View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

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

Jelle Hellings
Exploratory Systems Lab, Department of Computer Science,
University of California, Davis, CA, USA
jhellings@ucdavis.edu

Yuqing Wu
Computer Science Department, Pomona College, Claremont, CA, USA
melanie.wu@pomona.edu

Abstract

Many sources of data have temporal start and end attributes or are created in a time-ordered manner. Hence, it is only natural to consider joining datasets based on these temporal attributes. To do so efficiently, several internal-memory temporal join algorithms have recently been proposed. Unfortunately, these join algorithms are designed to join entire datasets and cannot efficiently join skewed datasets in which only few events participate in the join result.

To support high-performance internal-memory temporal joins of skewed datasets, we propose the *skip-join algorithm*, which operates on *stab-forests*. The stab-forest is a novel dynamic data structure for indexing temporal data that allows efficient updates when events are appended in a time-based order. Our stab-forests efficiently support not only traditional temporal stab-queries, but also more general *multi-stab-queries*. We conducted an experimental evaluation to compare the skip-join algorithm with state-of-the-art techniques using real-world datasets. We observed that the skip-join algorithm outperforms other techniques by an order of magnitude when joining skewed datasets and delivers comparable performance to other techniques on non-skewed datasets.

2012 ACM Subject Classification Information systems → Join algorithms; Information systems → Temporal data

Keywords and phrases Cache-friendly temporal joins, temporal data, skewed data, stab-queries, temporal indices

Digital Object Identifier 10.4230/LIPIcs.TIME.2020.18

Supplementary Material Open-source code of the full implementation of the data structures, algorithms, and supporting tooling used can be found at `https://jhellings.nl/projects/skipjoin/`.

Funding This material is based upon work supported by the National Science Foundation under Grant No. NSF 1606557.

1 Introduction

In practice, most sources of data have temporal attributes. Examples include news events, air travel records, employment records, and event logs. Temporal attributes also play a role in data that does not have explicit temporal attributes: e.g., in versioned databases the time of creation and replacement of each data element is recorded such that the evolution of the database is maintained. Given that temporal data is so ubiquitous, applications naturally expect support from DBMSs for efficient operations based on these temporal attributes. Examples of such operations are *stab-queries* and the *temporal join*:

▶ **Example 1.1.** Consider complex systems in which events are logged by (start, end)-time intervals. We want to use the event log to diagnose failures in the complex system. More specifically, if a failure at time t needs to be diagnosed, one does not want to search through

© Jelle Hellings and Yuqing Wu;
licensed under Creative Commons License CC-BY
27th International Symposium on Temporal Representation and Reasoning (TIME 2020).
Editors: Emilio Muñoz-Velasco, Ana Ozaki, and Martin Theobald; Article No. 18; pp. 18:1–18:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the entire event log, but use more directed ways to look for causes. A first step would be to perform a *stab-query* at time t to find all events that are active when the failure happened. A next step would be to combine event logs of the failed system with event logs of another system via a *(windowed) temporal join* that yields pairs of events that were active at the same time (and could have influenced each other) in a 24 h-window around time t.

Efficient temporal join algorithms are at the basis of many other efficient temporal operations. E.g., selecting all events in a dataset that occur during a given set of windows in time is equivalent to a temporal join between the dataset and these windows. Unfortunately, state-of-the-art techniques fail to cope with skewed datasets or fail to deliver high performance:

Temporal join algorithms. Many temporal join algorithms proposed in the literature fine-tune the usage of traditional relational database storage and join techniques towards temporal joining [3, 7, 13, 15, 26]. These relational-oriented temporal join algorithms are not optimized for *high-performance internal-memory operations* and achieve only acceptable performance.

Separately from these relational-oriented approaches, a few dedicated internal-memory join algorithms have been proposed that operate on ordered arrays of events. These algorithms use merge-join style methods that employ either sweeping-based techniques or forward-scan techniques [3, 6, 7, 13, 15, 21, 26]. Based on these merge-join style algorithms, Piatov et al. [21] recently introduced the endpoint join algorithm, a cache-friendly internal-memory temporal join algorithm. Due to the timestamp-based representation of events used by the endpoint join algorithm, the algorithm needs complicated data structures to maintain active lists of events while joining them. Bourus et al. [6] showed that a traditional forward-scan algorithm [7] operating on a simple event list will attain similar performance without all the complexities of the endpoint join algorithm. Unfortunately, these merge-join style algorithms inspect the entire dataset, due to which their performance degrades when the join output is restricted to a small window in time and when the datasets only have *few overlapping events*, the lather limiting their usability on skewed datasets.

Temporal data structures. Besides relying on temporal joins, one can also consider index structures that support answering temporal operations. Unfortunately, existing index structures either do not support temporal operations efficiently, are statically built or complex to maintain, or fail to provide high-performance cache-friendly internal-memory operations. Indeed, to support temporal operations over interval data, traditional relational indices such as binary search trees, B-trees, and range-trees *cannot be effectively used*, as these structures lack the information to efficiently perform stab-queries and other basic temporal operations [5, 13, 22, 26]. Alternatively, one can use specialized static interval data structures developed for geometric applications [1, 4, 5, 16, 18, 23]. Examples include interval trees, segment trees, and priority search trees [11, 12, 17]. These statically built data structures all support efficient stab-queries, but *do not support any form of updates*.

Unfortunately, dynamic general-purpose versions of these statically-built interval data structures are highly complex, have expensive maintenance algorithms, and rely completely on pointer-based tree structures [4, 9, 20]. The usage of such complex pointer-based data structure prevents cache-friendly traversal and, hence, prevent them from supporting *high-performance internal-memory operations* [4, 14, 24]. A few external memory interval data structures have been proposed, but these either place many restrictions on the inserted data [19, 25] or are highly complex and have not yet proven themselves in practice [4].

Our proposal: the skip-join algorithm. To address the shortcomings of existing techniques, we propose the *skip-join algorithm* (Section 2). Our skip-join algorithm is an efficient temporal join algorithm that can deal with all datasets and, additionally, supports windowed temporal joins. To do so, the skip-join algorithm uses the *stab-forest*, a novel temporal index data structure that is designed to efficiently support *stab-queries* and, more importantly, *multi-stab-queries* that yield the combined results of multiple stab-queries in a highly efficient manner (Section 3 and Section 4). We also present efficient ways to maintain stab-forests when events are appended (Section 5).

To show the effectiveness of the skip-join algorithm, we evaluate the performance of our algorithm using real-world datasets (Section 6). Our evaluation shows that the skip-join algorithm outperforms state-of-the-art join algorithms by an order of magnitude when joining skewed datasets. On dense datasets, the performance of the skip-join algorithm is comparable to the state-of-the-art.

2 The Skip-Join Algorithm

Before we propose the skip-join algorithm, we first introduce some event-related terminology. A *timestamp* represents a single point in time. We assume that *timestamps* are non-negative integers. An *event* is a pair $\langle v, w\rangle$ of timestamps that represents the interval $[v, w]$ in time. If $\mathrm{E} = \langle v, w\rangle$ is an event, then v is the *start-time* and w is the *end-time* and we write $\mathrm{E}.\mathtt{start}$ and $\mathrm{E}.\mathtt{end}$ to denote v and w, respectively. If $\mathrm{E}.\mathtt{start} \leq t \leq \mathrm{E}.\mathtt{end}$, then we also say that E is *active* at t. If E_1 and E_2 are events, then the *intersection* $\mathrm{E}_1 \cap \mathrm{E}_2$ is empty if $\mathrm{E}_1.\mathtt{end} < \mathrm{E}_2.\mathtt{start}$ or $\mathrm{E}_2.\mathtt{end} < \mathrm{E}_1.\mathtt{start}$. We say that E_1 and E_2 *overlap* if $\mathrm{E}_1 \cap \mathrm{E}_2 \neq \emptyset$.

▶ **Definition 2.1.** *Let R and S be sets of events. The* temporal join *of R and S, denoted by $R \bowtie S$, is defined by*

$$R \bowtie S = \{(\mathrm{E}_1, \mathrm{E}_2) \in R \times S \mid \mathrm{E}_1 \cap \mathrm{E}_2 \neq \emptyset\}.$$

Our skip-join algorithm relies on the ability to efficiently perform sequences of stab-queries:

▶ **Definition 2.2.** *Let S be a set of events, let t be a timestamp, and let ϕ be a sorted sequence of timestamps. The* stab-query of S by t *is defined by*

$$\textsc{Stab}(S, t) = \{\mathrm{E} \in S \mid \mathrm{E}.\mathtt{start} \leq t \leq \mathrm{E}.\mathtt{end}\},$$

and the multi-stab-query of S by ϕ *is defined by*

$$\textsc{MultiStab}(S, \phi) = \bigcup_{t \in \phi} \textsc{Stab}(S, t).$$

Several high-performance internal-memory temporal join algorithms have been proposed, most of which use a merge-join style method that employs either sweeping-based techniques or forward-scan techniques [3, 6, 7, 13, 15, 21, 26]. Unfortunately, these merge-join style algorithms cannot efficiently support windowed temporal joins or deal with skewed datasets.

To efficiently support windowed temporal joins and deal with skewed datasets, we will present our skip-join algorithm. To simplify presentation, we build skip-join on top of the forward-scan algorithm, the simplest among the merge-join style algorithms. Our techniques can, however, easily be translated to endpoint-based join algorithms, e.g., the algorithm of Piatov et al. [21].

Algorithm 1 Algorithm FwdScan, outputs $R \bowtie S$ (R and S sorted in ascending start-time order).

Algorithm FwdScan(R, S):

```
1: i, j := 0, 0
2: while i < |R| and j < |S| do
3:   if R[i].start ≤ S[j].start then
4:     k := j                                          #(Join R[i] with S[j...]).
5:     while k < |S| and S[k].start ≤ R[i].end do
6:       Output (R[i], S[k])
7:       k := k + 1
8:     i := i + 1
9:   else analogous (swap roles of R and S)
```

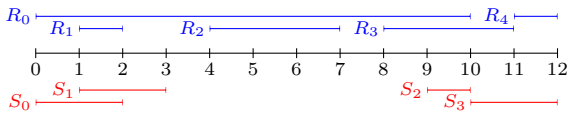

Figure 1 Two lists of events R and S visualized on an explicit timestamp scale.

The forward-scan temporal join algorithm. The forward-scan algorithm is a cache-friendly temporal join algorithm that can efficiently join non-skewed datasets represented by ordered lists of events (e.g., sorted arrays of events) [3, 6, 7, 13, 15, 21, 26]. The outline of such a forward-scan algorithm is shown in Algorithm 1.

▶ **Example 2.3.** Let $R = [\langle 0, 10\rangle, \langle 1, 2\rangle, \langle 4, 7\rangle, \langle 8, 11\rangle, \langle 11, 12\rangle]$ and $S = [\langle 0, 2\rangle, \langle 1, 3\rangle, \langle 9, 10\rangle, \langle 10, 12\rangle]$ be the lists of events visualized in Figure 1. We compute $R \bowtie S$ using Algorithm FwdScan. First, we join R_0 with $S[0\ldots]$, and output (R_0, S_0), (R_0, S_1), (R_0, S_2). Next, we join S_0 with $R[1\ldots]$, and output (R_1, S_0). Next, we join R_1 with $S[1\ldots]$, and output (R_1, S_1). Next we join S_1 with $R[2\ldots]$, and output nothing. Next, we join R_2 with $S[2\ldots]$, and output nothing. Next, we join R_3 with $S[2\ldots]$, and output (R_3, S_2), (R_3, S_3). Next, we join S_2 with $R[4\ldots]$, and output nothing. Next, we join S_3 with $R[4\ldots]$, and output (R_4, S_3). Finally, we stop, as we have reached the end of S.

If the event-list is implemented as an array, then these forward-scan algorithms will have high performance when most events in R and S are part of the join result:

▶ **Proposition 2.4.** *Let R and S be lists of events sorted in ascending start-time order. The algorithm* FwdScan(R, S) *computes $R \bowtie S$ in worst-case $\mathcal{O}(|R| + |S| + |output|)$.*

Dealing with skew in temporal joins. In practice, one can expect some skew in the data that causes standard forward-scan algorithm to waste time inspecting parts of R and/or S that are not part of the join result. To deal with this form of data skew, we need a way to detect and skip over parts of R and S that are irrelevant to the join result. To do so, we augment the forward-scan algorithm with the ability to use stab-queries to jump over irrelevant events: if, e.g., we are at an event $R[i]$ that ends before the event $S[j]$ starts, then we simply jump in R until we find the first event $R[i']$ that starts after $S[j]$. By jumping over events in R, we might miss events in $R[i\ldots i']$ that end after $S[j]$`.start`. To assure we do not miss such events, we jump over events in R via a stab-query and join the output of the stab-query with $S[j\ldots]$. This approach results in the skip-join algorithm of Algorithm 2.

▶ **Example 2.5.** Consider the lists of events R and S of Example 2.3 and visualized in Figure 1. We compute $R \bowtie S$ using the SkipJoin algorithm. First, we join R_0 with $S[0\ldots]$,

Algorithm 2 Algorithm SkipJoin, outputs $R \bowtie S$ (R and S sorted in ascending start-time order).

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

```
1:  i, j := 0, 0
2:  while i < |R| and j < |S| do
3:     if R[i].start ≤ S[j].start then
4:        if S[j].start ≤ R[i].end then
5:           Join R[i] with S[j ...] (see Algorithm 1, Lines 4–8)
6:           i := i + 1
7:        else
8:           (i, L) := Stab(R[i ...], R[i].start)
9:           For each event e ∈ L, join e with S[j ...] (see Algorithm 1, Lines 4–8)
10:    else analogous (swap roles of R and S)
```

and output (R_0, S_0), (R_0, S_1), (R_0, S_2). Next, we join S_0 with $R[1\dots]$, and output (R_1, S_0). Next, we join R_1 with $S[1\dots]$, and output (R_1, S_1). Next we join S_1 with $R[2\dots]$, and output nothing. Next, when we process the event $R_2 = \langle 4, 7\rangle$, we detect that the event $S_2 = \langle 9, 10\rangle$, the first event in $S[2\dots]$, starts after $R_2\texttt{.end}$ as $7 = R_2\texttt{.end} < S_3\texttt{.start} = 9$. Hence, we perform Stab($R[2\dots], 9$), which yields the list $[R_3]$ and the index 4 in R. We output (R_3, S_2) and continue with joining $R[4\dots]$ and $S[2\dots]$, which only yields (R_4, S_3).

The SkipJoin algorithm will only be efficient if the sequence of stab-queries can be performed efficiently. Obviously, such an ordered sequence of stab-queries can be seen as a single multi-stab-query (whose evaluation is interleaved with running the join algorithm). In Section 3, we introduce the stab-forest data structure which we will use to answer such multi-stab-queries efficiently, and in Section 4, we show how to efficiently query stab-forests.

▶ **Theorem 2.6.** *Let R and S be lists of events sorted in ascending start-time order. The algorithm* SkipJoin(R, S) *computes $R \bowtie S$ in worst-case $\mathcal{O}(M(R,S) + M(R,S) + |output|)$, in which $M(A,B)$ denotes the cost of either a multi-stab-query with $|A|$ timestamps on B or of fully traversing B, whichever is smaller.*

We notice that the focus of the skip-join algorithm is on supporting temporal joins of skewed datasets. The skip-join algorithm can easily be tuned to also support *windowed* temporal joins that only output events restricted to some window $\langle v, w\rangle$ in time, however: one simply starts with stab-queries to determine which events are active at v and stops whenever encountering events that start after w.

3 The Stab-Forest Data Structure

In the previous section, we introduced the SkipJoin algorithm. This algorithm requires an efficient manner to perform multi-stab-queries. To provide this, we introdce a novel index structure, the *stab-forest*. The stab-forest is a triple $\mathscr{S} = (\mathcal{E}, \mathcal{I}, i_{\mathsf{tail}})$ with $\mathcal{E}$ an *event-list* ordered lexicographically on (start, end)-times, $\mathcal{I}$ an *index* over the head of the event-list, and i_{tail} the *tail pointer* that holds the offset of the first event in $\mathcal{E}$ not yet part of $\mathcal{I}$. We call the part of the event-list starting at i_{tail} the *tail*. We define $|\mathscr{S}| = |\mathcal{E}|$. Next, we introduce stab-trees, which are at the basis of index $\mathcal{I}$. Then, we introduce the forest structure of $\mathcal{I}$, which stitches together the stab-trees used to index the event-list. Finally, we discuss the relevant parts of the physical layout we use for the index.

The stab-tree. A stab-tree is a binary tree that shares similarities with binary search trees and interval trees. First, we introduce the standard binary tree terminology and

notation. Let $\mathscr{S} = (\mathcal{E}, \mathcal{I}, i_{\mathsf{tail}})$ be a stab-forest and let T be a stab-tree indexing a portion of $\mathcal{E}$. We write $\mathsf{root}(T)$ to denote the *root node* of T. Let n be a node in T. By $\mathsf{left}(n)$ and $\mathsf{right}(n)$, we denote the *left* and *right* child of n. We call n a *leaf* if n does not have children ($\mathsf{left}(n) = \mathsf{right}(n) = \bot$). By $\mathsf{height}(T)$, we denote the *height* of the tree T, which we define as the number of nodes on the longest downward path from the root of the tree to a leaf node (the height of a tree without nodes is 0 and the height of a tree with a single node is 1).

Each node n has a *navigation key* and a *data key*, denoted by $\mathsf{nkey}(n)$ and $\mathsf{dkey}(n)$, respectively. The key $\mathsf{dkey}(n)$ is a timestamp present as the start-time of an event in the event-list. The *data pointer* $i_{\mathsf{data}}(n)$ holds the offset of the first event in $\mathcal{E}$ with start-time $\mathsf{dkey}(n)$. The key $\mathsf{nkey}(n)$ is the smallest timestamp such that no event in the event-list has a start-time in the range $[\mathsf{nkey}(n), \mathsf{dkey}(n))$.

Each node of a stab-tree represents events in $\mathcal{E}$ via the data key and data pointer: the node n represents those events $\mathrm{E} \in \mathcal{E}$ with $\mathrm{E}.\mathtt{start} = \mathsf{dkey}(n)$. The navigation key $\mathsf{nkey}(n)$ is derived from the event preceding $\mathcal{E}[i_{\mathsf{data}}(n)]$. Based on the definition of $\mathsf{nkey}(n)$, the only timestamp in $[\mathsf{nkey}(n), \mathsf{dkey}(n)]$ that has events $\mathrm{E} \in \mathcal{E}$ starting at it is $\mathsf{dkey}(n)$ – these are exactly the events represented by n. In Section 4, we will explain how the data and navigation keys are used while querying stab-forests. We define

$$\mathsf{min}(n) = \min\{\mathsf{nkey}(n') \mid n' \text{ in the subtree rooted at } n\};$$
$$\mathsf{max}(n) = \max\{\mathsf{dkey}(n') \mid n' \text{ in the subtree rooted at } n\}.$$

The *scope* of n is defined by $\mathsf{scope}(n) = [\mathsf{nkey}(n), \mathsf{dkey}(n)]$ and the *cover* by $\mathsf{cover}(n) = [\mathsf{min}(n), \mathsf{max}(n)]$. We say that timestamp t is in the *scope* of n if $t \in \mathsf{scope}(n)$ and is *covered* by n if $t \in \mathsf{cover}(n)$. For answering stab-queries, each node n is augmented with a *left-list*

$$\mathsf{LEFT}(n) = \{\langle v, w\rangle \in \mathcal{E} \mid \mathsf{min}(n) \leq v \leq \mathsf{dkey}(n) \wedge \mathsf{nkey}(n) \leq w \leq \mathsf{max}(n)\}.$$

Intuitively, the left-list $\mathsf{LEFT}(n)$ contains all events that are active in the *scope of* n, while starting and ending in the cover of n.

▶ **Example 3.1.** Consider the list of events $[\langle 0,3\rangle, \langle 0,11\rangle, \langle 1,2\rangle, \langle 2,3\rangle, \langle 4,5\rangle, \langle 5,5\rangle, \langle 5,6\rangle, \langle 6,8\rangle, \langle 7,7\rangle, \langle 7,9\rangle]$. This list is indexed by the stab-tree T visualized in Figure 2, *left*. We have $\mathsf{height}(T) = 3$. For the root node $r = \mathsf{root}(T)$, we have $\mathsf{nkey}(r) = 3$, $\mathsf{dkey}(r) = 4$, $\mathsf{min}(r) = 0$, $\mathsf{max}(r) = 7$, $\mathsf{scope}(r) = \langle 3,4\rangle$, $\mathsf{cover}(r) = \langle 0,7\rangle$, and $\mathsf{LEFT}(r) = [\langle 0,3\rangle, \langle 2,3\rangle, \langle 4,5\rangle]$.

Every node n in a stab-tree in $\mathcal{I}$ must satisfy the following four structural invariants:

(i) $\mathsf{nkey}(n) \leq \mathsf{dkey}(n)$;
(ii) if $\mathrm{E} \in \mathcal{E}$ with $\mathrm{E}.\mathtt{start} \in \mathsf{scope}(n)$, then $\mathrm{E}.\mathtt{start} = \mathsf{dkey}(n)$;
(iii) if $\mathsf{left}(n) \neq \bot$, then $\mathsf{nkey}(n) = 1 + \mathsf{max}(\mathsf{left}(n))$; and
(iv) if $\mathsf{right}(n) \neq \bot$, then $\mathsf{dkey}(n) = \mathsf{min}(\mathsf{right}(n)) - 1$.

To use the stab-trees for answering stab-queries efficiently, we also need to provide strong upper-bounds on the height of stab-trees. To do so, we put the following structural invariant on each stab-tree T used in $\mathcal{I}$:

(v) T has exactly $2^{\mathsf{height}(T)} - 1$ nodes.

Invariants i-iv imply the binary-search-tree property and Invariant v implies that each stab-tree is balanced and complete.

Let t be a timestamp. We say that a stab-tree T *covers* t if $\mathsf{root}(T)$ covers t. We say that an event $\mathrm{E} \in \mathcal{E}$ is *covered* by a stab-tree node or stab-tree if $\mathrm{E}.\mathtt{start}$ is covered by it. Given a stab-tree T and a timestamp t covered by T, Invariants i-iv guarantee that there exists exactly one node n in T that has t in its scope. Likewise, if $\mathrm{E} \in \mathcal{E}$ is covered by T, then there

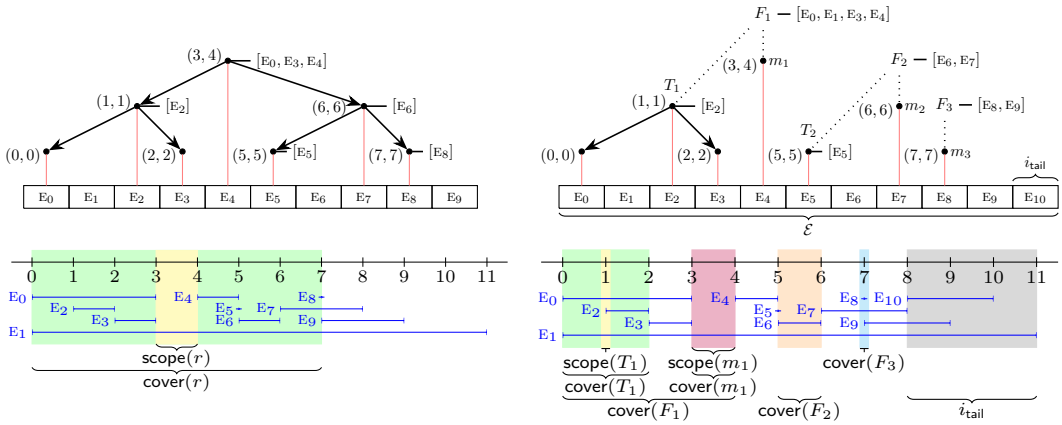

Figure 2 Examples of stab-trees and stab-forests. *Left*, a stab-tree indexing ten events. For each node n, the keys are visualized as $(\mathsf{nkey}(n), \mathsf{dkey}(n))$ and the left-list is only included if $\mathsf{LEFT}(n) \neq \emptyset$. *Right*, three forest-points over the same dataset, each with its own stab-tree, dummy stab-tree node, and max-list. Observe that the last forest-point's stab-tree is empty.

exists exactly one node n in T with $\mathsf{dkey}(n) = \mathtt{E.start}$. Given this node n, $i_{\mathsf{data}}(n)$ holds the offset of the first event in $\mathcal{E}$ with start-time $\mathtt{E.start}$. In this case, either $\mathtt{E}$ is part of exactly one left-list of a node n', ancestor of n, in T or $\mathtt{E.end} > \mathsf{max}(\mathsf{root}(T))$.

▶ **Example 3.2.** Consider the stab-tree T in Example 3.1, visualized in Figure 2, *left*. The timestamps $0, \ldots, 7$ are covered by T. More specifically, the timestamp 3 is covered by $\mathsf{root}(T)$, even though no event starts at 3. the timestamp 5 is covered by the leaf node with data key 5. No timestamp at-or-after 8 is covered by T, even though some events covered by T end at-or-after timestamp 8.

The stab-forest index. We require that all stab-trees are balanced and complete. Consequently, it is in most cases impossible to cover all events by a single stab-tree. Alternatively, we can cover consecutive parts of the event-list by a forest of stab-trees of decreasing heights. To use this forest of stab-trees for query answering, we need to maintain some metadata per stab-tree. This metadata is stored in *forest-points*.

Let $\mathscr{S} = (\mathcal{E}, \mathcal{I}, i_{\mathsf{tail}})$ be a stab-forest. A *forest-point* in $\mathcal{I}$ is a pair $F = (T, m)$, with T a stab-tree and m a dummy stab-tree node without left-list augmentation. We define $\mathsf{height}(F) = 1 + \mathsf{height}(T)$,

$$\mathsf{min}(F) = \begin{cases} \mathsf{nkey}(m) & \text{if } \mathsf{height}(T) = 0; \\ \mathsf{min}(\mathsf{root}(T)) & \text{if } \mathsf{height}(T) \neq 0, \end{cases}$$

and $\mathsf{max}(F) = \mathsf{dkey}(m)$. Each forest-point F is augmented with a *max-list*

$$\mathsf{MAX}(F) = \{\langle v, w\rangle \in \mathcal{E} \mid \mathsf{min}(m) \leq v \leq \mathsf{dkey}(m) \wedge \mathsf{nkey}(m) \leq w\}.$$

If we interpret $\mathsf{root}(T)$ as the left child of m, then, intuitively, the max-list $\mathsf{MAX}(F)$ contains all events that are active in the scope of m while starting in the cover of m (but not necessary ending in the cover of m). Hence, conceptually, forest-points and their max-lists can be seen as open-ended versions of stab-tree nodes and their left-lists: $\mathsf{MAX}(m)$ is a superset of a conceptual left-list of m with left child $\mathsf{root}(T)$ and with a yet undetermined right child. If a

sufficient number of events is appended to the event-list, then one is capable of constructing an appropriate right child for m, after which $\mathsf{MAX}(F)$ provides all the candidate events that might be part of $\mathsf{LEFT}(m)$.

▶ **Example 3.3.** Consider the event-list $\mathcal{E} = [\langle 0,3\rangle, \langle 0,11\rangle, \langle 1,2\rangle, \langle 2,3\rangle, \langle 4,5\rangle, \langle 5,5\rangle, \langle 5,6\rangle, \langle 6,8\rangle, \langle 7,7\rangle, \langle 7,9\rangle, \langle 8,10\rangle]$. This list is indexed by the stab-forest $\mathscr{S}$ visualized in Figure 2, *right*. The stab-forest $\mathscr{S}$ has three forest-points, $F_1 = (T_1, m_1)$, $F_2 = (T_2, m_2)$, and $F_3 = (T_3, m_3)$. Let $r_1 = \mathsf{root}(T_1)$. For the first forest-point, we have $\mathsf{nkey}(r_1) = \mathsf{dkey}(r_1) = 1$, $\mathsf{nkey}(m_1) = 3$, $\mathsf{dkey}(m_1) = 4$, and $\mathsf{MAX}(F_1) = [\langle 0,3\rangle, \langle 0,11\rangle, \langle 2,3\rangle, \langle 4,5\rangle]$. For the third forest-point, we observe that stab-tree T_3 is empty. Not all events are part of the index, as the tail-pointer points to the last event $\text{E}_{10} = \langle 8,10\rangle$. We observe that these forest-points cover the same set of events as the single stab-tree of Example 3.1. Due to the structural invariants we will place on stab-forests, the provided set of events does not yet contain sufficient information to merge these three forest-points to the stab-tree of Example 3.1, however.

We say that a forest-point *covers* timestamp t if either T or m covers t, we say that index $\mathcal{I}$ *covers* timestamp t if a forest-point $F \in \mathcal{I}$ covers t, and we say that the tail *covers* t if t is larger than any timestamp covered by $\mathcal{I}$. We say that an event $\text{E} \in \mathcal{E}$ is covered by a forest-point, index, or tail if $\text{E}.\texttt{start}$ is covered by it. We write $\mathsf{events}(F)$, $\mathsf{events}(T)$, $\mathsf{events}(n)$, and $\mathsf{events}(i_{\mathsf{tail}})$ to denote the set of events in the $\mathcal{E}$ covered by forest-point F, stab-tree T, stab-tree node n, or the tail, respectively.

To guarantee that $\mathcal{I}$ covers all events in $\mathcal{E}$ up to i_{tail} and that the index structure has strong upper-bounds on its size, we put the following structural invariants on the index:

(vi) the first forest-point in $\mathcal{I}$ covers the first event in $\mathcal{E}$;
(vii) all events in $\mathcal{E}$ at-or-after i_{tail} have the same start-time;
(viii) if $\mathcal{E} \neq \emptyset$, then i_{tail} is the offset of the first event $\text{E} \in \mathcal{E}$ not covered by $\mathcal{I}$;
(ix) if $(T, m) \in \mathcal{I}$ with $\mathsf{height}(T) \neq 0$, then $\mathsf{nkey}(m) = 1 + \mathsf{max}(\mathsf{root}(T))$; and
(x) if $F_2 \in \mathcal{I}$ directly follows $F_1 \in \mathcal{I}$, then $\mathsf{height}(F_1) > \mathsf{height}(F_2)$ and $\mathsf{min}(F_2) = 1 + \mathsf{max}(F_1)$.

We observe that the Invariants vi–x combined with Invariants i-iv guarantee that, for every event $\text{E} \in \mathcal{E}$ before offset i_{tail}, there exists exactly one forest-point $F \in \mathcal{I}$ that covers E. If $\text{E} \in \mathcal{E}$ and E is covered by $F = (T, m)$, then either E is part of exactly one left-list of a node in T or $\text{E} \in \mathsf{MAX}(F)$. Combining Invariant v with Invariant x allows us to upper bound the number and height of forest-points: if $\mathcal{E}$ has N distinct start-times and $F \in \mathcal{I}$ is the i-th forest-point in $\mathcal{I}$, $0 \leq i < |\mathcal{I}|$, then $|\mathcal{I}| \leq \lceil \log N \rceil$ and $\mathsf{height}(F) \leq \lceil \log N \rceil - (i+1)$.

Physical representation. The index structure will be used to support multi-stab-queries. To do so efficiently, we use specialized materializations of the left-lists and max-lists. Let n be a stab-tree node and let $F = (T, m)$ be a forest-point. The lists $\mathsf{LEFT}(n)$ and $\mathsf{MAX}(F)$ are each stored in two parts:

$$\mathsf{LEFT}_{\mathsf{n},\downarrow}(n) = \{\langle v, w\rangle \in \mathsf{LEFT}(n) \mid v < \mathsf{nkey}(n)\}; \quad \mathsf{LEFT}_{\mathsf{d},\downarrow}(n) = \mathsf{LEFT}(n) \setminus \mathsf{LEFT}_{\mathsf{n},\downarrow}(n);$$
$$\mathsf{MAX}_{\mathsf{n},\downarrow}(F) = \{\langle v, w\rangle \in \mathsf{MAX}(F) \mid v < \mathsf{nkey}(m)\}; \quad \mathsf{MAX}_{\mathsf{d},\downarrow}(F) = \mathsf{MAX}(F) \setminus \mathsf{MAX}_{\mathsf{n},\downarrow}(F).$$

In the above, each part is sorted on descending end-time order. We also maintain copies $\mathsf{LEFT}_{\mathsf{n},\uparrow}(n)$ and $\mathsf{MAX}_{\mathsf{n},\uparrow}(F)$ of $\mathsf{LEFT}_{\mathsf{n},\downarrow}(n)$ and $\mathsf{MAX}_{\mathsf{n},\downarrow}(F)$ that are sorted on ascending start-time order.

▶ **Proposition 3.4.** *Let L be a list of events. The stab-forest $\mathscr{S}$ indexing L can be stored in worst-case $\mathcal{O}(|L|)$ space.*

4 Query Evaluation on Stab-Forests

Previously, we discussed the structure of the stab-forest. Next, we show how the stab-forest supports answering multi-stab-queries efficiently. The definition of multi-stab-queries suggests a straightforward way to answer them: by simply executing multiple stab-queries and combining the results. This approach can become unnecessary inefficient if the dataset has events that appear in several of these stab-queries, as we have to explicitly eliminate duplicates.

▶ **Example 4.1.** Consider the stab-forest $\mathscr{S}$ of Example 3.3. We consider the multi-stab-query $\textsc{MultiStab}(\mathscr{S}, [0, 2, 5])$. We have $\textsc{Stab}(\mathscr{S}, 0) = \{\langle 0, 3\rangle, \langle 0, 11\rangle\}$, $\textsc{Stab}(\mathscr{S}, 2) = \{\langle 0, 3\rangle, \langle 0, 11\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle\}$, and $\textsc{Stab}(\mathscr{S}, 5) = \{\langle 0, 11\rangle, \langle 4, 5\rangle, \langle 5, 5\rangle\}$. By combining the results, we obtain $\textsc{MultiStab}(\mathscr{S}, [0, 2, 5]) = \{\langle 0, 3\rangle, \langle 0, 11\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle, \langle 4, 5\rangle, \langle 5, 5\rangle\}$. Observe that $\langle 0, 3\rangle$ appears in two stab-query results and $\langle 0, 11\rangle$ appears in all stab-query results.

To improve on this situation, we will present a direct multi-stab-query procedure that circumvents the need of deduplication. Let $\phi = [t_1, \ldots, t_{|\phi|}]$ be a sorted sequence of timestamps, let $R_i = \textsc{Stab}(\mathscr{S}, t_i)$, $1 \leq i \leq |\phi|$, let $S_1 = R_1$, and let $S_j = R_j \setminus R_{j-1}$, $2 \leq j \leq |\phi|$. Notice that $\textsc{MultiStab}(\mathscr{S}, \phi) = \bigcup_{1 \leq i \leq |\phi|} R_i = \bigcup_{1 \leq i \leq |\phi|} S_i$. By definition, the sets $S_1, \ldots, S_{|\phi|}$ are all pair-wise disjoint. Hence, we can answer multi-stab-queries efficiently if we can compute the sets S_i, $1 \leq i \leq |\phi|$, efficiently. Observe that an event is in S_j if and only if it is active at t_j but not at t_{j-1} (or any other timestamp in $[t_1, \ldots, t_{j-1}]$). First, we describe how to find parts of S_j stored in forest-points and the tail. Then, we describe how to find parts of S_j stored in a stab-tree, Finally, we provide necessary implementation details and analyze the complexity of the described multi-stab-query procedure. All details necessary to answer stab-queries efficiently can be derived from this multi-stab-query procedure.

Searching in forest-points and the tail. Let $\mathscr{S} = (\mathcal{E}, \mathcal{I}, i_{\mathsf{tail}})$. To simplify presentation, we assume that $\mathcal{E} \neq \emptyset$ and t_j is at-or-after the start of the first event in $\mathcal{E}$. (If these assumptions do not hold, we have $S_j = \emptyset$). We also assume that $t_{j-1} = -\infty$ if $t_j = t_1$. Under these assumptions, we need to search in the stab-forest to find all events in S_j. The first step is to identify if there exists a forest-point that covers t_j. If t_{j-1} is smaller than the start-time of any event in $\mathcal{E}$, then we start at the first forest-point in $\mathcal{I}$. Otherwise, we start at the forest-point that covers t_{j-1}. When visiting a forest-point $F = (T, m)$, we have one of the following four cases:

1. *F only covers events that start before t_j.* In this case, $\mathsf{max}(F) < t_j$ and events in $\mathsf{events}(T)$ start before-or-at $\mathsf{max}(F)$. Hence, $\mathsf{events}(F) \cap S_j = \mathsf{MAX}(F) \cap S_j$. We have $\mathsf{events}(F) \cap S_j \neq \emptyset$ only when $t_{j-1} < \mathsf{max}(F)$. In this case, we compute $\mathsf{events}(F) \cap S_j$ by traversing both $\mathsf{MAX}_{\mathsf{n},\downarrow}(F)$ and $\mathsf{MAX}_{\mathsf{d},\downarrow}(F)$ and stop when we find the first event that stops before t_j. During this traversal, we may encounter events already active at t_{j-1}; we skip over these events by not outputting them again. As an optimization, we notice that $\mathsf{MAX}(F) \cap S_j \subseteq \mathsf{MAX}_{\mathsf{d},\downarrow}(F)$ if $\mathsf{nkey}(m) \leq t_{j-1} < \mathsf{dkey}(m)$. In this case, we can skip traversing $\mathsf{MAX}_{\mathsf{n},\downarrow}(F)$. After processing this forest-point, proceed to the next forest-point.
2. *F represents events that start at t_j and t_j is covered by T.* In this case, $\mathsf{min}(F) \leq t_j < \mathsf{nkey}(m)$ and $\mathsf{events}(F) \cap S_j = (\mathsf{MAX}(F) \cup \mathsf{events}(T)) \cap S_j$. Due to $t_j < \mathsf{dkey}(m)$, we have $\mathsf{MAX}(F) \cap S_j = \mathsf{MAX}_{\mathsf{n},\uparrow}(F) \cap S_j$. We compute $\mathsf{MAX}_{\mathsf{n},\uparrow}(F) \cap S_j$ by traversing $\mathsf{MAX}_{\mathsf{n},\uparrow}(F)$ and stop when we find the first event that starts after t_j. Traversing $\mathsf{MAX}_{\mathsf{n},\uparrow}(F)$, we encounter events in $\mathsf{MAX}(F)$ that start before-or-at t_{j-1}, followed by those that start after t_{j-1} and before-or-at t_j, followed by those that start after t_j. Hence, to avoid

unnecessary deduplication, we traverse $\mathsf{MAX}_{\mathsf{n},\uparrow}(F)$ starting at the first event that starts after t_{j-1} (we detail how to do so later). Next, we search for all events in $\mathsf{events}(T) \cap S_j$. After searching in T, we have completed the computation of S_j.

3. *F represents events that start at t_j and t_j is not covered by T.* In this case, $\mathsf{nkey}(m) \leq t_j \leq \mathsf{dkey}(m)$. Events in $\mathsf{events}(T)$ start before $\mathsf{nkey}(m)$. Hence, $\mathsf{events}(F) \cap S_j = \mathsf{MAX}(F) \cap S_j$. We compute $\mathsf{events}(F) \cap S_j$ by traversing $\mathsf{MAX}_{\mathsf{n},\downarrow}(F)$ as in Case 1. If $t_j = \mathsf{dkey}(m)$, we also include $\mathsf{MAX}_{\mathsf{d},\downarrow}(F)$ entirely. We completed the computation of S_j.
4. *F only covers events that start after t_j.* In this case, $t_j < \mathsf{min}(F)$. We have $\mathsf{events}(F) \cap S_j = \emptyset$, and, as we process forest-points ordered on the events they cover, this forest-point will not be reached.

We have $\mathsf{events}(i_{\mathsf{tail}}) \cap S_j \neq \emptyset$ only if t_j is greater than any timestamp covered by $\mathcal{I}$. Let E be the event pointed at by i_{tail}. We have $\mathsf{events}(i_{\mathsf{tail}}) \cap S_j \neq \emptyset$ only if $t_{j-1} < \mathrm{E}.\mathtt{start} \leq t_j$. In this case, we find all events in the tail that are active at t_j by traversing $\mathcal{E}$ backwards starting at the end and stopping at either i_{tail} or at the first event that stops before t_j, whichever comes first. We notice that this traversal is a traversal on descending end-time order.

Searching in a stab-tree. The above only details how to process the max-lists of forest-points and the tail. To handle Case 2 above, we also need to describe how to compute $\mathsf{events}(T) \cap S_j$. Assume that $t_j \in \mathsf{cover}(F)$ and $t_j < \mathsf{nkey}(m)$. We perform a binary-search-tree search on T until we find the node n with $t \in \mathsf{scope}(n)$. For each node n' visited during this search, we have one of the following three cases:

5. *If $t < \mathsf{nkey}(n')$, then we need to continue the search in $\mathsf{left}(n')$.* We have $\mathsf{events}(n') \cap S_j = (\mathsf{events}(\mathsf{left}(n')) \cup \mathsf{LEFT}(n')) \cap S_j$. We compute $\mathsf{LEFT}(n') \cap S_j$ by traversing $\mathsf{LEFT}_{\mathsf{n},\uparrow}(n')$ and stop when we find the first event that starts after t_j. Traversing $\mathsf{LEFT}_{\mathsf{n},\uparrow}(n')$, we encounter events in $\mathsf{LEFT}(n')$ that start before-or-at t_{j-1}, followed by those that start after t_{j-1} and before-or-at t_j, followed by those that start after t_j. Hence, to avoid unnecessary deduplication, we traverse $\mathsf{LEFT}_{\mathsf{n},\uparrow}(n')$ starting at the first event that starts after t_{j-1} ((we detail how to do so later).
6. *If $\mathsf{nkey}(n') \leq t \leq \mathsf{dkey}(n')$, then we have found node n.* We have $\mathsf{events}(n') \cap S_j = \mathsf{LEFT}(n') \cap S_j$. We compute $\mathsf{events}(n') \cap S_j$ by traversing $\mathsf{LEFT}_{\mathsf{n},\downarrow}(n')$ and stop when we find the first event that stops before t_j. During this traversal, we may encounter events already active at t_{j-1}; we skip over these events by not outputting them again. If $t_j = \mathsf{dkey}(n')$, we also include $\mathsf{LEFT}_{\mathsf{d},\downarrow}(n')$ entirely. We completed the search in T.
7. *If $\mathsf{dkey}(n') < t$, then we need to continue the search in $\mathsf{right}(n')$.* We have $\mathsf{events}(n') \cap S_j = (\mathsf{events}(\mathsf{right}(n')) \cup \mathsf{LEFT}(n')) \cap S_j$. We have $\mathsf{LEFT}(n') \cap S_j \neq \emptyset$ only when $t_{j-1} < \mathsf{dkey}(n')$. In this case, we compute $\mathsf{LEFT}(n') \cap S_j$ by traversing both $\mathsf{LEFT}_{\mathsf{n},\downarrow}(n')$ and $\mathsf{LEFT}_{\mathsf{d},\downarrow}(n')$ and stop when we find the first event that stops before t_j. During this traversal, we may encounter events already active at t_{j-1}; we skip over these events by not outputting them again. As an optimization, we notice that $\mathsf{LEFT}(n') \cap S_j \subseteq \mathsf{LEFT}_{\mathsf{d},\downarrow}(n')$ if $\mathsf{nkey}(n') \leq t_{j-1} < \mathsf{dkey}(n')$. In this case, we can skip traversing $\mathsf{LEFT}_{\mathsf{n},\downarrow}(n')$.

Analysis of multi-stab-queries. To implement Cases 2 and 5 efficiently, we need to do some bookkeeping. For the relevant nodes n' and forest-point F on which Cases 2 and 5 applied while computing S_j, we need to keep track of the position of the first events in $\mathsf{LEFT}_{\mathsf{n},\uparrow}(n')$ and $\mathsf{MAX}_{\mathsf{n},\uparrow}(F)$ that start after t_j. In total, we need to keep track of at most $\lceil \log|\mathscr{S}| \rceil$ different positions.

▶ **Example 4.2.** We repeat the query $\textsc{MultiStab}(\mathscr{S}, [0, 2, 5])$ of Example 4.1 on the stab-forest $\mathscr{S}$ of Example 3.3. When stabbing with 0, we traverse $\mathsf{MAX}_{\mathsf{n},\uparrow}(F_1)$ and output the first two events $\langle 0, 3\rangle$ and $\langle 0, 11\rangle$. While searching in the stab-tree T_1, we do not find any further

events. Next, we stab with 2. When traversing $\mathsf{MAX}_{\mathsf{n},\uparrow}(F_1)$ we start at the third event, $\langle 2,3\rangle$, which we output. Next, we search in the stab-tree T_1, where we find $\langle 1,2\rangle$. During the stab with 5, we recognize that 5 is not covered by F_1. Hence, we traverse $\mathsf{MAX}_{\mathsf{n},\downarrow}(F_1)$ and $\mathsf{MAX}_{\mathsf{d},\downarrow}(F_1)$ to find any events that are still active at 5. The first event in $\mathsf{MAX}_{\mathsf{d},\downarrow}(F_1)$ is $\langle 4,5\rangle$, which we output. The first event in $\mathsf{MAX}_{\mathsf{n},\downarrow}(F_1)$ is $\langle 0,11\rangle$, which we skip over. We then search in F_2 to find and output $\langle 5,6\rangle$ and $\langle 5,5\rangle$.

We observe that the above multi-stab-query procedure will, in the worst case, read every event in the output of the multi-stab-query twice; once in a max-list or left-list that is sorted on ascending start-time order and once in a max-list or left-list that is sorted on descending end-time order. If the index $\mathcal{I}$ has N stab-tree nodes, then the approach to compute S_j will navigate through up to $\lceil \log N\rceil$ forest-points and stab-tree nodes. The multi-stab-query approach for computing S_j described above can easily be extended to also yield a pointer p_j to the first event in the event-list that starts after t_j, as used by the SkipJoin algorithm.

We can also compute S_j by traversing the event-list from the first event starting after t_{j-1} until the first event that starts after t_j. As long as this traversal of $\mathcal{E}$ performs at most $\lceil \log N\rceil$ memory operations, traversing $\mathcal{E}$ will be faster. To choose between these two methods to compute S_j, we can use a simple test. Let q be the position of the first event starting after t_{j-1}. Let c be the *threshold constant* representing the number of events one can read from the event-list in a single memory operation. To choose between the two approaches to compute S_j, we check if the event at position $q + c\lceil \log N\rceil$ does not exists or, otherwise, starts at-or-after t_{j+1}. With this approach, we need to change the processing of left-lists and max-lists to, additionally, skip over any events we found by traversing the event-list. Hence, with this change, the above process will read every event in the output at-most thrice.

▶ **Theorem 4.3.** *Let $\mathscr{S}$ be a stab-forest and let ϕ be a sorted sequence of timestamps. MultiStab$(\mathscr{S},\phi)$ can be answered in $\mathcal{O}(\min(|\phi|\log|\mathscr{S}|, |\phi| + |\mathscr{S}|) + |output|)$.*

5 Stab-Forest Maintenance

The stab-forest is designed to be a dynamic data structure to which events can be appended efficiently. Here, we show how to append events using the assumption that events are appended in lexicographical order on (start, end)-time. In Appendix A, we generalize the principles of the stab-forest to support less-restrictive semantics in equally efficient ways.

To support appending events that are ordered lexicographically on (start, end)-times, we start by describing an algorithm to put appended events in newly created forest-points. When appending a new event E' to stab-forest $\mathscr{S} = (\mathcal{E}, \mathcal{I}, i_{\mathsf{tail}})$, we distinguish the following cases:

1. If $|\mathcal{E}| = 0$, then append event E' to the end of $\mathcal{E}$ and set $i_{\mathsf{tail}} := 0$, the offset of E' in $\mathcal{E}$.
2. Else, if $\mathcal{E}[i_{\mathsf{tail}}]\texttt{.start} = \mathrm{E}'\texttt{.start}$, then append event E' to the end of $\mathcal{E}$.
3. In all other cases, $\mathcal{E}[i_{\mathsf{tail}}]\texttt{.start} < \mathrm{E}'\texttt{.start}$. Let L be the list of events in the event-list starting at i_{tail}. Create a fresh leaf node l and a fresh forest-point $F = (T, l)$ in $\mathcal{I}$ with T an empty tree. We set

$$\begin{aligned} \mathsf{dkey}(l) &= \mathrm{E}\texttt{.start}; & i_{\mathsf{data}}(l) &= i_{\mathsf{tail}}; \\ \mathsf{left}(l) &= \bot; & \mathsf{right}(l) &= \bot; \\ \mathsf{LEFT}(l) &= \emptyset; & \mathsf{MAX}(F) &= L. \end{aligned}$$

If $|\mathcal{I}| = \emptyset$, then set $\mathsf{nkey}(l) = \mathsf{dkey}(l)$. Else, set $\mathsf{nkey}(l) = \mathsf{max}(F') + 1$, with F' the last forest-point in $\mathcal{I}$. After constructing F, append F to the end of $\mathcal{I}$. Finally, append E' to $\mathcal{E}$ and set $i_{\mathsf{tail}} := |\mathcal{E}| - 1$, the offset of E' in $\mathcal{E}$.

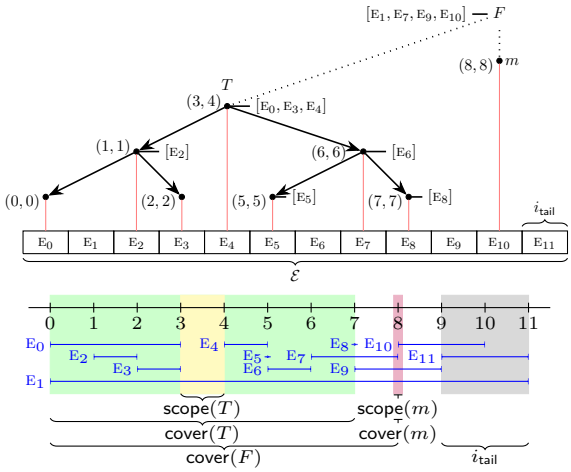

Figure 3 The stab-forest obtained after adding event $\langle 9, 11\rangle$ to the stab-forest of Figure 2, *right*.

Remember that the event-list is ordered lexicographically on (start, end)-times and that the current tail L only has events with a start-time $\mathrm{E}.\mathtt{start}$. Hence, by reversing L, we can directly construct $\mathsf{MAX}_{\mathsf{d},\downarrow}(F)$, and, in this case, we have $\mathsf{MAX}_{\mathsf{d},\downarrow}(F) = \mathsf{MAX}(F)$.

We observe that the above algorithm might invalidate Invariant x (Section 3), as a newly added forest-point can have the same height as the previous last forest-point in $\mathcal{I}$. To restore Invariant x, we will repeatedly merge the last two forest-points in $\mathcal{I}$ until they no longer have the same height. Let $F_1 = (T_1, m_1)$ and $F_2 = (T_2, m_2)$ be adjacent forest-points with $h = \mathsf{height}(F_1) = \mathsf{height}(F_2)$ and $\mathsf{min}(F_2) = \mathsf{max}(F_1) + 1$. We merge these forest-points into a single forest-point $F = (T, m_2)$ with

$$\begin{aligned}
\mathsf{root}(T) &= m_1;\\
\mathsf{left}(m_1) &= \mathsf{root}(T_1);\\
\mathsf{right}(m_1) &= \mathsf{root}(T_2);\\
\mathsf{LEFT}(m_1) &= \{\mathrm{E} \in \mathsf{MAX}(F_1) \mid \mathrm{E}.\mathtt{end} \leq \mathsf{max}(T_2)\};\\
\mathsf{MAX}(F) &= (\mathsf{MAX}(F_1) \setminus \mathsf{LEFT}(m_1)) \cup \mathsf{MAX}(F_2).
\end{aligned}$$

In the above, the necessary parts of $\mathsf{LEFT}(m_1)$ and $\mathsf{MAX}(F)$ can be constructed via straightforward merge-procedures on the parts of $\mathsf{MAX}(F_1)$ and $\mathsf{MAX}(F_2)$.

▶ **Example 5.1.** Consider the stab-forest of Example 3.3. We add the event $\langle 9, 11\rangle$, resulting in the stab-forest visualized in Figure 3.

One can show that the above forest-point merge maintains the Invariants i–iv, v, and ix (Section 3). Using this result, it is straightforward to prove that the described append method is sound. We conclude:

▶ **Theorem 5.2.** *Let L be a list of events ordered lexicographically on (start, end)-times. The structure obtained from starting with an empty stab-forest and appending each of the events in L in order is a stab-forest. This stab-forest is constructed in $\mathcal{O}(|L| \log|L|)$ and will use $\mathcal{O}(|L|)$ storage. Additional events can be added to the stab-forest in amortized $\mathcal{O}(\log|L|)$.*

Notice that the maintenance algorithm only operates on forest-points and does not change the stab-forests stored within them. Indeed, the constructed stab-trees are *static*, while merging of forest-points can be implemented via efficient array-merge-operations on their max-lists.

6 Empirical Evaluation

Finally, we provide an empirical study to showcase the practical performance of the stab-forest and the skip-join algorithm. We have implemented the stab-forest, the temporal query operations, and the temporal join algorithms presented in this paper in `C++`. Open-source code of the full implementation of the data structures, algorithms, and supporting tooling used can be found at `https://jhellings.nl/projects/skipjoin/`. In our implementation, we used 32bit unsigned integers to represent timestamps. The programs were compiled with the Microsoft C/C++ Compiler Version 19.13.26132 for x64, part of Visual Studio 2017, and run on a workstation with an Intel Core i5-4670 processor and 16GB of internal memory. In each experiment, the algorithms used write out their query results to a dynamic array (implemented by the standard `vector` data structure).

As a baseline for comparison, we implemented the forward-scan algorithm FwdScan, which is reported to be among the fastest internal-memory temporal join algorithms [6]. As our SkipJoin algorithm is based on FwdScan, our experiments not only showcase how SkipJoin performs compared to the state-of-the-art, but also allows for a detailed look at the benefits and costs of SkipJoin. To further examine the behavior of SkipJoin in detail, we tested with three variants; namely SkipJoin-$\mathcal{E}$ that uses the event-list exclusively for answering stab-queries, SkipJoin-$\mathcal{I}$ that uses the stab-forest index exclusively for answering stab-queries, and normal SkipJoin that uses a *threshold constant* $c = 16$ to choose between using the event-list and the stab-forest index.

In our experiments, we used two real-world datasets. The first real-world dataset we used is the *Airline On-Time Performance Data* (AOTPD) dataset [8], which contains flight-events (takeoff and duration) over a ten-year period. The second real-world dataset we used is the *Civil Unrest Event Data* (CUED) dataset [10], a set of civil unrest events in recent human history. The details of both datasets can be found in Figure 4, *left*. We also used a synthetically generated *gap dataset*. This dataset consists of two lists $\mathscr{R}$ and $\mathscr{S}$ that contain consecutive non-overlapping groups of G events (the *gap size*) that are placed alternatingly in either $\mathscr{R}$ or $\mathscr{S}$. Figure 4, *right*, visualizes such a dataset with twelve events grouped in groups of $G = 3$ events.

	AOTPD [8]	CUED [10]
Number of Events	$61,100,539$	$62,141$
Start date	July, 2007	February, 1946
End date	June, 2017	November, 2005
Minimal duration	0 minutes	0 days
Maximum duration	$1,350$ minutes	$18,407$ days

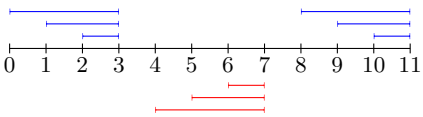

Figure 4 The datasets used in our evaluation. *Left*, statistics on the real-life datasets used. *Right*, gap datasets $\mathscr{R}$ and $\mathscr{S}$ with gap size 3.

Temporal joins on sparse datasets. First, we investigated the performance of temporal join algorithms in cases where only a few events are part of the join result, the situation for which our SkipJoin algorithm is designed. We used the temporal join algorithms to select a set of days from the AOTPD dataset. The temporal join algorithms select these specified days by joining the AOTPD dataset with a filter dataset that contains the to-be-selected days. As an additional point of reference, we compared the temporal join algorithms with a dedicated multi-window-query implementation that selects the same days. In this experiment, we selected the 7-th day from each of the first n out of 120 months. The results of our

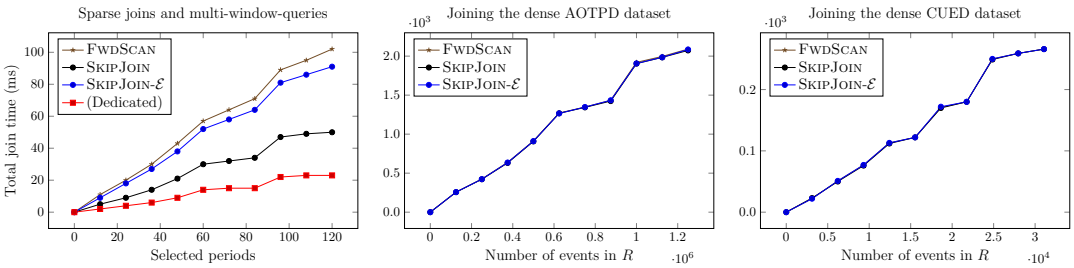

Figure 5 Temporal joins on sparse and dense datasets. On the *left*, we use temporal joins to select events from specific days in the AOTPD dataset. We compare these sparse temporal joins with a dedicated multi-window-query implementation to select the specific days. In the *middle* and on the *right*, we present the join performance on dense datasets, joining parts of the AOTPD dataset (*middle*) and parts of the CUED dataset (*right*). In these two cases, all three algorithms perform approximately the same.

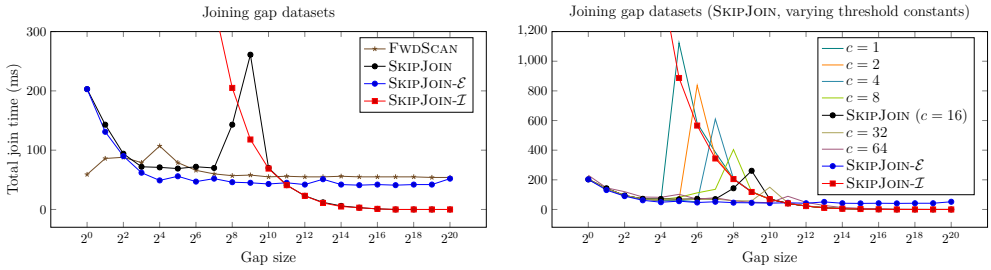

Figure 6 The behavior of SkipJoin: temporal join performance on sparse gap datasets, in which the gap-size determines the amount of data SkipJoin can skip over.

measurements can be found in Figure 5, *left*. As expected, SkipJoin benefits heavily from skipping over the non-relevant portions of the event-list. We also observe that the gain in performance comes from the usage of the stab-forest index, as SkipJoin-$\mathcal{E}$ only performs slightly better than FwdScan. Finally, we observe that the performance of SkipJoin comes close to our dedicated algorithm; showing that the performance of SkipJoin is even acceptable for implementing more specialized operators.

Temporal joins on dense datasets. Next, we investigated the performance of temporal join algorithms in cases where most events are part of the join result, e.g., in which almost every event in each dataset joins with an event in the other dataset. This is the situation for which the traditional FwdScan algorithm is designed. In this experiment, we used two datasets, namely the CUED dataset and a randomly selected fragment of $2,500,000$ events from the AOTPD dataset. For this experiment, we took a dataset, split that dataset into two halves R and S, shuffled R, and joined the first 0%, 10%, ..., 100% of the shuffled R with the entirety of S. The results of our measurements can be found in Figure 5, *middle* and *right*. We observe that in the setting of joining densely correlated datasets, there is no real difference between the SkipJoin-family of algorithms and the FwdScan algorithm, even though the SkipJoin-family of algorithms have higher complexity and overhead.

The behavior of SkipJoin. Third, we investigated the exact behavior of SkipJoin in situations where the algorithm is triggered to skip over data. To have full control over the amount of skipping possible, we used gap datasets with $64 \cdot 2^{20}$ events and a gap

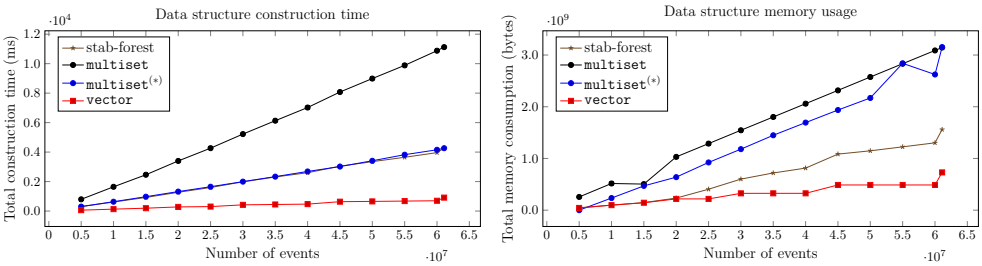

Figure 7 The costs of the stab-forest: construction time (*left*) and memory usage (*right*) of the stab-forest in comparison to various other data structures.

size of $G = 2^0, 2^1, \ldots, 2^{20}$. We ran the SkipJoin algorithm with threshold constants $c \in \{1, 2, 4, 8, 16, 32, 64\}$, and compared the SkipJoin algorithm with FwdScan, SkipJoin-$\mathcal{E}$, and SkipJoin-$\mathcal{I}$. The results of our measurements can be found in Figure 6. From the measurements, we conclude that the SkipJoin-family of algorithms is favorable as soon as they can skip over at least 4 events (the performance gain of skipping over a single event does not justify the overhead introduced by skipping). Skipping over events via the event-list, as SkipJoin-$\mathcal{E}$ does, provides a small improvement over FwdScan. Skipping over events via the index, as SkipJoin-$\mathcal{I}$ does, can provide order-of-magnitudes improvements over FwdScan, but only if sufficient events are skipped over. Due to this, it is important to use a threshold constant that is well-suited to the details of the underlying hardware. For our setting, the threshold constant $c = 16$, as used in SkipJoin, provides acceptable performance in all cases as it usually provides performance that is close to the fastest algorithm.

The costs of the stab-forest. In our final experiment, we looked at the costs of stab-forest maintenance. More specifically, we investigated the construction cost (by appending events one-by-one) and the memory consumption of a fully constructed stab-forest. We compared the construction of the stab-forest with the construction of three standard `C++` data structures:

1. `vector`. We use a `vector`, a bare bones dynamic array implementation, as a lower bound for representing the underlying event data without any indices.
2. `multiset`. We use a `multiset`, which is implemented as a self-balancing binary search tree. The `multiset` provides a lower bound on the cost of constructing and maintaining dynamic general-purpose interval data structures, as all dynamic general-purpose interval data structures are built using self-balancing binary search trees at their core [4, 9, 20].
3. `multiset`$^{(*)}$. Finally, we use a `multiset` in which each insert operation uses placement hints to allow the data structure to optimize for the append-only workload we provided. This `multiset` implementation provides a lower bound on the cost of constructing and maintaining dynamic general-purpose interval data structures that provide optimized append operations. We denote this usage of the `multiset` by `multiset`$^{(*)}$.

In this experiment, we used the AOTPD dataset. For each of the data structures, we measured the time it took to append the first N events from this dataset to the data structure ($N = 5 \cdot 10^6, 10 \cdot 10^6, \ldots, 60 \cdot 10^6$). The results of our measurements can be found in Figure 7. Unsurprisingly, appending data to the stab-forest is slower than appending to a `vector`, as the `vector` is the underlying representation of the event-list. The cost of appending to a stab-forest is on-par with the cost of appending to a `multiset`$^{(*)}$ binary search tree, showing that the stab-forest construction only incurs minimal overhead. Finally, appending to a fully dynamic `multiset` binary search tree, which provides the lower bound for dynamic general-purpose interval data structures, is much more costly than appending to

stab-forests. This supports our choice for designing stab-forests with append-only semantics. With respect to memory usage, we see that the stab-forest is much more compact than a binary search tree (even with all time-based augmentations), as it only requires a single stab-tree node per start-time, whereas the `multiset` uses a single search tree node per event.

7 Conclusion

We set out to develop high-performance internal-memory temporal join algorithms for dynamically generated heavily skewed data. Towards this goal, we proposed the stab-forest, the multi-stab-query, and the skip-join temporal join algorithm. In our evaluation, we showed that the skip-join algorithm is capable of significantly speeding up temporal joins of heavily skewed data. Our experiments also showed that the overhead of the skip-join algorithm when joining non-skewed data is insignificant, making our algorithm highly performant in all cases.

References

1. Pankaj K Agarwal and Jeff Erickson. *Geometric range searching and its relatives*, volume 223 of *Contemporary Mathematics*, pages 1–56. American Mathematical Society, 1999.
2. Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015. `doi:10.14778/2824032.2824076`.
3. Lars Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, and Jeffrey Scott Vitter. Scalable sweeping-based spatial join. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 570–581. Morgan Kaufmann Publishers Inc., 1998.
4. Lars Arge and Jeffrey Scott Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32(6):1488–1508, 2003. `doi:10.1137/S009753970240481X`.
5. Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, 2008.
6. Panagiotis Bouros and Nikos Mamoulis. A forward scan based plane sweep algorithm for parallel interval joins. *Proceedings of the VLDB Endowment*, 10(11):1346–1357, 2017. `doi:10.14778/3137628.3137644`.
7. Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using r-trees. *SIGMOD Record*, 22(2):237–246, 1993. `doi:10.1145/170036.170075`.
8. Bureau of Transportation Statistics, United States Department of Transportation. Airline on-time performance data, 2017. URL: `https://www.transtats.bts.gov/Tables.asp?DB_ID=120`.
9. Yi-Jen Chiang and Roberto Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE*, 80(9):1412–1434, 1992. `doi:10.1109/5.163409`.
10. Cline Center for Democracy. Speed project - civil unrest event data, 2012. URL: `https://clinecenter.illinois.edu/project/human-loop-event-data-projects/SPEED`.
11. Herbert Edelsbrunner. A new approach to rectangle intersections part I. *International Journal of Computer Mathematics*, 13(3–4):209–219, 1983. `doi:10.1080/00207168308803364`.
12. Herbert Edelsbrunner. A new approach to rectangle intersections part II. *International Journal of Computer Mathematics*, 13(3–4):221–229, 1983. `doi:10.1080/00207168308803365`.
13. Dengfeng Gao, Christian S. Jensen, Richard T. Snodgrass, and Michael D. Soo. Join operations in temporal databases. *The VLDB Journal*, 14(1):2–29, 2005. `doi:10.1007/s00778-003-0111-3`.
14. Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter Michael Fischer, Donald Kossmann, Franz Färber, and Norman May. Timeline index: A unified data structure for processing queries on temporal data in SAP HANA. In *Proceedings of the 2013 ACM*

SIGMOD International Conference on Management of Data, pages 1173–1184. ACM, 2013. doi:10.1145/2463676.2465293.

15 Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl. Managing intervals efficiently in object-relational databases. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 407–418. Morgan Kaufmann Publishers Inc., 2000.

16 Jiří Matoušek. Geometric range searching. *ACM Computing Surveys*, 26(4):422–461, 1994. doi:10.1145/197405.197408.

17 Edward M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985. doi:10.1137/0214021.

18 Dinesh P. Mehta and Sartaj Sahni. *Handbook Of Data Structures And Applications, Second Edition*. Chapman & Hall/CRC, 2017.

19 A. Montplaisir-Gonçalves, N. Ezzati-Jivan, F. Wininger, and M. R. Dagenais. State history tree: An incremental disk-based data structure for very large interval data. In *2013 International Conference on Social Computing*, pages 716–724. IEEE, 2013. doi:10.1109/SocialCom.2013.107.

20 Mark H. Overmars. *The Design of Dynamic Data Structures*. Springer, 1983.

21 Danila Piatov, Sven Helmer, and Anton Dignös. An interval join optimized for modern hardware. In *2016 IEEE 32nd International Conference on Data Engineering*, pages 1098–1109. IEEE, 2016. doi:10.1109/ICDE.2016.7498316.

22 Betty Salzberg and Vassilis J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(2):158–221, 1999. doi:10.1145/319806.319816.

23 Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Longman Publishing Co., Inc., 1990.

24 Peter Sanders. *Memory Hierarchies — Models and Lower Bounds*, pages 1–13. Springer, 2003. doi:10.1007/3-540-36574-5_1.

25 Arie Segev and Himawan Gunadhi. Event-join optimization in temporal relational databases. In *Proceedings of the 15th international conference on Very large data bases*, pages 205–215. Morgan Kaufmann Publishers Inc., 1989.

26 Donghui Zhang, Vassilis J. Tsotras, and Bernhard Seeger. Efficient temporal join processing using indices. In *Proceedings 18th International Conference on Data Engineering*, pages 103–113. IEEE, 2002. doi:10.1109/ICDE.2002.994701.

A Variants of Stab-Forests and their Maintenance

The stab-forest is designed to be a dynamic data structure to which events can be appended efficiently. In the main text, we showed how stab-forests support appending events using the assumption that events are appended in lexicographical order on (start, end)-time. Here, we the principles of the stab-forest to support two less-restrictive semantics in equally efficient ways.

Increasing start-time order semantics. The ordering on end-times is only used to assure that the tail is always lexicographically ordered on (start, end)-times: we only need to keep the tail ordered on end-times as all events in the tail have the same start-time. Hence, we can support appending events only ordered on start-time if we store the tail in a search tree. We then simply copy over the tail to the event-list whenever appending an event triggers the construction of a fresh forest-point.

One can also opt to not keep the tail sorted on end-times, but only enforce this ordering when creating a fresh forest-point. In such a design, queries can only access a *history* of the data that does not include the *current events* in the tail. This approach can also be used to support streams of events for which *watermarks* provide an after-the-fact guarantee on the ordering of past events [2].

Timestamp-based semantics. In streaming data processing and in versioned databases, the start-time and end-time of events are usually known when the event starts and ends, respectively [22]. For these applications, it is natural to append the start- and end-times when they happen, as separate operations. The stab-forest can be generalized to support these applications. In specific, we show how a stab-forest can support the following *timestamp-based semantics*. When an event starts, it is *appended* to the stab-forest by registering its *start-time*. On successful registration, the stab-forest returns an *event-handle* that can be used to update the event. When the event ends, one uses the event-handle to update the *end-time* of the event. We assume that all start- and end-times are appended and updated on increasing timestamps.

▶ **Example A.1.** Consider a versioned database. At t_1 record r gets created, at t_2 record r gets updated, and, finally, at t_3 record r gets deleted. At t_1, we append an event E_1 representing record r with start-time t_1 (and no end-time). At t_2, we update E_1 by setting the end-time t_2. We create a new event E_2 representing the updated record r with start-time t_2 (and no end-time). Finally, at t_3, we update E_2 by setting the end-time t_3.

Stab-forests with timestamp-based semantics are an obvious choice when adding skip-join style techniques to endpoint-based join algorithms, e.g., the algorithm of Piatov et al. [21].

To maintain all the invariants under the timestamp-based semantics, we need to make a few changes to the stab-forest structures. We represent open-ended events with start-time v and without an end-time by $\mathrm{E} = \langle v, \infty \rangle$. Each copy of such an open-ended event E in the event-list and in max-lists keeps a reference to the *event-handle*, which we describe in detail later. Each stab-tree node n, which represents events with start-time $\mathsf{dkey}(n)$, is augmented with an ∞-*pointer* $i_\infty(n)$ that holds the offset of the first open-ended event in $\mathcal{E}$ with start-time $\mathsf{dkey}(n)$ (if such an event exists). Each forest-point F is augmented with ∞-*pointers* $i_\infty(\mathsf{MAX}_{\mathsf{n},\downarrow}(F))$ and $i_\infty(\mathsf{MAX}_{\mathsf{d},\downarrow}(F))$ that hold the offsets of the last open-ended events in $\mathsf{MAX}_{\mathsf{n},\downarrow}(F)$ and $\mathsf{MAX}_{\mathsf{d},\downarrow}(F)$ (if such events exist). Finally, we use an ∞-*pointer* $i_{\infty\text{-}\mathsf{tail}}$ to hold the offset of the first open-ended event in the tail.

For every open-ended event $\mathrm{E} = \langle v, \infty \rangle$, we maintain an *event-handle*

$$\texttt{handle}(\mathrm{E}) = (i_{\mathcal{E}}, n, F, i_{\mathsf{MAX}_{\mathsf{n},\downarrow}(F)}, i_{\mathsf{MAX}_{\mathsf{d},\downarrow}(F)}, i_{\mathsf{MAX}_{\mathsf{n},\uparrow}(F)}),$$

in which $i_{\mathcal{E}}$ is the offset of the copy of E in $\mathcal{E}$, n is a reference to the stab-tree node with $\mathsf{dkey}(n) = v$ (if E is not in the tail), F is the reference to the forest-point that has E in its max-list (if E is not in the tail), and $i_{\mathsf{MAX}_{\mathsf{n},\downarrow}(F)}$, $i_{\mathsf{MAX}_{\mathsf{d},\downarrow}(F)}$, and $i_{\mathsf{MAX}_{\mathsf{n},\uparrow}(F)}$ are offsets of the copies of E in these max-lists (if such copies exist).

When a start-time v is appended to the stab-forest, the event $\mathrm{E} = \langle v, \infty \rangle$ and event-handle $\texttt{handle}(\mathrm{E})$ are constructed and E is appended to the tail. Appending an event to the tail can cause the construction and merging of forest-points. During this forest-point maintenance, the event-handles of open-ended events need to be kept up-to-date, which can be done in constant time per involved event. Finally, a reference to $\texttt{handle}(\mathrm{E})$ is returned.

When an end-time w for open-ended event E is updated in the stab-forest, one uses the reference to event-handle $\texttt{handle}(\mathrm{E}) = (i_{\mathcal{E}}, n, F, i_{\mathsf{MAX}_{\mathsf{n},\downarrow}(F)}, i_{\mathsf{MAX}_{\mathsf{d},\downarrow}(F)}, i_{\mathsf{MAX}_{\mathsf{n},\uparrow}(F)})$. First, we consider the steps necessary to update the end-time when E is not in the tail:

1. The copy of E in $\mathcal{E}$ is updated by setting $\mathcal{E}[i_{\mathcal{E}}]\texttt{.end} := w$. Then, to restore the lexicographic order on (start, end)-times in $\mathcal{E}$, the event at $\mathcal{E}[i_{\mathcal{E}}]$ is swapped with the event at $\mathcal{E}[i_\infty(n)]$. Next, the offsets $i_{\mathcal{E}}$ in the handles of the swapped events are updated. Finally, $i_\infty(n)$ is incremented by setting $i_\infty(n) := i_\infty(n) + 1$.
   This sequence of steps will update $\mathcal{E}$ and restore the lexicographic order in $\mathcal{E}$. Observe that the end-times arrive in order. Hence, the end-time w of E comes after all earlier-updated

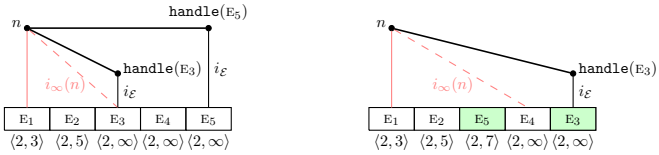

Figure 8 Updating an event by setting the end-time. On the *left*, the stab-forest before the update. On the *right*, the situation after setting $\text{E}_5\texttt{.end} := 7$. In this sketch, only details relevant to the update are included.

end-times and swapping E to offset $i_\infty(n)$ puts E directly after all other events with the same start-time and with a smaller end-time. All other open-ended events with the same start-time (including the event that got swapped with E) follow E and, hence, are still in a valid order. Consequently, incrementing $i_\infty(n)$ will assure that $i_\infty(n)$ once again points to the first open-ended event with start-time $\mathsf{dkey}(n)$ (if such an event exists).

2. If a copy of E is in a max-list $\mathsf{MAX}_{\mathsf{n},\uparrow}(F)$, then this copy of E is updated by setting $\mathsf{MAX}_{\mathsf{n},\uparrow}(F)[i_{\mathsf{MAX}_{\mathsf{n},\uparrow}(F)}] := w$. This update does not affect the start-time ordering of events in $\mathsf{MAX}_{\mathsf{n},\uparrow}(F)$, hence, no further change to $\mathsf{MAX}_{\mathsf{n},\uparrow}(F)$ is necessary.
3. If a copy of E is in a max-list $\mathsf{MAX}_{\mathsf{n},\downarrow}(F)$, then this copy of E is updated by setting $\mathsf{MAX}_{\mathsf{n},\downarrow}(F)[i_{\mathsf{MAX}_{\mathsf{n},\downarrow}(F)}] := w$. This update does affect the end-time ordering of events $\mathsf{MAX}_{\mathsf{n},\downarrow}(F)$. Similar to how the ordering of $\mathcal{E}$ is restored by a swap, the ordering in $\mathsf{MAX}_{\mathsf{n},\downarrow}(F)$ is restored by swapping value $\mathsf{MAX}_{\mathsf{n},\downarrow}(F)[i_{\mathsf{MAX}_{\mathsf{n},\downarrow}(F)}]$ and value $\mathsf{MAX}_{\mathsf{n},\downarrow}(F)[i_\infty(\mathsf{MAX}_{\mathsf{n},\downarrow}(F))]$, updating the relevant handles, and, finally, decrementing $i_\infty(\mathsf{MAX}_{\mathsf{n},\downarrow}(F))$ by setting $i_\infty(\mathsf{MAX}_{\mathsf{n},\downarrow}(F)) := i_\infty(\mathsf{MAX}_{\mathsf{n},\downarrow}(F)) - 1$.
4. Finally, if a copy of E is in a max-list $\mathsf{MAX}_{\mathsf{d},\downarrow}(F)$, then this copy is updated analogous to the previous case.

When E is in the tail, a swap of $\mathcal{E}[i_\mathcal{E}]$ and $\mathcal{E}[i_{\infty\text{-}\mathsf{tail}}]$, followed by incrementing $i_{\infty\text{-}\mathsf{tail}}$ suffices (once again similar to how the ordering of $\mathcal{E}$ is restored by a swap). After updating the end-time w for event E, the handle $\texttt{handle}(\text{E})$ can be destroyed.

▶ **Example A.2.** Consider a stab-tree node n with $\mathsf{dkey}(n) = 2$, pointing to an event-list with events $\langle 2, 3\rangle$, $\langle 2, 5\rangle$, $\langle 2, \infty\rangle$, $\langle 2, \infty\rangle$, and $\langle 2, \infty\rangle$. We wish to update the last event, $\text{E}_5 = \langle 2, \infty\rangle$, by setting its end-time to 7. In Figure 8, *left* and *right*, we sketched this setting before and after updating end-time 7.