Electronics and Telecommunications
Research Institute

SIMD Vectorization

Rethinking SIMD Vectorization for In-Memory Databases

Sung-Soo Kim

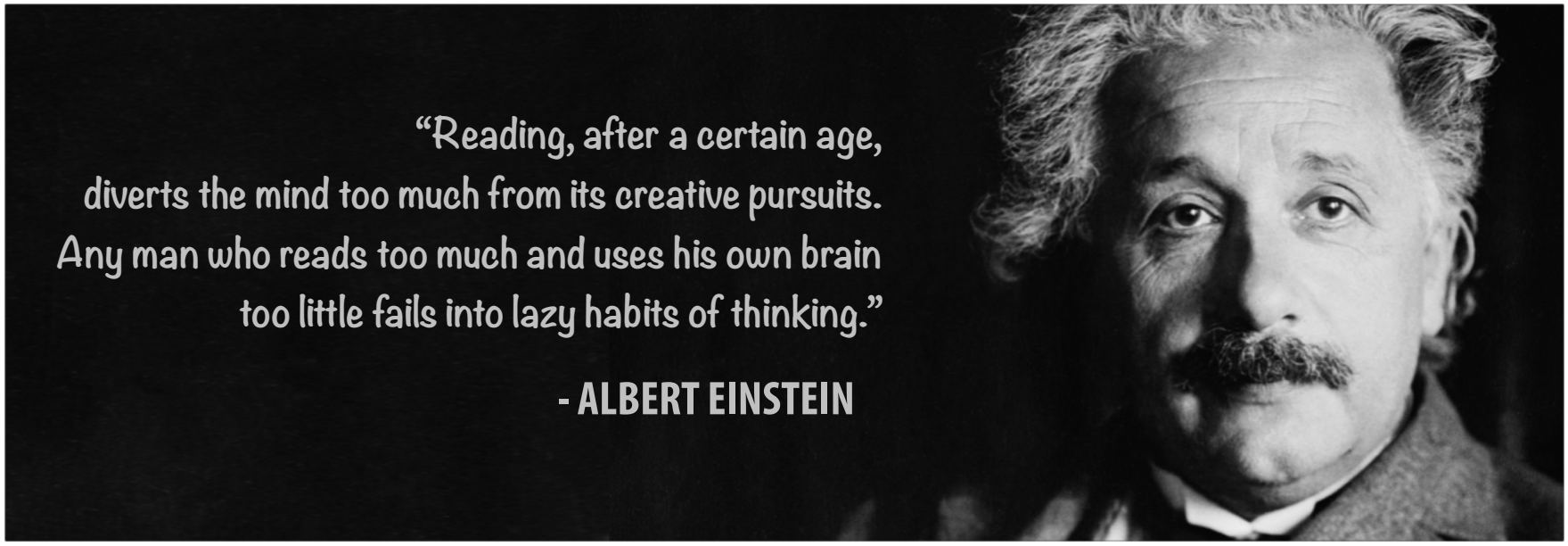
sungsoo@etri.re.kr

Data Platform Research Section

ETRI

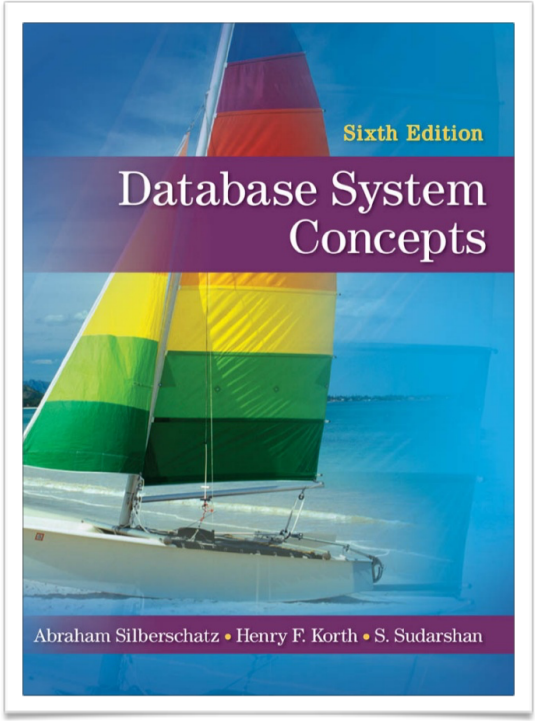
Outline

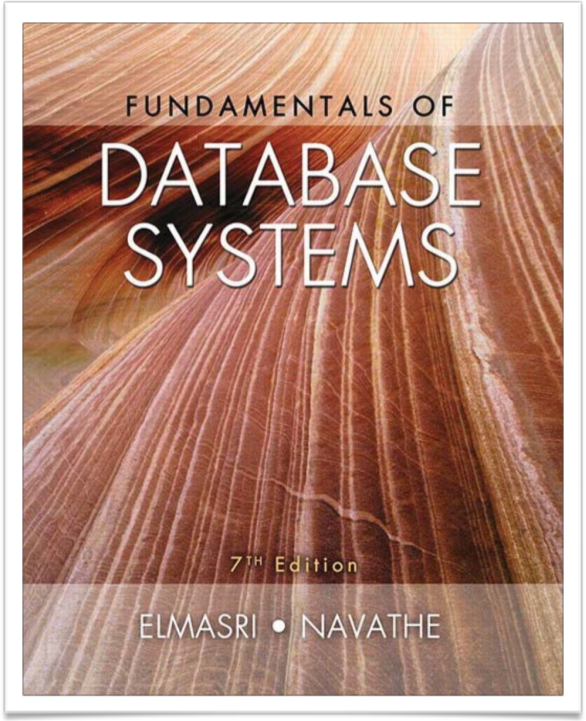
- Background
- SIMD Vectorization Algorithms
  - Selective Load/Store, Selective Gather/Scatter
  - Selective Scans
  - Hash Tables - Probing
  - Partitioning - Histogram
- Summary



References & Slide Credits

- *Database System Concepts, 6th Edition*
- *Fundamentals of Database Systems, 7th Edition*
- **Rethinking SIMD Vectorization for In-Memory Databases, *SIGMOD 2015*.**





Rethinking SIMD Vectorization for In-Memory Databases

Orestis Polychroniou* Columbia University orestis@cs.columbia.edu

Arun Raghavan Oracle Labs arun.raghavan@oracle.com

Kenneth A. Ross† Columbia University kar@cs.columbia.edu

ABSTRACT

Analytical databases are continuously adapting to the underlying hardware in order to saturate all sources of parallelism. At the same time, hardware evolves in multiple directions to explore different trade-offs. The MIC architecture, one such example, strays from the mainstream CPU design by packing a larger number of simpler cores per chip, relying on SIMD instructions to fill the performance gap. Databases have been attempting to utilize the SIMD capabilities of CPUs. However, mainstream CPUs have only recently adopted wider SIMD registers and more advanced instructions, since they do not rely primarily on SIMD for efficiency. In this paper, we present novel vectorized designs and implementations of database operators, based on advanced SIMD operations, such as gathers and scatters. We study selections, hash tables, and partitioning; and combine them to build sorting and joins. Our evaluation on the MIC-based Xeon Phi co-processor as well as the latest mainstream CPUs shows that our vectorization designs are up to an order of magnitude faster than the state-of-the-art scalar and vector approaches. Also, we highlight the impact of efficient vectorization on the algorithmic design of in-memory database operators, as well as the architectural design and power efficiency of hardware, by making simple cores comparably fast to complex cores. This work is applicable to CPUs and co-processors with advanced SIMD capabilities, using either many simple cores or fewer complex cores.

1. INTRODUCTION

Real time analytics are the steering wheels of big data driven business intelligence. Database customer needs have extended beyond OLTP with high ACID transaction throughput, to interactive OLAP query execution across the entire database. As a consequence, vendors offer fast OLAP solutions, either by rebuilding a new DBMS for OLAP [28, 37], or by improving within the existing OLTP-focused DBMS.

*Work partly done when first author was at the Oracle Labs.
†Supported by NSF grant IIS-1422488 and an Oracle gift.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
Copyright 2015 ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2747645.

The advent of large main-memory capacity is one of the reasons that blink-of-an-eye analytical query execution has become possible. Query optimization used to measure blocks fetched from disk as the primary unit of query cost. Today, the entire database can often remain main-memory resident and the need for efficient in-memory algorithms is apparent.

The prevalent shift in database design for the new era are column stores [19, 28, 37]. They allow for higher data compression in order to reduce the data footprint, minimize the number of columns accessed per tuple, and use column oriented execution coupled with late materialization [9] to eliminate unnecessary accesses to RAM resident columns.

Hardware provides performance through three sources of parallelism: *thread parallelism*, *instruction level parallelism*, and *data parallelism*. Analytical databases have evolved to take advantage of all sources of parallelism. Thread parallelism is achieved, for individual operators, by splitting the input equally among threads [3, 4, 5, 8, 14, 31, 40], and in the case of queries that combine multiple operators, by using the pipeline breaking points of the query plan to split the materialized data in chunks that are distributed to threads dynamically [18, 28]. Instruction level parallelism is achieved by applying the same operation to a block of tuples [6] and by compiling into tight machine code [16, 22]. Data parallelism is achieved by implementing each operator to use SIMD instructions effectively [7, 15, 26, 30, 39, 41].

The different sources of parallelism were developed as a means to deliver more performance within the same power budget available per chip. Mainstream CPUs have evolved on all sources of parallelism, featuring massively superscalar pipelines, out-of-order execution of tens of instructions, and advanced SIMD capabilities, all replicated on multiple cores per CPU chip. For example, the latest Intel Haswell architecture issues up to 8 micro-operations per cycle with 192 reorder buffer entries for out-of-order execution, 256-bit SIMD instructions, two levels of private caches per core and a large shared cache, and scales up to 18 cores per chip.

Concurrently with the evolution of mainstream CPUs, a new approach on processor design has surfaced. The design, named the many-integrated-cores [MIC] architecture, uses cores with a smaller area (transistors) and power footprint by removing the massively superscalar pipeline, out-of-order execution, and the large L3 cache. Each core is based on a Pentium 1 processor with a simple in-order pipeline, but is augmented with large SIMD registers, advanced SIMD instructions, and simultaneous multithreading to hide load and instruction latency. Since each core has a smaller area and power footprint, more cores can be packed in the chip.

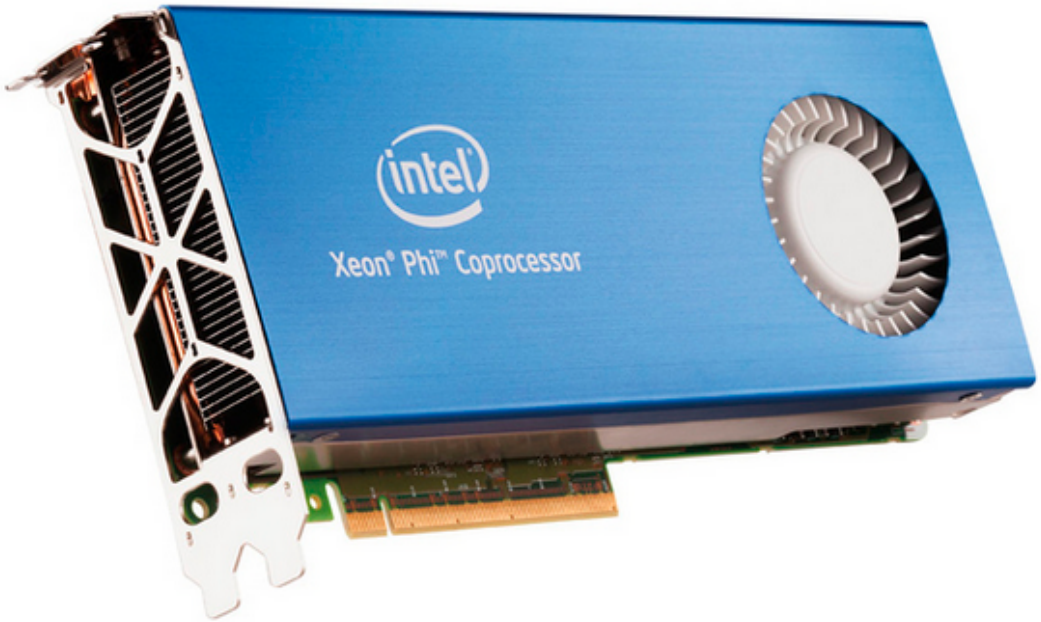
Multi-Core CPUs

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

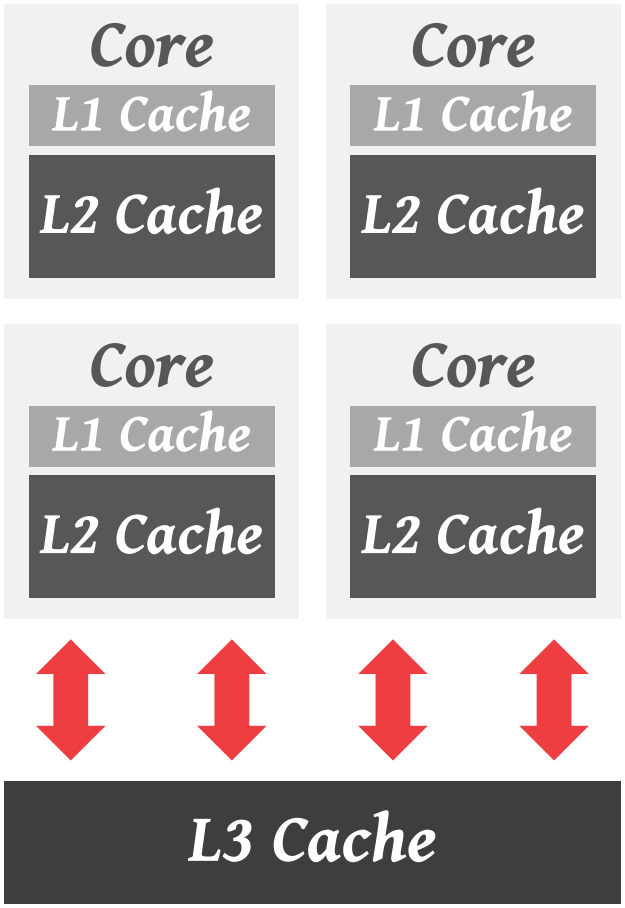
- Massively superscalar and aggressive out-of-order execution
  - Instructions are issued from a *sequential* stream.
  - Check for *dependencies* between instructions.
  - Process *multiple instructions* per clock cycle.

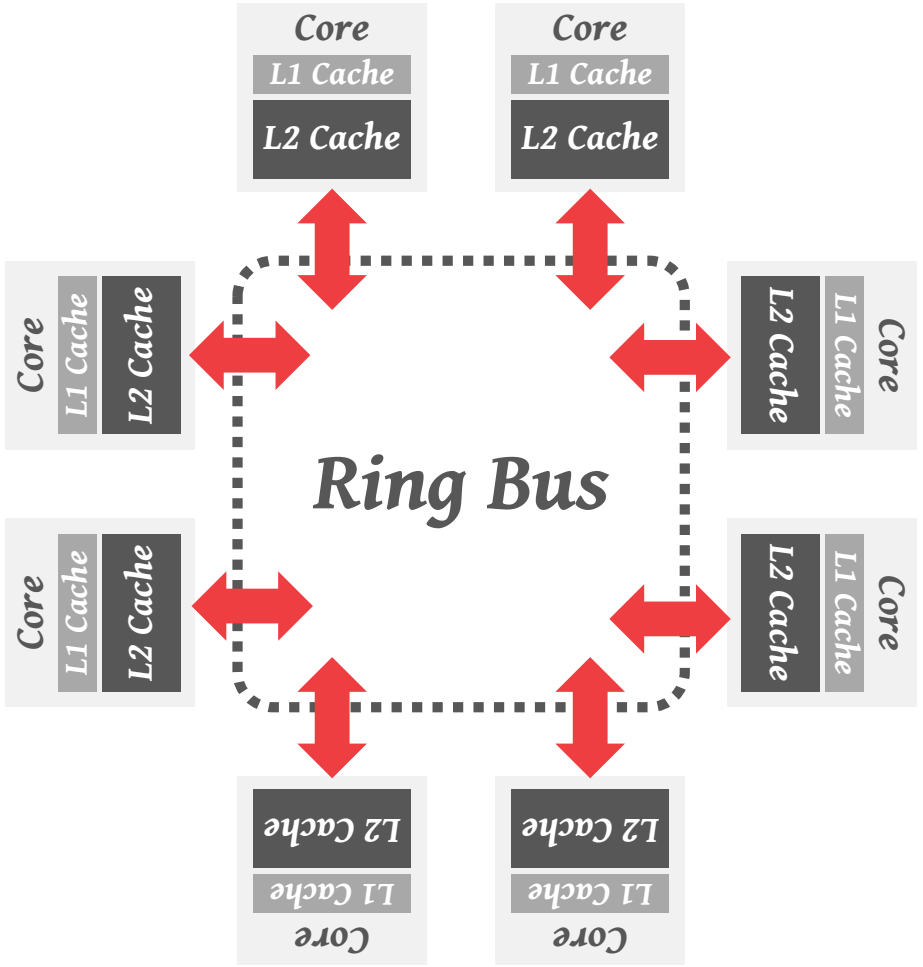
Many Integrated Cores (MIC)

- 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.



Multi-Core vs. MIC





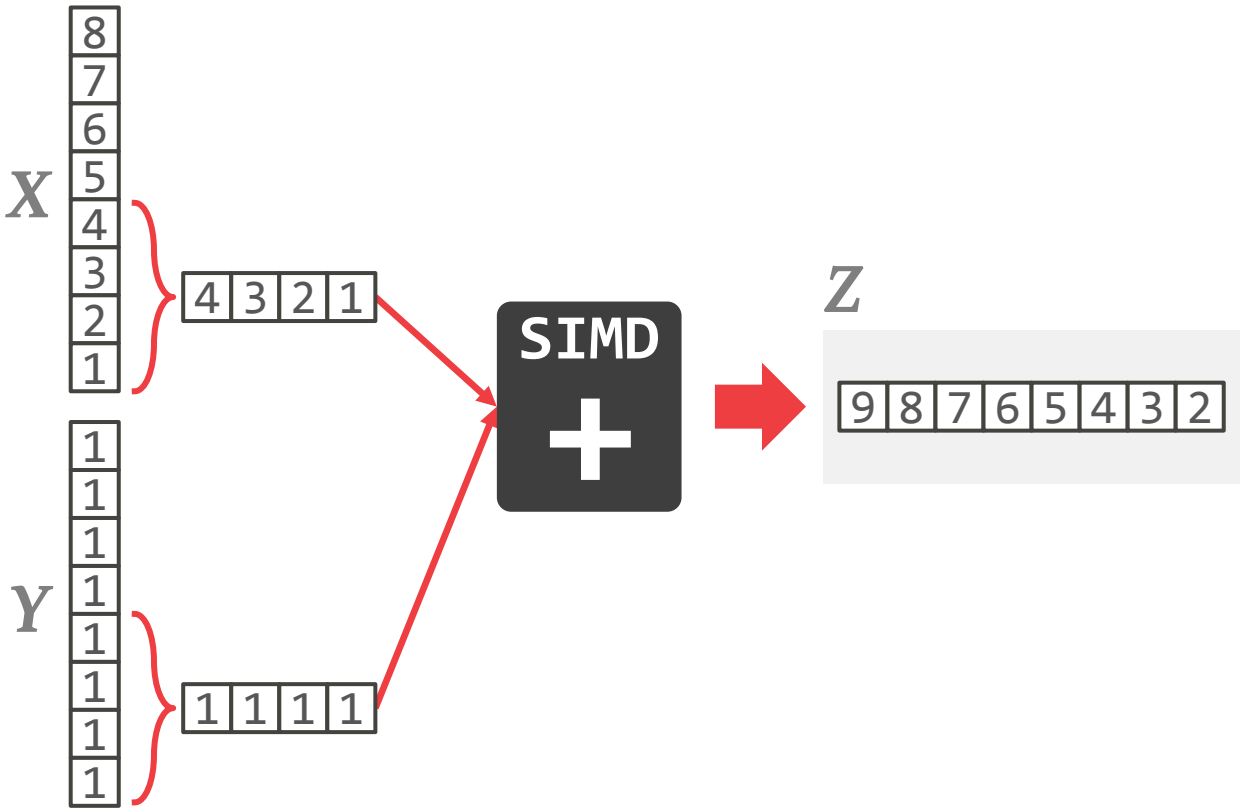
Vectorization

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

$X + Y = Z$

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

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



Automatic Vectorization

- The compiler can identify when instructions inside of a loop can be *rewritten* as a *vectorized* operation.
  - e.g., Intel ICC compiler

- Works for simple loops only and is rare in database operators.

- Requires hardware support for SIMD instructions.

Manual Vectorization

- **Linear Access Operators**
  - Predicative evaluation
  - Compression
- **Ad-hoc Vectorization**
  - Sorting
  - Merging
    - e.g., Sort-Merge Join
- **Composable Operations**
  - Multi-way trees
  - Bucketized hash tables

Single Instruction, Multiple Data (SIMD)

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

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

SIMD Example

$X + Y = Z$

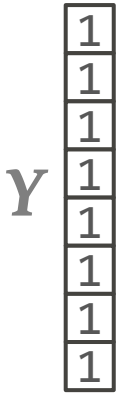
$$\begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \dots \\ x_n + y_n \end{pmatrix}$$

SIMD Example

$X + Y = Z$

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

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



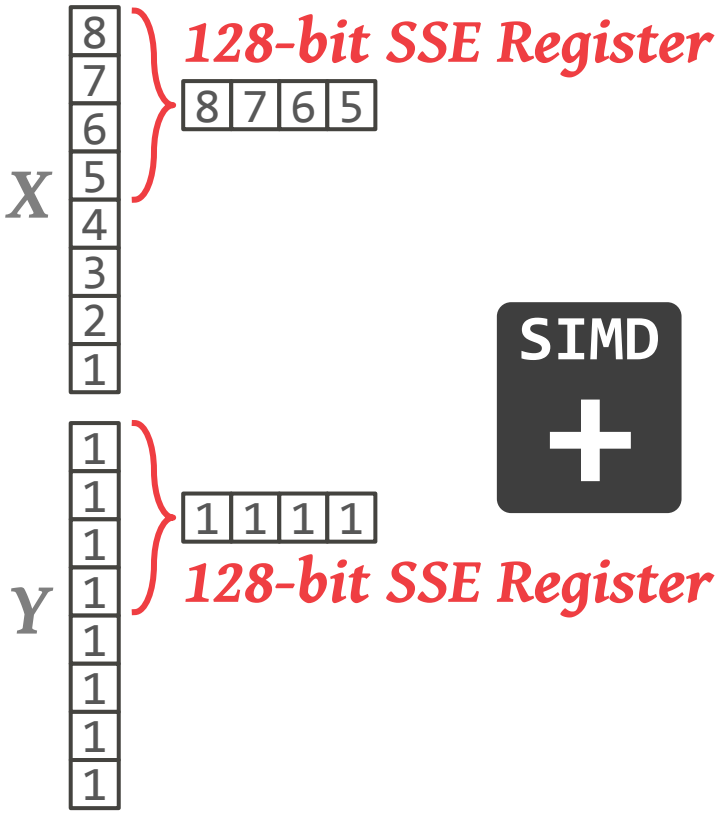
Z

SIMD Example

$X + Y = Z$

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

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

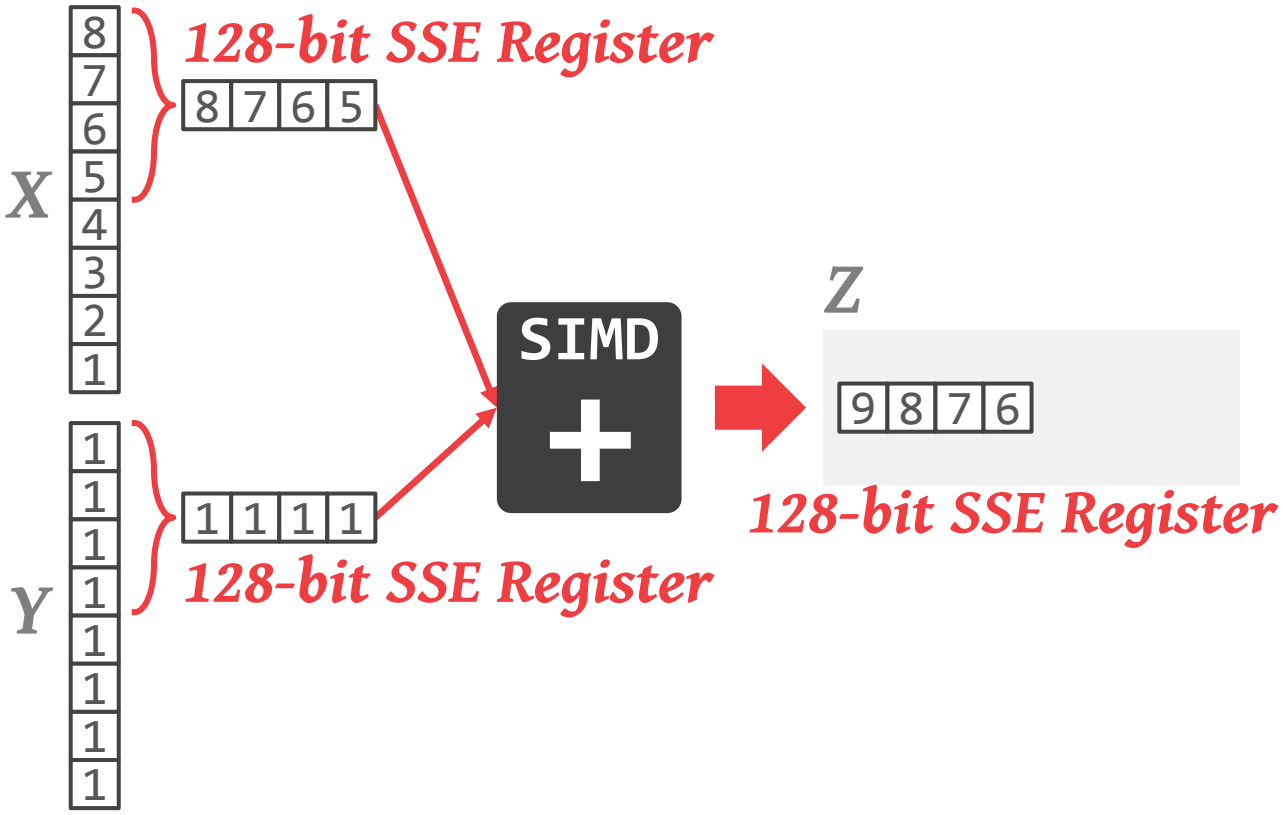


SIMD Example

$X + Y = Z$

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

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

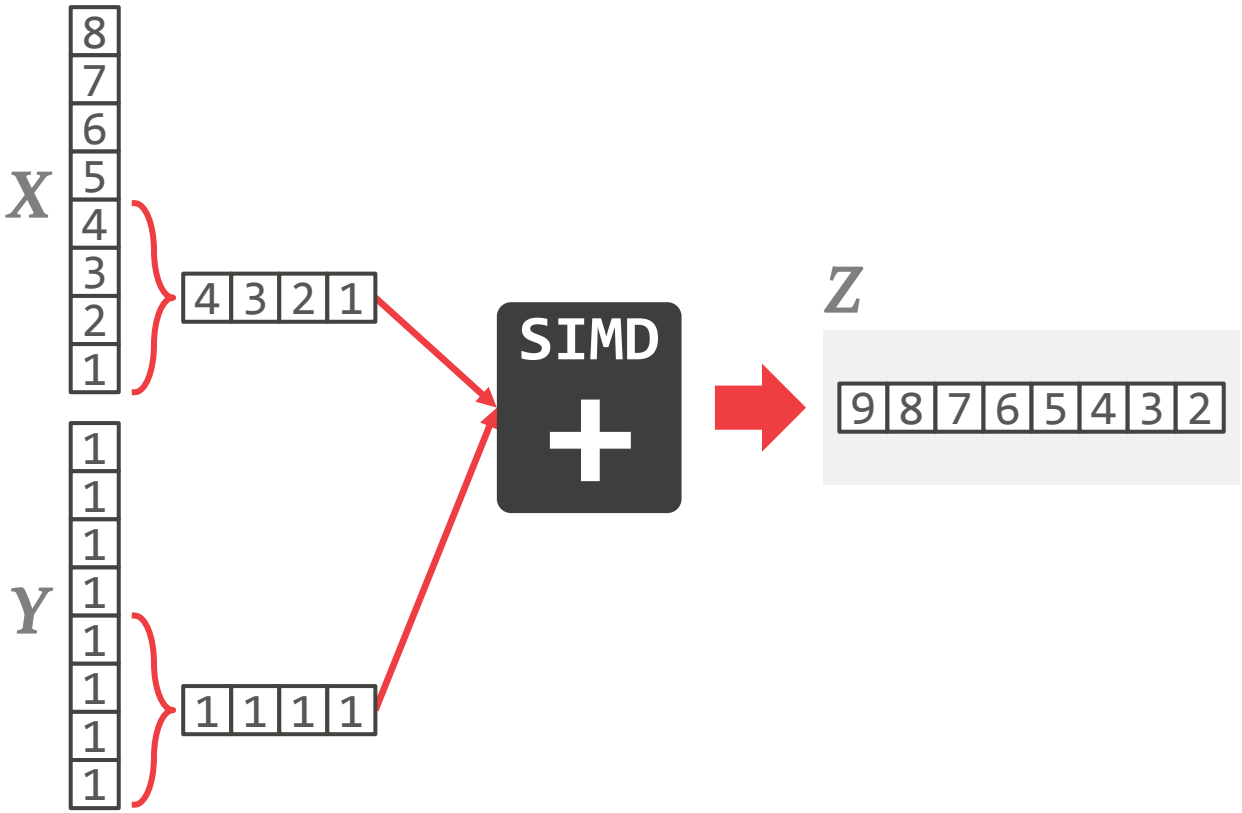


SIMD Example

$X + Y = Z$

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

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



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 four elements simultaneously.

- First introduced by Intel in 1999.

SSE Instructions

- **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
- **Comparison Instructions**
  - Comparing multiple data items (==, <, <=, >, >=, !=)
- **Shuffle Instructions**
  - Move data in between SIMD registers
- **Miscellaneous**
  - Conversion: Transform data between x86 and SIMD
  - Cache Control: Move data directly from SIMD registers to memory (bypassing CPU cache.)

Vectorized DBMS Algorithms

- **Principles for efficient vectorization by using *fundamental* vector *operations* to construct more advanced functionality.**
  - **Favor *vertical vectorization* by processing different input data per lane.**
  - **Maximize *lane utilization* by executing different things per lane subset.**

Rethinking SIMD Vectorization for In-Memory Databases

Orestis Polychroniou* Columbia University orestis@cs.columbia.edu

Arun Raghavan Oracle Labs arun.raghavan@oracle.com

Kenneth A. Ross* Columbia University kar@cs.columbia.edu

ABSTRACT

Analytical databases are continuously adapting to the underlying hardware in order to saturate all sources of parallelism. At the same time, hardware evolves in multiple directions to explore different trade-offs. The MIC architecture, one such example, strays from the mainstream CPU design by packing a larger number of simpler cores per chip, relying on SIMD instructions to fill the performance gap. Databases have been attempting to utilize the SIMD capabilities of CPUs. However, mainstream CPUs have only recently adopted wider SIMD registers and more advanced instructions, since they do not rely primarily on SIMD for efficiency. In this paper, we present novel vectorized designs and implementations of database operators, based on advanced SIMD operations, such as gathers and scatters. We study selections, hash tables, and partitioning; and combine them to build sorting and joins. Our evaluation on the MIC-based Xeon Phi co-processor as well as the latest mainstream CPUs shows that our vectorization designs are up to an order of magnitude faster than the state-of-the-art scalar and vector approaches. Also, we highlight the impact of efficient vectorization on the algorithmic design of in-memory database operators, as well as the architectural design and power efficiency of hardware, by making simple cores comparably fast to complex cores. This work is applicable to CPUs and co-processors with advanced SIMD capabilities, using either many simple cores or fewer complex cores.

1. INTRODUCTION

Real time analytics are the steering wheels of big data driven business intelligence. Database customer needs have extended beyond OLTP with high ACID transaction throughput, to interactive OLAP query execution across the entire database. As a consequence, vendors offer fast OLAP solutions, either by rebuilding a new DBMS for OLAP [28, 37], or by improving within the existing OLTP-focused DBMS.

*Work partly done when first author was at the Oracle Labs.
†Supported by NSF grant IIS-1422488 and an Oracle gift.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
Copyright 2015 ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2747645.

The advent of large main-memory capacity is one of the reasons that blink-of-an-eye analytical query execution has become possible. Query optimization used to measure blocks fetched from disk as the primary unit of query cost. Today, the entire database can often remain main-memory resident and the need for efficient in-memory algorithms is apparent.

The prevalent shift in database design for the new era are column stores [19, 28, 37]. They allow for higher data compression in order to reduce the data footprint, minimize the number of columns accessed per tuple, and use column oriented execution coupled with late materialization [9] to eliminate unnecessary accesses to RAM resident columns.

Hardware provides performance through three sources of parallelism: *thread parallelism*, *instruction level parallelism*, and *data parallelism*. Analytical databases have evolved to take advantage of all sources of parallelism. Thread parallelism is achieved, for individual operators, by splitting the input equally among threads [3, 4, 5, 8, 14, 31, 40], and in the case of queries that combine multiple operators, by using the pipeline breaking points of the query plan to split the materialized data in chunks that are distributed to threads dynamically [18, 26]. Instruction level parallelism is achieved by applying the same operation to a block of tuples [6] and by compiling into tight machine code [16, 22]. Data parallelism is achieved by implementing each operator to use SIMD instructions effectively [7, 15, 26, 30, 39, 41].

The different sources of parallelism were developed as a means to deliver more performance within the same power budget available per chip. Mainstream CPUs have evolved on all sources of parallelism, featuring massively superscalar pipelines, out-of-order execution of tens of instructions, and advanced SIMD capabilities, all replicated on multiple cores per CPU chip. For example, the latest Intel Haswell architecture issues up to 8 micro-operations per cycle with 192 reorder buffer entries for out-of-order execution, 256-bit SIMD instructions, two levels of private caches per core and a large shared cache, and scales up to 18 cores per chip.

Concurrently with the evolution of mainstream CPUs, a new approach on processor design has surfaced. The design, named the many-integrated-core (MIC) architecture, uses cores with a smaller area (transistors) and power footprint by removing the massively superscalar pipeline, out-of-order execution, and the large L3 cache. Each core is based on a Pentium 1 processor with a simple in-order pipeline, but is augmented with large SIMD registers, advanced SIMD instructions, and simultaneous multithreading to hide load and instruction latency. Since each core has a smaller area and power footprint, more cores can be packed in the chip.

RETHINKING SIMD VECTORIZATION FOR IN-MEMORY DATABASES
SIGMOD 2015

Contributions

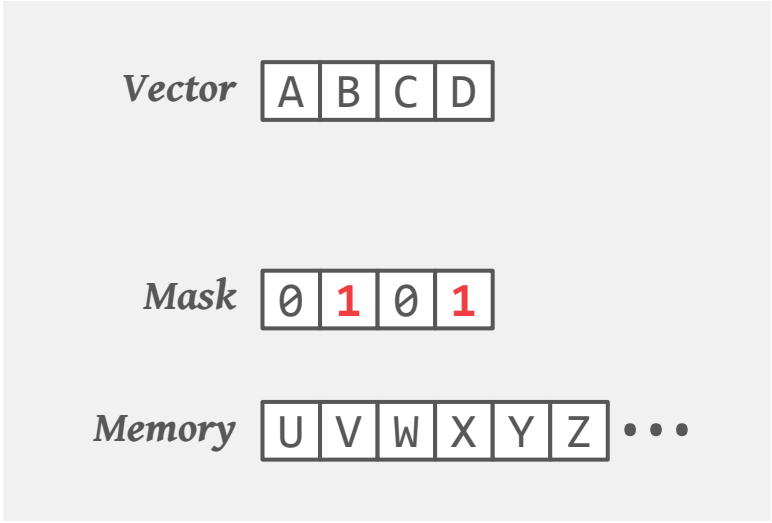
- ***Full* vectorization**
  - From *O(f(n))* scalar to *O(f(n)/W)* vector operations
    - Random accesses excluded
  - Principles for good (*efficient*) vectorization
    - Reuse fundamental *operations* across multiple vectorizations
    - Favor *vertical vectorization* by processing different input data per lane
    - Maximize *lane utilization* by executing different things per lane subset
- **Vectorize *basic* operators & build *advanced* operators (in memory)**
  - Selection scans
  - Hash Tables
  - Partitioning
  - Sorting
  - Joins
- **Show impact of *good* vectorization**
  - On *software* (database system) & *hardware* design

Fundamental Operations

- Selective *Load*
- Selective *Store*
- Selective *Gather*
- Selective *Scatter*

Fundamental Vector Operations

Selective Load

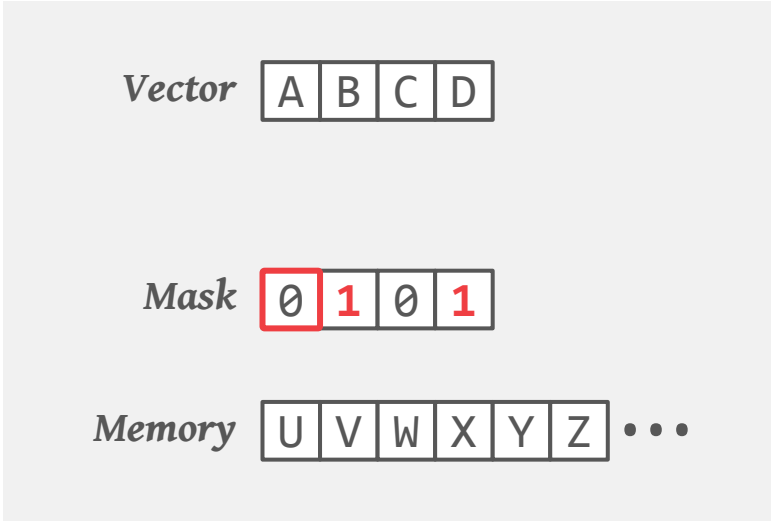


Fundamental Vector Operations

Selective Load

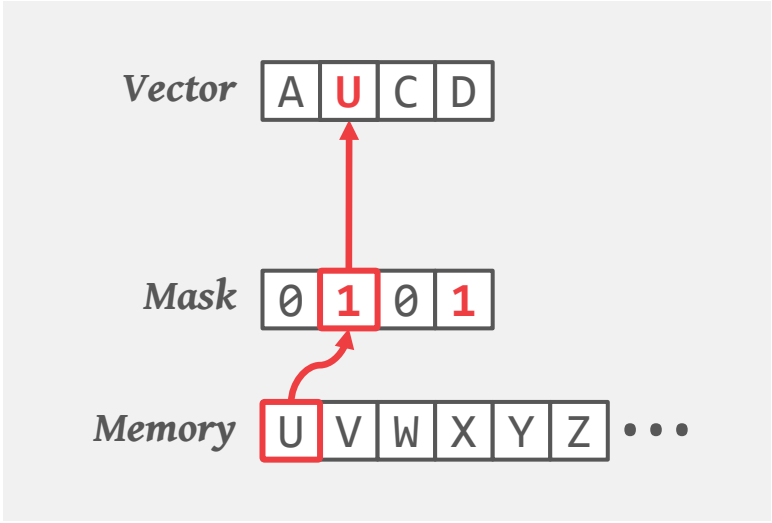
Fundamental Vector Operations

Selective Load



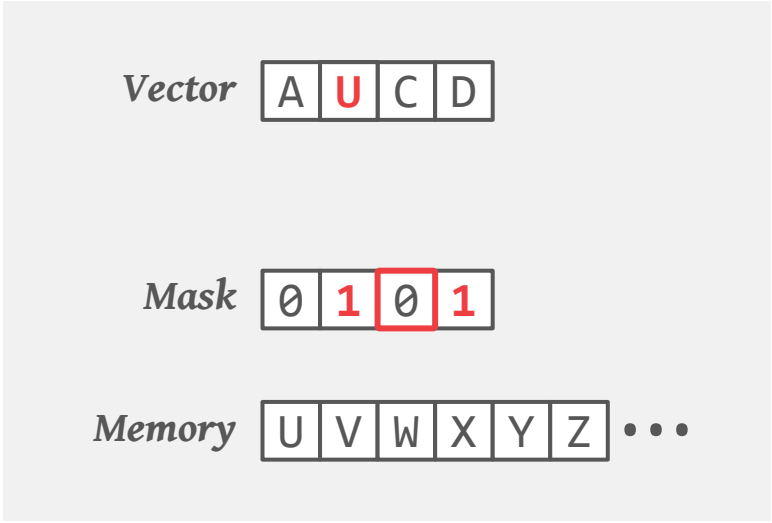
Fundamental Vector Operations

Selective Load



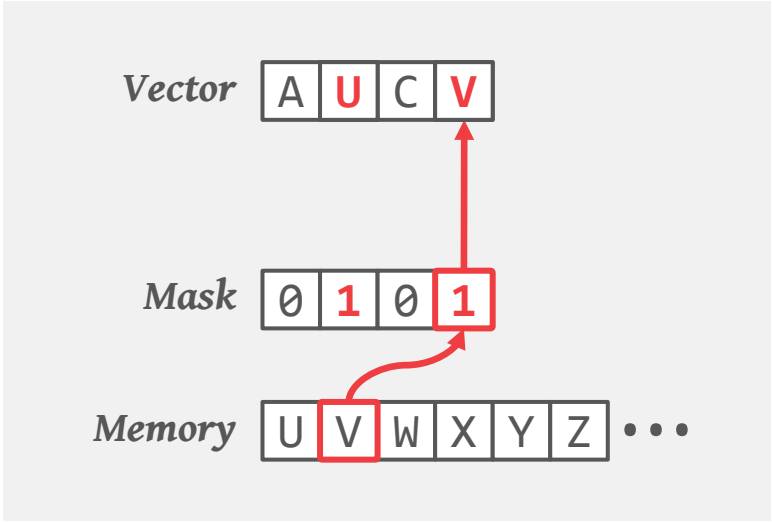
Fundamental Vector Operations

Selective Load



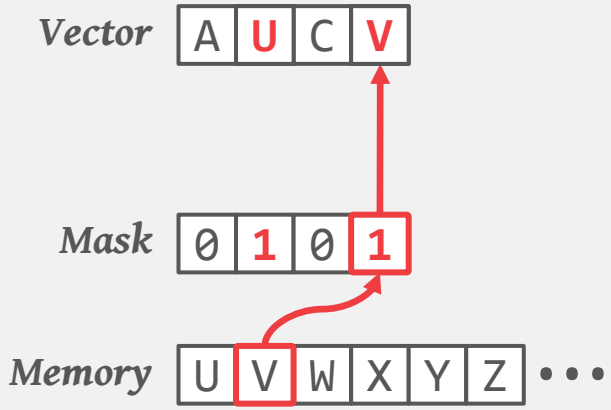
Fundamental Vector Operations

Selective Load

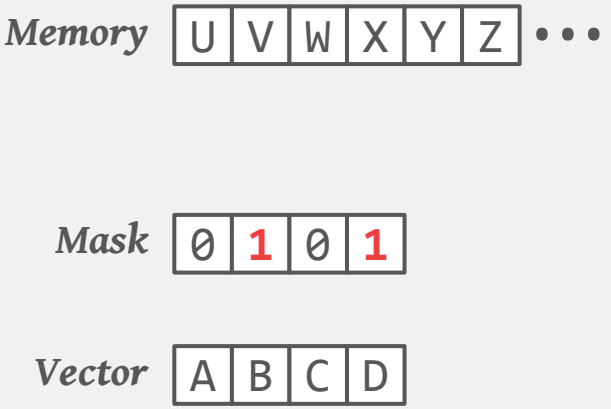


Fundamental Vector Operations

Selective Load

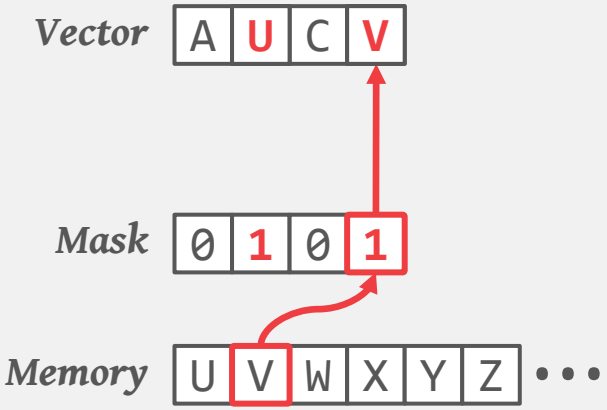


Selective Store

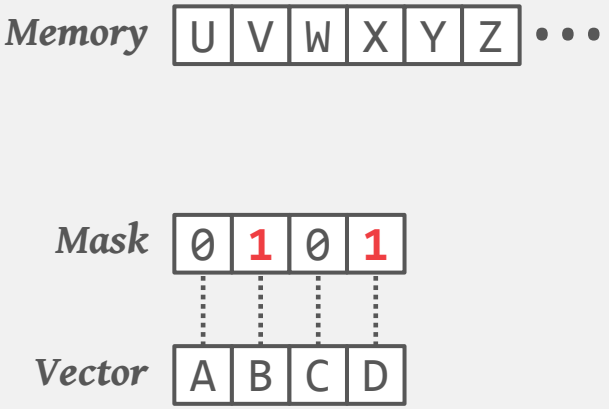


Fundamental Vector Operations

Selective Load

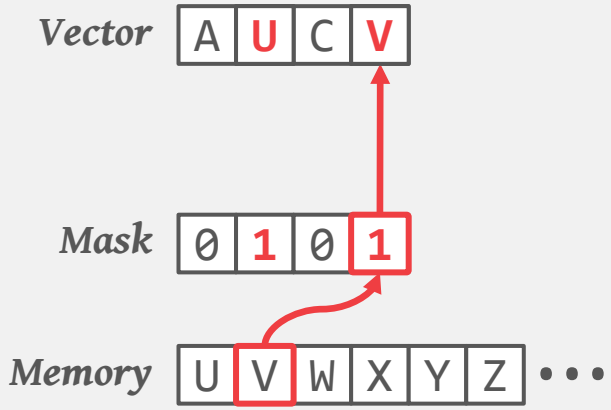


Selective Store

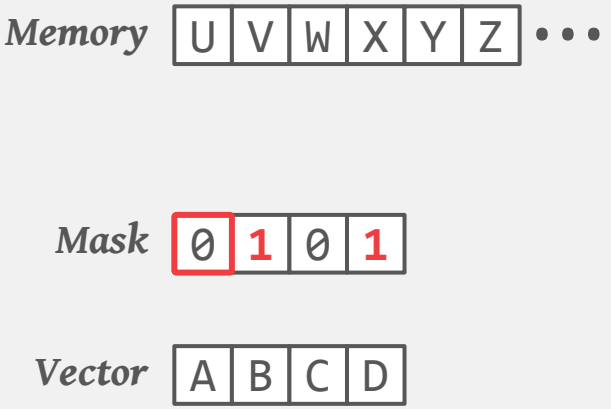


Fundamental Vector Operations

Selective Load

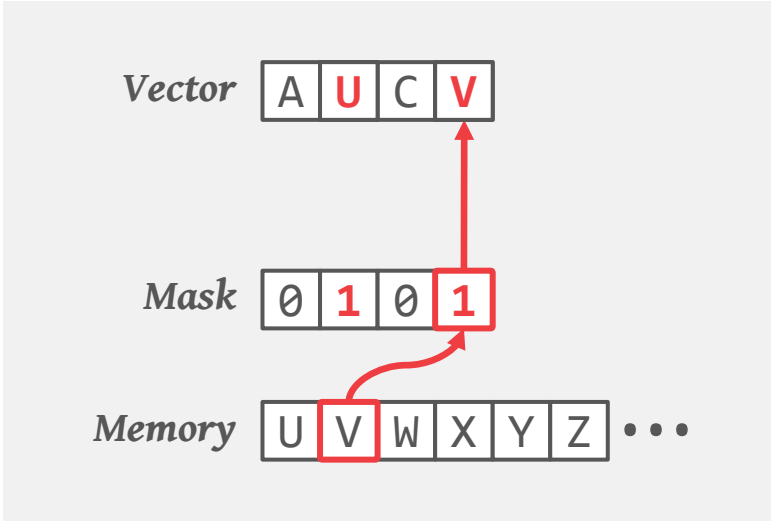


Selective Store

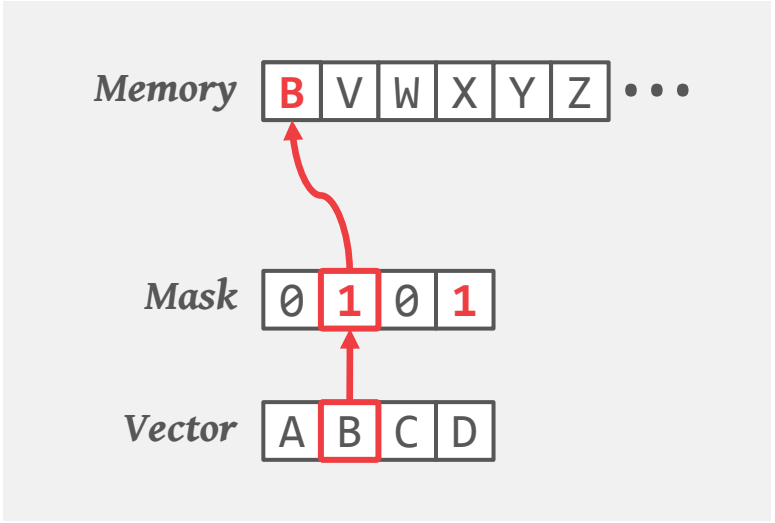


Fundamental Vector Operations

Selective Load

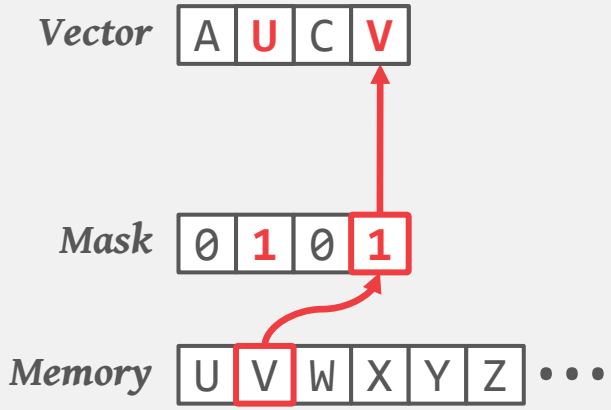


Selective Store

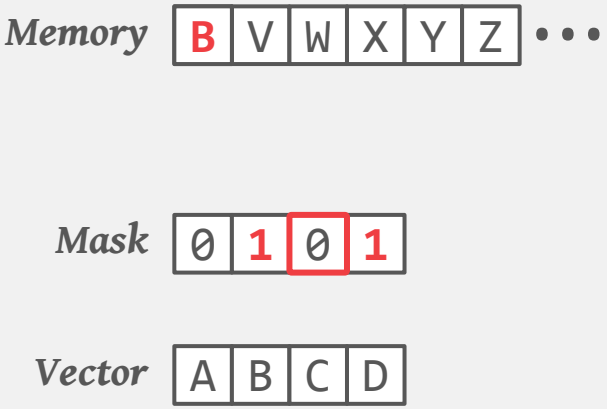


Fundamental Vector Operations

Selective Load

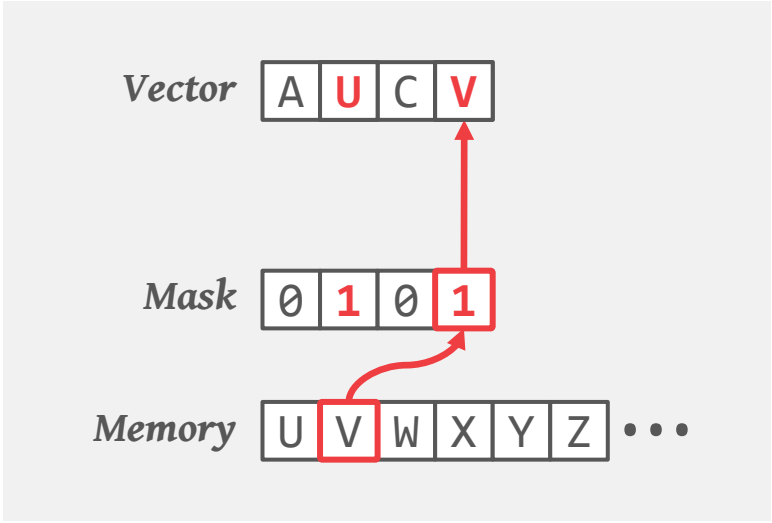


Selective Store

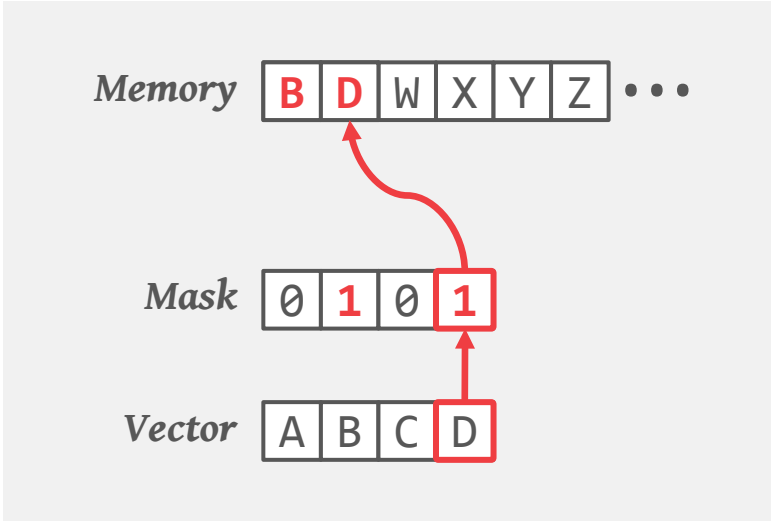


Fundamental Vector Operations

Selective Load

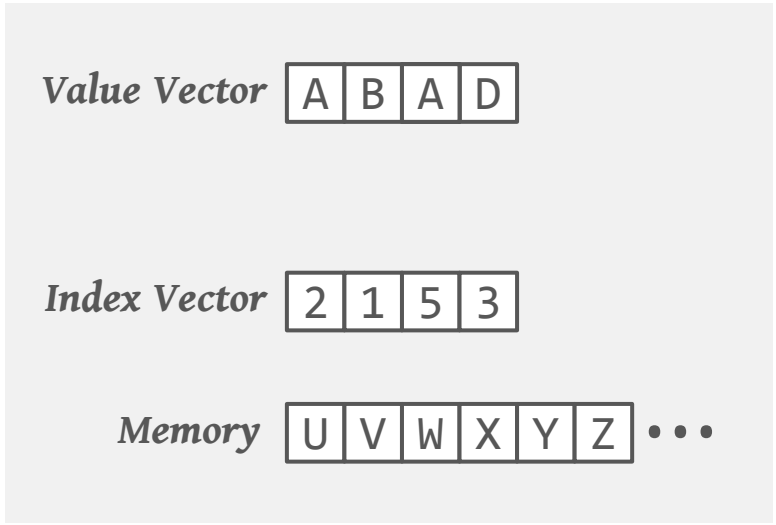


Selective Store



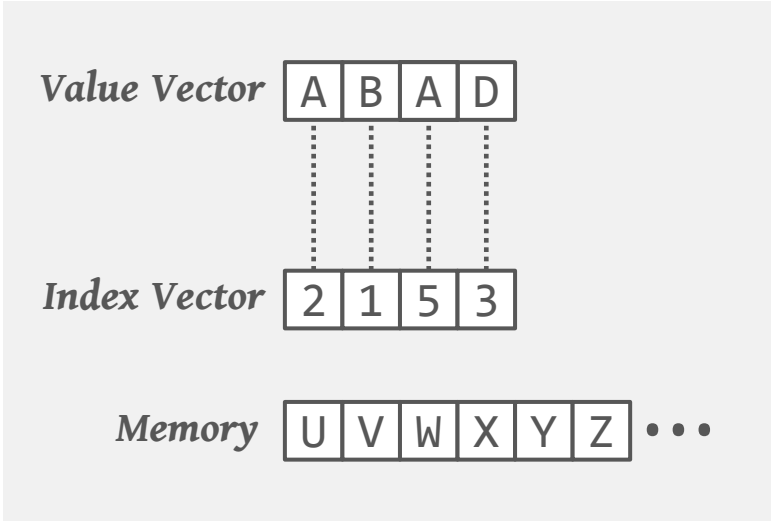
Fundamental Vector Operations

Selective Gather



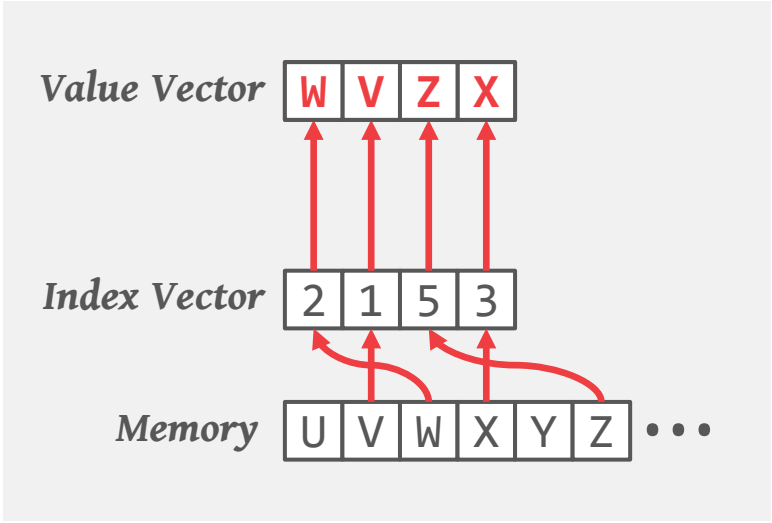
Fundamental Vector Operations

Selective Gather



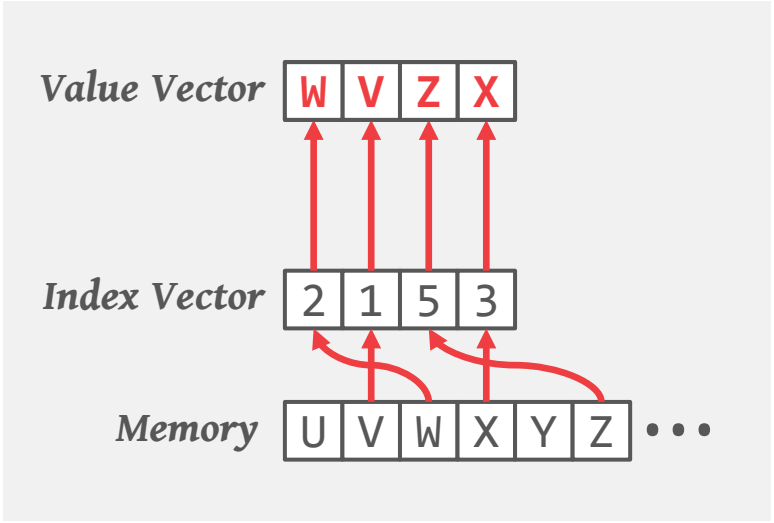
Fundamental Vector Operations

Selective Gather

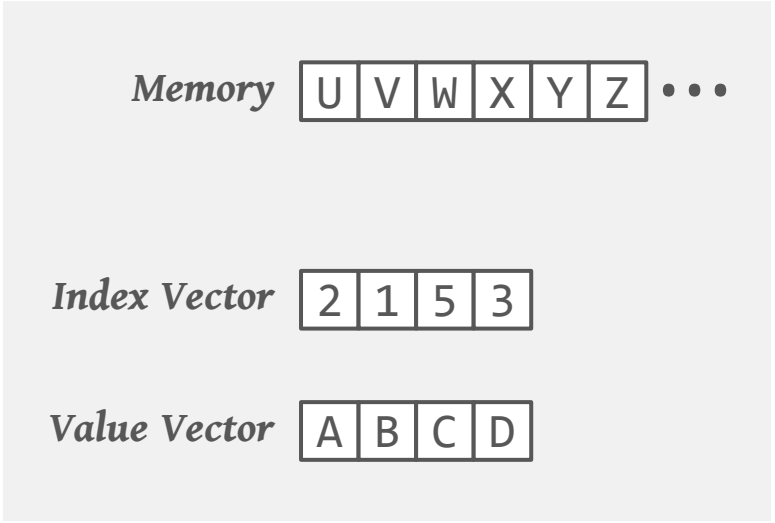


Fundamental Vector Operations

Selective Gather

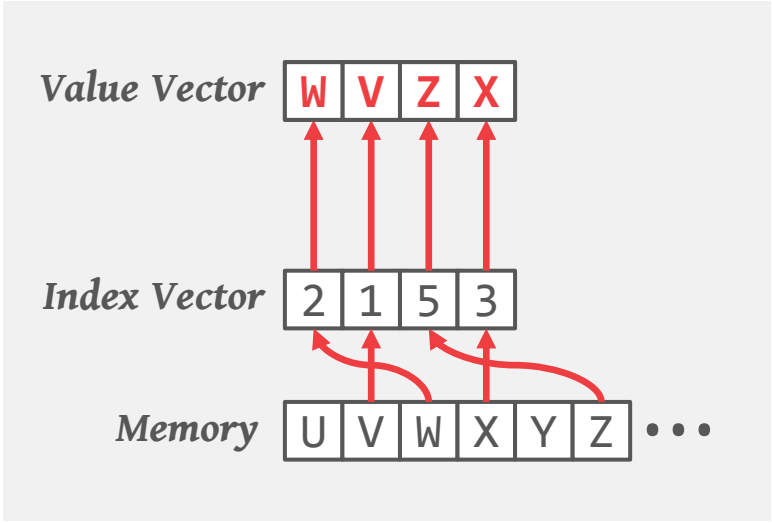


Selective Scatter

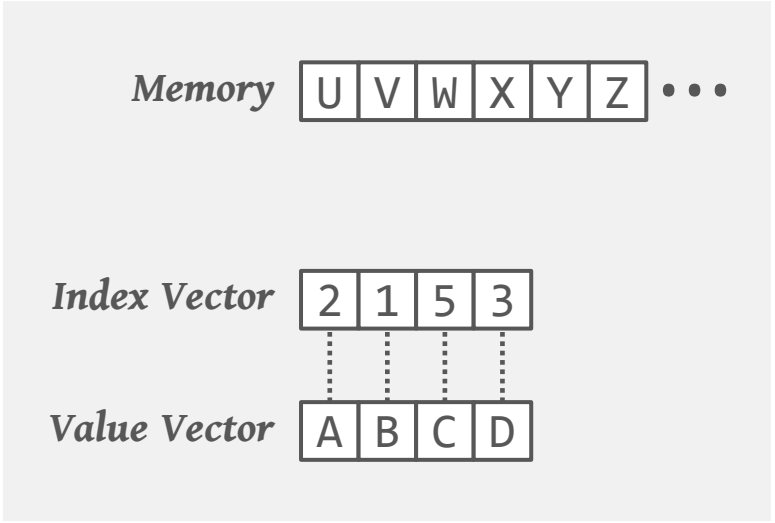


Fundamental Vector Operations

Selective Gather

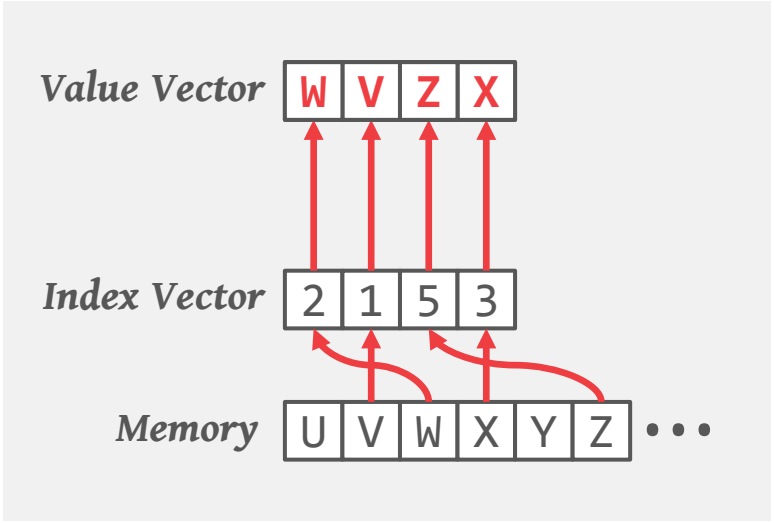


Selective Scatter

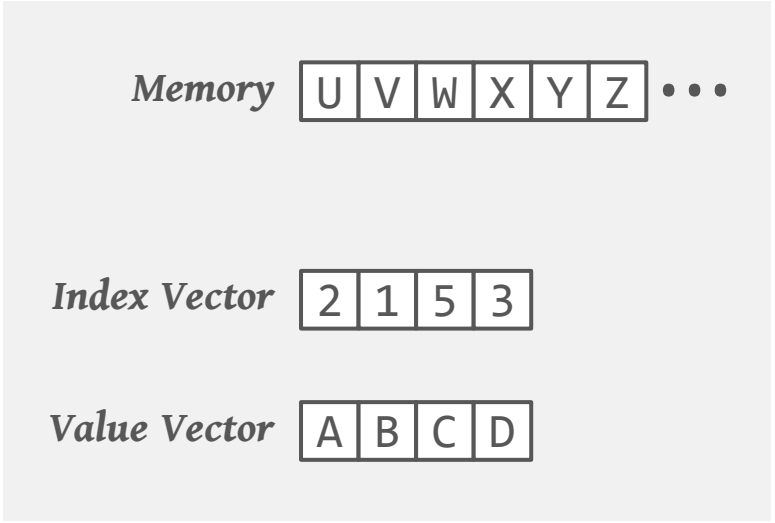


Fundamental Vector Operations

Selective Gather

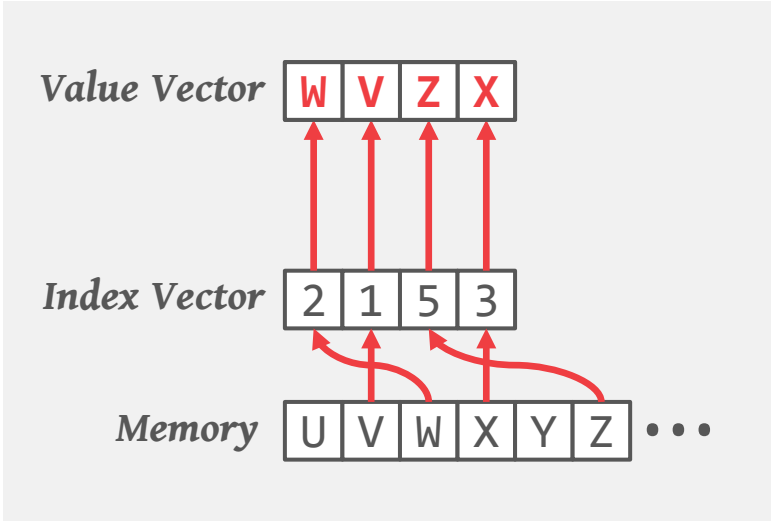


Selective Scatter

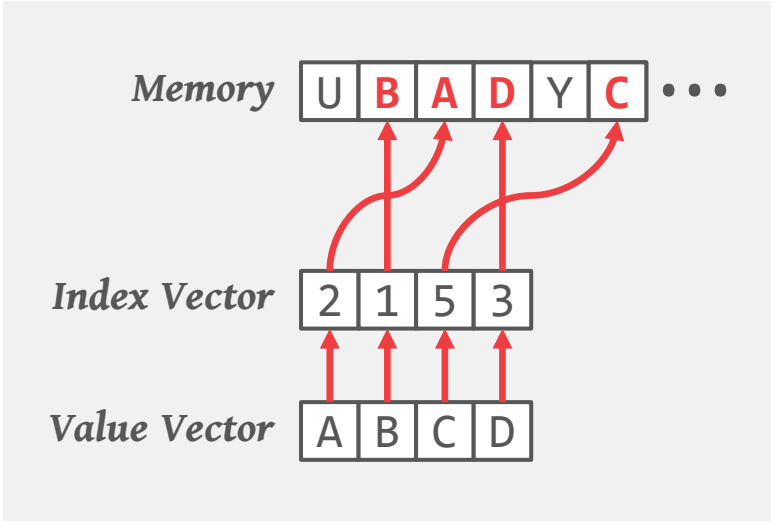


Fundamental Vector Operations

Selective Gather



Selective Scatter



Issues

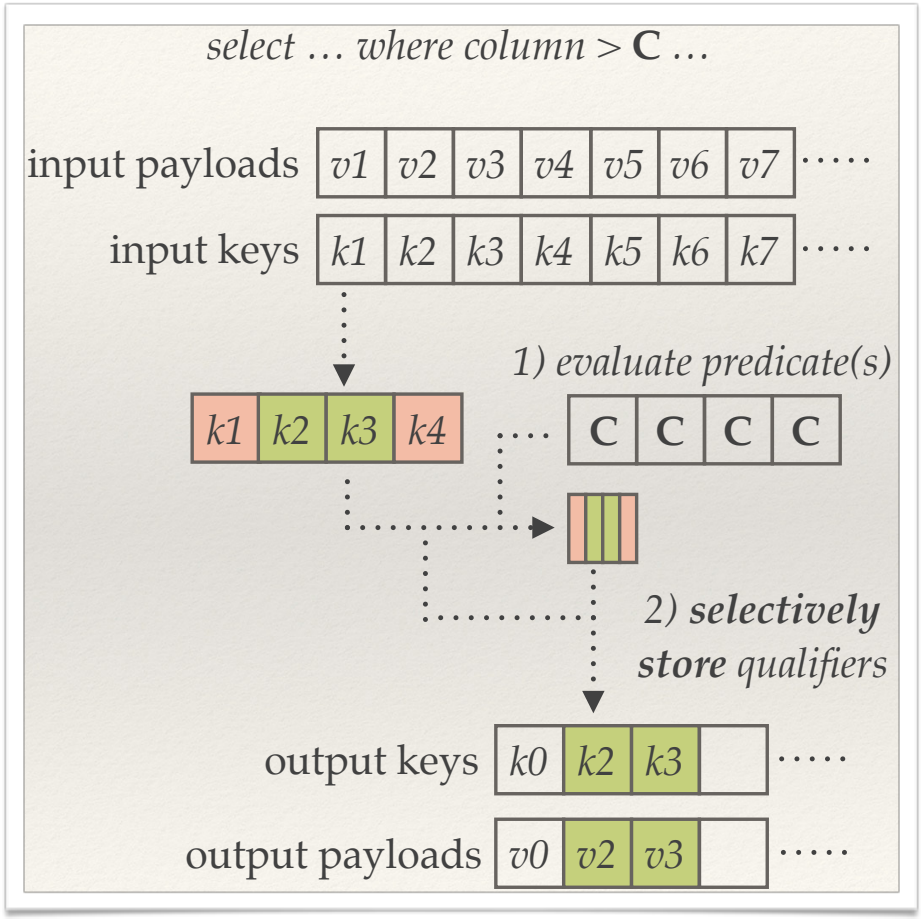
- Gathers and scatters are not really executed in parallel because the *L1 cache only allows one or two distinct accesses per cycle*.

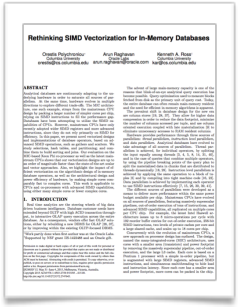
- Gathers are only supported in modern CPUs.
- Selective loads and stores are also emulated in Xeon CPUs using vector permutations.

Vectorized Operators

- ***Selection Scans***
- **Hash Tables**
- **Partitioning**

- **Paper provides additional Info:**
  - ***Joins, Sorting, Bloom filters.***





RETHINKING SIMD VECTORIZATION FOR IN-MEMORY DATABASES

SIGMOD 2015

Selection Scans


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

Selection Scans

Scalar (Branching)

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

Selection Scans

Scalar (Branching)

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

branch misprediction penalty!

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
```


***avoid* branch misprediction penalty!**

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|
```

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|
```

v_t *is a vector of tuples in table*

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|
```

v_k *is a key vector for predicate comparison*

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|
```

produce vector mask

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|
```

SIMD store

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|
```

Selection Scans (Example)

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"
```

Selection Scans (Example)

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

Column-oriented layout!

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

Selection Scans (Example)

Vectorized

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

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

Key Vector

J	O	Y	S	U	X

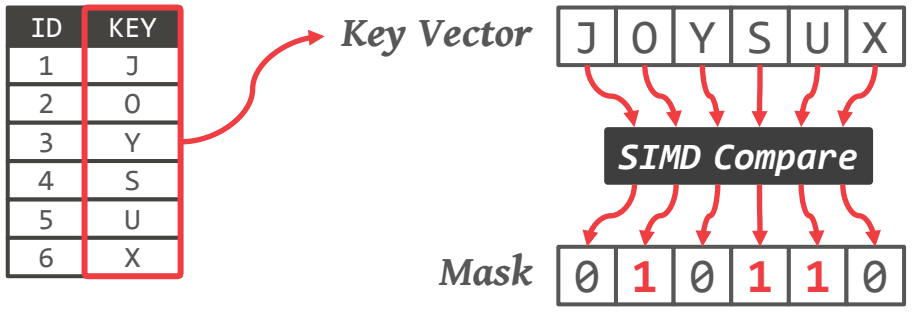
```
SELECT * FROM table
 WHERE key >= "O" AND key <= "U"
```

Selection Scans (Example)

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"
```

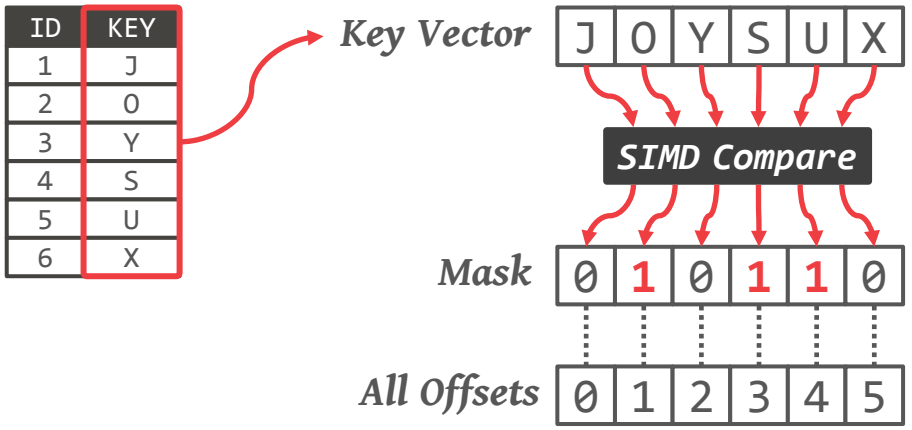


Selection Scans (Example)

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"
```

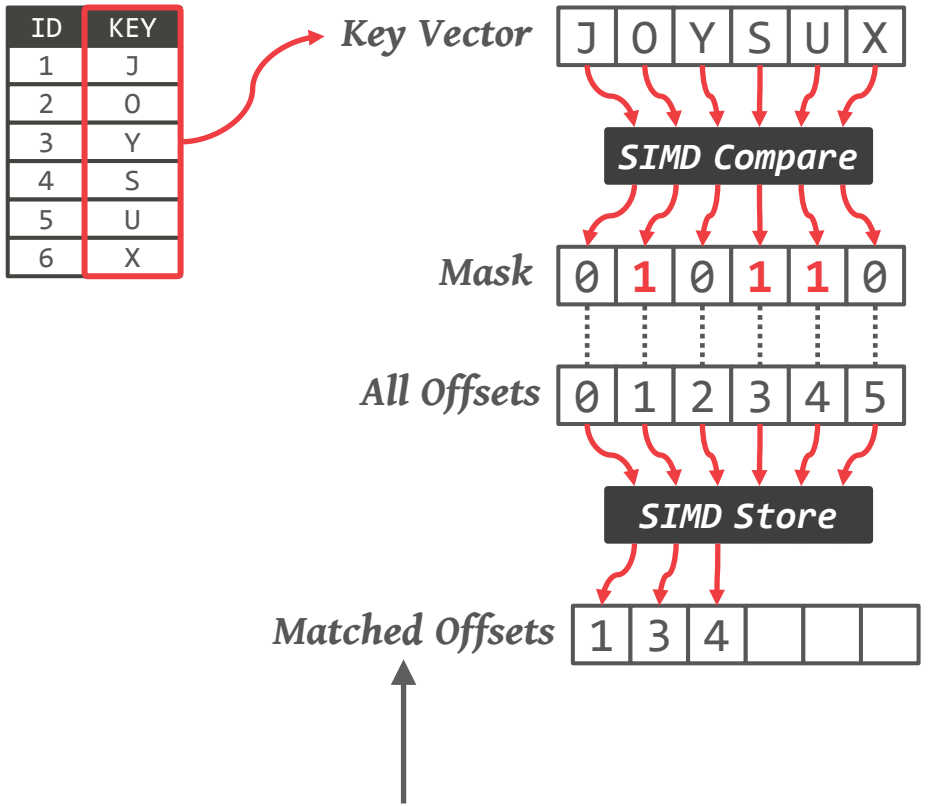


Selection Scans (Example)

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|
```

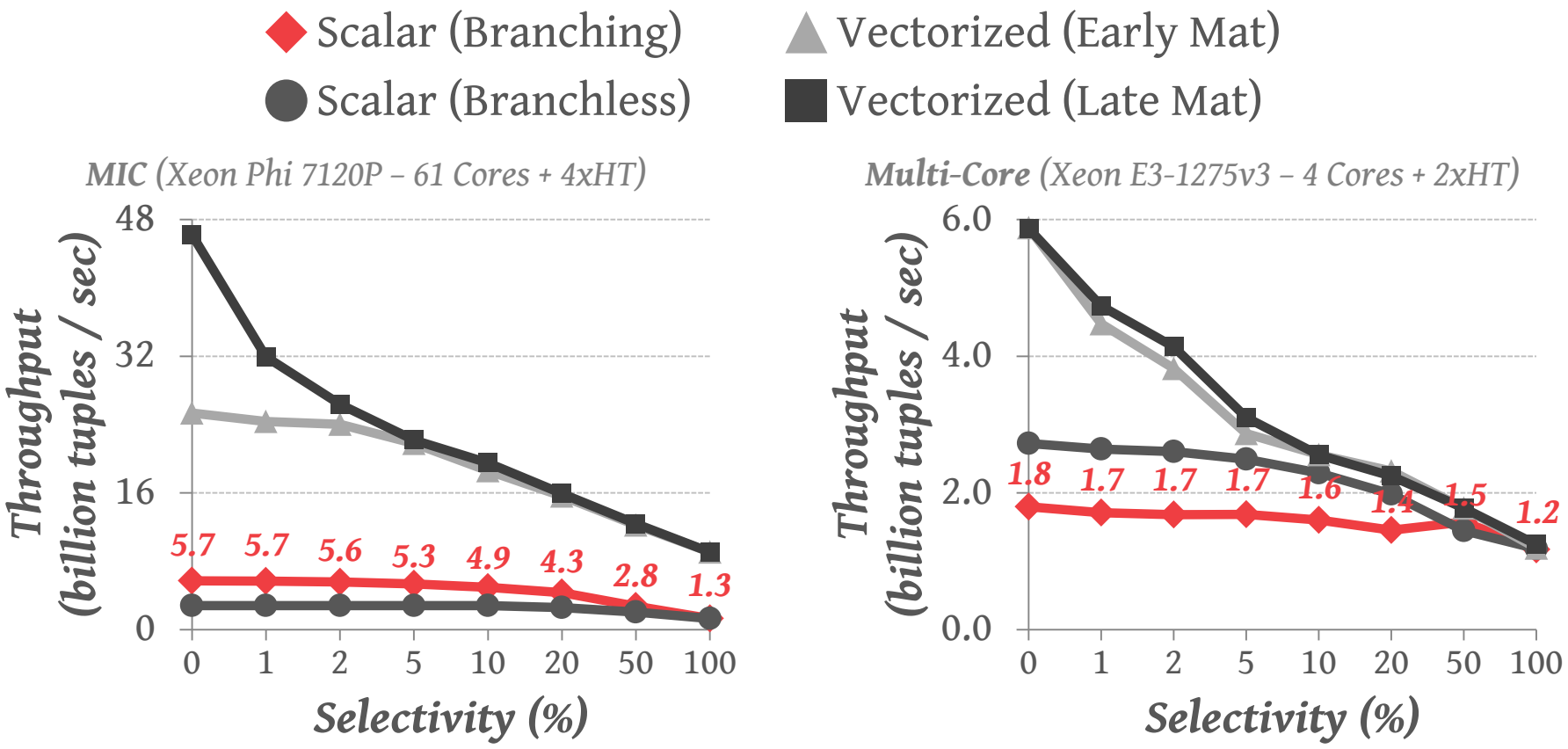
```
SELECT * FROM table
 WHERE key >= "O" AND key <= "U"
```



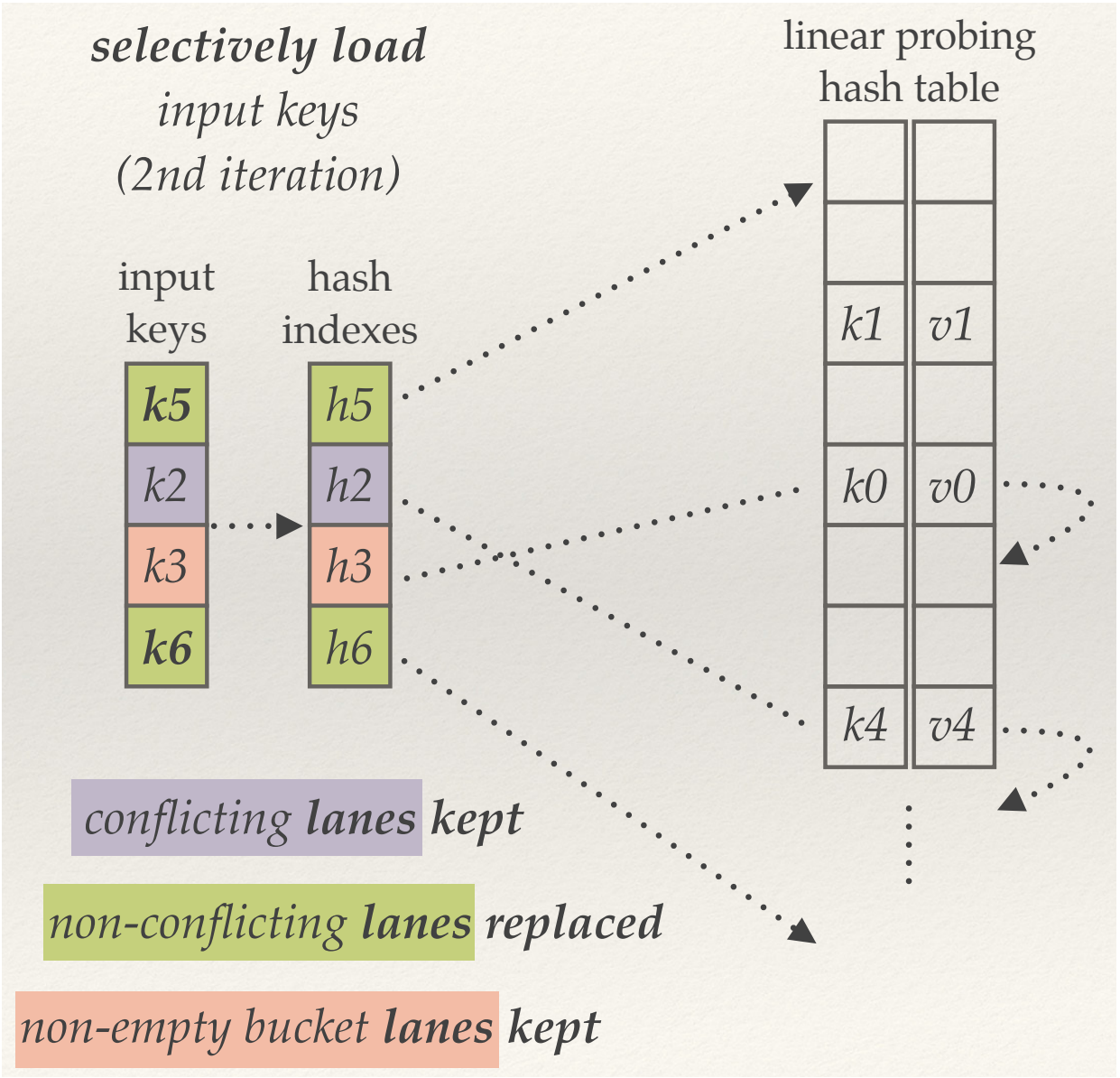
Further SIMD processing is required or not.

It depends on materialization strategy…

Performance Comparison: Selection Scans



Hash Tables - Probing



Assumption:

Use open addressing scheme

Linear Probing Hash Table

KEY	PAYLOAD

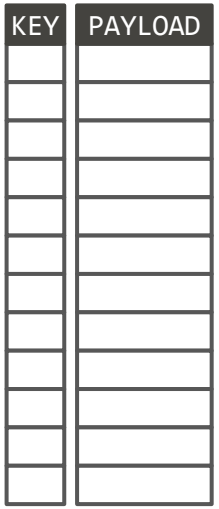
Hash Tables - Probing (Scalar)

Scalar

Input Key

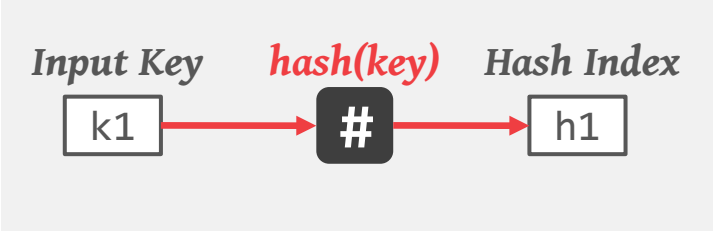
k1

Linear Probing Hash Table

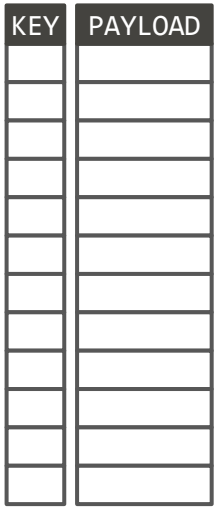


Hash Tables - Probing (Scalar)

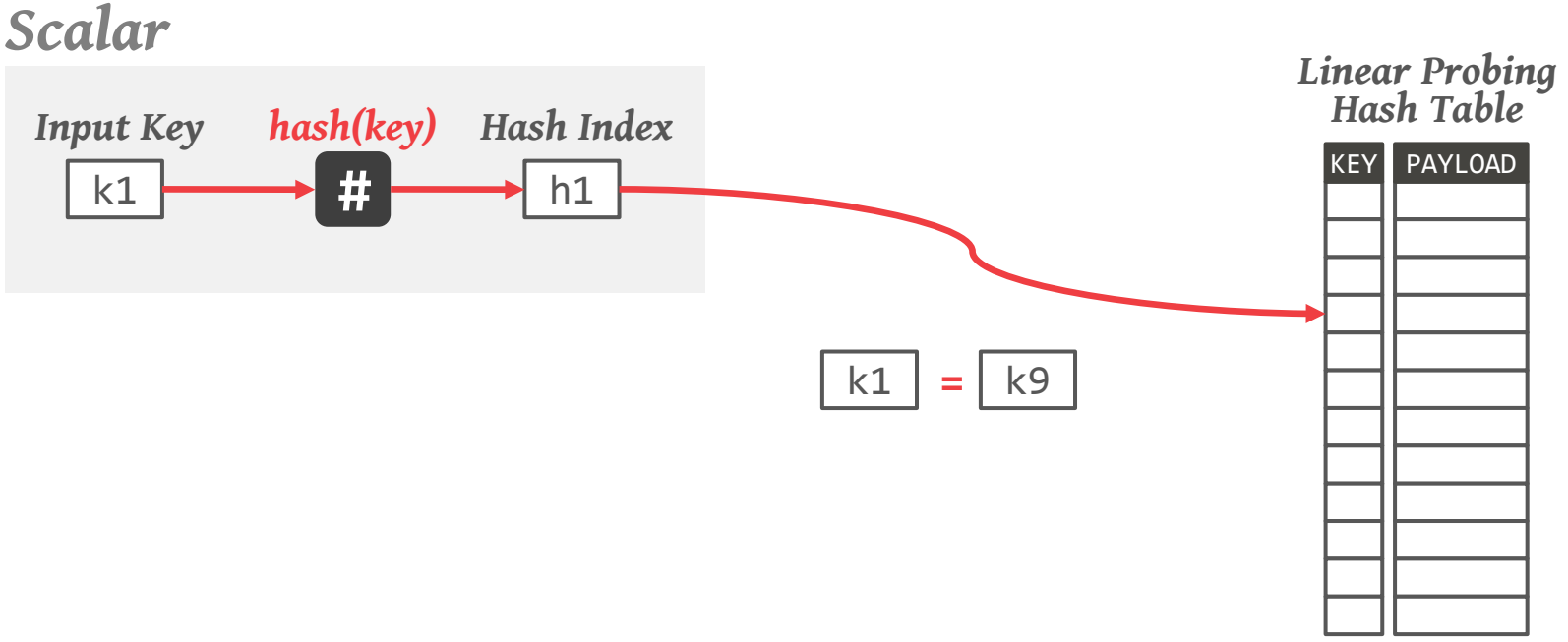
Scalar



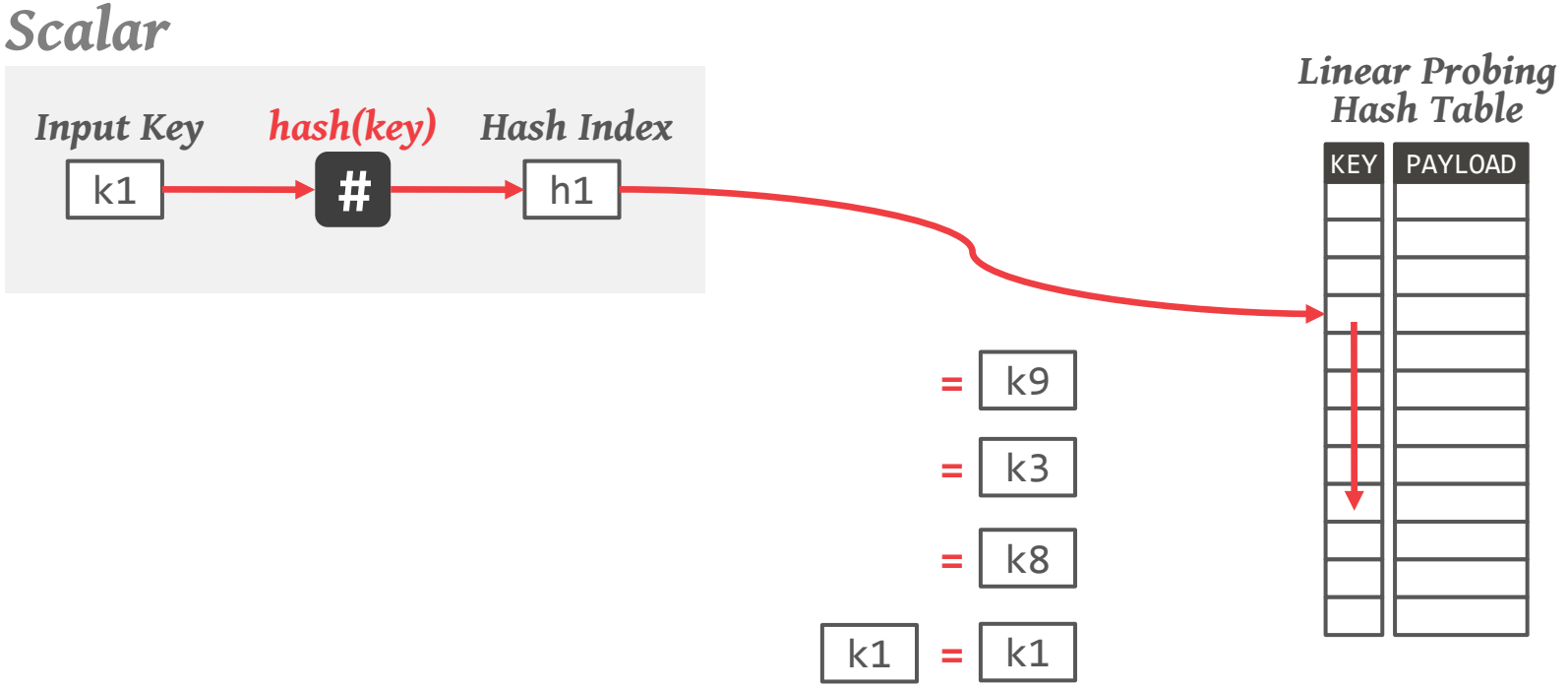
Linear Probing Hash Table



Hash Tables - Probing (Scalar)



Hash Tables - Probing (Scalar)

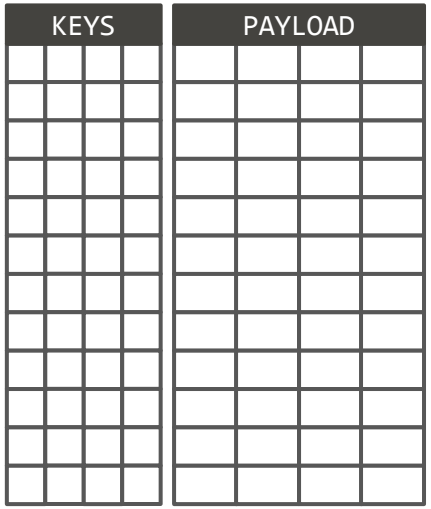


Hash Tables - Probing (Vector)

Scalar

Vectorized (Horizontal)

Linear Probing Bucketized Hash Table

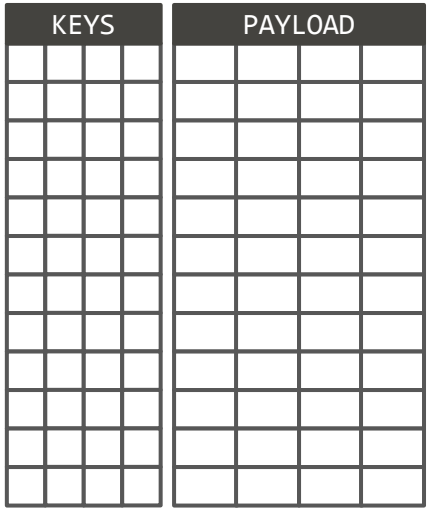


Hash Tables - Probing (Vector)

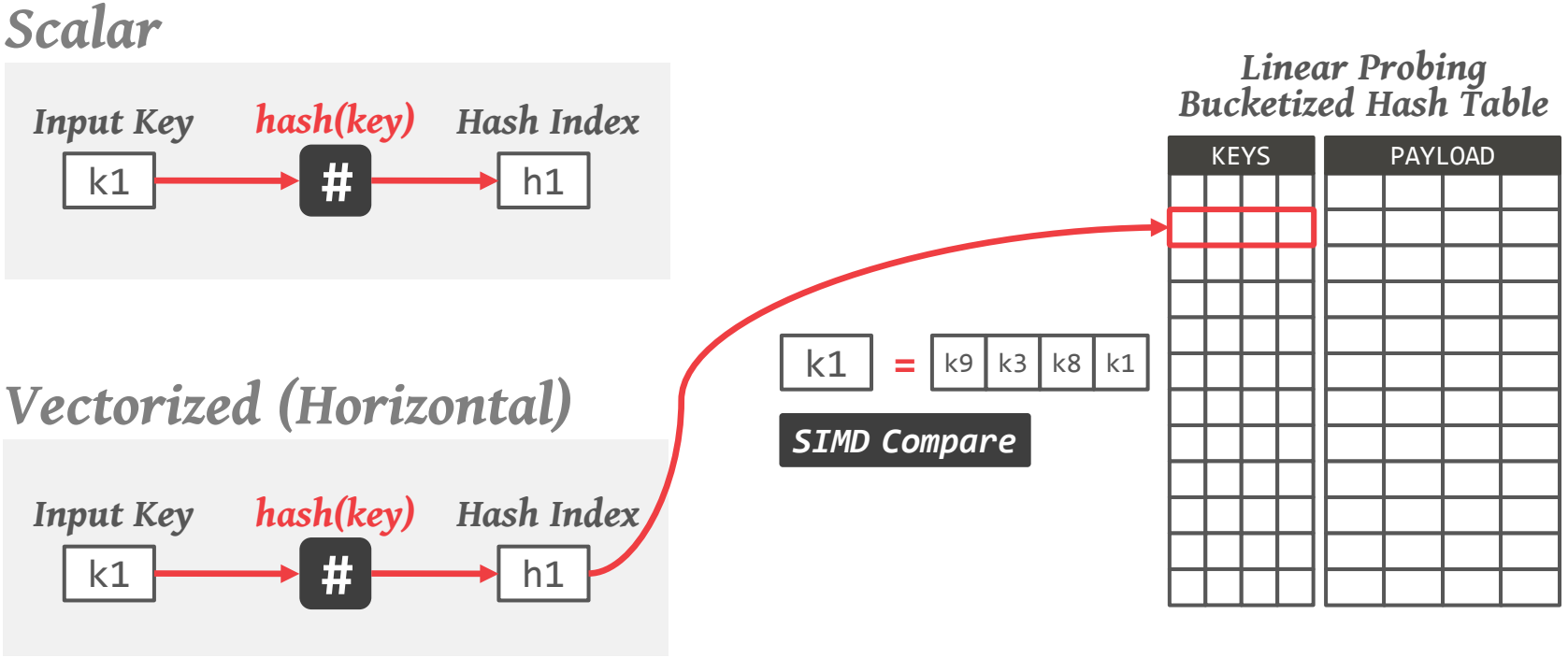
Scalar

Vectorized (Horizontal)

Linear Probing Bucketized Hash Table



Hash Tables - Probing (Vector)



Hash Tables - Probing: Vertical Vectorization

Vectorized (Vertical)

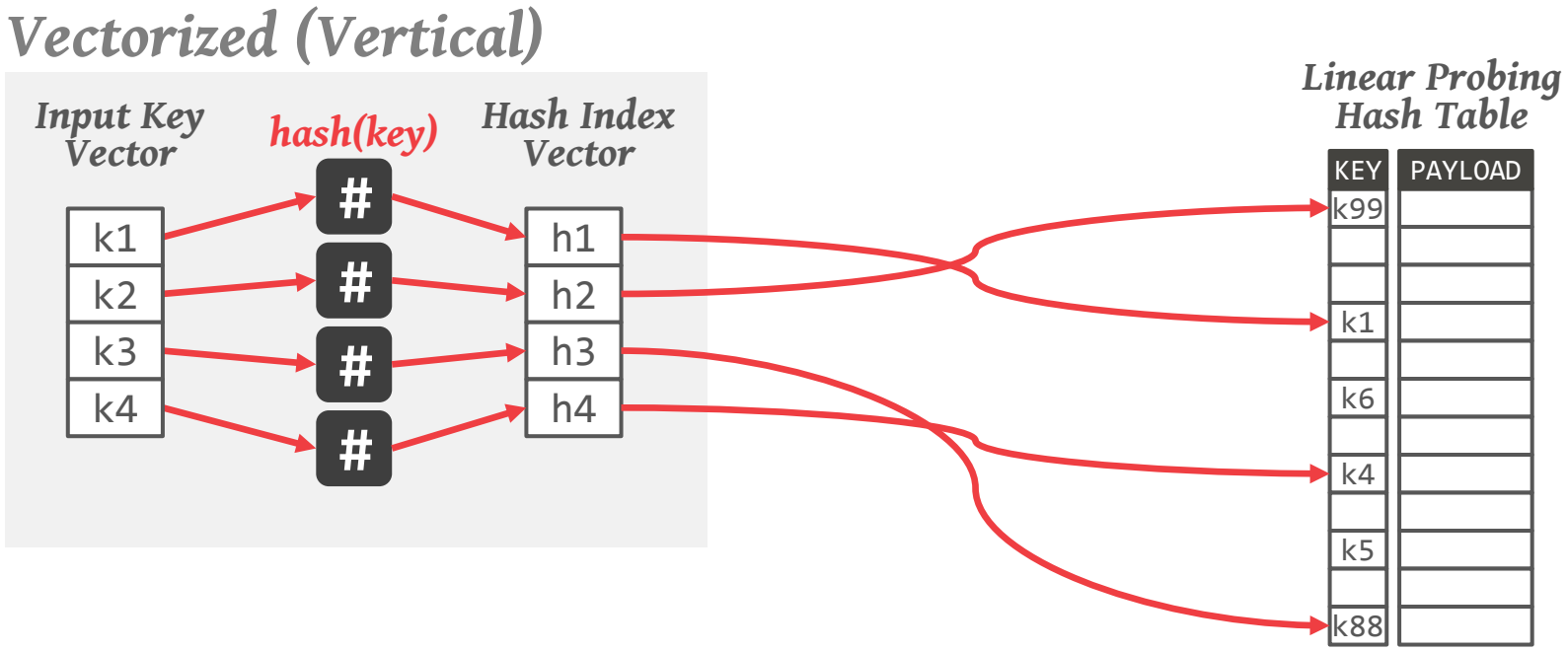
Input Key Vector

k1
k2
k3
k4

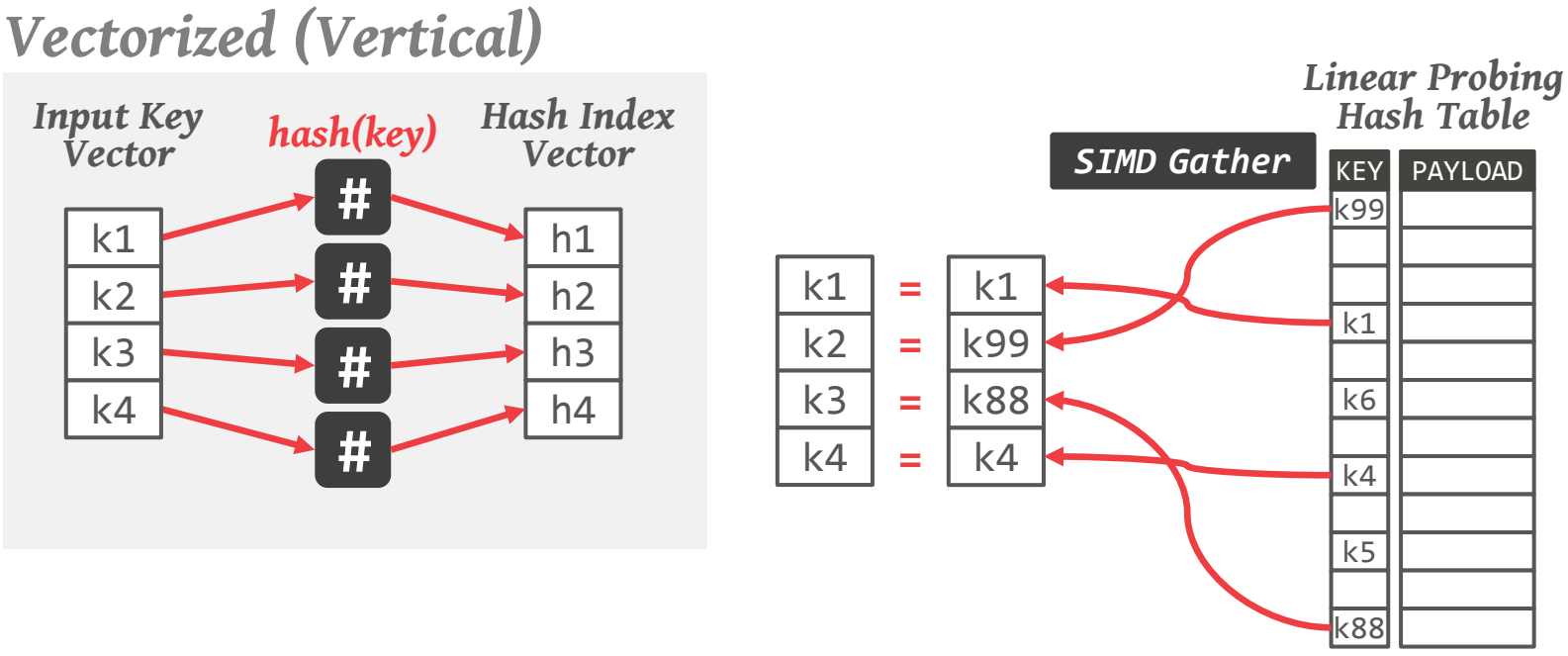
Linear Probing Hash Table

KEY	PAYLOAD
k99	
k1	
k6	
k4	
k5	
k88	

Hash Tables - Probing: Vertical Vectorization

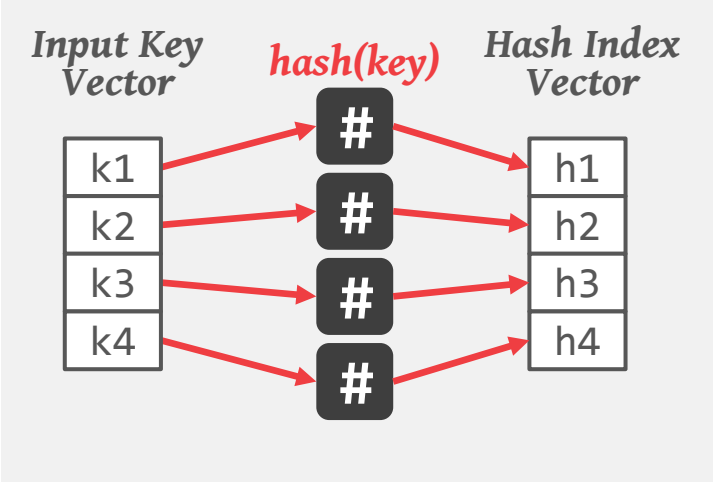


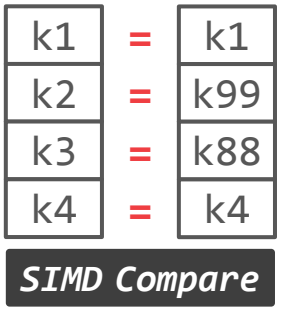
Hash Tables - Probing: Vertical Vectorization



Hash Tables - Probing: Vertical Vectorization

Vectorized (Vertical)



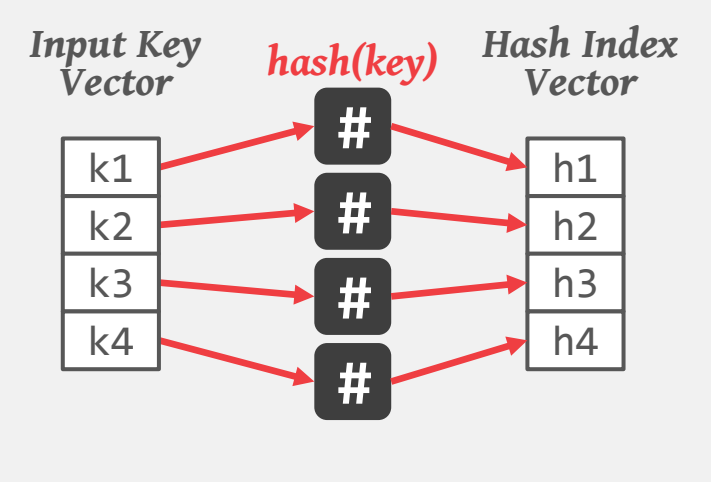


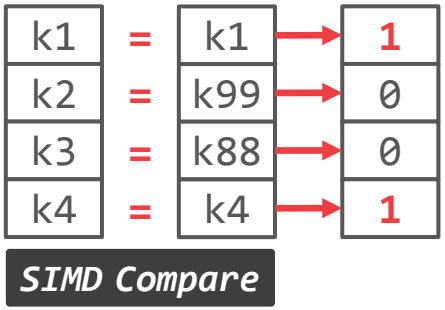
Linear Probing Hash Table

KEY	PAYLOAD
k99	
k1	
k6	
k4	
k5	
k88	

Hash Tables - Probing: Vertical Vectorization

Vectorized (Vertical)



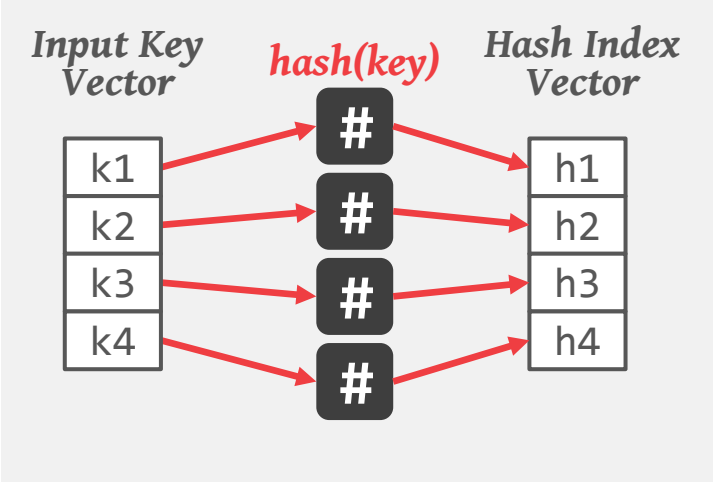


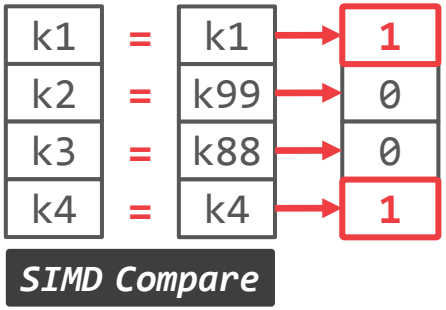
Linear Probing Hash Table

KEY	PAYLOAD
k99	
k1	
k6	
k4	
k5	
k88	

Hash Tables - Probing: Vertical Vectorization

Vectorized (Vertical)



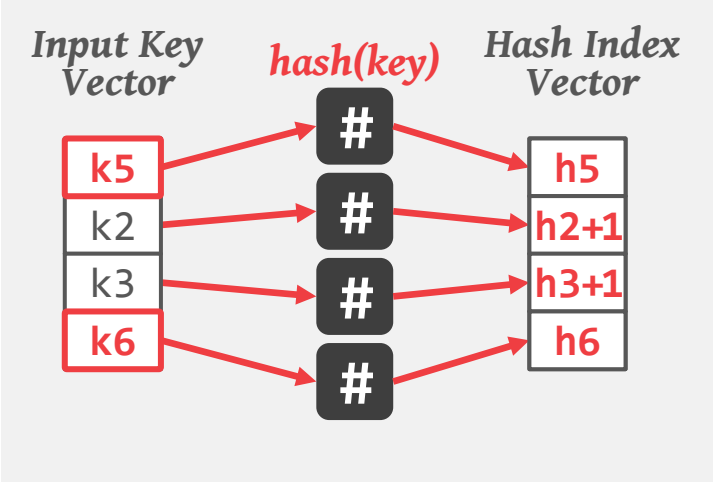


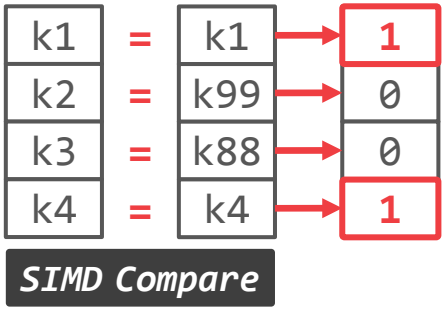
Linear Probing Hash Table

KEY	PAYLOAD
k99	
k1	
k6	
k4	
k5	
k88	

Hash Tables - Probing: Vertical Vectorization

Vectorized (Vertical)

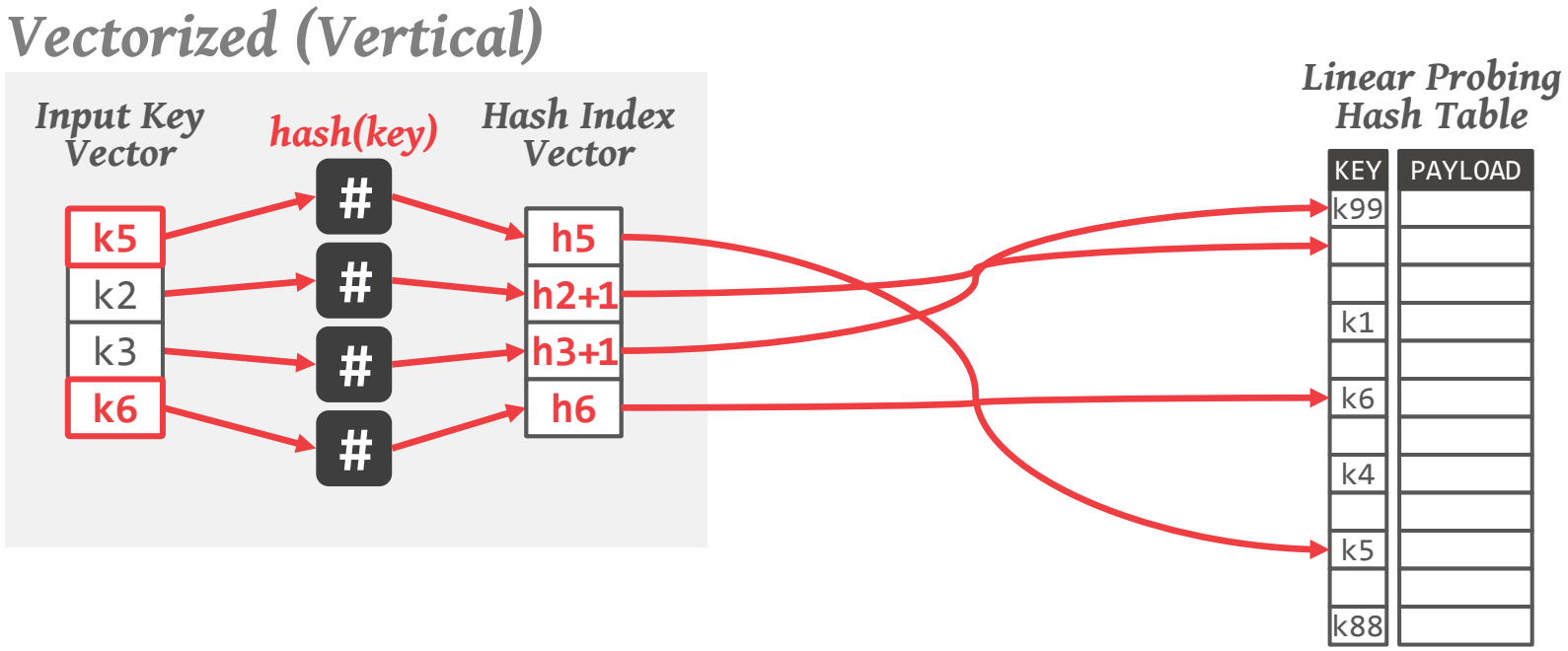




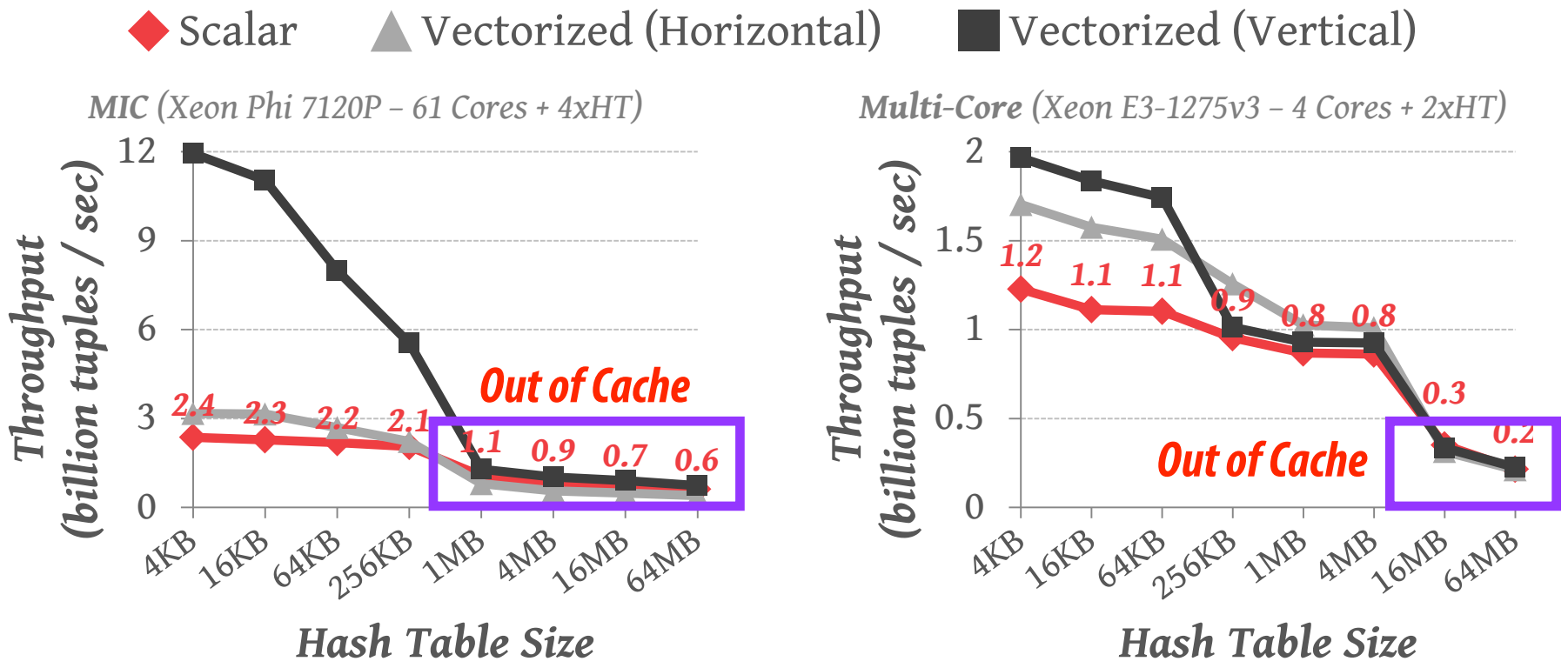
Linear Probing Hash Table

KEY	PAYLOAD
k99	
k1	
k6	
k4	
k5	
k88	

Hash Tables - Probing: Vertical Vectorization

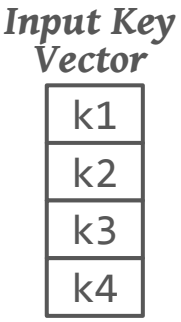


Hash Tables - Probing



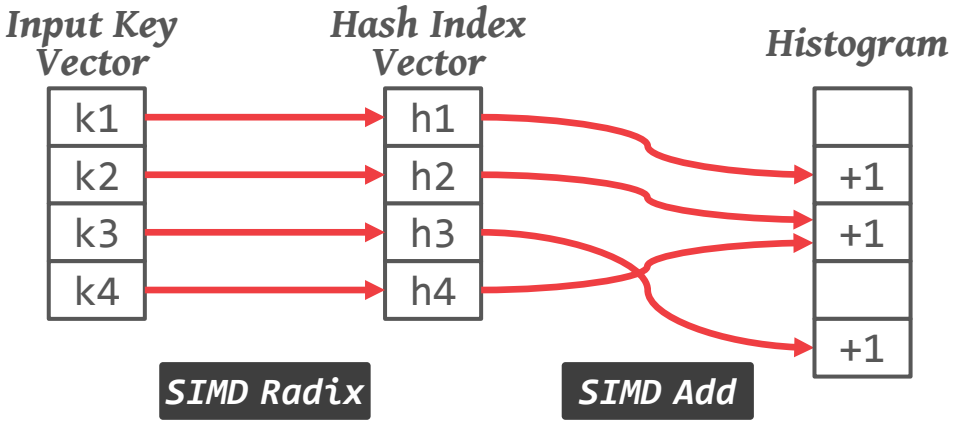
Partitioning - Histogram

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



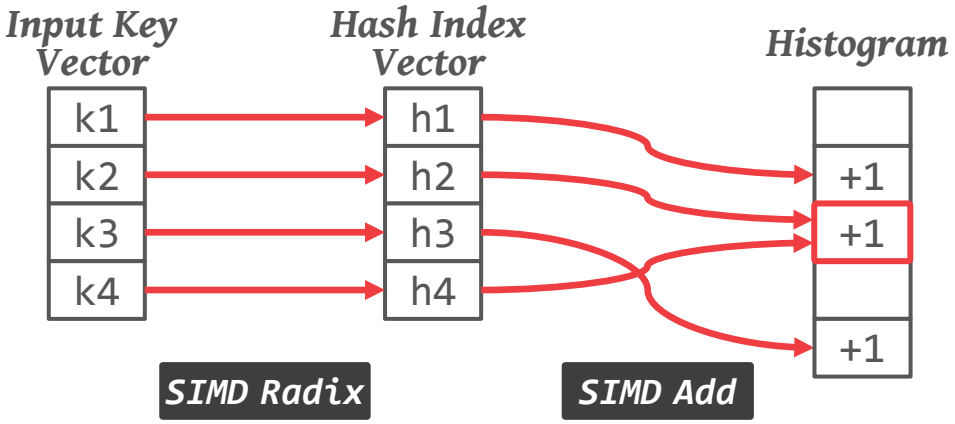
Partitioning - Histogram

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



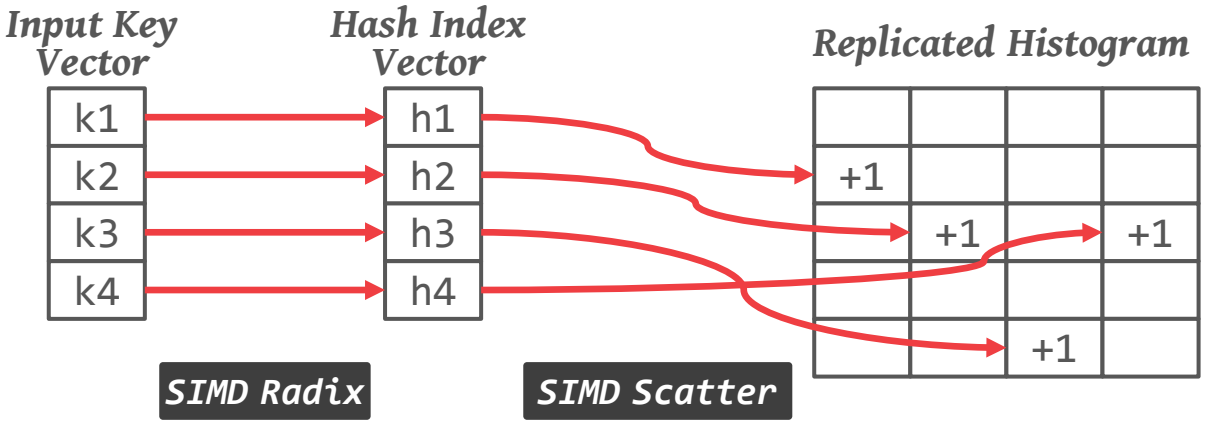
Partitioning - Histogram

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



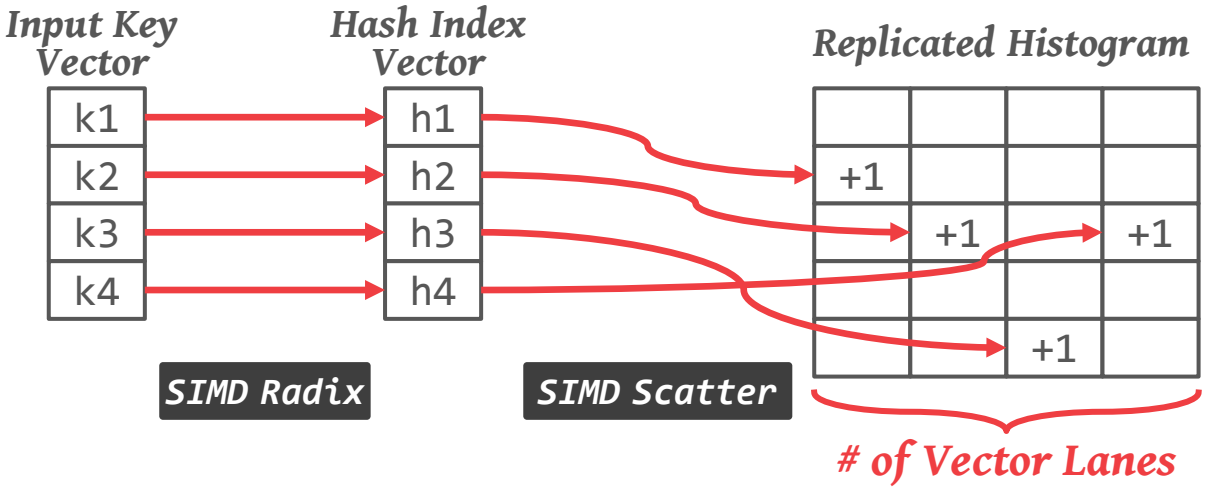
Partitioning - Histogram

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



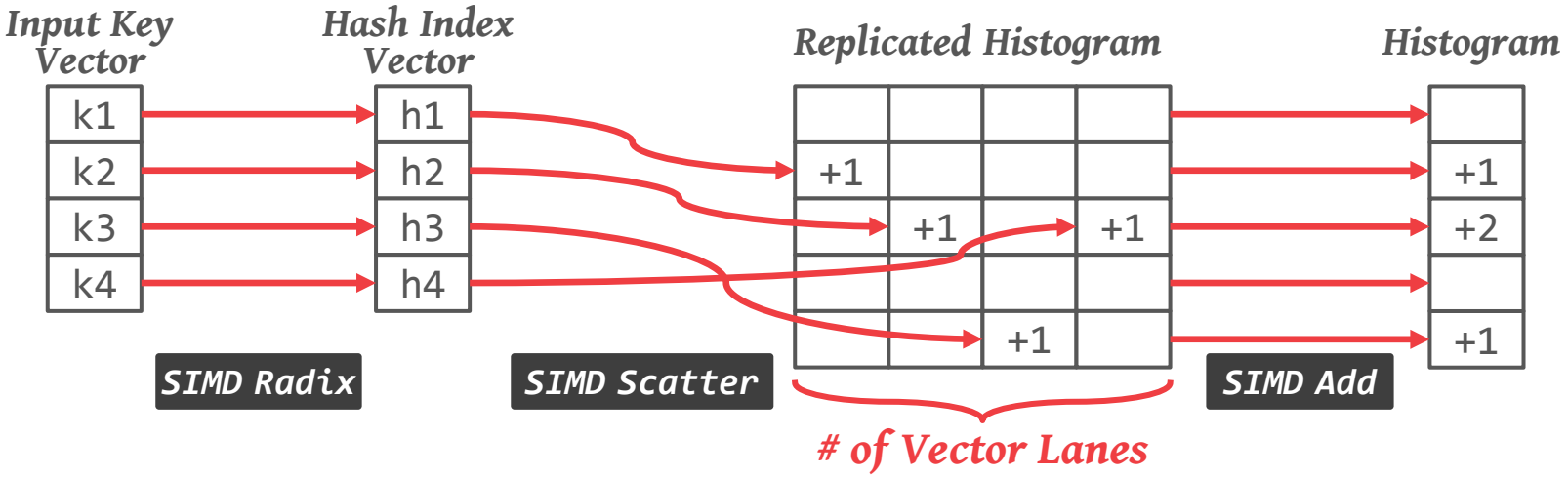
Partitioning - Histogram

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



Partitioning - Histogram

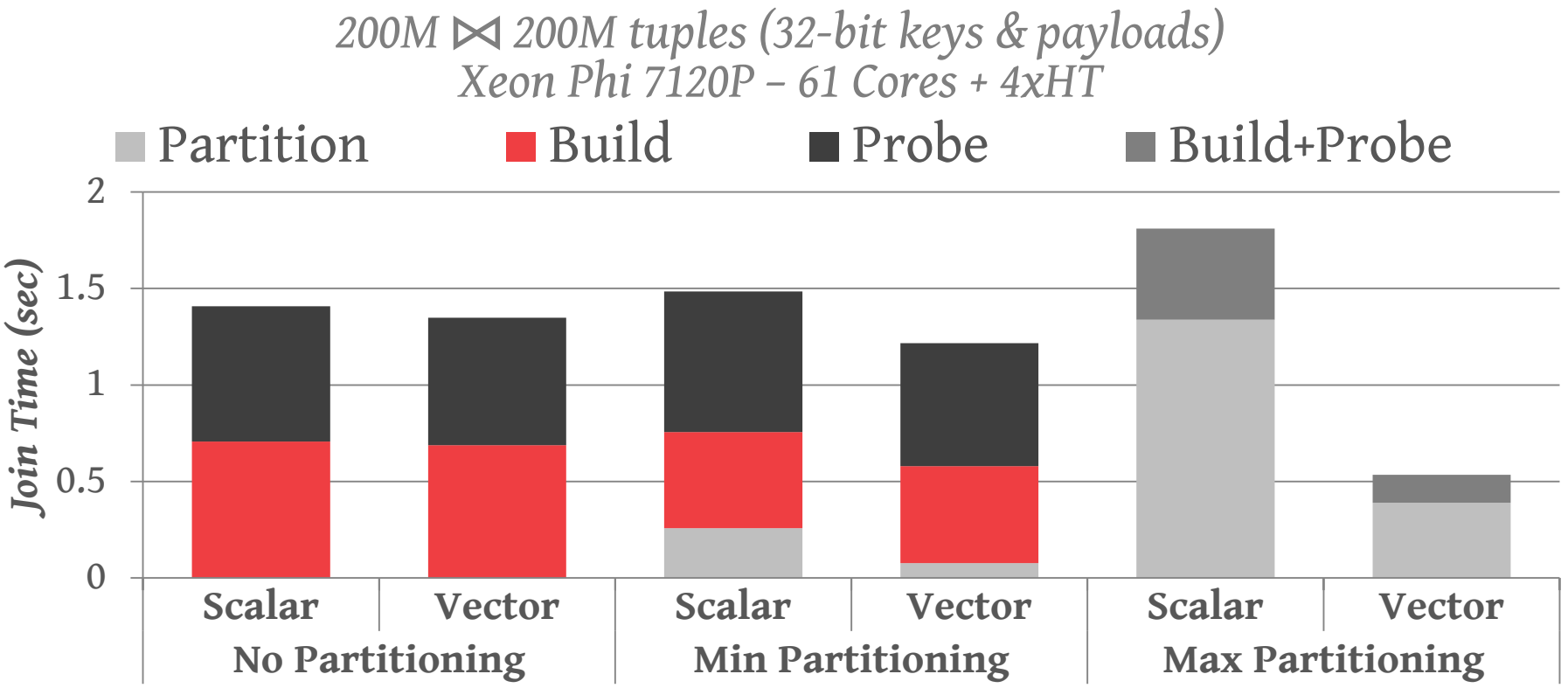
- Use *scatter* and *gathers* to increment counts.
- Replicate the *histogram* to handle collisions.



Joins

- **No Partitioning**
  - Build one shared hash table using *atomics*
  - *Partially* vectorized
- **Min Partitioning**
  - *Partition* building table
  - Build hash table *per thread*
  - *Fully* vectorized
- **Max Partitioning**
  - Partition both tables repeatedly
  - Build and probe *cache-resident hash tables*
  - *Fully* vectorized

Joins


200M ⋈ 200M tuples (32-bit keys & payloads)
Xeon Phi 7120P – 61 Cores + 4xHT
Partition
Build
Probe
Build+Probe
Join Time (sec)
2
1.5
1
0.5
0
Scalar
Vector
Scalar
Vector
Scalar
Vector
No Partitioning
Min Partitioning
Max Partitioning

Main Takeaways

- **Vectorization** is essential for **OLAP queries**.

- **Impact on hardware design**
  - *Simple* cores almost as fast as *complex* cores for OLAP
    - 61 simple P54C cores ~ 32 complex Sandy Bridge cores
  - Improved *power efficiency* for analytical databases

- We can combine all the *intra-query parallelism* optimizations.
  - *Multiple threads* processing the same query.
  - Each thread can execute a *compiled plan*.
  - The compiled plan can invoke *vectorized* operations.

Summary

- Background
- SIMD Vectorization Algorithms
  - Selective Load/Store, Selective Gather/Scatter
  - Selective Scans
  - Hash Tables - Probing
  - Partitioning - Histogram


thank you
рахмат
danke
謝謝
ngiyabonga
teşekkür ederim
Баярлалаа
mersi
kia ora
barka
welalin
tack
spas
mahalo
tapadh leat
спасибо
faafetai lava
vinaka
спасибі
blagodaram
dank je
misaotra
matondo
paldies
grazzi
gracias
хвала
kiitos
dankie
nanni
nandri
dhanyavad
maururu
köszönöm
aciu
dankon
akun
asante
manana
obrigada
tenki
bedankt
bayarlalaa
gràçie
hvala
enkosi
djiere dieuf
tau
mochchakkeram
murakoze
mamnun
дякую
dziękuję
chnorakaloutioun
gratias ago
gràcies
sulpáy
go raibh maith agat
chokrane
sobodi
dĕkuji
sagolun
sukriya
kop khun krap
taiku
arigatô
takk
dakujem
trugarez
obrigado
mèsi
didi madloba
ありがとう
najis tuke
tanemirt
grazie
terima kasih
rahmet
kam sah hamnida
rahmat
diolch
dhanyavadagalu
shukriya
mercê
мерси
ευχαριστώ
merci
তোমাকে ধন্যবাদ
xièxie
감사합니다