15-721 DATABASE SYSTEMS

Lecture #21 – Vectorized Execution

Andy Pavlo // Carnegie Mellon University // Spring 2016

PROJECT #3

MemSQL machines are ready to use.

In-class status updates next Wednesday.

TODAY'S AGENDA

Background
Vectorized Algorithms (Columbia)
BitWeaving (Wisconsin)

OBVIOUS OBSERVATIONS

OBVIOUS OBSERVATIONS

#1 – Building a DBMS is hard.

#2 – Taco Bell gives you diarrhea.

#3 – New CPUs are not getting faster.

MULTI-CORE CPUS

Use a small number of high-powered cores.
→ Intel Haswell / Skylake
→ High power consumption and area per core.

Massively **superscalar** and aggressive **out-of-order** execution
→ Instructions are issued from a sequential stream.
→ Check for dependencies between instructions.
→ Process multiple instructions per clock cycle.

MANY INTEGRATED CORES (MIC)

Use a larger number of low-powered cores.

→ Intel Xeon Phi

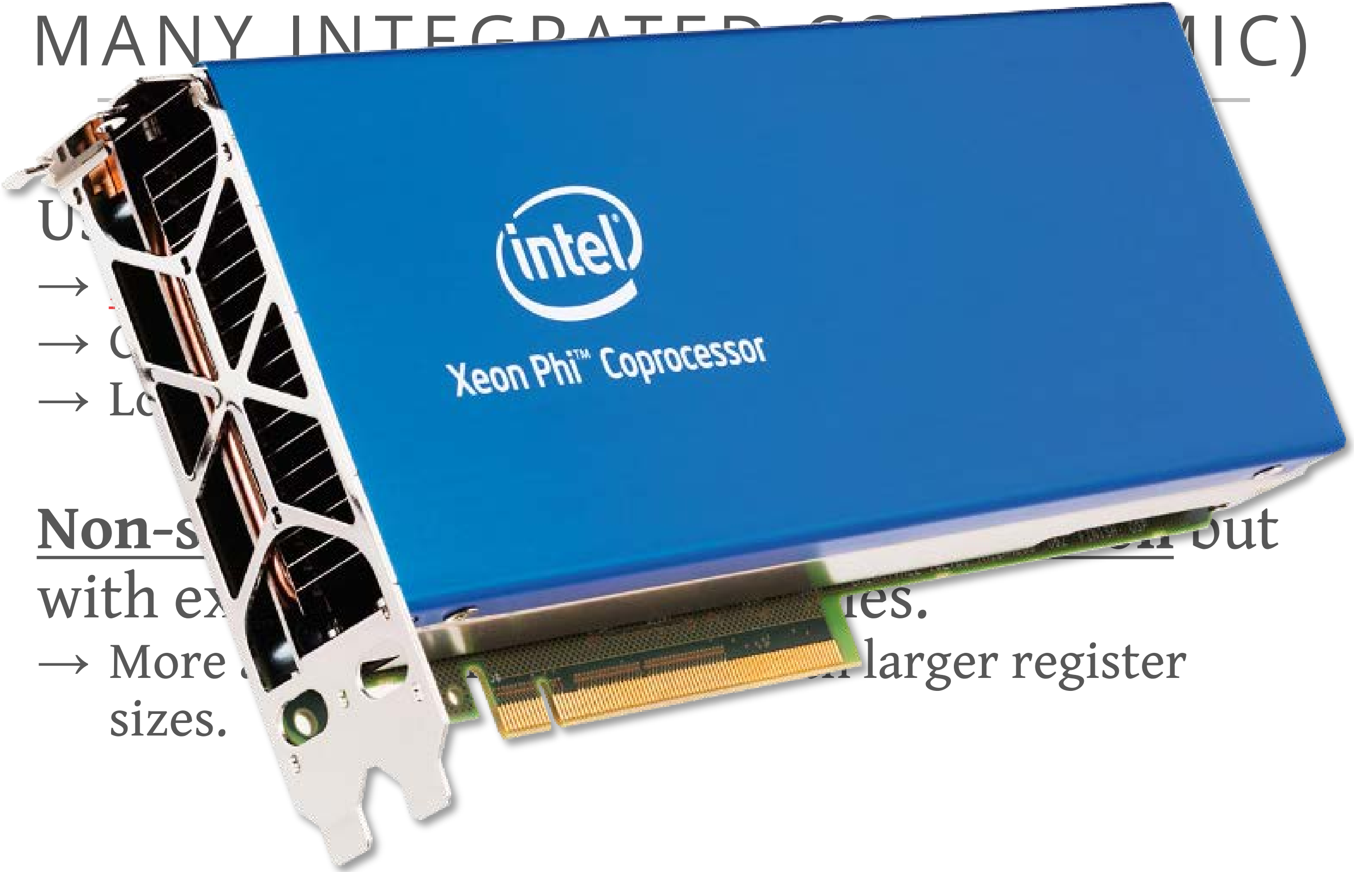
→ Cores = Intel P54C (aka Pentium from the 1990s).

→ Low power consumption and area per core.

Non-superscalar and **in-order execution** but with expanded SIMD capabilities.

→ More advanced instructions with larger register sizes.

MANY INTEGRATED ... (MIC)



U...

→ ...

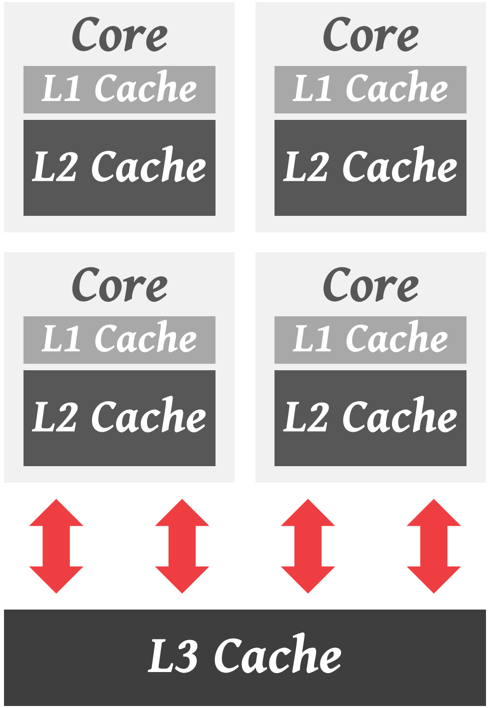
→ C...

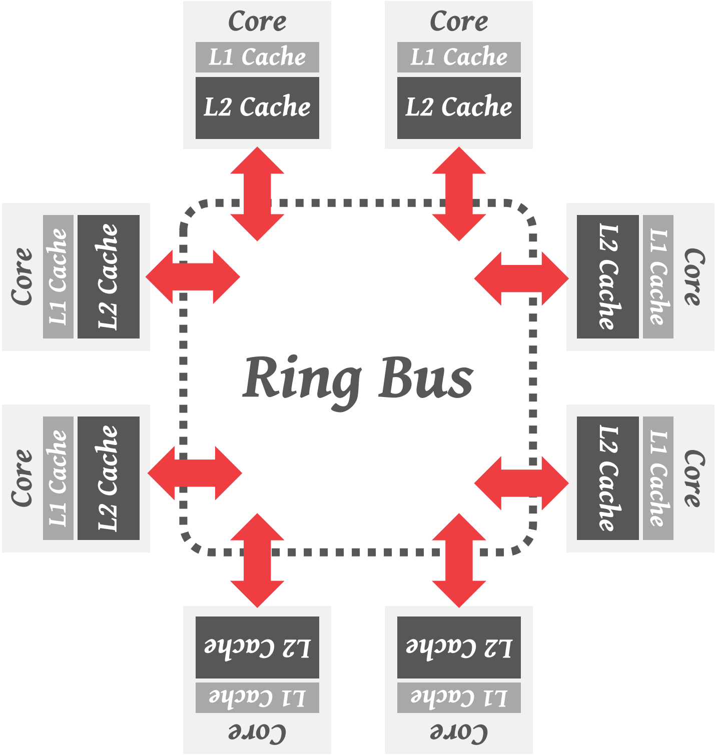
→ Lo...

Non-s... ...on but with ex... ...es.

→ More ... larger register sizes.

MULTI-CORE VS. MIC





Source: Orestis Polychroniou

VECTORIZATION

A program is converted from a scalar implementation that processes a single pair of operands at a time, to a vector implementation that processes one operation on multiple pairs of operands at once.

AUTOMATIC VECTORIZATION

The compiler can identify when instructions inside of a loop can be rewritten as a vectorized operation.

Works for simple loops only and is rare in database operators. Requires hardware support for SIMD instructions.

MANUAL VECTORIZATION

Linear Access Operators

→ Predicate evaluation
→ Compression

Ad-hoc Vectorization

→ Sorting
→ Merging

Composable Operations

→ Multi-way trees
→ Bucketized hash tables

Source: Orestis Polychroniou

SINGLE INSTRUCTION, MULTIPLE DATA

A class of CPU instructions that allow the processor to perform the same operation on multiple data points simultaneously.

All major ISAs have microarchitecture support SIMD operations.
→ **x86**: MMX, SSE, SSE2, SSE3, SSE4, AVX
→ **PowerPC**: Altivec
→ **ARM**: NEON

SIMD EXAMPLE

X + Y = Z

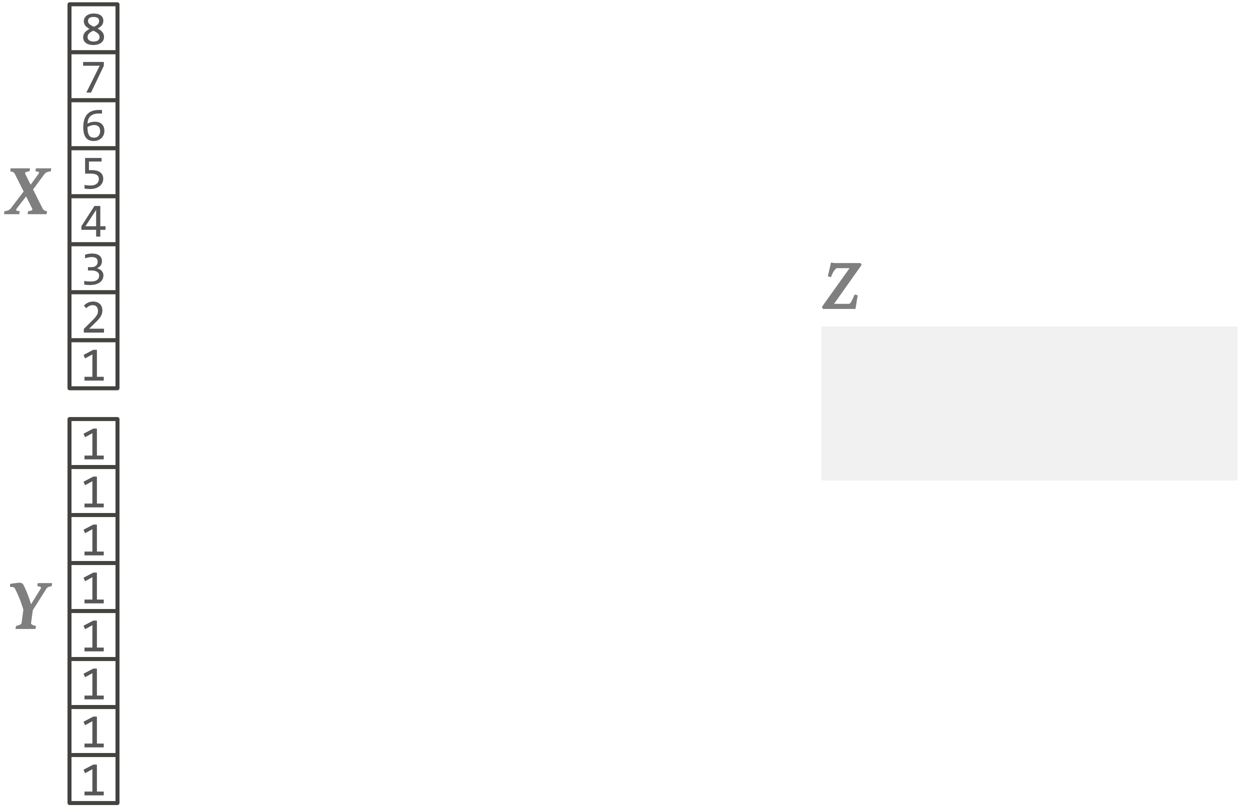
$$\begin{pmatrix} x_1 \\ x_2 \\ \ldots \\ x_n \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ \ldots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \ldots \\ x_n + y_n \end{pmatrix}$$

SIMD EXAMPLE

X + Y = Z

$$\begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \dots \\ x_n + y_n \end{pmatrix}$$

```
for (i=0; i<n; i++) {
  Z[i] = X[i] + Y[i];
}
```

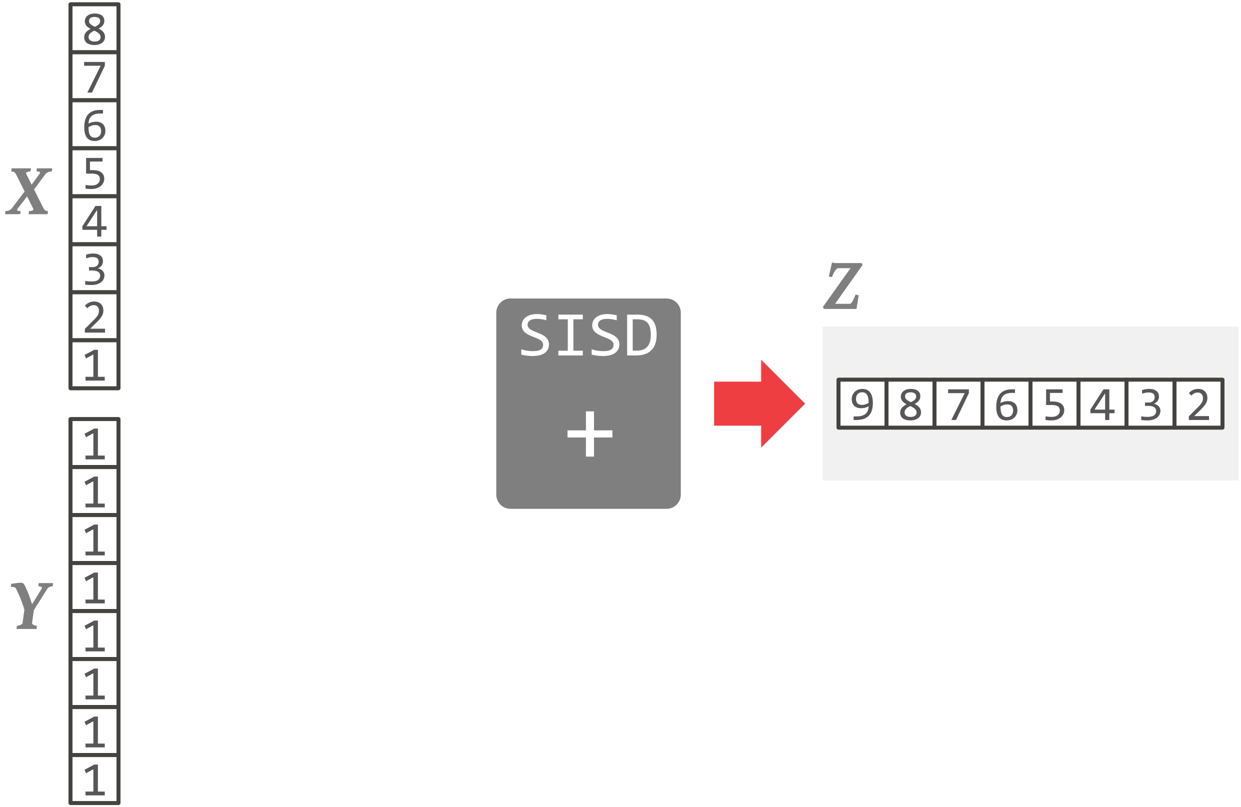


SIMD EXAMPLE

X + Y = Z

$$\begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \dots \\ x_n + y_n \end{pmatrix}$$

```
for (i=0; i<n; i++) {
  Z[i] = X[i] + Y[i];
}
```

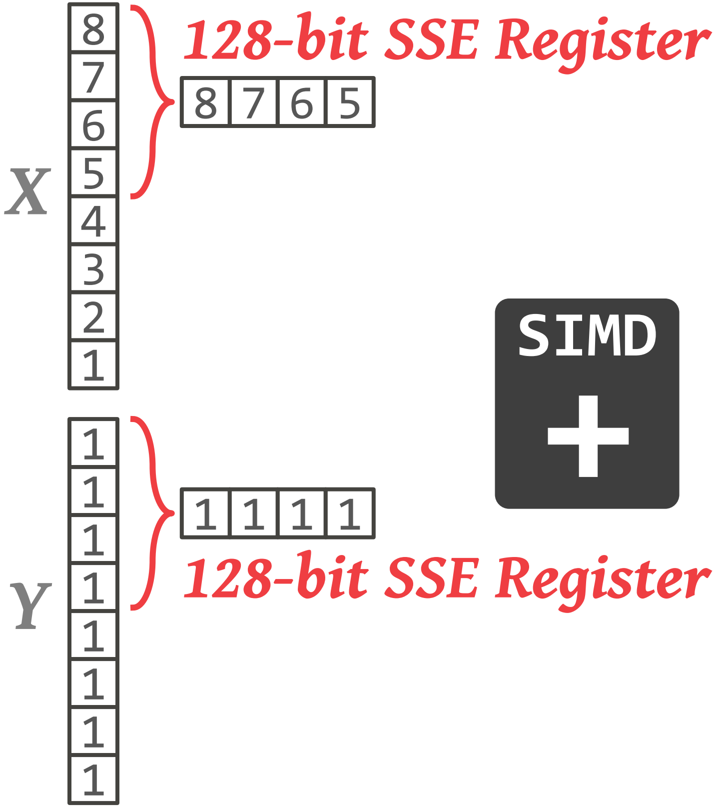


SIMD EXAMPLE

X + Y = Z

$$\begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \dots \\ x_n + y_n \end{pmatrix}$$

```
for (i=0; i<n; i++) {
  Z[i] = X[i] + Y[i];
}
```



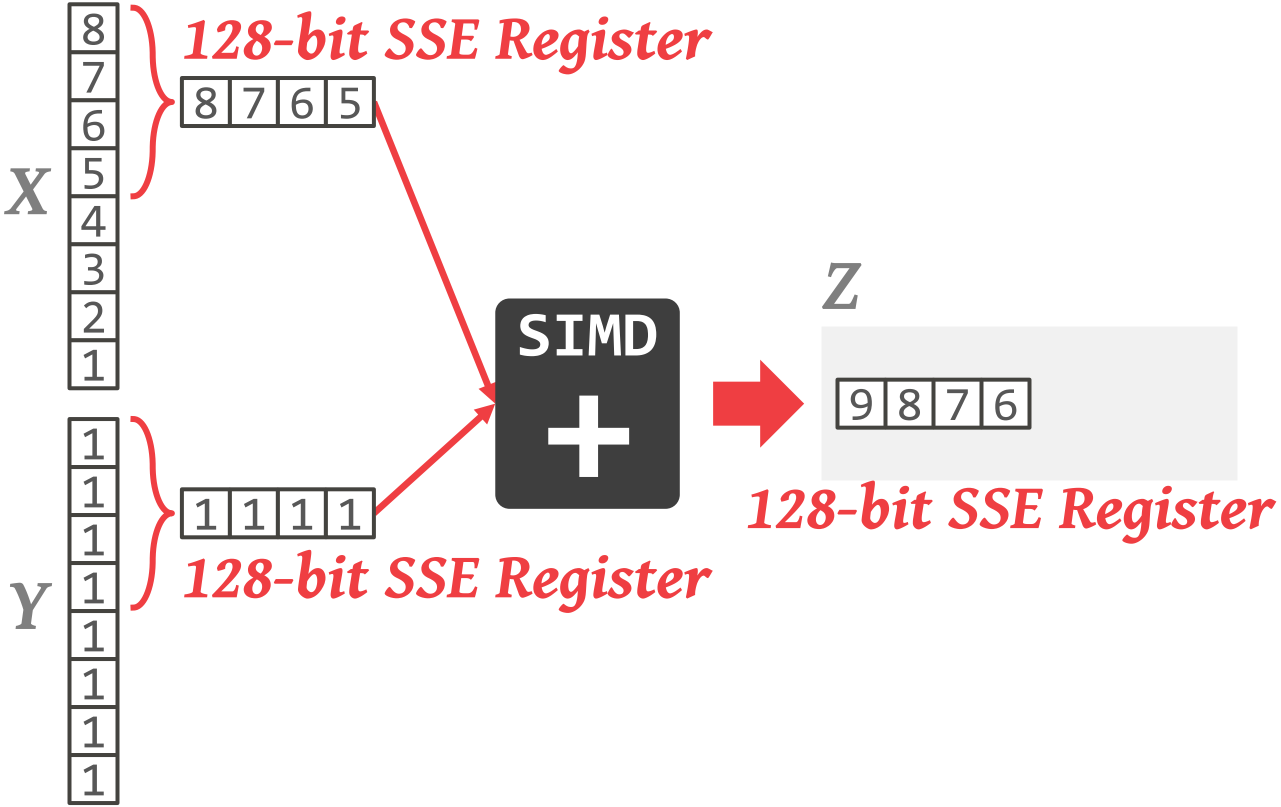
Z

SIMD EXAMPLE

X + Y = Z

$$\begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \dots \\ x_n + y_n \end{pmatrix}$$

```
for (i=0; i<n; i++) {
  Z[i] = X[i] + Y[i];
}
```

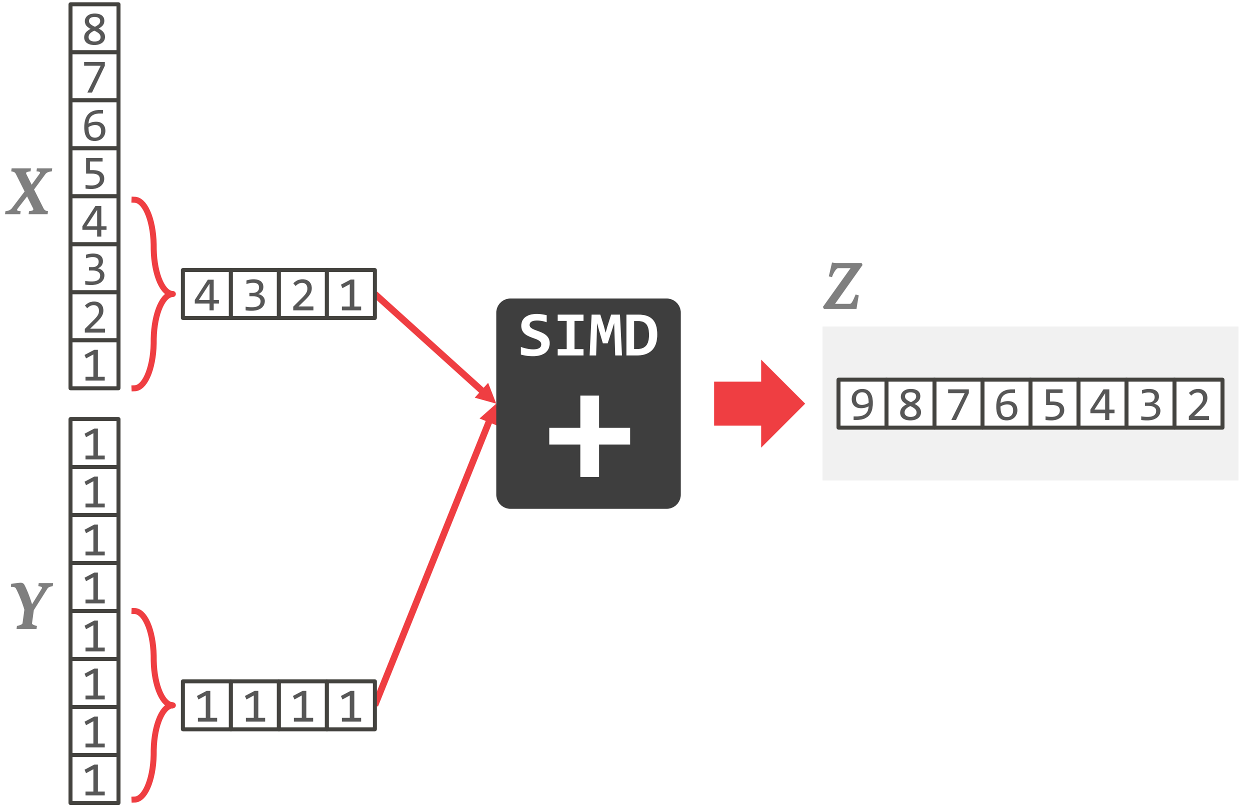


SIMD EXAMPLE

$X + Y = Z$

$$\begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ \cdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \cdots \\ x_n + y_n \end{pmatrix}$$

```
for (i=0; i<n; i++) {
  Z[i] = X[i] + Y[i];
}
```



STREAMING SIMD EXTENSIONS (SSE)

SSE is a collection SIMD instructions that target special 128-bit SIMD registers.

These registers can be packed with four 32-bit scalars after which an operation can be performed on each of the four elements simultaneously.

First introduced by Intel in 1999.

SSE INSTRUCTIONS (1)

Data Movement

→ Moving data in and out of vector registers

Arithmetic Operations

→ Apply operation on multiple data items (e.g., 2 doubles, 4 floats, 16 bytes)
→ Example: **ADD**, **SUB**, **MUL**, **DIV**, **SQRT**, **MAX**, **MIN**

Logical Instructions

→ Logical operations on multiple data items
→ Example: **AND**, **OR**, **XOR**, **ANDN**, **ANDPS**, **ANDNPS**

SSE INSTRUCTIONS (2)

Comparison Instructions

→ Comparing multiple data items (**==,<,<=,>,>=,!=**)

Shuffle instructions

→ Move data in between SIMD registers

Miscellaneous

→ Conversion: Transform data between x86 and SIMD registers.

→ Cache Control: Move data directly from SIMD registers to memory (bypassing CPU cache).

VECTORIZED DBMS ALGORITHMS

Principles for efficient vectorization by using **fundamental** vector operations to construct more advanced functionality.

→ Favor vertical vectorization by processing different input data per lane.

→ Maximize lane utilization by executing different things per lane subset.

RETHINKING SIMD VECTORIZATION FOR IN-MEMORY DATABASES
SIGMOD 2015

FUNDAMENTAL OPERATIONS

Selective Load

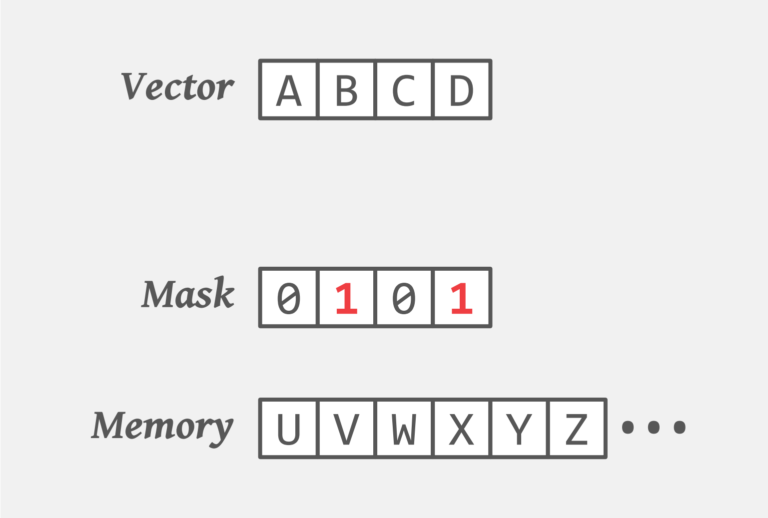
Selective Sore

Selective Gather

Selective Scatter

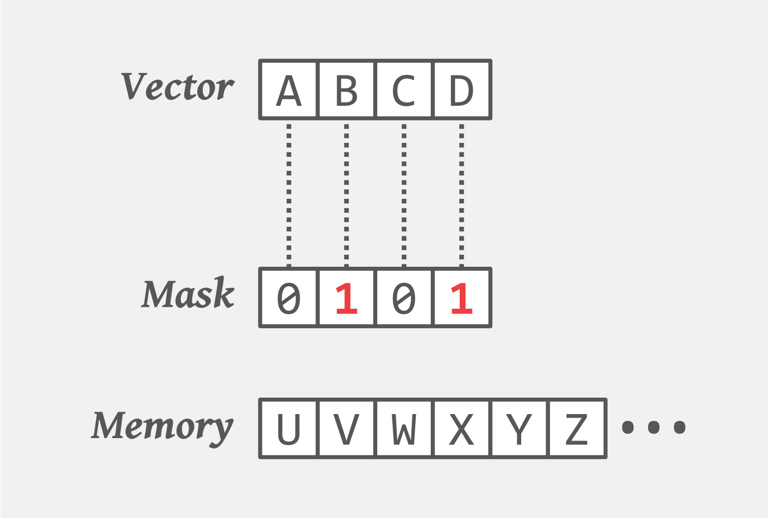
FUNDAMENTAL VECTOR OPERATIONS

Selective Load



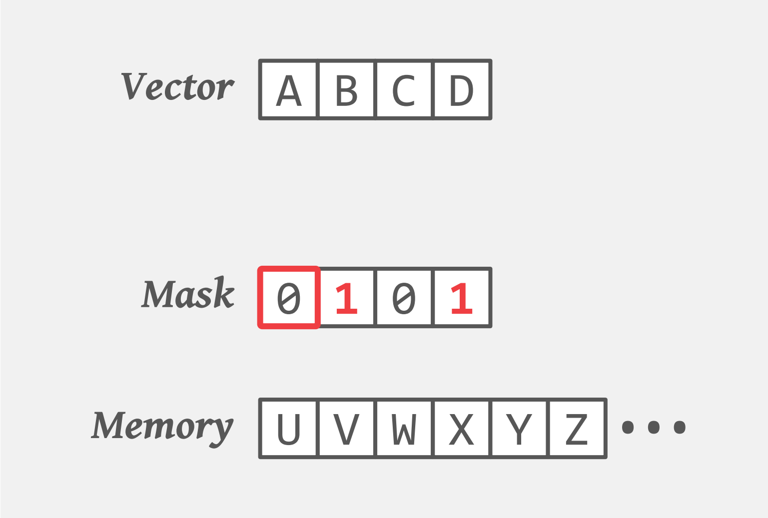
FUNDAMENTAL VECTOR OPERATIONS

Selective Load



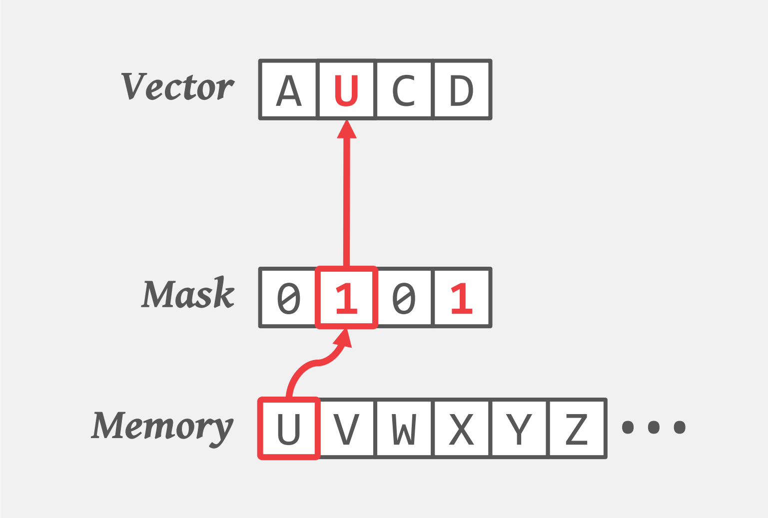
FUNDAMENTAL VECTOR OPERATIONS

Selective Load



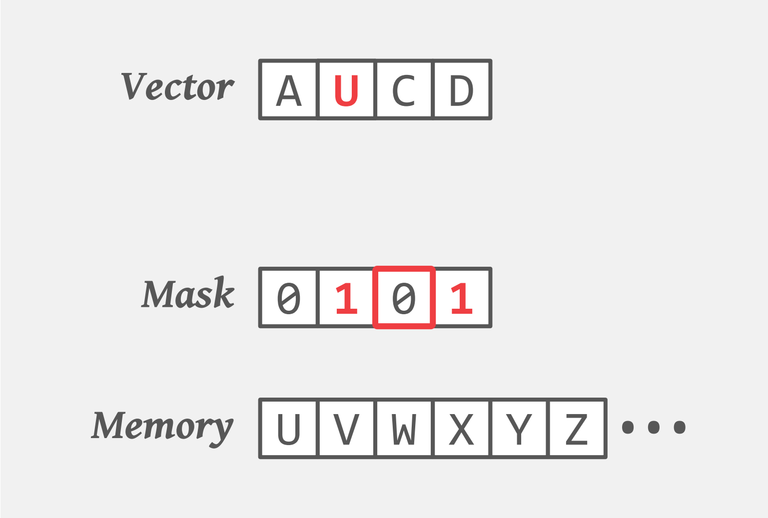
FUNDAMENTAL VECTOR OPERATIONS

Selective Load



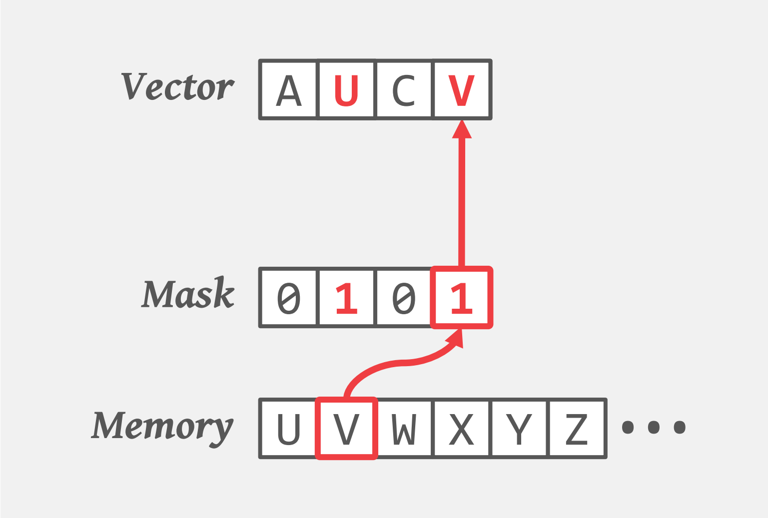
FUNDAMENTAL VECTOR OPERATIONS

Selective Load



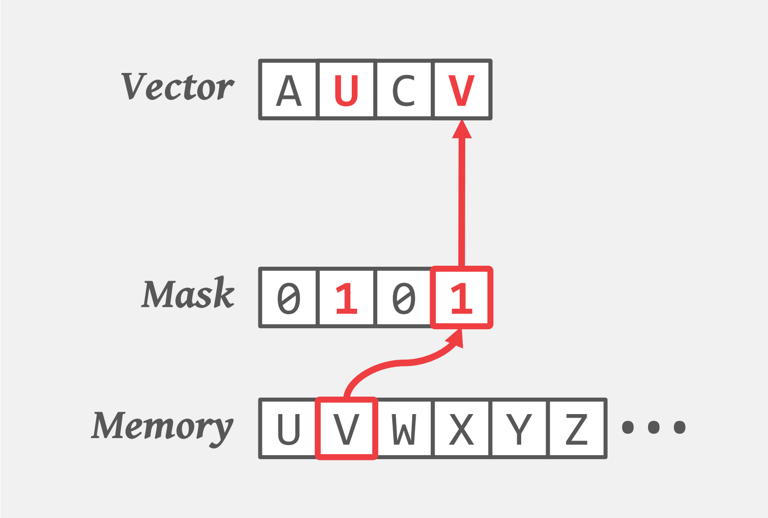
FUNDAMENTAL VECTOR OPERATIONS

Selective Load

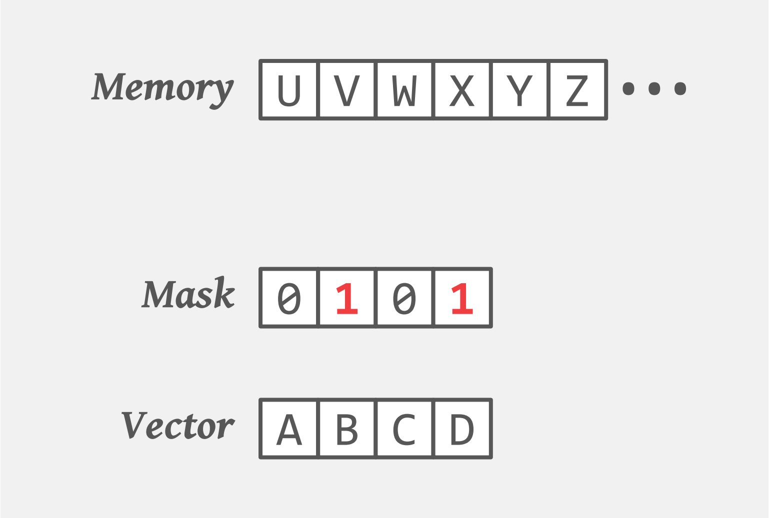


FUNDAMENTAL VECTOR OPERATIONS

Selective Load

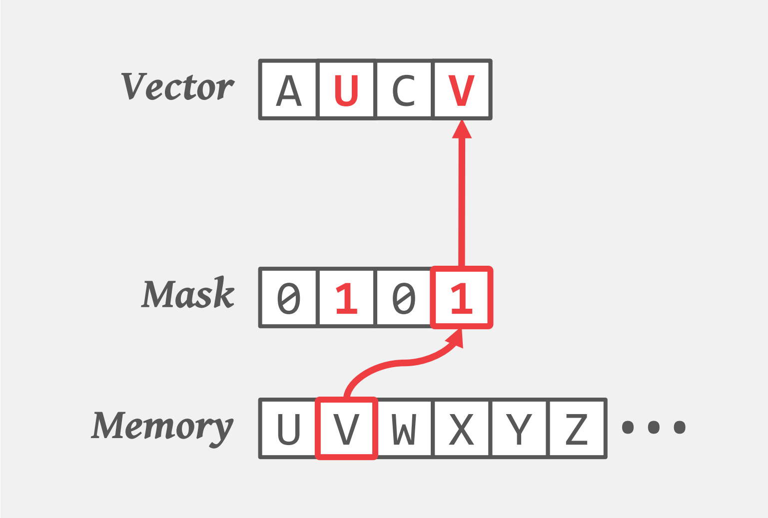


Selective Store

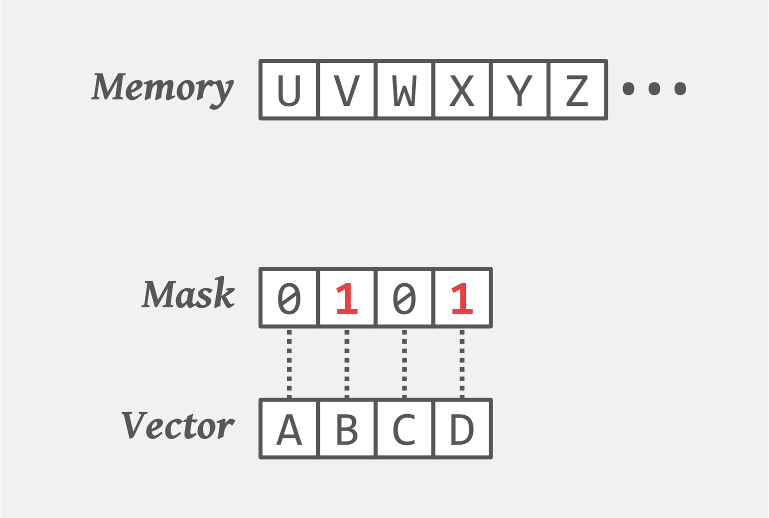


FUNDAMENTAL VECTOR OPERATIONS

Selective Load

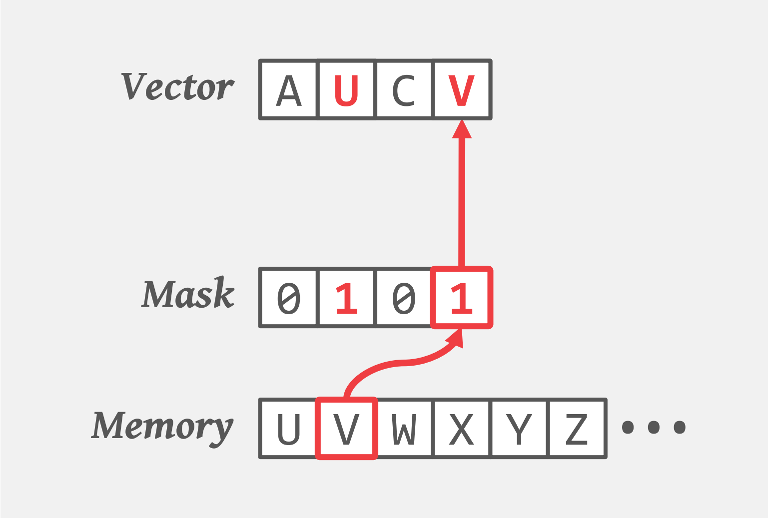


Selective Store

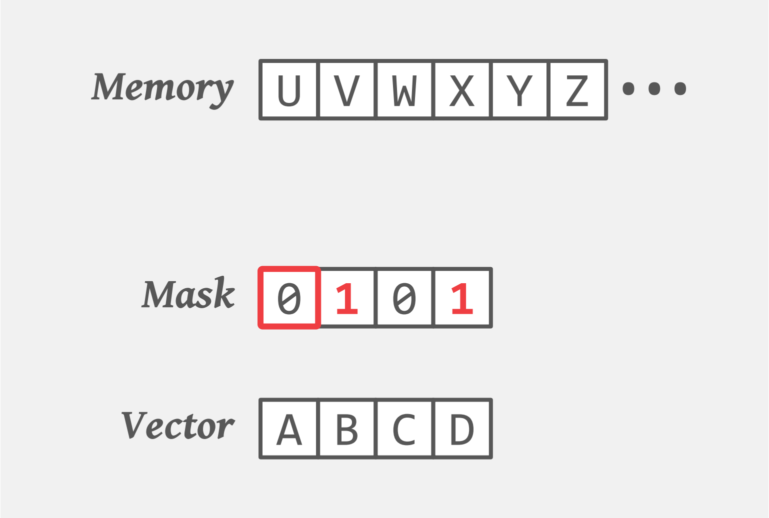


FUNDAMENTAL VECTOR OPERATIONS

Selective Load

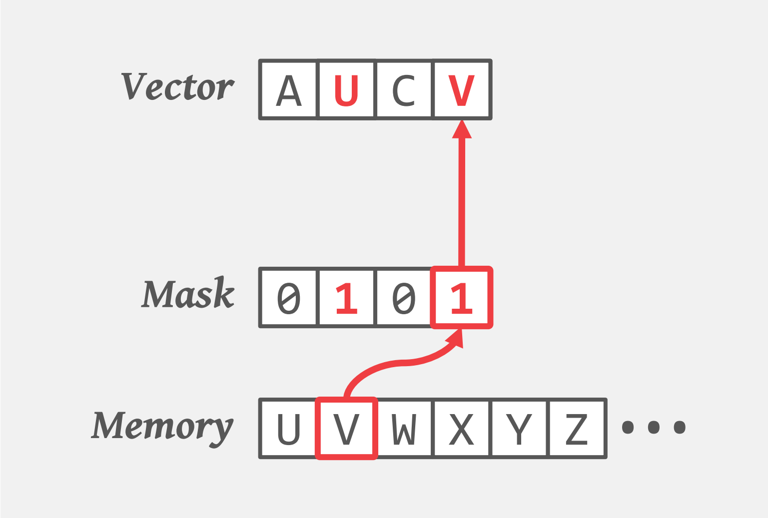


Selective Store

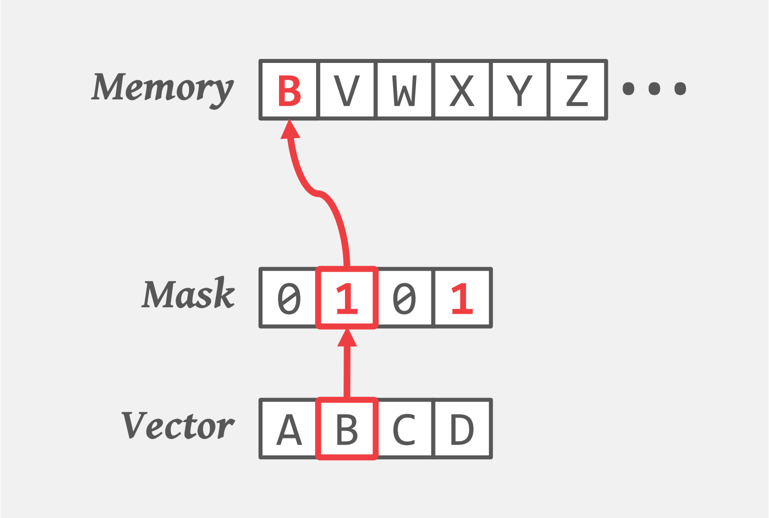


FUNDAMENTAL VECTOR OPERATIONS

Selective Load

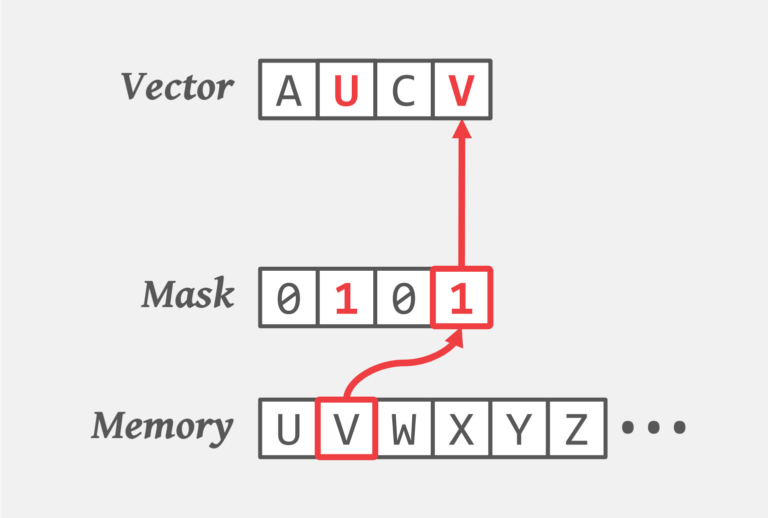


Selective Store

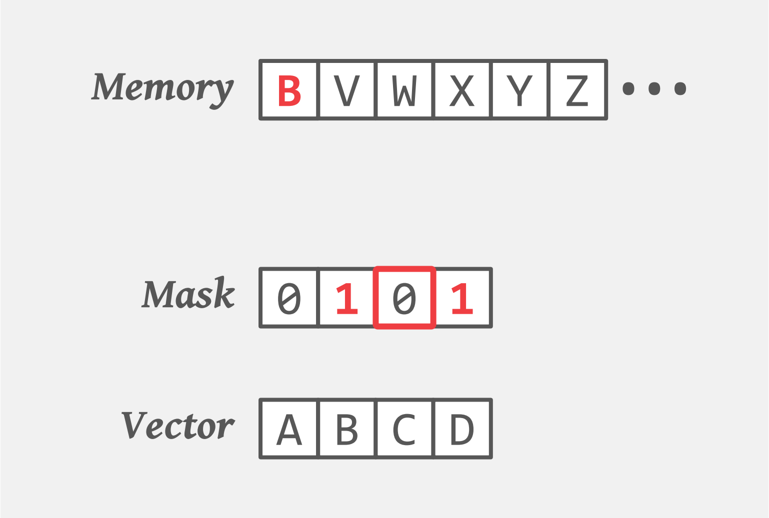


FUNDAMENTAL VECTOR OPERATIONS

Selective Load

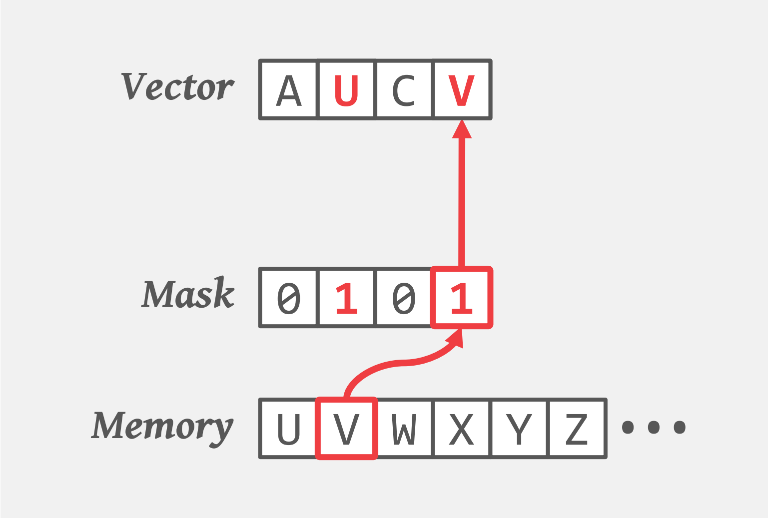


Selective Store

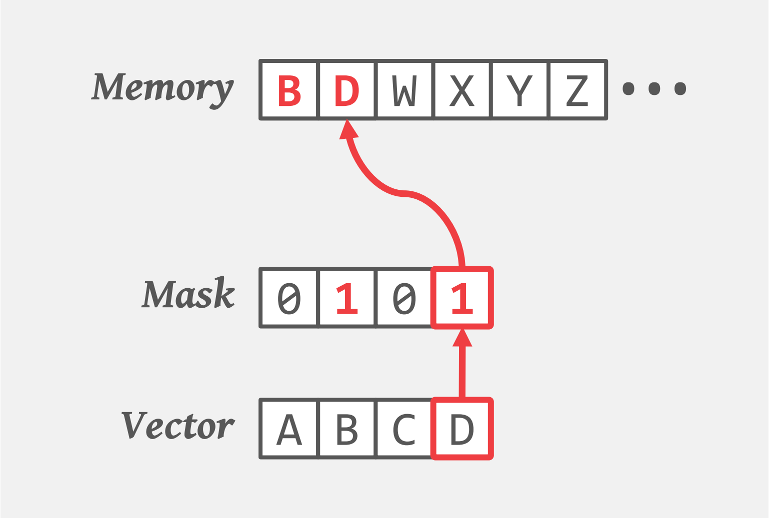


FUNDAMENTAL VECTOR OPERATIONS

Selective Load



Selective Store



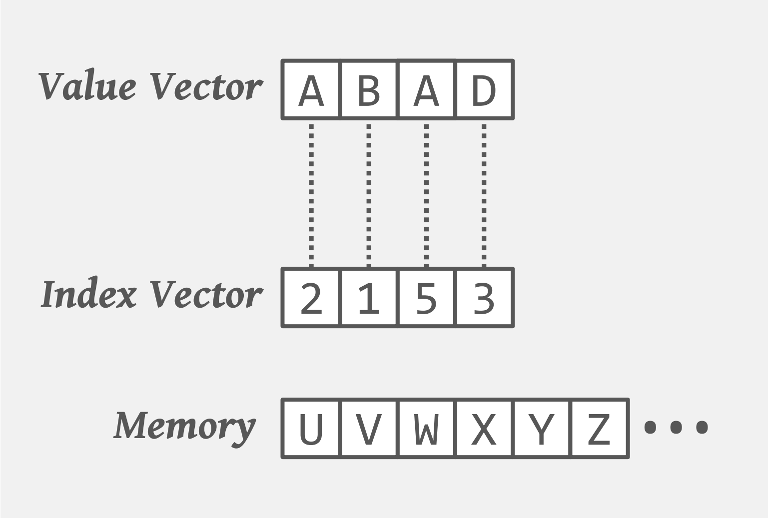
FUNDAMENTAL VECTOR OPERATIONS

Selective Gather

Value Vector	A	B	A	D			
Index Vector	2	1	5	3			
Memory	U	V	W	X	Y	Z	• • •

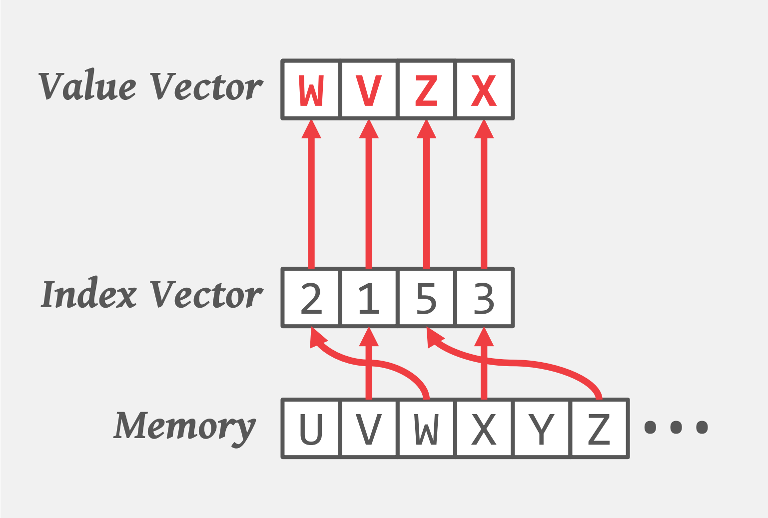
FUNDAMENTAL VECTOR OPERATIONS

Selective Gather



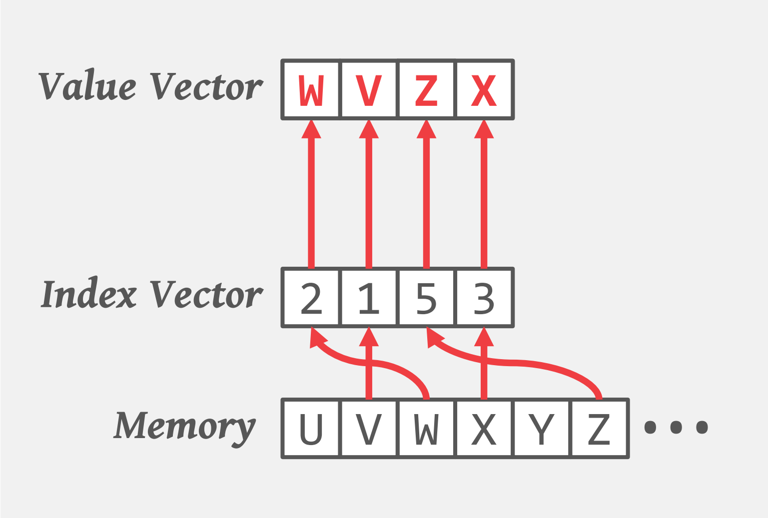
FUNDAMENTAL VECTOR OPERATIONS

Selective Gather

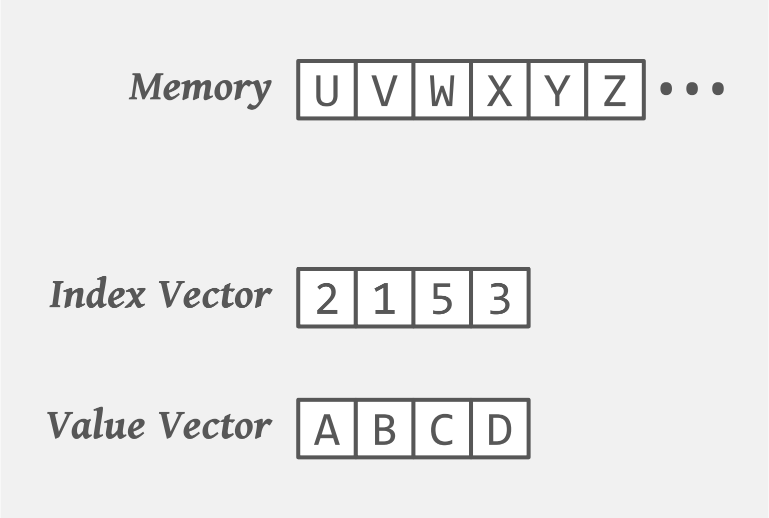


FUNDAMENTAL VECTOR OPERATIONS

Selective Gather

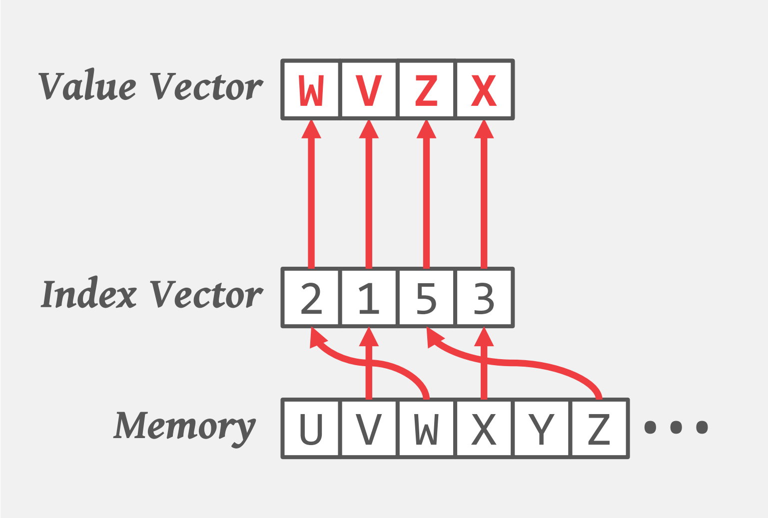


Selective Scatter

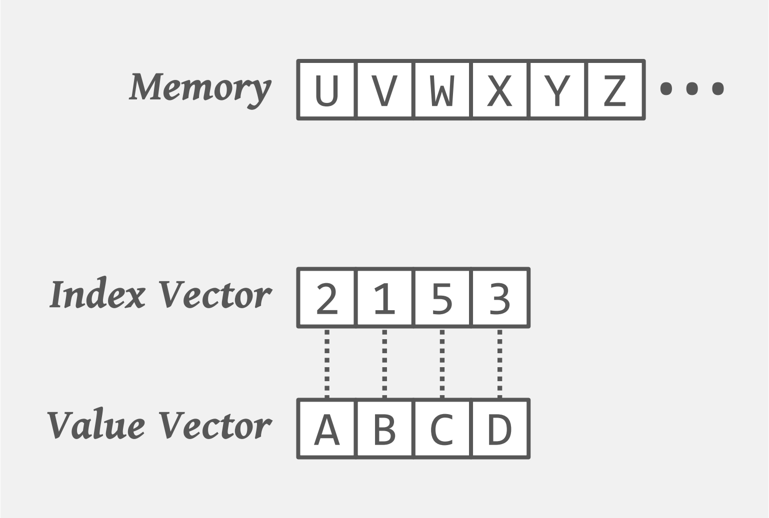


FUNDAMENTAL VECTOR OPERATIONS

Selective Gather

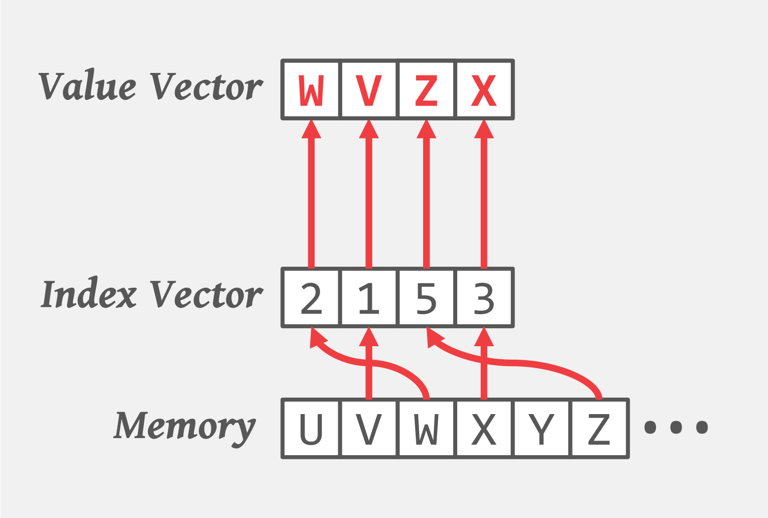


Selective Scatter

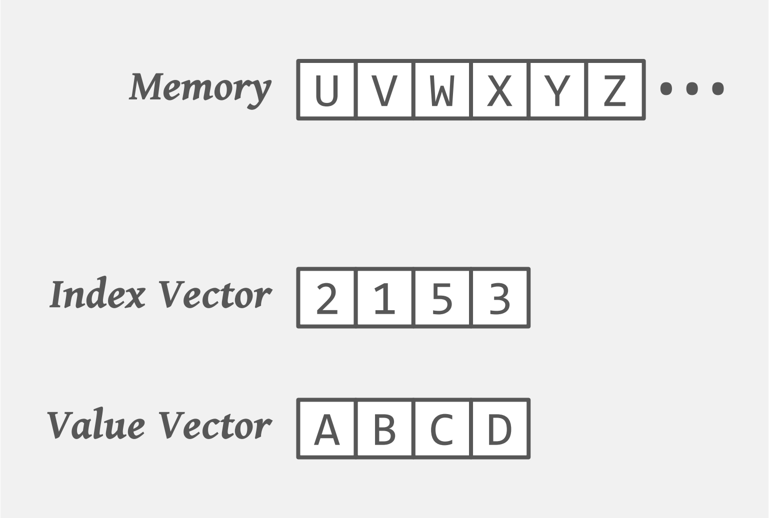


FUNDAMENTAL VECTOR OPERATIONS

Selective Gather

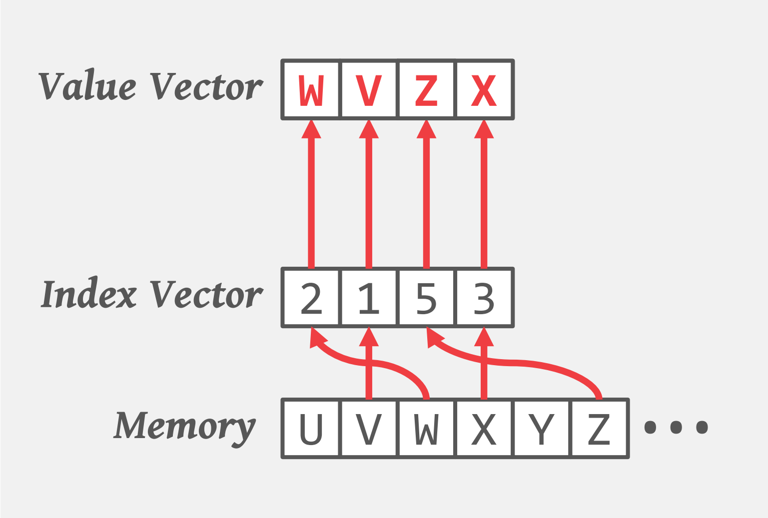


Selective Scatter

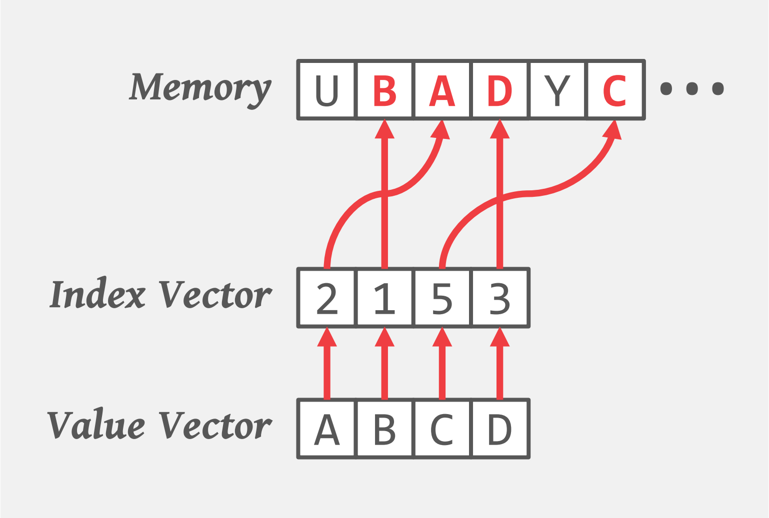


FUNDAMENTAL VECTOR OPERATIONS

Selective Gather



Selective Scatter



ISSUES

Gathers and scatters are not really executed in parallel because the L1 cache only allows one or two distinct accesses per cycle.

Gathers are only supported in modern CPUs.

Selective loads and stores are also emulated in Xeon CPUs using vector permutations.

VECTORIZED OPERATORS

Selection Scans

Hash Tables

Partitioning

Paper provides additional info:

→ Joins, Sorting, Bloom filters.

RETHINKING SIMD VECTORIZATION FOR IN-MEMORY DATABASES
SIGMOD 2015

SELECTION SCANS

```
SELECT * FROM table
 WHERE key >= $(low)
   AND key <= $(high)
```

SELECTION SCANS

Scalar (Branching)

```
i = 0
for t in table:
  key = t.key
  if (key≥low) && (key≤high):
    copy(t, output[i])
    i = i + 1
```

SELECTION SCANS

Scalar (Branching)

```
i = 0
for t in table:
  key = t.key
  if (key≥low) && (key≤high):
    copy(t, output[i])
    i = i + 1
```

SELECTION SCANS

Scalar (Branching)

```
i = 0
for t in table:
  key = t.key
  if (key≥low) && (key≤high):
    copy(t, output[i])
    i = i + 1
```

Scalar (Branchless)

```
i = 0
for t in table:
  copy(t, output[i])
  key = t.key
  m = (key≥low ? 1 : 0) &&
     ↪(key≤high ? 1 : 0)
  i = i + m
```

SELECTION SCANS

Scalar (Branching)

```
i = 0
for t in table:
  key = t.key
  if (key≥low) && (key≤high):
    copy(t, output[i])
    i = i + 1
```

Scalar (Branchless)

```
i = 0
for t in table:
  copy(t, output[i])
  key = t.key
  m = (key≥low ? 1 : 0) &&
     ↪(key≤high ? 1 : 0)
  i = i + m
```

SELECTION SCANS

Vectorized

```
i = 0
for vt in table:
  simdLoad(vt.key, vk)
  vm = (vk≥low ? 1 : 0) &&
       ↪(vk≤high ? 1 : 0)
  if vm ≠ false:
    simdStore(vt, vm, output[i])
    i = i + |vm≠false|
```

SELECTION SCANS

Vectorized

```
i = 0
for vt in table:
  simdLoad(vt.key, vk)
  vm = (vk≥low ? 1 : 0) &&
     ↪(vk≤high ? 1 : 0)
  if vm ≠ false:
    simdStore(vt, vm, output[i])
    i = i + |vm≠false|
```

SELECTION SCANS

Vectorized

```
i = 0
for vₜ in table:
  simdLoad(vₜ.key, vₖ)
  vₘ = (vₖ≥low ? 1 : 0) &&
      ↪(vₖ≤high ? 1 : 0)
  if vₘ ≠ false:
    simdStore(vₜ, vₘ, output[i])
    i = i + |vₘ≠false|
```

SELECTION SCANS

Vectorized

```
i = 0
for vt in table:
  simdLoad(vt.key, vk)
  vm = (vk≥low ? 1 : 0) &&
     ↪(vk≤high ? 1 : 0)
  if vm ≠ false:
    simdStore(vt, vm, output[i])
    i = i + |vm≠false|
```

SELECTION SCANS

Vectorized

```
i = 0
for vₜ in table:
  simdLoad(vₜ.key, vₖ)
  vₘ = (vₖ≥low ? 1 : 0) &&
     ↪(vₖ≤high ? 1 : 0)
  if vₘ ≠ false:
    simdStore(vₜ, vₘ, output[i])
    i = i + |vₘ≠false|
```

SELECTION SCANS

Vectorized

```
i = 0
for vt in table:
  simdLoad(vt.key, vk)
  vm = (vk≥low ? 1 : 0) &&
     ↪(vk≤high ? 1 : 0)
  if vm ≠ false:
    simdStore(vt, vm, output[i])
    i = i + |vm≠false|
```

SELECTION SCANS

Vectorized

```
i = 0
for vₜ in table:
  simdLoad(vₜ.key, vₖ)
  vₘ = (vₖ≥low ? 1 : 0) &&
     ↪(vₖ≤high ? 1 : 0)
  if vₘ ≠ false:
    simdStore(vₜ, vₘ, output[i])
    i = i + |vₘ≠false|
```

```
SELECT * FROM table
 WHERE key >= "O" AND key <= "U"
```

SELECTION SCANS

Vectorized

```
i = 0
for vₜ in table:
  simdLoad(vₜ.key, vₖ)
  vₘ = (vₖ≥low ? 1 : 0) &&
     ↪(vₖ≤high ? 1 : 0)
  if vₘ ≠ false:
    simdStore(vₜ, vₘ, output[i])
    i = i + |vₘ≠false|
```

ID	KEY
1	J
2	O
3	Y
4	S
5	U
6	X

```
SELECT * FROM table
 WHERE key >= “O” AND key <= “U”
```

SELECTION SCANS

Vectorized

```
i = 0
for v_t in table:
  simdLoad(v_t.key, v_k)
  v_m = (v_k≥low ? 1 : 0) &&
      ↪(v_k≤high ? 1 : 0)
  if v_m ≠ false:
    simdStore(v_t, v_m, output[i])
    i = i + |v_m≠false|
```

ID	KEY
1	J
2	O
3	Y
4	S
5	U
6	X

Key Vector

J	O	Y	S	U	X

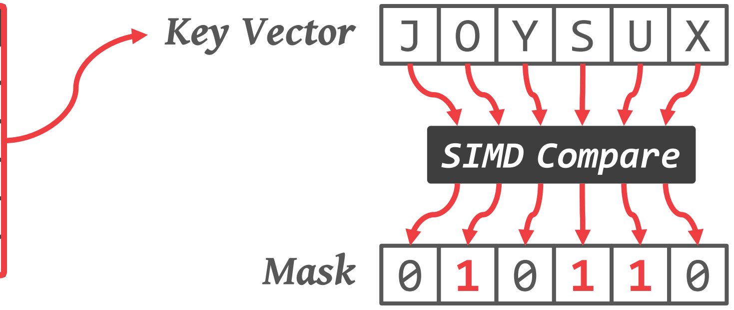
```
SELECT * FROM table
 WHERE key >= "O" AND key <= "U"
```

SELECTION SCANS

Vectorized

```
i = 0
for v_t in table:
  simdLoad(v_t.key, v_k)
  v_m = (v_k≥low ? 1 : 0) &&
      ↪(v_k≤high ? 1 : 0)
  if v_m ≠ false:
    simdStore(v_t, v_m, output[i])
    i = i + |v_m≠false|
```

ID	KEY
1	J
2	O
3	Y
4	S
5	U
6	X




```
SELECT * FROM table
 WHERE key >= "O" AND key <= "U"
```

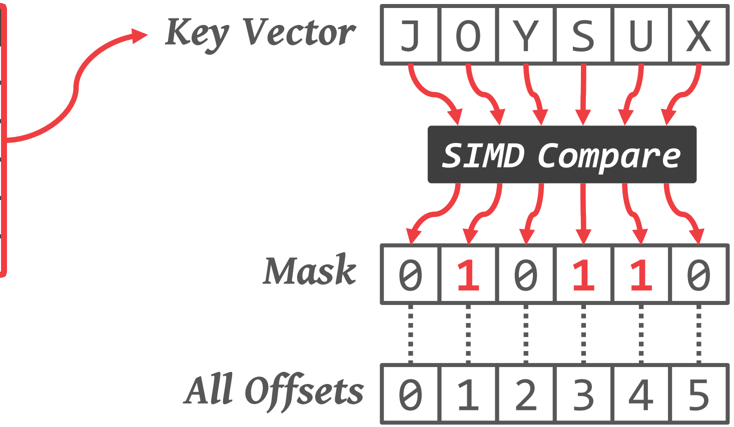
SELECTION SCANS

Vectorized

```
i = 0
for v_t in table:
  simdLoad(v_t.key, v_k)
  v_m = (v_k≥low ? 1 : 0) &&
       ↪(v_k≤high ? 1 : 0)
  if v_m ≠ false:
    simdStore(v_t, v_m, output[i])
    i = i + |v_m≠false|
```

```
SELECT * FROM table
 WHERE key >= "O" AND key <= "U"
```

ID	KEY
1	J
2	O
3	Y
4	S
5	U
6	X



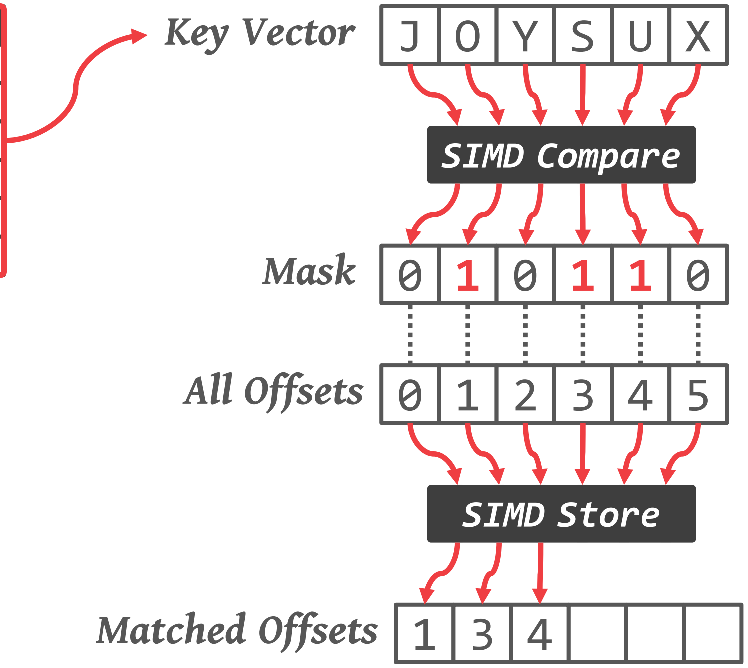
SELECTION SCANS

Vectorized

```
i = 0
for v_t in table:
  simdLoad(v_t.key, v_k)
  v_m = (v_k≥low ? 1 : 0) &&
     ↪(v_k≤high ? 1 : 0)
  if v_m ≠ false:
    simdStore(v_t, v_m, output[i])
    i = i + |v_m≠false|
```

```
SELECT * FROM table
 WHERE key >= "O" AND key <= "U"
```

ID	KEY
1	J
2	O
3	Y
4	S
5	U
6	X



SELECTION SCANS

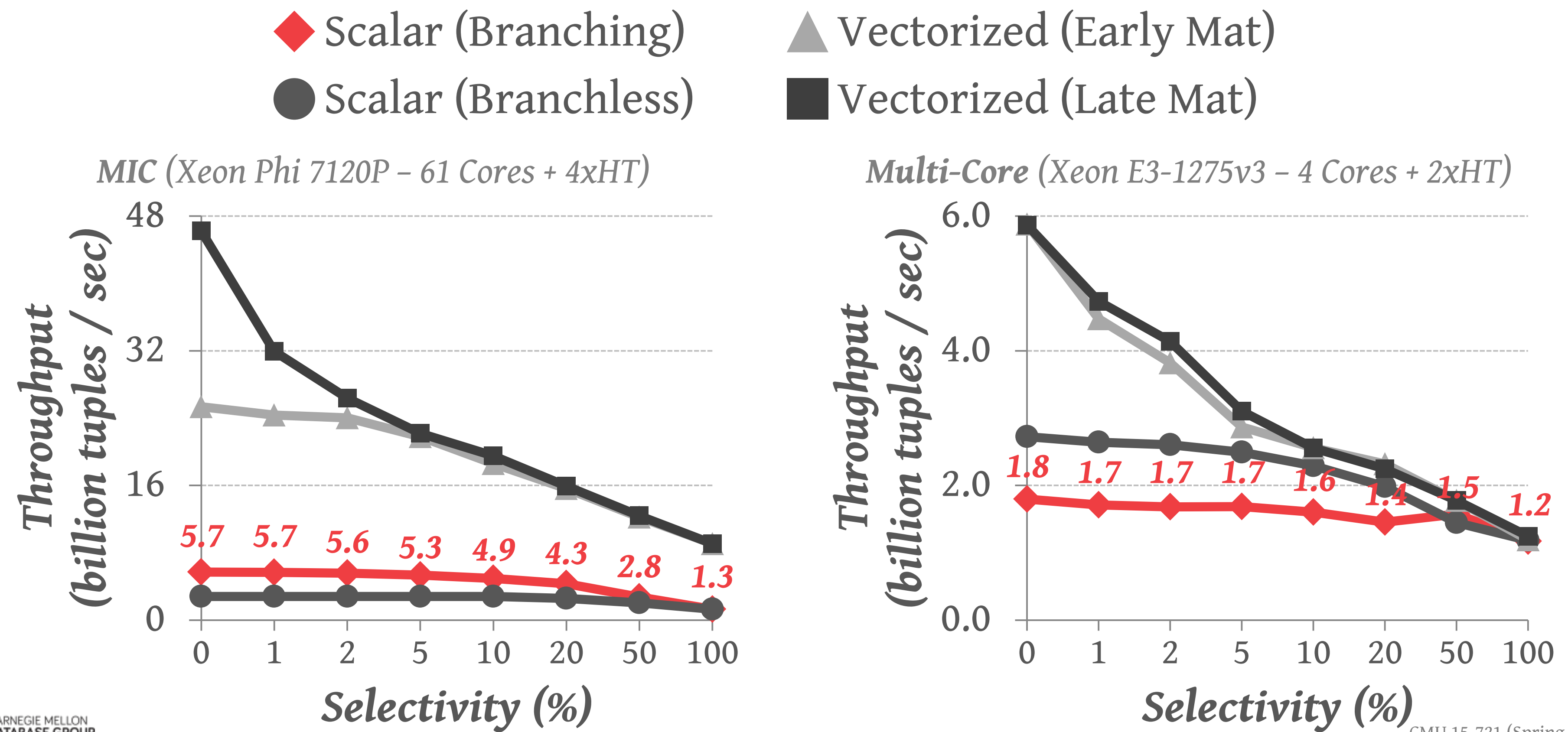
◆ Scalar (Branching) ▲ Vectorized (Early Mat)

● Scalar (Branchless) ■ Vectorized (Late Mat)

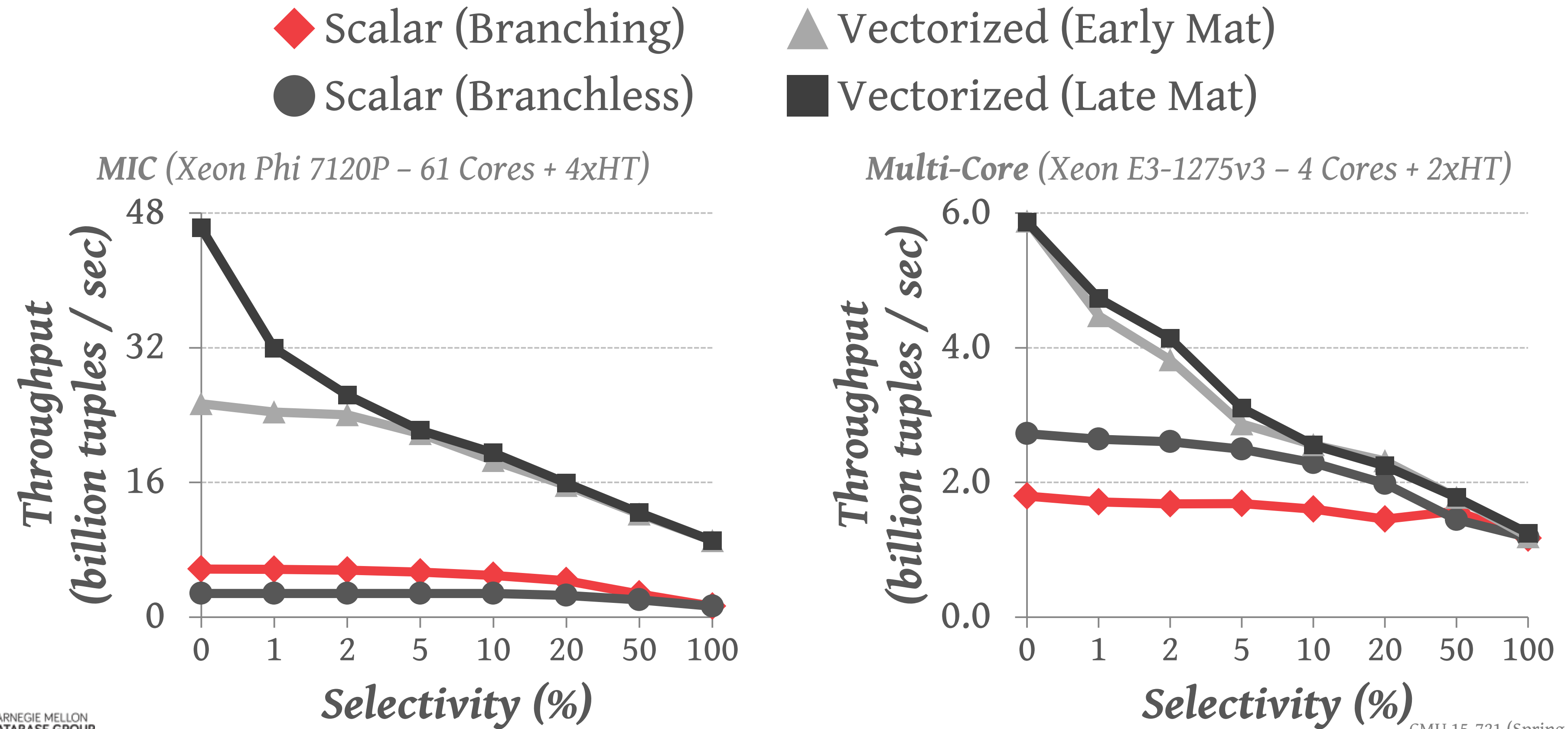
MIC *(Xeon Phi 7120P – 61 Cores + 4xHT)*

Multi-Core *(Xeon E3-1275v3 – 4 Cores + 2xHT)*

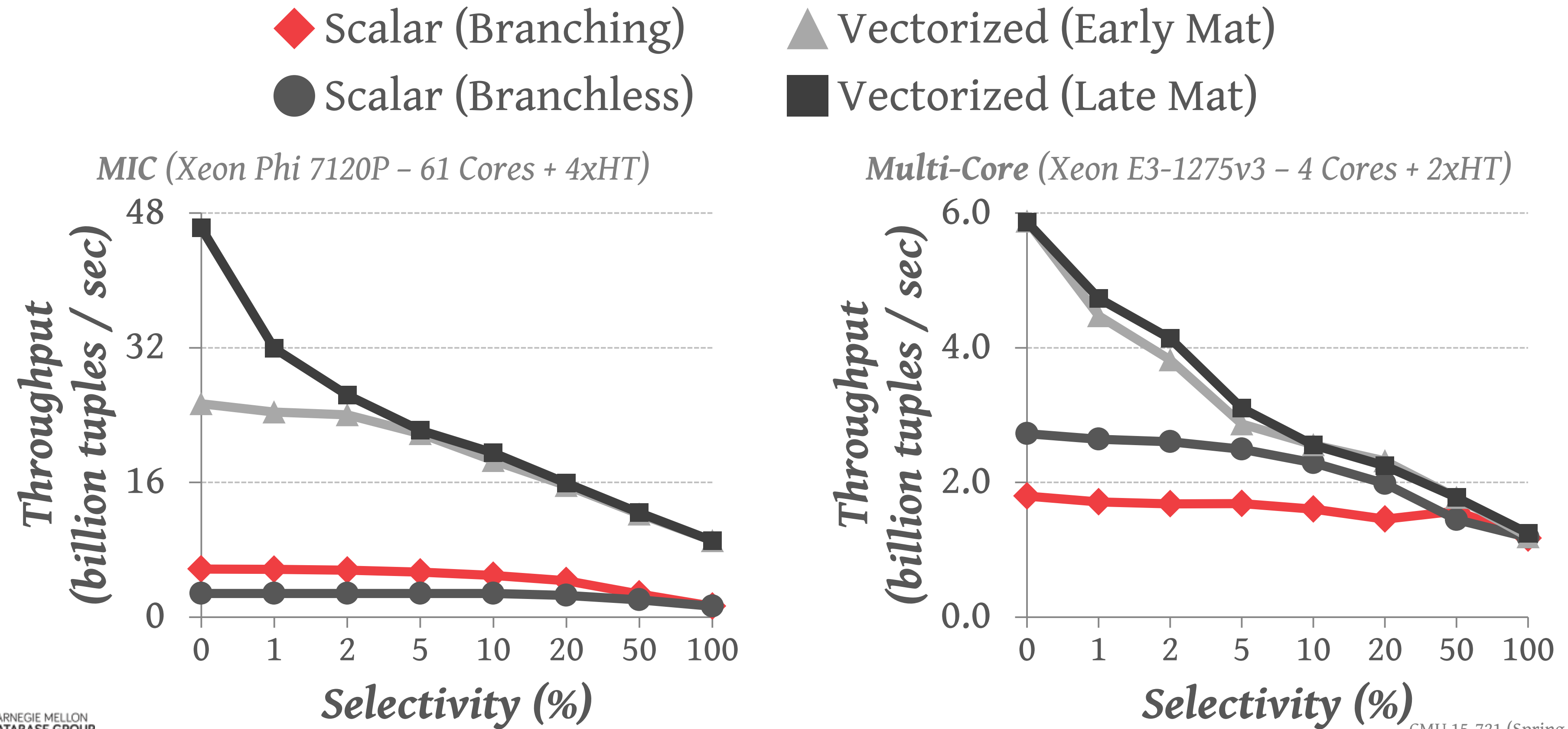
SELECTION SCANS



SELECTION SCANS



SELECTION SCANS



HASH TABLES – PROBING

Linear Probing Hash Table

KEY	PAYLOAD

HASH TABLES – PROBING

Scalar

Input Key

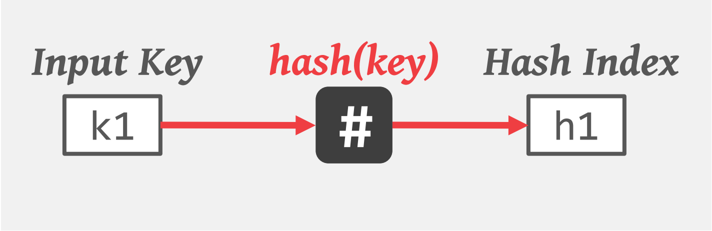
k1

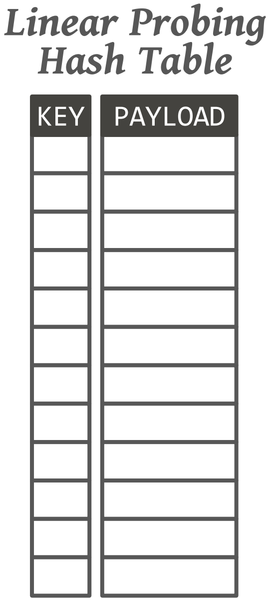
Linear Probing Hash Table

KEY	PAYLOAD

HASH TABLES – PROBING

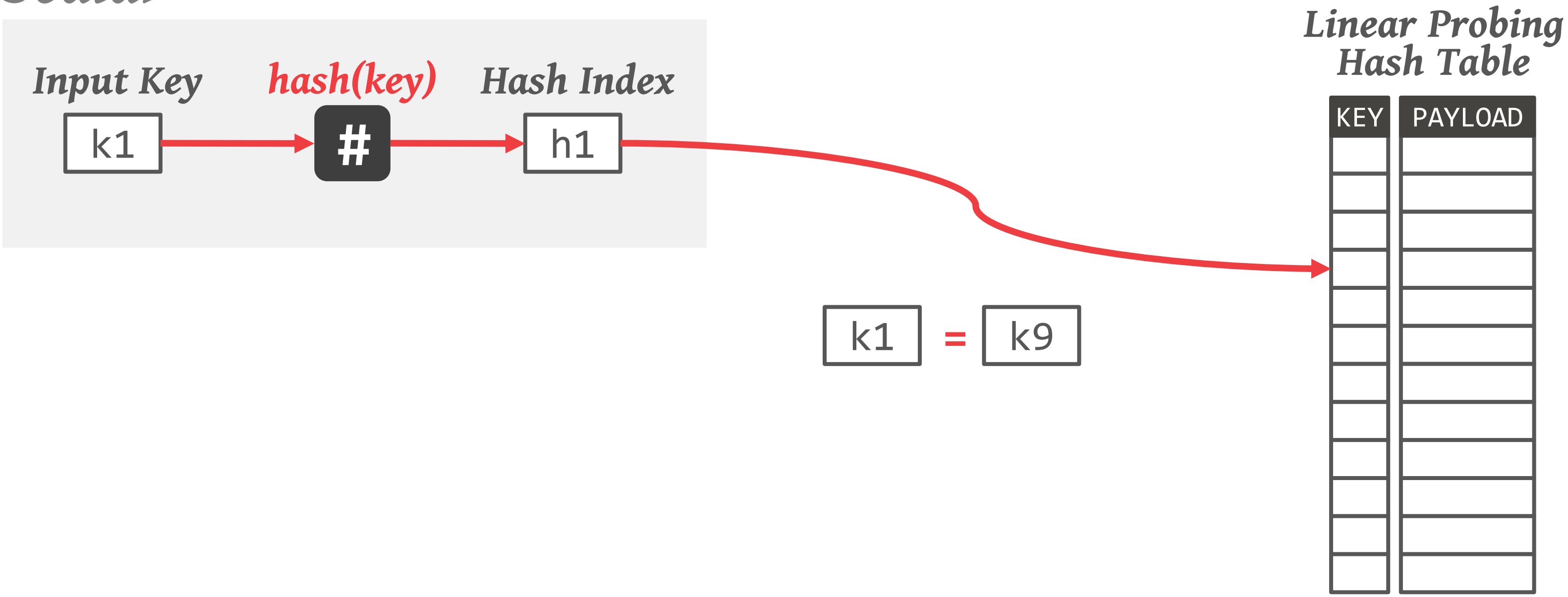
Scalar





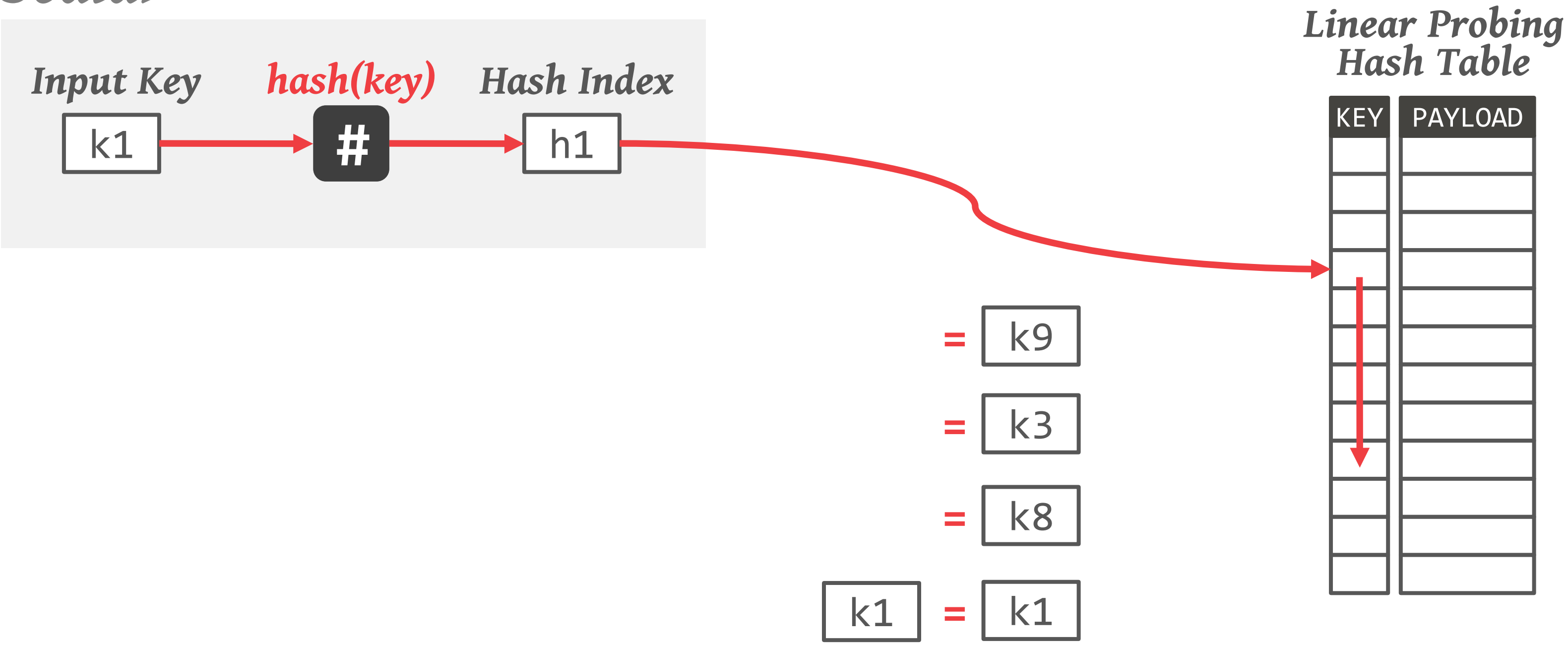
HASH TABLES – PROBING

Scalar



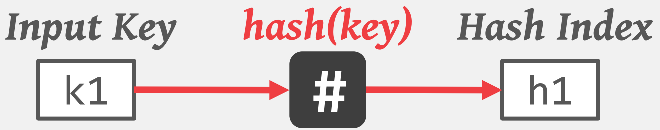
HASH TABLES – PROBING

Scalar



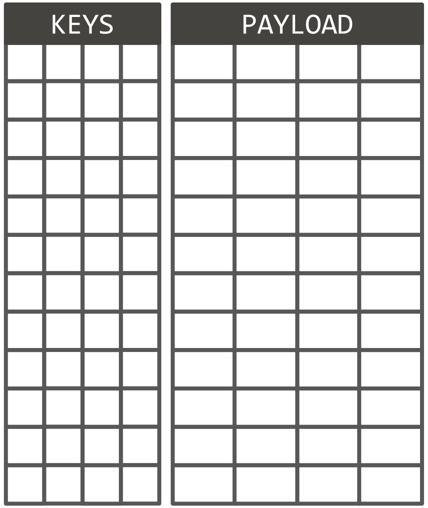
HASH TABLES – PROBING

Scalar



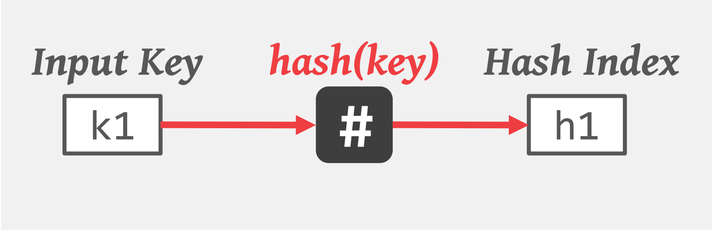
Vectorized (Horizontal)

Linear Probing
Bucketized Hash Table

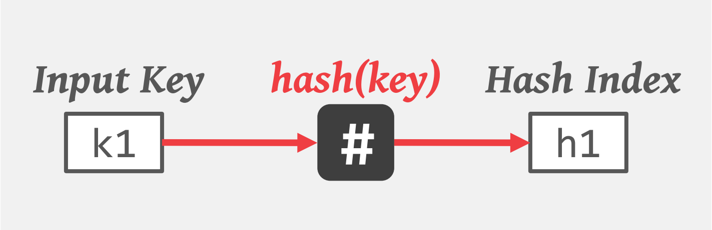


HASH TABLES – PROBING

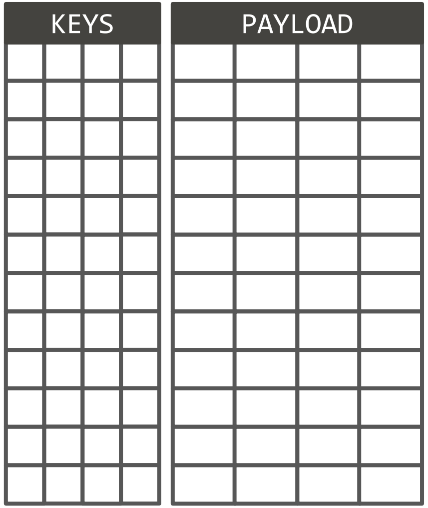
Scalar



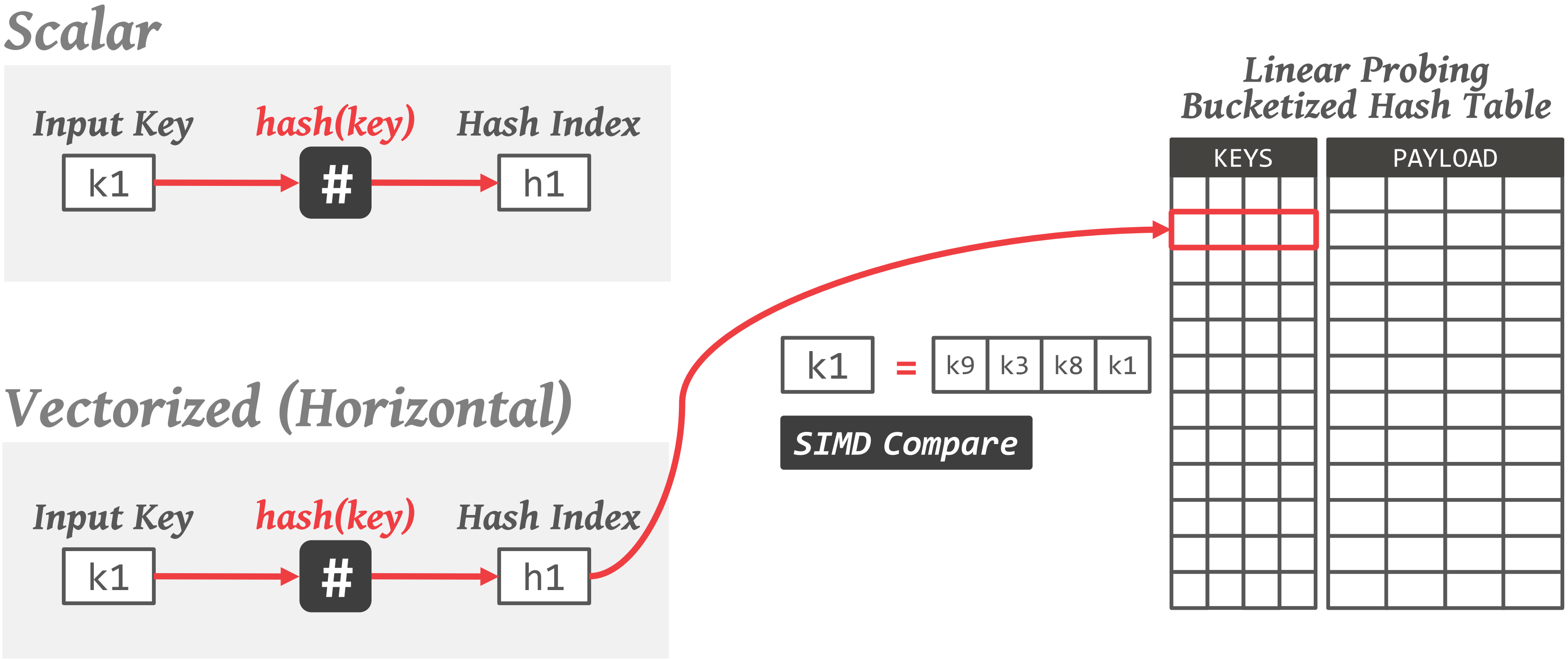
Vectorized (Horizontal)



Linear Probing Bucketized Hash Table

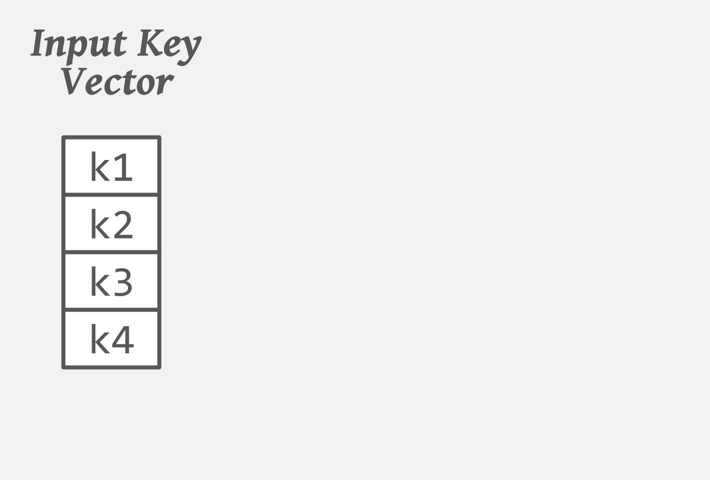


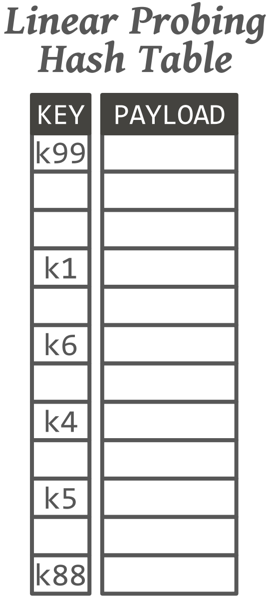
HASH TABLES – PROBING



HASH TABLES – PROBING

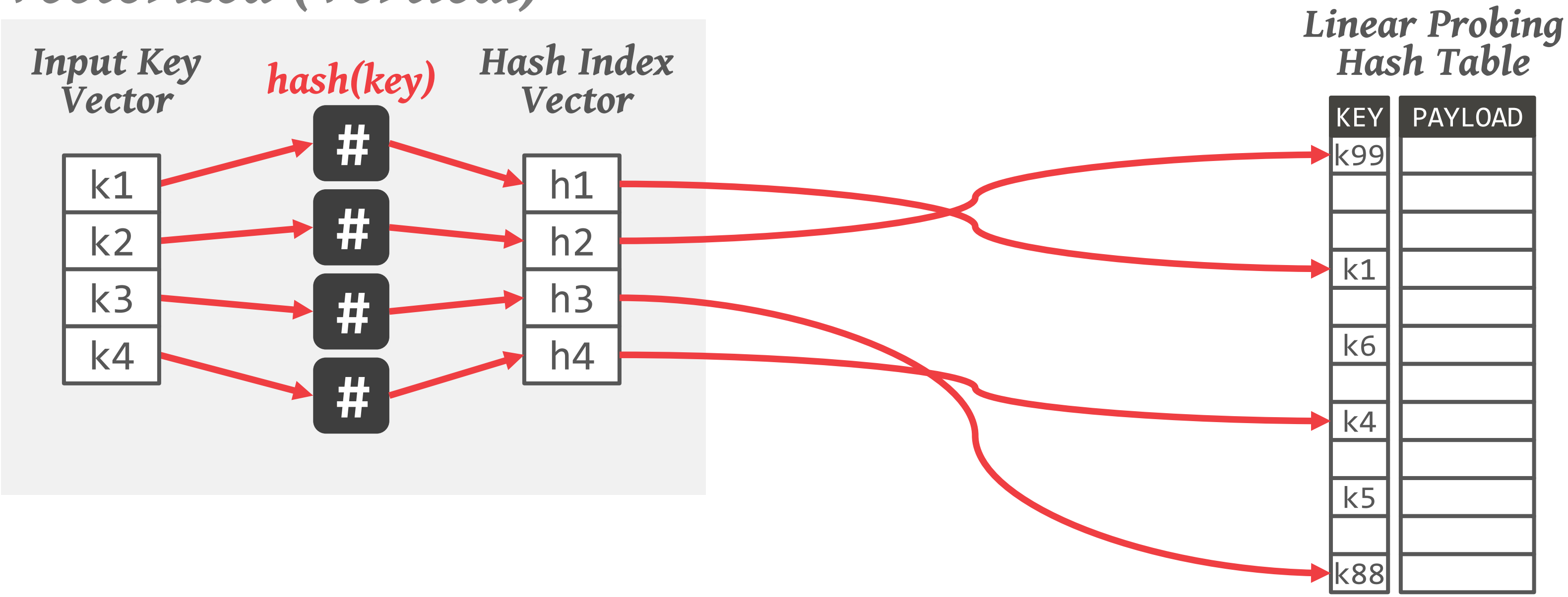
Vectorized (Vertical)





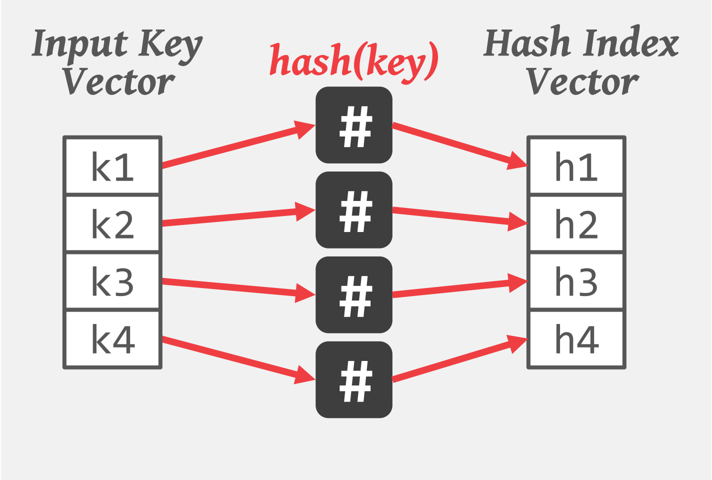
HASH TABLES – PROBING

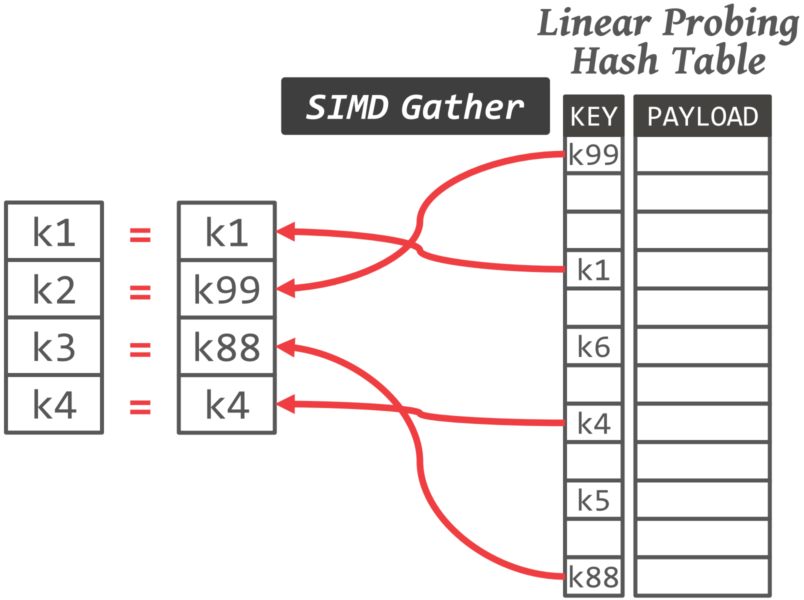
Vectorized (Vertical)



HASH TABLES – PROBING

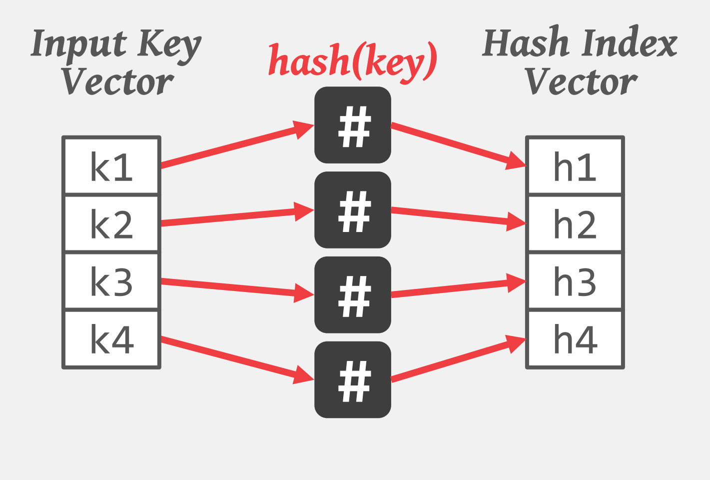
Vectorized (Vertical)

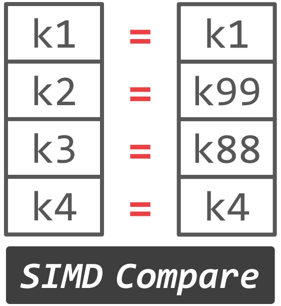




HASH TABLES – PROBING

Vectorized (Vertical)



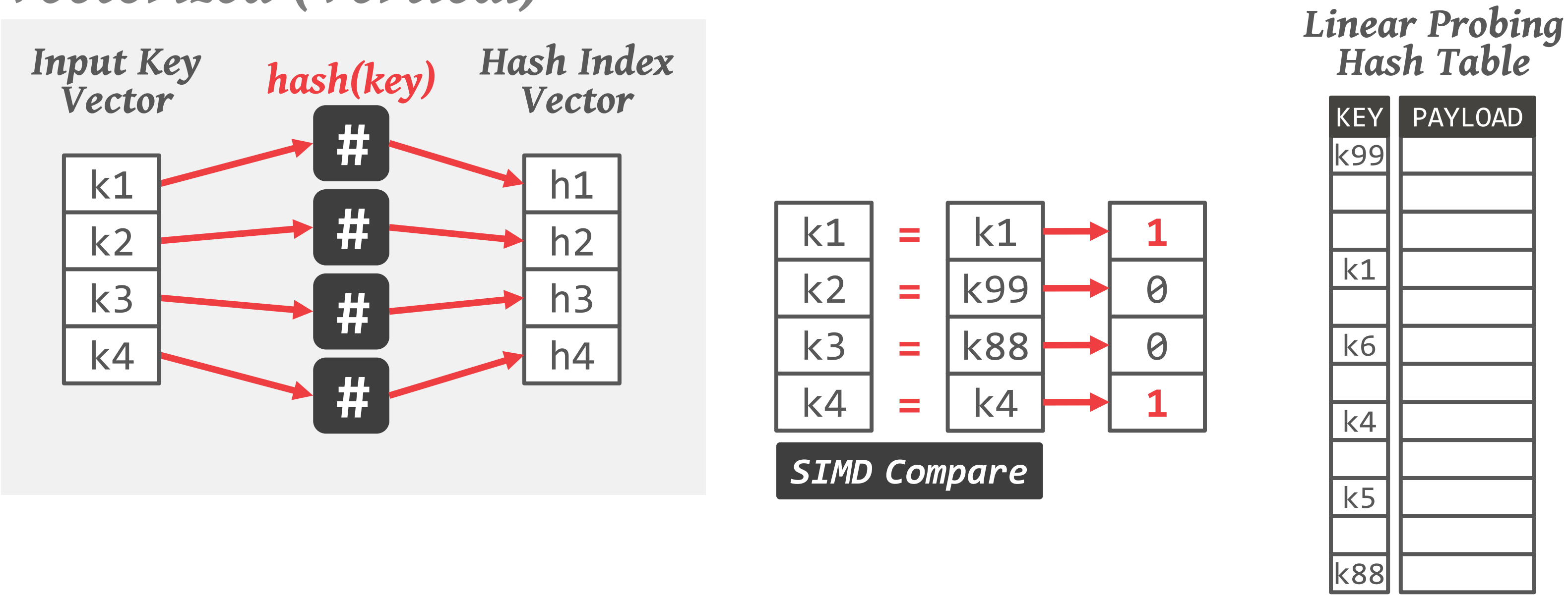


Linear Probing Hash Table

KEY	PAYLOAD
k99	
k1	
k6	
k4	
k5	
k88	

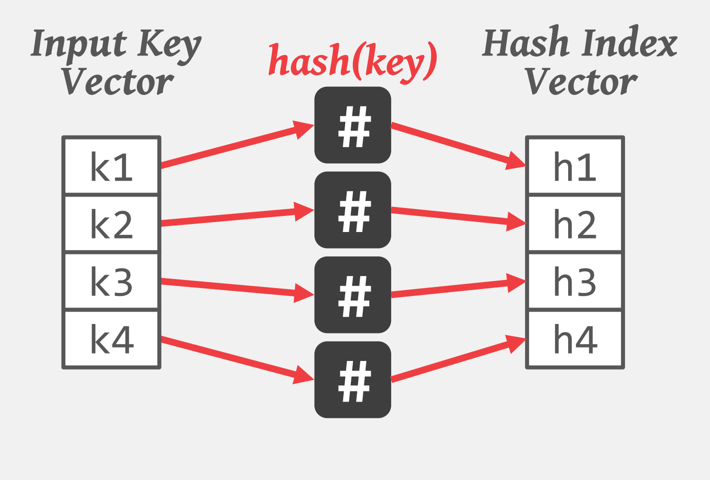
HASH TABLES – PROBING

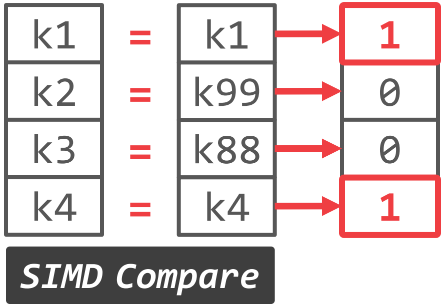
Vectorized (Vertical)

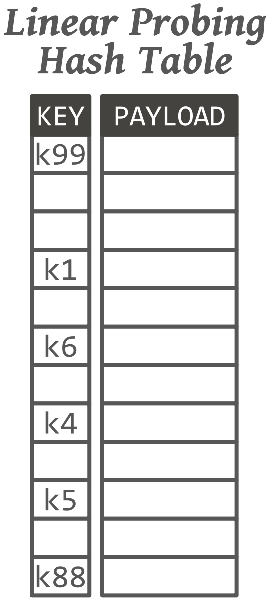


HASH TABLES – PROBING

Vectorized (Vertical)

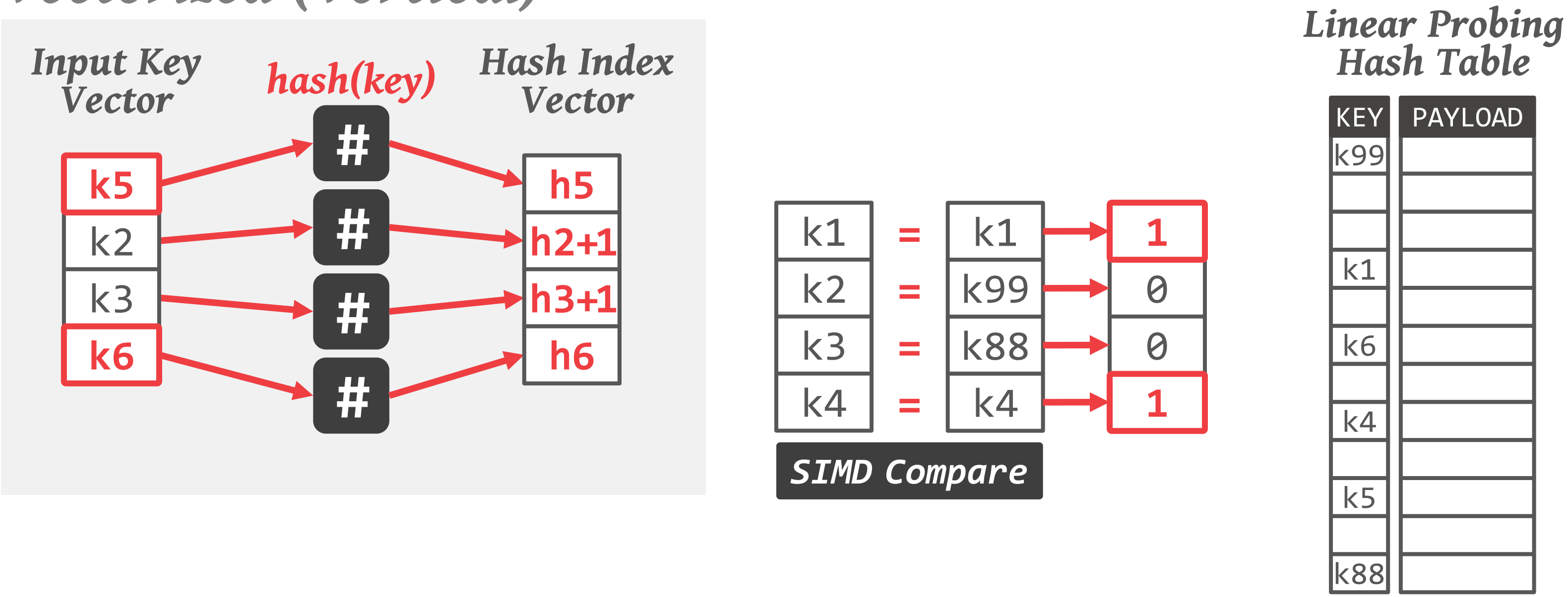






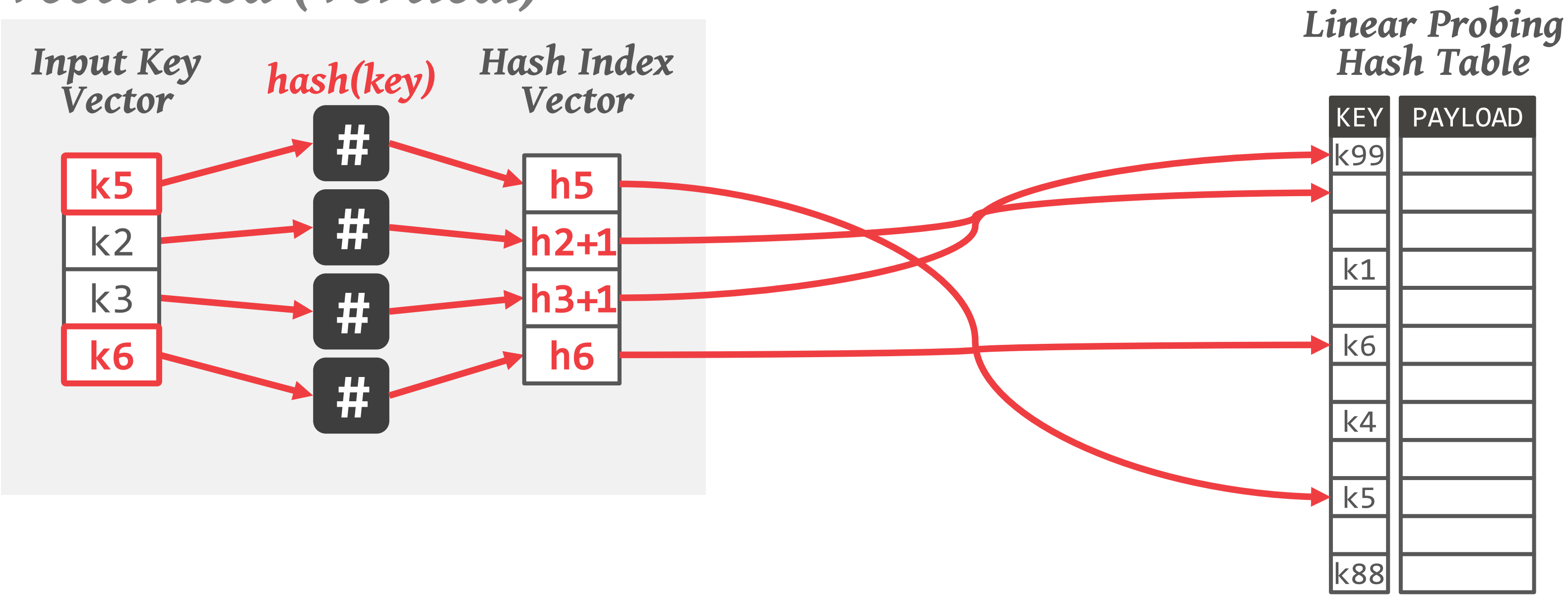
HASH TABLES – PROBING

Vectorized (Vertical)



HASH TABLES – PROBING

Vectorized (Vertical)



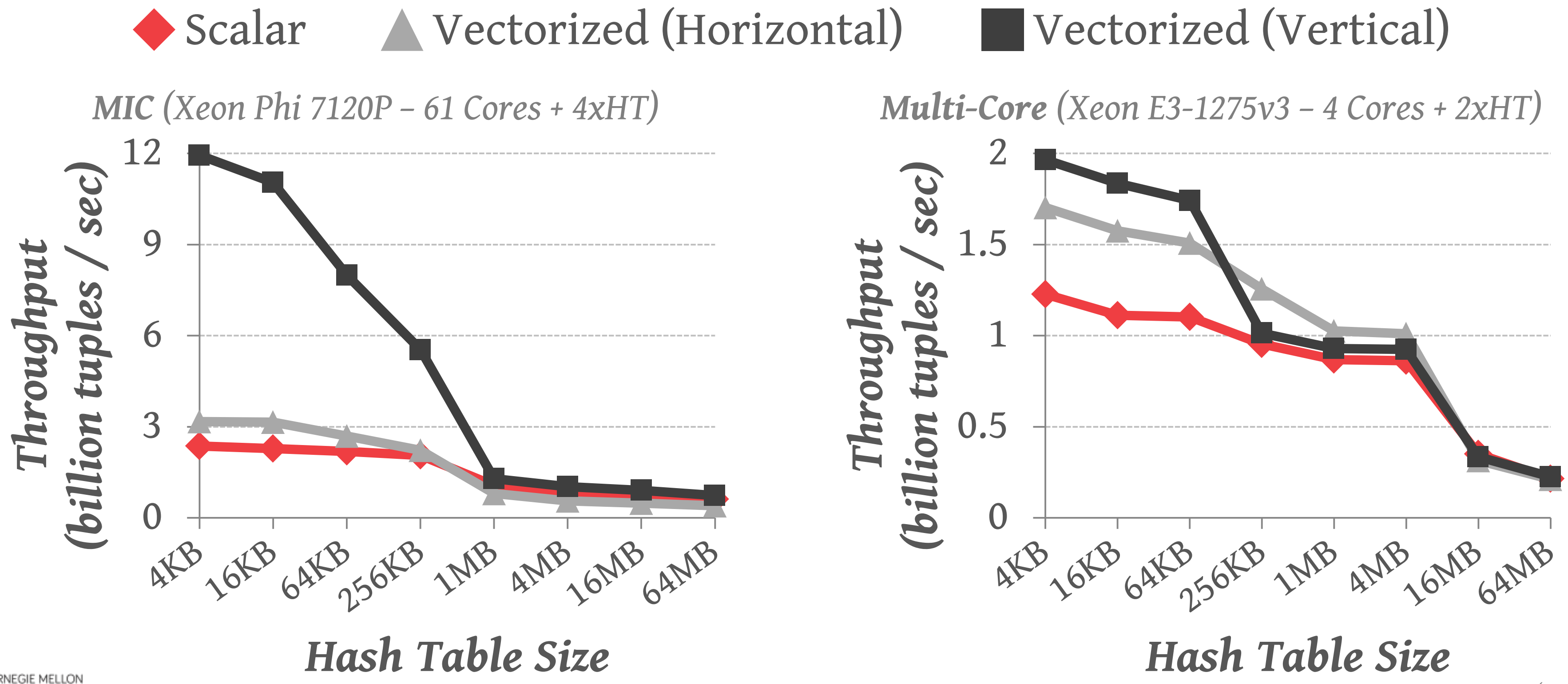
HASH TABLES – PROBING

◆ Scalar ▲ Vectorized (Horizontal) ■ Vectorized (Vertical)

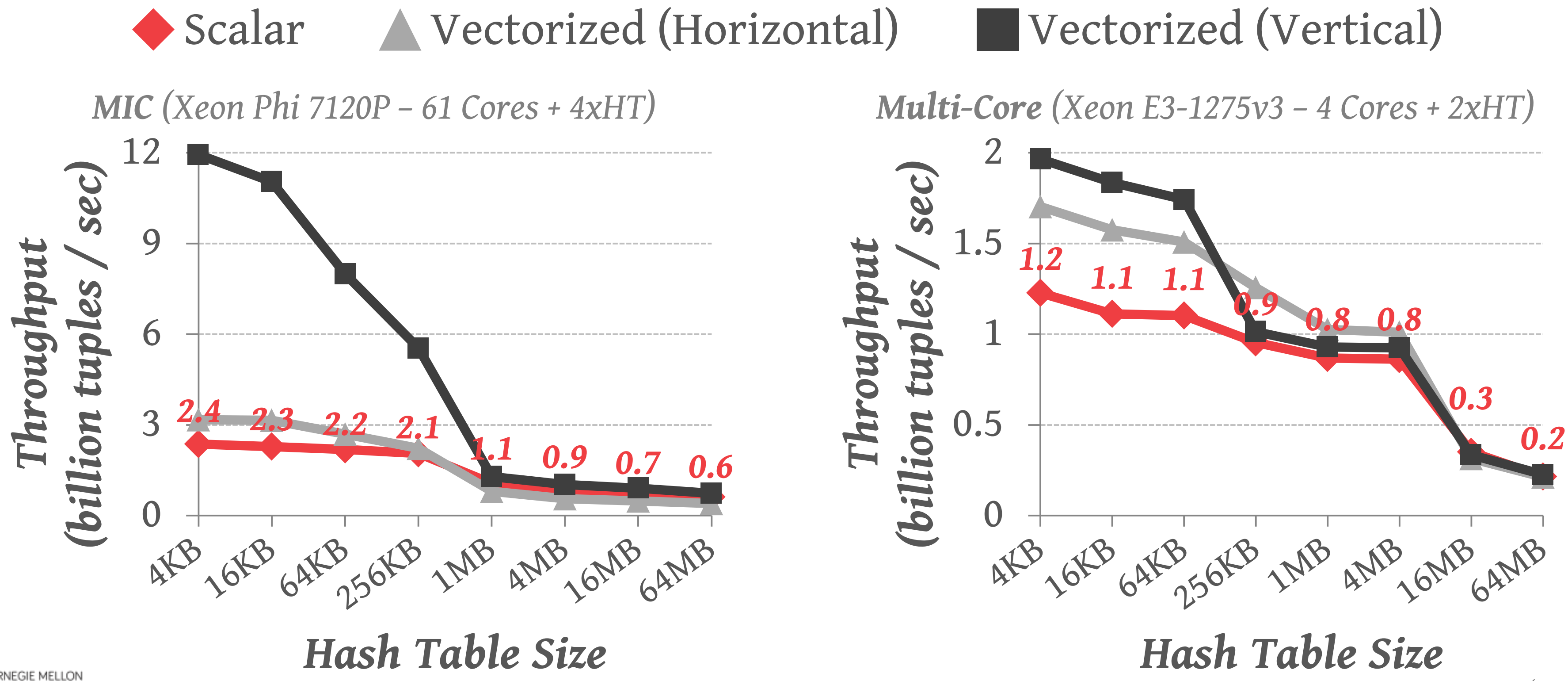
MIC *(Xeon Phi 7120P – 61 Cores + 4xHT)*

Multi-Core *(Xeon E3-1275v3 – 4 Cores + 2xHT)*

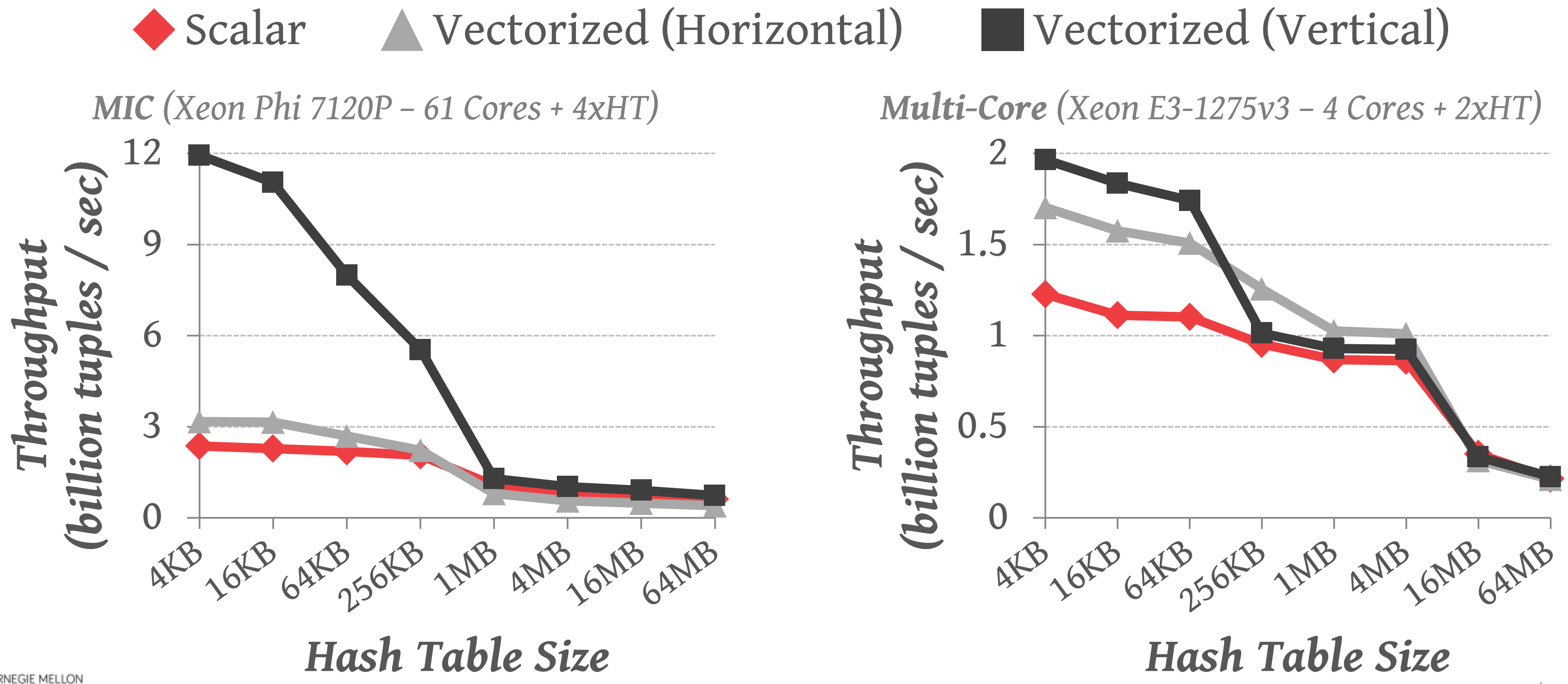
HASH TABLES – PROBING



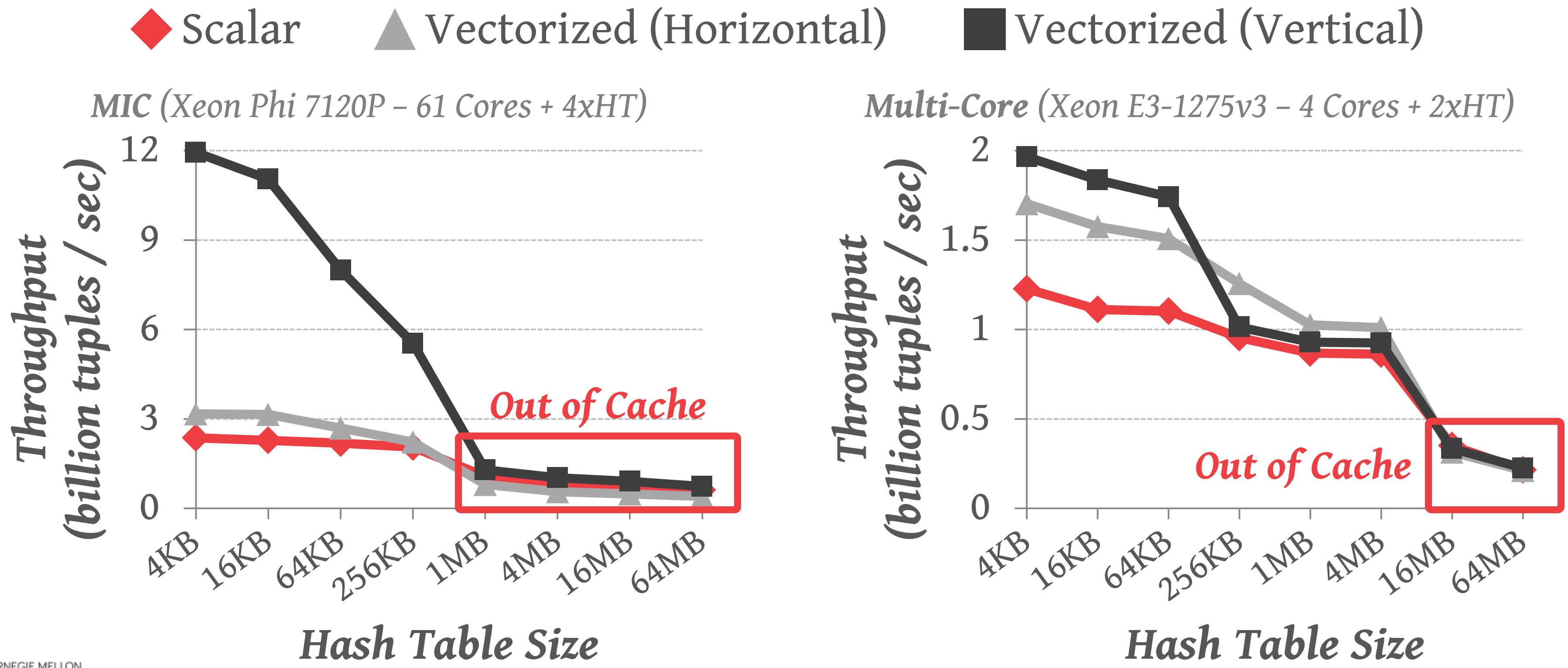
HASH TABLES – PROBING



HASH TABLES – PROBING



HASH TABLES – PROBING



PARTITIONING – HISTOGRAM

Use scatter and gathers to increment counts.

Replicate the histogram to handle collisions.

PARTITIONING – HISTOGRAM

Use scatter and gathers to increment counts.

Replicate the histogram to handle collisions.

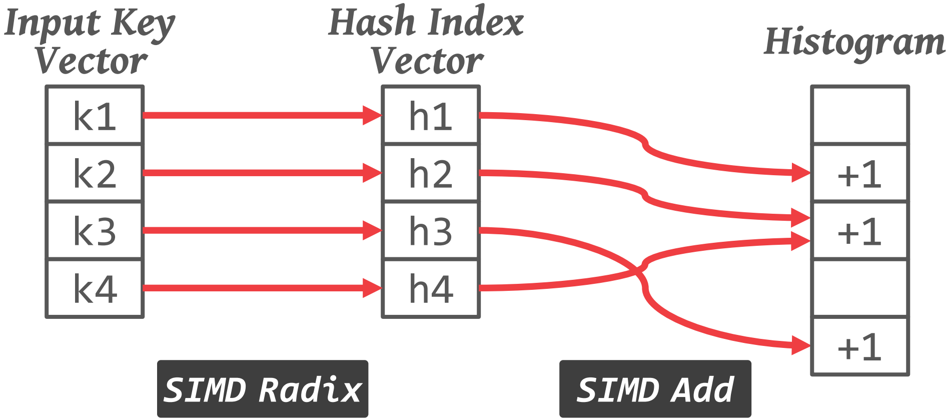
Input Key Vector

k1
k2
k3
k4

PARTITIONING – HISTOGRAM

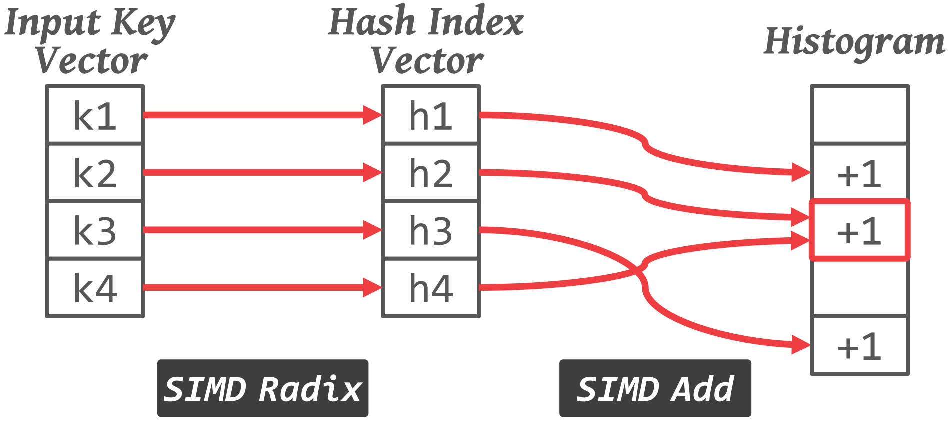
Use scatter and gathers to increment counts.

Replicate the histogram to handle collisions.



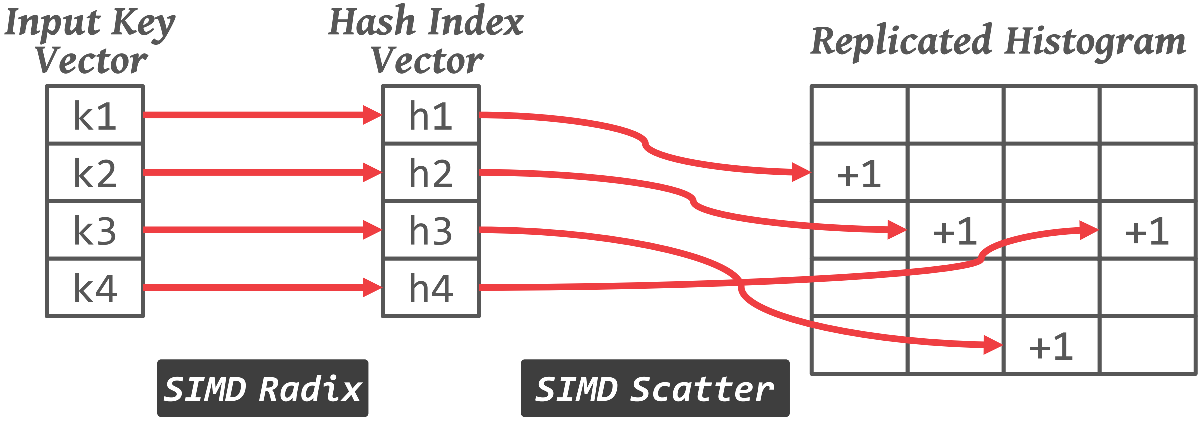
PARTITIONING – HISTOGRAM

Use scatter and gathers to increment counts.
Replicate the histogram to handle collisions.



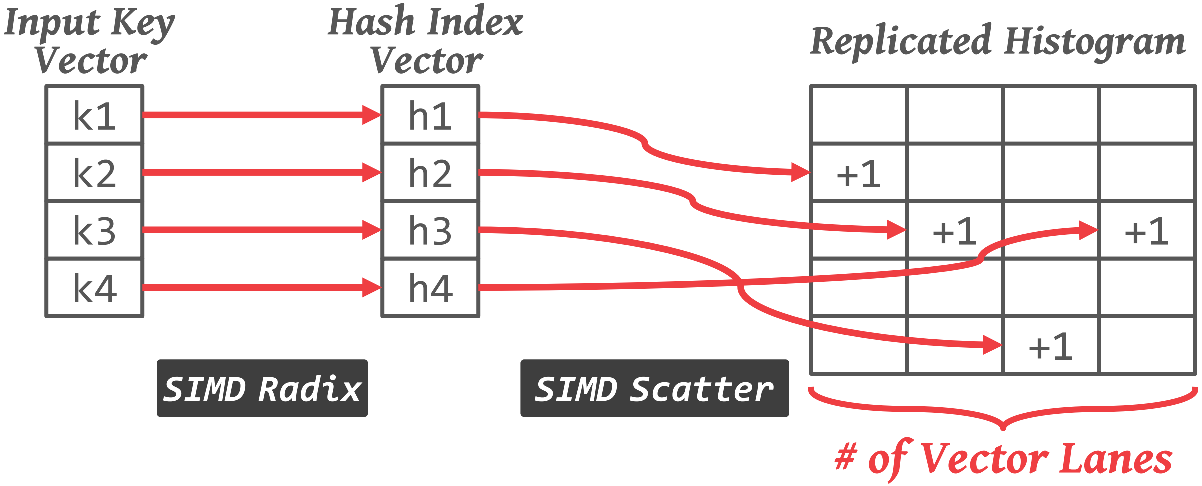
PARTITIONING – HISTOGRAM

Use scatter and gathers to increment counts.
Replicate the histogram to handle collisions.



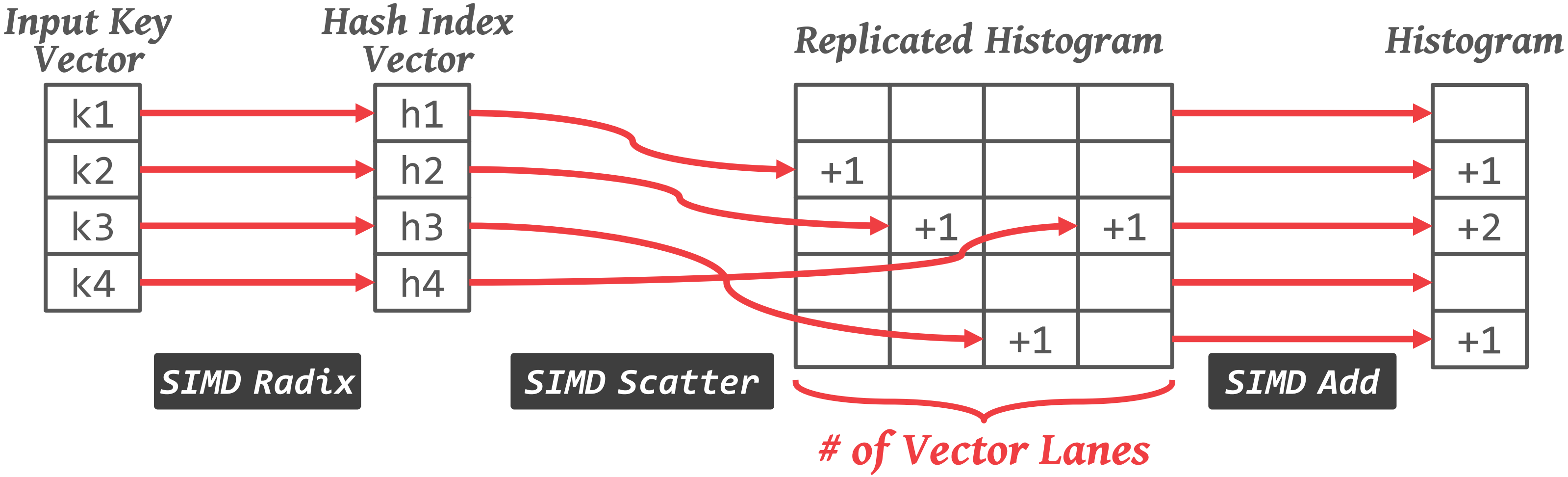
PARTITIONING – HISTOGRAM

Use scatter and gathers to increment counts.
Replicate the histogram to handle collisions.



PARTITIONING – HISTOGRAM

Use scatter and gathers to increment counts.
Replicate the histogram to handle collisions.



JOINS

No Partitioning

→ Build one shared hash table using atomics
→ Partially vectorized

Min Partitioning

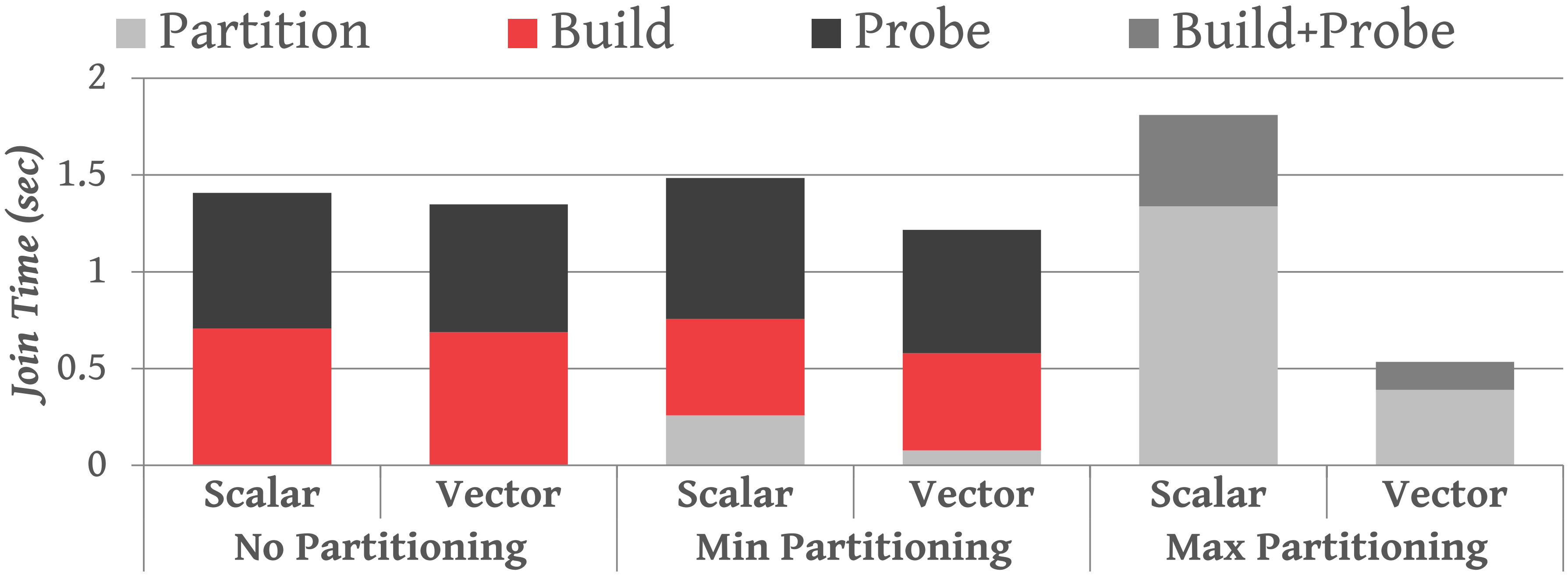
→ Partition building table
→ Build one hash table per thread
→ Fully vectorized

Max Partitioning

→ Partition both tables repeatedly
→ Build and probe cache-resident hash tables
→ Fully vectorized

JOINS

200M ⋈ 200M tuples (32-bit keys & payloads)
Xeon Phi 7120P – 61 Cores + 4xHT



BITWEAVING

Alternative storage layout for columnar databases that is designed for efficient predicate evaluation using SIMD.
→ Order-preserving dictionary encoding

Implemented in Wisconsin's QuickStep engine.

BITWEAVING: FAST SCANS FOR MAIN MEMORY DATA PROCESSING
SIGMOD 2013

BITWEAVING (VERTICAL)

t_0	0	0	1
t_1	1	0	1
t_2	1	1	0
t_3	0	0	1
t_4	1	1	0
t_5	1	0	0
t_6	0	0	0
t_7	1	1	1

t_8	1	0	0
t_9	0	1	1

BITWEAVING (VERTICAL)

Segment #1

t_0	0	0	1
t_1	1	0	1
t_2	1	1	0
t_3	0	0	1
t_4	1	1	0
t_5	1	0	0
t_6	0	0	0
t_7	1	1	1

Segment #2

t_8	1	0	0
t_9	0	1	1

BITWEAVING (VERTICAL)

Segment #1	t_0	0	0	1
	t_1	1	0	1
	t_2	1	1	0
	t_3	0	0	1
	t_4	1	1	0
	t_5	1	0	0
	t_6	0	0	0
	t_7	1	1	1
Segment #2	t_8	1	0	0
	t_9	0	1	1

Segment #1

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
v_0	0	1	1	0	1	1	0	1
v_1	0	0	1	0	1	0	0	1
v_2	1	1	0	1	0	0	0	1

Segment #2

	t_8	t_9	-	-	-	-	-	-
v_3	1	0	0	0	0	0	0	0
v_4	0	1	0	0	0	0	0	0
v_5	0	1	0	0	0	0	0	0

BITWEAVING (VERTICAL)

Segment #1

t_0	0	0	1
t_1	1	0	1
t_2	1	1	0
t_3	0	0	1
t_4	1	1	0
t_5	1	0	0
t_6	0	0	0
t_7	1	1	1

Segment #2

t_8	1	0	0
t_9	0	1	1

Segment #1

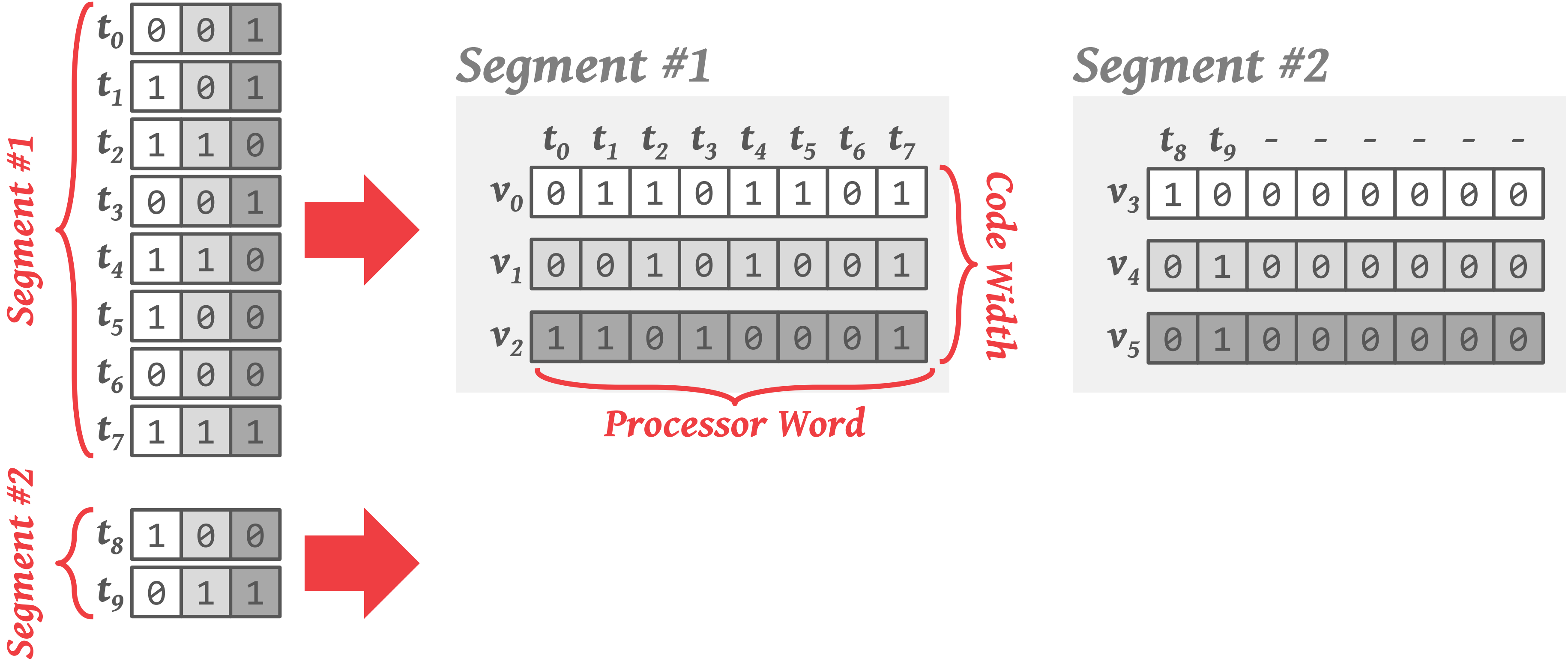
	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
v_0	0	1	1	0	1	1	0	1
v_1	0	0	1	0	1	0	0	1
v_2	1	1	0	1	0	0	0	1

Processor Word

Segment #2

	t_8	t_9	-	-	-	-	-	-
v_3	1	0	0	0	0	0	0	0
v_4	0	1	0	0	0	0	0	0
v_5	0	1	0	0	0	0	0	0

BITWEAVING (VERTICAL)



BITWEAVING (VERTICAL)

Segment #1

t_0	0	0	1
t_1	1	0	1
t_2	1	1	0
t_3	0	0	1
t_4	1	1	0
t_5	1	0	0
t_6	0	0	0
t_7	1	1	1

Segment #2

t_8	1	0	0
t_9	0	1	1

Segment #1

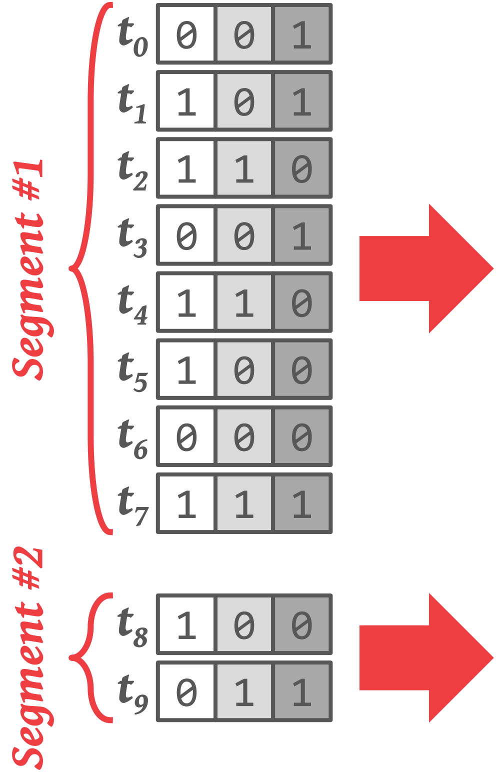
	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
v_0	0	1	1	0	1	1	0	1
v_1	0	0	1	0	1	0	0	1
v_2	1	1	0	1	0	0	0	1

Segment #2

	t_8	t_9	-	-	-	-	-	-
v_3	1	0	0	0	0	0	0	0
v_4	0	1	0	0	0	0	0	0
v_5	0	1	0	0	0	0	0	0

```
SELECT * FROM table
 WHERE key = "CMU"
```

BITWEAVING (VERTICAL)



Segment #1

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
v_0	0	1	1	0	1	1	0	1
v_1	0	0	1	0	1	0	0	1
v_2	1	1	0	1	0	0	0	1

Segment #2

	t_8	t_9	-	-	-	-	-	-
v_3	1	0	0	0	0	0	0	0
v_4	0	1	0	0	0	0	0	0
v_5	0	1	0	0	0	0	0	0

```
SELECT * FROM table
 WHERE key = "CMU"
```

0	1	0

BITWEAVING (VERTICAL)

Segment #1

t_0	0	0	1
t_1	1	0	1
t_2	1	1	0
t_3	0	0	1
t_4	1	1	0
t_5	1	0	0
t_6	0	0	0
t_7	1	1	1

Segment #2

t_8	1	0	0
t_9	0	1	1

Segment #1

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
v_0	0	1	1	0	1	1	0	1
v_1	0	0	1	0	1	0	0	1
v_2	1	1	0	1	0	0	0	1

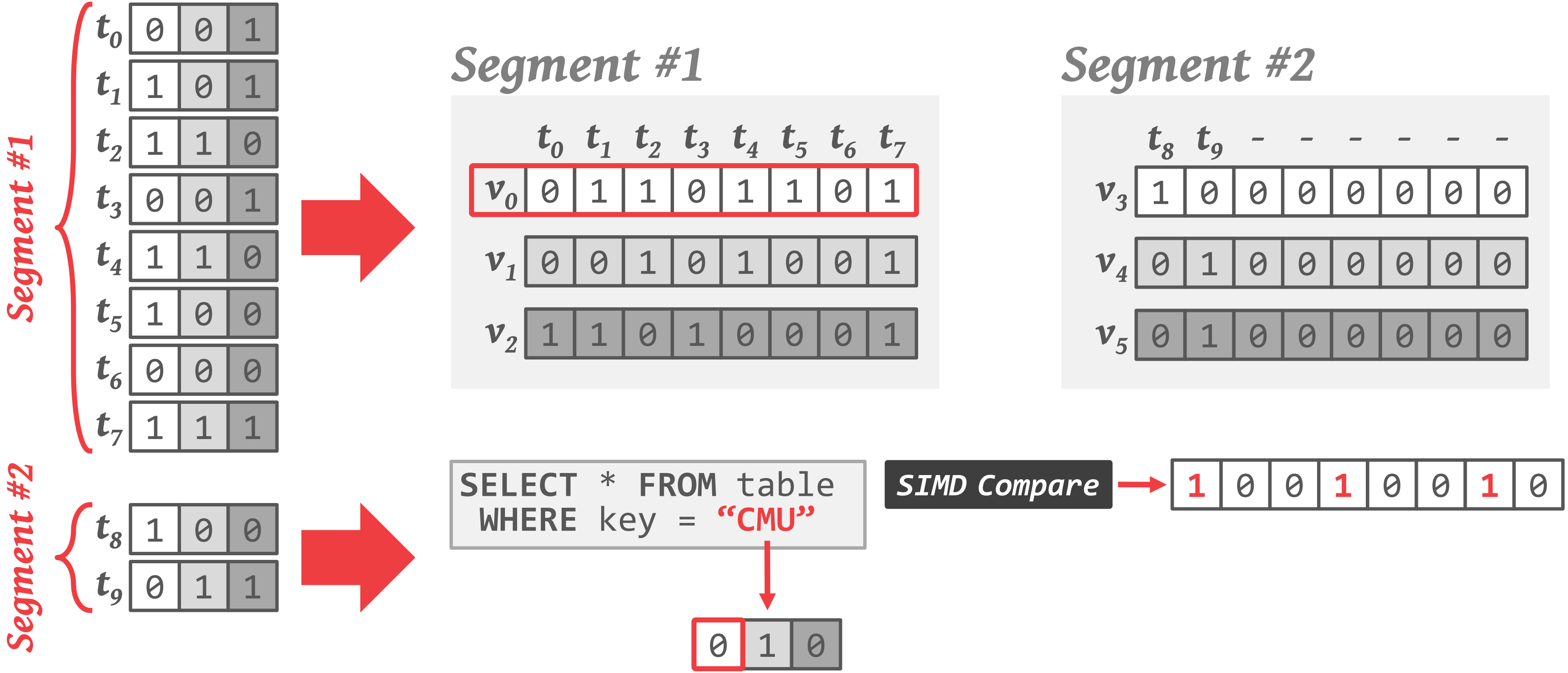
Segment #2

	t_8	t_9	-	-	-	-	-	-
v_3	1	0	0	0	0	0	0	0
v_4	0	1	0	0	0	0	0	0
v_5	0	1	0	0	0	0	0	0

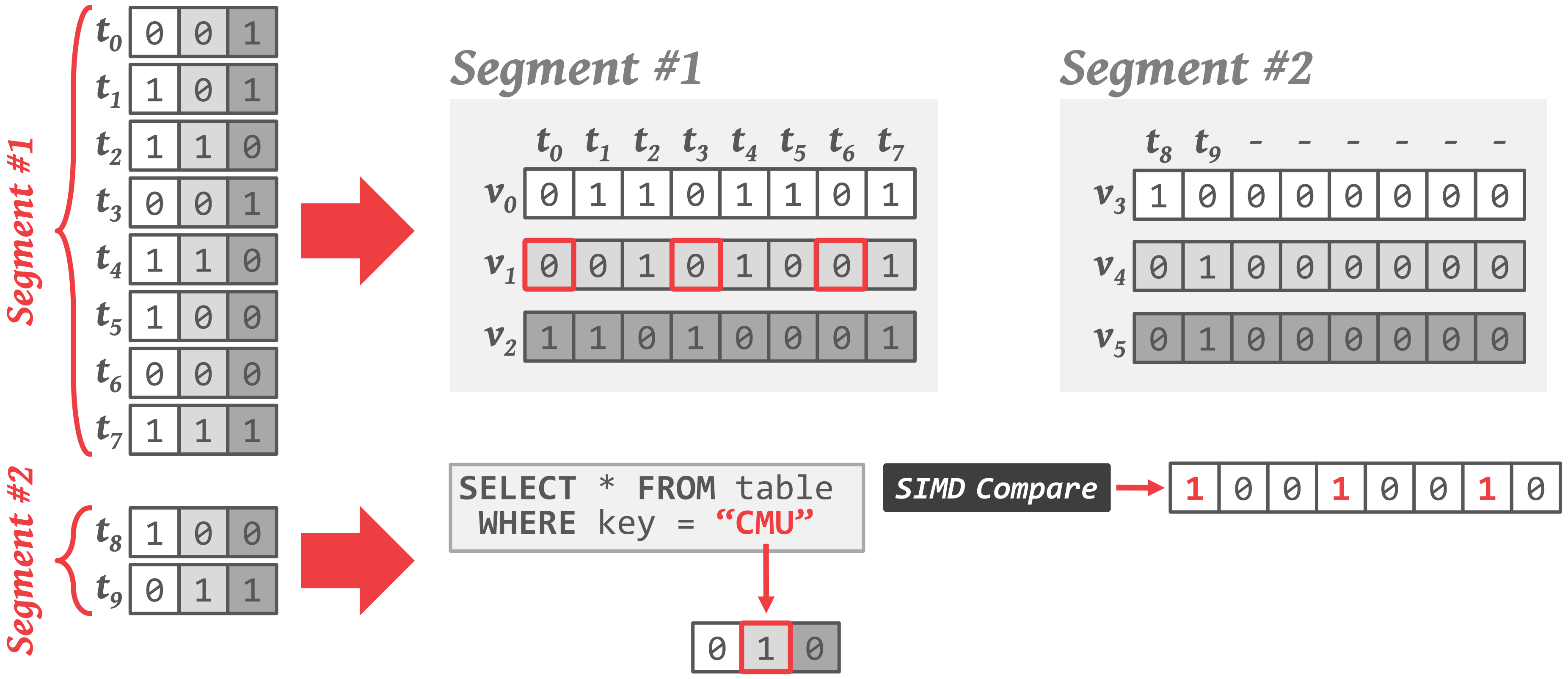
```
SELECT * FROM table
 WHERE key = "CMU"
```

0	1	0

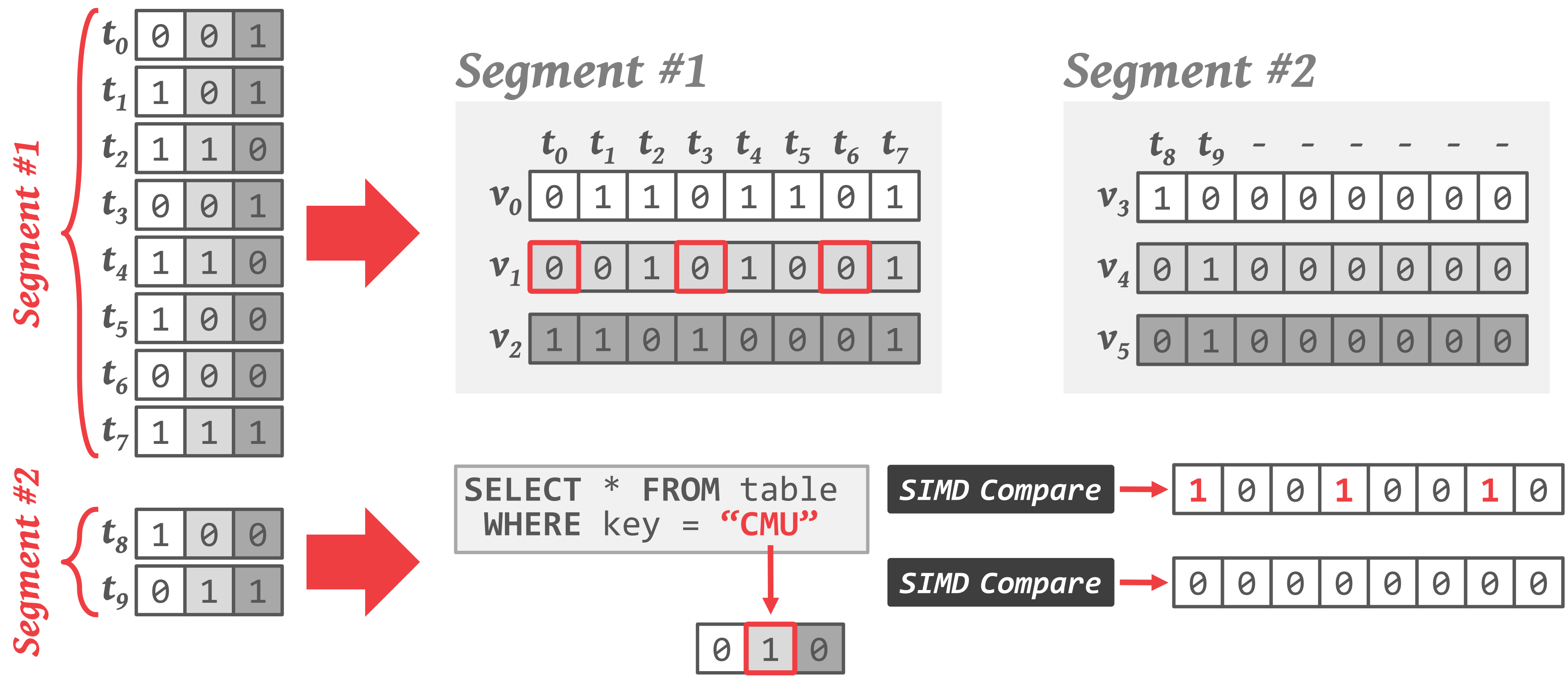
BITWEAVING (VERTICAL)



BITWEAVING (VERTICAL)



BITWEAVING (VERTICAL)



PARTING THOUGHTS

Vectorization is essential for OLAP queries.

We can combine all the intra-query parallelism optimizations we've talked about in a DBMS.

→ Multiple threads processing the same query.
→ Each thread can execute a compiled plan.
→ The compiled plan can invoke vectorized operations.

UPCOMING DATABASE TECH TALKS

Apache Samza @ LinkedIn (Yi Pan)

→ April 14th @ 12:00pm
→ CIC - 4th floor (ISTC Panther Hollow Room)
→ http://db.cs.cmu.edu/events/yi-pan-apache-samza-linkedin/

SpliceMachine (Monte Zweben)

→ April 15th @ 12:00pm
→ GHC 6115
→ http://db.cs.cmu.edu/events/monte-zweben-splice-machine/

NEXT CLASS

Project #3 Status Updates

Each group gets **five** minutes.

Send me a PDF of your PowerPoint slides immediately afterwards.