Time Constrained Continuous Subgraph Search over Streaming Graphs

Youhuan Li†,£, Lei Zou†,±,∓, M. Tamer Özsu‡, Dongyan Zhao†
†*Peking University, China;* £*Center for Data Science, Peking University, China;* ‡*University of Waterloo, Canada;*
±*Beijing Institute of Big Data Research, China;*
∓ *National Engineering Laboratory for Big Data Analysis Technology and Application (PKU), China*
†{liyouhuan,zoulei,zhaody}@pku.edu.cn, ‡tamer.ozsu@uwaterloo.ca

***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.**

***Keywords*—Streaming Graphs, Subgraph, Timing Order**

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

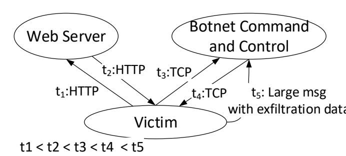


Fig. 1: Query example in Network Traffic (Taken from [1])

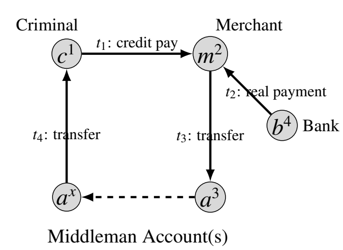


Fig. 2: Credit card fraud in transactions (Taken from [3])

Example 2. Credit-card-fraud pattern.

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.

2375-026X/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDE.2019.00100

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.

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.

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 8) 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

TABLE I: 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 j-th 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
n / n_i^j	A node in a MS-tree / The j-th node in the MS-tree for L_i
$TCsub(Q)$	The set of all TC-subqueries of query Q

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 a particular 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. 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 $\mathbb{G}$ is shown in Figure 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 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.

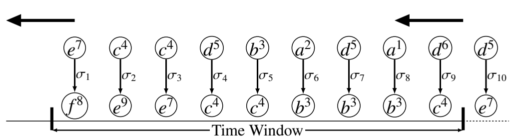
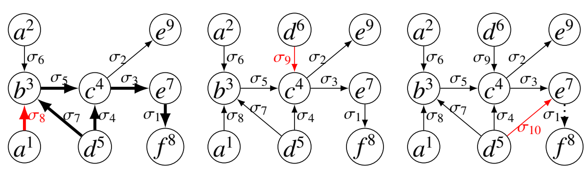
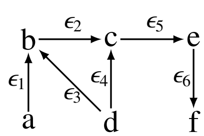


Fig. 3: Graph stream $\mathbb{G}$ under time window of size 9



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

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



(a) query graph

$\epsilon_6 \prec \epsilon_3 \prec \epsilon_1$

$\epsilon_6 \prec \epsilon_5 \prec \epsilon_4$

(b) timing order

Fig. 5: Running example query Q

Definition 2 (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 $\mathbb{E}_t$, namely:

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

The snapshots of graph stream $\mathbb{G}$ at time points $t = 8, 9, 10$ for $|W| = 9$ are given in Figure 4. Note that at timestamp $t = 10$, edge σ_1 expires since the time point of σ_1 is 1 and the timespan of time window W is $(1, 10]$. The expired edges are denoted with dotted edges in Figure 4c while newly added edges are in red.

Definition 3 (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 Figure 5. 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 ϵ_1 should arrive before edges matching ϵ_2 in subgraph matches of Q over the snapshot (see Definition 4) 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)**
   - $\forall u \in V(Q), L(u) = L(F(u))$.
   - $\overrightarrow{uv} \in E(Q) \Leftrightarrow \overrightarrow{F(u)F(v)} \in E(g)$.
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 Figure 4a (highlighted by bold line) is not only isomorphic to query Q but also conforms to the timing order constraints defined in Figure 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.

III. 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 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], Turbo$_{\text{ISO}}$ [7], Boost$_{\text{ISO}}$ [8]) on each snapshot $\mathbb{G}_i$ ($i = 1, ..., \infty$) 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 $\Delta(\mathbb{G}_i)$ of $\mathbb{G}_i$ that is affected by the updated edge (insertion/deletion) and then conduct subgraph isomorphism algorithm over $\Delta(\mathbb{G}_i)$ instead of the whole snapshot $\mathbb{G}_i$. While, if the query diameter is d, then $\Delta(\mathbb{G}_i)$ 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 $\Delta(\mathbb{G}_i)$ 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 $\mathbb{G}_i$ 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 Figure 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, it is inevitable to introduce much 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 5) by considering the timing order in the query graph.

Definition 5 (Discardable Edge): For a streaming graph $\mathbb{G}$ and a query graph Q, 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.

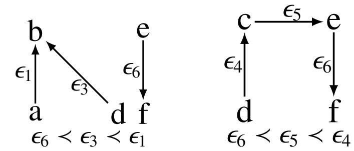
To better understand discardable edge, recall the streaming graph $\mathbb{G}$ in Figure 3. At time t_6, an incoming edge σ_6 (only matching ϵ_1) is added to the current time window. Consider the timing order constraints of query Q in Figure 5, which requires that edges matching ϵ_3 should come before 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 6 (Prerequisite Edge/Prerequisite 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 3. The subquery of Q induced by edges in $Preq(\epsilon)$ is called a *prerequisite subquery* of ϵ in query Q.

Consider two edges ϵ_1 and ϵ_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 σ in streaming graph $\mathbb{G}$ is *discardable*(All proofs of lemmas and theorems are presented in the full version of this paper [15]).



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

Fig. 6: Example of prerequisite subquery

Lemma 1: An incoming edge σ at time t_i is *NOT discardable* if and only if, at the current snapshot $\mathbb{G}_i$, 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 $\mathbb{G}_i$) 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, 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 incoming edge.

For certain types of queries that we call *timing-connected query* (Definition 8), it is easy to determine if an edge σ in streaming graph $\mathbb{G}$ 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.

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 4) 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.

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 $\forall j \in [1, k-1]$, $\epsilon_j \prec \epsilon_{j+1}$. 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 $\epsilon_6 \prec \epsilon_5 \prec \epsilon_4$ 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) 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 9 (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 item L^i corresponds to $\bigcup_{j=1}^{i}(\epsilon_j)$, i.e., $Preq(\epsilon_i)$.
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 $Preq(\epsilon_i)$ ($i \in [1, k]$). 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.

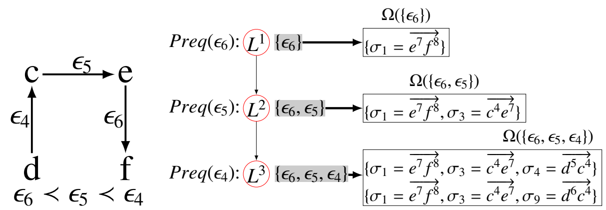
The shaded nodes in Figure 7 illustrate the prerequisite subqueries for a TC-query with timing sequence $\{\epsilon_6, \epsilon_5, \epsilon_4\}$. Since each node corresponds to a subquery $Preq(\epsilon_i)$, we also record the matches of $Preq(\epsilon_i)$, as shown in Figure 7. The last item stores 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 σ, 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 σ matches query edge ϵ_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 $i = 1$, 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 $\Omega(L^{i-1}) \overset{T}{\bowtie} \{\sigma\}$ should be inserted into L^i as new matches of $Preq(\epsilon_i)$, where $\Omega(L^{i-1})$ is the set of partial matches in L^{i-1}.



(a) TC-query (b) Expansion list

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

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 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.

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 σ that matches ϵ_i, the time to determine whether σ 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}.

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.

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.

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, $Q = \bigcup_{i=1}^{k}(Q^k)$ 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 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 ($i = 1, ..., k$) 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

Algorithm 1: INSERT(σ)

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

1 **for** *each query edge* ϵ *that* σ *matches* **do**
2   Assume that ϵ is the j-th edge in TC-subquery Q^i.
3   **if** $j == 1$ **then**
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   INSERT($\Delta(L_i^j)$, 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}) **// Read partial matches in** L_0^{i-1}
16       $\Delta(L_0^i) = \Delta(L_i^j) \overset{T}{\bowtie} \Omega(L_0^{i-1})$
17   INSERT($\Delta(L_0^i)$, 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   INSERT($\Delta(L_0^{i+1})$, L_0^{i+1}) **// Insert** $\Delta(L_0^{i+1})$ **into** L_0^{i+1}
22       $i++$
23   **if** $\Delta(L_0^k) \neq \emptyset$ **then**
24   **Report** $\Delta(L_0^k)$ **as new matches of** Q

Section VI-B. We use $L_i = \{L_1^1, L_i^2,..., L_i^{|E(Q^i)|}\}$ to denote the corresponding expansion list for each TC-subquery Q^i. Recall the definition of prefix-connected sequence (Definition 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 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 the full paper [15]. For this paper, we assume that the prefix-connected sequence $D = \{Q^1, Q^2,..., Q^k\}$ is given.

For example, Figure 8 illustrates a decomposition of query Q (Q^1, Q^2, Q^3). We obtain the matches of Q as $\Omega(Q) = \Omega(Q^1) \overset{T}{\bowtie} \Omega(Q^2)... \overset{T}{\bowtie} \Omega(Q^k)$. 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 $L_0 = \{L_0^1, L_0^2, L_0^3\}$ (for query Q) is given in Figure 8. Each item L_0^i records the intermediate join results $\Omega(\bigcup_{x=1}^{i} Q^x)$.

Assume that an incoming edge σ contributes to new matches of TC-subquery Q^i (denoted as $\Delta(L_i^{|L_i|})$) . If $i > 1$, we let $\Delta(L_0^i) = \Delta(L_i^{|L_i|}) \overset{T}{\bowtie} \Omega(L_0^{i-1})$ (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 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 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.

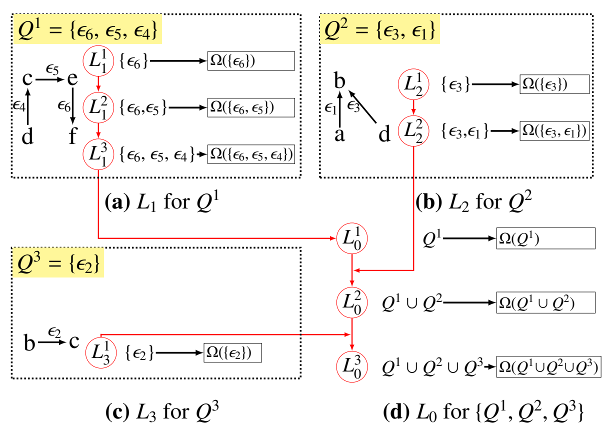


Fig. 8: An TC decomposition of query Q

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.

A. Match-Store Tree

Consider an expansion list $L = \{L^1, L^2,...,L^k\}$ over timing sequence $\{\epsilon_1, \epsilon_2,...,\epsilon_k\}$ where L^i stores all partial matches of $\{\epsilon_1, \epsilon_2,...,\epsilon_i\}$. For a match g of L^i $(1 \le i \le k)$, 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 $\sigma_{i'}$ $(1 \le i' \le i)$ is a match of $\epsilon_{i'}$. Furthermore, $g' = g \setminus \{\sigma_i\} = \{\sigma_1, \sigma_2,..,\sigma_{i-1}\}$, as a match of $\{\epsilon_1, \epsilon_2,...,\epsilon_{i-1}\}$, 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 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 in the expansion list.

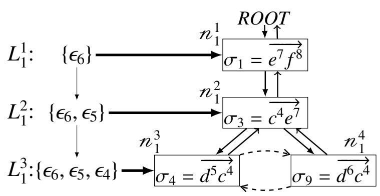


Fig. 9: MS-tree of expansion list $L_1 = \{L_1^1, L_1^2, L_1^3\}$

Definition 10 (Match-Store Tree): Given a TC-query Q with timing sequence $\{\epsilon_1,\epsilon_2,...,\epsilon_k\}$ and the corresponding expansion list $L = \{L^1,L^2,...,L^k\}$, 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 $(1 \le i \le k)$ in a MS-tree denotes a match of ϵ_i and all nodes along the path from the root to node n together constitute a match of $\{\epsilon_1,\epsilon_2,...,\epsilon_i\}$. 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 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 ($\sigma_1 \to \sigma_3 \to \sigma_4$) in the MS-tree. Furthermore, partial match $\{\sigma_1, \sigma_3, \sigma_9\}$ shares the same prefix path ($\sigma_1 \to \sigma_3$) with $\{\sigma_1, \sigma_3, \sigma_4\}$. Thus, MS-tree greatly reduces the space cost for storing all matches by compressing the prefix.

B. 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.,partial matches containing expired edge). These three 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 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 $\{\epsilon_1,\epsilon_2,...,\epsilon_i\}$: $g = \{\sigma_1, \sigma_2,...,\sigma_i\}$ where each σ_j matches ϵ_j, we need to insert a path $\{root \to \sigma_1 \to \sigma_2... \to \sigma_i\}$ 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 $\{root \to \sigma_1 \to \sigma_2... \to \sigma_{i-1}\}$ 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 $\{root \to \sigma_1 \to \sigma_3\}$ by adding σ_9 as a child of σ_3 (see Figure 9). Note that, 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 $\{root \to \sigma_1 \to \sigma_2... \to \sigma_{i-1}\}$ 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 Figure 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.

Although MS-tree is similar to trie, there are important differences between them. Due to space limits, we illustrate the difference in Section IV-C of the full paper [15].

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 8). Figure 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 Figure 10) 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 σ_i is denoted as $Ins(\sigma_i)$ while deletion of an expired edge σ_j is denoted as $Del(\sigma_j)$.

Figure 10 illustrates the steps of handling each incoming edge based on the discussion in Section III. 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 Figure 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).

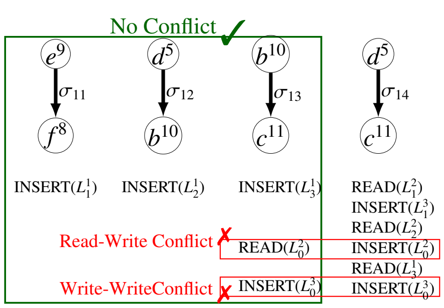


Fig. 10: Example of conflicts

Figure 10 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 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 III-B, 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 Figure 10. Thus, the concurrent execution requires a locking mechanism to guarantee the consistency.

Definition 11 (Streaming Consistency): Given 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.

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 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 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, $Ins(\sigma_{13})$ 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. Therefore, a finer-granularity locking strategy is desirable that allows higher DOC while guaranteeing streaming consistency. For example, in Figure 10, INSERT(L_0^2) in $Ins(\sigma_{13})$ should be processed before the same operation in $Ins(\sigma_{14})$; otherwise, it will lead to 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 (thread) only locks at most one item (i.e., "resource") 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 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.

C. 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 Figure 9. Assuming that a thread T_1 is reading partial matches of $\{\epsilon_6, \epsilon_5\}$, T_1 will backtrack from node n_1^2 (i.e., σ_3) to read n_1^1 (i.e., σ_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 $3 \times 3 \times 3 = 27$ 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 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 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 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.

Theorem 6: Parallel accesses with modified 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 [16]. 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

We propose a *cost model*-guided TC decomposition of query 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 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 ϵ_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 σ matches each edge ϵ_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 σ that matches one or more edges in query Q. The total expected number of join operations for $Ins(\sigma)$ 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.

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 $\{\epsilon_1$,...,$\epsilon_k\}$, according to the definition of TC-query, any *prefix* of the timing sequence constitutes a TC-subquery of Q^j. 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 $Q^i = \{\epsilon_1$,...,$\epsilon_j\}$ 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 $\{\epsilon_1$,...,ϵ_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 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.

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 [17]. We also present a case study in the full paper [15].

A. Datasets

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, we only report the experimental results over network dataset and social stream in this paper and that of wiki-talk are presented in the full paper [15]. The anonymous **network traffic data** contains about 500 millions communication records (edges) concerning about 2 million IP addresses (vertices). **The wiki-talk dataset** is from the Standford SNAP library [18] where a directed edge indicates that a user edit another user's talk page at a certain time point. This dataset contains 1,140,149 vertices and 7,833,140 edges. **Linked Stream Benchmark** [19] 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.

B. Query Generation

We generate query graphs by random walk over the data graph. For each subgraph g that is retrieved from data graph, we need to further generate the timing order. In fact, there is a full timing order between any two edges in g according to their inherent timestamps in the data graph. Hence, we can generate a subset of this full timing order to be that of g. We create a random permutation of g's edges and then for any two edges ϵ_i, $\epsilon_j \in E(g)$, we set $\epsilon_i \prec \epsilon_j$ if and only if (1) ϵ_i is before ϵ_j in the permutation and (2) the timestamp of ϵ_i in g is less than that of ϵ_j. The average selectivities of these queries are reported in Figure 25 of the full paper [15].

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

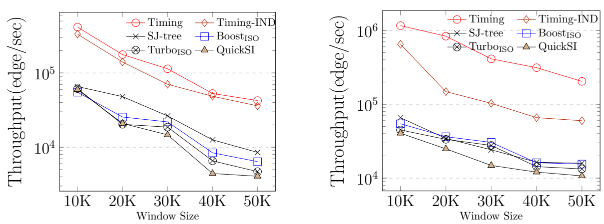
Since none of the existing works support concurrent execution, all codes (including ours) are run as a single thread; the evaluation of concurrency management is in Section VII-D. Our method, denoted as Timing, is compared with a number of related works. SJ-tree [1] is the closest work to ours. Since it does not handle the timing order constraints, we verify answers from SJ-tree posteriorly with the timing order constraints. 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 QuickSI [6], $\text{Turbo}_{\text{ISO}}$ [7], $\text{Boost}_{\text{ISO}}$[1] [8]. These methods are conducted over the

[1]We implement the $\text{Boost}_{\text{ISO}}$ by applying the speed-up strategy in [8] over $\text{Turbo}_{\text{ISO}}$, which is the state-of-the-art algorithm.

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.

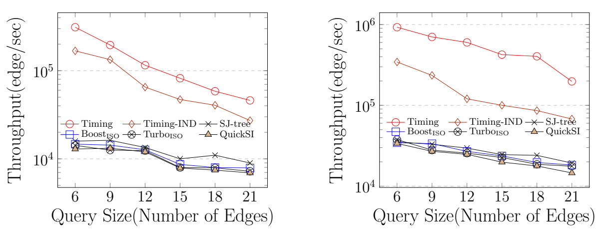
There are 5 different window sizes in our experiments: $10K$, $20K$, $30K$, $40K$ and $50K$ where each unit of the window size is the average time span between two consecutive arrivals of data edges in the dataset (i.e., the ratio of the total time span of whole dataset to the total number of data edges).

We evaluate the systems by varying window size $|W|$ and query size $|E(Q)|$. In Section VII-G of the full paper [15], 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.



(a) Network Flow (b) Social Stream

Fig. 11: Throughput over Different Window Size

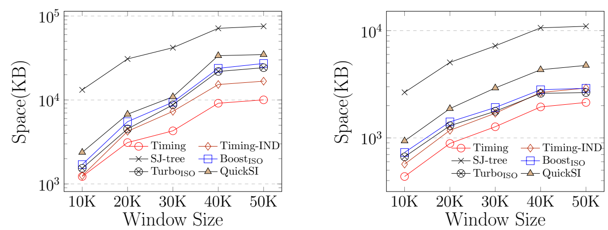


(a) Network Flow (b) Social Stream

Fig. 12: Throughput over Different Query Size

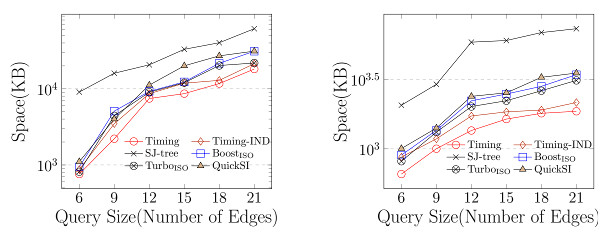
1) Time Efficiency Comparison: Figures 11-12 show that our method is clearly faster than other approaches over different window sizes and query sizes, respectively. The reason for the superior performance of our method lies in two aspects. First, our method can filter out lots of discardable partial 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 the space and maintenance cost, it is not as good as Timing, as shown in our experiments.

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 13-14. 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$_{\mathrm{ISO}}$ and Boost$_{\mathrm{ISO}}$ 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.



(a) Network Flow (b) Social Stream

Fig. 13: Space over Different Window Size



(a) Network Flow (b) Social Stream

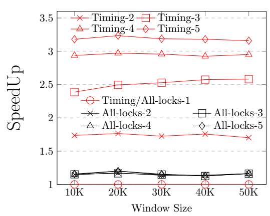
Fig. 14: Space over Different Query Size

D. Concurrency Evaluation

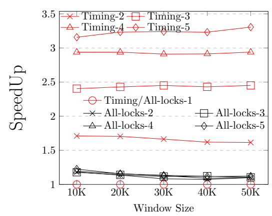
We evaluate the performance of our concurrency technique in this section by varying the number of threads running in parallel. We use **Timing-N** to differentiate different settings of parallel threads (N). We also implement, for comparison, a locking mechanism that requires a thread to obtain all locks before it is allowed to proceed (called **All-locks-N**). We present the speedup over single thread execution in Figures 15-16. 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. Figure 16 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.

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 $TCsub(Q)$ 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 17 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.

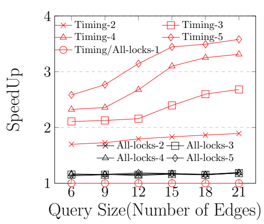


(a) Network Flow

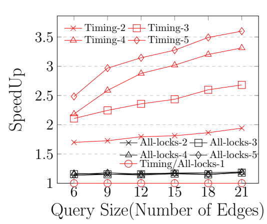


(b) Social Stream

Fig. 15: Speedup over Different Window Size

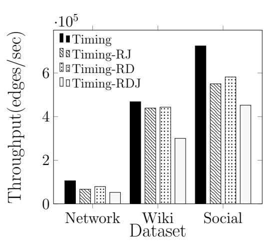


(a) Network Flow

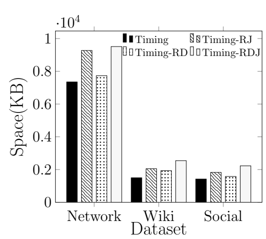


(b) Social Stream

Fig. 16: Speedup over Different Query Size



(a) Time Efficiency



(b) Space Efficiency

Fig. 17: Evaluating Optimizations

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.

Acknowledgment

This work was supported by The National Key Research and Development Program of China under grant 2018YFB1003504 and NSFC under grant 61622201 and 61532010. Ozsus research was supported in part by a grant from Natural Sciences and Engineering Research Council (NSERC) of Canada. Corresponding author: Lei Zou (zoulei@pku.edu.cn).

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/rp_DBIR_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. 413–424, 2014. [Online]. Available: http://www.vldb.org/pvldb/vol8/p413-ge.pdf
[15] "Full paper," http://arxiv.org/abs/1801.09240, 2018.
[16] A. Silberschatz and Z. Kedem, "Consistency in hierarchical database systems," *Journal of the ACM*, vol. 27, no. 1, pp. 72–80, 1980.
[17] "Code," https://github.com/pkumod/timingsubg.git, 2018.
[18] "Wikitalk," http://snap.stanford.edu/data/wiki-talk-temporal.html.
[19] "Lsbench code," https://code.google.com/archive/p/lsbench/, 2017.