Space-Efficient Subgraph Search Over Streaming Graph With Timing Order Constraint

Youhuan Li, Lei Zou, M. Tamer Özsu, *Fellow, IEEE*, and Dongyan Zhao

Abstract—The growing popularity of dynamic applications such as social networks provides a promising way to detect valuable information in real time. These applications create high-speed data that can be easily modeled as streaming graph. Efficient analysis over these data is of great significance. In this paper, we study the subgraph (isomorphism) search over streaming graph data that obeys timing order constraints over the occurrence of edges in the stream. The sliding window model is employed to focus on the most recent data. We propose an efficient solution to answer subgraph search, introduce optimizations to greatly reduce the space cost, and design concurrency management to improve system throughput. Extensive experiments on real network traffic data and synthetic social streaming data shows that our solution outperforms comparative ones by one order of magnitude with less space cost.

Index Terms—Streaming graphs, subgraph, timing order

1 Introduction

A recent development is the proliferation of high throughput, dynamic graph-structured data in many applications, such as social media streams and computer network traffic data. Efficient analysis of *streaming graphs* of this type is of great significance for tasks such as detecting anomalous events (e.g., in Twitter) and detecting adversarial activities in computer networks. Algorithms for various types of workloads over streaming graphs have been investigated, such as subgraph search, path computation, and triangle counting [1], as well as general navigational querying [2]. Subgraph search is one of the most fundamental problems, especially subgraph isomorphism that provides an exact topological structure constraint for the search.

In this paper, we study subgraph (isomorphism) search over streaming graph data that obeys timing order constraints over the occurrence of edges in the stream. Specifically, in a query graph, there exist some timing order constraints between different query edges specifying that one edge in the match is required to come before (i.e., have a smaller timestamp than) another one in the match. The timing aspect of streaming data is important for queries where sequential order between the query edges is significant. The following examples demonstrate the usefulness of subgraph (isomorphism) search with timing order constraints over streaming graph data.

Example 1. Cyber-attack pattern.

Fig. 1 demonstrates the pipeline of the information exfiltration attack pattern. A victim browses a compromised website (at time t_1), which leads to downloading malware scripts (at time t_2) that establish communication with the botnet C&C server (at times t_3 and t_4). The victim registers itself at the C&C server at time t_3 and receives the command from the C&C server at time t_4. Finally, the victim executes the command to send exfiltrated data back to C&C server at time t_5. Obviously, the time points in the above example follow a strict timing order $t_1 < t_2 < t_3 < t_4 < t_5$. Therefore, an attack pattern is modelled as a graph pattern (Q) as well as the timing order constraints over edges of Q. If we can locate the pattern (based on the subgraph isomorphism semantic) in the network traffic data, it is possible to identify the malware C&C Servers. US communications company Verizon has analyzed 100,000 security incidents over the past decade that reveal that 90 percent of the incidents fall into ten attack patterns [3], which can be described as graph patterns.

Example 2. Credit-card-fraud pattern.

Fig. 2 presents a credit card fraud example over a series transactions modeled by graph. A criminal tries to illegally cash out money by conducting a phony deal together with a merchant and a middleman. He first sets up a credit pay to the merchant (t_1); and when the merchant receives the real payment from the bank (t_2), he will transfer the money to a middleman (t_3) who will further transfer the money back to the criminal (t_4) to finish cashing out the money (middleman may have more than one account forming a transfer path). This pattern ($t_1 < t_2 < t_3 < t_4$) can be easily modeled as a query graph with timing order constraints.

Interactions of real world event patterns tends to happen within a certain period of time. For example, cyber-attack

• *Youhuan Li is with the Peking University, Beijing 100871, China, and the National Engineering Laboratory for Big Data Analysis Technology and Application (PKU), Beijing 100871, China, and also with the Center for Data Science, Peking University, Beijing 100871, China. E-mail: liyouhuan@pku.edu.cn.*
• *Lei Zou is with the Peking University, Beijing 100871, China, also with the National Engineering Laboratory for Big Data Analysis Technology and Application (PKU), Beijing 100871, China. E-mail: zoulei@pku.edu.cn.*
• *M. Tamer Özsu is with the University of Waterloo, Waterloo, ON N2L 3G1, Canada. E-mail: tamer.ozsu@uwaterloo.ca.*
• *Dongyan Zhao is with the Peking University, Beijing 100871, China. E-mail: zhaody@pku.edu.cn.*

Manuscript received 12 February 2020; revised 28 August 2020; accepted 22 October 2020. Date of publication 4 November 2020; date of current version 5 August 2022.
(Corresponding author: Lei Zou.)
Recommended for acceptance by A. Khan.
Digital Object Identifier no. 10.1109/TKDE.2020.3035902

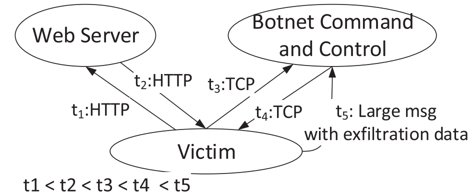


Fig. 1. Query example in network traffic (Taken from [1]).

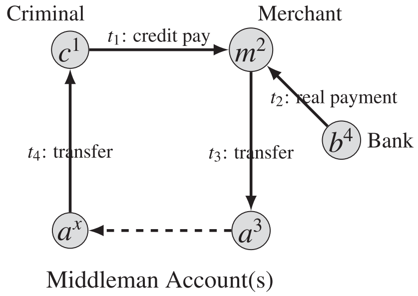


Fig. 2. Credit card fraud in transactions (Taken from [4]).

pattern usually happen within minutes even seconds. We, therefore, use sliding windows over the streaming graph to focus the most recent data.

1.1 Related Work

Although subgraph search has been extensively studied in literature [5], [6], [7], [8], [9], [10], most of these works focus on static graphs. Ullman [5] proposes a well-known subgraph isomorphism algorithm that is based on a state-space search approach; Cordella *et al.* [6] propose the VF2 algorithm that employs several important pruning strategies when searching for targeted subgraphs. Shang *et al.* [7] employ filtering and verification strategy for subgraph isomorphism. They propose QI-sequence to greatly reduce candidates from data graph before the verification phrase. Han *et al.* [8] transfer each query graph into a tree where they reduce duplicated subqueries to avoid redundant computation. They also utilize the tree to retrieve candidates from the data graph for further verification. Ren and Wang [9] define four vertex relationships over a query graph to reduce duplicate computation. Morari *et al.* [11] consider subgraph pattern over distributed semantic graphs and they apply multithreadling strategy to tolerate latency for communications.

The research on continuous query processing over high-speed streaming graph data is rather scarce. Fan *et al.* [12] propose an incremental solution for subgraph isomorphism based on repeated search over dynamic graph data, which cannot utilize previously computed results when new data come from the stream since they do not maintain any partial result. To avoid the high overhead in building complicated index, there is some work on approximate solution to subgraph isomorphism. Chen *et al.* [13] propose *node-neighbor tree* data structure to search multiple graph streams; they relax the exact match requirement and their solution needs to conduct significant processing on the graph streams. The input data that they consider is a sequence of small data graphs, which is not our focus. Gao *et al.* [14] study continuous subgraph search over a graph stream. They make specific assumptions over their query and their solution cannot guarantee exact answers for subgraph isomorphism. Pacaci *et al.* [2] propose an algorithm to answer navigational queries using the Recursive Path Query (RPQ) model.

Mackey *et al.* [15] consider subgraph search with timing order constraints. They require timing order in subgraph pattern to be total order, i.e., full chronological order over all edge. Also, they search the subgraph pattern only on static temporal graph instead of streaming graphs. Song *et al.* [16] is the first work to impose timing order constraint in streaming graphs, but the query semantics is based on *graph simulation* rather than *subgraph isomorphism*. The techniques for the former cannot be applied to the latter, since the semantics and, therefore, complexities are different. Furthermore, Song *et al.* perform post-processing to handle the timing constraints, i.e., finding all matches by ignoring the timing order constraints, and then filtering out the false positives based on the timing order constraints, which misses query optimization opportunities. Choudhury *et al.* [1] consider subgraph (isomorphic) match over streaming graphs, but this work ignores timing order constraints. They propose a subgraph join tree (SJ-tree) to maintain some intermediate results, where the root contains answers for the query while the other nodes store partial matches. This approach suffers from large space usage due to maintaining results. A similar work extends SJ-tree into distributed version with visualization enhancement [17].

A similar topic to continuous subgraph search is complex event processing (CEP) [18], which is a method of tracking and analyzing streams of information about things that happen, and deriving a conclusion from them. Timing constrained subgraph search can be expressed as time-constrained pattern in CEP. Each edge with a timestamp t could be expressed as a single event (an interaction between two objects, i.e., vertices) happening at t. Timing order between different edges could be expressed as the chronological order between different events. However, CEP does not consider the optimization strategy over graph structure data. Our solution formally and precisely defines target information requirement with subgraph and design optimization strategy over streaming graph to greatly improve the performance.

There are also incremental models for continuous subgraph search [19] that follow the append-only model that does not consider window and edge expiration. Designing of a data structure and algorithm that only need to consider incremental update is easier. However, computation over outdated data is unnecessary and adds to the latency and a mechanism to remove out-dated information (i.e., window model) is necessary.

Due to the high arrival rate of streaming graph data and the system's high throughput requirement, a concurrent computing (i.e., multi-threaded) algorithm is desirable – even required. It is not trivial to extend a serial single-threaded algorithm to a concurrent one, as it is necessary to guarantee the consistency of concurrent execution over streaming graphs.

1.2 Our Solution and Contributions

Our contributions are three-fold: (1) taking advantage of "timing order constraints" to reduce the search space, (2) compressing the space usage of intermediate results by

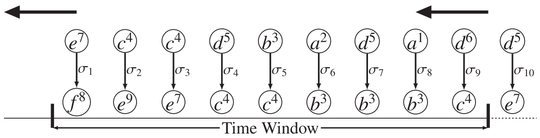


Fig. 3. Graph stream $\mathbb{G}$ under time window of size 9.

designing a trie-like data structure (called *match-store tree* and match-store DAG) and (3) proposing a concurrent computation framework with a fine-granularity locking strategy. The following is a summary of our methods and contributions:

Reducing Search Space. Considering the timing order constraints, we propose expansion list to avoid wasting time and space on *discardable partial matches*. Informally, an intermediate result (partial match) M is called "discardable" if M cannot be extended to a complete match of query Q no matter which edges would come in the future. Obviously, these should be pruned to improve the query performance. We define a query class, called *timing connected-query* (TC-query for short–see Definition 9) whose expansion list contains no discardable partial matches. We decompose a non-TC-query into a set of TC-queries and propose a two-step computation framework (Section 3).

Compressing Space Usage. The materialization of intermediate results inevitably increases space cost, which raises an inherent challenge to handling massive-scale, high-speed streaming graphs. We propose a trie variant data structure, called *match-store tree*, to maintain partial matches, which reduces both the space cost and the maintenance overhead without incurring extra data access burden (Section 4). Also, we further optimize MS-trees into a more condensed MS-DAG in Section 7.

Improving System Throughput. Existing works do not consider concurrent execution of continuous queries over streaming graphs. In a high-speed stream graphs, multiple edges may come at the same time. A naive solution is to process each edge one-at-a-time. In order to improve the throughput of the system, we propose to compute these edges concurrently. Concurrent computing may lead to conflicts and inconsistent results, which becomes even more challenging when different partial matches are compressed together on their common parts. We design a fine-granularity locking technique to guarantee the consistency of the results (Section 5).

Experiments show that our solution outperforms comparative ones by one order of magnitude. Also, our concurrency design is of good speedup and the time performance increase by more than three times.

2 Problem Definition

We list frequently-used notations in Table 1.

Definition 1 (Streaming Graph). *A streaming graph $\mathbb{G}$ is a constantly growing sequence of directed edges $\{\sigma_1, \sigma_2, ...\sigma_x\}$ where each σ_i arrives at time t_i ($t_i < t_j$ when $i < j$). t_i is also referred to as the timestamp of σ_i. Each edge σ_i has two labelled vertices and two edges are connected if and only if they share one common endpoint.*

For simplicity of presentation, we only consider vertex-labelled graphs and ignore edge labels, although handling the more general case is not more complicated.

TABLE 1
Frequently-Used Notations

Notation	Definition and Description
$\mathbb{G}$ / $\mathbb{G}_t$	Streaming graph / Snapshot at time point t
$\mathbb{E}_t$ / $\mathbb{V}_t$	Edge/Vertex set of $\mathbb{G}_t$
Q / $V(Q)$ / $E(Q)$	Continuous query / Query vertex set / Query edge set
ϵ_i / σ_i	Query edge / Data edge at time t_i
g	A subgraph of some snapshot
$\overrightarrow{uv}$	The directed edge from vertex u to v
W	Time window W
$\prec$	Timing order over query edges
$Preq(\epsilon_i)$	Prerequisite subquery of query edge ϵ_i
P_i	TC-subquery
$L_i (i > 0)$	Expansion list for TC-subquery P_i
L_0	Expansion list for joining matches of all TC-subqueries: $\{P_1, P_2,..., P_k\}$
L_i^j	The jth item in expansion list L_i
$\Omega(q)$	Matches of subquery q
$\Delta(q)$	New matches of subquery q
D	A decomposition (set of TC-subqueries) of query Q
$Ins(\sigma)$	Insertion for incoming edge σ
$Del(\sigma)$	Deletion for expired edge σ
$\setminus$ / $\setminus_i^j$	A node in a MS-tree / The jth node in the MS-tree for L_i
$TCsub(Q)$	The set of all TC-subqueries of query Q

An example of a streaming graph $\mathbb{G}$ is shown in Fig. 3. Note that edge σ_1 has two endpoints e^7 and f^8, where 'e' and 'f' are vertex labels and the superscripts are vertex IDs which we introduce to distinguish two vertices.

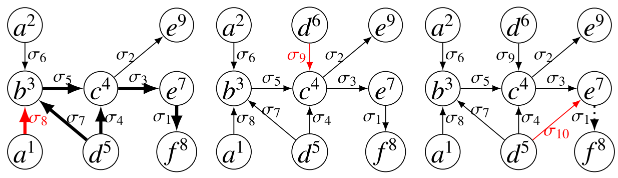
Definition 2 (Time-Based Sliding Window W). *Given current time t_i, a sliding window W defines a timespan $(t_i - |W|, t_i]$ with fixed duration $|W|$. All edges that occur in this time window form a consecutive block over the edge sequence.*

Obviously, as time window W slides, some edges may expire and some new edges may arrive.

Definition 3 (A Snapshot of a Streaming Graph). *Given a streaming graph $\mathbb{G}$ and a time window W at current time point t, the current snapshot of $\mathbb{G}$ is a graph $\mathbb{G}_t = (\mathbb{V}_t, \mathbb{E}_t)$ where $\mathbb{E}_t$ is the set of edges that occur in W and $\mathbb{V}_t$ is the set of vertices adjacent to edges in $\mathbb{E}_t$:*

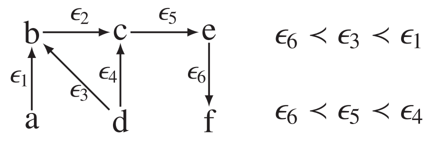
$$\mathbb{E}_t = \{\sigma_i | t_i \in (t - |W|, t]\}, \mathbb{V}_t = \{u | \overrightarrow{uv} \in \mathbb{E}_t \vee \overrightarrow{vu} \in \mathbb{E}_t\}.$$

The snapshots of graph stream $\mathbb{G}$ at time $t = 8, 9, 10$ for $|W| = 9$ are given in Fig. 4. At time $t = 10$, edge σ_1 expires since the time point of σ_1 is 1 and the timespan of time



(a) $t = 8$ (b) $t = 9$ (c) $t = 10$

Fig. 4. Graph stream under time window W of size 9.



(a) query graph (b) timing order

Fig. 5. Running example query Q.

window W is (1,10]. The expired edges are denoted with dotted edges while newly added edges are in red.

Definition 4 (Query Graph). *A query graph is a four-tuple $Q = (V(Q), E(Q), L, \prec)$, where $V(Q)$ is a set of vertices in Q, $E(Q)$ is a set of directed edges, L is a function that assigns a label for each vertex in $V(Q)$, and $\prec$ is a strict partial order relation over $E(Q)$, called the* timing order. *For $\epsilon_i, \epsilon_j \in E(Q)$, $\epsilon_i \prec \epsilon_j$ means that in a match g for Q where σ_i matches ϵ_i and σ_j matches ϵ_j ($\sigma_i, \sigma_j \in g$), timestamp of σ_i should be less than that of σ_j.*

An example of query graph Q is presented in Fig. 5. Any subgraph in the result must conform to the constraints on both structure and timing orders.

Definition 5 (Time-Constrained Match). *For a query Q and a subgraph g in current snapshot$\mathbb{G}_t$ formed by window W, g is a* time-constrained match *of Q if and only if there exists a bijective function F from $V(Q)$ to $V(g)$ such that the following conditions hold:*

1) Structure Constraint (Isomorphism)
   - $\forall u \in V(Q), L(u) = L(F(u))$.
   - $\overrightarrow{uv} \in E(Q) \Leftrightarrow \overrightarrow{F(u)F(v)} \in E(g)$.
2) Timing Order Constraint
   For any two edges $(\overrightarrow{u^{i_1}u^{i_2}}), (\overrightarrow{u^{j_1}u^{j_2}}) \in E(Q)$:

$$(\overrightarrow{u^{i_1}u^{i_2}}) \prec (\overrightarrow{u^{j_1}u^{j_2}}) \Rightarrow \overrightarrow{F(u^{i_1})F(u^{i_2})} \prec \overrightarrow{F(u^{j_1})F(u^{j_2})}.$$

Hence, the problem in this paper is to find all *time-constrained matches* of given query Q over each snapshot of graph stream $\mathbb{G}$ with window W. For simplicity, when the context is clear, we always use "match" to mean "time-constrained match".

For example, the subgraph g induced by edges σ_1, σ_3, σ_4, σ_5, σ_7 and σ_8 in Fig. 4a (highlighted by bold line) is not only isomorphic to query Q but also conforms to the timing order constraints defined in Fig. 5b. Thus, g is a match of query Q over stream $\mathbb{G}$ at time point $t = 8$. At time point $t = 10$, with the deletion of edge σ_1, g expires.

Theorem 1. *Subgraph isomorphism can be reduced to the proposed problem in polynomial time and therefore, the proposed problem is NP-hard.*

Proofs of lemmas and theorems are presented in Appendix A in the supplementary, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2020.3035902.

3 A Baseline Method

We propose a baseline solution that utilizes the timing order in reducing the search space. We first define and evaluate a class of queries (timing-connected query) in Section 3.1; we then discuss how to answer an arbitrary query in Section 3.2.

3.1 Timing-Connected Query

3.1.1 Intuition

A naive solution to executing a query Q with timing order is to run a classical subgraph isomorphism algorithm on each snapshot $\mathbb{G}_i$ ($i = 1, \ldots, \infty$) to first check the structure constraint followed by a check of the timing order constraint among the matches. However, an incoming/expired edge causes only a minor change between two consecutive snapshots $\mathbb{G}_i$ and $\mathbb{G}_{i-1}$; thus, it is wasteful to re-run the subgraph isomorphism algorithm from scratch on each snapshot. Therefore, we maintain *partial matches* of subqueries in the previous snapshots. Specifically, we only need to check whether there exist some partial matches (in the previous snapshots) that can join with an incoming edge σ to form new matches of query Q in the new snapshot $\mathbb{G}_i$. Similarly, we can delete all (partial) matches containing the expired edges at the new timestamp. For example, consider the query graph Q in Fig. 5. Assume that an incoming edge σ matches ϵ_1 at time point t_i. If we save all partial matches for subquery $Q\backslash\{\epsilon_1\}$, i.e., the subquery induced by edges $\{\epsilon_2, \epsilon_3, \epsilon_4, \epsilon_5, \epsilon_6\}$, at the previous time point t_{i-1} (i.e., $\mathbb{G}_{i-1}$), we only need to join σ with these partial matches to find new subgraph matches of query Q.

Although materializing partial matches can accelerate continuous subgraph query, this inevitably introduces considerable maintenance overhead. For example, in SJ-tree [1], each new coming edge σ requires updating the partial matches. In this section, we propose pruning *discardable* edges (see Definition 6) by considering the timing order in the query graph.

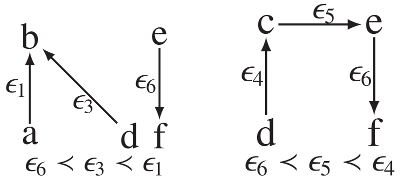
Definition 6 (Discardable Edge). *For a query graph Q and a streaming graph $\mathbb{G}$, an incoming edge σ is called a* discardable edge *if σ cannot be included in a complete match of Q, no matter what edges arrive in the future.*

To better understand discardable edges, recall the streaming graph $\mathbb{G}$ in Fig. 3. At time t_6, an incoming edge σ_6 (only matching ϵ_1) is added to the current time window. Consider the timing order constraints of query Q in Fig. 5, which requires that edges matching ϵ_3 should come before ones matching ϵ_1. However, there is no edge matching ϵ_3 before t_6 in $\mathbb{G}$. Therefore, it is impossible to generate a complete match (of Q) consisting of edge σ_6 (matching ϵ_1) no matter which edges come in the future. Thus, σ_6 is a *discardable edge* that can be filtered out safely. We design an effective solution to determine if an incoming edge σ is discardable. Before presenting our approach, we introduce an important definition.

Definition 7 (Prerequisite Edge/Subquery). *Given an edge ϵ in query graph Q, a set of* prerequisite edges *of ϵ (denoted as $Preq(\epsilon)$) are defined as follows:*

$$Preq(\epsilon) = \{\epsilon' | \epsilon' \prec \epsilon\} \cup \{\epsilon\},$$

where '$\prec$' denotes the timing order constraint as in Definition 4. The subquery of Q induced by edges in $Preq(\epsilon)$ is called a prerequisite subquery *of ϵ in query Q.*



(a) $Preq(\epsilon_1)$ (b) $Preq(\epsilon_4)$

Fig. 6. Example of prerequisite subquery.

Consider two edges ϵ_1 and ϵ_4 in query Q in Fig. 5. Prerequisite subqueries $Preq(\epsilon_1)$ and $Preq(\epsilon_4)$ are both illustrated in Fig. 6. The following lemma states the necessary and sufficient condition to determine whether an edge σ in streaming graph $\mathbb{G}$ is *discardable*.

Lemma 1. *An incoming edge σ at time t_i is* NOT discardable *if and only if, in $\mathbb{G}_{t_i}$, there exists at least one query edge ϵ ($\in Q$) such that (1) the prerequisite subquery $Preq(\epsilon)$ has at least one match g (subgraph of $\mathbb{G}_{t_i}$) containing σ; and (2) σ matches ϵ in the match relation between g and $Preq(\epsilon)$. Otherwise, σ is* discardable.

Lemma 1 can be used to verify whether or not an incoming edge σ is discardable. The straightforward way requires checking subgraph isomorphism between $Preq(\epsilon)$ and $\mathbb{G}_i$ in each snapshot graph, which is quite expensive. First, $Preq(\epsilon)$ may not be connected, even though query Q is connected. For example, $Preq(\epsilon_1)$ is disconnected. Computing subgraph isomorphism for disconnected queries will cause a Cartesian product among candidate intermediate results leading to lots of computation and huge space cost. Second, some different prerequisite subqueries may share common substructures, leading to common computation for different prerequisite subqueries. It is inefficient to compute subgraph isomorphism from scratch for each edge.

For certain types of queries that we call *timing-connected query* (Definition 9), it is easy to determine if an incoming edge σ is discardable. Therefore, we first focus on these queries for which we design an efficient query evaluation algorithm. We discuss non-TC-queries in Section 3.2.

We introduce the following concepts that will be used when illustrating our algorithm. Consider a query Q and two subqueries: Q^1, Q^2, assume that g_1 (g_2) is a time-constrained match of Q^1 (Q^2) in the current snapshot. Let F_1 and F_2 denote the *matching functions* (Definition 5) from $V(Q^1)$ and $V(Q^2)$ to $V(g_1)$ and $V(g_2)$, respectively. We say that g_1 is *compatible* with g_2 (denoted as $g_1 \sim g_2$) w.r.t Q^1 and Q^2 if and only if $g_1 \cup g_2$ is a time-constrained match of $Q^1 \cup Q^2$ on bijective match function $F_1 \cup F_2$. Furthermore, let $\Omega(Q^1)$ and $\Omega(Q^2)$ denote the set of matches of Q^1 and Q^2 in current snapshot, respectively. We define a new join operation over $\Omega(Q^1)$ and $\Omega(Q^2)$, denoted as $\Omega(Q^1) \overset{T}{\bowtie} \Omega(Q^2)$, as follows:

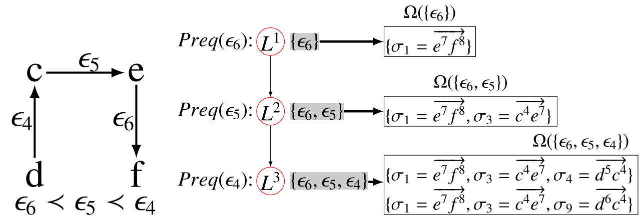
$$\Omega(Q^1) \overset{T}{\bowtie} \Omega(Q^2 = \{g_1 \cup g_2 | g_1 \in \Omega(Q^1) \sim g_2 \in \Omega(Q^2)\}.$$

Note that when $g_1 \sim g_2$ and $Q^1 \cap Q^2 \neq \emptyset$, F_1 and F_2 will never map the same query vertex to different data vertices since we require $F_1 \cup F_2$ to be a bijective function.

3.1.2 TC-Query

Definition 8 (Prefix-Connected Sequence). *Given a query Q of k edges, a* prefix-connected sequence *of Q is a permutation of all edges in Q: $\{\epsilon_1, \epsilon_2 ..., \epsilon_k\}$ such that $\forall j \in [1, k]$, the subquery induced by the first j edges in $\{\epsilon_1\} \cup ... \cup \{\epsilon_j\}$ is always* weakly connected.

Definition 9 (Timing-Connected Query). *A query Q is called a timing-connected query* (TC-query, *for short) if there exists a prefix-connected sequence $\{\epsilon_1, \epsilon_2 ..., \epsilon_k\}$ of Q such that $\forall j \in [1, k-1]$, $\epsilon_j \prec \epsilon_{j+1}$. We call the sequence $\{\epsilon_1 ,..., \epsilon_k\}$ the* timing sequence *of TC-query Q.*



(a) TC-query (b) Expansion list

Fig. 7. A TC-query $\{\epsilon_6, \epsilon_5, \epsilon_4\}$ and timing expansion list.

Recall the running example Q in Fig. 5, which is not a TC-query. However, the subquery induced by edges $\{\epsilon_6, \epsilon_5, \epsilon_4\}$ is a TC-query, since $\epsilon_6 \prec \epsilon_5 \prec \epsilon_4$ and $\{\epsilon_6\}$, $\{\epsilon_6, \epsilon_5\}$ and $\{\epsilon_6, \epsilon_5, \epsilon_4\}$ are all connected.

Given a TC-query Q with timing sequence $\{\epsilon_1 ,..., \epsilon_k\}$, the prerequisite subquery $Preq(\epsilon_j)$ is exactly the subquery induced by the first j edges in $\{\epsilon_1, \epsilon_2 ,..., \epsilon_j\}$ ($j \in [1, k]$). $Preq(\epsilon_{j+1}) = Preq(\epsilon_j) \cup \{e_{j+1}\}$ and $\Omega(Preq(\epsilon_{j+1})) = \Omega(Preq(\epsilon_j)) \overset{T}{\bowtie} \Omega(\epsilon_{j+1})$, where $\Omega(Preq(\epsilon_{j+1}))$ denotes matches for prerequisite subquery $Preq(\epsilon_{j+1})$, $\Omega(\epsilon_{j+1})$ denotes the matching edges for ϵ_{j+1}.

3.1.3 TC-Query Evaluation

We propose an effective data structure, called *expansion list*, to evaluate a TC-query Q. An expansion list for TC-query (1) can efficiently determine whether or not an incoming edge is discardable, and (2) can be efficiently maintained (which guarantees the efficient maintenance of the answers for TC-query Q).

Definition 10 (Expansion List). *Given a TC-query Q with timing sequence $\{\epsilon_1, \epsilon_2 ,..., \epsilon_k\}$, an expansion list $L = \{L^1, L^2 ,..., L^k\}$ over Q is defined as follows:*

1) *Each L^i corresponds to $\bigcup_{j=1}^{i}(\epsilon_j)$, i.e., $Preq(\epsilon_i)$.*
2) *Each L^i records $\Omega(\bigcup_{j=1}^{i}(\epsilon_j))$, i.e., a set of partial matches (in the current snapshot) of prerequisite subquery $Preq(\epsilon_i)$ ($i \in [1, k]$). We also use $\Omega(L^i)$ to denote the set of partial matches in L^i.*

Note that each item L^j corresponds to a distinct subquery $Preq(\epsilon_j)$ and we may use the corresponding subquery to denote an item when the context is clear.

The shaded nodes in Fig. 7 illustrate the prerequisite subqueries for a TC-query with timing sequence $\{\epsilon_6, \epsilon_5, \epsilon_4\}$. Since each node corresponds to a subquery $Preq(\epsilon_i)$, we also record the matches of $Preq(\epsilon_i)$. The last item stores matches of the TC-query in the current snapshot.

Maintaining the expansion list requires updating (partial) matches associated with each item in the expansion list. An incoming edge may result in insertion of new (partial) matches into the expansion list while an expired edge may lead to deletion of partial matches containing the expired one. We will discuss these two cases separately.

Case 1: New Edge Arrival. For an incoming edge σ, Theorem 2 indicates the partial matches associated with the expansion list that should be updated.

Theorem 2. *Given a TC-query Q with the timing sequence $\{\epsilon_1, \epsilon_2, ..., \epsilon_k\}$ and the corresponding expansion list $L = \{L^1, L^2, ..., L^k\}$. If an incoming edge σ matches query edge ϵ_i in the current time window, then only the matches of L^i ($Preq(\epsilon_i)$) should be updated.*

1) *If $i = 1$, σ should be inserted into L^1 as a new match of $Preq(\epsilon_1)$ since $Preq(\epsilon_1) = \{\epsilon_1\}$.*
2) *If $i \neq 1 \wedge \Omega(L^{i-1}) \overset{T}{\bowtie} \{\sigma\} \neq \emptyset$, then $\Omega(L^{i-1}) \overset{T}{\bowtie} \{\sigma\}$ contains new matches of $Preq(\epsilon_i)$ to be inserted into L^i. $\Omega(L^{i-1})$ is the set of partial matches in L^{i-1}.*

Hence, for a TC-query $Q = \{\epsilon_1, \epsilon_2 ..., \epsilon_k\}$ and the expansion list $L = \{L^1, L^2, ..., L^k\}$, the maintenance of L for an incoming edge σ can be done as follows:

1) if σ matches no query edge, discard σ;
2) if σ matches ϵ_1, then add σ into L^1;
3) if σ matches ϵ_i $(i > 1)$, then compute $\Omega(L^{i-1}) \overset{T}{\bowtie} \{\sigma\}$. If the join result is not empty, add all resulting (partial) matches (of $Preq(\epsilon_i)$) into L^i.

The above process is codified in Lines 1-10 of Algorithm 1. Note that an incoming edge σ may match multiple query edges; the above process is repeated for each matching edge ϵ. New matches that are inserted into the last item of the expansion list are exactly the new matches of TC-query Q.

Algorithm 1. INSERT(σ)

Input: σ: incoming edge to be inserted
Input: $L_i = \{L_i^1, L_i^2, \ldots, L_i^{|Q^i|}\}$: the expansion list for Q^i
Input: $L_0 = \{L_0^1, L_0^2, \ldots, L_0^k\}$: the expansion list over $\{Q^1, Q^2, \ldots, Q^k\}$

1 **for** *each query edge ϵ that σ matches* **do**
2   Assume that ϵ is the jth edge in TC-subquery Q^i.
3   **if** $j == 1$
4   Insert σ into L_i^j
5   **else**
6   Let $\Delta(\epsilon) = \{\sigma\}$
7   READ(L_i^{j-1}) // **Read partial matches in** L_i^{j-1}
8     $\Delta(L_i^j) = \Delta(\epsilon) \overset{T}{\bowtie} \Omega(L_i^{j-1})$
9   **if** $\Delta(L_i^j) \neq \emptyset$ **then**
10       $L_i^j += \Delta(L_i^j)$ // **Insert** $\Delta(L_i^j)$ **into** L_i^j
11   **if** $j = |L_i|$ *AND* $\Delta(L_i^j) \neq \emptyset$ **then**
12   **if** $i = 1$ **then**
13   Let $\Delta(L_0^i) = \Delta(L_i^j)$
14   **else**
15   READ(L_0^{i-1})
16       $\Delta(L_0^i) = \Delta(L_i^j) \overset{T}{\bowtie} \Omega(L_0^{i-1})$
17       $L_0^i += \Delta(L_0^i)$ // **Insert** $\Delta(L_0^i)$ **into** L_0^i
18   **while** $i < k$ *AND* $\Delta(L_0^i) \neq \emptyset$ **do**
19   READ($L_{i+1}^{|L_{i+1}|}$) // **Read** $\Omega(Q^{i+1})$
20       $\Delta(L_0^{i+1}) = \Delta(L_0^i) \overset{T}{\bowtie} \Omega(L_{i+1}^{|L_{i+1}|})$
21       $L_0^{i+1} += \Delta(L_0^{i+1})$
22       $i++$
23   **Report** $\Delta(L_0^k)$ **(if not** $\emptyset$**) as new matches of** Q

Case 2: Edge Expiry. When an edge σ expires, we can remove all expired partial matches (containing σ) in expansion list L by scanning L^1 to L^j where L^j is the rightmost item in L which contains expired partial matches.

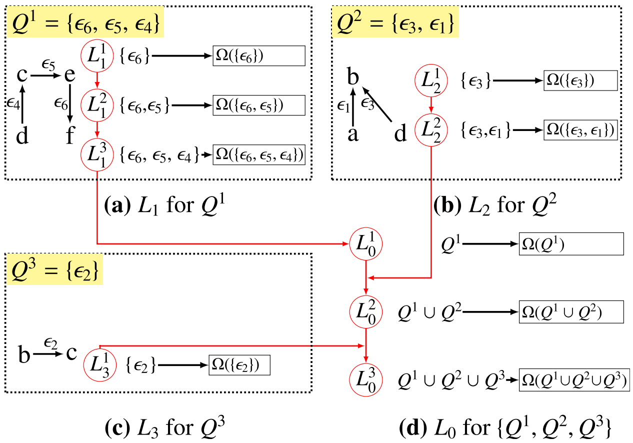


Fig. 8. An TC decomposition of query Q.

3.2 Answering Non-TC-Queries

We decompose a non-TC-query Q into a set of subqueries $D = \{Q^1, Q^2, ...Q^k\}$, where each Q^i is a TC-subquery, $Q = \bigcup_{i=1}^{k}(Q^k)$ and there is no common query edge between any two TC-subqueries. We call D a *TC decomposition* of Q. The example query Q is decomposed into $\{Q^1, Q^2, Q^3\}$, as shown in Fig. 8. Since each TC-subquery Q^i can be efficiently evaluated as described in the previous section, we focus on how to join those matches of Q^i $(i = 1, \ldots, k)$ into matches of Q in the stream scenario.

For the sake of presentation, we assume that the decomposition of query Q is given; decomposition is further discussed in Section 6.1. We use $L_i = \{L_i^1, L_i^2, ..., L_i^{|E(Q^i)|}\}$ to denote the corresponding expansion list for each TC-subquery Q^i. Recall the definition of prefix-connected sequence (Definition 8). We can find a permutation of D whose prefix sequence always constitutes a weakly connected subquery of Q as follows: we first randomly extract a TC-subquery Q^1 from D; and then we extract a second TC-subquery Q^2 who have common vertex with Q^1 (Since Q is weakly connected, we can always find such Q^2); repeatedly, we can always extract another TC-subquery from D who have common vertex with some previously extracted TC-subquery and finally form a prefix-connected permutation of D. Without loss of generality, we assume that $\{Q^1, Q^2, ..., Q^k\}$ is a prefix-connected permutation of D where the subquery induced by $\{Q^1, Q^2, ..., Q^i\}$ is always weakly connected $(1 \leq i \leq k)$. Actually, the prefix-connected permutation corresponds to a join order, based on which, we can obtain $\Omega(Q)$ by joining matches of each Q^i. Different join orders lead to different intermediate result sizes, resulting in different performance. We do not discuss join order selection in this paper due to space constraints; this is a well-understood problem. We include our approach to the problem in Appendix D, available in the online supplemental material. For this paper, we assume that the prefix-connected sequence $D = \{Q^1, Q^2, ..., Q^k\}$ is given.

For example, Fig. 8 illustrates a decomposition of query Q (Q^1, Q^2, Q^3). We obtain the matches of Q as $\Omega(Q) = \Omega(Q^1) \overset{T}{\bowtie} \Omega(Q^2) ... \overset{T}{\bowtie} \Omega(Q^k)$. As in TC-query, we can materialize some intermediate join results to speed up online processing. According to the prefix-connected sequence over Q, we can define the expansion list, denoted as L_0 for the entire query Q (similar to TC-query). For example, the corresponding expansion list $L_0 = \{L_0^1, L_0^2, L_0^3\}$ (for query Q) is given in Fig. 8. Each item L_0^i records the intermediate join results $\Omega(\bigcup_{x=1}^{i} Q^x)$.

Algorithm 2. Join With Timing Order Constraints

Input: Query Q, subqueries Q^1, Q^2 $(E(Q^1) \cap E(Q^2) = \emptyset)$
Input: $\Omega(Q^1)$: matches of Q^1
Input: $\Omega(Q^2)$: matches of Q^2
Output: $\Omega(Q^1) \overset{T}{\bowtie} \Omega(Q^2)$

Let $J = \emptyset$
for *each* $g_1 \in \Omega(Q^1)$ **do**
  for *each* $g_2 \in \Omega(Q^2)$ **do**
    F_1 is the bijective function from $V(Q^1)$ to $V(g_1)$
    F_2 is the bijective function from $V(Q^2)$ to $V(g_2)$
    if *$F_1 \cup F_2$ is bijective AND TimeCheck(g_1, g_2)* **then**
      $J = J \cup \{g_1 \cup g_2\}$
  RETURN J as $\Omega(Q^1) \overset{T}{\bowtie} \Omega(Q^2)$
Function (TimeCheck(g_1, g_2))
  for *Each* $\overrightarrow{u_1v_1} \prec \overrightarrow{u_2v_2}$ *where* $\overrightarrow{u_1v_1} \in E(Q^1)$ *and* $\overrightarrow{u_2v_2} \in E(Q^2)$ **do**
    Let t_i, t_j be the timestamps of edges $(F_1(u_1), F_1(v_1))$ and $(F_2(u_2), F_2(v_2))$, respectively
    if $t_i > t_j$ **then**
      RETURN $false$
  end function

Assume that an incoming edge σ contributes to new matches of TC-subquery Q^i_T(denoted as $\Delta(L_i^{|L_i|})$). If $i > 1$, we let $\Delta(L_0^i) = \Delta(L_i^{|L_i|}) \bowtie \Omega(L_0^{i-1})$ (Line 16 in Algorithm 1). If $\Delta(L_0^i) \neq \emptyset$, we insert $\Delta(L_0^i)$ into L_0^i as new matches of L_0^i . Then, $\Delta(L_0^i) \overset{T}{\bowtie} \Omega(Q^{i+1})$ may not be empty and the join results (if any) are new partial matches that should be stored in L_0^{i+1} ($\bigcup_{x=1}^{i+1}(Q^x)$). Thus, we need to further perform $\Delta(L_0^i) \overset{T}{\bowtie} \Omega(L_{i+1}^{|L_{i+1}|})$ to get new partial matches (denoted as $\Delta(L_0^{i+1})$) and insert them into L_0^{i+1} as new matches of $\bigcup_{x=1}^{i+1}(Q^x)$. We repeat the above process until no new partial matches are created (Lines 18-22). Note that when partial matches of different subqueries are joined, we verify both structure and timing order constraints.

When an edge σ expires where σ matches $\epsilon \in Q^i$, we discard all partial matches containing σ in expansion list L_i as illustrated previously. If there are expired matches for Q^i (i.e., matches of Q^i that contain σ), then we also scan L_0^i to L_0^k to delete partial matches containing σ.

3.3 Correctness Analysis

We discuss the correctness of our solution. Consider a query Q with decomposition $\{Q^1, Q^2, \ldots, Q^k\}$. For deletion, when an edge σ expires, the expired partial matches are exactly those containing σ, hence our deletion strategy is obviously correct. For insertion of incoming edge σ, we need to figure out all new partial matches resulting from σ and insert them into corresponding expansion lists. Consider the case when σ matches ϵ which is the jth edge in Q^i. There are two key parts to insertion: updating L_x where $0 < x \leq k$ (Lines 6-10 in Algorithm 1) and updating L_0 (Lines 12-22 Algorithm 1). Specifically, for updating L_x where $0 < x \leq k$, Theorem 2 tells us that we only need to add new partial matches $\Delta(L_i^j) = \{\sigma\} \overset{T}{\bowtie} \Omega(L_i^{j-1})$ into L_i^j. For updating L_0, according to the construction of L_0 in Section 3.2, new partial matches for each $L_0^{i'}$ $(i \leq i' \leq k)$ can be computed by $\Delta(L_0^{i'}) = \Delta(L_0^{i'-1}) \overset{T}{\bowtie} \Omega(L_{i'}^{|L_{i'}|})$. We can see that the key to correctness of these two parts lies in how the new join operation $\overset{T}{\bowtie}$ guarantees the time-constrained match (Definition 5). We present the pseudocode for the new join operation in Algorithm 2 and we prove in Theorem 3 that the new join operation guarantees the time-constrained match.

Theorem 3. *Given a query Q and two subqueries Q^1, Q^2 $(E(Q^1) \cap E(Q^2) = \emptyset)$, consider $g_1 \in \Omega(Q^1)$ and $g_2 \in \Omega(Q^2)$ where F_1 and F_2 are the* matching functions *(Definition 5) from $V(Q^1)$ and $V(Q^2)$ to $V(g_1)$ and $V(g_2)$, respectively. $g_1 \cup g_2$ is a time-constrained match of $Q^1 \cup Q^2$ if and only if the following conditions hold:*

1) *$F_1 \cup F_2$ is bijective*
2) *For each $\overrightarrow{u_1v_1} \prec \overrightarrow{u_2v_2}$ where $\overrightarrow{u_1v_1} \in E(Q^1)$ and $\overrightarrow{u_2v_2} \in E(Q^2)$, edge $(F_1(u_1), F_1(v_1))$ has a smaller timestamp than that of edge $(F_2(u_2), F_2(v_2))$.*

4 Match-Store Tree

We propose a tree data structure, called match-store tree (MS-tree, for short), to reduce the space cost of storing partial matches in an expansion list. Each tree corresponds to an expansion list. We first formally define MS-tree and then illustrate how to access partial matches in MS-tree for the computation.

4.1 Match-Store Tree

Consider an expansion list $L = \{L^1, L^2, \ldots, L^k\}$ over timing sequence $\{\epsilon_1, \epsilon_2, \ldots, \epsilon_k\}$ where L^i stores all partial matches of $\{\epsilon_1, \epsilon_2, \ldots, \epsilon_i\}$. For a match g of L^i $(1 \leq i \leq k)$, g can be naturally presented in a *sequential form*: $\{\sigma_1, \sigma_2, .., \sigma_i\}$ where $g = \bigcup_{j=1}^{i}(\sigma_j)$ and each $\sigma_{i'}$ $(1 \leq i' \leq i)$ is a match of $\epsilon_{i'}$. Furthermore, $g' = g \setminus \{\sigma_i\} = \{\sigma_1, \sigma_2, .., \sigma_{i-1}\}$, as a match of $\{\epsilon_1, \epsilon_2, \ldots, \epsilon_{i-1}\}$, must be stored in L^{i-1}. Recursively, there must be $g'' = g' \setminus \{\sigma_{i-1}\}$ in L^{i-2}. For example, see the expansion list in Fig. 7. For partial match $\{\sigma_1, \sigma_3, \sigma_4\}$ in item $\{\epsilon_6, \epsilon_5, \epsilon_4\}$, there are matches $\{\sigma_1, \sigma_3\}$ and $\{\sigma_1\}$ in items $\{\epsilon_6, \epsilon_5\}$ and $\{\epsilon_6\}$ of the expansion list, respectively. These partial matches share a prefix sequence. Therefore, we propose a trie variant data structure to store the partial matches in the expansion list.

Definition 11 (Match-Store Tree). *Given a TC-query Q with timing sequence $\{\epsilon_1, \epsilon_2, \ldots, \epsilon_k\}$ and the corresponding expansion list $L = \{L^1, L^2, \ldots, L^k\}$, the* Match-Store tree (MS-tree) *M of L is a trie variant built over all partial matches in L that are in sequential form. Each node \ of depth i $(1 \leq i \leq k)$ in a MS-tree denotes a match of ϵ_i and all nodes along the path from the root to node \ together constitute a match of $\{\epsilon_1, \epsilon_2, \ldots, \epsilon_i\}$. Also, for each node \ of a MS-tree, \ records its parent node. Nodes of the same depth are linked together in a doubly linked list.*

For example, see the MS-tree for the expansion list for subquery Q^1 with the timing sequence $\{\epsilon_6, \epsilon_5, \epsilon_4\}$ in Fig. 9. The three matches ($\{\sigma_1\}$ for node $\{\epsilon_6\}$, $\{\sigma_1, \sigma_3\}$ for node $\{\epsilon_6, \epsilon_5\}$ and $\{\sigma_1, \sigma_3, \sigma_4\}$ for node $\{\epsilon_6, \epsilon_5, \epsilon_4\}$) are stored only in a path $(\sigma_1 \rightarrow \sigma_3 \rightarrow \sigma_4)$ in the MS-tree. Furthermore, partial match $\{\sigma_1, \sigma_3, \sigma_9\}$ shares the same prefix path $(\sigma_1 \rightarrow \sigma_3)$ with $\{\sigma_1,$

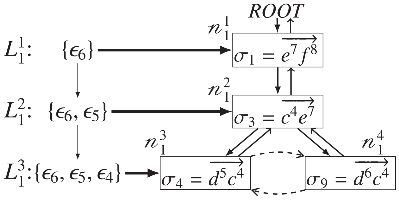


Fig. 9. MS-tree of expansion list $L_1 = \{L_1^1, L_1^2, L_1^3\}$.

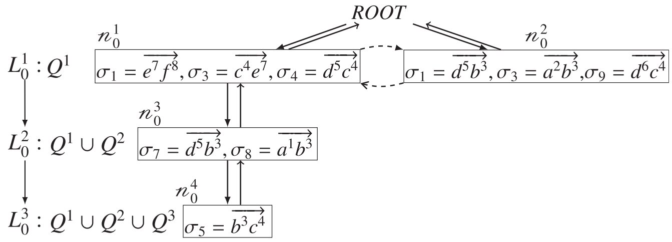


Fig. 10. MS-tree of expansion list L_0 for $\{Q^1, Q^2, Q^3\}$.

σ_3, $\sigma_4\}$. Thus, MS-tree greatly reduces the space cost for storing all matches by compressing the prefix. Apparently, MS-tree can be seamlessly defined over the expansion list for the decomposition of a non-TC-query. For example, the MS-tree for expansion list $\{L_0^1, L_0^2, L_0^3\}$ for whole query Q (see Fig. 8) is shown in Fig. 10. For convenience, we use M_i to denote the MS-tree for L_i ($0 \leq i \leq k$).

4.2 MS-Tree Accessibility

Given an expansion list $L = \{L^1, L^2, ..., L^k\}$ over timing sequence $\{\epsilon_1, \epsilon_2, ..., \epsilon_k\}$ and an MS-tree M that stores all partial matches in L, there are three operations that M needs to provide for computation: (1) reading all matches for some item L^i, i.e., $\Omega(L^i)$; (2) inserting a new match into some item L^i; (3) deleting expired partial matches (i.e.,those containing expired edge). These basic operations can be seamlessly applied to the MS-tree of expansion list L_0 over the decomposition of a non-TC-query.

Reading Matches of L^i. In a MS-tree, each i-length path starting from the root indicates a match of L^i, i.e., $\{\epsilon_1, \epsilon_2, ..., \epsilon_i\}$. We can obtain matches of L^i by enumerating nodes of depth i in M with the corresponding doubly linked list, and for each node of depth i, we can easily backtrack the i-length paths to get matches of L^i. Apparently, the time for reading partial matches in L^i is $O(|L^i|)$ where $|L^i|$ denotes the number of partial matches in L^i.

Inserting a New Match of L^i. For a new match of $\{\epsilon_1, \epsilon_2, ..., \epsilon_i\}$: $g = \{\sigma_1, \sigma_2, ..., \sigma_i\}$ where each σ_j matches ϵ_j, we need to insert a path $\{root \rightarrow \sigma_1 \rightarrow \sigma_2 ... \rightarrow \sigma_i\}$ into MS-tree. According to the insertion over expansion list, g must be obtained by $\{\sigma_1, \sigma_2, ..., \sigma_{i-1}\} \overset{T}{\bowtie} \{\sigma_i\}$ and there must already be a path $\{root \rightarrow \sigma_1 \rightarrow \sigma_2 ... \rightarrow \sigma_{i-1}\}$ in MS-tree. Thus, we can just add σ_i as a child of node σ_{i-1} to finish inserting g. For example, to insert a new match $\{\sigma_1, \sigma_3, \sigma_9\}$ of $\{\epsilon_6, \epsilon_5, \epsilon_4\}$, we only need to expand the path $\{root \rightarrow \sigma_1 \rightarrow \sigma_3\}$ by adding σ_9 as a child of σ_3 (see Fig. 9). We can easily record node σ_{i-1} when we find that $\{\sigma_1, \sigma_2, ..., \sigma_{i-1}\} \overset{T}{\bowtie} \{\sigma_i\}$ is not $\emptyset$, thus inserting a match of L^i cost $O(1)$ time. We can see that our insertion strategy does not need to wastefully access the whole path $\{root \rightarrow \sigma_1 \rightarrow \sigma_2 ... \rightarrow \sigma_{i-1}\}$ as the usual insertion of trie.

Deleting Expired Partial Matches. When an edge σ expires, we need to delete all partial matches containing σ. Nodes corresponding to expired partial matches in MS-tree are called *expired nodes* and we need to remove all expired nodes. Assuming that σ matches ϵ_i, nodes containing σ are exactly of depth i in M. These nodes, together with all their descendants, are exactly the set of expired nodes in M according to the Definition of MS-tree. We first remove all expired nodes of depth i (i.e., nodes which contain σ) from the corresponding doubly linked list, we further remove their children of depth $i+1$ from M. Recursively, we can remove all expired nodes from MS-tree. Consider the MS-tree in Fig. 9. When edge σ_1 (matching ϵ_6 in TC-query $\{\epsilon_6, \epsilon_5, \epsilon_4\}$) expires, we delete node σ_1 in the first level of MS-tree, after which we further delete its descendant nodes σ_3, σ_4 and σ_9 successively. When an edge expired, the time cost for the deletion update is linear to the number of the corresponding expired partial matches.

4.3 MS-tree and Trie

Although MS-tree is similar to trie, there are important differences between them.

From the perspective of data structure: Each node \ in MS-tree, besides the links to \'s children, includes extra links to \'s parent and siblings (doubly linked list). These extra links play an important role in reading matches of subqueries and avoiding inconsistency in the concurrent access over MS-tree (Section 5).

From the perspective of operation: All operations (search/insertion/deletion) over trie always begin at the root, but we often access MS-tree horizontally. Each level of MS-tree is linked from the corresponding item in the expansion list. For example in Fig. 10, when reading $\Omega(Q^1 \cup Q^2)$, we begin accessing from L_0^2 (in the expansion list L_0) and obtain all matches $\Omega(Q^1 \cup Q^2)$ by enumerating all nodes at the 2-nd level in the MS-tree with the corresponding doubly linked list, and then for each such node, we can easily backtrack the paths to the root to obtain the match of $\Omega(Q^1 \cup Q^2)$.

5 Concurrency Management

To achieve high performance, the proposed algorithms can (and should) be executed in a multi-thread way. Since multiple threads access the common data structure (i.e., expansion lists) concurrently, there is a need for concurrency management. Concurrent computing over MS-tree is challenging since many different partial matches share the same branches (prefixes). We propose a fine-grained locking strategy to improve the throughput of our solution with consistency guarantee. We first introduce the locking strategy over the expansion list without MS-tree in Sections 5.1 and 5.2 then illustrate how to apply the locking strategy over MS-tree in Section 5.3.

5.1 Intuition

Consider the example query Q in Fig. 5, which is decomposed into three TC-subqueries Q^1, Q^2 and Q^3 (see Fig. 8). Fig. 8 demonstrates expansion list L_i of each TC-subquery Q^i and the expansion list L_0 for the entire query Q. Assume that there are three incoming edges $\{\sigma_{11}, \sigma_{12}, \sigma_{13}\}$ (see Fig. 11) at consecutive time points. A conservative solution for inserting these three edges is to process each edge sequentially to avoid conflicts. However, as the following analysis shows, processing them in parallel does not lead to

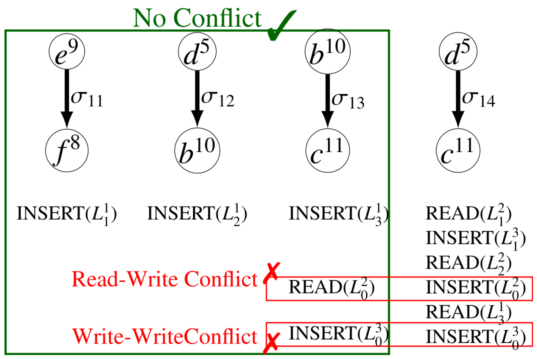


Fig. 11. Example of conflicts.

conflicts or wrong results. For convenience, insertion of an incoming edge σ_i is denoted as $Ins(\sigma_i)$ while deletion of an expired edge σ_j is denoted as $Del(\sigma_j)$.

Fig. 11 illustrates the steps of handling each incoming edge based on the discussion in Section 3. When σ_{11} is inserted (denoted as $Ins(\sigma_{11})$), σ_{11} matches query edge ϵ_6 and since ϵ_6 is the first edge in TC-subquery Q^1, we only need to insert match $\{\sigma_{11}\}$ into $\Omega(\epsilon_6)$ as the first item L_1^1 of expansion list L_1 (i.e., operation INSERT(L_1^1)). Similarly, handling $Ins(\sigma_{12})$ where σ_{12} matches ϵ_3 requires one operation: INSERT(L_2^1) (inserting $\{\sigma_{12}\}$ into $\Omega(\epsilon_3)$). For $Ins(\sigma_{13})$ where σ_{13} matches ϵ_2, we first insert σ_{13} into L_3^1 (INSERT(L_3^1)) as a new match of Q^3 (see Fig. 8) and then we need to join $\{\sigma_{13}\}$ with $\Omega(Q^1 \cup Q^2)$ (READ(L_0^2)) and insert join results into L_0^3 (INSERT(L_0^3)). Note that we consider the worst case in our analysis, namely, we always assume that the join result is not empty. Thus, to insert σ_{13}, we access the following expansion list items: INSERT(L_3^1), READ(L_0^2) and INSERT(L_0^3).

Fig. 11 shows that there is no common item to be accessed between $Ins(\sigma_{11})$, $Ins(\sigma_{12})$ and $Ins(\sigma_{13})$. Therefore, they can be processed concurrently.

Let us consider an incoming edge σ_{14} that matches $\{\epsilon_4\}$, which is the last edge in the timing sequence of TC-subquery Q^1. According to Algorithm 1, we need to read $\Omega(\{\epsilon_6, \epsilon_5\})$ and join $\Omega(\{\epsilon_6, \epsilon_5\})$ with $\{\sigma_{14}\}$. Since ϵ_4 is the last edge in Q^1, if $\Omega(\{\epsilon_6, \epsilon_5\}) \overset{T}{\bowtie} \{\sigma_{14}\} \neq \emptyset$, the join results are new matches of Q^1, and will be inserted into L_0^1. As discussed in Section 3.2, we need to join these new matches of Q^1 with $\Omega(Q^2)$ resulting in new matches of $Q^1 \cup Q^2$, which will be inserted into L_0^2. Finally, new matches of $Q^1 \cup Q^2$ will be further joined with $\Omega(Q^3)$, after which new matches of $Q^1 \cup Q^2 \cup Q^3$ will be inserted into L_0^3. Thus, the series of operations to be conducted for $Ins(\sigma_{14})$ are as follows: READ(L_1^2), INSERT(L_1^3), READ (L_2^2), INSERT(L_0^2), READ(L_3^1), INSERT(L_0^3). Obviously, $Ins(\sigma_{14})$ may conflict with $Ins(\sigma_{13})$ since both of them will conduct INSERT(L_0^3) as indicated in Fig. 11. Thus, the concurrent execution requires a locking mechanism to guarantee the consistency.

Definition 12 (Streaming Consistency). *For a streaming graph $\mathbb{G}$ with time window W and a query Q, the* streaming consistency *requires that at each time point, answers of Q are the same as the answers formed by executing insertion/deletion in chronological order of edges.*

Streaming consistency is different from *serializability*, since the latter only requires the output of the concurrent execution to be equivalent to some serial order of transaction execution, while streaming consistency specifies that the order must follow the timestamp order in $\mathbb{G}$. For example, a concurrent execution that executes $Ins(\sigma_{14})$ followed by $Ins(\sigma_{13})$ would be serializable but would violate streaming consistency.

5.2 Locking Mechanism and Schedule

We propose a locking mechanism to allow concurrent execution of the query execution algorithm while guaranteeing streaming consistency. The two main operations in streaming graphs, insertion of an incoming edge σ (i.e., $Ins(\sigma)$) and deletion of an expired edge σ' (i.e., $Del(\sigma')$), are modeled as *transactions*. Each transaction has a timestamp that is exactly the time when the corresponding operation happens. As discussed above, each edge insertion and deletion consists of elementary operations over items of the expansion lists, such as reading partial matches and inserting new partial matches. As analyzed in Section 5.1, concurrent execution of these operations may lead to conflicts that need to be guarded.

A naive solution is to lock all the expansion list items that may be accessed before launching the corresponding transaction. Obviously, this approach will degrade the system's degree of concurrency (DOC). For example, $Ins(\sigma_{13})$ and $Ins(\sigma_{14})$ conflict with each other only at items L_3^1, L_0^2 and L_0^3. The first three elementary operations of $Ins(\sigma_{13})$ and $Ins(\sigma_{14})$ can execute concurrently without causing any inconsistency. Thus, a finer-granularity locking strategy is desirable that allows higher DOC while guaranteeing streaming consistency. For example, in Fig. 11, INSERT(L_0^2) in $Ins(\sigma_{13})$ should be processed before the same operation in $Ins(\sigma_{14})$ to avoid inconsistency.

We execute each edge operation (inserting an incoming edge or deleting an expired edge) by an independent thread that is treated as a transaction, and there is a single main thread to launch each transaction. Items in expansion lists are regarded as "resources" over which threads conduct READ/INSERT/DELETE operations. Locks are associated with individual items in the expansion lists. An elementary operation (such as INSERT(L_3^1) in $Ins(\sigma_{13})$) accesses an item if and only if it has the corresponding lock over the item. The lock is released when the computation over L^j is finished. Note that deadlocks do not occur since each transaction only locks at most one item at a time.

Main Thread. Main thread is responsible for launching threads. Before launching a thread T, the main thread dispatches all *lock requests* of T to the *lock wait-lists* of the corresponding items. Specifically, a lock request is a triple $\langle tID, locktype, L^j\rangle$ indicating that thread tID requests a lock with type $locktype$ (shared – S, exclusive – X) over the corresponding item L^j . For each item L^j in expansion lists, we introduce a thread-safe wait-list consisting of all pending locks over L^j sorted according to the timestamps of transactions in the chronological order.

Since there is a single main thread, the lock request dispatch as well as thread launch is conducted in a serial way. Hence, when a lock request of a thread is appended to wait-list of an item L^j, then those lock requests of previous threads for L^j must have been in the wait-list since previous threads have been launched, which guarantees that lock requests in each wait-list are sorted in chronological order. Although thread launch is conducted in a serial way, once launched, all transaction threads are executed concurrently.

Transaction Thread Execution. Concurrently processing insertion/deletion follows the same steps as the sequential counterparts except for applying (releasing) locks before (after) reading (READ) or writing (INSERT/DELETE) expansion list items. Thus, in the remainder, we focus on discussing the lock and unlock processes. Note that, in this part, we assume that we materialize the partial matches ($\Omega(\cdot)$) using the naive representation (like Fig. 7) without MS-tree. The locking strategy over MS-tree is more challenging that will be discussed in Sections 5.3.

Consider a thread T that is going to access (READ/ INSERT/ DELETE) an item L^j. T can successfully obtain the corresponding lock of L^j if and only if the following two conditions hold: (1) the lock request of T is currently at the head of the wait-list of L^j, and (2) the current lock status of L^j is compatible with that of the request, namely, either L^j is free or the lock over L^j and that T applies are both shared locks. Otherwise, thread T will wait until it is woken up by the thread that finishes computation on L^j.

Once T successfully locks item L^j, the corresponding lock request is immediately removed from the wait-list of L^j and T will conduct its computation over L^j. When the computation is finished, thread T will release the lock and then wake up the thread (if any) whose lock request over L^j is currently at the head of the wait-list. Finally, thread T will continue its remaining computations.

Theorem 4. *The global schedule generated by the proposed locking mechanism is streaming consistent.*

Algorithm 3. Parallel Processing Streaming Graphs

Input: Streaming graph G; Query Graph Q
Output: query results at each time point
for *each time point* t_i **do**
  if *there is an incoming edge* σ_i **then**
    if σ_i does not match any edge in query Q **then**
      CONTINUE
    else
      Let Γ be all lock requests for adding edge σ_i
      for *each lock request in* Γ **do**
        /*DISPATCH lock requests*/
        append it to the *end* of the corresponding wait-list;
      CREATE a new **thread** over Ins(σ_i) (Algorithm 1)
  if *there is an expired edge* σ_j **then**
    if σ_j does not match any edge in query Q **then**
      CONTINUE
    else
      Let Γ be all lock requests for adding edge σ_j
      for *each lock request in* Γ **do**
        /*DISPATCH lock requests*/
        append it to the *end* of the corresponding waiting list;
      CREATE a new **thread** for Del(σ_j)

5.3 Concurrent Access over MS-tree

Consider an expansion list $\{L^1, L^2,...,L^k\}$ whose partial matches are stored in MS-tree M. Each partial match of L^i ($1 \leq i \leq k$) exactly corresponds to a distinct node of depth i in M. Thus, locking L^i is equivalent to locking over all nodes of depth i in M. Partial matches are not stored independently in MS-tree, which may cause inconsistency when concurrent accesses occur. For example, consider the MS-tree in Fig. 9. Assuming that a thread T_1 is reading partial matches of $\{\epsilon_6, \epsilon_5\}$, T_1 will backtrack from node $\backslash_1^2$ (i.e., σ_3) to read $\backslash_1^1$ (i.e., σ_1). Since T_1 only locks L_1^2, if another thread T_2 is deleting $\backslash_1^1$ at the same time, T_2 and T_1 will conflict. Therefore, we need to modify the deletion access strategy over the MS-tree to guarantee streaming consistency as follows.

Algorithm 4. Applies/Releases S/X-Lock

Input: An item L^i and the corresponding wait-list $waitlist(L^i)$
Input: Current thread T
Output: T successfully applies/releases S/X-lock over L^i
function (apply_S/X-lock())
  while the lock request of T is not at the head of $waitlist(L^i)$ OR the lock status of L^i is exclusive **do**
    $thread\_wait()$
  apply S/X-lock over L^i
  pop the head of $waitlist(L^i)$
  end function
function (release_S/X-lock())
  release S/X-lock over L^i
  If $waitlist(L^i)$ is not empty, wake up the thread whose lock request is at the head of $waitlist(L^i)$
  end function

Consider two threads T_1 and T_2 that are launched at time t_1 and time t_2 ($t_1 < t_2$), respectively. Assuming that T_1 is currently accessing partial matches of L^{d_1} in M while T_2 is accessing partial matches of L^{d_2}, let's discuss when inconsistency can happen. There are three types of accesses that each T_i can perform and there are three cases for node depths d_1 and d_2 ($d_1 < d_2, d_1 = d_2$ and $d_1 > d_2$). Thus, there are total $3 \times 3 \times 3 = 27$ different cases to consider, but the following theorem tells us that only two of these cases will cause inconsistency in concurrent execution.

Theorem 5. *Concurrent executions of T_1 and T_2 will violate streaming consistency if and only if one of these two cases occur:*

1) *$d_1 > d_2$, T_1 reads partial matches of L^{d_1} and T_2 deletes partial matches of L^{d_2}. When T_1 wants to read some node $\backslash$ during the backtrack to find the corresponding whole path, T_2 has already deleted $\backslash$, which causes the inconsistency.*
2) *$d_1 > d_2$, T_1 inserts partial match $g = \{\sigma_1, \sigma_2,...,\sigma_{d_1}\}$ of L^{d_1} and T_2 deletes partial matches of L^{d_2}. When T_1 wants to add σ_{d_1} as a child of σ_{d_1-1}, T_2 has deleted σ_{d_1-1}, which causes the inconsistency.*

Theorem 5 shows that inconsistency is always due to a thread T_2 deleting expired nodes that a previous thread T_1 wants to access without applying locks. However, if we make T_2 wait until previous thread T_1 finishes its execution, the degree of parallelism will certainly decrease. In fact, to avoid inconsistency, we only need to make sure that the

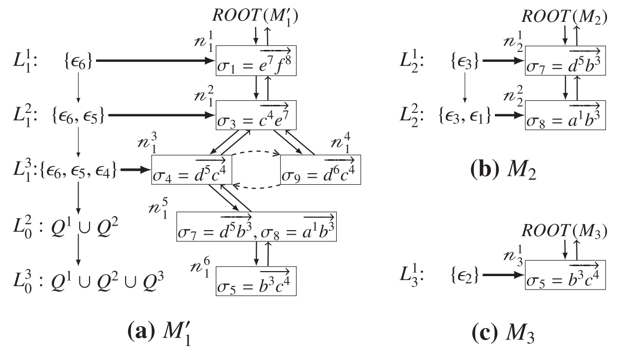


Fig. 12. MS-tree M_1' formed by merging M_0 into M_1.

expired nodes that T_2 wants to delete are invisible to threads launched later than T_2 while accessible to threads that are launched earlier. We achieve this by slightly modifying the deletion strategy over MS-tree with only negligible extra time cost. Specifically, consider the thread T_2 that deletes partial matches of L^{d_2}, when T_2 is going to delete expired node $\backslash_{d_2}$ of depth d_2 in M, T_2 does not "totally" remove $\backslash_{d_2}$ from M. Instead, T_2 "partially" removes $\backslash_{d_2}$ as follows: (1) T_2 removes $\backslash_{d_2}$ from the corresponding doubly linked list, and (2) T_2 disables the link from $\backslash_{d_2}$'s parent to $\backslash_{d_2}$ while the link from $\backslash_{d_2}$ to its parent remains.

Theorem 6. *Parallel accesses with new deletion strategy over MS-tree do not result in streaming inconsistency.*

Our scheduling strategy over the MS-tree is different from the traditional tree protocol [20]. The classical tree protocol only guarantees the conflict equivalence to *some* serial schedule, and there is no guarantee for *streaming consistency* that requires a special serial order.

6 Decomposition

We propose a *cost model*-guided TC decomposition of Q based on the intuition that an incoming edge σ should lead to as few join operations as possible. Cost of join operations varies in stream scenario and we focus on the expected number of join operations to handle an incoming edge. Finding the most appropriate cost function is a major research issue in itself and outside the scope of this paper. Due to space limitation, the discussion on the cost model and why we prefer to a TC decomposition of size as small as possible are presented in Appendix E, available in the online supplemental material; we focus on how to compute the target TC decomposition.

6.1 Decomposition Method

Given a query Q, to find a TC decomposition of size as small as possible, we first extract all possible TC-subqueries of Q, denoted as $TCsub(Q)$. For a TC-subquery Q^i of timing sequence $\{\epsilon_1,\ldots,\epsilon_k\}$, any *prefix* of the timing sequence constitutes a TC-subquery of Q^j. Thus, we can compute $TCsub(Q)$ by the following strategy: (1) We initialize $TCsub(Q)$ with all single edges of Q since each single edge of Q is certainly a TC-subquery of Q; (2) With all TC-subqueries of j edges, we can compute all TC-subqueries of $j+1$ edges as follows: for each TC-subquery $Q^i = \{\epsilon_1,\ldots,\epsilon_j\}$ with j edges, we find all edges ϵ_x such that $\epsilon_j \prec \epsilon_x$. If ϵ_x have common vertex with some $\epsilon_{j'}$ ($j' \in [1, j]$), then we add $\{\epsilon_1,\ldots,\epsilon_j, \epsilon_x\}$ into $TCsub(Q)$ as a new TC-subquery of $j+1$ edges; (3) Repeat Step (2) until there are no new TC-subqueries.

After computing $TCsub(Q)$, we compute a subset D of $TCsub(Q)$ as a TC decomposition of Q, where the subset cardinality $|D|$ should be as small as possible. We use a greedy algorithm to retrieve the desired TC-subqueries from $TCsub(Q)$. We always choose the TC-subquery of maximum size from the remaining ones in $TCsub(Q)$ and there should be no common edges between the newly chosen subquery and those previously chosen ones.

Given a decomposition $D = \{Q^1, Q^2, \ldots Q^k\}$ of query Q, we need to determine a prefix-connected sequence over D, which is in essence to select a join order. We provide a solution for this in Appendix D, available in the online supplemental material due to space limitations.

7 Match-Store DAG

In this section, we propose Match-Store DAG (MS-DAG) to further reduce the space cost for storing partial matches. To illustrate our optimization more explicitly, we focus on how to transfer those MS-trees into a much more condensed MS-DAG.

Consider a query Q and a TC Decomposition $D = \{Q^1, Q^2, \ldots, Q^k\}$. Let $L_i = \{L_i^1, L_i^2, \ldots, L_i^{|Q^i|}\}$ $(i > 0)$ denote the expansion list over TC-subquery Q^i and L_0 over $\{Q^1, Q^2, \ldots, Q^k\}$. Also, assume that M_i $(0 \leq i \leq k)$ is the MS-tree for storing partial matches in L_i. We transfer the MS-trees into a MS-DAG based on two important observations. We first illustrate these two observations, based on which we will further discuss how we conduct the transfer. Then we present the adjustment of the corresponding algorithms and illustrate that there is no drop on time efficiency after transferring MS-trees into MS-DAG.

7.1 Merge M_0 into M_1

7.1.1 Intuition

Observation 1. There is a one-to-one mapping between nodes of depth $|Q^1|$ in M_1 and that of depth 1 (first level) in M_0.

In fact, according to our insertion method, once a new match g of Q^1 is found, a new leaf node will be added in M_1 and in the meantime, we need to insert a new node (of depth 1) in M_0 corresponding to g. For example, node $\backslash_1^3$ in Fig. 9 corresponds to node $\backslash_0^1$ in Fig. 10 while node $\backslash_1^4$ corresponds to node $\backslash_0^2$.

7.1.2 Transfer

Based on Observation 1, we can directly replace each node of depth 1 in M_0 with the corresponding leaf node in M_1. In other words, we can merge M_0 into M_1. We use M_1' to denote the new M_1. In the running example, the M_1' is presented in Fig. 12a and we can see that we merge M_0 into M_1 by replacing $\backslash_0^1$ with $\backslash_1^3$ while $\backslash_0^2$ with $\backslash_1^4$.

7.1.3 Algorithm Adjustment

Since M_0 is merged into M_1, we only need to consider how to access (read/insert/delete) partial matches that were previously stored in M_0 over M_1'.

Theorem 7. *Consider a node $\backslash$ of depth d in M_0 and the corresponding partial match g (i.e., the path from root to $\backslash$ constitute g). After merging M_0 into M_1, the partial match formed by the*

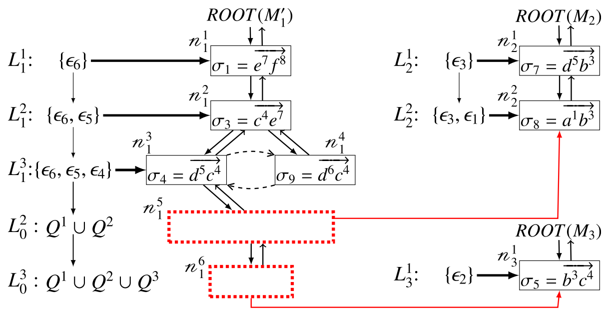


Fig. 13. MS-DAG and the corresponding virtual nodes.

path from the root of M_1' to $\backslash$ is exactly g, and the depth of $\backslash$ in M_1' is $|Q^1| + d - 1$.

For example, $\backslash_1^5$ in M_1' (Fig. 12a) is exactly the node $\backslash_0^3$ in M_0 (Fig. 10). Depth of $\backslash_1^5$ is $3 + 1 = 4$ and the path $\{root \rightarrow \backslash_1^1 \rightarrow \backslash_1^2 \rightarrow \backslash_1^3 \rightarrow \backslash_1^5\}$ exactly constitutes $\{\sigma_1, \sigma_3, \sigma_4, \sigma_7, \sigma_8\}$ as a match of $Q^1 \cup Q^2$. Thus, to read a partial match g that was previously stored in M_0 as node $\backslash$, we can just still backtrack from $\backslash$ in M_1' to get the whole path from the root to $\backslash$: $\{root \rightarrow \backslash_1 \ldots \rightarrow \backslash_x \rightarrow \backslash\}$. While, to insert g, according to our insertion, $\backslash_x$ must be already in M_1' and we need just to add $\backslash$ as a child of $\backslash_x$. Also, with Theorem 7, we can see that deleting $\backslash$ (when expired) can still be finished by removing the subtree rooted in $\backslash$. Hence, the way to access nodes of depth not less than $|Q^1|$ in M_1' is almost the same as that over M_0.

7.2 Virtual Node

7.2.1 Intuition

Observation 2. Consider a node $\backslash$ of depth $|Q^1| + d$ ($d > 0$) in M_1' and the corresponding partial match $g = \{g_1, g_2, \cdots g_{d+1}\}$ for subquery $Q^1 \cup \cdots \cup Q^{d+1}$ where g_i matches Q^i ($1 \leq i \leq d+1$). Then g_{d+1} (the partial matches stored in $\backslash$) can be constituted by the branch from root of M_{d+1} to some leaf node (of depth $|Q^{d+1}|$).

For example, in Fig. 12a, partial match $\{\sigma_7, \sigma_8\}$ in node $\backslash_1^5$ (of depth 4) exactly corresponds to the branch from root to node $\backslash_2^2$ of M_2 in Fig. 12b.

7.2.2 Transfer

Thus, for each node $\backslash$ of M_1' whose depth is $|Q^1| + d$, we can just replace the partial match stored in node $\backslash$ with a virtual node pointing to the corresponding leaf node in M_{d+1}. For example, in Fig. 13, we use red dotted square to denote all virtual nodes. We can find that all MS-trees are merged into a MS-DAG.

7.2.3 Algorithm Adjustment

We only need to consider the access of some new partial match g for subquery $Q^1 \cup \cdots \cup Q^{d+1}$ ($d > 0$). According to Observation 2, there is a branch $\{root \rightarrow \backslash_1 \rightarrow \backslash_2 \ldots \rightarrow \backslash_{|Q^{d+1}|}\}$ from root of M_{d+1} to leaf node $\backslash_{|Q^{d+1}|}$. We can hence create a virtual node $\backslash_x$ in L_0^{d+1} (of depth $d+1$ in M_1') and mark a link from $\backslash_x$ to $\backslash_{|Q^{d+1}|}$. When we access $\backslash_x$, we can easily backtrack from node $\backslash_{|Q^{d+1}|}$ in M_{d+1} to get partial match g using the link. For example, in Fig. 13, for partial match $\{\sigma_7, \sigma_8\}$ (of $Q^1 \cup Q^2$), the corresponding branch in M_2 is $\{\backslash_2^1 \rightarrow \backslash_n^2\}$. We can see that there is a virtual node $\backslash_1^5$ with link to $\backslash_2^2$. When we access $\backslash_1^5$, we can follow the link and backtrack from node $\backslash_2^2$ to get partial match $\{\sigma_7, \sigma_8\}$. Apparently, we need no adjustment for deletion operation since we can just remove those related virtual nodes.

7.3 Analysis

We discuss the space improvement and time efficiency of MS-DAG, as well as some possible issues that need to be addressed for concurrent computation over it.

For space reduction, the improvement lies in reducing space cost of partial matches in L_0 (M_0). Assume that the number of partial matches in L_0^i ($1 \leq i \leq k$) is λ_i. Previous space cost for M_0 is $\Sigma_{j=1}^{k}(\lambda_j * O(|Q^j|))$. While, in MS-DAG, the space cost for storing partial matches of L_0 is $\Sigma_{j=1}^{k}(\lambda_j * O(1))$, where $O(1)$ denotes a constant cost for pointers of virtual nodes. We can see that the space cost is significantly reduced.

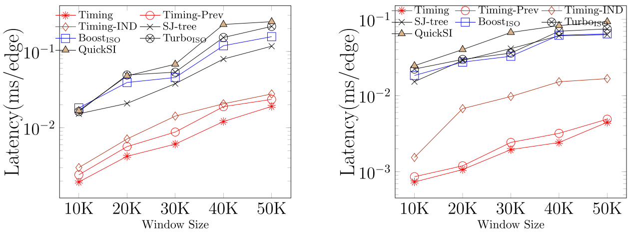
There is no drop in the time efficiency for computation (read, insertion and deletion) over MS-DAG compared that over MS-trees. It is obvious that complexity of insertion and deletion are the same as that over MS-trees according to our method. The difference lies in reading partial matches in L_0 (i.e., nodes whose depth is larger than $|Q^1|$ in M_1'). Consider a partial match g in L_0^i, when reading g over MS-tree M_0, we need just directly access the corresponding node to read the entire g, which costs $O(|g|)$ (i.e.,$O(OUTPUT)$) time. For g over MS-DAG, the corresponding (virtual) node only contains a link to a leaf node in M_i, from which we backtrack to get the entire g. In fact, the backtracking also costs only $O(|g|)$ time. For example, over MS-DAG, to access the corresponding partial match of $\backslash_1^5$ (i.e. $\{\sigma_1, \sigma_3, \sigma_4, \sigma_7, \sigma_8\}$), we need to backtrack from node $\backslash_2^2$ in M_2 (Fig. 13) to get $\{\sigma_7, \sigma_8\}$, and then further backtrack in M_1' from $\backslash_1^5$ to $\backslash_1^3$ (for σ_4), $\backslash_1^2$ (for σ_3) and $\backslash_1^1$ (for σ_1). While in M_0 (Fig. 10), we just need to backtrack from $\backslash_0^3$ to $\backslash_0^1$ to get the partial match. The time cost of both are linear in the size of partial matches and hence no improvement in time happens.

7.4 Concurrent Consistency Guaranteed

We need only a trivial adjustment to what we present in Section 5 for concurrent computation over MS-DAG. Since insertion and deletion for partial matches stored in MS-DAG are the same as that in MS-trees, we focus on the reading partial matches and the corresponding possible concurrent computing issues. According to Theorem 5, we can see that during the backtrack for reading partial matches, the inconsistency could be caused by reading a node that has already been removed and we design a new deletion method to avoid inconsistency of this kind. Thus, for a node $\backslash$ in MS-DAG, no inconsistency would happen if the backtrack is only over $\backslash$'s precedents. However, the backtracking from virtual nodes in MS-DAG need further backtrack from some leaf node of some M_i. In fact, it is easy to see that the further backtracking is just symmetrical to that over precedents with regard to consistency. Since the backtrack over precedents will not cause inconsistency, the reading partial matches in MS-DAG will neither cause any inconsistency.

8 Experimental Evaluation

We evaluate our solution against comparable approaches. All methods are implemented in C++ and run on a CentOS



(a) Network Flow (b) Social Stream

Fig. 14. Latency over different window size.

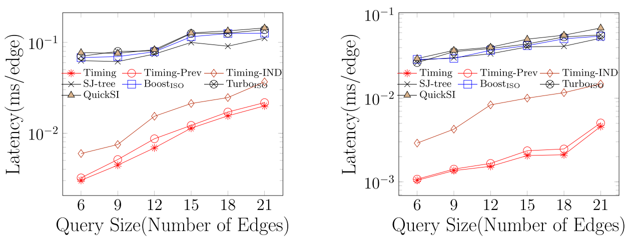
machine of 128G memory and two Intel(R) Xeon(R) E5-2640 2.6 GHz CPU. Codes are available at [21].

We use three datasets in our experiments: real-world network traffic dataset, wiki-talk network dataset and synthetic social stream benchmark. Due to space limits, the experimental results over wiki-talk are presented in Appendix C.1, available in the online supplemental material. The anonymous *network traffic data* contains about 500 millions communication records (edges) concerning about 2 million IP addresses (vertices). *Linked Stream Benchmark* [22] is a synthetic streaming social graph data on user's traces and posts information. This dataset contains 209,549,677 edges and 37,231,144 vertices.

We generate 300 queries of different query sizes and timing order for each dataset in our experiments. More detail on query generation are available in Appendix F, available in the online supplemental material. There are 5 different window sizes in our experiments: $10K$, $20K$, $30K$, $40K$ and $50K$ where each unit of the window size is the average time span between two consecutive arrivals of data edges in the dataset.

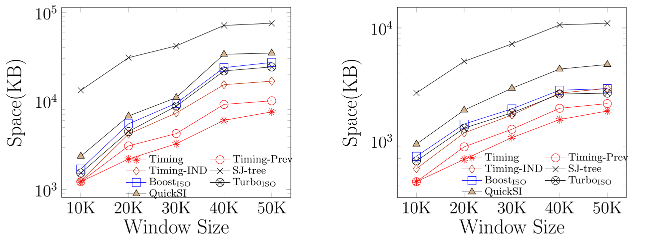
8.1 Comparative Evaluation

Since none of the existing works support concurrent execution, all codes (including ours) are run as a single thread; the evaluation of concurrency management is in Section 8.2. Our method, denoted as Timing, is compared with a number of related works. SJ-tree [1] is the closest work to ours. Since it does not handle the timing order constraints, we verify answers from SJ-tree posteriorly with the timing order constraints. IncMat [12] conducts static subgraph isomorphism algorithm when update happens over streaming graph. We apply three different state-of-the-art static subgraph isomorphism algorithms to IncMat, including QuickSI [7], Turbo$_{ISO}$ [8], Boost$_{ISO}$ [9]. These methods are conducted over the affected area (see [12]) window by window. To evaluate the effectiveness of MS-DAG, we also compare our approach with a counterpart without MS-DAG (called Timing-IND)



(a) Network Flow (b) Social Stream

Fig. 15. Latency over different query size.



(a) Network Flow (b) Social Stream

Fig. 16. Space over different window size.

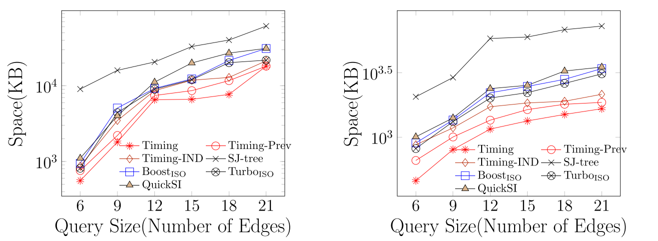
where every partial match is stored independently. We also compare our algorithm with previous version [23]. Due to space limits, we evaluate our TC Decomposition strategy in Appendix C.2, available in the online supplemental material. Note that the reported latency is the average time to handle an edge, i.e., edge insertion/deletion for updating answers.

8.1.1 Time Efficiency Comparison

Figs. 14 and 15 show that our method is clearly faster than other approaches over different window sizes and query sizes, respectively. The reason for the superior performance of our method lies in two aspects. First, our method can filter out lots of discardable partial matches based on the timing order constraint. Second is the efficiency of MS-DAG maintenance algorithms. For example, the deletion algorithm is linear to the total number of expired partial matches; while in SJ-tree, all partial matches need to be enumerated to find the expired ones. SJ-tree needs to maintain lots of discardable partial matches that can be filtered out by our approach. Furthermore, SJ-tree needs post-processing for the timing order constraint, which also increases running time. Also, Since Timing-IND does not use MS-DAG to optimize the space and maintenance cost, it is not as good as Timing. We can see that the time efficiency of Timing-Prev and Timing is the same.

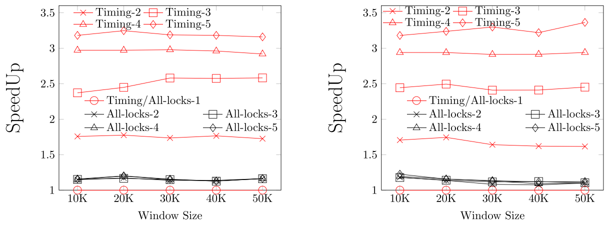
8.1.2 Space Efficiency Comparison

We compare the systems with respect to their space costs. Since the streaming data in the time window changes dynamically, we use the average space cost in each time window as the metric of comparison, as shown in Figs. 16 and 17. We can see that both Timing-IND and Timing have much lower space cost than comparative approaches. Our method is more efficient on space than SJ-tree because SJ-tree does not reduce the discardable partial matches, which wastes space. Our method only maintains partial matches without graph structure in the time window. However, QuickSI, Turbo$_{ISO}$ and Boost$_{ISO}$ need



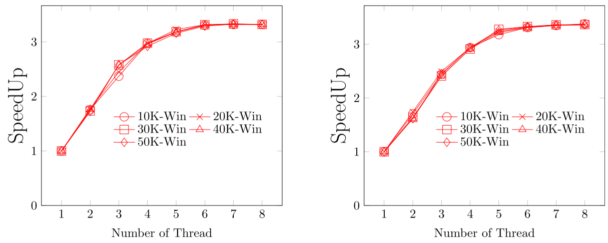
(a) Network Flow (b) Social Stream

Fig. 17. Space over different query size.



(a) Network Flow (b) Social Stream

Fig. 18. Speedup over different Window size.



(a) Network Flow (b) Social Stream

Fig. 20. Speedup varying number of threads.

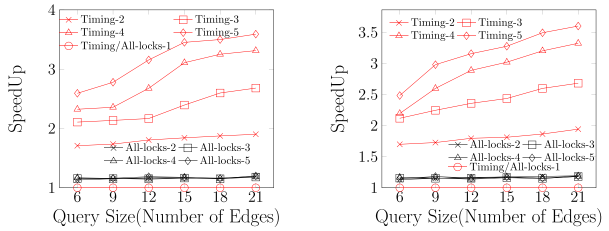
to maintain the graph structure (adjacent list) in each window to conduct search. Also, these comparative methods can not reduce discardable edges that will never exist in any partial match, which results in wasting space. Finally, we can see that our method outperform previous version (Timing-Prev) because of the proposed optimizations.

8.2 Concurrency Evaluation

We evaluate our concurrency technique in this section by varying the number of threads running in parallel. We use *Timing-N* to differentiate different settings of parallel threads (N). We also implement, for comparison, a locking mechanism that requires a thread to obtain all locks before it is allowed to proceed (called *All-locks-N*). We present the speedup over single thread execution in Figs. 18 and 19. We can see that our locking strategy outperforms All-locks-N. As the number of threads grows, the speedup of our locking mechanism improves, while the speedup of All-locks-N remains almost the same. Fig. 19 also shows that speedup of our solution improves as the query size gets larger. In fact, the larger the query size, the more items tend to be in the corresponding expansion lists, which further reduces the possibility of contention. Fig. 20 presents speedup of our solution over different number of threads for each window size. Speedup over different window size show little difference. Also, for each curve formed by increasing number of threads, speedup grows significantly when there are less than 6 threads, while, once the thread number is more than 6, speedup grows much slower. Locking mechanism over finer-grained data unit (single partial match, for example) would be an interesting future work.

9 Conclusion

The proliferation of high throughput, dynamic graph-structured data raises challenges for traditional graph data management techniques. This work studies subgraph isomorphism issues with the timing order constraint over high-speed streaming graphs. We propose an expansion list to efficiently answer subgraph search and propose MS-tree to greatly reduce the space cost. More importantly, we design effectively concurrency management in our computation to improve system's throughput. To the best of our knowledge, this is the first work that studies concurrency management on subgraph matching over streaming graphs. Finally, we evaluate our solution on both real and synthetic benchmark datasets. Extensive experimental results confirm the superiority of our approach compared with the state-of-the-arts subgraph match algorithms on streaming graphs.



(a) Network Flow (b) Social Stream

Fig. 19. Speedup over different query size.

Acknowledgments

This work was supported by The National Key Research and Development Program of China under grant 2018YFB1003504 and NSFC under grant 61932001, 61961130390, U20A20174. This work was also supported by Beijing Academy of Artificial Intelligence (BAAI). M.T. Özsu's work was supported by Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

[1] S. Choudhury, L. B. Holder, G. C. Jr, K. Agarwal, and J. Feo, "A selectivity based approach to continuous pattern detection in streaming graphs," in *Proc. 18th Int. Conf. Extending Database Technol.*, 2015, pp. 157–168.
[2] A. Pacaci, A. Bonifati, and M. T. Özsu, "Regular path query evaluation on streaming graphs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 1415–1430.
[3] Verizon, 2016. [Online]. Available: http://www.verizonenterprise.com/resources/reports/rp_DBIR_2016_Report_en_xg.pdf
[4] X. Qiu *et al.*, "Real-time constrained cycle detection in large dynamic graphs," *Proc. VLDB Endow.*, vol. 11, no. 12, 2018, pp. 1876–1888.
[5] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, 1976.
[6] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 10, pp. 1367–1372, Oct. 2004.
[7] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: An efficient algorithm for testing subgraph isomorphism," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 364–375, 2008.
[8] W.-S. Han, J. Lee, and J.-H. Lee, "Turbo ISO: Towards ultrafast and robust subgraph isomorphism search in large graph databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 337–348.
[9] X. Ren and J. Wang, "Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs," *Proc. VLDB Endowment*, vol. 8, no. 5, pp. 617–628, 2015.
[10] X. Wang *et al.*, "Efficient subgraph matching on large RDF graphs using MapReduce," *Data Sci. Eng.*, vol. 4, no. 1, pp. 24–43, 2019.
[11] A. Morari *et al.*, "Scaling semantic graph databases in size and performance," *IEEE Micro*, vol. 34, no. 4, pp. 16–26, Jul./Aug. 2014.
[12] W. Fan, X. Wang, and Y. Wu, "Incremental graph pattern matching," *ACM Trans. Database Syst.*, vol. 38, no. 3, 2013, Art. no. 18.
[13] L. Chen and C. Wang, "Continuous subgraph pattern search over certain and uncertain graph streams," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 8, pp. 1093–1109, Aug. 2010.

[14] J. Gao, C. Zhou, J. Zhou, and J. X. Yu, "Continuous pattern detection over billion-edge graph using distributed framework," in *Proc. 30th IEEE Int. Conf. Data Eng.*, 2014, pp. 556–567.

[15] P. Mackey, K. Porterfield, E. Fitzhenry, S. Choudhury, and G. Chin, "A chronological edge-driven approach to temporal subgraph isomorphism," in *Proc. IEEE Int. Conf. Big Data*, 2018, pp. 3972–3979.

[16] C. Song, T. Ge, C. X. Chen, and J. Wang, "Event pattern matching over graph streams," *Proc. VLDB Endowment*, vol. 8, no. 4, pp. 413–424, 2014. [Online]. Available: http://www.vldb.org/pvldb/vol8/p413-ge.pdf

[17] S. Choudhury, G. Chin, K. Agarwal, and S. J. Beus, "Performance and usability enhancements for continuous subgraph matching queries on graph-structured data," US Patent App. 15/594,376, Nov. 15, 2018.

[18] E. Wu, Y. Diao, and S. Rizvi, "High-performance complex event processing over streams," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2006, pp. 407–418.

[19] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar, "Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows," *Proc. VLDB Endow.*, vol. 11, no. 6, 2018, pp. 691–704.

[20] A. Silberschatz and Z. Kedem, "Consistency in hierarchical database systems," *J. ACM*, vol. 27, no. 1, pp. 72–80, 1980.

[21] "Code," 2018. [Online]. Available: https://github.com/pkumod/timingsubg.git

[22] "Lsbench code," 2012. [Online]. Available: https://code.google.com/archive/p/lsbench/

[23] Y. Li, L. Zou, M. T. Özsu, and D. Zhao, "Time constrained continuous subgraph search over streaming graphs," in *Proc. 35th Int. Conf. Data Eng.*, 2019, pp. 1082–1093.

Youhuan Li received the BS and PhD degrees from Peking University, in 2013 and 2018, respectively. He worked as a postdoc with Peking University and Tencent, (2018–2020). Starting March 2021, he is an associate professor with Hunan University, focusing on graph data management.

Lei Zou is a professor with the Institute of Computer Science and Technology, Peking University. He is also a faculty member in Big Data Center of Peking University and Beijing Institute of Big Data Research. His research interests include graph database and semantic data management.

M. Tamer Özsu (Fellow, IEEE) is a University professor in David R. Cheriton School of Computer Science, University of Waterloo. His current research interests include focuses on large scale data distribution and management of unconventional data (e.g., graphs, RDF, and streams). He is a fellow of the ACM , and a member of Sigma Xi.

Dongyan Zhao received the BS, MS, and PhD degrees from Peking University, in 1991, 1994 and 2000, respectively. Now, he is a professor with the Institute of Computer Science and Technology of Peking University. His research interests include information processing and knowledge management.