Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited

Presenter: Haonan Wang

Slides Credit: CMU 15-721 Spring 2018

haonanw@mit.edu

March 19, 2019
Overview

1. Background
   - Sort vs. Hash
   - Motivation

2. Merge - Sort Join
   - The basic idea
   - Sort Phase
   - Merge Phase
   - Multi-Way Merge

3. Experiment
Section 1

Background
Subsection 1

Sort vs. Hash
Sort vs. Hash

There are two main approaches for the PARALLEL JOIN ALGORITHMS:
→ Hash Join
→ Sort-Merge Join

History of Hash VS. Sort

- 1970s Sorting
- 1980s Hashing
- 1990s Equivalent
- 2000s Hashing
- 2010s Hashing (Partitioned vs. Non-Partitioned)
- 2020s ???
What Is Merge-Sort Join

Sort-merge join algorithm explained

Customers table

<table>
<thead>
<tr>
<th>Id</th>
<th>Login</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>User#2</td>
</tr>
<tr>
<td>1</td>
<td>User#1</td>
</tr>
<tr>
<td>4</td>
<td>User#4</td>
</tr>
<tr>
<td>3</td>
<td>User#3</td>
</tr>
</tbody>
</table>

Orders table

<table>
<thead>
<tr>
<th>Id</th>
<th>User id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>2</td>
</tr>
<tr>
<td>1002</td>
<td>4</td>
</tr>
<tr>
<td>1003</td>
<td>4</td>
</tr>
<tr>
<td>1004</td>
<td>1</td>
</tr>
</tbody>
</table>

Sorting

Merging

1. Takes the first row in the left - (1, User#1). Does it have a match in the right? Yes
What is SIMD?

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

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

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

→ MMX, 3DNow!, SSE, SSE2, SSE3, SSE4, AVX
SIMD Makes Sorting Better Than Hashing?

→ Hashing is faster than Sort-Merge.
→ Sort-Merge is faster w/ wider SIMD.

→ Sort-Merge is already faster than Hashing, even without SIMD.

→ New optimizations and results for Radix Hash Join.

→ Trade-offs between partitioning & non-partitioning Hash-Join.

→ Ignore what we said last year.
→ You really want to use Hashing!

→ Hold up everyone! Let's look at everything for real!
Section 2

Merge - Sort Join
The basic idea for the designing

- **Partition Phase (Optional)**
  → Partition R and assign them to workers / cores.

- **Sort Phase**
  → Sort the tuples of R and S based on the join key.

- **Merge Phase**
  → Scan the sorted relations and compare tuples.
  → The outer relation R only needs to be scanned once.
Subsection 2

Sort Phase
Sorting Networks (1)

Input

9 → Output

5 →

3 →

6 →

Presenter: Haonan Wang (MIT)

Sort vs Hash

March 19, 2019
Sorting Networks (2)

**Input**

- 9
- 5
- 3
- 6

**Output**

- 5
- 9
- 3
- 6
Sorting Networks (3)

Input

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

Output
Input

9 → 5 → 3 → Output

5 → 9 → 3

3 → 5

6 → 6
\[
e = \min(a, b) \\
f = \max(a, b) \\
g = \min(c, d) \\
h = \max(c, d) \\
i = \max(e, g) \\
j = \min(f, h) \\
w = \min(e, g) \\
x = \min(i, j) \\
y = \max(i, j) \\
z = \max(f, h)
\]
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.
Instructions:

→ 4 LOAD
Instructions:
→ 10 MIN/MAX
Instructions:
→ 8 SHUFFLE
→ 4 STORE
Subsection 3

Merge Phase
Bitonic Merge Networks

Presenter: Haonan Wang (MIT)
Algorithm 1: Merging larger lists with help of bitonic merge kernel bitonic_merge4() (k = 4).

1. \( a \leftarrow \text{fetch4}(in_1); b \leftarrow \text{fetch4}(in_2); \)
2. repeat
3. \( \langle a, b \rangle \leftarrow \text{bitonic_merge4}(a, b); \)
4. \( \text{emit } a \text{ to output}; \)
5. \( \text{if head}(in_1) < \text{head}(in_2) \text{ then} \)
6. \( a \leftarrow \text{fetch4}(in_1); \)
7. \( \text{else} \)
8. \( a \leftarrow \text{fetch4}(in_2); \)
9. until \( \text{eof}(in_1) \) or \( \text{eof}(in_2); \)
10. \( \langle a, b \rangle \leftarrow \text{bitonic_merge4}(a, b); \)
11. \( \text{emit4}(a); \text{emit4}(b); \)
12. if \( \text{eof}(in_1) \) then
13. \( \text{emit rest of } in_1 \text{ to output}; \)
14. else
15. \( \text{emit rest of } in_1 \text{ to output}; \)
Figure 3: Multi-way merging.
Merging-Sort Hierarchy (Summary)

- *in-register sorting*, with runs that fit into (SIMD) CPU registers;
- *in-cache sorting*, where runs can still be held in a CPU-local cache;
- *out-of-cache sorting*, once runs exceed cache sizes.
Subsection 4

Multi-Way Merge
Impact Of Numa

- In practice, at least some merging passes will inevitably cross NUMA boundaries.
- Multisocket systems show an increasing asymmetry, where the NUMA interconnect bandwidth stays further and further behind the aggregate memory bandwidth that the individual memory controllers could provide.
Figure 4: m-way: NUMA-aware sort-merge join with multi-way merge merge and SIMD.
Section 3

Experiment
- Intel Sandy Bridge with a 256-bit AVX instruction set.
- Four-socket configuration, with each CPU socket containing 8 physical cores and 16 thread contexts by the help of the hyper-threading.
- Cache sizes are 32 KiB for L1, 256 KiB for L2, and 20 MiB L3 (the latter shared by the 16 threads within the socket). The cache line size of the system is 64 bytes. TLB1 contains 64/32 entries when using 4 KiB/2 MiB pages (respectively) and 512 TLB2 entries (page size 4 KiB). Total memory available is 512 GiB (DDR3 at 1600 MHz).
Figure 13: Scalability of sorting-based joins. Workload A, (11.92 GiB × 11.92 GiB). Throughput metric is output tuples per second, i.e. $|S|/\text{execution time}$. 
Result(1)

![Graph showing performance comparison between partition, sort, merge, mjoin, and build-probe for different join workloads.]

- **cycles per output tuple**
  - **Partition**: 411ms
  - **Sort**: 5062ms
  - **Merge**: 4255ms
  - **Mjoin**: 1018ms
  - **Build-probe**: 15514ms

- **Join workloads in number of tuples**
  - 128M × 128M
  - 1.6B × 1.6B
  - 128M × 512M
  - 1.6B × 6.4B
Figure 18: Sort vs. hash join comparison with extended set of algorithms. All using 64 threads.
The End