Carnegie Mellon University

Database Systems

15-445/645 SPRING 2026
ANDY PAVLO
JIGNESH PATEL

Lecture #07

Static + Dynamic Hash Tables

ADMINISTRIVIA

Project #1 is due Sunday Feb 15th @ 11:59pm

- → Recitation Video + Slides (**@64**)
- → Perf Recitation on Wednesday Feb 4th @ 6:30pm (**@79**)
- → Special OH on Saturday Feb 14th @ 3:00-5:00pm (GHC 5207)

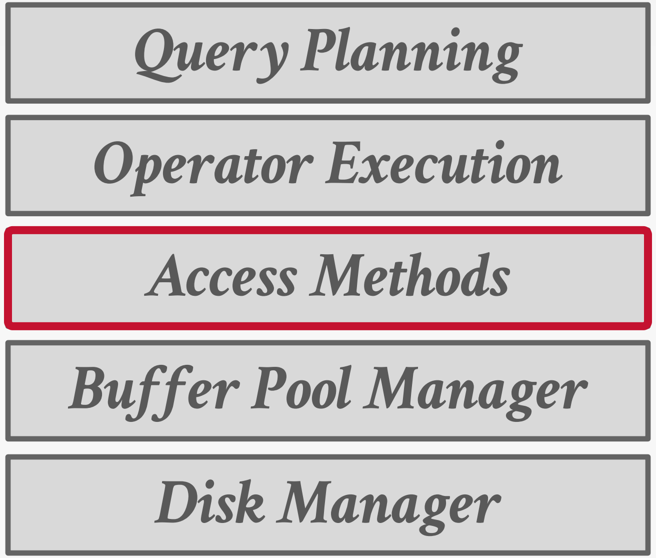
Homework #2 is due Sunday Feb 8th @ 11:59pm

COURSE OUTLINE

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 (Unordered)
→ Trees (Ordered)



Query Planning

Operator Execution

Access Methods

Buffer Pool Manager

Disk Manager

TODAY'S AGENDA

Background

Hash Functions

Static Hashing Schemes

Dynamic Hashing Schemes

⚡**DB Flash Talk: YugabyteDB**

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 this array for a given key, from which the desired value can be found.

Space Complexity: **O(n)**
Time Complexity:

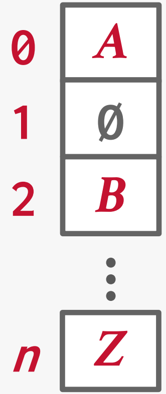
→ Average: **O(1)** ⬅ ***Databases care about constants!***
→ Worst: **O(n)**

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.

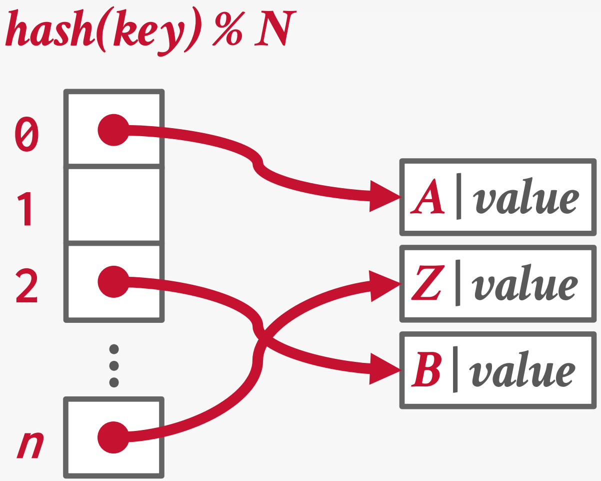
hash(key) % N



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.



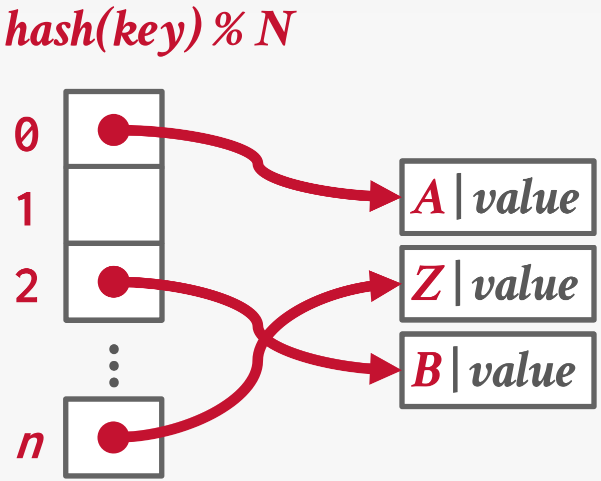
UNREALISTIC ASSUMPTIONS

Assumption #1: Number of elements is known ahead of time and fixed.

Assumption #2: Each key is unique.

Assumption #3: Perfect hash function guarantees no collisions.

→ 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 get/put keys.

HASH FUNCTIONS

For any input key, compute a one-way integer representation of that key (usually 32 or 64 bits).

→ Converts arbitrary byte array into a fixed-length code.

The only two properties of a hash function we care about in a DBMS is whether it is fast and has a low collision rate.

→ We do not want to use a cryptographic or reversible hash function for DBMS hash tables (e.g., SHA-2).

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.

← **State-of-the-art**

Google FarmHash (2014)

→ Newer version of CityHash with better collision rates.

RapidHash (2019)

→ Fast hash function without architecture-specific instructions.

STATIC HASHING SCHEMES

Approach #1: Linear Probe Hashing

Approach #2: Cuckoo Hashing

← **Open Addressing**

There are several other schemes covered in the Advanced DB course:

→ Robin Hood Hashing
→ Hopscotch Hashing
→ Swiss Tables
→ Concise Hash Tables

LINEAR PROBE HASHING

Single giant table of fixed-length slots.

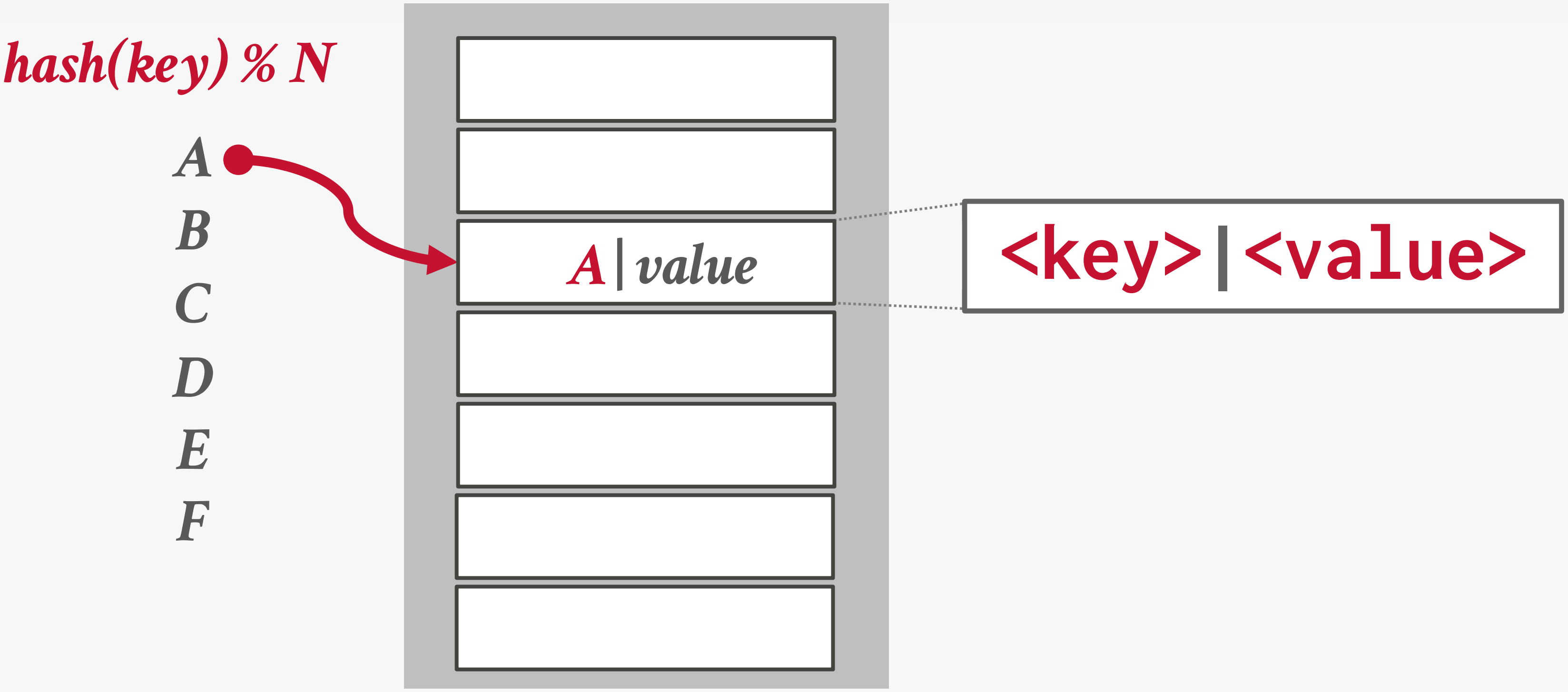
Resolve collisions by linearly searching for the next free slot in the table.

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

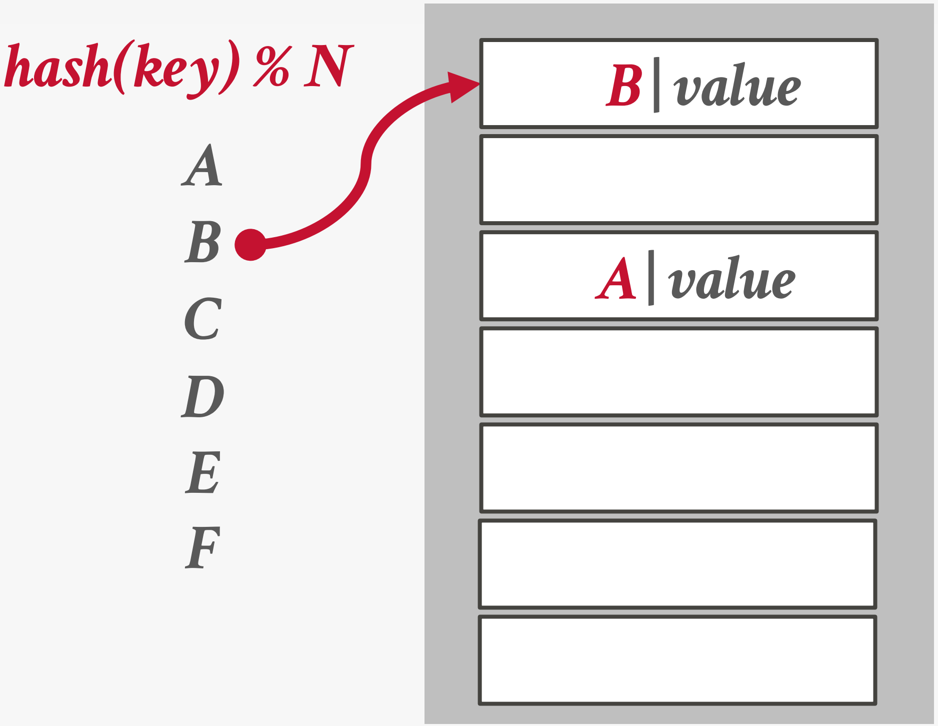
The table's **load factor** determines when it is becoming too full and should be resized.

→ Load Factor = Active Keys / # of Slots
→ Allocate a new table twice as large and rehash entries.

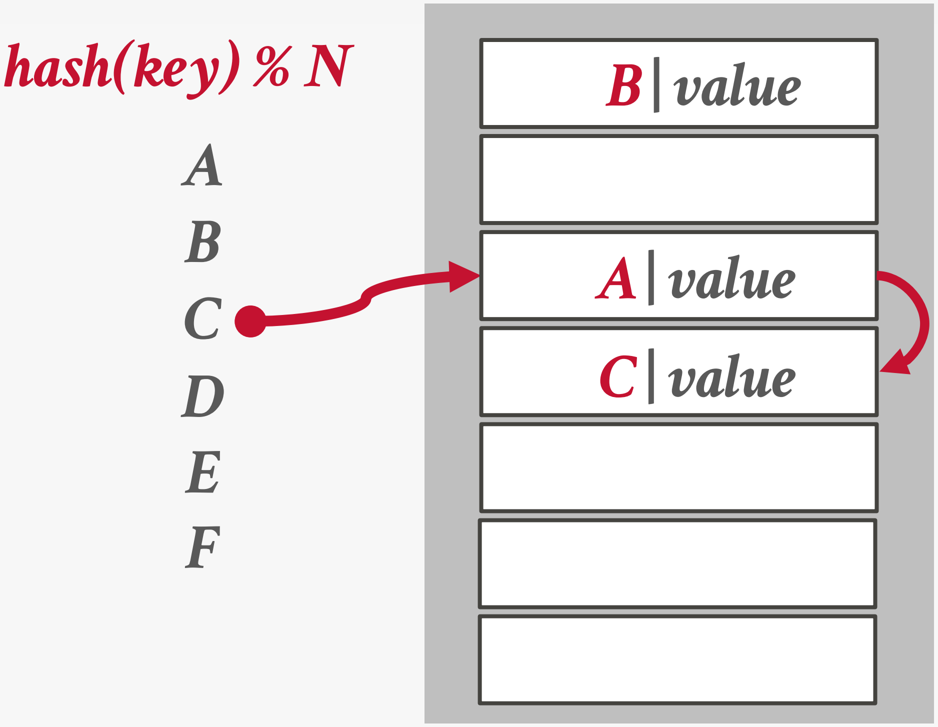
LINEAR PROBE HASHING



LINEAR PROBE HASHING

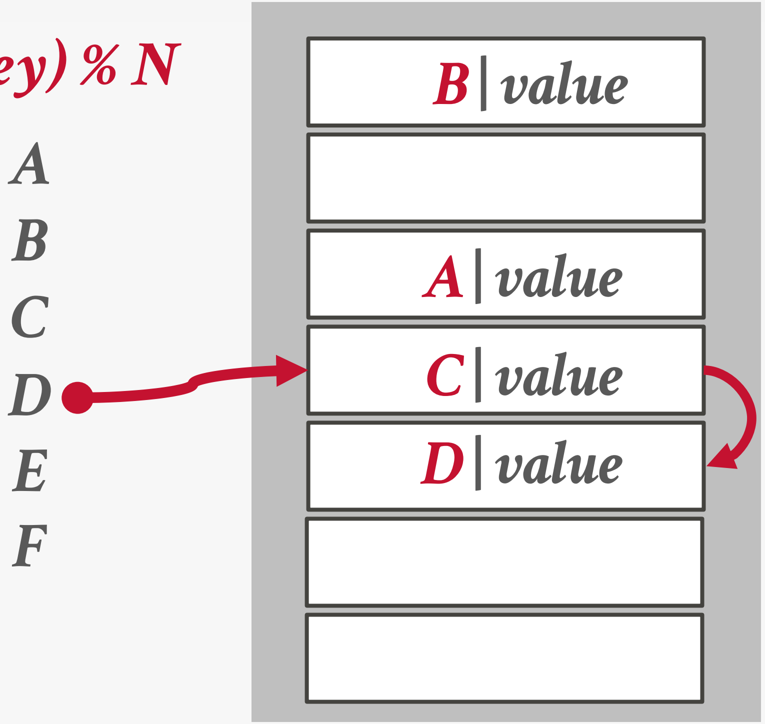


LINEAR PROBE HASHING



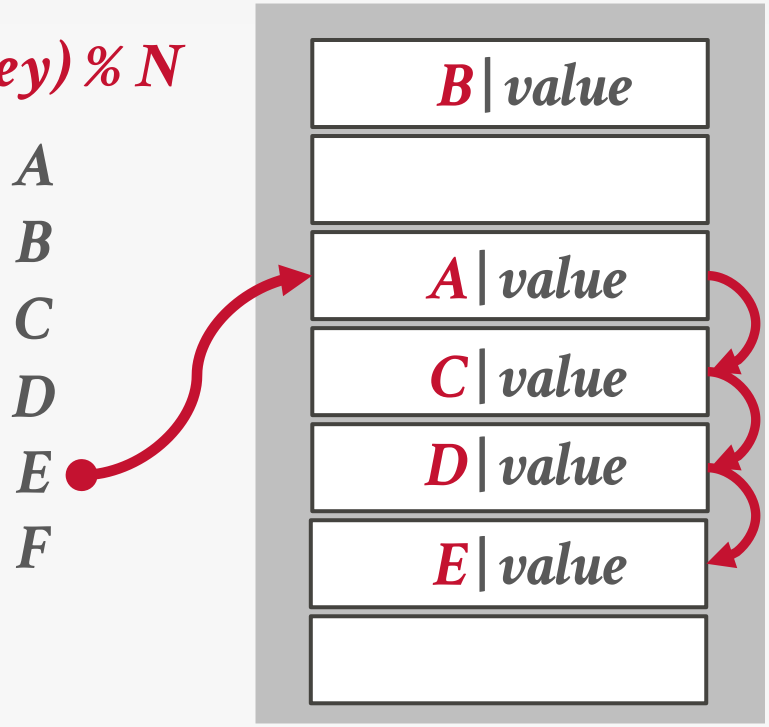
LINEAR PROBE HASHING

hash(key) % N

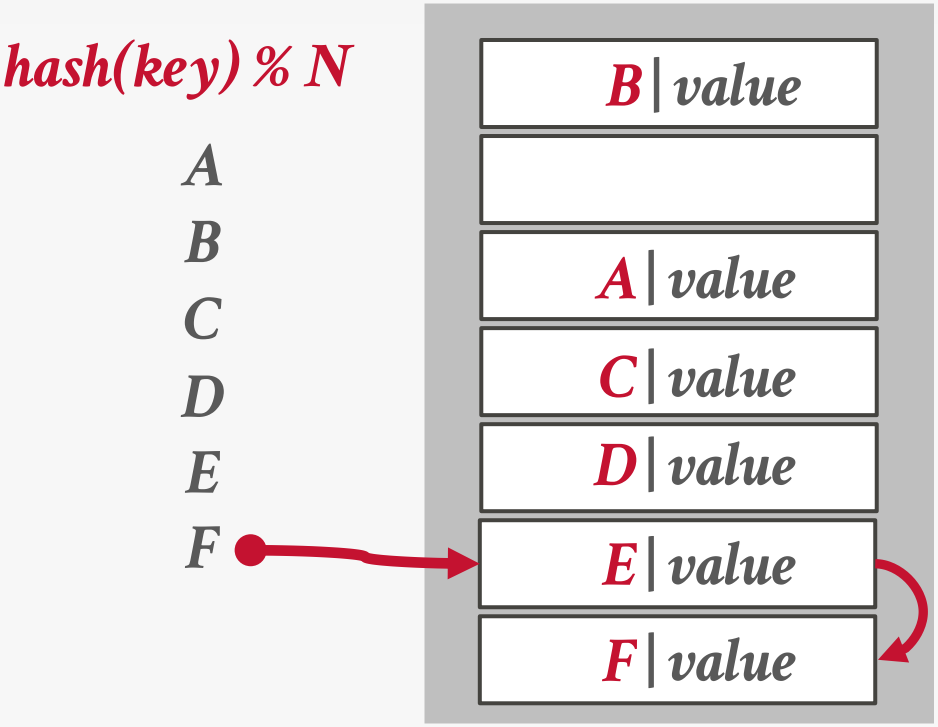


LINEAR PROBE HASHING

hash(key) % N



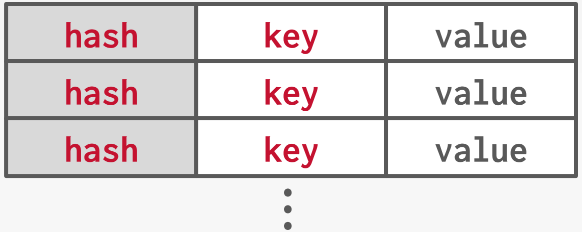
LINEAR PROBE HASHING



HASH TABLE: KEY/VALUE ENTRIES

Fixed-length Key/Values:

- → Store inline within the hash table pages.
- → Optional: Store the key's hash with the key for faster comparisons.

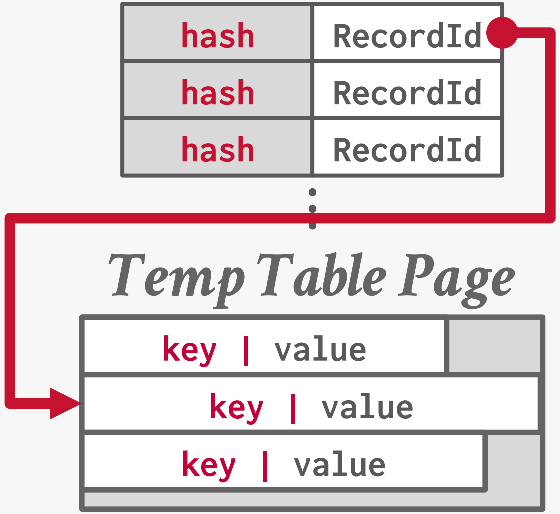


hash	key	value
hash	key	value
hash	key	value

⋮

Variable-length Key/Values:

- → Insert key/value data in separate a private temporary table.
- → Store the hash as the key and use the record id pointing to its corresponding entry in the temporary table as the value.



hash	RecordId
hash	RecordId
hash	RecordId

⋮

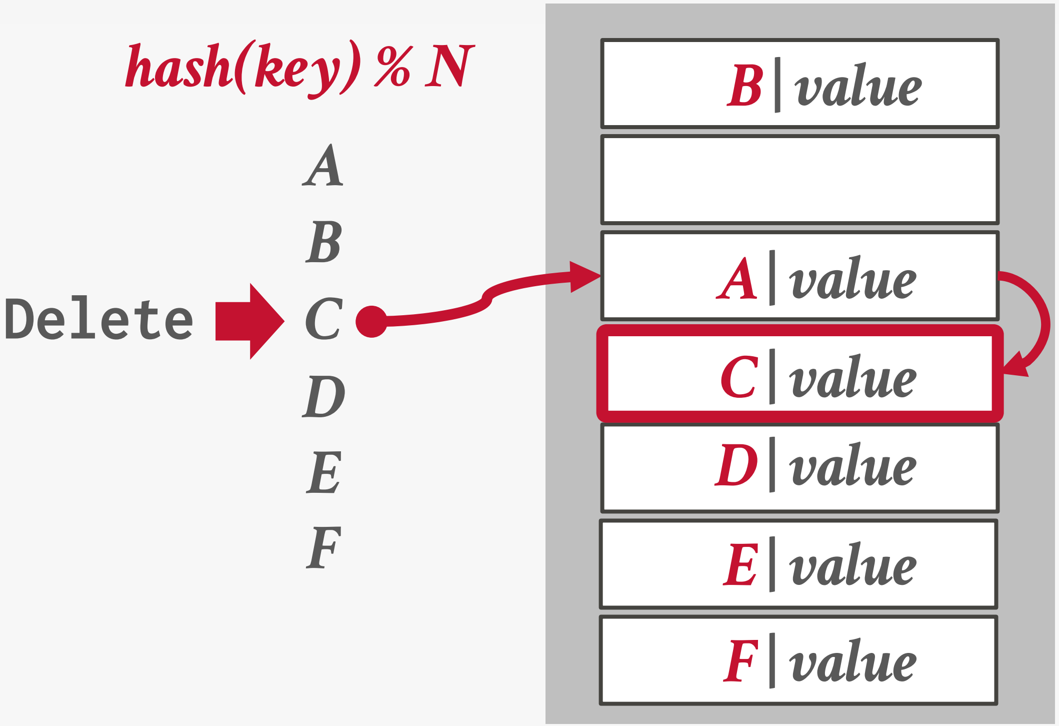
Temp Table Page

key | value

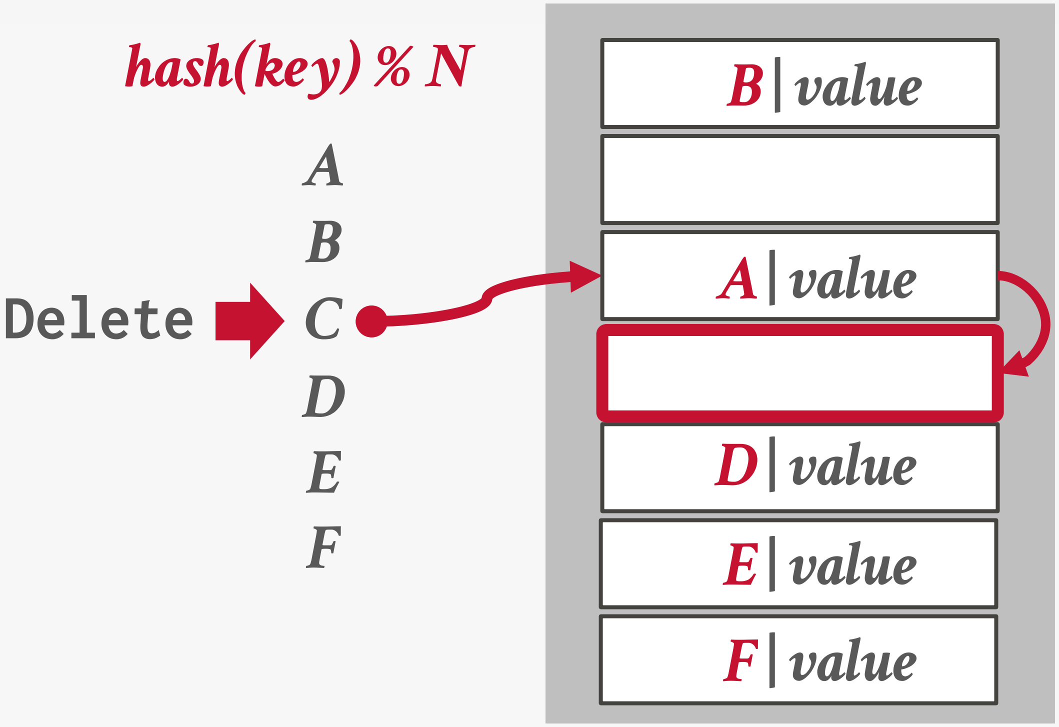
key | value

key | value

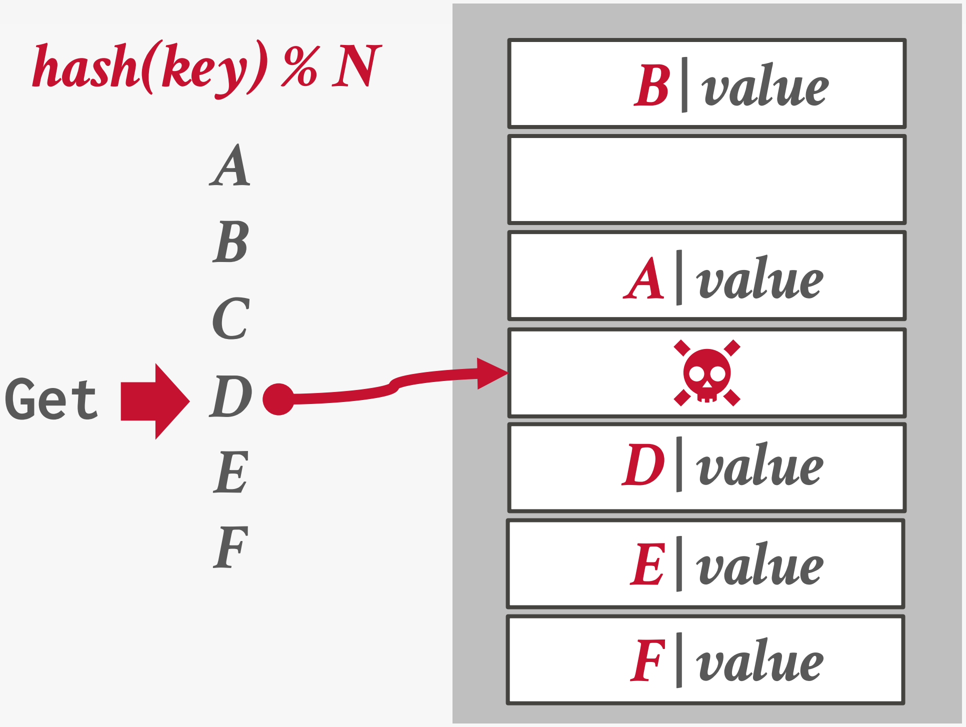
LINEAR PROBE HASHING: DELETES



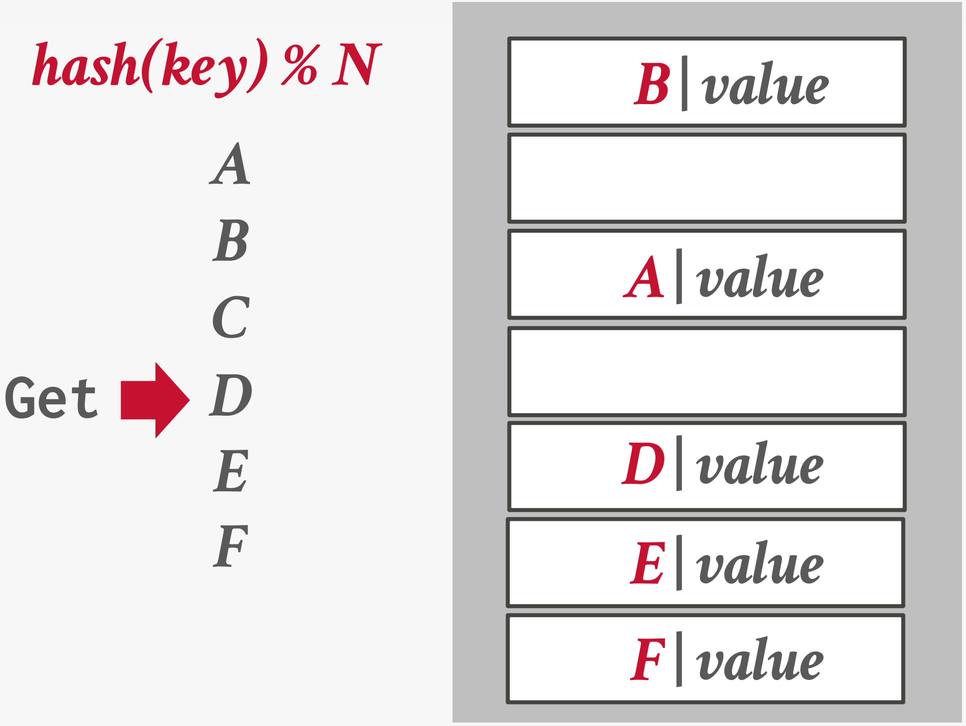
LINEAR PROBE HASHING: DELETES



LINEAR PROBE HASHING: DELETES



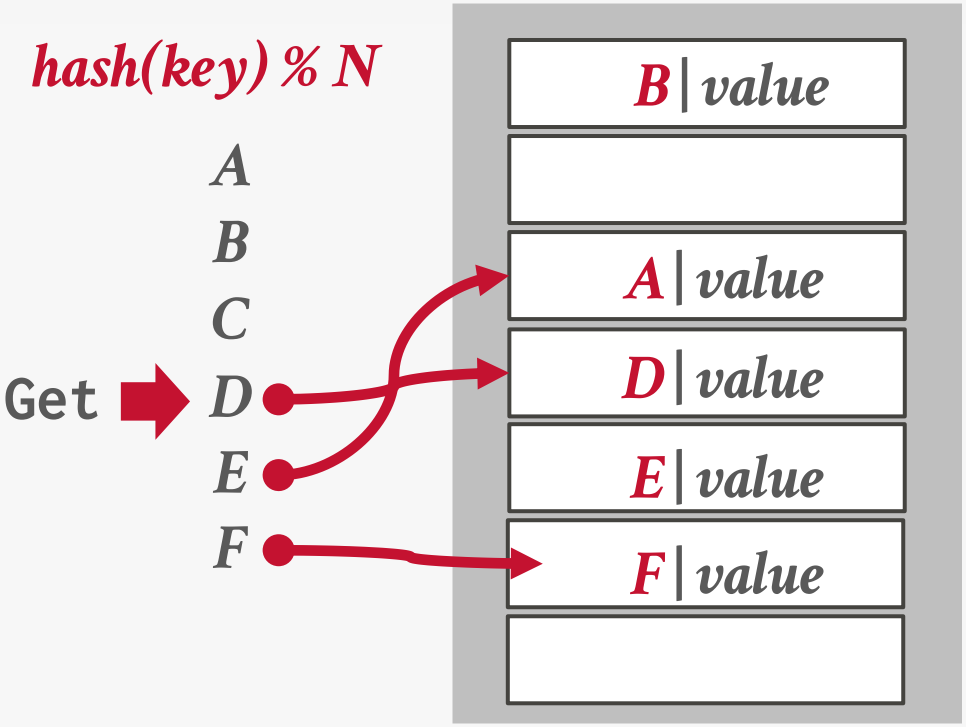
LINEAR PROBE HASHING: DELETES



Approach #1: Movement

→ Rehash keys until you find the first empty slot.

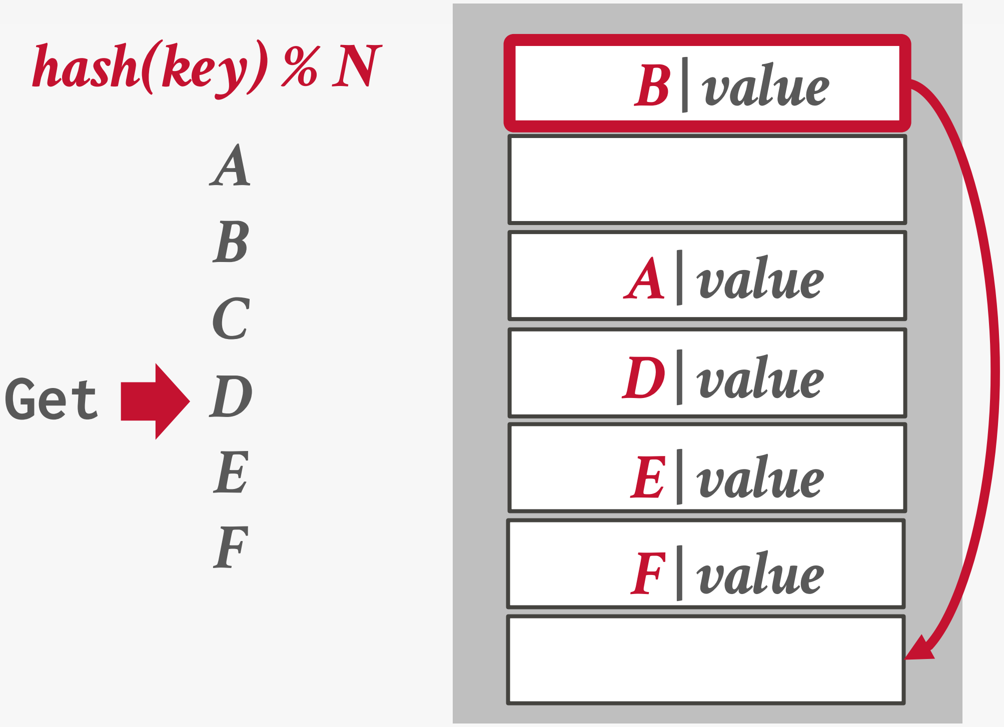
LINEAR PROBE HASHING: DELETES



Approach #1: Movement

→ Rehash keys until you find the first empty slot.

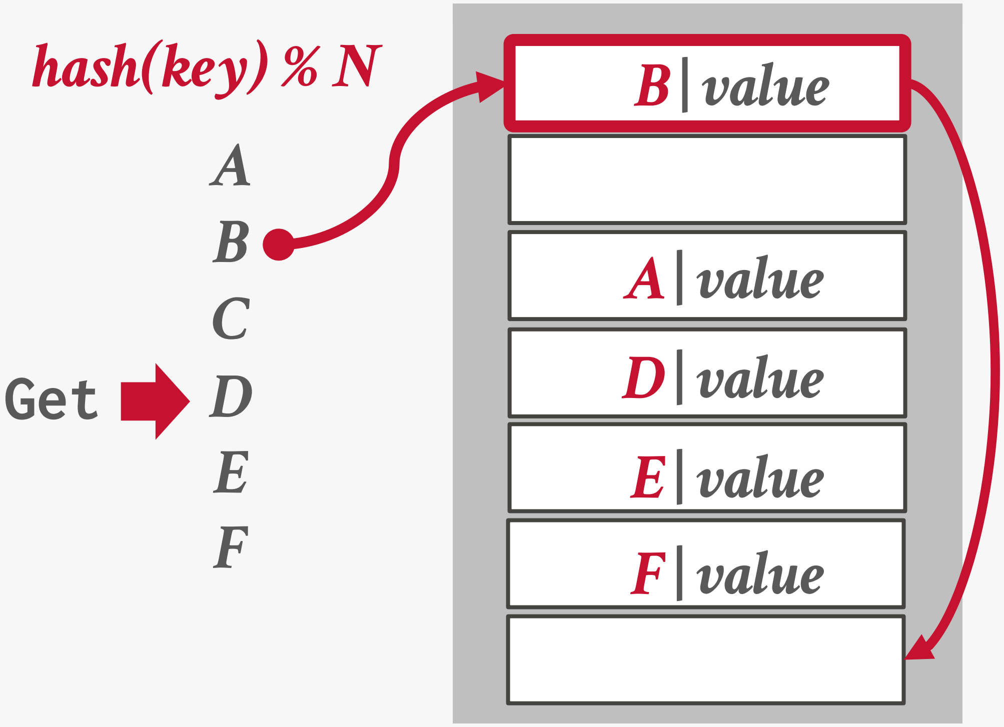
LINEAR PROBE HASHING: DELETES



Approach #1: Movement

→ Rehash keys until you find the first empty slot.

LINEAR PROBE HASHING: DELETES



Approach #1: Movement

→ Rehash keys until you find the first empty slot.

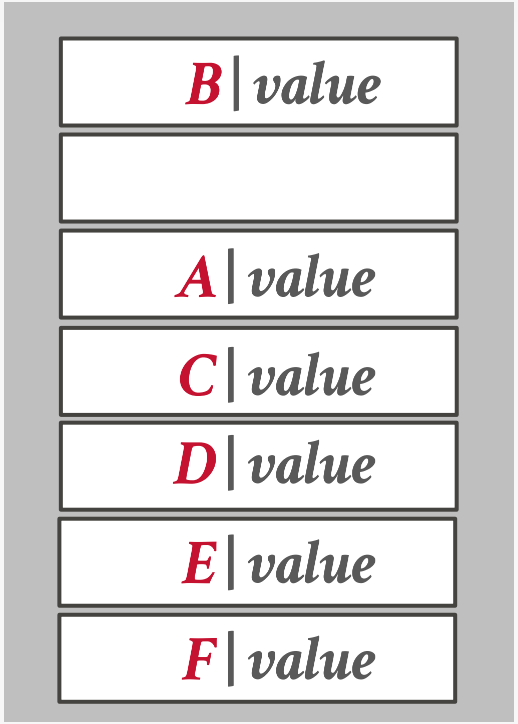
→ No DBMS does this.

→ to reorganize the entire table.

LINEAR PROBE HASHING: DELETES

hash(key) % N

A
B
C
D
E
F



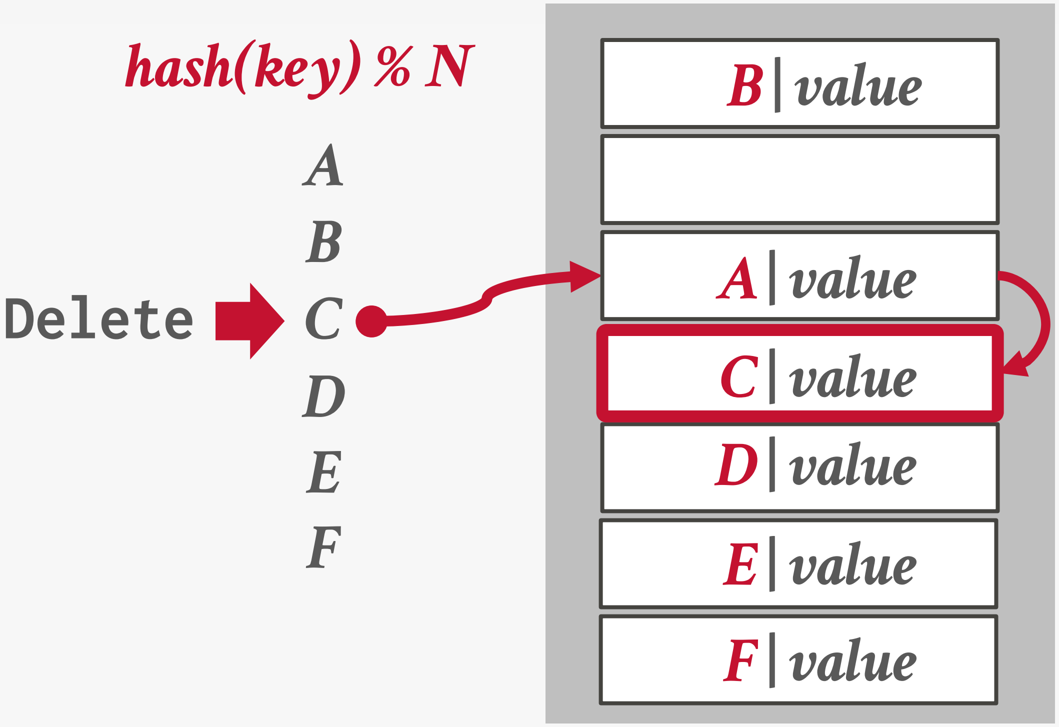
Approach #2: Tombstone

→ Maintain separate bit map to indicate that the entry in the slot is logically deleted.

→ Reuse the slot for new keys.

→ May need periodic garbage collection.

LINEAR PROBE HASHING: DELETES



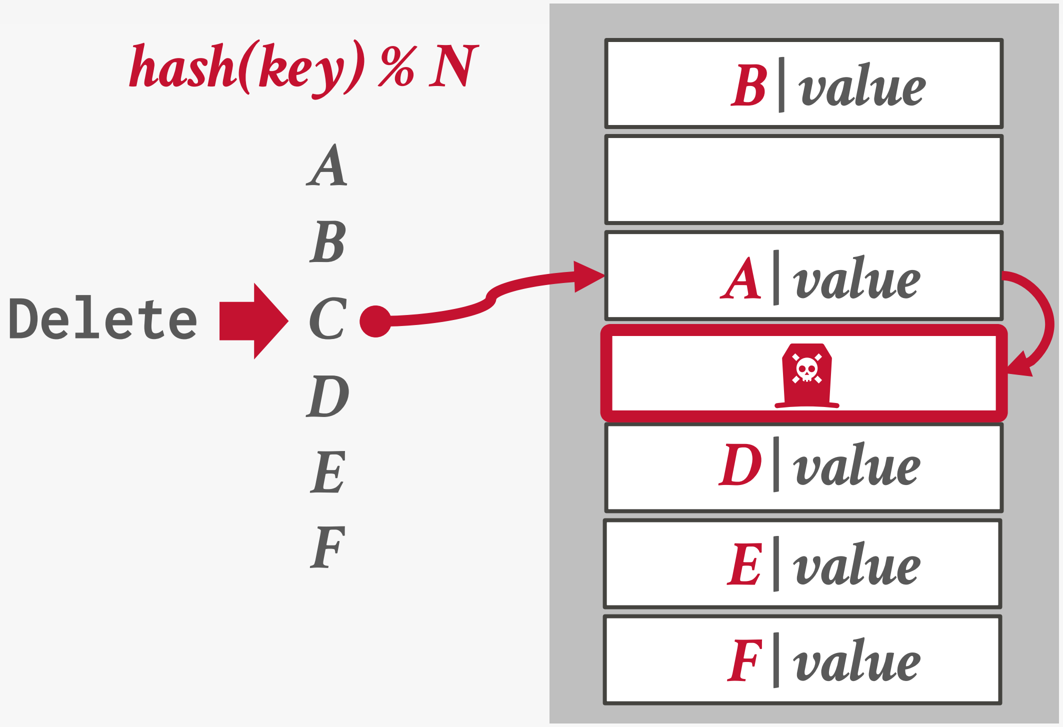
Approach #2: Tombstone

→ Maintain separate bit map to indicate that the entry in the slot is logically deleted.

→ Reuse the slot for new keys.

→ May need periodic garbage collection.

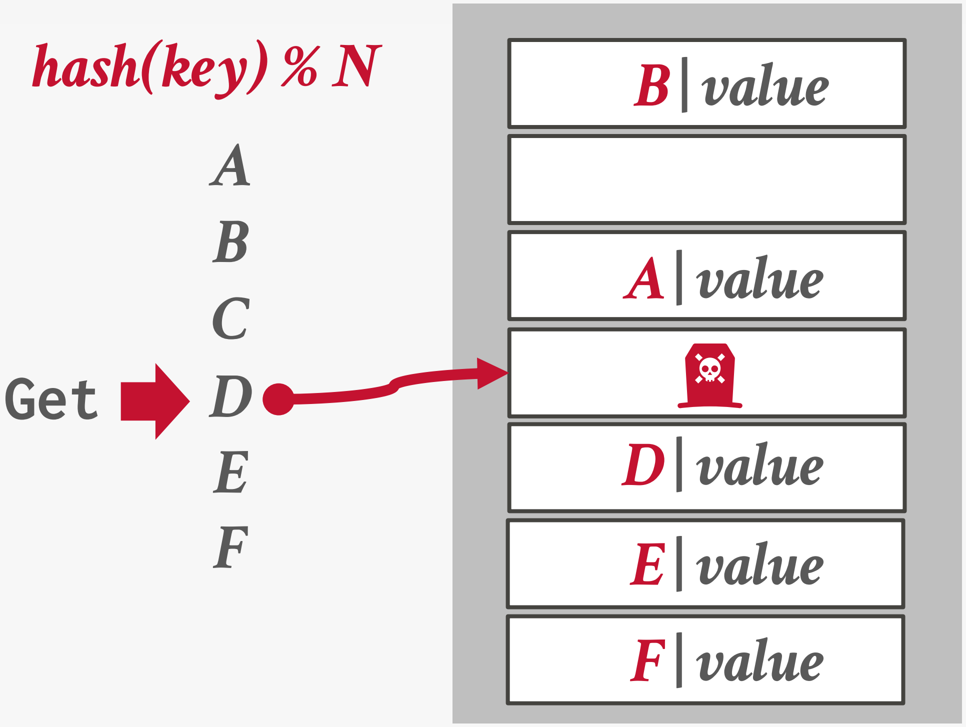
LINEAR PROBE HASHING: DELETES



Approach #2: Tombstone

- → Maintain separate bit map to indicate that the entry in the slot is logically deleted.
- → Reuse the slot for new keys.
- → May need periodic garbage collection.

LINEAR PROBE HASHING: DELETES



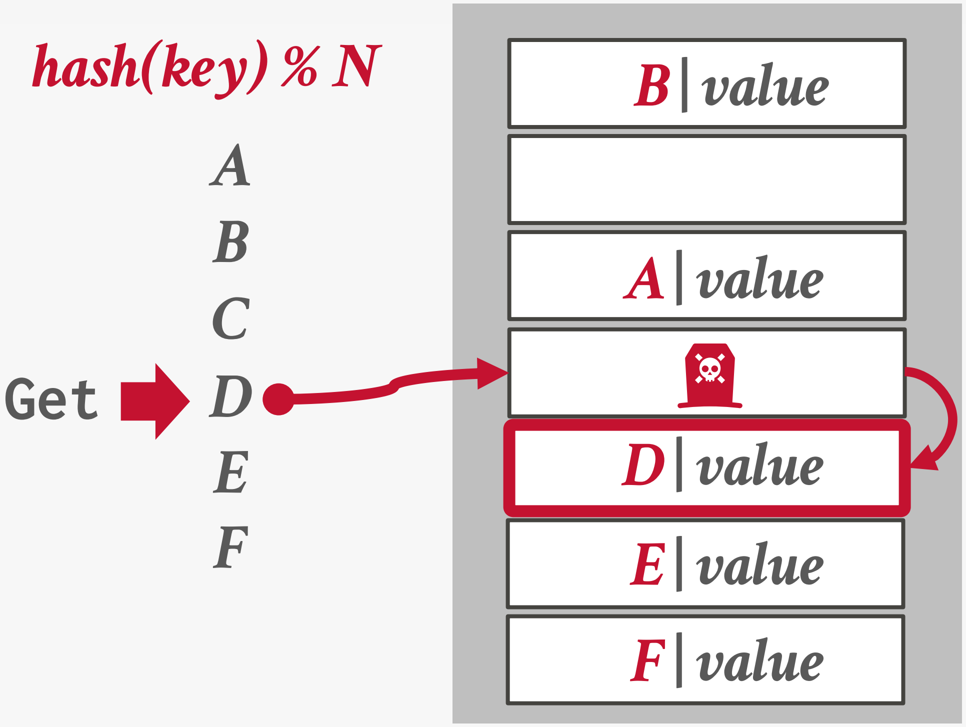
Approach #2: Tombstone

→ Maintain separate bit map to indicate that the entry in the slot is logically deleted.

→ Reuse the slot for new keys.

→ May need periodic garbage collection.

LINEAR PROBE HASHING: DELETES



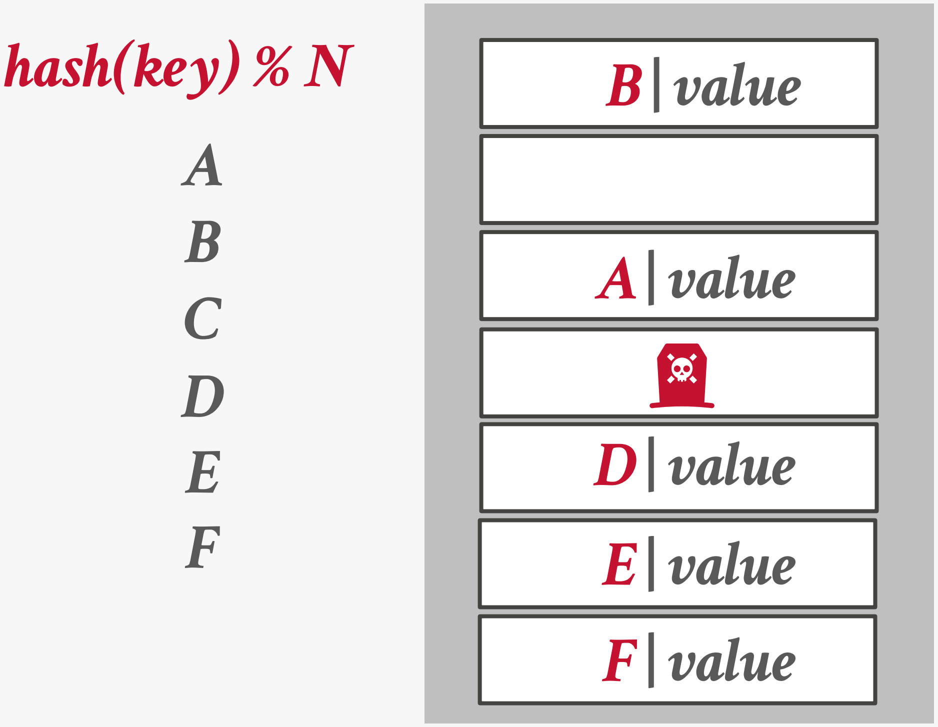
Approach #2: Tombstone

→ Maintain separate bit map to indicate that the entry in the slot is logically deleted.

→ Reuse the slot for new keys.

→ May need periodic garbage collection.

LINEAR PROBE HASHING: DELETES



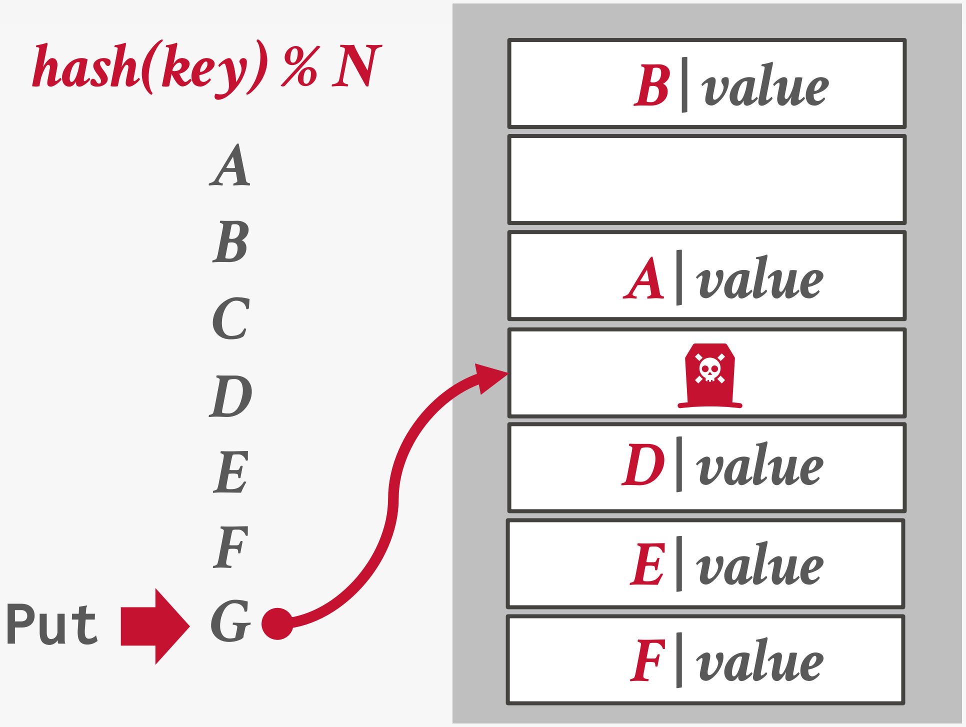
Approach #2: Tombstone

→ Maintain separate bit map to indicate that the entry in the slot is logically deleted.

→ Reuse the slot for new keys.

→ May need periodic garbage collection.

LINEAR PROBE HASHING: DELETES



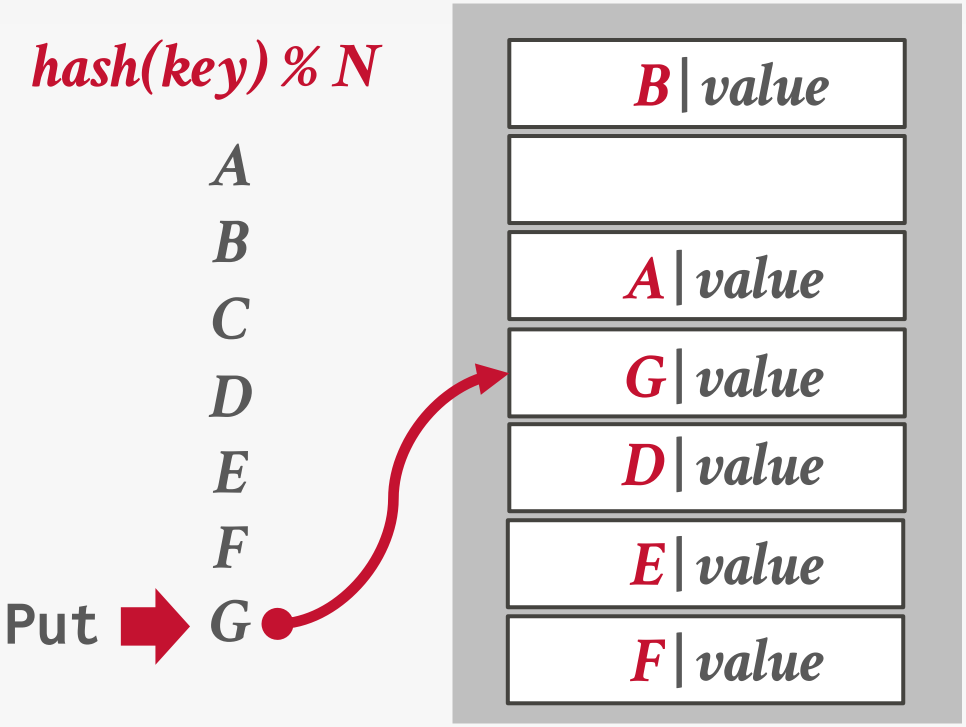
Approach #2: Tombstone

→ Maintain separate bit map to indicate that the entry in the slot is logically deleted.

→ Reuse the slot for new keys.

→ May need periodic garbage collection.

LINEAR PROBE HASHING: DELETES



Approach #2: Tombstone

→ Maintain separate bit map to indicate that the entry in the slot is logically deleted.

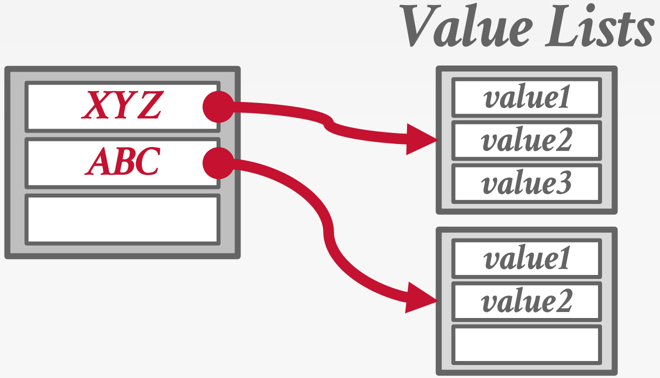
→ Reuse the slot for new keys.

→ May need periodic garbage collection.

HASH TABLE: NON-UNIQUE KEYS

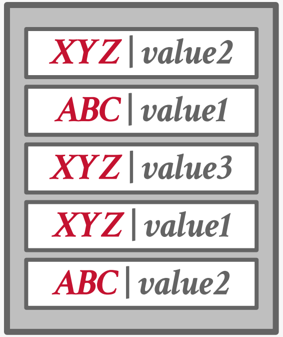
Choice #1: Separate Linked List

- → Store values in separate storage area for each key.
- → Value lists can overflow to multiple pages if the number of duplicates is large.



Choice #2: Redundant Keys

- → Store duplicate keys entries together in the hash table.
- → This is what most systems do.



OPTIMIZATIONS

Specialized impls. based on key type(s) and sizes.

→ Example: Maintain multiple hash tables for different string sizes

Store metadata separate in a separate array.

→ Use separate offset array (sparse) that points to entries in a data segment (dense).

→ Packed bitmap to track whether a slot is empty/tombstone.

Use table + slot versioning metadata to quickly invalidate all entries in the hash table.

→ Example: If table version does not match slot version, then treat the slot as empty.

Source: Maksim Kita

OPTIMIZATIONS

Specialized impls. based on key t

→ Example: Maintain multiple hash ta

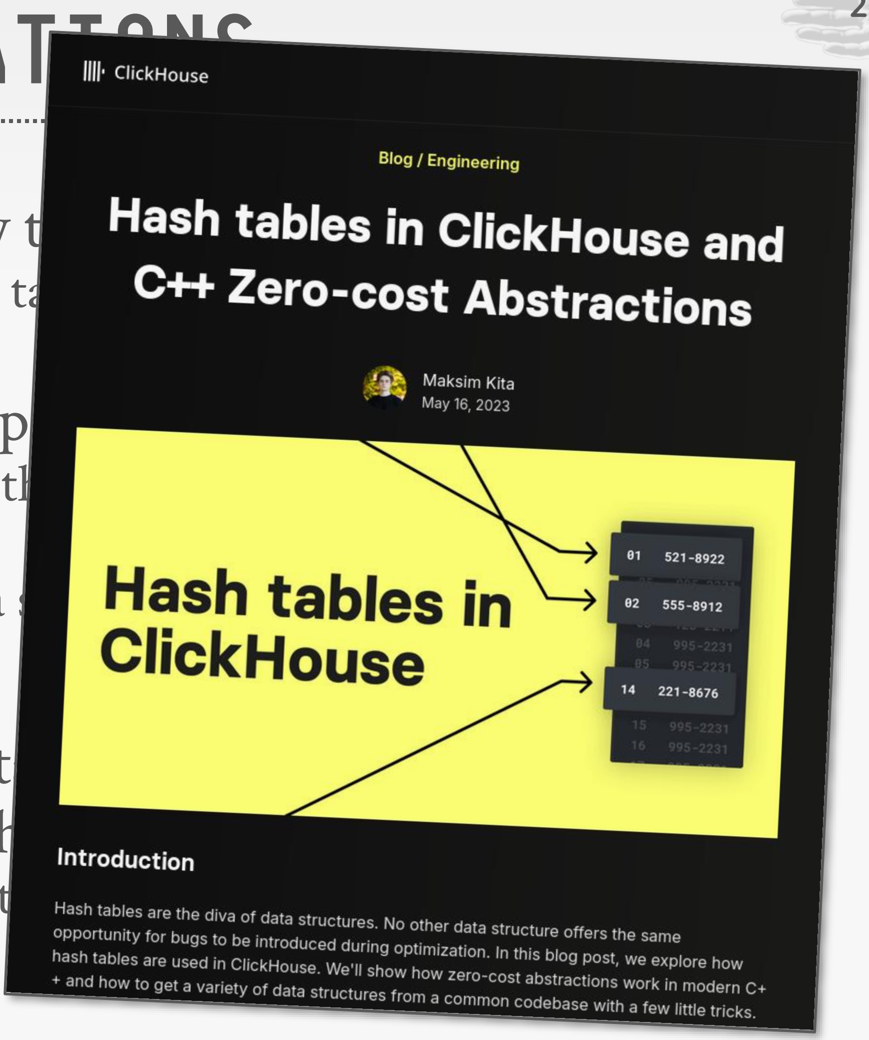
Store metadata separate in a sep

→ Use separate offset array (sparse) th segment (dense).

→ Packed bitmap to track whether a s

Use table + slot versioning met invalidate all entries in the hash

→ Example: If table version does not the slot as empty.



ClickHouse

Blog / Engineering

Hash tables in ClickHouse and C++ Zero-cost Abstractions

Maksim Kita
May 16, 2023

Introduction

Hash tables are the diva of data structures. No other data structure offers the same opportunity for bugs to be introduced during optimization. In this blog post, we explore how hash tables are used in ClickHouse. We'll show how zero-cost abstractions work in modern C++ and how to get a variety of data structures from a common codebase with a few little tricks.

Source: Maksim Kita

CUCKOO HASHING

Use multiple hash functions to find multiple locations in the hash table to insert records.

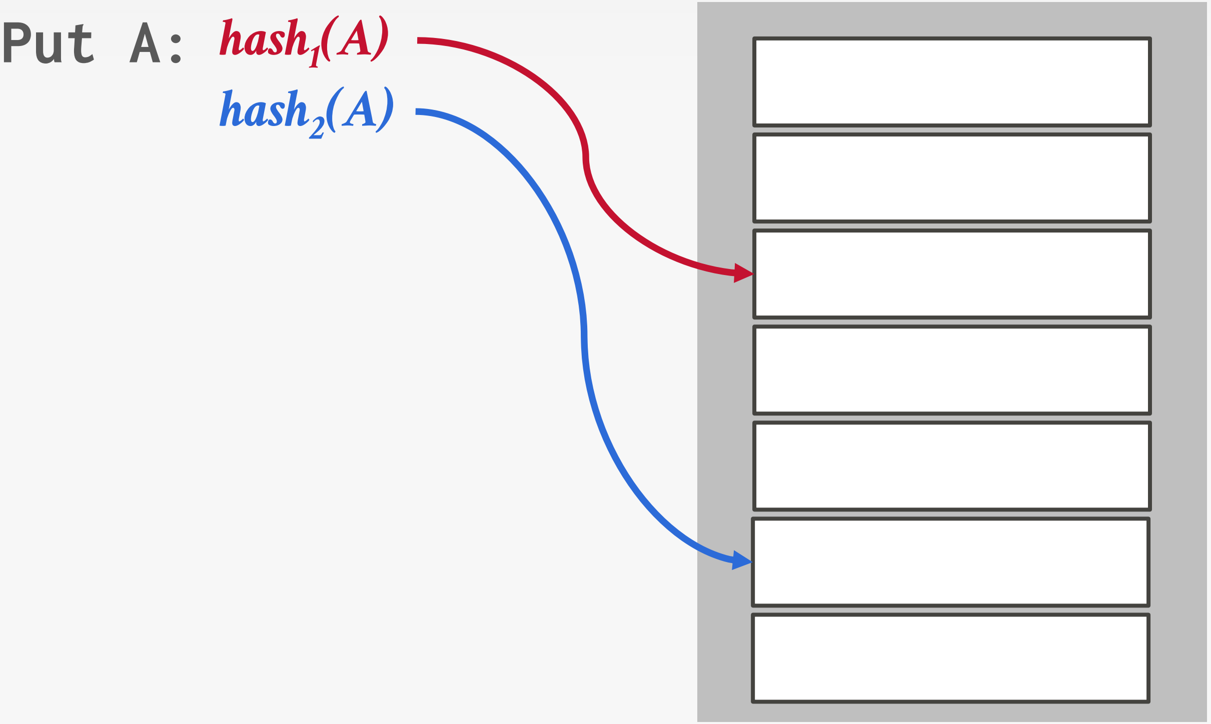
→ On insert, check multiple locations and pick the one that is empty.

→ If no location is available, 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.

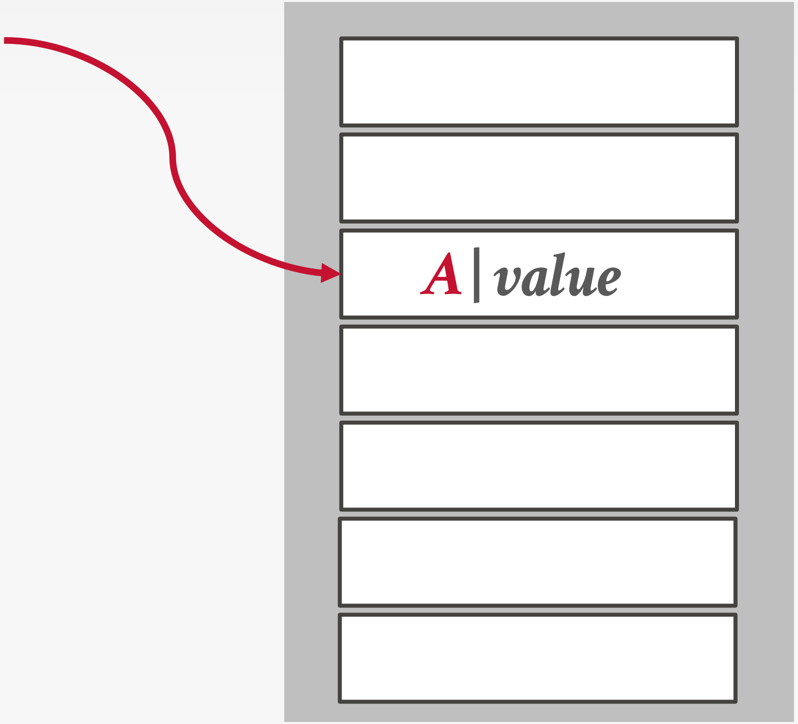
CUCKOO HASHING



CUCKOO HASHING

Put A: $hash_1(A)$

$hash_2(A)$



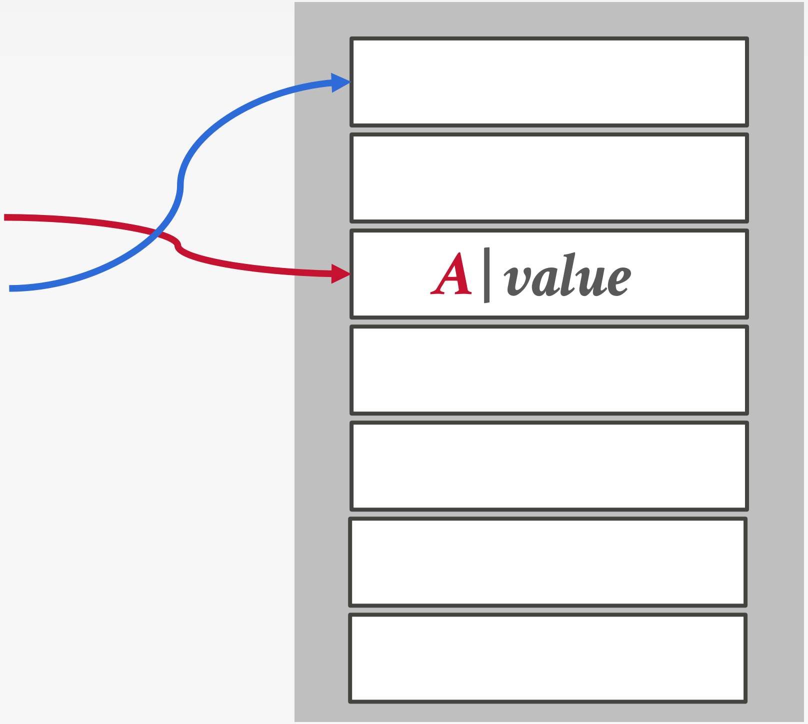
CUCKOO HASHING

Put A: $hash_1(A)$

$hash_2(A)$

Put B: $hash_1(B)$

$hash_2(B)$



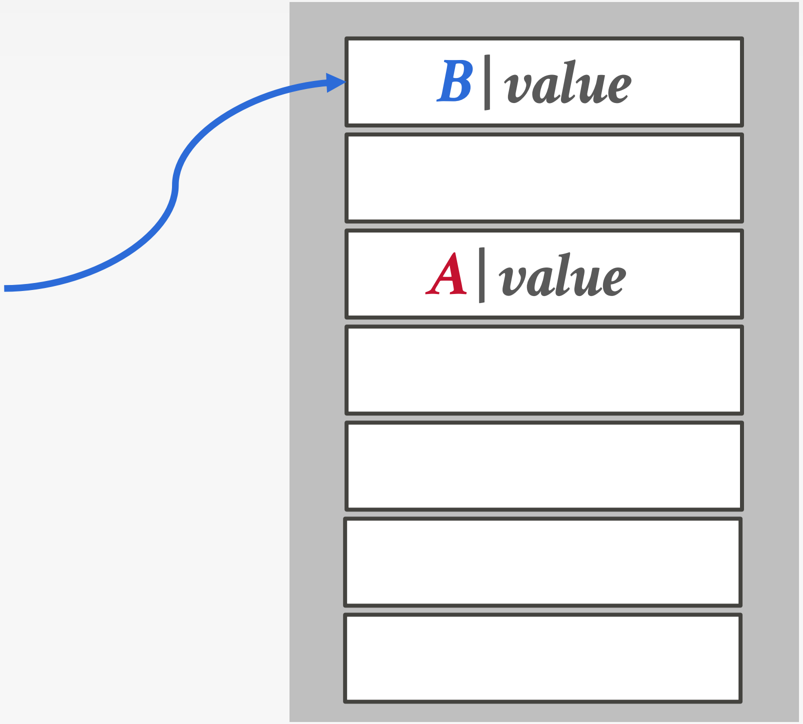
CUCKOO HASHING

Put A: $hash_1(A)$

$hash_2(A)$

Put B: $hash_1(B)$

$hash_2(B)$

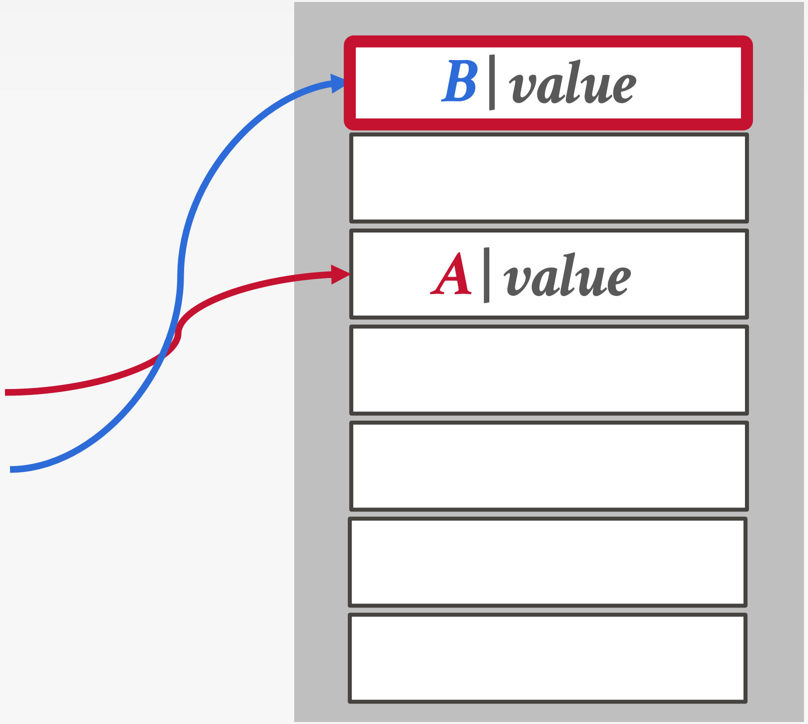


CUCKOO HASHING

Put A: $hash_1(A)$
$hash_2(A)$

Put B: $hash_1(B)$
$hash_2(B)$

Put C: $hash_1(C)$
$hash_2(C)$



CUCKOO HASHING

Put A: $hash_1(A)$

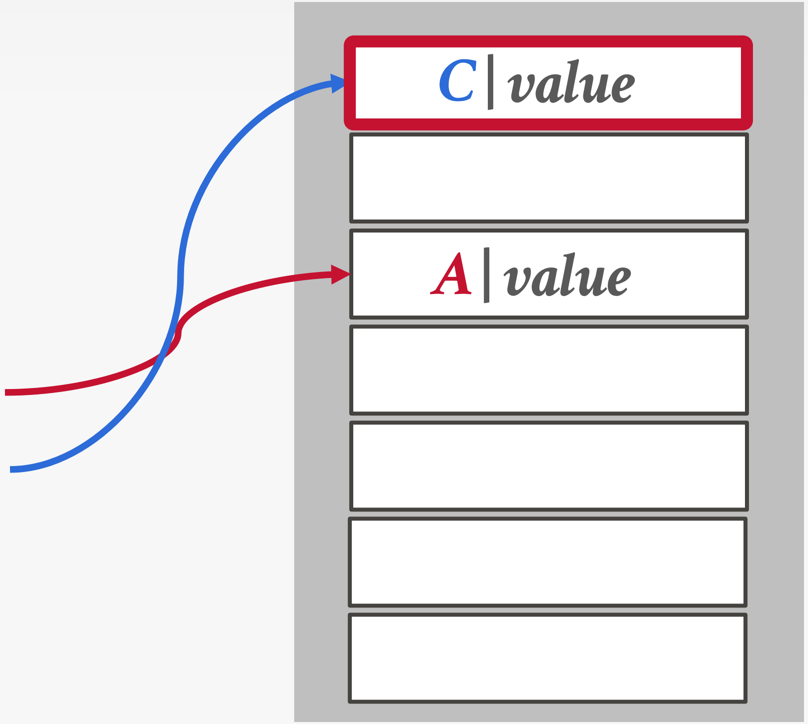
$hash_2(A)$

Put B: $hash_1(B)$

$hash_2(B)$

Put C: $hash_1(C)$

$hash_2(C)$



CUCKOO HASHING

Put A: $hash_1(A)$

$hash_2(A)$

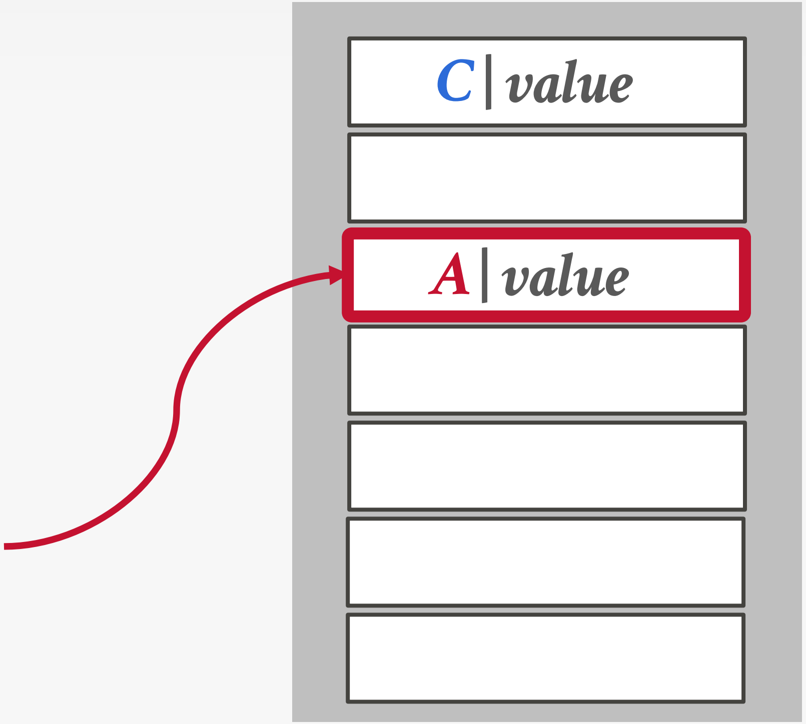
Put B: $hash_1(B)$

$hash_2(B)$

Put C: $hash_1(C)$

$hash_2(C)$

$hash_1(B)$



CUCKOO HASHING

Put A: $hash_1(A)$

$hash_2(A)$

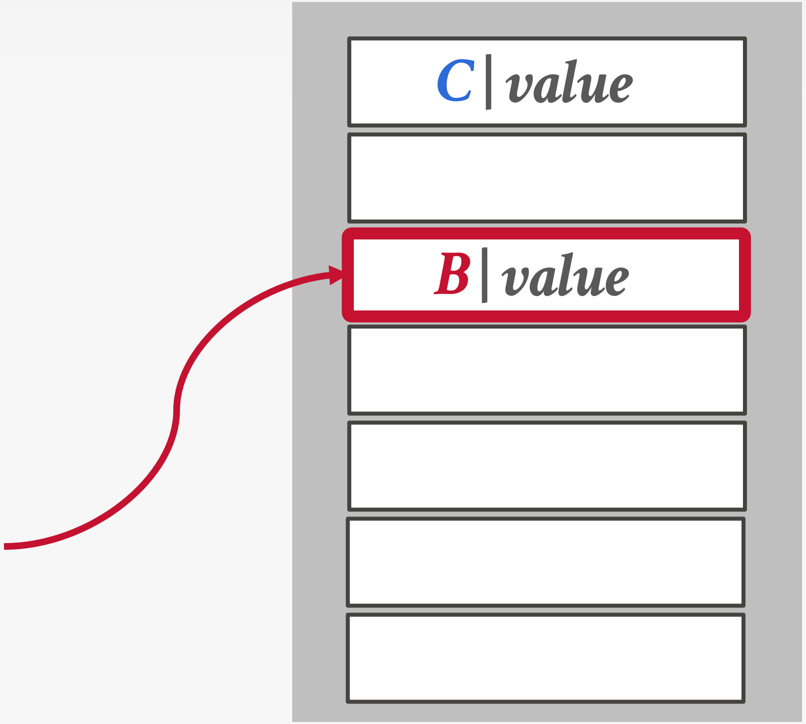
Put B: $hash_1(B)$

$hash_2(B)$

Put C: $hash_1(C)$

$hash_2(C)$

$hash_1(B)$



CUCKOO HASHING

Put A: $hash_1(A)$

$hash_2(A)$

Put B: $hash_1(B)$

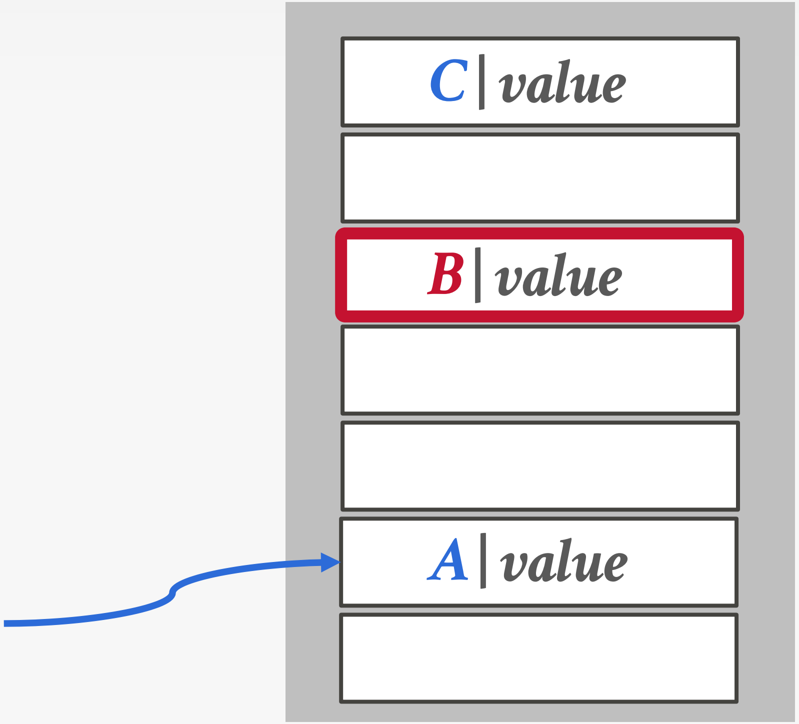
$hash_2(B)$

Put C: $hash_1(C)$

$hash_2(C)$

$hash_1(B)$

$hash_2(A)$



CUCKOO HASHING

Put A: $hash_1(A)$

$hash_2(A)$

Put B: $hash_1(B)$

$hash_2(B)$

Put C: $hash_1(C)$

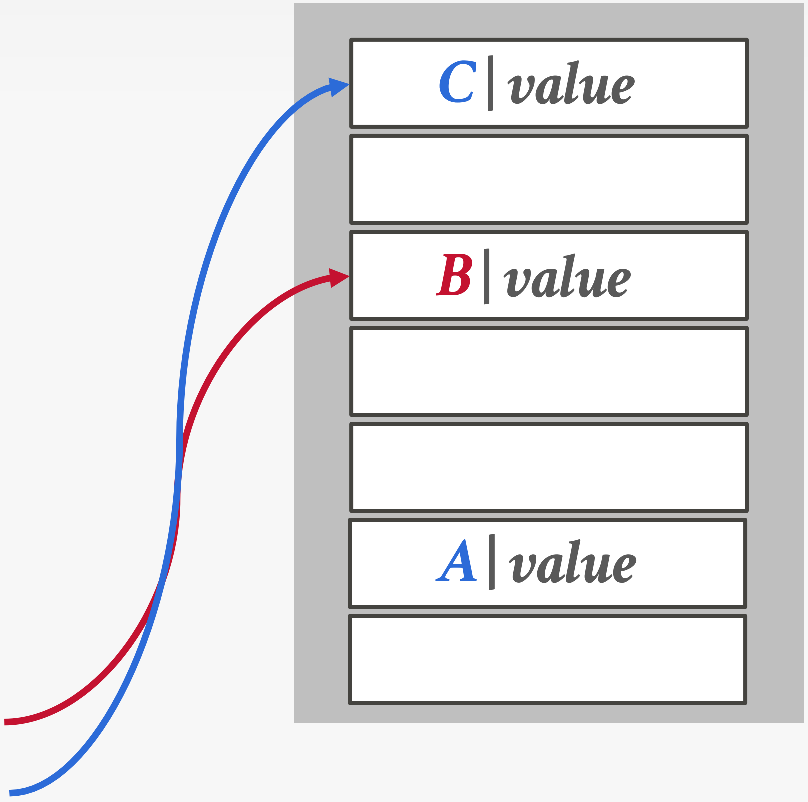
$hash_2(C)$

$hash_1(B)$

$hash_2(A)$

Get B: $hash_1(B)$

$hash_2(B)$



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 incrementally resize themselves as needed.

→ Chained Hashing
→ Extendible Hashing
→ Linear Hashing

CHAINED HASHING

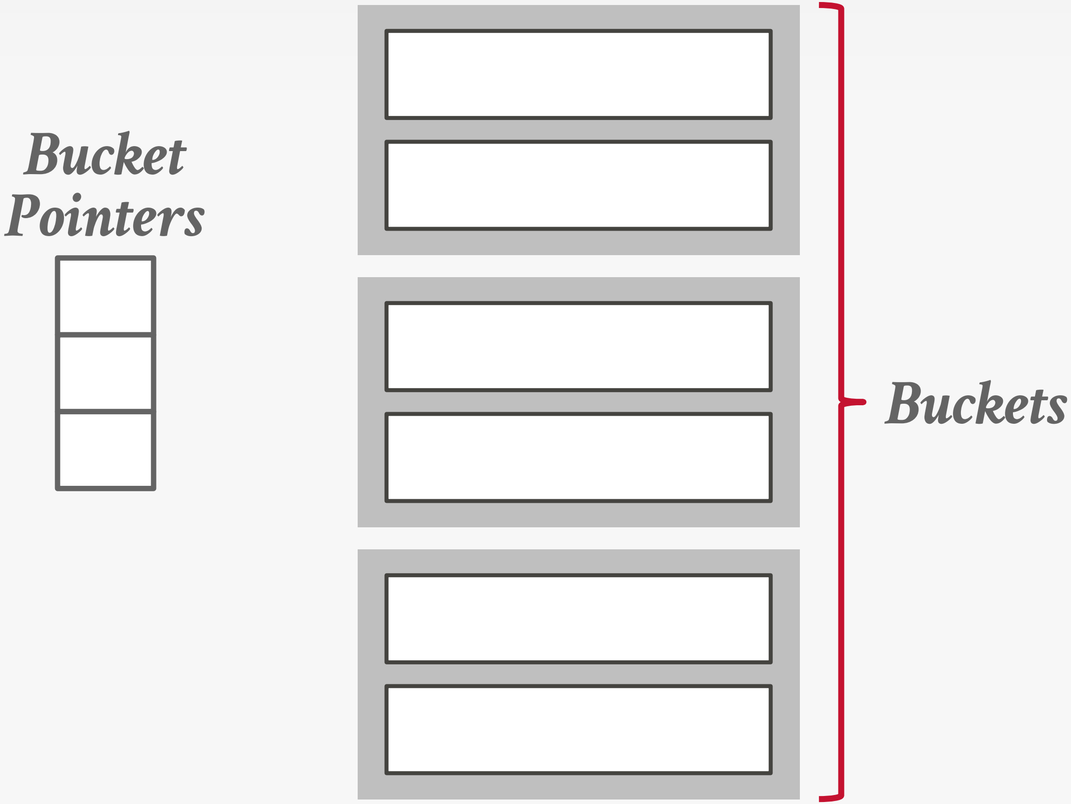
Maintain a linked list of buckets for each slot in the hash table.

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

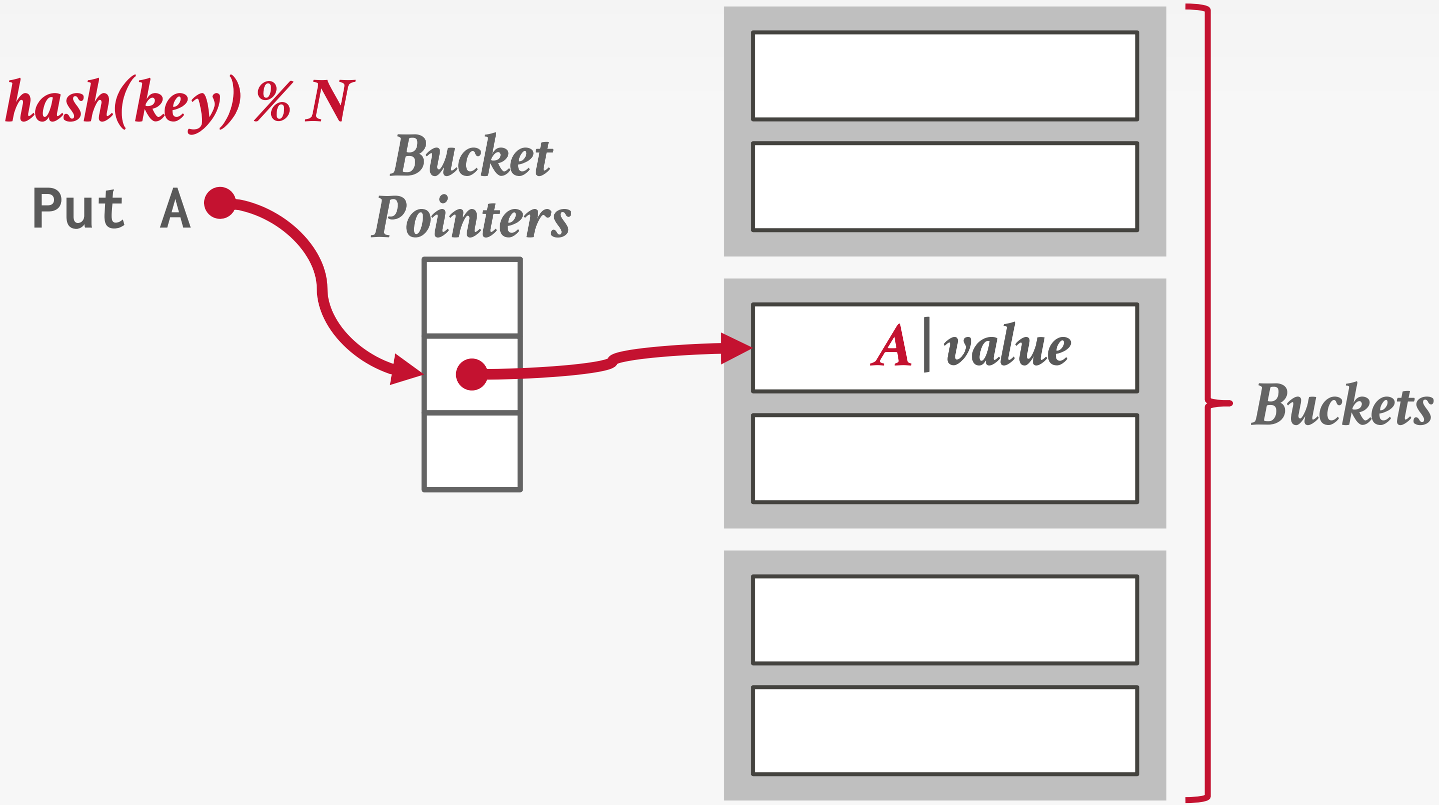
- → To determine whether an element is present, hash to its bucket and scan for it.
- → Insertions and deletions are generalizations of lookups.

CHAINED HASHING

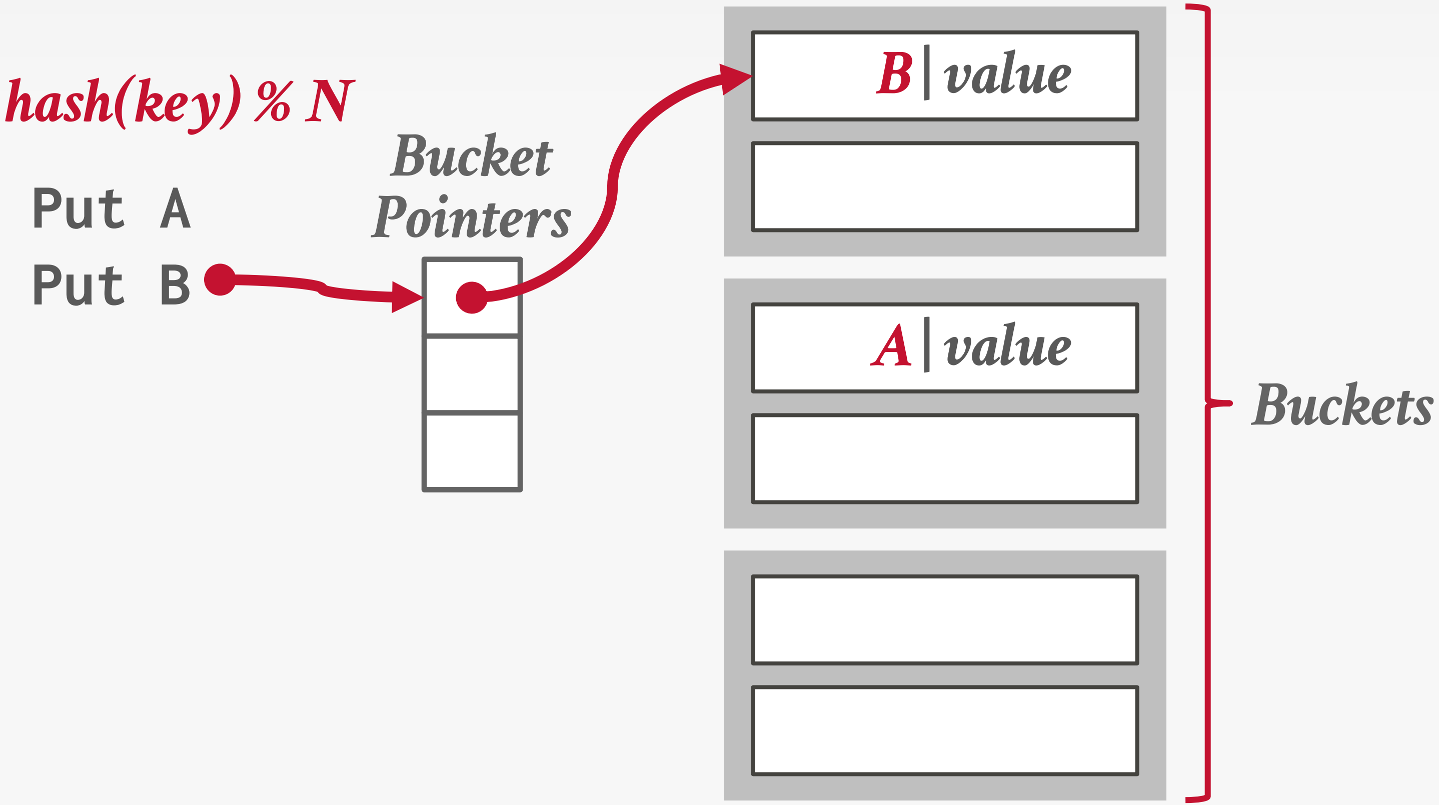
hash(key) % N



CHAINED HASHING



CHAINED HASHING



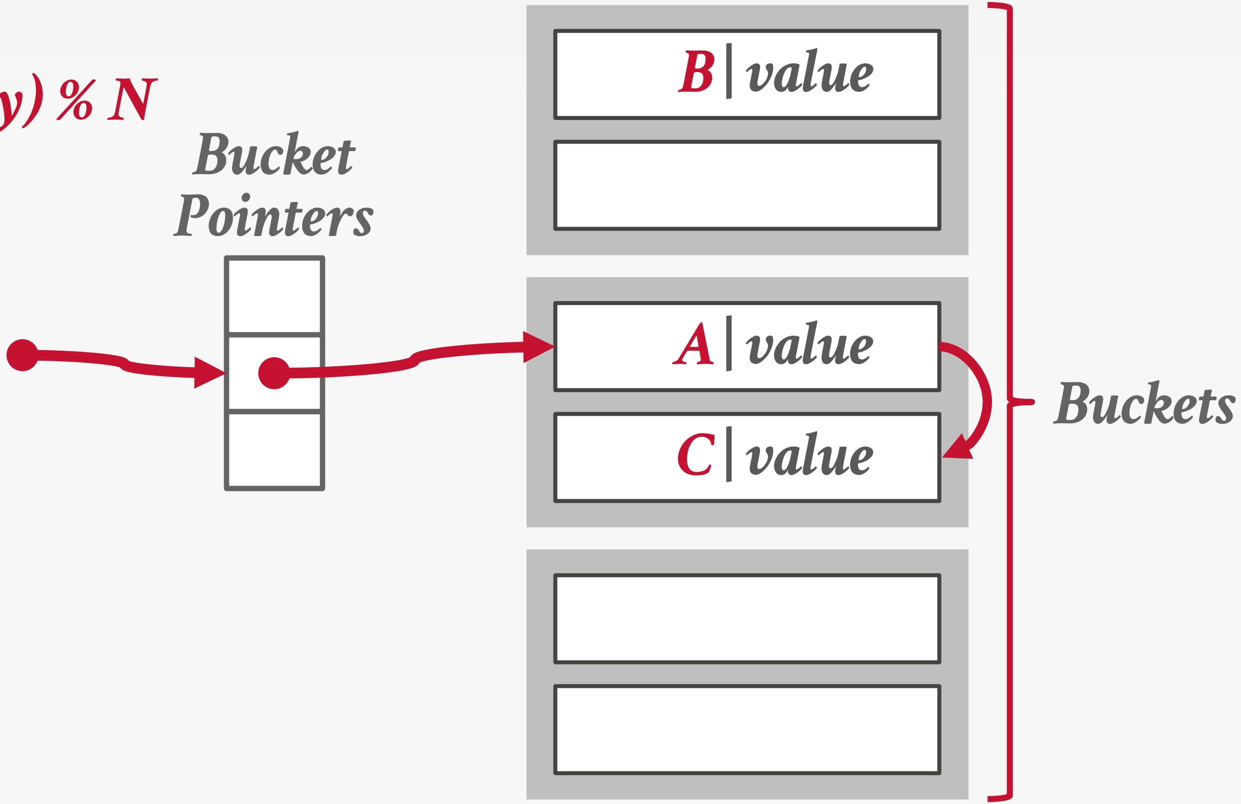
CHAINED HASHING

hash(key) % N

Put A

Put B

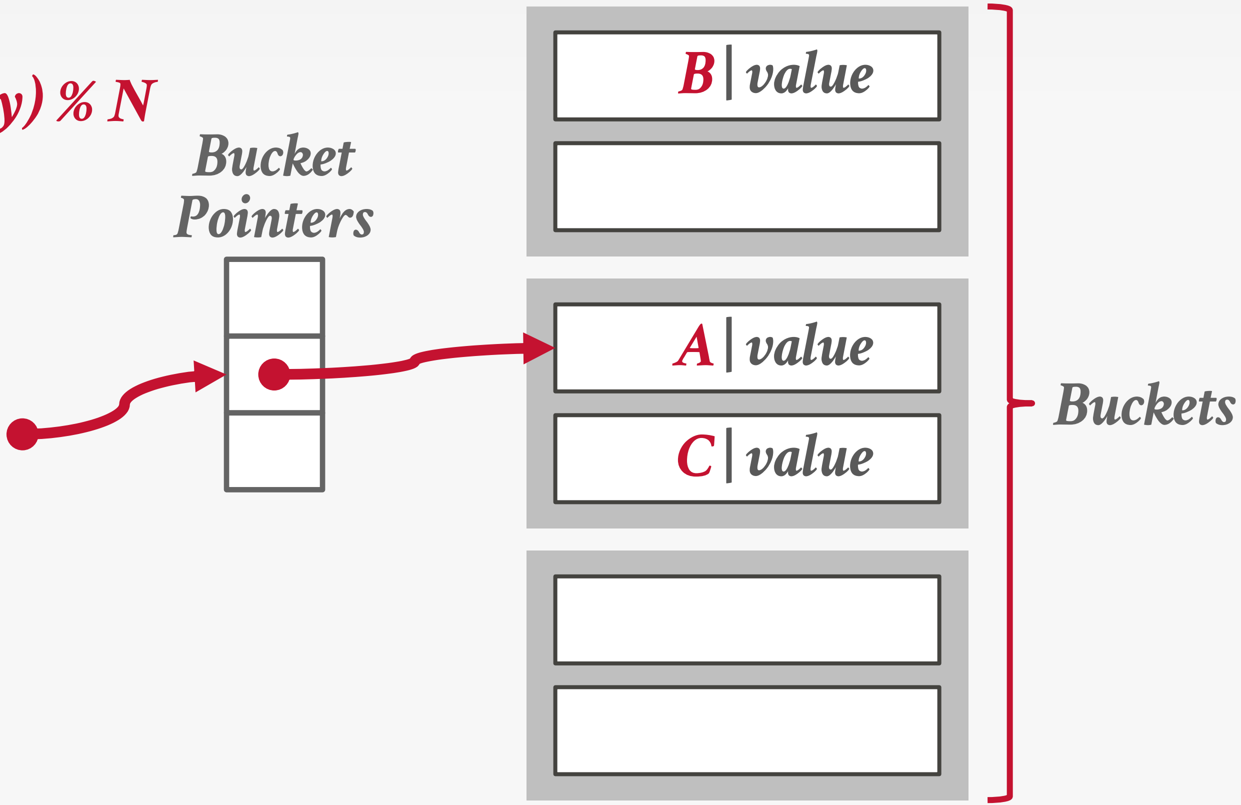
Put C



CHAINED HASHING

hash(key) % N

```
Put A
Put B
Put C
Put D
```



CHAINED HASHING

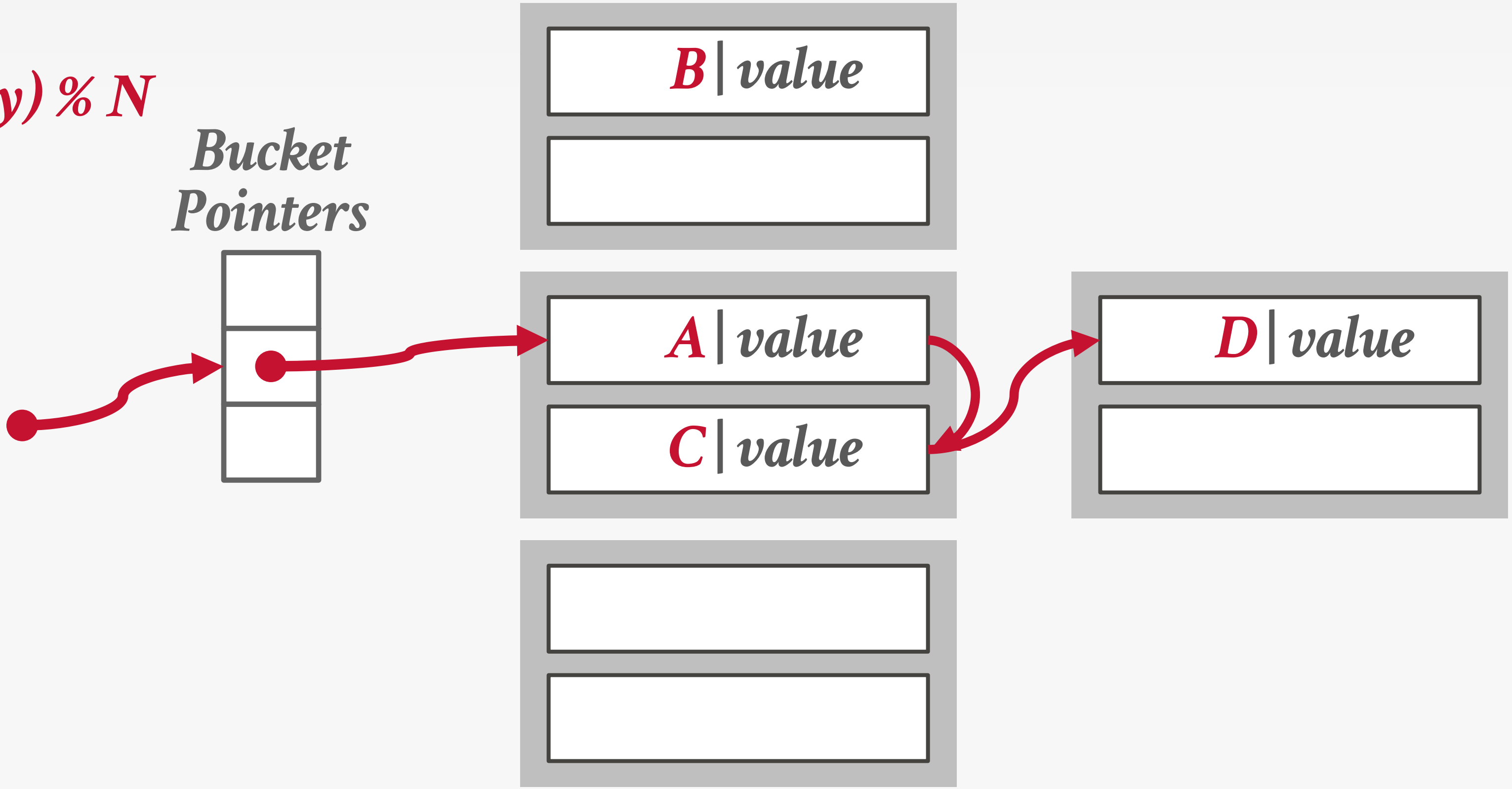
hash(key) % N

Put A

Put B

Put C

Put D



Bucket Pointers

B | value

A | value

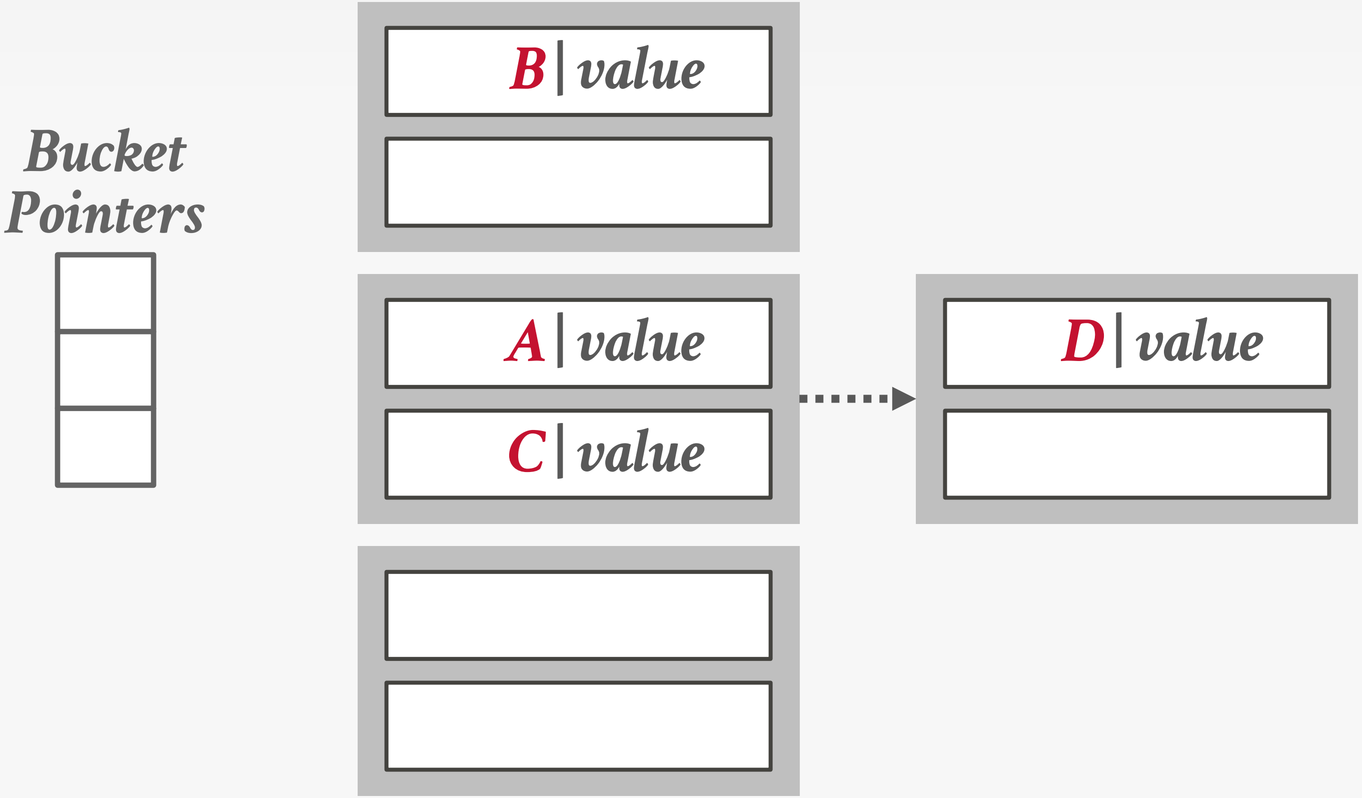
C | value

D | value

CHAINED HASHING

hash(key) % N

```
Put A
Put B
Put C
Put D
Put E
```



CHAINED HASHING

hash(key) % N

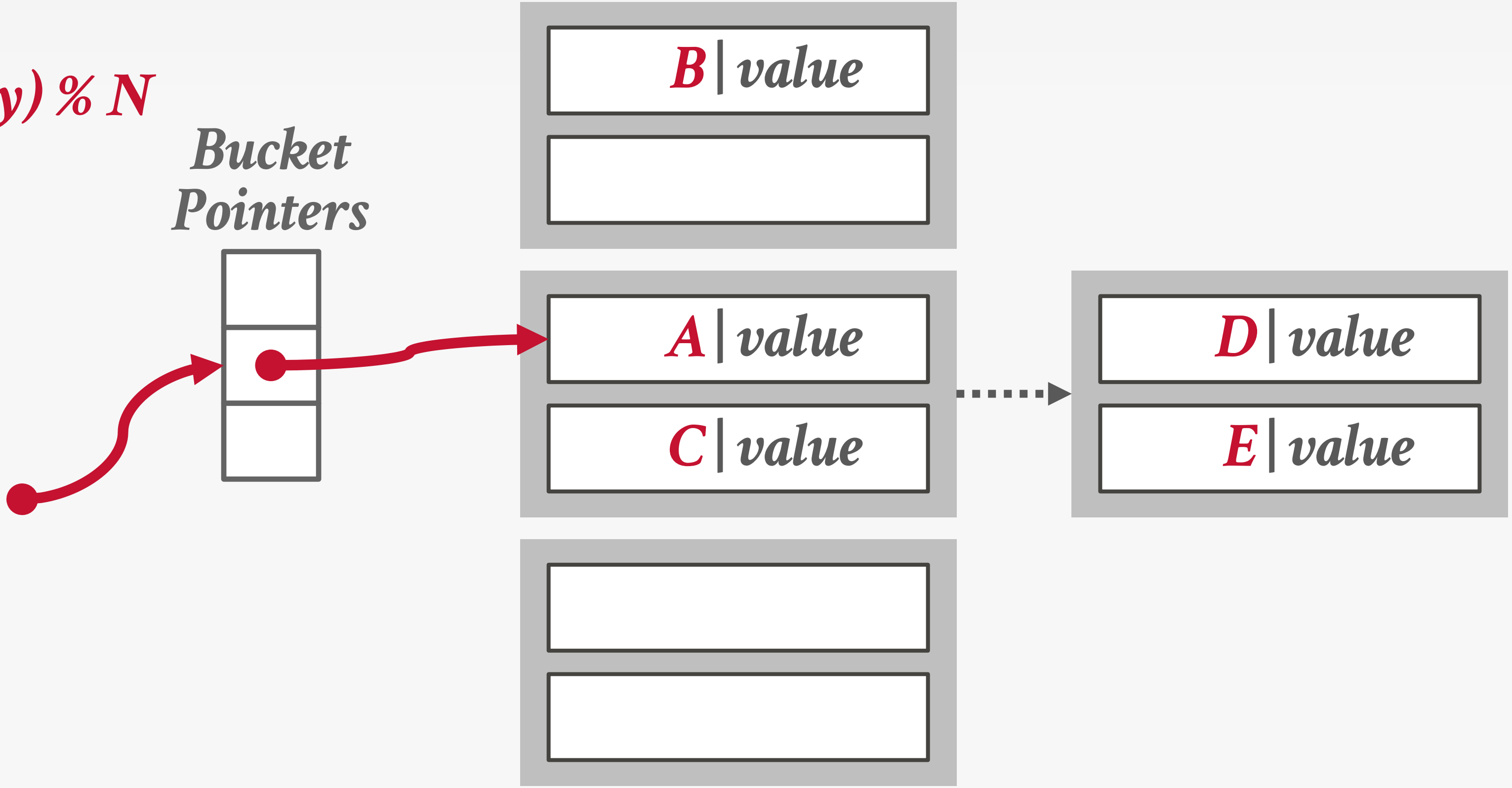
Put A

Put B

Put C

Put D

Put E



CHAINED HASHING

hash(key) % N

Put A

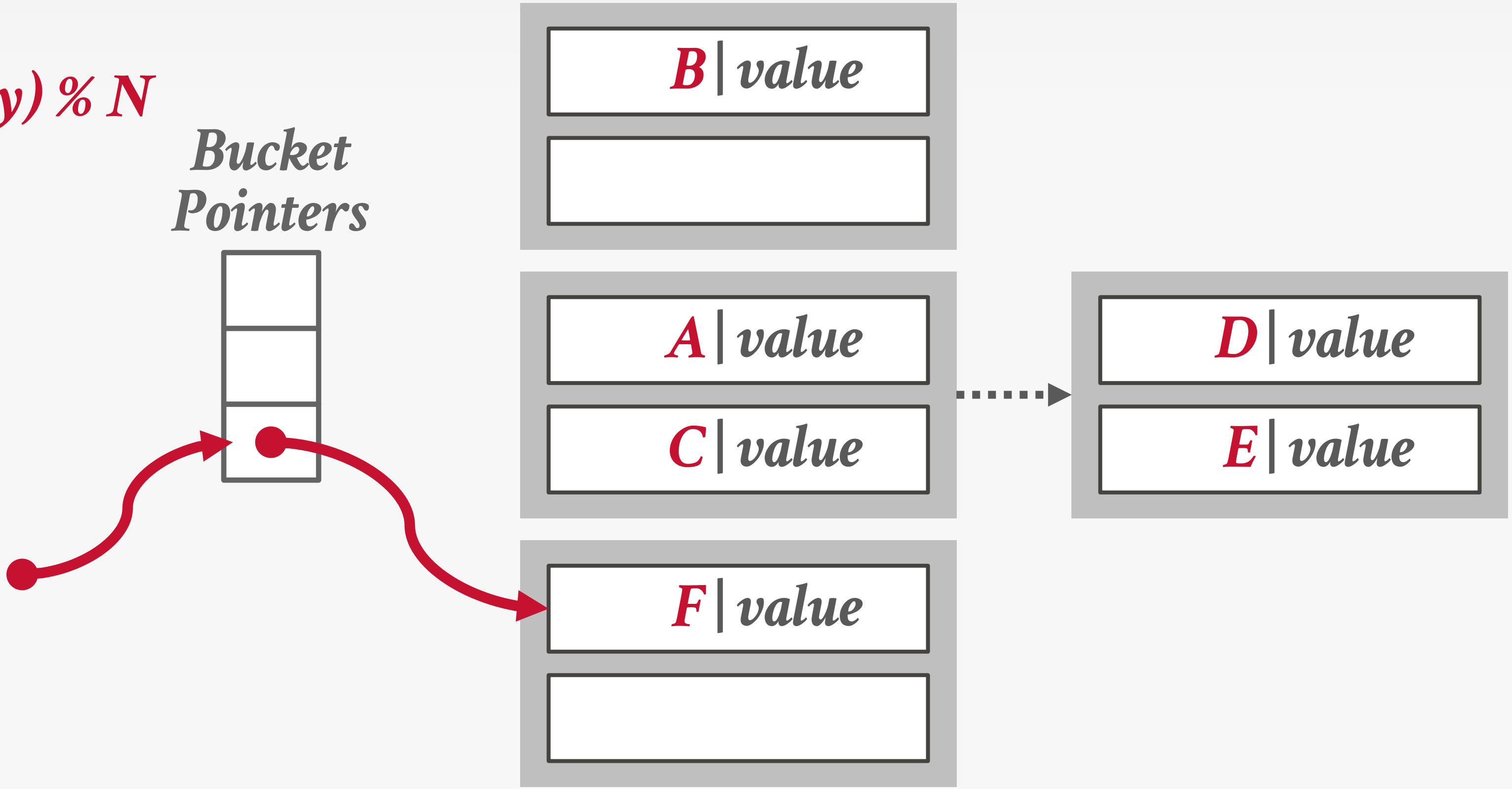
Put B

Put C

Put D

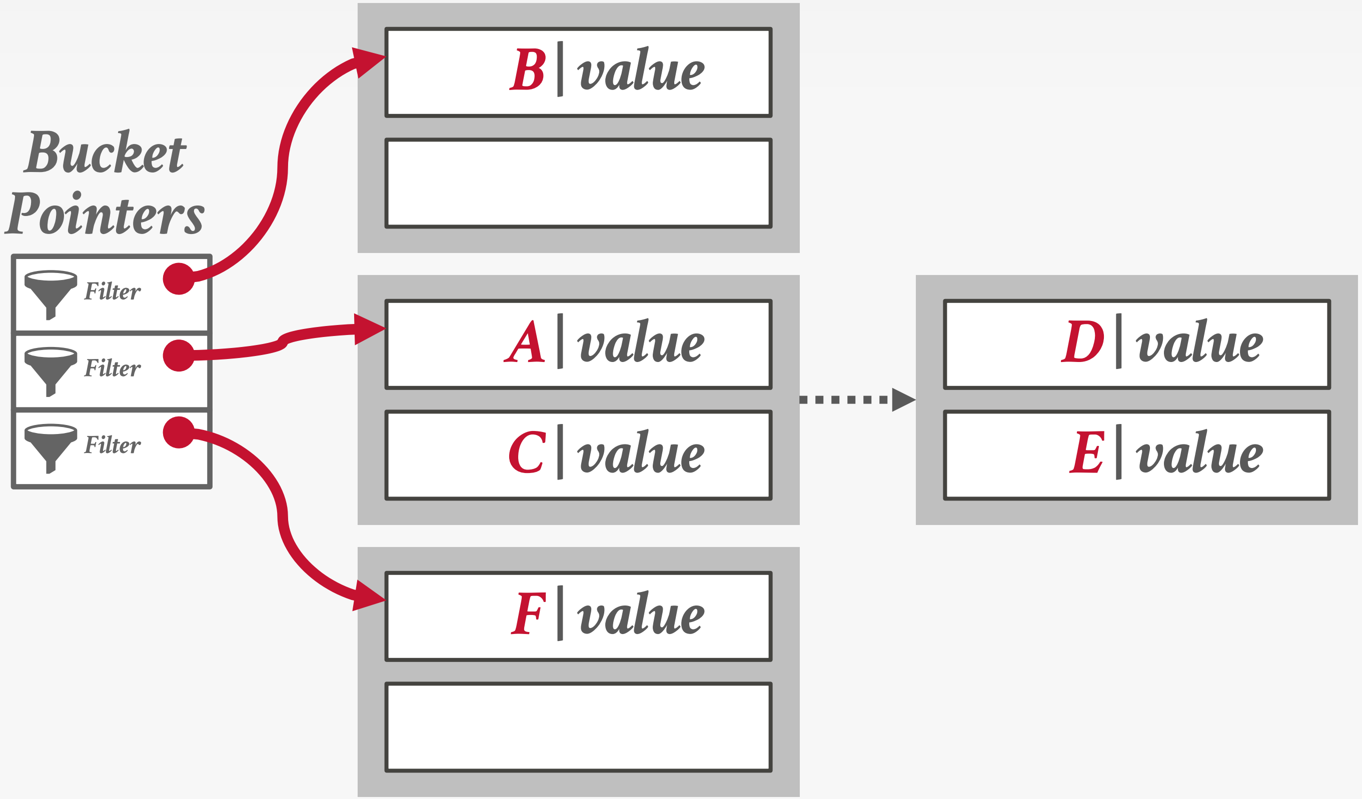
Put E

Put F

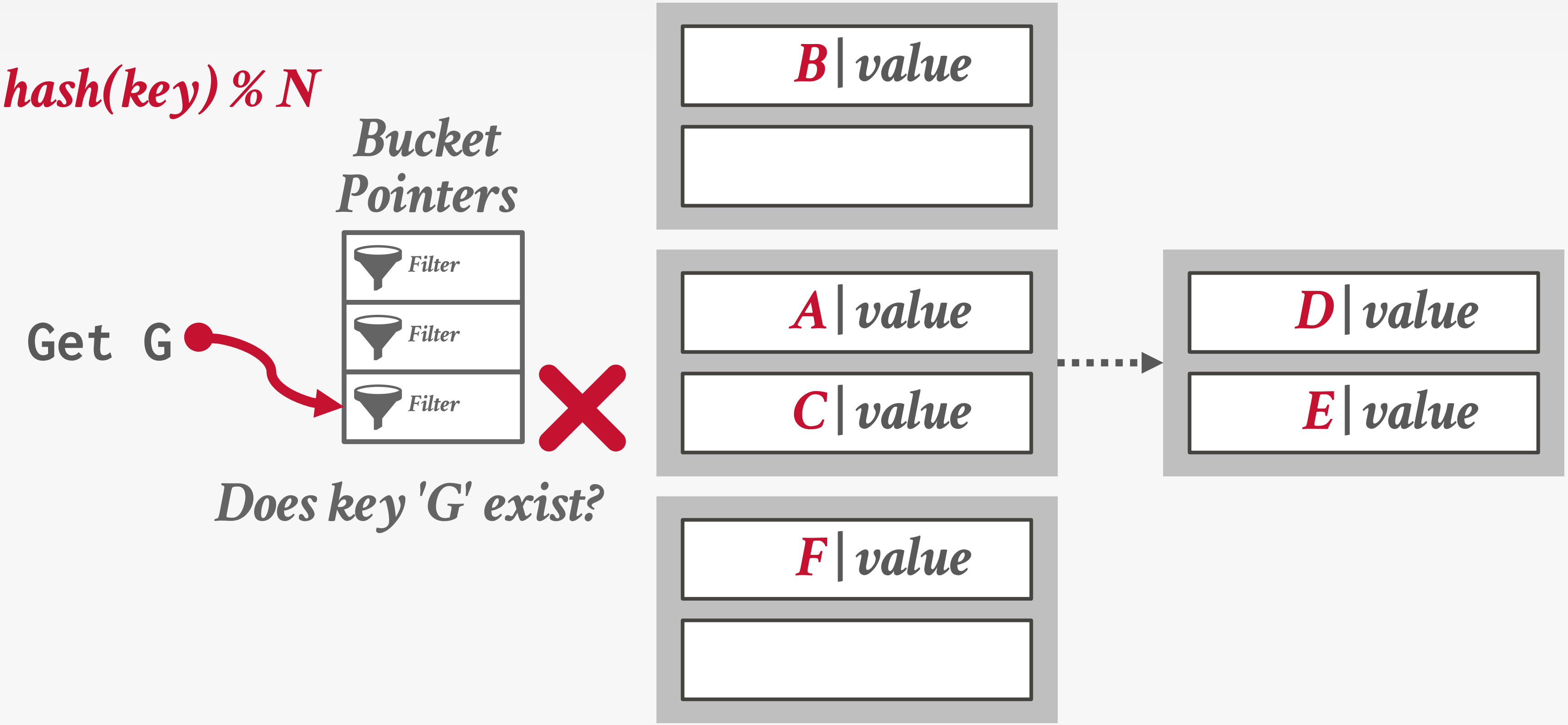


CHAINED HASHING

hash(key) % N



CHAINED HASHING



EXTENDIBLE HASHING

Chained-hashing approach that splits buckets incrementally instead of letting the linked list grow forever.

Multiple slot locations can point to the same bucket chain.

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

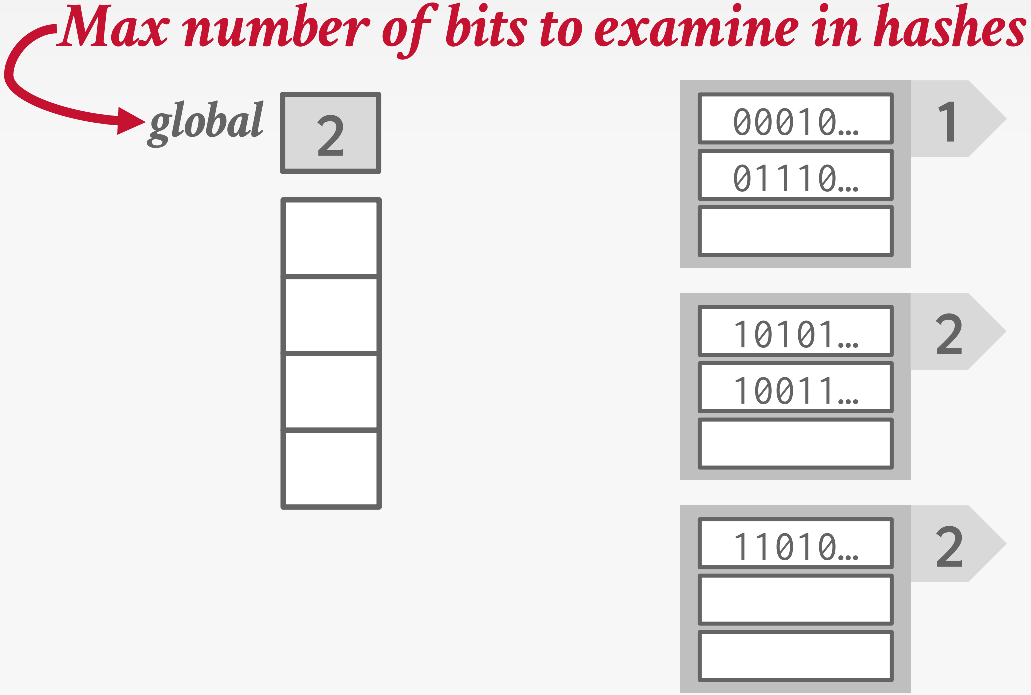
→ Data movement is localized to just the split chain.

GDBM

AsterixDB™

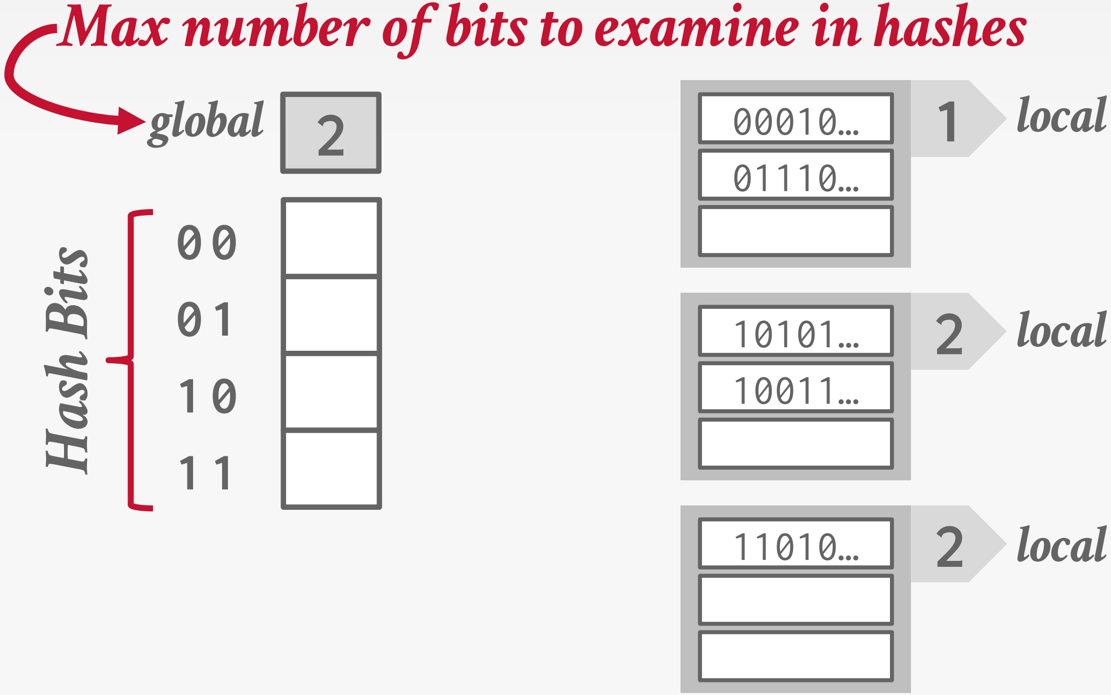
EXTENDIBLE HASHING

Max number of bits to examine in hashes

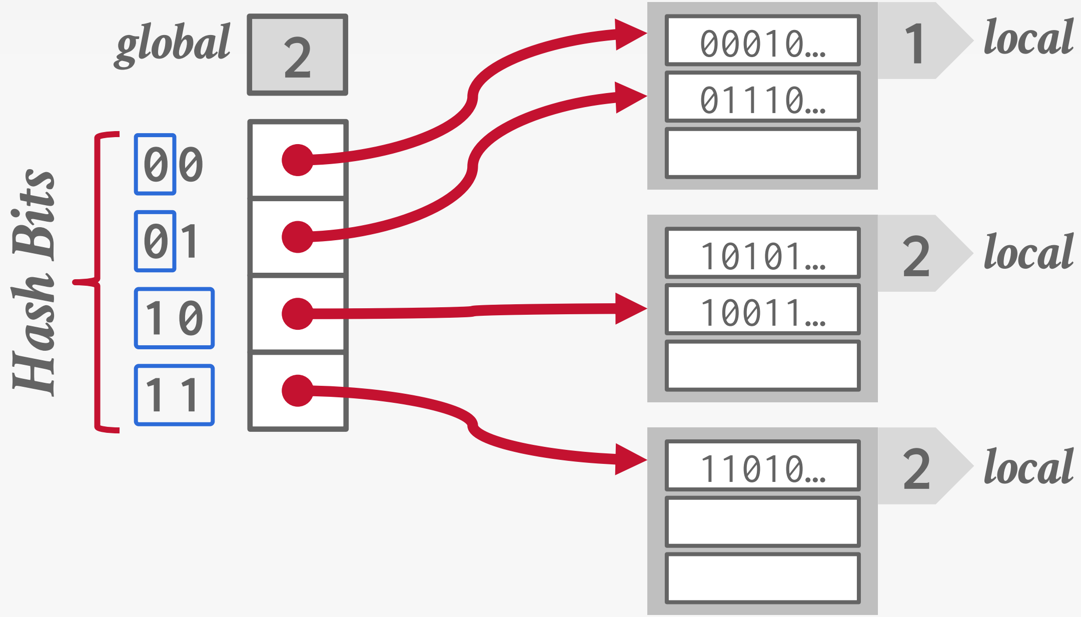


EXTENDIBLE HASHING

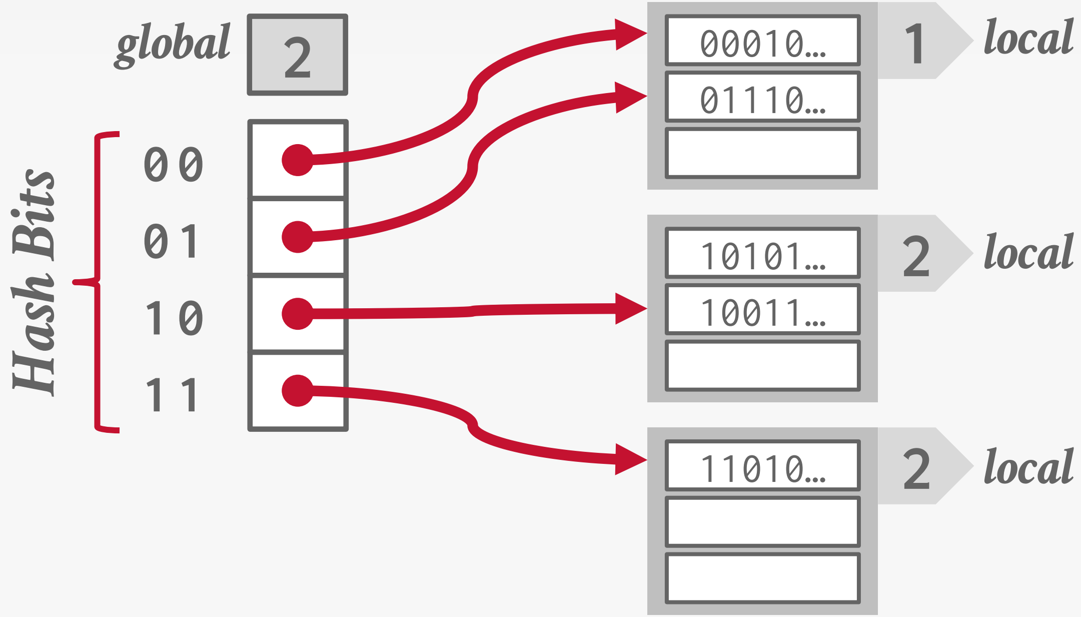
Max number of bits to examine in hashes



EXTENDIBLE HASHING



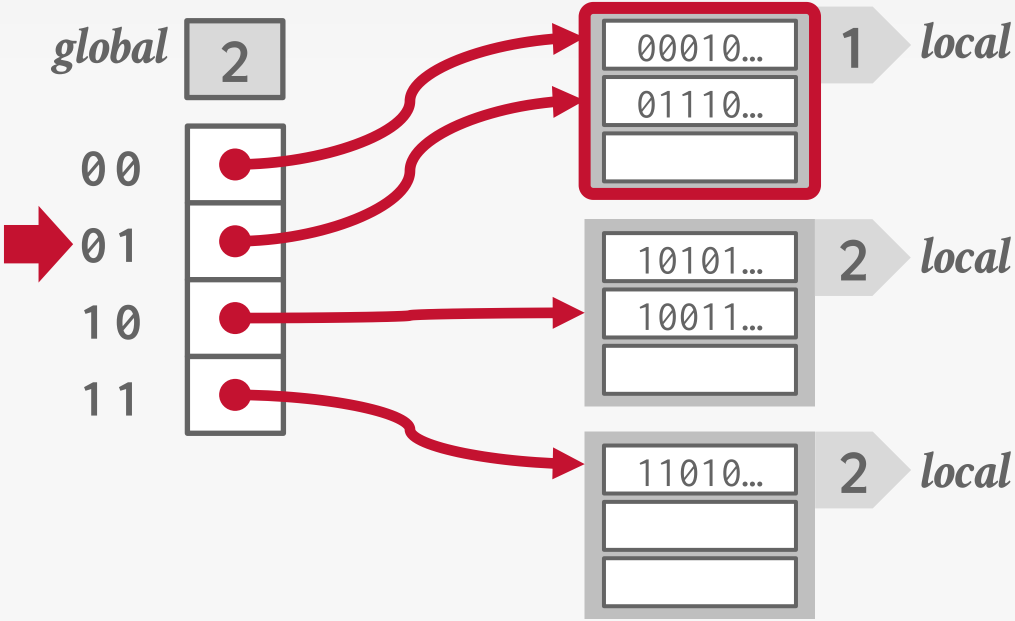
EXTENDIBLE HASHING



Get A

hash(A) = $\boxed{01}$110…

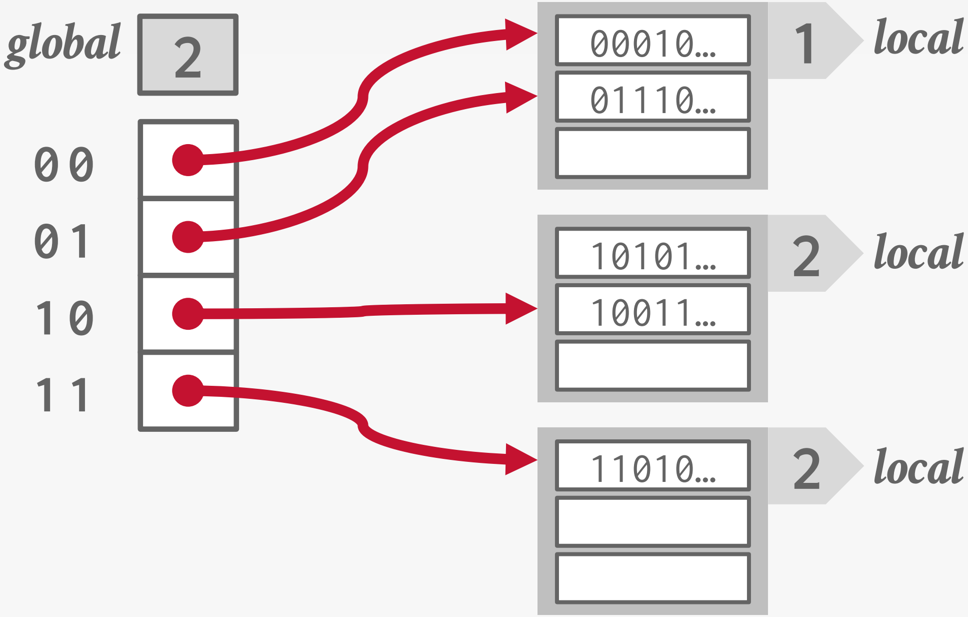
EXTENDIBLE HASHING



Get A

$hash(A) = \boxed{01}110\ldots$

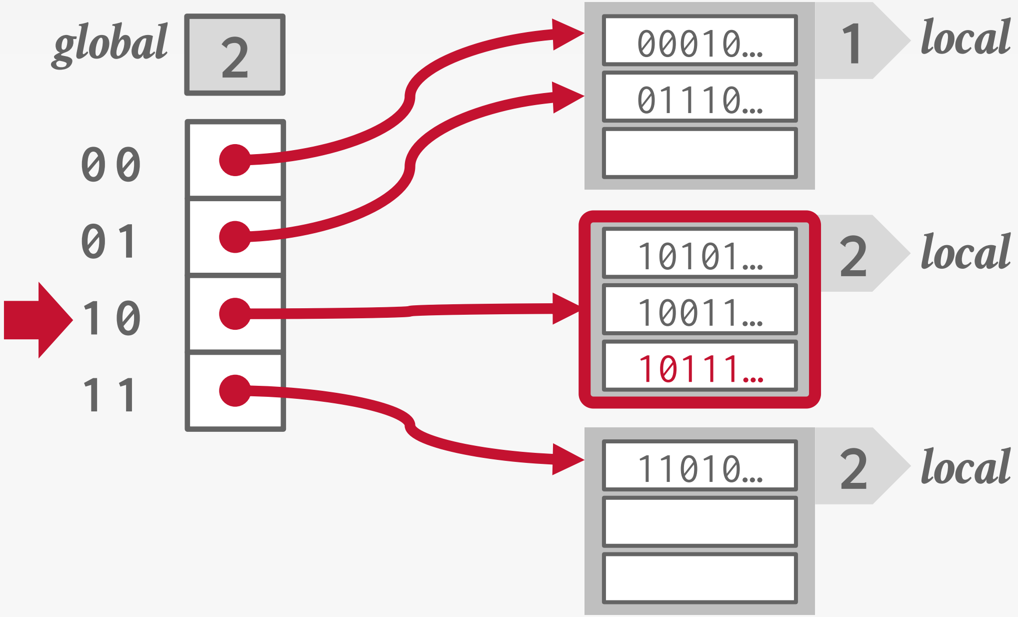
EXTENDIBLE HASHING



Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

EXTENDIBLE HASHING



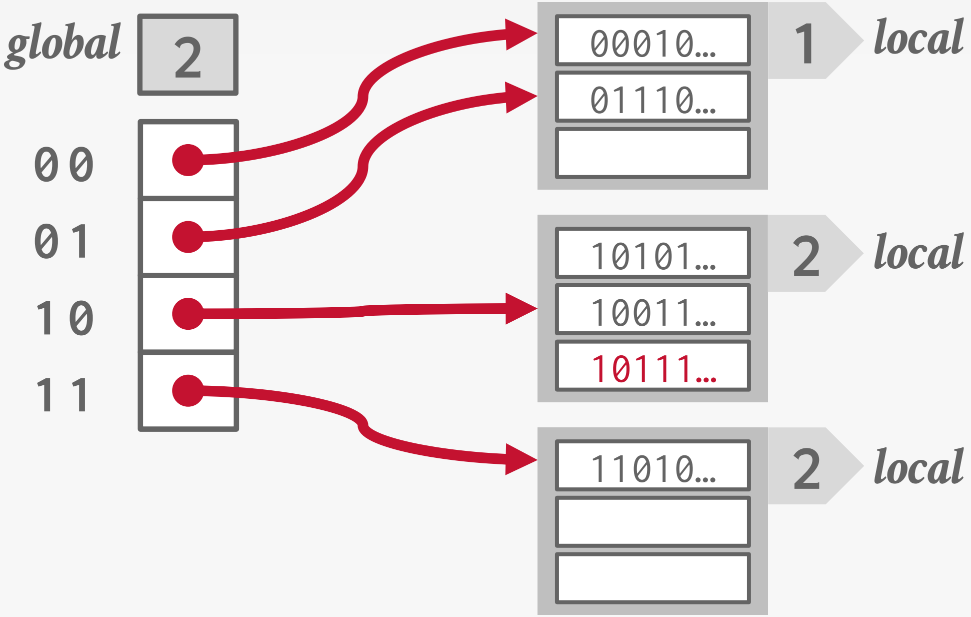
Get A

hash(A) = 01110…

Put B

hash(B) = 10111…

EXTENDIBLE HASHING



Get A

hash(A) = 01110…

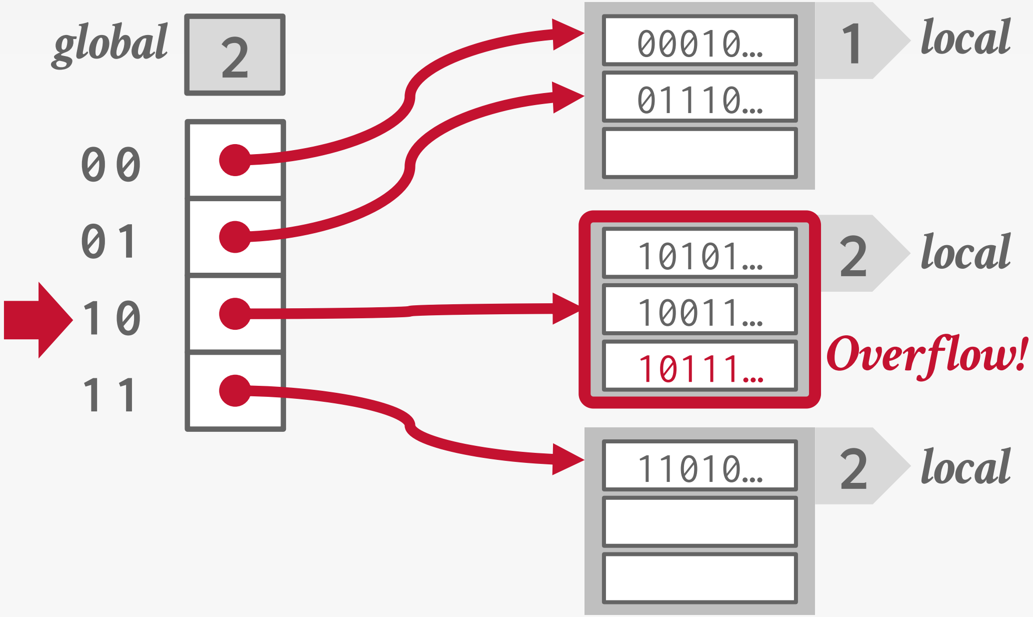
Put B

hash(B) = 10111…

Put C

hash(C) = 10100…

EXTENDIBLE HASHING

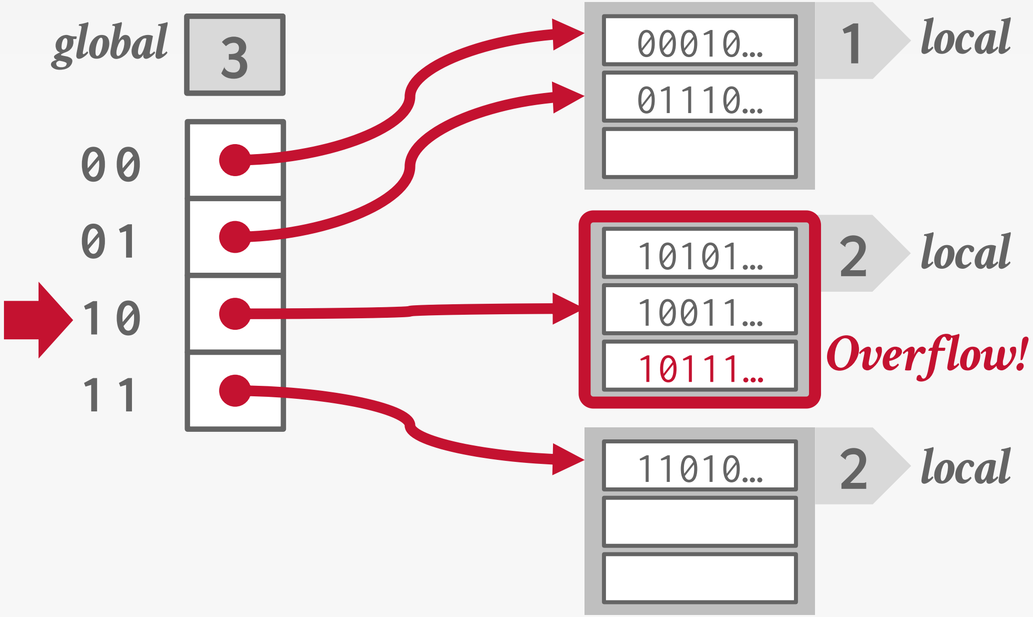


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = $\boxed{10}$100…

EXTENDIBLE HASHING



Get A

hash(A) = 01110…

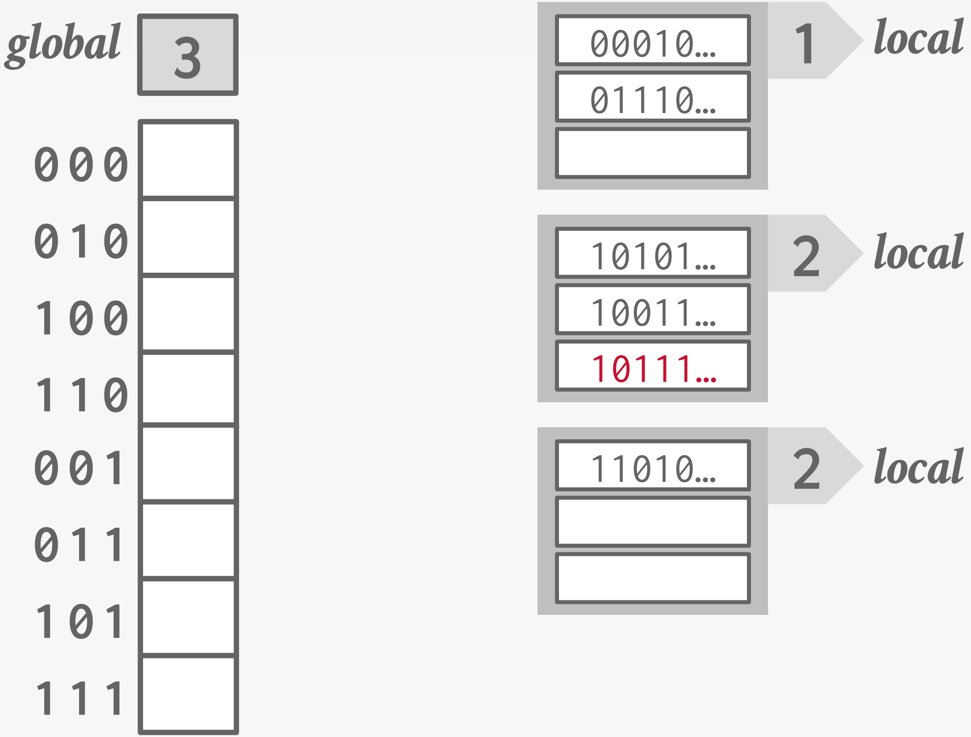
Put B

hash(B) = 10111…

Put C

hash(C) = $\boxed{10}$100…

EXTENDIBLE HASHING

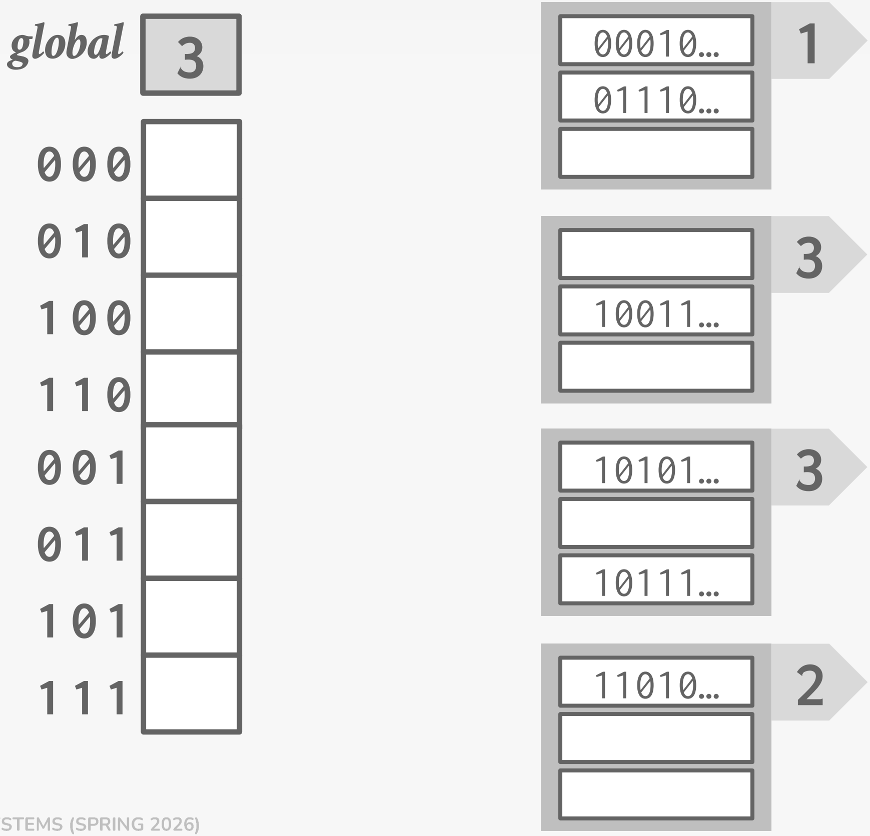


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = 10100…

EXTENDIBLE HASHING

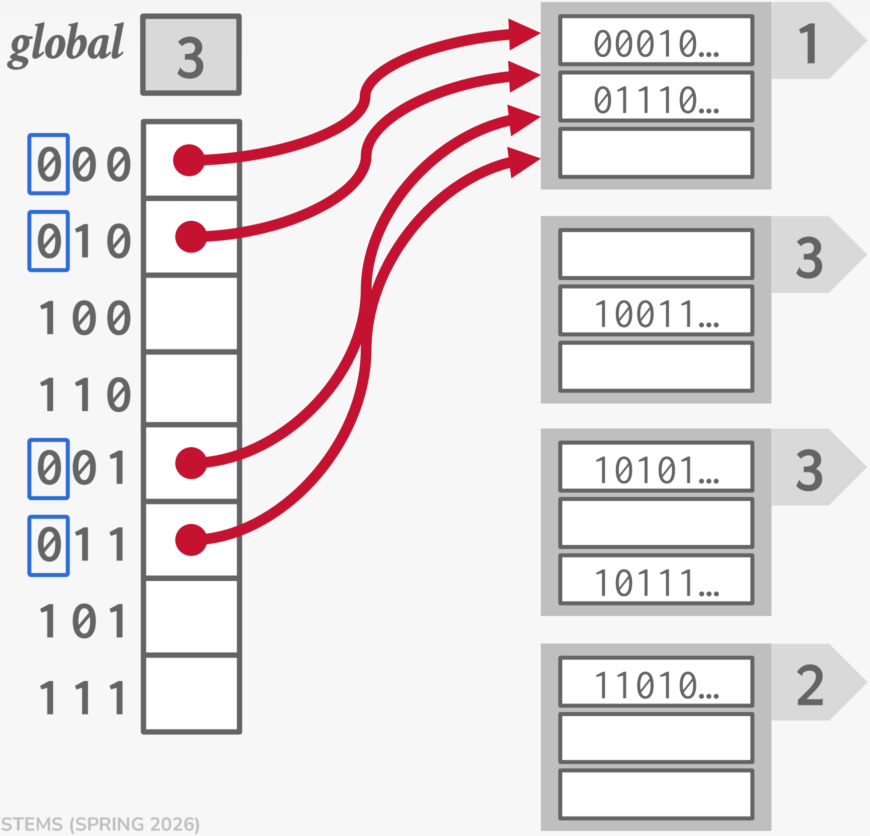


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = 10100…

EXTENDIBLE HASHING

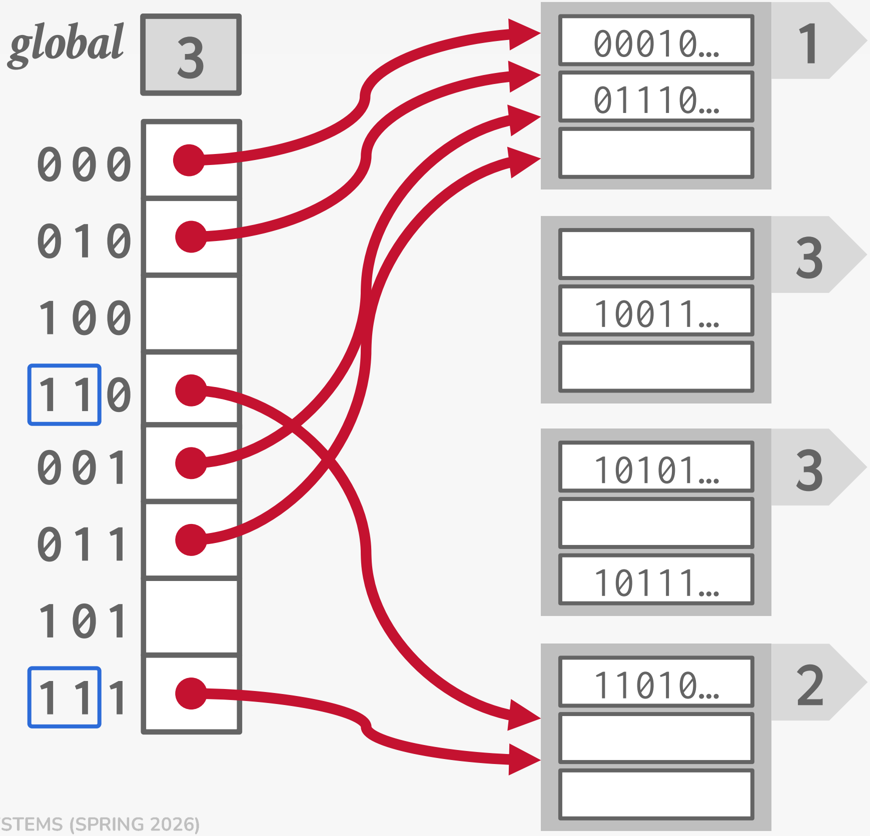


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = 10100…

EXTENDIBLE HASHING

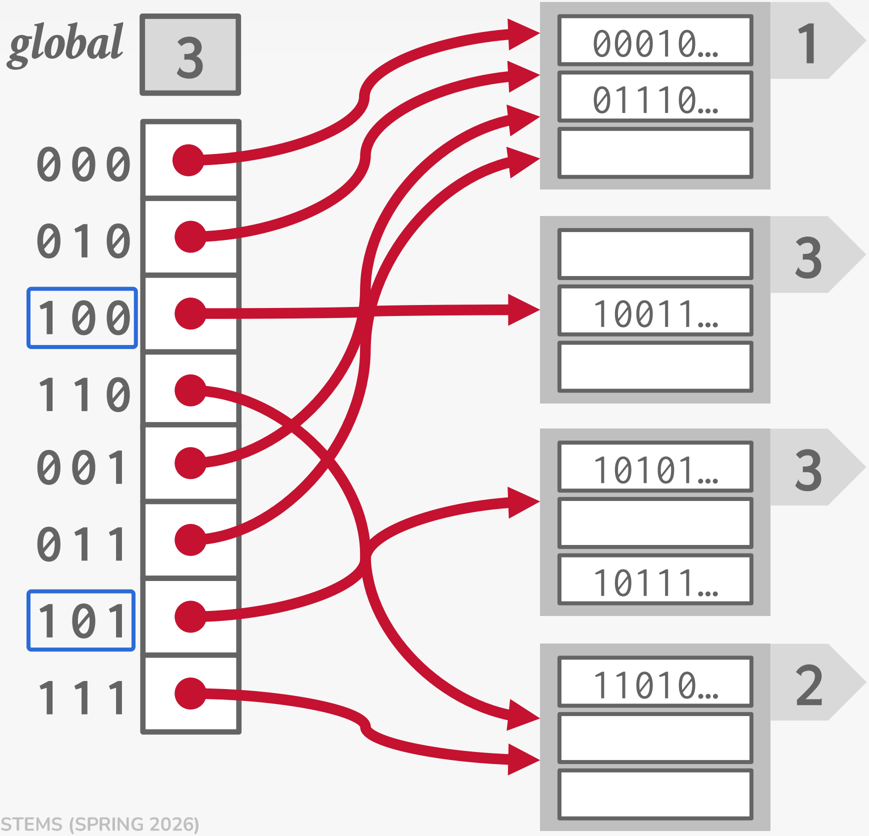


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = 10100…

EXTENDIBLE HASHING

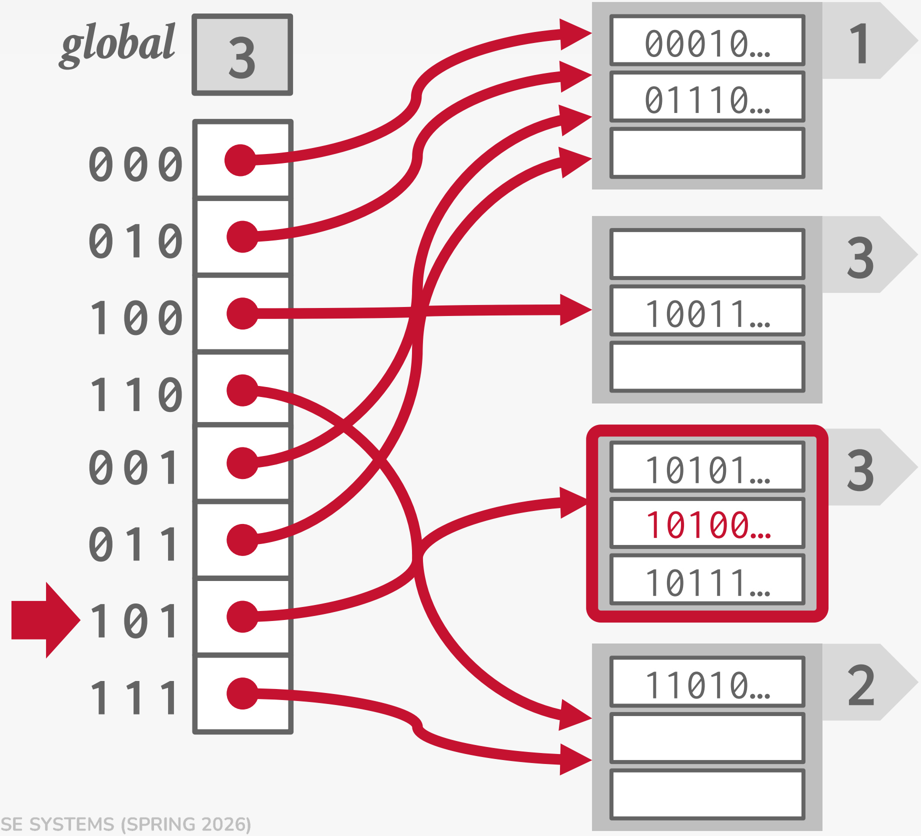


Get A
hash(A) = 01110…

Put B
hash(B) = 10111…

Put C
hash(C) = 10100…

EXTENDIBLE HASHING



Get A

hash(A) = 01110…

Put B

hash(B) = 10111…

Put 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.

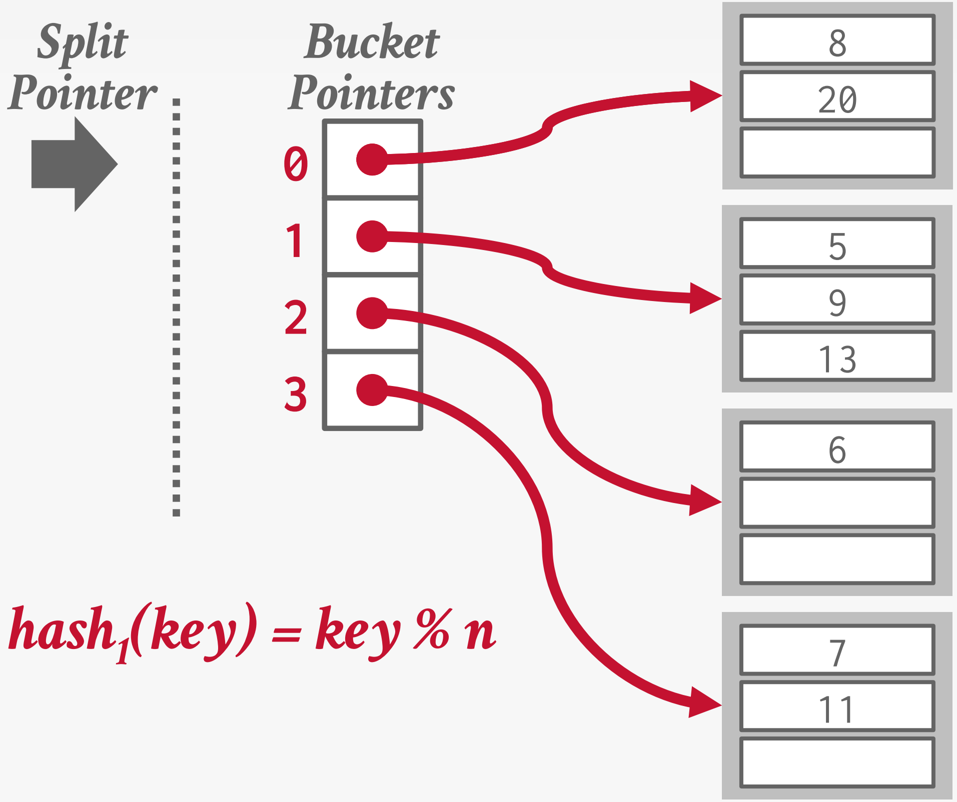
Can use different overflow criterion:

→ Space Utilization

→ Average Length of Overflow Chains

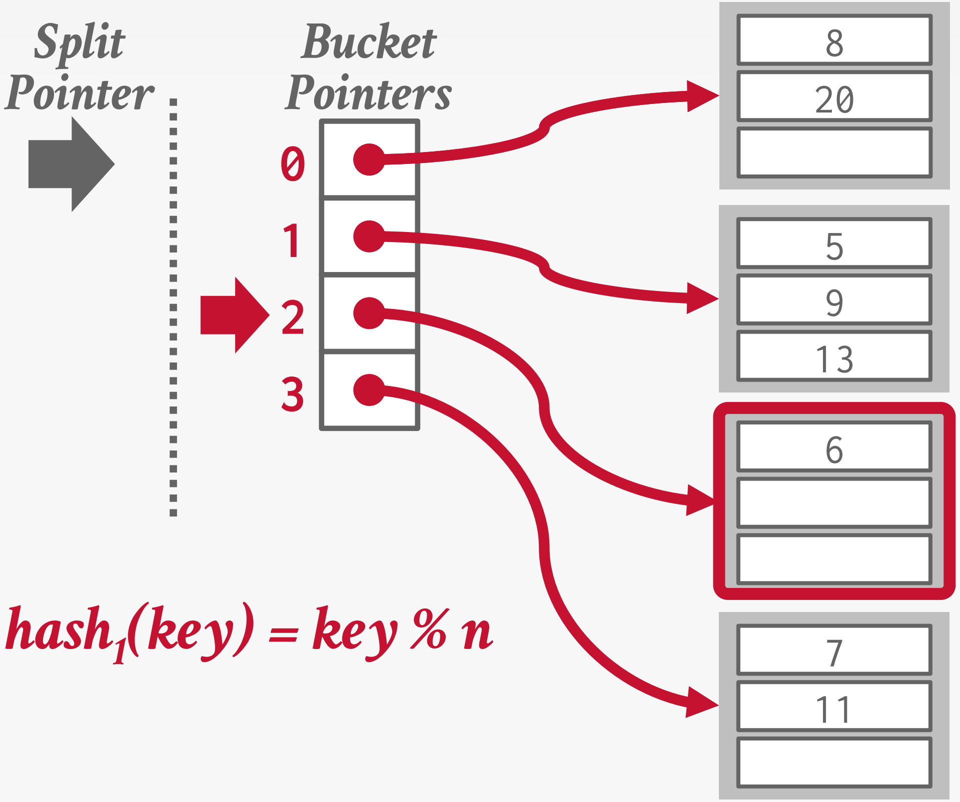
PostgreSQL SLEEPYCAT SOFTWARE Makers of Berkeley DB amazon DynamoDB

LINEAR HASHING



$hash_1(key) = key \% n$

LINEAR HASHING

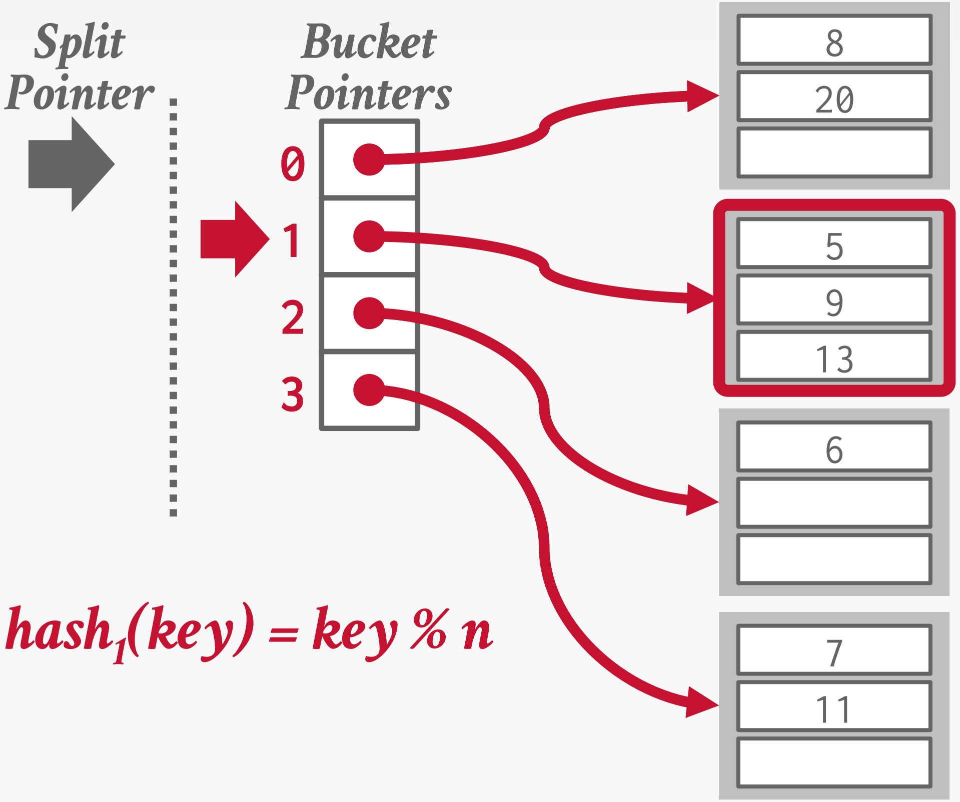


$hash_1(key) = key \% n$

Get 6

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

LINEAR HASHING



$hash_1(key) = key \% n$

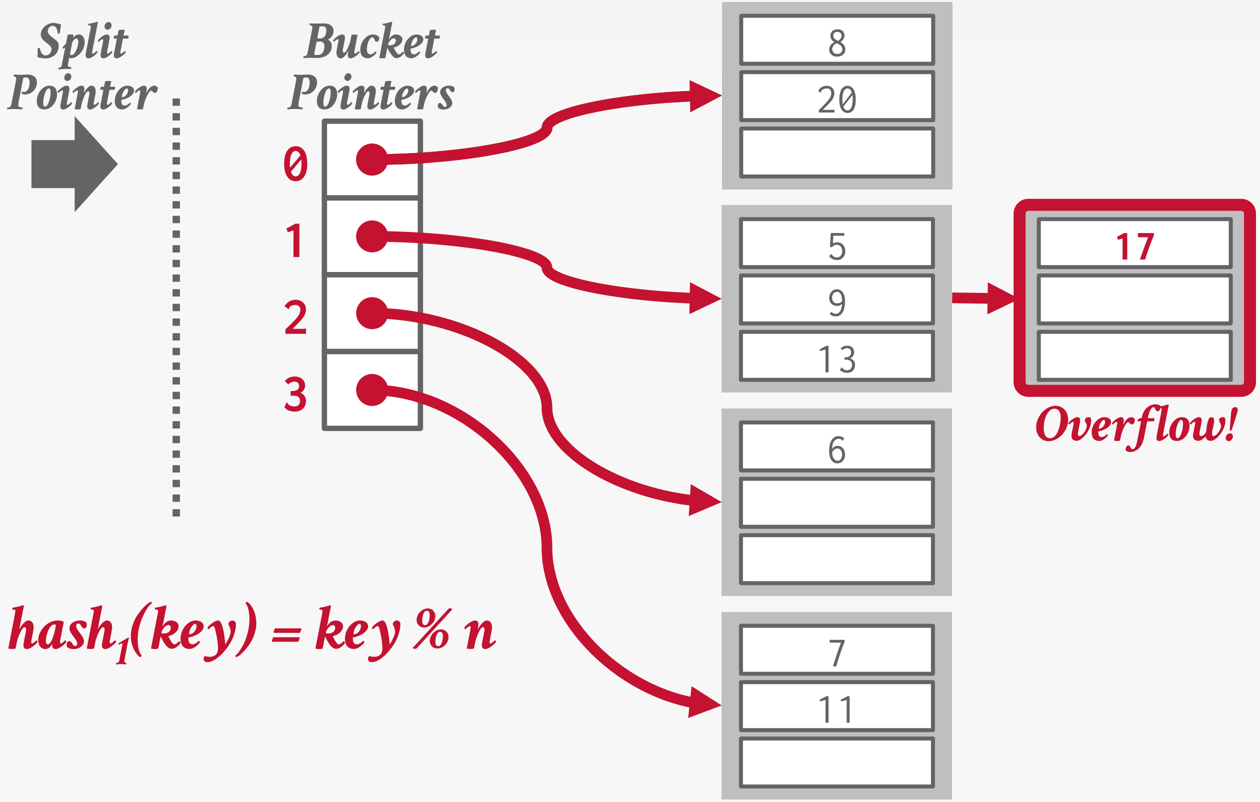
Get 6

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

Put 17

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

LINEAR HASHING



Get 6

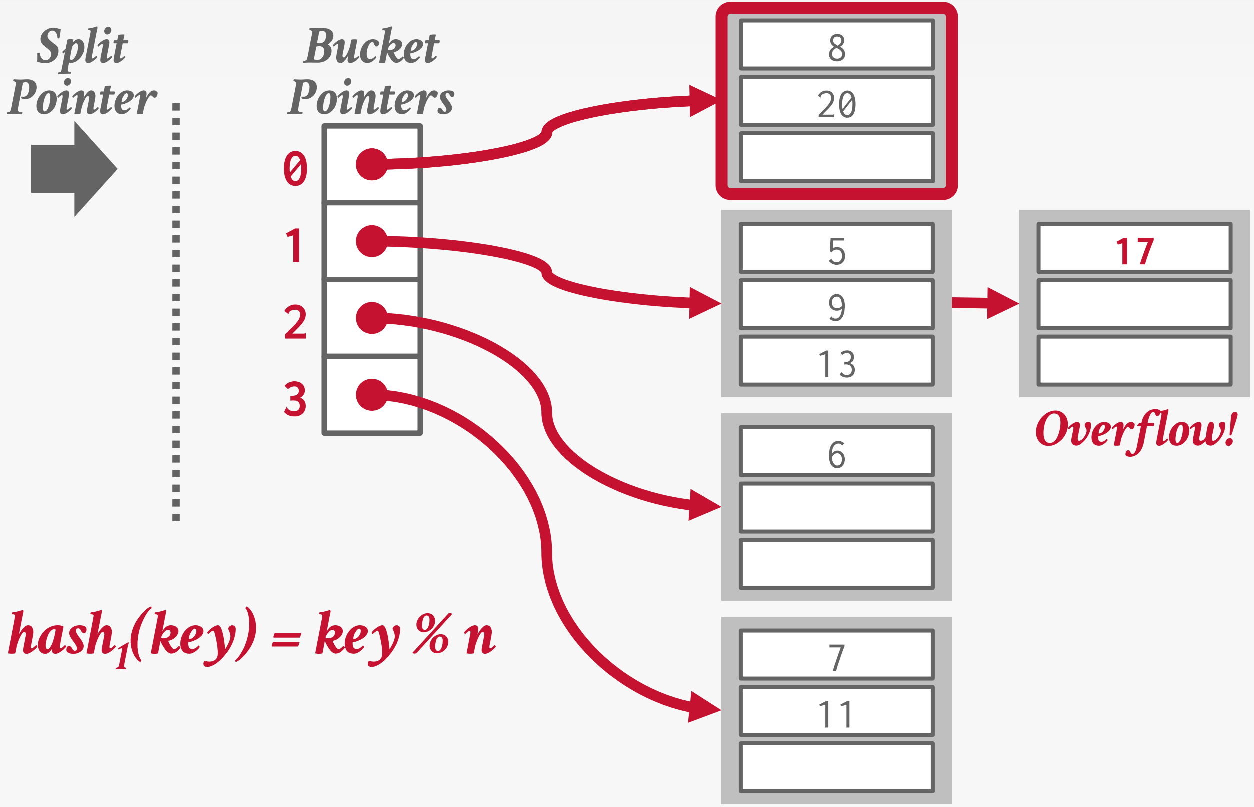
$hash_1(6) = 6 \% 4 = 2$

Put 17

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

$hash_1(key) = key \% n$

LINEAR HASHING



$hash_1(key) = key \% n$

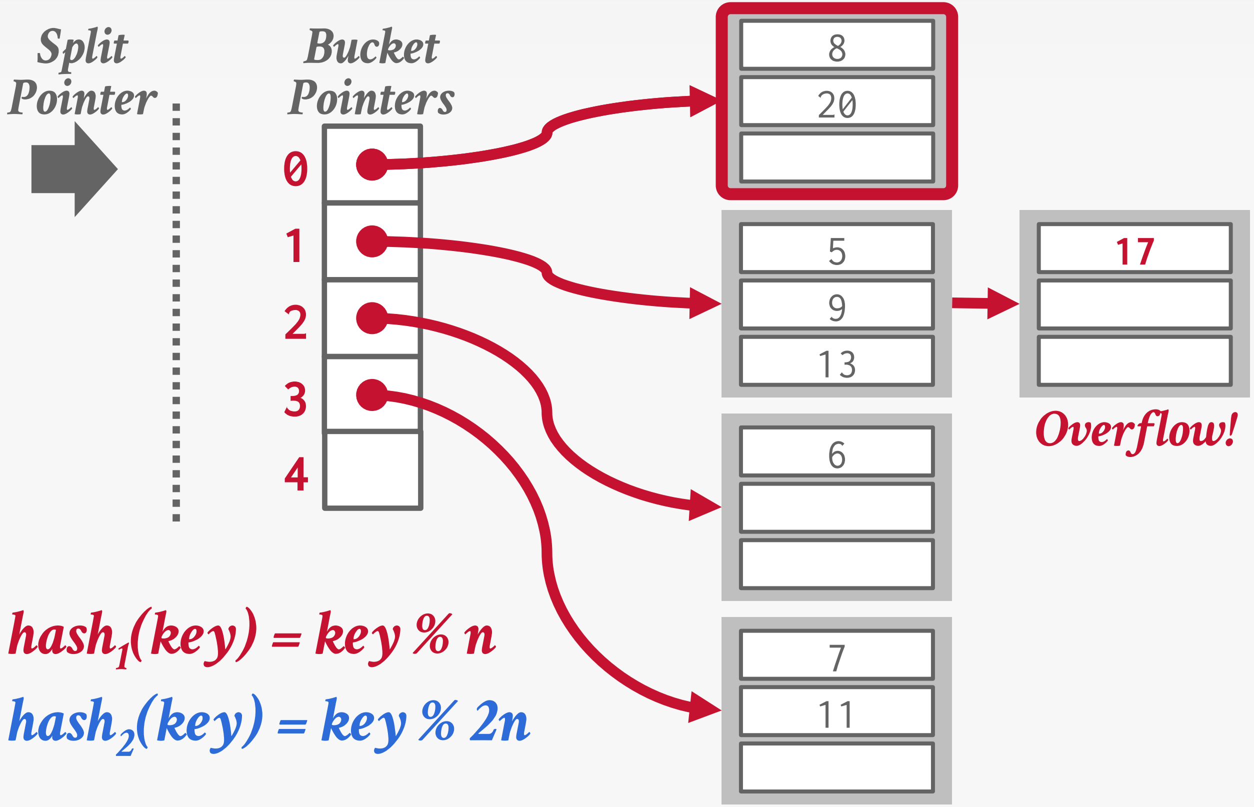
Get 6

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

Put 17

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

LINEAR HASHING



Get 6

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

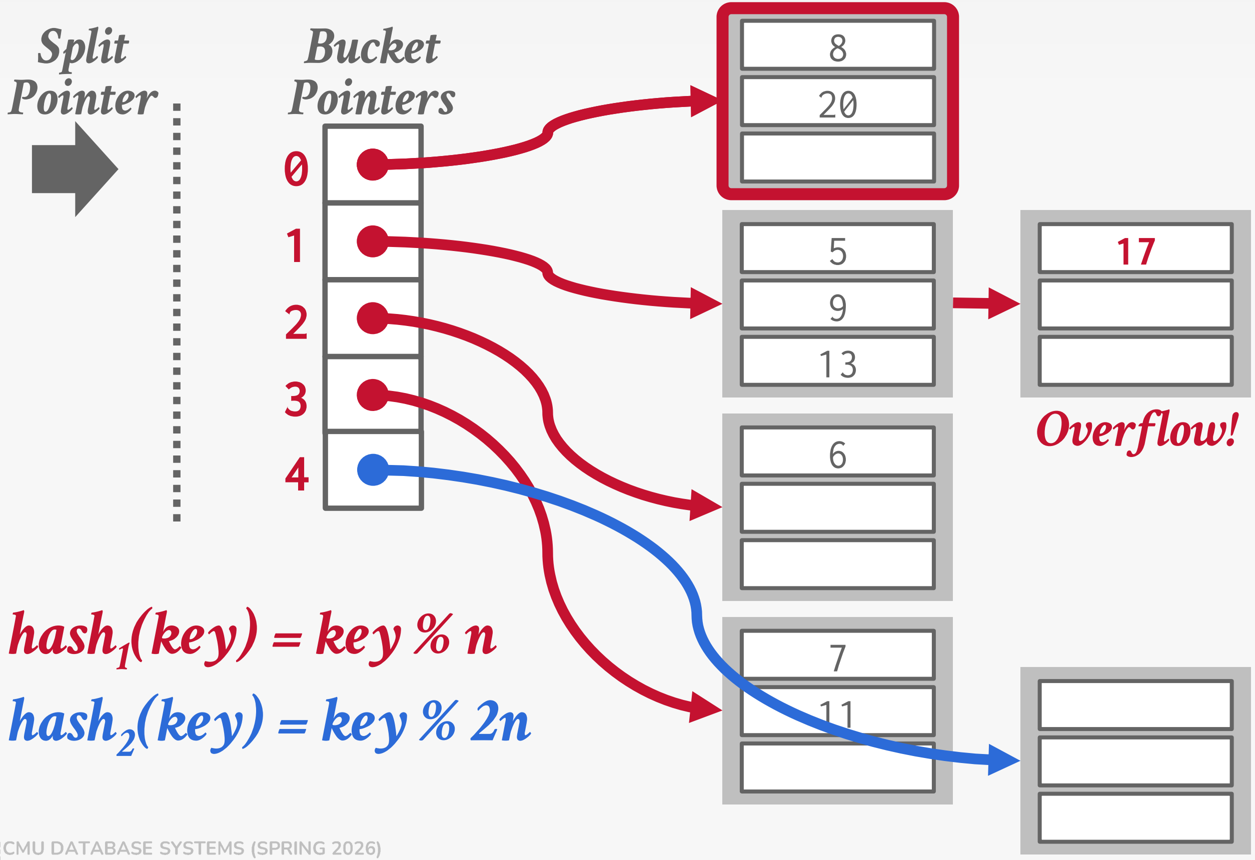
Put 17

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

$hash_1(key) = key \% n$

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

LINEAR HASHING

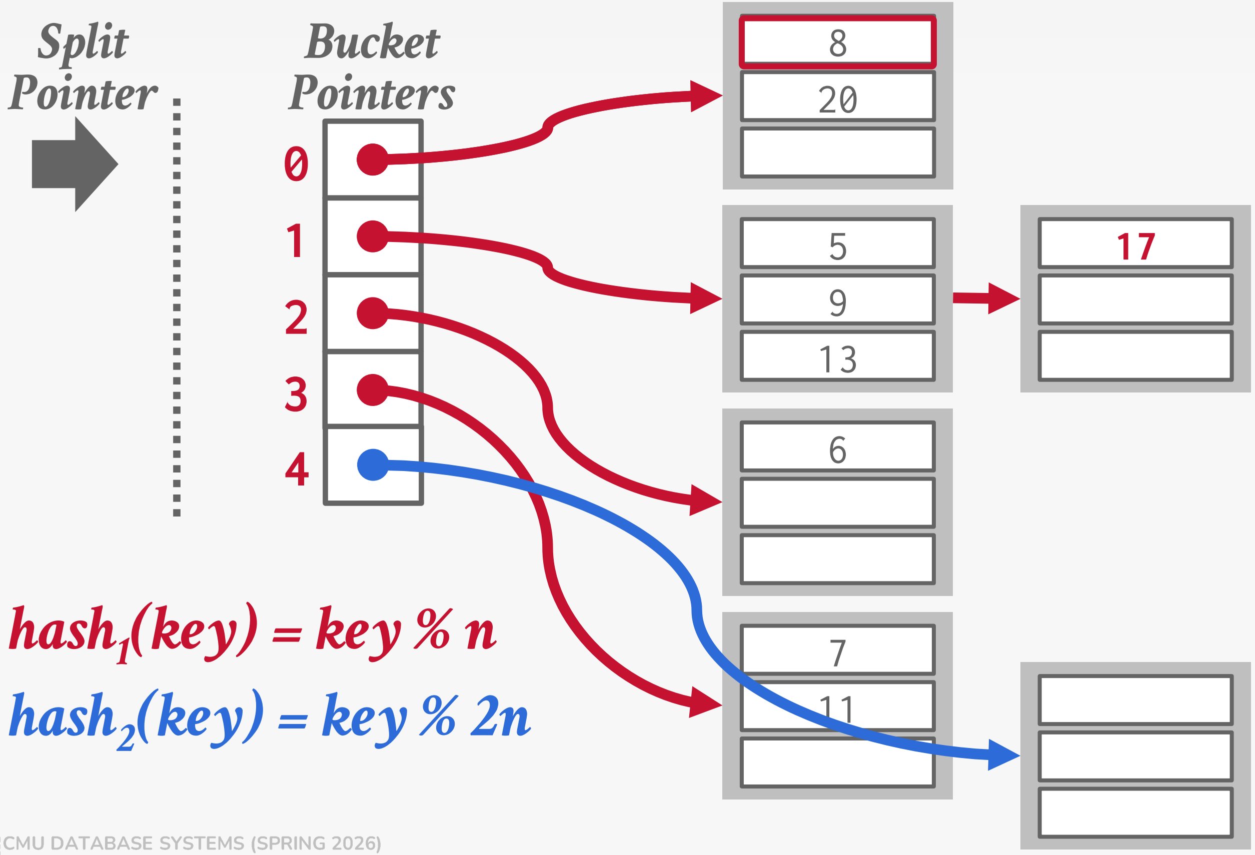


Get 6
$hash_1(6) = 6 \% 4 = 2$

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

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

LINEAR HASHING



Get 6

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

Put 17

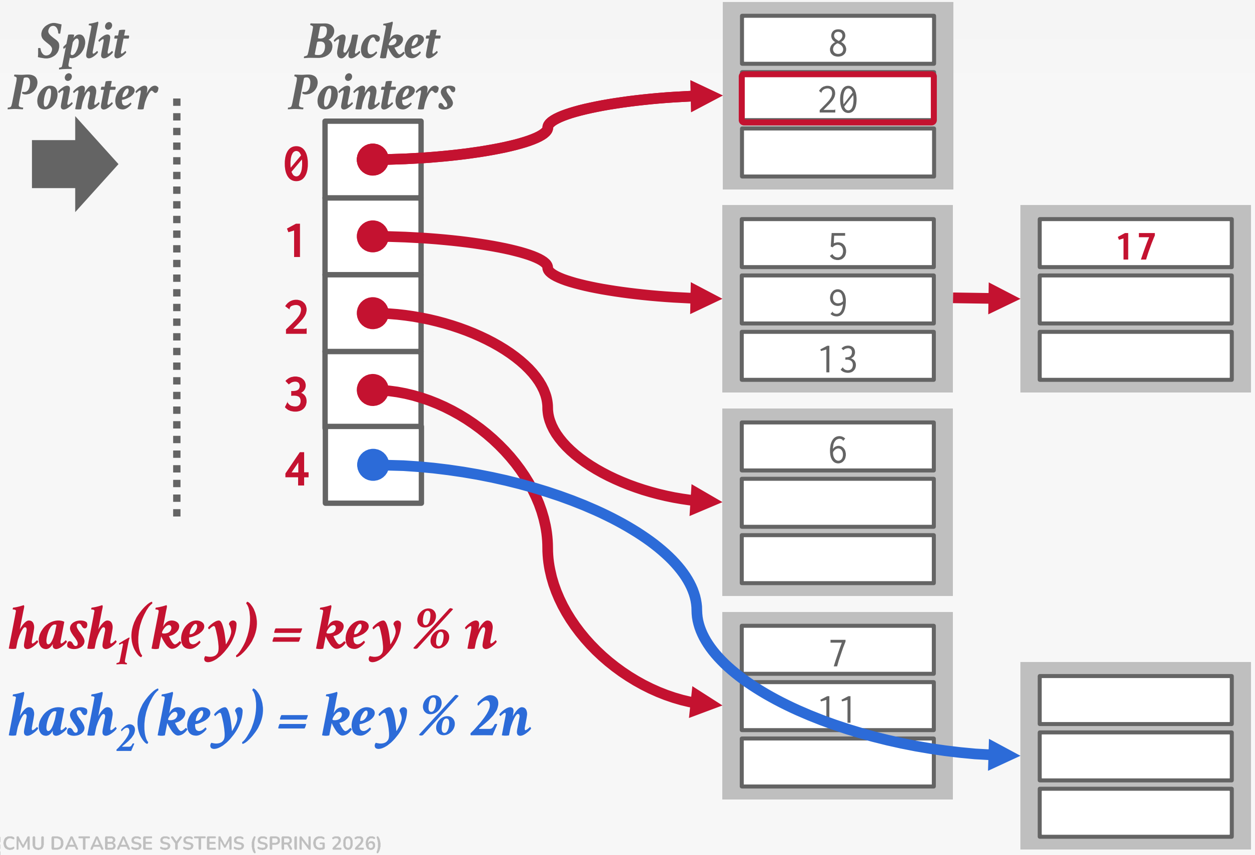
$hash_1(17) = 17 \% 4 = 1$

$hash_2(8) = 8 \% 8 = 0$

$hash_1(key) = key \% n$

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

LINEAR HASHING



Get 6

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

Put 17

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

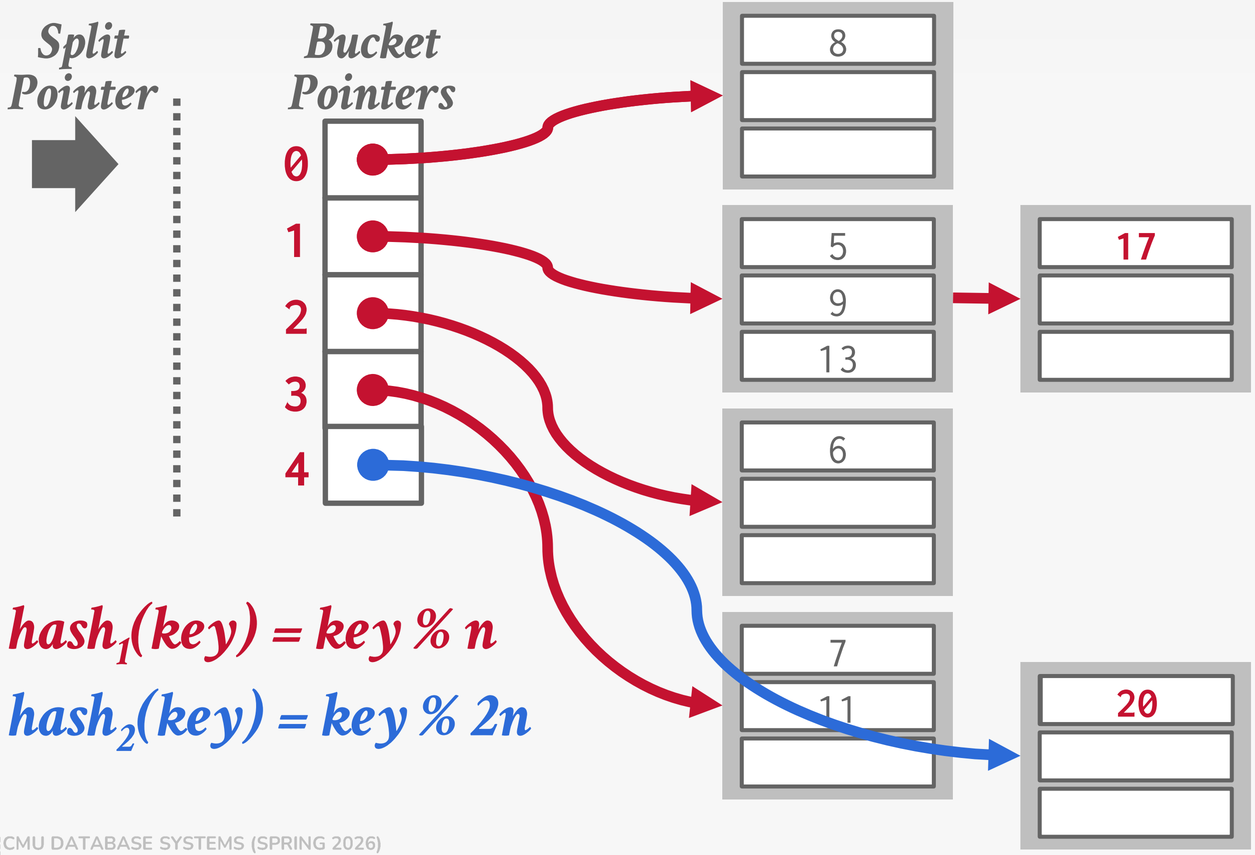
$hash_2(8) = 8 \% 8 = 0$

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

$hash_1(key) = key \% n$

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

LINEAR HASHING



$hash_1(key) = key \% n$

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

Get 6

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

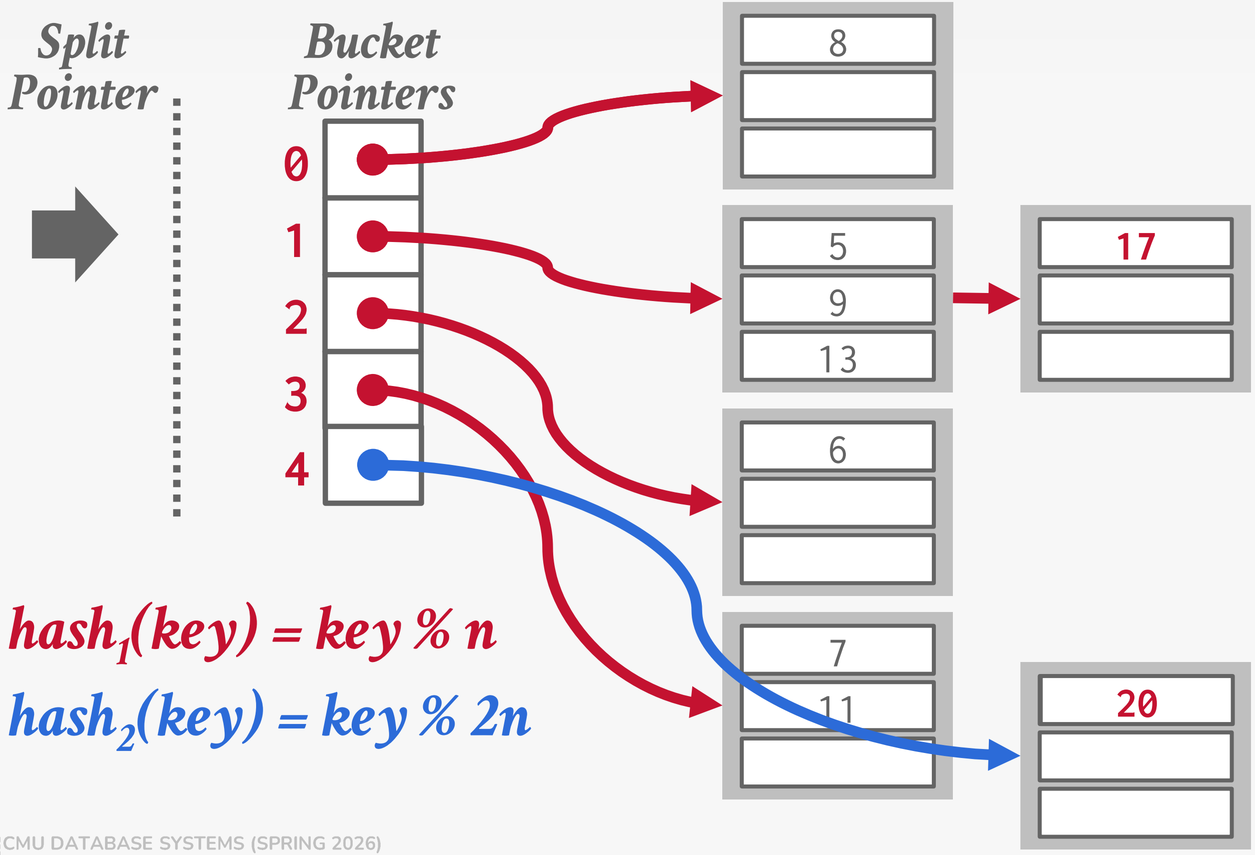
Put 17

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

$hash_2(8) = 8 \% 8 = 0$

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

LINEAR HASHING



$hash_1(key) = key \% n$

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

Get 6

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

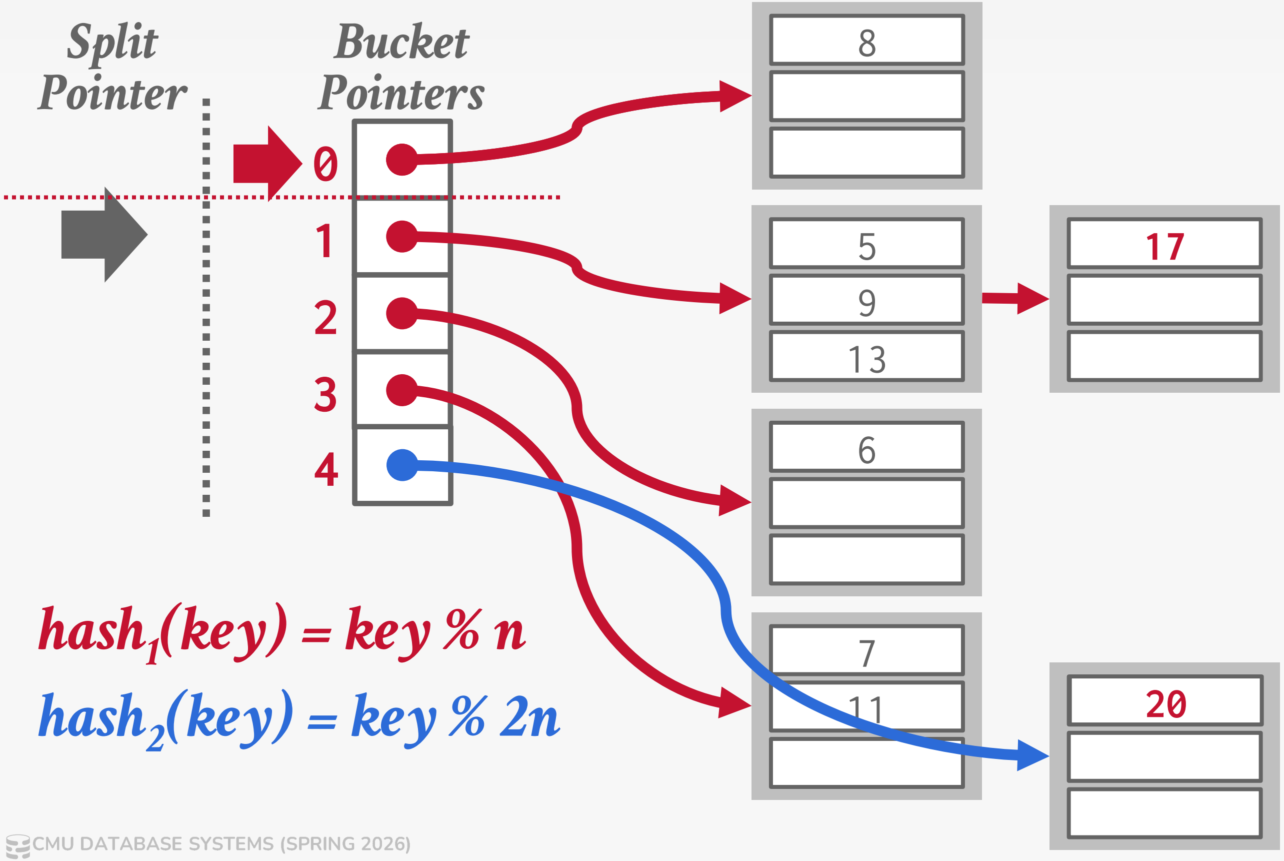
Put 17

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

$hash_2(8) = 8 \% 8 = 0$

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

LINEAR HASHING



$hash_1(key) = key \% n$

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

Get 6

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

Put 17

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

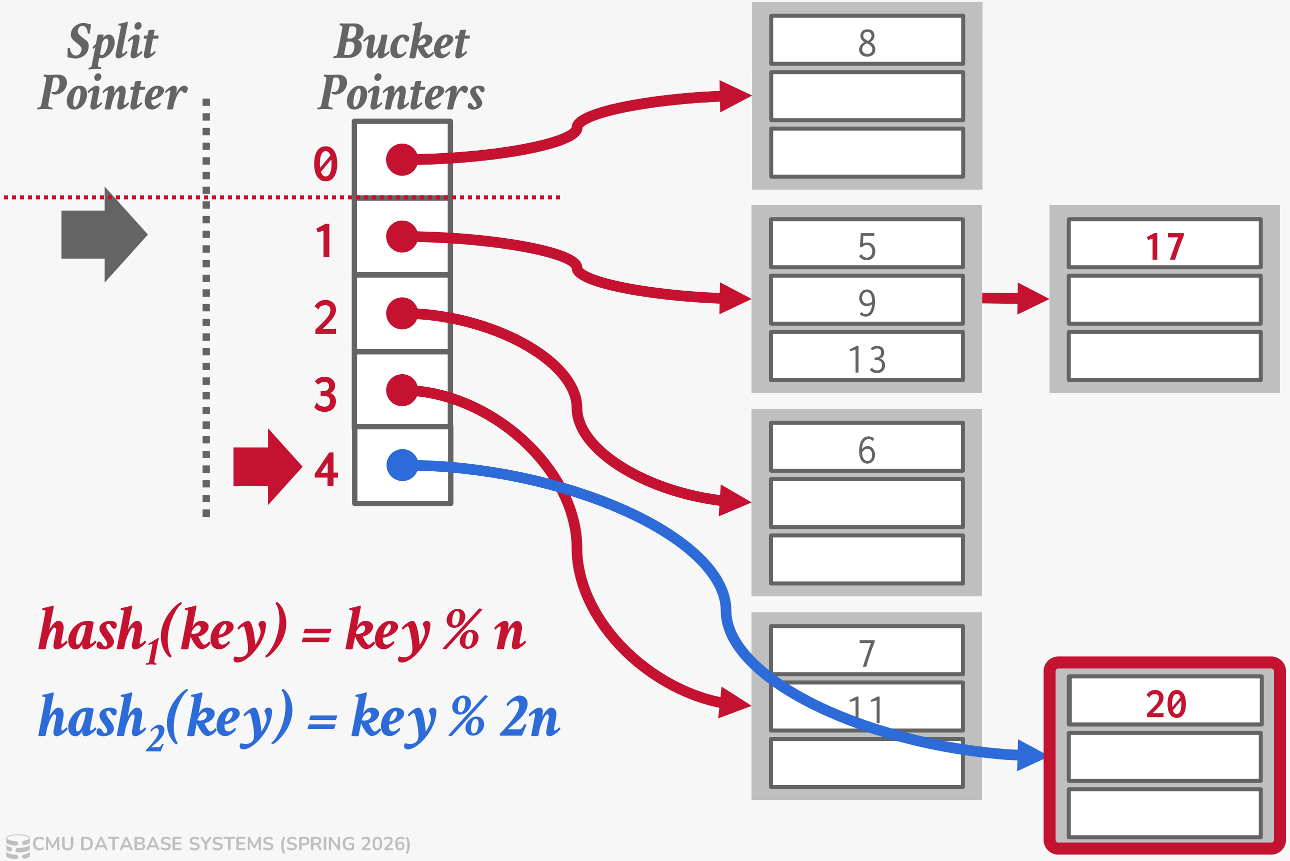
$hash_2(8) = 8 \% 8 = 0$

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

Get 20

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

LINEAR HASHING



$hash_1(key) = key \% n$

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

Get 6

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

Put 17

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

$hash_2(8) = 8 \% 8 = 0$

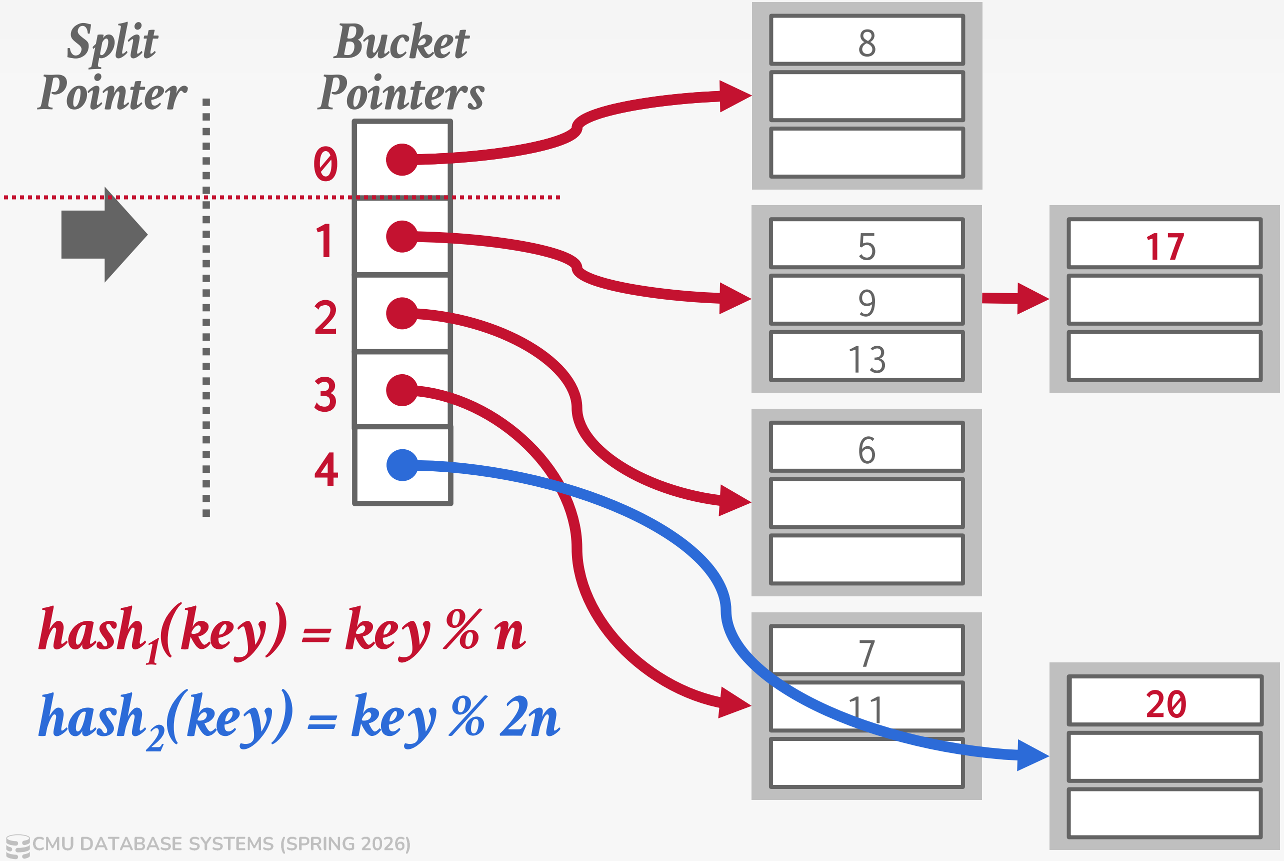
$hash_2(20) = 20 \% 8 = 4$

Get 20

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

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

LINEAR HASHING



$hash_1(key) = key \% n$

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

Get 6

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

Put 17

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

$hash_2(8) = 8 \% 8 = 0$

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

Get 20

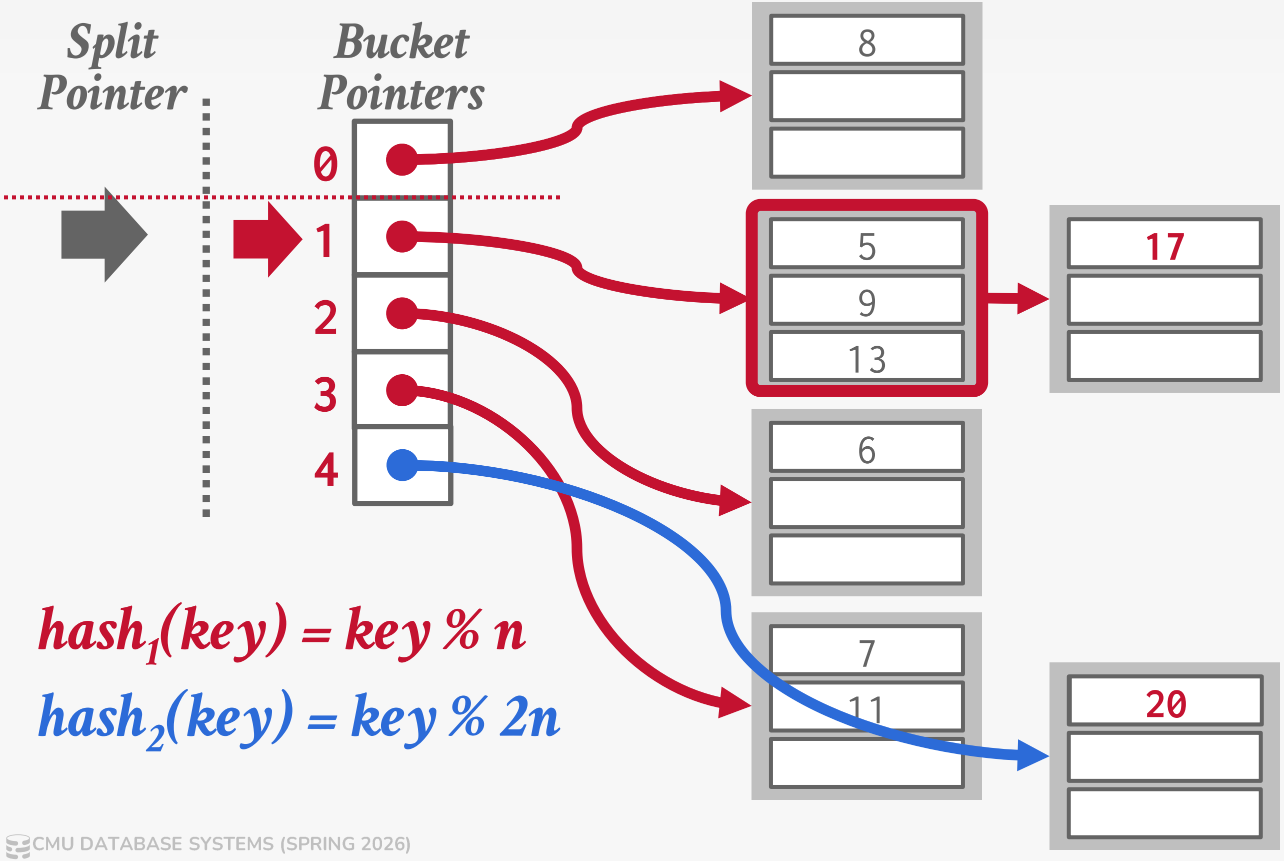
$hash_1(20) = 20 \% 4 = 0$

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

Get 9

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

LINEAR HASHING



$hash_1(key) = key \% n$

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

Get 6

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

Put 17

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

$hash_2(8) = 8 \% 8 = 0$

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

Get 20

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

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

Get 9

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

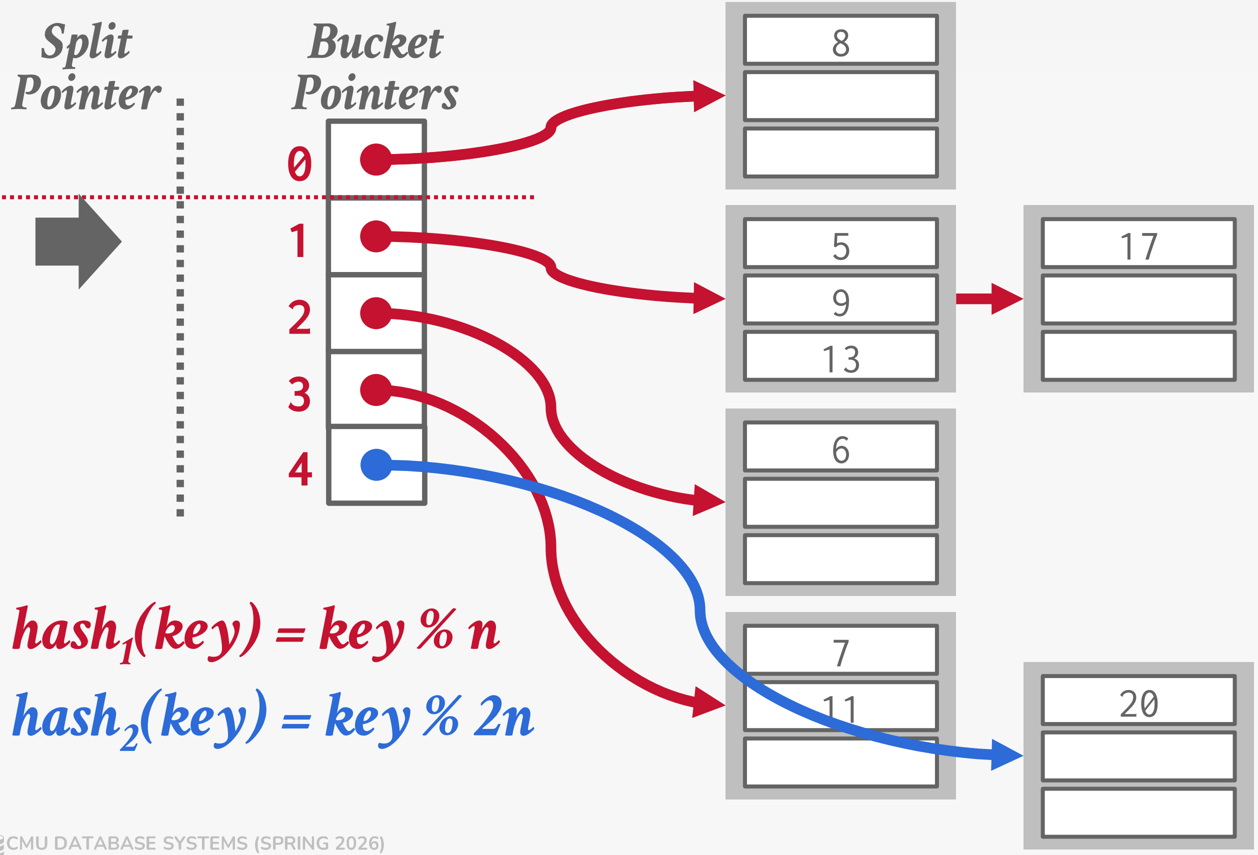
LINEAR HASHING: RESIZING

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

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

If the "highest" bucket below the split pointer is empty, the hash table could remove it and move the splinter pointer in reverse direction.

LINEAR HASHING: DELETES



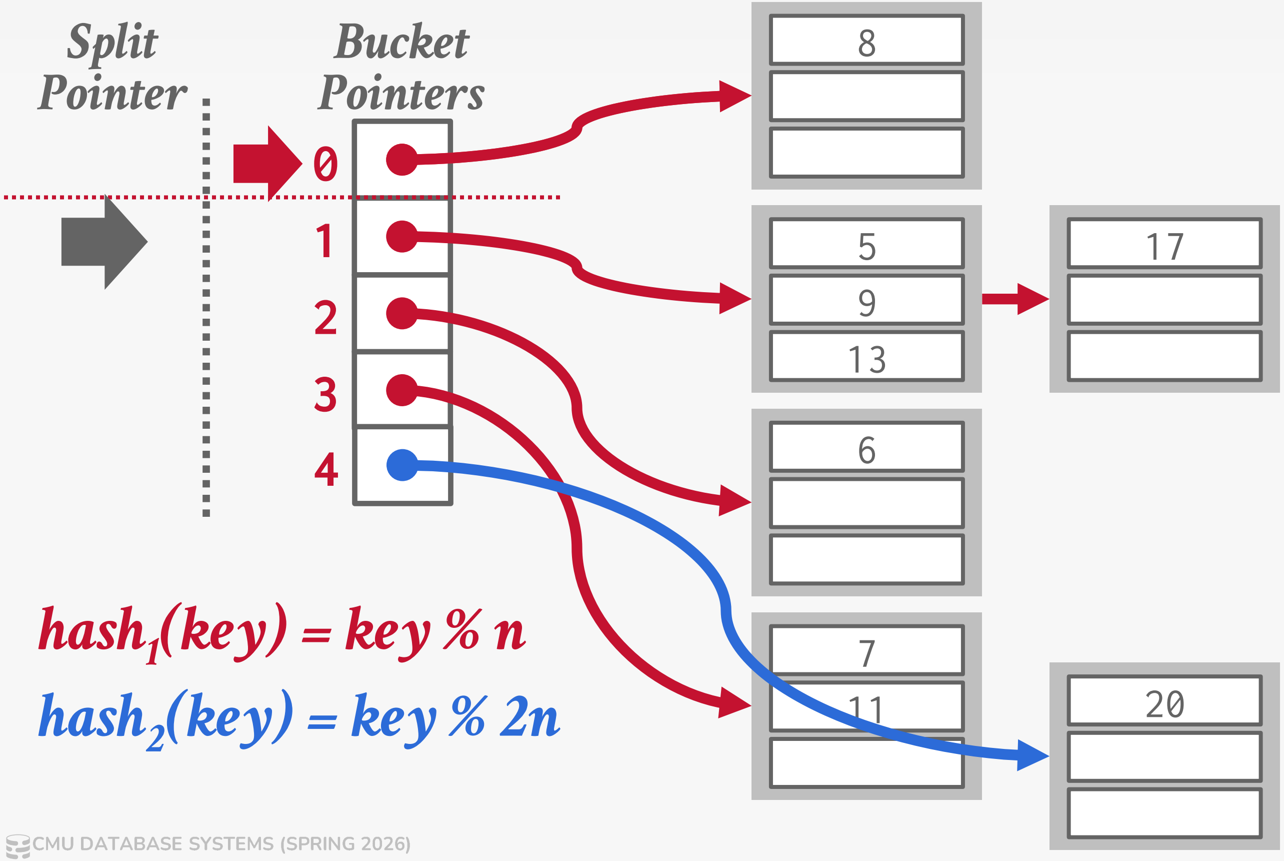
Delete 20

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

$hash_1(key) = key \% n$

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

LINEAR HASHING: DELETES



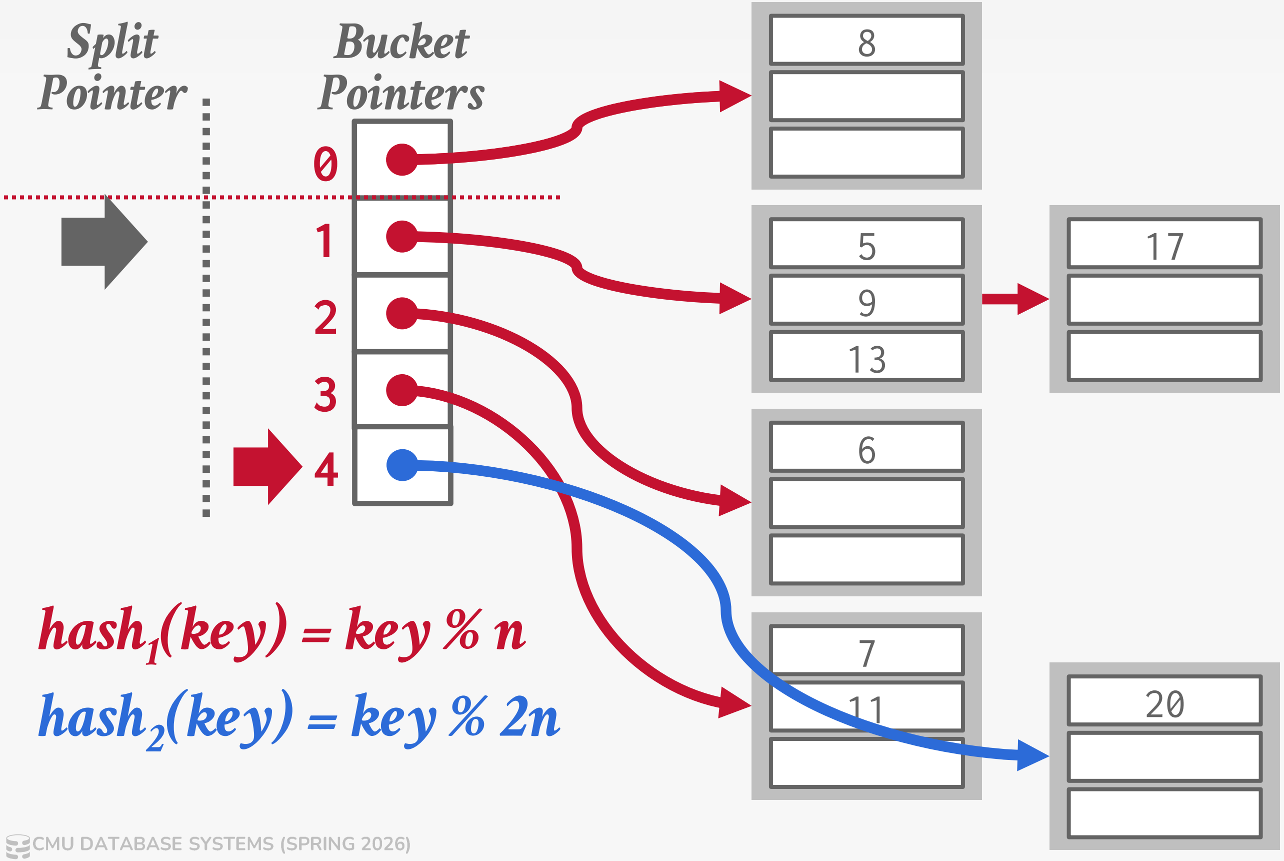
Delete 20

hash₁(20) = 20 % 4 = 0

$hash_1(key) = key \% n$

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

LINEAR HASHING: DELETES



Delete 20

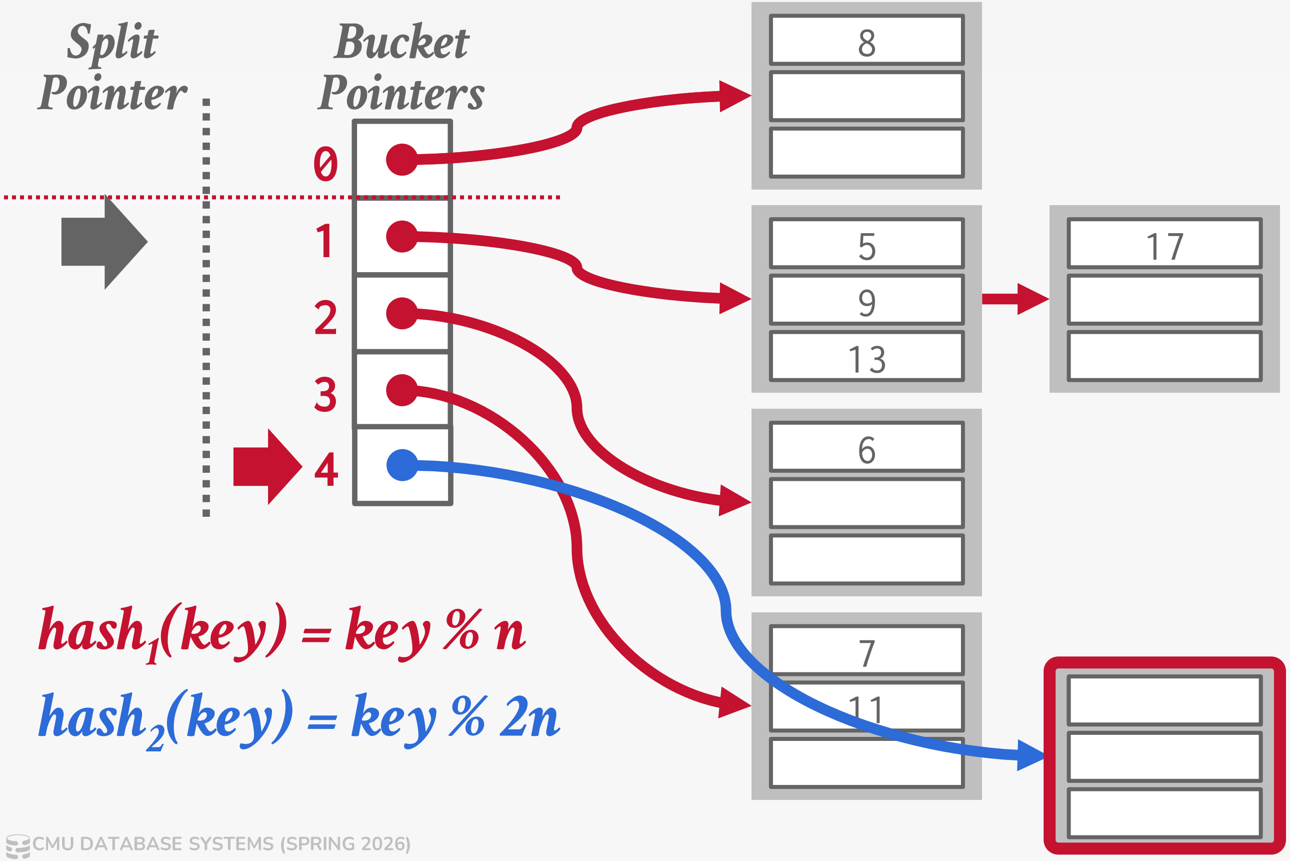
$hash_1(20) = 20 \% 4 = 0$

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

$hash_1(key) = key \% n$

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

LINEAR HASHING: DELETES



Delete 20

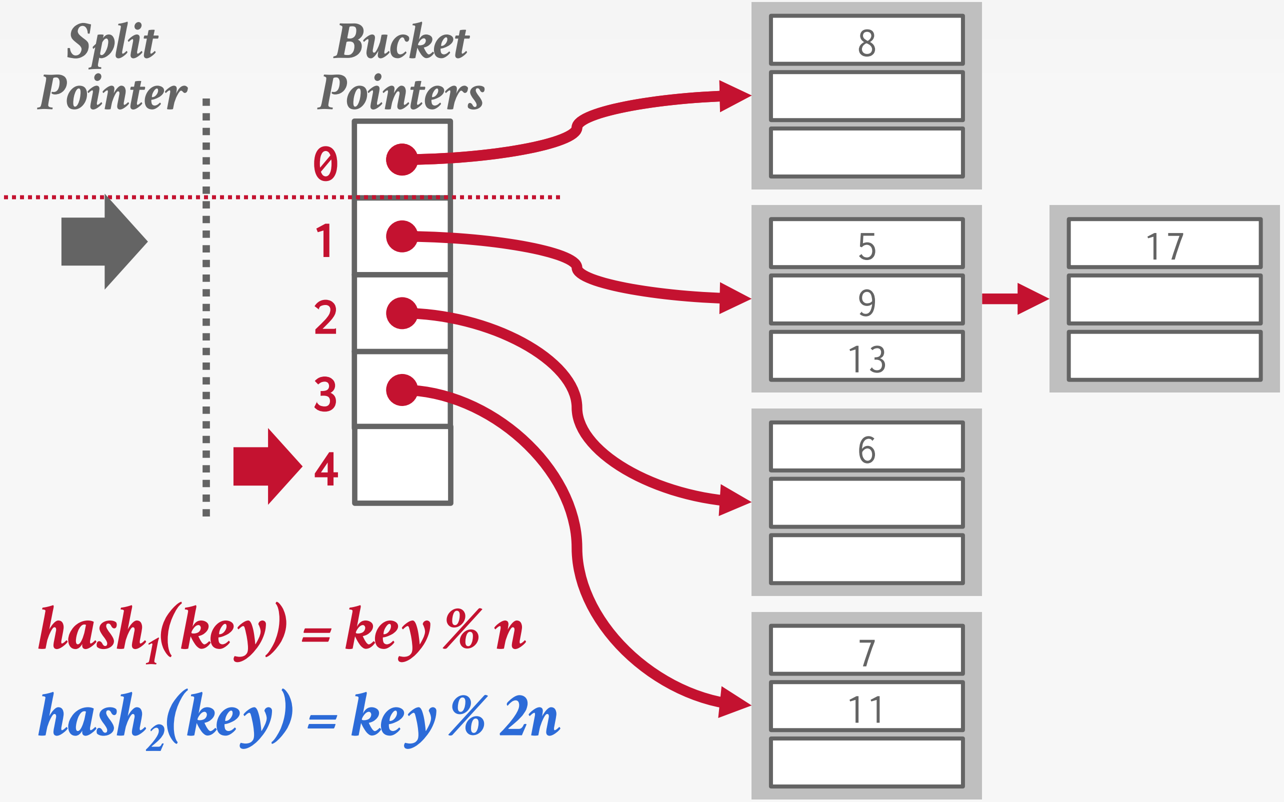
$hash_1(20) = 20 \% 4 = 0$

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

$hash_1(key) = key \% n$

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

LINEAR HASHING: DELETES



Delete 20

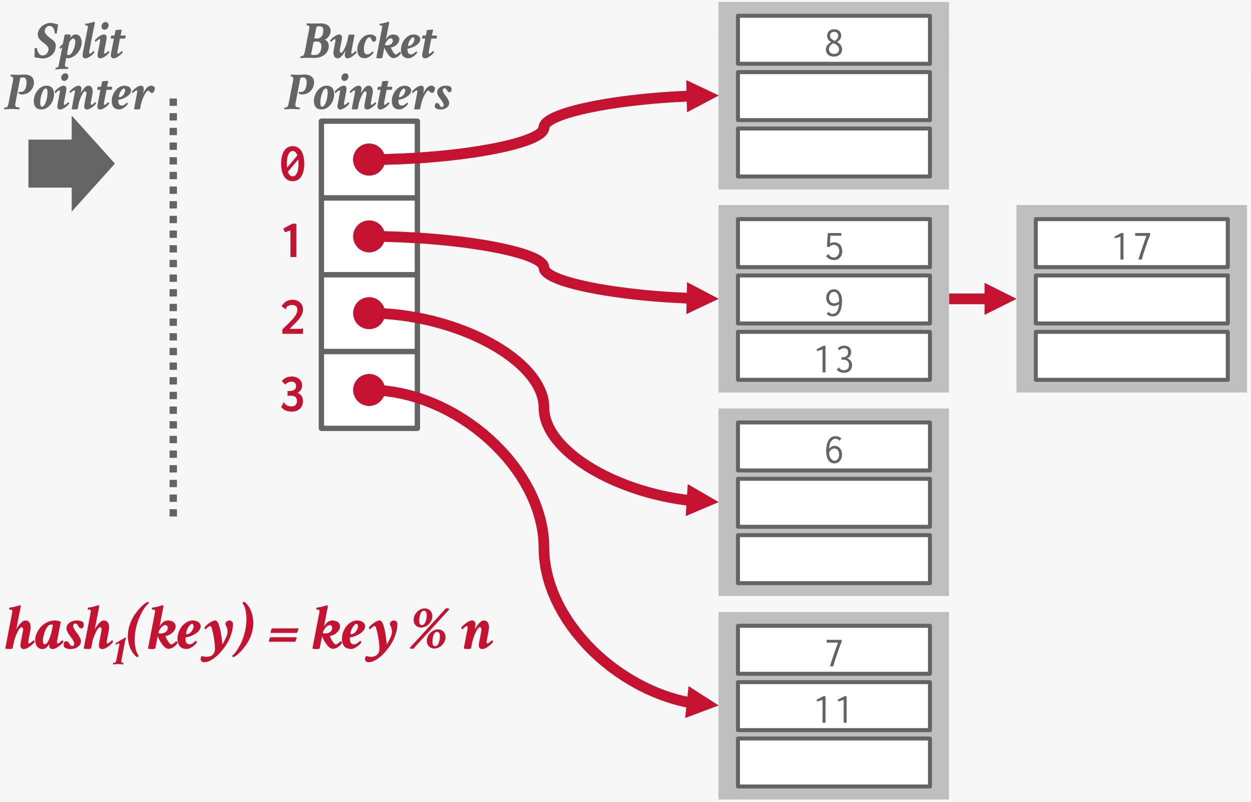
$hash_1(20) = 20 \% 4 = 0$

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

$hash_1(key) = key \% n$

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

LINEAR HASHING: DELETES



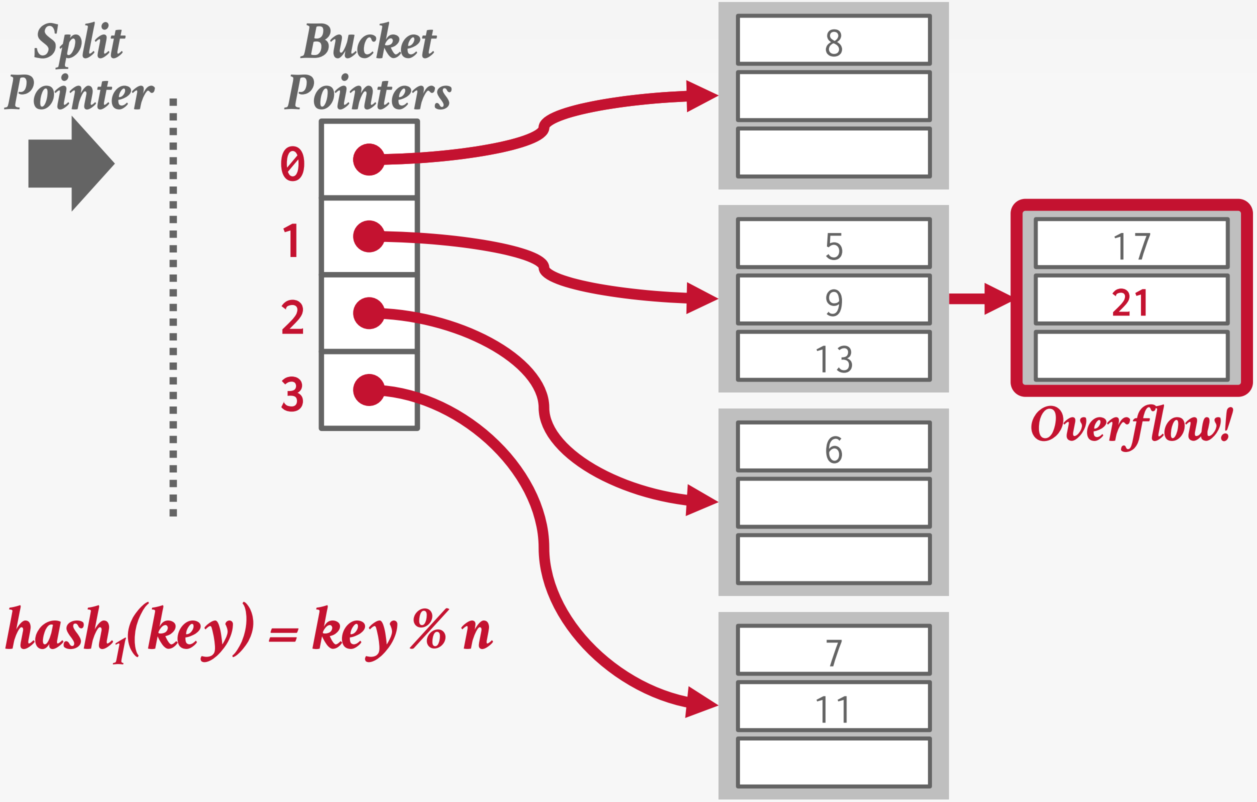
Delete 20

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

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

$hash_1(key) = key \% n$

LINEAR HASHING: DELETES



Delete 20

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

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

Put 21

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

$hash_1(key) = key \% n$

CONCLUSION

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

→ Trade-off between speed and flexibility.

→ Some DBMSs store all data in hash tables (key/value stores).

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

```
CREATE INDEX ON xxx (val);
```

```
CREATE INDEX ON xxx USING BTREE (val);
```

```
CREATE INDEX ON xxx USING HASH (val);
```

NEXT CLASS

Order-Preserving Indexes ft. B+Trees

→ aka "The Greatest Data Structure of All Time"