TODAY’S AGENDA

Background
SIMD
Parallel Sort-Merge Join
Evaluation

*SPOILER: This doesn’t work on current Xeon CPUs.*
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
**SIMD EXAMPLE**

\[
X + Y = Z
\]

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

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

\[ SISD \]

\[ + \]

\[ Z \]

\[ \begin{bmatrix}
  1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{bmatrix} \]

\[ \begin{bmatrix}
  8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 9 \\
\end{bmatrix} \]
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 \((i=0; i<n; i++)\) {
    \(Z[i] = X[i] + Y[i];\)
}

\(X\)

\(Y\)

\(Z\)

SISD

+
### SIMD Example

**Equation:**

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

**Code:**

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

**Diagram:**

- **X:** 128-bit SIMD Register
  - 8 7 6 5
  - 4 3 2 1
- **Y:** 128-bit SIMD Register
  - 1111
- **Z:** 128-bit SIMD Register
  - 1111

**Matrix Addition:**

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

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

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.

There are some newer GPU-enabled DBMSs
→ Examples: MapD, SQream, Kinetica

Emerging co-processors that can share CPU’s memory may change this.
→ Examples: AMD’s APU, Intel’s Knights Landing
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 $\bowtie$ S)
SORT-MERGE JOIN (R⋈S)

Relation R

Relation S

SORT!
SORT-MERGE JOIN (R⨝S)

Relation R

MERGE!

Relation S
SORT-MERGE JOIN ($R \bowtie S$)

Relation $R$

Relation $S$

MERGE!
SORT-MERGE JOIN (R⋈S)

Relation R

MERGE!

Relation S
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⨝S)

Phase #1: Partitioning (optional)
→ Partition R 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

Level #1

Level #2

Level #3

SORTED
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.
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.
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**

Instructions:
→ 4 LOAD
Sort Across Registers

Instructions:
→ 4 LOAD

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

Sort Across Registers

12 21 4 13
9 8 6 7
1 14 3 0
5 11 15 10

1 8 3 0
5 11 4 7
9 14 6 10
12 21 15 13

Transpose Registers

1 5 9 12
8 11 14 21
3 4 6 15
0 7 10 13

Instructions:
→ 4 LOAD

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

Sort Across Registers

Instructions:
→ 4 LOAD

Instructions:
→ 10 MIN/MAX

Transpose Registers

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 (½ cache size).

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

Input

Sorted Run

Reverse

Sorted Run

Output

\[ a_1 \]
\[ a_2 \]
\[ a_3 \]
\[ a_4 \]
\[ b_4 \]
\[ b_3 \]
\[ b_2 \]
\[ b_1 \]

SHUFFLE

SHUFFLE

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.
Sorted Runs

LEVEL #3 – MULTI-WAY MERGING

Cache-Sized Queue

MERGE

MERGE

MERGE

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  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 | Same steps as Outer Table

[Diagram showing the process of local-NUMA partitioning, sorting, and multi-way merge, along with the 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!

SORT!

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

Cross-NUMA Partitioning

Sort
MASSIVELY PARALLEL SORT-MERGE

Cross-NUMA Partitioning

Sort

SORT!

SORT!

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

Cross-Partitioning Sort

Merge Join

SORT!

SORT!
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*

![Graph showing comparison between C++ STL Sort and SIMD Sort](image)

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

2.5–3x Faster

*Source: Cagri Balkesen*
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 (MTuples/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>13.6</td>
<td>22.9</td>
</tr>
<tr>
<td>MPSM</td>
<td>22.9</td>
<td></td>
</tr>
</tbody>
</table>

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

Workload: 1.6B × 128M (8-byte tuples)

Throughput (M Tuples/sec)

- Multi-Way
- Massively Parallel

Source: Cagri Balkesen

Throughput (M Tuples/sec)

- 315 M/sec
- 105 M/sec

Number of Threads

Source: Cagri Balkesen
CMU 15-721 (Spring 2017)
SORT-MERGE JOIN VS. HASH JOIN

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

<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>128M</td>
<td>1.6B</td>
<td>1.6B</td>
<td>128M</td>
</tr>
<tr>
<td>128M</td>
<td>128M</td>
<td>1.6B</td>
<td>1.6B</td>
<td>128M</td>
</tr>
<tr>
<td>1.6B</td>
<td>1.6B</td>
<td>128M</td>
<td>512M</td>
<td>1.6B</td>
</tr>
<tr>
<td>1.6B</td>
<td>1.6B</td>
<td>128M</td>
<td>512M</td>
<td>1.6B</td>
</tr>
</tbody>
</table>

Source: Cagri Balkesen

CMU 15-721 (Spring 2017)
SORT-MERGE JOIN VS. HASH JOIN

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

Cycles / Output Tuple

| Source: Cagri Balkesen |

<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>128M</td>
<td>1.6B</td>
<td>1.6B</td>
<td>1.6B</td>
</tr>
<tr>
<td>512M</td>
<td>1.6B</td>
<td>1.6B</td>
<td>6.4B</td>
<td>6.4B</td>
</tr>
</tbody>
</table>

CMU 15-721 (Spring 2017)
SORT-MERGE JOIN VS. HASH JOIN

Varying the size of the input relations

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

Source: Cagri Balkesen
WHY DOESN’T ANY OF THIS WORK?

The DBMS has to sort values with their corresponding 64-bit tuple IDs.

Since we have to align our data in SIMD, that means we need to sort 128-bit values.

Intel Xeon (not Phi) only supports AVX-256. That means we can only store two values in a 256-bit SIMD register. Sort Networks need four!
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.
NEXT CLASS

Query Code Generation + Compilation

Reminder: First Code Review
April 11th @ 11:59pm