Parallel Processing of Temporal Anti-Joins in Memory

Ioannis Reppas[1 [0009−0006−8021−2338]], Meghdad Mirabi[2,1 [0000−0003−3803−2756]] ✉,
Leila Fathi[1 [0009−0002−7966−8873]], Carsten Binnig[1,2 [0000−0002−2744−7836]],
Anton Dignös[3 [0000−0002−7621−967X]], and Johann Gamper[3 [0000−0002−7128−507X]]

[1] Technical University of Darmstadt, Darmstadt, Germany
ioannis.reppas@stud.tu-darmstadt.de,
meghdad.mirabi@cs.tu-darmstadt.de, lfathi@savian.io,
carsten.binnig@cs.tu-darmstadt.de
[2] German Research Center for Artificial Intelligence (DFKI), Darmstadt, Germany
meghdad.mirabi@dfki.de, carsten.binnig@dfki.de
[3] Free University of Bozen-Bolzano, Bolzano, Italy
anton.dignoes@unibz.it, johann.gamper@unibz.it

Abstract. Efficient and scalable processing of temporal anti-joins remains a significant research challenge in temporal databases. To address this issue, this paper introduces a novel temporal primitive designed for transforming a temporal anti-join, including conjunctive equality predicates on non-temporal attributes, into an equivalent algebraic expression involving a temporal inner join. The rationale behind this transformation is that the new expression can be decomposed into subtasks, allowing for parallel execution across multiple CPUs. Experimental results using real-world datasets demonstrate the superior efficiency and scalability of our solution for in-memory processing compared to existing solutions.

Keywords: Temporal Anti-Join · Main Memory · Parallel Processing

1 Introduction

A temporal join is a database operation that combines two temporal relations and returns pairs of tuples from the two relations that match on a value predicate and have overlapping timestamps, indicating that tuples match over some time interval [19, 21, 33, 16]. This allows to retrieve data that is valid at the same time [22, 31]. In contrast, a *temporal anti-join* reports all tuples over sub-intervals of one input relation for which no matching tuple in the other relation exists [15, 12, 27, 13, 9]. This operation is crucial for identifying missing, non-overlapping, or orphaned tuples in a temporal context.

Example 1. Consider the two temporal relations *Employee Schedule* and *Employee Absence* in Fig. 1. *Employee Schedule* records the work schedule of employees using half-open intervals [Start Time, End Time). *Employee Absence* records the time periods when employees were absent from work for some reasons. For instance, John was scheduled to work from Jan 2 to Jan 15, but was absent during the time periods Jan 8–10 and Jan 12–14 due to, respectively, a doctoral appointment and for personal reasons.

For a correct payroll processing, the company needs to determine the time during which employees effectively worked. This information can be retrieved by a temporal

(a) Employee Schedule

Name	Dep	[Start Time, End Time)
John	Sales	[2023-01-02, 2023-01-15)
Bob	Marketing	[2023-01-01, 2023-01-10)
Mike	Sales	[2023-01-11, 2023-01-15)

(b) Employee Absence

Name	Reason	[Start Time, End Time)
Bob	Family Emergency	[2023-01-03, 2023-01-07)
John	Doctor Appointment	[2023-01-08, 2023-01-10)
John	Personal	[2023-01-12, 2023-01-14)

(c) Temporal Anti-Join

Name	Dep	[Start Time, End Time)
Bob	Marketing	[2023-01-01, 2023-01-03)
Bob	Marketing	[2023-01-07, 2023-01-10)
John	Sales	[2023-01-02, 2023-01-08)
John	Sales	[2023-01-10, 2023-01-12)
John	Sales	[2023-01-14, 2023-01-15)
Mike	Sales	[2023-01-11, 2023-01-15)

Fig. 1: Motivating Example

anti-join between the relations *Employee Schedule* and *Employee Absence* using an equality predicate on the attribute *Name*, as shown in Fig. 1c. The same information is also helpful for workforce management, allowing managers to monitor employee attendance, to track their productivity, and to identify patterns in employees' behavior.

While temporal anti-joins have been increasingly used in data analysis, the efficient processing and scalability of temporal anti-joins have not be studied thoroughly in the past. One promising avenue to address these issues is parallel processing, which has, so far, been applied to temporal joins, but not to temporal anti-joins [8, 7, 26, 6].

Based on the above observations, this paper focuses on in-memory processing of temporal anti-joins, leveraging parallel processing features found in multi-core hardware. The proposed solution works in two steps: (1) compute the complement of one of the two relations and (2) transform the temporal anti-join into an equivalent expression using the complement in combination with a temporal join. This new expression can be efficiently executed in parallel, utilizing multiple CPU cores or threads. The main contributions of this paper are summarized as follows:

- We introduce a temporal primitive capable of transforming a temporal anti-join, including conjunctive equality predicates on non-temporal attributes, into an equivalent algebraic expression that involves a temporal inner join.
- We show how to decompose all operators within this temporal counterpart into small subtasks, and propose a set of algorithms for parallel, in-memory computation of these subtasks across multiple CPU cores or threads.
- We experimentally evaluate the efficiency and scalability of the proposed solution on four real-world datasets.

2 Related Works

In this section, we first review related work on processing temporal joins and their parallel execution. Then, we review approaches for the temporal anti-join.

Generally, temporal join algorithms can be classified based on their data structures and underlying architecture into nested loop, sort-merge, index-based, partitioning-based, sweep-plane, and parallel algorithms.

Nested loops algorithms [32] compare all tuples of two input relations. They scan the outer relation once and then compare each tuple of the outer relation with all tuples of the inner relation to identify overlapping intervals. *Sort-merge algorithms* [20, 28, 16] first sort both input relations based on their join attributes and then merge tuples simultaneously when they have the same join attribute values and overlapping intervals. *Index-based algorithms* use various indexing structures, such as B-Tree [24, 18, 16], multi-version B-Tree [2], TimeLine [23]), (key, time) [34], or hierarchical indices [11, 10] to find time interval intersections between input relations. *Partitioning-based algorithms* [14, 33] first divide the time domain into non-overlapping partitions, and then assign each tuple of an input relation to a specific partition based on its start or end timestamps. Finally, tuples from the input relations within relevant partitions are compared to identify tuples with overlapping time intervals. *Sweep-plane algorithms* can be divided into two categories: backward scanning and forward scanning. In backward scanning [29, 30], the two input relations are split and sorted based on their start and end time points. Then, a backward scan is performed, i.e., a scan of already encountered (active) intervals, on a data structure that supports scans and updates. On the other hand, forward scanning [7, 8] sorts the two input relations based on their start time points. Subsequently, a forward scan is conducted to identify the resulting tuples of a temporal join, without the need for a special data structure to track active tuples in both input relations. Both types of algorithms involve sweeping a line, which stops at the start or end time points of all time intervals within the two input relations. At each position of the sweep line, join results are generated.

The *parallel execution* of temporal joins can be divided into the following partitioning approaches. No-partitioning, hash-based partitioning, and domain-based partitioning algorithms [8]. In no-partitioning [3], the input relations are never physically partitioned, but a hash table is built in shared memory for the inner input relation. Then, each thread reads a chunk of the outer relation and probes the shared hash table to produce join results. In hash-based partitioning [29], the time intervals of the input relations are first sorted based on their start time points before partitioning. They are then assigned to different disjoint partitions in a round-robin fashion using a hash function. Subsequently, a pairwise join is performed between partitions of the input relation. Since the partitions are disjoint, the pairwise joins run independently of each other. In domain-based partitioning [8, 7], the time domain is initially split into a set of non-overlapping partitions. Each time interval in the input relations is assigned to the corresponding partition based on its start time point and replicated across other partitions until the partition that includes its end time point. Following this, a pairwise join is performed for each partition separately using a single thread, and a join result is reported if at least one of the time intervals is not replicated to avoid duplication in the result set.

Existing works on temporal joins primarily focus on efficiently identifying and extracting pairs of tuples from two input relations with overlapping time intervals. Apart from the works mentioned below, they do not support temporal anti-joins. In this paper, we transform a temporal anti-join into an equivalent expression that contains a temporal

join. To compute the temporal join we leverage the state-of-the-art approach for parallel in-memory computation by Bouros et. al [7, 8]. Since this approach does not support equality predicates in the temporal join, we extend it in Section 4.3. Currently, only a few approaches support the processing of temporal anti-joins. The alignment framework [15, 12] and rewriting approach [17], which are implemented in PostgreSQL, support a large range of operators including outer and anti-joins. Since these approaches leverage existing database algorithms for query processing, they did not provide specific algorithms for efficiently finding overlapping intervals. Disjoint Interval Partitioning (DIP) [9] provides processing algorithms for all types of temporal joins, including temporal anti-joins. It divides an input relation into the minimum number of partitions, such that tuples in a partition are non-overlapping, resulting in an efficient sort-merge approach without backtracking. To compute a temporal anti-join, DIP uses a *lead*, i.e., the set of time periods that are not overlapped by the second input relation. The authors show that DIP is efficient for the temporal join, outperforming [14, 20, 23, 1], and anti-join, outperforming [15, 12]. Different from our solution, DIP does not support equality predicates. For the experimental evaluation, we extend DIP with equality predicates and demonstrate that our solution is more efficient than the extended version of DIP for computing a temporal anti-join with conjunctive equality predicates.

3 Problem Statement

We consider a discrete linearly ordered time domain Ω^T. A time interval is a set of contiguous time points, and we use $T = [T_s, T_e)$ to denote the half-open interval of time points from T_s (included) to T_e (excluded), similarly to SQL:2011 [4, 5, 25].

A temporal relation R with schema $\mathcal{R} = (\mathbf{A}, T)$ is a set of tuples, where each tuple has a set of non-temporal attributes $\mathbf{A} = A_1, \ldots, A_m$, each with a domain Ω_i and a time interval T. Similarly, we consider a temporal relation S with schema $(\mathbf{B}, T)$. To refer to the value of an attribute A_i in tuple $r \in R$, we use the notation $r.A_i$, and we abbreviate $(A_1, A_2, \ldots, A_m)$ and $(r.A_1, r.A_2, ..., r.A_m)$ as A and $r.\mathbf{A}$, respectively.

We use a relational algebra that includes two temporal join operators: inner join $\bowtie^T$ and anti-join $\triangleright^T$. These temporal operators are generalizations of the standard relational join operators with an additional temporal constraint. We consider a conjunctive equality predicate θ (possibly empty) between tuples in R and tuples in S, and we write $r.\mathbf{C} = s.\mathbf{D}$ with $\mathbf{C} \subseteq \mathbf{A}$, $\mathbf{D} \subseteq \mathbf{B}$ as a shorthand for $r.A_i = s.B_j \wedge \ldots \wedge r.A_k = s.B_l$.

Definition 1 (Temporal Anti-Join). *Let R and S be two temporal relations with schemas $(\mathbf{A}, T)$ and $(\mathbf{B}, T)$, respectively, and a conjunctive equality predicate $\theta \equiv r.\mathbf{C} = s.\mathbf{D}$. The temporal anti-join, $R \triangleright_\theta^T S$, between R and S is defined as*

$$R \triangleright_\theta^T S = \{r' \mid \exists r \in R(r'.\mathbf{A} = r.\mathbf{A} \wedge r'.T \subseteq r.T \wedge \quad (1)$$
$$\nexists s \in S(r.\mathbf{C} = s.\mathbf{D} \wedge r'.T \cap s.T \neq \emptyset) \wedge \quad (2)$$
$$\forall T' \supset r'.T(\exists s \in S(r.\mathbf{C} = s.\mathbf{D} \wedge T' \cap s.T \neq \emptyset) \vee T' \not\subseteq r.T))\} \quad (3)$$

The first line ensures that tuple r' has the same values for all non-temporal attributes as tuple r and its timestamp is a sub-interval of $r.T$. The second line ensures that there

is no tuple in S that satisfies predicate θ and overlaps with the result tuple r'. The third line ensures that for any larger time interval T' than $r'.T$, either it is value-equivalent and overlapping to a tuple in S, or T' is not completely covered by the timestamp of r.

We can observe that the temporal anti-join $R \triangleright^T_\theta S$ needs to split the timestamp of the tuples $r \in R$, based on the timestamps of the tuples in S for which the predicate θ is satisfied. This produces tuples composed of the non-temporal attribute values of r and maximal sub-intervals of $r.T$ that do not overlap with tuples in S satisfying θ.

Example 2. Fig. 2 shows a graphical illustration of two temporal relations R (blue) with schema $(\mathbf{A}, T)$ and S (green) with schema $(\mathbf{B}, T)$. The timestamps are depicted as a horizontal lines in the time domain $\Omega^T = \{1, \ldots, 15\}$. The result of the temporal anti-join $R \triangleright^T_\theta S$ with $\theta \equiv r.A_1 = s.B_1 \wedge r.A_2 = s.B_2$ is the set of tuples $p_1, \ldots, p_5$ (red). For instance, p_1 and p_2 are derived from $r_1 \in R$, which matches with $s_1 \in S$. The timestamps of p_1 and p_2 are the maximal sub-intervals of $r_1.T$ for which no matching tuple in S exists. Similarly, p_3 and p_4 are derived from r_2. Tuple p_5 has the same timestamp as r_3 since r_3 has no matching tuple in S.

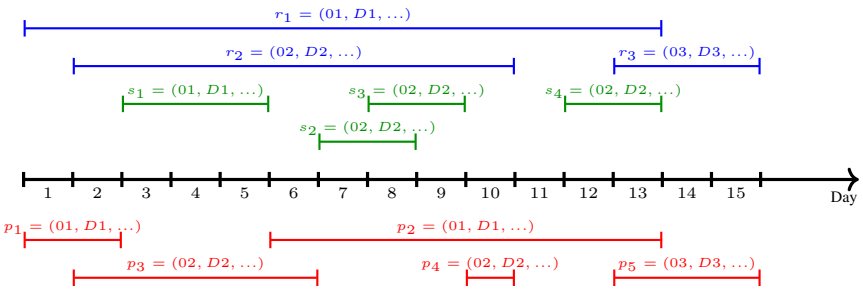


Fig. 2: Result Set of $R \triangleright^T_{R.\mathbf{C}=S.\mathbf{D}} S$

The focus of this paper is on the efficient parallel processing of temporal anti-joins with a conjunctive equality predicate by taking advantage of multi-core CPUs.

4 Proposed Solution

In this section, we present our 2-step solution for the efficient in-memory processing of temporal anti-joins with a conjunctive equality predicate: First, we introduce a new temporal primitive that computes the "complement" of a temporal relation with respect to the non-temporal join attributes. Second, we transform the temporal anti-join into an equivalent algebraic expression that contains a temporal join with the complement, which can be computed using parallel execution across multiple CPU cores.

4.1 Complement of a Temporal Relation

The first step applies a new temporal primitive, which computes the "time complement", $\overline{S}$, of a temporal relation S with respect to the non-temporal join attributes $\mathbf{D}$. This

primitive constructs, for each unique value combination of the attributes in $\mathbf{D}$, maximal time intervals that do not overlap with any tuple in S with the same values on $\mathbf{D}$.

Definition 2 (Complement of a Temporal Relation). *Let S be a temporal relation with schema $(\mathbf{B}, T)$ and let $\mathbf{D} \subseteq \mathbf{B}$ be a set of non-temporal attributes. The complement of S, denoted as $\overline{S}$, with respect to $\mathbf{D}$ is defined as*

$$\begin{aligned}\overline{S} = \{s' \mid\ & \exists s \in S(s'.\mathbf{D} = s.\mathbf{D} \wedge s'.T \subseteq \Omega^T \times \Omega^T) \wedge \\ & \nexists s \in S(s'.\mathbf{D} = s.\mathbf{D} \wedge s'.T \cap s.T \neq \emptyset) \wedge \\ & \forall T' \supset s'.T(\exists s \in S(s'.\mathbf{D} = s.\mathbf{D} \wedge T' \cap s.T \neq \emptyset \vee T' \not\subseteq \Omega^T \times \Omega^T))\}.\end{aligned}$$

The first two lines ensure that a tuple in the complement has the same values for the non-temporal attributes $\mathbf{D}$ as a tuple in S, but does not overlap with the timestamp of any of such tuples. The third line ensures that for any longer time interval T' covering $s'.T$, there is a value-equivalent tuple $s \in S$ that intersects with T', or T' goes beyond the considered time domain.

Example 3. Consider the temporal relation S in Fig. 3 (green) with schema $(\mathbf{B}, T)$. The non-temporal attributes in $\mathbf{D}$ are B_1 and B_2, and the time domain is from 1 to 15. The complement of S contains five tuples (orange). The tuples $\overline{s}_1$ and $\overline{s}_2$ are derived from s_1, and the tuples $\overline{s}_3$, $\overline{s}_4$, and $\overline{s}_5$ are derived from s_2, s_3, and s_4, which all have the same values for the non-temporal attributes B_1 and B_2.

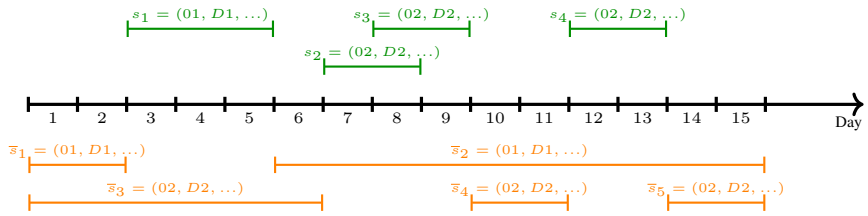


Fig. 3: Complement of Temporal Relation S

4.2 Transformation of the Temporal Anti-Join

The second step is to transform a temporal anti-join, $R \triangleright_\theta^T S$, into an equivalent algebraic expression with the help of the complement $\overline{S}$. The new expression is composed of two parts: a temporal join of R with $\overline{S}$ and the set of all tuples in R that have no matching tuples in S. Both parts can be processed in parallel on multiple CPU cores or threads.

Lemma 1. *Let R and S be two temporal relations with schemas $(\mathbf{A}, T)$ and $(\mathbf{B}, T)$, respectively, $\theta \equiv r.\mathbf{C} = s.\mathbf{D}$ be a conjunctive equality predicate over the non-temporal*

attributes $\mathbf{C} \subseteq \mathbf{A}$ *and* $\mathbf{D} \subseteq \mathbf{B}$*, and* $\overline{S}$ *be the complement of relation* S *wrt the attributes in* $\mathbf{D}$*. The temporal anti-join,* $R \triangleright_\theta^T S$*, can be expressed as*

$$R \triangleright_\theta^T S \equiv \prod_{R.\mathbf{A} \circ T} (R \bowtie_\theta^T \overline{S}) \cup \{r \in R \mid \nexists s \in S(r.\mathbf{C} = s.\mathbf{D})\},$$

where the temporal join $\bowtie_\theta^T$ *produces pairs of* θ*-matching tuples over maximally overlapping timestamps [19, 34, 4].*

Proof (Sketch). To show that our equivalence holds, we split the proof over two complete and disjoint partitions of relation R: (a) $\{r \in R \mid \exists s \in S(r.\mathbf{C} = s.\mathbf{D})\}$ and (b) $\{r \in R \mid \nexists s \in S(r.\mathbf{C} = s.\mathbf{D})\}$. We use $R' = \prod_{R.\mathbf{A} \circ R.T}(R \bowtie_\theta^T \overline{S})$ and $R'' = \{r \in R \mid \nexists s \in S(r.\mathbf{C} = s.\mathbf{D})\}$.

For (a) we have that a result tuple r' in Definition 1 takes the values for the non-temporal attributes $\mathbf{A}$ from a tuple $r \in R$ and the time interval of r' is contained or equal to the time interval of r (line 1). Since $\exists s \in S(r.\mathbf{C} = s.\mathbf{D})$, the timestamp of r' cannot overlap with the timestamp of any tuples with $r'.\mathbf{C} = s.\mathbf{D}$ (line 2). In R' we produce tuples (projected to $\mathbf{A}$) from tuples in R and $\overline{S}$ that satisfy $\mathbf{C} = \mathbf{D}$ using the intersection of intervals, thus we ensure the two conditions. All result tuples for $\{r \in R \mid \exists s \in S(r.\mathbf{C} = s.\mathbf{D})\}$ are produced, since $\overline{S}$ contains tuples for all values of $\mathbf{D}$ that exist in S. Since, $\overline{S}$ by definition contains all maximal time intervals for all values of $\mathbf{D}$ that exist in S, we have that any interval larger than the intersection between the interval of a tuple r and a tuple in $\overline{S}$ would either overlap a tuple in S with $\mathbf{C} = \mathbf{D}$ or extend beyond $r.T$ ensuring line 3 in Definition 1. R'' does not produce any results for partition $\{r \in R \mid \exists s \in S(r.\mathbf{C} = s.\mathbf{D})\}$.

For (b) we have that line 2 in Definition 1 evaluates to true, and the condition $\exists s \in S(r.\mathbf{C} = s.\mathbf{D} \wedge T' \cap s.T \neq \emptyset)$ in line 3 evaluates to false. Thus, a tuple r' takes the values for the non-temporal attributes $\mathbf{A}$ from a tuple $r \in R$, the time interval of r' is contained or equal to the time interval of r, and any larger time interval would extend beyond $r.T$, i.e., we have $r.\mathbf{A} = r'.\mathbf{A} \wedge r.T = r'.T$, and since $\nexists s \in S(r.\mathbf{C} = s.\mathbf{D})$, we have $\{r \in R \mid \nexists s \in S(r.\mathbf{C} = s.\mathbf{D})\} = R''$. R' does not produce any results for partition $\{r \in R \mid \nexists s \in S(r.\mathbf{C} = s.\mathbf{D})\}$, since $\overline{S}$ only contains values for $\mathbf{D}$ that are also in S and the join requires that $r.\mathbf{C} = s.\mathbf{D}$ exists. □

Lemma 1 shows how to transform a temporal anti-join, including a conjunctive equality predicate on non-temporal attributes, into an equivalent expression with a temporal join. The rationale behind this transformation lies in the fact that all operators in the transformed expression can be divided into subtasks that can be executed in parallel.

Example 4. Consider Fig. 4, which shows on top the temporal relation R (blue) and the complement $\overline{S}$ of relation S (orange) with $\mathbf{D} = \{B_1, B_2\}$. The result of the expression $\prod_{R.\mathbf{A} \circ T}(R \bowtie_{R.\mathbf{C}=S.\mathbf{D}}^T \overline{S})$ consists of the tuples p_1, p_2, p_3, and p_4. The tuple p_5 comes from the expression $\{r \in R \mid \nexists s \in S(r.\mathbf{C} = s.\mathbf{D})\}$, since tuple $r_3 = (03, D3, \dots)$ has not matching tuple in S.

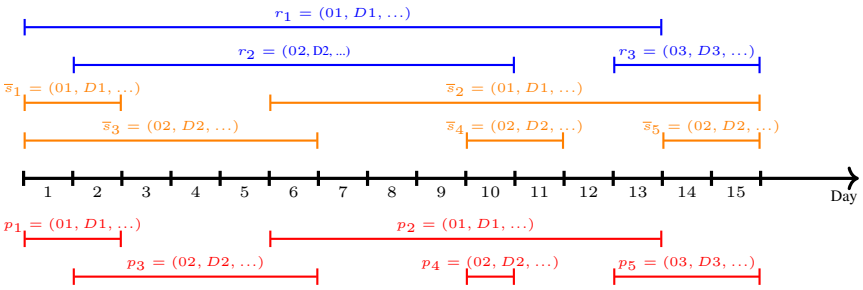


Fig. 4: Result of $R \triangleright^T_{R.\mathbf{C}=S.\mathbf{D}} S$ using the Complement

4.3 Parallel Processing of Temporal Anti-Join

Algorithm 1 computes the temporal anti-join $R \triangleright^T_{R.\mathbf{C}=S.\mathbf{D}} S$, taking advantage of parallel processing over multiple CPU cores. The algorithm begins by sorting the two input relations by the join attributes and the start timepoint. Then, for each of the two sorted relations a hash table is constructed: the key is composed of the join attributes, and the value is a pair representing the position of the first and the last tuple with the same values in the join attributes. This step is performed in parallel using Algorithm 2. Subsequently, Algorithm 3 is invoked to compute the complement of relation S with respect to the join attributes. Finally, Algorithm 5 computes the temporal join between relation R and the complement $\overline{S}$ of S projected to the nontemporal attributes of R and the intersection on T, including tuples from R that do not have tuples with matching join attribute values in $\overline{S}$. Notice that all steps allow a parallel execution.

Algorithm 1: TEMPORALANTIJOIN$(R, \mathbf{C}, S, \mathbf{D}, t)$

Input: Temporal relations R and S with join attributes $\mathbf{C}$ and $\mathbf{D}$, respectively, and number of threads t;
Output: Result of temporal anti-join $R \triangleright^T_{R.\mathbf{C}=S.\mathbf{D}} S$;

1 Sort relation R by the attributes $\mathbf{C}$ and start timepoint;
2 Sort relation S by the attributes $\mathbf{D}$ and start timepoint;
3 $h_R \leftarrow$ BORDERS$(R, \mathbf{C}, t)$; // Algorithm 2
4 $h_S \leftarrow$ BORDERS$(S, \mathbf{D}, t)$; // Algorithm 2
5 $(\overline{S}, h_{\overline{S}}) \leftarrow$ COMPLEMENT$(S, h_S, \mathbf{D}, t)$; // Algorithm 3
6 $res \leftarrow$ TEMPORALJOIN$(R, h_R, \overline{S}, h_{\overline{S}}, t)$; // Algorithm 5
7 **return** res;

Algorithm 2 computes the borders of unique values of the join attributes $\mathbf{C}$ in R in parallel. First, t threads are created together with a hash table. Then, each thread scans an equal-sized chunk of R. While scanning the chunk (lines 4–10), each thread j stores the positions in the chunk at which a unique set of values of the join attributes begins and ends. Once all threads have completed, the t hash-tables are merged into a single hash-table, named h_R (lines 13–16), which contains the positions at which a unique set of values of the join attributes begins and ends.

Algorithm 2: BORDERS$(R, \mathbf{C}, t)$

Input: Temporal relation R sorted by attributes $\mathbf{C}$ and start timepoint, and number of threads t;
Output: Hash table h_R with the positions in R where a set of value-equivalent tuples on $\mathbf{C}$ starts and ends;

```
1  create t threads;
2  create a hash table h_j for each thread j;
3  assign to each thread a chunk of R of size |R|/t;
4  foreach thread j do                                          // executed in parallel
5      first ← chunkStart;
6      for i ← chunkStart to chunkEnd − 1 do
7          if r_i.C ≠ r_{i+1}.C then
8              h_j[r_i.C] ← (first, i);
9              first ← i + 1;
10     h_j[r_{i+1}.C] ← (first, chunkEnd);
11 wait until all threads finish;                               // synchronization
12 h_R ← empty hash table;
13 foreach hash table h_j produced by the t threads do          // executed in single thread
14     foreach key ∈ h_j do
15         if key ∉ h_R then h_R[key] ← h_j[key] ;
16         else
               h_R[key] ← (min(h_R[key].start, h_j[key].start), max(h_j[key].end, h_R[key].end)) ;
17 return h_R;
```

Example 5. Consider relation S in Fig. 3. The join attributes are B_1 and B_2, resulting in the sorted relation $S = \{s_1, s_2, s_3, s_4\}$. The algorithm BORDERS with relation S in input computes the corresponding hash table $h_S = [((01, D1), (0, 0)), ((02, D2), (1, 3))]$. This indicates that there are two blocks of tuples: the first block with the join attribute values $(01, D1)$ contains the first tuple s_1, whereas the second block with join attribute values $(02, D2)$ contains the tuples s_2, s_3, and s_4.

Algorithm 3 computes the complement $\overline{S}$ and its associated hash table $h_{\overline{S}}$ for a relation S. In the first loop (lines 4–7), function PARTIALCOMPLEMENT is invoked in parallel to compute, for each combination of unique values of the join attributes $\mathbf{D}$, the number of tuples that will be in the complement $\overline{S}$; the result is stored in the cnt array. In the second step, once all slave threads have completed, the cnt array is used to determine the position pos in the output array $\overline{S}$, where to write the complement tuples (lines 9–11). This is required such that the complement tuples can be computed and written to $\overline{S}$ in parallel. In the third step (lines 12–20), the complement of relation S is computed. After initializing the data structures, the hash table $h_{\overline{S}}$ is filled and function PARTIALCOMPLEMENT is called in parallel to compute the complement $\overline{S}$.

Example 6. We continue our running example and call algorithm COMPLEMENT. The number of tuples in the complement relation for each combination of unique join attribute values is computed as $cnt = [2, 3]$, i.e., for the values $(01, D1)$ the complement contains two tuples and for the values $(02, D2)$ three tuples. Then, the positions in $\overline{S}$ are determined, where to write the complement tuples, as $pos = [0, 2]$. That is, the complement tuples for the join attribute values $(01, D1)$ are written to the positions 0–1, whereas the complement tuples for the values $(02, D2)$ are written to the positions 2–4.

Algorithm 4 implements two different functions depending on the variable $countFlag$. If the flag is true, the algorithm counts the number of complement tuples

Algorithm 3: COMPLEMENT$(S, h_S, \mathbf{D}, t)$

Input: Relation S sorted by $\mathbf{D}$ and start timepoint, hash table h_S, attribute set $\mathbf{D}$, and number of threads t;
Output: Sorted relation of $\overline{S}$ and hash table $h_{\overline{S}}$;

```
1  create t threads;
2  cnt ← array of size h_S;
3  i ← 0;
4  foreach key ∈ h_S do
5      wait for a thread to be available;
6      cnt[i] ← PARTIALCOMPLEMENT(S, h_S, null, null, key, true);
7      i ← i + 1;
8  wait until all threads finish;                          // synchronization
9  pos[0] ← 0;
10 for i ← 1 to h_S.size − 1 do
11     pos[i] ← pos[i − 1] + cnt[i − 1];
12 S̄ ← empty array of size pos[h_S.size − 1] + cnt[h_S.size − 1];
13 h_S̄ ← empty hash table of size h_S;
14 i ← 0;
15 foreach key ∈ h_S do
16     h_S̄[key].start ← pos[i];
17     h_S̄[key].end ← pos[i + 1] − 1;
18     wait for a thread to be available;
19     PARTIALCOMPLEMENT(S, h_S, S̄, h_S̄[key].start, key, false);
20     i ← i + 1;
21 wait until all threads finish;                          // synchronization
22 return (S̄, h_S̄);
```

for a chunk of tuples in S, which all have the same values key for the join attributes. Otherwise, the algorithm computes the complement tuples for the same chunk of tuples in S. The algorithm implements a sweep line approach. The sweep line (sl) is moved from the beginning of the time domain to the end. During this process, all tuples in the chunk of tuples in S from $h_S[key].start$ to $h_S[key].end$ are considered, which all have the same values for the join attributes $\mathbf{D}$. A new tuple is added to $\overline{S}$ whenever a temporal gap is detected, which is not covered by a tuple in the chunk.

It should be noted that if the union of time intervals of tuples with the same set of values of the non-temporal attributes in $S.\mathbf{D}$ spans the entire time domain, relation $\overline{S}$ does not contain any tuples related to this set of tuples in S, and $h_{\overline{S}}$ contains an invalid input with its start index greater than its end index for the specific key. This invalid input is checked in Algorithm 5 before producing the result tuples (line 13).

Algorithm 5 efficiently computes the temporal anti-join $R \triangleright^T_{R.\mathbf{C}=S.\mathbf{D}} S$ according to Lemma 1, tacking advantage of the hash tables and the complement relation $\overline{S}$. The basic idea is a merge join, reading the hash-tables h_R and $h_{\overline{S}}$ in key order until the hash table h_R is depleted (lines 5–20). In this process, we can distinguish three cases. If a key exists only in R (condition in line 6), all tuples in R matching key are added to the result, and key_R is set to the next key in key order (lines 7–8). Conversely, if a key exists only in $\overline{S}$ (line 10), no output tuple is produced for this key. The algorithm fetches the next key in key order from the hash table $h_{\overline{S}}$ (line 11). Finally, if the hash keys key_R and $key_{\overline{S}}$ are equal (line 12), we compute the temporal join between the two matching chunks of tuples from R and $\overline{S}$. The algorithm waits for an available thread and assigns the forward scan-based plane sweep algorithm *bguFS* from [8] to

Algorithm 4: PARTIALCOMPLEMENT$(S, h_S, \overline{S}, pos, key, countFlag)$

Input: Relation S sorted by $\mathbf{D}$ and start timepoint, hash table h_S, relation $\overline{S}$, position pos in $\overline{S}$ where to write the complement tuples, join attribute values key, and flag $countFlag$;
Output: If $countFlag$ is true, then number of tuples in $\overline{S}$ with $\mathbf{D} = key$, otherwise tuples in $\overline{S}$ computed for $\mathbf{D} = key$ starting from position pos.

```
1  j ← pos;
2  count ← 0;
3  sl ← beginning of time domain;
4  foreach tuple s_i ∈ S from h_S[key].start to h_S[key].end do
5      if sl < s_i.T_s then
6          if countFlag then
7              count ← count + 1
8          else
9              S̄[j] ← new tuple (key, [sl, s_i.T_s));
10             j ← j + 1;
11         sl ← s_i.T_e;
12     else
13         if sl < s_i.T_e then sl ← s_i.T_e ;
14 if sl < end of time domain then
15     if countFlag then
16         count ← count + 1
17     else
18         S̄[j] ← new tuple (key, [sl, end of time domain));
19         j ← j + 1;
20 if countFlag then return count;
21 else return 0 ;
```

that thread for execution. Once all threads complete their tasks, the result res of the temporal anti-join is returned.

5 Performance Evaluation

5.1 Setup and Data

All experiments were run on a server with 32 GB RAM and a 12 core CPU Intel(R) Xeon(R) Silver 4214R CPU clocked at 2.40 GHz running Linux. Hyper-threading allows to run up to 24 threads. All algorithms were implemented in C++, compiled using g++ (v11.4.0) with flags -O3, -mavx and -march=native. The code for *bguFS* proposed in [7, 8], (line 17 in Algorithm 5), was publicly available[1]. The source code for the algorithms implemented in this study is also publicly available[2].

We conducted the experiments using four real-world datasets: Taxis[3], Dig[4], Divvy[5], and Emergency[6]. Table 1 shows the main characteristics of these datasets. For each dataset, we randomly selected half of the data for the first input relation and used the other half for the second input relation.

[1] https://github.com/pbour/ijoin

[2] https://github.com/GiannisReppas/temporal_joins

[3] https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[4] https://data.cityofchicago.org/Transportation/Dig-Ticket-Notifications/cygx-ui4j

[5] https://data.cityofchicago.org/Transportation/Divvy-Trips/fg6s-gzvg

[6] https://data.cityofnewyork.us/browse?category=Health

Algorithm 5: TEMPORALJOIN$(R, h_R, \overline{S}, h_{\overline{S}}, t)$

Input: Relation R sorted by attributes $\mathbf{C}$ and start timepoint, hash table h_R, relation $\overline{S}$ sorted by attributes $\mathbf{D}$ and start timepoint, hash table $h_{\overline{S}}$, and number of threads t;
Output: Result of $R \triangleright^{T}_{R.\mathbf{C}=S.\mathbf{D}} S$;

```
1  create t threads;
2  res ← ∅;
3  key_R ← first key in h_R in key order;
4  key_S̄ ← first key in h_S̄ in key order;
5  while h_R not depleted do
6      if h_S̄ depleted or key_R < key_S̄ then
7          res ← res ∪ {tuples from R[h_R[key_R].start] to R[h_R[key_R].end]};
8          key_R ← next key in h_R in key order;
9      else
10         if key_R > key_S̄ then
11             key_S̄ ← next key in h_S̄ in key order;
12         else
13             if h_R[key_R].start ≤ h_S̄[key_S̄].end then
14                 R' ← tuples from R[h_R[key_R].start] to R[h_R[key_R].end];
15                 S̄' ← tuples from S̄[h_S̄[key_S̄].start] to S̄[h_S̄[key_S̄].end];
16                 wait for a thread to be available;
17                 res ← res ∪ bguFS(R', S̄');
18             key_R ← next key in h_R in key order;
19             key_S̄ ← next key in h_S̄ in key order;
20 wait until all threads finish;                          // synchronization
21 return res;
```

We compare our solution against a main memory implementation of DIP [9], which is the state-of-the-art for computing temporal anti-joins. Since DIP does not support equality predicates, we extended it similar to our solution using ordered hash tables.

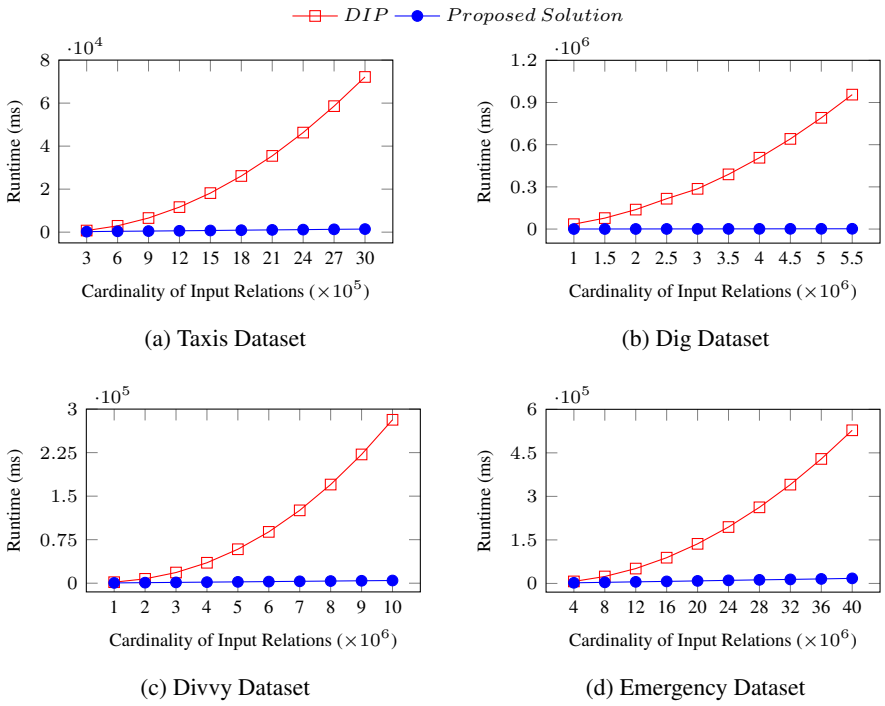
Table 1: Characteristics of Real-World Datasets

Dataset	**Taxis**	**Dig**	**Divvy**	**Emergency**
Cardinality	7,696,617	13,045,430	21,242,740	82,021,561
Domain Size	2,744,641,998	183,497,400	207,319,715	87,177,600
Shortest Interval	1	9,000	60	172,800
Longest Interval	2,618,881	79,867,800	13,453,220	87,177,600
Average Interval	995	3,142,491	1,151	29,449,118

5.2 Evaluation results

Fig. 5 shows the runtime for the single-threaded processing of a temporal anti-join for varying input cardinalities for the various datasets. Our proposed solution outperforms DIP in terms of query processing time. The reason is that our proposed solution employs a simple forward scanning plane sweep algorithm to find the borders of unique values for the non-temporal attributes of input relations involved in the conjunctive equality predicate, construct the complement of a temporal relation, and process the temporal

algebraic equivalent expression (cf. Lemma 1). In contrast, DIP partitions one input relation into the smallest possible number of partitions, each of which stores tuples with non-overlapping time intervals, and then applies a sequence of merges to these partitions. This process in DIP requires more time compared to our solution.



(a) Taxis Dataset

(b) Dig Dataset

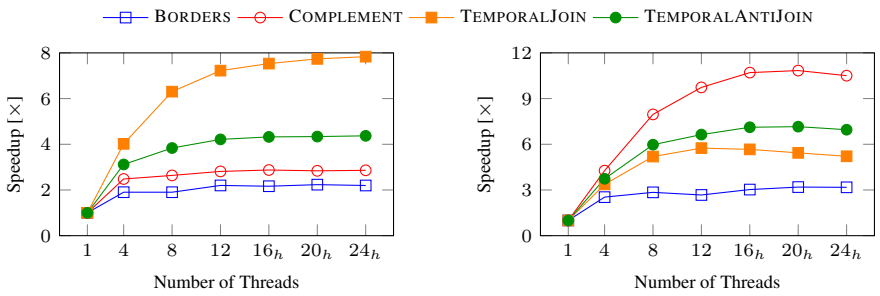
(c) Divvy Dataset

(d) Emergency Dataset

Fig. 5: Temporal Anti-Join Query Processing using Single Thread

Fig. 6 shows the speed-up depending on the number of threads achieved by each of the parallel algorithms presented in Section 4.3. We used the full datasets and varied the number of available parallel threads in the set $\{1, 4, 8, 12, 16_h, 20_h, 24_h\}$, where the subscript "$h$" indicates the activation of hyper-threading.

In general, we can see that all proposed algorithms for in-memory computation of subtasks related to processing a temporal anti-join experience performance improvements with the use of more threads. Furthermore, the general behavior of these algorithms indicates that performance is enhanced up to 12 threads in most cases. Beyond that point, a slight, yet stable improvement in terms of speedup is still achieved with the utilization of hyper-threading in most cases. As shown in Fig. 6, the algorithms designed for constructing the complement of a temporal relation (i.e., Algorithms 3 and 4) and processing the equivalent expression of a temporal anti-join (i.e., Algorithm 5), consistently offer the best speedup. The reason is that Algorithms 3 and 5 assign a slave thread for each group of tuples having the same value for the non-temporal attributes in-



(a) Taxis Dataset

(b) Dig Dataset

(c) Divvy Dataset

(d) Emergency Dataset

Fig. 6: Speedup in Temporal Anti-Join Query Processing

volved in the join predicates. Therefore, it is expected that these algorithms can achieve a higher speedup when the dataset contains groups with similar sizes, as seen in datasets like Dig, Divvy, and Emergency. However, in cases where such groups vary significantly in size, as in the complement construction for the Taxis dataset, containing over 4,000 different non-temporal values with substantial variations among them, a smaller speedup is expected. In contrast, the proposed algorithm to find borders for unique values of the non-temporal attributes in the join predicate (i.e., Algorithm 2) demonstrates a stable speed-up in most cases as the number of threads increases, albeit without significant growth. This is because, despite the effort to distribute the load between all threads evenly by assigning chunks of the relation with the same size to each thread, the merging of the local hash tables, along with key updates, nullifies the effects of equal load distribution. Finally, we observe that our proposed solution for processing a temporal anti-join, which employs all algorithms from Section 4.3, demonstrates scalability with increasing number of threads. In our setting, it achieved a speed-up of 4.36 for the Taxis dataset, 7.15 for the Dig dataset, 13.69 for the Divvy dataset, and 9.01 for the Emergency dataset.

6 Conclusion and Future Works

In this paper, we focused on an efficient and scalable in-memory computation of temporal anti-joins. We introduced the complement of a temporal relation as a new temporal primitive to transform a temporal anti-join, including conjunctive equality predicates on non-temporal attributes, into an equivalent algebraic expression, which contains a temporal inner join. The processing of the equivalent expression can be decomposed into smaller subtasks, which, utilizing multiple CPU cores, can be executed in parallel. In a single-threaded evaluation, our experimental results demonstrated that the proposed solution outperforms the state-of-the-art approach DIP. In a multi-threaded setting, depending on the characteristics of the datasets, our proposed solution achieves high speed-ups with an increasing number of available CPU cores or threads.

As a future work, we plan to extend our proposed solution by optimizing the algorithm for finding borders of a temporal relation in parallel, aiming to construct hash tables faster and consume less memory. Additionally, we plan to study the processing of temporal outer and anti-joins with disjunctive equality predicates on non-temporal attributes in an efficient and scalable manner.

Acknowledgment

This work was partially supported by hessian.AI at TU Darmstadt and DFKI Darmstadt as well as by a grant from the Autonomous Province of Bozen-Bolzano "Research Südtirol/Alto Adige 2019" (project ISTeP).

References

1. Arge, L., Procopiuc, O., Ramaswamy, S., Suel, T., Vitter, J.S.: Scalable sweeping-based spatial join. In: VLDB. p. 570–581. Morgan Kaufmann (1998)
2. Becker, B., Gschwind, S., Ohler, T., Seeger, B., Widmayer, P.: An asymptotically optimal multiversion b-tree. The VLDB Journal **5**, 264–275 (1996)
3. Blanas, S., Li, Y., Patel, J.M.: Design and evaluation of main memory hash join algorithms for multi-core cpus. In: SIGMOD. pp. 37–48. ACM (2011)
4. Böhlen, M.H., Dignös, A., Gamper, J., Jensen, C.S.: Temporal data management - an overview. In: eBISS. LNBIP, vol. 324, pp. 51–83. Springer (2017)
5. Böhlen, M.H., Dignös, A., Gamper, J., Jensen, C.S.: Database technology for processing temporal data (invited paper). In: TIME. LIPIcs, vol. 120, pp. 2:1–2:7. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018)
6. Bouros, P., Lampropoulos, K., Tsitsigkos, D., Mamoulis, N., Terrovitis, M.: Band joins for interval data. In: EDBT. pp. 443–446 (2020)
7. Bouros, P., Mamoulis, N.: A forward scan based plane sweep algorithm for parallel interval joins. In: VLDB. pp. 1346–1357. ACM (2017)
8. Bouros, P., Mamoulis, N., Tsitsigkos, D., Terrovitis, M.: In-memory interval joins. The VLDB Journal **30**, 667–691 (2021)
9. Cafagna, F., Böhlen, M.H.: Disjoint interval partitioning. The VLDB Journal **26**, 447–466 (2017)
10. Christodoulou, G., Bouros, P., Mamoulis, N.: Hint: A hierarchical index for intervals in main memory. In: SIGMOD. pp. 1257–1270. ACM (2022)

11. Christodoulou, G., Bouros, P., Mamoulis, N.: Hint: A hierarchical interval index for allen relationships. The VLDB Journal (2023)
12. Dignös, A., Böhlen, M.H., Gamper, J.: Temporal alignment. In: SIGMOD. pp. 433–444. ACM (2012)
13. Dignös, A., Böhlen, M.H., Gamper, J.: Query time scaling of attribute values in interval timestamped databases. In: ICDE. pp. 1304–1307 (2013)
14. Dignös, A., Böhlen, M.H., Gamper, J.: Overlap interval partition join. In: SIGMOD. pp. 1459–1470. ACM (2014)
15. Dignös, A., Böhlen, M.H., Gamper, J., Jensen, C.S.: Extending the kernel of a relational dbms with comprehensive support for sequenced temporal queries. ACM Trans. Database Syst. **41**(4), 26:1–26:46 (2016)
16. Dignös, A., Böhlen, M.H., Gamper, J., Jensen, C.S., Moser, P.: Leveraging range joins for the computation of overlap joins. VLDB J. **31**(1), 75–99 (2022)
17. Dignös, A., Glavic, B., Niu, X., Gamper, J., Böhlen, M.H.: Snapshot semantics for temporal multiset relations. Proc. VLDB Endow. **12**(6), 639–652 (2019)
18. Enderle, J., Hampel, M., Seidl, T.: Joining interval data in relational databases. In: SIGMOD Conference. pp. 683–694. ACM (2004)
19. Gao, D., Jensen, C.S., Snodgrass, R.T., Soo, M.D.: Join operations in temporal databases. The VLDB Journal **14**, 2–29 (2005)
20. Gunadhi, H., Segev, A.: Query processing algorithms for temporal intersection joins. In: ICDE. pp. 336–344. IEEE Computer Society (1991)
21. Hu, X., Sintos, S., Gao, J., Agarwal, P.K., Yang, J.: Computing complex temporal join queries efficiently. In: SIGMOD. p. 2076–2090. ACM (2022)
22. Jensen, C.S., Jensen: Temporal data management. IEEE Transactions on knowledge and data engineering **11**(1), 36–44 (1999)
23. Kaufmann, M., Manjili, A.A., Vagenas, P., Fischer, P.M., Kossmann, D., Färber, F., May, N.: Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In: SIGMOD. pp. 1173–1184. ACM (2013)
24. Kriegel, H., Pötke, M., Seidl, T.: Managing intervals efficiently in object-relational databases. In: VLDB. pp. 407–418. Morgan Kaufmann (2000)
25. Kulkarni, K.G., Michels, J.: Temporal features in SQL: 2011. SIGMOD Rec. **41**(3), 34–43 (2012)
26. Leung, T.C., Muntz, R.R.: Temporal query processing and optimization in multiprocessor database machines. In: VLDB. pp. 383–394. Morgan Kaufmann (1992)
27. Mirabi, M., Fathi, L., Dignös, A., Gamper, J., Binnig, C.: A new primitive for processing temporal joins. In: SSTD'23. p. 106–109. ACM (2023)
28. Pfoser, D., Jensen, C.S.: Incremental join of time-oriented data. In: SSDBM. pp. 232–243. Computer Society (1999)
29. Piatov, D., Helmer, S., Dignös, A.: An interval join optimized for modern hardware. In: ICDE. pp. 1098–1109. IEEE Computer Society (2016)
30. Piatov, D., Helmer, S., Dignös, A., Persia, F.: Cache-efficient sweeping-based interval joins for extended allen relation predicates. VLDB J. **30**(3), 379–402 (2021)
31. Raigoza, J., Sun, J.: Temporal join processing with hilbert curve space mapping. In: SAC. pp. 839–844. ACM (2014)
32. Segev, A., Gunadhi, H.: Event-join optimization in temporal relational databases. In: VLDB. pp. 205–215. Morgan Kaufmann (1989)
33. Soo, M.D., Snodgrass, R.T., Jensen, C.S.: Efficient evaluation of the valid-time natural join. In: ICDE. pp. 282–292. IEEE Computer Society (1994)
34. Zhang, D., Tsotras, V.J., Seeger, B.: Efficient temporal join processing using indices. In: ICDE. pp. 103–113. IEEE Computer Society (2002)