Vectorized query execution

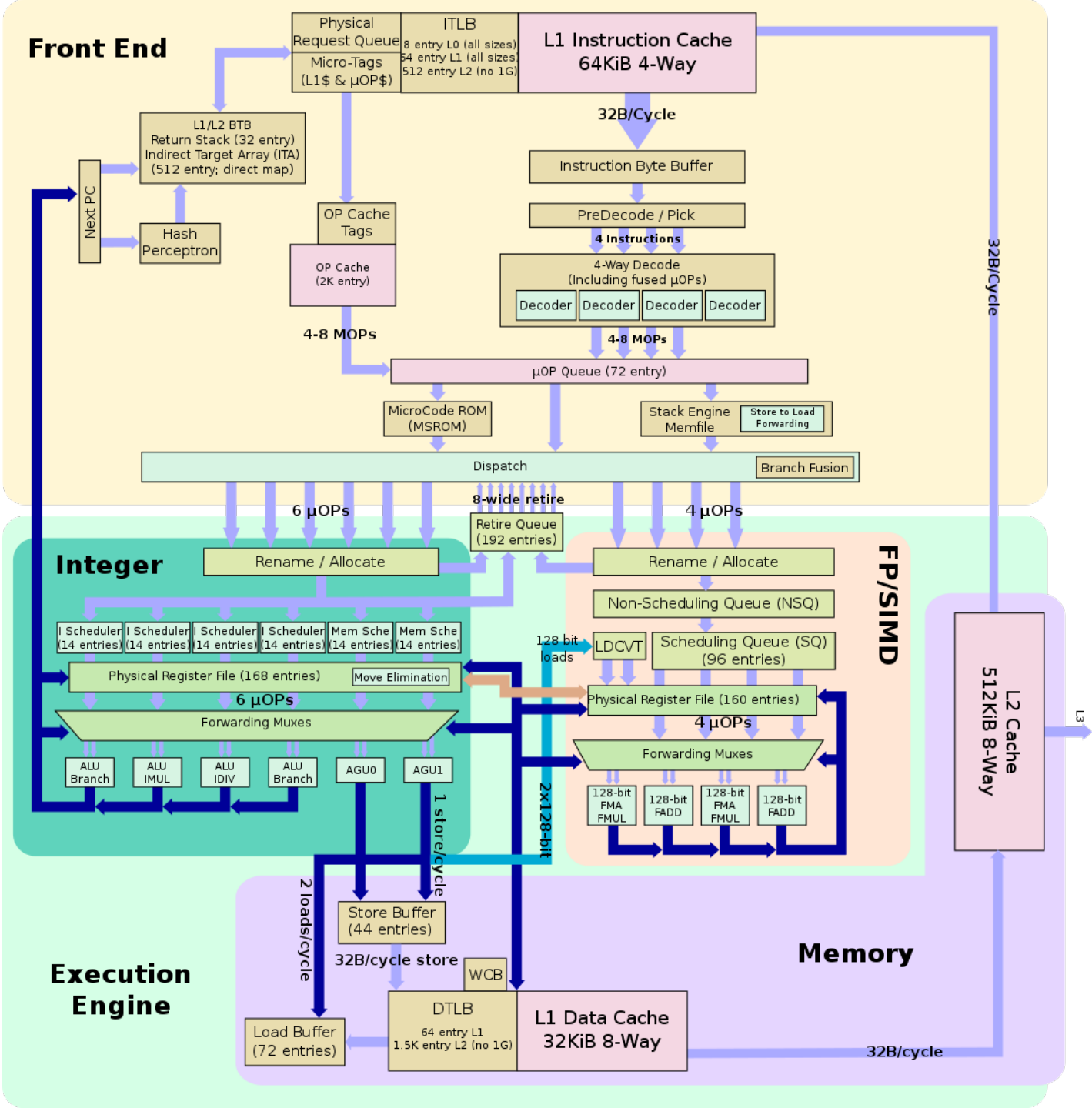
Iztok Savnik, FAMNIT

October, 2025.

Sources

- Course (Spring 2024):
  - Carnegie Mellon University (CMU)
  - **Advanced Database Systems**
  - Prof. Andy Pavlo
  - Transparencies
- Papers:
  - Conferences VLDB, SIGMOD, CIDR, etc.
  - Journals ACM TODS, IS, VLDBJ, etc.

AMD Ryzen 9 9950X (Zen 5)



TODAY’S AGENDA

Background
Implementation Approaches
Vectorization 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.

This technique is known as **Data Parallelization**.

WHY THIS MATTERS

Suppose the DBMS can parallelize some 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.

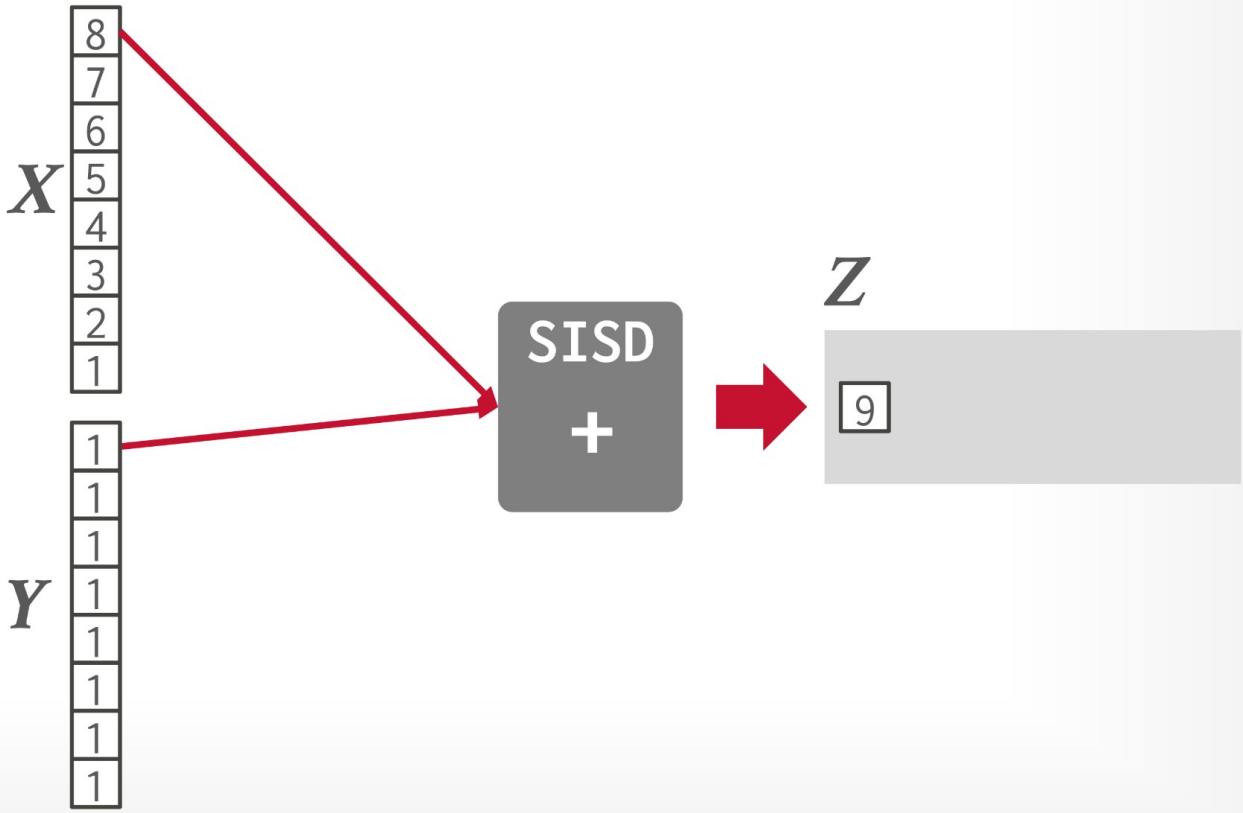
- x86: MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX2, AVX512
- PowerPC: Altivec
- ARM: NEON, SVE, SVE2
- RISC-V: RVV

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];
}
```



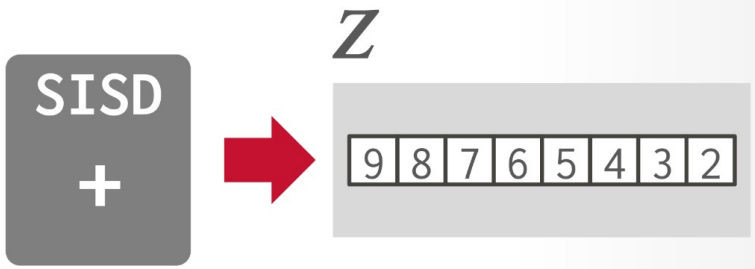
CMU·DB

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];
}
```



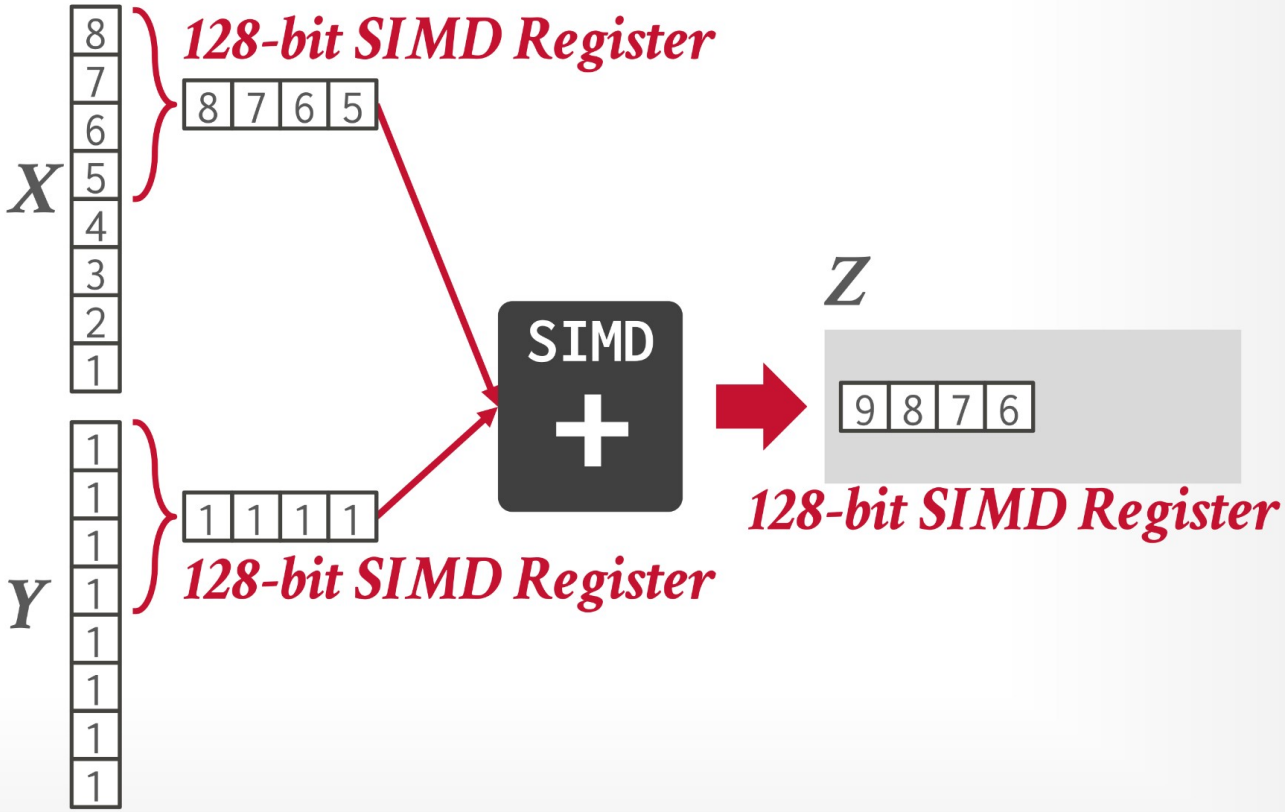
CMU·DB

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];
}
```



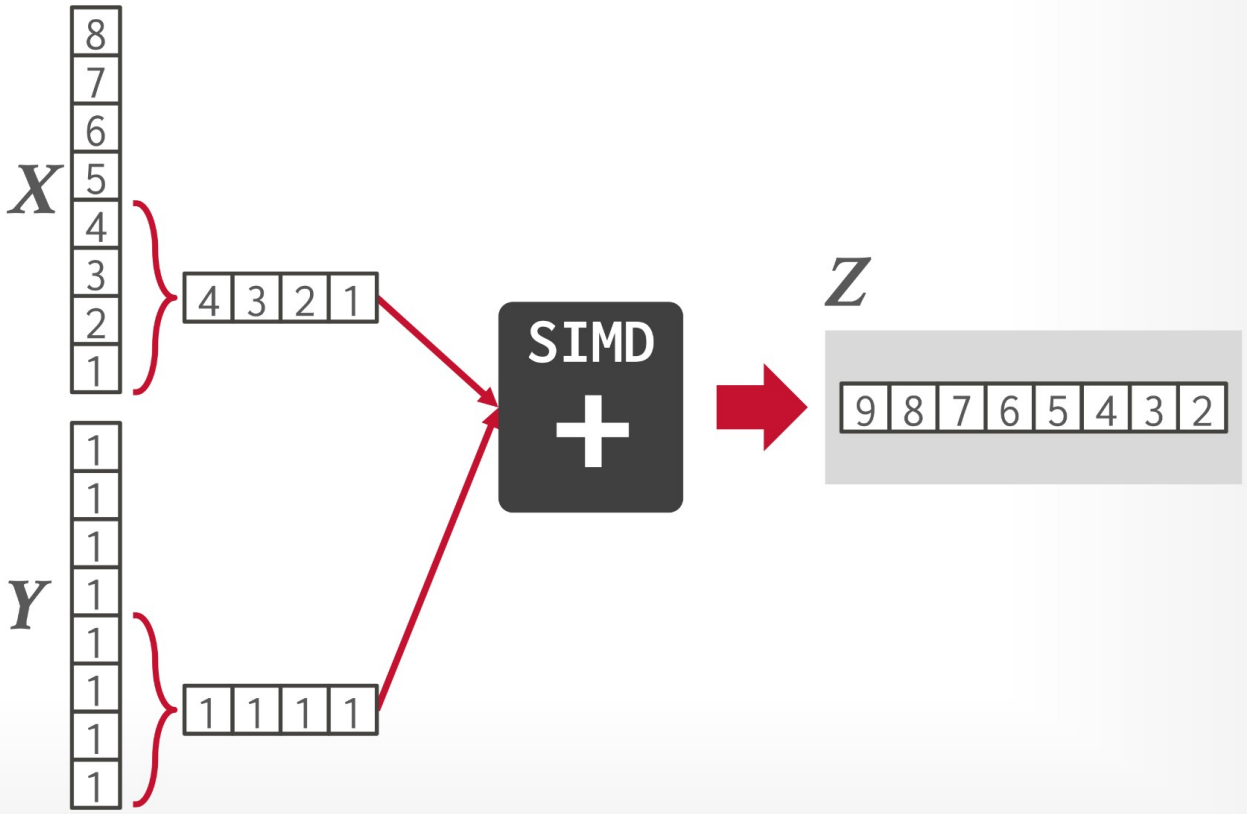
CMU·DB

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];
}
```

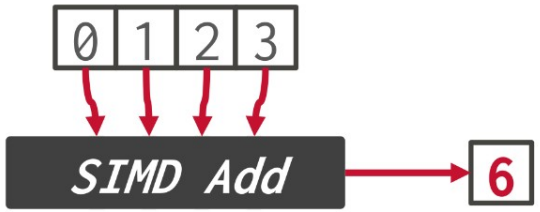


CMU·DB

SIMD EXAMPLE

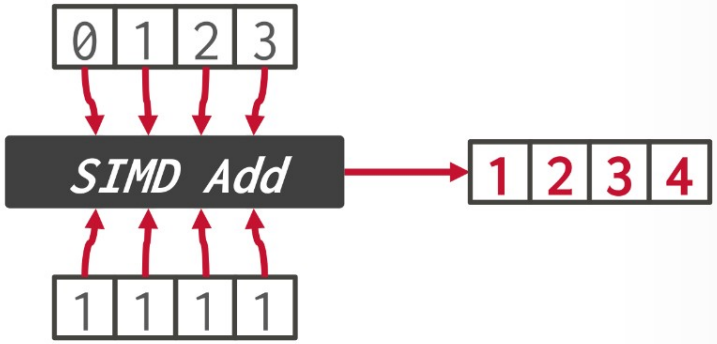
Approach #1: Horizontal

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



Approach #2: Vertical

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



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)**

James Reinders: Argonne Training Program on Extreme-Scale Computing, Summer 2016.

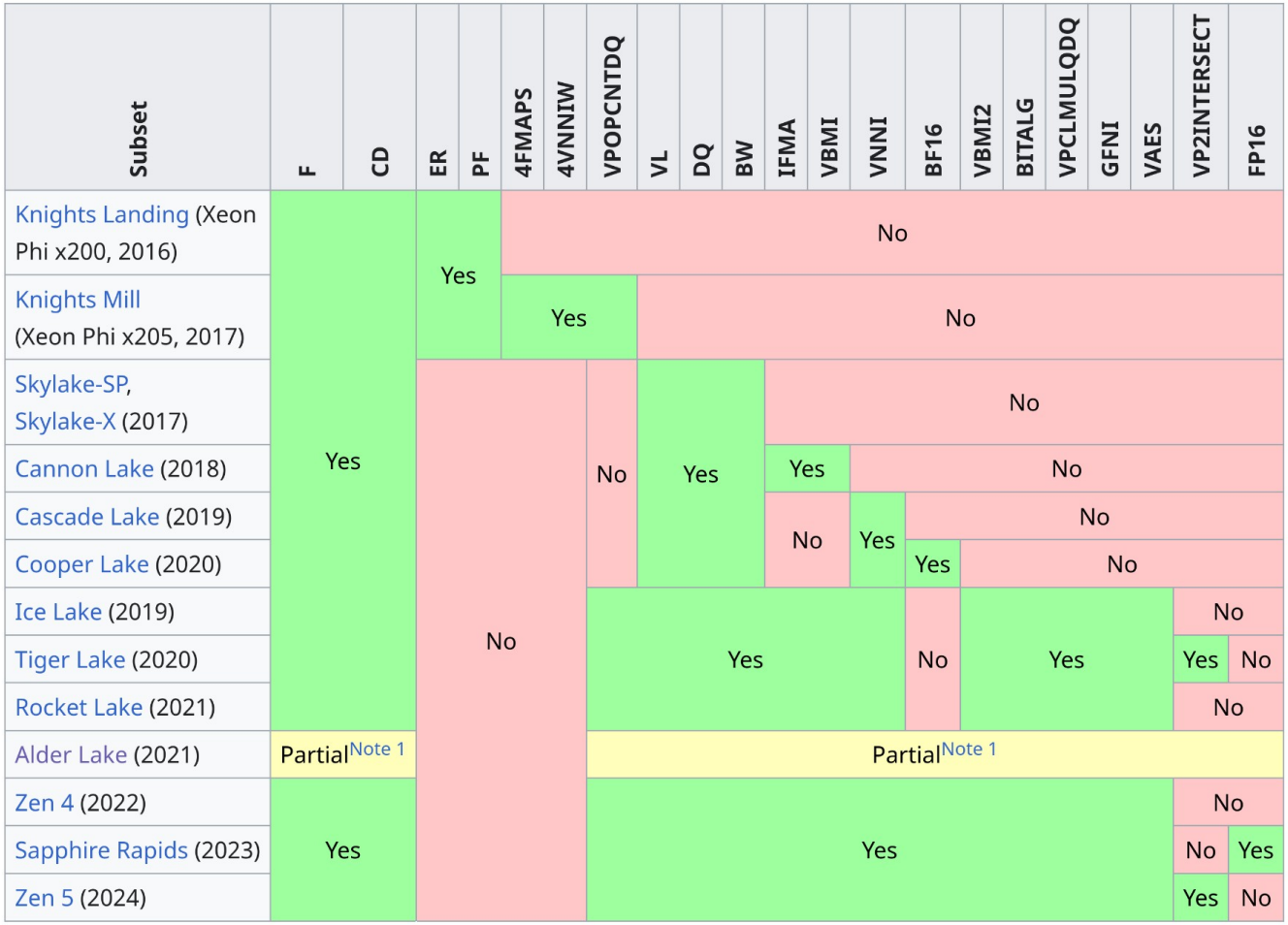
AVX-512

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



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		No																
Knights Mill (Xeon Phi x205, 2017)					Yes		No														
Skylake-SP, Skylake-X (2017)			No				No	Yes			No										
Cannon Lake (2018)											Yes	No									
Cascade Lake (2019)											No	Yes	No								
Cooper Lake (2020)														Yes	No						
Ice Lake (2019)							Yes							No	Yes					No	
Tiger Lake (2020)																				Yes	No
Rocket Lake (2021)																				No	
Alder Lake (2021)	Partial[Note 1]						Partial[Note 1]														
Zen 4 (2022)	Yes						Yes													No	
Sapphire Rapids (2023)																				No	Yes
Zen 5 (2024)																				Yes	No

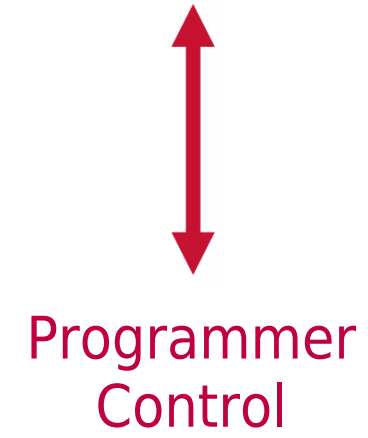
https://en.wikipedia.org/wiki/AVX-512#CPUs_with_AVX-512

IMPLEMENTATION

Choice #1: Automatic Vectorization

Choice #2: Compiler Hints

Choice #3: Explicit Vectorization



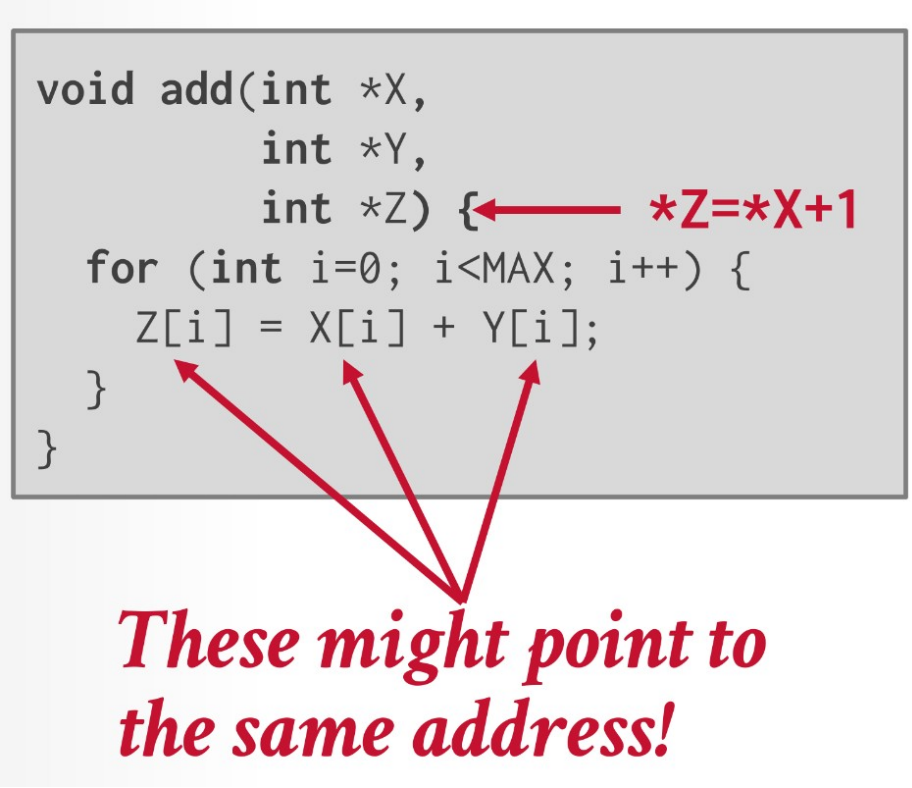
AUTOMATIC VECTORIZATION

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

This loop is not legal to automatically vectorize because 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

The **restrict** keyword in C/C++ tells the compiler that the arrays are distinct memory locations for the lifetime of the pointers.

The compiler can then infer that it is safe to vectorize operations on those pointers.

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

COMPILER HINTS

duckdb / src / common / vector_operations / **vector_hash.cpp**

Code | Blame | 520 lines (472 loc) · 19.4 KB | Raw

```
37  hash_t CombineHashScalar(hash_t a, hash_t b) {
39      a *= 0xd6e8feb86659fd93U;
40      return a ^ b;
41  }
42
43  template <bool HAS_RSEL, bool HAS_SEL_VECTOR, class T, bool INPUT_IS_ALREADY_HASH>
44  void TightLoopHash(const T *__restrict ldata, hash_t *__restrict result_data, const SelectionVector *rsel, idx_t count,
45                     const SelectionVector *__restrict sel_vector, const ValidityMask &mask) {
46      if (!mask.AllValid()) {
47          for (idx_t i = 0; i < count; i++) {
48              auto ridx = HAS_RSEL ? rsel->get_index_unsafe(i) : i;
49              auto idx = HAS_SEL_VECTOR ? sel_vector->get_index_unsafe(ridx) : ridx;
50              result_data[ridx] = INPUT_IS_ALREADY_HASH ? CachedHashOp::Operation(ldata[idx])
51                                                        : HashOp::Operation(ldata[idx], !mask.RowIsValidUnsafe(idx));
52          }
53      } else {
54          for (idx_t i = 0; i < count; i++) {
55              auto ridx = HAS_RSEL ? rsel->get_index_unsafe(i) : i;
56              auto idx = HAS_SEL_VECTOR ? sel_vector->get_index_unsafe(ridx) : ridx;
57              result_data[ridx] =
58                  INPUT_IS_ALREADY_HASH ? CachedHashOp::Operation(ldata[idx]) : duckdb::Hash<T>(ldata[idx]);
59          }
60      }
61  }
```

COMPILER HINTS

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.

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

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 (Rust Experimental)

EXPLICIT VECTORIZATION

Store the vectors in 128-bit SIMD registers.

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

```
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++));
  }
}
```

AUTOMATIC VECTORIZATION

Evaluate how well the compiler can automatically vectorize the Vectorwise primitives.

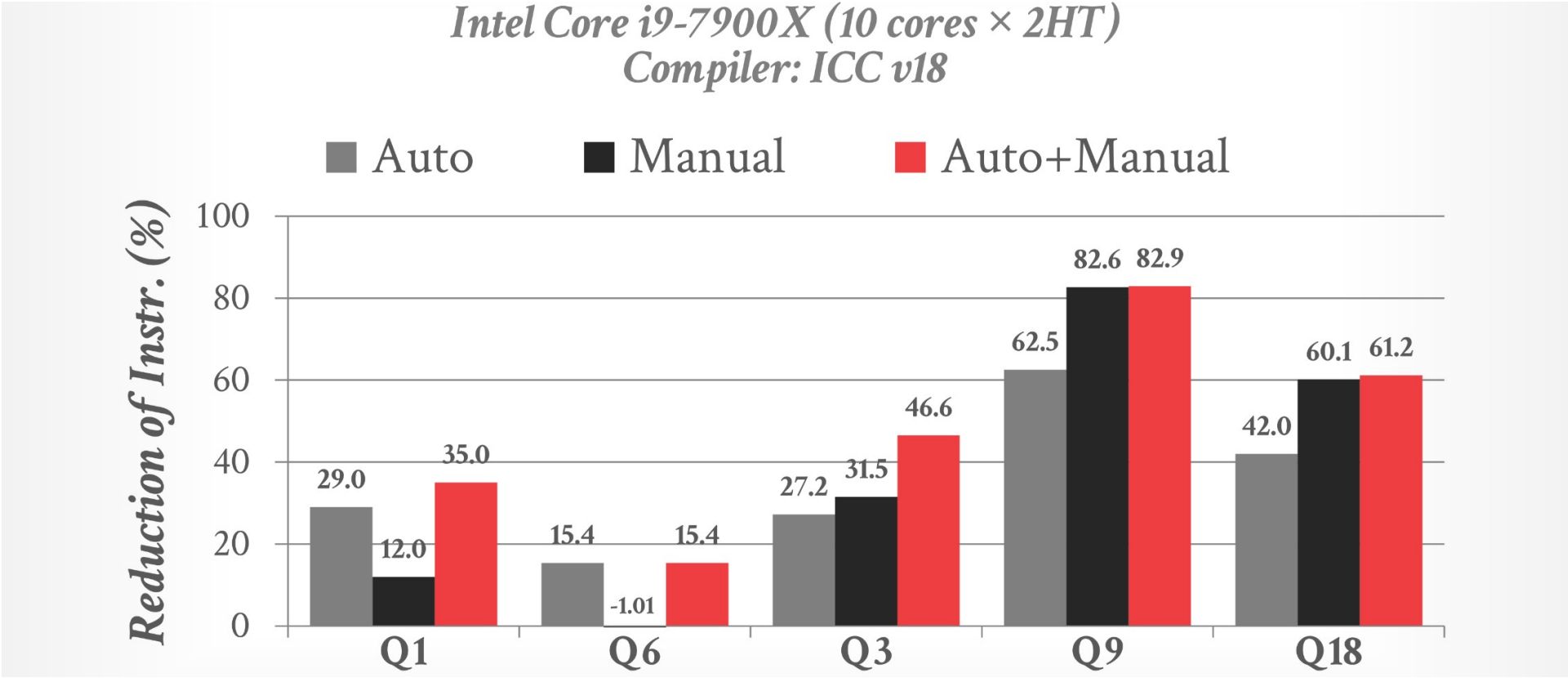
- Targets: GCC v7.2, Clang v5.0, ICC v18

ICC was able to vectorize the most primitives using AVX-512:

- Vectorized: Hashing, Selection, Projection
- Not Vectorized: Hash Table Probing, Aggregation

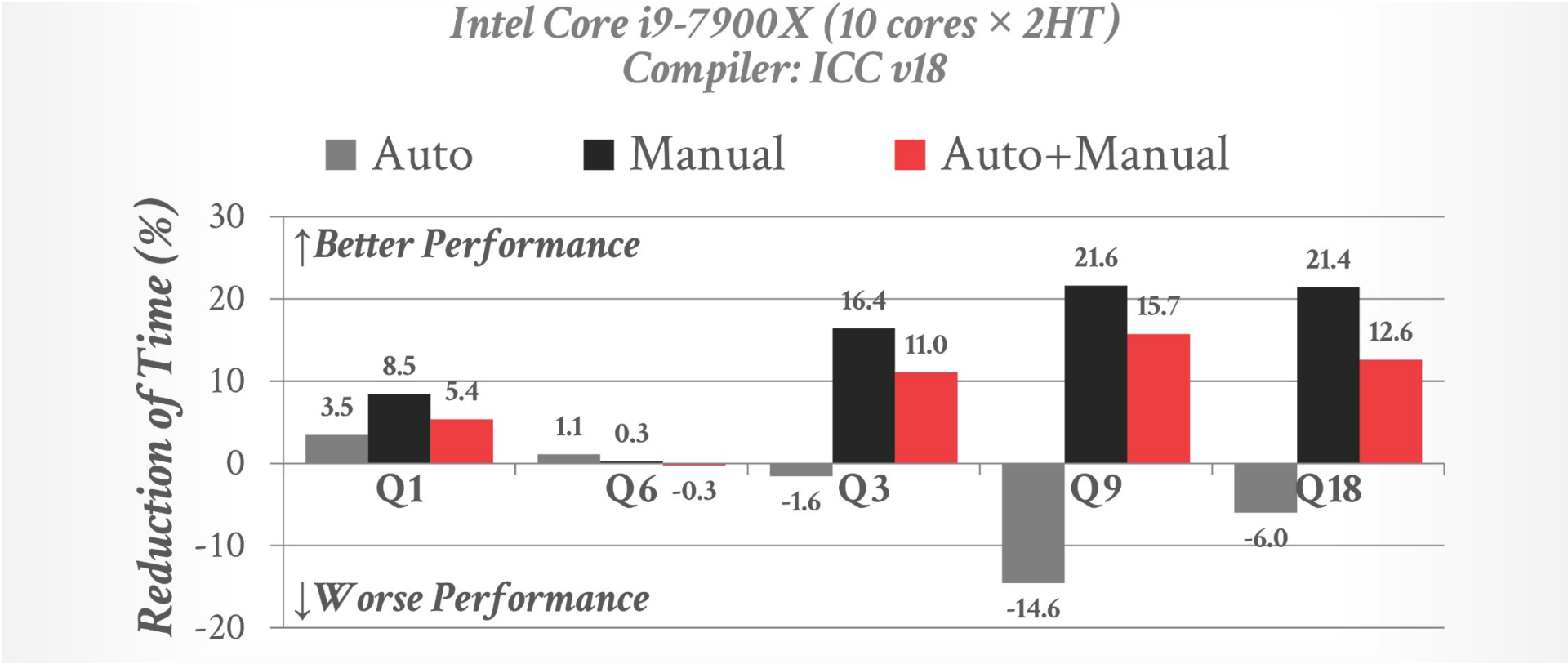
Kersten, et.al, Everything You Always Wanted to Know About Compiled and Vectorized Queries ..., VLDB, 2018.

AUTOMATIC VECTORIZATION



Kersten, et.al, Everything You Always Wanted to Know About Compiled and Vectorized Queries ..., VLDB, 2018.

AUTOMATIC VECTORIZATION



Kersten, et.al, Everything You Always Wanted to Know About Compiled and Vectorized Queries ..., VLDB, 2018.

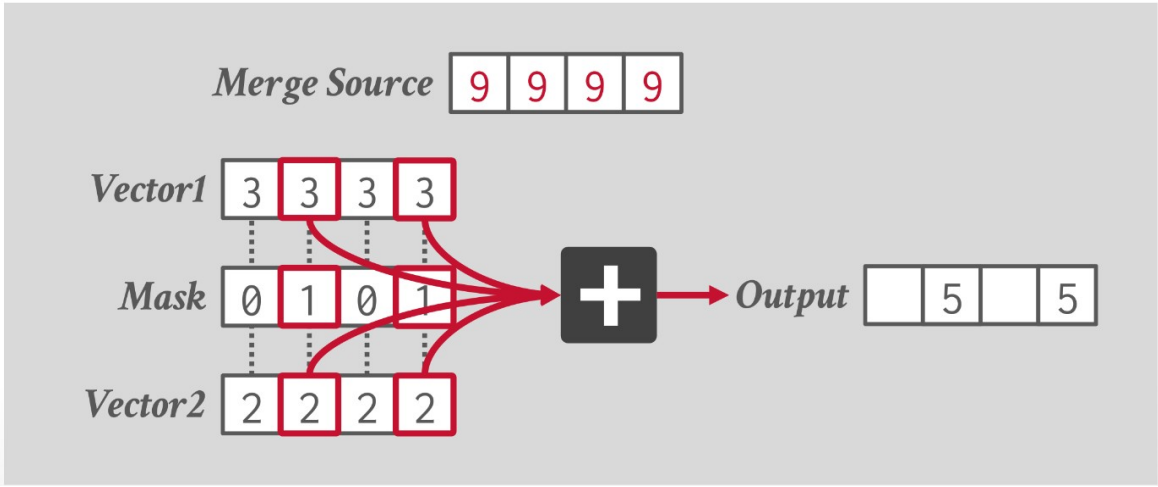
VECTORIZATION FUNDAMENTALS

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

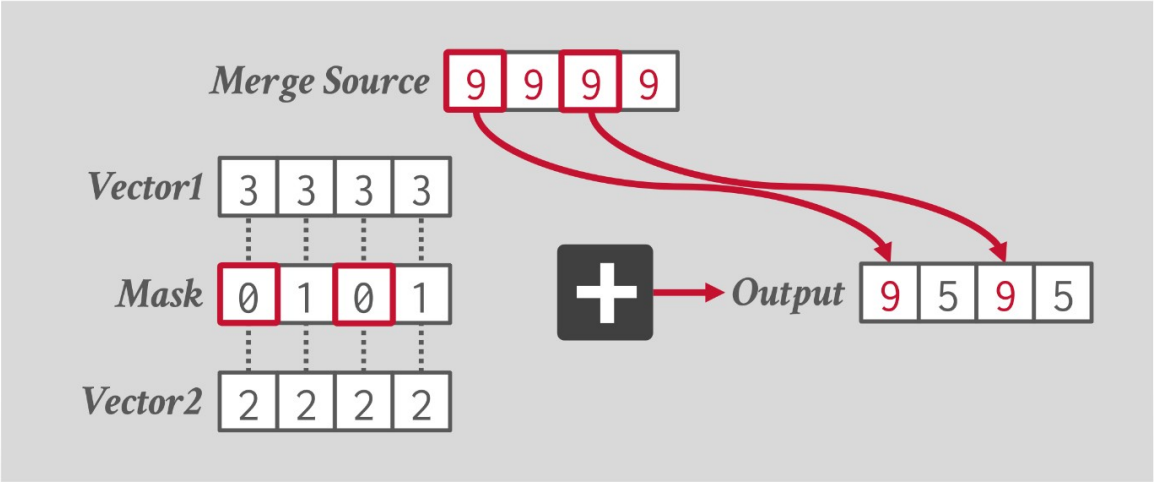
SIMD MASKING

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



SIMD MASKING

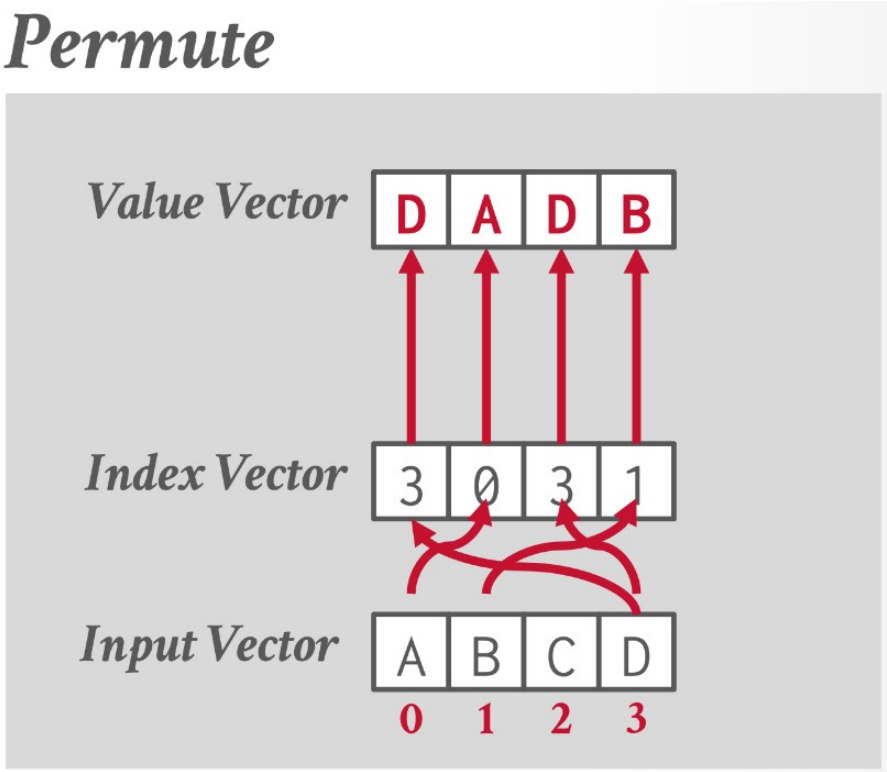
Almost all AVX-512 operations support **predication** variants whereby the CPU only performs operations on lanes specified by an input bitmask.



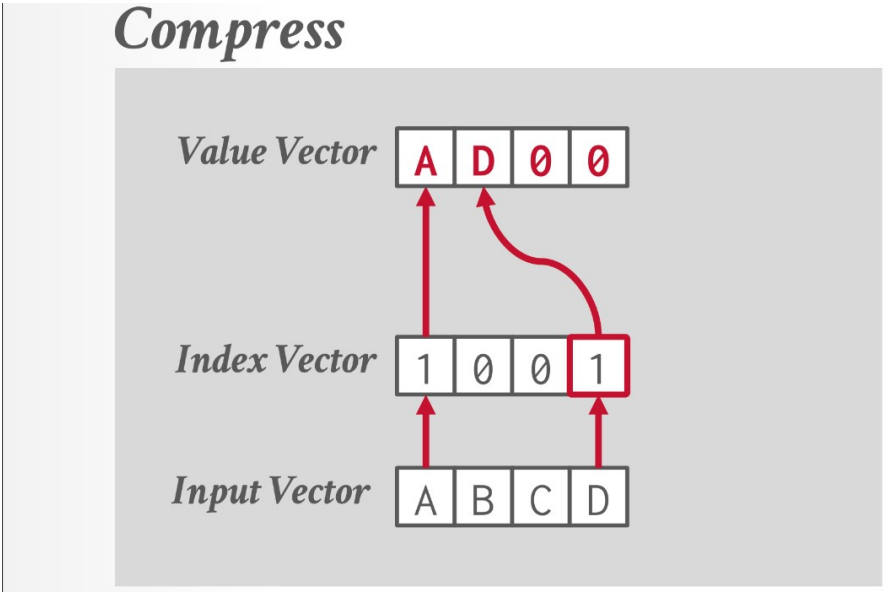
PERMUTE

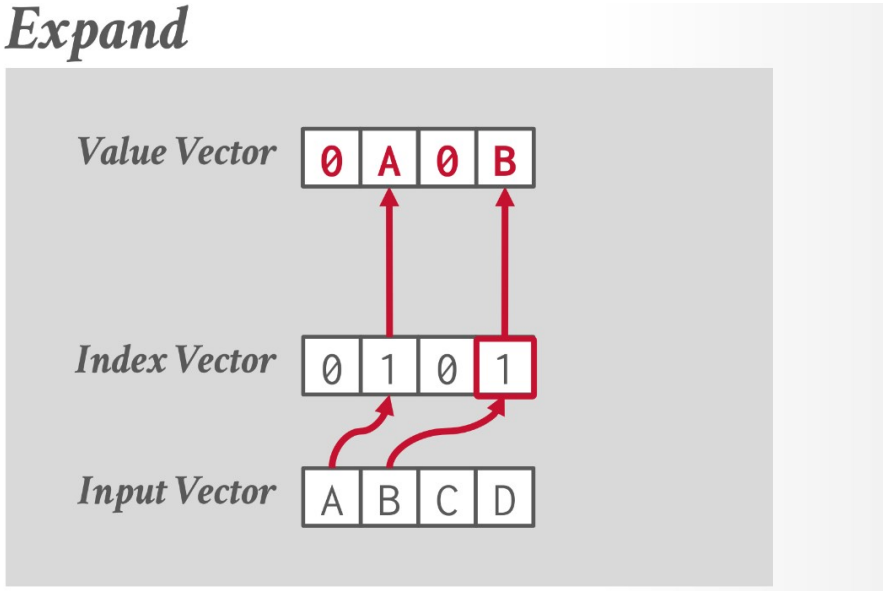
For each lane, copy values in the **input vector** specified by the offset in the **index vector** into the **destination vector**.

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

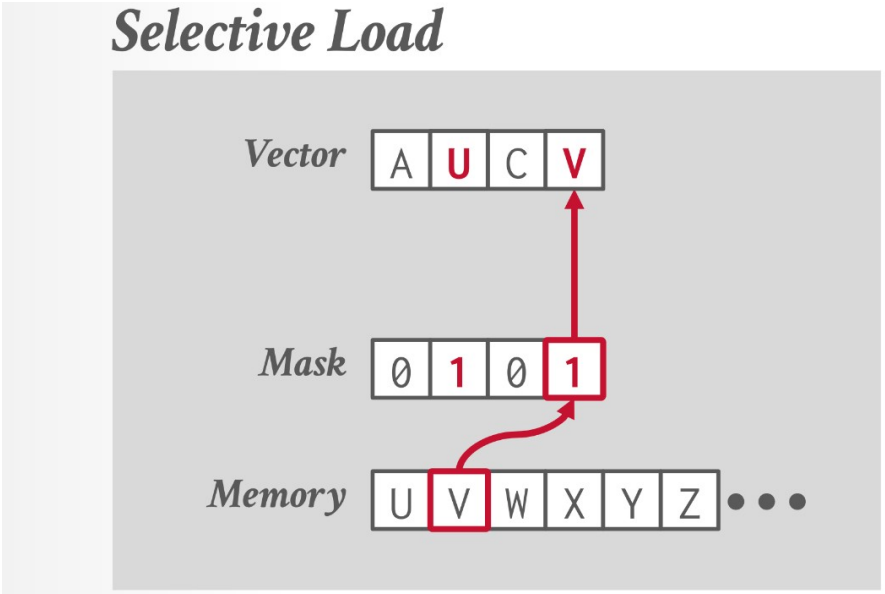


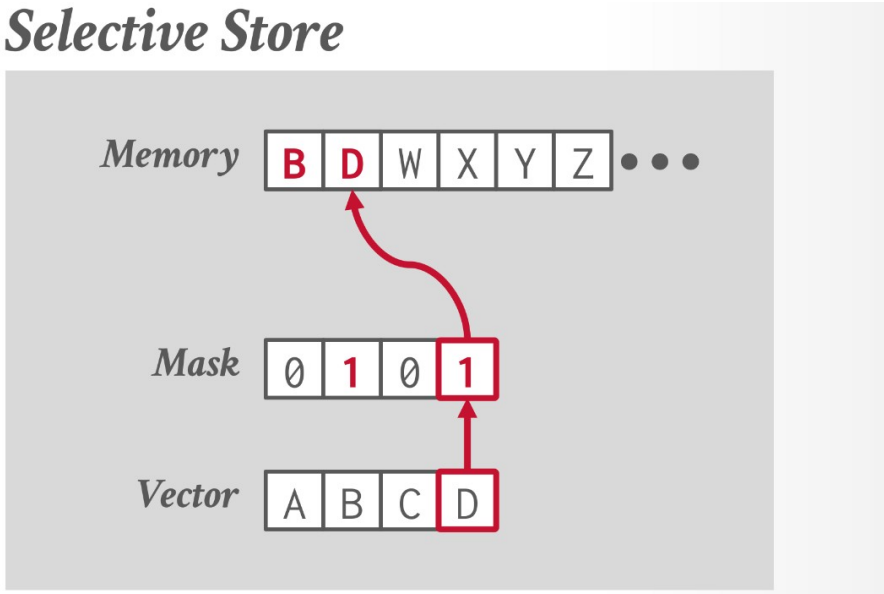
COMPRESS / EXPAND



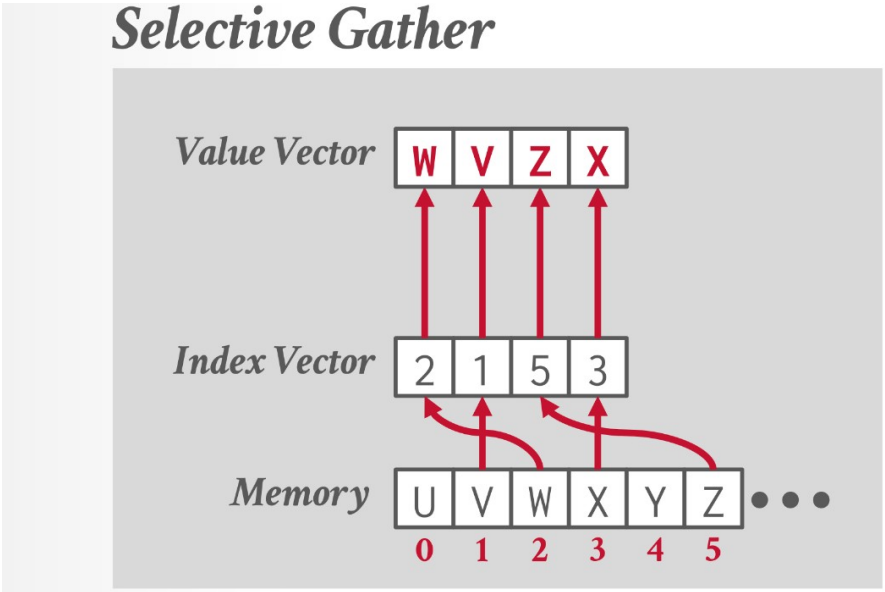


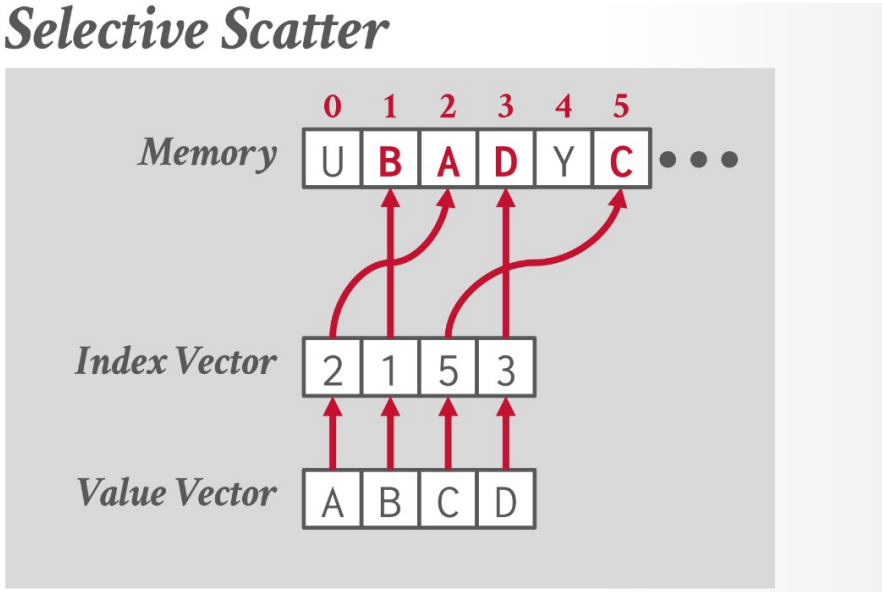
SELECTIVE LOAD/STORE





SELECTIVE SCATTER/GATHER





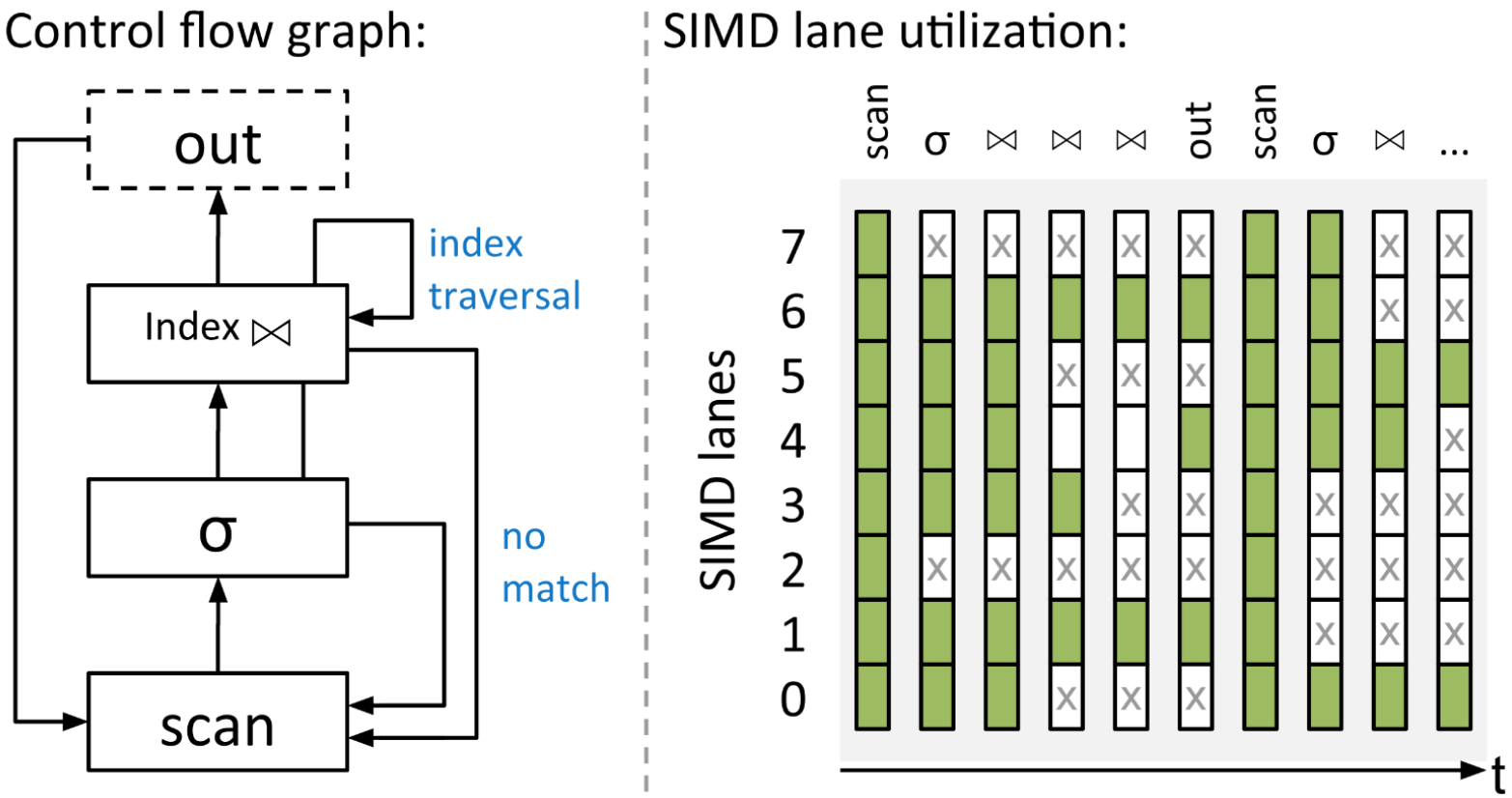
VECTORIZED DBMS ALGORITHMS

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).

Lang-et al.-Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines-vldbj2020

VECTORIZED DBMS ALGORITHMS



Lang-et al.-Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines-vldbj2020

VECTORIZED OPERATORS

Selection Scans

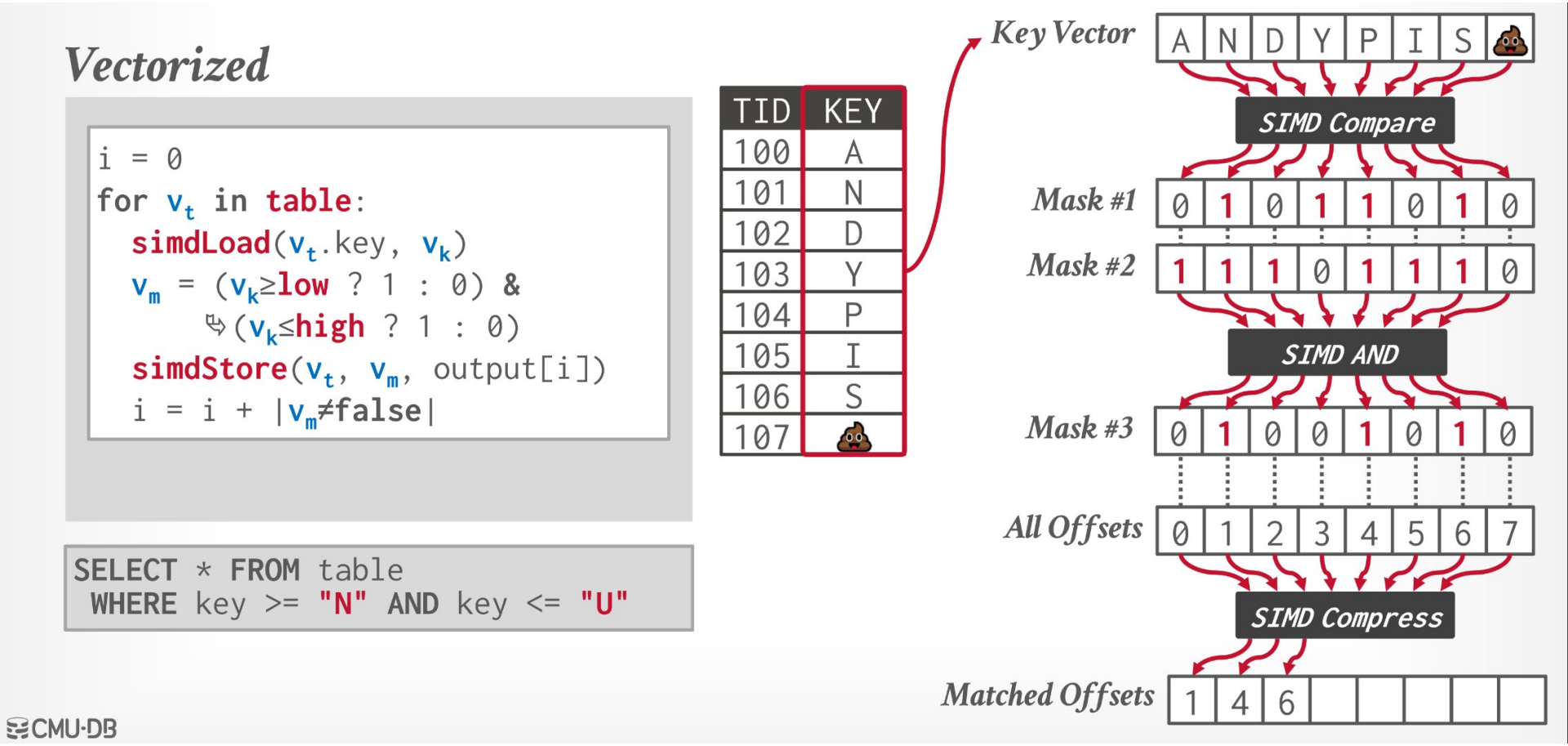
Relaxed operator fusion (ROF)

Vector Refill

Hash Tables

Partitioning / Histograms

SELECTION SCANS



SELECTION SCANS

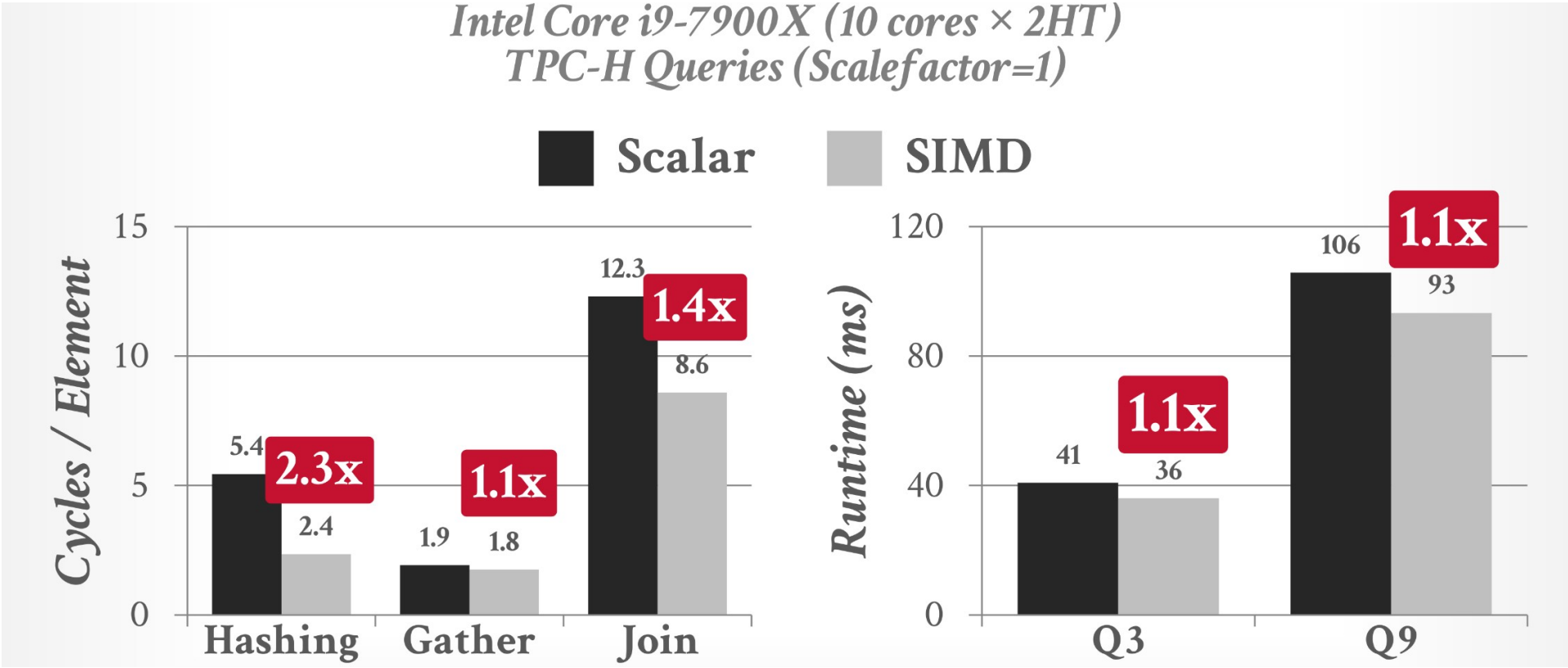
Evaluate branchless selection and hash probe in a open-source implementation of Vectorwise.

Use AVX-512 because it includes instructions to make it easier to implement algorithms using vertical vectorization.

- Selective operations using bitmask registers.

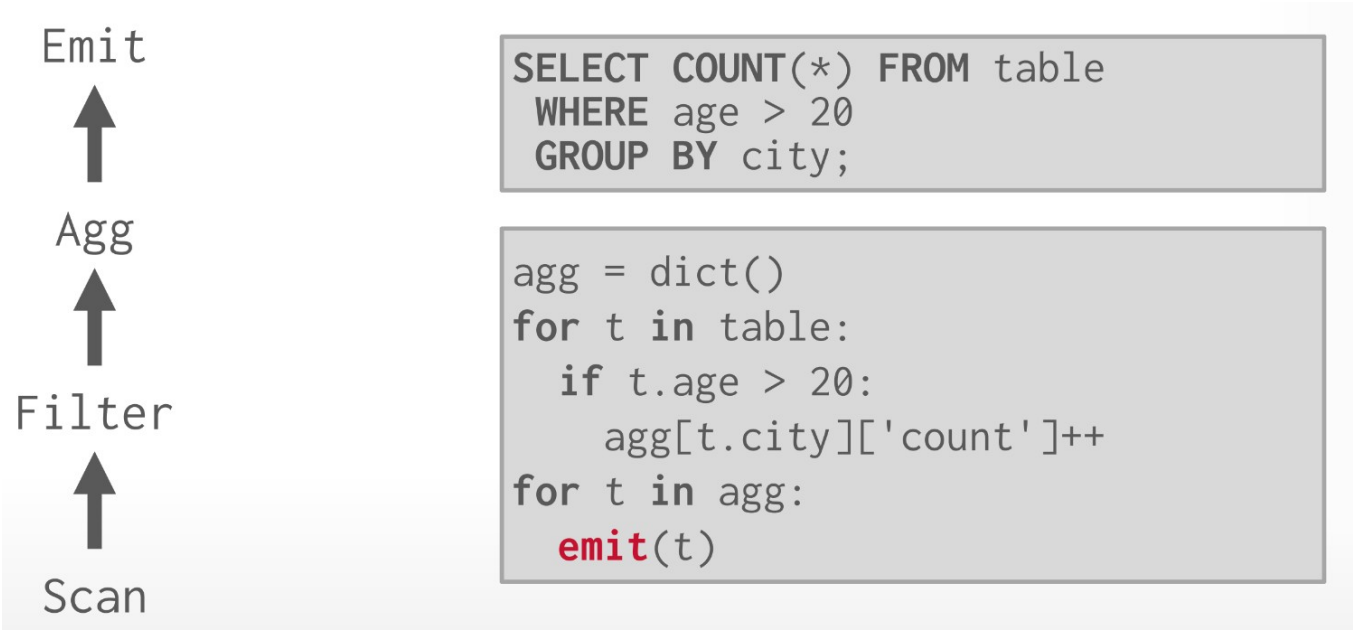
Kersten, et.al, Everything You Always Wanted to Know About Compiled and Vectorized Queries ..., VLDB, 2018.

SIMD EVALUATION



OBSERVATION

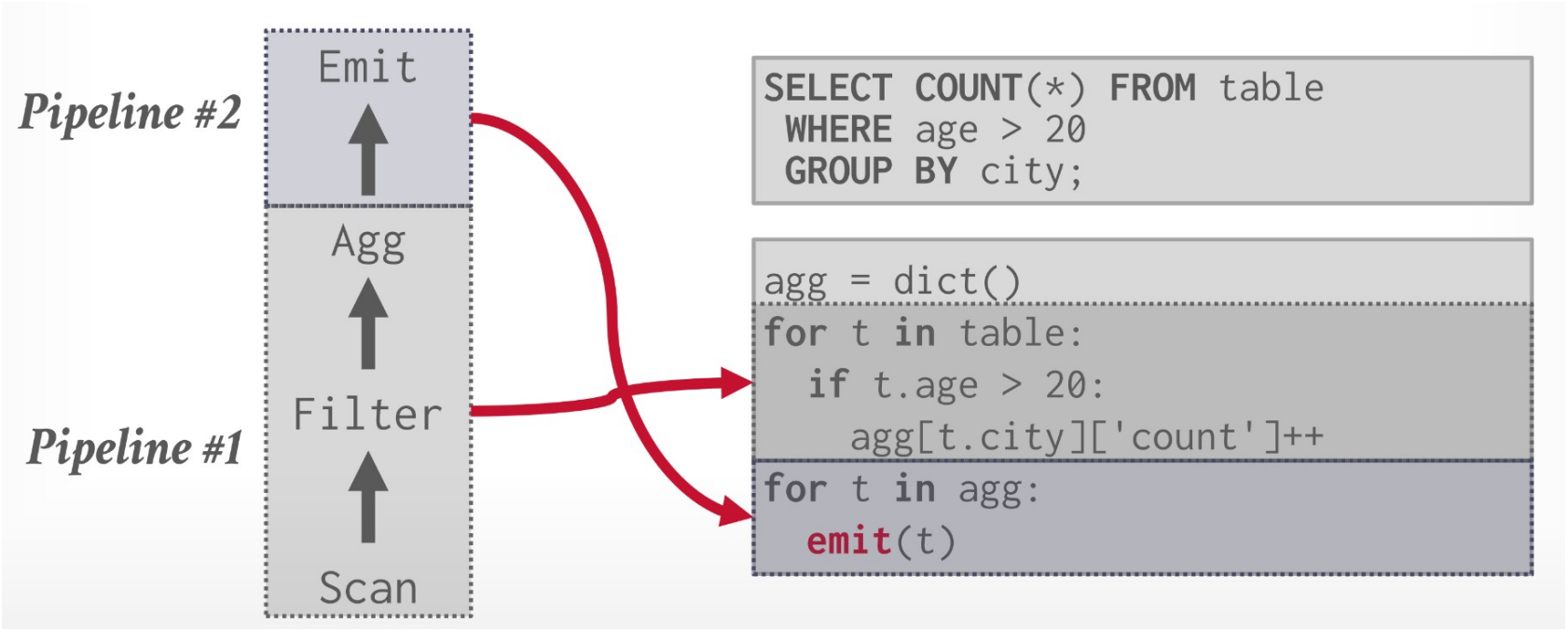
For each batch, the SIMD vectors may contain tuples that are no longer valid (they were disqualified by some previous check).



Menon-et.al.-Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last-VLDB-2017

OBSERVATION

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



RELAXED OPERATOR FUSION

Vectorized processing model designed for query compilation execution engines.

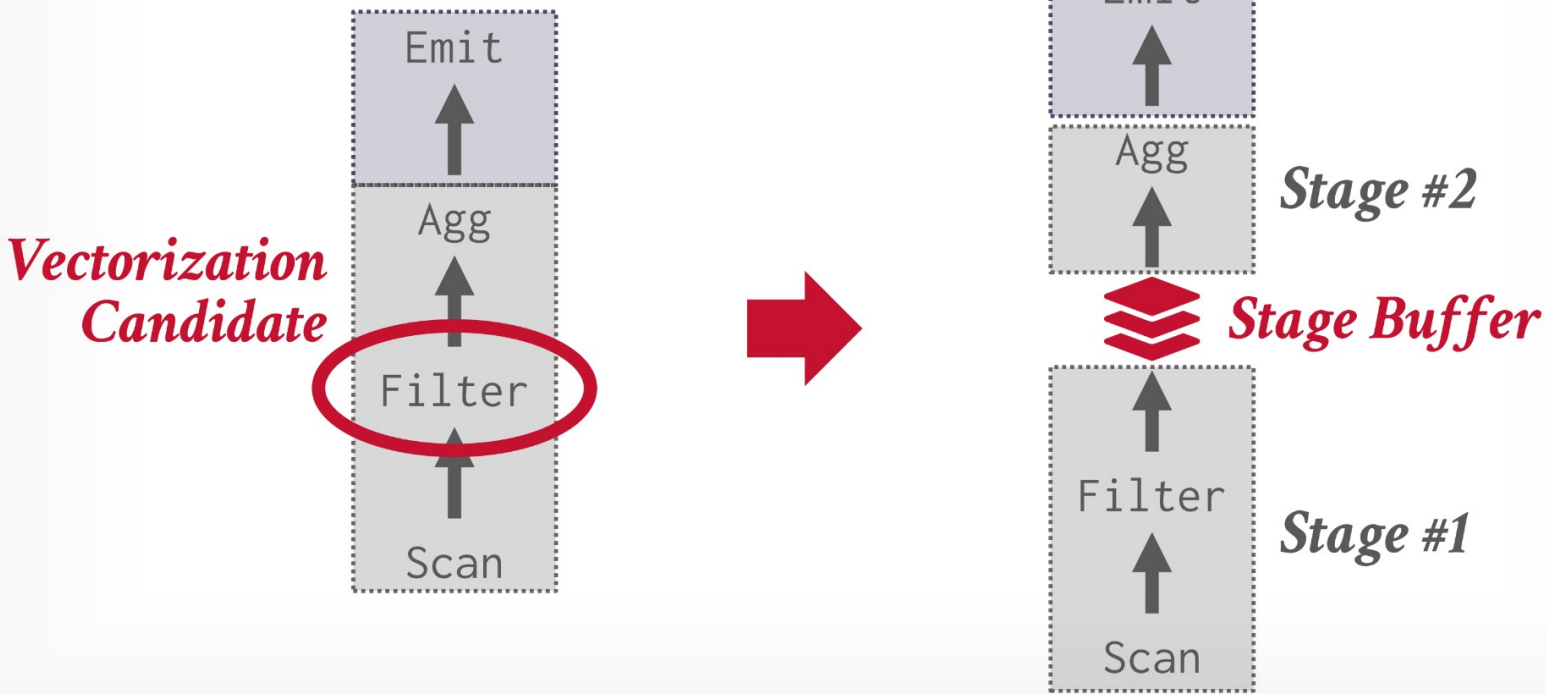
Decompose pipelines into **stages** that operate on vectors of tuples.

- Each stage may contain multiple operators.
- Communicate through cache-resident buffers.
- Stages are granularity of vectorization + fusion.

Menon-et.al.-Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last-VLDB-2017

ROF EXAMPLE

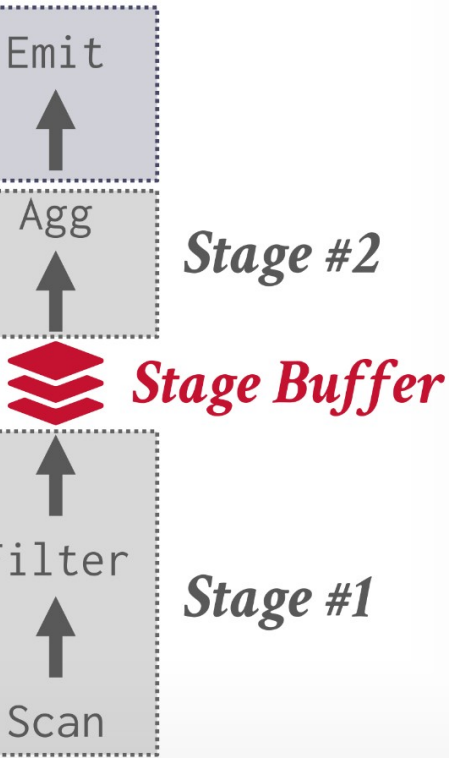
```
SELECT COUNT(*) FROM table
 WHERE age > 20 GROUP BY city;
```



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 SOFTWARE PREFETCHING

The DBMS can tell the CPU to grab the next vector while it works on the current batch.

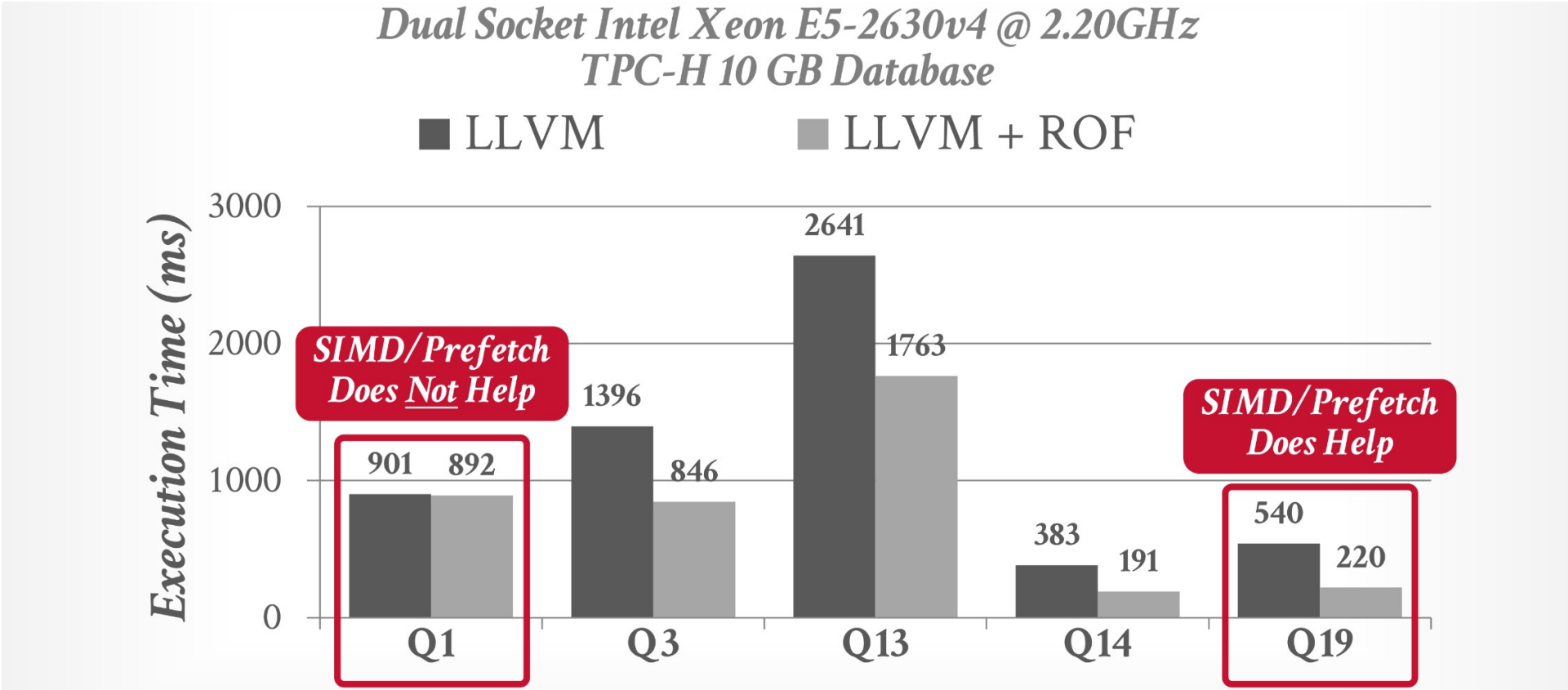
- Prefetch-enabled operators define start of new stage.
- Hides the cache miss latency.

Any prefetching technique is suitable

- Group prefetching, software pipelining, AMAC.
- Group prefetching works and is simple to implement.

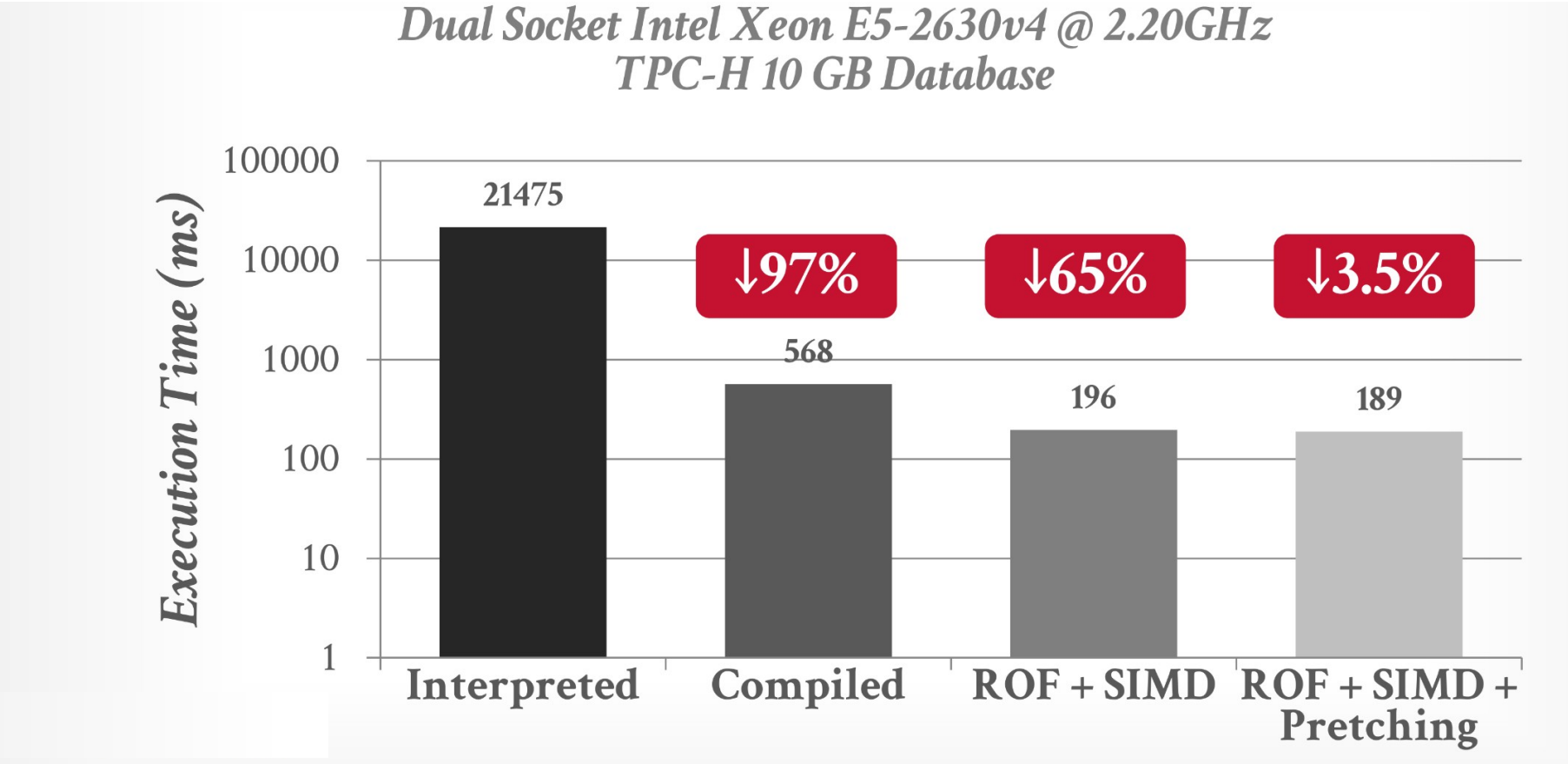
Menon-et.al.-Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last-VLDB-2017

ROF EVALUATION



Menon-et.al.-Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last-VLDB-2017

ROF EVALUATION



Menon-et.al.-Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last-VLDB-2017

VECTOR REFILL ALGORITHMS

Approach #1: Buffered

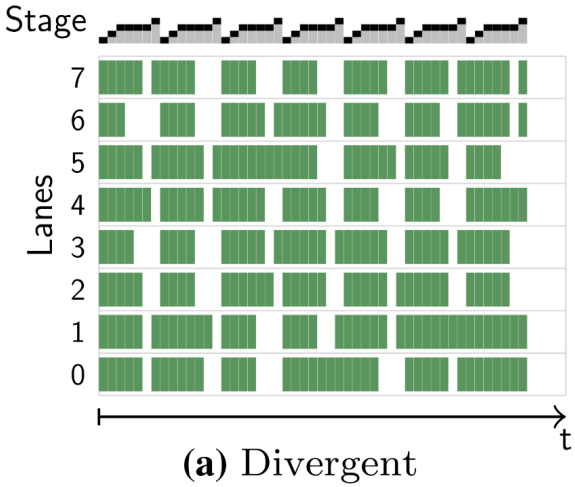
- Use additional SIMD registers to stage results within an operator and proceed with next loop iteration to fill in underutilized lanes vectors.

Approach #2: Partial

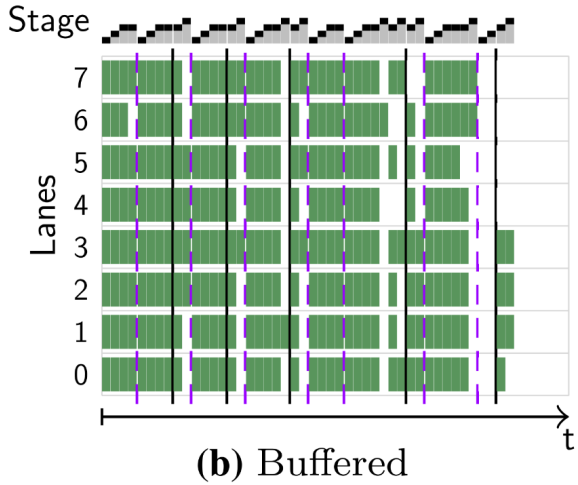
- Use additional SIMD registers to buffer results from underutilized vectors and then return to previous operator to process the next vector.
- Requires fine-grained bookkeeping to make sure other operators do not clobber deferred vectors.

Lang-et al.-Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines-vldbj2020

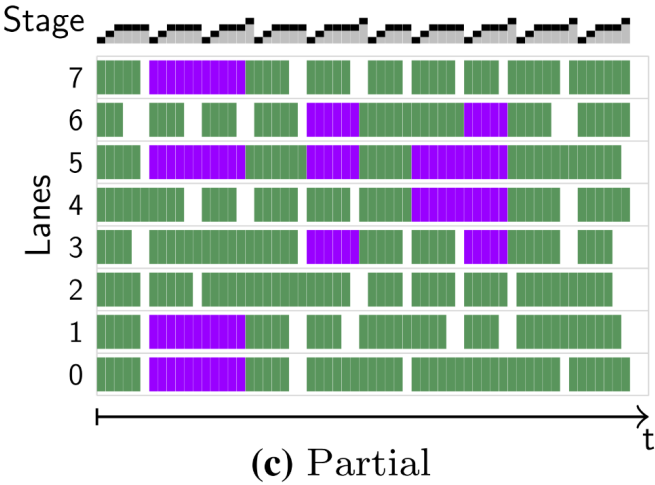
VECTOR REFILL ALGORITHMS



(a) Divergent



(b) Buffered



(c) Partial

Lang-et al.-Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines-vldbj2020

VECTORIZED OPERATORS

~~Selection Scans~~

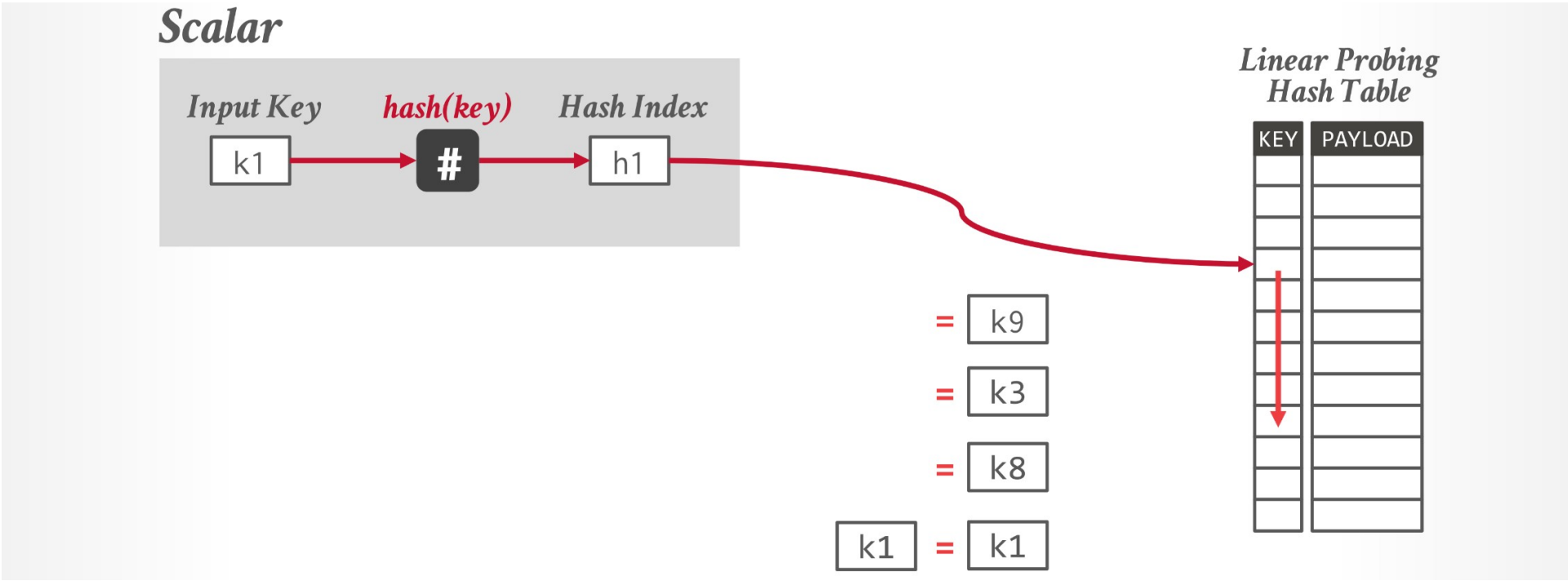
~~ROF~~

~~Vector Refill~~

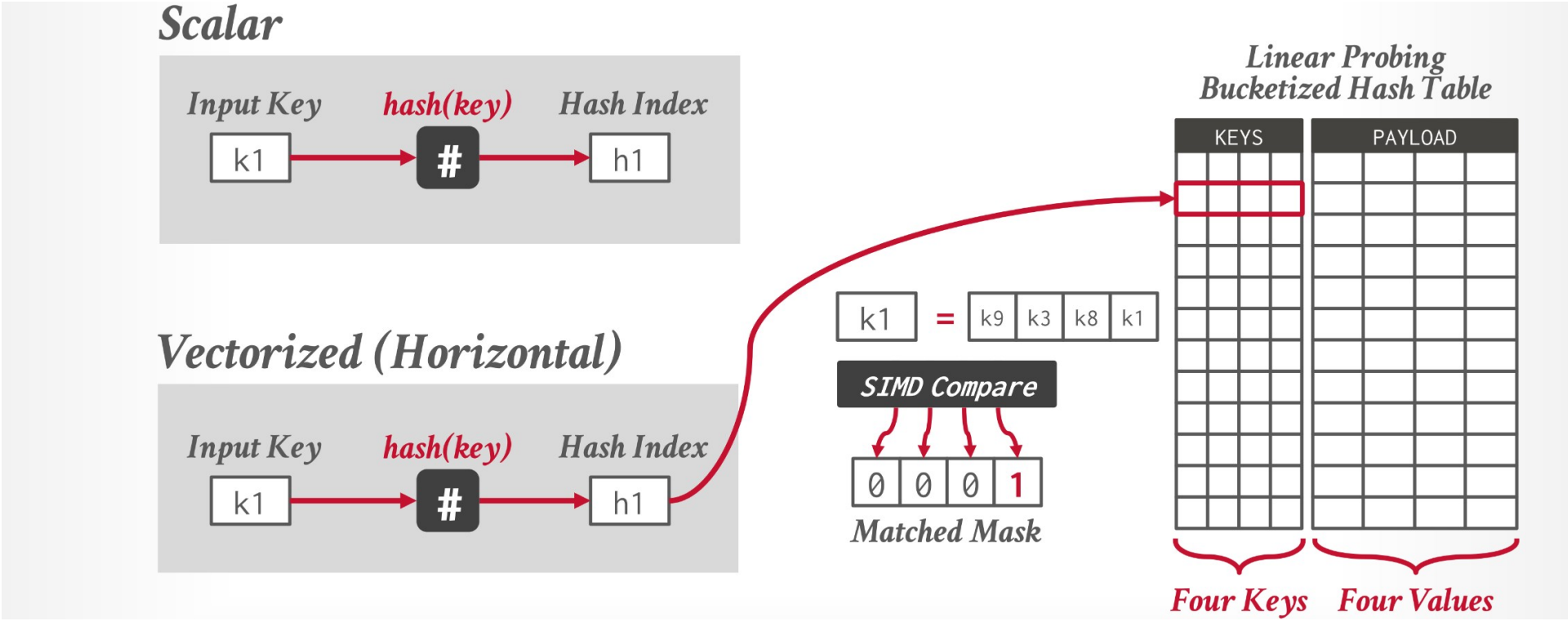
Hash Tables

Partitioning / Histograms

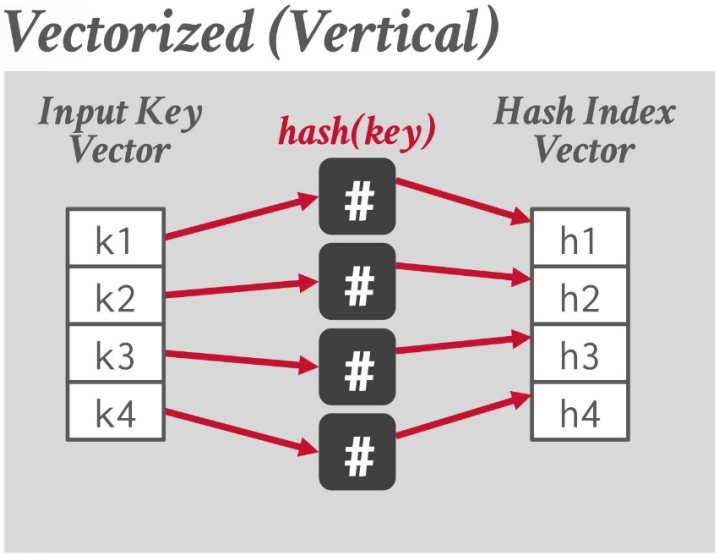
HASH TABLES – PROBING

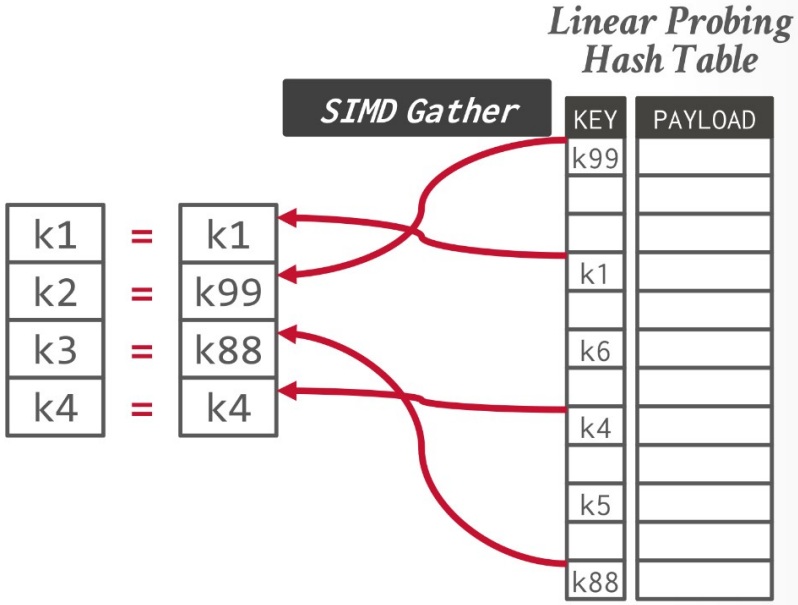


HASH TABLES – PROBING

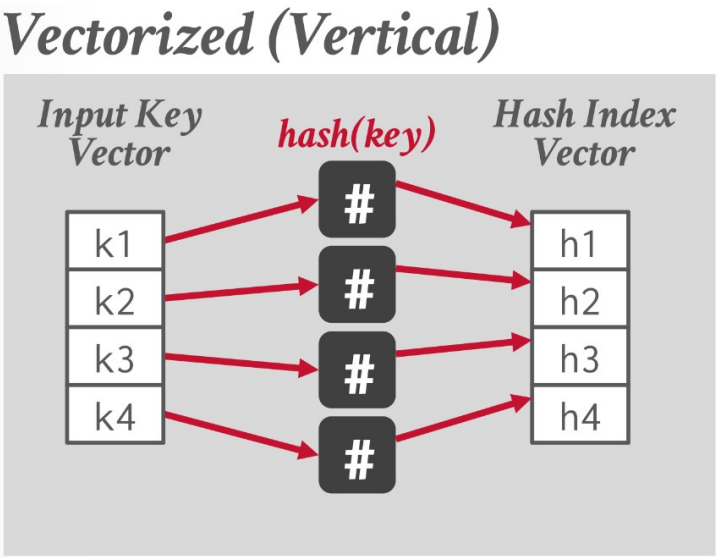


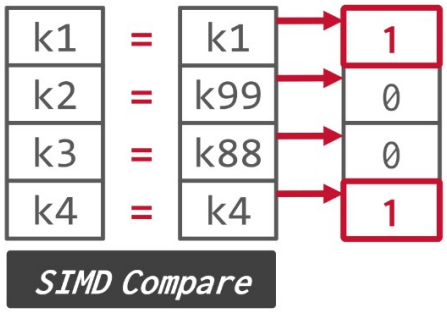
HASH TABLES – PROBING

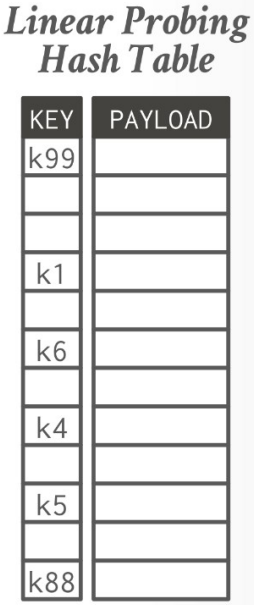




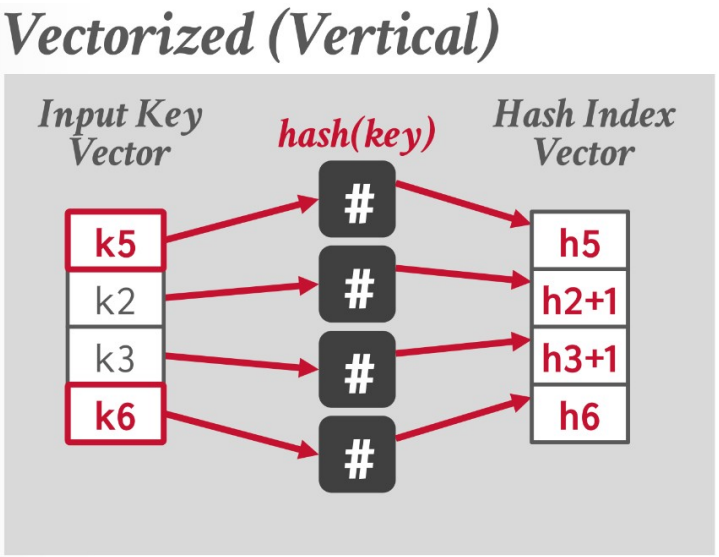
HASH TABLES – PROBING

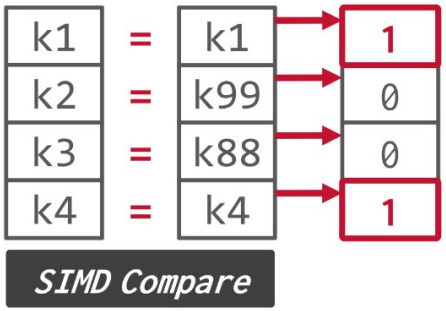


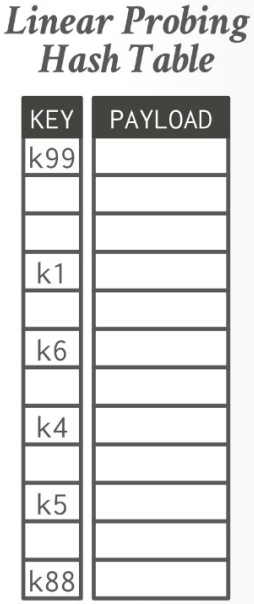




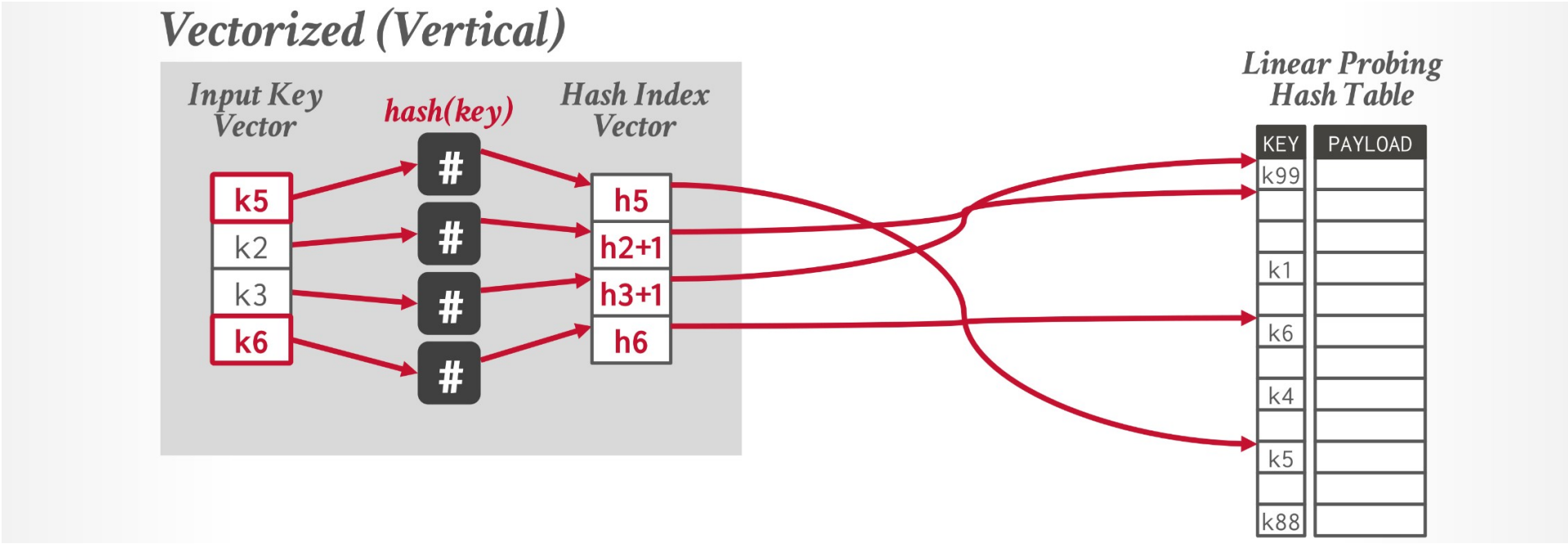
HASH TABLES – PROBING







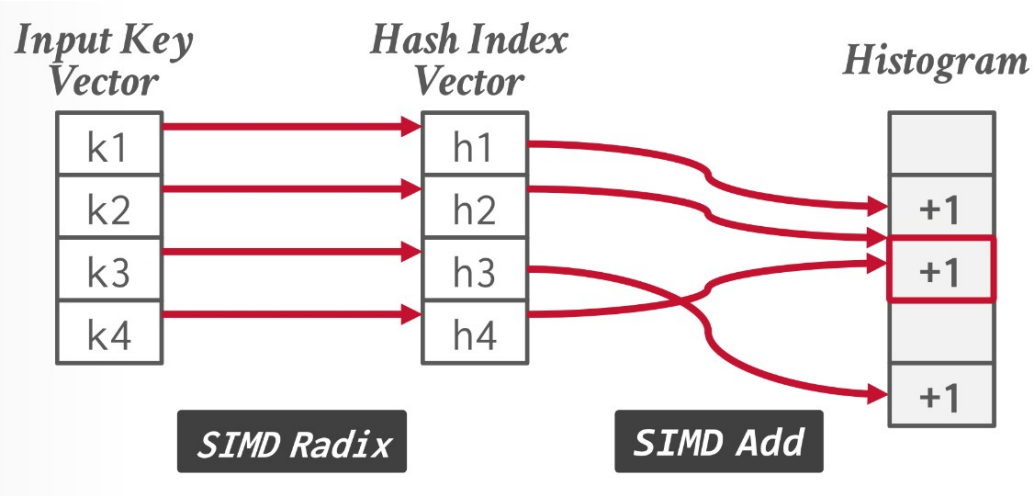
HASH TABLES – PROBING



PARTITIONING – HISTOGRAM

Use scatter and gathers to increment counts.

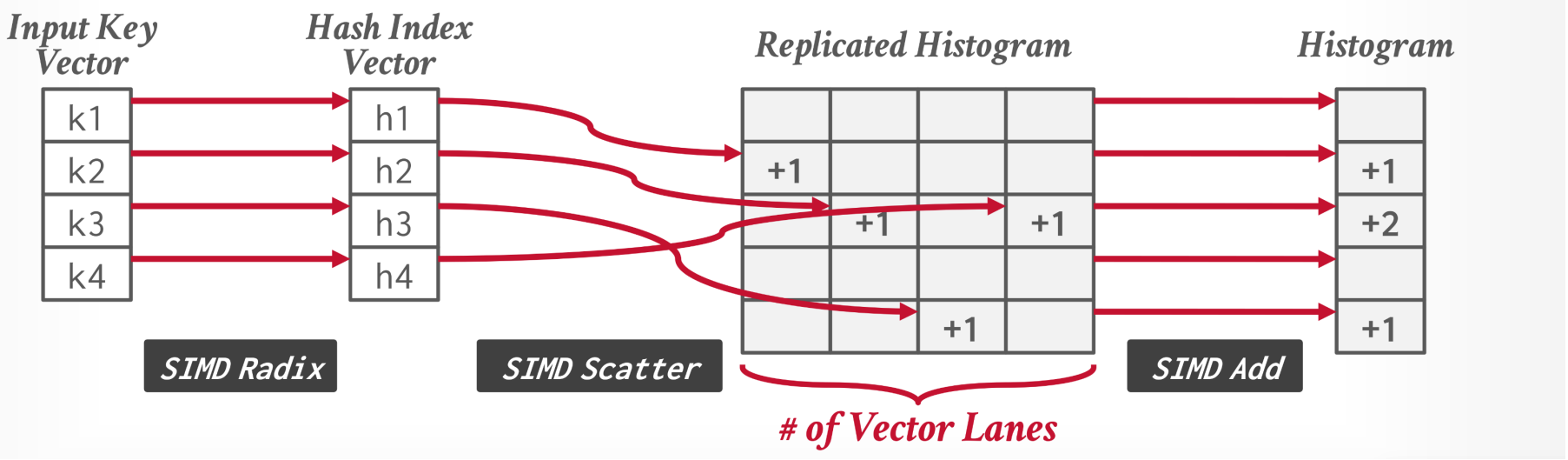
Replicate the histogram to handle collisions.



PARTITIONING – HISTOGRAM

Use scatter and gathers to increment counts.

Replicate the histogram to handle collisions.



Probems with AVX-512

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.

PARTING THOUGHTS

Vectorization is essential for OLAP queries.

But implementing an algorithm using SIMD is still mostly a manual process.

- Problems: a) algorithms? b) inefficient hardware?

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

- Join algorithms
- Using new hardware for accelerating join algorithms
- Parallelizing sort with SIMD
- Hash vs. Sort-based algorithms