Carnegie Mellon University

06 Hash Tables

Intro to Database Systems
15-445/15-645
Fall 2020

AP
Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Project #1 is due Sunday Sept 27th

Q&A Session about the project on **Monday Sept 21st @ 8:00pm ET**.

→ In-Person: GHC 4401

→ https://cmu.zoom.us/j/98100285498?pwd=a011L0E2eWFwTndKMG9KNVhzb2tDdz09

UPCOMING DATABASE TALKS

Snowflake Query Optimizer
→ Today @ 5pm ET

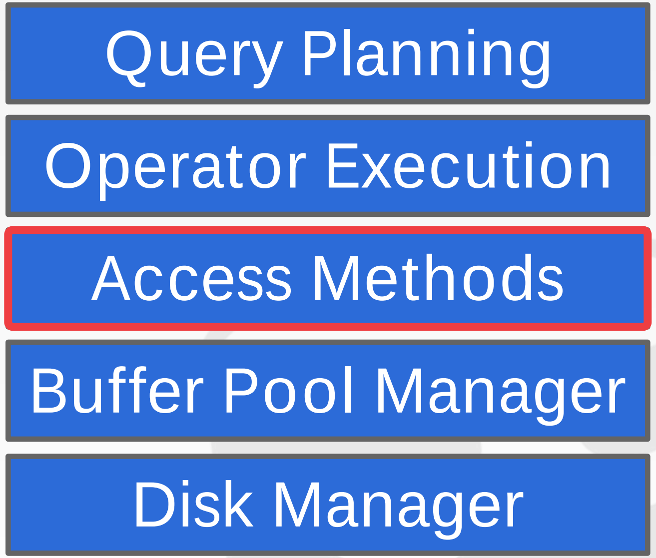
CockroachDB Query Optimizer
→ Monday Sept 28th @ 5pm ET

Apache Arrow
→ Monday Oct 5th @ 5pm

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

Core Data Storage

Temporary Data Structures

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 unordered associative array that maps keys to values.

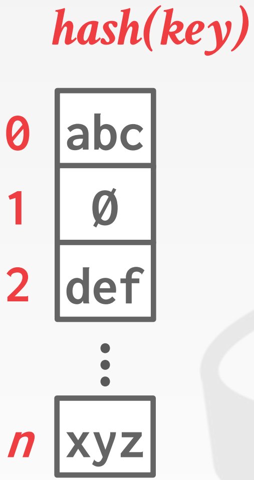
It uses a **hash function** to compute an offset into the array for a given key, from which the desired value can be found.

Space Complexity: **O(n)**
Operation Complexity:
→ Average: **O(1)** ⬅ ***Money cares about constants!***
→ Worst: **O(n)**

STATIC HASH TABLE

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

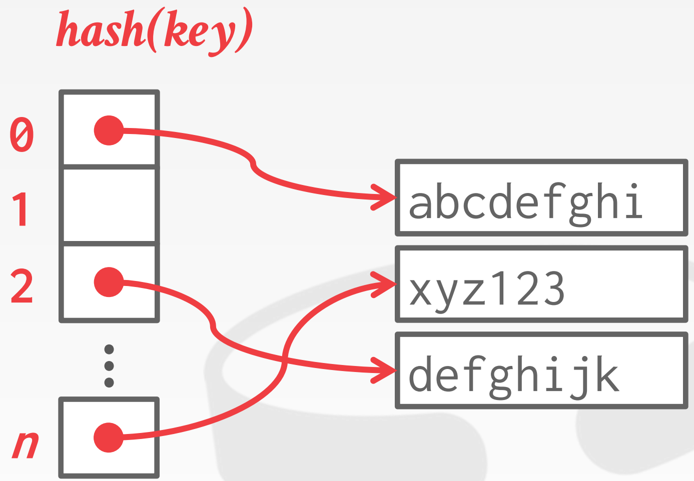
To find an entry, mod the key by the number of elements to find the offset in the array.



STATIC HASH TABLE

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

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



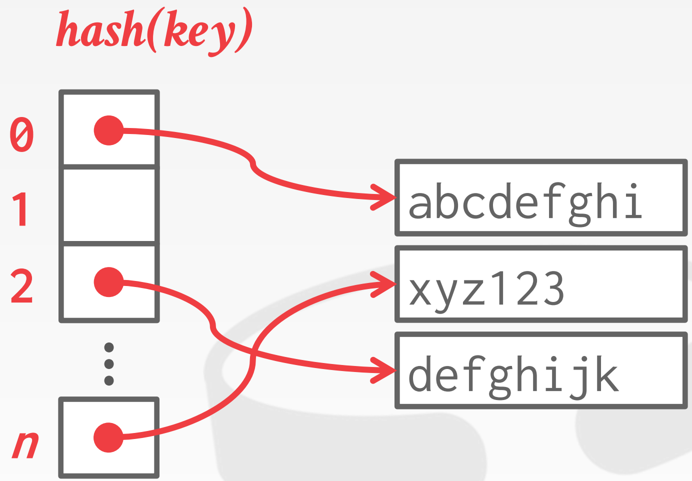
ASSUMPTIONS

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

Design Decision #1: Hash Function

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

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.

TODAY'S AGENDA

Hash Functions
Static Hashing Schemes
Dynamic Hashing Schemes

HASH FUNCTIONS

For any input key, return an integer representation of that key.

We do not want to use a cryptographic hash function for DBMS hash tables.

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

HASH FUNCTIONS

CRC-64 (1975)

→ Used in networking for error detection.

MurmurHash (2008)

→ Designed to a fast, general purpose hash function.

Google CityHash (2011)

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

Facebook XXHash (2012)

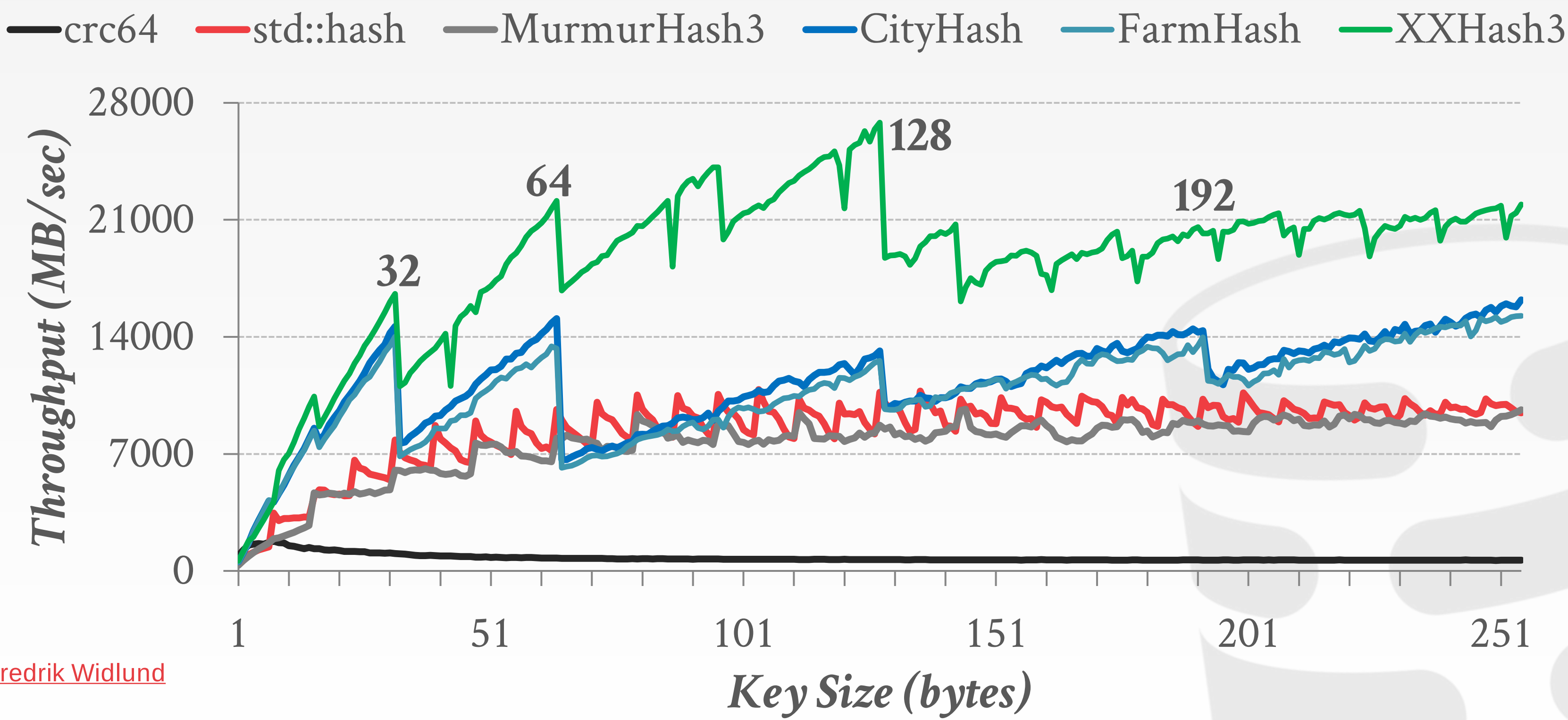
→ From the creator of zstd compression.

Google FarmHash (2014)

→ Newer version of CityHash with better collision rates.

HASH FUNCTION BENCHMARK

Intel Core i7-8700K @ 3.70GHz



Source: Fredrik Widlund

STATIC HASHING SCHEMES

Approach #1: Linear Probe Hashing

Approach #2: Robin Hood Hashing

Approach #3: Cuckoo Hashing

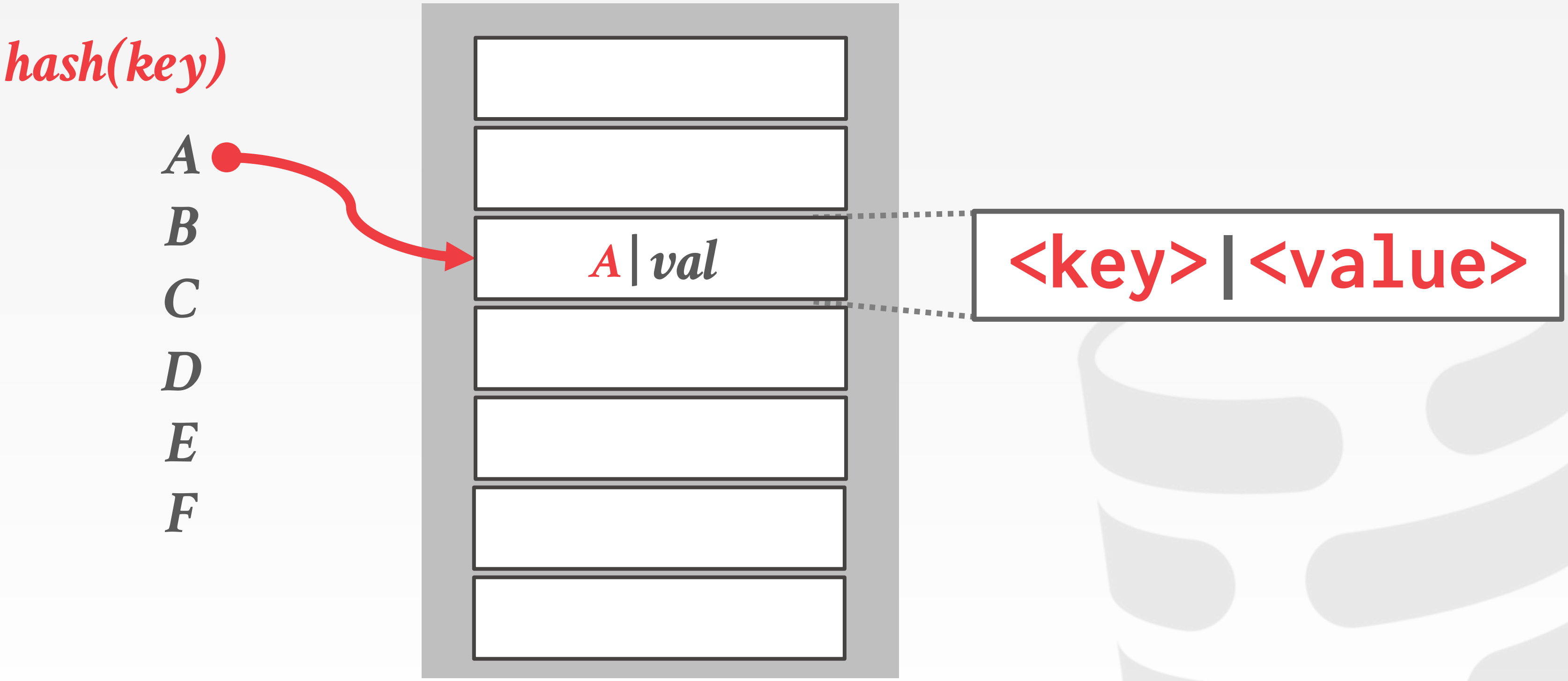
LINEAR PROBE HASHING

Single giant table of slots.

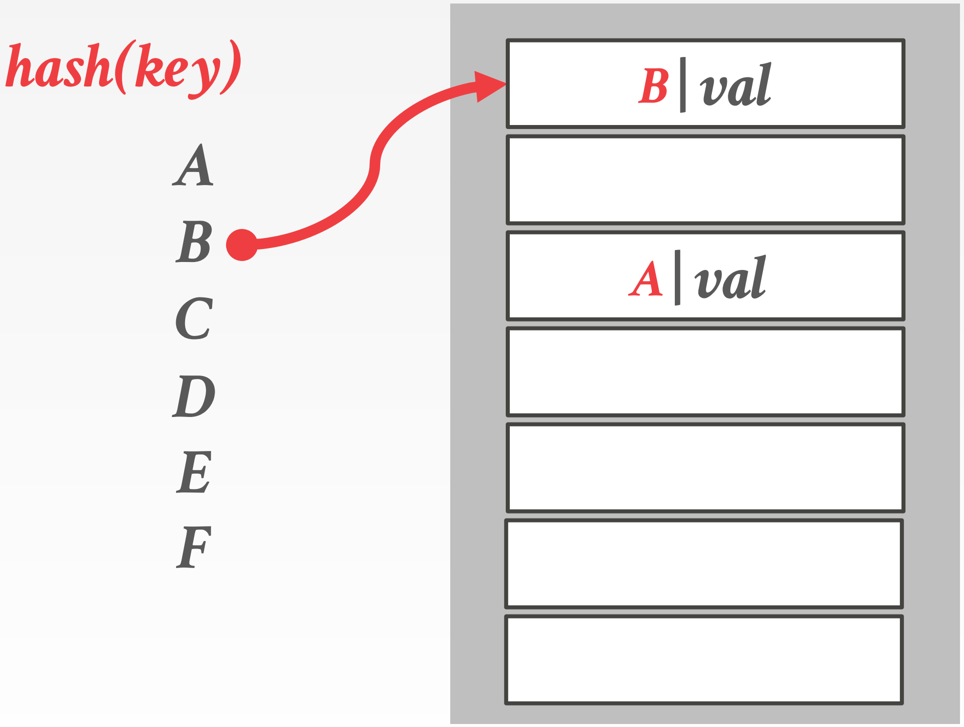
Resolve collisions by linearly searching for the next free slot in the table.

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

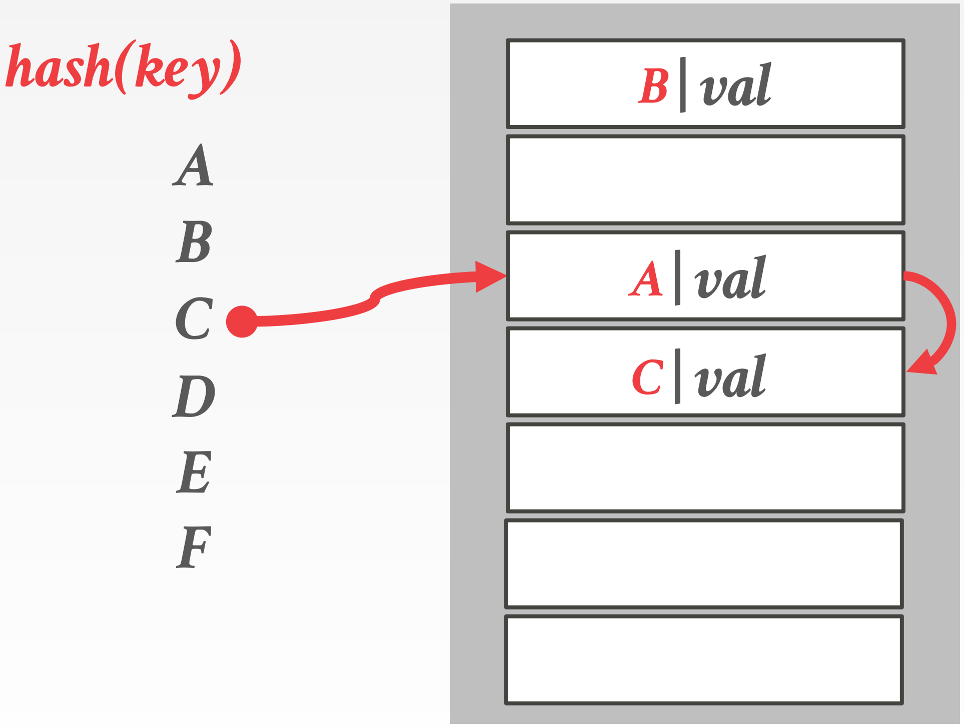
LINEAR PROBE HASHING



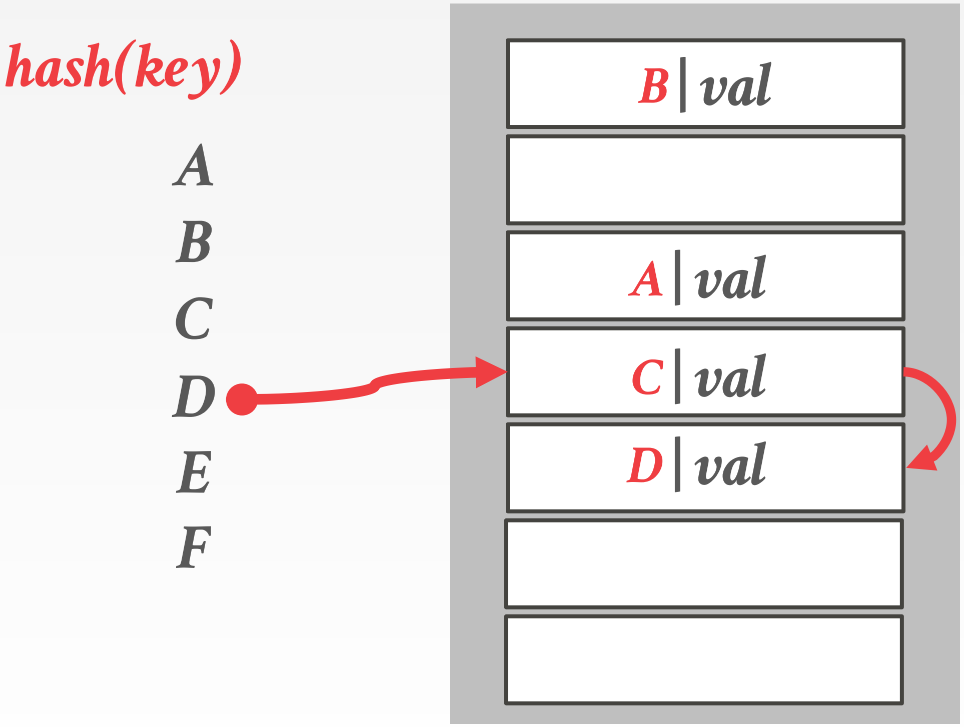
LINEAR PROBE HASHING



LINEAR PROBE HASHING



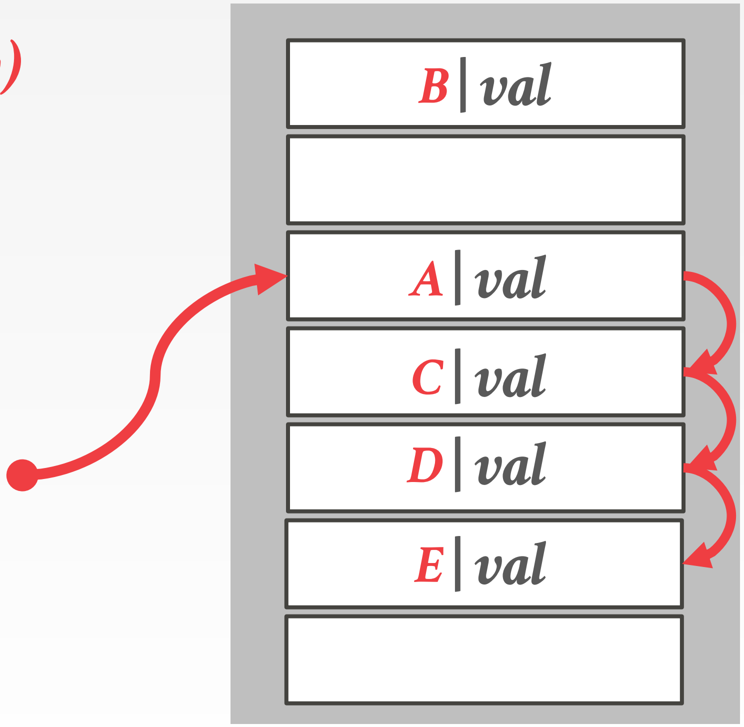
LINEAR PROBE HASHING



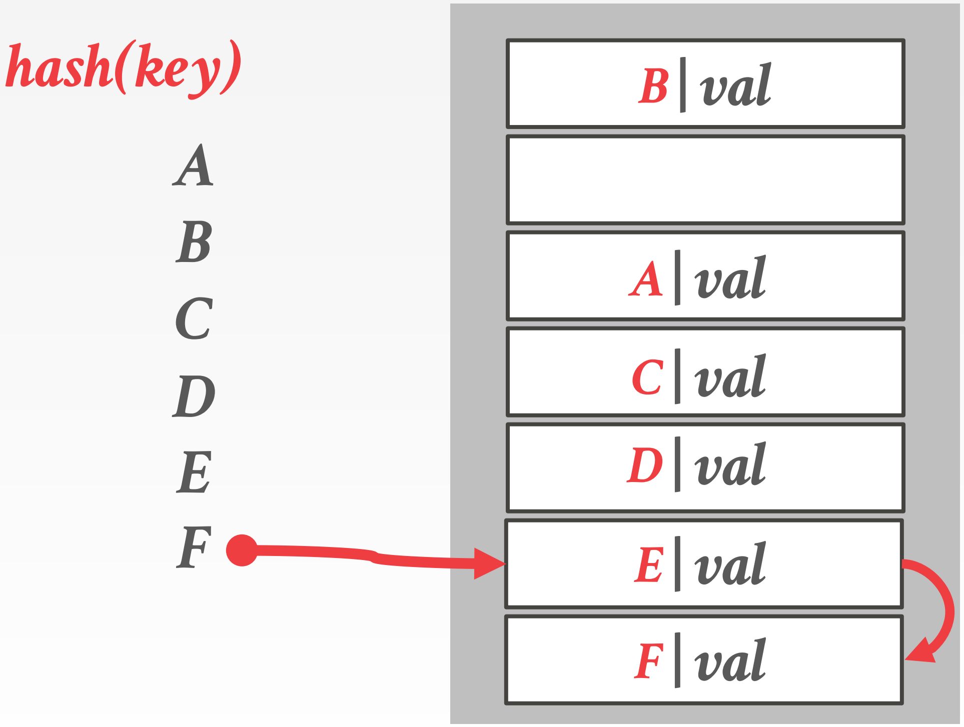
LINEAR PROBE HASHING

hash(key)

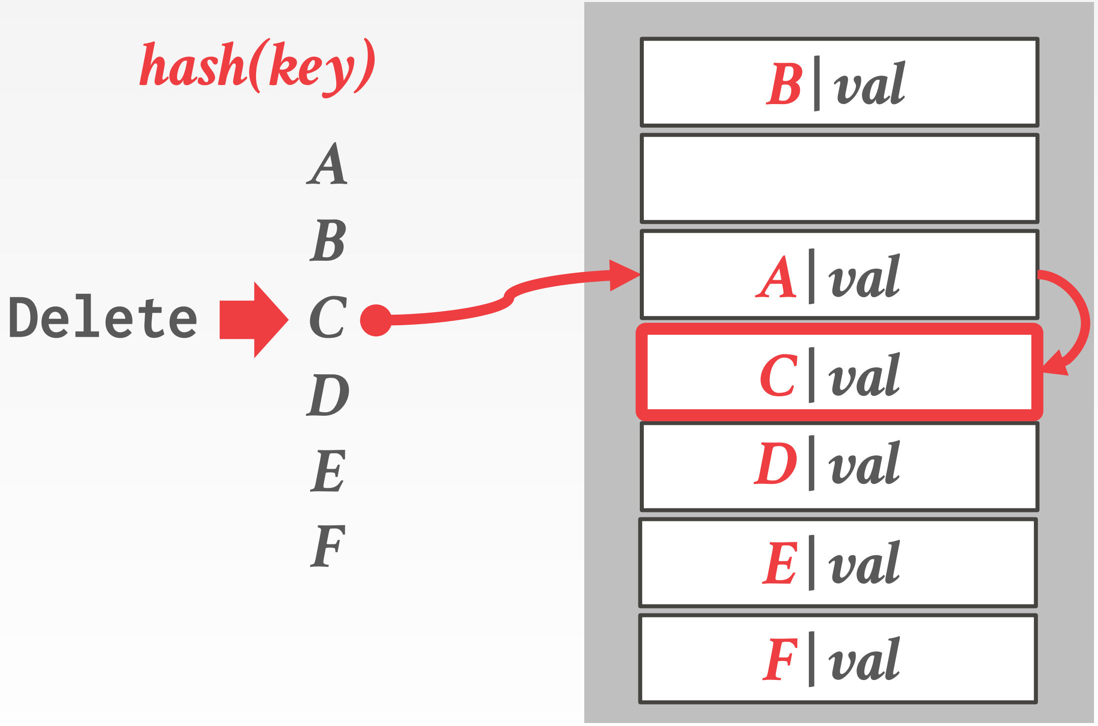
A
B
C
D
E
F



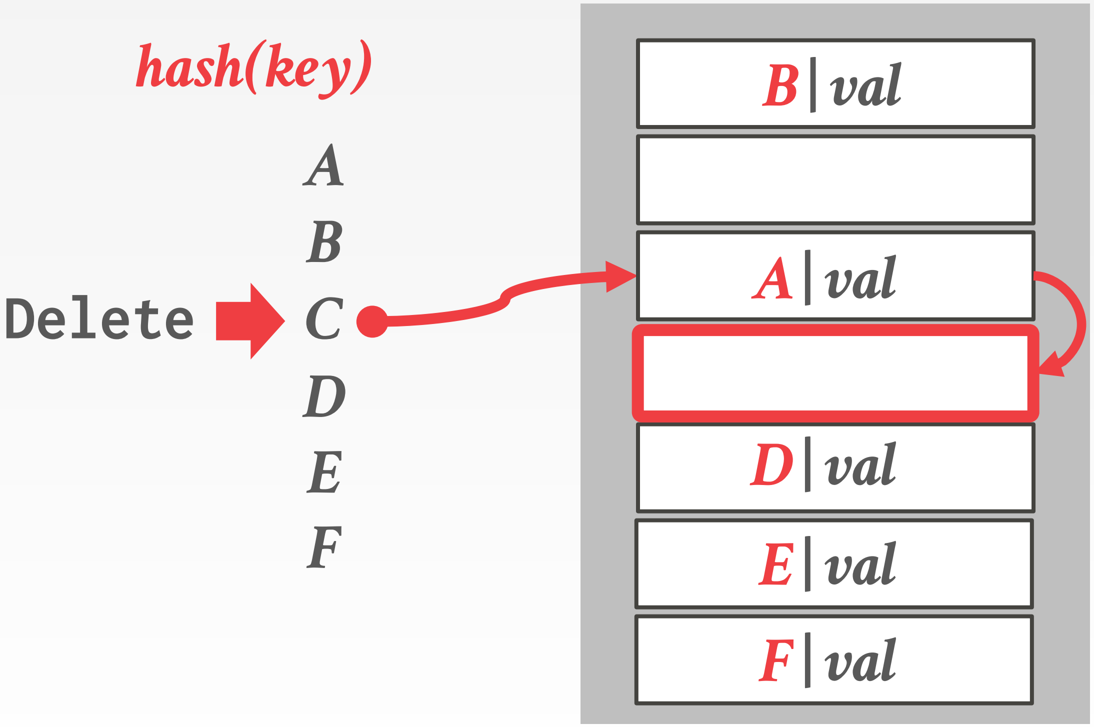
LINEAR PROBE HASHING



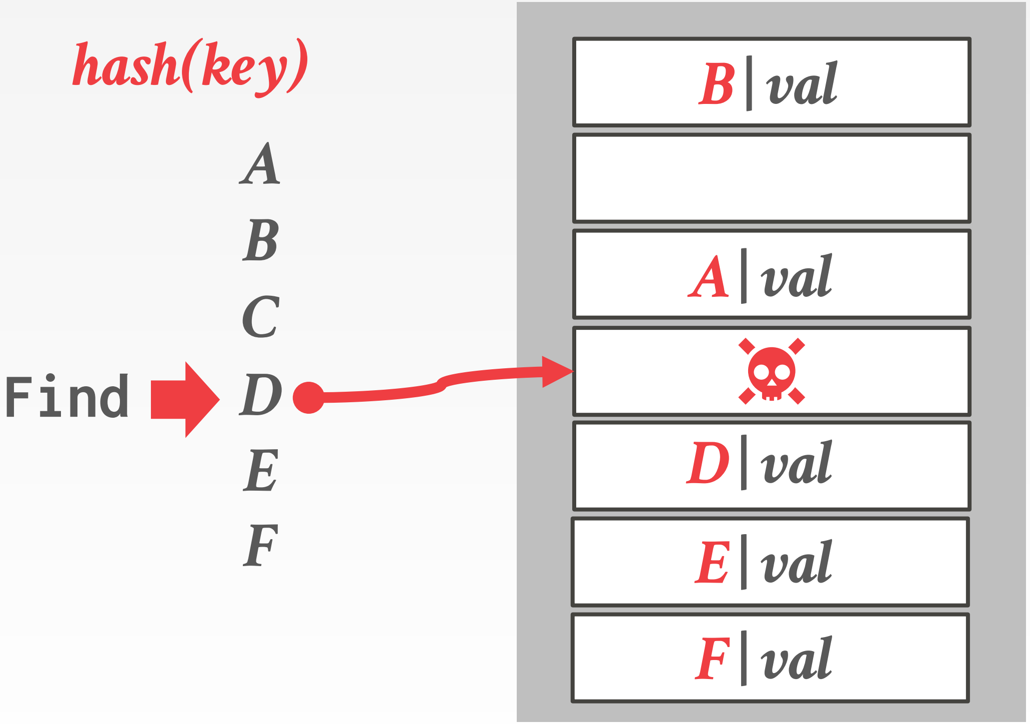
LINEAR PROBE HASHING – DELETES



LINEAR PROBE HASHING – DELETES



LINEAR PROBE HASHING – DELETES



LINEAR PROBE HASHING – DELETES

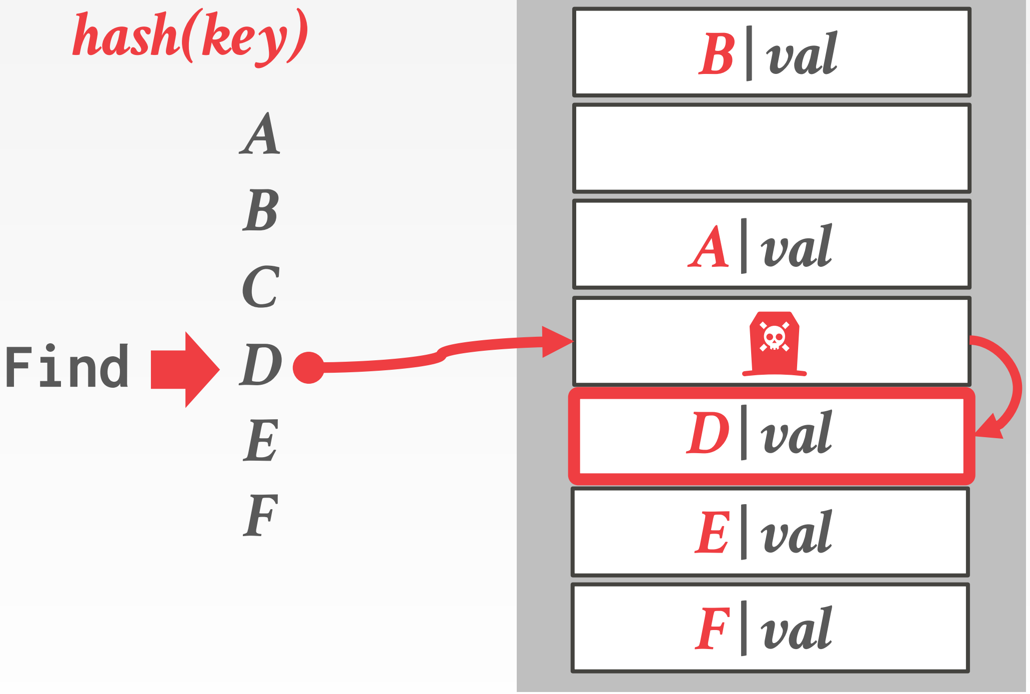
hash(key)

A
B
C
Find → *D*
E
F

B | *val*
A | *val*
D | *val*
E | *val*
F | *val*

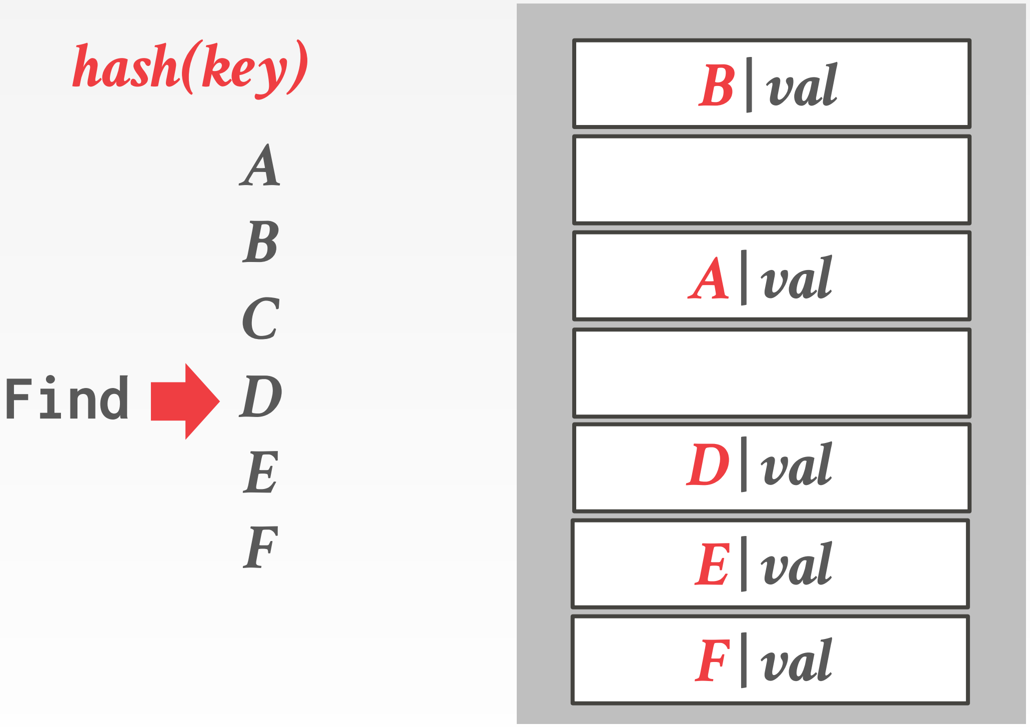
Approach #1: Tombstone

LINEAR PROBE HASHING – DELETES



Approach #1: Tombstone

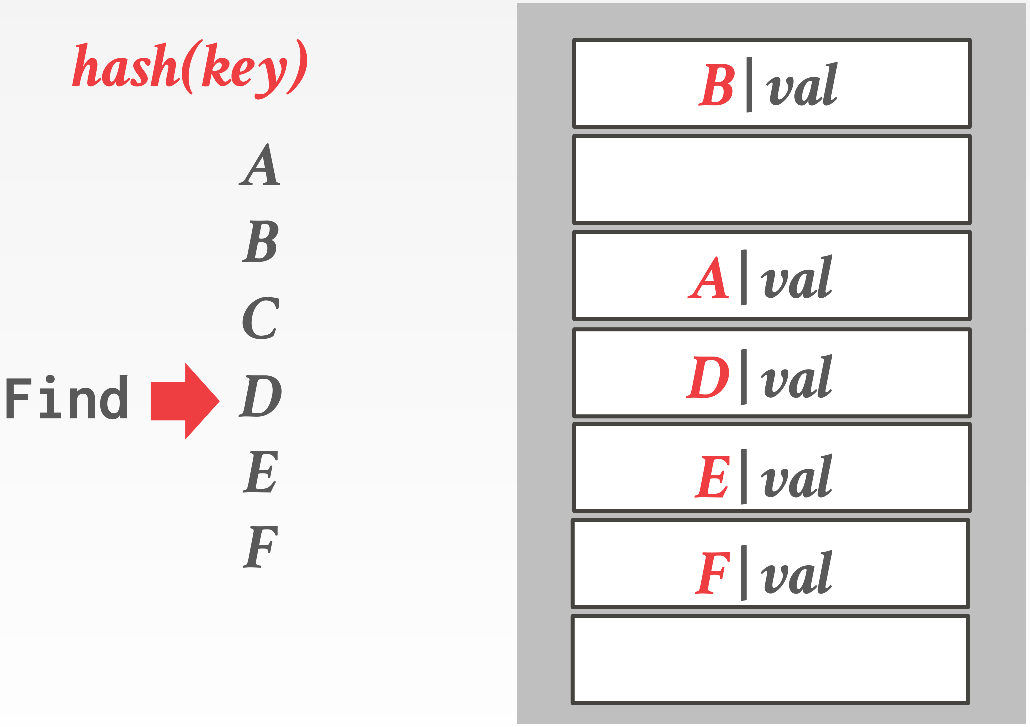
LINEAR PROBE HASHING – DELETES



Approach #1: Tombstone

Approach #2: Movement

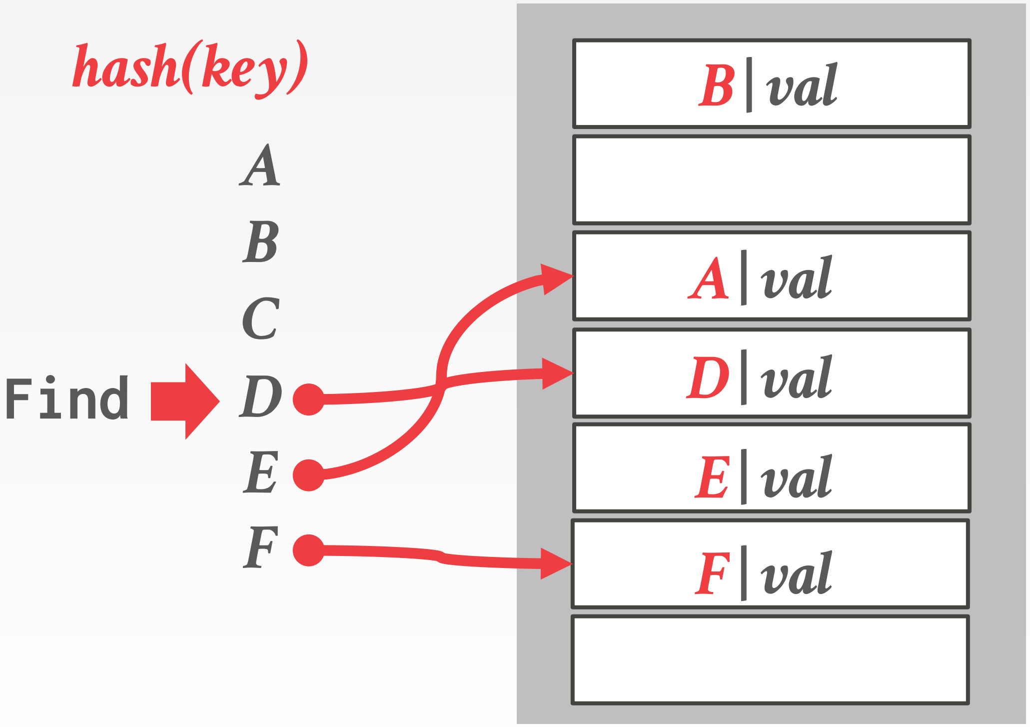
LINEAR PROBE HASHING – DELETES



Approach #1: Tombstone

Approach #2: Movement

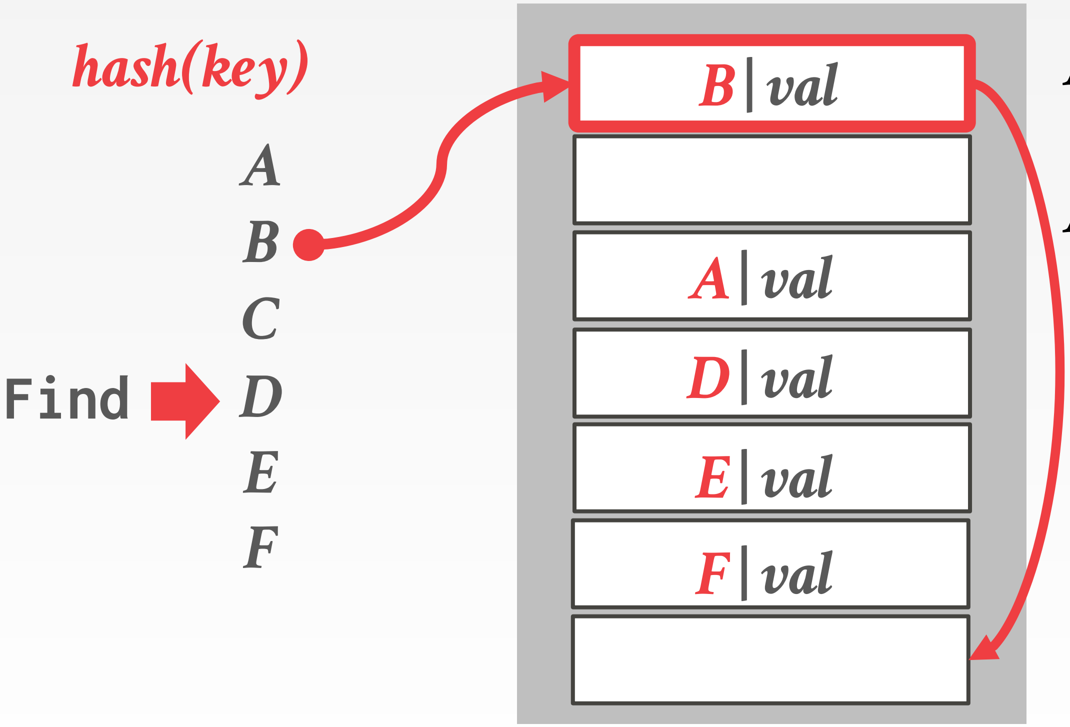
LINEAR PROBE HASHING – DELETES



Approach #1: Tombstone

Approach #2: Movement

LINEAR PROBE HASHING – DELETES



Approach #1: Tombstone

Approach #2: Movement

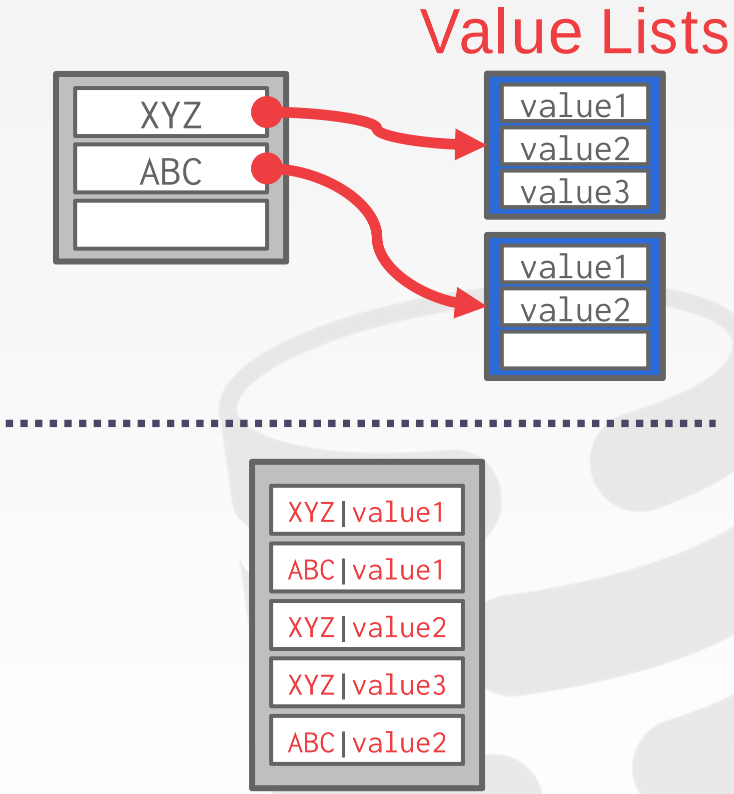
NON-UNIQUE KEYS

Choice #1: Separate Linked List

→ Store values in separate storage area for each key.

Choice #2: Redundant Keys

→ Store duplicate keys entries together in the hash table.



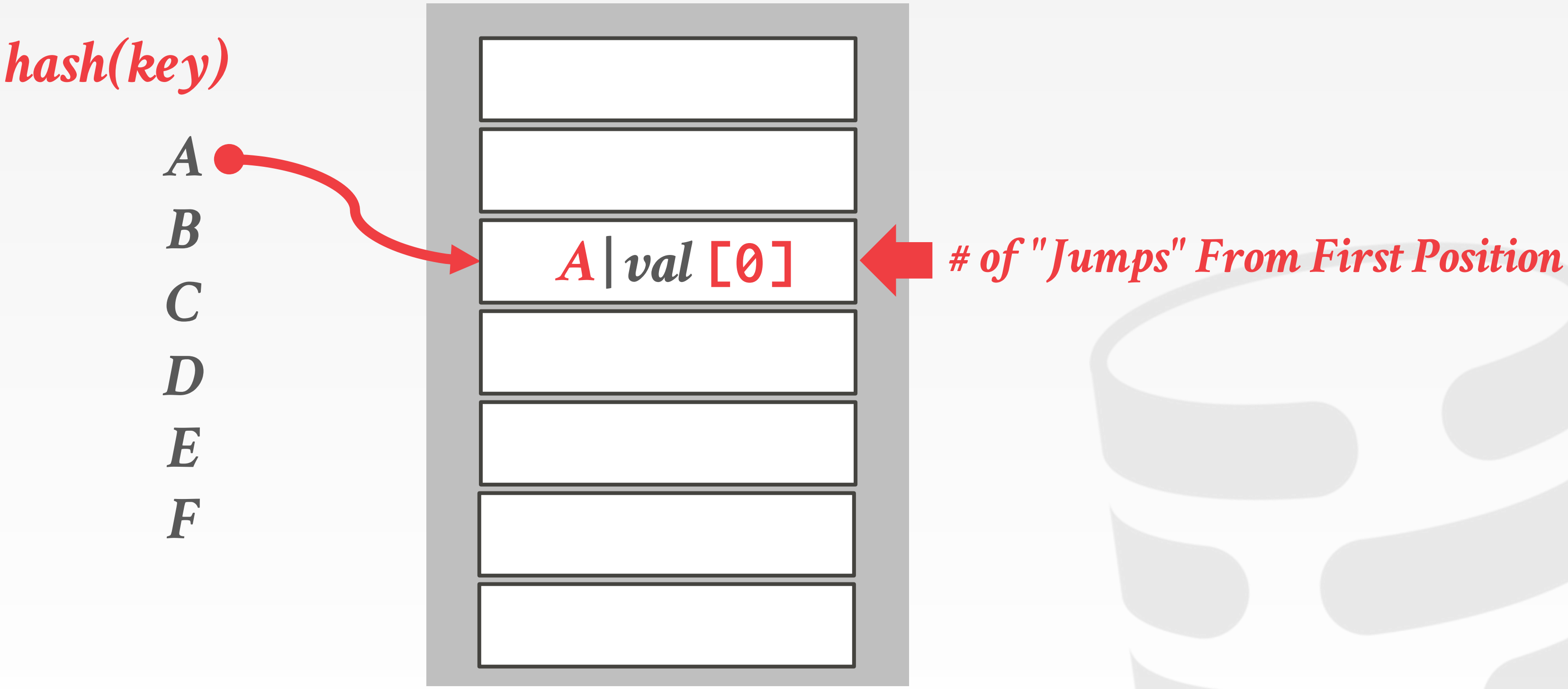
ROBIN HOOD HASHING

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

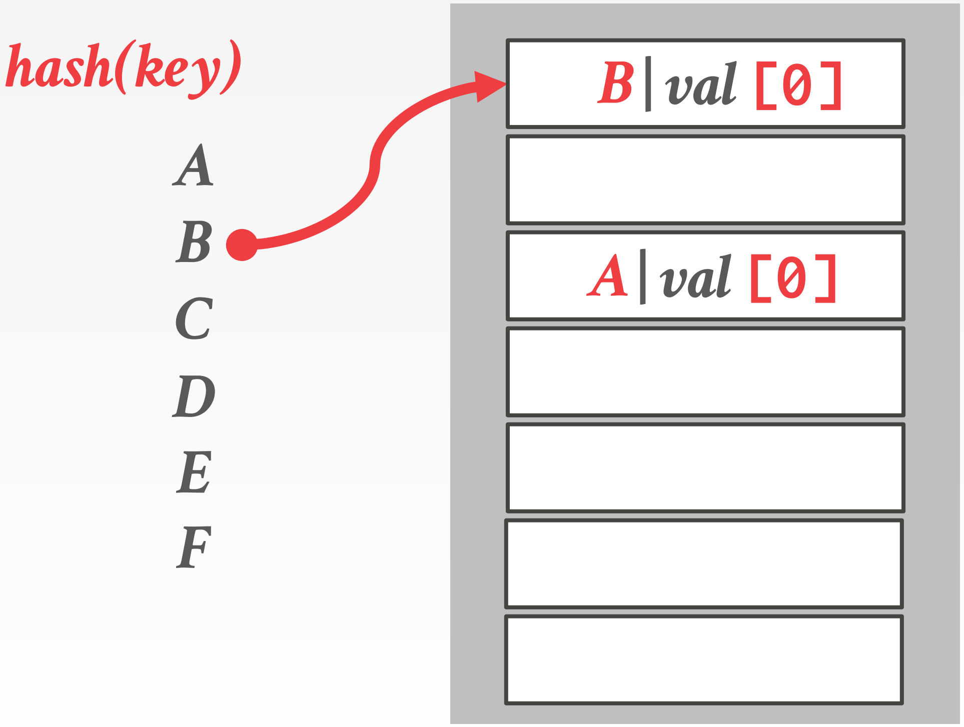
→ Each key tracks the number of positions they are from where its optimal position in the table.

→ 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.

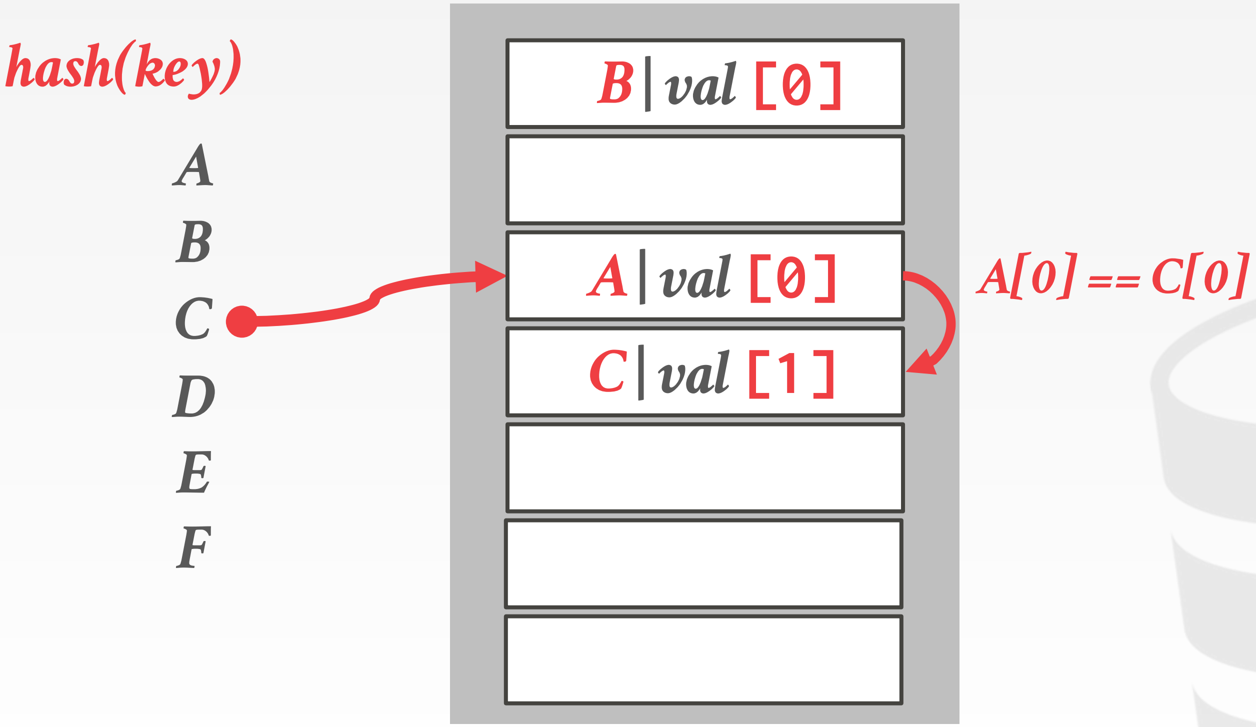
ROBIN HOOD HASHING



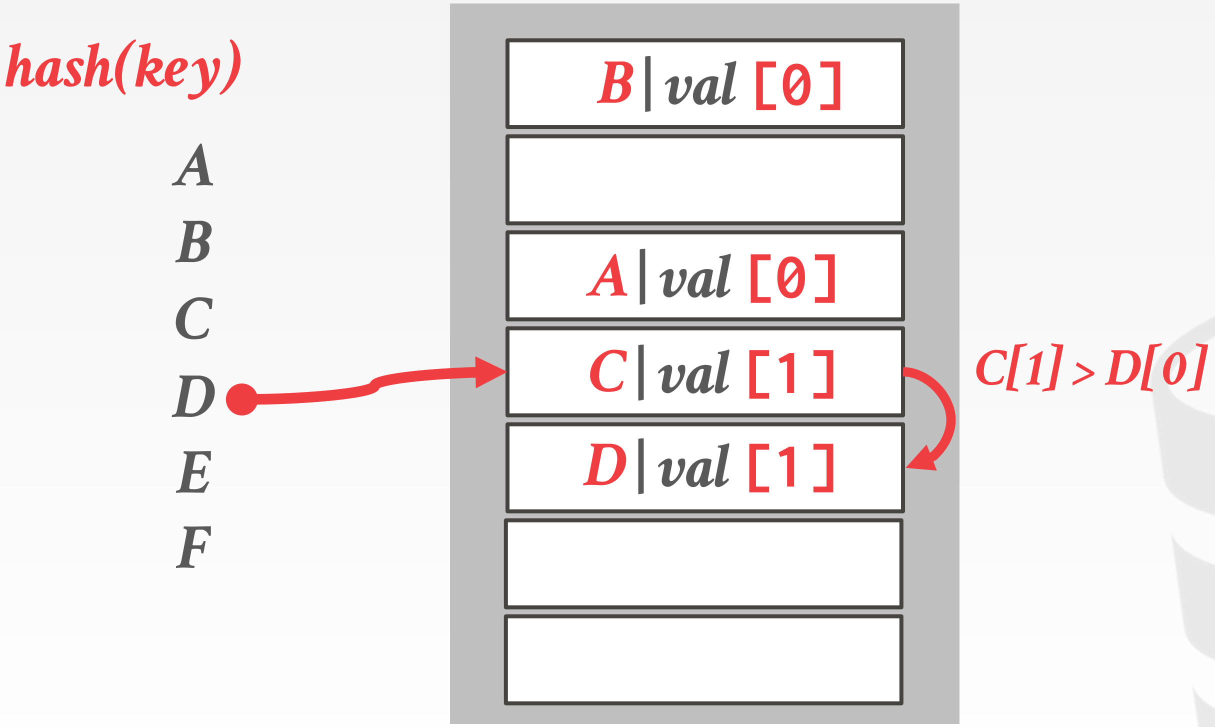
ROBIN HOOD HASHING



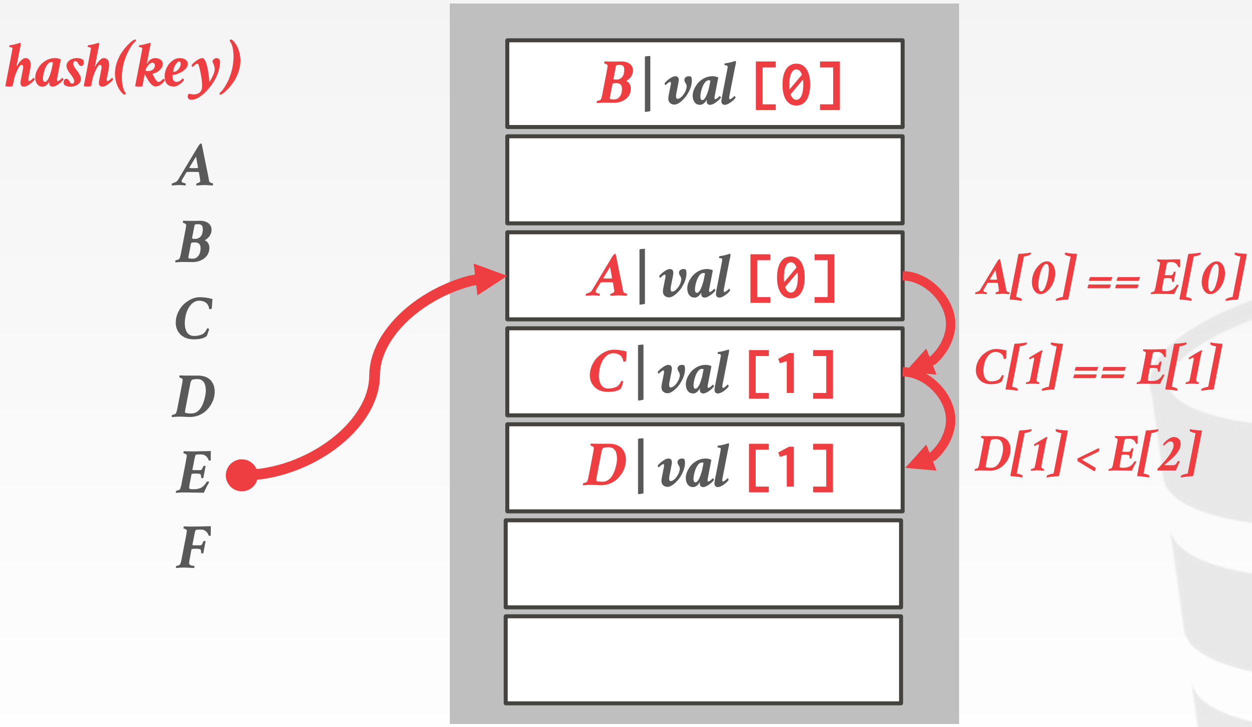
ROBIN HOOD HASHING



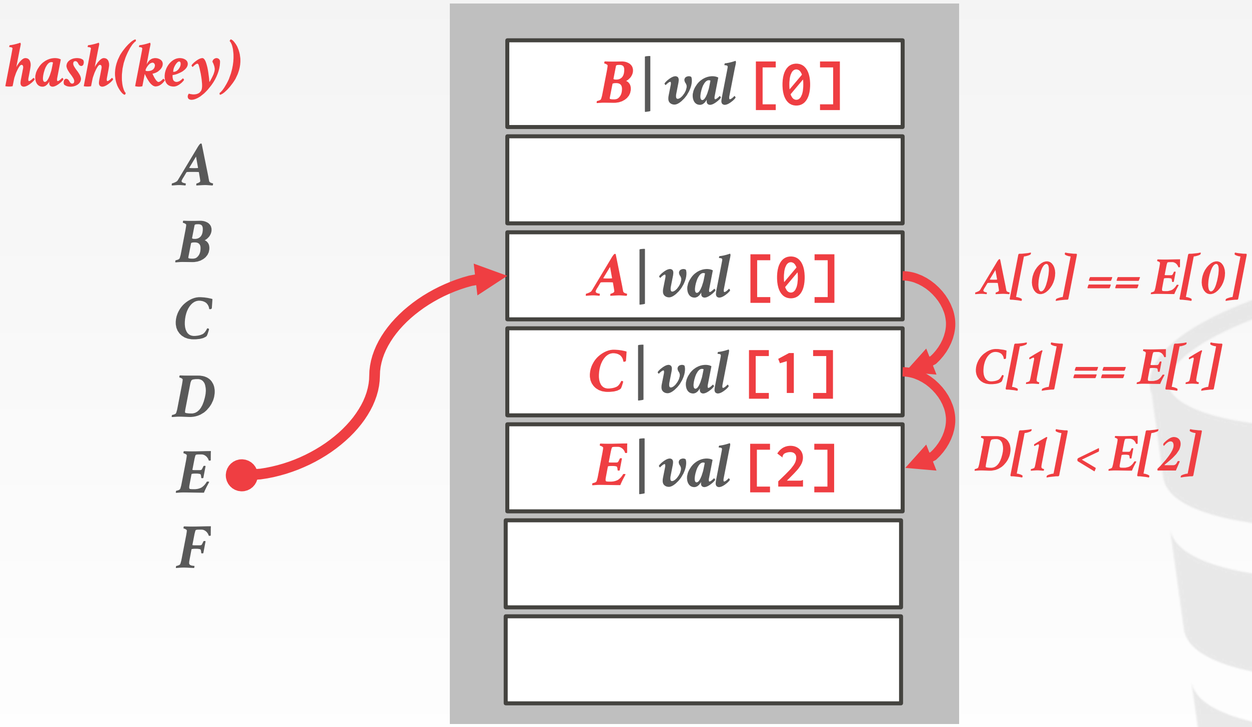
ROBIN HOOD HASHING



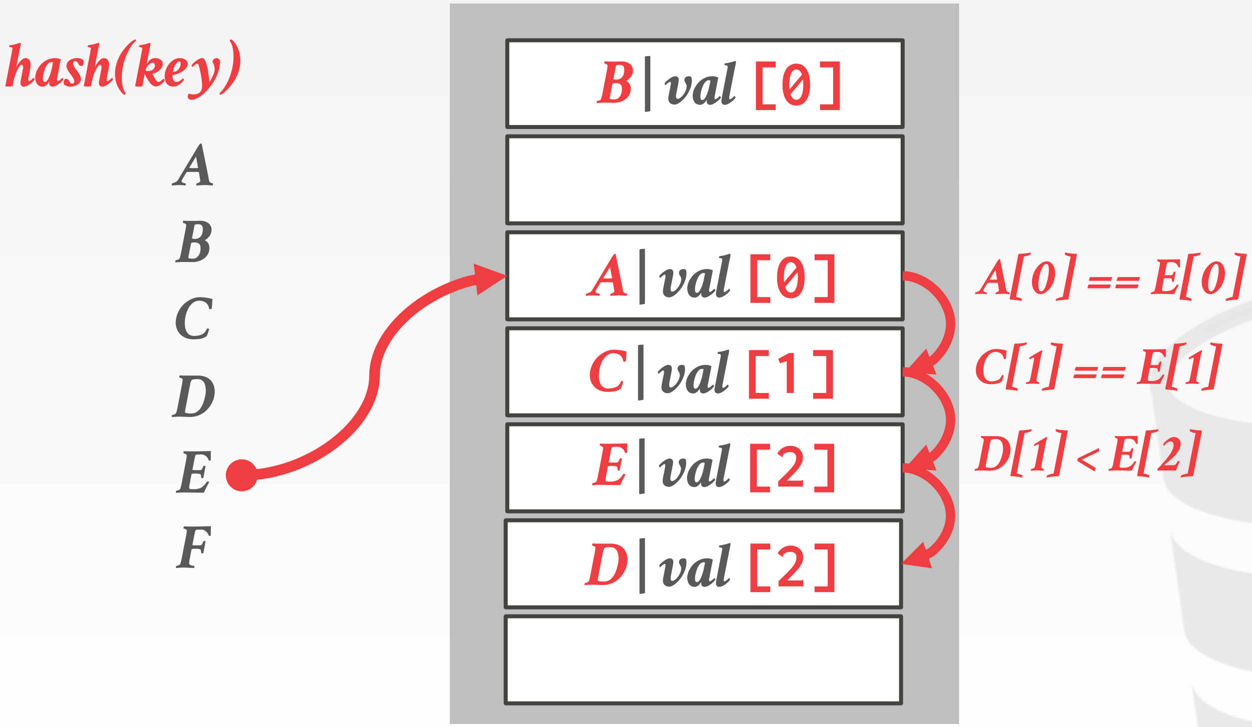
ROBIN HOOD HASHING



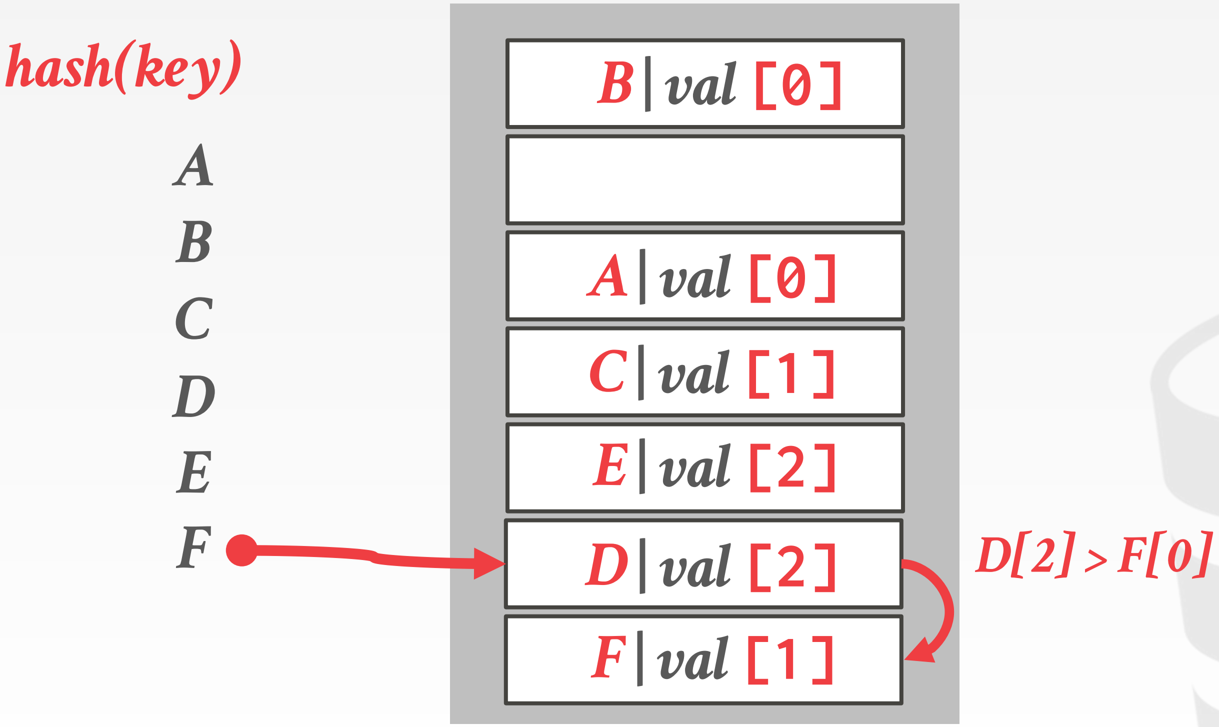
ROBIN HOOD HASHING



ROBIN HOOD HASHING



ROBIN HOOD HASHING



CUCKOO HASHING

Use multiple hash tables with different hash function seeds.

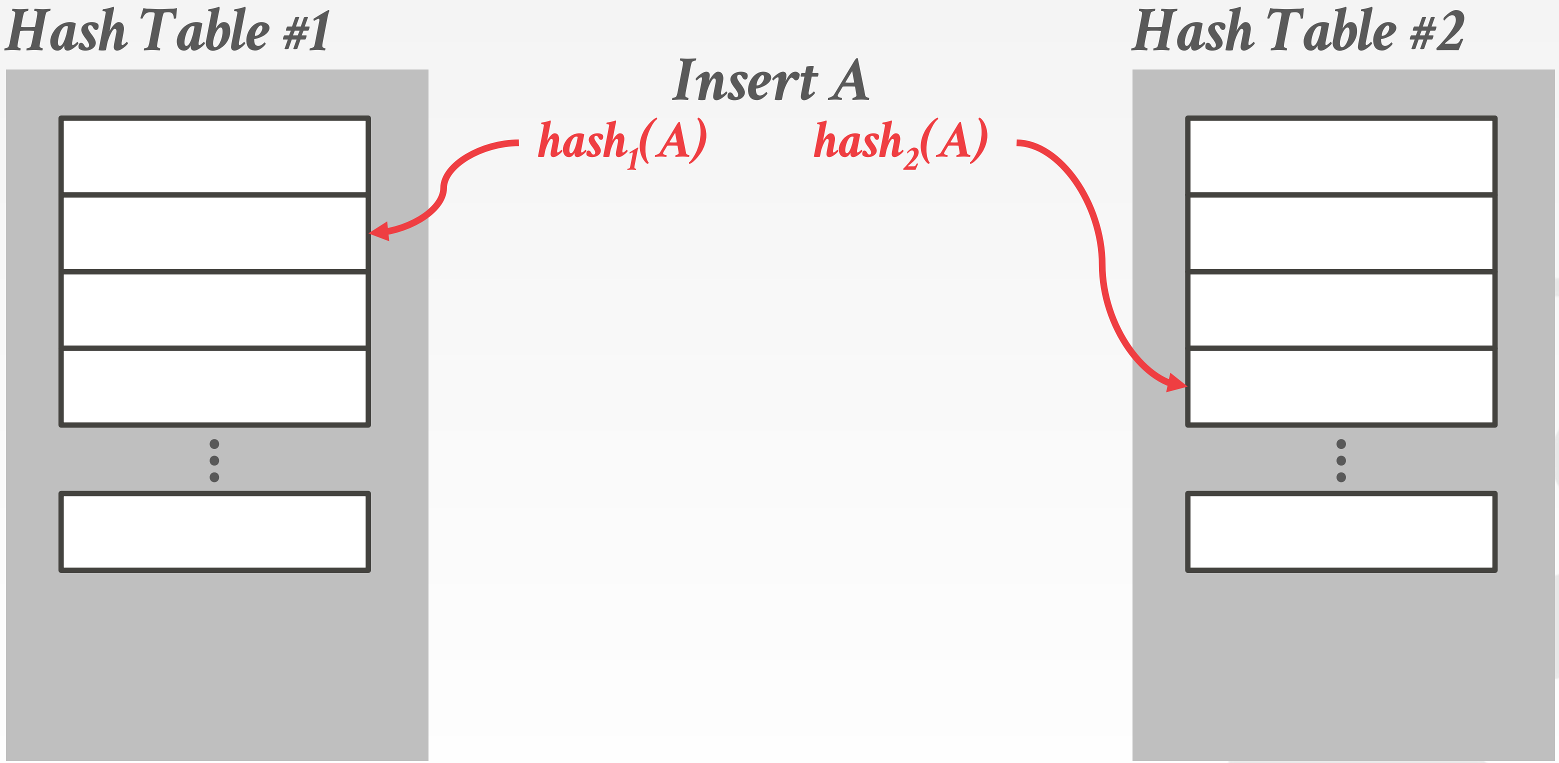
→ On insert, check every table and pick anyone that has a free slot.

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

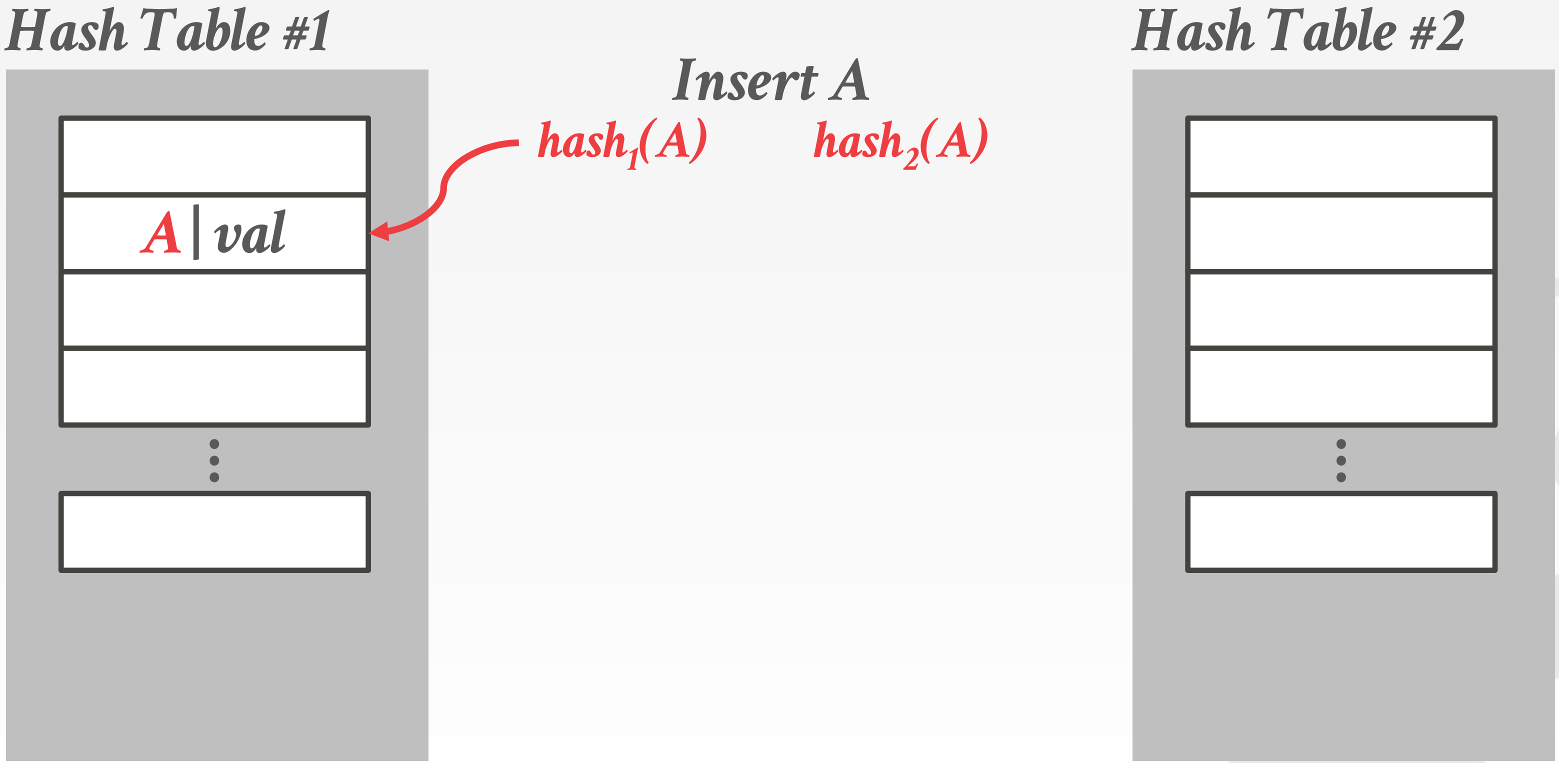
Look-ups and deletions are always **O(1)** because only one location per hash table is checked.

Best open-source implementation is from CMU.

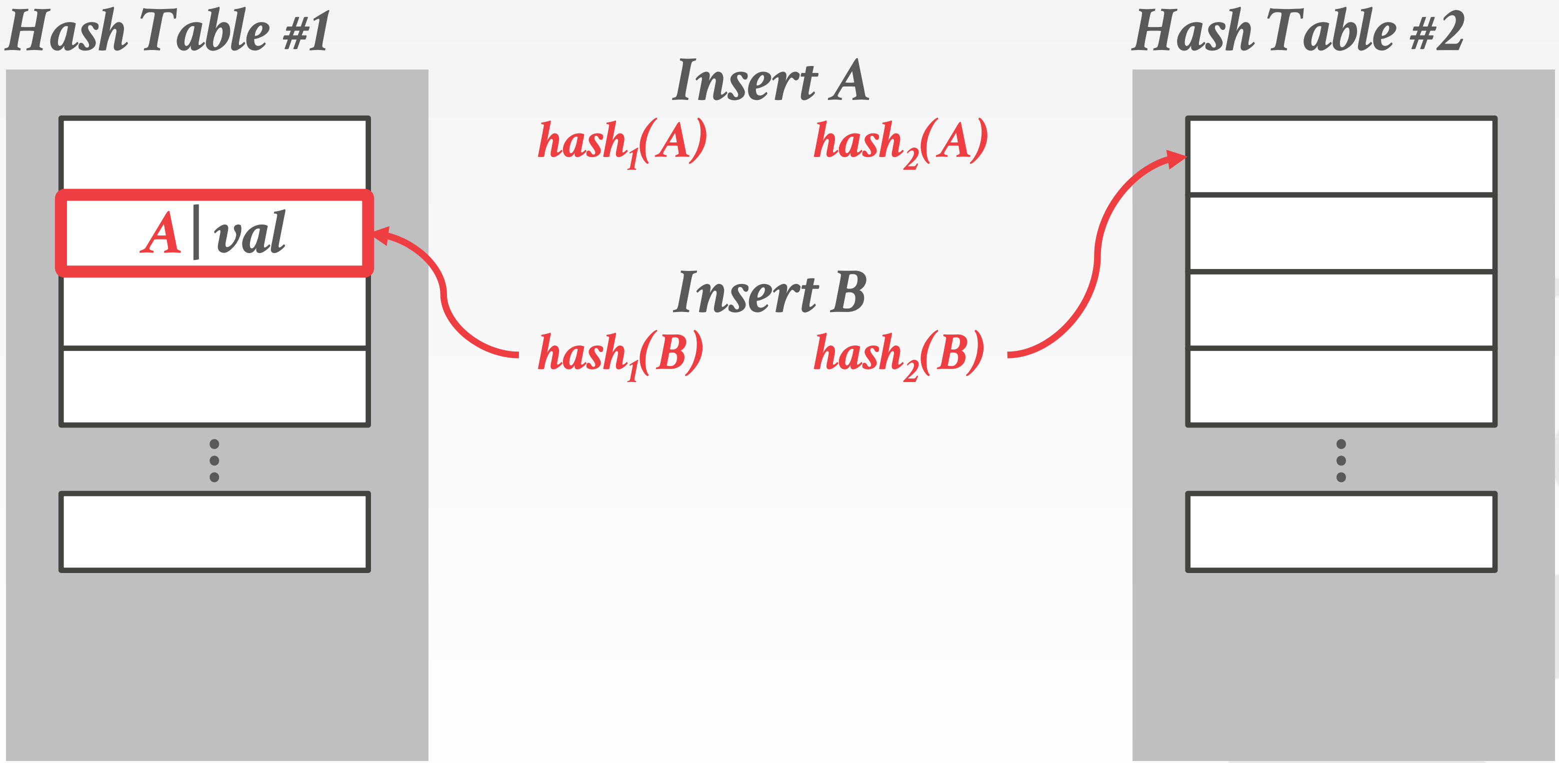
CUCKOO HASHING



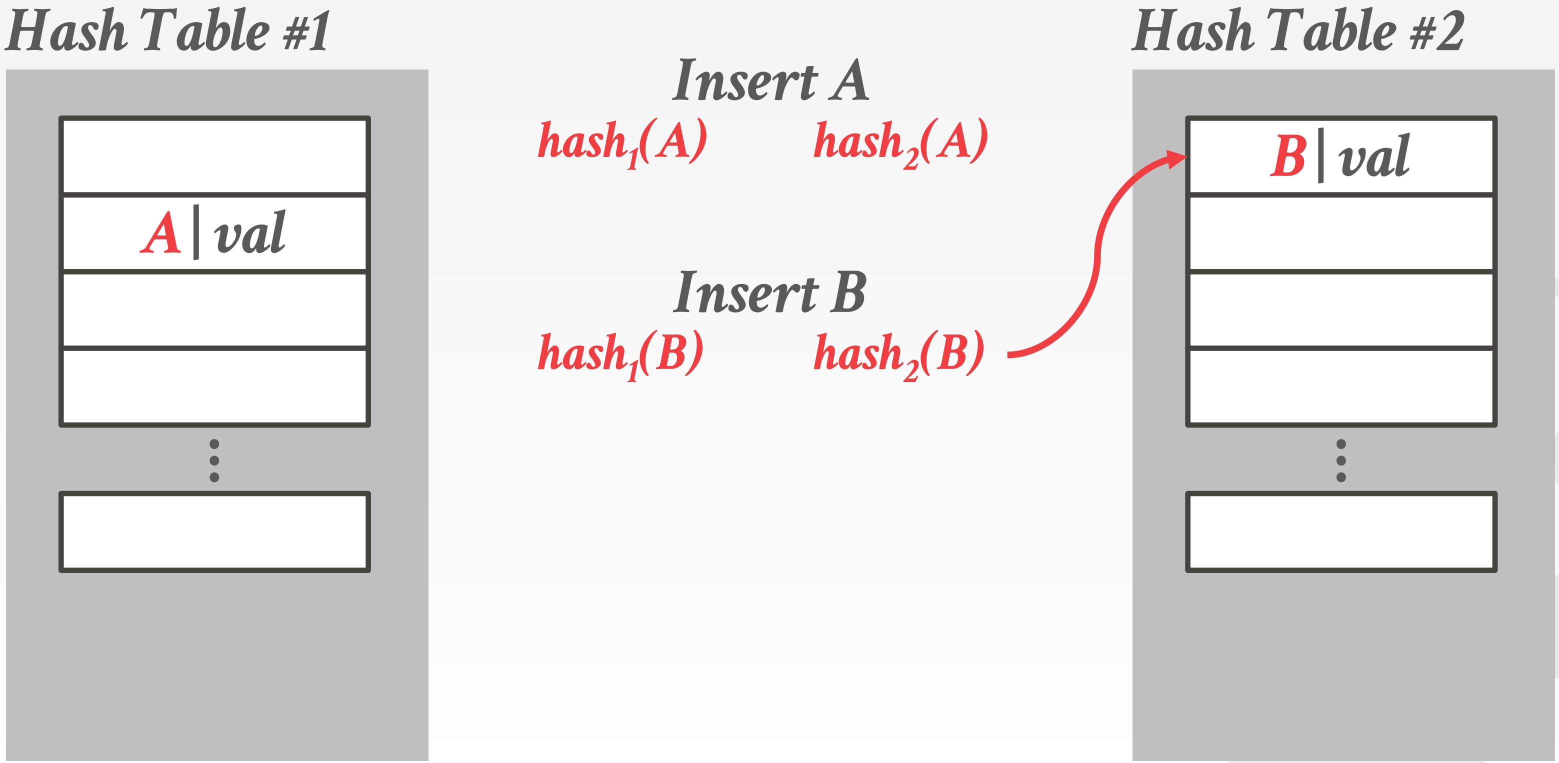
CUCKOO HASHING



CUCKOO HASHING



CUCKOO HASHING



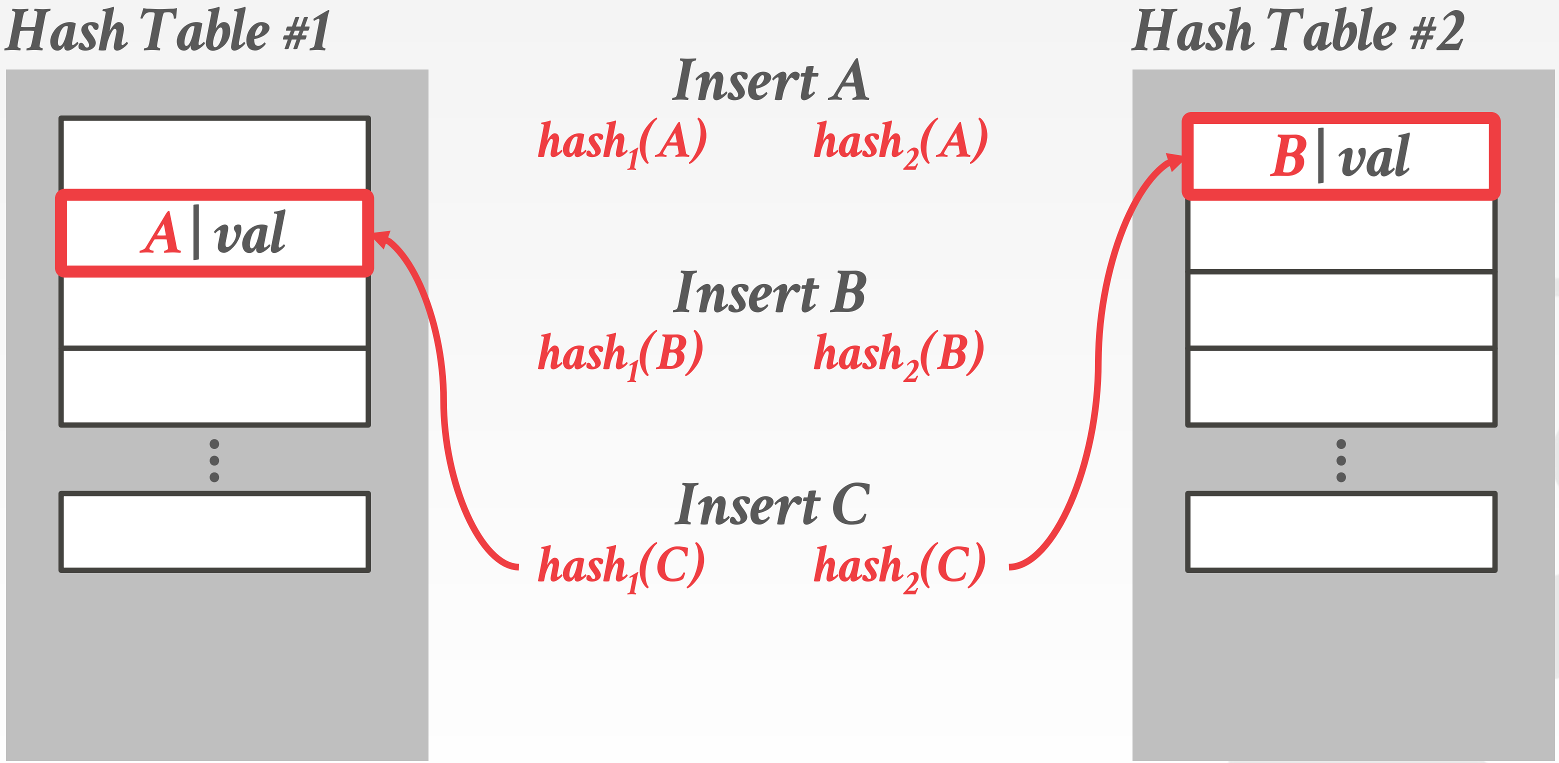
Insert A

$hash_1(A)$ $hash_2(A)$

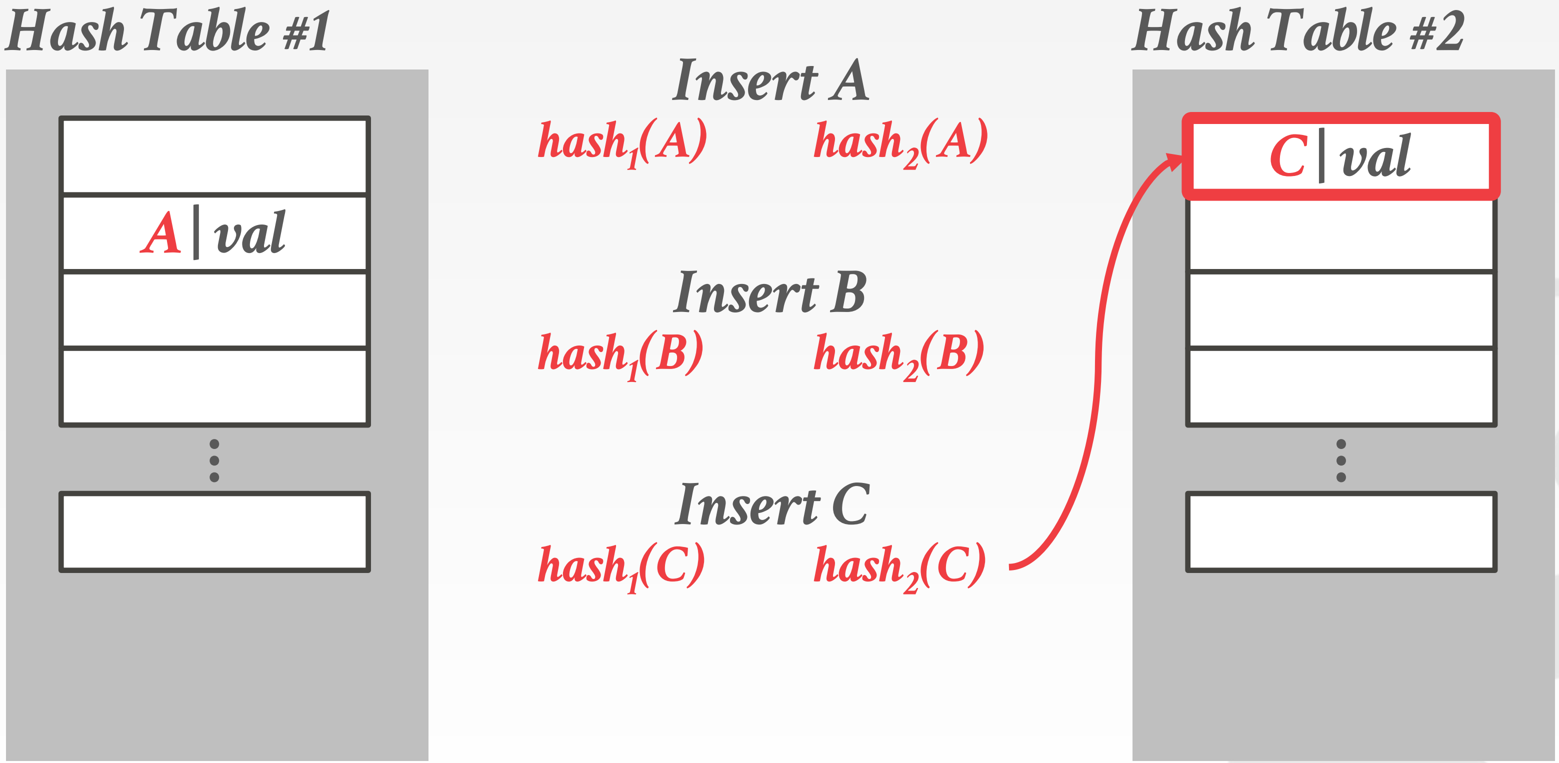
Insert B

$hash_1(B)$ $hash_2(B)$

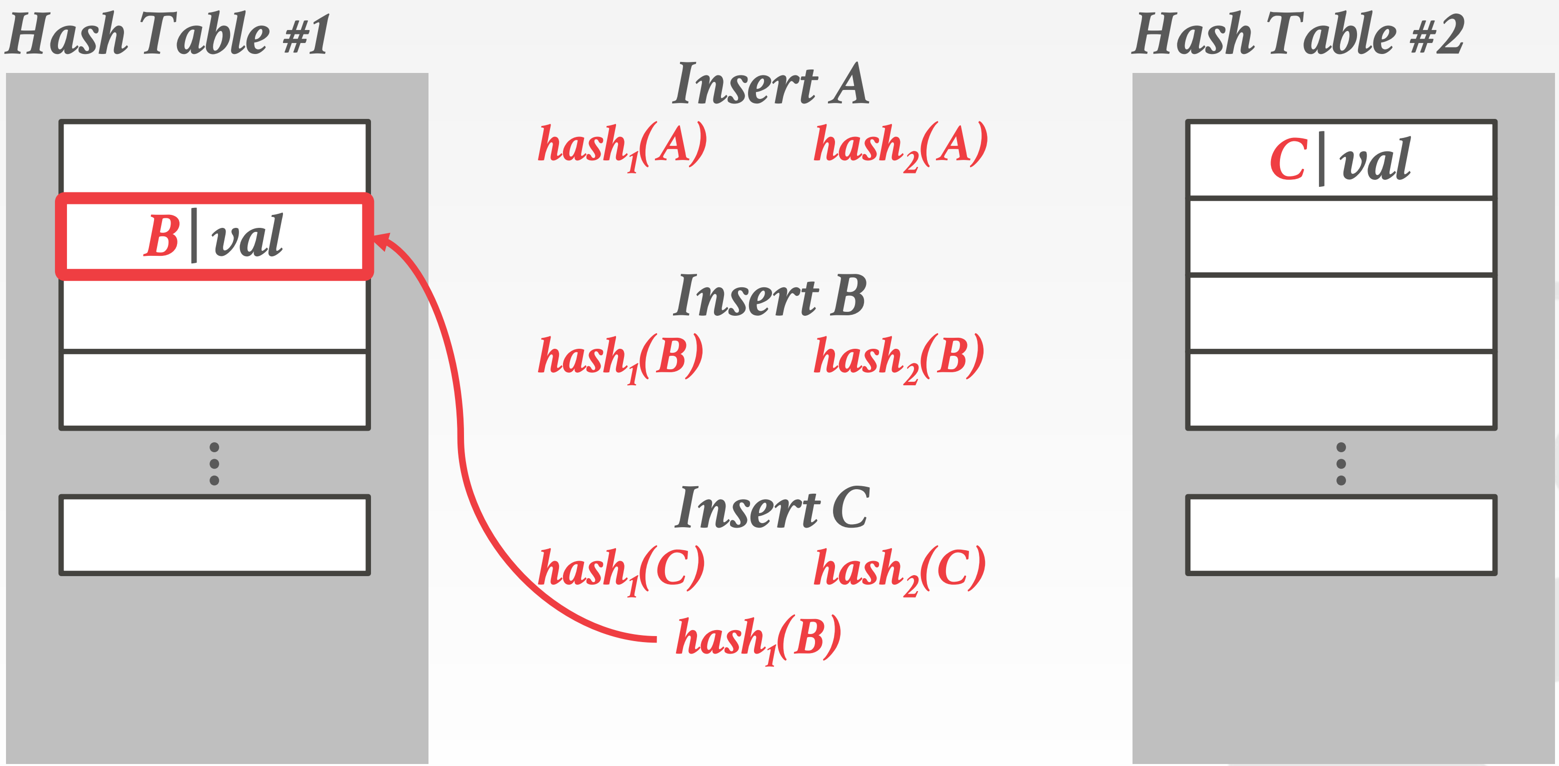
CUCKOO HASHING



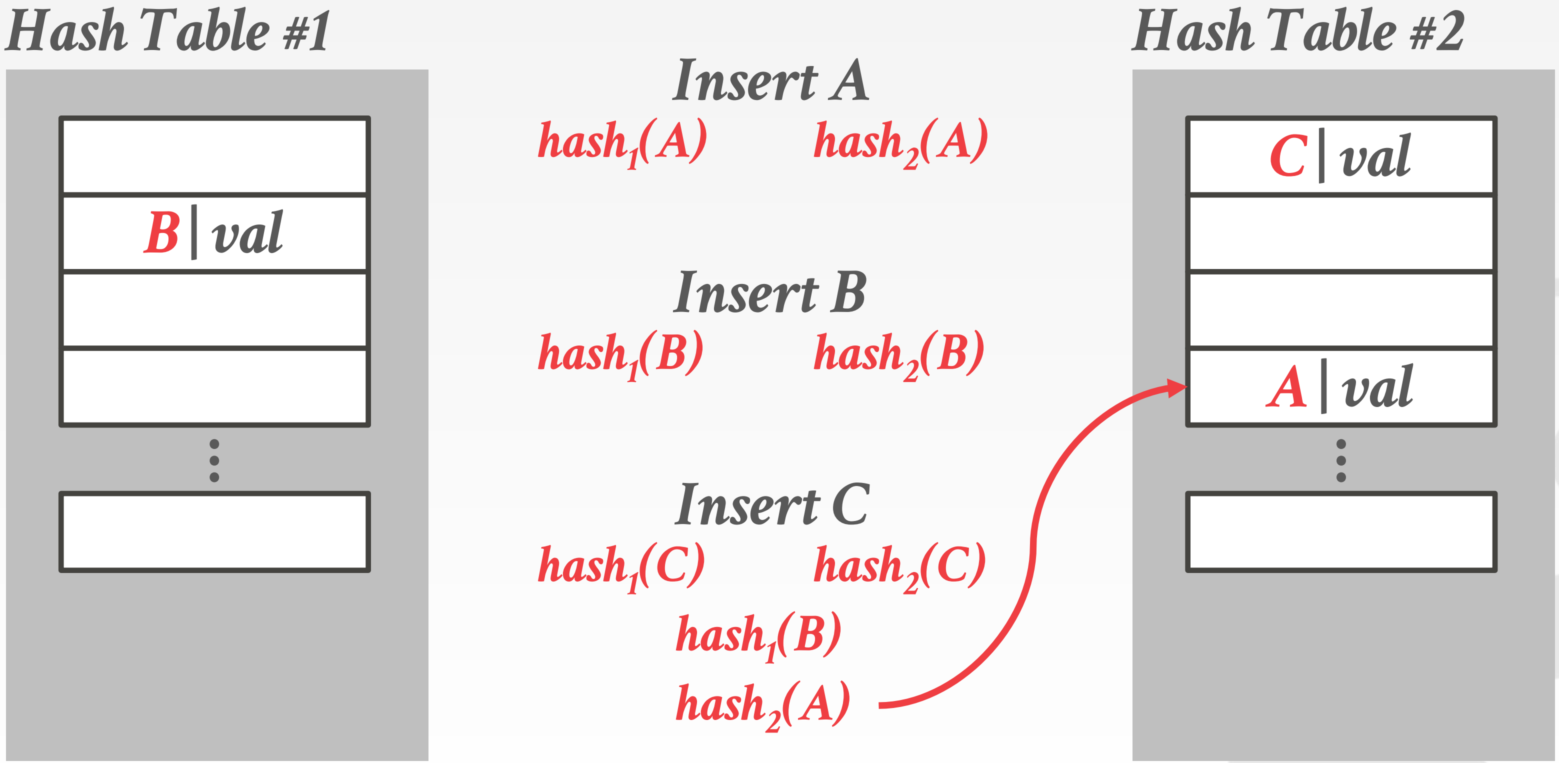
CUCKOO HASHING



CUCKOO HASHING



CUCKOO HASHING



Insert A

$hash_1(A)$ $hash_2(A)$

Insert B

$hash_1(B)$ $hash_2(B)$

Insert C

$hash_1(C)$ $hash_2(C)$

$hash_1(B)$

$hash_2(A)$

OBSERVATION

The previous hash tables require the DBMS to know the number of elements it wants to store.

→ Otherwise it must rebuild the table if it needs to grow/shrink in size.

Dynamic hash tables resize themselves on demand.

→ Chained Hashing
→ Extendible Hashing
→ Linear Hashing

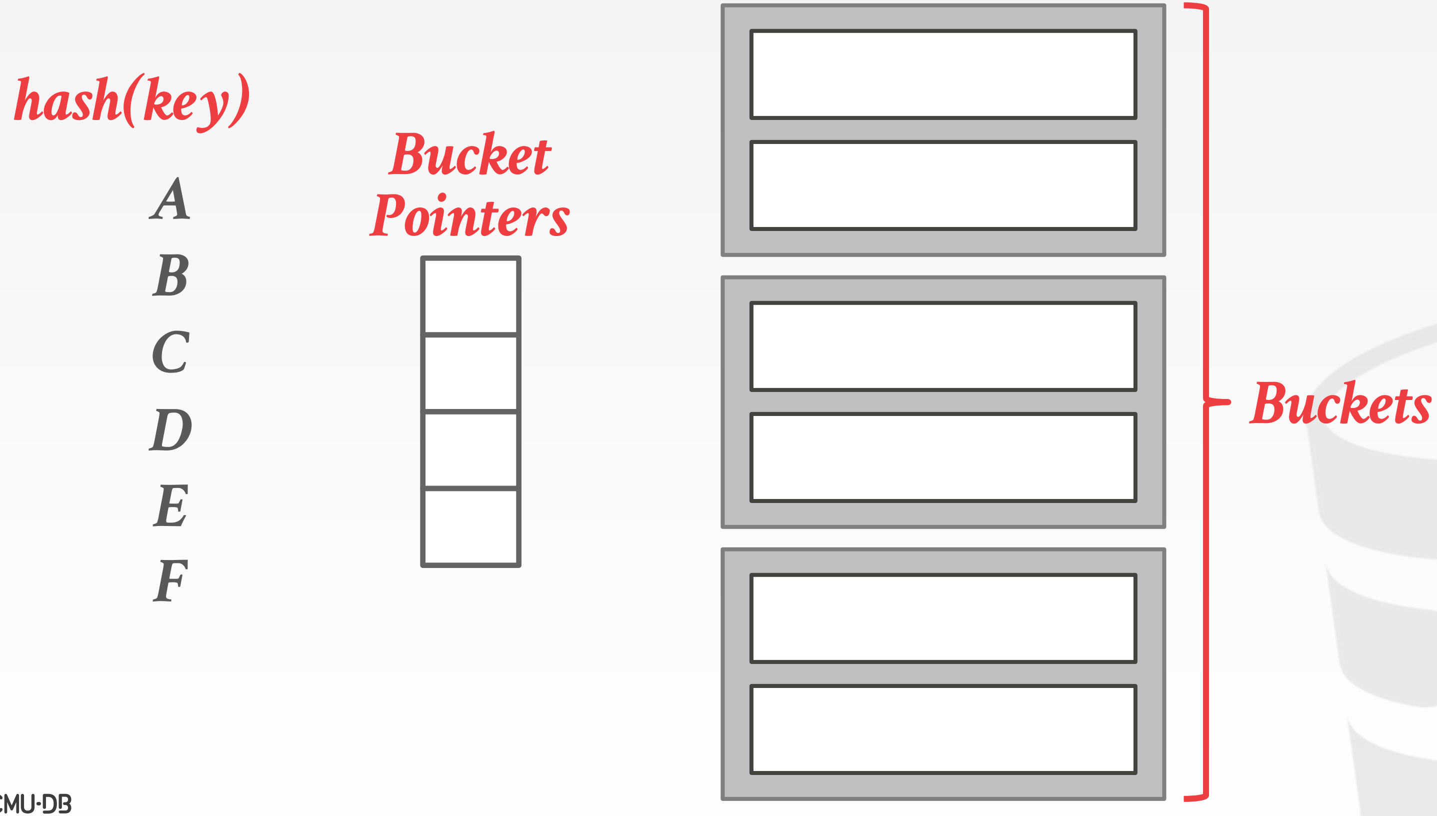
CHAINED HASHING

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

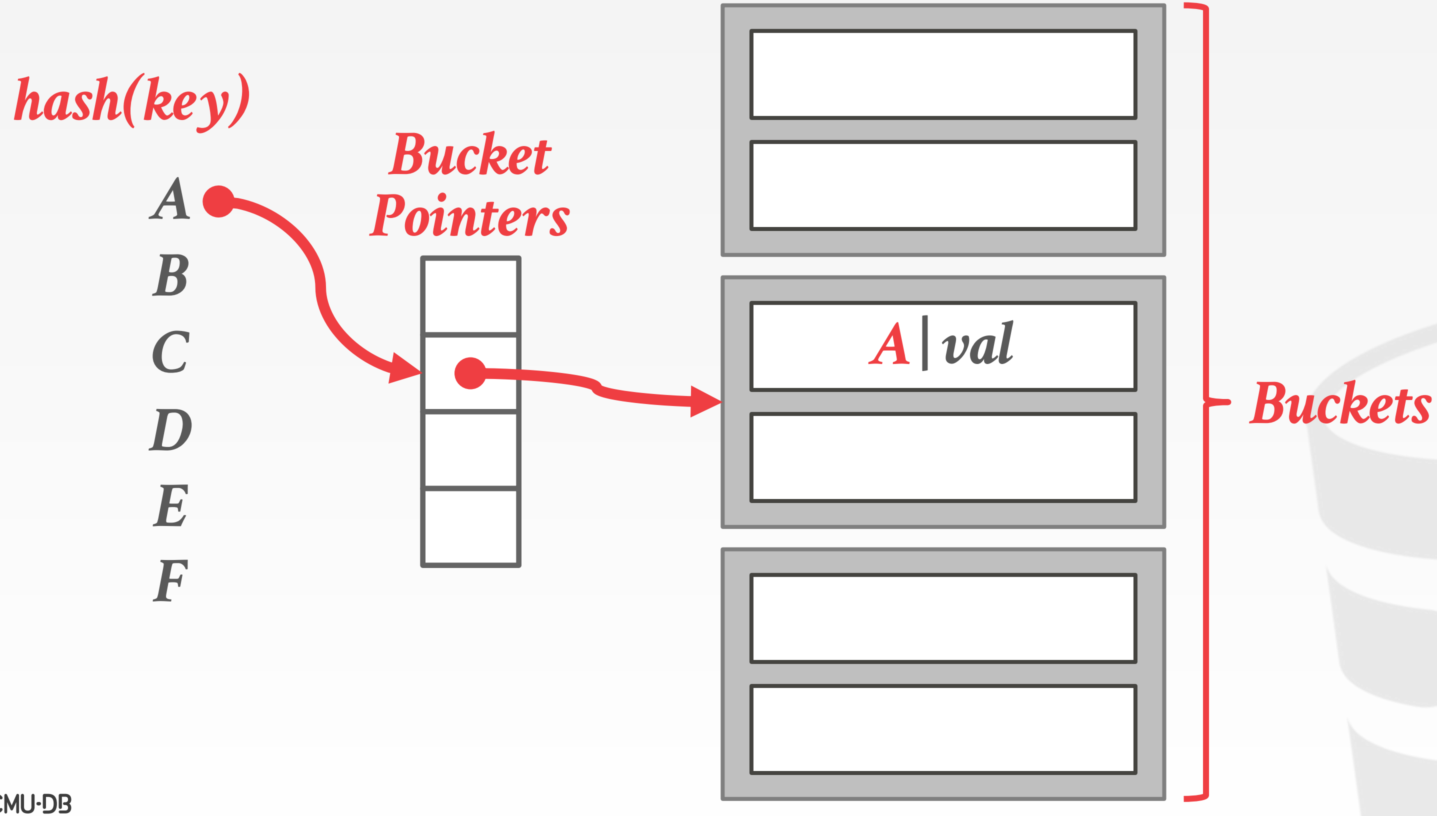
Resolve collisions by placing all elements with the same hash key into the same bucket.

→ To determine whether an element is present, hash to its bucket and scan for it.
→ Insertions and deletions are generalizations of lookups.

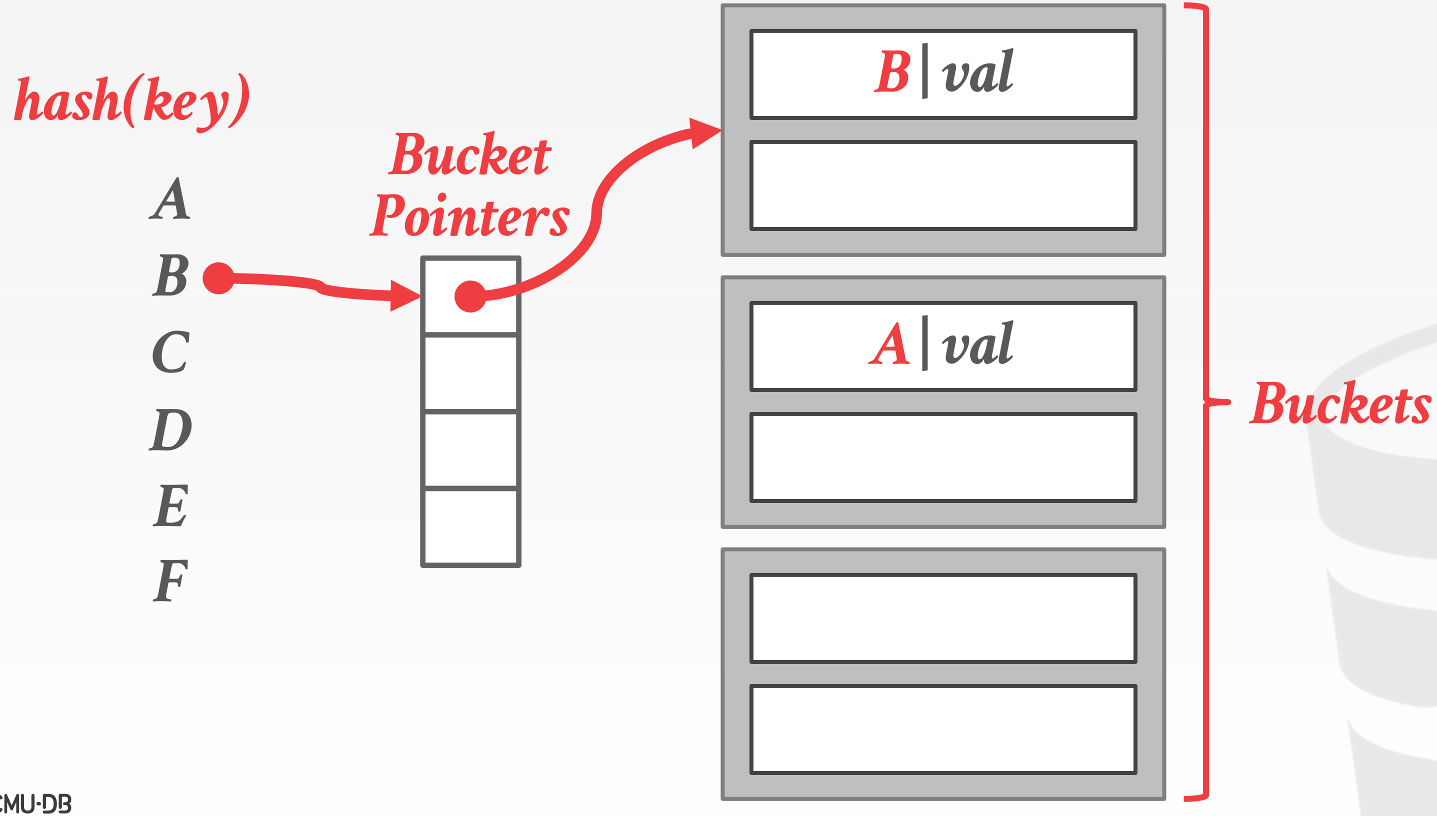
CHAINED HASHING



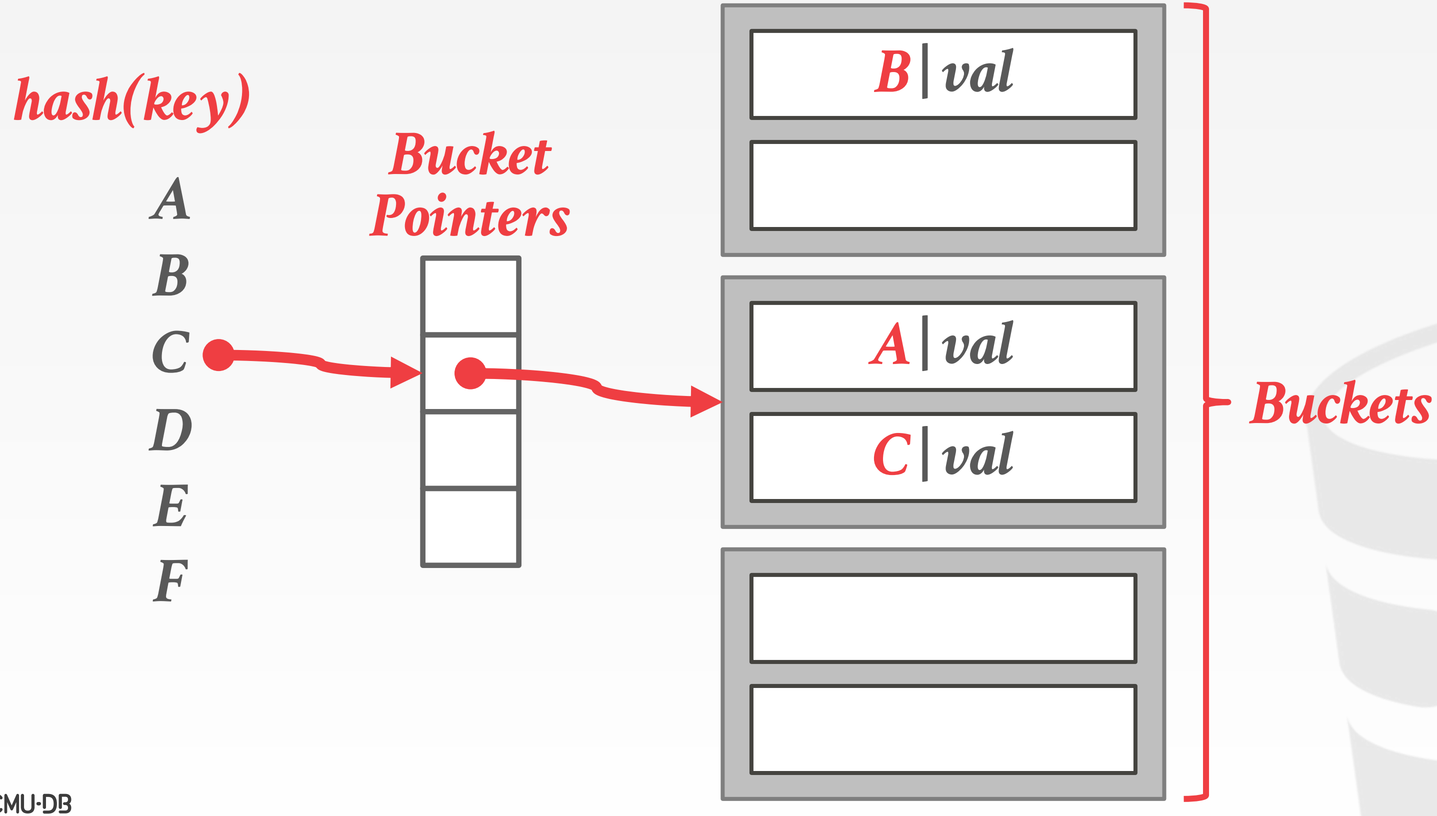
CHAINED HASHING



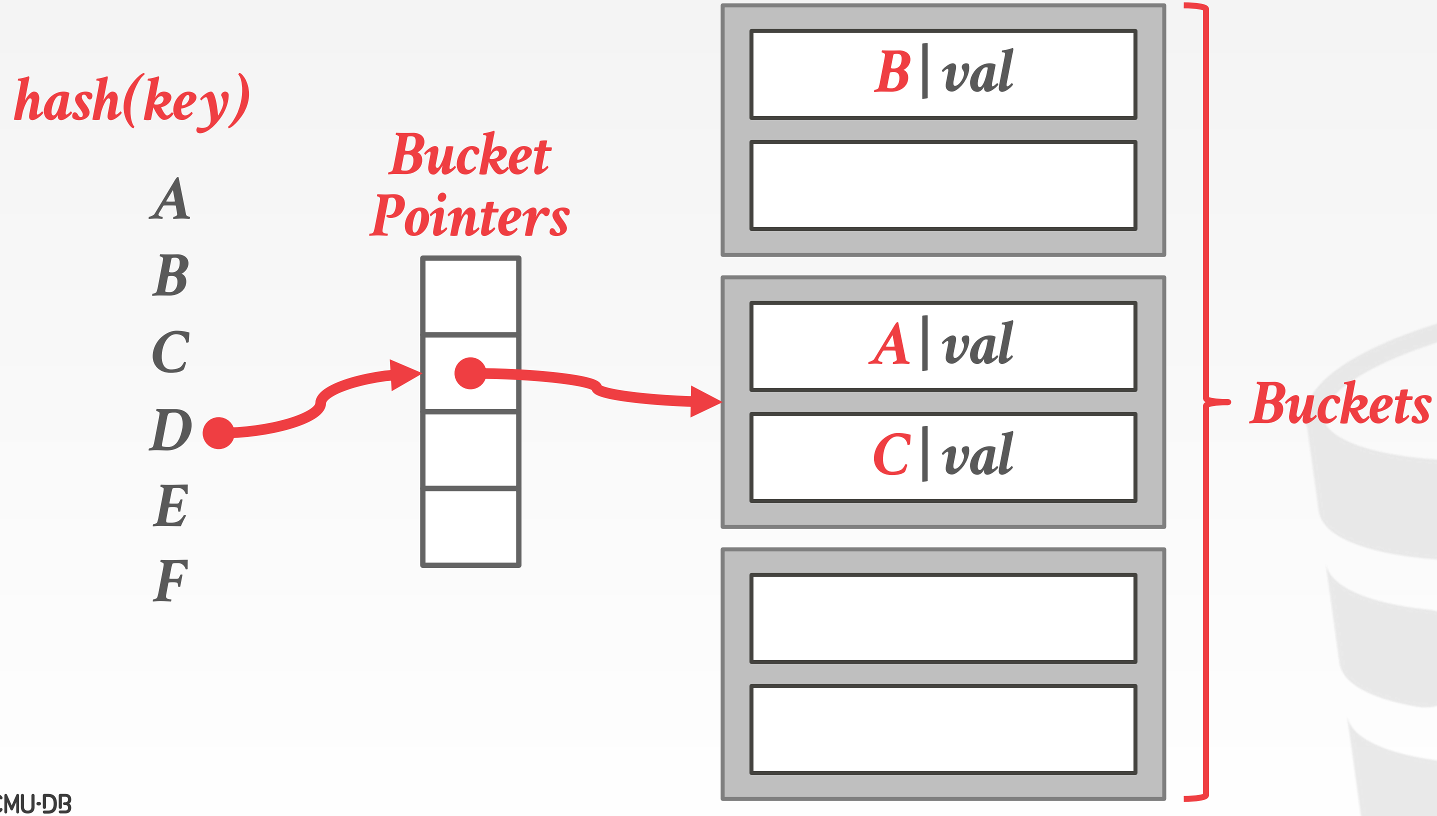
CHAINED HASHING



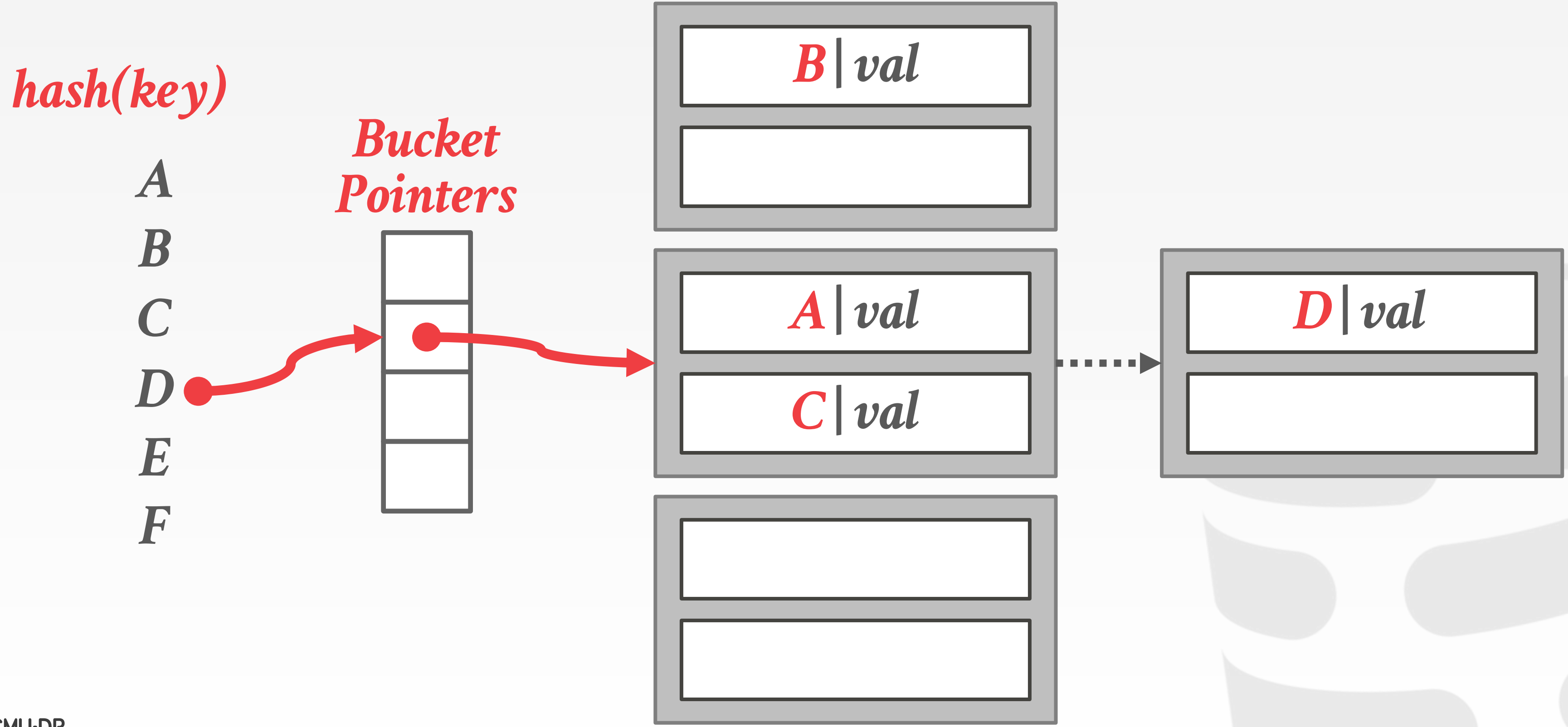
CHAINED HASHING



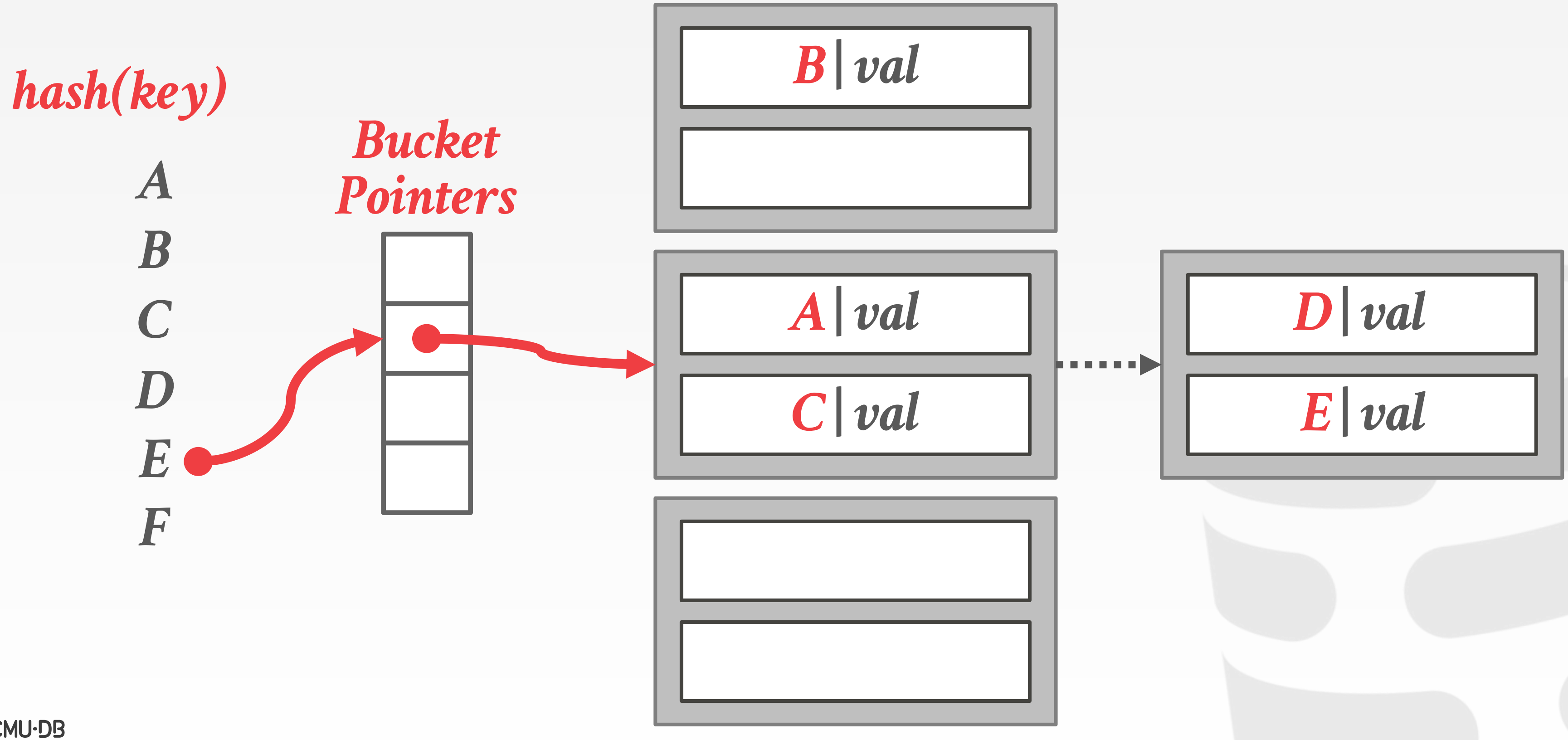
CHAINED HASHING



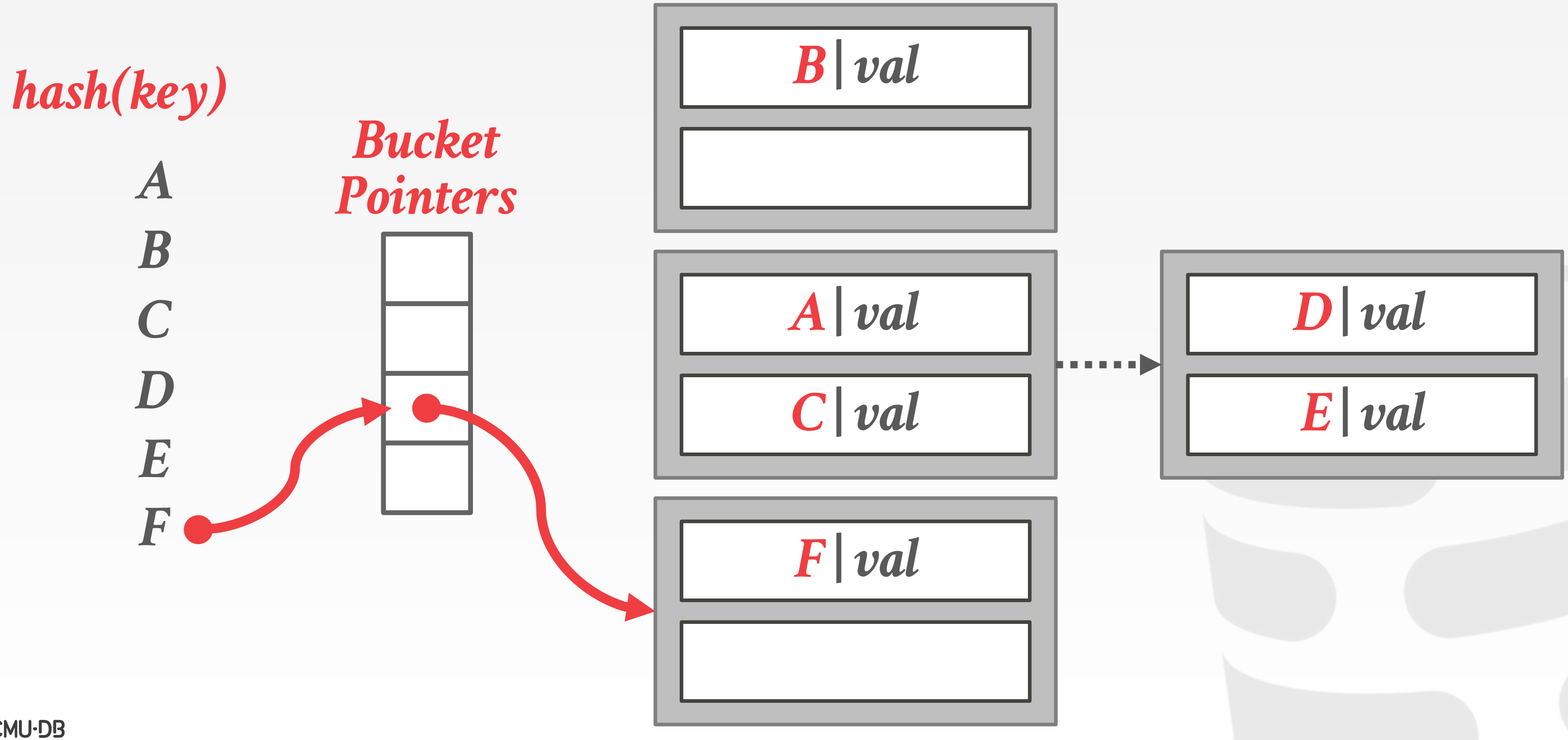
CHAINED HASHING



CHAINED HASHING



CHAINED HASHING



EXTENDIBLE HASHING

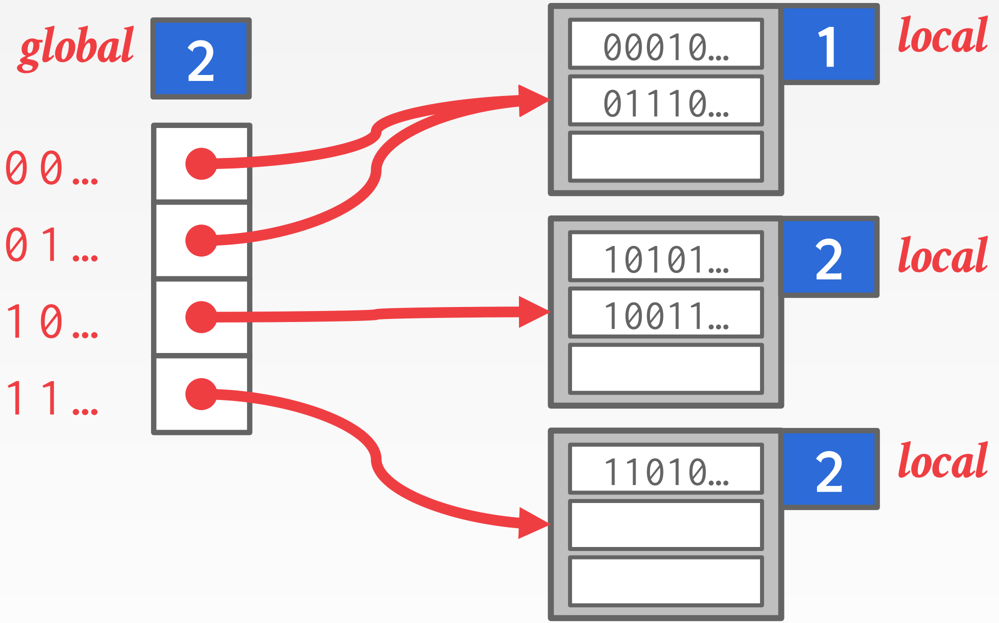
Chained-hashing approach where we split buckets instead of letting the linked list grow forever.

Multiple slot locations can point to the same bucket chain.

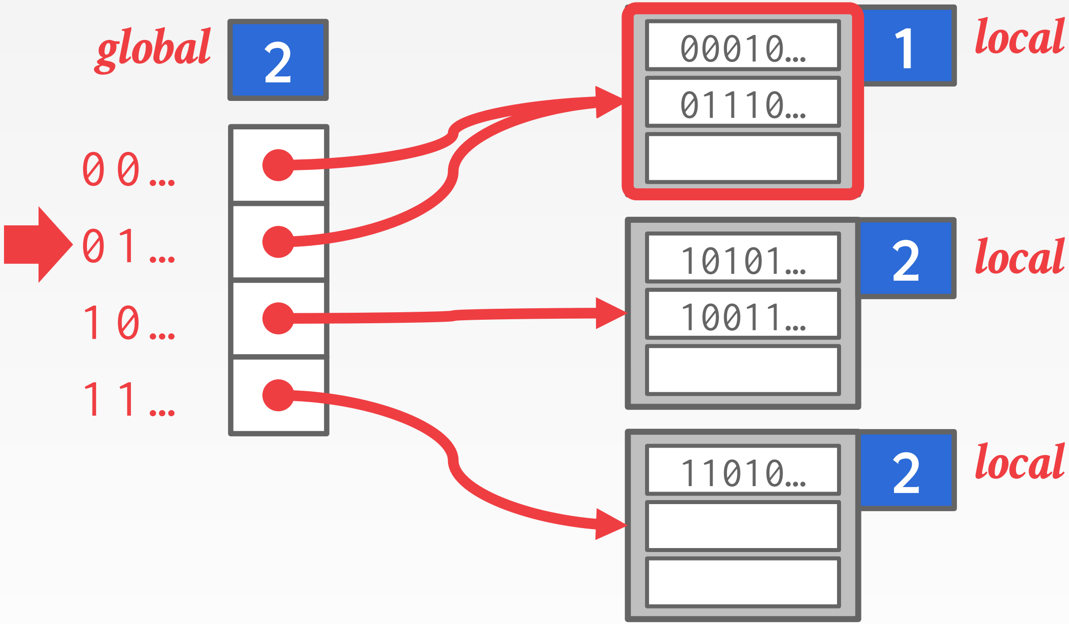
Reshuffling bucket entries on split and increase the number of bits to examine.

→ Data movement is localized to just the split chain.

EXTENDIBLE HASHING



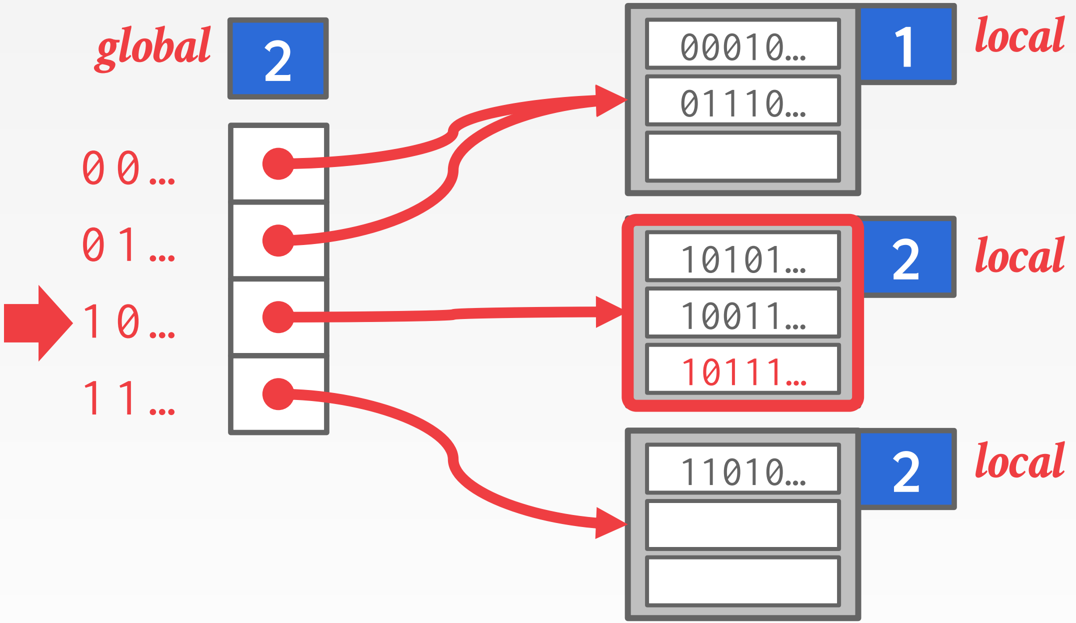
EXTENDIBLE HASHING



Find A

hash(A) = $\boxed{01}$110…

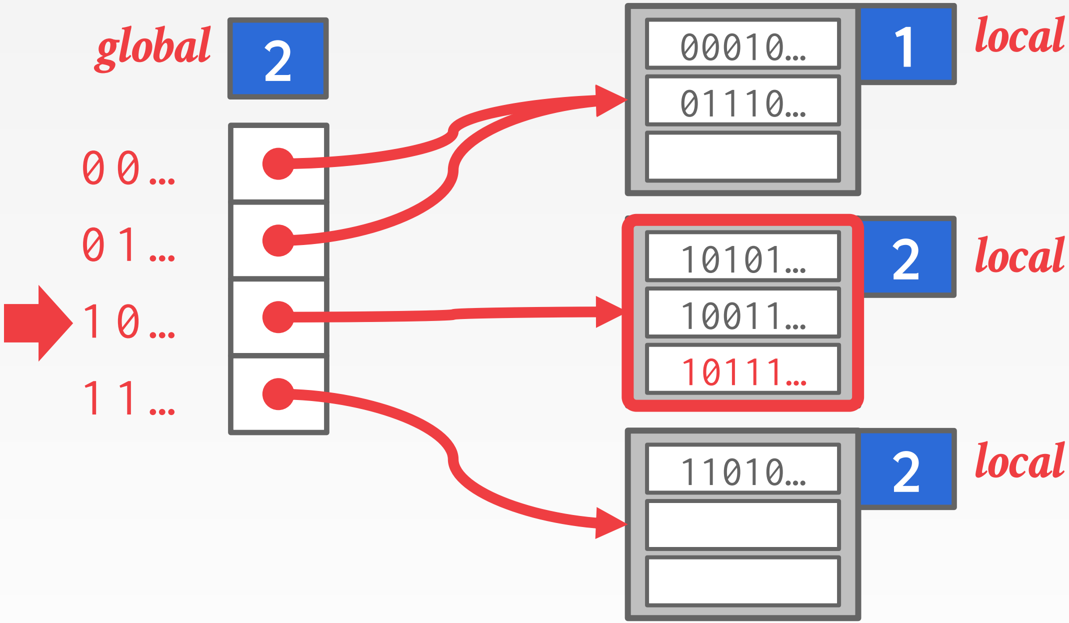
EXTENDIBLE HASHING



Find A
hash(A) = 01110…

Insert B
hash(B) = 10111…

EXTENDIBLE HASHING

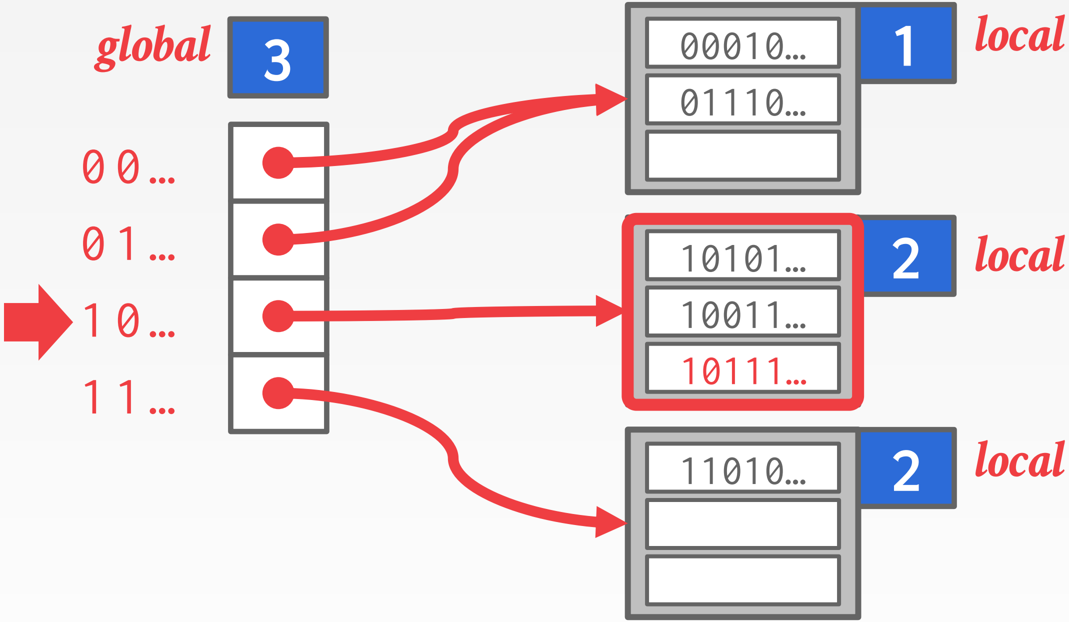


Find A
hash(A) = 01110…

Insert B
hash(B) = 10111…

Insert C
hash(C) = $\boxed{10}$100…

EXTENDIBLE HASHING

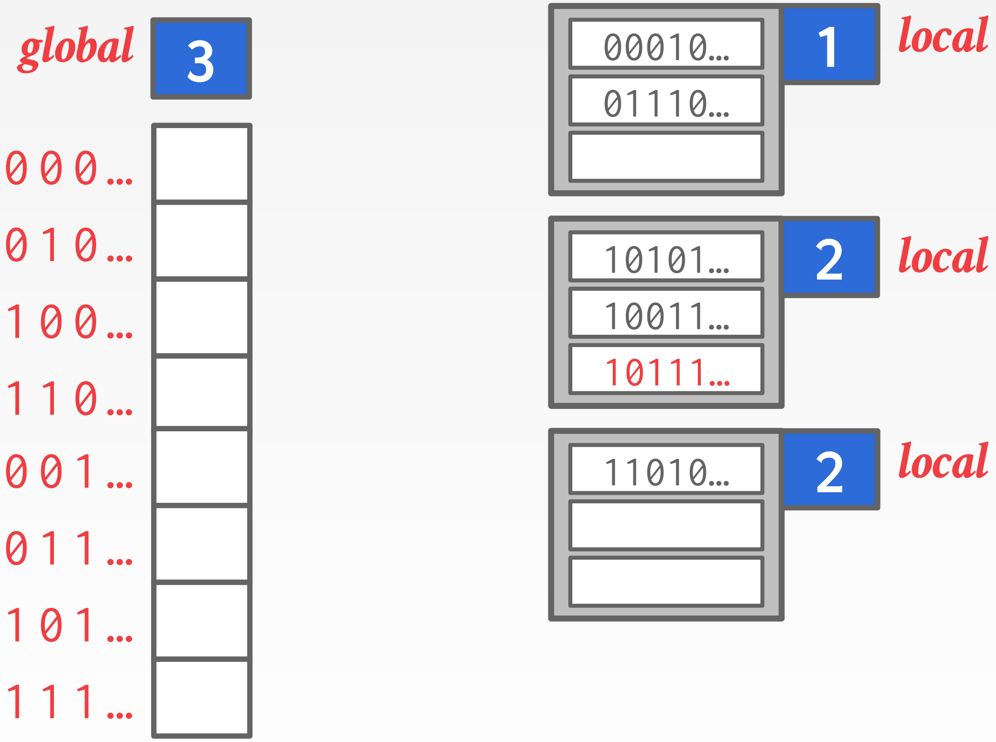


Find A
hash(A) = 01110…

Insert B
hash(B) = 10111…

Insert C
hash(C) = $\boxed{10}$100…

EXTENDIBLE HASHING

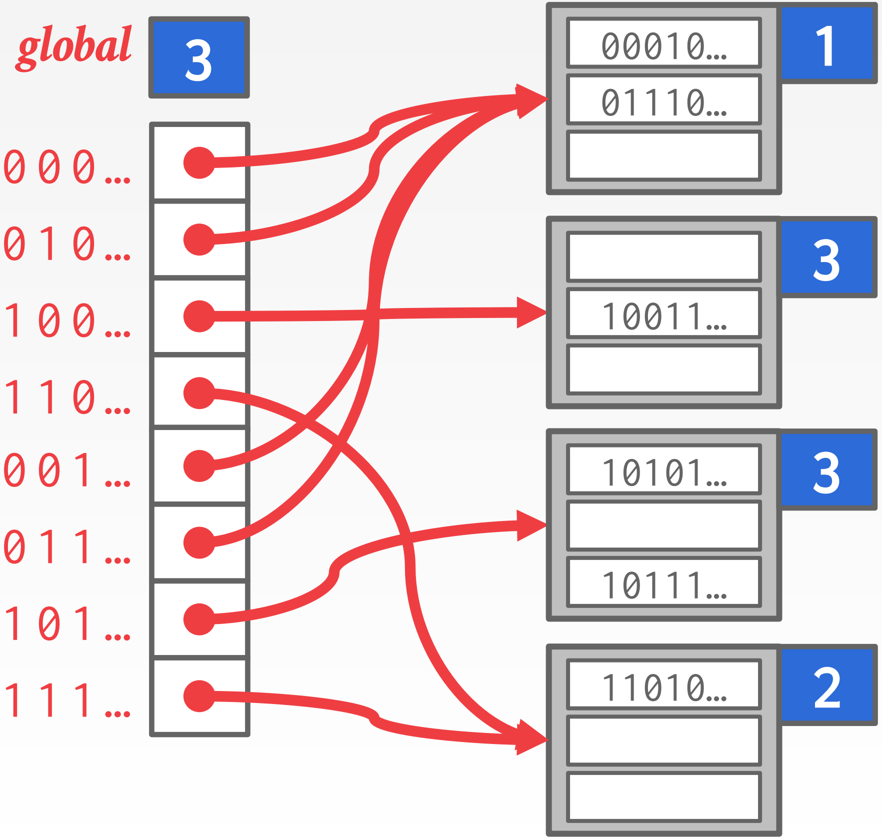


Find A
hash(A) = 01110…

Insert B
hash(B) = 10111…

Insert C
hash(C) = $\boxed{10}$100…

EXTENDIBLE HASHING

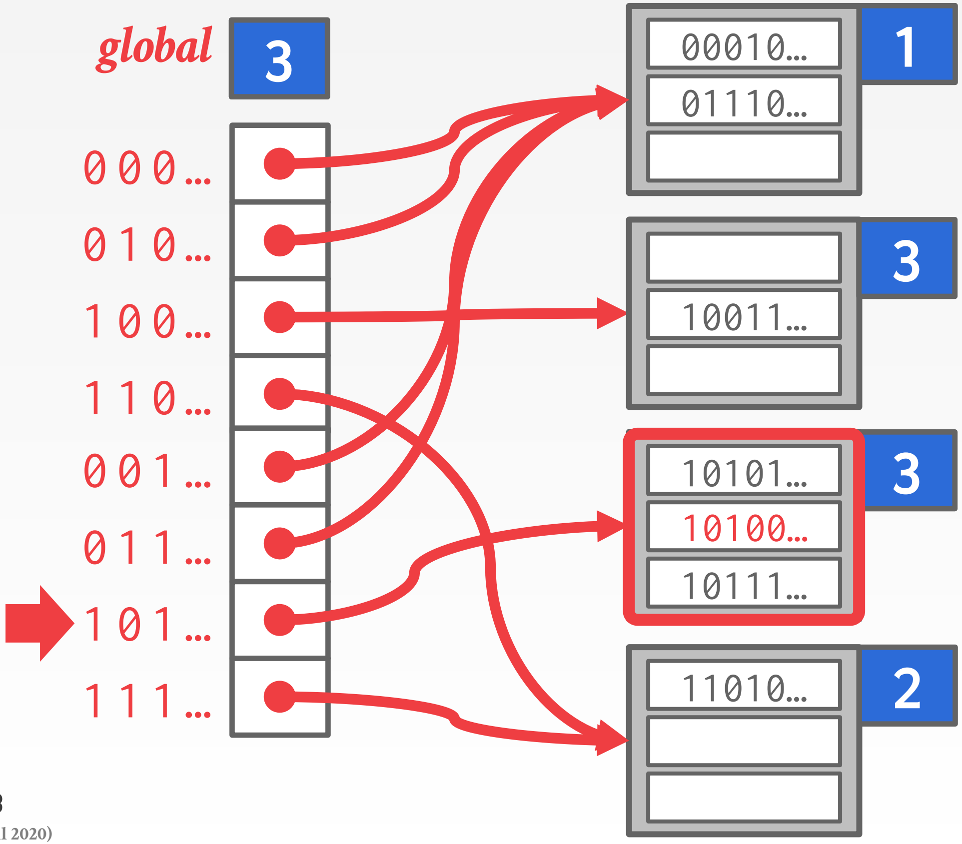


Find A
hash(A) = 01110…

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

The hash table maintains a pointer that tracks the next bucket to split.

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

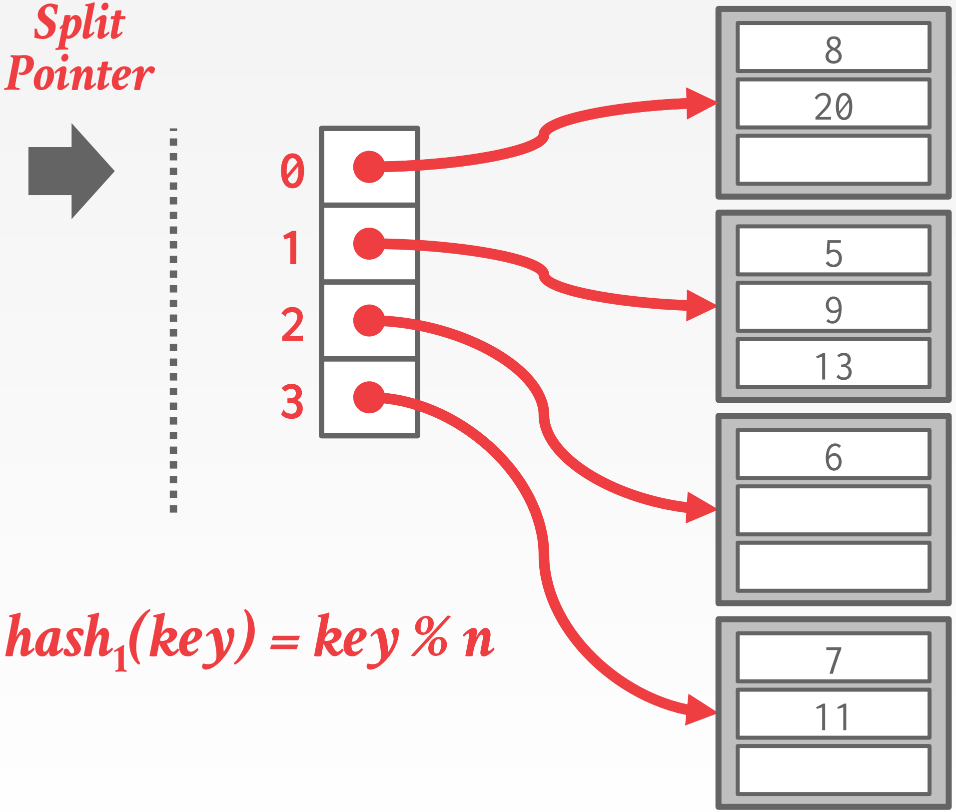
Use multiple hashes to find the right bucket for a given key.

Can use different overflow criterion:

→ Space Utilization

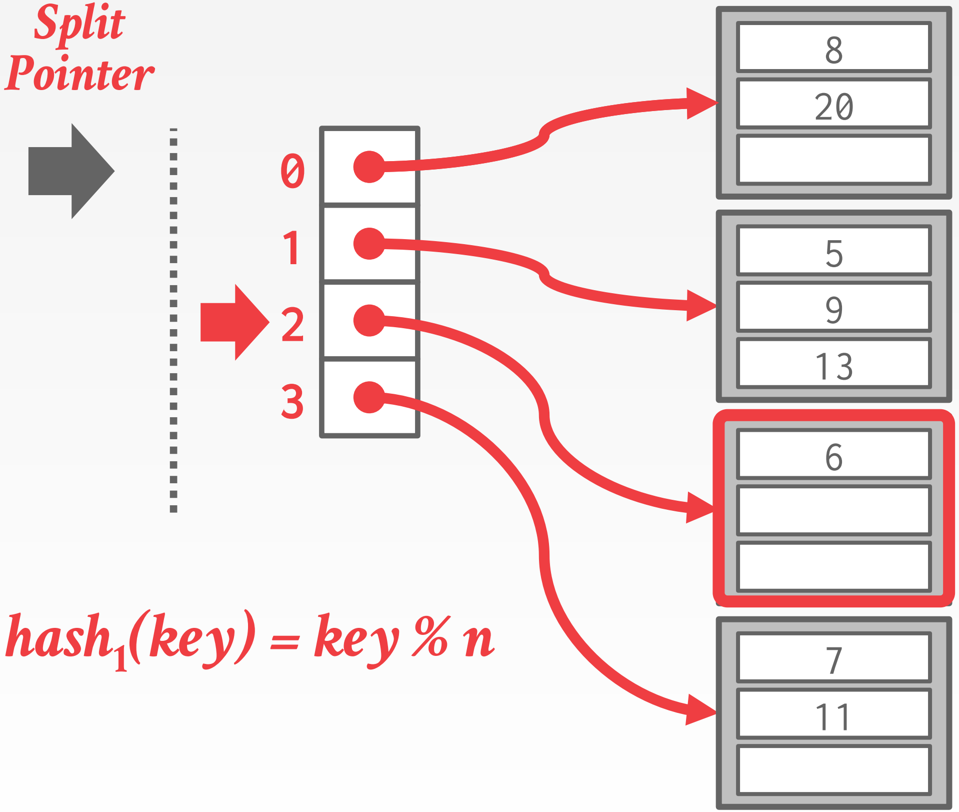
→ Average Length of Overflow Chains

LINEAR HASHING



$$hash_1(key) = key \,\%\, n$$

LINEAR HASHING

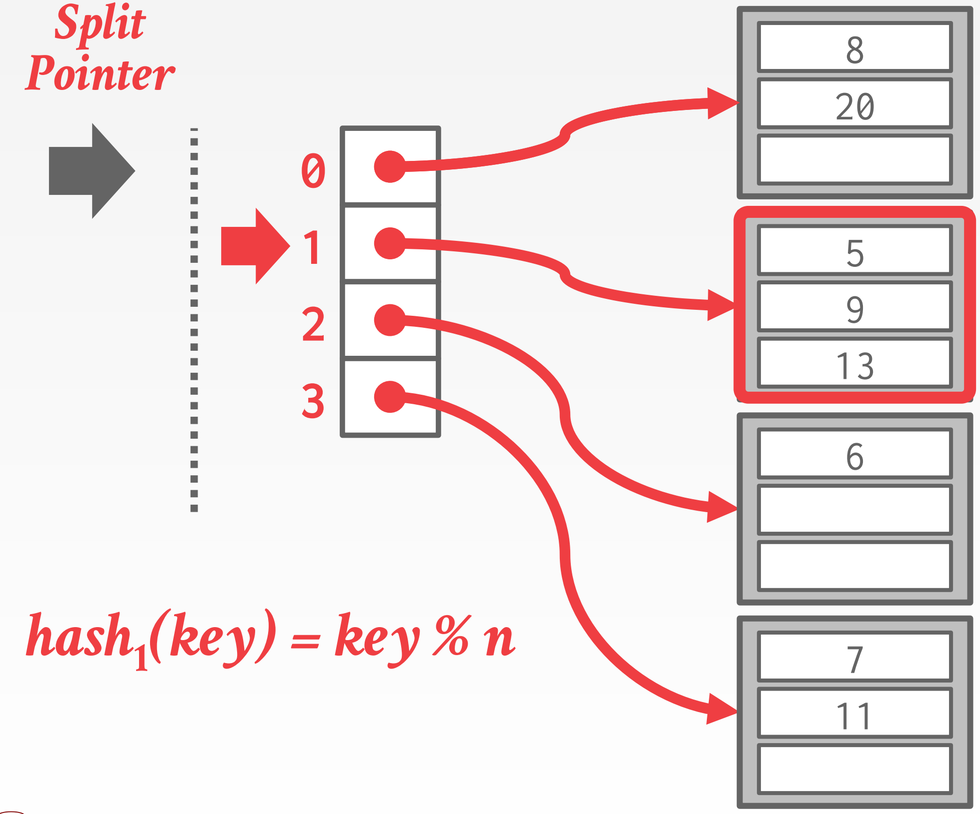


$$hash_1(key) = key \% n$$

Find 6

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

LINEAR HASHING



$hash_1(key) = key \% n$

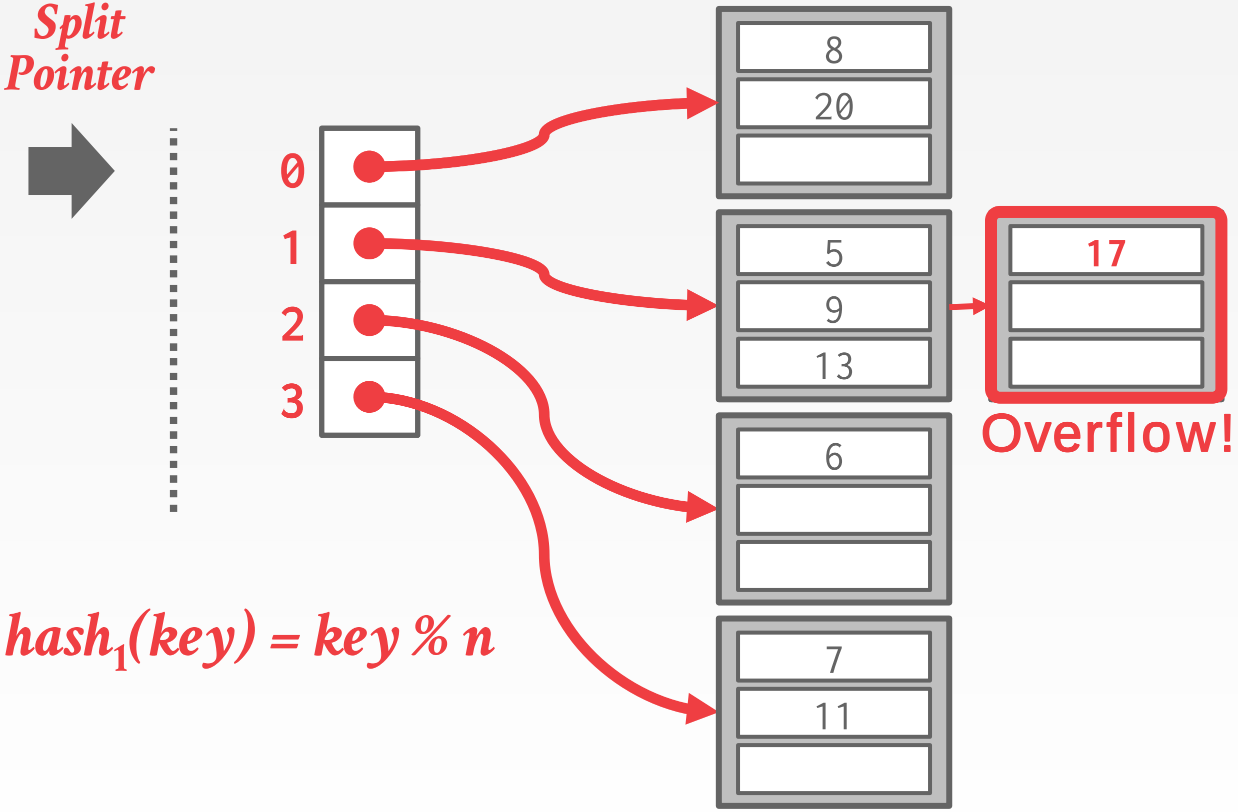
Find 6

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

Insert 17

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

LINEAR HASHING



$hash_1(key) = key \% n$

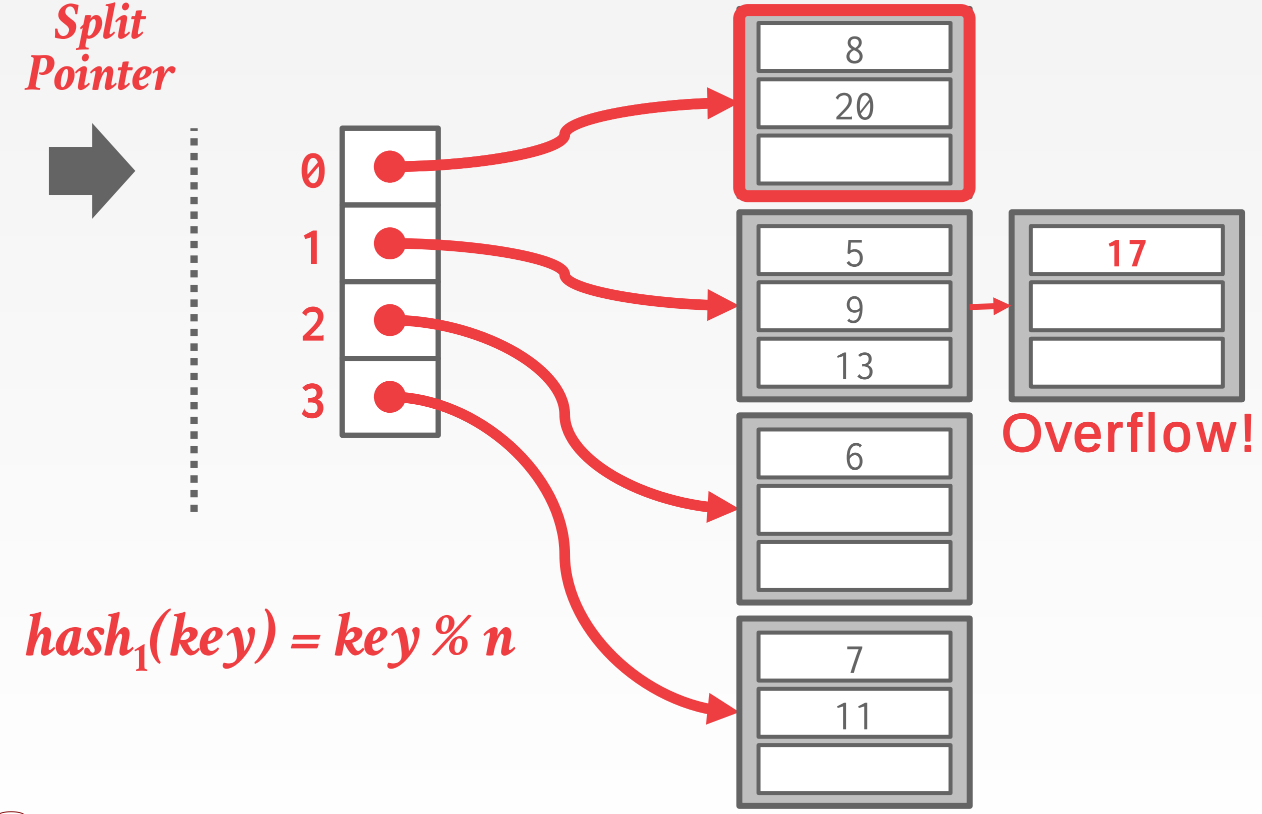
Find 6

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

Insert 17

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

LINEAR HASHING



$$hash_1(key) = key \,\%\, n$$

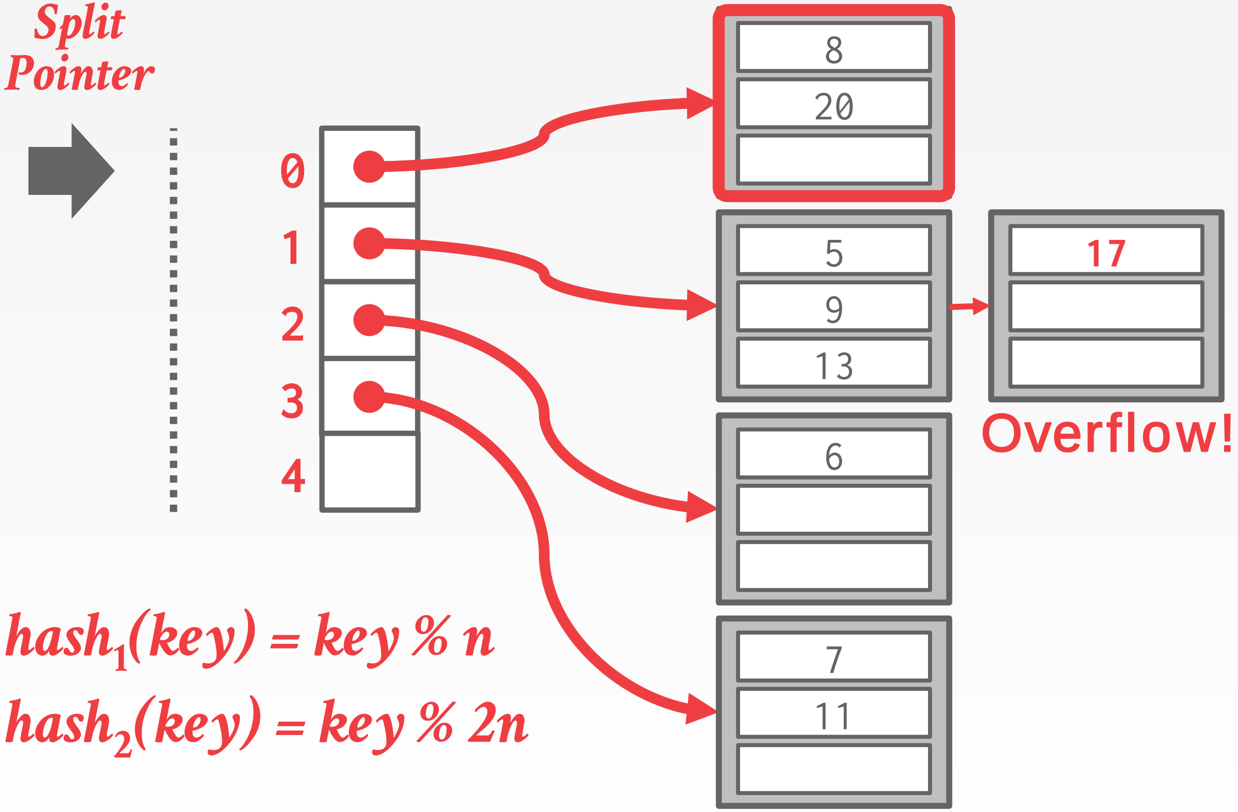
Find 6

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

Insert 17

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

LINEAR HASHING



Find 6

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

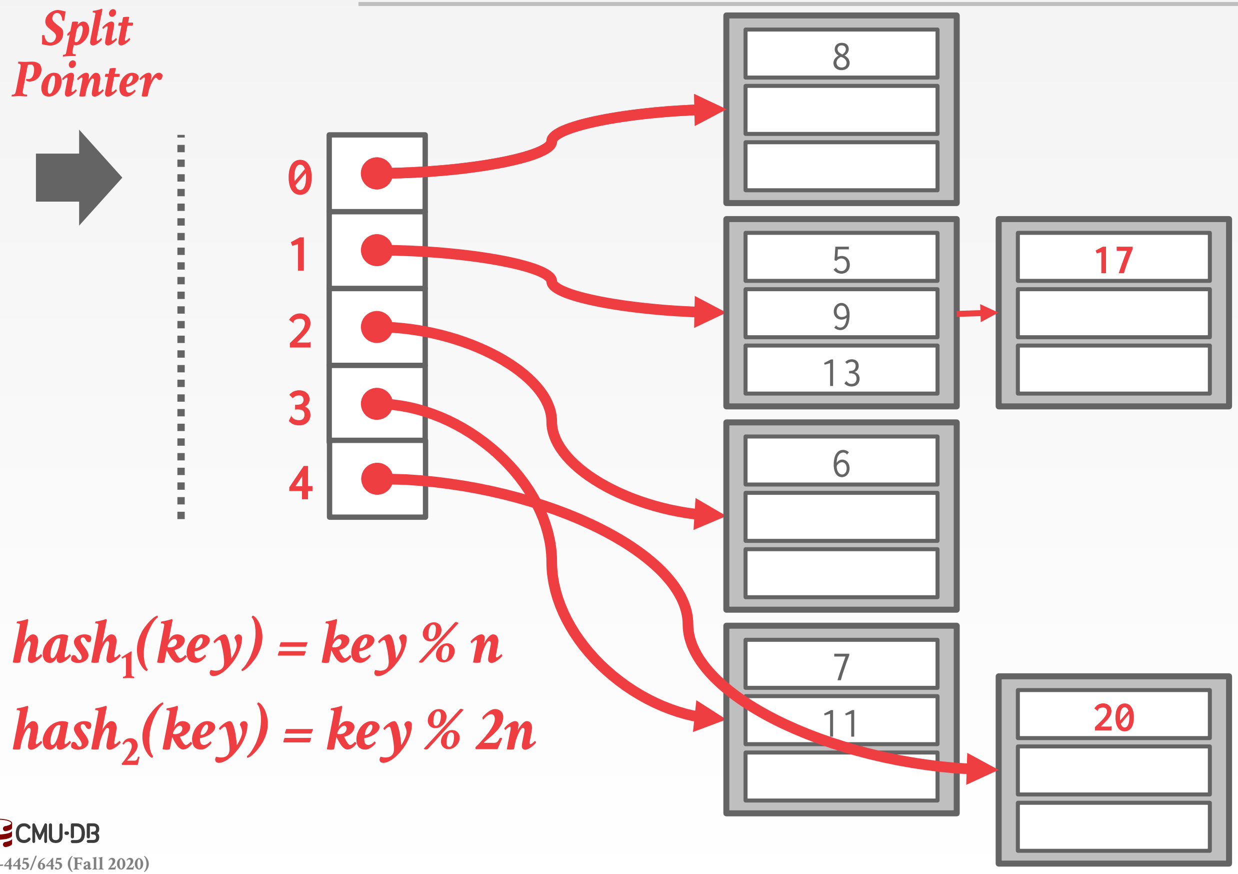
Insert 17

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

$hash_1(key) = key \% n$

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

LINEAR HASHING



Find 6

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

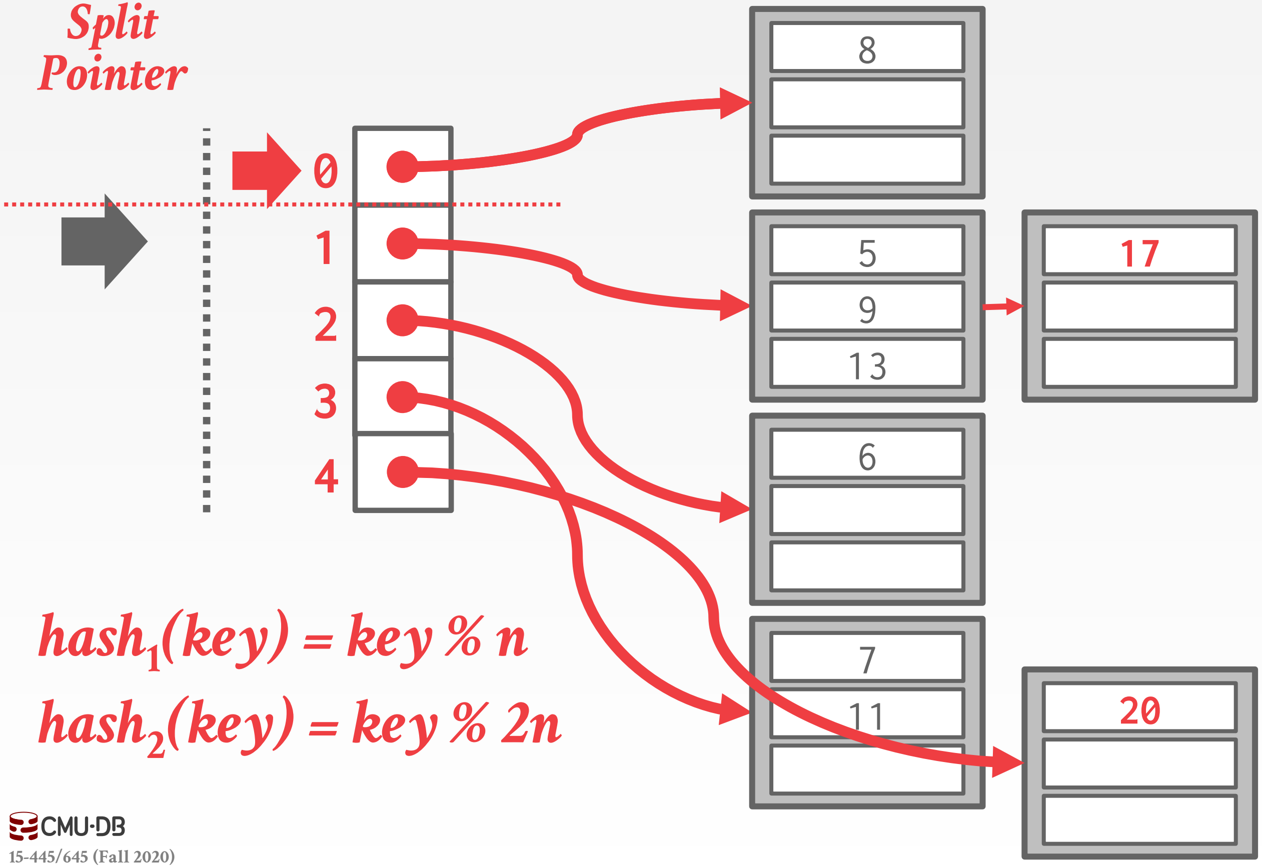
Insert 17

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

$hash_1(key) = key \% n$

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

LINEAR HASHING



Find 6

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

Insert 17

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

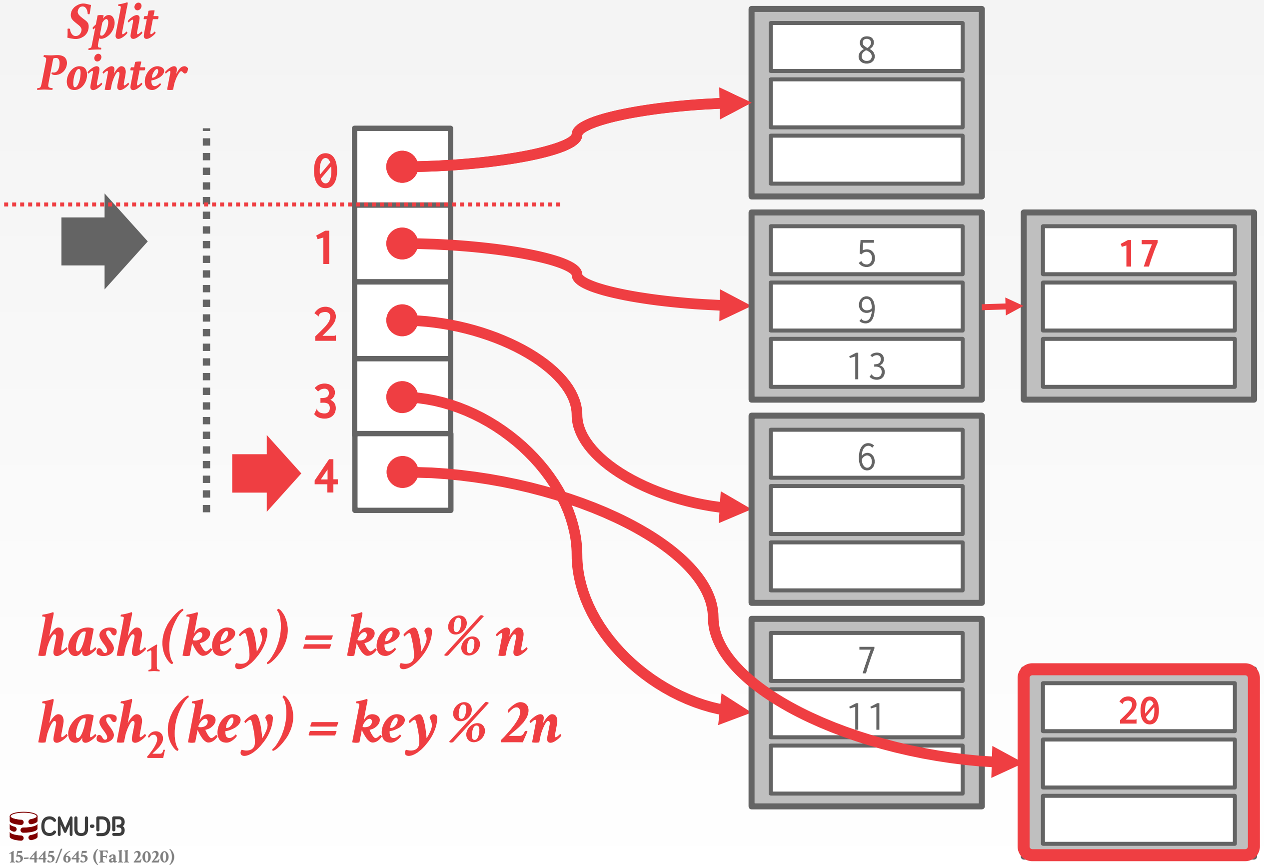
Find 20

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

$hash_1(key) = key \% n$

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

LINEAR HASHING



$hash_1(key) = key \% n$

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

Find 6

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

Insert 17

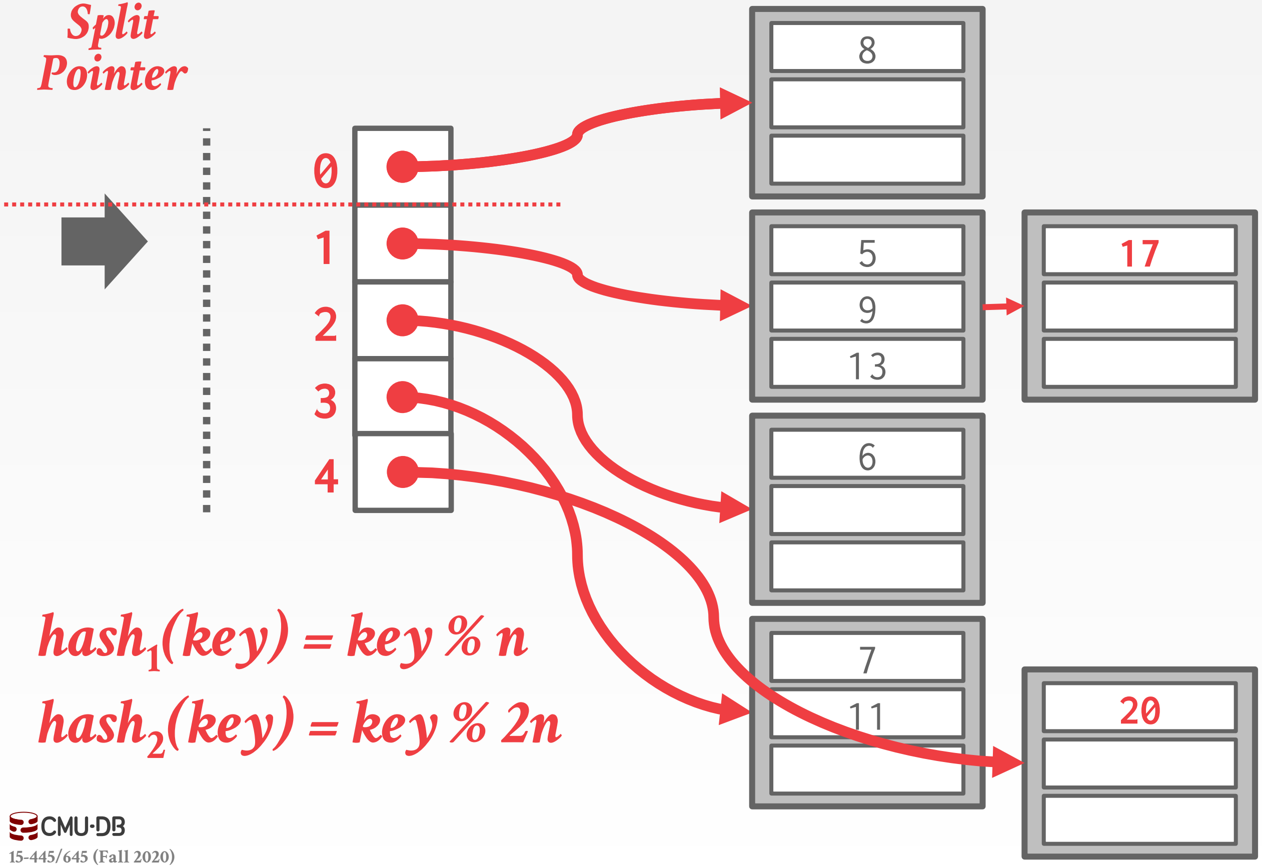
$hash_1(17) = 17 \% 4 = 1$

Find 20

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

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

LINEAR HASHING



$hash_1(key) = key \% n$

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

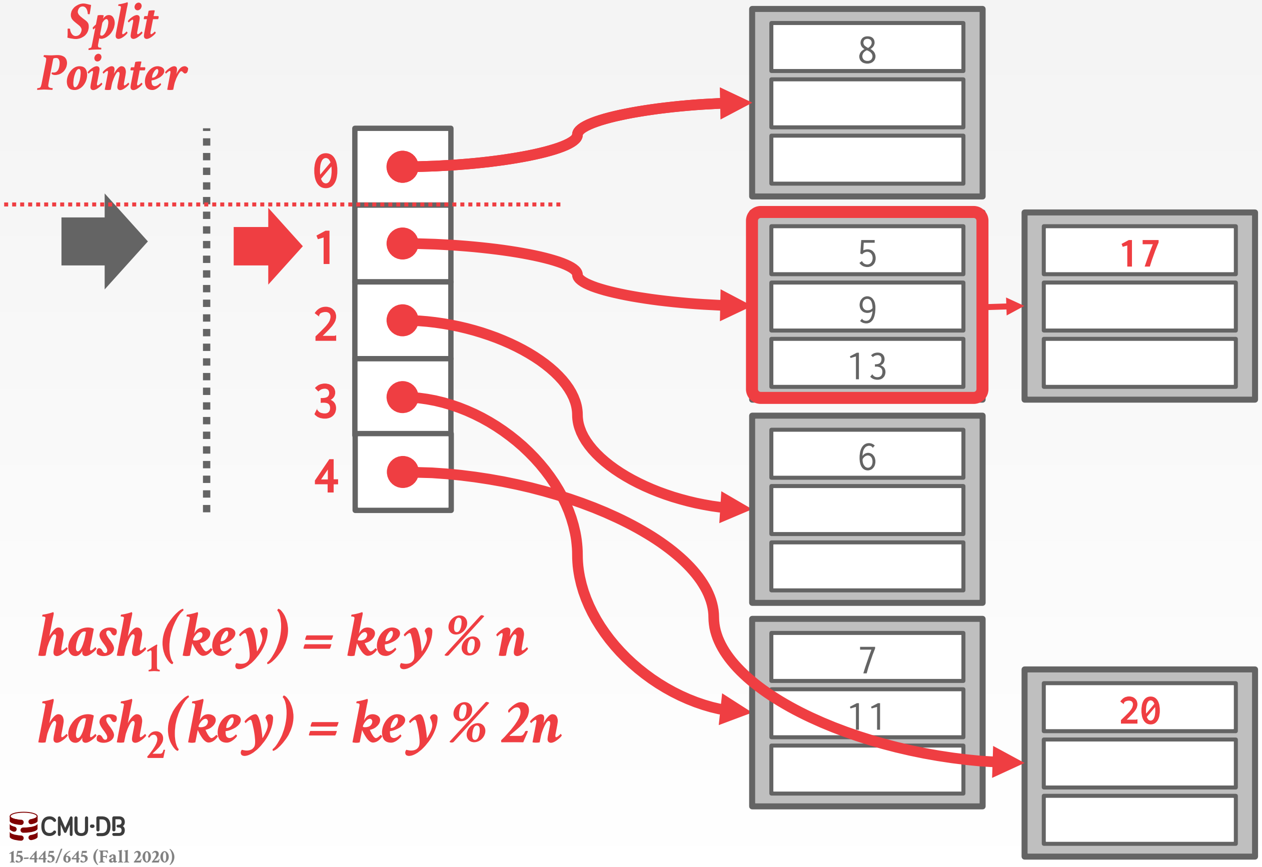
Find 6
$hash_1(6) = 6 \% 4 = 2$

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

Find 20
$hash_1(20) = 20 \% 4 = 0$
$hash_2(20) = 20 \% 8 = 4$

Find 9
$hash_1(9) = 9 \% 4 = 1$

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

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

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

Find 9

$hash_1(9) = 9 \% 4 = 1$

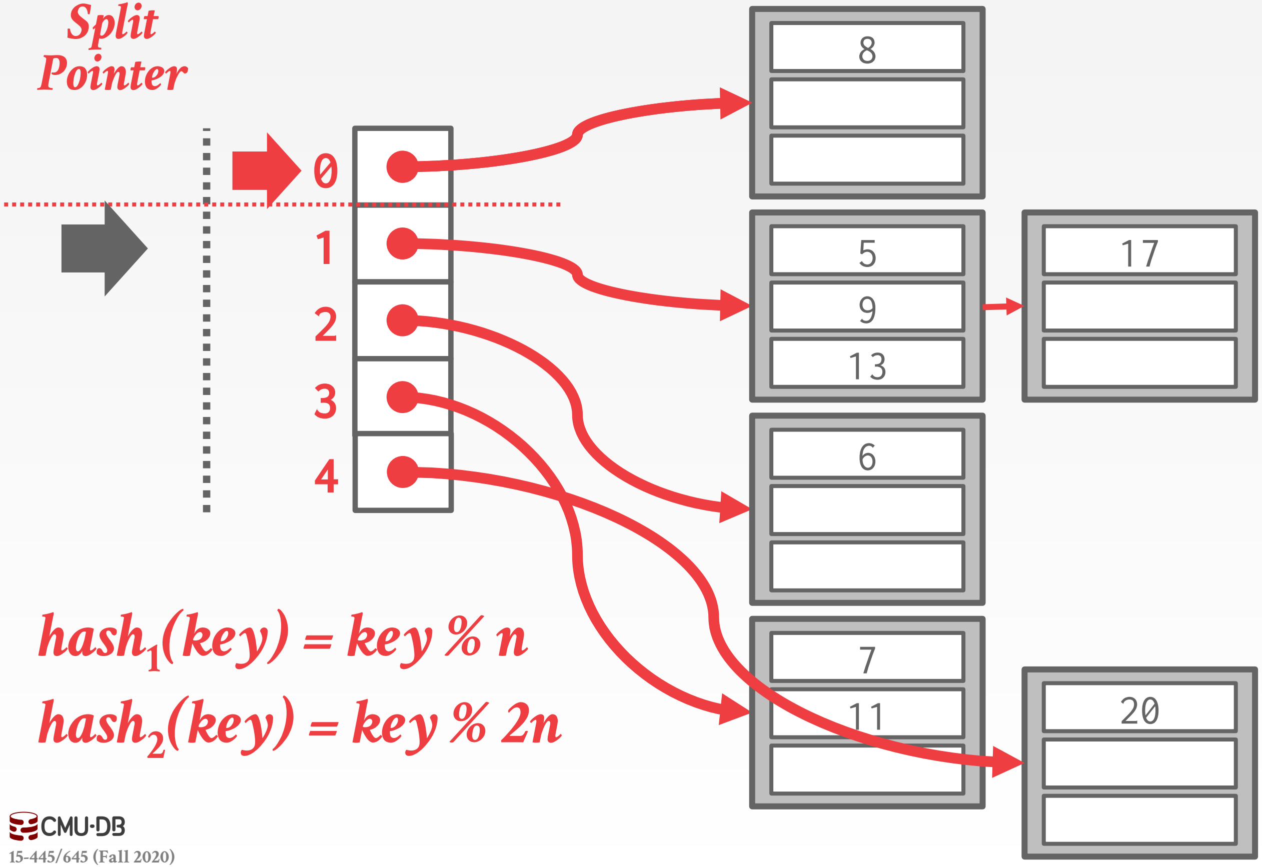
LINEAR HASHING

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.

LINEAR HASHING – DELETES



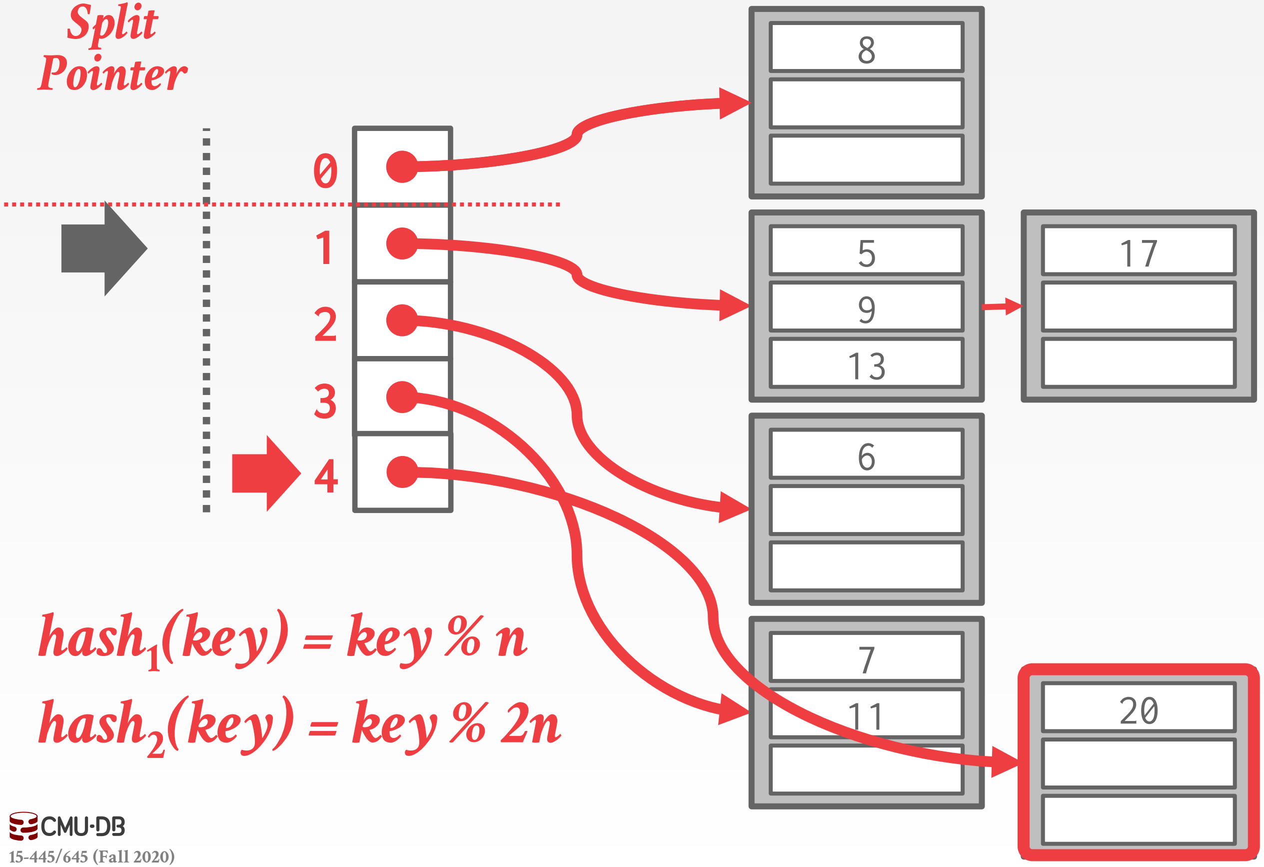
Delete 20

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

$hash_1(key) = key \% n$

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

LINEAR HASHING – DELETES



Delete 20

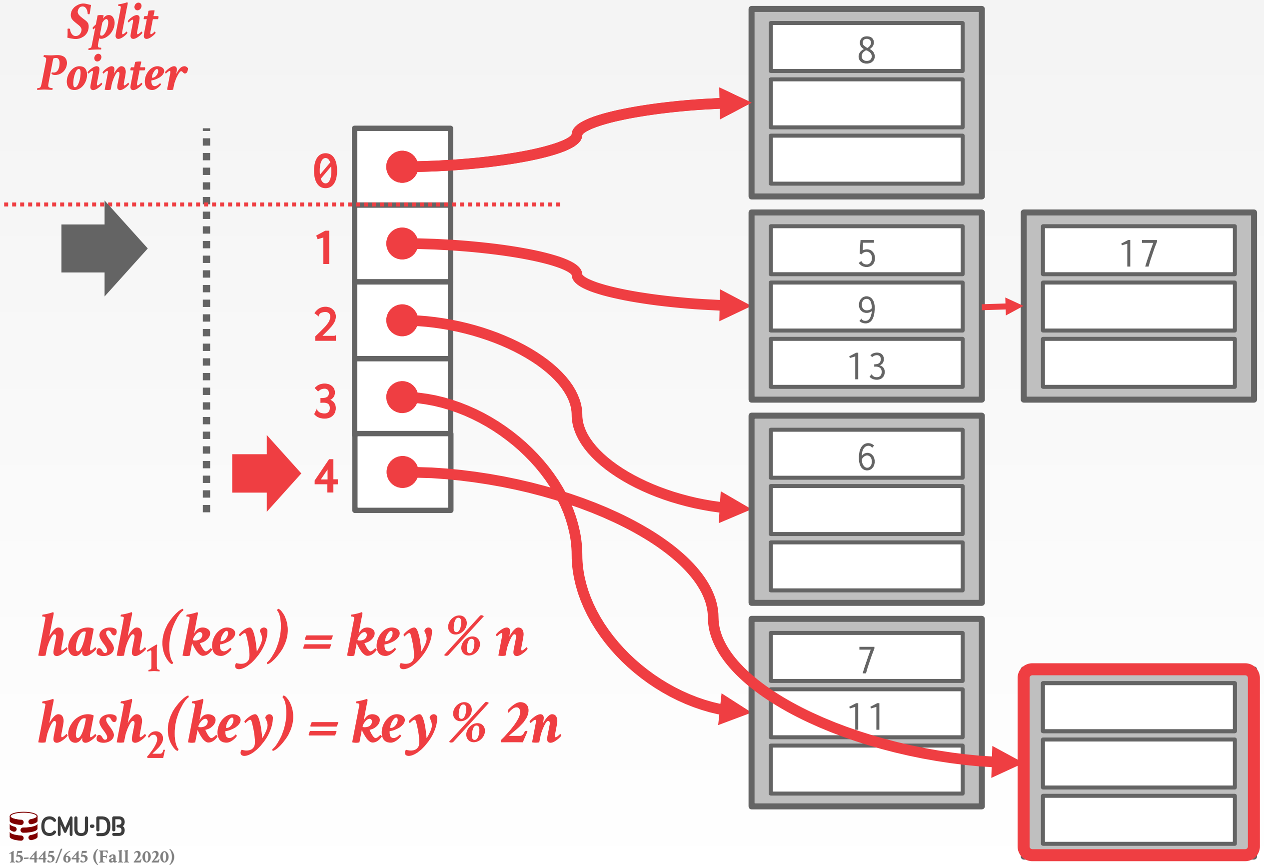
$hash_1(20) = 20 \% 4 = 0$

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

$hash_1(key) = key \% n$

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

LINEAR HASHING – DELETES



Delete 20

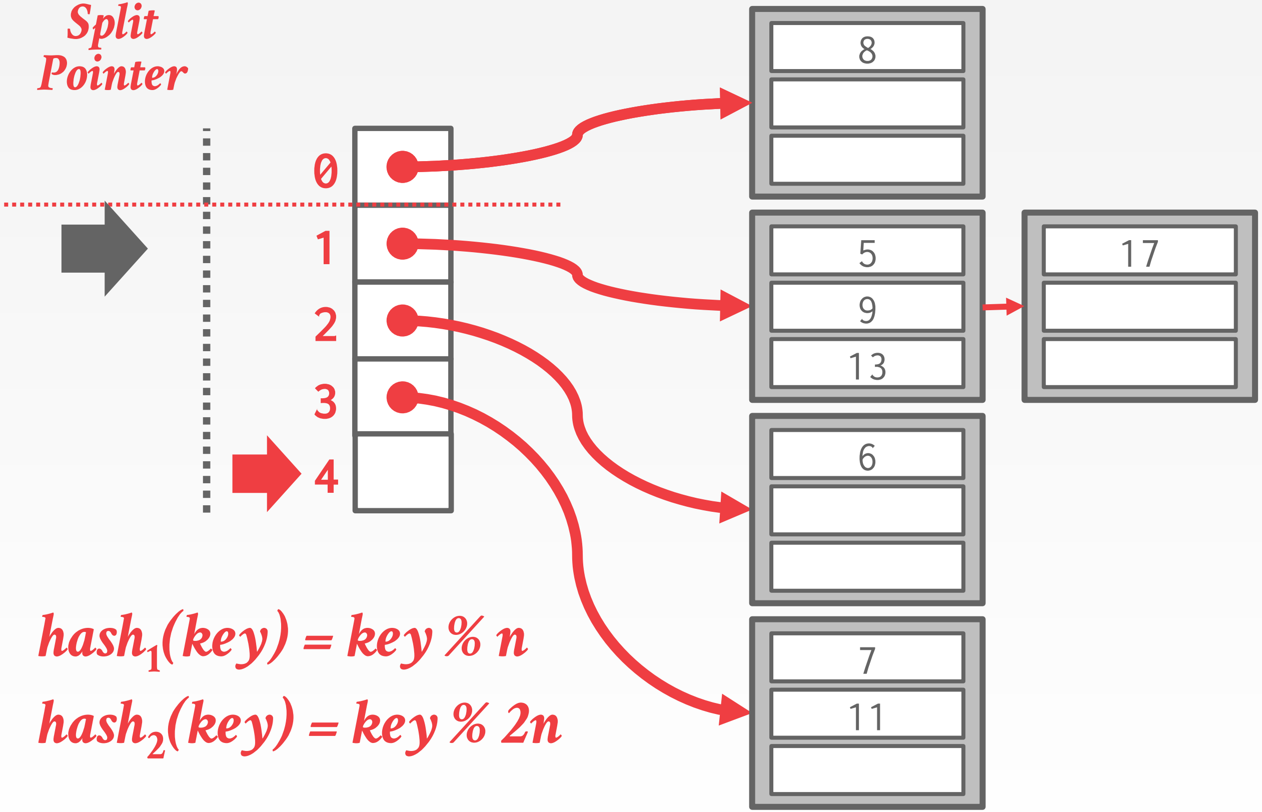
$hash_1(20) = 20 \% 4 = 0$

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

$hash_1(key) = key \% n$

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

LINEAR HASHING – DELETES



Delete 20

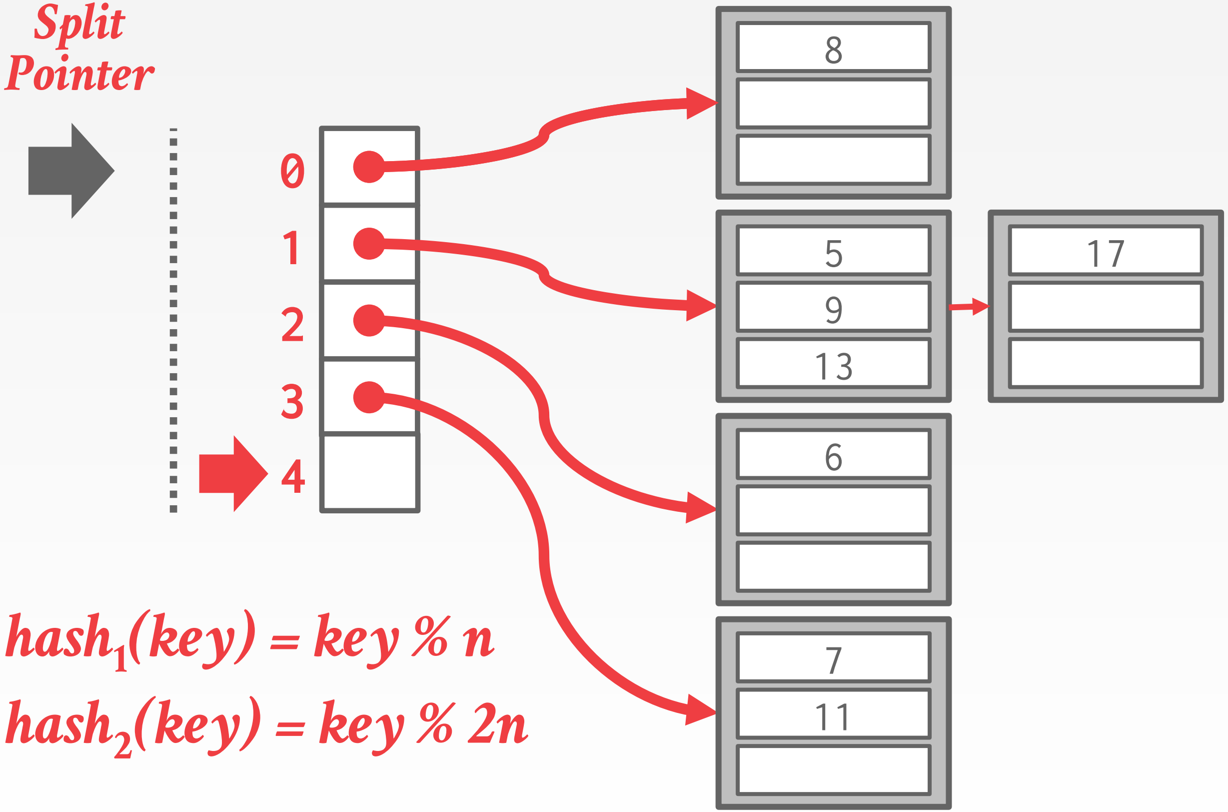
$hash_1(20) = 20 \% 4 = 0$

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

$hash_1(key) = key \% n$

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

LINEAR HASHING – DELETES



`Delete 20`

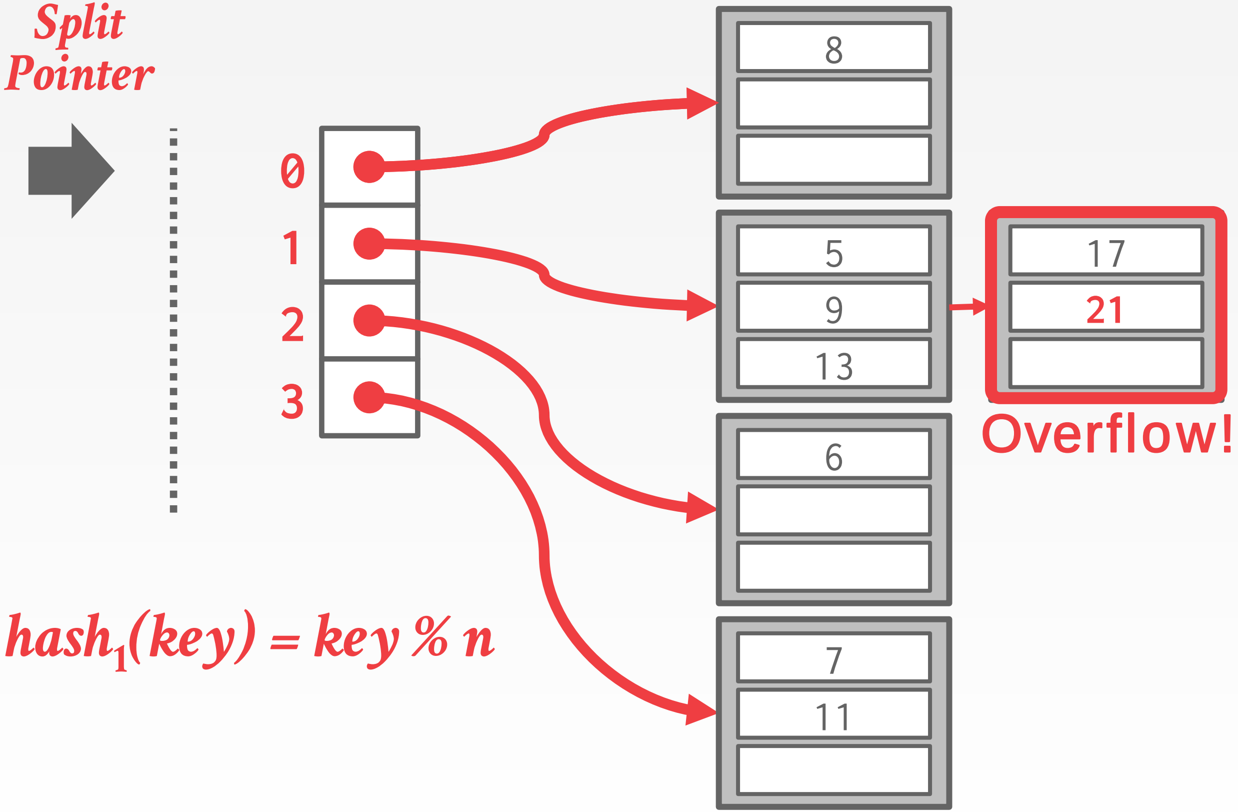
$hash_1(20) = 20 \% 4 = 0$

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

$hash_1(key) = key \% n$

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

LINEAR HASHING – DELETES



$hash_1(key) = key \% n$

Delete 20

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

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

Insert 21

$hash_1(21) = 21 \% 4 = 1$

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…

NEXT CLASS

B+Trees

→ aka "The Greatest Data Structure of All Time!"