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 "to**p**ology then **t**ime" (P^T) 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 "**t**ime then to**p**ology" (T^P) 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.

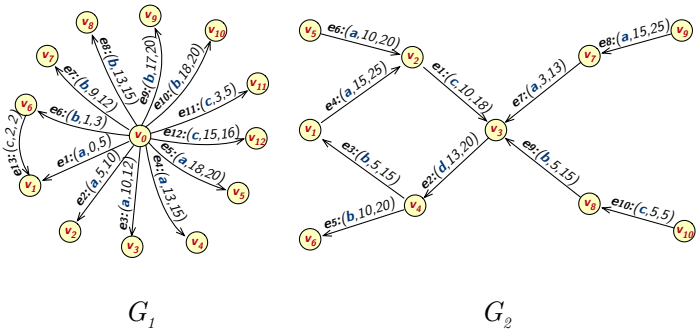

Fig. 1. Two example temporal graphs.

II. PRELIMINARIES

In this section, we introduce basic concepts and definition.

Temporal graph. Let L be a set of labels and T be a set of timestamps. A temporal graph is a structure $G = (V, E, \eta, \lambda, \tau)$, where: V and E are respectively sets of nodes and edges; $\eta : E \to V \times V$ is a function assigning to each edge an ordered pair of nodes, denoted $\eta(e) = (u, v)$, where $e \in E$ and $u, v \in V$; $\lambda : E \to L$ is a function associating each edge with a label, denoted $\lambda(e) = l$ where $e \in E$ and $l \in L$; and, $\tau : E \to T \times T$ is a function assigning to each edge a time interval, denoted $\tau(e) = [t_s, t_e]$ where $e \in E$, $t_s, t_e \in T$, and $t_s \leq t_e$. For convenience, we call l, u, v, t_s, t_e respectively the label, source, destination, start time, and end time of e.

Example. Figure 1 presents two temporal graphs G_1, G_2. In G_1, we have $V = \{v_0, \ldots, v_{12}\}$ and $E = \{e_1, \ldots, e_{13}\}$. Taking e_1 as an example, the label, source, destination, start time, and end time of e_1 are respectively $l = a$, $u = v_0$, $v = v_1$, $t_s = 0$, $t_e = 5$.

Temporal-clique subgraph query. A temporal-clique subgraph query is a pattern q of the form

$$(e_1, \ldots, e_n) \quad \leftarrow \quad l_1(u_1, v_1), \ldots, l_n(u_n, v_n), [q_s, q_e]$$

where $e_1, \ldots, e_n, u_1, v_1, \ldots, u_n, v_n$ are variables (possibly with repetition); $l_1, \ldots, l_n \in L$; and, $q_s, q_e \in T$ where $q_s \leq q_e$. Given a temporal graph $G = (V, E, \eta, \lambda, \tau)$, the evaluation of q on G is the set of all matches $\epsilon = (e_1, \ldots, e_n, [\epsilon_s, \epsilon_e])$ such that:

1) $e_1, ..., e_n \in E$ and $\epsilon_s, \epsilon_e \in T$;
2) there exists a function $f : \{u_1, v_1, ..., u_n, v_n\} \to V$ such that $f(u_i) = source(\eta(e_i))$, $f(v_i) = destination(\eta(e_i))$, and $\lambda(e_i) = l_i$, for $\forall i \in [1, n]$; and,
3) $[\epsilon_s, \epsilon_e] = \tau(e_1) \cap \cdots \cap \tau(e_n)$ and it holds that $[\epsilon_s, \epsilon_e] \cap [q_s, q_e] \neq \emptyset$.

For convenience, we call constraint (2) the topological predicate of q since this ensures the topological structure in a match. Similarly, we call constraint (3) the temporal predicate of q since it ensures the temporal overlapping behavior in a

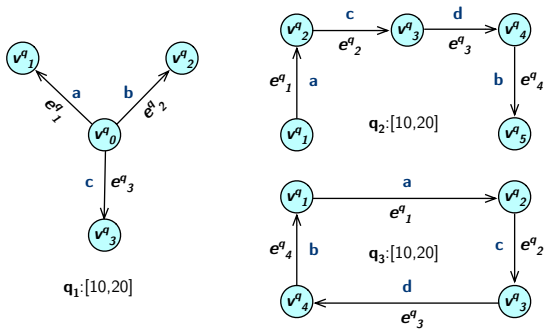

Fig. 2. Three temporal-clique subgraph queries.

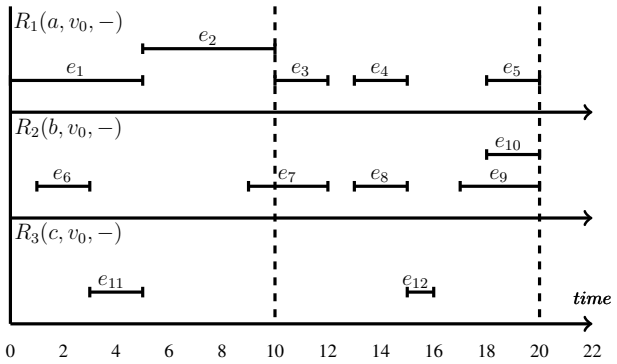

Fig. 3. The collection of r-TSRs of query edges in q_1 under v_0. Dash lines represent the query window.

match. We call $l_i(u_i, v_i)$ the ith query edge of q, denoted e_i^q. We call u_i, v_i the query vertices of q, denoted v_{2i-1}^q, v_{2i}^q. We call each match $\epsilon : (e_1, \ldots, e_n, [\epsilon_s, \epsilon_e])$ a complete match of q. We call e_i an edge match of e_i^q, denoted $e_i \sim e_i^q$. We call $\eta(e_i).u = source(\eta(e_i))$ and $\eta(e_i).v = destination(\eta(e_i))$ the vertex bindings of v_{2i-1}^q and v_{2i}^q, resp. We call $[\epsilon_s, \epsilon_e]$ the lifespan of ϵ. 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 $v_0^q, \ldots, v_3^q$; query edges $e_1^q = a(v_0^q, v_1^q)$, $e_2^q = b(v_0^q, v_2^q)$, and $e_3^q = c(v_0^q, v_3^q)$; 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 ϵ_1 and ϵ_2 are viewed as distinct matches if they differ on their bindings of at least one query edge.

Partial match. $(e'_1, \ldots e'_m, [\epsilon'_s, \epsilon'_e])$ 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 $e'_i \sim e_j^q$; and,
3) $[\epsilon'_s, \epsilon'_e] = \tau(e'_1) \cap \cdots \cap \tau(e'_m)$ and $[\epsilon'_s, \epsilon'_e] \cap [q_s, q_e] \neq \emptyset$.

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, [13,15]), (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 $R_1(a, v_0, *) = \{e_1, \ldots, e_5\}$, $R_2(b, v_0, *) = \{e_6, \ldots, e_{10}\}$, 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_1^q, e_2^q, and e_3^q under v_0. If the edge matches of e_1^q, e_2^q, and e_3^q have not been determined, R_1, R_2, and R_3 are all (v_0^q, v_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 worst-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 production 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 $\mathsf{P^T}$, $\mathsf{T^P}$, and T&P as discussed in Section I. For each class, we next introduce the general idea and relevant methods.

Topology-then-time. The basic idea in $\mathsf{P^T}$ 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. $\mathsf{P^T}$ methods can be very inefficient since the selectivity of temporal predicates are not fully taken advantage of during query processing.

Time-then-topology. $\mathsf{T^P}$ methods essentially index the temporal characteristics of the graph and process temporal predicates before topological predicates. Wu et al. [25] proposed a algorithm for mininum 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 $\mathsf{T^P}$ 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. Franzeke 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 Chrono-Graph 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 TSR-Join (TSRJoin, for short) designed for efficient processing of temporal-clique subgraph queries, which is a temporal extension of a TrieJoin. We choose TrieJoin 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 TrieJoin 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 TrieJoin 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 TrieJoin, our proposed TSRJoin 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 TrieJoin. Using v^b, the (v^q, v^b)-bound r-TSRs $R_1 \dots R_k$ 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 $R_1 \dots R_k$. 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 TrieJoin.

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 TrieJoin 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 tries, 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 $R_1(a, v_0, *)$, $R_2(b, v_0, *)$, $R_3(c, v_0, *)$ 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 $[q_s, q_e]$.

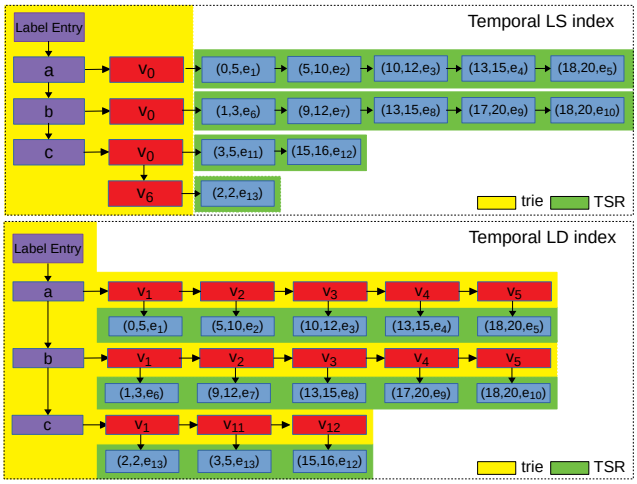

Fig. 4. The LS and LD structures of graph G_1. The yellow and green parts respectively refer to the ordering structure in a trie and TSRs

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_i, $Scan_{cur}[i]$ refers to the currently scanned edge in R_i. $Scan_{end}[i]$ refers to the ending of the edge-scanning in R_i. To start with, $Scan_{cur}$ is initialized at the first edge in TSRs (Line 2), which represent 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. $\min_{i \in [1,k]} \tau(Scan_{cur}[i].e).t_s$
7 $e \leftarrow sc.e$
8 **if** $\tau(e) \cap [w_s, w_e] \neq \emptyset$ **then**
9 $delActive(Active, \tau(e).t_s)$
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 $Result$

TABLE I
AN EXAMPLE OF LFTO ALGORITHM.

$Edge$	$Active$	$Enumerate$
e_1	$\emptyset$	$\emptyset$
e_6	$\emptyset$	$\emptyset$
e_{11}	$\emptyset$	$\emptyset$
e_2	[1]:$\{e_2\}$	$\emptyset$
e_7	[1]:$\{e_2\}$,[2]:$\{e_7\}$	$\emptyset$
e_3	[1]:$\{e_2, e_3\}$, [2] : $\{e_7\}$	$\emptyset$
e_4	[1]:$\{e_4\}$	$\emptyset$
e_8	[1]:$\{e_4\}$,[2]:$\{e_8\}$	$\emptyset$
e_{12}	[1]:$\{e_4\}$,[2]:$\{e_8\}$,[3]:$\{e_{12}\}$	$(e_4, e_8, e_{12}, [15, 15])$
e_9	[2]:$\{e_9\}$	$\emptyset$
e_{10}	[2]:$\{e_9, e_{10}\}$	$\emptyset$
e_5	[1]:$\{e_5\}$,[2]:$\{e_9, e_{10}\}$	$\emptyset$

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

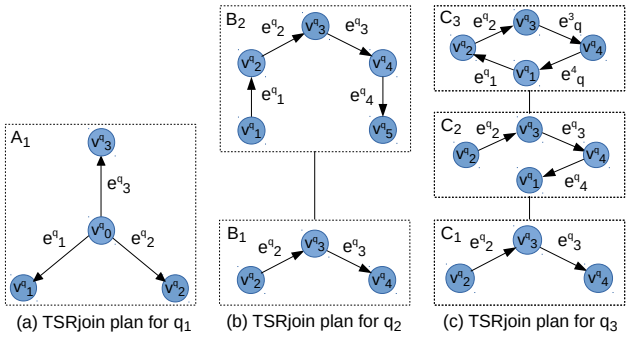

Fig. 5. TSRJOIN plans for queries q_1, q_2, and q_3.

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^q-centered 2-star pattern (i.e. (e_2^q, e_3^q)) since v_3^q is the most selective query vertex according to our cost model. This determines edge matches for e_2^q, e_3^q and bindings for v_2^q, v_3^q, v_4^q. B_2 extends each partial match produced by B_1 with e_1^q, e_4^q 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^q. 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_2^q,e_3^q) 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^q,v_4^q respectively. Based on the partial match, B_2 would process a 2-way interval join over $R(a, *, v_2), R(b, v_4, *)$, matches (e_1^q, e_4^q) with (e_4, e_3), (e_4, e_5), (e_6, e_3), (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_5,[15,18]), (e_6,e_1,e_2,e_3,[13,15]), (e_6,e_1,e_2,e_5,[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^q-centered 2-star pattern as in q_2. Then C_2 extends each match propagated from C_1 with e_4^q to find edge matches for e_4^q and determine bindings for v_1^q. Finally, C_3 extends each match propagated from C_2 with e_1^q 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 $O(Q_{\mathsf{P}}^* \log M + Q_{\mathsf{P}}^* \cdot k_{max} \cdot |R_{max}| \log |R_{max}| + P \cdot k_{max} \log M)$. $Q_{\mathsf{P}}^* \log M$ 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. $Q_{\mathsf{P}}^* \cdot k_{max} \cdot |R_{max}| \log |R_{max}|$ 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. $P \cdot k_{max} \log M$ is additional cost of filtering based on the cardinality P of partial matches produced by TSRjoins. Compared to the existing methods following P^{T} 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_5 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 $\{e_9, e_{10}, e_5\}$ 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 $R(l, s, *)$, $R(l, *, d)$, and $R(l, s, d)$. 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,

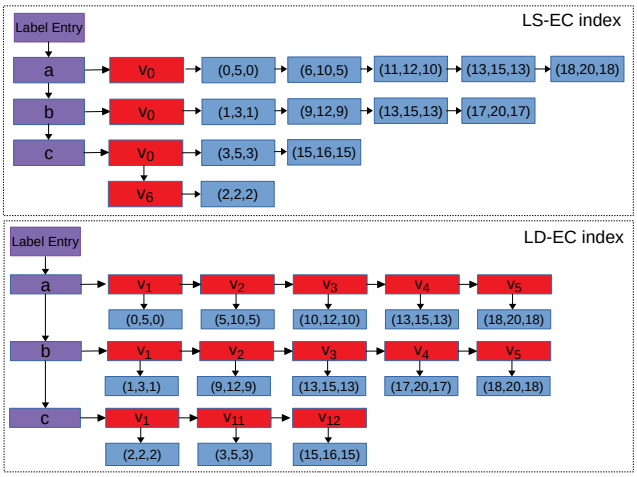

Fig. 6. The LS-EC and LD-EC structures of graph G_1.

return the first coverage tuple from correpsonding ECI such that $c_s > t$. 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 $getCoverageTuple(R(a, v_0, *), 1)$, tuple $(0, 5, 0)$ in LS-EC is returned. $(0, 5, 0)$ represents that in TSR $R(a, v_0, *)$, the earliest concurrent of time $t \in [0, 5]$ is 0. In this way, we obtain that $eC(1) = 0$. $getCoverageTuple(R(b, *, v_1), 1)$, returns $(1, 3, 1)$ in LD-EC.

The construction complexity of ECIs are $O(|L| + |V|^2 + |E| + \Theta \cdot \log \Theta)$, where Θ 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 $Scan_{cur}$s. The basic idea of the algorithm is to find the first collection of coverage tuples $\theta_1 \dots \theta_k$ for $R_1 \dots R_k$ such that the intersection of $[\theta_1.ec, \theta_1.c_e] \dots [\theta_k.ec, \theta_k.c_e]$ is not empty. Based on the notation of earliest concurrent, it can be deduced that each coverage tuple θ_i reveals that the longest interval in R_i starting at $t = \theta_i$.ec is [θ_i.ec, $\theta_i.c_e$]. In this way, the first non-empty $[\theta_1.ec, \theta_1.c_e] \cap \dots \cap [\theta_k.ec, \theta_k.c_e]$ in fact demonstrates that the first match is going to produced at the maximal time among $\theta_1.ec \dots \theta_k.ec$. Also, edges in R_i with start time no smaller than $\theta_i.ec$ can be relevant to partial matches. In this way, $Scan_{cur}[i]$ can start from the $\theta_i.ec$, 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 $R_1, \dots, R_k$, start time of valid window w_s
Output: optimized starting point $t_i^s, \dots, t_k^s$

```
1  t ← w_s
2  while true do
3      for i ∈ [1, k] do
4          θ_i ← getCoverageTuple(R_i, t_0)
5          if θ_i = ∅ then
6              return −1, ... − 1
7      if [θ_1.ec, θ_1.c_e] ∩ ... ∩ [θ_k.ec, θ_k.c_e] ≠ ∅ then
8          break
9      t ← max_{i∈[1,k]} θ_i.ec
10 return θ_1.ec, ..., θ_k.ec
```

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 ∈ [1, k] do
2      for e ∈ Active[i] do
3          if τ(e).t_e ≥ t then break
4          Active[i] ← Active[i]/{e}
5      if Active[i] = ∅ 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 $C \neq \emptyset$.

Considering n edges in R_i starting at time t, the complexity of enumeration in Algorithm 1 is $O(n \cdot \prod_{j=1}^{i-1} |Active(j)| \cdot \prod_{j=i+1}^{k} |Active(j)|)$. Using the lazy enumeration, the complexity is reduced to $O(\prod_{j=1}^{i-1} |Active(j)| \cdot \prod_{j=i+1}^{k} |Active(j)|)$. 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

$t_s[1, \ldots, k] \leftarrow OptimizeStartPoint(R_1 \ldots R_k, w_s)$
if $t_s[1] = -1$ **then** return $\emptyset$
for $i \in [1, k]$ **do**
  $Scan_{cur}[i] \leftarrow R_i.lower(t_s[i])$
  $Scan_{end}[i] \leftarrow R_i.edges.upper(w_e)$
$Active \leftarrow \emptyset, Result \leftarrow \emptyset, C \leftarrow \emptyset$
$inRange \leftarrow false$, $t' \leftarrow 0$, $i' \leftarrow -1$
while $\exists j \in [1, k]$ s.t. $Scan_{cur}[j]$ is not closed **do**
  $sc \leftarrow Scan_{cur}[i]$ s.t. $\min_{i \in [1,k]} \tau(Scan_{cur}[i].e).t_s$
  $e \leftarrow sc.e$
  if $e.t_s < w_s$ **then**
    if $e.t_e >= w_s$ **then** $insActive(Active, e, i)$
  else
    if $t' \neq e.t_e$ or $i' \neq i$ **then**
      if $inRange = false$ **then**
        $Result \leftarrow Result \cup enumLazy(Active, \emptyset)$
        $inRange \leftarrow true$
      else
        if $delSkip(Active, t', Scan_{cur}) = false$ **then** break
        $Result \leftarrow Result \cup enumLazy(Active, C)$
        $C \leftarrow \emptyset$
    $insActive(Active, e, i)$, $C \leftarrow C \cup \{e\}$
  $t' \leftarrow e.t_s, i' \leftarrow i, sc.next()$
  if $Scan_{cur}[i] = Scan_{end}[i]$ **then**
    Close $Scan_{cur}[i]$ and $Scan_{end}[i]$
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 II
AN EXAMPLE OF THE OPTIMIZED LFTO ALGORITHM.

$Edge$	$Active$	$Enumerate$
e_4	[1]:{e_4}	$\emptyset$
e_8	[1]:{e_4},[2]:{e_8}	$\emptyset$
e_{12}	[1]:{e_4},[2]:{e_8},[3]:{e_{12}}	$(e_4, e_8, e_{12}, [15, 15])$
e_9	[2] : {e_9}	$\emptyset$

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.[1] 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^{T} 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[2] are constructed to fully cover the access patterns during topological subgraph matching. In TIME,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

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

[2] Label adjacency index is a B-tree structure where graph edges are sorted in LSD or LDS order to represent the static adjacency.

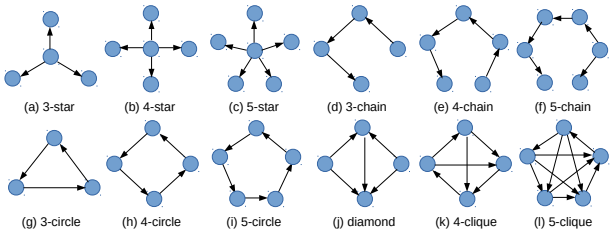

Fig. 7. Subgraph patterns used in the experiment evaluation.

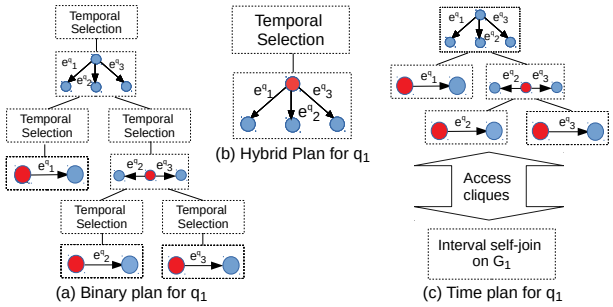

Fig. 8. Examples of plans used in each competitor. Red vertices highlight the topological joins.

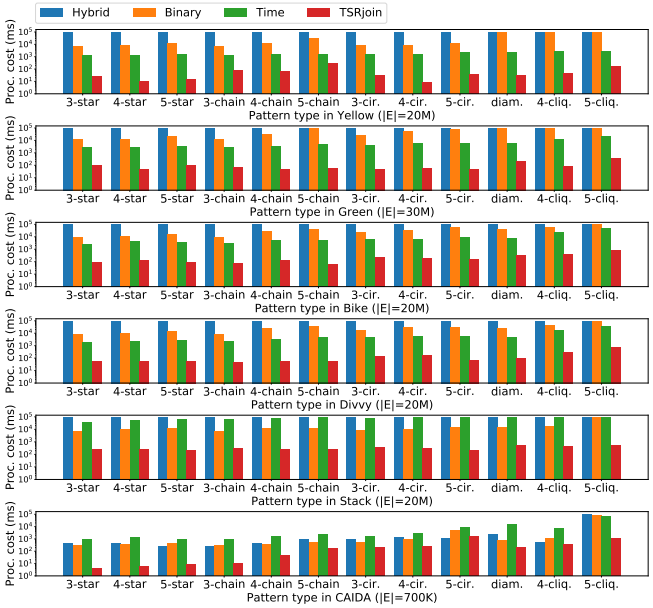

Fig. 9. Performance of algorithms with respect to pattern types.

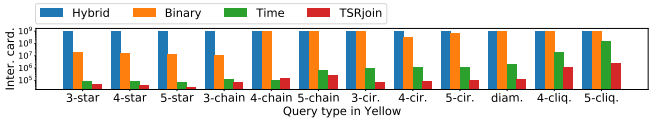

Fig. 10. Intermediate cardinality of various subgraph patterns in **Yellow** dataset ($|E| = 20M$).

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

TABLE III
OVERVIEW OF THE REAL-WORLD NETWORKS USED IN THE EXPERIMENTS.

Name	Num. of vertices	Num. of edges	Len. of domain
Yellow	261	20,000,000	89,741
Green	262	30,000,000	1,636,978
Bike	455	20,000,000	1,207,485
Divvy	1,097	20,000,000	3,245,197
Stack	2,465,111	20,000,000	961,820
CAIDA	31,379	714,016	121

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

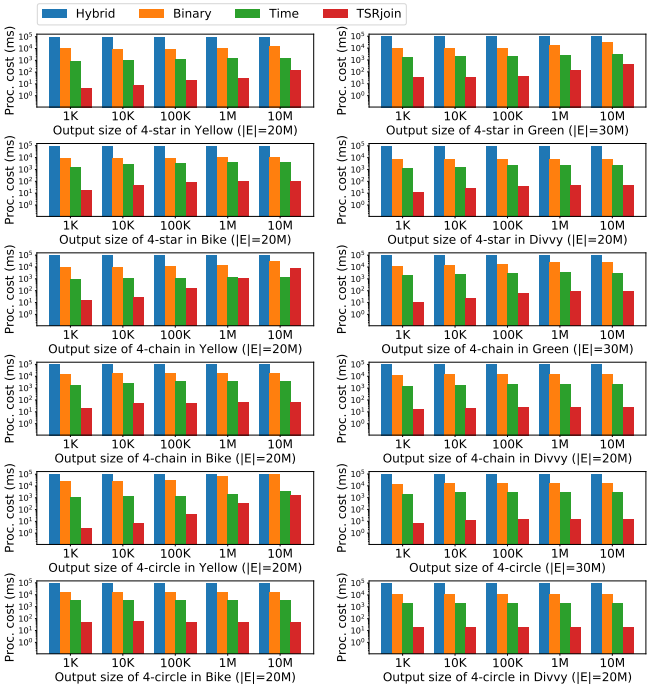

Fig. 11. Performance of algorithms with respect to query output size.

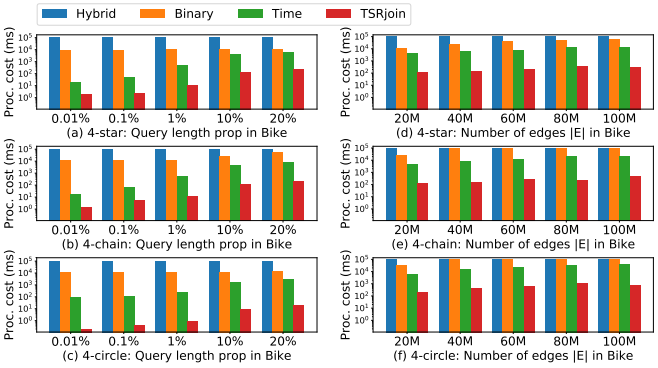

Fig. 12. Performance of algorithms with respect to the query window ((a)∼(c)) and network size ((d)∼(e)).

TABLE IV
STORAGE COST OF ALGORITHMS IN VARIOUS NETWORKS (GB).

Method	BINARY	HYBRID	TIME	TSRJOIN
Index	label adjacency	label adjacency	STI-CP	TAIs+ECIs
Yellow	4.5	4.5	5.0	7.4
Green	6.4	6.4	7.5	11.8
Bike	4.5	4.5	5.2	9.1
Divvy	4.5	4.5	4.8	9.0
Stack	5.9	5.9	4.5	17.4
CAIDA	0.2	0.2	0.2	0.4

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^{T} 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_4^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 v_2^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 V
PRE-PROCESSING COST OF ALGORITHMS IN VARIOUS NETWORKS (SECS).

Method	BINARY	HYBRID	TIME	TSRJOIN
Index	label adjacency	label adjacency	STI-CP	TAIs+ECIs
Yellow	60.3	61.2	87.7	118.9
Green	97.5	88.2	125.7	172.0
Bike	86.0	81.0	84.9	145.7
Divvy	86.3	77.2	79.7	133.4
Stack	66.6	67.8	270.0	196.8
CAIDA	1.2	1.2	7.0	4.1

algorithm in various networks. We note that in most networks, TSRJOIN requires at most twice as much space that its competitors in the worst case. And, in **Stack**, TSRJOIN requires higher space consumption. This is expected since TSRJOIN materializes additional structures to support efficient query processing. Comparing to BINARY and HYBRID in which label adjacency index is constructed, TSRJOIN 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, TSRJOIN 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 TSRJOIN. 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 TSRJOIN 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 TSRJOIN approach for temporal-clique subgraph query processing in modern graph database engine. TSRJOIN is designed to take full advantage of the selectivity of both topological and temporal predicates in query processing. Our experimental study demonstrated that TSRJOIN 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 TSRJOIN plans to resolve performance issues of TSRJOIN-only plans on certain graph patterns (e.g., selective chains).

REFERENCES

[1] The caida as relationships dataset, 2004-2007. https://www.caida.org/data/as-relationships/.

[2] Divvy System Data. https://www.divvybikes.com/system-data, 2019.

[3] NYC Citi Bike. https://www.citibikenyc.com/system-data, 2019.

[4] NYC TLC Trip Record Data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page, 2019.

[5] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)*, 42(4):1–44, 2017.

[6] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. *Querying Graphs*. Morgan & Claypool Publishers, 2018.

[7] Panagiotis Bouros and Nikos Mamoulis. A forward scan based plane sweep algorithm for parallel interval joins. *PVLDB*, 10(11):1346–1357, 2017.

[8] Jaewook Byun, Sungpil Woo, and Daeyoung Kim. Chronograph: Enabling temporal graph traversals for efficient information diffusion analysis over time. *IEEE Transactions on Knowledge and Data Engineering*, 32(3):424–437, 2019.

[9] Maximilian Franzke, Tobias Emrich, Andreas Züfle, and Matthias Renz. Pattern search in temporal social networks. In *Proceedings of the 21st International Conference on Extending Database Technology*, 2018.

[10] Aidan Hogan, Cristian Riveros, Carlos Rojas, and Adrián Soto. A worst-case optimal join algorithm for sparql. In *International Semantic Web Conference*, pages 258–275. Springer, 2019.

[11] Rohit Kumar and Toon Calders. Finding simple temporal cycles in an interaction network. In *TD-LSG@ PKDD/ECML*, pages 3–6, 2017.

[12] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. Scalable subgraph enumeration in mapreduce. *Proceedings of the VLDB Endowment*, 8(10):974–985, 2015.

[13] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. Scalable distributed subgraph enumeration. *Proceedings of the VLDB Endowment*, 10(3):217–228, 2016.

[14] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[15] Patrick Mackey, Katherine Porterfield, Erin Fitzhenry, Sutanay Choudhury, and George Chin. A chronological edge-driven approach to temporal subgraph isomorphism. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 3972–3979. IEEE, 2018.

[16] Amine Mhedhbi and Semih Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *arXiv preprint arXiv:1903.02076*, 2019.

[17] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *Journal of the ACM (JACM)*, 65(3):1–40, 2018.

[18] Dung Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q Ngo, Christopher Ré, and Atri Rudra. Join processing for graph patterns: An old dog with new tricks. In *Proceedings of the GRADES'15*, pages 1–8. 2015.

[19] Danila Piatov, Sven Helmer, and Anton Dignös. An interval join optimized for modern hardware. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 1098–1109. IEEE, 2016.

[20] Todd Plantenga. Inexact subgraph isomorphism in mapreduce. *Journal of Parallel and Distributed Computing*, 73(2):164–175, 2013.

[21] Shriram Ramesh, Animesh Baranawal, and Yogesh Simmhan. A distributed path query engine for temporal property graphs. *arXiv preprint arXiv:2002.03274*, 2020.

[22] Konstantinos Semertzidis and Evaggelia Pitoura. Top-k durable graph pattern queries on temporal graphs. *IEEE Transactions on Knowledge and Data Engineering*, 31(1):181–194, 2018.

[23] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. Efficient subgraph matching on billion node graphs. *arXiv preprint arXiv:1205.6691*, 2012.

[24] Todd L Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. *arXiv preprint arXiv:1210.0481*, 2012.

[25] Huanhuan Wu, James Cheng, Yiping Ke, Silu Huang, Yuzhen Huang, and Hejun Wu. Efficient algorithms for temporal path computation. *IEEE Transactions on Knowledge and Data Engineering*, 28(11):2927–2942, 2016.

[26] Huanhuan Wu, Yuzhen Huang, James Cheng, Jinfeng Li, and Yiping Ke. Reachability and time-based path queries in temporal graphs. In *ICDE*, pages 145–156. IEEE, 2016.

[27] Yanxia Xu, Jinjing Huang, Liu An, Zhixu Li, and Zhao Lei. Time-constrained graph pattern matching in a large temporal graph. In *APWeb-WAIM*, 2017.

[28] Kaijie Zhu, George Fletcher, Nikolay Yakovets, Odysseas Papapetrou, and Yuqing Wu. Scalable temporal clique enumeration. In *Proceedings of the 16th International Symposium on Spatial and Temporal Databases (SSTD), Vienna, Austria*, 08 2019.