ADVANCED DATABASE SYSTEMS

Parallel Hash Join Algorithms

09

Andy Pavlo
CMU 15-721
Spring 2024

Carnegie Mellon University

LAST CLASS

How to break up input data into smaller units (morsels) and allow workers to pull them from a global queue.

Distributed query scheduling is roughly the same as multi-core single node query scheduling.

DYNAMIC SCALING VS. WORK STEALING

There are two design decisions on how to handle queries that take longer to complete than expected.

Approach #1: Dynamic Scaling

→ Provision additional workers before a query starts.
→ Example: Snowflake Flexible Compute

Approach #2: Work Stealing

→ A worker takes tasks from a peer.
→ Example: Snowflake Scheduler

TODAY'S AGENDA

Background
Parallel Hash Join
Hash Functions
Hashing Schemes
Evaluation

PARALLEL JOIN ALGORITHMS

Perform a join between <u>two</u> relations on multiple threads simultaneously to speed up operation.

→ We will discuss multi-way joins next class.

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…

HASHING VS. SORTING JOINS

1970s – Sorting

1980s – Hashing

1990s – Equivalent

2000s – Hashing

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

2020s – Non-Partitioned Hashing

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

→ 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

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

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

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

MASSIVELY PARALLEL NUMA-AWARE HASH JOINS
IMDM 2013

→ 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

→ New optimizations and results for Radix Hash Join.

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

→ 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

→ Benefits of Radix Hash Join aren't worth dev costs. Hard to know when to use it.

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 due to cache misses.

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

PARTITION PHASE

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

→ Divide the inner/outer relations and redistribute among the CPU cores.

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

Explicitly partitioning the input relations before a join operator is sometimes called **Grace Hash Join**.

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**.

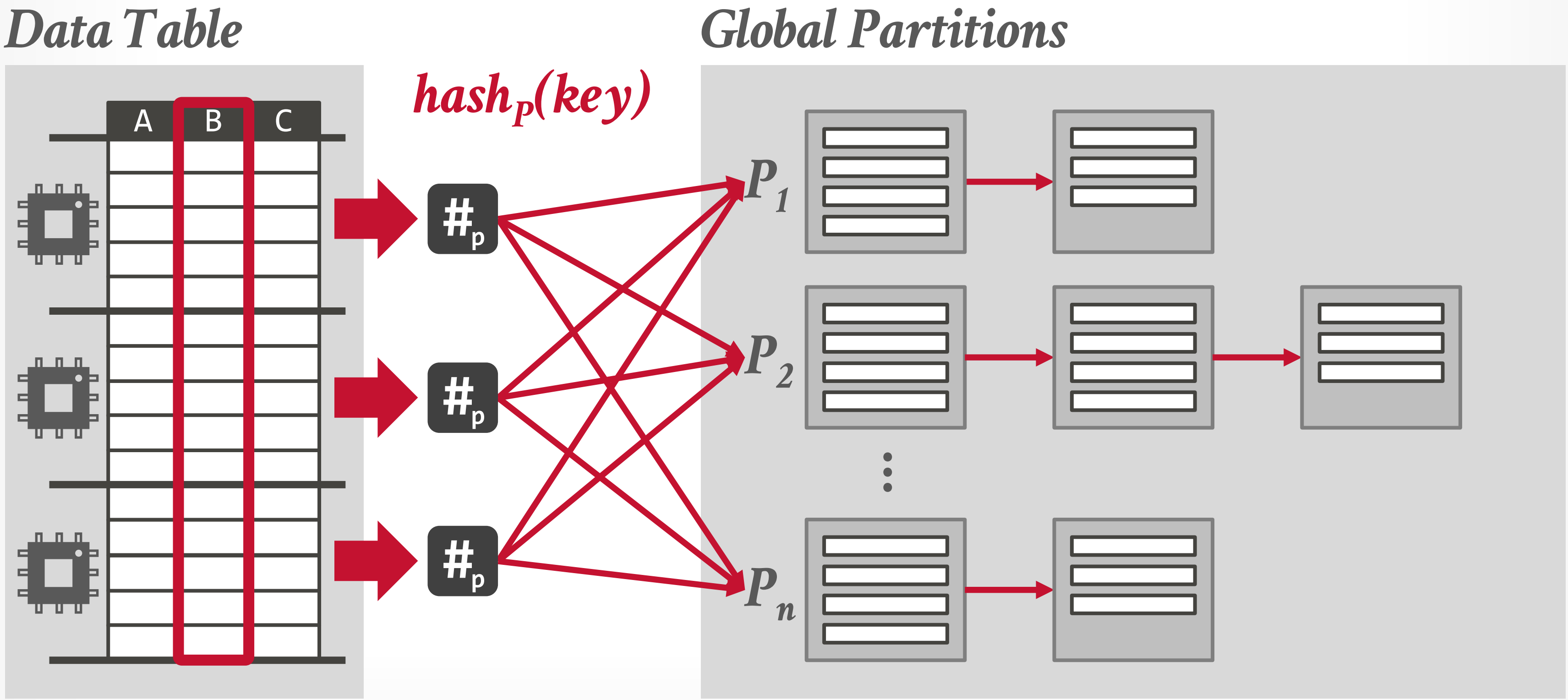
NON-BLOCKING PARTITIONING

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

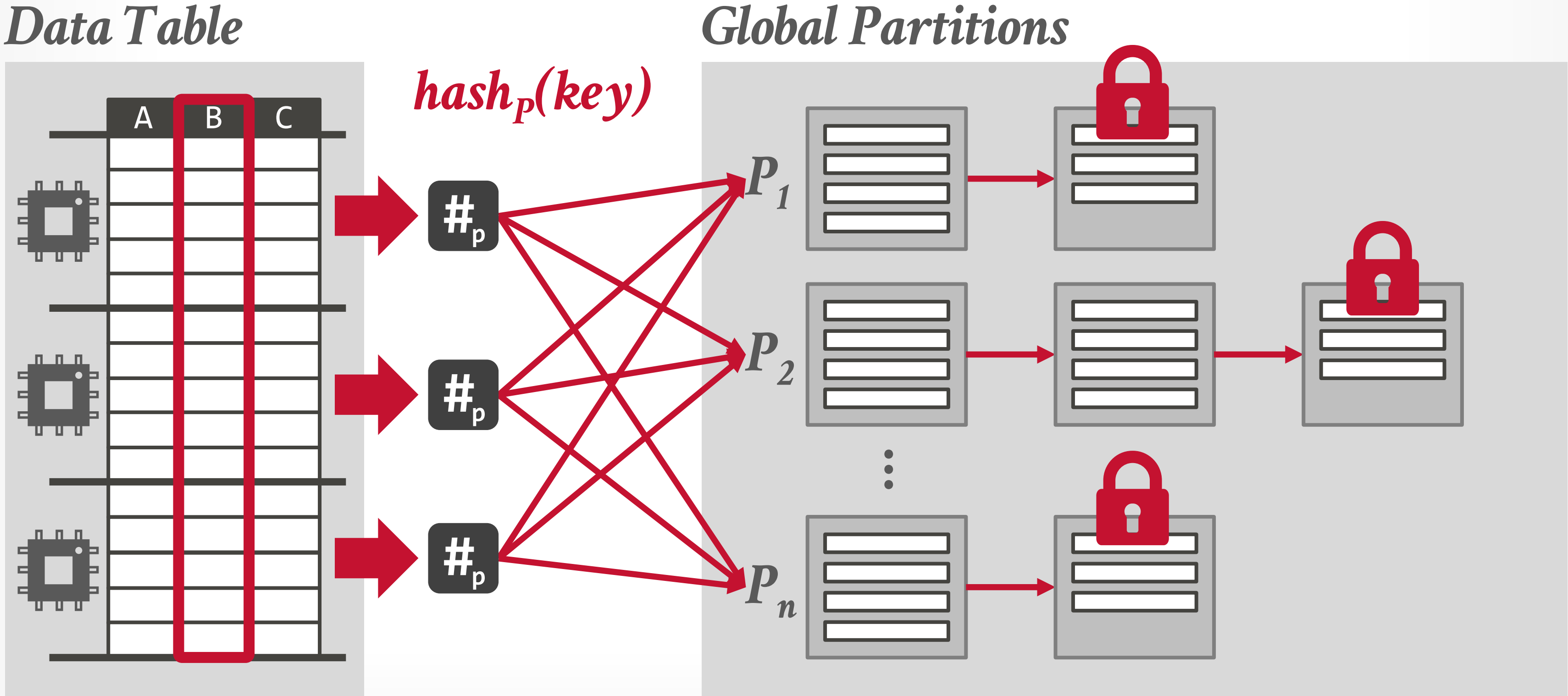
Approach #1: Shared Partitions
- → Single global set of partitions that all threads update.
- → Must use a latch to synchronize threads.

Approach #2: Private Partitions
- → Each thread has its own set of partitions.
- → Must consolidate them after all threads finish.

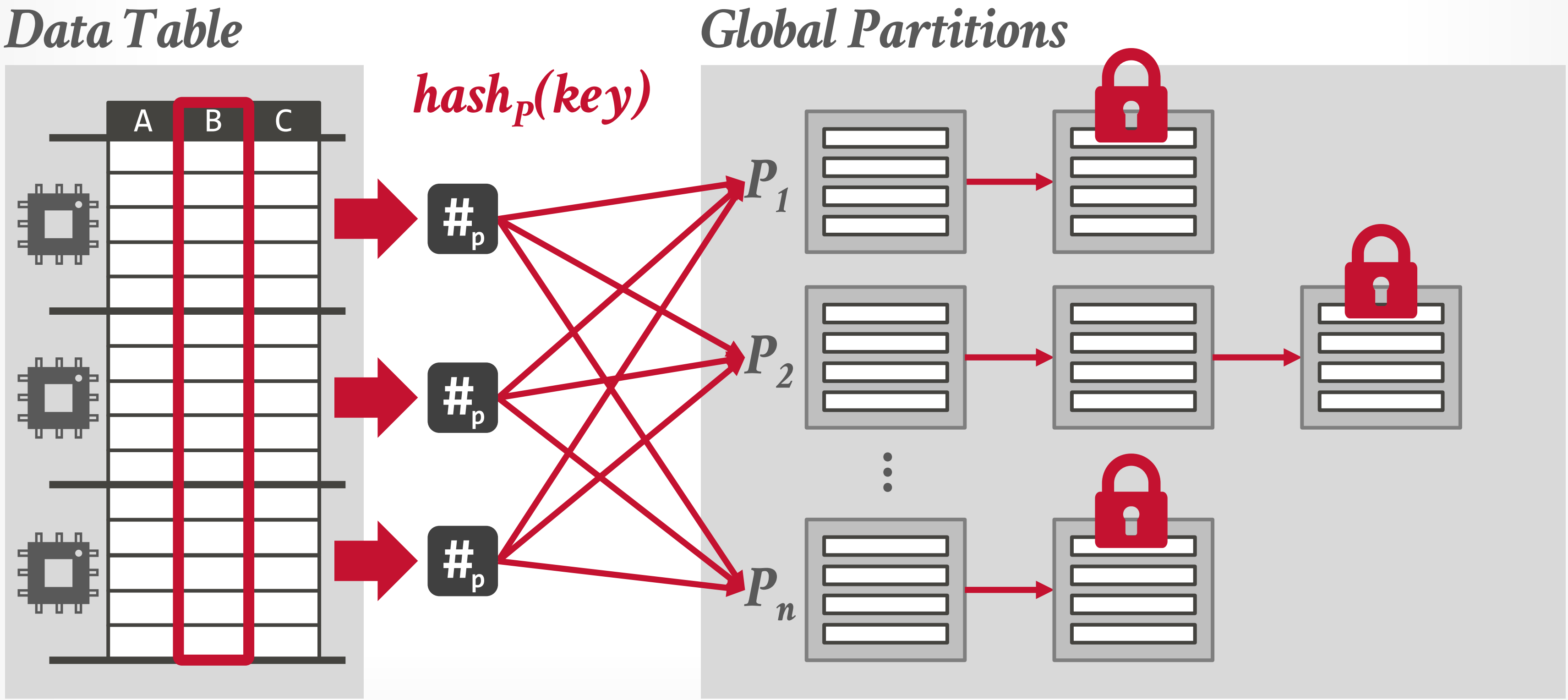
NON-BLOCKING: SHARED PARTITIONS



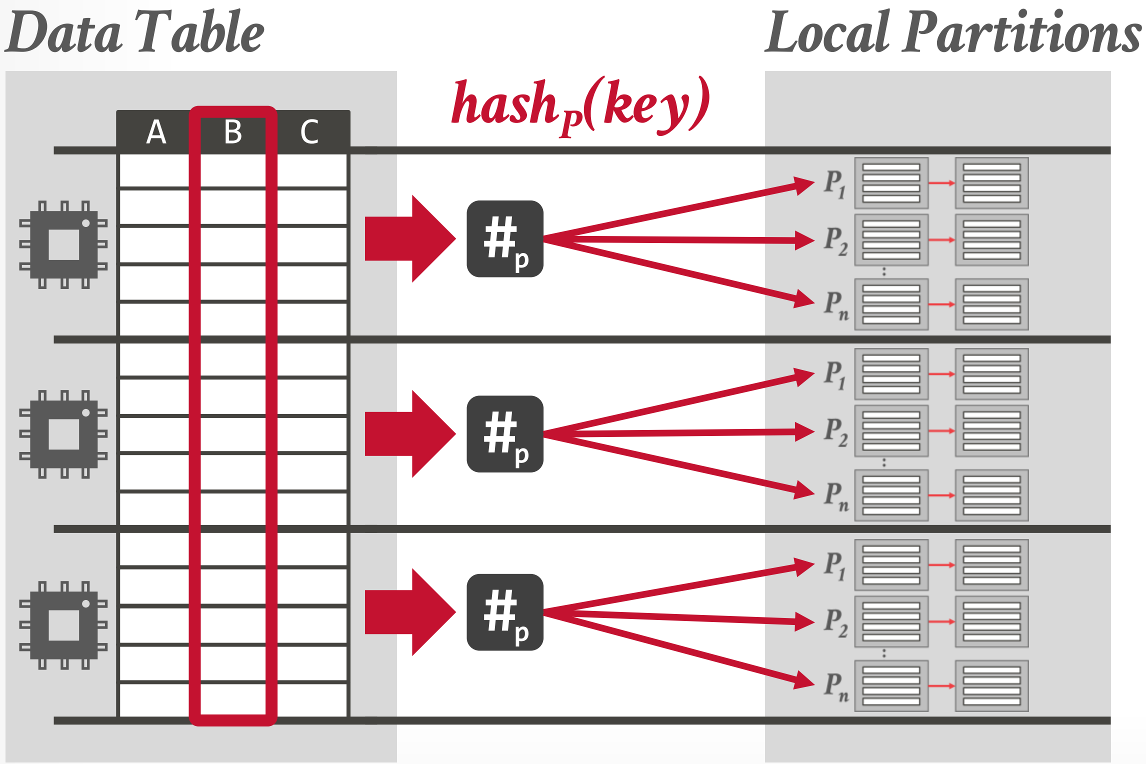
NON-BLOCKING: SHARED PARTITIONS



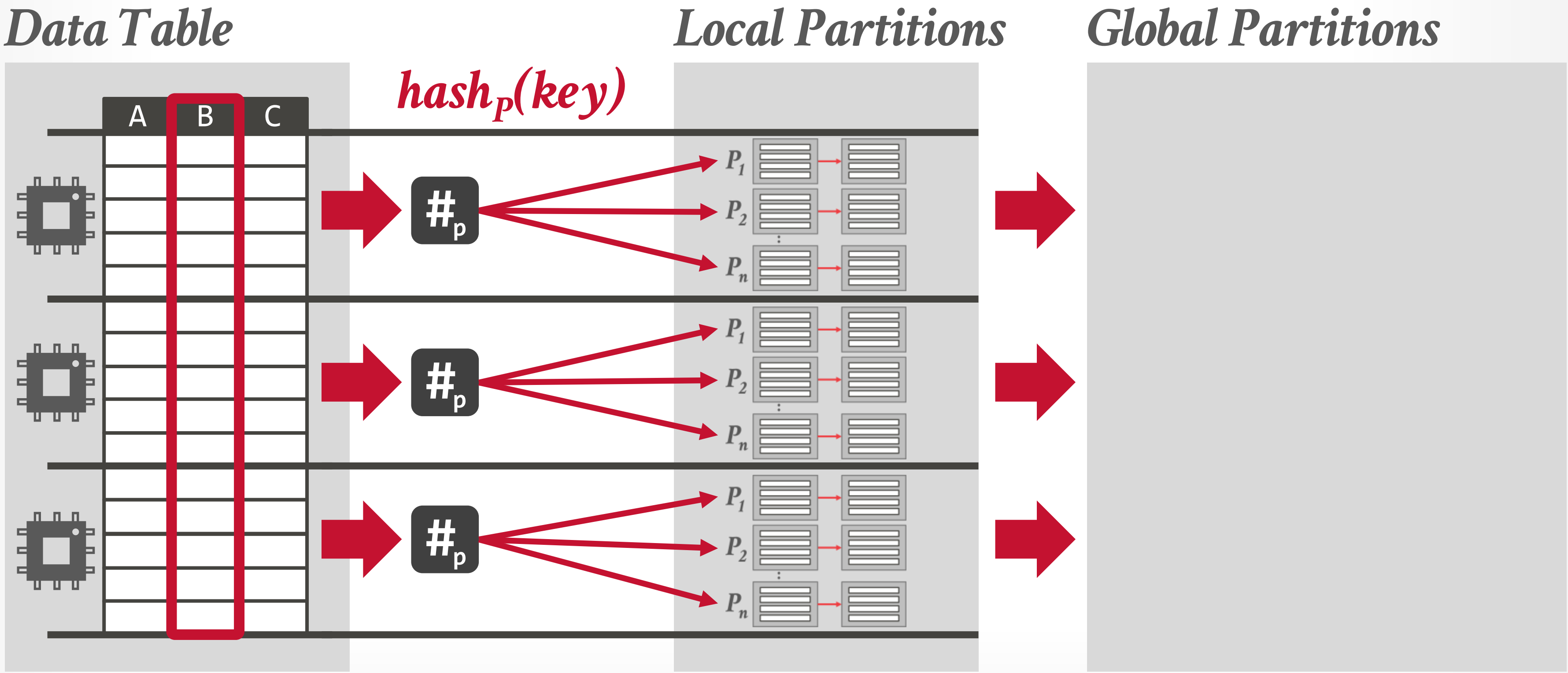
NON-BLOCKING: SHARED PARTITIONS



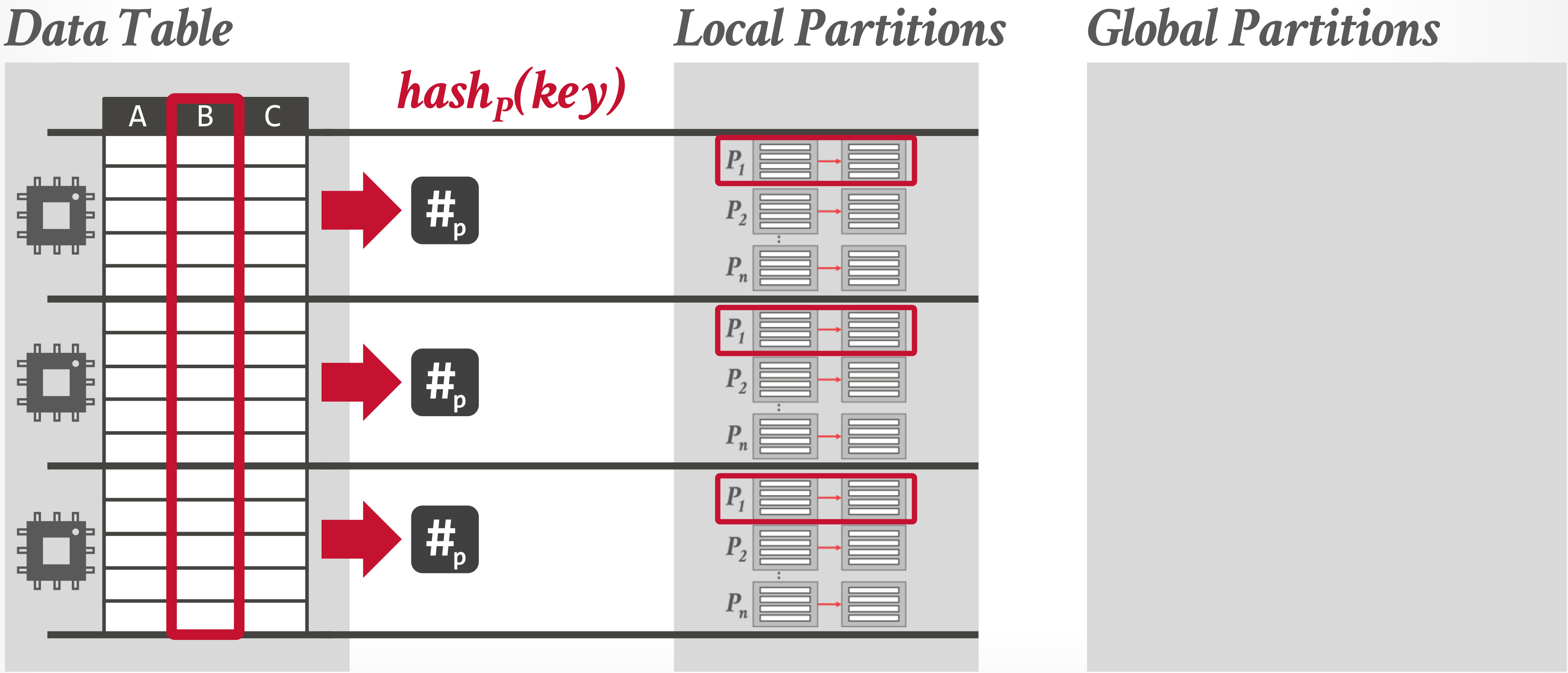
NON-BLOCKING: PRIVATE PARTITIONS



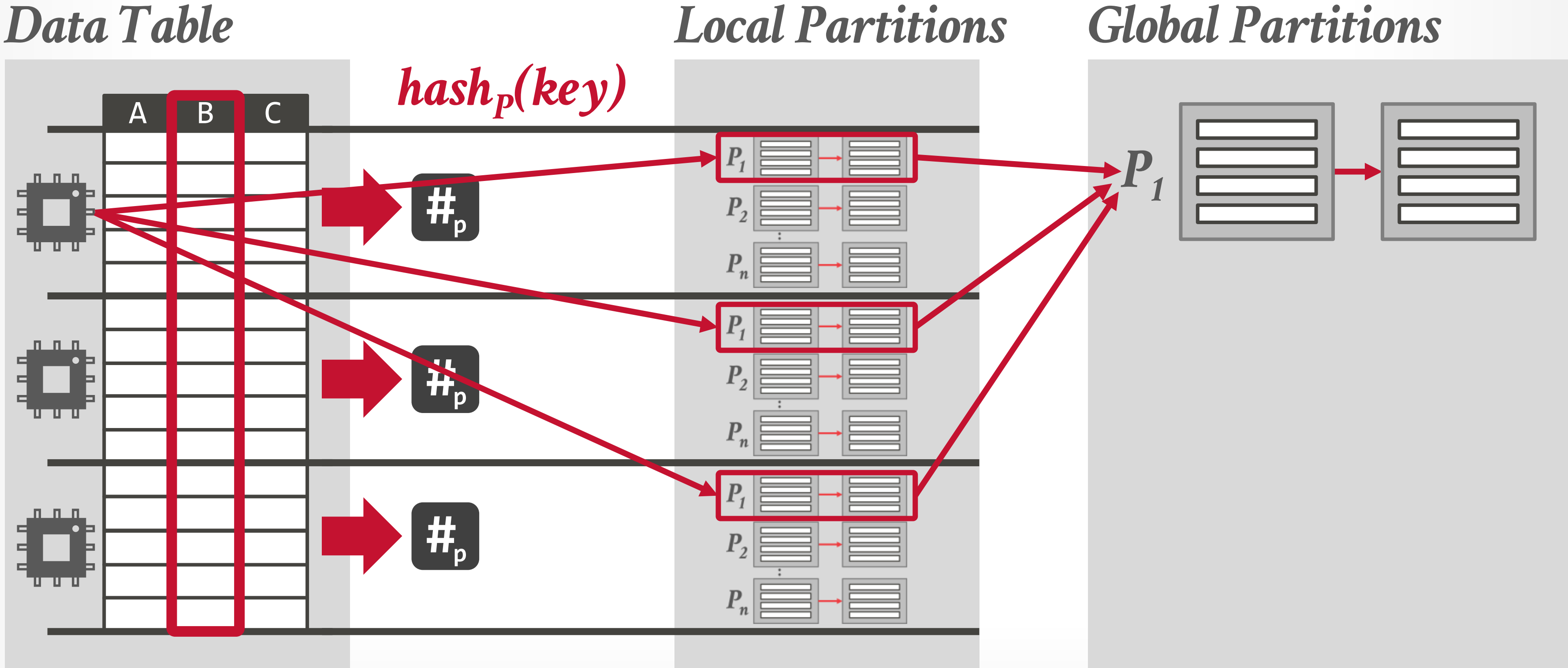
NON-BLOCKING: PRIVATE PARTITIONS



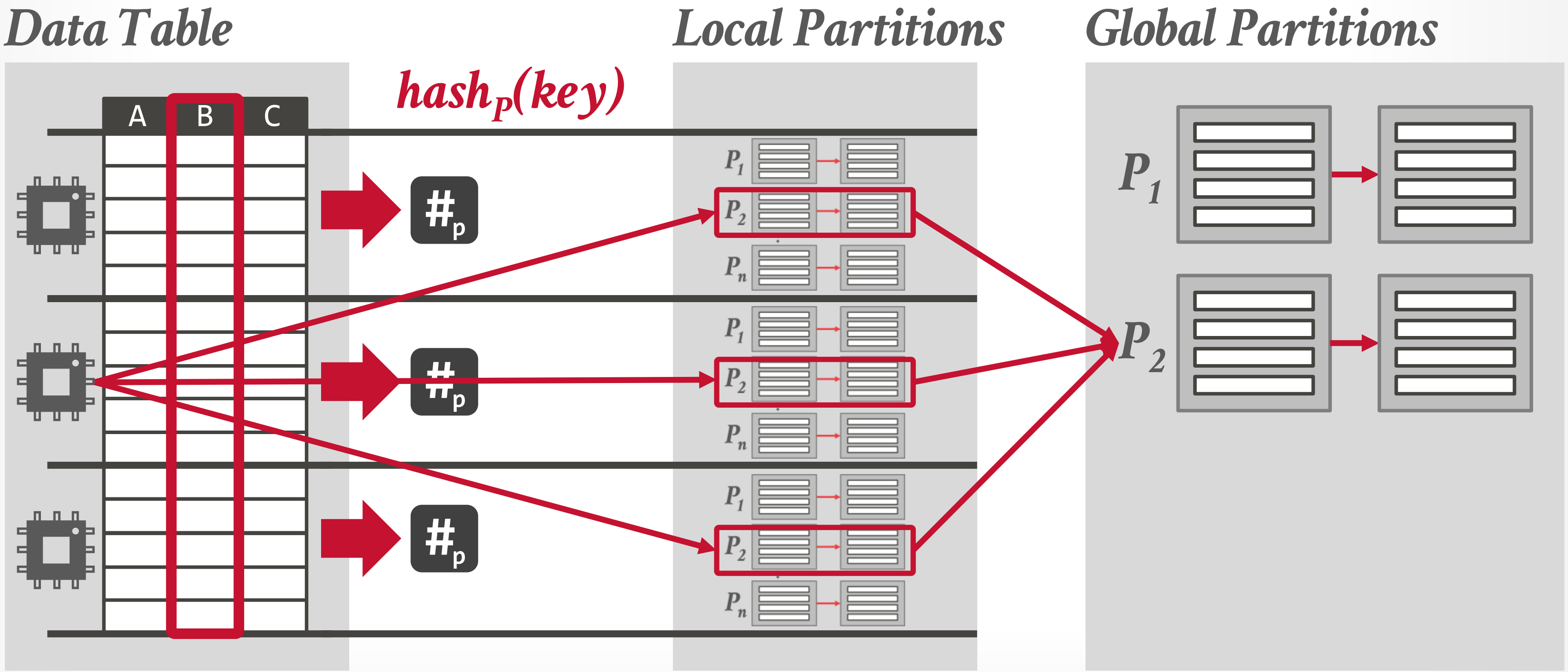
NON-BLOCKING: PRIVATE PARTITIONS



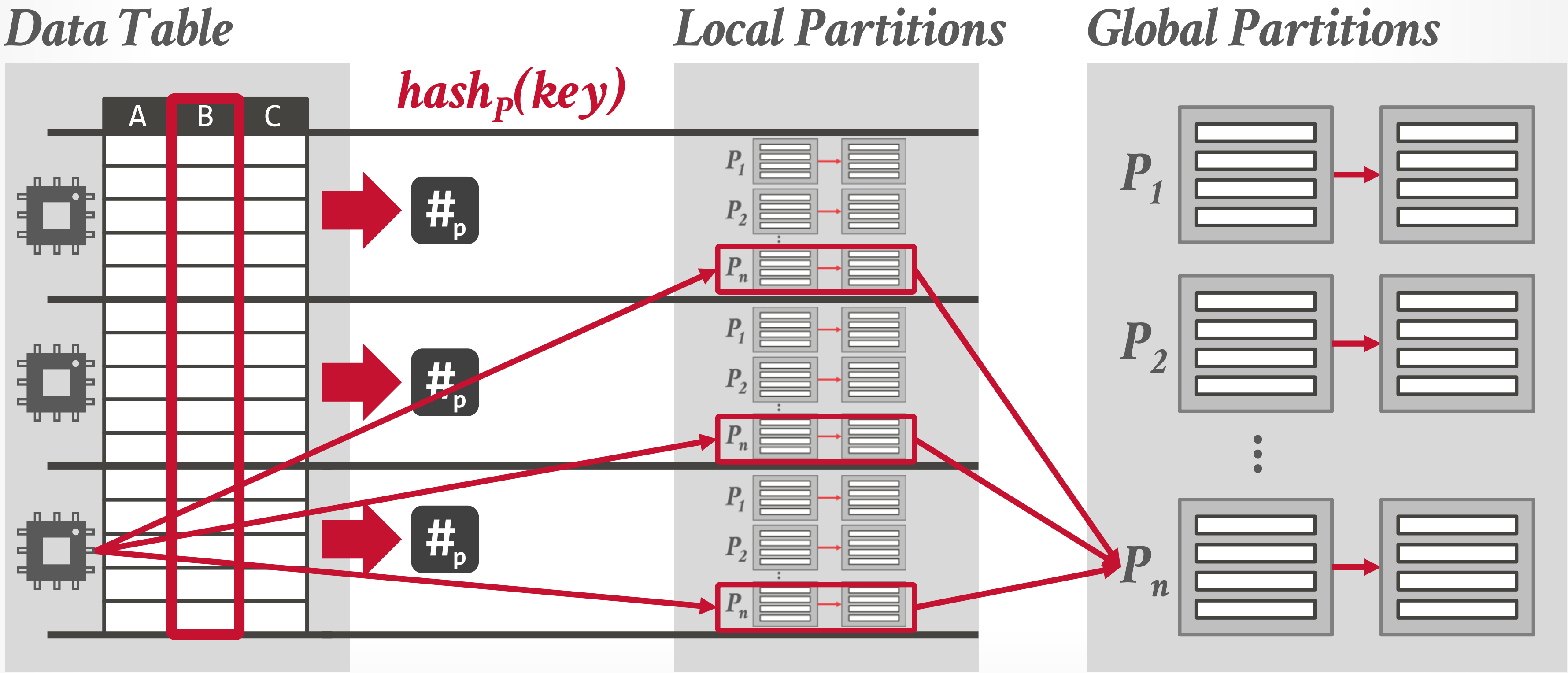
NON-BLOCKING: PRIVATE PARTITIONS



NON-BLOCKING: PRIVATE PARTITIONS



NON-BLOCKING: PRIVATE PARTITIONS



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**.

RADIX PARTITIONING

Scan the input relation multiple times to generate the partitions.

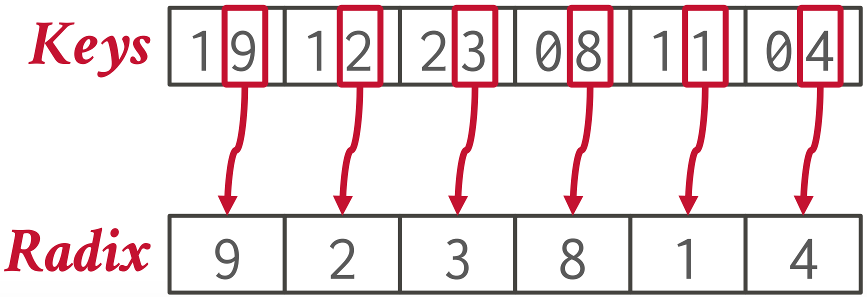
Two-pass algorithm:

→ **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).

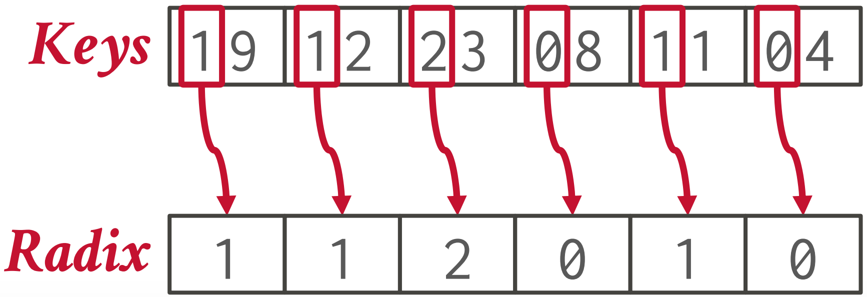
→ Efficient to compute with bitshifting + multiplication.



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.

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

Keys	19	12	23	08	11	04
Radix	1	1	2	0	1	0

Histogram (Key →Count)	0→2	1→3	2→1

PREFIX SUM

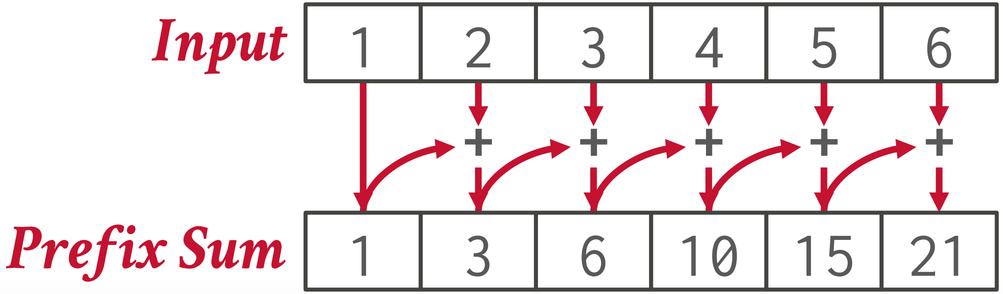
The prefix sum of a sequence of numbers

$(x_0, x_1, \ldots, x_n)$

is a second sequence of numbers

$(y_0, y1, \ldots, y_n)$

that is a running total of the input sequence.



PREFIX SU

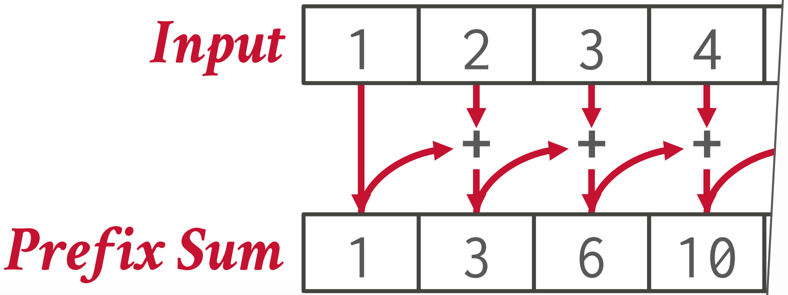
The prefix sum of a sequence of

$(x_0, x_1, \ldots, x_n)$

is a second sequence of numbers

$(y_0, y1, \ldots, y_n)$

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.

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.

Routine	t_e (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.

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.

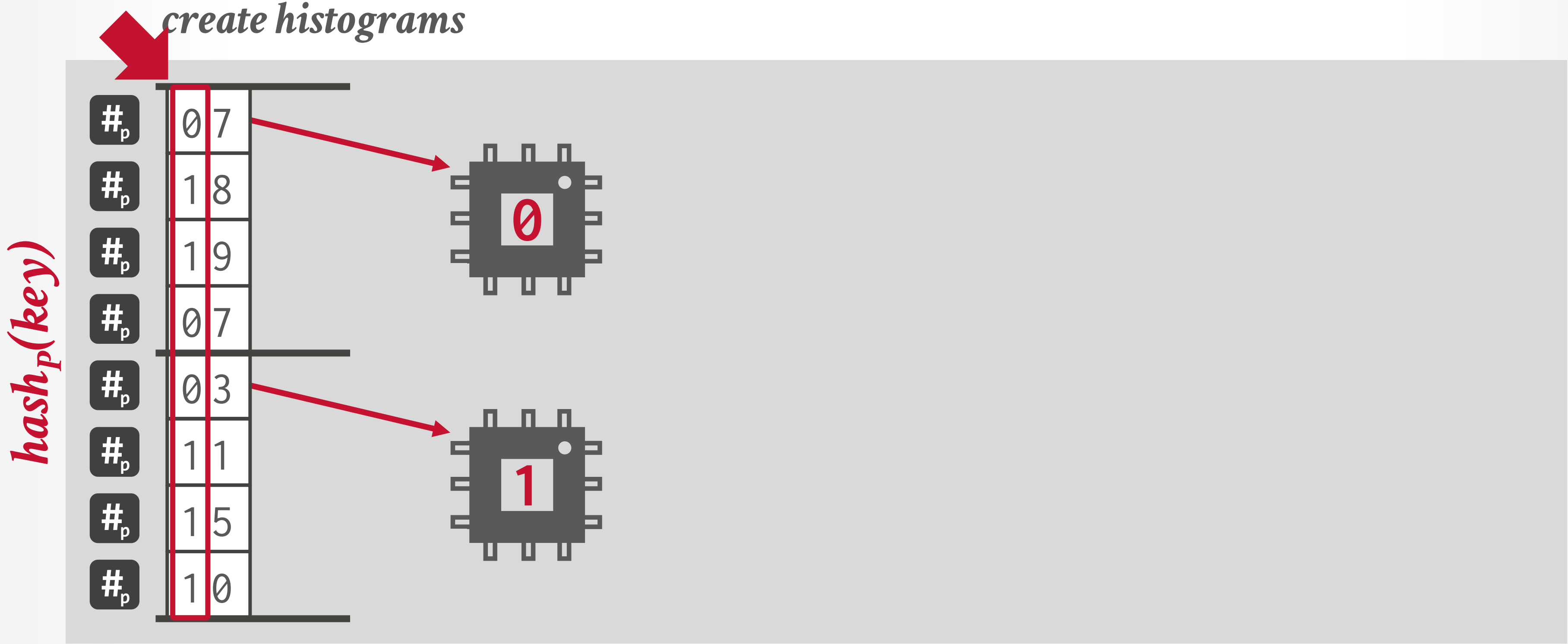
[1] CRAY Y-MP and CFT77 are trademarks of Cray Research, Inc.

[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

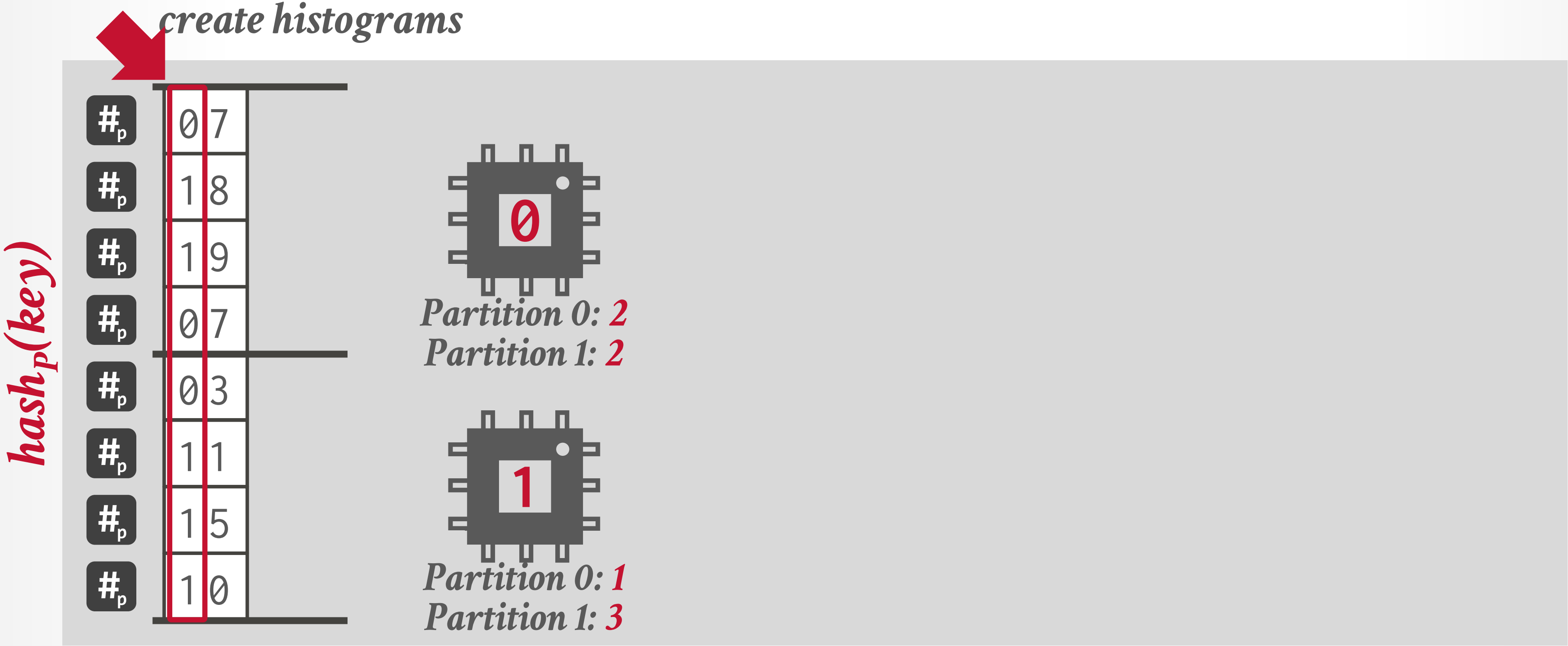
RADIX PARTITIONS

Step #1: Inspect input, create histograms



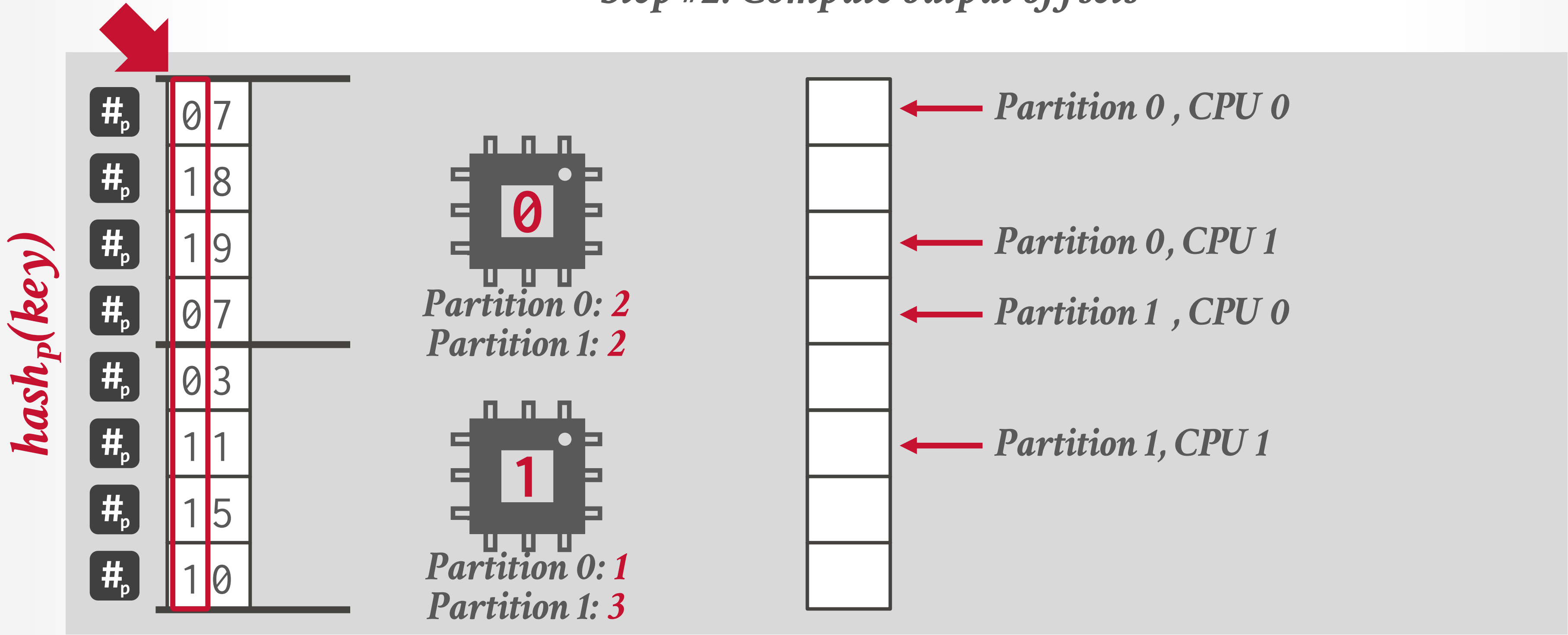
RADIX PARTITIONS

Step #1: Inspect input, create histograms



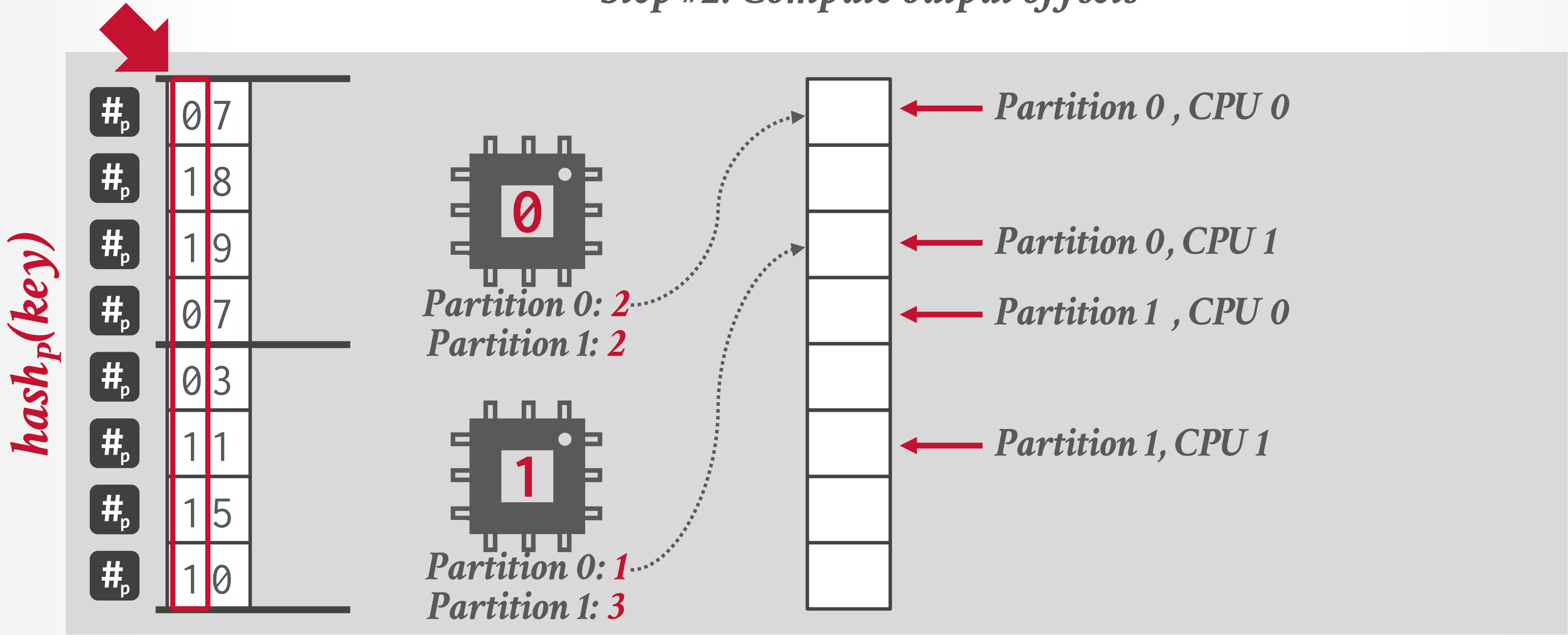
RADIX PARTITIONS

Step #2: Compute output offsets



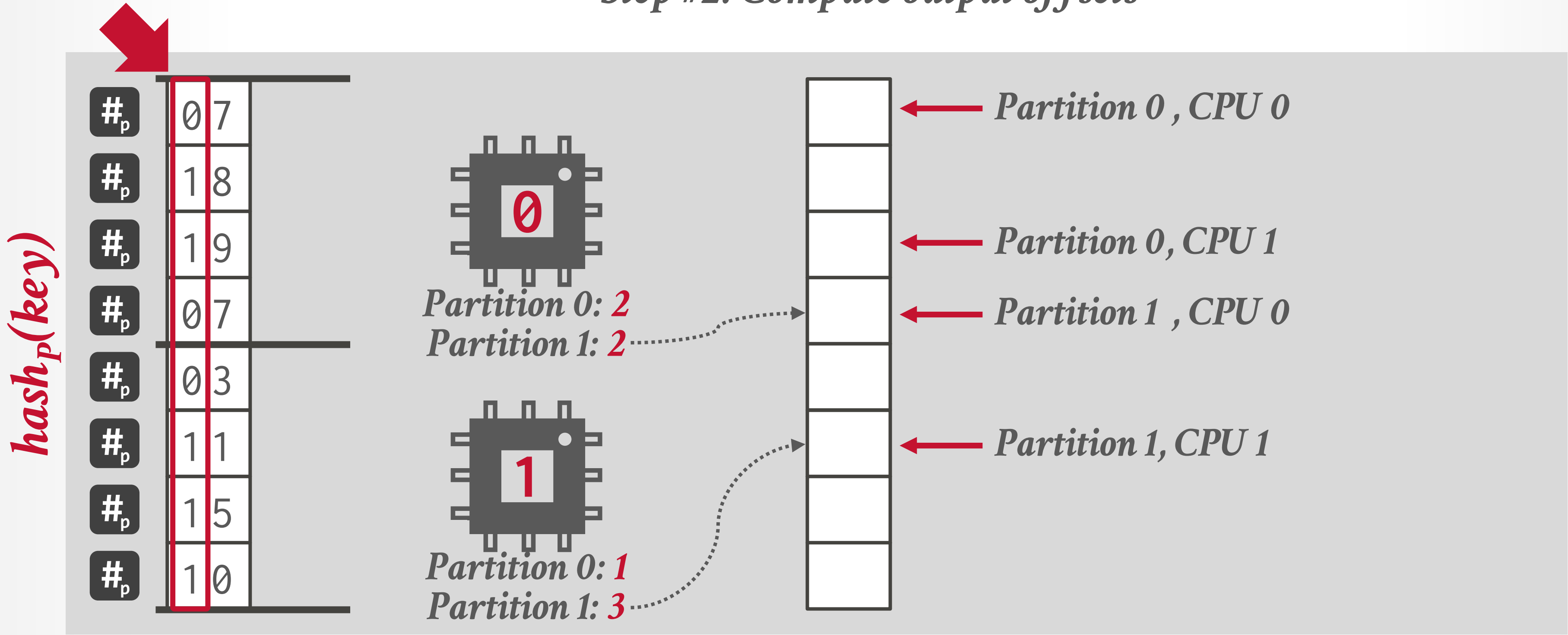
RADIX PARTITIONS

Step #2: Compute output offsets



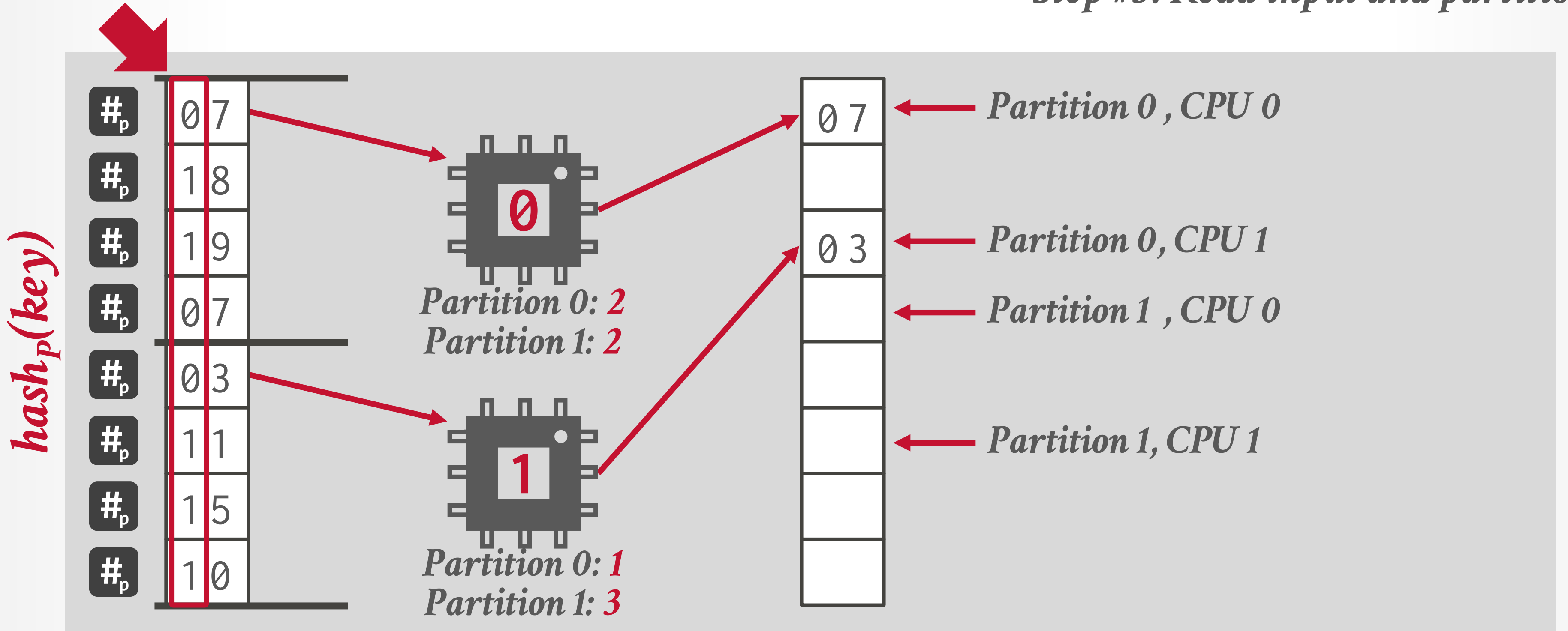
RADIX PARTITIONS

Step #2: Compute output offsets



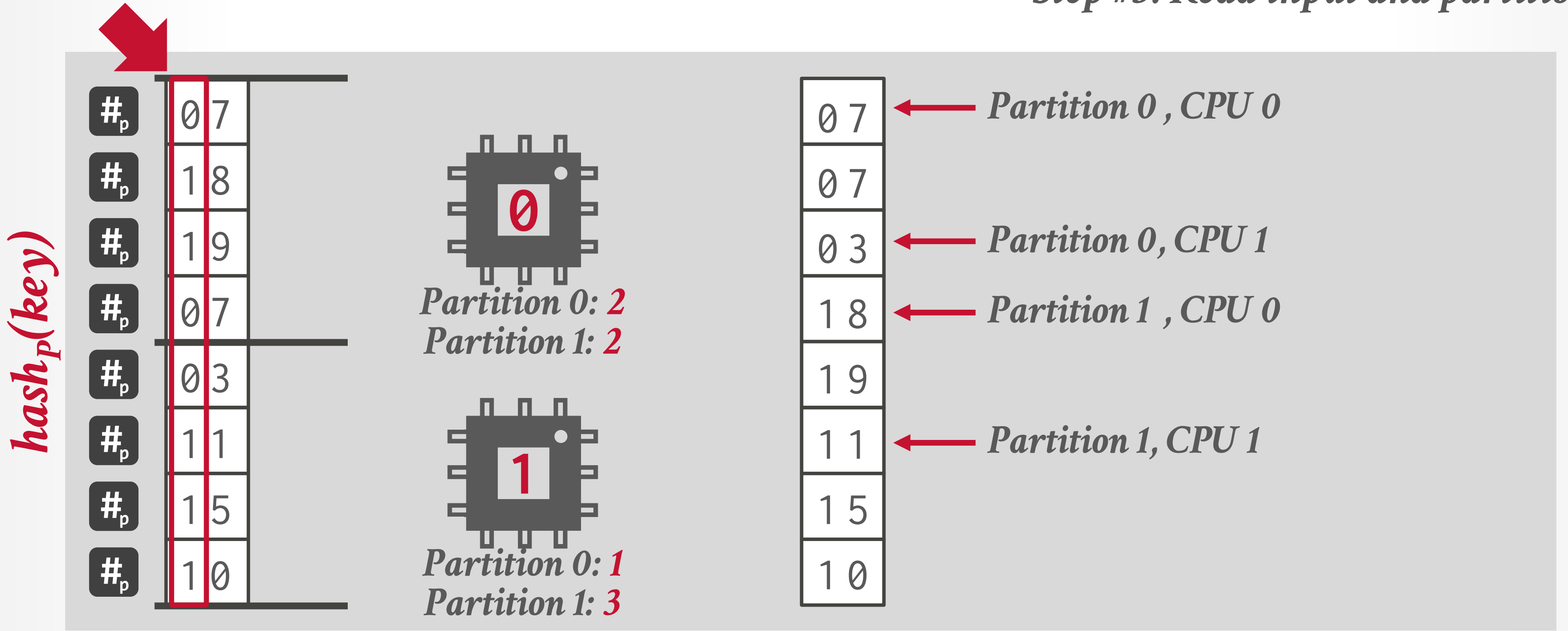
RADIX PARTITIONS

Step #3: Read input and partition



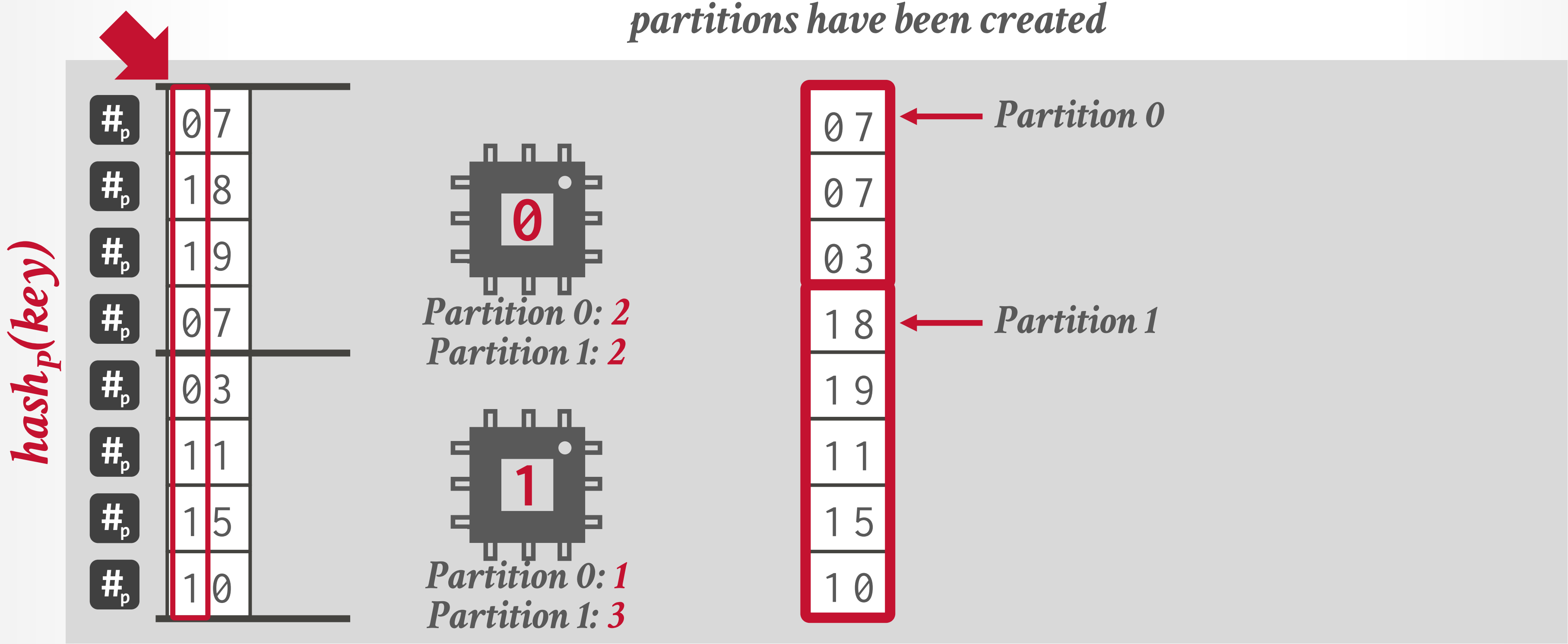
RADIX PARTITIONS

Step #3: Read input and partition



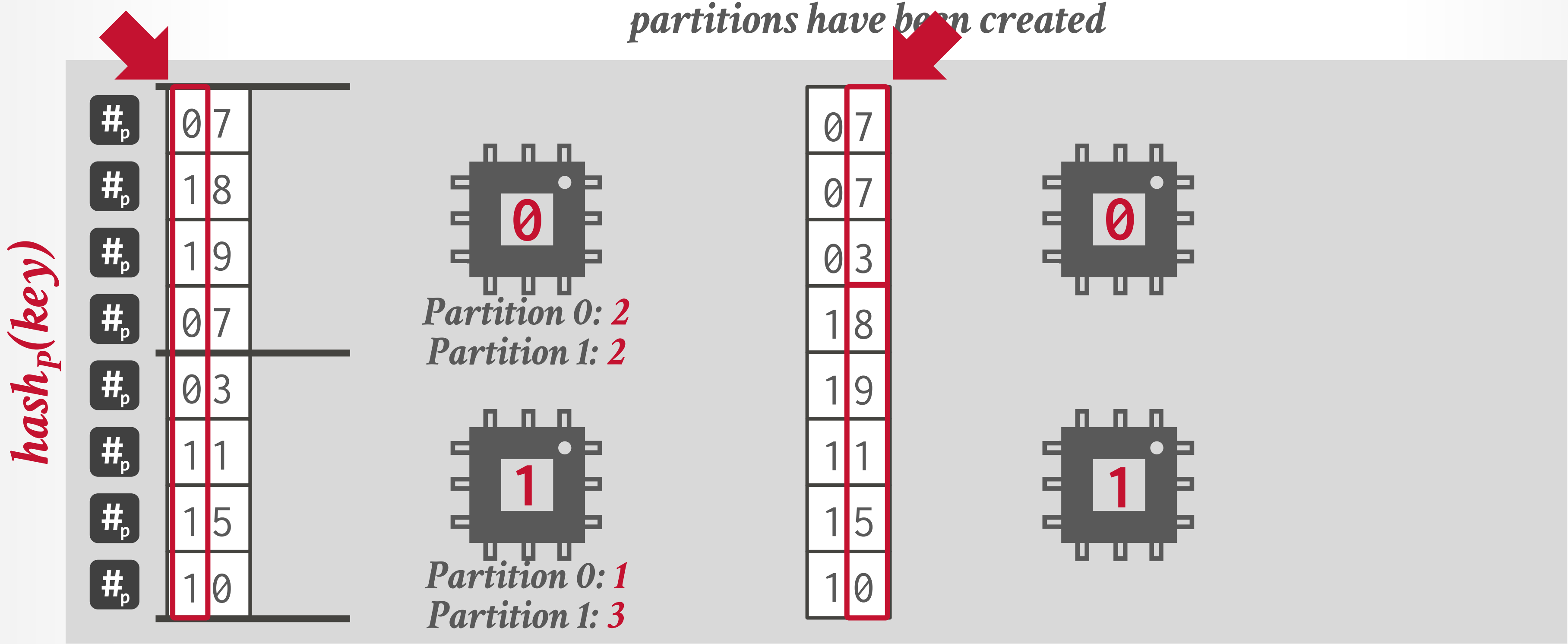
RADIX PARTITIONS

Recursively repeat until target number of partitions have been created



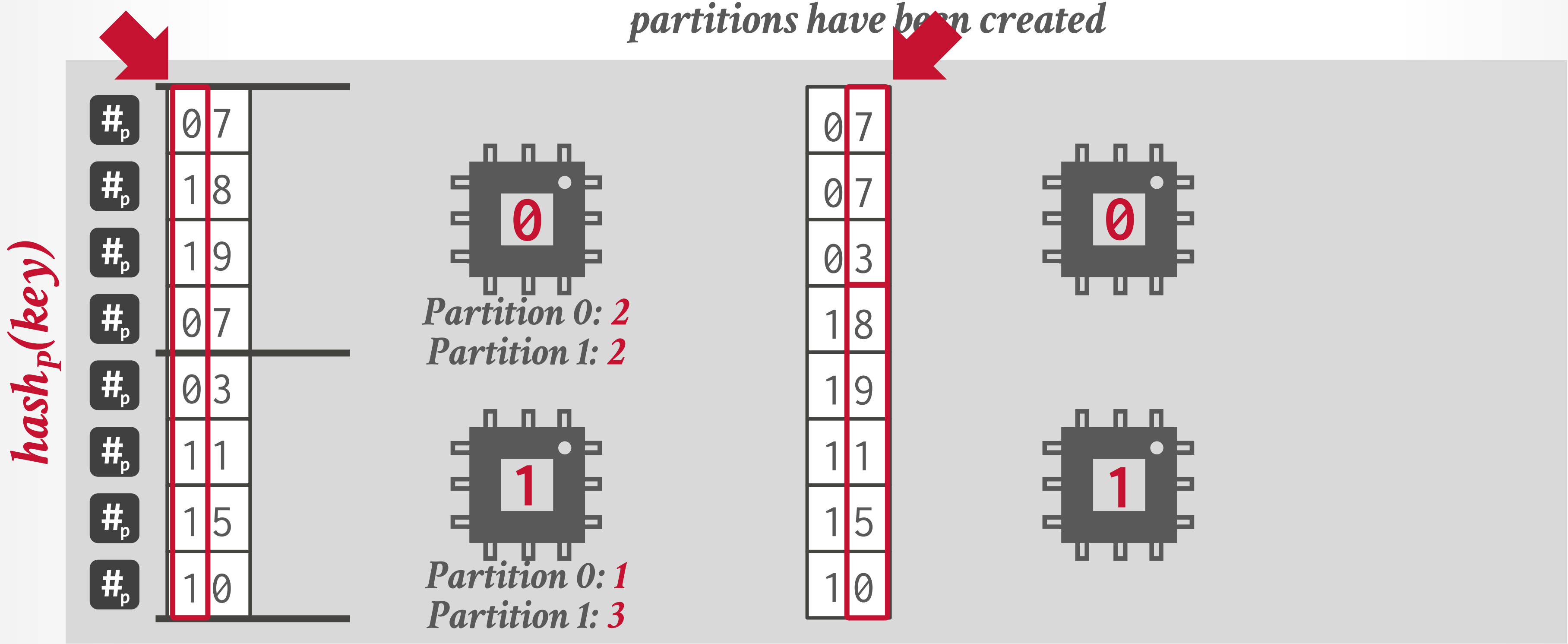
RADIX PARTITIONS

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(s) 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.

HASH FUNCTIONS

CRC-32/64 (1975)

→ Modern CPUs have explicit CRC instructions. Some DBMSs use this for hashing integers.

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 FUNCTIONS

s. Some

tion.

s).

on rates.

smhasher

SMhasher

Linux Build status build passing build failing

Hash function	MiB/sec	cycl./hash	cycl./map	size	Quality problems
donothing32	11149460.06	4.00	-	13	bad seed 0, test NOP
donothing64	11787676.42	4.00	-	13	bad seed 0, test NOP
donothing128	11745060.76	4.06	-	13	bad seed 0, test NOP
NOP_OAAT_read64	11372846.37	14.00	-	47	test NOP
BadHash	769.94	73.97	-	47	bad seed 0, test FAIL
sumhash	10699.57	29.53	-	363	bad seed 0, test FAIL
sumhash32	42877.79	23.12	-	863	UB, test FAIL
multiply_shift	8026.77	26.05	226.80 (8)	345	bad seeds & 0xfffffff0, fails most tests
pair_multiply_shift	3716.95	40.22	186.34 (3)	609	fails most tests
crc32	383.12	134.21	257.50 (11)	422	insecure, 8590x collisions, distrib, PerlinNoise
md5_32	350.53	644.31	894.12 (10)	4419	

HASH FUNCTIONS

smhasher

SMhasher

Linux Build status build passing build failing

Hash function	MiB/sec	cycl./hash	cycl./map	size	
donothing32	11149460.06	4.00	-	13	bad s
donothing64	11787676.42	4.00	-	13	bad s
donothing128	11745060.76	4.06	-	13	bad s
NOP_OAAT_read64	11372846.37	14.00	-	47	test
BadHash	769.94	73.97	-	47	bad
sumhash	10699.57	29.53	-	363	bad
sumhash32	42877.79	23.12	-	863	UB,
multiply_shift	8026.77	26.05	226.80 (8)	345	bad
pair_multiply_shift	3716.95	40.22	186.34 (3)	609	fai
crc32	383.12	134.21	257.50 (11)	422	ins
md5_32	350.53	644.31	894.12 (10)	4419	

Summary

I added some SSE assisted hashes and fast intel/arm CRC32-C, AES and SHA HW variants. See also the old https://github.com/aappleby/smhasher/wiki, the improved, but unmaintained fork https://github.com/demerphq/smhasher, and the new improved version SMHasher3 https://gitlab.com/fwojcik/smhasher3.

So the fastest hash functions on x86_64 without quality problems are:

- xxh3low
- wyhash
- ahash64
- t1ha2_atonce
- komihash
- FarmHash *(not portable, too machine specific: 64 vs 32bit, old gcc, ...)*
- halftime_hash128
- Spooky32
- pengyhash
- nmhash32
- mx3
- MUM/mir *(different results on 32/64-bit archs, lots of bad seeds to filter out)*
- fasthash32

HASHING SCHEMES

Approach #1: Chained Hashing

Approach #2: Linear Probe Hashing

Approach #3: Robin Hood Hashing

Approach #4: Hopscotch Hashing

Approach #5: Cuckoo Hashing

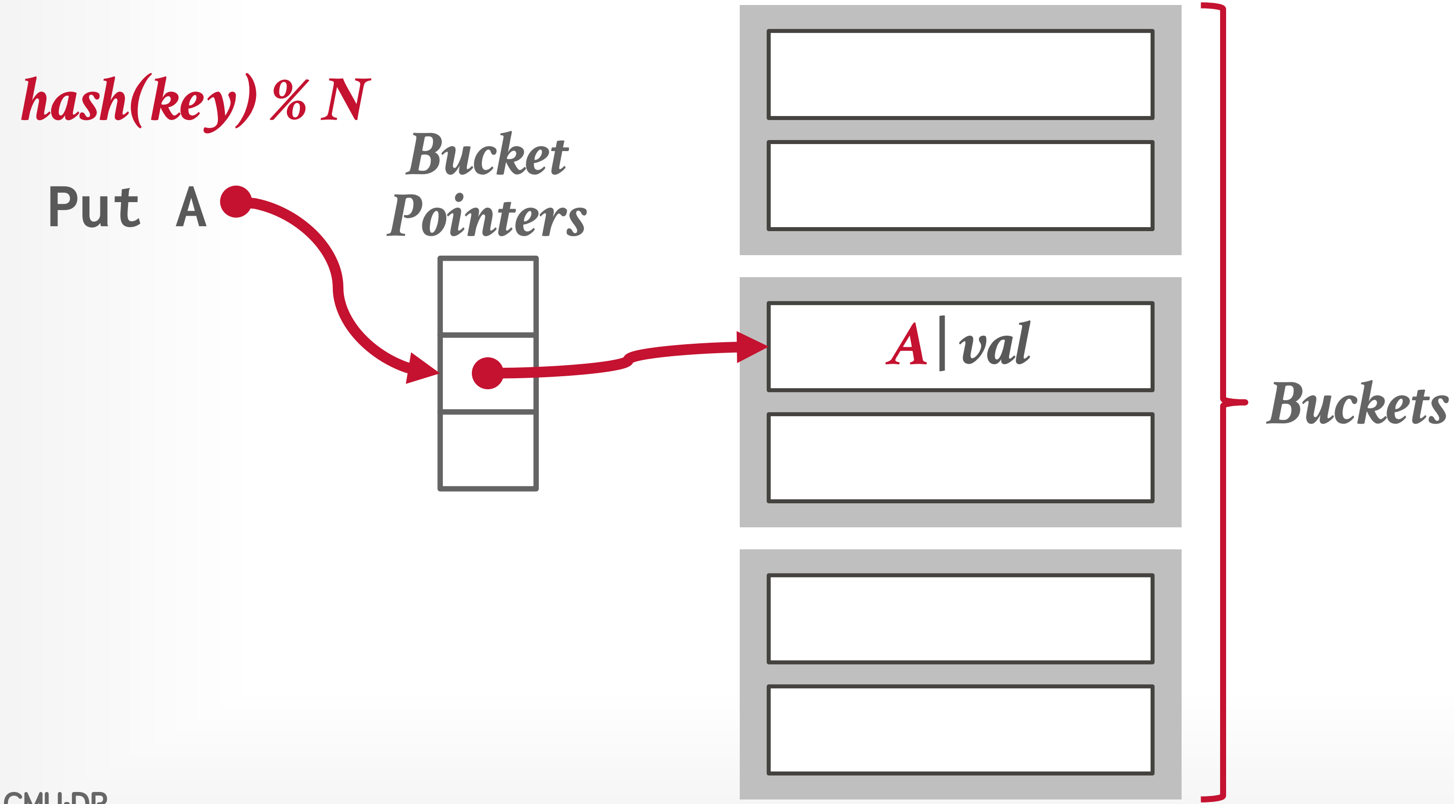
CHAINED HASHING

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

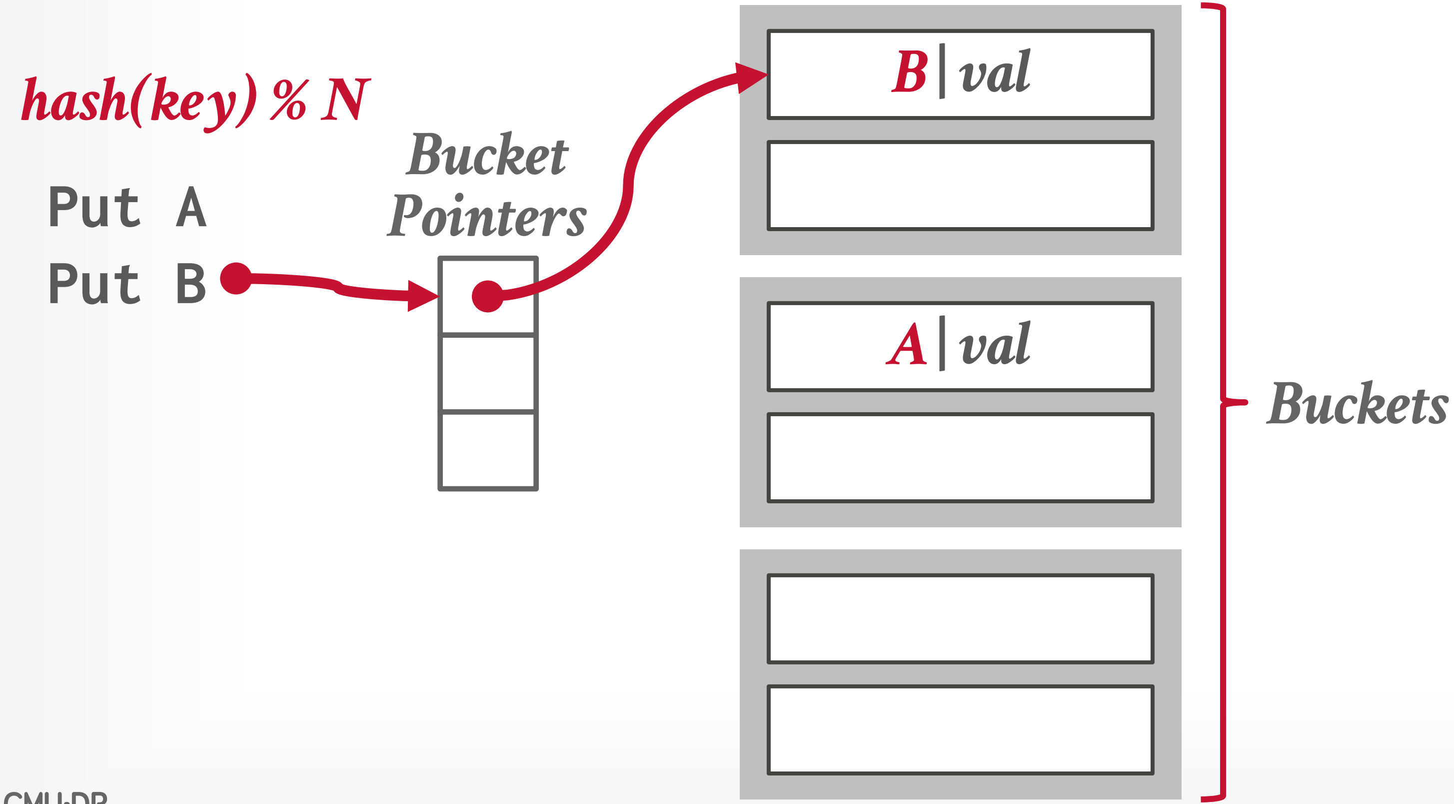
Resolve collisions by placing all elements with the same hash key into the same bucket.

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

CHAINED HASHING



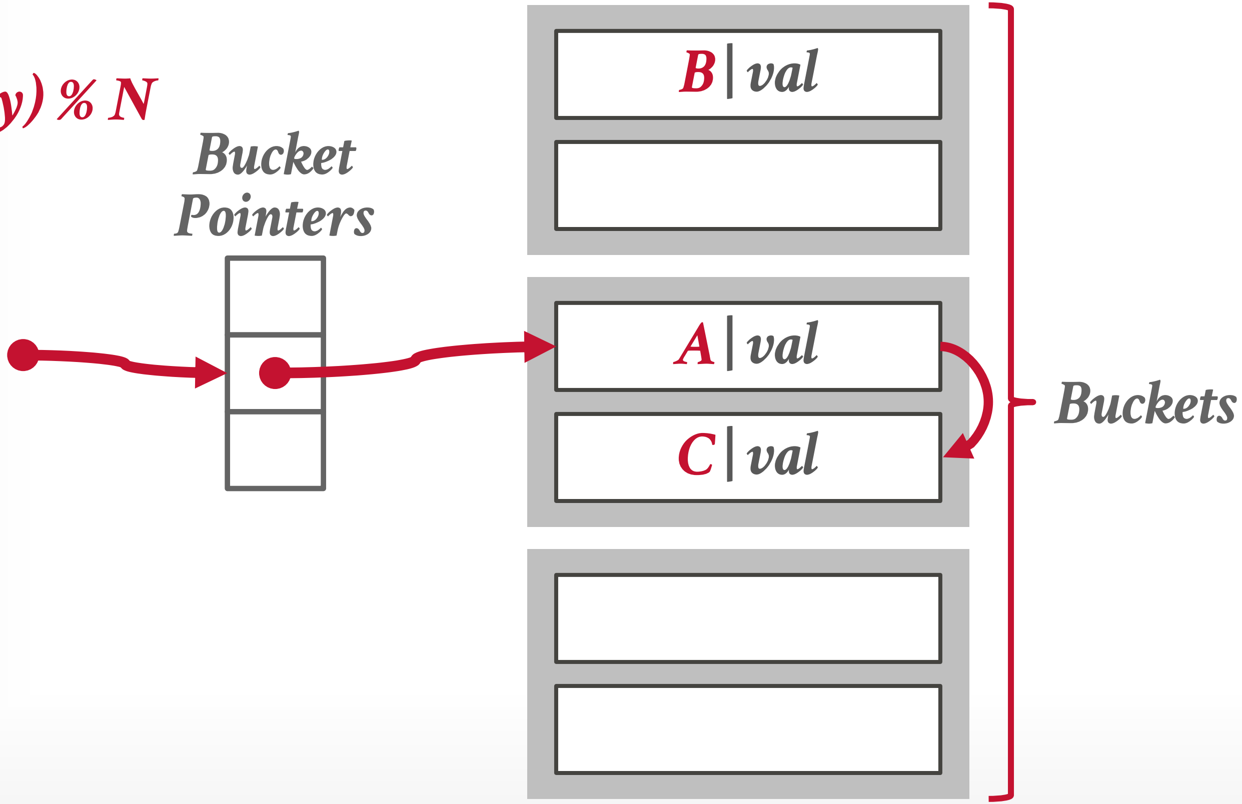
CHAINED HASHING



CHAINED HASHING

hash(key) % N

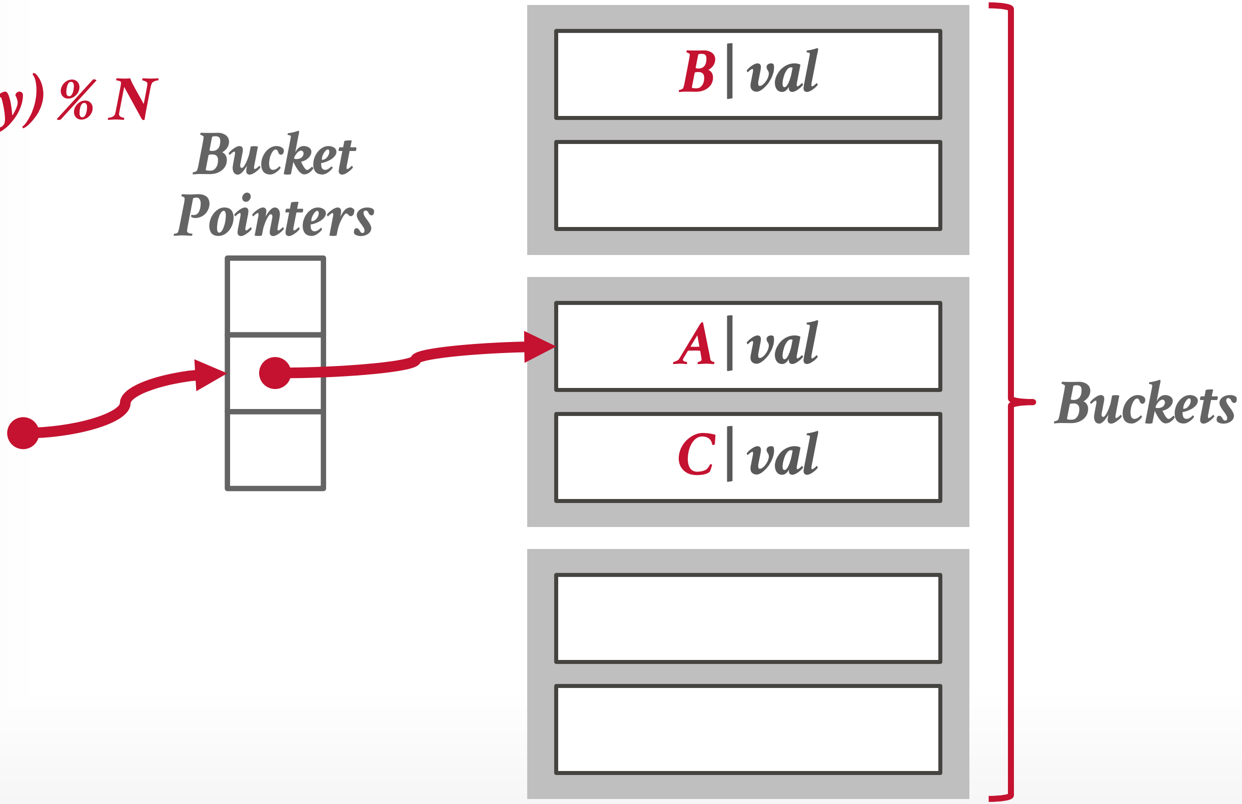
Put A
Put B
Put C



CHAINED HASHING

hash(key) % N

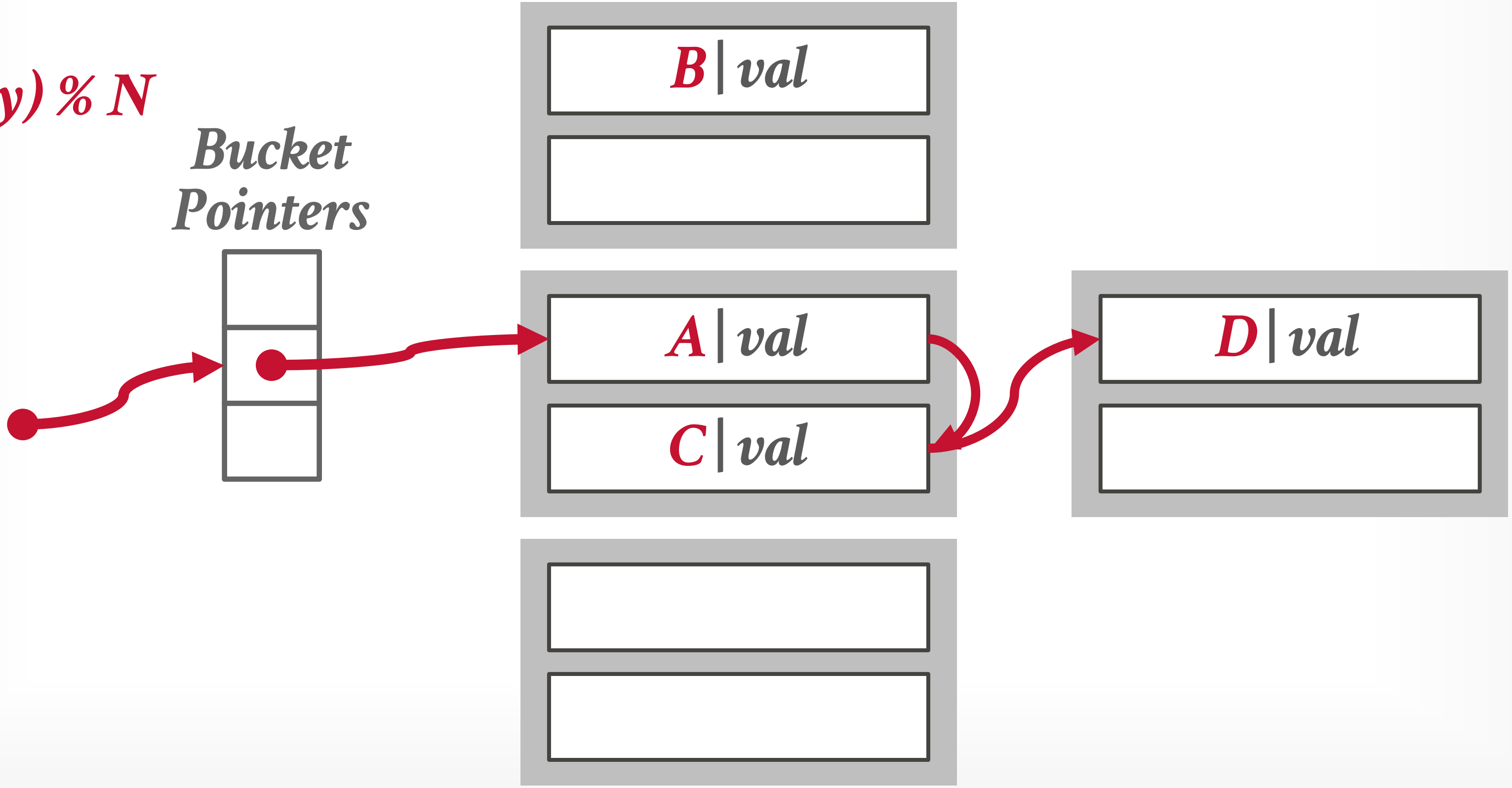
Put A
Put B
Put C
Put D



CHAINED HASHING

hash(key) % N

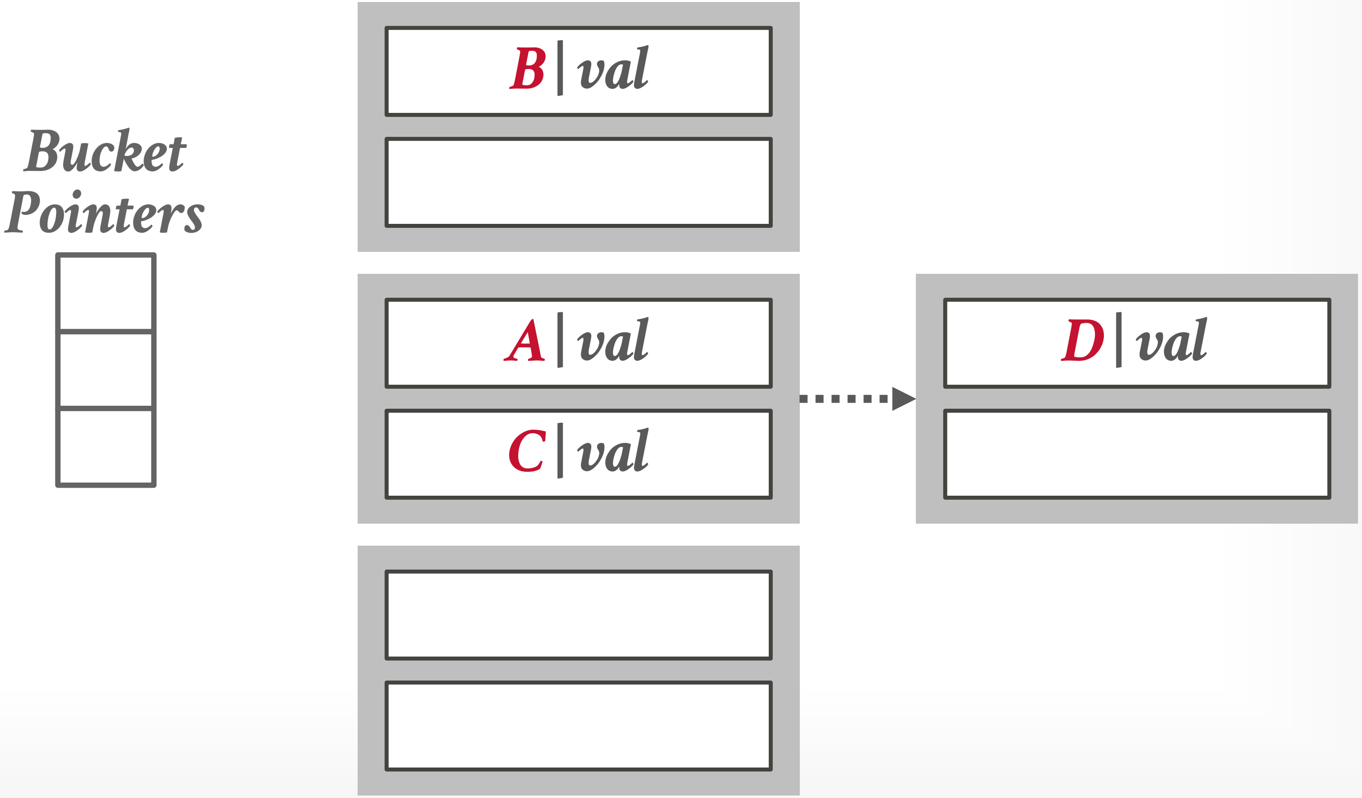
Put A
Put B
Put C
Put D



CHAINED HASHING

hash(key) % N

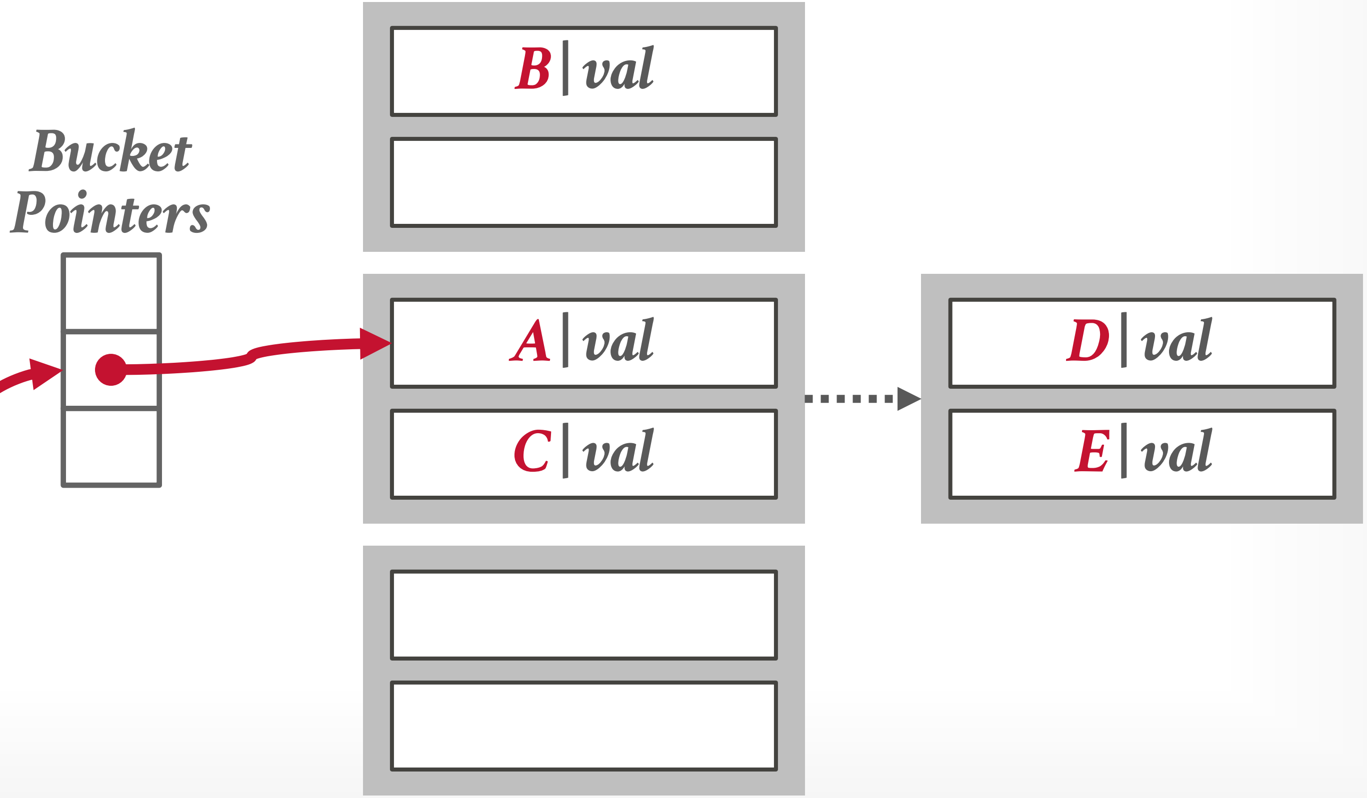
Put A
Put B
Put C
Put D
Put E



CHAINED HASHING

hash(key) % N

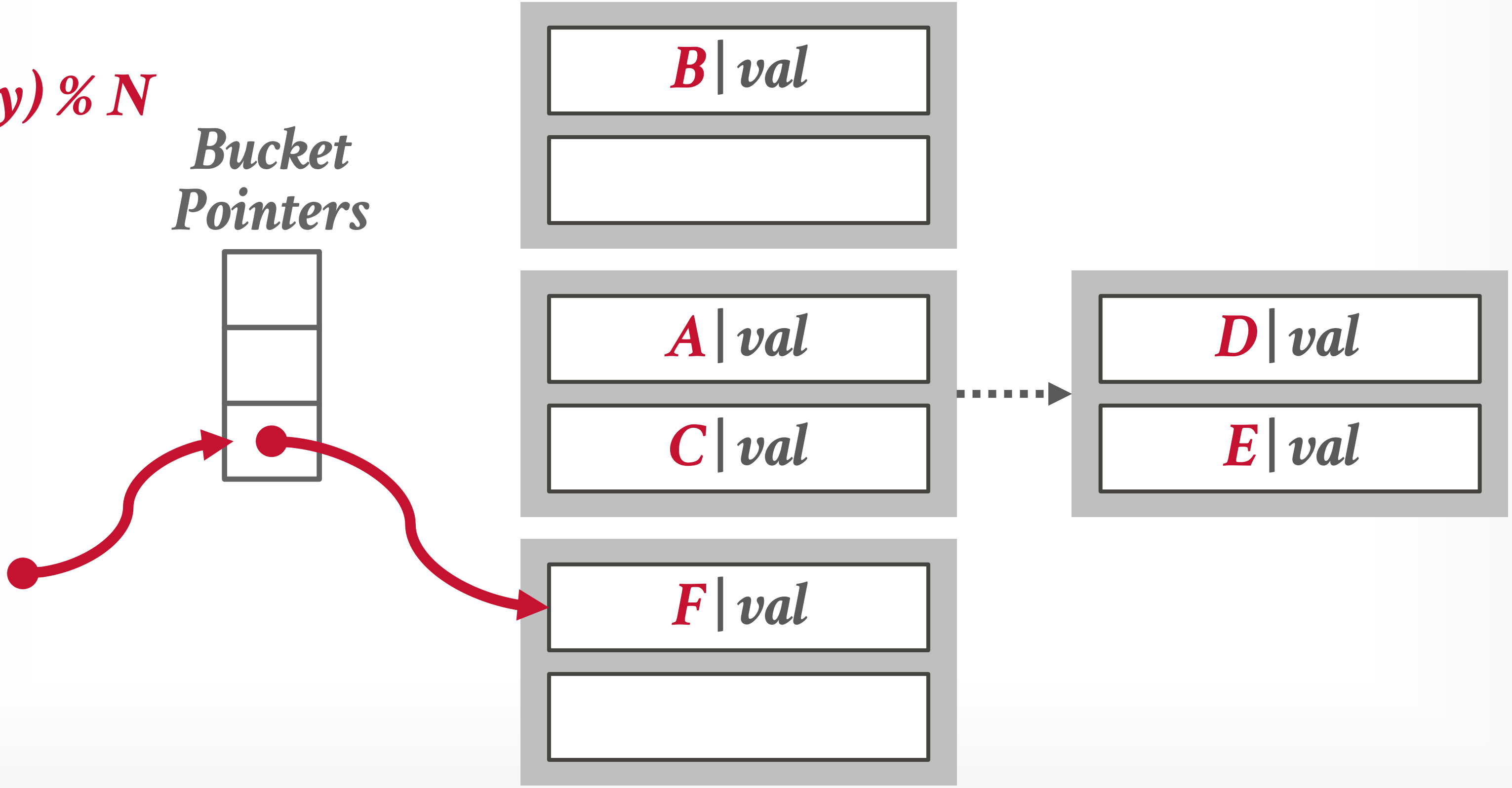
Put A
Put B
Put C
Put D
Put E



CHAINED HASHING

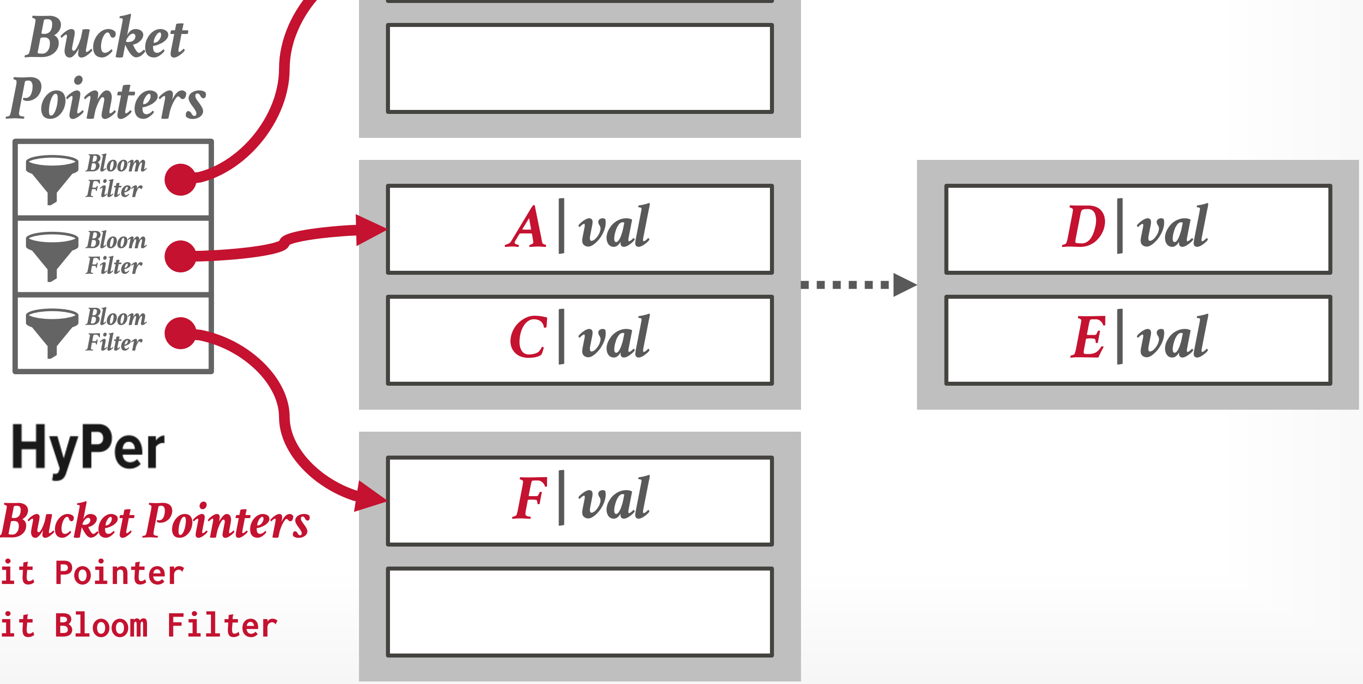
hash(key) % N

Put A
Put B
Put C
Put D
Put E
Put F



CHAINED HASHING

hash(key) % N

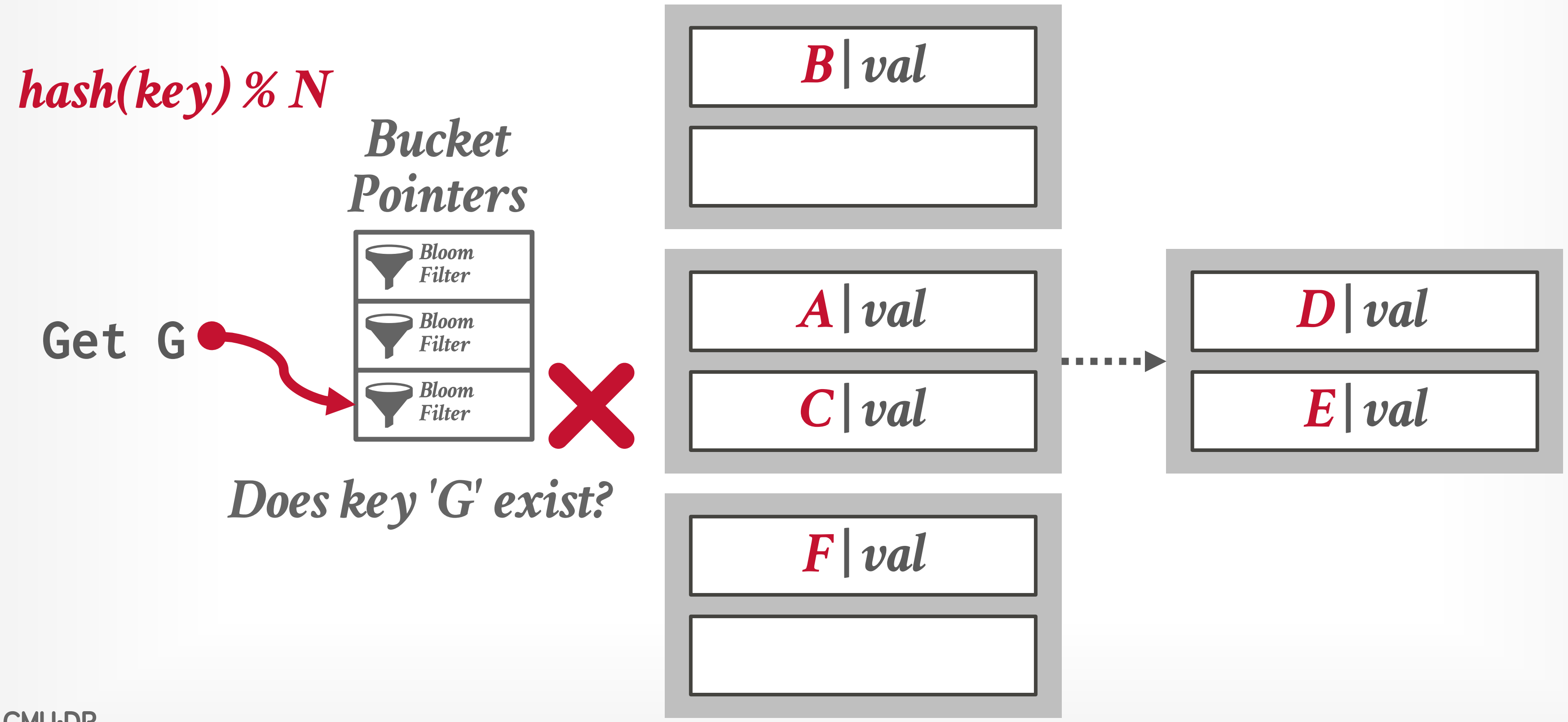


HyPer

64-bit Bucket Pointers

- 48-bit Pointer
- 16-bit Bloom Filter

CHAINED HASHING



OPEN-ADDRESSING HASHING

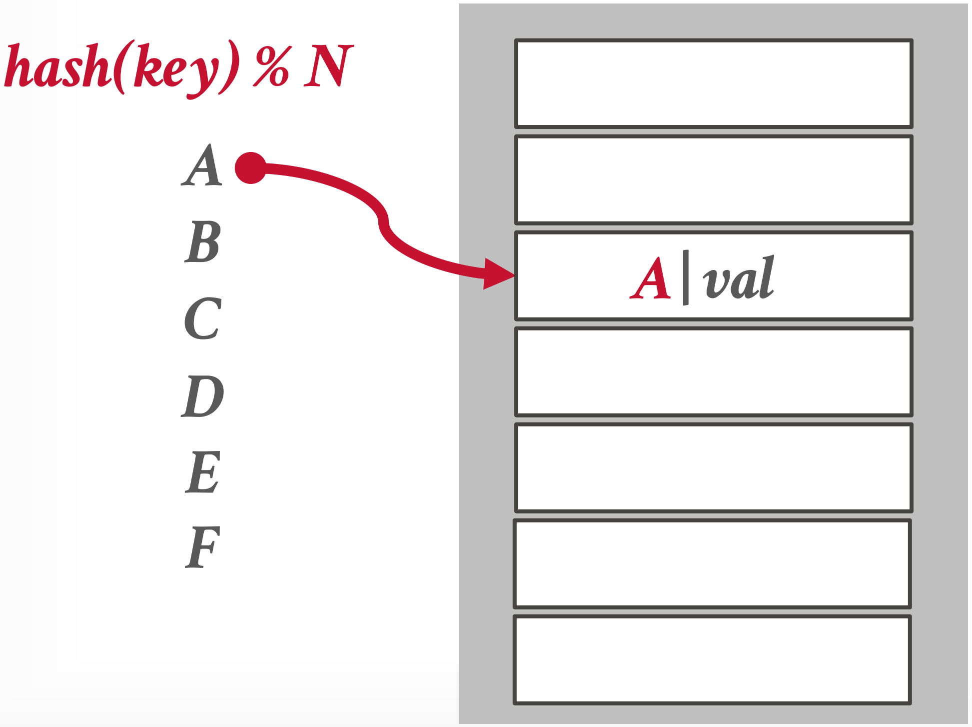
Single giant table of slots. Resolve collisions by searching for the next free slot in the table.

→ 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.

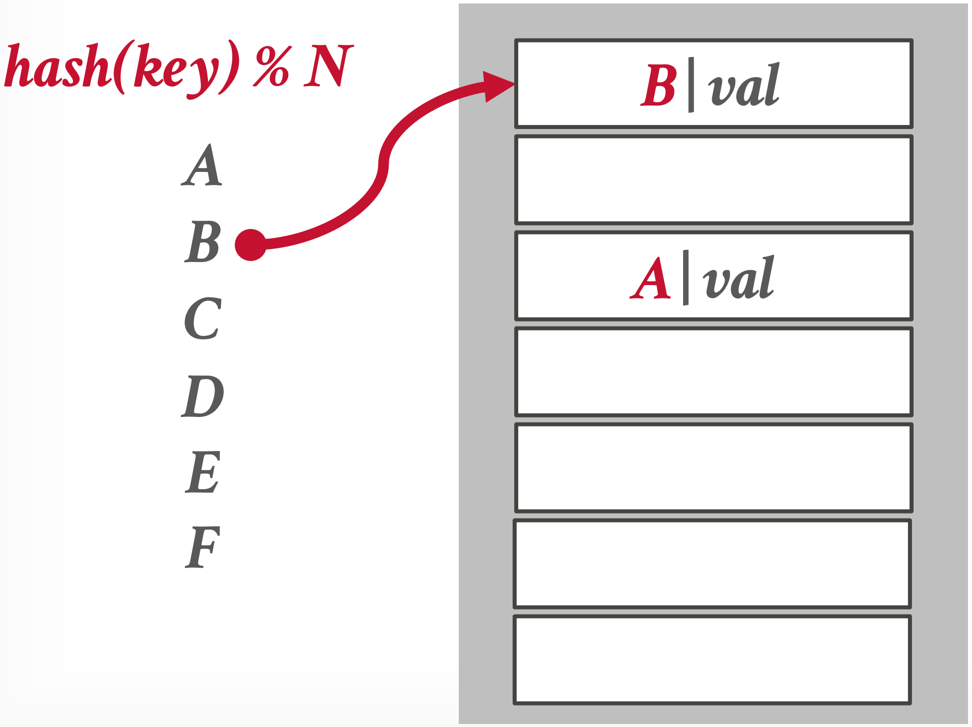
Different probing schemes:

→ **Linear**: Scan slots sequentially to find entry / empty slot.
→ **Quadratic**: Jump to slots based on quadratic equation.

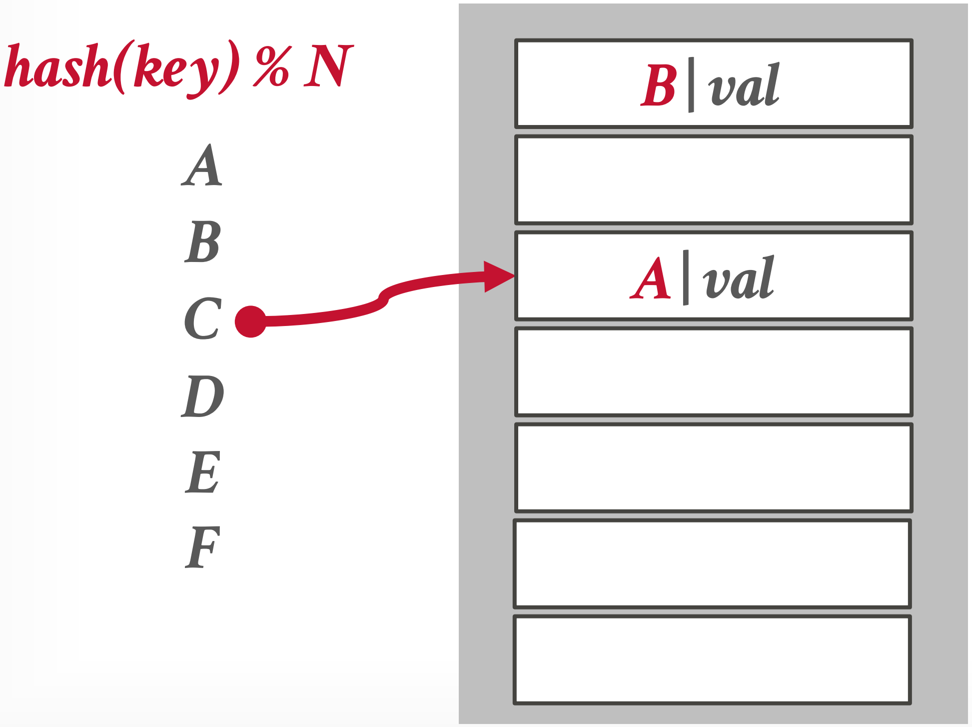
LINEAR PROBE HASHING



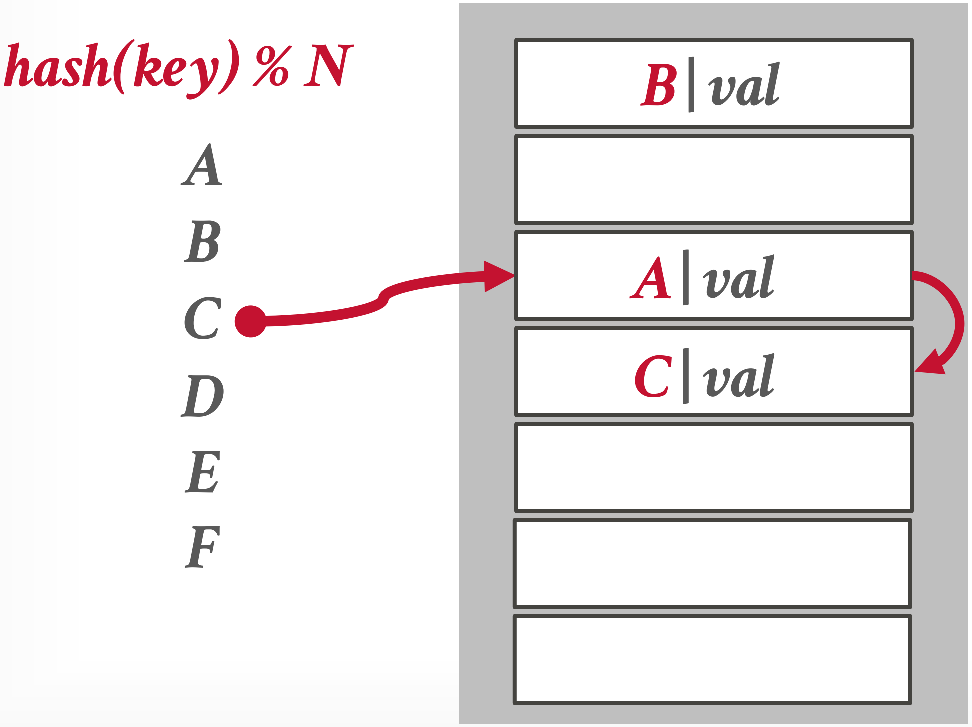
LINEAR PROBE HASHING



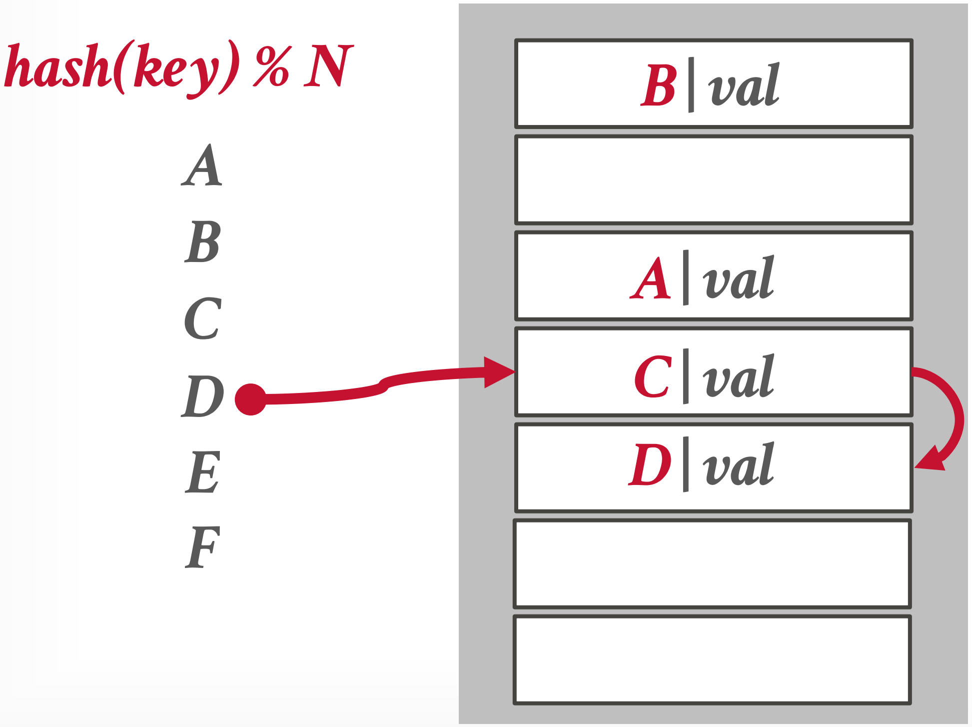
LINEAR PROBE HASHING



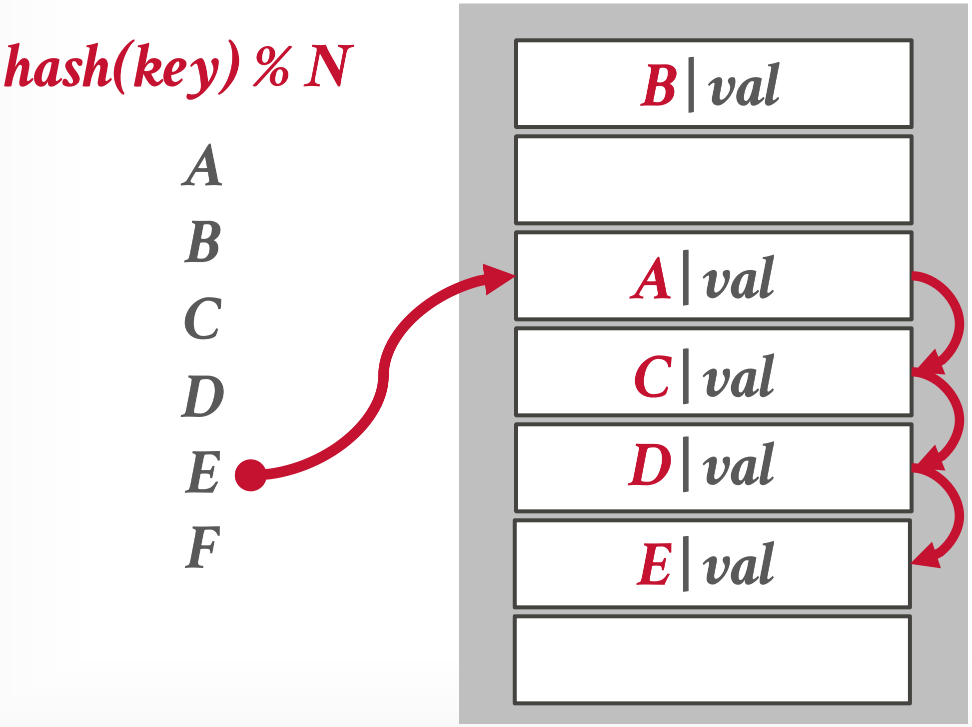
LINEAR PROBE HASHING



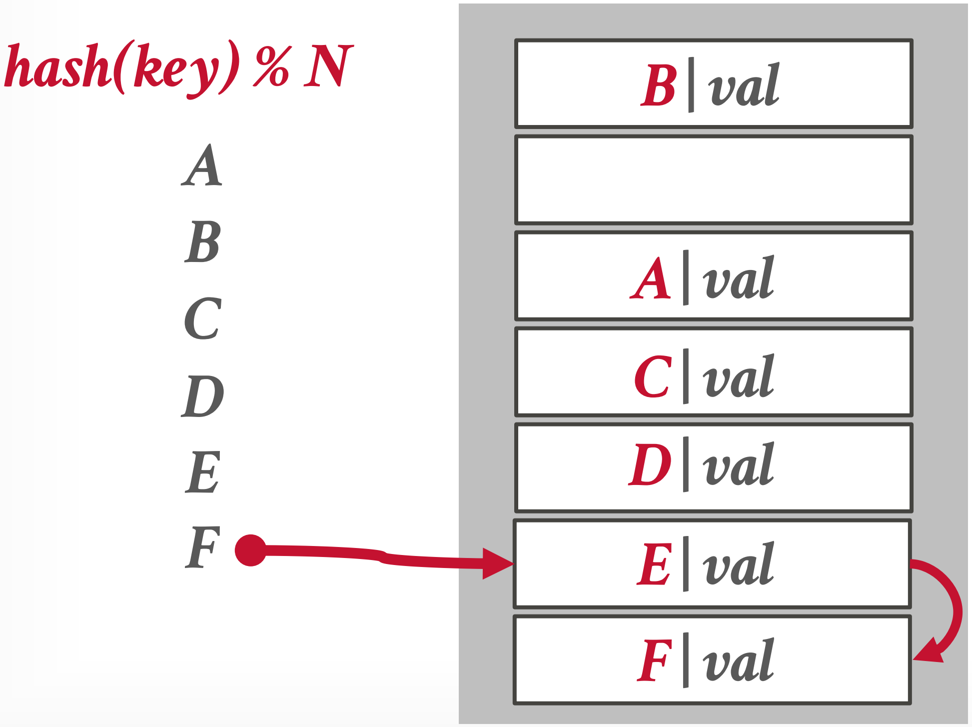
LINEAR PROBE HASHING



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.

Some research claims this is the best approach. Real-world results vary.

ROBIN HOOD HASHING
FOUNDATIONS OF COMPUTER SCIENCE 1985

ROBIN

Variant of linear pr
from "rich" keys an
→ Each key tracks the
where its optimal p
→ On insert, a key tak
is farther away fro
key.

Some research cla
Real-world results

QuestDB

Building a faster hash table for high performance SQL joins

November 23, 2023 · 11 min read

Andrey Pechkurov
Core Database Engineer

If you run a **JOIN** or a **GROUP BY** in a database of your choice, there is a good chance that there is a hash table at the core of the data processing. At QuestDB, we have **FastMap**, a hash table used for hash join and aggregate handling. While high performing, its design is a bit unconventional as it differs from most general-purpose hash tables.

In this article, we'll tell you why hash tables are important to databases, how QuestDB's **FastMap** works and why it speeds up SQL execution.

ROBIN HOOD HASHING
FOUNDATIONS OF COMPUTER SCIENCE 1985

ROBIN

Variant of linear pr
from "rich" keys an
→ Each key tracks the
where its optimal p
→ On insert, a key tal
is farther away fro
key.

Some research cla
Real-world result

QuestDB

Building a faster hash table for high performance SQL joins

November 23, 2023 · 11 min read

Andrey
Core Datab

If you run a JO
good chance t
QuestDB, we h
handling. While
differs from mo

In this article, w
QuestDB's FastM

N.B. Thanks to the valuable feedback we received from the community after publishing this post, we've started experimenting with further optimizations. The most noticeable one is [Robin Hood hashing](), a linear probing enhancement aimed to minimize the number of look-up probes for the keys. If you prefer jumping to the code, the pull request is [here]().

ROBIN HOOD HASHING
FOUNDATIONS OF COMPUTER SCIENCE 1985

ROBIN

Variant of linear pr
from "rich" keys an
→ Each key tracks the
where its optimal p
→ On insert, a key tal
is farther away f

QuestDB

Building a faster hash table for high performance SQL joins

November 23, 2023 · 11 min read

Andrey
Core Datab

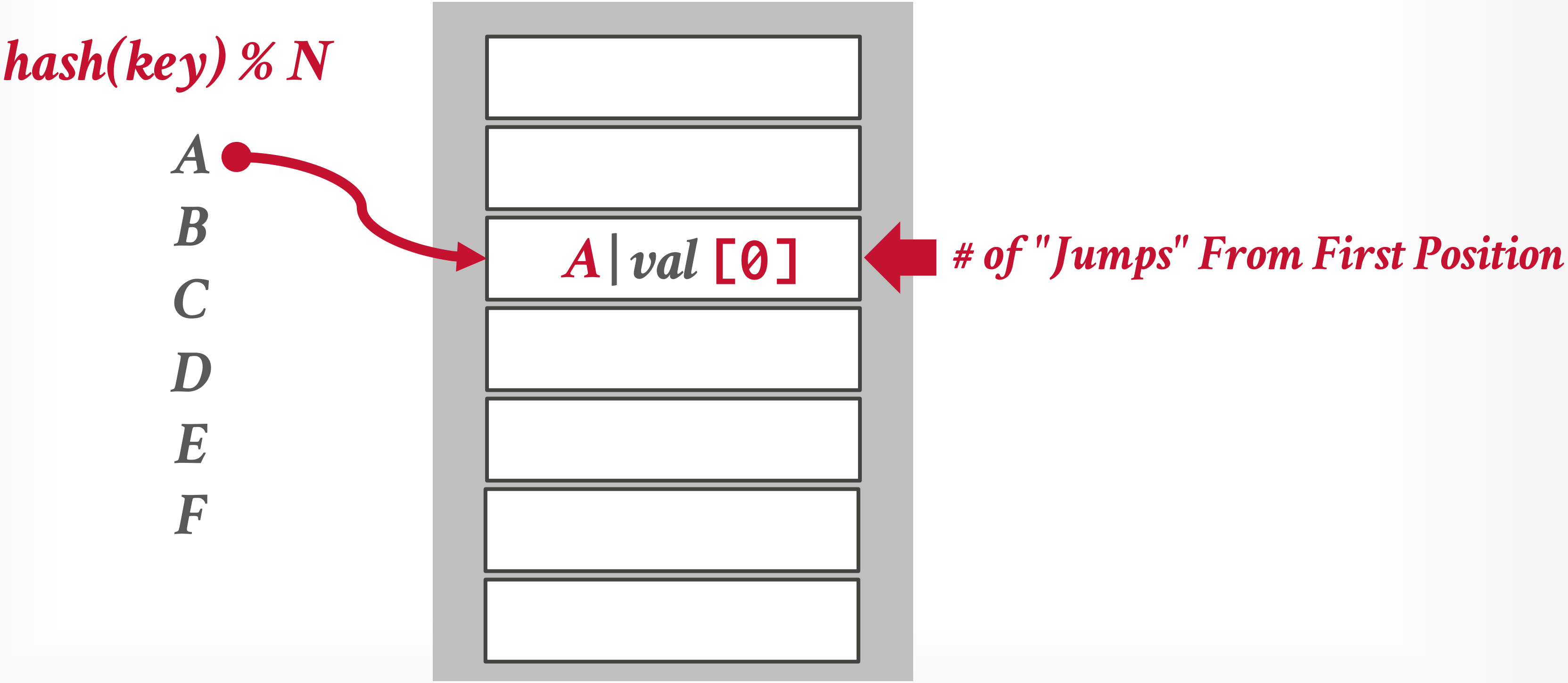
N.B. Thanks to the valuable feedback we received from the community after
t, we've started
further optimizations.
e one is Robin Hood
obing enhancement
the number of look-up
If you prefer jumping
l request is here.

Closed **chore(core): FastMap to use Robin Hood hashing** #4054
jerrinot wants to merge 7 commits into `master` from `jh_rh_hash`

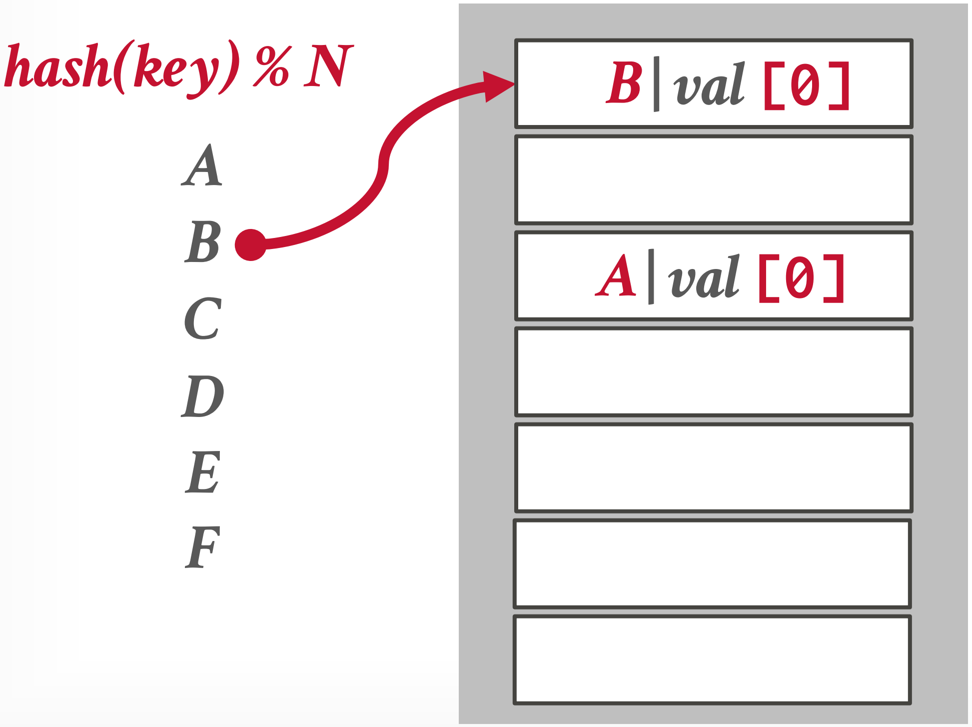
jerrinot commented on Jan 5 Contributor Author

closing as new results do not show any improvements with default load and actually it performs worse with higher loads. chances are this PR shifted bottlenecks: #4032

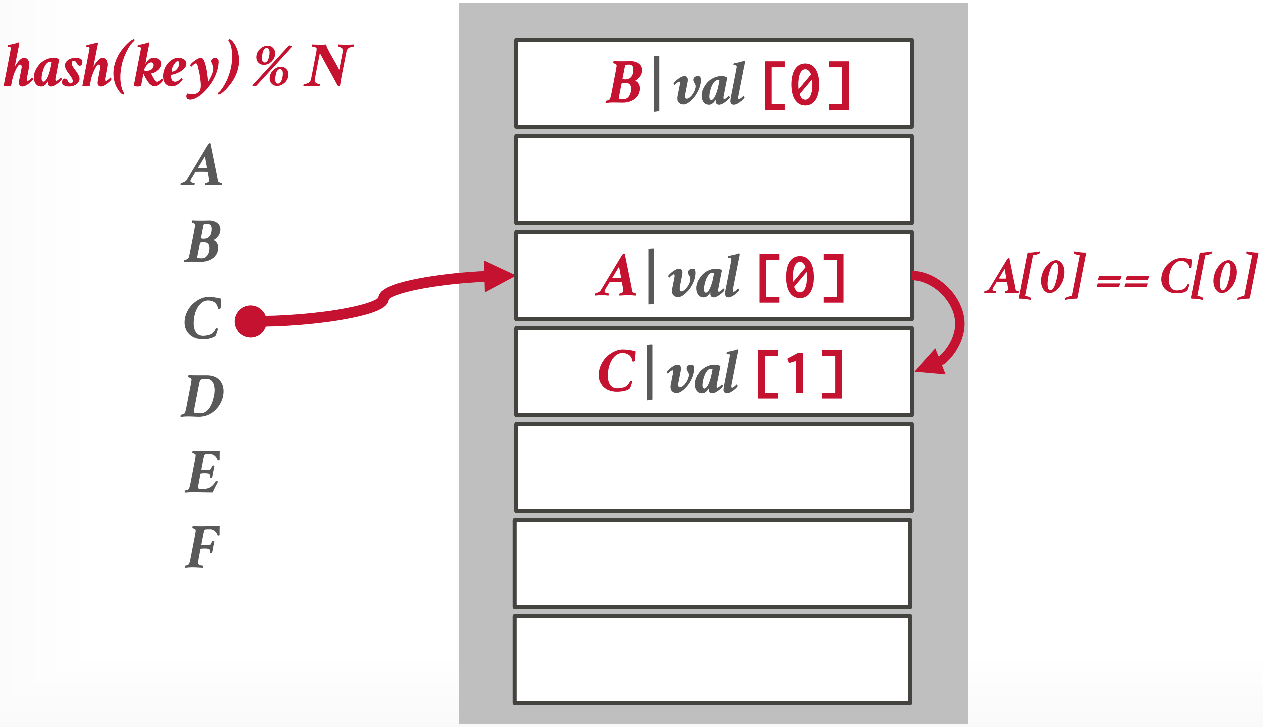
ROBIN HOOD HASHING



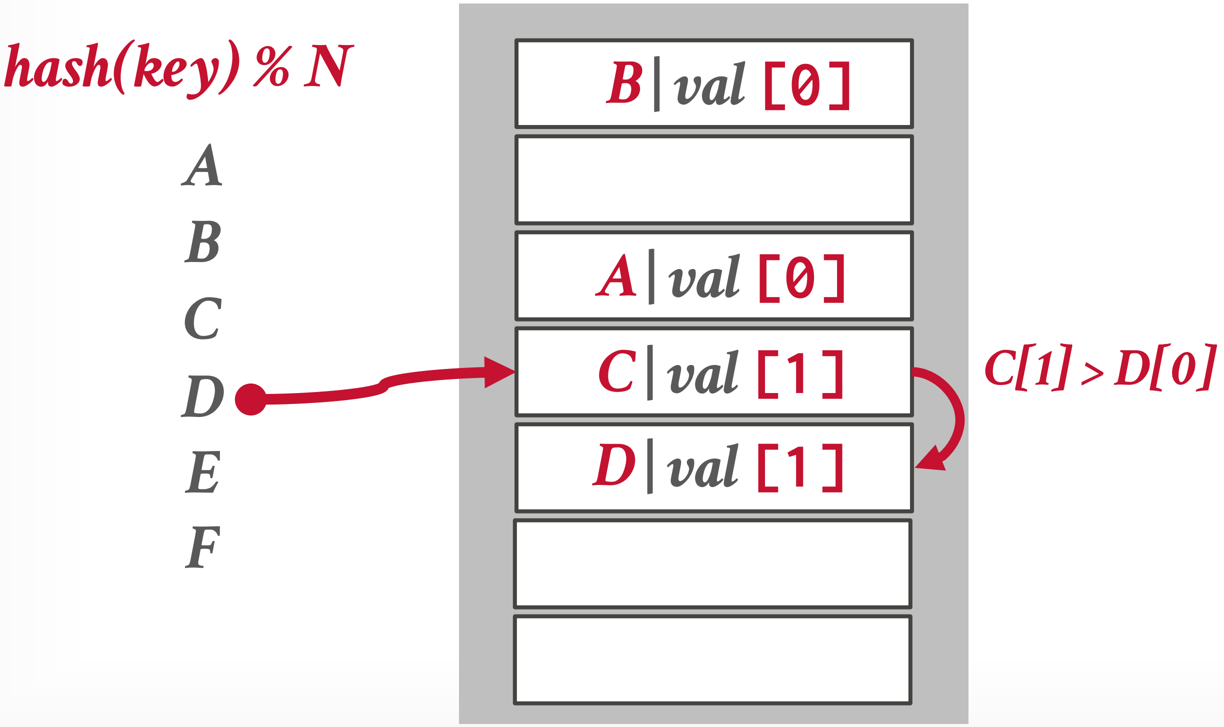
ROBIN HOOD HASHING



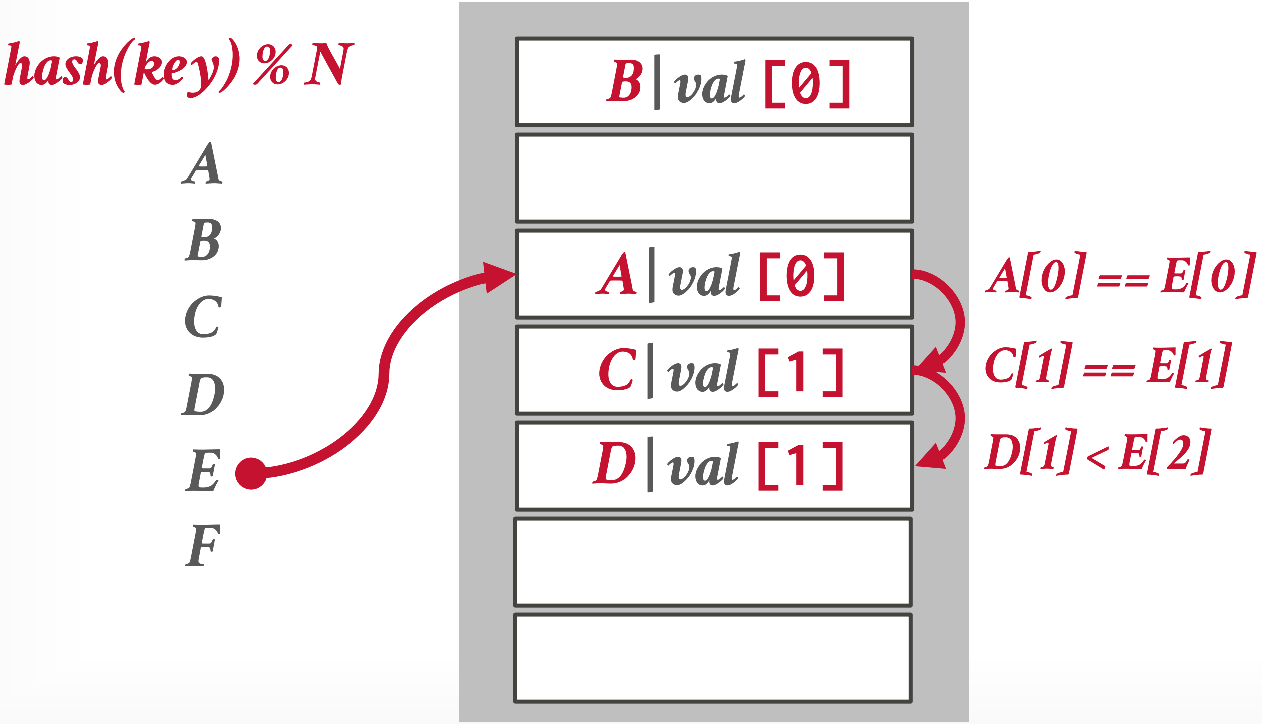
ROBIN HOOD HASHING



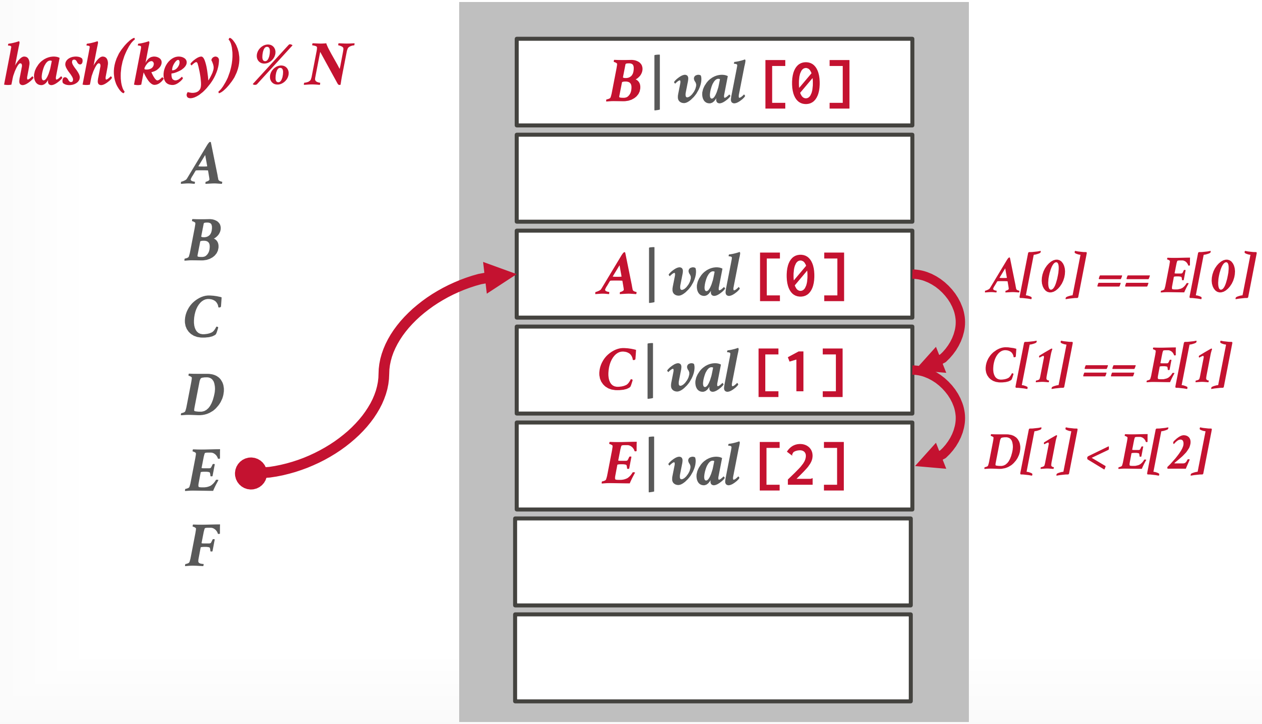
ROBIN HOOD HASHING



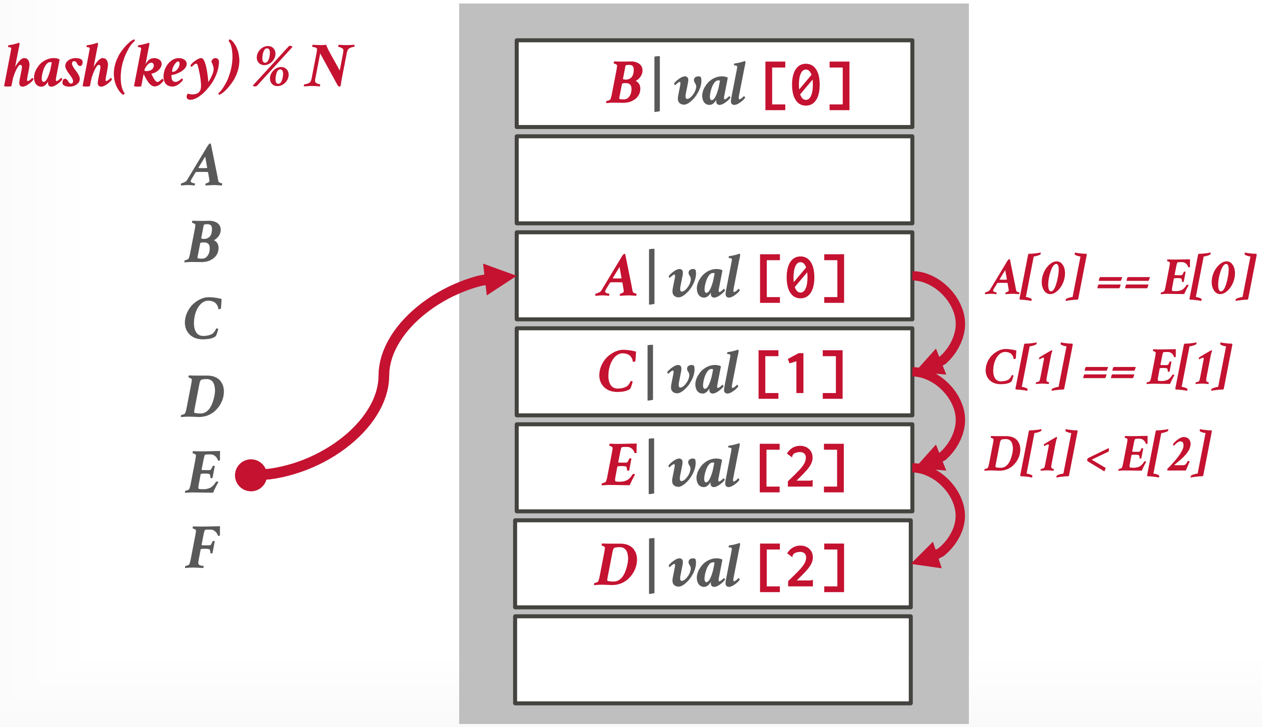
ROBIN HOOD HASHING



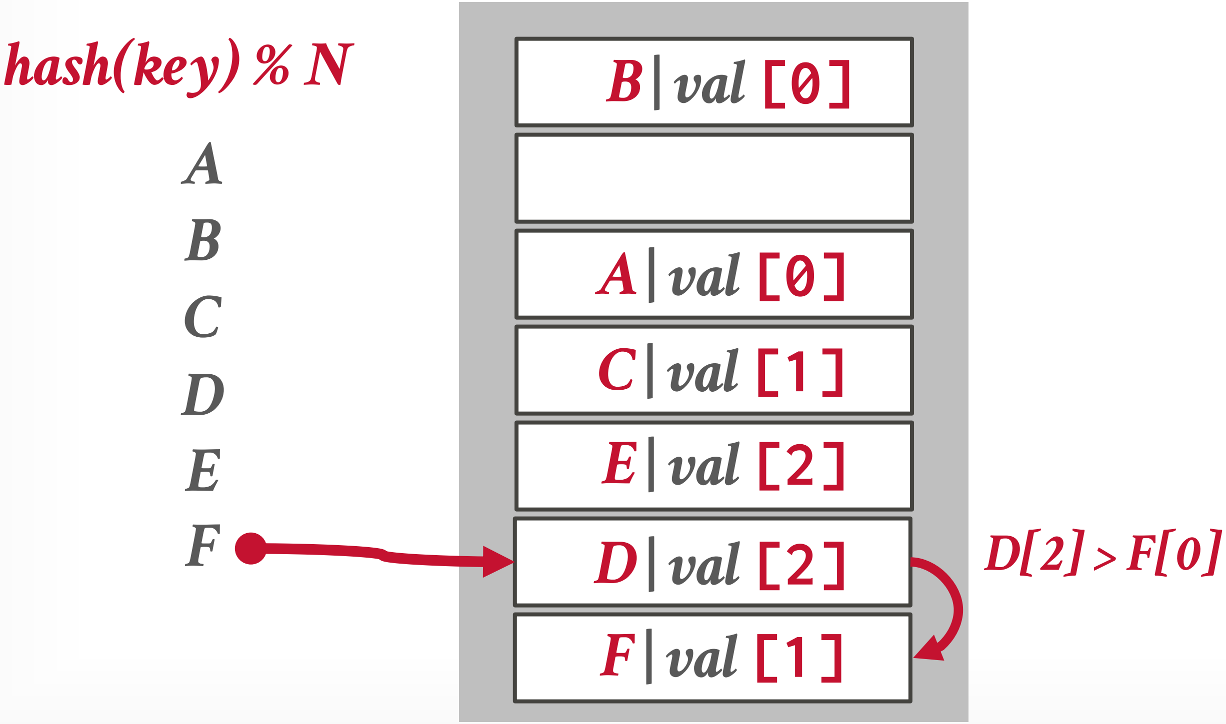
ROBIN HOOD HASHING



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.

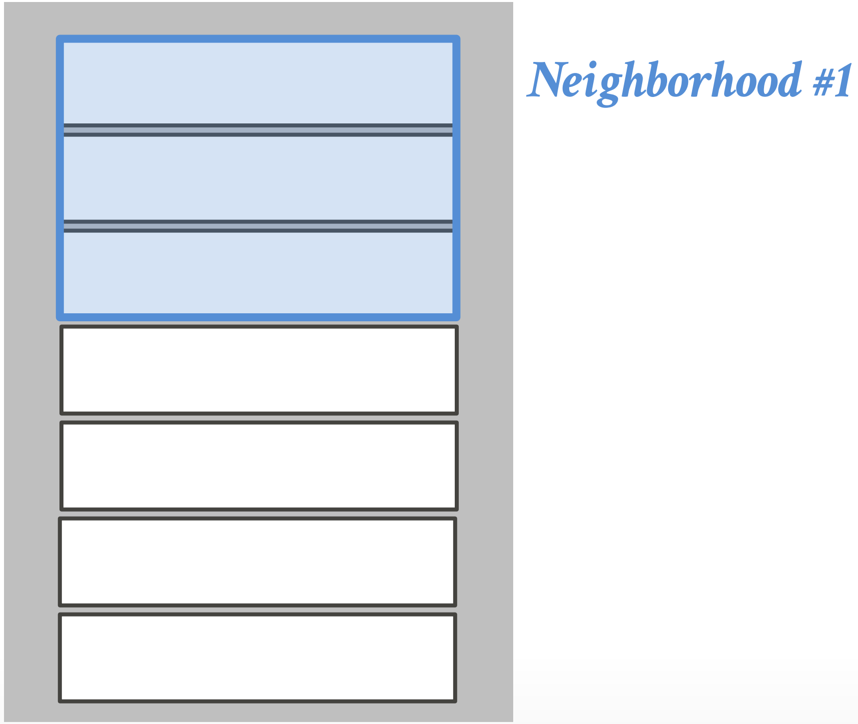
HOPSCOTCH HASHING
SYMPOSIUM ON DISTRIBUTED COMPUTING 2008

HOPSCOTCH HASHING

Neighborhood Size = 3

hash(key) % N

A
B
C
D
E
F



HOPSCOTCH HASHING

Neighborhood Size = 3

hash(key) % N

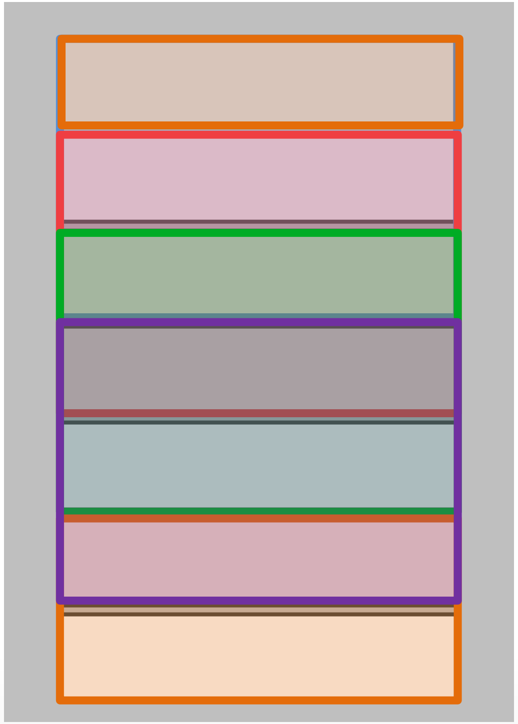
A
B
C
D
E
F

HOPSCOTCH HASHING

Neighborhood Size = 3

hash(key) % N

A
B
C
D
E
F



Neighborhood #1 *Neighborhood #6*

Neighborhood #2

Neighborhood #3

Neighborhood #4

⋮

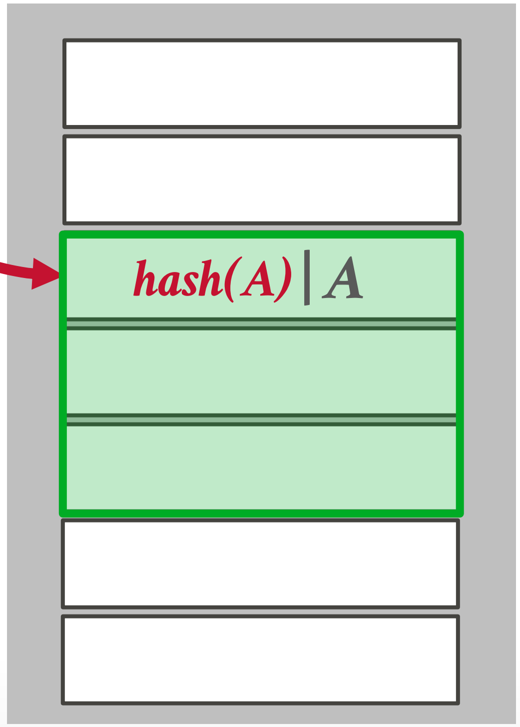
Neighborhood #6

HOPSCOTCH HASHING

Neighborhood Size = 3

hash(key) % N

A
B
C
D
E
F



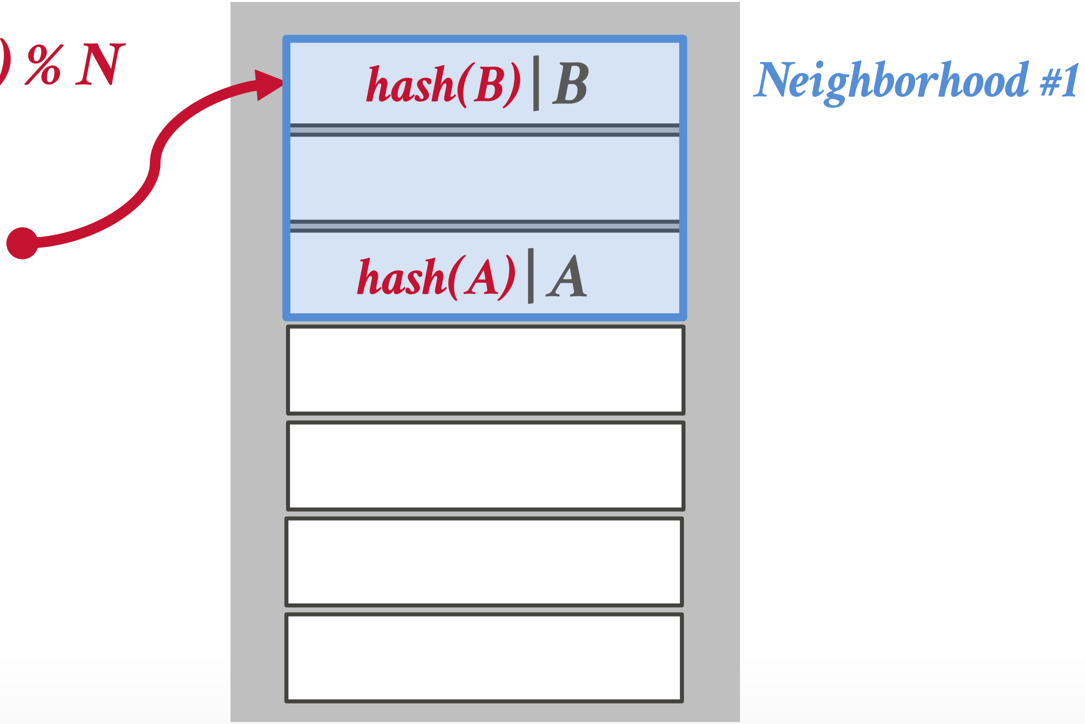
Neighborhood #3

HOPSCOTCH HASHING

Neighborhood Size = 3

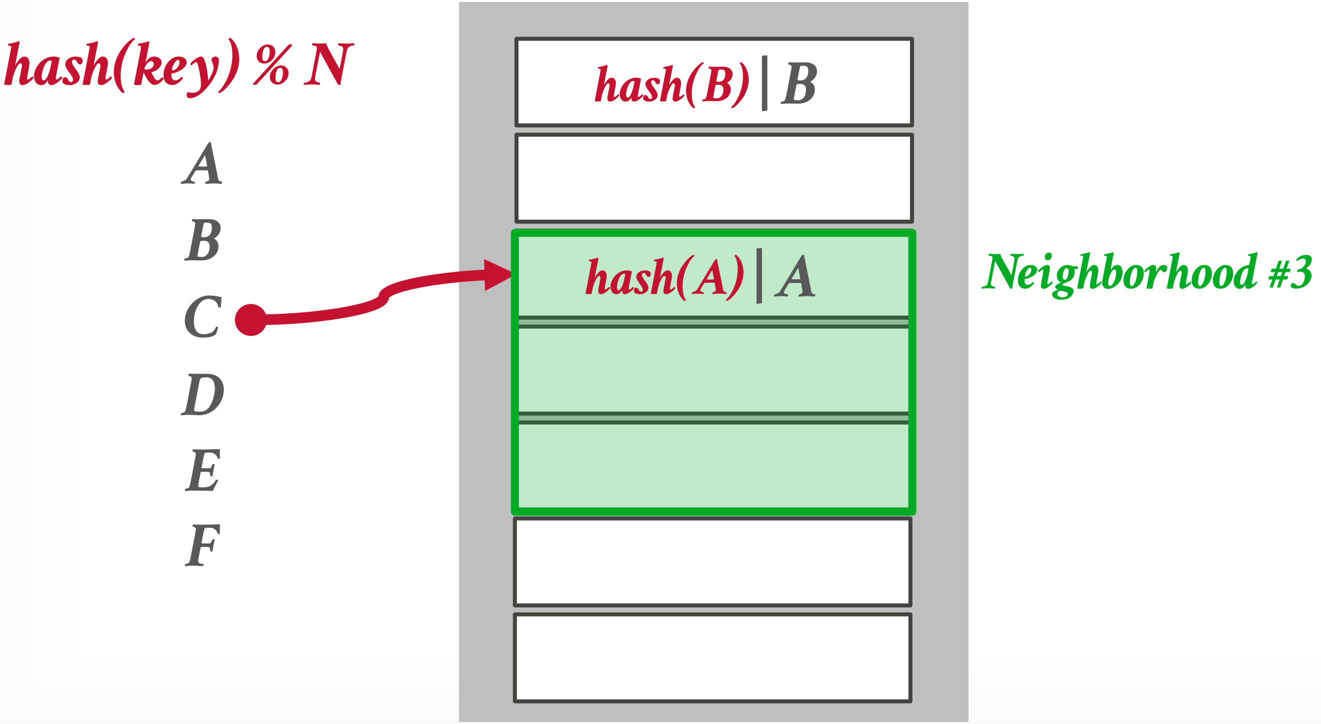
hash(key) % N

A
B
C
D
E
F



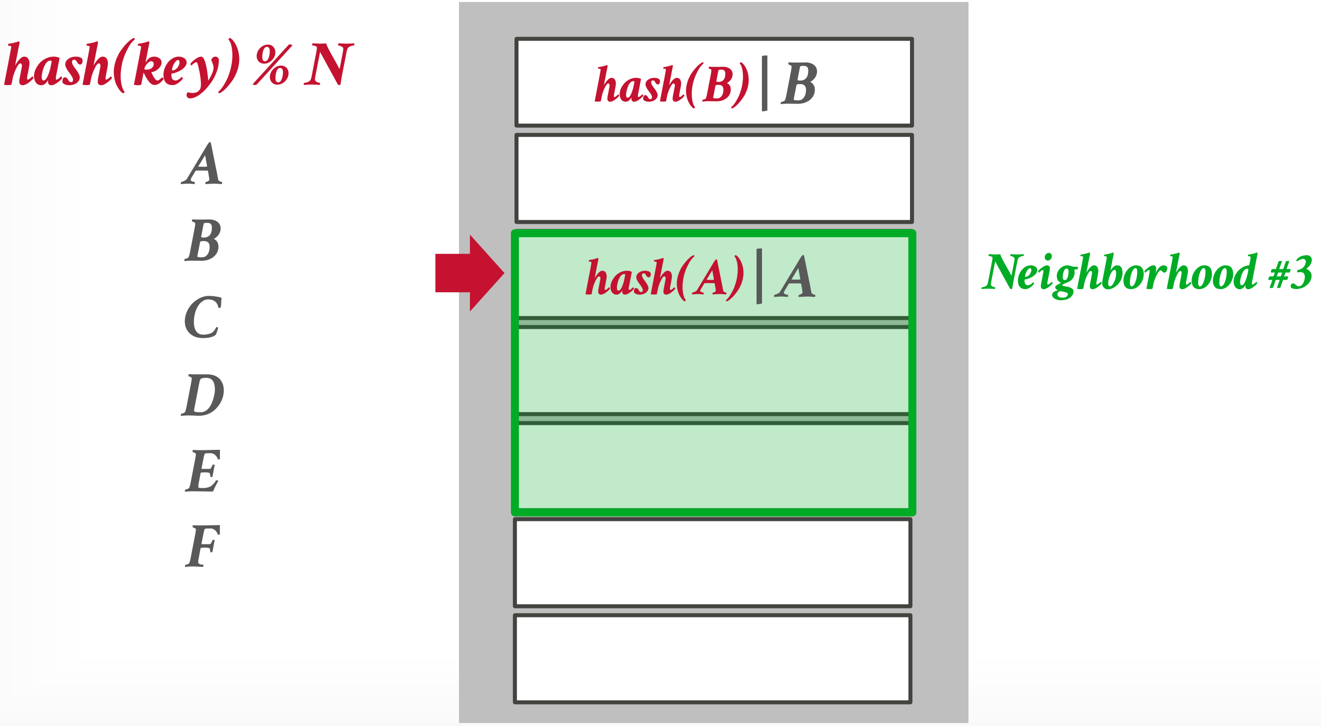
HOPSCOTCH HASHING

Neighborhood Size = 3



HOPSCOTCH HASHING

Neighborhood Size = 3

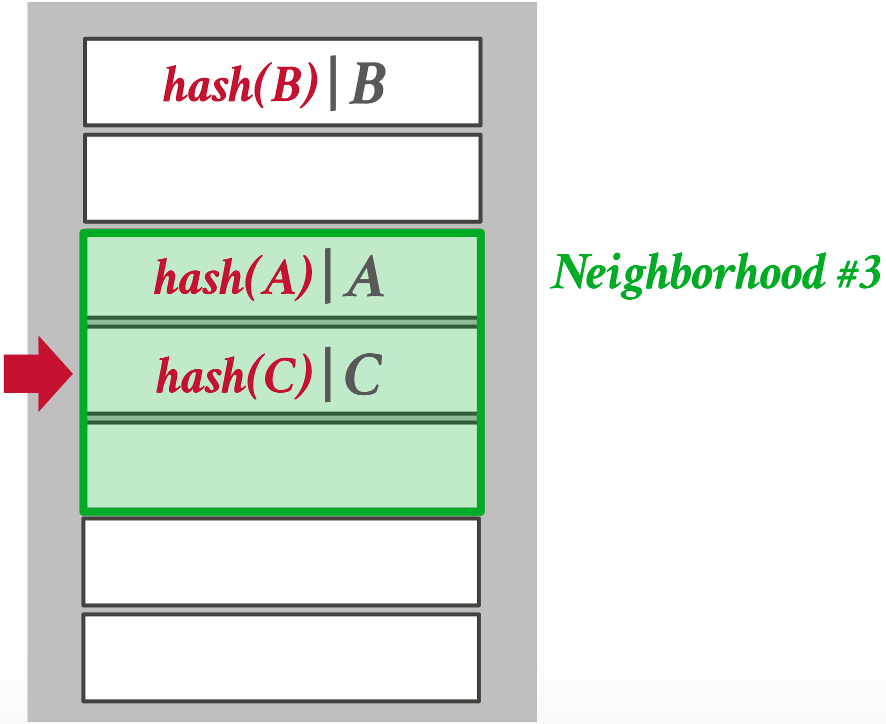


HOPSCOTCH HASHING

Neighborhood Size = 3

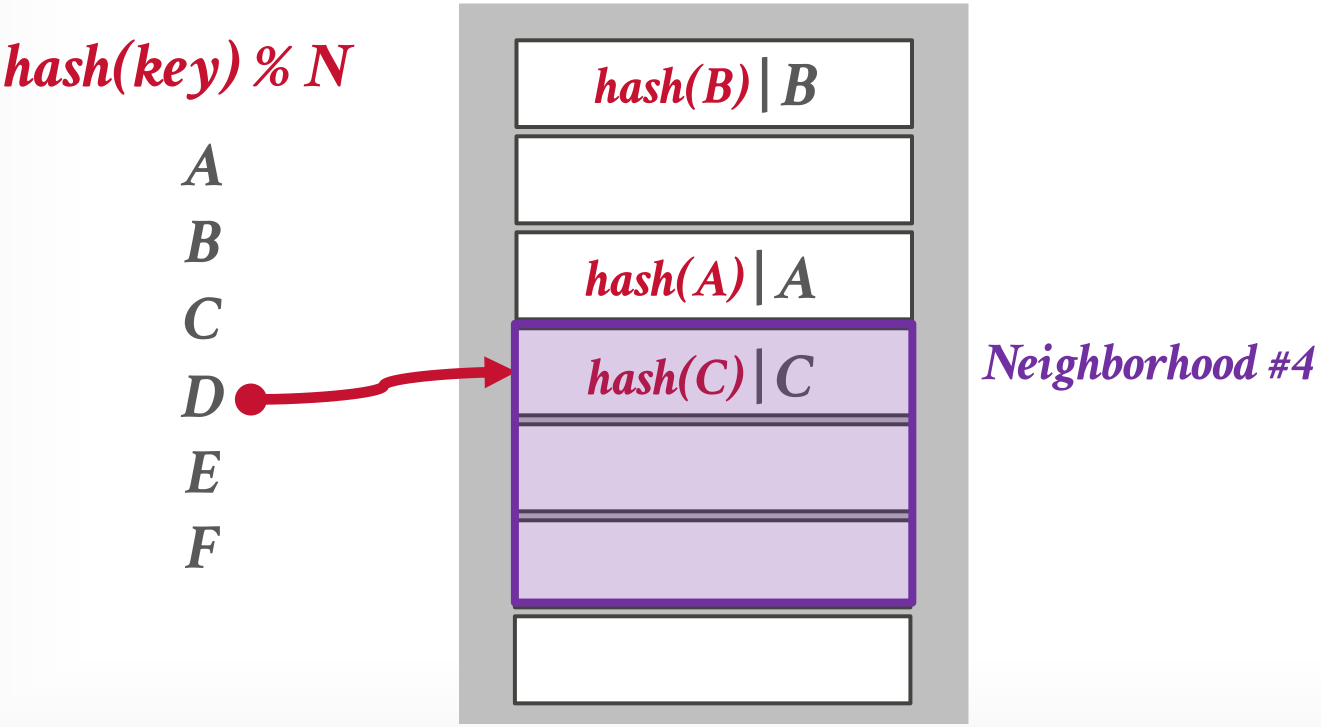
hash(key) % N

A
B
C
D
E
F



HOPSCOTCH HASHING

Neighborhood Size = 3

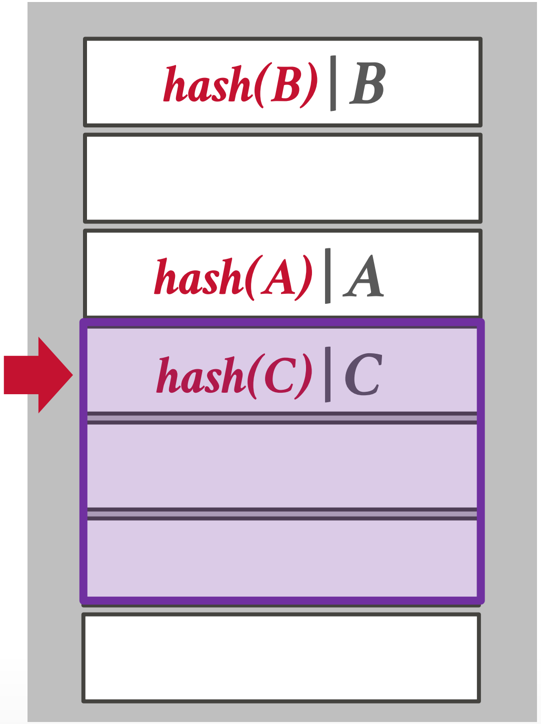


HOPSCOTCH HASHING

Neighborhood Size = 3

hash(key) % N

A
B
C
D
E
F



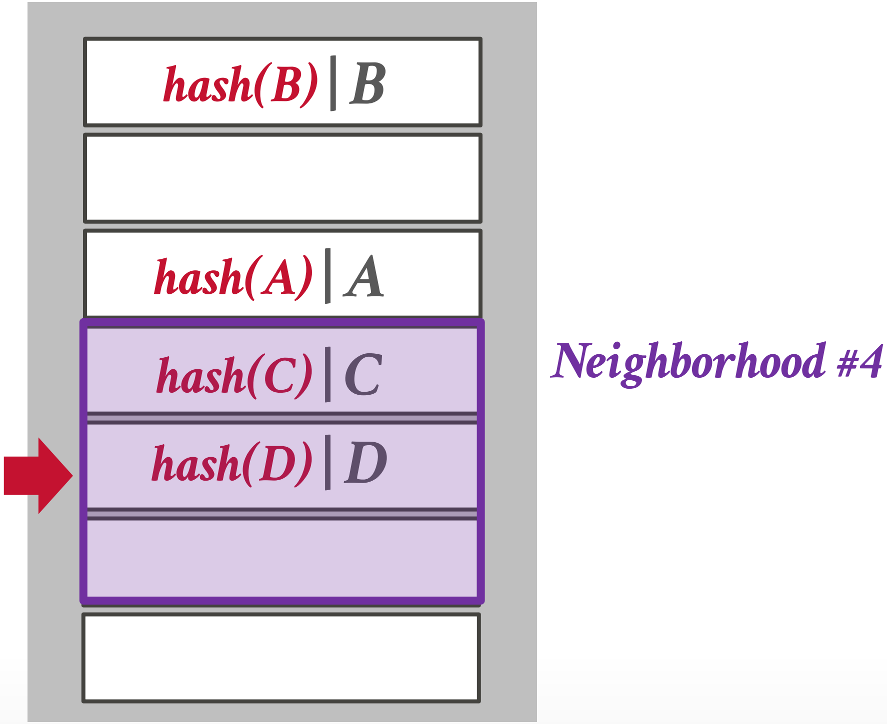
Neighborhood #4

HOPSCOTCH HASHING

Neighborhood Size = 3

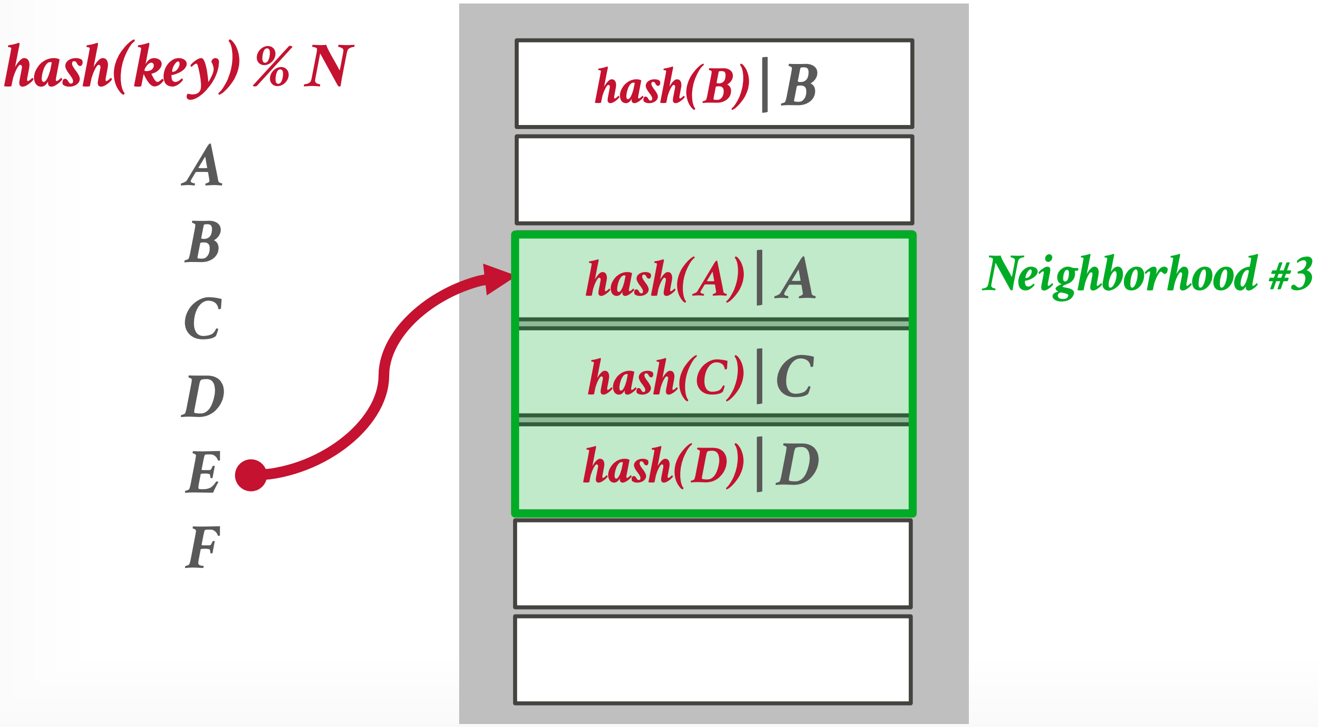
hash(key) % N

A
B
C
D
E
F



HOPSCOTCH HASHING

Neighborhood Size = 3

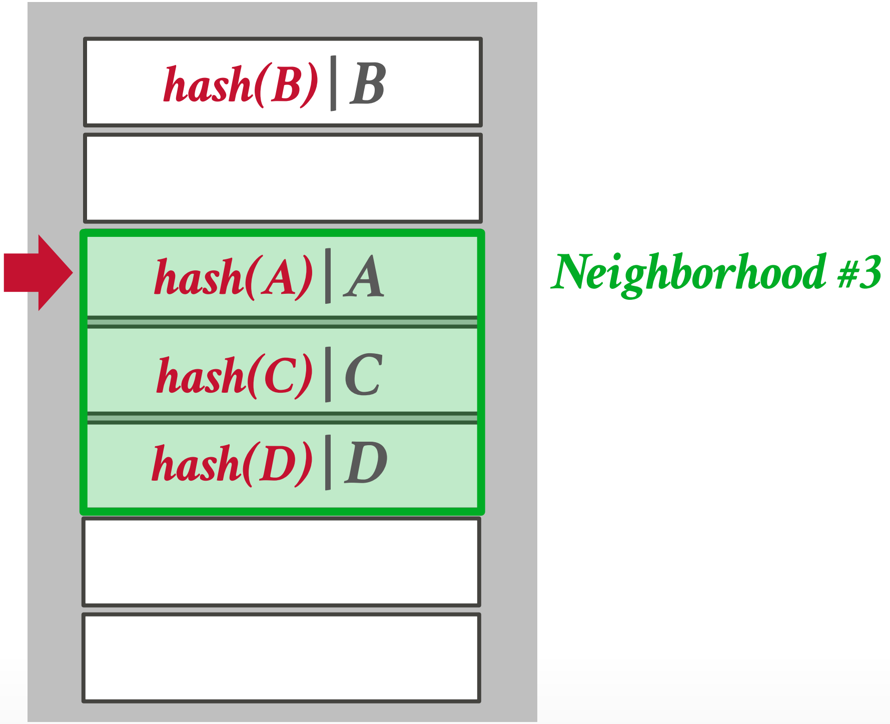


HOPSCOTCH HASHING

Neighborhood Size = 3

hash(key) % N

A
B
C
D
E
F

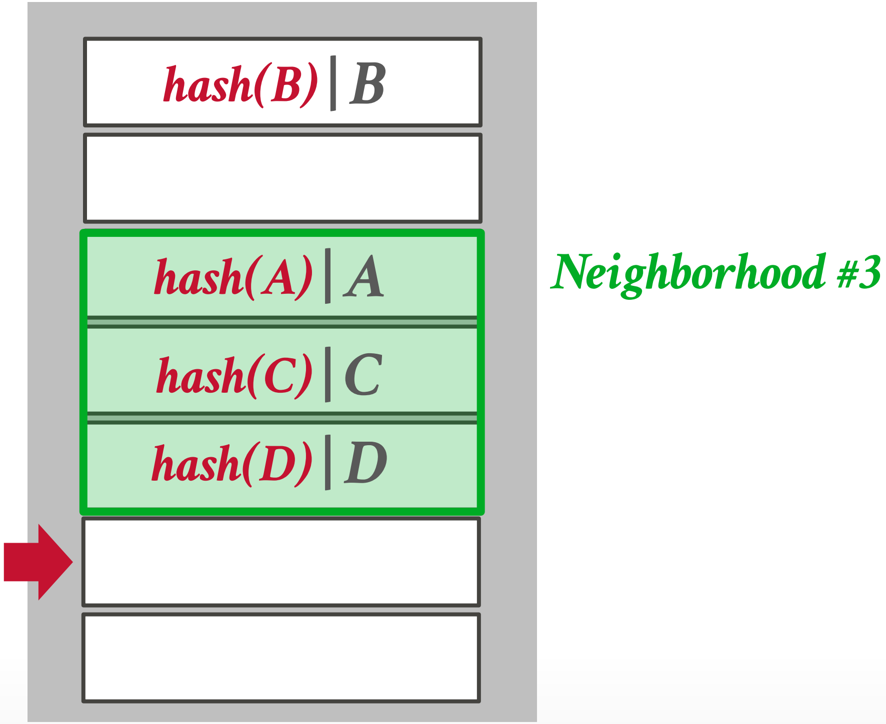


HOPSCOTCH HASHING

Neighborhood Size = 3

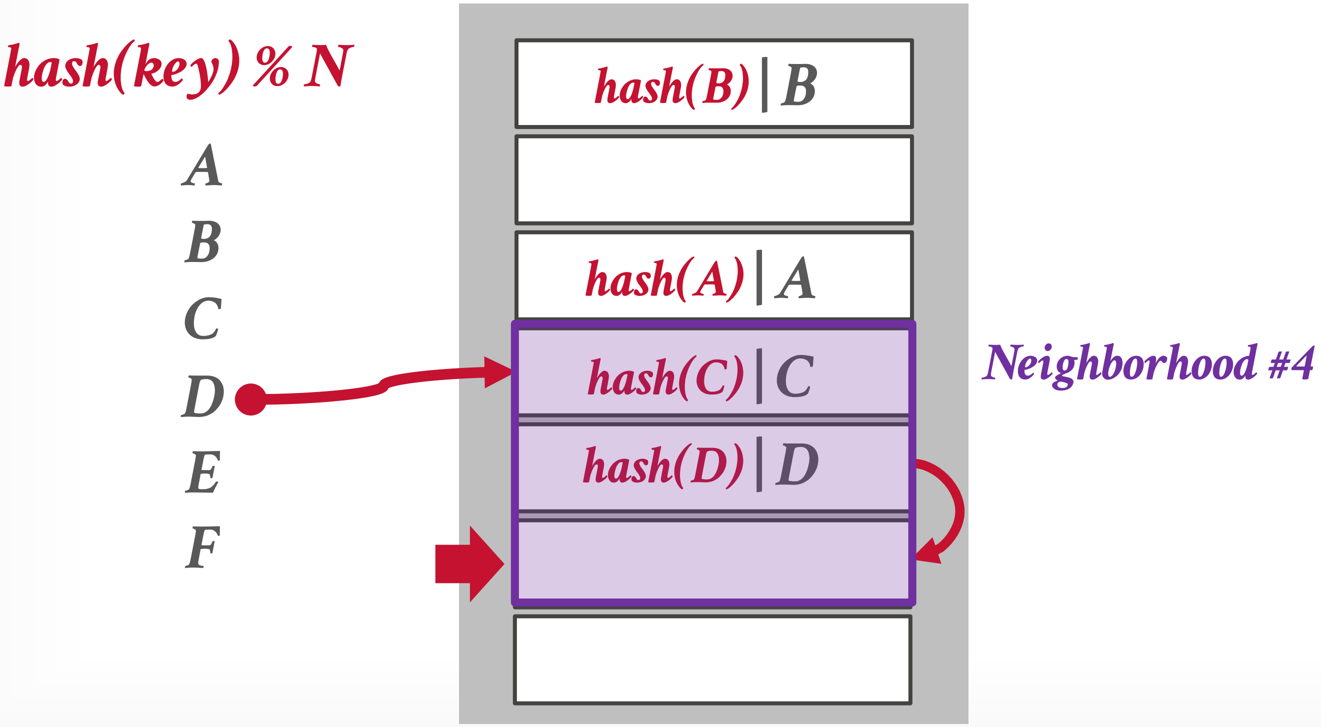
hash(key) % N

A
B
C
D
E
F



HOPSCOTCH HASHING

Neighborhood Size = 3

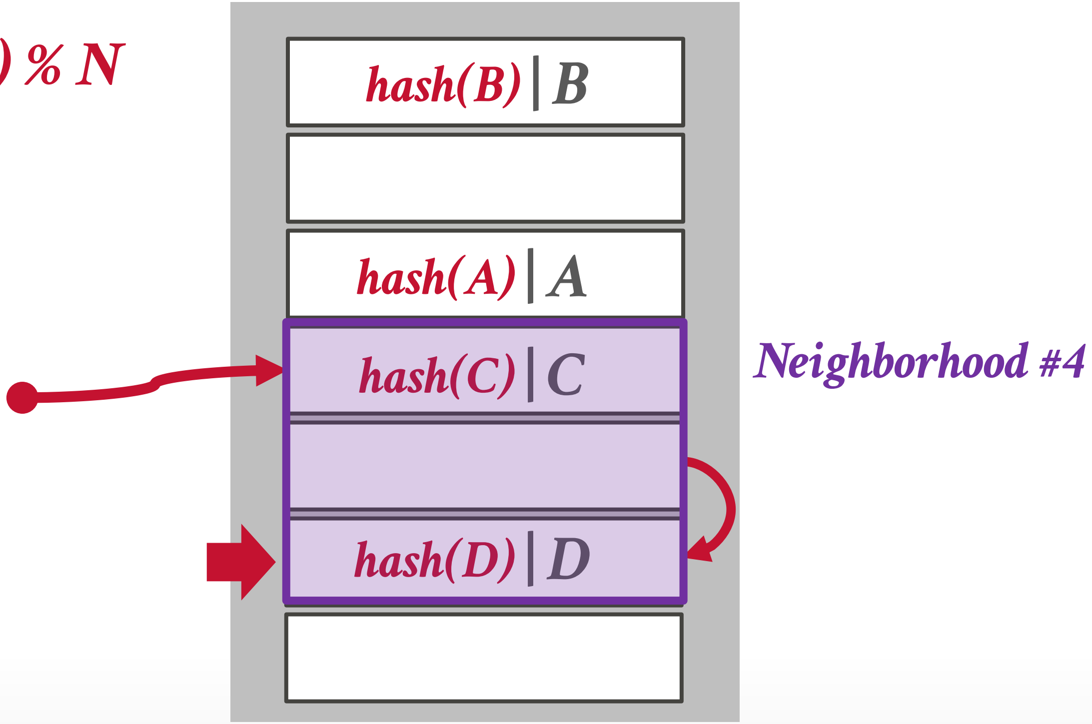


HOPSCOTCH HASHING

Neighborhood Size = 3

hash(key) % N

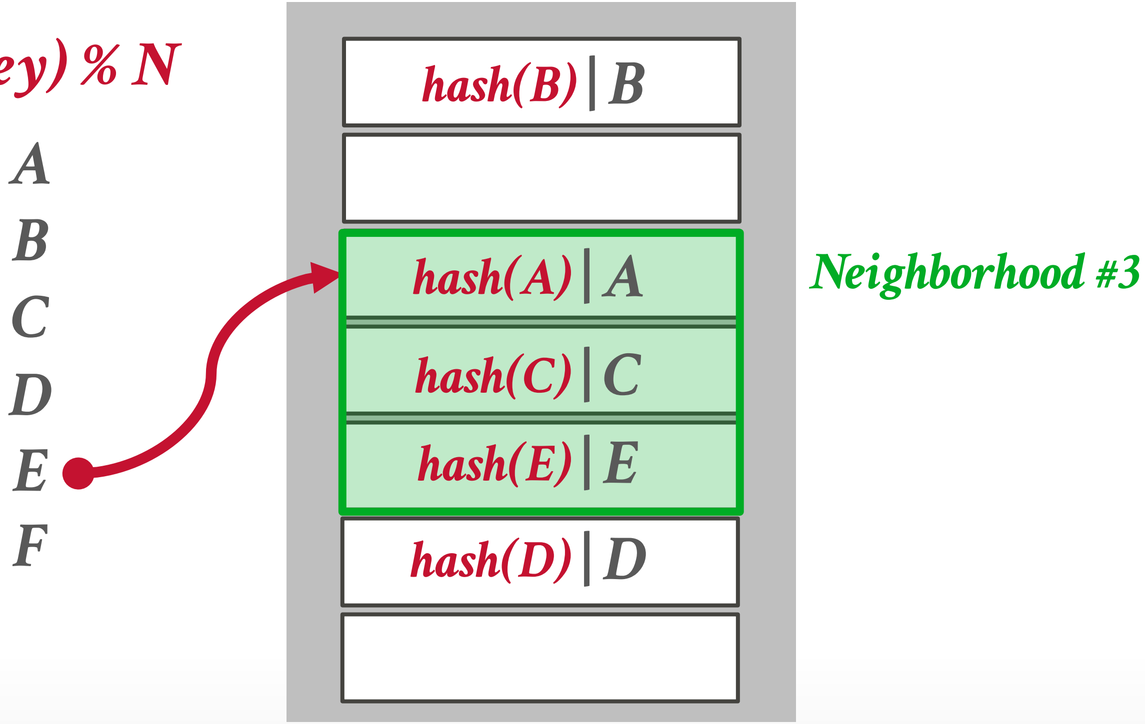
A
B
C
D
E
F



HOPSCOTCH HASHING

Neighborhood Size = 3

hash(key) % N

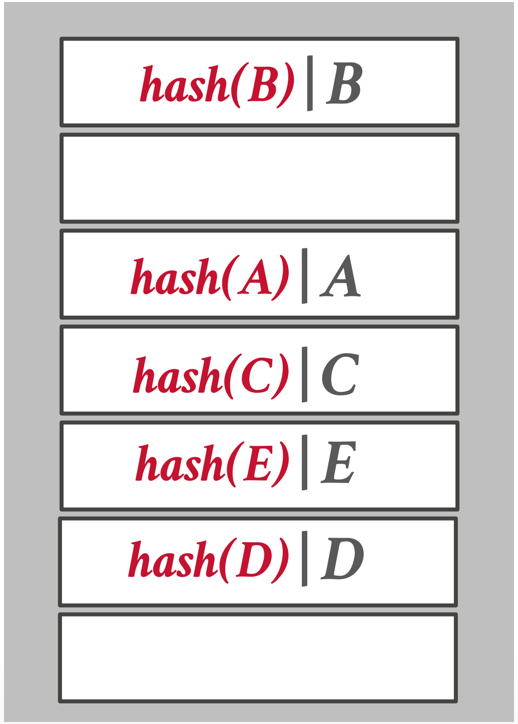


HOPSCOTCH HASHING

Neighborhood Size = 3

hash(key) % N

A
B
C
D
E
F

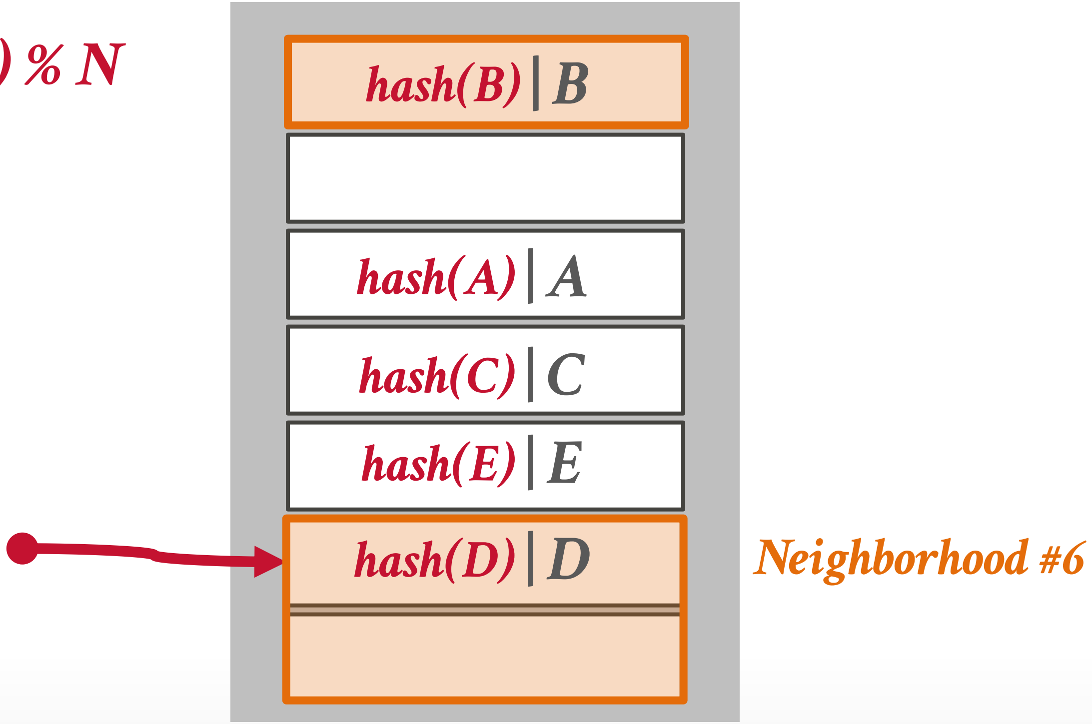


HOPSCOTCH HASHING

Neighborhood Size = 3

hash(key) % N

A
B
C
D
E
F

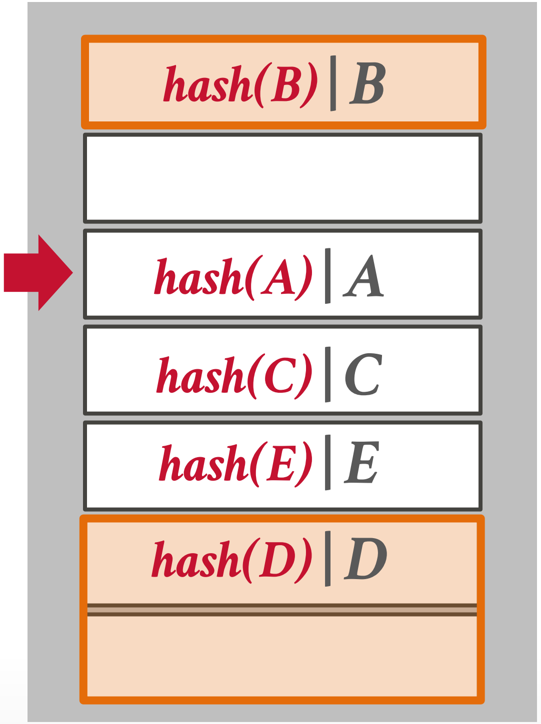


HOPSCOTCH HASHING

Neighborhood Size = 3

hash(key) % N

A
B
C
D
E
F



Neighborhood #6

HOPSCOTCH HASHING

Neighborhood Size = 3

hash(key) % N

A
B
C
D
E
F

hash(B) | B
hash(A) | A
hash(C) | C
hash(E) | E
hash(D) | D
hash(F) | 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.

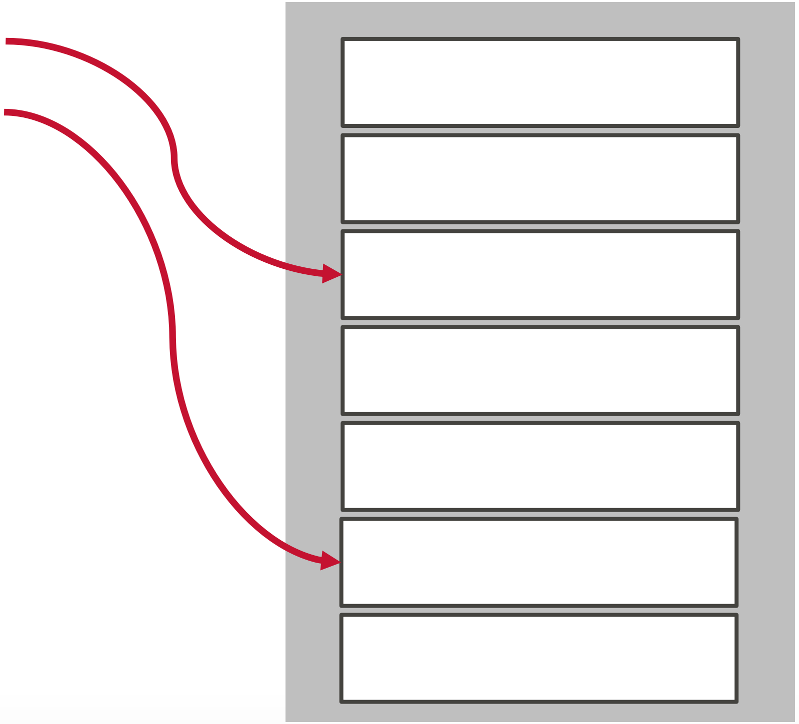
→ If no table has a free slot, evict the element from one of them and then re-hash it find a new location.

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

CUCKOO HASHING

Put A: $hash_1(A)$

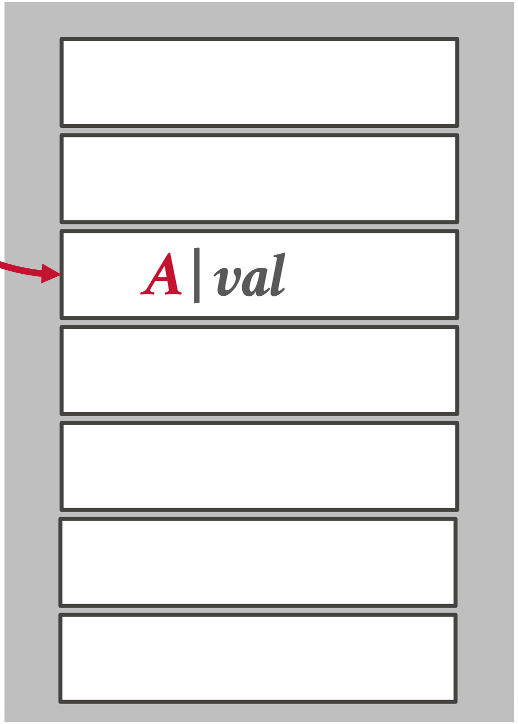
$hash_2(A)$



CUCKOO HASHING

Put A: $hash_1(A)$

$hash_2(A)$



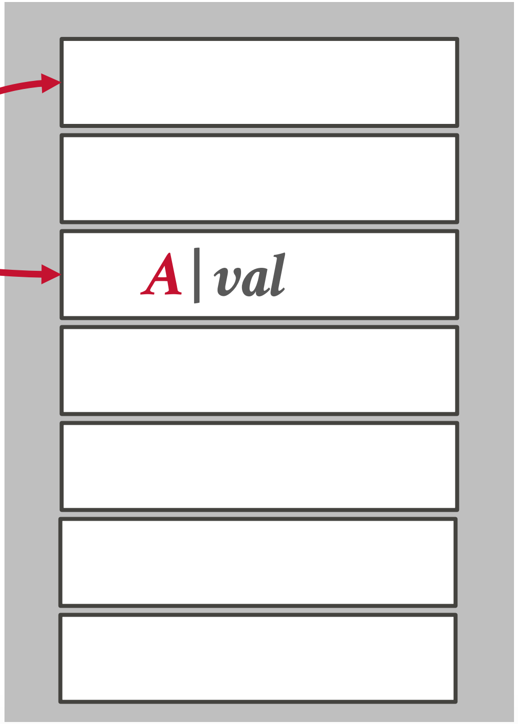
CUCKOO HASHING

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: $hash_1(B)$

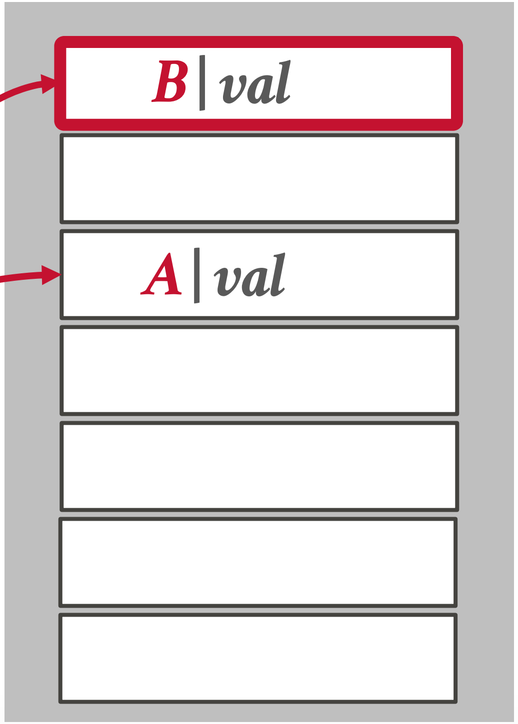
$hash_2(B)$

CUCKOO HASHING

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

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

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

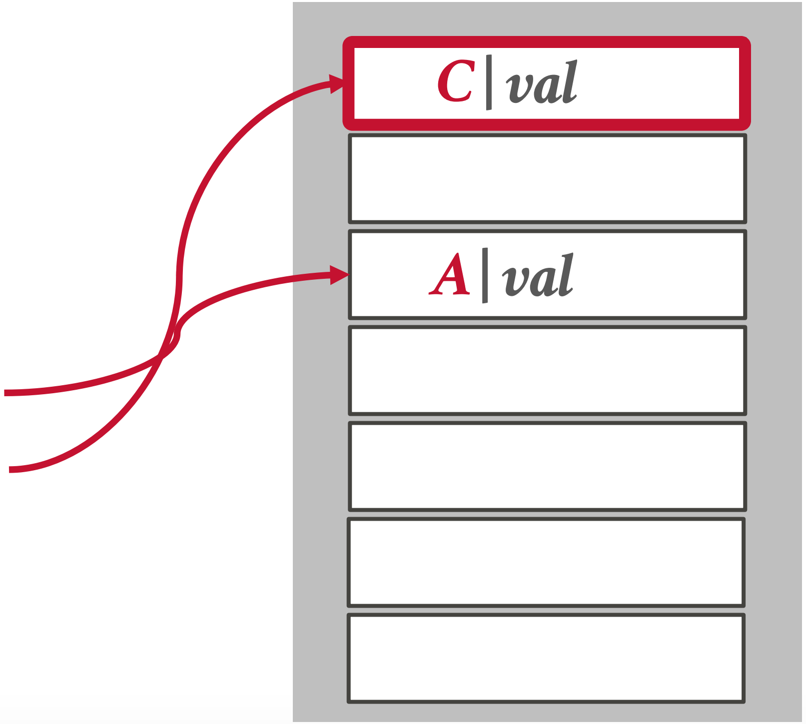


CUCKOO HASHING

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

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

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

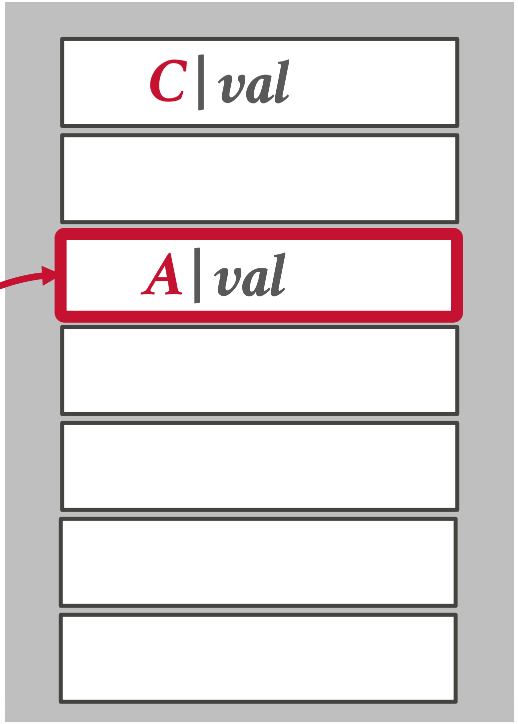


CUCKOO HASHING

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

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

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

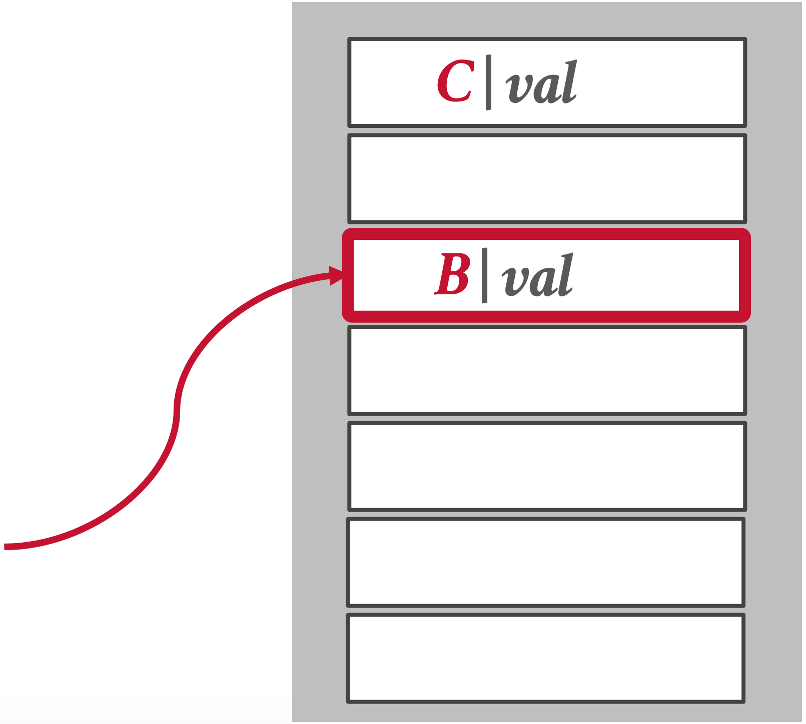


CUCKOO HASHING

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

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

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

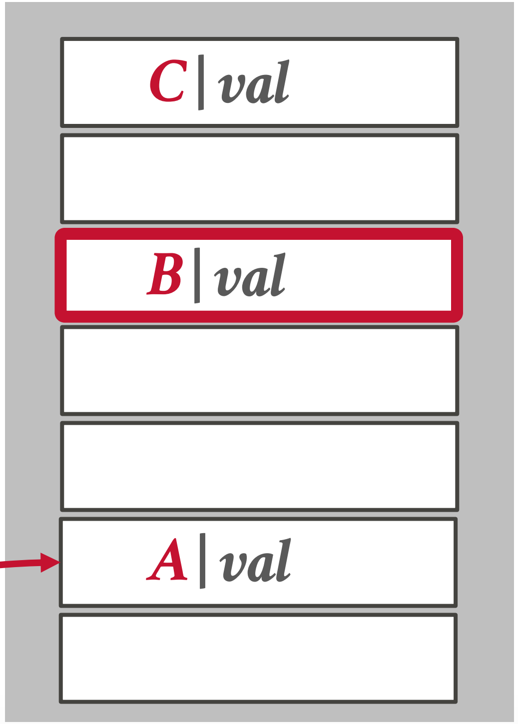


CUCKOO HASHING

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: $hash_1(C)$

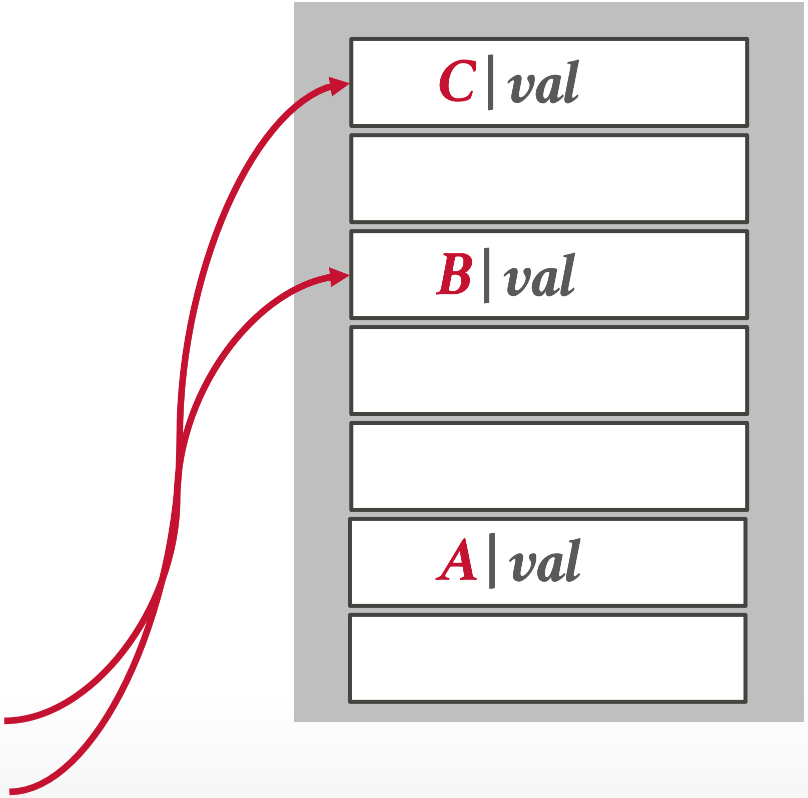
$hash_2(C)$

$hash_1(B)$

$hash_2(A)$

Get B: $hash_1(B)$

$hash_2(B)$



HASH TABLE CONTENTS

Tuple Data vs. Pointers/Offsets to Data

→ Whether to store the tuple directly inside of the hash table.
→ Storing tuples inside of the table not possible in open-addressing if there is variable length data.

Join Keys-only vs. Join Keys + Hashes

→ Whether to only store the original join key(s) in the hash table or also include the computed hashed key.
→ Classic compute vs. storage trade-off.

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.

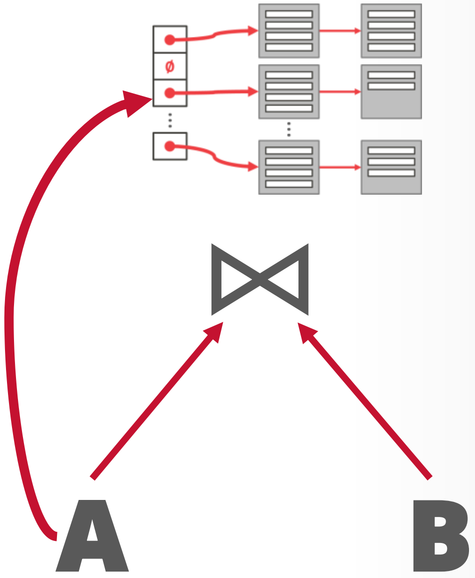
→ Otherwise, synchronize their access to the cursor on **S**.

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.

This enhancement is sometimes called **sideways information passing**.

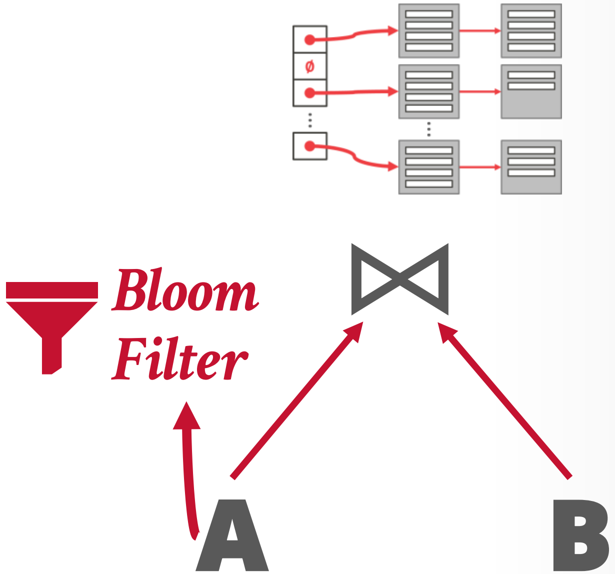


MICRO ADAPTIVITY IN VECTORWISE
SIGMOD 2013

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.



This enhancement is sometimes called **sideways information passing**.

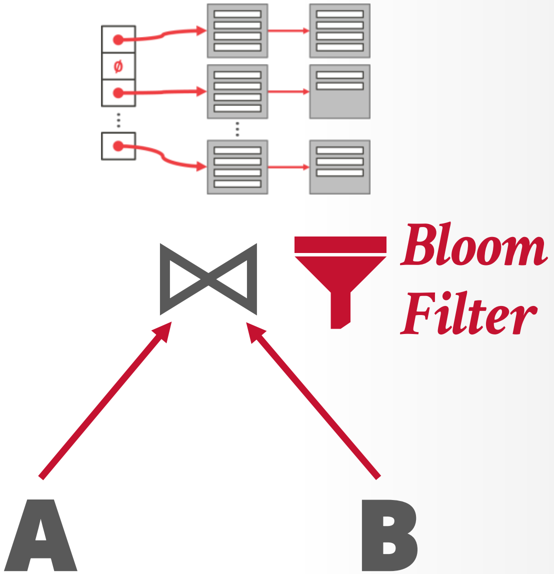
MICRO ADAPTIVITY IN VECTORWISE
SIGMOD 2013

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.

This enhancement is sometimes called **sideways information passing**.



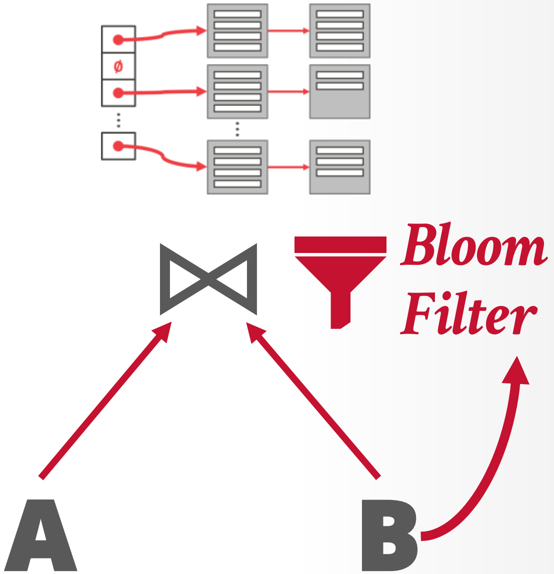
MICRO ADAPTIVITY IN VECTORWISE
SIGMOD 2013

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.

This enhancement is sometimes called **sideways information passing**.



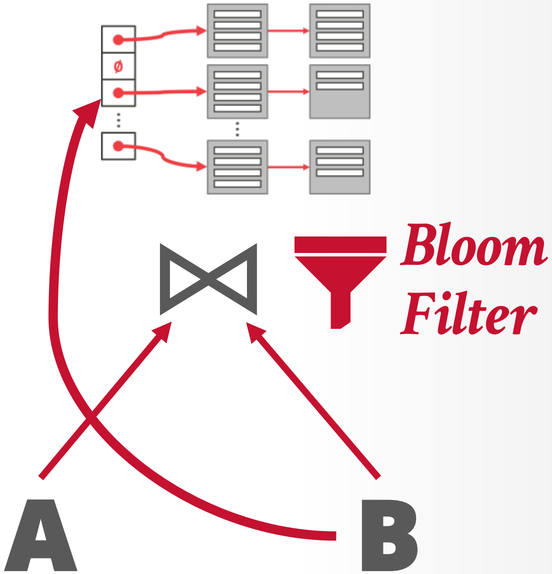
MICRO ADAPTIVITY IN VECTORWISE
SIGMOD 2013

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.

This enhancement is sometimes called **sideways information passing**.



MICRO ADAPTIVITY IN VECTORWISE
SIGMOD 2013

BENCHMARKS

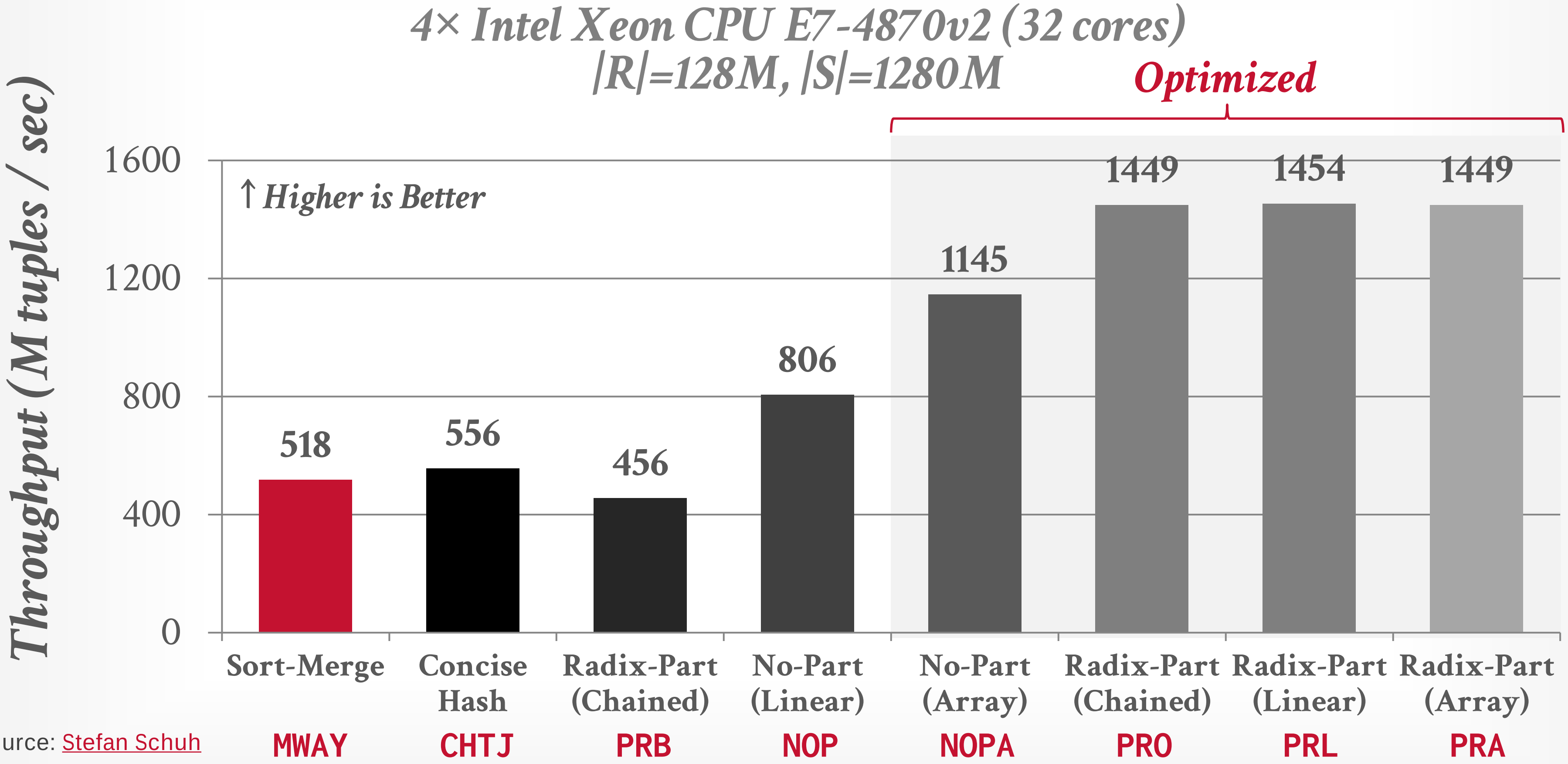
Implemented multiple variants of hash join algorithms based on previous literature and compare unoptimized vs. optimized versions.

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

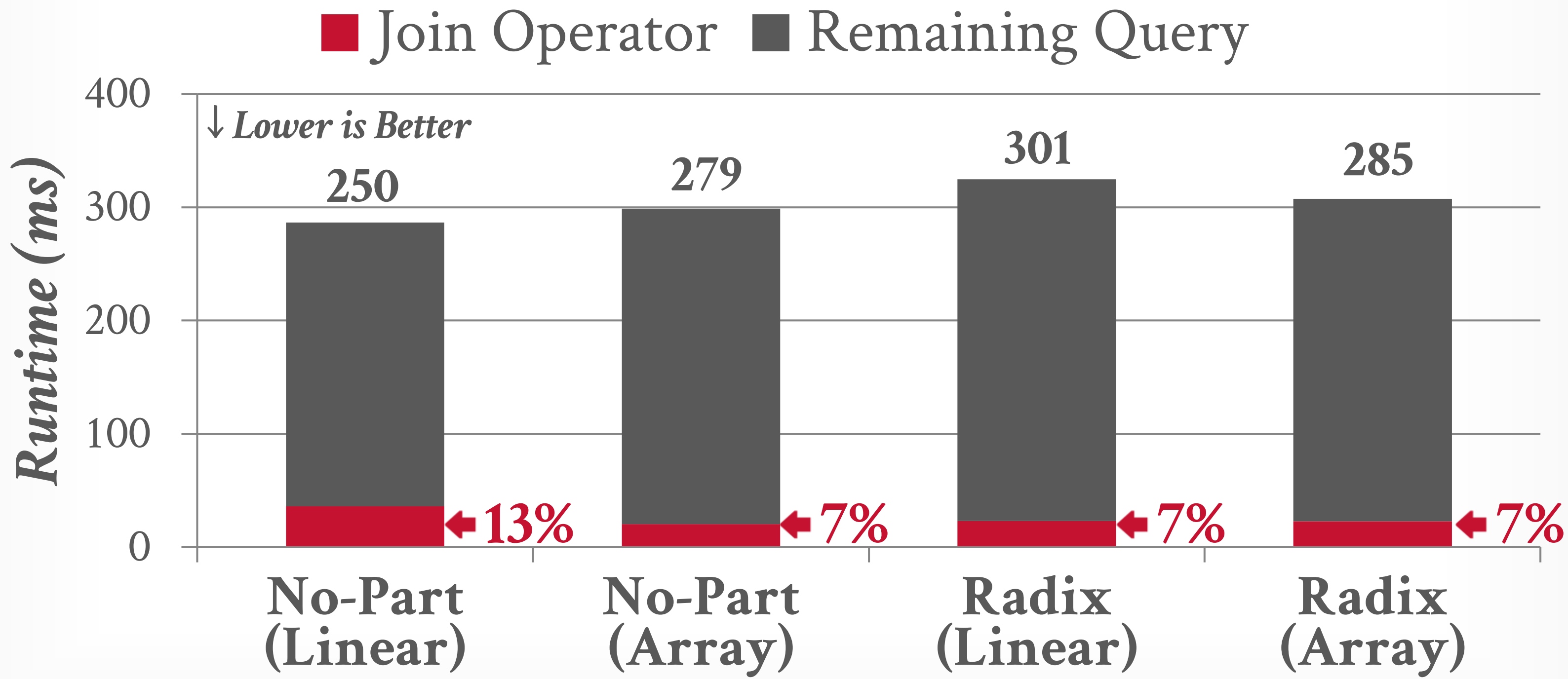
JOIN COMPARISON (R⋈S)



Source: Stefan Schuh

TPC-H Q19

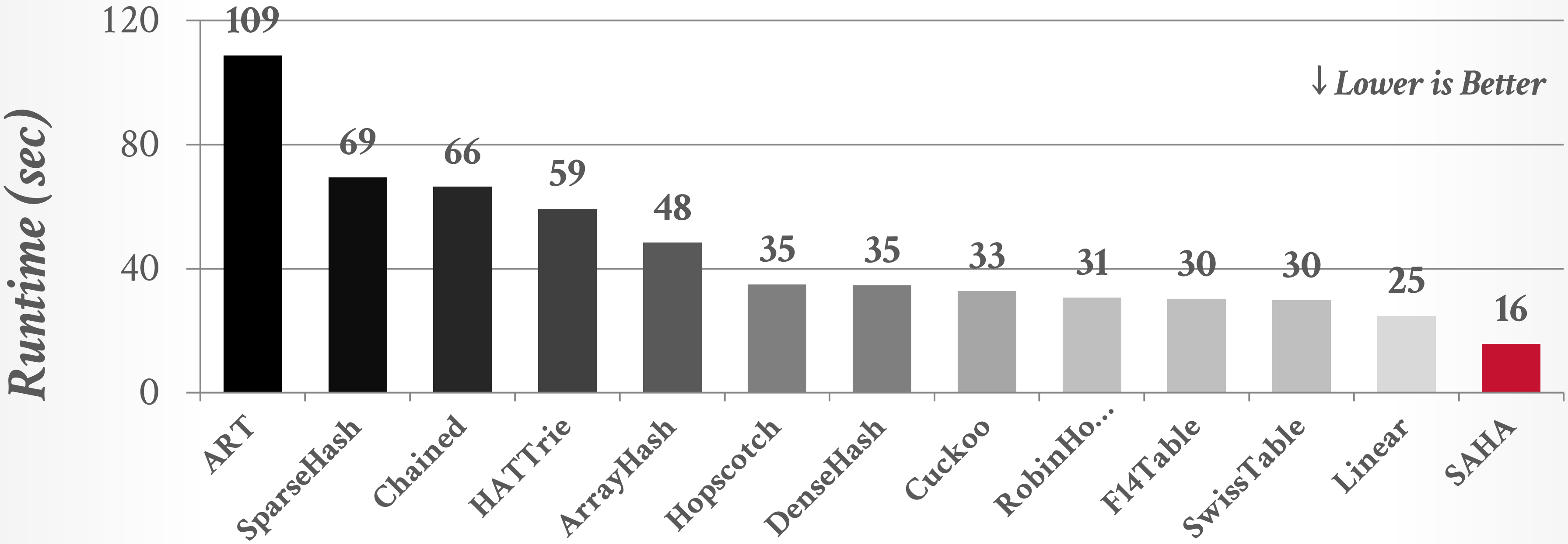
4× Intel Xeon CPU E7-4870v4 (32 cores)
Scale Factor 100



Source: Stefan Schuh

STRING HASH TABLES

2× Intel Xeon CPU E5-2460v4 (10 cores)
Join + Group By Microbenchmark



SAHA: A STRING ADAPTIVE HASH TABLE FOR ANALYTICAL DATABASES
APPL. SCI. 2020

PARTING THOUGHTS

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

AFAIK, most DBMSs picks one hash join implementation and does not try to be adaptive.

NEXT CLASS

Worst-Case Optimal Joins (aka multi-way joins)

Profiling with CPU Performance Counters