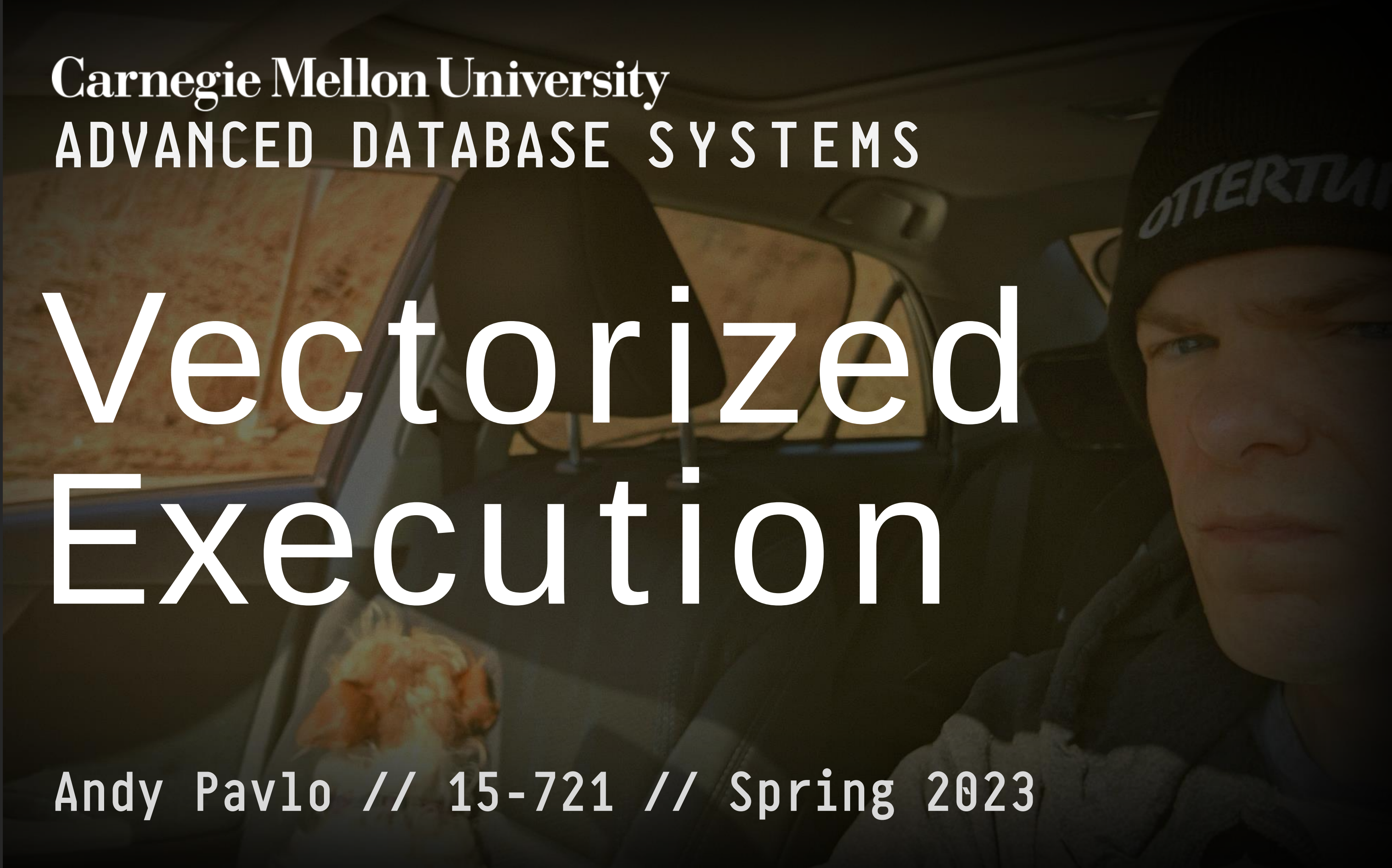
Lecture #08

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Vectorized Execution

Andy Pavlo // 15-721 // Spring 2023



LAST CLASS

We discussed how the DBMS will divide up tasks among its workers to execute a query.

The DBMS needs to be aware of the location of data to avoid non-local memory access.

TODAY'S AGENDA

Background
Implementation Approaches
SIMD Fundamentals
Vectorized DBMS Algorithms

VECTORIZATION

The process of converting an algorithm's 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.

WHY THIS MATTERS

Say we can parallelize our algorithm over 32 cores.

Assume each core has a 4-wide SIMD registers.

Potential Speed-up: 32x × 4x = 128x

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.

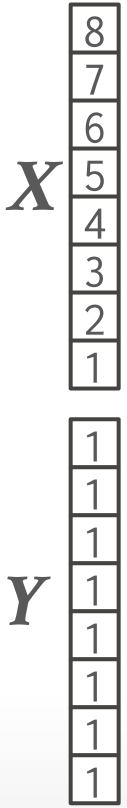
→ **x86**: MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX2, AVX512
→ **PowerPC**: Altivec
→ **ARM**: NEON, SVE
→ **RISC-V**: RVV

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];
}
```



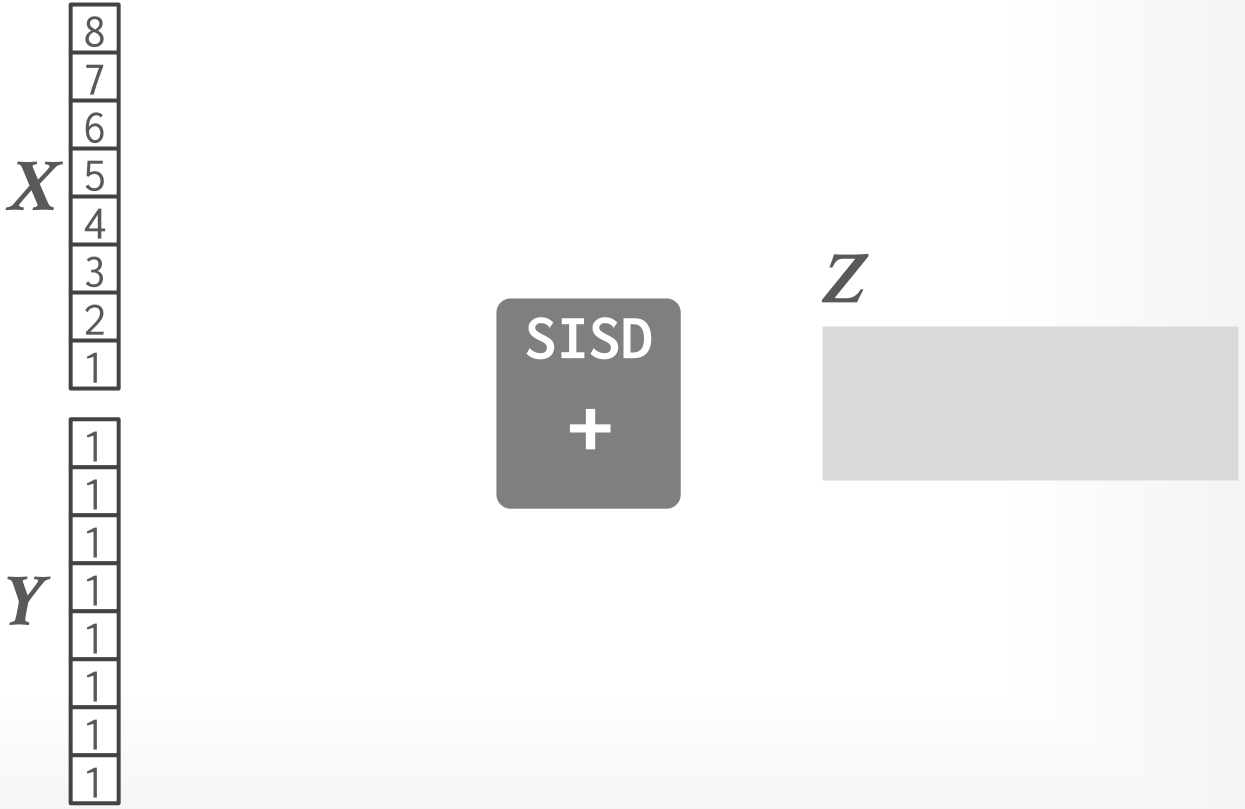
Z

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];
}
```

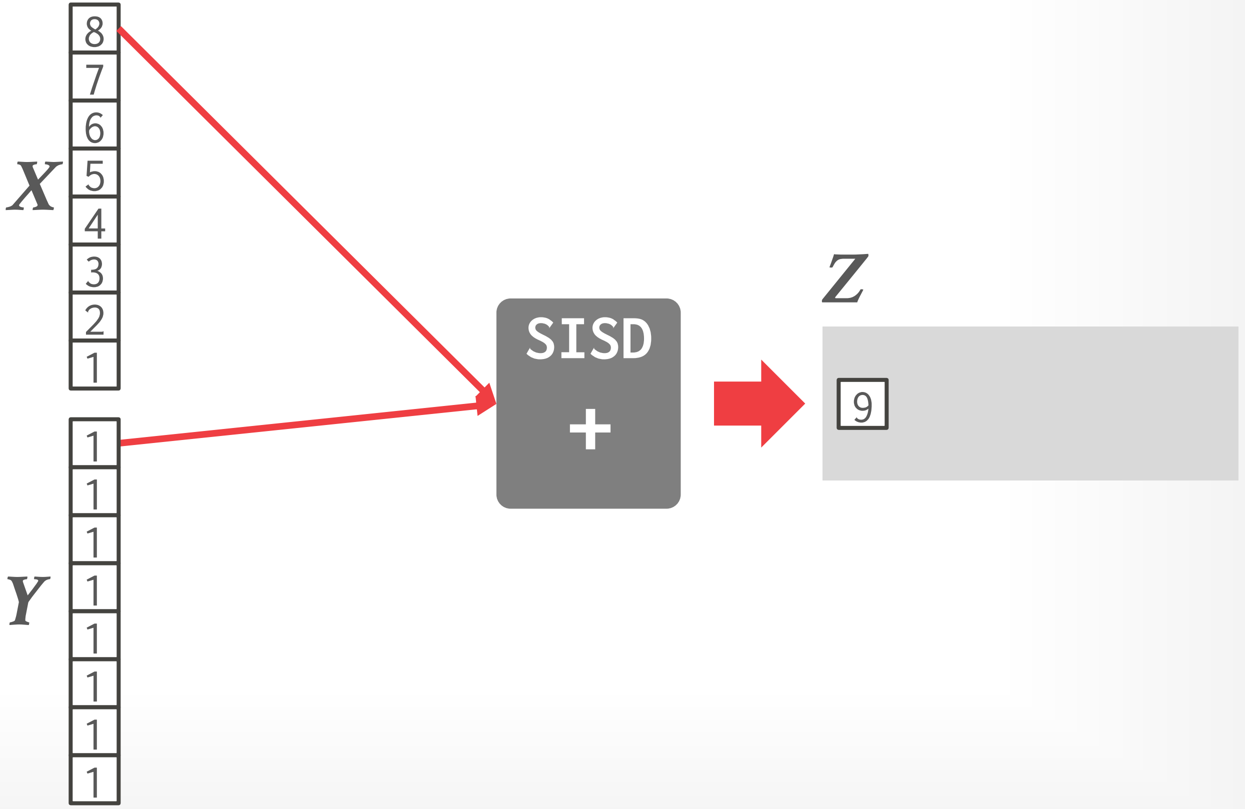


SIMD EXAMPLE

$X + Y = Z$

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

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

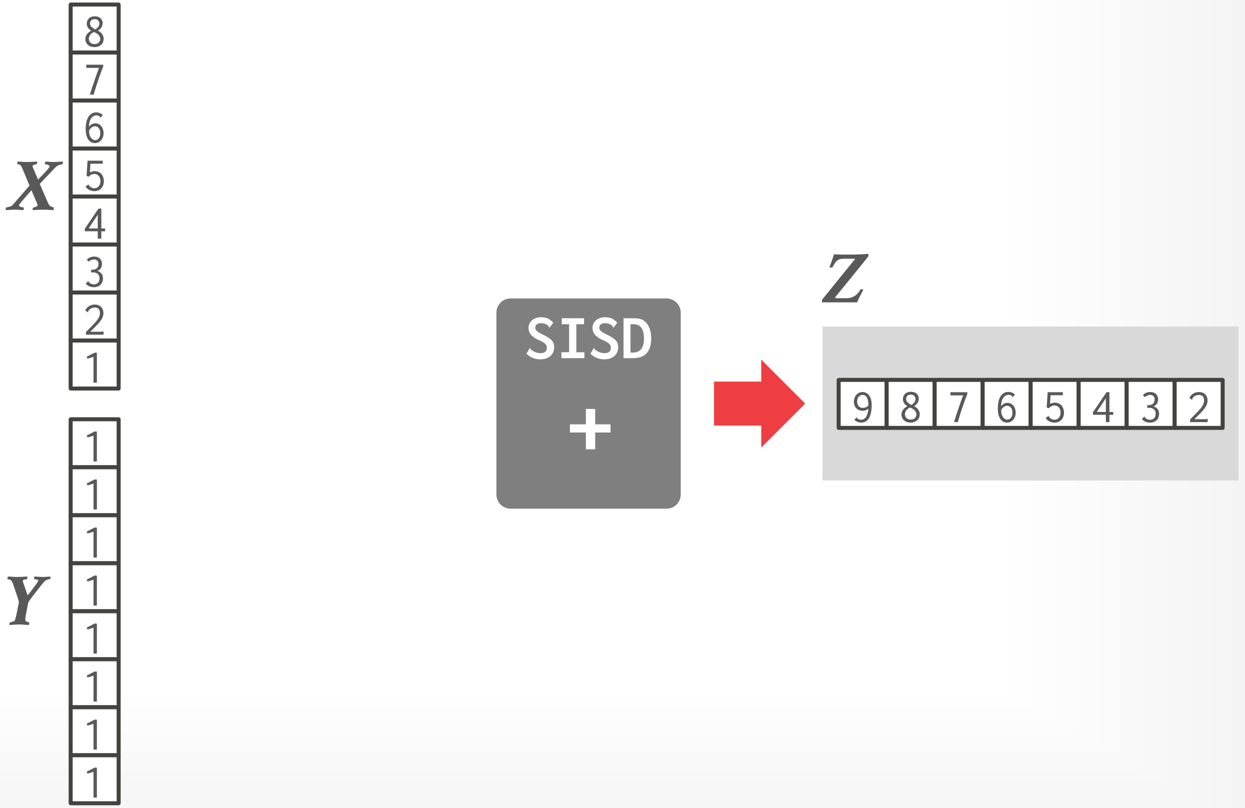


SIMD EXAMPLE

$X + Y = Z$

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

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

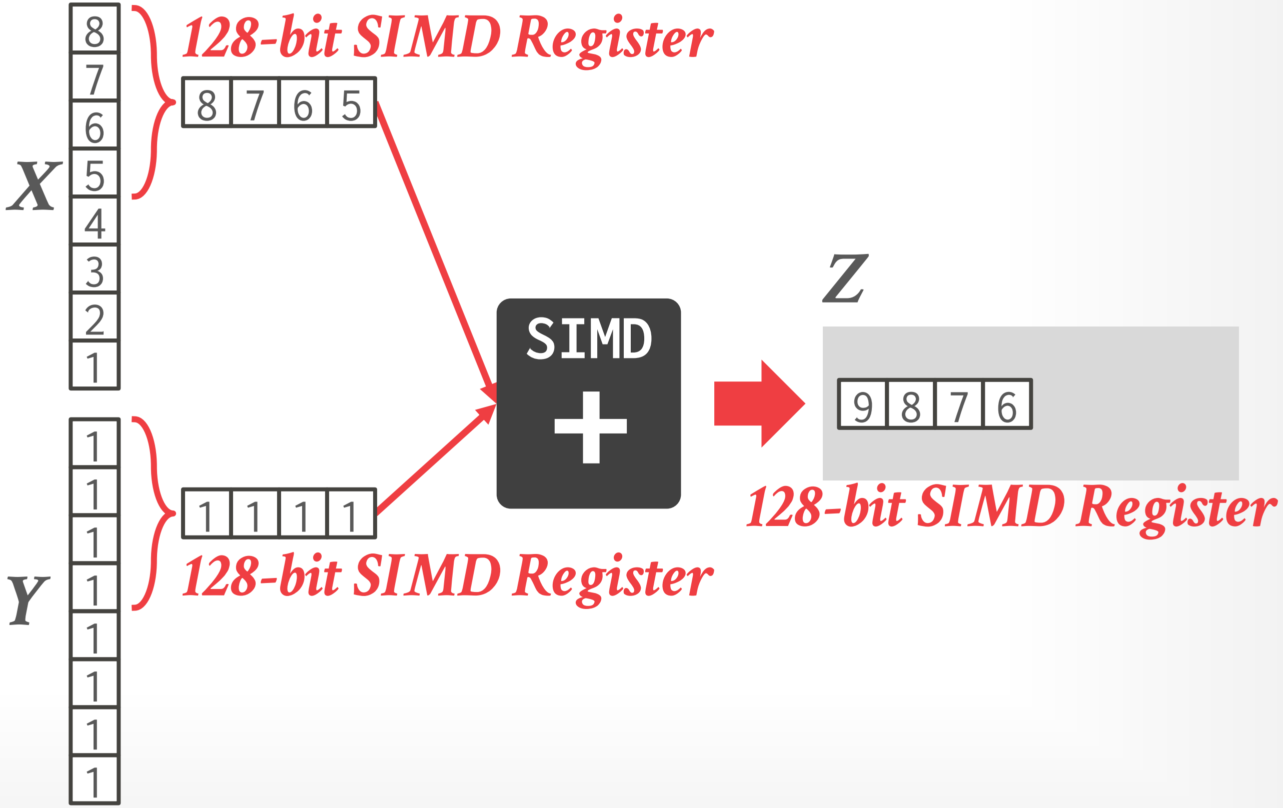


SIMD EXAMPLE

$X + Y = Z$

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

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

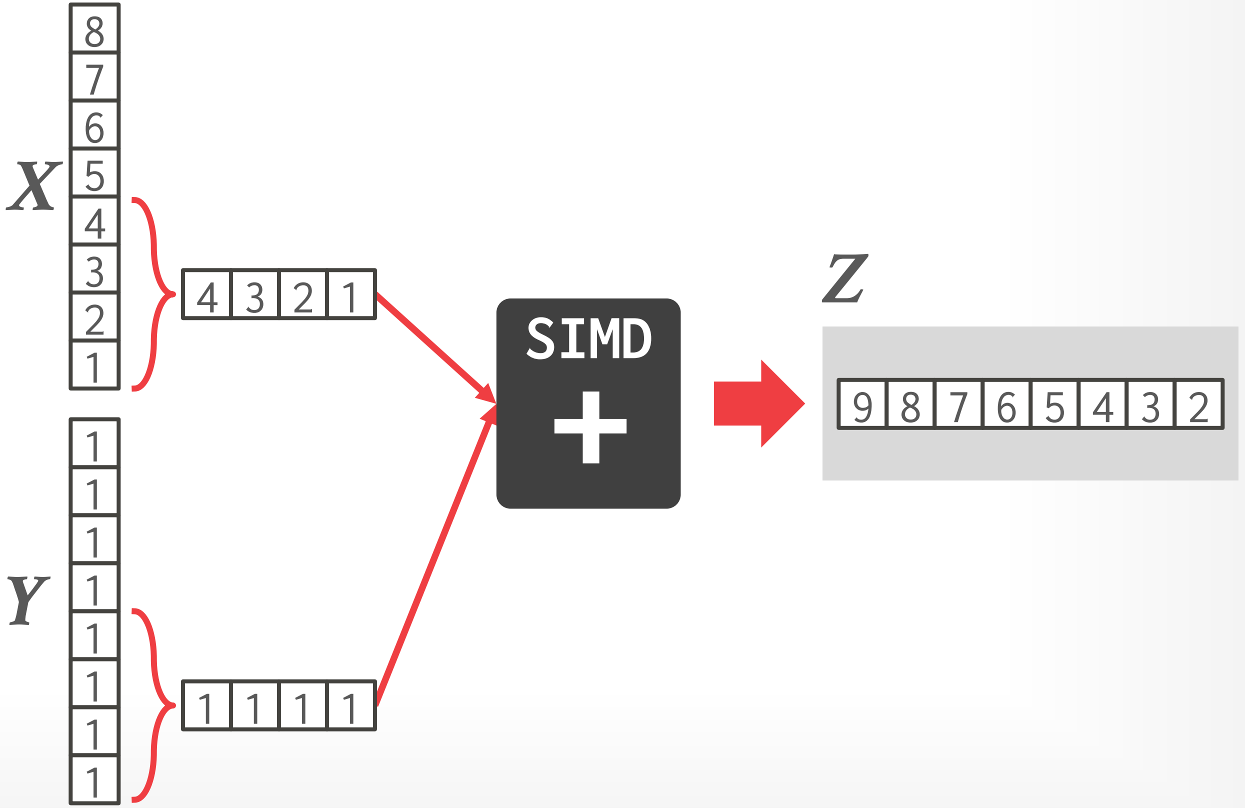


SIMD EXAMPLE

$X + Y = Z$

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

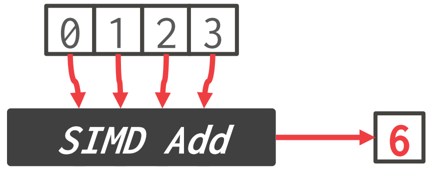
```
for (i=0; i<n; i++) {
  Z[i] = X[i] + Y[i];
}
```



VECTORIZATION DIRECTION

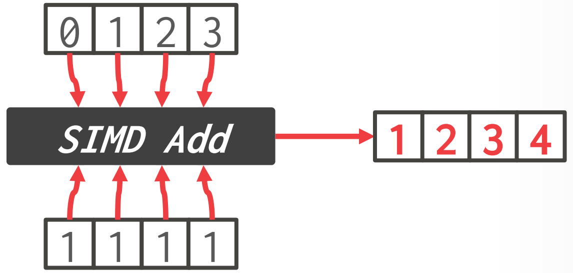
Approach #1: Horizontal

→ Perform operation on all elements together within a single vector.



Approach #2: Vertical

→ Perform operation in an elementwise manner on elements of each vector.



Source: Przemysław Karpiński

SIMD 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**

SIMD INSTRUCTIONS (2)

Comparison Instructions

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

Shuffle instructions

→ Move data 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

SIMD TRADE-OFFS

Advantages:

→ Significant performance gains and resource utilization if an algorithm can be vectorized.

Disadvantages:

→ Implementing an algorithm using SIMD is still mostly a manual process.
→ SIMD may have restrictions on data alignment.
→ Gathering data into SIMD registers and scattering it to the correct locations is tricky and/or inefficient.

No Longer True in AVX-512!

AVX-512

Intel's 512-bit extensions to the AVX2 instructions.

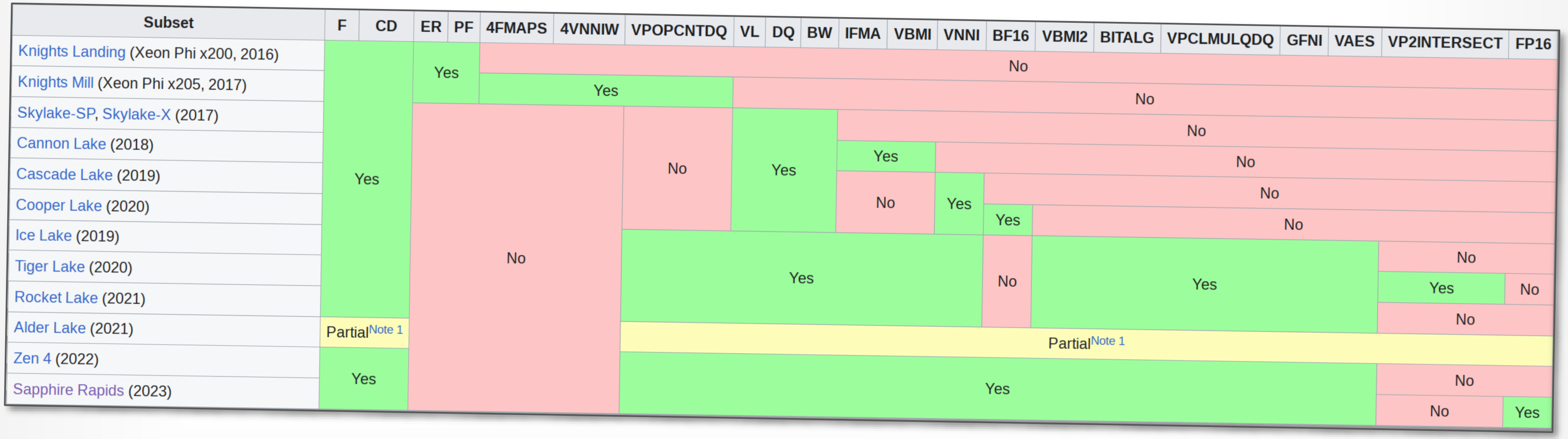
→ Provides new operations to support data conversions, scatter, and permutations.

Unlike previous SIMD extensions, Intel split AVX-512 into groups that CPUs can selectively provide (except for "foundation" extension AVX-512F).

AVX-512

Intel's 512-bit extensions to the AVX2 instructions.

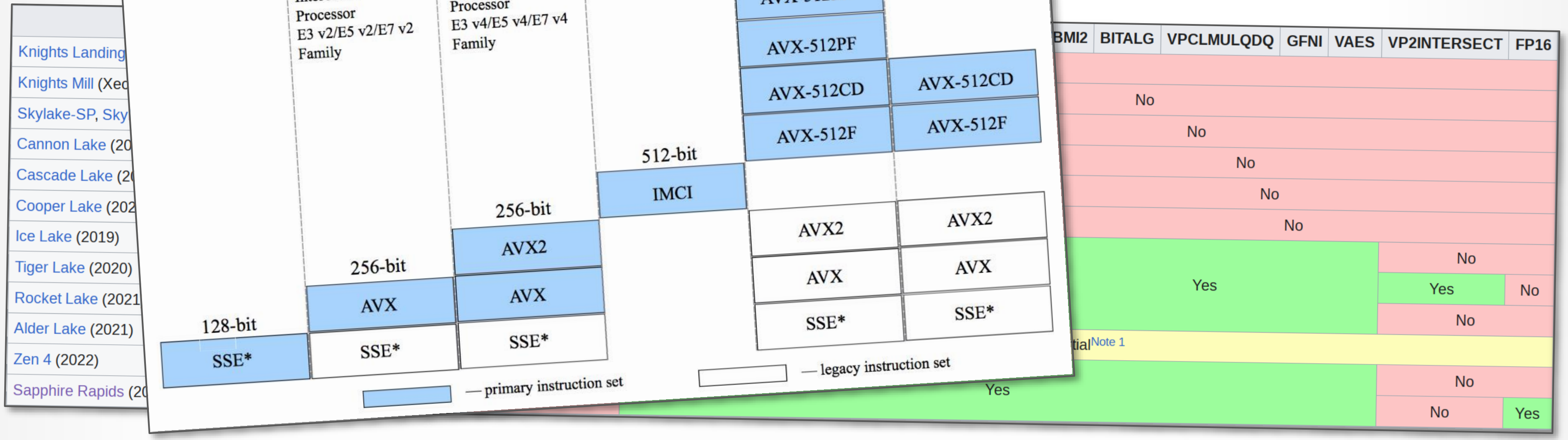
→ Provides new operations to support data conversions, scatter, and permutations.



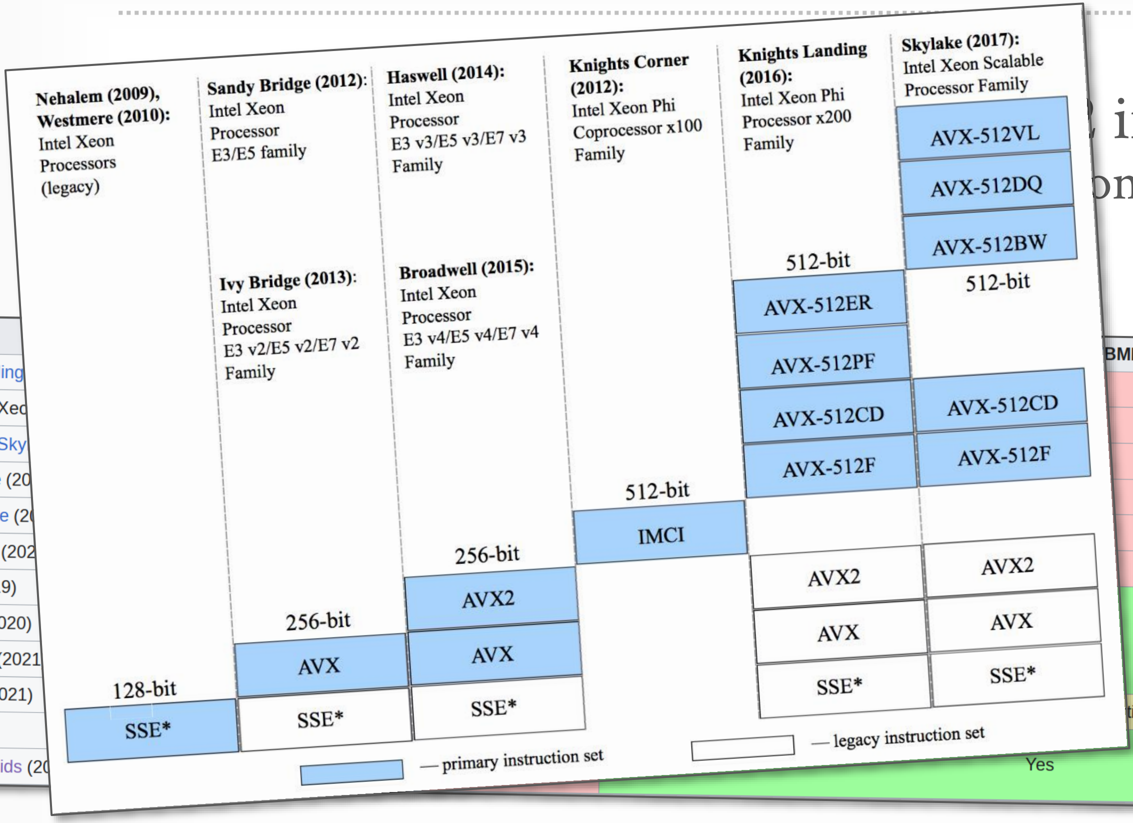
Subset	F	CD	ER	PF	4FMAPS	4VNNIW	VPOPCNTDQ	VL	DQ	BW	IFMA	VBMI	VNNI	BF16	VBMI2	BITALG	VPCLMULQDQ	GFNI	VAES	VP2INTERSECT	FP16
Knights Landing (Xeon Phi x200, 2016)	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No
Knights Mill (Xeon Phi x205, 2017)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No	No	No	No	No	No	No	No
Skylake-SP, Skylake-X (2017)	Yes	Yes	No	No	No	No	No	Yes	Yes	Yes	No	No	No	No	No	No	No	No	No	No	No
Cannon Lake (2018)	Yes	Yes	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No	No	No
Cascade Lake (2019)	Yes	Yes	No	No	No	No	No	Yes	Yes	Yes	No	No	Yes	No	No	No	No	No	No	No	No
Cooper Lake (2020)	Yes	Yes	No	No	No	No	No	Yes	Yes	Yes	No	No	Yes	Yes	No	No	No	No	No	No	No
Ice Lake (2019)	Yes	Yes	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	No	No
Tiger Lake (2020)	Yes	Yes	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	No
Rocket Lake (2021)	Yes	Yes	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	No	No
Alder Lake (2021)	Partial[Note 1]	Partial[Note 1]	No	No	No	No	Partial[Note 1]	Partial[Note 1]	Partial[Note 1]	Partial[Note 1]	Partial[Note 1]	Partial[Note 1]	Partial[Note 1]	Partial[Note 1]	Partial[Note 1]	Partial[Note 1]	Partial[Note 1]	Partial[Note 1]	Partial[Note 1]	Partial[Note 1]	Partial[Note 1]
Zen 4 (2022)	Yes	Yes	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No
Sapphire Rapids (2023)	Yes	Yes	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes

AVX-512

2 instructions.
onversions,



	BMI2	BITALG	VPCLMULQDQ	GFNI	VAES	VP2INTERSECT	FP16
Knights Landing							
Knights Mill (Xe		No					
Skylake-SP, Sky			No				
Cannon Lake (20				No			
Cascade Lake (2				No			
Cooper Lake (202					No		
Ice Lake (2019)			Yes			No	
Tiger Lake (2020)			Yes			Yes	No
Rocket Lake (2021			Yes			No	
Alder Lake (2021)	tialNote 1						
Zen 4 (2022)		Yes				No	
Sapphire Rapids (2		Yes				No	Yes

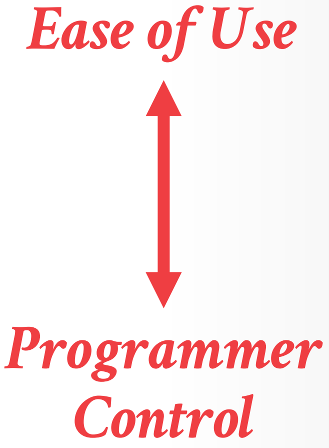

Nehalem (2009), Westmere (2010): Intel Xeon Processors (legacy)
Sandy Bridge (2012): Intel Xeon Processor E3/E5 family
Ivy Bridge (2013): Intel Xeon Processor E3 v2/E5 v2/E7 v2 Family
Haswell (2014): Intel Xeon Processor E3 v3/E5 v3/E7 v3 Family
Broadwell (2015): Intel Xeon Processor E3 v4/E5 v4/E7 v4 Family
Knights Corner (2012): Intel Xeon Phi Coprocessor x100 Family
Knights Landing (2016): Intel Xeon Phi Processor x200 Family
Skylake (2017): Intel Xeon Scalable Processor Family
128-bit
SSE*
256-bit
AVX
SSE*
256-bit
AVX2
AVX
SSE*
512-bit
IMCI
512-bit
AVX-512ER
AVX-512PF
AVX-512CD
AVX-512F
AVX2
AVX
SSE*
AVX-512VL
AVX-512DQ
AVX-512BW
512-bit
AVX-512CD
AVX-512F
AVX2
AVX
SSE*
— primary instruction set
— legacy instruction set

IMPLEMENTATION

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];
  }
}
```

*Z=*X+1

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 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 is up to the DBMS developer to make sure that this is correct.

EXPLICIT VECTORIZATION

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

→ Not portable across CPUs (ISAs / versions).

There are libraries that hide the underlying calls to SIMD intrinsics.

→ Google Highway
→ Simd
→ Expressive Vector Engine (EVE)
→ std::simd (Experimental)

EXPLICIT VECTORIZATION

```
void add(int *X,
         int *Y,
         int *Z) {
  __mm128i *vecX = (__m128i*)X;
  __mm128i *vecY = (__m128i*)Y;
  __mm128i *vecZ = (__m128i*)Z;
  for (int i=0; i<MAX/4; i++) {
    _mm_store_si128(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.

VECTORIZATION FUNDAMENTALS

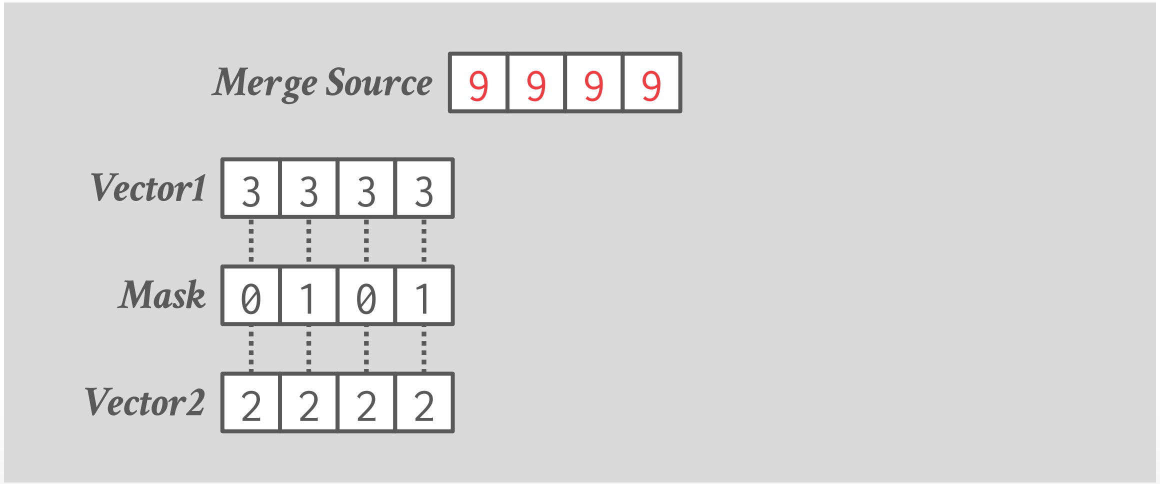
There are fundamental SIMD operations that the DBMS will use to build more complex functionality:

→ Masking
→ Permute
→ Selective Load/Store
→ Compress/Expand
→ Selective Gather/Scatter

MAKE THE MOST OUT OF YOUR SIMD INVESTMENTS: COUNTER CONTROL FLOW DIVERGENCE IN COMPILED QUERY PIPELINES
VLDB JOURNAL 2020

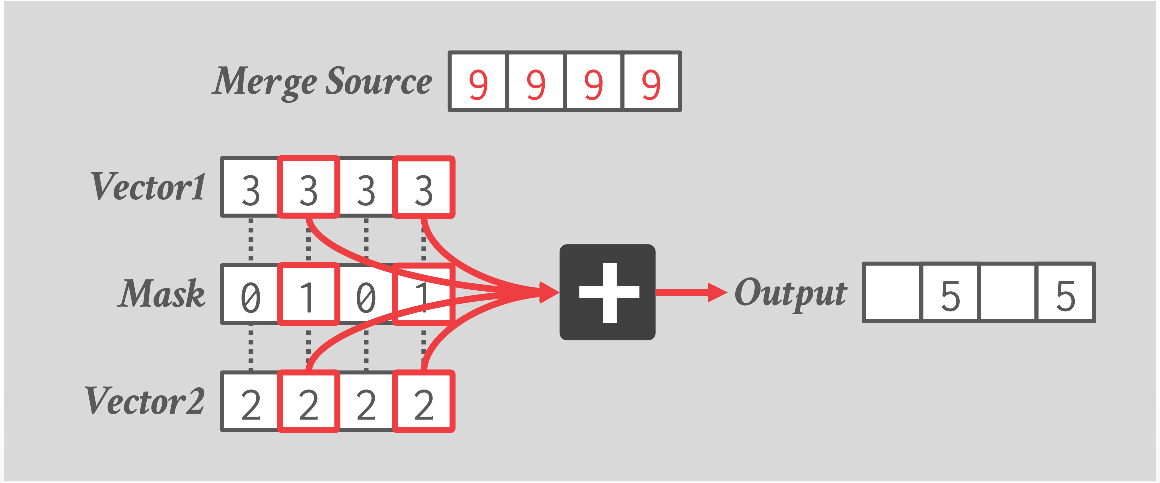
SIMD MASKING

Almost all AVX-512 operations support **predication** variants whereby the CPU only performs operations on lanes specified by an input bitmask.



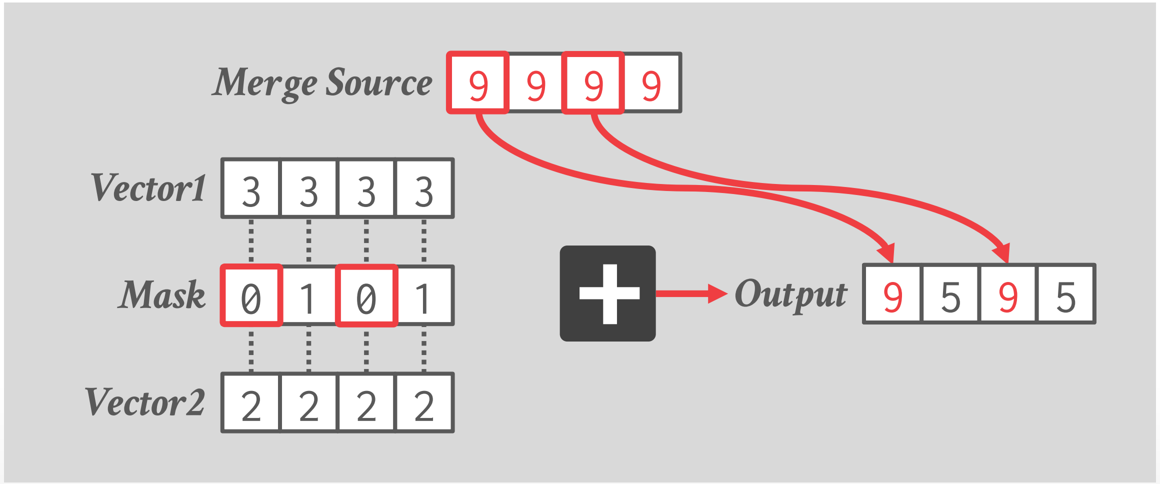
SIMD MASKING

Almost all AVX-512 operations support **predication** variants whereby the CPU only performs operations on lanes specified by an input bitmask.



SIMD MASKING

Almost all AVX-512 operations support **predication** variants whereby the CPU only performs operations on lanes specified by an input bitmask.

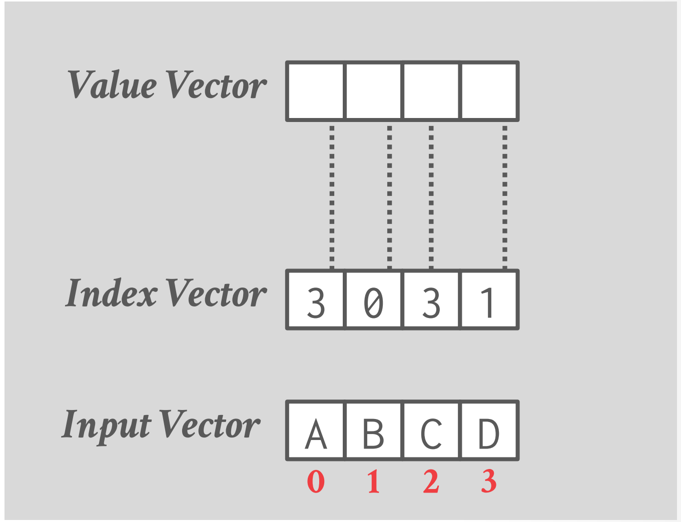


PERMUTE

For each lane, copy values in the **input vector** specified by the offset in the **index vector** into the **destination vector**.

Prior to AVX-512, the DBMS had to write data from the SIMD register to memory then back to the SIMD register.

Permute

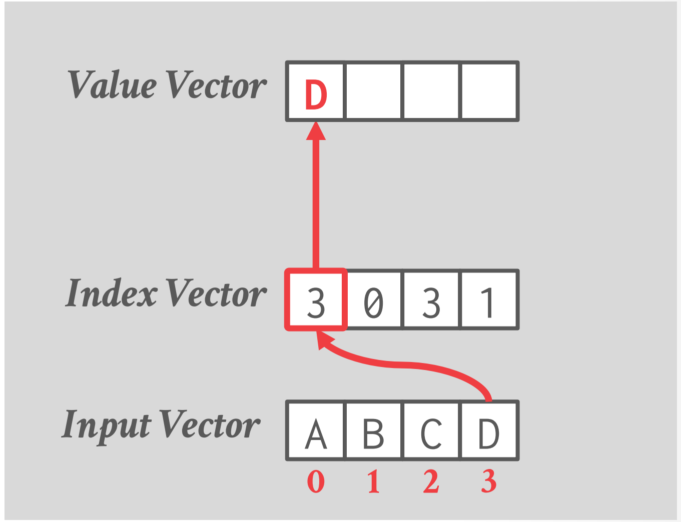


PERMUTE

For each lane, copy values in the **input vector** specified by the offset in the **index vector** into the **destination vector**.

Prior to AVX-512, the DBMS had to write data from the SIMD register to memory then back to the SIMD register.

Permute

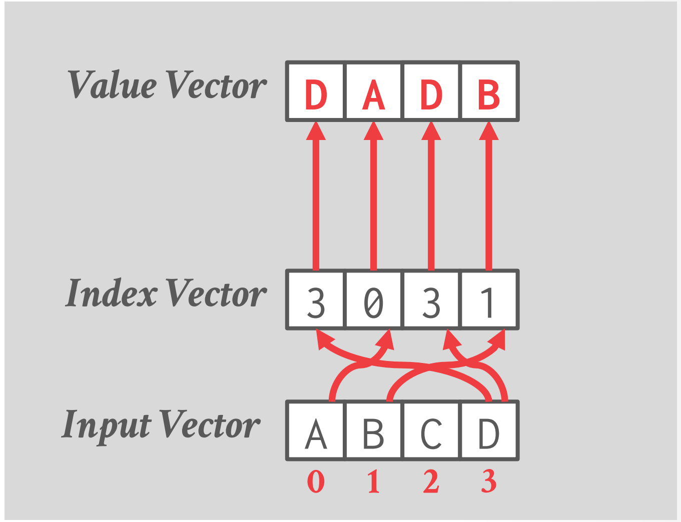


PERMUTE

For each lane, copy values in the **input vector** specified by the offset in the **index vector** into the **destination vector**.

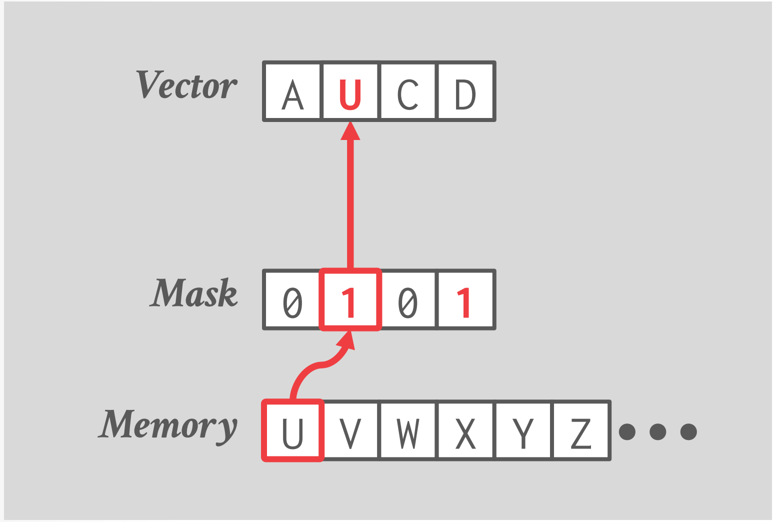
Prior to AVX-512, the DBMS had to write data from the SIMD register to memory then back to the SIMD register.

Permute



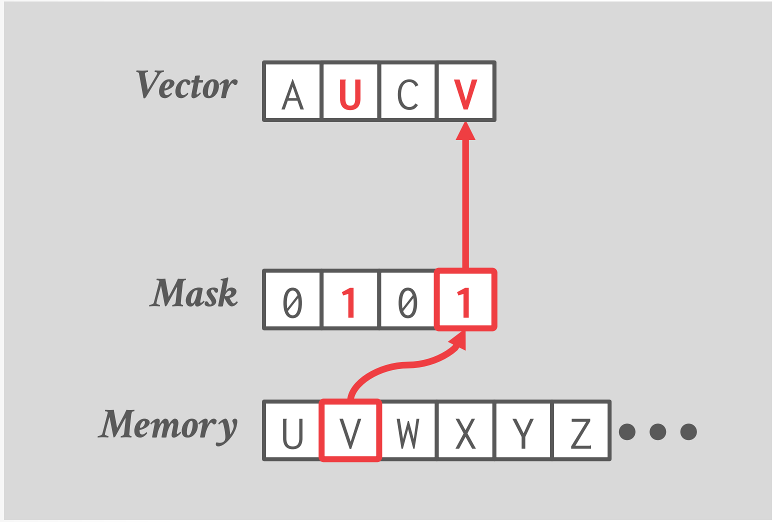
SELECTIVE LOAD/STORE

Selective Load



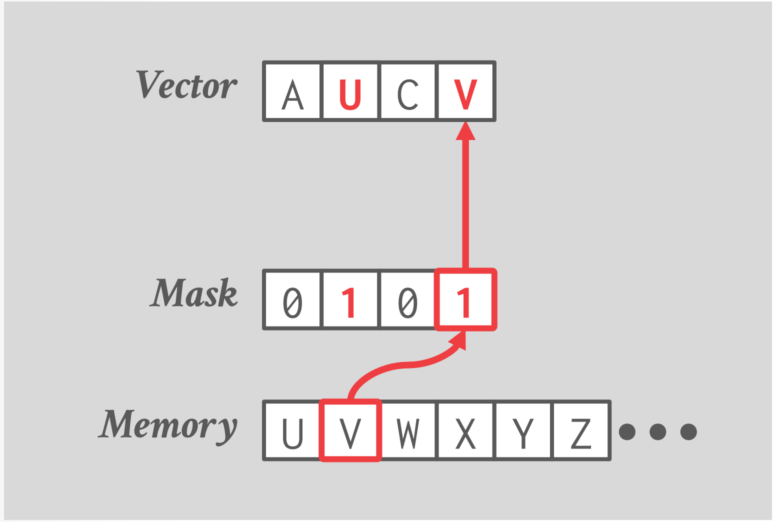
SELECTIVE LOAD/STORE

Selective Load

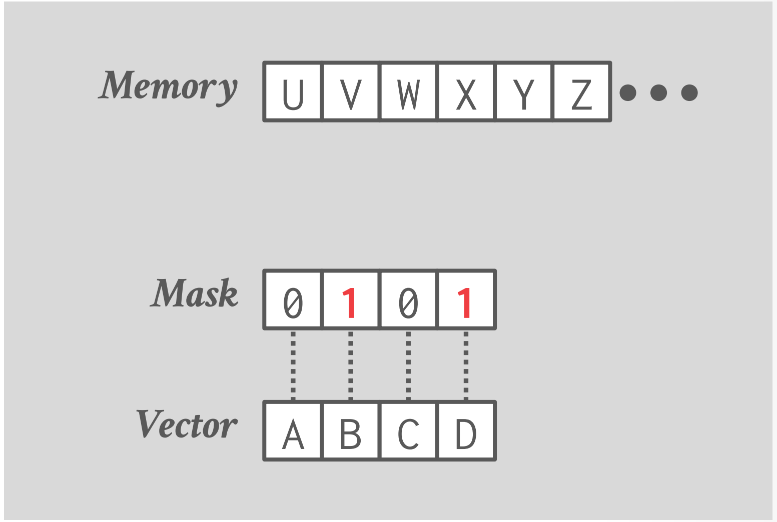


SELECTIVE LOAD/STORE

Selective Load

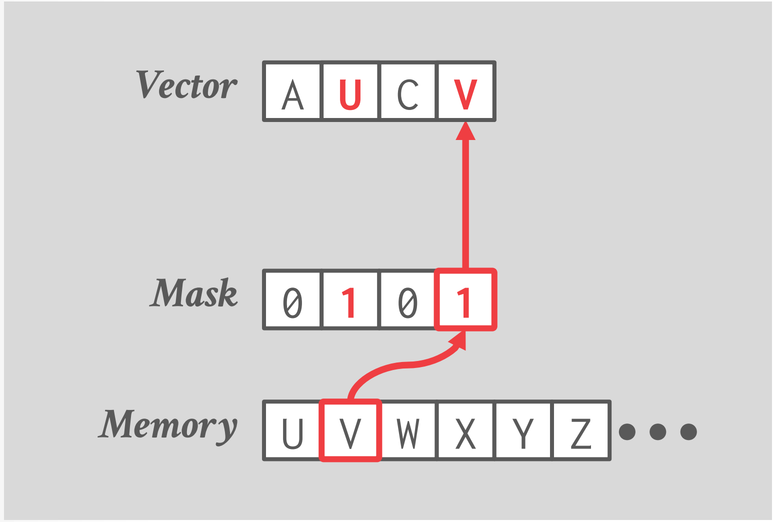


Selective Store

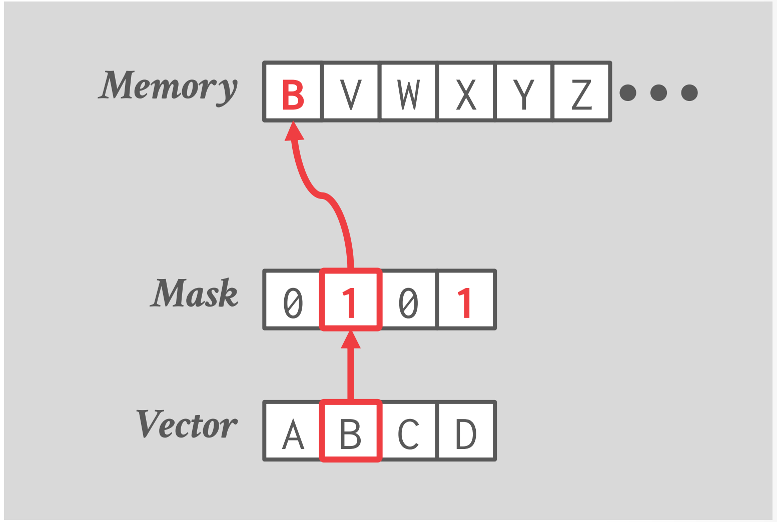


SELECTIVE LOAD/STORE

Selective Load

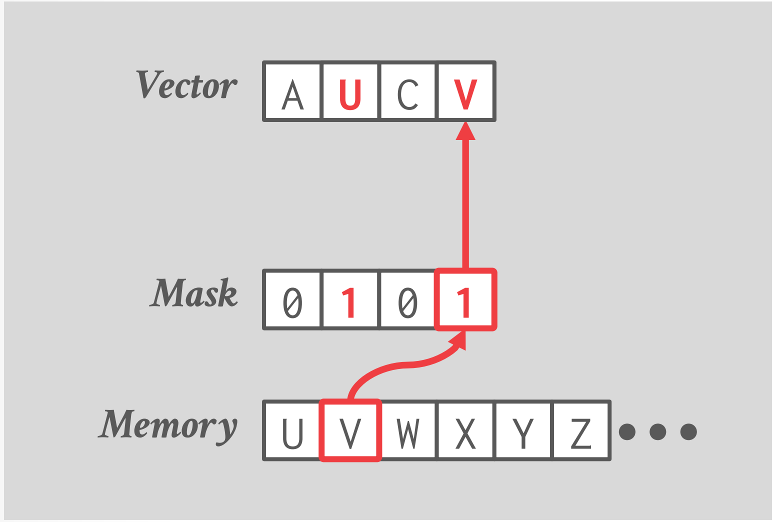


Selective Store

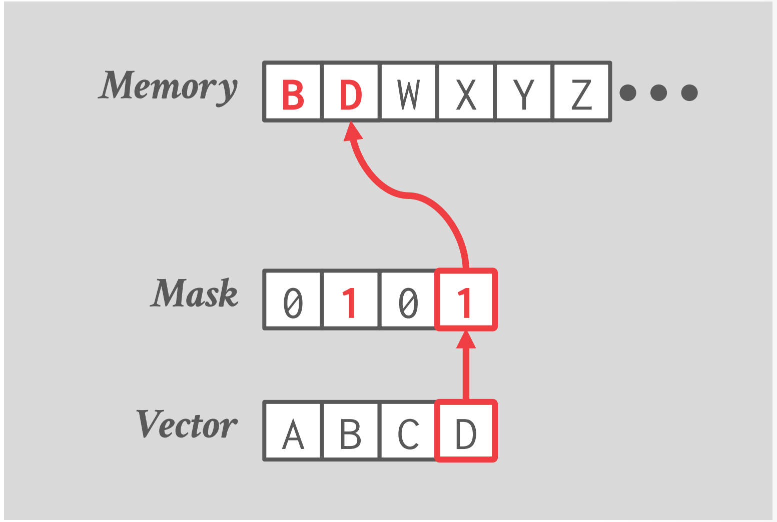


SELECTIVE LOAD/STORE

Selective Load

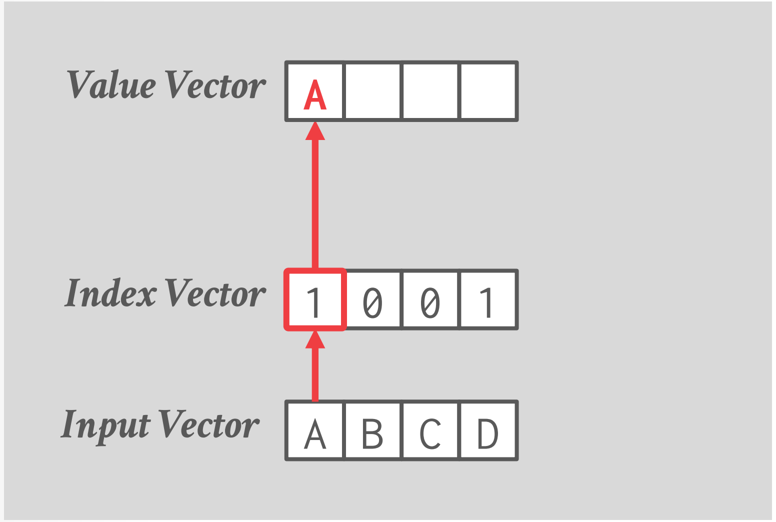


Selective Store



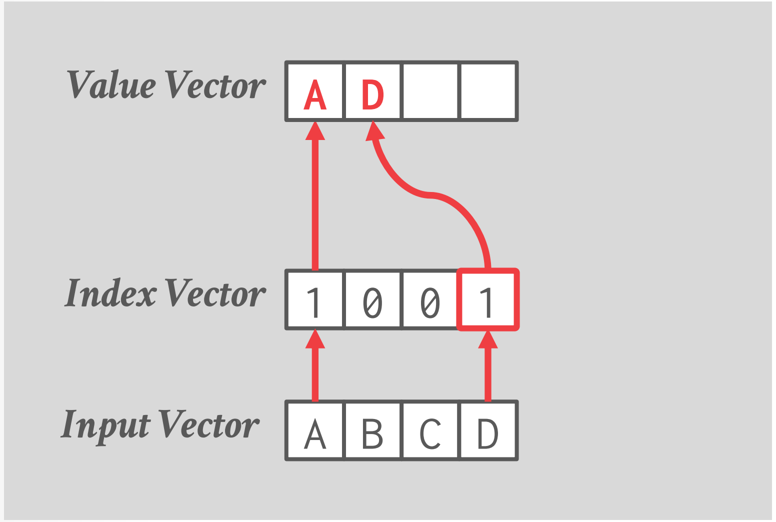
COMPRESS / EXPAND

Compress



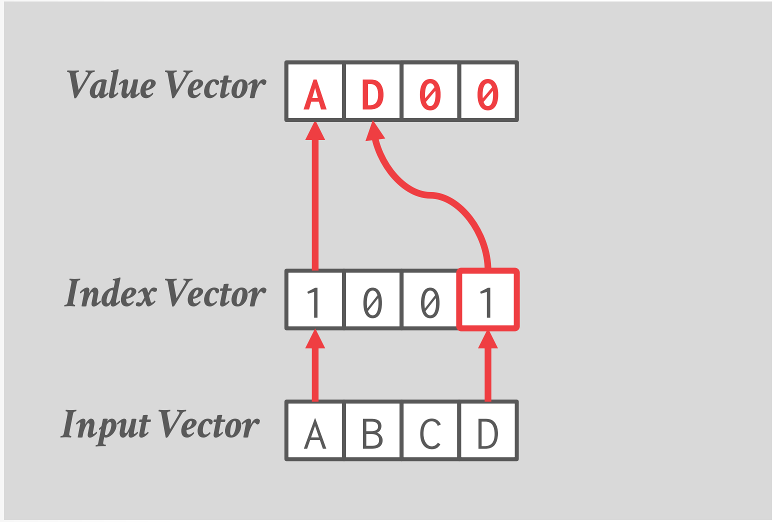
COMPRESS / EXPAND

Compress



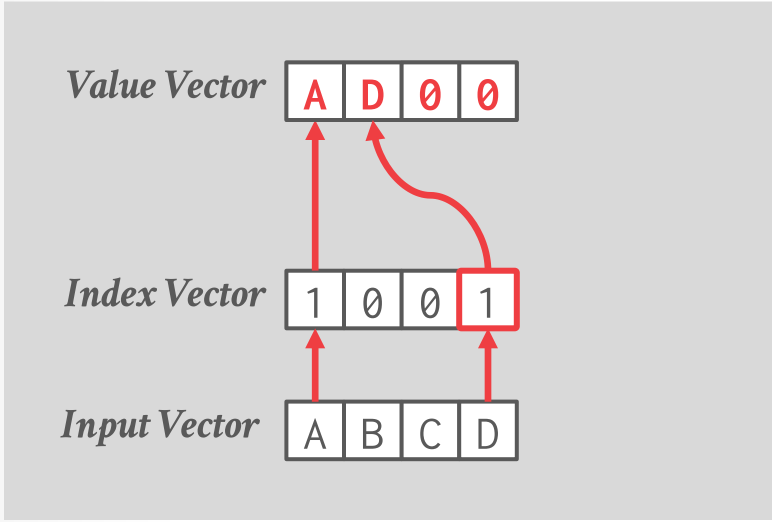
COMPRESS / EXPAND

Compress

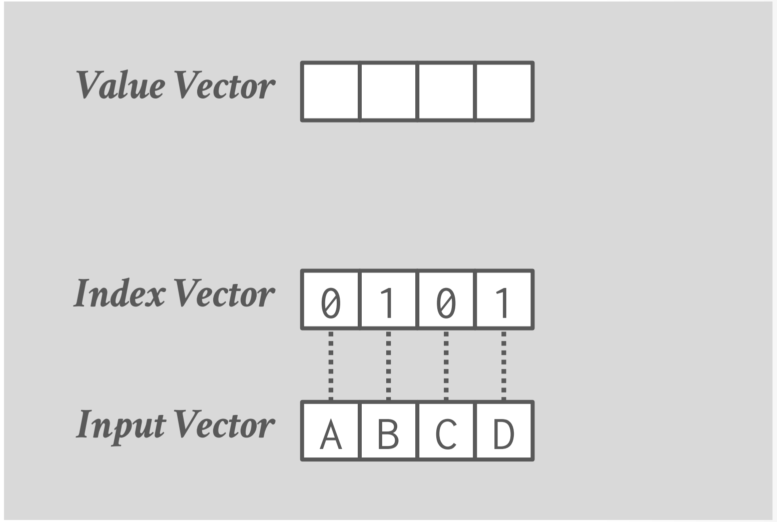


COMPRESS / EXPAND

Compress

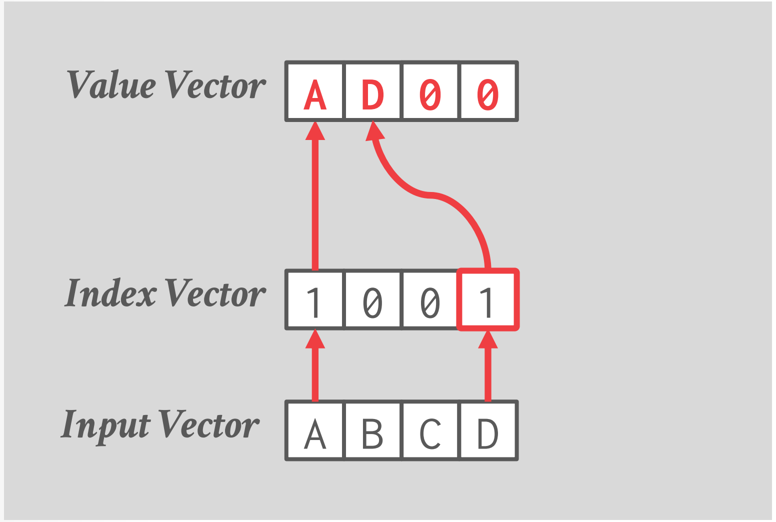


Expand

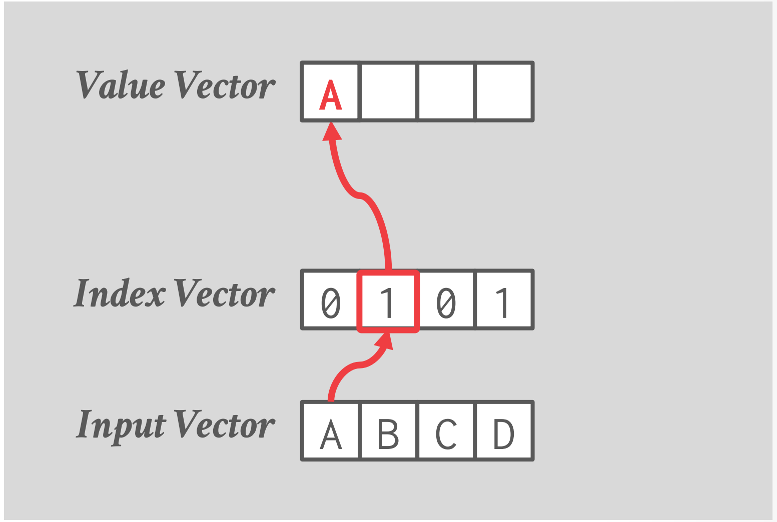


COMPRESS / EXPAND

Compress

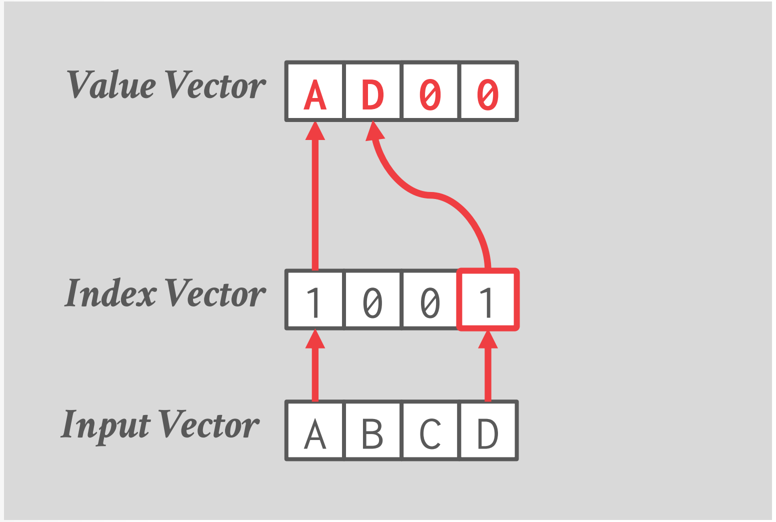


Expand

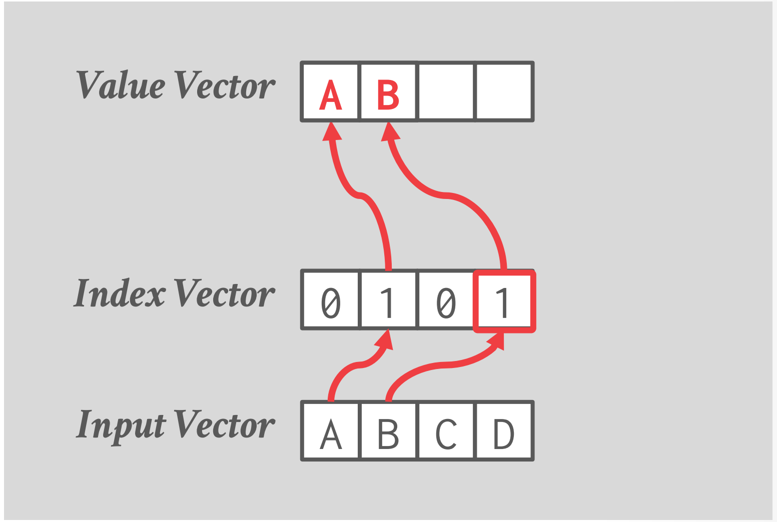


COMPRESS / EXPAND

Compress

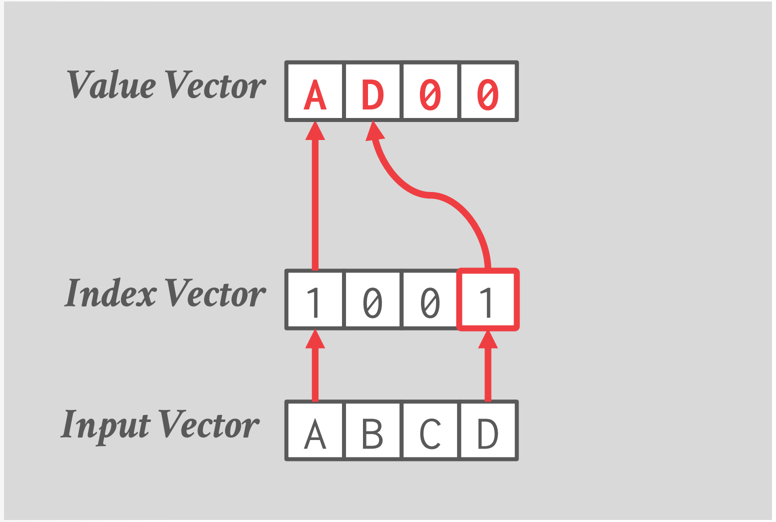


Expand

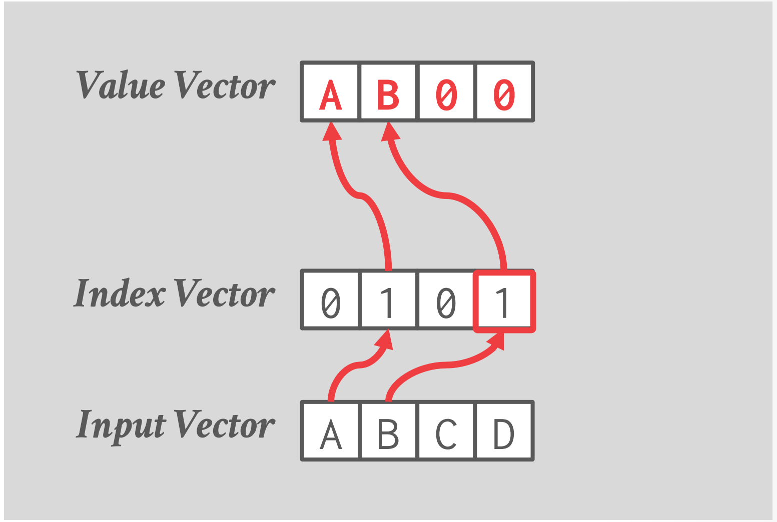


COMPRESS / EXPAND

Compress

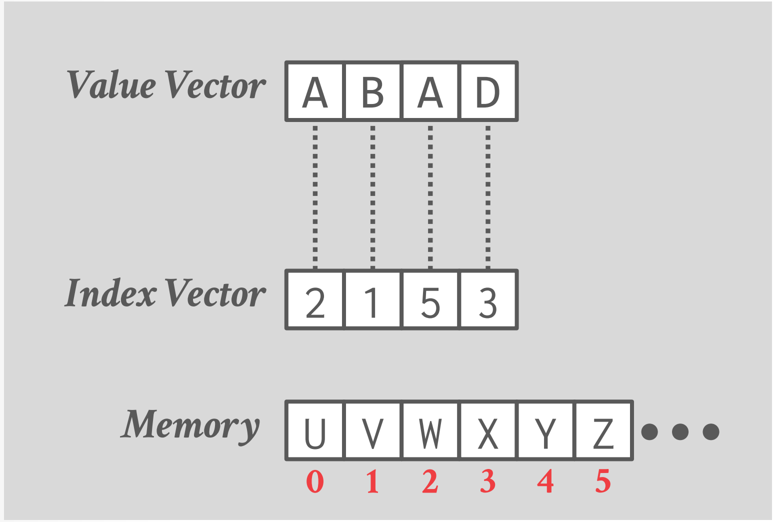


Expand



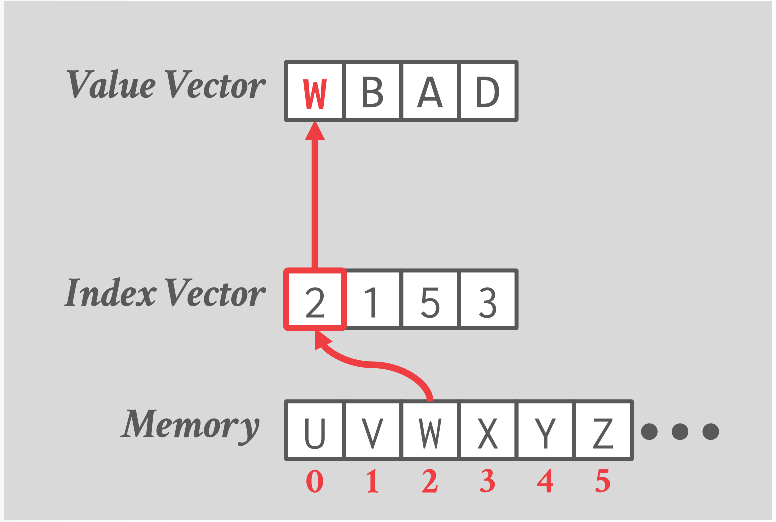
SELECTIVE SCATTER/GATHER

Selective Gather



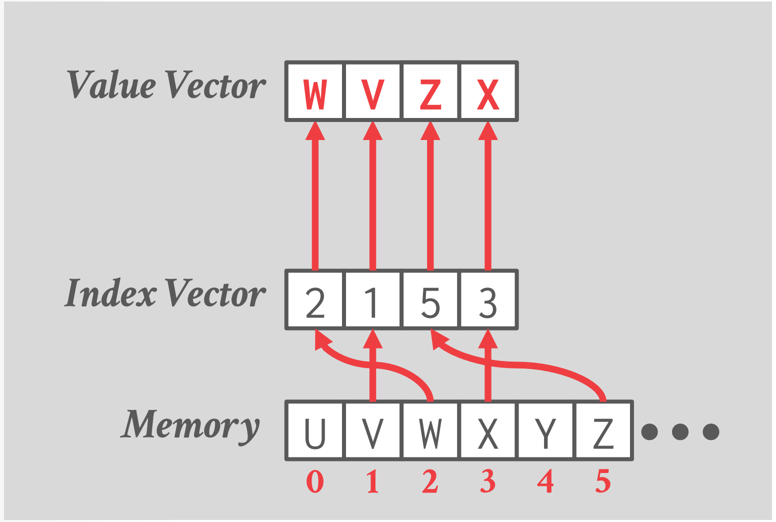
SELECTIVE SCATTER/GATHER

Selective Gather



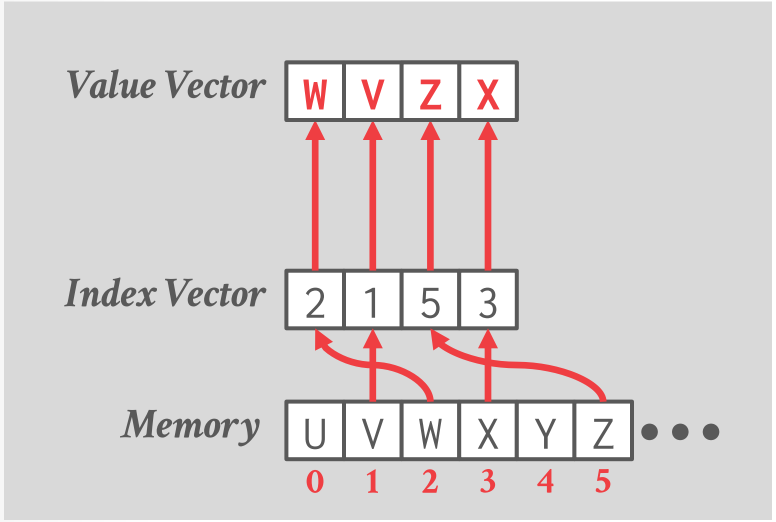
SELECTIVE SCATTER/GATHER

Selective Gather

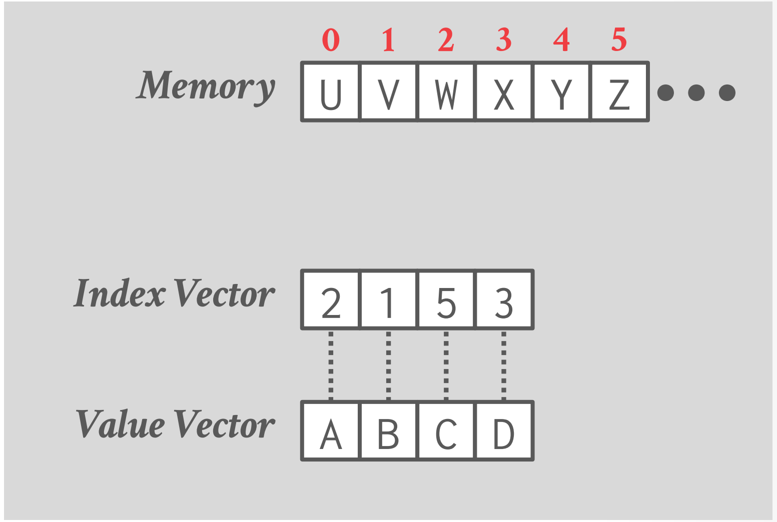


SELECTIVE SCATTER/GATHER

Selective Gather

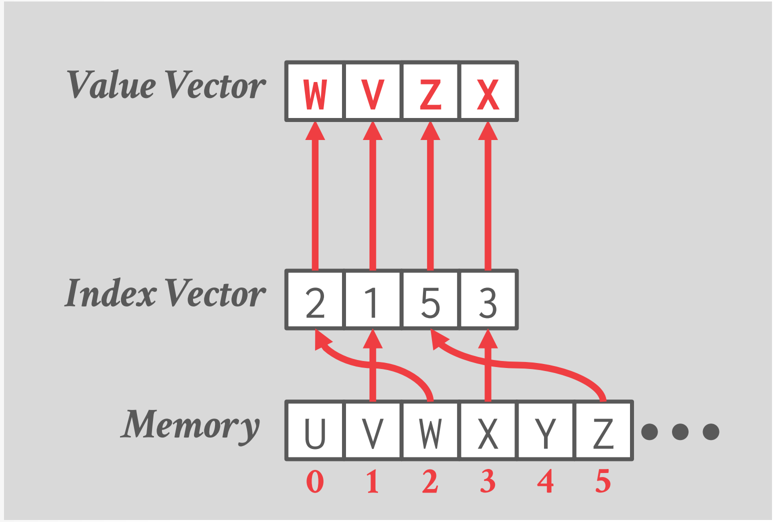


Selective Scatter

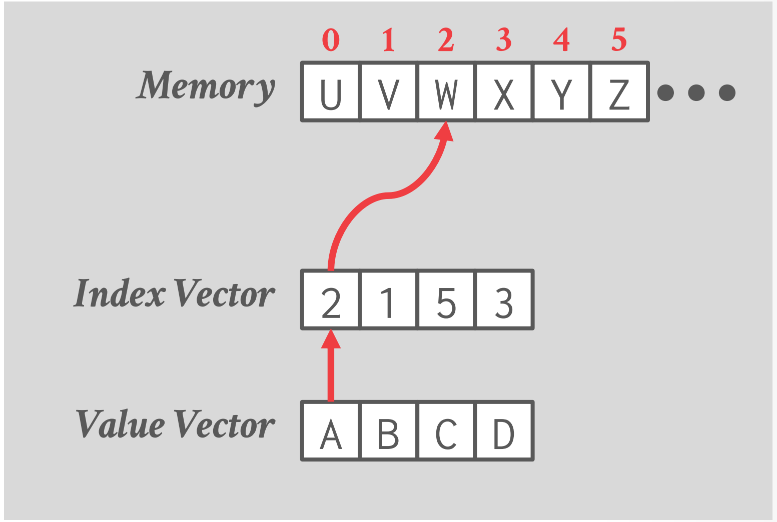


SELECTIVE SCATTER/GATHER

Selective Gather

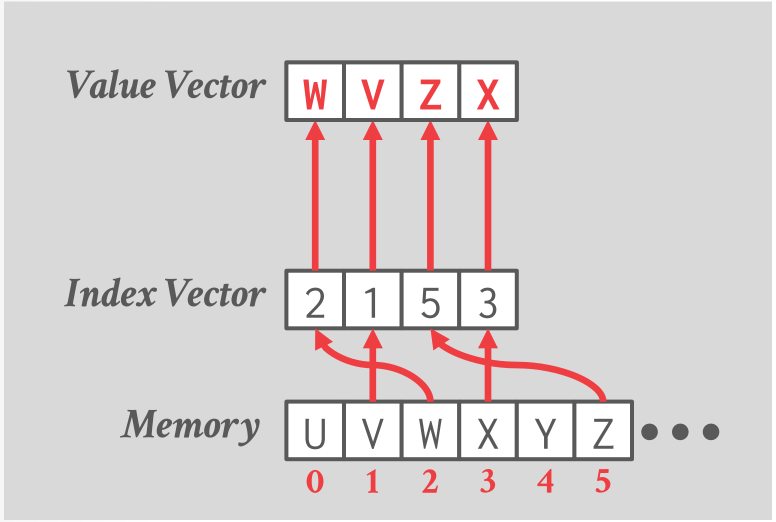


Selective Scatter

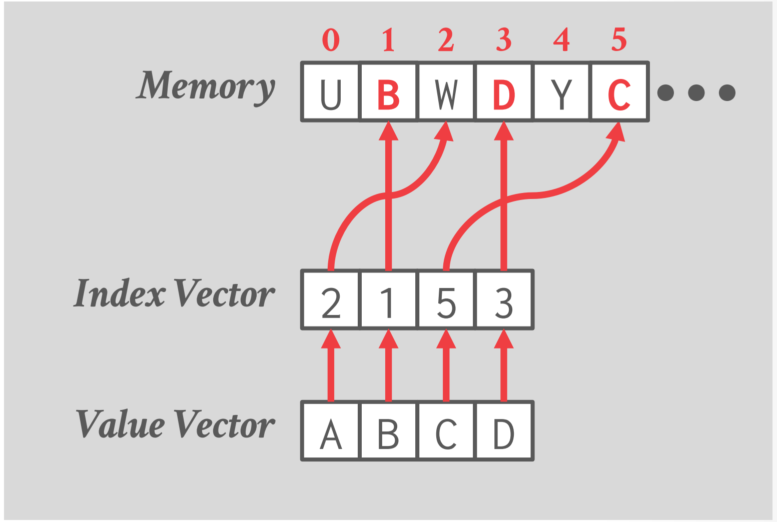


SELECTIVE SCATTER/GATHER

Selective Gather



Selective Scatter



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 unique data items per lane subset (i.e., no useless computations).

RETHINKING SIMD VECTORIZATION FOR IN-MEMORY DATABASES
SIGMOD 2015

VECTORIZED OPERATORS

Selection Scans

Hash Tables

Partitioning / Histograms

RETHINKING SIMD VECTORIZATION FOR IN-MEMORY DATABASES
SIGMOD 2015

SELECTION SCANS

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
```

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

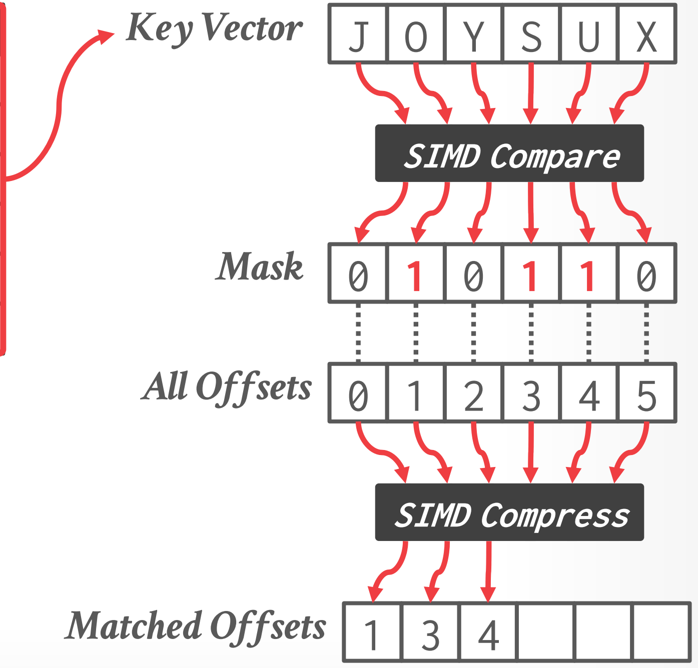
SELECTION SCANS

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)
  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

◆ Scalar (Branching)

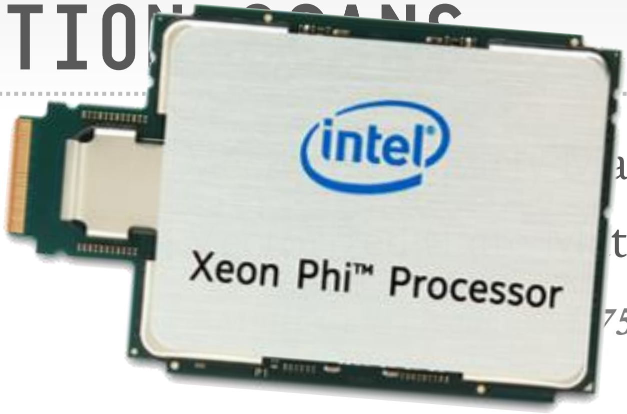
● Scalar (Branchless)

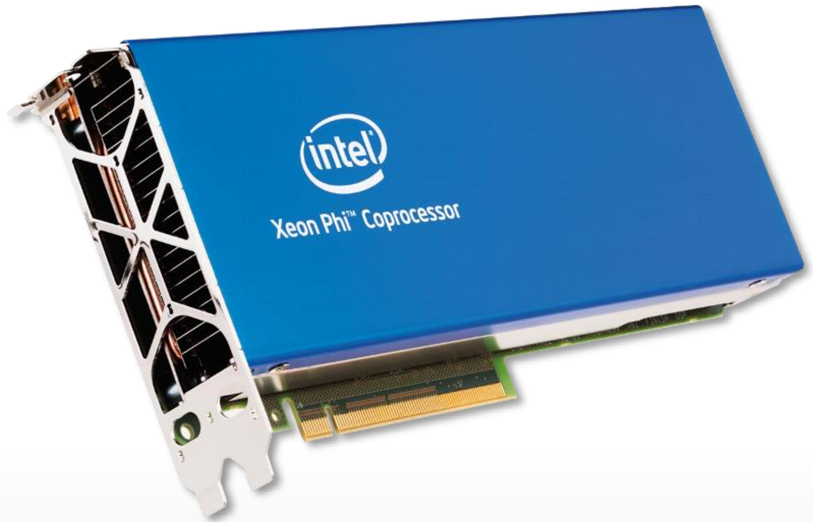
MIC (Xeon Phi 7120P – 61 Cores + 4×HT)

at)

t)

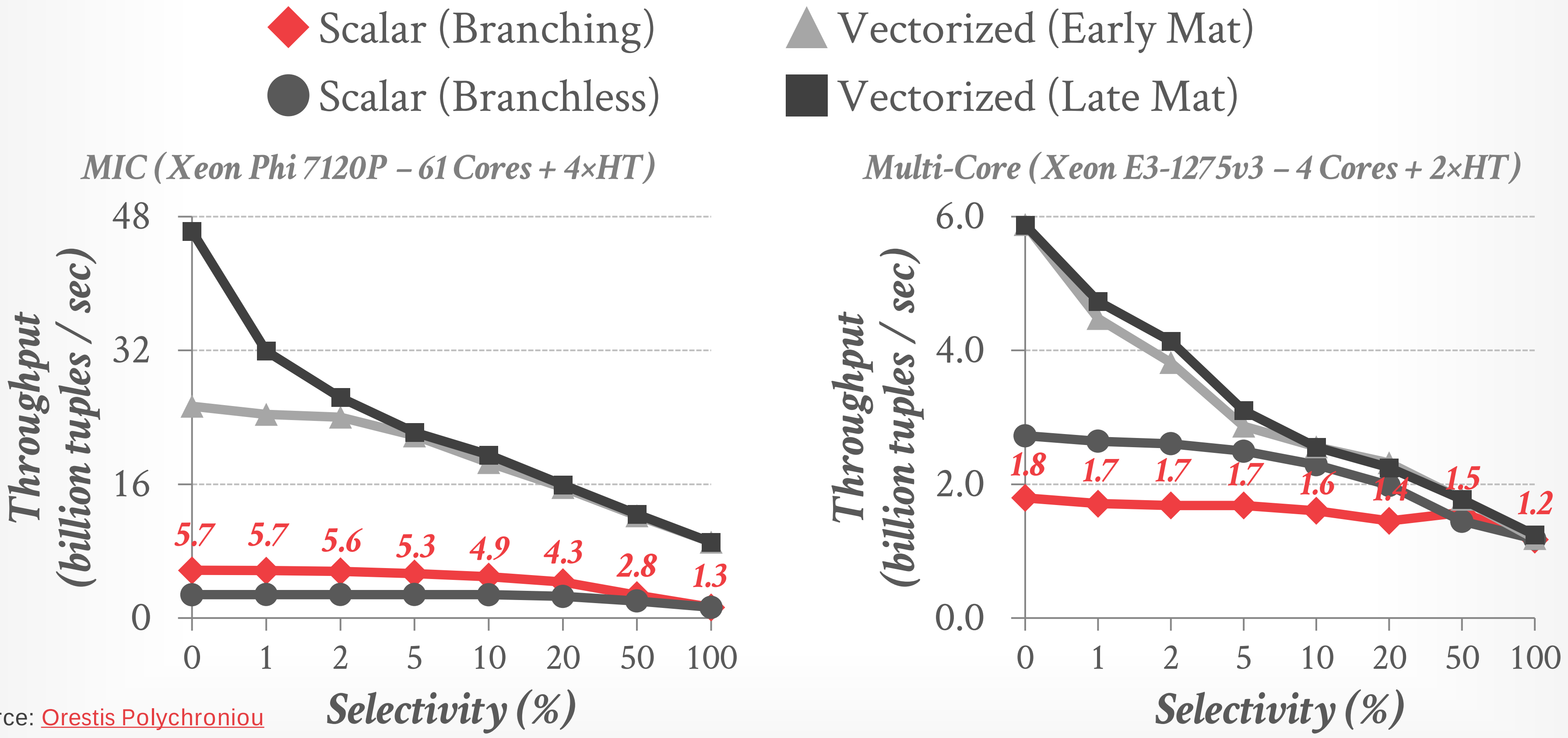
75v3 – 4 Cores + 2×HT)





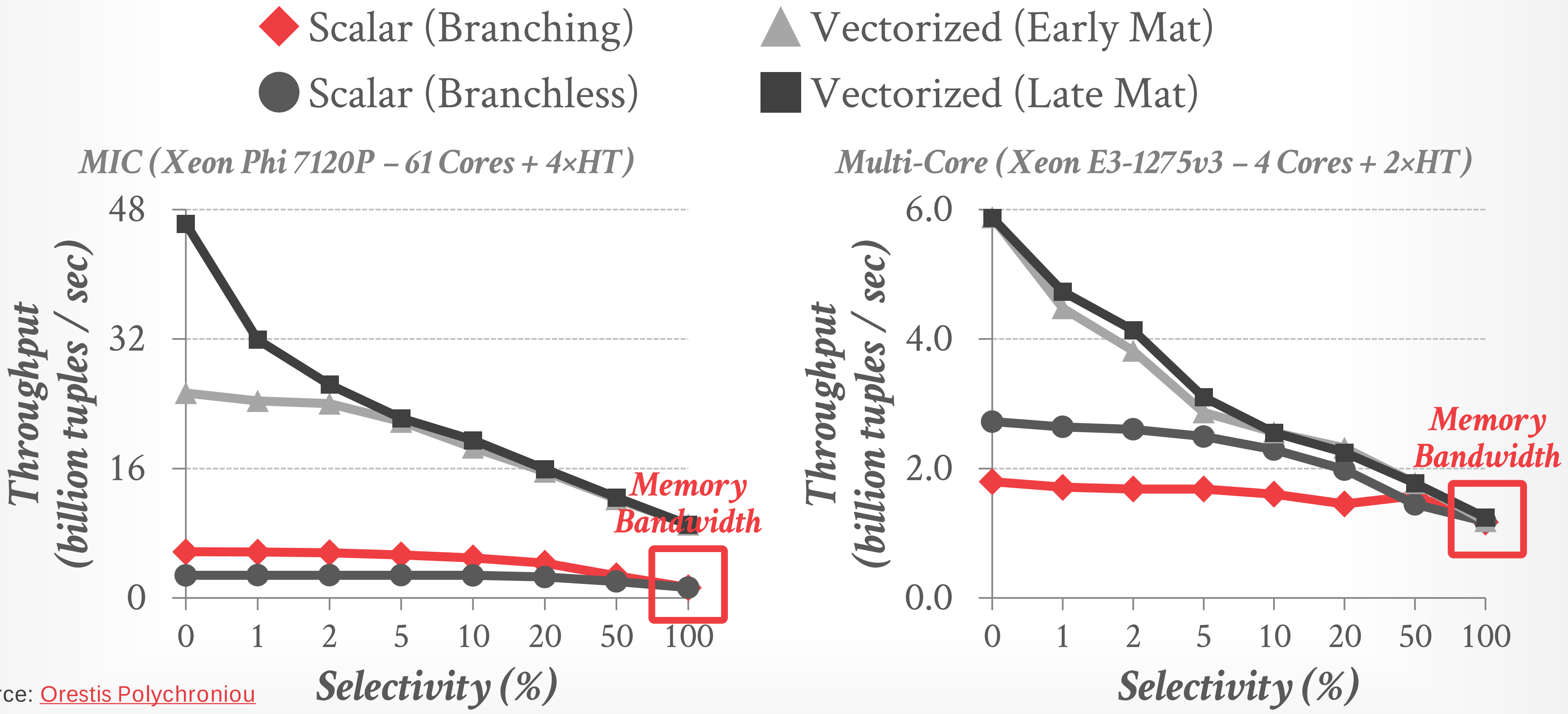
Source: Orestis Polychroniou

SELECTION SCANS



Source: Orestis Polychroniou

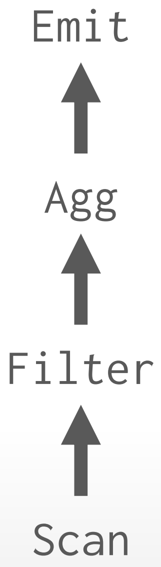
SELECTION SCANS



Source: Orestis Polychroniou

OBSERVATION

For each batch, the SIMD vectors may contain tuples that are no longer valid (they were disqualified by some previous check).

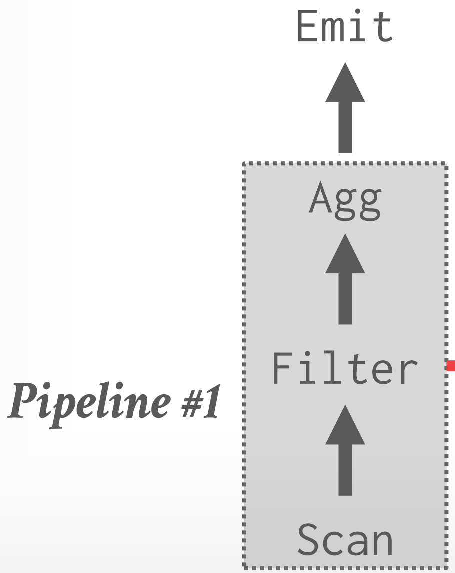


```
SELECT COUNT(*) FROM table
 WHERE age > 20
 GROUP BY city;
```

```
agg = dict()
for t in table:
  if t.age > 20:
    agg[t.city]['count']++
for t in agg:
  emit(t)
```

OBSERVATION

For each batch, the SIMD vectors may contain tuples that are no longer valid (they were disqualified by some previous check).

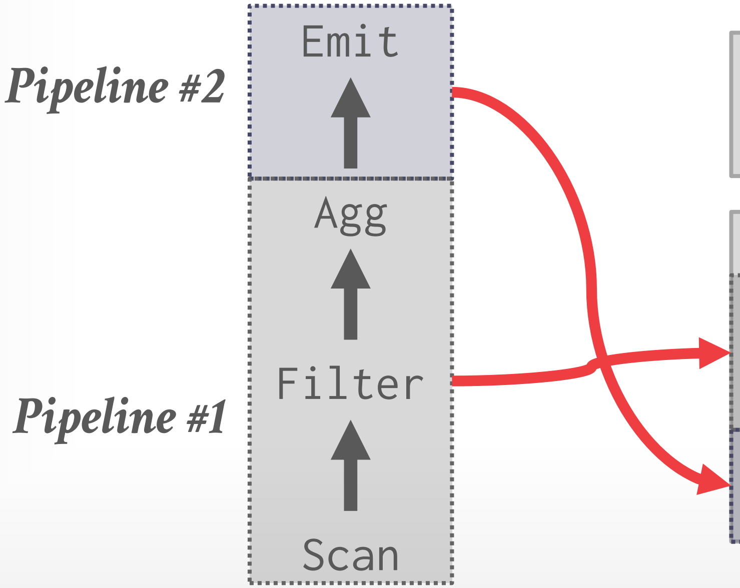


```
SELECT COUNT(*) FROM table
 WHERE age > 20
 GROUP BY city;
```

```
agg = dict()
for t in table:
  if t.age > 20:
    agg[t.city]['count']++
for t in agg:
  emit(t)
```

OBSERVATION

For each batch, the SIMD vectors may contain tuples that are no longer valid (they were disqualified by some previous check).



```
SELECT COUNT(*) FROM table
 WHERE age > 20
 GROUP BY city;
```

```
agg = dict()
for t in table:
  if t.age > 20:
    agg[t.city]['count']++
for t in agg:
  emit(t)
```

RELAXED OPERATOR FUSION

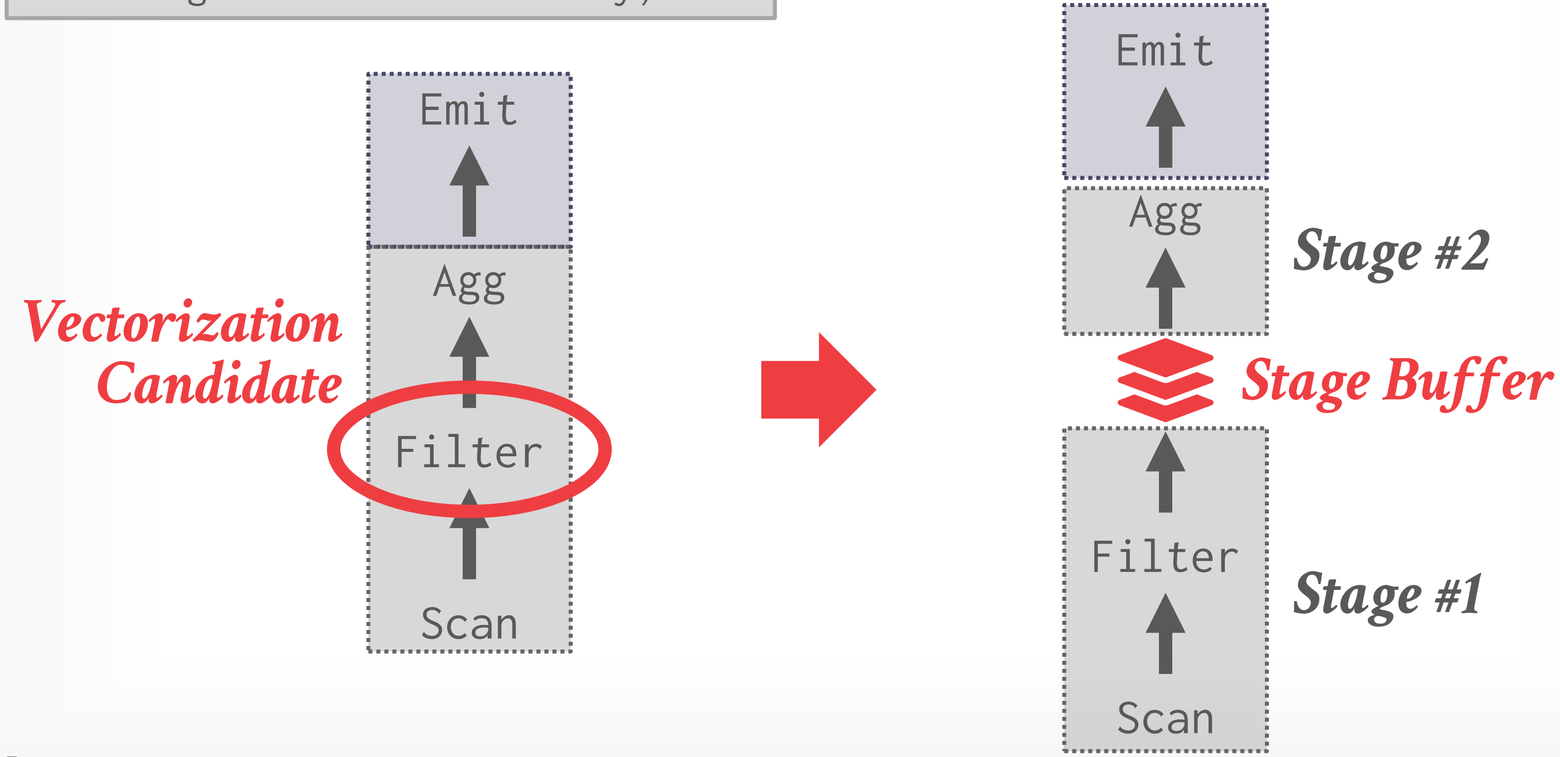
Vectorized processing model designed for query compilation execution engines.

Decompose pipelines into **stages** that operate on vectors of tuples.
→ Each stage may contain multiple operators.
→ Communicate through cache-resident buffers.
→ Stages are granularity of vectorization + fusion.

RELAXED OPERATOR FUSION FOR IN-MEMORY DATABASES: MAKING COMPILATION, VECTORIZATION, AND PREFETCHING WORK TOGETHER AT LAST
VLDB 2017

ROF EXAMPLE

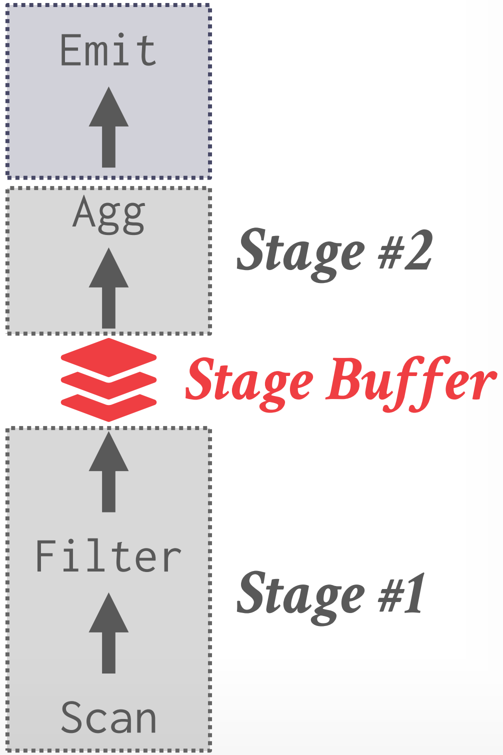
```
SELECT COUNT(*) FROM table
 WHERE age > 20 GROUP BY city;
```



ROF EXAMPLE

```
SELECT COUNT(*) FROM table
 WHERE age > 20 GROUP BY city;
```

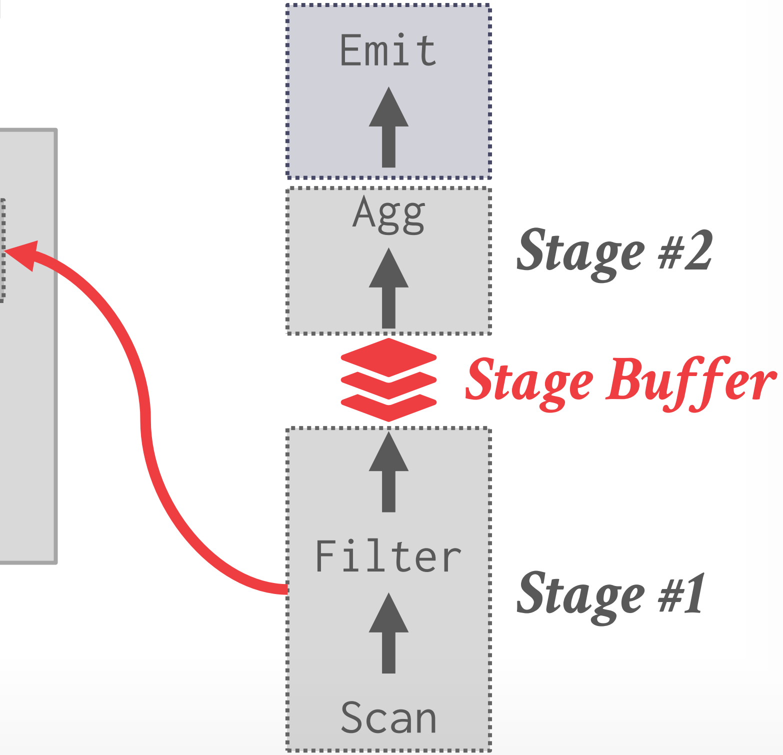
```
agg = dict()
for v_t in table step 1024:
  buffer = simd_cmp_gt(v_t, 20, 1024)
  if |buffer| >= MAX:
    for t in buffer:
      agg[t.city]['count']++
for t in agg:
  emit(t)
```



ROF EXAMPLE

```
SELECT COUNT(*) FROM table
 WHERE age > 20 GROUP BY city;
```

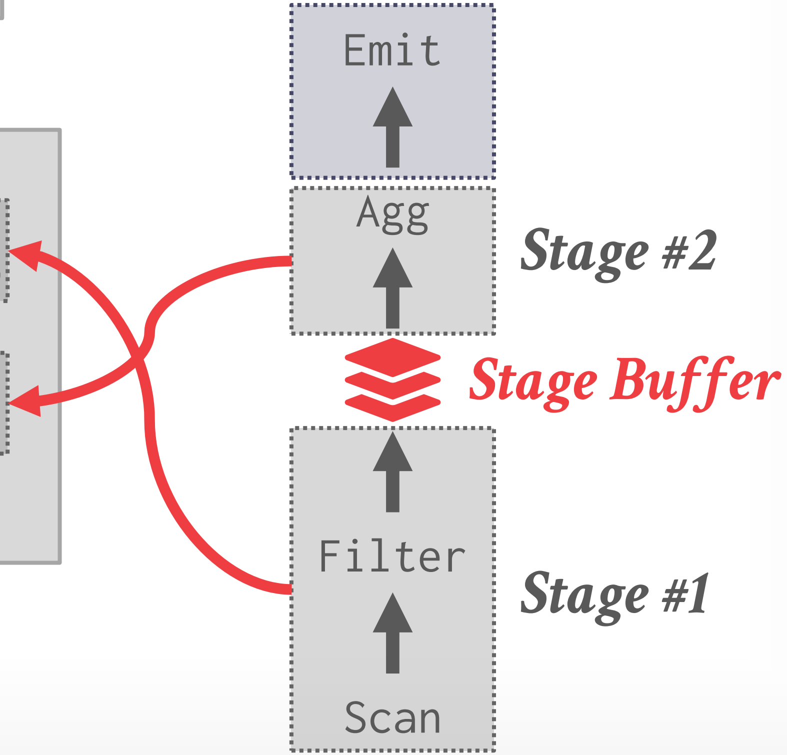
```
agg = dict()
for v_t in table step 1024:
  buffer = simd_cmp_gt(v_t, 20, 1024)
  if |buffer| >= MAX:
    for t in buffer:
      agg[t.city]['count']++
for t in agg:
  emit(t)
```



ROF EXAMPLE

```
SELECT COUNT(*) FROM table
 WHERE age > 20 GROUP BY city;
```

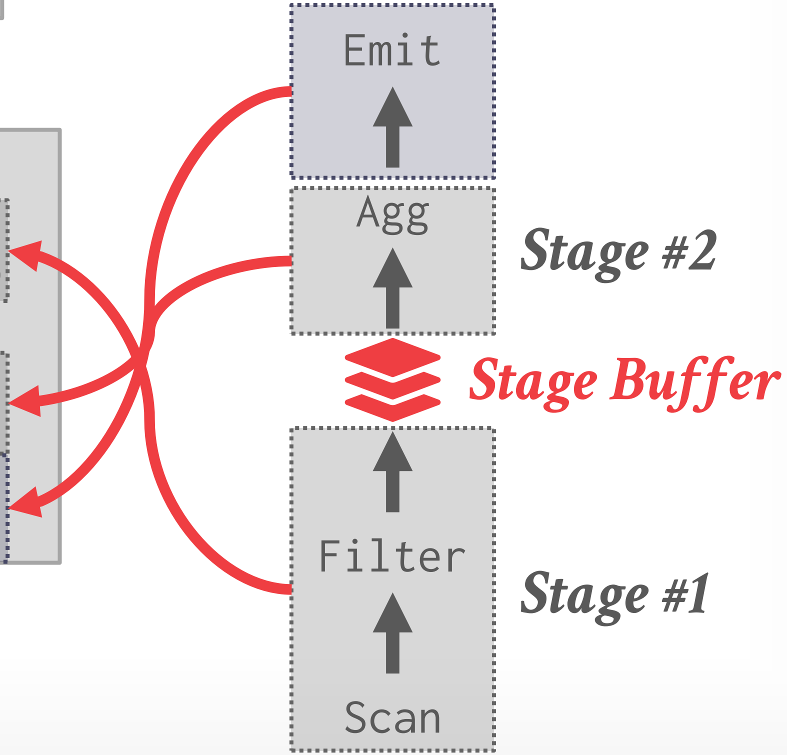
```
agg = dict()
for vt in table step 1024:
  buffer = simd_cmp_gt(vt, 20, 1024)
  if |buffer| >= MAX:
    for t in buffer:
      agg[t.city]['count']++
for t in agg:
  emit(t)
```



ROF EXAMPLE

```
SELECT COUNT(*) FROM table
 WHERE age > 20 GROUP BY city;
```

```
agg = dict()
for v_t in table step 1024:
  buffer = simd_cmp_gt(v_t, 20, 1024)
  if |buffer| >= MAX:
    for t in buffer:
      agg[t.city]['count']++
for t in agg:
  emit(t)
```



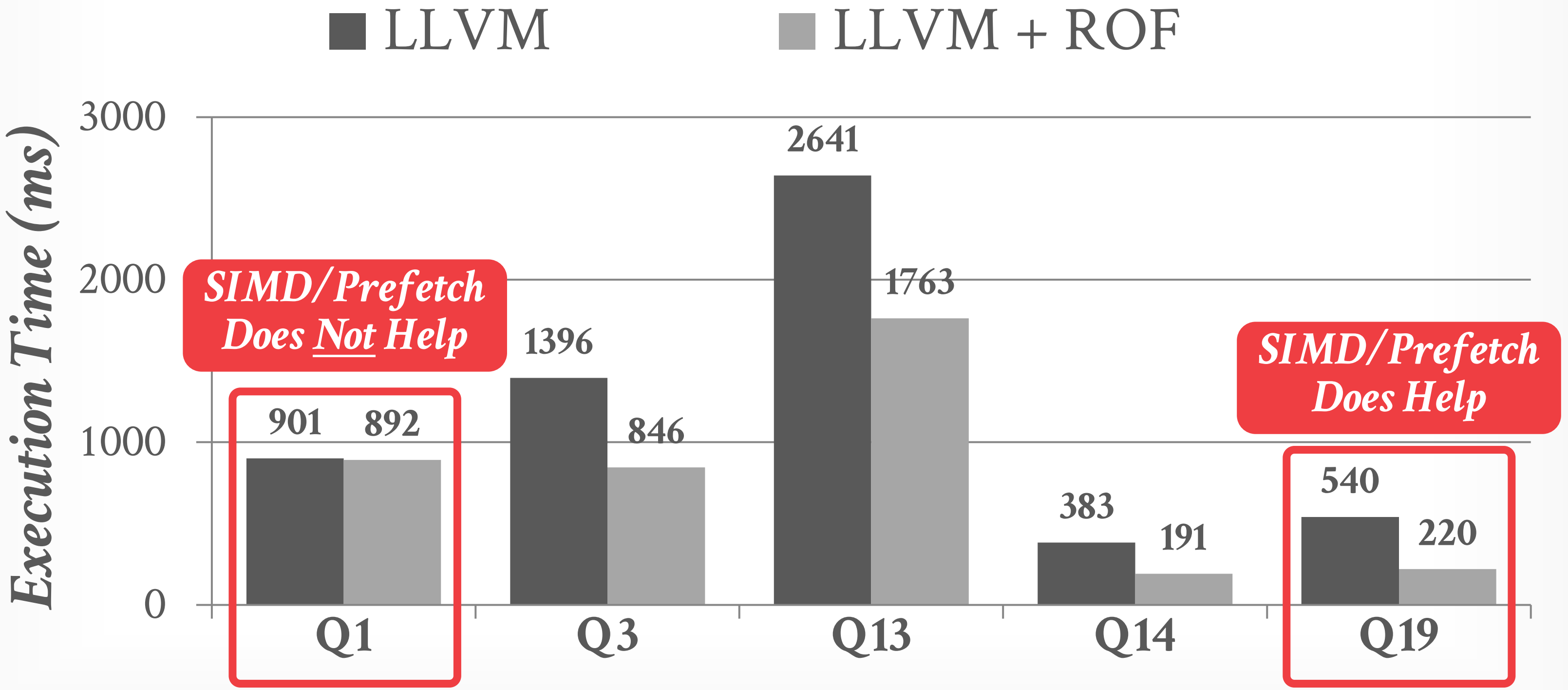
ROF SOFTWARE PREFETCHING

The DBMS can tell the CPU to grab the next vector while it works on the current batch.
→ Prefetch-enabled operators define start of new stage.
→ Hides the cache miss latency.

Any prefetching technique is suitable
→ Group prefetching, software pipelining, AMAC.
→ Group prefetching works and is simple to implement.

ROF EVALUATION

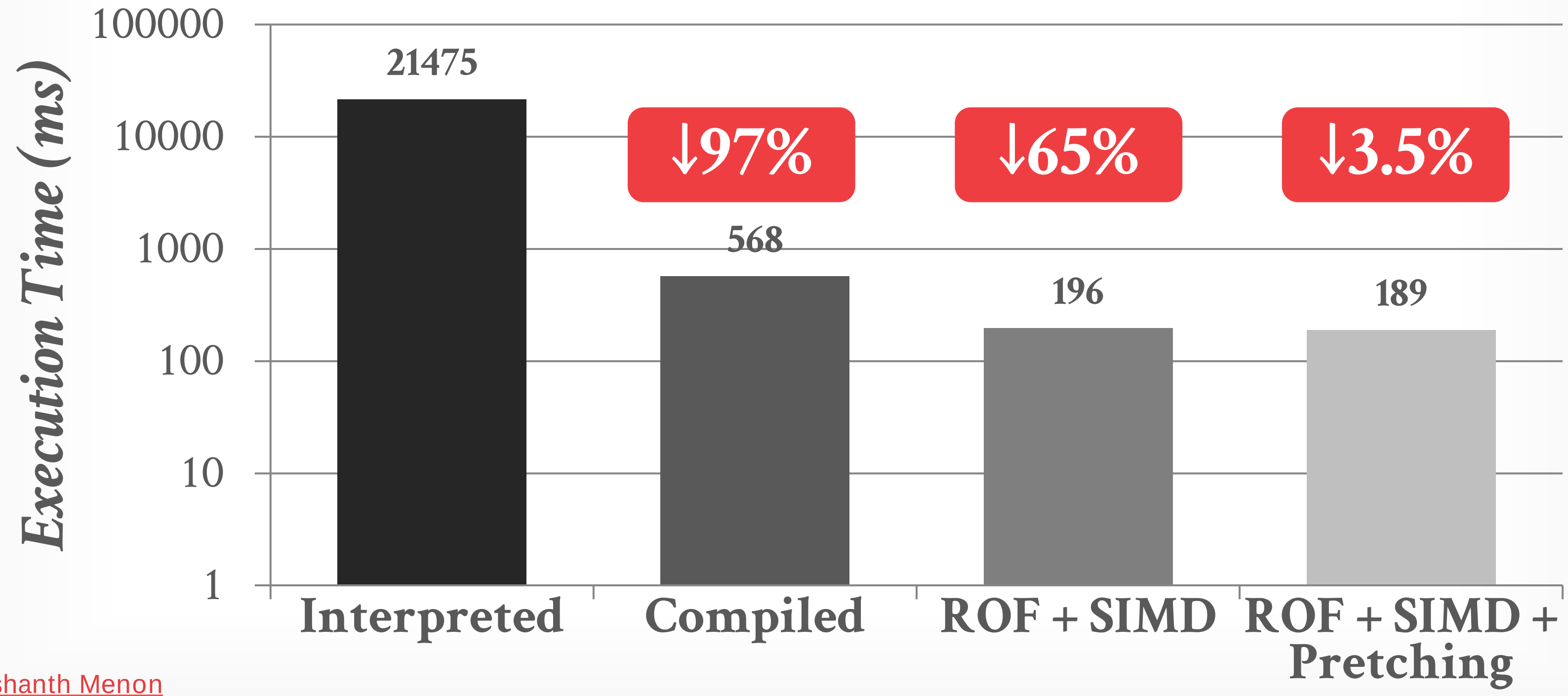
Dual Socket Intel Xeon E5-2630v4 @ 2.20GHz
TPC-H 10 GB Database



Source: Prashanth Menon

ROF EVALUATION – TPC-H Q19

Dual Socket Intel Xeon E5-2630v4 @ 2.20GHz
TPC-H 10 GB Database



Source: Prashanth Menon

VECTORIZED OPERATORS

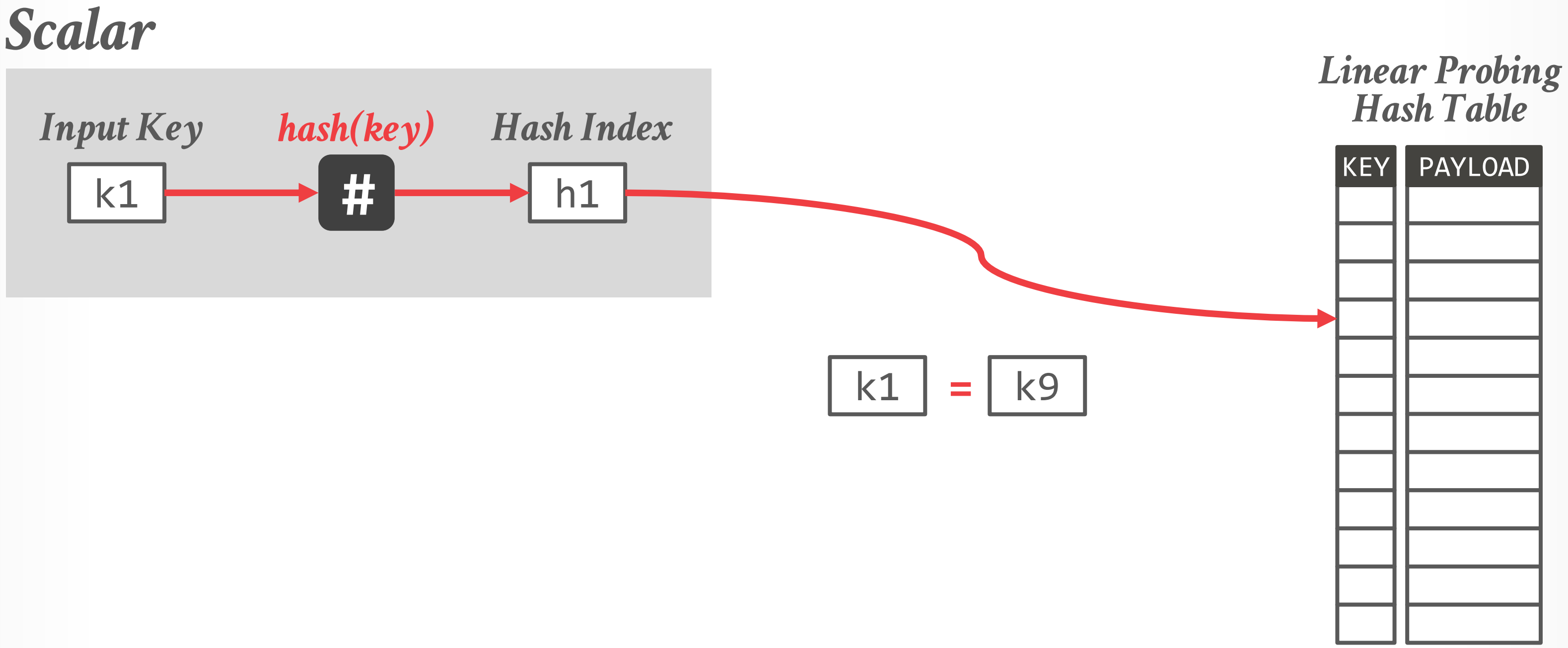
~~Selection Scans~~

Hash Tables

Partitioning / Histograms

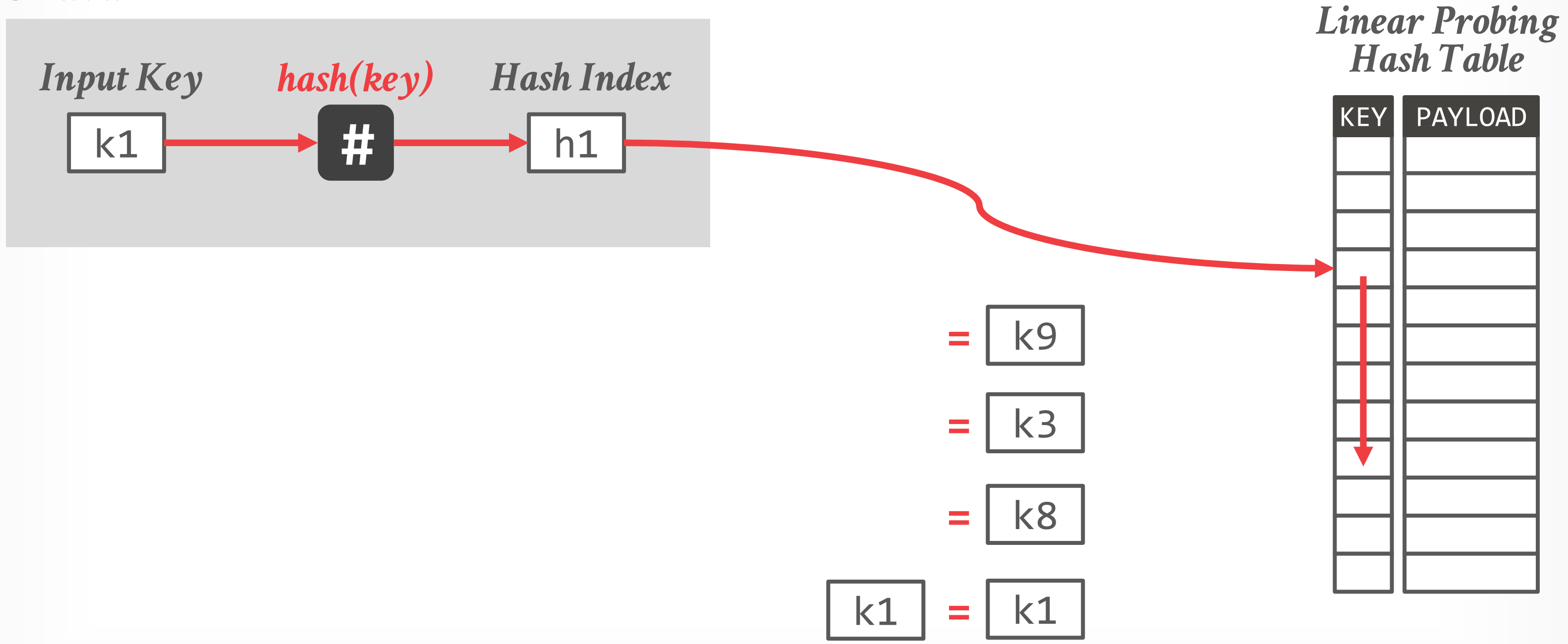
RETHINKING SIMD VECTORIZATION FOR IN-MEMORY DATABASES
SIGMOD 2015

HASH TABLES – PROBING



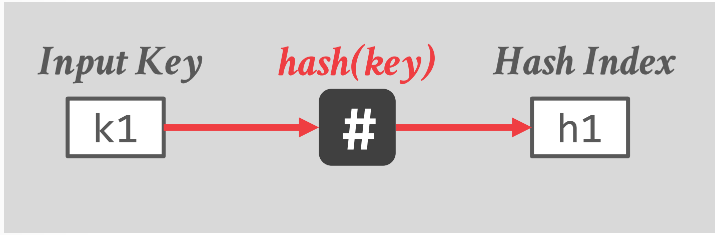
HASH TABLES – PROBING

Scalar

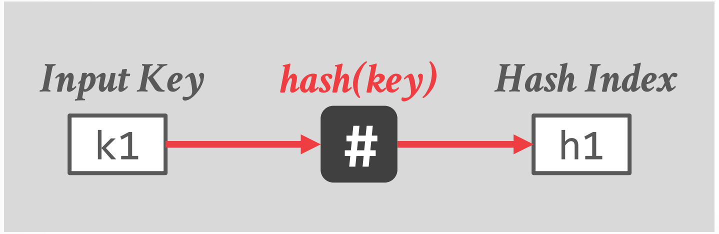


HASH TABLES – PROBING

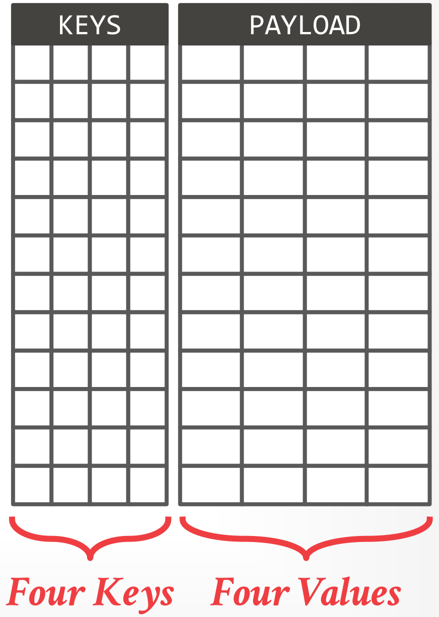
Scalar



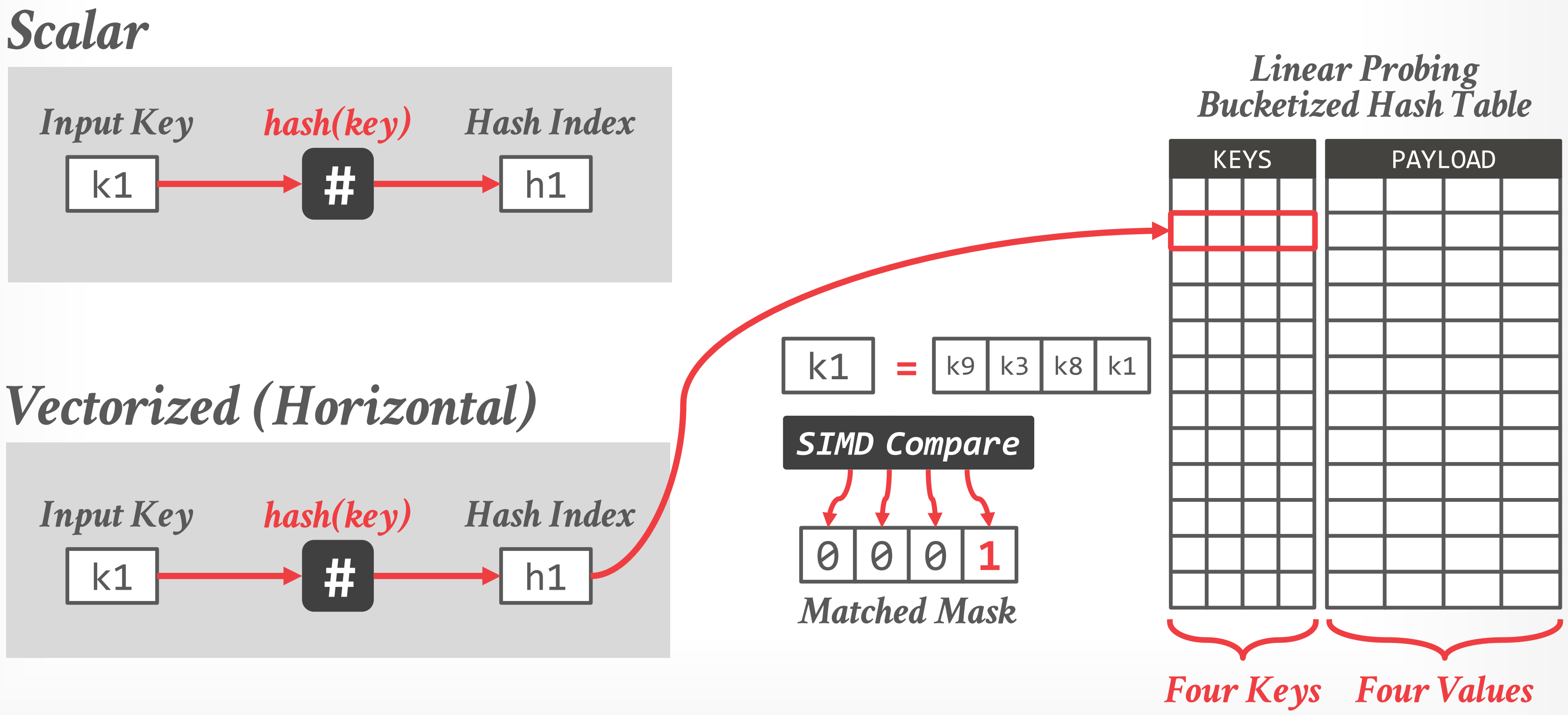
Vectorized (Horizontal)



Linear Probing Bucketized Hash Table

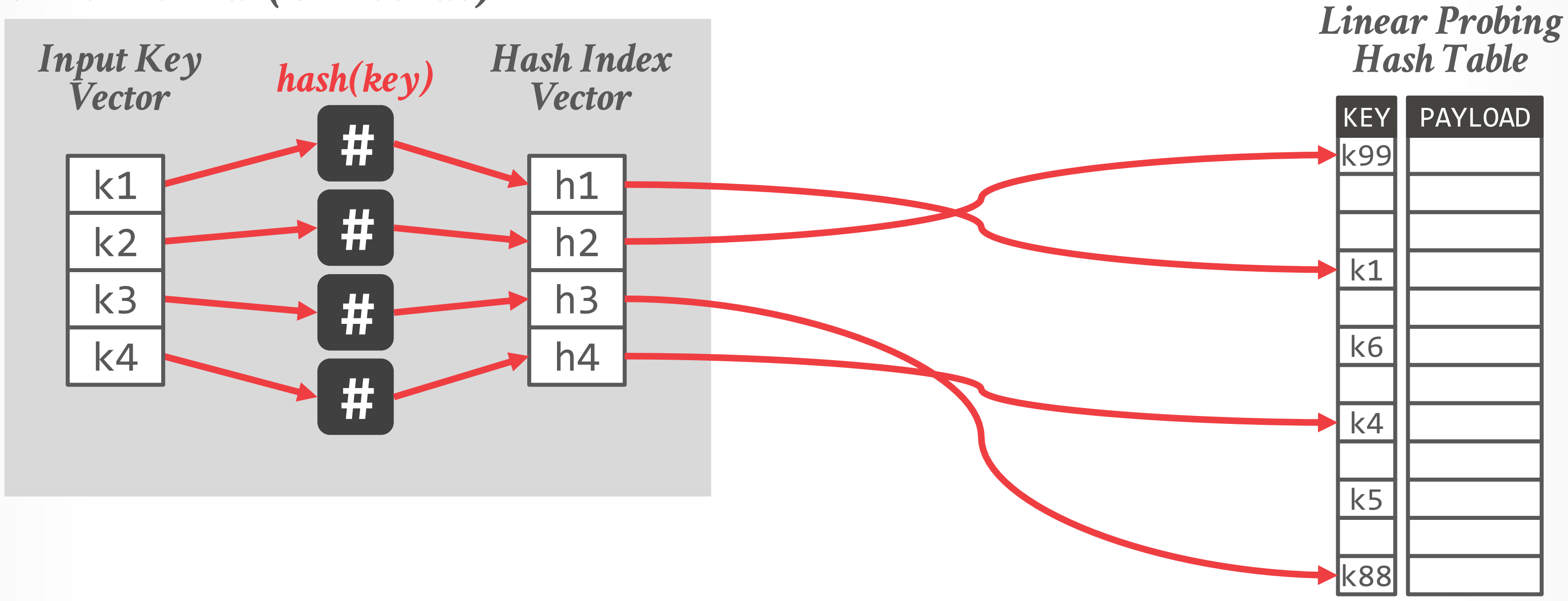


HASH TABLES – PROBING



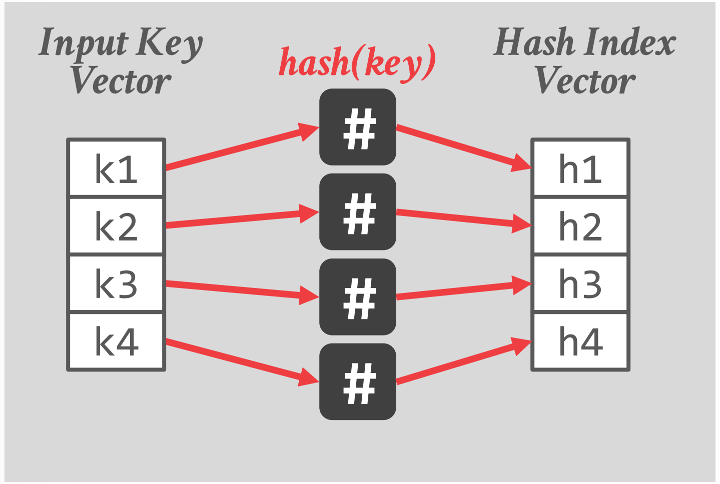
HASH TABLES – PROBING

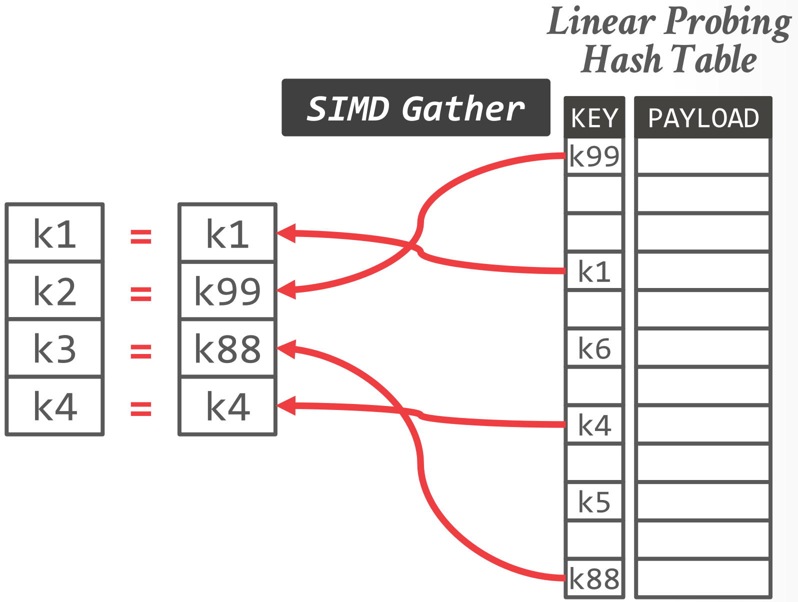
Vectorized (Vertical)



HASH TABLES – PROBING

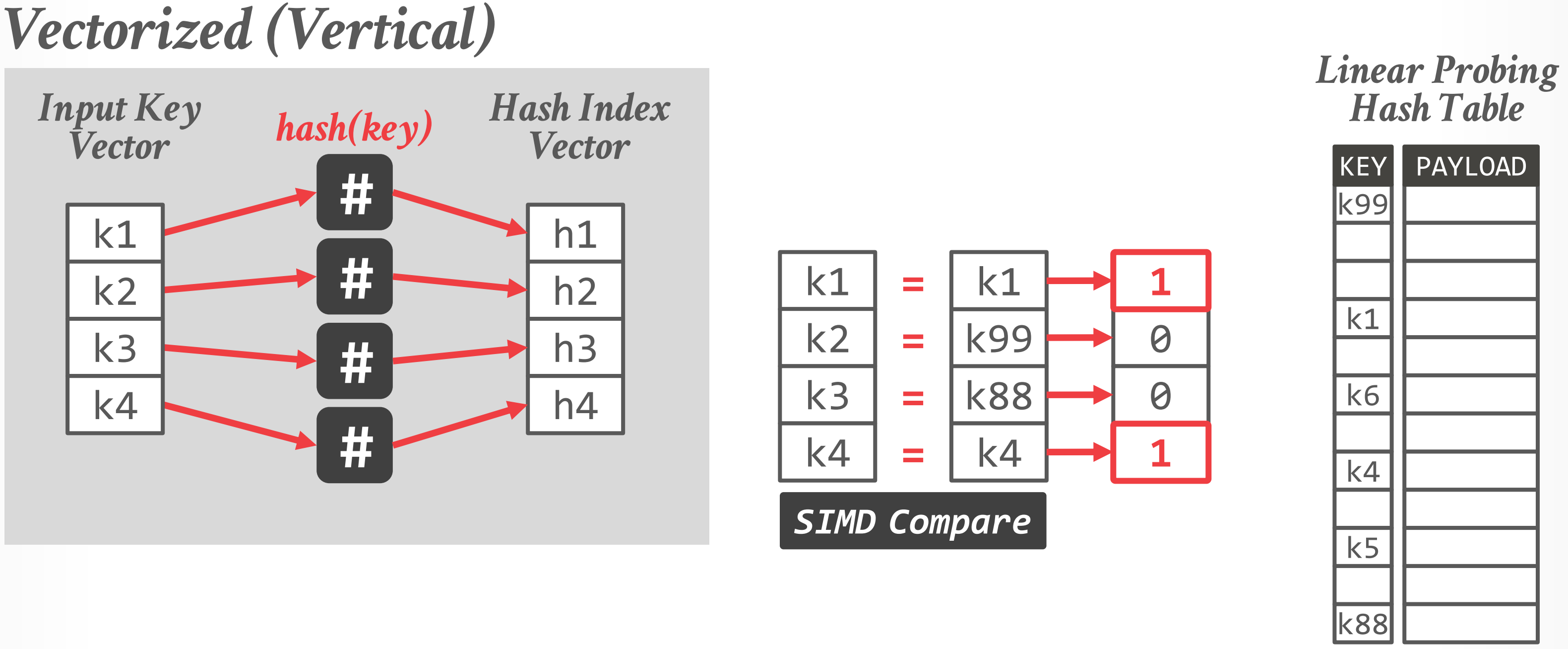
Vectorized (Vertical)





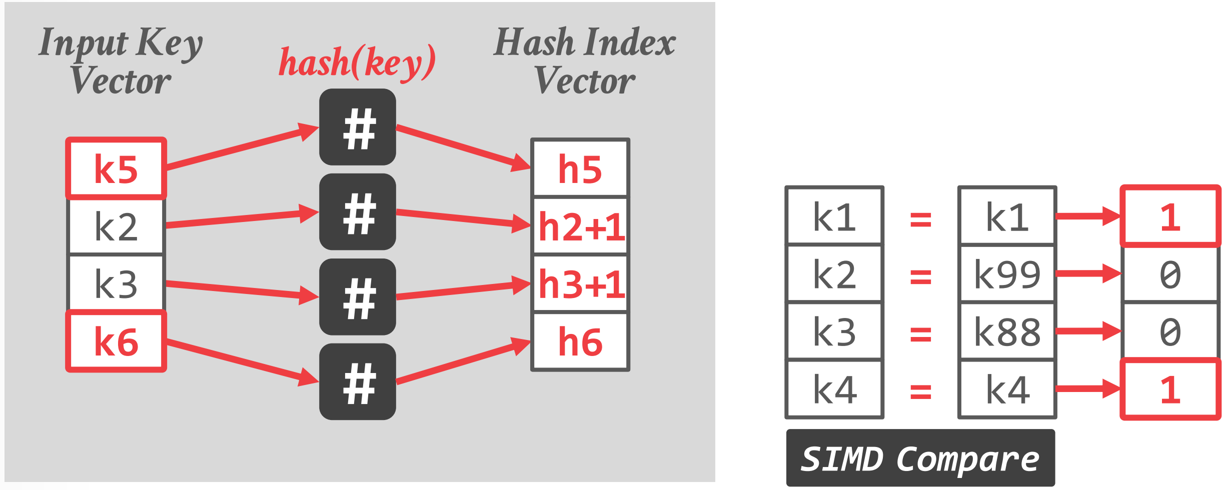
HASH TABLES – PROBING

Vectorized (Vertical)

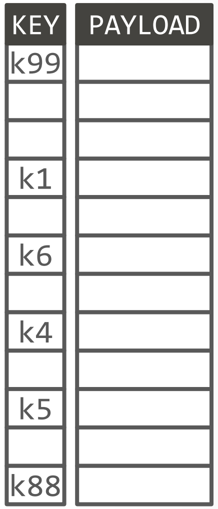


HASH TABLES – PROBING

Vectorized (Vertical)



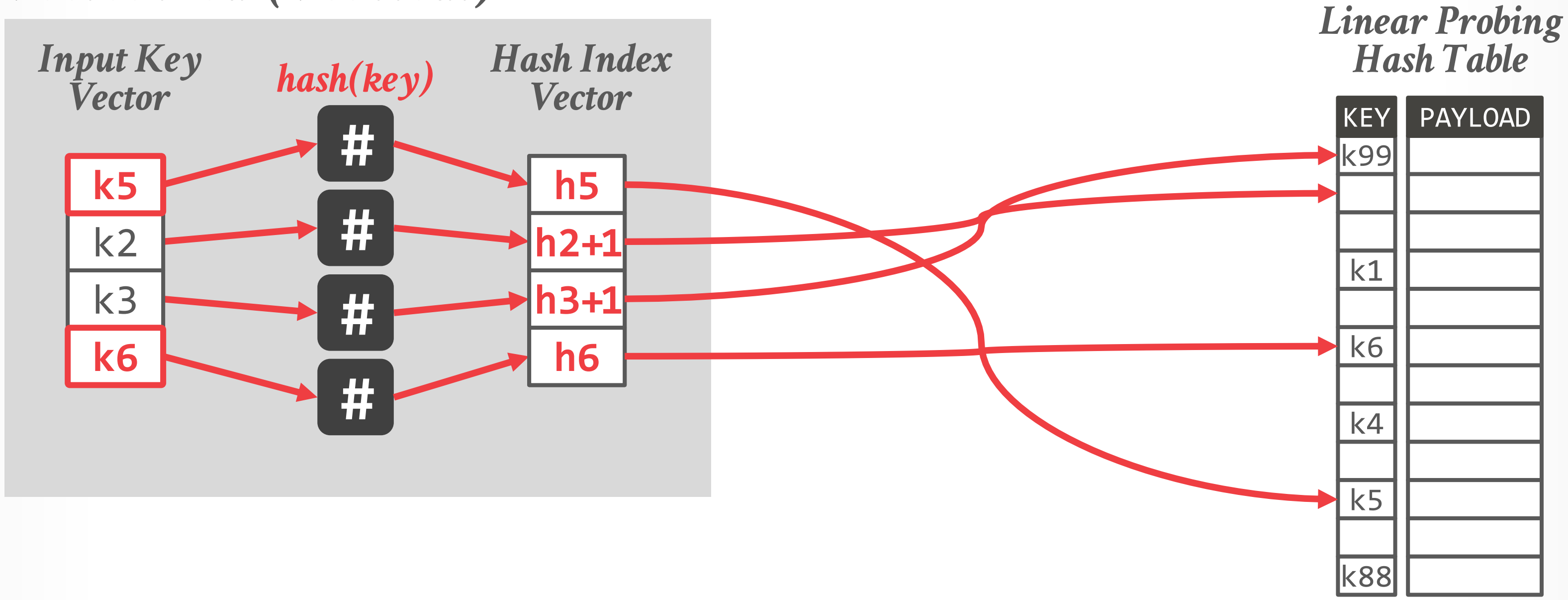
Linear Probing Hash Table



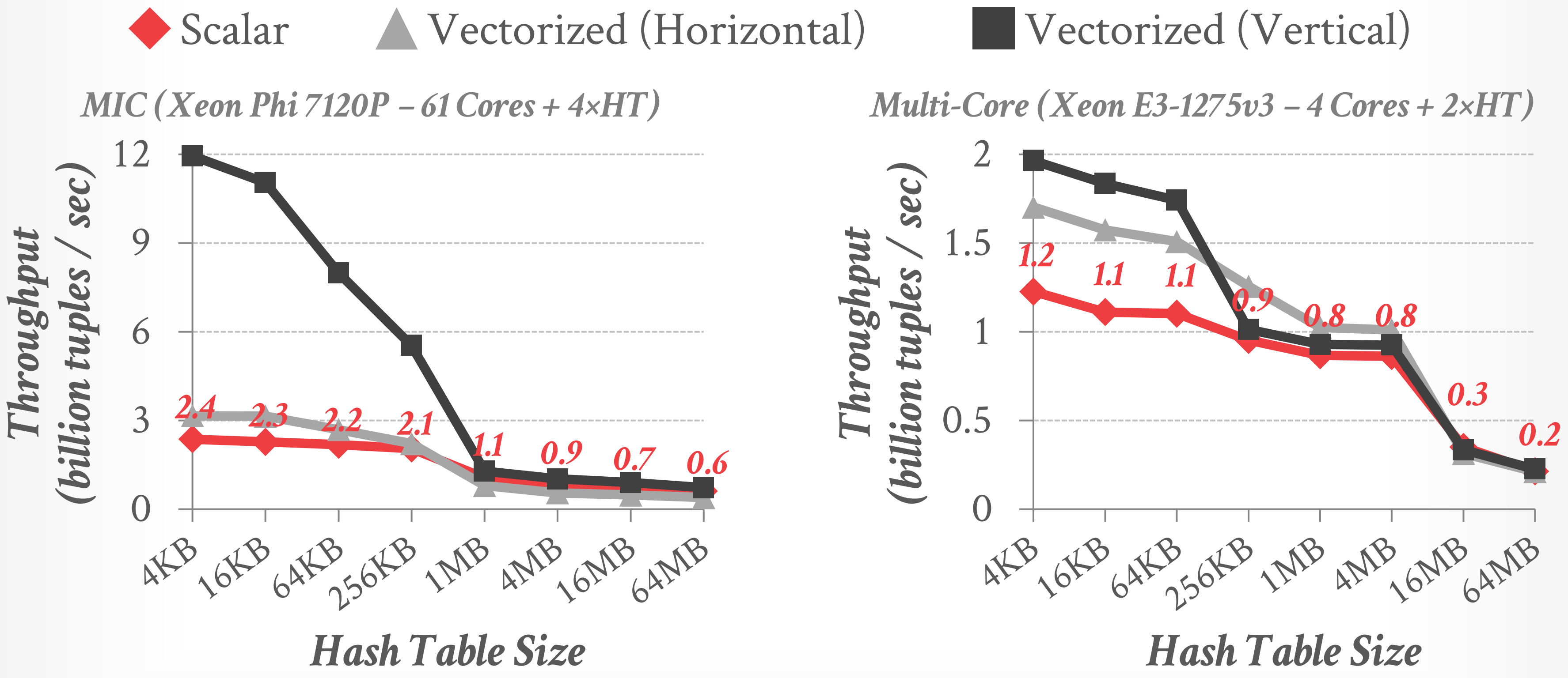
KEY	PAYLOAD
k99	
k1	
k6	
k4	
k5	
k88	

HASH TABLES – PROBING

Vectorized (Vertical)

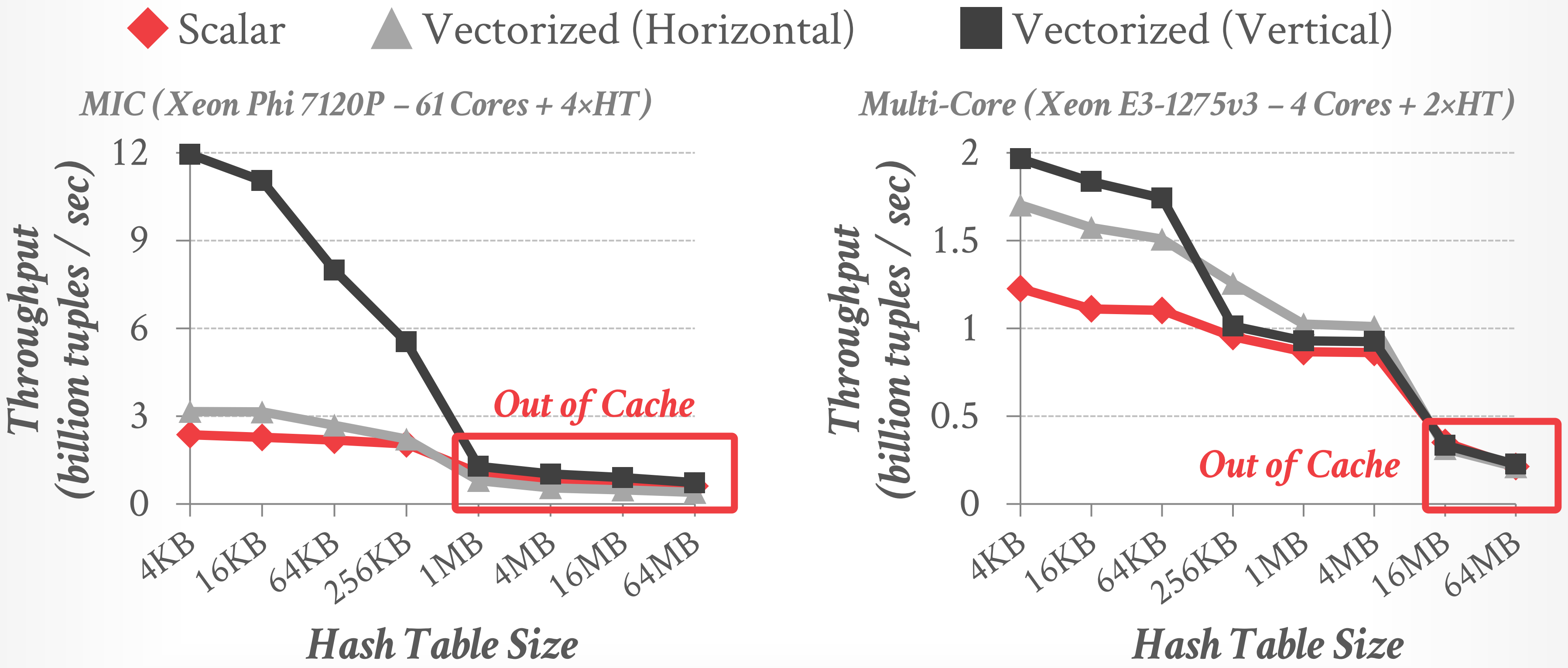


HASH TABLES – PROBING



Source: Orestis Polychroniou

HASH TABLES – PROBING

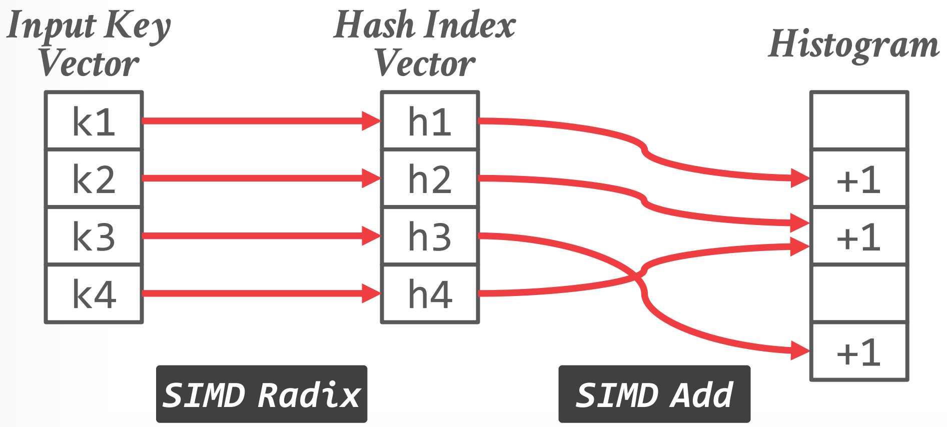


Source: Orestis Polychroniou

PARTITIONING – HISTOGRAM

Use scatter and gathers to increment counts.

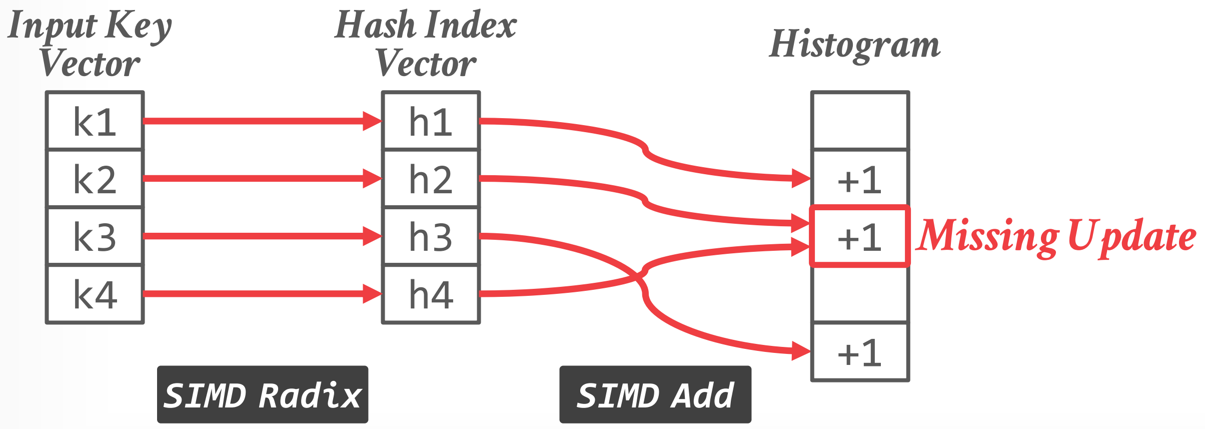
Replicate the histogram to handle collisions.



PARTITIONING – HISTOGRAM

Use scatter and gathers to increment counts.

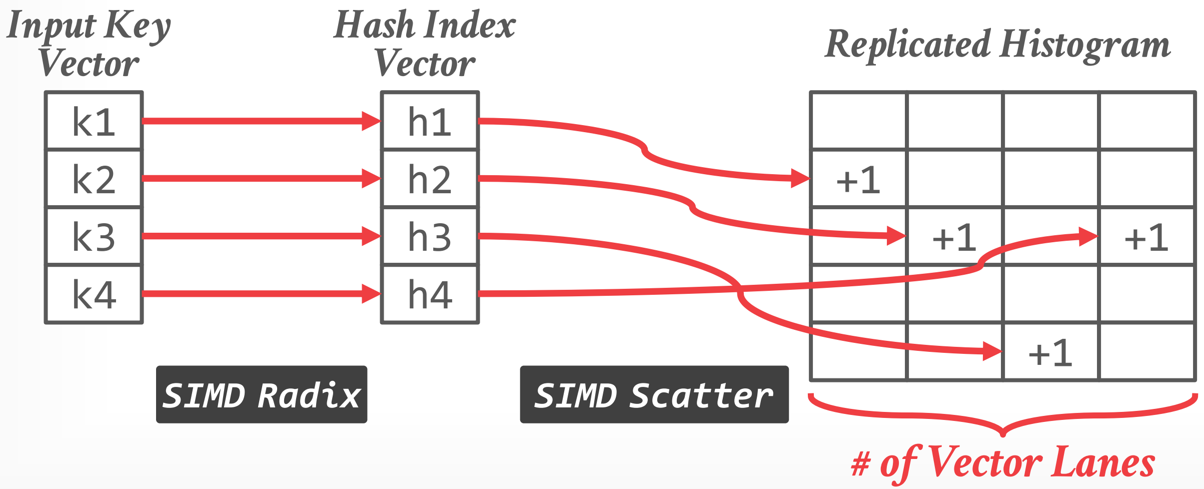
Replicate the histogram to handle collisions.



PARTITIONING – HISTOGRAM

Use scatter and gathers to increment counts.

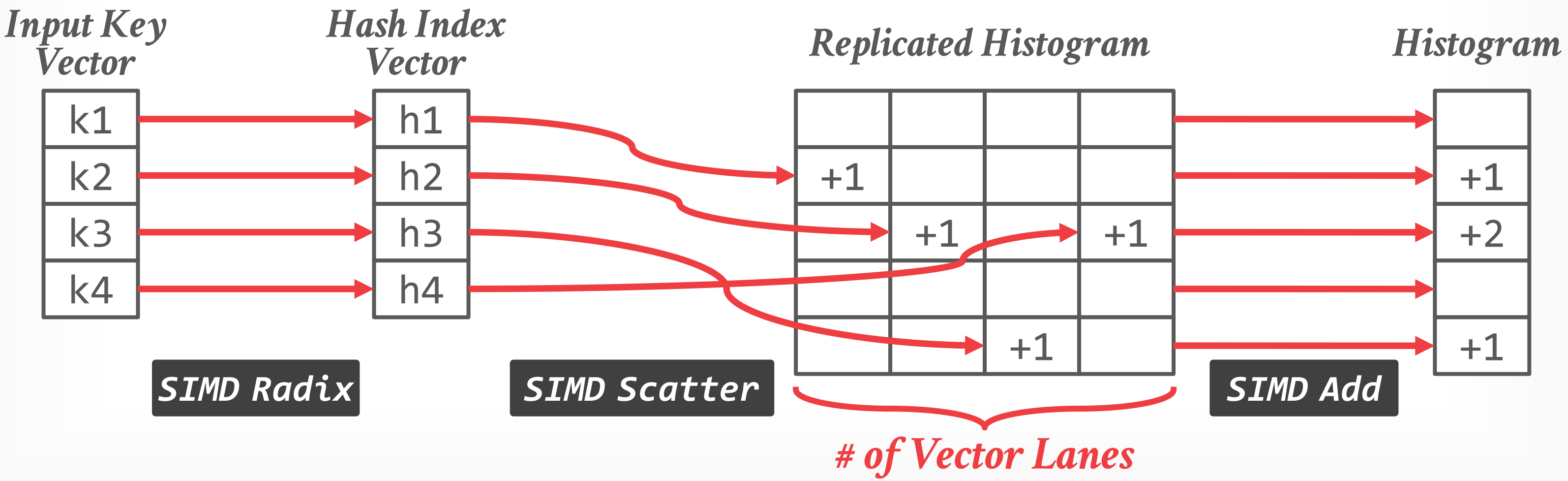
Replicate the histogram to handle collisions.



PARTITIONING – HISTOGRAM

Use scatter and gathers to increment counts.

Replicate the histogram to handle collisions.

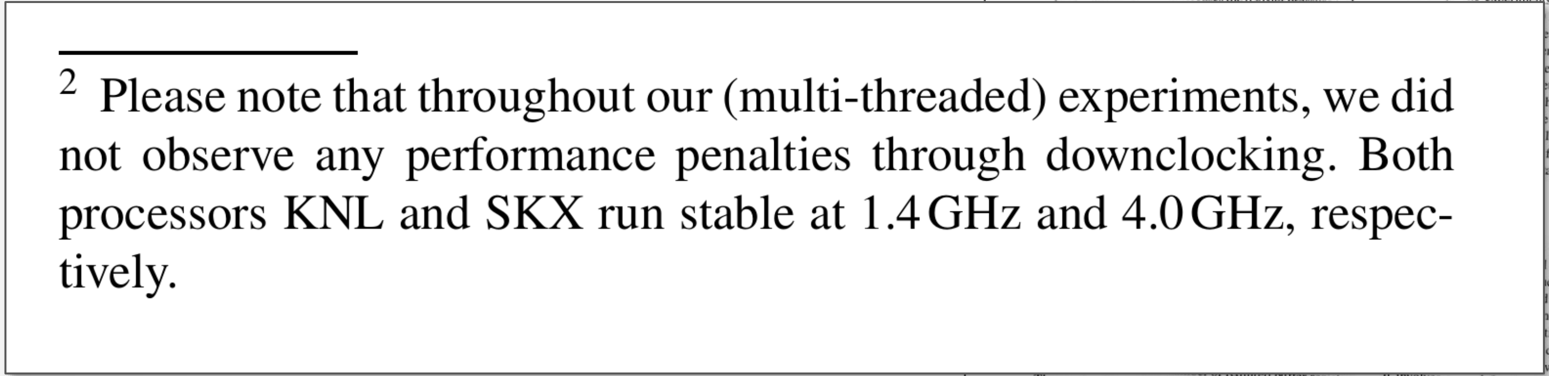


CAVEAT EMPTOR

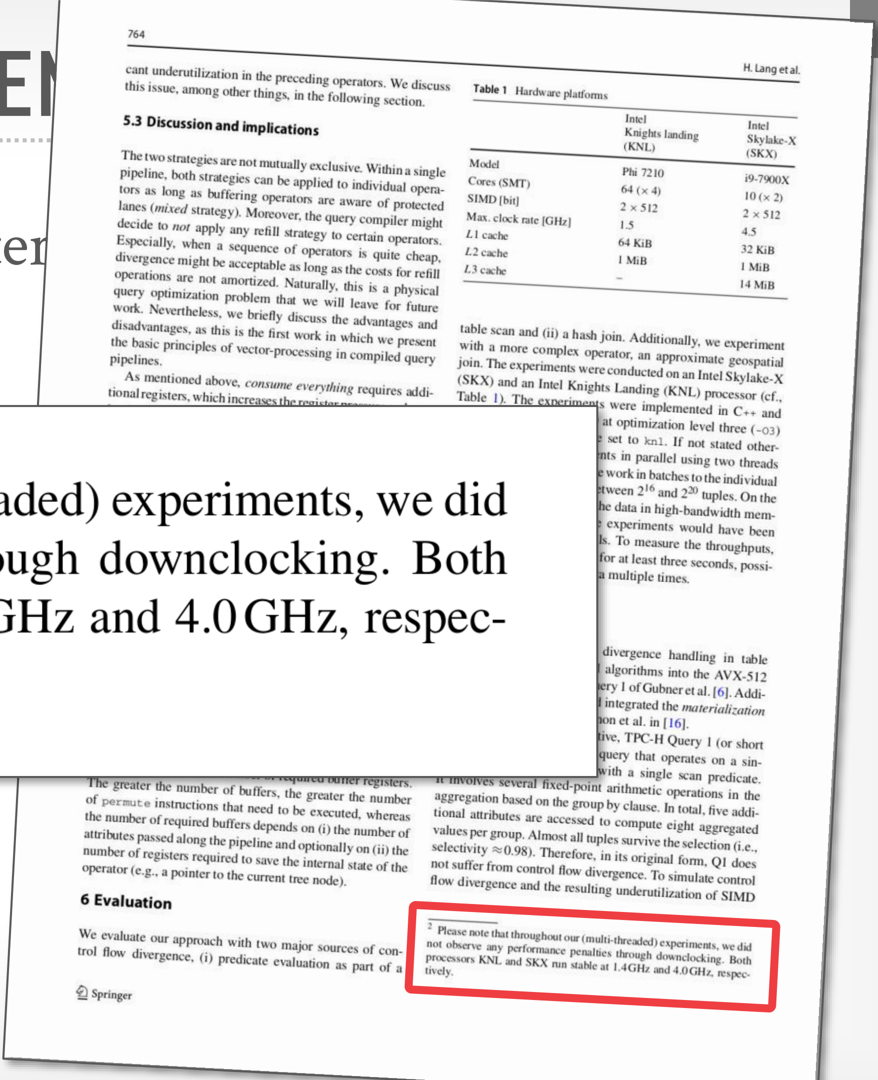
AVX-512 is **not** always faster than AVX2.

CAVEAT E

AVX-512 is **not** always faster



[2] Please note that throughout our (multi-threaded) experiments, we did not observe any performance penalties through downclocking. Both processors KNL and SKX run stable at 1.4 GHz and 4.0 GHz, respectively.



764 H. Lang et al.

cant underutilization in the preceding operators. We discuss this issue, among other things, in the following section.

5.3 Discussion and implications

The two strategies are not mutually exclusive. Within a single pipeline, both strategies can be applied to individual operators as long as buffering operators are aware of protected lanes (*mixed* strategy). Moreover, the query compiler might decide to *not* apply any refill strategy to certain operators. Especially, when a sequence of operators is quite cheap, divergence might be acceptable as long as the costs for refill operations are not amortized. Naturally, this is a physical query optimization problem that we will leave for future work. Nevertheless, we briefly discuss the advantages and disadvantages, as this is the first work in which we present the basic principles of vector-processing in compiled query pipelines.

As mentioned above, *consume everything* requires additional registers, which increases the register

The greater the number of buffers, the greater the number of `permute` instructions that need to be executed, whereas the number of required buffers depends on (i) the number of attributes passed along the pipeline and optionally on (ii) the number of registers required to save the internal state of the operator (e.g., a pointer to the current tree node).

6 Evaluation

We evaluate our approach with two major sources of control flow divergence, (i) predicate evaluation as part of a

Table 1 Hardware platforms

	Intel Knights landing (KNL)	Intel Skylake-X (SKX)
Model	Phi 7210	i9-7900X
Cores (SMT)	64 (× 4)	10 (× 2)
SIMD [bit]	2 × 512	2 × 512
Max. clock rate [GHz]	1.5	4.5
*L*1 cache	64 KiB	32 KiB
*L*2 cache	1 MiB	1 MiB
*L*3 cache	–	14 MiB

table scan and (ii) a hash join. Additionally, we experiment with a more complex operator, an approximate geospatial join. The experiments were conducted on an Intel Skylake-X (SKX) and an Intel Knights Landing (KNL) processor (cf., Table 1). The experiments were implemented in C++ and at optimization level three (`-O3`) set to `knl`. If not stated otherwise nts in parallel using two threads e work in batches to the individual tween 2^{16} and 2^{20} tuples. On the he data in high-bandwidth mem- experiments would have been ls. To measure the throughputs, for at least three seconds, possi- a multiple times.

divergence handling in table algorithms into the AVX-512 ery 1 of Gubner et al. [6]. Addi- integrated the *materialization* ion et al. in [16]. tive, TPC-H Query 1 (or short query that operates on a sin- with a single scan predicate. It involves several fixed-point arithmetic operations in the aggregation based on the group by clause. In total, five additional attributes are accessed to compute eight aggregated values per group. Almost all tuples survive the selection (i.e., selectivity ≈0.98). Therefore, in its original form, Q1 does not suffer from control flow divergence. To simulate control flow divergence and the resulting underutilization of SIMD

[2] Please note that throughout our (multi-threaded) experiments, we did not observe any performance penalties through downclocking. Both processors KNL and SKX run stable at 1.4GHz and 4.0 GHz, respectively.

Springer

CAVEAT EMPTOR

AVX-512 is **not** always faster than AVX2.

Some CPUs downgrade their clockspeed when switching to AVX-512 mode.
→ Compilers will prefer 256-bit SIMD operations.

If only a small portion of the process uses AVX-512, then it is not worth the downclock penalty.

EMPTOR

ster than AVX2.

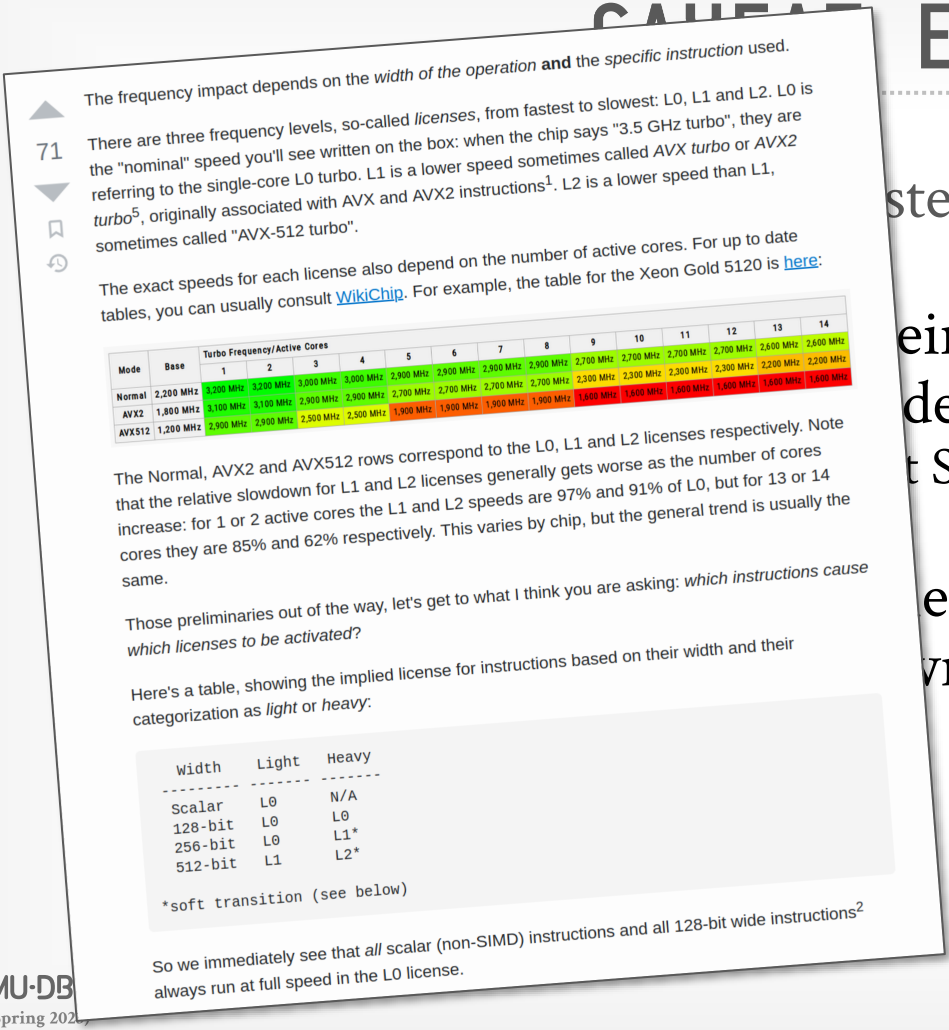
eir clockspeed when

de.

t SIMD operations.

e process uses AVX-512,

vnclock penalty.



71

The frequency impact depends on the *width of the operation* **and** the *specific instruction* used.

There are three frequency levels, so-called *licenses*, from fastest to slowest: L0, L1 and L2. L0 is the "nominal" speed you'll see written on the box: when the chip says "3.5 GHz turbo", they are referring to the single-core L0 turbo. L1 is a lower speed sometimes called *AVX turbo* or *AVX2 turbo*[5], originally associated with AVX and AVX2 instructions[1]. L2 is a lower speed than L1, sometimes called "AVX-512 turbo".

The exact speeds for each license also depend on the number of active cores. For up to date tables, you can usually consult WikiChip. For example, the table for the Xeon Gold 5120 is here:

Mode	Base	Turbo Frequency/Active Cores													
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
Normal	2,200 MHz	3,200 MHz	3,200 MHz	3,000 MHz	3,000 MHz	2,900 MHz	2,900 MHz	2,900 MHz	2,900 MHz	2,700 MHz	2,700 MHz	2,700 MHz	2,700 MHz	2,600 MHz	2,600 MHz
AVX2	1,800 MHz	3,100 MHz	3,100 MHz	2,900 MHz	2,900 MHz	2,700 MHz	2,700 MHz	2,700 MHz	2,700 MHz	2,300 MHz	2,300 MHz	2,300 MHz	2,300 MHz	2,200 MHz	2,200 MHz
AVX512	1,200 MHz	2,900 MHz	2,900 MHz	2,500 MHz	2,500 MHz	1,900 MHz	1,900 MHz	1,900 MHz	1,900 MHz	1,600 MHz	1,600 MHz	1,600 MHz	1,600 MHz	1,600 MHz	1,600 MHz

The Normal, AVX2 and AVX512 rows correspond to the L0, L1 and L2 licenses respectively. Note that the relative slowdown for L1 and L2 licenses generally gets worse as the number of cores increase: for 1 or 2 active cores the L1 and L2 speeds are 97% and 91% of L0, but for 13 or 14 cores they are 85% and 62% respectively. This varies by chip, but the general trend is usually the same.

Those preliminaries out of the way, let's get to what I think you are asking: *which instructions cause which licenses to be activated*?

Here's a table, showing the implied license for instructions based on their width and their categorization as *light* or *heavy*:

```
 Width      Light   Heavy
---------  ------- -------
 Scalar     L0      N/A
 128-bit    L0      L0
 256-bit    L0      L1*
 512-bit    L1      L2*

*soft transition (see below)
```

So we immediately see that *all* scalar (non-SIMD) instructions and all 128-bit wide instructions[2] always run at full speed in the L0 license.

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.

NEXT CLASS

Query Compilation

Project #3 Topics