Vectorized Query Execution
We described how the DBMS divides up pipelines and runs the in parallel (Task Parallelization).

We also discussed how a DBMS will evaluate expressions and introduced the idea of query adaptivity.
TODAY’S AGENDA

Background
Implementation Approaches
Vectorization Fundamentals
Vectorized DBMS Algorithms
The process of converting an algorithm's scalar implementation that processes a single pair of operands at a time, to a vector implementation that processes one operation on multiple pairs of operands at once.

This technique is known as **Data Parallelization**.
Suppose the DBMS can parallelize some 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, SVE2
→ **RISC-V**: RVV
SIMD EXAMPLE

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

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

\[ X + Y = Z \]

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

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

\[ X + Y = Z \]

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

```java
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

\[ X + Y = Z \]

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

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

\[\begin{array}{cccc}
8 & 7 \\
6 & 5 \\
4 & 3 \\
2 & 1 \\
\end{array}\] + 
\[\begin{array}{cccc}
1 & 1 \\
1 & 1 \\
1 & 1 \\
1 & 1 \\
\end{array}\] = 
\[\begin{array}{cccc}
9 & 8 & 7 & 6 \\
5 & 4 & 3 & 2 \\
\end{array}\]
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 Karpinski
## 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](https://www.jamesreinders.com)
AVX-512

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

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

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

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

The compiler can identify when instructions inside of a loop can be rewritten as a vectorized operation.

Works for simple loops only and is rare in database operators. Requires hardware support for SIMD instructions.
This loop is not legal to automatically vectorize because the code is written such that the addition is described sequentially.

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

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/C++ tells the compiler that the arrays are distinct memory locations for the lifetime of the pointers. The compiler can then infer that it is safe to vectorize operations on those pointers.
The `restrict` keyword in C/C++ tells the compiler that the arrays are distinct memory locations for the lifetime of the pointers. The compiler can then infer that it is safe to vectorize operations on those pointers.

```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];
    }
}
```
The `restrict` keyword in C/C++ tells the compiler that the arrays are distinct memory locations for the lifetime of the pointers. The compiler can then infer that it is safe to vectorize operations on those pointers.

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

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

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

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

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

void add(int *X, int *Y, int *Z) {
    __mm128i *vecX = (__m128i*)X;
    __mm128i *vecY = (__m128i*)Y;
    __mm128i *vecZ = (__m128i*)Z;
    for (int i=0; i<MAX/4; i++) {
        _mm_store_si128(vecZ++,
                        _mm_add_epi32(*vecX++,
                                      *vecY++));
    }
}

Store the vectors in 128-bit SIMD registers.

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

Evaluate how well the compiler can automatically vectorize the Vectorwise primitives.
→ Targets: GCC v7.2, Clang v5.0, ICC v18

ICC was able to vectorize the most primitives using AVX-512:
→ Vectorized: Hashing, Selection, Projection
→ Not Vectorized: Hash Table Probing, Aggregation
AUTOMATIC VECTORIZATION

Intel Core i9-7900X (10 cores × 2HT)
Compiler: ICC v18

Source: Timo Kersten

<table>
<thead>
<tr>
<th>Q1</th>
<th>Auto</th>
<th>Manual</th>
<th>Auto+Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>29.0</td>
<td>12.0</td>
<td>15.4</td>
<td>15.4</td>
</tr>
<tr>
<td>35.0</td>
<td>15.4</td>
<td>15.4</td>
<td>27.2</td>
</tr>
<tr>
<td>46.6</td>
<td>31.5</td>
<td>42.0</td>
<td>62.5</td>
</tr>
<tr>
<td>82.6</td>
<td>82.9</td>
<td>60.1</td>
<td>61.2</td>
</tr>
</tbody>
</table>

Source: Timo Kersten
AUTOMATIC VECTORIZATION

Intel Core i9-7900X (10 cores × 2HT)
Compiler: ICC v18

Source: Timo Kersten

![Chart showing reduction of time with different vectorization methods: Auto, Manual, and Auto+Manual. The chart compares Better Performance and Worse Performance across Q1, Q6, Q3, Q9, and Q18 with percentage reductions in time.](chart.png)
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
Almost all AVX-512 operations support **predication** variants whereby the CPU only performs operations on lanes specified by an input bitmask.
Almost all AVX-512 operations support **predication** variants whereby the CPU only performs operations on lanes specified by an input bitmask.
Almost all AVX-512 operations support *predication* variants whereby the CPU only performs operations on lanes specified by an input bitmask.
Almost all AVX-512 operations support **predication** variants whereby the CPU only performs operations on lanes specified by an input bitmask.

![Diagram showing SIMD masking](attachment:image.png)
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.

**Permute**

- **Value Vector**
  - D
- **Index Vector**
  - 3 0 3 1
- **Input Vector**
  - A B C D
  - 0 1 2 3
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, B, C, D
- **Mask**: 0, 1, 0, 1
- **Memory**: U, V, W, X, Y, Z, ...
Selective Load

Vector: A B C D

Mask: 0 1 0 1

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

Selective Load

Vector: A U C D

Mask: 0 1 0 1

Memory: U V W X Y Z • • •
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

The diagram illustrates how a selective load operation works, where only the elements specified by the mask (1 for included, 0 for excluded) are transferred from the vector to the memory.
**SELECTIVE LOAD/STORE**

**Selective Load**

- **Vector**: A U C V
- **Memory**: U V W X Y Z
- **Mask**: 0 1 0 1

**Selective Store**

- **Memory**: U V W X Y Z
- **Vector**: A B C D
- **Mask**: 0 1 0 1
**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**: A U C V
- **Mask**: 0 1 0 1
- **Memory**: U V W X Y Z

### Selective Store

- **Memory**: B V W X Y Z
- **Mask**: 0 1 0 1
- **Vector**: A B C D
**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 V W X Y Z

Mask: 0 1 0 1

Vector: A B C D
## 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 / Expand**

**Compress**

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

Value Vector

Index Vector

Input Vector
COMPRESS / EXPAND

Compress

Value Vector

Index Vector

Input Vector
**Compress**

- **Value Vector**: A D 0 0
- **Index Vector**: 1 0 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:  

Index Vector: 0 1 0 1

Input Vector: A B C D
COMPRESS / EXPAND

Compress

Value Vector

Index Vector

Input Vector

Expand

Value Vector

Index Vector

Input Vector
**COMPRESS / EXPAND**

**Compress**

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

**Expand**

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

Value Vector

Index Vector

Memory
SELECTIVE SCATTER/GATHER

**Selective Gather**

Value Vector

Index Vector

Memory

- **W** B A D
- **2 1 5 3**
- **U V W X Y Z**
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/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 V W X Y Z...
**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**

- **Memory**: U V A X Y Z
- **Index Vector**: 2 1 5 3
- **Value Vector**: A B C D
Selective Scatter/Gather

Selective Gather

Value Vector: \( W \rightarrow V \rightarrow Z \rightarrow X \)

Index Vector: \( 2 \rightarrow 1 \rightarrow 5 \rightarrow 3 \)

Memory: \( U \rightarrow V \rightarrow W \rightarrow X \rightarrow Y \rightarrow Z \)

Selective Scatter

Memory: \( U \rightarrow B \rightarrow A \rightarrow D \rightarrow Y \rightarrow C \)

Index Vector: \( 2 \rightarrow 1 \rightarrow 5 \rightarrow 3 \)

Value Vector: \( A \rightarrow B \rightarrow C \rightarrow D \)
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).
VECTORIZED OPERATORS

Selection Scans
Vector Refill
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
```
SELECT * FROM table
WHERE key >= $low AND key <= $high

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|
SELECTION SCANS

Vectorized

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

```
SELECT * FROM table
WHERE key >= "N" AND key <= "U"
```
### SELECTION SCANS

#### Vectorized

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

#### SQL Query

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

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

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

---

**TID** | **KEY**
--- | ---
100 | A
101 | N
102 | D
103 | Y
104 | P
105 | I
106 | S
107 | D

**Key Vector**
A N D Y P I S

**SIMD Compare**

**Mask #1**
0 1 0 1 1 0 1 0

**Mask #2**
1 1 1 0 1 1 1 0

**Mask #3**
0 1 0 0 1 0 1 0
Vectorized

\[
i = 0
\]
for \( v_t \) in table:

\[
simdLoad(v_t.key, v_k)
\]

\[
v_m = (v_k \geq low \ ? 1 : 0) \&
\]

\[
\&(v_k \leq high \ ? 1 : 0)
\]

\[
simdStore(v_t, v_m, output[i])
\]

\[
i = i + |v_m \neq false|
\]

SELECT * FROM table
WHERE key >= "N" AND key <= "U"
Vectorized

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

**SELECT** * FROM table
WHERE key >= "N" AND key <= "U"
SELECTION SCANS

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

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

→ Selective operations using bitmask registers.
SIMD EVALUATION

Intel Core i9-7900X (10 cores × 2HT)
TPC-H Queries (Scalefactor=1)

**Source:** Timo Kersten

---

**Cycles / Element**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Scalar</th>
<th>SIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hashing</td>
<td>5.4</td>
<td>2.4</td>
</tr>
<tr>
<td>Gather</td>
<td>2.4</td>
<td>1.9</td>
</tr>
<tr>
<td>Join</td>
<td>12.3</td>
<td>8.6</td>
</tr>
</tbody>
</table>

**Runtime (ms)**

<table>
<thead>
<tr>
<th>Query</th>
<th>Scalar</th>
<th>SIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q3</td>
<td>41</td>
<td>36</td>
</tr>
<tr>
<td>Q9</td>
<td>106</td>
<td>93</td>
</tr>
</tbody>
</table>

1.4x improvement for Join, 2.3x for Hashing, 1.1x for Gather.
OBSERVATION

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

\[
\begin{align*}
\text{agg} & = \text{dict}() \\
\text{for } t \text{ in table: } & \\
& \text{if } t.\text{age} > 20: \\
& \quad \text{agg}[t.\text{city}][\text{'count'}]++ \\
\text{for } t \text{ in agg: } & \\
& \text{emit}(t)
\end{align*}
\]

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

Pipeline #1

OBSERVATION

```
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**

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

**Pipeline #2**

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

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

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

Vectorization Candidate

Stage #1

Stage Buffer

Stage #2
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)
```

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

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

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

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

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

Source: Prashanth Menon
ROF EVALUATION – TPC-H Q19

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

<table>
<thead>
<tr>
<th>Method</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 + Pretching</td>
<td>189</td>
</tr>
</tbody>
</table>

Source: Prashanth Menon
VECTOR REFILL ALGORITHMS

Approach #1: Buffered
→ Use additional SIMD registers to stage results within an operator and proceed with next loop iteration to fill in underutilized lanes vectors.

Approach #2: Partial
→ Use additional SIMD registers to buffer results from underutilized vectors and then return to previous operator to process the next vector.
→ Requires fine-grained bookkeeping to make sure other operators do not clobber deferred vectors.
VECTORIZED OPERATORS

Selection Scans
Vector Refill
Hash Tables
Partitioning / Histograms
Scalar

**Input Key**  \( \text{hash(key)} \)  **Hash Index**

\[
\text{k1} \rightarrow \# \rightarrow \text{h1}
\]

**Linear Probing Hash Table**

\[
\text{k1} = \text{k9}
\]
**HASH TABLES – PROBING**

### Scalar

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

- \( k_1 \)
- \( \text{hash(key)} \)
- \( h_1 \)

**Linear Probing Hash Table**

- \( k_1 = k_1 \)
- \( k_9 \)
- \( k_3 \)
- \( k_8 \)
HASH TABLES – PROBING

Scalar

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

Vectorized (Horizontal)

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

Linear Probing
Bucketized Hash Table

Four Keys
Four Values

k1 = k9 k3 k8 k1
**HASH TABLES – PROBING**

### Scalar

<table>
<thead>
<tr>
<th>Input Key</th>
<th>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>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

**KEYS**

- $k_1$
- $k_9$
- $k_3$
- $k_8$
- $k_1$

**PAYLOAD**

- Four Keys
- Four Values

**SIMD Compare**

**Matched Mask**

0 0 0 1

**Matched Key**

$k_1 = k_9, k_3, k_8, k_1$
### Vectorized (Vertical)

**Input Key Vector**
- k1
- k2
- k3
- k4

**Hash Index Vector**
- h1
- h2
- h3
- 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>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>

**Vectorized (Vertical)**

- Input Key Vector
  - k1
  - k2
  - k3
  - k4

- Hash Index Vector
  - h1
  - h2
  - h3
  - 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>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>

**Linear Probing**

- hash(key)
- Input Key Vector
- Hash Index Vector
- Linear Probing Hash Table
**Vectorized (Vertical)**

<table>
<thead>
<tr>
<th>Input Key Vector</th>
<th>hash(key)</th>
<th>Hash Index Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
<td>#</td>
<td>h1</td>
</tr>
<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>

**Linear Probing Hash Table**

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

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

Input Key Vector  Hash Index 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>

### 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>k88</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>
Vectorized (Vertical)

<table>
<thead>
<tr>
<th>Input Key Vector</th>
<th>hash(key)</th>
<th>Hash Index Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
<td>#</td>
<td>h1</td>
</tr>
<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>

<table>
<thead>
<tr>
<th>SIMD Compare</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1 = k1</td>
</tr>
<tr>
<td>k2 = k99</td>
</tr>
<tr>
<td>k3 = k88</td>
</tr>
<tr>
<td>k4 = k4</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>
**Vectorized (Vertical)**

**Input Key Vector**
- k5
- k2
- k3
- k6

**Hash Table**
- \( hash(key) \)

**Hash Index Vector**
- h5
- h2+1
- h3+1
- h6

**SIMD Compare**

**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>1</td>
</tr>
<tr>
<td>k99</td>
<td>0</td>
</tr>
<tr>
<td>k88</td>
<td>0</td>
</tr>
<tr>
<td>k4</td>
<td>1</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

Vectorized (Vertical)

<table>
<thead>
<tr>
<th>Input Key Vector</th>
<th>hash(key)</th>
<th>Hash Index Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>k5</td>
<td>#</td>
<td>h5</td>
</tr>
<tr>
<td>k2</td>
<td>#</td>
<td>h2+1</td>
</tr>
<tr>
<td>k3</td>
<td>#</td>
<td>h3+1</td>
</tr>
<tr>
<td>k6</td>
<td>#</td>
<td>h6</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>
Hash Tables – Probing

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

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

Throughput (billion tuples / sec)

Hash Table Size

Source: Orestis Polychroniou
Throughput (billion tuples/sec) vs Hash Table Size

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

- **Scalar**
- **Vectorized (Horizontal)**
- **Vectorized (Vertical)**

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

- **Out of Cache**

Source: Orestis Polychroniou
PARTITIONING – HISTOGRAM

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

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

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

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

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

The two strategies are not mutually exclusive. Within a single application, both strategies can be applied to individual operators as well as to batches of operators (nested strategy). Moreover, the query compiler might decide to apply any of the two strategies to certain operators. Typically, when a sequence of operators is quite cheap, expensive neighbors are acceptable as long as the costs for self-operations are not dominated. Naturally, this is a physical query optimization problem that we will leave for future work. Nevertheless, we briefly discuss the advantages and disadvantages, as this is the first work in which we present the basic principles of vector-processing in compiled query pipelines.

As mentioned above, **container rewriting** requires additional registers, which increases the memory footprint and may lead to spilling; **variable bindings** allocates additional registers as well, but these are reused for (variable) mask registers. Therefore, it is unlikely to be affected by (potential) performance degradation due to spilling.

The second major difference lies in the cost of retyping empty lists. In a pipeline that follows the partial consnsate strategy, the very first operator that is the pipeline source is responsible for writing empty lists. If other operators experience amodification, they reparse the original flow to the previous operator while retaining ownership of the earlier lists. This overhead may rise when the source operator is reached, as shown in Fig. 5c. All operators between the pipeline source and the operator that returned the control flow may be impacted. As mentioned, between several lists and the number of lists that are parallelized. The cost of saving, therefore, depends on the length of the pipeline, and the cost of the preceding operators. In general, the cost increase due to the use of local storage. Nevertheless, parallelism can improve query performance if it is applied only in the very first operators. By contrast, the relative cost of spilling operators does not depend on the pipeline length, although the actual cost depends on the number of required buffer lists. The greater the number of surviving, the greater the number of memory interactions that need to be executed, whereas the number of required buffer lists depends on the number of attributes passed along the pipeline and it is not so much the number of entries retained to save the internal state of the operator (e.g., a pointer to the element type).
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.

AVX-512 is not always faster.
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 clockspeed when switching to AVX-512 mode.

Compilers will prefer 256-bit SIMD operations.

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

Vectorization is essential for OLAP queries. But implementing an algorithm using SIMD is still mostly a manual process.

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

Query Compilation
Project Status Discussion