2020 级

《物联网数据存储与管理》课程

课 程 报 告

选　题　Cuckoo-driven Way

姓　名　柳子淇

学　号　U202015628

班　号　本硕博 2001 班

日　期　2023.05.29

目　　录

一、研究背景

布谷鸟过滤器（Cuckoo Filter）是一种高效的数据结构，用于快速判断一个元素是否在集合中出现过。它的名字来自于布谷鸟的习性：布谷鸟喜欢在别的鸟巢里偷窃蛋或幼鸟，而这个过滤器也采用了类似的策略，在有冲突的情况下，将元素放到其他桶里，以此提高存储效率和查找速度。

布谷鸟过滤器最早由麻省理工学院的 researchers 在 2014 年提出，并在同年的 ACM SIGCOMM 会议上发表了论文。该技术在网络路由、缓存、搜索引擎等领域得到了广泛应用，已经成为一项重要的计算机科学研究成果。布谷鸟过滤器已经被广泛应用于各种场景中，如：

网络路由中的路由表匹配；
搜索引擎中的 URL 去重；
缓存系统中的 Key 去重；
数据库系统中的去重操作；
分布式系统中的负载均衡策略

未来，随着数据量的不断增大和计算机技术的不断发展，布谷鸟过滤器的应用将会越来越广泛。同时，也会有更多的研究者投入到这个领域中，探索更加高效、灵活的过滤器算法，以满足不断增长的数据存储和查询需求。

二、设计思路

1. 基本的布谷鸟 Hash

布谷鸟与普通 hash 不同，包含如下几个概念:

指纹：原始数据生成的 “唯一” 标志（来自哈希值）
桶：对应一个数组，每个桶可以存储多个指纹
桶数组：对应多个桶

所以， 可以认为布谷鸟 Hash 维护的是一个存指纹的二维数组。

插入数据

当要插入数据时，通过两个 hash 函数和原始数据 x 算出两个候选桶的位置，即 p1 = hash1(x) % 数组长度。从 h1 桶和 h2 桶中随机找一个放数据指纹。

但是一个桶可存储的指纹是有限的，如果这个桶没满，就往后追加存储即可；如果存满了，就在桶中随机找一个位置，把原有的踢出去，新的指纹替换这个位置。被替换出去的元素要去找它的 h1/h2 桶重新插入。

2. 布谷鸟哈希的优点和存在的问题

cuckoo hash 的假阳率更低，假设有一组特殊数据 a 和 b，他俩指纹相同，在传统 hash 中已经出现了冲突。但在 cuckoo hash 中通过 hash 映射到的候选桶却不一定相同，因为指纹的生成和桶的选取是两个过程。横向对比一下，也就是说达到同样目的（假阳率相同）cuckoo hash 可以使用更短的指纹，占用空间更小。

cuckoo hash 这样设计的数据结构，即占用空间小，而且判断迅速，准确率高等诸多优点。但是也存在一些问题：

指纹过短导致的冲突： 假设再有一个特殊的数据 c 和 d，他俩指纹相同，而且两个候选桶也一致，那么这两个数据就会有假阳性，也就是如果我把 c 放入了布谷鸟过滤器中，我判断 d 是否存在于过滤器中，会返回 true。指纹设置过短就可能引发这种情况，因为信息量有限。然而，即使扩大了空间也不能保证假阳率为 0。这是 hash 性质带来的不可避免的误判。但是我们可以使用更好的哈希函数，合适的空间分配来尽可能避免这种情况。

无限循环的问题：当桶内已经写入了很多数据时，再插入新的数据，发现空间已被占用，于是发生替换，但是由于桶内大部分空间已经被写入。被替换走的指纹的 h1/h2 桶大概率也被占用，于是他会再次替换，又有一个新的指纹被替换出来。这种情况影响了插入性能。甚至会出现无限循环导致插入失败。

例如一种极端情况： 连续插入相同元素。假设每个桶可以存 b 个指纹，重复元素最多能占用 2 * b 个空间，因为相同元素的指纹生成是相同的，而桶选择只有两个，重复元素最多能占用 2 * b 个空间，然后就没有位置给这个元素了，陷入无限循环。

3. 一种高效寻找候补桶的巧妙方法

利用异或运算的自反特性设计公式：

$h1(x) = hash(x), h2(x) = h1(x) \oplus hash(x\text{'s fingerprint})$

这样，使得 h1 和 h2 可以互相转换。

$h1(x) = h2(x) \oplus hash(fingerprint)$

$h2(x) = h1(x) \oplus hash(fingerprint)$

所以，踢出时只需要使用指纹和其中一个候选桶作异或运算，在 O（1）时间内就可以求到另一个候选桶的位置了。这样即使循环次数较多，也能迅速完成寻址过程。

注：也可以直接与 fingerprint 本身作异或，但是这样做会导致 h1 桶和 h2 桶位置间隔很近。尤其是在比较小的 hash 表中，这样无形之中增加了碰撞的概率。

4. 如何减少循环插入开销

源头：首先，hash 函数本身就会发生碰撞，选择更均匀的哈希函数从源头降低碰撞率。本次实验使用 sha256。

特判：针对重复元素插入很容易填满某个桶，轻则使得碰撞率升高，重责导致无限循环。所以尽量在进入 cuckoo hash 前去掉重复元素。

效率: 利用上文提到的异或自反更快地找到 hash2 的位置

动态增长：当 Cuckoo Filter 中的桶已经满载时，动态地增加新的桶。这样可以增加 Cuckoo Filter 的容量，从而减少循环次数。

最后防线: 设置插入次数阈值，超过阈值插入失败。

其他方法：

二次哈希（Double Hashing）：使用两个不同的哈希函数来决定元素存储在哪个桶中。这样可以降低碰撞的概率，从而减少循环次数。

空余区域监测（Vacancy Detection）：在 Cuckoo Filter 中加入一些额外的标记，用于检测

空闲的桶或者位置，这样在插入元素时可以直接利用这些空闲位置，从而减少循环次数。

压缩过滤器（Compressed Cuckoo Filter）：对 Cuckoo Filter 进行压缩，将多个桶压缩成一个单元，这样可以大大减少 Cuckoo Filter 的内存占用，并且也有助于减少循环次数。

三、具体实现

cuckoo.go

1.数据结构：

CuckooFilter 定义如下：

```
type CuckooFilter struct {
M uint64 // 有多少个桶，强制转换为 2 的次幂
B uint64 // 每个桶能存多少个指纹
f uint64
bucket []uint8 // 实际存放桶的空间
}
```

其中，M 表示桶的数量，B 表示每个桶内含指纹数量，f 表示指纹长度。bucket 则是一个字节切片，实际存储了所有桶内的指纹信息。

2.主要函数功能和调用关系：

New：初始化一个 CuckooFilter 对象，设定桶的数量、每个桶内含指纹数量和指纹长度参数。返回初始化后的 CuckooFilter 对象。

一系列用于辅助、计算的子函数

checkPow2：检查一个 unsigned int64 是否为 2 的整次幂。

fastRemain：求 a % b，前提是 b 需要是 2 的次幂，remain = a & (b - 1)。

CalcHash：计算哈希值 h1。对传入的 input 进行 sha256 哈希，并取其前 8 字节（共 32 字节），通过 binary.BigEndian.Uint64 将其转化为 uint64 类型。最后对结果进行 mod m 运算得到 h1。

CalcFinger：计算输入数据的指纹。对传入的 input 进行 sha256 哈希，并取其第 9 字节至第 8+f/8 字节，共 f/8 字节作为指纹。

AltHash：计算哈希值 h2 及 KickOut 的新位置。输入参数为原始哈希值 h1 和要插入的指纹 finger。根据公式 h2 = h1 ^ sha(finger) % m 计算出 h2；如果计算过程中有错误，则返回空值和错误信息。

CheckBucketFull：检查桶是否满了。获取指定桶内最后一个指纹，判断其是否为空（即是否为全 0）。

CheckBucketHaveFinger：检查桶内是否存在某个指纹。在指定桶内搜索，如果找到则返回 true，否则返回 false。

桶操作函数：

InsertBucket：向指定桶内插入指纹。在指定桶内查找一个空位，将指纹存放于此。如果没有空位，则返回错误信息。

ReplaceBucket：替换指定桶中某个指纹。在指定桶内搜索，将指定位置上的指纹替换为新指纹，并返回被替换掉的指纹。

GetBucketFinger：获取指定桶中指定位置上的指纹。在桶的实际存储空间 bucket 中，通过指

定桶和指定位置计算得到指定位置的指纹。
GetEmptyFinger：获取空指纹。返回一个长度为 f 的全 0 切片，用作“无指纹”的占位符。
RandomReplaceInFullBucket：随机替换满桶中的某个指纹。在指定桶中随机选择一个位置，将当前指纹替换为新指纹，并返回被替换掉的指纹。 DeleteFingerFromBucket：从桶中删除指纹。在指定桶内搜索，找到指定指纹后删除并移动其后面的指纹。如果没有找到指定指纹，则返回错误信息。
GetHashFinger：获取哈希值和指纹。传入 input，通过 CalcHash 计算出 h1，通过 CalcFinger 计算出 finger，再根据公式 h2 = h1 ^ sha(finger) % m 计算得到 h2。返回 h1、h2 和 finger。
Contain：判断是否包含某个输入数据。首先调用 GetHashFinger 获取哈希值和指纹，然后分别在 h1 和 h2 对应的桶中查找指纹，如果找到则返回 true，否则返回 false。如果哈希值对应的桶中不包含指纹，则在 h2 对应的桶中查找是否包含该指纹。
Insert：向布隆过滤器中插入一个输入数据。首先调用 Contain 函数判断要插入的数据是否已经存在于过滤器中，如果存在则直接返回；否则调用 GetHashFinger 获取哈希值和指纹，并依次尝试插入到 h1 和 h2 对应的桶中，如果某个桶已经满了，则随机选择其中一个指纹并将其替换为新指纹，并将被替换掉的指纹作为新指纹继续插入。如果连续插入 k 次都失败了，则认为插入失败。
Delete：从布隆过滤器中删除一个输入数据。首先调用 Contain 函数判断要删除的数据是否存在于过滤器中，如果不存在则直接返回；否则调用 GetHashFinger 获取哈希值和指纹，并依次在 h1 和 h2 对应的桶中查找指定指纹并删除之。

test.go

主用于调用 cuckoohash 和测试。用 Go 内部的 Map 结构作存储单元，配合使用 Cuckoo.go 中实现的布谷鸟过滤器。测试并打印性能指标

四、实验测试

表格 1 实验环境

硬件环境	
处理器	I5-1135G7 @ 2.4GHz x8
内存	16G
硬盘	512G SSD
操作系统	Ubuntu20.04
Golang 版本	12.5

实验过程中先开辟一个比较大的 Cuckoo Filter 然后逐渐减小空间使用量，观察对假阳率等其他性能产生的影响。

下面是部分测试数据：

表格 2 测试数据

桶个数	单桶指纹数	指纹位数	item：200	500	1000
128	8	8	随即写入: 200 in 2.000482ms 循环超限:0 循环超限率: 0.00% 误报数量：15876 误报率：1.5876% 平均延迟: 0.8808us	随即写入: 500 in 5.234161ms 循环超限:1 循环超限率: 0.20% 误报数量：33662 误报率：3.3662% 平均延迟: 0.9027us	随即写入: 1000 in 20.885868ms 循环超限:78 循环超限率: 7.80% 误报数量：25807 误报率：2.5807% 平均延迟: 0.9862us
256	8	8	随即写入: 200 in 2.441704ms 循环超限:0 循环超限率: 0.00% 误报数量：9720 误报率：0.9720% 平均延迟: 0.8689us	随即写入: 500 in 5.758268ms 循环超限:0 循环超限率: 0.00% 误报数量：18756 误报率：1.8756% 平均延迟: 0.8924us	随即写入: 1000 in 10.597923ms 循环超限:0 循环超限率: 0.00% 误报数量：33561 误报率：3.3561% 平均延迟: 0.9034us
128	16	8	随即写入: 200 in 2.149185ms 循环超限:0 循环超限率: 0.00% 误报数量：15724 误报率：1.5724% 平均延迟: 0.9089us	随即写入: 500 in 5.315782ms 循环超限:0 循环超限率: 0.00% 误报数量：33468 误报率：3.3468% 平均延迟: 0.9509us	随即写入: 1000 in 11.006319ms 循环超限:0 循环超限率: 0.00% 误报数量：61891 误报率：6.1891% 平均延迟: 0.9838us
128	8	16	随即写入: 200 in 2.35402ms 循环超限:0 循环超限率: 0.00% 误报数量：70 误报率：0.0070% 平均延迟: 0.9046us	随即写入: 500 in 5.069146ms 循环超限:0 循环超限率: 0.00% 误报数量：108 误报率：0.0108% 平均延迟: 0.9482us	随即写入: 1000 in 18.663133ms 循环超限:70 循环超限率: 7.00% 误报数量：92 误报率：0.0092% 平均延迟: 0.9748us

写入条目数量和桶的大小、单个桶内元素数量等参数都有关系。一般来说，较大的桶能够存储更多的元素，从而减少哈希冲突的数量，提高过滤器性能。

五．结论

本文的主要工作是我在学习了各种开源 Cuckoo Hashing 方案后的一个综合复现。理论分析和实验结果都表明， 本次实验基于 GO 的 Cuckoo Hashing 的实现能够通使用较少的计算和空间开销，提升了循环效率，减少了插入操作的总体时间和系统资源开销，增强了 Cuckoo Hashing 性能的稳定性。

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