LM-SRPQ: Efficiently Answering Regular Path Query in Streaming Graphs

Xiangyang Gou
The Chinese University of Hong Kong
xygou@se.cuhk.edu.hk

Xinyi Ye
Peking University
yexinyi@pku.edu.cn

Lei Zou
Peking University
zoulei@pku.edu.cn

Jeffrey Xu Yu
The Chinese University of Hong Kong
yu@se.cuhk.edu.hk

ABSTRACT

Regular path query (RPQ) is a basic operation for graph data analysis, and persistent RPQ in streaming graphs is a new-emerging research topic. In this paper, we propose a novel algorithm for persistent RPQ in streaming graphs, named LM-SRPQ. It solves persistent RPQ with a combination of intermediate result materialization and real-time graph traversal. Compared to prior art, it merges redundant storage and computation, thus gets higher memory and time efficiency. We carry out extensive experiments with both real-world and synthetic streaming graphs to evaluate its performance. Experiment results confirm its superiority compared to prior art in both memory and time efficiency.

PVLDB Reference Format:
Xiangyang Gou, Xinyi Ye, Lei Zou, and Jeffrey Xu Yu. LM-SRPQ: Efficiently Answering Regular Path Query in Streaming Graphs. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/StreamingTriangle/LM-SRPQ.

1 INTRODUCTION

Graph is an omnipresent data form for representation of large-scale entities and their relationships. It is used in various fields like biochemistry, social networks and knowledge graphs. Many graph-based applications require continuous updates and deal with **streaming graphs**. A streaming graph is an unbounded sequence of data items received from data sources. Each data item represents an edge between two vertices. Together these data items form a dynamic graph. For example, in Twitter, user communication can be organized as a streaming graph, where each data item represents an interaction (edge) between two users (vertex). Up to $12k$ new edges need to be processed per second in this streaming graph [15].

The unboundedness of streaming graphs means that the entire graph is not available to analysis algorithms. Additionally, many streaming graph applications focus on real-time analysis, and therefore focus on more recent data. These issues are usually addressed by using the sliding window model [10] to filter out old data items. A sliding window maintains data items in a recent time period, like 12 hours. When the window moves, older data that falls outside this interval is discarded and new data replaces it. Sliding window model is widely used in both research [13, 21, 24] and industrial applications [28].

In streaming graph applications, systems generally deal with **persistent queries** that are previously registered and continuously monitored. The answer set of a persistent query is continuously maintained and updated in real time as the streaming graph changes. For example, fraud detection can be specified as a persistent query continuously monitoring for the emergence of certain graph patterns. Persistent versions of many common graph queries have been studied, such as subgraph matching [17, 21], cycle detection[28], path navigation [24, 25] and triangle counting [13, 30].

In this paper, we focus on persistent Regular Path Query (RPQ for short) in streaming graphs with a sliding window model. A regular path query finds vertex pairs that are connected by a path satisfying a regular expression in a directed, edge-labeled multigraph. RPQ is an important path navigation operation in fields like social graphs and Semantic Web. It is also supported by many graph query languages such as SPARQL and Cypher.

RPQ in static graphs has been studied for decades [9, 19, 20, 23]. However, RPQ research in streaming graphs is in its infancy. Pacaci et.al. [24] are the first to define persistent RPQ over streaming graphs, and they propose a novel algorithm called RAPQ to solve this problem. This algorithm transforms RPQ problem into a reachability problem in a product graph, which is a cartesian product of the streaming graph and a query graph generated from the regular expression. In order to achieve persistent query, they materialize paths to all the successors of a qualified product graph node with a tree called Δ tree [1], and update Δ trees to find fresh reachable node pairs when the streaming graph is updated. In [25], they further combine persistent RPQ with persistent graph pattern matching and propose an algebra for complex queries in streaming graphs. RAPQ algorithm is also extended in [25], and the new algorithm is called S-PATH. Timestamps of regular paths are maintained in S-PATH to enable further analysis. An example of Δ trees in S-PATH is shown in Figure 2, which we will discuss in detail in Section 2.2.

S-PATH can efficiently process updates in the streaming graph but at the cost of high memory consumption. According to [5], 69% of organic RPQs (RPQs issued by browsers) in Wikidata query log are recursive, namely containing Kleene stars. For these recursive queries, Δ trees built in S-PATH have no depth constraint and can be very large. For example, according to our experiments with

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

[1]In the following sections, we call vertex in the product graph as "node", in order to distinguish them from "vertex" in the snapshot graph. Besides, the trees built in [24] are originally called spanning trees, we change their name to avoid confusion with the spanning tree in graph theory

StackOverflow, a real-world social network dataset, when answering RPQ a^*b^* in a sliding window with $180k$ edges and $59k$ vertices, there are $3.6k$ Δ trees with more than $21k$ nodes in S-PATH. The total number of tree nodes is $96M$, and the memory cost of S-PATH is more than 600 times larger than the original streaming graph. In real-world applications, we usually need to monitor multiple persistent queries at the same time, and the total memory cost will be multiple times higher. Such high memory consumption limits the scalability of the algorithm, especially when the application runs in embedded devices like hubs and routers.

In this paper, we decrease the size of the Δ tree forests while keeping a high update speed. We notice that there are many high-degree nodes appearing in multiple Δ trees, incurring multiple subtrees with the same root. The subtree with root v in Δ tree T contains a subset of v's successors, where the latest paths (paths with the largest timestamp between two nodes, as will be defined in Section 2.1) from the root of T to these successors pass v. Different subtrees with the same root v are not exactly the same, as they may contain different successor subsets, but they usually share common parts. We can replace these subtrees with one independent Δ tree, which contains all successors of v (as shown in Figure 3). In this case, we call v a landmark and the Δ tree built for it a landmark tree, or LM tree for short. When v is added into other Δ trees, we can find v's successors in its LM tree instead of building new subtrees. As we merge common parts of the subtrees, we can reduce memory usage. We hope to minimize the Δ tree forest in this manner and propose our LM-SRPQ algorithm (Landmark-based Streaming RPQ).

However, there are two major challenges in this method:

First, not every landmark reduces the Δ tree forest size, and it takes exponential cost to select an optimal landmark set that minimizes the Δ tree forest. For some product graph nodes v, if we select it as a landmark, the LM tree we build will be larger than the sum of subtrees rooted at v, resulting in increment in the forest size. Because each subtree only contains a subset of v's successors and is smaller than the LM tree which contains all the successors. Thus we need to trade off the benefit and cost carefully for each node. Moreover, selecting a node as a landmark will change the Δ tree forest, and cast an unpredictable influence on the benefit and cost of other nodes. As a result, there is no shortcut to find the optimal landmark set but to search in a solution space of $2^{|V_P|}$, where $|V_p|$ is the number of nodes in the product graph. However, as the streaming graph is continuously updated, we need to repeatedly redo the landmark selection with a fixed time interval. Therefore, we can afford limited time cost for each landmark selection in consideration of the time efficiency of the entire algorithm.

Second, the selection of landmarks will influence the update cost of the algorithm. When a Δ tree T contains a landmark v, there are some nodes reachable from the root of T but stored in the LM tree of v. In this case, we define a dependency relationship between T and the LM tree of v. As LM trees may also contain landmarks, such dependency relationship is recursive, and forms a dependency graph. In order to produce fresh RPQ results, we need to traverse this dependency graph in updates, as will be discussed in Section 3.1. The result of landmark selection influences the number of LM trees, and thus influences the size of this dependency graph and the update cost. Moreover, without optimization, the dependency graph traversal will become a performance bottleneck.

In summary, landmark selection needs to balance both the memory cost and the update cost, and should be finished in a reasonable time. Besides, we also need further optimization to control the cost of dependency graph traversal. In LM-SRPQ, We use a greedy algorithm in landmark selection, which tries to minimize size of the Δ tree forest while bounding the number of LM trees, in order to control size of the dependency graph and avoid high graph traversal cost. Besides, we also continuously maintain an additional index called TI map in each LM tree, which records timestamps of paths starting from the tree root. Based on these timestamps we propose a set of rules for pruning in dependency graph traversal.

We carry out extensive experiments in two real-world datasets and one synthetic dataset to evaluate the performance of our algorithm. The result shows that LM-SRPQ reduces the memory usage of prior art S-PATH by at most 40 times. Moreover, as common subtree merging reduces the Δ tree maintenance cost and efficient pruning reduces the dependency graph traversal cost, LM-SRPQ is at most 5 times faster than S-PATH.

In summary, we made the following contributions in this paper:

(1) We propose a novel algorithm for persistent RPQ in streaming graphs, named LM-SRPQ. It finds common subtrees in the Δ tree forest of prior art, and tries to minimize the memory usage by selectively merging these common subtrees. After subtree merging, it solves persistent RPQ by combining Δ tree maintenance and dependency graph traversal.
(2) To balance the time and memory cost of our algorithm, we propose a greedy landmark selection algorithm to balance the number of Δ trees and the size of the Δ tree forest, as well as an additional index that helps to prune the recursive traversal among Δ trees.
(3) We carry out extensive experiments to evaluate our algorithm, which confirm its superiority against prior art.

2 PRELIMINARIES

We formally define our problem in Section 2.1 and list frequently-used notations in Table 1. To better understand the motivation of our method, we briefly introduce S-PATH [25] in Section 2.2, which is the prior art for RPQ over streaming graphs in the literature.

2.1 Problem Definition

Definition 2.1. ***Graph.*** *A directed, edge-labeled graph is defined as* $G = (V, E, \Sigma, \phi)$, *where* V *is a vertex set,* $E \subset V \times V$ *is an edge set,* Σ *is a set of labels, and* $\phi : E \longrightarrow \Sigma$ *is an edge labeling function.*

Definition 2.2. ***Streaming Graph Tuple.*** *A streaming graph tuple is a triple* $sgt = (e, l, ts)$, *where* e *is a directed edge* $\overrightarrow{v_i, v_j}$ *from* v_i *to* v_j *with edge label* l *at timestamp* ts. *We call* ts *timestamp of edge* e, *denoted as* $e.ts$.

For ease of presentation, we abbreviate streaming graph tuples as *tuples* and use the terms "tuple" and "edge" interchangeably when the context is clear. Also, following the same assumption in [24] and [25], all tuples are generated by a single source and arrive in source timestamp order, which defines their ordering in the stream. More complicated scenarios, such as out-of-order delivery and multiple streaming sources, are left to future work.

Table 1: Notation Table

Notation	Meaning
W	Sliding window
β	Sliding interval
G_τ	Snapshot graph at time τ
$e.ts$ / $p.ts$	Timestamp of edge e / path p
$\phi(e)$ / $\phi(p)$	Label of edge e / path p
A_R	DFA built for regular expression R
$A_R.F$	Final state set in A_R
s_0 / s_f	Initial state / final state in DFA
$P(G, A)$	Product graph of graph G and DFA A
$\langle v_i, s_i \rangle$	Product graph node with vertex v_i and state s_i.
T_{v_i,s_i}	Δ tree with root node $\langle v_i, s_i \rangle$.
$T_{v_i,s_i}.\langle v_j, s_j \rangle.ts$	Timestamp of node $\langle v_j, s_j \rangle$ in tree T_{v_i,s_i}
$MaxTime(\langle v_i, s_i \rangle \rightarrow \langle v_j, s_j \rangle)$	The largest timestamp of the paths from $\langle v_i, s_i \rangle$ to $\langle v_j, s_j \rangle$ in the product graph
$T_{v_i,s_i}.TI(v_j, s_j)$	Timestamp of $\langle v_j, s_j \rangle$ in the TI map of tree T_{v_i,s_i}.

Definition 2.3. ***Streaming Graph.*** *A streaming graph is a sequence of streaming graph tuples $S = \{sgt_1, sgt_2, ...\}$ which arrives in the order of their timestamps.*

Obviously, it is impossible to process all streaming tuples due to the infinite volume of a streaming graph. In applications, we usually focus on the most recent tuples, which can be formalized with the sliding window model.

Definition 2.4. ***Sliding Window.*** *Let the current time point be τ. A sliding window W with time-scale length N and sliding interval $\beta(\beta >= 1)$ is a set of tuples whose timestamps are in range $(\lfloor \frac{\tau}{\beta} \rfloor - N, \tau]$. Tuples in the sliding window W are called* active*, while ones out of this set are considered to be* expired.

The sliding interval β allows us to handle edge expiration in a lazy manner, *i.e.*, removing expired tuples in a batch in every β time units [26]. On the other hand, we process new tuples in real time to produce fresh results. Furthermore, the graph induced by active tuples (edges) in the current sliding window W is called a **snapshot graph** (denoted as G_τ). Note that there may be multiple tuples with the same edge e in the sliding window. Those with the same label are combined into one edge in the snapshot graph, and the timestamp of this edge, denoted as $e.ts$, is the largest timestamp among them. Those with different labels are considered as parallel edges with the same endpoints but different labels.

Example 1. *A streaming graph is shown in Figure 1 (a), the sliding window size is set to 10. Timestamps are shown on the top of edges. The snapshot graph at current time $\tau = 12$ is shown in Figure 1 (b). Note that there may be multiple tuples arriving at one time point (like time 12), or no tuple arriving at some time points (like time 7). The sliding interval is $\beta = 2$, namely we only delete tuples from the sliding window at even timestamps.*

Usually, streaming graph systems deal with **persistent queries**. These queries are previously registered and continuously monitored as the snapshot graph changes due to arrival of new tuples and expiration of old ones. In this paper, we focus on persistent regular path queries over streaming graphs.

Definition 2.5. ***Regular Expression.*** *A regular expression R over an alphabet Σ is recursively defined as $R ::= \epsilon | a | R \circ R | R + R | R^* | R^+$ where*

1. *ϵ is the empty string.*
2. *a is a character in the alphabet.*
3. *$\circ$ means concatenation function.*
4. *$+$ means alternation function, namely OR operation.*
5. *R^* means Kleene star. R can repeat 0 or multiple times.*
6. *R^+ means 1 or more repetitions of R.*

$L(R)$ is a set of strings that can be described by regular expression R.

Definition 2.6. ***Regular Path Query (RPQ).*** *A regular path query Q_R over a graph G is to find all vertex pairs (v_i, v_j), where there is at least one path p from v_i to v_j in G and $\phi(p) \in L(R)$. The path label $\phi(p)$ is the concatenation of the edge labels along path p and $L(R)$ denotes all strings described by regular expression R. We denote the query result of Q_R as $Q_R(G)$.*

Consider a regular expression $R = (a \circ b)^*$ and the RPQ query Q_R. The current snapshot graph G_τ is given in Figure 1(b) and the path $p = v_1 \rightarrow v_4 \rightarrow v_5 \rightarrow v_2 \rightarrow v_4$ (marked in red) conforms to R due to $\phi(p) \in L(R)$. Thus, (v_1, v_4) is a part of the result set $Q_R(G_\tau)$.

Due to dynamic updates of the snapshot graph, we need to continuously maintain the result set of a given RPQ query Q_R. Besides, we also require to maintain a timestamp for each vertex pair (v_i, v_j) in the result set, which enables more streaming graph applications. Specifically, we have the following definition about the timestamps of paths and vertex pairs.

Definition 2.7. ***Timestamp of Path and Vertex Pair.*** *For any path p in the snapshot graph G_τ, the timestamp of p, denoted as $p.ts$, is defined as the minimum edge timestamp along the path, i.e., $p.ts = Min\{e.ts | e \in p\}$, where $e.ts$ is the timestamp associated with e (defined in Definition 2.2).*

Given a vertex pair (v_i, v_j) in the result set $Q_R(G_\tau)$ of an RPQ Q_R, let $PS = \{p_1, ..., p_m\}$ denotes the set of distinct paths from v_i to v_j in G_τ satisfying regular expression R. The timestamp of vertex pair (v_i, v_j) is defined as the maximum timestamp of all paths in PS, i.e., $(v_i, v_j).ts = Max\{p.ts | p \in PS\}$.

Our problem definitions are similar to [24], except that we maintain timestamps for vertex pairs in the result set. These timestamps reflect the life span of vertex pairs and enable us to use a direct approach to process expiration. We can directly find and delete all expired vertex pairs which have timestamps smaller than $\tau - N$. For these vertex pairs, all the regular paths connecting them have expired. These timestamps may also be used for further analysis in different applications. For example, different applications can customize their own sliding intervals by filtering the result set according to these timestamps.

The streaming graph algebra in [25] also maintains time information for query results, and it applies to streaming RPQ. But in [25], tuples, paths and vertex pairs all have validity intervals $(ts, exp]$ rather than timestamps. This model is an extension of the sliding window model, where tuples can have different life spans. In this paper, we focus on the sliding window model, in order to keep conciseness of the presentation. Besides, sliding window model is also the most widely used validity model for streaming graphs. We leave more complicated scenarios to future work.

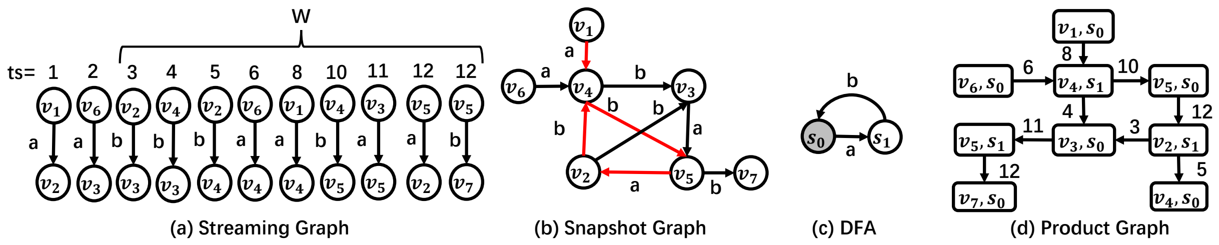


Figure 1: Streaming graph, snapshot graph, DFA for $(a \circ b)^*$ and the corresponding product graph

In the following sections, we solve the streaming RPQ problem with deterministic finite automaton and product graph:

Definition 2.8. ***Deterministic Finite Automaton (DFA).*** *Given a regular expression R, a deterministic finite automaton for R is $A = (S, \Sigma, \delta, s_0, F)$, where S is a set of states, Σ is the alphabet of R, δ is a transition function which belongs to $S \times \Sigma \longrightarrow S$, s_0 is an initial state and F is a set of finite states. δ^* is extended from δ and is recursively defined as $\delta^*(s, \omega \circ a) = \delta(\delta^*(s, \omega), a)$ where ω is a string made up of characters in Σ and $s \in S$, $a \in \Sigma$. A string ω can be accepted by A if $\delta^*(s_0, \omega) \in F$, and $\omega \in L(R)$ if and only if it can be accepted by A.*

With a regular expression R, we can create an NDFA with Thompson's construction algorithm [31] and transform it into a DFA using Hopcroft's algorithm [16]. We denote the DFA generated from regular expression R as A_R, and denote states in F as s_f by default.

Definition 2.9. ***Product Graph.*** *Given a graph $G = (V, E, \Sigma, \phi)$ and a DFA $A = (S, \Sigma, \delta, s_0, F)$, the corresponding product graph $P(G, A) = (V_P, E_P, \Sigma, \phi_P)$ is defined as: (1) $V_P = V \times S$. (2) $E_p \subset V_P \times V_P$, and $\overrightarrow{\langle v_i, s_i\rangle, \langle v_j, s_j\rangle}$ belongs to E_p if $\overrightarrow{v_i, v_j} \in E$ and $\delta(s_i, \phi(\overrightarrow{u, v})) = s_j$. (3) $\phi_P(\overrightarrow{\langle v_i, s_i\rangle, \langle v_j, s_j\rangle}) = \phi(\overrightarrow{v_i, v_j})$.*

If the product graph is built upon a snapshot graph G_τ, each edge $e = \overrightarrow{\langle v_i, s_i\rangle, \langle v_j, s_j\rangle}$ in the product graph also has a timestamp equal to the timestamp of $\overrightarrow{v_i, v_j}$ in G_τ. We can define paths and path timestamps in the product graph similar to the snapshot graph. We call path p a **latest path** if it has the largest timestamp among the paths connecting its source node $\langle v_i, s_i\rangle$ and destination node $\langle v_j, s_j\rangle$. And we denote its timestamp as $MaxTime(\langle v_i, s_i\rangle \rightarrow \langle v_j, s_j\rangle)$.

Given a regular expression $R = (a \circ b)^*$, the corresponding DFA is shown in Figure 1 (c). Considering the snapshot graph G_τ, the product graph $P(G_\tau, A)$ is given in Figure 1 (d). We add timestamps to the edges in the product graph, in order to help the understanding of examples in the following section. .

2.2 Existing Solution:S-PATH Algorithm

In this section, we introduce S-PATH algorithm proposed in [25]. It is an extension of RAPQ algorithm proposed in [24], which adds time information to the result set and adopts direct expiration approach. Note that we modify S-PATH algorithm by changing validity intervals of edges and paths to timestamps, in order to fit with our problem definition (as discussed in Section 2.1). S-PATH algorithm is based on the following theorem [24]:

Theorem 2.1. *There is a path p from v_i to v_j in the snapshot graph G_τ conforming to a regular expression R if and only if there is a path from $\langle v_i, s_0\rangle$ to $\langle v_j, s_f\rangle$ in the product graph $P(G_\tau, A_R)$ with the same timestamp.*

According to this theorem, S-PATH answers a regular query Q_R by finding connected node pairs $\langle v_i, s_0\rangle$ and $\langle v_j, s_f\rangle$ in the product graph. S-PATH materializes intermediate results in the traversal of $P(G_\tau, A_R)$ with Δ trees.

Definition 2.10. Δ ***Tree Index****: Given a regular expression R and a snapshot graph G_τ, S-PATH maintains Δ trees T_{v_i,s_0} which has initial-state root $\langle v_i, s_0\rangle$ and is built with following rules:*

1. *A node $\langle v_j, s_j\rangle$ is in T_{v_i,s_0} if there is a path from $\langle v_i, s_0\rangle$ to $\langle v_j, s_j\rangle$ in the product graph $P(G_\tau, A_R)$, or equivalently, there is path p from v_i to v_j in G_τ where $\delta^*(s_0, \phi(p)) = s_j$.*
2. *The path p from root $\langle v_i, s_0\rangle$ to node $\langle v_j, s_j\rangle$ in T_{v_i,s_0} is the latest path between them in $P(G_\tau, A_R)$, and node $\langle v_j, s_j\rangle$ is attached with a timestamp $T_{v_i,s_0}.\langle v_j, s_j\rangle.ts = p.ts$. We simplify $T_{v_i,s_0}.\langle v_j, s_j\rangle.ts$ as $\langle v_j, s_j\rangle.ts$ when there is no ambiguity.*

Result set RS contains all tuples $((v_i, v_j), ts)$ if there is node $\langle v_j, s_f\rangle$ in the Δ tree T_{v_i,s_0} with timestamp ts. Those Δ trees that only contain a root node, as well as self-join results are omitted (for example, results like (v_i, v_i) are omitted for query a^*).

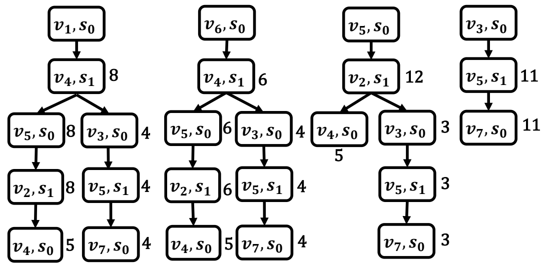


Figure 2: Δ tree index in S-PATH

Example 2. *Figure 2 shows the Δ tree index for the snapshot graph and regular expression in Figure 1 (b) and (c). Node $\langle v_3, s_0\rangle \in T_{v_1,s_0}$ and $T_{v_1,s_0}.\langle v_3, s_0\rangle.ts = 4$, because there is a path $p = \langle v_1, s_0\rangle \rightarrow \langle v_4, s_1\rangle \rightarrow \langle v_3, s_0\rangle$ in the product graph with timestamp 4. Besides, p is corresponding to the snapshot graph path $v_1 \rightarrow v_4 \rightarrow v_3$ with label ab, where $\delta^*(s_0, ab) = s_0$. Note that there is another path from $\langle v_1, s_0\rangle$ to $\langle v_3, s_0\rangle$ in the product graph, namely $p' = \langle v_1, s_0\rangle \rightarrow \langle v_4, s_1\rangle \rightarrow \langle v_5, s_0\rangle \rightarrow \langle v_2, s_1\rangle \rightarrow \langle v_3, s_0\rangle$. But this path has a smaller timestamp 3. Therefore, Δ tree index only stores path p rather than p'.*

Update: Algorithm 1 shows the steps to update Δ tree index and the result set upon receiving a new tuple $sgt = (e_b = \overrightarrow{v_b, v_d}, l, ts)$. S-PATH finds all state pairs (s_b, s_d) form DFA A_R where $\delta(s_b, l) = s_d$.

Algorithm 1: Processing new tuples in Δ tree index

```
Input: new tuple $sgt = (e_b = \overrightarrow{v_b, v_d}, l, ts)$
Output: updated $\Delta$ trees and result set $RS$
1  forall $(s_b, s_d) \in A_R.S$ where $\delta(s_b, l) = s_d$ do
2      if $s_b = s_0$ and $T_{v_b,s_0} \notin \Delta$ then
3          Add $T_{v_b,s_0}$ with root $\langle v_b, s_0 \rangle$ into $\Delta$, $\langle v_b, s_0 \rangle.ts = INF$.
4      forall $T_{v_x,s_0} \in \Delta$ where $\langle v_b, s_b \rangle \in T_{v_x,s_0}$ do
5          queue $Q.push(\langle v_b, s_b \rangle, \langle v_d, s_d \rangle, e_b)$.
6          while $!Q.empty$ do
7              $\langle v_i, s_i \rangle, \langle v_j, s_j \rangle, e = Q.top()$.
8              $Q.pop()$.
9              if $\langle v_j, s_j \rangle \notin T_{v_x,s_0}$ then
10                 Add $\langle v_j, s_j \rangle$ into $T_{v_x,s_0}$ with parent $\langle v_i, s_i \rangle$.
11                 $\langle v_j, s_j \rangle.ts = Min(\langle v_i, s_i \rangle.ts, e.ts)$
12             else if $\langle v_j, s_j \rangle.ts < Min(\langle v_i, s_i \rangle.ts, e.ts)$ then
13                 Set parent of $\langle v_j, s_j \rangle$ to $\langle v_i, s_i \rangle$
14                 $\langle v_j, s_j \rangle.ts = Min(\langle v_i, s_i \rangle.ts, e.ts)$
15             else
16                 Continue
17             if $s_j \in A_R.F$ then
18                 $UpdateMap(RS, (v_x, v_j), \langle v_j, s_j \rangle.ts)$
19             forall $\langle v_q, s_q \rangle$ where $e' = \overrightarrow{v_j, v_q} \in G_\tau$ and
                 $\delta(s_j, \phi(e')) = s_q$ do
20                 $Q.push(\langle v_j, s_j \rangle, \langle v_q, s_q \rangle, e')$
```

For each state pair, if $s_b = s_0$ and $T_{v_b,s_0} \notin \Delta$, it builds a new Δ tree with root $\langle v_b, s_0 \rangle$, (line 2-3). Then it finds all Δ trees containing node $\langle v_b, s_b \rangle$. It expands each tree T_{v_x,s_0} with a breadth-first search (BFS) in the product graph [2], starting from $\langle v_d, s_d \rangle$. For each node $\langle v_j, s_j \rangle$ it finds in BFS from precursor $\langle v_i, s_i \rangle$ through edge e, there is a path p from $\langle v_x, s_0 \rangle$ to $\langle v_j, s_j \rangle$, which is the concatenation of the path from $\langle v_x, s_0 \rangle$ to $\langle v_i, s_i \rangle$ in T_{v_x,s_0} and e. And $p.ts = Min(\langle v_i, s_i \rangle.ts, e.ts)$. There are 3 cases:

(1) If $\langle v_j, s_j \rangle \notin T_{v_x,s_0}$, S-PATH adds it into T_{v_x,s_0} with parent $\langle v_i, s_i \rangle$ and timestamp $p.ts$ (line 9-11).
(2) If $\langle v_j, s_j \rangle \in T_{v_x,s_0}$ but $\langle v_j, s_j \rangle.ts < p.ts$, it means a new path with larger timestamp is found. S-PATH sets the parent of $\langle v_j, s_j \rangle$ to $\langle v_i, s_i \rangle$, and $\langle v_j, s_j \rangle.ts$ to $p.ts$ (line 12-14).
(3) If $\langle v_j, s_j \rangle \in T_{v_x,s_0}$ and $\langle v_j, s_j \rangle.ts \geqslant p.ts$, S-PATH prunes this BFS branch (line 15-16).

Besides, if s_j is a final state, S-PATH updates the result set RS with function $UpdateMap(\cdot)$ (line 17-18). This function sets $(v_x, v_j).ts$ to $\langle v_j, s_j \rangle.ts$ if $(v_x, v_j) \notin RS$ or $(v_x, v_j).ts < \langle v_j, s_j \rangle.ts$.

Expire: S-PATH carries out expire procedure at the end of every sliding interval. It first deletes all the outdated edges from the snapshot graph. Then it deletes all the tree nodes with timestamps smaller than $\tau - N$ in Δ-tree index, where τ is the current time. These expired nodes can be located in the subtrees rooted at node $\langle v_j, s_j \rangle$, where there is an expired edge $\overrightarrow{v_i, v_j}$ and a state s_i so that $\delta(v_i, \phi(\overrightarrow{v_i, v_j})) = s_j$. At last, it deletes all result tuples with timestamps smaller than $\tau - N$ in the result set RS.

[2] [25] uses depth-first search (DFS) here. But according to our experiments, BFS is much faster. Because in BFS we can find multiple paths to a node before we further traverse to its successors, and choose the path with the largest timestamp to extend. While in DFS we may traverse the entire search branch following a not latest path, resulting in many vain updates. Therefore, we use BFS in both S-PATH and LM-SRPQ. Besides, we do not actually store the product graph, and traverse it by simultaneously traversing the snapshot graph and the DFA according to Definition 2.9.

Drawback: By materializing the latest paths in the product graph, S-PATH achieves high update speed. However, as a cost of materialization, the memory consumption of S-PATH is high. We notice that there are many common subtrees with the same root in Δ tree forests of S-PATH. For example, In Figure 2, the subtrees rooted at node $\langle v_4, s_1 \rangle$ are the same in both T_{v_1,s_0} and T_{v_6,s_0}. We can reduce memory usage by merging these common subtrees into independent Δ trees. Moreover, by reducing the forest size, we also reduce the maintenance cost and further improve the update speed.

3 LM-SRPQ ALGORITHM

In this section, we propose LM-SRPQ algorithm. Our idea is to merge common subtrees in the Δ tree forest of S-PATH. To be specific, we select a group of nodes as **landmarks**. For each landmark, there are multiple subtrees rooted at it in the Δ tree forest. We replace these subtrees with an independent Δ tree, which is built according to Definition 2.10. We call this tree a **landmark tree**, or **LM tree** for short. We call the original Δ trees in S-PATH as **normal trees**. Word "Δ trees" is used to denote the union of these two kinds. Note that the roots of normal trees always have state s_0, but it is not necessary for LM trees.

Example 3. *Figure 3 shows an example of mergeing subtrees. In this example, we replace the subtrees rooted at $\langle v_2, s_1 \rangle$ (colored in red) in the Δ tree forest of Figure 2 with an LM tree (colored with blue).*

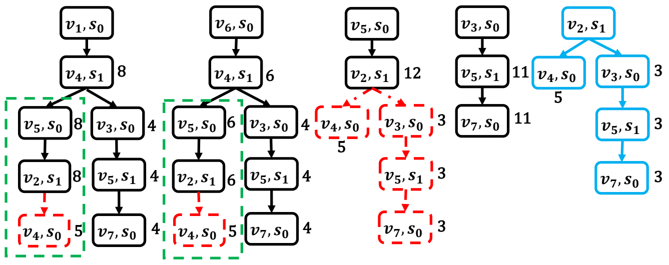


Figure 3: Merge subtrees in Δ tree index

In S-PATH, the subtree of $\langle v_j, s_j \rangle$ is built in T_{v_i,s_0} so that for each node $\langle v_q, s_q \rangle$ is this subtree, we can find latest paths p from root $\langle v_i, s_0 \rangle$ to it passing landmark $\langle v_j, s_j \rangle$. Besides this approach, we can also find such p by concatenating the latest path from $\langle v_i, s_0 \rangle$ to $\langle v_j, s_j \rangle$ with the latest path from $\langle v_j, s_j \rangle$ to $\langle v_q, s_q \rangle$, and the latter can be found in the LM tree. This is the foundation of our subtree merging. For example, in Figure 3, we can find the latest path from $\langle v_5, s_0 \rangle$ to $\langle v_7, s_0 \rangle$ by concatenating $\overrightarrow{\langle v_5, s_0 \rangle, \langle v_2, s_1 \rangle}$ in T_{v_5,s_0} with the path from $\langle v_2, s_1 \rangle$ to $\langle v_7, s_0 \rangle$ in T_{v_2,s_1}.

By replacing the subtrees of a landmark with the LM tree, we merge the common part of these subtrees and reduce memory usage. However, there are two problems in such subtree merging:

First, it takes exponential cost to find an optimal landmark set which minimizes the Δ tree forest, and we cannot afford such high cost. When selecting a node $\langle v_j, s_j \rangle$ as a landmark, we build a new LM tree and omit the subtrees rooted at $\langle v_j, s_j \rangle$ in other Δ trees. The former is the cost while the latter is the benefit brought by this landmark selection. The cost may be higher than the benefit, because a subtree rooted at $\langle v_j, s_j \rangle$ does not contain all its successors, and is smaller than the LM tree. For example, in Figure 3, node $\langle v_3, s_0 \rangle$ is reachable from $\langle v_2, s_1 \rangle$, but it is not in the subtree of $\langle v_2, s_1 \rangle$ in T_{v_1,s_0}. Because path $\langle v_1, s_0 \rangle \rightarrow \langle v_4, s_1 \rangle \rightarrow \langle v_3, s_0 \rangle$ has a larger timestamp than the path passing $\langle v_2, s_1 \rangle$. Therefore, we need to trade off the cost and benefit of each node. Moreover, selecting a node as a landmark will change the forest, and cast an unpredictable influence on the benefit and cost of other nodes. For example, in Figure 3, when we only select $\langle v_2, s_1 \rangle$ as a landmark, the benefit (red nodes) is larger than the cost (blue nodes). However, if we first select $\langle v_5, s_0 \rangle$ as a landmark and omit the subtrees marked in the green box. Then we will find the benefit of selecting $\langle v_2, s_1 \rangle$ is only to omit the red nodes in T_{v_5,s_0}, and is smaller than the cost of building T_{v_2,s_1}. As a result, we have no shortcut to find an optimal landmark set but to search in a solution space with size $2^{|V_p|}$, where $|V_p|$ is the number of nodes in the product graph. Such cost is unacceptable, especially when we need to redo landmark selection occasionally in order to keep up with the updated snapshot graph. We have to resort to a greedy algorithm.

Second, the landmark selection also influences the update cost, and may slow down the entire algorithm. We allow LM trees to contain other landmarks. As a result, we need to recursively visit LM trees in update, in order to find all the reachable nodes of a landmark or propagate the update in an LM tree T_{v_j,s_j} to all Δ trees that can reach $\langle v_j, s_j \rangle$. Such recursive visit can be formalized as a traversal in a dependency graph made up of Δ trees, as will be defined in Section 3.1. The result of landmark selection influences the size of the dependency graph, and thus influences the update cost. Moreover, we need to find latest paths in the dependency graph traversal and may revisit the same Δ tree multiple times. Thus its cost is considerable and may slow down the entire algorithm.

Therefore, we should carefully design a greedy algorithm to select a sub-optimal landmark set, and bound the dependency graph size at the same time. Besides, we need an efficient update algorithm to control the dependency graph traversal cost. We propose LM-SRPQ (Landmark-based Streaming RPQ) to solve these two problems. In LM-SRPQ, we carry out a landmark selection algorithm with a fixed time interval. This greedy algorithm finds a bounded-size candidate set in a heuristics method, which prioritizes nodes with large Δ trees. Then it scans all the candidates in descending order of approximated Δ tree size, and trades off the benefit and cost for each of them to produce the landmark set. Given the landmark set, LM-SRPQ materializes paths with no landmarks (called local paths) in Δ trees, and traverses the dependency graph to assemble these local paths. In order to avoid high traversal cost, we also continuously maintain an additional index called TI map in each LM tree, which records timestamps of all the latest paths starting from the tree root. Based on these timestamps we propose a set of rules for pruning in the graph traversal.

In the following sections, we first introduce the update algorithm of LM-SRPQ given a set of landmarks in Section 3.1. We discuss the dependency graph in detail in this section, which helps to understand the influence of landmark selection on the update cost. Then in Section 3.2, we introduce the landmark selection algorithm, which considers both the dependency graph size and the Δ tree forest size. In Section 3.3, we further decrease the dependency graph traversal cost with TI map.

3.1 Basic Update Algorithm

In this section, we suppose there is a given set of landmarks, and introduce the basic update algorithm of LM-SRPQ.

Given a group of landmarks, we build LM trees for them and build normal trees for product graph nodes with initial state s_0. Each Δ tree is built similarly to S-PATH, but the subtrees rooted at landmarks are omitted. In this case, there exists a dependency relationship among Δ trees, which is defined as follows:

Definition 3.1. *Δ tree T_{v_i,s_i} is said to* ***depend*** *on LM tree T_{v_j,s_j} if T_{v_i,s_i} contains landmark $\langle v_j, s_j \rangle$. The timestamp of the dependency relationship is the timestamp of node $\langle v_j, s_j \rangle$ in T_{v_i,s_i}.*

T_{v_i,s_i} can be either an LM tree or a normal tree. Such dependency relationship among Δ trees forms a dependency graph that changes along with the streaming graph. In this dependency graph, we can define paths similarly to the snapshot graph and the product graph, we call these paths **dependency paths**.

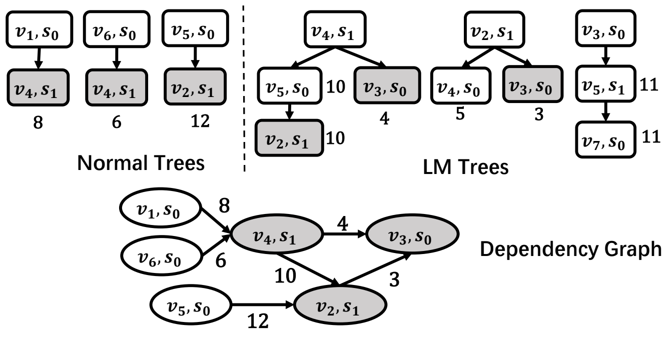


Figure 4: Δ trees and dependency graph in LM-SRPQ

Example 4. *An example of the Δ trees and dependency graph is shown in Figure 4. They are built for the snapshot graph and regular expression in Figure 1 (b) and (c). The landmark set contains $\langle v_4, s_1 \rangle$, $\langle v_2, s_1 \rangle$ and $\langle v_3, s_0 \rangle$. We mark the landmark nodes in Δ trees and LM trees in the dependency graph in shadow.*

As we maintain Δ trees with Algorithm 1, the path from root $\langle v_i, s_i \rangle$ to node $\langle v_j, s_j \rangle$ in T_{v_i,s_i} has the largest timestamp among the paths which connect $\langle v_i, s_i \rangle$ with $\langle v_j, s_j \rangle$ and contain no landmarks. We call it a **local path**. Each dependency edge between two Δ trees represents a local path connecting two tree roots, and each dependency path dp represents a product graph path, which is a concatenation of local paths. For example, in Figure 4, dependency edge $\overrightarrow{T_{v_4,s_1}, T_{v_2,s_1}}$ represents local path $p = \langle v_4, s_1 \rangle \rightarrow \langle v_5, s_0 \rangle \rightarrow \langle v_2, s_1 \rangle$ with timestamp 10. And dependency path $T_{v_4,s_1} \rightarrow T_{v_2,s_1} \rightarrow T_{v_3,s_0}$ is corresponding to the concatenation of p and local path $p' = \overrightarrow{\langle v_2, s_1 \rangle, \langle v_3, s_0 \rangle}$. In the following sections, with "concatenating dependency path dp with a local path p". we mean concatenating the product graph path which dp represents with p. The update in LM-SRPQ is based on the following theorem:

Theorem 3.1. *Suppose $\langle v_i, s_i \rangle$ is the root of a Δ tree T_{v_i,s_i} in LM-SRPQ and $\langle v_j, s_j \rangle$ is another node in the product graph. Then*

$$\begin{aligned} &MaxTime(\langle v_i, s_i \rangle \rightarrow \langle v_j, s_j \rangle) \\ =&Max\{Min(dp.ts, T_{v_q,s_q}.\langle v_j, s_j \rangle.ts) | dp \in PS\} \end{aligned} \tag{1}$$

PS is the set of dependency paths connecting T_{v_i,s_i} with any Δ tree T_{v_q,s_q} which contains node $\langle v_j, s_j \rangle$. Note that dp can be an empty path, namely $T_{v_q,s_q} = T_{v_i,s_i}$.

Proof. We divide the paths from $\langle v_i, s_i \rangle$ to $\langle v_j, s_j \rangle$ in the product graph into multiple groups:

Paths in the first group contain no landmarks. They are local paths around $\langle v_i, s_i \rangle$. When we update the Δ trees with algorithm 1, we store the latest local paths between the root and each node in the Δ tree. Therefore, the largest path timestamp in this group is equal to the timestamp of node $\langle v_j, s_j \rangle$ in T_{v_i,s_i}.

The remaining paths are divided into different groups according to the landmarks in them. Consider the group with landmark sequence $\{L_c | 0 \leqslant c \leqslant n\}$ where $L_n = \langle v_q, s_q \rangle$. Suppose the latest path in this group is p. We can split it into slices $\{p[c] = [H_c, H_{c+1}] | 0 \leqslant c \leqslant n+1\}$ where $H_0 = \langle v_i, s_i \rangle$, $H_{n+2} = \langle v_j, v_j \rangle$ and $H_c = L_{c-1} (1 \leqslant c \leqslant n+1)$. We have $p.ts = Min\{p[c].ts\}$. Each slice $p[c]$ must be the latest local path between H_c and H_{c+1}. Otherwise, by changing it to the latest local path, the timestamp of p will not get smaller. Timestamp of the latest local path between H_c and H_{c+1} is just the timestamp of H_{c+1} in the Δ tree of H_c, and when $c \leqslant n$, it is also the timestamp of the dependency edge between the Δ tree of H_c and H_{c+1}. Therefore, we have $Min\{p[c].ts | 0 \leqslant c \leqslant n\} = dp.ts$, where dp is the dependency path connecting T_{v_i,s_i} with the Δ tree of $L_n = \langle v_q, s_q \rangle$ and passing landmarks $\{L_c | 0 \leqslant c \leqslant n-1\}$. Besides, $p[n+1].ts = T_{v_q,s_q}.\langle v_j, s_j \rangle.ts$. Therefore, we have $p.ts = Min(dp.ts, T_{v_q,s_q}.\langle v_j, s_j \rangle.ts)$.

Combining these groups, we can get the above theorem. □

According to Theorem 3.1, we can maintain $MaxTime(\langle v_i, s_0 \rangle \rightarrow \langle v_j, s_f \rangle)$ and answer RPQ by concatenating dependency paths from T_{v_i,s_0} with local paths in the destination Δ trees. When a new tuple arrives, we can update the result set by finding new combinations of dependency paths and local paths.

To get new local paths, we update all Δ trees with Algorithm 1, but stop BFS at landmarks. The result set is updated in the same approach as Algorithm 1. New dependency edges are also generated in this procedure. When landmark $\langle v_j, s_j \rangle$ is added into T_{v_i,s_i}, there is a new dependency edge from T_{v_i,s_i} to T_{v_j,s_j}.

To get new dependency paths, we collect the new dependency edges generated in the Δ tree update. For each new dependency edge $de = \overrightarrow{T_{v_i,s_i}, T_{v_j,s_j}}$, we perform a traversal in the dependency graph to find the new dependency paths. We search forward from T_{v_j,s_j} to find the paths to its successors, and search backward from T_{v_i,s_i} to find the paths from its precursors to it. Note that we only need the precursors of T_{v_j,s_j} whose root has an initial state s_0. By concatenating the precursor paths, the new edge and the successor paths, we can get the new dependency paths. There may be multiple new dependency paths connecting two Δ trees T_{v_x,s_0} and T_{v_q,s_q}. We only need to find the one with the largest timestamp. Besides, there may be circles in the dependency graph, and we need to prune loops in these circles. Furthermore, the new dependency edge de should also be included in the new paths if $s_i = s_0$.

Then we concatenate above new paths and update result set RS:

(1) **Concatenate new dependency paths with local paths**. For each new dependency path dp connecting T_{v_i,s_0} with T_{v_j,s_j}, we scan all the final-state nodes in T_{v_j,s_j}. For each node $\langle v_q, s_f \rangle$, we compute $ts = Min(dp.ts, T_{v_j,s_j}.\langle v_q, s_f \rangle.ts)$. If vertex pair $(v_i, v_q) \notin RS$ or $(v_i, v_q).ts < ts$, we set $(v_i, v_q).ts = ts$.

(2) **Concatenate new local paths with dependency paths**. When new local paths to final-state nodes $\langle v_q, s_f \rangle$ are built in T_{v_j,s_j}, we perform a backward search from T_{v_j,s_j} to find its precursor trees T_{v_i,s_0}. Suppose the latest dependency path from T_{v_i,s_0} to T_{v_j,s_j} is dp, we compute $ts = Min(dp.ts, T_{v_j,s_j}.\langle v_q, s_f \rangle.ts)$. If vertex pair $(v_i, v_q) \notin RS$ or $(v_i, v_q).ts < ts$, we set $(v_i, v_q).ts = ts$.

Compared to S-PATH, LM-SRPQ merges common subtrees into independent LM trees, and thus omits a lot of redundant storage. However, traversal in the dependency graph may become a bottleneck of the algorithm. Without materialization like local paths, we need to build new dependency paths from scratch in each update. In this procedure, we may visit the same Δ tree $O(|E_d|)$ times from different dependency paths before we find the latest one, where $|E_d|$ is the number of dependency edges. To control this traversal cost, we have to bound the dependency graph size, and try to minimize the Δ tree forest size under such constraint.

3.2 Landmark Selection

Based on the discussion in Section 3.1, when selecting landmarks, we need to consider both the Δ tree forest size and the dependency graph size. Besides, we need to redo landmark selection with a fixed time interval (usually equal to the sliding interval) to keep up with the updated snapshot graph. Therefore the landmark selection can not consume too much time. Otherwise, it will slow down the entire algorithm. As the search space of selecting a landmark set is exponential, we cannot select an optimal solution in a reasonable time. Therefore, we propose a greedy algorithm to select landmarks.

First, we use a heuristic method to select landmark candidates. It both bounds the dependency graph size and bounds the time of landmark selection. In candidate selection, we prioritize nodes whose estimated Δ tree size is large. Because these nodes have a high probability to be roots of large subtrees in the Δ tree forest, and we can obtain high benefit by merging these subtrees. To be specific, we first filter out nodes that appear in less than 2 Δ trees. Then we assign each remaining node a score, which is an estimation of its Δ tree size. We sort the remaining nodes according to their scores, and use the top ρ percent as candidates.

The score of $\langle v_i, s_i \rangle$ is computed with the estimated width and depth of its Δ tree. The width is estimated as the degree of $\langle v_i, s_i \rangle$, namely the number of edges in the snapshot graph which have source v_i and labels acceptable to s_i. As product graph paths are guided by state transition in DFA, the depth of Δ tree rooted at $\langle v_i, s_i \rangle$ can be approximated as the maximum length of paths built in a BFS starting from s_i in the DFA. We allow a circle to repeat t times in the BFS, which is corresponding to a Kleene star in the regular expression. For example, in the DFA of a^*bc shown in Figure 5, the trees which materialize BFS starting from different states are shown on the right ($t = 2$). State s_0 gets depth 4 and state s_1 gets depth 1, while s_2 gets depth 0 (empty BFS tree of s_2 is omitted).

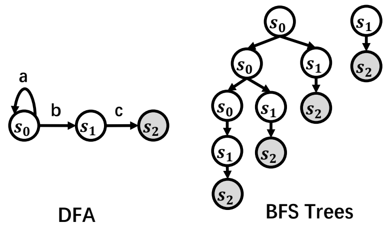


Figure 5: DFA of a^*bc and BFS trees in it

Then we check all the current landmarks. For each landmark $\langle v_i, s_i \rangle$, we check if it falls out of the candidates set. If so, we delete it from the landmark set. Otherwise, we continue to count the number of omitted nodes in subtrees of $\langle v_i, s_i \rangle$. For each Δ tree T_{v_j,s_j} containing $\langle v_i, s_i \rangle$, we count number of nodes which are in T_{v_i,s_i} but not in T_{v_j,s_j}. We add up such node numbers in all Δ trees. If the sum is smaller than the size of T_{v_i,s_i}, selecting $\langle v_i, s_i \rangle$ brings more cost than benefit, and we delete it from the landmark set. For the eliminated landmarks, we delete their LM trees and recover the subtrees rooted at them.

At last, we scan all the nodes in the candidate set in descending order of their estimated Δ tree size. For each candidate which is not a landmark yet, we first build an LM tree for it. Then we count the nodes in the subtrees rooted at it in the Δ tree forest. If the node number in subtrees is larger than the LM tree size, we accept it as a landmark and delete the subtrees. Otherwise, we do not select it as a landmark and delete its LM tree.

When checking existing landmarks and selecting new landmarks, we can also set a higher benefit threshold ϵ by demanding the benefit of a landmark to be ϵ times of the cost. Because the snapshot graph is constantly changing, the cost may exceed the benefit soon if they are close. Note that if a node has state s_0, there is always a Δ tree rooted at it whether it is a landmark or not. Thus we do not need to build a new LM tree when selecting it as a landmark. We add it to the landmark set as long as it is in the candidate set.

3.3 Accelerate Update With TI Map

Though LM-SRPQ merges common subtrees in S-PATH and achieves higher memory efficiency, there are several defects in its update algorithm which cause low time efficiency:

(1) **Some local paths are in vain.** We may build a local path p' to $\langle v_j, s_j \rangle$ in T_{v_i,s_i} when there is already a path p from $\langle v_i, s_i \rangle$ to $\langle v_j, s_j \rangle$ and $p.ts > p'ts$. But p is not a local path and we lack information about it.

(2) **The dependency graph traversal cost is high.** There may be multiple dependency paths between two Δ trees. It takes a high cost to find the one with the largest timestamp with traversal, even though we have bounded the size of the dependency graph.

(3) **Some path concatenations are in vain.** We may build multiple paths between two nodes with concatenation of different dependency paths and local paths. Among them only the latest one is effective, and the building of others should be pruned if possible.

In order to solve these problems, we propose to build another index for each LM tree, called TI map (Time Information map). This index is formally defined as follows:

Definition 3.2. *TI map: TI map is a index maintained in each LM tree T_{v_i,s_i}. It stores node-timestamp pairs $(\langle v_j, s_j \rangle, ts)$, where there is at least one path from $\langle v_i, s_i \rangle$ to $\langle v_j, s_j \rangle$ in the product graph, and $ts = MaxTime(\langle v_i, s_i \rangle \rightarrow \langle v_j, s_j \rangle)$.*

We use $T_{v_i,s_i}.TI(v_j, s_j)$ to denote the timestamp of $\langle v_j, s_j \rangle$ in TI map of T_{v_i,s_i}. Figure 6 shows an example of LM-SRPQ with TI map. We omit the dependency graph as it is the same as Figure 4.

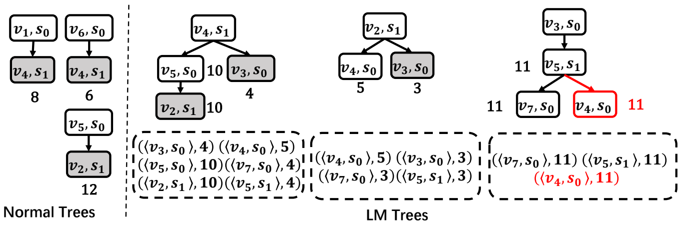

v_1, s_0
v_4, s_1
8
v_6, s_0
v_4, s_1
6
v_5, s_0
v_2, s_1
12
Normal Trees
v_4, s_1
v_5, s_0
10
v_3, s_0
4
v_2, s_1
10
$(\langle v_3, s_0 \rangle, 4)$ $(\langle v_4, s_0 \rangle, 5)$
$(\langle v_5, s_0 \rangle, 10)$$(\langle v_7, s_0 \rangle, 4)$
$(\langle v_2, s_1 \rangle, 10)$$(\langle v_5, s_1 \rangle, 4)$
v_2, s_1
v_4, s_0
5
v_3, s_0
3
$(\langle v_4, s_0 \rangle, 5)$ $(\langle v_3, s_0 \rangle, 3)$
$(\langle v_7, s_0 \rangle, 3)$$(\langle v_5, s_1 \rangle, 3)$
v_3, s_0
11
v_5, s_1
11
v_7, s_0
v_4, s_0
11
$(\langle v_7, s_0 \rangle, 11)$ $(\langle v_5, s_1 \rangle, 11)$
$(\langle v_4, s_0 \rangle, 11)$
LM Trees

Figure 6: Update in LM-SRPQ with TI map

TI map is used in the following 4 cases to solve the above defects:

Prune redundant local path building: When we try to add a new node into a Δ tree, we can first query the TI map to find the largest timestamp of existing paths. If the timestamp in TI map is larger than the new local path, we do not add this node.

Omit forward search in dependency graph: Given a new dependency edge $\overrightarrow{T_{v_i,s_i}, T_{v_j,s_j}}$, we can omit the forward search from T_{v_j,s_j}, which is meant to find paths to successors of $\langle v_j, s_j \rangle$ (as discussed in Section 3.1). Instead, we can fetch the timestamp of the latest path between $\langle v_j, s_j \rangle$ and each successor $\langle v_q, s_q \rangle$ in TI map.

Prune backward search in dependency graph: A backward search is used to find precursors of a Δ tree T_{v_i,s_i} when there are new paths starting from $\langle v_i, s_i \rangle$ (either T_{v_i,s_i} is the source of a new dependency edge, or a new local path is built in it, as discussed in Section 3.1). We can use TI map to check if we arrive at a Δ tree through the latest dependency path in the backward search. For example, in Figure 4, when we search backward from T_{v_3,s_0} and arrives at T_{v_4,s_1} passing T_{v_2,s_1}, we can find the timestamp of this path is 3, while the timestamp of $\langle v_3, s_0 \rangle$ in the TI map of T_{v_4,s_1} is 4. Then we can prune this search branch. It prevents us from further searching up following a not latest path. With TI map we can decrease the cost of each backward search form $O(|E_d| \times |V_d|)$ to $O(|E_d| + |V_d|)$, where $|E_d|$ and $|V_d|$ are the number of edges and vertices in the dependency graph.

Prune redundant path concatenation: In a backward search from T_{v_i,s_i}, we group new paths starting from $\langle v_i, s_i \rangle$ with a set PS, and concatenate the dependency path dp from each precursor T_{v_j,s_j} to T_{v_i,s_i} with paths in PS, in order to build new paths starting from $\langle v_j, s_j \rangle$. With TI map, we can check if we build the latest path by concatenating dp with each path $p \in PS$. If not, we do not need to concatenate dependency paths with p in this search branch later. For example, In Figure 6, a new product graph edge $\overrightarrow{\langle v_5, s_1 \rangle, \langle v_4, s_0 \rangle}$ is added (colored in red), inducing a new path in T_{v_3,s_0} to $\langle v_4, s_0 \rangle$. We carry out a backward search from T_{v_3,s_0} and arrives at T_{v_4,s_1}. We concatenate the dependency path from T_{v_4,s_1} to T_{v_3,s_0} with the new local path, and the timestamp of the concatenated path ($p_1 = \langle v_4, s_1 \rangle \rightarrow \langle v_3, s_0 \rangle \rightarrow \langle v_5, s_1 \rangle \rightarrow \langle v_4, s_0 \rangle$) is 4. However, we find that the timestamp of $\langle v_4, s_0\rangle$ in TI map of T_{v_4,s_1} is 5, indicating that there is an existing path with a larger timestamp ($p_2 = \langle v_4, s_1\rangle \rightarrow \langle v_5, s_0\rangle \rightarrow \langle v_2, s_1\rangle \rightarrow \langle v_4, s_0\rangle$). For the precursors of T_{v_4,s_1}, namely T_{v_1,s_0} and T_{v_6,s_0}, the new path starting from $\langle v_1, s_0\rangle$ (or $\langle v_6, s_0\rangle$) is built by concatenating the dependency path from T_{v_1,s_0} (or T_{v_6,s_0}) to T_{v_4,s_1} with p_1, and it must have smaller timestamp than an existing path, which is the concatenation of the same dependency path with p_2. Therefore, we can predict that we can not get effective updates in them and prune this search branch at T_{v_4,s_1}.

Note that we only build TI map for LM trees. There are much more normal trees than LM trees, and building TI map for all of them is memory consuming. Besides, TI map has limited use in normal trees. Normal trees do not have precursors in the dependency graph. Therefore, we will not visit them in forward search, and as backward search ends at them, pruning in them has little benefit. Given the high cost and low benefit, we give up building TI map for normal trees. But we can still compute $MaxTime(\langle v_i, s_i\rangle \rightarrow \langle v_q, s_q\rangle)$ in normal tree T_{v_i,s_i} as the maximum value among $T_{v_i,s_i}.\langle v_q, s_q\rangle.ts$ and $Max\{Min(T_{v_i,s_i}.\langle v_j, s_j\rangle.ts, T_{v_j,s_j}.TI(v_q, s_q))\}$ where $\langle v_j, s_j\rangle$ denotes landmarks in T_{v_i,s_i}.

Specifically speaking, with TI map, the update and expiration algorithm of LM-SRPQ are as follows:

Update: For a new tuple $sgt = (e = \overrightarrow{v_b, v_d}, l, ts)$, we find all state pairs (s_b, s_d) where $\delta(s_b, l) = s_d$. For each state pair (s_b, s_d), we add T_{v_b,s_b} into the forest if $\langle v_b, s_b\rangle$ is a landmark or $s_b = s_0$, and T_{v_b,s_b} does not exits yet. Then if $\langle v_b, s_b\rangle$ is a landmark, we only need to update LM tree T_{v_b,s_b}, as the subtrees rooted at $\langle v_b, s_b\rangle$ are omitted in other Δ trees. Otherwise, we update all Δ trees containing node $\langle v_b, s_b\rangle$. At last, we carry out a backward search from updated LM trees and update their precursors in the dependency graph.

Update in LM trees: In an LM tree T_{v_x,s_x} containing node $\langle v_b, s_b\rangle$, the update is similar to Algorithm 1, except 3 differences:

(1) Before the update, we check if the local path to $\langle v_b, s_b\rangle$ is the latest. If $T_{v_x,s_x}.\langle v_b, s_b\rangle.ts < T_{v_x,s_x}.TI(v_b, s_b)$, we stop the update.

(2) For each node $\langle v_i, s_i\rangle$ we encounter in BFS, we query the TI map to check if there is an existing path with the same, or larger timestamp than the new local path, namely $T_{v_x,s_x}.TI(v_i, s_i) \geq \langle v_i, s_i\rangle.ts$. If so, we prune this search branch. Otherwise, we need to update the TI map with $\langle v_i, s_i\rangle.ts$. We also need to update the result set if $s_x = s_0$ and $s_j \in A_R.F$.

(3) When we encounter a landmark $\langle v_i, s_i\rangle$ in BFS search, we stop this search branch. On the other hand, we visit the TI map of T_{v_i,s_i}. For each node $\langle v_j, s_j\rangle$ in the TI map, we compute timestamp of the path to it as $ts = Min(T_{v_x,s_x}.\langle v_i, s_i\rangle.ts, T_{v_i,s_i}.TI(\langle v_j, s_j\rangle))$, namely timestamp of concatenating the local path from $\langle v_x, s_x\rangle$ to $\langle v_i, s_i\rangle$ and the latest path from $\langle v_i, s_i\rangle$ to $\langle v_j, s_j\rangle$. We update the TI map of T_{v_x,s_x} with ts, and also update the result set if $s_x = s_0$ and $s_j \in A_R.F$.

Update in normal trees: In a normal tree T_{v_x,s_0} with node $\langle v_b, s_b\rangle$, the update is similar to update in LM trees. The only difference is that we need to compute $MaxTime(\langle v_x, s_0\rangle \rightarrow \langle v_i, s_i\rangle)$ when we try to determine whether a local path is the latest. As the cost of such computation is considerable, we only perform this check to the endpoints of the new edge, namely $\langle v_b, s_b\rangle$ and $\langle v_d, s_d\rangle$.

Backward search: A backward search is carried out from each updated LM tree T_{v_x,s_x}. We use a set RT to record the nodes $\langle v_q, s_q\rangle$ where $T_{v_x,s_x}.TI(v_q, s_q)$ has been updated. The timestamp of these nodes in $T_{v_x,s_x}.TI$ are also stored in RT and denoted as $RT(v_q, s_q)$. There is a new path p with timestamp $RT(v_q, s_q)$ from $\langle v_x, s_x\rangle$ to each node $\langle v_q, s_q\rangle$. We need to find precursors of T_{v_x,s_x} in the dependency graph with the backward search. By concatenating the latest dependency path dp from precursor T_{v_i,s_i} to T_{v_x,s_x} with p, we build a new path from $\langle v_i, s_i\rangle$ to $\langle v_q, s_q\rangle$, and update the result set accordingly. Besides, we also add the source node of the new edge $\langle v_b, s_b\rangle$ and its timestamp to RT, which will help in pruning.

Algorithm 2: *BackwardSearch*$(\cdot)$

```
Input: landmark $\langle v_x, s_x\rangle$, set of timestamp-updated nodes $RT$,
       new tuple $sgt = (e = \overrightarrow{v_b, v_d}, l, ts)$, state pair $(s_b, s_d)$
Output: updated $\Delta$ trees and result set $RS$
1  forall $\Delta$ trees $T_{v_i,s_i}$ where $\langle v_x, s_x\rangle \in T_{v_i,s_i}$ do
2      $ts_1 = Min(RT(v_b, s_b), T_{v_i,s_i}.\langle v_x, s_x\rangle.ts)$
3      $ts_2 = Min(RT(v_d, s_d), T_{v_i,s_i}.\langle v_x, s_x\rangle.ts)$
4      if $ts_1 < MaxTime(\langle v_i, s_0\rangle \rightarrow \langle v_b, s_b\rangle) || ts_2 \leqslant$
         $MaxTime(\langle v_i, s_0\rangle \rightarrow \langle v_d, s_d\rangle)$ then
5          continue
6      $\bar{RT} = \{(\langle v_b, s_b\rangle, ts_1)\}$
7      forall $(\langle v_q, s_q\rangle, ts) \in RT$ do
8          $ts = min(ts, T_{v_i,s_i}.\langle v_x, s_x\rangle.ts)$
9          if $T_{v_i,s_i}$ is an LM tree then
10             if $\langle v_q, s_q\rangle \notin T_{v_i,s_i}.TI$ or $T_{v_i,s_i}.TI(v_q, s_q) < ts$ then
11                 $UpdateMap(T_{v_i,s_i}.TI, \langle v_q, s_q\rangle, ts)$
12                 $\bar{RT} += (\langle v_q, s_q\rangle, ts)$
13         if $s_i = s_0 \&\& s_q \in A_R.F$ then
14             $UpdateMap(RS, (v_i, v_q), ts)$
15     if $T_{v_i,s_i}$ is an LM tree then
16         $backtrack(\langle v_i, s_i\rangle, \bar{RT}, sgt, (s_b, s_d))$
```

The backward search algorithm is shown in Algorithm 2, which is a recursive function. Given a landmark tree T_{v_x,s_x} and set RT, we check all the 1-hop precursors of T_{v_x,s_x} in the dependency graph, namely Δ trees containing $\langle v_x, s_x\rangle$. For precursor T_{v_i,s_i}, we concatenate the local path from $\langle v_i, s_i\rangle$ to $\langle v_x, s_x\rangle$ with the new paths from $\langle v_x, s_x\rangle$ to each node $\langle v_q, s_q\rangle$ in RT. The timestamp of the former is $T_{v_i,s_i}.\langle v_x, s_x\rangle.ts$, and the latter is $RT(v_q, s_q)$. Thus the timestamp of the concatenated new path is $Min(T_{v_i,s_i}.\langle v_x, s_x\rangle.ts, RT(v_q, s_q))$.

We first check the timestamp of the new path to $\langle v_b, s_b\rangle$, if it is smaller than $MaxTime(\langle v_i, s_i\rangle \rightarrow \langle v_b, s_b\rangle)$, we can prune this search branch (line 2-5). Because all the other nodes in RT are successors of $\langle v_b, s_b\rangle$. If the new path to $\langle v_b, s_b\rangle$ is not the latest, paths to them will neither be. Note that for LM trees, $MaxTime(\langle v_i, s_i\rangle \rightarrow \langle v_b, s_b\rangle)$ can be fetched in the TI map. But for normal trees, we need to compute it with the TI maps of landmarks in the normal tree. By checking the timestamp of $\langle v_b, s_b\rangle$, we also filter the case when we traverse to T_{v_i,s_i} following a not latest dependency path. A similar check is performed to the destination node $\langle v_d, s_d\rangle$. If $\langle v_d, s_d\rangle$ exists in TI map of T_{v_i,s_i} with no smaller timestamp, it means there are existing paths to it with no smaller timestamp, and it will be the same for other nodes in RT, as they are all successors of $\langle v_d, s_d\rangle$. In this case, we prune the search branch (line 2-5). Then if T_{v_i,s_i} is an LM tree, we update the TI map according to the timestamps of the new paths (line 10-12). If $s_i = s_0$, we also need to update the result set (line 13-14). Besides, we collect $\langle v_b, s_b \rangle$ and the nodes whose timestamp has been updated in TI map with another set $\vec{RT}$, and further searches backward (line 15-16).

Expiration: The expiration procedure is similar to S-PATH. We delete the expired edges from the snapshot graph, delete the expired tree nodes according to these expired edges, and delete the expired results from the result set. We also need to delete the expired tuples i.e., $(\langle v_q, s_q \rangle, ts)$ with $ts < \tau - N$, in the TI map of each LM tree, and delete expired edges in the dependency graph.

Note that we do not consider the memory usage of TI map in landmark selection. TI map is a light-weight integer type key-value map. With the number of LM trees bounded, the memory cost of TI map is controllable. Besides, TI map has a high benefit to the time efficiency of update, we should give it a privilege in memory using, in order to trade off memory efficiency and time efficiency.

Theorem 3.2. *Both LM-SRPQ and S-PATH have space cost $O(k|V|^2)$, where k is the number of states in DFA and $|V|$ is the number of vertices in the snapshot graph. The amortized time cost of each edge insertion and expiration procedure is also $O(k|V|^2)$.*

Proof. For S-PATH, there are at most $|V|$ Δ trees with different roots $\langle v_i, s_0 \rangle$, where v_i is a vertex in the snapshot graph. In each Δ tree, there are at most $k|V|$ nodes, equal to the number of nodes in the product graph. Therefore, the total memory cost is $O(k|V|^2)$. In edge insertions, a tree node in a spanning tree may be updated multiple times, each time assigned with a new path and a new timestamp. As the timestamp of a path is equal to one of its component edges, in a period of insertion n edges, each tree node will be updated at most n times. As there are $O(k|V|^2)$ tree nodes, the total cost of inserting n edges is $O(k|V|^2 \times n)$, and the amortized cost of each edge insertion is $O(k|V|^2)$. For expiration, as there are $O(k|V|^2)$ tree nodes and $O(|V|^2)$ result pairs, the time cost to check and delete them is also $O(k|V|^2)$.

For LM-SRPQ, there are at most $|V|$ normal trees and $\rho k|V|$ LM trees, where ρ is the candidate selection rate of landmarks. We can adjust ρ to make sure that ρk is a constant. Overall there are also $O(|V|)$ Δ trees. The number of omitted nodes in each spanning tree is hard to estimate, as it depends on the topology detail of the graph. We still assign a very loose upperbound $O(k|V|)$ to the number of nodes in each Δ tree. Overall there are $O(k|V|^2)$ nodes in the spanning forests. The cost of TI map in LM trees is also $O(k|V|^2)$, as there are $O(|V|)$ LM trees and each of them has a TI map with size at most $O(k|V|)$. Overall the memory cost is $O(k|V|^2)$, the same as S-PATH. The amortized cost of updating tree nodes in each edge insertion can be analyzed in the same way as S-PATH, and we will get the same cost estimation. The time cost of backward search procedure can be analyzed similarly. With the prune techniques we propose above, we prune the vain updates. In a period of n edge insertions, each TI map entry and result pair will be updated at most n times, and there are $O(k|V|^2)$ result pairs and FN map enries, thus the amortized cost of updating them in backtrack procedure in each edge insertion is also $O(k|V|^2)$. There is also a traversal cost is the dependency graph, which is $O(|V|)$ as each spanning tree will only be visited once. Overall, the amortized cost of update in each edge insertion is $O(k|V|^2)$. For expiration, as there are at most $O(k|V|^2)$ TI map entries, the cost to check and delete them is $O(k|V|^2)$. Adding the cost of deleting tree nodes and result pairs into consideration, the overall time cost of each expiration is still $O(k|V|^2)$. □

Generally speaking, as the number of omitted tree nodes in LM-SRPQ is hard to quantize, the estimated space and time cost of LM-SRPQ is the same as the baseline method. However, as will be shown in the next section, LM-SRPQ has higher speed and less memory usage in most experiments.

4 IMPLEMENTATION DETAILS

In this section, we introduce some notable details in the implementation of building/deleting LM trees and recovering/deleting subtrees after landmark selection.

When a new landmark $\langle v_i, s_0 \rangle$ has an initial state s_0, there is already a normal tree for it. We can directly re-classify the normal tree to LM trees. Besides, we need to build TI map for the new LM tree, which is simple. For each node $\langle v_j, s_j \rangle$, $T_{v_i,s_0}.TI(v_j, s_j)$ is the maximum value between $Max\{Min(T_{v_q,s_q}.TI(v_j, s_j), T_{v_i,s_0}.\langle v_q, s_q \rangle.ts)\}$ and $T_{v_i,s_0}.\langle v_j, s_j \rangle.ts$, where $\langle v_q, s_q \rangle$ represents landmarks in T_{v_i,s_0}

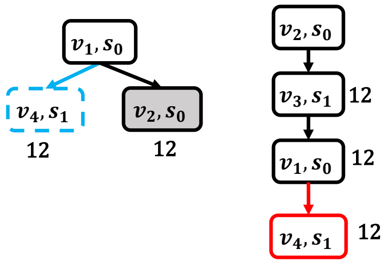


Figure 7: Example of missing nodes in LM tree building

However, there should be noted that we need to add some nodes into the new LM tree to fulfill it, to avoid missing paths due to the pruning in Section 3.3. Consider the following example in Figure 7. $\langle v_2, s_1 \rangle$ is an LM tree. T_{v_1,s_0} depends on T_{v_2,s_1}, but due to the existence of circles, $\langle v_1, s_0 \rangle$ is also in T_{v_2,s_1}. Suppose all the edges in this example has the same timestamp 12. When a new edge between $\langle v_1, s_0 \rangle$ and $\langle v_4, s_1 \rangle$ with the same timestamp 12 is added, we first update T_{v_2,s_1} and add a new path (colored in red). When we update T_{v_1,s_0}, we will find that there is already a path to $\langle v_4, s_1 \rangle$ passing landmark $\langle v_2, s_1 \rangle$, and will prune the local path (colored in blue). However, when we select $\langle v_1, s_0 \rangle$ as a landmark and delete the subtree of $\langle v_1, s_0 \rangle$ in T_{v_2,s_1}, the path to $\langle v_4, s_1 \rangle$ will be lost. To solve this problem, we need to gather nodes where the path from the new landmark to them in the deleted subtrees is the latest, and add these nodes to the new LM tree if they are not in it.

When the new landmark $\langle v_i, s_i \rangle$ does not have an initial state, we need to build a new LM tree for it. We can build the tree with a BFS from the root $\langle v_i, s_i \rangle$, and the procedure is the same to the update algorithm of LM trees described in Section 3.3. However, it should be noted that after building the LM tree, we need to fulfill it with the lost nodes in the same way as described above.

When a landmark $\langle v_i, s_i \rangle$ is eliminated. We need to recover the subtrees rooted at it and then delete the LM tree. When recovering subtree of $\langle v_i, s_i \rangle$ in T_{v_x,s_x}. We can carry out a BFS from $\langle v_i, s_i \rangle$. The procedure is the same as the update algorithm of LM trees described

Notation	Query	Notation	Query
Q1	a^*	Q6	ab^*c
Q2	$a?b^*$	Q7	$(a_1 + a_2 + ... + a_k)b^*$
Q3	ab^*	Q8	a^*b^*
Q4	abc	Q9	ab^*c^*
Q5	abc^*	Q10	$(a_1 + a_2 + ... + a_k)^*$

Table 2: Queries used in experiments

in Section 3.3. However, if a node found in BFS already exists in tree T_{v_x,s_x}, we cannot prune the search branch. Because the successors of it may not exist in T_{v_x,s_x} yet. Consider the example in Figure 7. Suppose in the landmark selection we do not select $\langle v_1, s_0\rangle$ as a landmark as discussed above, and delete $\langle v_2, s_1\rangle$ from the landmark set instead. When we recover the subtree of $\langle v_2, s_1\rangle$ in T_{v_1,s_0} with a BFS and search to $\langle v_1, s_0\rangle$, we find that it is already in the tree with a larger timestamp. We do not need to modify the node, but we need to carry on the BFS. Otherwise, we will lose node $\langle v_4, s_1\rangle$.

5 EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate our method over two real-world datasets and one synthetic dataset. Details about the datasets, workloads and experimental settings are discussed in Section 5.1 and Section 5.2, respectively. In Section 5.3 and Section 5.4, we compare LM-SRPQ with prior art S-PATH algorithm [25] on both memory usage and processing speed. In Section 5.5, we evaluate the scalability of LM-SRPQ and S-PATH. We have also implemented experiments on the candidate selection rate of LM-SRPQ in Section 5.6.

5.1 Datasets and Workloads

The regular path queries we use in experiments are shown in Table 2. They are the most common recursive queries found in real-world applications [5], plus a popular non-recursive query Q4 abc [3] . We set $k = 3$ for queries with a variable number of labels, as there are only 3 kinds of labels in StackOverflow, a dataset we use in experiments. We process edges in datasets in time order to simulate real-world streaming graphs. The following two real-world datasets and one synthetic dataset are used in experiments.

StackOverflow: [4] This is a temporal graph of interactions on the stack exchange website Stack Overflow. Vertices are users and edges represent user interactions. There are 63,497,050 edges and 2,601,977 vertices, spanning 8 years. There are three kinds of edges in this dataset, representing different types of user interactions. This dataset is much more homogeneous and cyclic than other datasets, as it only has one type of vertices and three kinds of labels. We set the window size to 20 days and sliding interval to 1 day unless otherwise specified.

LDBC: This is a synthetic dataset that simulates real-world interactions in social networks [11]. The generated workload has both a static part and an update stream, and we extract the update stream to use in experiments. There are 10 types of interactions, but only two kinds of edges are recursive: *ReplyOf* and *Knows*. Moreover, these two kinds of edges are connected with different types of vertices. As a result, Q8, Q9 and Q10 cannot be meaningfully expressed. Therefore, we only test Q1 ∼ Q7 in this dataset. With a scale factor of 10, there are 55,823,323 edges and 7,586,929 vertices, spanning 3.5 months. We set the window size to 3 days and sliding interval to 6 hours unless otherwise specified.

Yago2s: This is a real-world RDF dataset [5], with 244,800,042 edges and 10,852,613 vertices after transformed into a graph. It has 104 edge labels, and supports all queries in Table 2. Edges in this dataset do not have associated timestamps. We randomly shuffled the edges and assign a monotonically non-decreasing timestamp for each edge with a fixed rate. We set the window size to $2M$ edges and sliding interval to $256k$ edges unless otherwise specified.

5.2 Experiment Implementation

Both S-PATH and LM-SRPQ are implemented with C++ and compiled with GCC 5.4.0 and O2 option. As there are many queries to test and some queries are both memory and time consuming, we split these experiments on two servers. Experiments with StackOverflow are implemented on a server with 192GB memory and two Intel Xeon 2.30GHz 18-core CPU. Experiments with LDBC and Yago2s are implemented on another server with 384GB memory and two Intel Xeon 2.60GHz 8-core CPU. For LM-SRPQ, we set the candidate selection rate $\rho = 20\%$ and benefit threshold $\epsilon = 1.5$ unless otherwise specified. And we allow a circle to be repeated $t = 6$ times when carrying out BFS in DFA (details about these parameters are in Section 3.2). The interval of landmark selection is set to the sliding interval. We set a checkpoint whenever the largest timestamp of processed edges increases by N, namely after processing a sliding window. We measure metrics at checkpoints and use the average value of all checkpoints as experimental results.

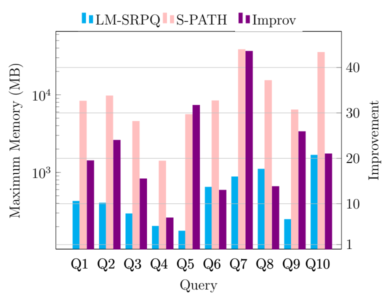
5.3 Memory Comparison

We measure the average memory usage and maximum memory usage of LM-SRPQ and S-PATH, as shown in Figure 9 and 8. The unit of memory usage is MB. We also present the improvements brought by LM-SRPQ compared to S-PATH, which is calculated as $\frac{memory\ of S-PATH}{memory\ of\ LM-SRPQ}$. LM-SRPQ outperforms S-PATH when the improvement is larger than 1. The higher the improvement is, the better LM-SRPQ performs. The memory usage excludes the cost of the streaming graph and the result set. Because this cost is the same for both methods. To complete our experiments in a reasonable time, we decrease the window size of some queries, as they are too complicated and have a very low processing speed, especially with S-PATH. Q9 and Q10 of StackOverflow have a window size of 10 days, and we only process edges in the first 800 days. Q8 of LDBC has a window size of 1.5 days.

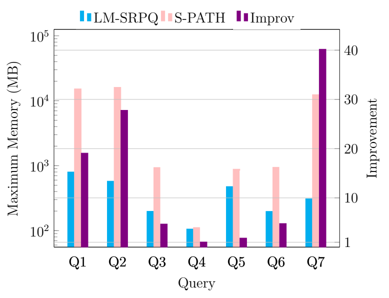
As shown in these figures, LM-SRPQ costs much less memory than S-PATH in most queries. The improvement of average memory usage reaches more than 30 and the the improvement of maximum memory usage reaches 40. As LM-SRPQ depends on merging common subtrees with the same root to save memory, the number and size of common subtrees influence the improvement significantly.

[3]there is another query $(a_1 + a_2 + ... + a_k)^+$ used in experiments of [24]. However, as we omit self-join results, the answer to $(a_1 + a_2 + ... + a_k)^+$ has no difference with Q10 $(a_1 + a_2 + ... + a_k)^*$. Thus we omit this query.
[4]http://snap.stanford.edu/data/sx-stackoverflow.html

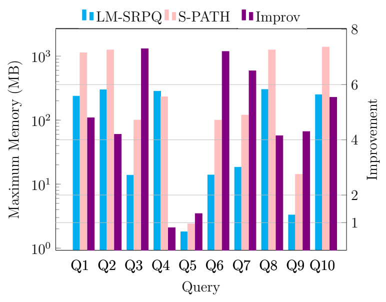
[5]https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/



(a) StackOverflow

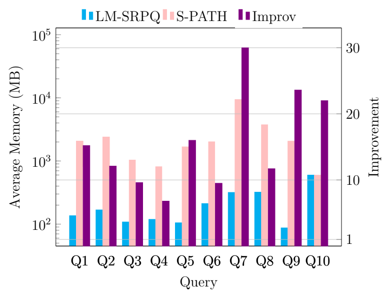


(b) LDBC

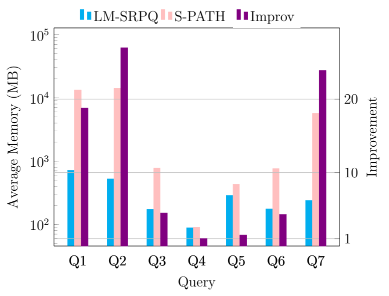


(c) Yago2s

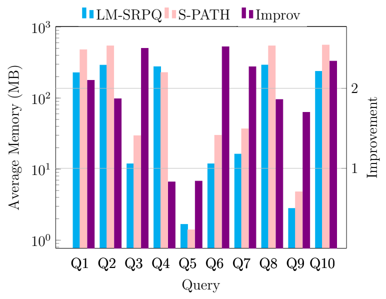
Figure 8: Maximum memory usage comparison



(a) StackOverflow



(b) LDBC



(c) Yago2s

Figure 9: Average memory usage comparison

For example, queries in StackOverflow have higher improvements. This dataset is much denser and more cyclic than other datasets, and has fewer labels. As a result, queries generate larger Δ trees in this dataset, and LM-SRPQ can merge more large subtrees. The lower improvements in Yago2s have the same reason. It has a large number of labels, resulting in a low density of single label and fewer, smaller common subtrees. Similarly, simple and less recursive queries like Q4 have lower improvements. Sometimes the improvement is even smaller than 1, indicating LM-SRPQ has higher memory usage. In this case, the additional cost of maintaining TI map and landmark set exceeds the benefits of merging common subtrees. But in these cases, the memory usage of LM-SRPQ is only higher by less than 20%. On the other hand, in highly recursive queries like Q10, the improvement is always significant.

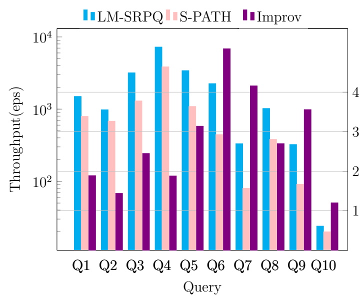
5.4 Processing Speed Comparison

We measure the throughput and tail latency (99_{th} percentile) of edge insertions, and the results are shown in Figure 10 and Figure 11, respectively. The unit of throughput is edge per second (eps), and the unit of tail latency is μs. We also present the improvement brought by LM-SRPQ. For throughput, the improvement is calculated as $\frac{throughput\ of\ LM-SRPQ}{throughput\ of\ S-PATH}$. For tail latency, the improvement is calculated as $\frac{tail\ latency\ of\ S-PATH}{tail\ latency\ of\ LM-SRPQ}$. The experiment settings are the same as the experiments in Section 5.3.

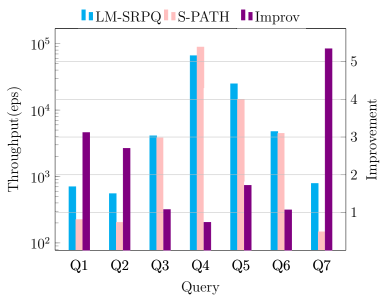
As shown in these figures, LM-SRPQ outperforms S-PATH in most queries. The improvement of throughput reaches at most 5, and the improvement of tail latency reaches at most 30. Similar to the memory cost, the number and size of common subtrees significantly influence the improvement brought by LM-SRPQ. In dense graphs like StackOverflow and highly recursive queries like Q10, the improvement is significant. On the other hand, in sparse graphs and simple queries, the improvement is lower.

5.5 Scalability Evaluation

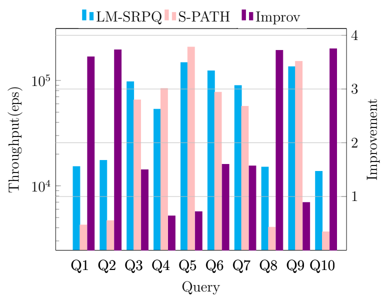
In this section, we vary the window size to test the scalability of LM-SRPQ and S-PATH. The experimental results are shown in Figure 12. The dataset we use in this experiment is StackOverflow. We choose Q1, Q4 and Q8 as representative queries in order to keep the figure legible. These queries cover three query types: non-recursive query (Q4), recursive query with Kleene star on a single label (Q1), and highly recursive query with Kleene stars on multiple labels (Q8). As shown in these figures, the memory usage of both methods grows quickly with the window size, and the time efficiency drops. Because when the sliding window becomes larger, the snapshot graph becomes larger and denser, and the size of Δ tree forest also grows. Therefore, both the memory cost and update cost increase. The advantage of LM-SRPQ grows with the window size. Because when the Δ tree forest enlarges, there are more and larger common subtrees to merge, and LM-SRPQ will obtain a higher advantage.



(a) StackOverflow

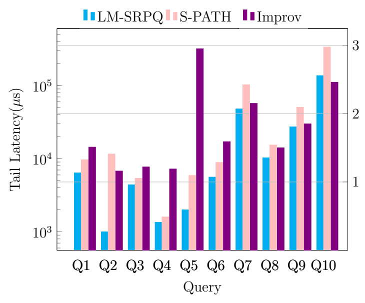


(b) LDBC

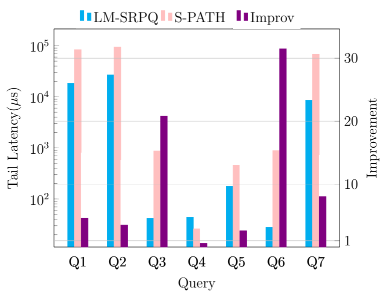


(c) Yago2s

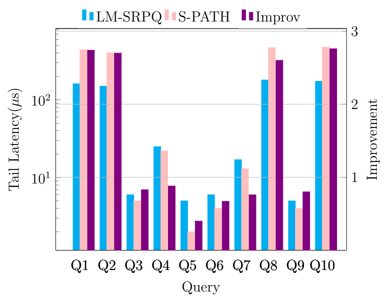
Figure 10: Throughput comparison



(a) StackOverflow

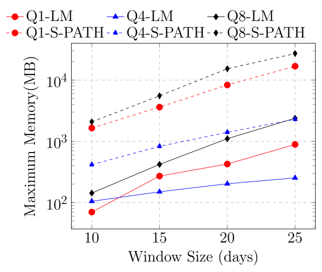


(b) LDBC

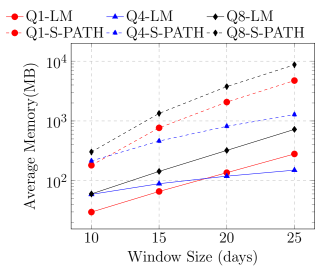


(c) Yago2s

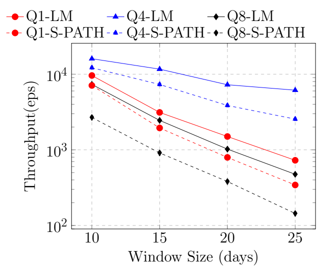
Figure 11: Tail latency comparison



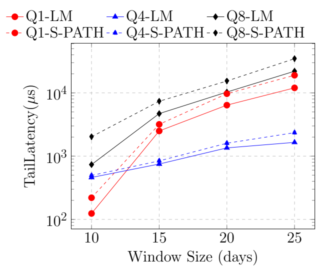
(a) Maximum Memory Usage



(b) Average Memory Usage



(c) Throughput



(d) Tail Latency

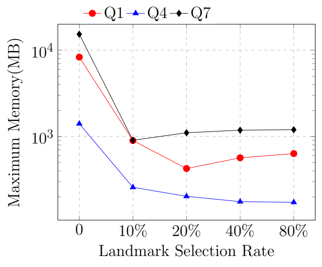
Figure 12: Performance vary with window size

5.6 Sensitivity Evaluation

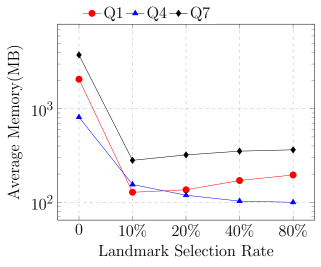
In this section, we test the influence of candidate selection rate, namely the percentage of qualified nodes (roots of common subtrees, as discussed in Section 3.2) which are selected as landmark candidates. The experimental results are shown in Figure 13. The dataset we use in this experiment is StackOverflow. Similar to the above experiment, we choose Q1, Q4 and Q8 as representative queries, and show the variation of average memory usage and throughput. From the figure, we can see that the memory usage drops significantly when the landmark selection rate increases from 0 to 10%, as most large common subtrees have been merged with a selection rate of 10%. Selecting more landmarks brings little benefit to memory, and even increases the cost. Because nodes with lower scores have few successors, the benefit of merging subtrees rooted at these nodes is low, and sometimes cannot cover the memory cost of maintaining LM trees and TI maps.

On the other hand, the time efficiency keeps increasing until 20% or 40%, as the time information in TI maps of new LM trees still has benefit to pruning in the dependency graph traversal. Overall, we suggest a selection rate of 20% for most queries.

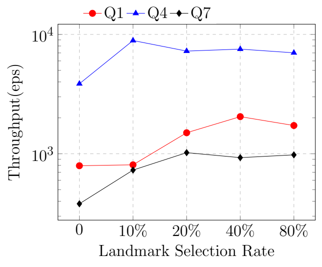
Summary: Generally speaking, LM-SRPQ has higher memory and time efficiency in most cases. In some simple queries, LM-SRPQ has slightly higher memory usage and lower throughput than S-PATH, as common subtrees are rare in these queries, and the cost of selecting landmarks, maintaining LM trees and TI maps exceeds the benefit. However, these queries naturally incur very low memory



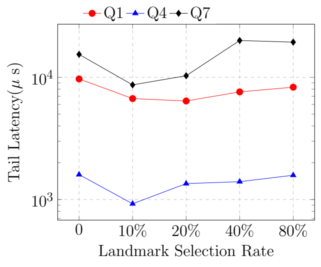
(a) Maximum Memory Usage



(b) Average Memory Usage



(c) Throughput



(d) Tail Latency

Figure 13: Performance varies with candidate selection rate

and time cost, and can hardly become the bottleneck of the system. Meanwhile, in highly recursive queries where the time and memory cost is high, LM-SRPQ will significantly improve the performance.

6 RELATED WORK

Streaming Graph Algorithms: Early researches in streaming graph settings are motivated by the limitation of memory: the graph data, usually static and stored on disk, is too large to fit in memory, thus has to be processed in a streaming manner. These works usually use insertion-only model, where only edge or vertex insertion is considered, and some algorithms allow the graph data to be processed in multiple passes. The topics of these researches are usually graph compression [2, 14, 18] and graph partition [6, 27, 32], which pave the way to further analysis of the massive graph data. Another kind of algorithms allows to store the entire streaming graph in memory and discusses how to maintain the output when the data is updated. This kind includes researches for persistent queries like subgraph matching [8, 17, 21], triangle counting [13, 30, 34], cycle detection [28], as well as building dynamic indexes to support adhoc queries like connectivity [7, 29, 36].

Regular Path Query (RPQ): RPQ has been widely used in graph query languages and systems [3, 4, 12]. There are 2 kinds of semantics for RPQ: arbitrary path and simple path. Simple-path RPQ requires that there are no repeated vertices in the path, while arbitrary-path RPQ does not have this requirement. It has been proved that simple-path RPQ is NP-hard unless the graph and query language meet certain demands [22]. Due to such high complexity of simple-path RPQ, most works use arbitrary path semantics.

Former algorithms for RPQ evaluation can be divided into two kinds: automaton-based approaches and algebra based approaches. For the first kind, there are G [9], a graph query language that builds an automaton to guide the traversal in a graph to answer RPQ, and the work of Kochut et.al. [19] which builds two automatons and performs bi-directional search with them. Besides, Koschmieder et al. [20] propose an algorithm that splits the RPQ into fragments with rare labels and performs bi-directional search for each fragment. And Inju Na et.al. [23] decrease the cost of graph traversal in RPQ evaluation by merging strongly connected components in the graph. A recent work [33] also approximately answers RPQ with random walks. The representative work of the second kind is α-RA based method [1], which extends traditional relational algebra with α operator for transitive closure computation. This method has been widely used in various SPARQL engines [12]. Yakovets et.al.[35] show that these 2 kinds of methods can be combined to explore a larger plan space.

There are only 2 existing algorithms for persistent RPQ. In [24], Pacaci et.al. first propose persistent RPQ, and propose Δ tree based methods for both arbitrary path semantics and simple path semantics. But the algorithm can only handle simple path semantics under certain conditions. In [25], they further extend the definition of persistent RPQ and merge it into an algebra for complex queries in streaming graphs. Different from [24], the algorithm for streaming RPQ in [25], named S-PATH, maintains time information for results in order to support further analysis. In this paper, we focus on arbitrary path semantics, and maintain time information for query results like [25]. But we use the most popular sliding window model to define the validity of edges, paths and query results, rather than the validity interval model in [25].

7 CONCLUSION

In this paper, we propose a novel algorithm for persistent regular path queries (RPQ) in streaming graphs, named LM-SRPQ. It is built on the foundation of prior art, which transforms RPQ problem in snapshot graph to a reachability problem in a product graph, and builds Δ trees which materialize paths in the product graph to solve the problem. LM-SRPQ further finds and merges common substrees in the Δ tree forest, and solves persistent RPQ with a combination of Δ tree maintenance and dependency graph traversal. According to our experiments, the memory usage of LM-SRPQ is at most 40 times smaller than prior art and the throughput is at most 5 times higher, and its superiority is more significant in complex queries and dense graphs.

REFERENCES

[1] Rakesh Agrawal. 1988. Alpha: An extension of relational algebra to express a class of recursive queries. *IEEE Transactions on Software Engineering* 14, 7 (1988), 879–885.

[2] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. 2012. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*. 5–14.

[3] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, et al. 2018. G-CORE: A core for future graph query languages. In *Proceedings of the 2018 International Conference on Management of Data*. 1421–1432.

[4] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)* 50, 5 (2017), 1–40.

[5] Angela Bonifati, Wim Martens, and Thomas Timm. 2019. Navigating the maze of Wikidata query logs. In *The World Wide Web Conference*. 127–138.

[6] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)* 5, 3 (2019), 1–39.

[7] Xin Chen, You Peng, Sibo Wang, and Jeffrey Xu Yu. 2022. DLCR: efficient indexing for label-constrained reachability queries on large dynamic graphs. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1645–1657.

[8] Sutanay Choudhury, Lawrence Holder, George Chin, Khushbu Agarwal, and John Feo. 2015. A selectivity based approach to continuous pattern detection in streaming graphs. *arXiv preprint arXiv:1503.00849* (2015).

[9] Isabel F Cruz, Alberto O Mendelzon, and Peter T Wood. 1987. A graphical query language supporting recursion. *ACM SIGMOD Record* 16, 3 (1987), 323–330.

[10] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 2002. Maintaining Stream Statistics over Sliding Windows. *Siam Journal on Computing* 31, 6 (2002), 1794–1813.

[11] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 619–630.

[12] Orri Erling and Ivan Mikhailov. 2009. RDF Support in the Virtuoso DBMS. *Networked Knowledge-Networked Media: Integrating Knowledge Management, New Media Technologies and Semantic Systems* (2009), 7–24.

[13] Xiangyang Gou and Lei Zou. 2021. Sliding window-based approximate triangle counting over streaming graphs with duplicate edges. In *Proceedings of the 2021 International Conference on Management of Data*. 645–657.

[14] Xiangyang Gou, Lei Zou, Chenxingyu Zhao, and Tong Yang. 2022. Graph stream sketch: Summarizing graph streams with high speed and accuracy. *IEEE Transactions on Knowledge and Data Engineering* (2022).

[15] Ajeet Grewal, Jerry Jiang, Gary Lam, Tristan Jung, Lohith Vuddemarri, Quannan Li, Aaditya Landge, and Jimmy Lin. 2018. Recservice: Distributed real-time graph processing at twitter. In *10th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 18)*.

[16] John Hopcroft. 1971. An n log n algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*. Elsevier, 189–196.

[17] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *Proceedings of the 2018 international conference on management of data*. 411–426.

[18] Jihoon Ko, Yunbum Kook, and Kijung Shin. 2020. Incremental Lossless Graph Summarization. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 317–327.

[19] Krys J Kochut and Maciej Janik. 2007. SPARQLeR: Extended SPARQL for semantic association discovery. In *The Semantic Web: Research and Applications: 4th European Semantic Web Conference, ESWC 2007, Innsbruck, Austria, June 3-7, 2007. Proceedings 4*. Springer, 145–159.

[20] André Koschmieder and Ulf Leser. 2012. Regular path queries on large graphs. In *Scientific and Statistical Database Management: 24th International Conference, SSDBM 2012, Chania, Crete, Greece, June 25-27, 2012. Proceedings 24*. Springer, 177–194.

[21] Youhuan Li, Lei Zou, M Tamer Özsu, and Dongyan Zhao. 2019. Time constrained continuous subgraph search over streaming graphs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1082–1093.

[22] Alberto O Mendelzon and Peter T Wood. 1995. Finding regular simple paths in graph databases. *SIAM J. Comput.* 24, 6 (1995), 1235–1258.

[23] Inju Na, Yang-Sae Moon, Ilyeop Yi, Kyu-Young Whang, and Soon J Hyun. 2022. Regular path query evaluation sharing a reduced transitive closure based on graph reduction. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1675–1686.

[24] Anil Pacaci, Angela Bonifati, and M Tamer Özsu. 2020. Regular path query evaluation on streaming graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1415–1430.

[25] Anil Pacaci, Angela Bonifati, and M Tamer Özsu. 2022. Evaluating complex queries on streaming graphs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 272–285.

[26] Kostas Patroumpas and Timos Sellis. 2006. Window specification over data streams. In *Current Trends in Database Technology–EDBT 2006: EDBT 2006 Workshops PhD, DataX, IIDB, IIHA, ICSNW, QLQP, PIM, PaRMA, and Reactivity on the Web, Munich, Germany, March 26-31, 2006, Revised Selected Papers 10*. Springer, 445–464.

[27] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. 2015. Hdrf: Stream-based partitioning for power-law graphs. In *Proceedings of the 24th ACM international on conference on information and knowledge management*. 243–252.

[28] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1876–1888.

[29] Liam Roditty and Uri Zwick. 2004. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*. 184–191.

[30] Lorenzo De Stefani, Alessandro Epasto, Matteo Riondato, and Eli Upfal. 2017. Triest: Counting local and global triangles in fully dynamic streams with fixed memory size. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 11, 4 (2017), 1–50.

[31] Ken Thompson. 1968. Programming techniques: Regular expression search algorithm. *Commun. ACM* 11, 6 (1968), 419–422.

[32] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*. 333–342.

[33] Sarisht Wadhwa, Anagh Prasad, Sayan Ranu, Amitabha Bagchi, and Srikanta Bedathur. 2019. Efficiently answering regular simple path queries on large labeled networks. In *Proceedings of the 2019 International Conference on Management of Data*. 1463–1480.

[34] Pinghui Wang, Yiyan Qi, Yu Sun, Xiangliang Zhang, Jing Tao, and Xiaohong Guan. 2017. Approximately counting triangles in large graph streams including edge duplicates with a fixed memory usage. *Proceedings of the VLDB Endowment* 11, 2 (2017), 162–175.

[35] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. 2016. Query planning for evaluating SPARQL property paths. In *Proceedings of the 2016 International Conference on Management of Data*. 1875–1889.

[36] Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. 2014. Reachability queries on large dynamic graphs: a total order approach. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1323–1334.