University of Michigan

15 Hash Tables

Database Management Systems
EECS 484
Fall 2024

LM
Lin Ma
Computer Science and
Engineering Division

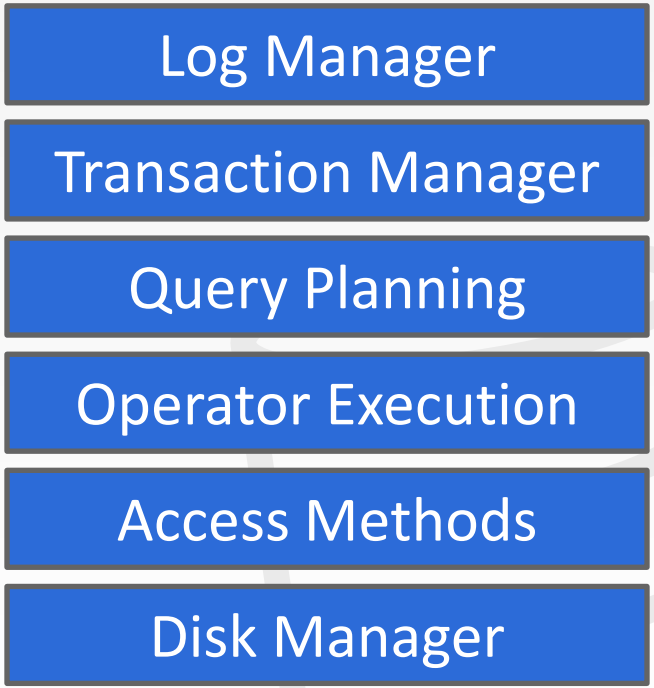
COURSE STATUS

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



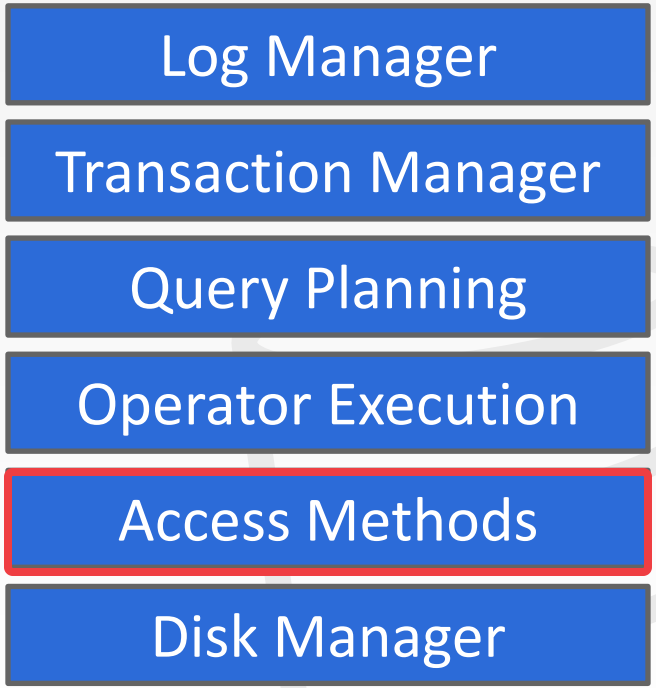
COURSE STATUS

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

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)**
Time Complexity:
⟶ Average: **O(1)**
⟶ Worst: **O(n)**

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)**

Time Complexity: ⬅ ***Need to care about constants!***

→ Average: **O(1)**

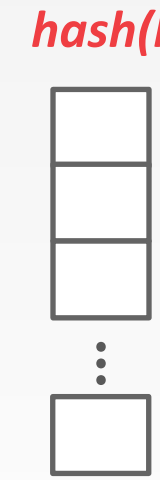
→ Worst: **O(n)**

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.

What assumptions do we make?

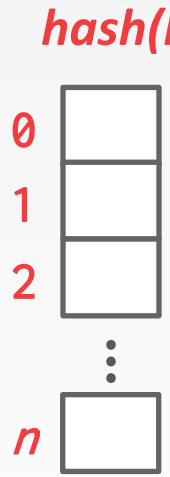


STATIC HASH TABLE

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.

What assumptions do we make?

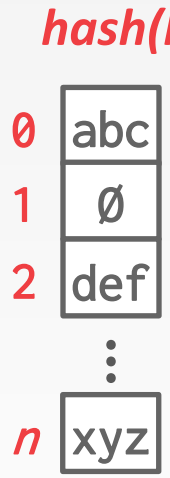


STATIC HASH TABLE

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.

What assumptions do we make?

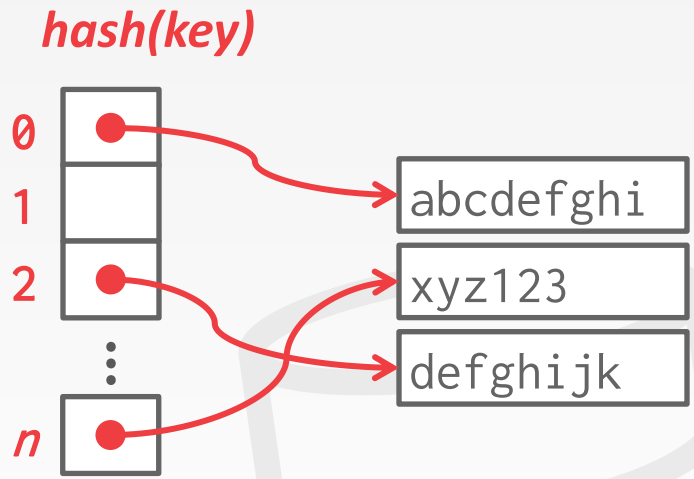


STATIC HASH TABLE

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.

What assumptions do we make?



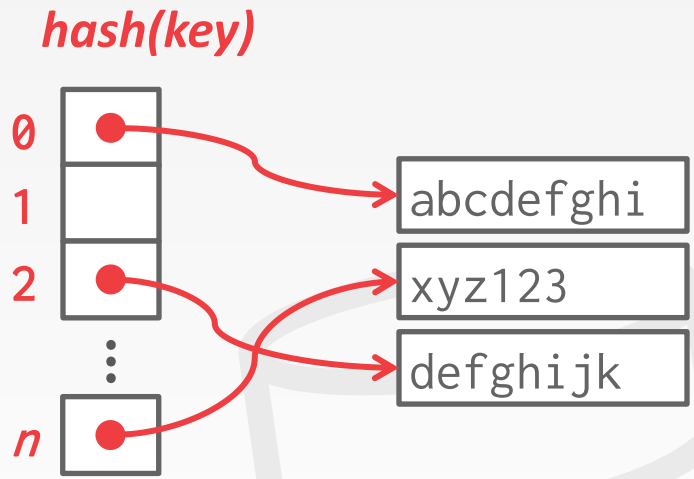
ASSUMPTIONS

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

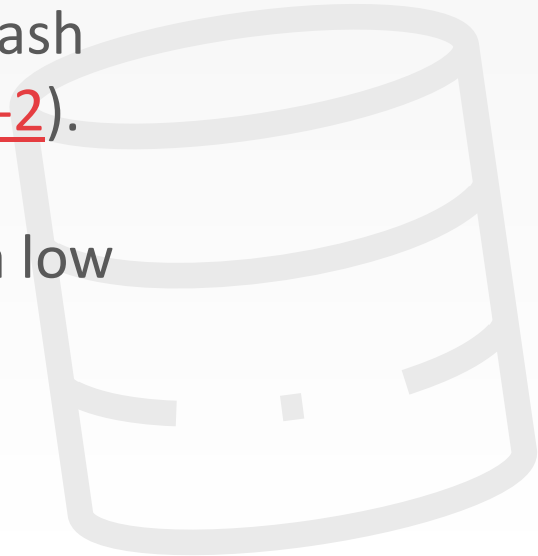
For any input key, return an integer representation of that key.

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 (e.g., SHA-2).

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 as a fast, general-purpose hash function.

Google CityHash (2011)

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

Facebook XXHash (2012)

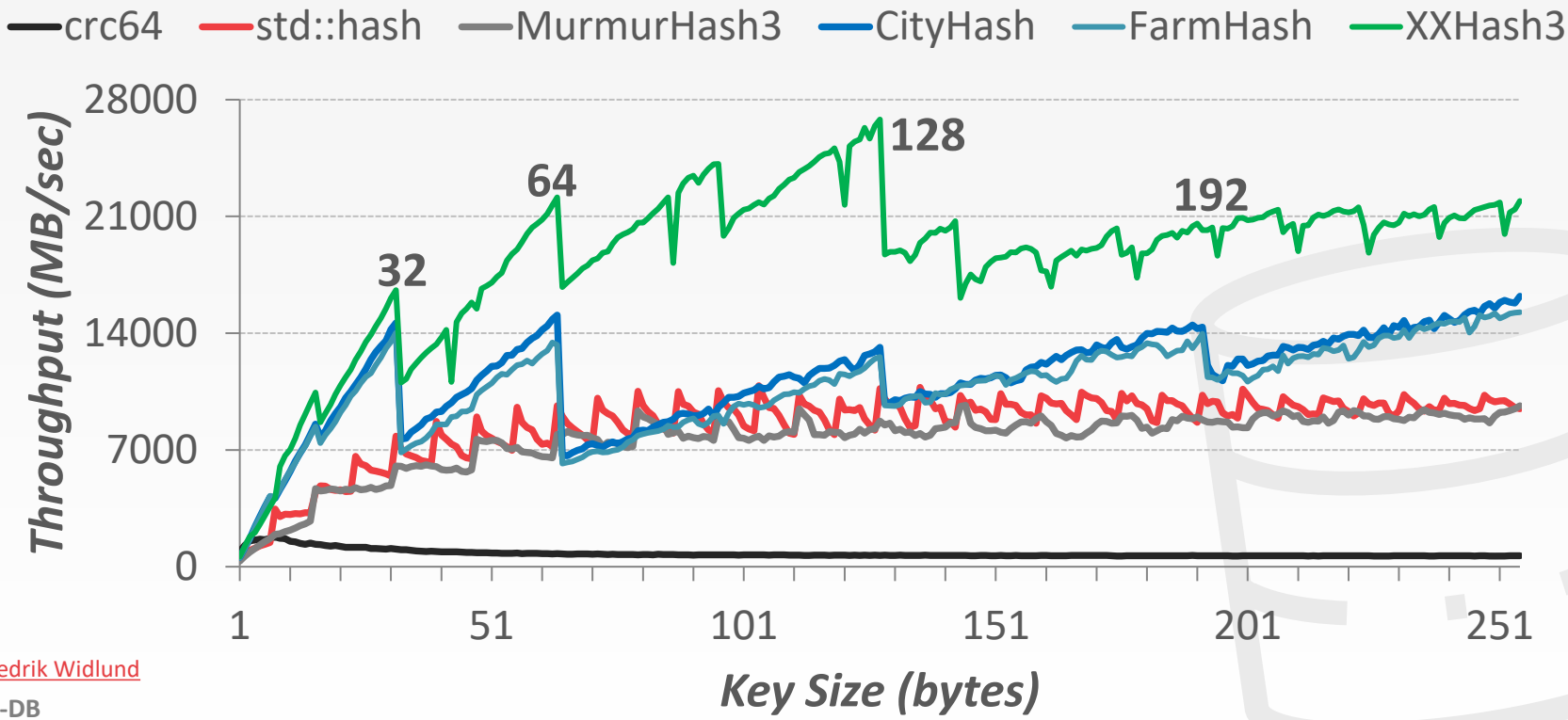
→ From the creator of zstd compression.

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

What Python implements for dictionaries

Single giant table of slots.

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

→ To insert a key, hash to a location in the index and scan for the next free slot

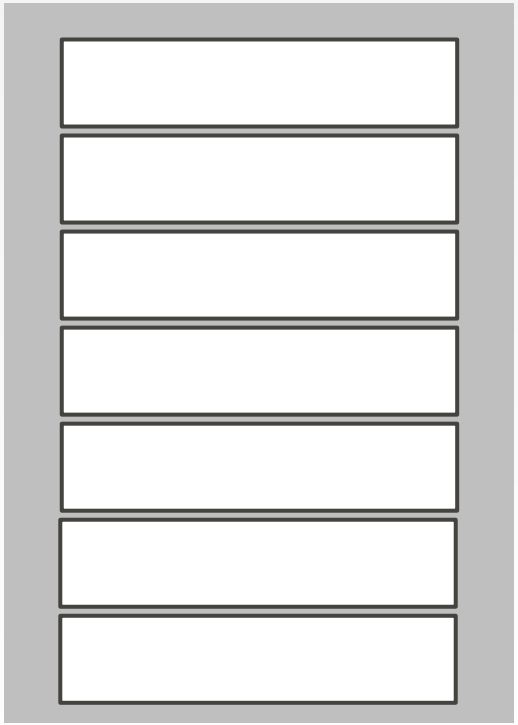
→ Must store the key in the index to know whether the key already exists in the hash table

→ Lookups and deletions are generalizations of inserts.

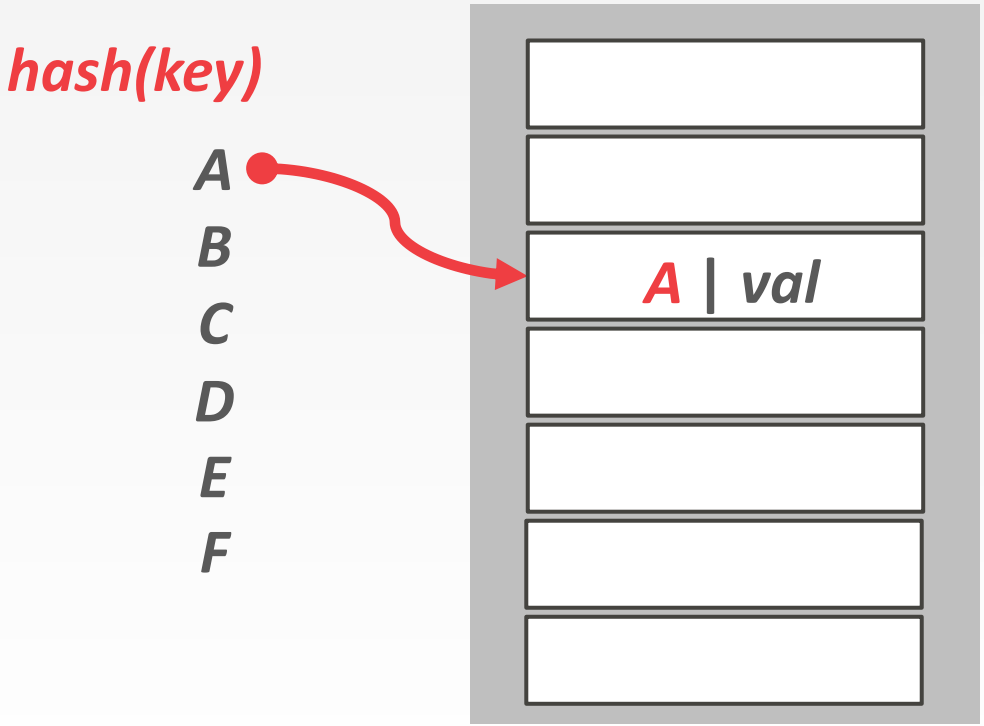
LINEAR PROBE HASHING

hash(key)

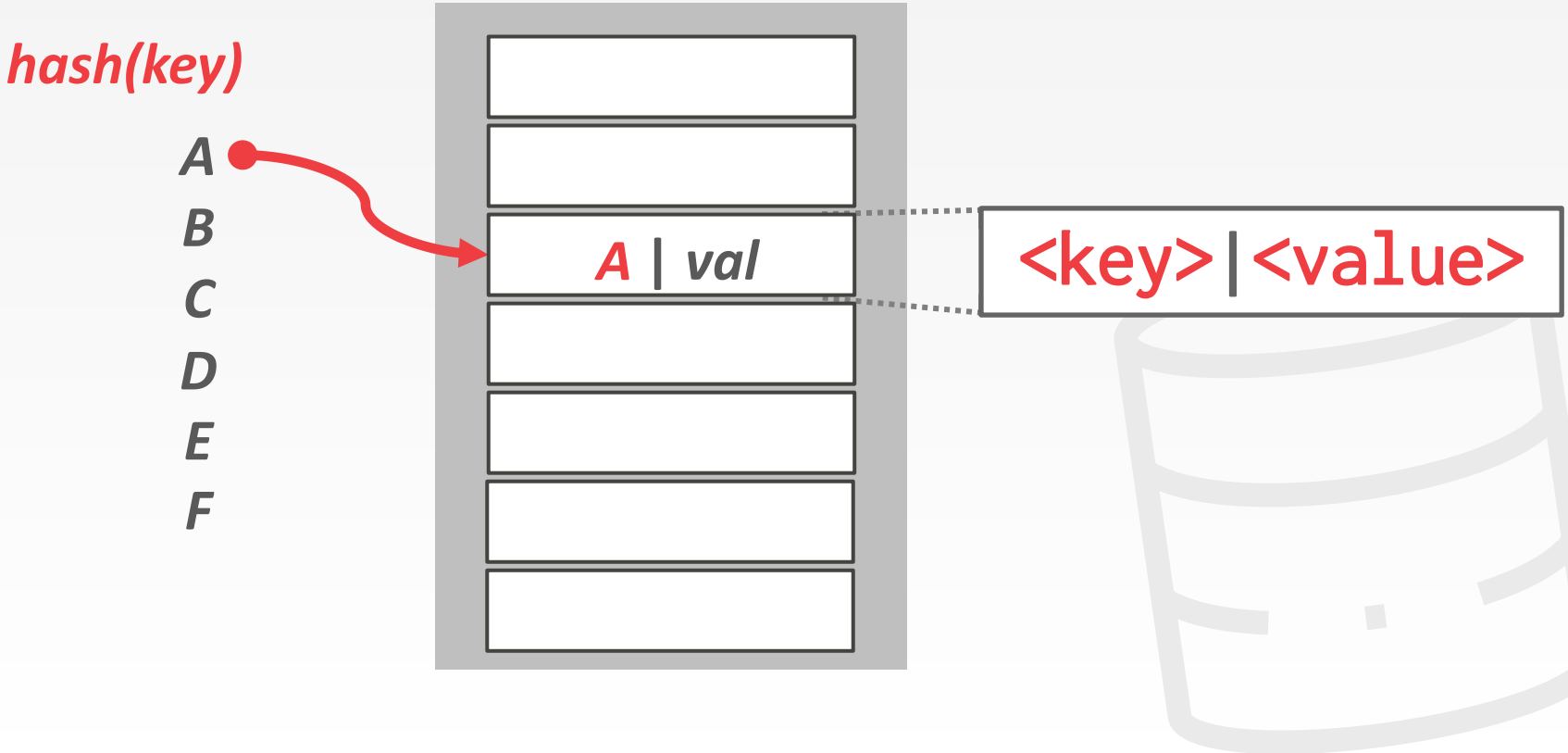
A
B
C
D
E
F



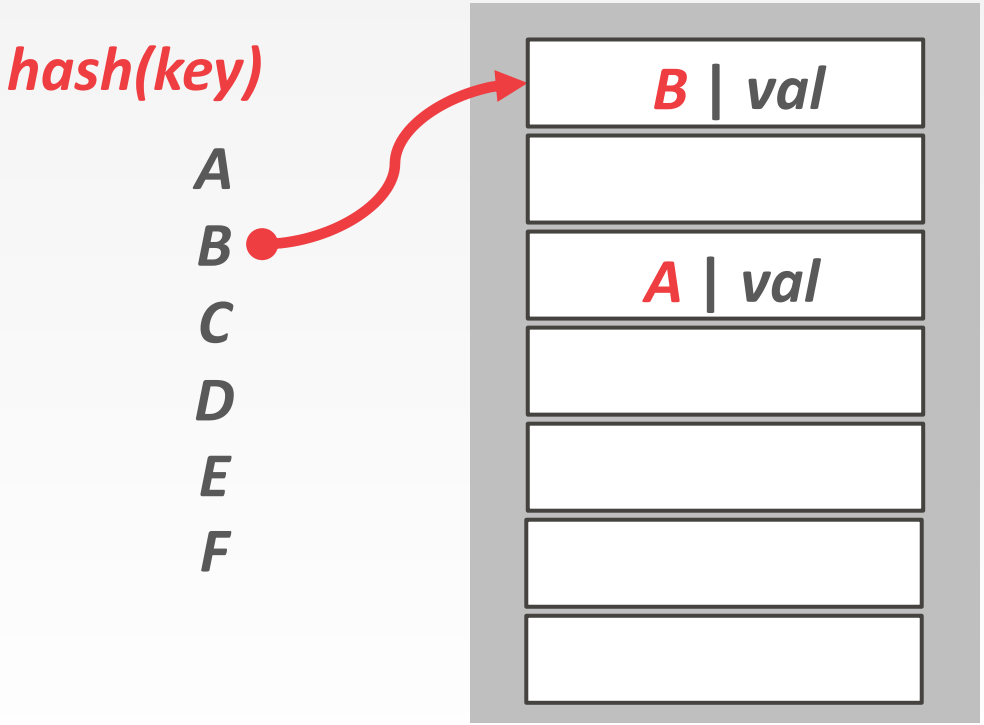
LINEAR PROBE HASHING



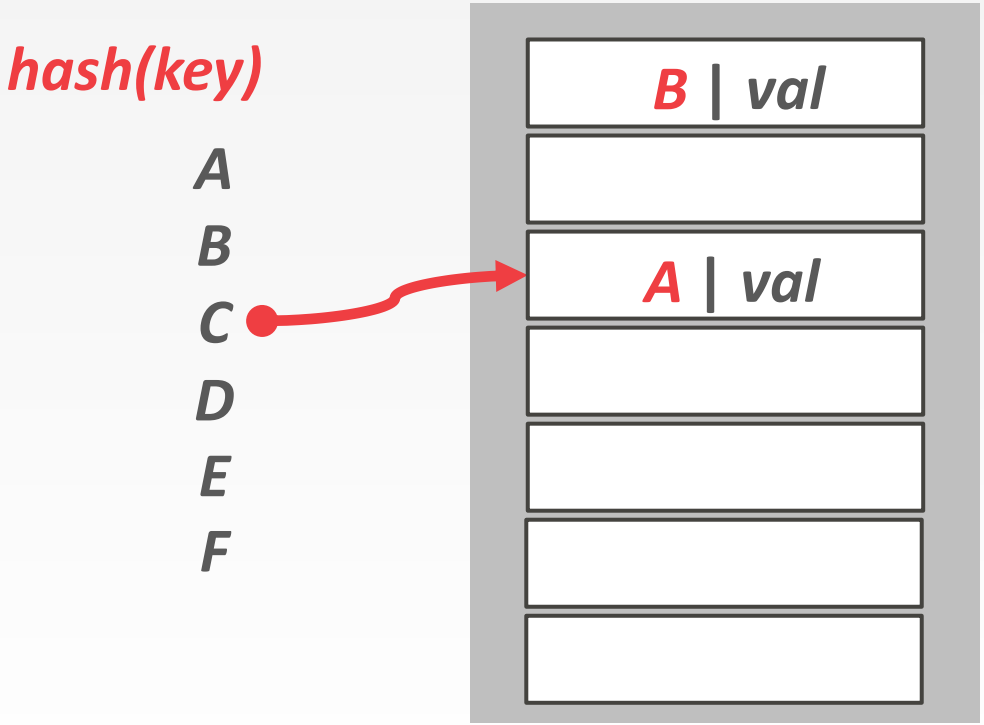
LINEAR PROBE HASHING



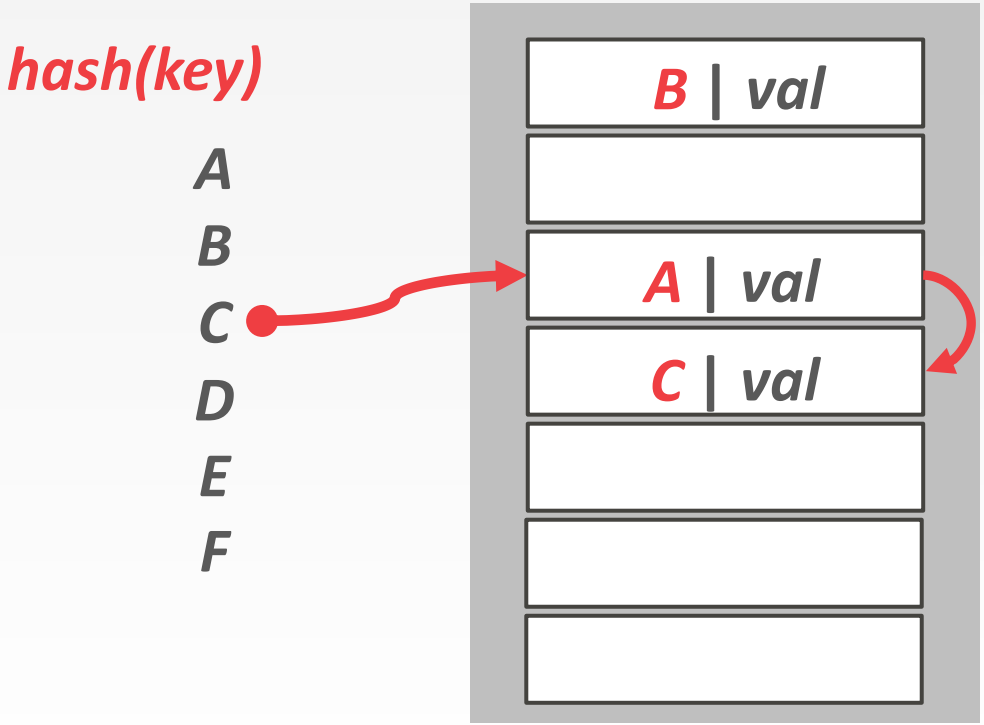
LINEAR PROBE HASHING



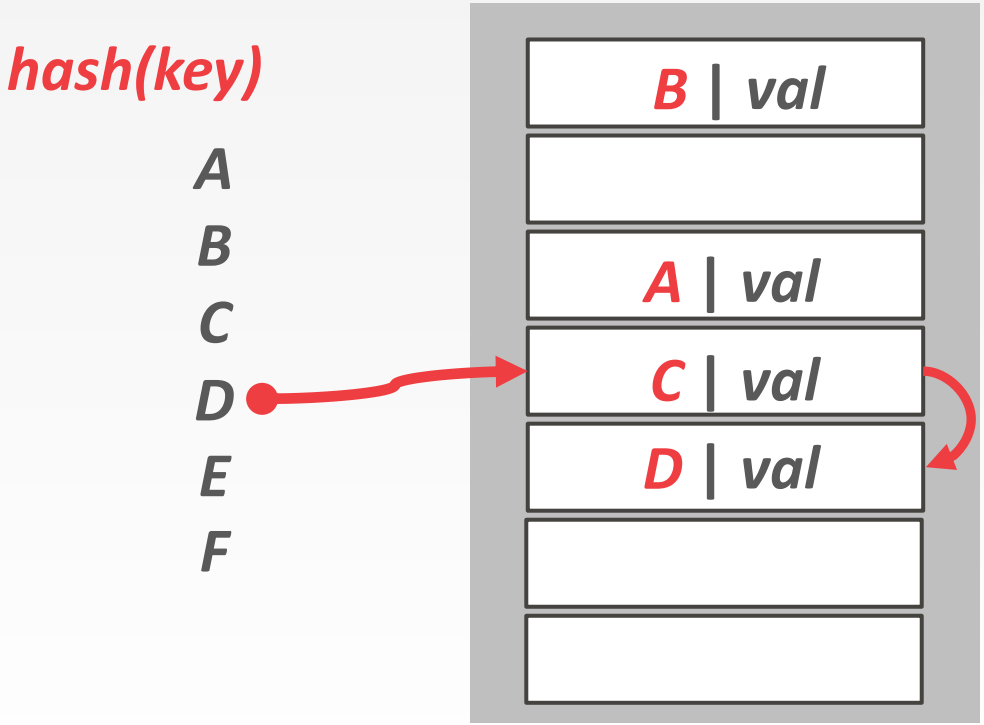
LINEAR PROBE HASHING



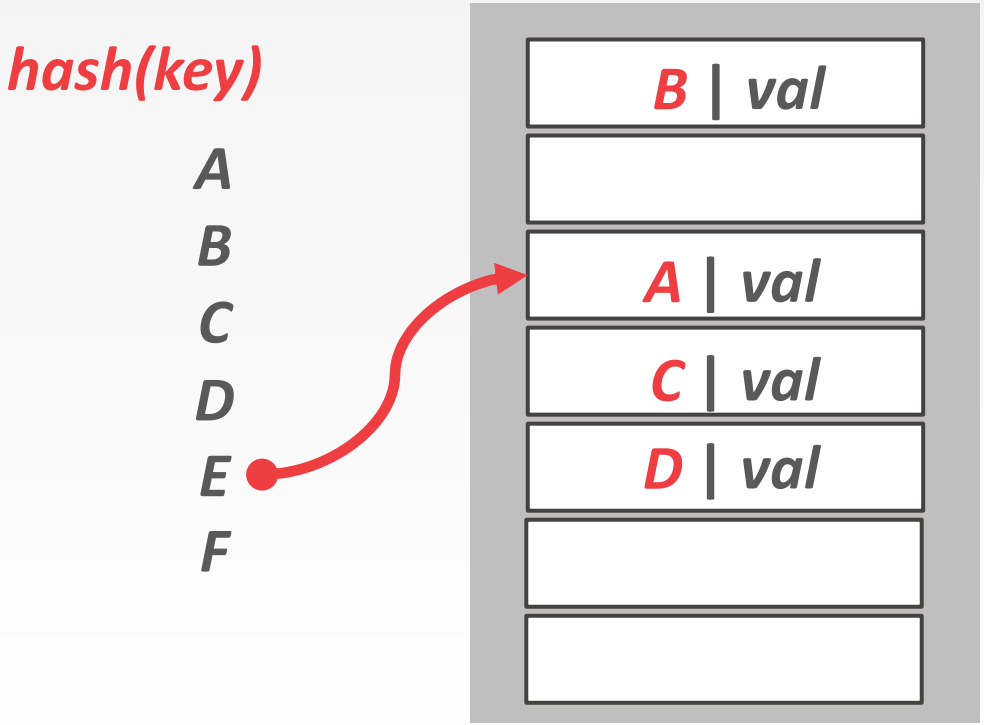
LINEAR PROBE HASHING



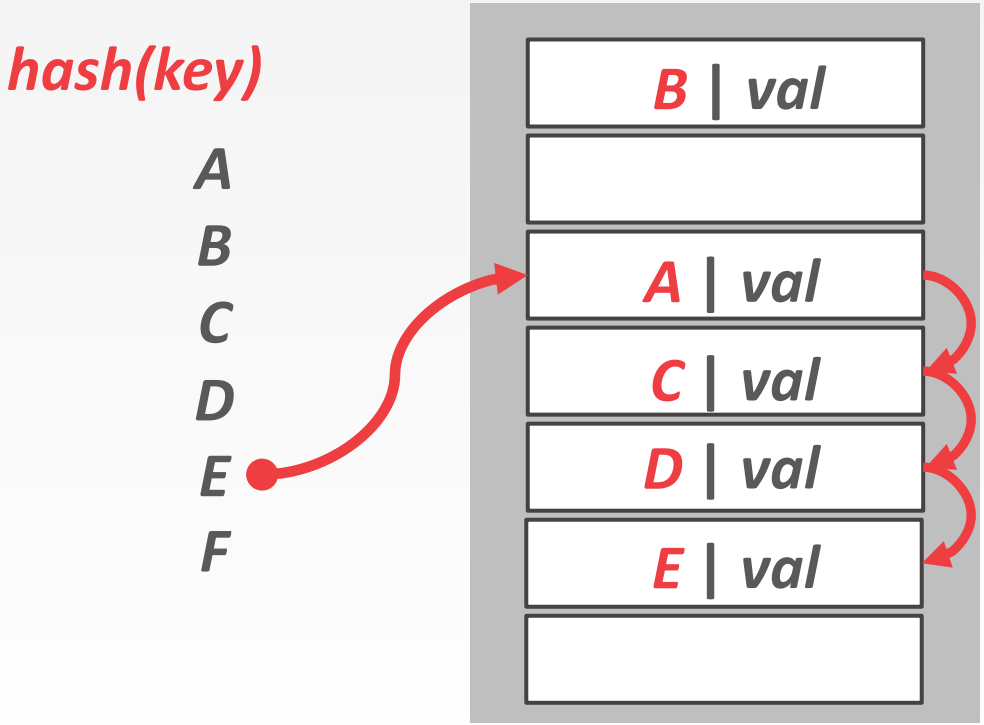
LINEAR PROBE HASHING



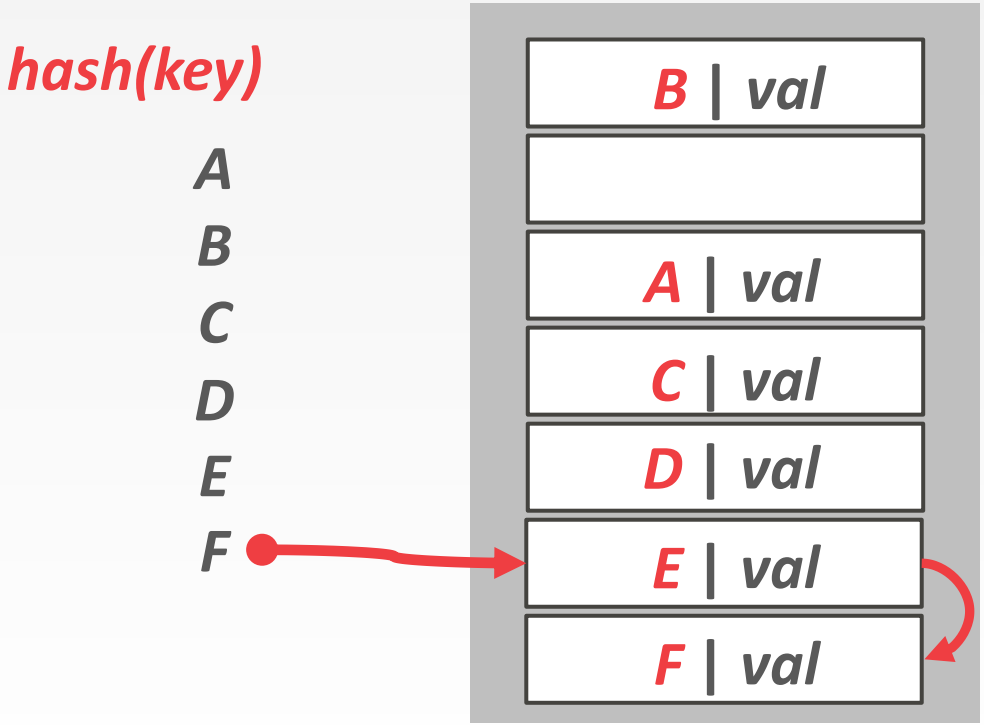
LINEAR PROBE HASHING



LINEAR PROBE HASHING



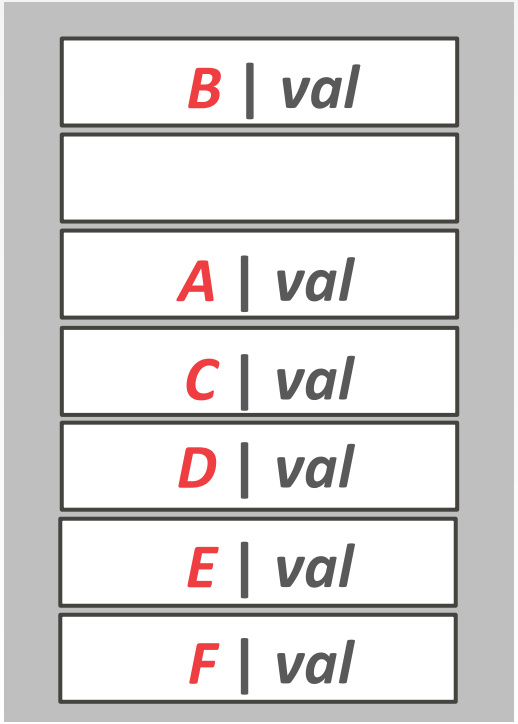
LINEAR PROBE HASHING



LINEAR PROBE HASHING – DELETES

hash(key)

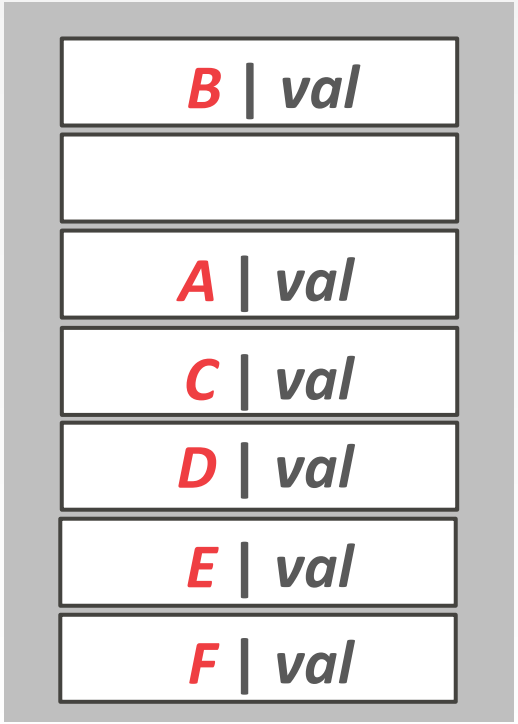
A
B
C
D
E
F



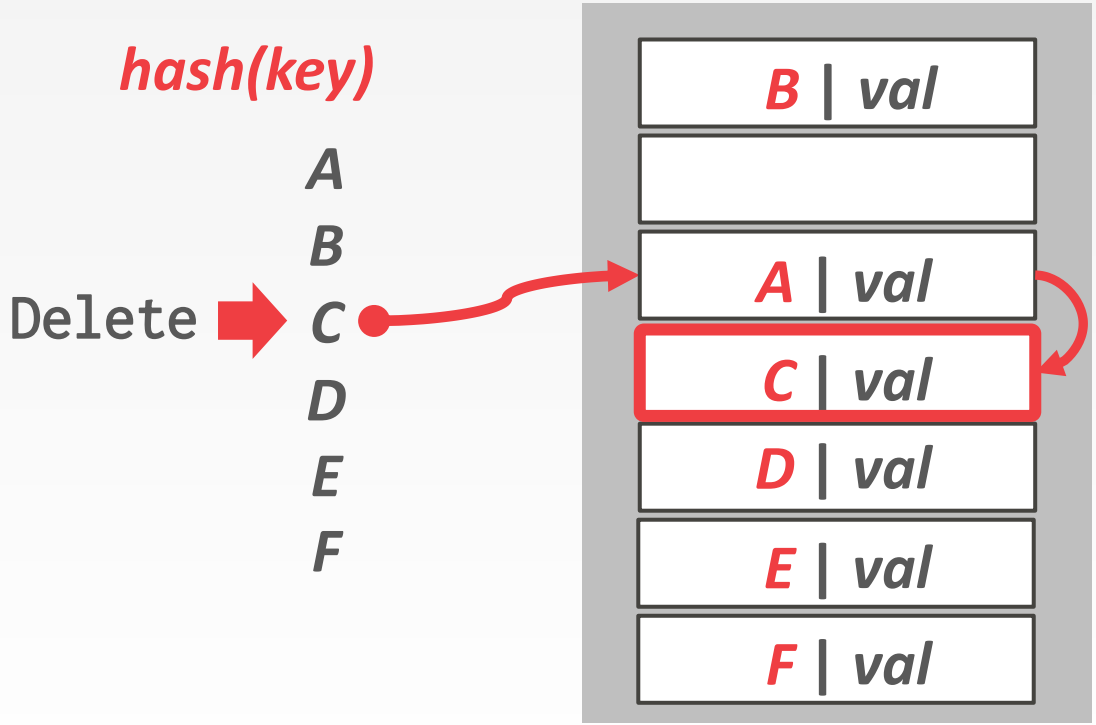
LINEAR PROBE HASHING – DELETES

hash(key)

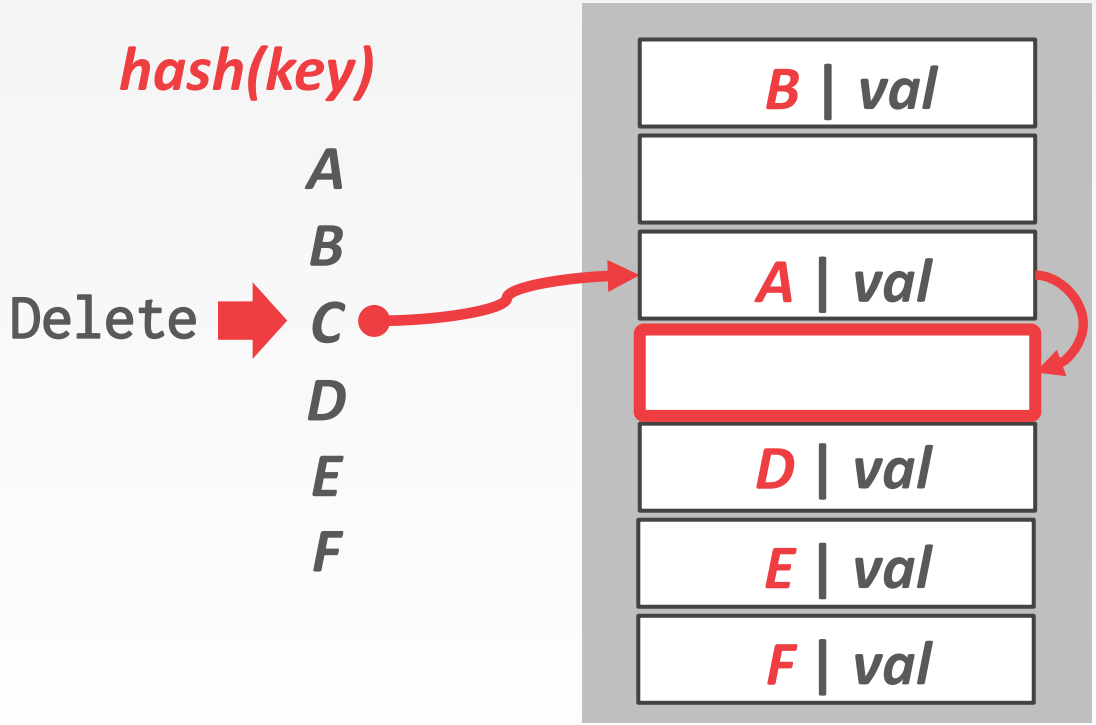
A
B
Delete → *C*
D
E
F



LINEAR PROBE HASHING – DELETES



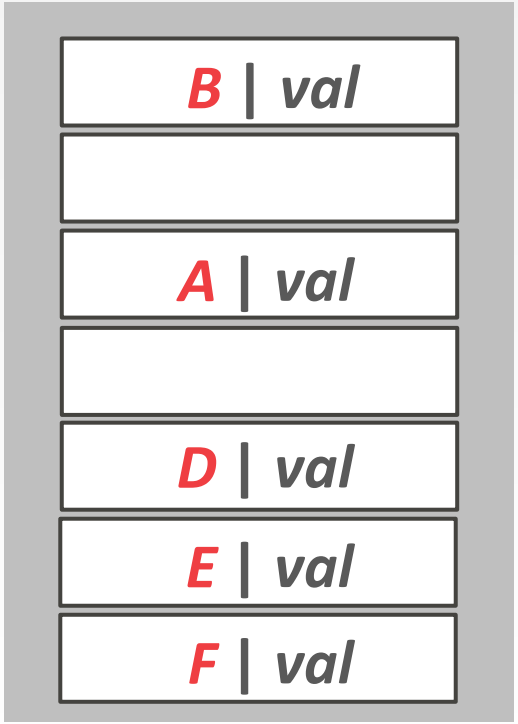
LINEAR PROBE HASHING – DELETES



LINEAR PROBE HASHING – DELETES

hash(key)

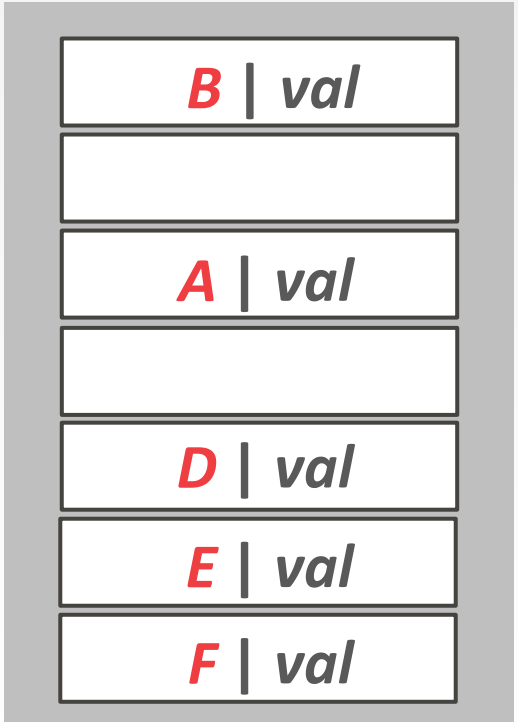
A
B
C
D
E
F



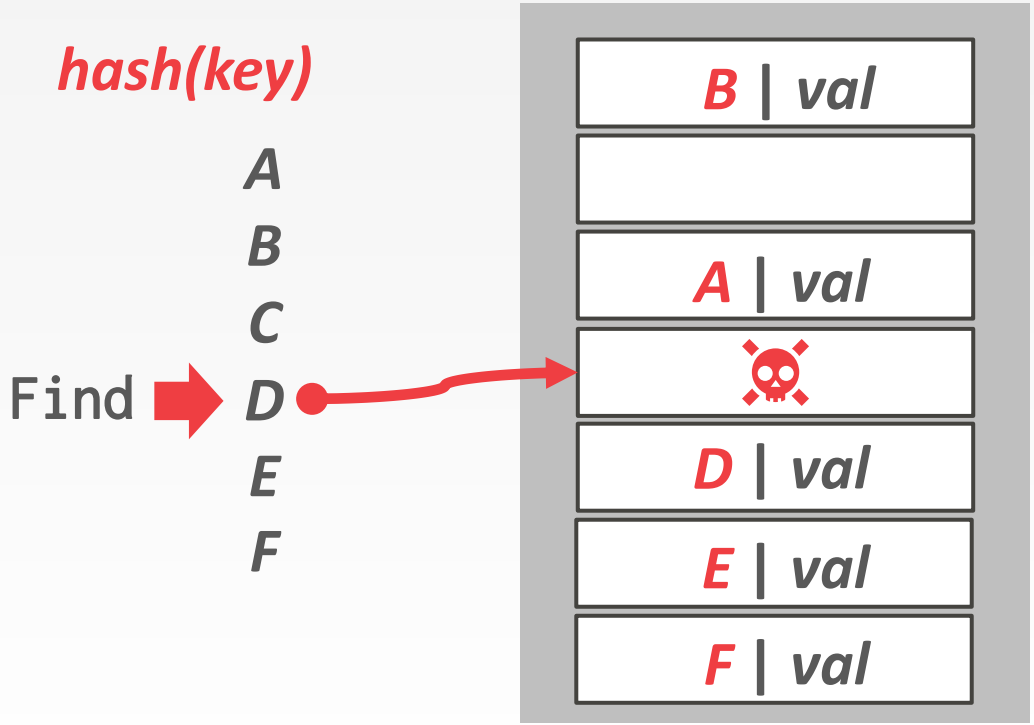
LINEAR PROBE HASHING – DELETES

hash(key)

A
B
C
Find → *D*
E
F



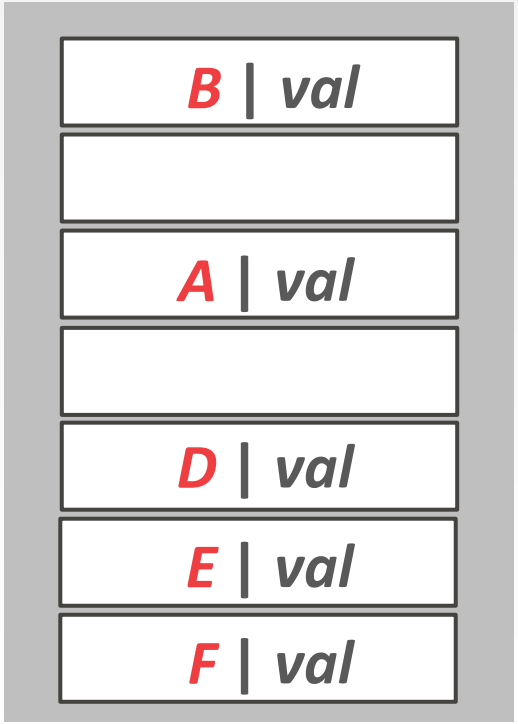
LINEAR PROBE HASHING – DELETES



LINEAR PROBE HASHING – DELETES

hash(key)

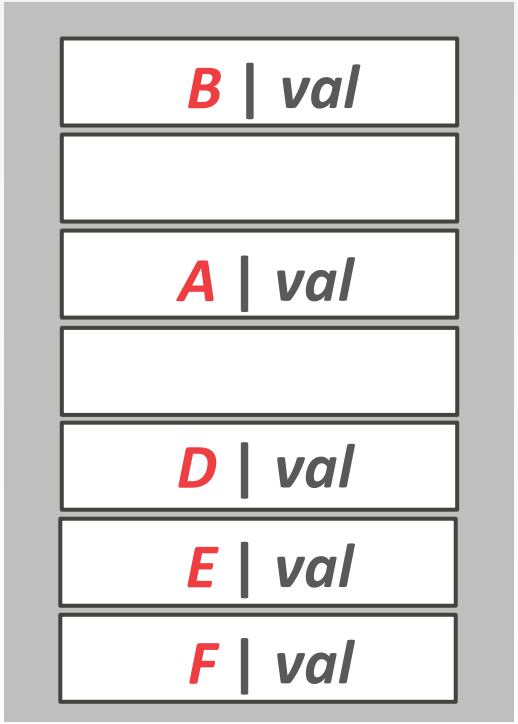
A
B
C
Find → *D*
E
F



LINEAR PROBE HASHING – DELETES

hash(key)

A
B
C
Find → *D*
E
F

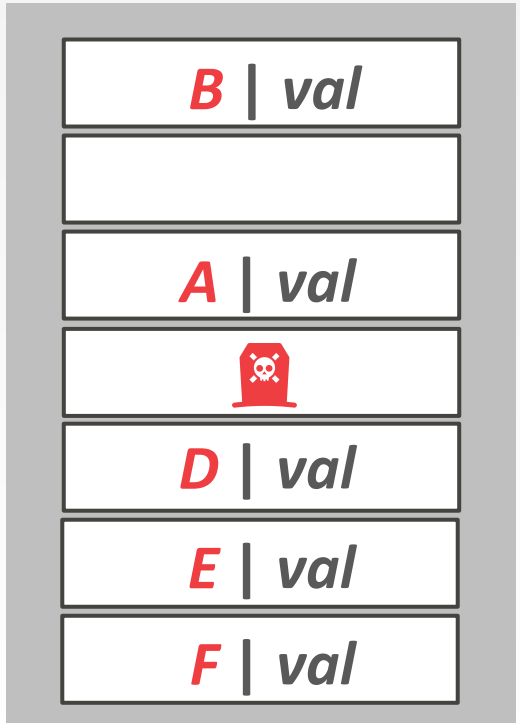


Approach #1: Tombstone

LINEAR PROBE HASHING – DELETES

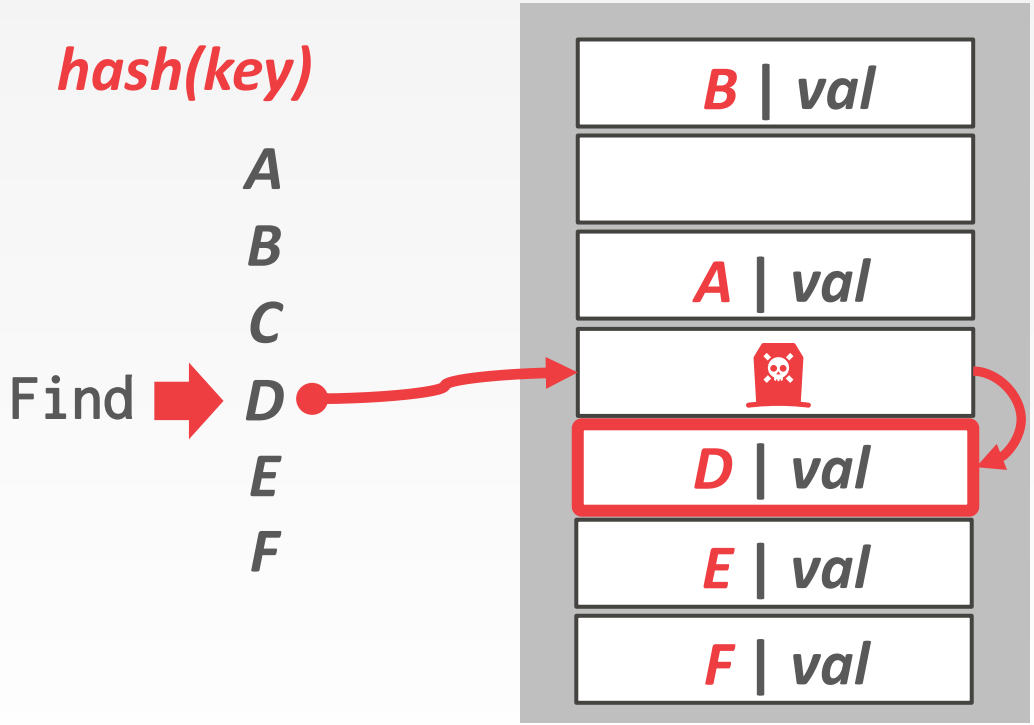
hash(key)

A
B
C
Find → D
E
F



Approach #1: Tombstone

LINEAR PROBE HASHING – DELETES

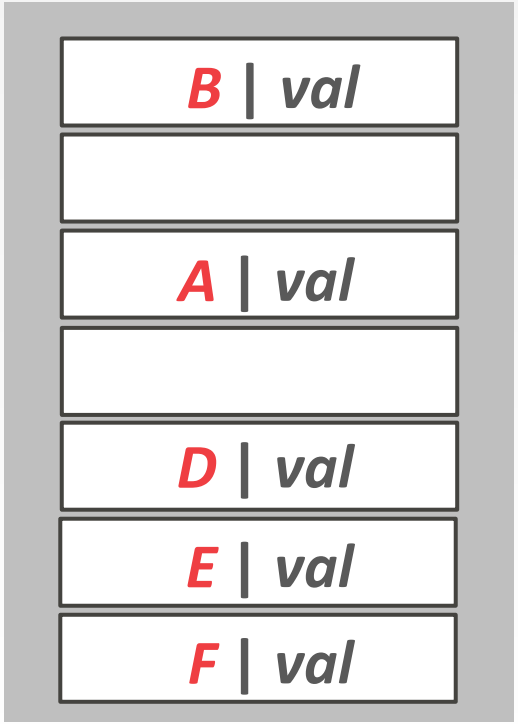


Approach #1: Tombstone

LINEAR PROBE HASHING – DELETES

hash(key)

A
B
C
Find → *D*
E
F



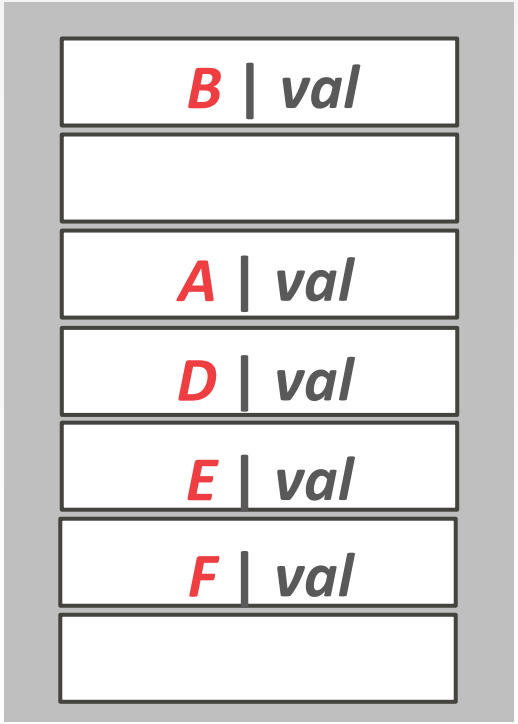
Approach #1: Tombstone

Approach #2: Movement

LINEAR PROBE HASHING – DELETES

hash(key)

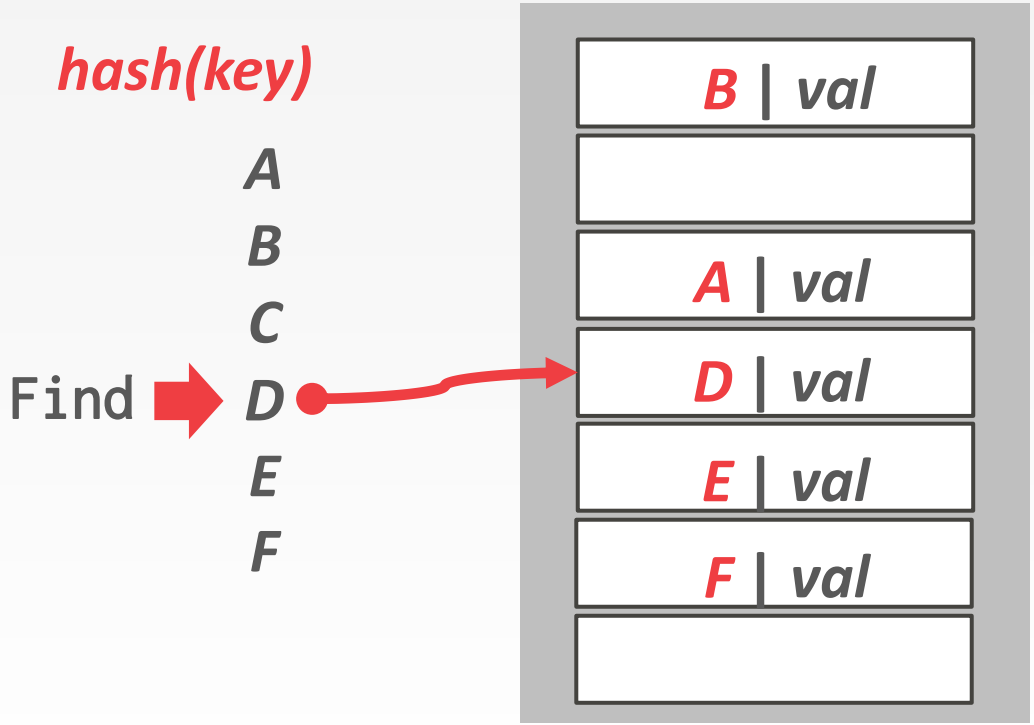
A
B
C
Find → *D*
E
F



Approach #1: Tombstone

Approach #2: Movement

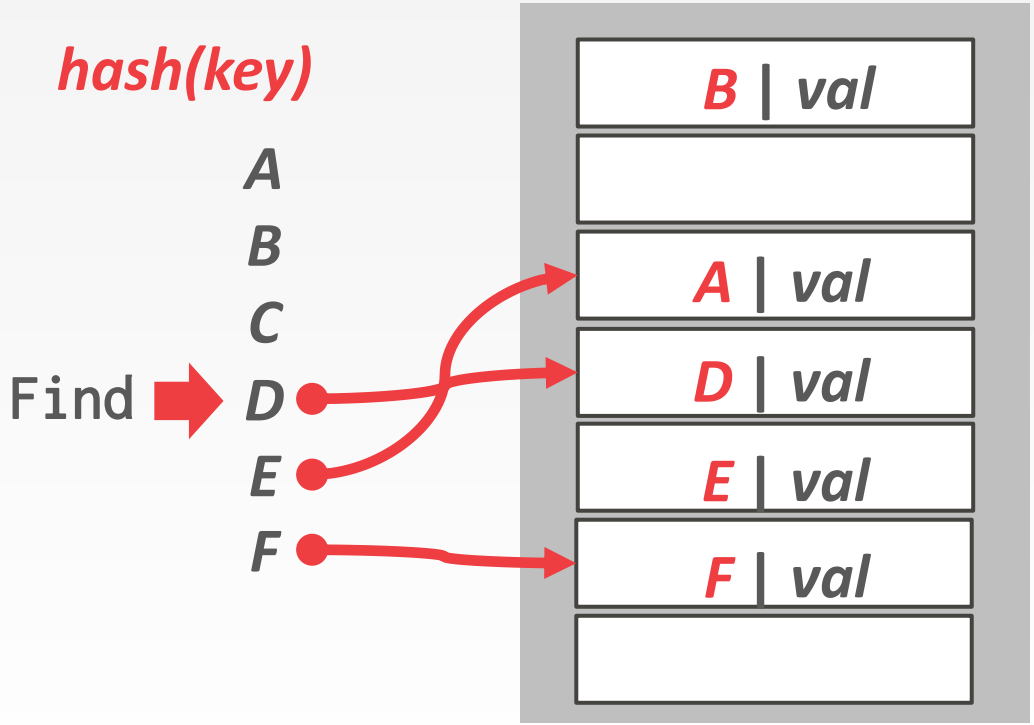
LINEAR PROBE HASHING – DELETES



Approach #1: Tombstone

Approach #2: Movement

LINEAR PROBE HASHING – DELETES



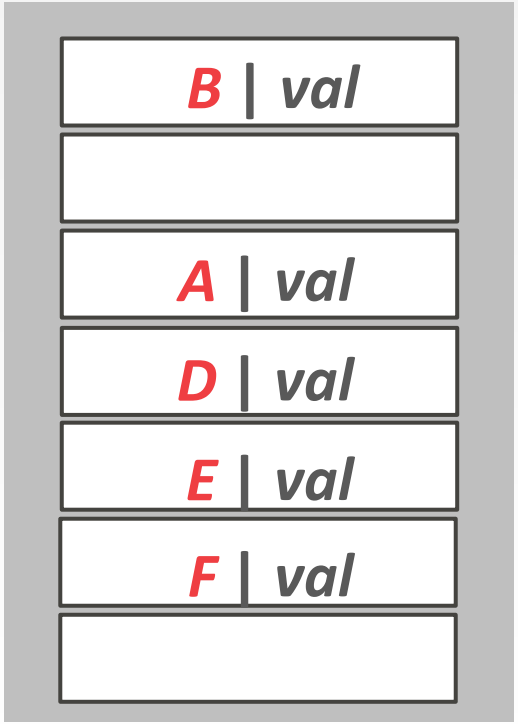
Approach #1: Tombstone

Approach #2: Movement

LINEAR PROBE HASHING – DELETES

hash(key)

A
B
C
Find → *D*
E
F



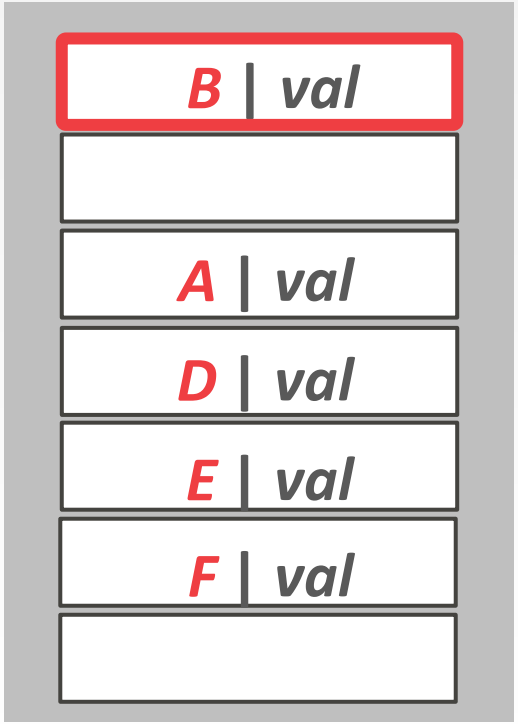
Approach #1: Tombstone

Approach #2: Movement

LINEAR PROBE HASHING – DELETES

hash(key)

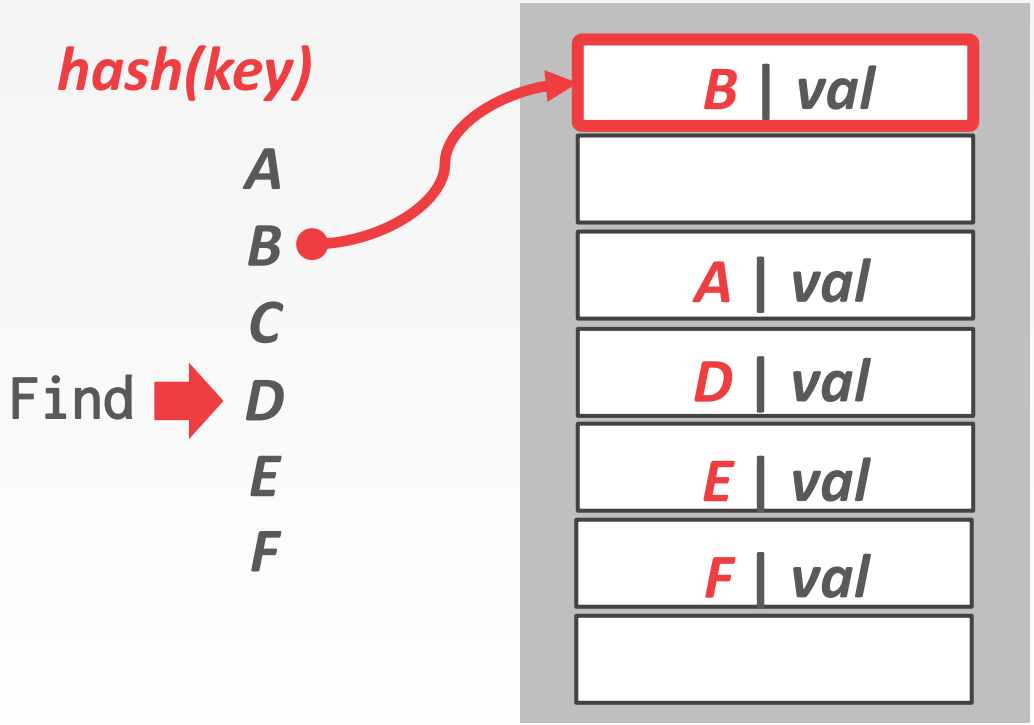
A
B
C
Find → *D*
E
F



Approach #1: Tombstone

Approach #2: Movement

LINEAR PROBE HASHING – DELETES



Approach #1: Tombstone

Approach #2: Movement

NON-UNIQUE KEYS

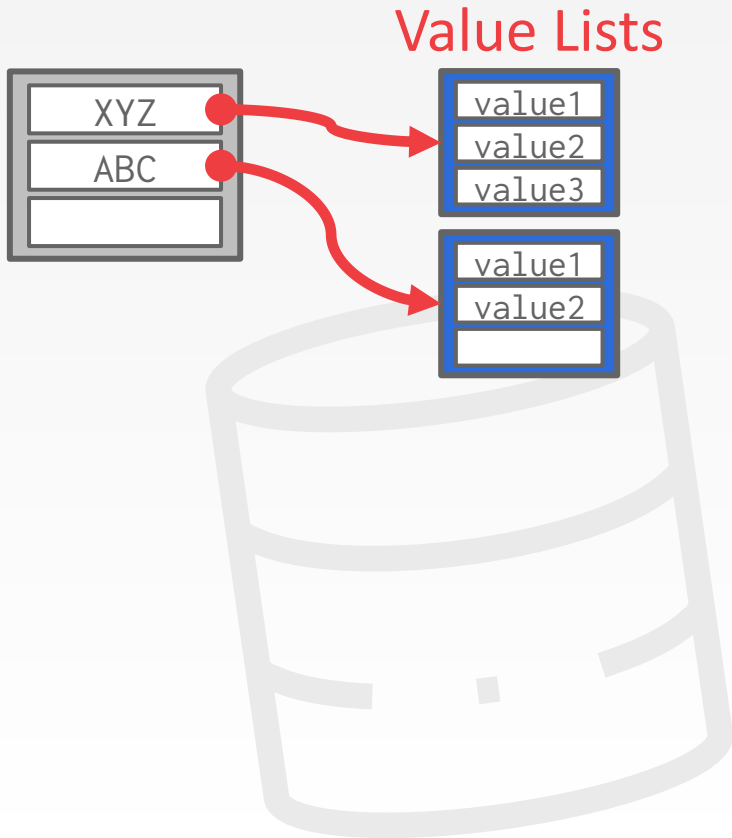
Choice #1: Separate Linked List

→ Store values in separate storage area for each key.

NON-UNIQUE KEYS

Choice #1: Separate Linked List

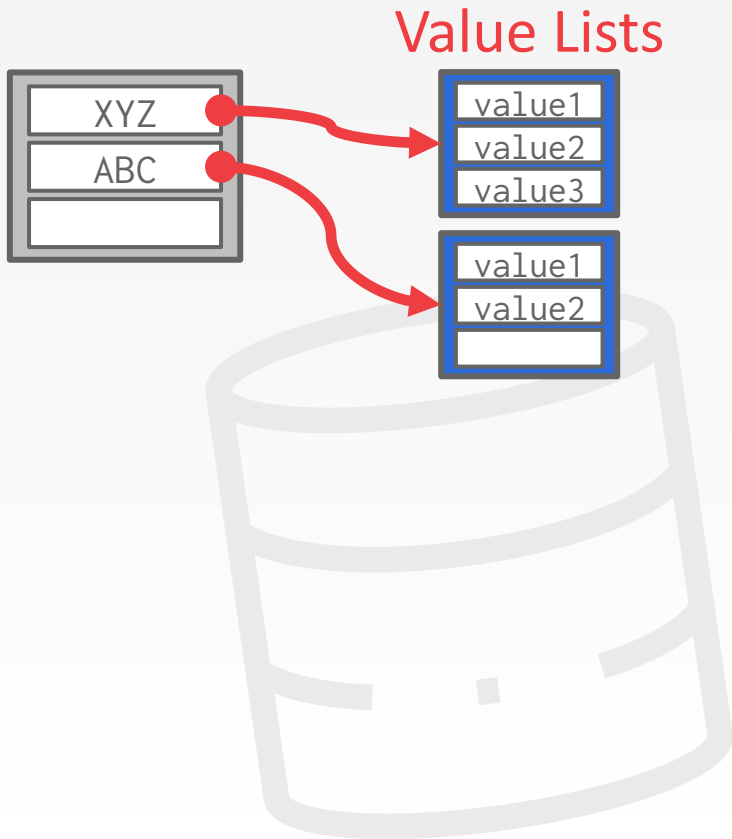
→ Store values in separate storage area for each key.



NON-UNIQUE KEYS

Choice #1: Separate Linked List

→ Store values in separate storage area for each key.



Choice #2: Redundant Keys

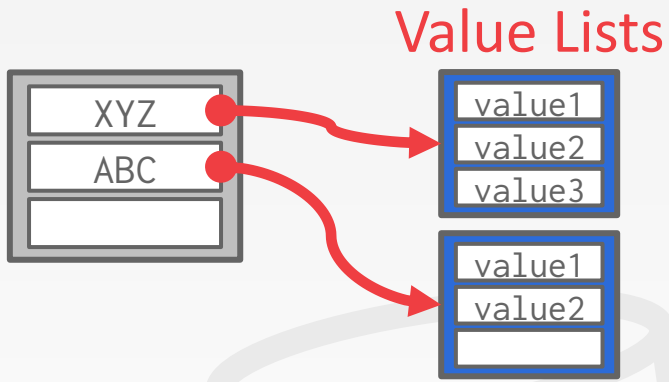
→ Store duplicate keys entries together in the hash table.

→ This is easier to implement so this is what most systems do.

NON-UNIQUE KEYS

Choice #1: Separate Linked List

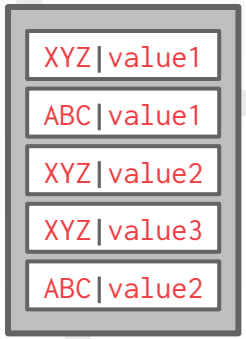
→ Store values in separate storage area for each key.



Choice #2: Redundant Keys

→ Store duplicate keys entries together in the hash table.

→ This is easier to implement so this is what most systems do.



ROBIN HOOD HASHING

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

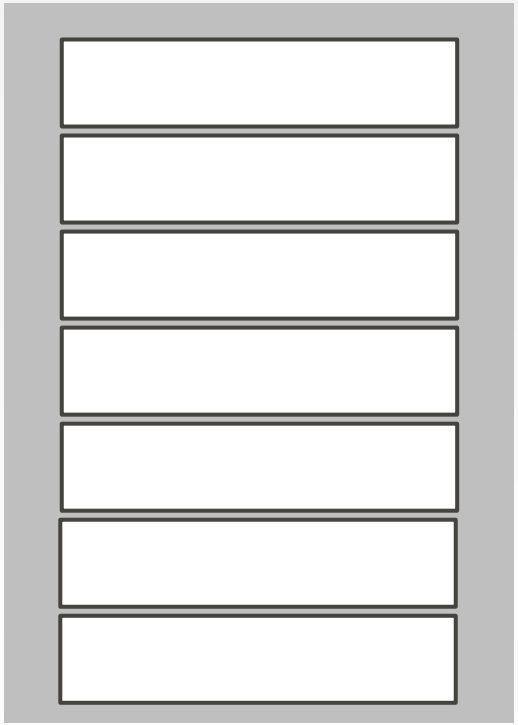
→ Each key tracks the number of positions they are from where its optimal position in the table.

→ 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.

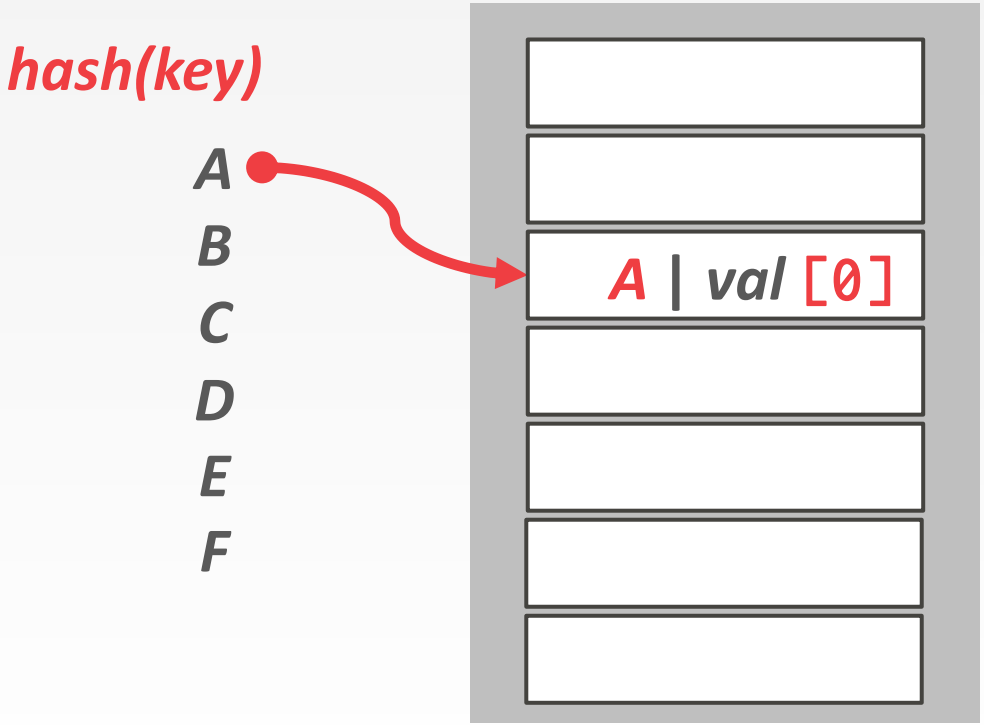
ROBIN HOOD HASHING

hash(key)

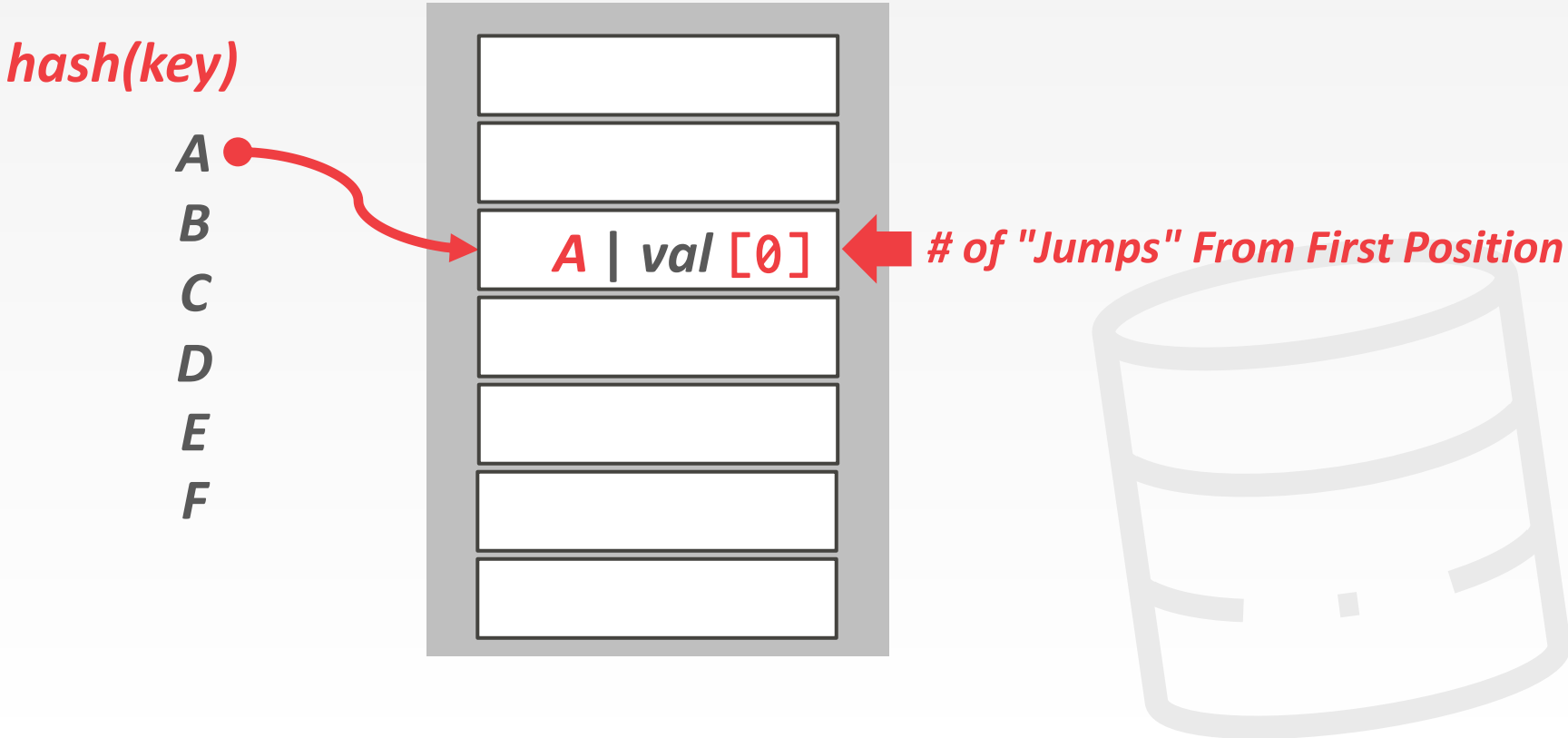
A
B
C
D
E
F



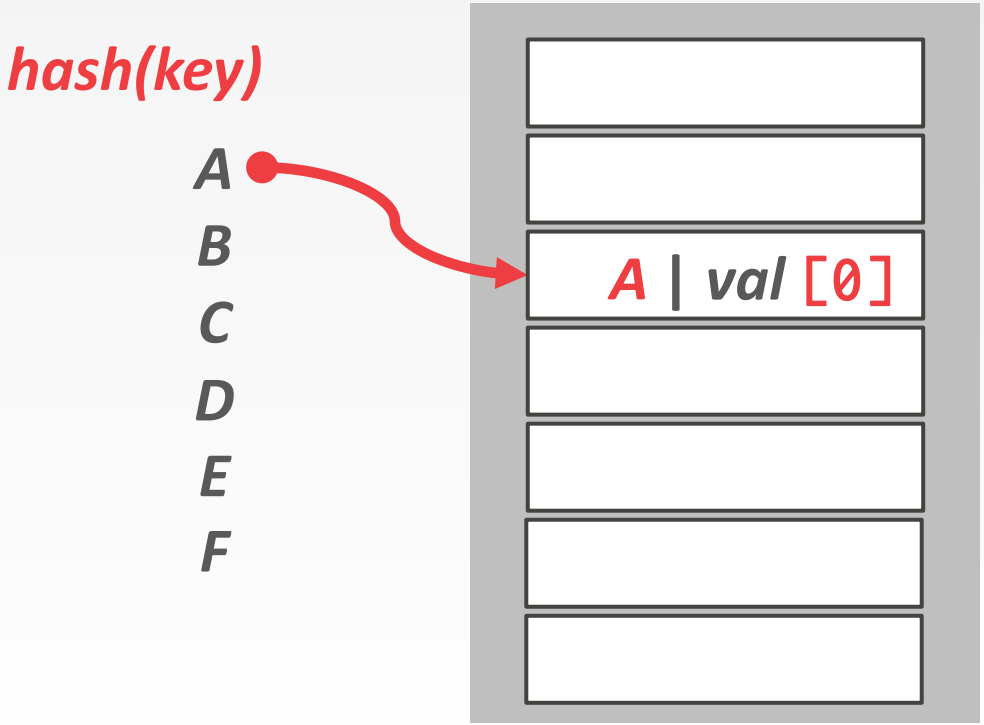
ROBIN HOOD HASHING



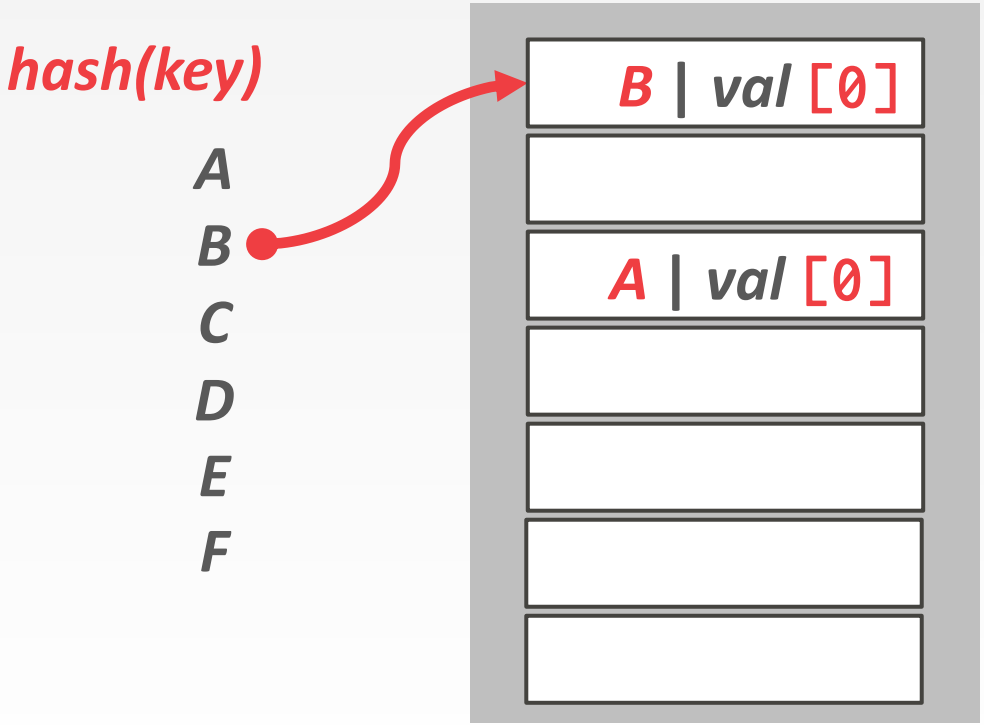
ROBIN HOOD HASHING



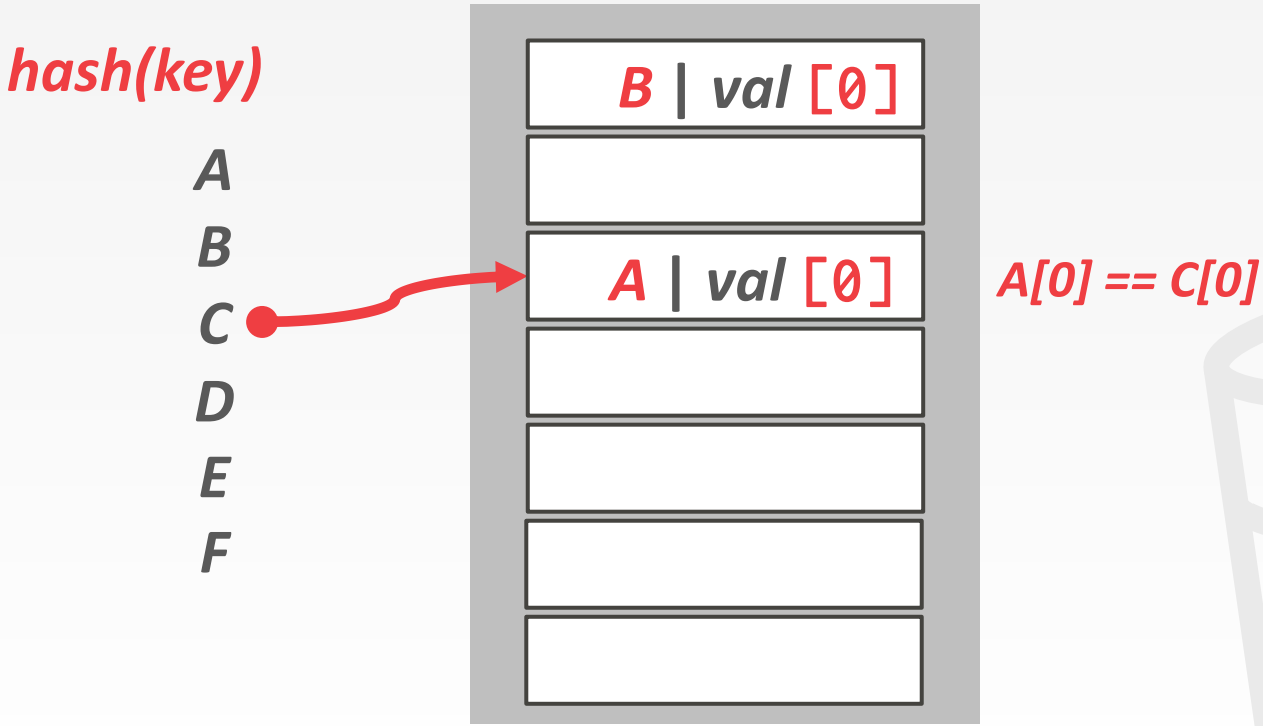
ROBIN HOOD HASHING



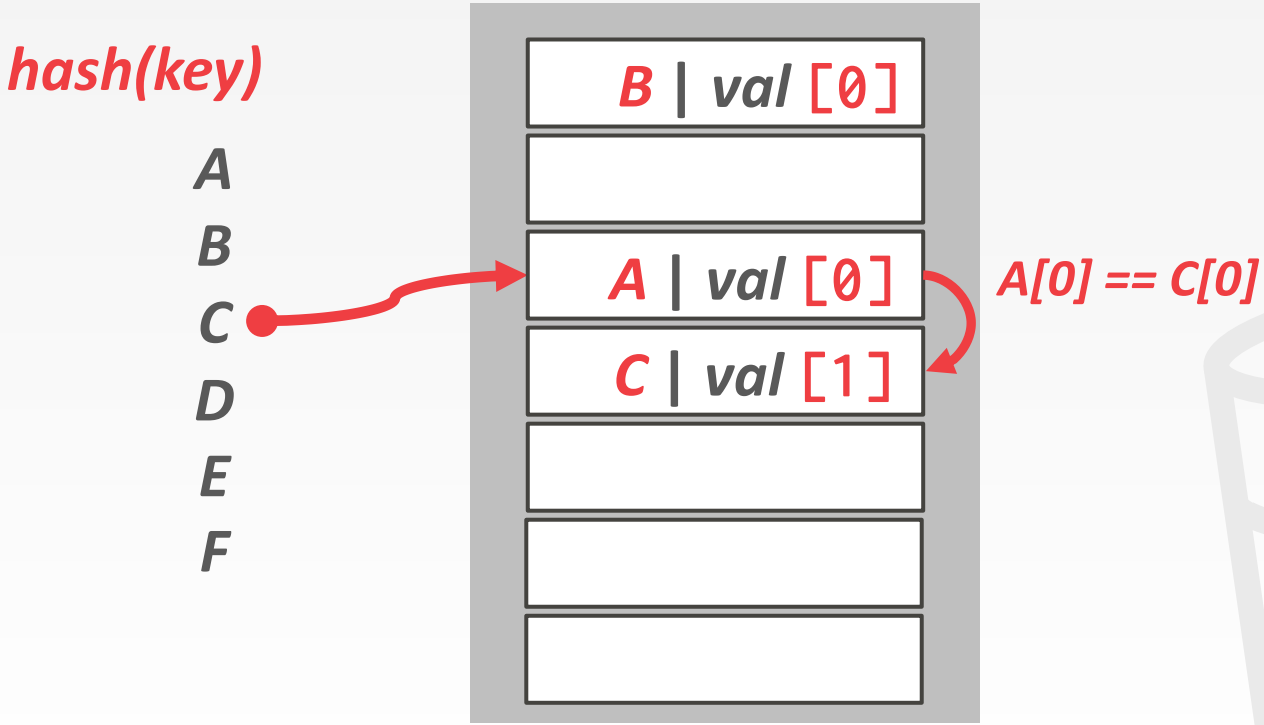
ROBIN HOOD HASHING



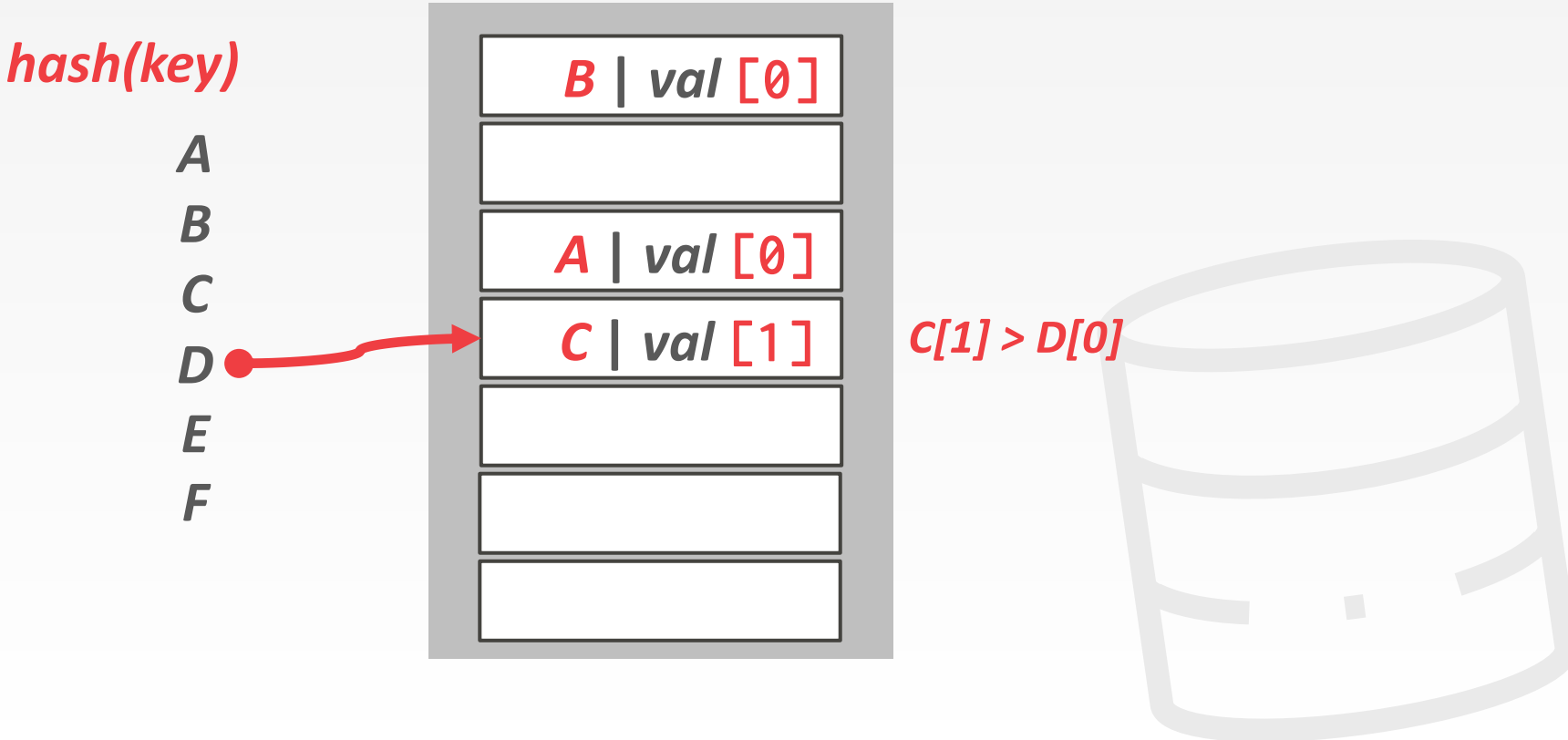
ROBIN HOOD HASHING



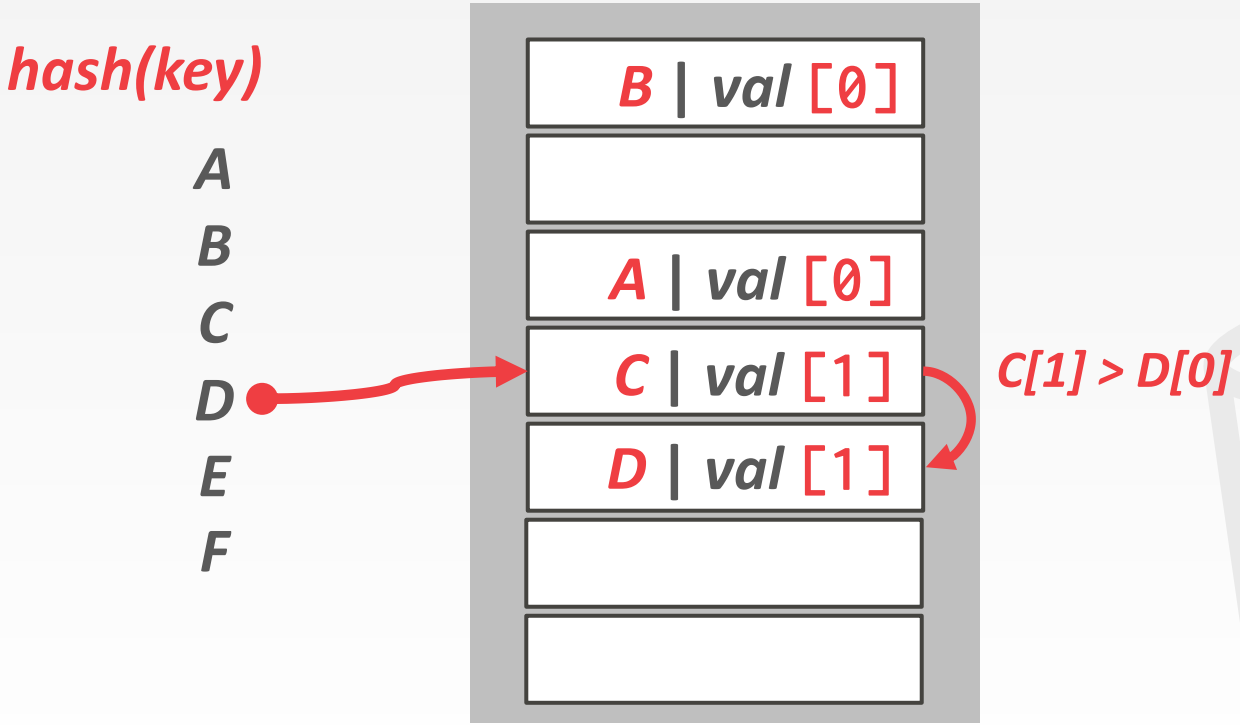
ROBIN HOOD HASHING



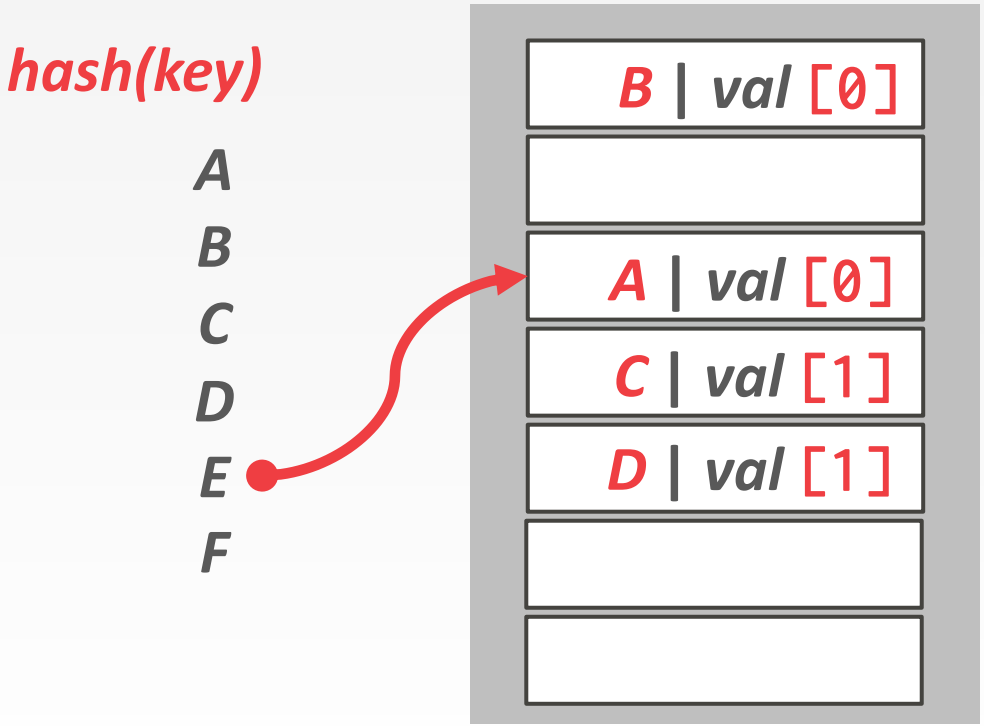
ROBIN HOOD HASHING



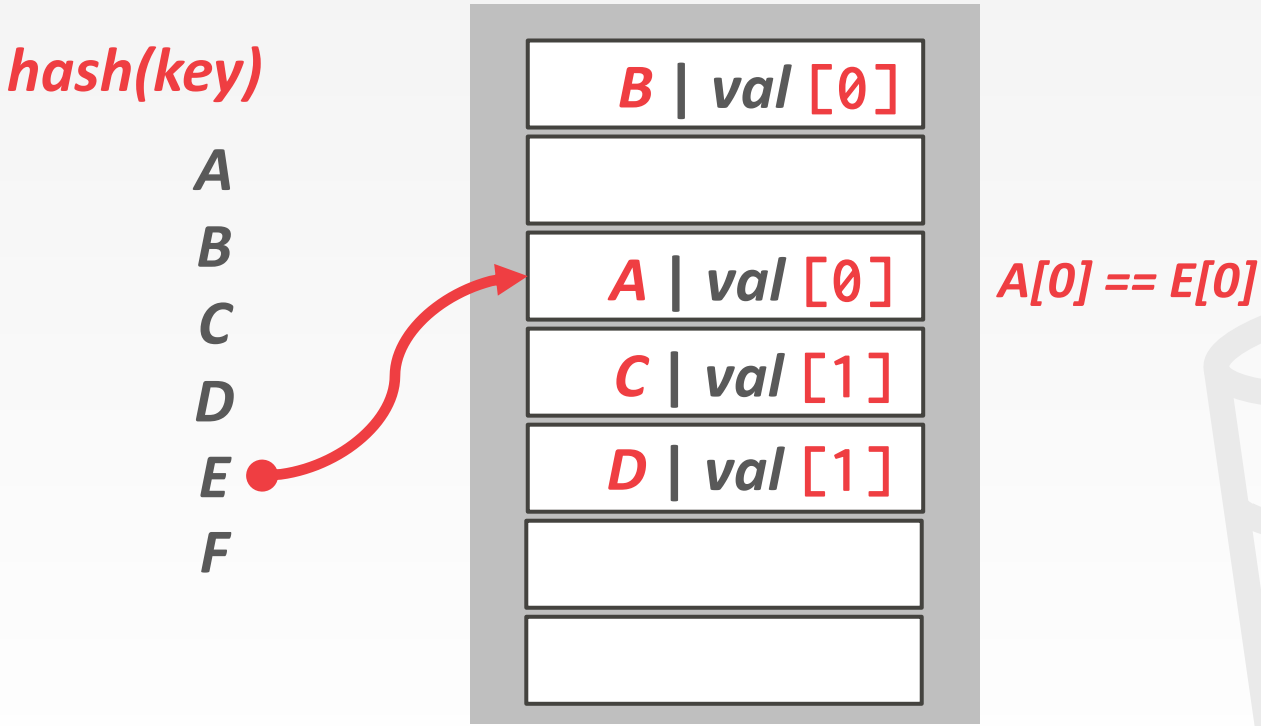
ROBIN HOOD HASHING



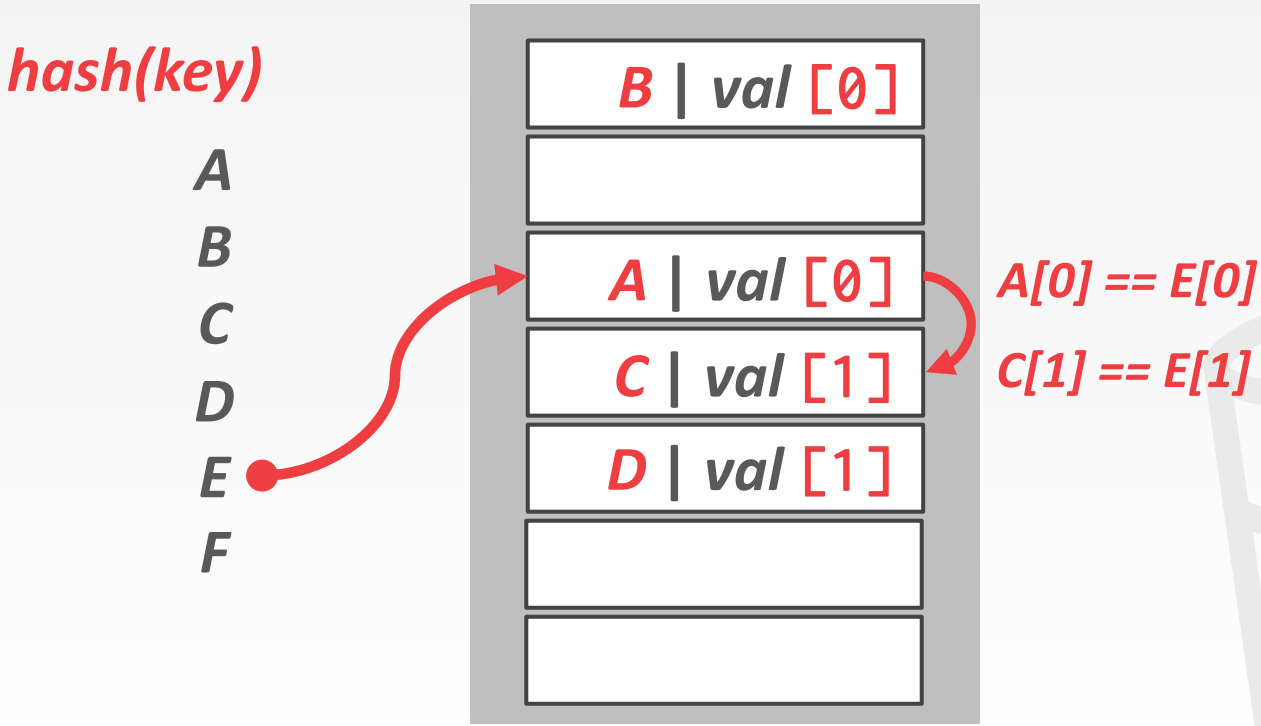
ROBIN HOOD HASHING



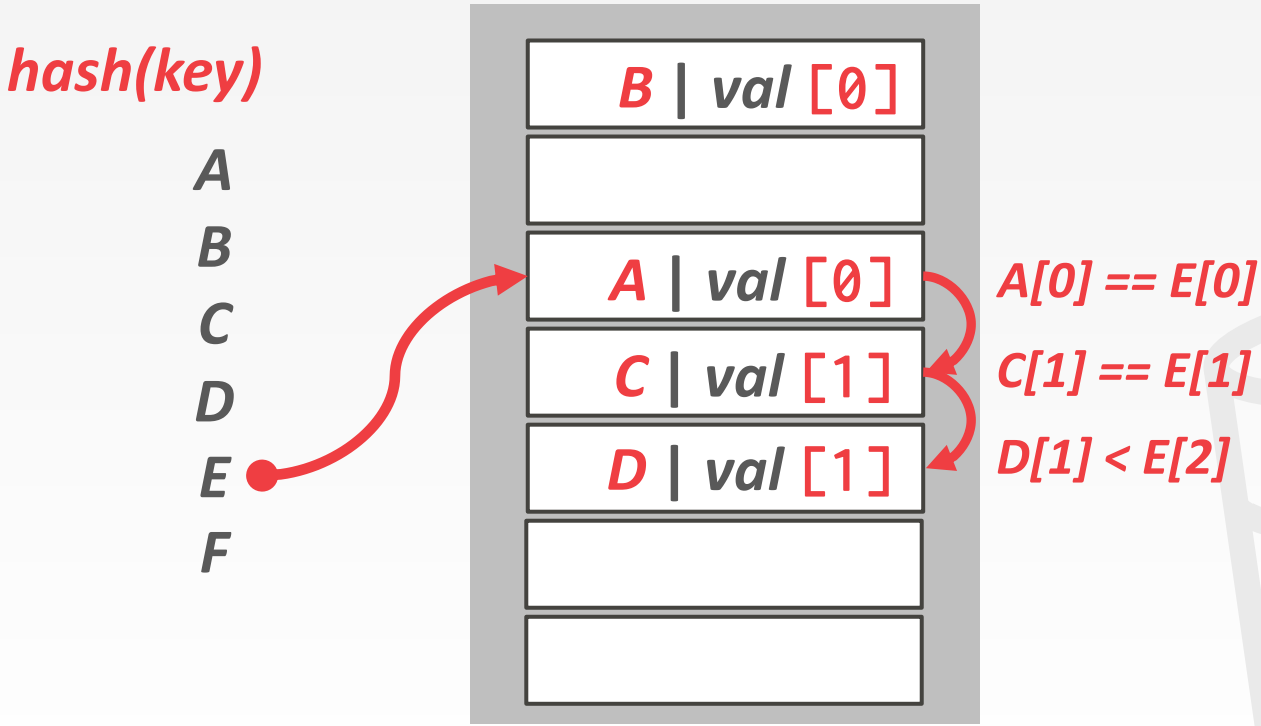
ROBIN HOOD HASHING



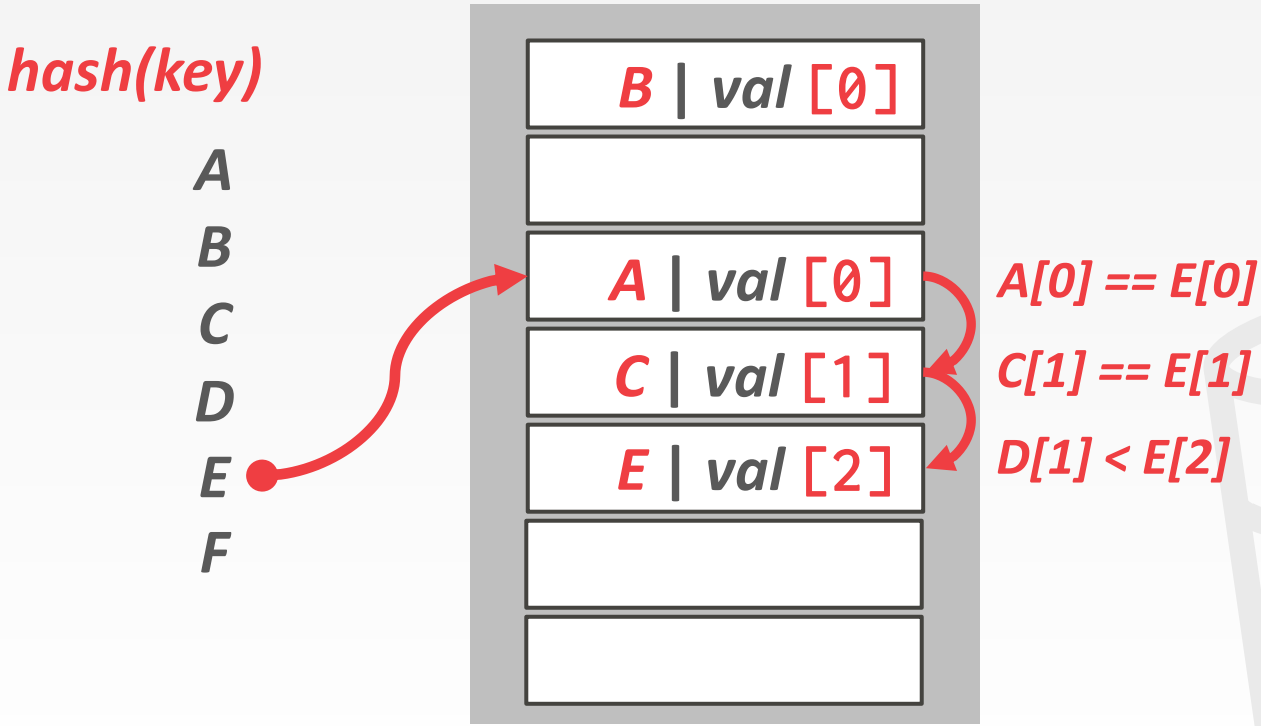
ROBIN HOOD HASHING



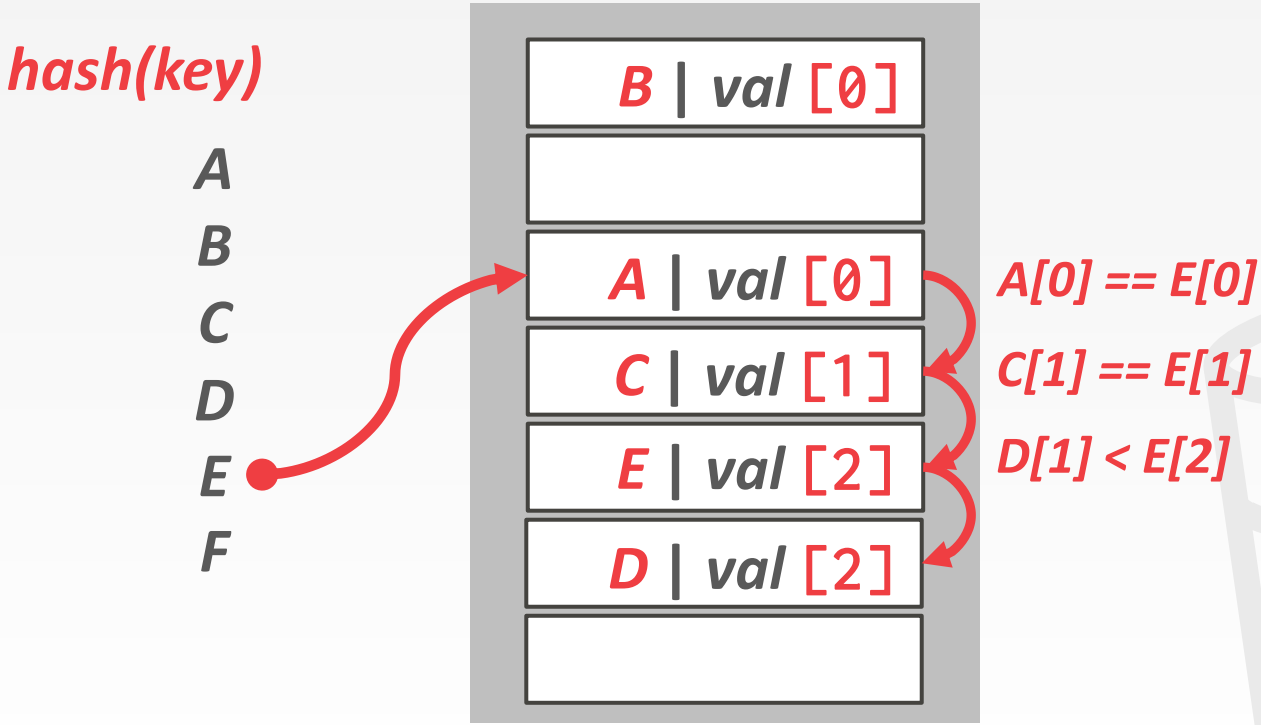
ROBIN HOOD HASHING



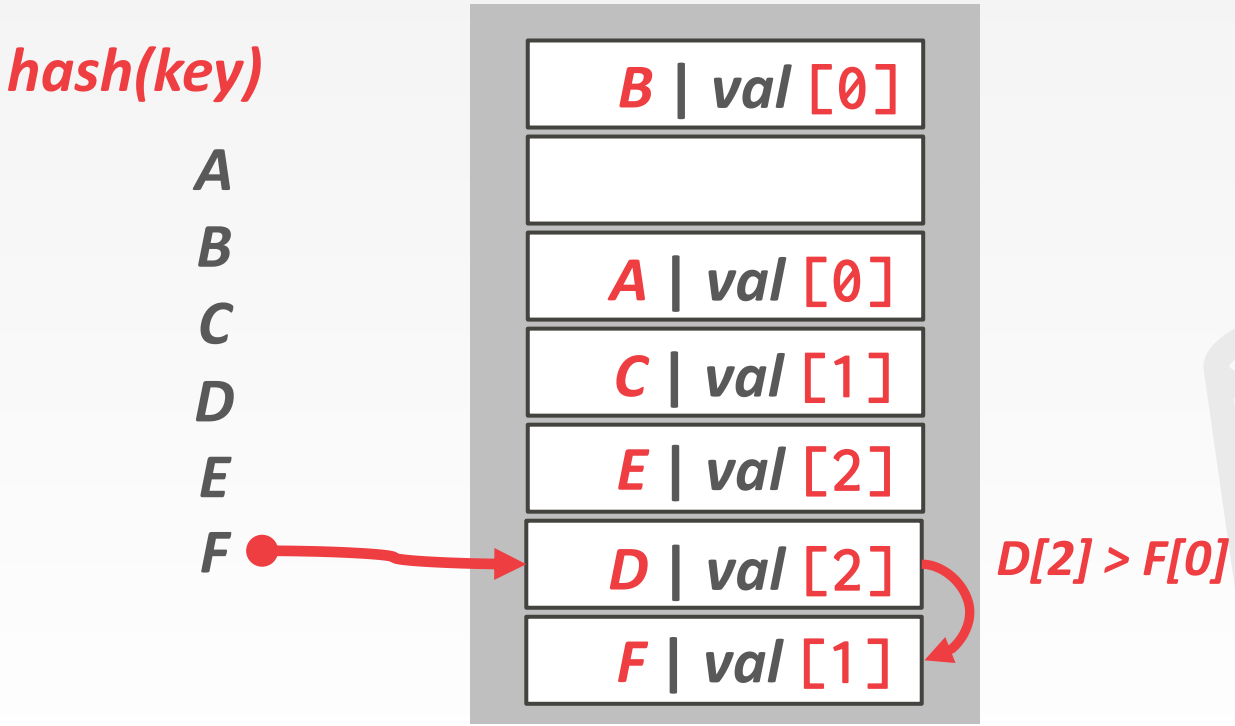
ROBIN HOOD HASHING



ROBIN HOOD HASHING



ROBIN HOOD HASHING



CUCKOO HASHING

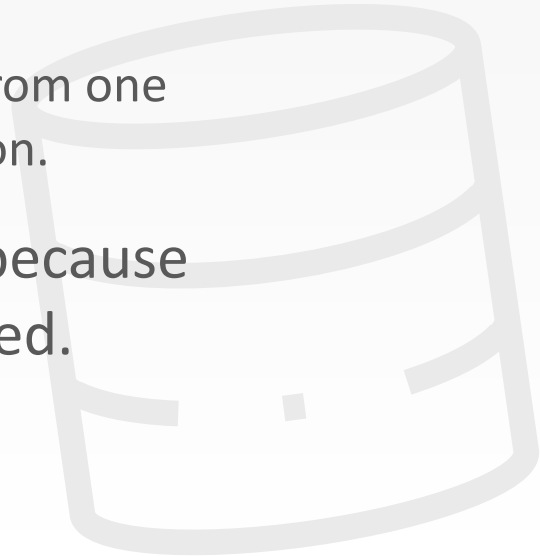
Use multiple hash tables with different hash function seeds.

→ 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.

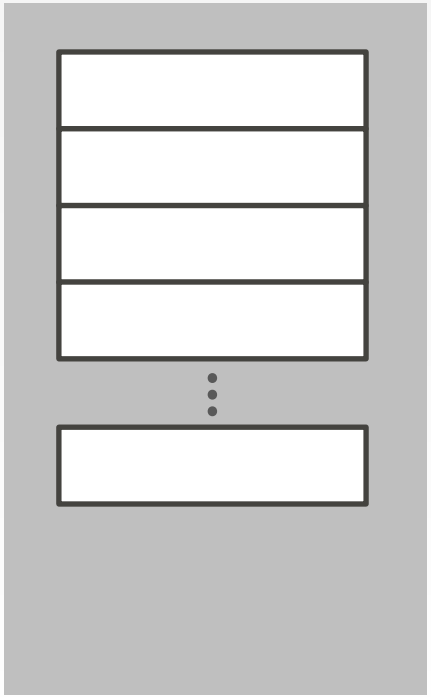
Look-ups and deletions are always **O(1)** because only one location per hash table is checked.

Open-source implementation

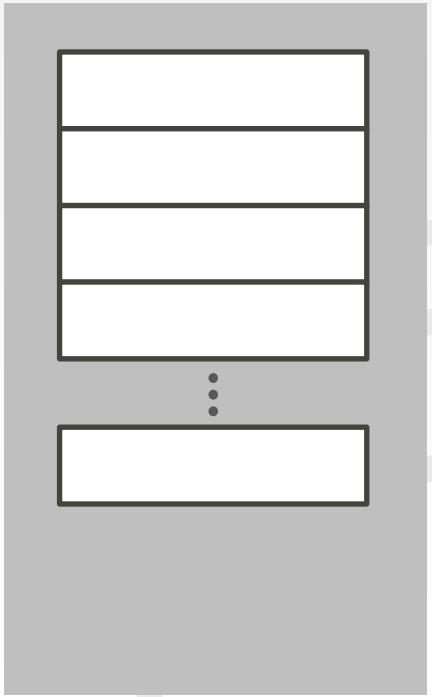


CUCKOO HASHING

Hash Table #1



Hash Table #2

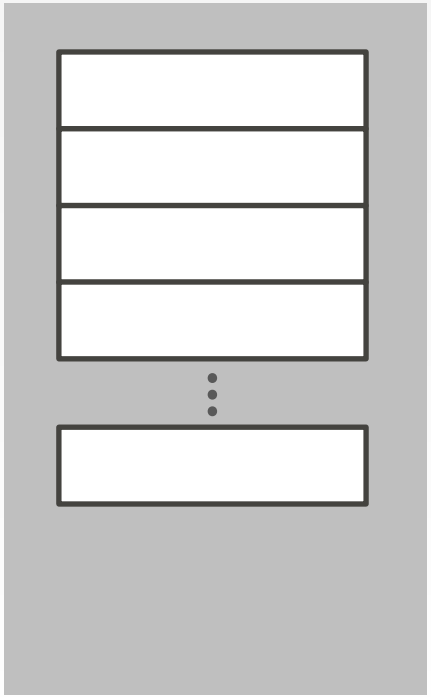


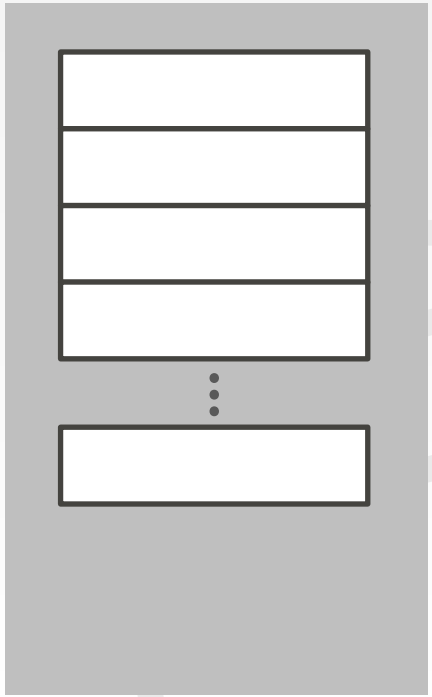
CUCKOO HASHING

Hash Table #1

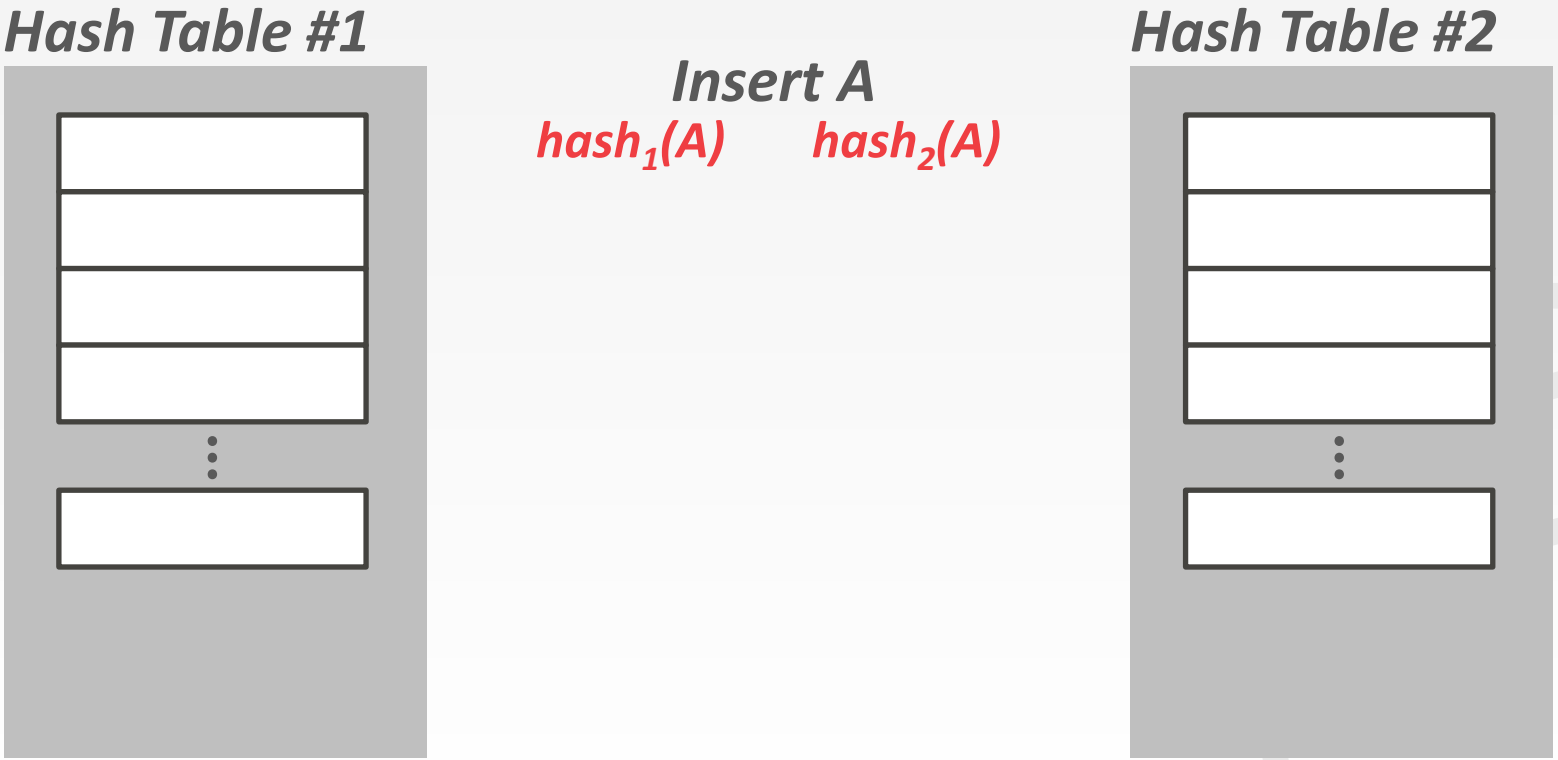
Insert A

Hash Table #2

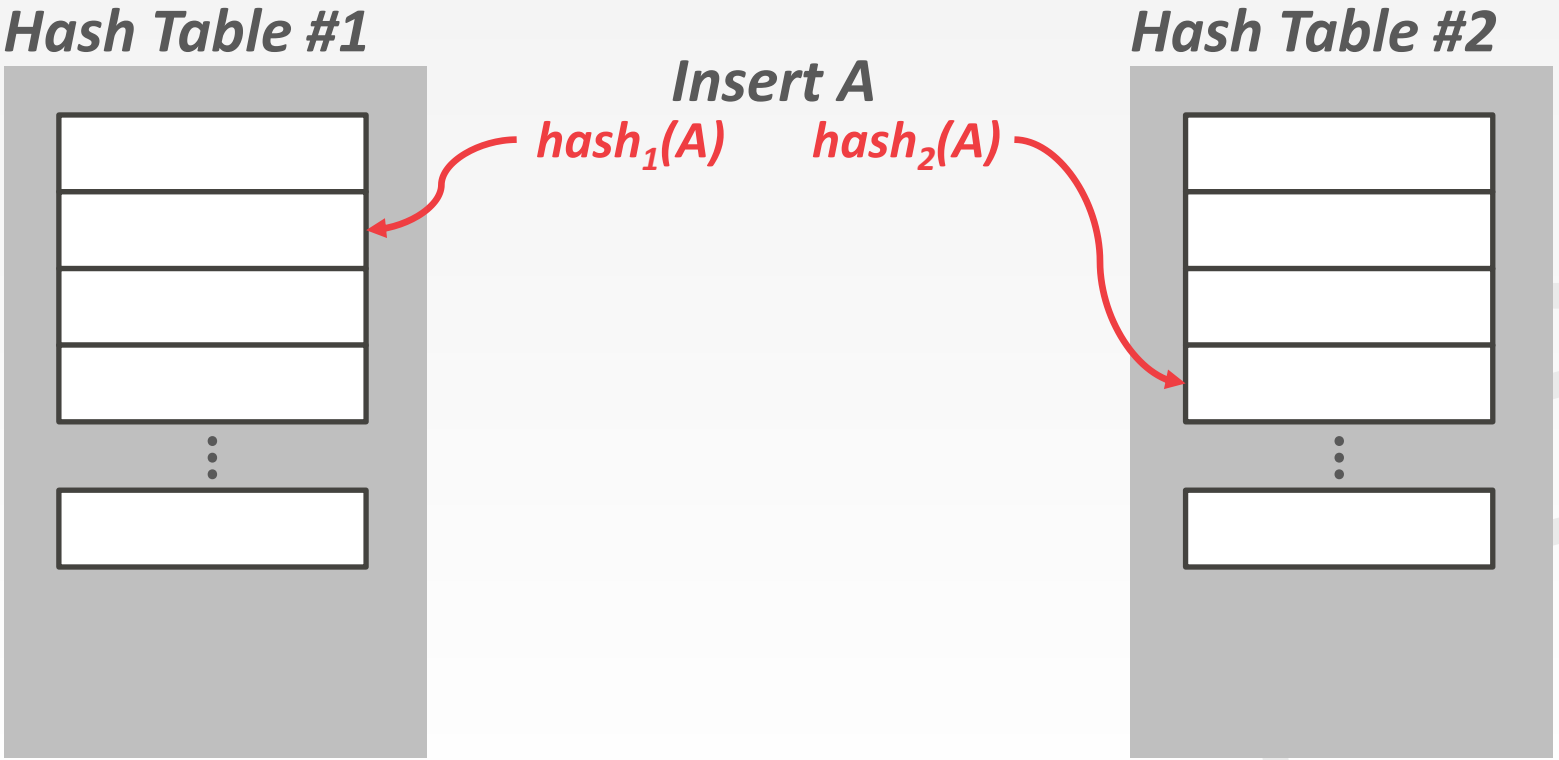




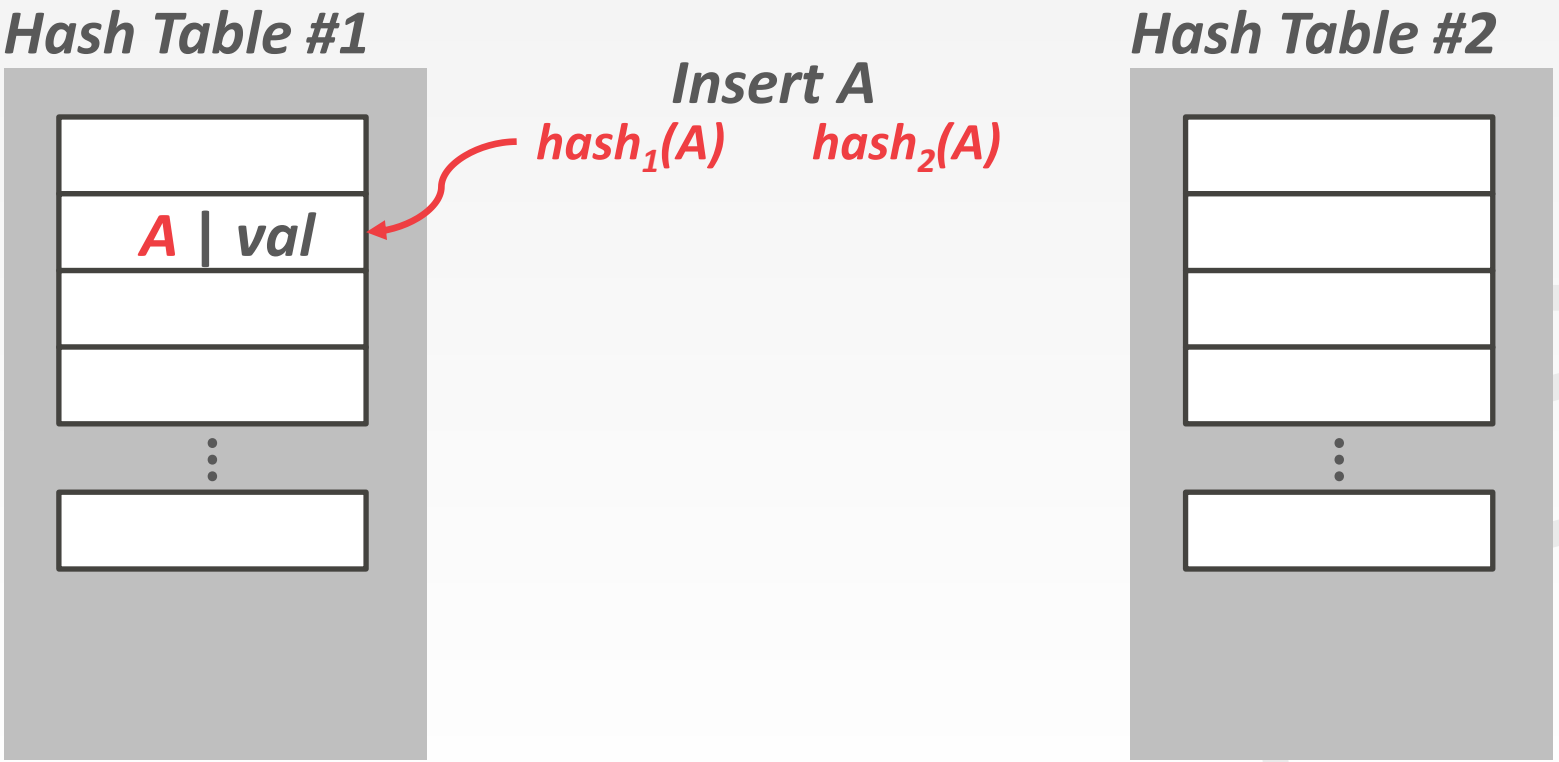
CUCKOO HASHING



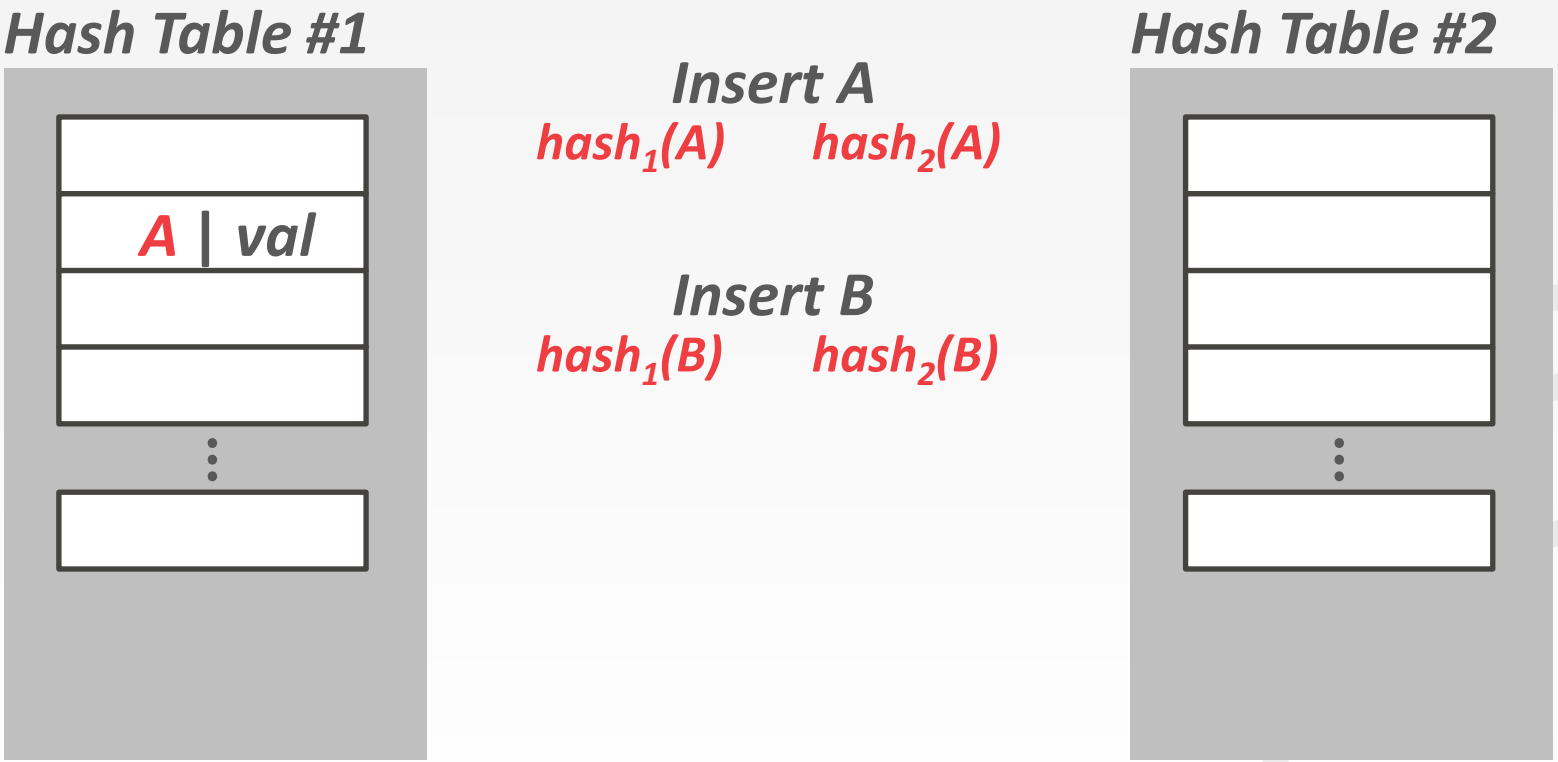
CUCKOO HASHING



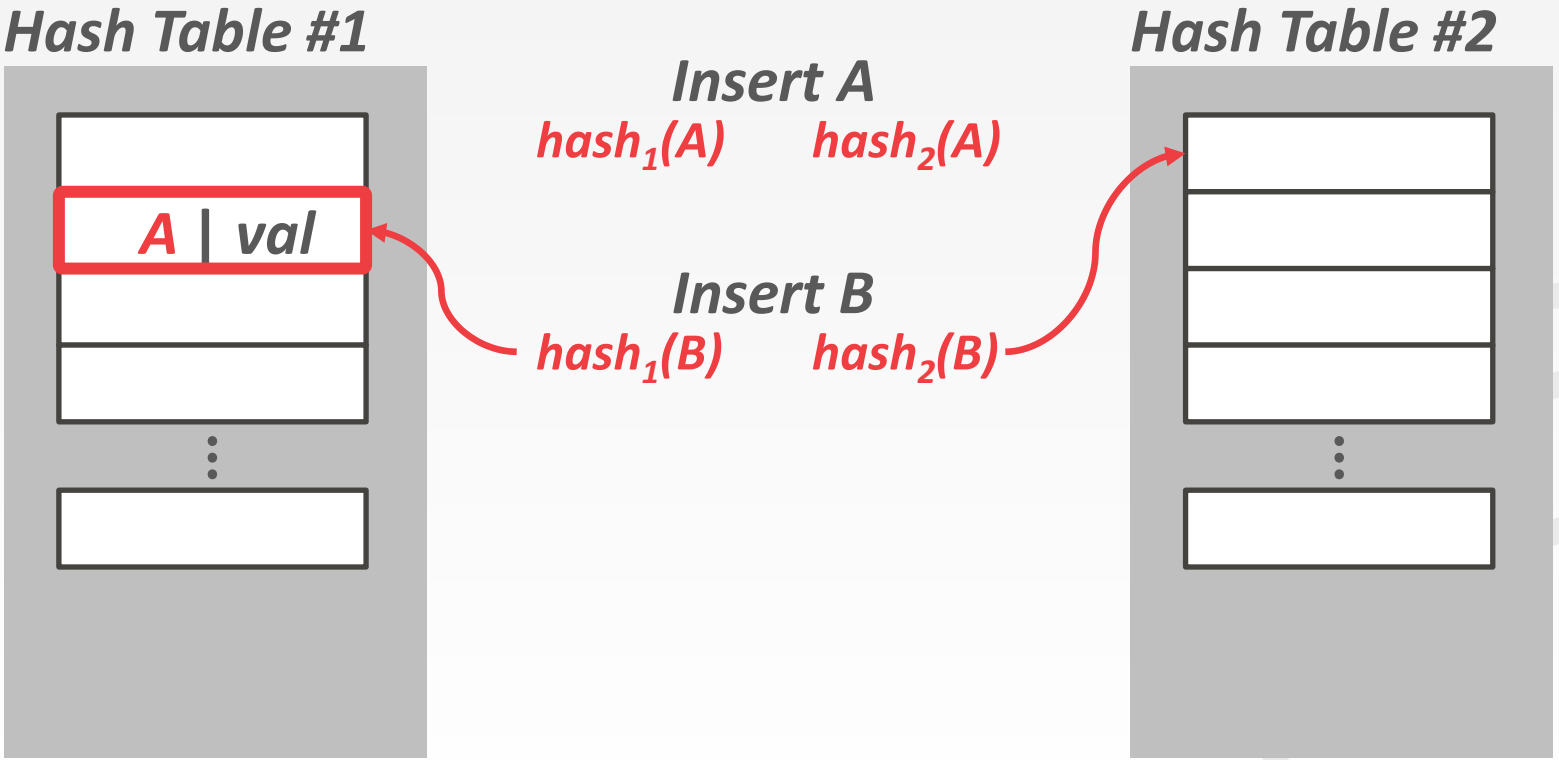
CUCKOO HASHING



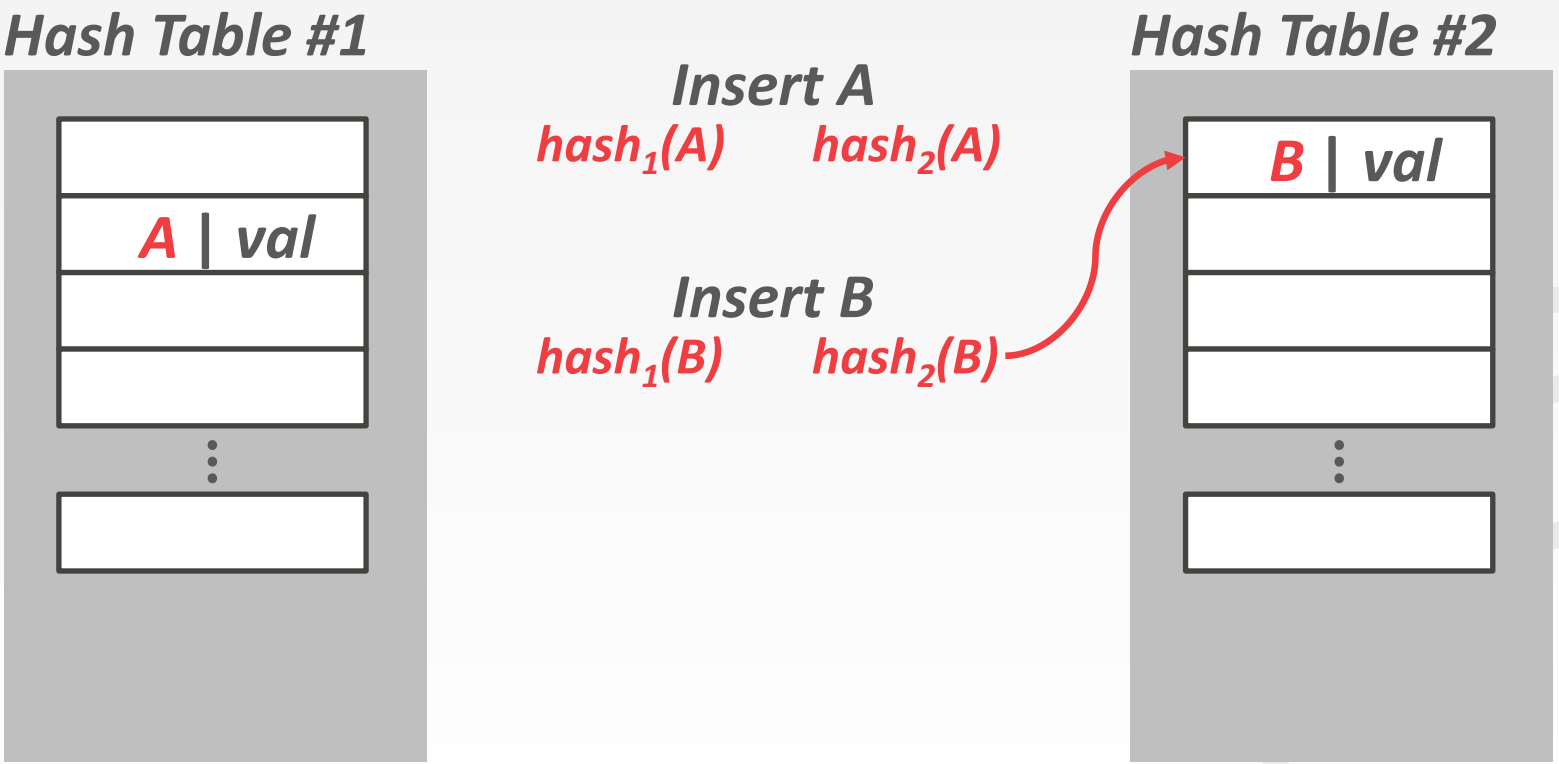
CUCKOO HASHING



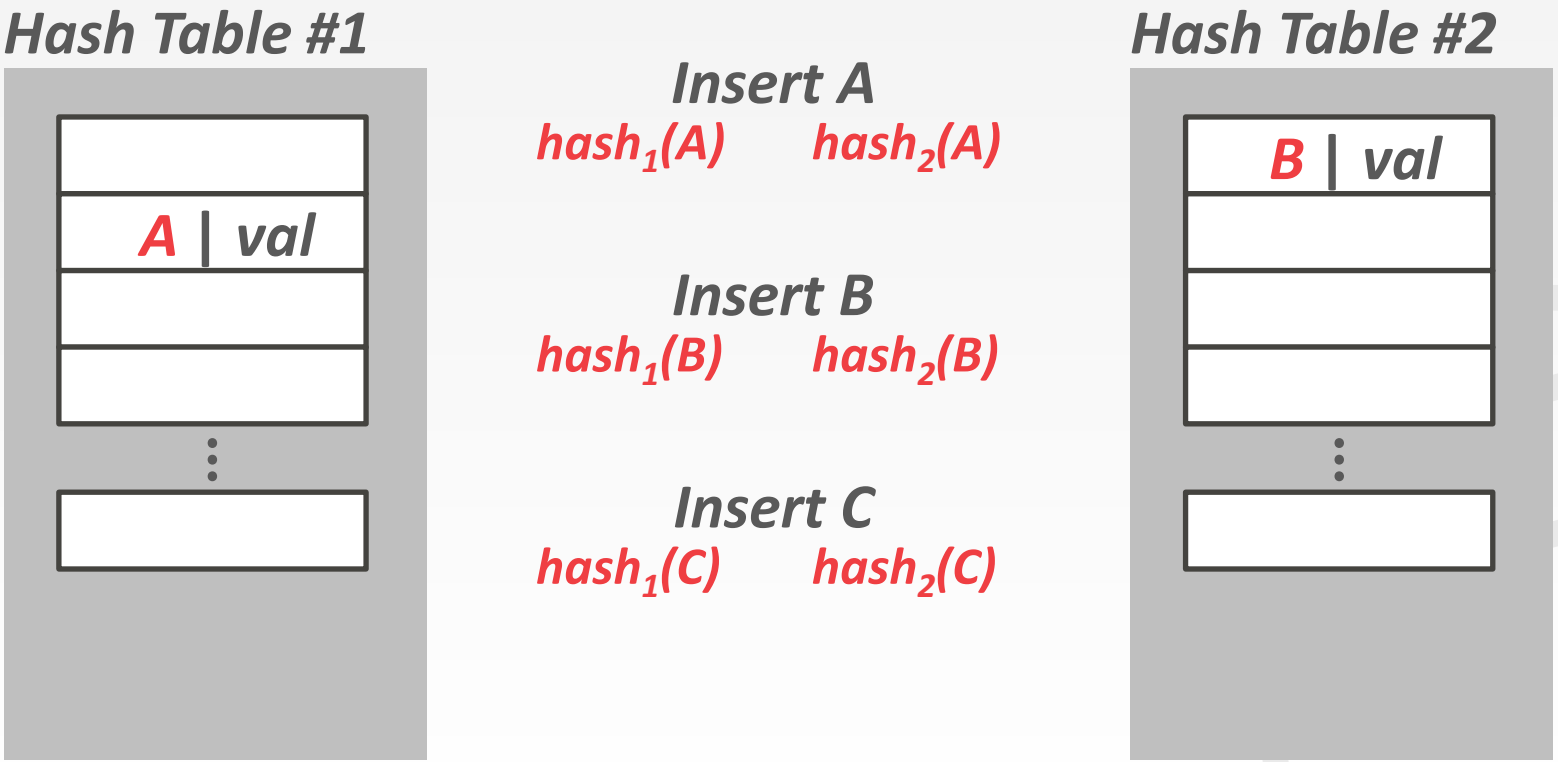
CUCKOO HASHING



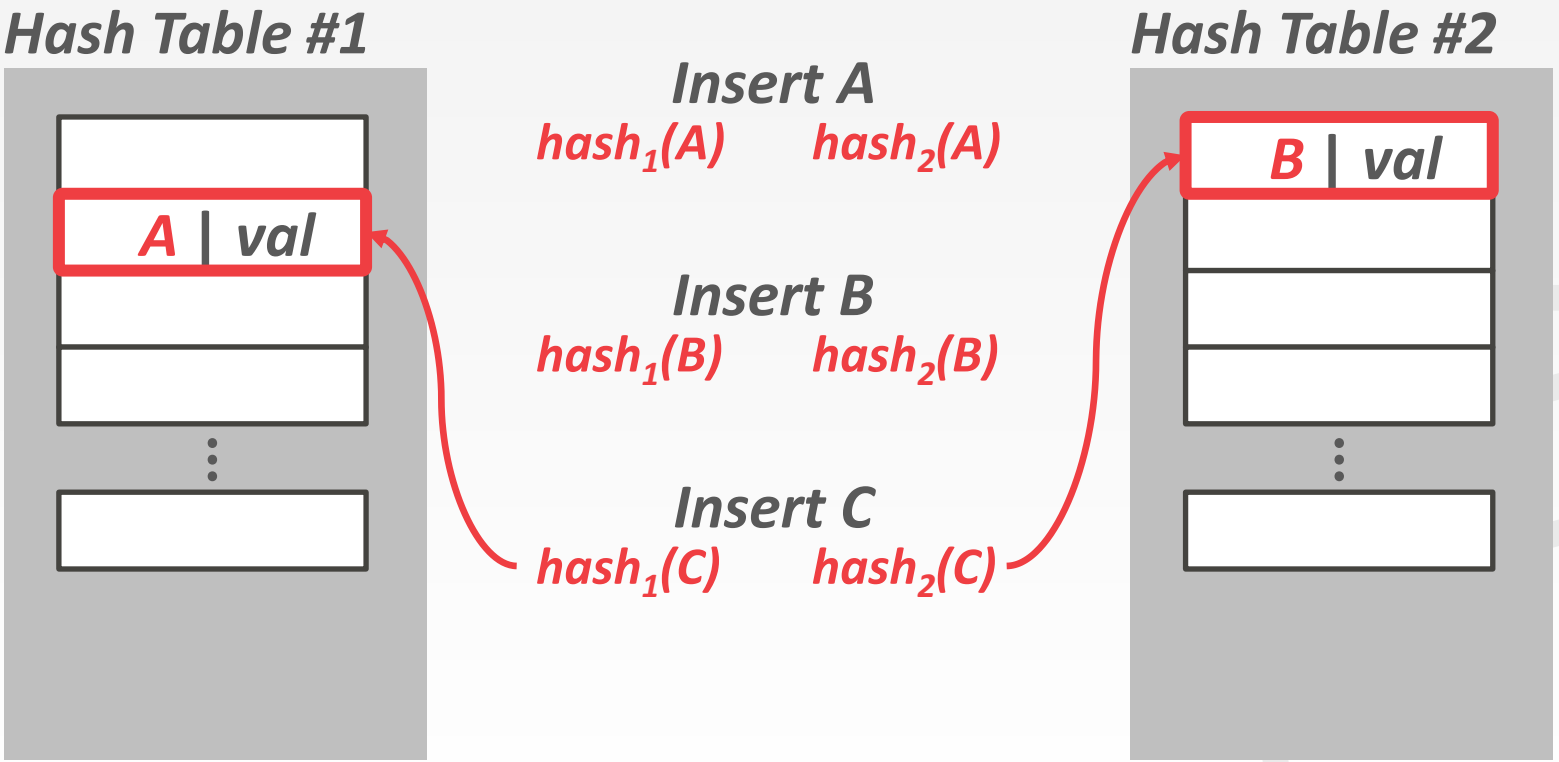
CUCKOO HASHING



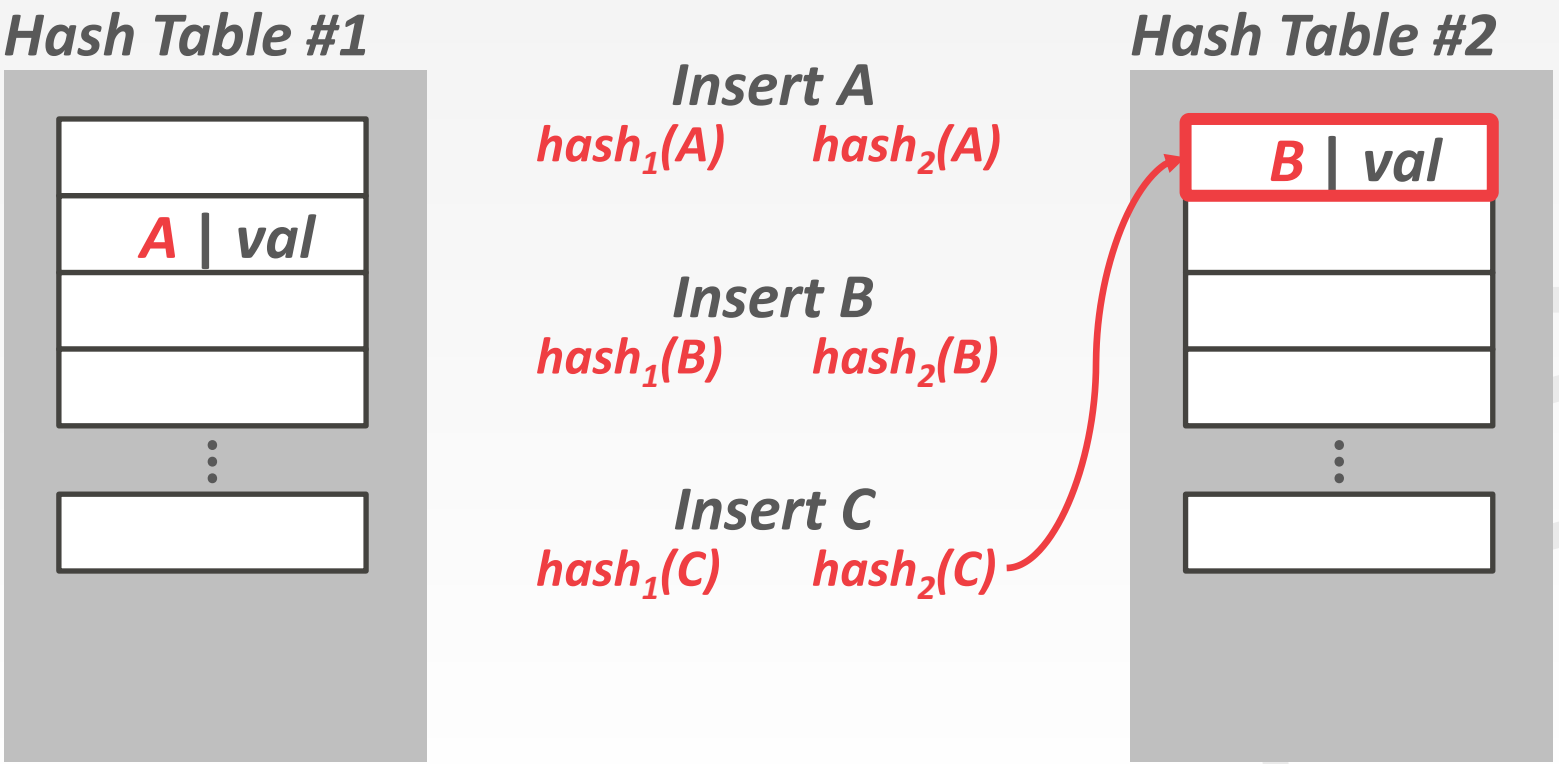
CUCKOO HASHING



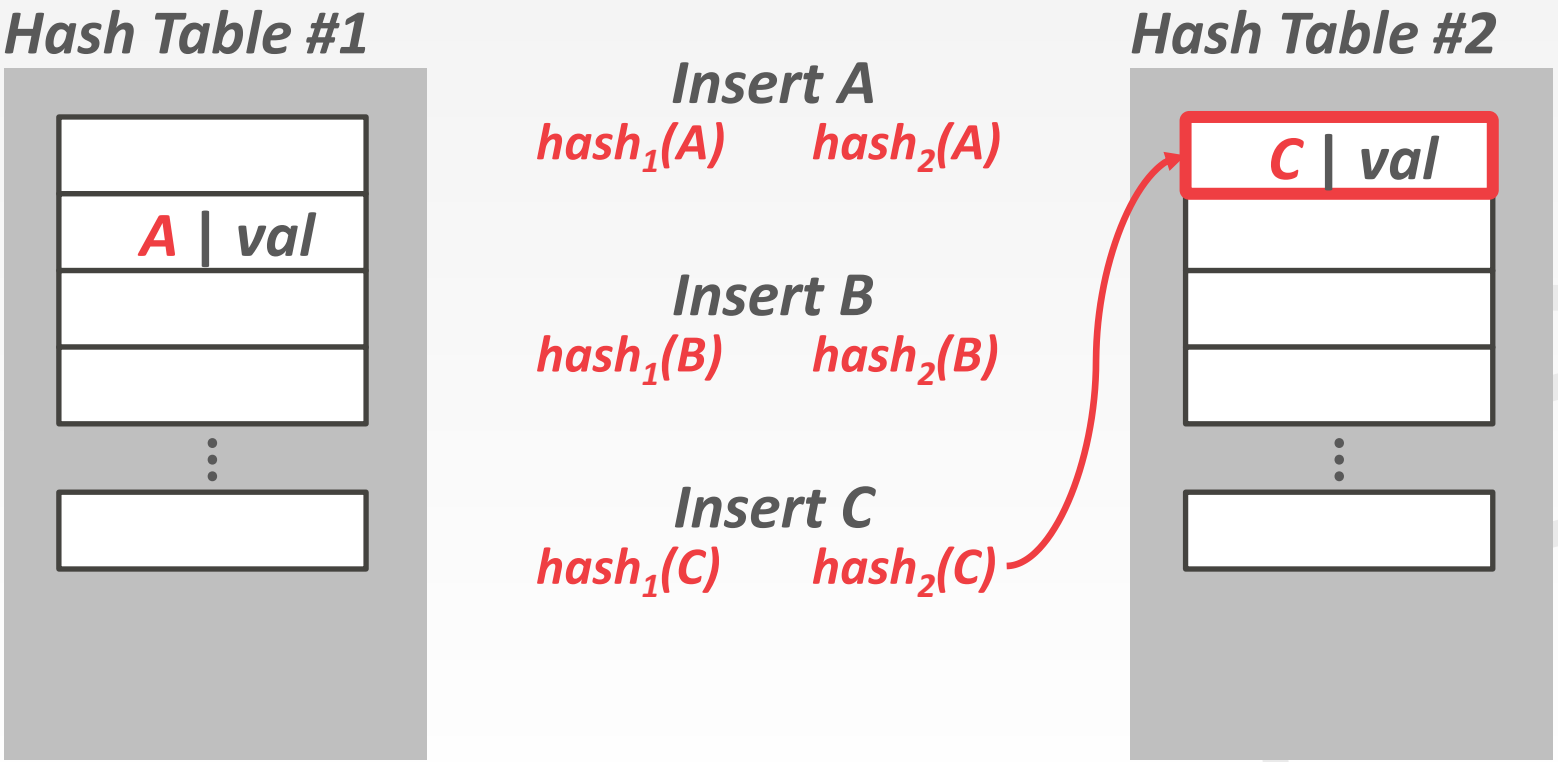
CUCKOO HASHING



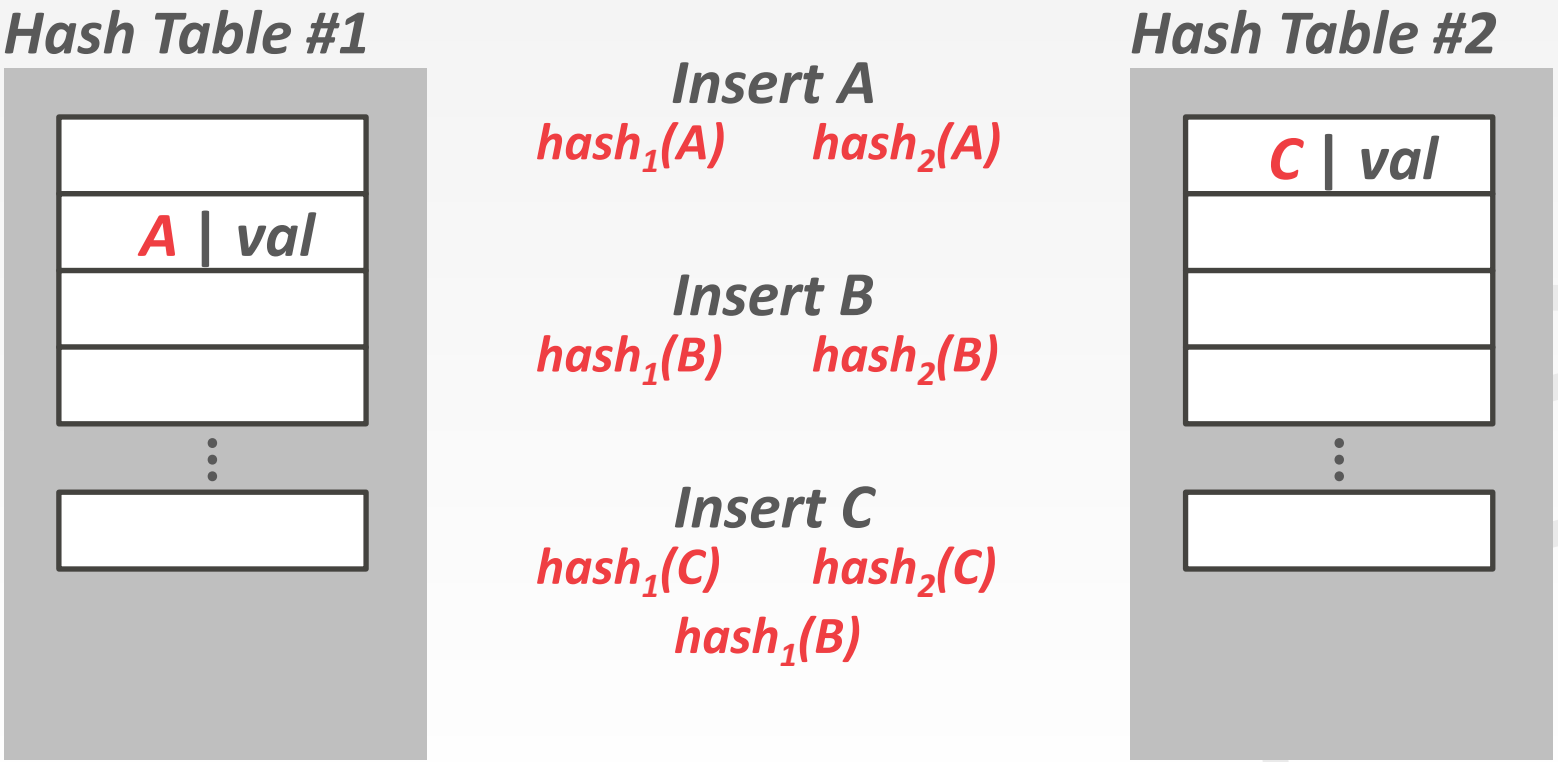
CUCKOO HASHING



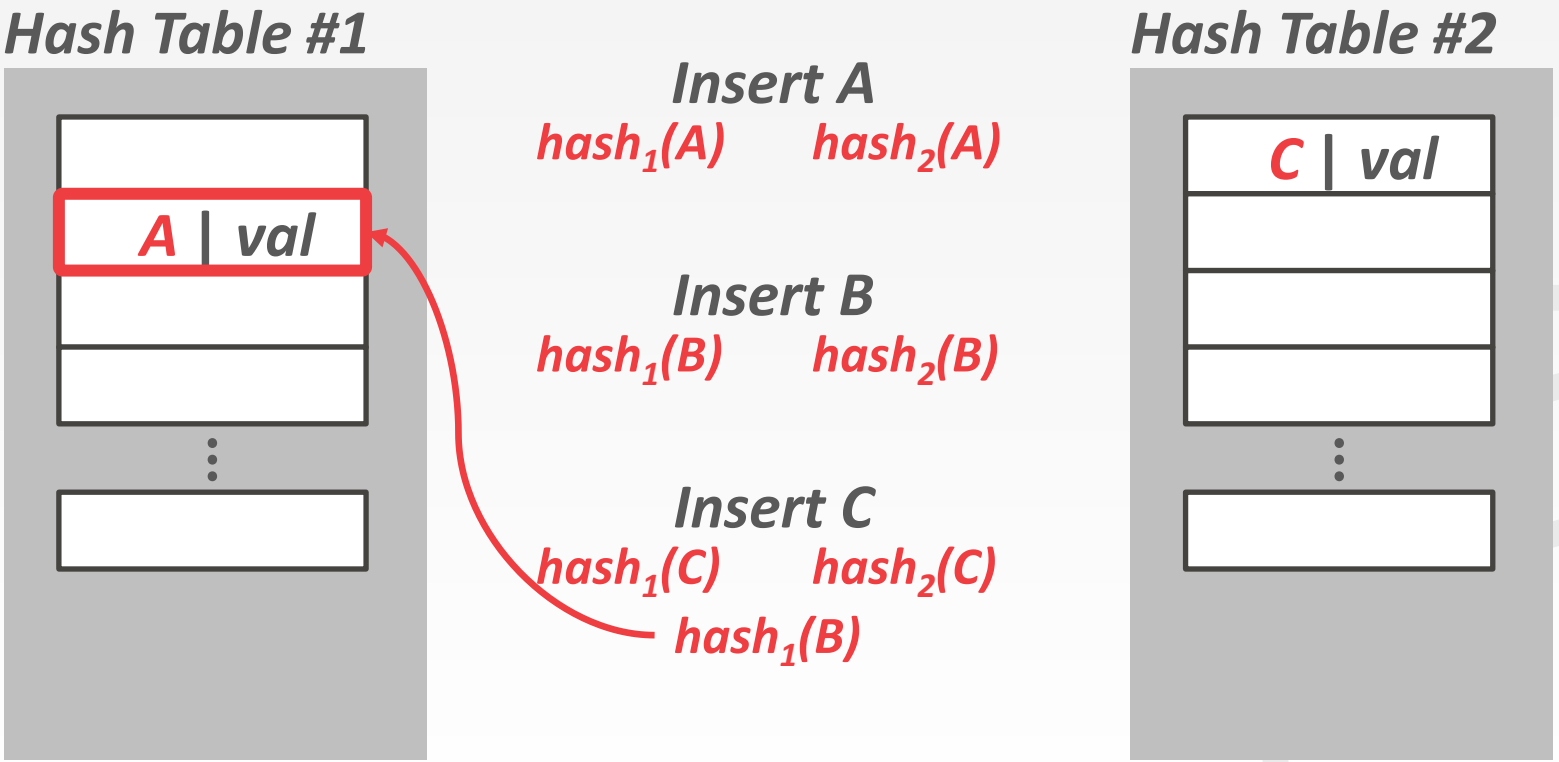
CUCKOO HASHING



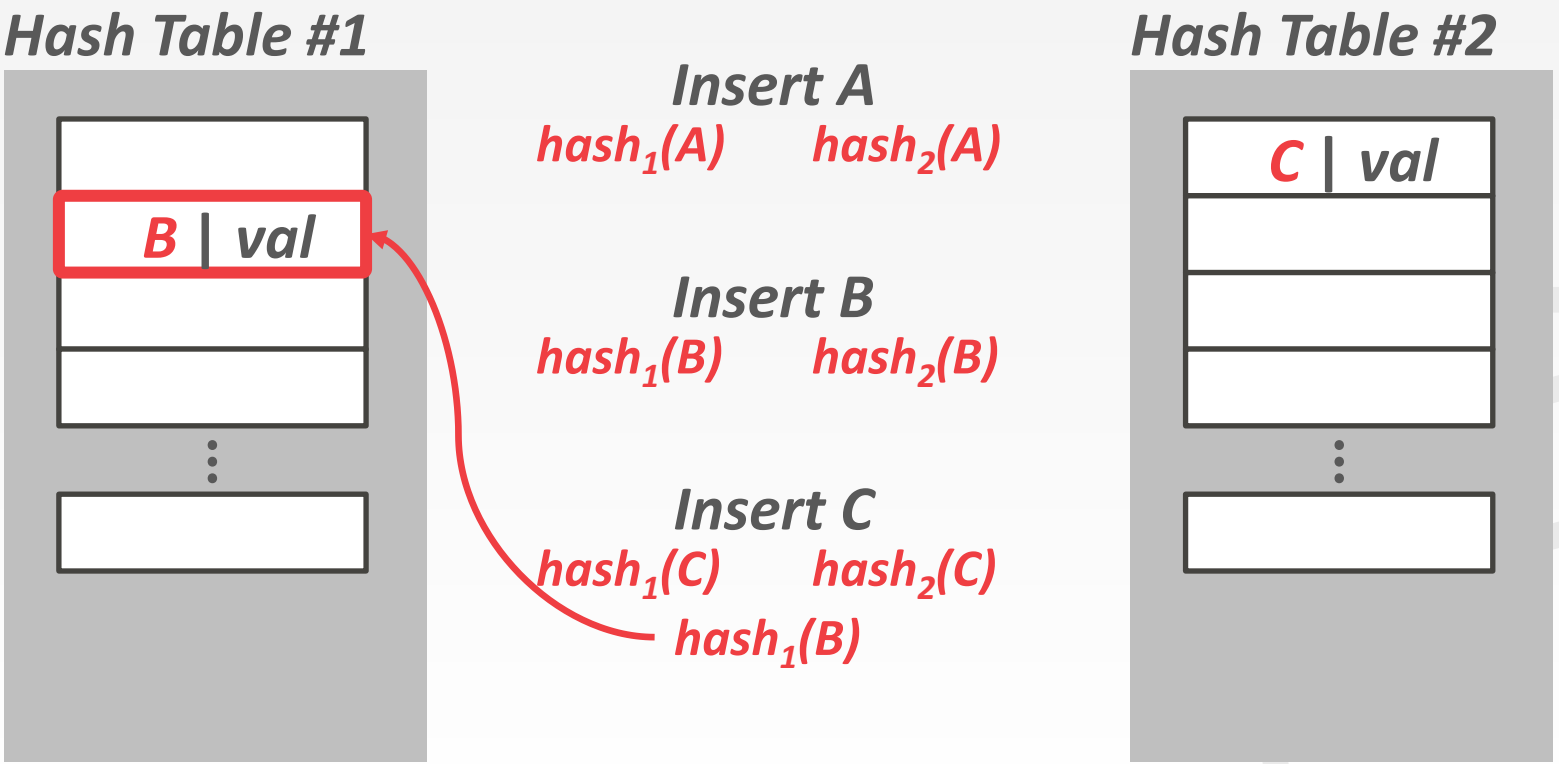
CUCKOO HASHING



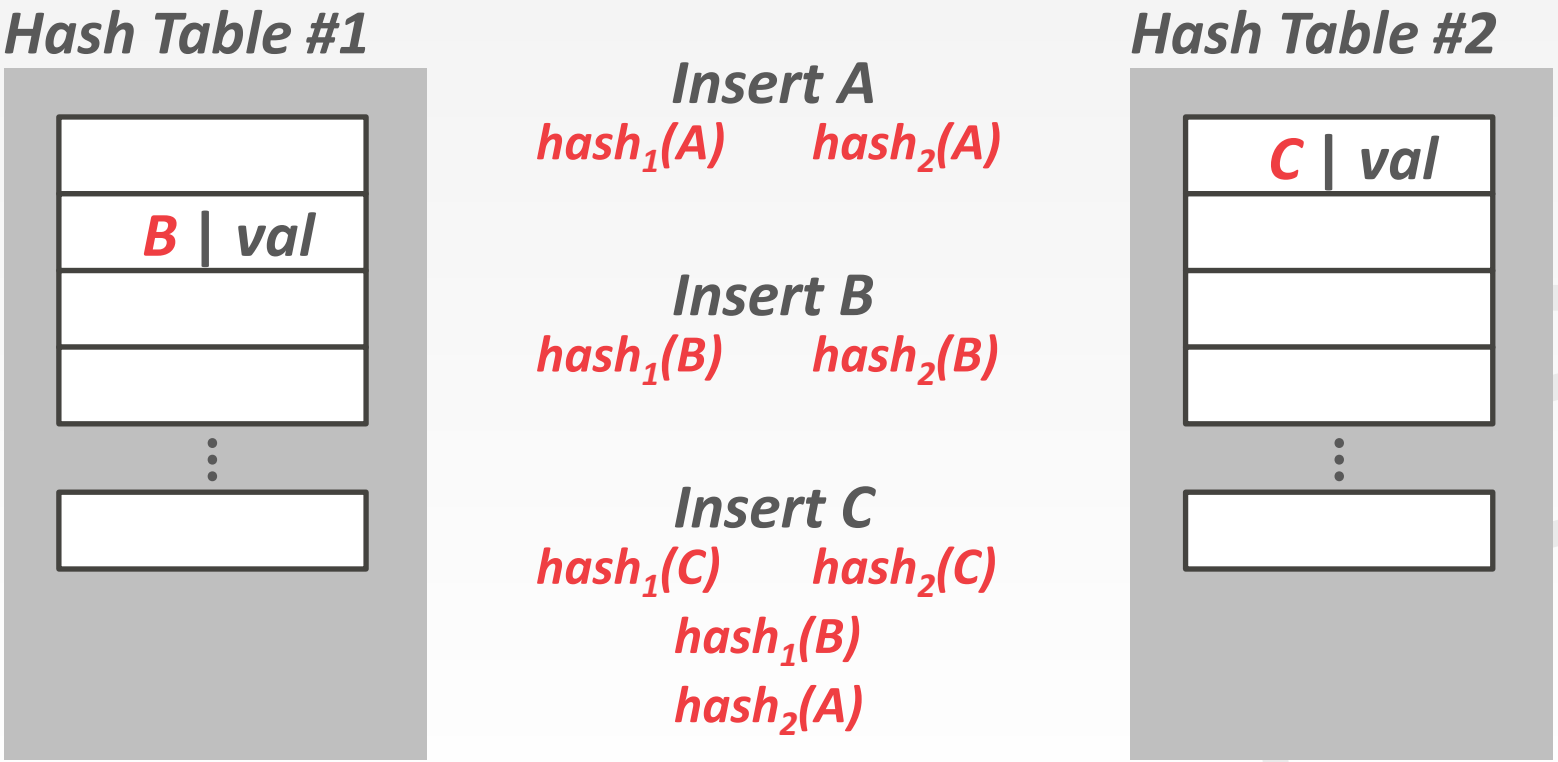
CUCKOO HASHING



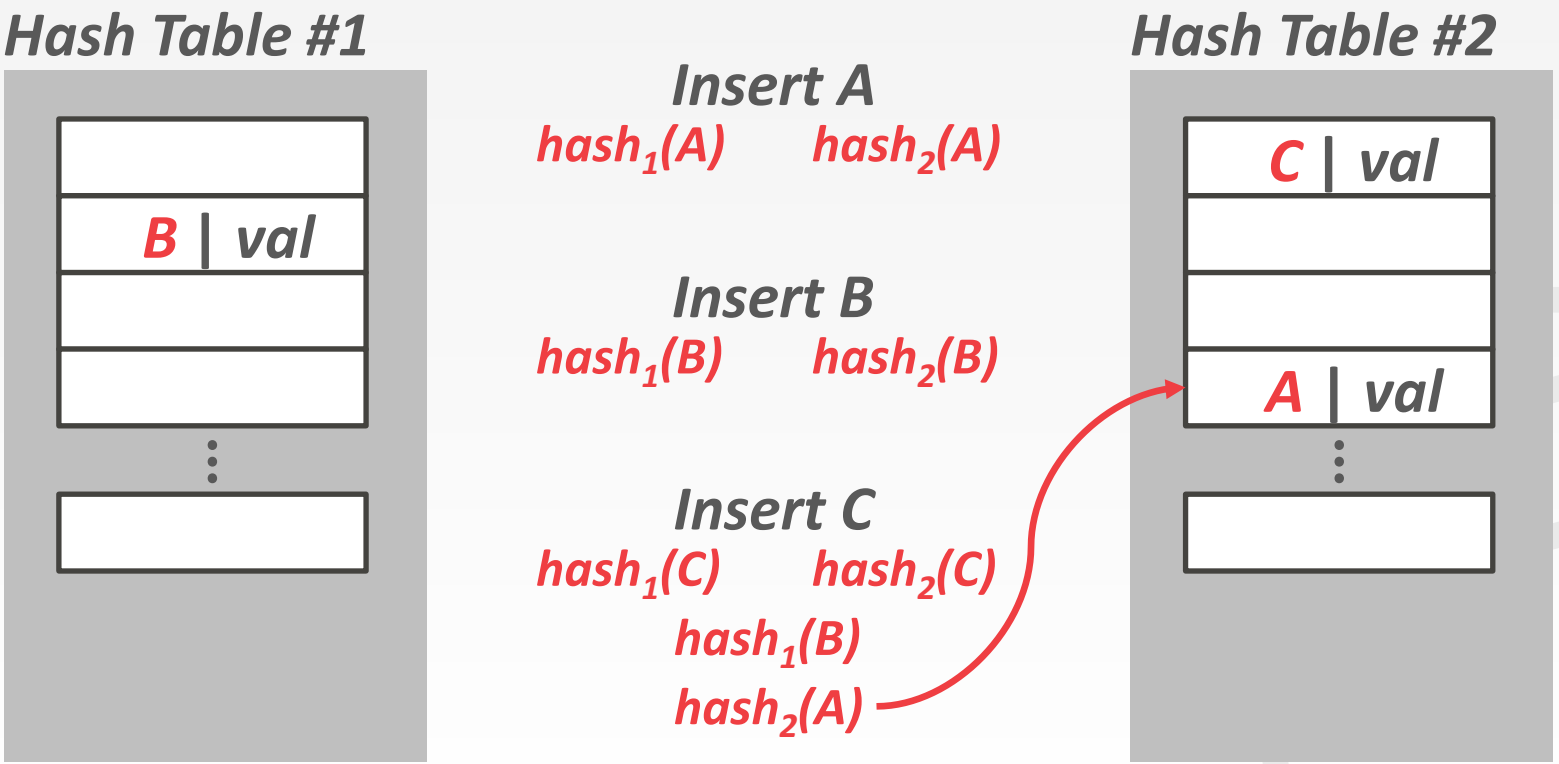
CUCKOO HASHING



CUCKOO HASHING



CUCKOO HASHING



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.

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

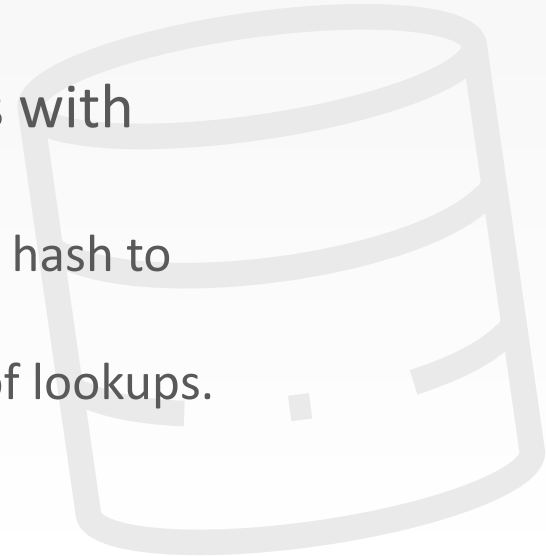
Maintain a linked list of buckets for each slot in the hash table.

→ Used in Java HashMap class

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.

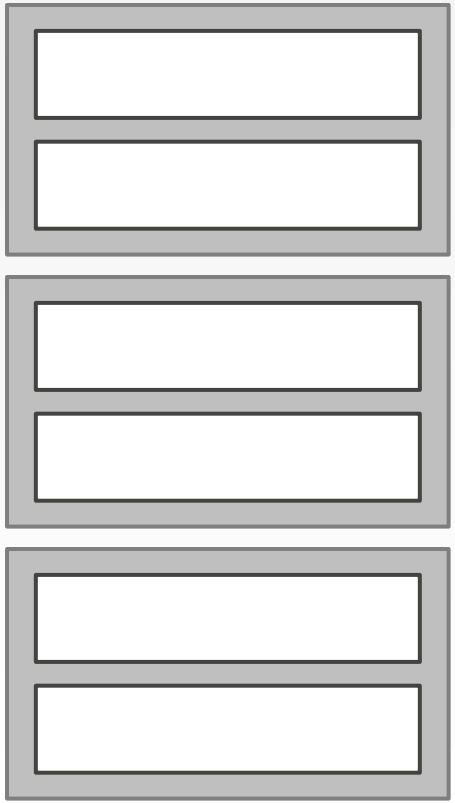
→ Insertions and deletions are generalizations of lookups.



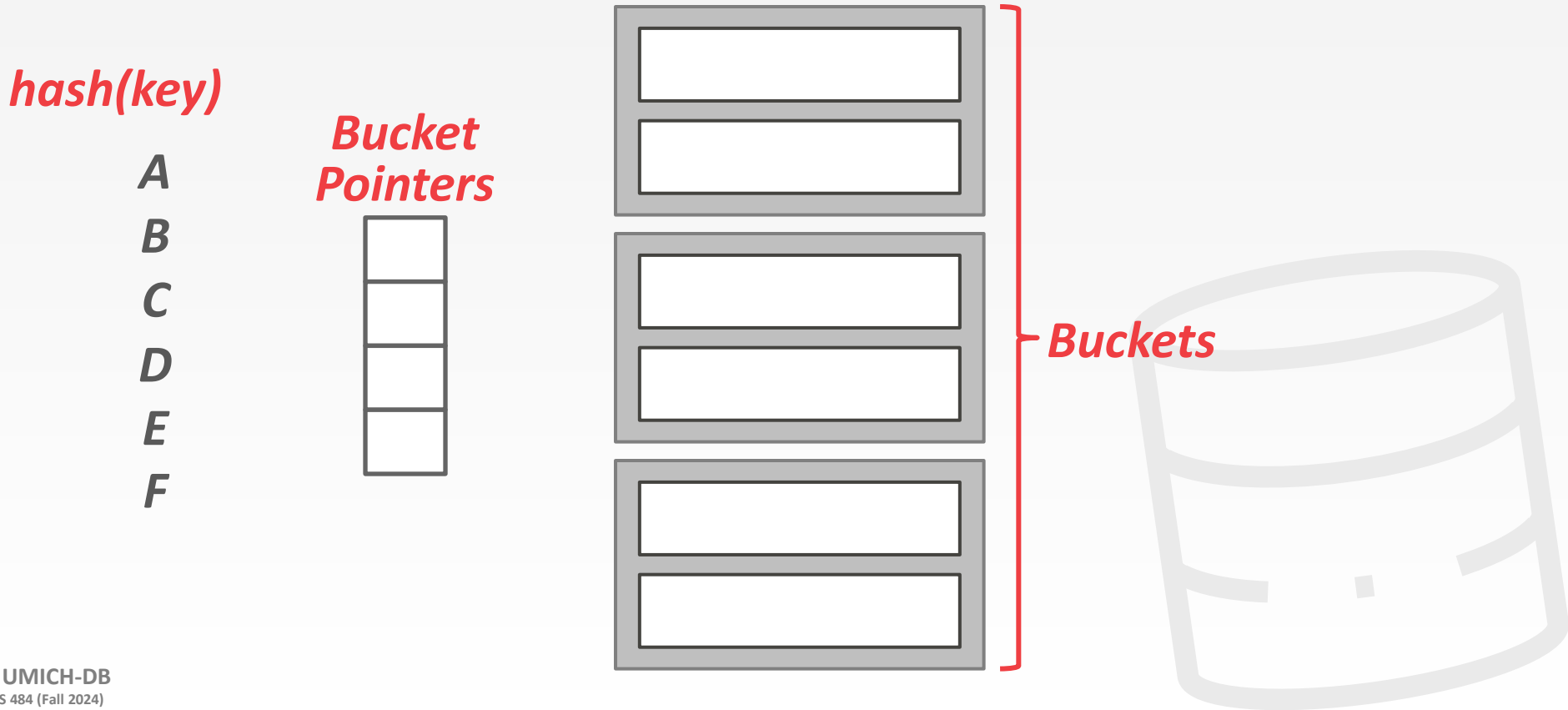
CHAINED HASHING

hash(key)

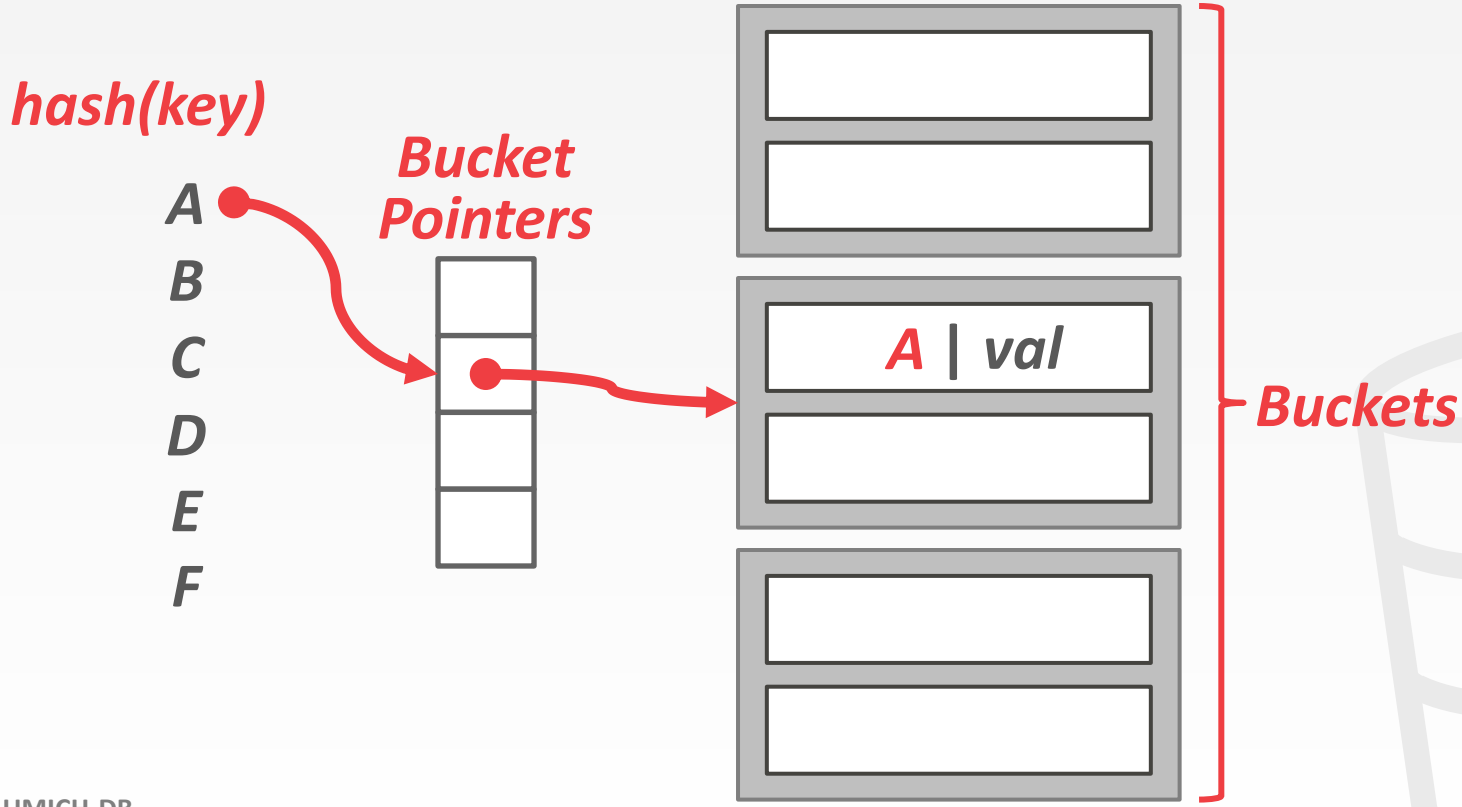
A
B
C
D
E
F



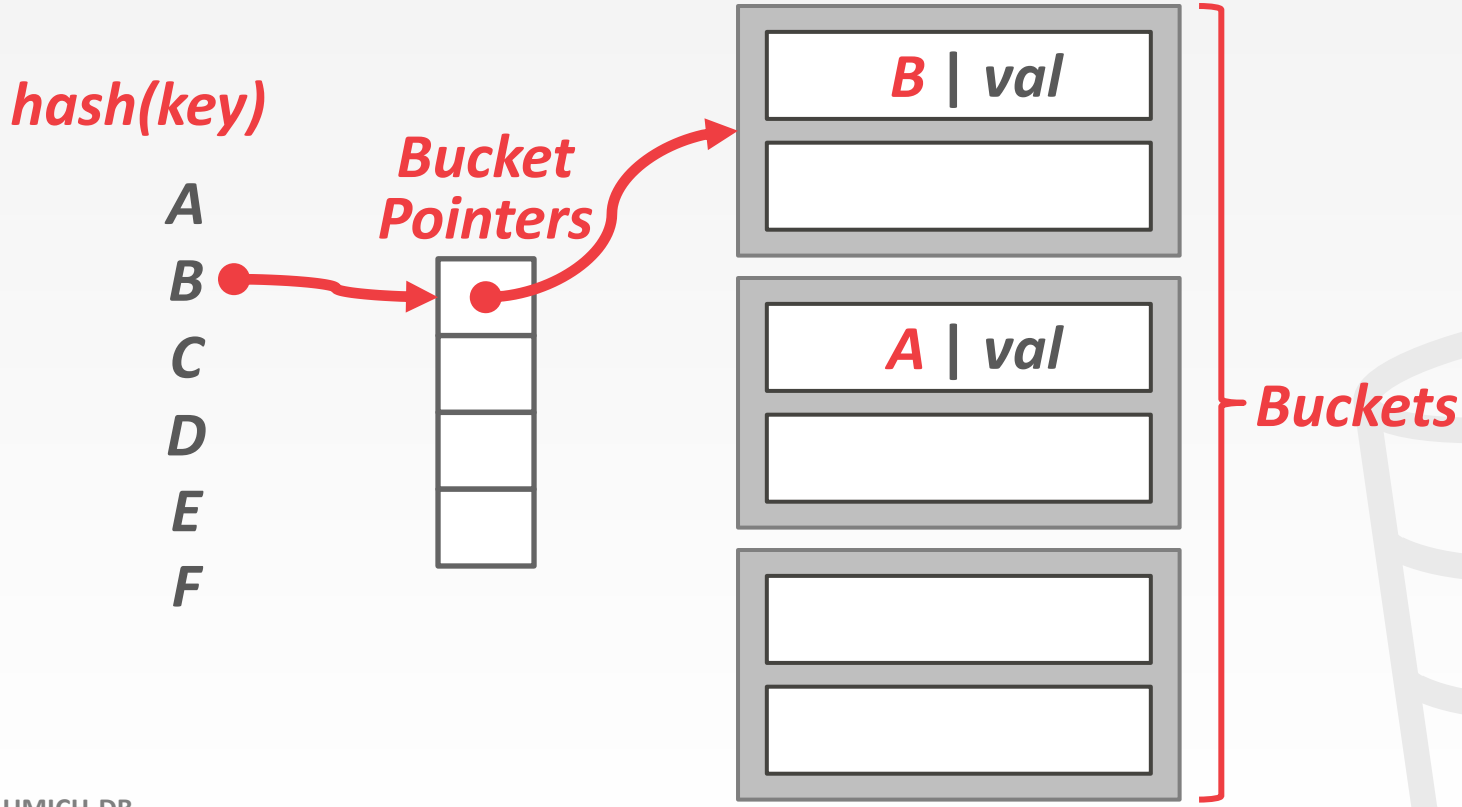
CHAINED HASHING



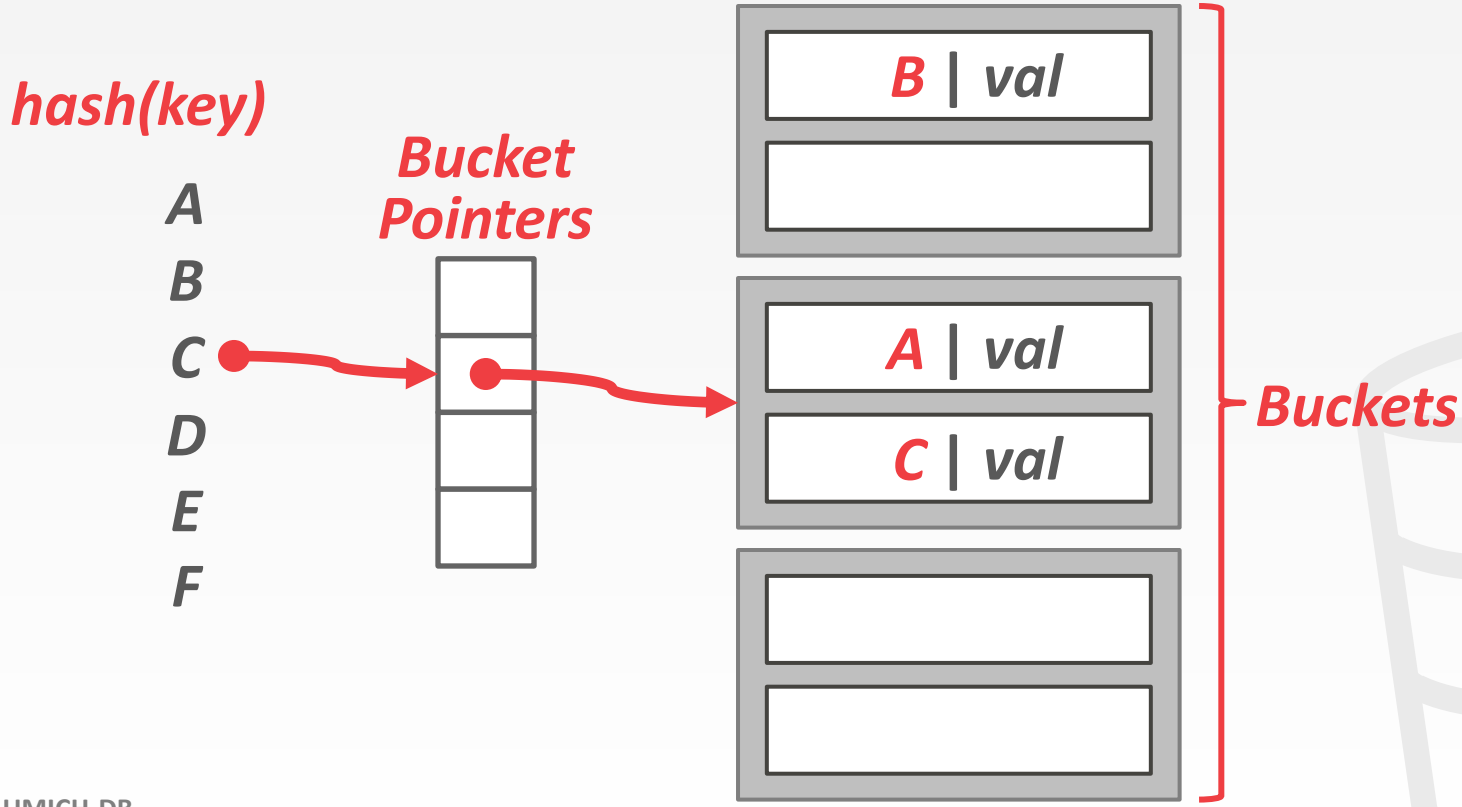
CHAINED HASHING



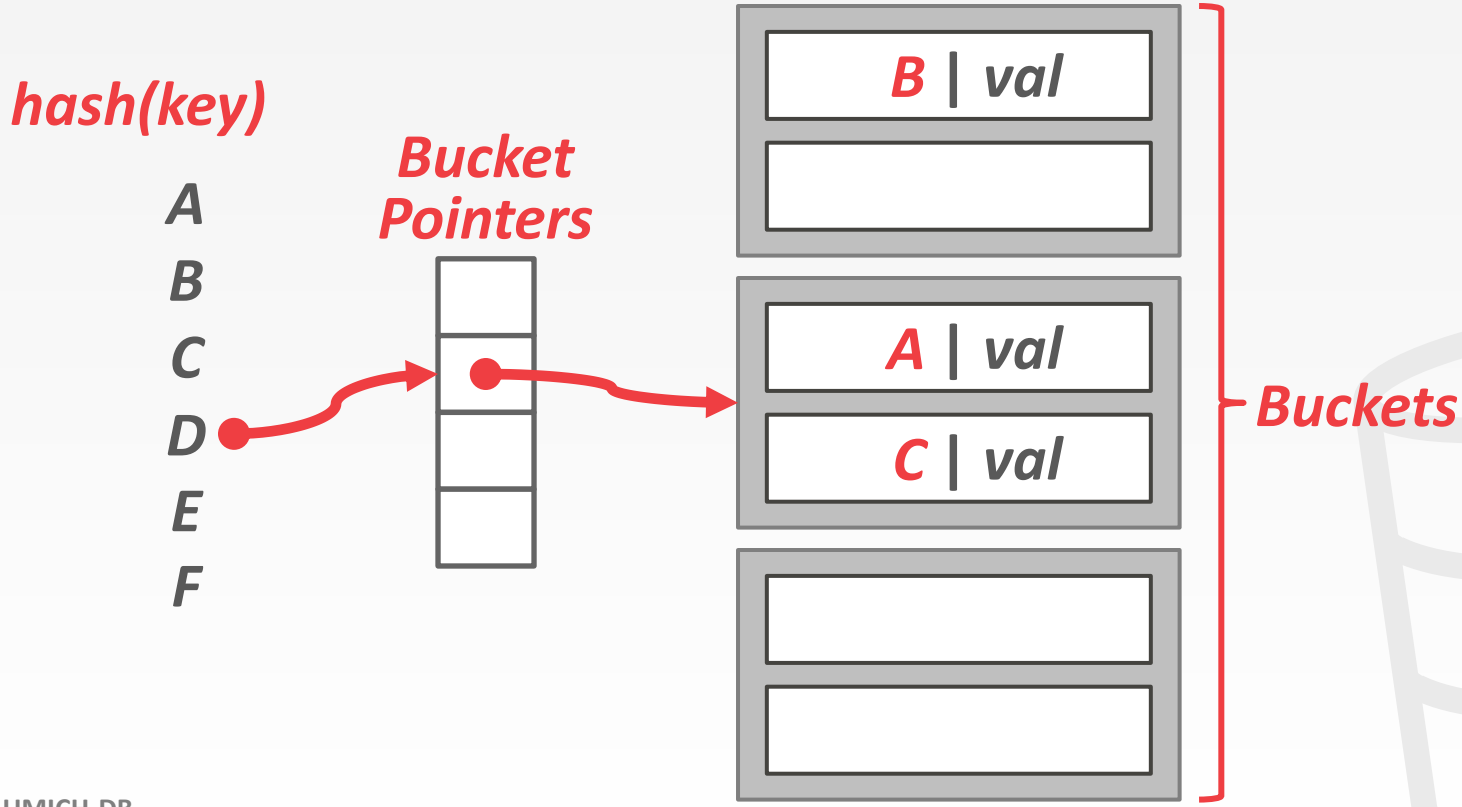
CHAINED HASHING



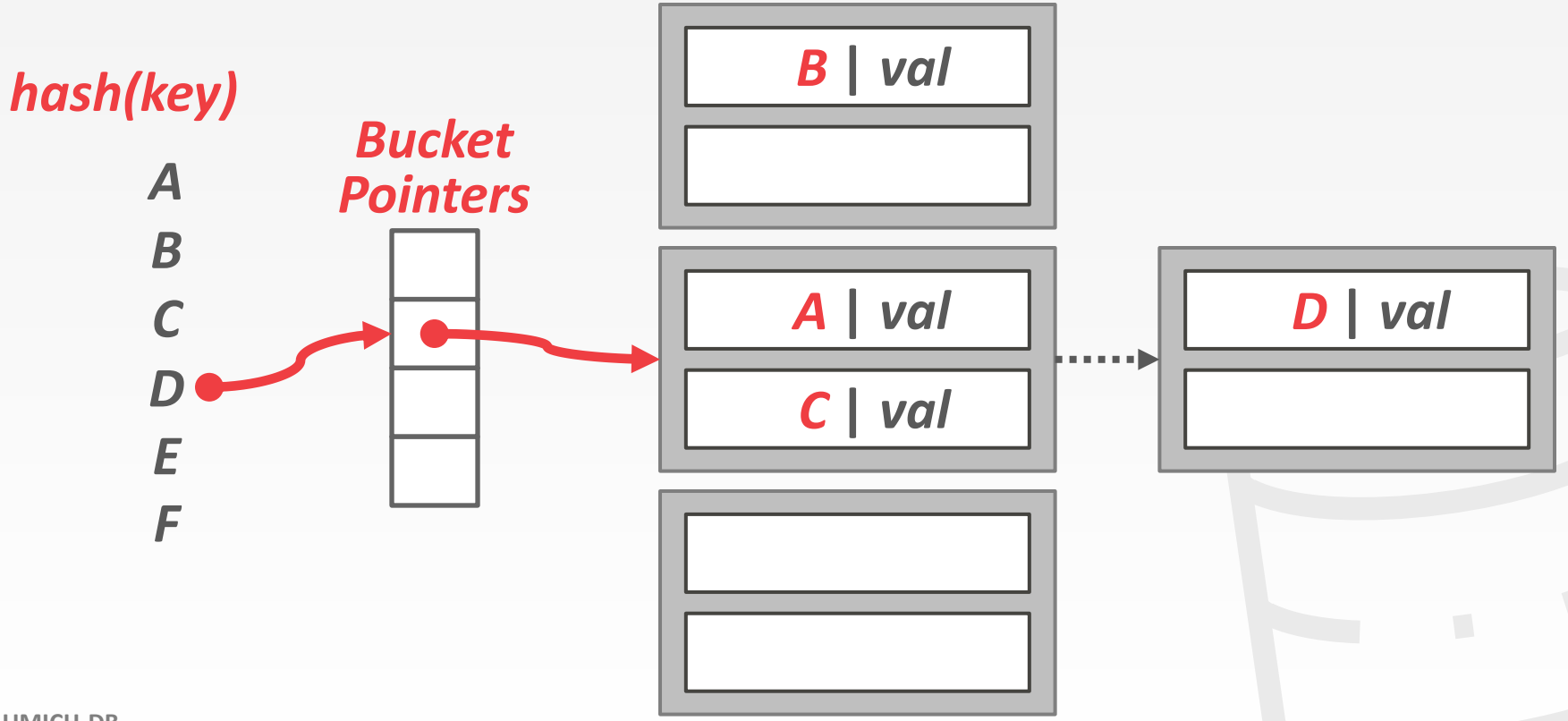
CHAINED HASHING



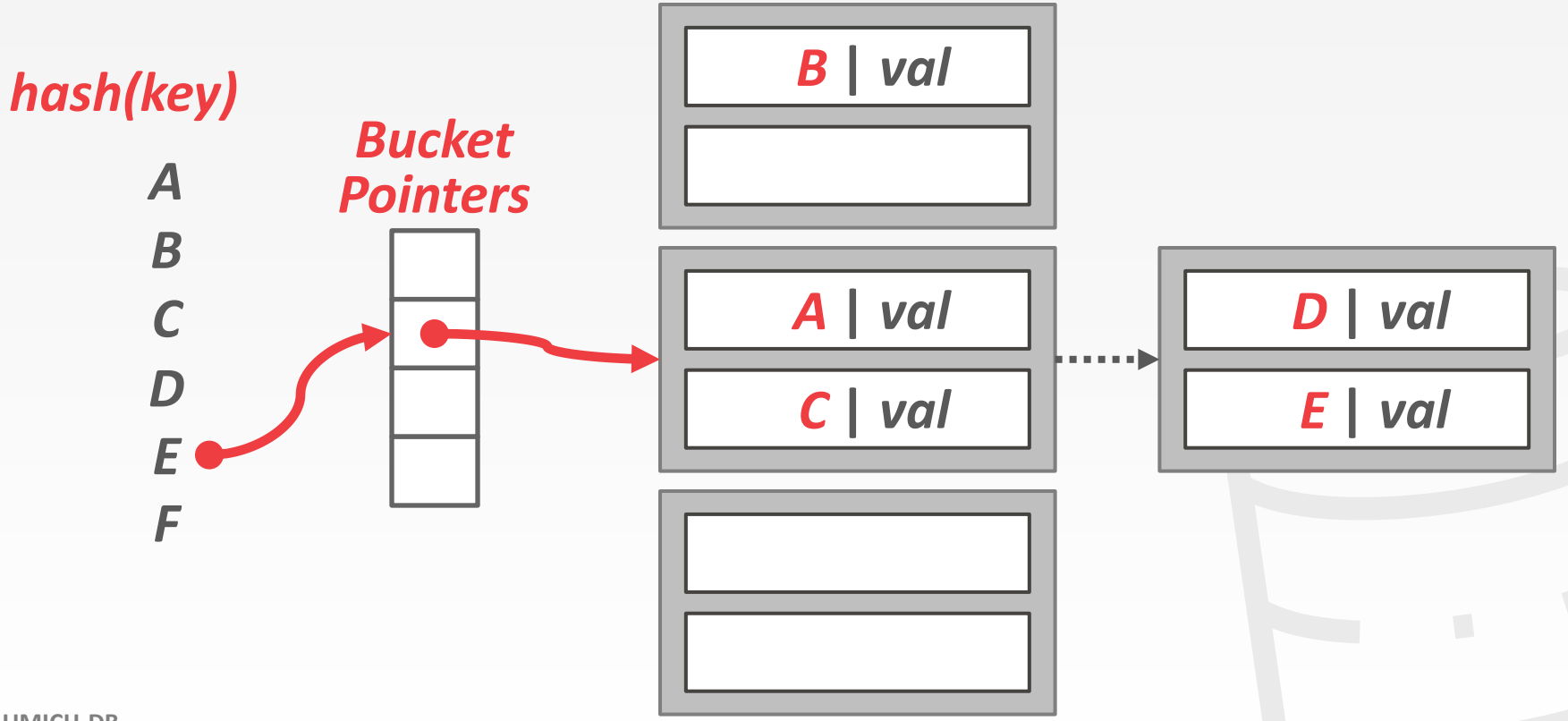
CHAINED HASHING



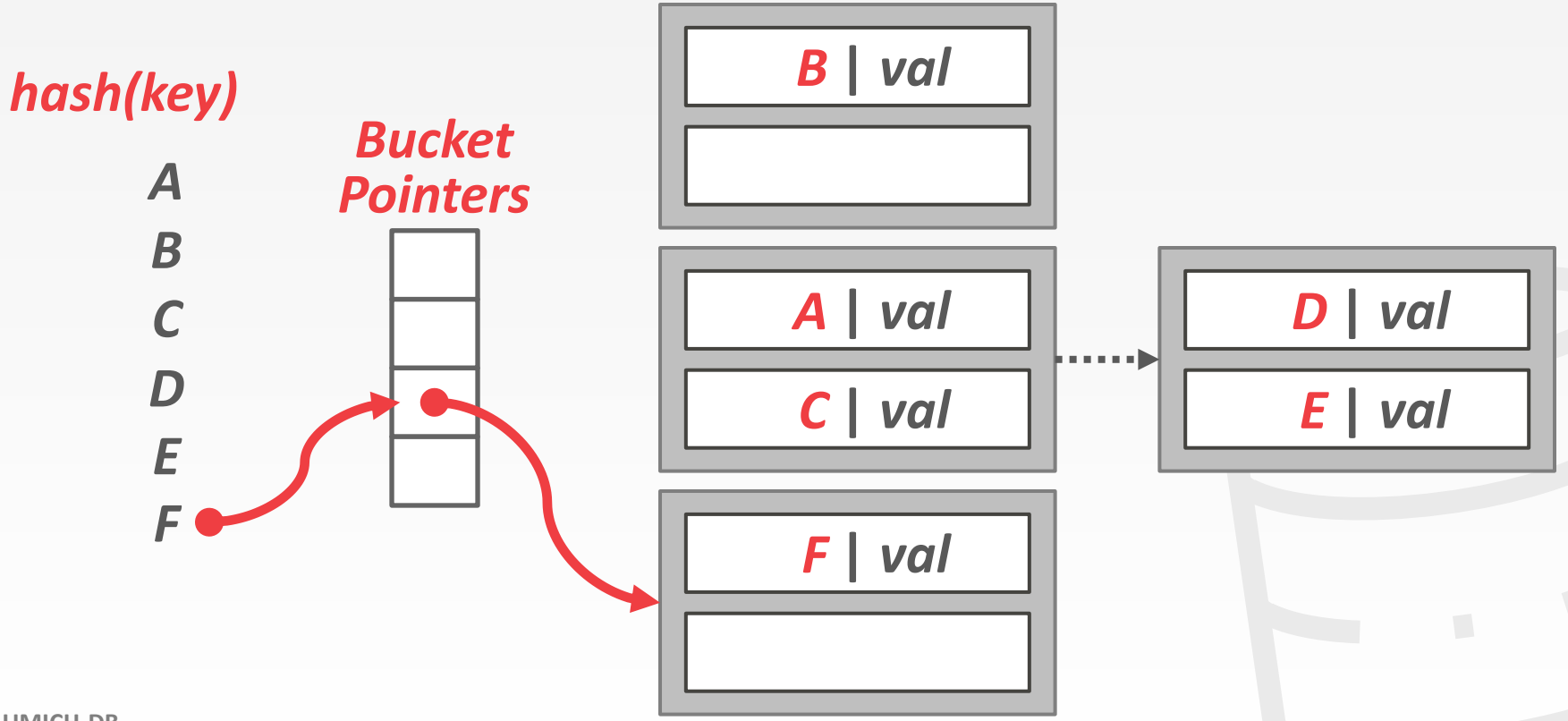
CHAINED HASHING



CHAINED HASHING



CHAINED HASHING



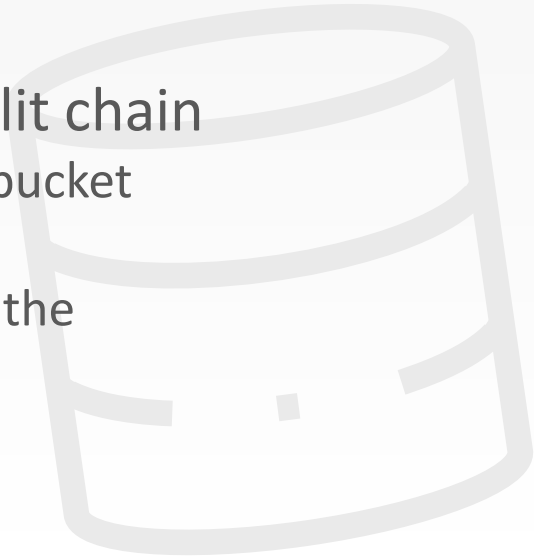
EXTENDIBLE HASHING

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

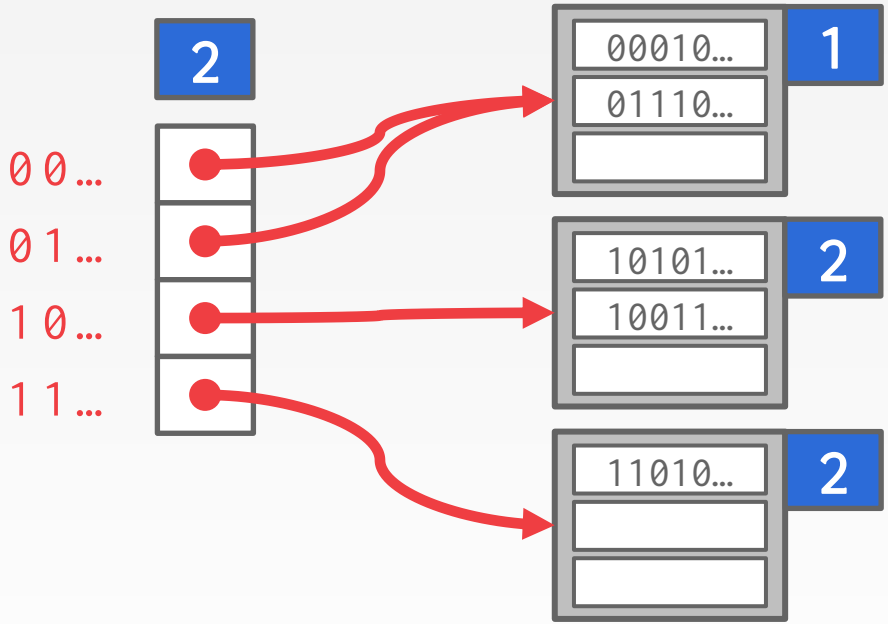
Data movement is localized to just the split chain

→ Multiple slot locations can point to the same bucket chain.

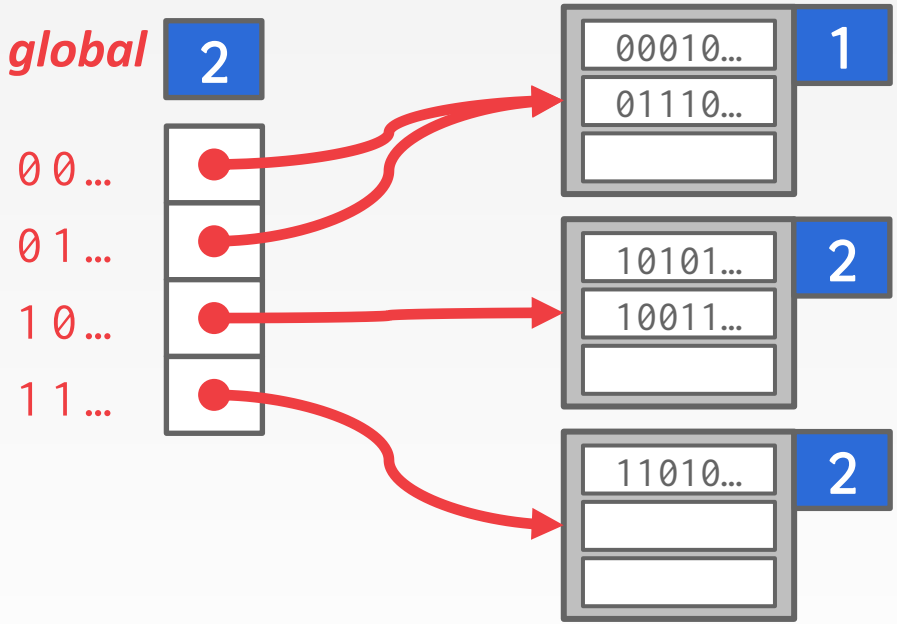
→ Reshuffle bucket entries on split and increase the number of bits to examine.



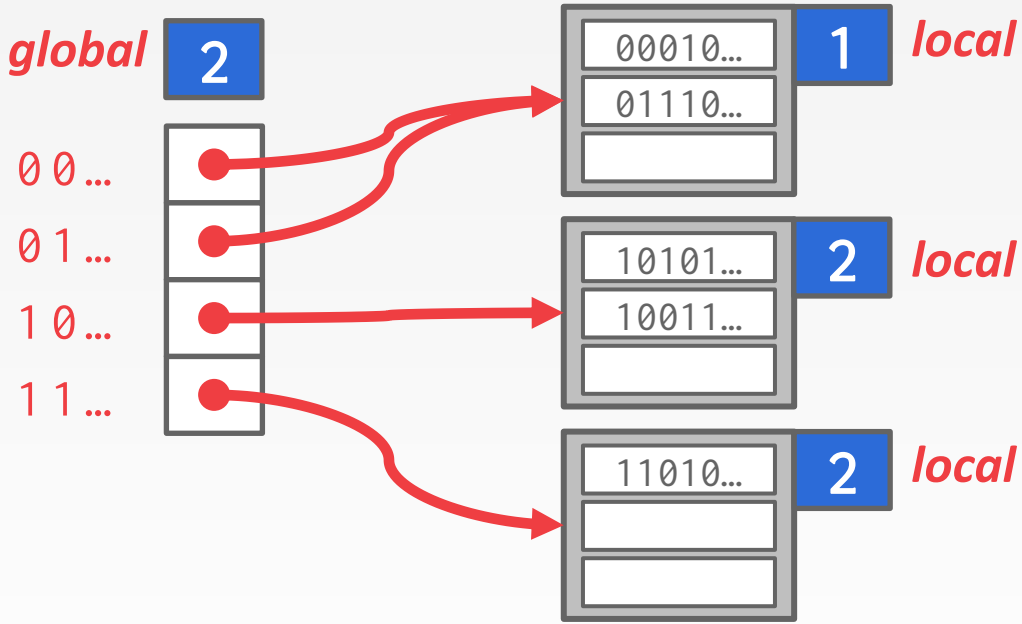
EXTENDIBLE HASHING



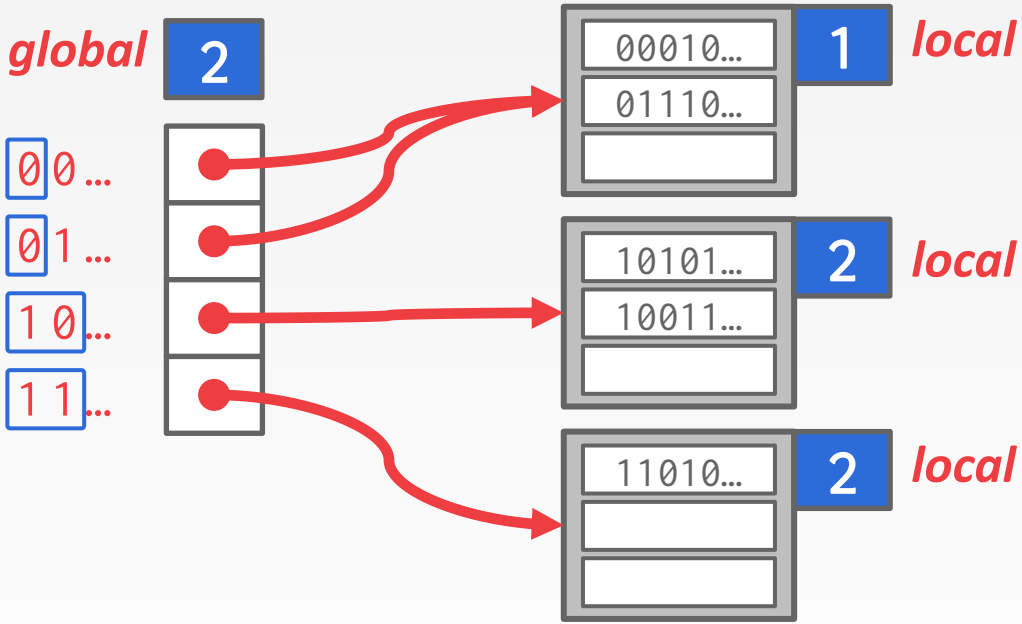
EXTENDIBLE HASHING



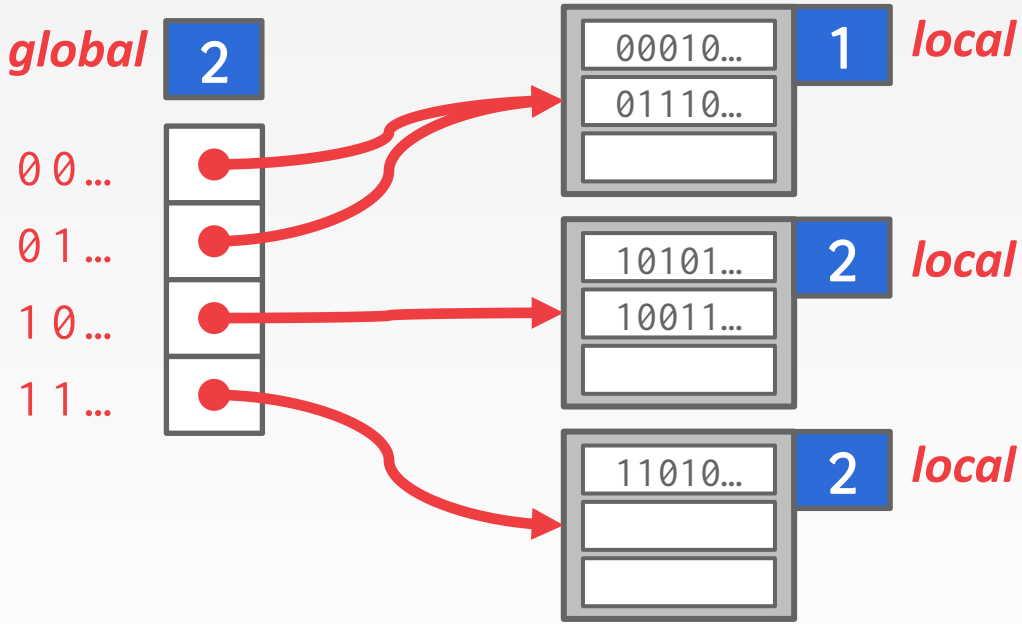
EXTENDIBLE HASHING



EXTENDIBLE HASHING

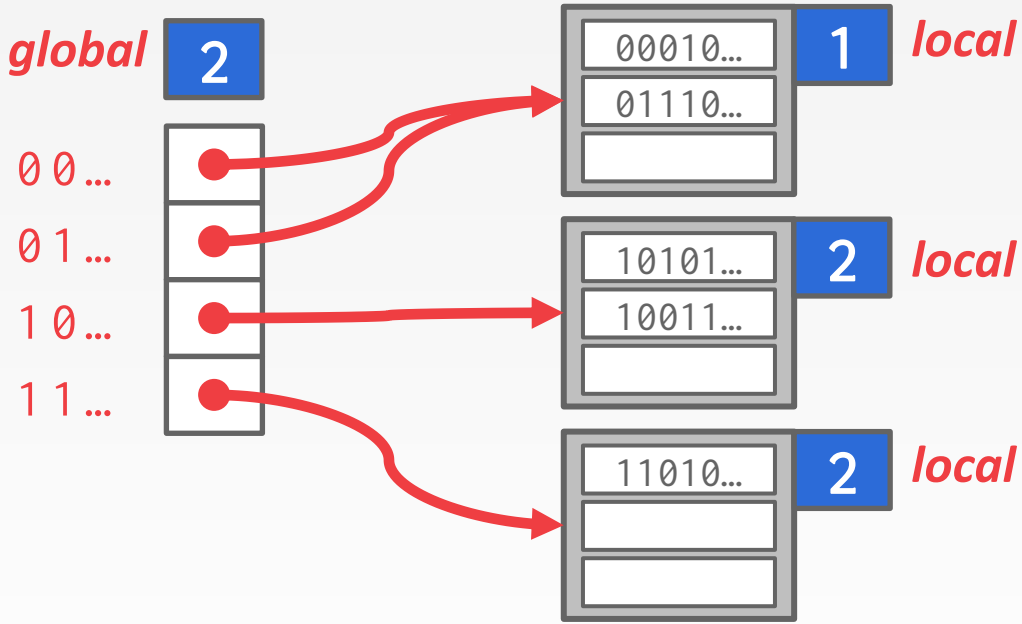


EXTENDIBLE HASHING



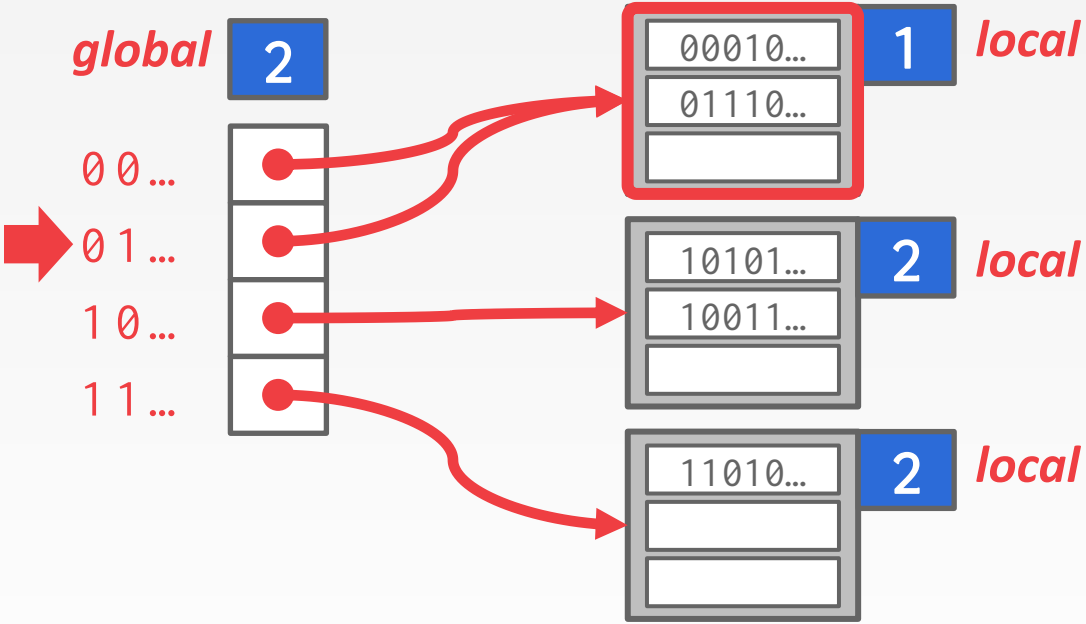
Find A
hash(A) = 01110...

EXTENDIBLE HASHING



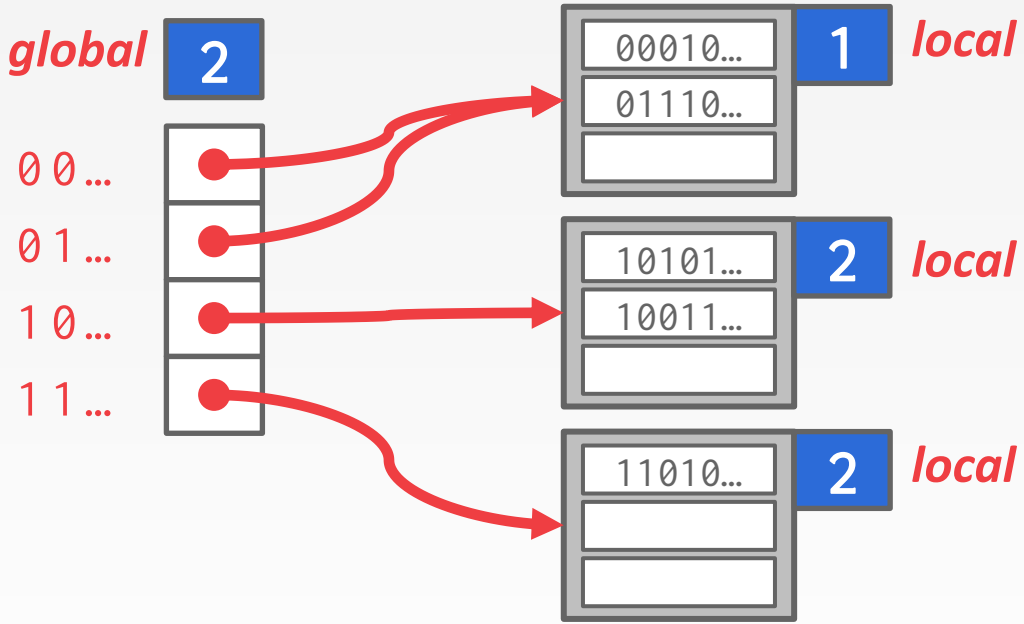
Find A
hash(A) = 01110...

EXTENDIBLE HASHING



Find A
hash(A) = 01110...

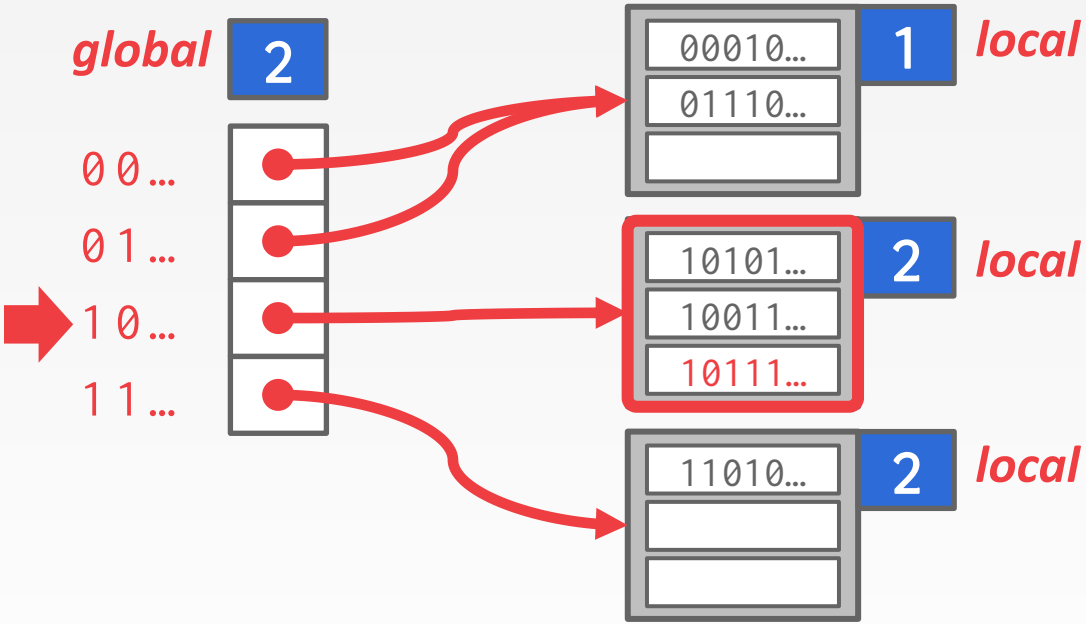
EXTENDIBLE HASHING



Find A
hash(A) = 01110...

Insert B
hash(B) = 10111...

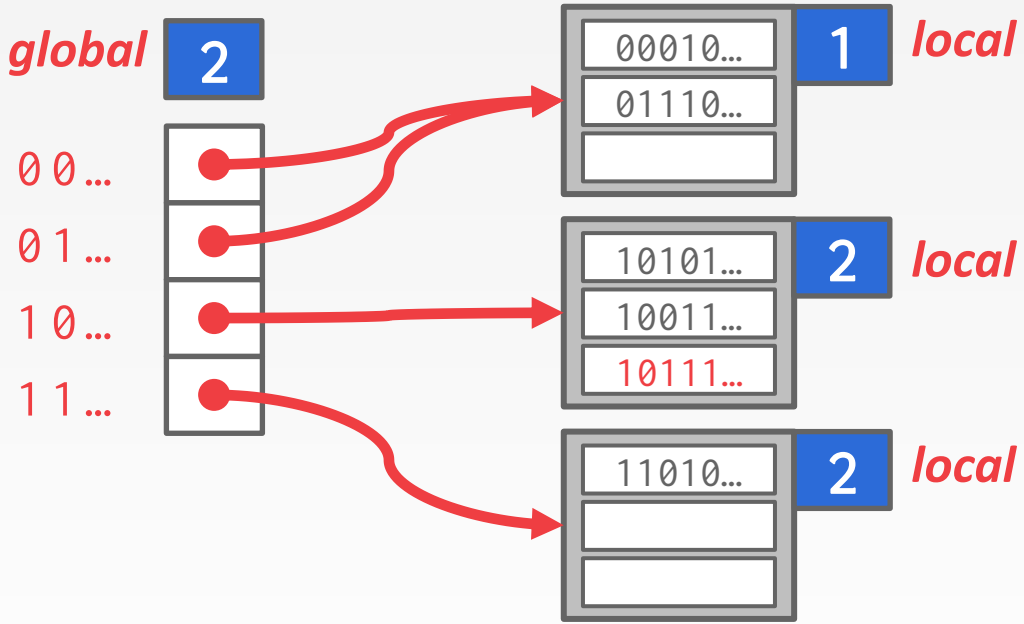
EXTENDIBLE HASHING



Find A
hash(A) = 01110...

Insert B
hash(B) = 10111...

EXTENDIBLE HASHING

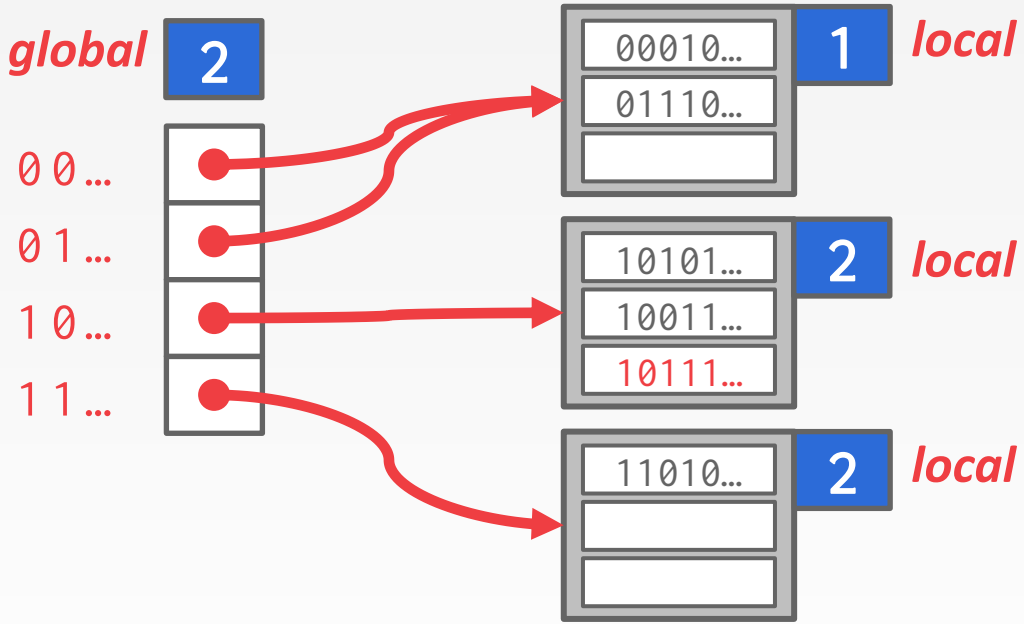


Find A
hash(A) = 01110...

Insert B
hash(B) = 10111...

Insert C
hash(C) = 10100...

EXTENDIBLE HASHING

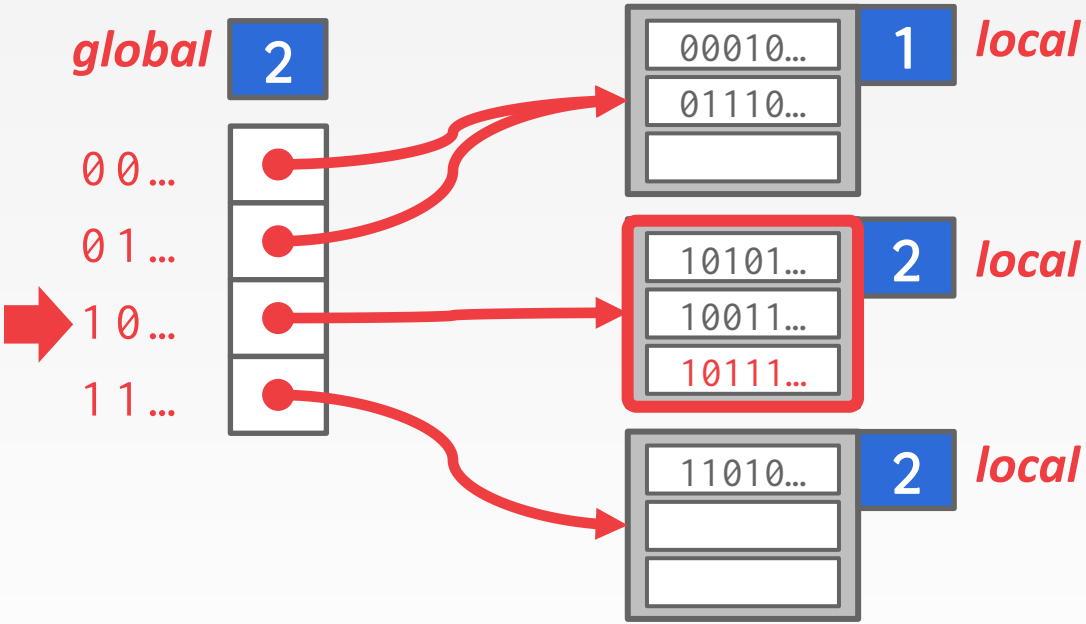


Find A
hash(A) = 01110...

Insert B
hash(B) = 10111...

Insert C
hash(C) = 10100...

EXTENDIBLE HASHING

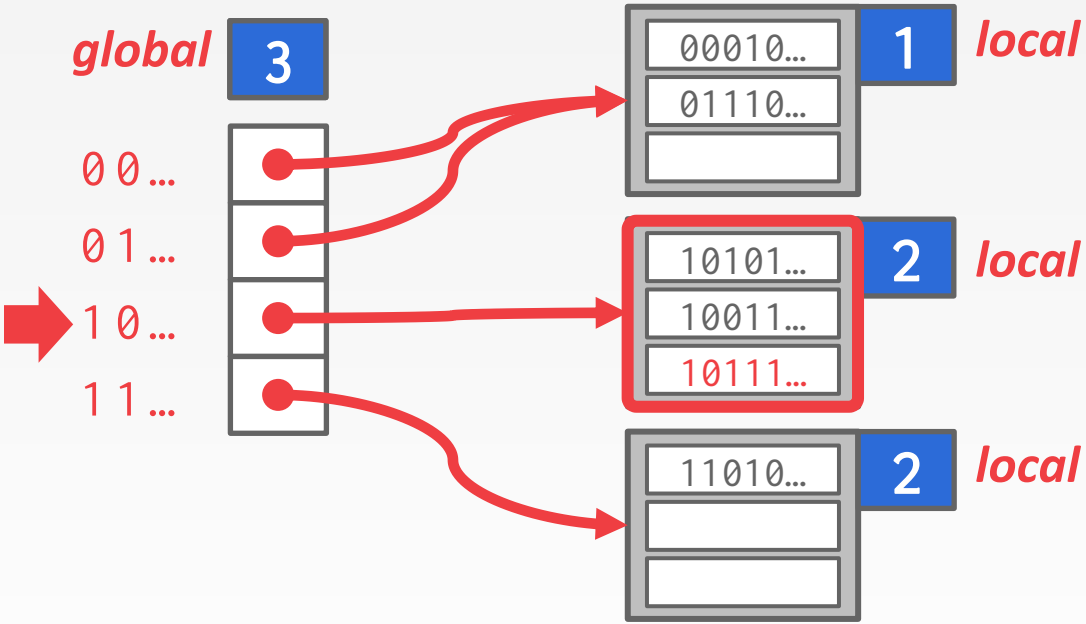


Find A
hash(A) = 01110...

Insert B
hash(B) = 10111...

Insert C
hash(C) = 10100...

EXTENDIBLE HASHING

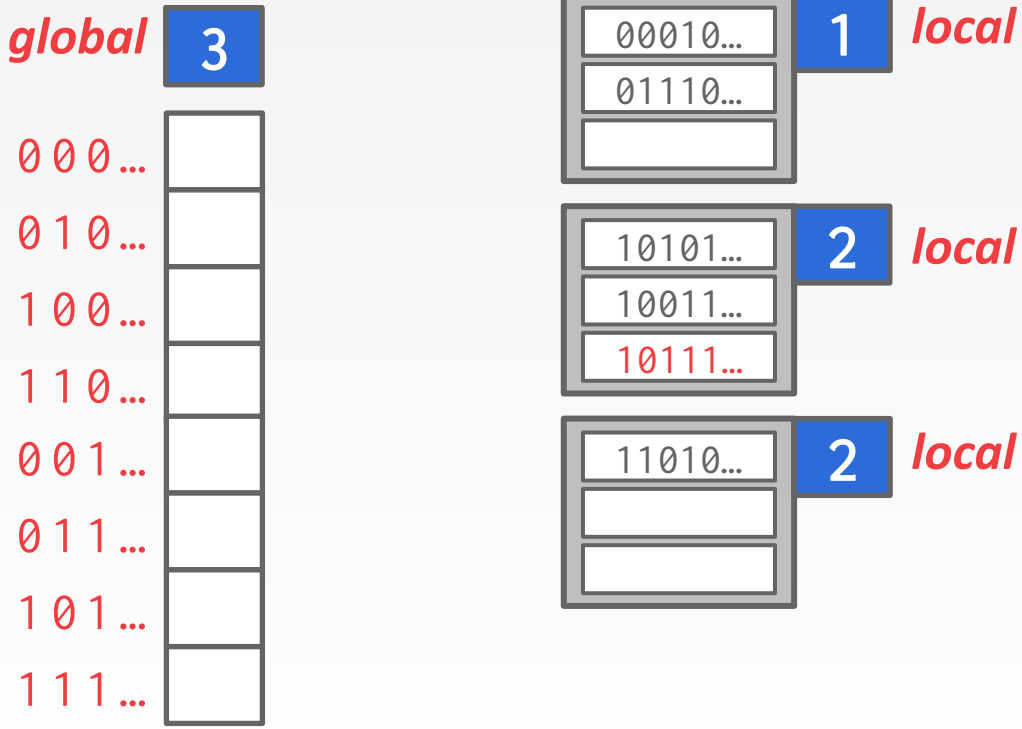


Find A
hash(A) = 01110...

Insert B
hash(B) = 10111...

Insert C
hash(C) = 10100...

EXTENDIBLE HASHING

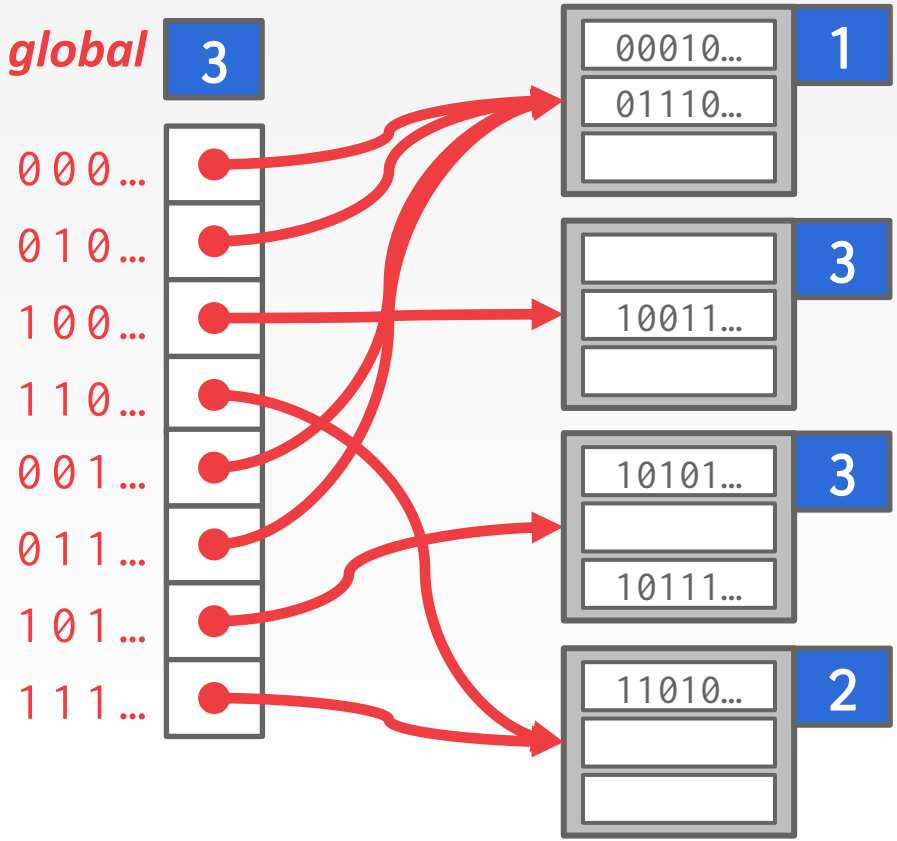


Find A
hash(A) = 01110…

Insert B
hash(B) = 10111…

Insert C
hash(C) = 10100…

EXTENDIBLE HASHING

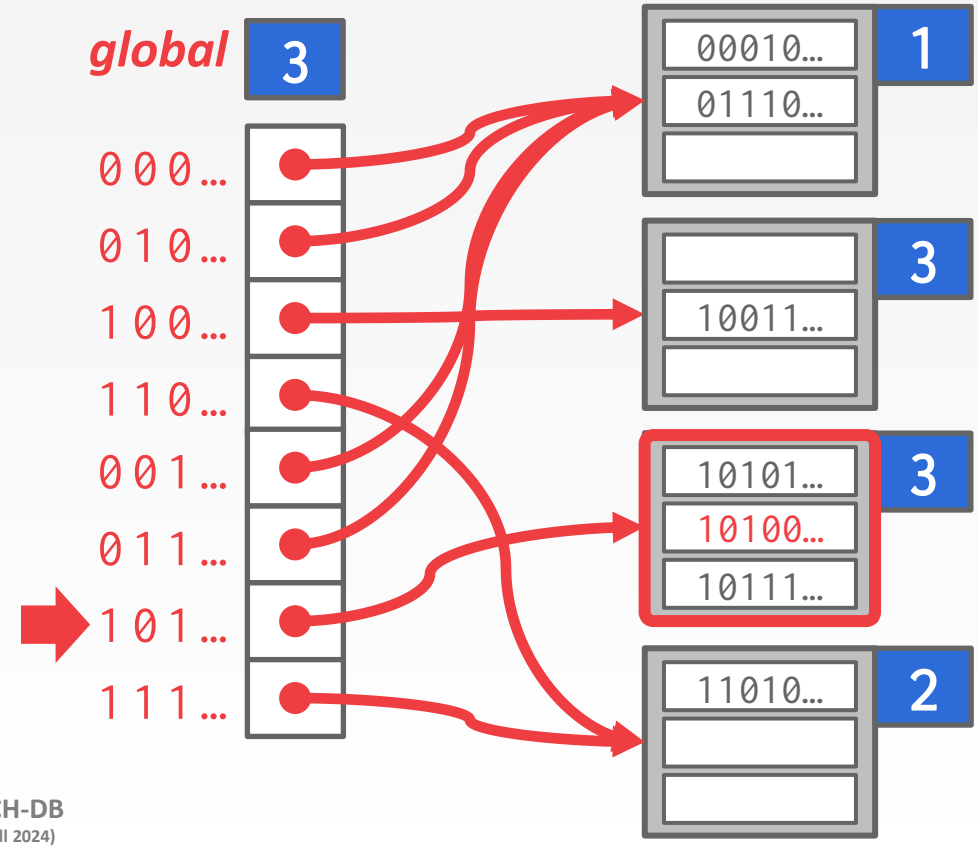


Find A
hash(A) = 01110…

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

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

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

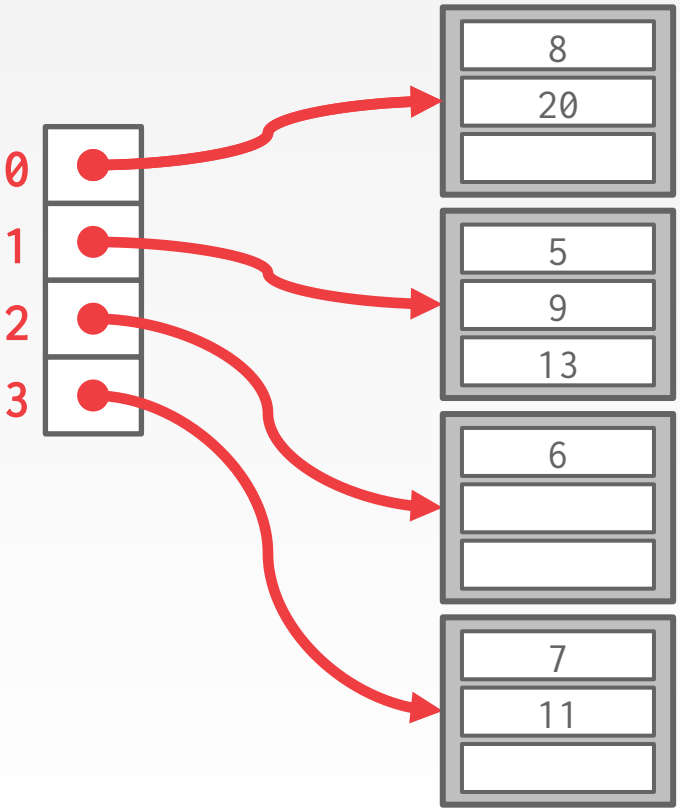
Use multiple hashes to find the right bucket for a given key.

Can use different overflow criterion:

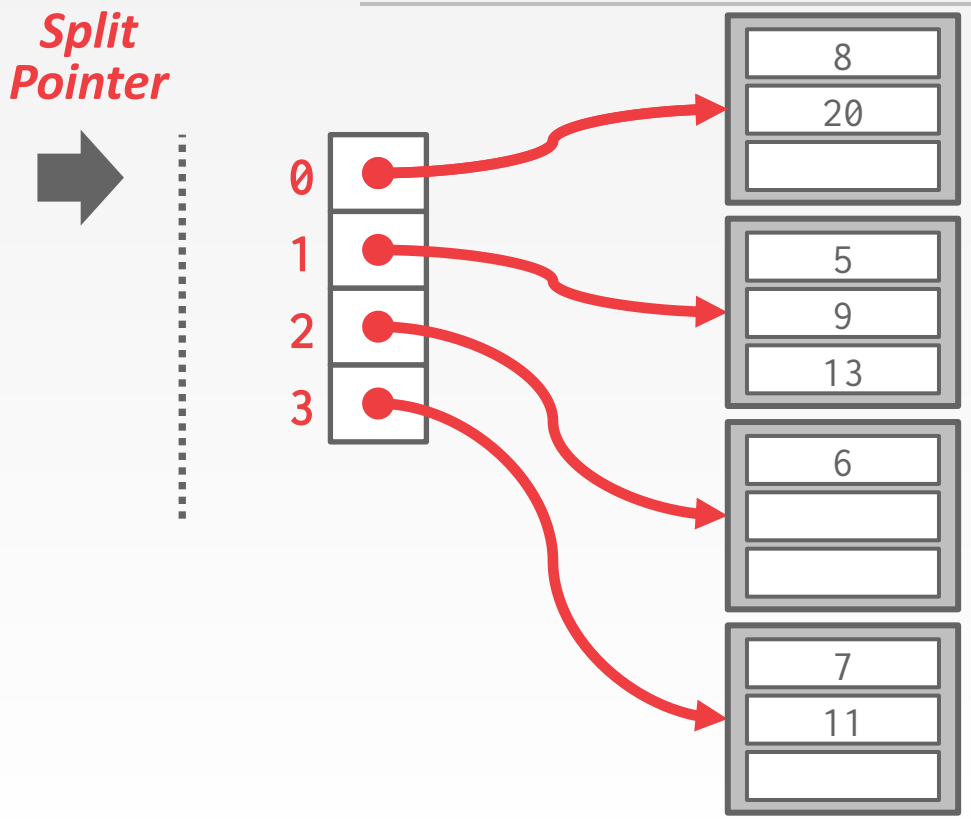
→ Space Utilization

→ Average Length of Overflow Chains

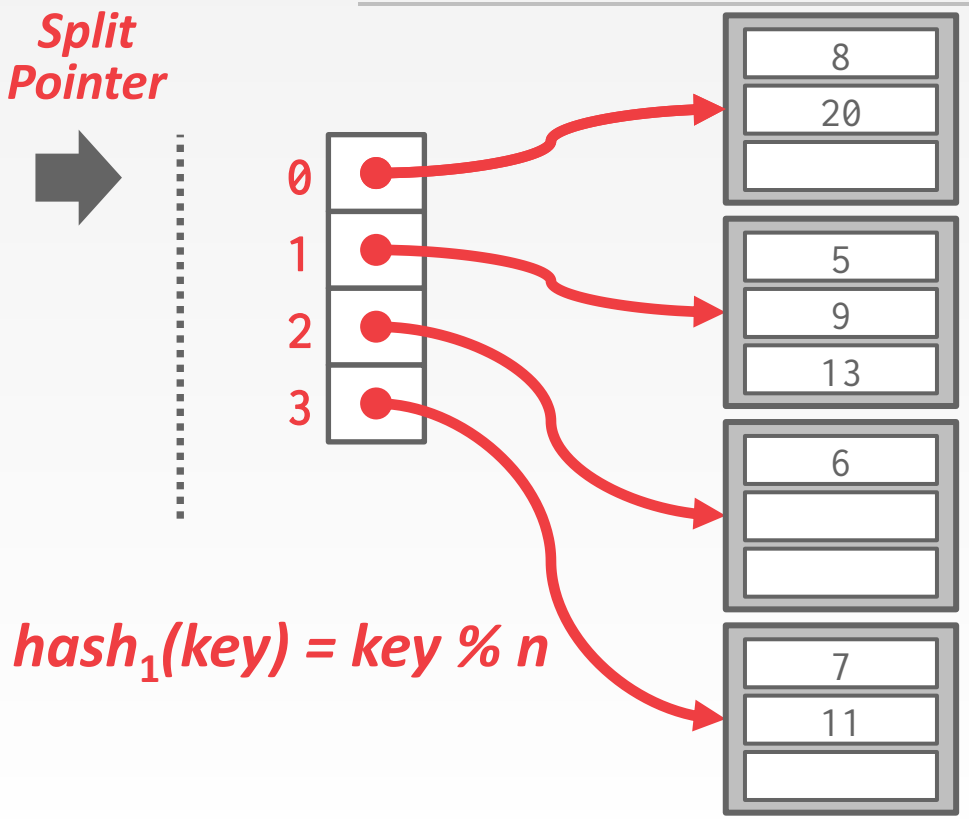
LINEAR HASHING



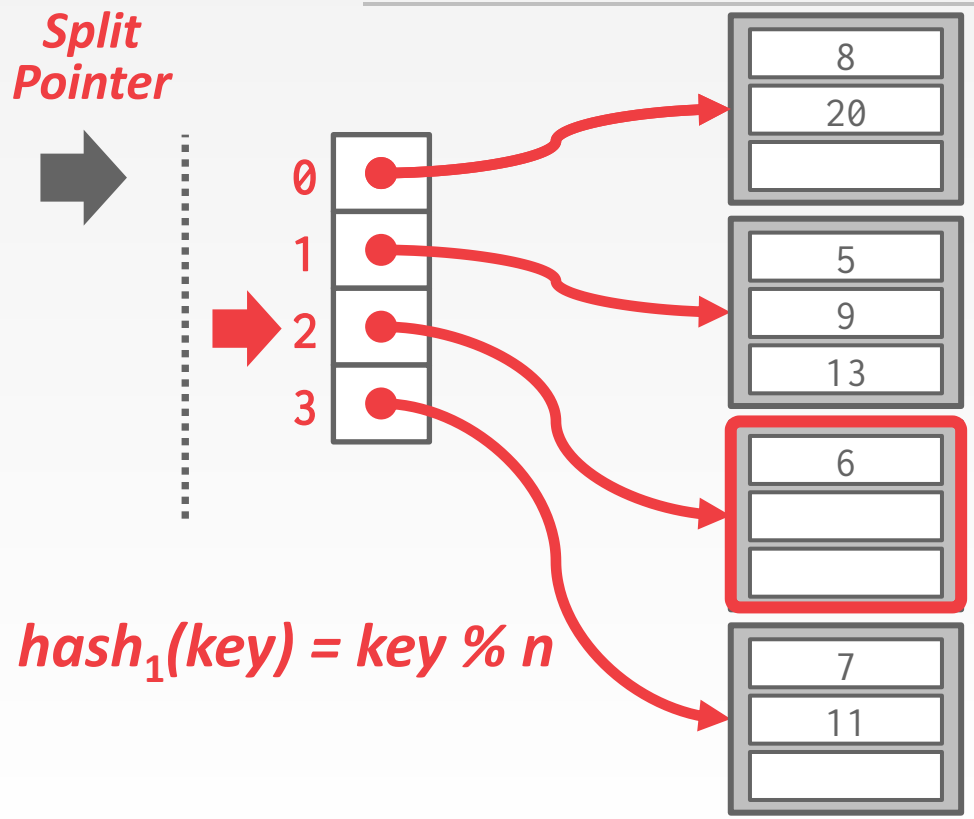
LINEAR HASHING



LINEAR HASHING



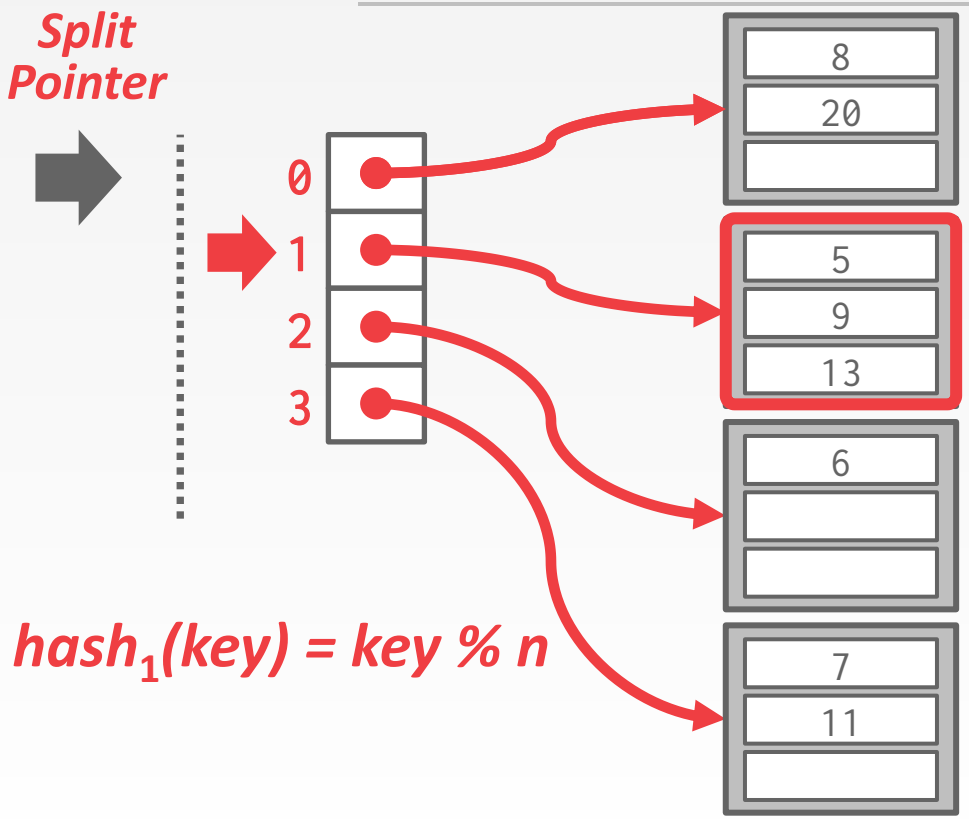
LINEAR HASHING



Find 6

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

LINEAR HASHING



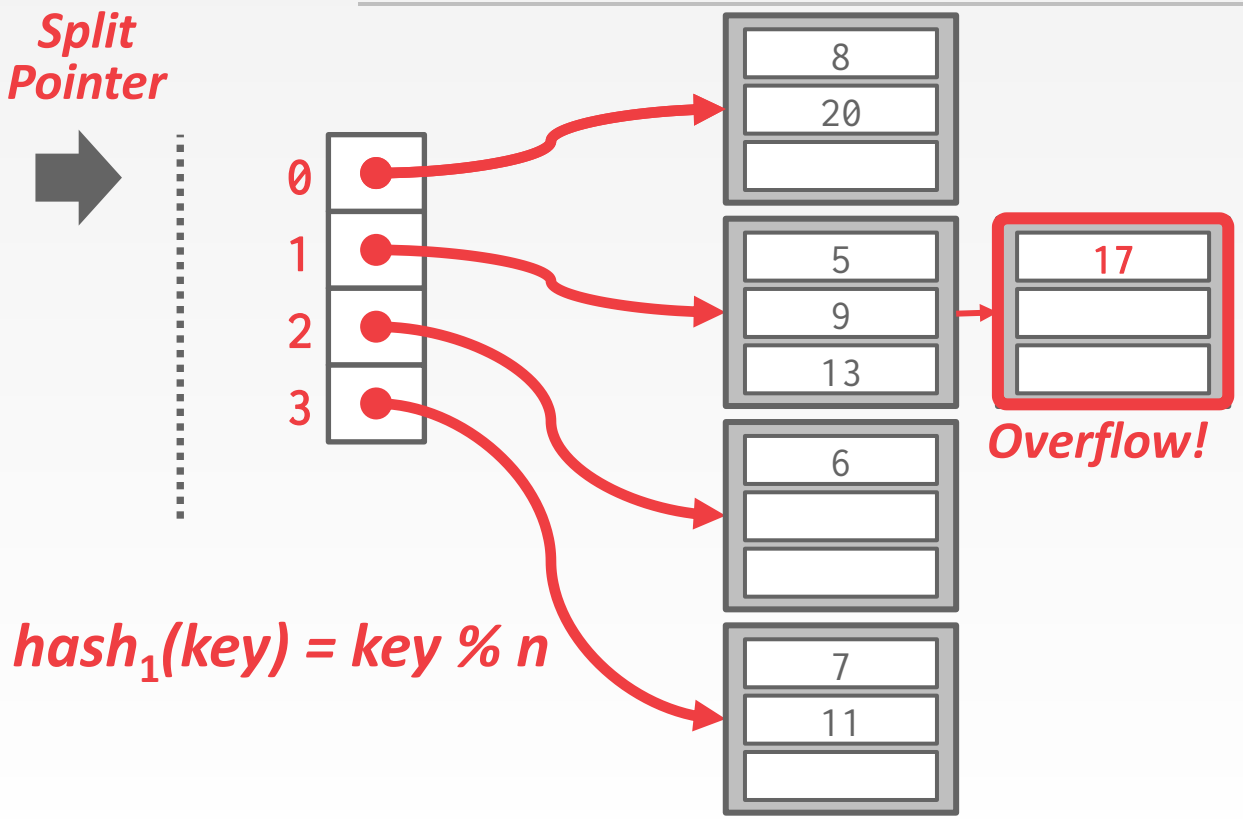
Find 6

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

Insert 17

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

LINEAR HASHING



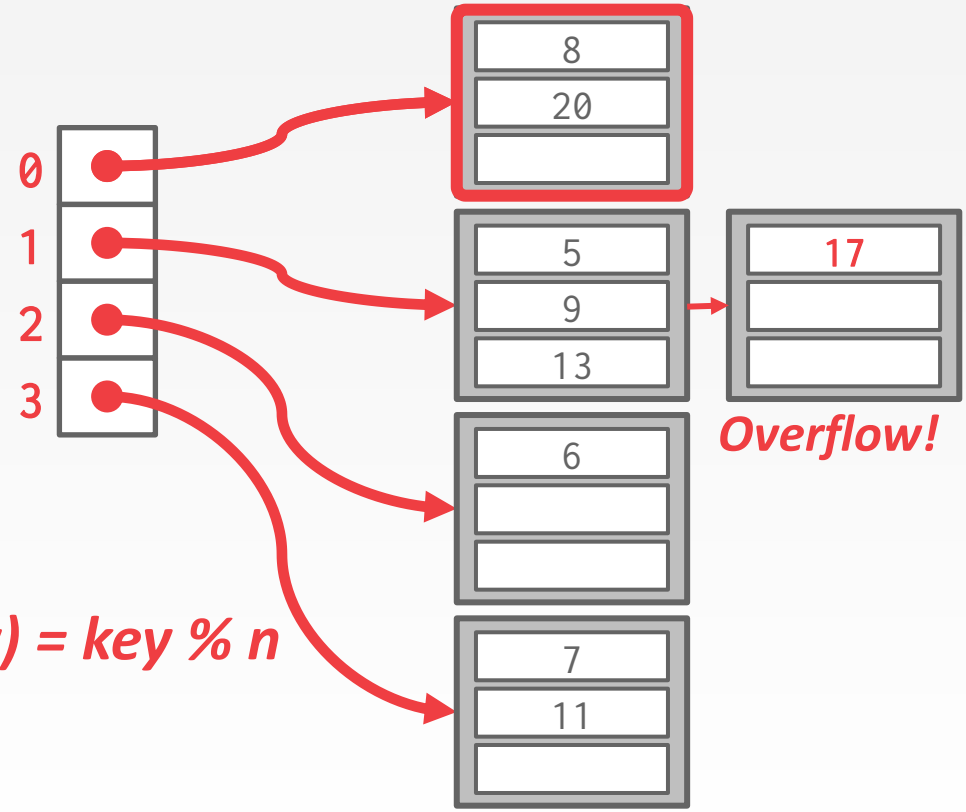
Find 6

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

Insert 17

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

LINEAR HASHING



$hash_1(key) = key \% n$

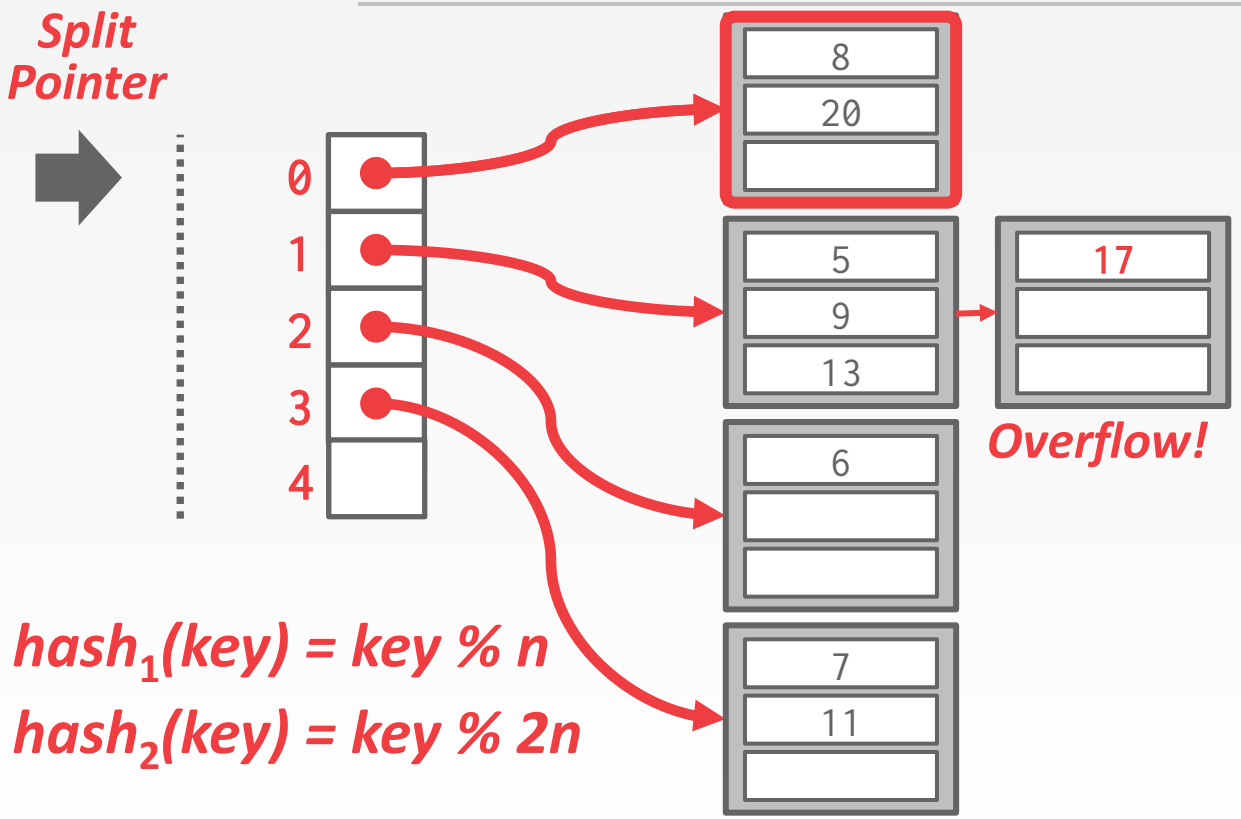
Find 6

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

Insert 17

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

LINEAR HASHING



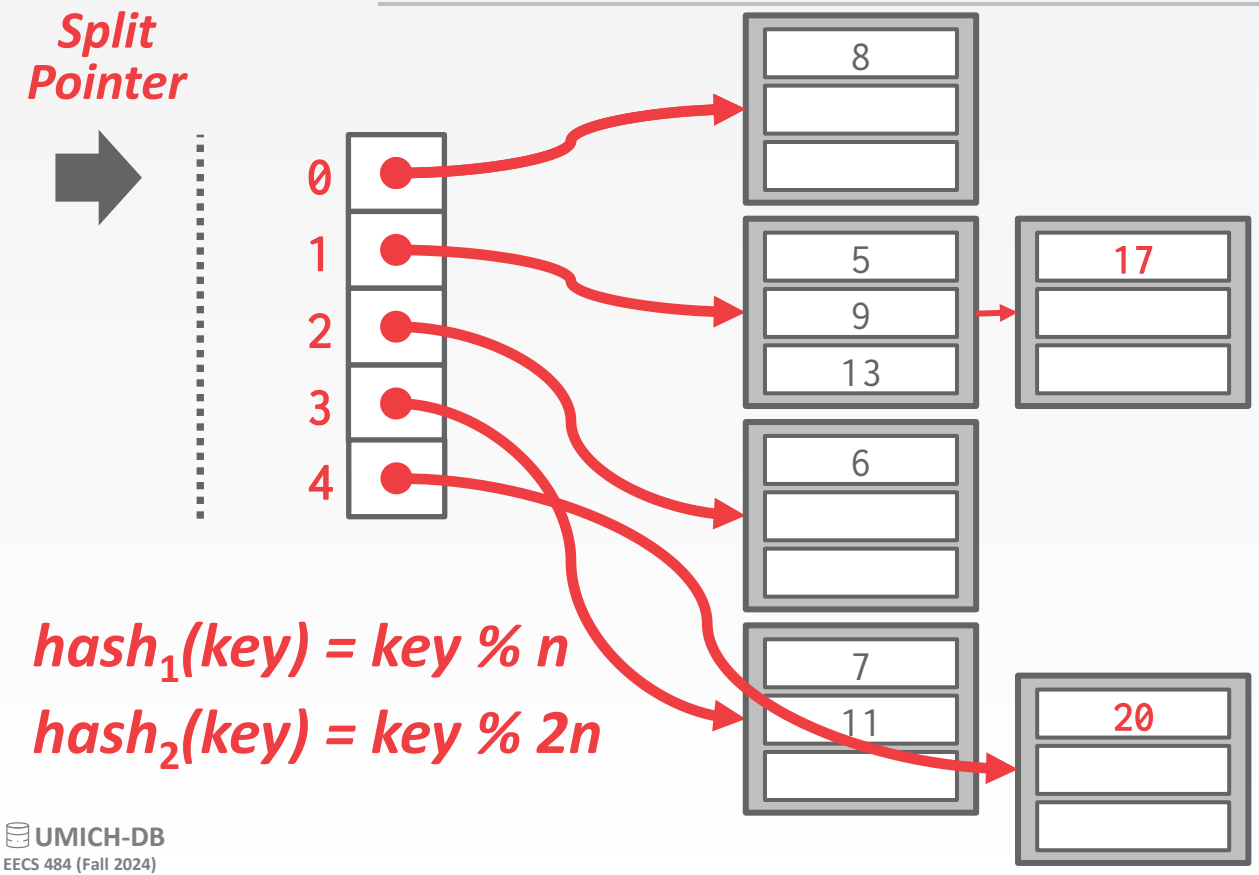
Find 6

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

Insert 17

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

LINEAR HASHING



$hash_1(key) = key \% n$

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

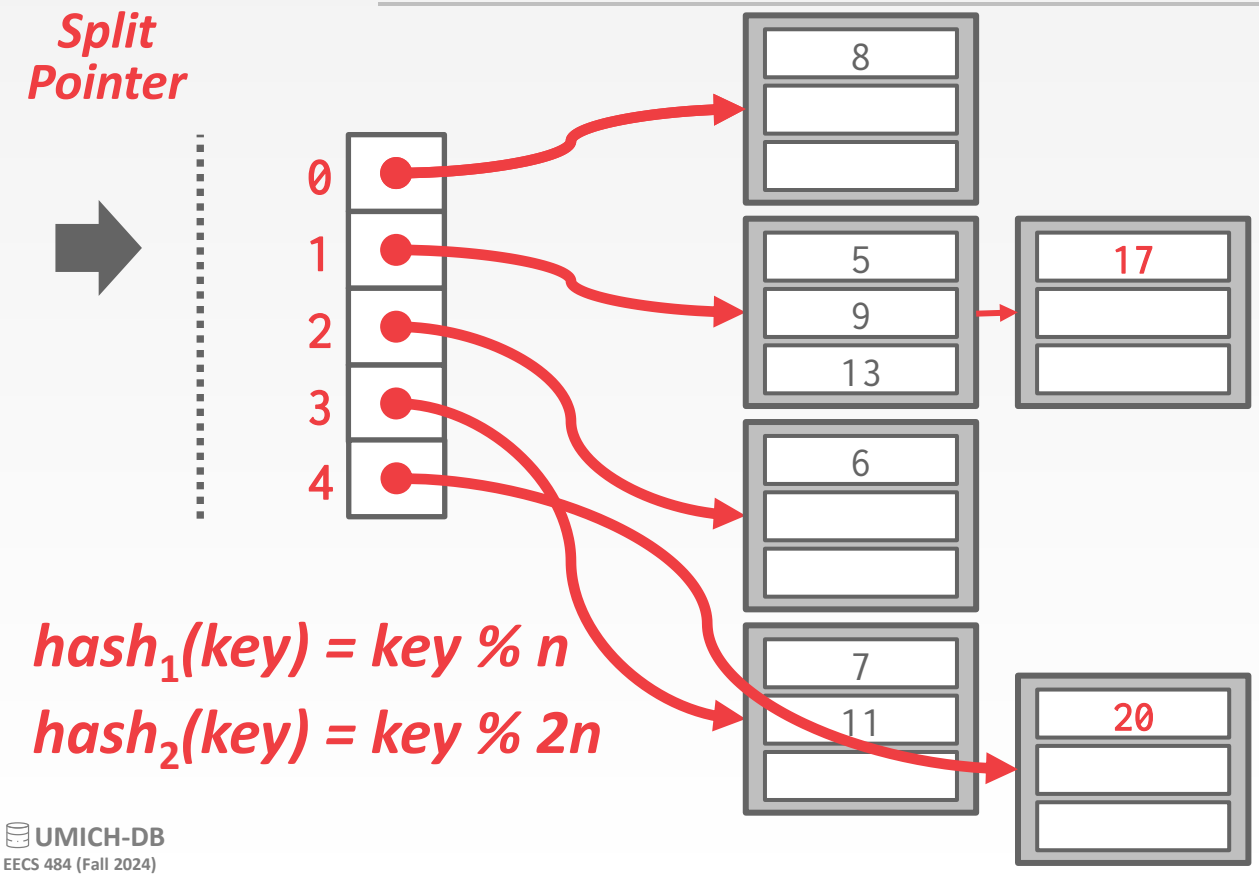
Find 6

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

Insert 17

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

LINEAR HASHING



$hash_1(key) = key \% n$

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

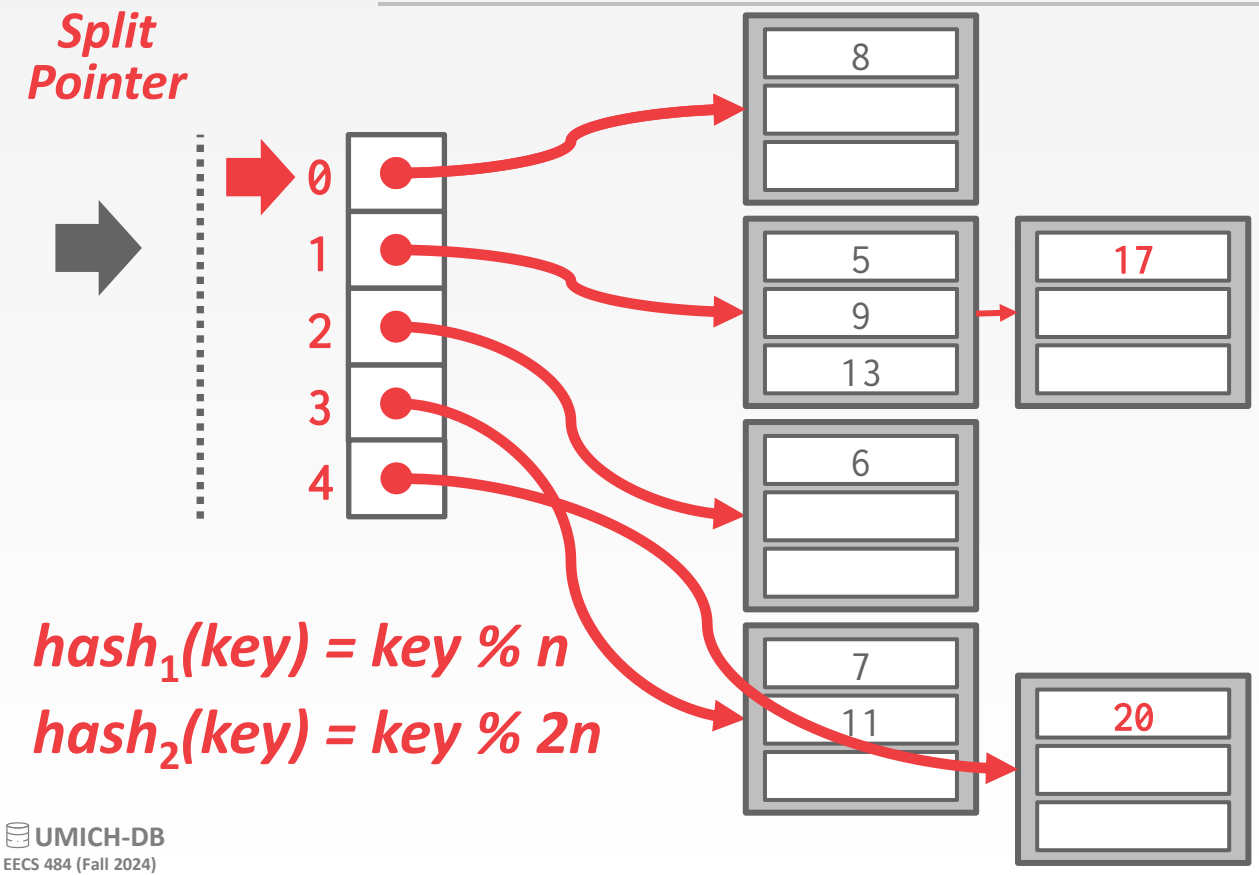
Find 6

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

Insert 17

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

LINEAR HASHING



Find 6

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

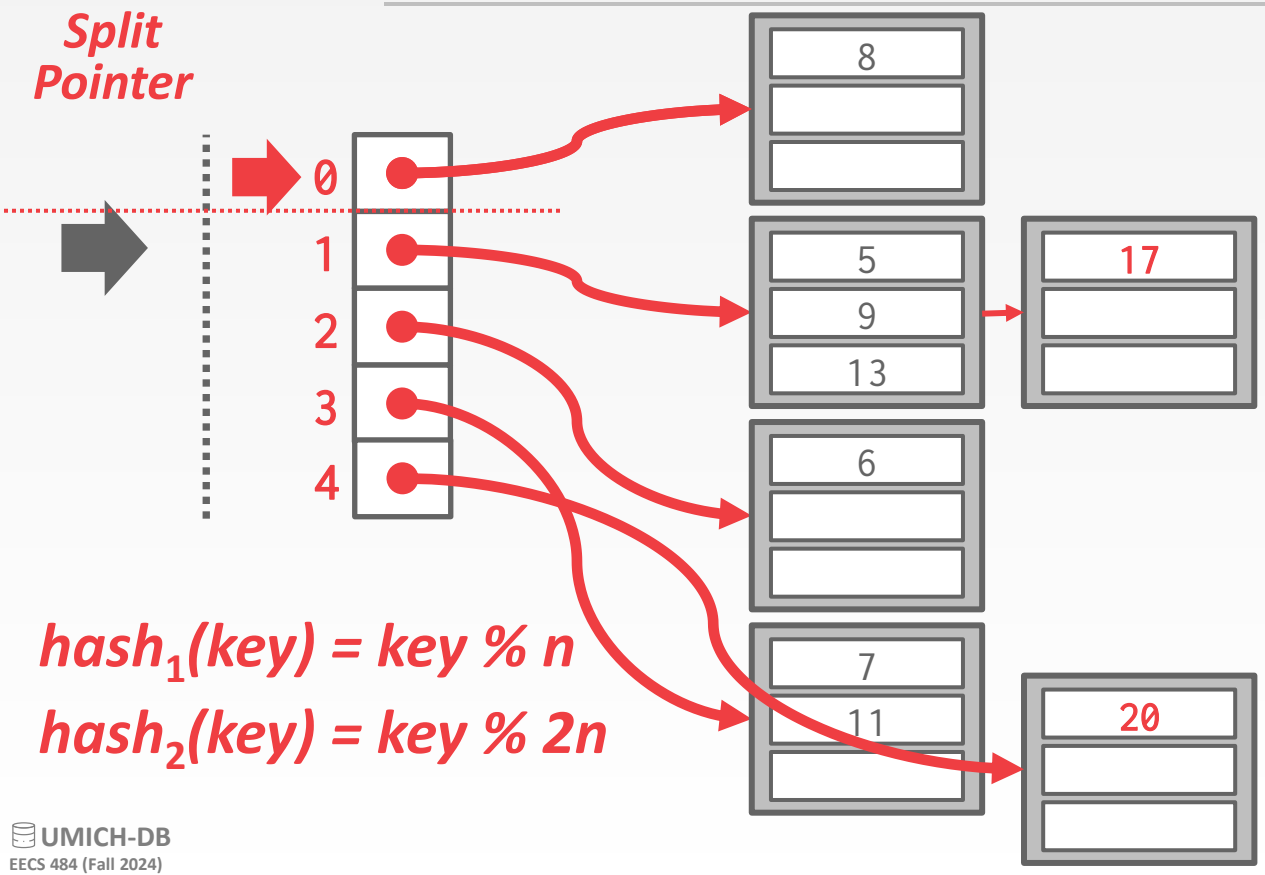
Insert 17

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

Find 20

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

LINEAR HASHING



$hash_1(key) = key \% n$

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

Find 6

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

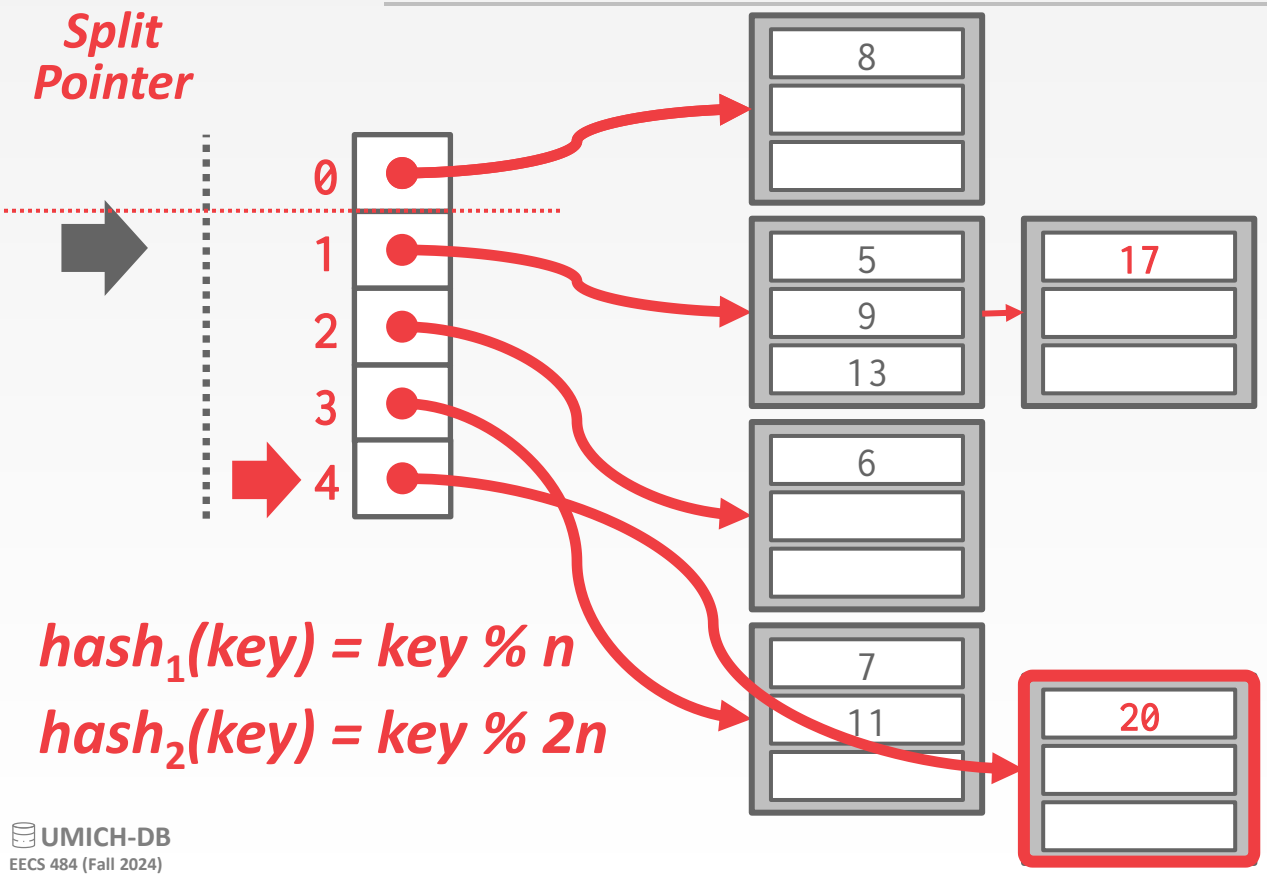
Insert 17

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

Find 20

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

LINEAR HASHING



Find 6

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

Insert 17

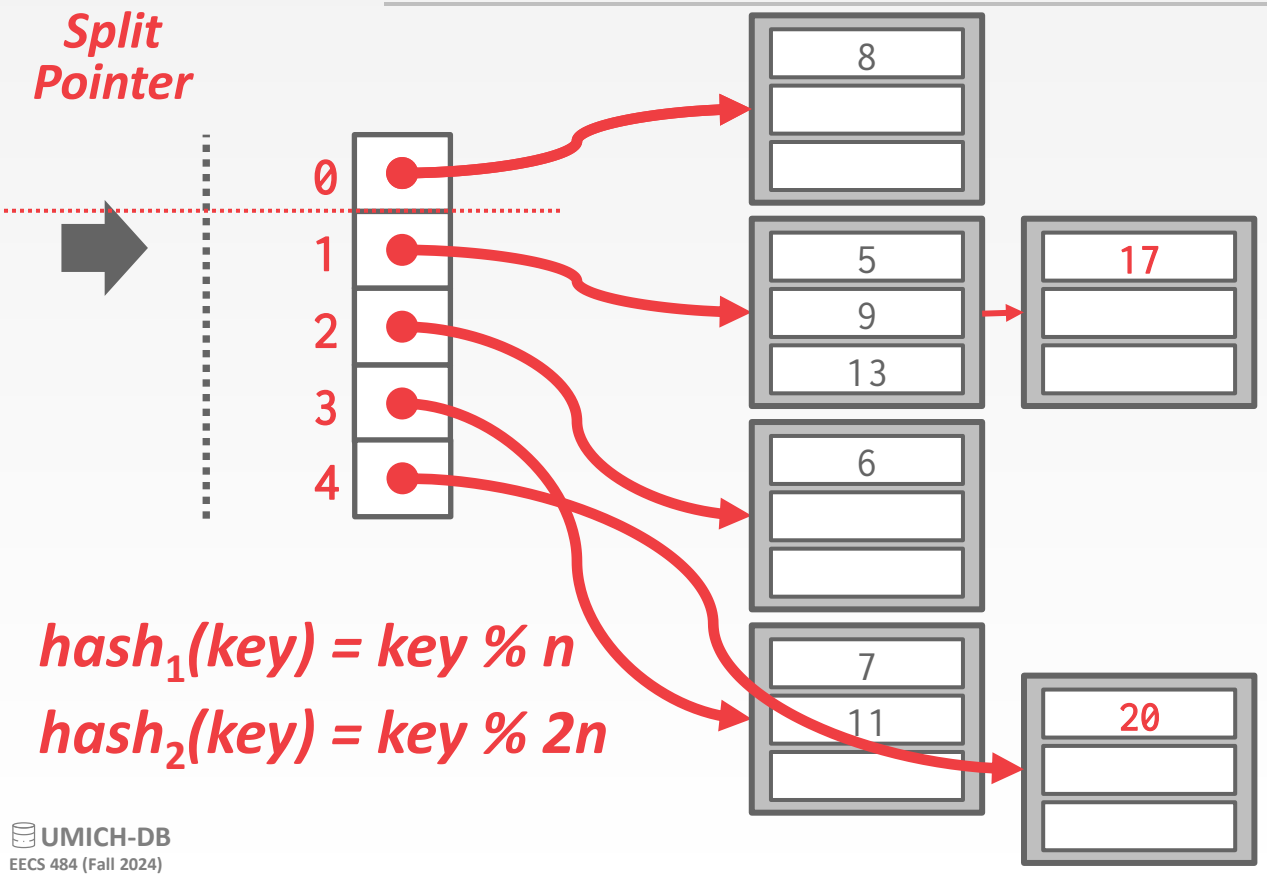
$hash_1(17) = 17 \% 4 = 1$

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

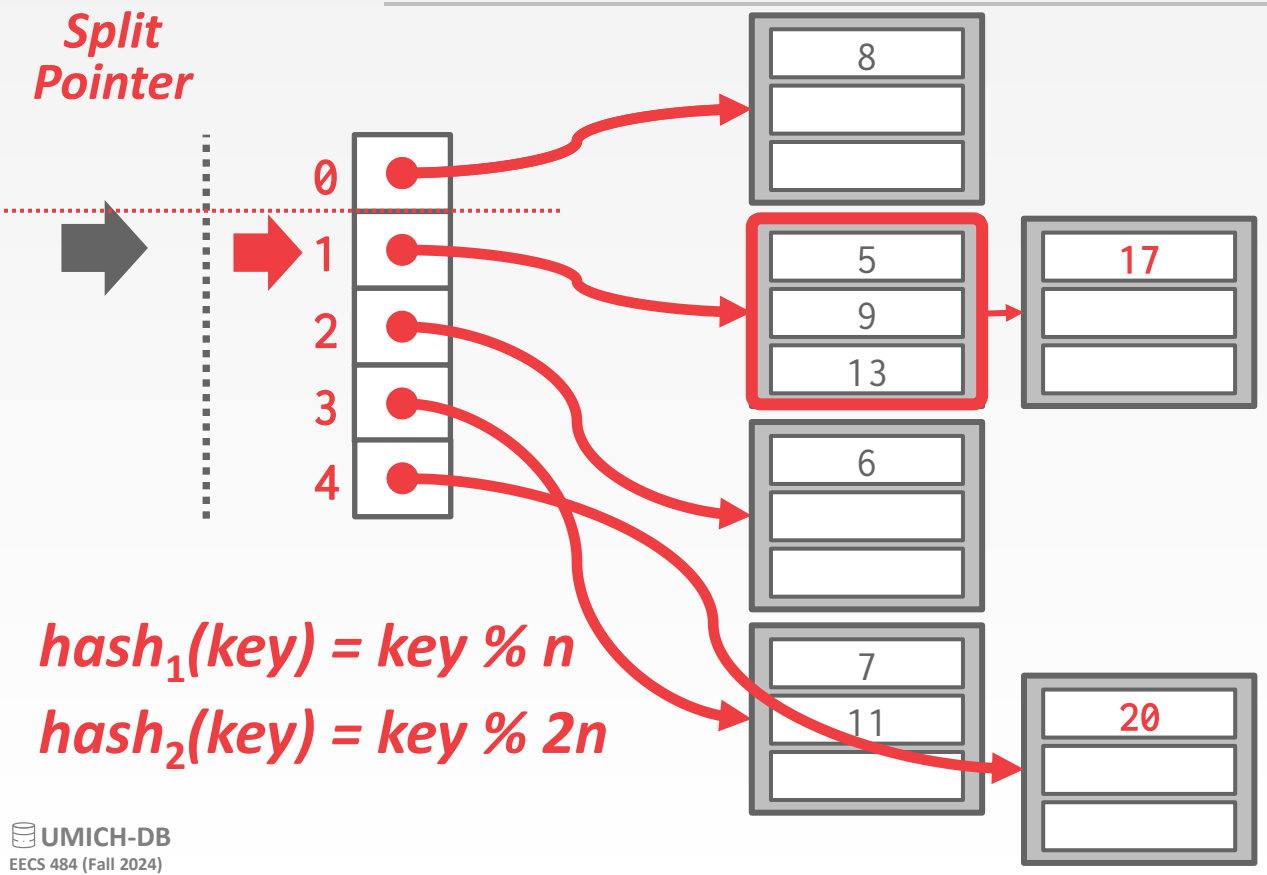
$hash_1(20) = 20 \% 4 = 0$

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

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.

CONCLUSION

Fast data structures that support **O(1)** look-ups that are used all throughout 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