2021 级

《大数据存储与管理》课程

实 验 报 告

姓　名　骆传威

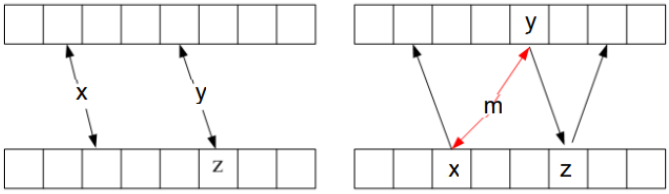
学　号　U202115451

班　号　计算机 2104 班

日　期　2024.04.18

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

2.1 技术背景

对于海量数据处理业务，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。

bloom filter 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“probably yes”，不允许元素的删除。

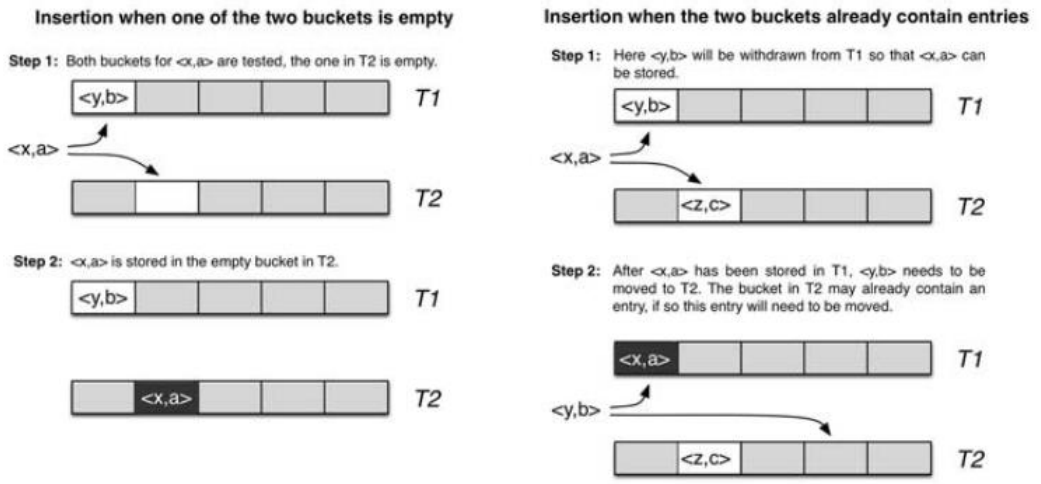
为了解决这一问题，引入了一种新的哈希算法——cuckoo filter，它既可以确保元素存在的必然性，又可以在不违背此前提下删除任意元素，仅仅比 bitmap 牺牲了微量空间效率。

2.2 Cuckoo Filter

Cuckoo Hash（布谷鸟散列）是为了解决哈希冲突问题而提出，利用较少的计算换取较大的空间。

它的哈希函数是成对的（具体的实现可以根据需求设计），每一个元素都是两个，分别映射到两个位置，一个是记录的位置，另一个是备用位置。这个备用

位置是处理碰撞时用的，这就要说到 cuckoo 这个名词的典故了，中文名叫布谷鸟，这种鸟有一种即狡猾又贪婪的习性，它不肯自己筑巢，而是把蛋下到别的鸟巢里，而且它的幼鸟又会比别的鸟早出生，布谷幼鸟天生有一种残忍的动作，幼鸟会拼命把未出生的其它鸟蛋挤出窝巢，今后以便独享“养父母”的食物。借助生物学上这一典故，cuckoo hashing 处理碰撞的方法，就是把原来占用位置的这个元素踢走，不过被踢出去的元素还要比鸟蛋幸运，因为它还有一个备用位置可以安置，如果备用位置上还有人，再把它踢走，如此往复。直到被踢的次数达到一个上限，才确认哈希表已满，并执行 rehash 操作。如下图所示：



优化方式：

①将一维改成多维，使用桶（bucket）的 4 路槽位（slot）；

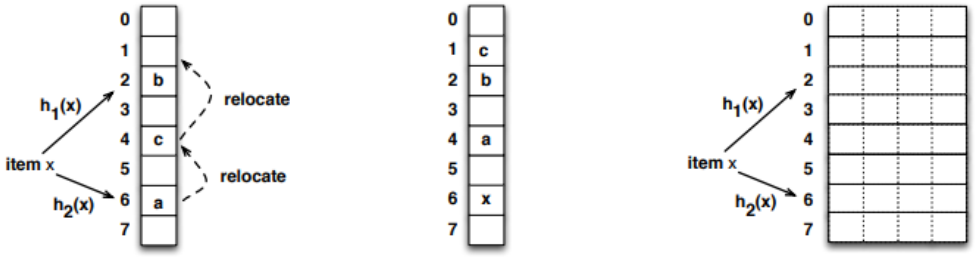
②一个 key 对应多个 value；

③增加哈希函数，从两个增加到多个；

④增加哈希表。

在发生哈希碰撞之前，一维数组的哈希表跟其它哈希函数没什么区别，空间利用率差不多为 50%。

一个改进的哈希表如下图所示，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 80%的占用率（CMU 论文数据据说达到 90%以上，应该是扩大了 slot 关联数目所致）。



(a) before inserting item *x*　　(b) after item *x* inserted　　(c) A cuckoo filter, two hash per item and functions and four entries per bucket

三、Cuckoo Filter 设计与实现

通过 cuckoo filter 把一段文本数据导入到一个虚拟的 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。

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。对于 tag，因其中一个哈希值已经对应于 bucket 的位置上了，所以只要保存另一个备用 bucket 的位置就行，这样万一被踢，只要用这个 tag 就可以找到它的另一个的位置。

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。这里把 slot_num 调小了点，为的是测试 rehash 的发生。

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很快。

cuckoo filter 最重要的三个操作——查询、插入还有删除。

查询操作对传进来的参数 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，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，再去 bucket[tag[1]]中比对，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。

删除操作中的 delete 仅将相应 slot 的状态值设置一下，并不会真正的到 flash 中擦出数据，以免增加设备损耗。

哈希表层面的插入逻辑其实跟查询差不多，不过多说明。这里主要说明如何

判断并处理碰撞，用 old_tag 和 old_offset 保存临时变量，以便一个元素被踢出去之后还能找到备用的位置。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。

Rehash 为将 buckets 和 slots 重新 realloc，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。需要注意的是，不能有相同的 key。

四 理论分析

减少无限循环的方法替换策略

使用一个链表来保存一个桶内的元素，插入元素如果两个桶均满，则随机选一个插入到链表尾部，并从链表首部弹出元素进行后续操作。这样子即便多次选择了同一个桶来进行替换，也能选中不同的元素，每个桶只有在桶内每个元素都被替换过以后才会选中重复的元素，能够大大减小出现无限循环的概率

动态扩容

每次插入元素时都会检测容器的填充因子，如果超过阈值就会触发动态扩容，能够极大减小无限循环的概率。扩容时并不是简单地将容量乘以 2， 而是精心预设了 28 个不同大小的质数作为容量的备选值，这些备选值有助
于使元素的分布更加均匀。

死循环（高次数循环）检测

使用一个 conflicts 变量来检测桶满时发生替换的频率。每次触发替换时，将 conflicts 加上一，每次成功插入时，将 conflicts 除以二。这样，当每次插入时需要替换的次数越来越多时，conflicts 就会开始增长。我对于 conflicts 的阈值设定的是 num_per_bucket*2，这说明平均每次插入时需要替换相当于满满两个桶的量才能成功插入，说明此时的容量或哈希函数已经不适合当前的数据范围，需要更换哈希函数或者扩容。

重新哈希

当 conflicts 达到阈值时会触发重新哈希，这个操作会重新生成随机的哈希函数（即便是用户为自定义的类型提供的哈希函数，也能自动重新生成），然后将元素在容器内重新排列。同时还会统计连续重新哈希的次数，如果连续三次重新哈希都没有解决问题，说明此时的问题已经不是修改哈希函数能解决了，这个时候就会触发动态扩容。

五 实验心得

在本课程中深入学习了大数据存储有关的知识，老师详细的解释了为什么需要大数据存储以及如何发展大数据存储技术，讲解了为什么电脑一掉电便会导致数据丢失和如何解决数据丢失问题。

在本实验中选择 Cuckoo-driven Way 作为选题，分析了如何减少死循环的方法，并在容器满后重新哈希，在实验过程中理解到了大数据存储中的关键问题，即在数据容量增大的同时如何保证原有的程序依旧能正常运行。这对编程思路有了极大的启发与开拓，今后也会加深这方面的思考学习。

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.