Main-Memory Hash Joins on Modern Processor Architectures

Paper by Balkesen, C. et al
6.886 Presentation Slides by Taylor Andrews
Presentation Agenda

● Part 1
  ○ Background & Problem Motivation
  ○ Optimized sequential & parallel hash-join algorithms

● Part 2
  ○ Experiments
  ○ Results
  ○ Discussion

https://en.wikipedia.org/wiki/Relational_algebra#Joins_and_join-like_operators
**Background: Join Operation**

- Relational algebra operation (natural join: \( R \bowtie S \))
- Focus on one algorithm family
  - “Main-memory hash-based” joins
  - Sequential and parallel variants

https://en.wikipedia.org/wiki/Relational_algebra#Joins_and_join-like_operators
**Problem Motivation:**

- Hash-based joins are common but computationally expensive.
- The *canonical sequential form*:

\[ O(|R| + |S|) \]

![Diagram of canonical hash join](https://en.wikipedia.org/wiki/Relational_algebra#Joins_and_join-like_operators)

---

**Fig. 1. Canonical hash join.**

---

https://en.wikipedia.org/wiki/Relational_algebra#Joins_and_join-like_operators
No-Partition Join Alg. (Parallel Improvement)

- Makes use of $P$ workers
- Divides the creation of shared hash table
- Still random memory access

\[ O\left(\frac{1}{P}(|R| + |S|)\right) \] (ideal)

Fig. 2. No partitioning join.
Partition Join Alg. (Parallel & Cache Improvement)

- Makes use of $P$ workers
- Divides creation of cache-aligned hash tables
- Better cache efficiency

$$O\left(\frac{1}{P}(|R| + |S|)\right)$$
(ideal)

Fig. 3. Partitioned hash join (following Shatdal et al. [9]).
**HW-Conscious Optimized Radix Join Algorithm** (ideal)

- Parallel, Cache & TLB improvements
- “Fan out” partitioning pass(es) dividing sub-problems among workers
- Calculates output memory ranges up front to avoid sync!
- Load distribution among threads by task queuing [6]

\[ O\left(\frac{1}{p}(|R| + |S|) \log |R|\right) \]

**Fig. 4. Radix join** (as proposed by Manegold et al. [10])
Experiments: Software Used

- All various Debian Linux OS (gcc == icc)
- “Bucket chaining” > SIMD, used globally
- Two previous papers’ workloads “A” & “B” ([3] & [1])
- Assumed unsorted input (simulate worst case)
- All test sets R and S have foreign key relationship
  - 1 join partner each
Experiments: (Diverse) Hardware Platforms Used

- Standard Intel Machines
  - 2 threads / core, shared 64-byte L3
- AMD Machine
  - 2 cores / module, shared instruction operations + FPU + L2
- Sun UltraSPARC T2
  - 8 threads / core, shared 16-byte L1, shared 64-byte L2
- High-End Multi-Core (Oracle Sparc T4 and Intel E5-4640)
  - 4 socket, 8 cores / socket, 8 threads / core, shared 64-byte L3
- See Table 1
**Results:** Making Oblivious More Conscious (Tuned Hash Table)

Fig. 7. Cycles per output tuple for hardware-oblivious no partitioning strategy (Workload A; Intel Xeon L5520, 2.26 GHz).

Fig. 5. Original hash table implementation in [3].

Fig. 6. Our hash table implementation.
**Results: Making Oblivious More Conscious**

- Aligning packed hash tables to avoid crossing cache lines
- Also tuned prefetching distance parameter
- Minimal returns from aligning alone (random access cost?)

---

**Fig. 17.** Impact of different optimizations on cycles per output tuple for *no partitioning* using Workload A (256 MiB × 4096 MiB); 8 threads, Intel Nehalem L5520.

**Fig. 18.** Impact of different optimizations on cycles per output tuple for *no partitioning* using Workload A (256 MiB × 4096 MiB); 16 threads, AMD Bulldozer Opteron 6276.
Results: Making Oblivious More Conscious

Fig. 5. Original hash table implementation in [3].

Fig. 6. Our hash table implementation.

Table 4
No partitioning join; cache misses per tuple (original code of Blanas et al. [3] vs. our own implementation).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Build</td>
<td>Probe</td>
</tr>
<tr>
<td>L2 misses</td>
<td>2.97</td>
<td>2.94</td>
</tr>
<tr>
<td>L3 misses</td>
<td>2.72</td>
<td>2.65</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Build</th>
<th>Probe</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.01</td>
<td>1.00</td>
</tr>
<tr>
<td>1.00</td>
<td>0.99</td>
</tr>
</tbody>
</table>
Quick Aside: Hardware Performance Counters

- General purpose counters that count events of interest
  - Event selection register
  - Various other control & overflow registers
- Intel details
  - Hardware: Intel SDM Chapter 18
  - Perf. Events: Intel SDM Chapter 19

| TABLE 2 |
|------------------------|------------------------|------------------------|
| CPU performance counter profiles for different radix join implementations (in millions); Workload A |
|                        | Part. | Build | Probe | Part. | Build | Probe |
| Cycles                 | 9,798 | 499   | 7,204 | 5614  | 171   | 542   |
| Instructions           | 3,520 | 2,000 | 3,681 | 17,506| 249   | 5650  |
| L2 misses              | 24    | 16    | 453   | 13    | 0.3   | 2     |
| L3 misses              | 5     | 5     | 40    | 7     | 0.2   | 1     |
| TLB load misses        | 9     | 0.3   | 2     | 13    | 0.1   | 1     |
| TLB store misses       | 425   | 0     | 0     | 170   | 0     | 0     |

<table>
<thead>
<tr>
<th>code from [3]</th>
<th>our code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part.</td>
<td>Build</td>
</tr>
<tr>
<td>5614</td>
<td>171</td>
</tr>
</tbody>
</table>
Results: Optimized Radix HW Performance Profile

- ~10x less instructions
- Less cache misses mostly improving build and probe
- Less TLB misses due to partitioning “fan out” pass(es)
- HW optimized radix performs best (except in a few cases)

<table>
<thead>
<tr>
<th></th>
<th>code from [3]</th>
<th>our code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Part.</td>
<td>Build</td>
</tr>
<tr>
<td>Cycles</td>
<td>9398</td>
<td>499</td>
</tr>
<tr>
<td>Instructions</td>
<td>33520</td>
<td>2000</td>
</tr>
<tr>
<td>L2 misses</td>
<td>24</td>
<td>16</td>
</tr>
<tr>
<td>L3 misses</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>TLB load misses</td>
<td>9</td>
<td>0.3</td>
</tr>
<tr>
<td>TLB store misses</td>
<td>325</td>
<td>0</td>
</tr>
</tbody>
</table>
Results: Optimized HW-Conscious Radix Improvements

- Tolerant across number of radix bits based on partition number (See Figure 8)
- Generally faster from hw-conscious optimization (packing and cache-aligning structs, TLB planning, avoiding calls and derefs)

Fig. 9. Overall join execution cost (cycles per output tuple) for hardware-conscious *radix join* strategy (Workload A; Intel Xeon L5520, 2.26 GHz).
Results: Final Rdx Optimization (Buffering, Write Combining, & TLB Saving)

Fig. 19. Partitioning performance comparison when using 4 KiB and 2 MiB pages (Using a single core on Intel Xeon L5520, 2.26 GHz).
Results: No-Part. and Optimized Radix Input Sizes

Fig. 12. Cycles per output tuple with varying build relation cardinalities in Workload A (Intel Xeon L5520, 2.26 GHz, Radix join was run with the best configuration in each experiment where radix bits varied from 13 to 15).
Results: Radix Parallel Scalability

Fig. 11. Throughput comparison of algorithms on different machines using Workload B. Computed as \frac{\text{input-size}}{\text{execution-time}} where input-size = |R| = |S|.
Three Hw-Conscious Optimized Radix Performance Outliers
Results: UltraSPARC T2 Niagara Performance Outlier 1

Figure 10:

(a) Workload A (256 MiB x 4096 MiB)

(b) Workload B (977 MiB x 977 MiB)
Results: Performance Outlier 1 (When Radix Is Slow)

- Oblivious N-partitioning faster than conscious radix
- UltraSPARC T2 Niagara 8kB virtual memory pages & fully associative TLB
- Extremely efficient thread synchronization
  - Performant ldstub instruction latch implementation

### Table 3

<table>
<thead>
<tr>
<th>Used Instruction</th>
<th>Nehalem</th>
<th>Sandy Bridge</th>
<th>Bulldozer</th>
<th>Niagara 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reported instruction latency in [18], [19]</td>
<td>~20 cycles</td>
<td>~25 cycles</td>
<td>~50 cycles</td>
<td>3 cycles</td>
</tr>
<tr>
<td>Measured impact per build tuple</td>
<td>7-9 cycles</td>
<td>6-9 cycles</td>
<td>30-34 cycles</td>
<td>1-1.5 cycles</td>
</tr>
</tbody>
</table>
Results: Parallel Scalability Outlier 2

- Note black square and triangle similar perf despite 2x threads
  - Why?
  - SMP not as effective due to lowered cache misses (less core idle time)
- See Figure 13 & 14 & 15

Fig. 13. Performance on recent multi-core servers, Sparc T4 and Sandy Bridge (SB) using Workload B. Throughput is in output tuples per second, i.e. \( \ell \) execution time.
**Results: Perf. Outlier 3 (More Radix Threads Slower)**

![Graph showing impact of number of threads on Sparc T4](image)

*Fig. 16. Impact of number of threads on Sparc T4 on overall performance for different algorithms. Using Workload B (977 MiB x 977 MiB); Oracle Sparc T4.*
Results: Perf. Outlier 3 Due To Sparc T4 Architecture

- Hardware supports 8 hardware threads using shared core resources (including cache space!)
- Actually induces higher cache misses
- Sometimes, “Less is more for hardware conscious [hash-join] algorithms” (Page 10)

From https://www.slideshare.net/solarisyougood/sparc-t4-systems-customer-presentation
Concluding Summary:

- HW-conscious, optimized radix maintains edge
  - Except on certain aggressive SMP hardware including low sync overhead
  - Except with enough core saturation to degrade SMP ability
  - Except when over-threading thrashes shared hardware caches
- Compared hash-join algorithms on real hardware
  - HW-oblivious n-partition through HW-conscious, optimized radix
- Optimized existing oblivious & conscious implementations
  - Packing and tuning the hash table structure
  - Pointer array indices avoiding calls and mem deref.
- Oblivious hash join algorithms can be competitive, but in special hardware circumstances
Related Work:

- Hash-join algorithm content origin, see [6]
- Similar partitioning spirit for aggregation [23]
  - Different problem, similar hardware findings
- NUMA added complexity: “handshake-join” [24]
- Sort-merge algorithms leveraging sequential memory [4]
- GPU-based join leveraging hardware SMT idea [25]
- Cache oblivious design at the database level [26]
Thanks and Discussion

6.886 Presentation Slides by Taylor Andrews (tandrews@mit.edu)

- What could implementation look like for real databases?
  - Library?
  - Service?

- Can additional caching of join results be leveraged?
  - Storage vs. compute costs

- Which other hardware architecture is most interesting?
  - 8 way threading on 8 cores, in 4 sockets (Sparc T4)
  - 2 cores per module, shared instruction operations, FPU & L2 (AMD)
  - 8 way threading on 8 cores, smaller cache lines (Sparc T6)
  - Ideas for specialized use cases?