TODAY’S AGENDA

Background
SIMD
Parallel Sort-Merge Join
Evaluation
Hate Mail
SORT-MERGE JOIN ($R \bowtie S$)

**Phase #1: Sort**
→ Sort the tuples of $R$ and $S$ based on the join key.

**Phase #2: Merge**
→ Scan the sorted relations and compare tuples.
→ The outer relation $R$ only needs to be scanned once.
SORT-MERGE JOIN (R ⋈ S)

Relation R

Relation S
SORT-MERGE JOIN (R⨝S)

Relation R

Relation S
SORT-MERGE JOIN (R $\bowtie$ S)
SORT-MERGE JOIN (R \Join S)
SORT-MERGE JOIN (R \(\bowtie\) S)
SORTING VS. HASHING

1970s – Sorting
1980s – Hashing
1990s – Both
2000s – Hashing
2010s – ???
IN-MEMORY JOINS

→ Hashing is faster than Sort-Merge.
→ Sort-Merge will be faster with wider SIMD.

→ Sort-Merge is already faster, even without SIMD.
→ New optimizations and results for Radix Hash Join.

Source: Cagri Balkesen
**SINGLE INSTRUCTION, MULTIPLE DATA**

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

Both current AMD and Intel CPUs have ISA and microarchitecture support SIMD operations.

→ MMX, 3DNow!, SSE, SSE2, SSE3, SSE4, AVX
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 the four elements simultaneously.

First introduced by Intel in 1999.
SIMD EXAMPLE

\[ X + Y = Z \]

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

\[ X + Y = Z \]

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

**Code Example:**

```plaintext
def add_vectors(x, y, n):
    z = [0] * n
    for i in range(n):
        z[i] = x[i] + y[i]
    return z
```

**Example Usage:**

```plaintext
x = [1, 2, 3, 4]
y = [5, 6, 7, 8]
result = add_vectors(x, y, 4)
print(result)  # Output: [6, 8, 10, 12]
```
SIMD EXAMPLE

\( X + Y = Z \)

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

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

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

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

\[ X + Y = Z \]

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

\textbf{SISD}

\[
\begin{array}{cccc}
    \text{for} & (i=0; & i<n; & i++) \{ \\
    & Z[i] = X[i] + Y[i]; & \}
\end{array}
\]

\begin{array}{cccc}
    1 & 1 & 1 & 1 \\
    1 & 1 & 1 & 1 \\
    1 & 1 & 1 & 1 \\
\end{array}

\begin{array}{cccc}
    8 & 7 & 6 & 5 \\
    4 & 3 & 2 & 1 \\
\end{array}

\begin{array}{cccc}
    \text{SISD} & + & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 \\
\end{array}

**SIMD EXAMPLE**

\[ X + Y = Z \]

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

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

\[ X + Y = Z \]

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

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

\[
X + Y = Z
\]

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

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

\[ X + Y = Z \]

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

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

\[ X + Y = Z \]

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

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

128-bit SSE Register

8 7 6 5

128-bit SSE Register

1 1 1 1

1 1 1 1

1 1 1 1
SIMD EXAMPLE

\[ X + Y = Z \]

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

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

128-bit SSE Register

128-bit SSE Register

128-bit SSE Register
SIMD EXAMPLE

\[ X + Y = Z \]

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

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

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

for \((i=0; \ i<n; \ i++)\) {
    \(Z[i] = X[i] + Y[i];\)
}
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.
WHY NOT GPUS?

Moving data back and forth between DRAM and GPU is slow over PCI-E bus.

Emerging co-processors that can share CPU’s memory may change this.
→ Examples: AMD’s APU, Intel’s Knights Landing
PARALLEL SORT-MERGE JOINS

Sorting is always the most expensive part.

Take advantage of new hardware to speed things up as much as possible.

→ Utilize as many CPU cores as possible.
→ Be mindful of NUMA boundaries.
PARALLEL SORT-MERGE JOIN (R \times S)

**Phase #1: Partitioning (optional)**
→ Partition \( S \) and assign them to workers / cores.

**Phase #2: Sort**
→ Sort the tuples of \( R \) and \( S \) based on the join key.

**Phase #3: Merge**
→ Scan the sorted relations and compare tuples.
→ The outer relation \( R \) only needs to be scanned once.
PARTITIONING PHASE

Divide the relations into chunks and assign them to cores.
→ Explicit vs. Implicit

Explicit: Divide only the outer relation and redistribute among the different CPU cores.
→ Can use the same radix partitioning approach we talked about last time.
SORT PHASE

Create **runs** of sorted chunks of tuples for both input relations.

It used to be that Quicksort was good enough. But NUMA and parallel architectures require us to be more careful...
CACHE-CONSCIOUS SORTING

Level #1: In-Register Sorting
→ Sort runs that fit into CPU registers.

Level #2: In-Cache Sorting
→ Merge the output of Level #1 into runs that fit into CPU caches.
→ Repeat until sorted runs are ½ cache size.

Level #3: Out-of-Cache Sorting
→ Used when the runs of Level #2 exceed the size of caches.
CACHE-CONSCIOUS SORTING

UNSORTED
CACHE-CONSCIOUS SORTING

Level #1

...
CACHE-CONSCIOUS SORTING

Level #1

UNSORTED
CACHE-CONSCIOUS SORTING

Level #1

Level #2
CACHE-CONSCIOUS SORTING

Level #1

Level #2

UNSORTED
CACHE-CONSCIOUS SORTING

UNSORTED

Level #1

Level #2

Level #3
CACHE-CONSCIOUS SORTING

Level #1

Level #2

Level #3

UNSORTED

SORTED
LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.
→ Always has fixed wiring “paths” for lists with the same number of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.
Abstract model for sorting keys.
→ Always has fixed wiring “paths” for lists with the same number of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.
LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.
→ Always has fixed wiring “paths” for lists with the same number of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.
LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.
→ Always has fixed wiring “paths” for lists with the same number of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.
Abstract model for sorting keys.
→ Always has fixed wiring “paths” for lists with the same number of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.
LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.
→ Always has fixed wiring “paths” for lists with the same number of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.
LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.
→ Always has fixed wiring “paths” for lists with the same number of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.
LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.
→ Always has fixed wiring “paths” for lists with the same number of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.
LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.
→ Always has fixed wiring “paths” for lists with the same number of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.

**Input**

<table>
<thead>
<tr>
<th>9</th>
<th>5</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

**Output**

| 3 | 9 |
LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.
→ Always has fixed wiring “paths” for lists with the same number of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.

Input  Output

9  5  3  9  6  5
5  9  3  5  6
3  6  5  6
6  9

3  5  6  9
LEVEL #1 – SORTING NETWORKS

12 21 4 13
9 8 6 7
1 14 3 0
5 11 15 10
LEVEL #1 – SORTING NETWORKS

Instructions:
→ 4 LOAD
Sort Across Registers

Instructions:
→ 4 LOAD
Sort Across Registers

Instructions:
→ 4 LOAD
LEVEL #1 – SORTING NETWORKS

Sort Across Registers

Instructions:
→ 4 LOAD
# Level #1 - Sorting Networks

**Sort Across Registers**

<table>
<thead>
<tr>
<th>12</th>
<th>21</th>
<th>4</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>8</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>14</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>15</td>
<td>10</td>
</tr>
</tbody>
</table>

Instructions:

→ **4 LOAD**

<table>
<thead>
<tr>
<th>1</th>
<th>8</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>11</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>14</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>21</td>
<td>15</td>
<td>13</td>
</tr>
</tbody>
</table>

Instructions:

→ **10 MIN/MAX**
LEVEL #1 – SORTING NETWORKS

Sort Across Registers

Instructions:
→ 4 LOAD

Transpose Registers

Instructions:
→ 10 MIN/MAX
LEVEL #1 – SORTING NETWORKS

Sort Across Registers

Instructions: → 4 LOAD

Transpose Registers

Instructions: → 10 MIN/MAX
LEVEL #1 – SORTING NETWORKS

Sort Across Registers

Instructions:
→ 4 LOAD

Transpose Registers

Instructions:
→ 10 MIN/MAX
LEVEL #1 – SORTING NETWORKS

Sort Across Registers

Instructions: → 4 LOAD

Transpose Registers

Instructions: → 10 MIN/MAX

Instructions: → 8 SHUFFLE
→ 4 STORE
LEVEL #2 – BITONIC MERGE NETWORK

Like a Sorting Network but it can merge two locally-sorted lists into a globally-sorted list.

Can expand network to merge progressively larger lists ($\frac{1}{2}$ cache size).

Intel’s Measurements
$\rightarrow$ 2.25–3.5x speed-up over SISD implementation.
LEVEL #2 – BITONIC MERGE NETWORK

Input

Output

a_1

a_2

a_3

a_4

b_4

b_3

b_2

b_1
LEVEL #2 – BITONIC MERGE NETWORK

Input

Sorted Run

Output
LEVEL #2 - BITONIC MERGE NETWORK

Sorted Run

Reverse Sorted Run

Input

Output

min/max

min/max

min/max
LEVEL #2 – BITONIC MERGE NETWORK

- Input
- Output

Sorted Run

Reverse Sorted Run

min/max min/max min/max
LEVEL #3 – MULTI-WAY MERGING

Use the Bitonic Merge Networks but split the process up into tasks.
→ Still one worker thread per core.
→ Link together tasks with a cache-sized FIFO queue.

A task blocks when either its input queue is empty or its output queue is full.

Requires more CPU instructions, but brings bandwidth and compute into balance.
LEVEL #3 – MULTI-WAY MERGING

Sorted Runs

Cache-Sized Queue

MERGE

MERGE

MERGE

MERGE

MERGE

MERGE
LEVEL #3 – MULTI-WAY MERGING

Sorted Runs

Cache-Sized Queue

MERGE

MERGE

MERGE

MERGE

MERGE

MERGE

MERGE

MERGE

MERGE

MERGE

MERGE
LEVEL #3 – MULTI-WAY MERGING

Sorted Runs

Cache-Sized Queue

MERGE

MERGE

MERGE

MERGE

MERGE
MERGE PHASE

Iterate through the outer table and inner table in lockstep and compare join keys. May need to backtrack if there are duplicates.

Can be done in parallel at the different cores without synchronization if there are separate output buffers.
SORT-MERGE JOIN VARIANTS

- Multi-Way Sort-Merge (M-WAY)
- Multi-Pass Sort-Merge (M-PASS)
- Massively Parallel Sort-Merge (MPSM)
MULTI-WAY SORT-MERGE

Outer Table
→ Each core sorts in parallel on local data (levels #1/#2).
→ Redistribute sorted runs across cores using the multi-way merge (level #3).

Inner Table
→ Same as outer table.

Merge phase is between matching pairs of chunks of outer/inner tables at each core.
MULTI-WAY SORT-MERGE
Local-NUMA Partitioning
MULTI-WAY SORT-MERGE

Local-NUMA Partitioning
MULTI-WAY SORT-MERGE

Local-NUMA Partitioning  Sort
MULTI-WAY SORT-MERGE

Local-NUMA Partitioning -> Sort -> Multi-Way Merge
MULTI-WAY SORT-MERGE

Local-NUMA Partitioning  Sort  Multi-Way Merge
MULTI-WAY SORT-MERGE

Local-NUMA Partitioning  Sort  Multi-Way Merge
MULTI-WAY SORT-MERGE

Local-NUMA Partitioning  Sort  Multi-Way Merge
MULTI-WAY SORT-MERGE

Local-NUMA Partitioning  Sort  Multi-Way Merge  Same steps as Outer Table
MULTI-WAY SORT-MERGE

Local-NUMA Partitioning  Sort  Multi-Way Merge  Same steps as Outer Table

- Sort!
- Sort!
- Sort!
- Sort!

Local-NUMA
Partitioning

Multi-Way Merge

Same steps as Outer Table
MULTI-WAY SORT-MERGE

Local-NUMA Partitioning Sort

Multi-Way Merge

Local Merge Join

Same steps as Outer Table

Local Merge Join

Sort!
MULTI-WAY SORT-MERGE

Local-NUMA Partitioning | Sort | Multi-Way Merge | Local Merge Join | Same steps as Outer Table

Local-NUMA Partitioning | Sort | Multi-Way Merge | Local Merge Join | Same steps as Outer Table

Local-NUMA Partitioning | Sort | Multi-Way Merge | Local Merge Join | Same steps as Outer Table

Local-NUMA Partitioning | Sort | Multi-Way Merge | Local Merge Join | Same steps as Outer Table
MULTI-WAY SORT-MERGE

Local-NUMA Partitioning Sort

Multi-Way Merge

Local Merge Join

Same steps as Outer Table

SORT!
MULTI-PASS SORT-MERGE

Outer Table
→ Same level #1/#2 sorting as M-WAY.
→ But instead of redistributing, it uses a multi-pass naïve merge on sorted runs.

Inner Table
→ Same as outer table.

Merge phase is between matching pairs of chunks of outer table and inner table.
MASSIVELY PARALLEL SORT-MERGE

**Outer Table**
- Range-partition outer table and redistribute to cores.
- Each core sorts in parallel on their partitions.

**Inner Table**
- Not redistributed like outer table.
- Each core sorts its local data.

Merge phase is between entire sorted run of outer table and a segment of inner table.
MASSIVELY PARALLEL SORT-MERGE
MASSIVELY PARALLEL SORT-MERGE

Cross-NUMA Partitioning
MASSIVELY PARALLEL SORT-MERGE

Cross-NUMA Partitioning
MASSIVELY PARALLEL SORT-MERGE

Cross-NUMA Partitioning

Sort

CMU 15-721 (Spring 2016)
MASSIVELY PARALLEL SORT-MERGE

Cross-NUMA Partitioning

Sort

[Diagram showing the process of Cross-NUMA Partitioning followed by sorting and merging]
MASSIVELY PARALLEL SORT-MERGE

Cross-NUMA Partitioning

Sort

Cross-Partition Merge Join

Cross-NUMA Partitioning

Sort

Cross-Partition Merge Join

Sort!
MASSIVELY PARALLEL SORT-MERGE

Cross-NUMA Partitioning

Sort

Cross-Partition Merge Join
MASSIVELY PARALLEL SORT-MERGE

Cross-NUMA Partitioning

Sort

Cross-Partition Merge Join
MASSIVELY PARALLEL SORT-MERGE

Cross-NUMA Partitioning

Sort

Cross-Partition Merge Join
MASSIVELY PARALLEL SORT-MERGE

Cross-NUMA Partitioning

Sort

Cross-Partition Merge Join
MASSIVELY PARALLEL SORT-MERGE

Cross-NUMA Partitioning  Sort  Cross-Partition Merge Join
HYPER’s RULES FOR PARALLELIZATION

Rule #1: No random writes to non-local memory
→ Chunk the data, redistribute, and then each core sorts/works on local data.

Rule #2: Only perform sequential reads on non-local memory
→ This allows the hardware prefetcher to hide remote access latency.

Rule #3: No core should ever wait for another
→ Avoid fine-grained latching or sync barriers.

Source: Martina-Cezara Albutiu
EVALUATION

Compare the different join algorithms using a synthetic data set.

→ **Sort-Merge**: M-WAY, M-PASS, MPSM
→ **Hash**: Radix Partitioning

**Hardware:**
→ 4 Socket Intel Xeon E4640 @ 2.4GHz
→ 8 Cores with 2 Threads Per Core
→ 512 GB of DRAM
RAW SORTING PERFORMANCE

Single-threaded sorting performance

Throughput (M Tuples/sec) vs. Number of Tuples (in $2^{20}$)

- C++ STL Sort
- SIMD Sort

Source: Cagri Balkesen
RAW SORTING PERFORMANCE

Single-threaded sorting performance

Throughput (M Tuples/sec) vs. Number of Tuples (in $2^{20}$)

- **C++ STL Sort**
- **SIMD Sort**

**Source:** Cagri Balkesen

**2.5–3x Faster**
COMPARISON OF SORT-MERGE JOINS

Workload: 1.6B \Join 128M (8-byte tuples)

<table>
<thead>
<tr>
<th>Method</th>
<th>Cycles / Output Tuple</th>
<th>Throughput (M Tuples/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>M-WAY</td>
<td>7.6</td>
<td>13.6</td>
</tr>
<tr>
<td>M-PASS</td>
<td>22.9</td>
<td></td>
</tr>
</tbody>
</table>

Source: Cagri Balkesen
COMPARISON OF SORT-MERGE JOINS

Workload: 1.6B ÷ 128M (8-byte tuples)

Source: Cagri Balkesen

Throughput (M Tuples/sec)

Cycles / Output Tuple

Partition  Sort  S-Merge  M-Join  Throughput

M-WAY  7.6
M-PASS  13.6
MPSM  22.9

Source: Carnegie Mellon University
COMPARISON OF SORT-MERGE JOINS

Workload: 1.6B ⋈ 128M (8-byte tuples)

<table>
<thead>
<tr>
<th>Method</th>
<th>Cycles / Output Tuple</th>
<th>Throughput (M Tuples/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>M-WAY</td>
<td>7.6</td>
<td></td>
</tr>
<tr>
<td>M-PASS</td>
<td>13.6</td>
<td>22.9</td>
</tr>
<tr>
<td>MPSM</td>
<td>22.9</td>
<td></td>
</tr>
</tbody>
</table>

Source: Cagri Balkesen
COMPARISON OF SORT-MERGE JOINS

Workload: 1.6B ⋈ 128M (8-byte tuples)

Throughput: (M Tuples/sec)

<table>
<thead>
<tr>
<th>Method</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>M-WAY</td>
<td>7.6</td>
</tr>
<tr>
<td>M-PASS</td>
<td>13.6</td>
</tr>
<tr>
<td>MPSM</td>
<td>22.9</td>
</tr>
</tbody>
</table>

Cycles / Output Tuple

- Partition
- Sort
- S-Merge
- M-Join

Source: Cagri Balkesen
M-WAY JOIN VS. MPSM JOIN

Workload: 1.6B \bowtie 128M (8-byte tuples)

- Multi-Way
- Massively Parallel

Source: Cagri Balkesen

CMU 15-721 (Spring 2016)
**M-WAY JOIN VS. MPSM JOIN**

*Workload: 1.6B $\bowtie$ 128M (8-byte tuples)*

- **Multi-Way**
- **Massively Parallel**

Source: Cagri Balkesen

**Number of Threads**

- 1
- 2
- 4
- 8
- 16
- 32
- 64

**Throughput (M Tuples/sec)**

- Multi-Way: 105 M/sec
- Massively Parallel: 315 M/sec
### SORT-MERGE JOIN VS. HASH JOIN

**4 Socket Intel Xeon E4640 @ 2.4GHz**

**8 Cores with 2 Threads Per Core**

| Source: Cagri Balkesen |

<table>
<thead>
<tr>
<th>Cycles / Output Tuple</th>
<th>Partition</th>
<th>Sort</th>
<th>S-Merge</th>
<th>M-Join</th>
<th>Build+Probe</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SORT</strong> 128M⨝128M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>HASH</strong> 1.6B⨝1.6B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SORT</strong> 128M⨝512M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>HASH</strong> 1.6B⨝6.4B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: [Cagri Balkesen](#)
SORT-MERGE JOIN VS. HASH JOIN

4 Socket Intel Xeon E4640 @ 2.4GHz
8 Cores with 2 Threads Per Core

Source: Cagri Balkesen

Cycles / Output Tuple

<table>
<thead>
<tr>
<th>Partition</th>
<th>Sort</th>
<th>S-Merge</th>
<th>M-Join</th>
<th>Build+Probe</th>
</tr>
</thead>
<tbody>
<tr>
<td>128M</td>
<td>1.6B</td>
<td>128M</td>
<td>1.6B</td>
<td>512M</td>
</tr>
<tr>
<td>1.6B</td>
<td>128M</td>
<td>512M</td>
<td>6.4B</td>
<td>512M</td>
</tr>
<tr>
<td>128M</td>
<td>1.6B</td>
<td>512M</td>
<td>6.4B</td>
<td>512M</td>
</tr>
</tbody>
</table>

Source: Cagri Balkesen
SORT-MERGE JOIN VS. HASH JOIN

Varying the size of the input relations

- Multi-Way Sort-Merge Join
- Radix Hash Join

Source: Cagri Balkesen
PARTING THOUGHTS

Both join approaches are equally important. Every serious OLAP DBMS supports both.

We did not consider the impact of queries where the output needs to be sorted.
HATE MAIL
You database skills are weak. You look like a freak that hasn’t showered for weeks in these videos. I feel sorry for the students that are sitting in the front row.

If I was there I would mug you in the hallway and steal your wallet.

Go fuck yourself.
You database skills are terrible. There are these videos. I found a bug that you missed.
If I was there I would have found it.
Go fuck yourself.

I hate you.
I hate your course.
I hate your fake ass lessons for the street.

If I see you out in the cut I am probably going to stab you to make you bleed.
I hope you burn in hell.
You database skills suck. I hate these videos. I feel like I'm watching a dog shit puppy.

If I was there I would have kicked your ass.

Go fuck yourself.

I hate you. I hate your course. I hate your fake courses. If I see you out I hope you burn in hell.

What is wrong with your face? Why do you look like a pile of hot brown dog sludge?

I hate your database course.
Your Database Course Sucks Ass

From: [Redacted]
To: [Redacted]
Date: 02/07/16 05:44

You database skills are terrible. I can't believe you were able to do these videos. I feel bad for the students.

If I was there I would have paid you to do them.

Go fuck yourself.

I hate you.
I hate your course.
I hate your fake degree.

If I see you out I hope you burn yourself.

CMU Database Course

From: [Redacted]
To: [Redacted]
Date: Wednesday 05:37

Phony

From: [Redacted]
To: "Andy Pavlo" <andy.pavlo@gmail.com>
Date: Today 11:01:23 AM

You are never going to be Stonebreaker. You are never going to be DeWitt. You are never going to be Gray. You should just stop now, refund the students money, and get a job at Taco Bell. That's where you belong.