Parallel Joins in Main-Memory Data(base) Processing

Alfons Kemper, et al.
Fakultät für Informatik, TUM
Joining two Relations R and S: $R \Join S$

<table>
<thead>
<tr>
<th>$R$</th>
<th>$S$</th>
<th>$R \Join S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$B$</td>
<td>$C$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$b_2$</td>
<td>$c_2$</td>
</tr>
<tr>
<td>$a_3$</td>
<td>$b_3$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>$a_4$</td>
<td>$b_4$</td>
<td>$c_2$</td>
</tr>
<tr>
<td>$a_5$</td>
<td>$b_5$</td>
<td>$c_3$</td>
</tr>
<tr>
<td>$a_6$</td>
<td>$b_6$</td>
<td>$c_2$</td>
</tr>
<tr>
<td>$a_7$</td>
<td>$b_7$</td>
<td>$c_6$</td>
</tr>
</tbody>
</table>
Algorithmics:

\[ R \cap S \]

R:
2
3
44
5
78
90
13
17
42
89

S:
44
17
97
5
6
27
2
13
9
Algorithmics:

\[ R \cap S \]

- **Nested Loop**: \( O(N^2) \)
- **Sort-Based**: \( O(N \log N) \)
- **Partitioning and Hashing**
Parallel Radix-Join

Diagram showing the process of parallel radix-join with tables S, S0, S1, T0, T1, and T.
Multiple Phase Radix-Joins: Cache-Locality
Hash-Join-Teams: Global Hash Table
Fig. 1: Pessimistic vs. optimistic write access to a hash table
Listing 1.1: Atomic insert function

```c
insertAtomic(uint64_t key, uint64_t value) {
    uint64_t hash = hashFunction(key);
    uint64_t pos = hash & mask;
    while (table[pos].h != 0
        || (! CAS(&table[pos].h, 0, hash))) {
        pos = (pos + 1) & mask;
    }
    table[pos].k = key;
    table[pos].v = value;
}
```
Alternative Hash Table with Overflow Buckets

```
insert(entry) {
  // determine slot in hash table
  slot = entry->hash >> hashTableShift
  do {
    old = hashTable[slot]
    // set next to old entry without tag
    entry->next = removeTag(old)
    // add old and new tag
    new = entry | (old&tagMask) | tag(entry->hash)
    // try to set new value, repeat on failure
    while (!CAS(hashTable[slot], old, new))
  }
}
```
Build performance using different synchronization mechanisms.
Figure 1: Idea of morsel-driven parallelism: $R \bowtie_A S \bowtie_B T$
Massively parallel processing
Figure 3: NUMA-aware processing of the build-phase
Figure 4: Morsel-wise processing of the probe phase
Lock-free Data Structures of Dispatcher
List of pending pipeline-jobs
(possibly belonging to different queries)

Example NUMA Multi-Core Server with 4 Sockets and 32 Cores
Massively Parallel Hash Aggregation

Figure 8: Parallel aggregation
Sorting in Parallel

Figure 9: Parallel merge sort
Massively Parallel Sort-Merge Joins (MPSM) in Main Memory Multi-Core Database Systems

Martina Albutiu, Alfons Kemper, and Thomas Neumann

Technische Universität München
Hardware trends ...

- Huge main memory
- Massive processing parallelism
- Non-uniform Memory Access (NUMA)
- Our server:
  - 4 CPUs
  - 32 cores
  - 1 TB RAM
  - 4 NUMA partitions
Ignoring NUMA
How much difference does NUMA make?

- Remote sort: 22756 ms
- Local sort: 7440 ms
- Synchronized sort: 417344 ms
- Sequential sort: 12946 ms
- Remote merge join: 1000 ms
- Local merge join: 837 ms

Scaled execution time: 100%
The three NUMA commandments

C1 Thou shalt not write thy neighbor's memory randomly -- chunk the data, redistribute, and then sort/

C2 Thou shalt read thy neighbor's memory only sequentially -- let the prefetcher hide the remote access latency.

C3 Thou shalt not wait for thy neighbors -- don't use fine-grained latching or locking and avoid synchronization points of parallel threads.
Basic idea of MPSM

R chunks

chunk R

S chunks

chunk S
Basic idea of MPSM

- **C1:** Work locally: sort
- **C3:** Work independently: sort and merge join
- **C2:** Access neighbor’s data only sequentially

![Diagram showing the steps of MPSM](image)
Range partitioning of private input $R$

- To constrain merge join work
- To provide scalability in the number of parallel workers
Range partitioning of private input $R$

- To constrain merge join work
- To provide scalability in the number of parallel workers

Range partitioning of $R$ chunks

range partitioned $R$ chunks

range partition $R$
Range partitioning of private input R

- To constrain merge join work
- To provide scalability in the number of parallel workers

→ S is implicitly partitioned

\[ \begin{align*}
\text{range} & \quad \text{partitioned} \\

\text{R chunks} & \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \\
\text{sort R chunks} & \\
\text{S chunks} & \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\
\text{sort S chunks} & 
\end{align*} \]
Range partitioning of private input R

- To constrain merge join work
- To provide scalability in the number of parallel workers

→ S is implicitly partitioned

R chunks

S chunks

Range partitioning of input R

Sort R chunks

Merge join only relevant parts

Sort S chunks
Range partitioning of private input

- Time efficient
  - branch-free
  - comparison-free
  - synchronization-free

and

- Space efficient
  - densely packed
  - in-place

by using radix-clustering and precomputed target partitions to scatter data to
Range partitioning of private input

chunk of worker $W_1$

<table>
<thead>
<tr>
<th>19</th>
<th>9</th>
<th>7</th>
<th>3</th>
<th>21</th>
<th>1</th>
<th>17</th>
</tr>
</thead>
</table>

chunk of worker $W_2$

| 2 | 23 | 4 | 31 | 8 | 20 | 26 |

histogram of worker $W_1$

| 4 | 3 |

prefix sum of worker $W_1$

| 0 | 1 |

histogram of worker $W_2$

| 3 | 4 |

prefix sum of worker $W_2$

| 5 | 3 | 19 |

$19 = 10011$

$7 = 00111$

$17 = 10001$

$2 = 00010$

$19 = 0011$

$2 = 0010$

$19 = 10011$

$2 = 00010$
Range partitioning of private input

**Histograms and prefix sums for workers W₁ and W₂**

- **Worker W₁**
  - Input chunk: 19, 9, 7, 3, 21, 1, 17
  - Histogram: 4, 3
  - Prefix sum: 0, 1

- **Worker W₂**
  - Input chunk: 2, 23, 4, 31, 8, 20, 26
  - Histogram: 3, 4
  - Prefix sum: 4, 5

**Example calculations**

- **Worker W₁, input 7:**
  - Binary: 0011
  - Prefix sum: 0 + 1 = 1

- **Worker W₂, input 2:**
  - Binary: 0010
  - Prefix sum: 0 + 1 = 1

**Chunk assignments**

- **Worker W₁**
  - Chunks: 19, 9, 7, 3, 21, 1, 17

- **Worker W₂**
  - Chunks: 2, 23, 4, 31, 8, 20, 26

**Notes**

- The prefix sum is updated as new inputs are processed.
- The example calculations show how increments are added to the prefix sum.
- The input and output are both in binary format.
Real C hacker at work …

\[ p_{S_i}[j] = \& R_j \left[ \left( \sum_{k=1}^{i-1} h_k[j] \right) \right] \]

\[
\text{memcpy}(p_{S_i}[sp[t.key \gg (64 - B)]]++, t, t.size)
\]