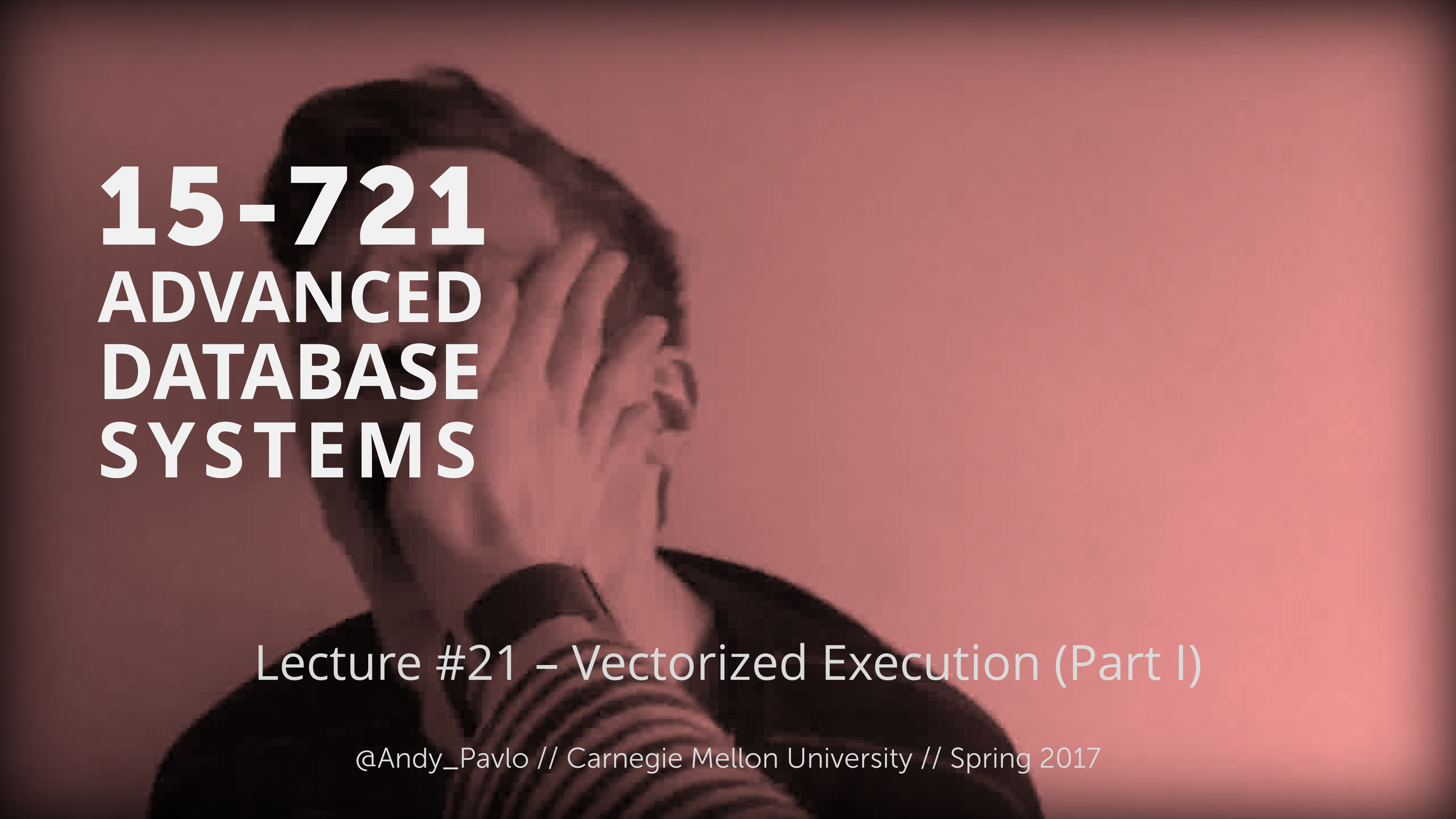


15-721 ADVANCED DATABASE SYSTEMS

Lecture #21 – Vectorized Execution (Part I)

@Andy_Pavlo // Carnegie Mellon University // Spring 2017

TODAY'S AGENDA

Background
Hardware
Vectorized Algorithms (Columbia)

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

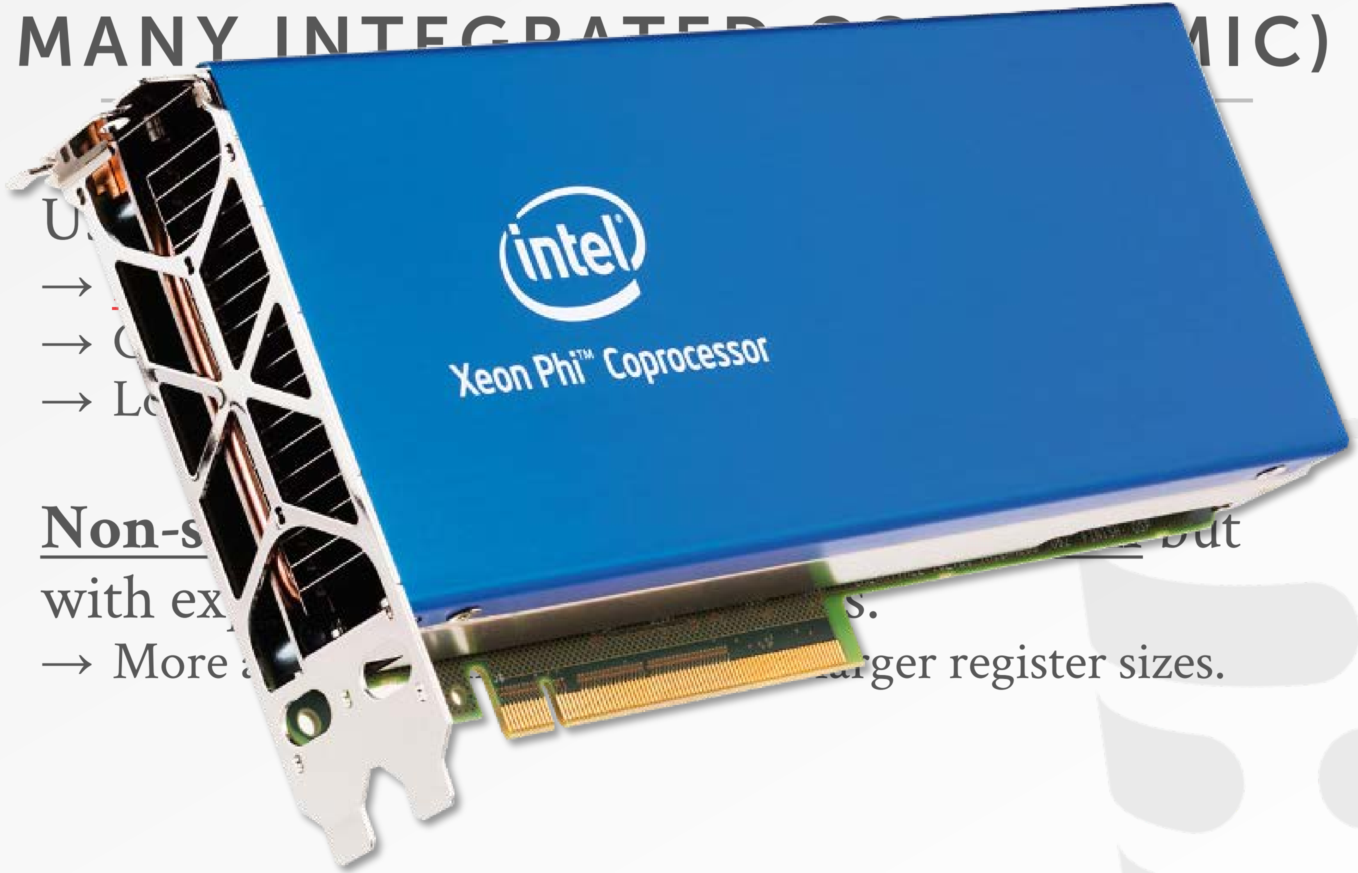
→ Cores = Intel P54C (aka Pentium from the 1990s).

→ 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

→ Le

Non-s but with ex s.

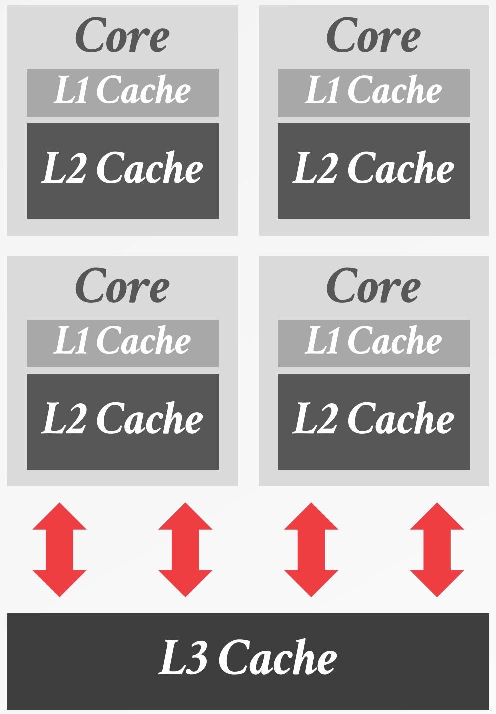
→ More larger register sizes.

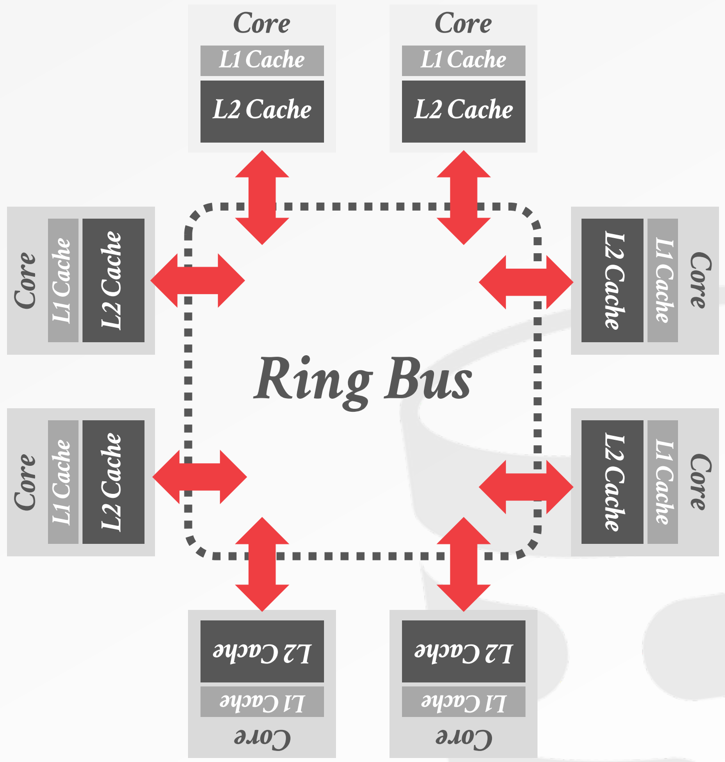
MANY INTEGRATED CORES (MIC)

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.

MULTI-CORE VS. MIC





Source: Orestis Polychroniou

WHY THIS MATTERS

Say we can parallelize our algorithm over 32 cores.
Each core has a 4-wide SIMD registers.

Potential Speed-up: 32x × 4x = 128x

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.

SINGLE INSTRUCTION, MULTIPLE DATA

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

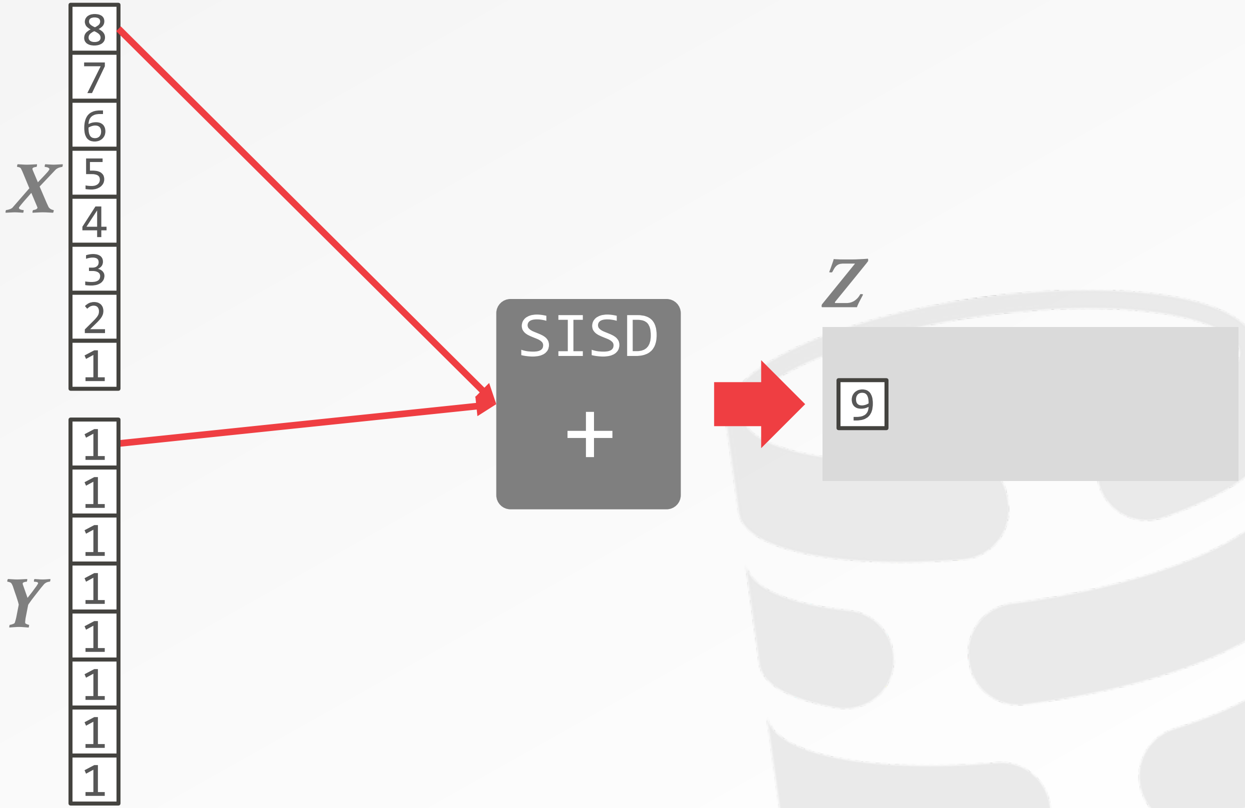
All major ISAs have microarchitecture support SIMD operations.
→ **x86**: MMX, SSE, SSE2, SSE3, SSE4, AVX
→ **PowerPC**: Altivec
→ **ARM**: NEON

SIMD EXAMPLE

$X + Y = Z$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

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

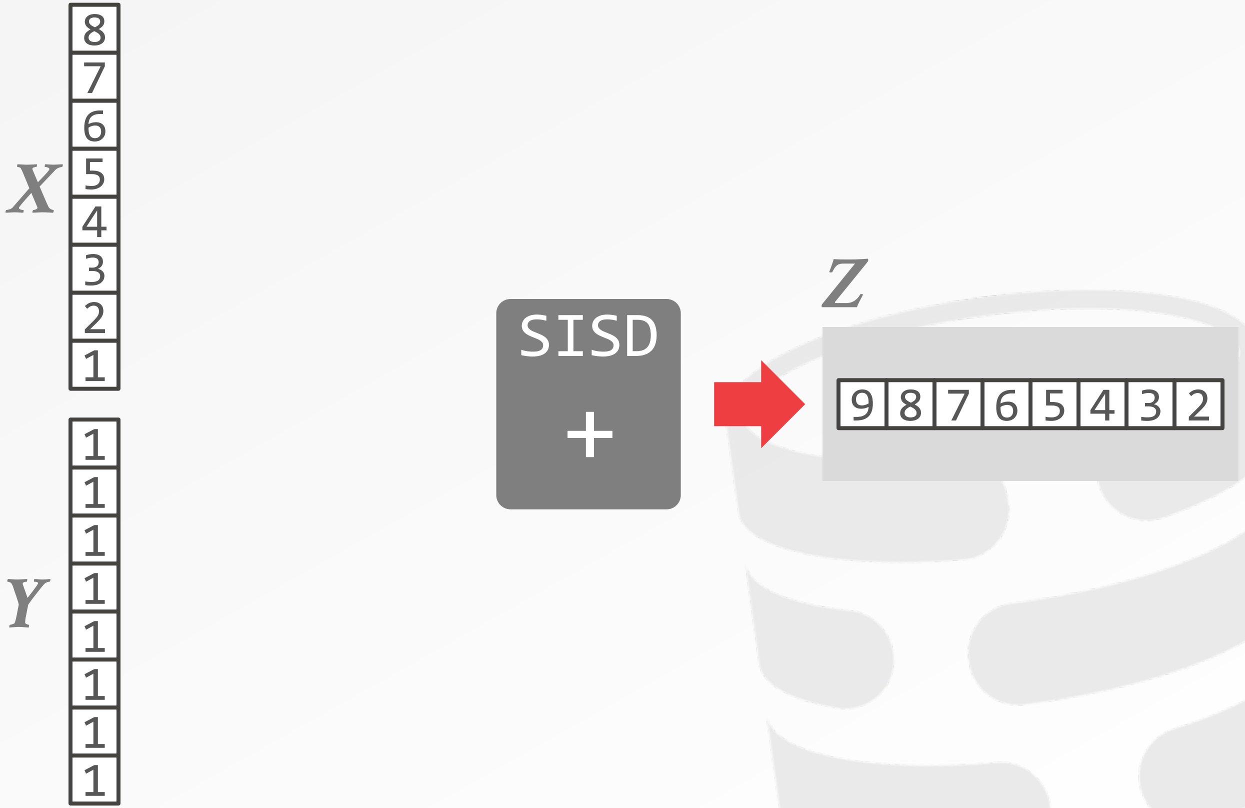


SIMD EXAMPLE

$X + Y = Z$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

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

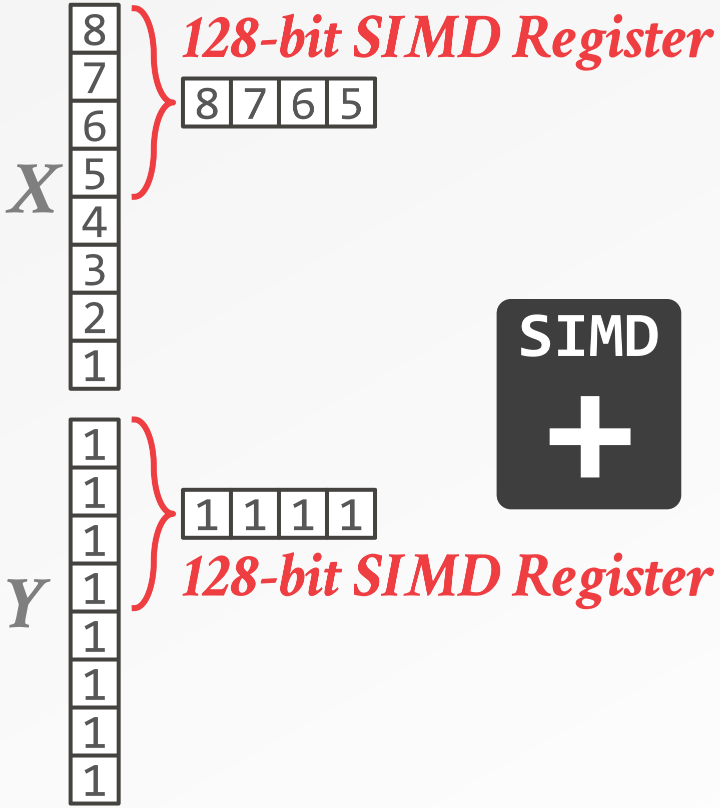


SIMD EXAMPLE

$X + Y = Z$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

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

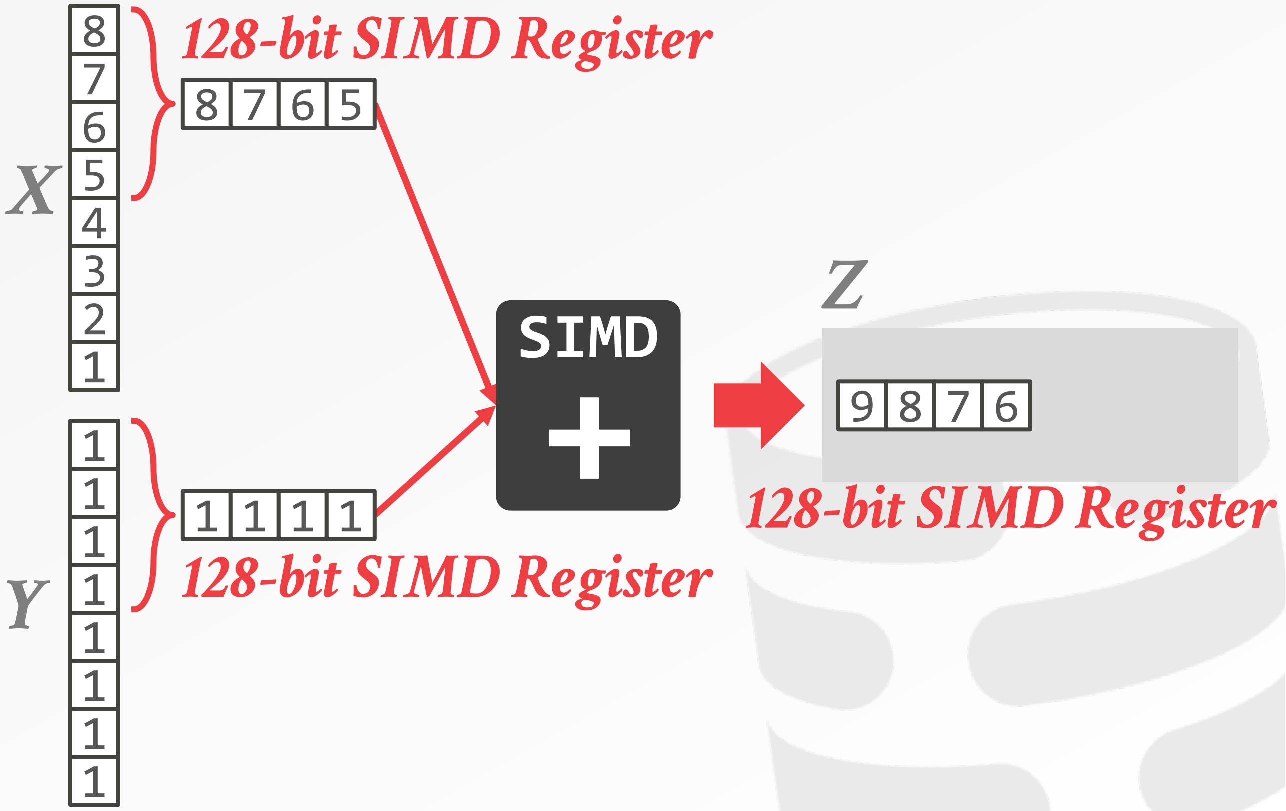


SIMD EXAMPLE

$X + Y = Z$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

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

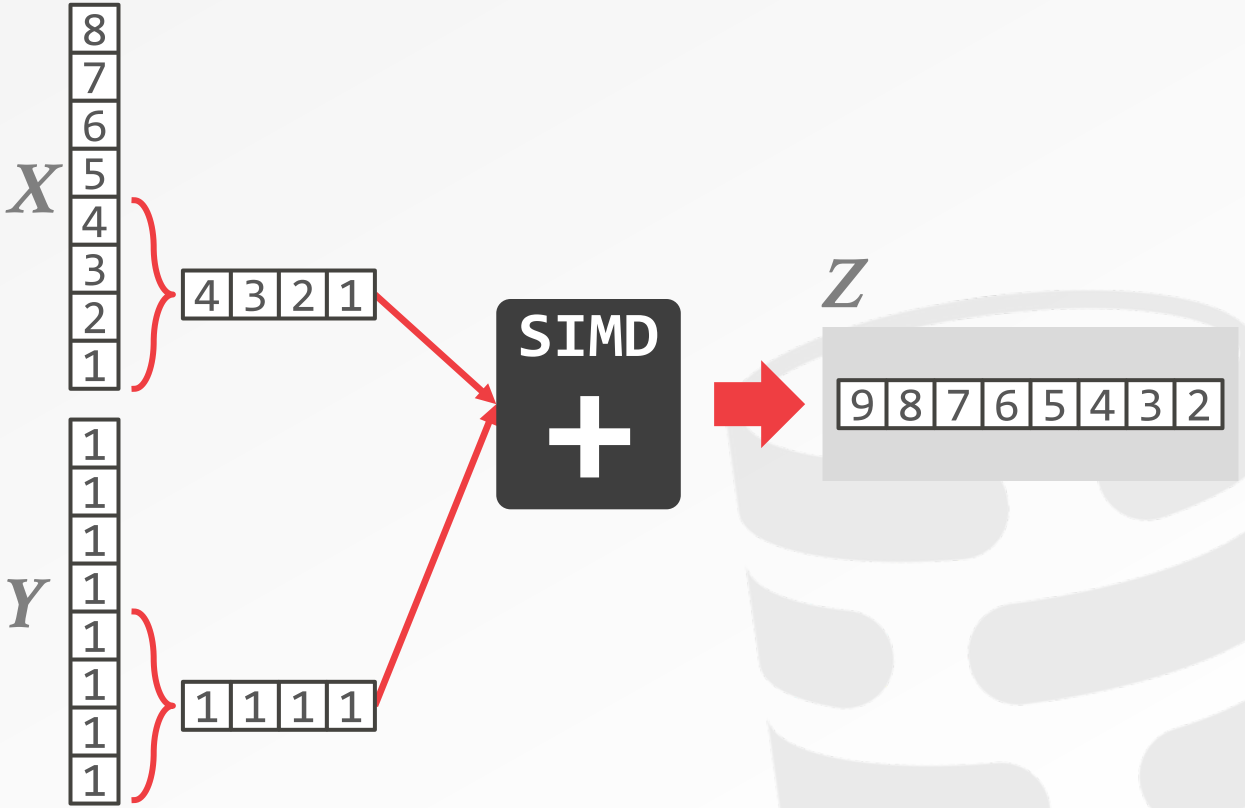


SIMD EXAMPLE

$X + Y = Z$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

```
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).

INTEL SIMD EXTENSIONS

		Width	Integers	Single-P	Double-P
1997	MMX	64 bits	✔		
1999	SSE	128 bits	✔	✔(×4)	
2001	SSE2	128 bits	✔	✔	✔(×2)
2004	SSE3	128 bits	✔	✔	✔
2006	SSSE 3	128 bits	✔	✔	✔
2006	SSE 4.1	128 bits	✔	✔	✔
2008	SSE 4.2	128 bits	✔	✔	✔
2011	AVX	256 bits	✔	✔(×8)	✔(×4)
2013	AVX2	256 bits	✔	✔	✔
2017?	**AVX-512**	**512 bits**	**✔**	**✔(×16)**	**✔(×8)**

Source: James Reinders

VECTORIZATION

Choice #1: Automatic Vectorization

Choice #2: Compiler Hints

Choice #3: Explicit Vectorization

Source: James Reinders

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.

AUTOMATIC VECTORIZATION

```
void add(int *X,
         int *Y,
         int *Z) {
  for (int i=0; i<MAX; i++) {
    Z[i] = X[i] + Y[i];
  }
}
```

These might point to the same address!

This loop is not legal to automatically vectorize.

The code is written such that the addition is described as being done sequentially.

COMPILER HINTS

Provide the compiler with additional information about the code to let it know that is safe to vectorize.

Two approaches:
→ Give explicit information about memory locations.
→ Tell the compiler to ignore vector dependencies.

COMPILER HINTS

```
void add(int *restrict X,
         int *restrict Y,
         int *restrict Z) {
  for (int i=0; i<MAX; i++) {
    Z[i] = X[i] + Y[i];
  }
}
```

The **restrict** keyword in C++ tells the compiler that the arrays are distinct locations in memory.

COMPILER HINTS

```
void add(int *X,
         int *Y,
         int *Z) {
#pragma ivdep
  for (int i=0; i<MAX; i++) {
    Z[i] = X[i] + Y[i];
  }
}
```

This pragma tells the compiler to ignore loop dependencies for the vectors.

It's up to you make sure that this is correct.

EXPLICIT VECTORIZATION

Use CPU intrinsics to manually marshal data between SIMD registers and execute vectorized instructions.

Potentially not portable.

EXPLICIT VECTORIZATION

```
void add(int *X,
         int *Y,
         int *Z) {
  __mm128 *vecX = (__m128*)X;
  __mm128 *vecY = (__m128*)Y;
  __mm128 *vecZ = (__m128*)Z;
  for (int i=0; i<MAX/4; i++) {
    *vecZ++ = _mm_add_epi32(
                 *vecX++, vecY++);
  }
}
```

Store the vectors in 128-bit SIMD registers.

Then invoke the intrinsic to add together the vectors and write them to the output location.

EXPLICIT VECTORIZATION

Linear Access Operators
→ Predicate evaluation
→ Compression

Ad-hoc Vectorization
→ Sorting
→ Merging

Composable Operations
→ Multi-way trees
→ Bucketized hash tables

Source: Orestis Polychroniou

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

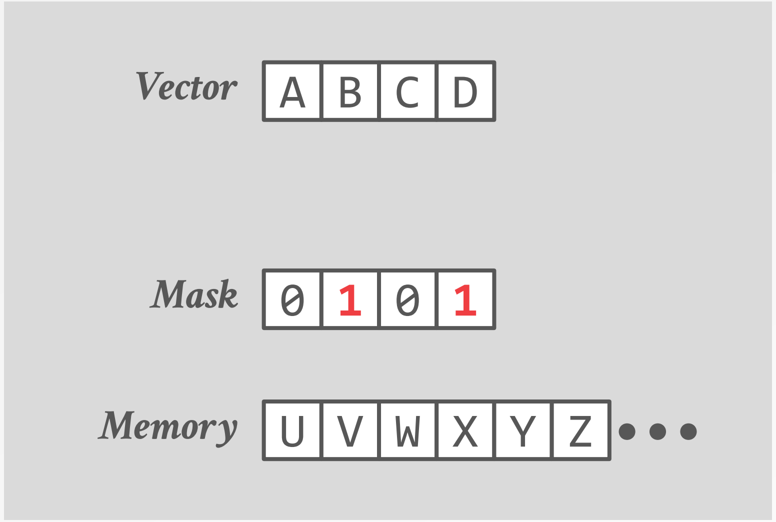
Selective Sore

Selective Gather

Selective Scatter

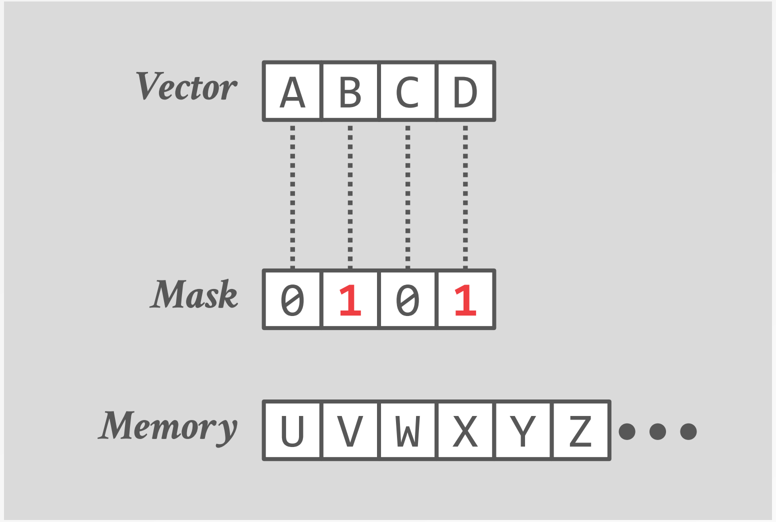
FUNDAMENTAL VECTOR OPERATIONS

Selective Load



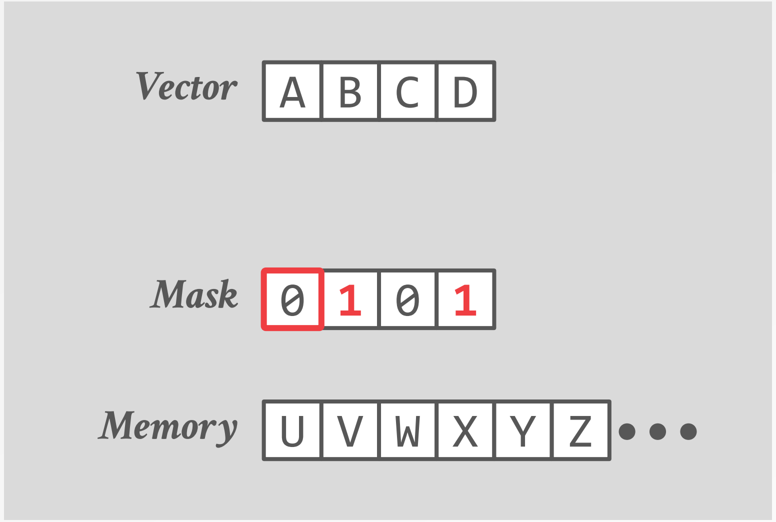
FUNDAMENTAL VECTOR OPERATIONS

Selective Load



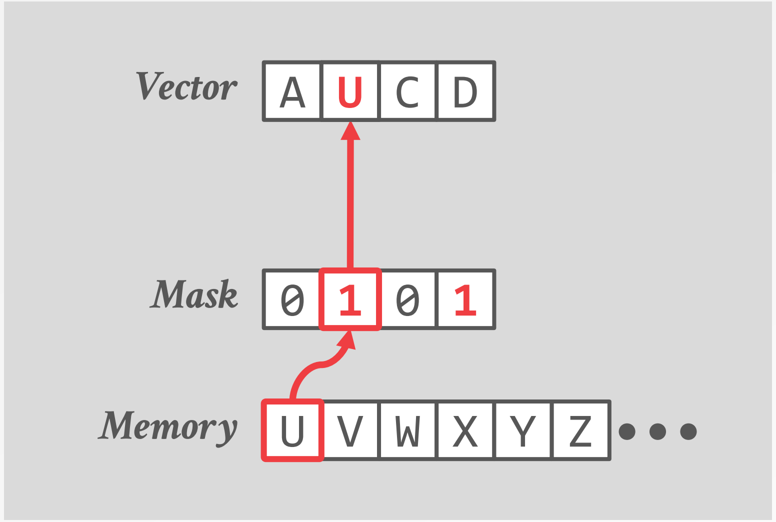
FUNDAMENTAL VECTOR OPERATIONS

Selective Load



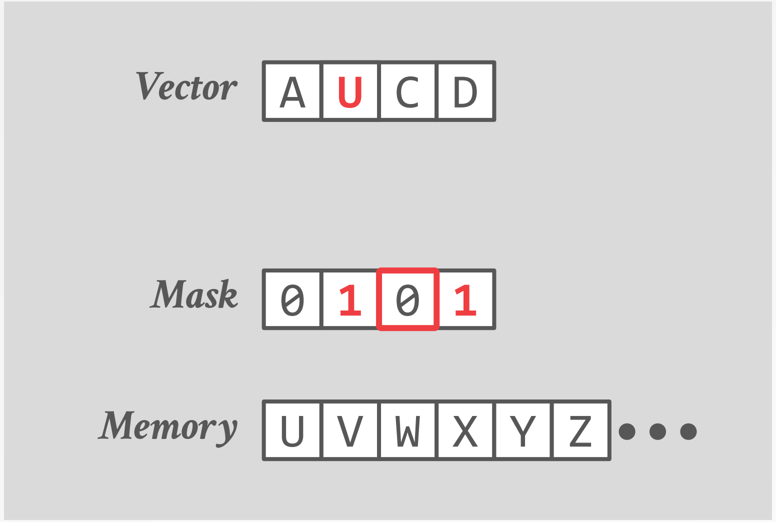
FUNDAMENTAL VECTOR OPERATIONS

Selective Load



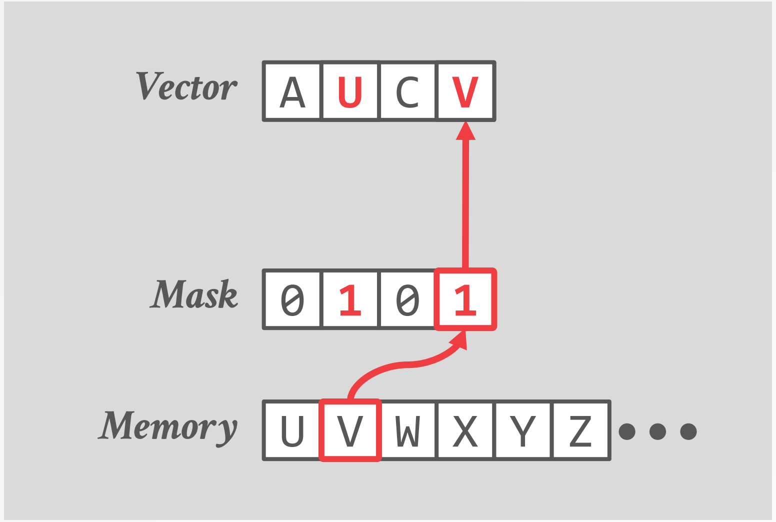
FUNDAMENTAL VECTOR OPERATIONS

Selective Load



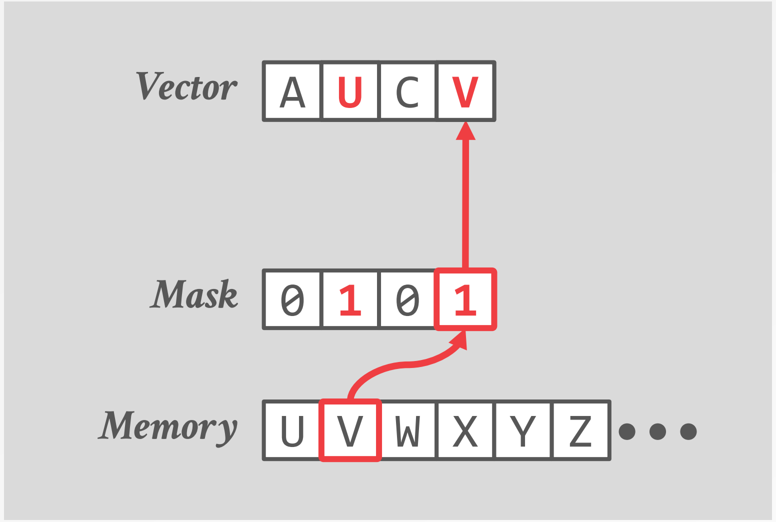
FUNDAMENTAL VECTOR OPERATIONS

Selective Load

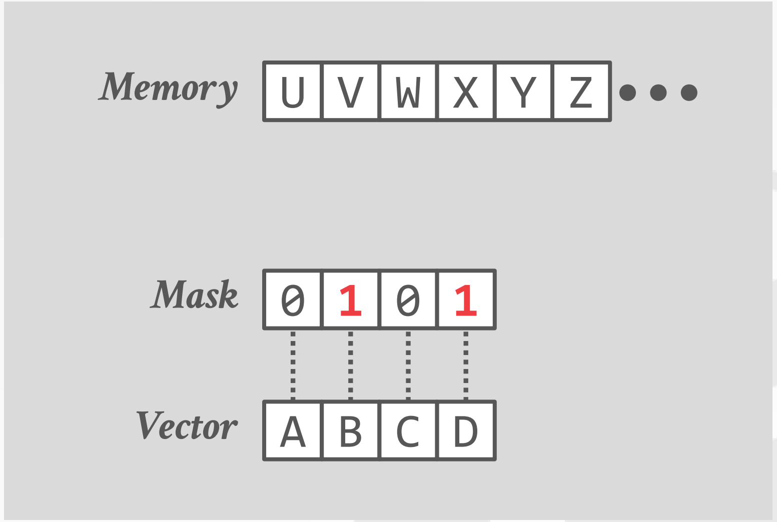


FUNDAMENTAL VECTOR OPERATIONS

Selective Load



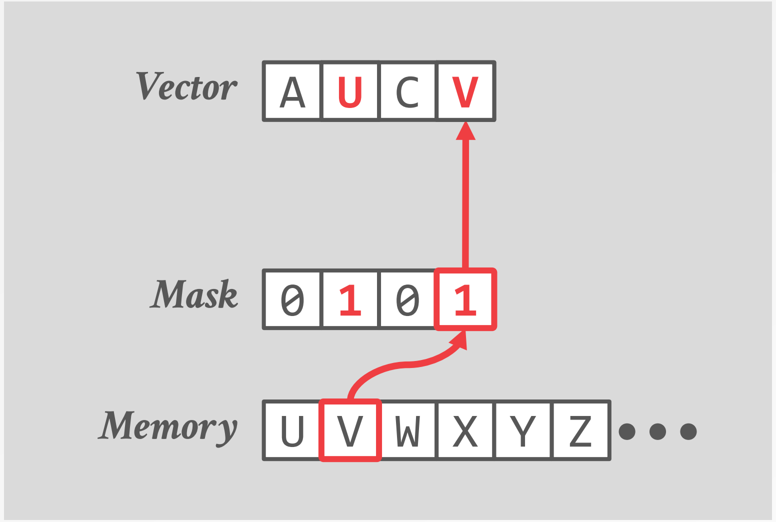
Selective Store



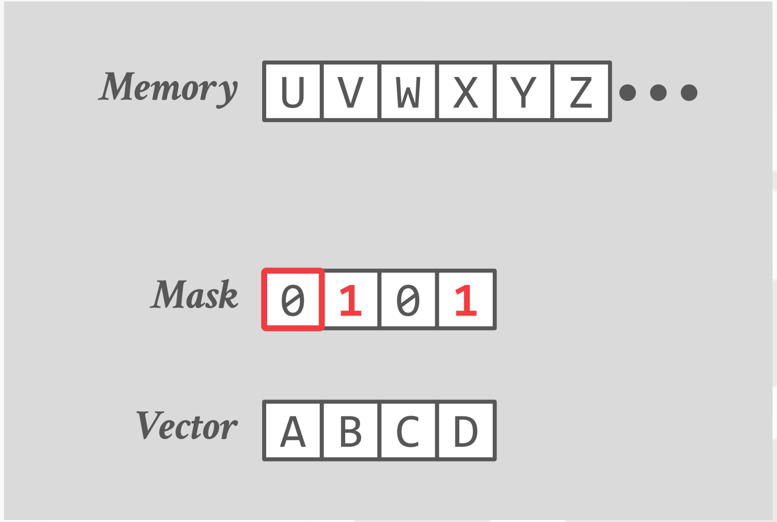
CARNEGIE MELLON
DATABASE GROUP

FUNDAMENTAL VECTOR OPERATIONS

Selective Load

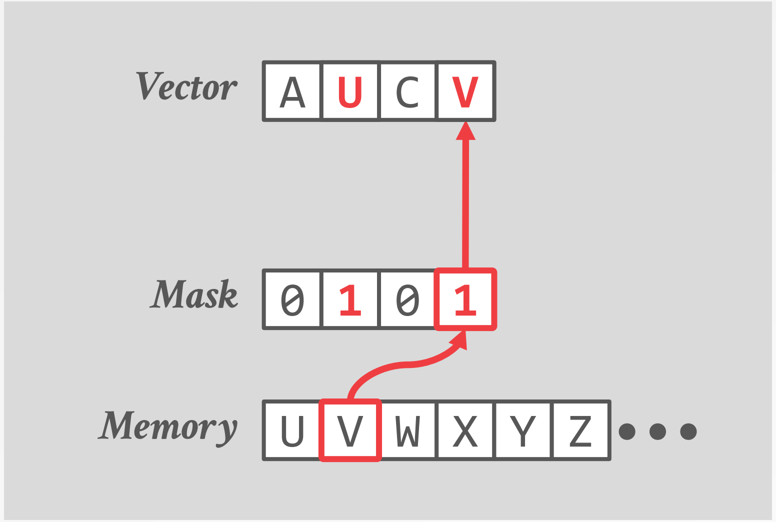


Selective Store

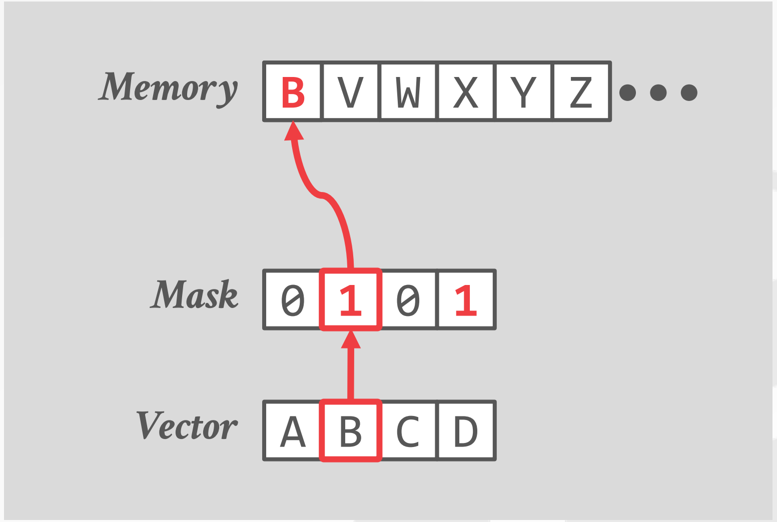


FUNDAMENTAL VECTOR OPERATIONS

Selective Load

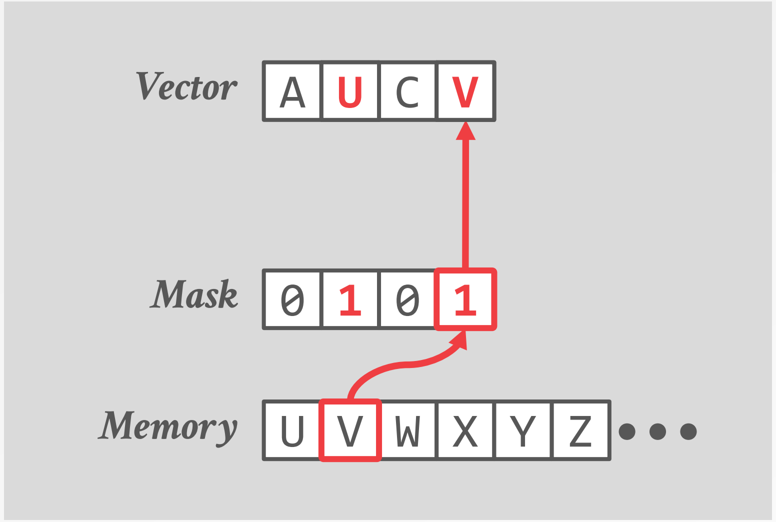


Selective Store

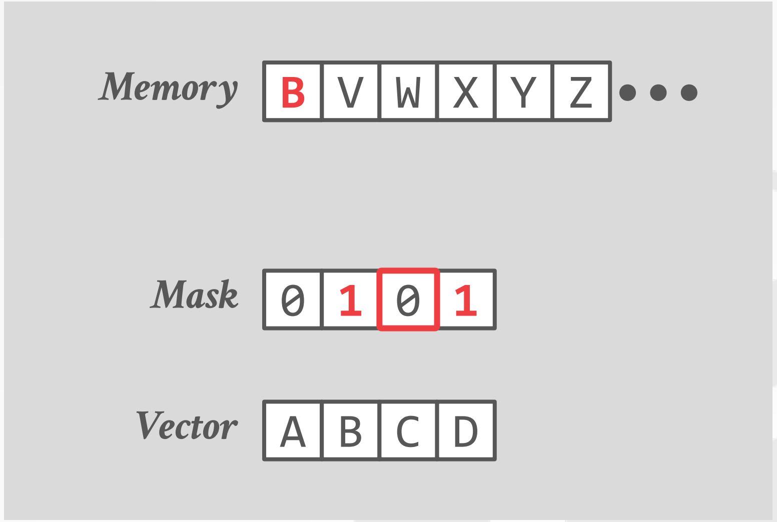


FUNDAMENTAL VECTOR OPERATIONS

Selective Load

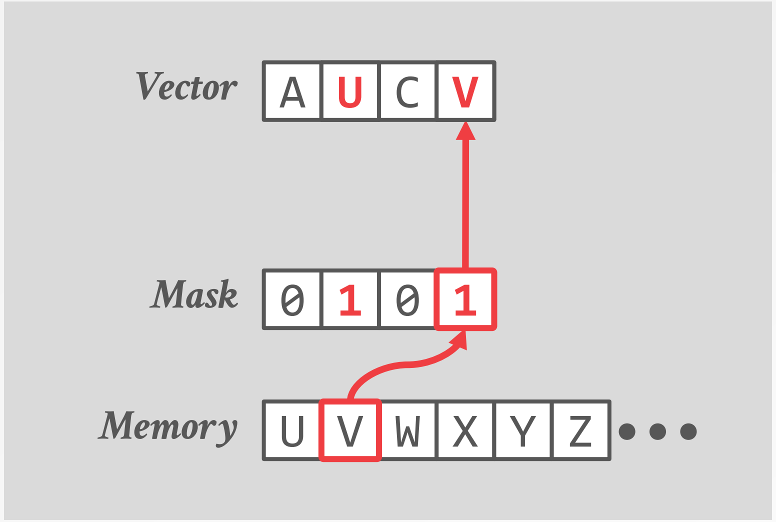


Selective Store

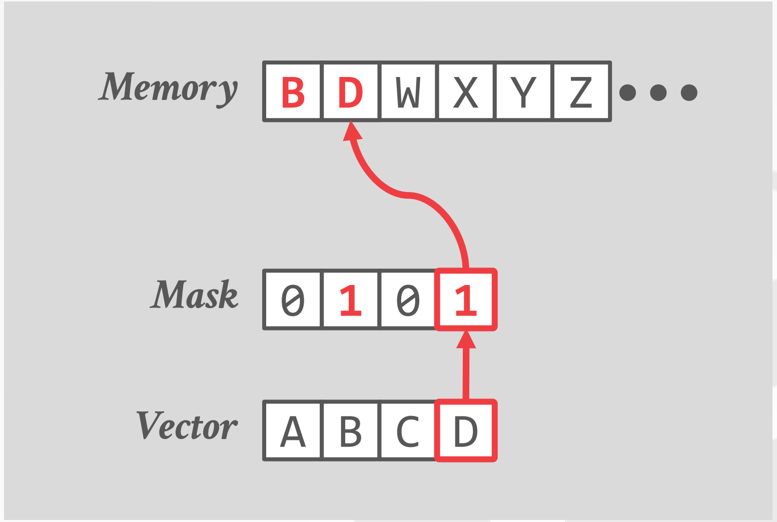


FUNDAMENTAL VECTOR OPERATIONS

Selective Load



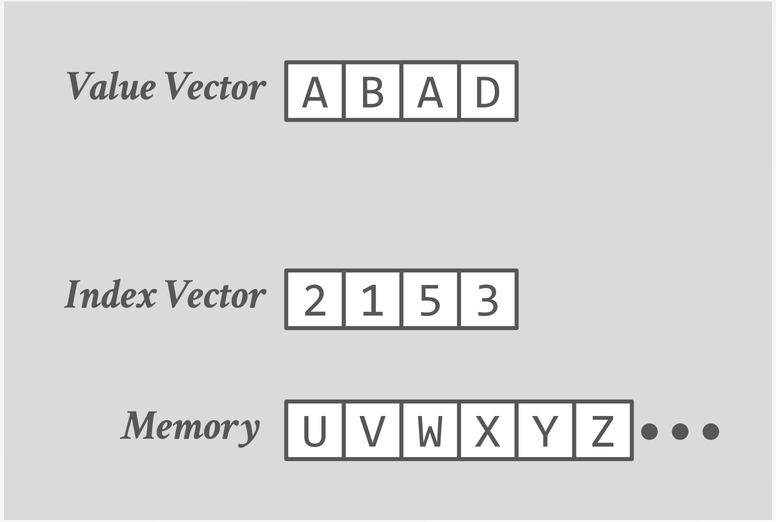
Selective Store



CARNEGIE MELLON DATABASE GROUP

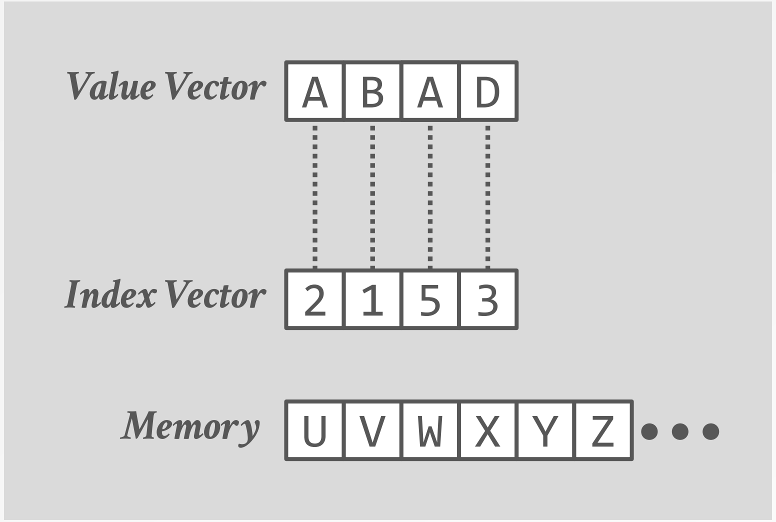
FUNDAMENTAL VECTOR OPERATIONS

Selective Gather



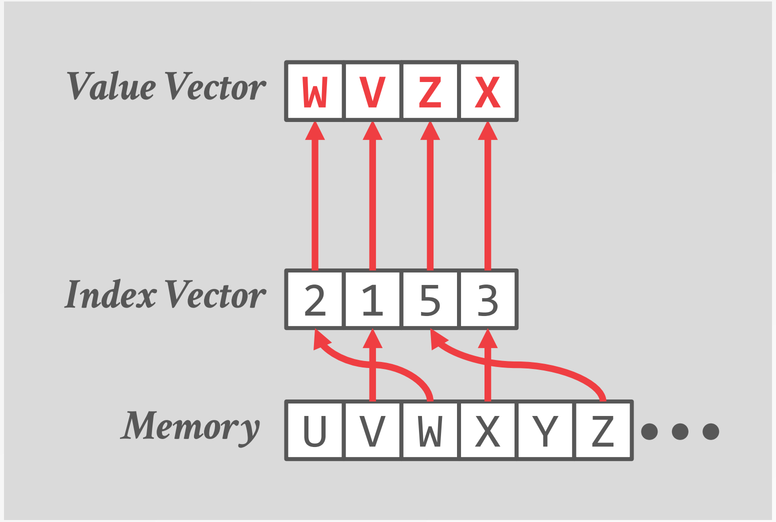
FUNDAMENTAL VECTOR OPERATIONS

Selective Gather



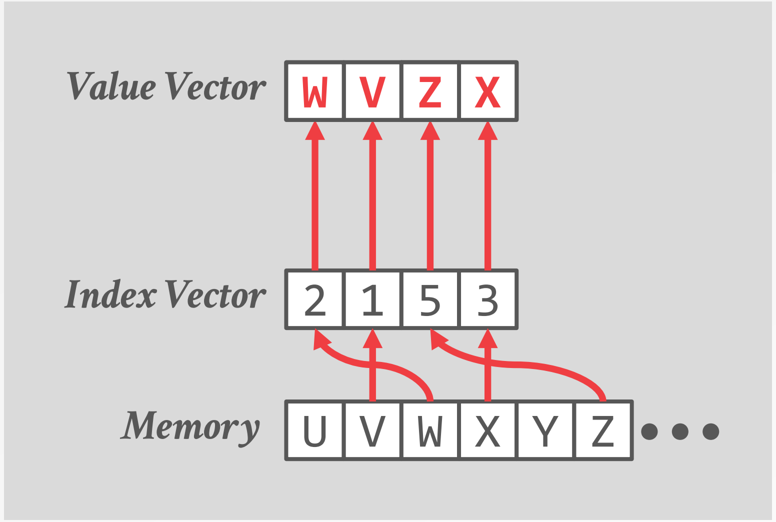
FUNDAMENTAL VECTOR OPERATIONS

Selective Gather

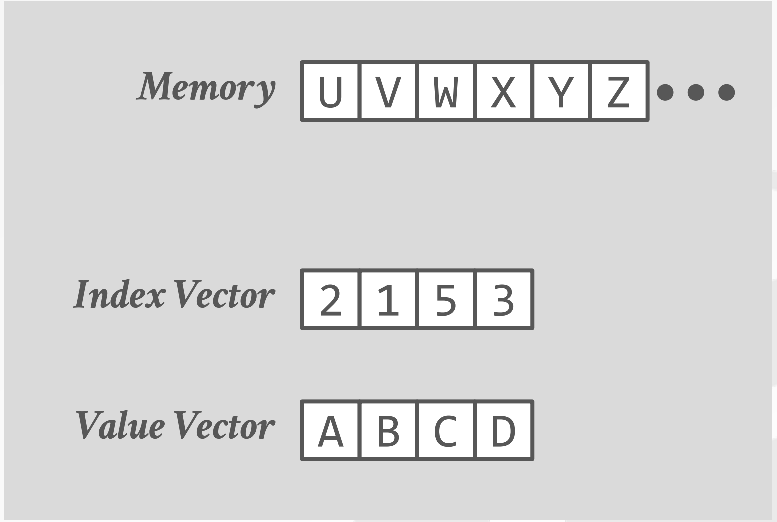


FUNDAMENTAL VECTOR OPERATIONS

Selective Gather

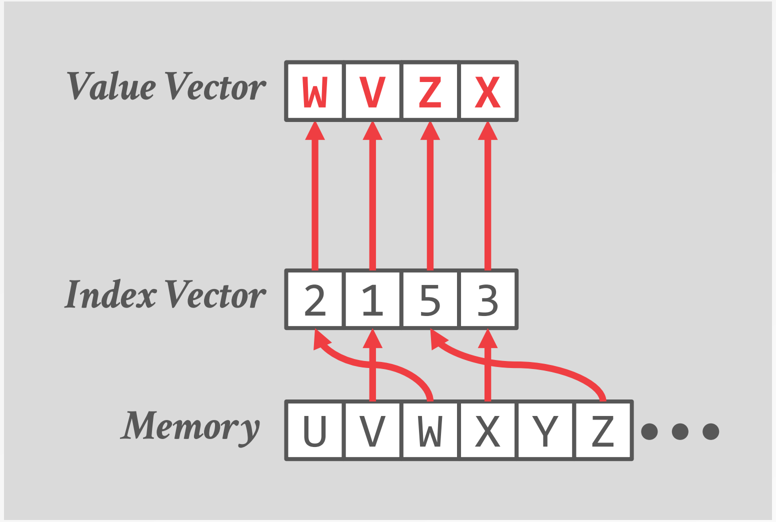


Selective Scatter

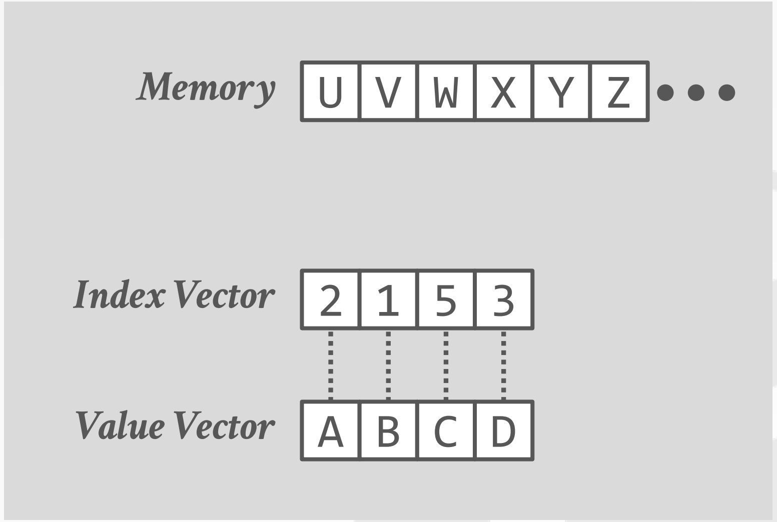


FUNDAMENTAL VECTOR OPERATIONS

Selective Gather

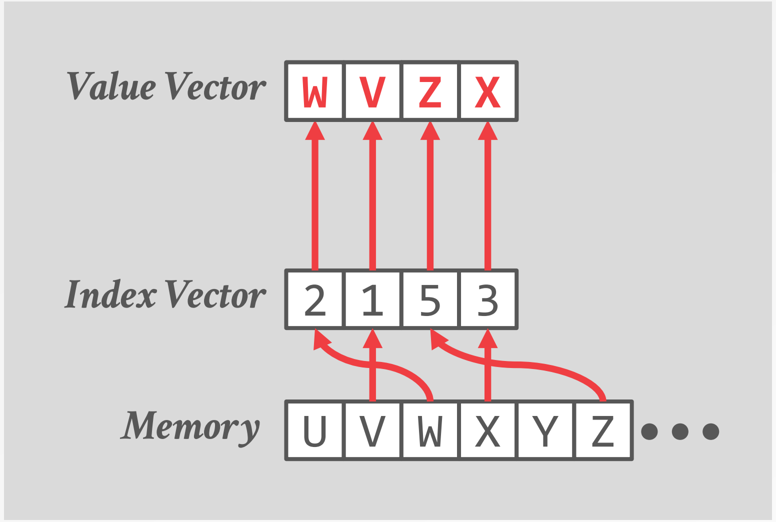


Selective Scatter

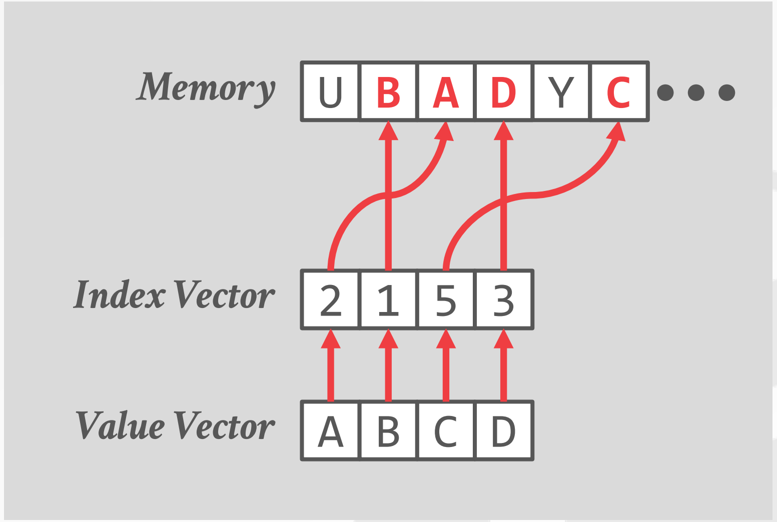


FUNDAMENTAL VECTOR OPERATIONS

Selective Gather



Selective Scatter



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 newer CPUs (Haswell's AVX2).

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)

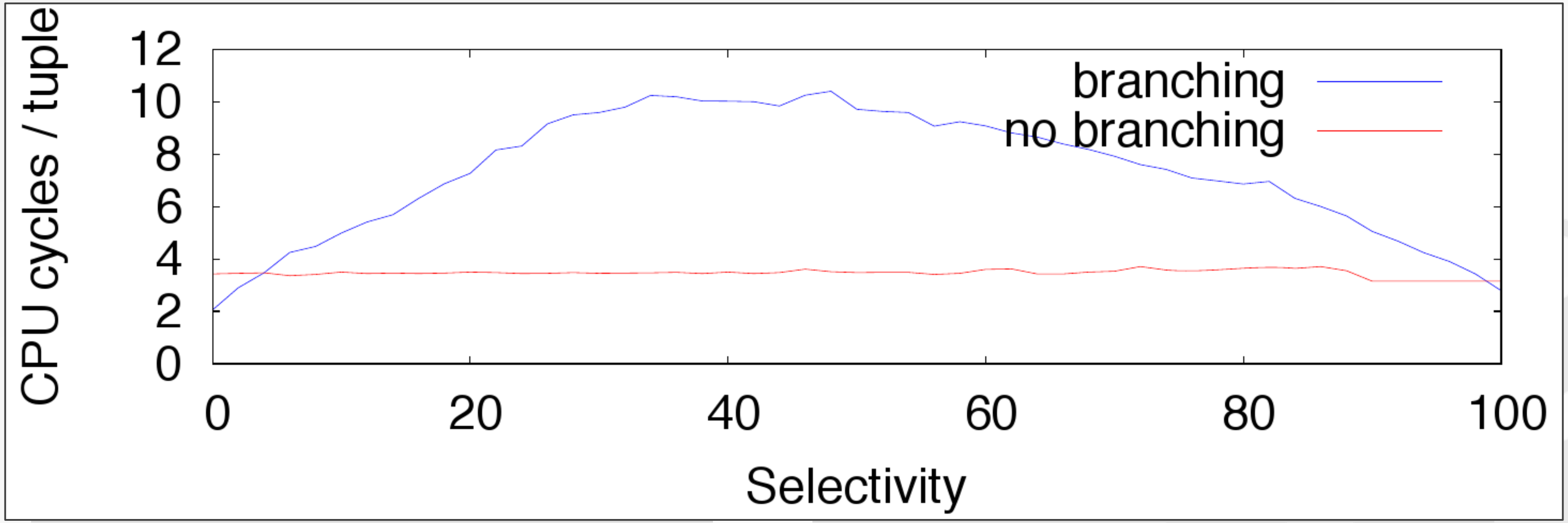
```
i = 0
for t in table:
  key = t.key
  if (key≥low) && (key≤high):
    copy(t, output[i])
    i = i + 1
```

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
```

CARNEGIE MELLON DATABASE GROUP

SELECTION SCANS



Source: Bogdan Raducanu

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"
```

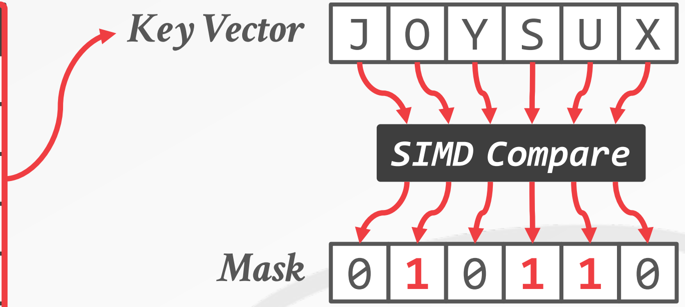
SELECTION SCANS

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

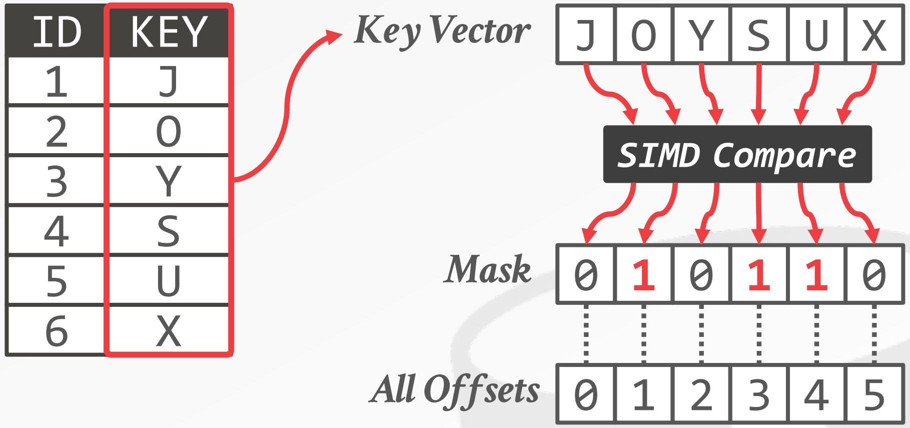


SELECTION SCANS

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

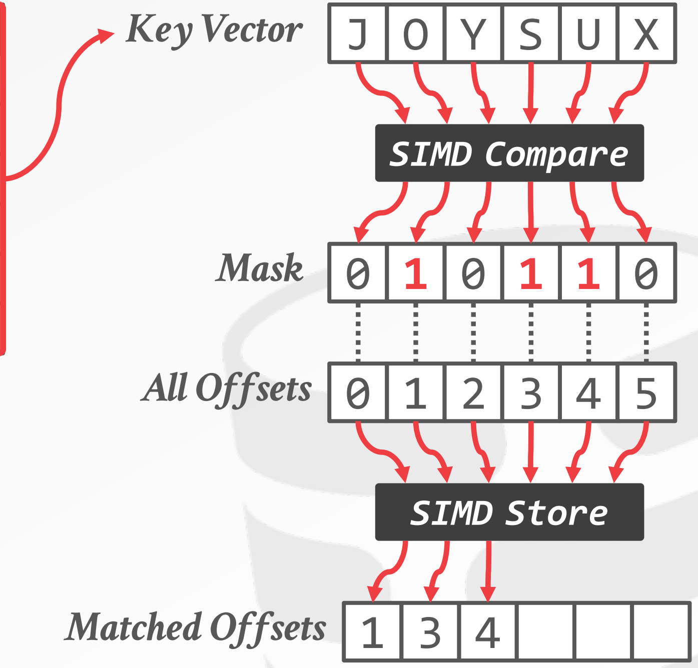
SELECTION SCANS

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|
```

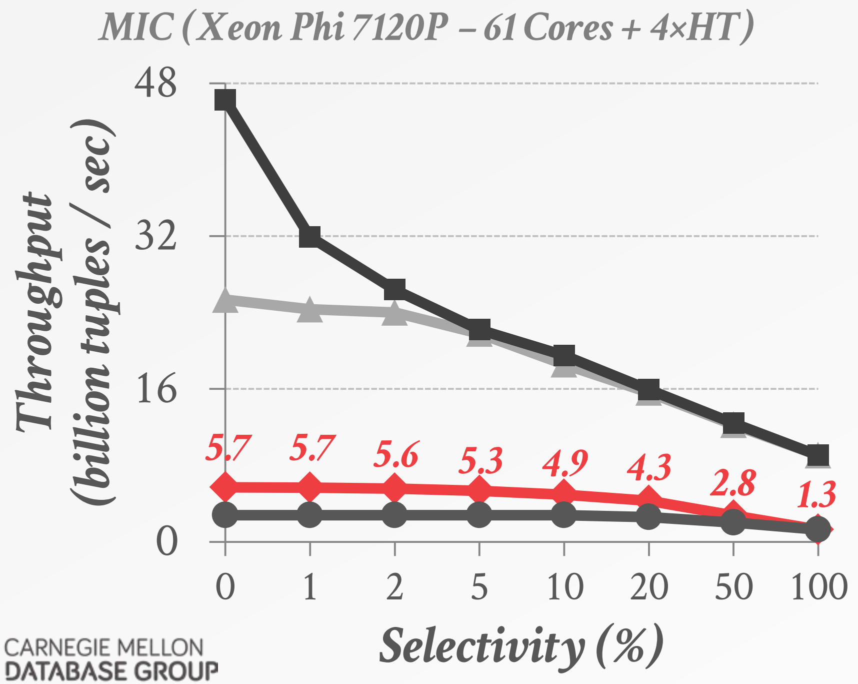
```
SELECT * FROM table
 WHERE key >= "O" AND key <= "U"
```

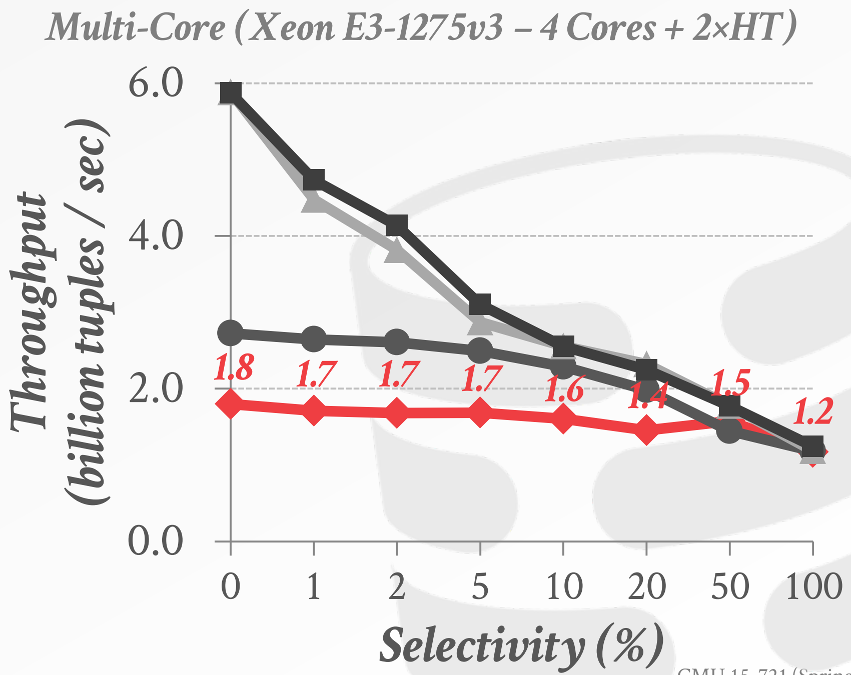
ID	KEY
1	J
2	O
3	Y
4	S
5	U
6	X



SELECTION SCANS

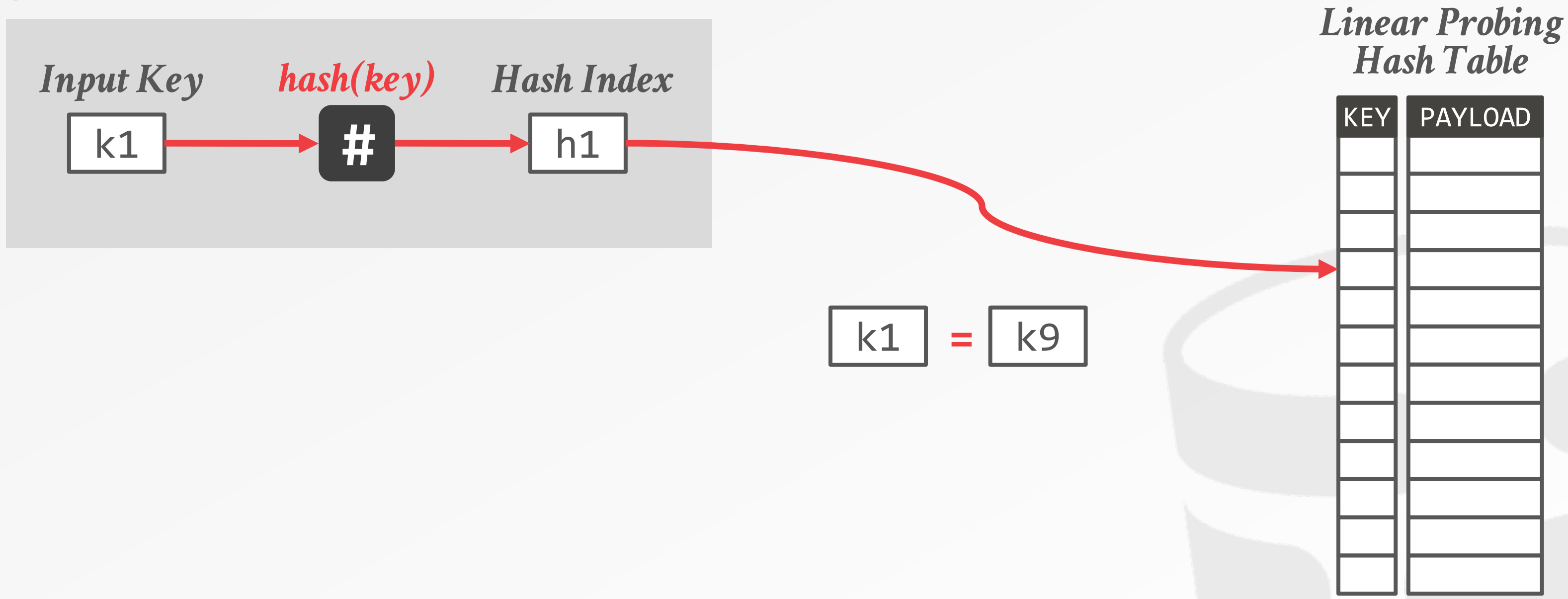
◆ Scalar (Branching) ▲ Vectorized (Early Mat)
● Scalar (Branchless) ■ Vectorized (Late Mat)





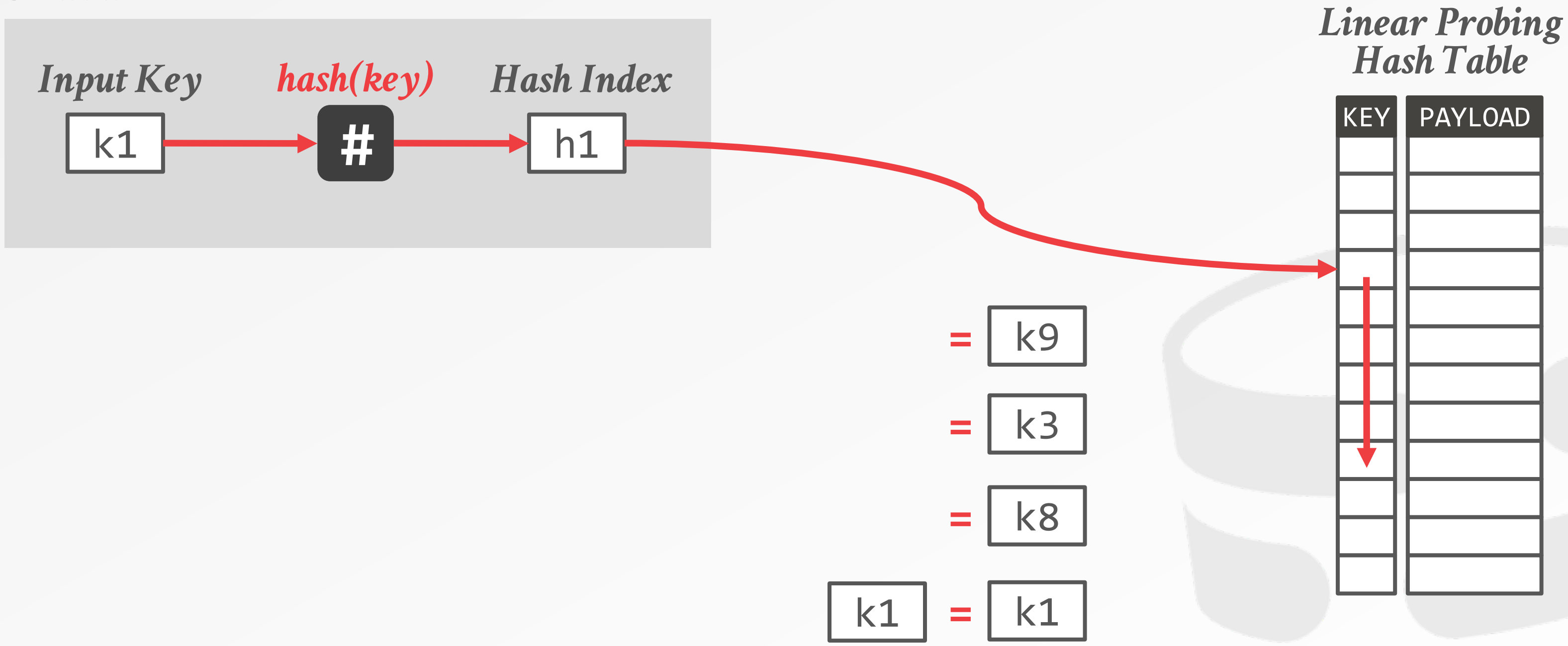
HASH TABLES – PROBING

Scalar

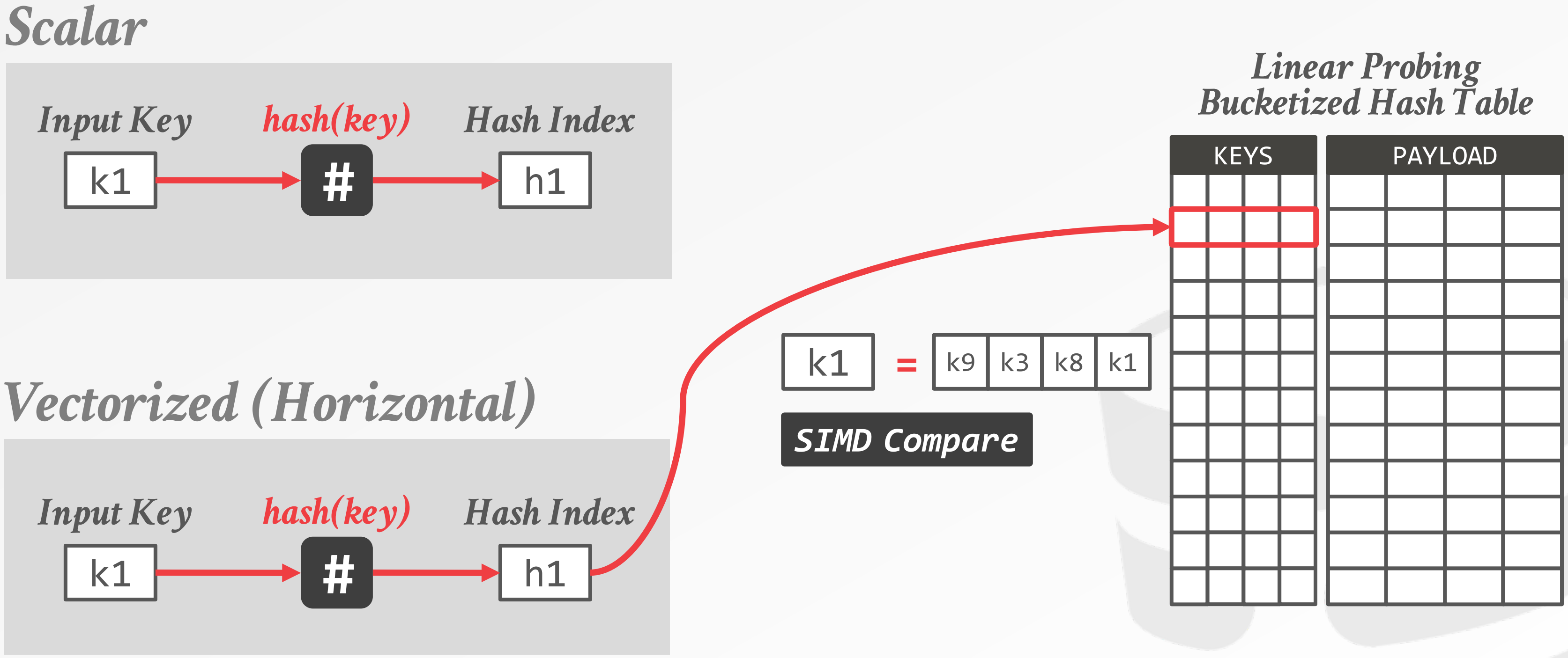


HASH TABLES – PROBING

Scalar

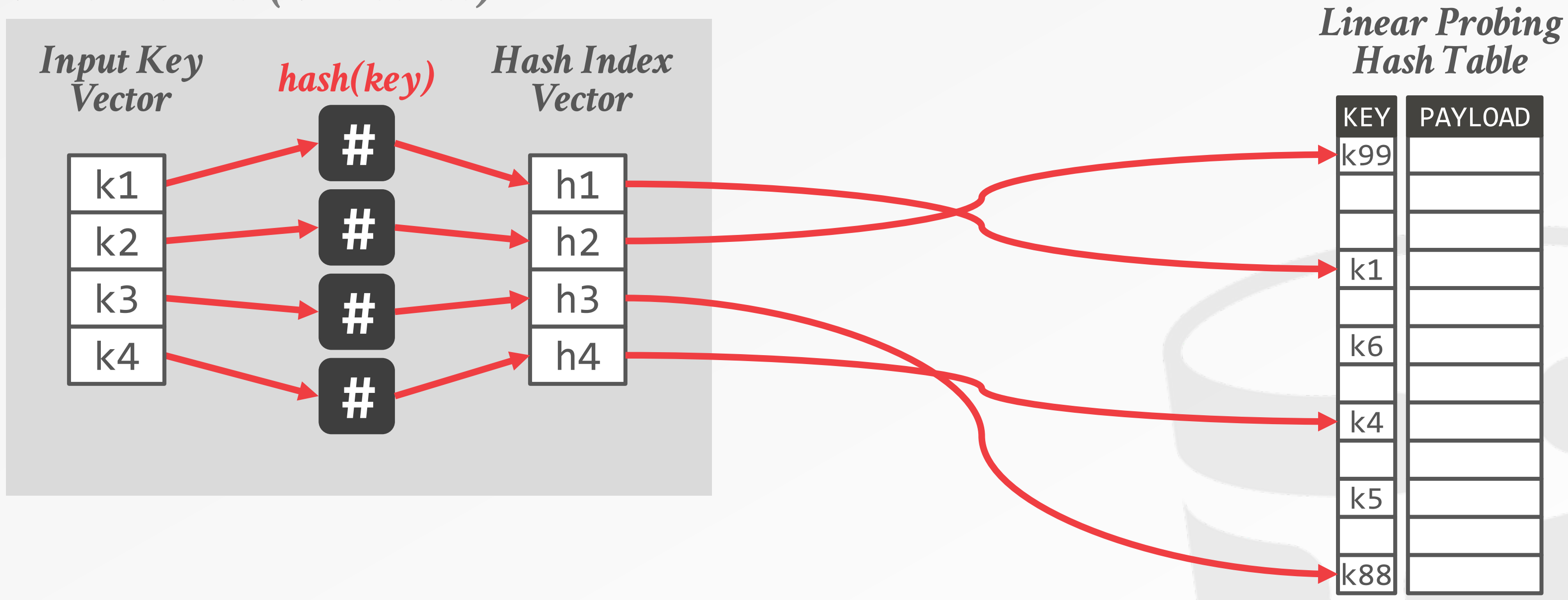


HASH TABLES – PROBING



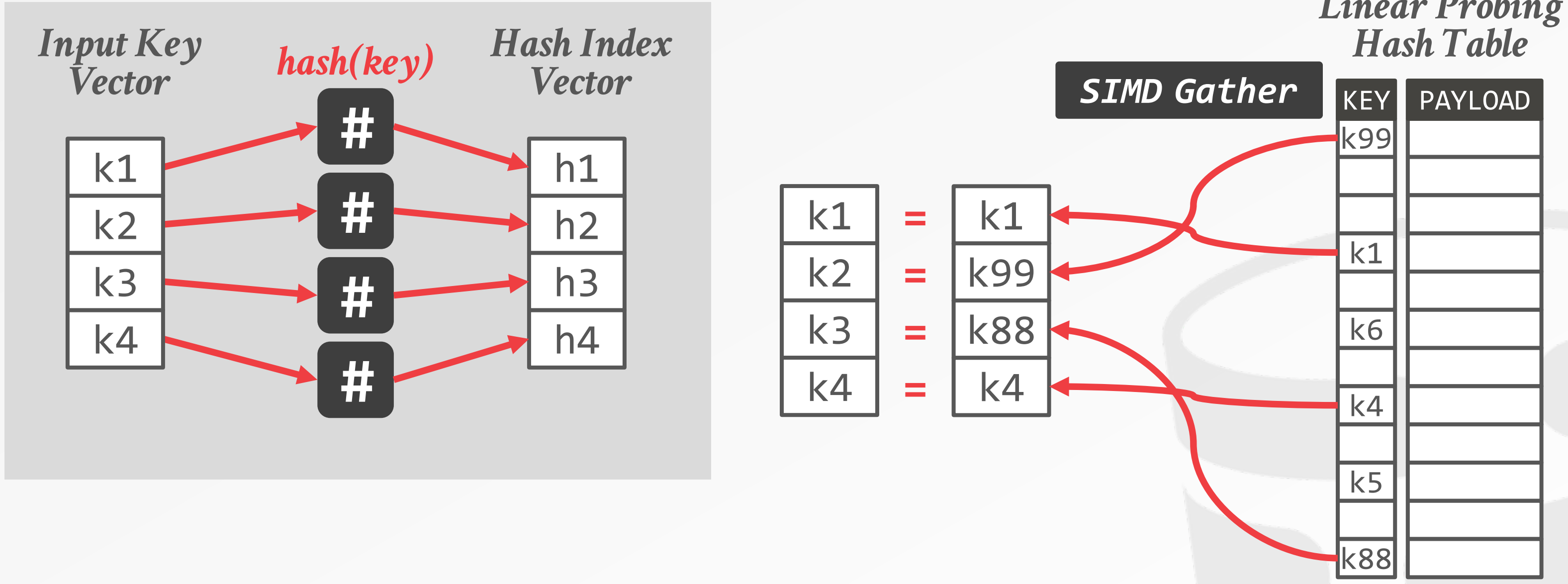
HASH TABLES – PROBING

Vectorized (Vertical)



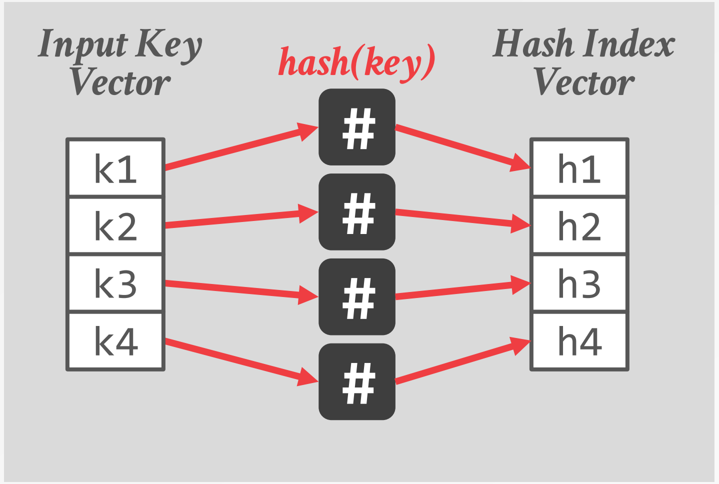
HASH TABLES – PROBING

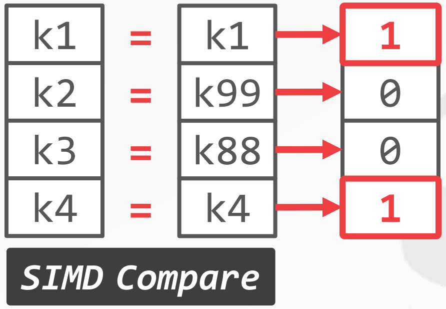
Vectorized (Vertical)

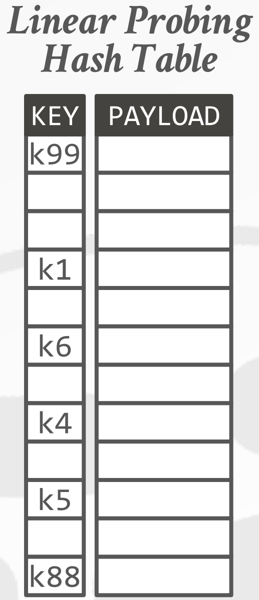


HASH TABLES – PROBING

Vectorized (Vertical)

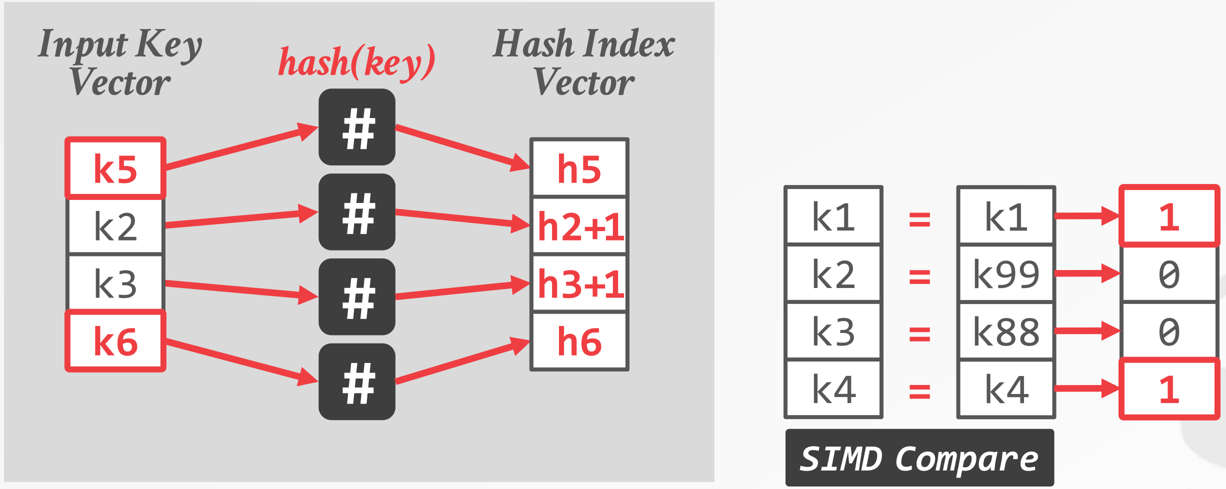


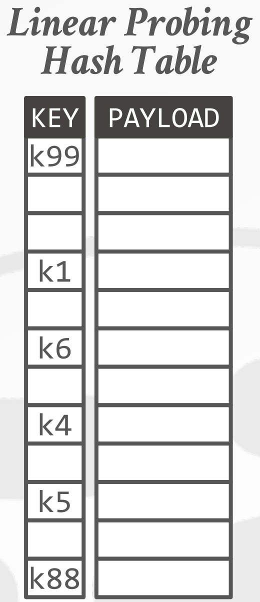




HASH TABLES – PROBING

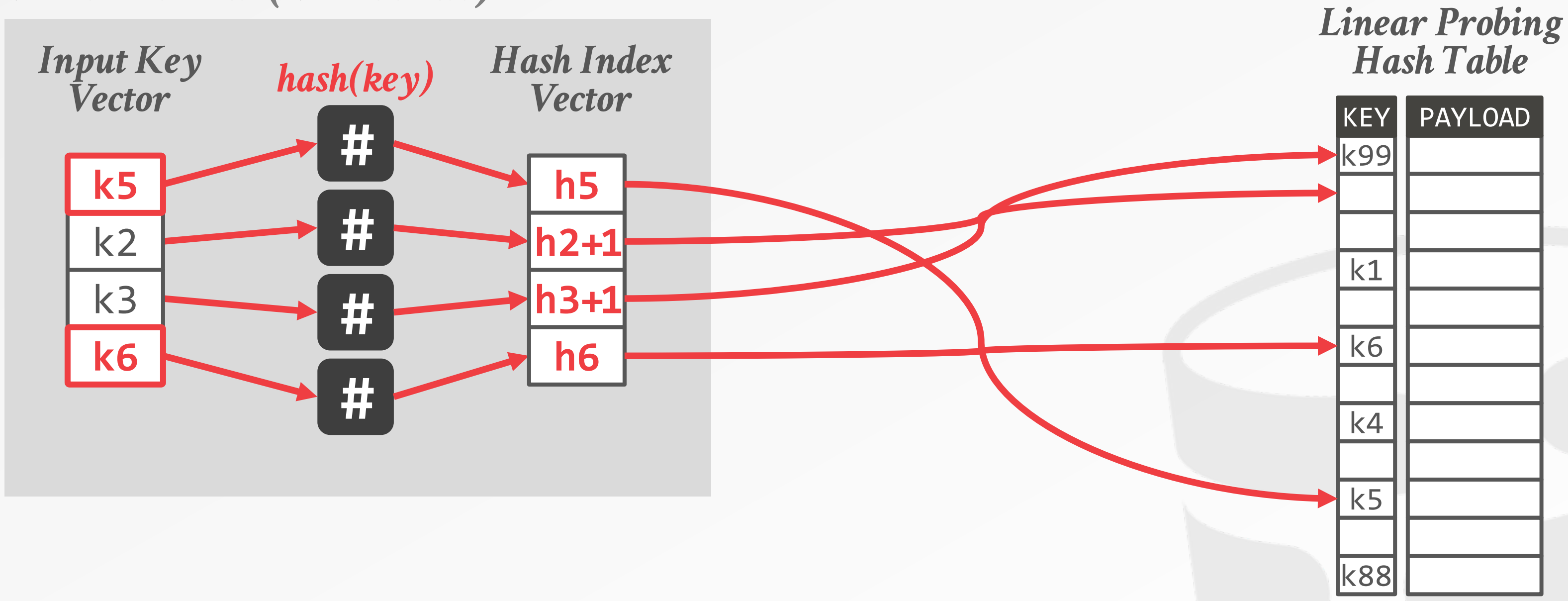
Vectorized (Vertical)



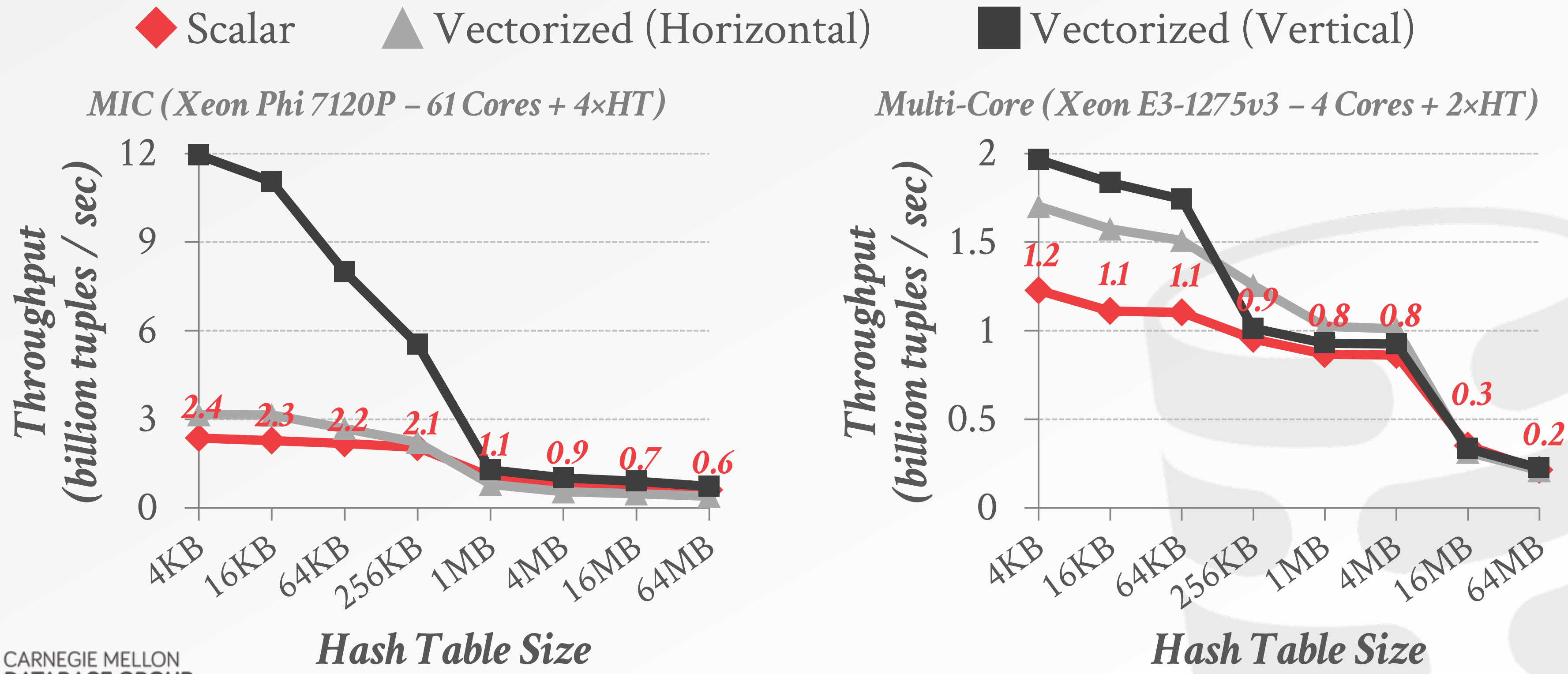


HASH TABLES – PROBING

Vectorized (Vertical)

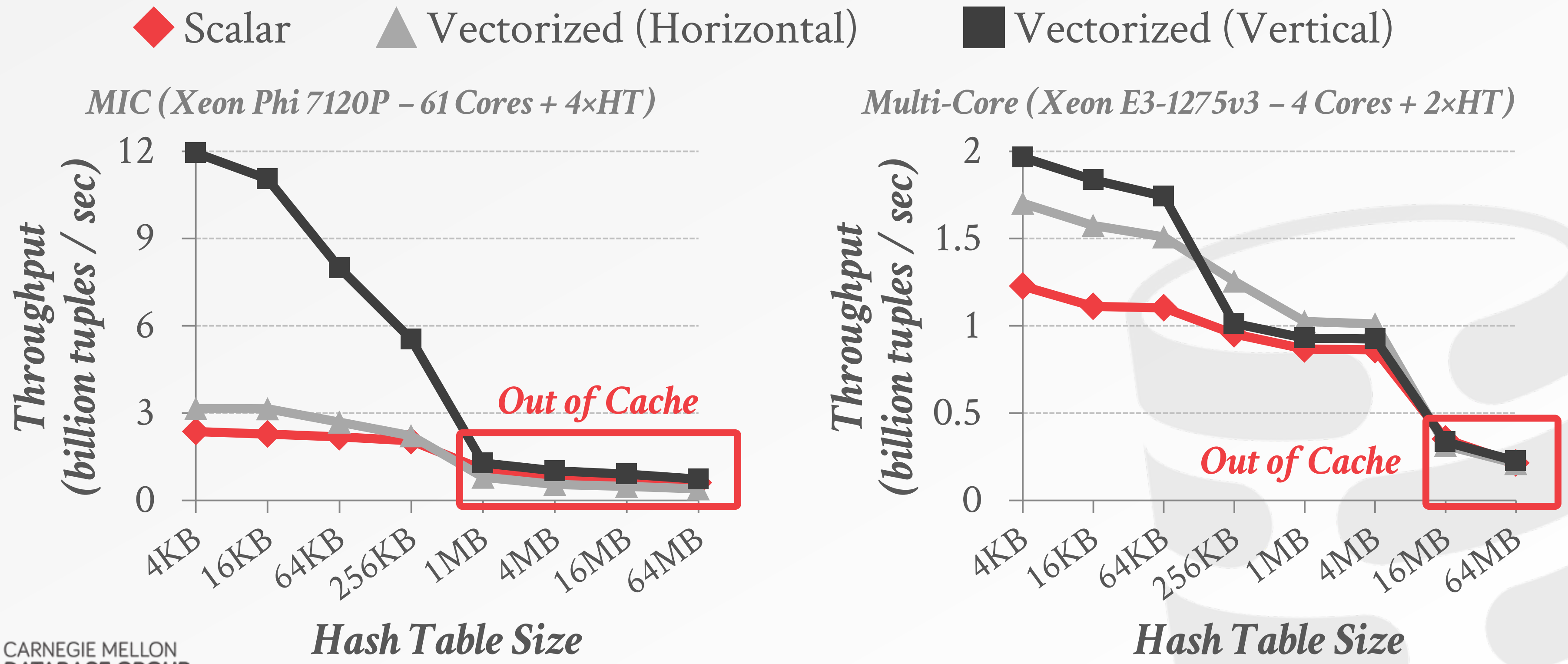


HASH TABLES – PROBING



CARNEGIE MELLON DATABASE GROUP

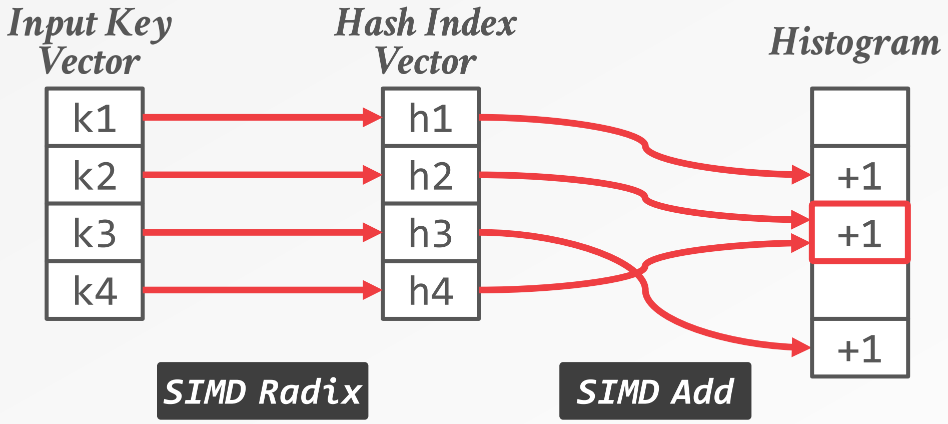
HASH TABLES – PROBING



PARTITIONING – HISTOGRAM

Use scatter and gathers to increment counts.

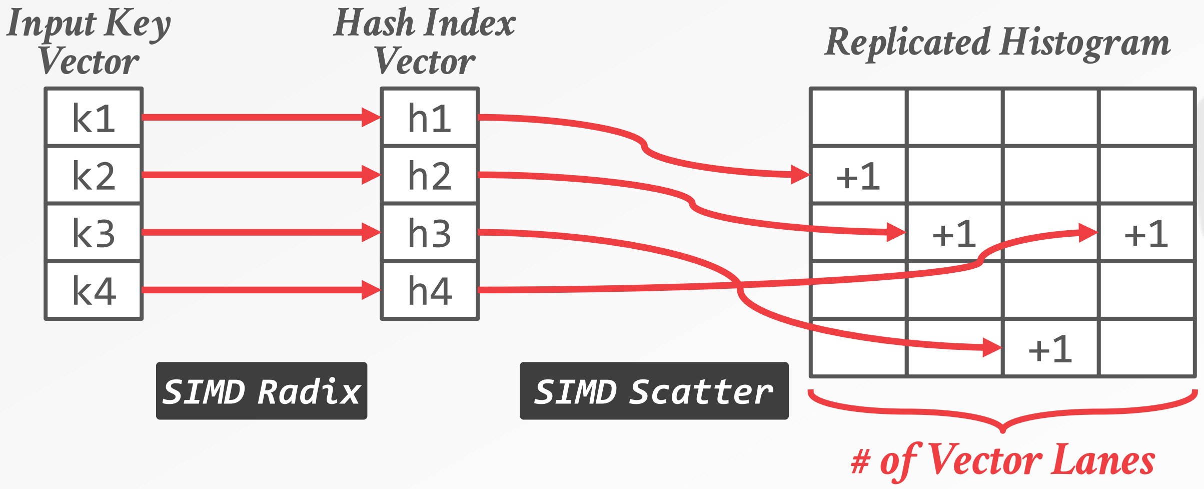
Replicate the histogram to handle collisions.



PARTITIONING – HISTOGRAM

Use scatter and gathers to increment counts.

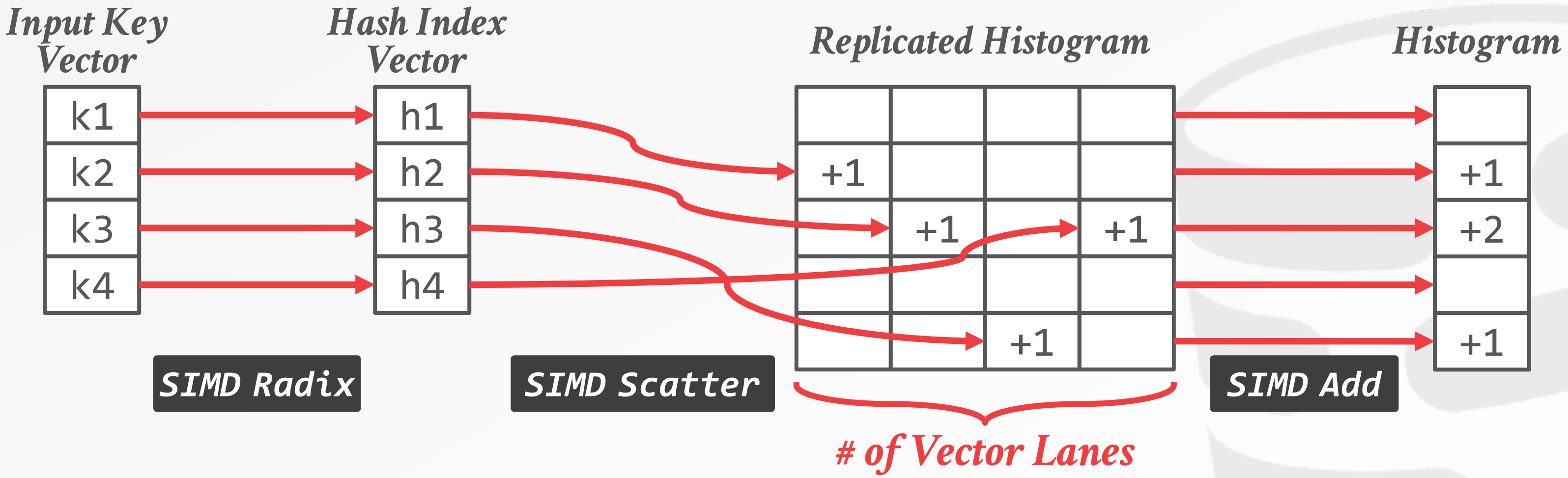
Replicate the histogram to handle collisions.



PARTITIONING – HISTOGRAM

Use scatter and gathers to increment counts.

Replicate the histogram to handle collisions.



JOINS

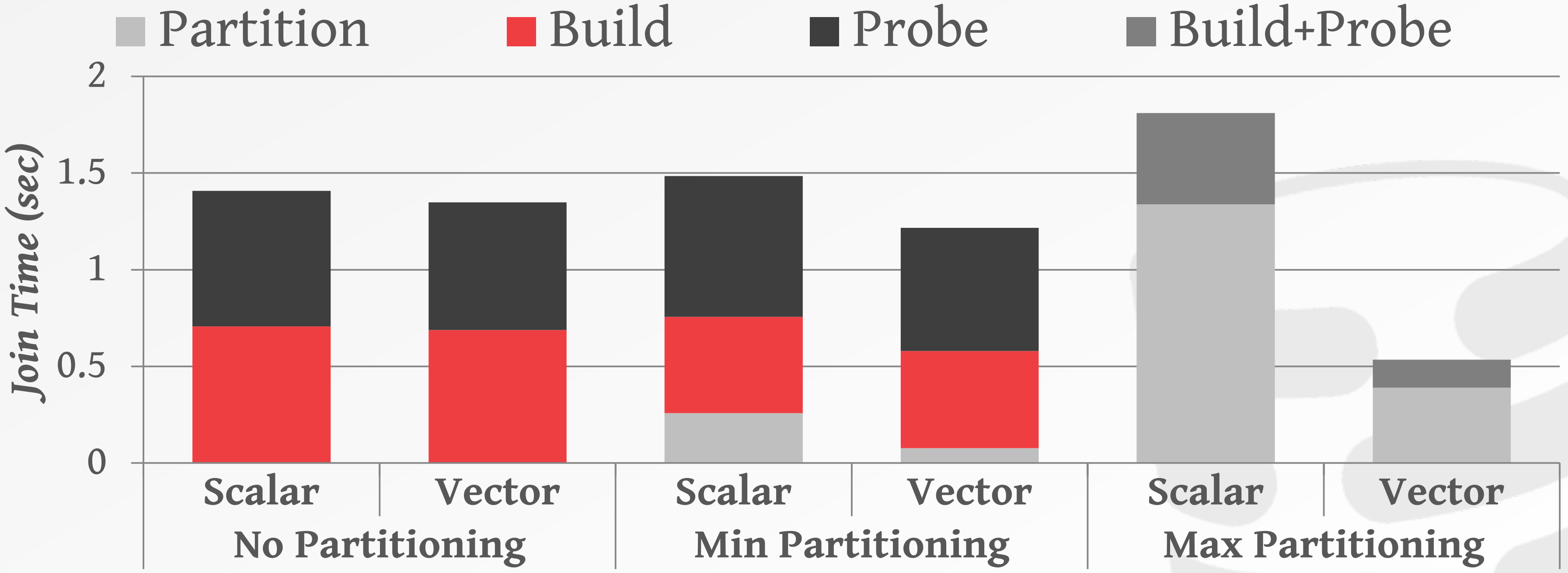
No Partitioning
→ Build one shared hash table using atomics
→ Partially vectorized

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 + 4×HT



PARTING THOUGHTS

Vectorization is essential for OLAP queries.

These algorithms don't work when the data exceeds your CPU cache.

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.

NEXT CLASS

Vectorization (Part II)

Code Review Submission: April 11th
Project Status Meetings: April 13th
Project #3 Status Updates: April 18th