Lecture #19

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Parallel Join Algorithms (Hashing)

@Andy_Pavlo // 15-721 // Spring 2018

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.

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

We won't discuss nested-loop joins…

OBSERVATION

Many OLTP DBMSs don’t implement hash join.

But an **index nested-loop join** with a small number of target tuples is more or less equivalent to a hash join.

HASHING VS. SORTING

1970s – Sorting

1980s – Hashing

1990s – Equivalent

2000s – Hashing

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

2020s – ???

PARALLEL JOIN ALGORITHMS

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 for real!

JOIN ALGORITHM DESIGN GOALS

Goal #1: Minimize Synchronization
→ Avoid taking latches during execution.

Goal #2: Minimize CPU Cache Misses
→ Ensure that data is always local to worker thread.

IMPROVING CACHE BEHAVIOR

Factors that affect cache misses in a DBMS:
→ Cache + TLB capacity.
→ Locality (temporal and spatial).

Non-Random Access (Scan):
→ Clustering 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

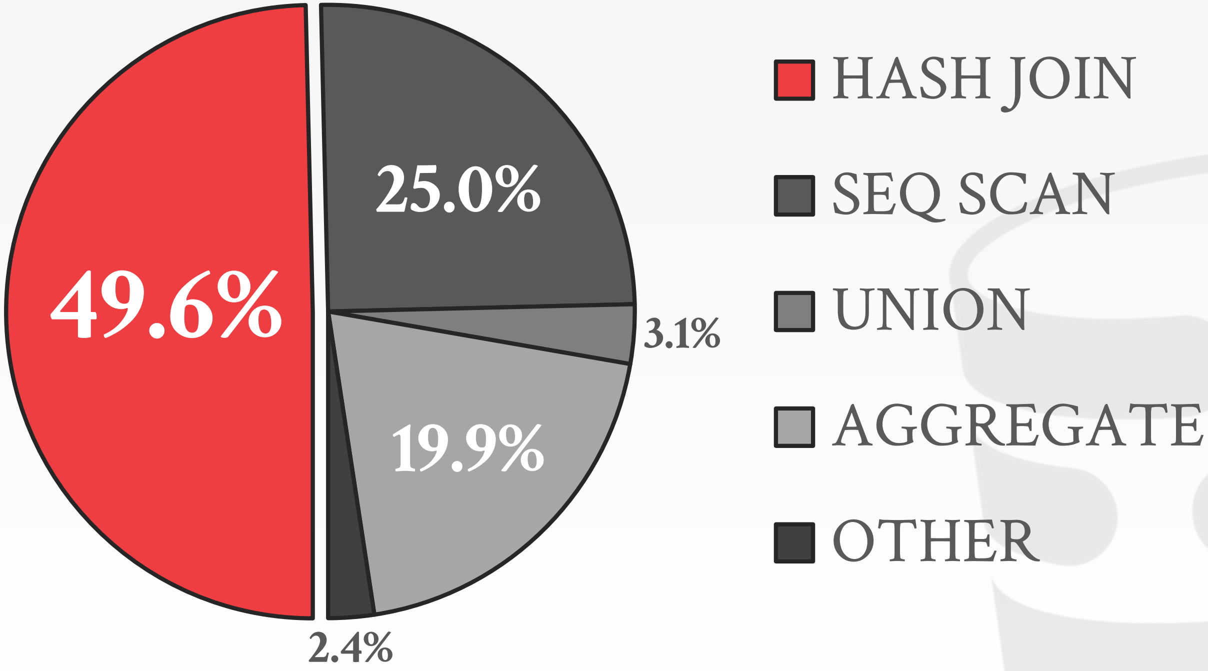
Hash join is the most important operator in a DBMS for OLAP workloads.

It's important that we speed it up by taking advantage of multiple cores.

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

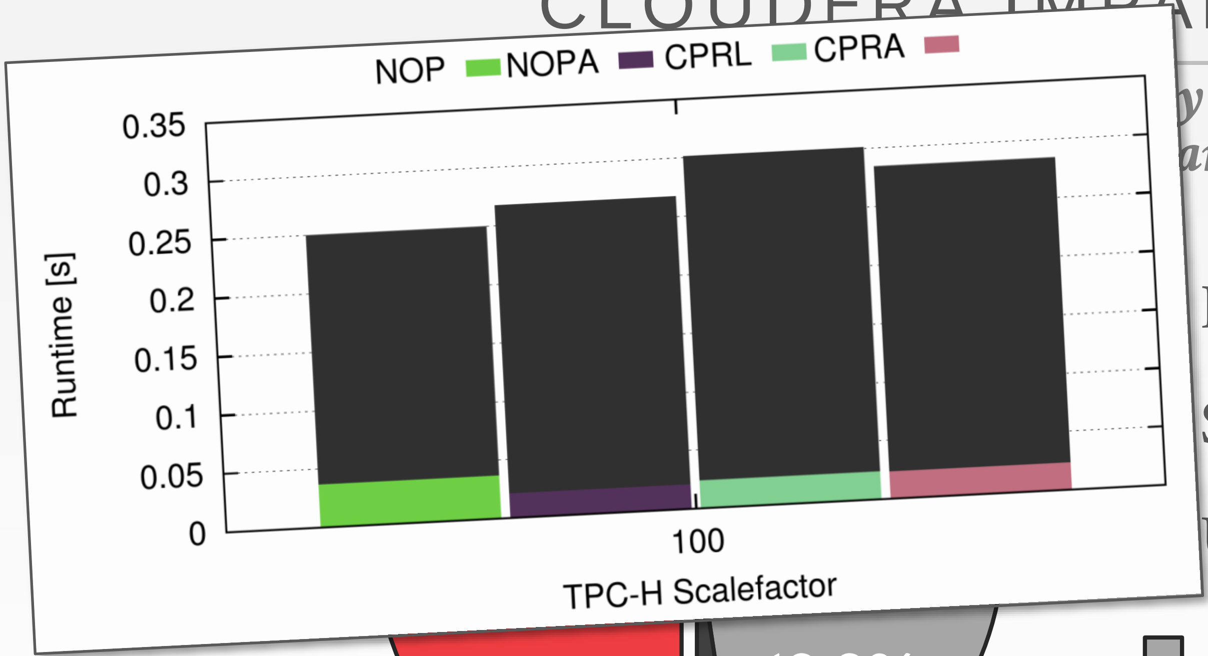
CLOUDERA IMPALA

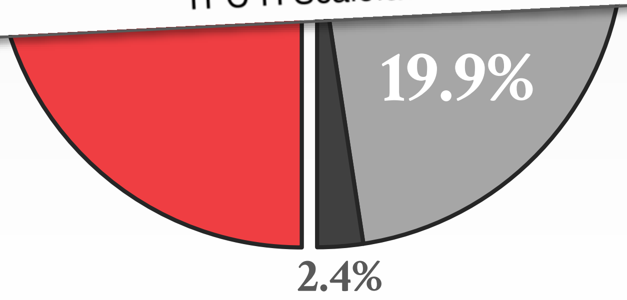
% of Total CPU Time Spent in Query Operators
Workload: TPC-H Benchmark



CLOUDERA IMPALA

y Operators
ark





- HASH JOIN
- SEQ SCAN
- UNION
- AGGREGATE
- OTHER

HASH JOIN (R⋈S)

Phase #1: Partition (*optional*)

→ Divide the tuples of **R** and **S** into sets using a hash 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.

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 ***hybrid hash join***.

Contents of buffers depends on storage model:

→ **NSM**: Either the entire tuple or a subset of attributes.

→ **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.

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

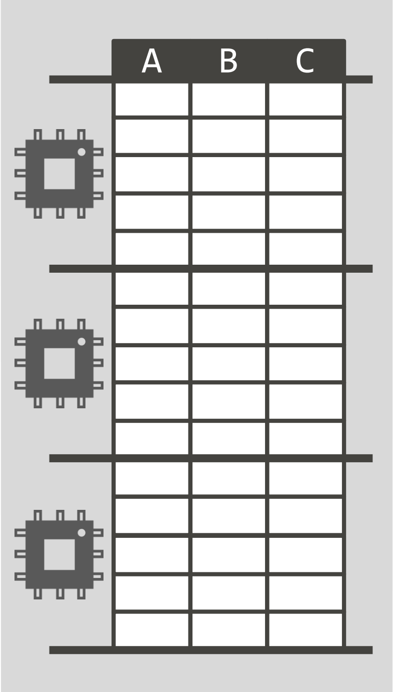
Scan the input relation only once and generate the output on-the-fly.

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

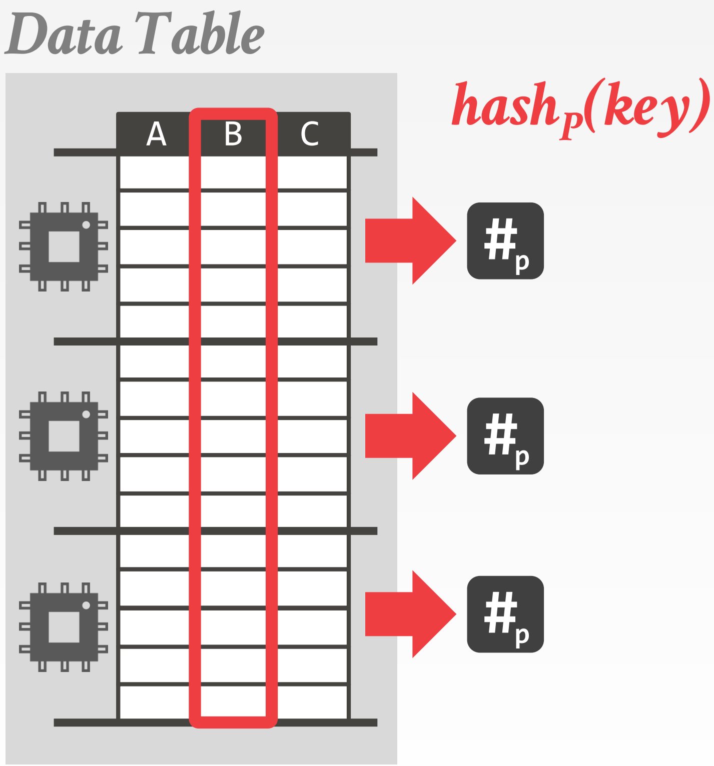
Approach #2: Private Partitions
→ Each thread has its own set of partitions.
→ Have to consolidate them after all threads finish.

SHARED PARTITIONS

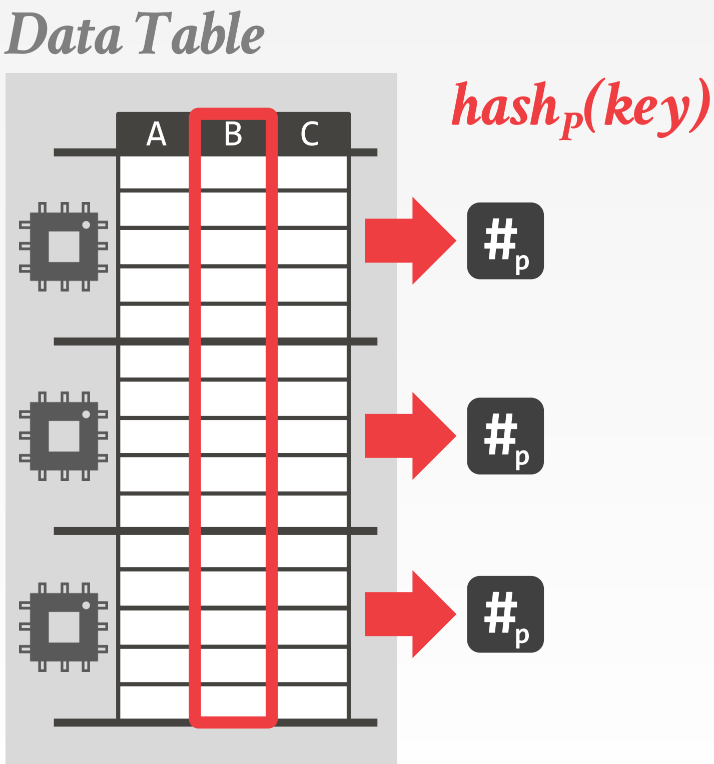
Data Table

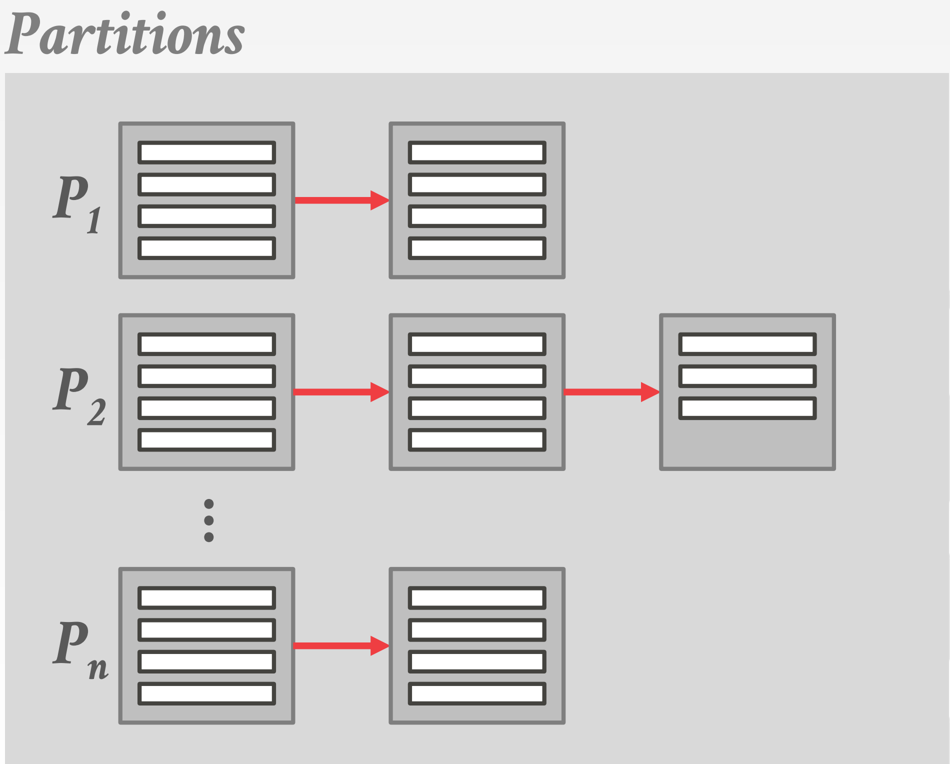


SHARED PARTITIONS

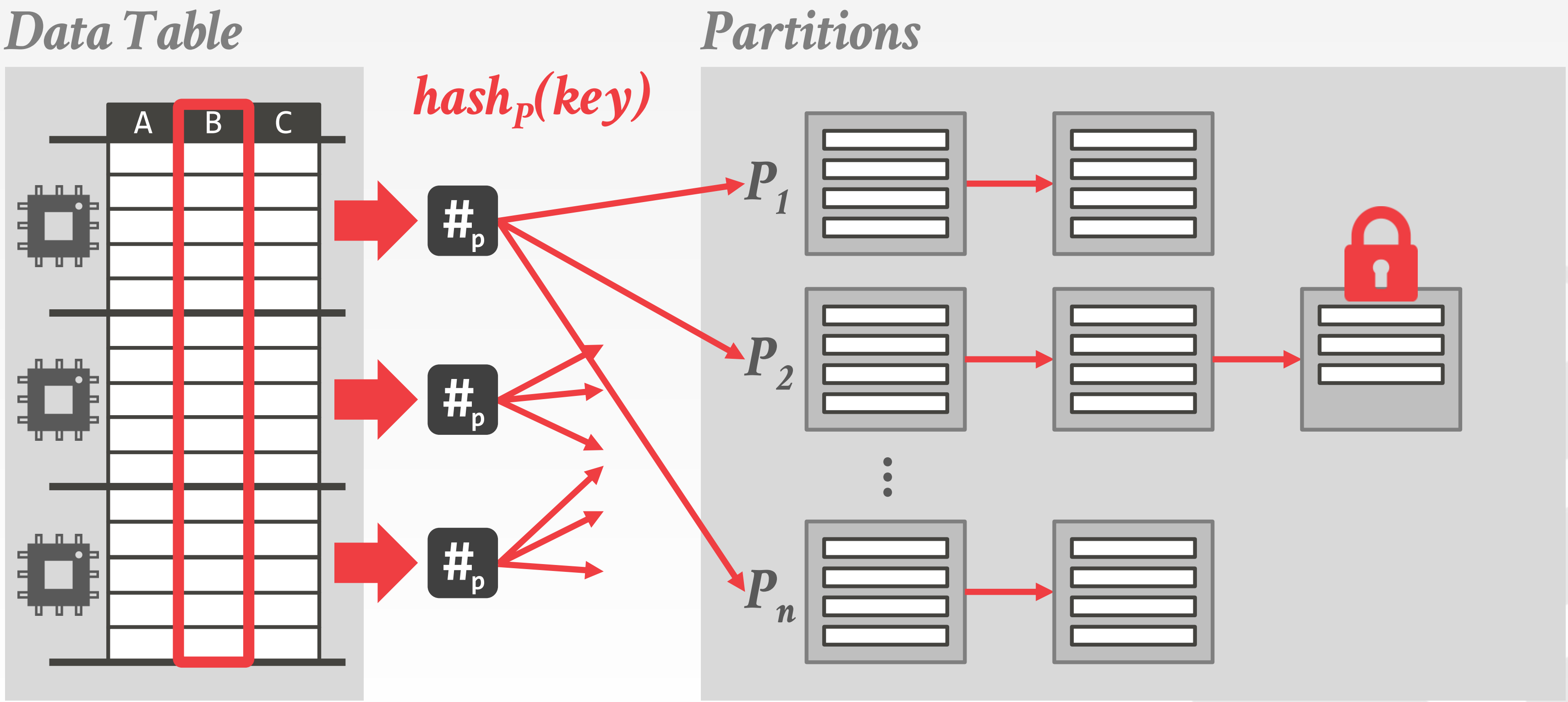


SHARED PARTITIONS

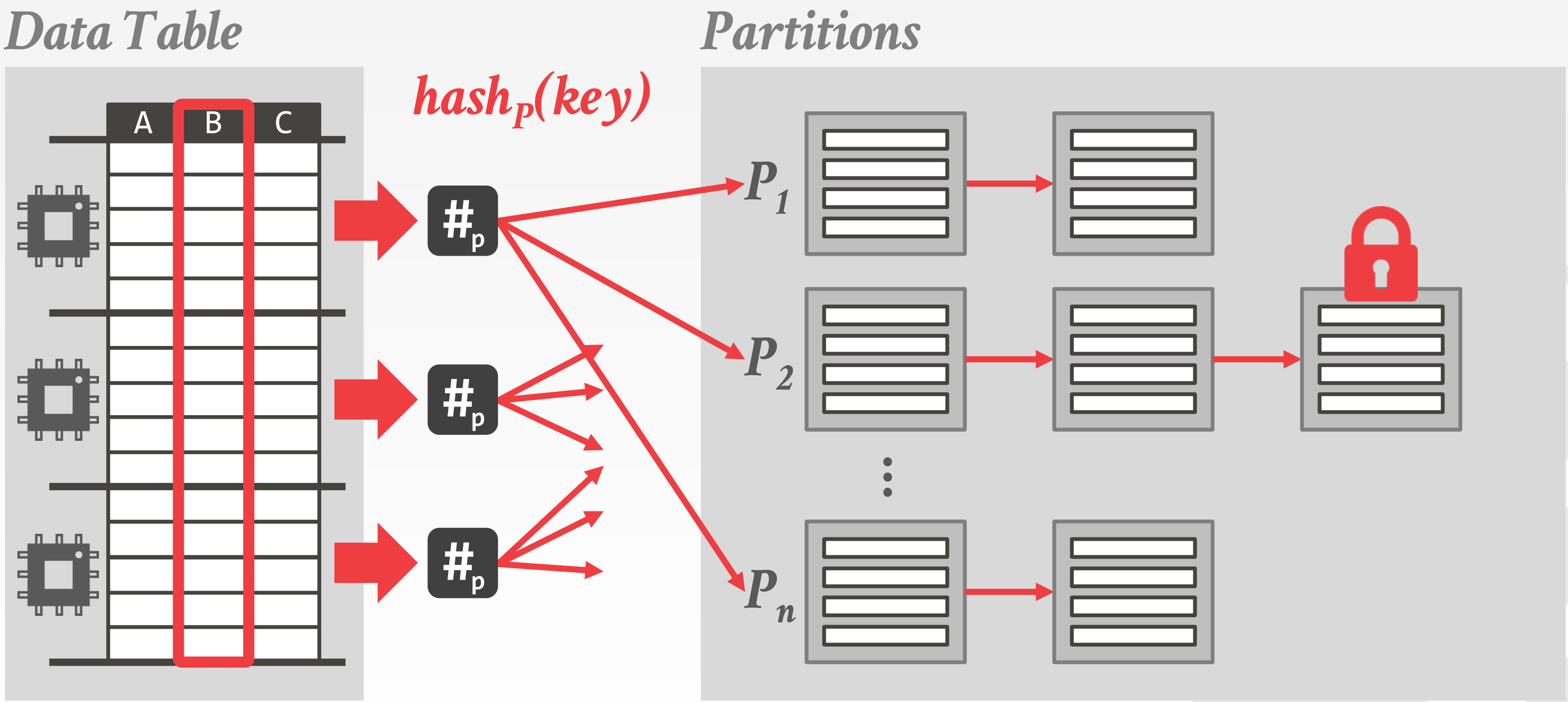




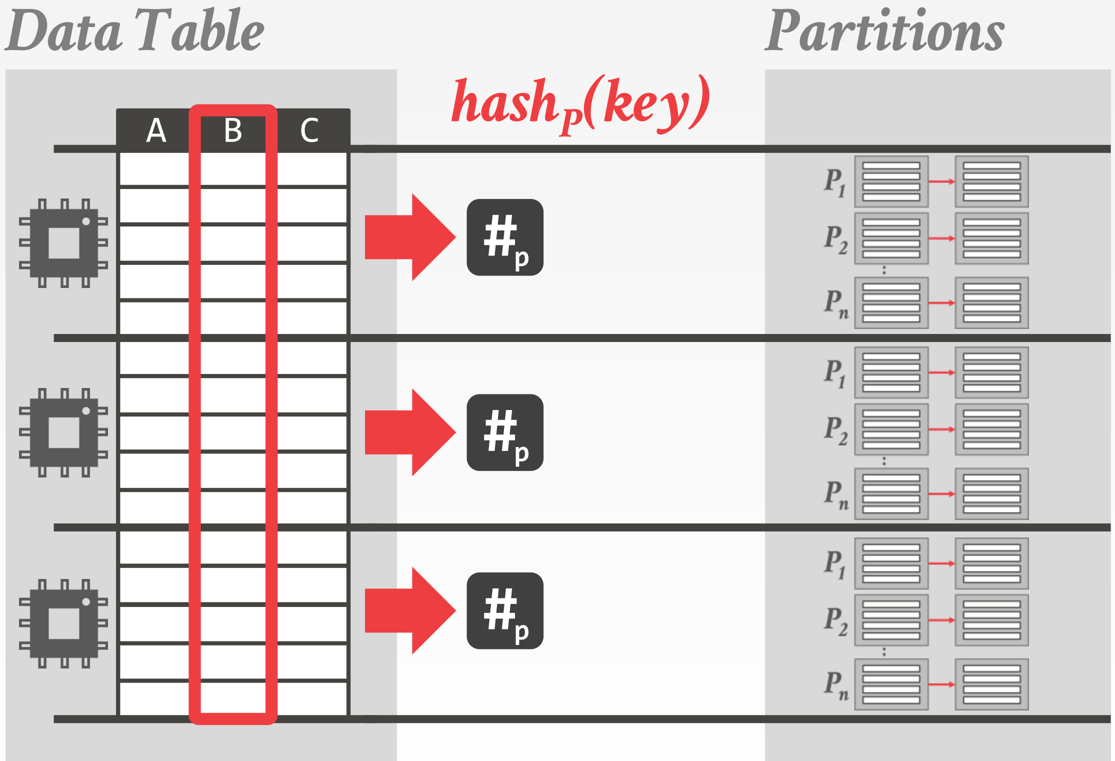
SHARED PARTITIONS



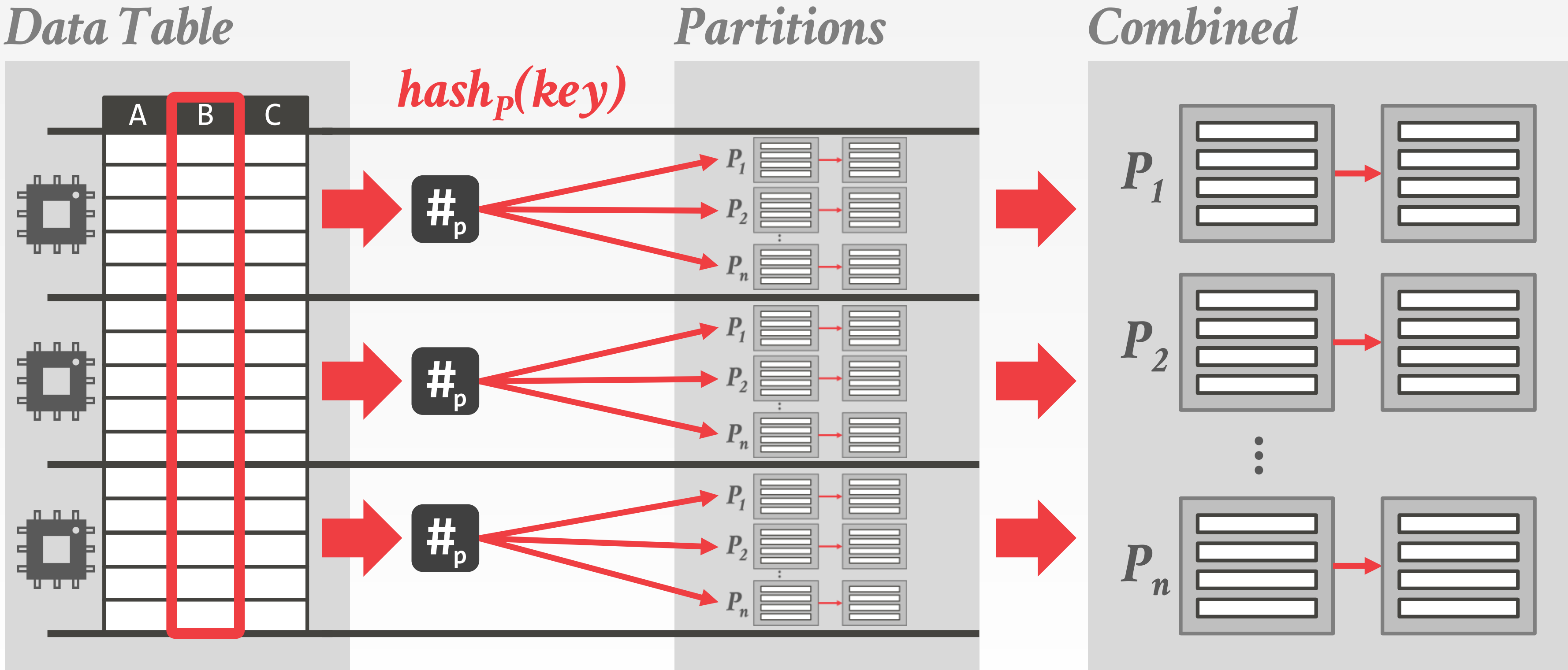
SHARED PARTITIONS



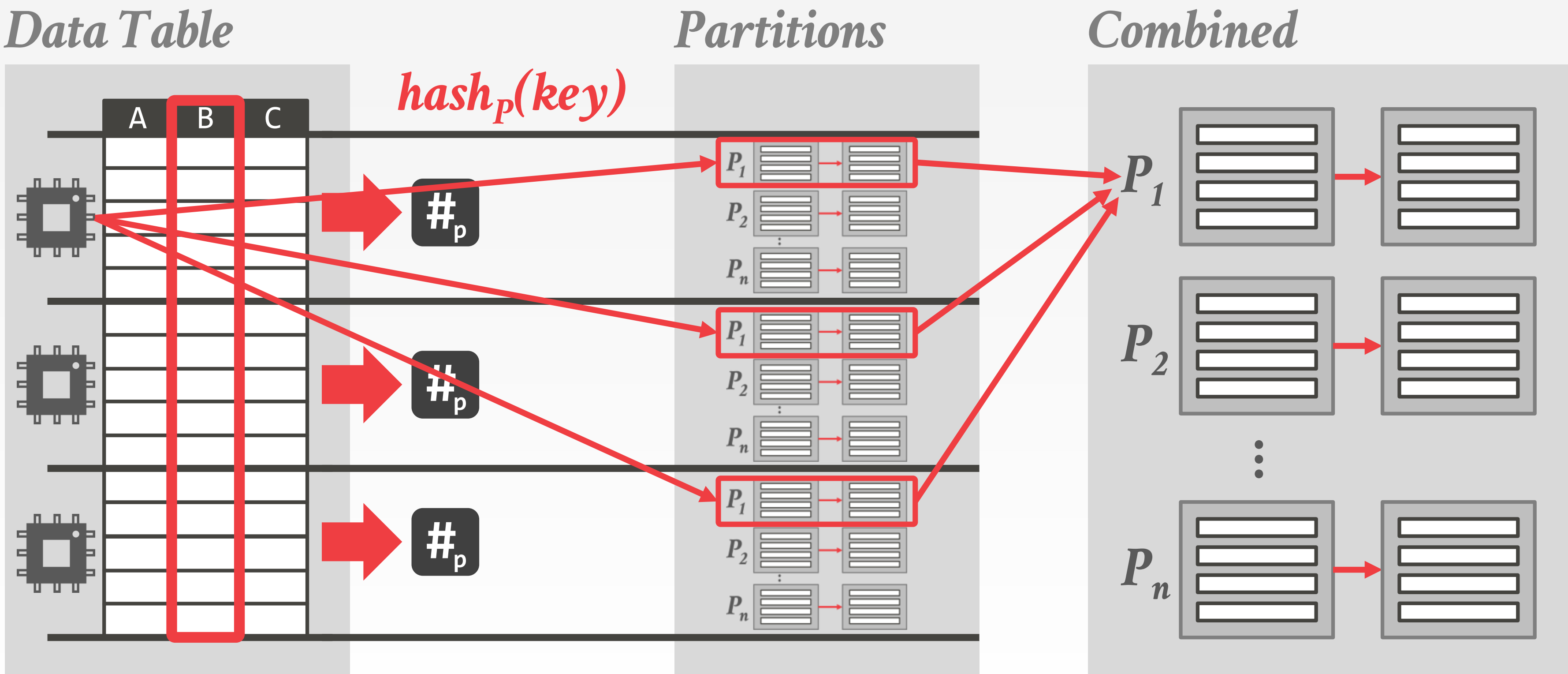
PRIVATE PARTITIONS



PRIVATE PARTITIONS



PRIVATE PARTITIONS



RADIX PARTITIONING

Scan the input relation multiple times to generate the partitions.

Multi-step pass over the relation:

→ **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 output offsets by computing the **prefix sum**.

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

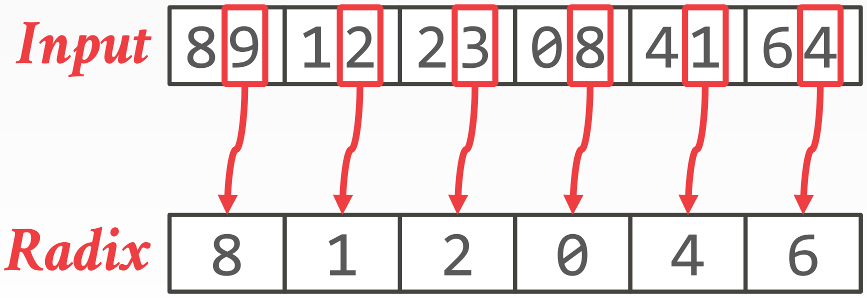
RADIX

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

Input	89	12	23	08	41	64
Radix	8	1	2	0	4	6

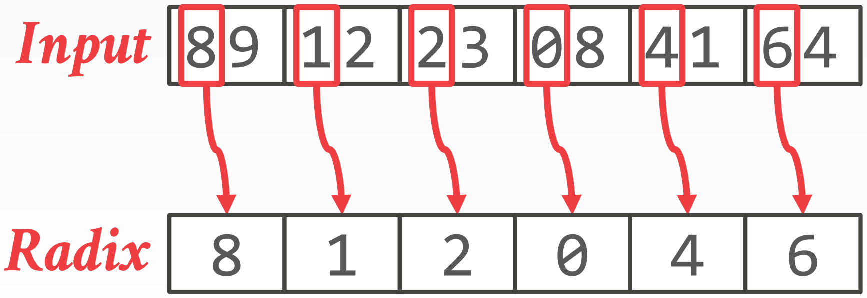
RADIX

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



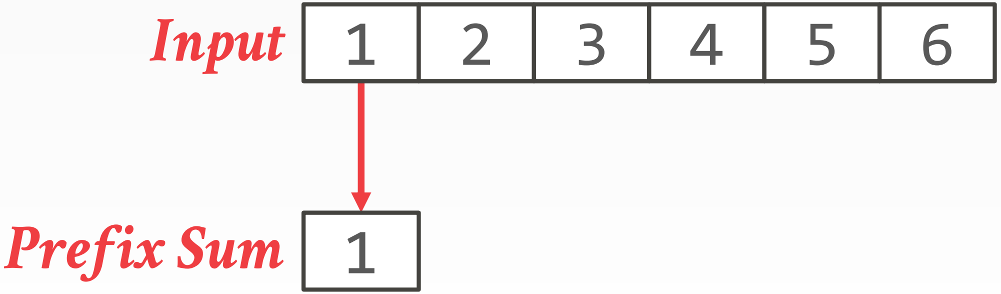
RADIX

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



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 SUM

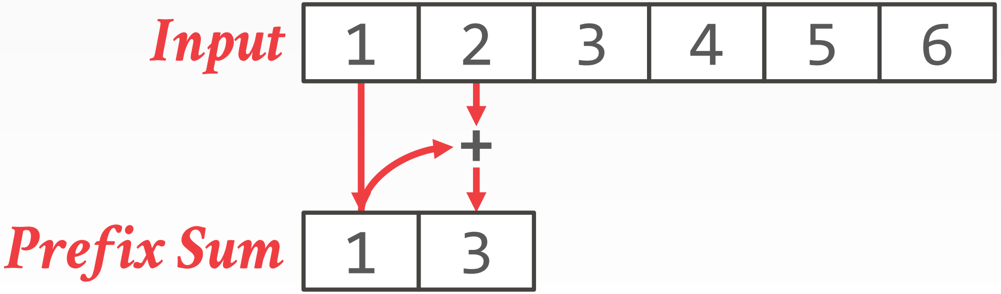
The prefix sum of a sequence of numbers

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

is a second sequence of numbers

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

that is a running total of the input sequence.



PREFIX SUM

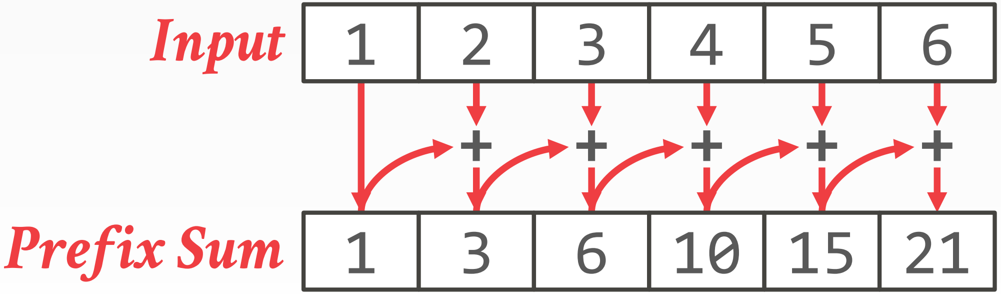
The prefix sum of a sequence of numbers

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

is a second sequence of numbers

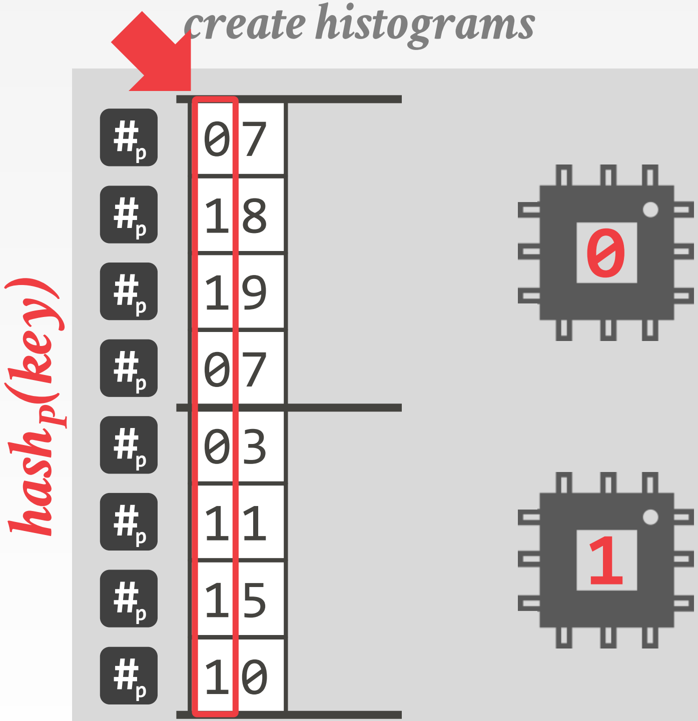
$(y_0, y1, \ldots, y_n)$

that is a running total of the input sequence.



RADIX PARTITIONS

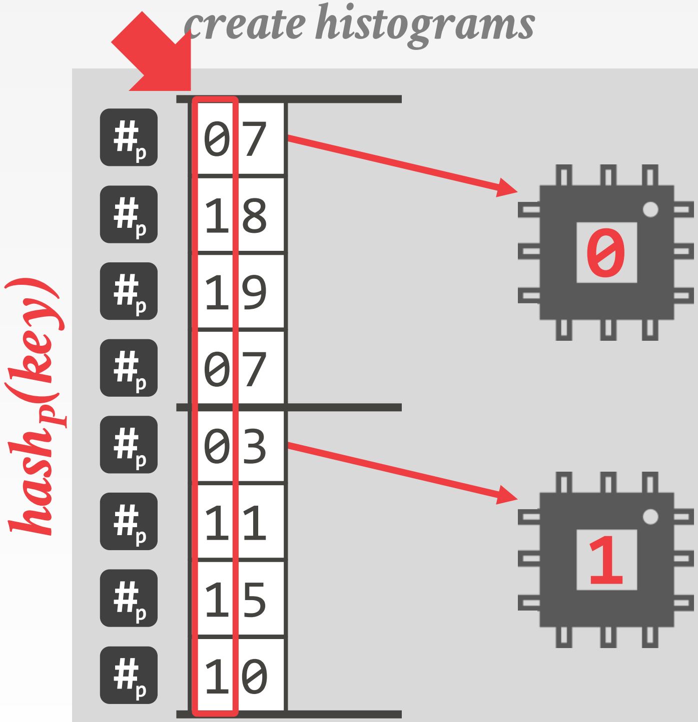
Step #1: Inspect input, create histograms



Source: Spyros Blanas

RADIX PARTITIONS

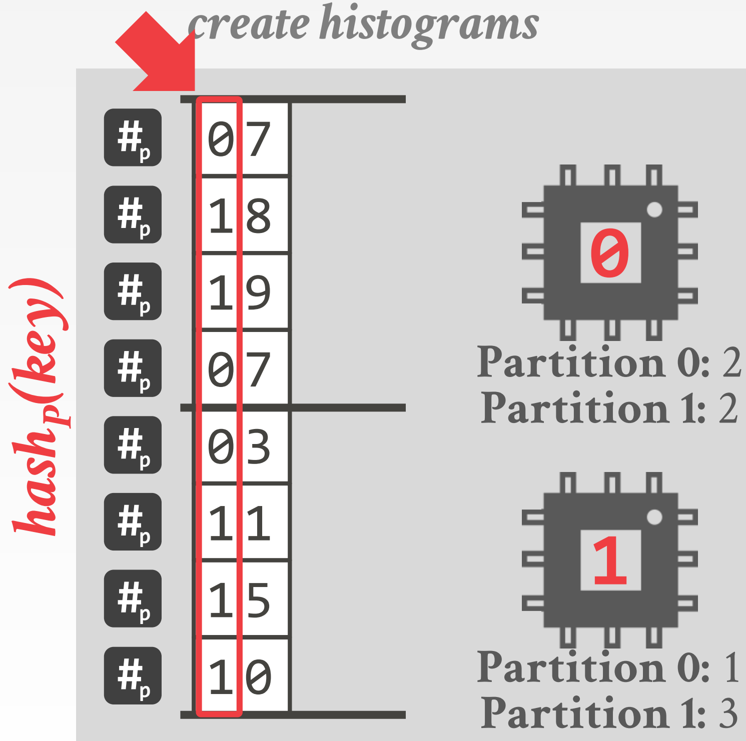
Step #1: Inspect input, create histograms



Source: Spyros Blanas

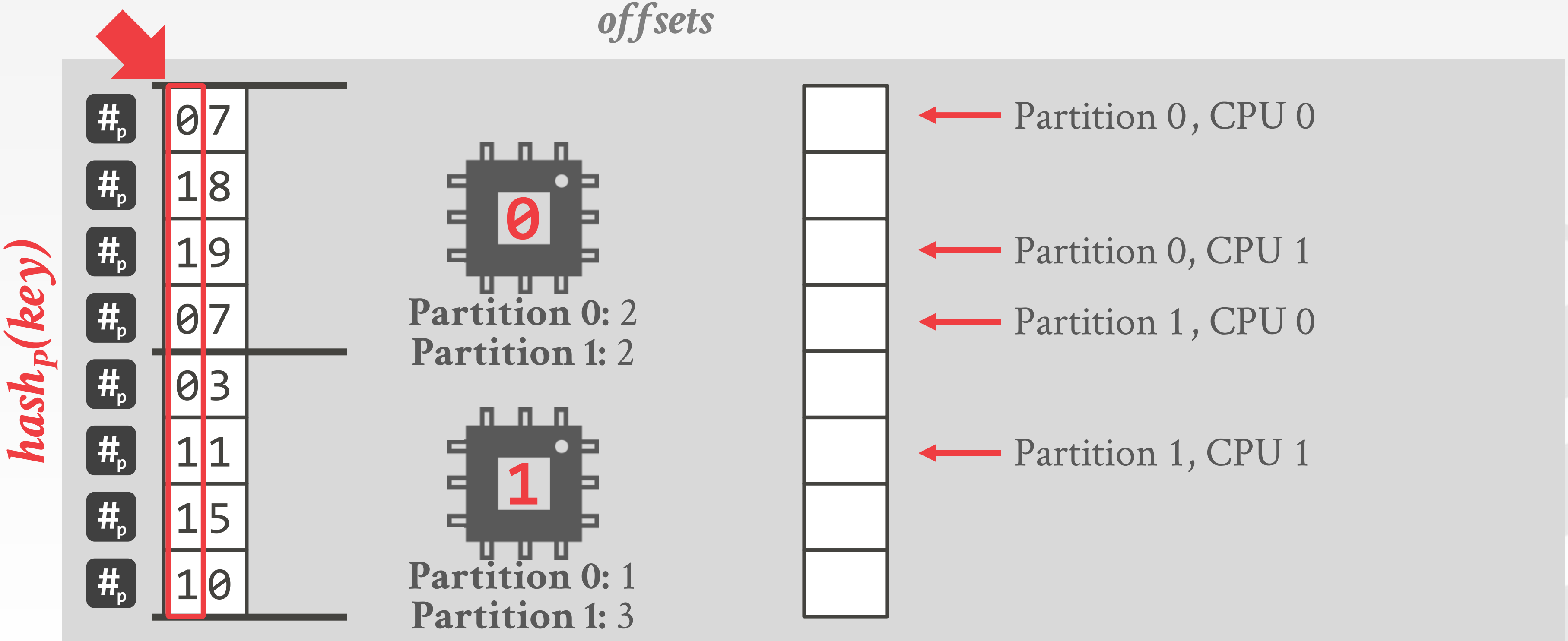
RADIX PARTITIONS

Step #1: Inspect input, create histograms



RADIX PARTITIONS

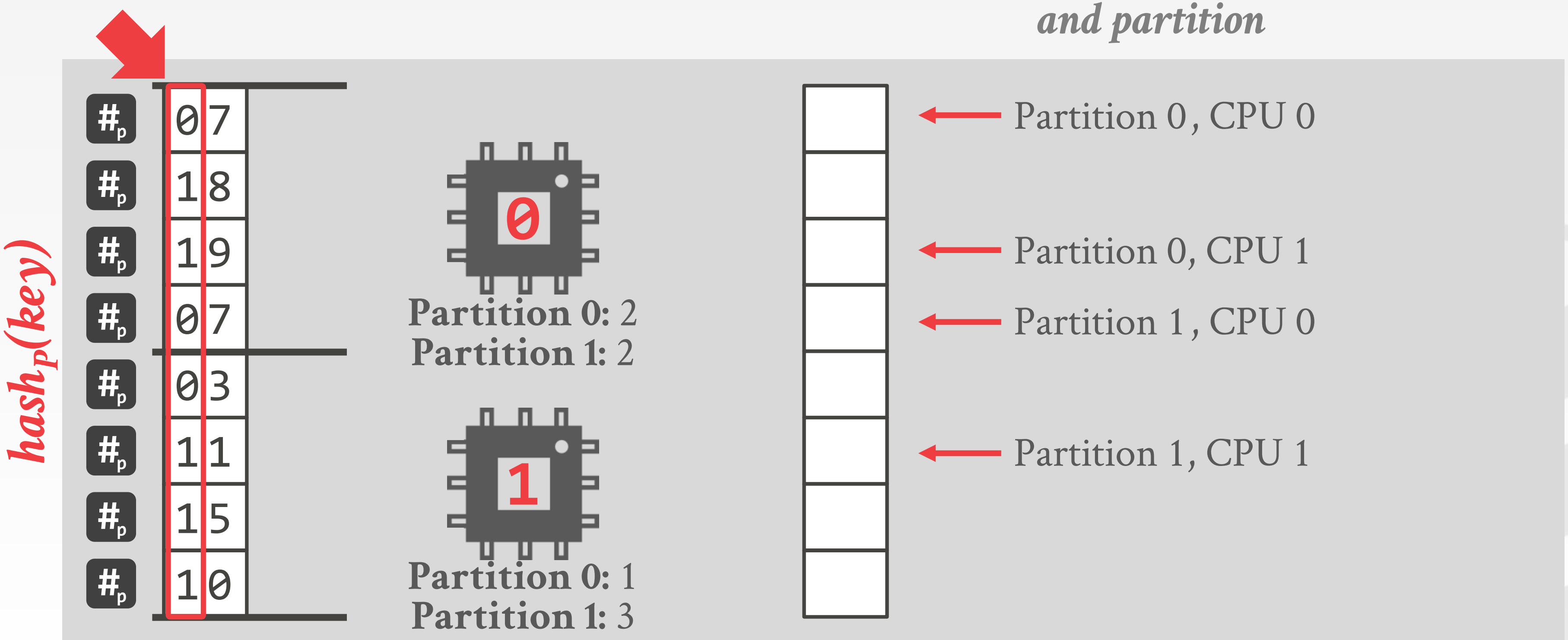
Step #2: Compute output offsets



Source: Spyros Blanas

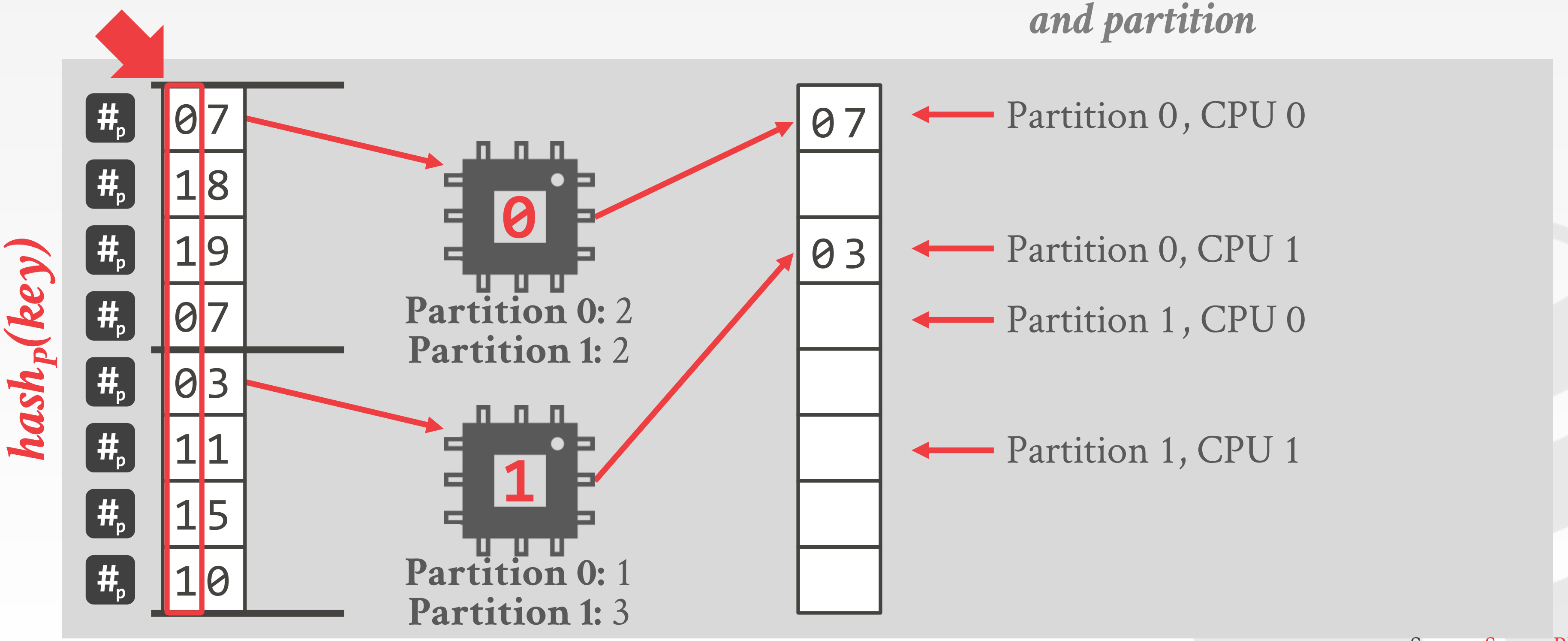
RADIX PARTITIONS

Step #3: Read input and partition



RADIX PARTITIONS

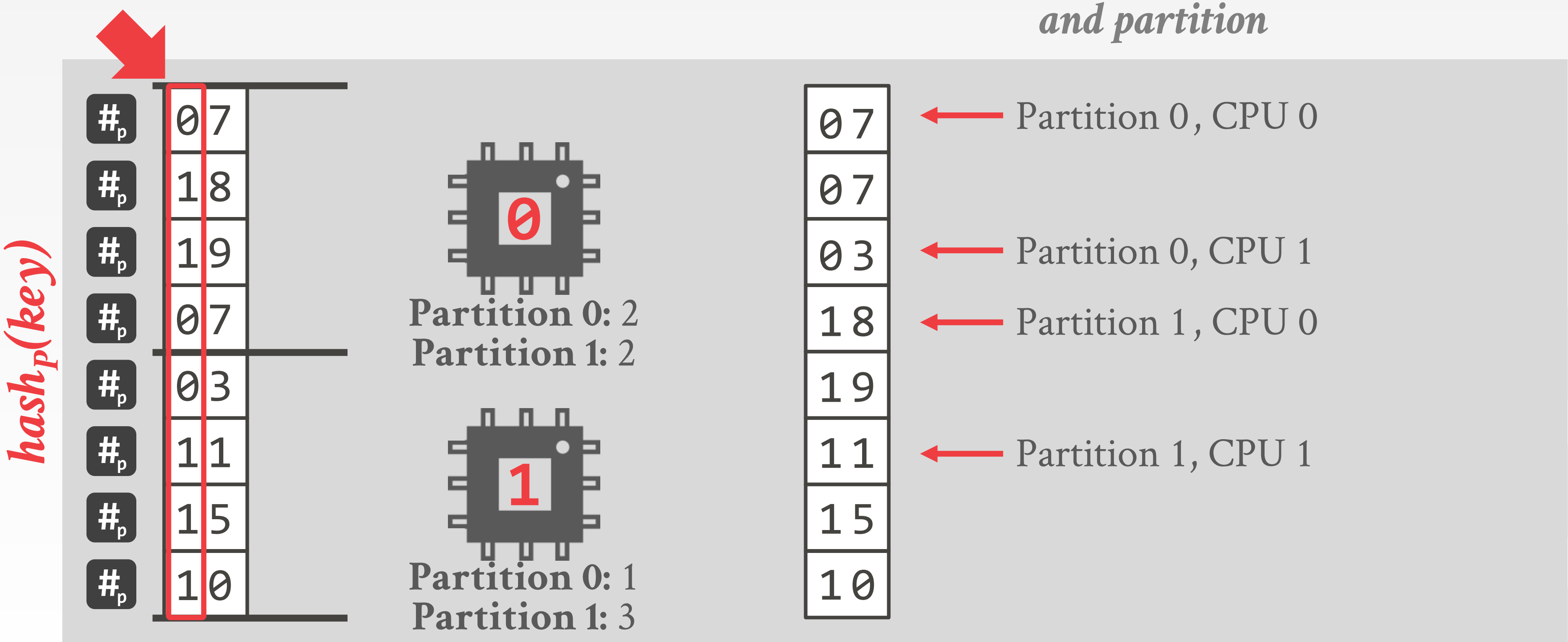
Step #3: Read input and partition



Source: Spyros Blanas

RADIX PARTITIONS

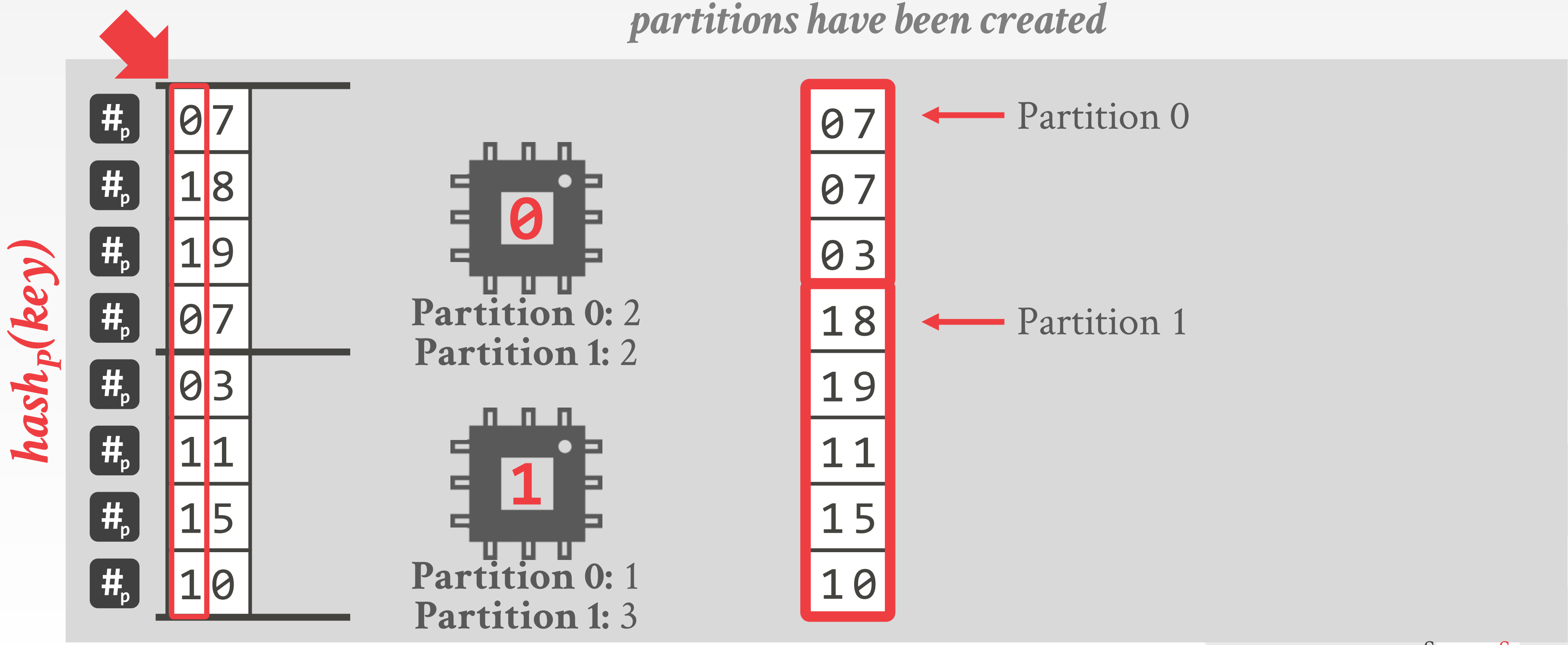
Step #3: Read input and partition



Source: Spyros Blanas

RADIX PARTITIONS

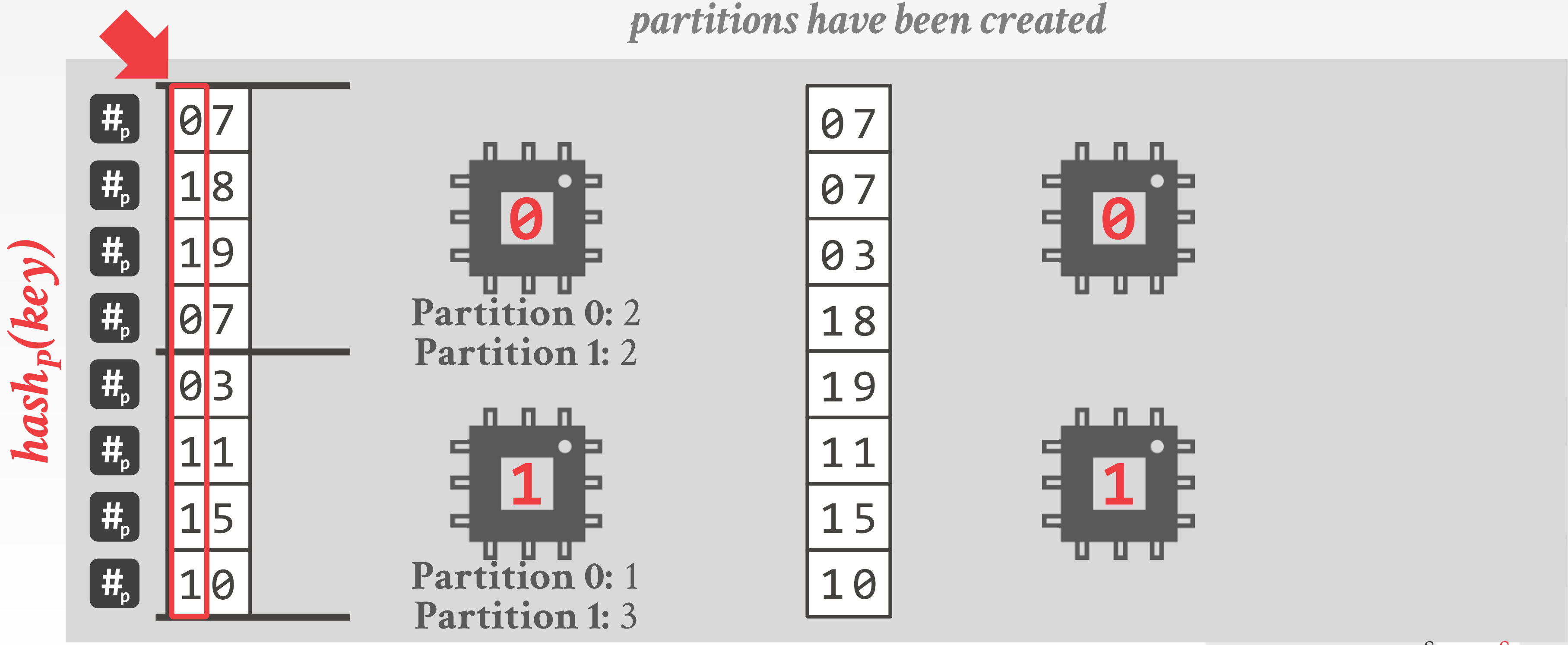
Recursively repeat until target number of partitions have been created



Source: Spyros Blanas

RADIX PARTITIONS

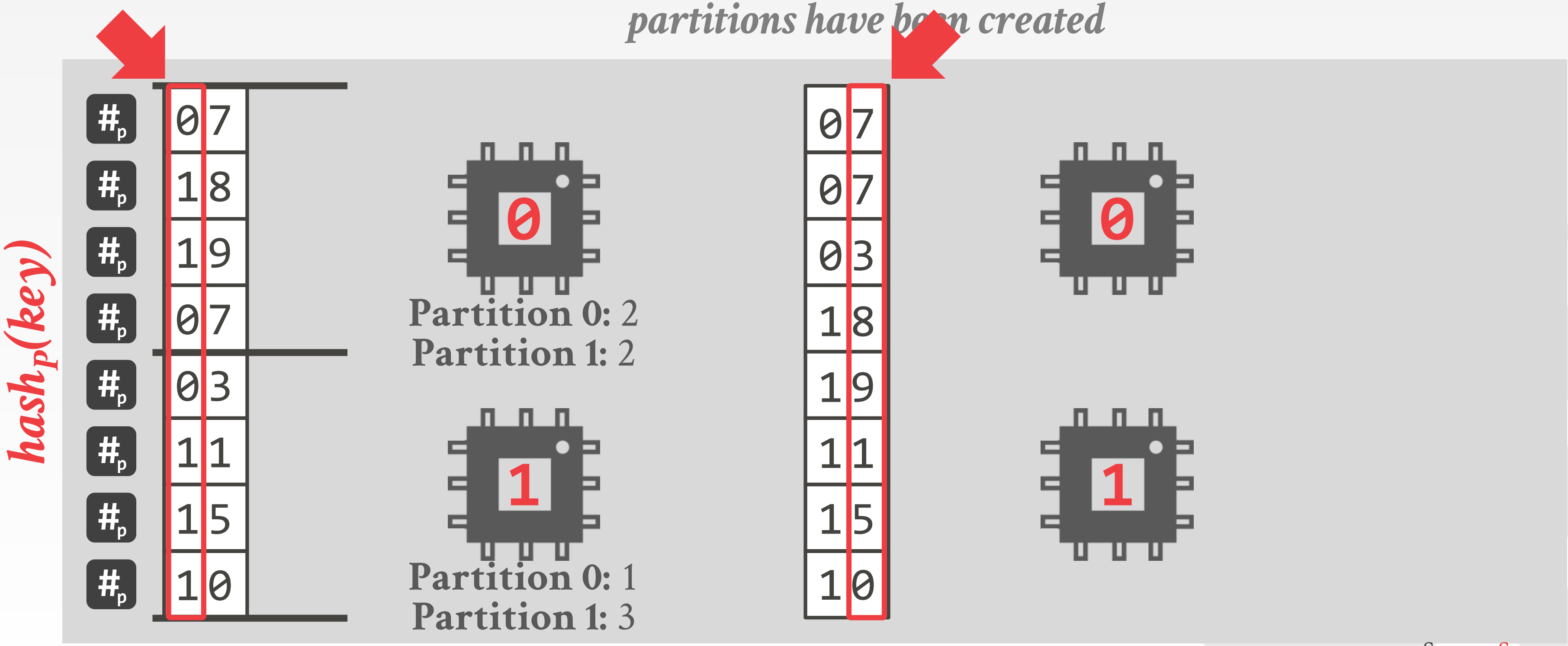
Recursively repeat until target number of partitions have been created



Source: Spyros Blanas

RADIX PARTITIONS

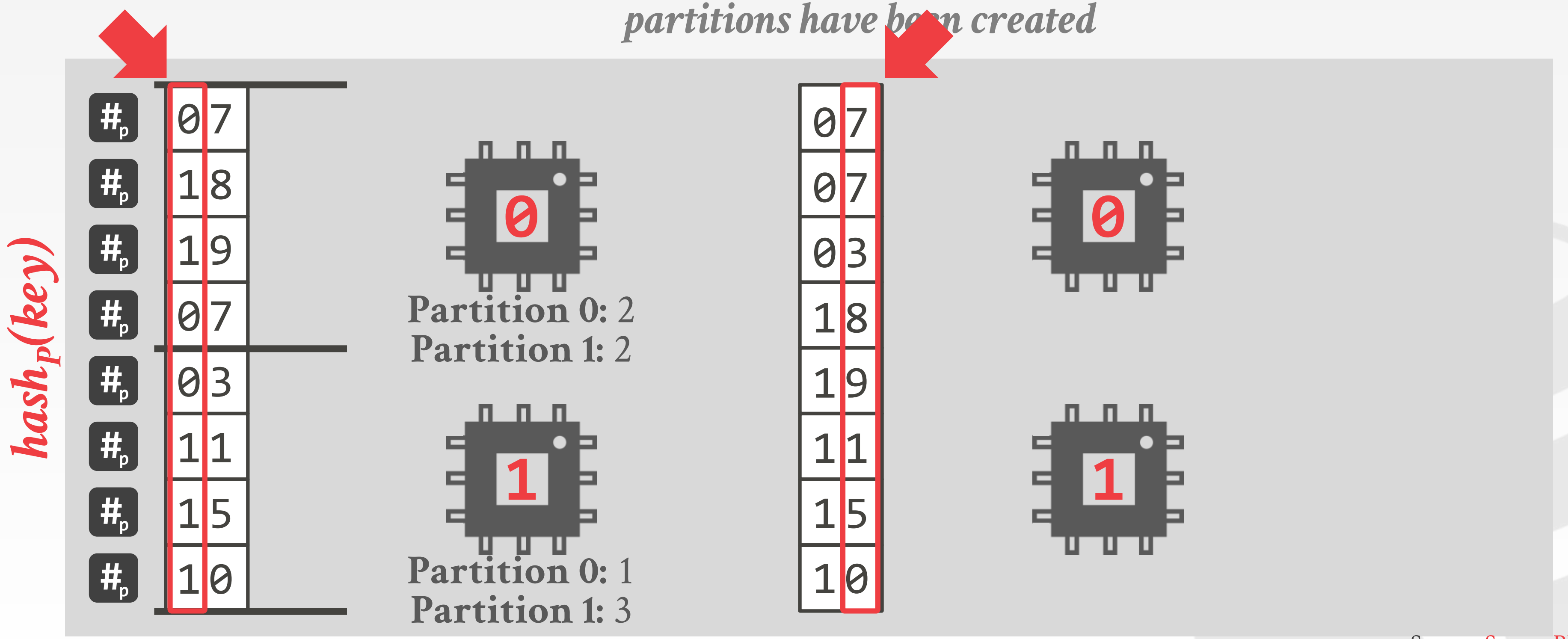
Recursively repeat until target number of partitions have been created



Source: Spyros Blanas

RADIX PARTITIONS

Recursively repeat until target number of partitions have been created



Source: Spyros Blanas

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 TABLE

Design Decision #1: Hash Function

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

Design Decision #2: Hashing Scheme

→ How to handle key collisions after hashing.
→ Trade-off between allocating a large hash table vs. additional instructions to find/insert keys.

HASH FUNCTIONS

We don't want to use a cryptographic hash function for our join algorithm.

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

HASH FUNCTIONS

MurmurHash (2008)

→ Designed to a fast, general purpose hash function.

Google CityHash (2011)

→ Based on ideas from MurmurHash2

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

Google FarmHash (2014)

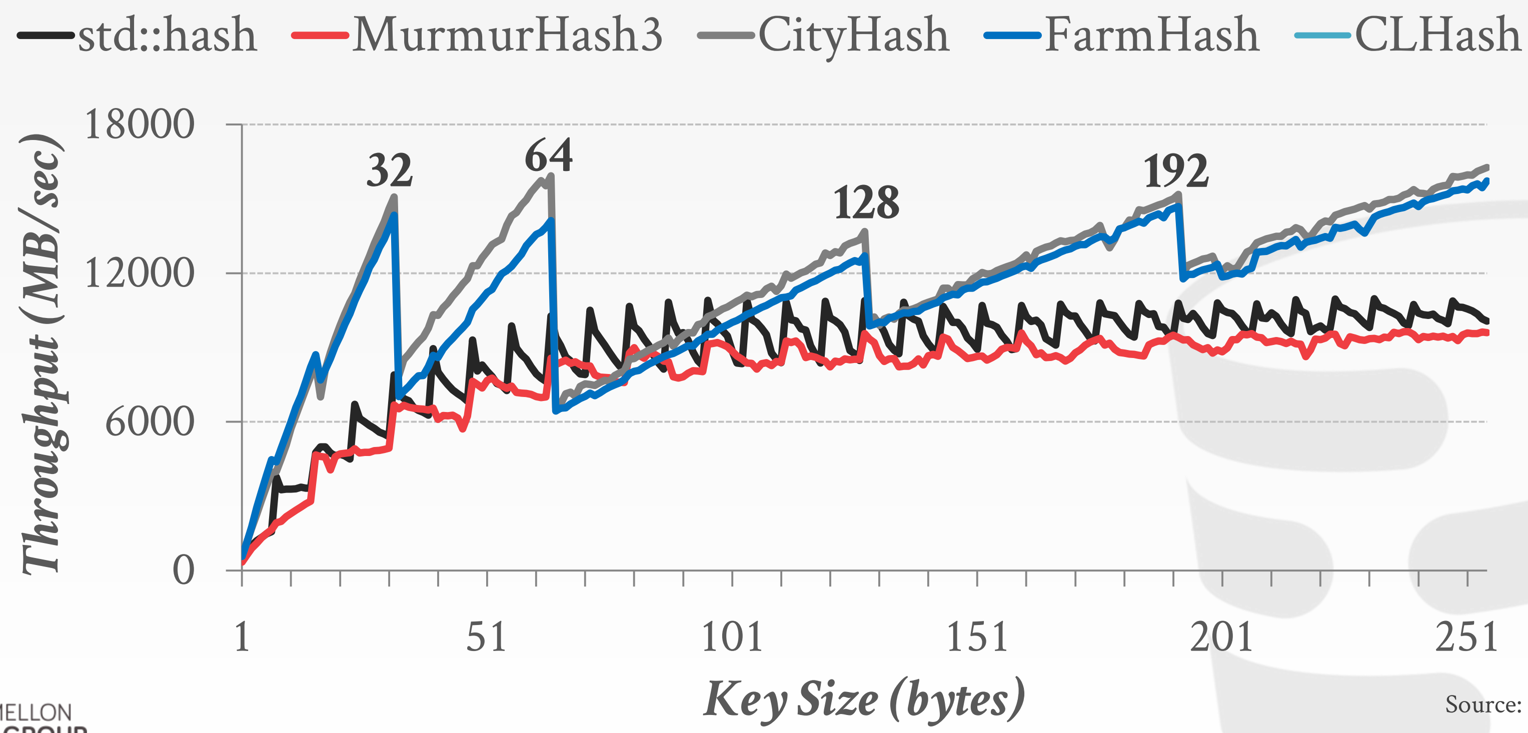
→ Newer version of CityHash with better collision rates.

CLHash (2016)

→ Fast hashing function based on carry-less multiplication.

HASH FUNCTION BENCHMARKS

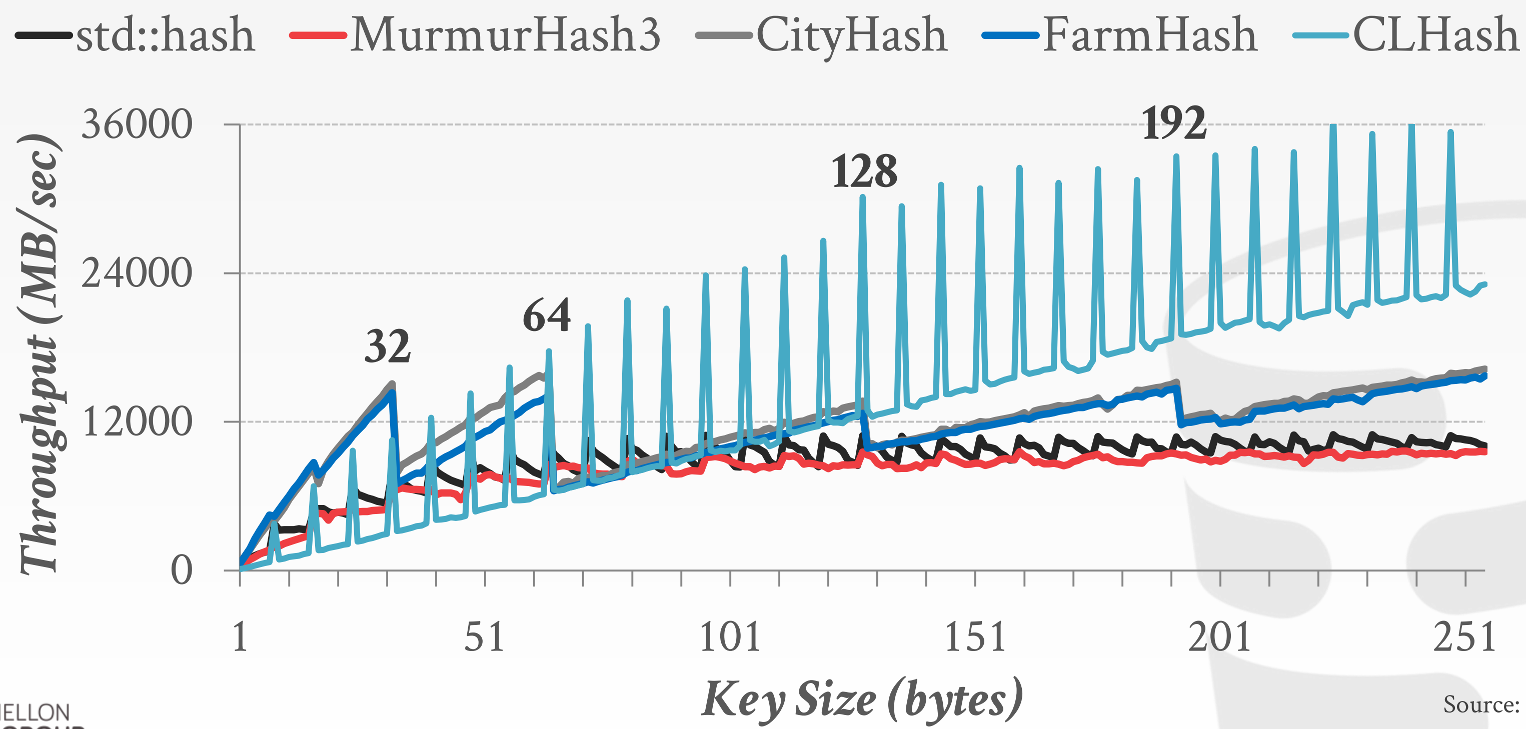
Intel Core i7-8700K @ 3.70GHz



Source: Fredrik Widlund

HASH FUNCTION BENCHMARKS

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: Cuckoo Hashing

CHAINED HASHING

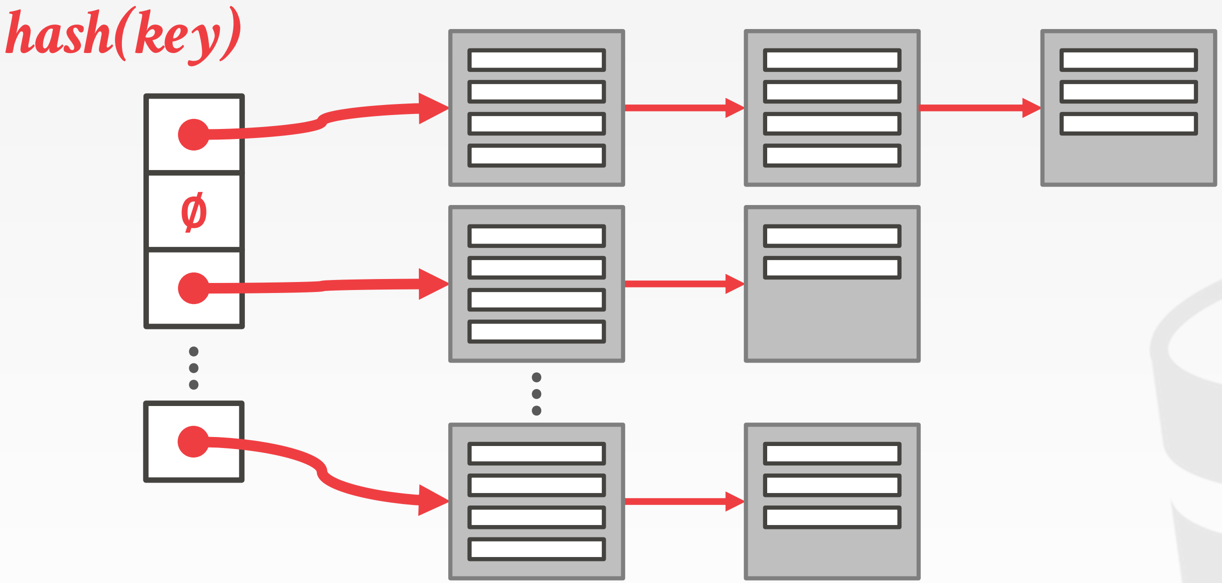
Maintain a linked list of "buckets" for each slot in the hash table.

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



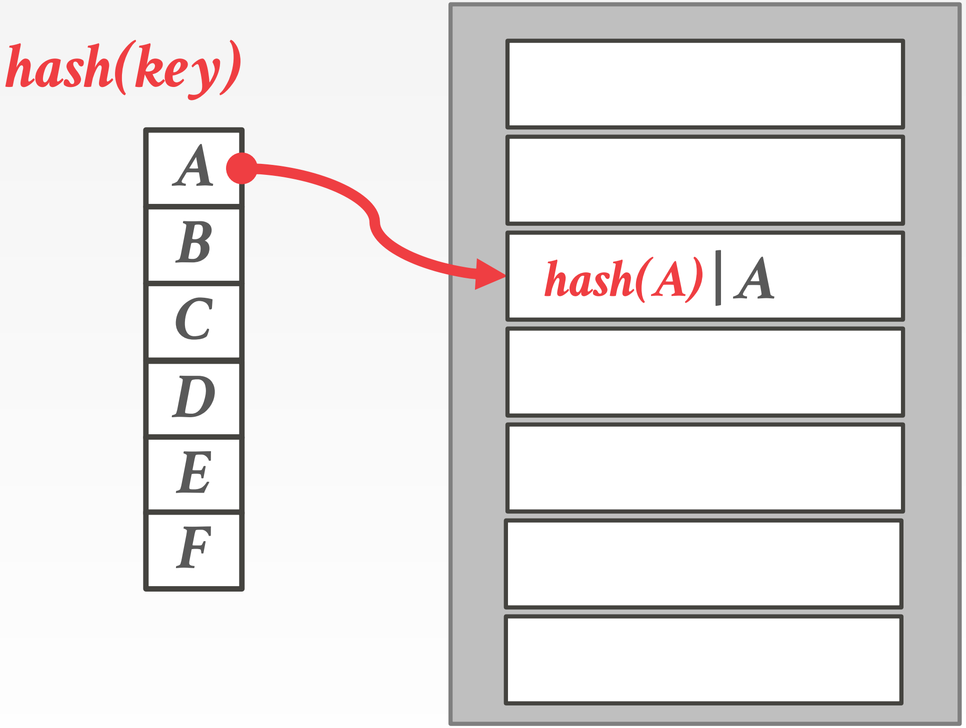
LINEAR PROBE HASHING

Single giant table of slots.

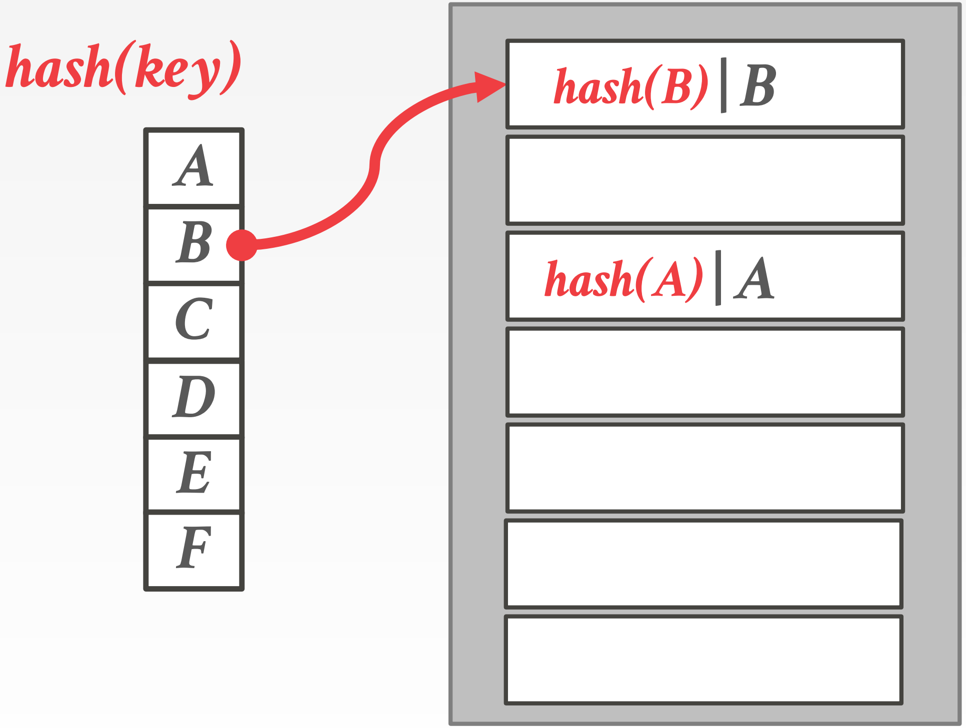
Resolve collisions by linearly searching for the next free slot in the table.

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

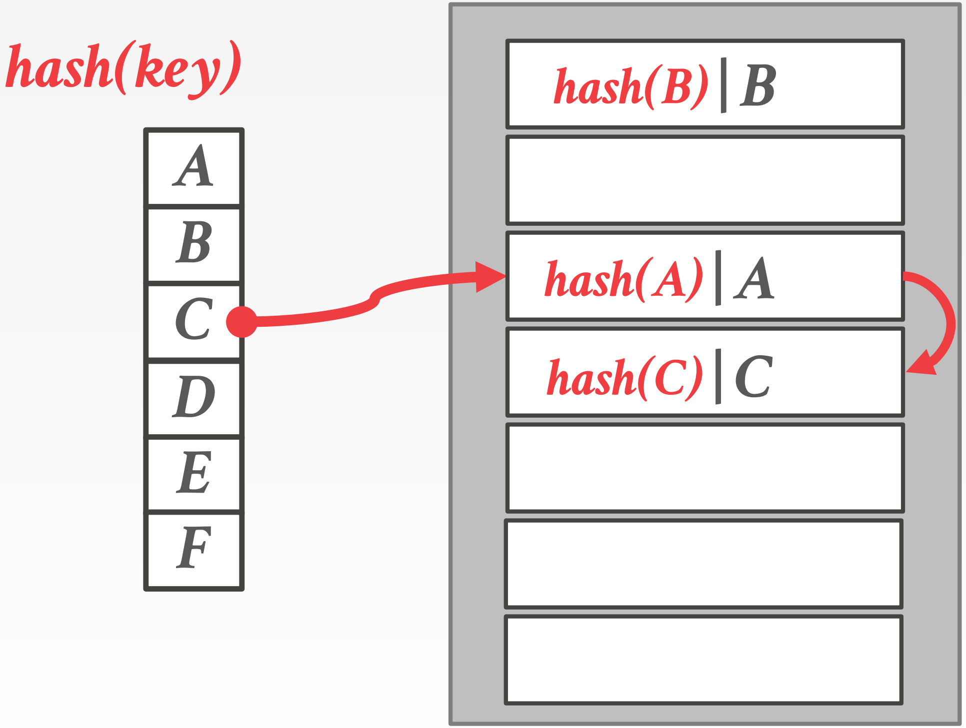
LINEAR PROBE HASHING



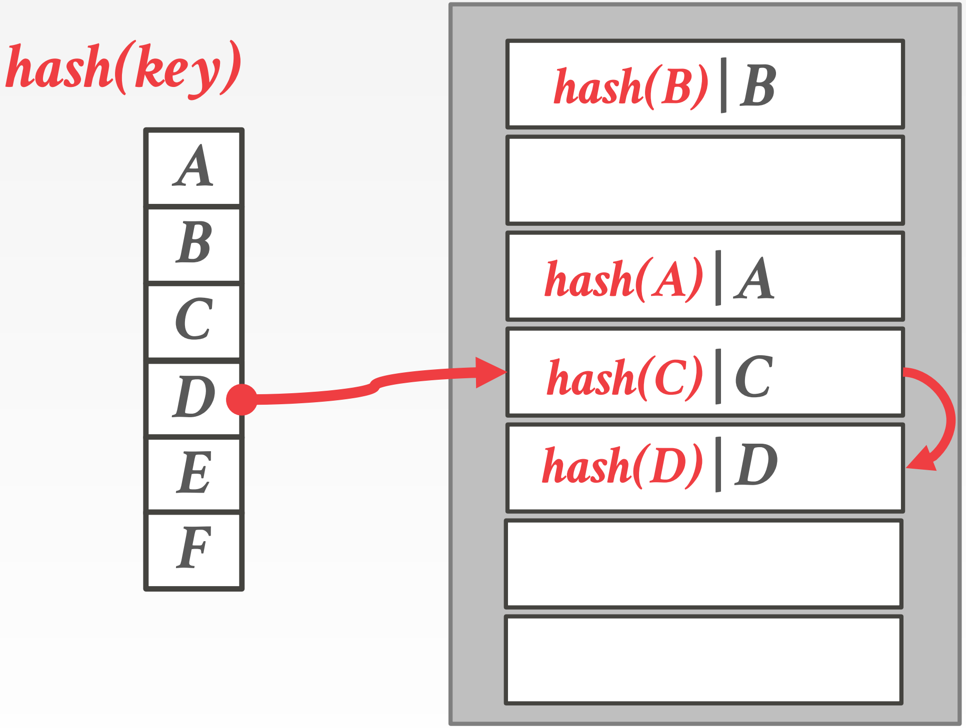
LINEAR PROBE HASHING



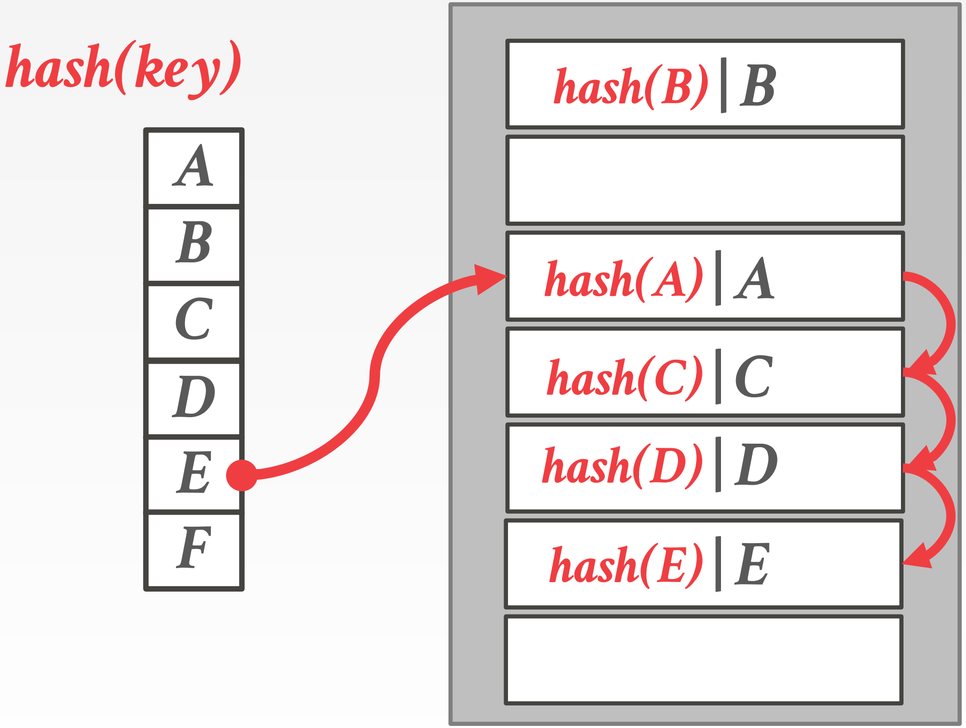
LINEAR PROBE HASHING



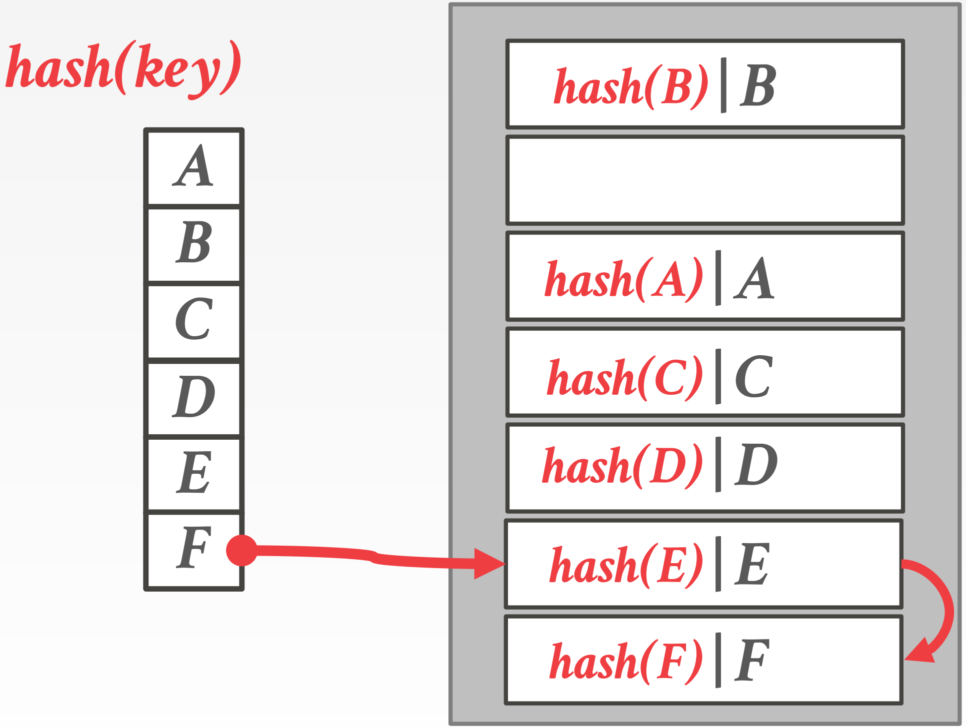
LINEAR PROBE HASHING



LINEAR PROBE HASHING



LINEAR PROBE HASHING



OBSERVATION

To reduce the # of wasteful comparisons during the join, it is important to avoid collisions of hashed keys.

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

ROBIN HOOD HASHING

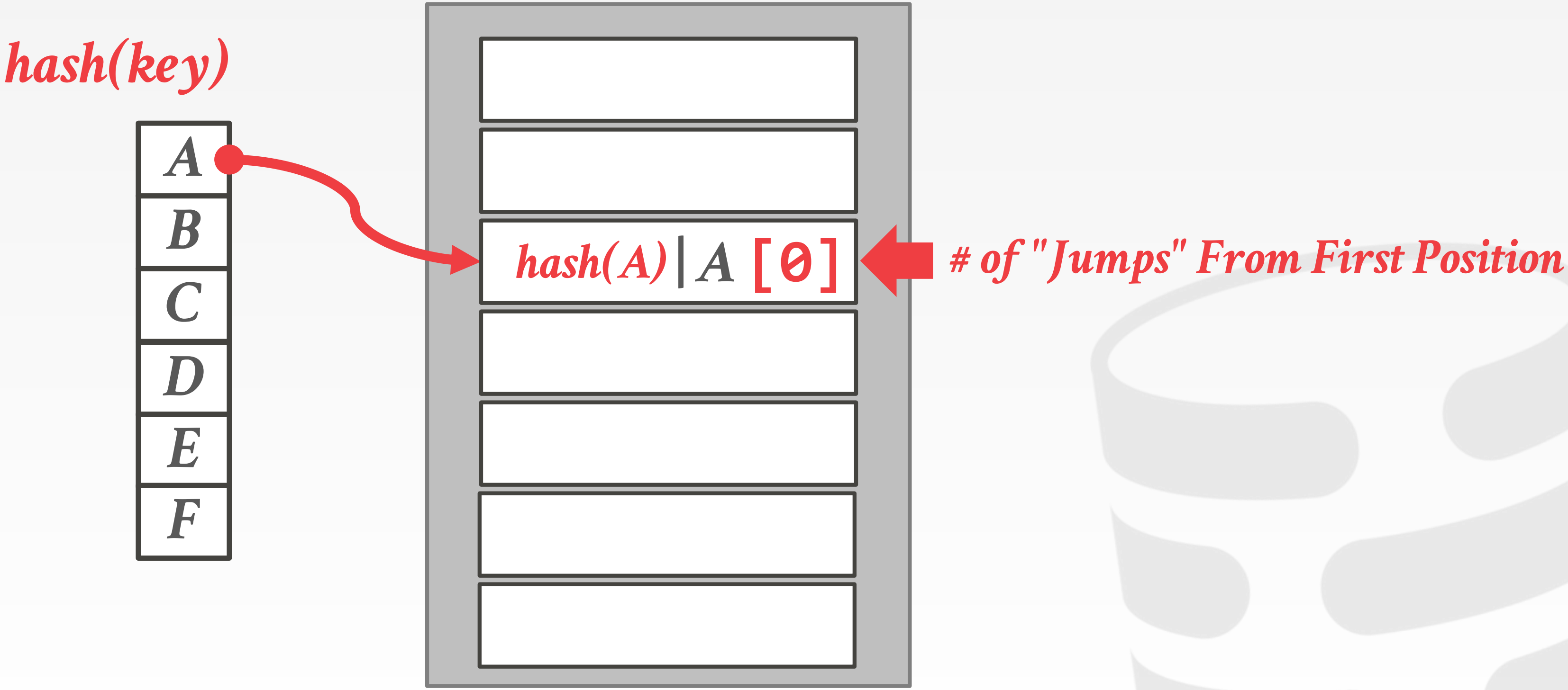
Variant of linear hashing that steals slots from "rich" keys and give them to "poor" keys.

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

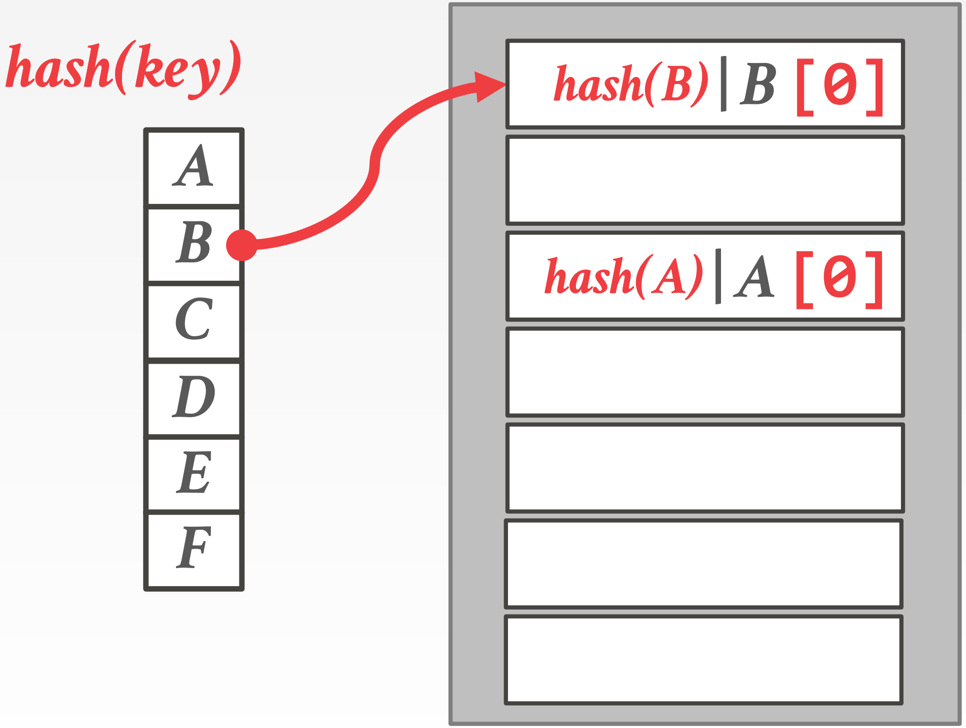
→ On insert, a key takes the slot of another key if the first key is farther away from its optimal position than the second key.

ROBIN HOOD HASHING
Foundations of Computer Science 1985

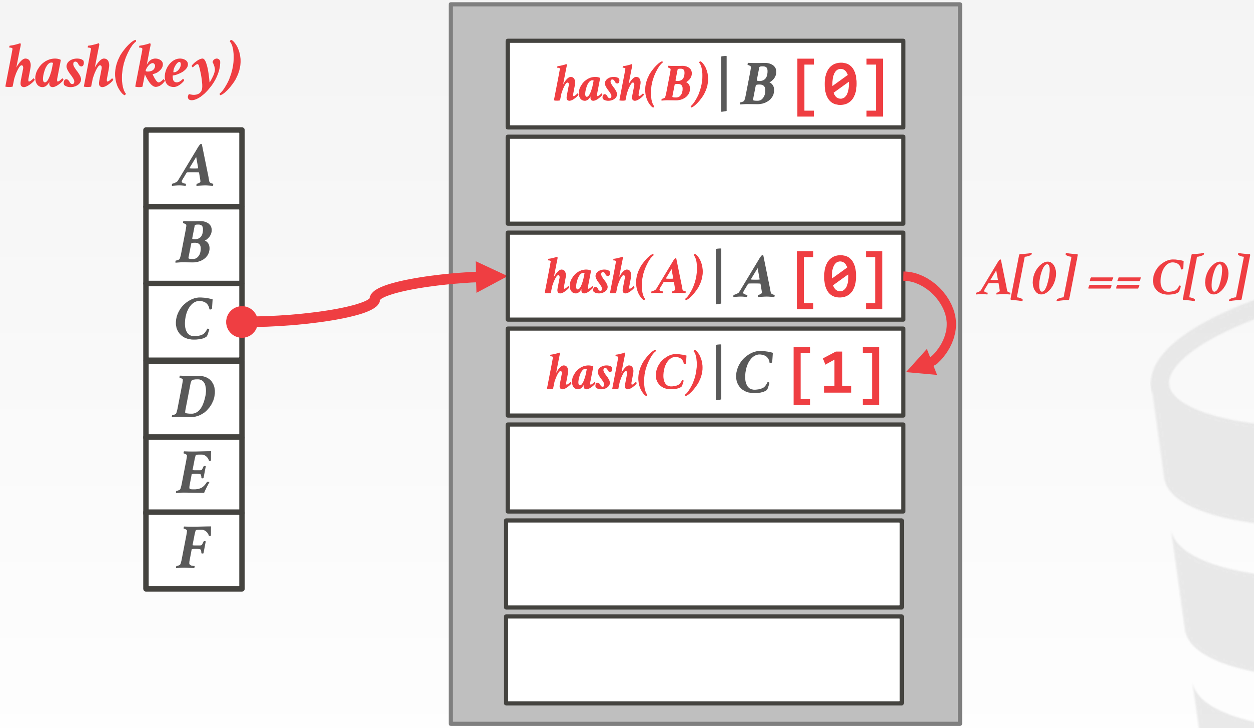
ROBIN HOOD HASHING



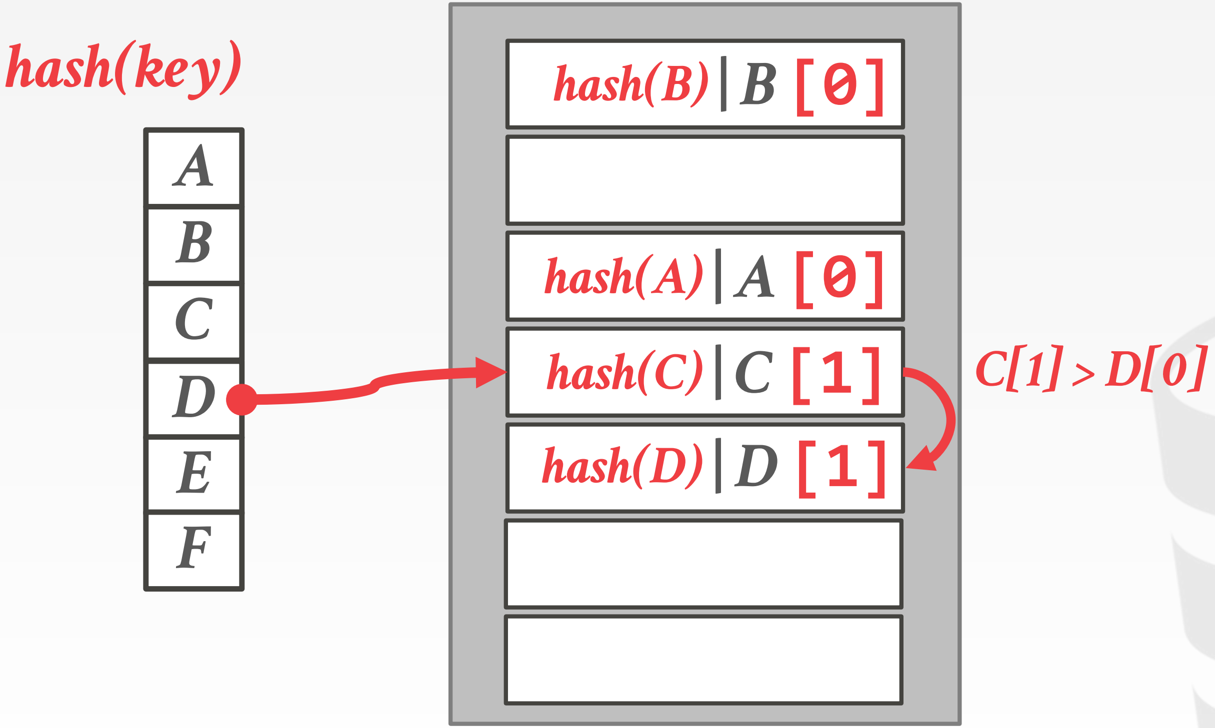
ROBIN HOOD HASHING



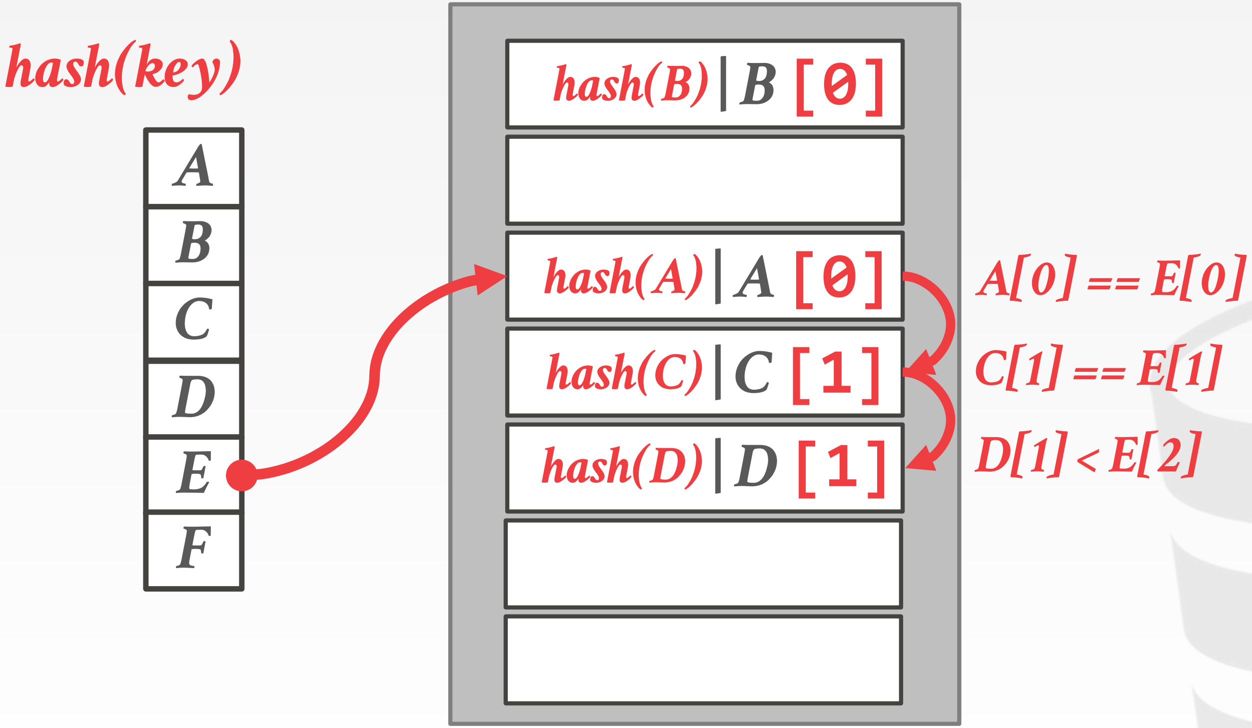
ROBIN HOOD HASHING



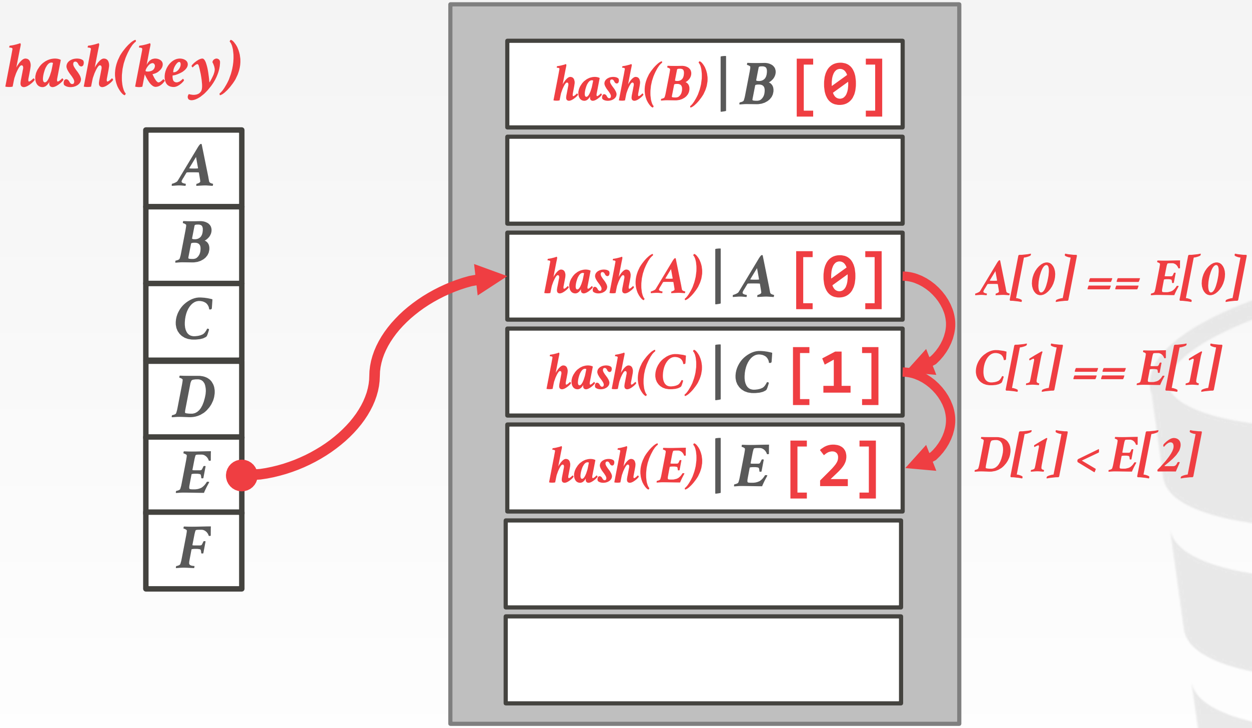
ROBIN HOOD HASHING



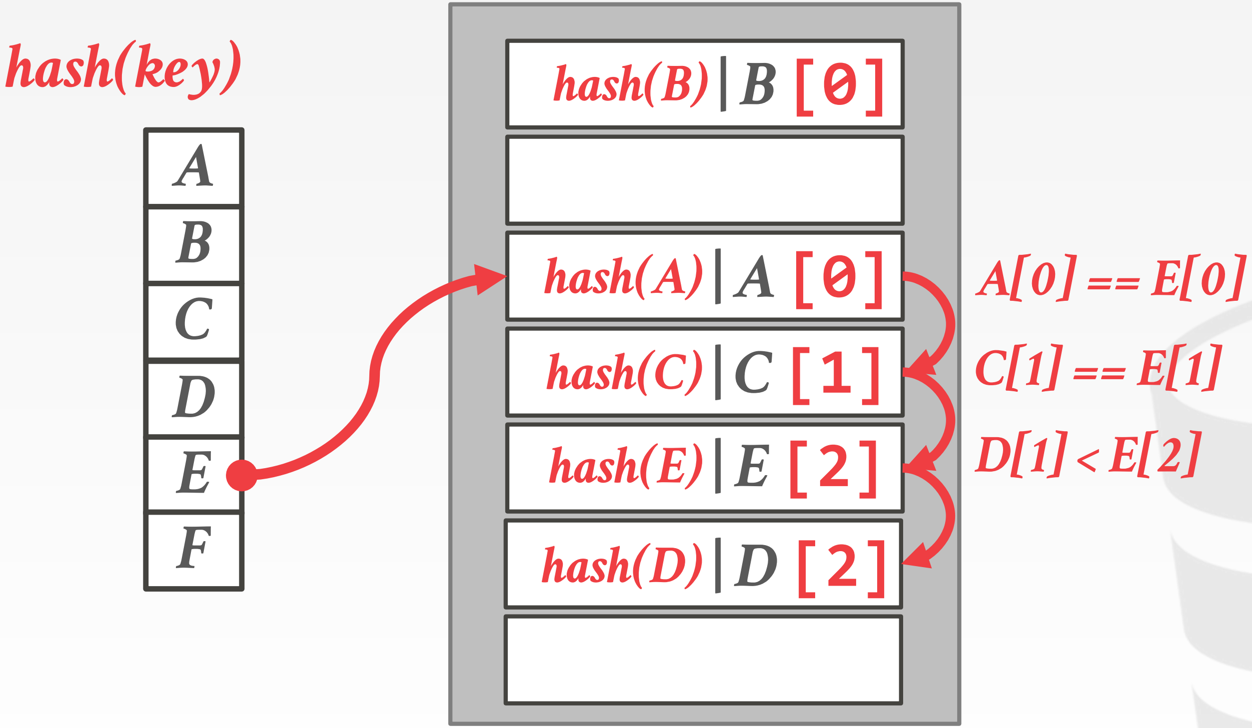
ROBIN HOOD HASHING



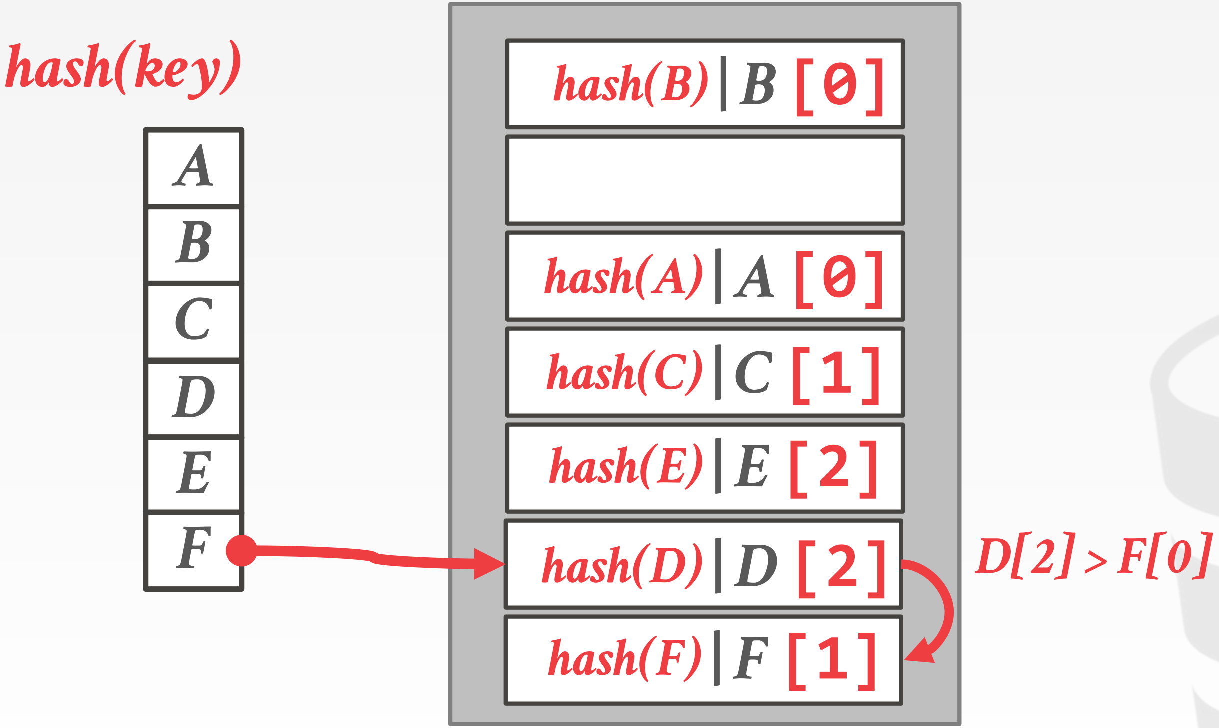
ROBIN HOOD HASHING



ROBIN HOOD HASHING



ROBIN HOOD HASHING



CUCKOO HASHING

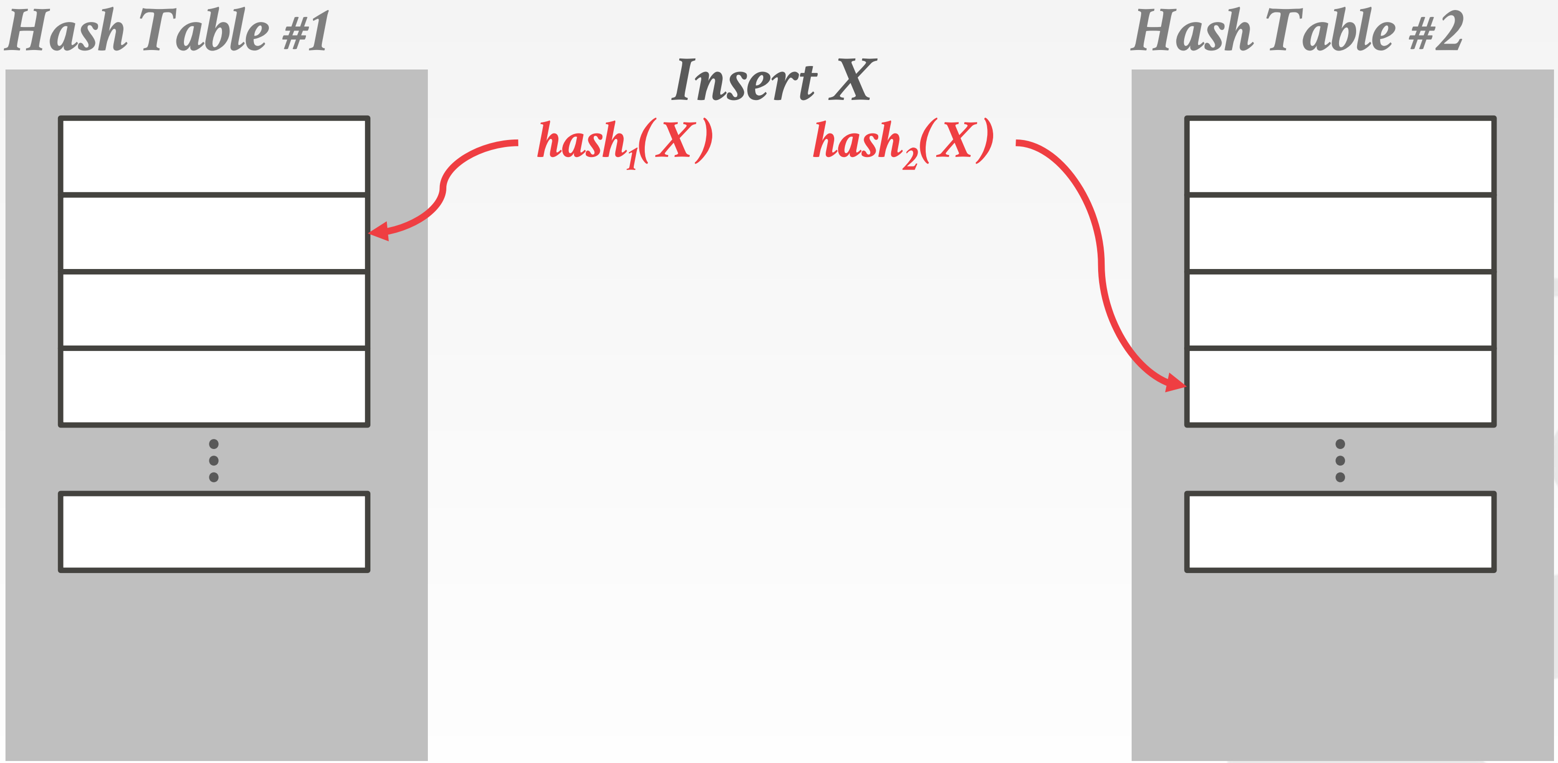
Use multiple tables with different hash functions.

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

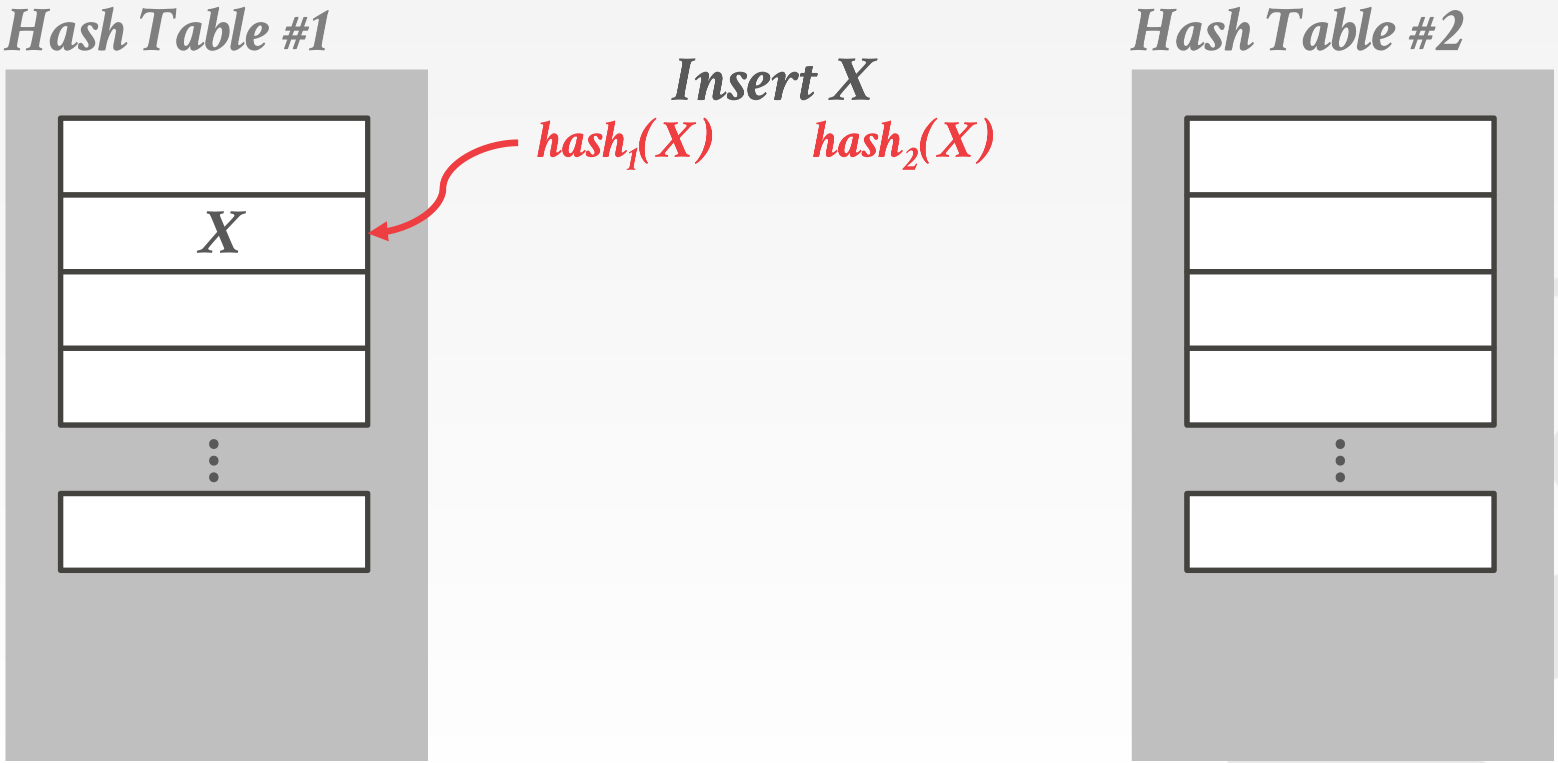
→ If no table has a free slot, evict the element from one of them and then re-hash it find a new location.

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

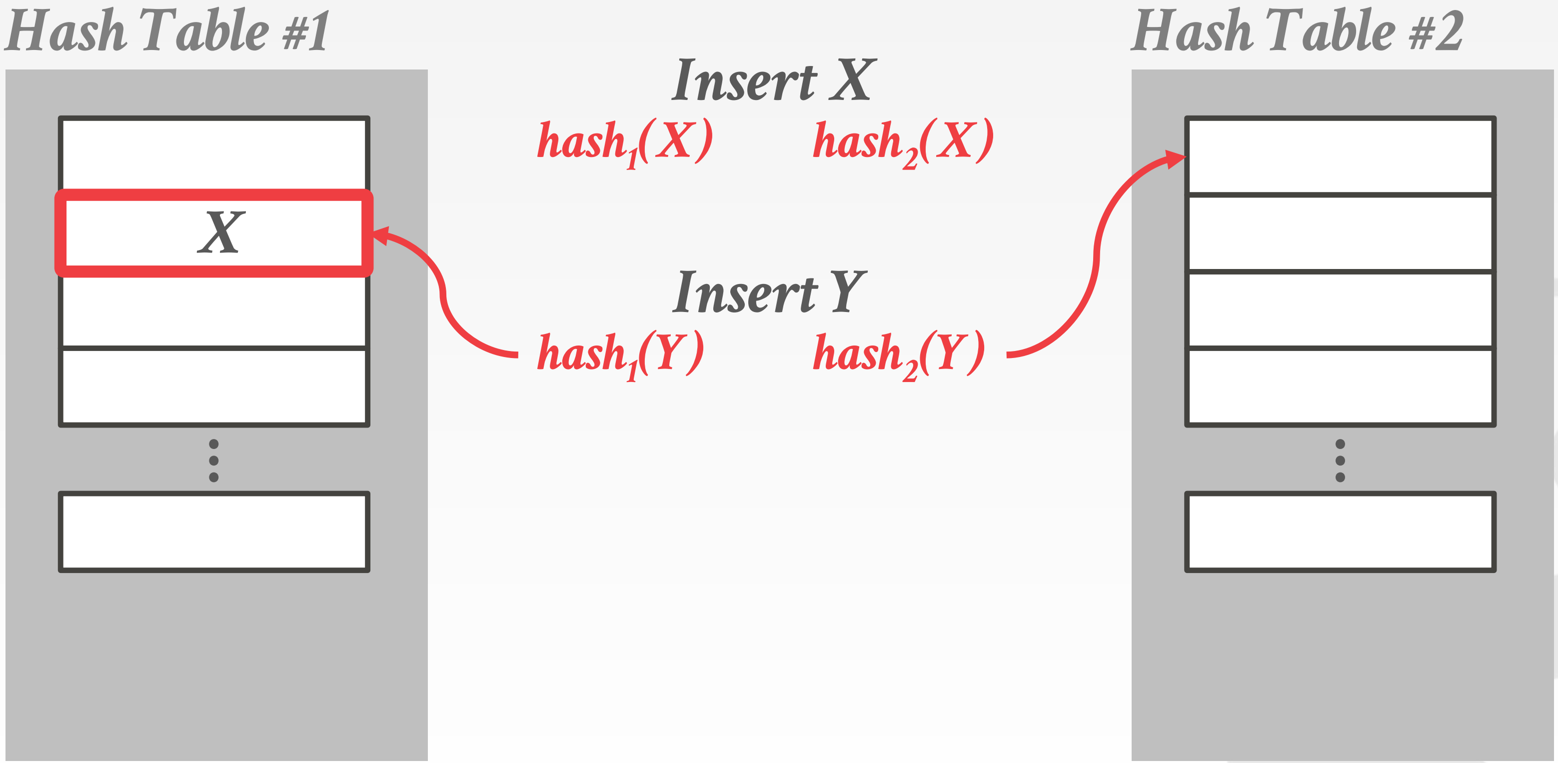
CUCKOO HASHING



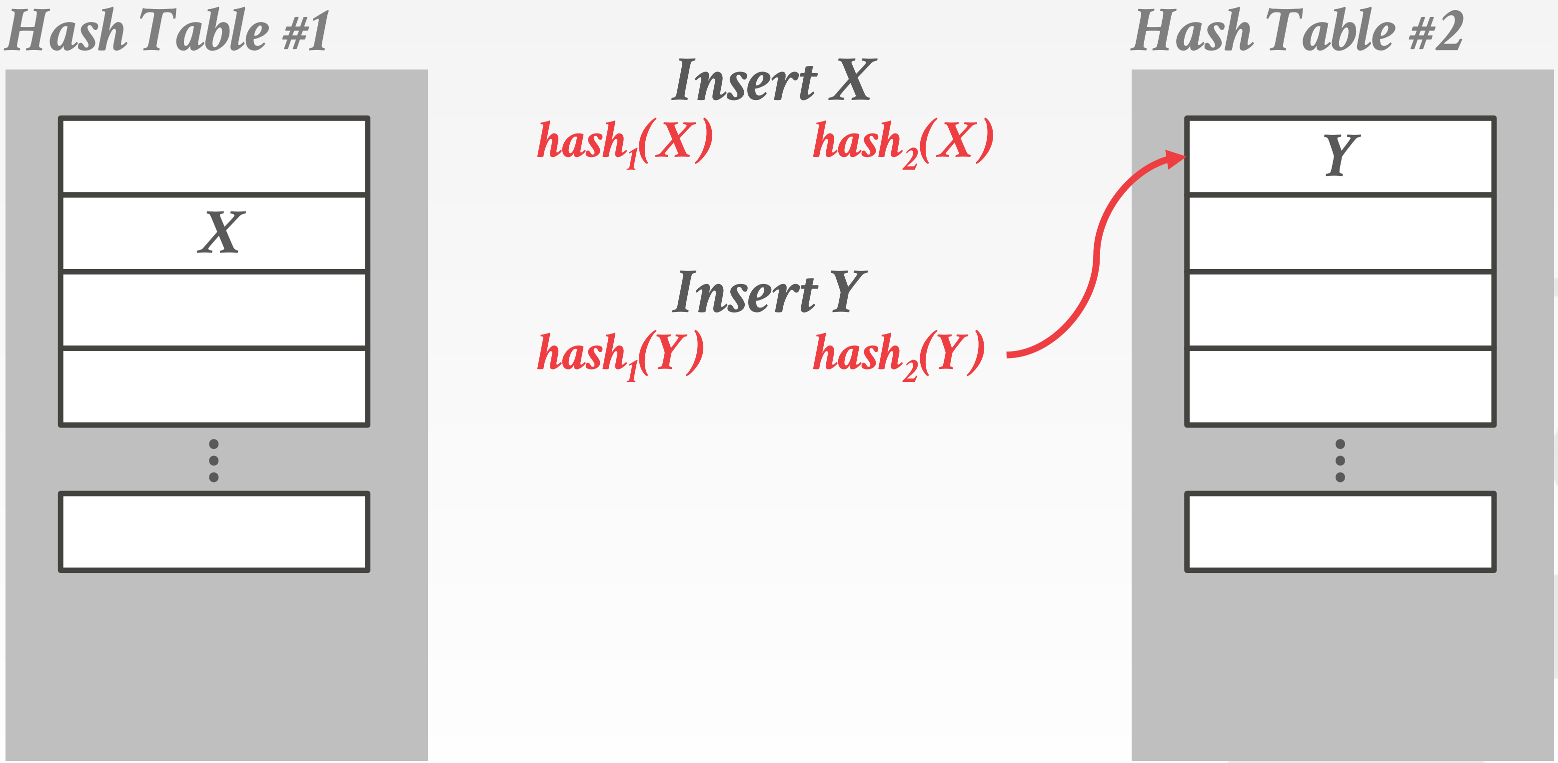
CUCKOO HASHING



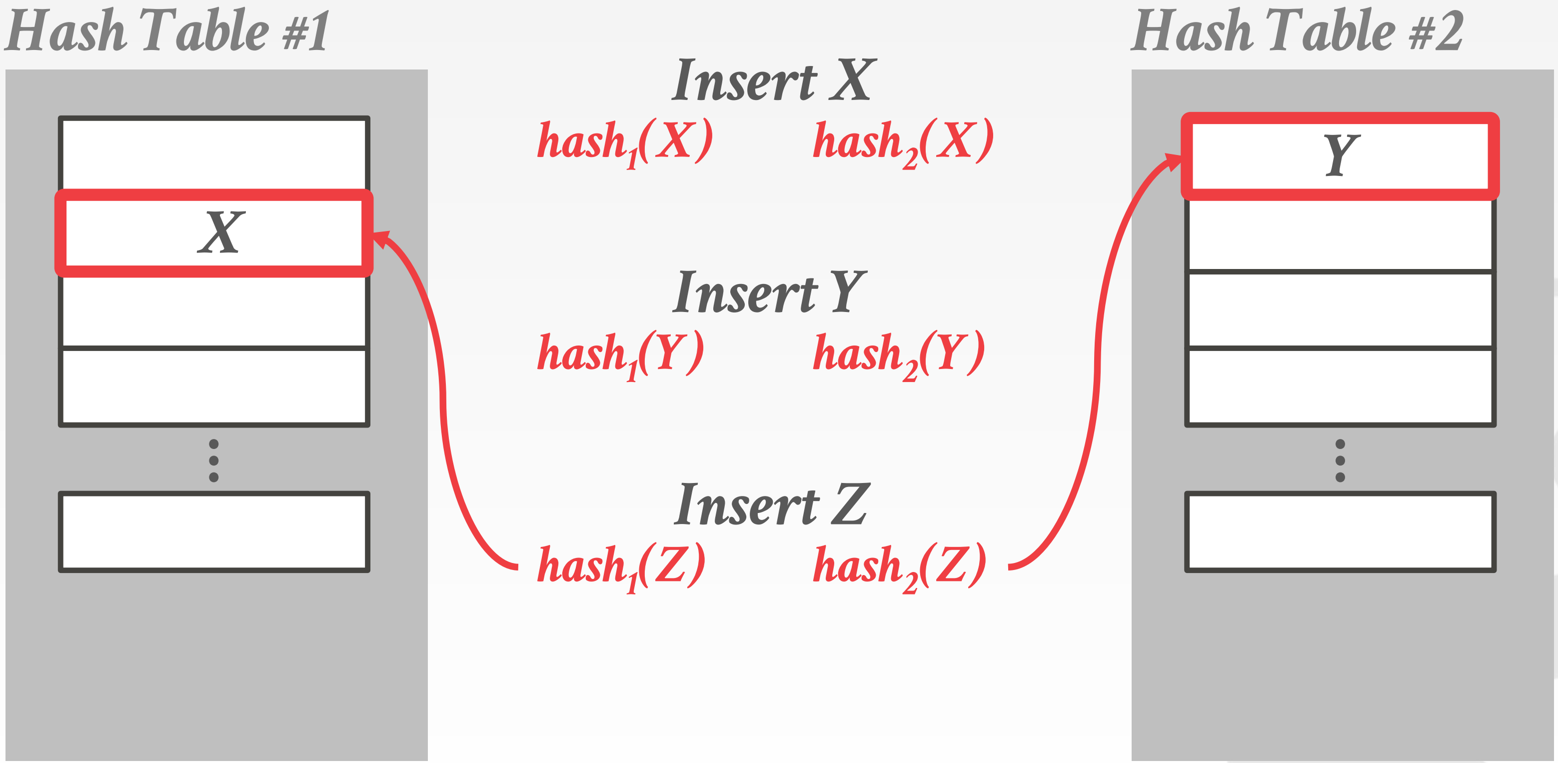
CUCKOO HASHING



CUCKOO HASHING



CUCKOO HASHING



Insert X

$hash_1(X)$ *$hash_2(X)$*

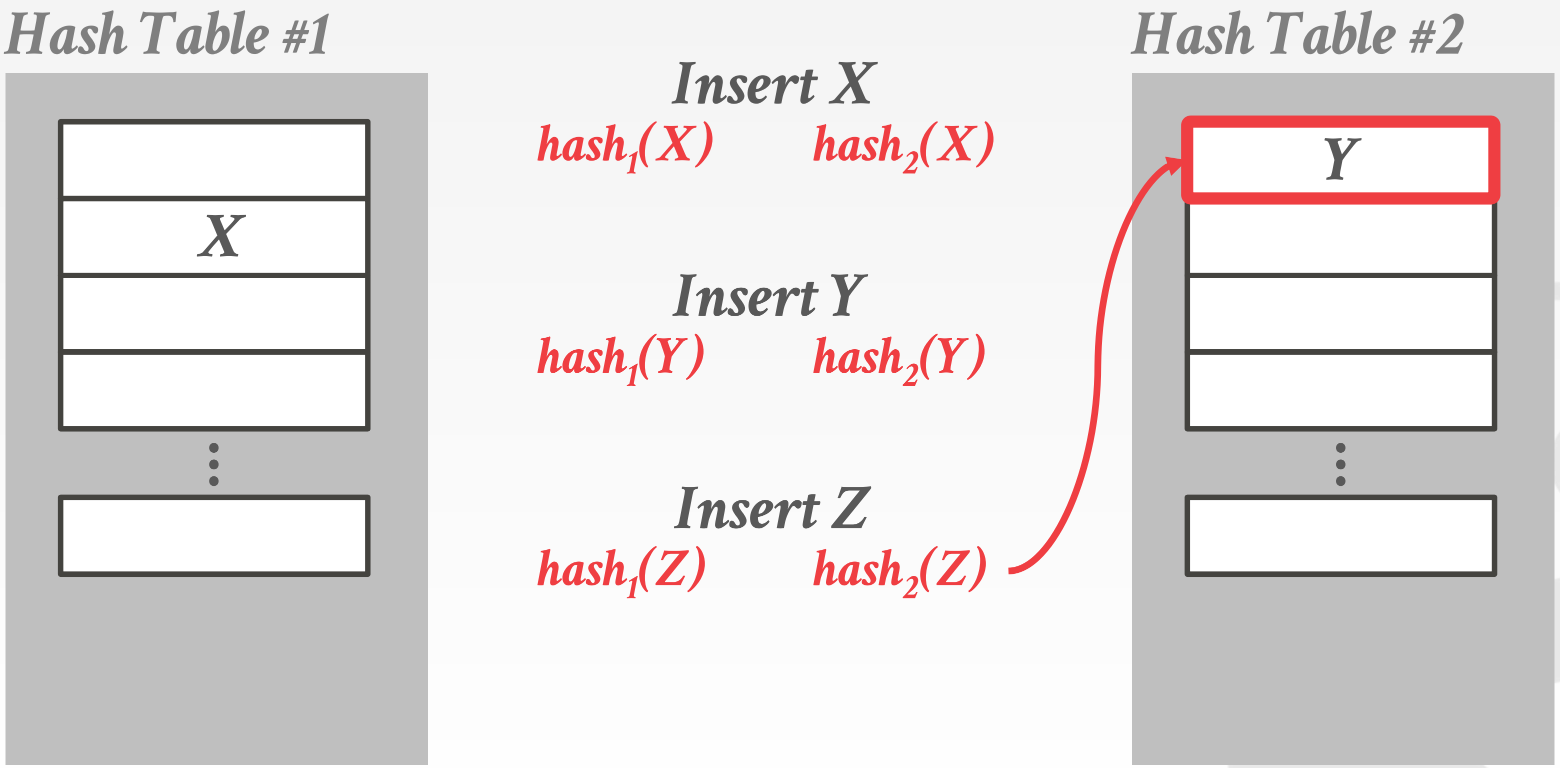
Insert Y

$hash_1(Y)$ *$hash_2(Y)$*

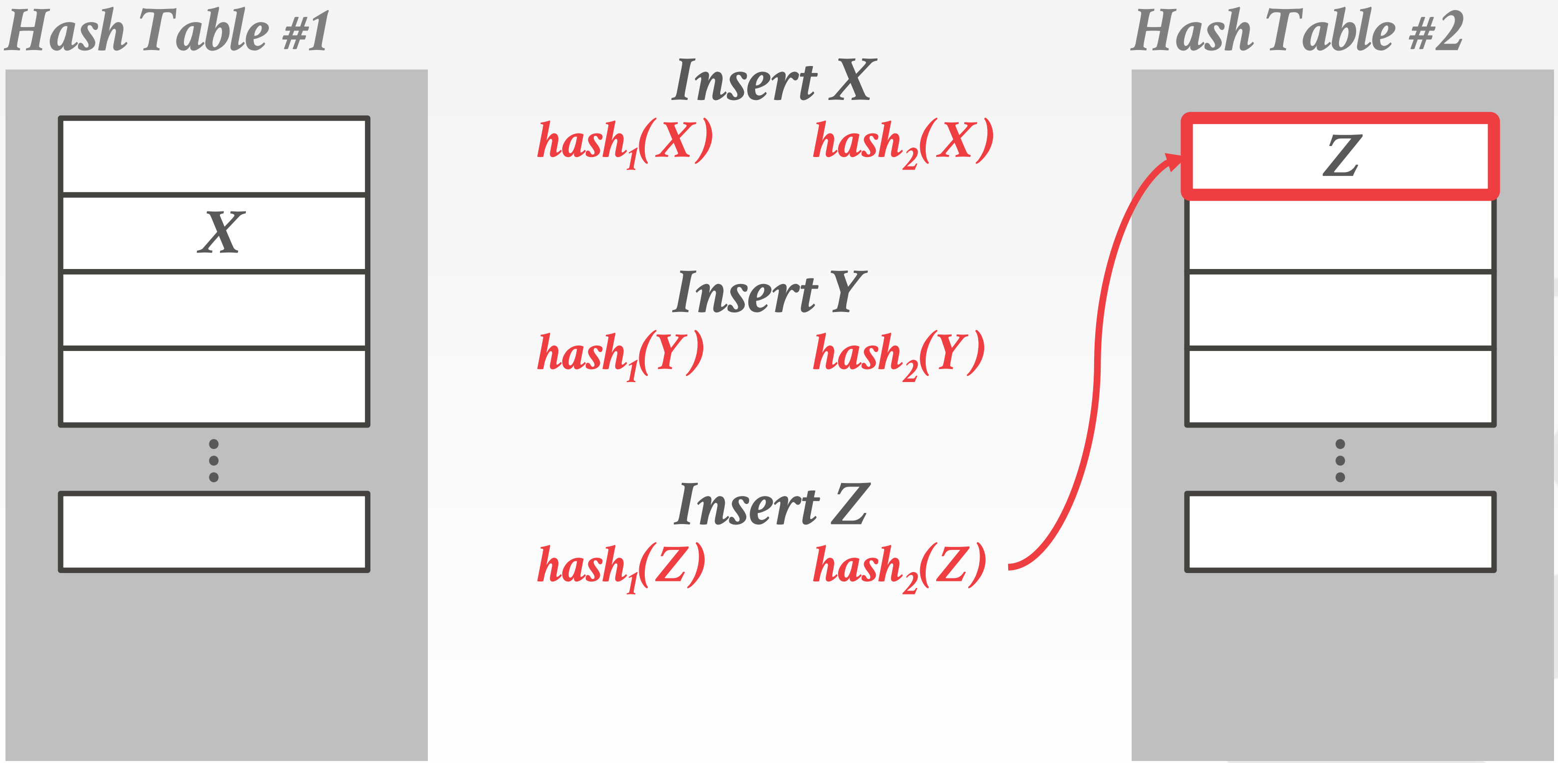
Insert Z

$hash_1(Z)$ *$hash_2(Z)$*

CUCKOO HASHING

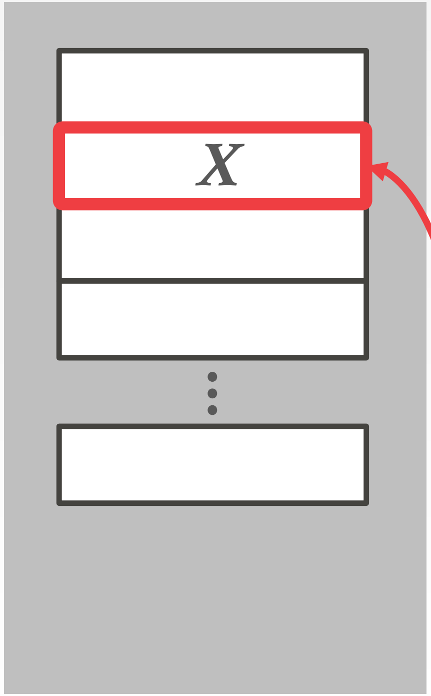


CUCKOO HASHING



CUCKOO HASHING

Hash Table #1



Insert X

$hash_1(X)$ $hash_2(X)$

Insert Y

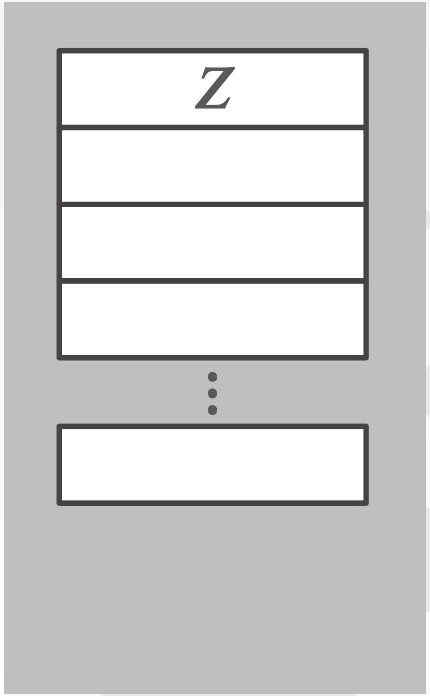
$hash_1(Y)$ $hash_2(Y)$

Insert Z

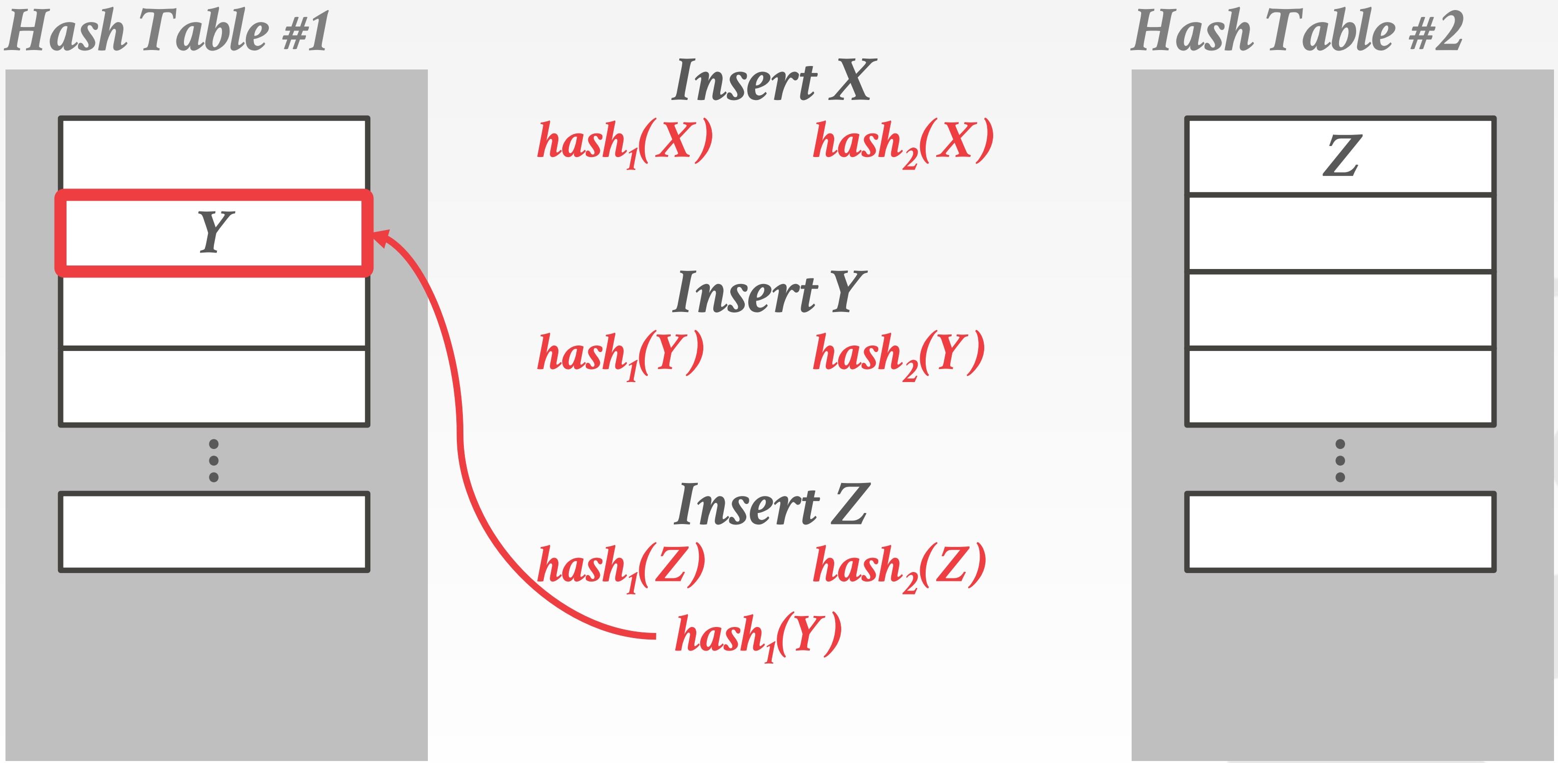
$hash_1(Z)$ $hash_2(Z)$

$hash_1(Y)$

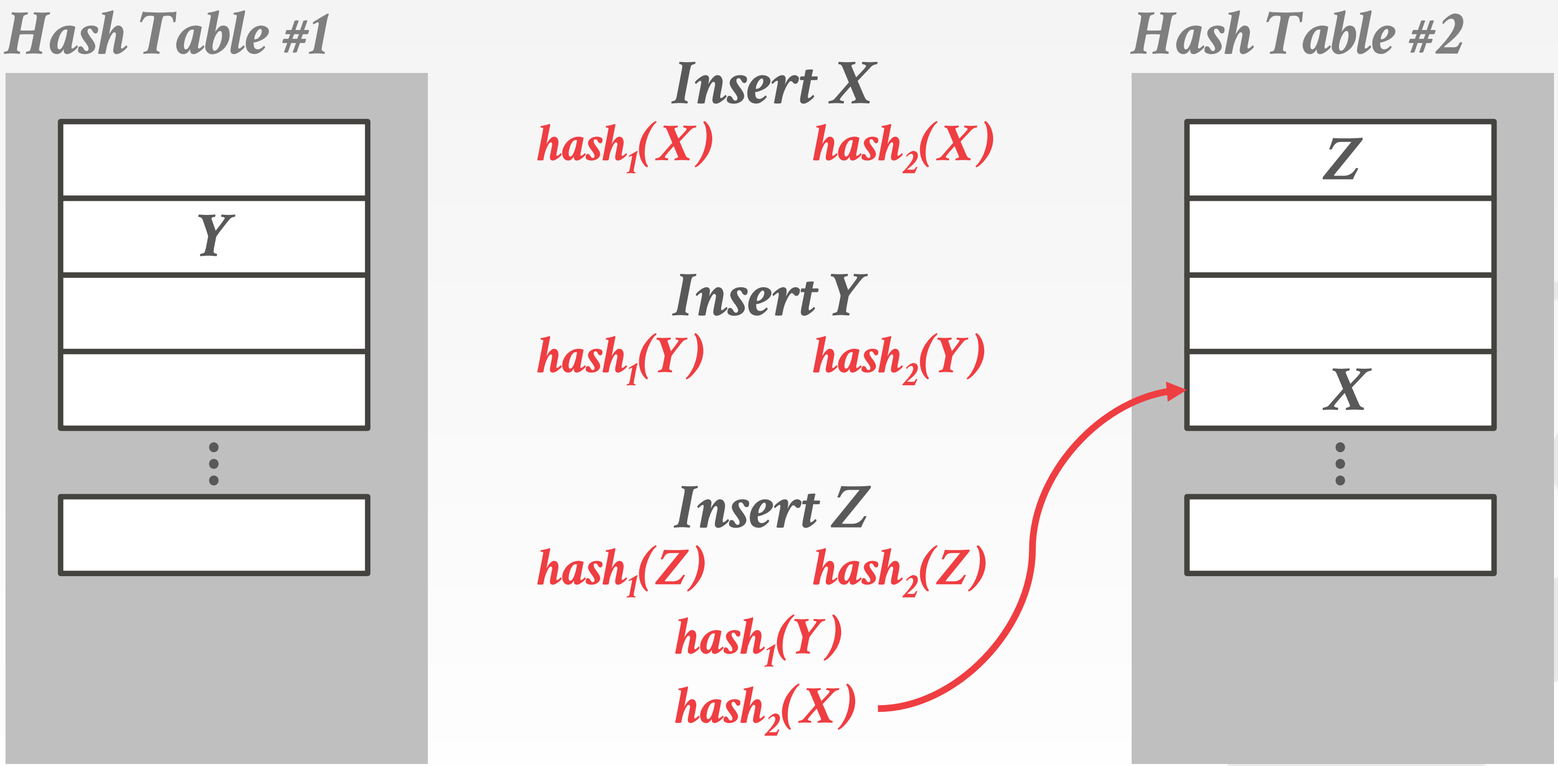
Hash Table #2



CUCKOO HASHING



CUCKOO HASHING



CARNEGIE MELLON
DATABASE GROUP

CUCKOO HASHING

Threads have to make sure that they don't get stuck in an infinite loop when moving keys.

If we find a cycle, then we can rebuild the entire hash tables with new hash functions.

→ With **two** hash functions, we (probably) won't need to rebuild the table until it is at about 50% full.
→ With **three** hash functions, we (probably) won't need to rebuild the table until it is at about 90% full.

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.

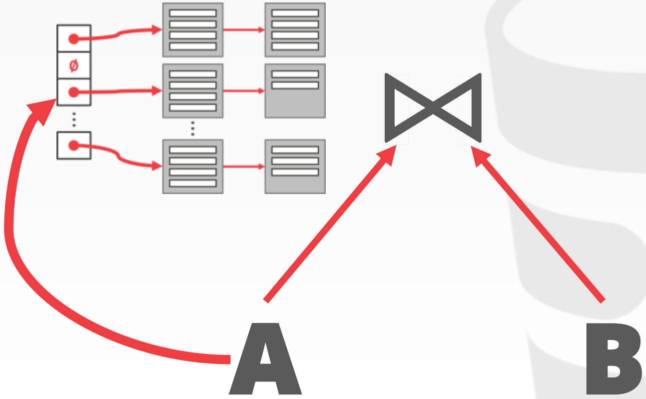
→ Otherwise, synchronize their access to the cursor on **S**

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.***



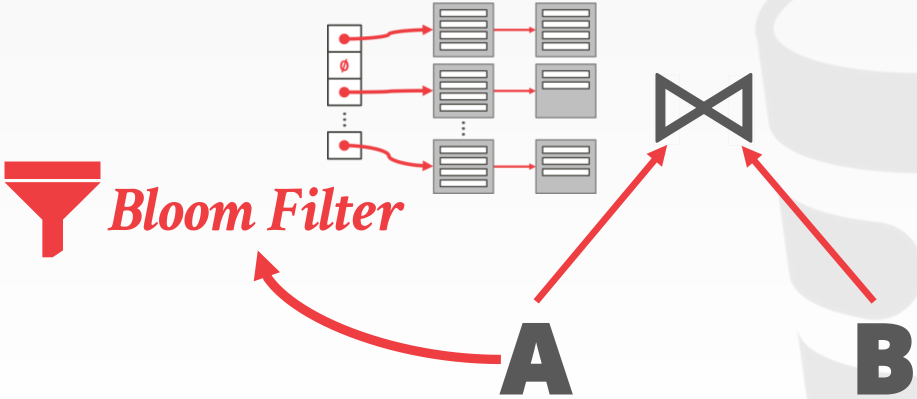
MICRO ADAPTIVITY IN VECTORWISE
SIGMOD 2013

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.***



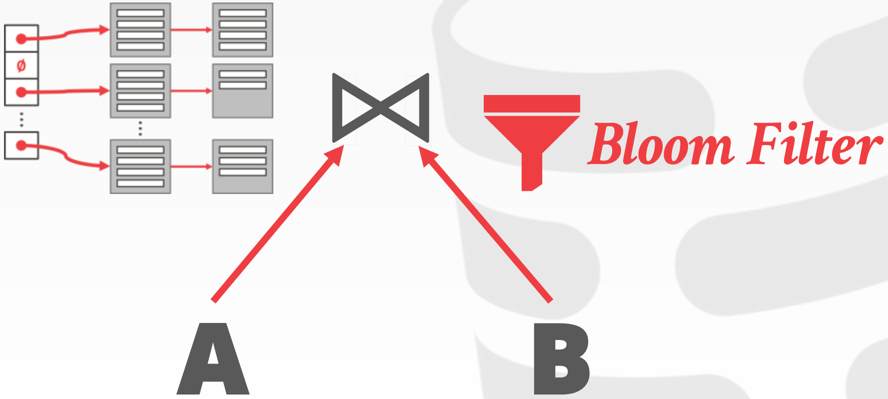
MICRO ADAPTIVITY IN VECTORWISE
SIGMOD 2013

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.***



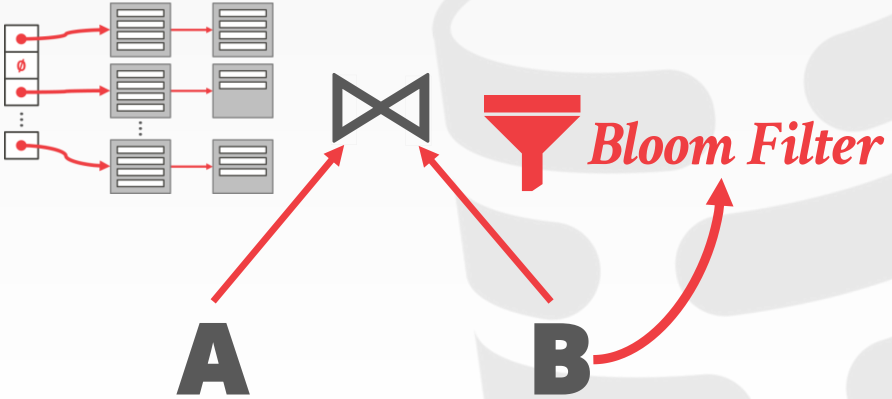
MICRO ADAPTIVITY IN VECTORWISE
SIGMOD 2013

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.***



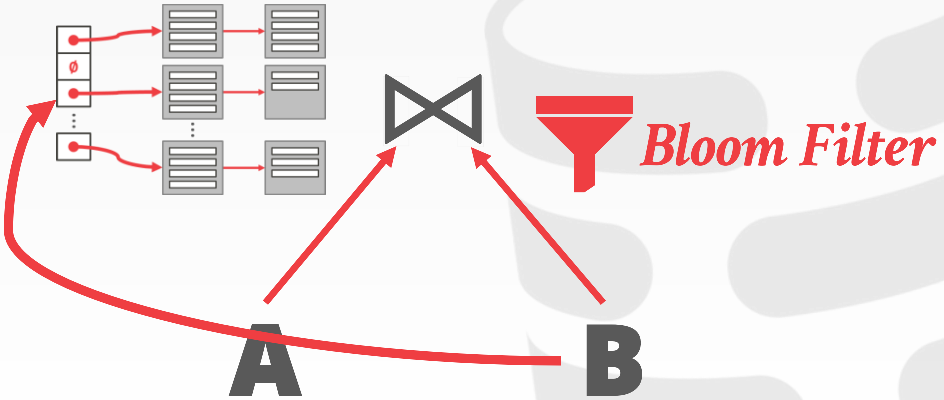
MICRO ADAPTIVITY IN VECTORWISE
SIGMOD 2013

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

Primary key – foreign key join
→ Outer Relation (Build): 16M tuples, 16 bytes each
→ Inner Relation (Probe): 256M tuples, 16 bytes each

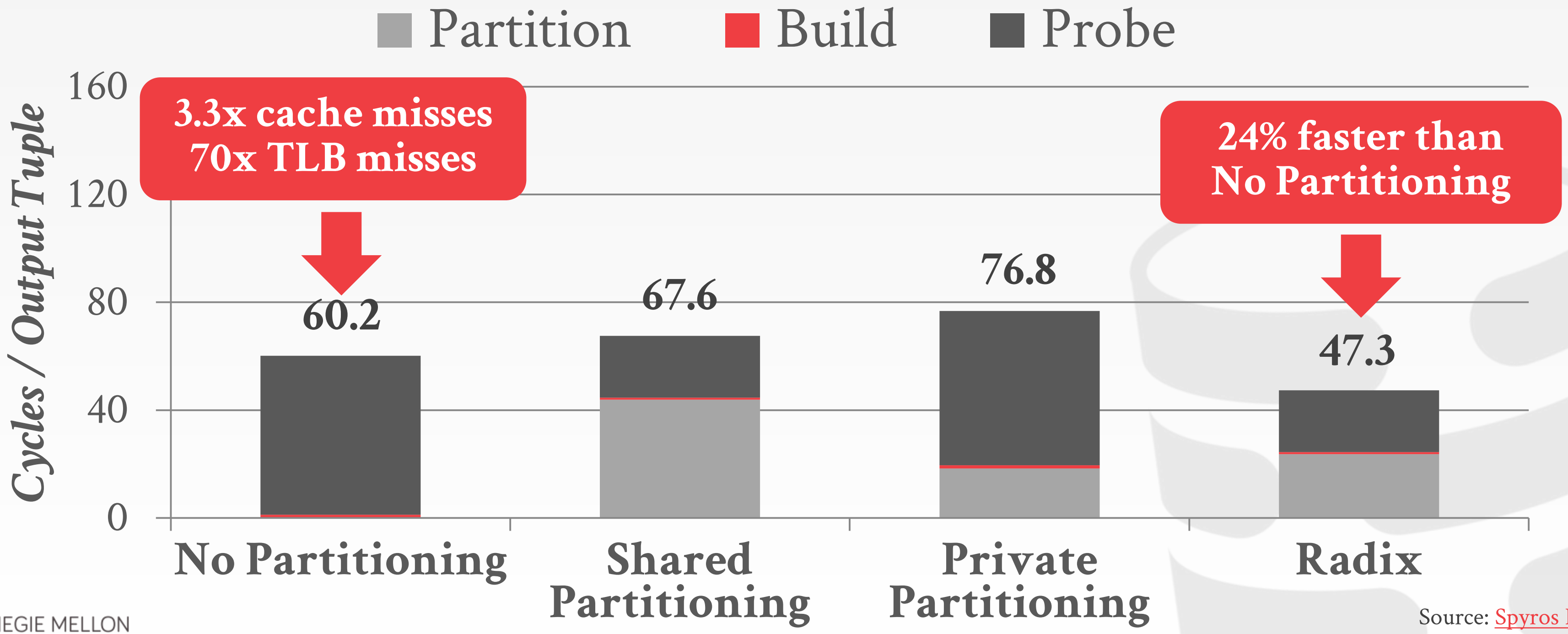
Uniform and highly skewed (Zipf; s=1.25)

No output materialization

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

HASH JOIN – UNIFORM DATA SET

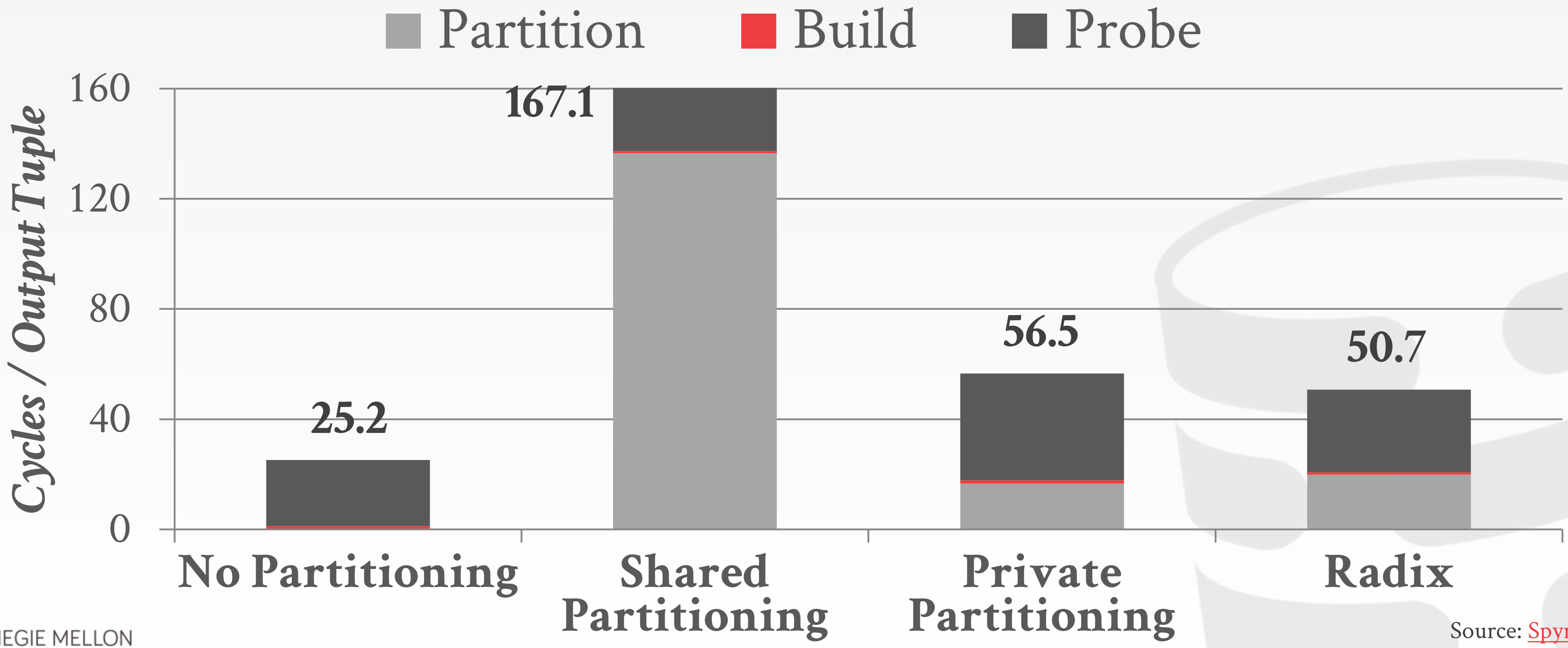
Intel Xeon CPU X5650 @ 2.66GHz
6 Cores with 2 Threads Per Core



Source: Spyros Blanas

HASH JOIN – SKEWED DATA SET

Intel Xeon CPU X5650 @ 2.66GHz
6 Cores with 2 Threads Per Core



Source: Spyros Blanas

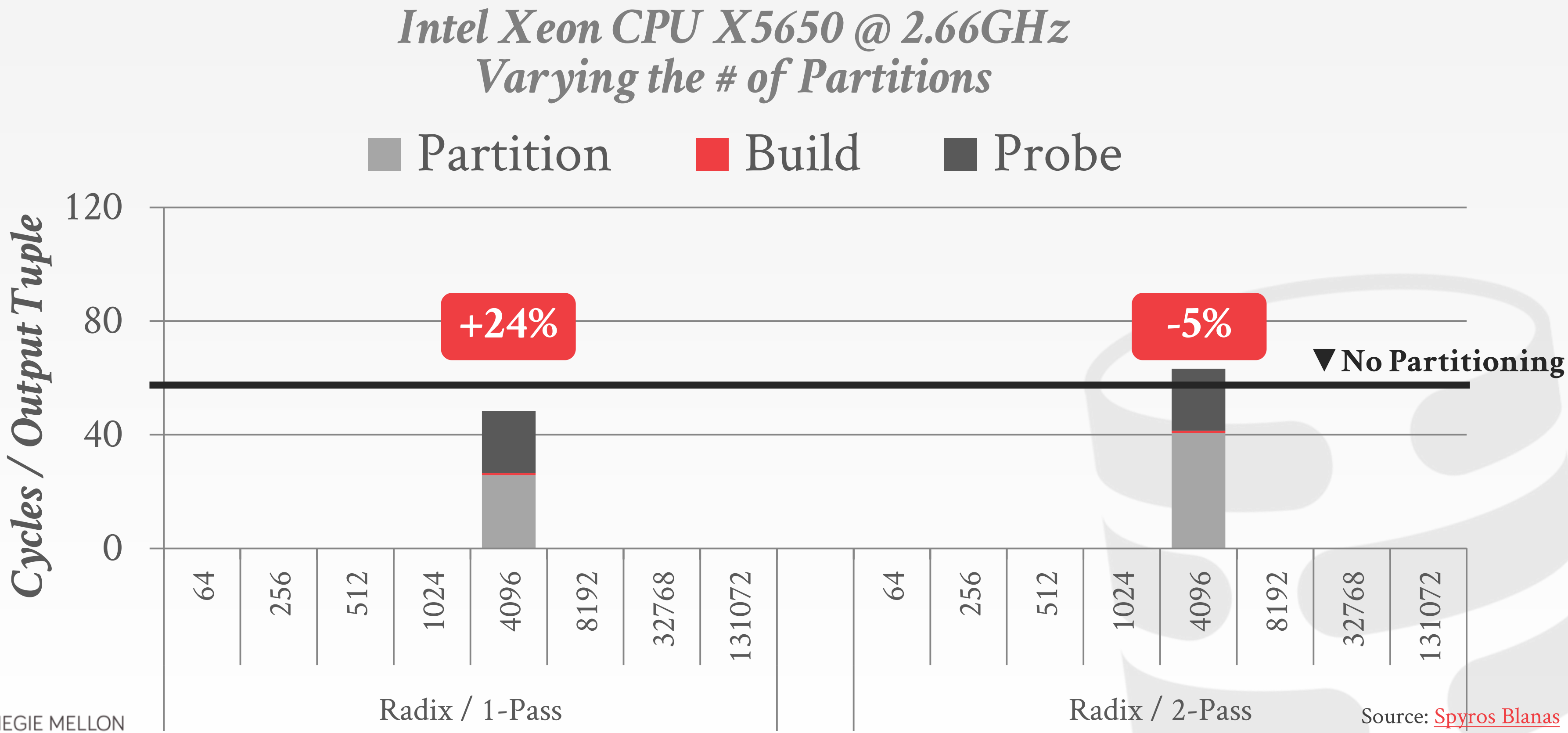
OBSERVATION

We have ignored a lot of important parameters for all of these algorithms so far.

→ Whether to use partitioning or not?
→ How many partitions to use?
→ How many passes to take in partitioning phase?

In a real DBMS, the optimizer will select what it thinks are good values based on what it knows about the data (and maybe hardware).

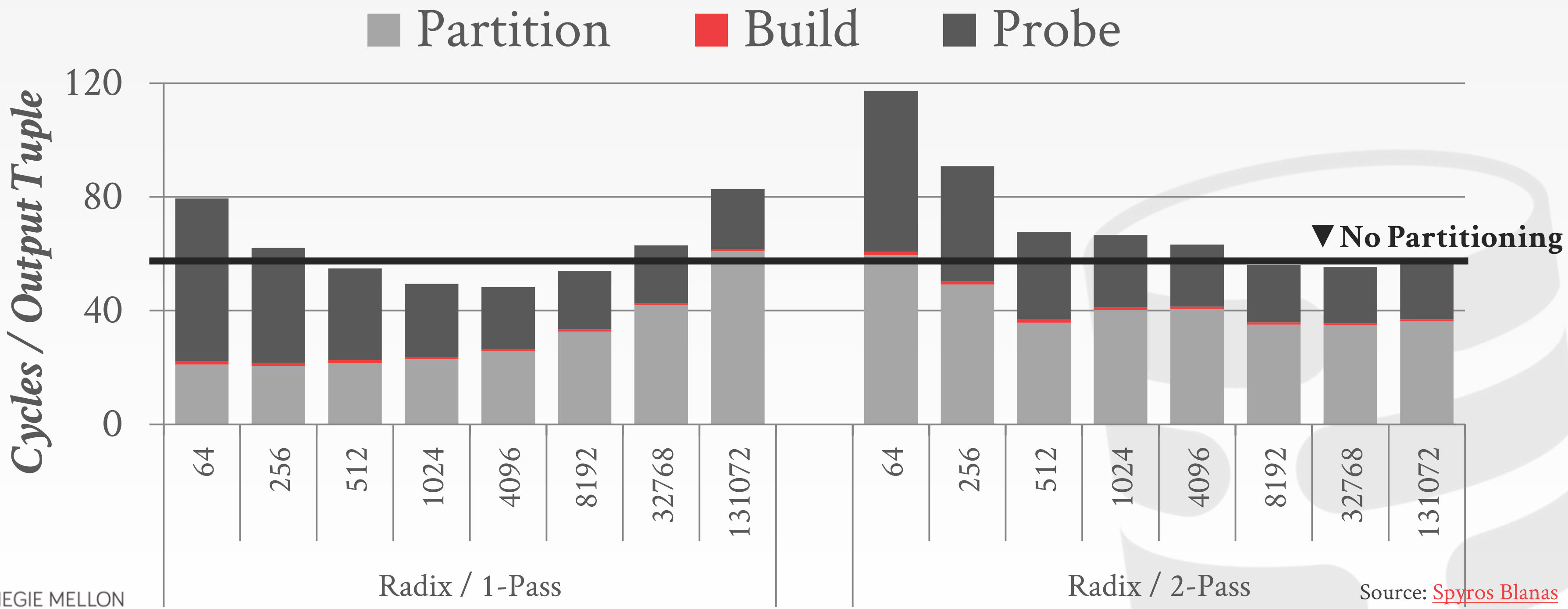
RADIX HASH JOIN – UNIFORM DATA SET



Source: Spyros Blanas

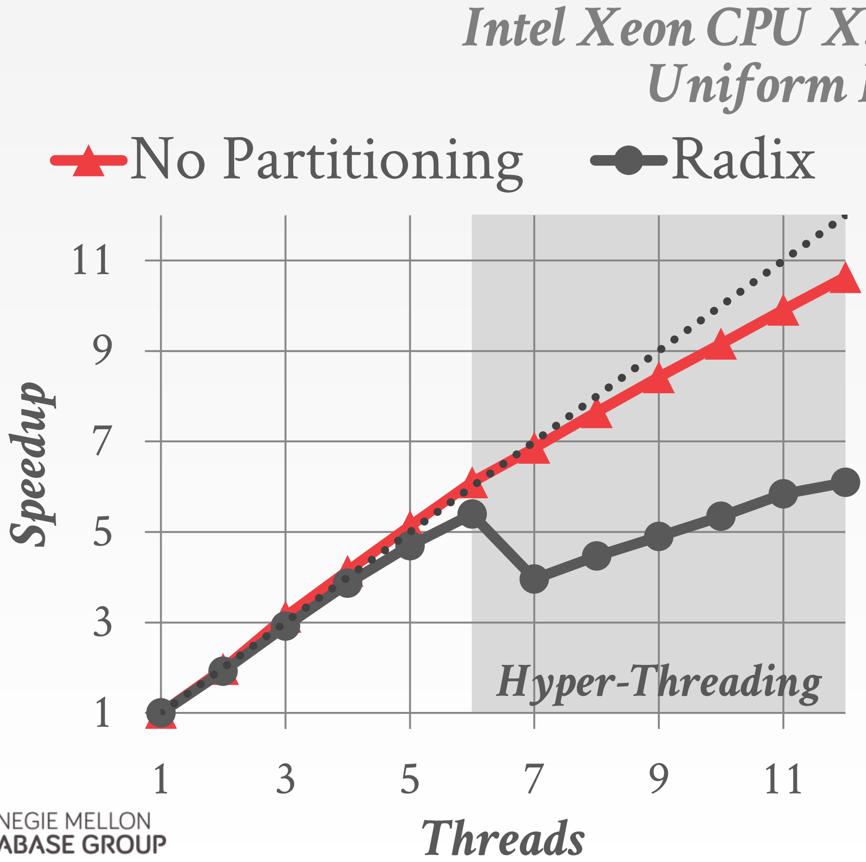
RADIX HASH JOIN – UNIFORM DATA SET

Intel Xeon CPU X5650 @ 2.66GHz
Varying the # of Partitions



Source: Spyros Blanas

EFFECTS OF HYPER-THREADING



Multi-threading hides cache & TLB miss latency.

Radix join has fewer cache & TLB misses but this has marginal benefit.

Non-partitioned join relies on multi-threading for high performance.

Source: Spyros Blanas

PARTING THOUGHTS

On modern CPUs, a simple hash join algorithm that does not partition inputs is competitive.

There are additional vectorization execution optimizations that are possible in hash joins that we didn't talk about. But these don't really help…

NEXT CLASS

Parallel Sort-Merge Joins