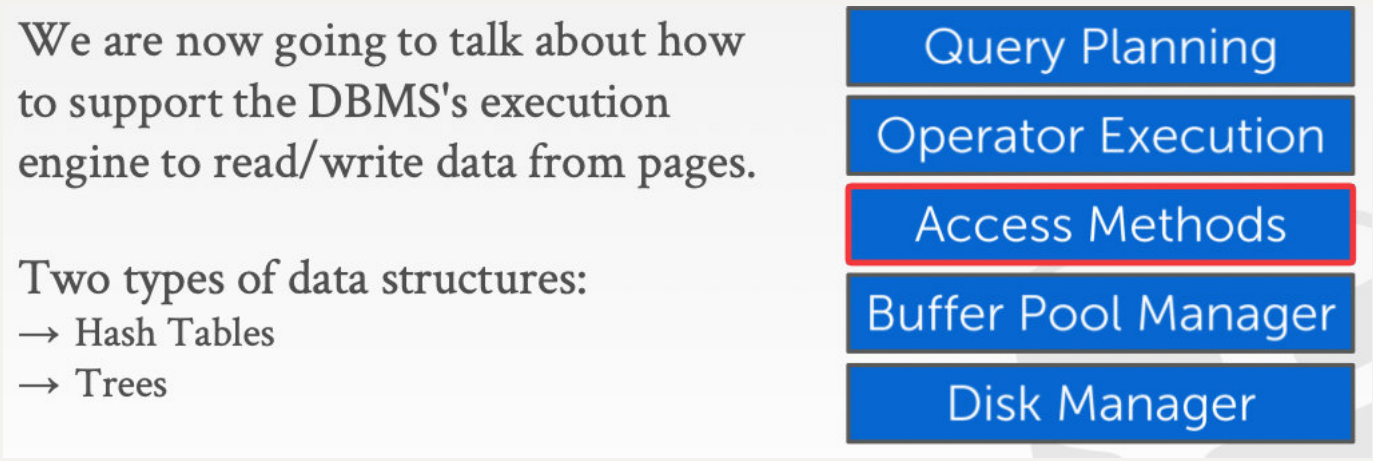
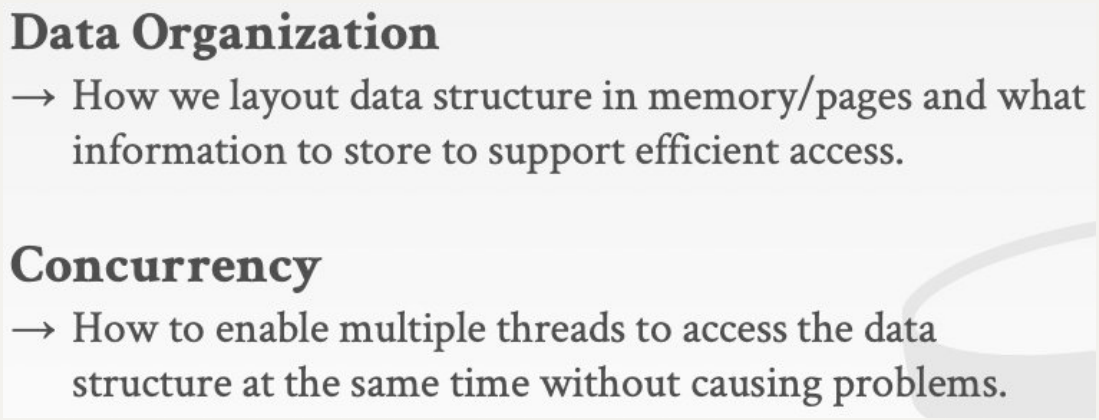
本Lecture和后面的B+树讨论的都是DBMS的access methods，我们需要特定的数据结构来帮助我们使用一种类似索引的方法从而帮助执行器找到所需要的文件页

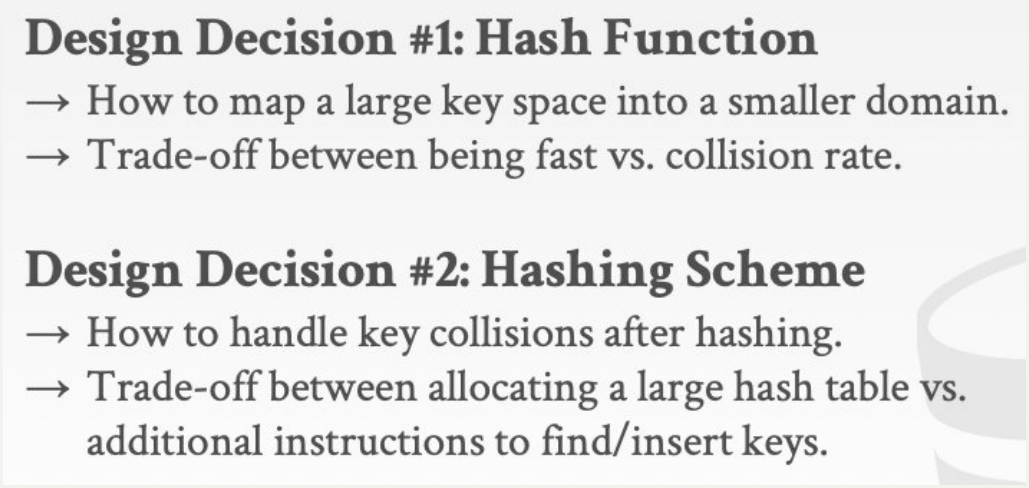


对于哈希表的设计，有如下两方面的考虑



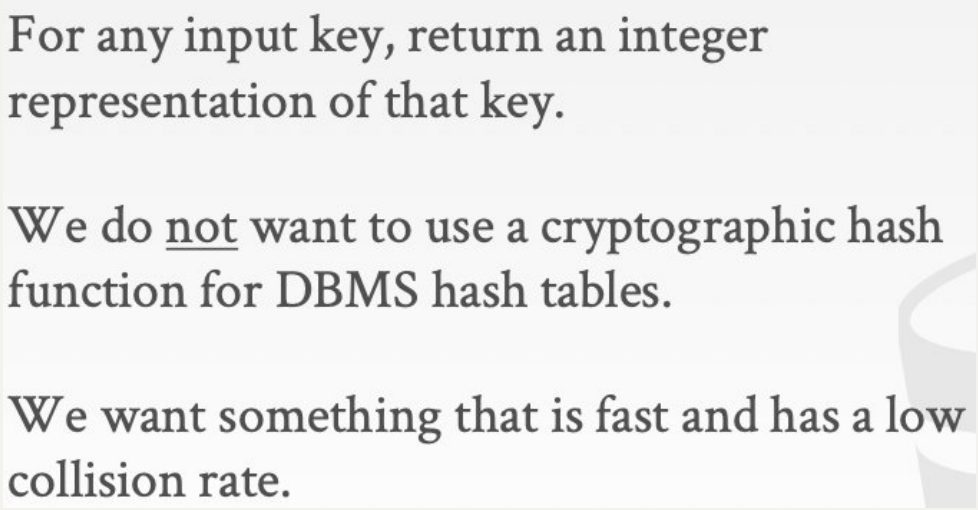
- 哈希表的数据结构具体应该是什么样的
- 如何让哈希表支持多线程并发的读写

哈希函数的速度和哈希冲突的概率往往是正相关的，因此在设计时会有一些取舍，应对哈希冲突的策略也有很多种



Hash Functions

DBMS的哈希表不会使用加密用的哈希函数（像SHA256这种加密用的哈希函数是无法使用哈希值反向推算出输入的值的，比如说用mod 100的运算做哈希函数，那通过哈希值8就能反推出输入值有可能是108，这种哈希函数就不可以用于加密），因为加密用的哈希函数时间复杂度很高，会带来巨大开销，我们DBMS的哈希函数的期望就是，可以被快速完成，并且不容易发生哈希冲突

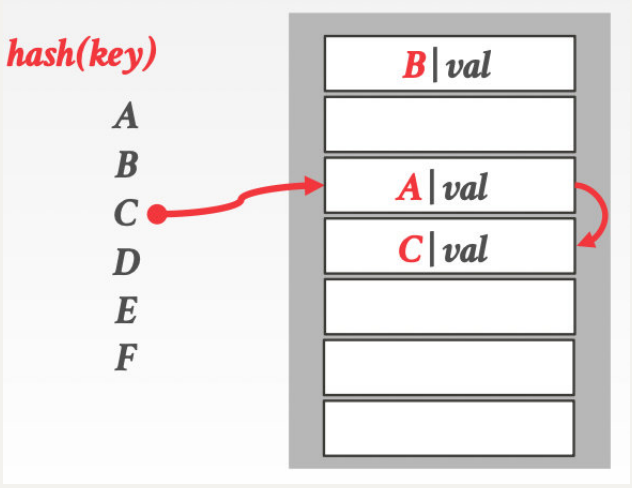


Static Hashing Schemes

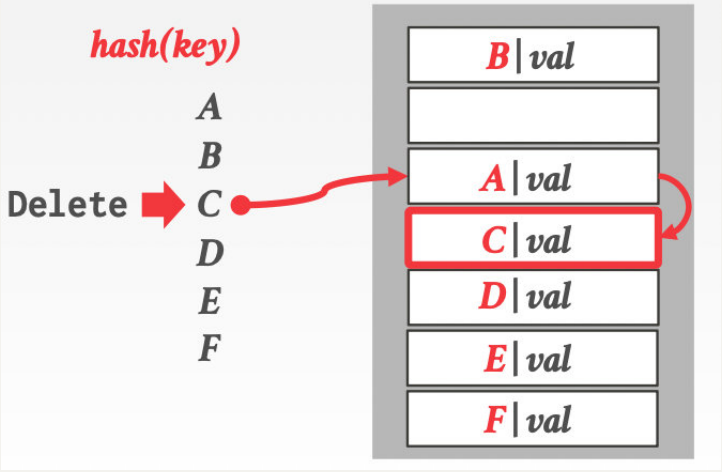
应对哈希冲突的静态策略

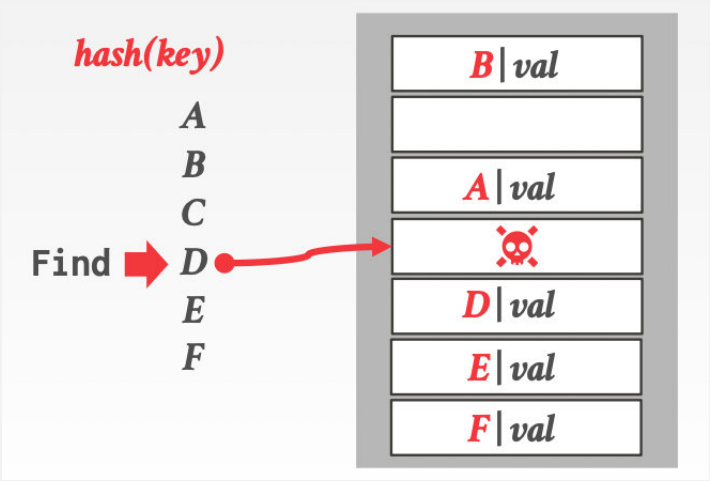
Linear Probe Hashing

翻译成中文是线性探测哈希，又名开放地址哈希，这种应对哈希冲突的方式是使用一个很大的数组，里面有很多哈希槽，在插入数据时，如果出现了哈希冲突，那么就把我们想存储的Key-Value存放到发生冲突的槽的下一个槽里



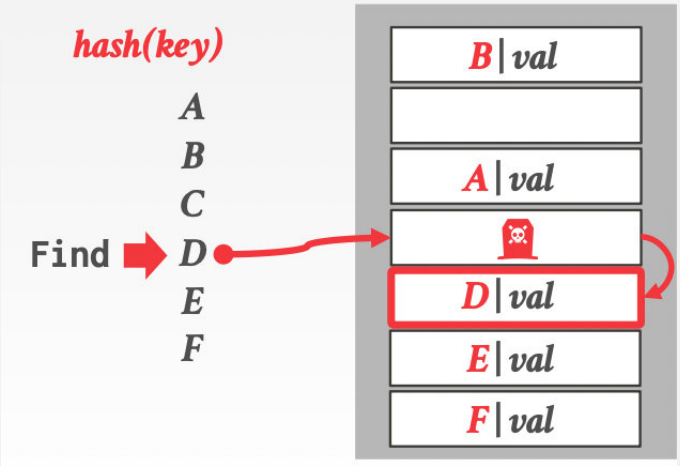
在删除数据时，不能简单的找到要删除的KV然后直接将其从所在哈希槽处抹去，这样可能会导致后续的查询发生错误（具体的场景还需参考slides）



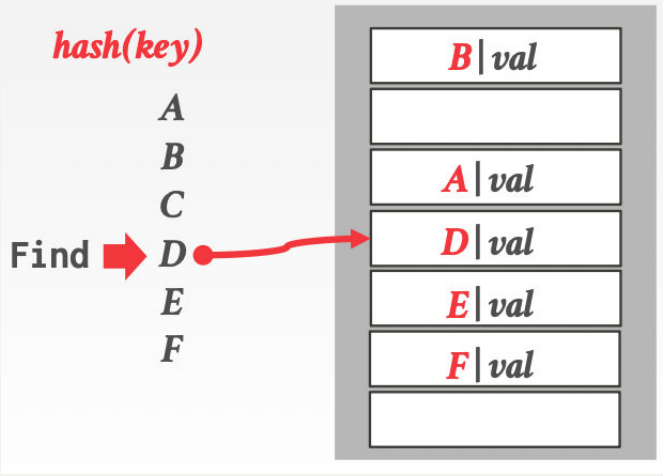


有如下几种解决问题的方法：

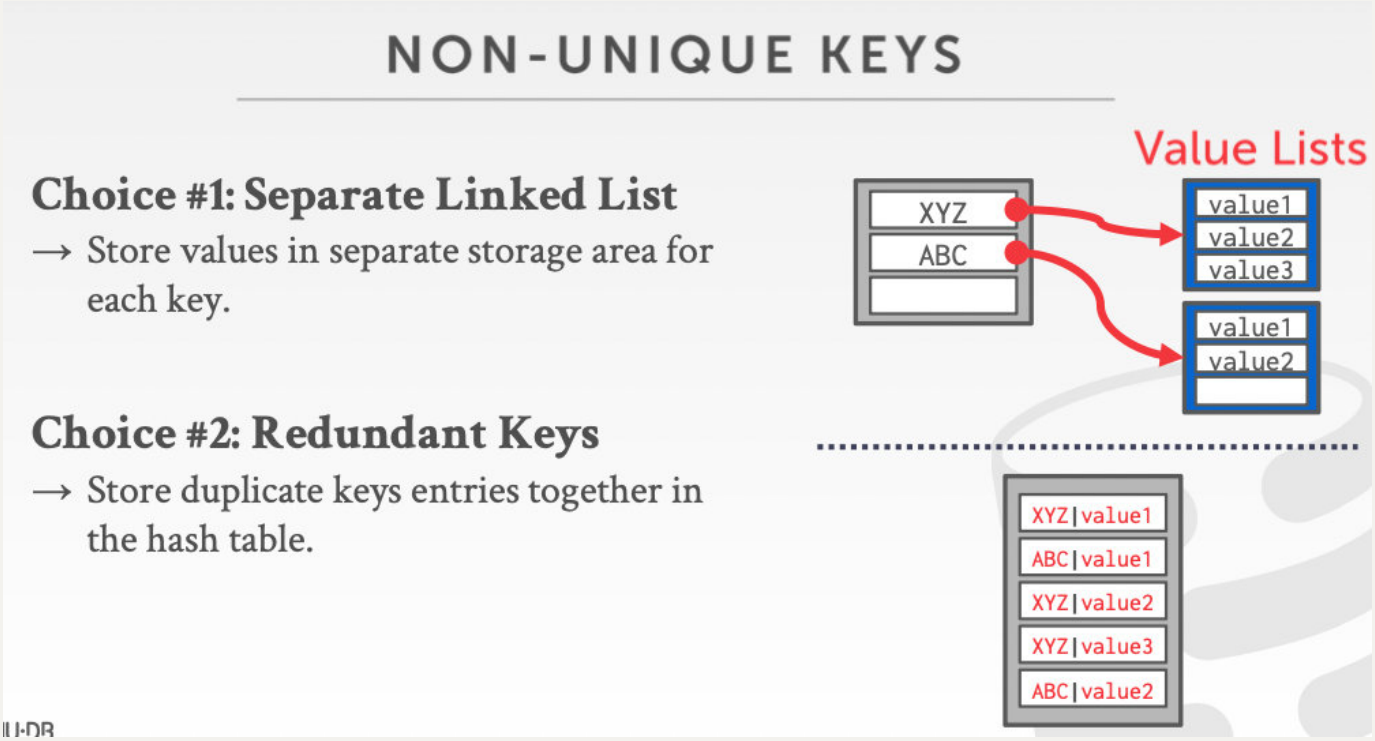
1. Tombstone，“墓碑”，在想要删除的哈希槽处做一个标记，这样的话后续的查询就可以看到这个标记，知道这里原本有一个KV，后来被删了，进而去下一个哈希槽检索，而不是直接返回“找不到数据”



2. Movement，在删掉某个哈希槽的KV后，对余下的哈希槽做一遍整理，这样就可以让下图的`D|val`这条KV数据放置在正确的哈希槽位里，后续的查询就不会有问题，在整理的过程中要做很多其他工作，因为在整理之前有些KV数据是在正确的槽位里的，所以在整理的过程中还要确想办法让我们的整理没有影响到它们（具体场景下的行为参考slides）



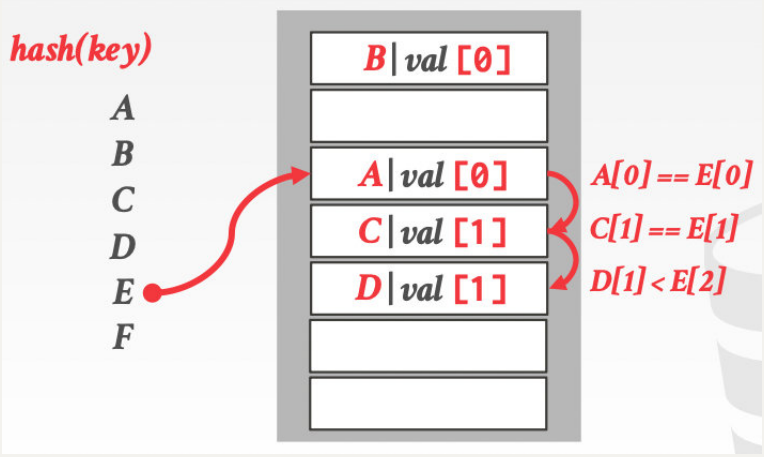
如果我们遇到了非唯一键的情况，即有多条KV的K相同这样的情况，这种情况在数据库的索引中有可能发生，有如下的解决方法（choice2是把key和value和在一起作为新的键，这个新的键一定是unique的）：

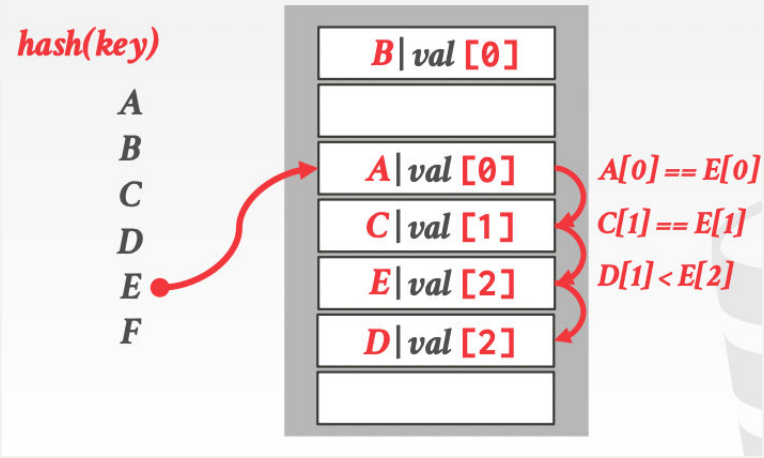


Robin Hood Hashing

这个应对哈希冲突的策略属于在开放地址哈希基础上的改进，在哈希槽位中除了要存储KV，还要记录“这个槽块被推过几格”，即“相对于正确的存放位置的偏移量”（后文简称偏移量）这样一个后缀，并且在发生哈希冲突时不断和被遍历到的槽位比较后缀的大小，如果被遍历到的槽位的后缀更小，那就占据这个槽位，把被遍历到的槽位往后推（具体的完整步骤还需查看slides），这样做比较符合Robin Hood“劫富济贫”的思想，不会让某个槽位的后缀值，即偏移量太大，在该哈希策略的执行过程当中，会将开放寻址时被遍历到的槽块的后

缀值平均化，不使它们之间的方差太大

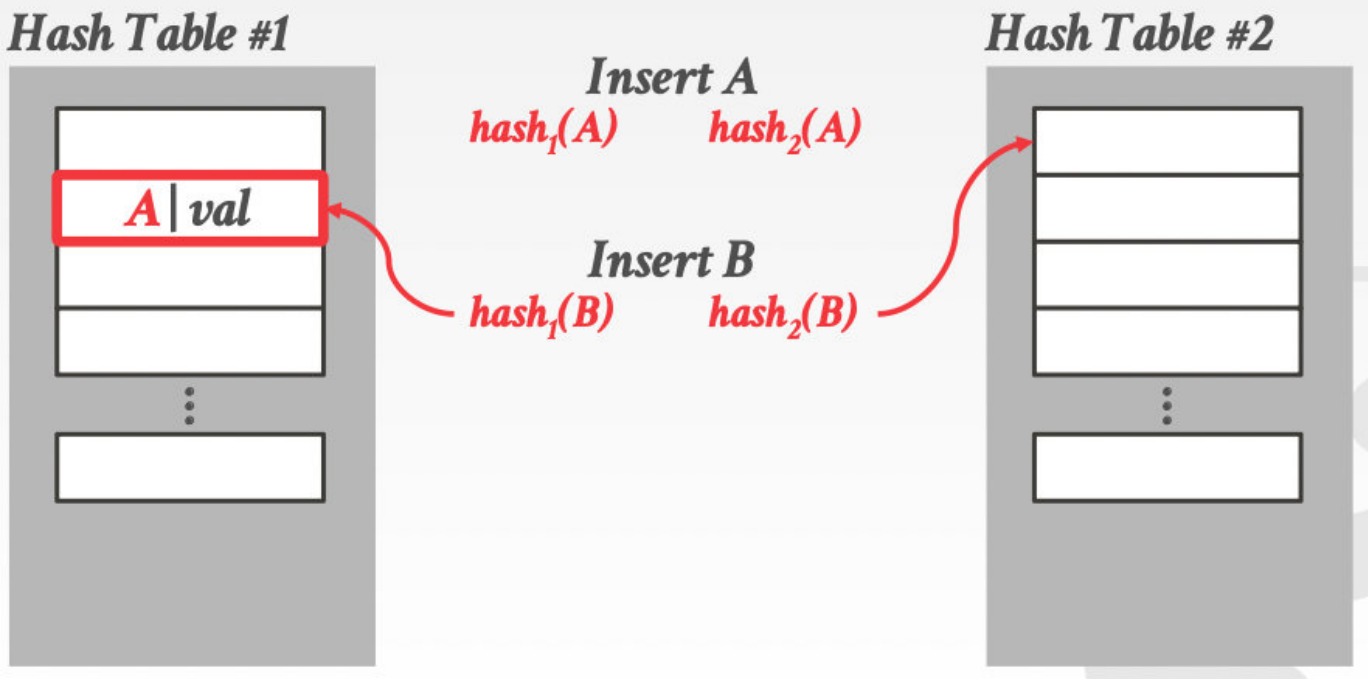




Cuckoo Hashing

cuckoo是杜鹃鸟，它总喜欢把蛋下在别人的窝里，它的这个习惯也和这个策略有相通之处

在这个策略中，我们可以有多个哈希表以及配套的哈希函数，当我们想插入一对KV时，可以把K送给前面所说的多个哈希函数分别计算，如果发现在我们所拥有的多个哈希表中，存在一个哈希表，如果插入这对KV，在其中不会发生哈希冲突，那就将这对KV插入该哈希表，要是所有的哈希表里都会发生哈希冲突，那么就会执行类似于匈牙利算法的一系列操作（具体参考slides），直到没有哈希冲突为止



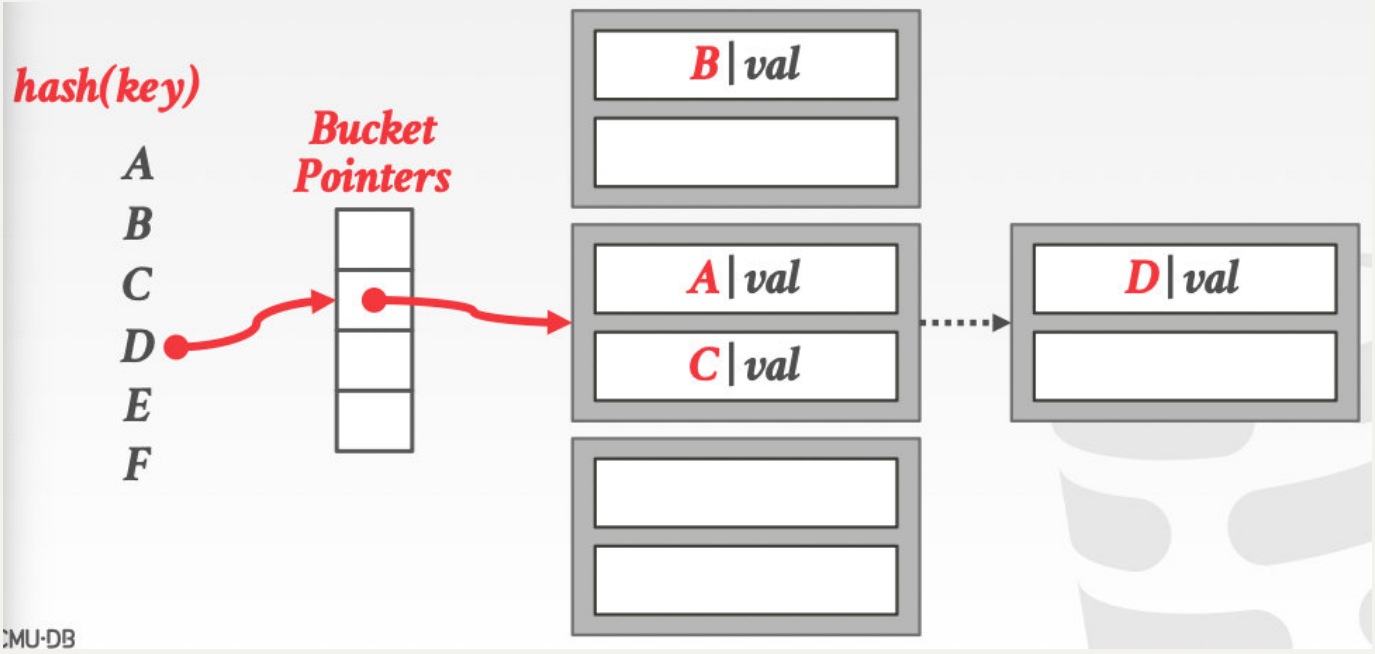
这些静态哈希策略有一些问题，最严重的就是，这种策略下能存储的KV（键值对）的数量有限，因为哈希槽的数量固定，这迫使DBMS必须提前知道未来最多能存多少数据，不能依据实际的负载去伸缩，因此便有了下文介绍的可以根据实际需求去动态伸缩的动态哈希结构

Dynamic Hashing Schemes

应对哈希冲突的动态策略

Chained Hashing

这就是指传统的拉链法解决哈希冲突，设有多个哈希桶，如果哈希桶被放满了，那就在满了的桶后面再增加一个桶，用链表形式连接起来

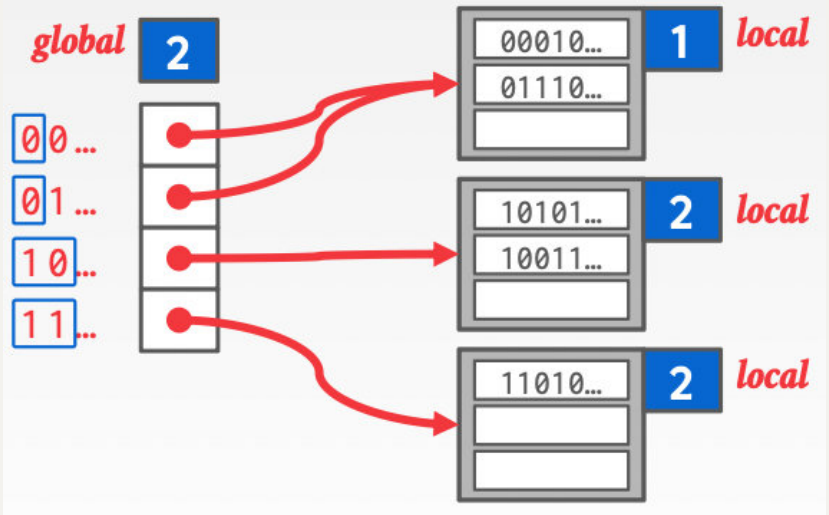


Go语言中的哈希表也是基于此方法来实现的，Go的`map`的每个哈希槽指向的第一个哈希桶中有八个空位，是通过一个结构体来实现的，结构体中有一个成员是拥有八个空位的数组，如果这八个空位被用完了，就会将这个桶链接到一个溢出桶

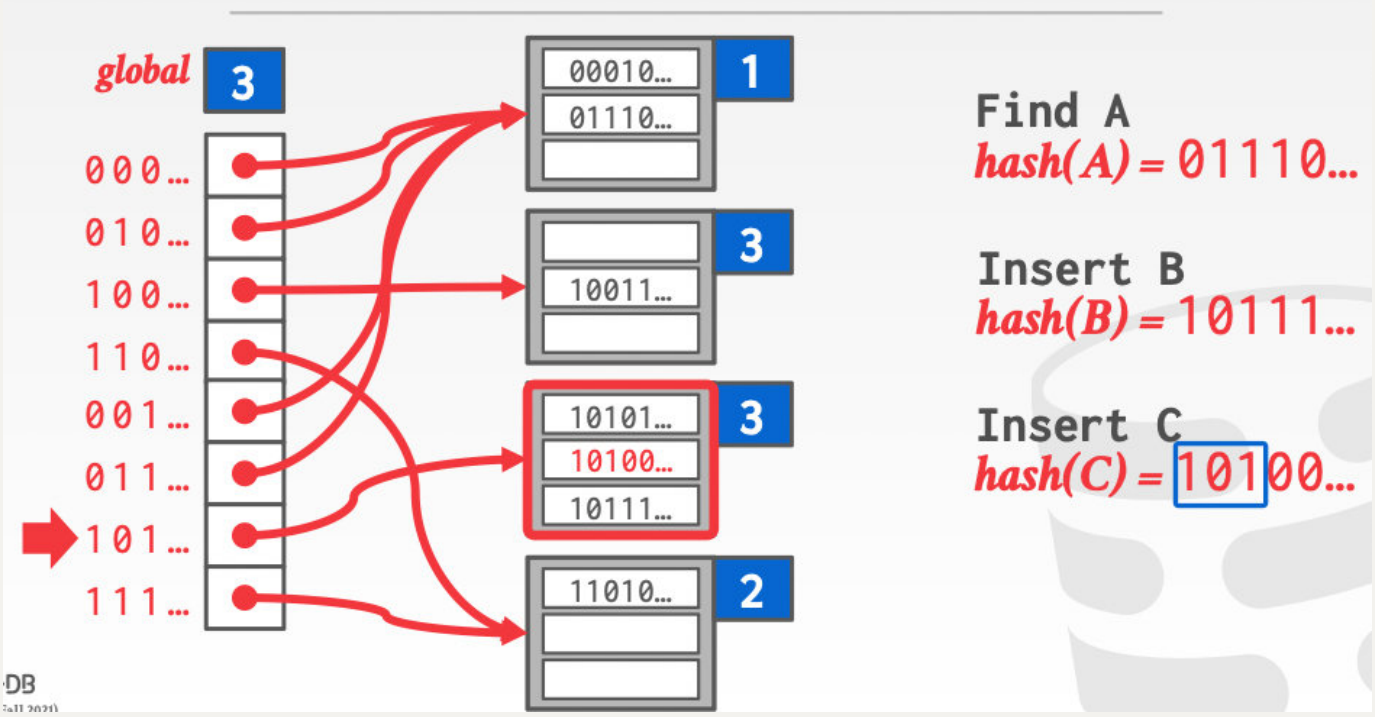
Extendible Hashing

这是指可扩展哈希，拉链式的哈希有一些问题，哈希槽的数目是固定的，发生的哈希冲突越多，哈希桶就越往链表退化（虽然Java的`map`会在退化到一定程度的时候把这个链表转为易于查找删除的红黑树），效率下降

实现可拓展哈希的方式如下图，哈希函数的返回值一般是整数类型，如果我们只看这个整数返回值的二进制编码的前两位（右侧最上面的哈希桶只看二进制编码的第一位，00和01编码的哈希槽都对应着它），也可以实现一个哈希表，



当这个哈希表的某个哈希桶被装满时，我们先不链接溢出桶，可以把原来的“只看哈希函数返回值前两位”这个条件改成“只看哈希函数返回值的前三位”，并且调整哈希桶的个数以及哈希桶和哈希槽之间的映射关系，整理每个哈希桶内的数据，就可以避免哈希表向链表退化，如下所示（这个较为复杂，具体参考slides）

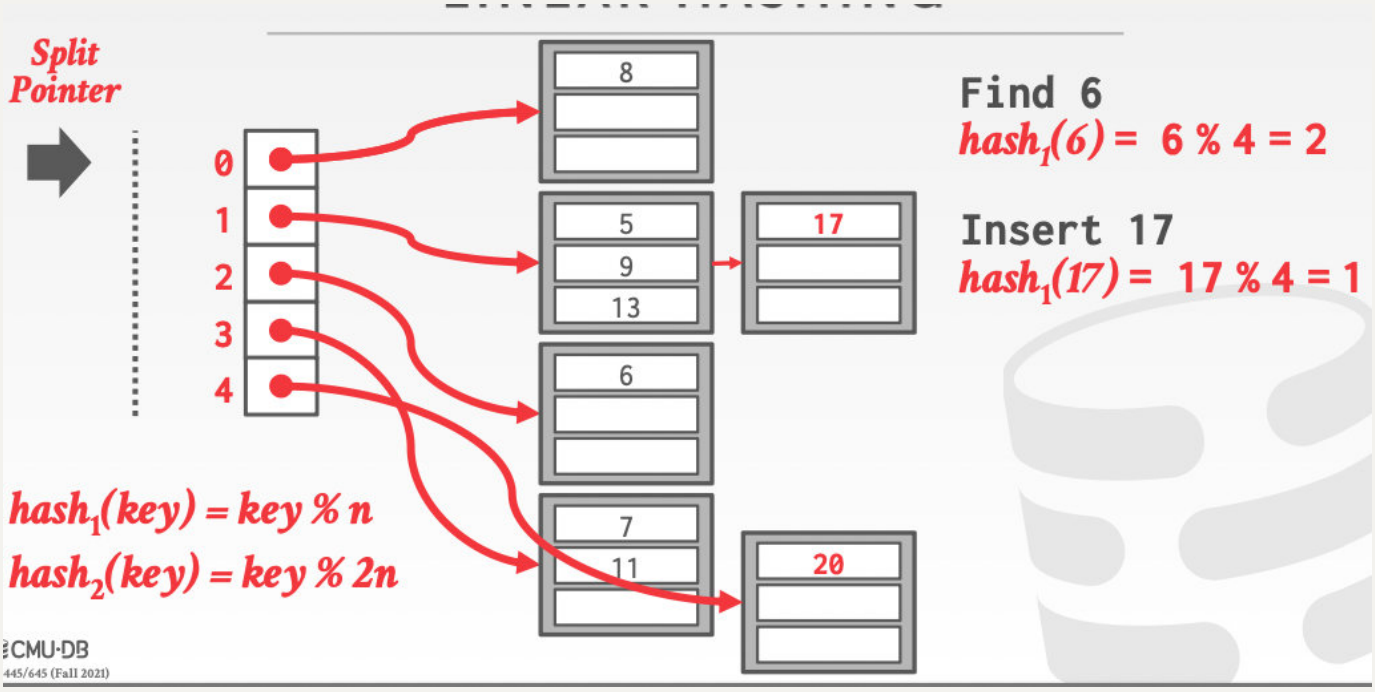


除了前面的红黑树，Java和Go中也会采用这种给哈希槽扩容的方式解决哈希冲突致使哈希表退化的问题，如果哈希表的负载系数达到阈值，就会自动触发这种扩容

Linear Hashing

extendible hashing的思路是直接对哈希槽做扩容，然后对哈希桶做整理，但这样做一次就会导致很大的开销，如果服务器处理用户请求时刚好赶上了这种扩容，那用户要等待很长时间，这个linear hashing的思路就是不进行一次性的直接扩容，而是线性地一点一点地扩容，这样就不至于出现前面的情况

这个策略的过程很复杂，还是要结合slides具体分析，在slides中可以看到，第一次发生哈希冲突后还是外接了溢出桶，但也触发了扩容，新增了一个哈希槽，并且虽然是1号哈希槽出现了溢出，但却新增了0号哈希槽的“分家槽”：4号槽，如果hash1(x)=0，还需要计算hash2(x)是0还是4，从而决定这个Key是对应到0号槽还是4号槽，基于这个“分家”的策略，就会把0号槽中原有的一些数据转移到4号槽



上文的这种“分家”的操作，是从小序号的哈希槽开始执行的，正如上图的split pointer最开始指向0号哈希槽，因而0号哈希槽最先开始“分家”，每执行完“分家”操作一次，split pointer指向下一个哈希槽，然后下一次扩容操作又被触发时，就又对此时的split pointer指向的哈希槽执行同样的“分家”操作，这样的每次往下移动一格的行为是线性变化的，因此该策略得名线性哈希

最终，当split pointer使得最初的所有哈希槽都被“分家”了之后，那这也就完成了Extendible Hashing想要做的那种扩容，split pointer会移回最初的起点0号槽，进行下一轮循环，hash1函数不再被使用，hash2函数作为唯一的哈希函数存在，就和直接从4个哈希槽扩容成了8个哈希槽一样

总结一下，哈希表是可以在理想情况下完成O(1)复杂度查询的数据结构，这是它在查询这方面巨大的优点，其他的数据结构很难企及，因此它在DBMS中应用广泛，并且哈希表的设计要在速度和灵活性之间做trade-off

但哈希表对于范围查询（比如说查id=3~10000）没有任何优势，因为原始Key连续的一组KV数据经过哈希运算之后，在哈希桶/哈希槽中的存放就不在连续，这也是“散列表”为何得名，因此往往不用于实现DBMS的索引，这个会由B+ Tree实现

Reference:

https://www.bilibili.com/video/BV14q4y1B7op/?spm_id_from=333.788