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

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

Background
Implementation Approaches
SIMD Fundamentals
Vectorized DBMS Algorithms
VECTORIZATION

The process of converting an algorithm's scalar implementation that processes a single pair of operands at a time, to a vector implementation that processes one operation on multiple pairs of operands at once.
WHY THIS MATTERS

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

Potential Speed-up: $32x \times 4x = 128x$
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
→ **RISC-V**: RVV
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

\(X + Y = Z\)

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

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

### SIMD Example

**$X + Y = Z$**

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

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

$$X + Y = Z$$

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

```c
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];
}

128-bit SIMD Register

128-bit SIMD Register

128-bit SIMD Register
**SIMD EXAMPLE**

\[ X + Y = Z \]

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

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

Approach #1: Horizontal
→ Perform operation on all elements together within a single vector.

Approach #2: Vertical
→ Perform operation in an elementwise manner on elements of each vector.

Source: Przemysław Karpiński
SIMD INSTRUCTIONS (1)

Data Movement
→ Moving data in and out of vector registers

Arithmetic Operations
→ Apply operation on multiple data items (e.g., 2 doubles, 4 floats, 16 bytes)
→ Example: ADD, SUB, MUL, DIV, SQRT, MAX, MIN

Logical Instructions
→ Logical operations on multiple data items
→ Example: AND, OR, XOR, ANDN, ANDPS, ANDNPS
SIMD INSTRUCTIONS (2)

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

Shuffle instructions
→ Move data between SIMD registers

Miscellaneous
→ Conversion: Transform data between x86 and SIMD registers.
→ Cache Control: Move data directly from SIMD registers to memory (bypassing CPU cache).
## INTEL SIMD EXTENSIONS

<table>
<thead>
<tr>
<th>Year</th>
<th>Extension</th>
<th>Width</th>
<th>Integers</th>
<th>Single-P</th>
<th>Double-P</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997</td>
<td>MMX</td>
<td>64 bits</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1999</td>
<td>SSE</td>
<td>128 bits</td>
<td>✔</td>
<td>✔(×4)</td>
<td></td>
</tr>
<tr>
<td>2001</td>
<td>SSE2</td>
<td>128 bits</td>
<td>✔</td>
<td>✔</td>
<td>✔(×2)</td>
</tr>
<tr>
<td>2004</td>
<td>SSE3</td>
<td>128 bits</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>2006</td>
<td>SSSE 3</td>
<td>128 bits</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>2006</td>
<td>SSE 4.1</td>
<td>128 bits</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>2008</td>
<td>SSE 4.2</td>
<td>128 bits</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>2011</td>
<td>AVX</td>
<td>256 bits</td>
<td>✔</td>
<td>✔(×8)</td>
<td>✔(×4)</td>
</tr>
<tr>
<td>2013</td>
<td>AVX2</td>
<td>256 bits</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>AVX-512</td>
<td>512 bits</td>
<td>✔</td>
<td>✔(×16)</td>
<td>✔(×8)</td>
</tr>
</tbody>
</table>

Source: James Reinders
SIMD TRADE-OFFS

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

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

No Longer True in AVX-512!
AVX-512

Intel's 512-bit extensions to the AVX2 instructions.  
→ Provides new operations to support data conversions, scatter, and permutations.

Unlike previous SIMD extensions, Intel split AVX-512 into groups that CPUs can selectively provide (except for "foundation" extension AVX-512F).
Intel's 512-bit extensions to the AVX2 instructions. → Provides new operations to support data conversions, scatter, and permutations.
AVX-512

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).
IMPLEMENTATION

Choice #1: Automatic Vectorization
Choice #2: Compiler Hints
Choice #3: Explicit Vectorization

Ease of Use

Programming Control
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.
This loop is not legal to automatically vectorize.

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

These might point to the same address!
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.
The `restrict` keyword in C++ tells the compiler that the arrays are distinct locations in memory.

```c++
void add(int *restrict X,
         int *restrict Y,
         int *restrict Z) {
    for (int i=0; i<MAX; i++) {
        Z[i] = X[i] + Y[i];
    }
}
```
This pragma tells the compiler to ignore loop dependencies for the vectors.

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

Use CPU intrinsics to manually marshal data between SIMD registers and execute vectorized instructions.
→ Not portable across CPUs (ISAs / versions).

There are libraries that hide the underlying calls to SIMD intrinsics.
→ Google Highway
→ Simd
→ Expressive Vector Engine (EVE)
→ std::simd (Experimental)
**EXPLICIT VECTORIZATION**

Store the vectors in 128-bit SIMD registers.

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

```c
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++));
    }
}
```
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.

```
<table>
<thead>
<tr>
<th>Merge Source</th>
<th>9 9 9 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector1</td>
<td>3 3 3 3</td>
</tr>
<tr>
<td>Mask</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>Vector2</td>
<td>2 2 2 2</td>
</tr>
</tbody>
</table>
```
Almost all AVX-512 operations support **predication** variants whereby the CPU only performs operations on lanes specified by an input bitmask.

![Diagram](image-url)
Almost all AVX-512 operations support **predication** variants whereby the CPU only performs operations on lanes specified by an input bitmask.
Permute

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

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

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.
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.
Selective Load

Vector: A U C D

Mask: 0 1 0 1

Memory: U V W X Y Z •••
**Selective Load/Store**

**Selective Load**

**Vector**

A  U  C  V

**Mask**

0  1  0  1

**Memory**

U  V  W  X  Y  Z  • • •
SELECTIVE LOAD/STORE

**Selective Load**

Vector: A U C V

Mask: 0 1 0 1

Memory: U V W X Y Z

**Selective Store**

Memory: U V W X Y Z

Mask: 0 1 0 1

Vector: A B C D
**Selective Load/Store**

**Selective Load**
- **Vector**: \(\begin{array}{cccc} A & U & C & V \end{array}\)
- **Mask**: \(\begin{array}{cccc} 0 & 1 & 0 & 1 \end{array}\)
- **Memory**: \(\begin{array}{cccc} U & V & W & X & Y & Z \end{array}\)

**Selective Store**
- **Memory**: \(\begin{array}{cccc} B & V & W & X & Y & Z \end{array}\)
- **Mask**: \(\begin{array}{cccc} 0 & 1 & 0 & 1 \end{array}\)
- **Vector**: \(\begin{array}{cccc} A & B & C & D \end{array}\)
SELECTIVE LOAD/STORE

**Selective Load**

Vector: A U C V

Mask: 0 1 0 1

Memory: U V W X Y Z

**Selective Store**

Memory: B D W X Y Z

Mask: 0 1 0 1

Vector: A B C D
**Compress**

<table>
<thead>
<tr>
<th>Value Vector</th>
<th>A</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Index Vector</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Input Vector</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>
**Compress**

**Value Vector**

| A | D |

**Index Vector**

| 1 | 0 | 0 | 1 |

**Input Vector**

| A | B | C | D |
**Compress**

**Value Vector**

\[
\begin{array}{cccc}
A & D & 0 & 0 \\
\end{array}
\]

**Index Vector**

\[
\begin{array}{cccc}
1 & 0 & 0 & 1 \\
\end{array}
\]

**Input Vector**

\[
\begin{array}{cccc}
A & B & C & D \\
\end{array}
\]

**COMPRESS / EXPAND**
**Compress / Expand**

**Compress**

- **Value Vector**: A D 0 0
- **Index Vector**: 1 0 0 1
- **Input Vector**: A B C D

**Expand**

- **Value Vector**: 
- **Index Vector**: 0 1 0 1
- **Input Vector**: A B C D
COMPRESS / EXPAND

**Compress**

- **Value Vector**: A D 0 0
- **Index Vector**: 1 0 0 1
- **Input Vector**: A B C D

**Expand**

- **Value Vector**: A
- **Index Vector**: 0 1 0 1
- **Input Vector**: A B C D
COMPRESSION / EXPANSION

**Compress**

<table>
<thead>
<tr>
<th>Value Vector</th>
<th>Index Vector</th>
<th>Input Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>A D 0 0</td>
<td>1 0 0 1</td>
<td>A B C D</td>
</tr>
</tbody>
</table>

**Expand**

<table>
<thead>
<tr>
<th>Value Vector</th>
<th>Index Vector</th>
<th>Input Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B</td>
<td>0 1 0 1</td>
<td>A B C D</td>
</tr>
</tbody>
</table>
COMPRESS / EXPAND

**Compress**

- **Value Vector**: A D 0 0
- **Index Vector**: 1 0 0 1
- **Input Vector**: A B C D

**Expand**

- **Value Vector**: A B 0 0
- **Index Vector**: 0 1 0 1
- **Input Vector**: A B C D
Selective Gather

**Value Vector**

A
B
A
D

**Index Vector**

2
1
5
3

**Memory**

U
V
W
X
Y
Z
0
1
2
3
4
5
Selective Gather

Value Vector: W B A D

Index Vector: 2 1 5 3

Memory: U V W X Y Z • • •
Selective Gather

Value Vector

Index Vector

Memory
SELECTIVE SCATTER/GATHER

Selective Gather

Value Vector

Index Vector

Memory

Selective Scatter

Value Vector

Index Vector

Memory
**Selective Gather**

Value Vector: W V Z X

Index Vector: 2 1 5 3

Memory: U V W X Y Z

---

**Selective Scatter**

Memory: U V W X Y Z

Index Vector: 2 1 5 3

Value Vector: A B C D
Selective Scatter/Gather

**Selective Gather**

Value Vector: W V Z X

Index Vector: 2 1 5 3

Memory: U V W X Y Z

**Selective Scatter**

Value Vector: A B C D

Index Vector: 2 1 5 3

Memory: U B W D Y C
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).
VECTORIZED OPERATORS

Selection Scans
Hash Tables
Partitioning / Histograms
Scalar (Branchless)

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

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

**Vectorized**

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

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

- Scalar (Branching)
- Scalar (Branchless)
- Vectorized (Early Mat)
- Vectorized (Late Mat)

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

Source: Orestis Polychroniou


**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)**

Throughput (billion tuples / sec) vs. Selectivity (%)

Source: Orestis Polychroniou
### SELECTION SCANS

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

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

- **Scalar (Branching)**
- **Scalar (Branchless)**
- **Vectorized (Early Mat)**
- **Vectorized (Late Mat)**

**Throughput** (billion tuples / sec) vs **Selectivity** (%)

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

```python
agg = dict()
for t in table:
    if t.age > 20:
        agg[t.city]["count"]++
for t in agg:
    emit(t)
```

```sql
SELECT COUNT(*) FROM table
WHERE age > 20
GROUP BY city;
```
For each batch, the SIMD vectors may contain tuples that are no longer valid (they were disqualified by some previous check).

Scan
Filter
Agg
Emit

Pipeline #1

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

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

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

Pipeline #1

- Scan
- Filter
- Agg
- Emit

Pipeline #2

- Emit
- Agg
- Filter

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

agg = dict()

for t in table:
    if t.age > 20:
        agg[t.city]["count"]++

for t in agg:
    emit(t)
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.
SELECT COUNT(*) FROM table 
WHERE age > 20 GROUP BY city;
ROF EXAMPLE

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

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

Stage #1
- **Scan**
- **Filter**
- **Agg**
- **Emit**

Stage #2
- **Stage Buffer**

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

agg = dict()
for vt in table step 1024:
    buffer = simd_cmp_gt(vt, 20, 1024)
    if |buffer| >= MAX:
        for t in buffer:
            agg[t.city]["count"]++
    for t in agg:
        emit(t)
SELECT COUNT(*) FROM table
WHERE age > 20 GROUP BY city;

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

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

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

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

- **LLVM**  
- **LLVM + ROF**

<table>
<thead>
<tr>
<th></th>
<th>LLVM</th>
<th>LLVM + ROF</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Q1</strong></td>
<td>901</td>
<td>892</td>
</tr>
<tr>
<td><strong>Q3</strong></td>
<td>1396</td>
<td>846</td>
</tr>
<tr>
<td><strong>Q13</strong></td>
<td>2641</td>
<td>1763</td>
</tr>
<tr>
<td><strong>Q14</strong></td>
<td>383</td>
<td>191</td>
</tr>
<tr>
<td><strong>Q19</strong></td>
<td>540</td>
<td>220</td>
</tr>
</tbody>
</table>

**SIMD/Prefetch Does Not Help**

**SIMD/Prefetch Does Help**

Source: Prashanth Menon

CMU-DB 15-721 (Spring 2023)
ROF EVALUATION – TPC-H Q19

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

<table>
<thead>
<tr>
<th></th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreted</td>
<td>21475</td>
</tr>
<tr>
<td>Compiled</td>
<td>568</td>
</tr>
<tr>
<td>ROF + SIMD</td>
<td>196</td>
</tr>
<tr>
<td>ROF + SIMD + Preetching</td>
<td>189</td>
</tr>
</tbody>
</table>

Source: Prashanth Menon
VECTORIZED OPERATORS

Selection Scans
Hash Tables
Partitioning / Histograms
HASH TABLES – PROBING

Scalar

Input Key \( k_1 \)  \( \xrightarrow{\text{hash(key)}} \) Hash Index \( h_1 \)

Linear Probing Hash Table

\[
k_1 = k_9
\]
HASH TABLES – PROBING

Scalar

- Input Key: k1
- hash(key): #
- Hash Index: h1

Linear Probing Hash Table

<table>
<thead>
<tr>
<th>KEY</th>
<th>PAYLOAD</th>
</tr>
</thead>
</table>

- k1 = k1
- k9
- k3
- k8

h1
## HASH TABLES – PROBING

### Scalar

<table>
<thead>
<tr>
<th>Input Key</th>
<th>$\text{hash(key)}$</th>
<th>Hash Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k_1$</td>
<td>$#$</td>
<td>$h_1$</td>
</tr>
</tbody>
</table>

### Vectorized (Horizontal)

<table>
<thead>
<tr>
<th>Input Key</th>
<th>$\text{hash(key)}$</th>
<th>Hash Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k_1$</td>
<td>$#$</td>
<td>$h_1$</td>
</tr>
</tbody>
</table>

### Linear Probing Bucketized Hash Table

<table>
<thead>
<tr>
<th>KEYS</th>
<th>PAYLOAD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Four Keys  Four Values
HASH TABLES – PROBING

Scalar

Input Key  \( \text{hash(key)} \)  Hash Index
k1  #  h1

Vectorized (Horizontal)

Input Key  \( \text{hash(key)} \)  Hash Index
k1  #  h1

Linear Probing Bucketized Hash Table

<table>
<thead>
<tr>
<th>KEYS</th>
<th>PAYLOAD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Matched Mask

\[ 0 \ 0 \ 0 \ 1 \]

SIMD Compare
Four Keys  Four Values
**Vectorized (Vertical)**

**Input Key Vector**
- $k_1$
- $k_2$
- $k_3$
- $k_4$

**hash(key)**
- #
- #
- #
- #

**Hash Index Vector**
- $h_1$
- $h_2$
- $h_3$
- $h_4$

**Linear Probing Hash Table**
- **KEY**
  - k99
  - k1
  - k6
  - k4
  - k5
  - k88
- **PAYLOAD**
  - ---
  - ---
  - ---
  - ---
  - ---
HASH TABLES – PROBING

Vectorized (Vertical)

Input Key Vector  hash(key)  Hash Index Vector

k1  #  h1
k2  #  h2
k3  #  h3
k4  #  h4

Linear Probing Hash Table

<table>
<thead>
<tr>
<th>KEY</th>
<th>PAYLOAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>k99</td>
<td></td>
</tr>
<tr>
<td>k1</td>
<td></td>
</tr>
<tr>
<td>k2</td>
<td></td>
</tr>
<tr>
<td>k6</td>
<td></td>
</tr>
<tr>
<td>k3</td>
<td></td>
</tr>
<tr>
<td>k88</td>
<td></td>
</tr>
<tr>
<td>k4</td>
<td></td>
</tr>
<tr>
<td>k5</td>
<td></td>
</tr>
<tr>
<td>k88</td>
<td></td>
</tr>
</tbody>
</table>

SimD Gather

k1 = k1
k2 = k99
k3 = k88
k4 = k4
Vectorized (Vertical)

Input Key Vector

<table>
<thead>
<tr>
<th>k1</th>
<th>#</th>
<th>h1</th>
</tr>
</thead>
<tbody>
<tr>
<td>k2</td>
<td>#</td>
<td>h2</td>
</tr>
<tr>
<td>k3</td>
<td>#</td>
<td>h3</td>
</tr>
<tr>
<td>k4</td>
<td>#</td>
<td>h4</td>
</tr>
</tbody>
</table>

Hash Index Vector

<table>
<thead>
<tr>
<th>h1</th>
<th>#</th>
<th>h2</th>
</tr>
</thead>
<tbody>
<tr>
<td>h3</td>
<td>#</td>
<td>h4</td>
</tr>
</tbody>
</table>

**Linear Probing Hash Table**

<table>
<thead>
<tr>
<th>KEY</th>
<th>PAYLOAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>k99</td>
<td></td>
</tr>
<tr>
<td>k1</td>
<td></td>
</tr>
<tr>
<td>k6</td>
<td></td>
</tr>
<tr>
<td>k4</td>
<td></td>
</tr>
<tr>
<td>k5</td>
<td></td>
</tr>
<tr>
<td>k88</td>
<td></td>
</tr>
</tbody>
</table>

**SIMD Compare**

- k1 = k1 → 1
- k2 = k99 → 0
- k3 = k88 → 0
- k4 = k4 → 1
**Vectorized (Vertical)**

**Input Key Vector** → **hash(key)** → **Hash Index Vector**

- **k5** → hash(key) → h5
- **k2** → hash(key) → h2+1
- **k3** → hash(key) → h3+1
- **k6** → hash(key) → h6

**Linear Probing Hash Table**

<table>
<thead>
<tr>
<th>KEY</th>
<th>PAYLOAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>k99</td>
<td></td>
</tr>
<tr>
<td>k1</td>
<td></td>
</tr>
<tr>
<td>k6</td>
<td></td>
</tr>
<tr>
<td>k4</td>
<td></td>
</tr>
<tr>
<td>k88</td>
<td></td>
</tr>
</tbody>
</table>

**SIMD Compare**

- **k1** = **k1** → 1
- **k2** = **k99** → 0
- **k3** = **k88** → 0
- **k4** = **k4** → 1
HASH TABLES – PROBING

Vectorized (Vertical)

Input Key Vector

| k5 | k2 | k3 | k6 |

hash(key)

| | # | | |

Hash Index Vector

| h5 | h2+1 | h3+1 | h6 |

Linear Probing Hash Table

<table>
<thead>
<tr>
<th>KEY</th>
<th>PAYLOAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>k99</td>
<td></td>
</tr>
<tr>
<td>k1</td>
<td></td>
</tr>
<tr>
<td>k6</td>
<td></td>
</tr>
<tr>
<td>k4</td>
<td></td>
</tr>
<tr>
<td>k5</td>
<td></td>
</tr>
<tr>
<td>k88</td>
<td></td>
</tr>
</tbody>
</table>
**HASH TABLES – PROBING**

- **MIC (Xeon Phi 7120P – 61 Cores + 4×HT)**
- **Multi-Core (Xeon E3-1275v3 – 4 Cores + 2×HT)**

Throughput (billion tuples/sec) vs Hash Table Size

**Source:** Orestis Polychroniou
**HASH TABLES – PROBING**

Scal"ar △ Vectorized (Horizontal) □ Vectorized (Vertical)

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

<table>
<thead>
<tr>
<th>Hash Table Size</th>
<th>Throughput (billion tuples/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4KB</td>
<td>12</td>
</tr>
<tr>
<td>16KB</td>
<td>9</td>
</tr>
<tr>
<td>64KB</td>
<td>6</td>
</tr>
<tr>
<td>256KB</td>
<td>3</td>
</tr>
<tr>
<td>1MB</td>
<td>1</td>
</tr>
<tr>
<td>4MB</td>
<td>0.5</td>
</tr>
<tr>
<td>16MB</td>
<td>0.3</td>
</tr>
<tr>
<td>64MB</td>
<td>0.1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Hash Table Size</th>
<th>Throughput (billion tuples/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4KB</td>
<td>2</td>
</tr>
<tr>
<td>16KB</td>
<td>1.5</td>
</tr>
<tr>
<td>64KB</td>
<td>1</td>
</tr>
<tr>
<td>256KB</td>
<td>0.5</td>
</tr>
<tr>
<td>1MB</td>
<td>0.3</td>
</tr>
<tr>
<td>4MB</td>
<td>0.1</td>
</tr>
<tr>
<td>16MB</td>
<td>0.05</td>
</tr>
<tr>
<td>64MB</td>
<td>0.01</td>
</tr>
</tbody>
</table>

*Source: Orestis Polychroniou*
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.
Use scatter and gathers to increment counts. Replicate the histogram to handle collisions.
Use scatter and gathers to increment counts. Replicate the histogram to handle collisions.
CAVEAT EMPTOR

AVX-512 is **not** always faster than AVX2.
AVX-512 is **not** always faster than AVX-2.

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

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

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

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

AVX-512 is not always faster than AVX2.

Some CPUs downgrade their clock speed when switching to AVX-512 mode.

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

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

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

The exact speeds for each license also depend on the number of active cores. For up to four cores, you can usually consult Wikipedia. For example, the table for the Xeon Gold 5121 is here:

```
<table>
<thead>
<tr>
<th>Width</th>
<th>Light</th>
<th>Heavy</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar</td>
<td>L0</td>
<td>W/A</td>
</tr>
<tr>
<td>128-Bit</td>
<td>L0</td>
<td>L0</td>
</tr>
<tr>
<td>256-Bit</td>
<td>L0</td>
<td>L1*</td>
</tr>
<tr>
<td>512-Bit</td>
<td>L1</td>
<td>L2*</td>
</tr>
</tbody>
</table>
```

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

Vectorization is essential for OLAP queries.

We can combine all the intra-query parallelism optimizations we’ve talked about in a DBMS.  
→ Multiple threads processing the same query.  
→ Each thread can execute a compiled plan.  
→ The compiled plan can invoke vectorized operations.
Query Compilation
Project #3 Topics