An Interval Join Optimized for Modern Hardware

Danila Piatov   Sven Helmer   Anton Dignös

Centre for Information and Database Systems Engineering (IDSE)  
Faculty of Computer Science  
Free University of Bozen-Bolzano, Italy

ICDE 2016, Helsinki, Finland
Interval Join

Problem:
Find all pairs of intervals from \( r \) and \( s \) that overlap in time.

Answer:
\( \{ r_1, s_1 \}, \{ r_1, s_2 \}, \{ r_1, s_3 \}, \{ r_1, s_4 \}, \{ r_2, s_1 \}, \{ r_2, s_2 \}, \{ r_2, s_3 \}, \{ r_2, s_4 \}, \{ r_2, s_5 \}, \{ r_3, s_2 \}, \{ r_3, s_5 \} \)
Interval Join

**Problem:** Find all pairs of intervals from \( r \) and \( s \) that overlap in time.

```
Problem: Find all pairs of intervals from r and s that overlap in time.
```
**Problem:** Find all pairs of intervals from $r$ and $s$ that overlap in time.

**Answer:** $\langle r_1, s_1 \rangle, \langle r_1, s_2 \rangle, \langle r_1, s_3 \rangle, \langle r_1, s_4 \rangle, \langle r_2, s_1 \rangle, \langle r_2, s_2 \rangle, \langle r_2, s_3 \rangle, \langle r_2, s_4 \rangle, \langle r_2, s_5 \rangle, \langle r_3, s_2 \rangle, \langle r_3, s_5 \rangle$. 

In the diagram, $r_1, r_2, r_3$ are intervals on the $r$ side and $s_1, s_2, s_3, s_4, s_5$ are intervals on the $s$ side. The overlapping intervals are indicated by the intersections.
So, what’s the problem?

```
SELECT *
FROM r, s
WHERE r.Ts <= s.Te AND s.Ts <= r.Te
```
So, what’s the problem?

SELECT *
FROM r, s
WHERE r.Ts <= s.Te AND s.Ts <= r.Te

Join on two independent inequality predicates
No optimization in standard RDBMSs.
Idea: List interval endpoints $\langle T_s, \text{start}, \text{TID} \rangle$ and $\langle T_e, \text{end}, \text{TID} \rangle$ in chronological order
Endpoint Index

**Idea:** List interval endpoints \( \langle T_s, \text{start}, TID \rangle \) and \( \langle T_e, \text{end}, TID \rangle \) in chronological order

**Result:** Endpoint index for relation \( r \) is \([\langle 1, \text{start}, 1 \rangle, \langle 1, \text{start}, 2 \rangle, \langle 5, \text{end}, 1 \rangle, \langle 7, \text{start}, 3 \rangle, \langle 10, \text{end}, 2 \rangle, \langle 11, \text{end}, 3 \rangle]\).
Endpoint-Index-Based Interval Join

Active $r$ tuples: $\{\}$
Active $s$ tuples: $\{\}$
Result: $\{\}$
Endpoint-Index-Based Interval Join

Active $r$ tuples: $\{r_1, r_2\}$
Active $s$ tuples: $\{\}$
Result:
**Endpoint-Index-Based Interval Join**

Active **r** tuples: \{r_1, r_2\}

Active **s** tuples: \{s_1\}

Result: \( \langle r_1, s_1 \rangle, \langle r_2, s_1 \rangle \)
Endpoint-Index-Based Interval Join

Active \textbf{r} tuples: \{r_1, r_2\}
Active \textbf{s} tuples: \{s_1, s_2\}
Result: \langle r_1, s_1 \rangle, \langle r_2, s_1 \rangle, \langle r_1, s_2 \rangle, \langle r_2, s_2 \rangle
Endpoint-Index-Based Interval Join

Active r tuples: \( \{r_1, r_2\} \)
Active s tuples: \( \{s_1, s_2, s_3\} \)
Result: \( \langle r_1, s_1 \rangle, \langle r_2, s_1 \rangle, \langle r_1, s_2 \rangle, \langle r_2, s_2 \rangle, \langle r_1, s_3 \rangle, \langle r_2, s_3 \rangle \)
Endpoint-Index-Based Interval Join

Active \( r \) tuples: \( \{r_1, r_2\} \)
Active \( s \) tuples: \( \{s_1, s_2, s_3, s_4\} \)
Result: \( \langle r_1, s_1 \rangle, \langle r_2, s_1 \rangle, \langle r_1, s_2 \rangle, \langle r_2, s_2 \rangle, \langle r_1, s_3 \rangle, \langle r_2, s_3 \rangle, \langle r_1, s_4 \rangle, \langle r_2, s_4 \rangle \)
Endpoint-Index-Based Interval Join

Active r tuples: \{r_1, r_2\}
Active s tuples: \{s_1, s_2, s_3, s_4\}
Result: \langle r_1, s_1 \rangle, \langle r_2, s_1 \rangle, \langle r_1, s_2 \rangle, \langle r_2, s_2 \rangle, \langle r_1, s_3 \rangle, \langle r_2, s_3 \rangle, \langle r_1, s_4 \rangle, \langle r_2, s_4 \rangle
Endpoint-Index-Based Interval Join

Active r tuples: \( \{r_1, r_2, r_3\} \)
Active s tuples: \( \{s_1, s_2, s_3, s_4\} \)
Result: \( \langle r_1, s_1 \rangle, \langle r_2, s_1 \rangle, \langle r_1, s_2 \rangle, \langle r_2, s_2 \rangle, \langle r_1, s_3 \rangle, \langle r_2, s_3 \rangle, \langle r_1, s_4 \rangle, \langle r_2, s_4 \rangle, \langle r_3, s_2 \rangle \)
Endpoint-Index-Based Interval Join

Active r tuples: \{r_1, r_2, r_3\}
Active s tuples: \{s_1, s_2, s_3, s_4, s_5\}
Result: \langle r_1, s_1 \rangle, \langle r_2, s_1 \rangle, \langle r_1, s_2 \rangle, \langle r_2, s_2 \rangle, \langle r_1, s_3 \rangle, \langle r_2, s_3 \rangle, \langle r_1, s_4 \rangle, \langle r_2, s_4 \rangle, \langle r_3, s_2 \rangle, \langle r_2, s_5 \rangle, \langle r_3, s_5 \rangle
Endpoint-Index-Based Interval Join

Active $r$ tuples: $\{r_1, r_2, r_3\}$
Active $s$ tuples: $\{s_1, s_2, s_3, s_4, s_5\}$
Result: $\langle r_1, s_1 \rangle, \langle r_2, s_1 \rangle, \langle r_1, s_2 \rangle, \langle r_2, s_2 \rangle, \langle r_1, s_3 \rangle, \langle r_2, s_3 \rangle, \langle r_1, s_4 \rangle, \langle r_2, s_4 \rangle, \langle r_3, s_2 \rangle, \langle r_2, s_5 \rangle, \langle r_3, s_5 \rangle$
Endpoint-Index-Based Interval Join

Active r tuples: \{r_1, r_2, r_3\}
Active s tuples: \{s_1, s_2, s_3, s_4, s_5\}
Result: \{r_1, s_1\}, \{r_2, s_1\}, \{r_1, s_2\}, \{r_2, s_2\}, \{r_1, s_3\}, \{r_2, s_3\}, \{r_1, s_4\}, \{r_2, s_4\}, \{r_3, s_2\}, \{r_2, s_5\}, \{r_3, s_3\}, \{r_3, s_5\}
Endpoint-Index-Based Interval Join

Active r tuples: \{ r_1, r_2, r_3 \}
Active s tuples: \{ s_1, s_2, s_3, s_4, s_5 \}
Result: \langle r_1, s_1 \rangle, \langle r_2, s_1 \rangle, \langle r_1, s_2 \rangle, \langle r_2, s_2 \rangle, \langle r_1, s_3 \rangle, \langle r_2, s_3 \rangle, \langle r_1, s_4 \rangle, \langle r_2, s_4 \rangle, \langle r_3, s_2 \rangle, \langle r_2, s_5 \rangle, \langle r_3, s_5 \rangle
Endpoint-Index-Based Interval Join

Active r tuples: \{r_1, r_2, r_3\}
Active s tuples: \{s_1, s_2, s_3, s_4, s_5\}
Result: \{r_1, s_1\}, \{r_2, s_1\}, \{r_1, s_2\}, \{r_2, s_2\}, \{r_1, s_3\}, \{r_2, s_3\}, \{r_1, s_4\}, \{r_2, s_4\}, \{r_3, s_2\}, \{r_2, s_5\}, \{r_3, s_5\}
Endpoint-Index-Based Interval Join

Active r tuples: \{r_1, r_2, r_3\}
Active s tuples: \{s_1, s_2, s_3, s_4, s_5\}
Result: \langle r_1, s_1 \rangle, \langle r_2, s_1 \rangle, \langle r_1, s_2 \rangle, \langle r_2, s_2 \rangle, \langle r_1, s_3 \rangle, \langle r_2, s_3 \rangle, \langle r_1, s_4 \rangle, \langle r_2, s_4 \rangle, \langle r_3, s_2 \rangle, \langle r_2, s_5 \rangle, \langle r_3, s_5 \rangle
Active tuple sets

- Associative arrays (maps) of TIDs to tuples
- Should support:
  - Tuple insertion (with TID)
  - Tuple removal by TID
  - Scanning of all tuples
Active tuple sets

- Associative arrays (maps) of TIDs to tuples
- Should support:
  - Tuple insertion (with TID)
  - Tuple removal by TID
  - Scanning of all tuples
- Good candidate is hash map...
Active tuple sets

- Associative arrays (maps) of TIDs to tuples
- Should support:
  - Tuple insertion (with TID)
  - Tuple removal by TID
  - Scanning of all tuples
- Good candidate is hash map...
- ...but it’s not very suited for scanning
- Existing solutions:
  - Scan through buckets (std::unordered_map, java.util.HashMap)
  - Connect elements via linked list (java.util.LinkedHashMap)
Standard Linked Hash Map

Hash table

<table>
<thead>
<tr>
<th>Key</th>
<th>List Prev</th>
<th>List Next</th>
<th>Bucket Next</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>Tuple 5</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td>Tuple 7</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td>Tuple 9</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>Tuple 2</td>
</tr>
</tbody>
</table>

Danila Piatov, Sven Helmer, Anton Dignös
Random vs. Sequential Memory Access

- Random memory access latency:
  - Within L1 cache (32 KB per core): 4 CPU cycles
  - Within L2 cache (256 KB per core): 11–12 cycles
  - Within L3 cache (3–45 MB): 30–40 cycles
  - Within RAM: approximately 70 ns (200 cycles)
Random vs. Sequential Memory Access

- **Random memory access latency:**
  - Within L1 cache (32 KB per core): 4 CPU cycles
  - Within L2 cache (256 KB per core): 11–12 cycles
  - Within L3 cache (3–45 MB): 30–40 cycles
  - Within RAM: approximately 70 ns (200 cycles)

- **Sequential RAM access speed:**
  - A thread can read RAM at \( \sim 10 \text{ GB/s} \)
  - 1 ns (about 3 CPU cycles) for reading every 10 bytes
Gapless Hash Map

Hash table

Key

Bucket Prev

Bucket Next

Value

Tail

Tuple 5

Tuple 7

Tuple 9

Tuple 2

5

7

9

2
Gapless Hash Map (Separated Values)
Even sequential, RAM scan is slower than L1 cache scan.
Even sequential, RAM scan is slower than L1 cache scan

**Observation:** In our demo the unmodified set of active \( r \) tuples was scanned 4 times in a row, once for each of \( s_1, s_2, s_3 \) and \( s_4 \).
Endpoint-Index-Based Interval Join

Active r tuples: \(\{r_1, r_2\}\)
Active s tuples: \(\{s_1\}\)
Result: \(\langle r_1, s_1 \rangle, \langle r_2, s_1 \rangle\)
Endpoint-Index-Based Interval Join

Active \( r \) tuples: \( \{r_1, r_2\} \)
Active \( s \) tuples: \( \{s_1, s_2\} \)
Result: \( \langle r_1, s_1 \rangle, \langle r_2, s_1 \rangle, \langle r_1, s_2 \rangle, \langle r_2, s_2 \rangle \)
Endpoint-Index-Based Interval Join

Active \( r \) tuples: \( \{r_1, r_2\} \)
Active \( s \) tuples: \( \{s_1, s_2, s_3\} \)
Result: \( \langle r_1, s_1 \rangle, \langle r_2, s_1 \rangle, \langle r_1, s_2 \rangle, \langle r_2, s_2 \rangle, \langle r_1, s_3 \rangle, \langle r_2, s_3 \rangle \)
Endpoint-Index-Based Interval Join

Active r tuples: \{r_1, r_2\}
Active s tuples: \{s_1, s_2, s_3, s_4\}
Result: \langle r_1, s_1 \rangle, \langle r_2, s_1 \rangle, \langle r_1, s_2 \rangle, \langle r_2, s_2 \rangle, \langle r_1, s_3 \rangle, \langle r_2, s_3 \rangle, \langle r_1, s_4 \rangle, \langle r_2, s_4 \rangle
Even sequential, RAM scan is slower than L1 cache scan

**Observation:** In our demo the unmodified set of active \( r \) tuples was scanned 4 times in a row, once for each of \( s_1, s_2, s_3 \) and \( s_4 \).

**Idea:** Collect these \( s \) tuples into small array fitting L1 CPU cache and produce cross-product with active set of \( r \) tuples by scanning it just once.
Hash map scanning performance

The diagram shows the latency of `getnext` operation for two types of hash maps: Linked hash map and Gapless hash map. The x-axis represents the map size in tuples (32 bytes) ranging from $10^2$ to $10^8$, while the y-axis shows the latency in nanoseconds (ns). The results are plotted on a logarithmic scale.

Key observations:
- **Linked hash map**
  - Shows significant latency spikes at specific map sizes.
  - Latency increases steeply with increasing map size.

- **Gapless hash map**
  - Exhibits a more stable latency pattern.
  - Latency remains relatively constant across different map sizes.

The graph highlights the performance differences between the two map types, with the Gapless hash map generally offering better performance across the range of map sizes tested.
EBI-Join vs. LEBI-Join

Joining time, s

EBI-Join, gapless, query only
LEBI-Join, gapless, query only

Workload

Danila Piatov, Sven Helmer, Anton Dignös
An Interval Join Optimized for Modern Hardware
Comparison With the State-of-the-Art

![Graph comparing LEBI-Join and OIPJoin](image_url)

- LEBI-Join, gapless, index + query
- OIPJoin, index + query

**Workload**

- Joining time, s

---

Danila Piatov, Sven Helmer, Anton Dignös

An Interval Join Optimized for Modern Hardware
Conclusion

- We took the endpoint-index-based interval join (EBI-Join)
- We introduced two memory-hierarchy-aware optimizations for it:
  - Gapless hash map
  - Lazy evaluation technique (LEBI-Join)
- With these optimizations we are able to outperform the state-of-the-art