Computing Complex Temporal Join Queries Efficiently*

Xiao Hu
xh102@cs.duke.edu
Duke University
NC, USA

Stavros Sintos
sintos@uchicago.edu
University of Chicago
IL, USA

Junyang Gao
jygao@google.com
Google
NY, USA

Pankaj K. Agarwal
pankaj@cs.duke.edu
Duke University
NC, USA

Jun Yang
junyang@cs.duke.edu
Duke University
NC, USA

ABSTRACT

This paper studies multi-way join queries over temporal data, where each tuple is associated with a *valid time* interval indicating when the tuple is valid. A temporal join requires that joining tuples' valid intervals intersect. Previous work on temporal joins has focused on joining two relations, but pairwise processing is often inefficient because it may generate unnecessarily large intermediate results. This paper investigates how to efficiently process complex temporal joins involving multiple relations. We also consider a useful extension, *durable temporal joins*, which further selects results with long enough valid intervals so they are not merely transient patterns.

We classify temporal join queries into different classes based on their computational complexity. We identify the class of *r-hierarchical* joins and show that a linear-time algorithm exists for a temporal join if and only it is r-hierarchical (assuming the 3SUM conjecture holds). We further propose output-sensitive algorithms for non-r-hierarchical joins. We implement our algorithms and evaluate them on both synthetic and real datasets.

CCS CONCEPTS

• **Theory of computation → Database query processing and optimization (theory)**; • **Information systems → Database query processing**.

KEYWORDS

temporal database, join queries, durable temporal joins

ACM Reference Format:
Xiao Hu, Stavros Sintos, Junyang Gao, Pankaj K. Agarwal, and Jun Yang. 2022. Computing Complex Temporal Join Queries Efficiently. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3514221.3517893

*This work was supported by NSF awards IIS-1814493, CCF-2007556, and IIS-2008107.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9249-5/22/06.
https://doi.org/10.1145/3514221.3517893

1 INTRODUCTION

Temporal data analysis [25, 35, 51, 52, 76] is an essential feature of modern database systems, as most of the data we encounter in practice are temporal in nature—from business transactions and social interactions to system logs and observations of natural phenomena. Temporal joins are fundamental to temporal data analysis. Consider a temporal database where each tuple is associated with a *valid(-time) interval*, which indicates when the tuple is "valid." A temporal join finds tuples that together satisfy, in an addition to the join condition on their non-temporal attributes, the implicit temporal join condition that the intersection of their valid intervals is non-empty. To illustrate, consider the following example.

Example 1. Consider a toy database storing the collaboration network among authors shown in Figure 1. Vertices represent authors and edges represent collaborations between authors. For example, tuple (B, C) with valid interval $[2011, 2015]$ indicates that authors B and C collaborated over this five-year time period. In practice, such a database may be extracted and constructed from the DBLP [1] dataset. A temporal join (involving three copies of the edge relation) can find a chain of four authors connected by three pairwise collaborations simultaneously at some point in time, e.g., (A, B, C, D) with collaborations happening simultaneously during 2013–2015. In contrast, a non-temporal join would find a non-answer (A, E, B, D), because even though each of the three collaborations existed at some point, they never took place simultaneously: the valid intervals of (A, E) and (B, D) do not intersect.

Beyond Binary Temporal Joins. Work on temporal joins to date has mostly focused on processing binary joins, or efficiently joining two relations at a time [41, 77]. The drawback of this approach is that for complex joins involving multiple relations, such as the example above, performing a sequence of binary joins may produce huge intermediate results, even though the final result may be small in size. Ideally, we would like the overall algorithm to run in time near linear in the input data size and linear in the final result size. In recent years, there have been promising results on efficiently processing non-temporal joins involving multiple relations [17, 65, 86]. A natural question is whether we can obtain similar results for temporal joins as well. However, techniques for non-temporal join processing fail to deliver in this case because they handle equality join conditions involving non-temporal attributes, while leaving out the temporal join condition involving the valid intervals. A simple *join-first approach*, which applies these techniques first to compute the (non-temporal) join result and the filters it using the temporal

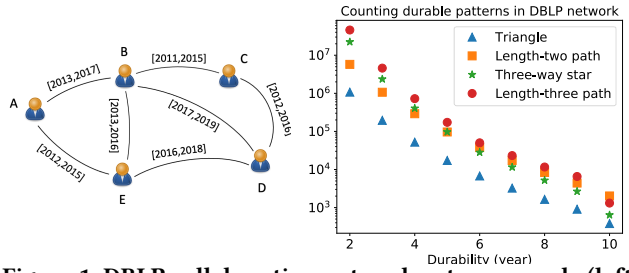

Figure 1: DBLP collaboration network: a toy example (left), and statistics of durable patterns (right).

join condition, runs the risk of producing intermediate results that are much bigger. To fill the gap in existing temporal join algorithms, we take the first step in investigating the hardness of temporal joins involving multiple relations, and propose a framework that handles temporal constraints head-on in join evaluation.

Durable Temporal Joins. Beyond joining multiple relations, we also consider other ways of enriching temporal join queries. One such enrichment is to add a final "durability" check to temporal joins. A temporal join computes the intersection of joining tuples' valid intervals as the valid interval of the result tuple. A very short result interval, however, implies that the result tuple is valid only briefly; it may represent more of a transient glitch than a robust pattern. Many data analysis tasks are thus more interested in *durable temporal joins*, which return only a result tuple if the length of its valid interval—which we call *durability*—is above some threshold τ specified as part of the query. To illustrate the practical use of durability analysis, consider the following example.

Example 2. We take a subset of the DBLP dataset pertaining to the "`inproceedings`" entries and convert it to a temporal coauthorship graph as in Example 1. This graph has 1,764,475 vertices and 9,460,140 edges, each labeled with a valid interval. An interesting exploratory analysis can be done with a variety of temporal joins designed to look for different coauthorship patterns among authors, including length-2 paths (a–b and b–c), length-3 paths (a–b, b–c, and c–d), 3-way stars (a–b, a–c, and a–d), and triangles (a–b, b–c, and c–a). The length of the valid interval of a join result tuple corresponds to the durability of the pattern it represents. Figure 1 counts the number of such patterns in the entire graph at different durability threshold levels. Each data point in this figure can be obtained by computing a durable temporal join with a desired threshold τ and counting the number of result tuples.

Once again, we are interested in efficient algorithms with running times linear in the final result size. This requirement rules out the naive join-first approach of computing the full temporal join and then filtering the intermediate result to obtain the durable tuples. Indeed, as Figure 1 illustrates, high durability thresholds lead to results that are many orders of magnitudes smaller than those of full temporal joins (which are equivalent to durable temporal joins with a trivial threshold 0). We show in this paper how to avoid the curse of large intermediate result size using a remarkably simple transformation of the input data, which then allows us to leverage our efficient temporal join algorithms to compute the results of durable temporal joins directly.

x	y	valid interval
A	B	$[2013, 2017]$
A	E	$[2012, 2015]$
B	C	$[2011, 2015]$
B	D	$[2017, 2019]$
B	E	$[2013, 2016]$
C	D	$[2012, 2016]$
D	E	$[2016, 2018]$

copies of $R_1(x_1, x_2)$, $R_2(x_2, x_3)$, $R_3(x_3, x_4)$

x_1	x_2	x_3	x_4	valid interval
A	B	C	D	$[2013, 2015]$
A	B	D	E	$[2017, 2017]$
B	C	D	E	$--$

join → $Q(\mathcal{R})$

Figure 2: Temporal database and temporal join query. *The left table is a temporal relation capturing the collaboration graph in Figure 1. We consider the directed version of edges in alphabetic ordering for simplicity. Each tuple corresponds to an edge. By making three copies of this temporal relation and renaming the attributes, we obtain a temporal instance* $\mathcal{R} = \{R_1(x_1, x_2), R_2(x_2, x_3), R_3(x_3, x_4)\}$. *The right table is the join result of temporal query* $Q = (\mathcal{V}, \mathcal{E})$ *over* $\mathcal{R}$, *where* $\mathcal{V} = \{x_1, x_2, x_3, x_4\}$ *and* $\mathcal{E} = \{\{x_1, x_2\}, \{x_2, x_3\}, \{x_3, x_4\}\}$, *finding all length-3 paths (vertices are in alphabetic ordering) in this graph.* (B, C, D, E) *is a valid non-temporal join result but not temporal join result, since it does not have a valid interval.*

In the remainder of this paper, we introduce our framework for systematic study of the evaluation of temporal joins involving multiple relations. We classify temporal join queries into different classes based on their computational complexity. We design efficient algorithms for these query classes, some of which are provably optimal. We also provide an experimental evaluation of our proposed algorithms over both synthetic and real-life datasets.

2 MODEL AND RESULTS

2.1 Problem Definition

Non-temporal Join. A (natural) join can be modeled as a hypergraph $Q = (\mathcal{V}, \mathcal{E})$ [18], where the set of vertices $\mathcal{V} = \{x_1, x_2, \cdots, x_n\}$ models the *attributes* and the set of hyperedges $\mathcal{E} = \{e_1, e_2, \ldots, e_m\} \subseteq 2^{\mathcal{V}}$ models the *relations*. Some examples are illustrated in Figure 3.

For each attribute $x \in \mathcal{V}$, let $\mathrm{dom}(x)$ denote its *domain*. For a subset of attributes $e \subseteq \mathcal{V}$, let $\mathbb{A}_e = \bigtimes_{x \in e} \mathrm{dom}(x)$. We call each element $\mathbf{a}$ of $\mathbb{A}_e$ a *tuple*, and we call e the *support* of $\mathbf{a}$, denoted by $\mathrm{supp}(\mathbf{a})$. For $e \subseteq \mathcal{V}$, a *relation* R_e over $\mathbb{A}_e$ is a set of tuples from $\mathbb{A}_e$, each representing an assignment of a value from $\mathrm{dom}(x)$ to x for each $x \in e$. We assume that all tuples in a relation are distinct. For a tuple $\mathbf{a} \in \mathbb{A}_e$ and a subset of the attributes $e' \subseteq e$, let $\pi_{e'}(\mathbf{a})$ denote the *projection* of $\mathbf{a}$ onto the subspace spanned by e'.

An *input instance* or *database* of Q is a set of relations $\mathcal{R} = \{R_e \mid e \in \mathcal{E}\}$, where each R_e is a relation over $\mathbb{A}_e$. The result of the *(non-temporal) join* of Q on $\mathcal{R}$, noted as $Q(\mathcal{R})$, is defined as

$$Q(\mathcal{R}) = \{\mathbf{a} \in \mathbb{A}_{\mathcal{V}} \mid \forall e \in \mathcal{E}, \exists \mathbf{a}_e \in R_e : \pi_e(\mathbf{a}) = \mathbf{a}_e\}. \quad (1)$$

i.e., all combinations of tuples from relations in $\mathcal{R}$, such that tuples in each combination have the same value(s) on common attribute(s).

Temporal Join. A *temporal input instance* or *database* of Q further associates each tuple $\mathbf{a}$ in a relation of $\mathcal{R}$ with an interval $I_{\mathbf{a}} = [t^-_{\mathbf{a}}, t^+_{\mathbf{a}}]$, called the *valid interval* of $\mathbf{a}$. To show both $\mathbf{a}$ and its valid interval $I_{\mathbf{a}}$ explicitly, we will use the notation $\langle \mathbf{a}, I_{\mathbf{a}} \rangle$. The *temporal join* of Q on $\mathcal{R}$ consists of those tuples $\mathbf{a} \in \mathbb{A}_{\mathcal{V}}$ that are returned by the non-temporal join defined in (1) and additionally satisfy the condition $I_{\mathbf{a}} = \bigcap_{e \in \mathcal{E}} I_{\pi_e(\mathbf{a})} \neq \emptyset$; $I_{\mathbf{a}}$ is associated with $\mathbf{a}$ as its valid interval in the output. To support joins between temporal

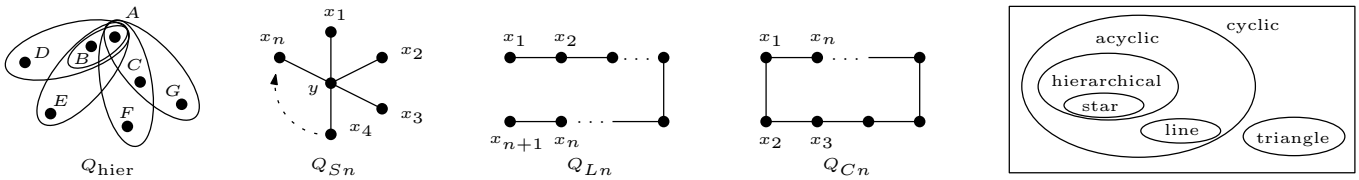

Figure 3: Hypergraphs of join queries: *(1)* $Q_{hier} = R_1(A,B) \bowtie R_2(A,B,D) \bowtie R_3(A,B,E) \bowtie R_4(A,C,F) \bowtie R_5(A,C,G)$*; (2) star join* $Q_{Sn} = R_1(x_1,y) \bowtie R_2(x_2,y) \bowtie \cdots \bowtie R_n(x_n,y)$*; (3) line join* $Q_{Ln} = R_1(x_1,x_2) \bowtie R_2(x_2,x_3) \bowtie \cdots \bowtie R_n(x_n,x_{n+1})$ *for* $n \geq 3$*; (4) cycle join* $Q_{Cn} = R_1(x_1,x_2) \bowtie R_2(x_2,x_3) \bowtie \cdots \bowtie R_{n-1}(x_{n-1},x_n) \bowtie R_n(x_n,x_1)$ *and triangle join* $Q_\triangle = Q_{C3}$*.* **Classification of join queries:** *line, star, triangle, hierarchical, acyclic and cyclic join.* Q_{hier} *and* Q_{Sn} *are hierarchical,* Q_{Ln} *(*$n \geq 3$*) is non-hierarchical but acyclic, and* Q_{Cn} *is cyclic.*

and non-temporal relations, we can simply set $I_{\mathbf{a}} = (-\infty, \infty)$ of all tuples $\mathbf{a}$ in a non-temporal relation. We focus on the non-empty intersection of valid intervals as the temporal join condition.

Let $N = \sum_{e \in \mathcal{E}} |R_e|$ be the input size of $\mathcal{R}$. Let $K = |Q(\mathcal{R})|$ be the output size of the temporal join. We study the *data complexity* of join algorithms, i.e., their running time in terms of N and K; we assume the size of Q to be bounded by a constant. An algorithm for computing temporal join is *linear* if its running time is $O(N+K)$, and *near-linear* if the running time is $O((N+K)\,\text{polylog}(N))$. Note that every algorithm for computing $Q(\mathcal{R})$ must spend $\Omega(N+K)$ time, to read every input tuple once and to report every result.

Remarks on Other Temporal Join Models. First, our proposed solution can be extended to the settings where each tuple is associated with a *set* of disjoint intervals, which arise when the same tuple can be inserted and deleted multiple times, or when projection causes distinct tuples to coalesce.

Second, our solution can be applied to the *τ-durable temporal join* for a parameter $\tau \geq 0$, which is the subset of temporal join result tuples whole durability is at least τ. It should be noted that a temporal join is simply an instance of the τ-durable join with $\tau = 0$, where the durability criterion is trivially satisfied. On the other hand, the general τ-durable temporal join of Q on $\mathcal{R}$ is equivalent to the temporal join of Q on $\mathcal{R}_\tau$, where $\mathcal{R}_\tau$ is a temporal instance derived from $\mathcal{R}$ using a simple "shrinking" transformation: each tuple $\mathbf{a}$ in $\mathcal{R}$ has its valid interval $[t^-_{\mathbf{a}}, t^+_{\mathbf{a}}]$ shrunk to $[t^-_{\mathbf{a}} + \frac{\tau}{2}, t^+_{\mathbf{a}} - \frac{\tau}{2}]$ (and removed if this interval is empty). The transformed instance $\mathcal{R}_\tau$ can be derived from $\mathcal{R}$ in $O(N)$ time; we can then directly apply our temporal join algorithms.

More generally, a broad class of temporal predicates can be reformulated in terms of the non-empty intersection of valid intervals by transforming the valid interval appropriately in the query procedure. We give three examples below, and more general applications of this overlap model are very interesting, but left as future work.

- For instance-stamped data, one looks for joining tuples whose valid timestamps lie within τ of all others. We can support such a query by transforming each valid timestamp t to interval $[t - \frac{\tau}{2}, t + \frac{\tau}{2}]$ and answer the query as a (0-durable) temporal join query on the interval-stamped data.
- For interval-stamped data, one looks for all pairs of joining tuples, where the first leads the second with a gap of at least τ. We can support such a query by transforming each intervals $t = [t^-, t^+]$ to $[t^+, +\infty)$ in the first relation and to $(-\infty, t^-]$ in the second relation, and answer the query as a τ-durable temporal join.
- For interval-stamped data, one may look for a triangle $(a,b,c) \in R_1(A,B) \bowtie R_2(B,C) \bowtie R_3(A,C)$ where *relative positioning* of the three edge intervals follows the pattern given by three intervals I_1, I_2, I_3 (possibly non-overlapping); more precisely, there exists some time shift Δ such that $I_{ab} + \Delta \subseteq I_1$, $I_{bc} + \Delta \subseteq I_2$, and $I_{ac} + \Delta \subseteq I_3$. We can support such a query by transforming intervals $t \in R_1$ into $[t^- - I_1^-, t^+ - I_1^+]$, $t \in R_2$ into $[t^- - I_2^-, t^+ - I_2^+]$, $t \in R_3$ into $[t^- - I_3^-, t^+ - I_3^+]$, and answer it as a (0-durable) temporal join query on the transformed data.

2.2 Classes of Join Queries

We introduce two important classes of join queries (see Figure 3), which are frequently used in this paper.

Acyclic joins [23]. A join query Q is *acyclic* if the hypergraph Q is acyclic, as defined by Beeri et al. [23] (called α-acyclicity in [37]). There are several equivalent notions of acyclic joins, and we use the one based on *join tree*: Q is acyclic if there exists a tree $\mathcal{T}$, called a *join tree* of Q, with the set $\mathcal{E}$ of nodes such that for any $x \in \mathcal{V}$, all nodes of $\mathcal{T}$ containing x form a connected subtree of $\mathcal{T}$.

Hierarchical joins [32]. An interesting subclass of acyclic join is *hierarchical join*, defined as follows. A join query $Q = (\mathcal{V}, \mathcal{E})$ is *hierarchical* if for every pair of vertices x, y, $\mathcal{E}_x \subseteq \mathcal{E}_y$, $\mathcal{E}_y \subseteq \mathcal{E}_x$, or $\mathcal{E}_x \cap \mathcal{E}_y = \emptyset$, where $\mathcal{E}_z = \{e \in \mathcal{E} : z \in e\}$. Efficient algorithms have been developed for hierarchical joins in probabilistic databases [32, 38] and dynamic query processing [24].

2.3 Our Contribution

Our theoretical results are summarized in Figure 4. In particular:

- **Time-first Approach.** Corresponding to the join-first approach, we present a *time-first approach*, which was also known as *sweep-plane-based algorithm* in the literature [20]. Intuitively, it sorts the endpoints of input tuples first, virtually sweeps a time axis, and computes the join results intersecting with this axis. Using this framework, we can obtain a near-linear algorithm for hierarchical temporal joins, by designing an efficient data structure over the very special query structures, and an output-sensitive algorithm for general temporal joins, by resorting to an output-sensitive non-temporal join algorithm. (**Section 3**)
- **Hybrid Approach.** To further improve general temporal joins, we propose a *hybrid approach* as a combination of *join-first* and *time-first* approaches. The complexity of this hybrid approach depends on two query-dependent quantities: the fractional hypertree width [43], measuring how far the join query is from being acyclic, and *hierarchical hypertree width*, measuring how far the join query is from being hierarchical. Moreover, we present a few simplification and improvement on some specific class of

Join Queries	Non-Temporal Join	Temporal Join
Hierarchical	$O(N+K)$ [86]	$O(N \cdot \log N + K)$ [Theorem 6]
Acyclic		$O\left(N^{\min\{\text{fhtw}+1,\text{hhtw}\}} + K\right)$ [Theorem 12]
General	$O(N^{\rho})$ [65, 66, 80] $O(N^{\text{fhtw}} + K)$ [43] $O(N^{\text{subw}} + K)$ [17]	

Figure 4: Summary of Results. *N is the input size. K is the output size of join results. ρ is the optimal fractional edge covering number of the query; subw is the sub-modular width of the query; fhtw is the fractional hypertree width of the query; hhtw is the hierarchical hypertree width of the query defined in Section 3.3.*

temporal joins. At last, we provide a guideline for this unified framework, on how to choose the best evaluation strategy for a temporal join, which depends on the query structure. (**Section 4**)

- **Hardness.** We show two hardness results for temporal joins. Firstly, any temporal join query can be reduced to its *non-temporal counterpart*, by converting the time interval into a join attribute. The hardness relies on the open question: is there a non-temporal join algorithm running in $O(N^{\text{subw}-\epsilon} + K)$ time, for arbitrarily small constant $\epsilon > 0$, where subw is the *submodular width* [17] of the query? Moreover, for any non-r-hierarchical temporal join, a slight subset of non-hierarchical temporal joins, we prove that any algorithm takes $\Omega(N^{\frac{4}{3}-\epsilon})$ time for arbitrarily small constant $\epsilon > 0$, assuming the 3SUM conjecture[1] holds. (**Section 5**)
- **Experimental evaluation.** We perform an extensive experimental evaluation for practical temporal joins on both synthetic and real-life datasets. We implement our proposed temporal join algorithms, together with the pairwise framework building on the mature binary temporal join algorithm, as the baseline. The experimental results verify the power of our toolkit of temporal join algorithms on different classes of queries. (**Section 6**)

3 TIME-FIRST APPROACH

In this section, we present the *time-first approach* for temporal join evaluation, by extending the *sweep-plane-based algorithm* to general temporal joins. As mentioned, it sorts the endpoints of input tuples first, virtually sweeps a time axis, and computes the join results intersecting with this axis. We first give a general framework in Section 3.1, and then show how to instantiate it for hierarchical temporal joins in Section 3.2, and general temporal join in Section 3.3.

3.1 Framework

We introduce the whole framework in Algorithm 1, and then give an analysis of its time complexity.

Overview of the algorithm. Let $\mathcal{R}$ be a temporal instance of the above join query. Our goal is to compute $Q(\mathcal{R})$. A tuple $\mathbf{a} \in R_e$ for some $e \in \mathcal{E}$, is *active* at time t if $t \in I_{\mathbf{a}}$. For a time t, let $R_e(t) \subseteq R_e$ be the set of active intervals at time t among the tuples in R_e, and let $\mathcal{R}(t) = \{R_e(t) \mid e \in \mathcal{E}\}$. Let $\mathbf{a}$ be a tuple in temporal join $Q(\mathcal{R})$ with valid interval $I_{\mathbf{a}} = [t_{\mathbf{a}}^-, t_{\mathbf{a}}^+]$. Suppose $\mathbf{a} = \bowtie_{e\in\mathcal{E}} \mathbf{a}_e$. Then

[1]The 3SUM conjecture states that given three sets $A, B, C \in \mathbb{R}$, there is no strongly sub-quadratic algorithm to determine whether there exists $(a, b, c) \in A \times B \times C$ such that $a + b = c$.

Algorithm 1: TimeFirst($Q, \mathcal{R}$)

Input : Join query $Q = (\mathcal{V}, \mathcal{E})$ and temporal database $\mathcal{R}$;
Output: Temporal join results $Q(\mathcal{R})$;

1 $S \leftarrow$ Endpoints of valid intervals in $\mathcal{R}$ sorted increasingly;
2 $\mathcal{D} \leftarrow \emptyset, L \leftarrow \emptyset$;
3 **foreach** $p \in S$ **do**
4 Assume $p \in \{t_{\mathbf{a}}^-, t_{\mathbf{a}}^+\}$ for some tuple $\mathbf{a} \in R_e$ with $e \in \mathcal{E}$;
5 **if** $p = t_{\mathbf{a}}^-$ **then**
6 $\mathcal{D} \leftarrow$ Insert($Q, \mathcal{R}, \mathcal{D}, \mathbf{a}$);
7 **else**
8 $L \leftarrow L \cup$ Enumerate($Q, \mathcal{R}, \mathcal{D}, \mathbf{a}$);
9 $\mathcal{D} \leftarrow$Delete($Q, \mathcal{R}, \mathcal{D}, \mathbf{a}$);
10 **return** L;

$I_{\mathbf{a}} = \cap_{e\in\mathcal{E}} I_{\mathbf{a}_e}$ and the right endpoint $t_{\mathbf{a}}^+$ is the same with the right endpoint of a tuple that defines $\mathbf{a}$, say $\mathbf{a}_{e'} \in R_{e'}$, i.e., $t_{\mathbf{a}}^+ = I_{\mathbf{a}_{e'}}^+$. Then, $\mathbf{a}$ is just a tuple in the natural join $Q(\mathcal{R}(t_{\mathbf{a}}^+))$ of $\mathcal{R}(t_{\mathbf{a}}^+)$, so the problem of temporal join reduces to a dynamic instance of natural join, where we maintain the join result over time as tuples are inserted and deleted according to their valid intervals. In view of this observation, here is an outline of the overall algorithm. The algorithm sweeps the time axis from the left to right and maintains the set $\mathcal{R}(t)$ in a data structure $\mathcal{D}$. It stops at the endpoints of the valid intervals, updates $\mathcal{D}$, and reports the tuples of $Q(\mathcal{R})$, as follows. Let S be the sequence of interval endpoints sorted in increasing order. The algorithm visits S from left to right. Suppose it reaches an endpoint t_0. If t_0 is the left endpoint of the valid interval $I_{\mathbf{a}}$ of a tuple $\mathbf{a}$, it inserts $\mathbf{a}$ into $\mathcal{D}$. If t_0 is the right endpoint of $I_{\mathbf{a}}$, then it checks whether $\mathbf{a}$ contributes to a tuple in the natural join $Q(\mathcal{R}(t_{\mathbf{a}}^+))$. If the answer is yes, then it uses the Enumerate procedure (described later) to enumerate all tuples of $Q(\mathcal{R}(t_{\mathbf{a}}^+))$ that involves $\mathbf{a}$. Finally, we delete $\mathbf{a}$ from $\mathcal{D}$.

Run-time Analysis. We next give an abstract analysis of the time complexity of Algorithm 1. Let N be the input size of $\mathcal{R}$. We assume that the data structure $\mathcal{D}$ can be updated in $O(f(N))$ time (line 6 and line 9), and the temporal join results involving tuple $\mathbf{a}$ can be enumerated in $O(g(N) + K(\mathbf{a}))$ time (line 8), where $K(\mathbf{a})$ is the number of temporal join results participated by $\mathbf{a}$. In Algorithm 1, the preprocessing step of sorting (line 1) can be done in $O(N \log N)$ time. In the for loop (lines 3-9), each tuple is inserted into $\mathcal{D}$ and deleted from $\mathcal{D}$ exactly once, thus Insert and Delete procedures together take $O(N \cdot f(N))$ time. Moreover, the procedure Enumerate is invoked for each tuple exactly once, when the right endpoint of its valid interval is visited. Summing over all tuples, this procedure takes $O\left(\sum_{\mathbf{a}\in\mathcal{R}} (g(N) + K(\mathbf{a}))\right) = O\left(N \cdot g(N) + K\right)$ time, where the equation is implied by the fact that each temporal join result is enumerated exactly once. Putting everything together, the time complexity of Algorithm 1 is $O(N \cdot f(N) + N \cdot g(N) + K)$.

A naive application of non-temporal join algorithm at each endpoint of valid interval would not give acceptable performance. The technical challenge is to design a data structure that can be efficiently updated while supporting enumeration of join results at

every interval's right endpoint. For example, simply performing the linear algorithm [86] (see Figure 4) for non-temporal acyclic joins leads to an algorithm of time complexity $O(N^2 + K)$ for hierarchical temporal joins. Using a novel data structure as described in Section 3.2, we improve this result to $O(N \log N + K)$. Moreover, this specially designed algorithm serves as an important building block for general temporal join algorithm in Section 4.

3.2 Hierarchical Temporal Join

We now focus on the class of hierarchical temporal joins and present a near-linear time algorithm based on the general framework.

Data Structure. The *attribute tree* of Q, denoted by $\mathcal{T} := \mathcal{T}(Q)$, is a tree with $\mathcal{V}$ as its nodes such that x is a descendant of y if $\mathcal{E}_x \subseteq \mathcal{E}_y$ (see definition of hierarchical join in Section 2); any path from the root to a leaf corresponds to a hyper-edge (relation) in Q. A path from the root to an internal node may also corresponds to a hyperedge of $\mathcal{E}$ (e.g. *AB* in Figure 5). We first obtain a generalized join tree [49], as follows. Each node $u \in \mathcal{T}$ is associated with the subset $\mathcal{V}_u \subseteq \mathcal{V}$ of attributes appearing on the path from u to the root of $\mathcal{T}$ (Figure 5). Let $p(u)$ be the parent of u, and let $C(u)$ be the set of children of u; $p(u) = \emptyset$ for the root and $C(u) = \emptyset$ for the leaves. Let $\mathcal{T}_u$ be the subtree rooted at u and let $L(u)$ be the set of leaves in $\mathcal{T}_u$. Observe that $\mathcal{V}_{p(u)} \subseteq \mathcal{V}_u$. For an internal node u, if $\mathcal{V}_u$ is a hyperedge of $\mathcal{E}$, i.e., $\mathcal{V}_u \in \mathcal{E}$, we add a leaf node w as a child of u with $\mathcal{V}_w = \mathcal{V}_u$. After this transformation, each relation in $\mathcal{E}$ corresponds to a root-to-leaf path, as shown in Figure 5. Note that $\mathcal{T}$ is independent of $\mathcal{R}$ and does not change during the algorithm.

We are ready now to describe the dynamic data structure $\mathcal{D}$ building on $\mathcal{T}$, which is a simplified version of that used by dynamic query evaluation in [49]. We define the projection $\pi_u(\mathbf{a}) = \pi_{\mathcal{V}_u}(\mathbf{a})$. At any given time t, each node $u \in \mathcal{T}$ stores a set $X_u(t) \subseteq \mathbb{A}_u := \Pi_{x \in \mathcal{V}_u} \mathrm{dom}(x)$ of relations. If u is a leaf, $\mathbb{A}_u$ is a hyperedge of $\mathcal{E}$. For the leaf u, we store $R_u(t)$, the set of active tuples of $R_{\mathcal{V}_u}$. For an internal node u, $X_u(t)$ is the projection on $\mathcal{V}_u$ of (natural) join results of tuples stored at the leaves of $\mathcal{T}_u$, i.e., $X_u(t) = \pi_u(\bowtie_{z \in L(u)} X_z(t))$. By definition, $X_u(t) = \bigcap_{v \in C(u)} \pi_u(X_v(t))$. An example of $\mathcal{D}$ is illustrated in Figure 5. The next lemma shows a nice property of $\mathcal{D}$, which can be proved by induction.

Lemma 3. *For any node $u \in \mathcal{T}$ and any time t, $X_u(t)$ stores the projection of temporal join results induced by relations in the subtree $\mathcal{T}_u$ on attributes $\mathcal{V}_u$, i.e., $X_u(t) = \pi_u(\bowtie_{z \in L(u)} R_z(t))$.*

To update $X_u(t)$ efficiently, tuples in $X_u(t)$ are stored in groups by their values over attributes $\mathcal{V}_{p(u)}$. The set of distinct values over attributes $\mathcal{V}_{p(u)}$ are stored in a binary-search tree as indexes. Moreover, tuples in $X_u(t)$ with the same value over attributes $\mathcal{V}_{p(u)}$ are stored in a min-heap by $t^+_{\mathbf{a}}$.

Initially, $\mathcal{D}$ consists of $\mathcal{T}$ with sets X_u being empty at all nodes $u \in \mathcal{T}$. Since we will only refer to the set $X_u(t)$ at time t, we simply drop the argument t and write X_u to denote the current $X_u(t)$.

Enumerate. As described in Algorithm 2, we divide the enumeration for $\mathbf{a}$ into two steps: (1) check whether $\mathbf{a}$ participates in any temporal results (line 2-5); (2) if no, we just stop (and return an empty set); otherwise, we invoke Report$(\mathcal{D}, root, \mathbf{a})$ to list out all temporal join results participated by $\mathbf{a}$ (line 6).

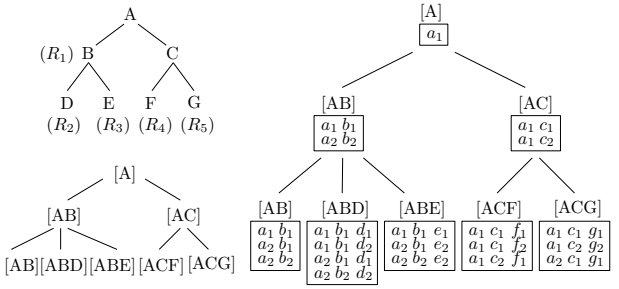

Figure 5: Data structure $\mathcal{D}$ for $Q_{\text{hier}} = R_1(A, B) \bowtie R_2(A, B, D) \bowtie R_3(A, B, E) \bowtie R_4(A, C, F) \bowtie R_5(A, C, G)$. *The left is the attribute tree (upper) and generalized join tree (lower).*

Algorithm 2: Enumerate$(Q, \mathcal{R}, \mathcal{D}, \mathbf{a})$

Input : Join query $Q = (\mathcal{V}, \mathcal{E})$, temporal database $\mathcal{R}$, tuple $\mathbf{a}$, and data structure $\mathcal{D}$ built on Q over $\mathcal{R}$;
Output : Temporal join results $Q(\mathcal{R}(t^+_{\mathbf{a}}))$;

1 $u \leftarrow$ the leaf node corresponding to $e \in \mathcal{E}$ such that $\mathbf{a} \in R_e$;
2 **while** $u \neq null$ **do**
3 **if** $\pi_u(\mathbf{a}) \notin X_u(t^+_{\mathbf{a}})$ **then**
4 **return** $\emptyset$;
5 $u \leftarrow p(u)$;
6 **return** Report$(\mathcal{D}, root, \mathbf{a})$;

Algorithm 3: Report$(\mathcal{D}, u, \mathbf{a})$

Input : Data structure $\mathcal{D}$, node u in $\mathcal{D}$ and tuple $\mathbf{a}$;
Output : Temporal join results of active tuples stored in the subtree $\mathcal{T}_u$ of $\mathcal{D}$, that can be joined with $\mathbf{a}$;

1 **if** *u is a leaf* **then return** $X_u \ltimes \{\mathbf{a}\}$;
2 $\mathcal{S} \leftarrow \emptyset$;
3 **if** $\mathcal{V}_u \subseteq \mathrm{supp}(\mathbf{a})$ **then**
4 **if** $\pi_u(\mathbf{a}) \in X_u$ **then**
5 **foreach** $v \in C(u)$ **do**
6 $\mathcal{S}(v, \mathbf{a}) \leftarrow$Report$(\mathcal{D}, v, \mathbf{a})$;
7 $\mathcal{S} \leftarrow \bigtimes_{v \in C(u)} \mathcal{S}(v, \mathbf{a})$;
8 **else**
9 $\mathcal{L} \leftarrow X_u \ltimes \{\mathbf{a}\}$;
10 **foreach** $\mathbf{b} \in \mathcal{L}$ **do**
11 $\mathcal{S} \leftarrow \mathcal{S} \cup$ Report$(\mathcal{D}, u, \mathbf{b})$;
12 **return** $\mathcal{S}$;

Given a tuple $\mathbf{a} \in R_e$ for some $e \in \mathcal{E}$, line 1-5 checks whether $\mathbf{a}$ participates in any natural join result $Q(\mathcal{R}(t^+_{\mathbf{a}}))$ of $\mathcal{R}(t^+_{\mathbf{a}})$, the currently active sets of tuples. Let u be the leaf of $\mathcal{T}$ corresponding to e, i.e., $e = \mathcal{V}_u$. Algorithm 2 shows that it can be done by checking for every node v lying on the path from root to u, whether $X_v(t^+_{\mathbf{a}})$, i.e., the current set X_v at node v, contains the tuple $\pi_v(\mathbf{a})$. This step takes $O(\log N)$ time, as only $O(1)$ nodes lie on any root-to-leaf path and the check procedure takes $O(\log N)$ time for each node.

We next show a recursive procedure $\textsc{Report}(\mathcal{D}, u, \mathbf{a})$ that outputs $\mathcal{S}(u, \mathbf{a}) = \left(\bowtie_{y \in L(u)} X_y\right) \ltimes \{\mathbf{a}\}$, i.e., the (natural) join results of relations in $\mathcal{T}_u$ that can be joined with $\mathbf{a}$ at timestamp $t^+_{\mathbf{a}}$. Then, our original problem of enumerating all temporal join results participated by $\mathbf{a}$ can be solved by invoking $\textsc{Report}(\mathcal{D}, root, \mathbf{a})$. The following definition of $\mathcal{S}(u, \mathbf{a})$ forms the basis of $\textsc{Report}(\mathcal{D}, u, \mathbf{a})$.

Lemma 4. *Given node $u \in \mathcal{T}$ and tuple $\mathbf{a}$, $\mathcal{S}(u, \mathbf{a})$ is defined as:*

$$\mathcal{S}(u, \mathbf{a}) = \begin{cases} X_u \ltimes \{\mathbf{a}\} & \text{if } u \text{ is a leaf} \\ \bigtimes_{v \in C(u)} \mathcal{S}(v, \mathbf{a}) & \text{if } \mathcal{V}_u \subseteq \text{supp}(\mathbf{a}) \\ \bigcup_{\mathbf{b} \in X_u \ltimes \{\mathbf{a}\}} \mathcal{S}(u, \mathbf{b}) & \text{if } \mathcal{V}_u \not\subseteq \text{supp}(\mathbf{a}) \end{cases} \tag{2}$$

Using (2), $\textsc{Report}(\mathcal{D}, u, \mathbf{a})$ is straightforward, as described in Algorithm 3. Let z be the node of $\mathcal{T}$ such that $\mathbf{a} \in X_z$. In the base case when u is a leaf, **Report** just returns the set of tuples in X_u whose projection on attributes $e \cap \mathcal{V}_u$ is the same with $\mathbf{a}$ (line 1). Intuitively, these tuples will form semi-join results with $\mathbf{a}$. If u is not a leaf, **Report** distinguishes u into two cases.

In the first case (lines 3-7), when $\mathcal{V}_u$ is a subset of $\mathcal{V}_z$, it first checks whether there is a tuple $\mathbf{a}' \in X_u$ with $\mathbf{a}' = \pi_u(\mathbf{a})$. If yes, it invokes this whole procedure recursively for every child node of u with $\mathbf{a}$ (line 5-6), and returns the Cartesian product of enumerated results over its all children as the final join result (line 7).

In the second case (line 8-11), when $\mathcal{V}_u$ is not a subset of $\mathcal{V}_z$, it finds all tuples in X_u whose projection on attributes $\mathcal{V}_z \cap \mathcal{V}_u$ is the same with that of tuple $\mathbf{a}$ (line 9), denoted as $\mathcal{L}$. Then, it invokes this procedure recursively on u for each tuple in $\mathcal{L}$, and returns the union of enumerated results over all tuples in $\mathcal{L}$ (line 10-11).

It can be shown by induction that after spending $O(\log N)$ time, $\mathcal{S}(u, \mathbf{a})$ can be reported in $O(|\mathcal{S}(u, \mathbf{a})|)$ time.

Example 5. In Figure 5, enumeration for tuple $\mathbf{a} = (a_1, b_1) \in R_1$ proceeds by invoking $\textsc{Report}(\mathcal{D}, \text{root}, \mathbf{a})$. The query result $\mathcal{S}(\text{root}, \mathbf{a})$ is essentially $\mathcal{S}(\{AB\}, \mathbf{a}) \times (\mathcal{S}(\{AC\}, \mathbf{b}) \cup \mathcal{S}(\{AC\}, \mathbf{c}))$, for $\mathbf{b} = (a_1, c_1)$ and $\mathbf{c} = (a_1, c_2)$. Moreover, $\mathcal{S}(\{AB\}, \mathbf{a}) = \{(a_1, b_1)\} \times \{(a_1, b_1, d_1), (a_1, b_1, d_2)\} \times \{(a_1, b_1, e_1)\}$, $\mathcal{S}(\{AC\}, \mathbf{b}) = \{(a_1, c_1, f_1), (a_1, c_1, f_2)\} \times \{(a_1, c_1, g_1)\}$, $\mathcal{S}(\{AC\}, \mathbf{c}) = \{(a_1, c_2, f_1)\} \times \{(a_1, c_2, g_2)\}$.

Insert/Delete. Assume that Algorithm 1 visits an endpoint of $I_{\mathbf{a}}$ for tuple $\mathbf{a} \in R_e$ and $e \in \mathcal{E}$. Let z be the leaf of $\mathcal{T}$ corresponding to e, i.e., $e = \mathcal{V}_z$. If we reach the left (resp. right) endpoint of $I_{\mathbf{a}}$, we insert $\mathbf{a}$ into $\mathcal{D}$ (resp. delete $\mathbf{a}$ from $\mathcal{D}$). We only describe the insertion procedure, and the deletion is symmetric.

We first insert $\mathbf{a}$ to $X_z(t)$. Next, we update every node lying on the path from z to the root r, in a bottom-up way. Consider such a node u. If there is an insertion of tuple $\mathbf{a}'$ in X_v for some child $v \in C(u)$, we check whether a tuple $\pi_u(\mathbf{a}')$ needs to be inserted to X_u. In particular, if there exists a tuple $\mathbf{a}'' \in X_{v'}$ with $\pi_v(\mathbf{a}'') = \pi_v(\mathbf{a}')$ for every child $v' \in C(u) - \{v\}$, we insert $\pi_u(\mathbf{a}')$ into $X_u(t)$. This procedure takes $O(\log N)$ time. It updates at most one tuple for every node lying on the path from z to the root. Note that tuples in X_u with the same value over attributes $\mathcal{V}_{p(u)}$ are maintained by a min-heap. The insertion of $\mathbf{a}$ into X_u takes $O(\log N)$ time.

Putting everything together, we come to the following result:

Theorem 6. *For a hierarchical join Q and a temporal instance $\mathcal{R}$, Algorithm 1 computes $Q(\mathcal{R})$ in $O(N \log N + K)$ time.*

Remark. Theorem 6 can be extended to *r-hierarchical join* [47], a slightly larger class of hierarchical join. A join is *r-hierarchical* if its reduced join is hierarchical, where a join is *reduced* if there exists no pair of $e, e' \in \mathcal{E}$ such that $e \subseteq e'$. Any temporal join query can be reduced in linear time.[2]

3.3 General Temporal Join

We now turn to general temporal joins, however, the data structure designed for hierarchical joins cannot be applied. Now, let's take one step back. A straightforward instantiation of TimeFirst framework is to maintain active tuples and apply any non-temporal join algorithm on active tuples, whenever needed. Surprisingly, plugging an output-sensitive non-temporal join algorithm into the TimeFirst framework automatically yield an output-sensitive temporal join algorithm, since the non-temporal join results of active tuples are essentially the temporal join results. In this section, we show how to incorporate an output-sensitive non-temporal join algorithm [43] into the TimeFirst framework.

Data structure. We now use a simple data structure $\mathcal{D}$ storing active tuples of $\mathcal{R}$. More specifically, active tuples from each relation, say R_e, are hashed by attributes in e. The insertion or deletion of a tuple becomes trivial, such that each update takes $O(1)$ time.

Enumerate. Similar to Section 3.2, this procedure enumerates all temporal join results participated by tuple $\mathbf{a}$, i.e., the non-temporal join results over active tuples $\mathcal{R}(t^+_{\mathbf{a}})$ participated by $\mathbf{a}$. We resort to the classical non-temporal join algorithm based on *generalized hypertree decomposition (GHD)* [43] (see Figure 6):

Definition 7 (Generalized Hypertree Decomposition). Given a join query $Q = (\mathcal{V}, \mathcal{E})$, a GHD of Q is a pair $(\mathcal{T}, \lambda)$, where $\mathcal{T}$ is a tree as an ordered set of nodes and $\lambda : \mathcal{T} \to 2^{\mathcal{V}}$ is a labeling function which associates to each vertex $u \in \mathcal{T}$ a subset of attributes in $\mathcal{V}$, λ_u, such that the following conditions are satisfied:

- (coverage) For each $e \in \mathcal{E}$, there is a node $u \in \mathcal{T}$ such that $e \subseteq \lambda_u$;
- (connectivity) For each $x \in \mathcal{V}$, the set of nodes $\{u \in \mathcal{T} : x \in \lambda_u\}$ forms a connected subtree of $\mathcal{T}$.

As described in Algorithm 4, $\textsc{Enumerate}(Q, \mathcal{R}, \mathcal{D}, \mathbf{a})$ first constructs an instance $\mathcal{R}_{\mathcal{T}}$ for a GHD $(\mathcal{T}, \lambda)$ of Q, over active tuples $\mathcal{R}(t^+_{\mathbf{a}})$. This step is quite standard: (i) each node u derives a subjoin over attributes λ_u and relations $\mathcal{E}_u$, the projection of active tuples on attributes λ_u (line 6); (ii) it materializes the result for subjoin $(\lambda_u, \mathcal{E}_u)$ over instance $\mathcal{R}_u$, by invoking the GenericJoin algorithm [66] (line 7). After obtaining an acyclic join query $\mathcal{T}$ over instance $\mathcal{R}_{\mathcal{T}}$, $\textsc{Enumerate}(Q, \mathcal{R}, \mathcal{D}, \mathbf{a})$ essentially invokes the classical Yannakakis algorithm [86] for enumerating all join results participated by $\mathbf{a}$ (line 9).

We note that procedure $\textsc{GenericJoin}(Q, \mathcal{R})$ takes as input an arbitrary join query Q and a non-temporal database $\mathcal{R}$, and outputs the non-temporal join results $Q(\mathcal{R})$. While, procedure Yannakakis

[2] In removing hyperedge $e \in \mathcal{E}$, we update $R_{e'}$ with $R_{e'} \bowtie R_e$, for $e' \in \mathcal{E}$ with $e \subseteq e'$. Recall that there is no pair of tuples in R_e which have the same value on all attributes in e. Together with the fact that $e \subseteq e'$, we can rewrite the temporal join $R_{e'} \bowtie R_e$ as:

$$R_{e'} \bowtie R_e = \{\langle \mathbf{a}, I_{\mathbf{a}} \cap I_{\mathbf{b}} \rangle \mid \mathbf{a} \in R_{e'}, \mathbf{b} \in R_e, \mathbf{b} = \pi_e(\mathbf{a})\}$$

which can be done by computing a non-temporal binary join and then checking validity intervals for joins result. This way, an r-hierarchical temporal join can be reduced to a hierarchical temporal join through $O(1)$ temporal binary joins in linear time.

$(Q, \mathcal{R})$ takes as input an acyclic join query Q and a non-temporal database $\mathcal{R}$, and outputs the non-temporal join results $Q(\mathcal{R})$.

Algorithm 4: EnumerateJoin$(Q, \mathcal{R}, \mathcal{D}, \mathbf{a})$

Input : Join query $Q = (\mathcal{V}, \mathcal{E})$, temporal database $\mathcal{R}$, tuple $\mathbf{a}$, and data structure $\mathcal{D}$ built on Q over $\mathcal{R}$;
Output : Temporal join results $Q(\mathcal{R}(t^+_{\mathbf{a}}))$;

1 Let $(\mathcal{T}, \lambda)$ be a GHD of Q, and $\mathcal{R}_{\mathcal{T}} \leftarrow \emptyset$;
2 **foreach** *node* $u \in \mathcal{T}$ **do**
3 $\mathcal{R}_u \leftarrow \emptyset, \mathcal{E}_u \leftarrow \emptyset$;
4 **foreach** $e \in \mathcal{E}$ *with* $e \cap \lambda_u \neq \emptyset$ **do**
5 $\mathcal{E}_u \leftarrow \mathcal{E}_u \cup \{e \cap \lambda_u\}$;
6 $\mathcal{R}_u \leftarrow \mathcal{R}'_u \cup \{\pi_{e \cap \lambda_u} t \mid t \in R_e(t^+_{\mathbf{a}})\}$;
7 $S_u \leftarrow$ GenericJoin$((\lambda_u, \mathcal{E}_u), \mathcal{R}_u)$;
8 $\mathcal{R}_{\mathcal{T}} \leftarrow \mathcal{R}_{\mathcal{T}} \cup \{S_u\}$;
9 **return** Yannakakis$(\mathcal{T}, \mathcal{R}_{\mathcal{T}})$;

Run-time of Enumerate. Before analysing the time complexity of this procedure, we review the complexity for several building blocks first. A fractional edge cover of a join query $Q = (\mathcal{V}, \mathcal{E})$ is a point $\mathsf{x} = \{\mathsf{x}_e \mid e \in \mathcal{E}\} \in \mathbb{R}^{\mathcal{E}}$ such that for any vertex $v \in \mathcal{V}$, $\sum_{e \in \mathcal{E}_v} \mathsf{x}_e \geq 1$. As proved in [21], the maximum output size of a join query Q is $O(N^{\|\mathsf{x}\|_1})$. The running time of GenericJoin[3] is bounded by $O(N^{\|\mathsf{x}\|_1})$ [66]. Since the above bound holds for any fractional edge cover, we define $\rho = \rho(Q)$ to be the fractional cover with the smallest ℓ_1-norm, i.e., $\rho(Q)$ is the value of the objective function of the optimal solution of linear programming (LP):

$$\min \sum_{e \in \mathcal{E}} \mathsf{x}_e, \text{ s.t. } \forall e \in E : \mathsf{x}_e \geq 0 \text{ and } \forall v \in \mathcal{V} : \sum_{e \in \mathcal{E}_v} \mathsf{x}_e \geq 1. \quad (3)$$

Moreover, Yannakakis can compute the join results of an acyclic join query Q over a non-temporal database $\mathcal{R}$ in $O(N + K)$ time.

Given a join query Q, one of its GHD $(\mathcal{T}, \lambda)$ and a node $u \in \mathcal{T}$, the width of u is defined as the optimal fractional edge covering number of its derived hypergraph $(\lambda_u, \mathcal{E}_u)$, where $\mathcal{E}_u = \{e \cap \lambda_u : e \in \mathcal{E}\}$ (line 5). Given a join query and a GHD $(\mathcal{T}, \lambda)$, the width of $(\mathcal{T}, \lambda)$ is defined as the maximum width over all nodes in $\mathcal{V}_{\mathcal{T}}$. Then, the fractional hypertree width of a join query follows:

Definition 8 (Fractional Hypertree Width [43]). The fractional hypertree width of a join query Q, denoted as $\mathsf{fhtw}(Q)$, is

$$\mathsf{fhtw}(Q) = \min_{(\mathcal{T}, \lambda)} \max_{u \in \mathcal{T}} \rho(\lambda_u, \mathcal{E}_u)$$

i.e., the minimum width over all GHDs.

Basically, $O(N^{\mathsf{fhtw}})$ is an upper bound on the number of join results materialized for each node in $\mathcal{T}$, as well as the time complexity of GenericJoin (line 7). Hence, Algorithm 4 can materialize $O(|\mathcal{R}(t^+_{\mathbf{a}})|^{\mathsf{fhtw}})$ join results for each node in $O(|\mathcal{R}(t^+_{\mathbf{a}})|^{\mathsf{fhtw}})$ time. By resorting to the complexity of Yannakakis algorithm, the last step (line 9) incurs a time cost of $O\left(|\mathcal{R}(t^+_{\mathbf{a}})|^{\mathsf{fhtw}} + Q(\mathcal{R}(t^+_{\mathbf{a}})) \ltimes \{\mathbf{a}\}\right)$, dominating the enumeration step.

[3] Ngo et al. [66] give a more refined bound on the running time but since we assume the size of Q to be a constant, we use $O(N^{\|\mathsf{x}\|_1})$ as a bound on the running time.

Putting everything together, we come to the following result for general temporal joins[4]:

Theorem 9. *For a join query Q and a temporal instance $\mathcal{R}$, Algorithm 1 computes $Q(\mathcal{R})$ in $O(N^{\mathsf{fhtw}+1} + K)$ time.*

As acyclic joins have $\mathsf{fhtw} = 1$, we obtain:

Corollary 10. *For an acyclic join query Q and a temporal instance $\mathcal{R}$, Algorithm 1 computes $Q(\mathcal{R})$ in $O(N^2 + K)$ time.*

4 A HYBRID APPROACH

So far, we are able to tackle a temporal join query using join-first and time-first approaches separately. We highlight the following two from existing extensive results: a near-linear algorithm for hierarchical temporal joins (optimal), and a quadratic-time algorithm for general acyclic temporal joins (the best theoretical result we can achieve in this work). For general cyclic joins, existing results can be further improved by combing these two approaches together, noted as *hybrid approach*.

Our hybrid approach for general temporal joins is still built on the notion of GHD (see Section 3.3), but involving two observations:

- **Hybrid:** We first compute an instance for GHD, by materializing the temporal join results for each node using the join-first approach, and invoke the time-first approach only once to compute the derived acyclic temporal join.
- **Hierarchical GHD:** We identify the *hierarchical GHD* for a general join query, to which the hierarchical temporal join (Section 3.2) can be applied, which provides another choice of applying time-first approach to non-hierarchical temporal join queries.

To characterize the time complexity of such a hybrid approach, we use both the notion of fractional hypertree width (fhtw) of Q from Section 3.3 and the new notion of *hierarchical hypertree width* of Q, denoted by $\mathsf{hhtw}(Q)$, which roughly measures how close Q is to being hierarchical; $\mathsf{hhtw}(Q) = 1$ if Q is hierarchical. The running time of this hybrid approach, as described in Section 4.1 is $O(N^{\min\{\mathsf{fhtw}(Q)+1, \mathsf{hhtw}(Q)\}} + K)$, which is strictly better than both join-first and time-first approach. In Section 4.2 we give some simplification and potential improvement on some specific temporal join queries. At last, we conclude with a general guideline for handling temporal join queries in Section 4.3.

4.1 General Temporal Join Algorithm

As described in Algorithm 5, the overall algorithm follows the standard GHD-based framework. In lines 1-9, we construct a temporal instance of $\mathcal{R}$ with respect to GHD $(\mathcal{T}, \lambda)$ of Q, denoted as $\mathcal{R}_{\mathcal{T}}$. This step is quite similar to Algorithm 4, while the only difference is how to preserve temporal information in the GHD: with respect to the temporal instance $\mathcal{R}_u$ defined for node u, we note that validity intervals of tuples from R_e are carried to $\mathcal{R}_u$ if $e \subseteq \lambda_u$; otherwise, we just set them to be $(-\infty, +\infty)$. By Definition 7, each relation $e \in \mathcal{E}$ has its attributes fully contained by at least one node u, therefore all validity intervals in R_e are preserved in $\mathcal{R}_u$ for some node $u \in \mathcal{T}$, guaranteeing the correctness of the temporal join results.

[4] The exponent of fhtw in Theorem 9 can be further improved to the sub-modular width of input query by rewriting the join query into a union of multiple sub-queries, and apply the (best) GHD-based algorithm for each one [17].

After obtaining the temporal instance $\mathcal{R}_\mathcal{T}$, we invoke the TimeFirst framework (line 10): more specifically, we use the hierarchical temporal join algorithm in Section 3.2 if GHD $(\mathcal{T}, \lambda)$ is hierarchical, and acyclic temporal join algorithm in Section 3.3 otherwise.

Algorithm 5: Hybrid$(\mathcal{Q}, \mathcal{R})$

Input : Join query $\mathcal{Q} = (\mathcal{V}, \mathcal{E})$ and temporal database $\mathcal{R}$;
Output: Temporal join results $\mathcal{Q}(\mathcal{R})$;

1 Let $(\mathcal{T}, \lambda)$ be a GHD of $\mathcal{Q}$; $\mathcal{R}_\mathcal{T} \leftarrow \emptyset$;
2 **foreach** *node* $u \in \mathcal{T}$ **do**
3 $\quad \mathcal{R}_u \leftarrow \emptyset, \mathcal{E}_u \leftarrow \emptyset$;
4 $\quad$ **foreach** $e \in \mathcal{E}$ *with* $e \cap \lambda_u \neq \emptyset$ **do**
5 $\quad\quad \mathcal{E}_u \leftarrow \mathcal{E}_u \cup \{e \cap \lambda_u\}$;
6 $\quad\quad$ **if** $e - \lambda_u = \emptyset$ **then** $\mathcal{R}_u \leftarrow \mathcal{R}_u \cup \{R_e\}$;
7 $\quad\quad$ **else** $\mathcal{R}_u \leftarrow \mathcal{R}_u \cup \{\langle \mathbf{a}, (-\infty, +\infty)\rangle \mid \exists \mathbf{b} \in R_e, \mathbf{a} = \pi_{e \cap \lambda_u}(\mathbf{b})\}$;
8 $\quad S_u \leftarrow$ GenericJoin$((\lambda_u, \mathcal{E}_u), \mathcal{R}_u)$;
9 $\quad \mathcal{R}_\mathcal{T} \leftarrow \mathcal{R}_\mathcal{T} \cup \{\langle \mathbf{a}, I_\mathbf{a}\rangle : \exists \mathbf{a} \in S_u, I_\mathbf{a} \neq \emptyset\}$;
10 **return** TimeFirst$(\mathcal{T}, \mathcal{R}_\mathcal{T})$;

Run-time Analysis. First, we consider the case when $(\mathcal{T}, \lambda)$ is not hierarchical. From Section 3.3, we note that the size of materialized join result for each node in the GHD, as well as the time complexity of GenericJoin invoked for each node, can be bounded by $O(N^{\mathsf{fhtw}})$.[5] Plugging to Corollary 10, the last invocation of TimeFirst on the acyclic join $\mathcal{T}$ takes $O(N^{2\cdot\mathsf{fhtw}} + K)$ time, which also dominates the overall runtime. However, this analysis is not tight. Recall that the time-first approach invokes enumeration procedure at each right endpoint of a valid interval. The number of distinct endpoints of valid intervals in $\mathcal{R}_\mathcal{T}$ is $O(N)$, since applying intersection does not create new endpoints. Hence, the number of enumeration invocations of is $O(N)$, each taking $O(N^{\mathsf{fhtw}} + K(\mathbf{a}))$ time for enumerating results participated by $\mathbf{a}$. Putting everything together, we can improve it to $O(N^{\mathsf{fhtw}+1} + K)$, matching Theorem 9.

Next, we consider the case when $(\mathcal{T}, \lambda)$ is hierarchical. The main observation is that previous analysis could possibly be improved if there exists a hierarchical GHD of $\mathcal{Q}$, on which the hierarchical temporal join algorithm in Section 3.2 can be invoked. To capture it, we define the *hierarchical hypertree width* of a join as follows:

Definition 11 (Hierarchical Hypertree Width). The hierarchical hypertree width of a join query $\mathcal{Q}$, denoted as $\mathsf{hhtw}(\mathcal{Q})$, is

$$\mathsf{hhtw}(\mathcal{Q}) = \min_{(\mathcal{T},\lambda):\mathcal{T} \text{ is hierarchical}} \max_{u \in \mathcal{T}} \rho(\lambda_u, \mathcal{E}_u)$$

i.e., the minimum width over all hierarchical GHDs.

In plain language, hhtw is the minimum width over all hierarchical GHDs of input join. In this way, we obtain another upper bound $O(N^{\mathsf{hhtw}(\mathcal{Q})})$ on the size of materialized join results for each node in the hierarchical GHD, as well as the time cost of GenericJoin invoked for each node. Plugging to Theorem 6, time-first approach takes $O(N^{\mathsf{hhtw}} + K)$ time, which also dominates the overall cost. Combining these two upper bounds, we come to the main result:

[5] When the context is clear, we always use fhtw as short for $\mathsf{fhtw}(Q)$.

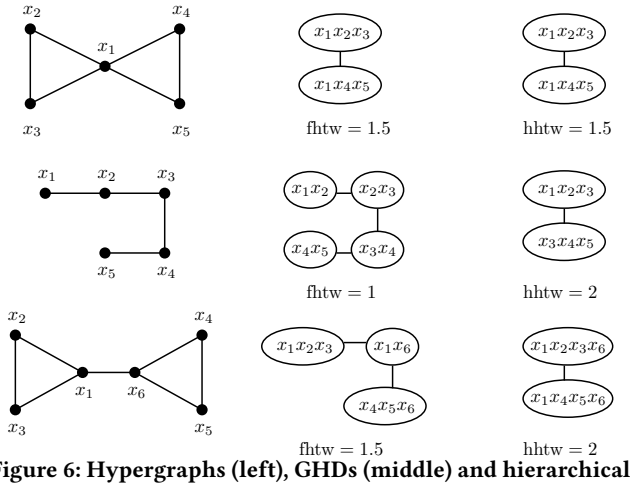

Figure 6: Hypergraphs (left), GHDs (middle) and hierarchical GHDs (right). The first join has fhtw = hhtw = 1.5 since both $(x_1x_2x_3), x_1x_4x_5)$ derive a triangle join with $\rho = 1.5$. The second join is acyclic, thus any join tree is a GHD with fhtw = 1. But the minimum hierarchical GHD has hhtw = 2, with two nodes $\{(x_1x_2x_3), (x_3x_4x_5)\}$. The third join has a GHD with three nodes, where $(x_1x_2x_3), (x_4x_5x_6)$ derive a triangle join with $\rho = 1.5$, so fhtw = 1.5. The minimum hierarchical GHD has hhtw = 2 with two nodes $\{(x_1x_2x_3x_6), (x_1x_4x_5x_6)\}$.

Theorem 12. *Given a join query $\mathcal{Q}$, a temporal instance $\mathcal{R}$ and a parameter $\tau \geq 0$, the τ-durable join result $\mathcal{Q}(\mathcal{R})$ can be computed in $O(N^{\min\{\mathsf{fhtw}+1, \mathsf{hhtw}\}} + K)$ time.*

Remark. The relative ordering between $\mathsf{fhtw}(\mathcal{Q}) + 1$ and $\mathsf{hhtw}(\mathcal{Q})$ is still unclear for general joins. In Figure 6, we give three examples and show their relative orderings. On acyclic joins, we observe:

- If $\mathcal{Q}$ is hierarchical, $\mathsf{hhtw}(\mathcal{Q}) = 1 < \mathsf{fhtw}(\mathcal{Q}) + 1 = 2$; and
- If $\mathcal{Q}$ is acyclic but non-hierarchical, $\mathsf{fhtw}(\mathcal{Q}) + 1 = 2 \leq \mathsf{hhtw}(\mathcal{Q})$,

which implies that (1) time-first approach is the best for hierarchical temporal joins; (2) hybrid approach does not asymptotically improve time-first approach for acyclic but non-hierarchical temporal joins, but may provide another choice in practice.

4.2 Further simplification and improvement

We note that Algorithm 5 can be significantly simplified on some specific GHDs, and further improved by leveraging *interval join*. We need to introduce some terminologies first. In a join query $\mathcal{Q} = (\mathcal{V}, \mathcal{E})$, for a subset of attributes $I \subseteq \mathcal{V}$, let $\mathcal{E}_I = \{e \in \mathcal{E} : e \cap I \neq \emptyset\}$ be the set of hyperedges containing at least one attribute in I, and $\mathcal{Q}_I = (I, \{e \cap I : e \in \mathcal{E}_I\})$ be the subhypergraph induced by I. Then, we lay out the condition for a guarded GHD (see Figure 6):

Definition 13 (Guarded GHD). For a join query $\mathcal{Q}$, a GHD $(\mathcal{T}, \lambda)$ for $\mathcal{Q}$ is *guarded* if all nodes in $\mathcal{T}$ is a one-to-one mapping with $\{e \cup J : e \in \mathcal{E}_I\}$, for $J = \cap_{u \in \mathcal{T}} \lambda_u$ and $I = \mathcal{V} - J$.

How does Algorithm 5 behave on a guarded GHD? Each node $u \in \mathcal{T}$ is labeled with attributes $J \cup e$ for some $e \in \mathcal{E}_I$. Recall that it materializes the temporal join results for every node and then applies TimeFirst to the derived acyclic join. It is very costly to sort all materialized join results and build indexes on top of them.

Algorithm 6: HybridGuarded$(Q, \mathcal{R}, I, J)$

Input : Join query $Q = (\mathcal{V}, \mathcal{E})$ and temporal database $\mathcal{R}$;
Output: Temporal join results $Q(\mathcal{R})$;

1 $\mathcal{S} \leftarrow \emptyset, \widetilde{\mathcal{E}} \leftarrow \{e \in \mathcal{E} \mid e \subseteq J\}$;
2 $\mathcal{L} \leftarrow$ GenericJoin$(Q_J, \{\pi_J R_e \mid e \in \mathcal{E}_J\})$;
3 **foreach** $\mathbf{a} \in \mathcal{L}$ **do**
4 **if** $\widetilde{\mathcal{E}} \neq \emptyset$ **then** $I_{\mathbf{a}} \leftarrow \cap_{e \in \widetilde{\mathcal{E}}} I_{\pi_e(\mathbf{a})}$;
5 **foreach** $e \in \mathcal{E}_I$ **do**
6 $R_e(\mathbf{a}) \leftarrow \{\langle \pi_I(\mathbf{a}'), I_{\mathbf{a}'} \rangle \mid \exists \mathbf{a}' \in R_e, \pi_{e \cap J}(\mathbf{a}) = \pi_{e \cap J}(\mathbf{a}')\}$;
7 $Q_{\mathbf{a}} \leftarrow$ TimeFirst $(Q_I, \{R_e(\mathbf{a}) \mid e \in \mathcal{E}_I\})$;
8 $\mathcal{S} \leftarrow \mathcal{S} \cup (Q_{\mathbf{a}} \times \{\mathbf{a}\})$;
9 **return** $\mathcal{S}$;

Simplification by Rewriting Algorithm 5. We next simplify Algorithm 5 on a guarded GHD. As described in Algorithm 6, HybridGuarded takes a partition (I, J) of attributes $\mathcal{V}$ as input, where $J = \cap_{u \in \mathcal{T}} \lambda_u$ is the set of common attributes appearing in all nodes of $\mathcal{T}$. Our simplified algorithm first computes the temporal join results, denoted by $\mathcal{L}$, on the subquery Q_J induced by J, using GenericJoin. Each tuple $\mathbf{a} \in \mathcal{L}$ derives a residual join Q_I involving only attributes of I, which is then solved by invoking the TimeFirst algorithm. As a comparison, TimeFirst only sorts the input tuples in relations R_e for $e \in \mathcal{E}_I$ and builds indexes on top of them.

Further improvement by Interval Join. We show some further improvement by leveraging the *interval join*[6]. The idea is to replace TimeFirst (line 7) by an interval join, when Q_I is a Cartesian product. We use line-3 join $Q_{L3} = R_1(x_1, x_2) \bowtie R_2(x_2, x_3) \bowtie R_3(x_3, x_4)$ for illustration. Tuples in relation R_1, R_3 are grouped by attribute x_2, x_3 respectively. Distinct values in $\text{dom}(x_2)$ are sorted in a binary-search tree, and the similar applies to $\text{dom}(x_3)$. Moreover, tuples in R_1 (resp. R_3) with the same value on attribute x_2 (resp. x_3) are stored in an interval tree by their validity intervals. These indexes can be built in $O(N \log N)$ time using $O(N \log N)$ space.

As described in Algorithm 6, we identify a partition of $\mathcal{V}$ with $J = \{x_2, x_3\}$. Computing Q_J degenerates to two semi-joins. For tuple $\mathbf{a} \in \mathcal{L}$, let $R_1(\mathbf{a}) = \{\langle \mathbf{a}', I_{\mathbf{a}'} \rangle \mid \mathbf{a}' \in R_1, \pi_{x_2}(\mathbf{a}') = \pi_{x_2}(\mathbf{a})\}$ and $R_3(\mathbf{a}) = \{\langle \mathbf{a}', I_{\mathbf{a}'} \rangle \mid \mathbf{a}' \in R_3, \pi_{x_3}(\mathbf{a}') = \pi_{x_3}(\mathbf{a})\}$. Each $\mathbf{a} \in L$ derives an residual join of $R_1(\mathbf{a}) \times \{\mathbf{a}\} \times R_3(\mathbf{a})$, which can be handled by interval join. It can be shown that this interval-join-based method can improve our existing result from $O(N^2 + K)$ to $O(N^{1.5} + K)$. Investigating how to use interval join to speedup general temporal joins is very interesting, and left as future work.

4.3 Summary: A Guideline for Temporal Joins

Last but not least, we conclude this section by providing a guideline[7] of choosing the best evaluation strategy for temporal joins (see Figure 7). This guideline is built on the worst-case analysis, hence multiple best candidate algorithms could exist for some queries. We don't distinguish those theoretically-equivalent methods, but we can see their differences in empirical evaluation (Section 6).

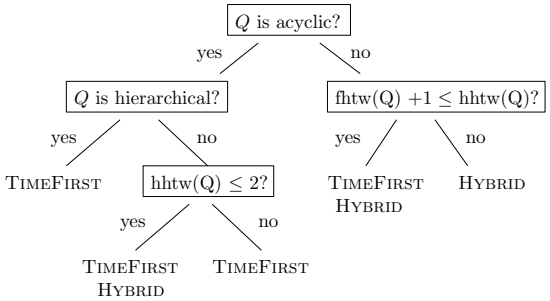

Figure 7: A guideline of choosing temporal Join algorithms.

The guideline only takes as input a temporal join query Q, and works as follows. It starts with determining whether Q is acyclic or not. If Q is acyclic, it further distinguishes whether Q is hierarchical or not. If Q is hierarchical, we directly apply the TimeFirst approach based on the attribute tree (see Section 3.2). Otherwise, Q is acyclic but non-hierarchical. In this case, we always have the TimeFirst approach based on GHDs (see Section 3.3) in hand. If $\text{hhtw}(Q) = 2$, the Hybrid approach based on hierarchical GHD is also competitive. If Q is cyclic, we always have Hybrid in hand. In this case, we note that if $\text{fhtw}(Q) + 1 \leq \text{hhtw}(Q)$, TimeFirst approach based on the GHD is also a candidate solution. When Hybrid approach is invoked, we can always play with the simplification and optimization on guarded GHD if applicable. As the join query has constant size, we can decide which algorithm to pick in $O(1)$ time. Overall, the time complexity of temporal join algorithm chosen by this guideline matches Theorem 12.

5 HARDNESS

In this section, we show hardness of computing temporal joins by relating them to non-temporal joins. The first hardness result is derived for non-r-hierarchical temporal joins based on the 3SUM conjecture [40]. The second hardness result is derived for general temporal joins, by resorting to the open question [17]: whether there exists a faster output-sensitive algorithm for improving the sub-modular width of non-temporal joins.

5.1 Non-R-hierarchical Temporal Joins

Our lower bound as stated in Theorem 14 is built upon the 3SUM conjecture: Given a set S of N numbers, it is conjectured that finding distinct $x, y, z \in S$ such that $x + y = z$ requires $\Omega(N^{2-\epsilon})$ time, for any small constant $\epsilon > 0$ [40].

Theorem 14. *The worst-case running time of any algorithm for the temporal instance $\mathcal{R}$ of the non-r-hierarchical join Q of size N is $\Omega(N^{4/3-\epsilon})$ for any constant $\epsilon > 0$, under the 3SUM conjecture, even if the output size is $O(N)$.*

Proof of Sketch. Hu et al. [47] (Lemma 5.2) proved that any non-r-hierarchical join $Q = (\mathcal{V}, \mathcal{E})$ has a minimal path of length 3, i.e., it is always feasible to find $e_1, e_2, e_3 \in \mathcal{E}$ and $x_1, x_2, x_3, x_4 \in \mathcal{V}$ such that $x_1 \in e_1 - e_2 - e_3$, $x_2 \in e_1 \cap e_2 - e_3$, $x_3 \in e_2 \cap e_3 - e_1$ and $x_4 \in e_3 - e_2 - e_1$. It thus suffices to prove the theorem for line-3 join $Q_{L3} = R_1(x_1, x_2) \bowtie R_2(x_2, x_3) \bowtie R_3(x_3, x_4)$.

[6] Given two sets R, S of intervals, it asks to find all pairs $(r, s) \in R \times S$ such that $r \cap s \neq \emptyset$. W.l.o.g., assume $|R| \leq |S|$. After $O(|S| \log |S|)$ pre-processing time, the query result can be returned in $O(|R| \log |S| + K)$ time.

[7] This guideline can be implemented by taking a temporal join query as input and going through the tests in the decision tree (Figure 7) automatically. The leaf node it reaches is the best algorithm suggested from our theoretical analysis. .

We show a reduction from triangle-listing problem to computing temporal join Q_{L3}. Patrascu [69] has proved that in an undirected graph G, listing N triangles takes $\Omega(N^{4/3-\epsilon})$ time for any constant $\epsilon > 0$, assuming the 3SUM conjecture. Given an undirected graph G, we construct a temporal instance $\mathcal{R}$ for Q_{L3}. For simplicity, assume vertices in G and the domain of attributes in $\mathcal{R}$ are integers. For each edge (u, v) in G, we add following tuples to $\mathcal{R}$:

- $\langle (u + v, u), [v, v] \rangle$ and $\langle (u + v, v), [u, u] \rangle$ to R_1;
- $\langle (u, v), (-\infty, +\infty) \rangle$ and $\langle (v, u), (-\infty, +\infty) \rangle$ to R_2;
- $\langle (u, u + v), [v, v] \rangle$ and $\langle (v, u + v), [u, u] \rangle$ to R_3.

Note that there are no identical tuples in one relation, since edges in G are distinct. There is a one-to-one correspondence between $Q_{L3}(\mathcal{R})$ and the set of triangles in G. The triple $\{u, v, w\}$ forms a triangle in G if and only if $\langle (u+w, u, v, v+w), [w, w] \rangle$,$\langle (v+w, v, u, u+w), [w, w] \rangle$, $\langle (u + v, u, w, w + v), [v, v] \rangle$, $\langle (w + v, w, u, u + v), [v, v] \rangle$, $\langle (v+u, v, w, w+u), [u, u] \rangle$, $\langle (w+u, w, v, v+u), [u, u] \rangle$ are in $Q_{L3}(\mathcal{R})$. If there are N triangles in G, there are $O(N)$ temporal join results in $Q_{L3}(\mathcal{R})$. Any algorithm correctly computing $Q_{L3}(\mathcal{R})$ in $O(N^\gamma)$ time can list all triangles in G in $O(N^\gamma)$ time. Implied by the lower bound for listing triangles, we can show that computing $Q_{L3}(\mathcal{R})$ in $O(N^{4/3-\epsilon})$ time is 3SUM-hard, for any $\epsilon > 0$. □

5.2 Non-temporal Counterpart

Although we have shown a lower bound $\Omega(N^{4/3} + K)$ for line-3 temporal join in Theorem 14, it seems quite difficult to improve our current upper bound of $O(N^{1.5} + K)$ further (Section 4.2). The intuition is that evaluating a temporal line-3 join is equivalent to evaluating a non-temporal triangle join, which is formally captured by Theorem 15 and generalized to arbitrary temporal joins.

Theorem 15. *A temporal join query $Q = (\mathcal{V}, \mathcal{E})$ is as hard as any non-temporal join Q_S for any subset $S \subseteq \mathcal{E}$, where $Q_S = (\mathcal{V} \cup \{x\}, \mathcal{E} - S + \{e \cup \{x\} \mid e \in S\})$.*

The proof of Theorem 15 can be found in the full version [15]. We refer Q_S to be a *non-temporal counterpart* of Q. An example of non-temporal counterpart of line-3 join is $Q_S = R_1(x_1, x_2, x) \bowtie R_2(x_2, x_3) \bowtie R_3(x_3, x_4, x)$ for $S = \{R_1, R_3\}$. So far, a non-temporal join algorithm of $O(N^{\text{subw}(Q)} + K)$ time complexity has been proposed in [17], where $\text{subw}(Q)$ is the sub-modular width of Q. No lower bounds are known to rule out faster algorithms for any specific query, but the known results [17, 62] suggested it very unlikely that an algorithm with $O(N^{\text{subw}(Q)-\epsilon} + K)$ time complexity exists, for any small constant $\epsilon > 0$. In view of Theorem 15, we make the following conjecture:

Conjecture 16. *For a temporal join query $Q = (\mathcal{V}, \mathcal{E})$, there is an instance $\mathcal{R}$ such that it is impossible to compute $Q(\mathcal{R})$ in $O(N^{w-\epsilon}+K)$ time where $w = \max_{S \subseteq \mathcal{E}} subw(Q_S)$, for any small constant $\epsilon > 0$.*

6 EXPERIMENTS

6.1 Setup

All our experiments were implemented in C++, and performed on a Linux machine with two Intel Xeon E5-2640 v4 2.4GHz processor with 256GB of memory. All codes are public at [14].

Algorithms. We have implemented three algorithms for evaluation. (1) **TimeFirst**: We have implemented Algorithm 1 for both hierarchical temporal joins and general temporal acyclic joins. (2) **Hybrid**: We have implemented Algorithm 5 for general temporal joins, and its optimization version **Hybrid-Interval** as described in Algorithm 6. (3) **Baseline**: One baseline algorithm for general temporal join queries sequentially picks a pair of relations to join and materializes their join results as a new relation to be further joined (if applicable, we always pick the best join order). Two relations are joined by resorting to the forward-scan-based algorithm [26], which has been experimentally verified as the most efficient temporal join algorithm. (4) **JoinFirst**: Another baseline algorithm for temporal graph query processing is an instantiation of the join-first approach, which computes all subgraphs matching the query pattern using mature subgraph matching techniques [8] and then checks the validity interval for each subgraph. We note that the same approach has been adopted by previous work [39].

Datasets and Queries. We use both synthetic and real datasets for evaluating different classes of temporal join queries.

Synthetic Dataset. The idea is to enlarge the *intermediate temporal join size* while keeping the final (temporal/durable) join size small, i.e., a large number of intermediate results are dangling without participating in final results. This can be achieved by specifying the distribution of validity intervals of input tuples and adding two additional relations for controlling intermediate results. Details can be found in the full version [15].Overall, we guarantee that no pairwise join ordering can easily compute the join results.

TPC-BiH [50] is the bi-temporal version of the TPC-H benchmark dataset, extended with different types of history classes, such as degenerated, fully bi-temporal or multiple user times. Note that 5 (*partsupp*, *part*, *lineitem*, *orders*, *customer*) out of 8 relations have temporal validity intervals. We select out the following 4 join queries:

- Q_{tpc3} = customer ⋈ order ⋈ lineitem;
- Q_{tpc5} = customer ⋈ order ⋈ lineitem ⋈ supplier;
- Q_{tpc9} = partsupp ⋈ lineitem ⋈ order;
- Q_{tpc10} = partsupp ⋈ lineitem ⋈ order ⋈ customer.

of the 22 standard queries from the benchmark [13] by identifying the underlying temporal join query involving at least 3 relations.

Flights [2] is a graph with 650 vertices and 1,700 edges, storing the flight information with 7 attributes: *id*, *flight-number*, *departure-airport*, *arrival-airport*, *aircraft-id*, *departure-time* and *arrival-time*.

DBLP [58] is a common ego-network from SNAP (Stanford Network Analysis Project) [58] for DBLP. A collaboration graph is constructed where two authors are connected if they publish at least one paper together in any inproceeding. This graph has 2,786,059 authors and 9,460,140 edges. Each edge is associated with a set of disjoint intervals, each one indicating a continuous period in which these two authors keep publishing paper in every years.

TPC-E [12] is an online transaction processing benchmark for stock exchange. We aggregate over the temporal dataset and create a new table *R(CustomerKey, SecurityId, StartTime, EndTime)* for customer and security. An interesting task is to mine customers with similar trading behaviors, e.g., $Q_{tpce} = \sigma_{\text{count} \geq 4} \sum_S R(C_1, S) \bowtie R(C_2, S) \bowtie \cdots \bowtie R(C_5, S)$ finds all sets of 5 customers who held more than 4 common active securities at some timestamp.

LDBC-SNB [3] is a transactional graph processing benchmark, mimicking a social network's activity with the evolving of time.

Table 1: Execution plans for temporal join queries in Section 6. For TimeFirst, we show the GHD for line joins Q_{L3}, Q_{L4}, Q_{L5} and the attribute tree for star joins Q_{S3}, Q_{S4}, Q_{S5}. For Hybrid, we show the GHD for all joins, where each $(.)$ denotes one node. For Hybrid-Interval, we show the partition (I, J) for the set of attributes, used by Algorithm 6.

Join	TimeFirst	Hybrid	Hybrid-Interval	Datasets
Q_{L3}	(x_1x_2)-(x_2x_3)-(x_3x_4)	$(x_1x_2x_3)$ -(x_3x_4)	$I = \{x_1, x_4\}, J = \{x_2, x_3\}$	all
Q_{L4}	(x_1x_2)-(x_2x_3)-(x_3x_4)-(x_4x_5)	$(x_1x_2x_3)$ -$(x_3x_4x_5)$	$I = \{x_1, x_5\}, J = \{x_2, x_3, x_4\}$	
Q_{L5}	(x_1x_2)-(x_2x_3)-(x_3x_4)-(x_4x_5)-(x_5x_6)	$(x_1x_2x_3x_4)$ -$(x_4x_5x_6)$	$I = \{x_1, x_6\}, J = \{x_2, x_3, x_4, x_5\}$	Synthetic
Q_{S3}, Q_{S4}, Q_{S5}	$(x_1) - \{(x_2), (x_3), \cdots, \}$	–	–	Flights, DBLP
Q_{C3}	–	$(x_1x_2x_3)$	–	Synthetic
Q_{C4}	–	$(x_1x_2x_3)$-$(x_1x_4x_3)$	–	Flights
Q_{C5}	–	$(x_1x_2x_3x_4)$ - $(x_1x_4x_5)$	–	
Q_{bowtie}	–	$(x_1x_2x_3)$-$(x_1x_4x_5)$	–	Flights

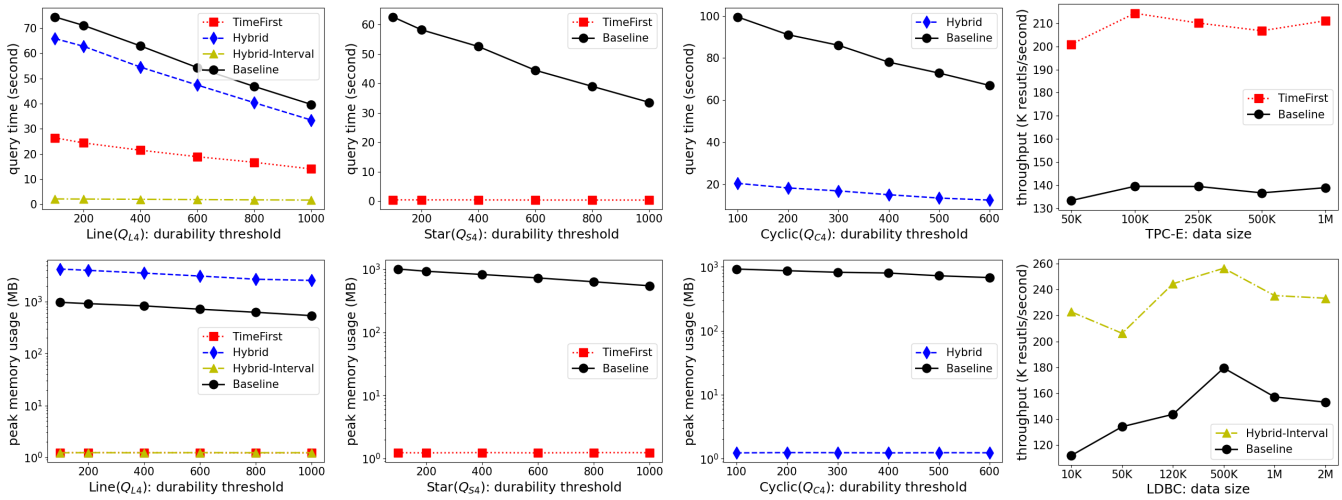

Figure 8: Running time (above) and peak memory usage (below) on synthetic datasets. From left to right: line (Q_{L4}), star (Q_{S4}), and cyclic (Q_{C4})

Figure 9: Scalability.

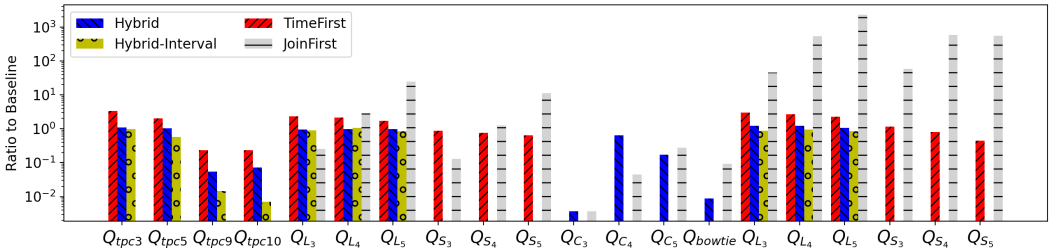

Figure 10: Running time on real datasets (from left to right): TPC-BiH (Q_{tpc3}, Q_{tpc5}, Q_{tpc9}, Q_{tpc10}), Flight ($Q_{L_3}, Q_{L_4}, Q_{L_5}, Q_{S_3}, Q_{S_4}, Q_{S_5}, Q_{C_3}, Q_{C_4}, Q_{C_5}, Q_{bowtie}$) and DBLP ($Q_{L_3}, Q_{L_4}, Q_{L_5}, Q_{S_3}, Q_{S_4}, Q_{S_5}$).

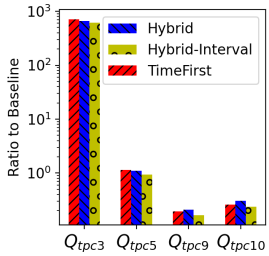

Figure 11: Peak Memory usage on TPC-BiH.

A temporal table *PersonKnowsPerson(PersonId, PersonId, StartTime, CurruentTime)* is used to model relationships among people.

Queries. The set of queries to be evaluated together with their execution plans are summarized in Table 1.

6.2 Experimental Results

Results on Synthetic Dataset. We generate synthetic datasets for line (Q_{L4}), star (Q_{S4}), and cyclic (Q_{C4}) joins, and run our algorithms with Baseline. For each query/dataset combination, we compare the query time and the maximum memory usage for different values of durability threshold τ. The results are shown in Figure 8. We choose $\tau \leq 1000$ since the number of final results already decreases to 0 for $\tau \geq 1000$. Generally, the number of final temporal join results increases as τ decreases. More specifically, on the synthetic dataset over Q_{L4}, there are 109, 98, 69, 10, 8 final results, corresponding to $\tau = 100, 200, 400, 800$ respectively. As verified in Figure 8, the runtime of Baseline increases as τ decreases. In all cases, our algorithms perform better than Baseline which suffers from a large number of intermediate results. For line join, the best algorithms, as expected, are TimeFirst and Hybrid-Interval. Hybrid-Interval runs 70× faster and uses 1000× less space than Baseline. The execution of Hybrid is similar to Baseline on line join queries; there are only small differences in their runtime and memory usage. We also observe this phenomenon on real datasets

later, because Hybrid over line joins just degenerates to the pairwise framework by Baseline, as shown in Table 1. For star join, TimeFirst outperforms Baseline significantly since its running time and memory usage only depend on the input size and final temporal join size (recall Theorem 6), instead of the large number of intermediate results. In some cases, TimeFirst runs 60× faster while using 1000× less memory than Baseline. While Hybrid performs similarly with Baseline over the line join, it outperforms Baseline for the cyclic join query, as the number of intermediate join results generated by Hybrid, i.e., the size of materialized relations for nodes in the GHD, is much smaller than the intermediate join results generated by Baseline. For example, on length-4 cycle join $Q_{C4} = R_1(x_1, x_2) \bowtie R_2(x_2, x_3) \bowtie R_3(x_3, x_4) \bowtie R_4(x_4, x_1)$, Baseline has to materialize a line-3 join (say $R_1 \bowtie R_2 \bowtie R_3$) as intermediate join results, while Hybrid only materializes line-2 joins (say $R_1 \bowtie R_2$) in the process. In most cases, Hybrid runs 5× faster using 1000× less memory than Baseline.

We also experimented with more complicated join queries using the synthetic data generator in Section 6.1. In all cases, our algorithms run much faster than Baseline while using much less space. Interestingly, even with small-size tables, Baseline sometimes could not finish its execution because it ran out of memory with too many intermediate results.

Results on TPC-BiH. In Figure 10, we evaluate different algorithms on four line join queries, and report their runtime as a ratio to Baseline's runtime. We also report the peak memory consumption in Figure 11. On query Q_{tpc3}, only Hybrid-Interval can slightly win over Baseline—the ratio is 0.96. Hybrid roughly equals Baseline, but TimeFirst is nearly 3 times slower than Baseline. The main reason is that relations involved in these two queries (e.g., *customer*, *order* and *lineitem*) generally have low multiplicity between join keys. For example, most customers only place a single order, and most orders only contain one lineitem. Hence, Baseline would not suffer from huge intermediate results. In fact, after the first binary temporal join, we observe that the intermediate table size has almost shrunk to the final answer size. Unfortunately, TimeFirst and Hybrid still have to build and maintain auxiliary data structures for pruning, but these efforts are essentially wasted because there are so few intermediate results. Because of these data characteristics, the overhead of TimeFirst and Hybrid makes them less efficient than Baseline. The peak memory consumption from Figure 11 also confirms this finding—on Q_{tpc3}, Baseline used significantly less memory than other approaches. Results over Q_{tpc5} are similar to that of Q_{tpc3}, but more relations in the query slow Baseline down since longer joins lead to more intermediate results. On Q_{tpc5}, TimeFirst and Hybrid still do not show advantages over Baseline, while Hybrid-Interval can achieve about 50% speedup.

However, when it comes to Q_{tpc9}, we can see our proposed algorithms taking a clear lead — all of them can be at least 10× faster than Baseline, with Hybrid-Interval providing more than 100× speedup. The reason of this dramatic inversion of relative performance still comes down to the data characteristics of joining relations. In Q_{tpc9}, a one-to-many relationship exists between *partsupp* and *lineitem*, hence the intermediate results explode as soon as these relations were joined. Again, peak memory consumption from Figure 11 confirms this behavior. The memory usages of TimeFirst, Hybrid and Hybrid-Interval are only 20% of that of Baseline, demonstrating their pruning power on skipping those unnecessary intermediate results. Similar conclusions can be drawn on Q_{tpc10}.

Results on Flight & DBLP. Both of these datasets are graph-structured. We evaluate a larger class of join queries, including line joins, star joins and general cyclic joins, by conducting self-joins on the edge table. For comprehensiveness, we also implemented JoinFirst for subgraph matching over temporal graphs. Results are summarized in Figure 10 (due to space limit, the results on peak memory consumption of each approach are included in the full version [15]). Same as before, we report running time as a ratio to that of Baseline. On DBLP, for each type of query, JoinFirst performs the worst, up to 3 orders of magnitude slower than Baseline, since it completely ignores temporal predicates until the last. In contrast, at least one approach from our proposed temporal join toolbox wins over Baseline, offering up to 2× speedup. On Flight, a much smaller graph, JoinFirst wins on simpler query patterns (Q_{L3}, Q_{S3}) by an order of magnitude, but can be more than 10× slower on more complex queries (Q_{L5}, Q_{S5}). On the other hand, we can see JoinFirst generally performs well on cyclic queries (Q_{C3}, Q_{C4}) with a 10-100× speedup over Baseline. But similarly, Hybrid can beat JoinFirst on complex patterns (Q_{C5}, Q_{bowtie}) up to an order of magnitude. Overall, JoinFirst outperforms other methods when the number of non-temporal join results is very small, due to the rather simple structure of input query. Moreover, for other types of query on both datasets, at least one approach from our toolbox performs better than Baseline, achieving 2-100× speedup. Though JoinFirst can be an attractive option when dealing with simple patterns on small datasets, our proposed solutions are generally more robust and efficient across datasets and query patterns.

It is worth mentioning that the improvement on graphs is not as significant as that on the synthetic datasets or TPC-BiH. One reason is that self-joins produce significantly different number of intermediate results depending on the input queries. More specifically, no dangling results will be generated for line, star and even-length cycle joins, as it is always possible to extend an intermediate result into a final result, for example $(a-b, b-a, a-b, \cdots)$ is a line or an even-length cycle, $(a-b, a-b, \cdots)$ is a star, etc. These trivial patterns make it difficult to trim intermediate results, so such data characteristic favors Baseline and weakens the pruning power of our techniques. As verified in Figure 10, Baseline performs competitively with the best of our algorithms on Q_{L3}, Q_{L5}, Q_{L6}, Q_{S3}, Q_{S4}, Q_{S5} and Q_{C4}. However, large number of intermediate join results could be generated for odd-length cycles, for example (a, b, a, b, a, b) is not a length-5 cycle. On these queries, Hybrid performs much better than Baseline, confirmed by Figure 10.

Scalability Results on TPC-E & LDBC-SNB. We evaluate the scalability of our algorithms on the TPC-E and LDBC-SNB datasets. For TPC-E, we consider a star join with $\tau = 170$ and vary the input size N from $50K$ to $1M$. For LDBC-SNB, we use a line join with $\tau = 11$ and vary N from $10K$ to $2M$. In order to normalize the performance numbers for better comparison across different datasets, we define a new measurement *throughput* as the average number of join results generated per time unit—the higher the throughput the better. As shown in Figure 9, the throughput for all algorithms roughly stays the same across different input size, despite small

variations, which demonstrates that TimeFirst, Hybrid-Interval and Baseline are output-sensitive, when the output size dominates the input size. On average, TimeFirst outperforms Baseline with 1.5× higher throughput on star join, and Hybrid-Interval beats Baseline on line join with roughly 1.6× higher throughput.

6.3 Summary

From our experiments, we make the following observations. (i) JoinFirst performs well only when the input query has a rather simple structure and the dataset size is small, verifying our theoretical observation that it benefits from small number of non-temporal join results. But JoinFirst behaves the worst when the input query is complex or the dataset is large, due to the large number of non-temporal join results, since it first ignores temporal predicates in join processing. (ii) In almost all cases, at least one of our proposed algorithms (TimeFirst, Hybrid and Hybrid-Interval) is more efficient than Baseline. Baseline (or JoinFirst), and provides a good option even in "easy" scenarios, e.g., when the input size is small, when there are very few dangling intermediate results, or when the query is very simple. (iii) Most importantly, our temporal toolkit is more robust and scalable to query types and input size. Overall, they are fast over all possible scenarios, and can efficiently handle hard instances where Baseline/JoinFirst performs poorly.

Meanwhile, we find that the performance of our proposed algorithms (TimeFirst, Hybrid and Hybrid-Interval) varies depending on the query structures, which verifies our theoretical findings in previous sections. Roughly speaking, TimeFirst behaves the best on hierarchical temporal joins (e.g., star join). For acyclic but non-hierarchical temporal joins (e.g., line join), Hybrid-Interval outperforms both TimeFirst and Hybrid if the GHD is guarded, which illustrates the power of simplification and interval join in Section 4.2. For cyclic temporal joins, Hybrid is always better than the TimeFirst on cycle joins, but JoinFirst could be competitive as well depending on the data statistics. All these observations conform our guideline shown in Figure 7.

An important avenue for future work would be a cost-based optimizer that is aware of both query structure and the underlying data characteristics, and can make intelligent decisions on the best algorithm to use—be it one of the algorithms in our toolbox, or just Baseline, or JoinFirst—for a given occasion.

7 RELATED WORK

Temporal Join and Temporal Support in DBMS. Most of previous efforts are put to binary temporal join, involving only two relations. Temporal binary join reduces to a set of interval joins, so most of previous temporal join algorithms are based on interval joins. Many different techniques have been proposed such as sort/merge-based [45], sweep-plane-based [20, 26, 27, 70], index-based [22, 36, 51, 87], partitioning-based techniques [28, 34, 60, 74, 75, 77] and relational algebra [33]. There are some other works [31, 56] in parallel/distribution settings; and we will focus on in-memory processing in this work. Moreover, how to extend these techniques to a temporal join query involve multiple relations is still unclear.

The adoption of temporal features in industrial database management systems (DBMS) was much slower. SQL included temporal features as part of the SQL:2011 standard [57]. Last decade has witness a big burst of temporal support in conventional database management systems, e.g., MariaDB [4], Oracle [6], IBM DB2 [72], Teradata [19], PostgreSQL [7], Microsoft SQL server [10], Microsoft Trill Temporal Analytical Engine [30]. Other non-relational database management systems also provide temporal features [5, 9, 11, 16].

Query processing over Temporal Graphs/Networks. Extensive research has been performed over temporal graphs and networks for various applications (and we refer interested readers to some nice surveys [29, 46, 54, 63]), depending on different temporal sources (such as nodes, edges, or both), temporal predicates (such as overlap, non-overlap but with bounded gap, chronological ordering), pattern constraints (such as isomorphic subgraphs, motifs), etc. Several representative works include temporal journey/path and its applications [53, 67, 79, 81–83], temporal community detection [42, 59, 84, 85], and temporal motifs search [48, 55, 68, 88]. Closely related to our work, temporal join over graphs/networks degenerates to the subgraph isomorphism problem as a self-join, while edges participating in the subgraph are required to have non-empty intersection among their validity intervals. Temporal subgraph isomorphism has also been widely studied in [61, 71, 78], but in a different setting where edges are put into a temporal sequence and all timestamps fall into a bounded-size window.

To the best of our knowledge, temporal subgraph isomorphism under the non-empty overlap constraint on edges has been only considered in [39] and [73], both of which consider graph patterns as a special case of our temporal joins over hypergraph. [39] designs a general index for searching temporal patterns, while our work provides a toolkit for temporal join that exploits input query structure. [73] finds the top-k durable subgraphs, while our work aims to return all durable join results satisfying the durability condition τ. Moreover, these two works focus on empirical evaluation; our work provide a combination of theoretical and empirical analysis.

Non-temporal Join Algorithms. Numerous variants of the problem have been proposed and hundreds of algorithms have been presented for non-temporal joins. We refer readers to [64] for a survey on join processing. Here we briefly mention some of the work that is directly related to this paper. The tractability of relational join queries is often characterized by the "acyclicity" of the underlying hypergraph of the join queries. The classical Yannakakis algorithm [86] computes an acyclic joins in $O(N + K)$ time. As shown in [69], a triangle join, which is one of the simplest example of cyclic join, takes $\Omega(N^{4/3-\epsilon})$ time for any constant $\epsilon > 0$, even when $K = O(N)$, assuming the 3SUM conjecture. One standard way of handling cyclic joins is to build a decomposition tree of the hypergraph, such that each node defined by a subquery will be computed first and then apply the Yannakakis algorithm on the decomposition tree. Algorithms in this line have their time complexity in terms of $O(N^w + K)$, where w is the width of the decomposition tree such that $O(N^w)$ time is needed for computing every node and materializing their results in this tree; see [43, 62]. Grohe and Marx [44] (see also [21]) established a relationship between the size of a join query and the fractional edge cover ρ of the join. Building on their work, Ngo et al [65] presented a worst-case optimal algorithm for arbitrary join queries of time complexity $O(N^\rho)$, which is simplified in subsequent work [66].

REFERENCES

[1] DBLP. https://snap.stanford.edu/data/com-DBLP.html.
[2] Flights Dataset. https://github.com/IITDBGroup/2019-PVLDB-Reproducibility-Snapshot-Semantics-For-Temporal-Multiset-Relations/tree/master/datasets/flights.
[3] LDBC's Social Network Benchmark. https://ldbcouncil.org/.
[4] MariaDB. https://mariadb.com/kb/en/library/system-versioned-tables/.
[5] MarkLogic. https://www.marklogic.com/.
[6] Oracle. https://www.oracle.com.
[7] PostgreSQL. https://www.postgresql.org.
[8] RapidMatch. https://github.com/RapidsAtHKUST/RapidMatch.
[9] SirixDB. https://sirix.io/.
[10] SQL Server. https://www.microsoft.com/en-us/sql-server/.
[11] TerminusDB. https://terminusdb.com/.
[12] TPC-E Benchmark. http://www.tpc.org/tpce/.
[13] TPC-H Benchmark. http://www.tpc.org/tpch/.
[14] https://github.com/huxiao2010/TemporalJoin.
[15] https://github.com/huxiao2010/TemporalJoin/blob/main/Temporal_Join_SIGMOD_Full.pdf.
[16] XTDB. https://github.com/xtdb/xtdb.
[17] M. A. Khamis, H. Q. Ngo, and D. Suciu. 2017. What do Shannon-type Inequalities, Submodular Width, and Disjunctive Datalog have to do with one another?. In *PODS*. 429–444.
[18] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases*. Vol. 8. Addison-Wesley Reading.
[19] M. Al-Kateb, A. Ghazal, A. Crolotte, R. Bhashyam, J. Chimanchode, and S. Pakala. 2013. Temporal query processing in Teradata. In *EDBT*. 573–578.
[20] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. 1998. Scalable sweeping-based spatial join. In *VLDB*, Vol. 98. 570–581.
[21] A. Atserias, Ma. Grohe, and D. Marx. 2008. Size bounds and query plans for relational joins. In *FOCS*. IEEE, 739–748.
[22] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. 1996. An asymptotically optimal multiversion B-tree. *The VLDB Journal* 5, 4 (1996), 264–275.
[23] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. 1983. On the desirability of acyclic database schemes. *JACM* 30, 3 (1983), 479–513.
[24] C. Berkholz, J. Keppeler, and N. Schweikardt. 2017. Answering conjunctive queries under updates. In *PODS*. 303–318.
[25] M. Böhlen, J. Gamper, and C. S. Jensen. 2006. Multi-dimensional aggregation for temporal data. In *EDBT*. 257–275.
[26] P. Bouros, N. Mamoulis, D. Tsitsigkos, and M. Terrovitis. 2021. In-Memory Interval Joins. *The VLDB journal* (2021), 1–25.
[27] T. Brinkhoff, H. Kriegel, and B. Seeger. 1993. Efficient processing of spatial joins using R-trees. *ACM SIGMOD Record* 22, 2 (1993), 237–246.
[28] F. Cafagna and M. H. Böhlen. 2017. Disjoint interval partitioning. *The VLDB Journal* 26, 3 (2017), 447–466.
[29] A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro. 2012. Time-varying graphs and dynamic networks. *Int. J. Parallel Emergent Distrib. Syst.* 27, 5 (2012), 387–408.
[30] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. 2014. Trill: A high-performance incremental query processor for diverse analytics. *The VLDB journal* 8, 4 (2014), 401–412.
[31] B. Chawda, H. Gupta, S. Negi, T. A. Faruquie, L. V. Subramaniam, and M. K. Mohania. 2014. Processing Interval Joins On Map-Reduce.. In *EDBT*. 463–474.
[32] N. Dalvi and D. Suciu. 2007. Efficient query evaluation on probabilistic databases. *The VLDB Journal* 16, 4 (2007), 523–544.
[33] A. Dignös, M. H. Böhlen, and J. Gamper. 2012. Temporal alignment. In *SIGMOD*. 433–444.
[34] A. Dignös, M. H. Böhlen, and J. Gamper. 2014. Overlap interval partition join. In *SIGMOD*. 1459–1470.
[35] R. Elmasri, G. T. Wuu, and Y. Kim. 1990. The time index: An access structure for temporal data. In *VLDB*. 1–12.
[36] J. Enderle, M. Hampel, and T. Seidl. 2004. Joining interval data in relational databases. In *SIGMOD*. 683–694.
[37] R. Fagin. 1983. Degrees of acyclicity for hypergraphs and relational database schemes. *JACM* 30, 3 (1983), 514–550.
[38] R. Fagin and D. Olteanu. 2016. Dichotomies for Queries with Negation in Probabilistic Databases. *TODS* 41, 1 (2016).
[39] M. Franzke, T. Emrich, A. Züfle, and M. Renz. 2018. Pattern search in temporal social networks. In *EDBT*.
[40] A. Gajentaan and M. H. Overmars. 1995. On a class of O (n2) problems in computational geometry. *Computational geometry* 5, 3 (1995), 165–185.
[41] D. Gao, C. S. Jensen, R. T. Snodgrass, and Michael D. Soo. 2005. Join operations in temporal databases. *The VLDB journal* 14, 1 (2005), 2–29.
[42] M. Gong, L. Zhang, J. Ma, and L. Jiao. 2012. Community detection in dynamic social networks based on multiobjective immune algorithm. *JCST* 27, 3 (2012), 455–467.
[43] G. Gottlob, G. Greco, and F. Scarcello. 2014. Treewidth and hypertree width. *Tractability: Practical Approaches to Hard Problems* 1 (2014).
[44] M. Grohe and D. Marx. 2014. Constraint solving via fractional edge covers. *TALG* 11, 1 (2014), 1–20.
[45] H. Gunadhi and A. Segev. 1991. Query processing algorithms for temporal intersection joins. In *ICDE*. 336–344.
[46] P. Holme and J. Saramäki. 2012. Temporal networks. *Physics reports* 519, 3 (2012), 97–125.
[47] X. Hu and K. Yi. 2019. Instance and Output Optimal Parallel Algorithms for Acyclic Joins. In *PODS*. 450–463.
[48] Yuriy Hulovatyy, Huili Chen, and Tijana Milenković. 2015. Exploring the structure and function of temporal networks with dynamic graphlets. *Bioinformatics* 31, 12 (2015), i171–i180.
[49] M. Idris, M. Ugarte, and S. Vansummeren. 2017. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *SIGMOD*. 1259–1274.
[50] M. Kaufmann, P. M. Fischer, N. May, A. Tonder, and D. Kossmann. 2013. Tpc-bih: A benchmark for bitemporal databases. In *TPCTC*. Springer, 16–31.
[51] M. Kaufmann, A. A. Manjili, P. Vagenas, P. M. Fischer, D. Kossmann, F. Färber, and N. May. 2013. Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In *SIGMOD*. 1173–1184.
[52] N. Kline and R. T. Snodgrass. 1995. Computing temporal aggregates. In *ICDE*. 222–231.
[53] G. Kossinets, J. Kleinberg, and D. Watts. 2008. The structure of information pathways in a social communication network. In *SIGKDD*. 435–443.
[54] V. Kostakos. 2009. Temporal graphs. *Physica A: Statistical Mechanics and its Applications* 388, 6 (2009), 1007–1023.
[55] Lauri Kovanen, Márton Karsai, Kimmo Kaski, János Kertész, and Jari Saramäki. 2011. Temporal motifs in time-dependent networks. *Journal of Statistical Mechanics: Theory and Experiment* 2011, 11 (2011), P11005.
[56] H. Kriegel, P. Kunath, M. Pfeifle, and M. Renz. 2005. Distributed intersection join of complex interval sequences. In *DASFAA*. Springer, 748–760.
[57] K. Kulkarni and J. Michels. 2012. Temporal features in SQL: 2011. *ACM SIGMOD Record* 41, 3 (2012), 34–43.
[58] J. Leskovec and A. Krevl. June 2014. SNAP Datasets: Stanford large network dataset collection. (June 2014).
[59] Y. Lin, Y. Chi, S. Zhu, H. Sundaram, and B. L. Tseng. 2008. Facetnet: a framework for analyzing communities and their evolutions in dynamic networks. In *WWW*. 685–694.
[60] H. Lu, B. C. Ooi, and K. Tan. 1994. On spatially partitioned temporal join. In *VLDB*. 546–557.
[61] P. Mackey, K. Porterfield, E. Fitzhenry, S. Choudhury, and G. Chin. 2018. A chronological edge-driven approach to temporal subgraph isomorphism. In *Big Data*. 3972–3979.
[62] D. Marx. 2013. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *JACM* 60, 6 (2013), 1–51.
[63] O. Michail. 2016. An introduction to temporal graphs: An algorithmic perspective. *Internet Mathematics* 12, 4 (2016), 239–280.
[64] H. Q. Ngo. 2018. Worst-case optimal join algorithms: Techniques, results, and open problems. In *PODS*. 111–124.
[65] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. 2018. Worst-case optimal join algorithms. *JACM* 65, 3 (2018), 1–40.
[66] H. Q. Ngo, C. Ré, and A. Rudra. 2014. Skew strikes back: New developments in the theory of join algorithms. *ACM SIGMOD Record* 42, 4 (2014), 5–16.
[67] R. K. Pan and J. Saramäki. 2011. Path lengths, correlations, and centrality in temporal networks. *Physical Review E* 84, 1 (2011), 016105.
[68] A. Paranjape, A. R. Benson, and J. Leskovec. 2017. Motifs in temporal networks. In *WSDM*. 601–610.
[69] M. Patrascu. 2010. Towards polynomial lower bounds for dynamic problems. In *STOC*. 603–610.
[70] D. Piatov, S. Helmer, and A. Dignös. 2016. An interval join optimized for modern hardware. In *ICDE*. 1098–1109.
[71] U. Redmond and P. Cunningham. 2013. Temporal subgraph isomorphism. In *ASONAM*. IEEE, 1451–1452.
[72] C. M. Saracco, M. Nicola, and L. Gandhi. 2010. *A matter of time: Temporal data management in DB2 for z*. Technical Report. IBM Corporation, New York.
[73] K. Semertzidis and E. Pitoura. 2016. Durable graph pattern queries on historical graphs. In *ICDE*. 541–552.
[74] H. Shen, B. C. Ooi, and H. Lu. 1994. The TP-Index: A dynamic and efficient indexing mechanism for temporal databases. In *ICDE*. 274–281.
[75] I. Sitzmann and P. J. Stuckey. 2000. Improving temporal joins using histograms. In *DEXA*. Springer, 488–498.
[76] R. T. Snodgrass, I. Ahn, G. Ariav, D. Batory, J. Clifford, C. E. Dyreson, R. Elmasri, F. Grandi, C. S. Jensen, W. Käfer, et al. 1994. TSQL2 language specification. *ACM SIGMOD Record* 23, 1 (1994), 65–86.
[77] M. D. Soo, R. T. Snodgrass, and C. S. Jensen. 1994. Efficient evaluation of the valid-time natural join. In *ICDE*. 282–292.

[78] X. Sun, Y. Tan, Q. Wu, B. Chen, and C. Shen. 2019. TM-Miner: TFS-based algorithm for mining temporal motifs in large temporal network. *IEEE Access* 7 (2019), 49778–49789.
[79] J. Tang, M. Musolesi, C. Mascolo, and V. Latora. 2010. Characterising temporal distance and reachability in mobile and online social networks. *SIGCOMM* 40, 1 (2010), 118–124.
[80] T. L. Veldhuizen. 2014. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *ICDT*. 96–106.
[81] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu. 2014. Path problems in temporal graphs. *The VLDB journal* 7, 9 (2014), 721–732.
[82] H. Wu, Y. Huang, J. Cheng, J. Li, and Y. Ke. 2016. Reachability and time-based path queries in temporal graphs. In *ICDE*. 145–156.
[83] B. B. Xuan, A. Ferreira, and A. Jarry. 2003. Computing shortest, fastest, and foremost journeys in dynamic networks. *Int. J. Found. Comput. Sci.* 14, 02 (2003), 267–285.
[84] Y. Yang, D. Yan, H. Wu, J. Cheng, S. Zhou, and J. Lui. 2016. Diversified temporal subgraph pattern mining. In *SIGKDD*. 1965–1974.
[85] Z. Yang, A. W. Fu, and R. Liu. 2016. Diversified top-k subgraph querying in a large graph. In *SIGMOD*. 1167–1182.
[86] M. Yannakakis. 1981. Algorithms for acyclic database schemes. In *VLDB*, Vol. 81. 82–94.
[87] D. Zhang, V. J. Tsotras, and B. Seeger. 2002. Efficient temporal join processing using indices. In *ICDE*. 103–113.
[88] Q. Zhao, Y. Tian, Q. He, N. Oliver, R. Jin, and W. Lee. 2010. Communication motifs: a tool to characterize social communications. In *CIKM*. 1645–1648.