15-721 DATABASE SYSTEMS

Lecture #10 – Join Algorithms (Hashing)

Andy Pavlo // Carnegie Mellon University // Spring 2016

TODAY'S AGENDA

Parallel Hash Join

Hash Functions

Hash Table Implementations

Evaluation

PARALLEL HASH JOINS

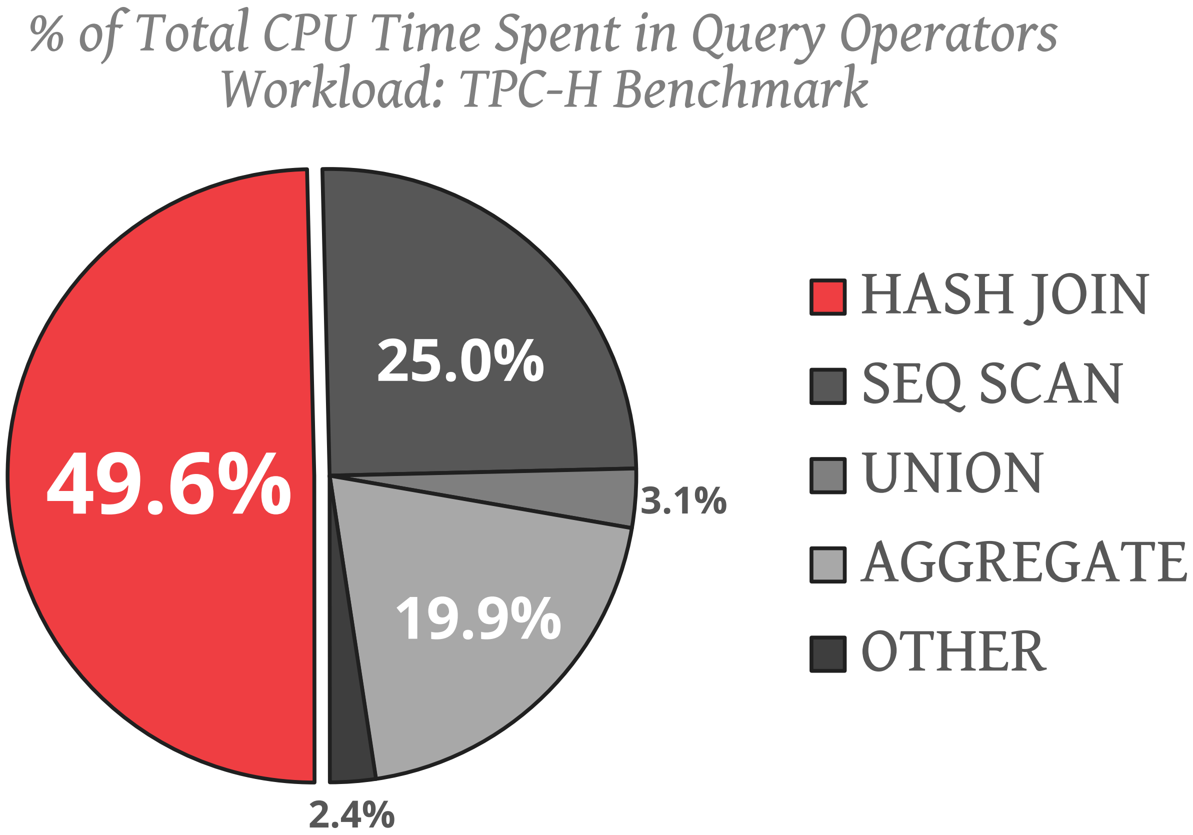
Hash join is the most important operator in a DBMS for OLAP workloads.

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

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

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

CLOUDERA IMPALA



OBSERVATION

Some OLTP DBMSs don't implement hash join.

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

HASH JOIN DESIGN GOALS

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

Choice #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, Index Traversal):
→ Clustering to a cache line.
→ Execute more operations per cache line.

Random Access (Hash Join):
→ Partition data to fit in cache + TLB.

Source: Johannes Gehrke

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).

→ The hash function used for this phase should be different than the one used in the build phase.
→ Ideally the cost of partitioning is less than the cost of cache misses during build phase.

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.

NON-BLOCKING PARTITIONING

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

Approach #1: Shared Partitions
→ 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 each thread finishes.

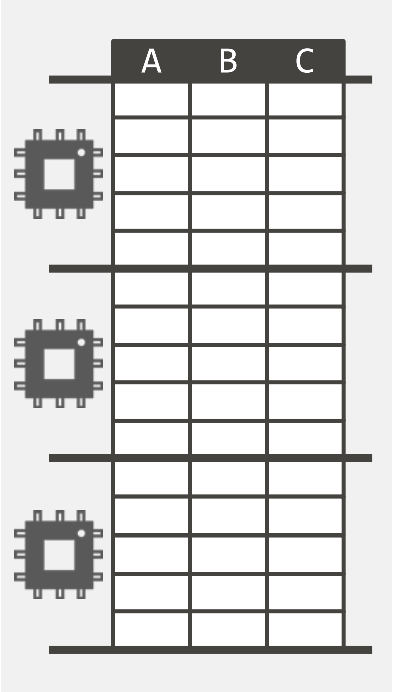
SHARED PARTITIONS

Data Table

A	B	C

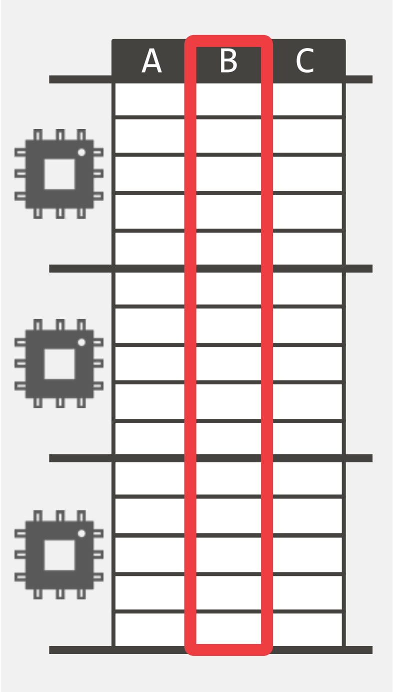
SHARED PARTITIONS

Data Table



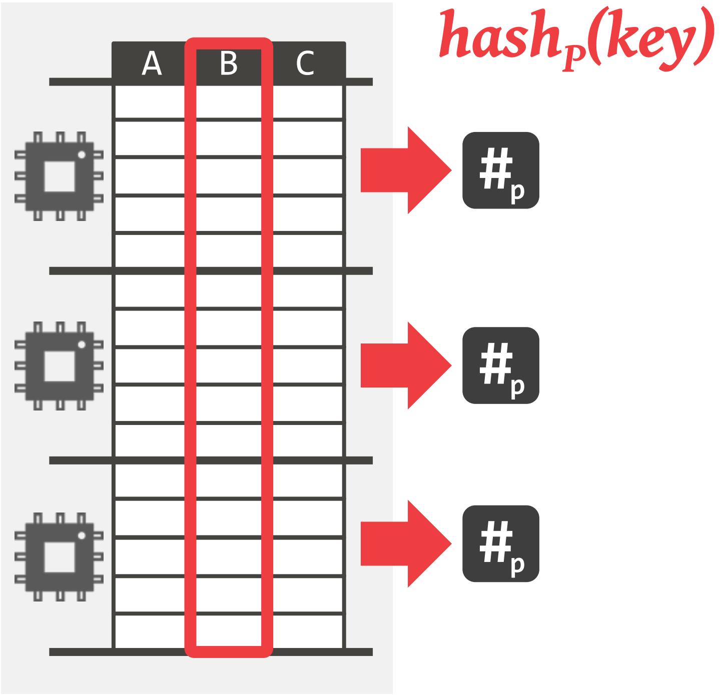
SHARED PARTITIONS

Data Table

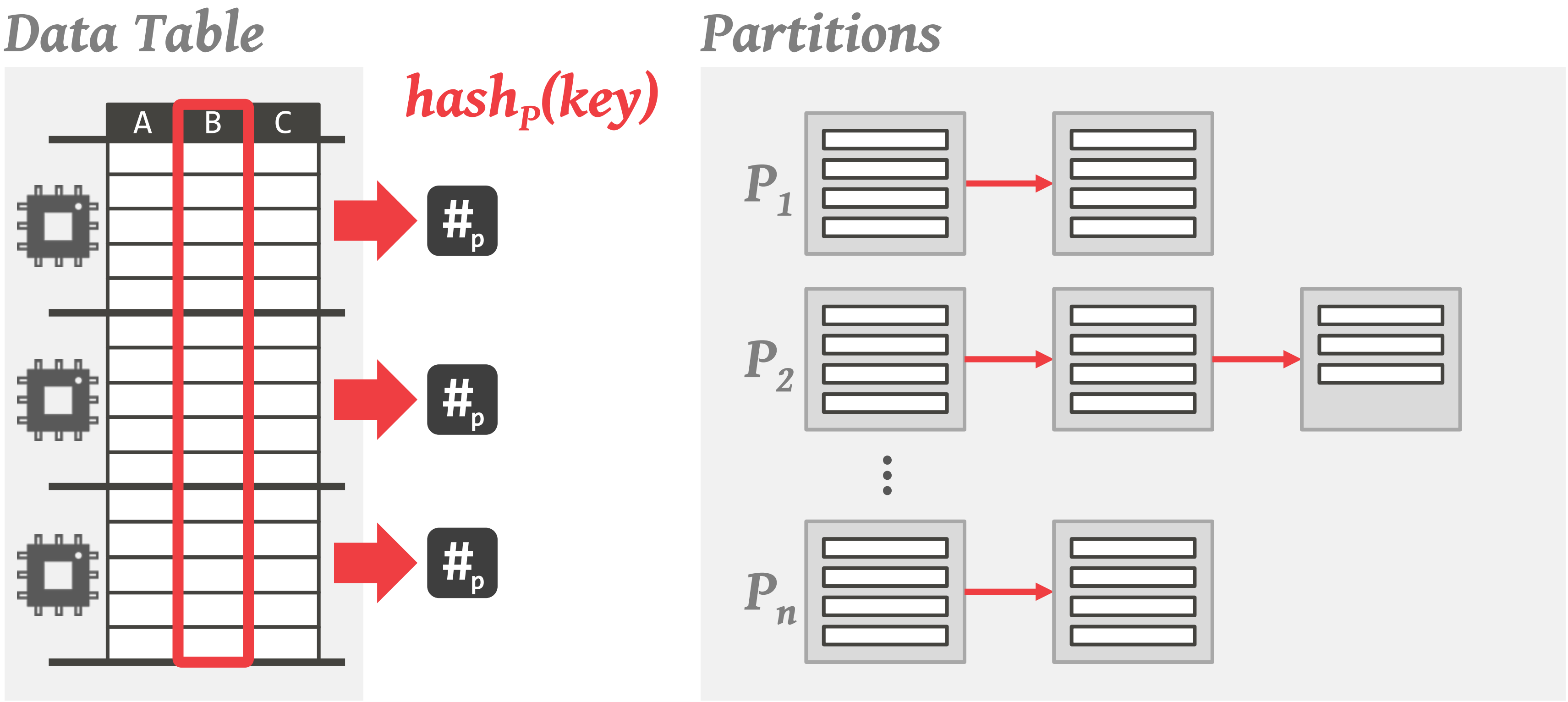


SHARED PARTITIONS

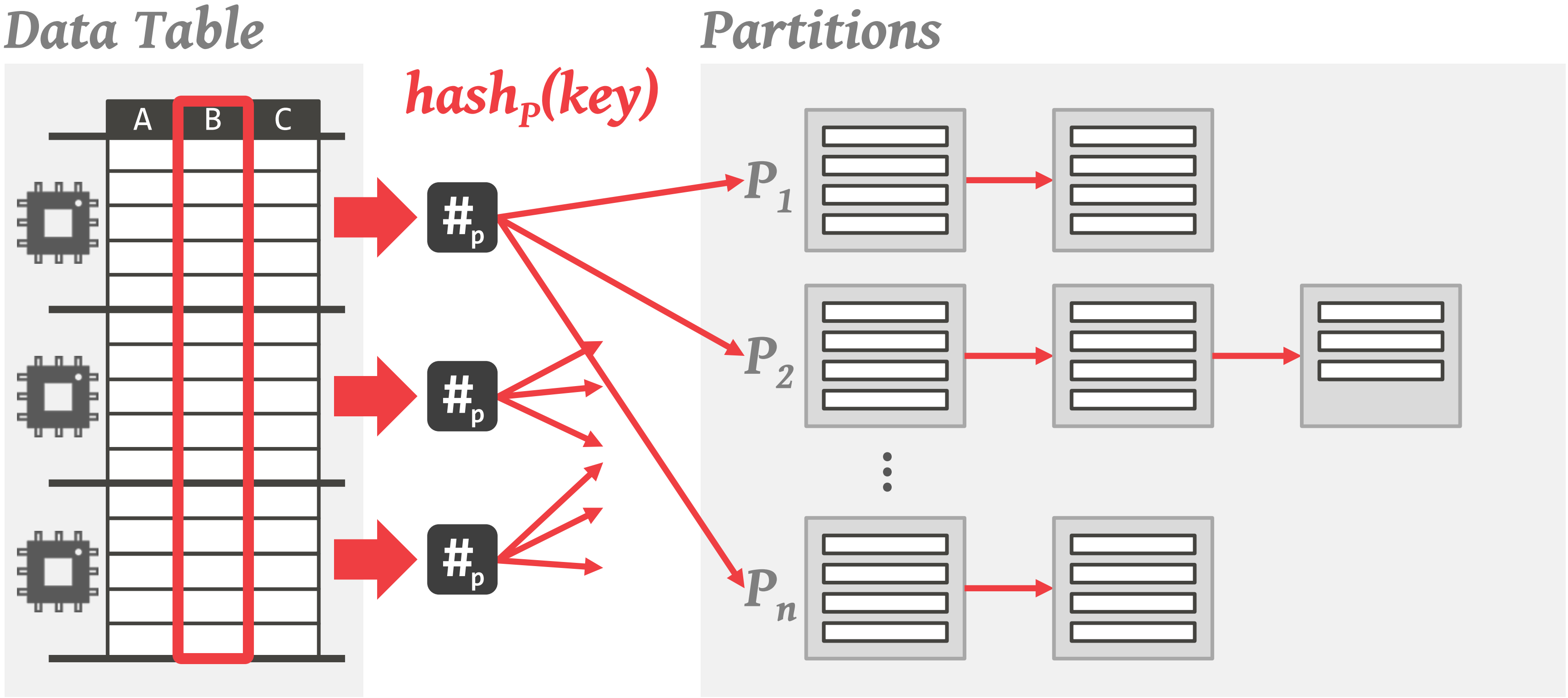
Data Table



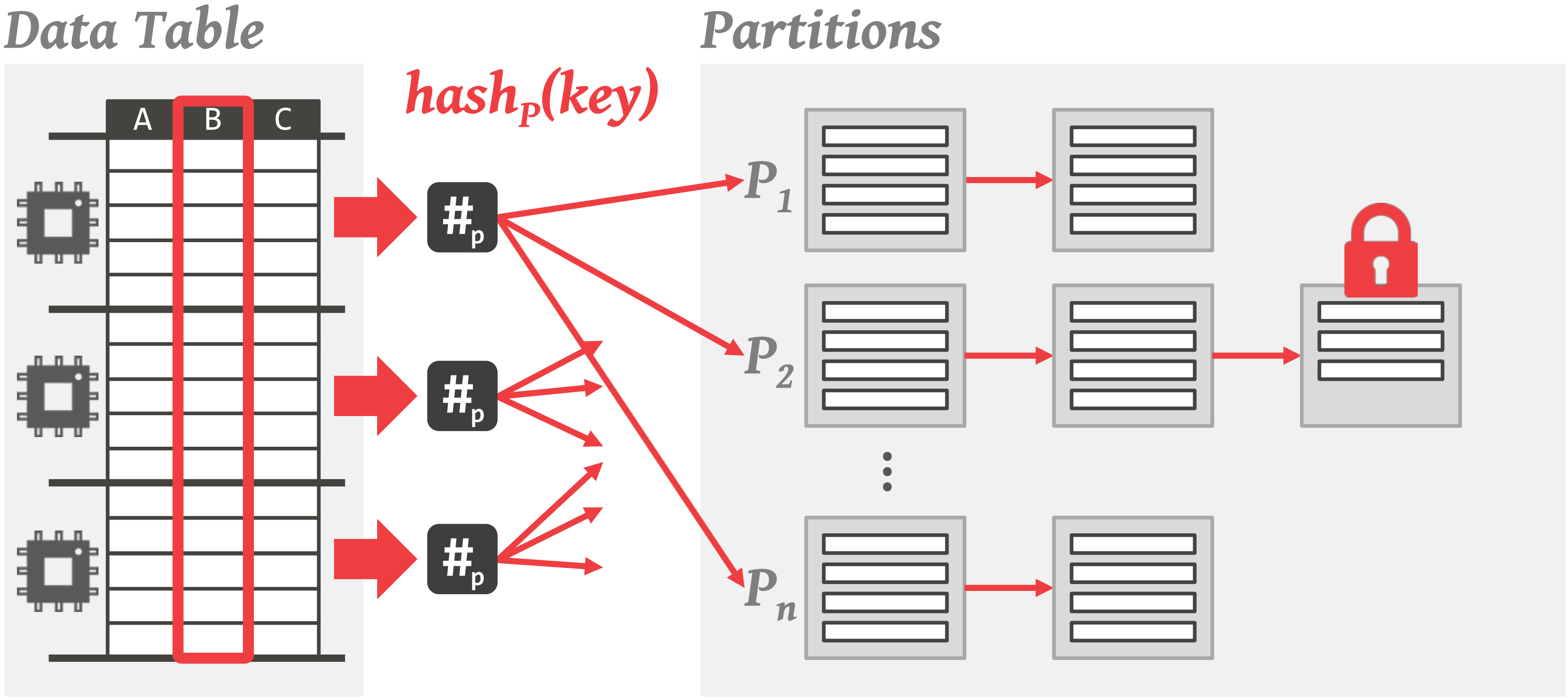
SHARED PARTITIONS



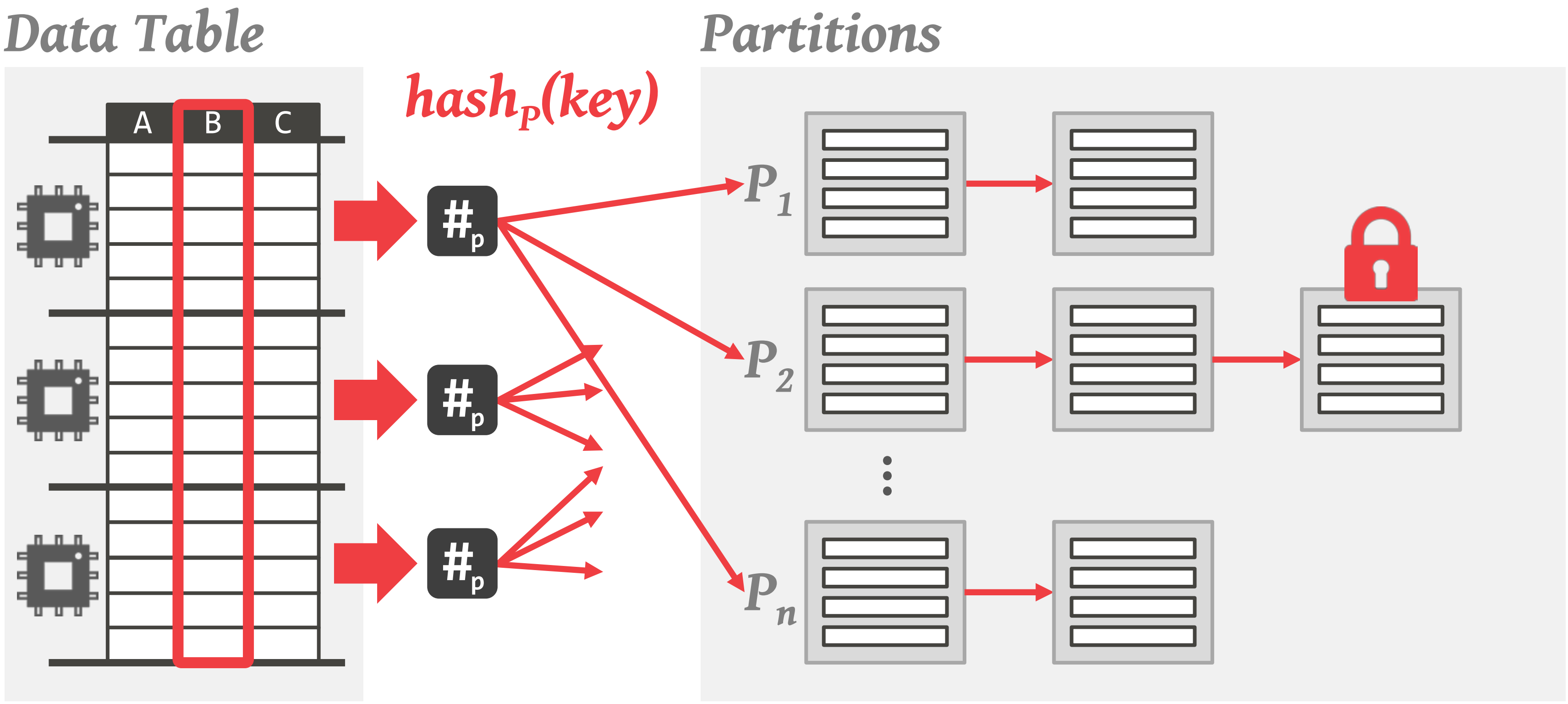
SHARED PARTITIONS



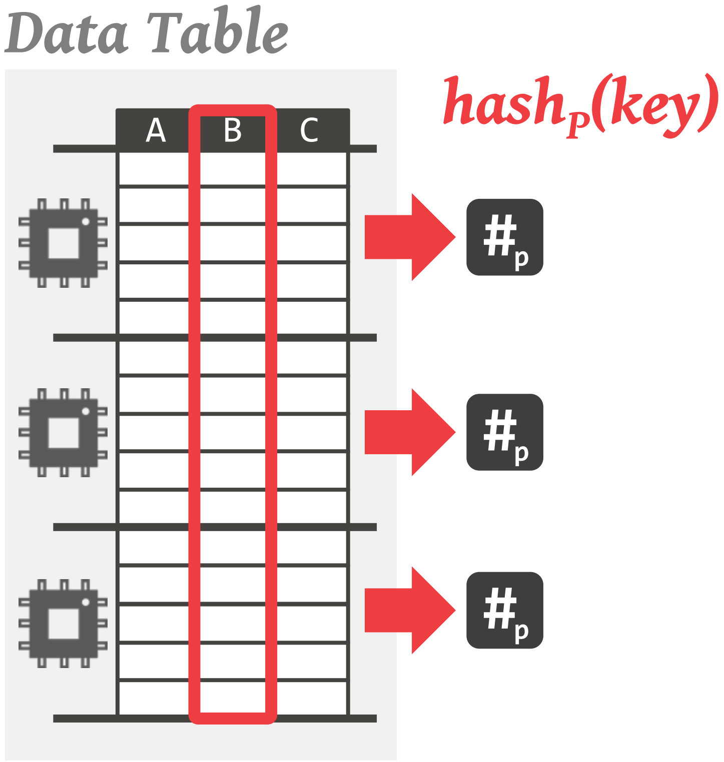
SHARED PARTITIONS



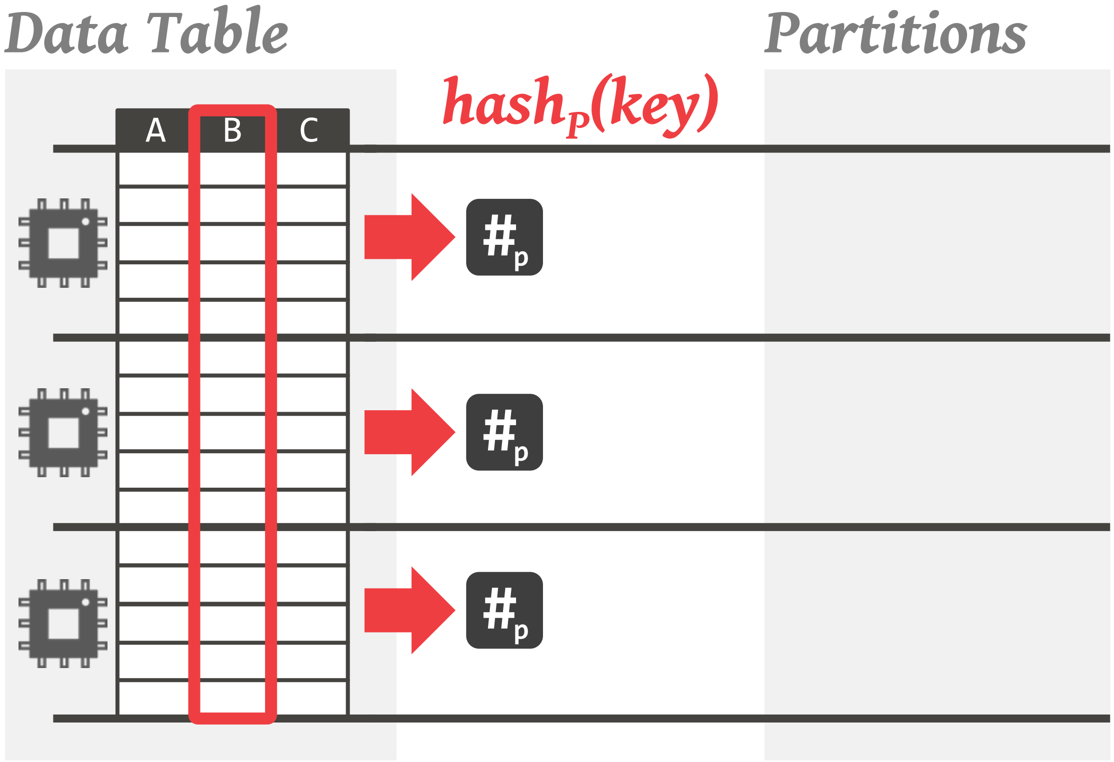
SHARED PARTITIONS



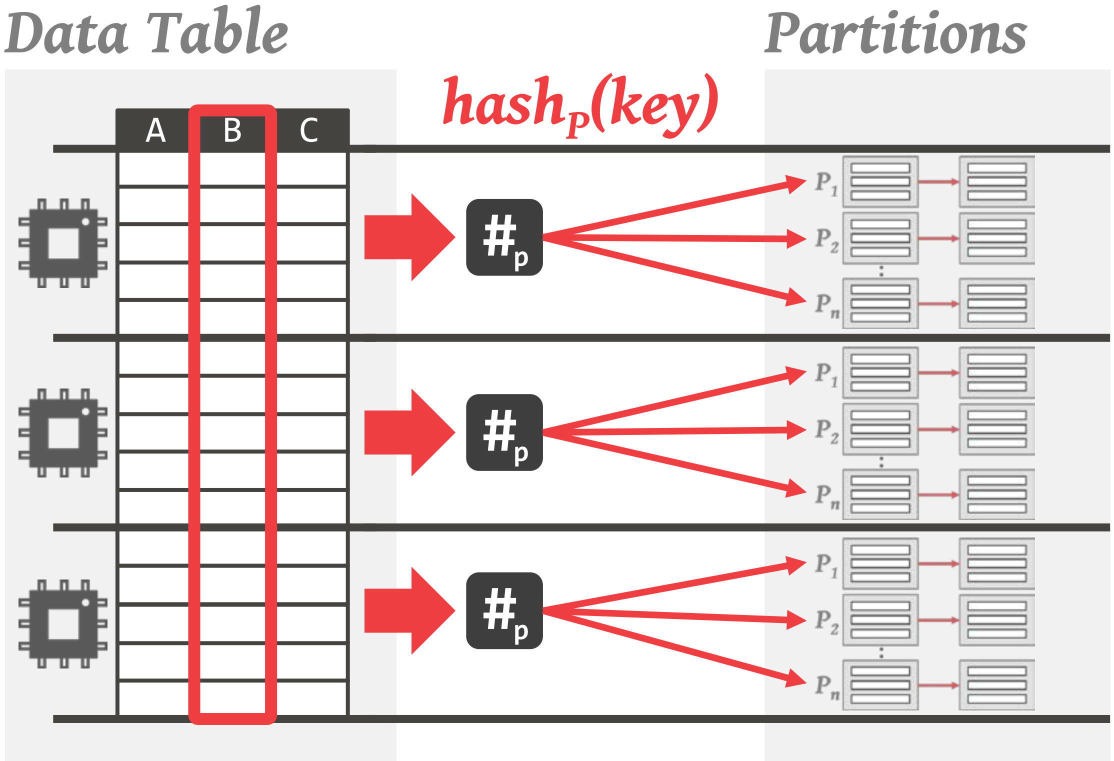
PRIVATE PARTITIONS



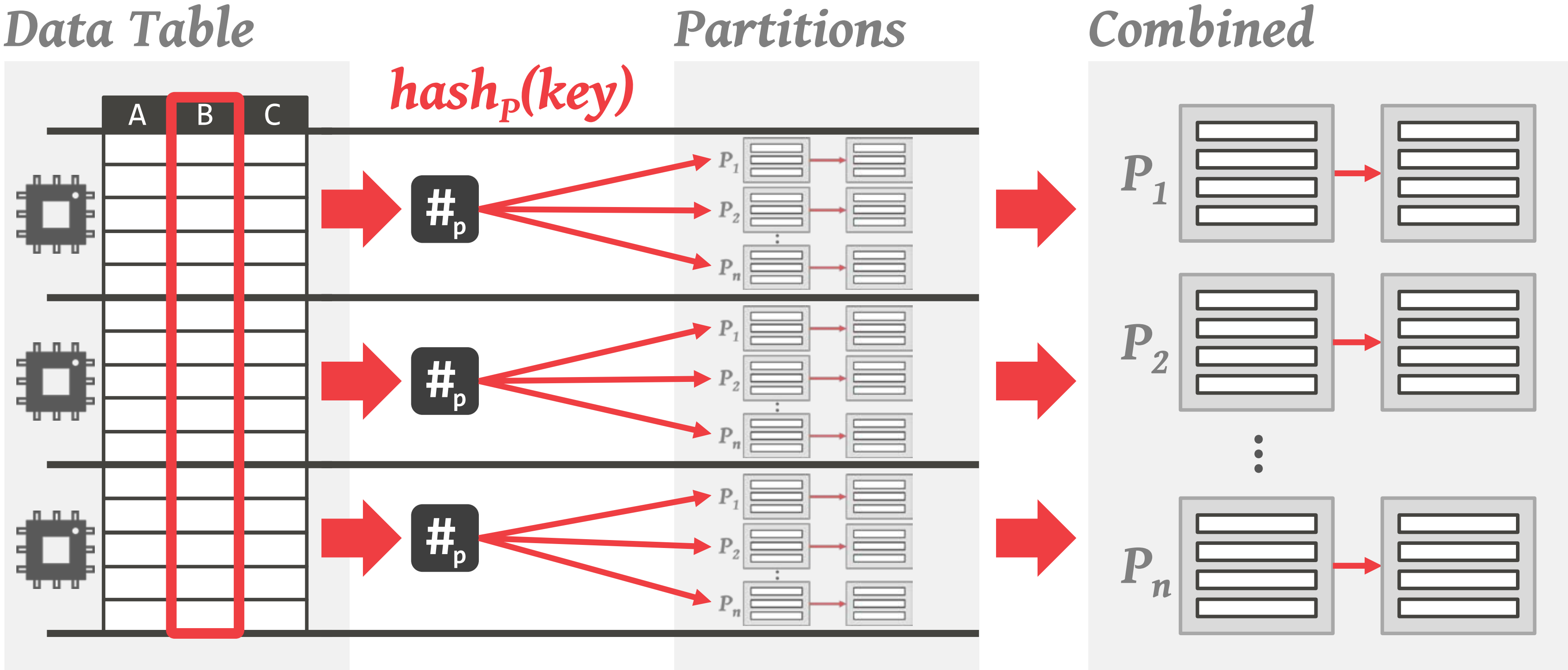
PRIVATE PARTITIONS



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

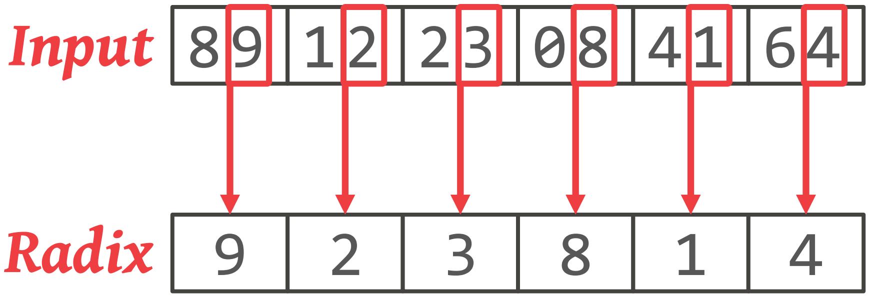
RADIX

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

Input	89	12	23	08	41	64

RADIX

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



RADIX

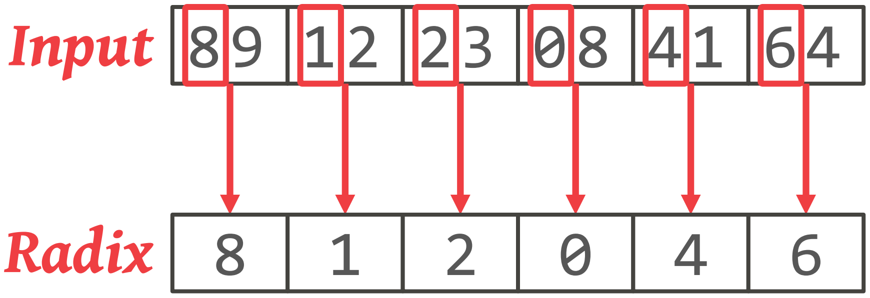
The radix is the value of an integer at a particular position (using its base).

Input	89	12	23	08	41	64

Radix

RADIX

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



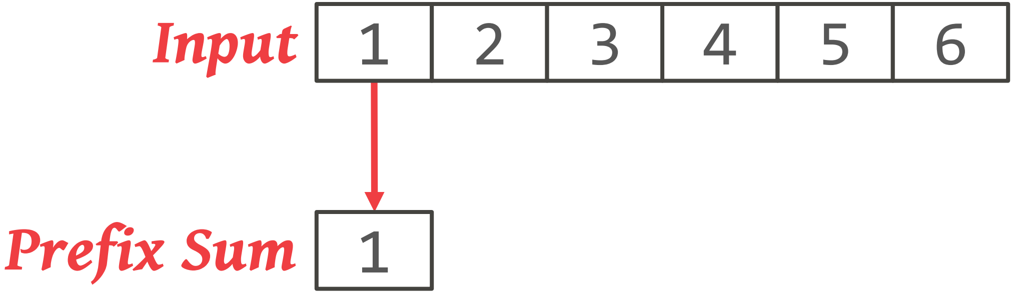
PREFIX SUM

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.

Input	1	2	3	4	5	6

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

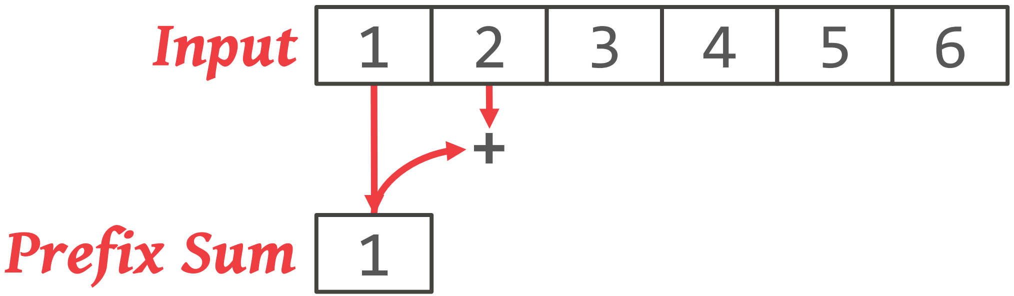
The prefix sum of a sequence of numbers

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

is a second sequence of numbers

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

that is a running total of the input sequence.



PREFIX SUM

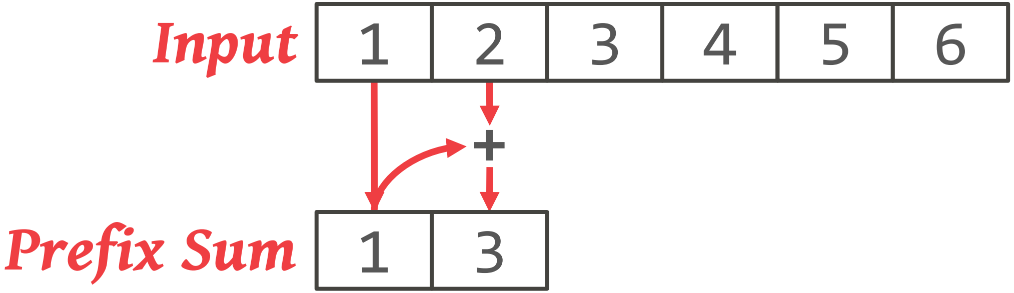
The prefix sum of a sequence of numbers

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

is a second sequence of numbers

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

that is a running total of the input sequence.



PREFIX SUM

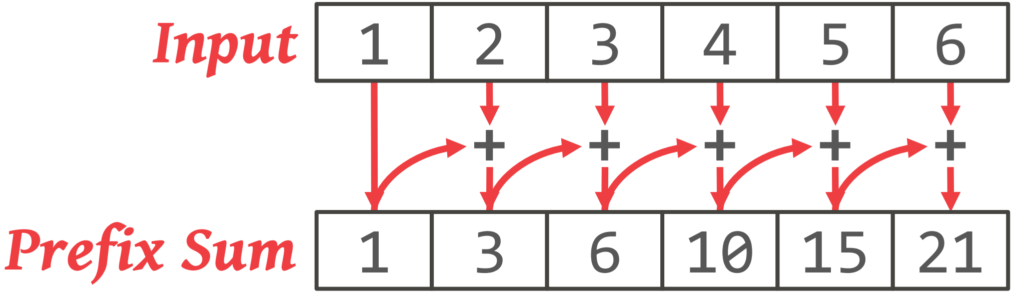
The prefix sum of a sequence of numbers

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

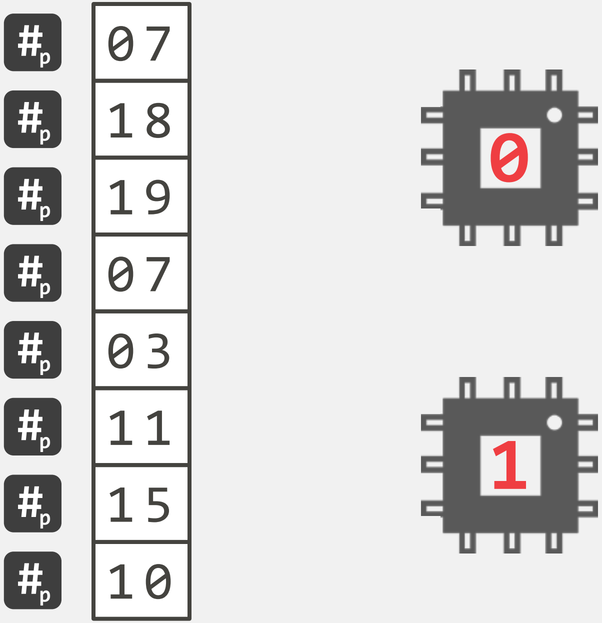
is a second sequence of numbers

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

that is a running total of the input sequence.



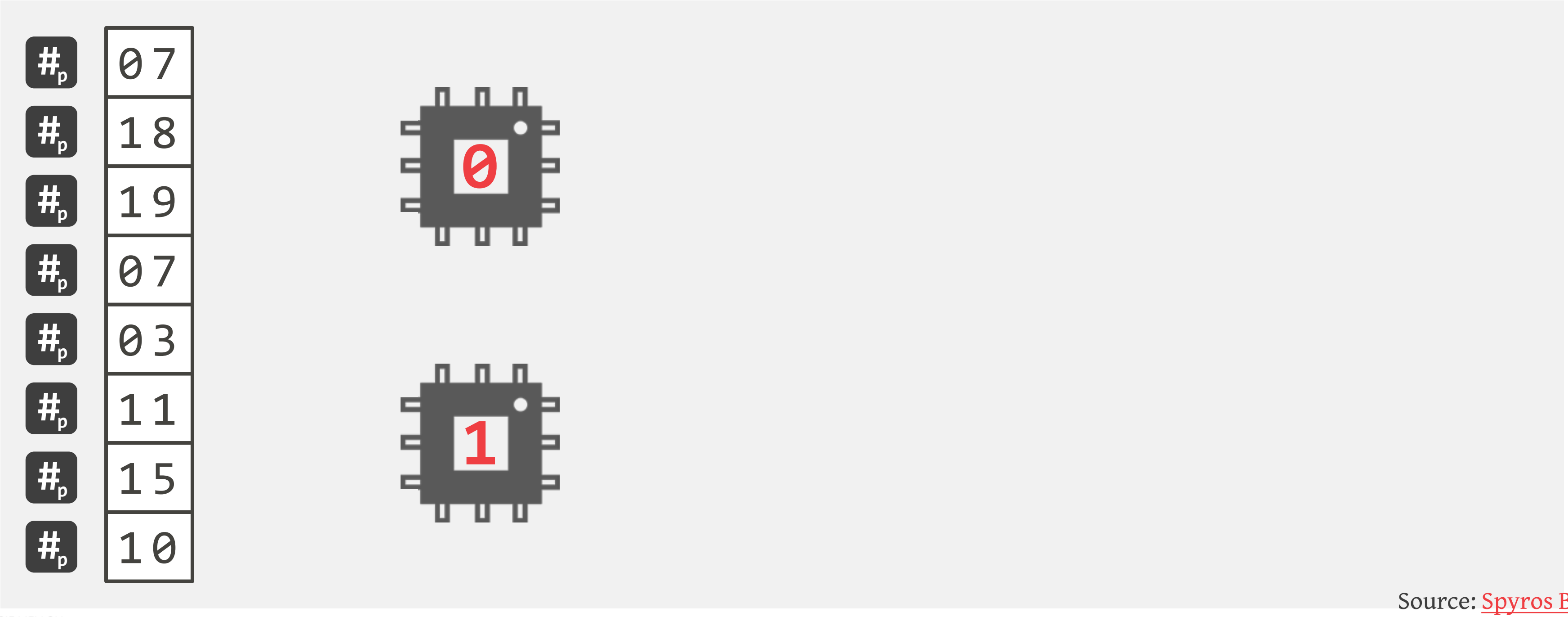
RADIX PARTITIONS



Source: Spyros Blanas

RADIX PARTITIONS

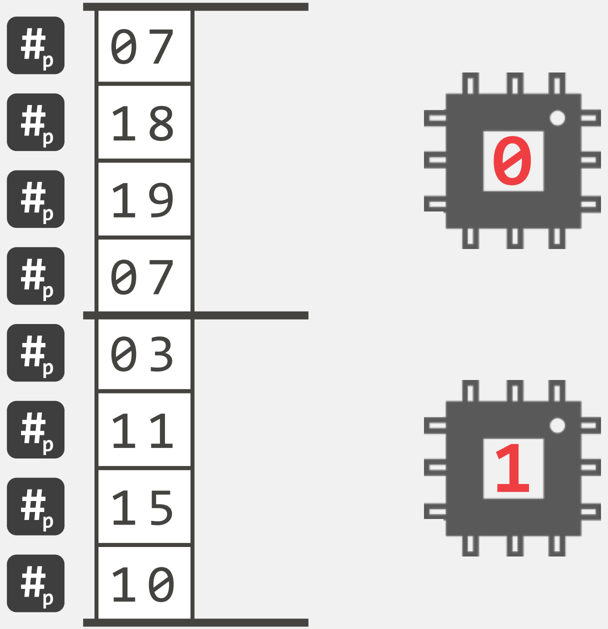
Step #1: Inspect input, create histograms



Source: Spyros Blanas

RADIX PARTITIONS

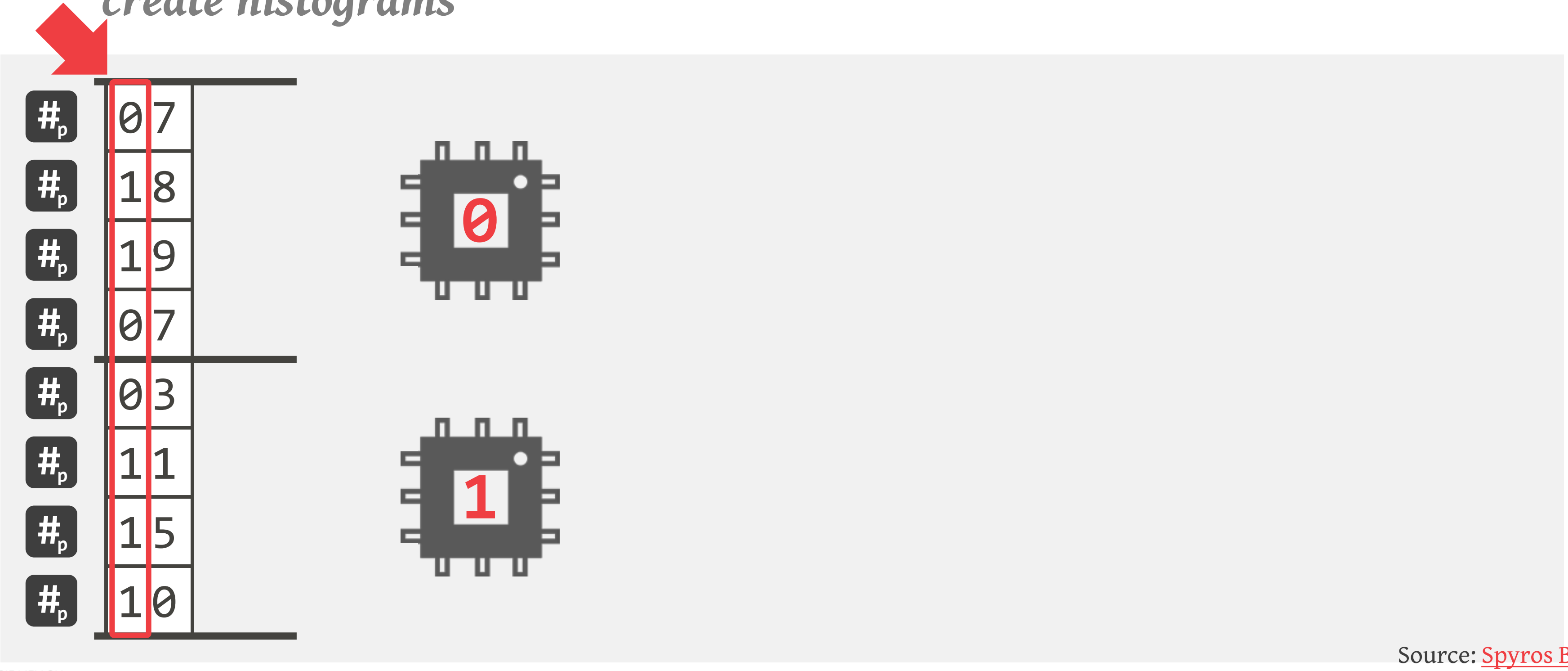
Step #1: Inspect input, create histograms



Source: Spyros Blanas

RADIX PARTITIONS

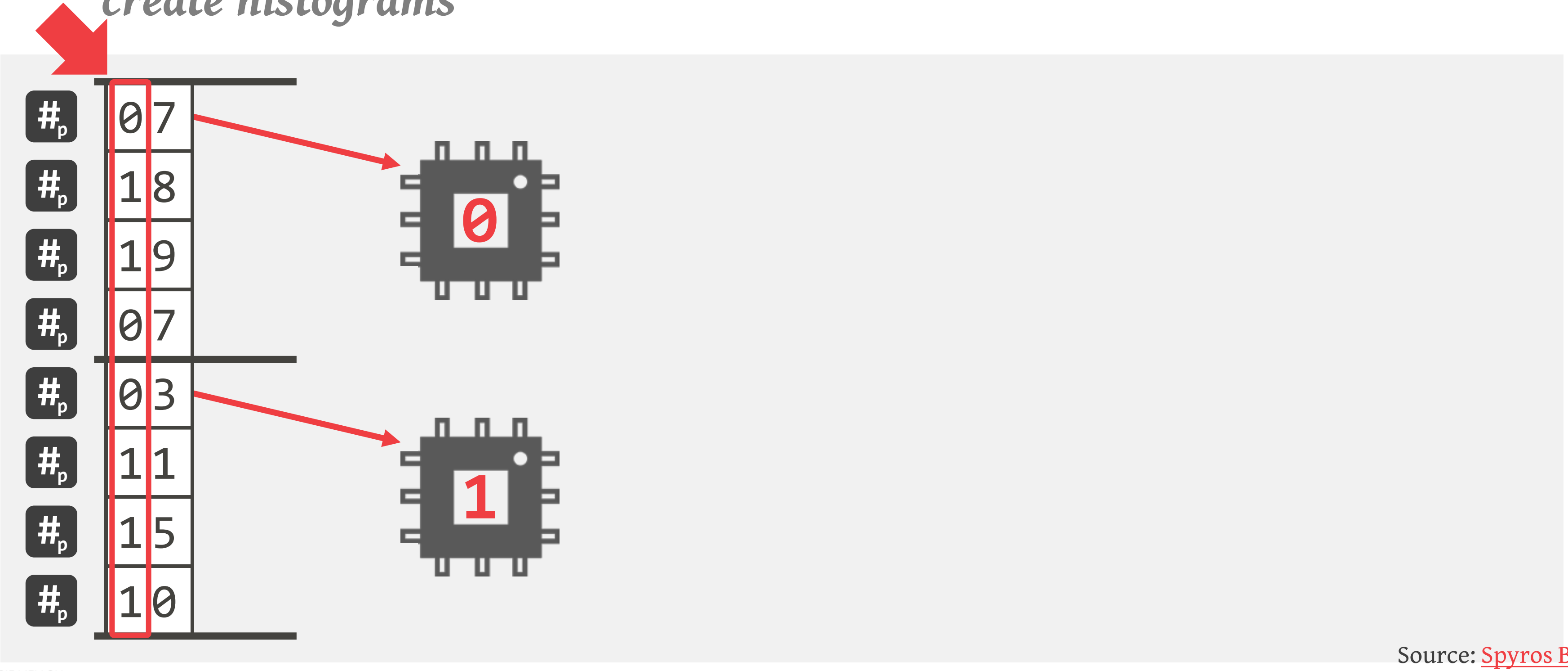
Step #1: Inspect input, create histograms



Source: Spyros Blanas

RADIX PARTITIONS

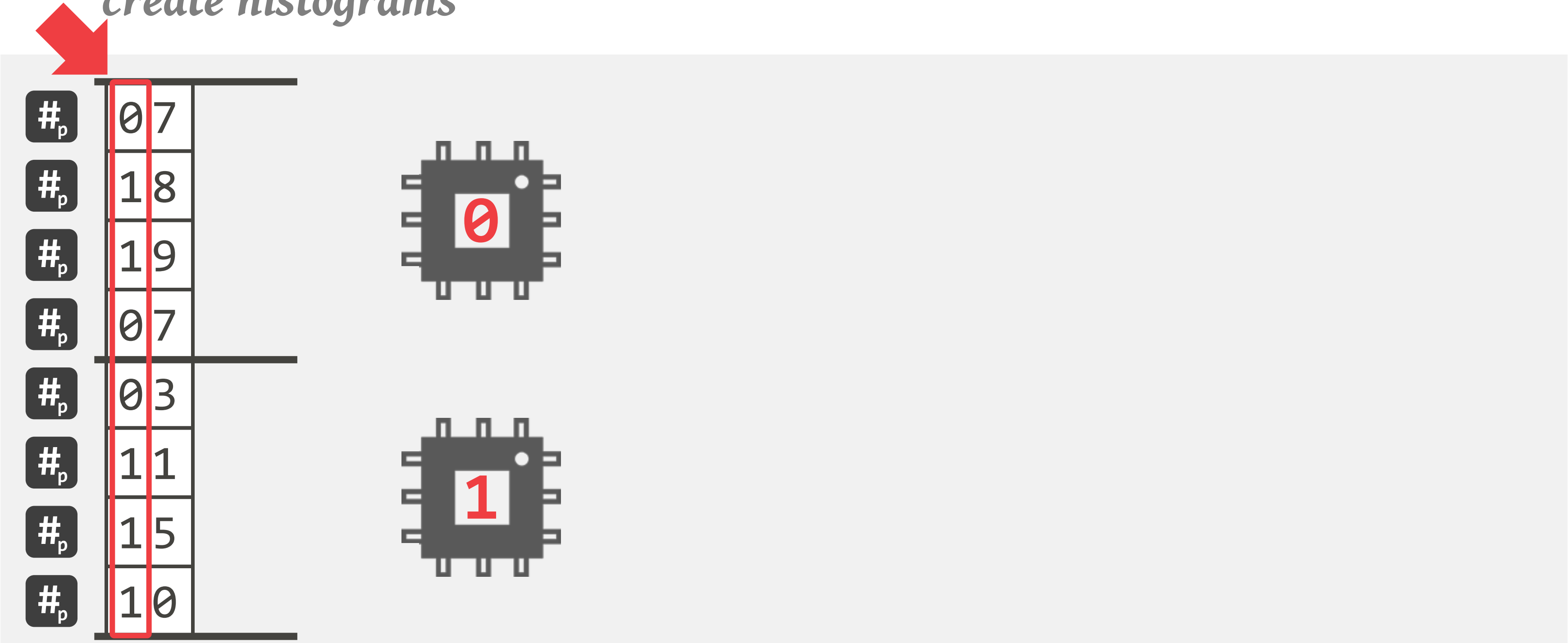
Step #1: Inspect input, create histograms



Source: Spyros Blanas

RADIX PARTITIONS

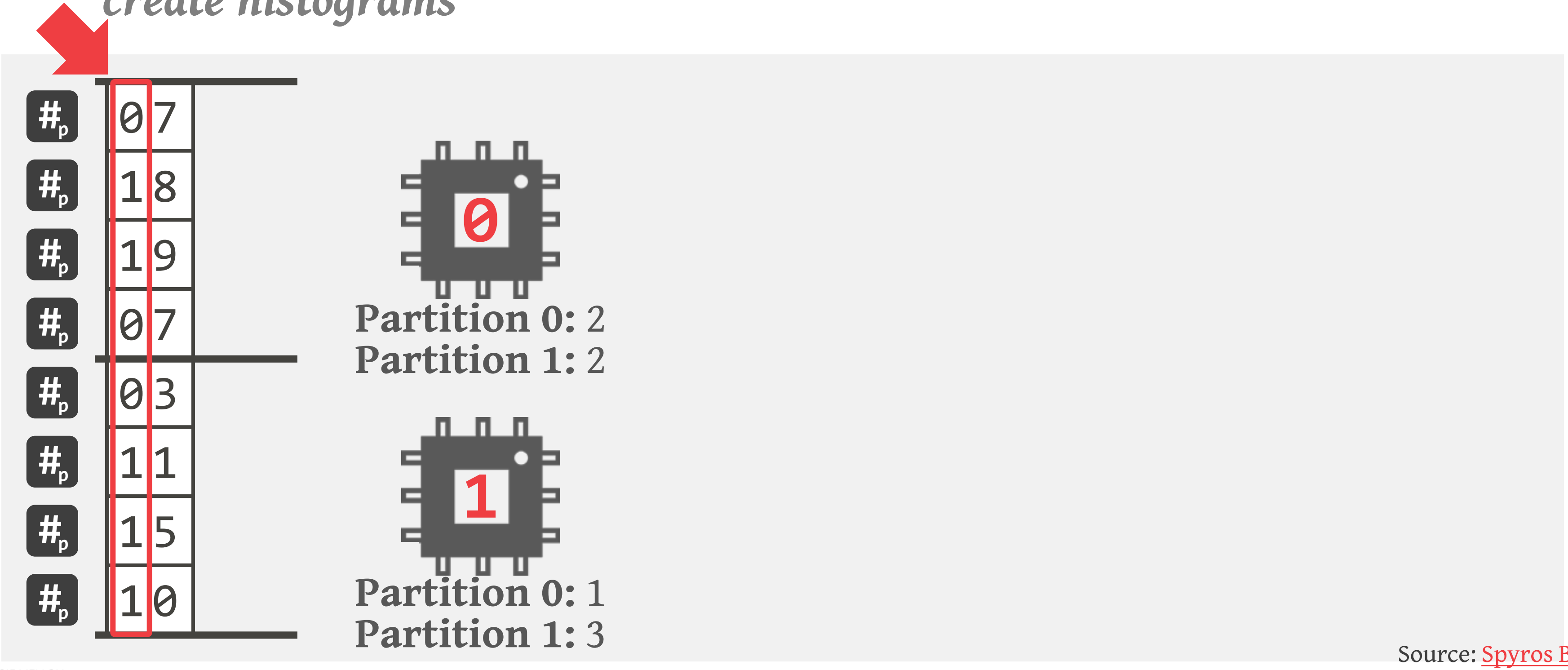
Step #1: Inspect input, create histograms



Source: Spyros Blanas

RADIX PARTITIONS

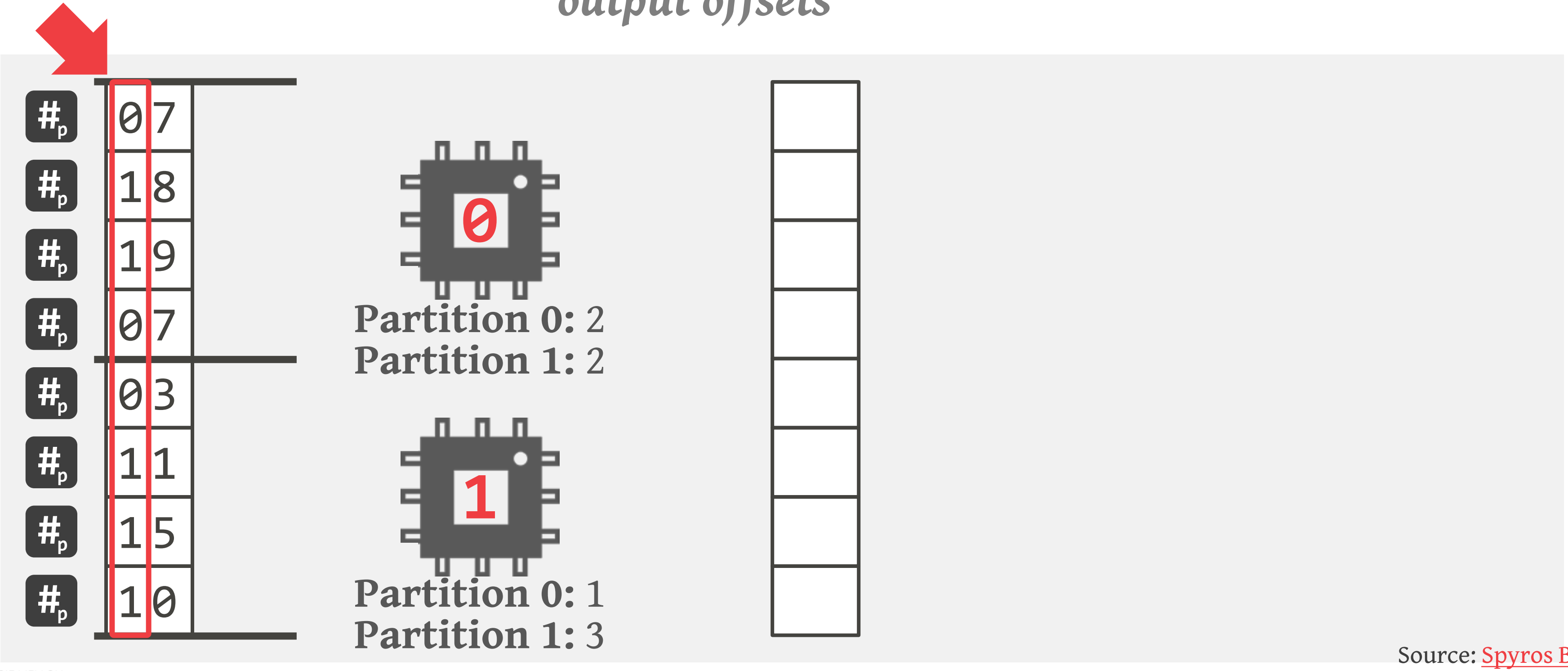
Step #1: Inspect input, create histograms



Source: [Spyros Blanas]

RADIX PARTITIONS

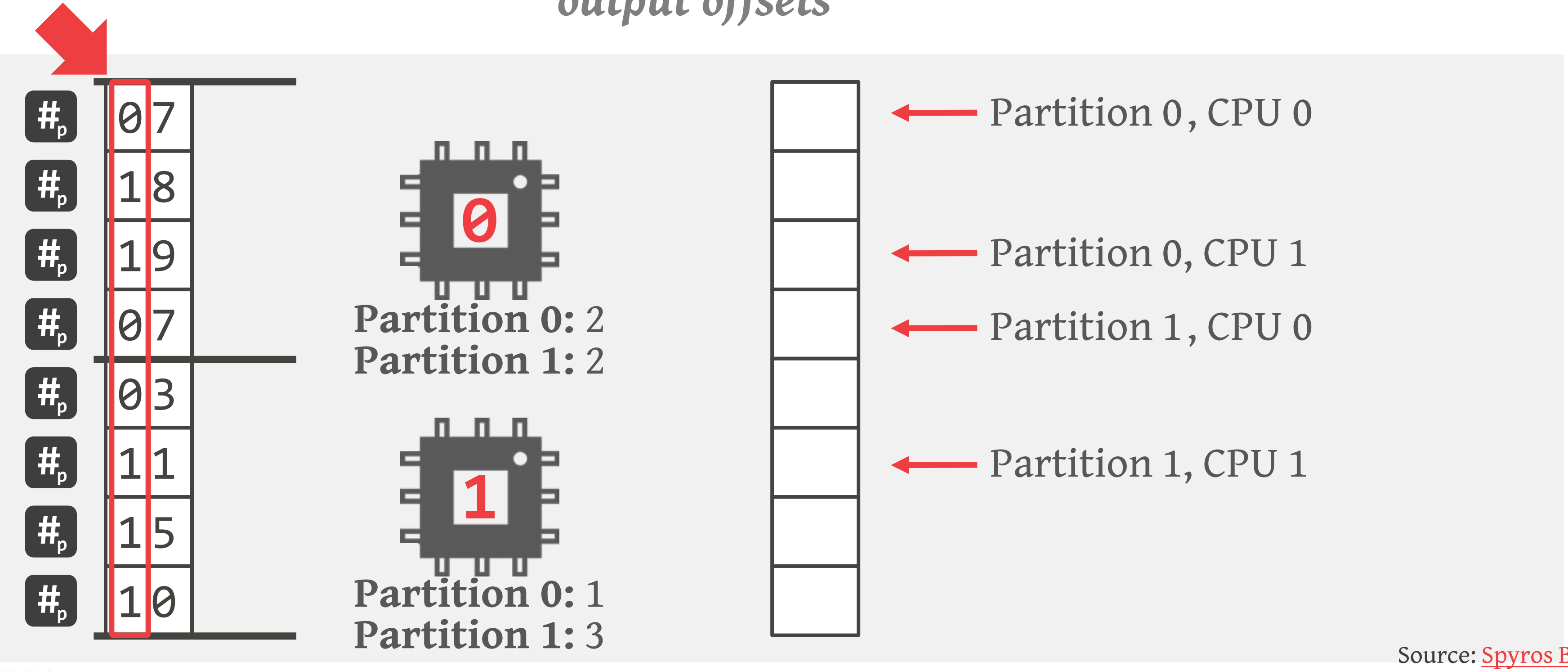
Step #2: Compute output offsets



Source: Spyros Blanas

RADIX PARTITIONS

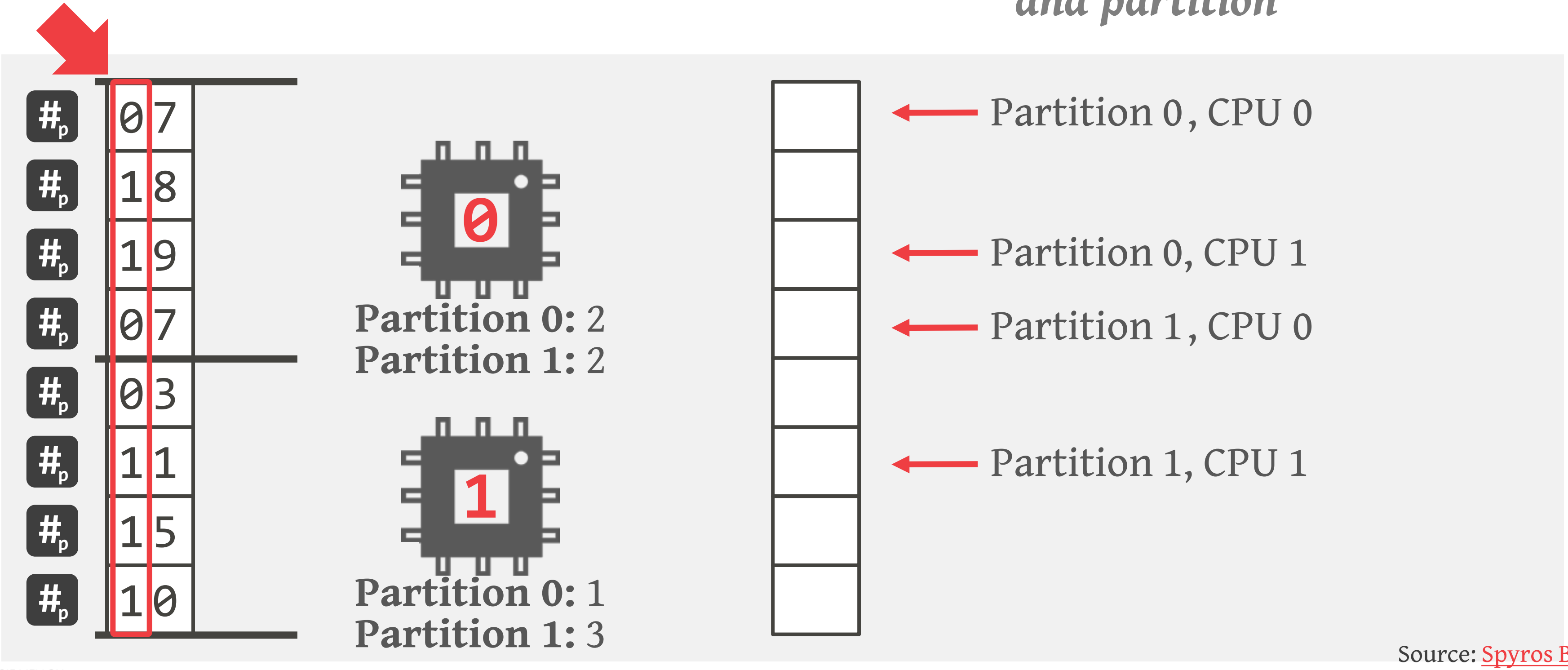
Step #2: Compute output offsets



Source: Spyros Blanas

RADIX PARTITIONS

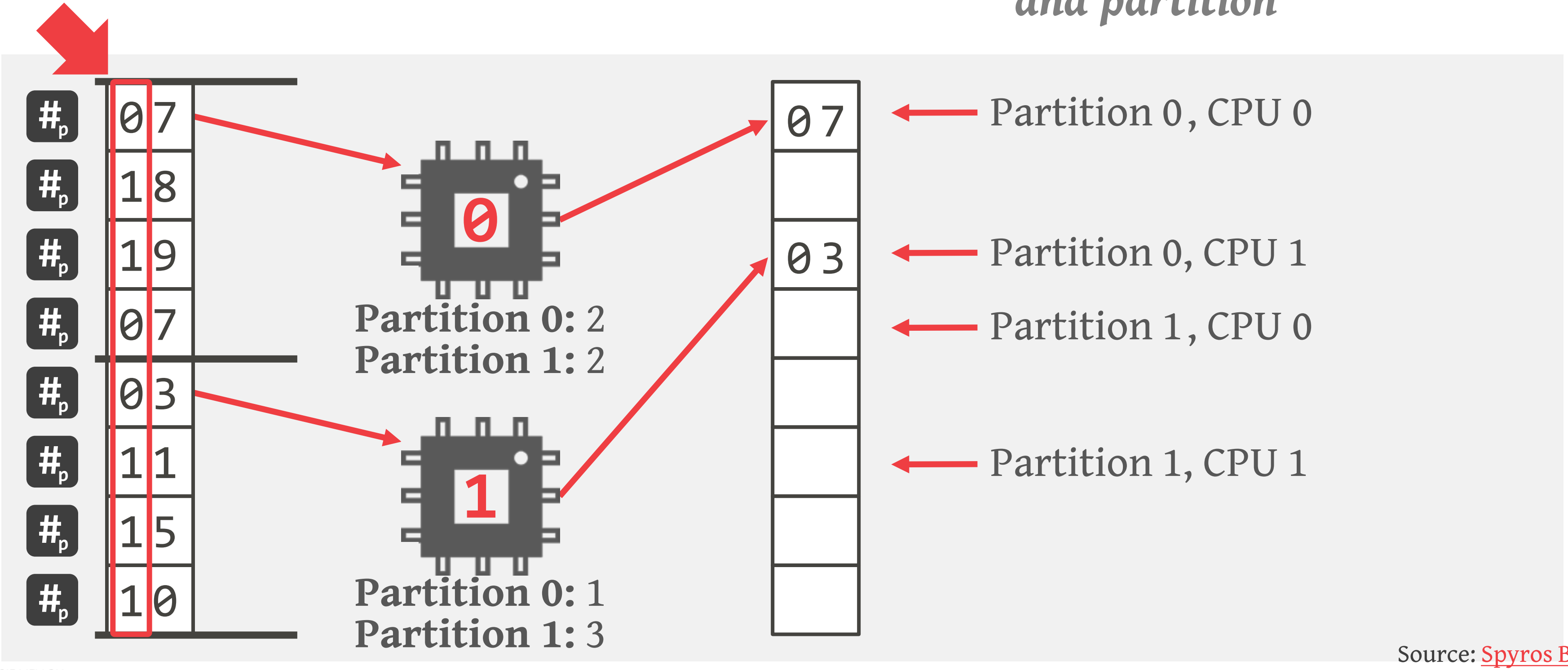
Step #3: Read input and partition



Source: Spyros Blanas

RADIX PARTITIONS

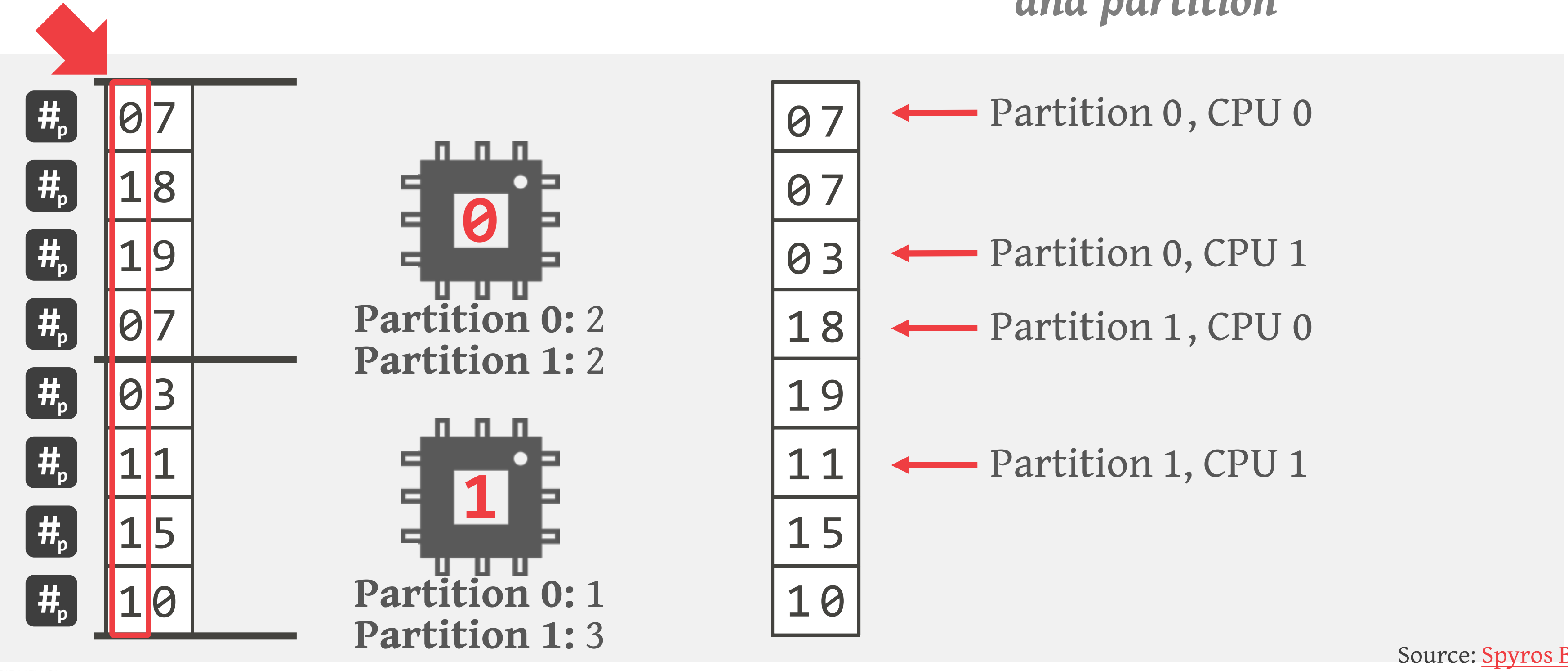
Step #3: Read input and partition



Source: Spyros Blanas

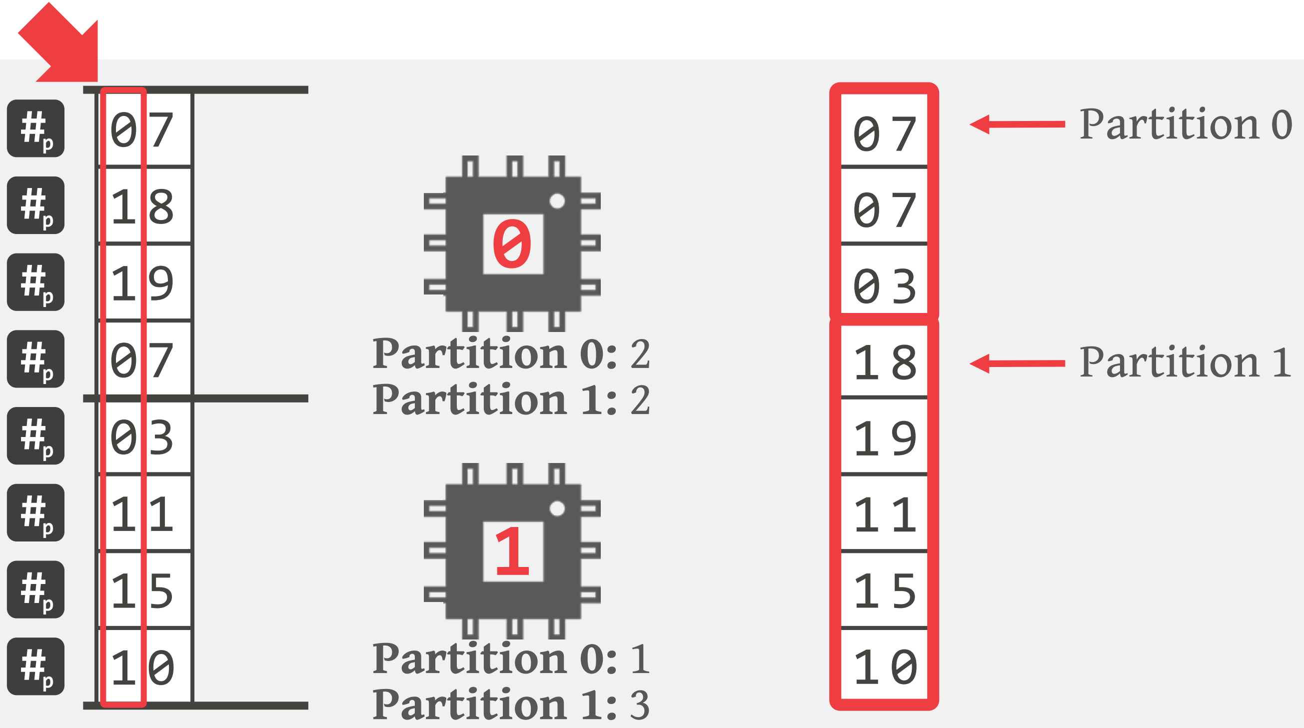
RADIX PARTITIONS

Step #3: Read input and partition



Source: Spyros Blanas

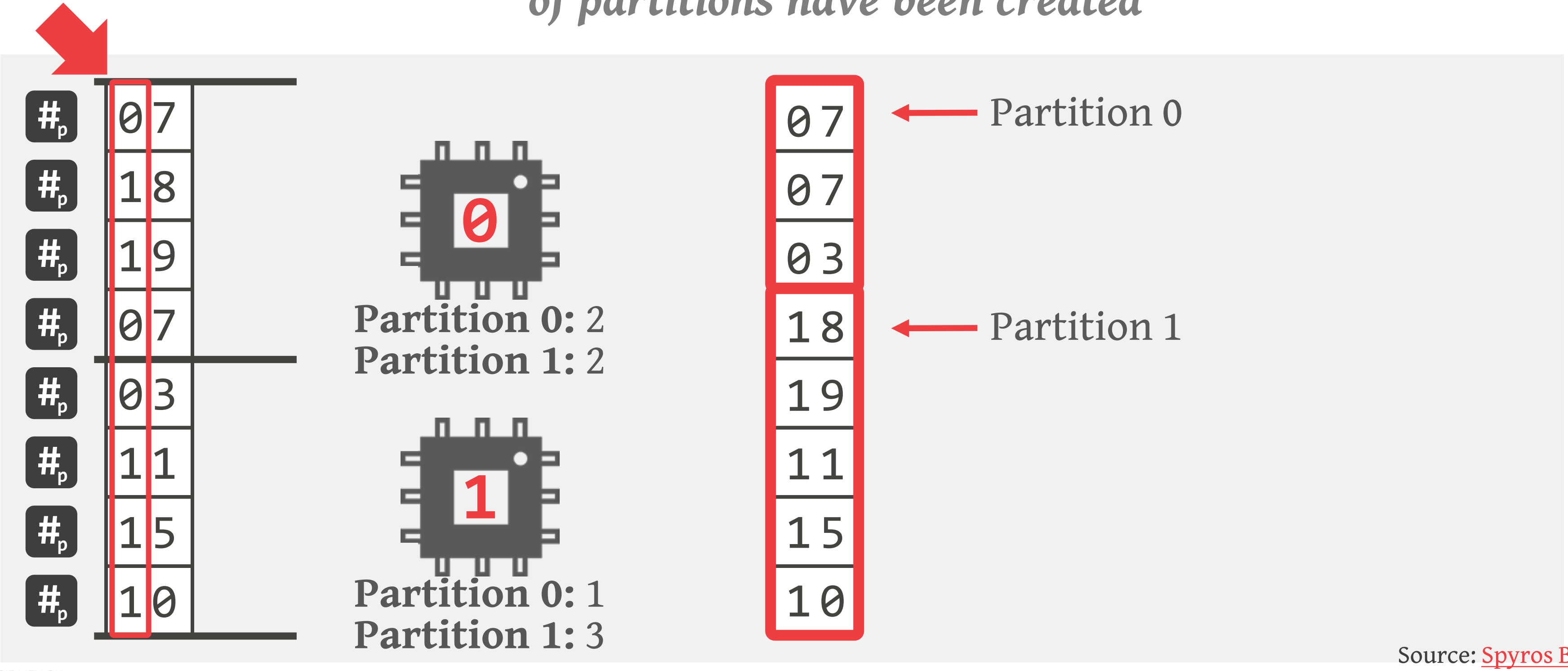
RADIX PARTITIONS



Source: Spyros Blanas

RADIX PARTITIONS

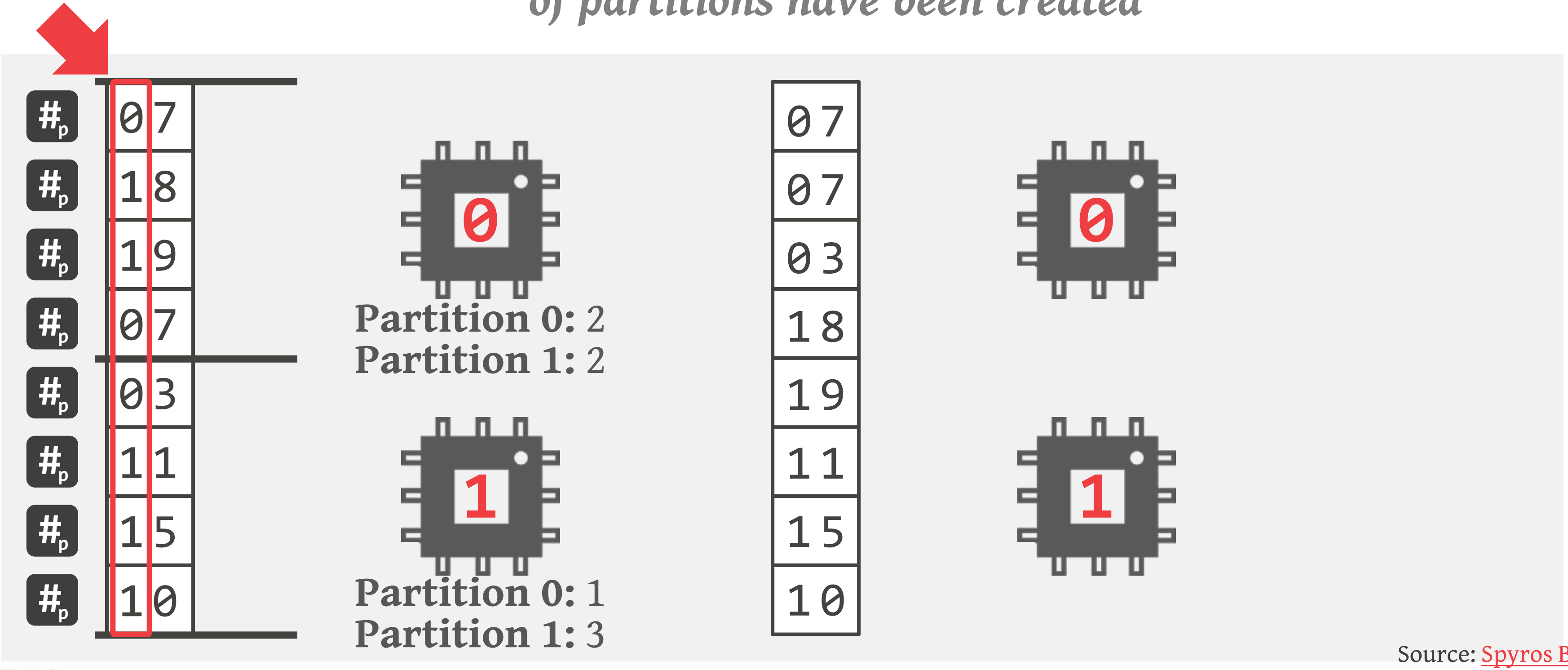
Recursively repeat until target number of partitions have been created



Source: Spyros Blanas

RADIX PARTITIONS

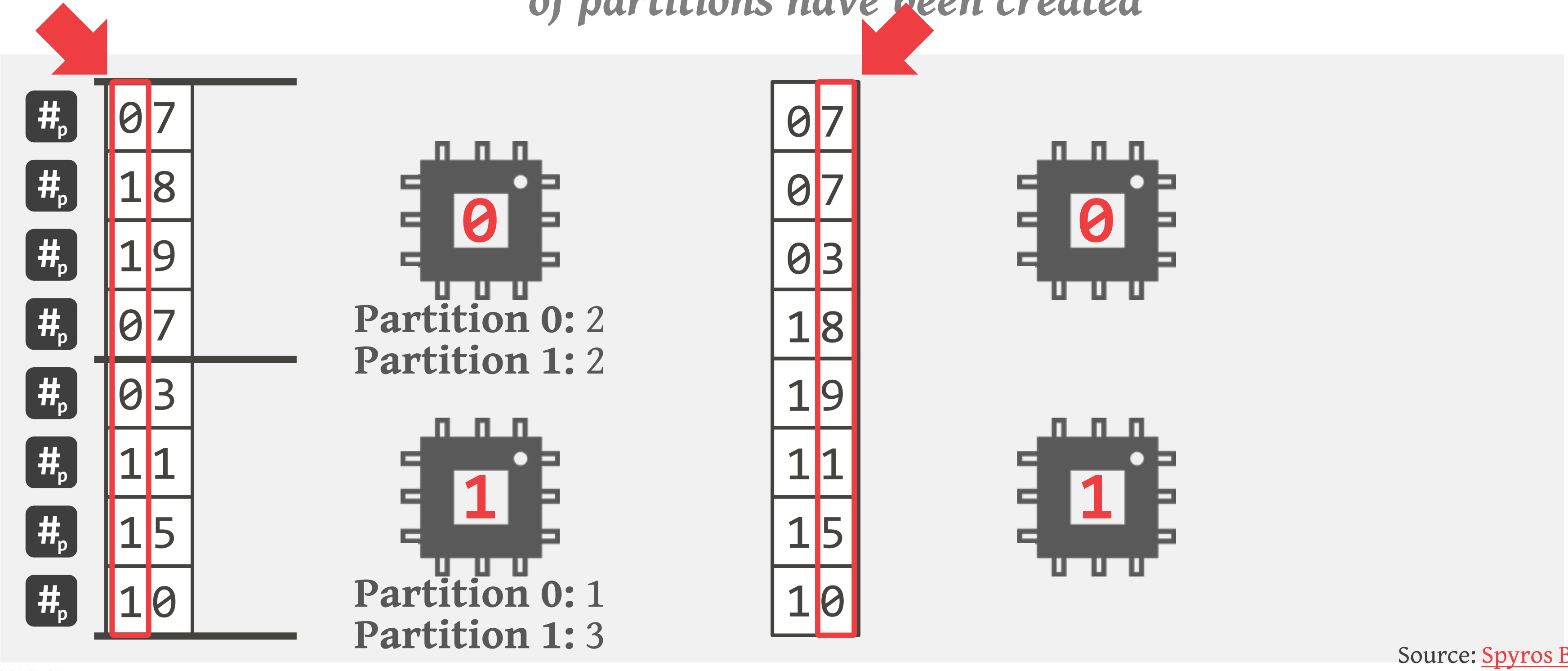
Recursively repeat until target number of partitions have been created



Source: Spyros Blanas

RADIX PARTITIONS

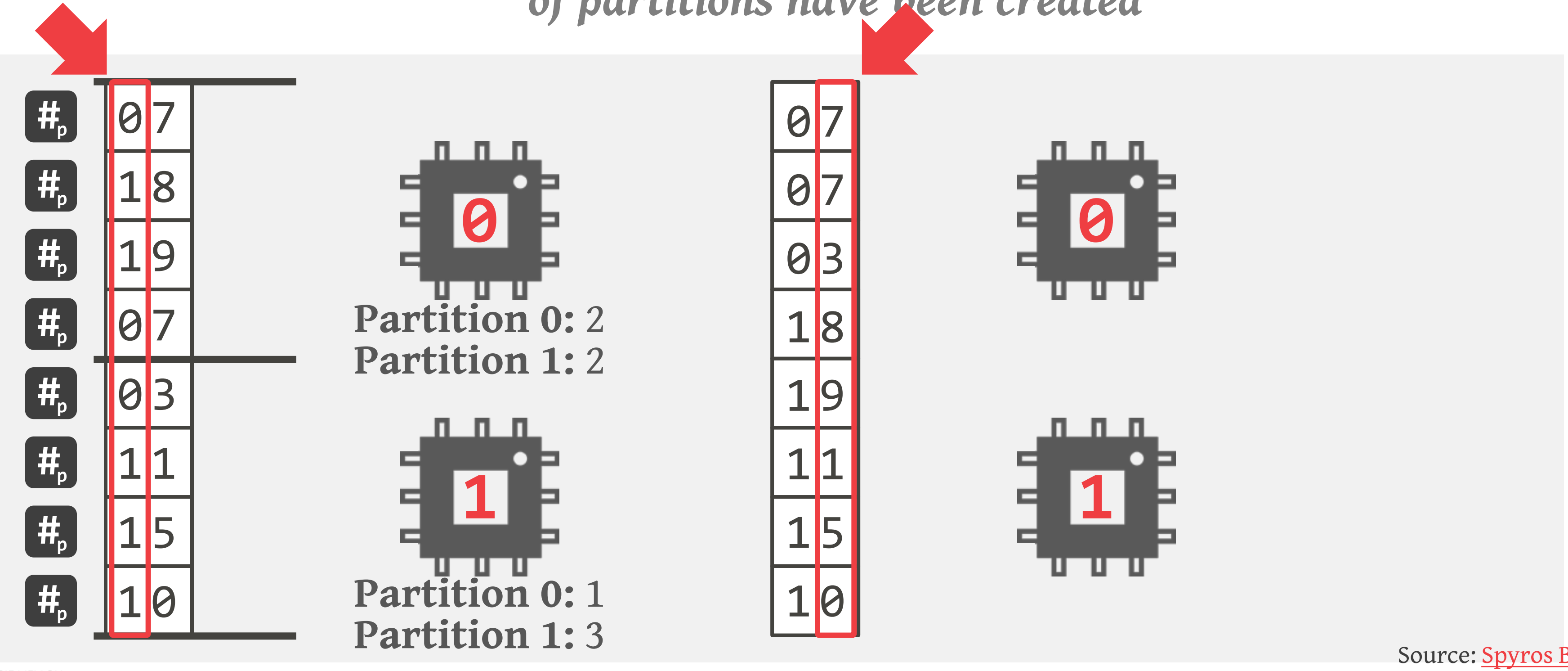
Recursively repeat until target number of partitions have been created



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.
→ The hash function must be different than the one that was used in the partition phase.

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

→ Designed to a fast, general purpose hash function.

Google CityHash

→ Based on ideas from MurmurHash2

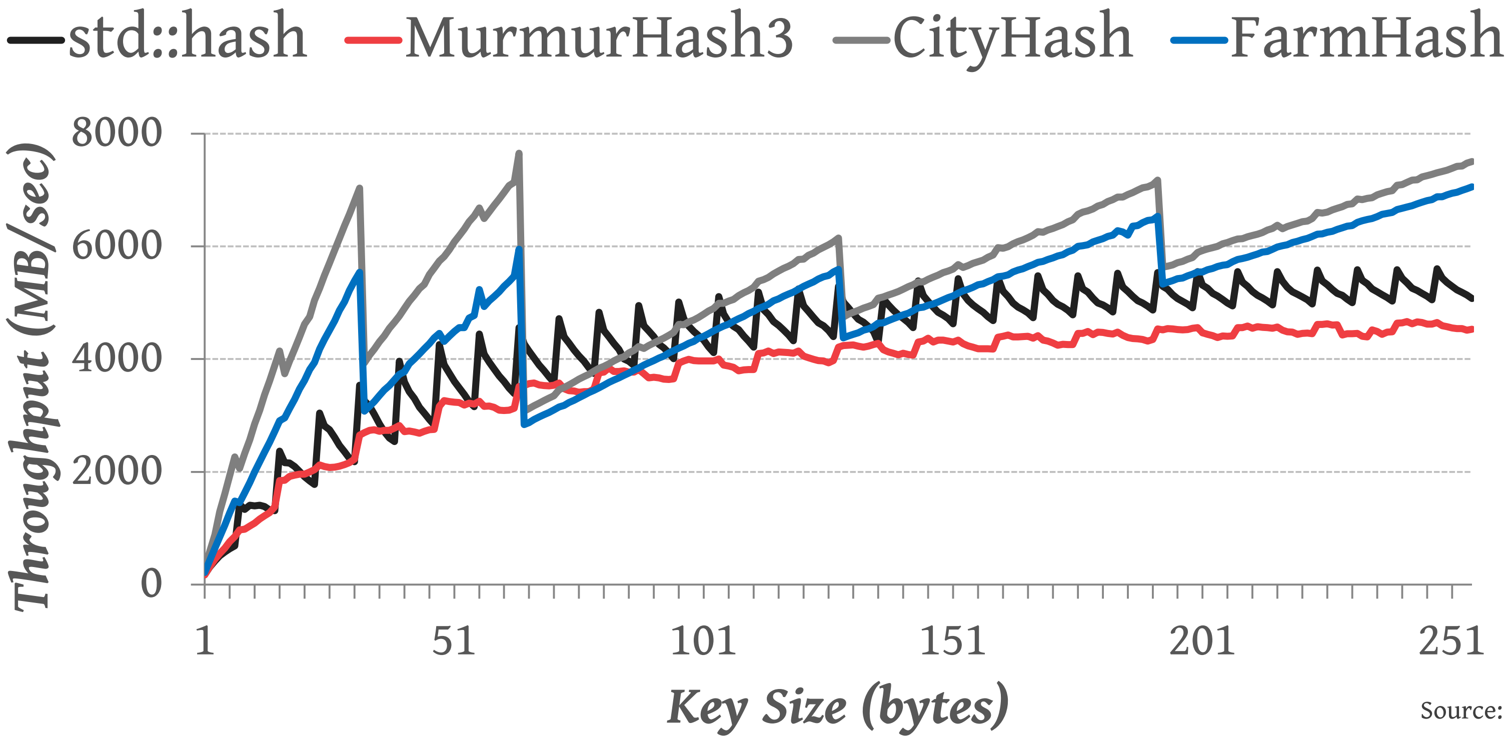
→ Designed to be faster for short keys (>64 bytes).

Google FarmHash

→ Newer version of CityHash with better collision rates.

HASH FUNCTION BENCHMARKS

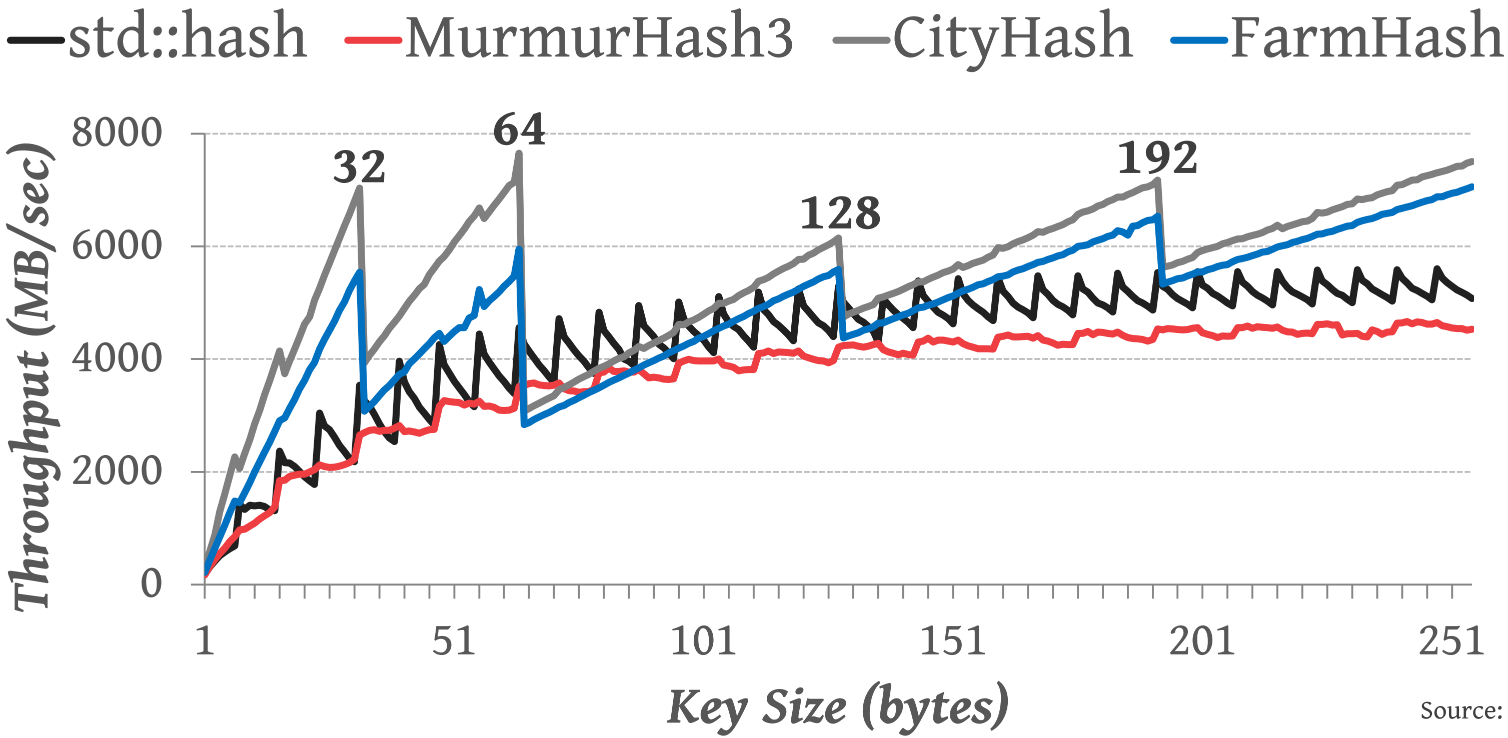
Intel Xeon CPU E5-2420 @ 2.20GHz



Source: Fredrik Widlund

HASH FUNCTION BENCHMARKS

Intel Xeon CPU E5-2420 @ 2.20GHz



Source: Fredrik Widlund

HASH TABLE IMPLEMENTATIONS

Approach #1: Chained Hash Table

Approach #2: Cuckoo Hash Table

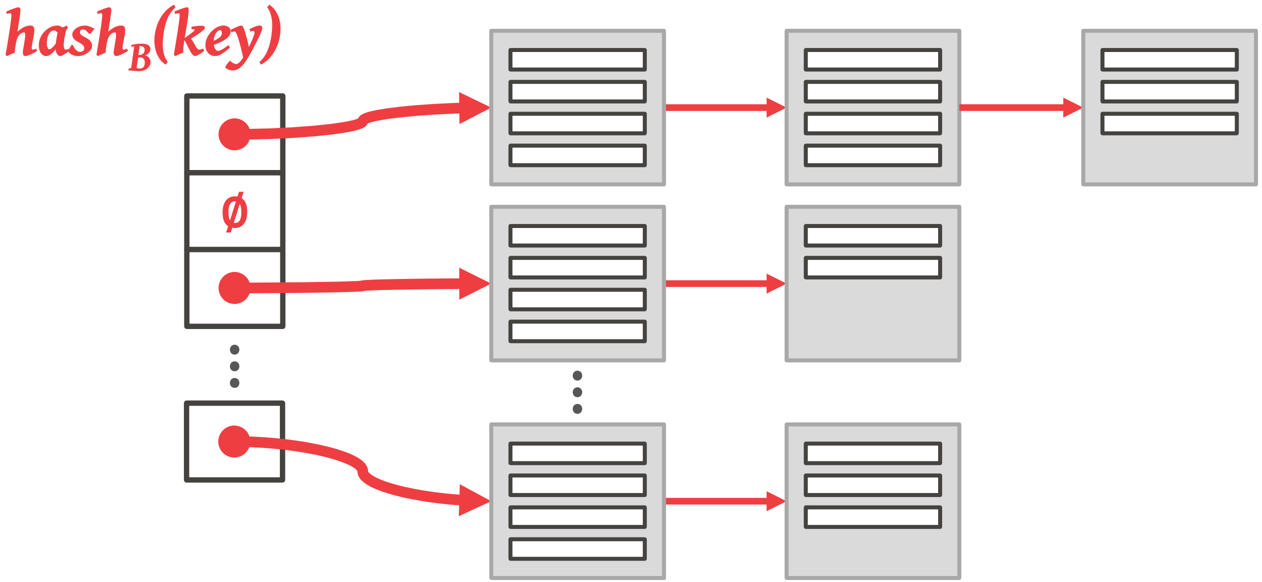
CHAINED HASH TABLE

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 HASH TABLE



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**.

CUCKOO HASH TABLE

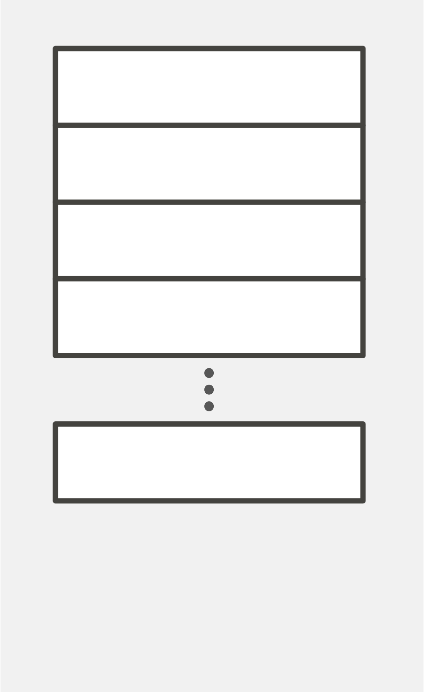
Use multiple hash tables with different hash functions.

→ 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.

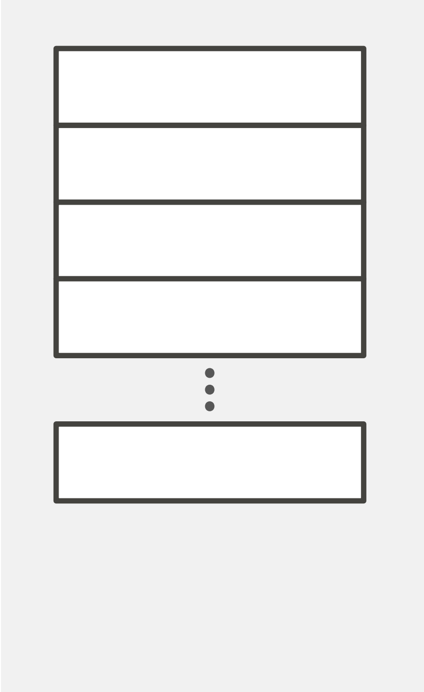
Look-ups and deletions are always O(1) because only one location per hash table is checked.

CUCKOO HASH TABLE

Hash Table #1

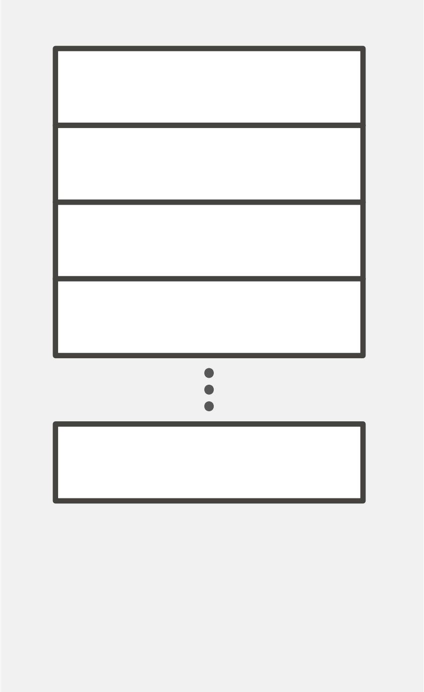


Hash Table #2



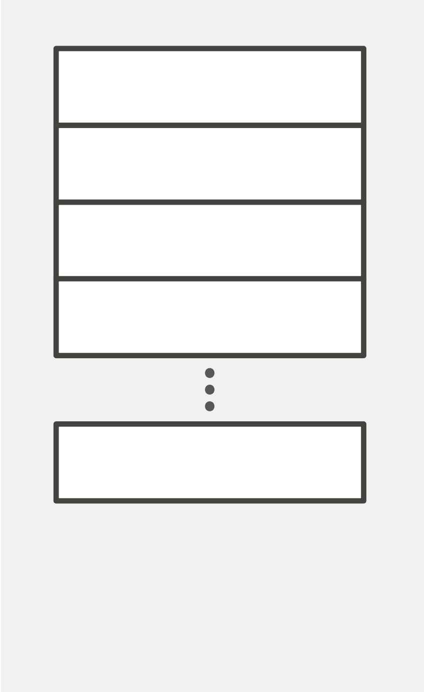
CUCKOO HASH TABLE

Hash Table #1



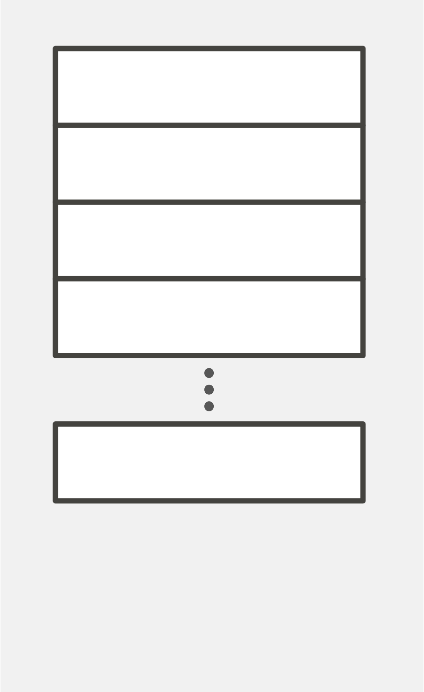
Insert X

Hash Table #2



CUCKOO HASH TABLE

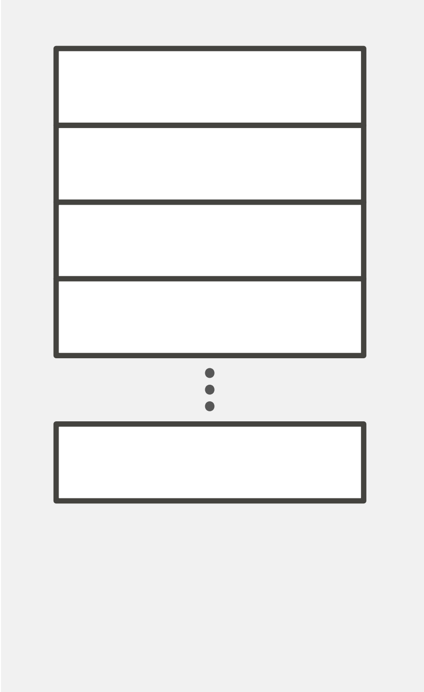
Hash Table #1



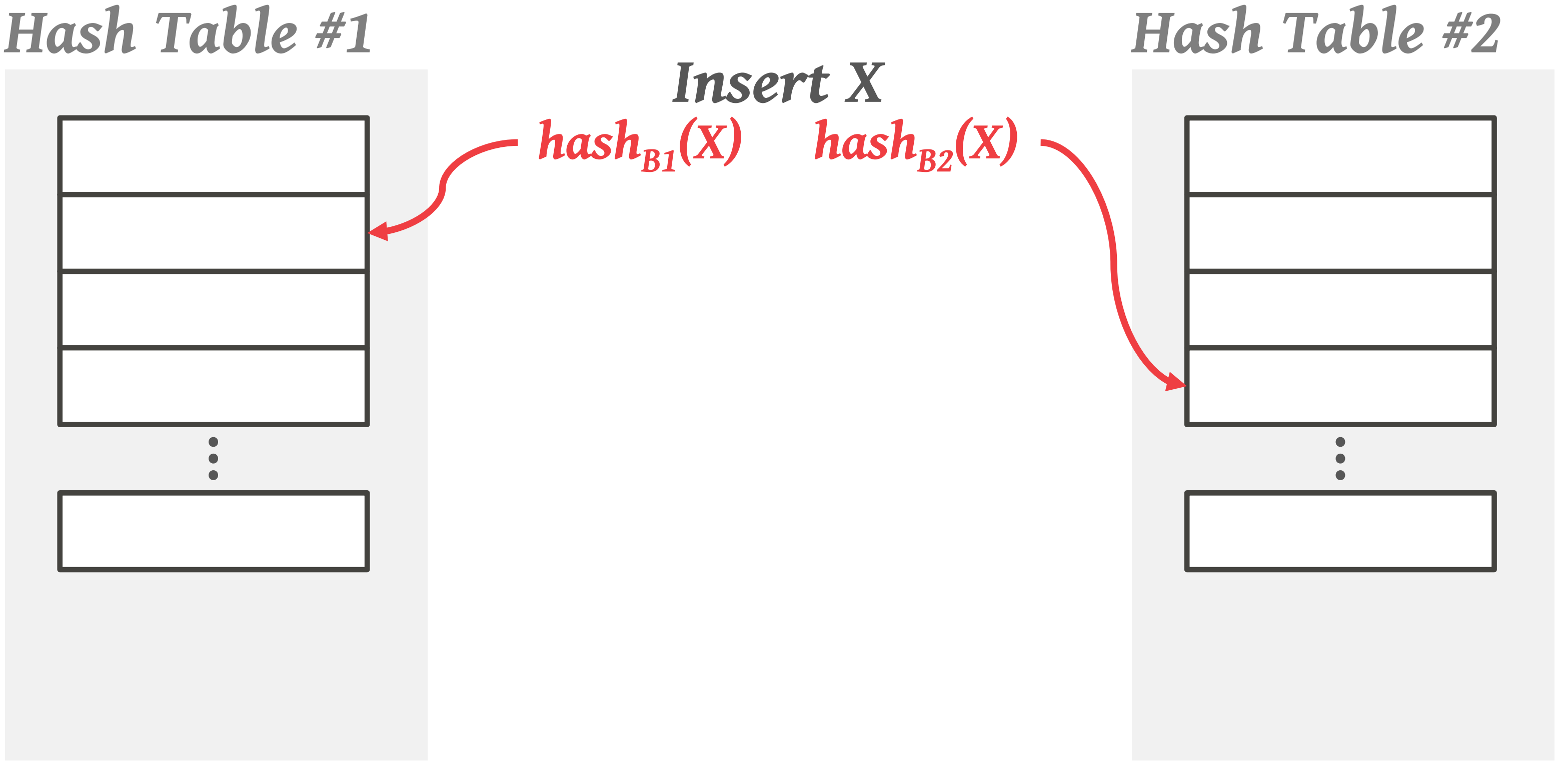
Insert X

$hash_{B1}(X)$ $hash_{B2}(X)$

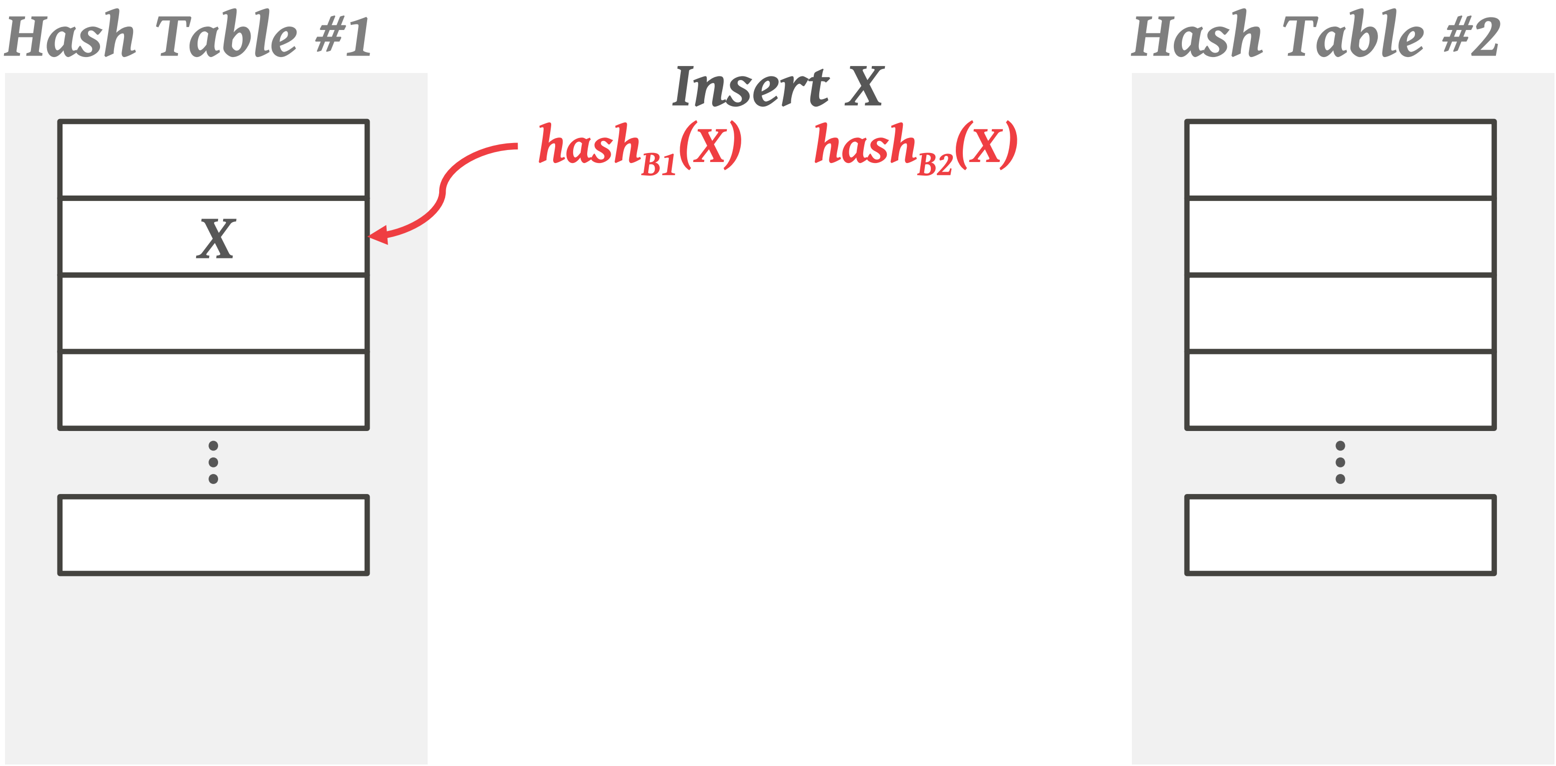
Hash Table #2



CUCKOO HASH TABLE

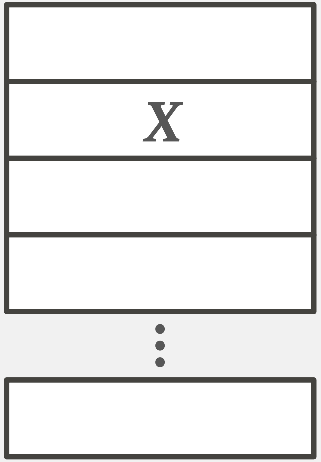


CUCKOO HASH TABLE



CUCKOO HASH TABLE

Hash Table #1



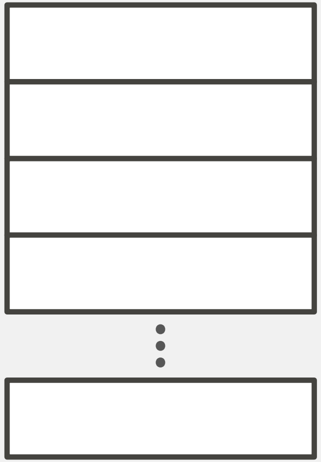
Insert X

$hash_{B1}(X)$ $hash_{B2}(X)$

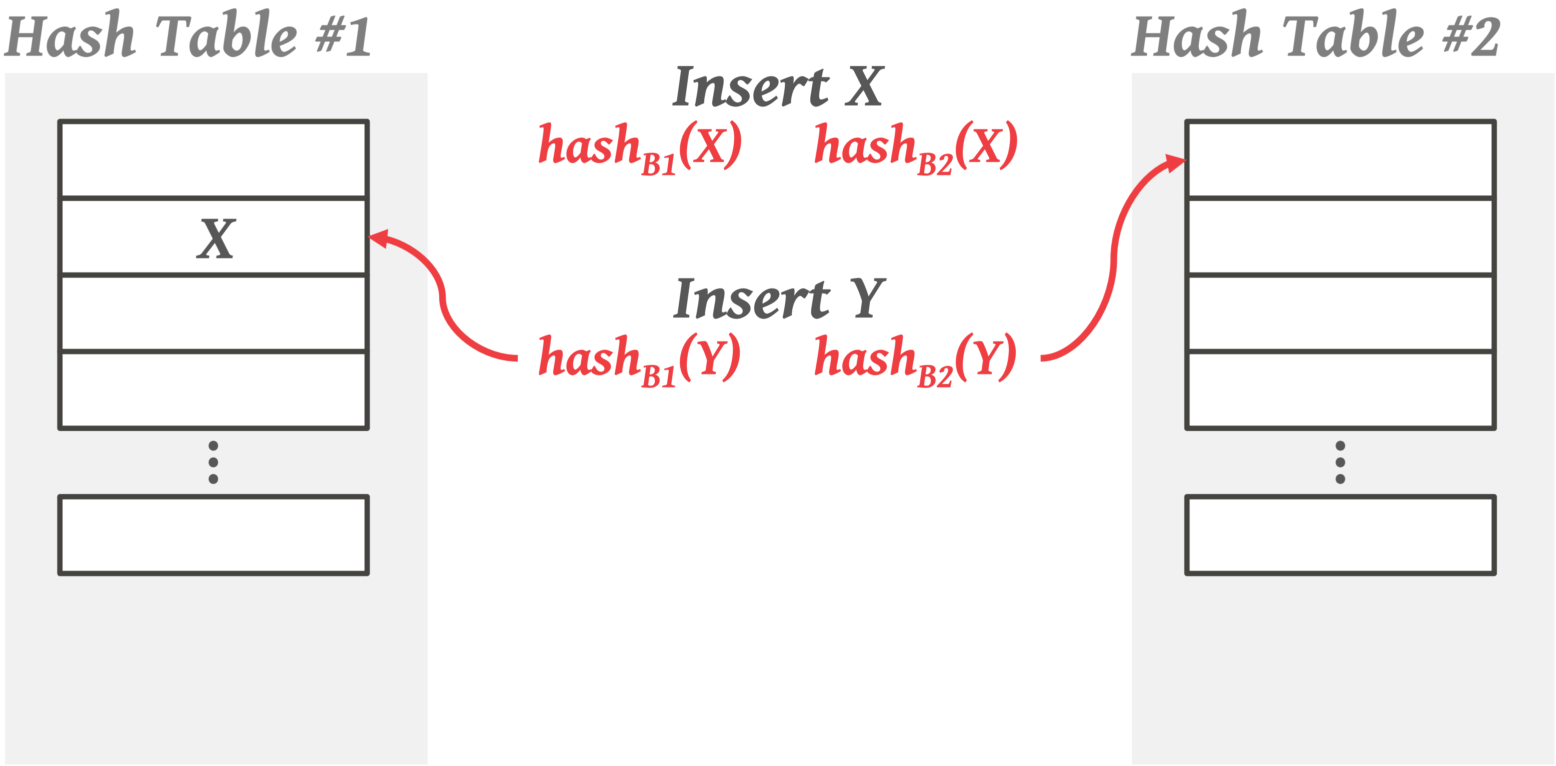
Insert Y

$hash_{B1}(Y)$ $hash_{B2}(Y)$

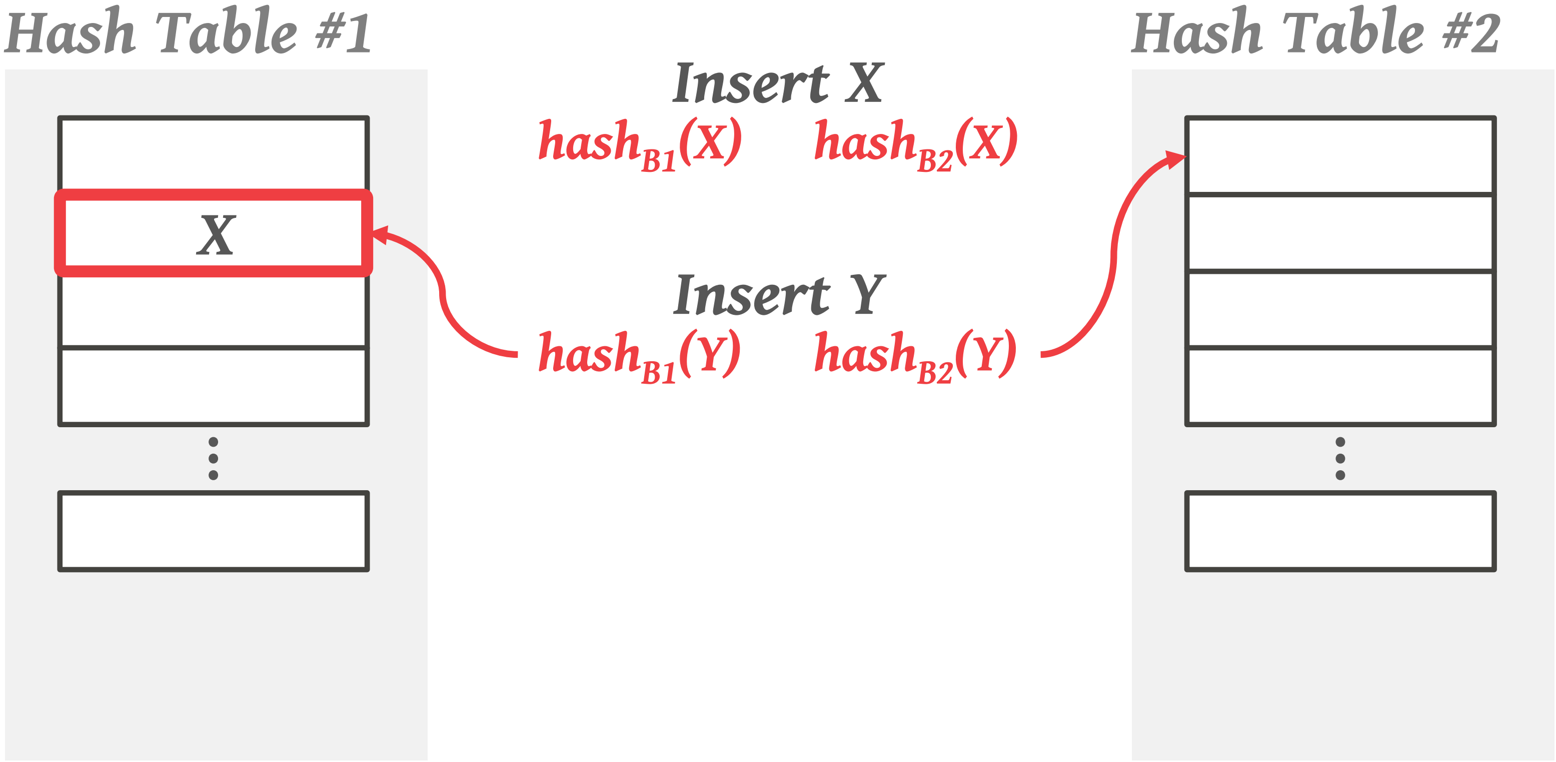
Hash Table #2



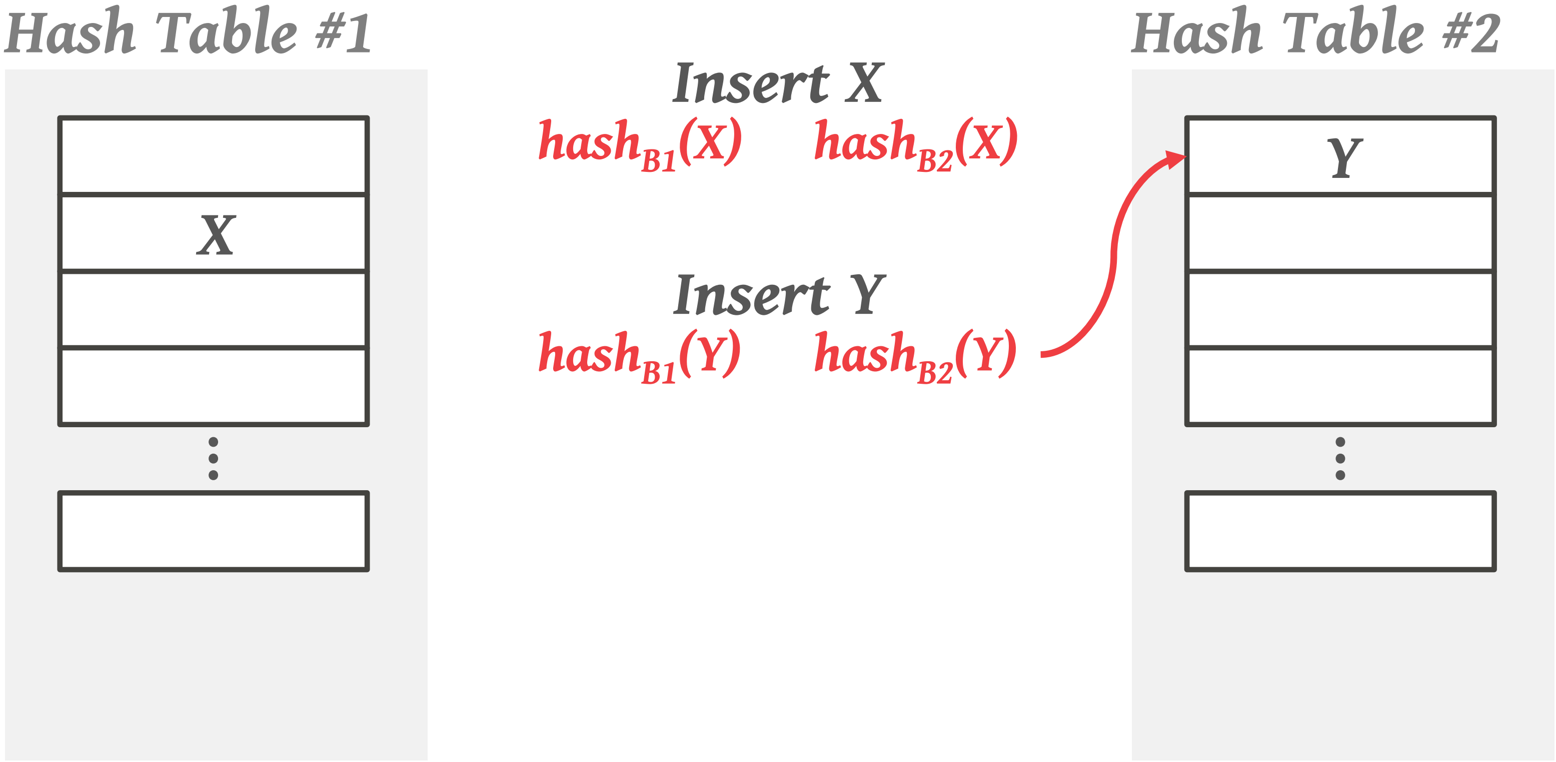
CUCKOO HASH TABLE



CUCKOO HASH TABLE

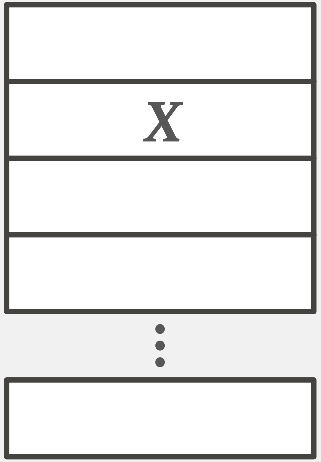


CUCKOO HASH TABLE



CUCKOO HASH TABLE

Hash Table #1



Insert X

$hash_{B1}(X)$ $hash_{B2}(X)$

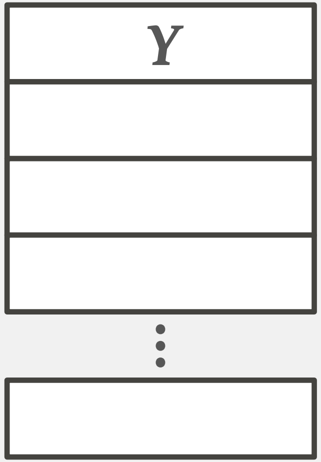
Insert Y

$hash_{B1}(Y)$ $hash_{B2}(Y)$

Insert Z

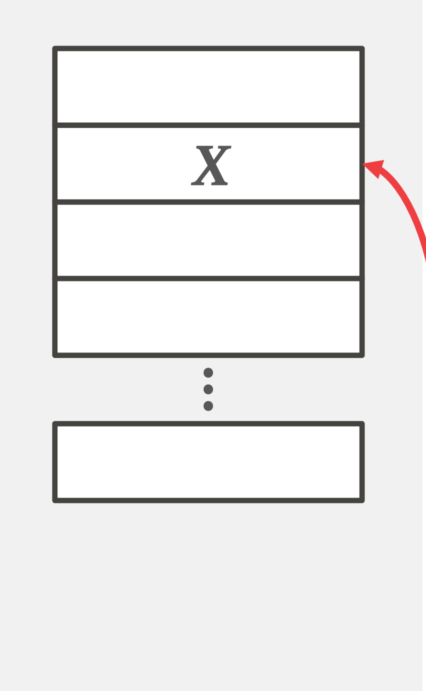
$hash_{B1}(Z)$ $hash_{B2}(Z)$

Hash Table #2



CUCKOO HASH TABLE

Hash Table #1

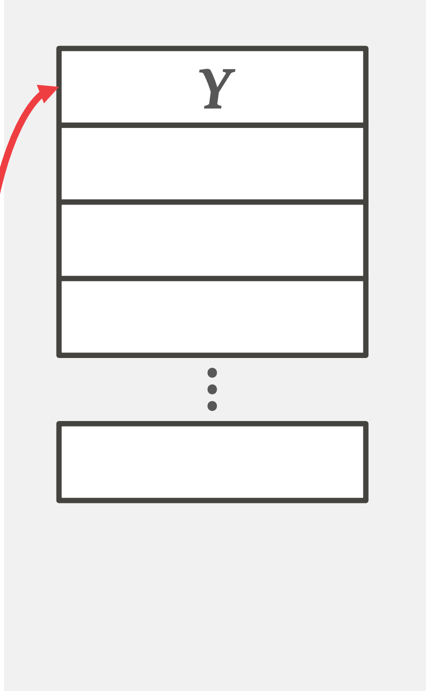


Insert X
$hash_{B1}(X)$ $hash_{B2}(X)$

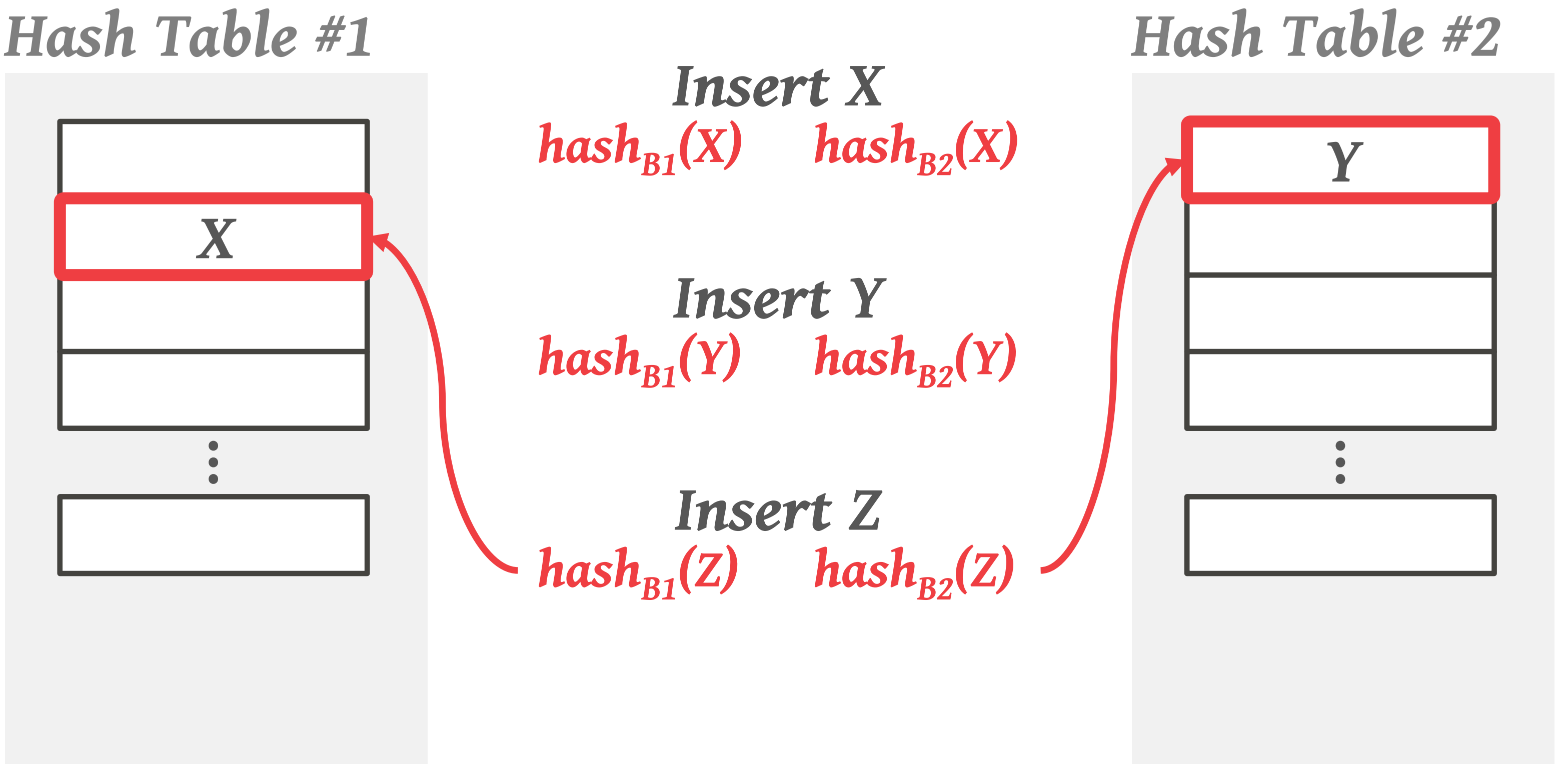
Insert Y
$hash_{B1}(Y)$ $hash_{B2}(Y)$

Insert Z
$hash_{B1}(Z)$ $hash_{B2}(Z)$

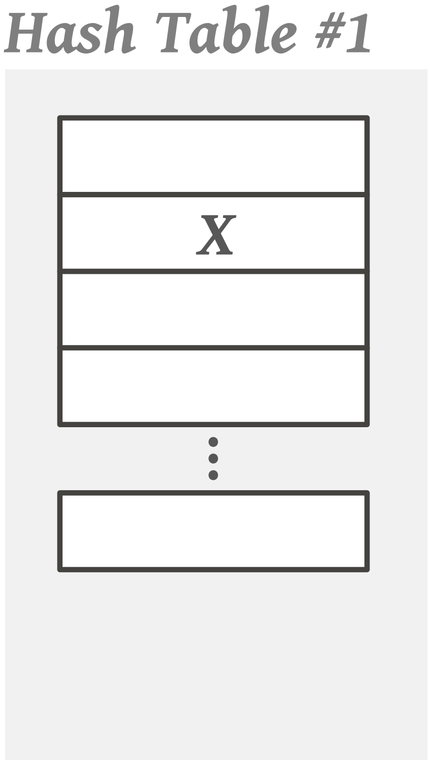
Hash Table #2



CUCKOO HASH TABLE



CUCKOO HASH TABLE



Insert X

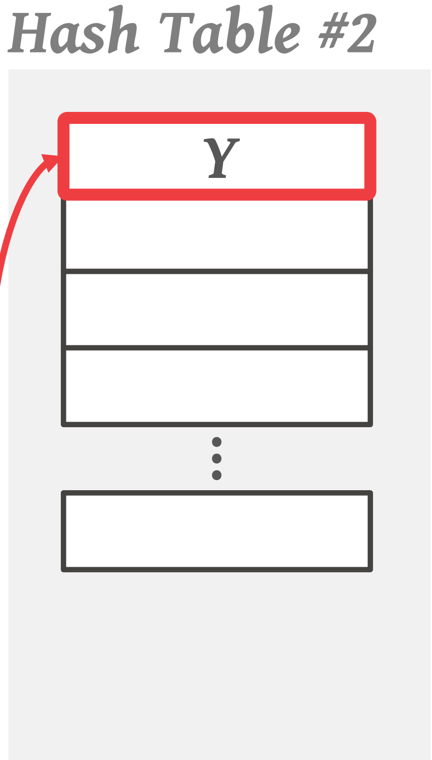
$hash_{B1}(X)$ $hash_{B2}(X)$

Insert Y

$hash_{B1}(Y)$ $hash_{B2}(Y)$

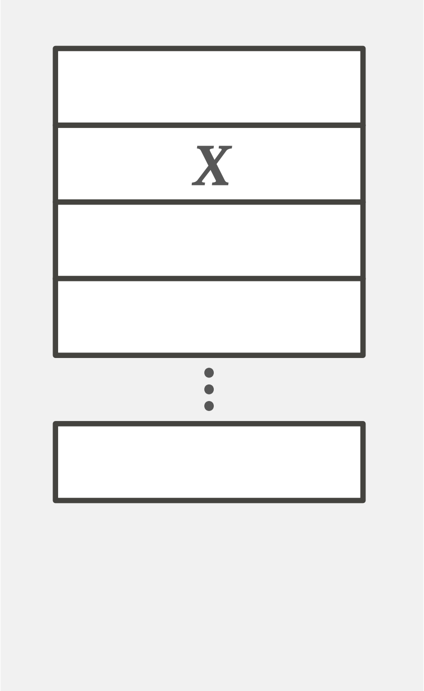
Insert Z

$hash_{B1}(Z)$ $hash_{B2}(Z)$



CUCKOO HASH TABLE

Hash Table #1



Insert X

$hash_{B1}(X)$ $hash_{B2}(X)$

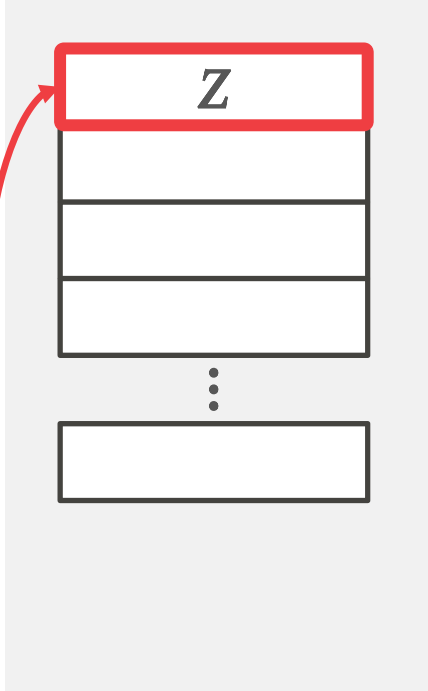
Insert Y

$hash_{B1}(Y)$ $hash_{B2}(Y)$

Insert Z

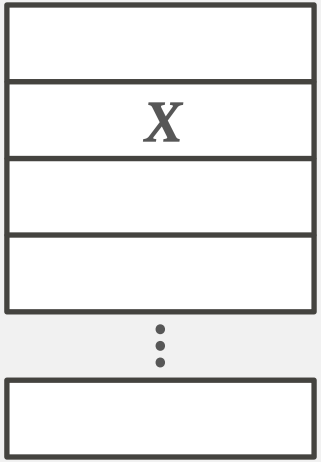
$hash_{B1}(Z)$ $hash_{B2}(Z)$

Hash Table #2



CUCKOO HASH TABLE

Hash Table #1



Insert X

$hash_{B1}(X)$ $hash_{B2}(X)$

Insert Y

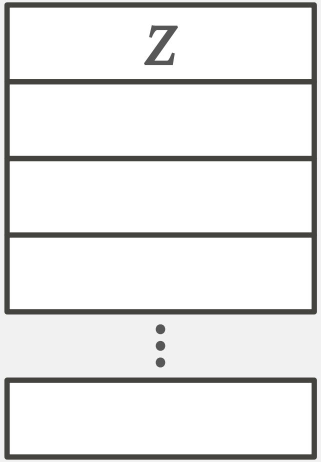
$hash_{B1}(Y)$ $hash_{B2}(Y)$

Insert Z

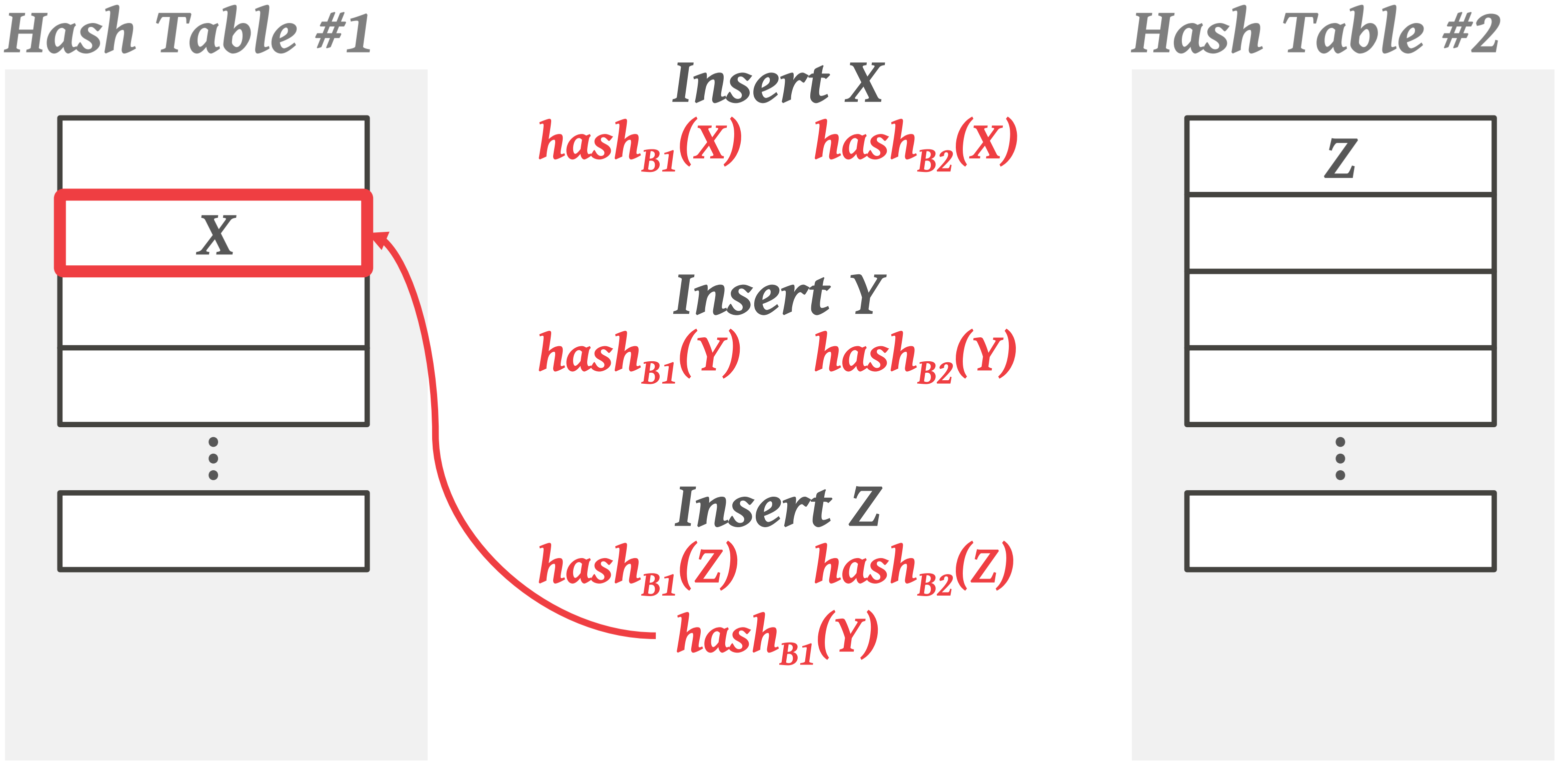
$hash_{B1}(Z)$ $hash_{B2}(Z)$

$hash_{B1}(Y)$

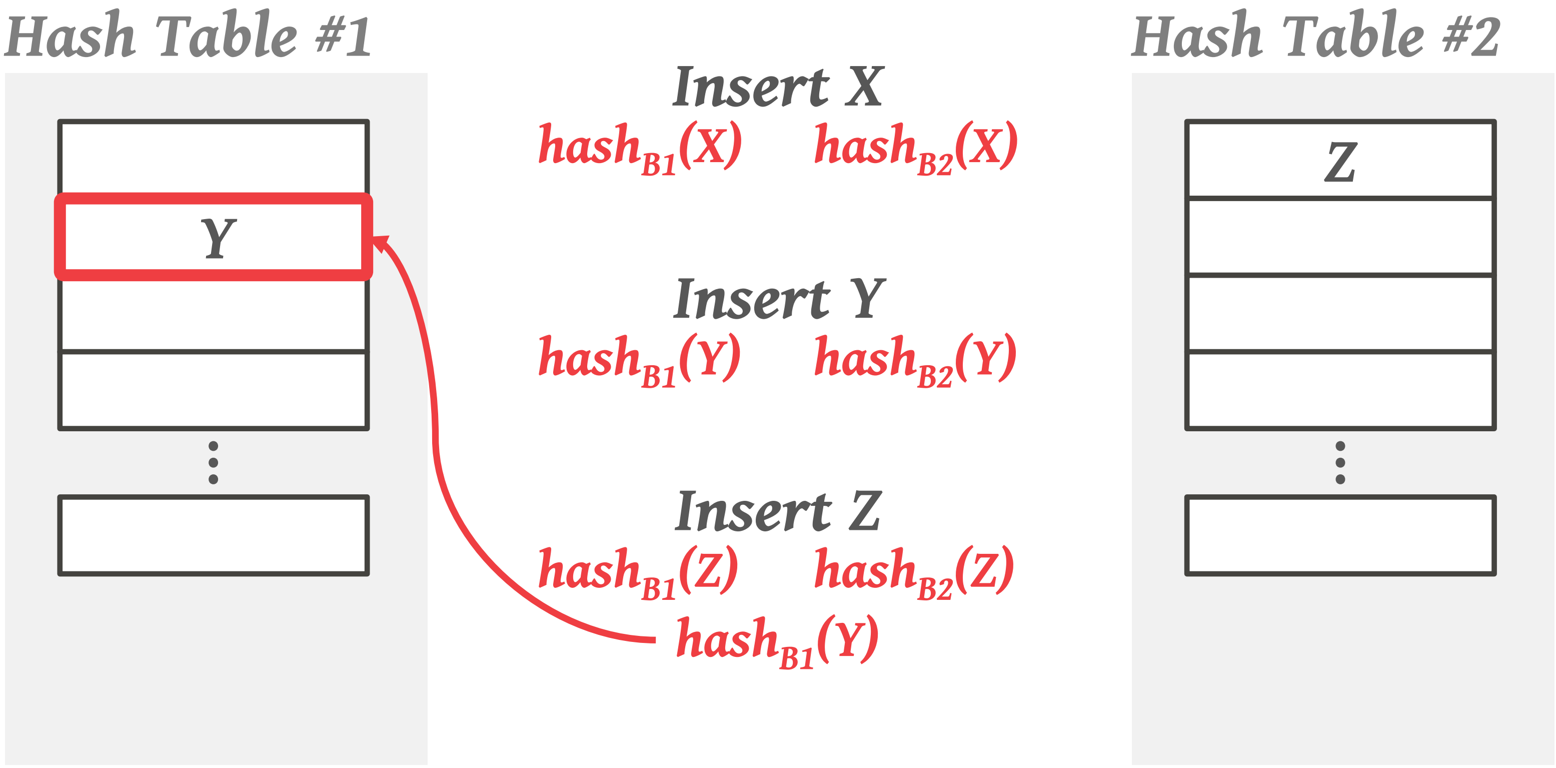
Hash Table #2



CUCKOO HASH TABLE

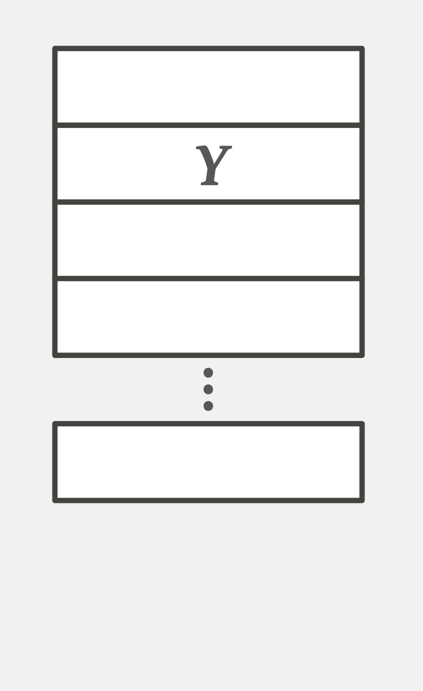


CUCKOO HASH TABLE



CUCKOO HASH TABLE

Hash Table #1



Insert X

$hash_{B1}(X)$ $hash_{B2}(X)$

Insert Y

$hash_{B1}(Y)$ $hash_{B2}(Y)$

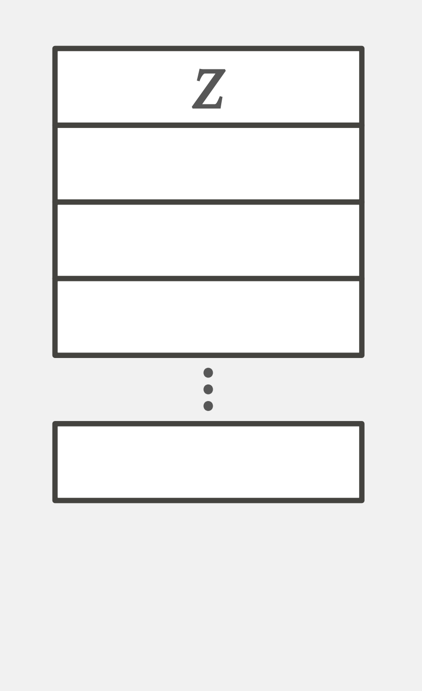
Insert Z

$hash_{B1}(Z)$ $hash_{B2}(Z)$

$hash_{B1}(Y)$

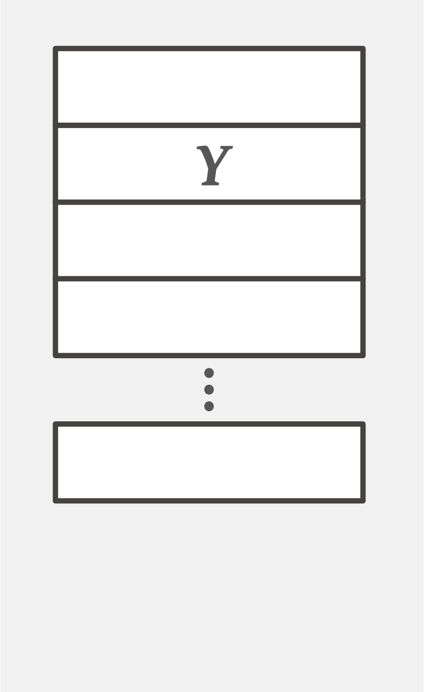
$hash_{B2}(X)$

Hash Table #2



CUCKOO HASH TABLE

Hash Table #1



Insert X

$hash_{B1}(X)$ $hash_{B2}(X)$

Insert Y

$hash_{B1}(Y)$ $hash_{B2}(Y)$

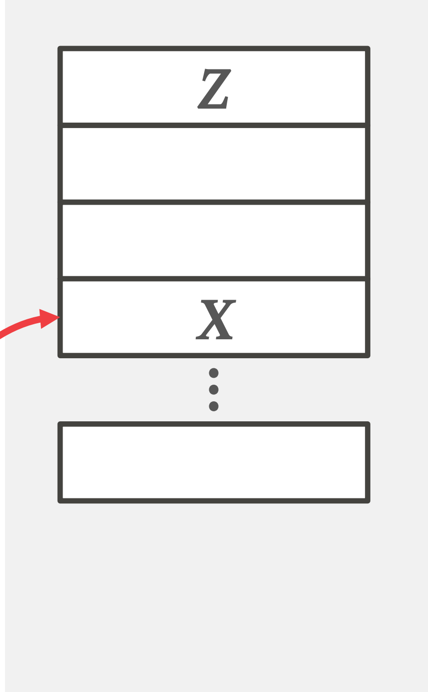
Insert Z

$hash_{B1}(Z)$ $hash_{B2}(Z)$

$hash_{B1}(Y)$

$hash_{B2}(X)$

Hash Table #2



CUCKOO HASH TABLE

We have to make sure that we 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**

HASH JOIN VARIANTS

No Partitioning + Shared Hash Table

Non-Blocking Partitioning + Shared Buffers

Non-Blocking Partitioning + Private Buffers

Blocking (Radix) Partitioning

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

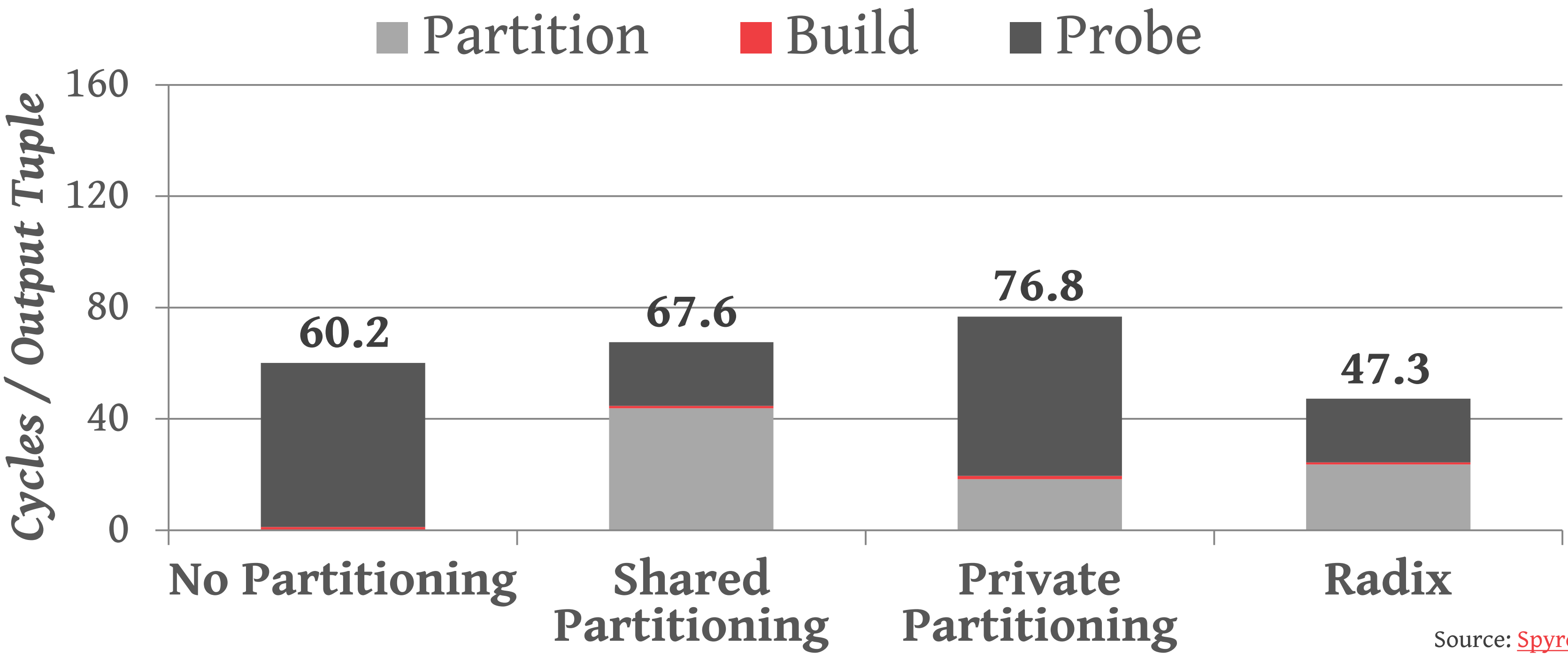
Uniform and highly skewed (Zipf; s=1.25)

No output materialization

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

HASH JOIN – UNIFORM DATA SET

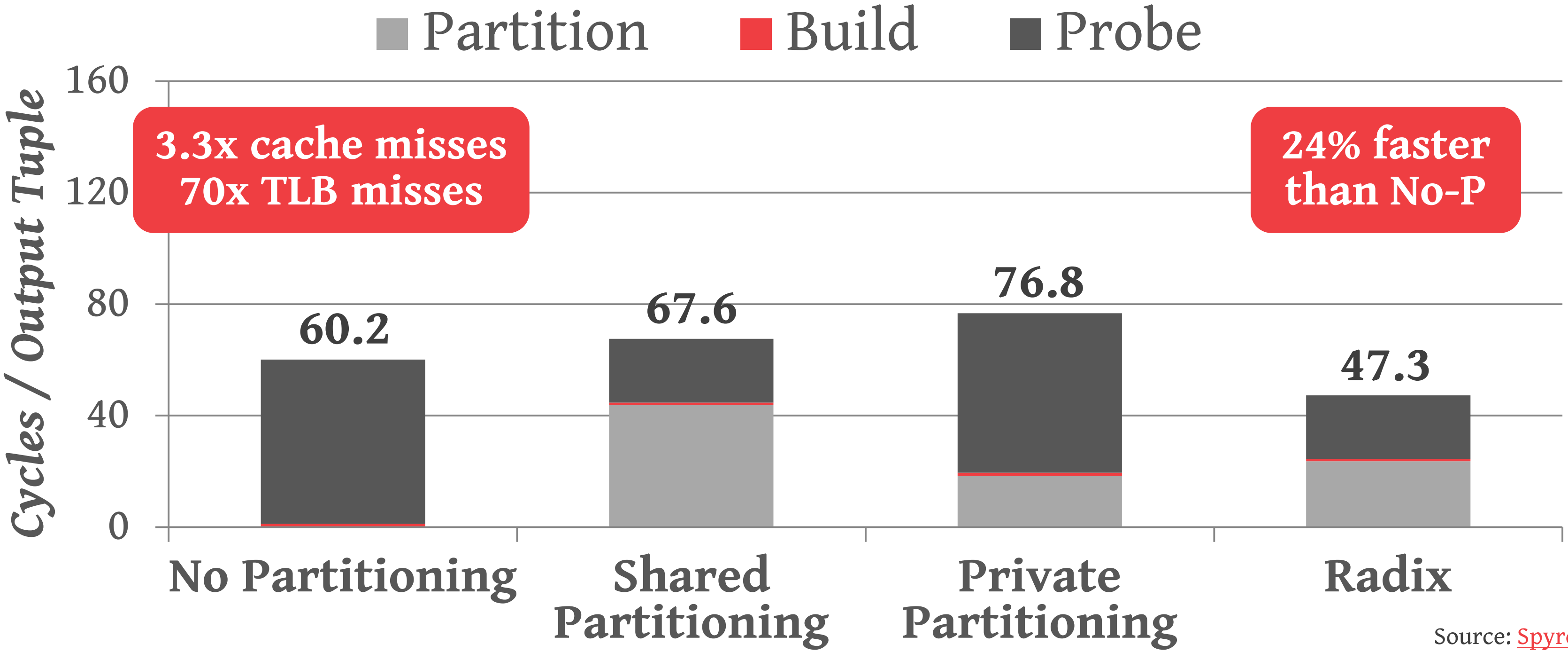
Intel Xeon CPU X5650 @ 2.66GHz
6 Cores with 2 Threads Per Core



Source: Spyros Blanas

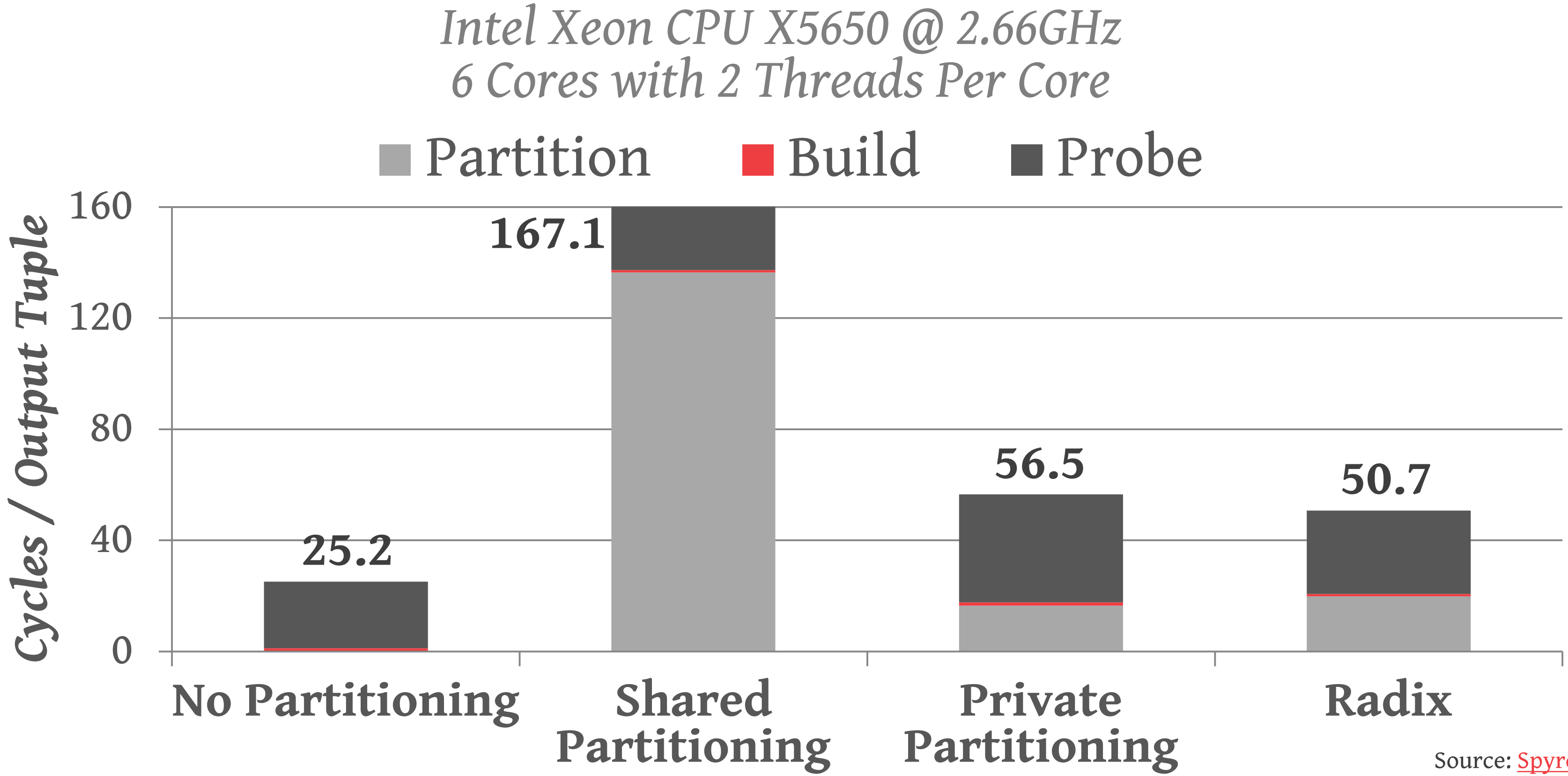
HASH JOIN – UNIFORM DATA SET

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



Source: Spyros Blanas

HASH JOIN – SKEWED DATA SET



Source: Spyros Blanas

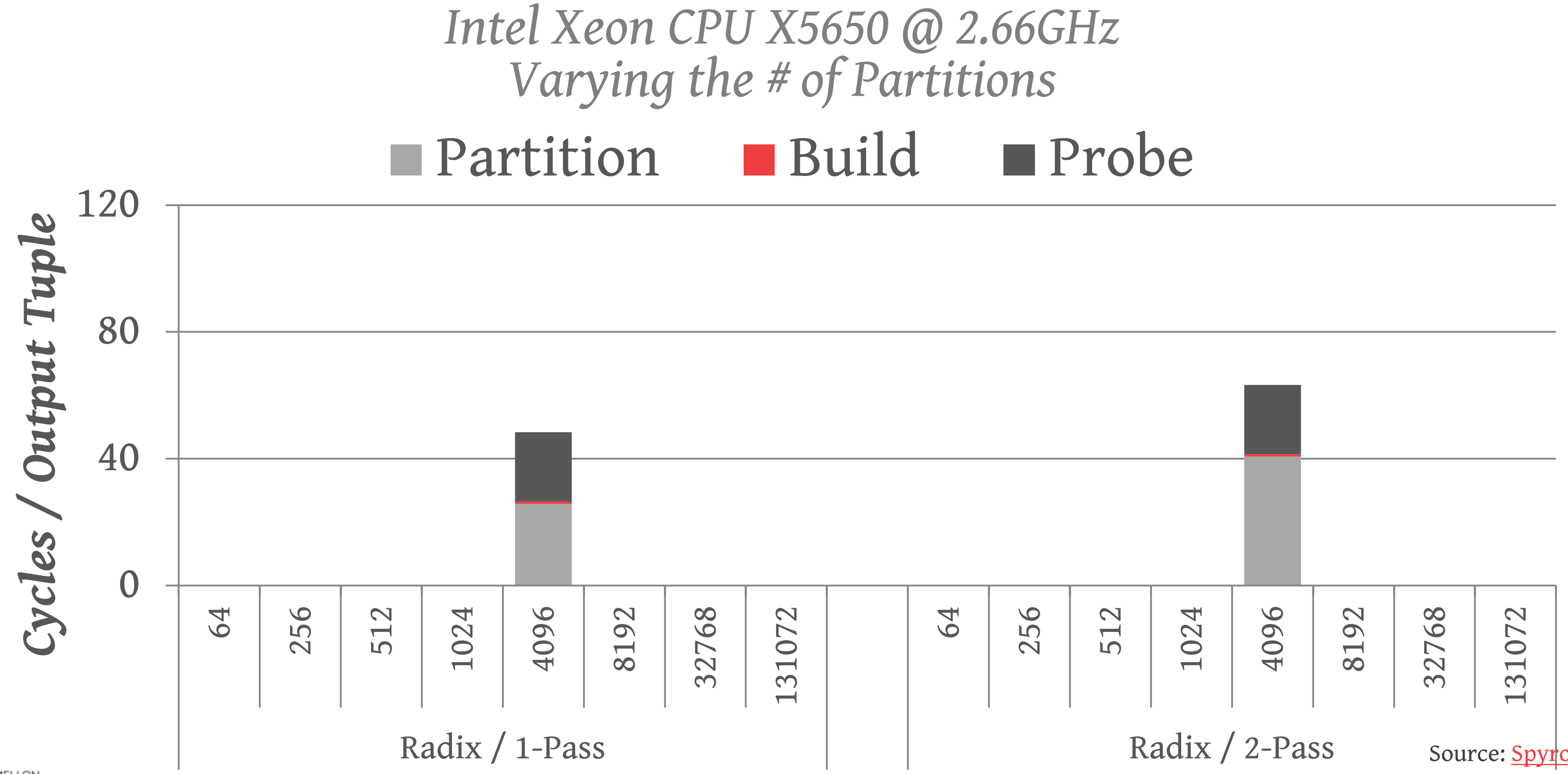
OBSERVATION

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).

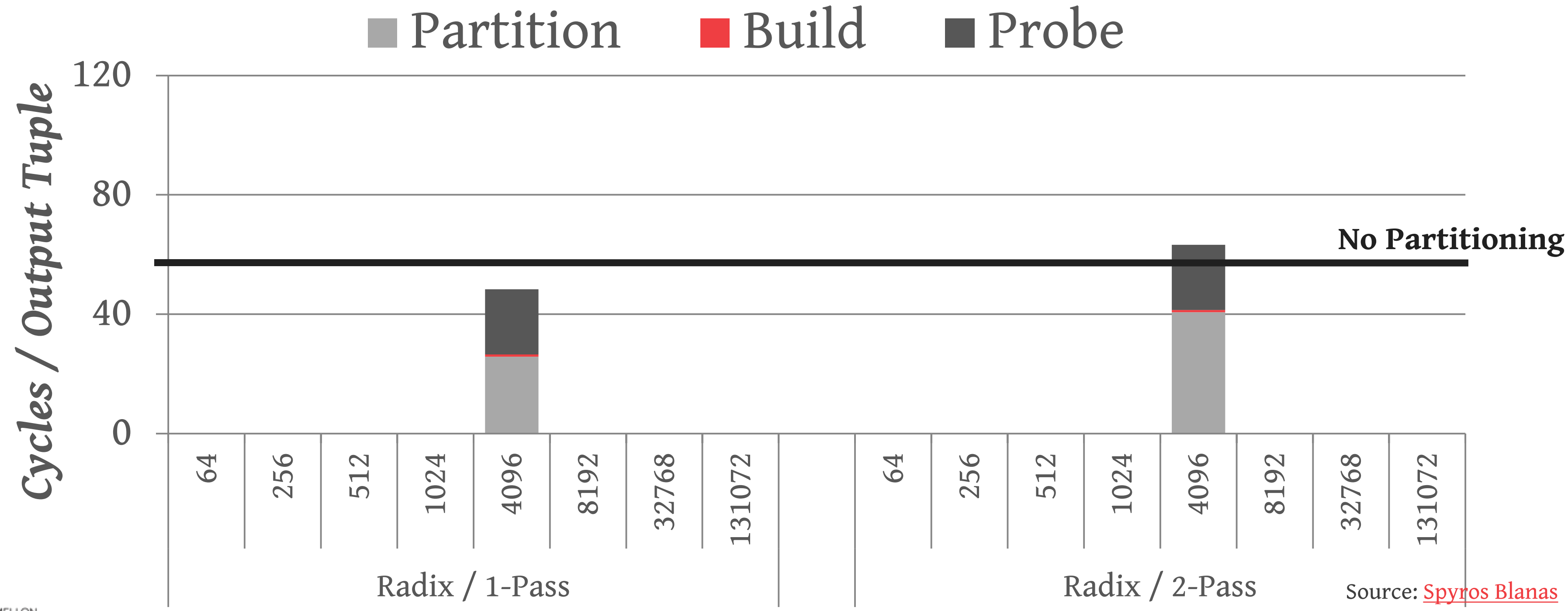
RADIX HASH JOIN – UNIFORM DATA SET



Source: Spyros Blanas

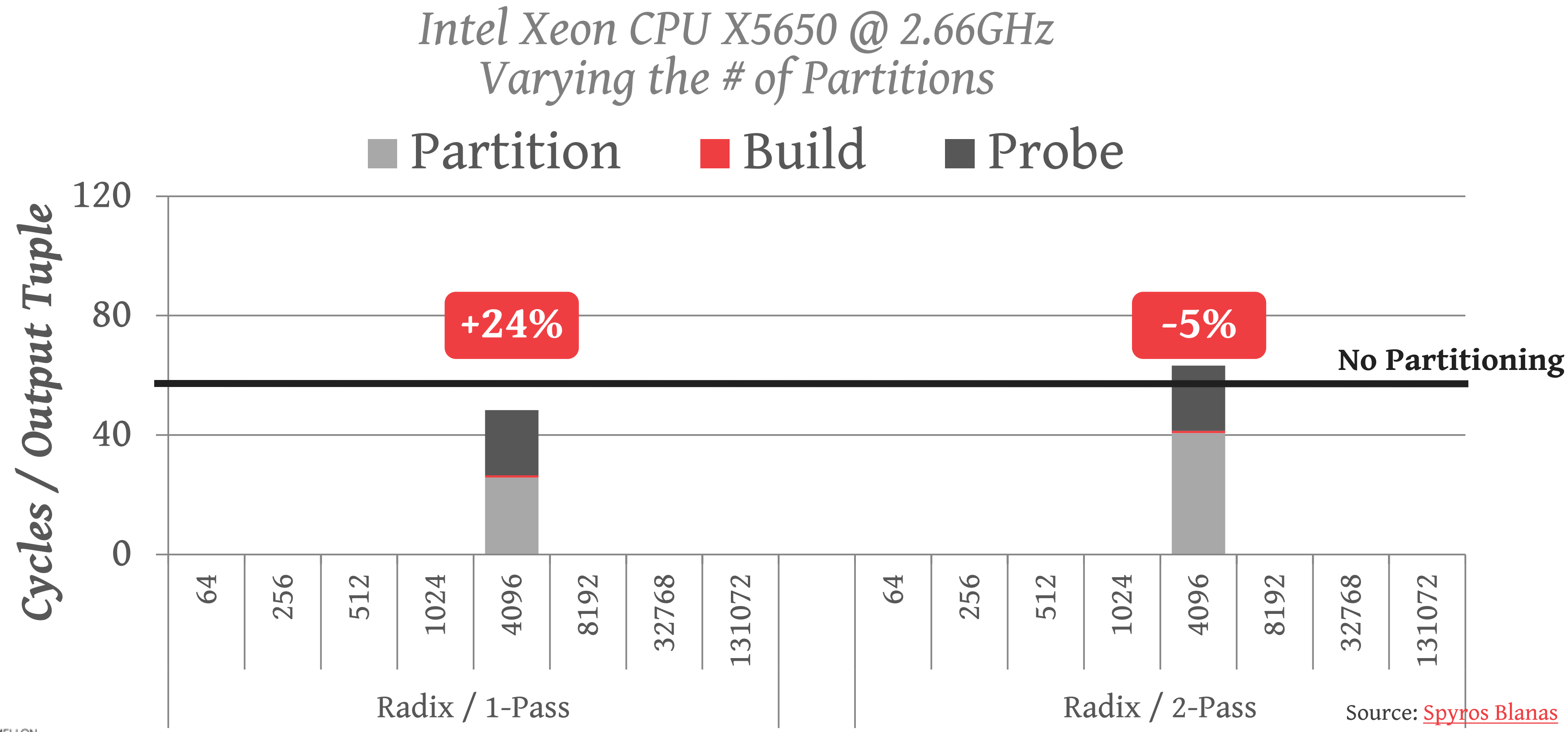
RADIX HASH JOIN – UNIFORM DATA SET

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



Source: Spyros Blanas

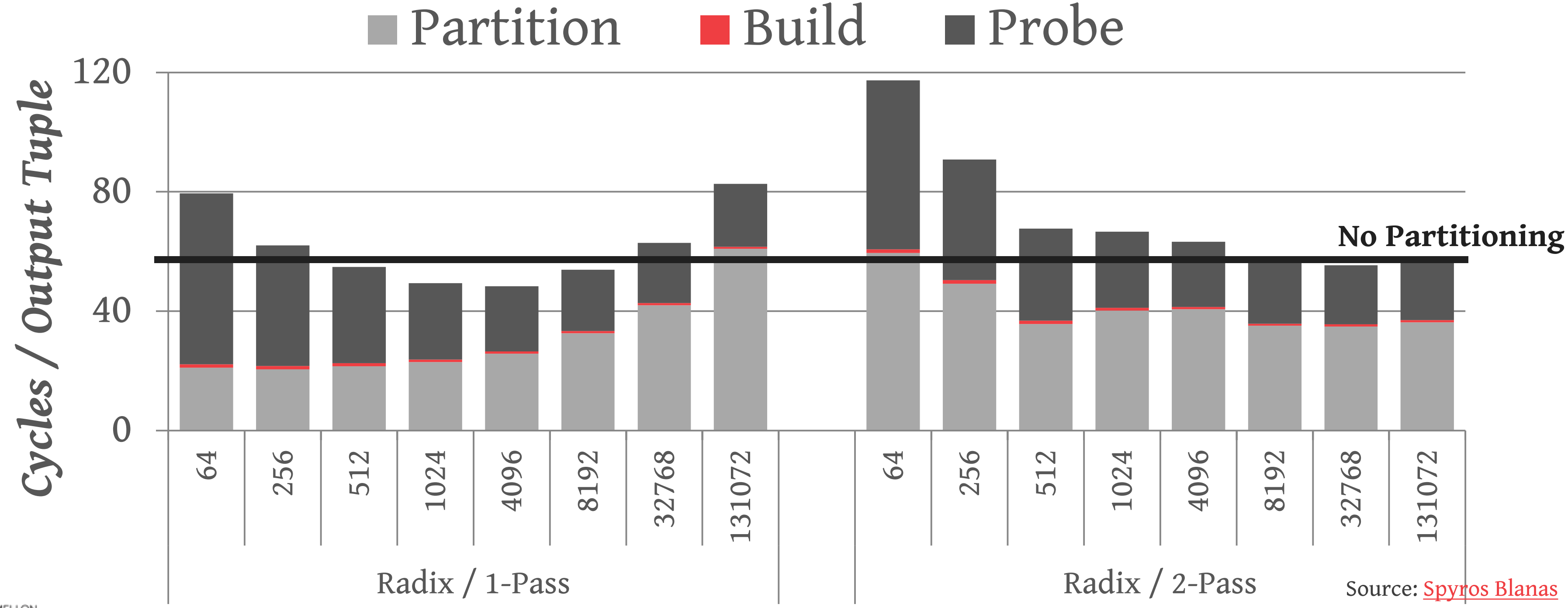
RADIX HASH JOIN – UNIFORM DATA SET



Source: Spyros Blanas

RADIX HASH JOIN – UNIFORM DATA SET

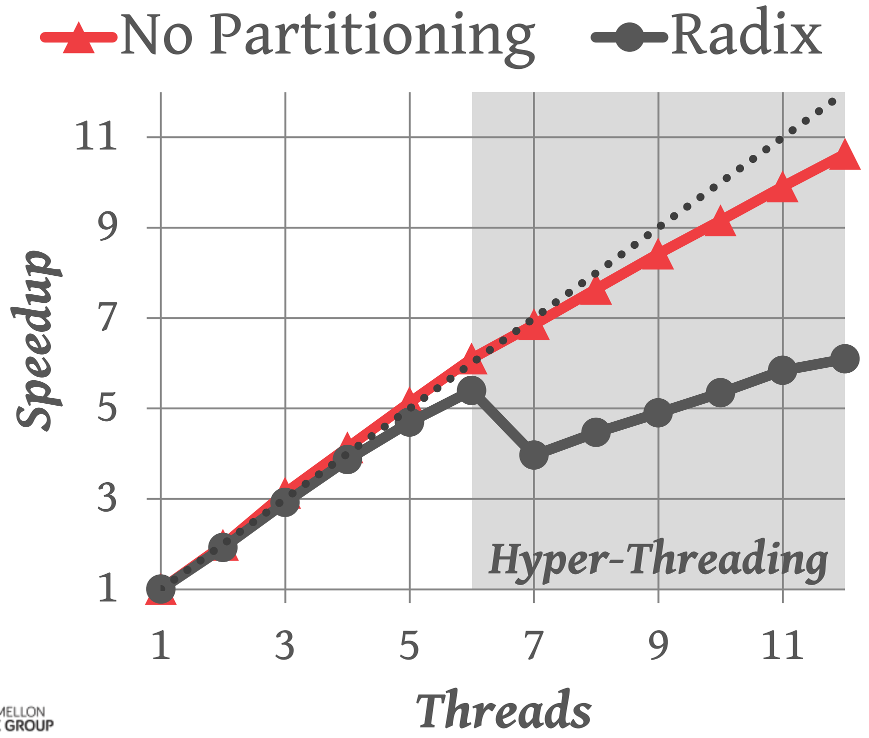
Intel Xeon CPU X5650 @ 2.66GHz
Varying the # of Partitions



Source: Spyros Blanas

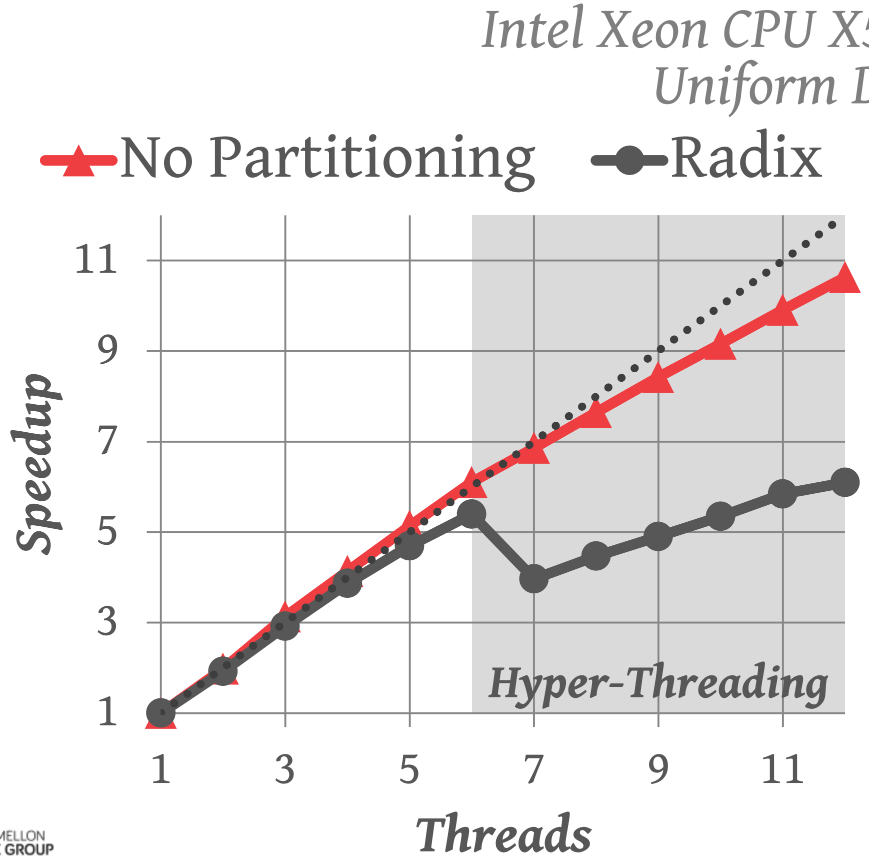
EFFECTS OF HYPER-THREADING

Intel Xeon CPU X5650 @ 2.66GHz
Uniform Data Set



Source: Spyros Blanas

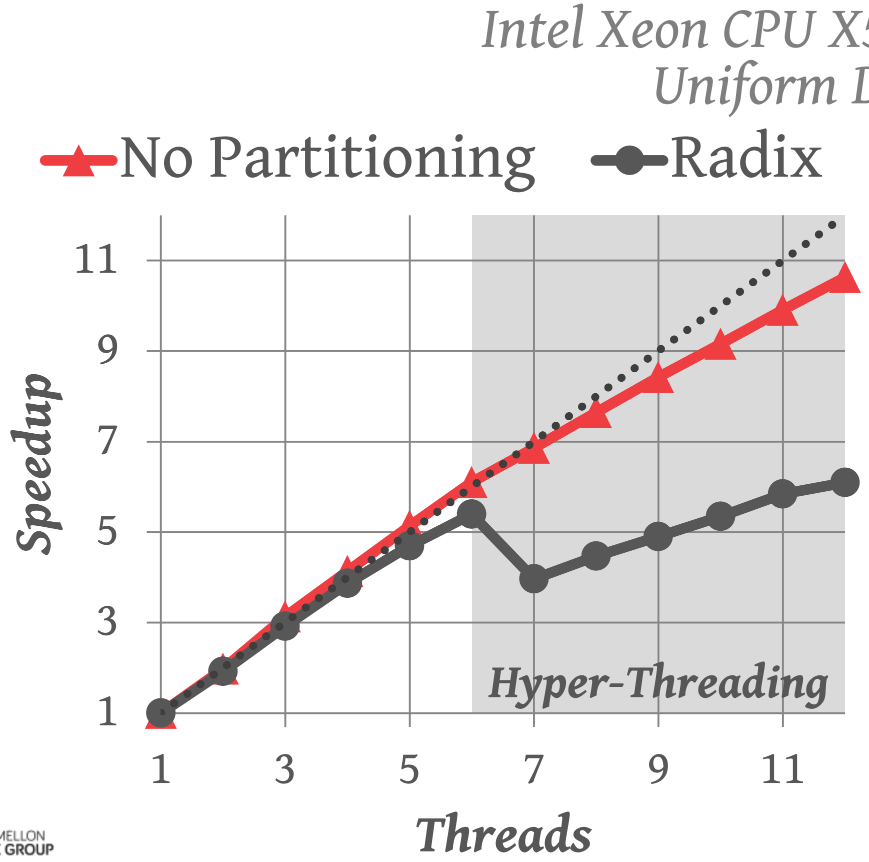
EFFECTS OF HYPER-THREADING



Multi-threading hides cache & TLB miss latency.

Source: Spyros Blanas

EFFECTS OF HYPER-THREADING



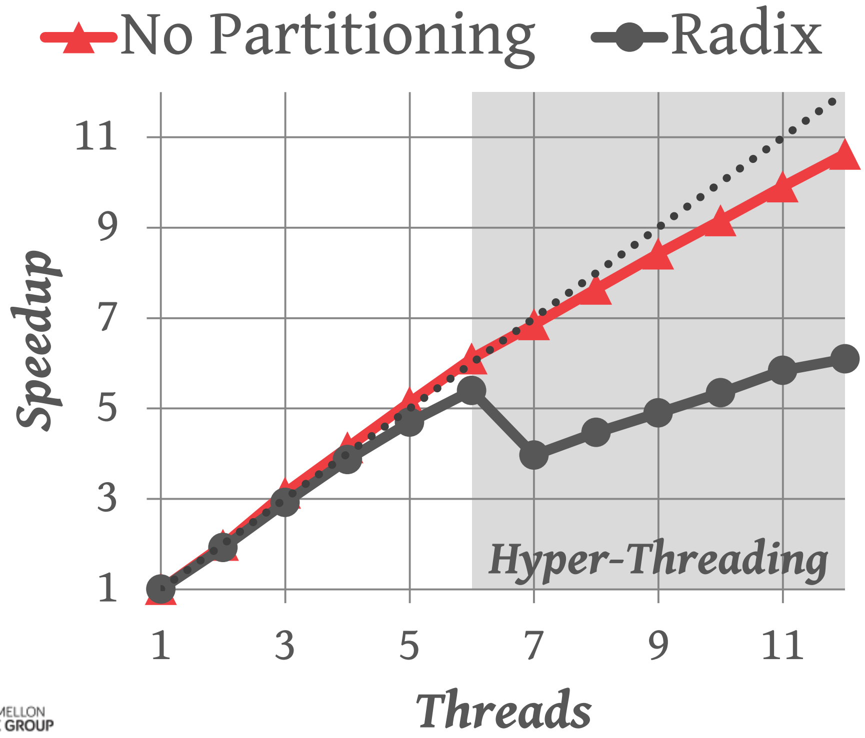
Multi-threading hides cache & TLB miss latency.

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

Source: Spyros Blanas

EFFECTS OF HYPER-THREADING

Intel Xeon CPU X5650 @ 2.66GHz
Uniform Data Set



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 vectored execution optimizations that are possible in hash joins that we didn't talk about.

NEXT CLASS

Parallel Sort-Merge Joins
Hate Mail