The VLDB Journal (2021) 30:379–402
https://doi.org/10.1007/s00778-020-00650-5

REGULAR PAPER

Cache-efficient sweeping-based interval joins for extended Allen relation predicates

Danila Piatov[1] · **Sven Helmer**[2] · **Anton Dignös**[1] · **Fabio Persia**[1,3]

Received: 27 January 2020 / Revised: 2 September 2020 / Accepted: 23 November 2020 / Published online: 13 February 2021

Abstract
We develop a family of efficient plane-sweeping interval join algorithms for evaluating a wide range of interval predicates such as Allen's relationships and parameterized relationships. Our technique is based on a framework, components of which can be flexibly combined in different manners to support the required interval relation. In temporal databases, our algorithms can exploit a well-known and flexible access method, the Timeline Index, thus expanding the set of operations it supports even further. Additionally, employing a compact data structure, the gapless hash map, we utilize the CPU cache efficiently. In an experimental evaluation, we show that our approach is several times faster and scales better than state-of-the-art techniques, while being much better suited for real-time event processing.

Keywords Interval join · Allen relations · Plane sweep · Cache optimization

1 Introduction

Temporal data are found in many financial, business, and scientific applications running on top of database management systems (DBMSs), i.e., supporting these applications through efficient temporal operator implementations is crucial. For example, Kaufmann states that there are several temporal queries in the hundred most expensive queries executed on SAP ERP [23], many of which have to be implemented in the application layer, as the underlying infrastructure does not directly support the processing of temporal data. According to [23], customers of SAP desperately need (advanced) temporal operators for efficiently running queries pertaining to legal, compliance, and auditing processes.

Although the introduction of temporal operators into the SQL standard has started with SQL:2011 [29], current implementations are far from complete or lacking in performance. There is a renewed interest in temporal data processing, and researchers and developers are busy filling the gaps. An example is join operators involving temporal predicates: there are several recent publications on overlap interval joins [8,14,36]. However, this is not the only possible join predicate for matching (temporal) intervals. Allen defined a set of binary relations between intervals originally designed for reasoning about intervals and interval-based temporal descriptions of events [2]. These relations have been extended for event detection by parameterizing them [20]. Strictly speaking, all these relationships could be formulated in regular SQL `WHERE` clauses (see also the right-hand column of Table 1 for a formal definition of Allen's relations and extensions). The evaluation of these predicates using the implementation present in contemporary relational database management systems (RDBMSs) would be very inefficient, though, as a lot of inequality predicates are involved [26]. We also note that the predicates in the aforementioned overlap interval joins ([8,14,36]) only check for any form of overlap between intervals. Basically, they do not distinguish between many of the relationships defined by Allen and do not cover

✉ Sven Helmer
helmer@ifi.uzh.ch

Danila Piatov
danila.piatov@unibz.it

Anton Dignös
anton.dignoes@unibz.it

Fabio Persia
fabio.persia@unibz.it

1 Faculty of Computer Science, Free University of Bolzano, Piazza Domenicani 3, 39100 Bolzano, Italy

2 Department of Informatics, University of Zurich, Binzmühlestrasse 14, 8050 Zurich, Switzerland

3 Department of Information Engineering, Computer Science and Mathematics University of L'Aquila, Via Vetoio, 67100 L'Aquila, Italy

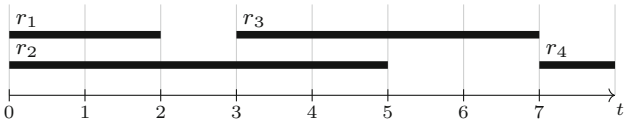


Fig. 1 Example temporal relation r

the BEFORE and MEETS relations at all. Additionally, many of the approaches so far lack parameterized versions, in which further range-based constraints can be formulated directly in the join predicate.

In Fig. 1 we see an example relation showing which employees (r_1, r_2, r_3, and r_4) were working on a certain project during which month. The tuple validity intervals (visualized as line segments) are as follows: tuple r_1 is valid from time 0 to time 2 (exclusive) with a starting time stamp $T_s = 0$ and an ending time stamp $T_e = 2$, tuple r_2 is valid on interval $[0, 5)$ with $T_s = 0$ and $T_e = 5$, and so on. With a simple overlap interval join we can merely detect that r_1 and r_2 worked together on the project for some time (as did r_2 and r_3). However, we may also be interested in who started working at the same time (r_1 and r_2), who started working after someone had already left (r_3 coming in after r_1 had left and r_4 starting after everyone else had left), or even who took over from someone else, i.e., the ending of one interval coincides with the beginning of another one (r_3 and r_4). For even more sophisticated queries, we may want to add thresholds: who worked together with someone else and then left the project within two months of the other person (r_3 and r_2).

Allen's relationships are not only used in temporal databases, but also in event detection systems for temporal pattern matching [20,27,28]. In this context, it is also important to be able to specify concrete time frames within which certain patterns are encountered, introducing the need for parameterized versions of Allen's relationships. For instance, Körber et al. use their TPStream framework for analyzing real-time traffic data [27,28], while we previously employed a language called ISEQL to specify events in a video surveillance context [20]. Event detection motivated us to develop an approach that is also applicable for event stream processing environments, meaning that our join operators are non-blocking and produce output tuples as early as logically possible, without necessarily waiting for the intervals to finish. Moreover, we demonstrate how these joins can be processed efficiently in temporal databases using a sweeping-based framework that is supported by cache-efficient data structures and by a Timeline Index—a flexible and general temporal index—supporting a wide range of temporal operators, used in a prototype implementation of SAP HANA [25]. We therefore extend the set of operations Timeline Index supports, increasing its usefulness even further.

In particular, we make the following contributions:

- We develop a family of plane-sweeping interval join algorithms that can evaluate a wide range of interval relationship predicates going even beyond Allen's relations.
- At the core of this framework sits one base algorithm, called interval-time stamp join, that can be parameterized using a set of iterators traversing a Timeline Index. This offers an elegant way of configuring and adapting the base algorithm for processing different interval join predicates, improving code maintainability.
- Additionally, our algorithm utilizes the CPU cache efficiently, relying on a compact hash map data structure for managing data during the processing of the join operation. Together with the index, in many cases we can achieve linear run time.
- In an experimental evaluation, we show that our approach is faster than state-of-the-art methods: an order of magnitude faster than a direct competitor and several orders of magnitude faster than an inequality join.

2 Related work

There is a renewed interest in employing Allen's interval relations in different areas, e.g., for describing complex temporal events in event detection frameworks [20,27,28] as well as for querying temporal relationships in knowledge graphs via SPARQL [11]. One reason is that it is more natural for humans to work with chunks of information, such as labeled intervals, rather than individual values [21].

2.1 Allen's interval relations joins

Leung and Muntz worked on efficiently implementing joins with predicates based on Allen's relations in the 1990s [30,31], and it turns out that their solution is still competitive today. In fact, they also apply a plane-sweeping strategy, but impose a total order on the tuples of a relation. Theoretically, there are four different orders tuples can be sorted in for this algorithm: T_s ascending, T_s descending, T_e ascending, and T_e descending. When joining two relations, they can be sorted in different orders independently of each other.

The actual algorithm is similar to a sort-merge join. A tuple is read from one of the relations (outer or inner) and placed into the corresponding set of active tuples for that relation. Each tuple in the set of the other relation is checked whether it matches the tuple that was just read. When a matching pair is found, it is transferred to the result set. While searching for matching tuples, the algorithm also performs a garbage collection, removing tuples that will no longer be able to find a matching partner. (Not all join predicates and sort orders allow for a garbage collection, though.) A heuristic, based on tuple distribution and garbage collection statistics, decides from which relation to read the next tuple. In a follow-

up paper, further strategies for parallelization and temporal query processing are discussed [32].

In contrast to our approach, in which we handle tuple starting and ending events separately (an idea also covered more generally in [15,33,39]), the algorithm of Leung and Muntz requires streams of whole tuples. A tuple is not complete until its ending endpoint T_e is encountered. This has a major impact for applications such as real-time event detection. Waiting for a tuple to finish can delay the whole joining process, as tuples following it in the sort order cannot be reported yet.

Chekol et al. claim to cover the complete set of Allen's relations in their join algorithm for intervals in the context of SPARQL, but the description for some relations is missing [11]. It seems they are using our algorithm from [36] as a basis. They are not able to handle parameterized versions and have to create different indexes for different relations, though.

There is also research on integrating Allen's predicate interval joins in a MapReduce framework [10,37]. However, these approaches focus on the effective distribution of the data over MapReduce workers rather than on effective joins.

2.2 Overlap interval joins

One of the earliest publications to look at performance issues of temporal joins is by Segev and Gunadhi [18,38], who compare different sort-merge and nested-loop implementations of their event join. They refined existing algorithms by applying an auxiliary access method called an append-only tree, assuming that temporal data are only appended to relations and never updated or deleted.

Some of the work on spatial joins can also be applied to interval joins. Arge et al. [4] used a sweeping-based interval join algorithm as a building block for a two-dimensional spatial rectangle join, but did not investigate it as a standalone interval join. It was picked up again by Gao et al. [16], who give a taxonomy of temporal join operators and provide a survey and an empirical study of a considerable number of non-index-based temporal join algorithms, such as variants of nested-loop, sort-merge, and partitioning-based methods.

The fastest partitioning join, the overlap interval partitioning (OIP) join, was developed by Dignös et al. [14]. The (temporal) domain is divided into equally-sized granules and adjacent granules can be combined to form containers of different sizes. Intervals are assigned to the smallest container that covers them and the join algorithm then matches intervals in overlapping containers.

The Timeline Index, introduced by Kaufmann et al. [25], and the supported temporal operators have also received renewed attention recently. Kaufmann et al. showed that a single index per temporal relation supports such operations as time travel, temporal joins, overlap interval joins and temporal aggregation on constant intervals (temporal grouping by instant). The work was done in the context of developing a prototype for a commercial temporal database and later extended to support bitemporal data [24].

In earlier work [36], we defined a simplified, but functionally equivalent version of the Timeline Index, called Endpoint Index (from now on we will use both terms interchangeably). We introduced a cache-optimized algorithm for the overlap interval join, based on the Endpoint Index, and showed that the Timeline Index is not only a universal index that supports many operations, but that it can also outperform the state-of-the-art specialized indexes (including [14]). The technique of sweeping-based algorithms has recently been applied to temporal aggregation as well [9,35], extending the set of operations supported by the Timeline Index even further.

The basic idea of Bouros and Mamoulis is to do a *forward scan* on the input collection to determine the join partners; hence, their algorithm is called forward scan (FS) join [8]. In contrast, our approach does a *backward scan* by traversing already encountered intervals, which have to be stored in a hash table. In FS, both input relations, $\boldsymbol{r}$ and $\boldsymbol{s}$, are sorted on the starting endpoint of each interval and then the algorithm sweeps through the endpoints of $\boldsymbol{r}$ and $\boldsymbol{s}$ in order. Every time in encounters an endpoint of an interval, it scans the other relation and joins the interval with all matching intervals. Bouros and Mamoulis introduce a couple of optimizations to improve the performance. First, consecutively swept intervals are *grouped* and processed in batch (this is called gFS). Second, the (temporal) domain is split into tiles and intervals starting in such a tile are stored in a *bucket* associated with this tile. While scanning for join partners for a tuple r, all intervals in buckets corresponding to tiles that are completely covered by the interval of r can be joined without further comparisons. Combined with the previous technique, this results in a variant called bgFS. Doing a forward scan or a backward scan has certain implications. Introducing their optimizations to FS, Bouros and Mamoulis showed that forward scanning is on a par with backward scanning, without the need of a data structure to store already encountered tuples. However, there is also a downside: forward scanning needs to have access to the complete relations to work, while backward scanning considers only already encountered endpoints, i.e., backward scanning can be utilized in a streaming context (for forward scanning this is not possible). We take a closer look at parallelization in Sect. 7.6.

2.3 Generic inequality joins

As we will see in Table 1, most of the interval joins can be broken down into inequality joins, which becomes very inefficient as soon as more than one inequality predicate is involved: Khayyat et al. [26] point out that these joins are handled via naive nested-loop joins in contemporary RDBMSs.

They develop a more efficient inequality join (IEJoin), which first sorts the relations according to the join attributes. For the sake of simplicity, we just consider two inequality predicates here, i.e., for every relation $\boldsymbol{r}$, we have two versions, $\boldsymbol{r^1}$ and $\boldsymbol{r^2}$ sorted by the two join attributes, which helps us to find the values satisfying an inequality predicates more efficiently. (The connections between tuples from $\boldsymbol{r^1}$ and $\boldsymbol{r^2}$ are made using a permutation array.) Some additional data structures, offset arrays and bit arrays, help the algorithm to take shortcuts, but essentially the basic join algorithm still consists of two nested loops, leading to a quadratic run-time complexity (albeit with a performance that is an order of magnitude better than a naive nested-loop join).

3 Background

Interval Data

We define a *temporal tuple* as a relational tuple containing two additional attributes, T_s and T_e, denoting the start and end of the half-open tuple validity interval $T = [T_s, T_e)$. We will use a period (.) to denote an attribute of a tuple, e.g., $r.T_s$ or $s.T$. The length of the tuple validity interval, $|r|$, is therefore $r.T_e - r.T_s$. We use the terms *interval* and *tuple* interchangeably. With r and s we denote the left-hand-side and the right-hand-side tuples in a join, respectively. We use integers for the time stamps to simplify the explanations. Our approach would work with any discrete time domain: we require a total order on the time stamps and a fixed granularity, i.e., given a time stamp we have to be able to unambiguously determine the following one.

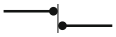
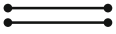
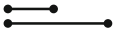
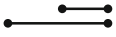
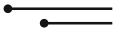
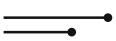
Interval Relations

As intervals accommodate the human perception of time-based patterns much better than individual values [21], intervals and their relationships are a well-known and widespread approach to handle temporal data [22]. Here we look at two different ways to define binary relationships between intervals: Allen's relations [2] and the Interval-based Surveillance Event Query Language (ISEQL) [6,20].

Allen designed his framework to support reasoning about intervals and it comprises thirteen relations in total. The seven basic *Allen's relations* are shown in the top half of Table 1. For example, interval r MEETS interval s when r finishes immediately before s starts. This is illustrated by the doodle in the table. We will use smaller versions of the doodles also in the text: (⇁↽). The first six relations in the table also have an inverse counterpart (hence thirteen relations). For example, relation 'r INVERSE MEETS s' describes s immediately finishing before r starts: (↼↽).

ISEQL originated in the context of complex event detection and covers a different set of requirements. The list of the five basic ISEQL relations is presented in the bottom half of Table 1; each of them has an inverse counterpart. Additionally, ISEQL relations are parameterized. The parameters control additional constraints and allow a much more fine-grained definition of join predicates. This is similar to the simple temporal problem (STP) formalism, which defines an interval that restricts the temporal distance between two events [3,13]. Let us consider the BEFORE ISEQL relation (⇁↼). It has one parameter δ, which controls the maximum allowed distance between the intervals (events). When $\delta = 0$, this relation is equivalent to the Allen's MEETS (⇁↽). When $\delta > 0$, it is a disjunction of Allen's MEETS and BEFORE, and the maximum allowed distance between the events is δ time points. Any ISEQL relation parameter can be *relaxed* (set to infinity), which removes the corresponding constraint.

Table 1 Allen's and ISEQL interval relations

Relation	Doodle	Formal definition
OVERLAPS		$r.T_s < s.T_s < r.T_e < s.T_e$
DURING		$s.T_s < r.T_s \wedge r.T_e < s.T_e$
BEFORE		$r.T_e < s.T_s$
MEETS		$r.T_e = s.T_s$
EQUALS		$r.T_s = s.T_s \wedge r.T_e = s.T_e$
STARTS		$r.T_s = s.T_s \wedge r.T_e < s.T_e$
FINISHES		$s.T_s < r.T_s \wedge r.T_e = s.T_e$
START PRECEDING		$r.T_s \leqslant s.T_s < r.T_e$ $s.T_s - r.T_s \leqslant \delta$
END FOLLOWING		$r.T_s < s.T_e \leqslant r.T_e$ $r.T_e - s.T_e \leqslant \varepsilon$
BEFORE		$r.T_e \leqslant s.T_s$ $s.T_s - r.T_e \leqslant \delta$
LEFT OVERLAP		$r.T_s \leqslant s.T_s < r.T_e \leqslant s.T_e$ $s.T_s - r.T_s \leqslant \delta$ $s.T_e - r.T_e \leqslant \varepsilon$
DURING		$s.T_s \leqslant r.T_s \wedge r.T_e \leqslant s.T_e$ $r.T_s - s.T_s \leqslant \delta$ $s.T_e - r.T_e \leqslant \varepsilon$

Joins on Interval Relations

For each binary interval relation (Allen's or ISEQL) we define a predicate $P(r, s)$ as its indicator function: its value is 'true' if the argument tuples satisfy the relation and 'false' otherwise. From now on we will use the terms 'predicate' and 'binary interval relation' interchangeably. We perceive ISEQL relation parameters, if not relaxed, as part of the definition of P (e.g., P_δ). We also define a *temporal relation* (not to be confused with binary relations described before) as a set of temporal tuples: $\boldsymbol{r} = \{r_1, r_2, \ldots, r_n\}$.

Let us take a generic relational predicate $P(r, s)$. We define the *P-join* of two relations $\boldsymbol{r}$ and $\boldsymbol{s}$ as an operator that

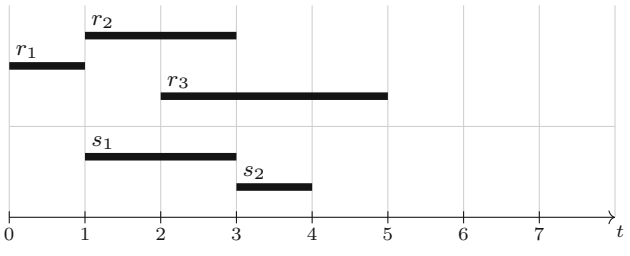


Fig. 2 Example relations $\boldsymbol{r}$ and $\boldsymbol{s}$

returns all pairs of $\boldsymbol{r}$ and $\boldsymbol{s}$ tuples that satisfy predicate P. We can express this in pseudo-SQL as '`SELECT * FROM` $\boldsymbol{r}$, $\boldsymbol{s}$ `WHERE` $P(r, s)$'. For example, we define the ISEQL LEFT OVERLAP JOIN as '`SELECT * FROM` $\boldsymbol{r}$, $\boldsymbol{s}$ `WHERE` r `LEFT OVERLAP` s'. If the predicate is parameterized, the join operator will also be parameterized.

Example 1 As an example (see Fig. 2), let us assume that we have two temporal relations: $\boldsymbol{r} = \{r_1 = [0, 1), r_2 = [1, 3), r_3 = [2, 5)\}$ and $\boldsymbol{s} = \{s_1 = [1, 3), s_2 = [3, 4)\}$.

Let us now take the ISEQL BEFORE JOIN ($\rightarrow$) with the parameter $\delta = 1$. Its result consists of two pairs $\langle r_1, s_1\rangle$ and $\langle r_2, s_2\rangle$, because only they satisfy this particular join predicate. If we relax the parameter (set δ to ∞), then an additional third pair $\langle r_1, s_2\rangle$ would be added to the result of the join.

By replacing the relational predicate by its formal definition (Table 1), we can implement an interval join in any relational database. However, such an implementation results in a relational join with inequality predicates, which is not efficiently supported by RDBMSs: they have to fall back on the nested-loop implementation in this case [26].

4 Formalizing our approach

We opt for a relational algebra representation to be able to make formal statements (e.g., proofs) about the different operators. Before fully formalizing our approach, we first introduce a map operator (χ) as an addition to the standard selection, projection, and join operators in traditional relational algebra. This operator is used for materializing values as described in [7]. We also introduce our new interval-time stamp join ($\bowtie$), allowing us to replace costly non-equi-join predicates with an operator that, as we show later, can be implemented much more efficiently.

Definition 1 The map operator $\chi_{a:e}(\boldsymbol{r})$ evaluates the expression e on each tuple of $\boldsymbol{r}$ and concatenates the result to the tuple as attribute a:

$$\chi_{a:e}(\boldsymbol{r}) = \{r \circ [a : e(r)] \mid r \in \boldsymbol{r}\}.$$

If the attribute a already exists in a tuple, we instead overwrite its value.

Definition 2 The interval-time stamp join $\boldsymbol{r} \bowtie \boldsymbol{s}$ matches the intervals in the tuples of relation $\boldsymbol{r}$ with the time stamps of the tuples in $\boldsymbol{s}$. It comes in two flavors, depending on the time stamp chosen for s, i.e., T_s or T_e. So, the interval-starting-time stamp join is defined as

$$\boldsymbol{r} \bowtie^{\theta}_{\text{start}} \boldsymbol{s} = \{r \times s \mid r \in \boldsymbol{r}, s \in \boldsymbol{s} : r.T_s \ \theta \ s.T_s < r.T_e\}$$

with $\theta \in \{<, \leq\}$, whereas the interval-ending-time stamp join boils down to

$$\boldsymbol{r} \bowtie^{\theta}_{\text{end}} \boldsymbol{s} = \{r \times s \mid r \in \boldsymbol{r}, s \in \boldsymbol{s} : r.T_s < s.T_e \ \theta \ r.T_e\}$$

with $\theta \in \{<, \leq\}$.

Now we are ready to formulate the joins on interval relations shown in Table 1 in relational algebra extended by our new join operator $\bowtie$. We first cover the non-parameterized versions (i.e., setting δ and ε to infinity) and then move on to the parameterized ones. Table 2 gives an overview of the relational algebra formulations. Proof for the correctness of our rewrites can be found in "Appendix A". Proof for the correctness of our rewrites can be found in "Appendix A" of our technical report.[1]

4.1 Non-parameterized joins

The non-parameterized joins include all the Allen's relations and the ISEQL joins with relaxed parameters. The EQUALS, STARTS, FINISHES, and MEETS predicates could be evaluated using a regular equi-join. Nevertheless, we formulate them via interval-time stamp joins, which are better suited to streaming environments and can be processed more quickly for low numbers of matching tuples. We present the joins roughly in the order of their complexity, i.e., how many other different operators we need to define them.

ISEQL Start Preceding Join

The START PRECEDING predicate ($\rightarrowtail$) joins two tuples r and s if they overlap and r does not start after s (we relax the parameter δ here). This can be expressed in our extended relational algebra in the following way:

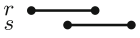
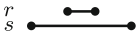
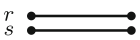
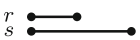
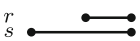
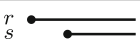
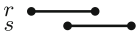
$$\boldsymbol{r} \bowtie^{\leq}_{\text{start}} \boldsymbol{s}.$$

ISEQL End Following Join

As before, we first consider the ISEQL END FOLLOWING predicate ($\rightarrowtail$) with a relaxed ε parameter, meaning that

[1] http://arxiv.org/abs/2008.12665.

Table 2 Mapping of interval relations to relational algebra

Relation	Doodle	Formal definition
OVERLAPS		$r.T_s < s.T_s < r.T_e < s.T_e$
DURING		$s.T_s < r.T_s \wedge r.T_e < s.T_e$
BEFORE		$r.T_e < s.T_s$
MEETS		$r.T_e = s.T_s$
EQUALS		$r.T_s = s.T_s \wedge r.T_e = s.T_e$
STARTS		$r.T_s = s.T_s \wedge r.T_e < s.T_e$
FINISHES		$s.T_s < r.T_s \wedge r.T_e = s.T_e$
START PRECEDING		$r.T_s \leqslant s.T_s < r.T_e$ $s.T_s - r.T_s \leqslant \delta$
END FOLLOWING		$r.T_s < s.T_e \leqslant r.T_e$ $r.T_e - s.T_e \leqslant \varepsilon$
BEFORE		$r.T_e \leqslant s.T_s$ $s.T_s - r.T_e \leqslant \delta$
LEFT OVERLAP		$r.T_s \leqslant s.T_s < r.T_e \leqslant s.T_e$ $s.T_s - r.T_s \leqslant \delta$ $s.T_e - r.T_e \leqslant \varepsilon$
DURING		$s.T_s \leqslant r.T_s \wedge r.T_e \leqslant s.T_e$ $r.T_s - s.T_s \leqslant \delta$ $s.T_e - r.T_e \leqslant \varepsilon$

tuples r and s should overlap and r is not allowed to end before s. In relational algebra this boils down to

$$\boldsymbol{r} \ltimes^{\leq}_{\text{end}} \boldsymbol{s}.$$

Overlap Join

If we look at the LEFT OVERLAP join (⬰), we notice that it looks very similar to a START PRECEDING join. The main difference is that it has one additional constraint: the $\boldsymbol{r}$ tuple has to end before the $\boldsymbol{s}$ tuple. Formulated in relational algebra this is equal to

$$\sigma_{\boldsymbol{r}.T_e \leq \boldsymbol{s}.T_e}(\boldsymbol{r} \ltimes^{\leq}_{\text{start}} \boldsymbol{s}).$$

For the RIGHT OVERLAP join (⬲), we could just swap the roles of $\boldsymbol{r}$ and $\boldsymbol{s}$, or we could use an END FOLLOWING join combined with a selection predicate stating that the $\boldsymbol{s}$ tuple has to start before the $\boldsymbol{r}$ tuple:

$$\sigma_{\boldsymbol{s}.T_s \leq \boldsymbol{r}.T_s}(\boldsymbol{r} \ltimes^{\leq}_{\text{end}} \boldsymbol{s}).$$

For the more strict Allen's LEFT OVERLAP and INVERSE OVERLAP joins we use '$<$' for the θ of the join and the selection predicate (or, alternatively, the parameterized version of the OVERLAP join, which is introduced later).

During Join

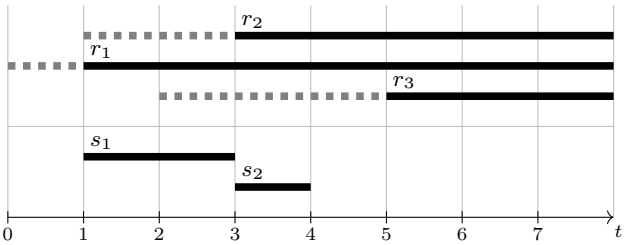


Fig. 3 Formulating Allen's BEFORE JOIN

For the DURING join (⬳), we have to swap the roles of $\boldsymbol{r}$ and $\boldsymbol{s}$. Formulated with the help of a START PRECEDING join, it becomes

$$\sigma_{\boldsymbol{r}.T_e \leq \boldsymbol{s}.T_e}(\boldsymbol{s} \ltimes^{\leq}_{\text{start}} \boldsymbol{r})$$

or, alternatively, with an END FOLLOWING join we get

$$\sigma_{\boldsymbol{r}.T_s \geq \boldsymbol{s}.T_s}(\boldsymbol{s} \ltimes^{\leq}_{\text{end}} \boldsymbol{r}).$$

A REVERSE DURING join maps more naturally to a START PRECEDING or END FOLLOWING join, i.e., we do not have to swap the roles of $\boldsymbol{r}$ and $\boldsymbol{s}$. For the Allen relation we use '$<$' for the θ of the join and the selection predicate (or the parameterized version of the DURING join).

Before Join

In the case of the BEFORE predicate (⬵), the tuples should not overlap at all. We achieve this by converting the ending events of $\boldsymbol{r}$ into starting ones and setting the ending events to infinity (see Fig. 3, the dashed lines are the original tuples and the solid lines the newly created ones). Formulated in relational algebra we get for the ISEQL version of BEFORE:

$$\chi_{T_e:\infty}(\chi_{T_s:T_e}(\boldsymbol{r})) \ltimes^{\leq}_{\text{start}} \boldsymbol{s}.$$

For the Allen relation we use $\theta =$'$<$' for the join.

Meets Join

For the MEETS predicate (⬴) each tuple of relation $\boldsymbol{r}$ should only be active for a short interval of length one when it ends. We achieve this by converting the end events of tuples in $\boldsymbol{r}$ into start events and adding a new end event that shifts the old end event by one. Expressed in relational algebra this looks as follows:

$$\chi_{T_e:T_e+1}(\chi_{T_s:T_e}(\boldsymbol{r})) \ltimes^{\leq}_{\text{start}} \boldsymbol{s}.$$

Equals Join

For the EQUALS predicate (⥱) we check that starting events match via an interval-timestamp join and then add a selection to check the ending events:

$$\sigma_{\boldsymbol{r}.T'_e = \boldsymbol{s}.T_e}(\chi_{T_e:T_s+1}(\chi_{T'_e:T_e}(\boldsymbol{r})) \ltimes^{\leq}_{\text{start}} \boldsymbol{s}).$$

Starts Join

For a STARTS predicate ($\rightrightarrows$) we first check that the starting events are the same, the ending events are again compared in a selection afterward. Although one of the predicates uses a comparison based on inequality, this happens in the selection, not the join:

$$\sigma_{\boldsymbol{r}.T_e<\boldsymbol{s}.T_e}(\chi_{T_e:T_s+1}(\chi_{T'_e:T_e}(\boldsymbol{r}))\ \ltimes^{\leq}_{\text{start}}\ \boldsymbol{s}).$$

Finishes Join

The FINISHES predicate ($\dashv\!\!=$) works similar to a STARTS predicate. We use an interval-ending-time stamp join and swap the roles of the starting and ending events (due to the different definition of the $\ltimes^{\leq}_{\text{end}}$ join, we also have to shift the time stamps by one):

$$\sigma_{\boldsymbol{s}.T_s<\boldsymbol{r}.T'_s}(\chi_{T_s:T_e-1}(\chi_{T'_s:T_s}(\boldsymbol{r}))\ \ltimes^{\leq}_{\text{end}}\ \boldsymbol{s}).$$

4.2 Parameterized joins

We now move on to the parameterized versions of the ISEQL join operators found in Table 1.

Start Preceding Join with δ

Defining a value for the parameter δ for the ISEQL START PRECEDING join ($=\!\!\rightarrow$) means that the tuple from $\boldsymbol{s}$ has to start between the start of the $\boldsymbol{r}$ tuple and within δ time units of the $\boldsymbol{r}$ tuple starting or the end of the $\boldsymbol{r}$ tuple (whichever happens first). Basically, we shorten the long tuples. Expressed in relational algebra, this becomes

$$\chi_{T_e:\min(T_e,T_s+\delta+1)}(\boldsymbol{r})\ \ltimes^{\leq}_{\text{start}}\ \boldsymbol{s}.$$

End Following Join with ε

For the parameterized END FOLLOWING ($\rightarrow\!\!=$) join we have to make sure that the $\boldsymbol{s}$ tuple ends within ε time units distance from the end of the $\boldsymbol{r}$ tuple (but after the start of the $\boldsymbol{r}$ tuple). The formal definition in relational algebra is

$$\chi_{T_s:\max(T_s,T_e-\varepsilon-1)}(\boldsymbol{r})\ \ltimes^{\leq}_{\text{end}}\ \boldsymbol{s}.$$

The Before Join with δ

For the parameterized BEFORE join we have to make sure that the $\boldsymbol{s}$ tuple starts within a time window of length δ after the $\boldsymbol{r}$ tuple ends:

$$\chi_{T_e:T_e+\delta+1}(\chi_{T_s:T_e}(\boldsymbol{r}))\ \ltimes^{\leq}_{\text{start}}\ \boldsymbol{s}$$

Overlap Join with δ and ε

Similar to the non-parameterized version we use the START PRECEDING join to define the parameterized LEFT OVERLAP join:

$$\sigma_{\boldsymbol{r}.T'_e\leq\boldsymbol{s}.T_e\leq\boldsymbol{r}.T'_e+\varepsilon}(\chi_{T_e:\min(T_e,T_s+\delta+1)}(\chi_{T'_e:T_e}(\boldsymbol{r}))\ \ltimes^{\leq}_{\text{start}}\ \boldsymbol{s}).$$

For the parameterized RIGHT OVERLAP join we can either swap the roles of $\boldsymbol{r}$ and $\boldsymbol{s}$ or use the END FOLLOWING join:

$$\sigma_{\boldsymbol{r}.T'_s-\delta\leq\boldsymbol{s}.T_s\leq\boldsymbol{r}.T'_s}(\chi_{T_s:\max(T_s,T_e-\varepsilon-1)}(\chi_{T'_s:T_s}(\boldsymbol{r}))\ \ltimes^{\leq}_{\text{end}}\ \boldsymbol{s}).$$

During Join with δ and ε

We can use a parameterized START PRECEDING or END FOLLOWING join as a building block for a parameterized DURING join, swapping the roles of $\boldsymbol{r}$ and $\boldsymbol{s}$. With a START PRECEDING join we get

$$\sigma_{\boldsymbol{s}.T'_e-\varepsilon\leq\boldsymbol{r}.T_e\leq\boldsymbol{s}.T'_e}(\chi_{T_e:\min(T_e,T_s+\delta+1)}(\chi_{T'_e:T_e}(\boldsymbol{s}))\ \ltimes^{\leq}_{\text{start}}\ \boldsymbol{r}),$$

whereas with an END FOLLOWING join it boils down to

$$\sigma_{\boldsymbol{s}.T'_s\leq\boldsymbol{r}.T_s\leq\boldsymbol{s}.T'_s+\delta}(\chi_{T_s:\max(T_s,T_e-\varepsilon-1)}(\chi_{T'_s:T_s}(\boldsymbol{s}))\ \ltimes^{\leq}_{\text{end}}\ \boldsymbol{r}).$$

5 Our framework

After introducing the interval joins formally, we now turn to their efficient implementation. We develop a framework to express the different interval joins with the help of just one core join algorithm. The framework also includes an index and several iterators for scanning through sets of intervals to increase the performance and flexibility.

5.1 The endpoint index

We can gain a lot of speedup by sweeping through the interval endpoints in chronological order using an Endpoint Index, which is a simplified version of the Timeline Index [25]. The idea of the *Endpoint Index* is that intervals, which can be seen as points in a two-dimensional space, are mapped onto one-dimensional *endpoints* or *events*.

Let $\mathbf{r}$ be an interval relation with tuples r_i, where $1 \leq i \leq n$. A tuple r_i in an Endpoint Index is represented by two events of the form $e = \langle timestamp, type, tuple\_id\rangle$, where *timestamp* is the T_s or T_e of the tuple, *type* is either a start or end flag, and *tuple_id* is the tuple identifier, i.e., the two events for a tuple r_i are $\langle r_i.T_s, \text{start}, i\rangle$ and $\langle r_i.T_e, \text{end}, i\rangle$. For instance, for $r_3.T = [3, 5)$, the two events are $\langle 3, \text{start}, 3\rangle$ and $\langle 5, \text{end}, 3\rangle$, which can be seen as 'at time 3 tuple 3 started' and 'at time 5 tuple 3 ended'.

Since events represent time stamps, we can impose a total order among events, where the order is according to *timestamp* and ties are broken by *type*. In our case of half-open intervals, the order of *type* values is: end $<$ start. Endpoints with equal time stamps and types but different tuple identifiers are considered equal. An Endpoint Index for interval relation $\mathbf{r}$ is built by first extracting the interval endpoints from the relation and then creating the ordered list of events $[e_1, e_2, \ldots, e_{2n}]$ sorted in ascending order. In case of

event detection, the endpoints (events) can be taken directly from the event stream and we do not even have to construct an index.

Consider exemplary interval relation $\boldsymbol{r}$ from Fig. 2. The Endpoint Index for it is [⟨0, start, 1⟩, ⟨1, end, 1⟩, ⟨1, start, 2⟩, ⟨2, start, 3⟩, ⟨3, end, 2⟩, ⟨5, end, 3⟩].

5.2 Endpoint iterators

Before continuing with the join algorithms, we introduce the concept of the Endpoint Iterator, upon which our family of algorithms is based. An *Endpoint Iterator* represents a cursor, which allows forward traversing a list of endpoints (e.g., an Endpoint Index). More formally, it is an abstract data type (an interface), which supports three operations:

- `getEndpoint`: returns the endpoint, which the iterator is currently pointing to (initially returns the first endpoint in the list);
- `moveToNextEndpoint`: advances the cursor to the next endpoint;
- `isFinished`: return `true` if the cursor is pointing beyond the last endpoint of the list, `false` otherwise.

More details on the implementation of Endpoint Iterators can be found in "Appendix A".

The basic implementation of the Endpoint Iterator is the *Index Iterator*, which provides an Endpoint Iterator interface to a physical Endpoint Index. Given an instance of the index, such an iterator traverses all Endpoint Index elements using the native method applicable to the Endpoint Index. In the text and in the algorithm descriptions we use the terms 'Endpoint Index' and 'Index Iterator' interchangeably, i.e., we create an Index Iterator for an Endpoint Index implicitly if needed.

There are also *wrapping* iterators. Such iterators do not have direct access to an Endpoint Index, but modify, filter and/or combine the output of one or several *source* Endpoint Iterators. In software design pattern terminology such iterators are called *decorators*. We conclude this subsection by introducing one such wrapping iterator. We introduce more of them later, as needed. The simplest wrapping Endpoint Iterator is the *Filtering Iterator*. It receives, upon construction, a source Endpoint Iterator and an endpoint type (start or end). It then traverses only endpoints having the specified type.

5.3 The core algorithm `JoinByS`

We are now ready to define the core algorithm, which forms the basis of all our joins. This algorithm receives the relations to be joined $\boldsymbol{r}$ and $\boldsymbol{s}$, Endpoint Iterators for them, a comparison predicate, and a callback function that will be called for each result pair. The algorithm performs the interleaved scan of the endpoint iterators. While doing so, it maintains the set of active $\boldsymbol{r}$ tuples. *Every* endpoint for relation $\boldsymbol{s}$ triggers the output—the Cartesian product of the corresponding tuple s and the set of active $\boldsymbol{r}$ tuples. The comparison predicate is used to define the order in which equal endpoints of different relations are handled ('equal' meaning endpoints having the same time stamp and type).

The pseudocode for the core algorithm *JoinByS* is presented in Algorithm 1. The algorithm starts by initializing an active $\boldsymbol{r}$ tuple set implemented via a map (an associative array) of tuple identifiers to tuples.

Algorithm 1: JoinByS($\boldsymbol{r}$, $\boldsymbol{s}$, itR, itS, comp, consumer)

```
   input : argument relations r and s, corresponding Endpoint
           Iterators itR and itS, endpoint comparison predicate
           comp ('<' or '≤'), function consumer(r, s) for result
           pairs
1  var activeR ← new Map of tuple identifiers to tuples
2  while not itR.isFinished and not itS.isFinished do
3      if comp(itR.getEndpoint, itS.getEndpoint) then
           // handle an r endpoint (maintain active r tuples)
4          tid ← itR.getEndpoint.tuple_id
5          if itR.getEndpoint = start then
6              r ← r[tid]                          // load the tuple
7              activeR.insert(tid, r)
8          else
9              activeR.remove(tid)
10         itR.moveToNextEndpoint
11     else
       // handle an s endpoint (trigger output)
12         s ← s[itS.getEndpoint.tuple_id]         // load tuple s
13         foreach r ∈ activeR do                  // with every active tuple r
14             consumer(r, s)                      // produce output pair ⟨r, s⟩
15         itS.moveToNextEndpoint
```

The main loop (line 2) and the main 'if' (line 3) implement the interleaved scan of the endpoint indices (like in a sort-merge join). The tricky part here is that instead of a hard-wired comparison operator ('$<$' or '$\leq$'), we use the function comp, that we pass as an argument to the algorithm. In case of the START PRECEDING JOIN, for instance, if both current endpoints of $\boldsymbol{r}$ and $\boldsymbol{s}$ are equal, we have to handle the $\boldsymbol{r}$ endpoint first (Sect. 4.1 on page 6), and thus we have to use the '$\leq$' predicate. In case of the END FOLLOWING JOIN, on the other hand, if both current endpoints of $\boldsymbol{r}$ and $\boldsymbol{s}$ are equal, we have to handle the $\boldsymbol{s}$ endpoint first (Sect. 4.1 on page 6), and thus we have to use the '$<$' predicate. Having the predicate as an argument of the algorithm allows us to choose the needed predicate upon using the algorithm, which prevents code duplication.[2]

[2] Note that the comparison function is not the same as the parameter θ of the interval-time stamp join. The comparison operator in JoinByS makes sure that the events are processed in the right order.

The rest of the algorithm consists of two parts. The first part (lines 4–10) handles an $\boldsymbol{r}$ endpoint and manages the active $\boldsymbol{r}$ tuple set. When a tuple starts, the algorithm loads it from the relation by the tuple identifier stored in the endpoint and puts the tuple in the map using the identifier as the key. When a tuple ends, the algorithm removes it from the active tuple map, again using the tuple identifier as the key.

The second part (lines 12–15) handles an $\boldsymbol{s}$ endpoint. It first loads the corresponding tuple s from the relation. Then it iterates through all elements in the active $\boldsymbol{r}$ tuple map. For every active r tuple the algorithm outputs the pair $\langle r, s\rangle$ by passing it into the *consumer* function, which is another function-type argument of the algorithm. In some cases, the consumer has to do additional work, e.g., evaluating a selection predicate. We call these consumers *filteringConsumers*. If they have access to the full tuple, they can check the predicate and immediately output the result. In a streaming environment, we do not have access to the end events immediately, which means that a filteringConsumer also needs to buffer data until these events become available, but as soon as this happens, the output can be generated.

6 Assembling the parts

We now show how to construct the different interval relations using our JoinByS operator and iterators. We start with the expressions from Sect. 4 that do not include map operators, followed by those that do.

6.1 Expressions without map operators

Start Preceding and End Following Joins

These two join predicates are the easiest to implement, as they can be mapped directly to the JoinByS operator. For the START PRECEDING JOIN ($\overset{\bullet}{=}$) we have to keep track of the active $\boldsymbol{r}$ tuples, and trigger the output by the *start* of an $\boldsymbol{s}$ tuple. If two tuples start at the same time, we have to handle the $\boldsymbol{r}$ tuple first. Therefore, we call the JoinByS function, passing to it only the starting $\boldsymbol{s}$ endpoints. This is achieved by using a Filtering Iterator (Section 5.2 on page 8). We also have to pass the '$\leq$' predicate as the comparison function. A START PRECEDING JOIN then boils down to a single call of JoinByS (see Algorithm 2).

Algorithm 2: StartPrecedingJoin($\boldsymbol{r}$, $\boldsymbol{s}$, itR, itS, consumer)

```
1 JoinByS(r, s, itR, FilteringIterator(itS, start), '≤', consumer)
```

The algorithm StartPrecedingJoin receives iterators to the Endpoint Indexes. When using this algorithm with Endpoint Indices, we simply wrap each index in an Index Iterator—an operation, which, as noted before, we consider implicit.

We define the algorithm for the END FOLLOWING JOIN ($\overset{\bullet}{=}$) similarly, but filter the *ending* endpoints of $\boldsymbol{s}$, and pass the '$<$' as the comparison function. The pseudocode of the EndFollowingJoin is presented in Algorithm 3.

Algorithm 3: EndFollowingJoin($\boldsymbol{r}$, $\boldsymbol{s}$, itR, itS, consumer)

```
1 JoinByS(r, s, itR, FilteringIterator(itS, end), '<', consumer)
```

Overlap Joins

The LEFT OVERLAP JOIN ($\overset{\bullet}{\frown}$) can be implemented using the StartPrecedingJoin algorithm with an additional constraint $r.T_e \leq s.T_e$. The pseudocode is shown in Algorithm 4. The RIGHT OVERLAP JOIN ($\overset{\bullet}{\smile}$) is implemented along similar lines using the EndFollowingJoin algorithm and the selection predicate $s.T_s \leq r.T_s$.

Algorithm 4: LeftOverlapJoin($\boldsymbol{r}$, $\boldsymbol{s}$, idxR, idxS, consumer)

```
1 filteringConsumer ← function (r, s)
2     if r.T_e ≤ s.T_e then consumer(r, s)
3 StartPrecedingJoin(r, s, idxR, idxS, filteringConsumer)
```

For the Allen versions of the overlap joins, we use strict versions of Algorithms 2 and 3, *StartPrecedingStrictJoin* and *EndFollowingStrictJoin*, which do not allow a tuple r to start with a tuple s or a tuple s to end with a tuple r, respectively. They are just simple variations: StartPrecedingStrictJoin merely replaces the '$\leq$' in Algorithm 2 with '$<$' and EndFollowingStrictJoin replaces the '$<$' in Algorithm 3 with '$\leq$'. Additionally, we change the '$\leq$' in the selection predicates in the filteringConsumer functions to '$<$'.

During Joins

Implementing the DURING join ($\overset{\bullet}{-}$) is similar to Algorithm 4: we just have to swap the arguments for $\boldsymbol{r}$ and $\boldsymbol{s}$ (alternatively, we could also use the END FOLLOWING variant). For the Allen version of DURING joins, we replace the StartPrecedingJoin, EndFollowingJoin, and selection predicates with their strict counterparts.

If we simply call an algorithm with swapped arguments, the elements of the result pairs appear in a different order, i.e., $\langle s, r\rangle$ instead of the expected $\langle r, s\rangle$. If this is an issue, we can swap them back using a lambda function as the consumer. Putting everything together, we get Algorithm 5.

Algorithm 5: DuringJoin($\boldsymbol{r}$, $\boldsymbol{s}$, idxR, idxS, consumer)

```
1 reversingConsumer ← function (s, r)
2     consumer(r, s)
3 filteringConsumer ← function (s, r)
4     if r.T_e ≤ s.T_e then reversingConsumer(s, r)
5 StartPrecedingJoin(s, r, idxS, idxR, filteringConsumer)
```

6.2 Expressions with map operators

In order to avoid physically changing tuple values or even the Endpoint Index, we apply the changes made by the map operators virtually with an iterator. While performing an interleaved scan of two Endpoint Indexes, instead of simply comparing the two endpoints r^e and s^e (as in $r^e < s^e$), we shift the time stamp of one of them when comparing: $r^e + \delta < s^e$. In this way the algorithm performs an interleaved scan of the indexes as if we had shifted all $\boldsymbol{r}$ tuples in time by $+\delta$.

During an interleaved scan, instead of forcing the iterators of the two Endpoint Indexes (for the relations $\boldsymbol{r}$ and $\boldsymbol{s}$) to move synchronously as in all the operators so far, now one of the iterators lags behind by a constant offset. This behavior can be easily incorporated into our framework by using a special Endpoint Iterator that shifts the time stamp of every endpoint it returns on-the-fly.

There is a second issue: the new *starting* endpoint often is actually a shifted *ending* endpoint or vice versa. Consequently, we have to change the endpoint type as well. With the help of our *Shifting Iterator*, we can shift time stamps and also change endpoint types. As input parameters a shifting iterator receives a source Endpoint Iterator, the shifting distance, and an endpoint type (start or end).

The final issue is separately shifting the starting and ending endpoints by different amounts. We solve this by having independent iterators for both starting and ending endpoints and merging them on the fly in an interleaved fashion. The input parameters of the *Merging Iterator* are two other iterators, the events of which it merges. See "Appendix A" for more details.

Before and Meets Joins

We are now ready to create a *GeneralBeforeJoin* (see Algorithm 6 and Fig. 4 for a schematic representation); we already handle the parameterized version here as well. This algorithm performs a virtual three-way sort-merge join of the two Endpoint Indexes. One pointer will traverse the Endpoint Index for relation $\boldsymbol{s}$, and two pointers will traverse the Endpoint Index for relation $\boldsymbol{r}$, all three pointers moving synchronously, but at different positions. This is why we had to (implicitly) create two Index Iterators for the same index (lines 4 and 6)—each of them represents a physical pointer to the same Endpoint Index; therefore we need two of them.

Algorithm 6: GeneralBeforeJoin($\boldsymbol{r}$, $\boldsymbol{s}$, idxR, idxS, β, δ, consumer)

```
1 StartPrecedingJoin(r,s,
2     MergingIterator(
          // T_e + β → T_s
3         ShiftingIterator(
4             FilteringIterator(idxR, end), β, start),
          // T_e + δ + 1 → T_e
5         ShiftingIterator(
6             FilteringIterator(idxR, end), δ + 1, end)),
7     IndexIterator(idxS),
8     consumer)
```

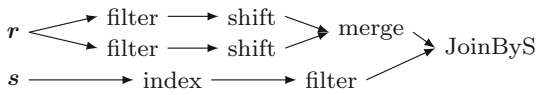


Fig. 4 Schematic representation of *GeneralBeforeJoin*

We express the Allen's BEFORE JOIN ($\rightarrow$) by substituting 1 and $+\infty$ for β and δ, respectively; Allen's MEETS JOIN ($\rightarrow_{\leftarrow}$) by substituting 0 and 0, respectively; and the ISEQL BEFORE JOIN by substituting 0 for β and only using the δ for the parameterized version. The parameter β distinguishes between the strict (Allen) and non-strict (ISEQL) versions of the operator.

Equals and Starts Joins

For the EQUALS JOIN ($\rightleftarrows$) we keep the original starting endpoints of $\boldsymbol{r}$ and use as ending endpoints the starting endpoints shifted by one and then execute a StartPrecedingJoin. This matches tuples from $\boldsymbol{r}$ and $\boldsymbol{s}$ with the same starting endpoints. We check that we have matching ending endpoints in the filteringConsumer function, which receives the actual tuples as input and thus has access to the time stamp attributes of the original tuples (see Algorithm 7) for the pseudocode.

Algorithm 7: EqualsJoin($\boldsymbol{r}$, $\boldsymbol{s}$, idxR, idxS, consumer)

```
1 filteringConsumer ← function (r, s)
2     if r.T_e = s.T_e then consumer(r, s)
3 StartPrecedingJoin(r,s,
4     MergingIterator(
          // keep the original r starting endpoints
5         FilteringIterator(idxR, start),
          // T_s + 1 → T_e
6         ShiftingIterator(
7             FilteringIterator(idxR, start), 1, end)),
8     IndexIterator(idxS),
9     filteringConsumer)
```

For a STARTS JOIN ($\rightarrow\!\!=$) we just have to change the predicate in the filteringConsumer function from '=' to '<'.

Finishes Join

For the tuples in $\boldsymbol{r}$ we turn the ending events into starting events and shift the ending events by one before joining them to the tuples in $\boldsymbol{s}$ via an EndFollowingJoin (Algorithm 3).

Finally, we check that the tuple from s started before the one from r. For the pseudocode of the FINISHES JOIN ($\dashv$), see Algorithm 8.

Algorithm 8: FinishesJoin(r, s, idxR, idxS, consumer)

```
filteringConsumer ← function (r, s)
    if s.T_s < r.T_s then consumer(r, s)
EndFollowingJoin(r,s,
    MergingIterator(
        // T_e − 1 → T_s
        ShiftingIterator(
            FilteringIterator(idxR, end), −1, start),
        // T_e → T_e
        ShiftingIterator(
            FilteringIterator(idxR, end), 0, end)),
    IndexIterator(idxS),
    filteringConsumer)
```

Parameterized Start Preceding Join

We now turn to the parameterized variant of the START PRECEDING JOIN ($\rightharpoonup$), which has the parameter δ constraining the maximum distance between tuple starting endpoints. The basic idea is to take the starting endpoints of relation r, shift them by $\delta+1$, change their type to ending endpoints, and add these virtual endpoints to the original endpoints of r. This way each r tuple will be represented by three endpoints: the original starting and ending endpoints and the virtual ending endpoint. Then the parameterless StartPrecedingJoin algorithm (Algorithm 2) is applied to both streams of r and s endpoints. When encountering the second ending endpoint in the merged iterator, it can simply be ignored when its corresponding tuple cannot be found in the active tuple set (see "Appendix A"). Algorithm 9 depicts the pseudocode.

Algorithm 9: PStartPrecedingJoin(r, s, idxR, idxS, δ, consumer)

```
StartPrecedingJoin(r,s,
    FirstEndIterator(
        MergingIterator(
            // keep the original r endpoints
            IndexIterator(idxR),
            // T_s + δ + 1 → T_e
            ShiftingIterator(
                FilteringIterator(idxR, start), δ + 1, end))),
        IndexIterator(idxS),
        consumer)
```

Parameterized End Following Join

A similar parameterized END FOLLOWING JOIN ($\rightharpoondown$) is more complicated. The problem here is that each r tuple will have to be represented by two starting endpoints. The algorithm must consider a tuple activated only if *both* starting endpoints (and no ending endpoint) have been encountered.

We achieve this by introducing an iterator, called *Second Start Iterator*, that stores the tuple identifiers of events for which we have only encountered one starting endpoint in a hash set (see "Appendix A"). Only the second starting endpoint of this tuple will return the starting event. The pseudocode for the parameterized END FOLLOWING JOIN is shown in Algorithm 10.

Algorithm 10: PEndFollowingJoin(r, s, idxR, idxS, ε, consumer)

```
EndFollowingJoin(r,s,
    SecondStartIterator(
        MergingIterator(
            // keep the original r endpoints
            IndexIterator(idxR),
            // T_e − ε − 1 → T_s
            ShiftingIterator(
                FilteringIterator(idxR, end), −ε − 1, start))),
        IndexIterator(idxS),
        consumer)
```

Parameterized Overlap Join

Now that we have an algorithm for the parameterized StartPrecedingJoin, we can define the parameterized LEFT OVERLAP JOIN ($\rightharpoonup$) by combining PStartPrecedingJoin with a filteringConsumer function, similarly to what we have done for the non-parameterized overlap join. Algorithm 11 shows the pseudocode. Alternatively, we can use a PEndFollowingJoin and then check the predicate for the starting endpoint of the s tuple in the filteringConsumer function.

Algorithm 11: PLeftOverlapJoin(r, s, idxR, idxS, δ, ε, consumer)

```
filteringConsumer ← function (r, s)
    if r.T_e ≤ s.T_e ≤ r.T_e + ε then consumer(r, s)
PStartPrecedingJoin(r, s, idxR, idxS, δ, filteringConsumer)
```

The RIGHT OVERLAP JOIN ($\leftharpoondown$) uses a PEndFollowingJoin with the corresponding predicate in the filteringConsumer function.

Parameterized During Join

The parameterized DURING JOIN ($\dashv\!\vdash$) looks similar to Algorithm 11, we apply changes along the lines of those shown in the paragraph for the non-parameterized DURING JOIN. (There is also an alternative version using an PEndFollowingJoin.)

6.3 Correctness of algorithms

Showing the correctness of our algorithms boils down to illustrating that we handle the map operators correctly and demonstrating the correctness of the StartPreceding and End-

Following joins, as our algorithms are either StartPreceding and EndFollowing joins or are built on top them.

Iterators and Map Operators

Here we show how to implement map operators with the help of iterators. Instead of materializing the result (e.g., on disk), we make the corresponding changes in a tuple as it passes through an iterator. If we still need a copy of the old event later on, we feed this event through another iterator and merge the two tuple streams using a merge iterator.

StartPreceding Join

We have to show that all tuples created by Algorithm 2 satisfy the predicate $r.T_s \leq s.T_s < r.T_e$. A Filter Iterator removes all the ending events from $\boldsymbol{s}$, so we only have to deal with starting events from $\boldsymbol{s}$ and with both types of events from $\boldsymbol{r}$. As comparison operator we use '$\leq$'. This determines the order in which events are dealt with.

First, let us look at the case that both upcoming events in itR and itS are starting events. If $r.T_s \leq s.T_s$, then r will be inserted into the active tuple set before s is processed, meaning that the (later) arrival of s will trigger the join with r. If $r.T_s > s.T_s$, then s will be processed first, not encountering r in the active tuple set, meaning that the two will not join.

Second, if the next event in $\boldsymbol{r}$ is an ending event and the next event in $\boldsymbol{s}$ a starting event, then the two events can never be equal. Even if they have the same time stamp, the ending endpoint of r will always be considered less than the starting endpoint of s. Therefore, if $r.T_e \leq s.T_s$, r will be removed first, so r and s will not join, and if $r.T_e > s.T_s$, s will still join with r.

So, in summary, all the tuples generated by Algorithm 2 satisfy the predicate $r.T_s \leq s.T_s < r.T_e$.

For a StrictStartPreceding join we run Algorithm 2 with the comparison operator '$<$', yielding output tuples that satisfy the predicate $r.T_s < s.T_s < r.T_e$. If both upcoming events in itR and itS are starting events, we get the correct behavior: $r.T_s < s.T_s$ will lead to a join, $r.T_s \geq s.Ts$ will not. If the r event is an ending event and the s event is a starting one, we also get the correct behavior: $r.T_e \leq s.T_s$ will not join the r and s tuple, $r.T_e > s.T_s$ will (the ending event of r is always less than the starting event of s).

EndFollowing Join

We show that all tuples created by Algorithm 3 satisfy the predicate $r.T_s < s.T_e \leq r.T_e$. This time a Filter Iterator removes all the starting events from $\boldsymbol{s}$, so we only have to deal with ending events from $\boldsymbol{s}$ and with both types of events from $\boldsymbol{r}$. The comparison operator used for the non-strict version is '$<$'.

First, assume that the next event in itR is a starting event and the next event in itS is an ending event. As an ending event takes precedence over a starting event, if $r.T_s = s.T_e$, the s event will come first. In turn this means that if $r.T_s < s.T_e$, r is added to the active set first, resulting in a join, and if $r.T_s \geq s.T_e$, s is processed first, meaning there is no join.

Second, we now look at the case that both events are ending events. Due to the comparison operator '$<$', the events are handled in the right way: if $r.T_e < s.T_e$, we remove r first, so there is no join, and if $r.T_e \geq s.T_e$ we handle s first, resulting in a join.

For a StrictEndFollowing join we run Algorithm 3 with '$\leq$' as comparison operator to obtain tuples that satisfy the predicate $r.T_s < s.T_e < r.T_e$. Let us first look at a starting event for $\boldsymbol{r}$ and an ending event for $\boldsymbol{s}$. As ending events are processed before starting events with the same time stamp, we get: if $r.T_s < s.T_e$, then r is added first, resulting in a join, and if $r.T_s \geq s.T_e$, then s is removed first, meaning there is no join. Finally, we investigate the case that both events are ending events: if $r.T_e \leq s.T_e$, then r is removed first, i.e., no join, and if $r.T_e > s.T_e$, then s is processed first, joining r and s.

7 Implementation considerations

In this section we look at techniques to implement our framework efficiently, in particular how to represent an active tuple set, utilizing contemporary hardware. We also investigate the overhead caused by our heavy use of abstractions (such as iterators).

7.1 Managing the active tuple set

For managing the active tuple set we need a data structure into which we can `insert` key-value pairs, `remove` them, and quickly enumerate (scan) one by one all the values contained in the data structure via the operation `getnext`. In our case, the keys are tuple identifiers and the values are the tuples themselves. The data structure of choice here is a map or associative array.

The most efficient implementation of a map optimizing the `insert` and `remove` operations is a hash table (with $O(1)$ time complexities for these operations). However, hash tables are not well suited for scanning. The *std::unordered_map* class in the C++ Standard Template Library and the *java.util.HashMap* in the Java Class Library, for instance, scan through all the buckets of a hash table, making the performance of a scan operation linear with respect to the capacity of the hash table and not to the actual amount of elements in it.

In order to achieve an $O(1)$ complexity for `getnext`, the elements in the hash table can be connected via a doubly linked list (see Fig. 5). The hash table stores pointers to elements, which in turn contain a key, a value, two pointers for the doubly linked list (*list prev* and *list next*) and a pointer for chaining elements of the same bucket for collision resolution (pointer *bucket next*). This approach is employed in the *java.util.LinkedHashMap* in the Java Class Library.

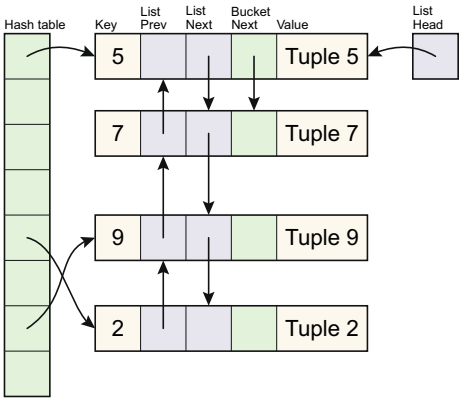


Fig. 5 Linked hash map

While this data structure offers a constant complexity for `getnext`, the execution times of different calls of `getnext` can vary widely in practice, depending on the memory footprint of the map. After a series of insertions and deletions the elements of the linked list become randomly scattered in memory, which has an impact on caching: sometimes the next list element is still in the cache (resulting in fast retrieval), sometimes it is not (resulting in slow random memory accesses). Additionally, the pointer structure make it hard for a prefetcher to determine where the next elements are located. However, for our approach it is crucial that `getnext` can be executed very efficiently, as it is typically called much more often than `insert` and `remove`. We will see in Sect. 7.4 how to implement a hash map more efficiently.

7.2 Lazy joining of the active tuple set

The fastest `getnext` operations are actually those that are not executed. We modify our algorithm to boost its performance by significantly reducing the number of `getnext` operations needed to generate the output.

We illustrate our point using the example setting in Fig. 6. Assume we have just encountered the left endpoint of s_1, which means that our algorithm now scans the tuple set $active^r$, which contains r_1 and r_2. After that we scan it again and again when encountering the left endpoints of s_2, s_3, and s_4. However, since no endpoints of **r** were encountered during that time, we scan the same version of $active^r$ four times. We can reduce this to one scan if we keep track of the tuples s_1, s_2, s_3, and s_4 in a (contiguous) buffer, delaying the scan until there is about to be a change in $active^r$.

To remedy this situation, we collect all consecutively encountered s tuples in a small buffer that fits into the L1 cache. Scanning the active tuple set when producing the output now requires only one traversal. Thanks to the design of our join algorithms we can incorporate this optimization into the whole framework by modifying JoinByS. The optimized version is shown in Algorithm 12. This technique has been introduced for overlap joins in [36], here we generalize it to the JoinByS algorithm. We recommend using a size for the buffer c that is smaller than the size of the L1d CPU cache (usually 32 Kilobytes) for this method to be effective.

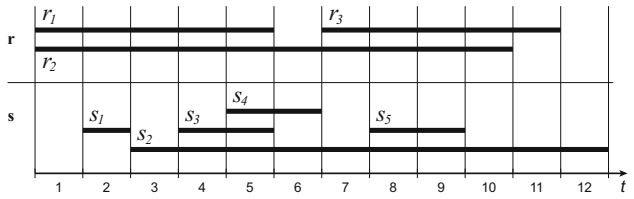


Fig. 6 Example interval relations

Algorithm 12: LazyJoinByS(r, s, itR, itS, comp, consumer)

input : argument relations r and s, corresponding Endpoint Iterators itR and itS, endpoint comparison predicate comp ('$<$' or '$\leq$'), function consumer(r, s) for result pairs

```
1  var activeR ← new Map of tuple identifiers to tuples
2  var buffer ← new array of capacity c
3  while not itR.isFinished and not itS.isFinished do
4      if comp(itR.getEndpoint, itS.getEndpoint) then
           // handle an r endpoint (maintain active r tuples)
5          tid ← itR.getEndpoint.tuple_id
6          if itR.getEndpoint = start then
7              r ← r[tid]                          // load the tuple
8              activeR.insert(tid, r)
9          else
10             activeR.remove(tid)
11         itR.moveToNextEndpoint
12     else
           // get sequence of s tuples uninterrupted by r events
13         repeat
14             s ← s[itS.getEndpoint.tuple_id]
15             buffer.insert(s)
16             itS.moveToNextEndpoint
17         until itS.isFinished or comp(itR.getEndpoint,
             itS.getEndpoint) or buffer.isFull
           // produce Cartesian product with active r tuples
18         foreach r ∈ activeR do                  // scan the active r tuples once
19             foreach s ∈ buffer do               // the inner loop, in L1 cache
20                 consumer(r, s)                  // produce output pair
21         buffer.clear
```

For the sake of simplicity, we only refer to the JoinByS algorithm in the following section. It can be replaced by the LazyJoinByS algorithm without any change in functionality.

7.3 Features of contemporary hardware

Before describing further optimizations, we briefly review mechanisms employed by contemporary hardware to decrease

main memory latency. This latency can have a huge impact, as fetching data from main memory may easily use up more than a hundred CPU cycles.

Mechanisms

Usually, there is a hierarchy of caches, with smaller, faster ones closer to CPU registers. Cache memory has a far lower latency than main memory, so a CPU first checks whether the requested data are already in one of the caches (starting with the L1 cache, working down the hierarchy). Not finding data in a cache is called a *cache miss*, and only in the case of cache misses on all levels, main memory is accessed. In practice an algorithm with a small memory footprint runs much quicker, because in the ideal case, when an algorithm's data (and code) fit into the cache, the main memory only has to be accessed once at the very beginning, loading the data (and code) into the cache.

Besides the size of a memory footprint, the access pattern also plays a crucial role, as contemporary hardware contains *prefetchers* that speculate on which blocks of memory will be needed next and preemptively load them into the cache. The easier the access pattern can be recognized by a prefetcher, the more effective it becomes. Sequential access is a pattern that can be picked up by prefetchers very easily, while random access effectively renders them useless.

Also, programs do not access physical memory directly, but through a virtual memory manager, i.e., virtual addresses have to be mapped to physical ones. Part of the mapping table is cached in a so-called *translation lookaside buffer* (TLB). As the size of the TLB is limited, a program with a high level of locality will run faster, as all lookups can be served by the TLB.

Out-of-order execution (also called *dynamic execution*) allows a CPU to deviate from the original order of the instructions and run them as the data they process become available. Clearly, this can only be done when the instructions are independent of each other and can be run concurrently without changing the program logic.

Finally, certain properties of DRAM (dynamic random access memory) chips also influence latency. Accessing memory using fast page or a similar mode means accessing data stored within the same page or bank without incurring the overhead of selecting it. This mechanism favors memory accesses with a high level of locality.

Performance Numbers

We provide some numbers to give an impression of the performance of currently used hardware. For contemporary processors, such as 'Core' and 'Xeon' by Intel[3], one random memory access within the L1 data (*L1d*) cache (32 KB per core) takes 4 CPU cycles. Within the L2 cache (256 KB per core) one random memory access takes 11–12 cycles. Within the L3 cache (3–45 MB) one random memory access takes 30–40 CPU cycles. Finally, one random physical RAM access takes around 70–100 ns (200–300 processor cycles). It follows that the performance gap between an L1 cache access and a main memory access is huge: two orders of magnitude.

[3] We use the cache and memory latencies obtained for the Sandy Bridge family of Intel CPUs using the SiSoftware Sandra benchmark, http://www.sisoftware.net/?d=qa&f=ben_mem_latency.

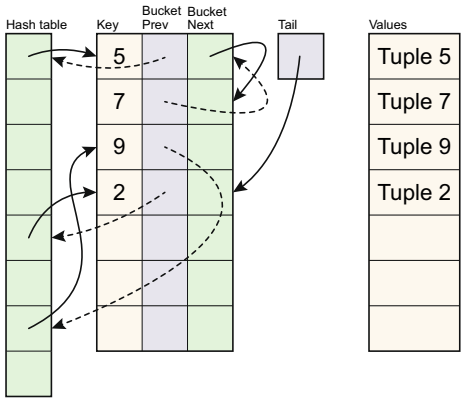


Fig. 7 Gapless hash map

7.4 Implementation of the active tuple set

As we will see later in an experimental evaluation, managing the active tuple set efficiently in terms of memory accesses is crucial for the performance of the join algorithm. Otherwise we run the risk of starving the CPU while processing a join. Our goals have to be to store the active tuple set as compactly as possible and to access it sequentially, allowing the hardware to get the data to the CPU in an efficient manner.

We store the elements of our hash map in a contiguous memory area. For the `insert` operation this means that we always append a new element at the end of the storage area. Removing the last element from the storage area is straightforward. If the element to be removed is not the last in the storage area, we swap it with the last element and then remove it. When doing so, we have to update all the references to the swapped elements. Scanning involves stepping through the contiguous storage area sequentially. We call our data structure a *gapless hash map* (see Fig. 7).

We also separate the tuples from the elements, storing them in a different contiguous memory area in corresponding locations. Assuming fixed-size records, all basic element operations (append and move) are mirrored for the corresponding tuples. This slightly increases the costs for insertions and removal of tuples. However, scanning the tuples is as fast as it can become, because we do not need to read any metadata, only tuple information.

The hash table stores pointers to elements, which contain a key, a pointer for chaining elements of the same bucket when resolving collisions (pointer *bucket next*, solid arrows),

and a pointer *bucket prev* to a hash table entry or an element (whichever holds the forward pointer to this element, dashed arrows). The latter is used for updating the reference to an element when changing the element position. The main difference to the random memory access of a linked hash map (Fig. 5) is the allocation of all elements in a contiguous memory area, allowing for fast sequential memory access when enumerating the values.

Example 2 Assume we want to remove tuple 7 from the structure depicted in Fig.7. First of all, the bucket-next pointer of the element with key 5 is set to NULL. Next, the last element in the storage area (tuple 2) is moved to the position of the element with key 7. Following the bucket-prev pointer of the just moved element we find the reference to the element in the hash table and update it. Finally, the variable *tail* is decremented to point to the element with key 9.

7.5 Overhead for abstractions

All the abstractions we use (iterators, predicates passed as function arguments, and lambda functions) allow us to express all joins by means of a single function, which is extremely practical due to the huge simplification of implementation and subsequent maintenance of the code. In this section we explain why the impact of this architecture on the performance is minimal for C++ and not significant for Java.

We compare our implementation empirically to a manual rewrite without abstractions of a selected join algorithm. Here, we show the results for our most complicated implementation, Algorithm 6. We compare its performance to a version that was fully inlined manually into a single leaf function. We did so for C++ and also for Java. We then launched each one of the four versions separately using the synthetic dataset of 10^6 tuples with an average number of active tuples equal to 10 (see Sect. 9.1 for the dataset). Each version was executing the join several times sequentially to allow the JVM to perform all necessary optimizations. The results are shown in Fig. 8. We see that the C++ version is several times faster than the Java version. Moreover, we see that the C++ compiler was able to optimize our abstracted code so well that its performance is indistinguishable from the manually optimized version. The situation with Java is more complicated, in the end the manually optimized version was ∼10% faster.

C++

This language was designed to support penalty-free abstractions. Not all abstractions in C++ are penalty free, though. We first implemented the family of Endpoint Iterators as a hierarchy of virtual classes and found that the compilers we used (GCC and Clang) were not able to inline virtual method calls (even though they had all the required information to do so). We then rewrote the code using templates and functors, each iterator becoming a non-virtual class, passed into the join algorithm as template argument. The comparator used by the core algorithm was a functor `std::less` or `std::less_equals`. The consumers were defined as C++11 lambda-functions, also passed as a template parameter. This time both compilers were able to inline all method calls and generate very optimized code with all variables (including iterator fields) kept in CPU registers.

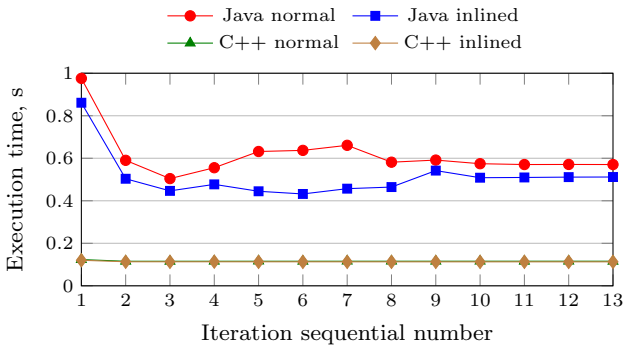


Fig. 8 Overhead of the abstractions used in the algorithms

Java

We face a different situation with Java, as the optimization is not performed by the compiler, but the Java virtual machine (JVM) during runtime. The JVM (in particular, the standard Oracle HotSpot implementation) compiles, optimizes and recompiles the code while executing it. It can potentially apply a wider range of optimizations (e.g., speculative optimization) than a C++ compiler can, as it actively learns about the actual workload, but in the case of Java we have limited control over this process. As we show in Fig. 8, Java does in fact optimize the code with abstractions. Not as well as C++, but the performance difference is very small compared to a manually rewritten join.

7.6 Parallel execution

While parallelization is not a main focus of this paper, we know how to parallelize our scheme and have implemented a parallel version of our earlier EBI-Join operator [36]. We give a brief description here: the tuples in both input relations, $\boldsymbol{r}$ and $\boldsymbol{s}$, are sorted by their starting time and then partitioned in a round-robin fashion, i.e., the ith tuple of a relation is assigned to partition ($i \mod k$) of that relation, where k is the number of partitions. By assigning close neighbors to different partitions, we lower the size of the active tuple sets, which is a crucial parameter for the performance of our algorithm. We then do a pairwise join between all partitions of $\boldsymbol{r}$ with all partitions of $\boldsymbol{s}$. As all partitions are disjoint, the joins can run in parallel independently of each other. A downside of this approach is that we need k^2 processes. Nevertheless, we achieved an average speedup of 2.7,

4.3, and 5.3 for $k = 2$, 3, and 4, respectively, on a machine with two CPUs (eight cores each). Bouros and Mamoulis utilize domain-based rather than hash-based partitioning, which gives FS a slight edge over EBI-Join during parallel execution [8]. However, as also shown in [8], domain-based partitioning can also be applied to EBI-Join, resulting in comparable performance.

Furthermore, one major difference between JoinByS and EBI-Join is that JoinByS maintains only one active tuple set (for $\boldsymbol{r}$), whereas EBI-Join maintains two (one for $\boldsymbol{r}$ and one for $\boldsymbol{s}$). So, in order to keep the active tuple set small, for JoinByS we only need to partition $\boldsymbol{r}$, resulting in one process for each of the k partitions, while still being able to do this on the fly. The tuples in $\boldsymbol{s}$ are then fed to each of these processes.

8 Theoretical analysis

One-dimensional Overlap

Our approach is related to finding all the intersecting line segments, or intervals, given a set of n segments in one-dimensional space. The optimal (plane-sweeping) algorithm for doing so has complexity $O(n \log n + k)$, where k is the number of intersecting segments [12]. In the worst case, when we have a large number of intersecting segments, the complexity becomes $O(n \log n + n^2)$. In this case, the run time of the algorithm is dominated by the output cardinality.

Each segment s_i is split up into a left starting event $\langle l_i, \text{start}\rangle$ and a right ending event $\langle r_i, \text{end}\rangle$. Afterward the events of all segments are sorted, which takes $O(n \log n)$ time. We then traverse the sorted list of events. When encountering a left endpoint, we insert it into a data structure D, which keeps track of the currently active segments. When encountering a right endpoint, we remove it from D and join it with all the segments currently stored in D. If we use a balanced search tree for D (e.g., a red-black tree), then inserting and removing an endpoint will cost us $O(\log n)$. As we have $2n$ endpoints, we arrive at a total of $O(2n \log 2n) = O(n \log n)$. Generating all the output will take $O(k)$. If we use a hash table, insertion and removal of endpoints can be done in $O(1)$, for a total of $O(n)$. As long as we make sure that the entries in the hash table are linked or packed compactly (as in our gapless hash map), this will have an overall complexity of $O(k)$.

Generalization

Joins with predicates involving Allen or ISEQL relations are not exactly the same as the one-dimensional line segment intersection. Nevertheless, the joins can be mapped onto orthogonal line segment intersection, which is a special case of two-dimensional line segment intersection that can also be done in $O(n \log n + k)$, with $k = n^2$ in the worst case, using a plane-sweeping algorithm that traverses the segments sorted by one dimension [12]. This also explains why there were no further developments for interval joins recently, as the state-of-the-art algorithms achieve this complexity. However, when generating the output, we cannot just join a segment with all active ones, we need to check additional constraints: two segments can overlap on the x-axis, but may or may not do so on the y-axis. As we will see shortly, this has implications for the data structure D.

Complexity of Different Join Predicates

Let us now have a closer look at the different join predicates. For all of them, we need the relations $\boldsymbol{r}$ and $\boldsymbol{s}$ to be sorted. Either we keep them in a Timeline Index or operate in a streaming environment, in which they are already sorted, or we need to sort them in $O(n \log n)$. The non-parameterized and parameterized versions of START PRECEDING, END FOLLOWING, and BEFORE (which includes MEETS in its parameterized version) are not hard to analyze. They all have a complexity of $O(n \log n + k)$. For START PRECEDING, we maintain the active tuple set of $\boldsymbol{r}$ in a gapless hash map, which means $O(1)$ for the insertion and removal of a single tuple, or $O(2n) = O(n)$ in total. Additionally, whenever we encounter a starting event of $\boldsymbol{s}$, we generate result tuples, resulting in a total of $O(k)$ for generating all the output. For the parameterized version, we merely shift the endpoints of the tuples in $\boldsymbol{r}$. END FOLLOWING is very similar, the only differences being that we generate output when encountering ending events of $\boldsymbol{s}$ and for the parameterized variant, we shift the starting points of $\boldsymbol{r}$. BEFORE is not much different, we shift both events of tuples in $\boldsymbol{r}$ and whenever we encounter starting events of $\boldsymbol{s}$, we generate the output.

We now turn to OVERLAP and DURING joins, which we implement using START PRECEDING (or END FOLLOWING) joins; the same reasoning also holds for our implementation of the EQUALS, STARTS, and FINISHES joins. Processing a LEFT OVERLAP or a REVERSE DURING join, we cannot just output the results in a straightforward way when encountering a starting event in $\boldsymbol{s}$ as before, as at this point we cannot determine whether two intervals are in a left overlap or reverse during relationship: the relationship between the starting events both look the same, we need to see the ending events to make a final decision. A similar argument holds for implementing OVERLAP and DURING joins with END FOLLOWING joins: the role of the starting and ending events are switched in this case. The textbook solution is to keep the intervals sorted by ending events, e.g., in a tree. We can then search quickly for the qualifying tuples in this tree and generate the output, resulting in an overall complexity of $O(n \log n + k)$.[4] However, it is more difficult to do this in a cache-friendly manner, as a tree traversal entails more random I/O than a sequential scan. Using a gapless hash map instead, we go through *all* the tuples in the active tuple set. Compared to the tree data structure, the processing of the join

[4] Assums that insertion and removal cost us $O(\log n)$.

generates a larger intermediate result, as we join all intervals that satisfy an OVERLAP or DURING join predicate. We filter out the tuples satisfying the predicate we are not interested in afterward with a selection operator. Consequently, our approach has an overall complexity of $O(n \log n + k')$ for OVERLAP and DURING joins, with $k' geq k$. However, we utilize a sequential scan during the processing and as we will see in the experimental evaluation, introducing random I/O into the traversal of the active tuple set (like in a tree data structure) starves the CPU and slows down the whole process by two orders of magnitude. On paper, our approach looks worse, but in practice it outperforms the allegedly better method.

9 Experimental evaluation

9.1 Setup

Environment

All algorithms were implemented in-memory in C++ by the same author and compiled with GCC 4.9.4 to 64-bit binaries using the `-O3` optimization flag. We executed the code on a machine with two Intel Xeon E5-2667 v3 processors under Linux. All experiments used 12-byte tuples containing two 32-bit time stamp attributes (T_s and T_e) and a 32-bit integer payload. All experiments were repeated (also with bigger tuple sizes) on a seven-year-old Intel Xeon X5550 processor and on a notebook processor i5-4258U, showing a similar behavior.

Algorithms

We compare our approach with the Leung–Muntz family of sweeping algorithms [30,31] and with an algorithm for generic inequality joins, IEJoin [26]. We implemented the Leung–Muntz algorithms in the most effective way, i.e., performing all stages of the algorithm simultaneously, as recommended by the authors. For a fair comparison, we stored the set of started tuples in a Gapless List, adapting the Gapless Hash Map technique (Sect. 7.4) to the Leung–Muntz algorithms to boost their performance. We implemented IEJoin using all optimizations from the original paper. Our algorithms were implemented as described before, i.e., using abstractions and lambda functions.

The workload for all algorithms consisted of accumulating the sum of T_s attributes of the joined tuples. For benchmarking, we implemented the tuples as structures and the relations as `std::vector` containers. The Endpoint Index was implemented analogously, using structures for the endpoints and a vector for the index.

Synthetic Datasets

To show particular performance aspects of the algorithms we create synthetic datasets with uniformly distributed starting points of the intervals in the range of $[1, 10^6]$. The duration of the intervals is distributed exponentially with rate parameter λ (with an average duration $1/\lambda$). To perform a join, both relations in an individual workload follow the same distribution, but are generated independently with a different seed. In the experiments, for a specific value of λ, we varied the cardinality of the generated relations.

Table 3 Real-world dataset statistics

dataset	n	$\|r.T\|$			$r.T_s$ and $r.T_e$	
		Min	Avg	Max	Domain	#Distinct (k)
Flight	58 k	61	8 k	86 k	812 k	10
Inc	84 k	2	184	574	9 k	2.7
Web	1.2 M	1	60 M	352 M	352 M	110
Feed	3.7 M	1	432	8.5 k	8.6 k	5.6
Basf	5.3 M	1	127 k	16 M	16 M	760

Real-World Datasets

We use five real-world datasets that differ in size and data distribution. The main properties of them are summarized in Table 3. Here n is the number of tuples, $|r.T|$ is the tuple interval length, '$r.T_s$ and $r.T_e$ domain' is the size of the time domain of the dataset and '$r.T_s$ and $r.T_e$ #distinct' is the number of distinct time points in the dataset.

The *flight* dataset [5] is a collection of international flights for November 2014; start and end of the intervals represent plane departure and arrival times with minute precision. The Incumbent (*inc*) dataset [17] records the history of employees assigned to projects over a sixteen year period at a granularity of days. The *web* dataset [1] records the history of files in the SVN repository of the Webkit project over an eleven year period at a granularity of seconds. The valid times indicate the periods in which a file did not change. The *feed* dataset records the history of measured nutritive values of animal feeds over a 24 year period at a granularity of days; a measurement remains valid until a new measurement for the same nutritive value and feed becomes available [14]. Finally, rather than using time as a domain, the dataset *basf* contains NMR spectroscopy data describing the resonating frequencies of different atomic nuclei [19]. As these frequencies can shift, depending on the bonds an atom forms, they are defined as intervals. For the experiments we used self-joins of these datasets, the only exception are the 'wi' and 'fi' workloads, where we joined the 'web' and 'feed' datasets with 'incumbent'.

9.2 Experiments and results

9.2.1 Cache efficiency

First, we look at the impact of improving the cache efficiency of the data structure used for maintaining the active tuple set. We investigate the average latency of a `getnext` operation,

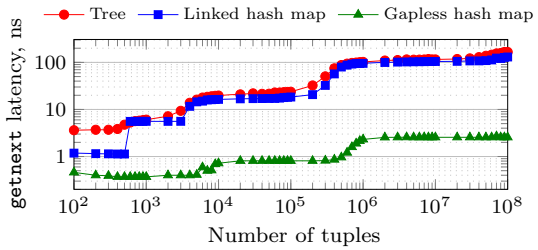


Fig. 9 Latency of `getnext` operation

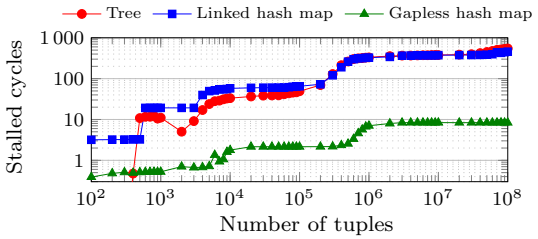


Fig. 10 Stalled CPU cycles per `getnext` operation

which is crucial for generating the result tuples. We compare a linked hash map (Sect. 7.1), a gapless hash map (Sect. 7.4), and a tree structure (mentioned in Sect. 8). The tree was implemented using a red-black tree (std::map) from the C++ Standard Library.

We filled the data structures with various numbers of 32-byte tuples, then randomly added and removed tuples to simulate the management of an active tuple set. Afterward, we performed several scans of the data structures. Figure 9, shows the average latency of a `getnext` operation depending on the number of tuples (note the double-logarithmic scale).

We see that the latency of a `getnext` operation is not constant but grows depending on the memory footprint of the tuples. In order to find the cause of this, we used the Performance Application Programming Interface (PAPI) library to read out the CPU performance counters [34]. When looking at the average number of stalled CPU cycles (PAPI-RES-STL) per `getnext` operation, we get a very similar picture (see Fig. 10). Therefore, the latency is clearly caused by the CPU memory subsystem.

In Fig. 9 we can easily identify three distinct transitions. In case of a small number of tuples, all of them fit into the L1d CPU cache (32 KB per core) and we have a low latency. For the tree and linked hash map, as the tuple count grows toward 500 tuples, we start using the L2 cache (256 KB per core) with a greater latency. When we increase the number of tuples further and start reaching 4000 tuples, the data are mostly held in the L3 cache (20 MB in total, shared by all cores) and, finally, after arriving at a tuple count of around 300 000, the tuples are mostly located in RAM.[5] We make a couple of important observations. First, due to the more compact storage scheme of the gapless hash map, the transitions set in later (at 5000, 10 000, and 600 000 tuples, respectively). Second, the improvement gains of the gapless hash map are considerable and can be measured in orders of magnitude (note the logarithmic scale). Third, the latency of a `getnext` operation for the gapless hash map plateaus at around 2.7 ns, while the latency for the linked hash map and the tree reaches 100 ns.

Cache misses alone do not explain all of the latency. Figure 11 shows the average number of cache misses for the L1d (PAPI-L1-DCM), the L2 (PAPI-L2-TCM), and the L3 cache (PAPI-L3-TCM). While in general the average number of cache misses per `getnext` operation is lower for the gapless hash map, the factor between the data structures in terms of stalled CPU cycles is disproportionately higher (please note the double-logarithmic scale in Fig. 10). Also, the cache misses do not explain the leftmost part of Fig. 10, in which there are no cache misses at all. The additional performance boost stems from out-of-order execution. Examining the different (slightly simplified) versions of the machine code generated for `getnext` makes this clear. For the gapless hash map, the code looks like this:

```
loop:
  add   rax, [rdx]
  add   rdx, 32          ; pointer += 32 (increment)
  cmp   rcx, rdx
  jne   loop
```

while for the linked hash map we have the following picture (we omit the code for the tree, as it is much more complex):

```
loop:
  add   rax, [rdx]
  mov   rdx, [rdx + 32] ; pointer = pointer->field (dereference)
  cmp   rcx, rdx
  jne   loop
```

When scanning through a gapless hash map, we add a constant to the pointer, which means that there is no data dependency between loop iterations. Consequently, the CPU is able to predict the instructions that will be executed in the future and can already start preparing them out of order (i.e., issue cache misses up front for the referenced data) while some of the instructions are still waiting for data from the L1 cache. For the linked hash map and the tree the CPU has to wait until a pointer to the next item has been dereferenced. In summary, multiple parallel cache misses in a sequential access pattern are processed much faster than isolated requests to random memory locations.

[5] All CPUs we used have 32 KB and 256 KB per core for the L1d and L2 cache, respectively. The L3 cache for the Xeon X5550 is 8 MB and for the i5-4258u 3 MB, which means that they reach the last phase earlier.

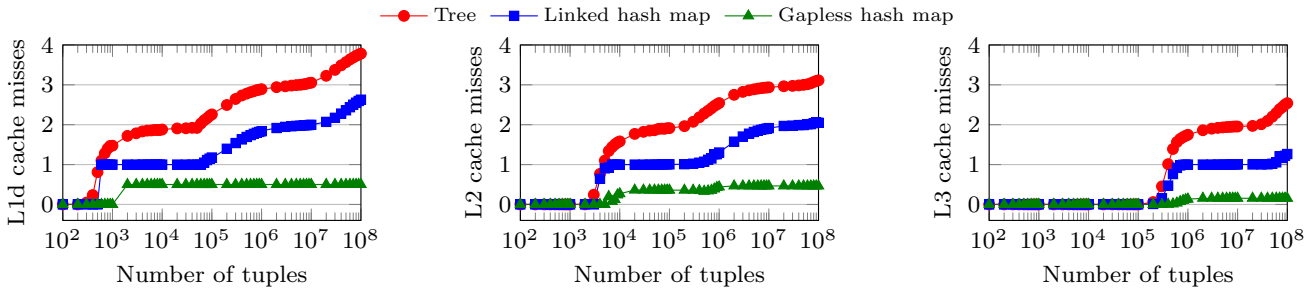


Fig. 11 Cache misses per `getnext` operation

We made another observation: there were no L1 instruction (L1i) cache misses. The increase in L1d cache misses for the linked hash map and the tree for large numbers of tuples is caused by TLB cache misses.

We obtained very similar results for different CPUs on different machines (the diagrams shown here are for an Intel Xeon E5-2667 v3 processor), which led us to the conclusion that the techniques we employ will generally improve the performance on CPU architectures with a cache hierarchy, prefetching, and out-of-order execution. For the remainder of the experiments we only consider the gapless hash map, as it clearly outperforms the linked hash map.

9.2.2 Lazy joining

For every tuple in **s**, the basic JoinByS algorithm (Sect. 5.3) scans the current set of active tuples in **r**. Using the improved LazyJoinByS algorithm from Sect. 7.2, we can reduce the number of scans considerably. As long as we only encounter starting events of tuples in **s** and no events caused by tuples in **r**, we can delay the scanning of the active tuple set of **r**.

Analyzing the Data

We now take a closer look at how frequently such uninterrupted sequences of events of one relation appear. Figure 12 shows these data for the table 'Incumbent' from the real-world datasets when joining it with itself. On the x-axis we have the length of uninterrupted sequences of starting events and on the y-axis their relative frequency of appearance. In 60% of the cases we have sequences of length ten or more, meaning that our lazy joining technique can avoid a considerable number of scans on active tuple sets.

We found that starting events of intervals are generally not uniformly distributed in real-world datasets, but tend to cluster around certain time points. This can be recognized by looking at the number of distinct points in Table 3. For example, for the 'Incumbent' dataset, employees are usually not assigned to new projects on random days, the assignments tend to happen at the beginning of a week or month. For the 'Feed' dataset, multiple measurements (which are valid until the next one is made) are taken in the course of a day, resulting in a whole batch of intervals starting at the same

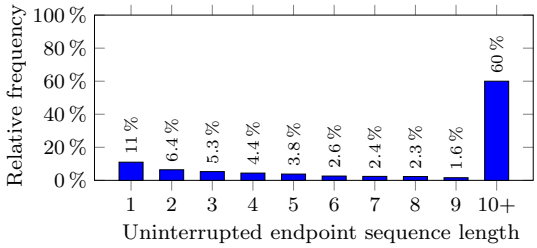


Fig. 12 Distribution of uninterrupted sequence lengths for self-join of the 'inc' dataset

time. The clustering is not just due to the relatively coarse granularity (one day) of these two datasets. The 'Webkit' repository dataset, which looks at intervals in which files are not modified, has a granularity measured in milliseconds. Still we observe a clustering of starting events: a commit usually affects and modifies several files. The 'Flight' dataset, which has a granularity of minutes, also exhibits a similar pattern in the form of batched departure times. Even for the frequency data of the 'BASF' dataset, the values for the start and end points of the intervals seem to be clustered around multiples of one hundred.

Reduction Factor

The real performance implication is that LazyJoinByS executes fewer `getnext` operations than JoinByS in such a scenario. The actual reduction depends not only on the clusteredness of the events, but also on the size of the corresponding active tuple set and the buffer capacity reserved in LazyJoinByS. We define a `getnext` *operation reduction factor* (*GNORF*), changing the cost for scanning through active tuple sets for the LazyJoinByS to $^{k \cdot c_{getnext}}/_{GNORF}$, where k is the cardinality of the result set.

For the self-join of the 'Incumbent' dataset and for buffer capacity of 32 the *GNORF* is equal to 23.6, which corresponds to huge savings in terms of run time. We also calculated this statistic for self-joins of other real-world datasets ('feed: 31.2, 'web': 9.73, flight': 7.14, 'basf': 11.2). Even for self-joins we get a considerable reduction factor: when encountering multiple starting events with the same time stamp, we first deal with all those of one relation before those of the other.

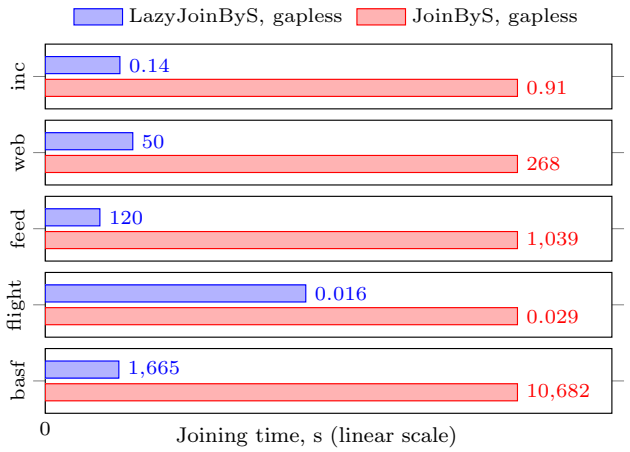


Fig. 13 JoinByS versus LazyJoinByS, real-world data

Join Performance

Next we investigate the relative performance of an actual join operation, employing JoinByS and LazyJoinByS for an overlap join. Figure 13 depicts the results for the 'Incumbent' (inc), 'Webkit' (web), 'Feed'(feed), "flight", and 'basf' datasets, showing that LazyJoinByS outperforms JoinByS by up to a factor of eight. Therefore, we only consider LazyJoinByS from here on.

9.2.3 Scalability

To test the scalability of our algorithms we compared them to the Leung–Muntz and IEJoin algorithms while varying the cardinality of the synthetic datasets. For the IEJoin we used the algebraic expressions in Table 1. Due to space constraints, we limit ourselves to three characteristic joins (join-only, join with selection, and parameterized join with map operators): START PRECEDING ($\preceq$), INVERSE DURING ($\succeq$), and BEFORE join ($\prec$), where δ is set to the average tuple length in the outer relation. We tested the algorithms using short, medium-length and long tuples with average lengths of $0.5 \cdot 10^2$, $0.5 \cdot 10^4$, and $0.5 \cdot 10^6$ time points, respectively. The speedup of our approach compared to Leung–Muntz and IEJoin is shown in Figs. 14 and 15, respectively. We see that our solution quickly becomes faster than the Leung–Muntz algorithms and that the difference grows with increasing numbers of tuples and their lengths. Only for small relations and short tuples, Leung–Muntz is faster. Leung–Muntz is a simpler algorithm, so for light workloads, i.e., small relations and short intervals (resulting in smaller result sets), it shows a good performance as it does not have an initialization overhead. However, it does not scale as well as our algorithm. In the left most diagram of Fig. 14 (INVERSE DURING JOIN), we also show the difference between using gapless hash maps and trees for managing the active tuple set. We only do so for the INVERSE DURING JOIN, as for the other joins a tree would only add overhead without any benefits. For the INVERSE DURING JOIN, with a tree we generate only valid tuples, meaning that we do not need a selection operator as needed for the gapless hash map. While the tree-based active tuple set seems to pay off for long tuples (meaning larger active tuple sets), for shorter tuples the situation is not that clear. Consequently, we propose to use gapless hash maps, as this avoids the implementation and integration of an additional data structure that is only useful for some interval relations and even then does not always show superior performance. For the remainder of Sect. 9, we restrict ourselves to gapless hash maps.

For the IEJoin, the performance differs by one to several orders of magnitude depending on relation cardinalities and tuple lengths. Even though the IEJoin is highly optimized, it still has quadratic complexity and cannot compete with specialized algorithms. Therefore, we drop it from the further investigation. Because the Leung–Muntz and the IEJoin algorithms do not scale well, we stopped running experiments for larger relation cardinalities when they took a few hours to conduct. We also restrict ourselves to the INVERSE DURING join from now on, since Leung–Muntz showed the best performance for it.

9.2.4 Real-world workloads

We repeated the experiments on the real-world workloads. The speedup is shown in Fig. 16. Here the performance difference is two orders of magnitude in some cases. On the one hand, this is due to the lazy joining cache optimization, which is more effective for the real-world datasets (cf. [36]). On the other hand, the heuristics used in the Leung–Muntz algorithm work worse for real-worlds workloads and especially those where the relation cardinalities differ substantially.

9.2.5 Comparison with bgFS

Our approach and bgFS [8] follow different paradigms for processing the data: backward scanning versus forward scanning. The iterator framework we utilize has been geared completely toward the backward scanning paradigm, allowing us to introduce changes to endpoint time stamps on the fly. This makes it challenging to integrate bgFS into our iterator framework effectively. Clearly, we can add shifted intervals to the tuples in the relations before executing an bgFS join. However, this requires an additional sweep over the relations, which cannot be done on the fly (an intermediate relation has to be materialized and sorted): bgFS will start producing output tuples at the very end of the processing time. Figure 17 shows the run time of processing a (general) BEFORE join using bgFS and our approach (this was run on an i7-4870HQ CPU with four cores, 32 KB and 256 KB per core for the L1d and L2 cache, respectively, and 6 MB L3 cache).

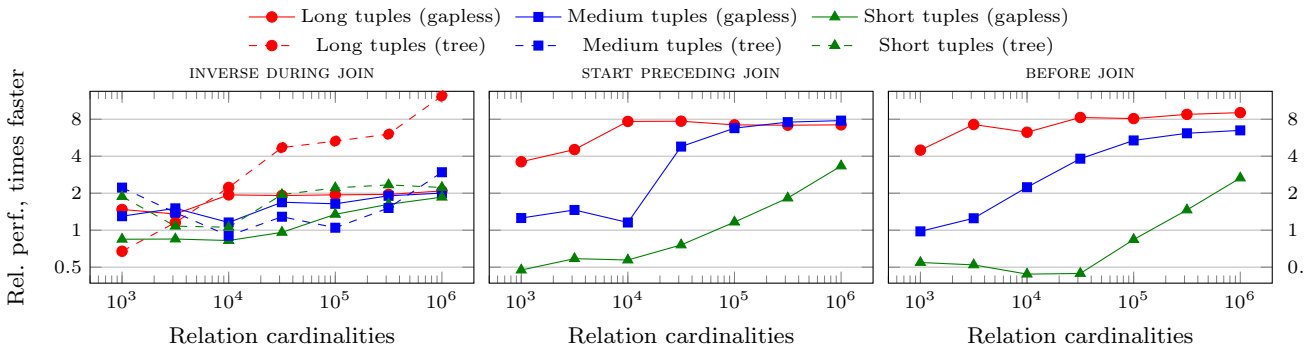


Fig. 14 Performance of our solution w.r.t. Leung–Muntz algorithm, synthetic data

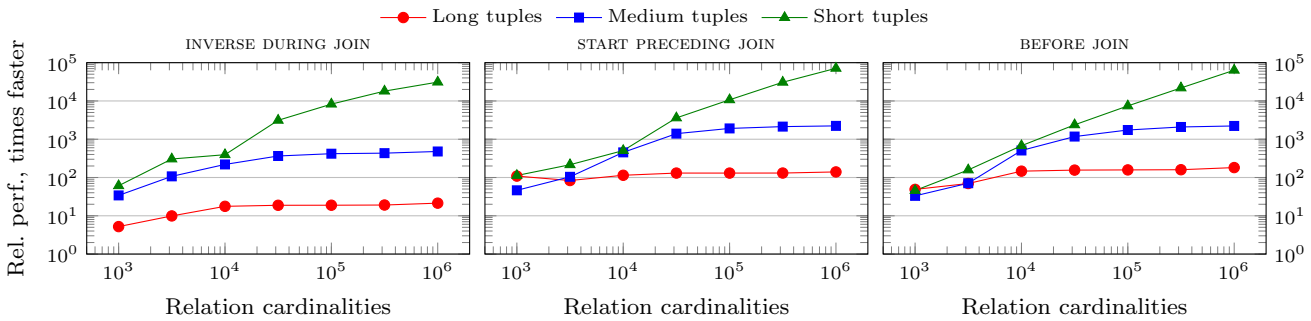


Fig. 15 Performance of our solution w.r.t. IEJoin, synthetic data

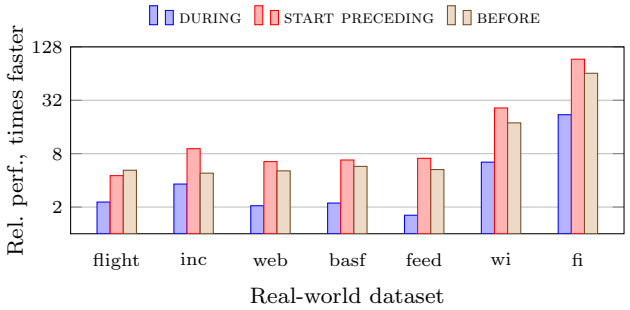


Fig. 16 Performance of our solution w.r.t. Leung–Muntz, real-world data

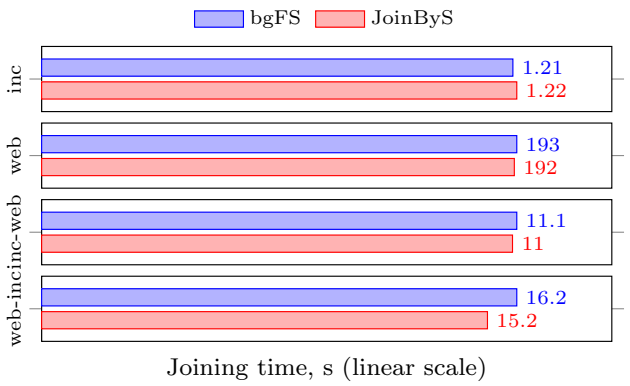


Fig. 17 Comparison of JoinByS with bgFS, real-world data

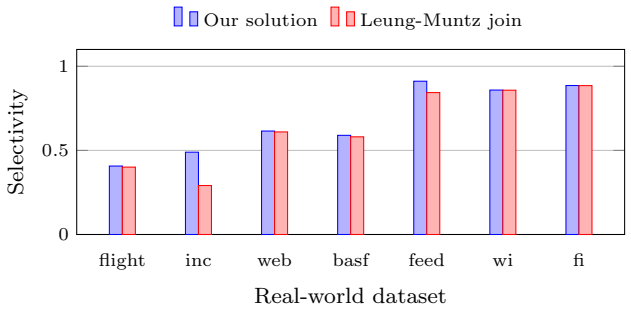


Fig. 18 Selectivity of the filtering selection operator after the main join, INVERSE DURING JOIN

9.2.6 Selection efficiency

To explore the source of the performance difference between the algorithms, we tested the selectivity of the selection operation that is applied after the join in the Leung–Muntz algorithms and, in some cases, in ours. The results for the INVERSE DURING JOIN are shown in Fig. 18. The other joins exhibit a similar behavior. We see that both algorithms are keeping the sizes of the working sets similar. Our algorithm is slightly more effective, but insufficiently so to explain the performance difference. We look at the real cause in the next section.

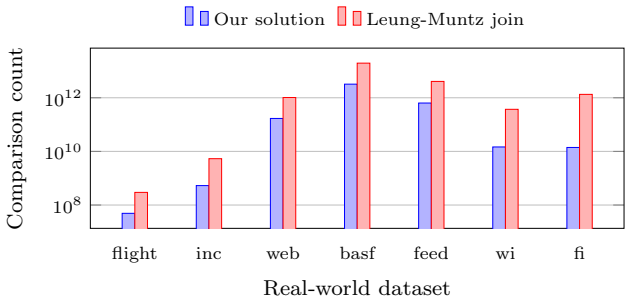


Fig. 19 Join comparison counts, real-world data

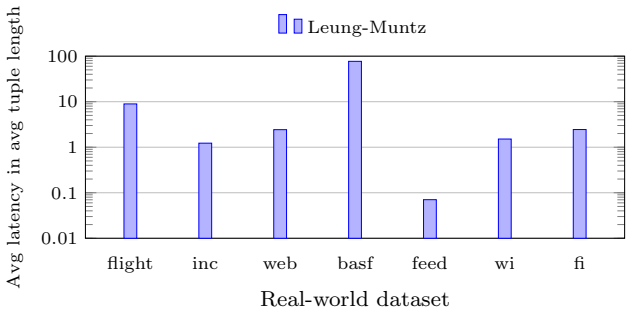


Fig. 20 Algorithm reporting latency, REVERSE DURING JOIN, real-world data

9.2.7 Comparison count

In this experiment we measured the number of tuple endpoint comparison operations (e.g., '$r.T_s < s.T_s$'. The results for the INVERSE DURING JOIN are shown in Fig. 19. We see that the difference in the number of comparisons is huge. The Leung–Muntz algorithm performs many more comparisons because it has to heuristically estimate the next tuple to be read and to perform the garbage collection of the outdated tuples. The selection operation of the Leung–Muntz algorithm requires two comparisons. Our algorithm, on the other hand, requires a single comparison in the selection operation for the INVERSE DURING JOIN, and no tuple comparisons at all for the BEFORE and START PRECEDING JOIN.

9.2.8 Latency

In this experiment we measure the delay in producing output tuples of the Leung–Muntz algorithms. A low latency is an important feature for event detection systems. While our algorithms generate output as soon as possible, when all of the required endpoints have been spotted, the Leung–Muntz has a delay caused by the fact that it requires streams of complete and ordered tuples as the input. The average latency (expressed in average tuple lengths, as the different datasets have vastly different granularities) is shown in Fig. 20. Depending on the workload the differences in latency can in some cases reach ten or even a hundred times the average tuple length.

10 Conclusions and future work

We developed a family of effective, robust and cache-optimized plane-sweeping algorithms for interval joins on different interval relationship predicates such as Allen's or parameterized ISEQL relations. The algorithms can be used in temporal databases, exploiting the Timeline Index, which made its way into a prototype of a commercial temporal RDBMS as the main universal index supporting temporal joins, temporal aggregation and time travel. We thus extend the set of operations supported by this index. Our solution is based on a flexible framework, which allows combining its components in different ways to elegantly and efficiently express any interval join in terms of a single core function. Additionally, our approach makes good use of the features of contemporary hardware, utilizing the cache infrastructure well.

We compared the performance of our solution with the state of the art in interval joins on Allen's predicates—the Leung–Muntz and the IEJoin algorithm. The results show that our solution is several times faster, scales better and is more stable. Another major advantage of our approach is that it can be directly applied to real-time stream event processing, as it will report the results as soon as logically possible for the applied predicate, without necessarily waiting for intervals to finish. The Leung–Muntz algorithm has to wait for the tuples to finish before processing them. Moreover, the requirement for tuples to be processed chronologically allows any unfinished tuple to block the processing of all following tuples. The IEJoin is also not suitable for a streaming environment: it needs the complete relations to work.

For future work, we want to explore the possibilities of embedding our solution into a real-time complex event processing framework. In particular, we want to combine the results of multiple joins to detect patterns within n streams of events of different types. Additional research directions are working out the details of parallelizing our approach and the handling of multiple predicates in a join operation.

Funding Open Access funding provided by Universität Zürich

A Implementation of iterators

Index Iterator

We use the `std::vector` container of the C++ Standard Template Library (STL) as the implementation of the Endpoint Index, resulting in the following code:

```
– IndexIterator(endpointIndex):
    this.it = endpointIndex.begin();
    this.end = endpointIndex.end();
– getEndpoint:
    return *it;
– moveToNextEndpoint:
    ++it;
– isFinished:
    return it == end;
```

Filtering Iterator

```
– FilteringIterator(iterator, type):
    this.iterator = iterator;
    this.type = type;
    while getEndpoint.type ≠ type do
      moveToNextEndpoint;
– getEndpoint:
    return iterator.getEndpoint;
– moveToNextEndpoint:
    do
      iterator.moveToNextEndpoint;
    while not isFinished and
          getEndpoint.type ≠ type;
– isFinished:
    return iterator.isFinished;
```

Shifting Iterator

```
– ShiftingIterator(iterator, delta, type):
    this.iterator = iterator;
    this.delta = delta;
    this.type = type;
– getEndpoint:
    var endpoint = iterator.getEndpoint;
    endpoint.timestamp += delta;
    endpoint.type = type;
    return endpoint;
– moveToNextEndpoint:
    iterator.moveToNextEndpoint;
– isFinished:
    return iterator.isFinished;
```

Merging Iterator

```
– MergingIterator(iterator1, iterator2):
    this.it1 = iterator1;
    this.it2 = iterator2;
    moveToNextEndpoint;
– getEndpoint:
    return this.endpoint;
– moveToNextEndpoint:
    if it2.isFinished or not it1.isFinished
       and it1.getEndpoint < it2.getEndpoint
    then
      this.endpoint = it1.getEndpoint;
      it1.moveToNextEndpoint;
    else
      this.endpoint = it2.getEndpoint;
      it2.moveToNextEndpoint;
    end
– isFinished:
    return it1.isFinished and it2.isFinished;
```

First End Iterator

```
– FirstEndIterator(iterator):
    this.iterator = iterator;
    this.hs = new HashSet;

– getEndpoint:
    return this.endpoint;
– moveToNextEndpoint:
    do
      iterator.moveToNextEndpoint;
      if getEndpoint.type = end then
        if getEndpoint.tuple_id ∉ hs then
          insert getEndpoint.tuple_id into hs;
          break;
        else
          remove getEndpoint.tuple_id from hs;
    while not isFinished;
– isFinished:
    return iterator.isFinished;
```

Second Start Iterator

```
– SecondStartIterator(iterator):
    this.iterator = iterator;
    this.hs = new HashSet;
    while getEndpoint.type = start and
          getEndPoint.tuple_id ∉ hs
    do
      moveToNextEndpoint;
– getEndpoint:
    return this.endpoint;
– moveToNextEndpoint:
    do
      iterator.moveToNextEndpoint;
      if getEndpoint.type = start then
        if getEndpoint.tuple_id ∈ hs then
          remove getEndpoint.tuple_id from hs;
          break;
        else
          insert getEndpoint.tuple_id into hs;
    while not isFinished;
– isFinished:
    return iterator.isFinished;
```

References

1. The webkit open source project. https://webkit.org/ (2012)
2. Allen, J.F.: Maintaining knowledge about temporal intervals. Commun. ACM **26**(11), (1983)
3. Álvarez, M.R., Félix, P., Cari Nena, P.: Discovering metric temporal constraint networks on temporal databases. Artif. Intell. Med. **58**(3), 139–154 (2013)
4. Arge, L., Procopiuc, O., Ramaswamy, S., Suel, T., and Vitter, J.S.: Scalable sweeping-based spatial join. In: VLDB (1998)
5. Behrend, A., and Schüller, G.: A case study in optimizing continuous queries using the magic update technique. In: SSDBM (2014)
6. Bettini, F., Persia, F., and Helmer, S.: An interactive framework for video surveillance event detection and modeling. In: CIKM (2017)
7. Blakeley, J.A., McKenna, W.J., and Graefe, G.: Experiences building the open OODB query optimizer. In: SIGMOD (1993)
8. Bouros, P., Mamoulis, N.: A forward scan based plane sweep algorithm for parallel interval joins. PVLDB **10**(11), (2017)
9. Bouros, P., and Mamoulis, N.: Interval count semi-joins. In: EDBT (2018)
10. Chawda, B., Gupta, H., Negi, S., Faruquie, T.A., Subramaniam, L.V., and Mohania, M.: Processing interval joins on map-reduce. In: EDBT (2014)
11. Chekol, M.W., Pirrò, G., and Stuckenschmidt, H.: Fast interval joins for temporal SPARQL queries. In: Companion of WWW, pp 1148–1154 (2019)
12. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
13. Dechter, R., Meiri, I., Pearl, J.: Temporal constraint networks. Artif. Intell. **49**(1–3), 61–95 (1991)
14. Dignös, A., Böhlen, M.H., and Gamper, J.: Overlap interval partition join. In: SIGMOD (2014)
15. Freksa, C.: Temporal reasoning based on semi-intervals. Artif. Intell. **54**(1–2), 199–227 (1992)
16. Gao, D., Jensen, C.S., Snodgrass, R., and Soo, M.D.: Join operations in temporal databases. VLDB J. **14**(1), (2005)
17. Gendrano, J.A.G., Shah, R., Snodgrass, R.T., and Yang, J.: University information system (UIS) dataset. TimeCenter CD-1, (1998)
18. Gunadhi, H., and Segev, A.: Query processing algorithms for temporal intersection joins. In: ICDE (1991)
19. Helmer, S.: An interval-based index structure for structure elucidation in chemical databases. In: IFSA (2007)
20. Helmer, S., Persia, F.: ISEQL, an interval-based surveillance event query language. IJMDEM **7**(4), (2016)
21. Höppner, F., Peter, S.: Temporal interval pattern languages to characterize time flow. WIREs Data Min. Knowl. Discov. **4**(3), 196–212 (2014)
22. Jensen, C.S., Snodgrass, R.T.: Temporal data management. IEEE Trans. Knowl. Data Eng. **11**(1), 36–44 (1999)
23. Kaufmann, M.: Storing and processing temporal data in main memory column stores. PhD thesis, ETH Zurich (2014)
24. Kaufmann, M., Fischer, P.M., May, N., Ge, C., Goel, A.K., and Kossmann, D.: Bi-temporal timeline index: a data structure for processing queries on bi-temporal data. In: ICDE (2015)
25. Kaufmann, M., Manjili, A.A., Vagenas, P., Fischer, P.M., Kossmann, D., Färber, F., and May, N.: Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In: SIGMOD (2013)
26. Khayyat, Z., Lucia, W., Singh, M., Ouzzani, M., Papotti, P., Quiané-Ruiz, J.-A. Tang, N., and Kalnis, P.: Fast and scalable inequality joins. VLDB J. **26**(1), (2017)
27. Körber, M., Glombiewski, N., Morgen, A., and Seeger, B.: Tpstream: low-latency and high-throughput temporal pattern matching on event streams. Distrib. Parallel Databases (2019)
28. Körber, M. Glombiewski, N., and Seeger, B.: Tpstream: Low-latency temporal pattern matching on event streams. In: EDBT pp 313–324, (2018)
29. Kulkarni, K.G., Michels, J.: Temporal features in SQL: 2011. SIGMOD Rec. **41**(3), 34–43 (2012)
30. Leung, T.Y.C., Muntz, R.R.: Query Processing for Temporal Databases. Technical Report CSD-890020. University of California, California (1989)
31. Leung, T.Y.C., and Muntz, R.R.: Query processing for temporal databases. In ICDE (1990)
32. Leung, T.Y.C., and Muntz, R.R.: Temporal query processing and optimization in multiprocessor database machines. In: VLDB (1992)
33. Ma, J., Hayes, P.J.: Primitive intervals versus point-based intervals: Rivals or allies? Comput. J. **49**(1), 32–41 (2006)
34. Moore, S., and Ralph, J.: User-defined events for hardware performance monitoring. In: ICCS Workshop (2011)
35. Piatov, D., and Helmer, S.: Sweeping-based temporal aggregation. In: SSTD (2017)
36. Piatov, D., Helmer, S., and Dignös, A.: An interval join optimized for modern hardware. In: ICDE (2016)
37. Pilourdault, J., Leroy, V., and Amer-Yahia, S.: Distributed evaluation of top-k temporal joins. In: SIGMOD (2016)
38. Segev, A., and Gunadhi, H.: Event-join optimization in temporal relational databases. In: VLDB (1989)
39. Wu, S.-Y., Chen, Y.-L.: Discovering hybrid temporal patterns from sequences consisting of point- and interval-based events. Data Knowl. Eng. **68**(11), 1309–1330 (2009)