

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Vectorized Execution

@Andy_Pavlo // 15-721 // Spring 2020

TODAY'S AGENDA

Background

Vectorized Algorithms (Columbia)

Project #3 Topics



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.
Each core has a 4-wide SIMD registers.

Potential Speed-up: $32x \times 4x = 128x$



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

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];
}
```

$$X$$

8
7
6
5
4
3
2
1

$$Y$$

1
1
1
1
1
1
1
1

SISD

+

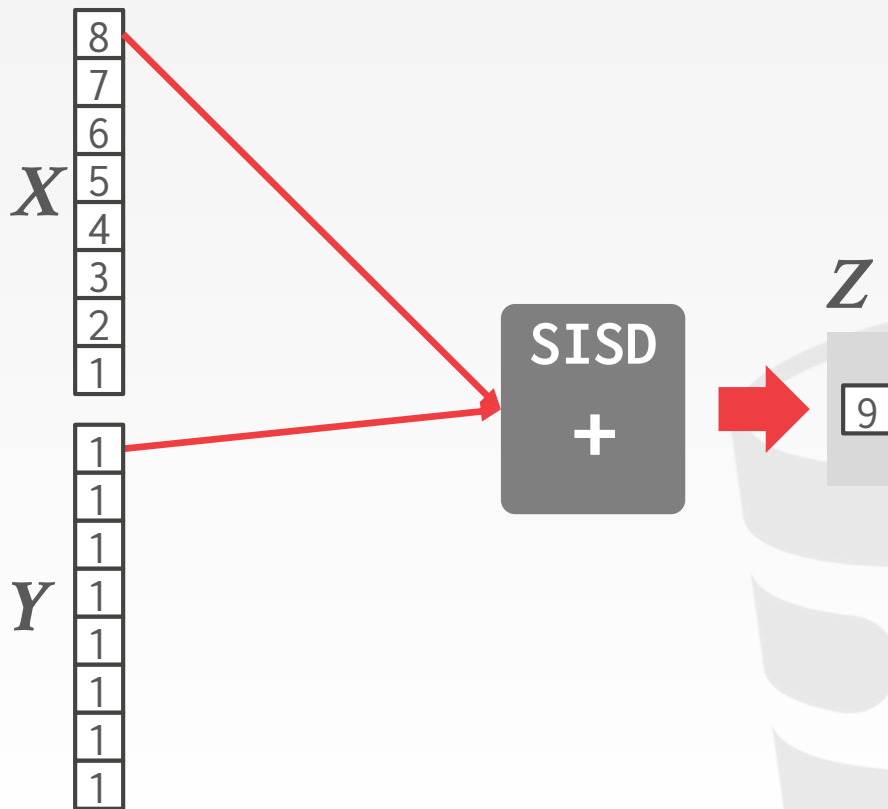
Z

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];
}
```

X

8
7
6
5
4
3
2
1

Y

1
1
1
1
1
1
1
1

SISD
+



Z

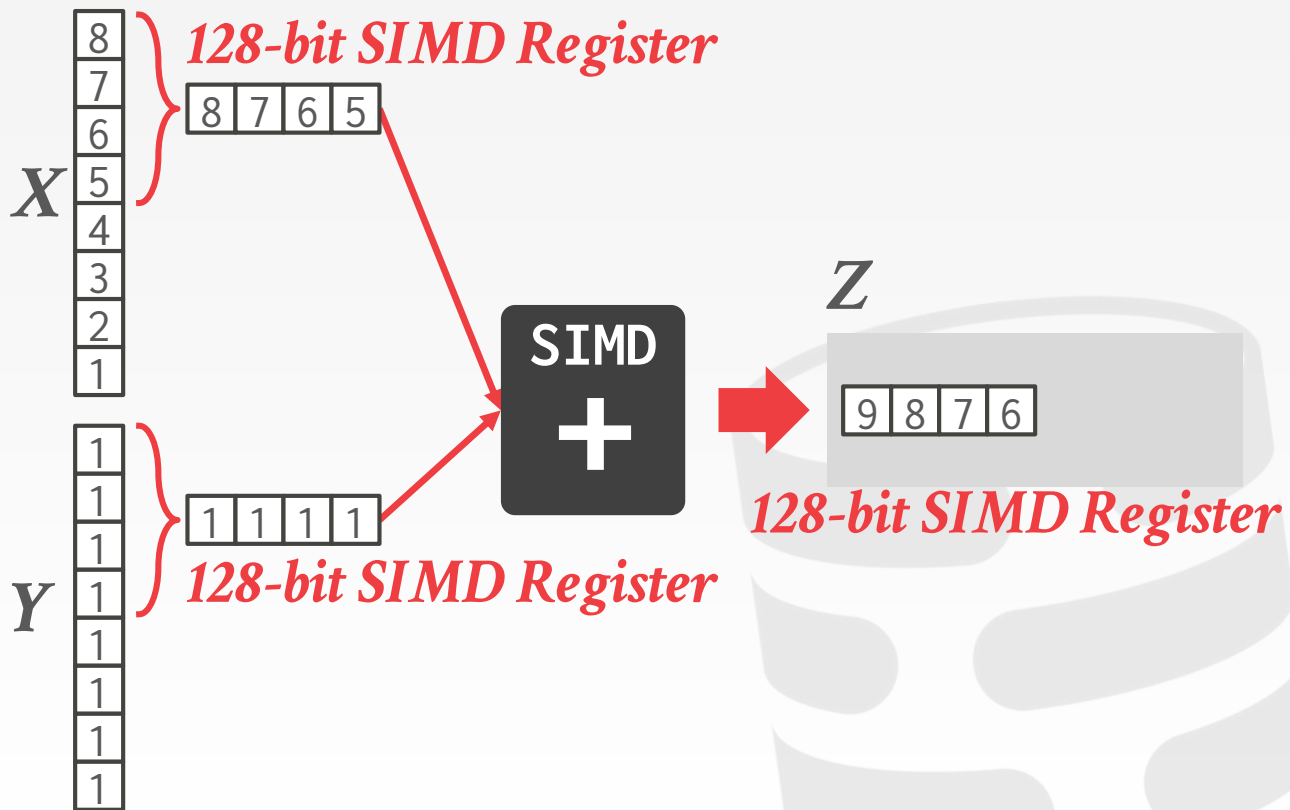
9	8	7	6	5	4	3	2
---	---	---	---	---	---	---	---

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];
}
```

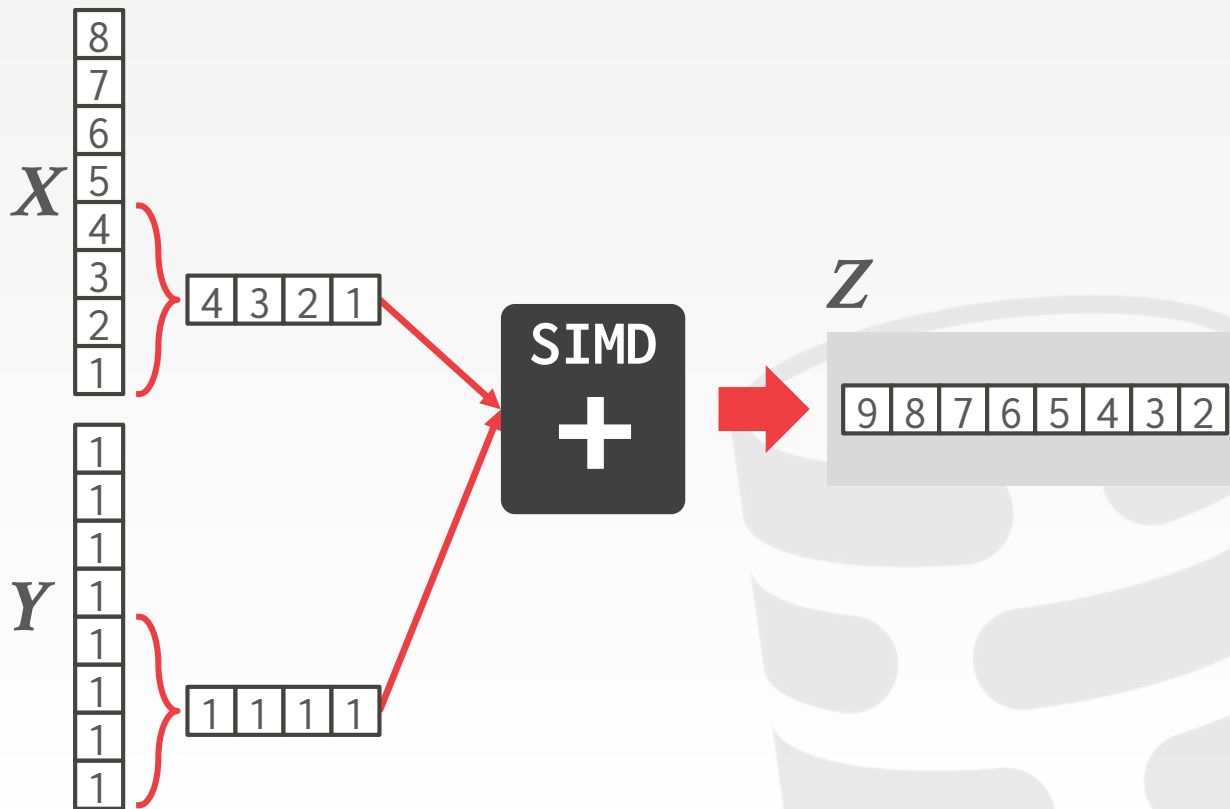


SIMD EXAMPLE

$$X + Y = Z$$

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

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



SIMD 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 in between SIMD registers

Miscellaneous

→ Conversion: Transform data between x86 and SIMD registers.

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

INTEL SIMD EXTENSIONS

		<i>Width</i>	<i>Integers</i>	<i>Single-P</i>	<i>Double-P</i>
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.

VECTORIZATION

Choice #1: Automatic Vectorization

Choice #2: Compiler Hints

Choice #3: Explicit Vectorization



Source: [James Reinders](#)

VECTORIZATION

Choice #1: Automatic Vectorization

Choice #2: Compiler Hints

Choice #3: Explicit Vectorization

Ease of Use



*Programmer
Control*

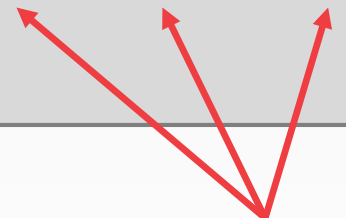
AUTOMATIC VECTORIZATION

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) { *Z=*X+1  
    for (int i=0; i<MAX; i++) {  
        Z[i] = X[i] + Y[i];  
    }  
}
```



These might point to the same address!

This loop is not legal to automatically vectorize.

The code is written such that the addition is described sequentially.

COMPILER HINTS

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

Two approaches:

- Give explicit information about memory locations.
- Tell the compiler to ignore vector dependencies.

COMPILER HINTS

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

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



COMPILER HINTS

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

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

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

EXPLICIT VECTORIZATION

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

Potentially not portable.



EXPLICIT VECTORIZATION

```
void add(int *X,  
        int *Y,  
        int *Z) {  
    __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++));  
    }  
}
```

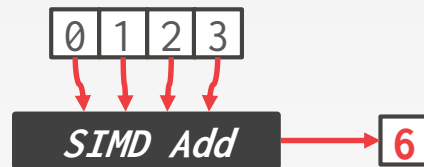
Store the vectors in 128-bit SIMD registers.

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

VECTORIZATION DIRECTION

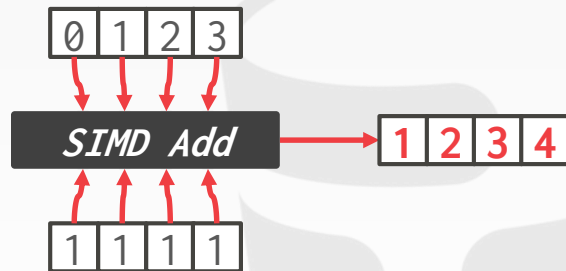
Approach #1: Horizontal

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



Approach #2: Vertical

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



EXPLICIT VECTORIZATION

Linear Access Operators

- Predicate evaluation
- Compression

Ad-hoc Vectorization

- Sorting
- Merging

Composable Operations

- Multi-way trees
- Bucketized hash tables

Source: [Orestis Polychroniou](#)



VECTORIZED DBMS ALGORITHMS

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

- Favor *vertical* vectorization by processing different input data per lane.
- Maximize lane utilization by executing unique data items per lane subset (i.e., no useless computations).



FUNDAMENTAL OPERATIONS

Selective Load

Selective Store

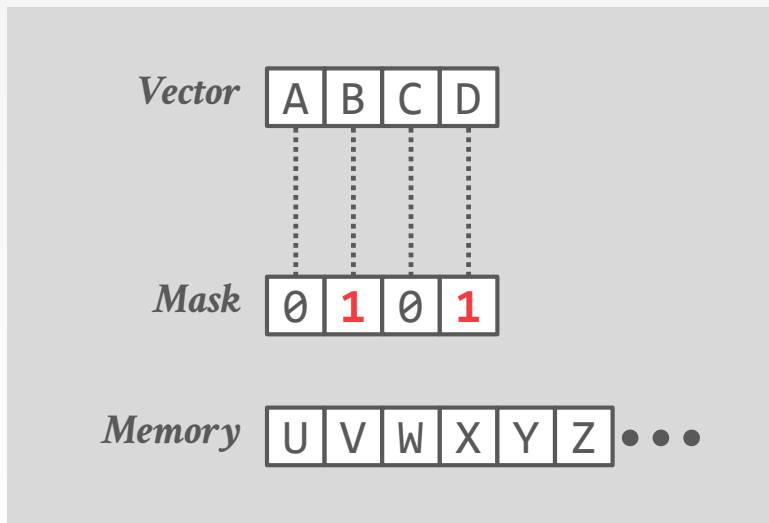
Selective Gather

Selective Scatter



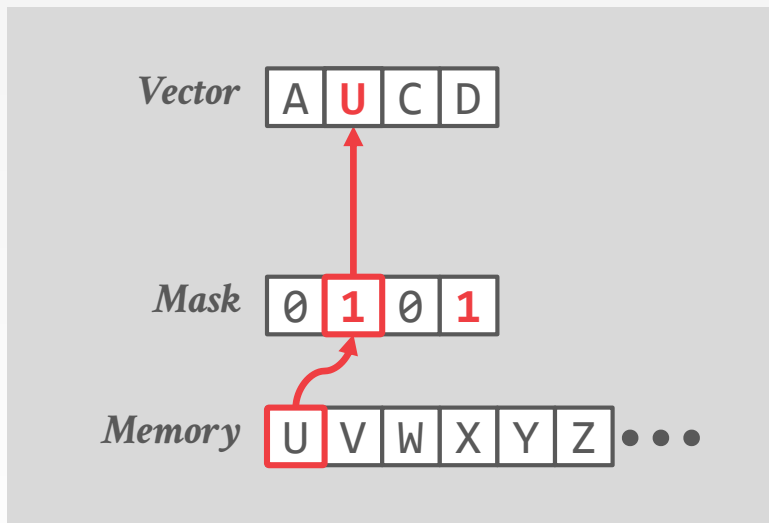
FUNDAMENTAL VECTOR OPERATIONS

Selective Load



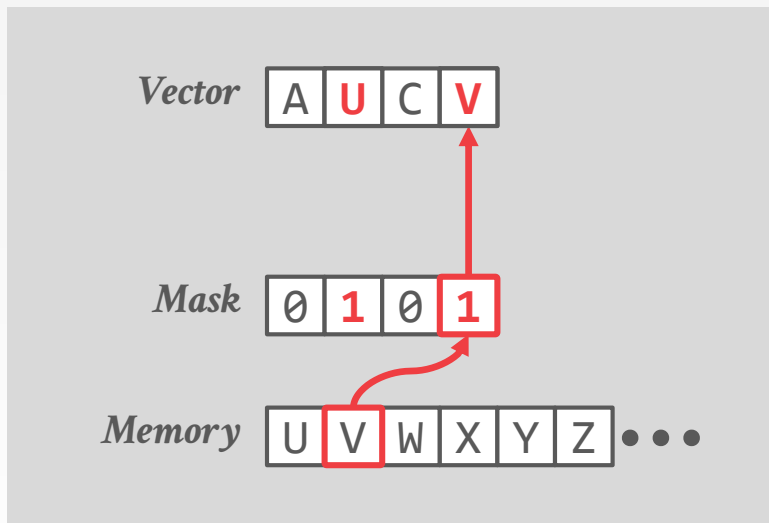
FUNDAMENTAL VECTOR OPERATIONS

Selective Load



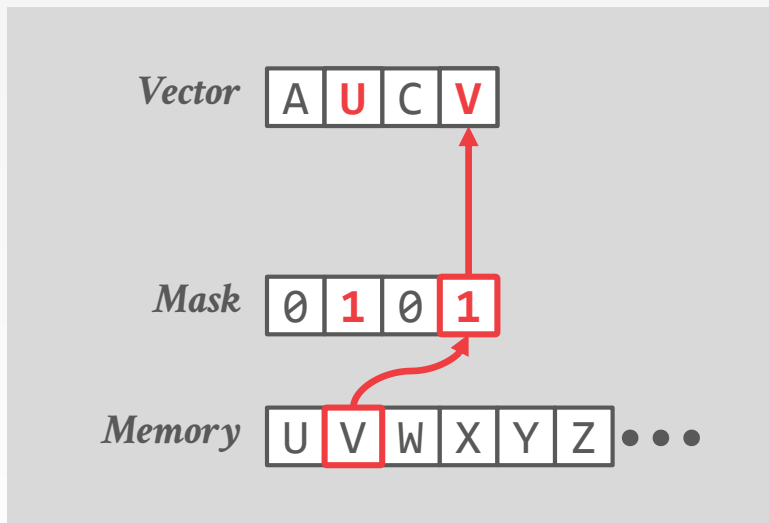
FUNDAMENTAL VECTOR OPERATIONS

Selective Load

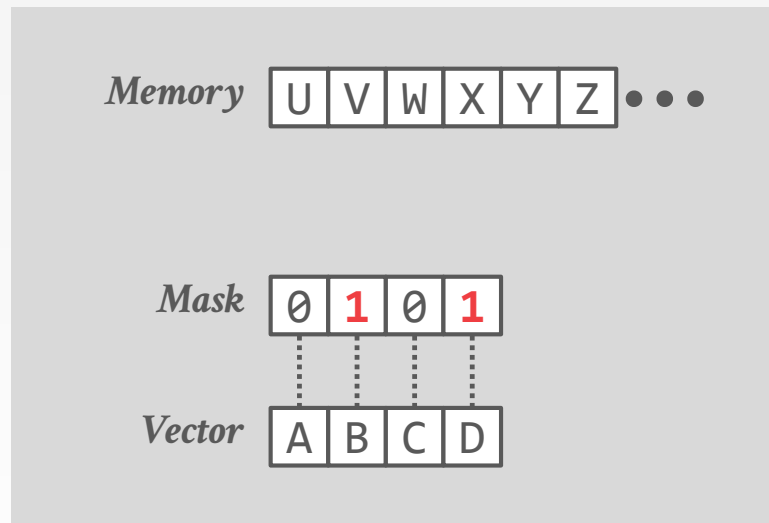


FUNDAMENTAL VECTOR OPERATIONS

Selective Load

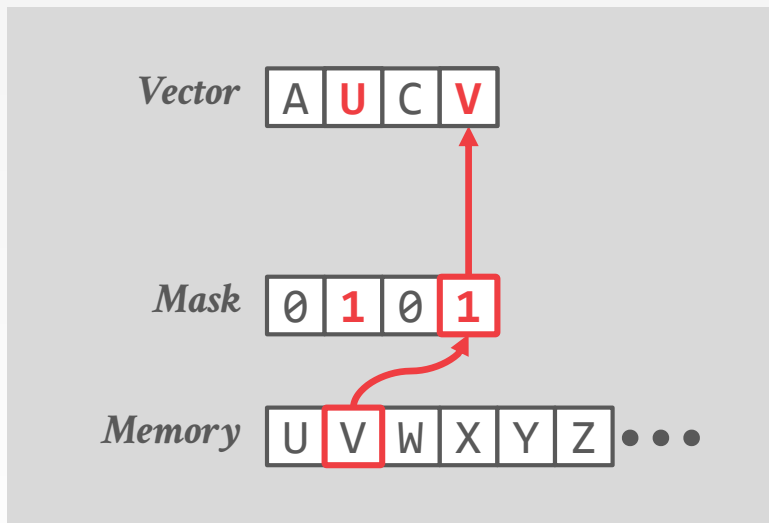


Selective Store

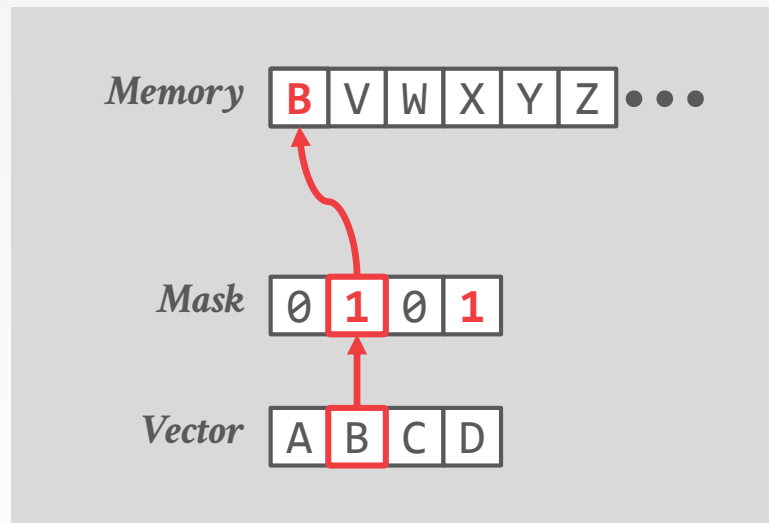


FUNDAMENTAL VECTOR OPERATIONS

Selective Load

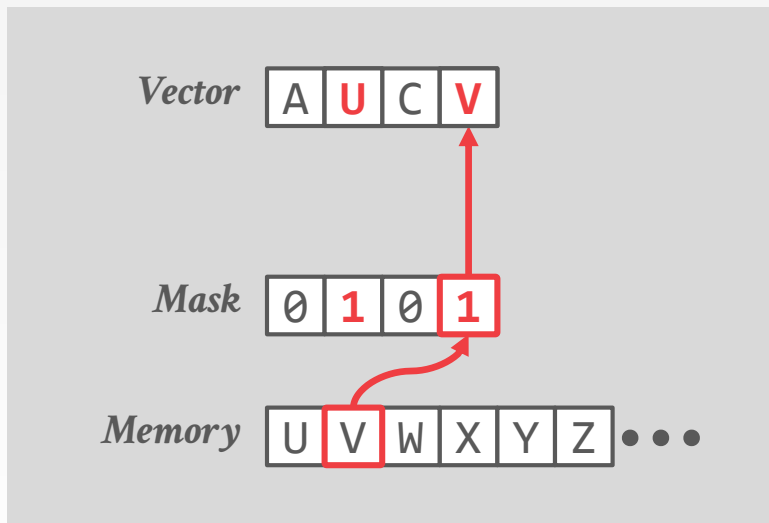


Selective Store

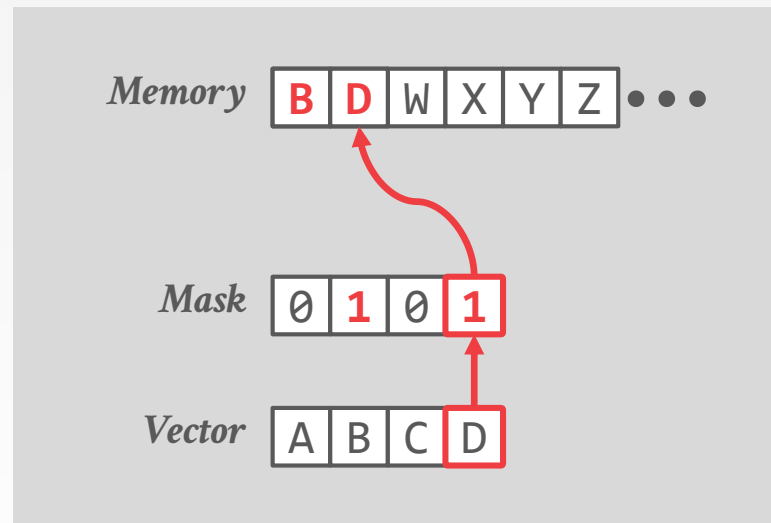


FUNDAMENTAL VECTOR OPERATIONS

Selective Load

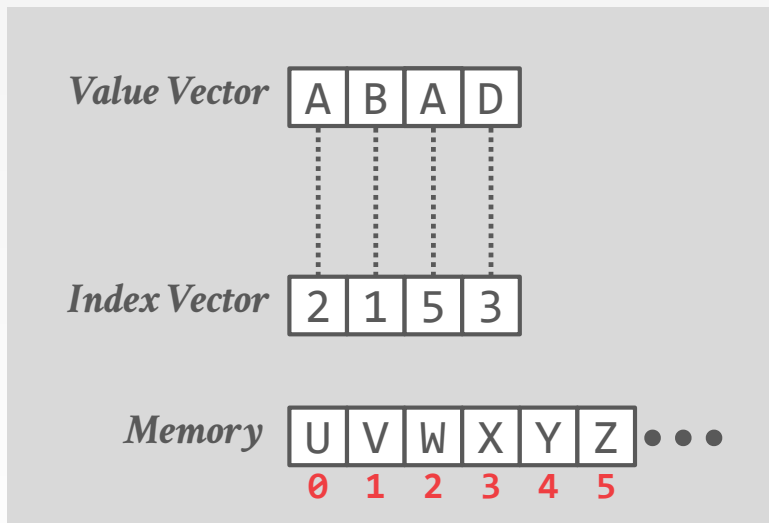


Selective Store



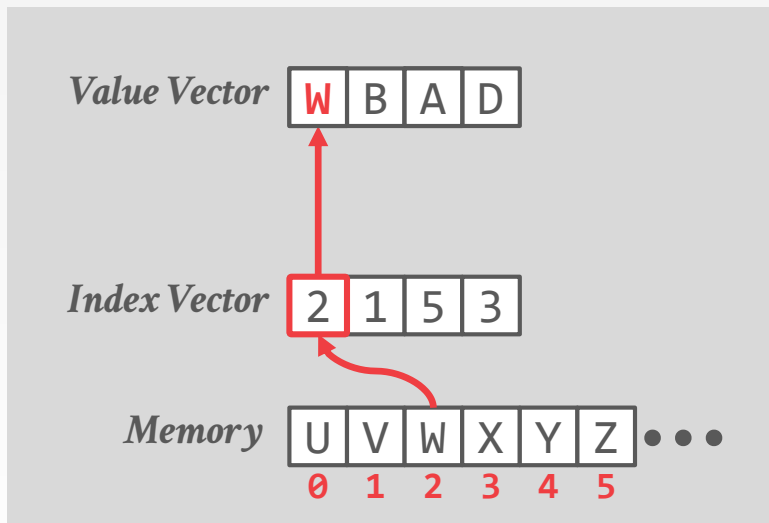
FUNDAMENTAL VECTOR OPERATIONS

Selective Gather



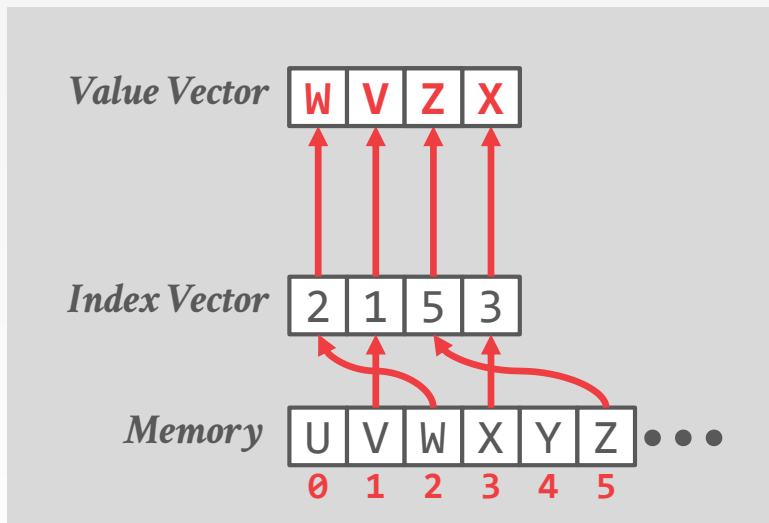
FUNDAMENTAL VECTOR OPERATIONS

Selective Gather



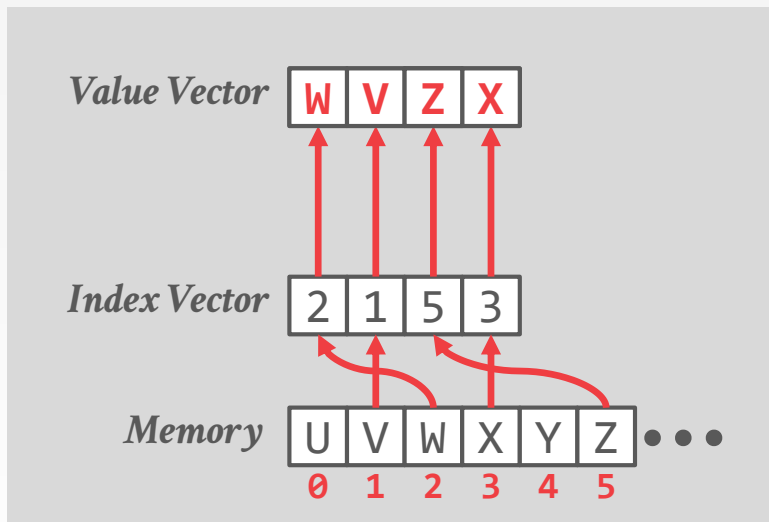
FUNDAMENTAL VECTOR OPERATIONS

Selective Gather

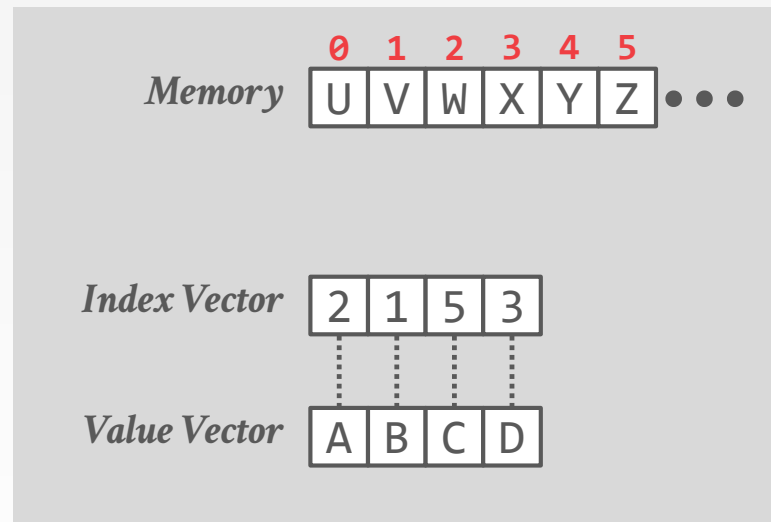


FUNDAMENTAL VECTOR OPERATIONS

Selective Gather

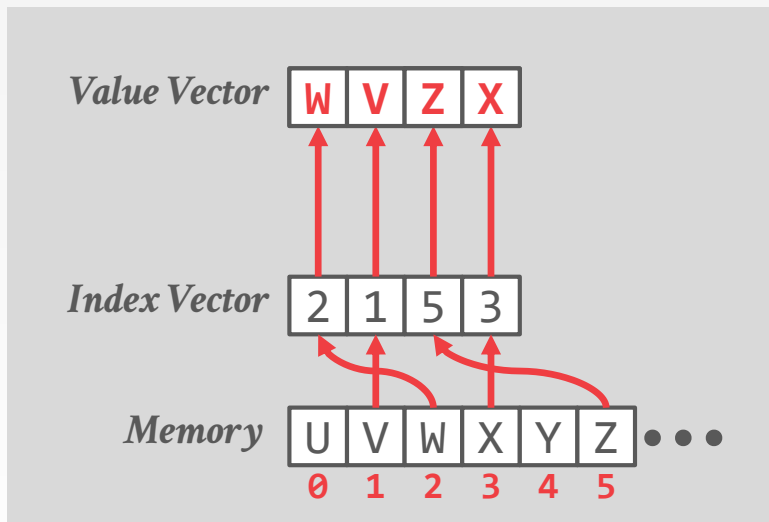


Selective Scatter

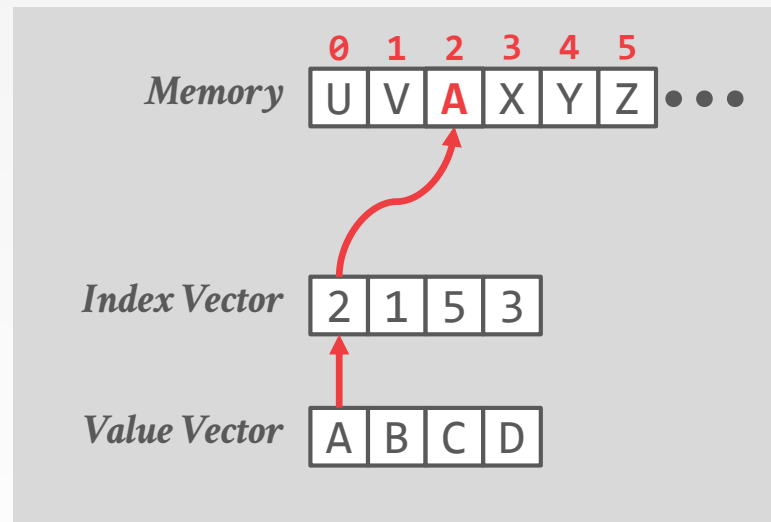


FUNDAMENTAL VECTOR OPERATIONS

Selective Gather

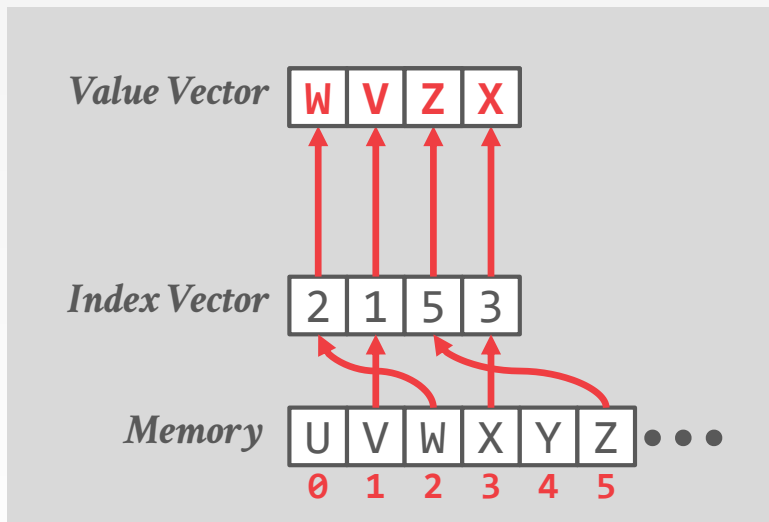


Selective Scatter

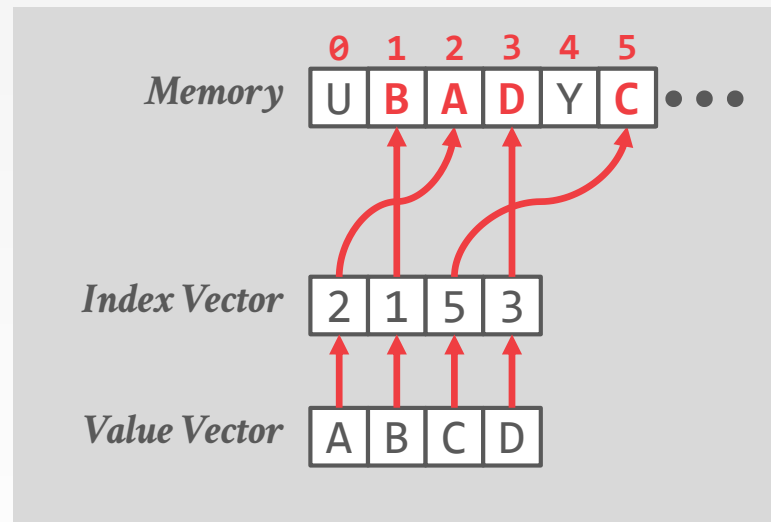


FUNDAMENTAL VECTOR OPERATIONS

Selective Gather



Selective Scatter



ISSUES

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

Gathers are only supported in newer CPUs.

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

VECTORIZED OPERATORS

Selection Scans

Hash Tables

Partitioning / Histograms

Paper provides additional vectorized algorithms:

→ Joins, Sorting, Bloom filters.



SELECTION SCANS

Scalar (Branching)

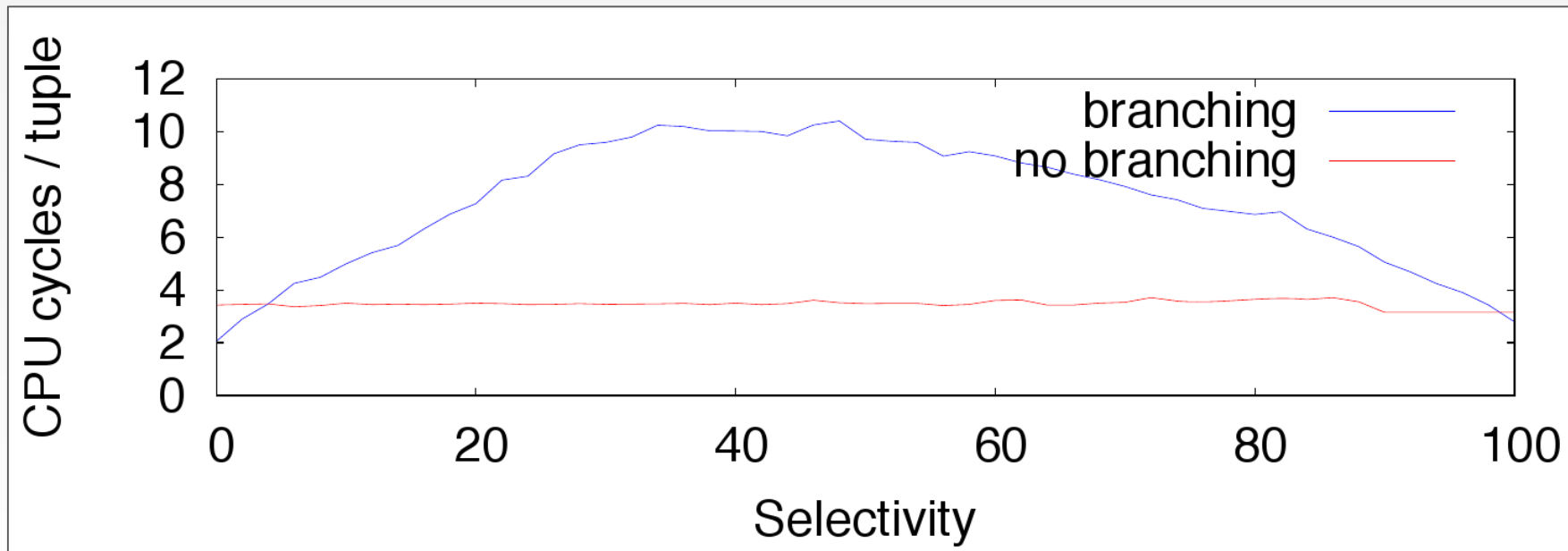
```
i = 0
for t in table:
    key = t.key
    if (key ≥ low) && (key ≤ high):
        copy(t, output[i])
    i = i + 1
```

```
SELECT * FROM table
WHERE key ≥ $low AND key ≤ $high
```

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



Source: [Bogdan Raducanu](#)

SELECTION SCANS

Vectorized

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

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



SELECTION SCANS

Vectorized

```
i = 0
for  $v_t$  in table:
    simdLoad( $v_t$ .key,  $v_k$ )
     $v_m$  = ( $v_k \geq \text{low}$  ? 1 : 0) &
         $\hookrightarrow$  ( $v_k \leq \text{high}$  ? 1 : 0)
    simdStore( $v_t$ ,  $v_m$ , output[i])
    i = i + | $v_m \neq \text{false}$ |
```

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



SELECTION SCANS

Vectorized

```

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

```

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

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

Key Vector

J	O	Y	S	U	X
---	---	---	---	---	---

SELECTION SCANS

Vectorized

```

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

```

SELECT * FROM table
WHERE key ≥ "O" AND key ≤ "U"
  
```

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

Key Vector

J O Y S U X

SIMD Compare

Mask

0 1 0 1 1 0

SELECTION SCANS

Vectorized

```

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

```

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

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

Key Vector

J O Y S U X

SIMD Compare

Mask

0 1 0 1 1 0

All Offsets

0 1 2 3 4 5

SELECTION SCANS

Vectorized

```

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

```

SELECT * FROM table
WHERE key ≥ "O" AND key ≤ "U"
  
```

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

Key Vector

J O Y S U X

SIMD Compare

Mask

0 1 0 1 1 0

All Offsets

0 1 2 3 4 5

SIMD Store

Matched Offsets

1 3 4

SELECTION

◆ Scalar (Branching)

● Scalar (Branchless)

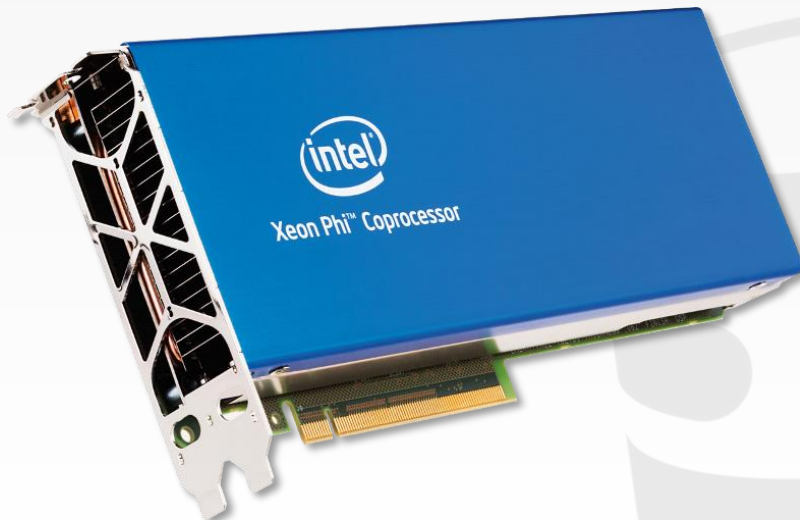
MIC (Xeon Phi 7120P – 61 Cores + 4×HT)



at)

t)

75v3 – 4 Cores + 2×HT)



SELECTION SCANS

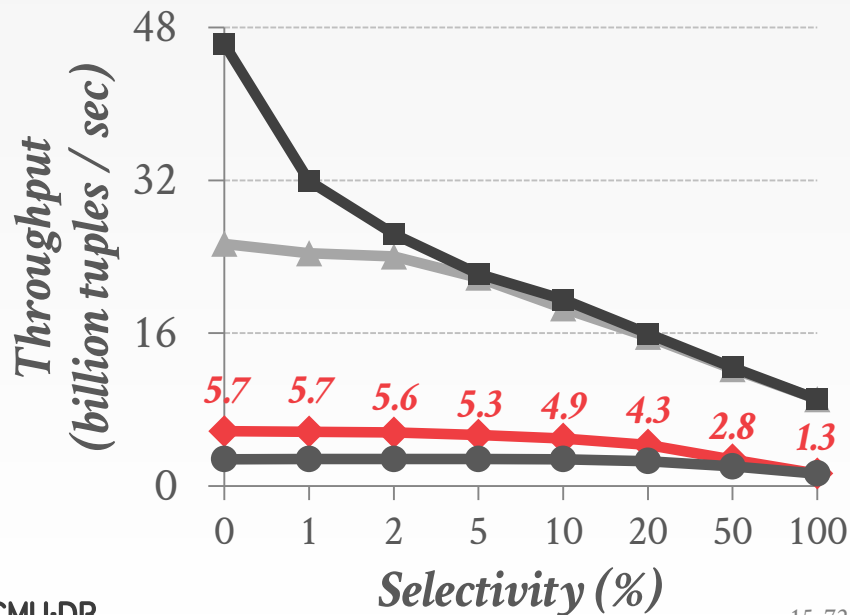
◆ Scalar (Branching)

● Scalar (Branchless)

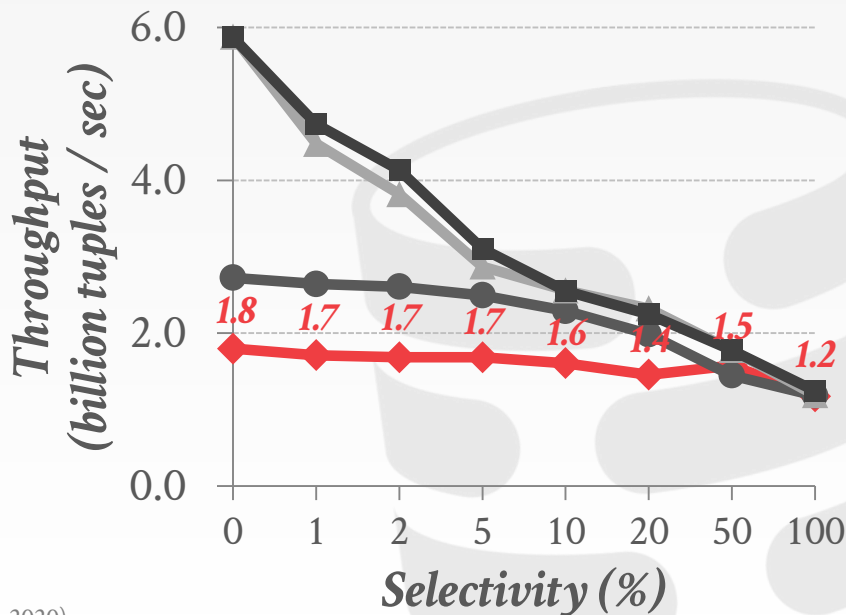
▲ Vectorized (Early Mat)

■ Vectorized (Late Mat)

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



Multi-Core (Xeon E3-1275v3 – 4 Cores + 2×HT)



SELECTION SCANS

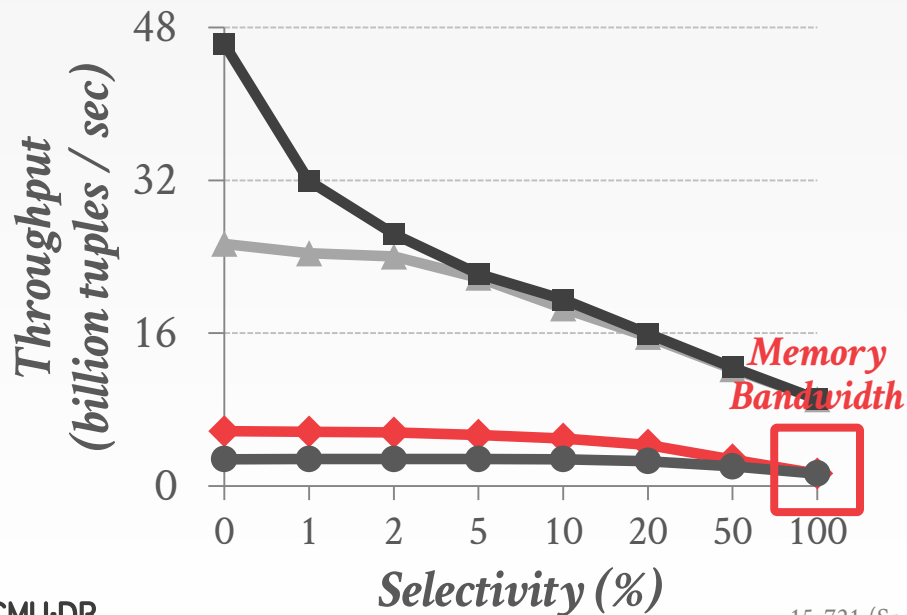
◆ Scalar (Branching)

▲ Vectorized (Early Mat)

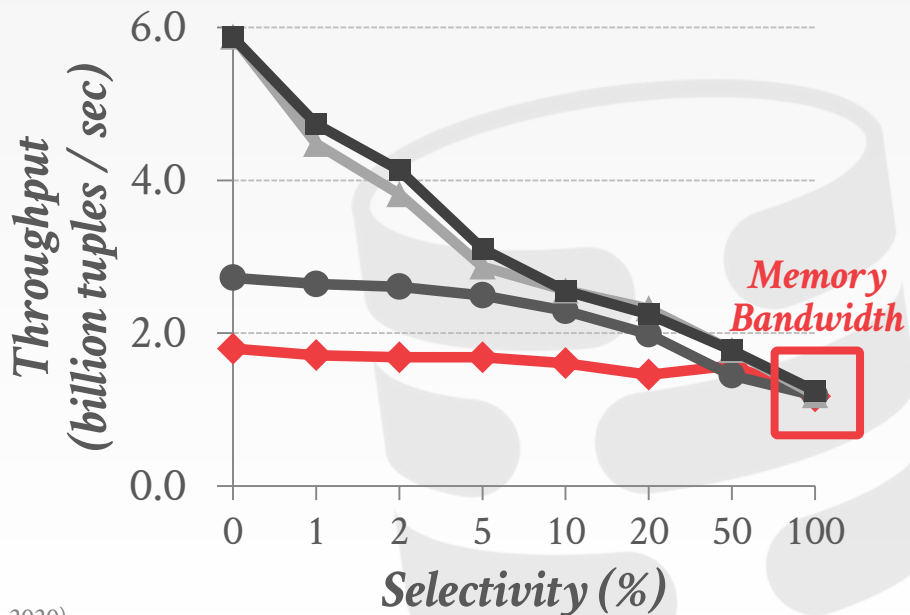
● Scalar (Branchless)

■ Vectorized (Late Mat)

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

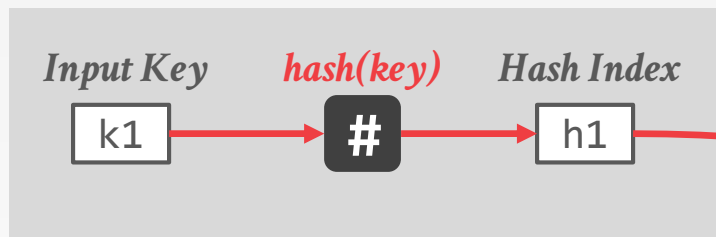


Multi-Core (Xeon E3-1275v3 – 4 Cores + 2×HT)



HASH TABLES – PROBING

Scalar



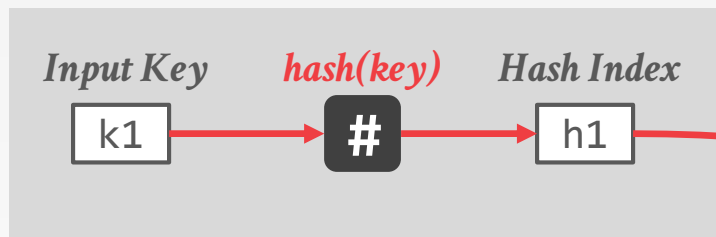
$k1 = k9$

Linear Probing Hash Table

KEY	PAYLOAD

HASH TABLES – PROBING

Scalar



Linear Probing Hash Table

KEY	PAYLOAD

= k9

= k3

= k8

k1 = k1

HASH TABLES – PROBING

Vectorized (Vertical)

*Input Key
Vector*

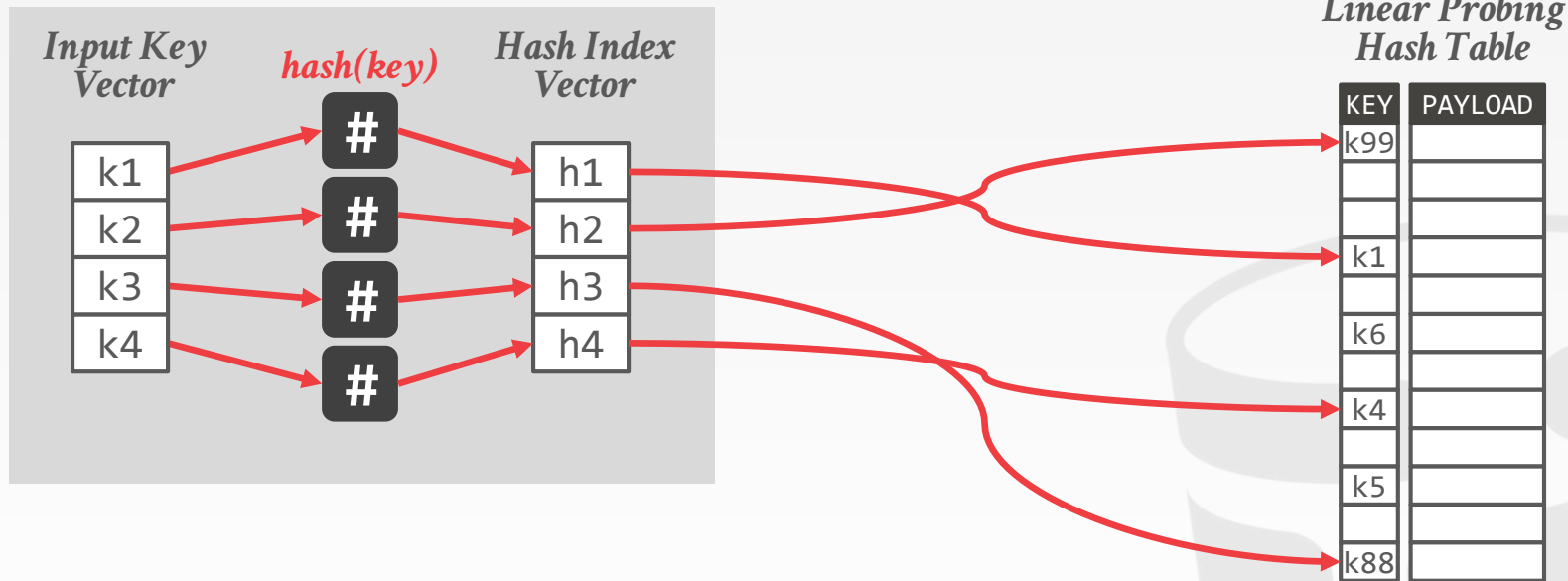
k1
k2
k3
k4

*Linear Probing
Hash Table*

KEY	PAYLOAD
k99	
k1	
k6	
k4	
k5	
k88	

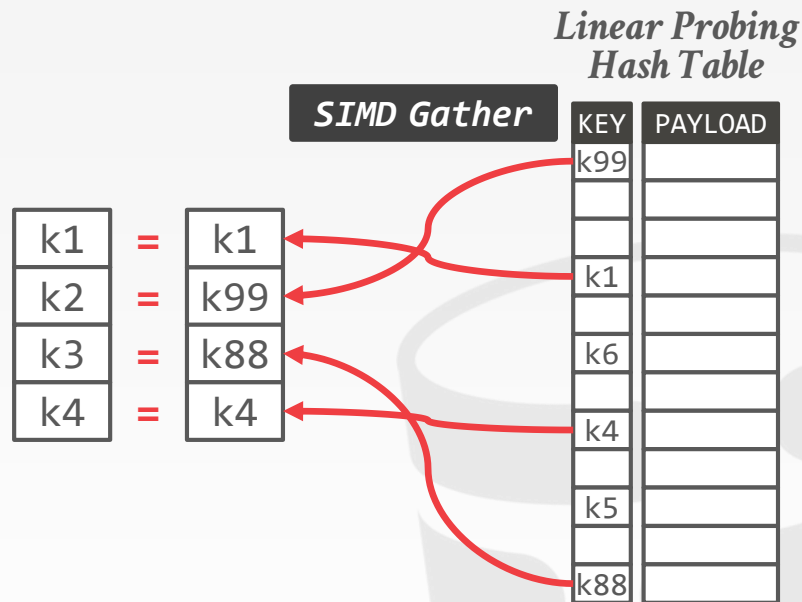
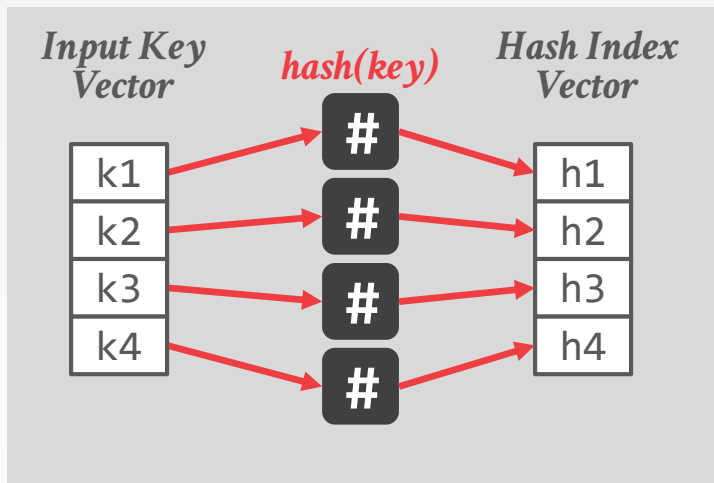
HASH TABLES – PROBING

Vectorized (Vertical)



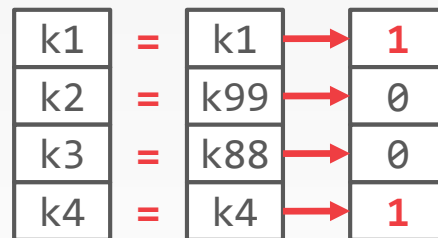
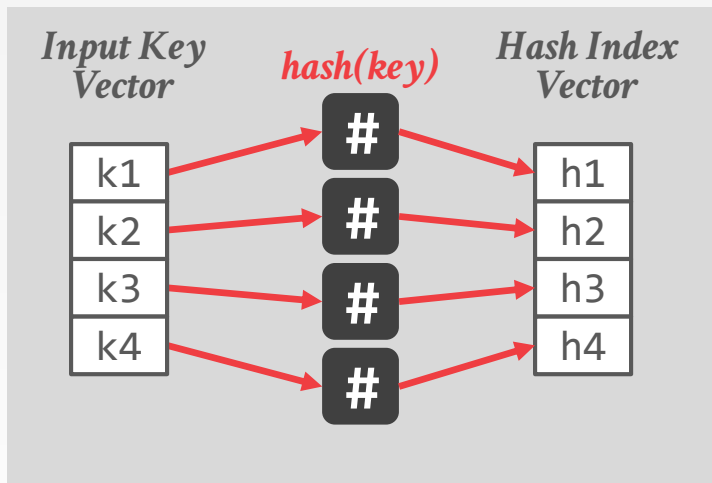
HASH TABLES – PROBING

Vectorized (Vertical)



HASH TABLES – PROBING

Vectorized (Vertical)

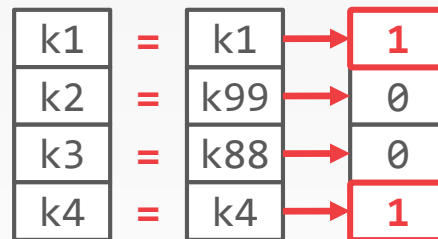
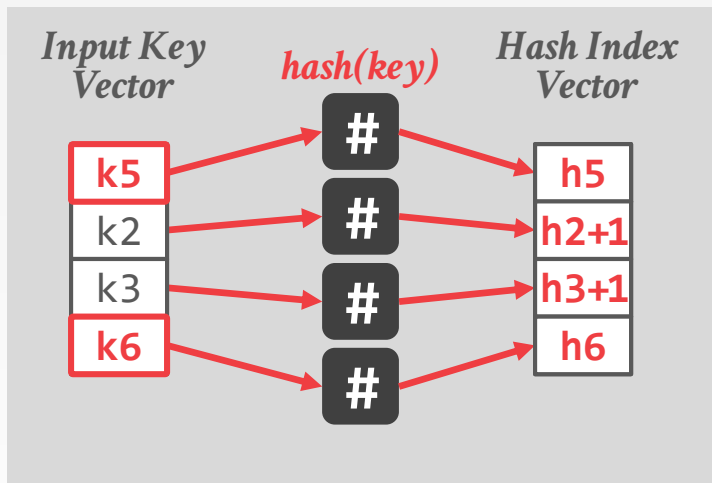


Linear Probing Hash Table

KEY	PAYLOAD
k99	
k1	
k6	
k4	
k5	
k88	

HASH TABLES – PROBING

Vectorized (Vertical)



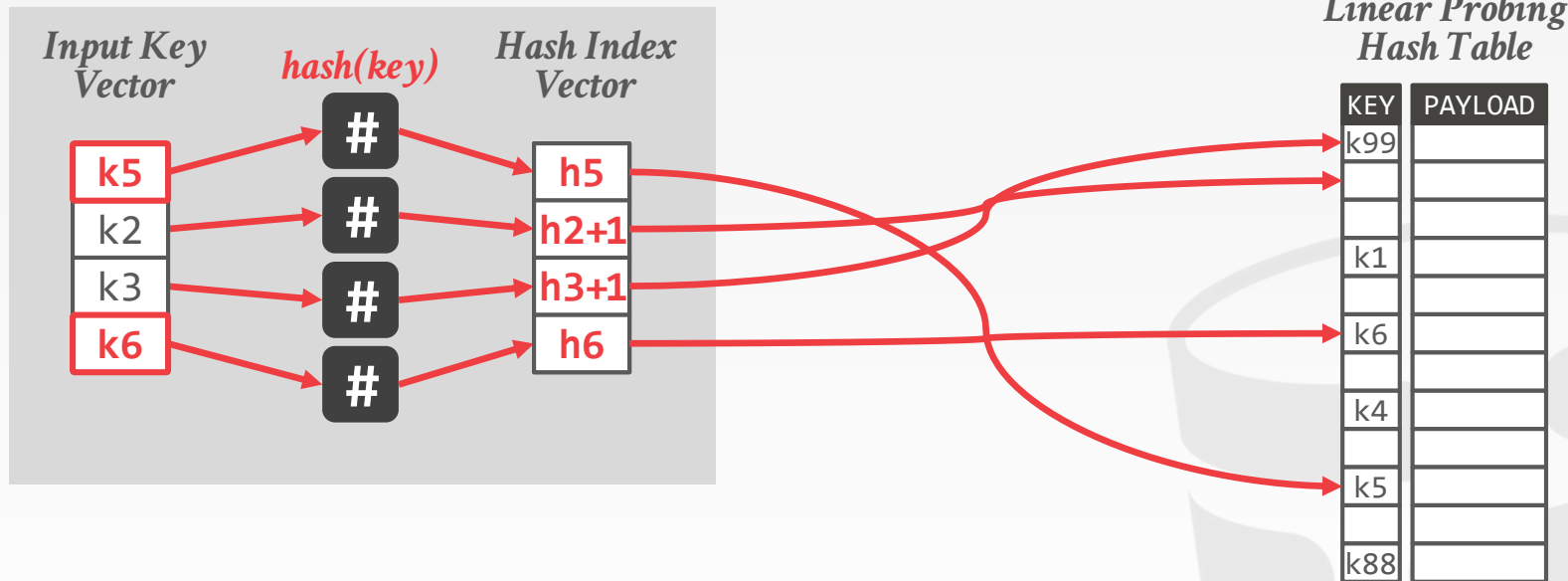
SIMD Compare

Linear Probing Hash Table

KEY	PAYLOAD
k99	
k1	
k6	
k4	
k5	
k88	

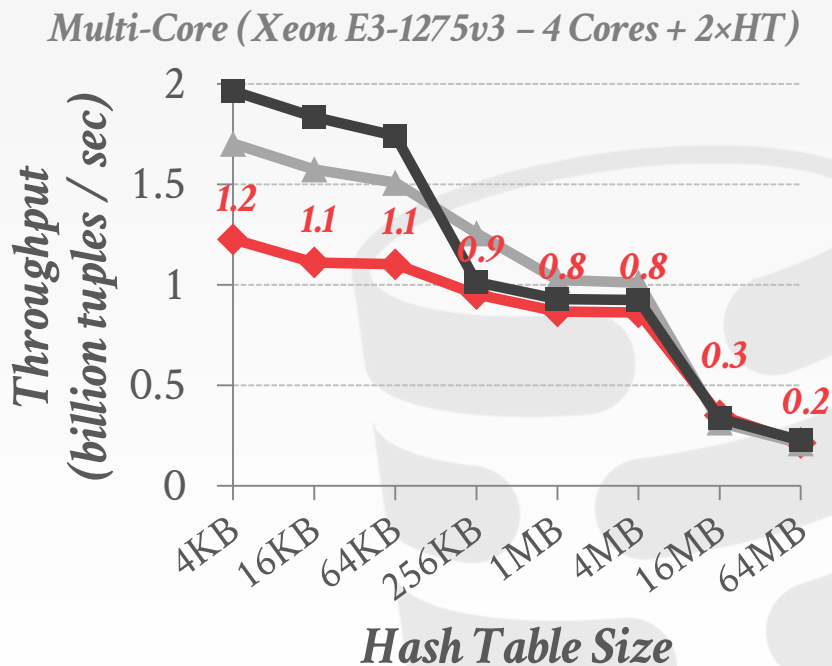
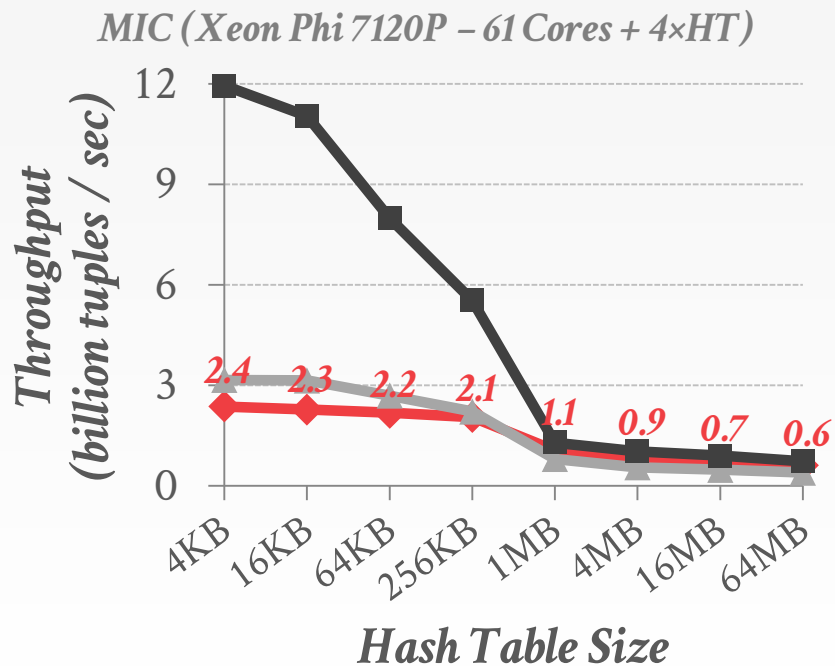
HASH TABLES – PROBING

Vectorized (Vertical)



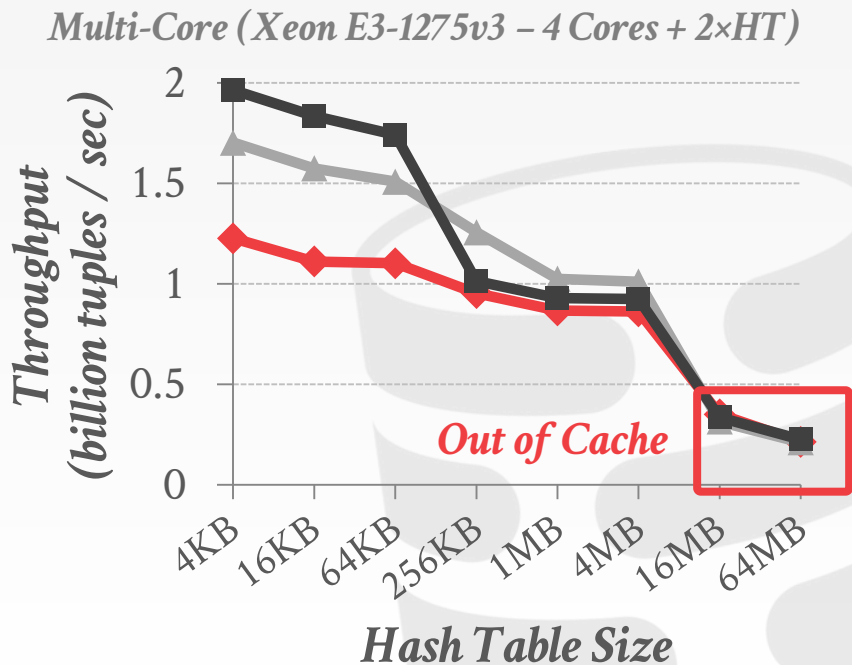
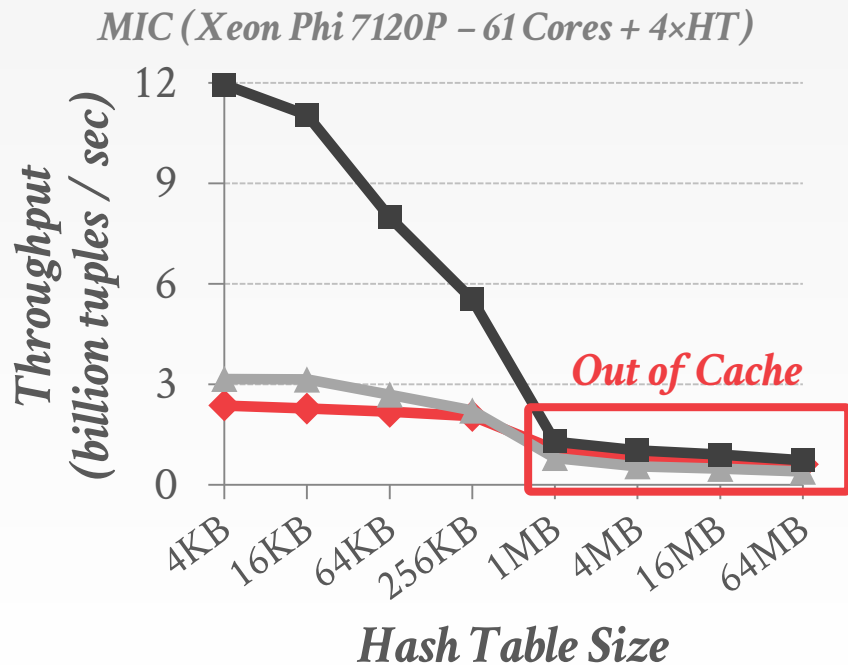
HASH TABLES – PROBING

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



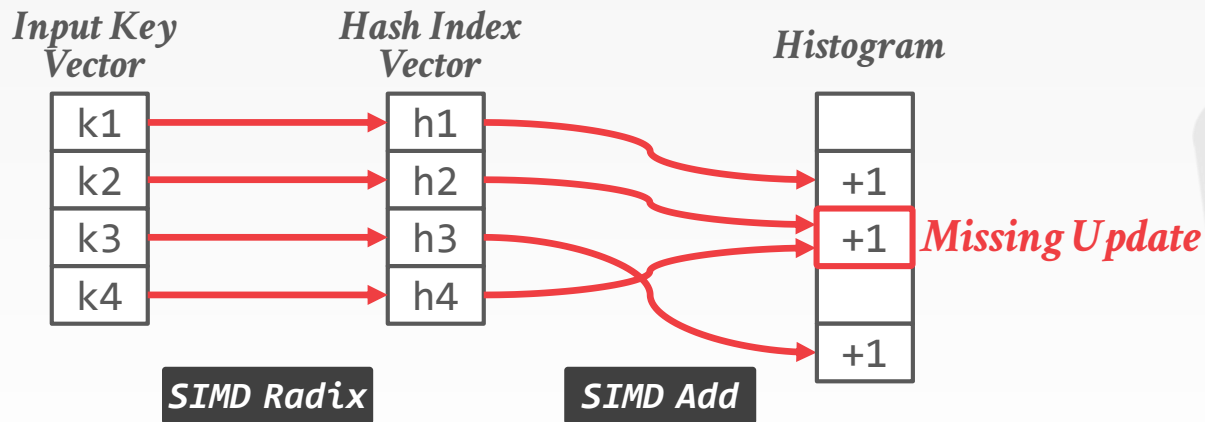
HASH TABLES – PROBING

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



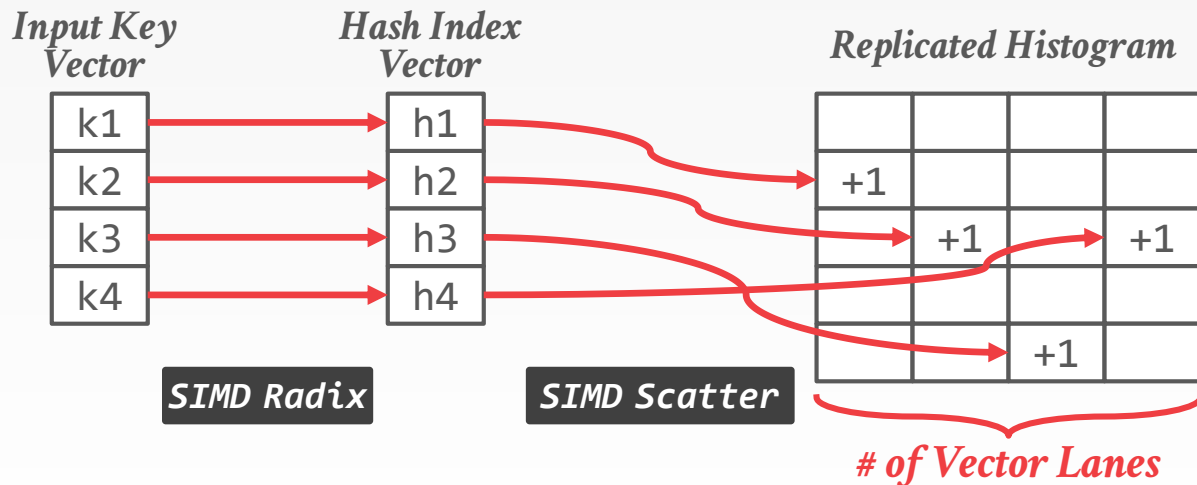
PARTITIONING – HISTOGRAM

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



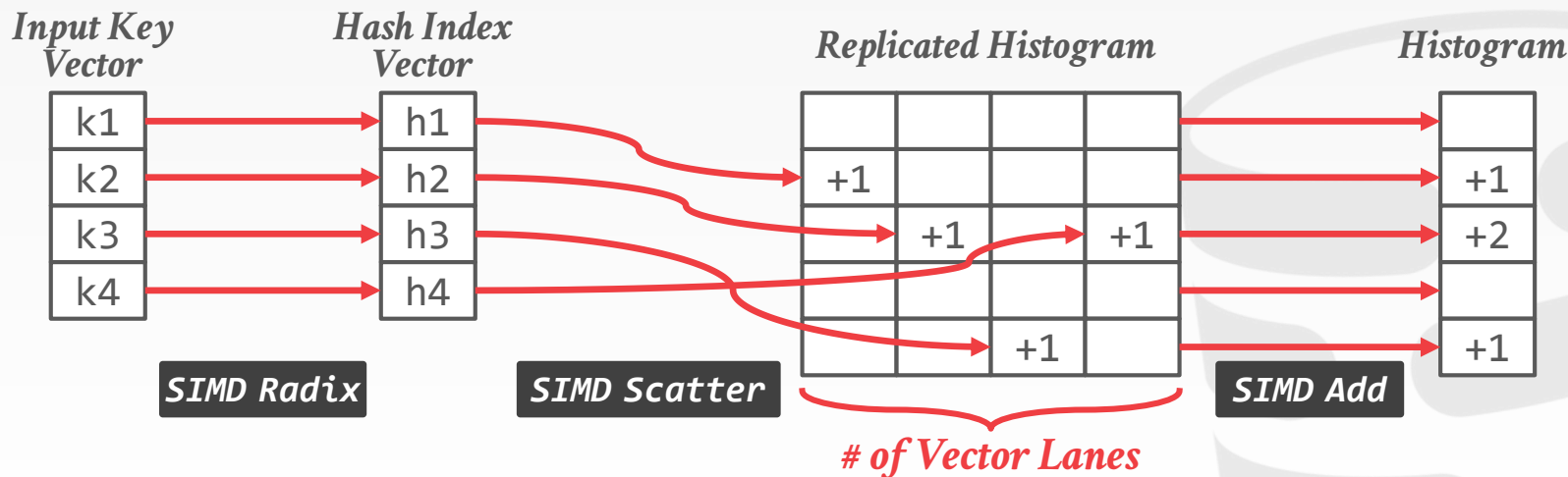
PARTITIONING – HISTOGRAM

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.



PARTING THOUGHTS

Vectorization is essential for OLAP queries.
These algorithms don't work when the data exceeds your CPU cache.

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

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

PROJECT #3

Group project to implement some substantial component or feature in a DBMS.

Projects should incorporate topics discussed in this course as well as from your own interests.

Each group must pick a project that is unique from their classmates.

PROJECT #3

Project deliverables:

- Proposal
- Status Update
- Design Document
- Code Review
- Final Presentation
- Code Drop



PROJECT #3 – PROPOSAL

Five-minute presentation to the class that discusses the high-level topic.

Each proposal must discuss:

- What files you will need to modify.
- How you will test whether your implementation is correct.
- What workloads you will use for your project.

PROJECT #3 – STATUS UPDATE

Five-minute presentation to update the class about the current status of your project.

Each presentation should include:

- Current development status.
- Whether your plan has changed and why.
- Anything that surprised you during coding.



PROJECT #3 – DESIGN DOCUMENT

As part of the status update, you must provide a design document that describes your project implementation:

- Architectural Design
- Design Rationale
- Testing Plan
- Trade-offs and Potential Problems
- Future Work



PROJECT #3 – CODE REVIEW

Each group will be paired with another group and provide feedback on their code.

There will be two separate code review rounds.

Grading will be based on participation.



PROJECT #3 – FINAL PRESENTATION

10-minute presentation on the final status of your project during the scheduled final exam.

You should include any performance measurements or benchmarking numbers for your implementation.

Demos are always hot too...

PROJECT #2 – CODE DROP

A project is **not** considered complete until:

- The code can merge into the master branch without any conflicts.
- All comments from code review are addressed.
- The project includes test cases that correctly verify that implementation is correct.
- Source code contains clear documentation / comments.

We will select the merge order randomly.

COMPUTING RESOURCES

We will provide additional Amazon AWS credits.

Submitting a PR to our repo will invoke builds on Travis and local Jenkins cluster.

Let me know if you think you need special hardware.

DISCLAIMER

The DBMS is a major work-in-progress.

We do not support a bunch of things yet.

We should work together to add in features /
components that we all need.

PROJECT TOPICS

Query Optimizer

Statistics + Sampling

Common Table Expressions

Add/Drop Index

Checkpoints + Recovery

Multi-Threaded Queries

Constraints

Sequences

Additional Data Types

Views

Schema Changes

Compression

QUERY OPTIMIZER

We have a sophisticated query optimizer based on the Cascades model.

Project: Expand features in DBMS optimizer

- Outer Joins
- Nested Queries
- Cost Models
- **Note: You must send me your CV if you choose this project because companies want to hire you. Seriously.**

STATISTICS + SAMPLING

We currently do not maintain any statistics about the database. This is needed for our query optimizer.

Project: Implement a mechanism for collecting statistics about tables for the query optimizer.

- Can choose lazy or eager sampling.
- Add this data to the catalog.
- **Bonus:** Implement a new cost model.

COMMON TABLE EXPRESSIONS

TPC-DS is a more complex workload than TPC-H, but it requires support for CTEs.

Project: Add support for CTEs

- Extend parser to support **WITH** and **UNION**.
- Modify optimizer to reason about derived tables (should be similar to nested queries).
- Extend execution engine to process CTEs.

ADD/DROP INDEXES

We need to support building indexes in a transactionally consistent manner.

Project: Correct index creation/deletion

- Maintain a delta storage for capturing changes made to table while the index is being built.
- Temporarily halt transactions when the index is built and then apply missed changes.
- Bonus: Support building indexes with multiple threads.

MULTI-THREADED QUERIES

The DBMS currently only uses a single worker thread per txn/query.

Project: Implement support for intra-query parallelism with multiple threads.

- Will need to implement this to work with the new LLVM execution engine.
- **Bonus:** Add support for NUMA-aware data placement. Will need to update internal catalog.

CHECKPOINTS + RECOVERY

We currently support a WAL scheme without checkpoints. We can replay log upon restart but not re-install catalogs.

Project: Implement full recovery from checkpoints + WAL.

- First store catalog table in checkpoint
- Add recovery from checkpoint + WAL.
- Implement consistent checkpoints for data tables.

CONSTRAINTS

Constraints are important feature in DBMSs to ensure database integrity.

Project: Implement support for enforcing integrity constraints.

- Will want to start with simple constraints first.
- Final goal will be to implement foreign key constraints.
- Bonus: Online constraint changes with **ALTER**.

SEQUENCES

Global counters that can be used as auto-increment keys for tables.

Project: Add support for Sequences

- Store sequences in the catalog (follow Postgres v12) and make sure increments are durable in WAL.
- Provide helper methods for efficient access.
- Need to special case them from the DBMS's txn manager.
- Add support for **nextval** native function.
- Add support for **SERIAL** attribute type.

NUMERIC TYPE

The Germans claim that fixed-point decimals are faster than floating point decimals.

Project: Add support for NUMERIC type.

- Modify data table to support inline 16-byte values.
- Add new built-in type and operator functions.
- Modify binder with new casting + type-checking methods.
- Modify network layer to serialize values for Postgres wire protocol.

ENUM TYPE

The ENUM type allows the programmer to map names to values and restrict the domain.

Project: Add support for ENUM type

- Update catalogs to store ENUM information.
- Modify binder to enforce the ENUM constraint and then map entries to integers.
- Support ENUM type in LLVM expression evaluation.

VIEWS

A view is a "virtual" table that is based on a **SELECT** query. The DBMS then rewrite queries on that view to be on the underlying query.

Project: Implement support for views.

- Extend the catalog to store view information.
- Modify the binder to transform the view to use the originally query.
- Should not need to modify execution engine.

CONCURRENT SCHEMA CHANGES

A DBMS needs to be able to support updating the database schema (e.g., add/drop column) while it continues to execute txns and queries.

Project: Implement support for concurrent schema changes with low overhead.

- You will want to use a lazy method that propagates changes to blocks only when they are updated.
- Will want to extend internal catalog to keep track of different schema versions.

DATABASE COMPRESSION

We currently support Apache Arrow's dictionary compression scheme, but it is not turned on

Project: Enable Arrow dictionary compression.

- Implement support to convert uncompressed blocks to compressed blocks. Must also update indexes.
- Extend data table API to allow execution engine scan operations to process on compressed data.

HOW TO START

Form a team.

Meet with your team and discuss potential topics.

Look over source code and determine what you will need to implement.

I am able during Spring Break for additional discussion and clarification of the project idea.

NEXT CLASS

Proposal Presentations

