# DCuckoo：基于片内摘要的高性能哈希表

蒋捷[1] 杨仝[1*] 张梦瑜[2] 代亚非[1] 黄亮[3] 郑廉清[4]

[1]北京大学信息科学技术学院，北京；[2]对外经济贸易大学信息学院，北京；[3]94782部队65分队；[4]西京学院控制工程系，西安

## 引言

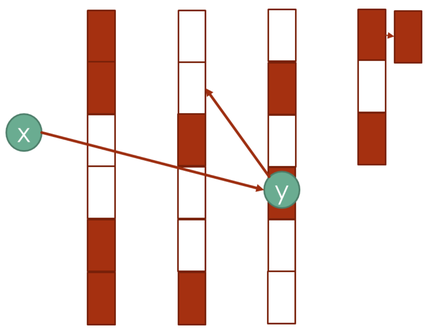
哈希表由于其支持高效的记录更新与检索操作，在计算机相关的各个领域中有着广泛的应用。但哈希表有两个明显的缺点：冲突和低效的内存利用。最小完美哈希使用 N 个位置存储 N 条记录，解决了冲突和空间效率的问题，但该算法不支持增量的更新。本文的目标是设计一种高效的哈希表，能够支持高速查询、最坏情况可以保证的高速更新、高效的空间使用、以及动态的容量改变。结合 Cuckoo 哈希和 d-left 哈希的实现，本文提出了一个新的哈希表设计方案——DCuckoo。DCuckoo 使用多级子表并应用了 Cuckoo 哈希中移动已有元素的机制以提高装载率，且只保留了最末级子表的指针以减少空间浪费。为了进一步优化查询性能，DCuckoo 在片内内存中使用指纹和位图作为摘要，在查询时先匹配指纹，以减少对片外内存的访问次数。本文对 DCuckoo 进行了一系列实验，与其它五种哈希表进行比较，发现 DCuckoo 达到了设计目标，并且在各项指标上均好于已有的哈希表设计。

## 相关工作：Cuckoo哈希与d-left哈希

Cuckoo哈希使用两个哈希函数$h_1(x)$和$h_2(x)$，将元素映射到两个不同的位置，即每个元素有两个候选位置。Cuckoo哈希的原理如下：插入某元素x时，若两个候选位置中至少有一个为空，则将元素直接插入；若两个位置都被其他元素占据，则任意选择一个元素将其“踢出”（kick），并插入他不属于x，被踢走的元素y则需要去查看它的另一个候选位置，若该位置为空，则直接插入，否则继续将占据这个位置的元素踢走，将y插入，重复这一踢的过程，直到所有的元素都被插入到表中，或者踢的次数达到一定的上限（如500次），这也是该算法被命名为布谷鸟（Cuckoo）的原因。通过这种方式，哈希表中的元素位置不断的被调整到合适的空位，因此可以实现较高的装载率（>95%）。且其查询十分简单：最多只需要探测两个位置。Cuckoo 哈希的不足之处在于：首先更新低效且可能产生更新失败；其次尽管Cuckoo哈希最坏情况只需要两次记录探测，但平均情况下也需要1.5次，而理想的查询访存次数应当为1，也即 Cuckoo 哈希的平均查询性能较差。

与Cuckoo哈希不同，d-left哈希使用d个哈希子表，而非一个完整的哈希表，每一个子表都有一个与之对应的哈希函数$h_i(x)$。所以任意元素在每一个子表中都有一个候选位置，且由于哈希函数不同，元素在每个子表中的位置往往不同。因此当两个元素在某一个子表中冲突时，在很大概率下这两个元素不会在其他子表中冲突，因此d-left哈希同样可以实现较高的装载率（>90%）。该算法中每一个子表都是一个标准的链式哈希表，用于解决多级子表不能处理的冲突。当插入一个元素x时，算法从左到右依次检查每一级子表，若$h_i(x)$的位置为空，则将x直接插入到这个子表中；若所有的候选位置都被其他元素所占据，则选择一个链表长度最短（负载最小）的桶插入。而查询时，算法依然从左到右依次探测每一个候选位置，直到找到合适的元素。无论是在插入还是查询，算法总是从左到右依次探测各级子表，虽然这一策略导致各个子表的装载率并不相同，但由于元素总是优先被插入到最左边，所以实际上查询时从左到右的顺序可以减少所需记录探测的数量。该算法通过多候选位置降低了冲突的概率，但缺点在于每次查询也需要探测多个候选位置，造成查询性能的底下。此外，d-left的每一个桶中都包含一个链表，而在实际的应用中，这些指针往往为空缺也需要占据一定的内存空间，造成了空间浪费。

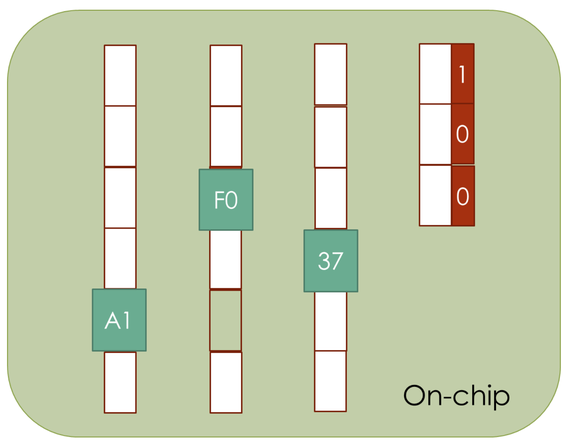
## DCuckoo算法描述

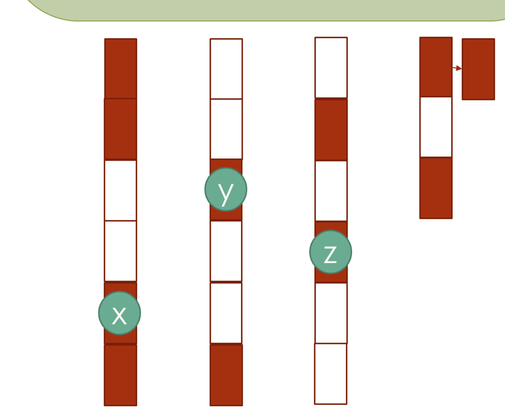


DCuckoo原理图

如左图所示，DCuckoo的表结构与d-left哈希类似，由d个哈希子表组成，元素在每一级子表中都有一个候选位置。为了提高内存利用率，DCuckoo只保留最末级子表的链表，并将该级子表的长度减半，从而减少了指针的空间浪费。在构建哈希表和插入元素的过程中，DCuckoo应用Cuckoo哈希移动已有元素的机制：插入元素x时，若x在d个候选位置中没有找到空位，首先查看其他元素是否可以通过一次移动找到合适的位置，若有则移动该元素；若没有，则随机地从这d个候选位置中选出一个，用x将原先位于这个位置的元素y置换出，再对元素y进行插入操作。若在进行了θ次随机踢操作之后仍无法为元素找到合适的位置，则将目前未被插入的元素插入到对应的末级链表之中。通过这种方式，DCuckoo实现了较高的装载率（>95%）。

**片内摘要：**一般的哈希表规模较大，只能存储在片外内存（off-chip memory）中，相对于哈希值的计算，对内存的访问需要消耗更多的CPU指令周期。以d-left为代表的多选择哈希通过多候选位置的方式提高了哈希表的装载率，同时带来的问题是查询时对片外访问的需求增加。以d-left为基础的DCuckoo同样存在这个问题。因此，我们在DCuckoo中引入片内摘要以加快查询速度。片内内存（on-chip memory）具有容量小、速度快的特点。由于容量小，片内内存并不能容纳完整的哈希表，但可以存储哈希表的摘要（summary），为片外内存中哈希表的搜索提供一些指引，如“元素可能在哪些位置出现”这一类信息。



DCuckoo原理图

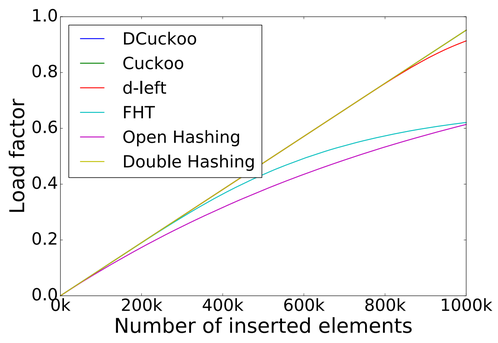
**指纹镜像：**DCuckoo使用指纹（fingerprint）和位图（bitmap）作为片内摘要，以提高哈希表的性能。一个指纹对应由连续r位（bit）组成的一段内存空间，计算一个关键字的指纹时相当于对这个关键字进行了一次哈希操作，将其映射到$[0, 2^r)$这个空间中。因此指纹是对原关键字的一个摘要，它用很小的空间记录该关键字的信息。指纹可能会产生冲突，即一个指纹可能会对应多个关键字。DCuckoo为片外的所有子表建立一个指纹镜像，即片外的每一个桶都对应片内一个指纹。在查询一个关键字时，先检查片内的指纹镜像中对应的指纹是否与该关键字的指纹相匹配，若匹配，再到片外的子表中去检索该关键字，从而大幅度减少了查询操作所需要的访存次数。由于指纹可能会产生冲突，指纹匹配并不一定代表片外对应位置的桶中存储着与该关键字匹配的元素，也即指纹的方式也具有一定的假阳性，与指纹的长度有关。

**位图镜像：**由于最后一级子表中可能存在着链表，而该子表中对应的指纹镜像只能提供位于桶中元素的信息，所以在查询时，如果在指纹报告元素存在的位置没有找到对应元素，则一定需要检查最后一级子表对应的链表，因为该链表的信息没有在摘要中表示。当查询操作所查询的元素不在哈希表中时，这会造成大量额外的访存。对于这一问题，DCuckoo针对最后一级子表对应的指纹镜像做一些额外的处理：这些指纹比标准的指纹多一位，用来表示片外子表中是否包含链表。在检索时，若这一位为0，则无需对链表进行额外的探测。

**子表动态调整：**在很多应用中，需要插入到哈希表中的元素集合可能会发生很大的变化，这时最开始的哈希表可能就会显得过大或过小。处理这一问题最简单的方法是重建整个表，很多哈希表使用这一方法进行重构，如Cuckoo哈希、开散列哈希表等。整体重构的处理方式缺点在于需要消耗大量的计算资源与内存空间，因为重构需要消耗大量的时间，而在这过程中系统应避免阻塞在重构过程中，所以重构的实现往往是在新的地址空间创建一个新的更大或更小的哈希表，此时旧的哈希表继续支持查询操作，当新表建立完毕后，再将旧表抛弃。得益于多级哈希表的结构，在DCuckoo中可以很容易地实现增加或者删除一个子表，避免因为链表长度过长或者装载率过高造成的潜在的性能下降问题。具体操作如下；当链表元素总数超过某一阈值或者整体哈希表的装载率超过某一阈值时，增加一级哈希子表，并将之前最后一级子表中链表上的元素进行重新插入操作；当哈希表的整体装载率低于某一阈值时，为了避免空闲的桶造成的空间浪费，移除其中的某一级子表，并将该子表上的元素进行一次重新插入操作。因此DCuckoo可以很方便地支持哈希表的重构操作。

## 实验结果与分析

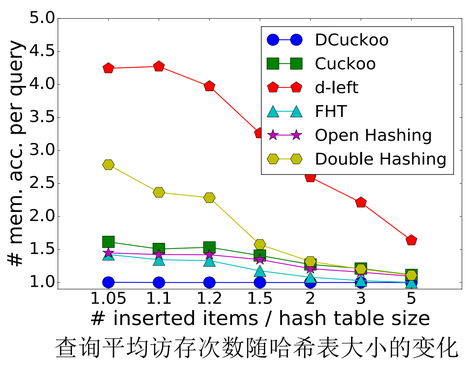
实验中采用随机生成的数据集，数据关键字（key）的长度为8字节，值（value）类型为32位整形（int），插入数据集的规模为1M (106个元素)。查询数据集符合Zipf分布（在键值存储的实际应用场景中，查询请求大多符合Zipf分布），规模为10M，Zipf分布偏度（skewness）为0.99，与数据库测试中经常使用的YCSB相同。哈希表设置方面，DCuckoo实现使用8级哈希子表，最后一级子表的长度为普通子表长度的一半，指纹长度为15位；在插入过程中如果发生冲突，只允许一次移动元素而不允许盲踢操作（θ=0）。为了方便比较，本文另外实现了其他五种哈希算法：开散列法（open hashing）、双散列法（double hashing）、Cuckoo哈希、d-left哈希、FHT，其中双散列和Cuckoo哈希另外使用了一个大小与原始哈希表大小相同的链式哈希表用于解决冲突；双散列最大探测长度为16，Cuckoo哈希踢操作上限为500次。d-left哈希使用8级子表，FHT使用4个哈希函数。实验结果表明DCuckoo在装载率、查询效率等方面均优于已有的哈希表设计，且在不同数据集下均有良好的表现。具体结果如下所示。



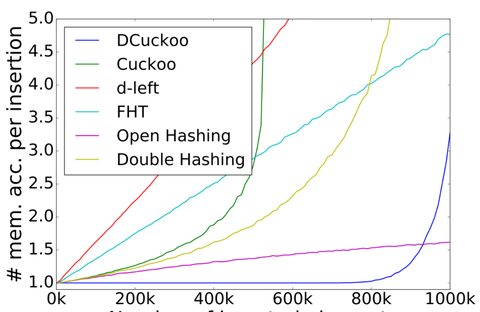
装载率随插入元素数量的变化

**装载率：**在这一实验中对所有的六种哈希表使用相同的数据集，所有哈希表大小均为插入数据集规模的1.05倍，即每个哈希表包含1.05M个桶，每插入10000个元素后记录此时哈希表的装载率，结果如图所示。可以看到，DCuckoo、双散列法、Cuckoo都达到了非常理想的装载率，分别为95.17%，95.20%，95.18%，表明几乎所有的元素都在哈希表中（满装载率约为100/105≈95.24%）。但双散列法和Cuckoo哈希都使用了额外的链式哈希表用于解决冲突。装载率高意味着使用相同的桶可以容纳更多的元素，因此在插入相同数量的元素的情况下，装载率高的哈希表可以预先分配更少的内存空间。

**平均查询访存：**右图表示了将同样的元素插入到不同规模的哈希表之后，哈希表的查询性能。横坐标代表哈希表中桶的总数与插入元素数目的比值，纵坐标表示哈希表构造完毕后，一次查询所需的访存次数。如前文所述，相对于哈希值的计算，对内存的访问需要消耗更多的CPU指令周期，因此访问内存的次数可以模拟哈希表的查询性能。可以看到DCuckoo在所有实验条件下都能达到接近1的平均查询访存次数，而其他哈希表只有在哈希表足够大的情况下才能达到这一水平。



查询平均访存次数随哈希表大小的变化



插入平均访存次数随插入元素数量的变化

**平均插入访存：**插入性能的实验过程同装载率测试，即在构建哈希表的不同阶段进行测试。可以看到随着哈希表逐渐变满各个哈希表的插入性能都有所下降，开散列法的插入访存相对稳定，这是因为冲突时只需要将新建的元素放到哈希表表头即可。DCuckoo在插入前900k个元素时性能最优，尽管在最后有所下降，但扩充子表的方式可以保证最坏情况下可控。而Cuckoo哈希由于最多可能执行500次踢操作，插入性能较差。当不需要检索重复插入时，开散列法只需要将元素插入到链表的头部，因此性能相对稳定。插入平均访存次数随插入元素数量的变化如左图所示。

| Dataset | Distribution | Avg. mem acc ( no blind kick ) | Avg. mem acc ( 6 blind kicks ) |
|---|---|---|---|
| 1 | Zipf, 0.99, out-of-order | 1.002727 | 1.000003 |
| 2 | Zipf, 0.99, in order | 1.000052 | 1.000003 |
| 3 | Zipf, 0.99, reversed order | 1.091544 | 1.000004 |
| 4 | Zipf, 0.50, out-of-order | 1.000618 | 1.000015 |
| 5 | Uniform | 1.000693 | 1.000010 |

**不同数据集下DCuckoo的表现：**当DCuckoo不允许盲踢时，待插入元素若不能找到合适的位置，则会被直接放入链表，因此后插入的元素更有可能被放入到链表中。当查询数据集不是均匀分布时，插入的顺序可能会对查询性能产生影响，实验结果上表所示。其中顺序插入（in order）指按查询数据集元素出现频数从高到低插入，逆序插入（reversed order）指按频数从低到高插入，乱序插入（out-of-order）即插入顺序与频数无关。可以看出，当不允许盲踢时，查询性能受插入顺序影响较大；而当允许盲踢6次时，平均查询访存则与插入顺序无关，且都达到了理想的水平（≈1次）。这是由于一方面盲踢减少了在链表上的元素数目，另一方面后插入元素不再直接插入链表，即链表上的元素与插入顺序无关。这一实验结果表明的另一个问题是，出现在链表中的高频元素会影响查询性能，因此进一步的优化可牺牲一定的空间为元素标记“热度”，定期将“热度”较高的元素从链表中移出，以提高查询性能。