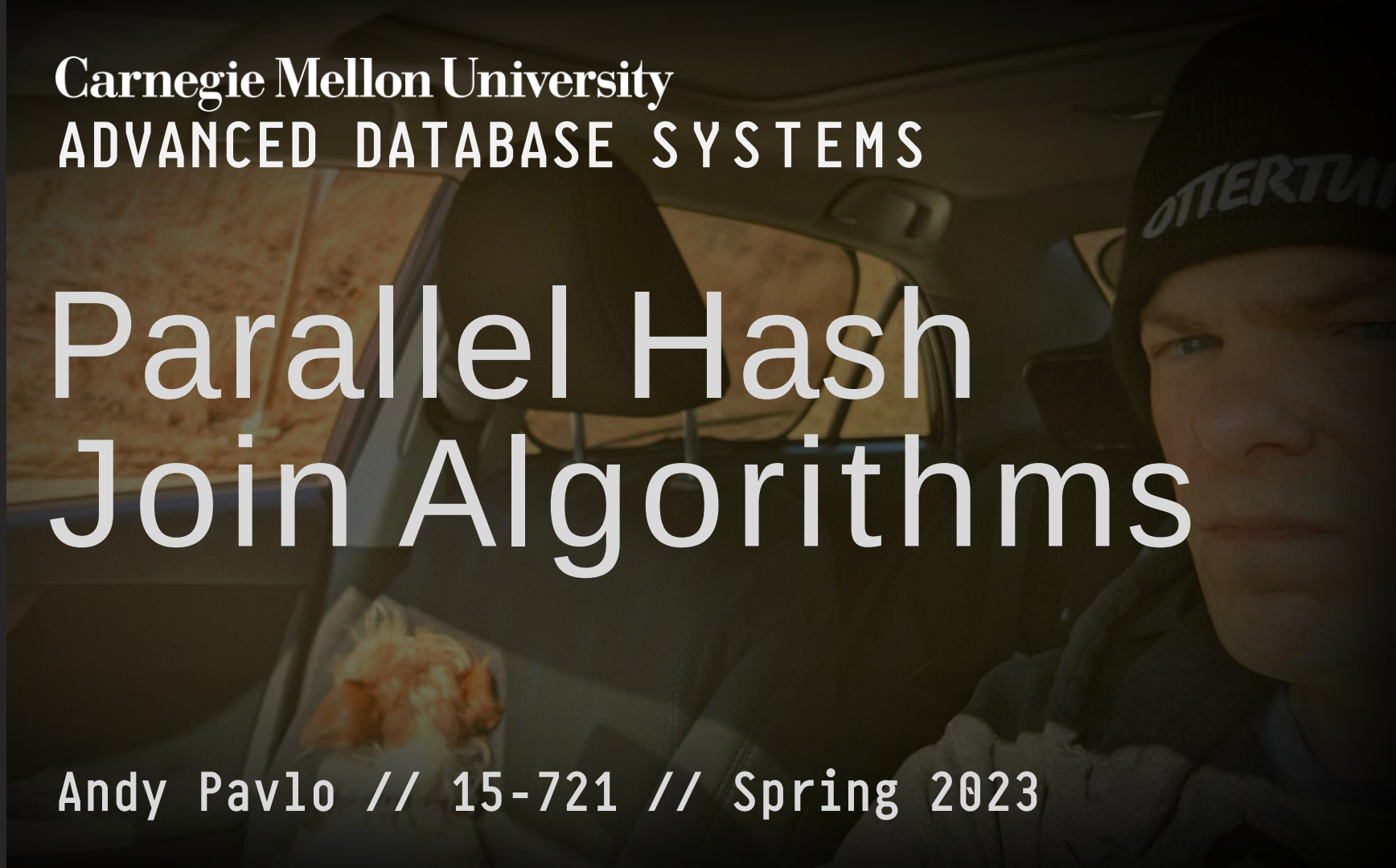
Lecture #11

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

# Parallel Hash Join Algorithms



Andy Pavlo // 15-721 // Spring 2023

# ADMINISTRIVIA

## Project #2:

→ Feedback Submission: **Saturday April 1st**
→ Final Submission: **Monday May 1st**
→ Sign up for a system if you haven't yet!

## Project #3

→ Proposal Presentation: **Wednesday March 1st**
→ Status Update Presentation: **Monday April 3rd**
→ Final Presentations: TBA

# TODAY'S AGENDA

Background

Parallel Hash Join

Hash Functions

Hashing Schemes

Evaluation

# PARALLEL JOIN ALGORITHMS

Perform a join between two relations on multiple threads simultaneously to speed up operation.
→ We will discuss multi-way joins in Lecture #13.

Two main approaches:
→ **Hash Join**
→ **Sort-Merge Join**

We won't discuss nested-loop joins because an OLAP DBMS almost never wants to use this…

# OBSERVATION

Many OLTP DBMSs do not implement hash join.

But an **index nested-loop join** is conceptually equivalent to a hash join.

→ Index NL joins typically means using an existing B+Tree.
→ Hash join will build a hash table (index) on the fly and then discard immediately after the operation is complete.

# HASHING VS. SORTING JOINS

**1970s** – Sorting

**1980s** – Hashing

**1990s** – Equivalent

**2000s** – Hashing

**2010s** – Hashing (Partitioned vs. Non-Partitioned)

**2020s** – Non-Partitioned Hashing

# PARALLEL JOIN ALGORITHMS

SORT VS. HASH REVISITED: FAST JOIN IMPLEMENTATION ON MODERN MULTI-CORE CPUS
VLDB 2009

ORACLE intel

→ Hashing is faster than Sort-Merge.
→ Sort-Merge is faster w/ wider SIMD.

DESIGN AND EVALUATION OF MAIN MEMORY HASH JOIN ALGORITHMS FOR MULTI-CORE CPUS
SIGMOD 2011

WISCONSIN UNIVERSITY OF WISCONSIN-MADISON

→ Trade-offs between partitioning & non-partitioning Hash-Join.

MASSIVELY PARALLEL SORT-MERGE JOINS IN MAIN MEMORY MULTI-CORE DATABASE SYSTEMS
VLDB 2012

HyPer

→ Sort-Merge is already faster than Hashing, even without SIMD.

MASSIVELY PARALLEL NUMA-AWARE HASH JOINS
IMDM 2013

HyPer

→ Ignore what we said last year.
→ You really want to use Hashing!

MAIN-MEMORY HASH JOINS ON MULTI-CORE CPUS: TUNING TO THE UNDERLYING HARDWARE
ICDE 2013

Systems@ETH zürich

→ New optimizations and results for Radix Hash Join.

AN EXPERIMENTAL COMPARISON OF THIRTEEN RELATIONAL EQUI-JOINS IN MAIN MEMORY
SIGMOD 2016

UNIVERSITÄT DES SAARLANDES

→ Hold up everyone! Let's look at everything more carefully!

TO PARTITION, OR NOT TO PARTITION, THAT IS THE JOIN QUESTION IN A REAL SYSTEM
SIGMOD 2021

UMBRA

→ Benefits of Radix Hash Join aren't worth engineering costs.

# JOIN ALGORITHM DESIGN GOALS

These goals matter whether the DBMS is using a **hardware-conscious** vs. **hardware-oblivious** algorithm for joins.

**Goal #1: Minimize Synchronization**

→ Avoid taking latches during execution.

**Goal #2: Minimize Memory Access Cost**

→ Ensure that data is always local to worker thread.

→ Reuse data while it exists in CPU cache.

# IMPROVING CACHE BEHAVIOR

Factors that affect cache misses in a DBMS:

→ Cache + TLB capacity.

→ Locality (temporal and spatial).

**Non-Random Access (Scan):**

→ Clustering data to a cache line.

→ Execute more operations per cache line.

**Random Access (Lookups):**

→ Partition data to fit in cache + TLB.

Source: Johannes Gehrke

# PARALLEL HASH JOINS

Hash join is one of the most important operators in a DBMS for OLAP workloads.

→ But it is still not the dominant cost.

It is important that we speed up our DBMS's join algorithm by taking advantage of multiple cores.

→ We want to keep all cores busy, without becoming memory bound.

# HASH JOIN (R⋈S)

## Phase #1: Partition (*optional*)

→ Divide the tuples of **R** and **S** into disjoint subsets using a hash function on the join key.

## Phase #2: Build

→ Scan relation **R** and create a hash table on join key.

## Phase #3: Probe

→ For each tuple in **S**, look up its join key in hash table for **R**. If a match is found, output combined tuple.

AN EXPERIMENTAL COMPARISON OF THIRTEEN RELATIONAL EQUI-JOINS IN MAIN MEMORY
SIGMOD 2016

# PARTITIONING PHASE

## Approach #1: Implicit Partitioning

→ The data was partitioned on the join key when it was loaded into the database.
→ No extra pass over the data is needed.

## Approach #2: Explicit Partitioning

→ Divide only the outer relation and redistribute among the different CPU cores.
→ Can use the same radix partitioning approach we talked about last time.

# PARTITION PHASE

Split the input relations into partitioned buffers by hashing the tuples' join key(s).

→ Ideally the cost of partitioning is less than the cost of cache misses during build phase.

→ Sometimes called ***Grace Hash Join / Radix Hash Join.***

Contents of buffers depends on storage model:

→ **NSM**: Usually the entire tuple.

→ **DSM**: Only the columns needed for the join + offset.

# PARTITION PHASE

## Approach #1: Non-Blocking Partitioning

→ Only scan the input relation once.
→ Produce output incrementally and let other threads build hash table at the same time.

## Approach #2: Blocking Partitioning (Radix)

→ Scan the input relation multiple times.
→ Only materialize results all at once.
→ Sometimes called ***radix hash join***.

# NON-BLOCKING PARTITIONING

Scan the input relation only once and generate the output on-the-fly.

**Approach #1: Shared Partitions**

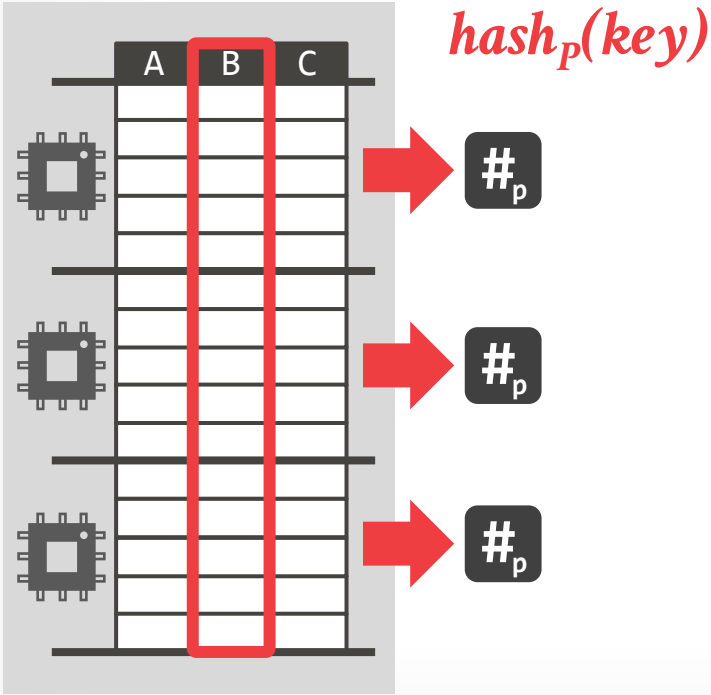
→ Single global set of partitions that all threads update.
→ Must use a latch to synchronize threads.

**Approach #2: Private Partitions**

→ Each thread has its own set of partitions.
→ Must consolidate them after all threads finish.

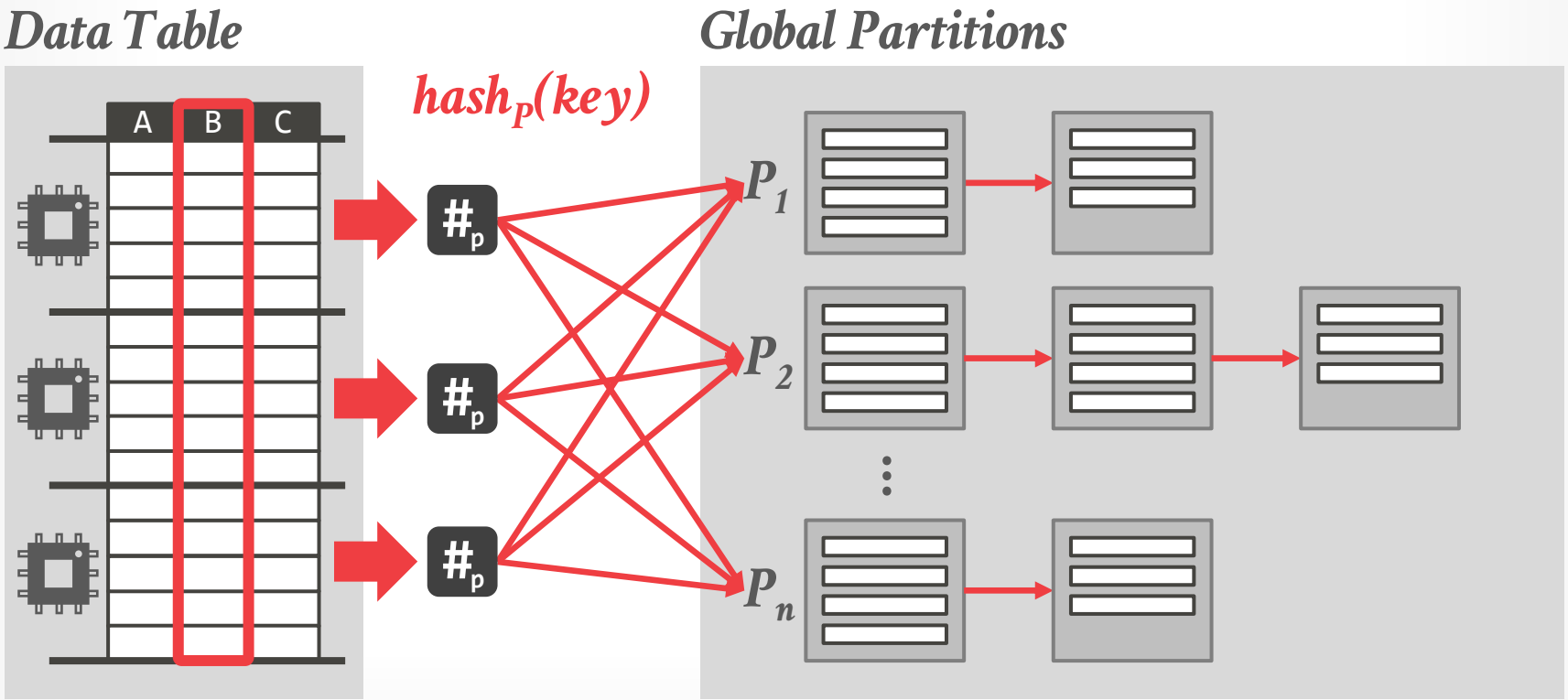
# SHARED PARTITIONS

*Data Table*

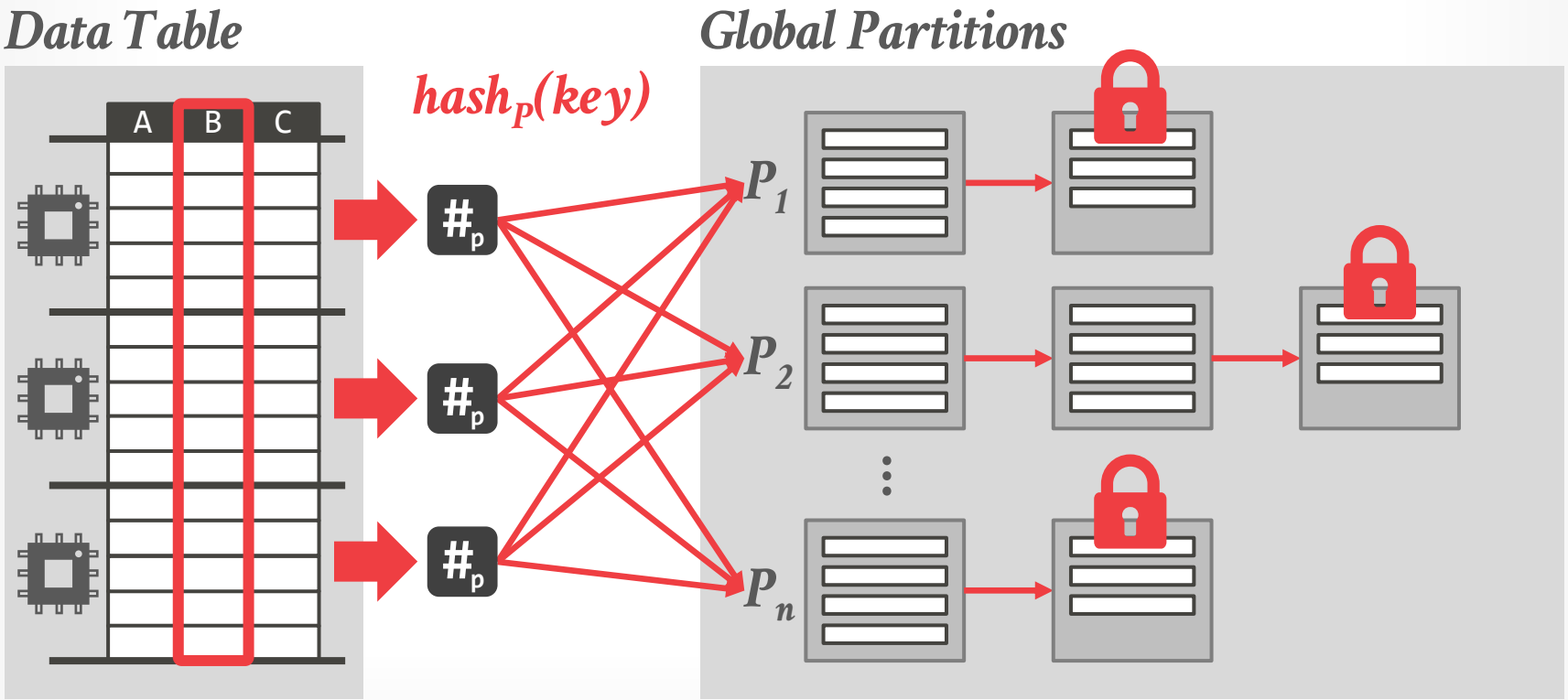



$hash_P(key)$

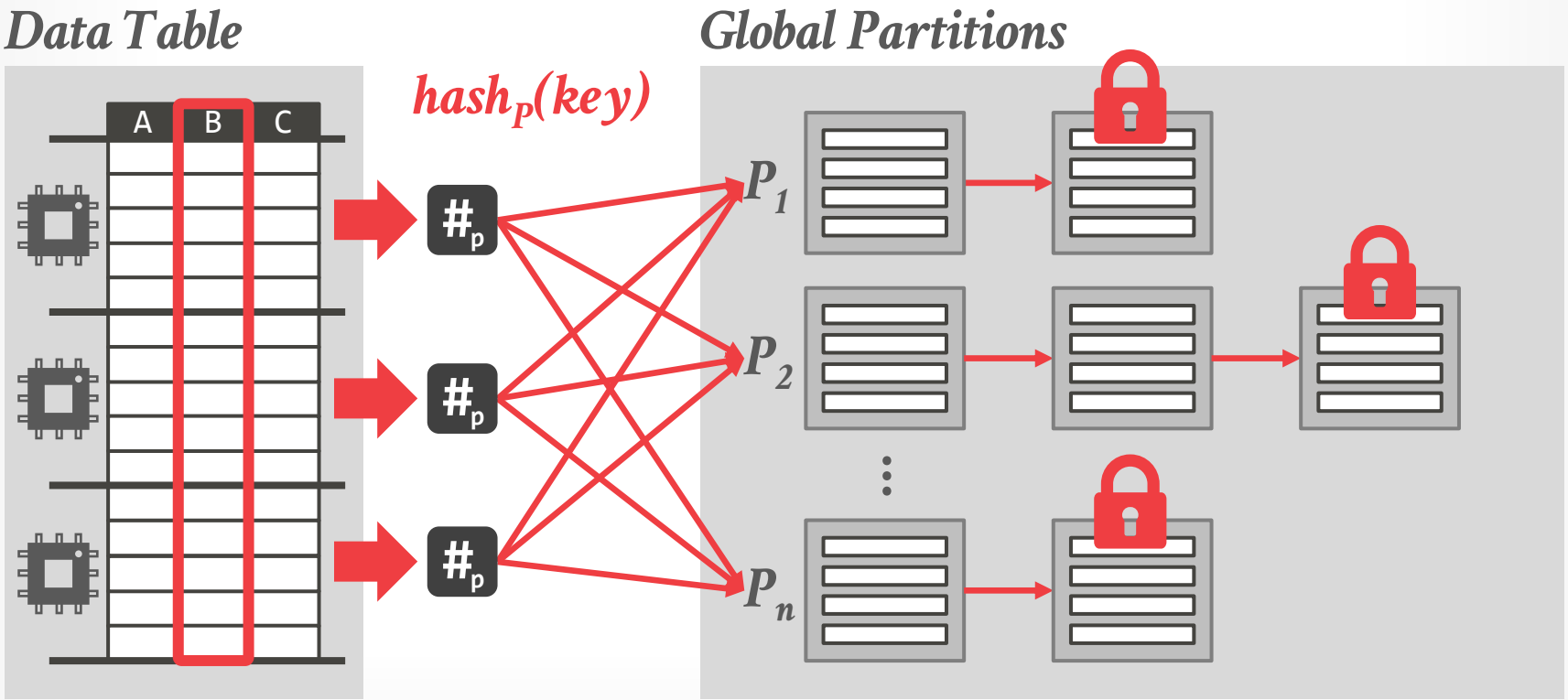
# SHARED PARTITIONS



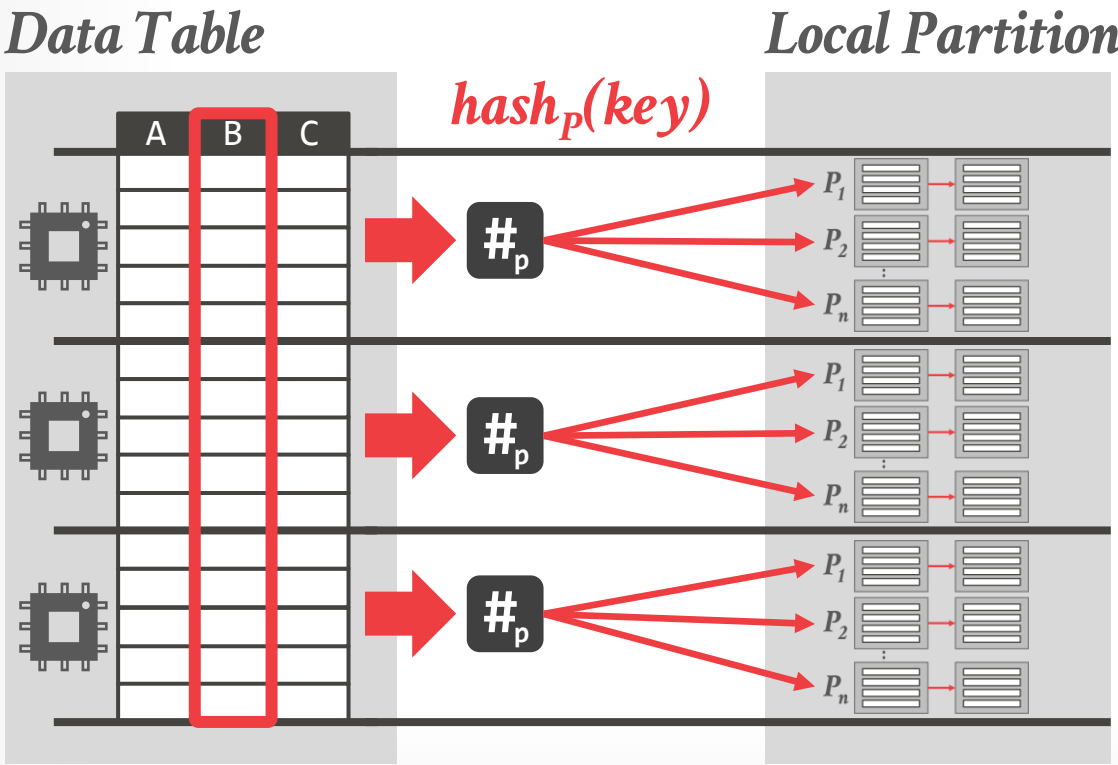
# SHARED PARTITIONS



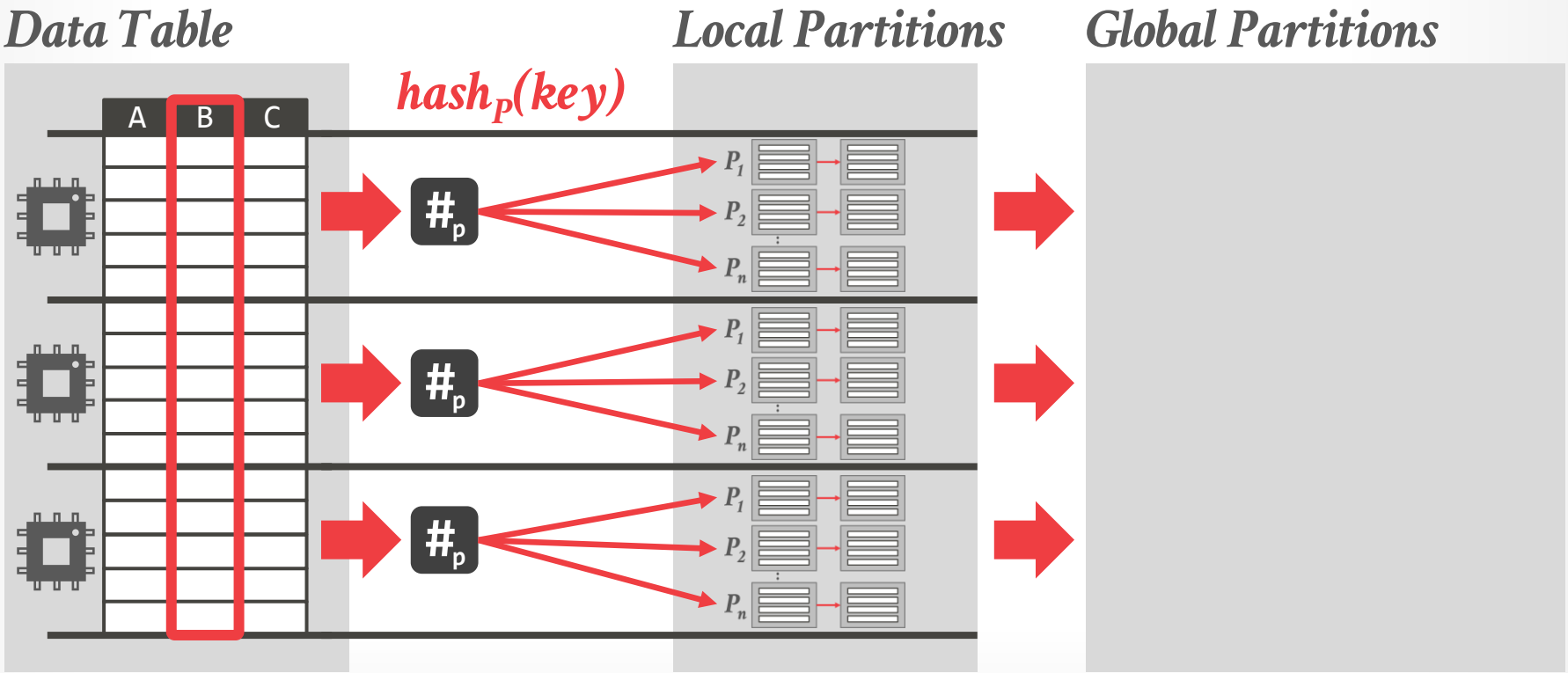
# SHARED PARTITIONS



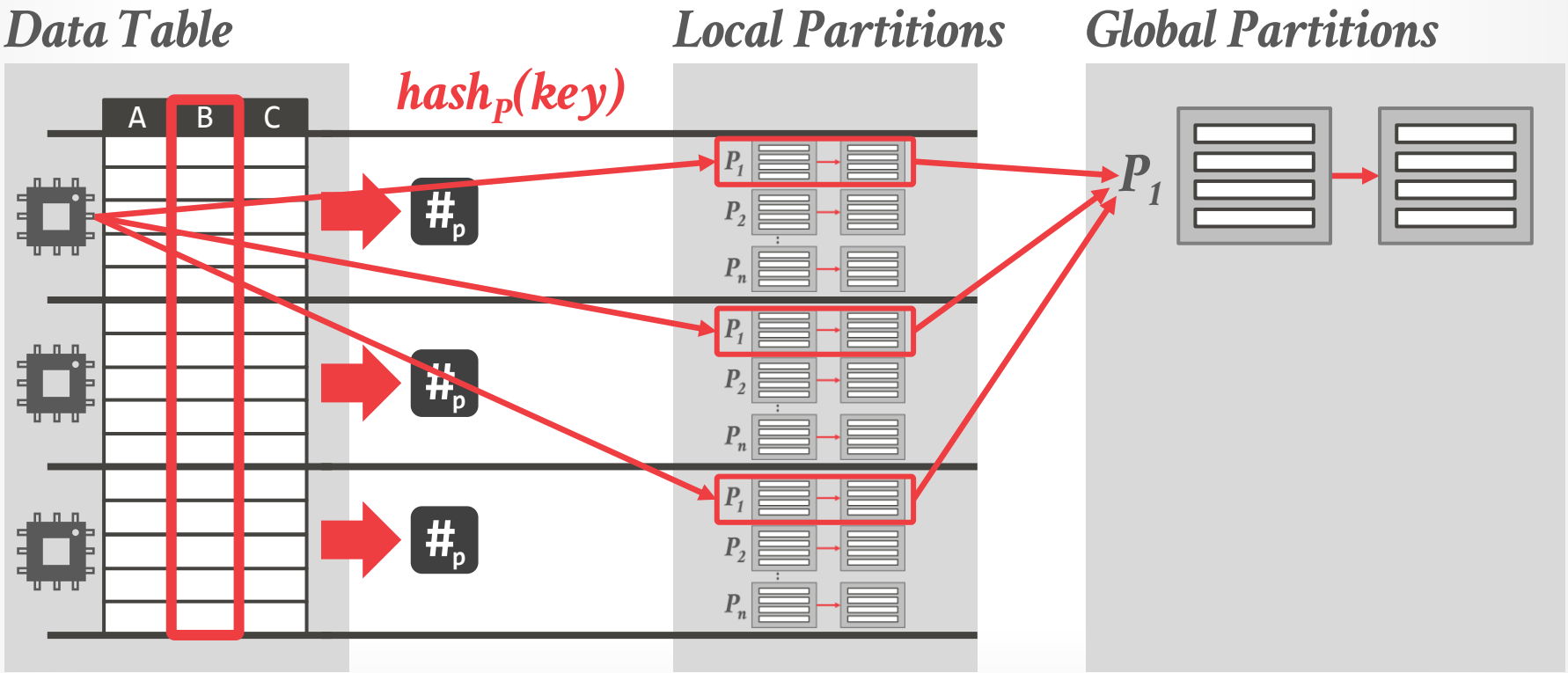
# PRIVATE PARTITIONS



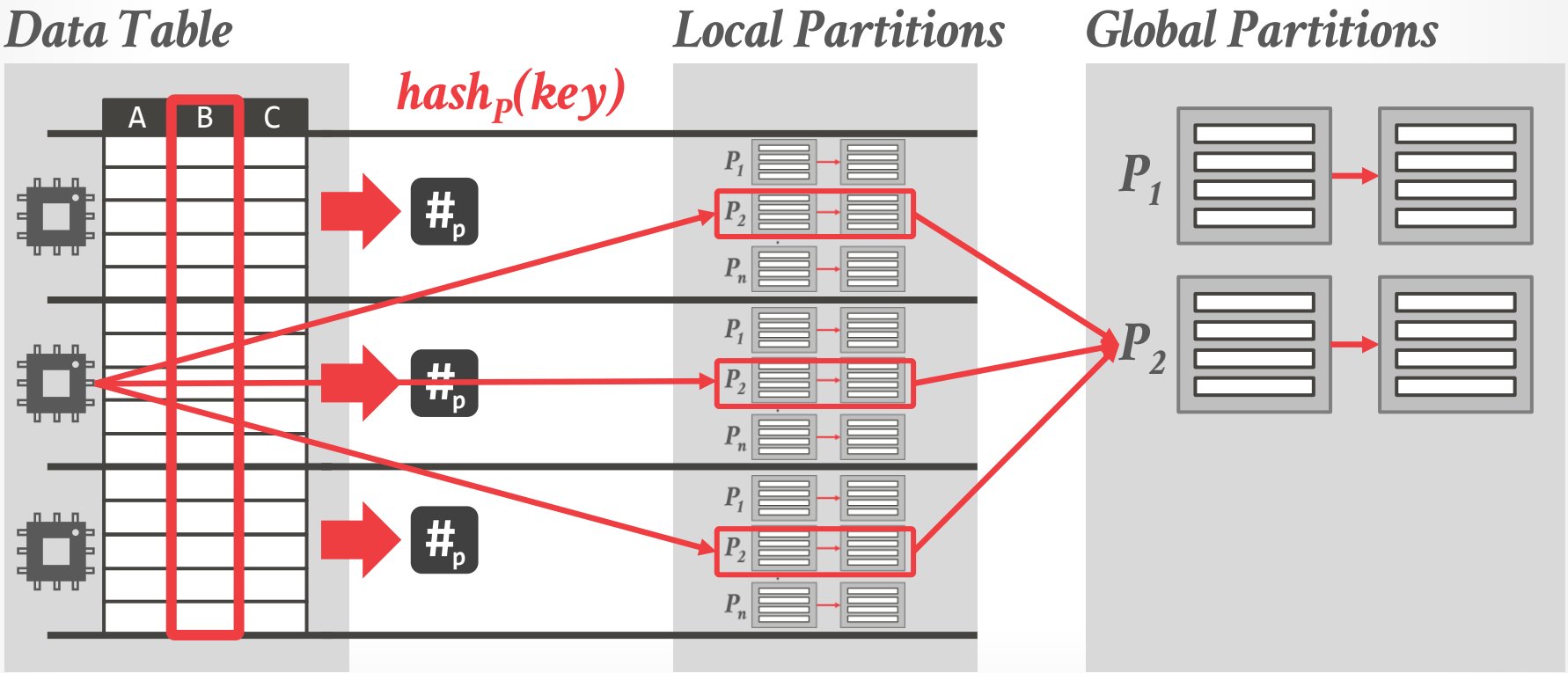
# PRIVATE PARTITIONS



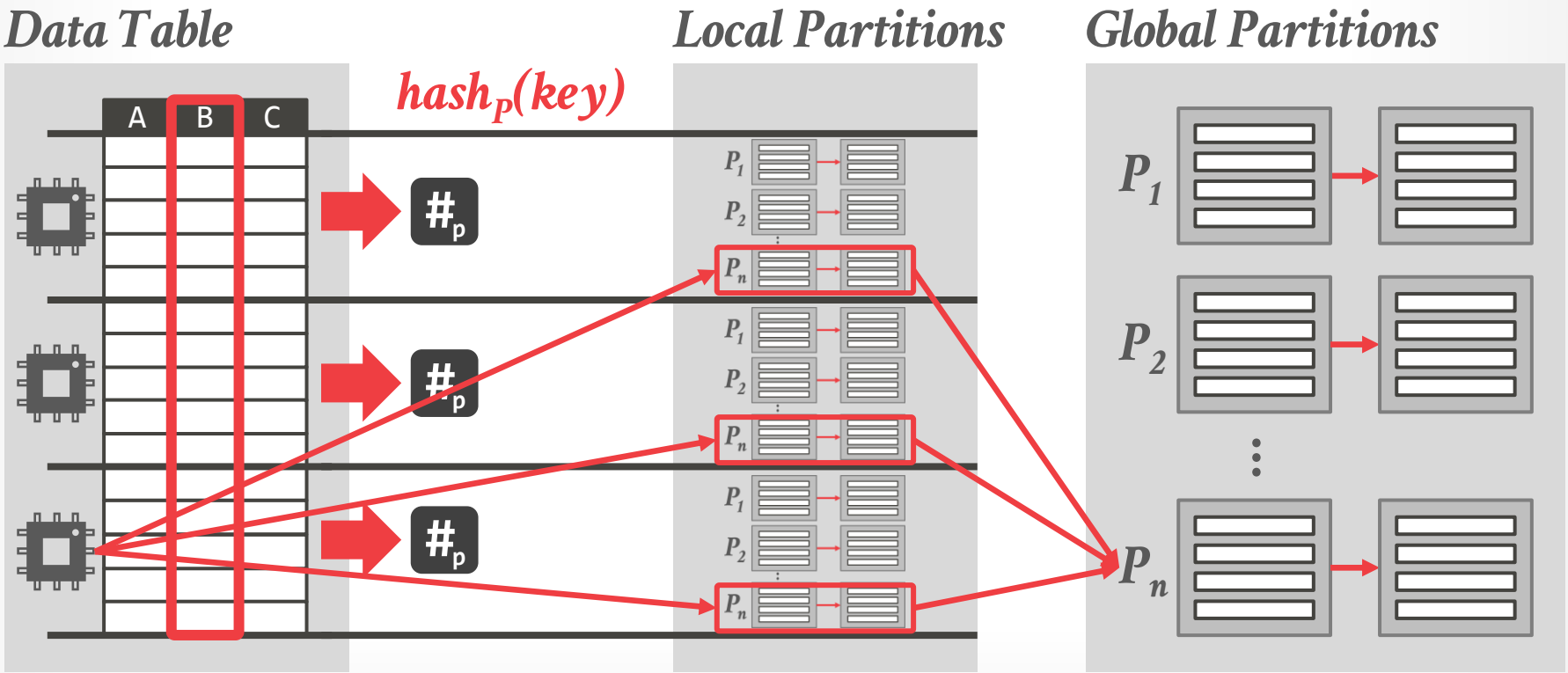
# PRIVATE PARTITIONS



# PRIVATE PARTITIONS



# PRIVATE PARTITIONS



# RADIX PARTITIONING

Scan the input relation multiple times to generate the partitions.

Two-pass algorithm:

→ **Step #1:** Scan R and compute a histogram of the # of tuples per hash key for the radix at some offset.

→ **Step #2:** Use this histogram to determine per-thread output offsets by computing the **prefix sum**.

→ **Step #3:** Scan R again and partition them according to the hash key.

# RADIX

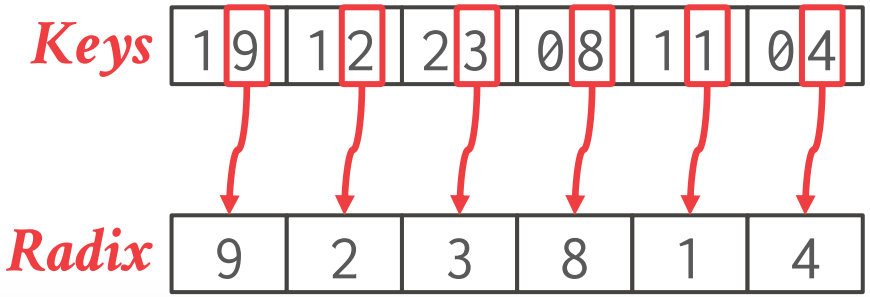
The radix of a key is the value of an integer at a position (using its base).

→ Efficient to compute with bitshifting + multiplication.

# RADIX

The radix of a key is the value of an integer at a position (using its base).

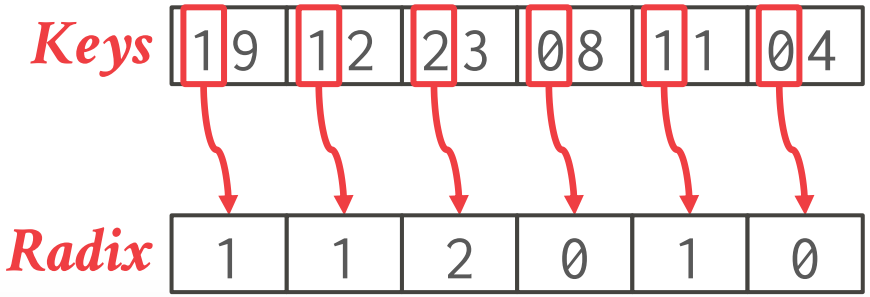
→ Efficient to compute with bitshifting + multiplication.



# RADIX

The radix of a key is the value of an integer at a position (using its base).

→ Efficient to compute with bitshifting + multiplication.

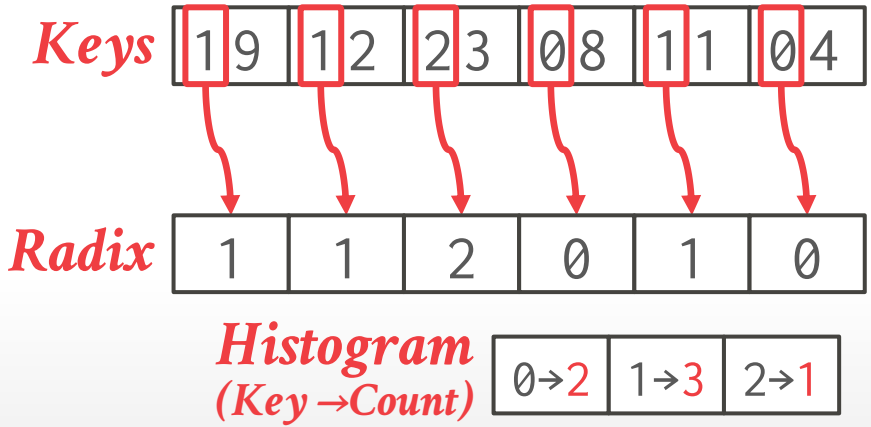


# RADIX

The radix of a key is the value of an integer at a position (using its base).

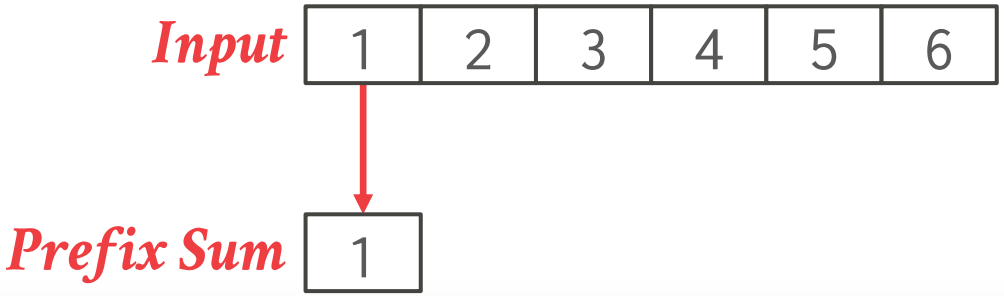
→ Efficient to compute with bitshifting + multiplication.

Compute radix for each key and populate histogram of counts per radix.



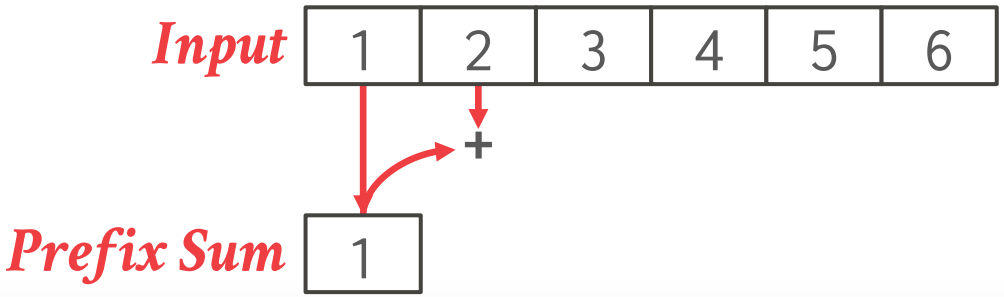
# PREFIX SUM

The prefix sum of a sequence of numbers
$(x_0, x_1, ..., x_n)$
is a second sequence of numbers
$(y_0, y1, ..., y_n)$
that is a running total of the input sequence.



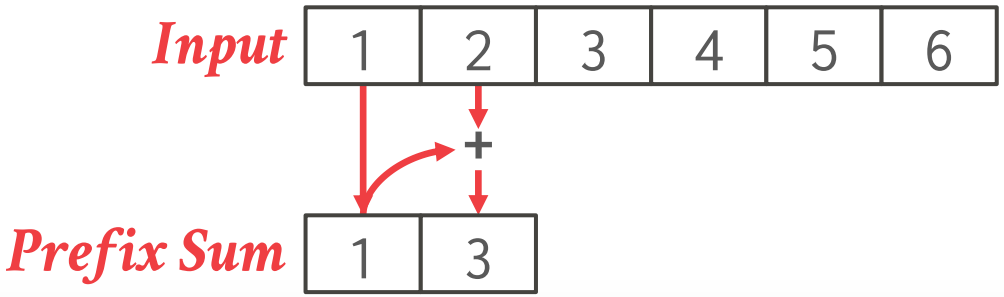
# PREFIX SUM

The prefix sum of a sequence of numbers
$(x_0, x_1, ..., x_n)$
is a second sequence of numbers
$(y_0, y1, ..., y_n)$
that is a running total of the input sequence.



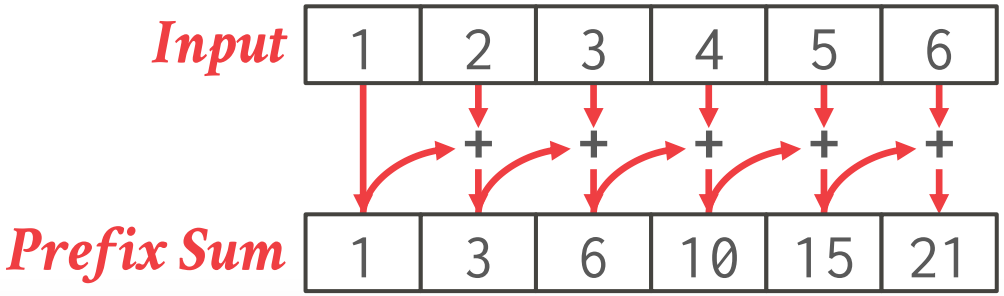
# PREFIX SUM

The prefix sum of a sequence of numbers
$(x_0, x_1, ..., x_n)$
is a second sequence of numbers
$(y_0, y1, ..., y_n)$
that is a running total of the input sequence.



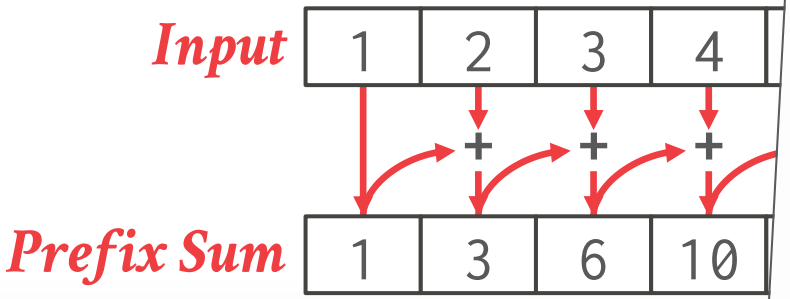
# PREFIX SUM

The prefix sum of a sequence of numbers
$(x_0, x_1, ..., x_n)$
is a second sequence of numbers
$(y_0, y1, ..., y_n)$
that is a running total of the input sequence.



# PREFIX SU

The prefix sum of a sequence of
$(x_0, x_1, ..., x_n)$
is a second sequence of numbers
$(y_0, y1, ..., y_n)$
that is a running total of the inp




Scan Primitives for Vector Computers*

Siddhartha Chatterjee
Guy E. Blelloch
Marco Zagha
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

**Abstract**

This paper describes an optimized implementation of a set of *scan* (also called all-prefix-sums) primitives on a single processor of a CRAY Y-MP, and demonstrates that their use leads to greatly improved performance for several applications that cannot be vectorized with existing compiler technology. The algorithm used to implement the scans is based on an algorithm for parallel computers and is applicable with minor modifications to any register-based vector computer. On the CRAY Y-MP, the asymptotic running time of the *plus-scan* is about 2.25 times that of a vector add, and is within 20% of optimal. An important aspect of our implementation is that a set of *segmented* versions of these scans are only marginally more expensive than the unsegmented versions. These segmented versions can be used to execute a scan on multiple data sets without having to pay the vector startup cost ($n_{1/2}$) for each set.

The paper describes a radix sorting routine based on the scans that is 13 times faster than a Fortran version and within 20% of a highly optimized library sort routine, three operations on trees that are between 10 and 20 times faster than the corresponding C versions, and a connectionist learning algorithm that is 10 times faster than the corresponding C version for sparse and irregular networks.

| Routine | $l_i$ (clks/elt) | |
|---|---|---|
| | Scan version | Scalar version |
| plus scan (int) | 2.5 | 7.5 (*) |
| segmented plus scan (int) | 3.7 | 74.8 (*) |
| parallel radix sort (64 bits) | 896.8 | 11730.0 (*) |
| branch_sums | 12.2 | 206.5 (†) |
| root_sums | 9.8 | 208.6 (†) |
| delete_vertices | 19.2 | 276.2 (†) |

Table 1: Incremental processing times per element for primitives and applications discussed in this paper, for both scan and scalar versions. All numbers are for a single processor of a CRAY Y-MP. 1 clock tick = 6 ns. Items marked with (*) were written in Fortran and those marked with (†) were written in C.

## 1 Introduction

Vector supercomputers have been used to supply the high computing power needed for many applications. However, the performance obtained from these machines critically depends on the ability to produce code that vectorizes well. Two distinct approaches have been taken to meet this goal—vectorization of "dusty decks" [18], and language support for vector intrinsics, as seen in the proposed Fortran 8x standard [1]. In both cases, the focus of the work has been in speeding up "scientific" computations, characterized by regular and static data structures and predominantly regular access patterns within these data structures. These alternatives are not very effective for problems that create and manipulate more irregular and dynamically varying data structures such as trees and graphs.

Elsewhere, scan (prefix sum) operations have been shown to be extremely powerful primitives in designing parallel algorithms for manipulating such irregular and dynamically changing data structures [3]. This paper shows how the scan operations can also have great benefit for such algorithms on pipelined vector machines. It describes an optimized implementation of the scan primitives on the CRAY Y-MP[1], and gives performance numbers for several applications based on the primitives (see Table 1). The approach in the design of these algorithms is similar to that of the Basic Linear Algebra Subprograms (BLAS) developed in the context of linear algebra computations [14] in that the algorithms are based on a set of primitives whose implementations are optimized rather than having a compiler try to vectorize existing code.

The remainder of this paper is organized as follows. Section 2 introduces the scan primitives and reviews previous work on vectorizing scans. Section 3 discusses our implementation in detail and presents performance numbers. Section 4 presents other primitives used in this paper, and three applications using these primitives. Finally, future work and conclusions are given in Section 5.[2]

*This research was supported by the Defense Advanced Research Projects Agency (DOD) and monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-87-C-1499, ARPA Order No. 4976, Amendment 20.

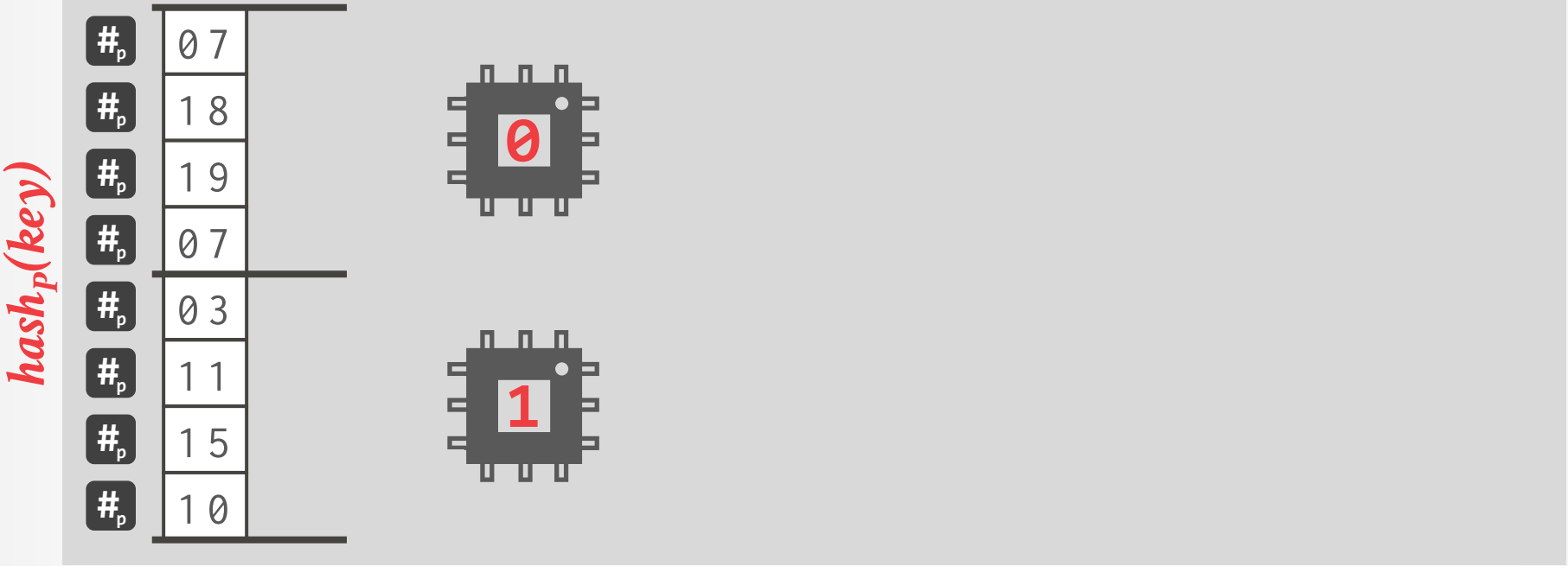
The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. government.

[1]CRAY Y-MP and CFT77 are trademarks of Cray Research, Inc.
[2]All Fortran code discussed in this paper was compiled with CFT77 Version 3.1. All C code was compiled with Cray PCC Version XMP/YMP 4.1.8.

CH2916-5/90/0000/0666/$01.00 © IEEE

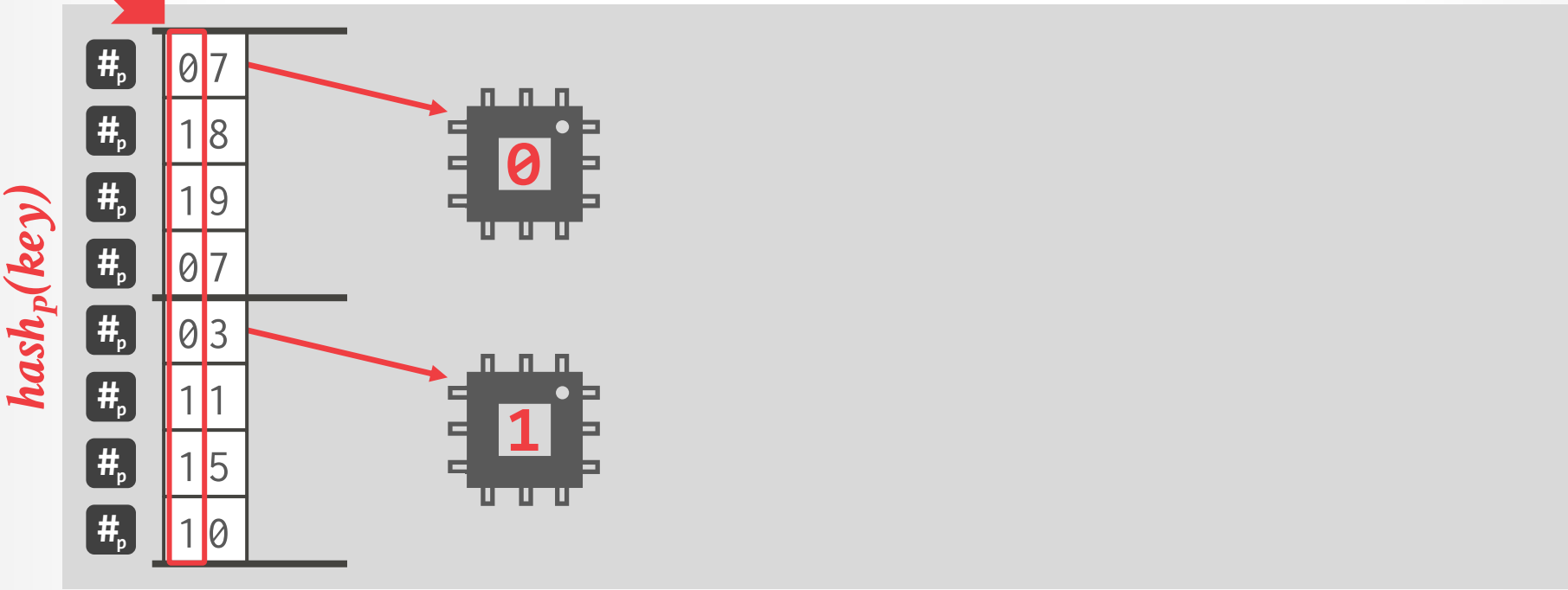
# RADIX PARTITIONS

***Step #1: Inspect input, create histograms***



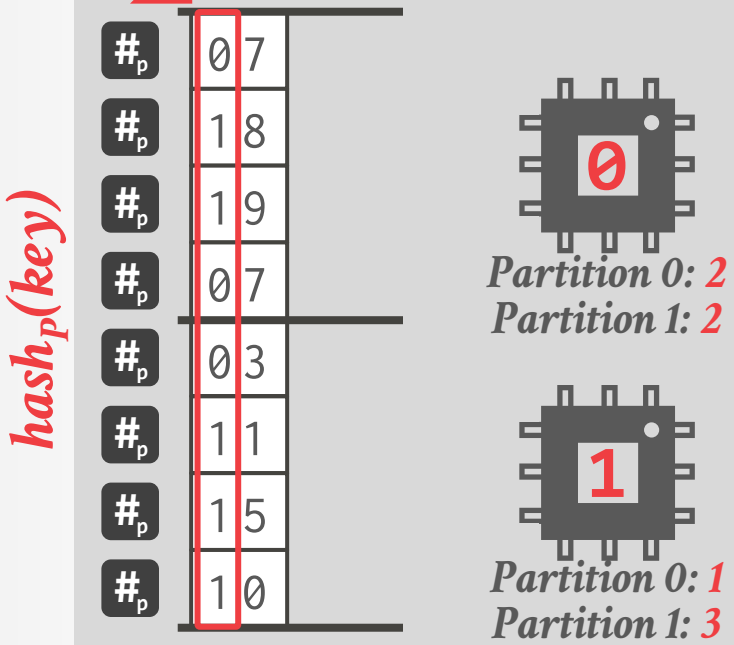
# RADIX PARTITIONS

*Step #1: Inspect input, create histograms*



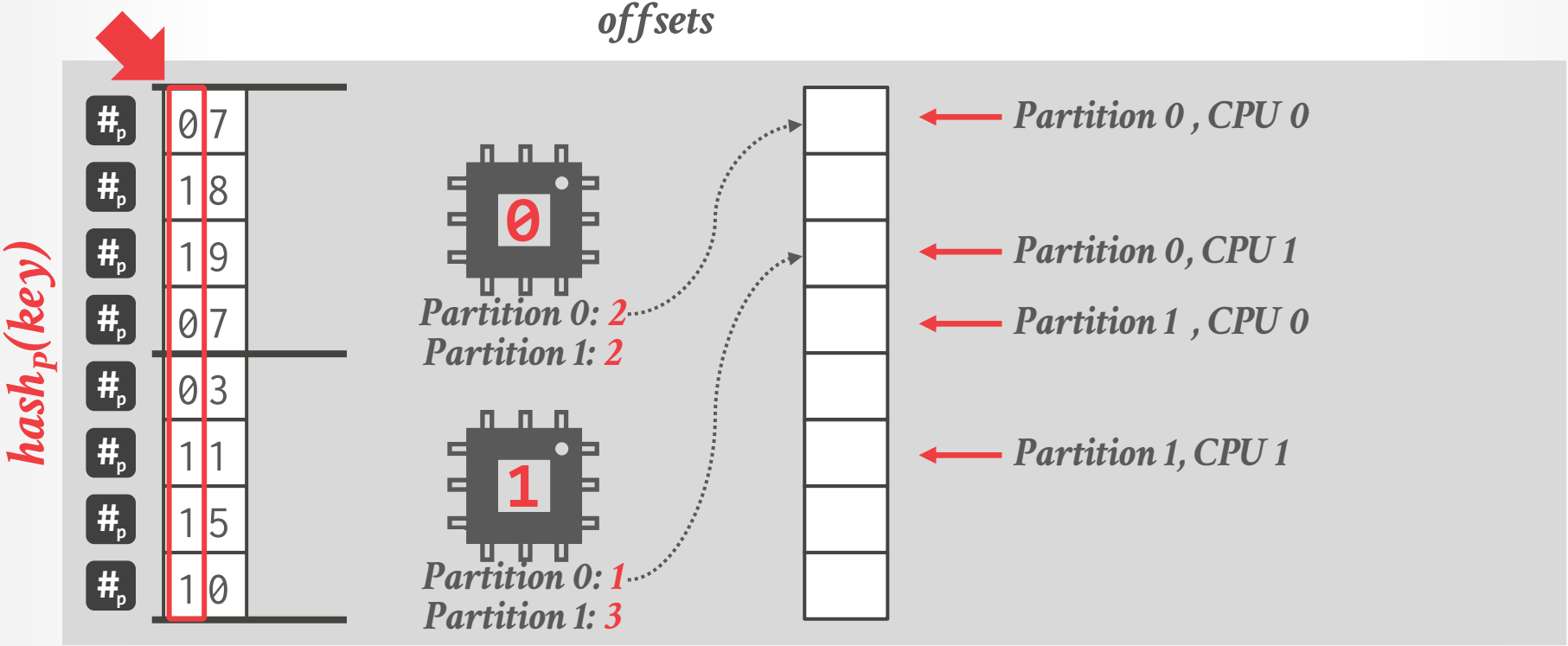
# RADIX PARTITIONS

***Step #1: Inspect input, create histograms***



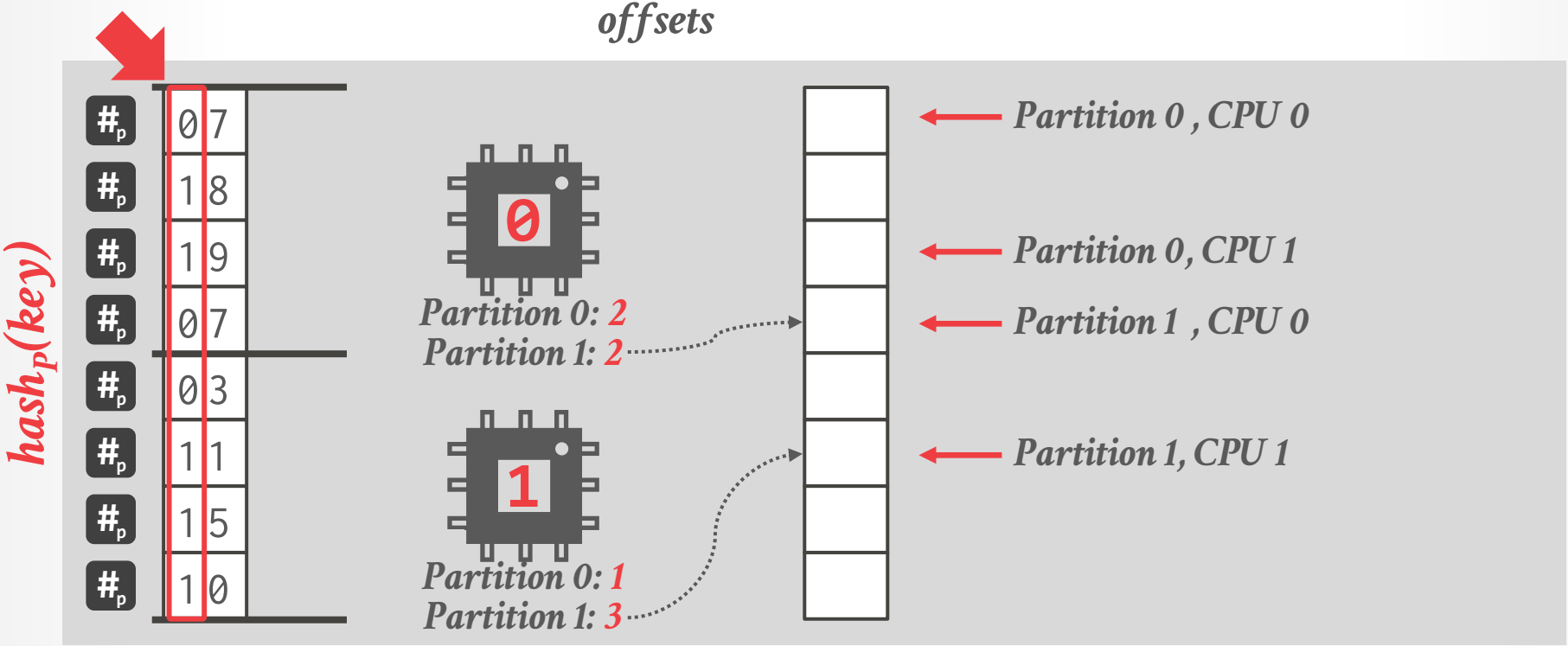
# RADIX PARTITIONS

*Step #2: Compute output offsets*



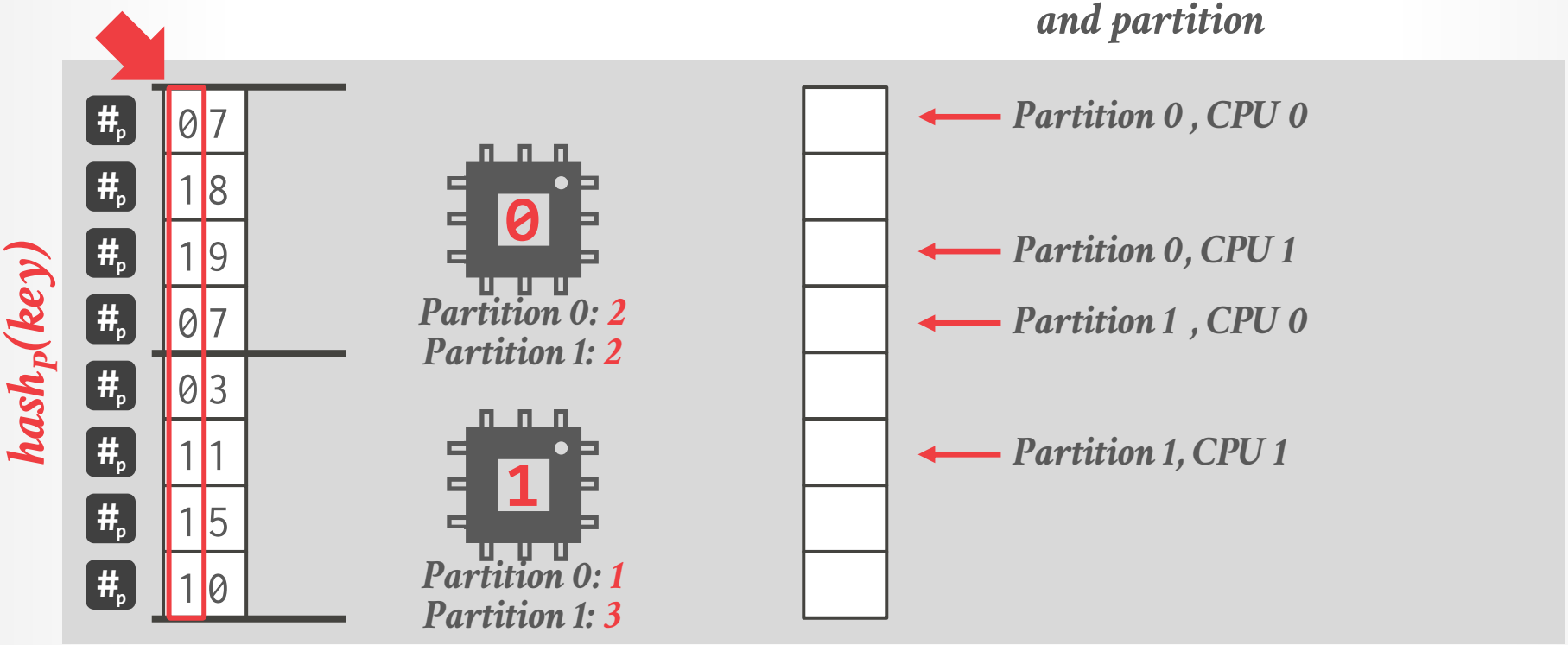
# RADIX PARTITIONS

*Step #2: Compute output offsets*



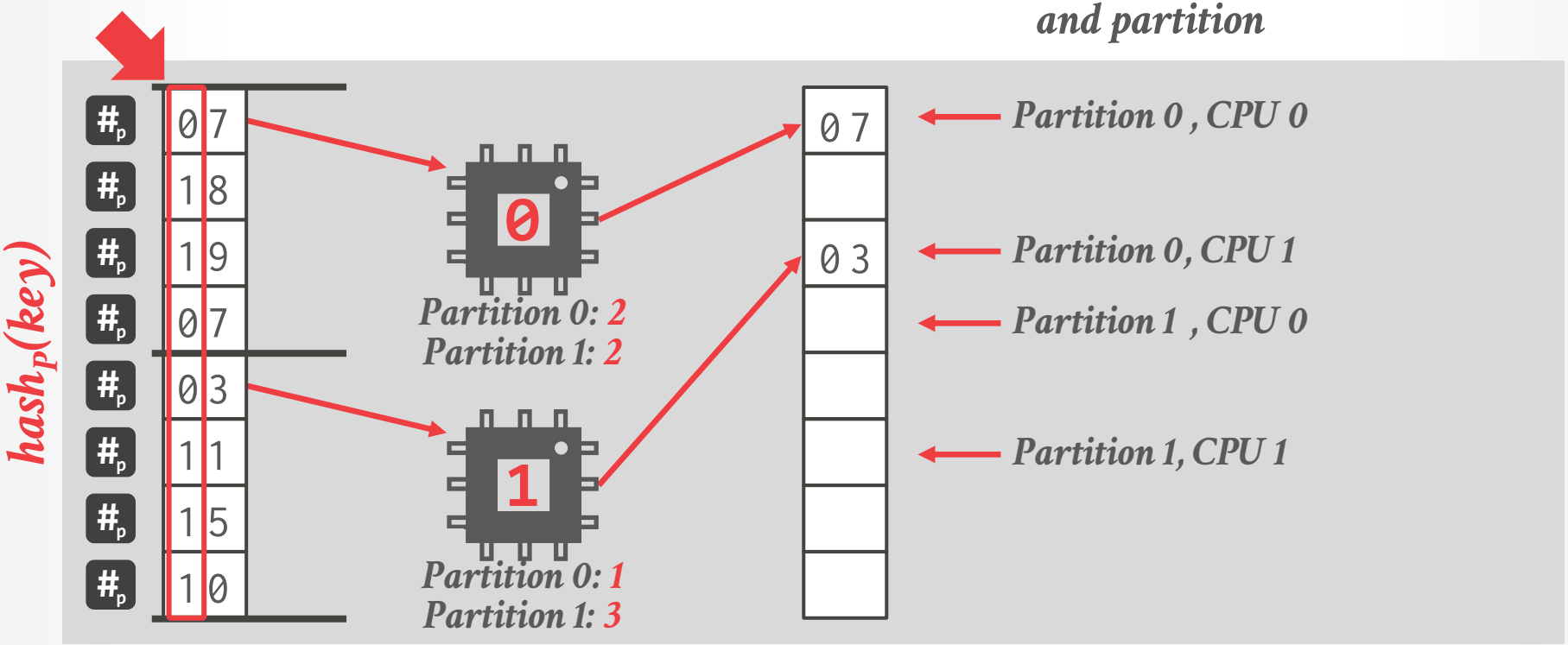
# RADIX PARTITIONS

*Step #3: Read input and partition*



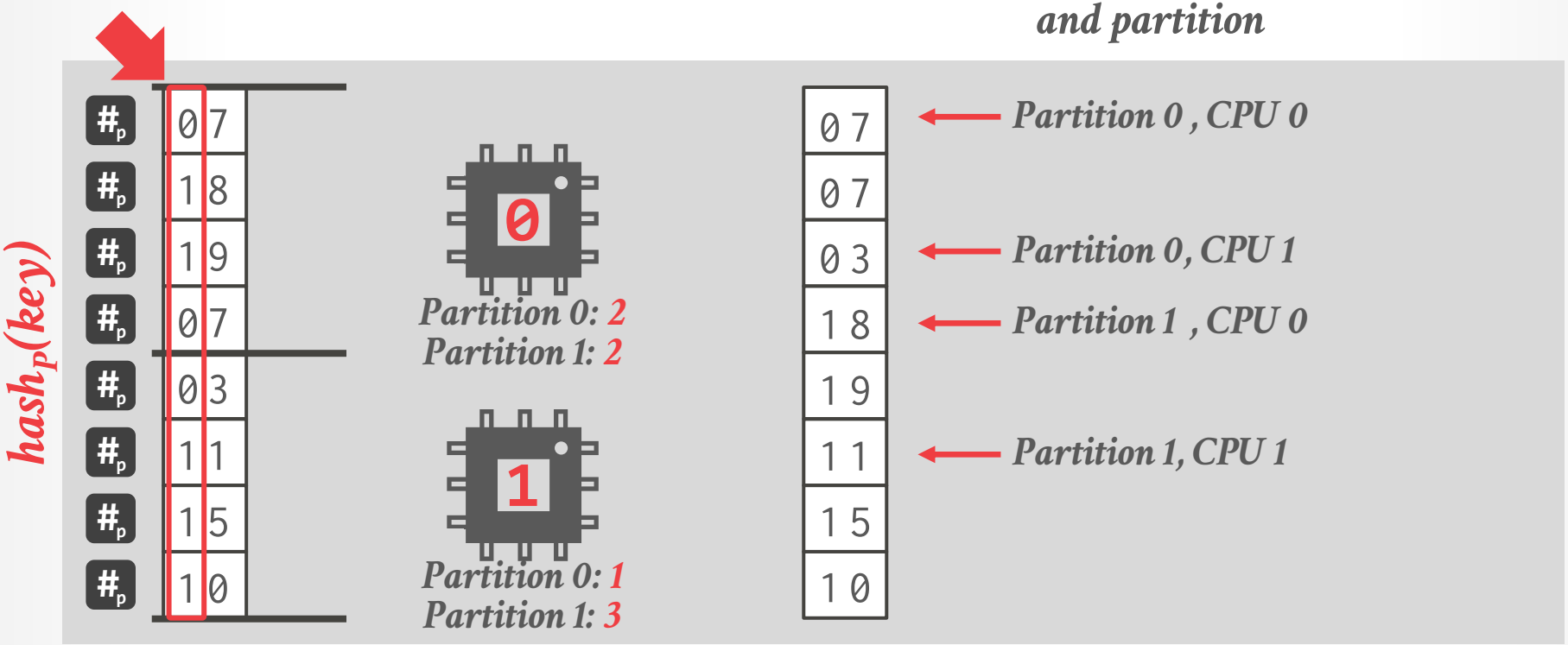
# RADIX PARTITIONS

*Step #3: Read input and partition*



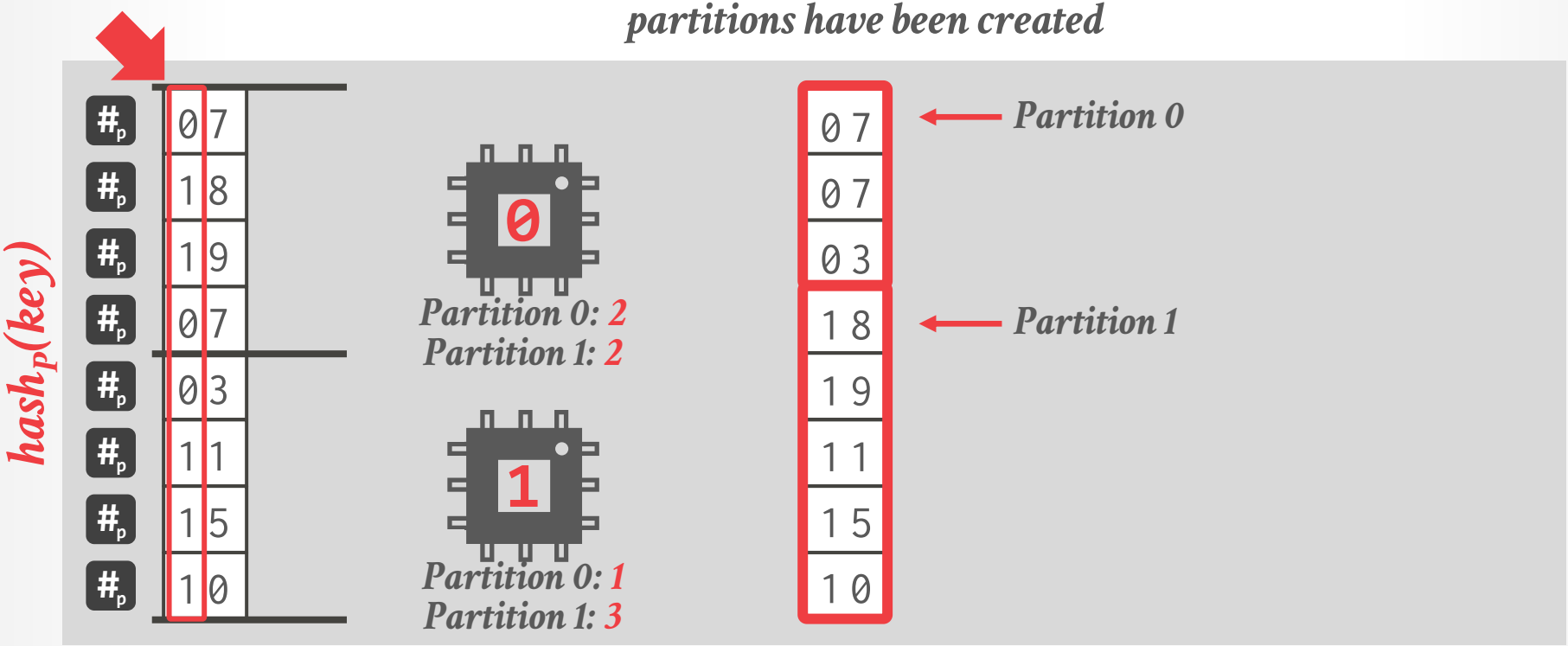
# RADIX PARTITIONS

*Step #3: Read input and partition*



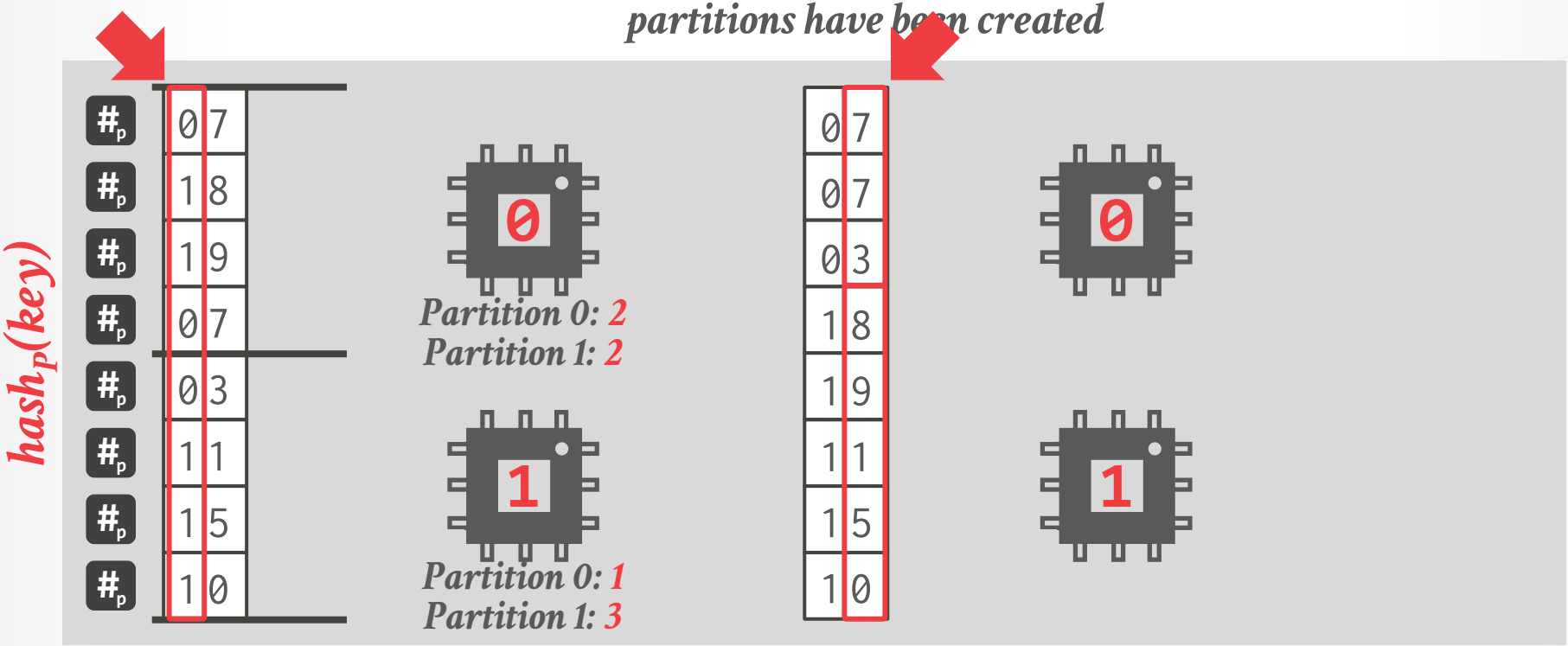
# RADIX PARTITIONS

*Recursively repeat until target number of partitions have been created*



# RADIX PARTITIONS

*Recursively repeat until target number of partitions have been created*



# OPTIMIZATIONS

## Software Write Combine Buffers:

→ Each worker maintains local output buffer to stage writes.
→ When buffer full, write changes to global partition.
→ Similar to private partitions but without a separate write phase at the end.

## Non-temporal Streaming Writes

→ Workers write data to global partition memory using streaming instructions to bypass CPU caches.

ON THE SURPRISING DIFFICULTY OF SIMPLE THINGS: THE CASE OF RADIX PARTITIONING
VLDB 2015

# BUILD PHASE

The threads are then to scan either the tuples (or partitions) of **R**.

For each tuple, hash the join key attribute for that tuple and add it to the appropriate bucket in the hash table.

→ The buckets should only be a few cache lines in size.

# HASH TABLES

## 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.

# HASH FUNCTIONS

We do not want to use a cryptographic hash function for our join algorithm.

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

→ **Best Speed:** Always return '1'
→ **Best Collision Rate:** Perfect hashing

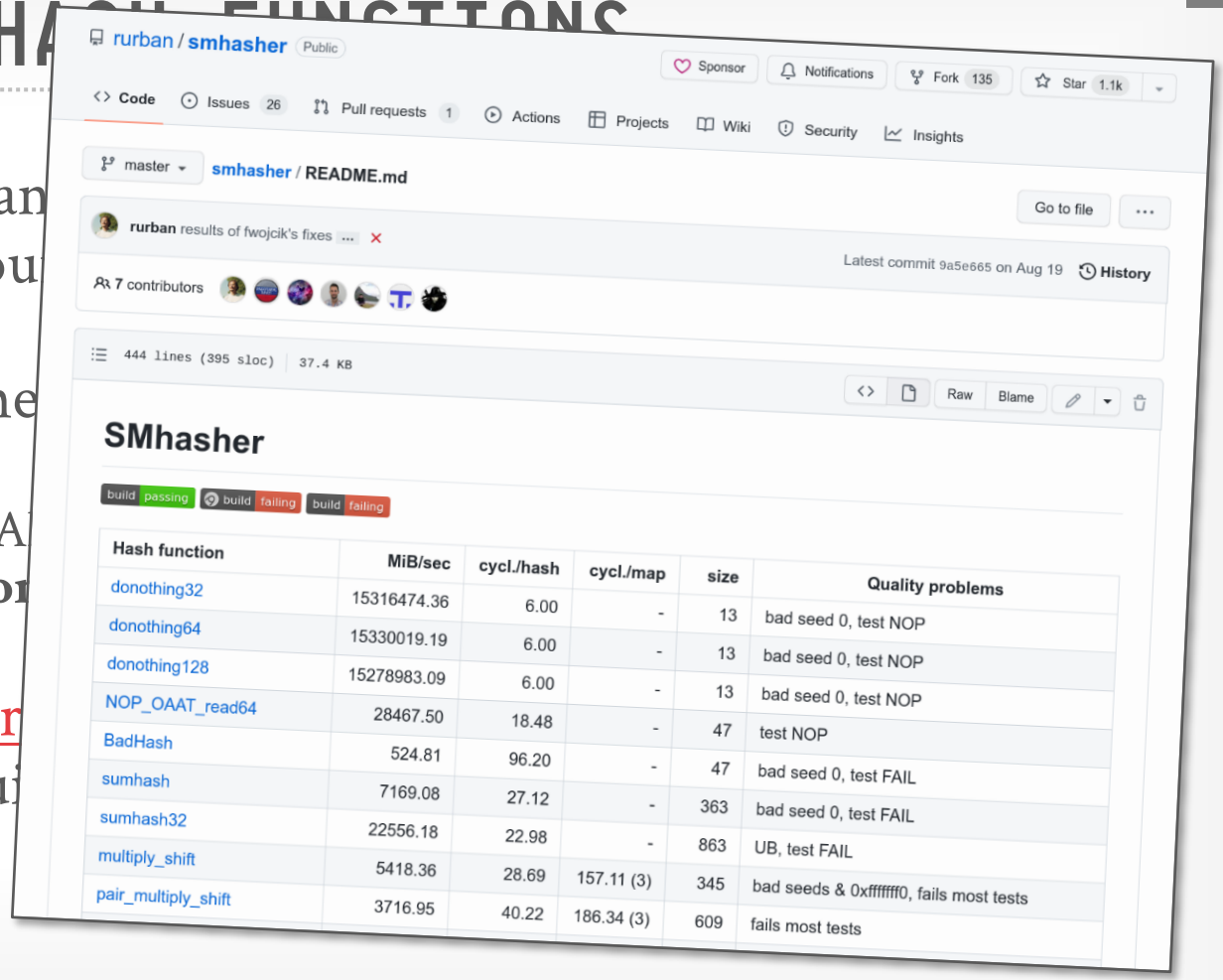
See SMHasher for a comprehensive hash function benchmark suite.

# HA FUNCTIONS

We do not wan
function for ou

We want some
collision rate.
→ **Best Speed:** A
→ **Best Collisio**

See SMHasher
benchmark su




| Hash function | MiB/sec | cycl./hash | cycl./map | size | Quality problems |
|---|---|---|---|---|---|
| donothing32 | 15316474.36 | 6.00 | - | 13 | bad seed 0, test NOP |
| donothing64 | 15330019.19 | 6.00 | - | 13 | bad seed 0, test NOP |
| donothing128 | 15278983.09 | 6.00 | - | 13 | bad seed 0, test NOP |
| NOP_OAAT_read64 | 28467.50 | 18.48 | - | 47 | test NOP |
| BadHash | 524.81 | 96.20 | - | 47 | bad seed 0, test FAIL |
| sumhash | 7169.08 | 27.12 | - | 363 | bad seed 0, test FAIL |
| sumhash32 | 22556.18 | 22.98 | - | 863 | UB, test FAIL |
| multiply_shift | 5418.36 | 28.69 | 157.11 (3) | 345 | bad seeds & 0xffffffff0, fails most tests |
| pair_multiply_shift | 3716.95 | 40.22 | 186.34 (3) | 609 | fails most tests |

# 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)**

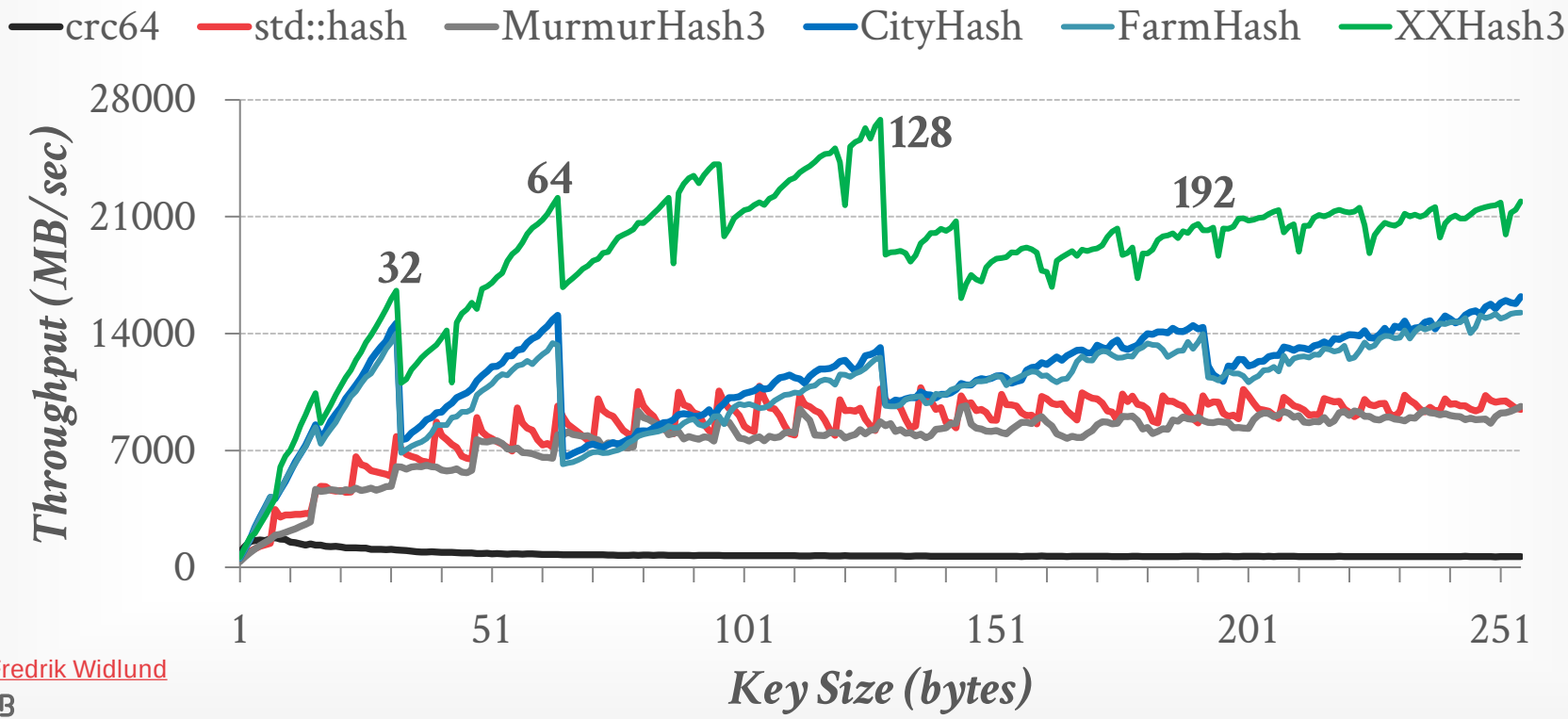
→ From the creator of zstd compression.

**Google FarmHash (2014)**

→ Newer version of CityHash with better collision rates.

# HASH FUNCTION BENCHMARK

*Intel Core i7-8700K @ 3.70GHz*




Source: Fredrik Widlund

# HASHING SCHEMES

**Approach #1: Chained Hashing**

**Approach #2: Linear Probe Hashing**

**Approach #3: Robin Hood Hashing**

**Approach #4: Hopscotch Hashing**

**Approach #5: Cuckoo Hashing**

# CHAINED HASHING

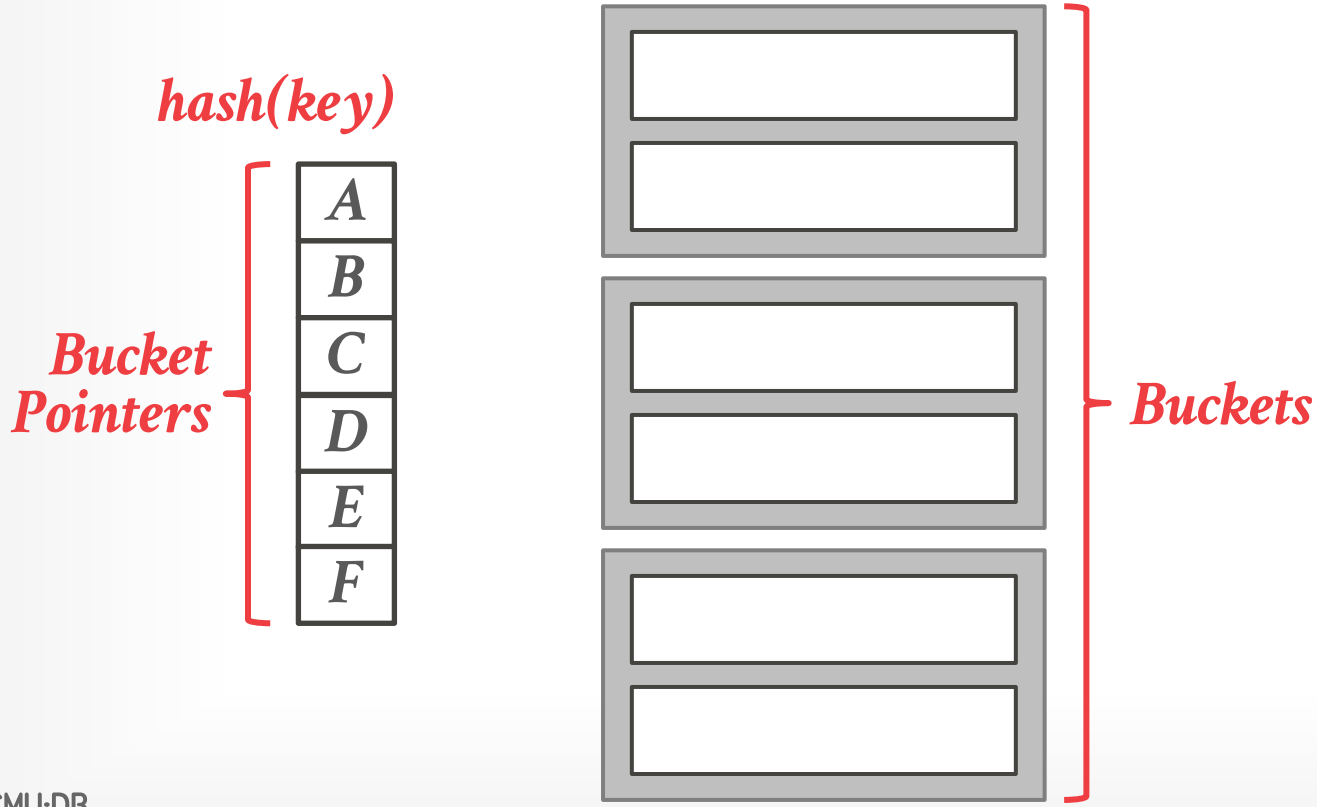
Maintain a linked list of buckets for each slot in the hash table.

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

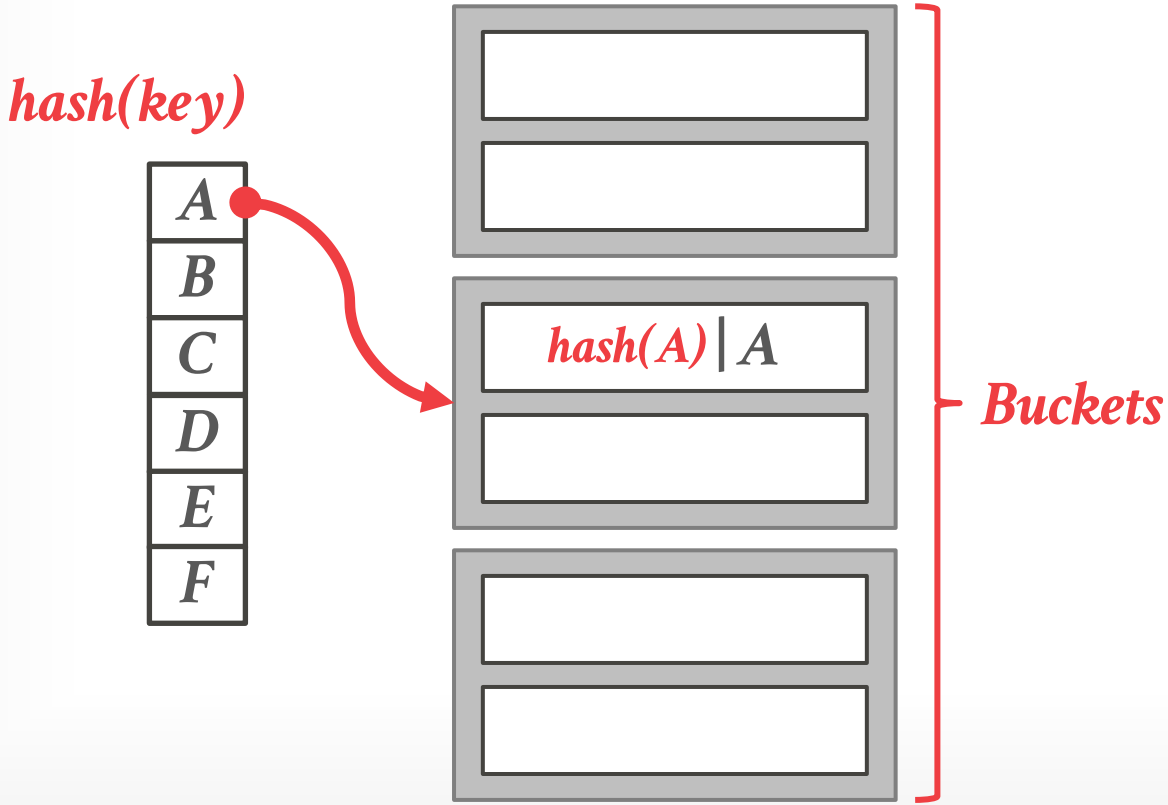
→ To determine whether an element is present, hash to its bucket and scan for it.

→ Insertions and deletions are generalizations of lookups.

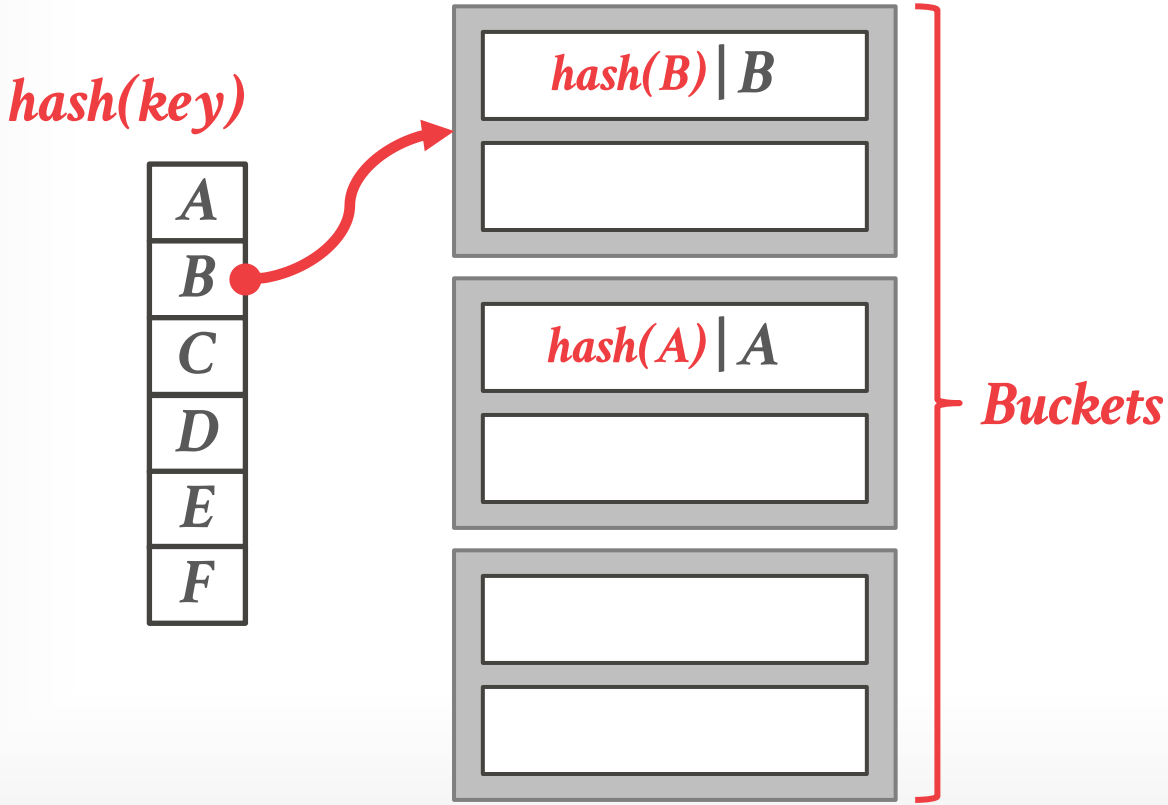
# CHAINED HASHING



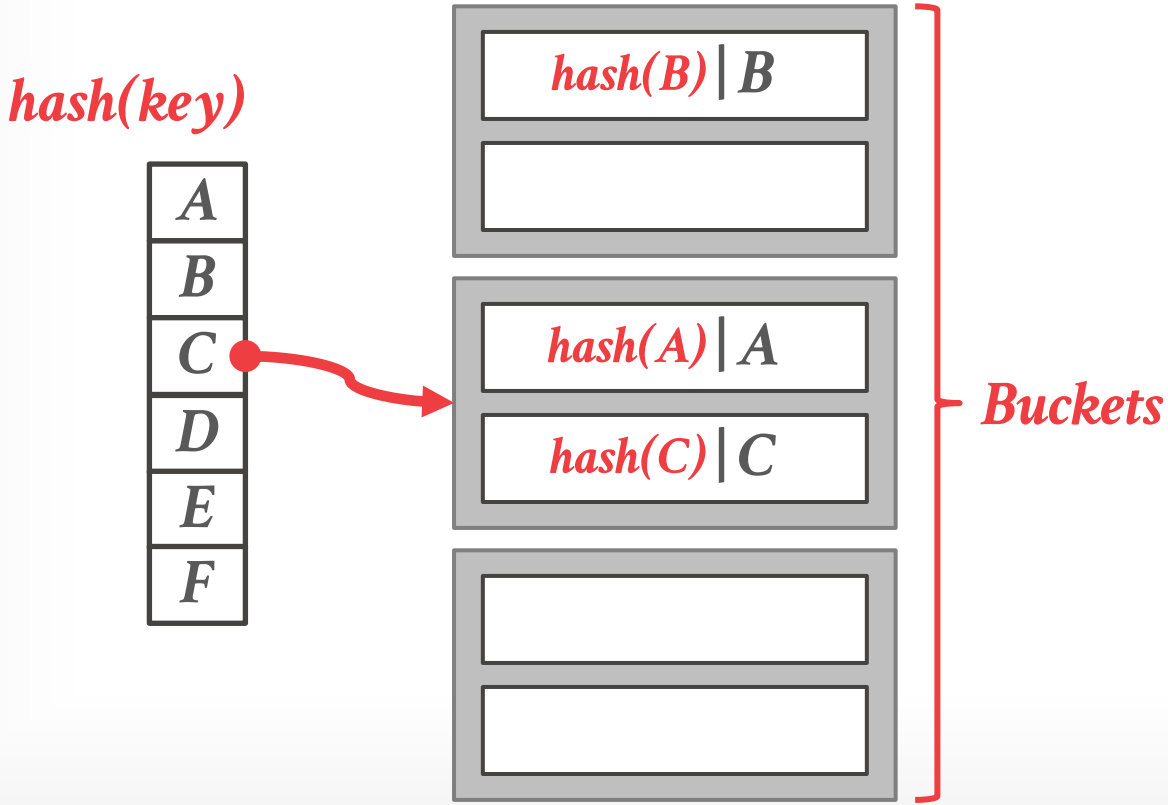
# CHAINED HASHING



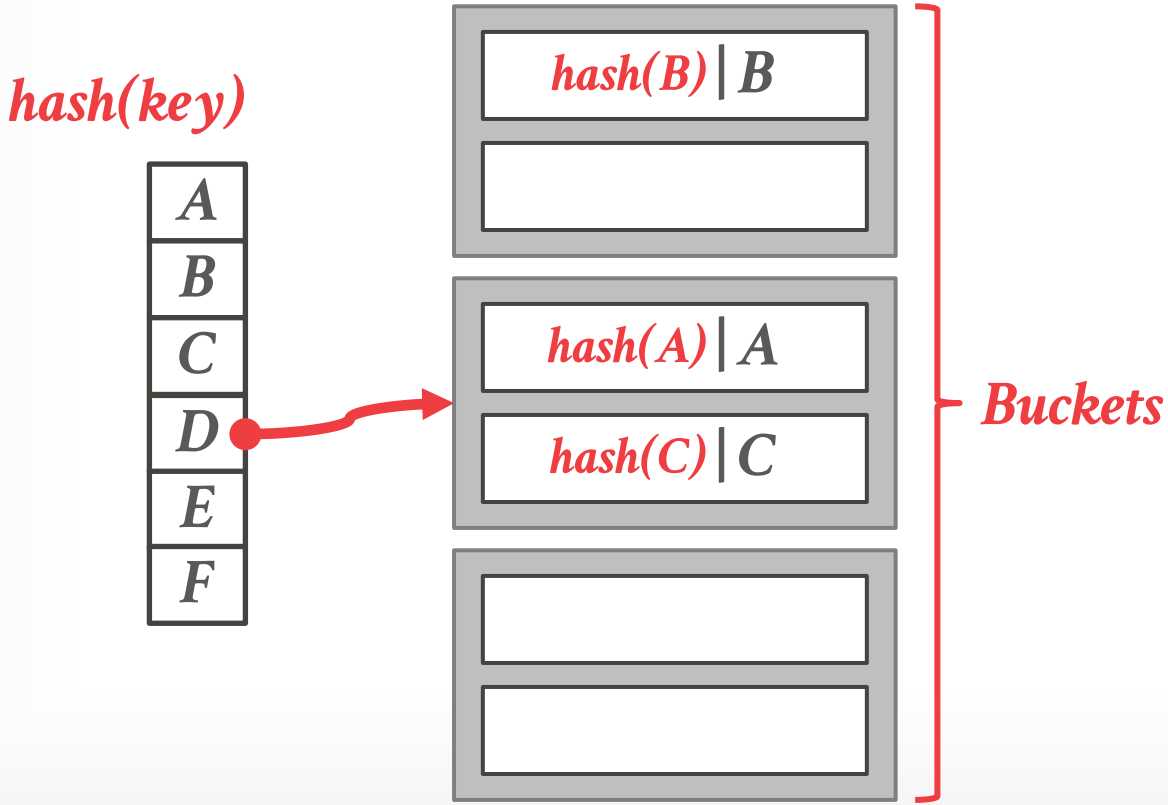
# CHAINED HASHING



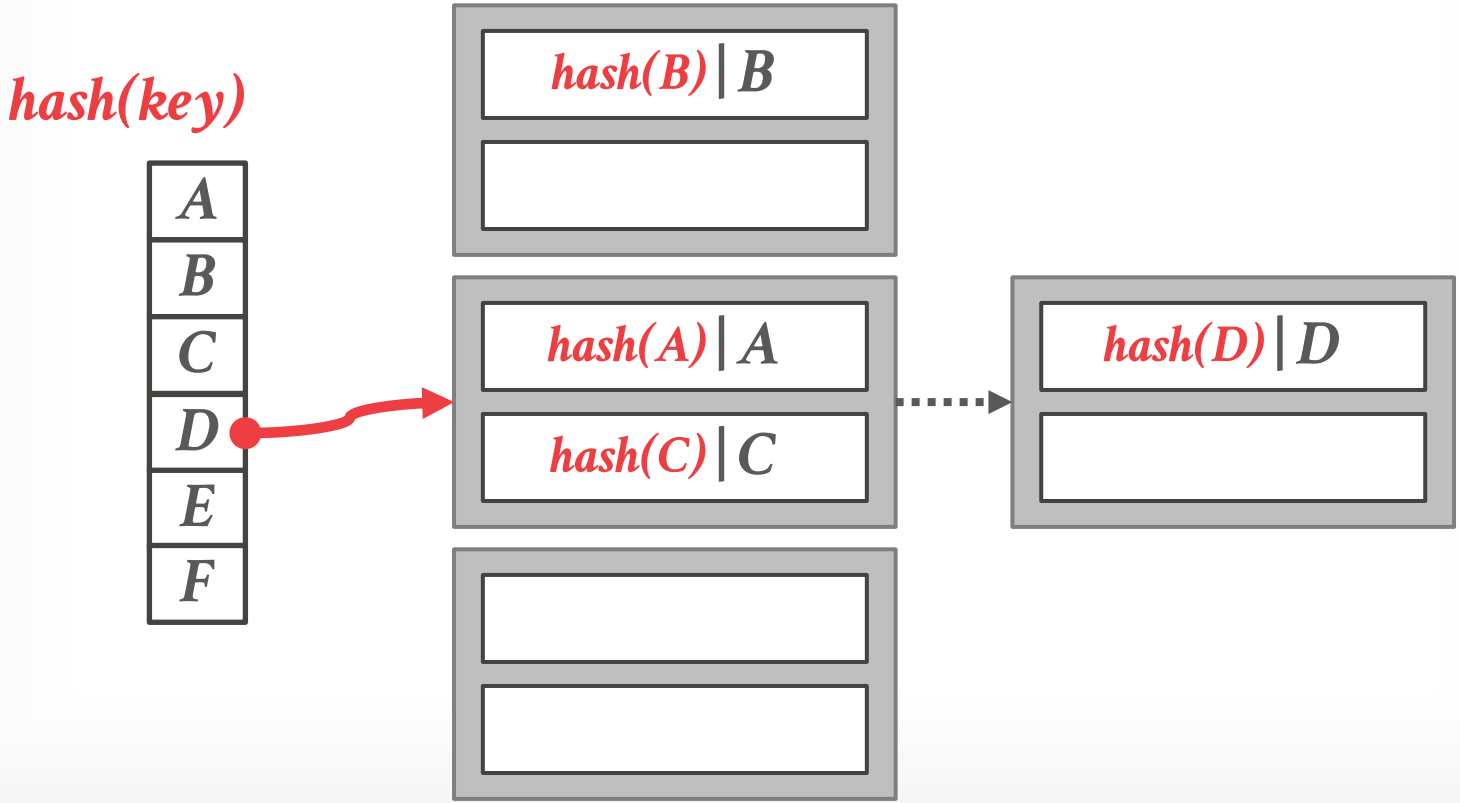
# CHAINED HASHING



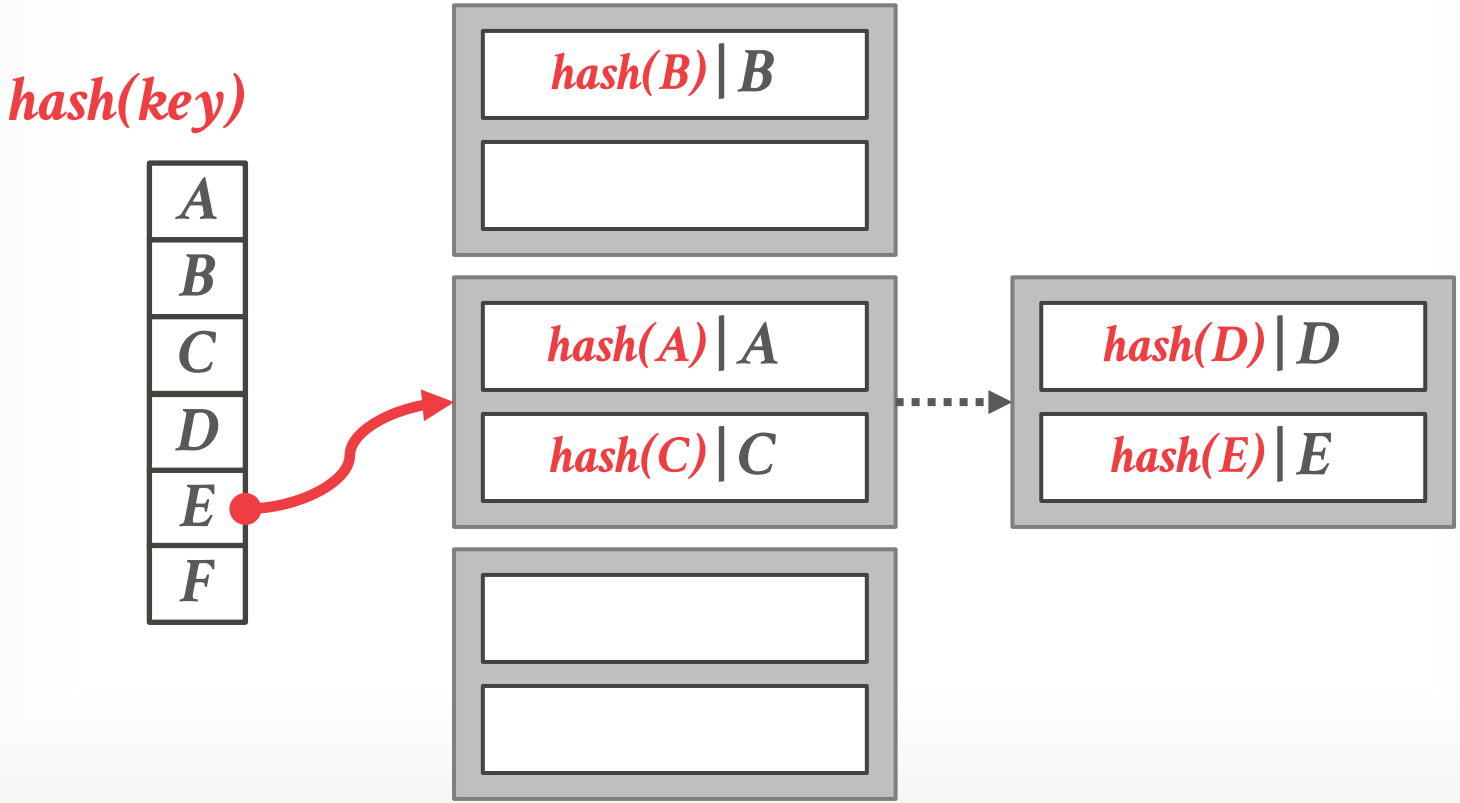
# CHAINED HASHING



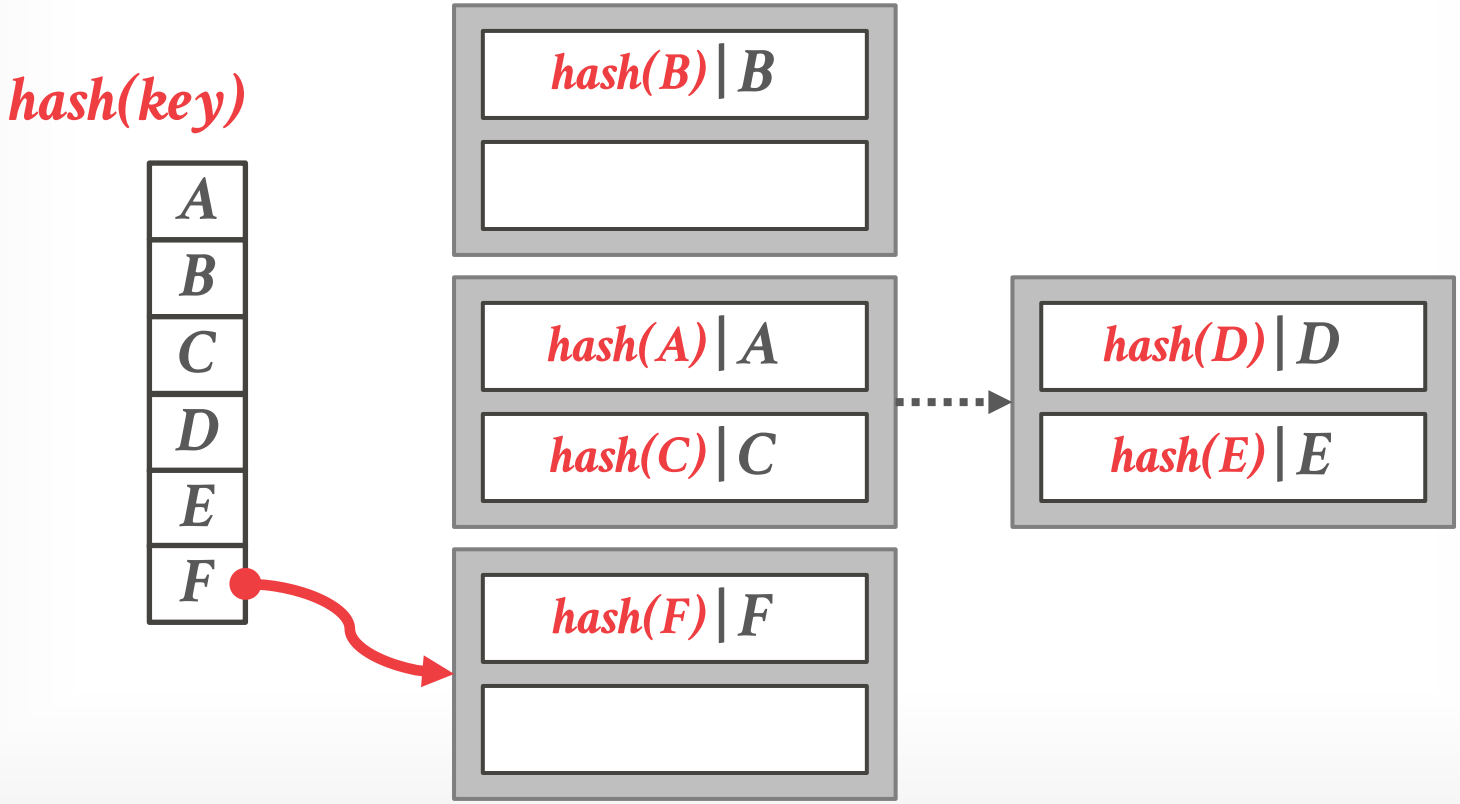
# CHAINED HASHING



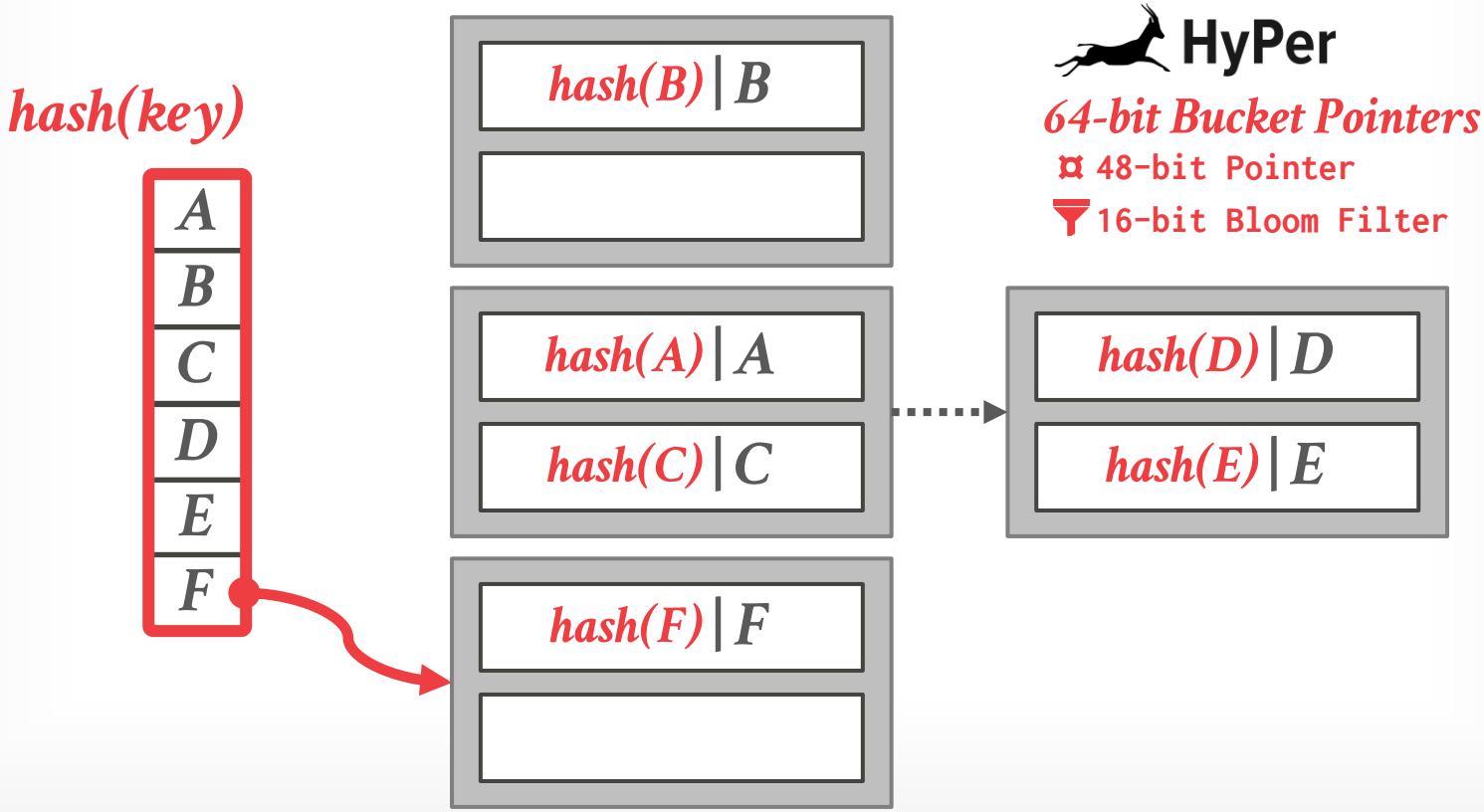
# CHAINED HASHING



# CHAINED HASHING



# CHAINED HASHING



# LINEAR PROBE HASHING

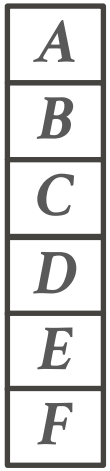
Single giant table of slots.

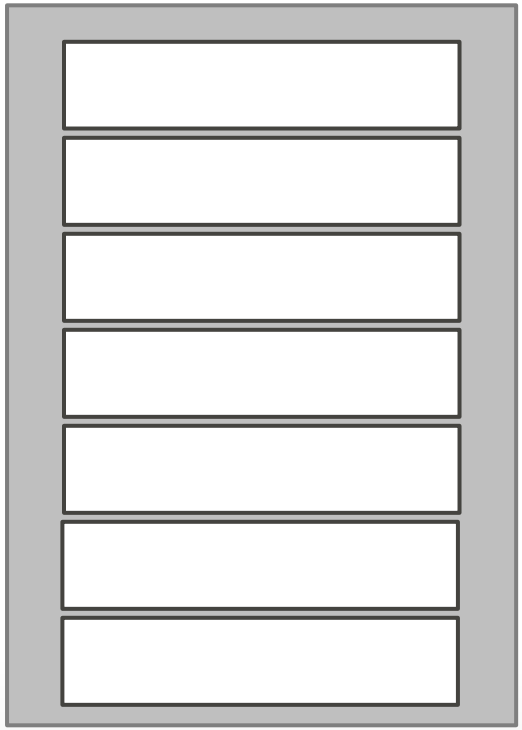
Resolve collisions by linearly searching for the next free slot in the table.

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

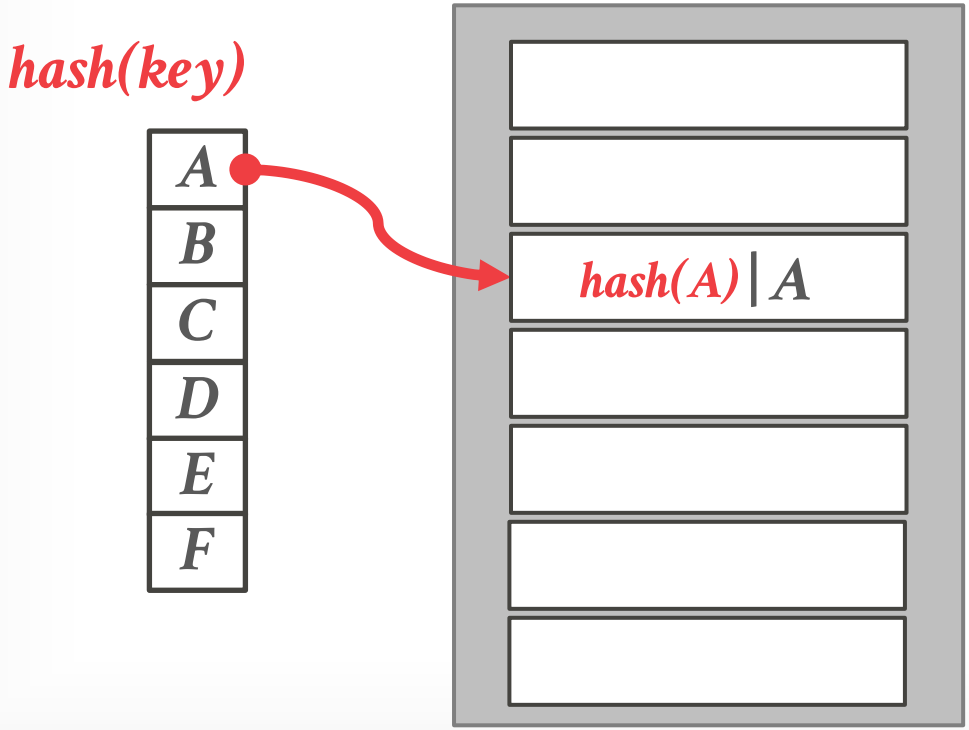
# LINEAR PROBE HASHING

*hash(key)*

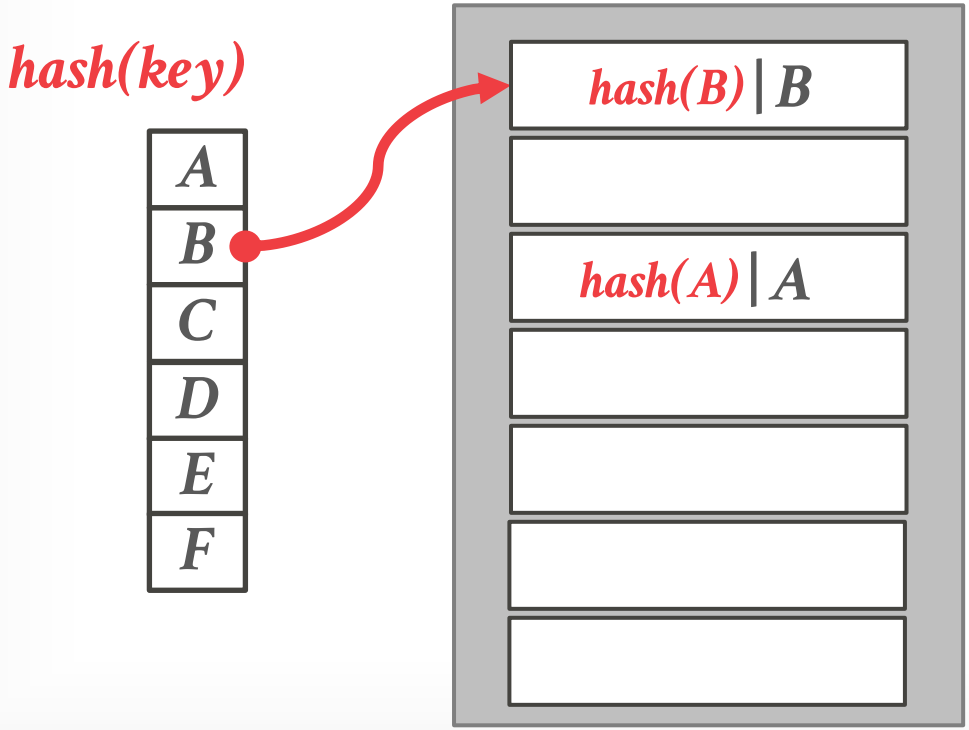


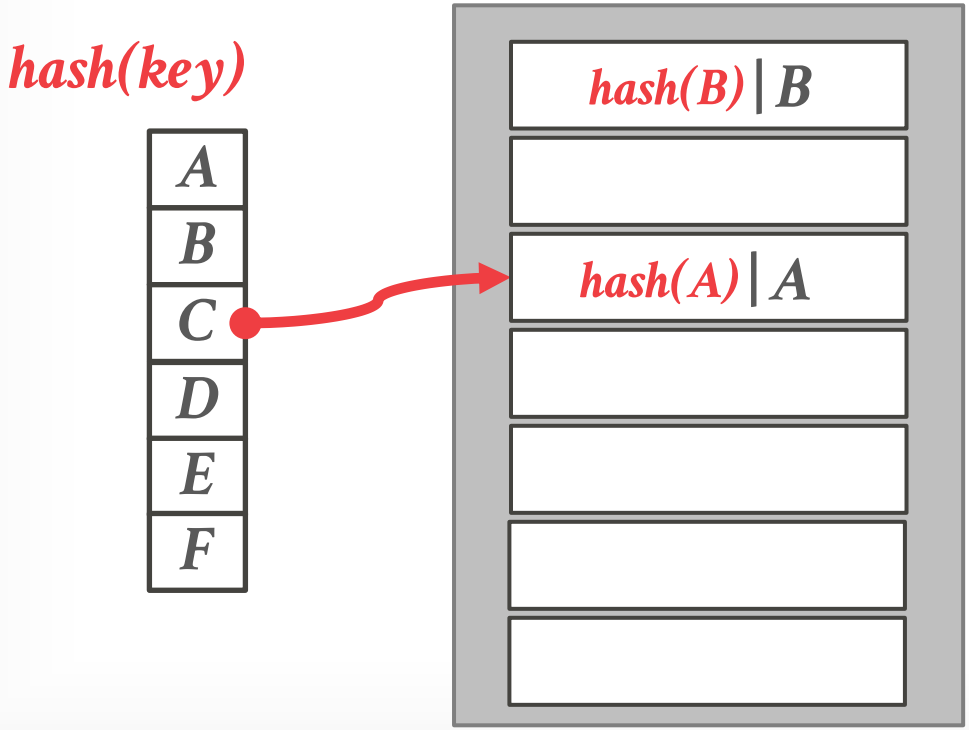
# LINEAR PROBE HASHING



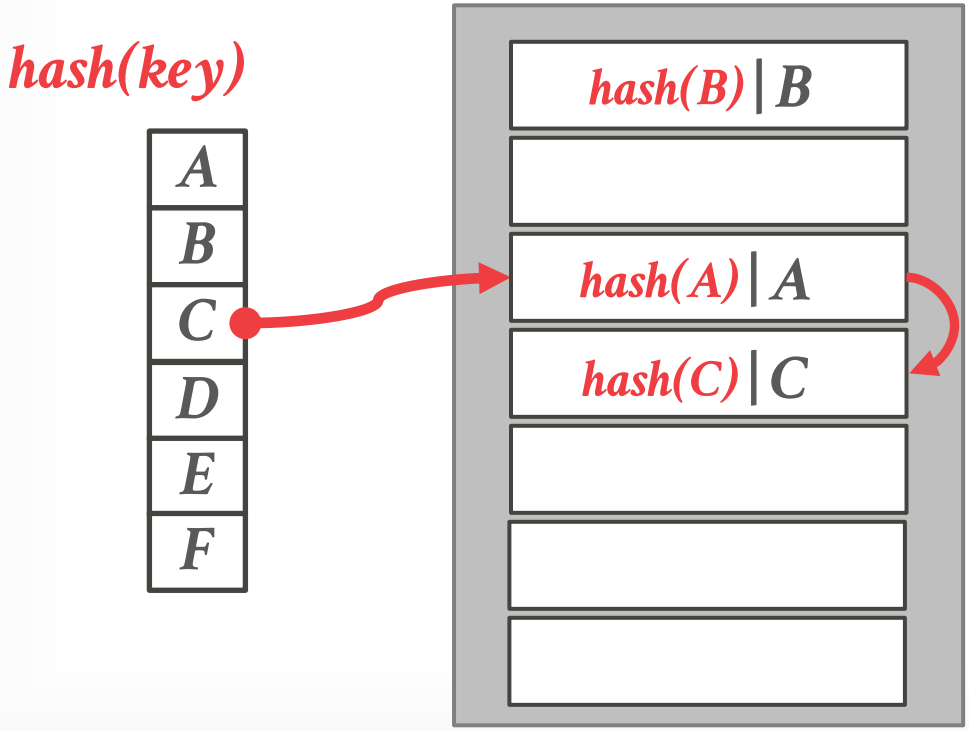
# LINEAR PROBE HASHING



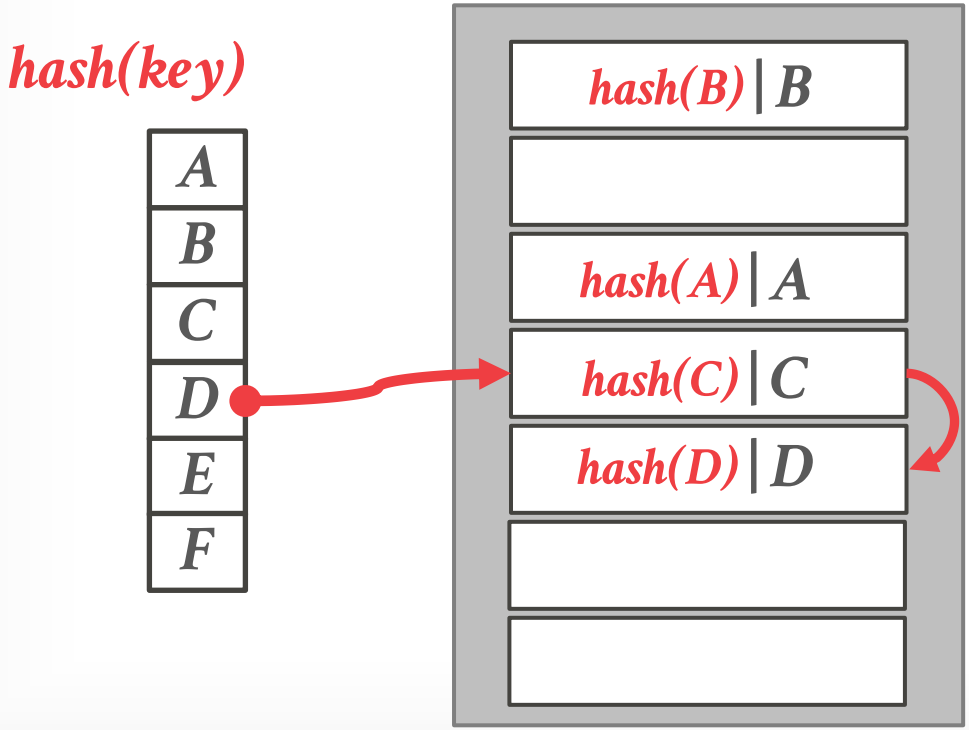
# LINEAR PROBE HASHING



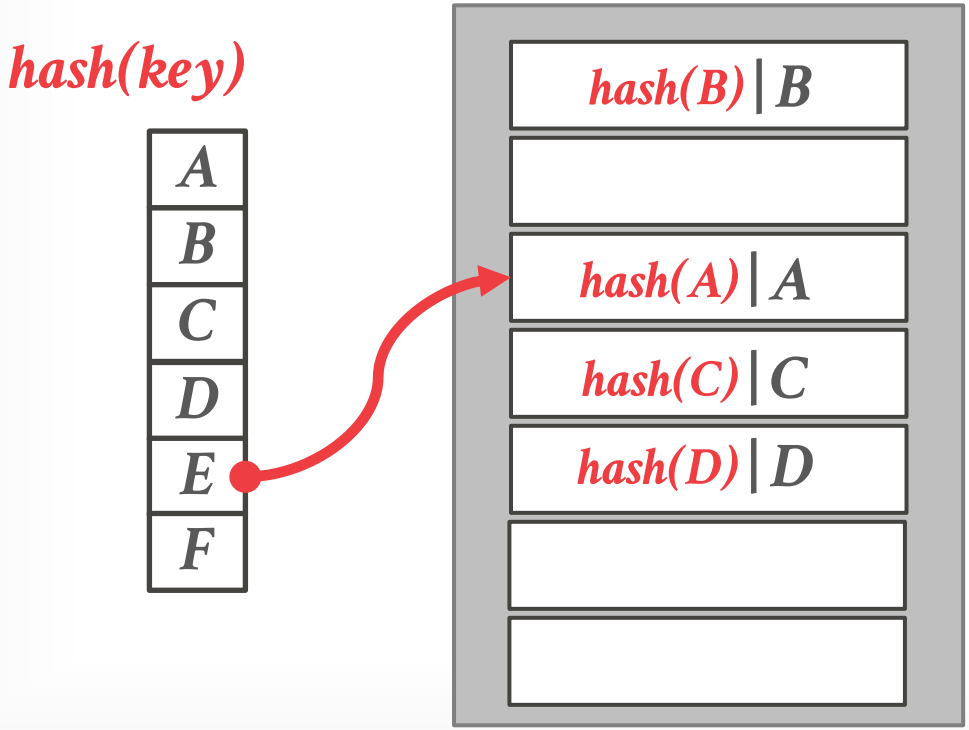
# LINEAR PROBE HASHING



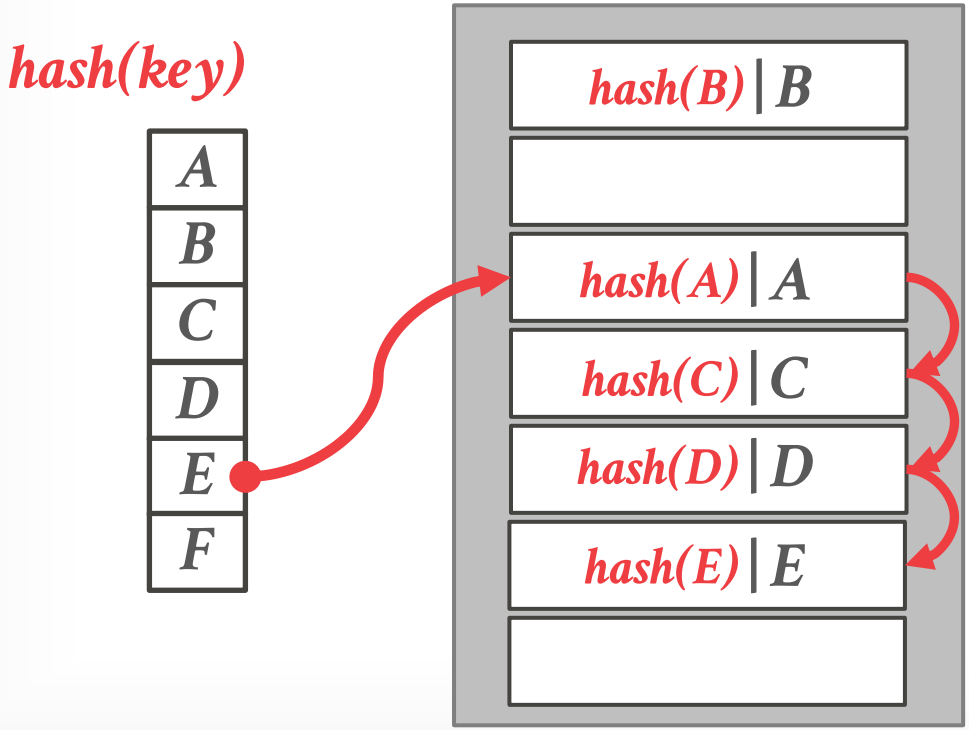
# LINEAR PROBE HASHING



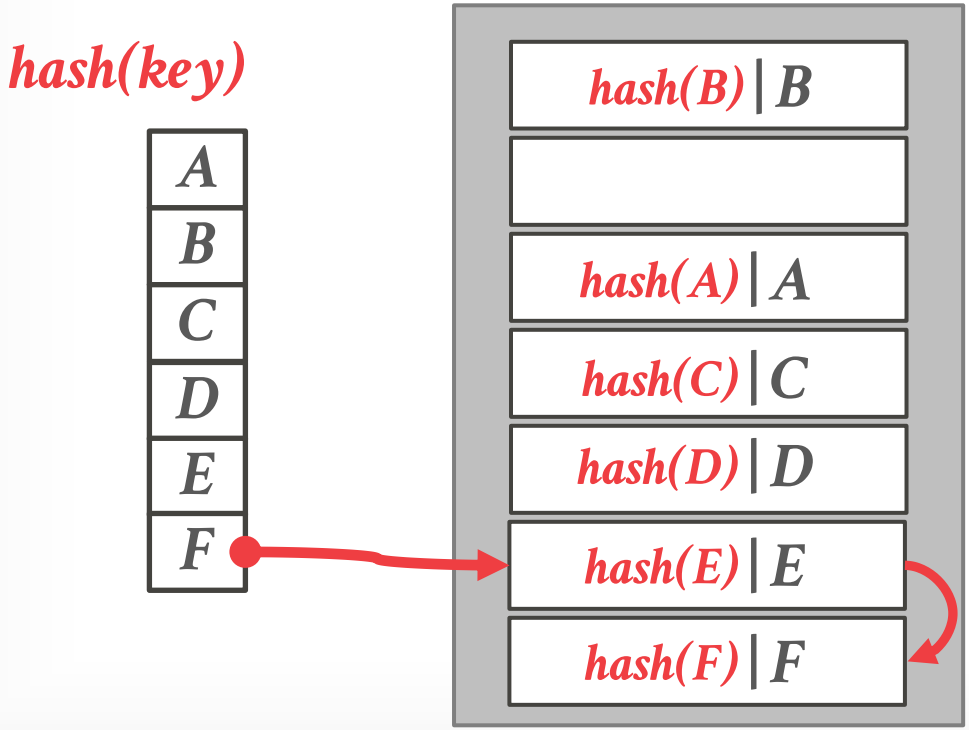
# LINEAR PROBE HASHING



# LINEAR PROBE HASHING



# LINEAR PROBE HASHING



# OBSERVATION

To reduce the number of wasteful comparisons during the build/probe phases, it is important to avoid collisions of hashed keys.

This requires a hash table with ~2× the number of slots as the number of elements in **R**.

# ROBIN HOOD HASHING

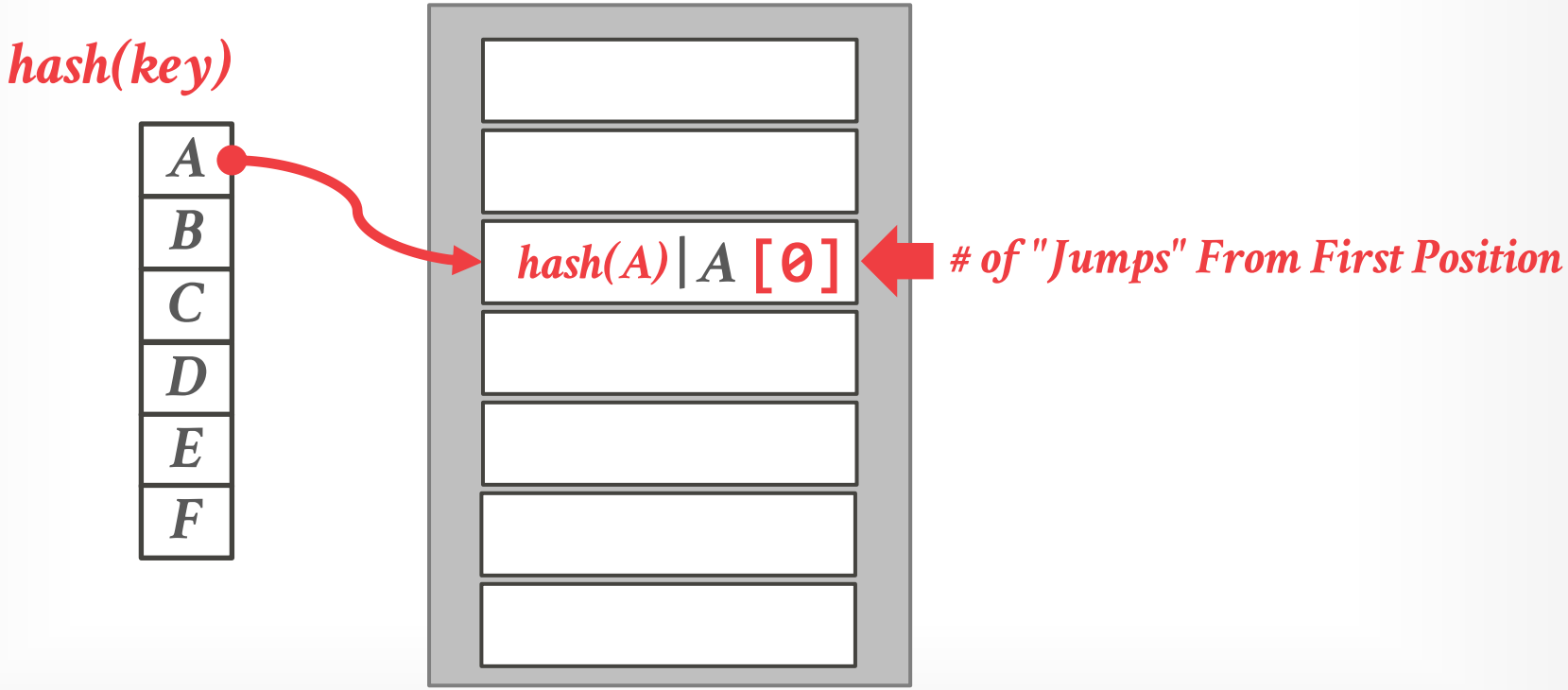
Variant of linear probe hashing that steals slots from "rich" keys and give them to "poor" keys.

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

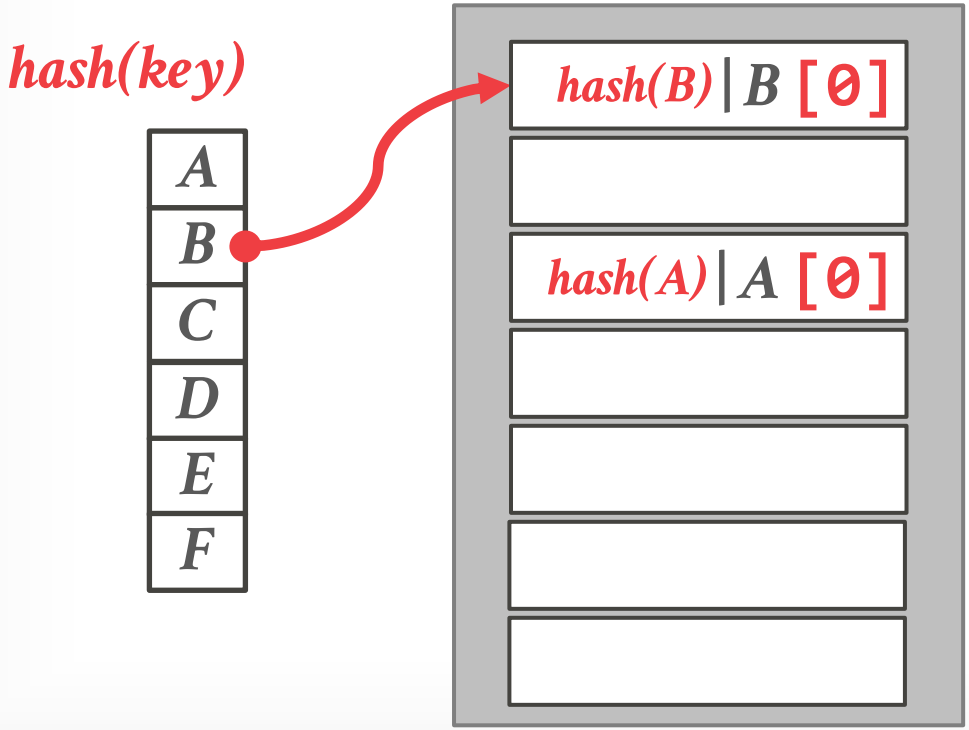
→ On insert, a key takes the slot of another key if the first key is farther away from its optimal position than the second key.

ROBIN HOOD HASHING
FOUNDATIONS OF COMPUTER SCIENCE 1985

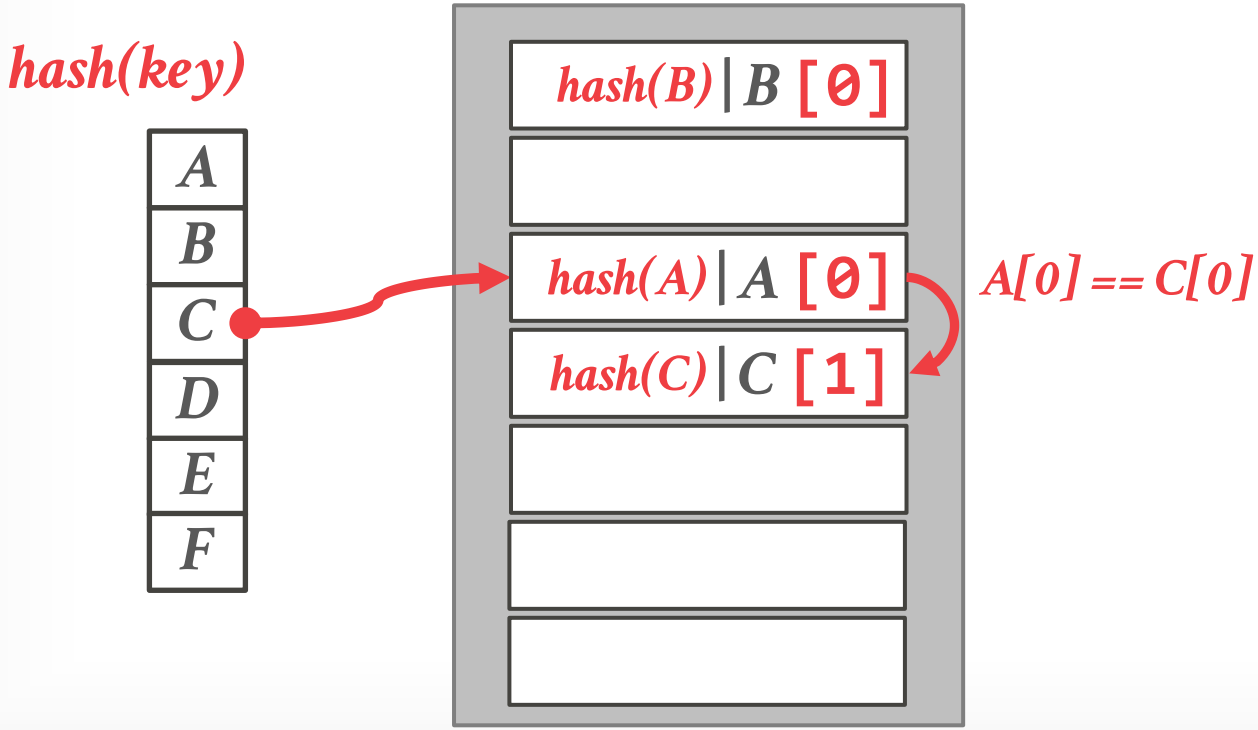
# ROBIN HOOD HASHING



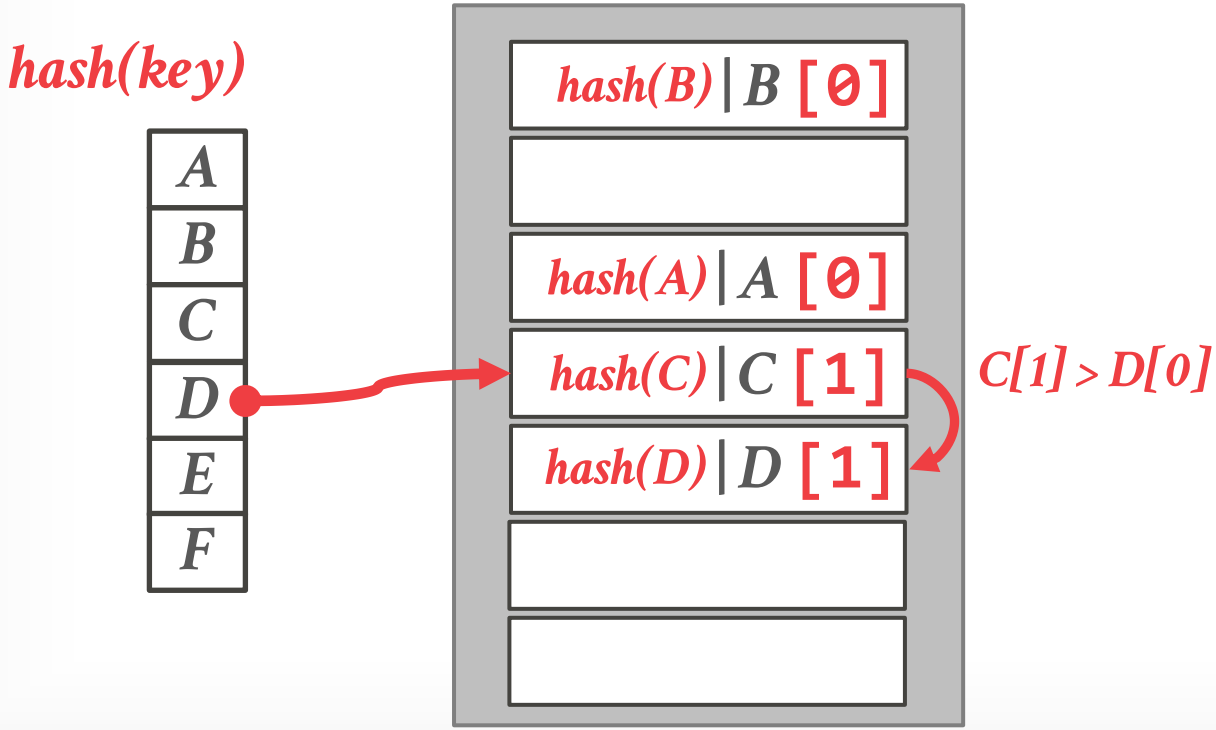
# ROBIN HOOD HASHING



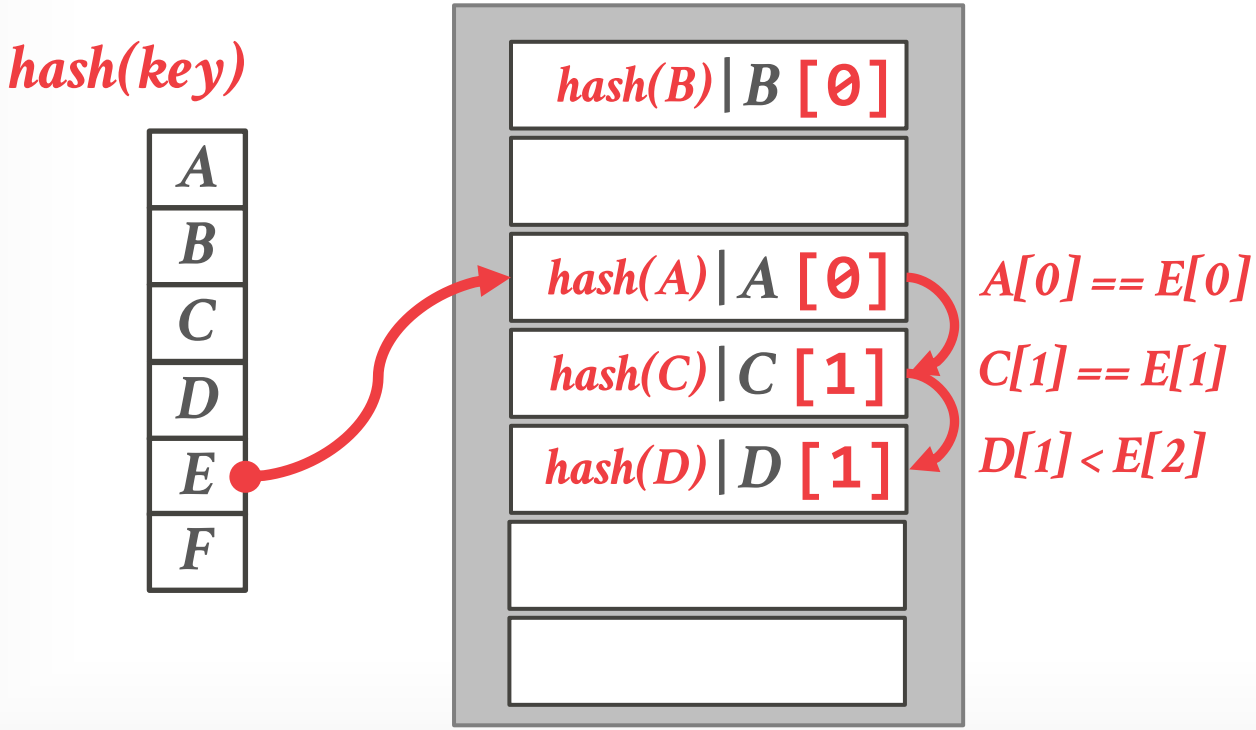
# ROBIN HOOD HASHING



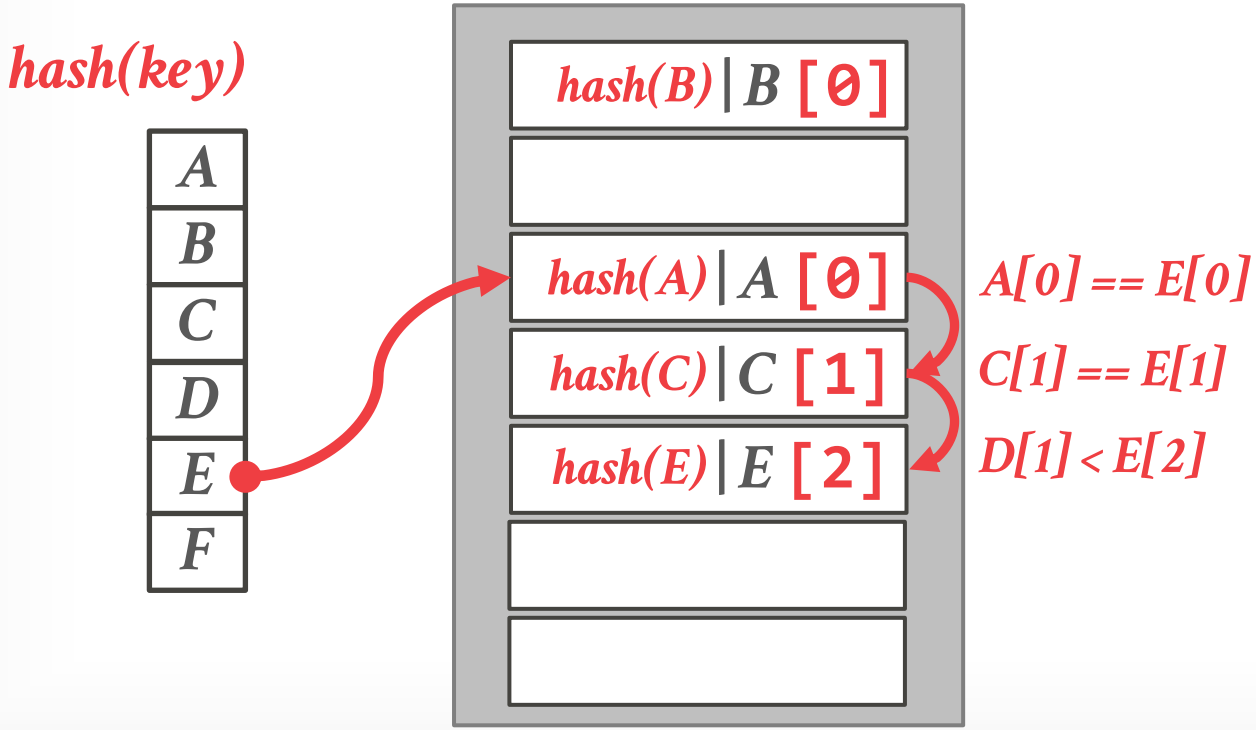
# ROBIN HOOD HASHING



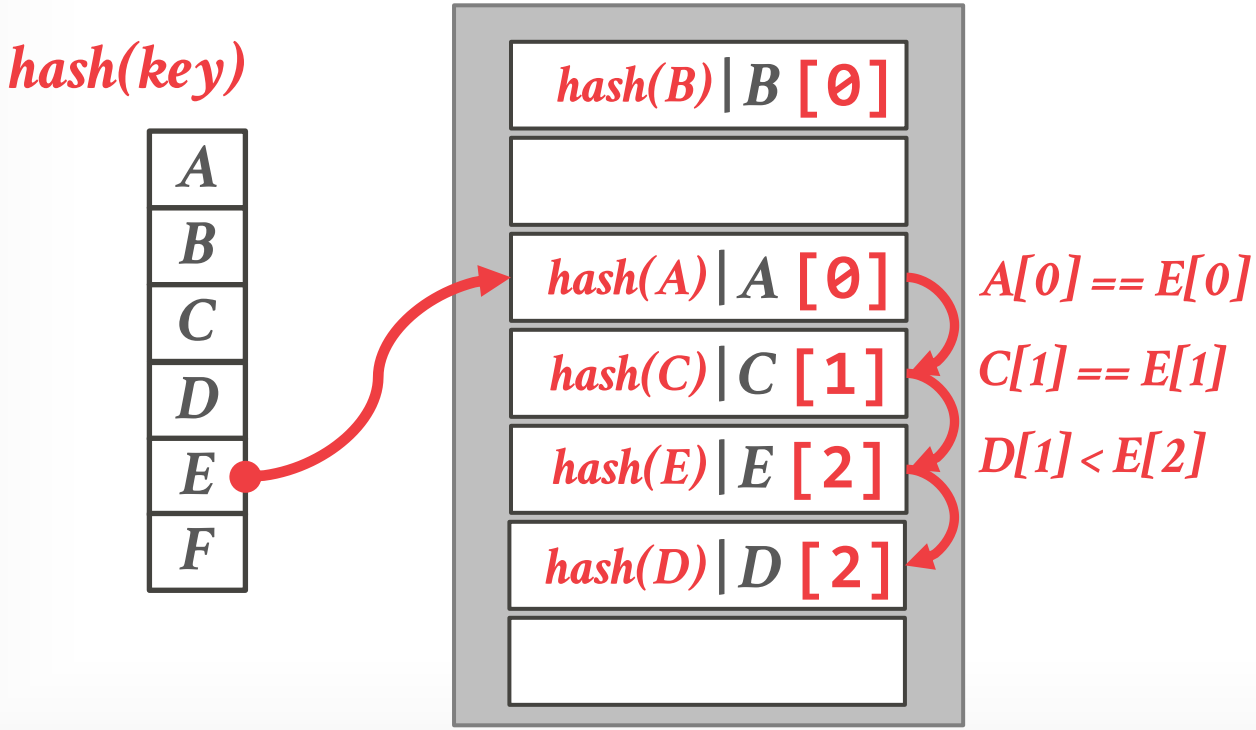
# ROBIN HOOD HASHING



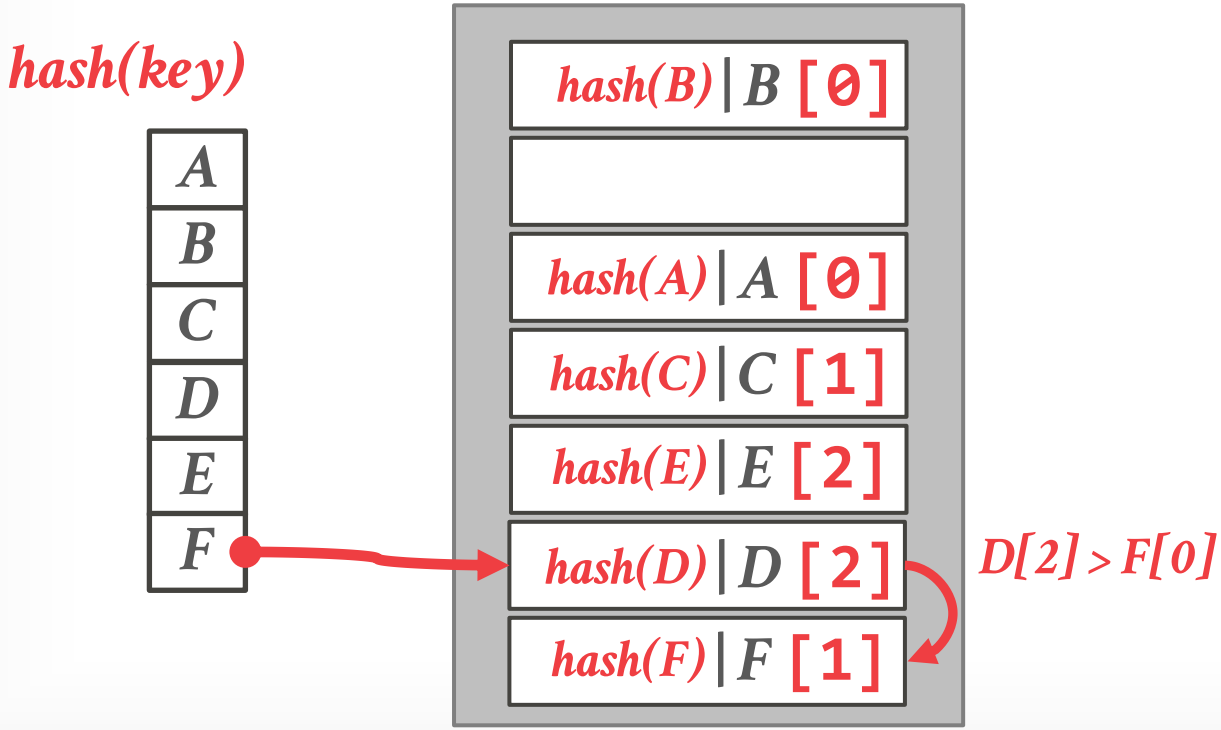
# ROBIN HOOD HASHING



# ROBIN HOOD HASHING



# ROBIN HOOD HASHING



# HOPSCOTCH HASHING

Variant of linear probe hashing where keys can move between positions in a **neighborhood**.

→ A neighborhood is contiguous range of slots in the table.
→ The size of a neighborhood is a configurable constant (ideally a single cache-line).
→ A key is guaranteed to be in its neighborhood or not exist in the table.

The goal is to have the cost of accessing a neighborhood to be the same as finding a key.

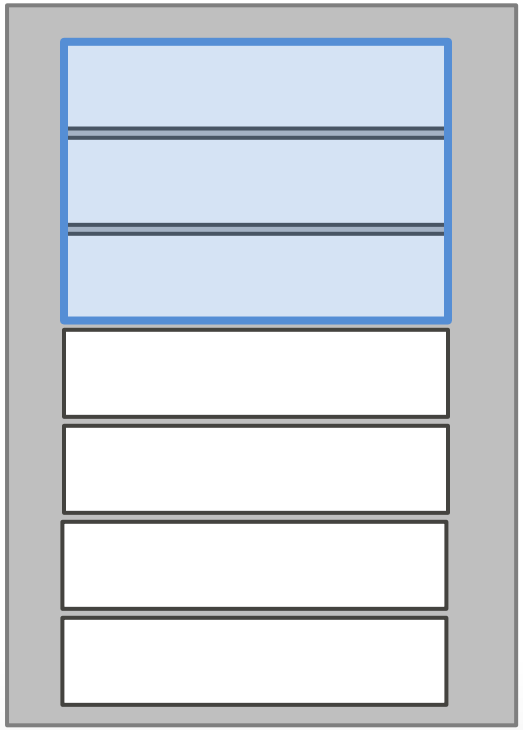
HOPSCOTCH HASHING
SYMPOSIUM ON DISTRIBUTED COMPUTING 2008

# HOPSCOTCH HASHING

*Neighborhood Size = 3*

*hash(key)*

| |
|---|
| *A* |
| *B* |
| *C* |
| *D* |
| *E* |
| *F* |



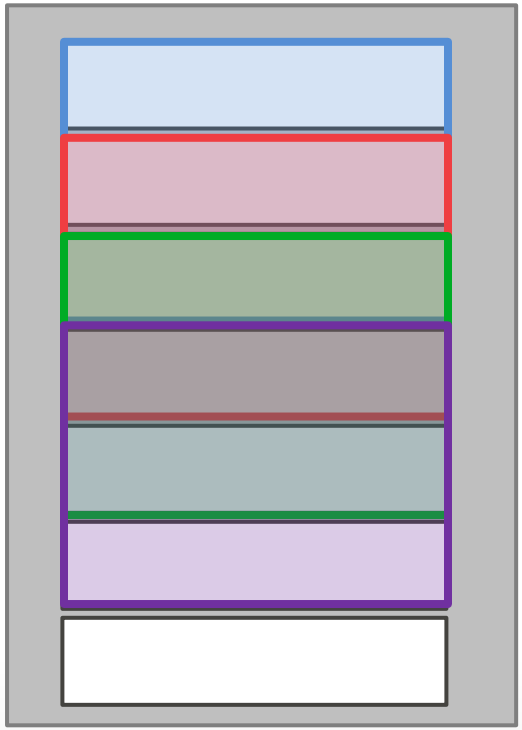
*Neighborhood #1*

# HOPSCOTCH HASHING

*Neighborhood Size = 3*

*hash(key)*

| |
|---|
| A |
| B |
| C |
| D |
| E |
| F |



*Neighborhood #1*

*Neighborhood #2*

*Neighborhood #3*

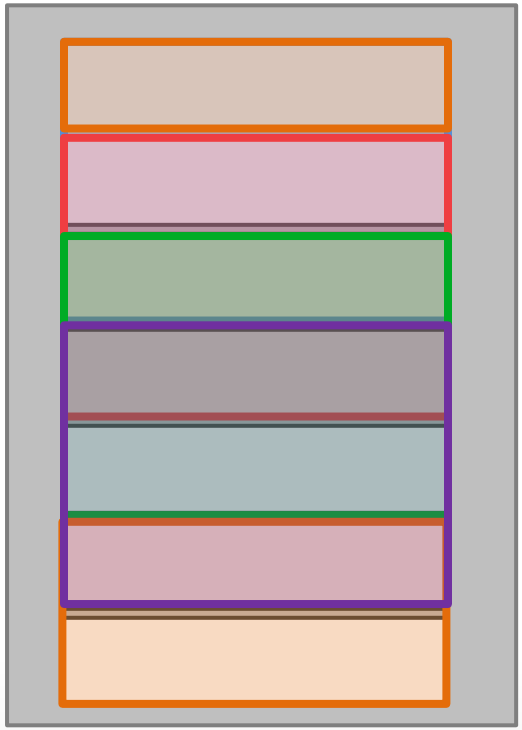
*Neighborhood #4*

⋮

# HOPSCOTCH HASHING

*hash(key)*

| |
|---|
| A |
| B |
| C |
| D |
| E |
| F |



*Neighborhood Size = 3*

*Neighborhood #1* *Neighborhood #6*

*Neighborhood #2*

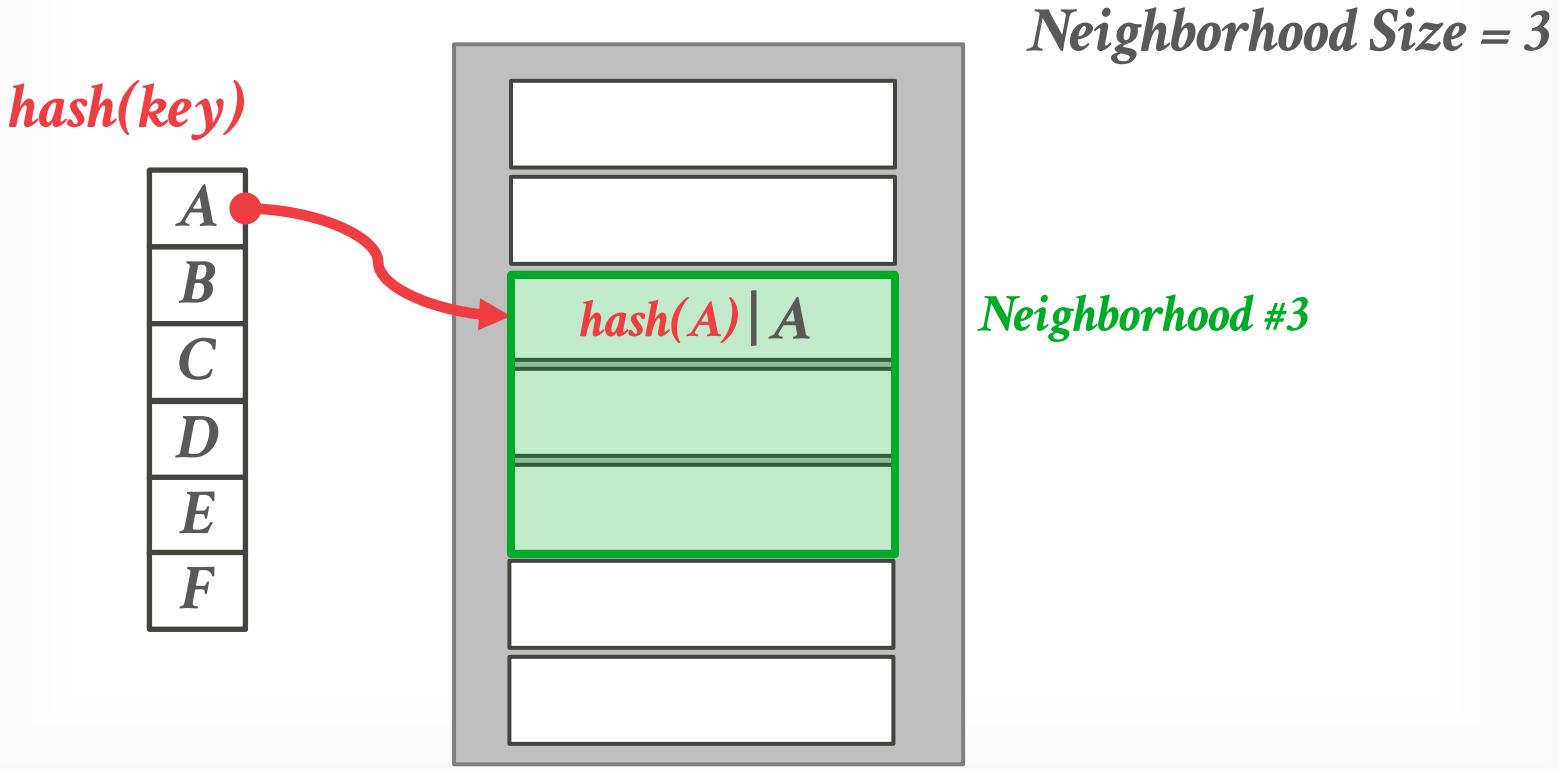
*Neighborhood #3*

*Neighborhood #4*

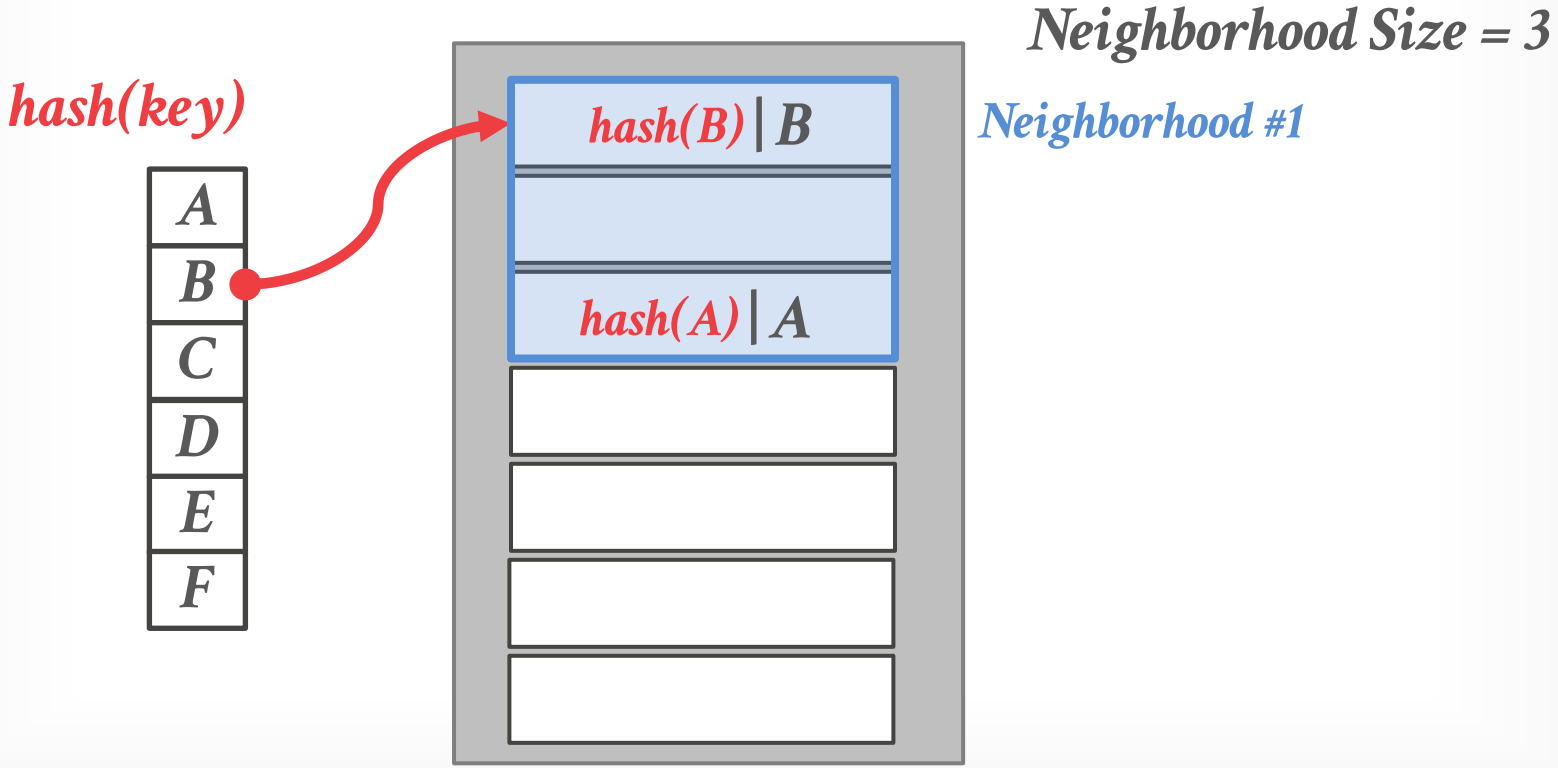
⋮

*Neighborhood #6*

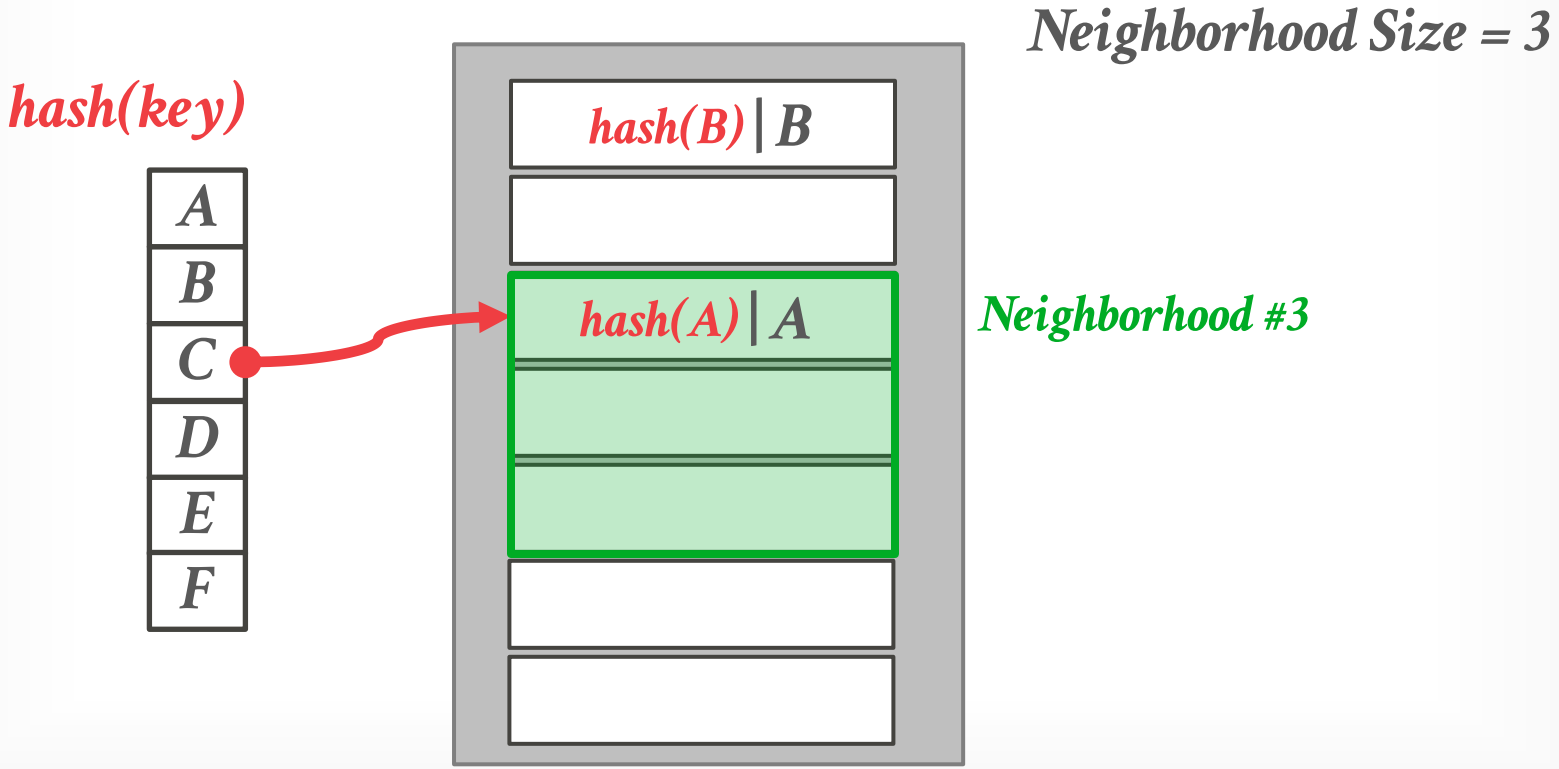
# HOPSCOTCH HASHING



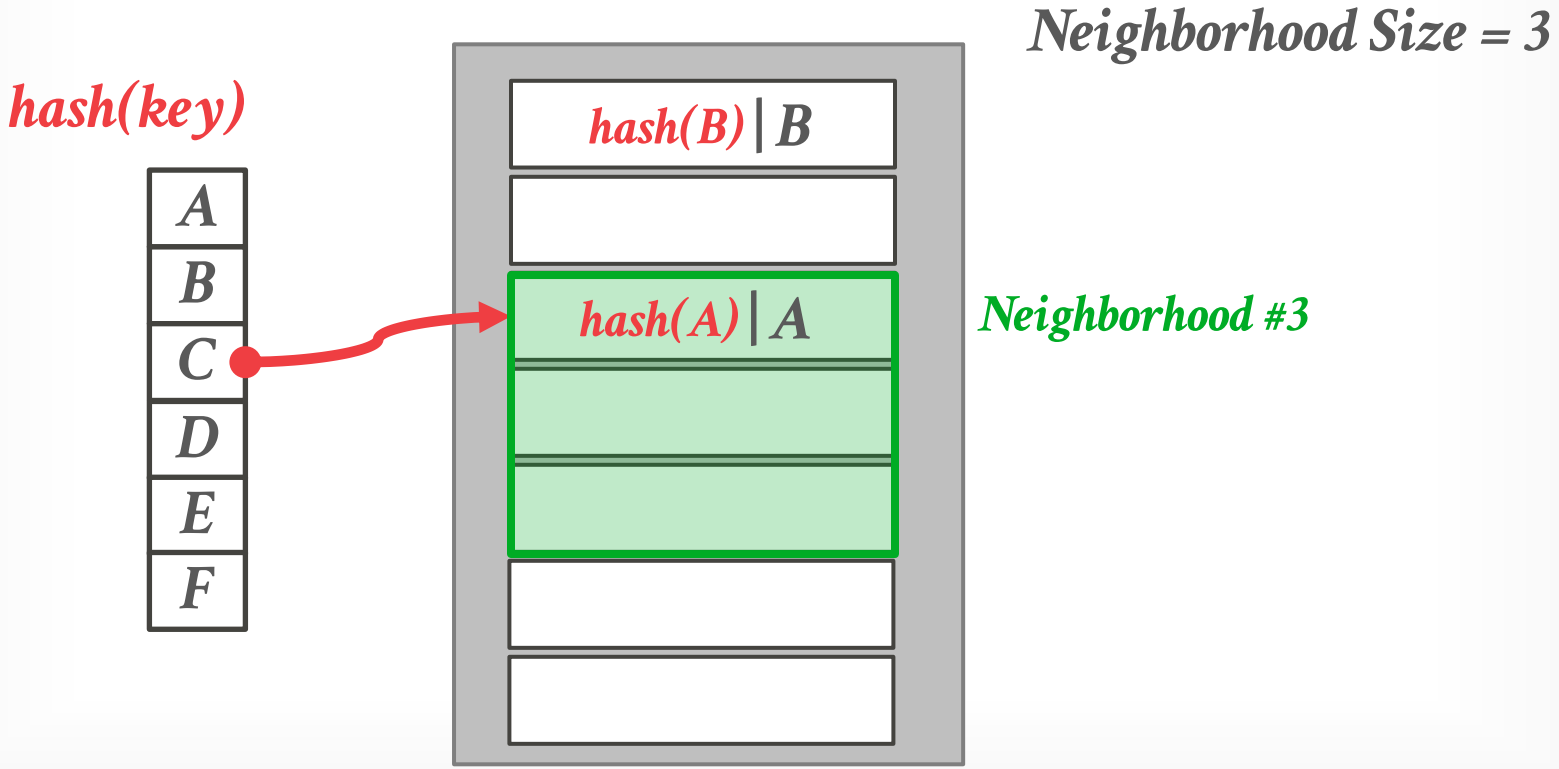
# HOPSCOTCH HASHING



# HOPSCOTCH HASHING



# HOPSCOTCH HASHING

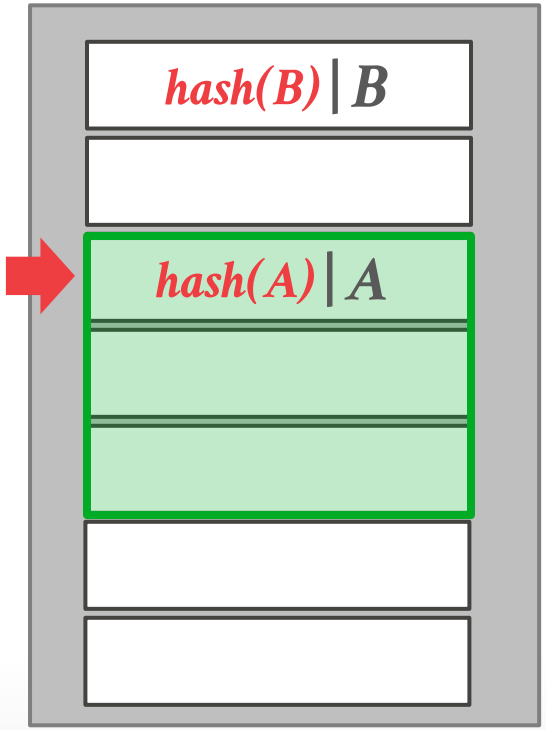


# HOPSCOTCH HASHING

*Neighborhood Size = 3*

*hash(key)*

| |
|---|
| *A* |
| *B* |
| *C* |
| *D* |
| *E* |
| *F* |



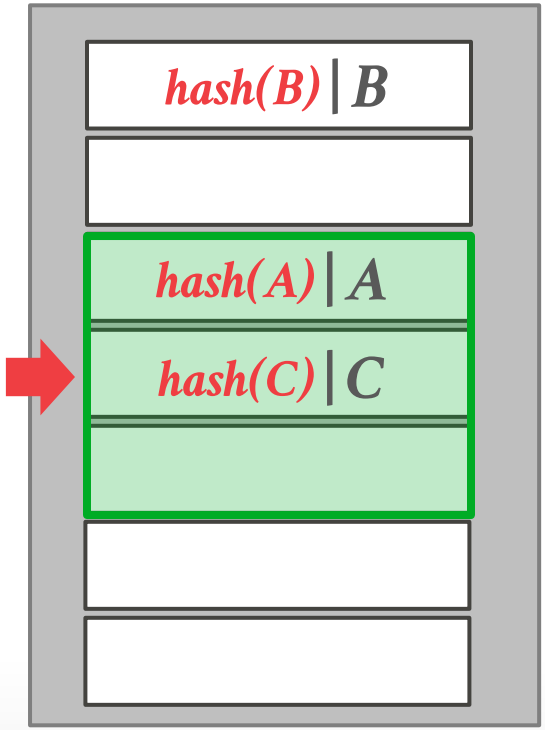

*Neighborhood #3*

# HOPSCOTCH HASHING

*Neighborhood Size = 3*

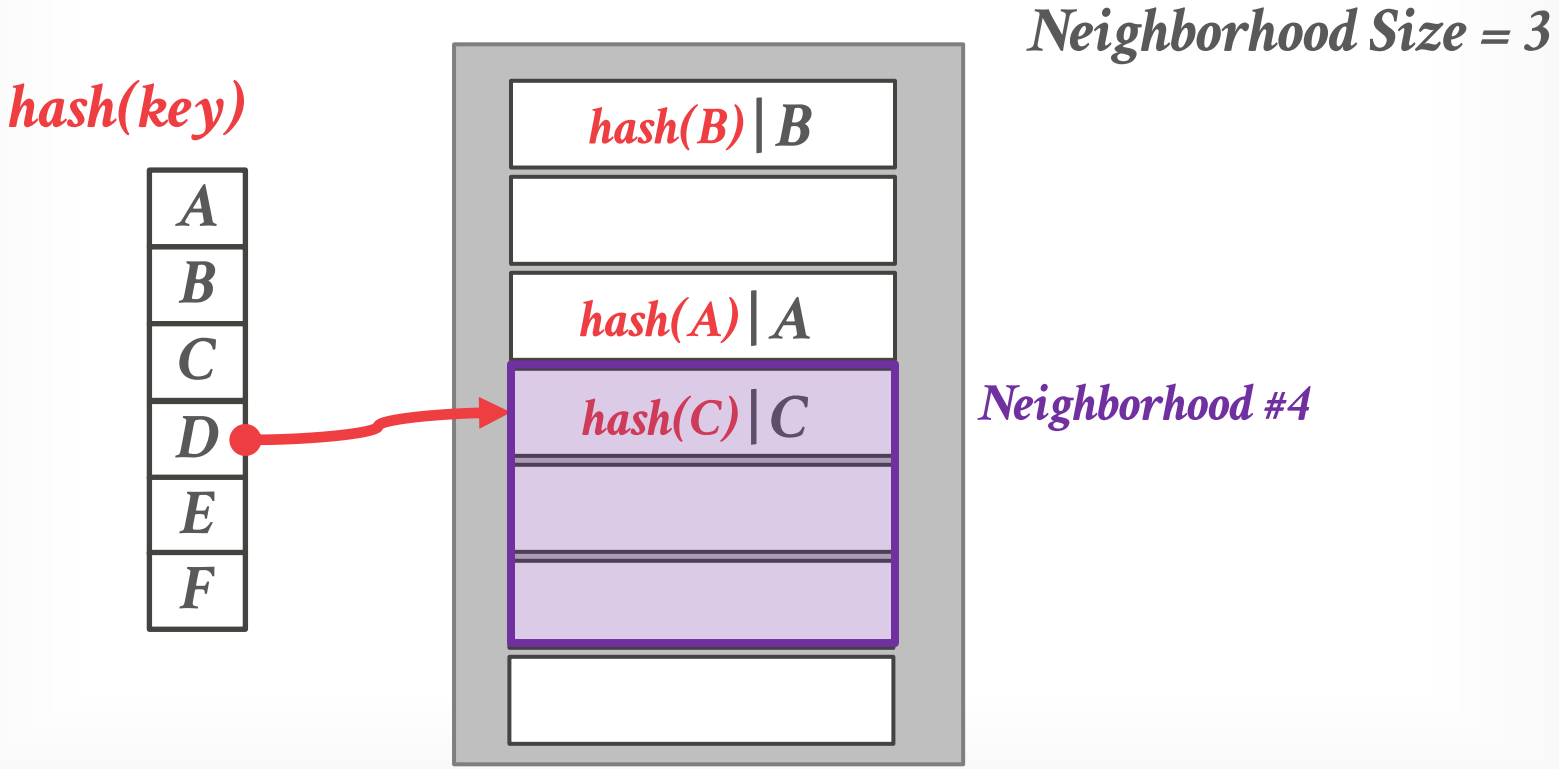
*hash(key)*

| |
|---|
| *A* |
| *B* |
| *C* |
| *D* |
| *E* |
| *F* |




*Neighborhood #3*

# HOPSCOTCH HASHING

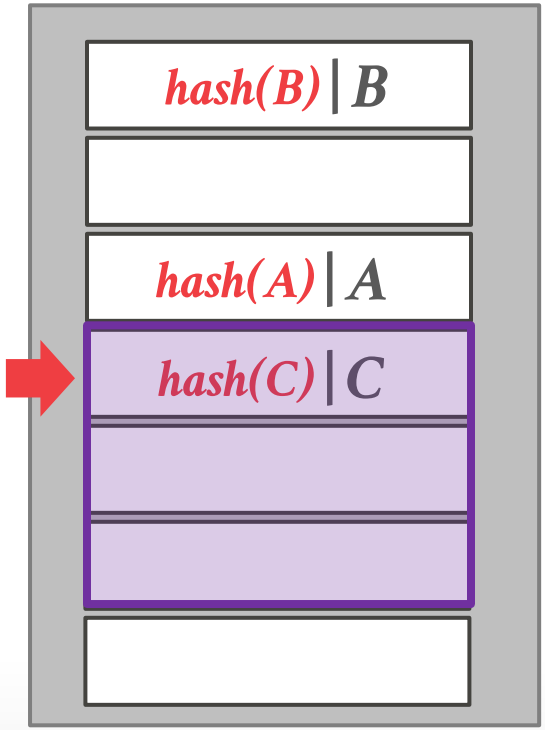


# HOPSCOTCH HASHING

*Neighborhood Size = 3*

*hash(key)*

| |
|---|
| *A* |
| *B* |
| *C* |
| *D* |
| *E* |
| *F* |

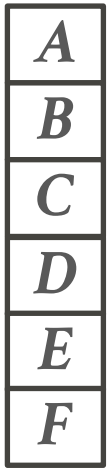


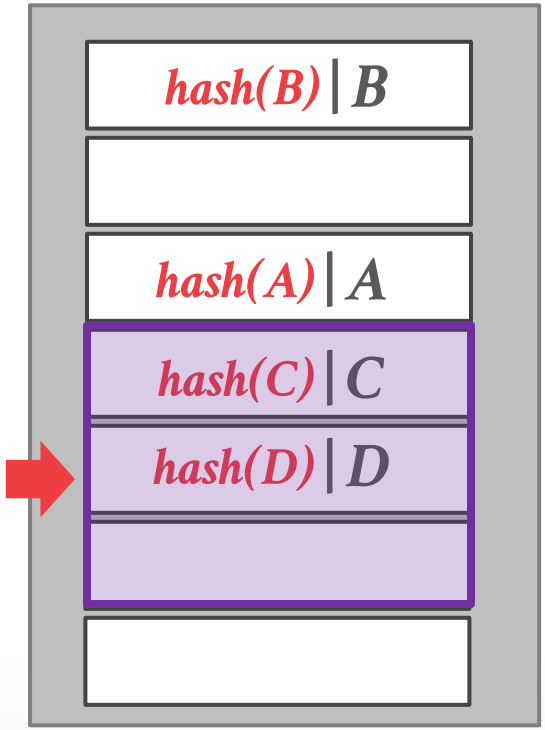

*Neighborhood #4*

# HOPSCOTCH HASHING

*Neighborhood Size = 3*

*hash(key)*

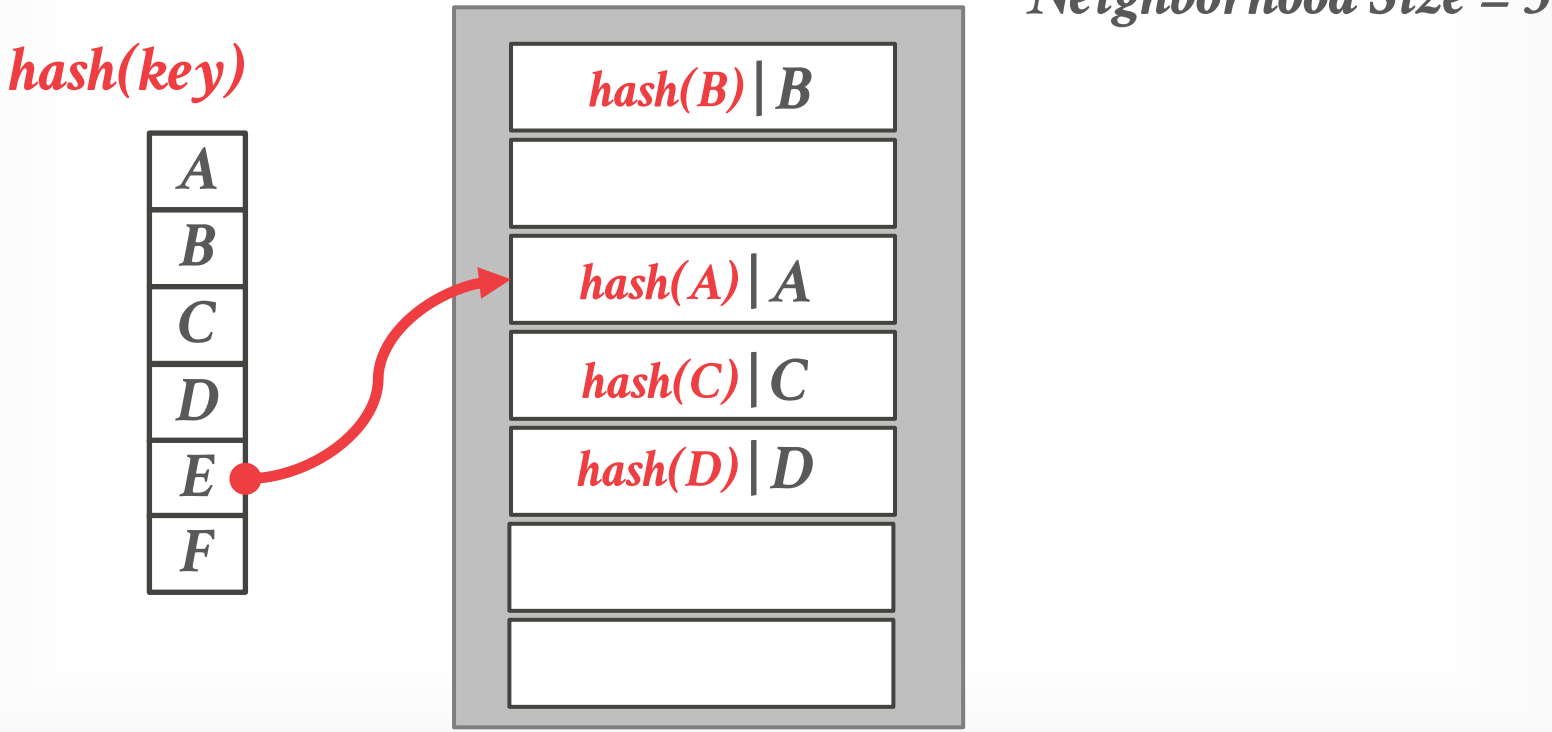


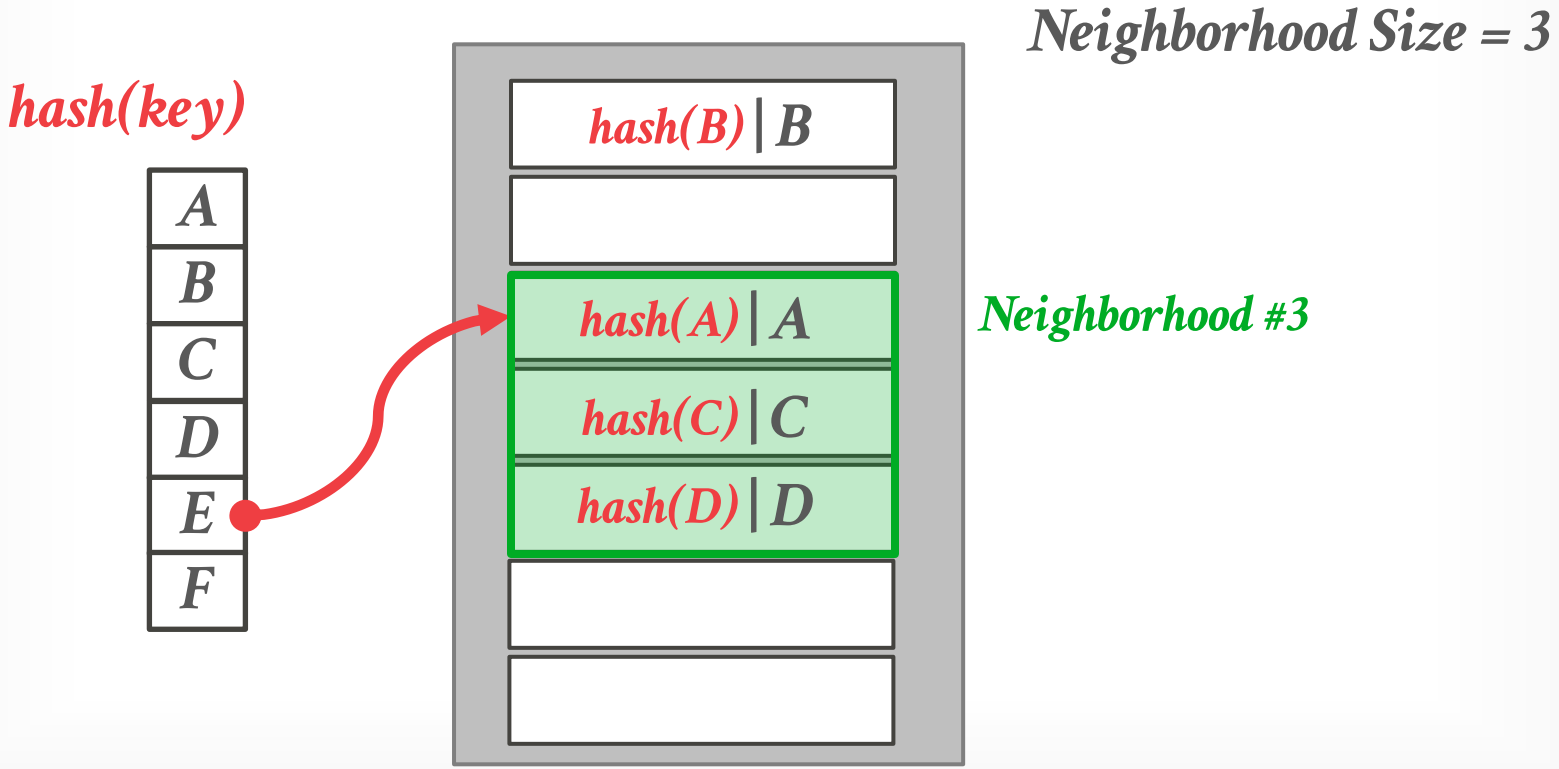

*Neighborhood #4*

# HOPSCOTCH HASHING

*Neighborhood Size = 3*



# HOPSCOTCH HASHING

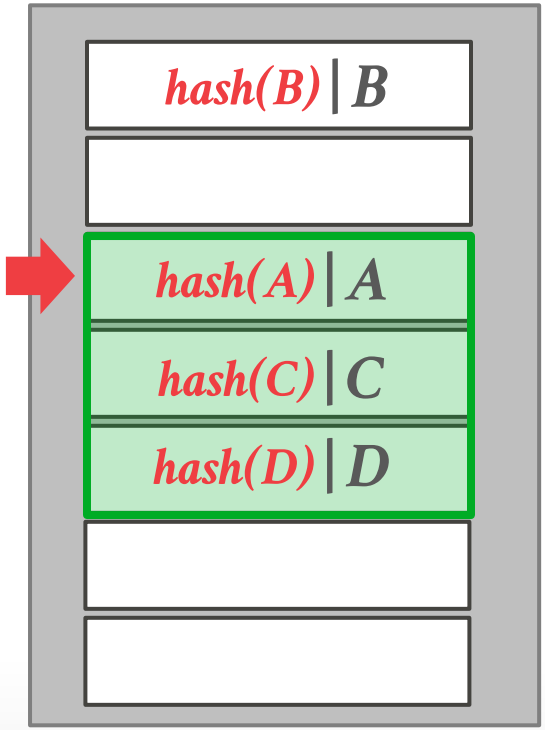


# HOPSCOTCH HASHING

*Neighborhood Size = 3*

*hash(key)*

| |
|---|
| *A* |
| *B* |
| *C* |
| *D* |
| *E* |
| *F* |

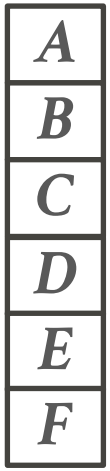


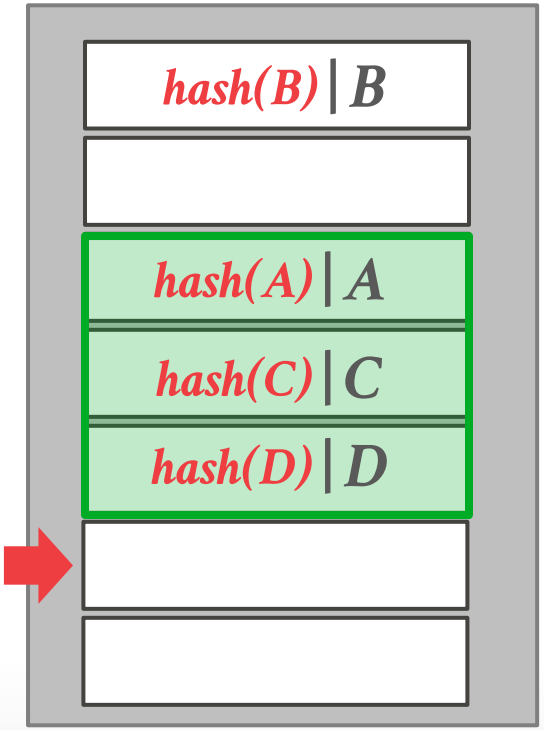

*Neighborhood #3*

# HOPSCOTCH HASHING

*Neighborhood Size = 3*

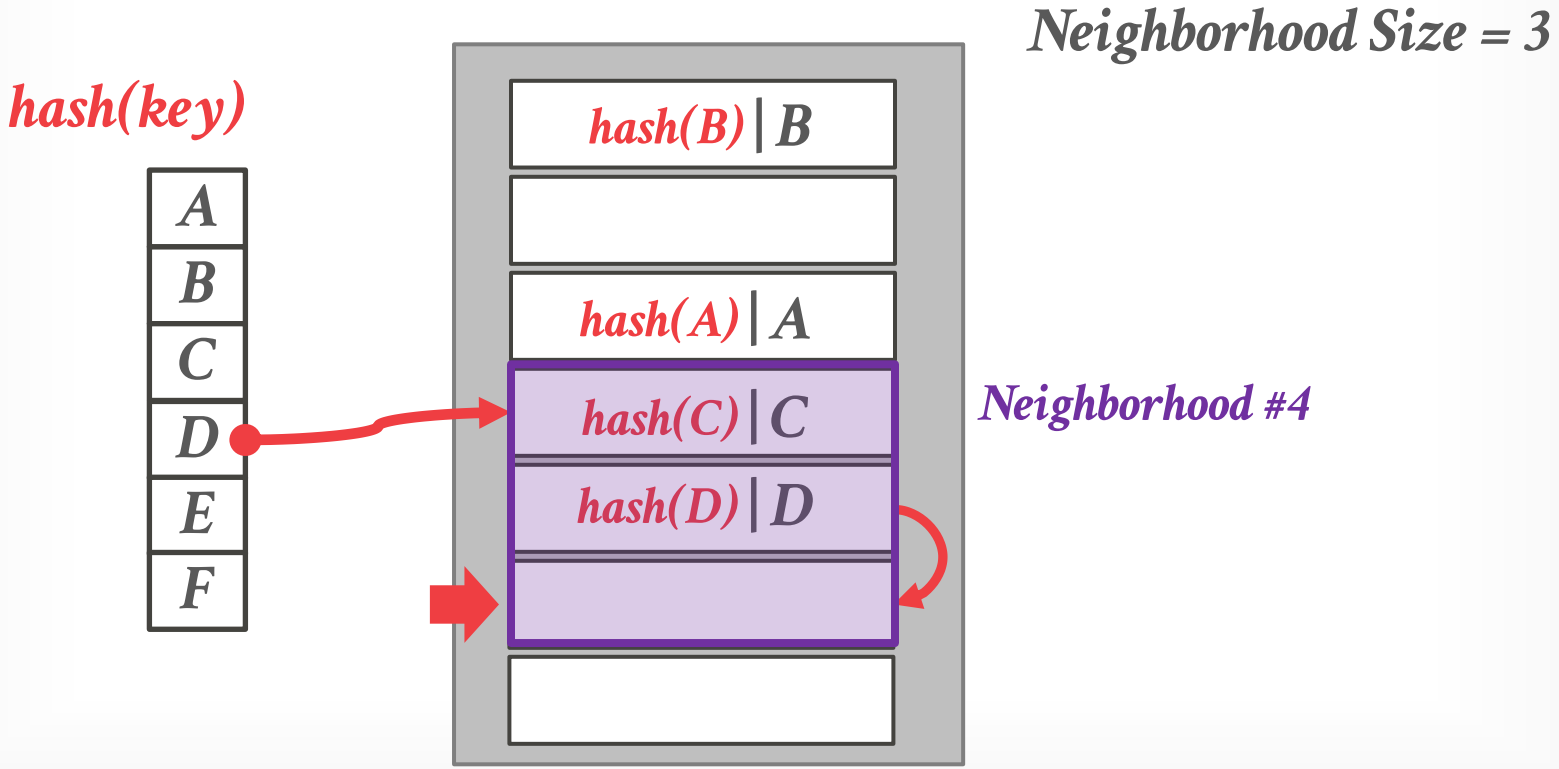
*hash(key)*



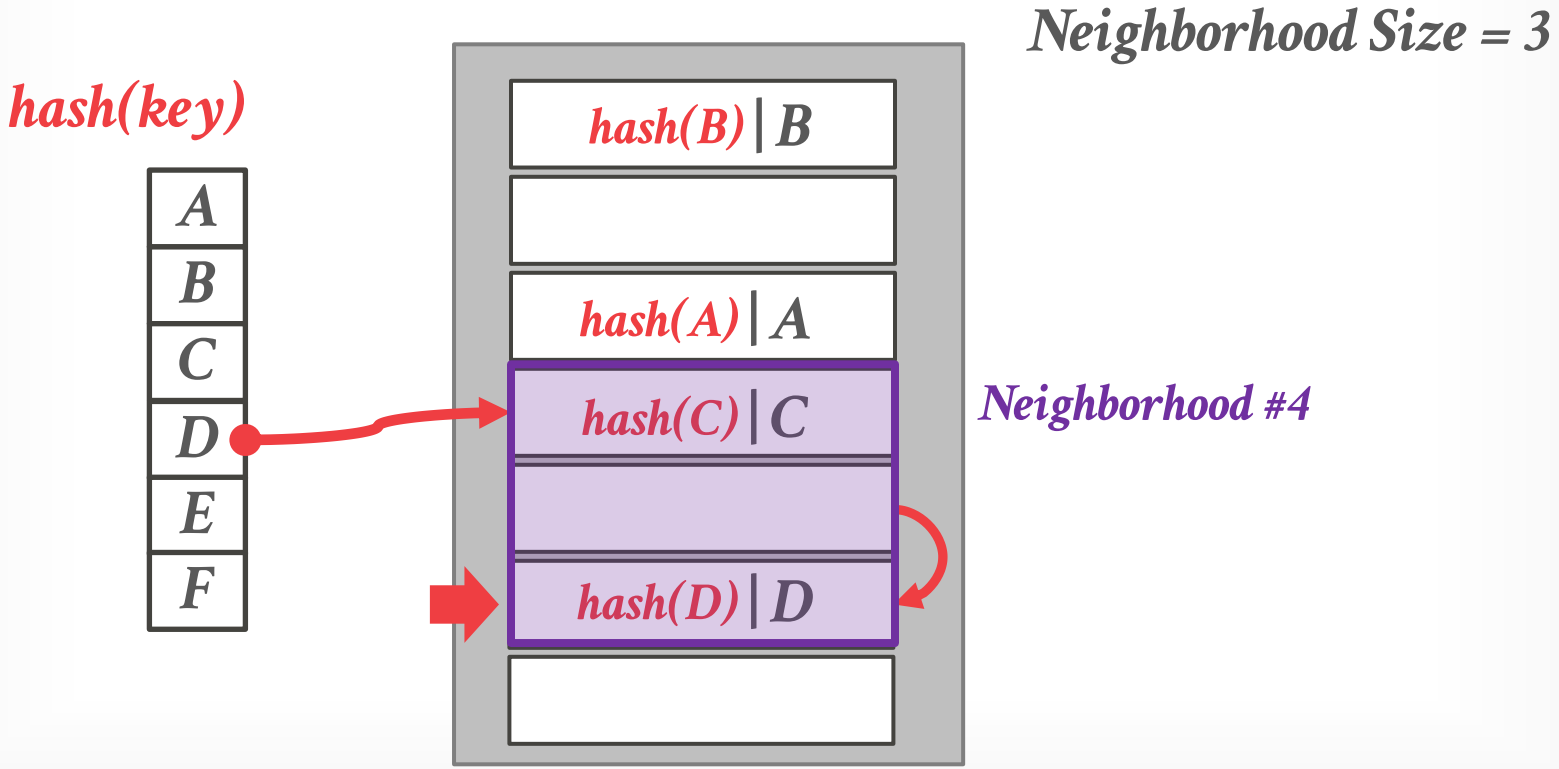




*Neighborhood #3*

# HOPSCOTCH HASHING



# HOPSCOTCH HASHING

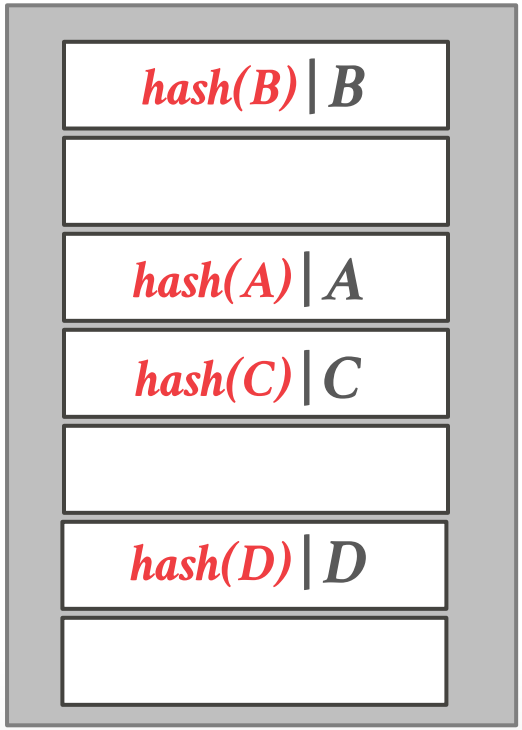


# HOPSCOTCH HASHING

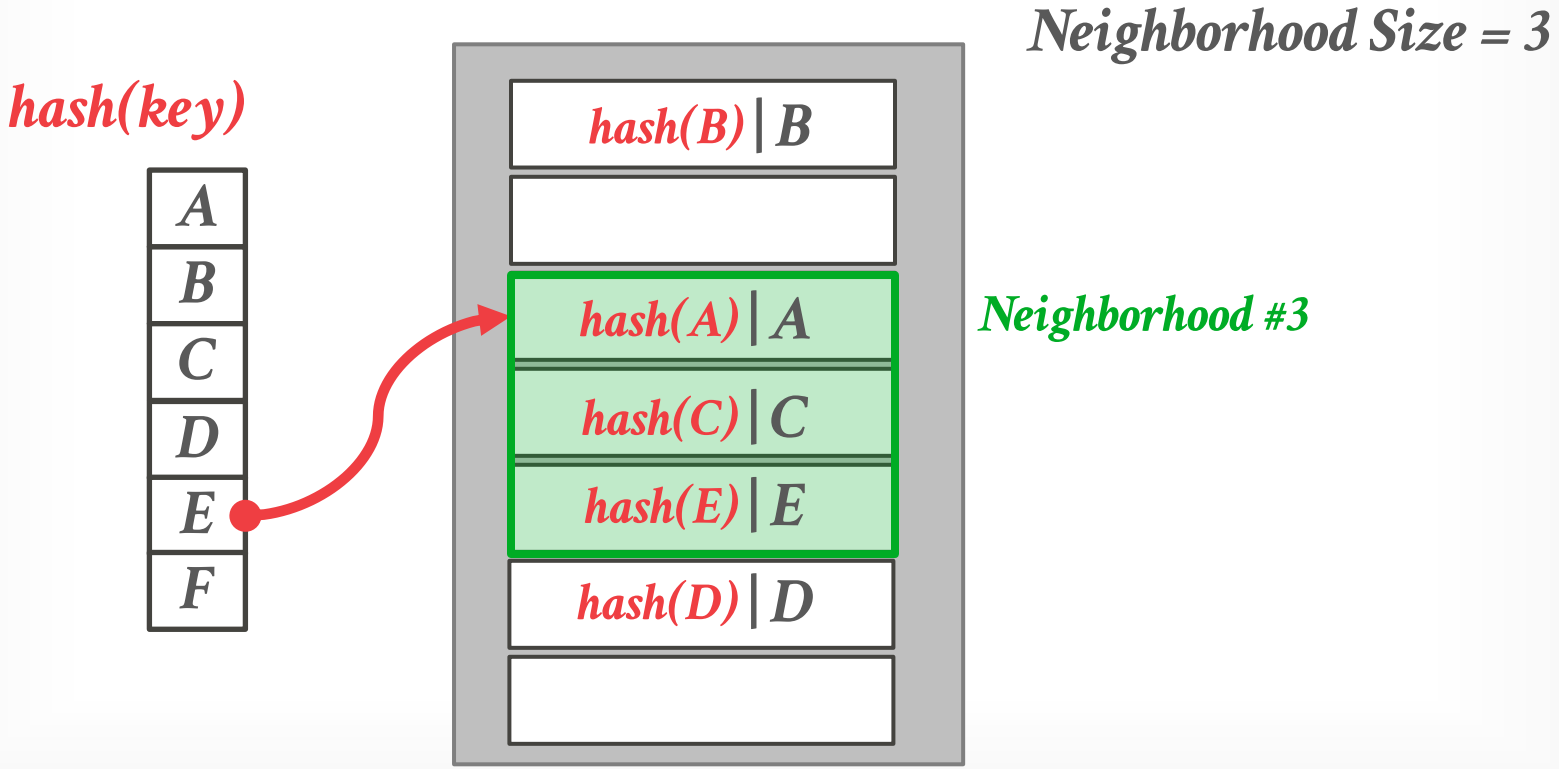
*Neighborhood Size = 3*

*hash(key)*

| |
|---|
| *A* |
| *B* |
| *C* |
| *D* |
| *E* |
| *F* |



# HOPSCOTCH HASHING

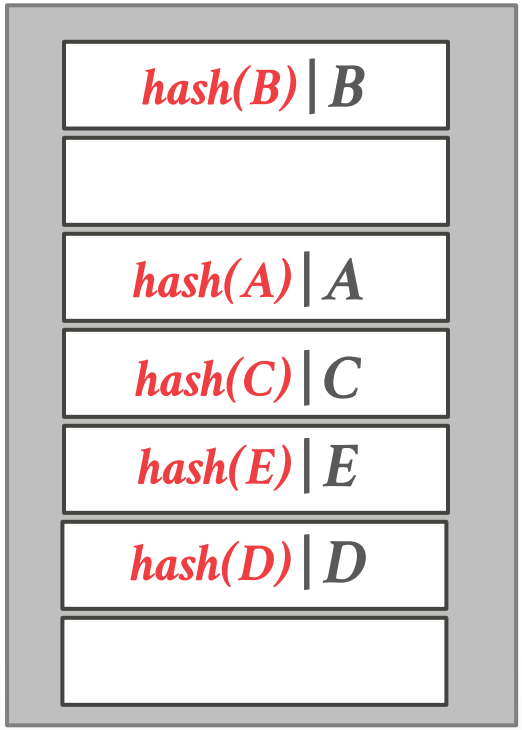


# HOPSCOTCH HASHING

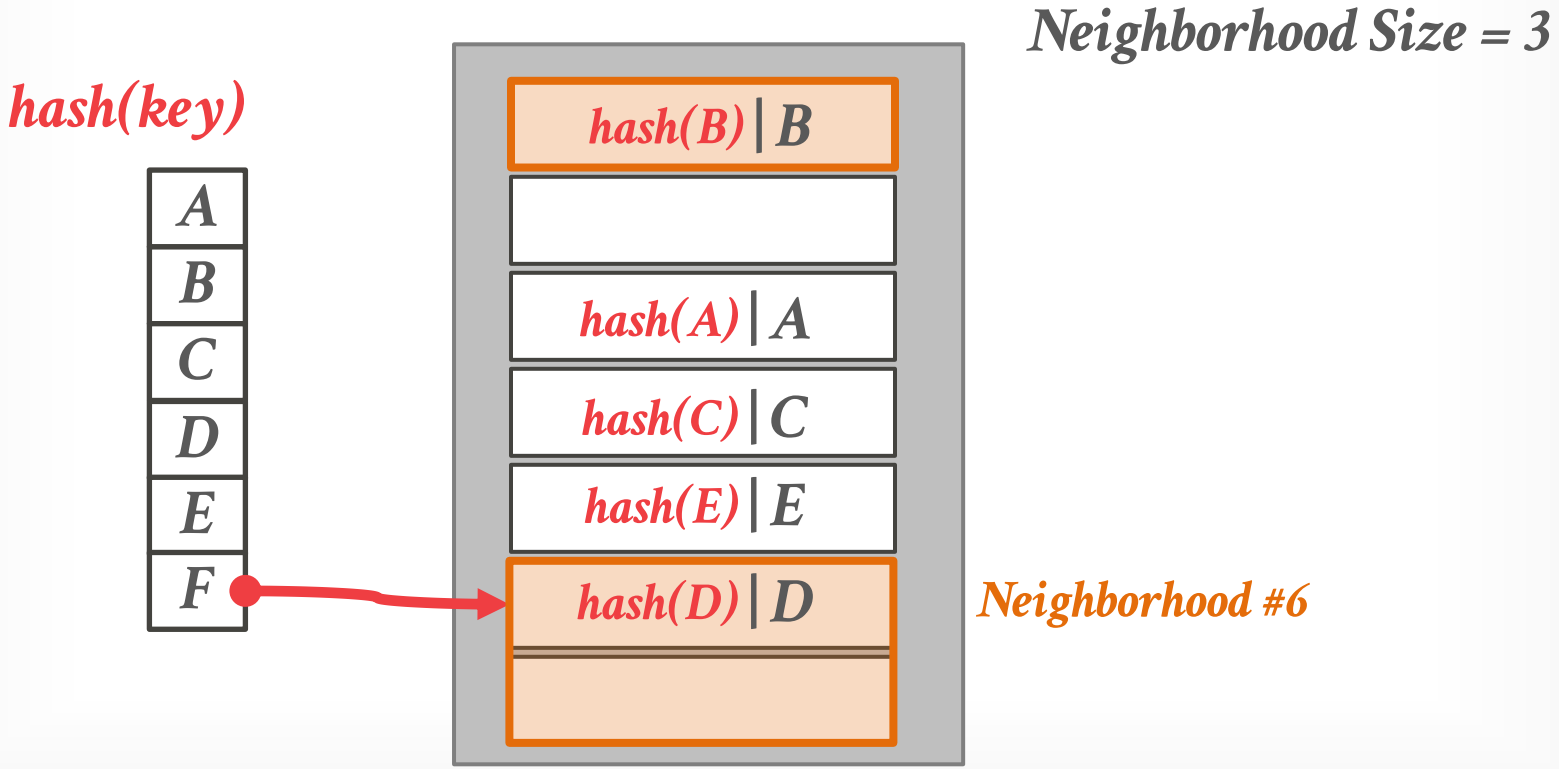
*Neighborhood Size = 3*

*hash(key)*

| |
|---|
| *A* |
| *B* |
| *C* |
| *D* |
| *E* |
| *F* |



# HOPSCOTCH HASHING

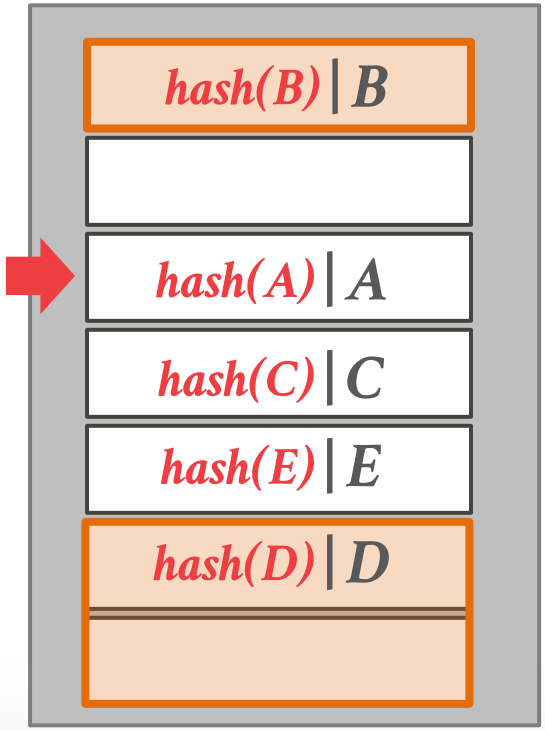


# HOPSCOTCH HASHING

*Neighborhood Size = 3*

*hash(key)*

| |
|---|
| A |
| B |
| C |
| D |
| E |
| F |



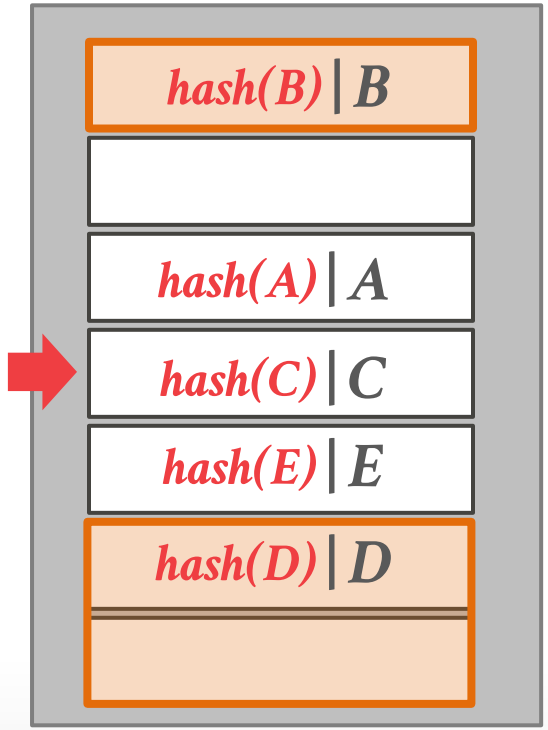

*Neighborhood #6*

# HOPSCOTCH HASHING

*Neighborhood Size = 3*

*hash(key)*

| |
|---|
| A |
| B |
| C |
| D |
| E |
| F |



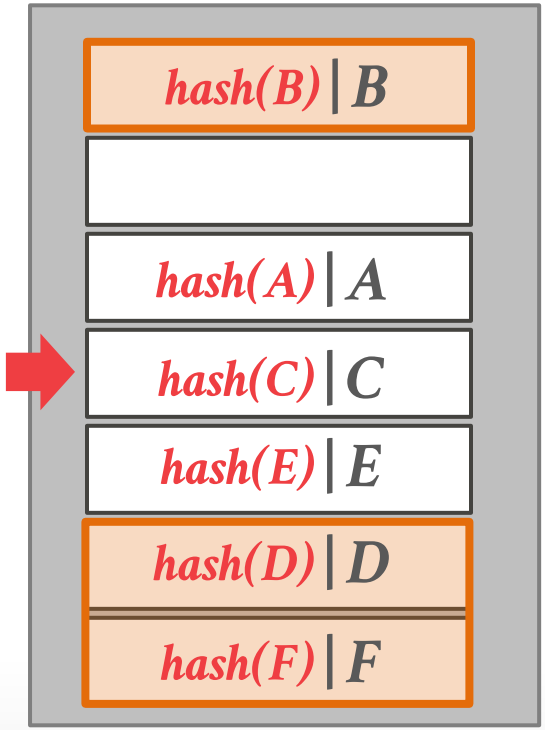

*Neighborhood #6*

# HOPSCOTCH HASHING

*Neighborhood Size = 3*

*hash(key)*

| |
|---|
| *A* |
| *B* |
| *C* |
| *D* |
| *E* |
| *F* |



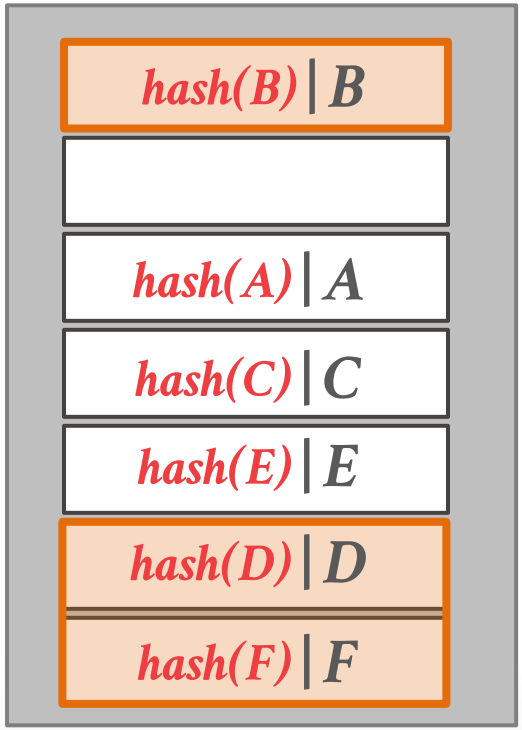

*Neighborhood #6*

# HOPSCOTCH HASHING

*Neighborhood Size = 3*

*hash(key)*

| |
|---|
| *A* |
| *B* |
| *C* |
| *D* |
| *E* |
| *F* |




*Neighborhood #6*

# CUCKOO HASHING

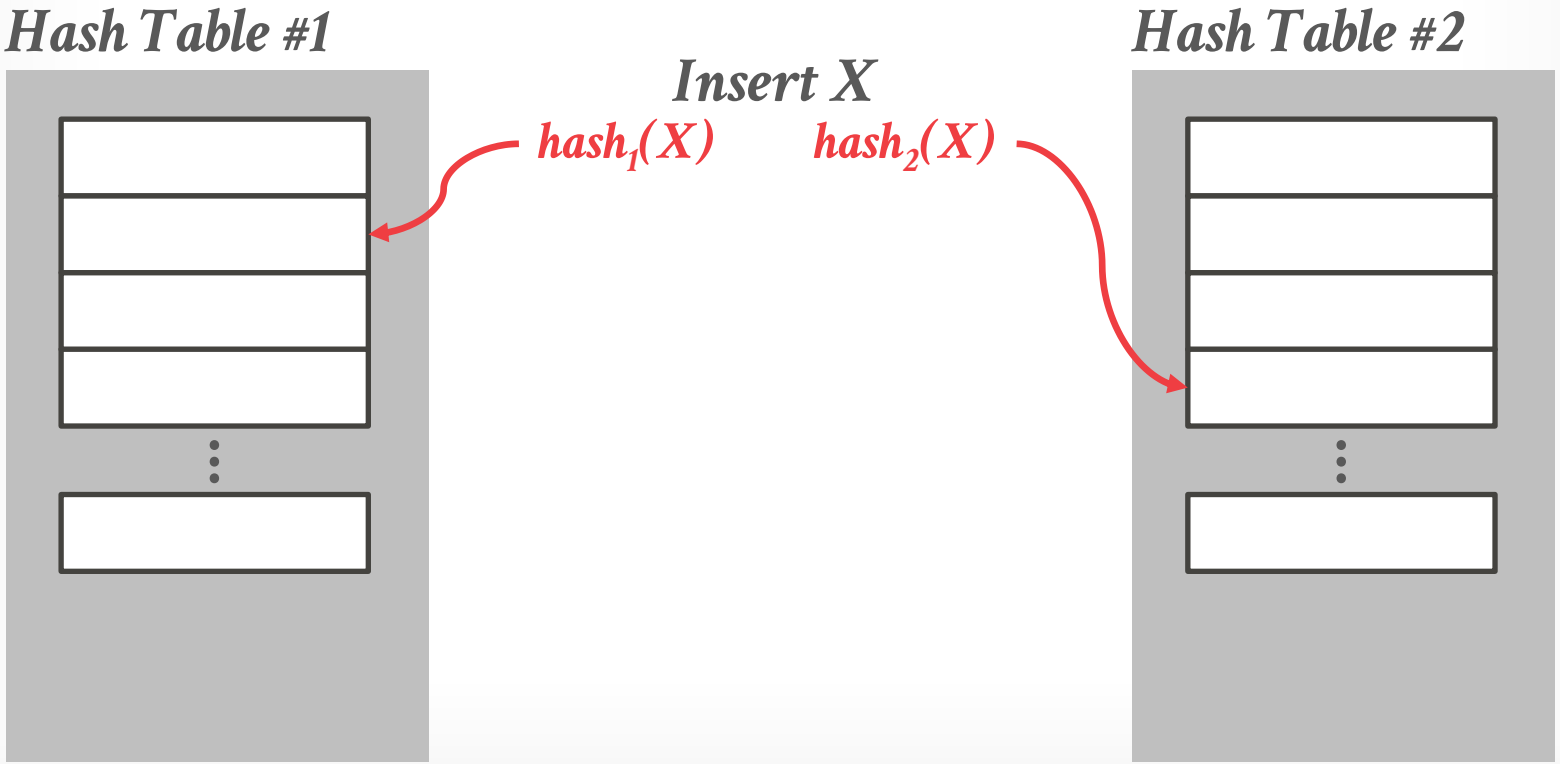
Use multiple tables with different hash functions.

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

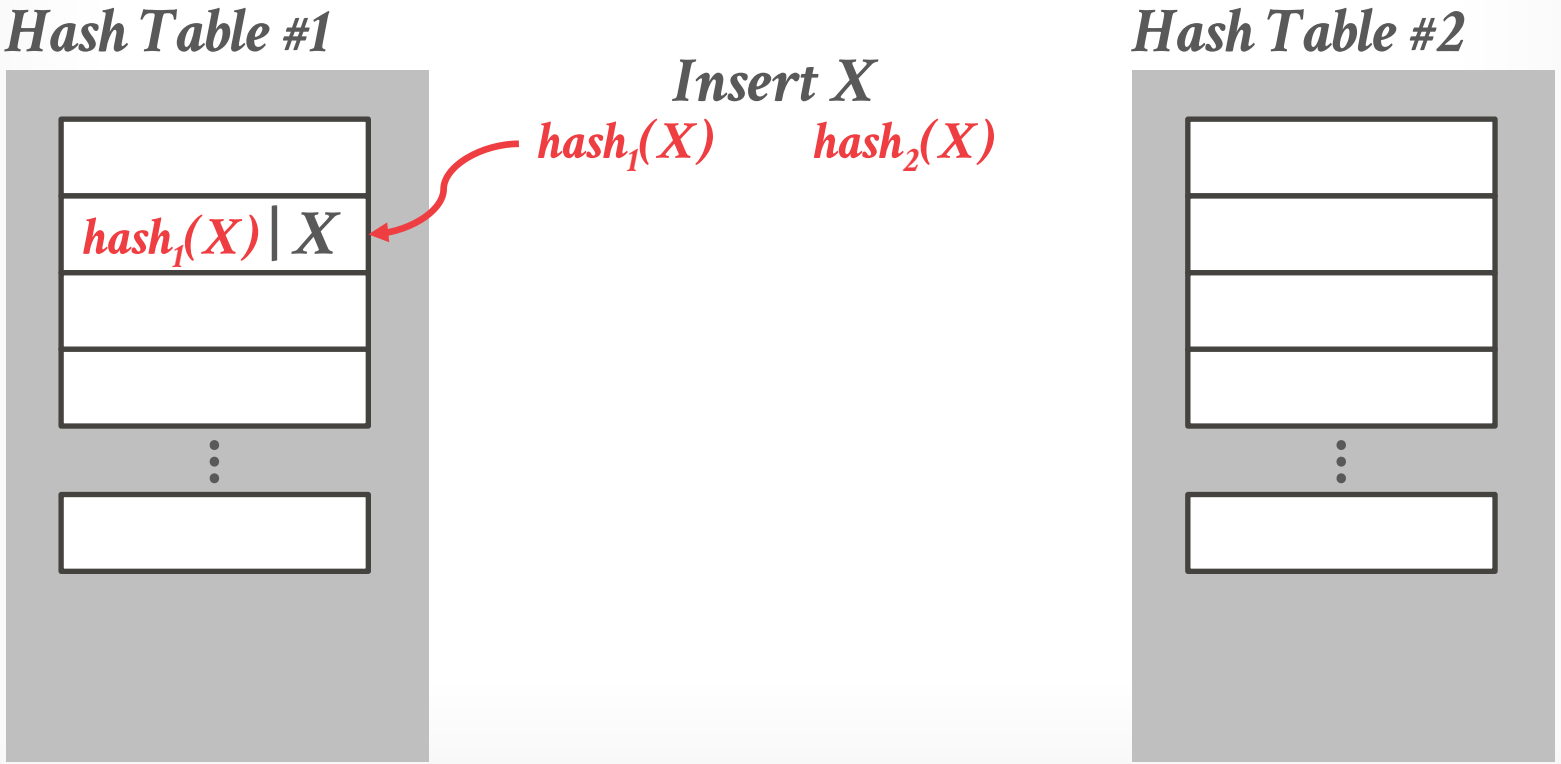
→ If no table has a free slot, evict the element from one of them and then re-hash it find a new location.

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

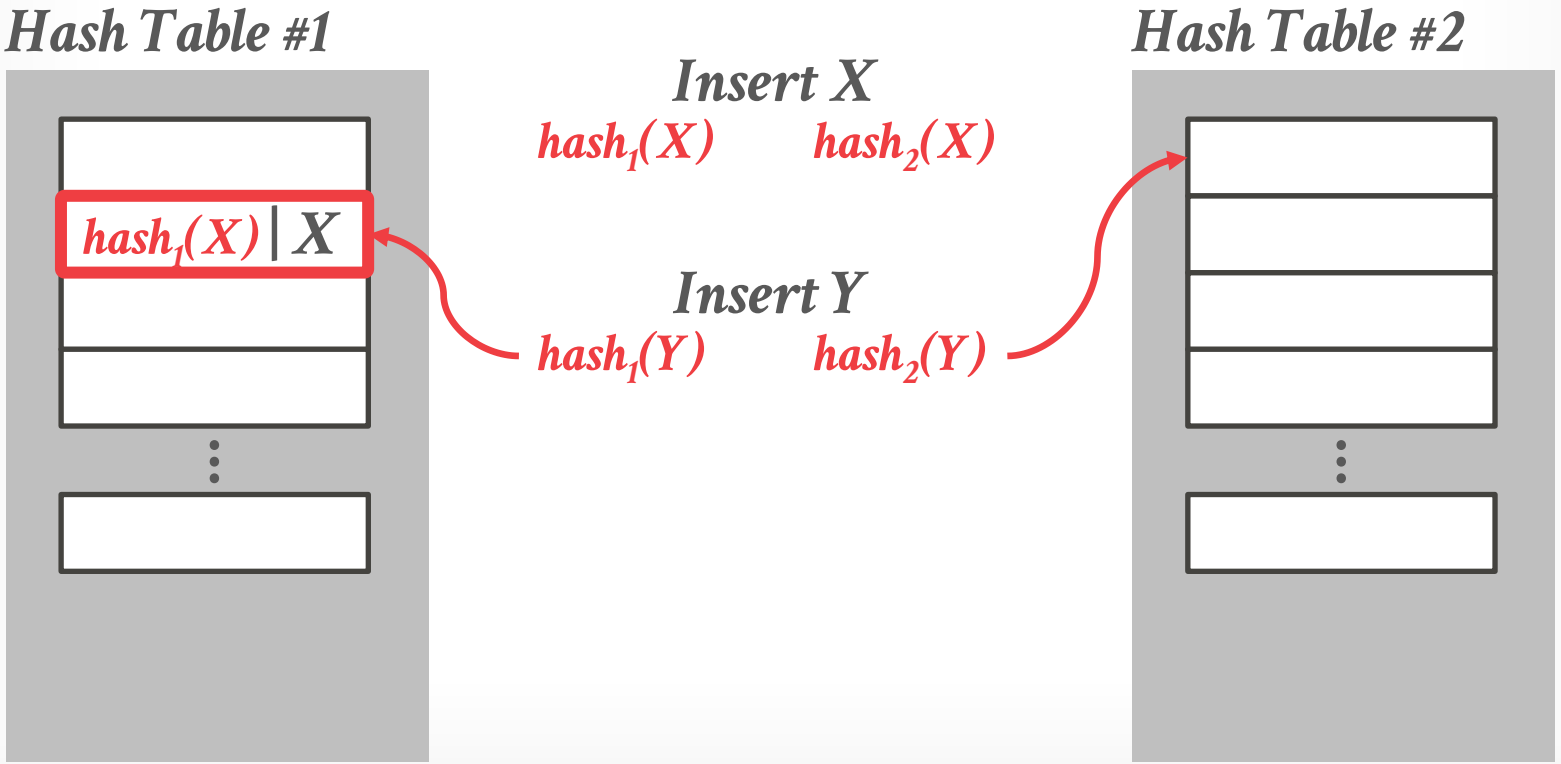
# CUCKOO HASHING



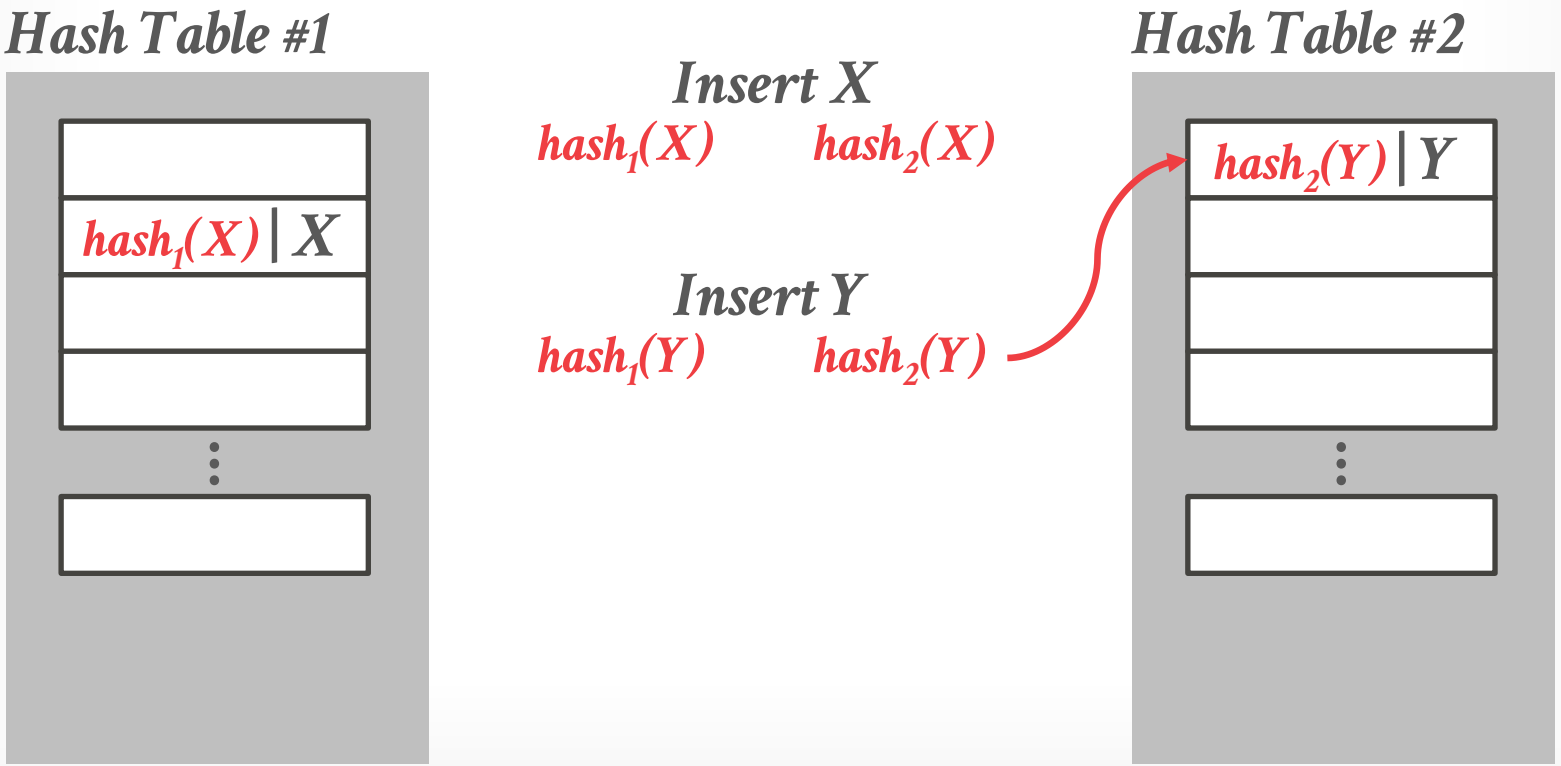
# CUCKOO HASHING



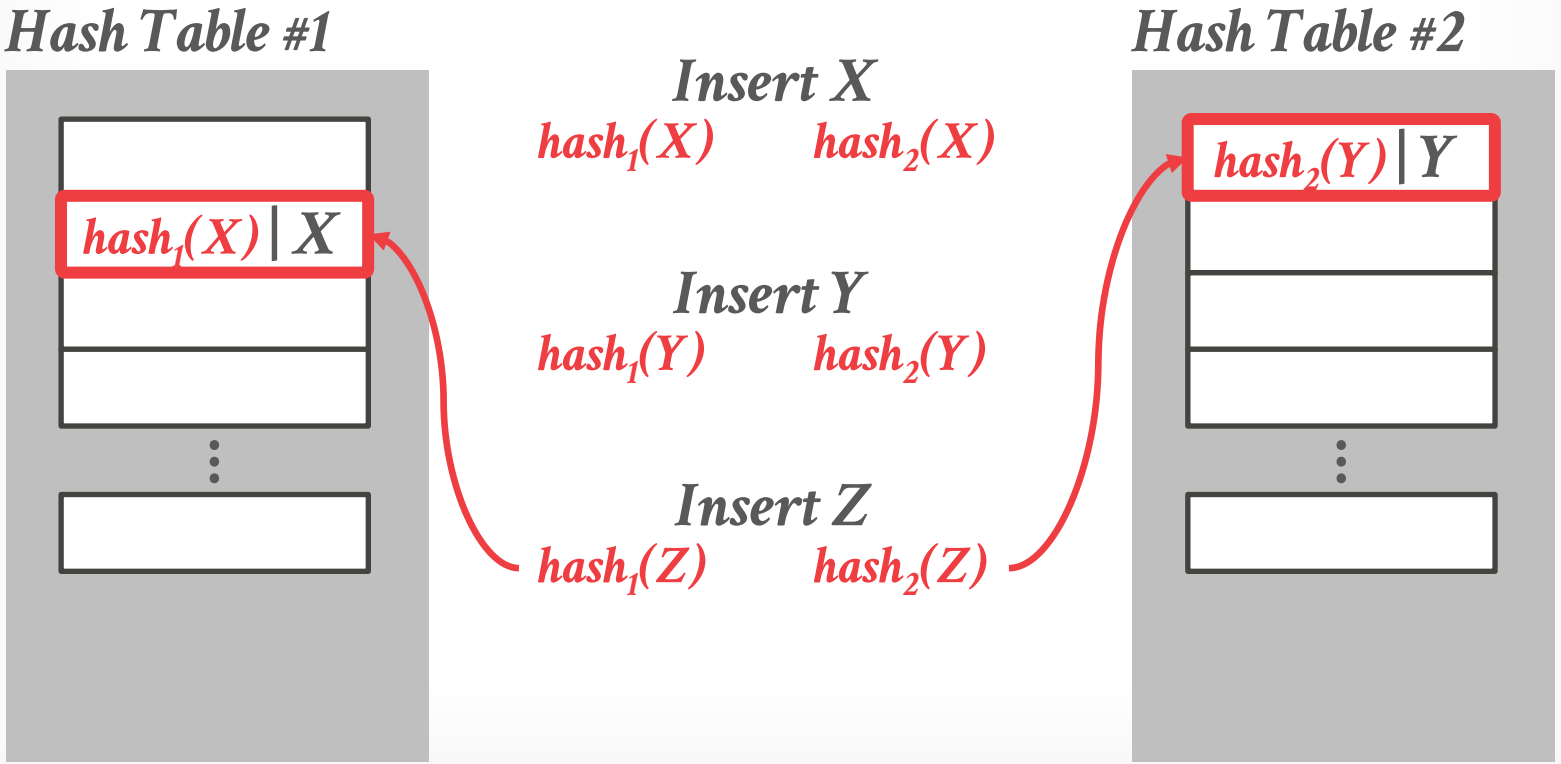
# CUCKOO HASHING



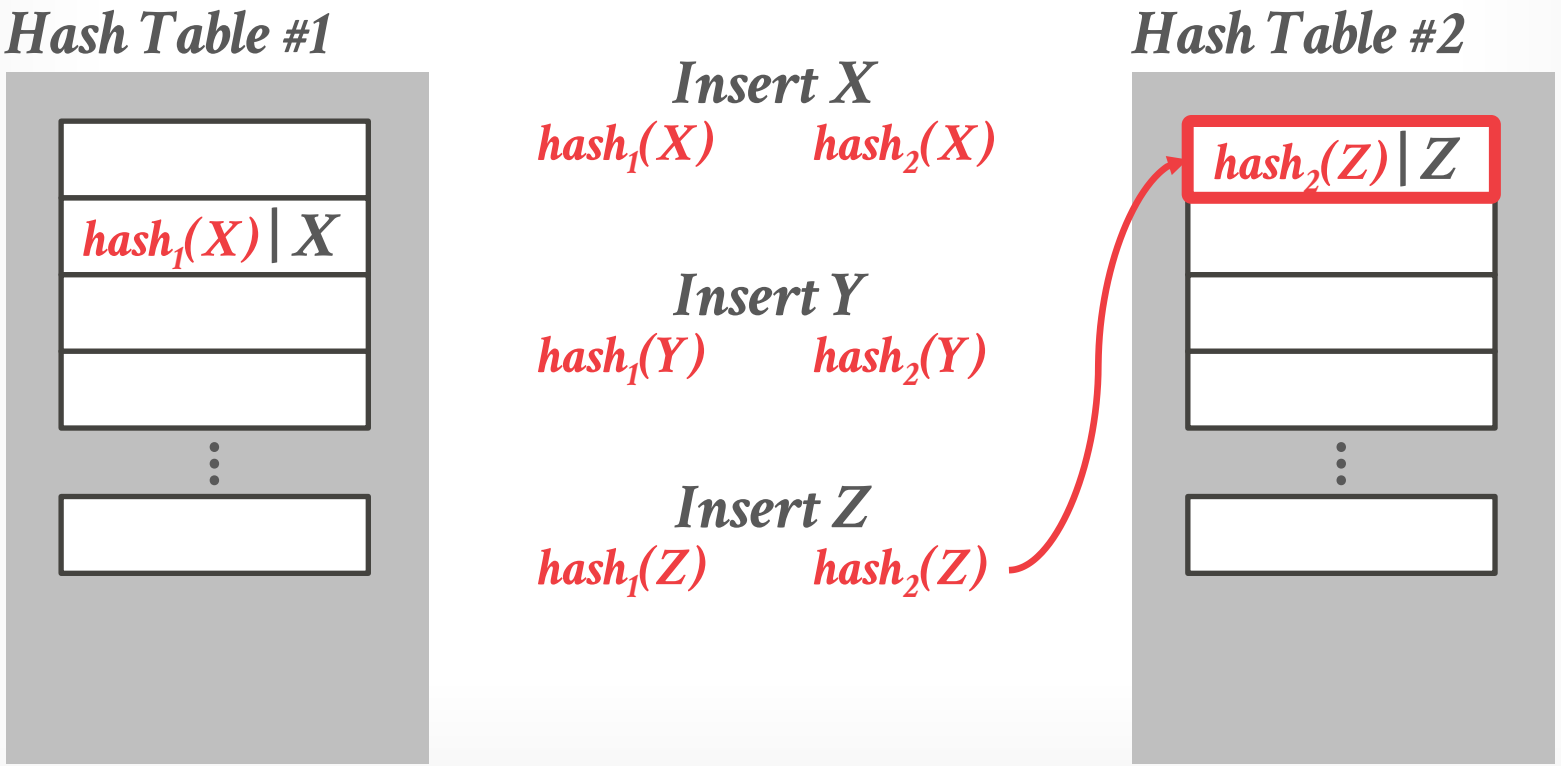
# CUCKOO HASHING



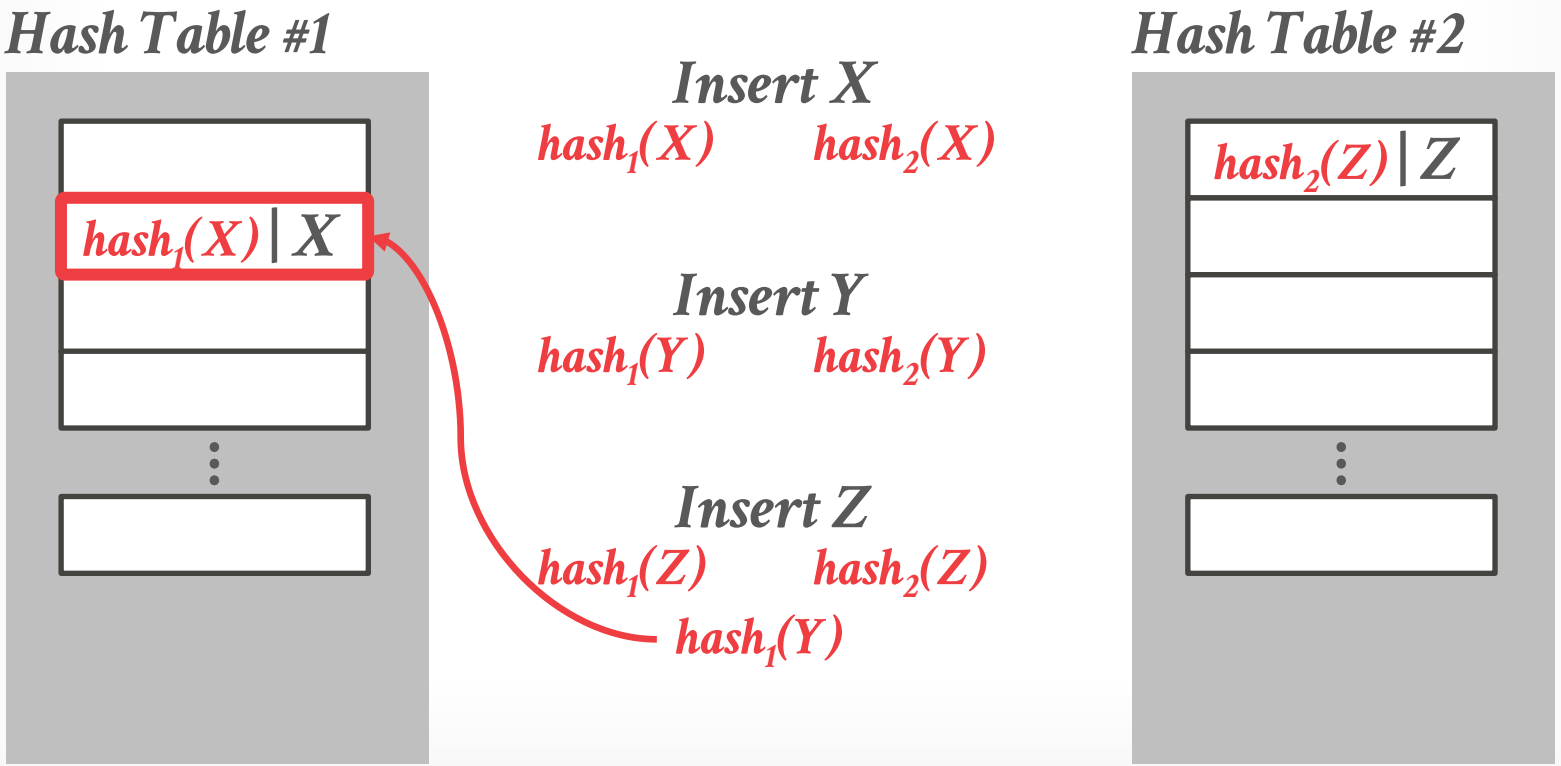
# CUCKOO HASHING



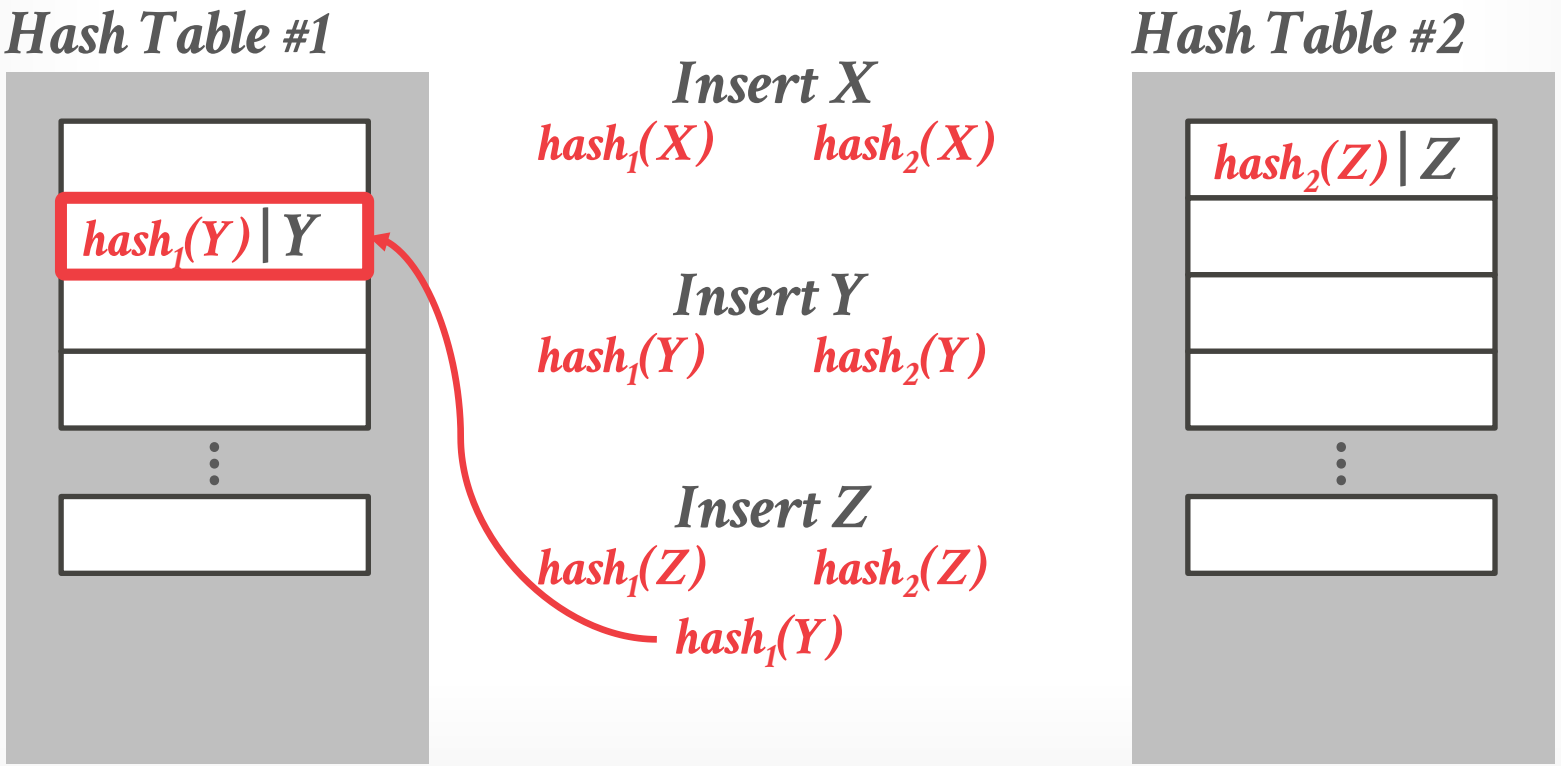
# CUCKOO HASHING



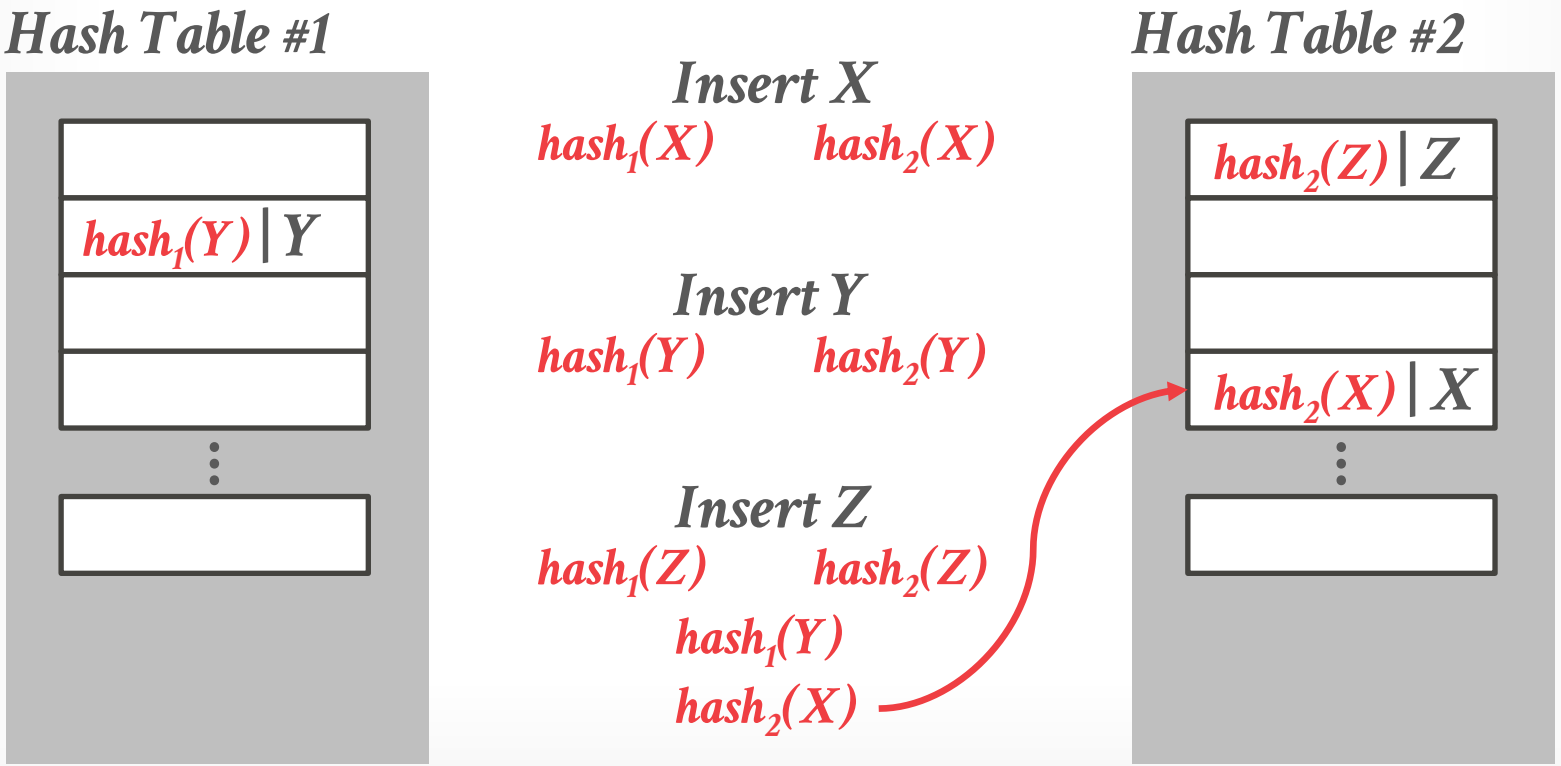
# CUCKOO HASHING



# CUCKOO HASHING



# CUCKOO HASHING



# PROBE PHASE

For each tuple in **S**, hash its join key and check to see whether there is a match for each tuple in corresponding bucket in the hash table constructed for **R**.

→ If inputs were partitioned, then assign each thread a unique partition.

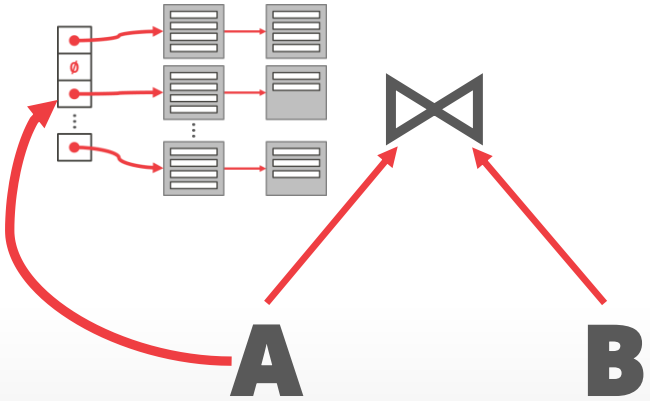
→ Otherwise, synchronize their access to the cursor on **S**.

# PROBE PHASE - BLOOM FILTER

Create a Bloom Filter during the build phase when the key is likely to not exist in the hash table.

→ Threads check the filter before probing the hash table. This will be faster since the filter will fit in CPU caches.

→ Sometimes called ***sideways information passing.***



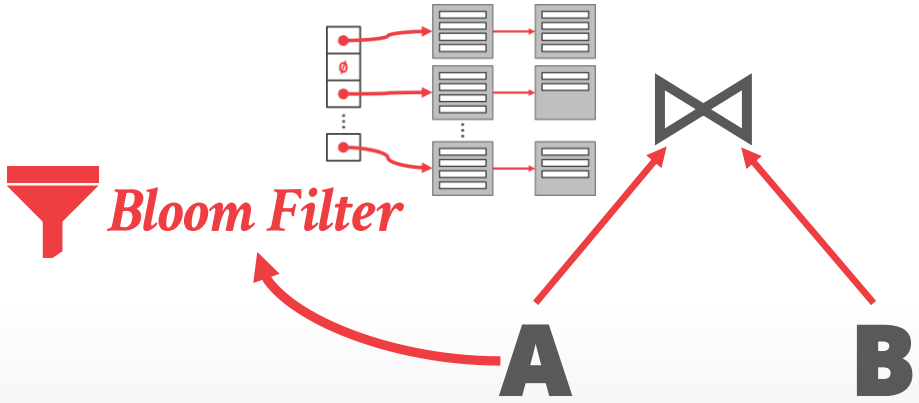

MICRO ADAPTIVITY IN VECTORWISE
SIGMOD 2013

# PROBE PHASE - BLOOM FILTER

Create a Bloom Filter during the build phase when the key is likely to not exist in the hash table.

→ Threads check the filter before probing the hash table. This will be faster since the filter will fit in CPU caches.

→ Sometimes called ***sideways information passing.***



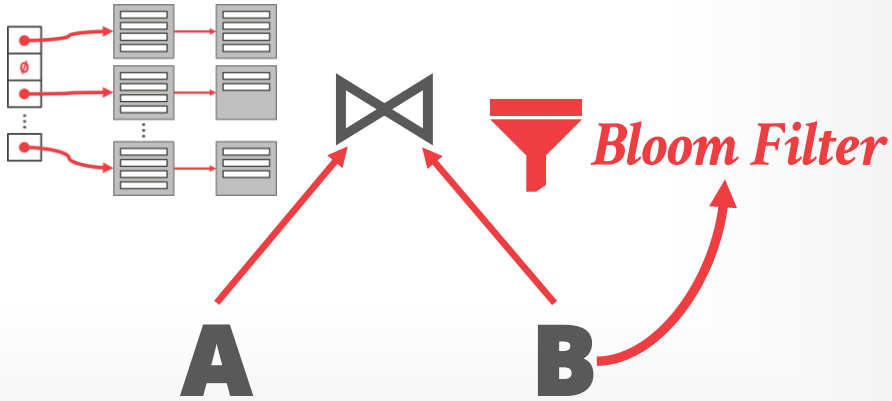

MICRO ADAPTIVITY IN VECTORWISE
SIGMOD 2013

# PROBE PHASE - BLOOM FILTER

Create a Bloom Filter during the build phase when the key is likely to not exist in the hash table.

→ Threads check the filter before probing the hash table. This will be faster since the filter will fit in CPU caches.

→ Sometimes called ***sideways information passing.***



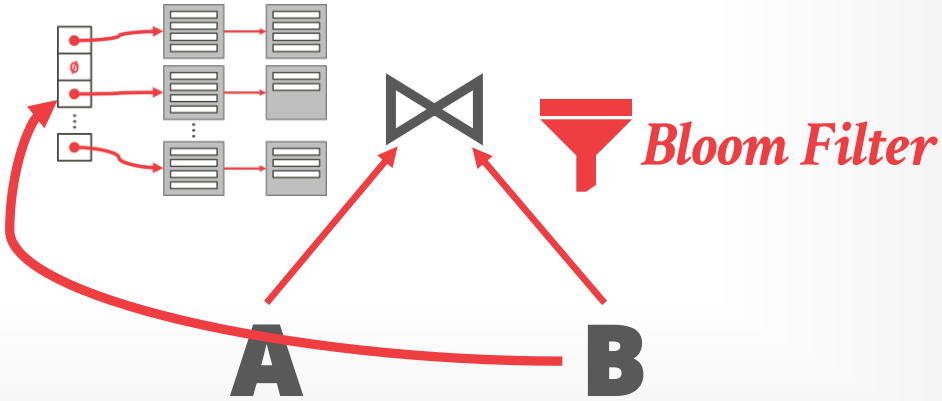

MICRO ADAPTIVITY IN VECTORWISE
SIGMOD 2013

# PROBE PHASE - BLOOM FILTER

Create a Bloom Filter during the build phase when the key is likely to not exist in the hash table.

→ Threads check the filter before probing the hash table. This will be faster since the filter will fit in CPU caches.

→ Sometimes called ***sideways information passing.***




MICRO ADAPTIVITY IN VECTORWISE
SIGMOD 2013

# HASH JOIN VARIANTS

| | *No-P* | *Shared-P* | *Private-P* | *Radix* |
|---|---|---|---|---|
| Partitioning | No | Yes | Yes | Yes |
| Input scans | 0 | 1 | 1 | 2 |
| Sync during partitioning | – | Spinlock per tuple | Barrier, once at end | Barrier, 4 · #passes |
| Hash table | Shared | Private | Private | Private |
| Sync during build phase | Yes | No | No | No |
| Sync during probe phase | No | No | No | No |

# BENCHMARKS

Implemented multiple variants of hash join algorithms based on previous literature and compare unoptimized vs. optimized versions.

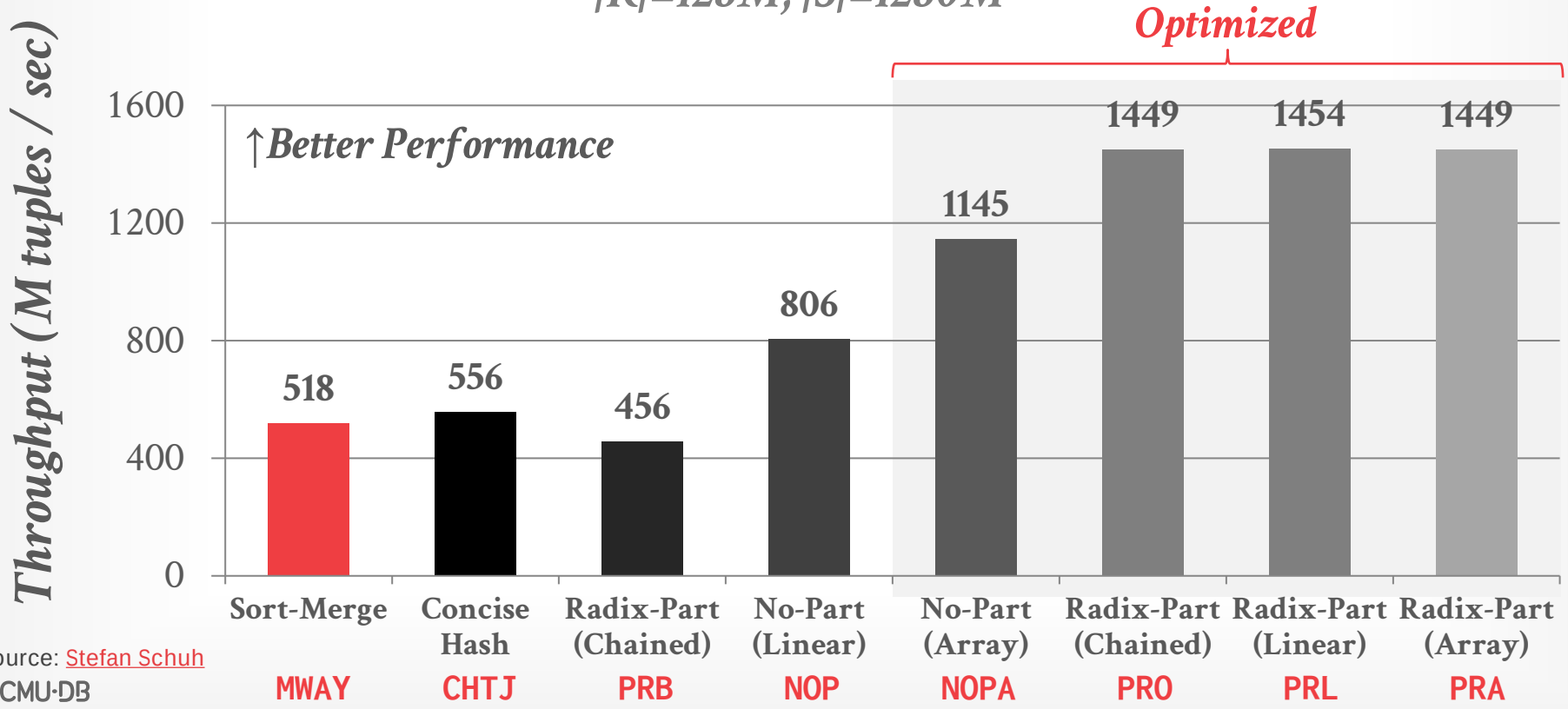
Core approaches:
→ No Partitioning Hash Join
→ Concise Hash Table Join
→ 2-pass Radix Hash Join (Chained vs. Linear)

Special Case: Arrays for monotonic primary keys.

AN EXPERIMENTAL COMPARISON OF THIRTEEN RELATIONAL EQUI-JOINS IN MAIN MEMORY
SIGMOD 2016

# JOIN COMPARISON (R⋈S)

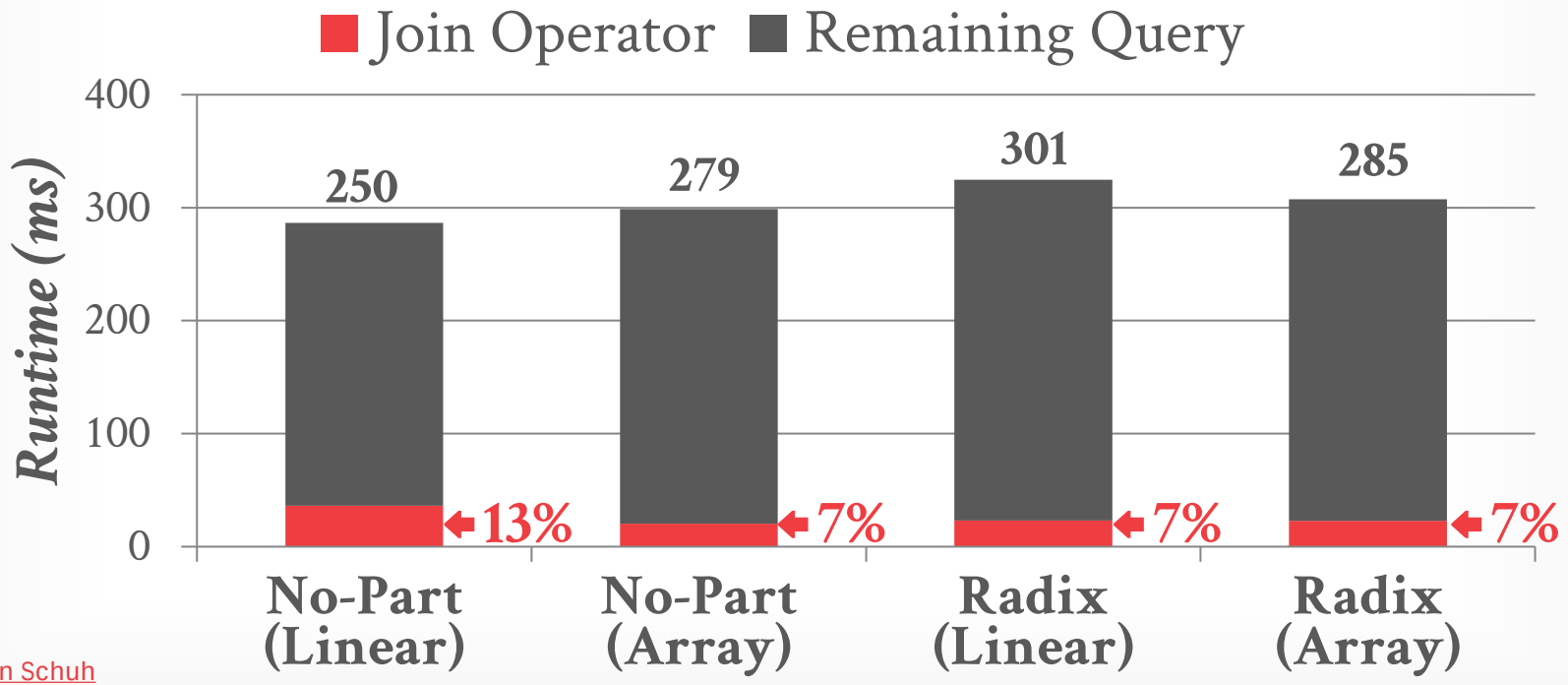
*4× Intel Xeon CPU E7-4870v2 (Only 32 cores)*
*|R|=128M, |S|=1280M*




Source: Stefan Schuh

# TPC-H Q19

*4× Intel Xeon CPU E7-4870v4*
*Scale Factor 100*




Source: Stefan Schuh

# PARTING THOUGHTS

Partitioned-based joins outperform no-partitioning algorithms in most settings, but it is non-trivial to tune it correctly.

AFAIK, every DBMS vendor picks one hash join implementation and does not try to be adaptive.

# NEXT CLASS

Parallel Sort-Merge Joins