Stab-Forests: Dynamic Data Structures for Efficient Temporal Query Processing

Jelle Hellings

Yuqing Wu
Temporal event-based data

Consider *events* of the form \( \langle \text{start-time}, \text{end-time} \rangle \).

**Example**

- Process runtime in computer systems.
- Airline On-Time Performance Data (AOTPD).
- Civil Unrest Event Data (CUED).

**Definition**

Let \( R \) and \( S \) be sets of events. The temporal join \( R \bowtie S \) is defined by

\[
R \bowtie S = \{(e_1, e_2) \in R \times S \mid e_1 \cap e_2 \neq \emptyset \}.
\]
High-performance temporal joins $R \Join S$ via FwdScan

- All data in internal memory
  $\implies R$ and $S$: arrays of events, ordered on time.

- Cache-friendly traversal of data
  $\implies$ sequential scan of $R$ and $S$.

Proposition

$\text{FwdScan}(R, S)$ computes $R \Join S$ in worst-case $O(|R| + |S| + |\text{output}|)$. 
Temporal joins via FwDS\textsc{can} (example)

\[(R_0, S_0), (R_0, S_1), (R_0, S_2), (R_1, S_0), (R_1, S_1), (R_3, S_2), (R_3, S_3), (R_4, S_3)\]

Can we prevent inspection of $R_2$?
Temporal joins via FwDScan (example)

\[ [(R_0, S_0), (R_0, S_1), (R_0, S_2), (R_1, S_0), (R_1, S_1), (R_3, S_2), (R_3, S_3), (R_4, S_3)] \]
Temporal joins via FwdScan (example)

\[ (R_0, S_0), (R_0, S_1), (R_0, S_2), (R_1, S_0), (R_1, S_1), (R_3, S_2), (R_3, S_3), (R_4, S_3) \]
Temporal joins via FwdScan (example)

\[ [(R_0, S_0), (R_0, S_1), (R_0, S_2), (R_1, S_0), (R_1, S_1), (R_3, S_2), (R_3, S_3), (R_4, S_3)] \]
Temporal joins via FwDScan (example)

\[
\begin{align*}
(R_0, S_0), (R_0, S_1), (R_0, S_2), (R_1, S_0), (R_1, S_1), (R_3, S_2), (R_3, S_3), (R_4, S_3)
\end{align*}
\]
Temporal joins via FwdScan (example)

\[(R_0, S_0), (R_0, S_1), (R_0, S_2), (R_1, S_0), (R_1, S_1), (R_3, S_2), (R_3, S_3), (R_4, S_3)\]
Temporal joins via FwdScan (example)

\[(R_0, S_0), (R_0, S_1), (R_0, S_2), (R_1, S_0), (R_1, S_1), (R_3, S_2), (R_3, S_3), (R_4, S_3)\]
Temporal joins via FwDScan (example)

\[ \{(R_0, S_0), (R_0, S_1), (R_0, S_2), (R_1, S_0), (R_1, S_1), (R_3, S_2), (R_3, S_3), (R_4, S_3)\} \]
Temporal joins via FwdScan (example)

\[(R_0, S_0), (R_0, S_1), (R_0, S_2), (R_1, S_0), (R_1, S_1), (R_3, S_2), (R_3, S_3), (R_4, S_3)\]
Temporal joins via FwDScan (example)

\[
[(R_0, S_0), (R_0, S_1), (R_0, S_2), (R_1, S_0), (R_1, S_1), (R_3, S_2), (R_3, S_3), (R_4, S_3)]
\]

Question
Can we prevent inspection of \( R_2 \)?
The **SkipJoin** algorithm

**Algorithm** `SkipJoin(R, S)`:  

1. $i, j := 0, 0$
2. while $i < |R|$ and $j < |S|$ do  
3. if $R[i].\text{start} \leq S[j].\text{start}$ then  
4. if $S[j].\text{start} \leq R[i].\text{end}$ then  
5. Join $R[i]$ with $S[j]$  
6. $i := i + 1$
7. else  
8. $(i, L) := \text{STAB}(R[i...], S[j].\text{start})$
9. For each event $e \in L$, join $e$ with $S[j...]$
10. else analogous (swap roles of $R$ and $S$)  

\[ \text{STAB}(R, t) = \{ e \in R \mid e.\text{start} \leq t \leq e.\text{end} \} \]
High-performance sequences of stab queries

Problem
Cannot efficiently query an array $A$ for all events active at $t$.

Solution
Stab-forest: index-structure supporting sequences of stab-queries.
- A fully-balanced binary search tree to minimize size.
- Event-based augmentations via compact static arrays.
- Efficiently supports append-only operations.
Stab-forests: Collections of stab-trees
Stab-forests: Collections of stab-trees

⟨0, 3⟩ ⟨0, 11⟩ ⟨1, 2⟩ ⟨2, 3⟩ ⟨4, 5⟩ ⟨5, 5⟩ ⟨5, 6⟩ ⟨6, 8⟩ ⟨7, 7⟩ ⟨7, 9⟩

E₀ E₁ E₂ E₃ E₄ E₅ E₆ E₇ E₈ E₉

(0, 0) (1, 1) (2, 2) (3, 4) (6, 6) (5, 5) (7, 7)
Stab-forests: Querying a stab-tree

Query: “MultiStab(\mathcal{S}, [0, 2, 7])”.
Stab-forests: Querying a stab-tree

Query: “MultiStab(\mathcal{S}, [0, 2, 7])”.
Stab-forests: Querying a stab-tree

Query: “MultiStab(\(\mathcal{S}, [0, 2, 7]\))”.
Stab-forests: Querying a stab-tree

Query: “MultiStab(\(S, [0, 2, 7]\))”. 
Stab-forests: Cache-friendly querying a stab-tree

<table>
<thead>
<tr>
<th></th>
<th>E₀</th>
<th>E₁</th>
<th>E₂</th>
<th>E₃</th>
<th>E₄</th>
<th>E₅</th>
<th>E₆</th>
<th>E₇</th>
<th>E₈</th>
<th>E₉</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨₀, ₀⟩</td>
<td>⟨₀, ₃⟩</td>
<td>⟨₁, ₂⟩</td>
<td>⟨₂, ₁⟩</td>
<td>⟨₃, ₄⟩</td>
<td>⟨₄, ₅⟩</td>
<td>⟨₅, ₅⟩</td>
<td>⟨₆, ₆⟩</td>
<td>⟨₇, ₇⟩</td>
<td>⟨₇, ₉⟩</td>
<td>⟨₀, ₀⟩</td>
</tr>
</tbody>
</table>

Query: “\textbf{MultiStab}(S, [0, 2, 7])”.
Stab-forests: Cache-friendly querying a stab-tree

Query: “MultiStab(\mathcal{S}, [0, 2, 7])”.
Stab-forests: Cache-friendly querying a stab-tree

Query: “MultiStab(\mathcal{S}, [0, 2, 7])”.
Stab-forests: Cache-friendly querying a stab-tree

Query: “MultiStab(S, [0, 2, 7])”.
SkipJoin with stab-forests: Results

Theorem
A stab-forest indexing $L$
- can be stored in worst-case $O(|L|)$ space.
- can be constructed in $O(|L| \log |L|)$.
- can be appended-to in amortized $O(\log |L|)$.
- can be stabbed for timestamps $\phi$ in $O(\min(|\phi| \log |L|, |\phi| + |L|) + |output|)$.

Theorem
\texttt{SkipJoin}(R, S) computes $R \bowtie S$ in worst-case $O(M(R, S) + M(S, R) + |output|)$, with $M(A, B)$ the cost of stabbing $A$ with $|B|$ timestamps.
Self-joining real-world datasets

- **AOTPD dataset**
  - 61,100,539 events in 10 years.
  - Shortest: 0 min. Longest: 1350 min.
  - Used: 2,500,000 events.

- **CUED dataset**
  - 62,141 events in 60 years.
  - Shortest: 0 d. Longest: 18,407 d.
Sparse joins and dedicated multi-window-queries

Query: “select all events on the 7-th day from the first n months in AOTP4”.
The behavior of **SkipJOIN**

Query: “$R \bowtie S$ with (gap size 3)".
The costs of stab-forests

![Graph 1: Construction time vs. Number of events](image1)

- stab-forest
- multiset
- multiset(*)
- vector

![Graph 2: Memory consumption vs. Number of events](image2)

- stab-forest
- multiset
- multiset(*)
- vector
Conclusion

**SkipJoin**: (windowed) temporal joins on skewed datasets.
**Stab-forests**: append-only index for sequences of stab-queries.

Future Work

- Using stab-forests for other temporal operations.
- Block-based external-memory designs based on the append-only stab-forests.
- Parallel and distributed high-performance temporal joins.

https://jhellings.nl/