Leveraging Temporal and Topological Selectivities in Temporal-clique Subgraph Query Processing

Kaijie Zhu⋆†, George Fletcher⋆, Nikolay Yakovets⋆
⋆Eindhoven University of Technology, The Netherlands
†NDSC, Zhengzhou, China
{k.zhu, g.h.l.fletcher, hush}@tue.nl

Abstract—We study the problem of temporal-clique subgraph pattern matching. In such patterns, edges are required to jointly overlap in time within a given temporal window in addition to forming a topological sub-structure. This problem arises in many application domains, e.g., in social networks, life sciences, smart cities, telecommunications, and others. State-of-the-art subgraph matching techniques, however, are shown to be limited and inefficient in processing queries with both temporal and topological constraints. We propose an approach that takes full advantage of both topological and temporal selectivities during the processing of temporal-clique subgraph queries. Additionally, we investigate a number of optimizations that can be introduced into our approach to improve its efficiency. Our experimental results demonstrate that our approach outperforms the existing methods by a wide margin at a small additional storage cost.

Index Terms—temporal graph, query processing, database system, join

I. INTRODUCTION

Motivation. Temporal graphs, where time intervals are associated with each edge of the graph, arise in a variety of contemporary applications. Consider a temporal graph representing road traffic in New York City from 2009 to 2021, where vertices represent road intersections and edges represent the flow of vehicles in road segments between intersections. Each edge is labeled with a status of ‘fluid’ or ‘congested’ and carries a time interval representing the duration of the status. For traffic planning, engineers are interested in chains of roads of various length wherein each road in the chain is congested at the same time (i.e., a traffic jam). Furthermore, engineers are interested to find traffic jams at varying time scales, e.g., all traffic jams involving 4 roads occurring in the month of April 2011, and, focusing on a particular period of a particular day, all traffic jams involving 4 roads occurring on 14 April 2011 between 5pm and 7pm, i.e., during rush hour. The key elements of this problem are: the chain pattern (i.e., the topological structure of interest) and a time window in which all edges of the chain pattern jointly overlap in time (i.e., the temporal structure of interest).

In general, we are looking for all embeddings of a topological structure in a temporal graph occurring in a given time window such that all edges of the embedding form a “temporal-clique”. Here “temporal-clique” emphasizes that the edges are tightly interconnected in time, in addition to satisfying the topological pattern of interest. This is in contrast to traditional “cliques” in which nodes are tightly interconnected in topology. This basic problem arises in a wide range of applications beyond the transportation domain.

- In a social network where vertices represent users and edges represent the ‘following’ relationship, find, in the first week of August 2020, all pairs of users who at the same time followed at least three other users in common.
- For malicious network attack detection, where vertices and edges represent IPs and connections, resp., find all Denial-of-Service attack occurring last night between 11pm and 3am, where attackers, bot machines, and victims were connected at the same point in time.
- For deeper understanding of scientific collaborations in a bibliographic database, find all triangles in which 3 people collaborated with each other at the same time, at some point in time in the 1990’s.
- In a sports knowledge graph, find those footballers who, at some moment in 2007, lived in the UK, were managed by Alex Ferguson, and played for Manchester United.

It is important to note that in all of these applications, it is not sufficient to obtain pattern matches which overlap with the query window but do not necessarily jointly occur at a given point in the window, i.e., do not form a temporal-clique. The joint overlap in time is crucial for correct query results (e.g., a traffic jam does not happen if congestion on the roads of the chain occurs on different days in April 2011 for different edges of the chain).

It is also important to note that being able to specify a time window (instead of just a time point or a small fixed window size) for the search is fundamental to the analyses in each of these applications. Indeed, the query time window captures the period of user interest, which can range from seconds to decades (or even longer) depending on the application. Furthermore, while it is possible to convert the search for matches in a time window into a set of queries, one query for each timestamp in the query window, independently solving each of these queries leads to highly inefficient query evaluation. Indeed, there is potentially a tremendous amount of redundant work at each time point, which could be shared and reused across the time points in the window [28].

The problem. Motivated by these observations, we study the temporal-clique subgraph query problem, which generalizes
our examples above: given (1) a temporal graph $G$ where each edge has an associated temporal window; (2) a subgraph query pattern $q$; and, (3) a query time window, find all matches of $q$ in $G$ where the match life-span (i.e., the time interval on which all of the matched edges overlap) is non-empty and overlaps the query time window.

Studies of graph query processing have primarily focused on leveraging the selectivity of topological predicates (e.g., edge labels, value predicates, and join predicates) [6]. However, the selectivity of temporal predicates in real-world temporal networks can have a significant impact on query processing costs, yet there has been relatively little work on leveraging temporal selectivity in temporal graph query processing.

To support efficient temporal-clique subgraph queries, a direct method is to treat the time intervals associated with edges as edge properties and process the temporal predicates using an existing pipeline (i.e., parser, optimizer, and operators). To be more specific, matches that only satisfy the topological join predicates are first produced. Then the selection operators are used to filter the matches that do not satisfy the temporal predicates. We call this class of methods “topology then time” (T&P) since the temporal predicates are processed after topology predicates. Such pipelines can be inefficient since the selectivity of temporal predicates are not fully used. Another class of methods that we name “time then topology” (T&T) processes the temporal predicates using existing techniques (e.g., start time index [28]) and then processes the topological predicates using an existing query evaluation pipeline. This class can also be very inefficient since the topological predicates can be more selective than temporal predicates in some scenarios. We call state of the art methods which process both temporal and topological predicates at the same time to fully take advantage of their selectivities as T&P. However, existing T&P methods concentrate on specific query patterns (e.g., temporal paths) instead of supporting the processing of queries with general and arbitrarily complex patterns.

**Our contributions.** In this paper, we directly address the temporal-clique subgraph query processing problem following a T&P pipeline. In our approach, both temporal and topological characteristics of graphs are indexed. During query processing, temporal and topological predicates are applied at the same time. In this way, the selectivities of both predicates are fully taken advantage of. In particular:

- We propose a novel method, leapfrog TSRJOIN, for efficient temporal-clique subgraph query processing following T&P pipeline, which is efficient in processing queries with general patterns (Section IV).
- We develop several mechanisms to further optimize the processing efficiency in the TSRJOIN (Section V).
- We present the results of an in-depth experimental study which demonstrates significant improvement in performance introduced by our new methods (Section VI).

Our approach is the first T&P method which efficiently supports general and arbitrarily complex query patterns on temporal graphs.

![Fig. 1. Two example temporal graphs.](image-url)
match. We call $l_i(u_i, v_i)$ the $i$th query edge of $q$, denoted $e^q_i$. We call $u_i, v_i$ the query vertices of $q$, denoted $v^q_{2i-1}, v^q_{2i}$. We call each match $e = (e_1, ..., e_n, [e_s, e_e])$ a complete match of $q$. We call $e_i$ an edge match of $e^q_i$, denoted $e_i \sim e^q_i$. We call $\eta(e_i).s = source(\eta(e_i))$ and $\eta(e_i).v = destination(\eta(e_i))$ the vertex bindings of $v^q_{2i-1}$ and $v^q_{2i}$ resp. We call $[e_s, e_e]$ the lifespan of $e$. We call the set of all complete matches the complete result of $q$.

Example. Figure 2 presents three temporal-clique subgraph queries $q_1, q_2,$ and $q_3$. In $q_1$, we have query vertices $v^q_{10}, v^q_{11}$, query edges $e^q_1 = a(v^q_{10}, v^q_{11})$, $e^q_2 = b(v^q_{10}, v^q_{12})$, and $e^q_3 = c(v^q_{11}, v^q_{12})$; and, query window $[10, 20]$. $q_1$ aims to return all complete matches of 3-star pattern overlapping $[10, 20]$ in a given temporal graph.

Note that we aim to find matches for query edges instead of vertices, since multiple edges (e.g., associated with different time intervals) can exist between the same pair of vertices. Two matches $e_1$ and $e_2$ are viewed as distinct matches if they differ on their bindings of at least one query edge.

Partial match. $(e'_1, ..., e'_m, [e'_s, e'_e])$ is a partial match of $q$ if:

1) $e'_1, ..., e'_m \in E$ and $m < n$;
2) For each $i \in [1, m]$ there exists $j \in [1, n]$ such that $e'_i \sim e^q_j$; and,
3) $[e'_s, e'_e] = \tau(e'_1) \cap ... \cap \tau(e'_m)$ and $[e'_s, e'_e] \cap [q_s, q_e] \neq \emptyset$.

Example. Consider $q_1$ evaluated on $G_1$, $(e_4, e_8, e_{12}, [15, 15])$ would be produced as a complete match, where $[15, 15]$ is the life-span of the match. Moreover, $(e_4, [13, 15]), (e_8, [13, 15]), (e_{12}, [15, 16]), (e_4, e_8, [11, 15]), (e_4, e_{12}, [15, 15])$, and $(e_8, e_{12}, [15, 15])$ are all partial matches of the $q_1$.

Problem statement. We study the problem of “temporal-clique subgraph query evaluation”. Given a temporal graph $G$ and a temporal-clique subgraph query $q$, we aim to find the set of all complete matches of $q$ over $G$.

Temporal selective relation (TSR). Given a temporal graph $G = (V, E, \eta, \lambda, \tau)$, a temporal selective relation $R$ in $G$ is a ternary relation $R(l, s, d)$, where

1) $l \in L$ is the label constraint,
2) $s$ is the source constraint, which can be either a vertex $v \in V$ or $*$ (any vertex), and
3) $d$ is the destination constraint, which can be either a vertex $v \in V$ or $*$.

$R$ represents a relation composed by edge $e \in E$ such that $\lambda(e) = l$ and $\eta(e) = (s, d)$. Specifically, $R(l, s, *)$ (or $R(l, *, d)$) denotes all of $s$’s outgoing (or $d$’s in-going) edges associated with label $l$.

Relevant TSR (r-TSR). Suppose for a query edge $e^q = l(u^q, v^q)$ that $v_1$ and $v_2$ are respectively bindings of $u^q$ and $v^q$. We say $R(l, v_1, v_2)$ is relevant to $e^q$ under $v_1, v_2$, and $R(l, v_1, v_2)$ is the relevant TSR of $e^q$ under $v_1, v_2$. If $v_1 = *$ (or $v_2 = *$), we call $R(l, *, v_2)$ (or $R(l, v_1, *)$) the r-TSR of $e^q$ under $v_2$ (or $v_1$).

Bound r-TSR. Given temporal-clique subgraph query $q$ and a binding $v^b$ of query vertex $v^q$ in a partial match of $q$, a r-TSR $R$ is called a $(v^q, v^b)$-bound r-TSR in $q$ if there exists a query edge $e^q$ such that

- $e^q$ is adjacent to $v^q$,
- $e^q$ has not been matched yet, and
- $R$ is relevant to $e^q$ under $v^b$.

Example. Continuing our running example, Figure 3 presents, for $G_1$, three TSRs $R_1(a, v_0, *) = \{e_1, ..., e_5\}$, $R_2(b, v_0, *) = \{e_6, ..., e_{10}\}$, and $R_3(c, v_0, *) = \{e_{11}, e_{12}\}$, which are respectively composed of $v_0$’s outgoing edges associates with label $a, b$, and $c$ in $G_1$. $R_1, R_2$, and $R_3$ are respectively r-TSRs of $e^q_1, e^q_2$, and $e^q_3$ under $v_0$. If the edge matches of $e^q_1, e^q_2$, and $e^q_3$ have not been determined, $R_1, R_2$, and $R_3$ are all $(v^q_0, v^b_0)$-bound r-TSRs in $q_1$.

III. Related work

A. General subgraph query processing

Subgraph queries are processed by executing a guided search over a given graph. During the search, query vertices are bound to graph vertices to produce (partial) matches. A number of different search strategies exist along with a number of pruning strategies which aim to minimize the part of the graph explored during the search.
In breadth-first-search-based approaches, partial matches are produced by processing a query graph edge-at-a-time \([12, 13, 20, 23]\). These methods are based on binary joins (BJs) which extend the partial match by matching query edges to corresponding edges in a graph. In depth-first-search-based approaches, on the other hand, the matches are extended by matching query vertices to vertices in a graph, vertex-at-a-time, by using efficient multi-way joins which are worse-case optimal (WCO). A series of WCO-join algorithms (e.g., NPRR, Leapfrog Triejoin (TrieJOIN), [24], Generic-join [17], Minesweeper [18]) have been proposed as the core of this category. Our approach is based on the TrieJOIN.

The TrieJOIN is a WCO-join algorithm that is currently used in several state-of-the-art database systems (e.g., in LogicBox, in AvantGraph, and others). The basic idea of a TrieJOIN is to iteratively extend the determined bindings for query vertices and filter the candidates by looking ahead similar to the depth-first search algorithm. We identify three key ingredients of a TrieJOIN: (1) the trie representation, (2) the binding production, and (3) the binding propagation. The trie representation indexes the entities (e.g., labels, sources, and targets) of a graph in sorted order so that they can be used as a support for binding production. The binding propagation determines the vertex bindings in a sort-merge algorithm on a pre-constructed trie by using multi-way intersection and leapfrogging. The multi-way intersection technique joins multiple relations by a series of nested intersections, and leapfrogging technique skips over data that is guaranteed not to result in a binding. The binding propagation hands over the determined bindings to the parent operator of a TrieJOIN so that they can be further extended in later processing.

We will now present the details of the binding production in TrieJOIN since it is directly used in our proposed method. Considering that \(k\) sorted unary relations (e.g., each containing vertex IDs) are going to be processed, a method named leapfrog-init() is first invoked to initialize the relations. leapfrog-init() represents each relation by an iterator initially positioned at its first vertex, and then sorts the iterators by their positioned keys in ascending order. Following leapfrog-init(), the main workhorse leapfrog-search() is invoked to find the next binding in the intersection of the \(k\) relations. The basic idea of leapfrog-search() is that, in each turn, considering \(v_{\max}\) is the current highest-value key among the \(k\) iterators, the method takes the iterator positioned the lowest-value key and seeks to \(v_{\max}\) in the corresponding relation. If such key value does not exist, the iterator is positioned to the first key that is no smaller than \(v_{\max}\) and updates the positioned key value as the new \(v_{\max}\). Otherwise, the algorithm returns \(v_{\max}\) as a vertex binding. Subsequent bindings are obtained by invoking a method named leapfrog-next(). leapfrog-next() first positions current iterator at its next key and then invokes leapfrog-search() to find the next binding in the intersection. The procedure is repeated until the vertices in a relation are consumed. In this way, all bindings in the intersection of the \(k\) relations are produced. The overall complexity of TrieJOIN is \(O(Q^* \log M)\), where \(Q^*\) is the upper bound of result size and \(M\) is the largest cardinality among the unary relations.

### B. Temporal subgraph query processing

According to the processing order of topological and temporal predicates, prior methods for temporal subgraph query processing can be classified into \(P^T\), \(T^P\), and \(T^P^T\) as discussed in Section I. For each class, we next introduce the general idea and relevant methods.

**Topology-then-time.** The basic idea in \(P^T\) methods is to index the topological characteristics of graph and process the topological predicates before temporal predicates. Xu et al. [27] proposed the TCGPM-E algorithm, which first produces the topological matches over the subgraphs centering at selective edges and then filters the matches with pruning rules based on temporal predicates. Since query models over property graphs and hybrid planning engines [5], [16] are supported in modern database systems (e.g., PostgreSQL, AvantGraph), a common idea of query processing is to treat the temporal aspects as general selection properties. In this way, physical plans, which are composed of join operators to process topological predicates and selection operators to filter the intermediates that do not satisfy temporal predicates, can be generated and used for temporal-clique subgraph query processing. \(P^T\) methods can be very inefficient since the selectivity of temporal predicates are not fully taken advantage of during query processing.

**Time-then-topology.** \(T^P\) methods essentially index the temporal characteristics of the graph and process temporal predicates before topological predicates. Wu et al. [25] proposed an algorithm for minimum temporal paths. Kumar and Calders [11] proposed an algorithm for circle-enumeration in temporal networks. Edges in these works are sorted temporally to be processed. In recent years, studies on interval join processing methods have provided effective ways to index temporal aspects and process temporal predicates. Query processing pipelines can first processes the temporal predicates using interval join processing methods to find the temporal overlapping cliques. Then physical plans composed of join operators can be applied to the cliques to find valid topological matches for queries. Currently, the best performing solutions for interval joins are based on plane-sweep methods [7], in which relations are sorted in temporal ordering so that they can be processed in a streaming fashion. Piatov et al. [19] proposed two sweep-based interval join algorithms EBI and LEBI, which outperforms prior plan-sweep methods while suffers from a large sweeping range and high maintenance cost on intermediate results. Bouros and Mamoulis [7] proposed two forward scan algorithms gFS and bgFS. However, the two methods suffer when extremely long intervals exist in the processed relations. Based upon an analysis of these shortcomings, Zhu et al. [28] proposed a new sweep-based method based on start time index (STI), which outperforms previous methods. However, even leveraging these solutions, the \(T^P\) methods can still be inefficient since scenarios commonly occur where topological predicates are more selective than temporal predicates.
**Time-and-topology.** Based on the shortcoming of these two classes, T&P methods index both topological and temporal aspects and process the two predicates together to fully take advantage of their selectivities. Mackey et al. [15] proposed an algorithm for temporal subgraph isomorphism which processes directly on edges sorted by time. Franzek et al. [9] extended classical subgraph isomorphism algorithms and proposed a processing algorithm using indexed motifs. However, the definition of temporal networks and queries in these researches are different from ours. Semertzidis and Pitoura [22] proposed an algorithm to find the top-k most durable patterns along the graph. The data and queries in this research is more similar to our own but still important differences exist (e.g., edges do not have labels). Wu et al. [26] proposed the TopChain approach for reachability queries using indexed time-respecting chain coverage in network. Byun et al. [8] proposed the ChronoGraph system for temporal graph traversals, which can support traversal queries such as temporal reachability and shortest path query. Ramesh et al. [21] proposed a distributed execution model for temporal path queries using the interval-centric computing model. These investigations have mostly concentrated on querying specific patterns (e.g., temporal paths) and do not provide a method for querying general subgraph patterns.

To summarize, to the best of our knowledge, state of the art T&P methods do not support processing of arbitrary subgraph patterns on temporal graphs. In this paper, we propose a general T&P method that provides efficient processing for such queries.

**IV. PROPOSED METHOD**

In this work, we propose an operator named Leapfrog TSRJoin (TSRJ, for short) designed for efficient processing of temporal-clique subgraph queries, which is a temporal extension of a TIEJOIN. We choose TIEJOIN as our baseline for its excellent performance in processing of general conjunctive queries on graphs [10] which correspond to resolving the topological predicates in our investigated queries. These merits of TIEJOIN allow us concentrate on a remaining challenge: how can we inject an efficient processing of temporal predicates? A straightforward solution is to insert selection operators after each TIEJOIN as in $P^T$. However, this solution suffers from rigidity in predicate ordering due to its fixed $P^T$ order and vertex-at-a-time matching. In Section VI, we would further illustrate its inefficiency by detailed experiments.

Comparing to TIEJOIN, our proposed TSRJ is composed of four key components: the TSR representation, binding production, partial match production, and partial match propagation. Our method starts from the TSR representation which indexes the TSRs as support for both binding and partial match production. Based on the represented TSRs, the binding production can produce a binding $v^b$ of query vertex $v^q$ as in TIEJOIN. Using $v^b$, the $(v^q,v^b)$-bound r-TSRs $R_1 \ldots R_k$ can be retrieved from the represented TSR. Then, partial match production determines the matches for subgraph composed of $v^q$’s adjacent edges by processing a k-way interval join over $R_1 \ldots R_k$. This, in fact, extends $v_b$ to a series of partial matches, where the endpoints of edge matches are regarded as the bindings of corresponding vertices. Finally, partial match propagation hands over the partial matches to the parent operator so that match’s lifespans and bindings can be later used in processing the remainder of the query. In the remainder of this section, we present the details of all the components except the binding production, which is the same as in a TIEJOIN.

**TSR representation.** Since the aim of TSR representation is to support other components, we should consider the following prerequisites. First, the ordering structure in a trie should be inherited to support the binding production. Second, the TSRs should also be temporally sorted to support the partial match production in which a k-way interval join algorithm is carried out.

Based on these prerequisites, we propose temporal adjacency indexes (TAIs) to represent the TSRs, which are the temporal extensions of a trie in TIEJOIN and adjacency indexes in a database system. The proposed TAIs are composed of four distinct indexes named temporal LS, LD, LSD, LDS indexes. In each index, edges are categorized by the keys in the order as indicated in its name (L: edge labels, S: edge sources, D: edge destinations).

The TAI is constructed as a trie in the order as prescribed by its name thus facilitating efficient topological binding production. Next, corresponding edge-values in a trie are sorted by their start time in ascending order. As a result, the temporally sorted TSRs can be directly obtained from TAIs, which provides a support for partial match production. The construction complexity of TAIs is $O(|E| \cdot (\log |L| + \log |V| + \log |E|))$, where $|L|, |V|, |E|$ are the number of labels, vertices, and edges. To save the cost, LDS can free its attached TSRs while keep the trie structure since $R(l,s,d)$ can still be obtained from the LSD alone.

**Example.** Figure 4 presents the LS and LD of $G_1$. We note the LS and LD-indexing structure (colored in yellow) in the two indexes have inherited ordering structure from trees, which can provide a support to binding production. Besides, we note that attached TSRs (colored in green) are sorted by start time so that they can be directly fed to plane-sweep interval join methods (e.g., FS or STI [28]). More specifically, start-time-sorted $R_1(a,v_1,\ast)$, $R_2(b,v_0,\ast)$, $R_3(c,v_0,\ast)$ can be directly obtained and processed with the plane-sweep interval join methods.

**Partial match production.** We define a second workhorse named leapfrog-temporaloverlap() (denoted LFTO) for efficient partial match production. Every time a binding $v^b$ of query vertex $v^q$ is determined, LFTO is invoked to process the temporal predicates among $(v^b,v^q)$-bound r-TSRs, to find matches for $v^q$’s adjacent edges, and to produce the collection of partial matches. Algorithm 1 presents the procedure of LFTO. Besides bound r-TSRs, a valid time window $[w_s,w_e]$ is used as the input parameter to filter the invalid edges. More specifically, if $v^b$ is determined by the initial leapfrog-search(), the valid window should be exactly the query window $[q_s,q_e]$. 

---

**Algorithm 1:**

1. $R_0(b,v_0,\ast)$ is added to the partial matches.
2. $v_0$ is inserted into the TAI.
3. $R_1(a,v_1,\ast)$ is added to the partial matches.
4. $v_1$ is inserted into the TAI.
5. $R_2(b,v_0,\ast)$ is added to the partial matches.
6. $v_0$ is inserted into the TAI.
7. $R_3(c,v_0,\ast)$ is added to the partial matches.
8. $v_0$ is inserted into the TAI.
9. The partial matches are returned.

---

**Example:** Figure 4 presents the LS and LD of $G_1$. We note the LS and LD-indexing structure (colored in yellow) in the two indexes have inherited ordering structure from trees, which can provide a support to binding production. Besides, we note that attached TSRs (colored in green) are sorted by start time so that they can be directly fed to plane-sweep interval join methods (e.g., FS or STI [28]). More specifically, start-time-sorted $R_1(a,v_1,\ast)$, $R_2(b,v_0,\ast)$, $R_3(c,v_0,\ast)$ can be directly obtained and processed with the plane-sweep interval join methods.

**Partial match production.** We define a second workhorse named leapfrog-temporaloverlap() (denoted LFTO) for efficient partial match production. Every time a binding $v^b$ of query vertex $v^q$ is determined, LFTO is invoked to process the temporal predicates among $(v^b,v^q)$-bound r-TSRs, to find matches for $v^q$’s adjacent edges, and to produce the collection of partial matches. Algorithm 1 presents the procedure of LFTO. Besides bound r-TSRs, a valid time window $[w_s,w_e]$ is used as the input parameter to filter the invalid edges. More specifically, if $v^b$ is determined by the initial leapfrog-search(), the valid window should be exactly the query window $[q_s,q_e]$. 

---

**Algorithm 1:**

1. $R_0(b,v_0,\ast)$ is added to the partial matches.
2. $v_0$ is inserted into the TAI.
3. $R_1(a,v_1,\ast)$ is added to the partial matches.
4. $v_1$ is inserted into the TAI.
5. $R_2(b,v_0,\ast)$ is added to the partial matches.
6. $v_0$ is inserted into the TAI.
7. $R_3(c,v_0,\ast)$ is added to the partial matches.
8. $v_0$ is inserted into the TAI.
9. The partial matches are returned.

---

**Example:** Figure 4 presents the LS and LD of $G_1$. We note the LS and LD-indexing structure (colored in yellow) in the two indexes have inherited ordering structure from trees, which can provide a support to binding production. Besides, we note that attached TSRs (colored in green) are sorted by start time so that they can be directly fed to plane-sweep interval join methods (e.g., FS or STI [28]). More specifically, start-time-sorted $R_1(a,v_1,\ast)$, $R_2(b,v_0,\ast)$, $R_3(c,v_0,\ast)$ can be directly obtained and processed with the plane-sweep interval join methods.

**Partial match production.** We define a second workhorse named leapfrog-temporaloverlap() (denoted LFTO) for efficient partial match production. Every time a binding $v^b$ of query vertex $v^q$ is determined, LFTO is invoked to process the temporal predicates among $(v^b,v^q)$-bound r-TSRs, to find matches for $v^q$’s adjacent edges, and to produce the collection of partial matches. Algorithm 1 presents the procedure of LFTO. Besides bound r-TSRs, a valid time window $[w_s,w_e]$ is used as the input parameter to filter the invalid edges. More specifically, if $v^b$ is determined by the initial leapfrog-search(), the valid window should be exactly the query window $[q_s,q_e]$. 

---

**Algorithm 1:**

1. $R_0(b,v_0,\ast)$ is added to the partial matches.
2. $v_0$ is inserted into the TAI.
3. $R_1(a,v_1,\ast)$ is added to the partial matches.
4. $v_1$ is inserted into the TAI.
5. $R_2(b,v_0,\ast)$ is added to the partial matches.
6. $v_0$ is inserted into the TAI.
7. $R_3(c,v_0,\ast)$ is added to the partial matches.
8. $v_0$ is inserted into the TAI.
9. The partial matches are returned.
If $v^b$ is determined by a propagated partial match, the valid window should be its lifespan. Two groups of $k$-scanners are defined to support the plane-sweep on r-TSRs: the $Scan_{cur}$ and $Scan_{end}$. For each $R$, $Scan_{cur}[i]$ refers to the currently scanned edge in $R$. $Scan_{end}[i]$ refers to the ending of the edge-scanning in $R$. To start with, $Scan_{cur}$ is initialized at the first edge in TSRs (Line 2), which represents collection of starting points of the edge-scanners. Similarly, $Scan_{end}$ is initialized at the first edge which starts later than $w_e$ in TSRs (Line 3), which represents the edge-scanning end point in each relation. A dedicated structure $Active$ is maintained to record the edges at current time that can be used to produce partial matches (Line 4). We use $Active[i]$ to represent $R_i$’s currently active edges in $Active$ sorted by their end-time in ascending order. We define following operations to maintain $Active$:

- $insActive(Active, e, i)$: insert the edge $e$ into $Active[i]$;
- $delActive(Active, t)$: delete all edges $e$ s.t. $t > \tau(e).t_e$ from $Active$;
- $enumActive(Active,e)$: enumerate all partial matches over the elements in $Active$ which contains exactly one occurrence of edge $e$.

In each iteration, the algorithm first obtains the scanner $sc$ (Line 6), the positioned edge of which has the minimal start time, and reads the positioned edge (Line 7). If the edge overlaps the valid window (Line 8), the algorithm deletes the expired edges from $Active$ (Line 9), enumerates the matches in $Active$ containing $e$ (Line 10), and inserts $e$ into $Active$ (Line 11). Finally, the algorithm positions $sc$ to its next edge (Line 12). If $sc$ reaches its scanning end (Line 13), the algorithm closes $sc$ (Line 14). The algorithm keeps iterating until all iterators are closed (Line 5). In this way, LFTO solves the temporal predicates and produces a collection of partial results extended from $v^b$.

Algorithm 1: Leapfrog temporal overlap

Input: bound r-TSRs $R_1, \ldots, R_k$, time window $[w_s, w_e]$

Output: partial match collection $Result$

1. for $i \in [1, k]$ do
   2. $Scan_{cur}[i] \leftarrow R_i.begin()$
   3. $Scan_{end}[i] \leftarrow R_i.upper(w_e)$
   4. $Active \leftarrow \emptyset$, $Result \leftarrow \emptyset$
   5. while $\exists j \in [1, k]$ s.t. $Scan_{cur}[j]$ is not closed do
      6. $sc \leftarrow Scan_{cur}[i]$ s.t. $\min(\tau(Scan_{cur}[i],e).t_e)$
      7. $e \leftarrow sc.e$
      8. if $\tau(e) \cap [w_s, w_e] \neq \emptyset$ then
         9. $delActive(Active, \tau(e).t_e)$
      10. $Result \leftarrow Result \cup enumActive(Active, e)$
      11. $insActive(Active, e, i)$
      12. $sc.next()$
      13. if $Scan_{cur}[i] = Scan_{end}[i]$ then
          14. Close $Scan_{cur}[i]$ and $Scan_{end}[i]$
     15. return $Result$

Example. Continuing our examples and considering $v_0^q$ is bound with $v_0$ and $R_1, R_2, R_3$ shown in Figure 3 are first obtained as the $(v_0^q, v_0)$-bound r-TSRs. $Scan_{cur}[1, 2, 3]$ are initially set at $e_1, e_6, e_{11}$ and $Scan_{end}[1, 2, 3]$ are set at the end of each relation. The processing procedure is shown in Table I. In this way, a star query can be processed in a single TSRJOIN, as shown in Figure 5(a).

The complexity of Algorithm 1 is $O(k \cdot |R_{max}| \log |R_{max}|)$, where $|R_{max}|$ is the cardinality of the largest participating TSR.

Partial match propagation. We consider TSRJOIN as the only join operators used in a physical plan. A plan to process more complex queries (e.g., a chain, a circle) can be composed of more than one TSRJOIN. Given a partial match produced by the LFTO algorithm, partial match propagation allows the match to be handed over to a parent operator so that it can be further extended to the remaining predicates in a query. Comparing to the binding propagation in TRIEJOIN which only hands over the bindings, partial match propagation hands over both bindings and a life-span of the partial result, which fully takes advantage of the selectivity of both topological and
temporal predicates.

Example. Consider a 4-chain query \( q_2 \) and 4-circle query \( q_3 \) (shown in Figure 2) over \( G_2 \) (shown in Figure 1). Figure 5(b) and 5(c) present the physical plans for processing of \( q_2 \) and \( q_3 \). The plan for 4-chain query \( q_2 \), is composed of two TSRJOINS \( B_1 \) and \( B_2 \). \( B_1 \) produces the partial matches of the \( v_3^2 \)-centered 2-star pattern (i.e. \( e_3^2, e_3^3 \)) since \( e_3^2 \) is the most selective query vertex according to our cost model. This determines edge matches for \( e_3^2, e_3^3 \) and bindings for \( v_3^2, v_3^3, v_3^4 \). \( B_2 \) extends each partial match produced by \( B_1 \) with \( e_4^2, e_4^3 \) so that the complete result of \( q_2 \) can be processed. To be more specific, consider in \( B_1 \), \( leapfrog-search() \) has determined \( v_3 \) a binding of \( v_3^2 \). The following LFTO algorithm would process a 2-way interval join over \( R(c, *, v_3) \), \( R(d, v_3, *) \). From the interval join, \( (e_3^2, e_3^3) \) are matched with \( (e_1, e_2) \) and \( (e_1, e_2, [13,18]) \) is produced as a partial match of \( q_2 \). The partial match in fact determines \( v_2, v_4 \) to be the bindings of \( v_2^2, v_2^4 \) respectively. Based on the partial match, \( B_2 \) would process a 2-way interval join over \( R(a, v_3, v_2) \), \( R(b, v_4, *) \) matches \( (e_1^3, e_2^3) \) with \( (e_4, e_5) \), \( (e_6, e_5) \), \( (e_6, e_5) \), and produce complete matches \( (e_4, e_1, e_2, e_3, [15,15]), (e_4, e_1, e_2, e_3, [15,18]), (e_6, e_1, e_2, e_3, [13,15]), (e_6, e_1, e_2, e_3, [13,18]) \). In this way, the complete result of \( q_2 \) over \( G_2 \) is produced.

Similarly, the plan for \( q_3 \) is composed of three TSRJOINS \( C_1, C_2, \) and \( C_3 \). \( C_1 \) produces the matches of the \( v_3^2 \)-centered 2-star pattern as in \( q_2 \). Then \( C_2 \) extends each match propagated from \( C_1 \) with \( e_4^3 \) to find edge matches for \( e_4^2 \) and determine bindings for \( v_4^3 \). Finally, \( C_3 \) extends each match propagated from \( C_2 \) with \( e_4^3 \) to produce complete matches for \( q_3 \). In this way, \( (e_4, e_1, e_2, e_3, [15,15]) \) is finally produced as the complete result of \( q_3 \).

Challenges. The worst-case complexity of a TSRJOIN plan is \( O(Q_p \log M + Q_p \cdot k_{max} \cdot |R_{max}| \cdot \log |R_{max}| + P \cdot k_{max} \cdot \log M) \). \( Q_p \cdot \log M \) is the complexity of original TRIEJOIN as presented in Section III-A, where \( Q_p \) refers to the upper bound on the result size of the query with only topological constraints. \( Q_p \cdot k_{max} \cdot |R_{max}| \cdot \log |R_{max}| \) is the additional complexity introduced by the temporal overlap, where \( k_{max} \) is the largest number of TSRs processed in a single TSRJOIN in a plan. \( P \cdot k_{max} \cdot \log M \) is additional cost of filtering based on the cardinality \( P \) of partial matches produced by TSRJOINS.

Compared to the existing methods following \( P^1 \) and \( T^p \) pipelines, our TSRJOIN approach takes full advantage of the selectivity of both topological and temporal predicates. Also, the logarithmic complexity of the TSR representation and partial match production guarantees that little additional cost is introduced in query processing. In this way, TSRJOIN is expected to be more efficient in temporal-clique subgraph query processing. However, there are several areas in which the efficiency of TSRJOIN can be further improved. We summarize these opportunities as follows:

- Many irrelevant edges can be scanned in partial match production. Continuing our running example in Table I, \( e_1, e_6, e_{11}, e_2, e_7, e_3, e_9, e_{10}, e_3 \) are all irrelevant edges since only \( (e_4, e_8, e_{12}, [15, 15]) \) is produced as a match. This demonstrates that the edges can introduce significant scanning cost in processing selective queries.
- The enumeration of new partial matches can be costly since each scanned edge would lead to the invoking of \( enumActive \), which normally traverses almost the whole \( Active \).

V. OPTIMIZATIONS

Skipping irrelevant edges. We start by categorizing the irrelevant edges by the time they are scanned (i.e., start time). We call the irrelevant edges scanned before the first partial match is produced the backward edges. We call the irrelevant edges scanned after the last partial match is produced the forward edges. Continuing our running example, \( \{e_1, e_6, e_{11}, e_2, e_7, e_3\} \) and \( \{e_9, e_{10}, e_3\} \) are respectively the collection of backward and forward edges.

We introduce the earliest concurrent in TSRs to skip backward edges. The notation of earliest concurrent is first proposed by Zhu et.al. [28] in temporal overlapping clique enumeration. Given a temporal relation \( R \) and timestamp \( t \), the earliest concurrent \( eC(t) \) is the start time of the earliest interval that overlaps \( t \). To support fast look-up, Zhu et al. store the earliest concurrent as a property of each tuple. We consider this to be inefficient since a large number of temporally consecutive tuples can share the same earliest concurrent, which generally leads to large redundancy in storage. In this work, we propose early coverage indexes (ECIs) for more efficient earliest concurrent retrieving in TSRs. ECIs are composed of three indexes LS-EC, LD-EC, LSD-EC, which are respectively used to record the earliest concurrent distribution of TSRs in form of \( R(l, s, t), R(l, s, d), R(l, s, d) \). For each TSR, earliest concurrent distribution is represented by a series of early coverage tuples. Each early coverage tuple is formalized as \( \theta : (c_s, c_e, ec) \), representing that the earliest concurrent of each time \( t \in [c_s, c_e] \) is \( ec \). For convenience of retrieving, tuples for each TSR are sorted by \( c_s \) in ascending order. We provide the following interface to perform a look-up in ECIs:

- \( getCoverageTuple(R, t) \), given a timestamp \( t \) and a TSR \( R \), retrieve the first coverage tuple from corresponding ECI such that \( t \in [c_s, c_e] \). Else, if no such tuple exists,
return the first coverage tuple from corresponding ECI such that \( c_x > t \). Otherwise, return \( \emptyset \).

**Example.** Figure 6 presents the structure of LS-EC and LD-EC, where the early coverage tuples are categorized in the same way as in LS and LD. Using `getCoverageTuple(R(a, v0, *), 1)`, tuple \((0, 5, 0)\) in LS-EC is returned. \((0, 5, 0)\) represents that in TSR \( R(a, v0, *) \), the earliest concurrent of time \( t \in [0, 5] \) is 0. In this way, we obtain that \( eC(1) = 0 \). `getCoverageTuple(R(b, *, v1), 1)`, returns \((1, 3, 1)\) in LD-EC.

The construction complexity of ECIs are \( O(|L| + |V|^2 + |E| + \Theta \cdot \log \Theta) \), where \( \Theta \) is the number of coverage tuples. Comparing to storing the earliest concurrent as tuple property in [28], ECIs stores the earliest concurrent in form of early coverage tuples. This guarantees distinct earliest concurrent value is only stored once in a relation, which significantly improves the efficiency of storage.

Using ECIs, Algorithm 2 presents our final method to skip backward edges by computing the optimized starting points for `Scan_cur`. The basic idea of the algorithm is to find the first collection of coverage tuples \( \theta_1 \ldots \theta_k \) for \( R_1 \ldots R_k \) such that the intersection of \( [\theta_1.cc, \theta_1.cc] \ldots [\theta_k.cc, \theta_k.cc] \) is not empty. Based on the notation of the earliest concurrent, it can be deduced that each coverage tuple \( \theta_i \) reveals that the longest interval in \( R_i \) starting at \( t = \theta_i.ec \) is \([\theta_i.cc, \theta_i.cc] \). In this way, the first non-empty \([\theta_1.cc, \theta_1.cc] \cap \cdots \cap [\theta_k.cc, \theta_k.cc] \) in fact demonstrates that the first match is going to produced at the maximal time among \( \theta_1.cc \ldots \theta_k.cc \). Also, edges in \( R_i \) with start time no smaller than \( \theta_i.cc \) can be relevant to partial matches. In this way, `Scan_cur[i]` can start from the \( \theta_i.cc \), instead of very beginning of \( R_i \).

**Example.** Continuing our running example and using Algorithm 2, `Scan_cur[1, 2, 3]` can respectively start from \( e_4, e_8, e_{12} \) instead of \( e_1, e_6, e_{11} \). In this way, the irrelevant edges \( e_1, e_6, e_{11}, e_2, e_7, e_3 \) can be skipped.

**Algorithm 2: OptimizeStartPoint**

**Input:** \( r-TSRs R_1, \ldots, R_k \), start time of valid window \( w_s \)

**Output:** optimized starting point \( t_i^*, \ldots, t_k^* \)

1. \( t \leftarrow w_s \)
2. while true do
   3. for \( i \in [1, k] \) do
      4. \( \theta_i \leftarrow \text{getCoverageTuple}(R_i, t_0) \)
      5. if \( \theta_i = \emptyset \) then
         6. return \(-1, \ldots, -1\)
      7. if \([\theta_1.cc, \theta_1.cc] \cap \cdots \cap [\theta_k.cc, \theta_k.cc] \neq \emptyset\) then
         8. break
      9. \( t \leftarrow \max \theta_i.cc \)
      10. return \( \theta_1.cc, \ldots, \theta_k.cc \)

To skip the forward edges, we propose `delSkip` operation as a replacement of `delActive`. Besides removing expired edges from `Active`, `delSkip` can additionally identify and skip some forward edges. Algorithm 3 presents the procedure in `delSkip`. If the operation deletes all edges in an `Active[i]` and finds `Scan_cur[i]` is closed, it returns `false` indicating that the following scanned edges are all forward edges and should be skipped.

**Algorithm 3: delSkip**

**Input:** active-list `Active`, timestamp \( t \), scanner list `Scan_cur`

**Output:** `false` if subsequent edges-scanning are non-productive

1. for \( i \in [1, k] \) do
   2. for \( e \in Active[i] \) do
      3. if \( \tau(e).t_s \geq t \) then break
      4. \( Active[i] \leftarrow Active[i]/\{e\} \)
   5. if `Active[i] = \emptyset` and `Scan_cur[i]` is closed then
      6. return `false`
   7. return `true`

**Example.** Continuing our running example, when \( e_9 \) starting at \( t = 17 \) is scanned, `delSkip` is first invoked to remove the expired edges \( e_4, e_8, e_{12} \) from `Active`. Since `Active[3]` becomes empty after `delSkip` and \( R_3 \) has been closed when \( e_{12} \) is scanned, `delSkip` returns `false` which indicates later edge-scans are irrelevant and should be stopped. In this way, the scanning on irrelevant edges \( e_5, e_{10} \) can be skipped.

**Lazy enumeration.** We introduce lazy enumeration to reduce the traversal cost on `Active` in partial match production. Given current timestamp \( t \) and scanner `Scan_cur[i]`, the basic idea of lazy enumeration is not to carry out enumeration until all edges with \( t_s = t \) in \( R_i \) have been scanned by `Scan_cur[i]`. A dedicated structure named `candidate list (C)` is maintained to record the edges starting at current time. For each edge \( e \) scanned by `Scan_cur[i]`, if \( \tau(e).t_s = t \), algorithm adds \( e \) into \( C \) instead of traversing `Active` to enumerate partial results containing \( e \). Otherwise, if \( \tau(e).t_s > t \) or \( sc \) is switched to
another r-TSR, algorithm traverses Active, enumerates partial matches containing each element in C, and finally cleans all elements in C. We define the following operation to replace enumActive and support lazy enumeration in LFTO.

- enumLazy(Active, C): enumerate all partial matches over the elements in Active, which contains an occurrence of edge in C if C ≠ ∅.

Considering n edges in R_t starting at time t, the complexity of enumeration in Algorithm 1 is O(n · \(\prod_{j=1}^{i} |Active(\cdot)| \cdot \prod_{j=i+1}^{k} |Active(\cdot)|\)). Using the lazy enumeration, the complexity is reduced to O(\(\prod_{j=1}^{i} |Active(\cdot)| \cdot \prod_{j=i+1}^{k} |Active(\cdot)|\)). In this way, the traversal cost on Active is significantly reduced.

Optimized LFTO. Using our proposed optimization, Algorithm 4 presents the procedure of optimized LFTO method. Comparing to the original LFTO in Algorithm 1, scanning and enumeration cost in Algorithm 4 are significantly reduced according to our analysis in this section. In this way, the performance of the TSRJOIN is improved.

**Algorithm 4: Optimized Leapfrog temporal overlap**

```
Input: Output: the same as Algorithm 1
1 \(t_s[1, \ldots, k] \leftarrow \text{OptimizeStartPoint}(R_1 \ldots R_k, w_s)\)
2 if \(t_s[1] = -1\) then return \(\emptyset\)
3 for \(i \in [1, k]\) do
4 \(\text{Scan}_i \leftarrow R_i.\text{upper}(t_s[i])\)
5 \(\text{Scan}_{\text{end}}[i] \leftarrow R_i.\text{edges.\text{upper}}(w_k)\)
6 Active ← ∅, Result ← ∅, C ← ∅
7 inRange ← false, \(t' \leftarrow 0, i' \leftarrow -1\)
8 while \(\exists j \in [1, k]\) s.t. Scan\_cur[j] is not closed do
9 \(s_c \leftarrow \text{Scan}_i[s_c] \text{ s.t. } \min_{i \in [1, k]} \tau(\text{Scan}_i[s_c], e).t_s\)
10 \(e \leftarrow s_c.e\)
11 if \(e.t_s < w_s\) then
12 \(\text{insActive}(Active, e, i)\)
13 else
14 if \(t' \neq e.t_s\) or \(i' \neq i\) then
15 if inRange = false then
16 Result ← Result ∪ enumLazy(Active, ∅)
17 inRange ← true
18 else
19 if delSkip(Active, \(t', \text{Scan}_i[\cdot]) = \text{false then break}\)
20 Result ← Result ∪ enumLazy(Active, C)
21 C ← ∅
22 \(\text{insActive}(Active, e, i, C \leftarrow C \cup \{e\})\)
23 \(t' \leftarrow e.t_s, i' \leftarrow i, s_c.next()\)
24 if Scan\_end[i] = Scan\_end[i] then
25 \(\text{Close Scan}_i[\cdot] \text{ and Scan}_{\text{end}}[\cdot]\)
26 return Result
```

**Example.** Continuing our running example, the \(q_1\) processing procedure in Table I can be significantly optimized by using Algorithm 4, as shown in Table II.

<table>
<thead>
<tr>
<th>Edge</th>
<th>Active</th>
<th>Enumerate</th>
</tr>
</thead>
<tbody>
<tr>
<td>e_4</td>
<td>[1]</td>
<td>{e_4}</td>
</tr>
<tr>
<td>e_8</td>
<td>[1]</td>
<td>{e_4, e_8}</td>
</tr>
<tr>
<td>e_12</td>
<td>[1]</td>
<td>{e_4, e_8, e_12}</td>
</tr>
<tr>
<td>e_9</td>
<td>[2]</td>
<td>{e_9}</td>
</tr>
</tbody>
</table>

VI. Experiments

**Environment.** Our experiments were carried out on a server with 192GB RAM and 2 Intel(R) Xeon(R) CPU X5670 with 6 cores at 2.93GHz running a Linux operating system. We implemented the in-memory versions of TSRJOIN in AVANTGRAPH. In all experiments, we used vectorized execution and set the tuple output (i.e., the maximal number of tuples produced in each pull) of each operator to 1024.

**Competitors.** We use three standard state-of-the-art methods, two (namely HYBRID and BINARY) following \(P^1\) pipeline and one (namely TIME) following \(T^p\) pipeline, as competitors to TSRJOIN. These three competitors are all implemented in AVANTGRAPH. In BINARY, a plan composed of binary joins and selection operators is used for each query’s processing. In HYBRID, however, both binary and TRIEJOIN are used since hybrid plans are considered to be more efficient in many situations in static subgraph query processing [16]. Note that HYBRID is a straightforward extension of TRIEJOIN which allows to process temporal-clique subgraph queries. In both BINARY and HYBRID, label adjacency indexes are constructed to fully cover the access patterns during topological subgraph matching. In TIME, the STI-CP [28] algorithm is used, the STI-CP [28] algorithm is first used to find the overlapping cliques and solve the temporal predicates. Then, an execution plan composed of only binary join operators is used to solve the topological predicates in cliques. A corresponding STI-CP index is constructed to support the enumeration of overlapping cliques in the STI-CP algorithm. All implementations are single-core.

**Example.** Figure 8 presents the execution plans in BINARY, HYBRID, and TIME approaches which process \(q_1\) over \(G_1\). In BINARY and HYBRID plan, temporal selections follow each topological join (either a binary join or a TRIEJOIN) to filter the tuples which do not satisfy the temporal predicates. In TIME plan, an interval self-join algorithm is first invoked to solve the temporal predicates and produce the overlapping cliques in a given query window by using an STI-CP index. Then, a topological plan is used to produce matches based on the obtained temporal cliques.

**Query generation.** Our query generation model requires the following parameters to be specified: (1) the number of queries \(N\) in the workload, (2) the proportion of the size of the query

1AVANTGRAPH is a new-generation graph processing engine developed in the Database Group at TU Eindhoven. For more information, please refer to http://avantgraph.io.

2Label adjacency index is a B-tree structure where graph edges are sorted in LSD or LDS order to represent the static adjacency.
window in relation to the entire time domain \( l \in [0, 1] \), (3) the pattern type to be queried, (4) the edge label set \( L \), and (5) the maximal result size \( M \) of each query (this is used to avoid extremely long running queries). Figure 7 presents the patterns used in our experiment evaluation. For each \( k \)-sized query, \( k \) distinct elements are uniformly drawn from \( L \) as the associated labels of query edges respectively. Then we process the query and add it to the workload if its result size is in \([1, M]\). For each workload, 100 queries are generated in our experiment. The \( M \)-parameterized uniform generating method can produce the workloads with appropriate selectivity in short time.

Types of experiments. We run four experiments to investigate the performance of the TSRJOIN: (1) We investigate the performance of algorithms with respect to query patterns shown in Figure 7. (2) We process the queries with maximum output sizes in \([1K, 10K, 100K, 1M, 10M]\) to investigate the performance of algorithms with respect to query selectivity. (3) We process queries with varying query window \( l \) in \([10^{-2}, 10^{-1}, 1, 10, 20]\%\) to investigate the scalability of algorithms in dealing with both long- and short-window queries. (4) We experiment with real-world networks of different sizes. The largest network used in our experiment has 100 million edges.

For each algorithm, we use the average execution time (i.e., its processing cost) and memory consumed by indexes (i.e., its storage cost) to evaluate its efficiency. Each workload is set to timeout in \( 10^5 \) seconds.

Datasets. Table III presents the overview of our six real-world temporal graphs from transportation, social, and networking domains. Yellow [4], Green [4], Bike [3], Divvy [2] record the trips from source to target locations. Stack [14] records the communication among users in StackOverflow. CAIDA [1] records the relationships among autonomous systems.

Results. Figure 9 presents the processing cost of HYBRID, BINARY, TIME, and TSRJOIN with respect to queried patterns in various networks. We fix the length of query window to 10% of the time domain. The maximal result size of each query is set to 100K tuples. We note that the TSRJOIN outperforms all of its competitors in all situations since it fully leverages the selectivity of both topological and temporal predicates. Also, we note that TSRJOIN becomes less efficient in Bike and Divvy, in which edge intervals are much smaller than those in Yellow and Green. This is because of existence...
of more shorter intervals which are more likely to become irrelevant compared to long intervals as we use sweep-plane interval join algorithms. In the three competitors, we note that TIME outperforms the other two $P^J$ methods in the four transportation networks, and loses the advantage in Stack and CAIDA, where the topological predicates are more selective than the temporal predicates. In addition, we note that HYBRID performs the slowest processing in most cases, where all of its instances have timed out. HYBRID allows for efficient processing of the topological part of the query by using TrieJOIN, but effectively disallows the injection of temporal predicates into the TrieJOIN.

To further investigate the findings above, we carry out an additional experiment, in which the total intermediate result cardinality is used to evaluate each instance. A threshold of $10^9$ tuples is set for produced intermediate cardinality in each instance, after which an instance is be forcefully stopped. To save space, Figure 10 presents the intermediate cardinality of the additional experiment in Yellow with query output size fixed to 1000, as an example. We first note that TSRJOIN produces the smallest intermediate cardinality so that it can outperform its competitors in most situations. We also note the intermediate cardinality produced in TIME vs. TSRJOIN while their difference in processing cost is more significant. This is due to the scanning of the irrelevant edges in STI-CP and the significant cost construction of the hash table especially in selective queries. Finally, we note that at most 2% of workload in HYBRID has completed which explains the inefficiency of the HYBRID approach in the previous experiment.

Figure 11 presents the performance of TSRJOIN with respect to query selectivity, where only results of 4-star, 4-chain, 4-circle in transportation networks are presented as examples due to lack of space. We note that the TSRJOIN outperforms all of its competitors in most situations. The only exception is the chain processing in Yellow network, in which our proposal becomes less efficient as selectivity decreases. This demonstrates that a TSRJOIN-only plan generated by our proposed planner is not suitable in processing non-selective chain queries. To illustrate this, consider a variation $q'$ of our 4-chain query $q_2$ in which $e_1^q, e_2^q, e_3^q$ are far more selective than $e_3^q$. Consider a TSRJOIN plan $q'$ in which the bottom operator processes the 2-star sub-query centered at $e_3^q$ for its selectivity. The intermediate cardinality produced by the next operator would be large because partial matches produced by the bottom operator could only be further extended with $e_3^q$ which is regarded as non-selective. This example demonstrates that the inefficiency in chain processing is due to the low-arity of chain pattern, which leads to limited choices for TSRJOIN to bypass non-selective edges. We defer the optimization of such queries to future work.

Figure 12 presents the performance of TSRJOIN with respect to query length ((a)~(c)) and network size ((d)~(e)). The networks of varying size in this experiment are obtained by selecting subsets of predetermined sizes from full networks. We only present the result in Bike since the performance trend in other networks is the same. The result demonstrates that TSRJOIN scales better than its competitors with respect to query length and network size.

Finally, Table IV presents the index storage cost of each

![Table IV](image)
algorithm in various networks. We note that in most networks, TSRJJOIN requires at most twice as much space that its competitors in the worst case. And, in Stack, TSRJJOIN requires higher space consumption. This is expected since TSRJJOIN materializes additional structures to support efficient query processing. Comparing to Binary and Hybrid in which label adjacency index is constructed, TSRJJOIN additionally constructs LS and LD trie structure, a copy of edges, and ECIs for partial result production. Comparing to Time in which STI-CP index is constructed, TSRJJOIN stores additionally two copies of edges and the trie structures. Generally, the storage cost of the three copies is significantly compressed since the trie structure is used to index the edges sharing labels and endpoints. Moreover, ECIs significantly reduce the redundancy in the earliest concurrent storage, which also improves the efficiency of index storage in TSRJJOIN. However, when the number of vertices become much larger, more additional storage cost is introduced due to increase in the size of trie structures. We also present the pre-processing cost on index construction in Table V, from which the similar conclusion can be drawn. Summarizing, we consider the time-space trade-off introduced by the TSRJJOIN to be very reasonable as processing cost is decreased by several orders of magnitude for a small index construction and storage overhead.

VII. CONCLUDING REMARKS

We proposed the TSRJJOIN approach for temporal-clique subgraph query processing in modern graph database engine. TSRJJOIN is designed to take full advantage of the selectivity of both topological and temporal predicates in query processing. Our experimental study demonstrated that TSRJJOIN outperforms current methods by a wide margin with small additional storage cost. In future work, we plan to study query optimization strategies which target a hybrid of binary and TSRJJOIN plans to resolve performance issues of TSRJJOIN-only plans on certain graph patterns (e.g., selective chains).

REFERENCES