大数据存储管理课程报告

姓　　名：　　陈柳伊

学　　院：　　计算机科学与技术

专　　业：　　计算机科学与技术

班　　级：　　CS2105

学　　号：　　U202113863

指导教师：　　华宇

分数	
教师签名	

2024 年 4 月 20 日

利用 Odd-Even Hash Algorithm 改进 Cuckoo Hashing

1. 实验背景

基于哈希的数据结构和算法目前在蓬勃发展。它是存储大量信息的有效方式，尤其是对于与测量、监控和安全相关的应用程序。目前，有许多哈希表算法，如：Cuckoo Hash、Peacock Hash、Double Hash、Link Hash 和 D-left Hash 等算法。

其中，Cuckoo Hashing 的基本思想是使用两个哈希函数 h1(x)和 h2(x)，每个元素 x 可以放在两个位置 h1(x)或 h2(x)中的任意一个。如果一个新插入的元素在两个位置都已被占用，则会踢出一个已有元素，将其放到另一个位置。

然而，Cuckoo Hashing 算法仍然存在一些问题，如内存空间过大，以及由需要重新哈希的无限循环引起的插入失败。

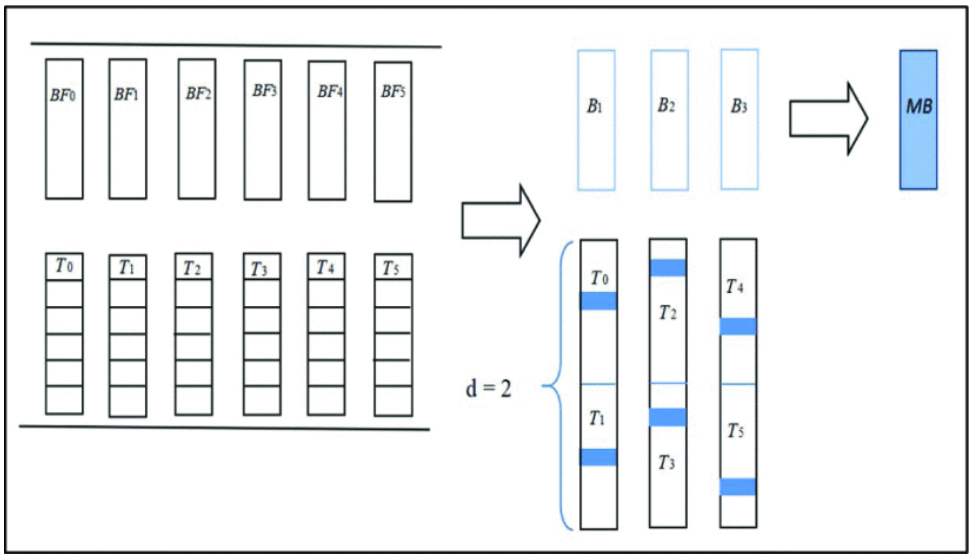
本文从 Cuckoo Hashing 的踢出机制着手，使用了一种新的哈希表结构——Odd-Even Hashing。实验结果表明，OE Hash 算法比现有的 Link Hash 算法、Linear Hash 算法和 Cuckoo Hash 算法等更高效。OE Hash 方法在占用最少空间的同时兼顾了查询时间和插入时间的性能，并且避免盲目踢出操作带来的无限循环问题，适合于海量数据存储。

2. 数据结构的设计

OE 哈希结构由两部分组成：哈希表部分和辅助数据结构。

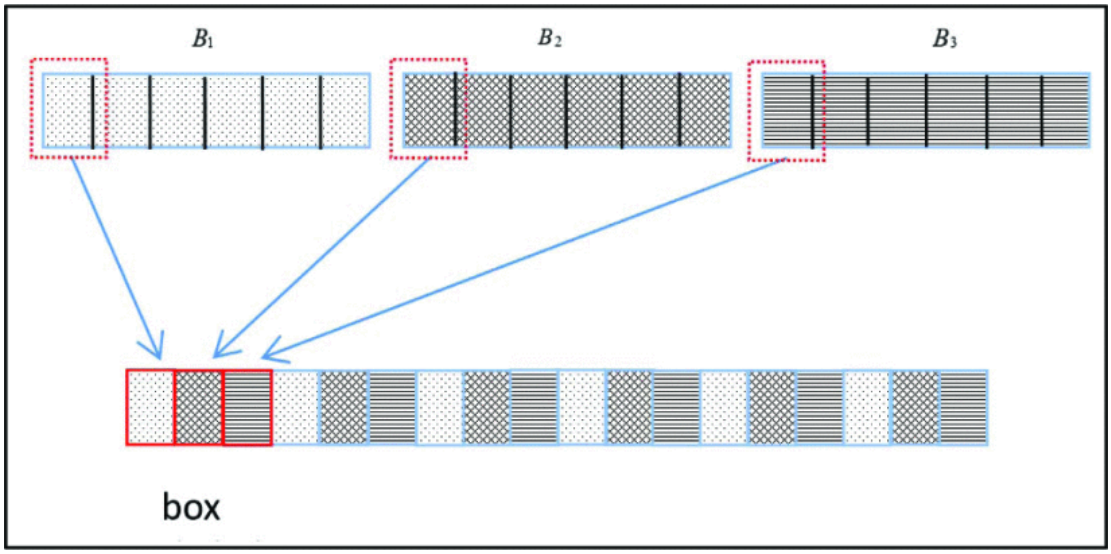
2.1 哈希表结构

哈希表包含 t 个子表，t 是偶数，并且所有的子表的大小相等，其中最后一个子表（第 t 个表）是链表。将每个第 2n+1 个和第 2n+2 个哈希表划分为一组（0≤n≤t/2－1），以获得相同大小的 t/2 组哈希表。每个子表包含 k 个 bucket，每个 bucket 可以存储一个键值对。具体结构见下图。



2.2 辅助表结构

辅助数据结构包含 t 个子表一一对应 t 个 Bloom Filter。所有 Bloom Filter 大小相等，每个 2n+1 个 Bloom Filter 与 2n+2 个（0≤n<t/2）相结合，得到 t/2 个大小相等的 Bloom Filter 并用 B1、B2、……、Bt/2 表示每组 Bloom Filters，然后将这些大小相等的滤波器叠加在一起，形成统一的 Multi-bit Bloom Filter（MB）。每个 Box 都包含所有组的 Bloom Filter。Bloom Filters 的组合是在物理片上存储器中执行的，但每组哈希子表的组合只是概念性的。每个子表都有一个相应的 Bitmap[16]，Bitmap 中的每个比特都对应于相应子表中的一个 bucket。空 bucket 对应位图中的位为 0，非空 bucket 则对应位图中位为 1。具体结构见下图。



3 操作流程分析

3.1 插入操作

当插入给定的键值对（key，value）时，首先使用主哈希函数计算键值以获得哈希值 hkey，并根据 hkey 是奇数还是偶数（总共 t/2）来确定候选哈希表的

奇偶属性，OE 哈希算法的名称由此而来。

然后通过位图判断 t/2 个哈希表中的候选桶是否为空，并将键值对插入到映射位置为空、加载率最小的哈希表中。如果同一属性的所有子表中都没有空的 bucket，则使用踢出机制来存储第一个踢出的值；如果踢出仍然失败，则执行盲踢（盲踢类似于踢出机制，并且使用相同的方法来找到第二个踢出值的候选桶）。如果盲踢达到上限阈值 θ，则将最后一个子表，即第 t 个子表链接到链表，并使用指针将键值对挂在链表上。假设要插入的子表的索引为 m（0≤m≤t－1），则更新 m 所在组的 Bloom Filter，并更新相应子表的 Bitmap。

每当插入键值对时，总是在满足插入条件的所有子表中选择负载率最小的子表，以平衡所有子表的负载因子。

```
for each (key, value)
  h_key = main_hash(key)
  use p to record the parity of h_key
  for i = p; i < t; i+= 2// traverse sub-tables
    if ∃sub_hash[i].value == 0//exist empty candidate bucket(s)
      insert value//insert into the table with the smallest load rate
    else kick sub_hash[i].value//perform kick operation
      if kick.succ == true//exist empty bucket(s)
        insert sub_hash[i].value// insert into the table with the smallest load rate
      else blind-kick// perform blind-kick operation
        if blind − kick.succ == true// blind-kick succeeded
          break
        else insert sub_hash[i].value into the last linked list
end for
```

3.2 查询操作

如果要查询给定键的值字段值，或者确定哈希表中是否存在键值对，可以通过查询操作进行查询。主要思路如下：

1)首先，在多位 Bloom Filter 中查询键值的返回值。

a)如果返回 i，则表示密钥组为 Bi，执行第二步。

b)如果返回 false，则表示哈希表中不存在该键。

2)通过主哈希函数计算奇数或偶数哈希子表中是否存在键值对。

3)确定返回的哈希子表的对应位图中是否存在键值对。

a)如果存在，则查找相应哈希子表的映射位置的键是否与其相同。

i)如果它们相同，则返回它们的值，查询结束。

ii)如果它们不相同，并且是最后一个哈希链表，请查询链表中的键值对。

b)如果它不存在，则表示它不存在于哈希子表中。

input key
if $MB \rightarrow query(key) == -1$// -1 means not found
  return false
else $B_i = MB \rightarrow query(key)$// record return subscript
  $h_{key} = main\_hash(key)$
  if $h_{key}\%2 == 1$//calculate sub-table parity
    $p = 2 * x$
  else $p = 2 * x + 1$
    if $sub\_hash[i] \rightarrow key == key$//indicates that the key is found
      return i//return to query result
    else query in the last hash list
end if

3.3 删除操作

首先查询哈希表中的特定值。如果找到的对应键的值与需要删除的键值对相同，则清空 bucket 内部，最终将对应位置的位图设置为零；如果值不相同，则表示删除失败。

4 理论分析

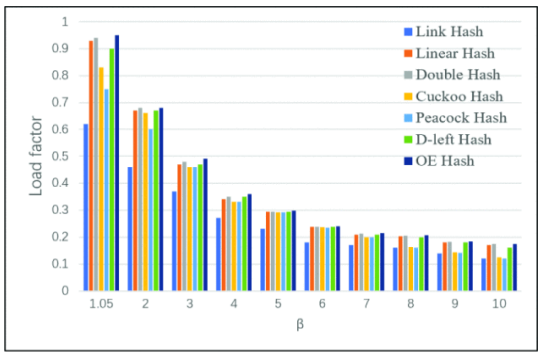
4.1 无限循环问题

当插入发生冲突时，冲突处理逻辑会首先寻找同一属性下空的子表，然后再实行盲踢机制，同时也设置了循环的阈值，因此 OE Hashing 完全避免了无限循环的问题。

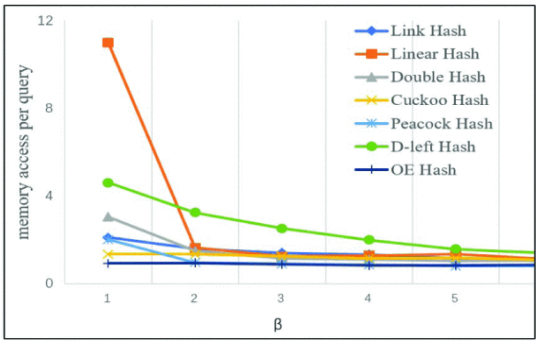
4.2 平衡负载问题

每当插入键值对时，总是在满足插入条件的所有子表中选择负载率最小的子表，以平衡所有子表的负载因子。

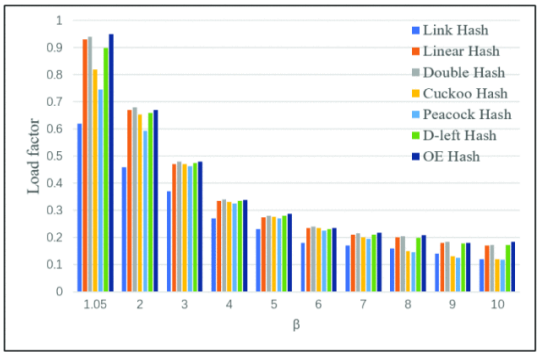
5 测试性能



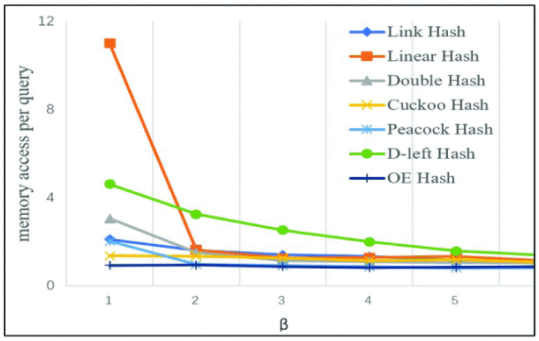
（a）dataset8



（a）dataset8



（b）dataset48



（b）dataset48

从上图可以看出，OE 哈希算法的负载因子最大。线性哈希和双哈希算法以更高的内存访问为代价，实现了相似的负载因子。当 ß 较大时，Peacock Hash 算法可以获得较高的负载因子。与 OE 哈希算法相比，Peacock 哈希算法需要更多的内存访问才能获得高负载因子。这表明，在相同的空间大小下，OE Hash 算法可以实现最大的负载因子，即可以容纳最多的键值对，并且具有最佳的性能。

从查询速度方面来看，最快的是 OE Hash 算法，其次是 Cuckoo Hash、Link Hash 和 Double Hash 算法。当 ß 很小时，Link Hash 算法的搜索时间很长。当 ß 大于 2 时，几乎所有哈希表的内存访问次数都在 4 以下，当\223 小于 2 时，线性哈希算法的访问次数最多，OE 哈希算法的最小。当 ß 为 1.05 时，D-left-Hash 算法的数量达到 5，OE 哈希算法的数量约为 1，是 D-left-Hash 算法数量的 4 倍。OE 哈希算法是除 Link Hash 算法外所有哈希表中访问内存次数最少的哈希表。这证明了 OE Hash 算法由于 Bloom Filters 的高负载因子和低误报率而实现了更快的查询速度。