，也即哈希函数的个数；HASH_SIZE 表示每个哈希表的长度；BUCKET_SIZE 表示哈希表项所存储的桶大小。

然后是查找和删除操作的实现，这一部分还是比较简单的：利用哈希函数找到相应的映射值，接着在哈希表中对应位置进行各自的操作即可。下面给出两个哈希函数下，插入、删除操作的伪代码：

**Algotithm 1:** Get(k)

**Input:** key **k**, Hash Function1 **h1(x)**, Hash Function2 **h2(x)**, Hash Table **array**
**Output:** Done or Failure

```
1. tag1 = h1(k);
2. tag2 = h2(k);
3. for (i = 0; i < bucket_size; i++) do
4.     if (array[tag1][i] == k or array[tag2][i] == k)
5.         return Done;
6. return Failure;
```

**Algotithm 2:** Delete(k)

**Input:** key **k**, Hash Function1 **h1(x)**, Hash Function2 **h2(x)**, Hash Table **array**
**Output:** Success or Failure

```
1.  tag1 = h1(k);
2.  tag2 = h2(k);
3.  for (i = 0; i < bucket_size; i++) do
4.      if (array[tag1][i] == k) do
5.          array[tag1][i] = 0;
6.          return Done;
7.  for (i = 0; i < bucket_size; i++) do
8.      if (array[tag2][i] == k) do
9.          array[tag2][i] = 0;
10.         return Done;
11.
12. return Failure;
```

最后是插入操作，也是布谷鸟哈希的难点。没有哈希冲突的情况，找到 key 对应的映射值，再到相应位置进行相关操作即可。对于存在哈希冲突的情况，循环进行 kick out 操作，知道哈希冲突消除位置，如果循环进行次数达到上限，则返回插入失败，提示需要扩展空间即可。同样给出伪代码如下：

**Algotithm 3:** Insert(k)

**Input:** key **k**, Hash Function1 **h1(x)**, Hash Function2 **h2(x)**, Hash Table **array**
**Output:** Done or Failure

```
1.  tag1 = h1(k);
2.  tag2 = h2(k);
3.  for (i = 0; i < bucket_size; i++) do
4.      if (array[tag1][i] == 0) do
5.          array[tag1][i] = k;
6.          return Done;
7.  for (i = 0; i < bucket_size; i++) do
8.      if (array[tag2][i] == 0) do
9.          array[tag2][i] = k;
10.         return Done;
11.
12. loop_cnt = 0;
13. while (loop_cnt < loop_max) do
14.     randomly pick a tag using h1 or h2 and mark unused hash function as h'
```

```
15.     randomly pick an entry e in array[tag]
16.     swap k and e
17.     new_tag = h'(k);
18.     for (i = 0; i < bucket_size; i++) do
19.         if (array[new_tag][i] == 0) do
20.             array[new_tag][i] = k;
21.         return Done;
22.     loop_cnt += 1;
23.
24.  return Failure;
```

## 四、测试与分析

首先是标准测试，设置在负载比例为 0.95、哈希函数个数为 2、桶大小为 4，哈希位数为 12 的条件下，要求不出现循环次数超过最大循环次数 1000 的情况。

实际测试时，共计跑了 10000 次测试，没有出现超过循环次数的情形。综上，标准测试通过。

接下来是算法性能的分析。这里主要关注两点：一是哈希表数据结构的设计对算法性能带来的影响；二是不同负载比例下布谷鸟哈希的性能测试。

首先是测试哈希表数据结构的设计对算法性能的影响。实验中测试了哈希函数中桶的大小这两个因素对性能的影响。为了控制变量，哈希表的空间占用率设置为 0.85，哈希函数的个数为 2、哈希表的总大小设置为 2^18 个 int 类型，也就是 1MB，我们将通过调整哈希表的参数来测试其性能。下面是测试结果，所有时间开销均以时钟周期为单位：

| 哈希桶大小 | 总插入时间 | 总循环时间 | 总查找时间 |
|---|---|---|---|
| 16 | 11288 | 62 | 9727 |
| 8 | 11470 | 180 | 9405 |
| 4 | 15180 | 651 | 9181 |
| 2 | 12038 | 1693 | 8915 |

实验结果让我有些意外，总循环时间在总插入时间中占比很小，这与前面理论分析中循环时间占比大的结论相悖。分析可能的原因是我实现的代码中在遇到超过最大循环次数的情况时，并没有执行 Rehash 操作，而是直接退出。而在完整的布谷鸟哈希算法中，Rehash 操作会占据插入操作的相当一部分时间开销。然而我的实现代码中哈希表结构是利用静态数组实现而非使用指针动态申请，这么做虽然简化了插入、查找、删除等操作的代码，但是也给我的程序带来不可扩展的问题。不过总的来说，哈希桶越大，总循环时间越小，这与先前的猜想一致。

其次是不同负载比例下布谷鸟哈希的性能测试。除了负载比例外，其余参数均统一：哈希函数个数为 2、桶大小为 4、哈希位数为 12。测试结果如下：

| 负载比例 | 总插入时间 | 总循环时间 | 总查找时间 |
|---|---|---|---|
| 0.75 | 5723 | 102 | 3890 |
| 0.8 | 6009 | 178 | 4287 |
| 0.85 | 6377 | 322 | 4692 |
| 0.9 | 6236 | 498 | 4901 |
| 0.95 | 8106 | 957 | 5235 |

不难看出，随着负载比例的增大，算法整体的时间开销都在变大。不过由于负载比例小会带来空间的严重浪费，所以我们不能简单的认为负载比例越小算法效率越好。不妨令

总开销=（总插入时间 + 总查找时间）/ 负载比例

如此一来我们得到计算结果如下：

| 负载比例 | 总开销 |
|---|---|
| 0.75 | 12817.3 |
| 0.8 | 12870 |
| 0.85 | 13022.35 |
| 0.9 | 12374.4 |
| 0.95 | 14043.16 |

从上表中我们可以看到，在这样的假设下，负载比例为 0.9 时，总开销最小，算法最优。

## 五、总结

本次实验中，我学习了布谷鸟哈希算法并动手实现了一个相对简化的版本。在确定选题后，我在网上查阅了相关资料并阅读了关于布谷鸟哈希的相关博客、论文，锻炼了我自学以及查找资料的能力；实验对我的另一大收获是锻炼了我的测试能力，这一点更多集中在测试程序的设计上：以往我做实验都是用现成的评测工具或评测程序，而这次需要自己全部实现。如何设计测试去收集一些有用的信息，这一点在以往我并没有过多接触。通过本次实验，我渐渐意识到这一点的重要性，尤其对于将来研究生阶段的深入学习。

当然，针对本次实验我也有许多遗憾之处。首先就是因为一开始的思考不够充分，没有实现动态指针版本的哈希表，缺少了 rehash 部分，导致后面开展测试时发现测试不好展开；同时，在我阅读资料的过程中，我发现布谷鸟哈希有很多优化的方向，比如循环 kick out 算法，是否可以加入一些启发式的思路来减少循环次数？这些方向由于时间问题再加上本人水平有限，最终没能深入研究。将来，我会重写我的布谷鸟哈希算法，在更好的开展测试的同时，增加一些本次没能加入的深入研究。

总的来说，实验带给我的收获还是非常大的。实验本身周期不长，但是在完成本次实验的过程中，我学习到了很多知识。通过测试观察分析程序性能的过程，也让我初步体会到一些做研究的感觉。

## 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