

**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. We propose a solution to efficiently 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 confirms the efficiency and effectiveness of our solution.

I. 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 such streaming graph data is of great significance for tasks such as detecting anomalous events (e.g., in Twitter) and detecting adversarial activities in computer networks. Various types of queries over streaming graphs have been investigated, such as subgraph search, path computation, and triangle counting [1]. Among these, 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.**

Figure [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 \(t_5\) 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 from the past decade that reveal that 90% of the incidents fall into ten attack patterns [2], which can be described as graph patterns.

![Query example in Network Traffic](image1)

**Example 2. Credit card fraud in transactions.**

Figure [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 accounts forming transfer path). Apparently, this pattern where \(t_1 < t_2 < t_3 < t_4\) can be easily modeled as a query graph with timing order constraints.

![Credit card fraud transactions](image2)
A. Related Work

Although subgraph search has been extensively studied in literature [4]–[10], most of these works focus on static graphs. Ullman [4] proposes a well-known subgraph isomorphism algorithm that is based on a state-space search approach; Cordella et al. [5] propose the VF2 algorithm that employs several important pruning strategies when searching for targeted subgraphs. Shang et al. [6] 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. [7] 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 [8] define four vertex relationships over a query graph to reduce duplicate computation.

The research on continuous query processing over high-speed streaming graph data is rather scarce. Fan et al. [11] 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. [12] 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. Also, graph stream in [12] is a sequence of small data graphs, which is not our focus. Gao et al. [13] study continuous subgraph search over a graph stream. They make specific assumption over their query and their solution cannot guarantee exact answers for subgraph isomorphism. Song et al. [14] 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.

To the best of our knowledge, this is the first work that investigates subgraph (isomorphism) matching over streaming graphs that take into account both structural and edge timing constraints. Table I summarizes the differences between our work with the ones discussed above.

Due to the high speed of streaming graph data and the system’s high-throughput requirement, a concurrent computing (i.e., multi-threaded) algorithm is desirable or 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.

### Table I: Related work VS. Our Method

| Method          | Subgraph Isomorphism | Timing Order | Exact Solution |
|-----------------|----------------------|--------------|----------------|
| Our Method      | ✓                    | ✓            | ✓              |
| Choudhury et al.| ✓                    | x            | ✓              |
| Song et al.     | ✓                    | ✓            | ✓              |
| Gao et al.      | ✓                    | ✓            | ✓              |
| Chen et al.     | ✓                    | ✓            | ✓              |
| Fan et al.      | ✓                    | ✓            | ✓              |

B. 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 designing a Trie-like data structure (called match-store tree) and (3) proposing a concurrent computing 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 5 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 computing framework (Section III).

**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 IV).

**Improving system throughput.** Existing works do not consider concurrent execution of continuous queries over streaming graphs. For a high-speed graph stream, some edges may come at the same time. A naive solution is to process each edge one-by-one. 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 turns 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 V).

II. PROBLEM DEFINITION

We list frequently-used notations in Table II.

**Definition 1 (Streaming Graph):** A streaming graph $G$ is a constantly growing sequence of directed edges
{\sigma_1, \sigma_2, \ldots, \sigma_z} \) in which each \( \sigma_i \) arrives at a particular time \( t_i \) (\( t_i < t_j \) when \( 0 < i < j \)). This time point is also referred to as the timestamp of \( \sigma_i \). Each edge \( \sigma_i \) 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. For example, since vertex labels and edge labels are from two different label sets, we can introduce an imaginary vertex to represent an edge of interest and assign the edge label to the new imaginary vertex.

An example of a streaming graph \( \mathcal{G} \) is shown in Figure 4. Note that edge \( \sigma_5 \) has two endpoints \( e^7 \) and \( f^8 \), where \( 'e' \) and \( 'f' \) are vertex labels and the superscripts are vertex IDs that we introduce to distinguish two vertices with the same label.

In this paper, we use the time-based sliding window model, where a sliding window \( W \) defines a timespan with fixed duration \(|W|\). If the current time is \( t_i \), the time window \( W \) defines the timespan \((t_i - |W|, t_i]\). Obviously, all edges that occur in this time window form a consecutive block over the edge sequence and as time window \( W \) slides, some edges may expire and some new edges may arrive.

**Definition 2 (A Snapshot of a Streaming 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 \( \sigma_i \) matches \( \epsilon_i \) and \( \sigma_j \) matches \( \epsilon_j \) \((\sigma_i, \sigma_j \in g)\), \( \epsilon_i \) should be less than that of \( \sigma_j \).

An example of query graph \( Q \) is presented in Figure 4. Any subgraph in the result must conform to the constraints on both structure and timing orders. For example, in query \( Q, \epsilon_1 \prec \epsilon_2 \) \((\epsilon_1, \epsilon_2 \in E(Q))\) means that edges matching \( \epsilon_1 \) should arrive before edges matching \( \epsilon_2 \) in subgraph matches of \( Q \) over the snapshot (see Definition 3) in the current time window.

**Definition 4 (Time-Constrained Match):** For a query \( Q \) and a subgraph \( g \) in current snapshot, \( g \) is a time-constrained match of \( Q \) if only if there exists a bijective function \( F \) from \( V(Q) \) to \( V(g) \) such that the following conditions hold:

1. **Structure Constraint (Isomorphism)**
   - \( \forall u \in V(Q), L(u) = L(F(u)) \)
   - \( \overrightarrow{uv} \in E(Q) \iff \overrightarrow{F(u)F(v)} \in E(g) \)

2. **Timing Order Constraint**
   - For any two edges \( (u^1, u^2), (u'^1, u'^2) \in E(Q) \):
     \[ (u^1, u^2) \prec (u'^1, u'^2) \implies F(u^1)F(u^2) \prec F(u'^1)F(u'^2) \]

Hence, the problem in this paper is to find all time-constrained matches of given query \( Q \) over each snapshot of graph stream \( \mathcal{G} \) 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 \( \sigma_1, \sigma_3, \sigma_4, \sigma_5, \sigma_7 \) and \( \sigma_8 \) in Figure 4 (highlighted by bold line) is not only isomorphic to query \( Q \) but also conforms to the timing...
order constraints defined in Figure 5(b). Thus, $g$ is a match of query $Q$ over stream $G$ at time point $t = 8$. At time point $t = 10$, with the deletion of edge $\sigma_1$, $g$ expires.

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

**Proof:** Consider an arbitrary subgraph isomorphism problem: given two graphs $G$ and $g$, answering whether there is a subgraph $g'$ of $G$ that is isomorphic to $g$. Let’s reduce this problem into a time constrained continuous subgraph isomorphism over streaming graph. Assuming that there are $m$ edges in $G$: $\{\sigma_1, \sigma_2, ..., \sigma_m\}$, we transform $G$ into a streaming graph $\mathcal{G}$ by randomly assigning timestamp $t_i$ to each edge $\sigma_i$ such that $0 < t_i < t_j$ if $i < j$. We set a time window $W$ of size $(t_m - t_1)$ and initial timespan $(0, t_1)$. Let $g$ be a continuous query graph of timing order $\prec$. In this way, let’s apply our solution over the streaming graph $\mathcal{G}$ of time window $W$ with query $g$, if there are answers for $g$ at time $t_m$, then there must be subgraph $g'$ of $G$ that is isomorphic to $g$. Apparently, it costs only polynomial time to finish reducing a subgraph isomorphism problem to the proposed one. Hence, the proposed problem is NP-hard.

III. A Baseline Method

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

A. Timing-Connected Query

1) Intuition: A naive solution to executing a query $Q$ with timing order is to run a classical subgraph isomorphism algorithm (such as QuickSI [6], TurboISO [7], BoostISO [8]) on each snapshot $G_i$ ($i = 1, 2, ..., \infty$) to first check the structure constraint followed by a check of the timing order constraint among the matches. Obviously, this is quite expensive. A better approach is to identify the subgraph $\Delta(G_i)$ of $G_i$ that is affected by the updated edge (insertion/deletion) and then conduct subgraph isomorphism algorithm over $\Delta(G_i)$ instead of the whole snapshot $G_i$. While, if the query diameter is $d$, then $\Delta(G_i)$ is the subgraph induced by all vertices that is $d$-hop reachable to/from the adjacent vertices of the updated edge [11]. Hence, the size of $\Delta(G_i)$ could be huge if query diameter is large which results in the inefficiency of the computation.

However, an incoming/expired edge causes only a minor change between two consecutive snapshots $G_i$ and $G_{i+1}$; 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 $\sigma$ to form new matches of query $Q$ in the new snapshot $G_i$. Similarly, we can delete all (partial) matches containing the expired edges at the new timestamp. For example, consider the query graph $Q$ in Figure 6. Assume that an incoming edge $\sigma$ matches $\epsilon_1$ at time point $t_i$. If we save all partial matches for subquery $Q \setminus \{\epsilon_1\}$, 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., $G_{i-1}$), we only need to join $\sigma$ with these partial matches to find new subgraph matches of query $Q$.

Although materializing partial matches can accelerate continuous subgraph query, it is inevitable to introduce much maintenance overhead. For example, in SJ-tree [1], each new coming edge $\sigma$ 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 streaming graph $\mathcal{G}$ and a query graph $Q$, an incoming edge $\sigma$ is called a discardable edge if $\sigma$ cannot be included in a complete match of $Q$, no matter what edges arrive in the future.

To better understand discardable edge, recall the streaming graph $\mathcal{G}$ in Figure 3. At time $t_6$, an incoming edge $\sigma_6$ (only matching $\epsilon_1$) is added to the current time window. Consider the timing order constraints of query $Q$ in Figure 5 which requires that edges matching $\epsilon_3$ should come before ones matching $\epsilon_1$. However, there is no edge matching $\epsilon_3$ before $t_6$ in $\mathcal{G}$. Therefore, it is impossible to generate a complete match (of $Q$) consisting of edge $\sigma_6$ (matching $\epsilon_1$) no matter which edges come in the future. Thus, $\sigma_6$ is a discardable edge that can be filtered out safely. We design an effective solution to determine if an incoming edge $\sigma$ is discardable. Before presenting our approach, we introduce an important definition.

**Definition 6 (Prerequisite Edge/Prerequisite Subquery):** Given an edge $\epsilon$ in query graph $Q$, a set of prerequisite edges of $\epsilon$ (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 3. The subquery of $Q$ induced by edges in $Preq(\epsilon)$ is called a prerequisite subquery of $\epsilon$ in query $Q$.

![Fig. 6: Example of prerequisite subquery](image)

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

**Lemma 1:** An incoming edge $\sigma$ at time $t_i$ is NOT discardable if and only if, at the current snapshot $G_i$, there exists at least one query edge $\epsilon \in Q$ such that (1) the prerequisite subquery $Preq(\epsilon)$ has at least one match $g$ (subgraph of $G_i$) containing $\sigma$; and (2) $\sigma$ matches $\epsilon$ in the match relation between $g$ and $Preq(\epsilon)$. Otherwise, $\sigma$ is discardable.

**Proof:** If $\sigma$ is not discardable and $g'$ is the match in the future that includes $\sigma$, then there must be a query edge $\epsilon$ that
σ matches. Also, since \( \text{Preq}(\epsilon) \) is a subquery of query \( Q \), we can always find a subgraph \( g \subseteq g' \) such that \( g \) matches \( \text{Preq}(\epsilon) \) and \( \sigma \in g \). On the contrary, if \( \sigma \) matches \( \epsilon \) and there is a subgraph \( g \) where \( \sigma \in g \) and \( g \) matches \( \text{Preq}(\epsilon) \), then all edges that are required to be before \( \sigma \) have been in the time window and it is possible that there will be a series edges in the future that extend \( g \) into a match of \( Q \), indicating that \( \sigma \) is not discardable.

Lemma 1 can be used to verify whether or not an incoming edge \( \sigma \) is discardable. The straightforward way requires checking subgraph isomorphism between \( \text{Preq}(\epsilon) \) and \( G_\sigma \), which is quite expensive. First, \( \text{Preq}(\epsilon) \) may not be connected, even though query \( Q \) is connected. For example, \( \text{Preq}(\epsilon_1) \) in Figure 6 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 incoming edge.

For certain types of queries that we call timing-connected query (Definition 8), it is easy to determine if an edge \( \sigma \) in streaming graph \( G \) 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 III-B.

For ease of presentation, 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 3) 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 \( \Omega(Q_1) \bowtie \Omega(Q_2) \), as follows:

\[
\Omega(Q_1) \bowtie \Omega(Q_2) = \{ g_1 \cup g_2 | g_1 \in \Omega(Q_1) \land g_2 \in \Omega(Q_2) \}
\]

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.

2) TC-query:

Definition 7 (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 8 (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 \( \forall j \in [1, k-1] \), \( \epsilon_j < \epsilon_{j+1} \). In this case, we call the sequence \( \{\epsilon_1, ..., \epsilon_k\} \) the timing sequence of TC-query \( Q \).

Recall the running example \( Q \) in Figure 5, which is not a TC-query. However, the subquery induced by edges \( \{\epsilon_6, \epsilon_5, \epsilon_4\} \) is a TC-query, since \( \epsilon_6 < \epsilon_5 < \epsilon_4 \) 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 \( \text{Preq}(\epsilon_j) \) is exactly the subquery induced by the first \( j \) edges in \( \{\epsilon_1, \epsilon_2, ..., \epsilon_j\} \) \( (j \in [1, k]) \).

Definition 9 (Expansion List): 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 item \( L^i \) corresponds to \( \bigcup_{j=1}^{i} (\epsilon_j) \), i.e., \( \text{Preq}(\epsilon_i) \).
2) Each item \( L^i \) records \( \Omega(\bigcup_{j=1}^{i} (\epsilon_j)) \), i.e., a set of partial matches (in the current snapshot) of prerequisite subquery \( \text{Preq}(\epsilon_i) \) \((i \in [1, k]) \). We also use \( \Omega(L^i) \) to denote the set of partial matches in \( L^i \).

Note that each item \( L^i \) corresponds to a distinct subquery \( \text{Preq}(\epsilon_i) \) and we may use the corresponding subquery to denote an item when the context is clear.

The shaded nodes in Figure 7 illustrate the prerequisite subqueries for a TC-query with timing sequence \( \{\epsilon_1, \epsilon_2, \epsilon_4\} \). Since each node corresponds to a subquery \( \text{Preq}(\epsilon_i) \), we also record the matches of \( \text{Preq}(\epsilon_i) \), as shown in Figure 7. 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 \( \sigma \), Theorem 2 tells us which (partial) matches associated with the expansion list 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 $\sigma$ matches query edge $\epsilon_i$ in the current time window, then only the (partial) matches of $L_i$ ($Preq(\epsilon_i)$) should be updated in the current snapshot.

1. If $i = 1$, $\sigma$ should be inserted into $L^1$ as a new match of $Preq(\epsilon_1)$ since $Preq(\epsilon_1) = \{\epsilon_1\}$.

2. If $i \neq 1 \land \Omega(L_i^{-1}) \sim \sigma \neq \emptyset$, then $\Omega(L_i^{-1}) \sim \sigma$ should be inserted into new matches of $Preq(\epsilon_i)$, where $\Omega(L_i^{-1})$ is the set of partial matches in $L_i^{-1}$.

Proof: Assume edge $\sigma$ matches edge $\epsilon_i$, which means there will be no partial matches in $L_i$ containing $\sigma$. However if $i' > i$, assuming that $g = \{\sigma_1, \sigma_2, ..., \sigma_{i'}\}$ is a new match to be inserted into $L_i$ where $\sigma_{i'}$ matches $\epsilon_{i'}$ ($1 \leq i' \leq i$ and $\sigma = \epsilon_i$), then $\sigma$ has smaller timestamp than that of $\sigma_{i'}$ because $\epsilon_i < \epsilon_{i'}$, which is impossible since $\sigma$ is the incoming edge of largest timestamp in the current window. Thus, $\epsilon_i$ must be $i$.

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

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

Theorem 3: Given a TC-query $Q = \{\epsilon_1, \epsilon_2, ..., \epsilon_k\}$ and the corresponding expansion list $L = \{L^1, L^2, ..., L^k\}$, for an incoming edge $\sigma$ that matches $\epsilon_i$, the time to determine whether $\sigma$ is discardable (to be filtered) or not is $O(|L_i^{-1}|)$, which is linear to the number of partial matches in $L_i^{-1}$.

Proof: Obviously, the main cost for determining whether $\sigma$ is discardable (to be filtered) or not lies in the join between $\sigma$ and $\Omega(L_i^{-1})$, which costs $O(|L_i^{-1}|)$ time. $L_i^{-1}$ contains all matches of subquery $\{\epsilon_1, \epsilon_2, ..., \epsilon_{i-1}\}$ and its size range from 0 to $|\epsilon_i| - 1$.

The above process is codified in Lines 11-16 of Algorithm 1, which Note that an incoming edge $\sigma$ may match multiple query edges; the above process is repeated for each matching edge $\epsilon_i$. New matches that are inserted into the last item of the expansion list are exactly the new matches of TC-query $Q$. For example, consider the TC-query $Q = \{\epsilon_6, \epsilon_5, \epsilon_4\}$ in Figure 7a and the streaming graph $G$ in Figure 3. At time $t = 2$, when the edge $\sigma_2$ (matching $\epsilon_5$) arrives, $\Omega(Preq(\epsilon_5)) = \{\sigma_1\} \sim \{\sigma_2\} = \emptyset$, so there is no new match that is inserted into $\Omega(Preq(\epsilon_5))$.

Case 2: Edge expiry. When an edge $\sigma$ expires, we can remove all expired partial matches (containing $\sigma$) in expansion list $L_i$ by scanning $L^1$ to $L^i$ where $L^i$ is the rightmost item in $L$ which contains expired partial matches (Lines 2-6 in Algorithm 2).

B. 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, $Q = \bigcup_{i=1}^k Q^i$ and there is no common query edge between any two TC-subqueries. We call $D$ as a TC-decomposition of $Q$. The example query $Q$ is decomposed into $\{Q^1, Q^2, Q^3\}$, as shown in Figure 9. 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 $Q^i (i = 1, ..., k)$ into matches of $Q$ in the stream scenario.

For the sake of presentation, we assume that the decompos-
tion of query $Q$ is given; decomposition is further discussed in Section VI-B. We use $L_i = \{L_1^i, L_2^i, \ldots, L_{e_i}^i\}$ to denote the corresponding expansion list for each TC-subquery $Q^i$. Recall the definition of prefix-connected sequence (Definition 7). 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 $Q_1^1$ from $D$; and then we extract a second TC-subquery $Q_2^2$ who have common vertex with $Q_1^1$ (Since $Q$ is weakly connected, we can always find such $Q_2^2$); 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 $\{Q_1^1, Q_2^2, \ldots, Q_k^k\}$ is a prefix-connected permutation of $D$ where the subquery induced by $Q_i^i$ for $1 \leq i \leq k$. 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 discuss join order selection in Section VI-C. Until then, we assume that the prefix-connected sequence $D = \{Q_1^1, Q_2^2, \ldots, Q_k^k\}$ is given.

![Fig. 8: A TC decomposition of query $Q$](image)

For example, Figure 9 illustrates a decomposition of query $Q$ ($Q_1^1, Q_2^2, Q_3^3$). We obtain the matches of $Q$ as $\Omega(Q) = \Omega(Q_1^1) \times \Omega(Q_2^2) \times \Omega(Q_3^3)$. Like TC-query, we can also 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 $L_0 = \{L_1^1, L_2^2, L_3^3\}$ for query $Q$ is given in Figure 9. Each item $L_i^j$ records the intermediate join results $\Omega(\bigcup_{x=1}^{n} Q_x)$.

Assume that an incoming edge $\sigma$ contributes to new matches of TC-subquery $Q_i^i$ (denoted as $\Delta(L_i^i)$). If $i > 1$, we let $\Delta(L_0^i) = \Delta(L_i^i) \times \Delta(L_{e_i}^{i-1})$ (Line 16 in Algorithm 1). If $\Delta(L_0^i) = \emptyset$, we insert $\Delta(L_0^i)$ into $L_0^i$ as new matches of $L_0^i$. Then, $\Delta(L_0^i)$ may not be empty and the join results (if any) are new partial matches that should be stored in $L_{e_i}^{i+1} (\bigcup_{x=1}^{n} Q_x)$). Thus, we need to further perform $\Delta(L_0^i) \times \Delta(L_{e_i}^{i+1})$ to get new partial matches (denoted as $\Delta(L_{e_i}^{i+1})$) and insert them into $L_{e_i}^{i+1}$ as new matches of $\bigcup_{x=1}^{n} Q_x$.

We repeat the above process until no new partial matches are created or the new partial matches are exactly answers of the entire query $Q$ (Lines 18-22). Note that when partial matches of different subqueries are joined, we verify both structure and timing order constraints.

When an edge $\sigma$ expires where $\sigma$ matches $\epsilon \in Q_i^i$, we discard all partial matches containing $\sigma$ in expansion list $L_i^i$ as illustrated previously. If there are expired matches for $Q_i^i$ (i.e., matches of $Q_i^i$ that contain $\sigma$), then we also scan $L_0^i$ to $L_{e_i}^{i+1}$ to delete partial matches containing $\sigma$. (Pseudo codes for deletion are presented in Algorithm 2).

IV. 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. Let’s formally define MS-tree to present how the corresponding partial matches are stored and then illustrate how to access partial matches in MS-tree for the computation.

![Fig. 10: MS-tree of expansion list $L_1 = \{L_1^1, L_2^2, L_3^3\}$](image)

A. Match-Store Tree

Consider an expansion list $L = \{L_1^1, L_2^2, \ldots, L_k^k\}$ over timing sequence $\{\epsilon_1, \epsilon_2, \ldots, \epsilon_k\}$ where $L_i^j$ stores all partial matches of $\{\epsilon_1, \epsilon_2, \ldots, \epsilon_i\}$. For a match $g$ of $L_i^j$ ($1 \leq i \leq k$), $g$ can be naturally presented in a sequential form: $\{\sigma_1, \sigma_2, \ldots, \sigma_i\}$ where $g = \bigcup_{i=1}^{n}(\sigma_i)$ and each $\sigma_i$ ($1 \leq i \leq l$) is a match of $\epsilon_i$. Furthermore, $g' = g \setminus \{\sigma_i\} = \{\sigma_1, \sigma_2, \ldots, \sigma_{i-1}\}$, as a match of $\{\epsilon_1, \epsilon_2, \ldots, \epsilon_{i-1}\}$, must be stored in $L_{i-1}^{i-1}$. Recursively, there must be $g'' = g' \setminus \{\sigma_{i-1}\}$ in $L_{i-2}^{i-2}$. For example, see the
expansion list in Figure 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 three partial matches share a prefix sequence. Therefore, we propose a trie variant data structure to store the partial matches.

**Definition 10 (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 \( n \) of depth \( i \) \((1 \leq i \leq k)\) in a MS-tree denotes a match \( \epsilon_i \) and all nodes along the path from the root to node \( n \) together constitute a match \( \{\epsilon_1, \epsilon_2, \ldots, \epsilon_i\} \). Also, for each node \( n \) of a MS-tree, \( n \) 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 Figure 10. 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 \( \{\sigma_1, \sigma_3, \sigma_4\} \). 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 storing all matches by compressing the prefix. Thus, MS-tree greatly reduces the space cost for reading partial matches in \( L \).

**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., \( \{\epsilon_1, \epsilon_2, \ldots, \epsilon_i\} \). We can obtain all matches of \( L^i \) by enumerating all nodes of depth \( i \) in \( M \) with the corresponding doubly linked list, and then for each node of depth \( i \), we can easily backtrack the \( i \)-length paths to get the match 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 \( \{\epsilon_1, \epsilon_2, \ldots, \epsilon_i\} \): \( g = \{\sigma_1, \sigma_2, \ldots, \sigma_i\} \) where each \( \sigma_j \) matches \( \epsilon_j \), we need to insert a path \( \{\text{root} \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \ldots \rightarrow \sigma_i\} \) into MS-tree. According to the insertion over expansion list, \( g \) must be obtained by \( \{\sigma_1, \sigma_2, \ldots, \sigma_{i-1}\} \) \( \bigcirc \{\sigma_i\} \) and there must already be a path \( \{\text{root} \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \ldots \rightarrow \sigma_{i-1}\} \) in MS-tree. Thus, we can add \( \sigma_i \) as a child of node \( \sigma_{i-1} \) 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 \( \{\text{root} \rightarrow \sigma_1 \rightarrow \sigma_3\} \) by adding \( \sigma_9 \) as a child of \( \sigma_3 \) (see Figure 10). Note that, we can easily record node \( \sigma_{i-1} \) when we find that \( \{\sigma_1, \sigma_2, \ldots, \sigma_{i-1}\} \) \( \bigcirc \{\sigma_i\} \) 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 \( \{\text{root} \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \ldots \rightarrow \sigma_{i-1}\} \) as the usual insertion of trie.

**Deleting expired partial matches:** When an edge \( \sigma \) expires, we need to delete all partial matches containing \( \sigma \). Nodes corresponding to expired partial matches in MS-tree are called expired nodes and we need to remove all expired nodes. Assuming that \( \sigma \) matches \( \epsilon_i \), nodes containing \( \sigma \) 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 \( \sigma \)) 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 Figure 10. When edge \( \sigma_1 \) (matching \( \epsilon_6 \) in TC-query \( \{\epsilon_6, \epsilon_5, \epsilon_4\} \)) expires, we delete node \( \sigma_1 \) in the first level of MS-tree, after which we further delete its descendant nodes \( \sigma_3, \sigma_4 \) and \( \sigma_5 \) successively. When an edge expired, the time cost for the deletion update is linear to the number of the corresponding expired partial matches.

**B. MS-Tree Accessibility**

Let’s discuss how to access the partial matches stored in MS-tree. Note that these are basic serial accesses to MS-tree and the access issues in concurrent mode will be discussed in Section V.

**C. MS-tree and Trie**

*From the perspective of data structure:* Each node \( n \) in MS-tree, besides the links to \( n \)’s children, there are extra links to \( n \)’s father and siblings (doubly linked list). These extra links take an important role in reading matches of subqueries and
V. 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 V-A and V-B then illustrate how to apply the locking strategy over MS-tree in Section V-C.

A. Intuition

Consider the example query $Q$ in Figure 5 which is decomposed into three TC-subqueries $Q^1$, $Q^2$ and $Q^3$ (see Figure 9). Figure 9 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 Figure 12) 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 conflicts or wrong results. For convenience, insertion of an incoming edge $\sigma_i$ is denoted as $Ins(\sigma_i)$ while deletion of an expired edge $\sigma_j$ is denoted as $Del(\sigma_j)$.

Figure 12 illustrates the steps of handling each incoming edge based on the discussion in Section III. When $\sigma_{11}$ is inserted (denoted as $Ins(\sigma_{11})$), $\sigma_{11}$ matches query edge $e_6$ and since $e_6$ is the first edge in TC-subquery $Q^1$, we only need to insert match $\{e_1\}$ into $\Omega(e_6)$ 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 $\sigma_{12}$ matches $e_3$ requires one operation: $INSERT(L_2^1)$ (inserting $\{\sigma_{12}\}$ into $\Omega(e_3)$). For $Ins(\sigma_{13})$ where $\sigma_{13}$ matches $e_2$, we first insert $\sigma_{13}$ into $L_3^1$ (inserting $\{\sigma_{13}\}$) as a new match of $Q^3$ (see Figure 9) and then we need to join $\{e_{13}\}$ with $\Omega(Q^1 \cup Q^2)$ (READ($L_3^1$)) and insert join results into $L_0$ (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 $\sigma_{13}$, we access the following expansion list items: $INSERT(L_3^1)$, $READ(L_0^3)$ and $INSERT(L_0^3)$.

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

Let us consider an incoming edge $\sigma_{14}$ that matches $\{e_4\}$, which is the last edge in the timing sequence of TC-subquery $Q^3$. According to Algorithm 1 we need to read $\Omega(\{e_6, e_5\})$ and join $\Omega(\{e_6, e_5\})$ with $\{\sigma_{14}\}$. Since $e_4$ is the last edge in $Q^3$, if $\Omega(\{e_6, e_5\}) \times \{\sigma_{14}\} \neq \emptyset$, the join results are new matches of $Q^1$, and will be inserted into $L_0$. As discussed in Section III-B we need to join these new matches of $Q^1$ with $\Omega(Q^3)$ resulting in new matches of $Q^1 \cup Q^2$, which will be inserted into $L_0$. Finally, new matches of $Q^1 \cup Q^2$ will be further joined with $\Omega(Q^3)$, after which new matches of $Q^1 \cup Q^2$ and $Q^3$ will be inserted into $L_0$. Thus, the series of operations to be conducted for $Ins(\sigma_{14})$ are as follows: $READ(L_0^3)$, $INSERT(L_1^1)$, $READ(L_2^1)$, $INSERT(L_3^1)$, $READ(L_4^1)$, $INSERT(L_0^3)$. Obviously, $Ins(\sigma_{14})$ may conflict with $Ins(\sigma_{13})$ since both of them will conduct $INSERT(L_0^3)$ as indicated in Figure 12. Thus, the concurrent execution requires a locking mechanism to guarantee the consistency.

Definition 11 (Streaming Consistency): Given a streaming graph $G$ 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 $G$. For example, a concurrent execution that executes $Ins(\sigma_{14})$ followed by $Ins(\sigma_{13})$ would be serializable but would violate streaming consistency.

B. 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 $\sigma$ (i.e., $Ins(\sigma)$) and deletion of an expired edge $\sigma'$ (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 V-A, 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, \( \text{Ins}(\sigma_{13}) \) and \( \text{Ins}(\sigma_{14}) \) conflict with each other only at items \( L_1^0, L_2^0 \) and \( L_3^0 \). The first three elementary operations of \( \text{Ins}(\sigma_{13}) \) and \( \text{Ins}(\sigma_{14}) \) can execute concurrently without causing any inconsistency. Therefore, a finer-granularity locking strategy is desirable that allows higher DOC while guaranteeing streaming consistency. Without loss of generality, consider two threads \( T_1 \) and \( T_2 \) which are launched at time \( t_1 \) and \( t_2 \), respectively (\( t_1 < t_2 \)). Assume that \( o_1 \) and \( o_2 \) are two consecutive operations of \( T_1 \) and \( T_2 \), respectively. If \( o_1 \) conflicts with \( o_2 \), \( o_1 \) will be executed before \( o_2 \) in the proposed scheme. Thus, if \( o_1 \) happened after \( o_2 \), then \( o_1 \) would not conflict with \( o_2 \) and we can swap the execution order of \( o_1 \) and \( o_2 \) without causing any inconsistency. 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. An example for lock request dispatching is presented in Figure 13.

**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 (\( \Omega(j) \)) using the naïve representation (like Figure 7) without MS-tree. The locking strategy over MS-tree is more challenging that will be discussed in Sections V-C.

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 the lock that \( T \) applies are both shared locks. Otherwise, thread \( T \) will wait until it is woken up by the thread that just 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.

**Proof:** Without loss of generality, consider two threads \( T_1 \) and \( T_2 \) which are launched at time \( t_1 \) and \( t_2 \), respectively (\( t_1 < t_2 \)). Assume that \( o_1 \) and \( o_2 \) are two consecutive operations of \( T_1 \) and \( T_2 \), respectively. If \( o_1 \) conflicts with \( o_2 \), \( o_1 \) will be executed before \( o_2 \) in the proposed scheme. Thus, if \( o_1 \) happened after \( o_2 \), then \( o_1 \) would not conflict with \( o_2 \) and we can swap the execution order of \( o_1 \) and \( o_2 \) without causing any inconsistency. Hence, the execution of \( T_1 \) and \( T_2 \) in the generated schedule can be always converted to the schedule where \( T_2 \) does not start until \( T_1 \) finishes without causing inconsistency. Consequently, the generated schedule is conflict-equivalent to the schedule where transactions are executed serially in chronological order based on their timestamps. Therefore, the transaction schedule obeys streaming consistency.

**C. Concurrent Access over MS-tree**

Consider an expansion list \( \{L^1, L^2, \ldots, L^k\} \) 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.
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
$\sigma_i$ is an incoming edge at $t_i$
$\sigma_i$ is an expired edge at $t_i$
if $\sigma_i$ does not match any edge in query $Q$ then
CONTINUE
else
Let $\Gamma$ be all lock requests for adding edge $\sigma_i$
for each lock request in $\Gamma$ do
/*DISPATCH lock requests*/
append it to the end of the corresponding wait-list;
CREATE a new thread over $\text{Ins}(\sigma_i)$ (Algorithm 1)
if $\sigma_j$ does not match any edge in query $Q$ then
CONTINUE
else
Let $\Gamma$ be all lock requests for adding edge $\sigma_j$
for each lock request in $\Gamma$ do
/*DISPATCH lock requests*/
append it to the end of the corresponding wait-list;
CREATE a new thread for $\text{Del}(\sigma_j)$ (Algorithm 2)
end
end
end

Algorithm 4: Applies/releases S/X-lock

Input: An item $L^i$ and the corresponding wait-list $\text{waitlist}(L^i)$
Input: Current thread $T$
Output: whether successfully applies/releases S/X-lock over $L^i$

function apply_S/X-lock()
do
while the lock request of $T$ is not at the head of $\text{waitlist}(L^i)$ OR the lock status of $L^i$ is exclusive do
thread_wait()
apply S/X-lock over $L^i$
pop the head of $\text{waitlist}(L^i)$
end
function release_S/X-lock()
release S/X-lock over $L^i$

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 Figure 10. Assuming that a thread $T_1$ is reading partial matches of $\{e_6, e_5\}$, $T_1$ will backtrack from node $n_2^1$ (i.e., $\sigma_3$) to read $n_1^1$ (i.e., $\sigma_1$). Since $T_1$ only locks $L_1^2$, if another thread $T_2$ is deleting $n_1^1$ 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.

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 $3 \times 3 = 9$ 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 $n$ during the backtrack to find the corresponding whole path, $T_2$ has already deleted $n$, 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 $\sigma_{d_1}$ as a child of $\sigma_{d_i-1}$, $T_2$ has deleted $\sigma_{d_i-1}$, which causes the inconsistency.

Proof: clearly, the two cases identified will cause inconsistency; let’s discuss why the remaining 25 cases will not cause inconsistency. When $d_1 = d_2$, $T_1$ and $T_2$ must be reading the same item and they will not conflict. Thus, we consider the cases where $d_1 \neq d_2$ or both $T_1$ and $T_2$ are reading partial matches.

1) If $d_1 < d_2$:

   a) If $T_1$ is reading partial matches of $L^{d_1}$: (1) if $T_2$ is inserting new partial matches into $L^{d_2}$, $T_2$ will only add new children for some nodes of depth $d_2 - 1$ and $T_1$ will not conflict with $T_2$; (2) if $T_2$ is deleting partial matches in $L^{d_2}$, the depth of expired nodes is not less than $d_2$ and $T_1$ will not conflict with $T_2$.

   b) If $T_1$ is inserting partial matches into $L^{d_1}$: if $T_1$ has lock request in the wait-list of $L^{d_2}$, then the lock request must be before that of $T_2$ ($T_1$ is launched before $T_2$), which means $T_2$ will not access $L^{d_2}$ until $T_1$ finishes accessing $L^{d_2}$ (in the future), thus, $T_1$ will not conflict with $T_2$. Otherwise, $T_1$ will not access partial matches in $L^{d_2}$ and (1) if $T_2$ is reading partial matches of $L^{d_2}$, the backtrack of $T_2$ will not involve the new nodes $T_1$ adds, and $T_1$ will not conflict with $T_2$; (2) if $T_2$ is not reading partial matches of $L^{d_2}$, there will be no common data that both $T_1$ and $T_2$ will access, thus, $T_1$ will not conflict with $T_2$.

   c) If $T_1$ is deleting partial matches in $L^{d_1}$, then $T_1$ must have lock request in the wait-list of $L^{d_2}$ before that of $T_2$ ($T_1$ is launched before $T_2$), and $T_2$ can not currently access $L^{d_2}$. Thus, $T_1$ will not conflict with $T_2$.

2) If $d_2 < d_1$:

   a) If $T_2$ is reading partial matches of $L^{d_2}$, similar to the case where $d_1 < d_2$, $T_1$ will not conflict with $T_2$.

   b) If $T_2$ is inserting partial matches into $L^{d_2}$, similar to the case where $d_1 < d_2$, either $T_2$ can not access $L^{d_2}$ until $T_1$ finish accessing $L^{d_2}$ (in the future) or there is no common data that both $T_1$ and $T_2$ will access. Thus, $T_1$ will not conflict with $T_2$.

   c) If $T_2$ is deleting partial matches in $L^{d_2}$ and $T_1$ is
deleting partial matches in \( L^{d_1} \), then \( T_1 \) will not access the expired nodes \( T_2 \) delete and \( T_1 \) will not conflict with \( T_2 \).

Thus, only two cases of the 27 will cause inconsistency in our computation.

Theorem \([3]\) 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 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 \( n_{d_2} \) of depth \( d_2 \) in \( M \), \( T_2 \) does not “totally” remove \( n_{d_2} \) from \( M \). Instead, \( T_2 \) “partially” removes \( n_{d_2} \) as follows: (1) \( T_2 \) removes \( n_{d_2} \) from the corresponding doubly linked list, and (2) \( T_2 \) disables the link (pointer) from \( n_{d_2} \)’s parent to \( n_{d_2} \) while the link from \( n_{d_2} \) to its parent remains. For example, Figure 14 demonstrates how an expired node \( n_{d_2} \) is removed. After \( T_2 \) partially removes all expired nodes, \( T_2 \) will finally remove all expired nodes from MS-tree \( M \).

![Fig. 14: Demonstration of partially remove](image)

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

**Proof:** Consider the two cases that will cause inconsistency (Theorem \([3]\)). For the first case, since \( T_1 \) is reading partial matches of \( L^{d_1} \) (\( d_1 > d_2 \) and \( t_1 < t_2 \)), \( T_2 \) does not partially remove expired nodes of depth \( d_1 \) yet, which means that \( T_2 \) does not totally remove any expired nodes according to the modified deletion strategy. Thus, \( T_1 \) can still safely access any nodes it wants to. When \( T_2 \) starts to totally remove expired nodes, it must has already partially removed all expired nodes – that is, \( T_2 \) must have removed its lock requests from all \( L^{d_1} \) where \( d_1 > d_2 \), which also means that threads launched before \( T_2 \) have already finished their computation over the MS-tree \( M \) (Otherwise, \( T_2 \) would not have partially removed all expired nodes) and thus, it is safe for \( T_2 \) to finally remove all expired nodes. Similarly, the second case will neither cause inconsistency.

Our scheduling strategy over the MS-tree is different from the traditional tree protocol \([15]\). 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.

### VI. Decomposition

For a non-TC-query \( Q \), the query evaluation method in Section III-B needs to decompose \( Q \) into a set of TC-subqueries \( D = \{Q^1, ..., Q^k\} \). Although the query decomposition has been studied in subgraph query problem \([1], [16]\), none of these existing methods guarantees the decomposed subquery is a TC-query. Thus, none of them can be used in our problem.

We propose a cost model-guided TC decomposition of query \( Q \) based on the intuition that an incoming edge \( \sigma \) should lead to as few join operations as possible. Cost of join operations varies in stream scenario and we only 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.

#### A. Cost Model

Assume that \( Q \) has \(|E(Q)|\) query edges \( \epsilon_j (j=1,...,|E(Q)|) \) and \( Q \) is decomposed into \( k \) TC-subqueries \( Q^i (i = 1, ..., k) \). For simplicity, we assume that the probability of any incoming edge \( \sigma \) matches each edge \( \epsilon_j \) in \( Q \) is \( 1/d \), where \( d \) is the number of distinct term edge labels (i.e., the label combining edge label and the connected node labels) in \( Q \). Theorem 7 tells us the expected number of join operation (in worst case) for an incoming edge.

**Theorem 7:** Consider an incoming edge \( \sigma \) that matches one or more edges in query \( Q \). The total expected number of join operations for \( InvS(\sigma) \) is

\[
N = \frac{1}{d}(|E(Q)| - 1) + \frac{k}{2}(k - 1)
\]

where \( k \) is the number of TC-subqueries in the decomposition and \( d \) is the number of distinct edge labels in \( Q \).

**Proof:** Although an incoming edge may match more than one query edges in \( Q \), the probability for each query edge in \( Q \) to be matched by an incoming edge is \( 1/d \) where \( d \) is the number of distinct edge labels in \( Q \). When an incoming edge \( \sigma \) matches an edge \( \epsilon \) in \( Q \), our method first computes the matches of TC-subquery \( Q^i \) (Section III-A3) and then joins TC-subquery matches to find the matches of query \( Q \) (Section III-B). In the first step, if \( \sigma \) matches the first edge \( \epsilon_1 \) of the timing sequence \( \{\epsilon_1, ..., \epsilon_{|Q|}\} \) for a TC-subquery \( Q^1 \), there exists no join operation. If \( \sigma \) matches edge \( \epsilon_i \) (\( i > 1 \)), it needs to join with \( \Omega(P_{req}(\epsilon_{i-1})) \) that is recorded in MS-tree. Thus, it leads to one join operation. Assume that \( Q \) is decomposed into \( k \) TC-subqueries. Therefore, the expected number of join operations is

\[
N_1 = \frac{1}{d}(|E(Q)| - k) = \frac{k}{2}(|E(Q)| - k)
\]

In the second step, we join TC-subquery matches to obtain matches of query \( Q \). We consider the number of join operations in the worst case. If \( \sigma \) contributes to a new match of TC-subquery \( Q^1 \), it needs to join all other TC-subqueries \( Q^2, ..., Q^k \). Therefore, it leads to \( (k - 1) \) join operations in the worst case. If \( \sigma \) contributes to a new match of TC-subquery \( Q^i \) (\( i > 1 \)), it needs to join \( \Omega(\sum_{j=1}^{i-1}(|Q^j|)) \), which
is recorded in the upper levels of $M_0$ tree, and then join with the left TC-subqueries $Q_i^{1}, \ldots, Q_i^k$. Thus, if $i > 1$, it leads to $(k-i)+1$ join operations. Therefore, the expected number of join operations in the second step is

$$N_2 = \frac{1}{d} \sum_{i=2}^{k} ((k-i+1) + (k-1)) = \frac{1}{d} \left( \frac{k^2 + k}{2} - 1 \right)$$

Thus, the total expected number of join operations is

$$N = N_1 + N_2 = \frac{1}{d} |E(Q)| - 1 + \frac{k}{2} (k-1)$$

Since $|E(Q)|$ and $d$ are fixed, the total expected number of join operations $(N)$ increases with $k$. Therefore, we prefer to find a TC decomposition of size as small as possible.

### B. Decomposition Method

Given a query $Q$, to find a TC decomposition of size as small as possible, we propose the following solution. We first extract all possible TC-subqueries of $Q$, denoted as $TCsub(Q)$. For a TC-subquery $Q_i$ of timing sequence $\{\epsilon_1, \ldots, \epsilon_k\}$, according to the definition of TC-query, any prefix of the timing sequence constitutes a TC-subquery of $Q_i$. Thus, we can compute $TCsub(Q)$ by dynamic programming:

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 $\epsilon_x$ such that $\epsilon_x \prec \epsilon_j$. If $\epsilon_x$ has 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.

The pseudocode is presented in Algorithm 5. After computing $TCsub(Q)$, we need to 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.

For example, consider query $Q$ in Figure 5. The corresponding $TCsub(Q)$ contains 10 TC-subqueries: $\{\epsilon_6, \epsilon_5, \epsilon_4\}, \{\epsilon_3, \epsilon_1\}, \{\epsilon_5, \epsilon_4\}, \{\epsilon_6, \epsilon_5\}, \{\epsilon_2\}, \{\epsilon_3\}, \{\epsilon_4\}, \{\epsilon_5\}, \{\epsilon_6\}$. We first choose $\{\epsilon_6, \epsilon_5, \epsilon_4\}$ from $TCsub(Q_0)$, followed by $\{\epsilon_3, \epsilon_1\}$ and then $\{\epsilon_2\}$. Thus, the TC decomposition of $Q$ will be $\{(\epsilon_6, \epsilon_5, \epsilon_4), (\epsilon_3, \epsilon_1), (\epsilon_2)\}$ (Figure 6). The pseudo code for conducting decomposition over $TCsub(Q)$ is presented in Algorithm 6.

### C. Join order

Given a decomposition $D = \{Q^1, Q^2, \ldots, Q^k\}$ of query $Q$, we obtain the matches of $Q$ as $\Omega(Q) = \Omega(Q^1) \preceq \Omega(Q^2) \ldots \preceq \Omega(Q^k)$, in which the join order follows the prefix-connected sequence. Obviously, a good join order should lead to fewer intermediate results. Intuitively, we first find two TC-subqueries $Q^i$ and $Q^j$, where $|\Omega(Q^i) \bowtie \Omega(Q^j)|$ is minimum among $D$ as the first two TC-subqueries in the targeted decomposition. Iteratively, we find other TC-subqueries resulting in small intermediate result sizes. Join selectivity can be estimated according to the data distribution that has been well-studied. However, this is infeasible for streaming graph data due to dynamic data distribution. Thus, we propose a simple yet effective heuristic rule.

**Definition 12 (Joint Number):** Given two TC-subqueries $Q^i$ and $Q^j$ ($i \neq j$), the joint number between $Q^i$ and $Q^j$, denoted as $JN(Q^i, Q^j) = n_v + n_e$, where $n_v$ is the number of common vertices between $V(Q^i)$ and $V(Q^j)$ and $n_e$ is the number of edge pairs $(\epsilon_i, \epsilon_j) \in E(Q^i) \times E(Q^j)$ such that there is timing order between $\epsilon_i$ and $\epsilon_j$.

Given a decomposition $D = \{Q^1, Q^2, \ldots, Q^k\}$ of query $Q$, we first find two TC-subqueries $Q^1$ and $Q^2$ that are connected with each other, and the joint number between them is maximum among all pairwise TC-subqueries. Iteratively, we find the TC-subquery $Q^3$, which is connected to $Q^1 \cup Q^2$ and the joint number between $Q^1 \cup Q^2$ and $Q^3$ is minimum among all left TC-subqueries. We repeat the above process to find the whole prefix-connected sequence over decomposition $D$ of query $Q$, which specifies the corresponding join order.
VII. Experimental Evaluation

We evaluate our solution against comparable approaches. All methods are implemented in C++ and run on a CentOS machine of 128G memory and two Intel(R) Xeon(R) E5-2640 2.6GHz CPU. Codes and query sets are available at [18]. We also present a case study in the full paper [17].

A. Datasets

We use three datasets in our experiments: real-world network traffic dataset, wiki-talk network dataset and synthetic social stream benchmark.

The network traffic data is the “CAIDA Internet Anonymized Traces 2015 Dataset” obtained from www.caida.org. The network data contains 445,440,480 communication records (edges) concerning 2,601,005 different IP addresses (vertices). Each edge is associated with a timestamp indicating the communication time.

A network communication record is a five-tuple that includes the source IP address/port number, the destination IP address/port number and the protocol in use. We transform these five-tuples into a vertex/edge labelled streaming graph. Note that although we only study the vertex-labelled graphs in this paper, it is straightforward to extend our method to edge-labelled graphs. The vertex label is fixed as “IP”. Each edge label is a triple (source port, destination port, protocol). Since the source ports vary a lot resulting in very low matching ratio of query edges, we replace source port by a wildcard “*” that can match any source port. In fact, there are 65520 different destination ports where the top 6 (i.e., the top 0.01%) frequent ports exist in more than 50% (i.e., 222,720,240) communication records.

The wiki-talk dataset is from the Standford SNAP library [19] where a directed edge (A, B, t) indicates that user A edit user B’s talk page at time t. This dataset contains 1,140,149 vertices and 7,833,140 edges and the total time span is 2,320 days. We use the first character of the user’s name to be the label of a vertex.

Linked Stream Benchmark [20] is a synthetic streaming social graph data. There are three different components in the streaming data. The GPS stream contains user’s trace specified by longitudes, latitudes and the corresponding tracking time. The Post stream contain posts from users and the information of photos uploaded by users. We set the user number parameter as 1 million and the entire time span of the streaming data as 10 days. Parameters except the user number and time span are applied in the default setting. The streaming social data contains 209,549,677 edges and 37,231,144 vertices. The data generator is available in Google Code [21]. Each record in the streaming data is a five-tuple consisting of subject type/id, predicate, object type/id. We also build a streaming graph over the streaming data where vertex labels are the corresponding subject/object types and edge labels are the predicates.

B. Query Generation

A usual method for generating query graph is to perform the random walk over the data graph. However, generating query graph with timing order constraints is non-trivial since we need to make sure that (1) timing order should be generated inherently with randomness to be representative and (2) query graph with the timing order should still have embedding (including the chronological order between edges) in the data graph. We propose a method generating queries satisfying these conditions and the corresponding average selectivities of these queries are reported in Figure 25 in Section VII-H. When generating a query, we first conduct a random walk over the data graph and retrieve a subgraph g where we generate random permutation of edges, assumed as \( \{\epsilon_1, \epsilon_2, ..., \epsilon_k\} \), then we set \( \epsilon_i \prec \epsilon_j \) if and only if (1) \( \epsilon_i \) is before \( \epsilon_j \) in the permutation; and (2) the timestamp of \( \epsilon_i \) in g is less than that of \( \epsilon_j \). In this way, we generate a query Q with graph structure g and timing order \( \prec \). For edges in g, the positional order in random permutation and the order that they appear in the stream (chronological order of the corresponding timestamps) are independent of each other and hence the way we create timing order \( \prec \) is of randomness. In this way, the query Q we generate not only guarantees the representativeness but also makes sure that there exist subgraph in data graph that satisfies both time order and structure constraints of Q.

We generate 300 queries over each dataset in our experiments. For each dataset, we set six different query sizes: 6, 9, 12, 15, 18, 21. For each query size, we generate 10 query graphs by random walks over data graph. For each query graph g, we create 5 different timing orders over g where one is set as full order, one is set as \( \emptyset \) and the other three are created by random permutations as illustrated previously.

C. 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 VII-D. 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. Since it does not handle the timing order constraints, we verify answers from SJ-tree posteriorly with the timing order. IncMat [11] 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 QuickSIFT [6], TurboLSO [7]. BoostSO [3]. These methods are conducted over the affected area (see [11]) window by window. To evaluate the effectiveness of MS-tree, we also compare our approach with a counterpart without MS-trees (called Timing-IND) where every partial match is stored independently.

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 (i.e., the ratio of the total time span of whole dataset to the total number of data edges).

We evaluate the systems by varying window size |W| and query size |E(Q)|. In Section VII-G, we also compare our methods with comparative ones when varying the decomposition size k. The reported throughput (The number of edges
handled per second) and space under a given group settings are obtained by averaging those from the corresponding generated queries.

the space and maintenance cost, it is not as good as Timing, as shown in our experiments.

Figures 15-16 show that the time efficiency comparison: Figures [15][16] 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-tree 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. Finally, since Timing-IND does not use MS-tree to optimize

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 Figures [17][18] 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, and Boost need 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.
D. Concurrency Evaluation

We evaluate the performance of 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 Figures 19 and 20. 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. Our method applies the fine-grained locking strategy and different threads may not conflict. In this case, the degree of concurrency increases when the number of threads grows. However, All-locks-\(N\) always locks all items that may be accessed in a thread. In other words, All-locks-\(N\) is almost the same as the sequential algorithm. This is why the speed up of All-locks-\(N\) is always about 1.2. Figure 20 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.

![Speedup over Different Window Size](image1)

![Speedup over Different Query Size](image2)

E. Decomposition and Join Order

We evaluate the effectiveness of our decomposition strategy and selection of the join order. We implement three alternative solutions: to evaluate the decomposition strategy, we design an alternative that randomly retrieves a decomposition from \(TC_{sub}(Q)\) for a given query \(Q\) (denoted as Timing-RD); to evaluate the join order selection, we design a second alternative that randomly chooses a prefix-connected sequence (join order) over a given decomposition \(D = \{P_1, P_2, ..., P_k\}\) (denoted as Timing-RJ), and a third that applies random decomposition and uses random prefix-connected sequence (denoted as Timing-RDJ). In the evaluation, we fix the window size to 30,000. Figure 21 shows that our solution outperforms the alternatives. The main reason is that the decomposition and join order strategy reduces the partial matches we need to maintain, which further helps reduce the time cost for computation over those partial matches.

F. Case Study

We evaluate our solution over an internal non-anonymous network traffic data. Note that we did not use the network traffic data from CAIDA since it is anonymous and there is no way for us to verify whether the detected patterns indicate attacks or not. Our collected dataset contains all traffic data of more than 20 windows/linux servers/PCs. The time span of the traffic starts from Sept. 1st, 2017 while ends at Dec. 11th, 2017, when one windows server (of IP address 59.**.222.36) was found compromised as slaves of a Zeus botnet. It is confirmed that the windows server was compromised at Nov. 28th, 2017 when it started to frequently communicate with a C&C server (of IP address 101.*.81.189) through domain names that are created by Gameover DGA.

The pattern (query graph) we monitor is exactly the one in Figure 1. We set the window size of 30 seconds which is long enough for an attack of such pattern. We remove all traffic accessing top 10,000 websites in Alexa Rank\(^1\) which is a common source of whitelist of websites in cyber-security field. We find that our algorithms successfully detected a match happened at Nov. 28th as presented in Figure 22. The timestamp

---

\(^1\)https://www.alexa.com/topsites
of each edge is in the original format when processed by tcpdump. Specifically, the window server \((V^1)\) of IP 59.**.222.36 communicated with a web server (IP: 125.**.254.78 through website address “www.***.edu.cn/27/ketiyanshou.htm”) and then built connection with a C&C server (IP: 101.**.81.189 through domain name “tevwugbt8yjychael1ggb.net”). Apparently, if we monitor this pattern on the Windows server at that time, we can stop the connections to the C&C server much sooner.

Table 1: Detected attack graph

| t1: 11:05:22.381250 | t2: 11:05:25.86193 | t3: 11:05:25.58652 | t4: 11:05:26.78199 | t5: 11:05:24.781250 |
|-------------------|-------------------|-------------------|-------------------|-------------------|

Fig. 22: Detected attack graph

G. Decomposition Size \(k\)

We evaluate our solution with comparative ones over query of different decomposition size \((k)\). We fix the query size as 12 and window size as 30,000. We set five different \(k\): 1, 3, 6, 9, 12. To generate a query of a specific decomposition size \(k\), we can constantly create timing order \(\prec\) over a retrieved subgraph \(g\) (by varying permutation of \(g\)'s edges) until \(g\) and \(\prec\) constitute a query that can be decomposed into \(k\) TC-subqueries according to our decomposition strategy. In fact, for \(k = 1\), we assign the timing order between every two edges in \(g\) according to their timestamps in the data graph, while for \(k = 12\), we just set the timing order as \(\emptyset\). We present the throughput and space cost of all methods over different \(k\) in Figure 23 and Figure 24, respectively. Our method outperform existing ones obviously. Specifically, the throughput of our method is higher than others by nearly one order of magnitude and our data structure cost much less space than comparative ones do. With the decomposition size increases, the throughput of our method decrease while the space cost increase, which confirms our claim that \(k\) should be as small as possible.

H. Selectivity of Query Set

We report the selectivity of the generated queries varying window size and query size. We present the corresponding average number of answers of these generated queries in Figure 25. We can see that the number of answers almost decreases with the growth of the query size while increases with the growth of the window size.

VIII. CONCLUSIONS

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.

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 International Conference on Extending Database Technology, 2015, pp. 157–168.

[2] Verizon, http://www.verizonenterprise.com/resources/reports/2016_Report_en_xg.pdf, 2016.

[3] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou, “Real-time constrained cycle detection in large dynamic graphs,” Proceedings of the VLDB Endowment, vol. 11, no. 12, 2018.

[4] J. R. Ullmann, “An algorithm for subgraph isomorphism,” Journal of the ACM, vol. 23, no. 1, pp. 31–42, 1976.

[5] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, “A (sub) graph isomorphism algorithm for matching large graphs,” IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 26, no. 10, pp. 1367–1372, 2004.

[6] 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.

[7] 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 International Conference on Management of Data. ACM, 2013, pp. 337–348.

[8] 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.

[9] H. He and A. K. Singh, “Closure-tree: An index structure for graph queries,” in Proc. 22nd IEEE International Conference on Data Engineering. IEEE, 2006, pp. 38–38.

[10] E. B. Krissinel and K. Henrick, “Common subgraph isomorphism detection by backtracking search,” Software: Practice and Experience, vol. 34, no. 6, pp. 591–607, 2004.

[11] W. Fan, X. Wang, and Y. Wu, “Incremental graph pattern matching,” ACM Transactions on Database Systems, vol. 38, no. 3, p. 18, 2013.

[12] L. Chen and C. Wang, “Continuous subgraph pattern search over certain and uncertain graph streams,” IEEE Transactions on Knowledge and Data Engineering, vol. 22, no. 8, pp. 1093–1109, 2010.

[13] J. Gao, C. Zhou, J. Zhou, and J. X. Yu, “Continuous pattern detection over billion-edge graph using distributed framework,” in Proc. 30th IEEE International Conference on Data Engineering, 2014, pp. 556–567.

[14] C. Song, T. Ge, C. X. Chen, and J. Wang, “Event pattern matching over graph streams,” Proc. VLDB Endowment, vol. 8, no. 4, pp. 423–424, 2014. [Online]. Available: http://www.vldb.org/pvldb/vol8/p413-ge.pdf

[15] A. Silberschatz and Z. Kedem, “Consistency in hierarchical database systems,” Journal of the ACM, vol. 27, no. 1, pp. 72–80, 1980.

[16] B. T. Messmer and H. Bunke, “A new algorithm for error-tolerant subgraph isomorphism detection,” IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 20, no. 5, pp. 493–504, 1998.

[17] “Code,” https://github.com/pkumod/timingsubg.git, 2018.

[18] “Full paper,” http://arxiv.org/abs/1801.09240, 2018.

[19] “Wikitalk,” http://snap.stanford.edu/data/wiki-talk-temporal.html

[20] D. Le-Phuoc, M. Dao-Tran, M.-D. Pham, P. Boncz, T. Eiter, and M. Fink, “Linked stream data processing engines: Facts and figures,” in The 11th International Semantic Web Conference. Springer, 2012, pp. 300–312.

[21] “Lsbench code,” https://code.google.com/archive/p/lsbench/ 2017.