Carnegie Mellon University

Hash Tables

Lecture #06

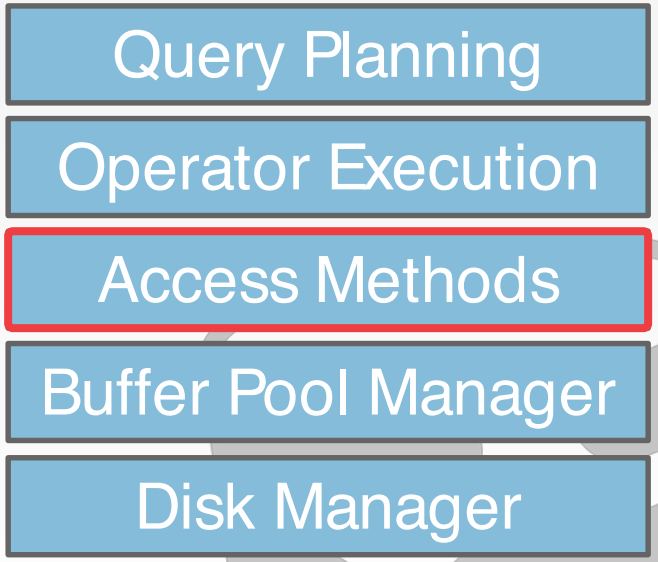
Database Systems
15-445/15-645
Fall 2018

AP
Andy Pavlo
Computer Science
Carnegie Mellon Univ.

COURSE STATUS

We are now going to talk about how to support the DBMS's execution engine to read/write data from pages.

Two types of data structures:
→ Hash Tables
→ Trees



DATA STRUCTURES

Internal Meta-data

Buffer Pool: build a hash table for the page table to map page IDs to frames in the buffer pool

Core Data Storage

Temporary Data Structures

build a hash table on the fly while hash join

Table Indexes

DESIGN DECISIONS

Data Organization

→ How we layout data structure in memory/pages and what information to store to support efficient access.

Concurrency

→ How to enable multiple threads to access the data structure at the same time without causing problems.

HASH TABLES

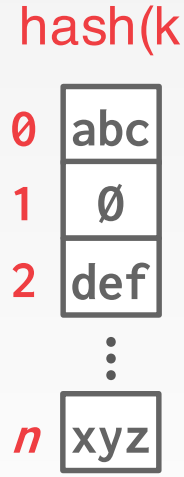
A hash table implements an associative array abstract data type that maps keys to values.

It uses a hash function to compute an offset into the array, from which the desired value can be found.

STATIC HASH TABLE

Allocate a giant array that has one slot for every element that you need to record.

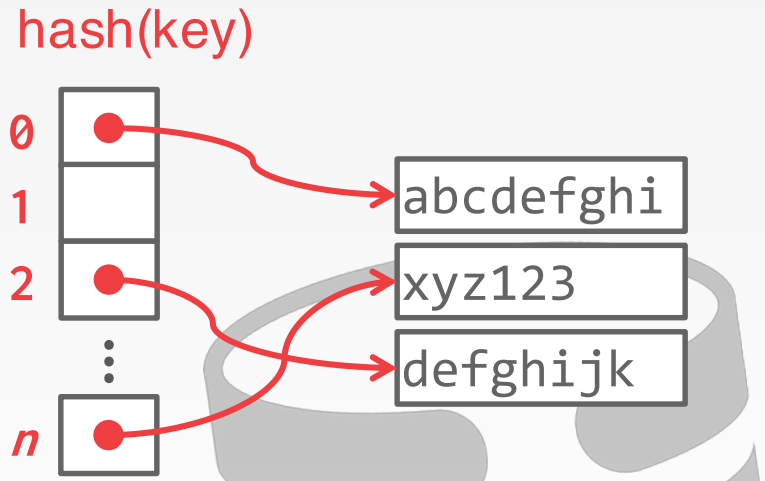
To find an entry, mod the key by the number of elements to find the offset in the array.



STATIC HASH TABLE

Allocate a giant array that has one slot for every element that you need to record.

To find an entry, mod the key by the number of elements to find the offset in the array.

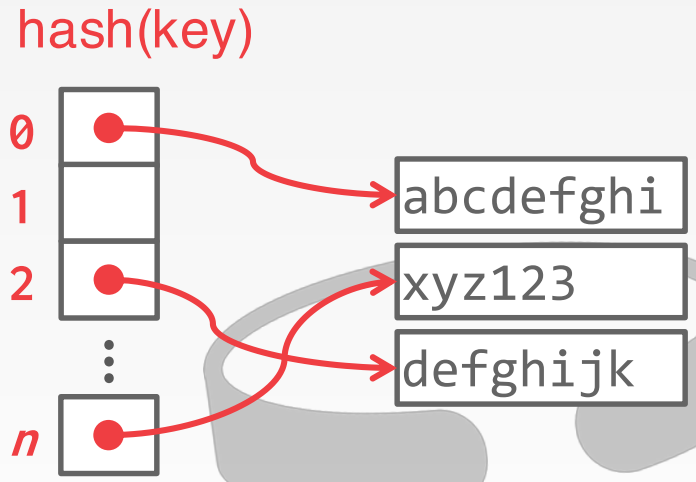


ASSUMPTIONS

You know the number of elements ahead of time.

Each key is unique.

Perfect hash function.
→ If key1≠key2, then hash(key1)≠hash(key2)



HASH TABLE

eg. fastest hash function: always return 1.

Design Decision #1: Hash Function

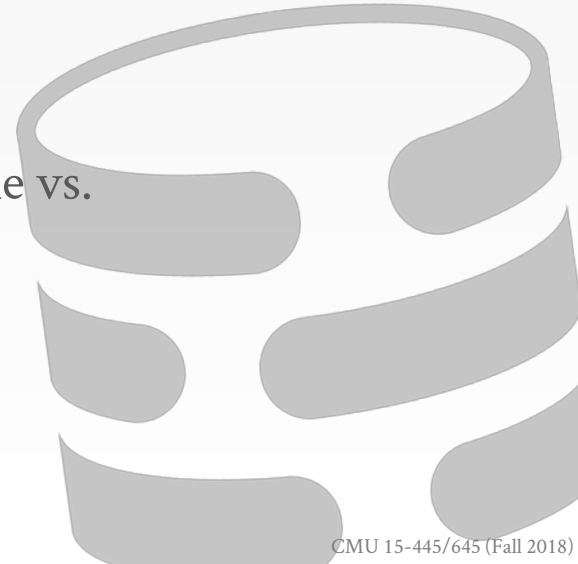
→ How to map a large key space into a smaller domain.
→ Trade-off between being fast vs. collision rate.

速度 vs. 冲突率

Design Decision #2: Hashing Scheme

→ How to handle key collisions after hashing.
→ Trade-off between allocating a large hash table vs. additional instructions to find/insert keys.

内存 vs. 读写时间



TODAY'S AGENDA

Hash Functions
Static Hashing Schemes
Dynamic Hashing Schemes

HASH FUNCTIONS

We don't want to use a cryptographic hash function for our join algorithm.

We want something that is fast and will have a low collision rate.

冲突是不可避免的：生日问题（如果在一个房间要多少人，则两个人的生日相同的概率要大于 50%？）

HASH FUNCTIONS

MurmurHash (2008)

→ Designed to a fast, general purpose hash function.

Google CityHash (2011)

→ Based on ideas from MurmurHash2

→ Designed to be faster for short keys (<64 bytes).

Google FarmHash (2014)

→ Newer version of CityHash with better collision rates.

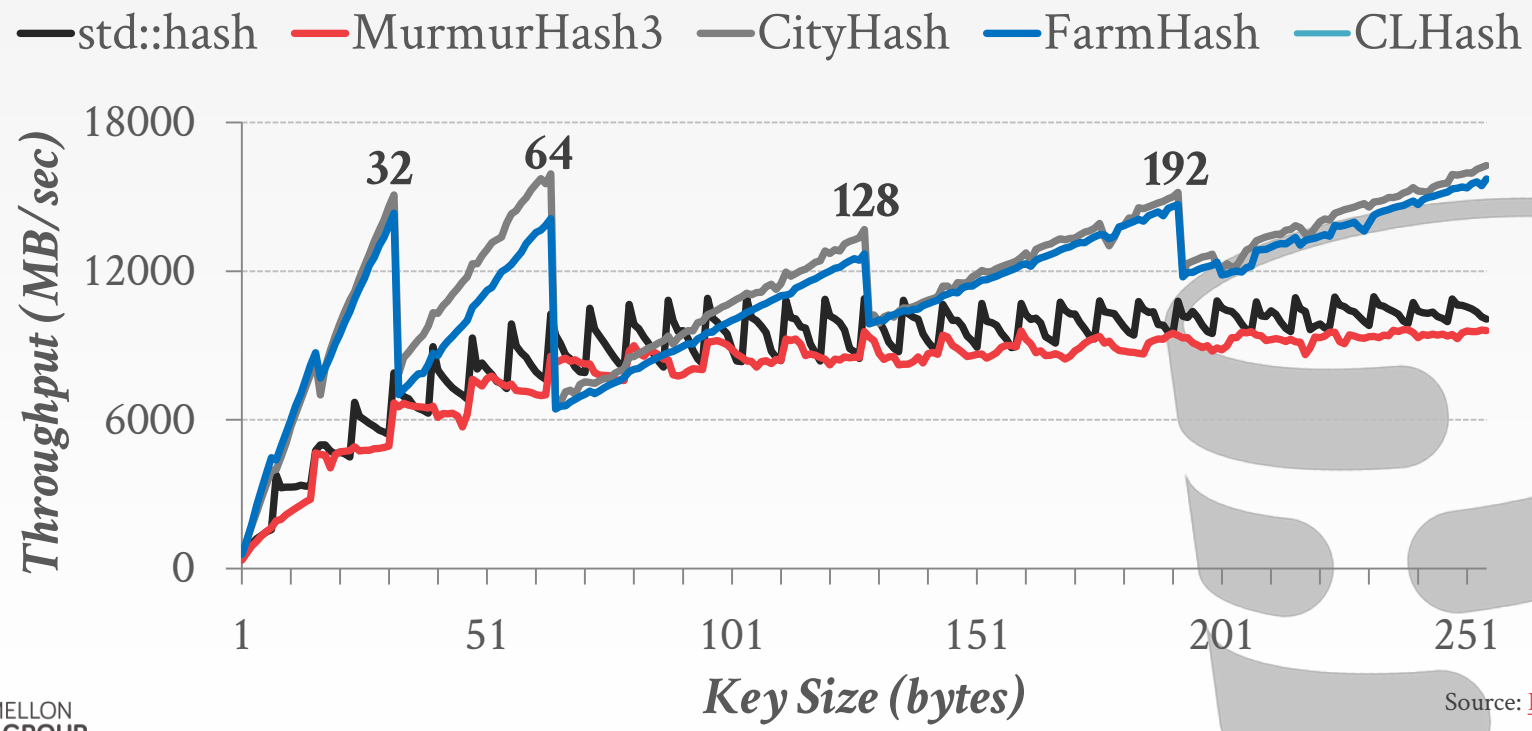
CLHash (2016)

→ Fast hashing function based on carry-less multiplication.

不是新出来的，2014年Intel、AMD 在硬件上面做了优化，所以它变得很快了

HASH FUNCTION BENCHMARKS

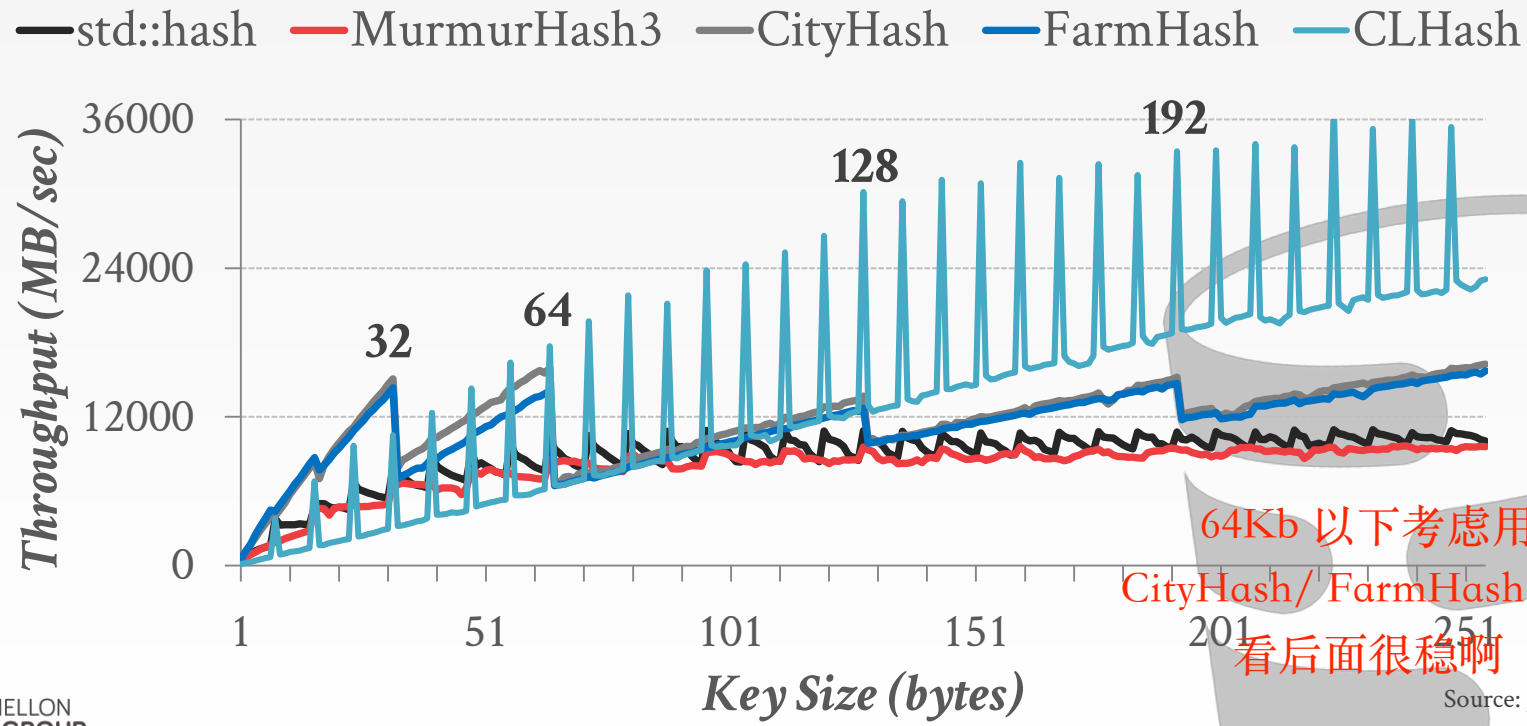
Intel Core i7-8700K @ 3.70GHz



Source: Fredrik Widlund

HASH FUNCTION BENCHMARKS

Intel Core i7-8700K @ 3.70GHz



64Kb 以下考虑用 CityHash/ FarmHash？我看后面很稳啊

Source: Fredrik Widlund

STATIC HASHING SCHEMES

Approach #1: Linear Probe Hashing

Approach #2: Robin Hood Hashing

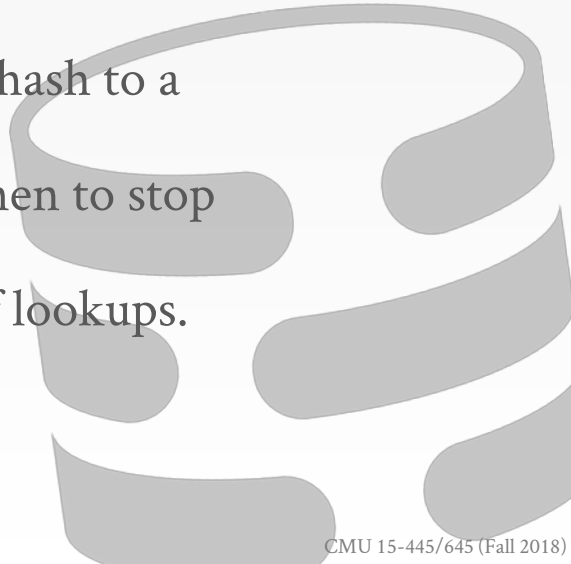
Approach #3: Cuckoo Hashing

LINEAR PROBE HASHING

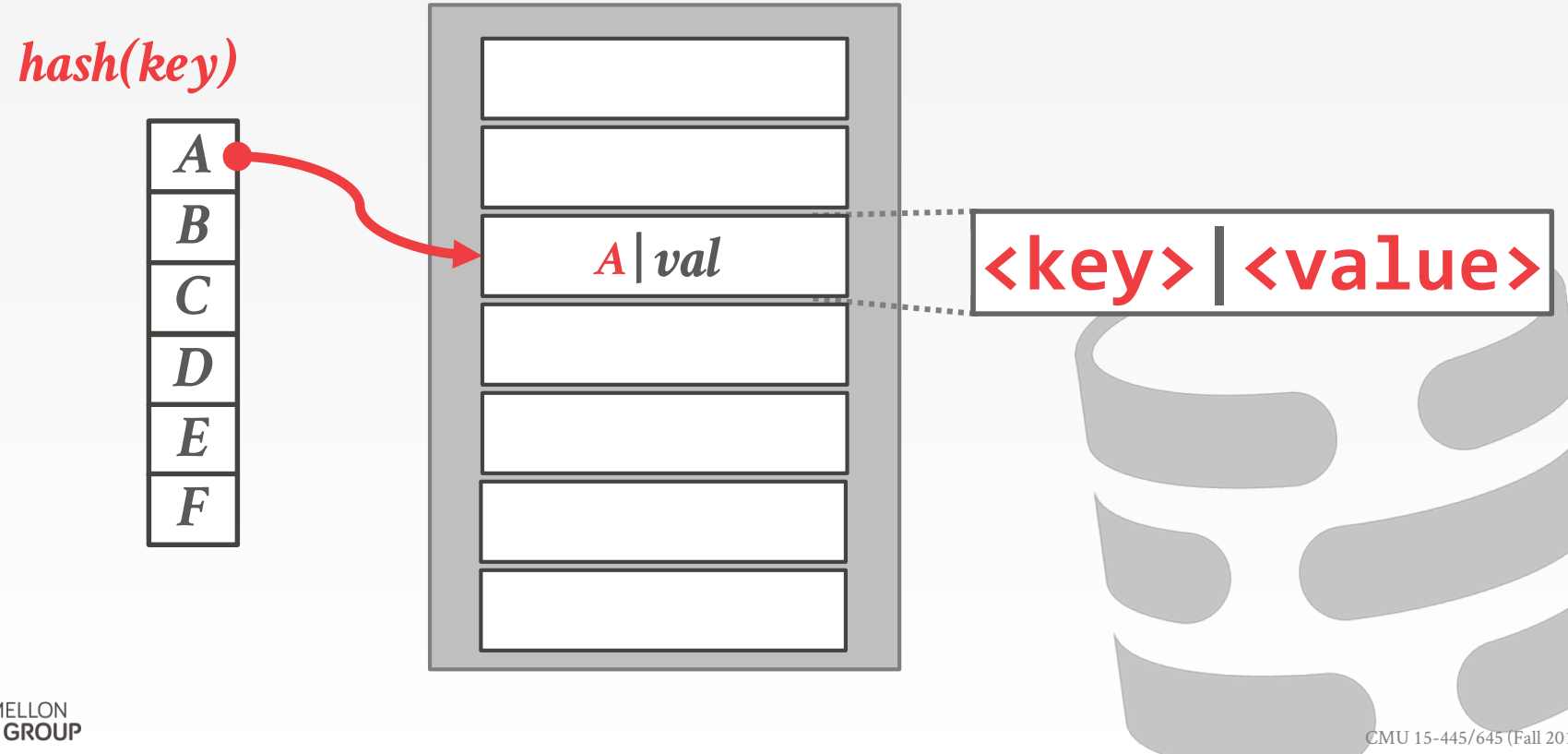
Single giant table of slots.

Resolve collisions by linearly searching for the next free slot in the table.

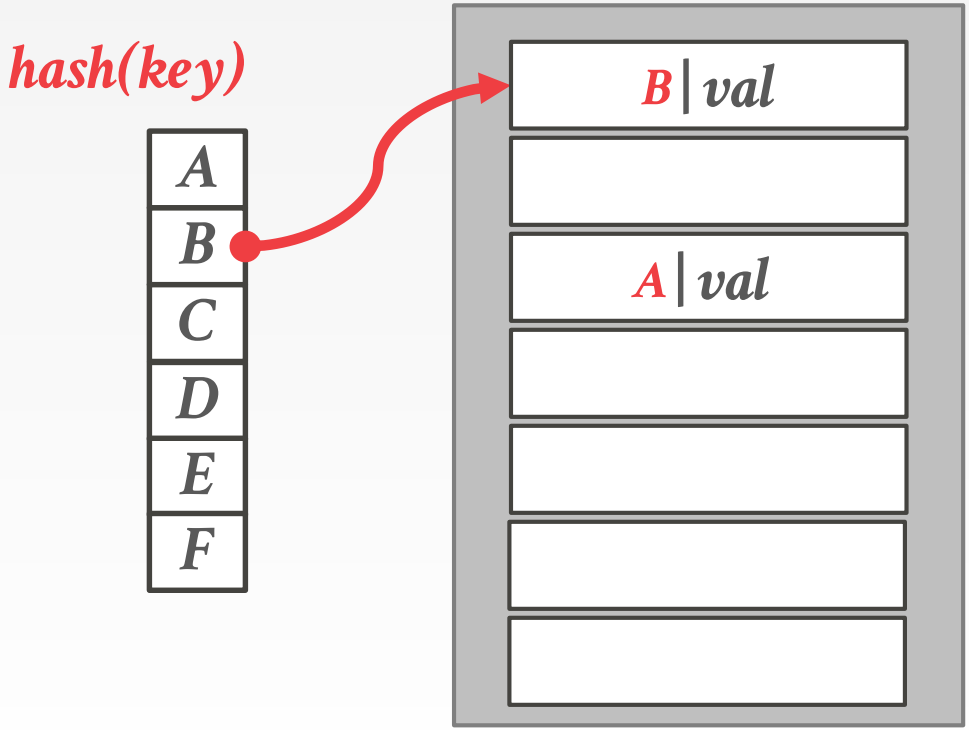
→ To determine whether an element is present, hash to a location in the index and scan for it.
→ Have to store the key in the index to know when to stop scanning.
→ Insertions and deletions are generalizations of lookups.



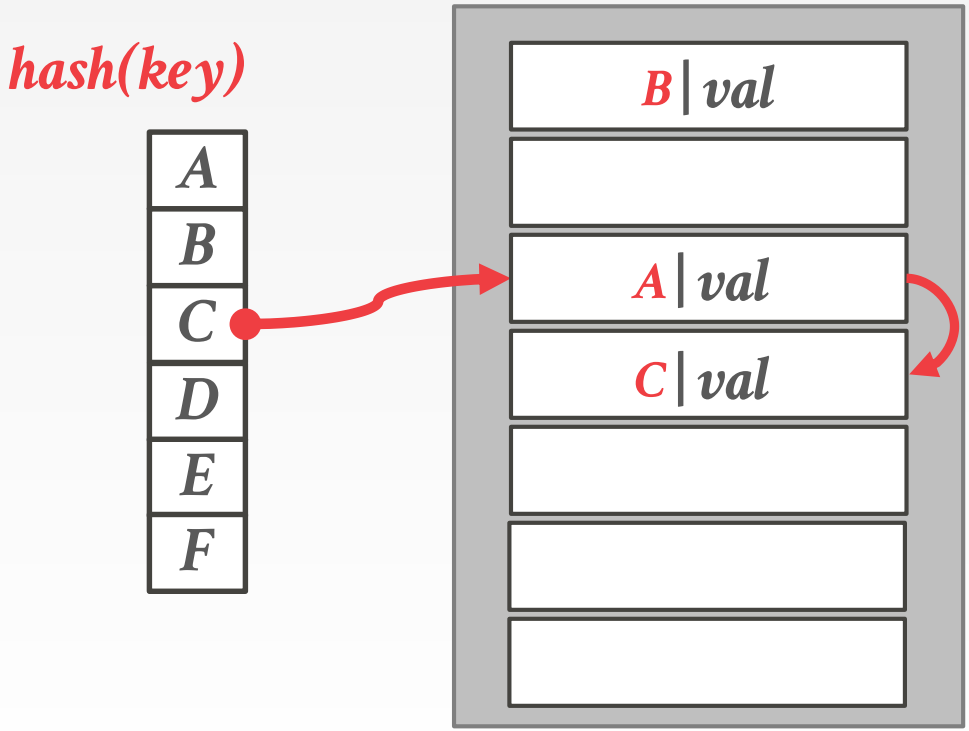
LINEAR PROBE HASHING



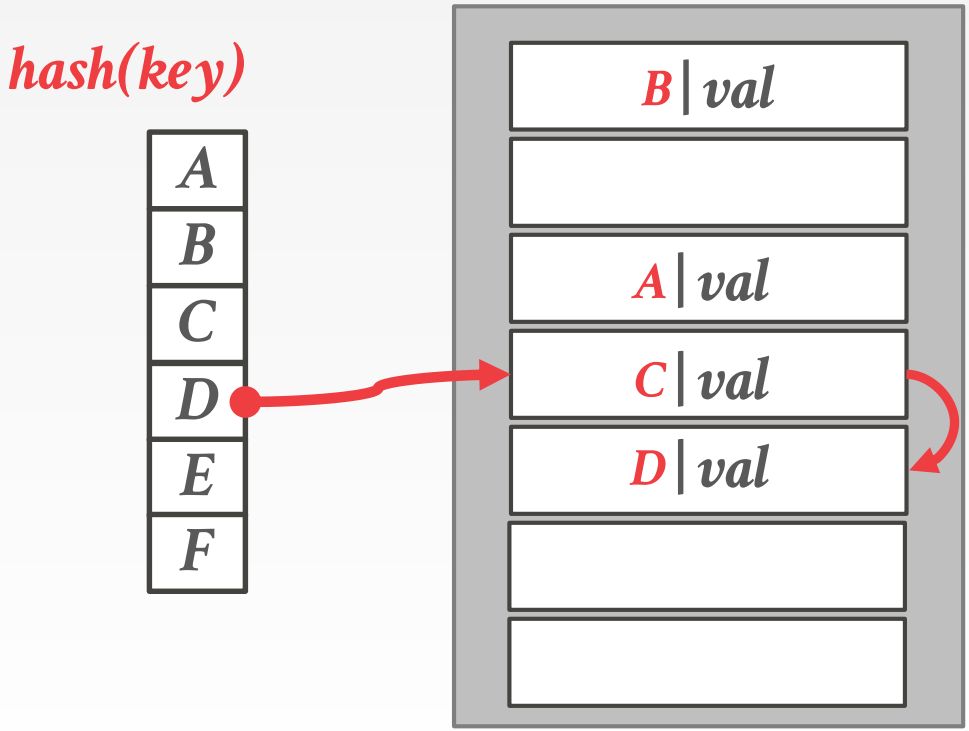
LINEAR PROBE HASHING



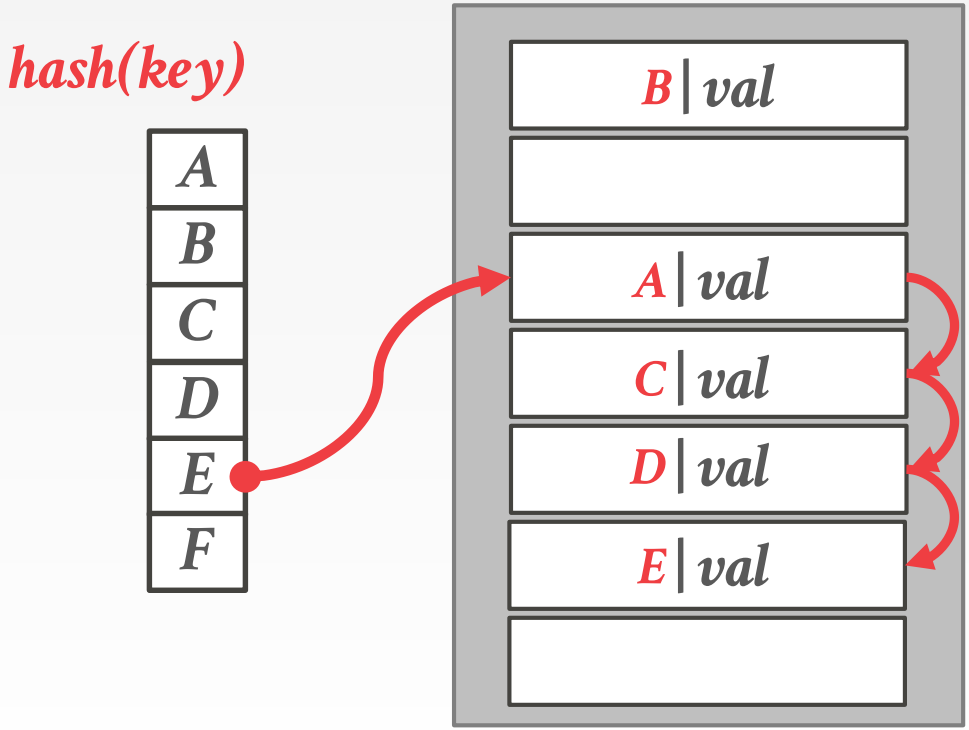
LINEAR PROBE HASHING



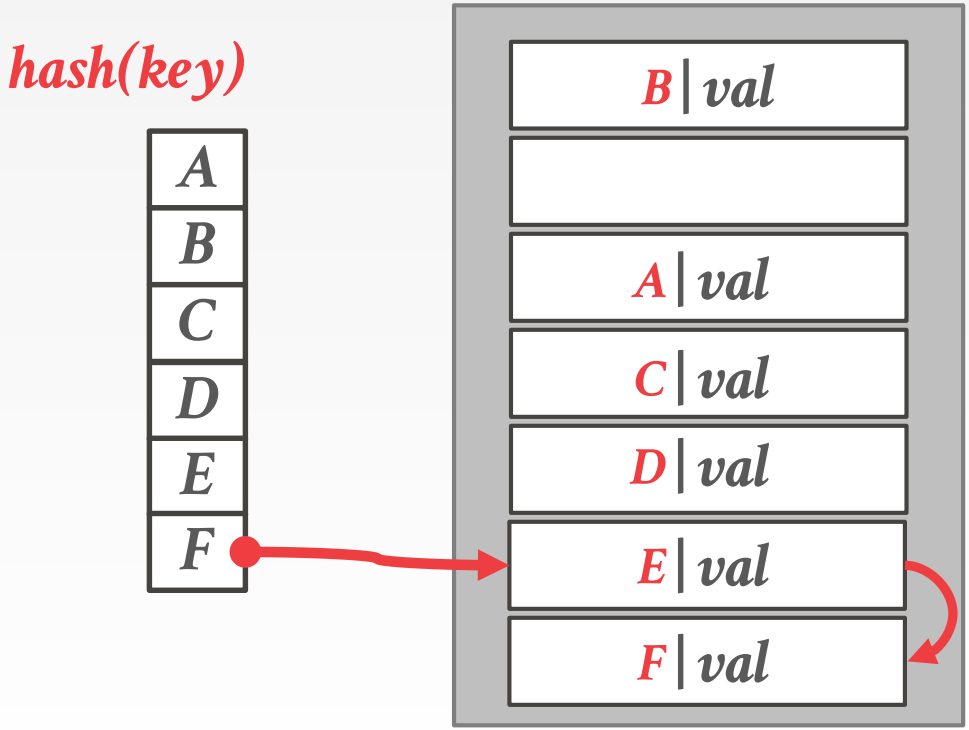
LINEAR PROBE HASHING



LINEAR PROBE HASHING



LINEAR PROBE HASHING



NON-UNIQUE KEYS

Choice #1: Separate Linked List

→ Store values in separate storage area for each key.

缺点：只有一个值也会创建一个 list， 内存浪费

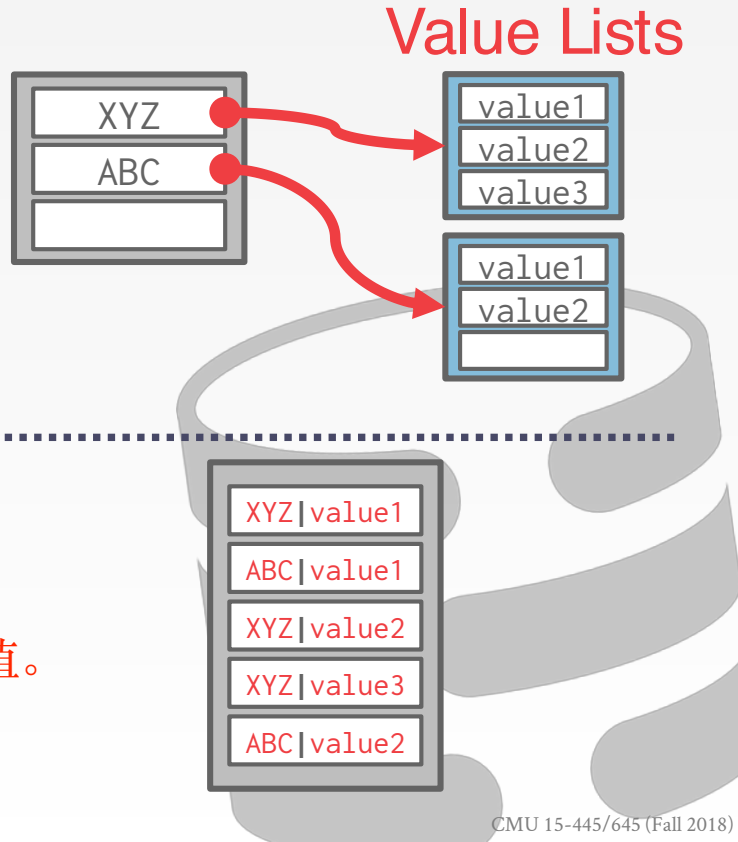
优点：可以快速取出所有的值

Choice #2: Redundant Keys

→ Store duplicate keys entries together in the hash table.

大多数人用2来处理重复 key 值

缺点？：需要遍历所有的 entry 获取特定 key 的所有值。



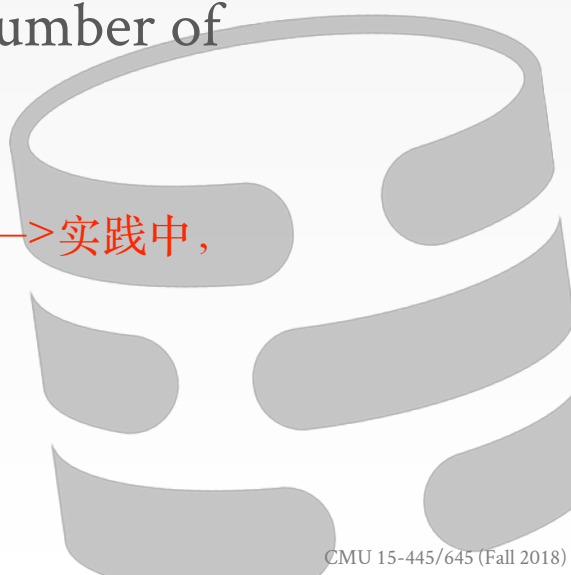
OBSERVATION

最坏的情况，O(1)变 O(n)

To reduce the # of wasteful comparisons, it is important to avoid collisions of hashed keys.

This requires a hash table with ~2x the number of slots as the number of elements.

为了降低冲突 —> 分配更多的内存（更多的 slots）—>实践中，2倍于需要存储的 entry 数目



ROBIN HOOD HASHING

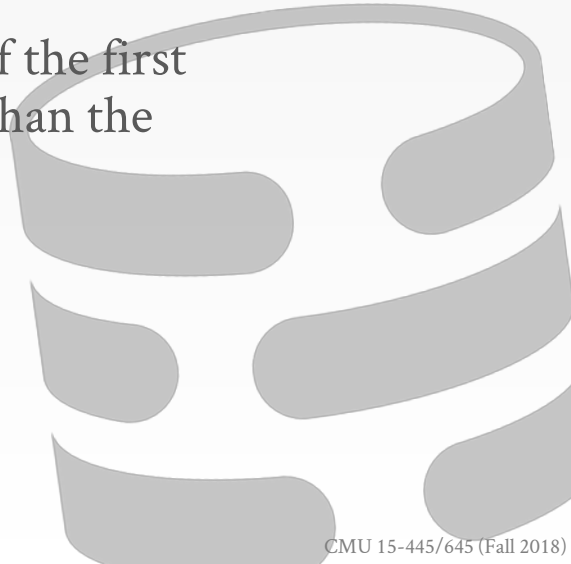
Variant of linear probe hashing that steals slots from "rich" keys and give them to "poor" keys.

→ Each key tracks the number of positions they are from where its optimal position in the table.

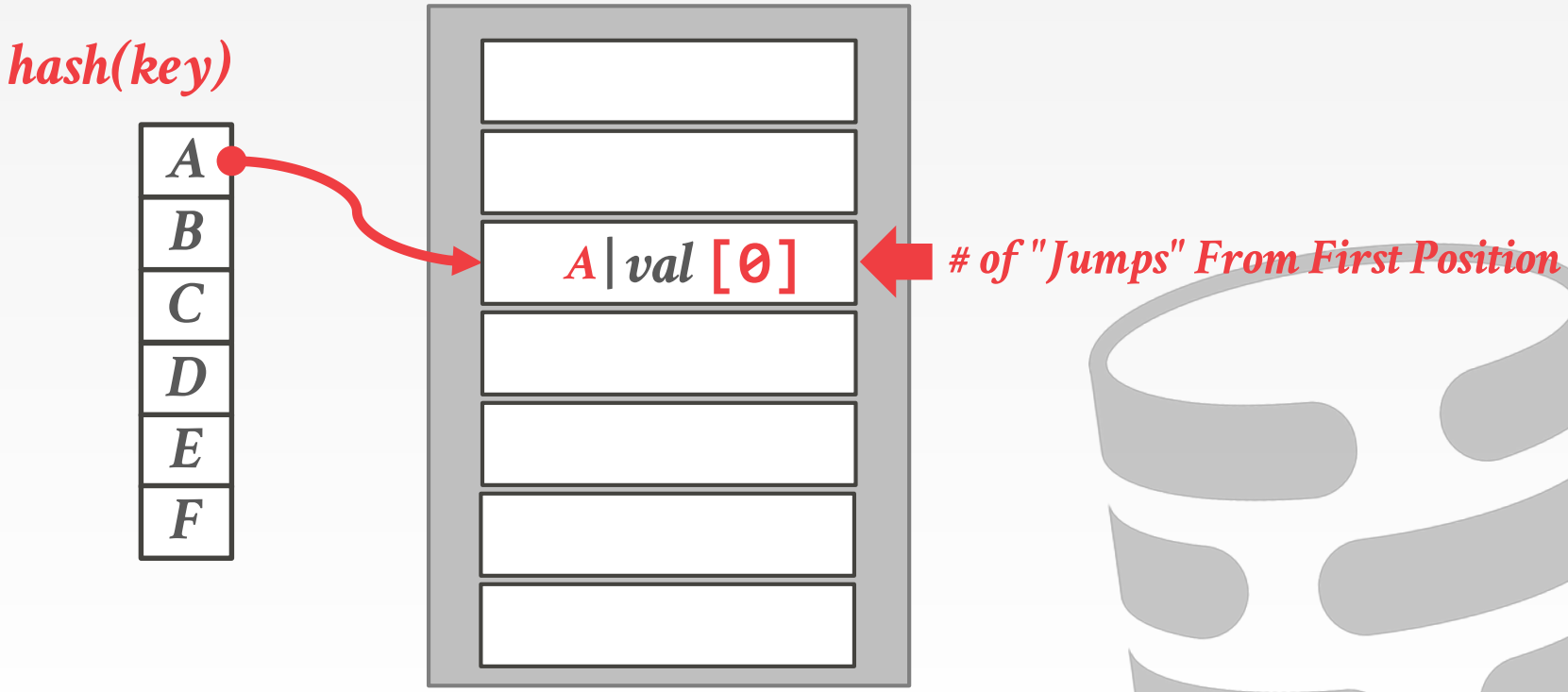
→ On insert, a key takes the slot of another key if the first key is farther away from its optimal position than the second key.

并没有比linear probe hashing 快，scanning is cheap.

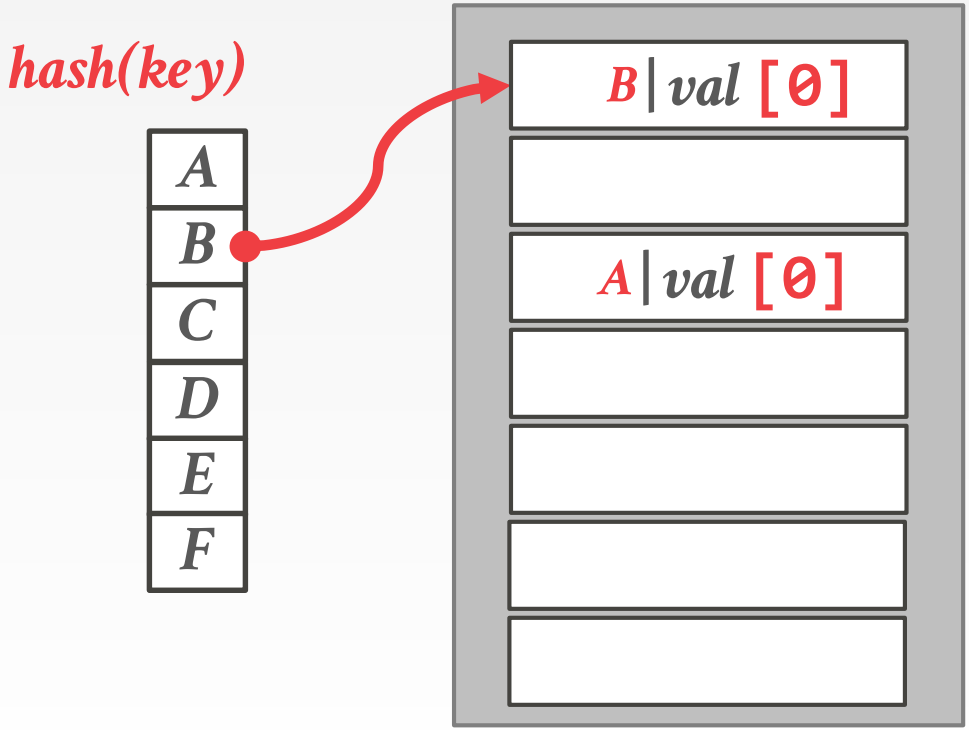
挪动key 对应的位置会导致 CPU 误判（cache）



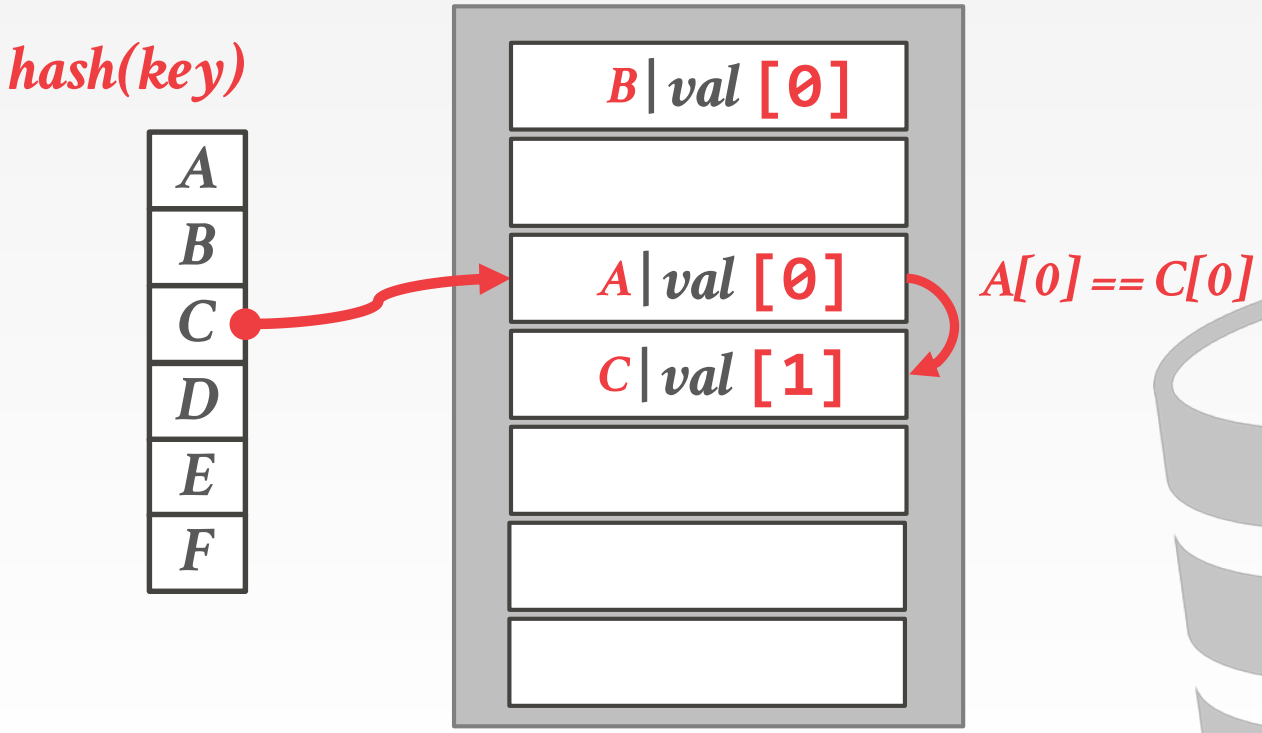
ROBIN HOOD HASHING



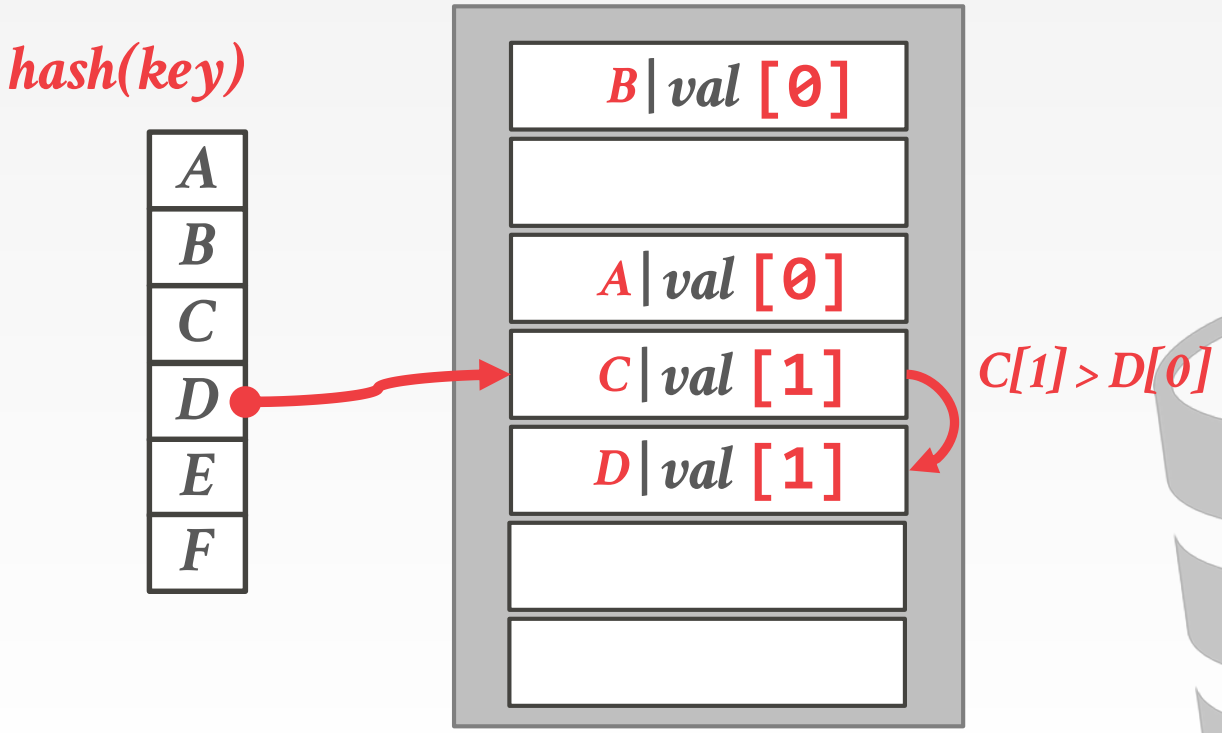
ROBIN HOOD HASHING



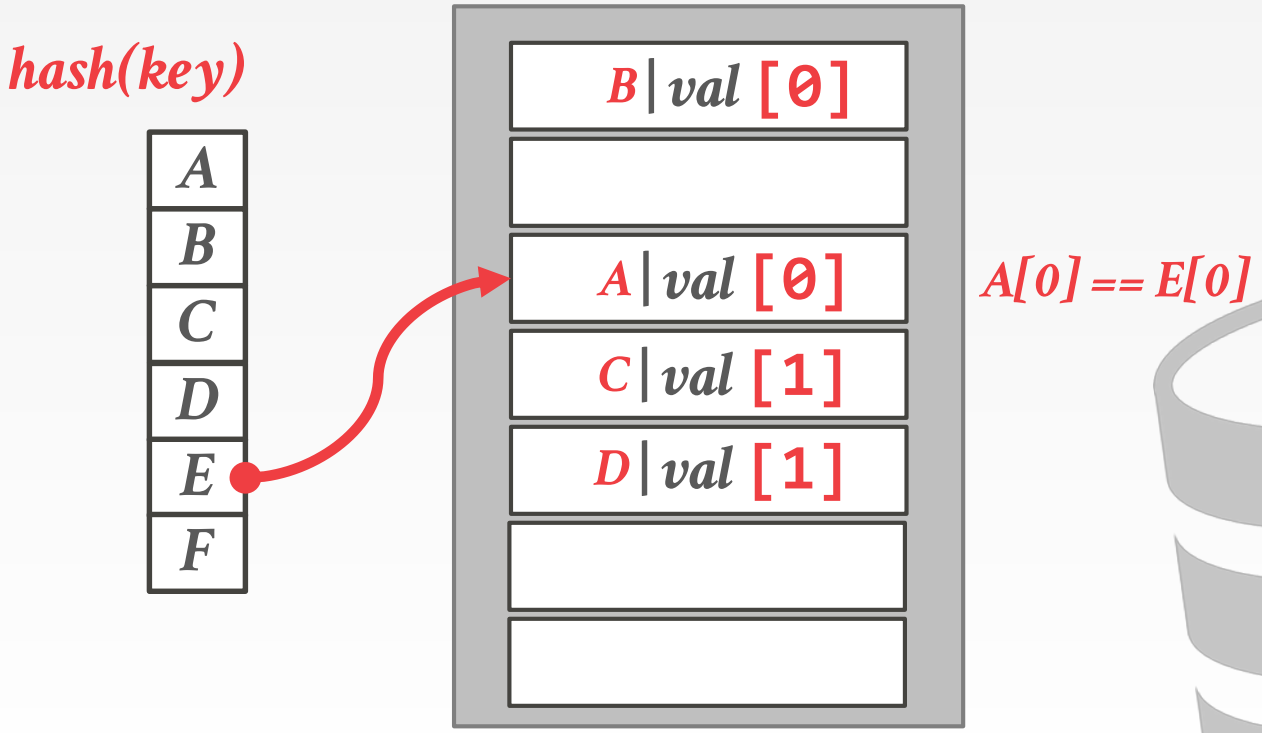
ROBIN HOOD HASHING



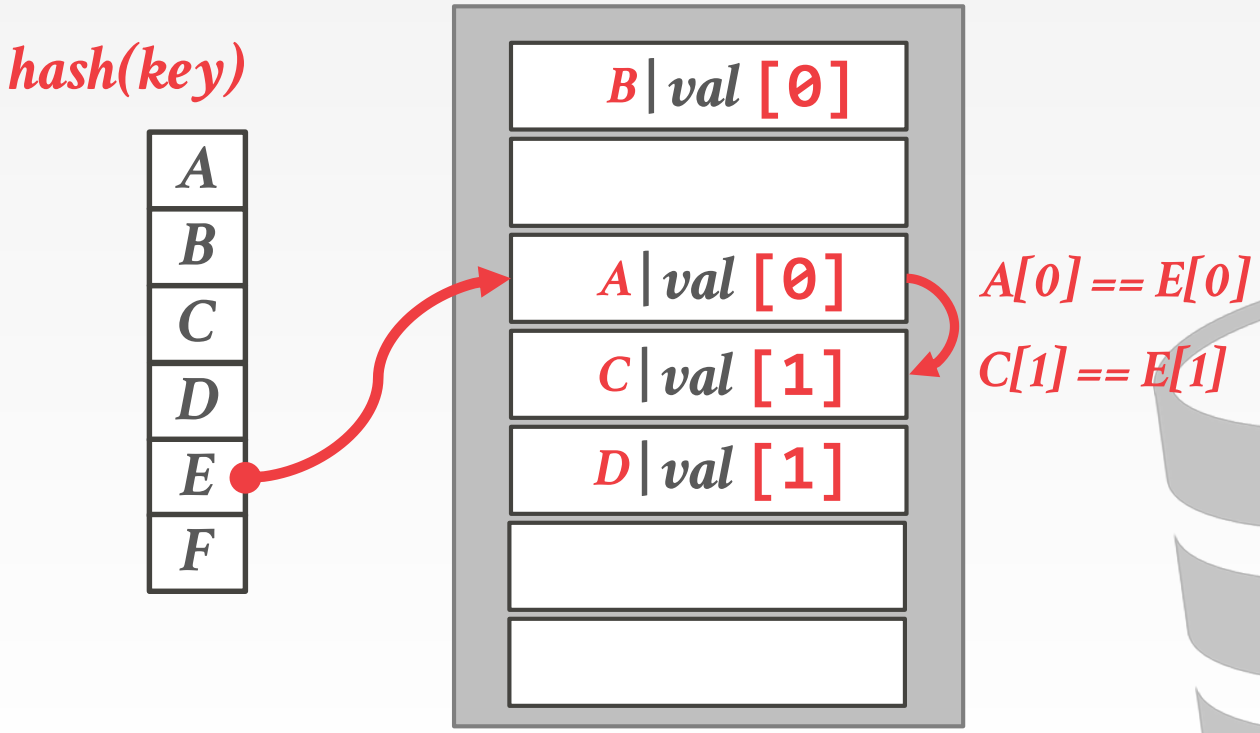
ROBIN HOOD HASHING



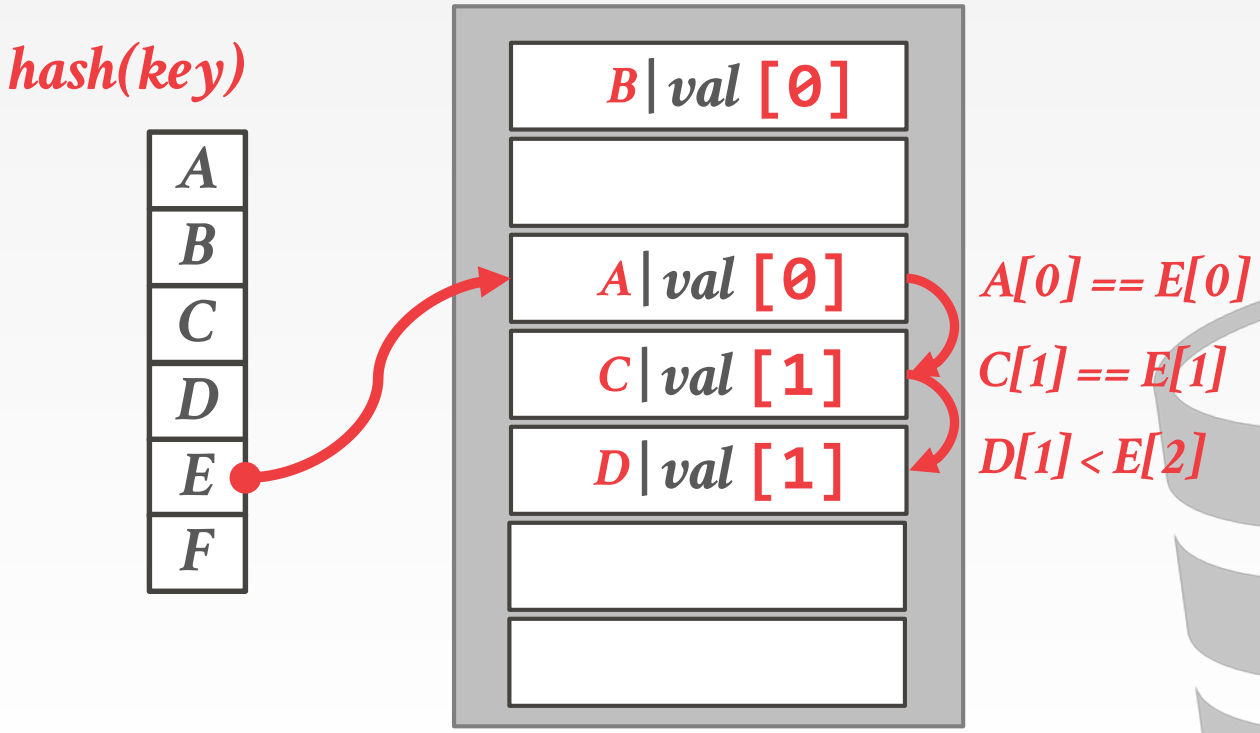
ROBIN HOOD HASHING



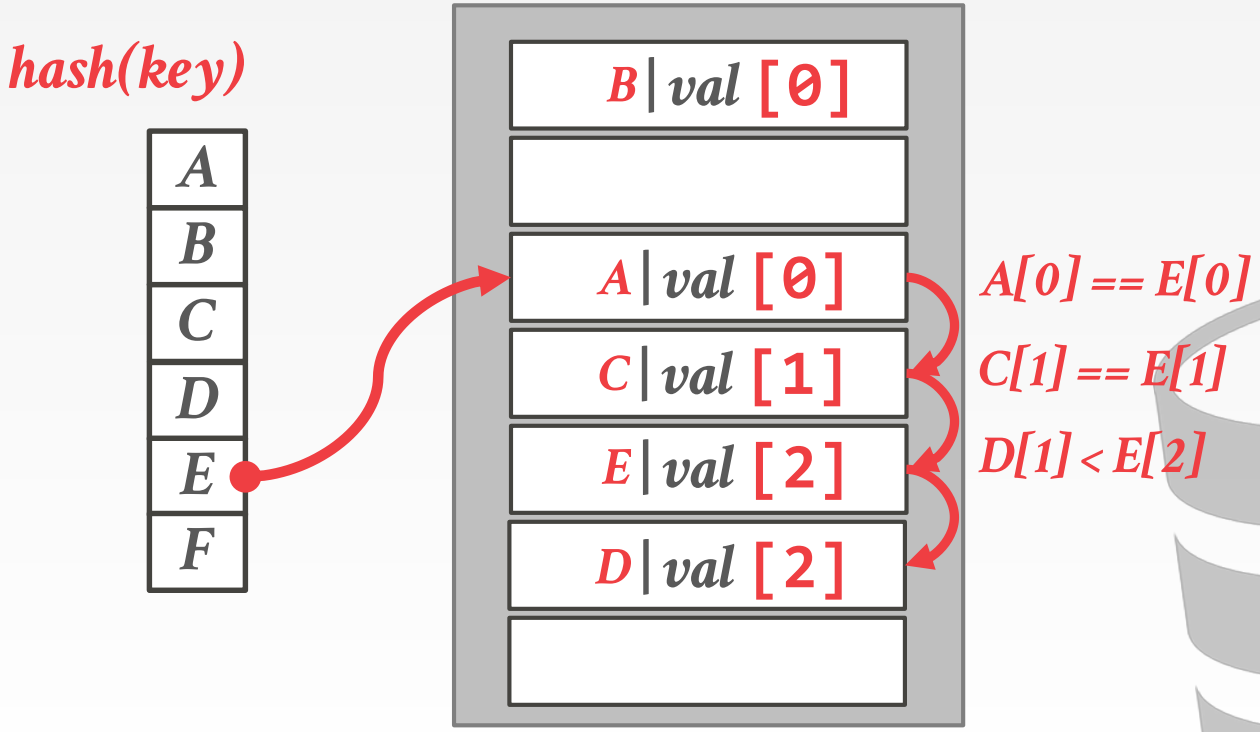
ROBIN HOOD HASHING



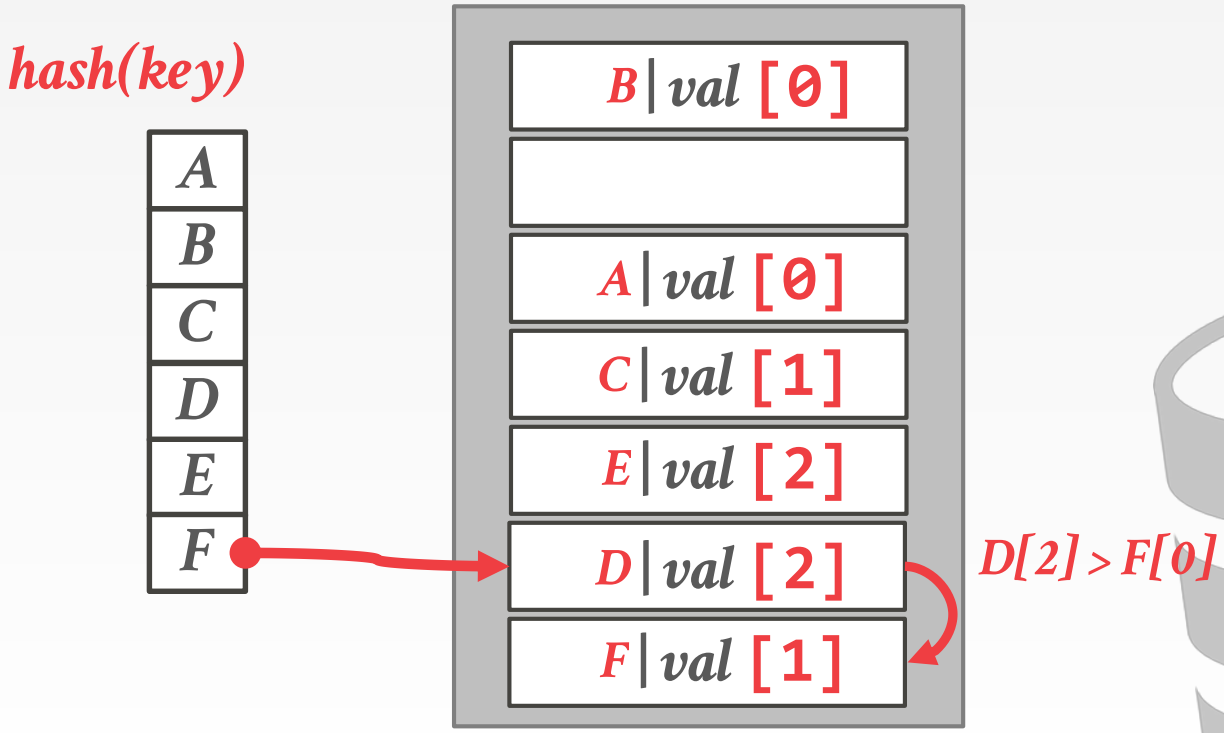
ROBIN HOOD HASHING



ROBIN HOOD HASHING



ROBIN HOOD HASHING



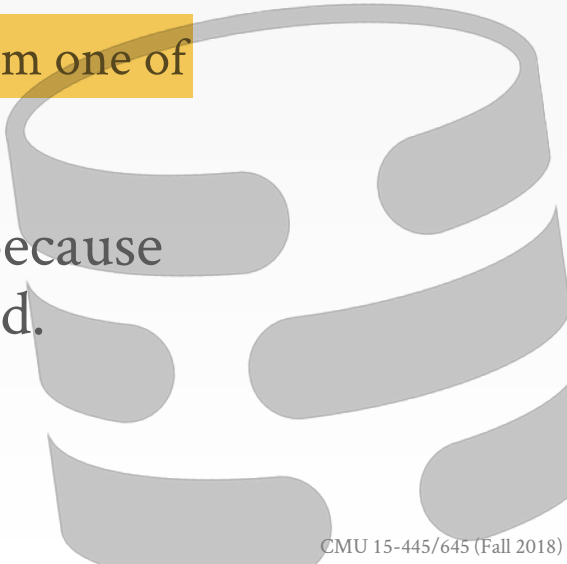
CUCKOO HASHING

Use multiple hash tables with different hash functions.

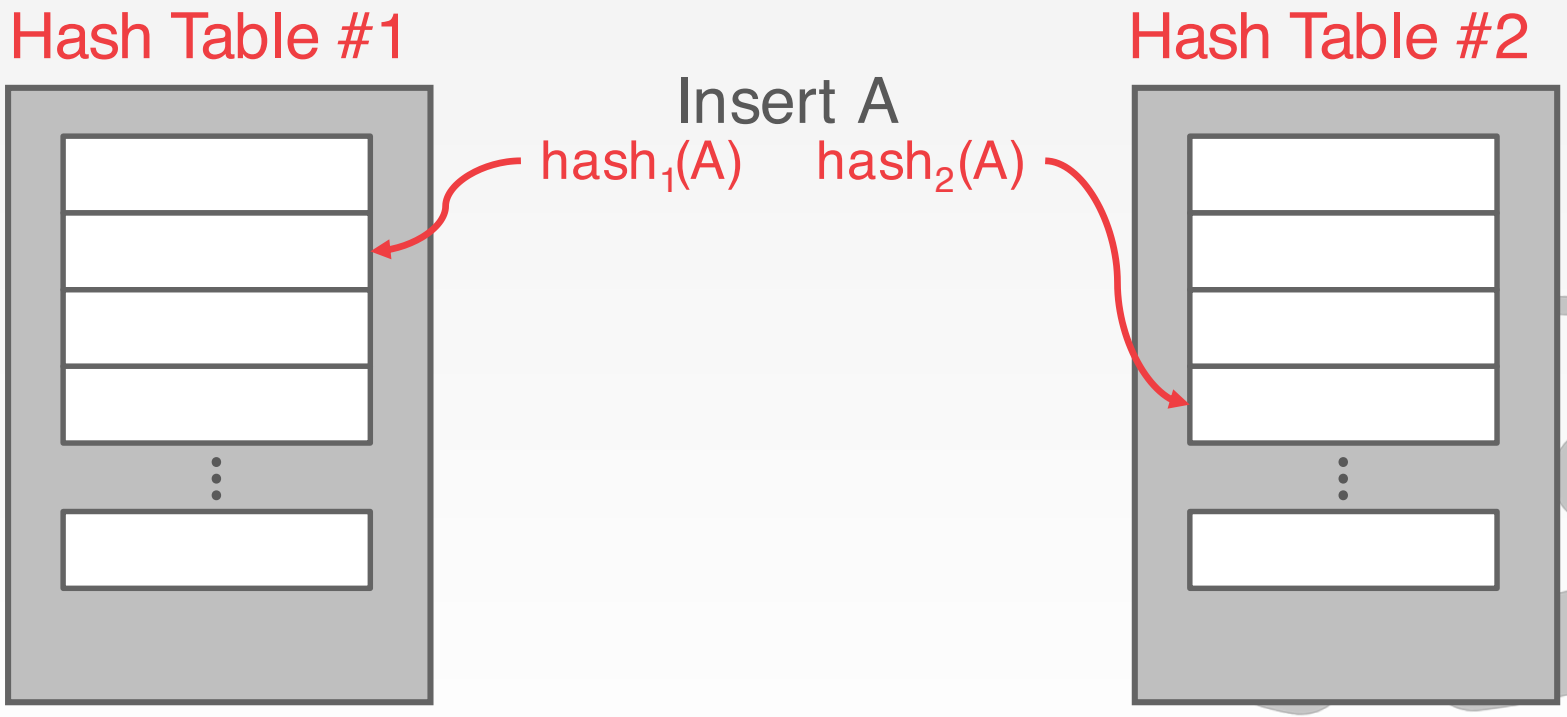
→ On insert, check every table and pick anyone that has a free slot.

→ If no table has a free slot, evict the element from one of them and then re-hash it find a new location.

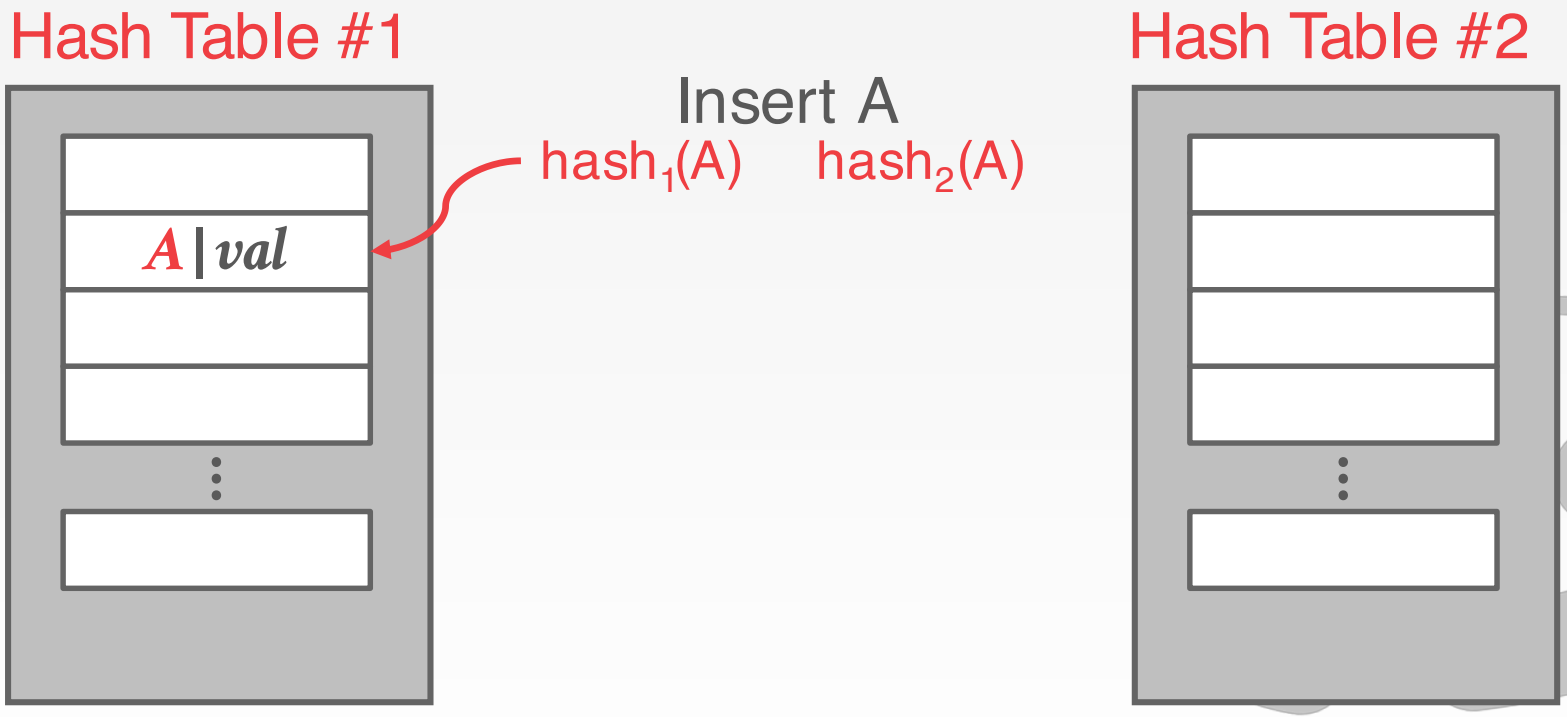
Look-ups and deletions are always O(1) because only one location per hash table is checked.



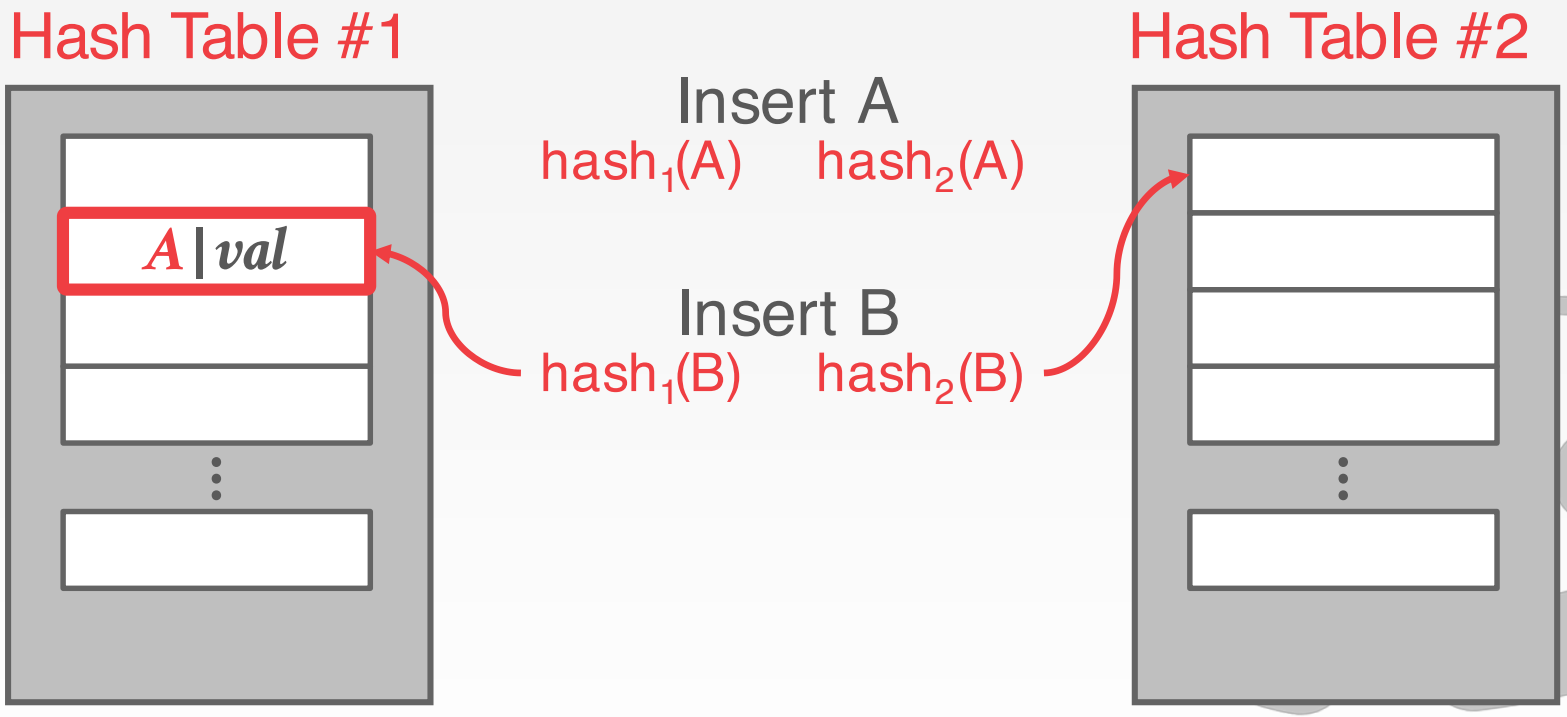
CUCKOO HASHING



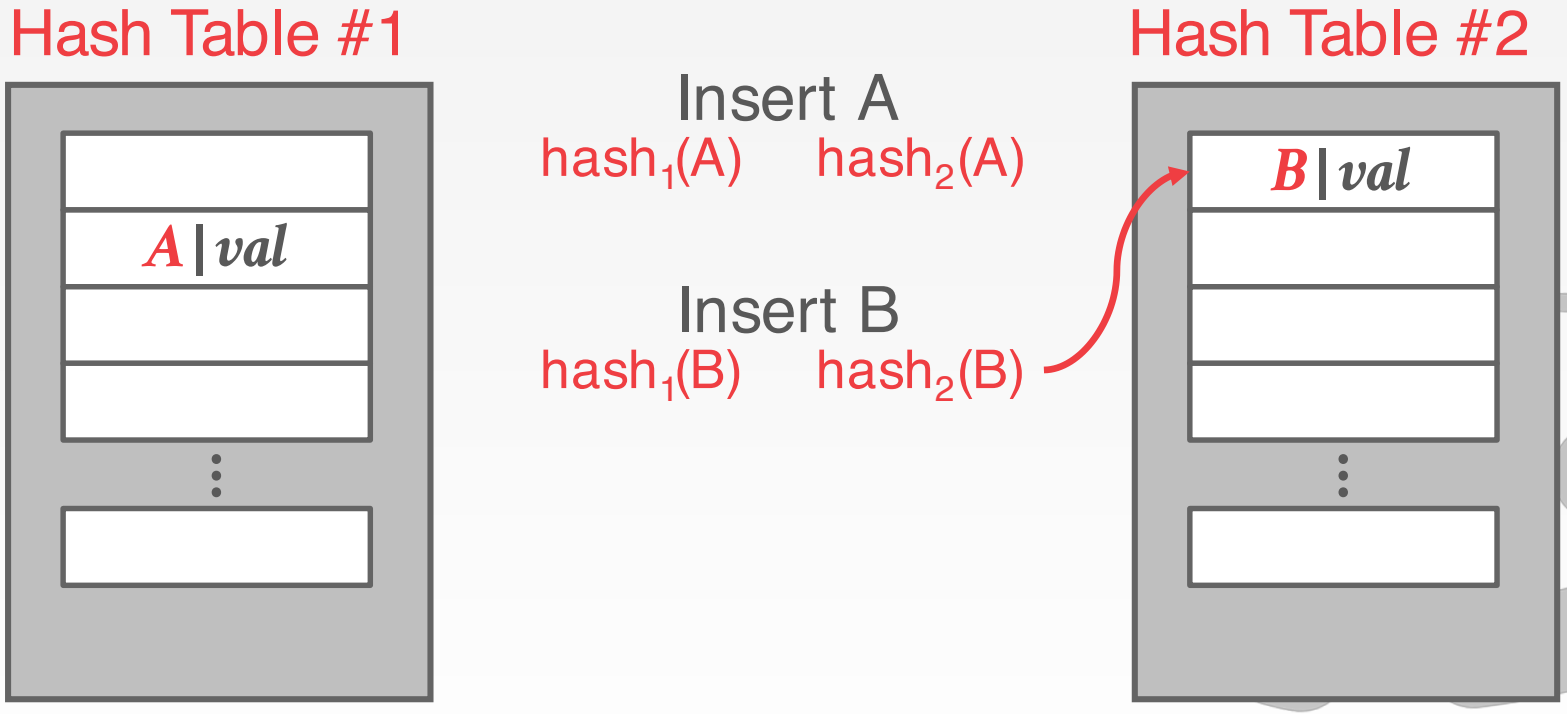
CUCKOO HASHING



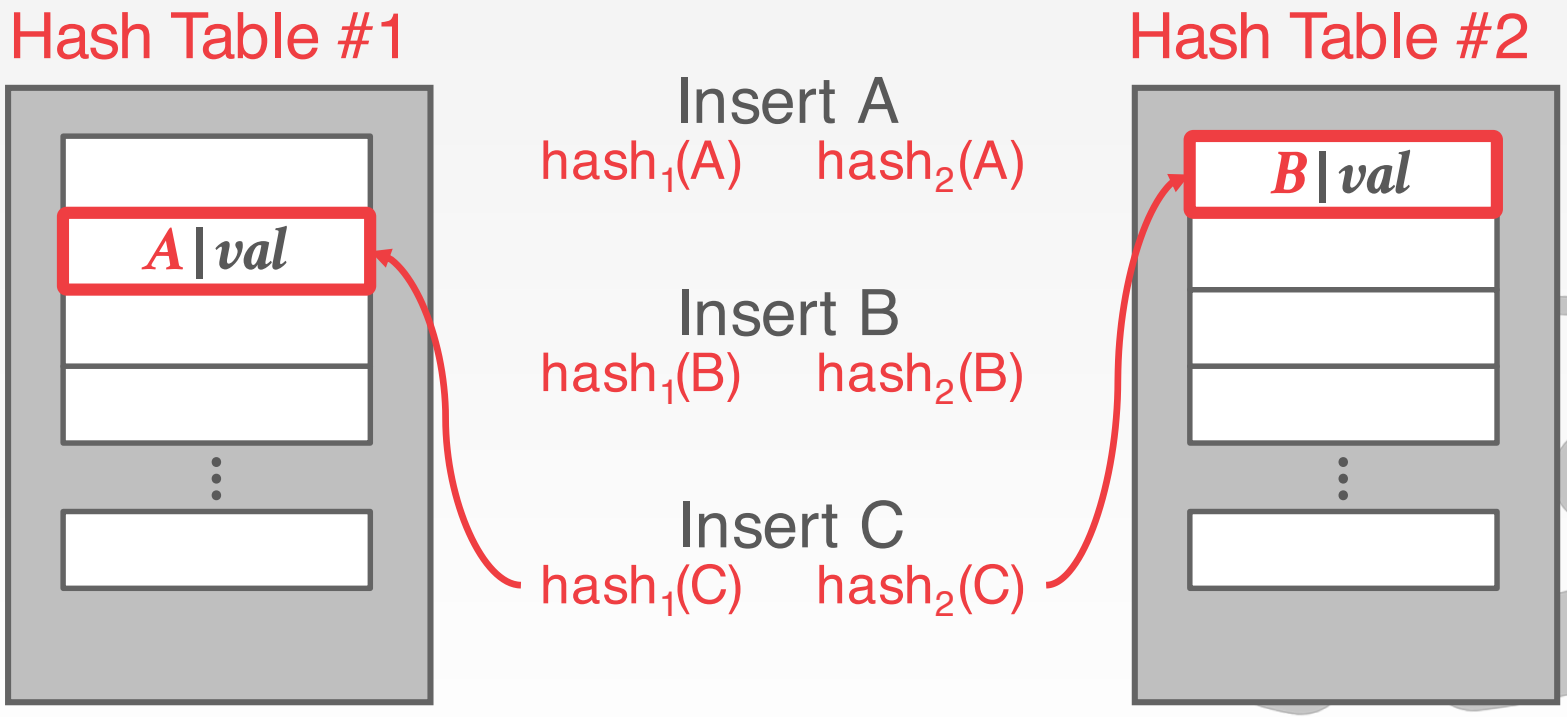
CUCKOO HASHING



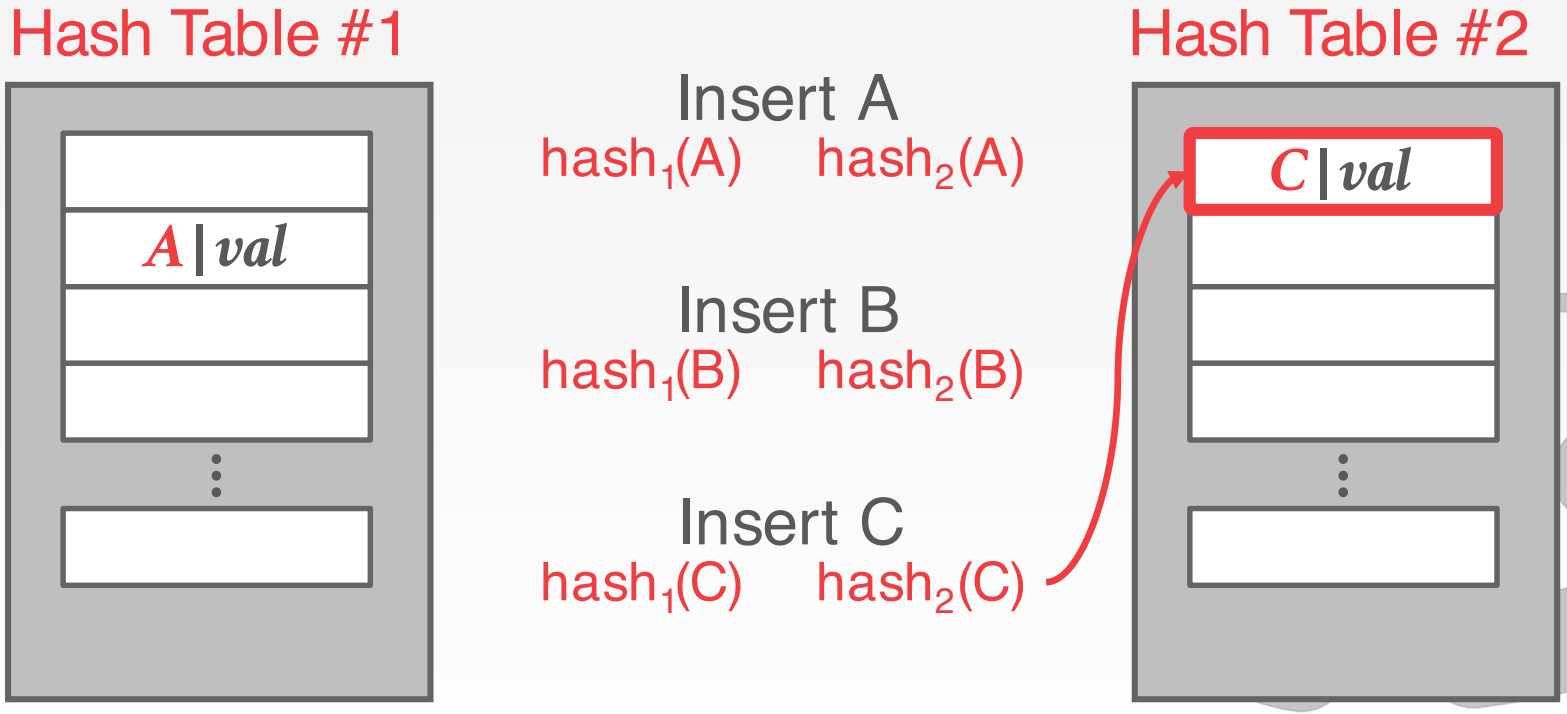
CUCKOO HASHING



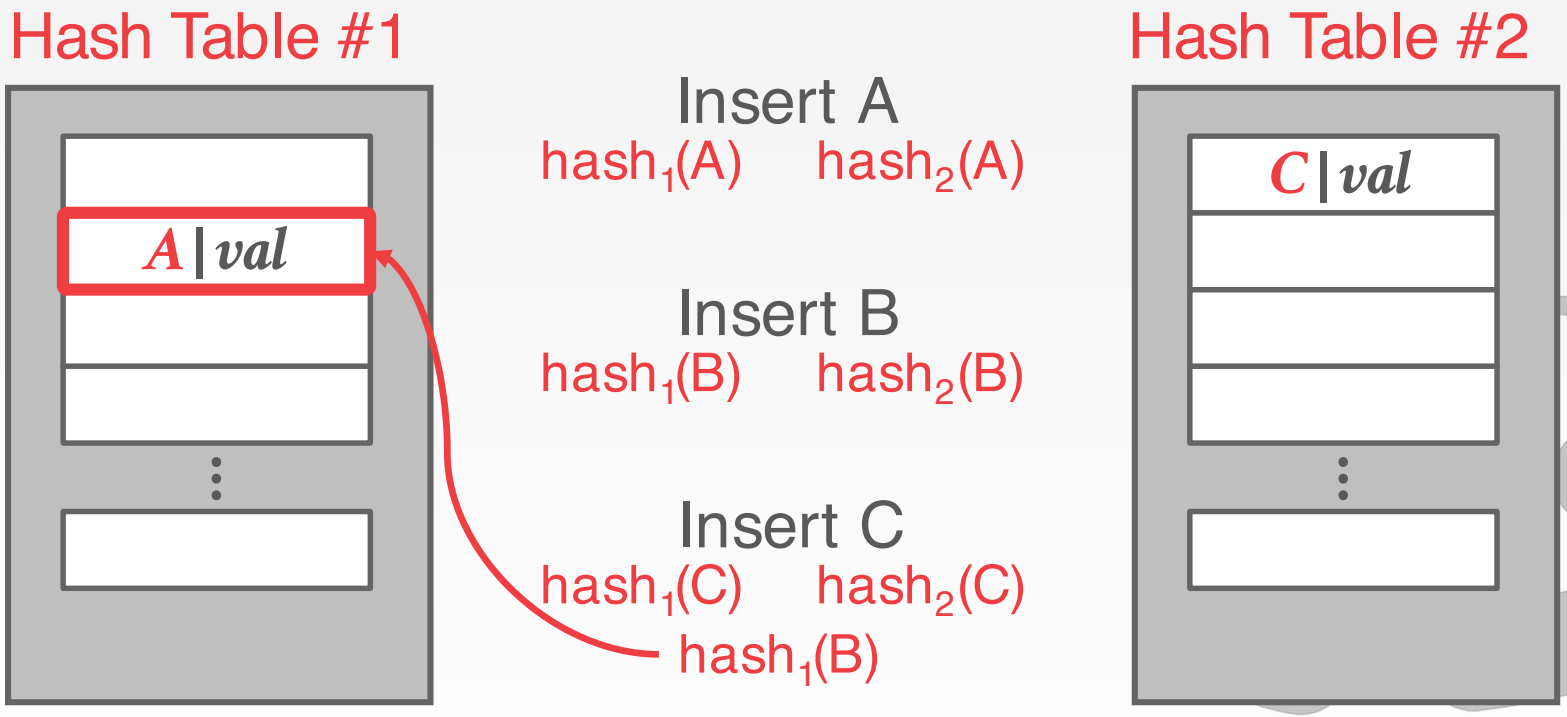
CUCKOO HASHING



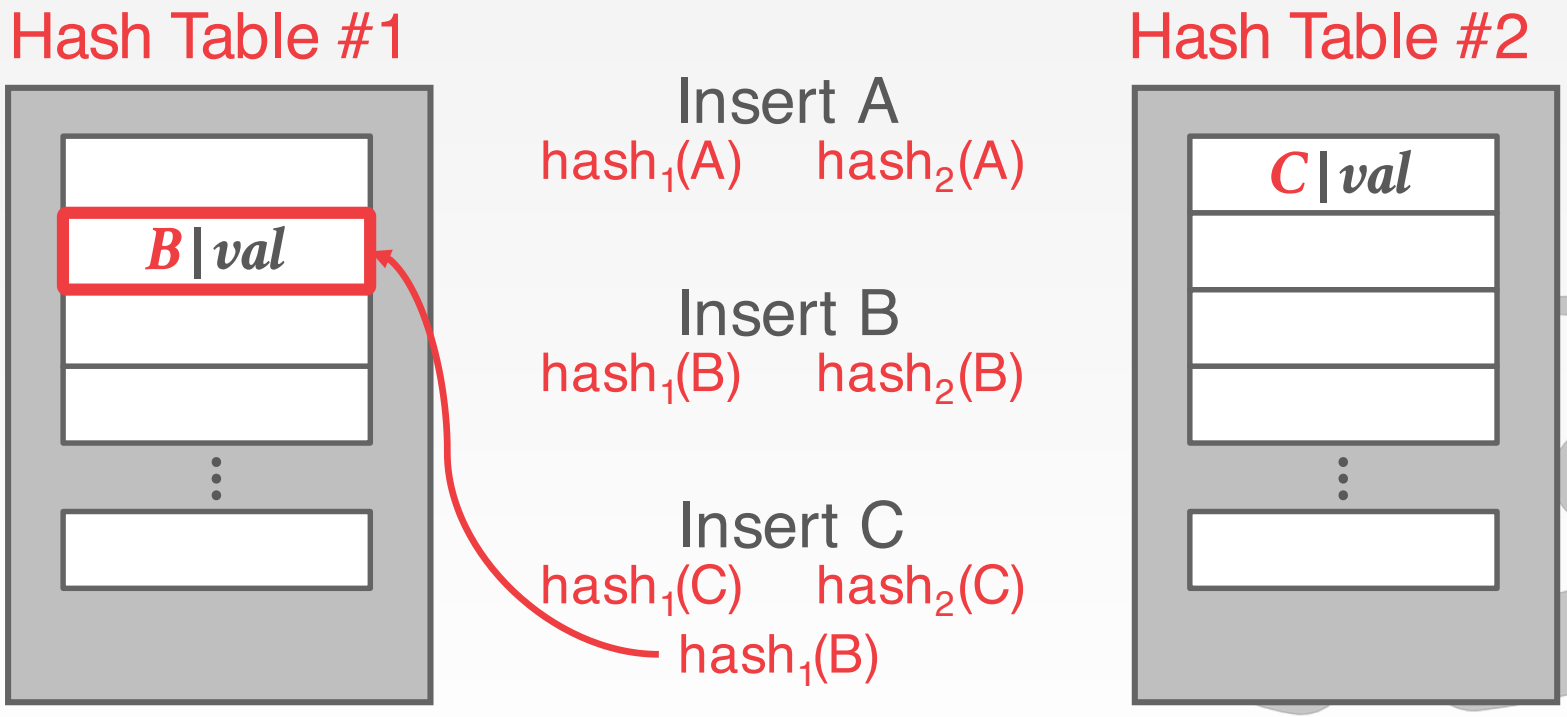
CUCKOO HASHING



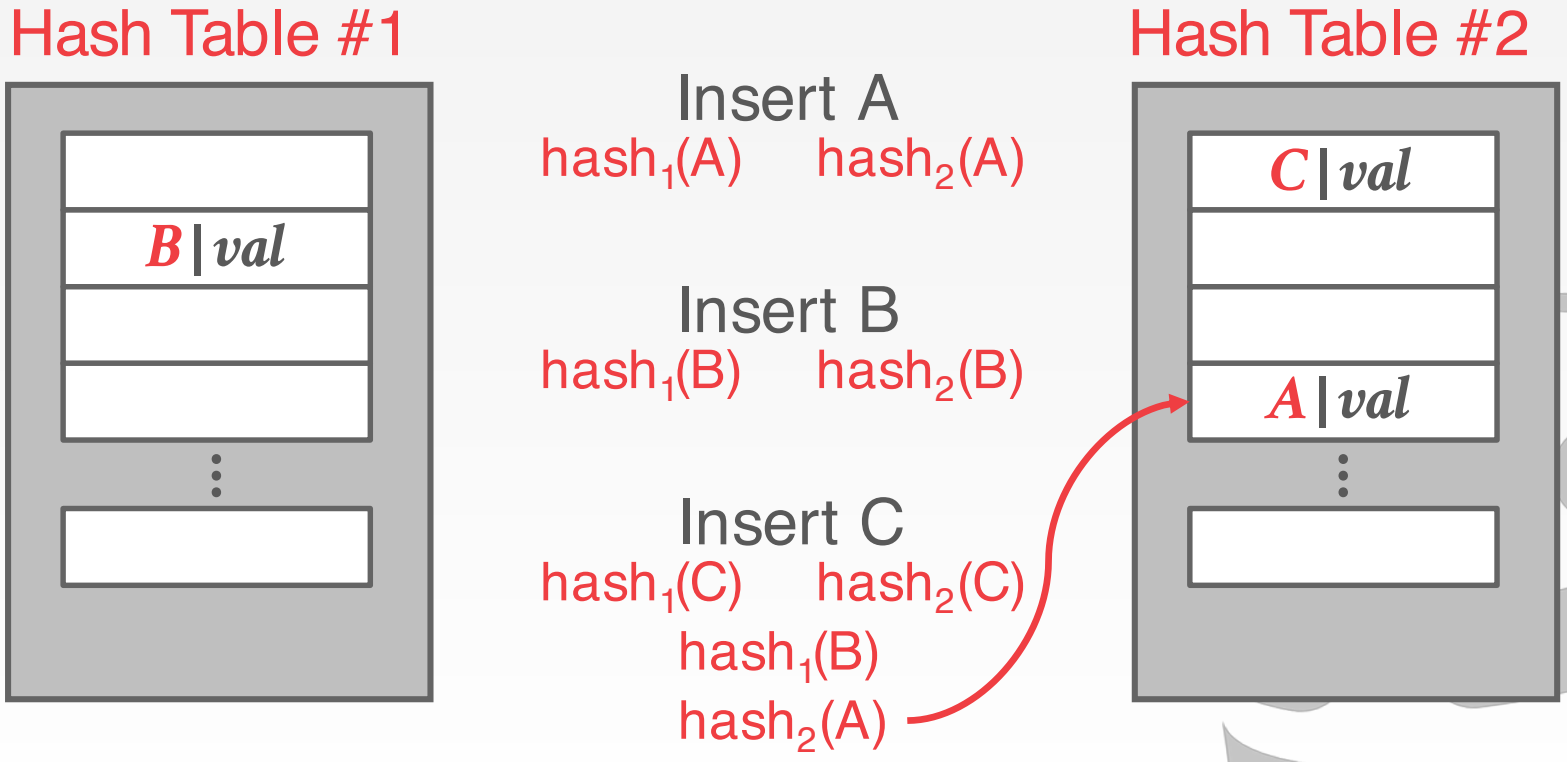
CUCKOO HASHING



CUCKOO HASHING



CUCKOO HASHING



CUCKOO HASHING

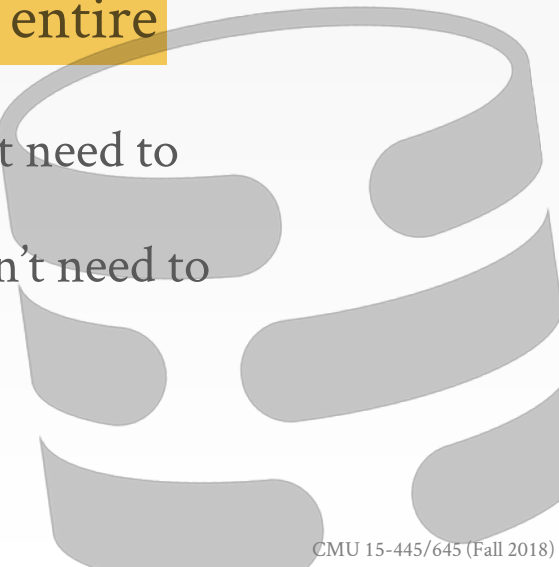
IBM DB2

Make sure that we don't get stuck in an infinite loop when moving keys.

If we find a cycle, then we can rebuild the entire hash tables with new hash functions.

→ With **two** hash functions, we (probably) won't need to rebuild the table until it is at about 50% full.

→ With **three** hash functions, we (probably) won't need to rebuild the table until it is at about 90% full.

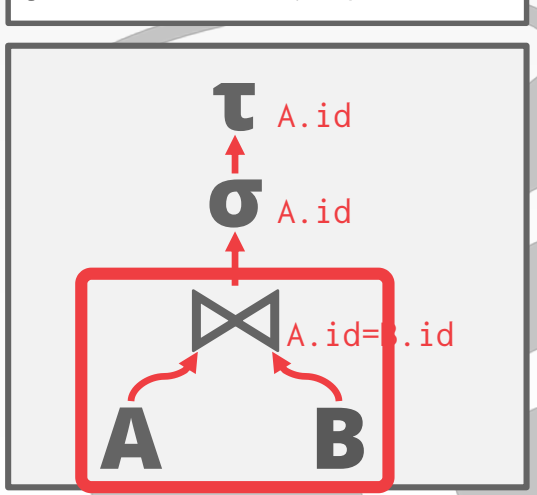


OBSERVATION

The previous hash tables require knowing the number of elements you want to store ahead of time.

→ Otherwise you have rebuild the entire table if you need to grow/shrink.

```
SELECT A.id
  FROM A, B
 WHERE A.id = B.id
ORDER BY A.id
```



OBSERVATION

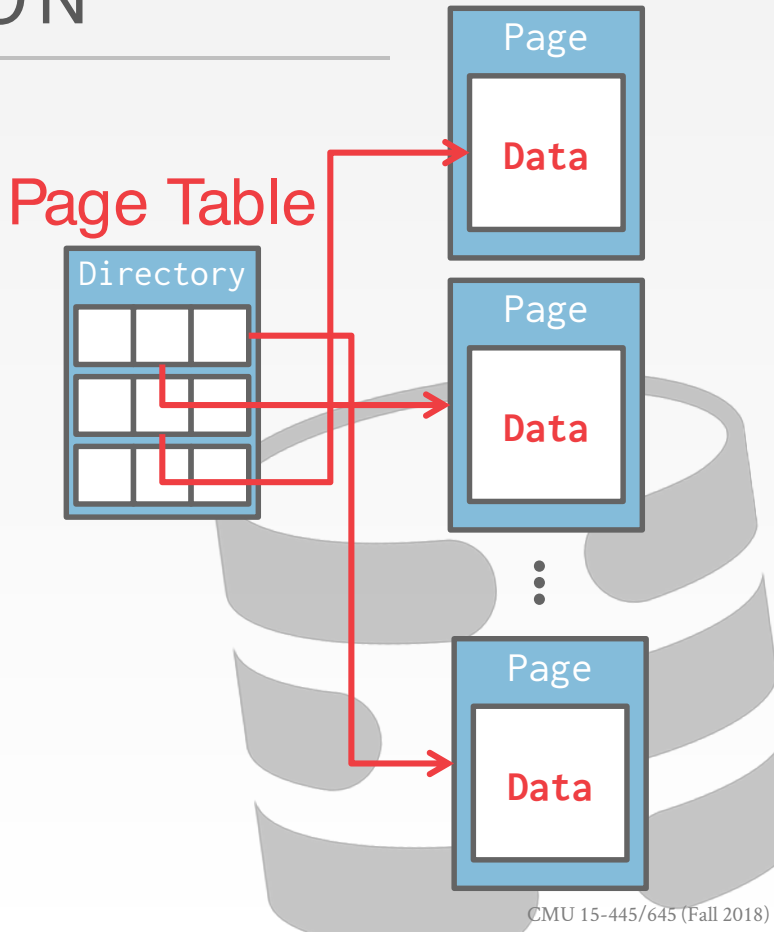
The previous hash tables require knowing the number of elements you want to store ahead of time.

→ Otherwise you have rebuild the entire table if you need to grow/shrink.

Dynamic hash tables are able to grow/shrink on demand.

→ Extendible Hashing

→ Linear Hashing



CHAINED HASHING

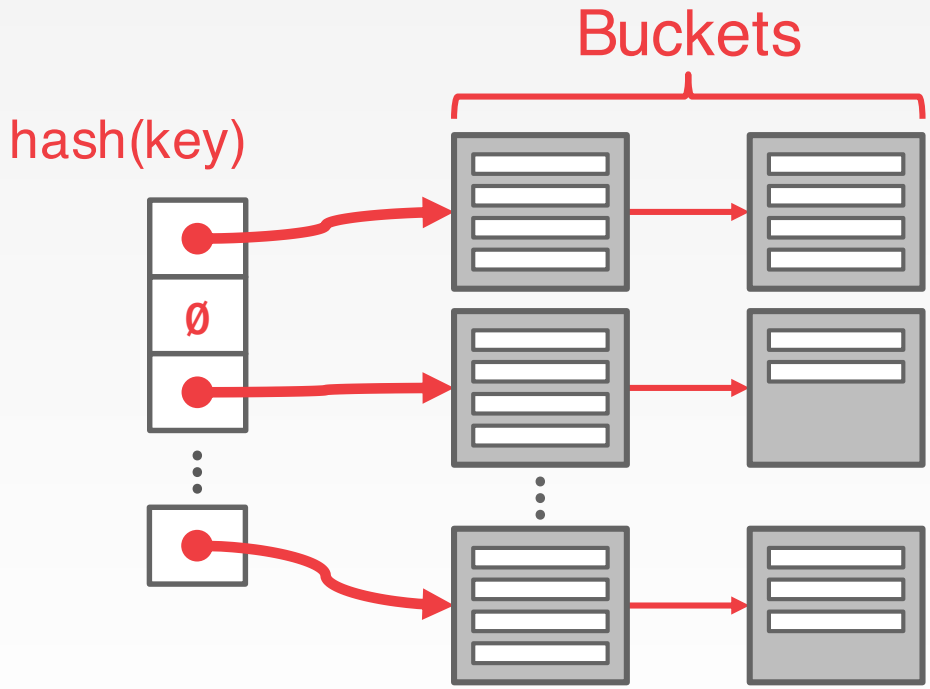
Maintain a linked list of buckets for each slot in the hash table.

Resolve collisions by placing all elements with the same hash key into the same bucket.

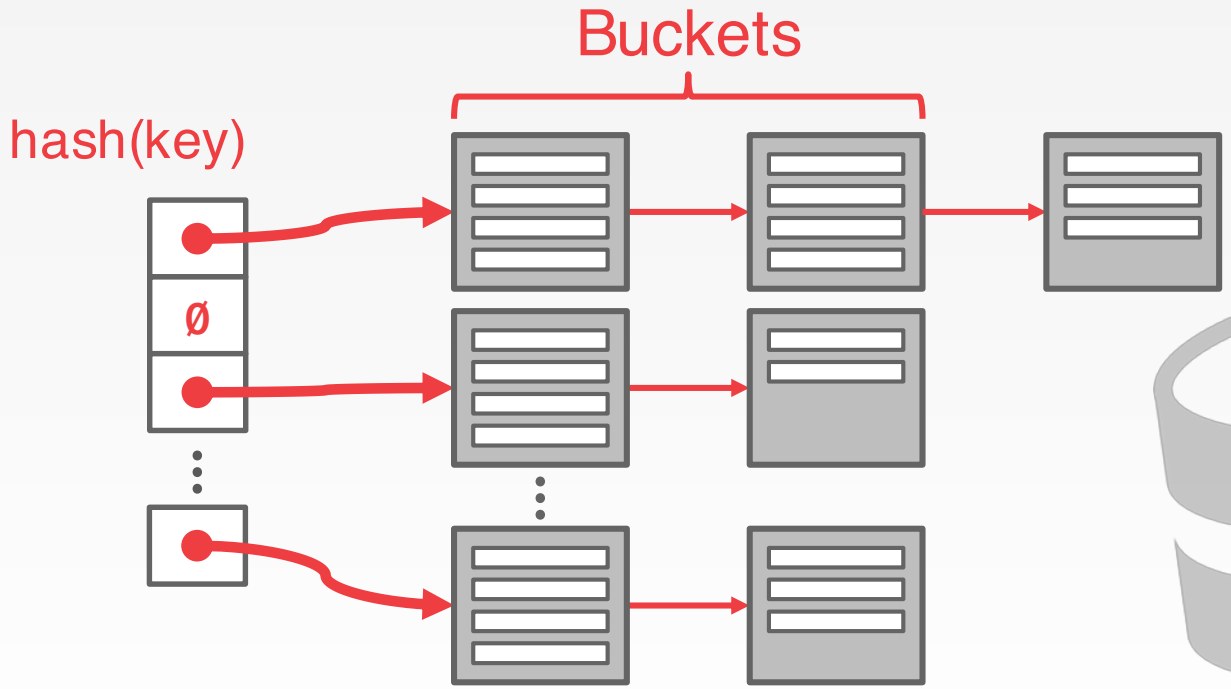
→ To determine whether an element is present, hash to its bucket and scan for it.

→ Insertions and deletions are generalizations of lookups.

CHAINED HASHING



CHAINED HASHING

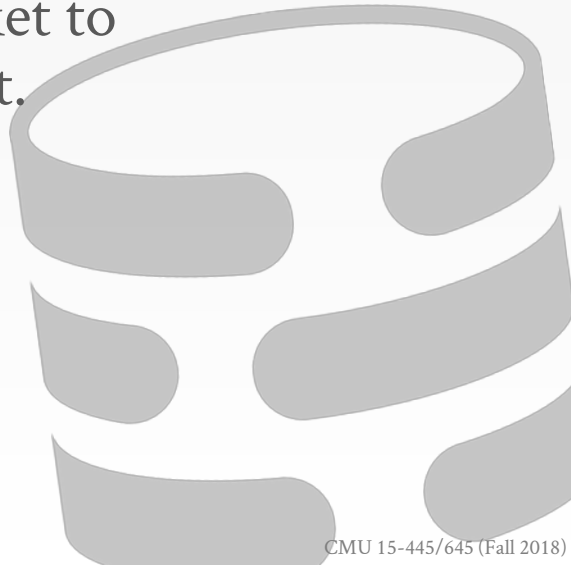


CHAINED HASHING

The hash table can grow infinitely because you just keep adding new buckets to the linked list.

You only need to take a latch on the bucket to store a new entry or extend the linked list.

缺点：当 buckets 数量增长到很大，
扩容时 rehash/rebuild 就会阻塞其他操作较长时间



EXTENDIBLE HASHING

提供更细粒度的锁

Chained-hashing approach where we split buckets instead of letting the linked list grow forever.

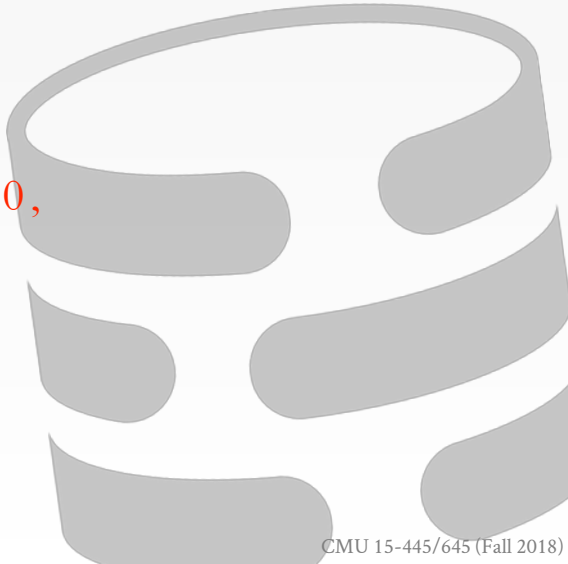
This requires reshuffling entries on split, but the change is localized.

两级， 第一级用于路由；

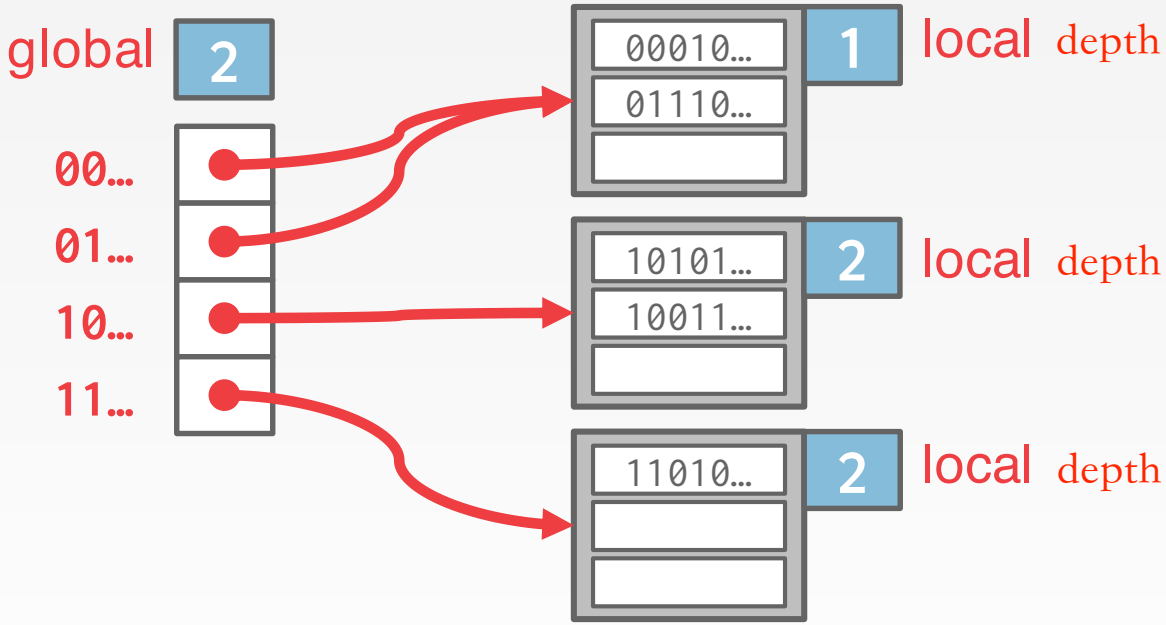
global depth：用于标记需要检查的bit 数，比如，hash 值为011110，全局深度为2， 即取01去查找 bucket;

local depth：代表 split 次数;

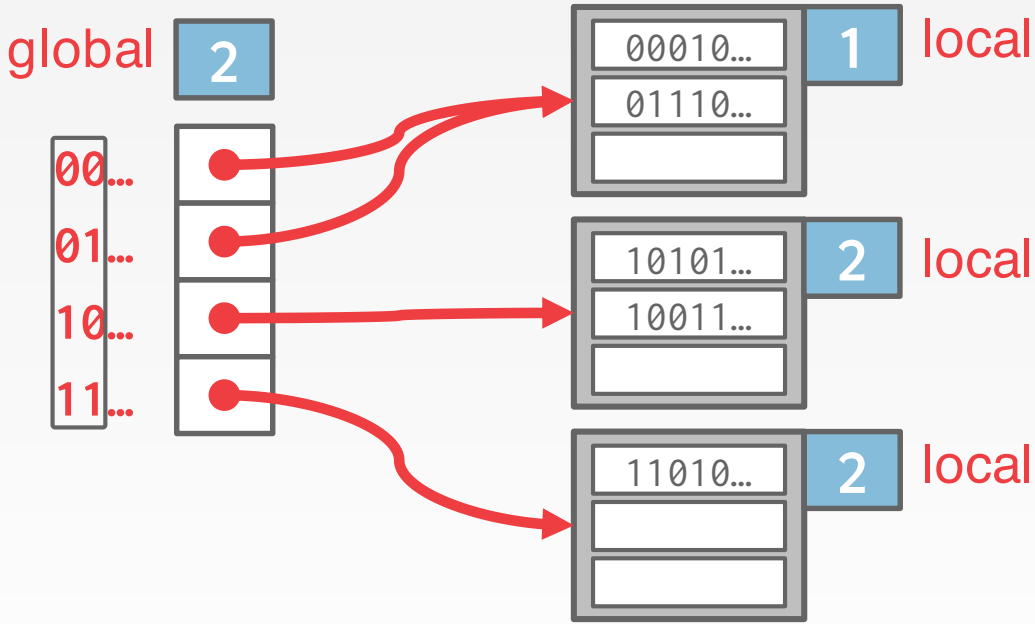
global depth = max(local depth)



EXTENDIBLE HASHING

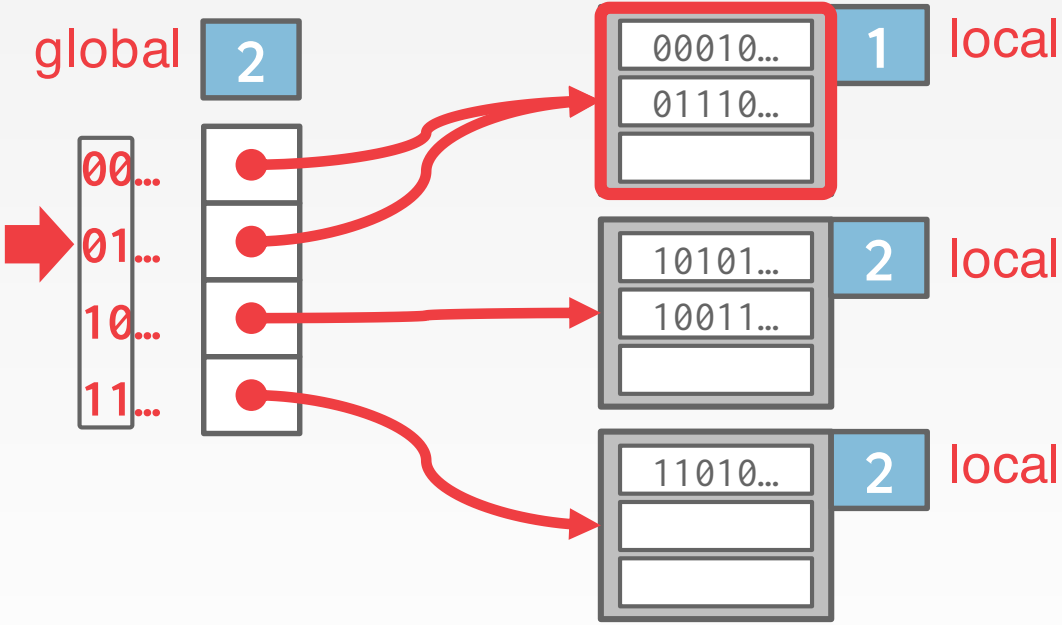


EXTENDIBLE HASHING



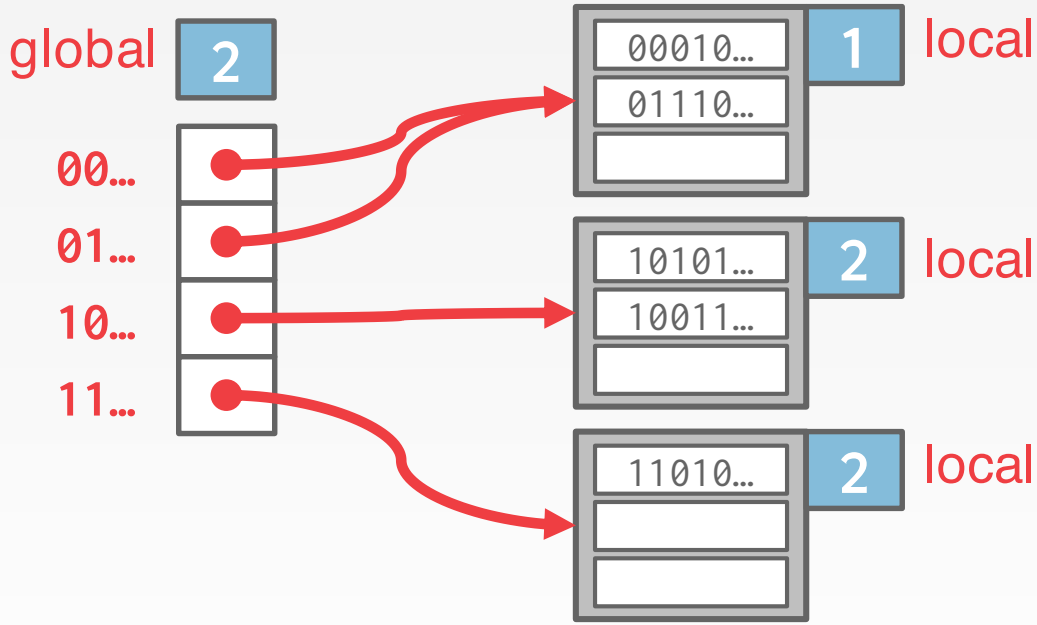
Find A
hash(A) = 01110...

EXTENDIBLE HASHING



Find A
hash(A) = 01110...

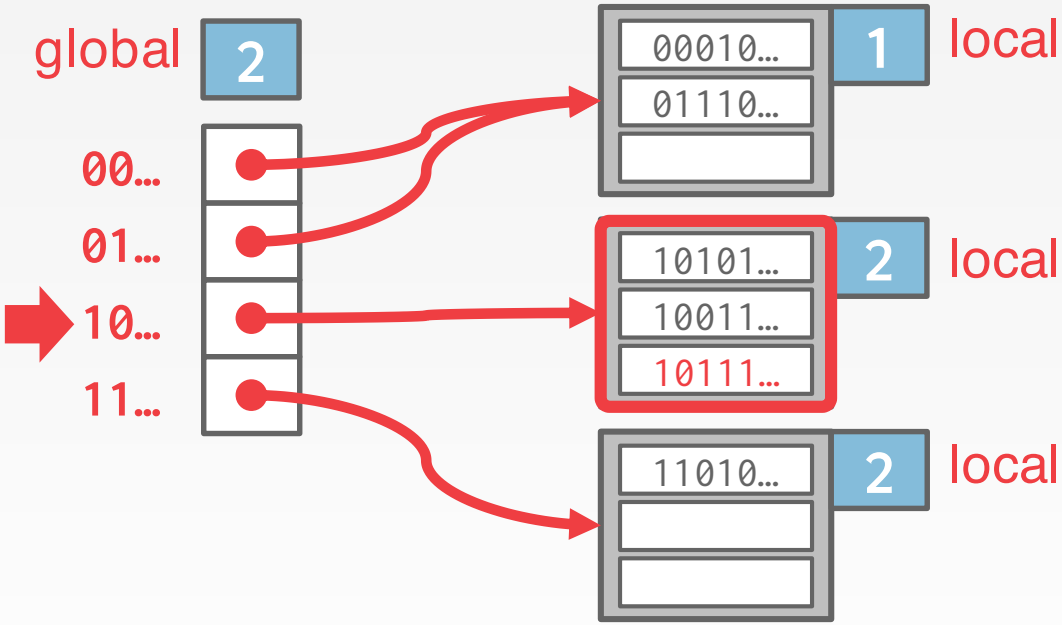
EXTENDIBLE HASHING



Find A
hash(A) = 01110…

Insert B
hash(B) = 10111…

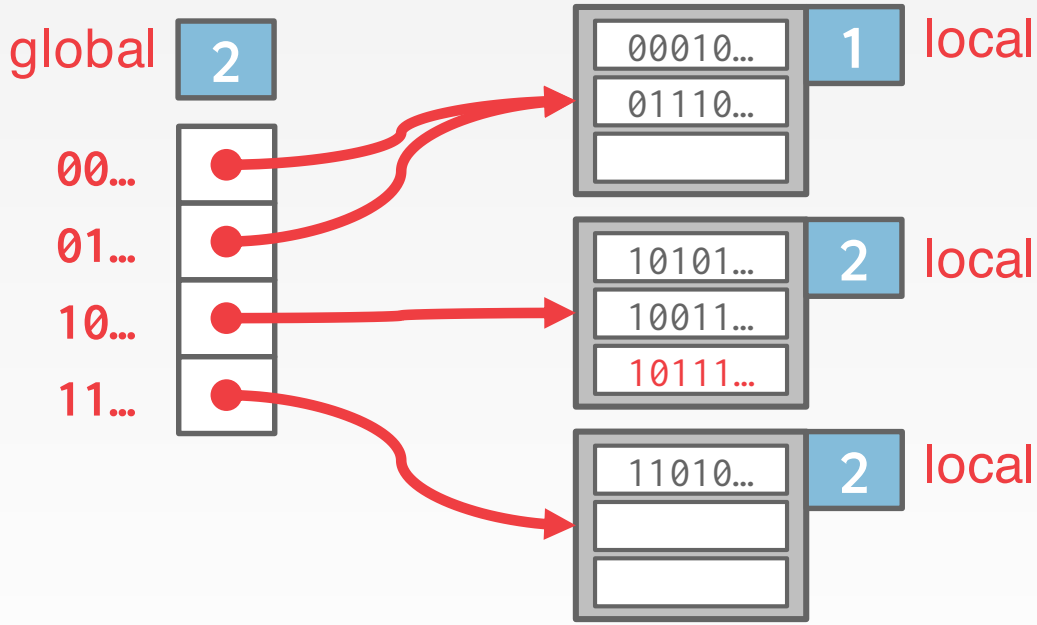
EXTENDIBLE HASHING



Find A
hash(A) = 01110...

Insert B
hash(B) = 10111...

EXTENDIBLE HASHING

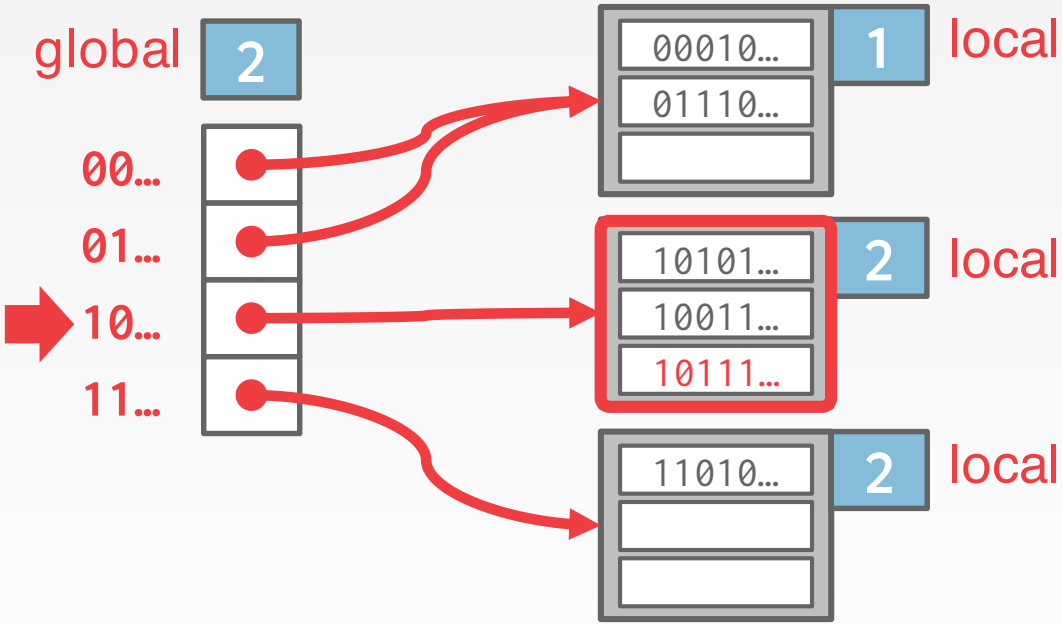


Find A
hash(A) = 01110…

Insert B
hash(B) = 10111…

Insert C
hash(C) = 10100…

EXTENDIBLE HASHING

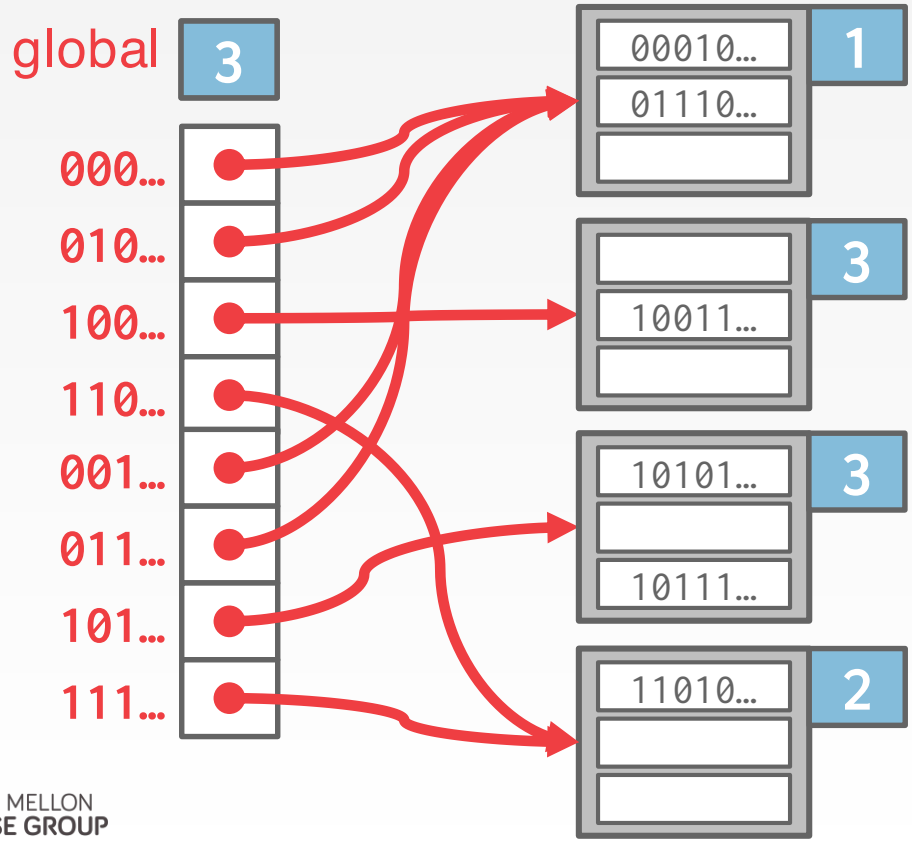


Find A
hash(A) = 01110…

Insert B
hash(B) = 10111…

Insert C
hash(C) = 10100…

EXTENDIBLE HASHING

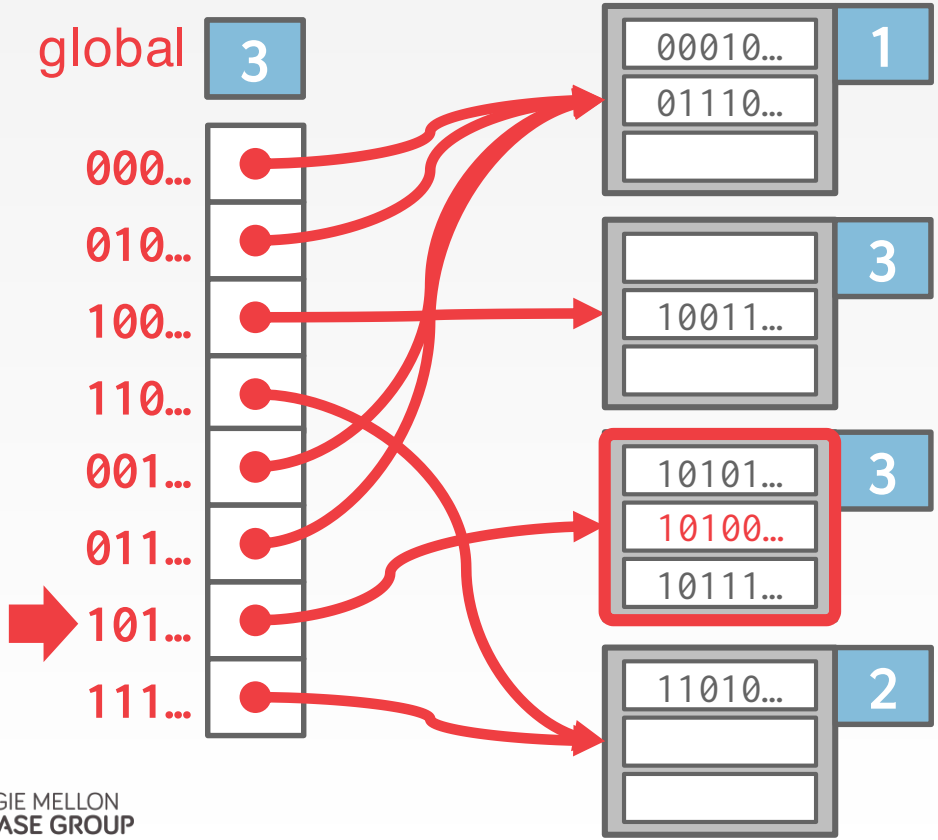


Find A
hash(A) = 01110...

Insert B
hash(B) = 10111...

Insert C
hash(C) = 10100...

EXTENDIBLE HASHING



Find A
hash(A) = 01110...

Insert B
hash(B) = 10111...

Insert C
hash(C) = 10100...

LINEAR HASHING

Maintain a pointer that tracks the next bucket to split.

When any bucket overflows, split the bucket at the pointer location.

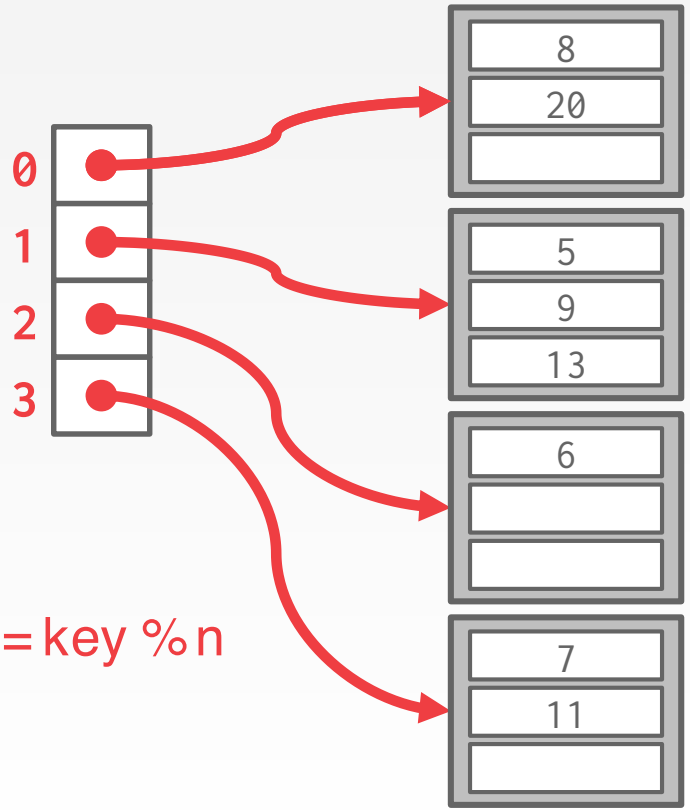
Overflow criterion is left up to the implementation.

多 hash function，split 的时候会用到

→ Space Utilization
→ Average Length of Overflow Chains

LINEAR HASHING

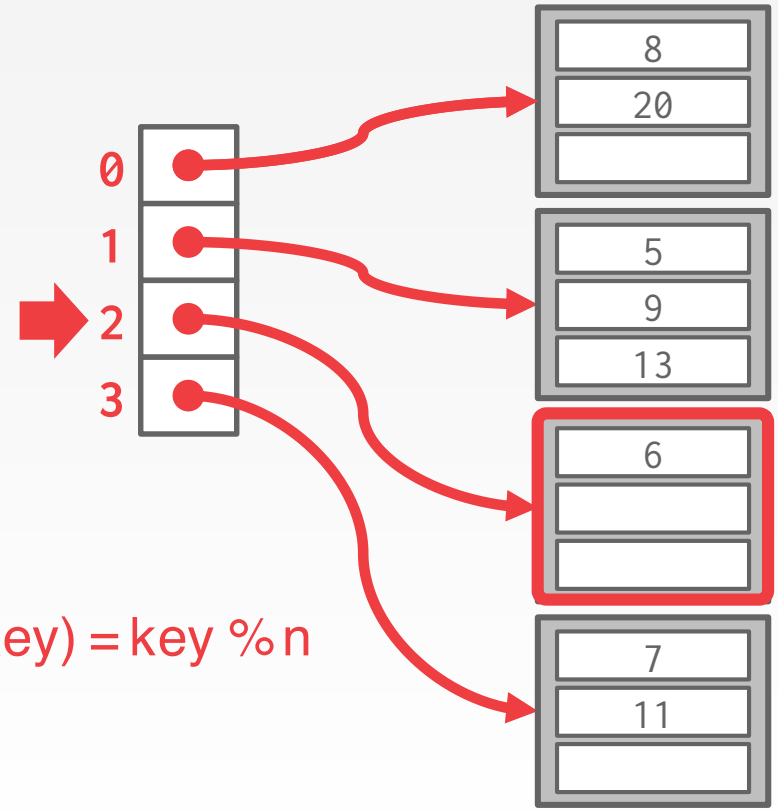
Split Pointer



$hash_1(key) = key \% n$

LINEAR HASHING

Split Pointer



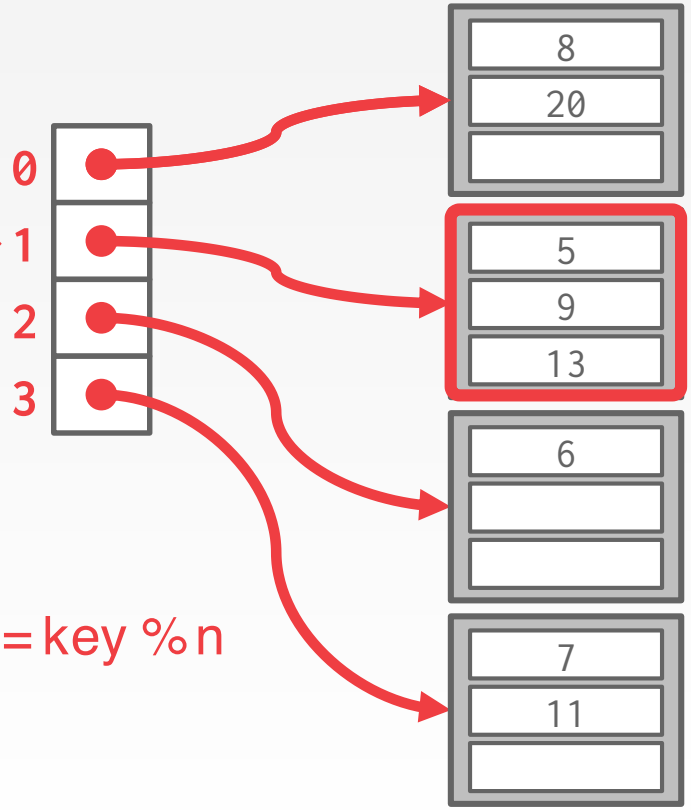
$hash_1(key) = key \% n$

Find 6

$hash_1(6) = 6\%4 = 2$

LINEAR HASHING

Split Pointer



$hash_1(key) = key \% n$

Find 6

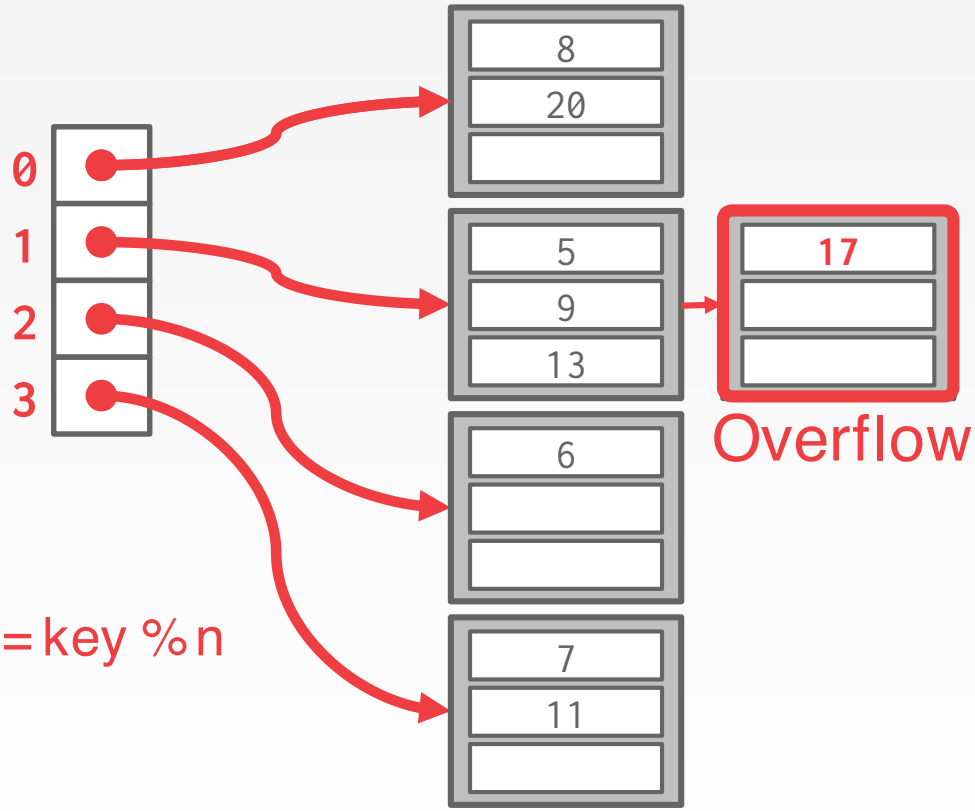
$hash_1(6) = 6\%4 = 2$

Insert 17

$hash_1(17) = 17\%4 = 1$

LINEAR HASHING

Split Pointer



Overflow!

Find 6

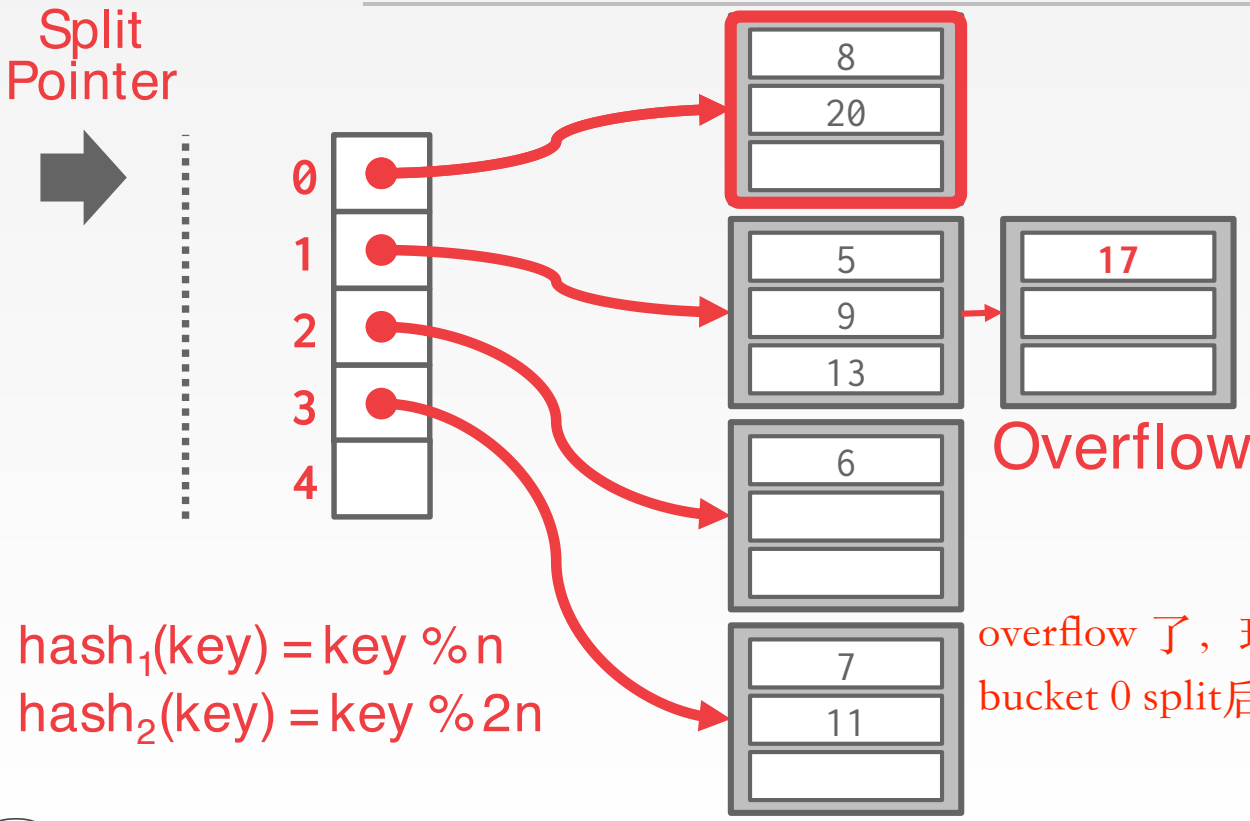
$hash_1(6) = 6\%4 = 2$

Insert 17

$hash_1(17) = 17\%4 = 1$

$hash_1(key) = key \% n$

LINEAR HASHING



Find 6

$hash_1(6) = 6\%4 = 2$

Insert 17

$hash_1(17) = 17\%4 = 1$

$hash_1(key) = key \% n$

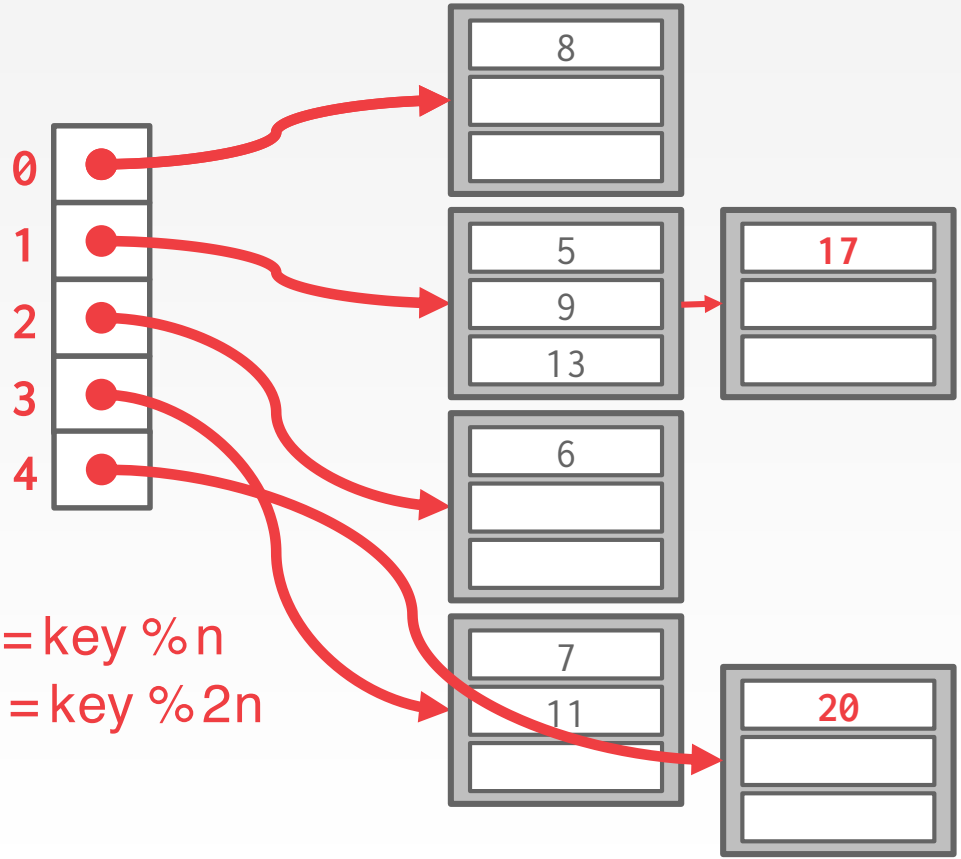
$hash_2(key) = key \% 2n$

overflow 了，现在 split pointer指向的bucket 0，

bucket 0 split后，用 hash2来 rehash 0 里面的元素

LINEAR HASHING

Split Pointer



$hash_1(key) = key \% n$

$hash_2(key) = key \% 2n$

Find 6

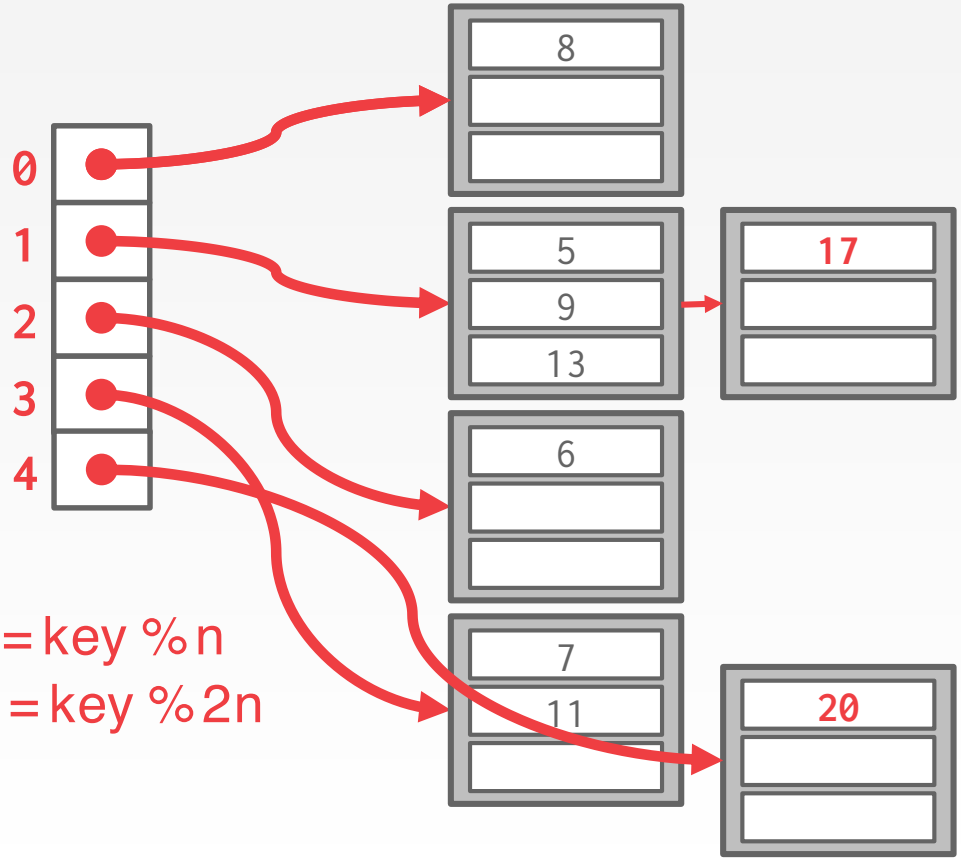
$hash_1(6) = 6\%4 = 2$

Insert 17

$hash_1(17) = 17\%4 = 1$

LINEAR HASHING

Split Pointer

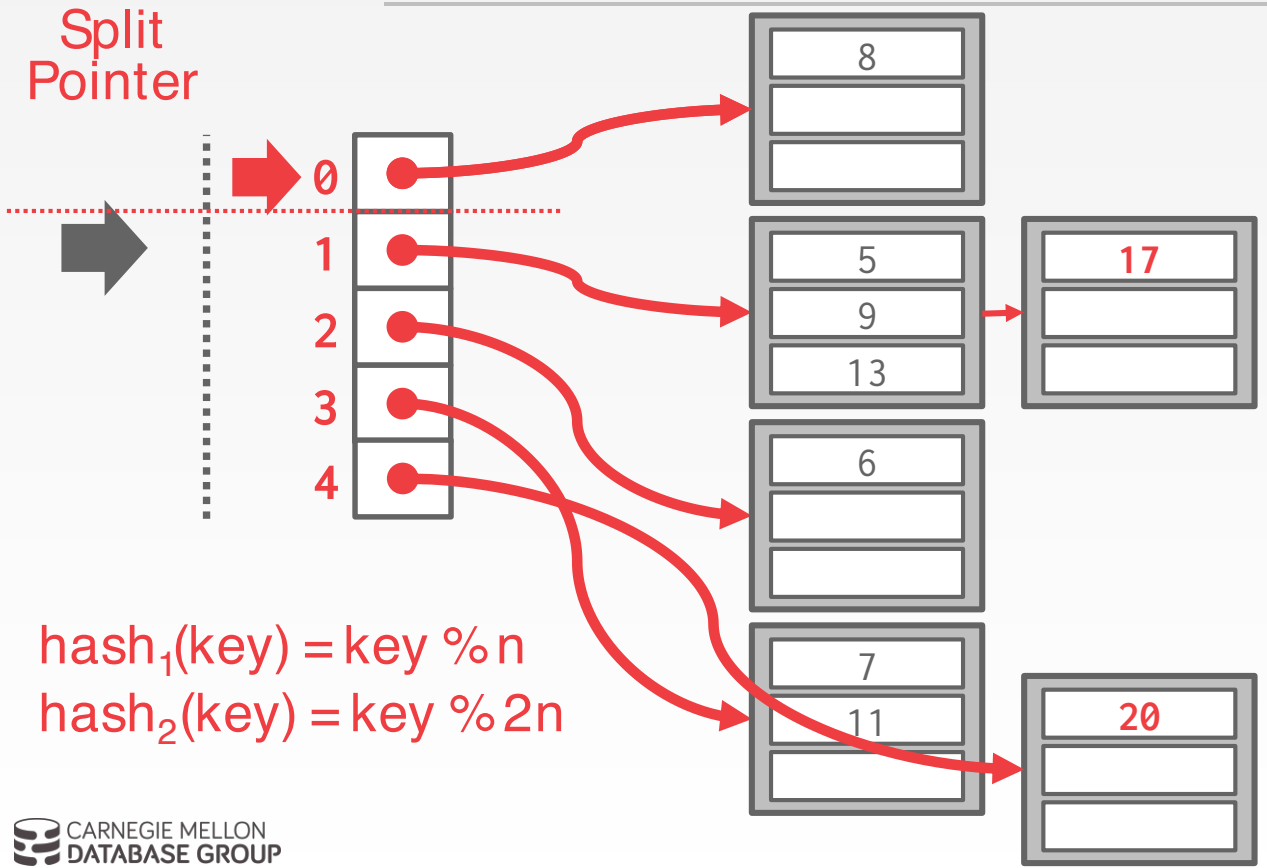


Find 6
$hash_1(6) = 6\%4 = 2$

Insert 17
$hash_1(17) = 17\%4 = 1$

$hash_1(key) = key \% n$
$hash_2(key) = key \% 2n$

LINEAR HASHING



$hash_1(key) = key \% n$

$hash_2(key) = key \% 2n$

Find 6

$hash_1(6) = 6\%4 = 2$

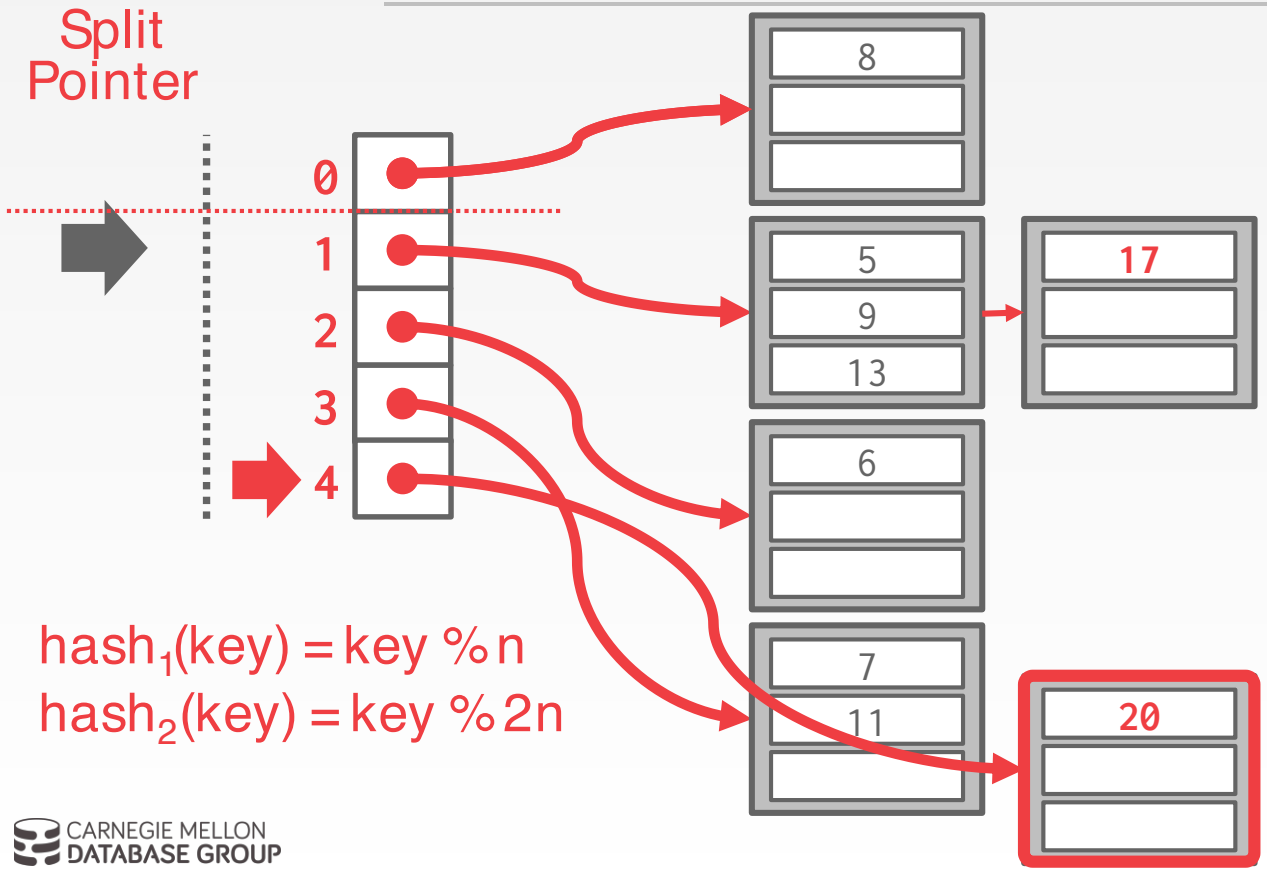
Insert 17

$hash_1(17) = 17\%4 = 1$

Find 20

$hash_1(20) = 20\%4 = 0$

LINEAR HASHING



$hash_1(key) = key \% n$

$hash_2(key) = key \% 2n$

Find 6

$hash_1(6) = 6\%4 = 2$

Insert 17

$hash_1(17) = 17\%4 = 1$

Find 20

$hash_1(20) = 20\%4 = 0$

$hash_2(20) = 20\%8 = 4$

LINEAR HASHING

Splitting buckets based on the split pointer will eventually get to all overflowed buckets.

→ When the pointer reaches the last slot, delete the first hash function and move back to beginning.

The pointer can also move backwards when buckets are empty.

数据倾斜严重，一直创建 overflow bucket，
就会一直 split，性能会很差

优点：未知。插入速度快?

CONCLUSION

Fast data structures that support O(1) look-ups that are used all throughout the DBMS internals.
→ Trade-off between speed and flexibility.

Hash tables are usually **not** what you want to use for a table index…

range scan / partial keys

Postgres Demo

NEXT CLASS

B+Trees

Skip Lists

Radix Trees