2020 级

《物联网数据存储与管理》课程

实 验 报 告

选　题　Cuckoo-driven Way

姓　名　吾米提江·阿不来孜

学　号　U202015518

班　号　IOT2001

日　期　2023 年 5 月 20 日

目录

一、研究背景

1.1 研究目的

对于目前日益增长的海量数据处理业务，我们通常需要一个**索引数据结构**用来帮助查询，快速判断数据记录是否存在，这种数据结构通常又叫**过滤器(filter)**。

索引的存储又分为**有序和无序**，前者使用**关联式容器**，比如 **B 树**，后者使用**哈希算法**。这两类算法各有优劣：**关联式容器**时间复杂度稳定 **O(logN)**，且**支持范围查询**；又比如**哈希算法**的查询、**增删都比较快 O(1)**，但这是在理想状态下的情形，遇到**碰撞严重的情况**，哈希算法的时间复杂度会**退化到 O(n)**。因此，**选择一个好的哈希算法是很重要的。**

1.2 常见过滤器的比较

对于此类问题，起初也出现了一些常见的过滤器，但是都存在一定的弊端，下面以布隆过滤器为例：

bloom filter(布隆过滤器)：是一种利用很长的二进制向量和一系列随机映射函数来判断某个或某些元素是否属于某个集。但是他的的位图模式存在两个问题：一个是**误报**，在查询时能提供“一定不存在”，但只能提供“可能存在”，**因为存在其它元素被映射到部分相同 bit 位上，导致该位置 1**，那么一个**不存在的元素可能会被误报成存在**；另一个是**漏报**，如果**删除了某个元素，导致该映射 bit 位被置 0，那么本来存在的元素会被漏报成不存在**。由于**后者问题严重得多**，所以 bloom filter 必须**确保“definitely no”**从而**容忍了“probably yes”，通常就不允许元素的删除了。**

为了解决这一问题，引入了一种**新的哈希算法——cuckoo filter，它既可以确保元素存在的必然性，又可以在不违背此前提下删除任意元素。**

二、基础模型

2.1 CuckooHash 的由来

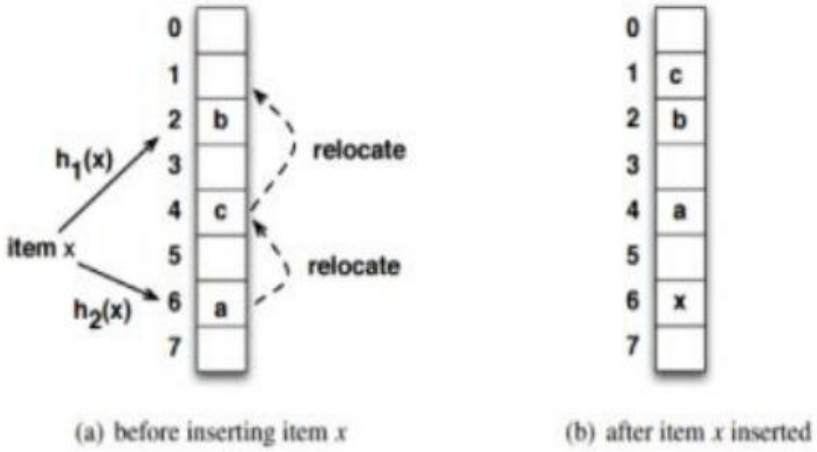
CuckooHash（布谷鸟散列），**最早于 2001 年由 Rasmus Pagh 和 Flemming FricheRodler 提出**，是为了解决哈希冲突问题，利用较少的计算换取较大的空间。**Cuckoo 是布谷鸟的意思**。和普通的哈希表不同，在 Cuckoo Hash 表中，每个元素在 bucket list 中有两个位置可以存放。如果两个位置都被其他元素占了，则随机选择一个位置将其踢走。被踢走的元素被移动到它的备用位置，如果也被其他元素占了则也将其踢走，如此反复（**鸠占鹊巢，其中的鸠就是指布谷鸟**）。分别使用 hash1、hash2 计算对应的 key 位置：

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，则随机踢出一个位置上的 keyx，被踢出的 keyx 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，并进行重新哈希及 rehash。

2.2 CuckooHash 的工作流程

在图(a)中通过计算 x 的哈希值得到两个可用位置，且发现两个可用位置分别都被 a, b 占用了，此时将随机选择一个位置，比如选择元素 a 存放的位置。此时需要将元素 a 从 x 的位置踢出，并将 a 转移到他的备用位置，也就是元素 c 现在

的位置。类似的再对 c 做迁移。最后得到图（b）。（**但是如果 c 的候选位置在 b 这里，b 的又在 x 这里，那么显然会产生无限循环**）



(a) before inserting item *x*　　(b) after item *x* inserted

2.3 CuckooHash 存在的问题

·**写入（插入）性能差**：数据项写入 Cuckoo 哈希表时,如果发生哈希冲突,则将候选桶中任意一个数据项迁移到该数据项的其他候选桶中,并且可能迁移另一个数据项,**直到找到一个空槽或者达到最大搜索阈值**。本质上,**深度优先搜索**的随机替换策略导致 Cuckoo 哈希的 Cuckoo **路径很长**,频繁的“踢出”操作导致其在多个桶之间进行密集的数据迁移,使得哈希表写入性能低下,同时**细粒度锁的大量使用存在死锁和活锁的风险**.基于随机替换策略的 Cuckoo 哈希表的桶之间的负载不均衡,导致在高负载率下易出现**高延迟插入**和**无限循环**。

2.4 CuckooHash 常见的优化

对于常见的哈希表一般通过**开放寻址**（线性探测法）和**链表法**来处理哈希冲突，其中开放寻址在发生地址冲突时，**按照某种方法继续探测哈希表中的其他存储单元，直到找到空位置为止。**特点是不使用指针，直接使用哈希函数来算出存取的位置，这样也可以**加快存取速度**。但开放寻址法可能会**导致之后的 key 被占用而再次产生冲突**。

而在**链表法**中哈希表中的每个元素都是一个类似链表或者其他数据结构的 head。当**出现冲突时，会在其对应位置的链表后面直接添加元素。**虽然这样**完全避免了冲突的发生**，但这也就意味着，如果某一个位置**冲突过多**的话，插入的时间复杂度将**退化为 O(N)。**

而对于 **Cuckoo Hashing** 较为主流的解决哈希冲突的方式主要有下面几种：

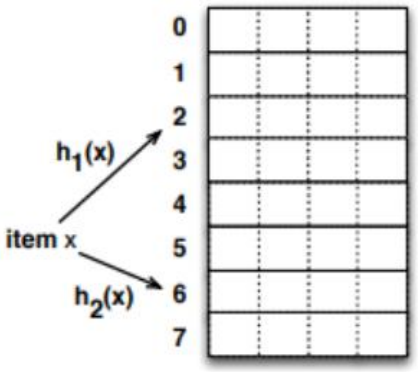
1. 做标记：一种直观的方法是对踢出路径上**经过的位置打上标记**，当遇到已标记的位置时，**表明踢出路径形成了环路**。但是存在**存储标记信息需要额外的空间开销**的问题。

2. 设置阈值：是**最为常见的方法**，当**踢出次数超过踢出阈值后，判定踢出操作陷入环路**。但存在踢出阈值设置的**过大会导致判定环路的开销越大**，设置的**过小又有可能导致较长的非环路被误判成环路引发不必要的 rehash 操作。**

3. 使用多路哈希表和使用多个哈希函数：主要原理就是为存储对象提供更多的备选位置，从而一定程度上减少互踢循环操作的发生。但是会导致其查询性能受到一定的负面影响。

4. 使用广度优先搜索寻找空槽路径：由于大多数 Cuckoo 哈希表**使用随机替换策略搜索空槽**（任意踢走一个再说），故当**用广度优先方法搜索空槽**,桶中的**每个数据项都被视为可能的路径**,并以相同的方式将路径拓展到其他候选桶.广度优先搜索能够以**对数的形式缩短**采用**深度优先搜索 Cuckoo。**

5. 使用多维哈希桶：如下图所示**每个桶**（bucket）有 **4 路槽位**（slot）。当哈希函数映射到同一个 bucket 中，在其它三路 slot 未被填满之前，是不会有元素被踢的，这**大大缓冲了碰撞的几率**。采用二维哈希表（4 路 slot）**大约能达到 80% 的空间利用率**（一维 50%左右）;



(c) A cuckoo filter, two hash per item and functions and four entries per bucket

三、模型优化

3.1 cuckoo 模型优化

经过查阅现有资料以及近年来对于 cuckoo hash 的优化，比较早期且常用的方法已经在上述章节 2.4 中提出，如做标记、设置踢出阈值、使用多路哈希、用广度优先搜索查找空槽路径和使用多维哈希桶等。并且通过阅读我校发布的较新的论文来学习了 **smartcuckoo** 以及 **cocuckoo** 等对于 cuckoo hash 插入性能的**新型优化方法。**

下面主要将介绍 smartcuckoo 的相关理论；

3.2 smartcuckoo 介绍

3.2.1 引言

在处理大量文件时，快速的查询服务对于提高大规模存储系统的整体性能非常重要。开放寻址布谷鸟哈希方案以其简单易用的显著特点被广泛用于支持查询服务。不幸的是，**传统的模式不足以解决在条目插入过程中存在无限循环的潜在问题**，这

会降低查询性能。为了解决这个问题，我们提出了一个成本效益的布谷鸟哈希方案，命名为 SmartCuckoo。

云存储系统快速提供查询服务变得越来越具有挑战性，这通常需要消耗大量资源来支持与查询相关的操作。云**管理系统通常要求支持低延迟和高吞吐量查询**。为了应对这些挑战，**查询服务受到了许多关注**。

SmartCuckoo 背后的想法是将哈希关系表示为有向伪森林，并使用它来跟踪项目位置，以准确地预先确定无限循环的发生。其可以有效地预先确定插入失败，而无需支付进行逐步探测的高成本。我们已经在一个大型云存储系统中实现了 SmartCuckoo。使用三条真实轨迹和 YCSB 基准进行的广泛评估证明了 SmartCuckoo 的效率和功效。

Cuckoo 哈希是一种快速而简单的哈希结构，具有恒定时间的最坏情况查找**(O(ln（1/x）))**，并且消耗**(1+x)n 内存消耗**，其中 **x 是一个小常数**。由于其理想的开放寻址特性和对低查找延迟的支持，布谷鸟哈希已经广泛应用于实际的云应用中。然而，由于**无限循环的出现，布谷鸟哈希遭受了巨大的性能损失**。

大量资源消耗：在无限循环中，插入失败只有在大量内存操作之后才能知道，而这种惩罚可能会极大地损害布谷鸟哈希方案的效率。

不确定的性能：布谷鸟哈希法本质上是通过随机游走来寻找插入项的空桶，因为这种游走的路径上的结果并没有提前获得，故人们很难知道是否存在一个无尽的循环。

SmartCuckoo 利用快速和经济高效的预先确定操作，以帮助**避免**由于无休止的循环而**不必要的踢出过程**。此操作在使用辅助结构开始插入项之前运行。

此外，**插入失败可以在没有任何踢出操作和手动设置迭代阈值的情况下识别**。SmartCuckoo 可以避免杜鹃哈希的无限循环，并提供高性能。

3.2.2 传统布谷鸟图

定义 1：传统的布谷鸟哈希。设 d 为哈希表的个数，S 为键的集合。对于 d = 2 的情况，传统的布谷鸟哈希使用两个大小为 n 的哈希表 T1 和 T2，以及两个哈希函

数 h1, h2: S→{0;:::;n−1}。Key k∈S 可以插入 T1 的 Slot h1(k)或 T2 的 Slot h2(k)，但不能同时插入。两个哈希函数 hi (i = 1 或 2)是独立且均匀分布的。

如下图所示，我们使用一个示例来说明传统布谷鸟哈希中的插入过程。

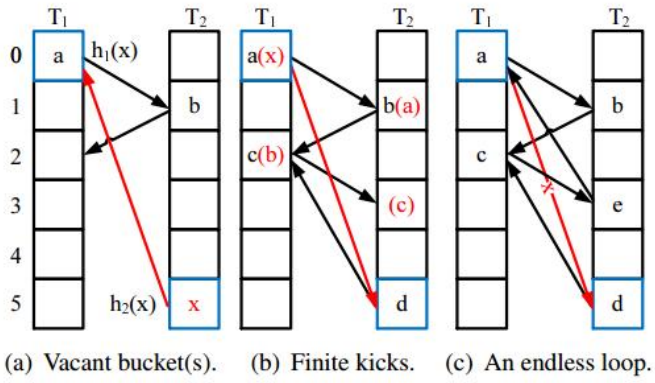


Figure 2: The conventional cuckoo hashing data structure.

在布谷鸟图中，**一条边的起点表示物品的实际存储位置，终点表示备份位置。**例如，存储 Item b 的桶 T2[1]是 Item a 的备份位置。我们打算插入 Item x，它有两个候选位置 T1[0]和 T2[5](蓝色桶)。插入 x 项存在三种情况:

1.两个项(a 和 b)最初位于哈希表中，如图 2(a)所示。当插入项目 x 时，x 的两个候选位置中的一个是空的（T2[5]）。然后将项目 x 放入 T2[5]，并**添加一条指向备份位置(T1[0])的边。**

2.将项目 c 和 d 插入到项目 x 之前的哈希表中，如图 2(b)所示。选项 x 的两个候选位置分别由选项 a 和选项 d 占据。我们必须踢出一个被占用的项目(例如，a)来容纳项目 x。然后将踢出的项目(a)插入其备份位置(T2[1])。这个过程迭代地执行，直到在哈希表中找到一个空桶(T2[3])。**踢出路径是 x→a→b→c。**

3.项目 e 在项目 x 之前插入哈希表，如图 2(c)所示。即使在大量踢出操作之后，也没有可用的空桶来存储 Item x，这**导致了无限循环**。布谷鸟哈希需要进行一次**重哈希操作**（大量资源消耗）。

3.2.3 伪森林理论

伪森林是图论中的无向图，每个最大连通分量(称为子图)**最多有一个循环。**换句话说，它是一个无向图，其中**每个子图的边数不超过顶点数。**在伪森林中，由连续边组成的两个环彼此不共享顶点，并且不能通过连续边的路径相互连接。

为了显示 items（哈希表中的元素） 的实际位置和后备位置的差异，我们**考虑了踢出操作的方向。**在**有向图中**，每条边都是从它的一个端点指向另一个端点。**哈希表中的每个桶最多存储一个项，因此有向伪森林中的每个顶点的出度最多为 1 。如果子图包含一个出度为 0 的顶点，则不包含一个循环，并且该顶点对应于一个空槽。否则，它包含一个循环，任何插入子图都将进入一个无限循环。**

定义 2：极大有向伪森林。极大有向伪森林是一个有向图，其中每**个顶点的出度正好为 1 。**

我们**称顶点数等于边数的子图为极大子图。极大子图包含一个环。极大有向假森林中的任何子图都是极大子图。**

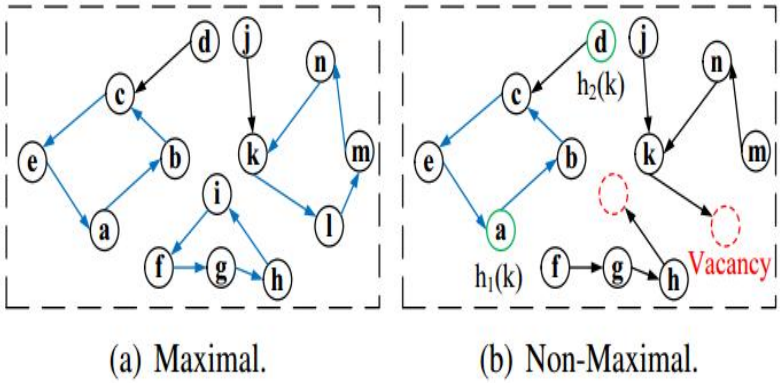


Figure 3: The Directed Pseudoforest.

图 3(a)显示了一个极大有向假森林的例子。极大有向伪林中有三个极大子图。相反，**一个非极大有向伪森林至少有一个非极大子图**，即至少有一个出度为零的顶点。如图 3(b)所示，非极大有向假森林有三个子图，其中两个没有任何环。通过添加一条新边，将任何出线度为零的顶点(图 3(b)中的虚线圆)与图中的任何其他顶点连接起来，即可将其转换为最大有向伪森林。

我们可以认为杜鹃图是一个有向伪森林。伪森林的每个顶点对应于哈希表的一个桶，每个边对应于 itme 的两个候选位置之间的一个 item。

插入的项因此产生一条边。根据这一性质，**极大子图没有容纳新边的空间，当遍历有向边时，最终会导致无限循环**。如此无尽循环。而在**非极大子图中不会遇到循环。**

3.2.4 插入顶点的多种情形讨论

在布谷鸟哈希中插入的项形成布谷鸟图。我们将杜鹃图表示为一个有向假森林，它可以揭示插入的踢出操作的路径，由有向边组成。因此，**有向图可以用来跟踪和提前告诉无尽的循环**，以避免它。

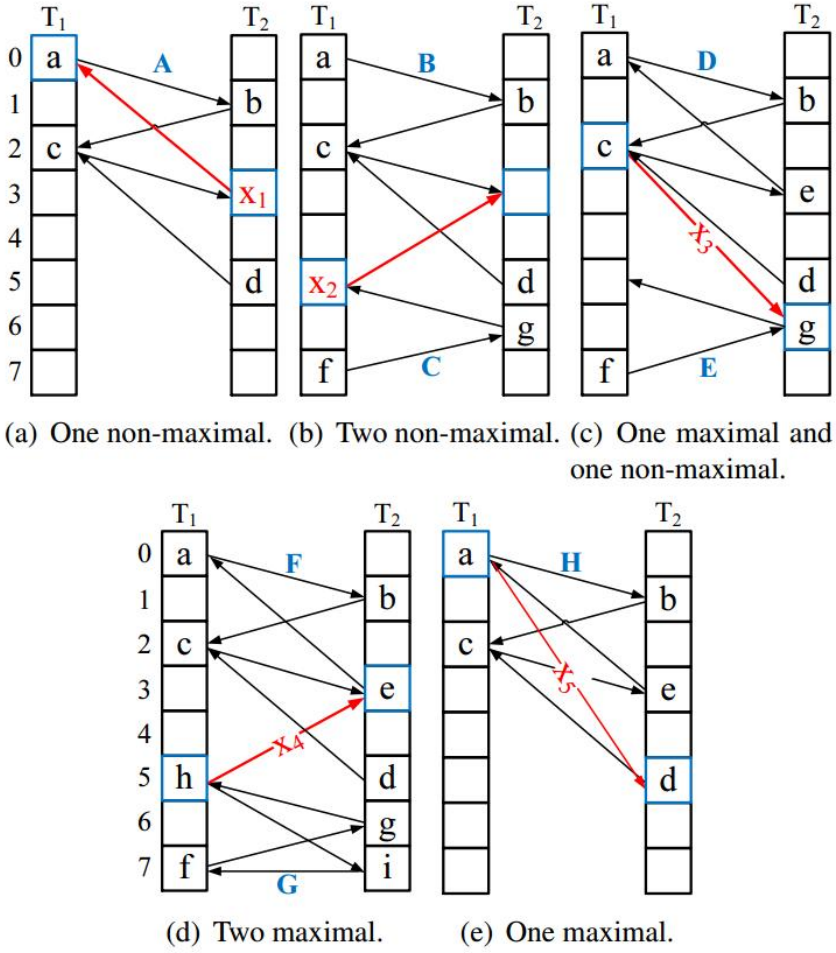
每条边都有一个表示项目实际存储位置的起点和一个表示备份位置的终点。在有向图中，**每个顶点对应一个桶，每个边对应一个项**。由于**存储在哈希表中的项总是不超过桶**，因此在有向图中，**顶点的数量不会小于边的数量**。因此，**一个子图中最多存在一个环**。因此根据有向假森林的性质，**用于表征 SmartCuckoo 中项目放置的有向图是一个有向伪森林。**

当向**伪森林的非极大有向子图插入新项时**，它将**被存储在一个候选桶中**，然后一个被**踢出的项将被存储**在与有向布谷鸟路径的**最后一个顶点对应的空桶中**（**如之前图 3 中的 hk 节点后**）。**如果试图将项目插入到最大有向伪森林中，将不可避免地发生无限循环。**

要在有向伪森林中到达一个空顶点以插入项目，项目的候选桶中必须至少有一个位于包含空位置的子图中。图 3(b)展示了插入项目 k 的过程，它的两个候选位置目前被项目 a 和 d 占用(绿色顶点)，并且在一个没有空缺位置的子图中。尽管假森林中存在两个空位(红色顶点)，但它的插入会遇到一个无尽的循环并失败。因为只**有非极大子图包含空缺位置**，所以插入项目的成功取决于其候选位置是否**至少有一个在非极大子图中。**

·在有向伪森林中，每条边对应一个插入项，每个顶点对应一个桶。

因此，对于要插入哈希表的每个项，**边的数量增加 1。**然而，顶点数(v)的增加有三种情况，即 **v+0、v+1 和 v+2** 的情况。在最**后两种情况下，总能成功插入新项（因为有顶点数的增加必然使得其为一个非极大子图）**，稍后将对此进行解释。首先讨论了有向假森林在 **v+0 情况下**的状态:



(a) One non-maximal. (b) Two non-maximal. (c) One maximal and one non-maximal.

(d) Two maximal. (e) One maximal.

Figure 4: Five scenarios for Case $v+0$.

在不增加顶点数的情况下插入一个项目时，有向伪林中应该存在两个对应于项目的两个候选桶的顶点，这将导致五种可能的场景，如图 4 所示。

（a）item x1 的两个候选桶，如图 4(a)中蓝色桶所示，存在于**同一个非极大有向子图 A 中**。由于踢出操作总是会到达子图中的一个空位置，因此**可以选择任何一个**

桶来成功插入。如图 4(a)所示，Item x1 直接插入到 Bucket T2[3]中，并在 Bucket T2[3]到 Bucket T1[0]之间新建一条边，即 Item x1 的备份位置。插入项目 x1 后，**将原来的非极大有向子图 A 变换为极大有向子图 A '**（a-b-c-x-a），该子图没有空位容纳新项目。

（b）Item x2 的两个候选桶分别位于**两个不同的非极大有向子图 B 和 C 中**，如图 4(B)所示。在这个场景中，插入操作也会成功，因为两个非最大有向子图中的**每一个都提供了一个空桶**。Item x2 位于 Bucket T1[5]中，在有向伪森林中构造一条从 Bucket T1[5]到 Bucket T2[3]的新有向边，**将两个子图 B 和 C 合并成一个新的非极大直接子图(BC)**，其中有一个空顶点(T2[3])。

（c）item x3 的**一个候选桶在非极大有向子图 E 中，另一个在极大有向子图 D 中**，如图 4(c)所示。如果项目从 Bucket T1[2]进入哈希表，则在最大有向子图 D 中会遇到一个无限循环，并进行不必要的踢出操作。然而，如果 Item x3 进入到 Bucket T2[6]的哈希表中，则在多次踢出操作之后，条目插入将成功(在图 5(a)所示的示例中，简单地将 Item g 踢出到 Bucket T1[5])。

（d）item x4 的**两个候选桶被分别在两个极大有向子图(F 和 G)，**如图 4(d)所示。因为在任何子图中都**不存在任何空桶，**所以新项(x4)的插入将**始终进入一个无限循环**，如图 5(b)所示。在传统的布谷鸟散列实现中，**这是最糟糕的插入场景。**

（e）Item x5 的两个候选桶位于**相同的极大有向子图(H)中**，如图 4(e)所示。与前面的场景类似，在**无限循环**中多次踢出后，插入将失败，如图 5(c)所示。

V+1 和 V+2 的情况下的状态:

v+1（x 的某一侯选位置在一个图里）v+2（x 的俩位置都没有在任何图里）

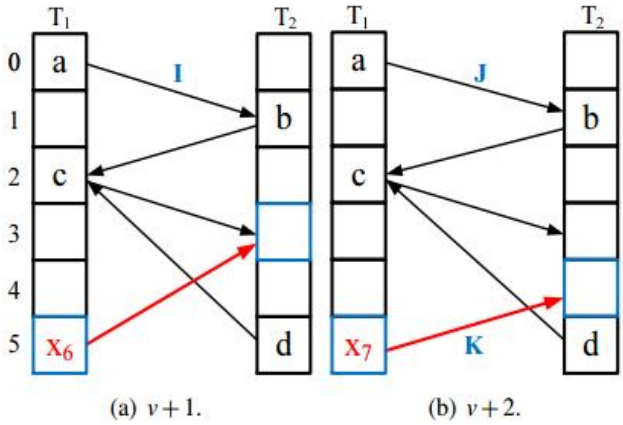


Figure 6: The cases of $v+1$ and $v+2$.

v + 1 表示在插入一个 item 后，有向伪森林中的顶点数增加 1 的情况。如图 6(a)所示，在这种情况下，**Item x6 的两个候选位置中的一个对应于有向假森林中的现有顶点 T2[3]。另一个将是项目插入后的新顶点 T1[5]**（当其插入到 T2[3]时也会产生一个到 T1[5]的边和新顶点）。

在 **v + 2** 的情况下，要插入的项目的**两个候选位置尚未由伪森林中的任何顶点表示。**因此，它们无人居住。该项目可以插入两个可用位置中的任何一个。因此，在伪森林中添加两个顶点，每个顶点对应一个位置。此外，还**添加了一条从对应于项目实际放置位置的顶点到对应于其备份位置的另一个顶点的边。这两个顶点和新边组成一个新的子图 K**，这是**一个非极大有向子图**。这种情况如图 6(b)所示，其中两个顶点分别是桶 T1[5]和 T2[4]，当 Item x7 插入到桶 T1[5]后，新边是从桶 T1[5]到桶 T2[4]。

3.2.5 代码逻辑

根据之前的内容，如果我们事先知道条目插入属于哪一种情况，我们就可以预先确定条目的候选位置是否在无限循环中。这是通过**跟踪子图的状态来实现的，子图要么是最大有向的，要么是非最大有向的。如果一个候选位置在最大有向子图中，它就在一个无限循环中。否则，它就不是一个无限循环。**

在 item 插入操作期间，对于它的每个候选位置，我们需要**找出**它**属于有向伪森林的哪一个或两个子图**。为此，我们对给定的候选位置应用 **Find 操作来确定它所**

属的子图。另外，如果**插入项的两个候选位置属于两个子图，则在子图之间引入一条边，两个子图需要合并**。为此，**需要 Union 操作来合并它们。**为了在 SmartCuckoo 中启用 Find 和 Union 操作，我们为**每个子图分配一个唯一的 ID。子图的每个成员顶点在其对应的桶中记录 ID。当两个子图合并成一个新的子图时，我们引入了 ID 树**，而不是在哈希表上穷尽地搜索一个或两个原始子图的成员顶点来更新它们的子图 ID。在**合并子图的树中，表示合并前子图的桶的 ID 是叶节点，新子图的 ID 是父节点。**新的子图很可能再次与另一个子图合并，并具有其父。**最后，树的根处的 ID 表示从所有前面的子图合并而来的子图。**

为了确定子图在伪森林中的状态，我们跟踪它的边数和顶点数。**如果一个子图的边数等于它的顶点数，那么这个子图就是最大有向子图。**在这种情况下，子图没有空间容纳新的边。否则，边数小于顶点数，子图为非极大有向子图。

·我们可以根据插入所属的三种情况中相关子图的状态来预先确定条目插入的结果：

1.v + 2:当项目 x 的两个候选位置(a 和 b)尚未由有向伪森林中的任何顶点表示时，**插入将创建一个新的子图，**该子图是非极大有向的。因此，可以成功插入新的 Item x。**子图的增加顶点数为 2，边数为 1。**

2.v + 1:运行 **Find(a)和 Find(b)**并找到其中的一个后，检测到这种情况候选位置对应于子图中现有的顶点，而另一个尚未被任何顶点表示。在这种情况下，无论子图处于哪个状态，由于引入了一个新顶点（边不会多于顶点），**插入将是成功的。子图的顶点数和边数都增加 1。**

3.v + 0：在这种情况下，两个候选位置都是子图中的顶点。要知道插入的结果，我们需要确定子图的状态。只有当至少有一个子图是非极大的，插入才算成功。否则，插入将在进入无限循环后失败。**对应子图的边数增加 1。（逐渐逼近极大有向子图**）。

我们描述了 SmartCuckoo 如何支持两种常见的哈希表操作，即**插入和删除**，因为它的查找操作实现与传统的杜鹃哈希基本相同。

插入：

我们使用 B[*]来表示桶中的项。算法 1 描述了插入 x 项所涉及的步骤。首先，我们确定插入属于哪种情况，并增加相应的顶点计数(v)，如算法 2 所述。t 值表示插入的三种情况(v+2、v+1 和 v+0)之一。在 v + 1 和 v + 2 的情况下，Item x 可以直接插入，如算法 3 所示。如果插入情况是 v + 0，我们使用算法 4 来确定对应子图的以下五种情况中的哪一种适用:**(1)一个非极大，(2)两个非极大(第 4-6 行)，(3)一个非极大和一个极大(第 7-13 行)，(4)两个极大，(5)一个极大**。SmartCuckoo 避免走到最大有向子图。由于没有环路，SmartCuckoo 能够有效地减少一条路径上的重复次数，从而减少插入操作的延迟。

删除：

一个 item 只能存储在哈希表的候选位置之一。在删除操作期间，我们只需要探测候选位置，如果在其中一个位置找到，则将其从该位置删除(第 3-4 行)。从哈希表中**删除一个 item 相当于删除相应子图中的一条边，这会导致子图被分成两个子图。**

我们分配给每**两个子图创建一个新的 ID**，并更新两个子图在**其相应桶中的每个成员顶点的 ID**(第 5-6 行)。此外，**更新了两个子图的顶点数和边数。**算法 5 描述了在删除项目 x 时如何维护伪森林。

Algorithm 1 Insert(Item x)

```
1: a ← Hash1(x)
2: b ← Hash2(x) /*Two candidate positions of Item x*/
3: t ← Determine-v-add(a,b)
4: if t == v+2 then
5:     Assign a unique ID to the new subgraph
6:     Union(a,b)
7:     DirectInsert(x,a,b)
8:     Return Ture /*Finish the insertion*/
9: else if t == v+1 then
10:    Union(a,b)
11:    DirectInsert(x,a,b)
12:    Return True /*Finish the insertion*/
13: else
14:    InDirectInsert(x,a,b)
15: end if
```

算法 1 描述了插入 x 项所涉及的步骤。

Algorithm 2 Determine-v-add(Hash a, Hash b)

if neither a nor b have yet existed in the pseudoforest **then**
    Return $v+2$
else if both a and b have existed in the pseudoforest **then**
    Return $v+0$
else
    Return $v+1$
end if

算法 2 描述了，t 值表示插入的三种情况(v+2、v+1 和 v+0)之一

Algorithm 3 DirectInsert(Item x, Hash a, Hash b)

/*a and b are two candidate positions of Item x*/
if $B[a]$ is empty **then**
    $B[a] \leftarrow x$ /*Insert Item x into the empty bucket*/
else
    $B[b] \leftarrow x$
end if

算法 3 描述了可直接插入 x 的情形

Algorithm 4 InDirectInsert(Item x, Hash a, Hash b)

/*Determine type of the corresponding subgraphs*/
if *one non-maximal* **then**
    Kick-out$(x, B[a])$
else if *two non-maximal* **then**
    Kick-out$(x, B[a])$
    Union(a, b)
else if *one non-maximal and one maximal* **then**
    if the subgraph containing a is non-maximal **then**
        Kick-out$(x, B[a])$
    else
        Kick-out$(x, B[b])$
    end if
    Union(a, b)
else
    Rehash()
end if

算法 4 描述了确定对应子图的典型的五种情况

Algorithm 5 Deletion(Item x)

$a \leftarrow Hash_1(x)$
$b \leftarrow Hash_2(x)$ /*Two candidate positions of Item x*/
if $x == B[a]$ *or* $x == B[b]$ **then**
    Delete x from the corresponding position
    Assign two unique IDs to two new subgraphs respectively
    Update subgraph ID
    Update vertex and edge count
    Return *True*
else
    Return *False*
end if

算法 5 描述了删除节点后的森林维护（如果 x 存在，则删除，并分配两个 id，更新子图 id，更新顶点和边数量）。

3.2.6 实验比较

实验使用的服务器配备 Intel 2.8GHz 16 核 CPU, 12GB 峰值带宽为 32GB/s 的 DDR3 RAM, 500GB 硬盘。CPU 的 L1 缓存为 32KB, L2 缓存为 256KB。我们使用三个 trace (RandomInteger ， MacOS，和 DocWords)和 YCSB 基准测试在 Linux 内核 2.6.18 中运行 SmartCuckoo 原型以评估其性能。此外，SmartCuckoo 是基于 CHS 实现的（cuckoo hash with a stash）。

由于篇幅有限，这里仅给出在 YCSB 基准测试在 Linux 内核 2.6.18 中用混合操作检测 SmartCuckoo 原型以评估其性能

·我们使用 YCSB（Yahoo 公司的一个用来对云服务进行基础测试的工具）生成五个工作负载，每个工作负载有 1000 万个键值对，遵循 **zipf 分布**（一种定性的原则，定量来说内容访问近似符合 zipf 定律，如常说的对于内容的访问遵循 80/20 原则）。工作负载中的每个键为 16 字节，每个值为 32 字节。表 1 显示了每个工作负载中不同类型查询的分布。

Workload	Insert	Lookup	Update
YCSB-1	100	0	0
YCSB-2	75	25	0
YCSB-3	50	50	0
YCSB-4	25	75	0
YCSB-5	0	95	5

Figure 13 shows the throughputs of SmartCuckoo

下图显示了运行每个 YCSB 工作负载时 SmartCuckoo 和其他散列表的吞吐量比较。随着插入在工作负载中所占百分比的降低，由于在执行插入操作期间进行昂贵的踢出操作，**所有布谷鸟哈希表的吞吐量都会增加。**其性能比其余三种方法高出 **10%~75%**；且其**插入操作**上的优化以及吞吐量较于其余三类是**更为明显**的。

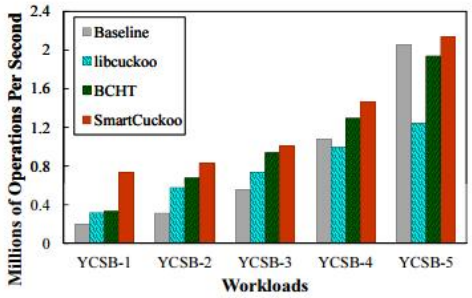


Figure 13: Throughput of mixed operations with YCSB.

四、简易代码实现

假设一个应用场景：有一段文本数据，我们把它通过 cuckoo filter 导入到一个虚拟的 flash 中，再把它导出到另一个文本文件中。flash 存储的单元页面是一个 log_entry，里面包含了一对 key/value，value 就是文本数据，key 就是这段大小的数据的 SHA1 值（照理说 SHA1 是可以通过数据源生成，没必要存储到 flash，但这里主要为了测试而故意设计的，万一 key 和 value 之间没有推导关系呢）。

```
#define SECTOR_SIZE    (1 << 10)
#define DAT_LEN        (SECTOR_SIZE - 20)  /* minus sha1 size */

/* The log entries store key-value pairs on flash and the
 * size of each entry is assumed just one sector fit.
 */
struct log_entry {
        uint8_t sha1[20];
        uint8_t data[DAT_LEN];
};
```

以上是 flash 的存储结构，至于哈希表里的 slot 有三个成员 tag，status 和 offset，分别是哈希值，状态值和在 flash 的偏移位置。其中 status 有三个枚举值：AVAILIBLE，OCCUPIED，DELETED，分别表示这个 slot 是空闲的，占用的还是被删除的。至于 tag，按理说应该有两个哈希值，对应两个哈希函数，但其中一个已经对应 bucket 的位置上了，所以我们只要保存另一个备用 bucket 的位置就行了，这样万一被踢，只要用这个 tag 就可以找到它的另一个备选位置。

```
enum { AVAILIBLE, OCCUPIED, DELETED, };

/* The in-memory hash bucket cache is to filter keys (which is assumed SHA1) via
 * cuckoo hashing function and map keys to log entries stored on flash.
 */
struct hash_slot_cache {
        uint32_t tag : 30;  /* summary of key */
        uint32_t status : 2;  /* FSM */
        uint32_t offset;  /* offset on flash memory */
};
```

其中 size 可以根据情况调整数据类型大小。

至于哈希表以及 bucket 和 slot 的创建见初始化代码。buckets 是一个二级指针，每个 bucket 指向 4 个 slot 大小的缓存，即 4 路 slot，那么 bucket_num 也就是 slot_num 的 1/4。这里我们故意把 slot_num 调小了点，为的是测试 rehash 的发生。

```
#define ASSOC_WAY  (4)  /* 4-way association */

struct hash_table {
    struct hash_slot_cache **buckets;
    struct hash_slot_cache *slots;
    uint32_t slot_num;
    uint32_t bucket_num;
};

int cuckoo_filter_init(size_t size)
{
    ...
    /* Allocate hash slots */
    hash_table.slot_num = nvrom_size / SECTOR_SIZE;
    /* Make rehashing happen */
    hash_table.slot_num /= 4;
    hash_table.slots = calloc(hash_table.slot_num, sizeof(struct hash_slot_cache));
    if (hash_table.slots == NULL) {
        return -1;
    }

    /* Allocate hash buckets associated with slots */
    hash_table.bucket_num = hash_table.slot_num / ASSOC_WAY;
    hash_table.buckets = malloc(hash_table.bucket_num * sizeof(struct hash_slot_cache
*));
    if (hash_table.buckets == NULL) {
        free(hash_table.slots);
        return -1;
    }
    for (i = 0; i < hash_table.bucket_num; i++) {
        hash_table.buckets[i] = &hash_table.slots[i * ASSOC_WAY];
    }
}
```

下面是哈希函数的设计，这里有两个，前面提到既然 key 是 20 字节的 SHA1 值，我们就可以分别是对 key 的低 32 位和高 32 位进行位运算，只要 bucket_num 满足 2 的幂次方，我们就可以将 key 的一部分同 bucket_num － 1 相与，就可以定位到相应的 bucket 位置上，注意 bucket_num 随着 rehash 而增大，哈希函数简单的好处是求哈希值十分快。

```
#define cuckoo_hash_lsb(key, count)  (((size_t *)(key))[0] & (count - 1))
#define cuckoo_hash_msb(key, count)  (((size_t *)(key))[1] & (count - 1))
```

cuckoo filter 最重要的三个操作了——**查询、插入还有删除**。查询操作是简单的，我们对传进来的参数 key 进行两次哈希求值 tag[0]和 tag[1]，并先用 tag[0]定位到 bucket 的位置，从 4 路 slot 中再去对比 tag[1]。只有比中了 tag 后，由于只是 key 的一部分，我们再去从 flash 中验证完整的 key，并把数据在 flash 中的偏移值 read_addr 输出返回。相应的，如果 bucket[tag[0]]的 4 路 slot 都没有比中，我们再去 bucket[tag[1]]中比对（代码略），如果还比不中，可以肯定这个 key 不存在。这种设计的好处就是减少了不必要的 flash 读操作，每次比对的是内存中的 tag 而不需要完整的 key。

```
static int cuckoo_hash_get(struct hash_table *table, uint8_t *key, uint8_t **read_addr)
{
    int i, j;
    uint8_t *addr;
    uint32_t tag[2], offset;
    struct hash_slot_cache *slot;

    tag[0] = cuckoo_hash_lsb(key, table->bucket_num);
    tag[1] = cuckoo_hash_msb(key, table->bucket_num);

    /* Filter the key and verify if it exists. */
    slot = table-&gt;buckets[tag[0]];
    for (i = 0; i bucket_num) == slot[i].tag) {
        if (slot[i].status == OCCUPIED) {
            offset = slot[i].offset;
            addr = key_verify(key, offset);
            if (addr != NULL) {
                if (read_addr != NULL) {
                    *read_addr = addr;
                }
                break;
            }
        } else if (slot[i].status == DELETED) {
            return DELETED;
        }
    }
    ...
}
```

接下来先将简单的删除操作，之所以简单是因为 delete 除了将相应 slot 的状态值设置一下之外，其实什么都没有干，也就是说它不会真正到 flash 里面去把数据清除掉。为什么？很简单，没有必要。还有一个原因，flash 的写操作之前需要擦除整个页面，这种擦除是会折寿的，所以很多 flash 支持随机读，但必须保持顺序写。

```
static void cuckoo_hash_delete(struct hash_table *table, uint8_t *key)
{
    uint32_t i, j, tag[2];
    struct hash_slot_cache *slot;

    tag[0] = cuckoo_hash_lsb(key, table->bucket_num);
    tag[1] = cuckoo_hash_msb(key, table->bucket_num);

    slot = table->buckets[tag[0]];
    for (i = 0; i bucket_num) == slot[i].tag) {
        slot[i].status = DELETED;
        return;
    }
    ...
}
```

如何判断并处理碰撞，其实这里也没啥玄机，就是用 old_tag 和 old_offset 保存一下临时变量，以便一个元素被踢出去之后还能找到备用的安身之所。但这里会有一个判断，每次踢人都会计数，当 alt_cnt 大于 512 时候表示哈希表真的快满了，这时候需要 rehash 了。

```
static int cuckoo_hash_collide(struct hash_table *table, uint32_t *tag, uint32_t *p_offset)
{
    int i, j, k, alt_cnt;
    uint32_t old_tag[2], offset, old_offset;
    struct hash_slot_cache *slot;

    /* Kick out the old bucket and move it to the alternative bucket. */
    offset = *p_offset;
    slot = table->buckets[tag[0]];
    old_tag[0] = tag[0];
    old_tag[1] = slot[0].tag;
    old_offset = slot[0].offset;
    slot[0].tag = tag[1];
    slot[0].offset = offset;
    i = 0 ^ 1;
    k = 0;
    alt_cnt = 0;

KICK_OUT:
    slot = table->buckets[old_tag[i]];
    for (j = 0; j < ASSOC_WAY; j++) {
        if (offset == INVALID_OFFSET && slot[j].status == DELETED) {
            slot[j].status = OCCUPIED;
            slot[j].tag = old_tag[i ^ 1];
            *p_offset = offset = slot[j].offset;
```

```
            break;
        } else if (slot[j].status == AVAILIBLE) {
            slot[j].status = OCCUPIED;
            slot[j].tag = old_tag[i ^ 1];
            slot[j].offset = old_offset;
            break;
        }
    }

    if (j == ASSOC_WAY) {
        if (++alt_cnt > 512) {
            if (k == ASSOC_WAY - 1) {
                /* Hash table is almost full and needs to be resized */
                return 1;
            } else {
                k++;
            }
        }
        uint32_t tmp_tag = slot[k].tag;
        uint32_t tmp_offset = slot[k].offset;
        slot[k].tag = old_tag[i ^ 1];
        slot[k].offset = old_offset;
        old_tag[i ^ 1] = tmp_tag;
        old_offset = tmp_offset;
        i ^= 1;
        goto KICK_OUT;
    }

    return 0;
}
```

rehash 的逻辑也很简单，无非就是把哈希表中的 buckets 和 slots 重新 realloc 一下，空间扩展一倍，然后再从 flash 中的 key 重新插入到新的哈希表里去。这里有个陷阱要注意，千万不能有相同的 key 混进来！虽然 cuckoo hashing 不像开链法那样会退化成 O(n)，但由于每个元素有两个哈希值，而且每次计算的哈希值随着哈希表 rehash 的规模而不同，相同的 key 并不能立即检测到冲突，但当相同的 key 达到一定规模后，噩梦就开始了，由于 rehash 里面有插入操作，一旦在这里触发碰撞，又会触发 rehash，这时就是一个 rehash 不断递归的过程，由于其中老的内存没释放，新的内存不断重新分配，整个程序就如同陷入 DoS 攻击一般瘫痪了。所以每次插入操作前一定要判断一下 key 是否已经存在过，并且对 rehash 里的插入使用碰撞断言防止此类情况发生。

```
lstatic void cuckoo_rehash(struct hash_table *table)
2{
3    ...
4    uint8_t *read_addr = nvrom_base_addr;
5    uint32_t entries = log_entries;
6    while (entries--) {
7        uint8_t key[20];
8        uint32_t offset = read_addr - nvrom_base_addr;
9        for (i = 0; i &lt; 20; i++) {
10           key[i] = flash_read(read_addr);
11           read_addr++;
12       }
13       /* Duplicated keys in hash table which can cause eternal
14        * hashing collision! Be careful of that!
15        */
16       assert(!cuckoo_hash_put(table, key, &offset));
17       if (cuckoo_hash_get(&old_table, key, NULL) == DELETED) {
18           cuckoo_hash_delete(table, key);
19       }
20       read_addr += DAT_LEN;
21   }
22   ...
23}
```

运行效果：找个大文件 input 来测试一下，利用一个嵌入式数据库源代码，共 88888 行代码。作为需要导入的文件，编译我们的 cuckoo filter，然后执行程序。会发现生成 output.c 正好也是 88888 行代码，一分不差，同时也可以看到，cuckoo filter 运行素的也是比较快的。至此完成一个简单的布谷鸟哈希源程序的运行。

五、课程总结

通过这学期大数据存储与管理课程的学习，我了解到当今数字时代每天所产生的数据量有多么的庞大，每过几年总的数据量甚至能跨一个数量级。让我深刻体会到对于海量数据的高效、经济的存储是多么的迫在眉睫。并且在云存储，大数据相关技术，人工智能等新型技术发展的带动下，对于低延迟和高吞吐的查询需求也愈加重要，查询服务也受到了许多关注。相应的寻找一种高效的插入删除等其他附带操作的实现也是如此。

在这门课的理论课程部分，老师主要带我们认识了一些存储介质和存储设备，如一些新型的非易失存储器闪存、相变存储器和一些新型光存储、量子存储和生物存储等新型的研究方向。重点介绍了闪存，ssd 相关的存储结构，工作原理等。也介绍了一些大规模的存储场景中存储设备的应用等。后继又介绍了一些在大规模存储中常用的优化存储的方案，如数据去重。介绍了一些数据去重常用的方法，如分块算法、哈希摘要和指纹查找等等。也相信谈了它们各自的利与弊以及当今的发展现状等等。紧贴当今最新技术的发展，带我们思考了了目前研究人员正在干什么，后来又可能会做点什么等问题。最后在元数据管理的章节，给我们明确了元数据的定义，以及在很多应用场景中元数据查询，及查询某个元素的属性或查询该元素存不存在这个操作的重要性和必要性，介绍了常见的过滤器（filter）如 bloom filter（布隆过滤器）、LHS 和 cuckoo hash 等方案，并让我们从中选择自己感兴趣的课题进行进一步资料查阅和学习。

在本次课程报告中，我选择了 cuckoo hash 去进一步学习探索，主要是觉得当时老师讲的鸠占鹊巢的典故非常有趣，该方案的实现也比较能引起我的兴趣。通过一些从布谷鸟哈希 2001 年被提出到至今的一些论文和资料的查阅。比较了它存在的如设置阈值、使用多路哈希、多维哈希桶、打标记避免循环和用广度优先算法进行空槽查询等一些优化其插入性能的方法。通过华宇参与老师发的两篇文章 smartcuckoo 和 cocuckoo 也重点了解了一下布谷鸟哈希的相对较新的优化方案，并有了一定的理解和启发。对我在体系结构上进一步学习和理解前辈们的论文研究又很大的帮助。

最后我也通过在本地运行较为简单传统布谷鸟哈希方案来体会了一下其查询性能方面较于其他过滤器的优越性，也验证了其相较于 bloom filter 的优越性，及它既可以确保元素存在的必然性，又可以在不违背此前提下删除任意元素。

最后，非常感谢无论是在理论课还是在实验课指导我的华宇老师和施展老师，也非常给学弟学妹们推荐选这门专选进行学习。在这门专选可以非常直观的体验到作为我们华科计算机优势的存储和体系结构方面的知识，也能了解到我们学校自己自主研发的如 SSD 以及和模拟器等一些前沿结果。是一门性价比极高，且真正能够学到东西的专选课！

参考文献

[1] R. Pagh, F. Rodler, "Cuckoo hashing", Proc. ESA, 2001, pages: 121–133.

[2] Yuanyuan Sun, Yu Hua, Song Jiang, Qiuyu Li, Shunde Cao, Pengfei Zuo, "SmartCuckoo: A Fast and Cost-Efficient Hashing Index Scheme for Cloud Storage Systems", Proceedings of USENIX Annual Technical Conference (USENIX ATC), 2017, pages: 553-566.

[3] 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, pages: 50-55.

[4] 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), 2014, pages: 754-765.

[5] 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), 2013, pages: 1327-1335.

[6] A. Kirsch, M. Mitzenmacher, U. Wieder, "More robust hashing: Cuckoo hashing with a stash", SIAM Journal on Computing, 2010, pages: 1543-1561.

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

[8] Yuanyuan Sun, Yu Hua*, Song Jiang Qiuyu Li, Shunde Cao, Pengfei Zuo,"SmartCuckoo: A Fast and Cost-Efficient Hashing Index Scheme for Cloud Storage Systems", Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC ’ 17).

[9] Yuanyuan Sun, Yu Hua, Zhangyu Chen, and Yuncheng Guo, Huazhong University of Science and Technology,"Mitigating Asymmetric Read and Write Costs in Cuckoo Hashing for Storage Systems",Proceedings of the 2019 USENIX Annual Technical Conference.

[10] K. Schwarz. "Cuckoo Hashing", http://web.stanford.edu/class/archive/cs/cs166/cs166.1216/lectures/07/Slides07.pdf, 2021.