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, achieving 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, 17(5): 1047 - 1059, 2024.
doi:10.14778/3641204.3641214

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 [16].

Due to the unboundedness of streaming graphs, 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 with the sliding window model [11]. 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. The sliding window model is widely used in both research [14, 22, 25] and industrial applications [29].

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 monitored for the emergence of certain graph patterns. Persistent versions of many common graph queries have been studied, such as subgraph matching [18, 22], cycle detection[29], path navigation [25, 26] and triangle counting [14, 31].

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 connected by a path satisfying a regular expression in a directed, edge-labeled multi-graph. 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 [10, 20, 21, 24]. However, RPQ research in streaming graphs is in its infancy. Pacaci et.al. [25] are the first to define persistent RPQ over streaming graphs, and they propose a novel algorithm called RAPQ to solve this problem. RAPQ 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. Persistent RPQ can be answered by incrementally finding new paths in this product graph. However, it will be time-consuming to search new paths from scratch upon each streaming graph update. RAPQ materializes existing paths to all the successors of a qualified product graph node with a tree called Δ tree [1]. Then it updates Δ trees and extends existing paths to find new paths when the streaming graph is updated. In [26], 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 [26], 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 [6], 69% of organic RPQs (RPQs issued by browsers) in Wikidata query

∗ Corresponding author is Lei Zou.

Proceedings of the VLDB Endowment, Vol. 17, No. 5 ISSN 2150-8097.
doi:10.14778/3641204.3641214

[1] In the following sections, we call vertex in the product graph as "node", in order to distinguish them from "vertex" in the streaming graph. Besides, the trees built in [25] are originally called spanning trees, we change their name to avoid confusion with the spanning tree in graph theory.

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 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 size above $21k$ 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 propose landmark-based streaming RPQ (LM-SRPQ) to decrease the size of the Δ tree forests while keeping a high update speed. We notice that there are common subtrees with the same root in different Δ trees, resulting in redundant storage. The subtree with root v_i in a Δ tree T stores paths from v_i to a subset of its successors, which are fragments of paths from the root of T to these successors. Common subtrees in different Δ trees are induced by common path fragments. Therefore, we can decrease memory usage by merging these common path fragments, which leads to the merging of common subtrees. We propose to select a group of nodes as landmarks. Each path in the product graph can be split into a sequence of **local paths**, which are path fragments containing no landmarks. We build a Δ tree for each landmark v_i to maintain local paths starting from v_i. Such a Δ tree is called a landmark tree, or LM tree for short. We also build Δ trees for the original tree roots in S-PATH, and they maintain local paths starting from these tree roots. Then we compute paths in the product graph and answer persistent RPQ by continuously computing concatenations of local paths. Each local path from a landmark v_i to its successor v_j may participate in the computation of multiple product graph paths, corresponding to common fragments in these paths. In prior art, this local path will be stored multiple times in the common subtrees of v_i. But in our method, we only need to store it once in the LM tree of v_i. Therefore, redundant storage is eliminated.

However, there are two major challenges in LM-SRPQ:

First, computing local path concatenations raises a performance issue. In LM-SRPQ, we build a dependency graph G_d to aid us in the local path concatenation. Each node in G_d represents a Δ tree and each edge represents the local path connecting the roots of two Δ trees. Concatenated local path sequences can be represented as paths in G_d. When a new tuple arrives in the streaming graph, we first update the Δ trees, producing new local paths and adding new edges in G_d. Then we need to traverse in G_d to find new paths containing the new edge, which represents new local path sequences. The cost of such traversal is high and may slow down the entire algorithm. We need additional acceleration techniques.

Second, how to select a proper landmark set raises another challenge. Different landmark sets result in different Δ tree forest sizes. Moreover, as discussed above, the dependency graph traversal is the bottleneck of the algorithm, and the landmark number influences the number of LM trees, as well as the dependency graph size. We need to bound the landmark number to control the dependency graph traversal cost. Therefore, we hope to select a landmark set with a bounded size and try to minimize the Δ tree forest size. The search space of such landmark selection is exponential, making it computationally impossible to find an optimal solution. We have to resort to a greedy algorithm. Besides, we need to continuously update the landmark set to keep up with the streaming graph.

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}

In LM-SRPQ, we use a greedy algorithm in landmark selection, which tries to minimize the size of the Δ tree forest while bounding the number of Δ trees. Besides, we also continuously maintain an additional index called TI-map in each LM tree, which records the 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 30 times. Moreover, as common subtree merging reduces the Δ tree update cost and efficient pruning reduces the dependency graph traversal cost, LM-SRPQ is at most 4.5 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 decreases the memory cost as well as the update cost of the prior art by eliminating redundant storage and computation.
(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 in the dependency graph.
(3) We carry out extensive experiments to evaluate our algorithm, which confirms 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 [26] 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 $\phi : E \to \Sigma$ 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 [25] and [26], 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.

Definition 2.3. ***Streaming Graph.*** *A streaming graph is a sequence of streaming graph tuples $S = \{sgt_1, sgt_2, ...\}$ which arrive 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 $(\lfloor \frac{\tau}{\beta} \rfloor \times \beta - N, \tau]$. 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 [27]. On the other hand, we process new tuples in real time to produce fresh results. Furthermore, the graph induced by active tuples 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 $\tau = 13$ 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* 2*). 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 when the snapshot graph changes due to new tuples' arrival and old tuples' expiration. 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 $R ::= \epsilon | a | R \circ R | R + R | R^* | R^+ | R?$ 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.*
(7) *$R?$ means 0 or 1 repetition 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. 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 $p = v_6 \rightarrow v_3 \rightarrow v_5 \rightarrow v_4 \rightarrow v_2$ (marked in red) conforms to R due to $\phi(p) \in L(R)$. Thus, (v_6, v_2) 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. Specifically, we have the following definition of 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 $PS = \{p_1, ..., p_m\}$ denote 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., $(v_i, v_j).ts = Max\{p.ts | p \in PS\}$.

The timestamps of vertex pairs reflect their life span and enable us to use a direct approach to process expiration. We can directly find and delete all expired vertex pairs with timestamps smaller than $\tau - N$. All the regular paths connecting these vertex pairs have expired. These timestamps may also be used for further analysis in different applications. Besides, for path concatenation $p = p_1 \circ p_2$, we get $p.ts = Min\{p_1.ts, p_2.ts\}$ according to the above definition.

In the following sections, we solve the persistent 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 $S \times \Sigma \rightarrow S$, 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.*

We use s_f to denote states in the final state set F. With a regular expression R, we can create an NDFA with Thompson's construction algorithm [32] and transform it into a DFA using Hopcroft's algorithm [17]. 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*

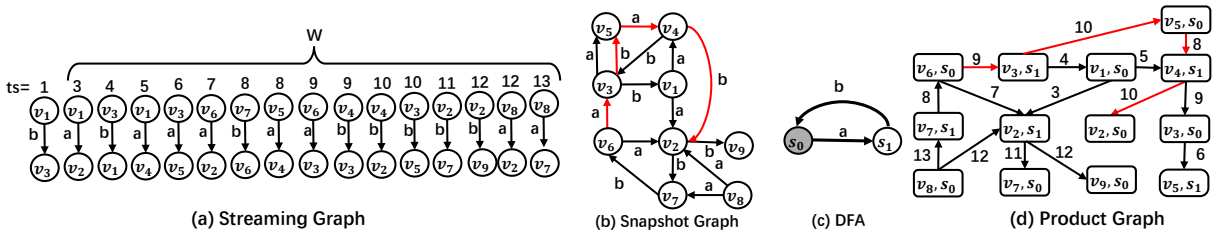


Figure 1: Streaming graph, snapshot graph, DFA for $(a \circ b)^*$ and the corresponding product graph

$P(G, A) = (V_P, E_P, \Sigma, \phi_P)$ *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* $\delta(s_i, \phi(\overrightarrow{v_i, v_j})) = s_j$. *(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 $e' = \overrightarrow{v_i, v_j}$ in G_τ, where the label of e' is l and $\delta(s_i, l) = s_j$. 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 $\langle v_j, s_j\rangle$.

Given a regular expression $R = (a \circ b)^*$, the corresponding DFA is shown in Figure 1 (c). s_0 is both the initial state and the final state, and is marked in shadow. 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 sections.

2.2 Existing Solution:S-PATH Algorithm

In this section, we introduce S-PATH algorithm proposed in [26]. It is an extension of RAPQ algorithm proposed in [25], which adds time information to the result set and adopts direct expiration approach. Note that the original S-PATH algorithm uses a validity interval model, where tuples and paths have validity intervals rather than timestamps. This model is an extension of the sliding window model, which enables to manipulate tuples with different life spans. In this paper, we focus on the sliding window model to keep conciseness of the presentation, and leave more complicated scenarios to future work. Therefore, we modify S-PATH algorithm by changing validity intervals of edges and paths to timestamps, S-PATH algorithm is based on the following theorem [25]:

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^*).

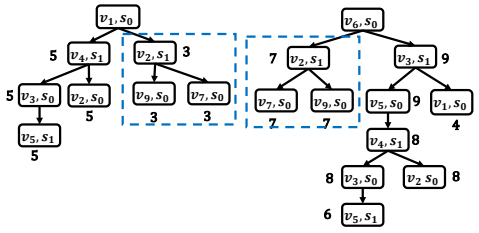


Figure 2: Δ trees in S-PATH

Example 2. *Figure 2 shows Δ trees for the snapshot graph and regular expression in Figure 1 (b) and (c). Note that we only present T_{v_1,s_0} and T_{v_6,s_0} as examples due to space limitation. There is a node $\langle v_2, s_0\rangle \in T_{v_6,s_0}$ with timestamp 8, because there is a path $p = \langle v_6, s_0\rangle \rightarrow \langle v_3, s_1\rangle \rightarrow \langle v_5, s_0\rangle \rightarrow \langle v_4, s_1\rangle \rightarrow \langle v_2, s_0\rangle$ in the product graph with timestamp 8 (marked in red). Besides, p is corresponding to the snapshot graph path $v_6 \rightarrow v_3 \rightarrow v_5 \rightarrow v_4 \rightarrow v_2$ with label abab, where $\delta^*(s_0, abab) = s_0$. Note that there is another path from $\langle v_6, s_0\rangle$ to $\langle v_2, s_0\rangle$ in the product graph, namely $p' = \langle v_6, s_0\rangle \rightarrow \langle v_3, s_1\rangle \rightarrow \langle v_1, s_0\rangle \rightarrow \langle v_4, s_1\rangle \rightarrow \langle v_2, s_0\rangle$. But this path has a smaller timestamp 4. 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) from DFA A_R where $\delta(s_b, l) = s_d$. 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_x} with a Dijkstra-based search, which maintains source-destination-edge triples $(\langle v_i, s_i\rangle, \langle v_j, s_j\rangle, e)$ with a heap Q, sorted in decreasing order of the destination node's

Algorithm 1: Processing new tuples in Δ tree index

Input: new tuple $sgt = (e_b = \overrightarrow{v_b, v_d}, l, ts)$
Output: updated Δ trees and result set RS

1 **forall** $(s_b, s_d) \in A_R.S \times A_R.S$ *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 Δ, $\langle v_b, s_0\rangle.ts = INF$.
4 **forall** $T_{v_x,s_x} \in \Delta$ *where* $\langle v_b, s_b\rangle \in T_{v_x,s_x}$ **do**
5 heap $Q.push(\langle v_b, s_b\rangle, \langle v_d, s_d\rangle, e_b)$.
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** $\langle v_j, s_j\rangle \notin T_{v_x,s_x}$ **then**
10 Add $\langle v_j, s_j\rangle$ into T_{v_x,s_x} 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** $s_x = s_0$ *and* $s_j \in A_R.F$ **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 $Q.push(\langle v_j, s_j\rangle, \langle v_q, s_q\rangle, e')$

timestamp, namely $Min\{\langle v_i, s_i\rangle.ts, e.ts\}$. [2] For each node $\langle v_j, s_j\rangle$ it finds from source node $\langle v_i, s_i\rangle$ through edge e in the search, there is a path p from $\langle v_x, s_x\rangle$ to $\langle v_j, s_j\rangle$, which is the concatenation of the path from $\langle v_x, s_x\rangle$ to $\langle v_i, s_i\rangle$ in T_{v_x,s_x} and e. And $p.ts = Min\{\langle v_i, s_i\rangle.ts, e.ts\}$. There are 3 cases:

(1) If $\langle v_j, s_j\rangle \notin T_{v_x,s_x}$, S-PATH adds it into T_{v_x,s_x} with parent $\langle v_i, s_i\rangle$ and timestamp $p.ts$ (line 9-11).
(2) If $\langle v_j, s_j\rangle \in T_{v_x,s_x}$ 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 $\langle v_j, s_j\rangle \in T_{v_x,s_x}$ and $\langle v_j, s_j\rangle.ts \geqslant p.ts$, S-PATH prunes this search branch (line 15-16).

Besides, if $s_x = s_0$ and 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 $(v_x, v_j).ts < \langle v_j, s_j\rangle.ts$. Note that $s_x = s_0$ always holds in S-PATH, but it is not necessary in LM-SRPQ, which we will discuss in Section 3.

Expire: S-PATH carries out an expiration 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. At last, it deletes all result tuples with timestamps smaller than $\tau - N$ in the result set RS.

[2] [26] uses depth-first search (DFS) here. But according to our experiments, Dijkstra-based search is much faster. Because it guarantees that we visit each edge only once in the search, and the complexity is $O(|E|log(|V|))$. While in DFS we may visit the same edge multiple times through different paths, with a search cost of $O(|E|^2)$. 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, like the subtree of $\langle v_2, s_1\rangle$ in Figure 2. We can reduce memory usage by eliminating redundant storage of these subtrees. Moreover, by reducing the forest size, we also reduce the maintenance cost and further improve the update speed.

3 LM-SRPQ ALGORITHM

3.1 Overview

In this section, we propose LM-SRPQ algorithm. Our idea is to eliminate the redundant storage by merging common subtrees in the Δ tree forest of S-PATH. These common subtrees are induced by common fragments in different paths. For example, in Figure 3, the subtree of $\langle v_2, s_1\rangle$ in T_{v_1,s_0} stores paths from $\langle v_2, s_1\rangle$ to $\langle v_7, s_0\rangle$ and $\langle v_9, s_0\rangle$, which are fragments of paths from tree root $\langle v_1, s_0\rangle$ to $\langle v_7, s_0\rangle$ and $\langle v_9, s_0\rangle$, respectively. It is similar in T_{v_6,s_0}. The same subtree of $\langle v_2, s_1\rangle$ in T_{v_1,s_0} and T_{v_6,s_0} are induced by the same path fragments. We can use $\langle v_2, s_1\rangle$ as a landmark and split these paths, as shown in Figure 3. The common postfix $\langle v_2, s_1\rangle \rightarrow \langle v_7, s_0\rangle$ and $\langle v_2, s_1\rangle \rightarrow \langle v_9, s_0\rangle$ can be stored in an independent Δ tree with root $\langle v_2, s_1\rangle$, and in T_{v_1,s_0} (or T_{v_6,s_0}), we only need to store the prefix, namely the path from $\langle v_1, s_0\rangle$ (or $\langle v_6, s_0\rangle$) to $\langle v_2, s_1\rangle$. We can recover the full path by concatenating the prefix and the postfix. For example, timestamp of the full path from $\langle v_1, s_0\rangle$ to $\langle v_7, s_0\rangle$ can be computed as $Min\{T_{v_1,s_0}.\langle v_2, s_1\rangle.ts, T_{v_2,s_1}.\langle v_7, s_0\rangle.ts\}$. After such modification, we do not need to store duplicated subtrees of $\langle v_2, s_1\rangle$.

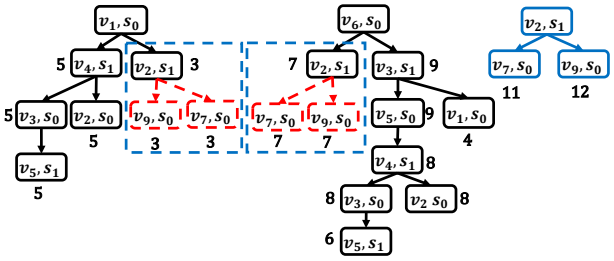


Figure 3: Merge common subtrees

More generally, we select a group of nodes as landmarks and split paths in the product graph into fragments with these landmarks. We name path fragments that pass no landmarks as **local paths**. We build Δ trees for a node $\langle v_i, s_i\rangle$ if $s_i = s_0$ or $\langle v_i, s_i\rangle$ is a landmark. In this Δ tree we maintain the latest local paths from $\langle v_i, s_i\rangle$ to other nodes. In the following sections, we use "local paths" to implicitly denote the latest local paths stored in Δ trees, as other local paths will not be considered in our algorithm. We name the Δ trees whose roots are landmarks as landmark trees (LM trees for short), and name other Δ trees whose roots have initial state s_0 as normal trees.

Figure 4 shows an example of our LM-SRPQ algorithm. In this example, we select $\langle v_1, s_0\rangle$, $\langle v_2, s_1\rangle$, $\langle v_3, s_0\rangle$, $\langle v_4, s_1\rangle$ and $\langle v_6, s_0\rangle$ as landmarks. We present detailed LM tree T_{v_2,s_1} and normal tree T_{v_8,s_0} as examples (the TI-map will be discussed in Section 3.3, and we ignore it here). Landmarks in Δ trees are marked in shadow.

We have the following theorem about local paths:

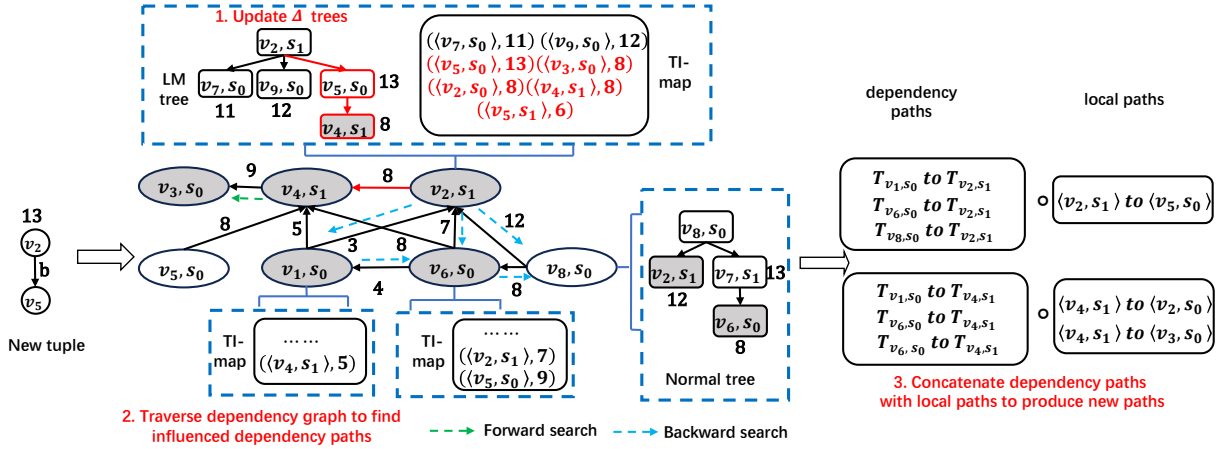


Figure 4: Overview of LM-SRPQ

Theorem 3.1. *For any node pair $\langle v_i, s_0\rangle$ and $\langle v_j, s_f\rangle$, we can get the latest path between them by concatenating a sequence of local paths $\{lp_0, lp_1 ....lp_n\}$, where the source node of lp_0 is $\langle v_i, s_0\rangle$ and the destination node of lp_n is $\langle v_j, s_f\rangle$. Other endpoints are landmarks, and each $lp_c (0 \leqslant c \leqslant n)$ is the latest local path between its endpoints.*

Detailed proof can be found in the technique report [1]. Note that there may be multiple landmark sequences connecting a node pair, and we need to find the latest one. According to this theorem, we can answer RPQ by maintaining latest local paths with Δ trees and continuously computing local path sequences that connect each node pair $\langle v_i, s_0\rangle$ and $\langle v_j, s_f\rangle$.

In order to maintain local paths, we can update LM trees and normal trees with Algorithm 1. The only difference is that we prune the search branch when we meet a landmark in graph traversal. In order to continuously compute concatenations of local paths, we build a dependency graph G_d, where each node represents a Δ tree, and each edge represents the local path between two tree roots. An example of the dependency graph can be found in Figure 4, where we mark LM trees in shadow. G_d is gradually built along with the streaming graph's update. When new streaming graph tuples arrive, Δ trees are updated and new local paths are built, inducing new edges in G_d. Edges in G_d expire when their timestamps fall out of the sliding window. Each path in G_d corresponds to a concatenation of local paths, and we call it a **dependency path**. A local path sequence connecting $\langle v_i, s_0\rangle$ with $\langle v_j, s_f\rangle$ can be split into a dependency path from T_{v_i,s_0} to an LM tree T_{v_q,s_q} and a local path in T_{v_q,s_q} which leads to $\langle v_j, s_f\rangle$. For example, in Figure 1, there is a the path from $\langle v_6, s_0\rangle$ to $\langle v_2, s_0\rangle$: $p = \langle v_6, s_0\rangle \rightarrow \langle v_3, s_1\rangle \rightarrow \langle v_1, s_0\rangle \rightarrow \langle v_4, s_1\rangle \rightarrow \langle v_2, s_0\rangle$. This path can be divided into a local path sequence $\{lp_1, lp_2, lp_3\}$ where $lp_1 = \langle v_6, s_0\rangle \rightarrow \langle v_3, s_1\rangle \rightarrow \langle v_1, s_0\rangle$, $lp_2 = \langle v_1, s_0\rangle \rightarrow \langle v_4, s_1\rangle$, $lp_3 = \langle v_4, s_1\rangle \rightarrow \langle v_2, s_0\rangle$. While in Figure 4, lp_1, lp_2 and lp_3 are stored in tree T_{v_6,s_0}, T_{v_1,s_0} and T_{v_4,s_1}, respectively. Among them, lp_1 and lp_2 can be represented as edges $\overrightarrow{T_{v_6,s_0}, T_{v_1,s_0}}$ and $\overrightarrow{T_{v_1,s_0}, T_{v_4,s_1}}$. Therefore, p_1 can be divided into the dependency path $T_{v_6,s_0} \rightarrow T_{v_1,s_0} \rightarrow T_{v_4,s_1}$ and lp_3 in T_{v_4,s_1}.

we can maintain sequences of local paths by traversing the dependency graph to find dependency paths, and concatenating dependency paths with local paths. More specifically, we process a new tuple in the streaming graph with the following steps:

First, we update the Δ trees with Algorithm 1 but stop graph traversal at landmarks. When a landmark $\langle v_j, s_j\rangle$ is added (or updated) in a Δ tree T_{v_i,s_i} with timestamp t, we add a new edge $\overrightarrow{T_{v_i,s_i}, T_{v_j,s_j}}$ with timestamp t into the dependency graph. In Figure 4, a new streaming graph edge $\overrightarrow{v_2, v_5}$ with label b arrives, and a new local path in T_{v_2,s_1} is built, namely $\langle v_2, s_1\rangle \rightarrow \langle v_5, s_0\rangle \rightarrow \langle v_4, s_1\rangle$. As $\langle v_4, s_1\rangle$ is a landmark, a new dependency edge $\overrightarrow{T_{v_2,s_1}, T_{v_4,s_1}}$ is added.

Second, we traverse the dependency graph to find dependency paths influenced by this update, which can be divided into two kinds: 1) dependency paths from a tree T_{v_i,s_0} to T_{v_q,s_q}, where T_{v_q,s_q} is an updated LM tree. For example, in Figure 4, we need to find the dependency paths from T_{v_6,s_0}, T_{v_1,s_0} and T_{v_8,s_0} to T_{v_2,s_1} with a backward traversal. 2) new dependency paths containing new dependency graph edges. These paths can be found by concatenating paths from precursors to the source node of the new edge, paths from the destination node to its successors, and the new edge itself. For example, in Figure 4, we concatenate path from T_{v_6,s_0} to T_{v_2,s_1}, path from T_{v_4,s_1} to T_{v_3,s_0} and new edge $\overrightarrow{T_{v_2,s_1}, T_{v_4,s_1}}$ to get the new dependency path from T_{v_6,s_0} to T_{v_3,s_0}. Note that we may find multiple dependency paths between the same Δ tree pair in traversal, and we only use the latest one in the update.

At last, we concatenate dependency paths with local paths and update the result set. There are two kinds of concatenations 1) concatenations of new dependency paths with existing local paths 2) concatenations of existing dependency paths with new local paths. For example, in Figure 4, we concatenate the new local path from $\langle v_2, s_1\rangle$ to $\langle v_5, s_0\rangle$ with the dependency path from T_{v_6,s_0} to T_{v_2,s_1} to build new path from $\langle v_6, s_0\rangle$ to $\langle v_5, s_0\rangle$. We also concatenate the new dependency path from T_{v_6,s_0} to T_{v_4,s_1} with the local path from $\langle v_4, s_1\rangle$ to $\langle v_2, s_0\rangle$ to build new path from $\langle v_6, s_0\rangle$ to $\langle v_2, s_0\rangle$. At last, for each new path p from $\langle v_i, s_0\rangle$ to $\langle v_j, s_f\rangle$, we set the timestamp of (v_i, v_j) in the result set RS to $p.ts$ if it is not in RS or has a smaller timestamp. We omit the new dependency paths to T_{v_3,s_0} in Figure

4, as there are no local paths to final-state nodes in T_{v_3,s_0}. We also omit local paths to non-final-state nodes, like $\langle v_5, s_1 \rangle$.

However, there are two problems in the above algorithm:

First, how to select a good landmark set is a challenge. We hope the landmark set helps us to minimize the Δ tree forest, but we also need to consider the update cost. We can minimize the Δ tree forest by selecting all nodes as landmarks. In this case, each Δ tree only stores the 1-hop successors of a node, and the forest becomes an adjacency list of the product graph. There will be no redundant storage. However, the dependency graph will be as large as the product graph, and traversal in it will be terribly slow. Therefore, we hope to select the landmark set with a bounded size m, and try to minimize the Δ tree forest under such restriction. There will be $\sum_{i=1}^{m} \binom{|V_p|}{i}$ methods to select the landmark set in this case, where $|V_p|$ is the number of nodes in the product graph. This is an exponential search space, which makes it computationally impossible to find the optimal landmark set. We have to resort to a greedy method. Moreover, the landmark set needs to be continuously updated to keep up with the streaming graph.

Second, even bounded in size, the dependency graph traversal still brings a high update cost. Without materializing existing paths like Δ trees, we need to build dependency paths from scratch in each update. In the above example, for a new dependency edge $\overrightarrow{T_{v_2,s_1}, T_{v_4,s_1}}$, we need to carry out both forward search from T_{v_4,s_1} and backward search from T_{v_2,s_1}. With a Dijkstra-based method, each search takes a cost of $O(|E_d|log(|V_d|)$, where $|E_d|$ and $|V_d|$ are numbers of edges and vertices in the dependency graph, respectively. As there may be $O(|V_d|^2)$ new dependency edges upon one streaming graph tuple arrival, such traversal will be implemented multiple times. Even when we bound the size of the dependency graph, such traversal cost is still a system bottleneck.

We will solve these two problems in the following sections.

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 continuously update the landmark set to keep up with the updated snapshot graph.

To keep the landmark set updated, we perform a landmark selection algorithm to update the entire landmark set at the end of each sliding interval. Compared to inserting or deleting individual landmarks in real time, such a batch update method will decrease the cost of modifying the Δ tree forest. Besides, at the end of a sliding interval, we will delete expired edges, leading to a dramatic change. We need to check the landmark set after this expiration anyway. Note that the landmark selection is not a total rebuild. We will first check if current landmarks are still qualified, and leave them unchanged if so. The Δ tree forest is modified only based on the difference between the new landmark set and the old one.

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. We try to select a bounded number of "good" landmarks. The cost of selecting a node as a landmark is the size of the LM tree we need to build for it, and the benefit is the size of all subtrees rooted at it in the Δ tree forest, which we will delete. A landmark is good if the gap between its benefit and cost is large. For example, in Figure 3, by selecting $\langle v_2, s_1 \rangle$ as a landmark, we omit the subtrees in T_{v_6,s_0} and T_{v_1,s_0}, but need to build a new LM tree for T_{v_2,s_1}. At last, the forest size is decreased by one node. The gap between the subtree size and LM tree size is hard to predict. The subtree sizes of one node in different Δ trees may be different. For example, in Figure 3, though $\langle v_1, s_0 \rangle$ can reach all the nodes in T_{v_1,s_0}, there are no successors in the subtree of $\langle v_1, s_0 \rangle$ in T_{v_6,s_0}. Because all its successors can be reached from $\langle v_6, s_0 \rangle$ through other paths with larger timestamps. Moreover, selecting one node as a landmark will change the forest, and influence the benefit and cost of other nodes.

As it is time-consuming to compute cost and benefit for all nodes, we choose to select a candidate set with a heuristic method. 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 benefits by merging these subtrees. Then we find nodes with higher benefit than cost in the candidate set as landmarks.

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 $\langle v_i, s_i \rangle$. As product graph paths are guided by state transition in DFA, the depth of Δ tree of $\langle v_i, s_i \rangle$ can be approximated as the maximum length of paths built in a traversal starting from s_i in the DFA. We allow a circle to repeat t times in the traversal, which corresponds to a Kleene star in the regular expression. For example, in the DFA of $(a \circ b)^*$ shown in Figure 1, both state s_0 and s_1 get a score of 4 if we set $t = 2$. Because from each state we can traverse 4 steps ahead, repeating the circles between them by 2 times.

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 that 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 Dependency Graph Traversal

As discussed above, the bottleneck of LM-SPRQ lies in traversing the dependency graph. Without additional indexes, we have to build paths from scratch in each update. The most intuitive method to accelerate such traversal is to build Δ trees in the dependency graph. We call these Δ trees dependency trees, in order to distinguish them with Δ trees that store local paths. We call the LM-SRPQ with dependency trees LM-DF.

To be specific, we build a dependency tree for each T_{v_i,s_0} whose root has an initial state s_0. When a new tuple arrives, we first update Δ trees. In this procedure, we build new local paths and produce dependency graph edges, as discussed in Section 3.1. For each new edge $\overrightarrow{T_{v_i,s_i}, T_{v_j,s_j}}$, we find all the dependency trees containing T_{v_i,s_i}, add T_{v_j,s_j} into them and extend the dependency trees like Algorithm 1. At last, we concatenate the dependency paths with local paths and update the result set as discussed in Section 3.1.

Figure 5 shows the dependency trees influenced by the update in Figure 4. After updating Δ tree T_{v_2,s_1}, we can directly fetch the dependency paths from its precursors T_{v_6,s_0}, T_{v_1,s_0} and T_{v_8,s_0} to it in the dependency trees, and concatenate them with the new local path $\langle v_2, s_1\rangle \rightarrow \langle v_5, s_0\rangle$ in T_{v_2,s_1} to produce new paths. The backward search is omitted. However, we still need to search forward from T_{v_4,s_1} to extend these dependency trees. In the dependency trees of T_{v_6,s_0} and T_{v_1,s_0}, we find existing paths with larger timestamps and prune the search. In the dependency tree of T_{v_8,s_0}, we build new paths to T_{v_4,s_1} and T_{v_3,s_0}. We concatenate these new dependency paths with local paths in T_{v_4,s_1} and T_{v_3,s_0}, and update the result set.

The dependency graph is usually very dense. A lot of dependency trees need to be updated when inserting a new dependency edge, and multiple dependency edges may be induced upon one tuple arrival. We need to carry out a forward search with cost $O(|E_d|log(|V_d|))$ in each of these dependency trees to update them, where $|E_d|$ and $|V_d|$ are the numbers of edges and vertices in the dependency graph. Even with pruning, such update cost is high.

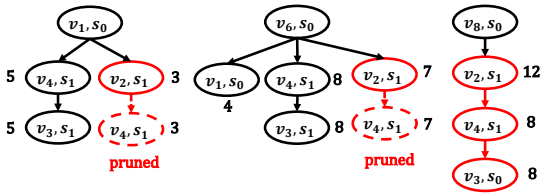


Figure 5: Example of Dependency Forest

We further notice that in the dependency graph, LM trees are much fewer than normal trees, but contribute to the major part of traversal. Because normal trees only have out-edges, and can only become roots (in forward search) or leaves (in backward search) in traversal. Therefore, we propose another acceleration method, which builds indexes with rich information only for LM trees. We build a time information map (TI-map) in each LM tree. It stores timestamps of the latest paths to all successors of the tree root, no matter whether they are in the LM tree or not. In Figure 4, we show the TI-map in tree T_{v_2,s_1} as an example. We process a new streaming graph tuple with the following steps:

Step 1: Upon the insertion of the new tuple, we first update Δ trees with Algorithm 1. Note that we only build local paths and stop traversal at landmarks. We need to update the result set (if the tree root has an initial state) and TI-maps (only for LM trees) in this procedure. In Figure 4, we add tree nodes $\langle v_5, s_0\rangle$ and $\langle v_4, s_1\rangle$ into LM tree T_{v_2,s_1}. Besides, we add $(\langle v_5, s_0\rangle, 13)$ and $(\langle v_4, s_1\rangle, 8)$ into the TI-map of T_{v_2,s_1}. As the root of T_{v_2,s_1} has state s_1 rather than initial state s_0, we do not need to update the result set. We also need to produce new dependency edges when landmarks are added or updated in a Δ tree. In Figure 4, we add a new dependency edge $\overrightarrow{T_{v_2,s_1}, T_{v_4,s_1}}$ as landmark $\langle v_4, s_1\rangle$ is added into T_{v_2,s_1}.

Step 2: For each new dependency edge $de = \overrightarrow{T_{v_i,s_i}, T_{v_j,s_j}}$, we fetch the TI-map of T_{v_j,s_j}. For each $(\langle v_q, s_q\rangle, ts)$ in this TI-map, there is a new path from $\langle v_i, s_i\rangle$ to $\langle v_q, s_q\rangle$ with timestamp $Min\{de.ts, ts\}$. It is a concatenation of de and the latest path from $\langle v_j, s_j\rangle$ to $\langle v_q, s_q\rangle$. We update the result set (if $s_i = s_0$) and the TI-map of T_{v_i,s_i} (if T_{v_i,s_i} is an LM tree) according to the timestamps of these new paths. In Figure 4, we insert $(\langle v_2, s_0\rangle, 8)$, $(\langle v_3, s_0\rangle, 8)$ and $(\langle v_5, s_1\rangle, 6)$ into the TI-map of T_{v_2,s_1}, which are computed with the TI-map of T_{v_4,s_1}

Step 3: For each updated LM tree T_{v_i,s_i}, suppose DP denotes the set of new dependency edges with source T_{v_i,s_i}. We carry out a backward search from T_{v_i,s_i} in the dependency graph. For each Δ tree T_{v_x,s_x} we find in the traversal, suppose the dependency path from it to T_{v_i,s_i} is dp. We can build two kinds of new paths for it: 1) Concatenation of dp with new local paths in T_{v_i,s_i}. 2) Concatenation of new dependency path $dp \circ de_j$ with paths recorded in the TI-map of T_{v_j,s_j} for each $de_j = \overrightarrow{T_{v_i,s_i}, T_{v_j,s_j}}$ in DP. Note that we only compute timestamps of these new paths, as we only record time information in TI-maps. We need to update the result set (if $s_x = s_0$) and the TI-map of T_{v_x,s_x} (if T_{v_x,s_x} is an LM tree) according to the timestamps of these new paths.

In Figure 4, when we travel from T_{v_2,s_1} to T_{v_1,s_0} following a dependency path $dp = T_{v_1,s_0} \rightarrow T_{v_2,s_1}$ with timestamp 3. We first concatenate dp with new local paths in T_{v_2,s_1} and produce a group of new paths. Then we build the new dependency path from T_{v_1,s_0} to T_{v_4,s_1} as $dp' = dp \circ \overrightarrow{T_{v_2,s_1}, T_{v_4,s_1}}$ with timestamp 3. We further concatenate dp' with paths starting from $\langle v_4, s_1\rangle$. For each each $(\langle v_q, s_q\rangle, ts)$ in the TI-map of T_{v_4,s_1}, timestamp of the new path from $\langle v_6, s_0\rangle$ to $\langle v_q, s_q\rangle$ is $Min\{dp'.ts, ts\}$.

The TI-map in each LM tree saves us from forward search in the dependency graph. Moreover, it pre-computes some concatenations of dependency paths with local paths, which further accelerates the update. For example, the TI-map of T_{v_2,s_1} in Figure 4 not only records timestamps of local paths in T_{v_2,s_1}, but also concatenations of dependency paths to T_{v_4,s_1} and T_{v_3,s_0} with local paths in these trees. Though we need to maintain these TI-map entries, its benefit outweighs the cost.

The above procedure is the basic update algorithm. We can further perform 4 kinds of prune with TI-maps. The first 3 kinds are performed in the backward search in Step 3. When we arrive at an LM tree T_{v_x,s_x} from T_{v_i,s_i} following a dependency path dp, we perform the following pruning:

First, we compare the timestamp of $\langle v_i, s_i\rangle$ in TI-map of T_{v_x,s_x} with $dp.ts$. If $dp.ts$ is smaller, it is not the latest dependency path, and we can prune this search branch. With this prune, we can use depth-first search (DFS) to replace Dijikstra-based method in the

backward search, saving the cost of heap maintenance without inducing additional cost in wrong search branches. In the backward search from T_{v_2,s_1} in Figure 4, if we arrive at T_{v_6,s_0} following the dependency path $dp = T_{v_6,s_0} \rightarrow T_{v_1,s_0} \rightarrow T_{v_2,s_1}$ in a DFS, we find its timestamp 3 is smaller than the timestamp of $\langle v_2, s_1\rangle$ in the TI-map of T_{v_6,s_0}, which is 7. It indicates that there is another dependency path with a larger timestamp, and we can prune the search branch.

Second, we compare timestamp of $\langle v_b, s_b\rangle$ in TI-map of T_{v_x,s_x} with $Min\{dp.ts, T_{v_i,s_i}.\langle v_b, s_b\rangle.ts\}$. $\langle v_b, s_b\rangle$ is the source node of the new product graph edge which triggers the update in T_{v_i,s_i}. If the timestamp in the TI-map is larger, we can prune this search branch. A similar check is performed for $\langle v_d, s_d\rangle$, the destination node of the new product graph edge. When the timestamp of $\langle v_d, s_d\rangle$ in the TI-map is no smaller than the new path, we prune the branch. In Figure 4, the new product graph edge is $\overrightarrow{\langle v_2, s_1\rangle, \langle v_5, s_0\rangle}$. When we arrive at T_{v_6,s_0} following the dependency path $dp = T_{v_6,s_0} \rightarrow T_{v_2,s_1}$ in the backward search, the new path to $\langle v_5, s_0\rangle$ is $p = dp \circ \overrightarrow{\langle v_2, s_1\rangle, \langle v_5, s_0\rangle}$, with timestamp 7. We find the timestamp of $\langle v_5, s_0\rangle$ in the TI-map of T_{v_6,s_0} is 9, indicating that there is an existing path p' with a larger timestamp. Check the product graph in Figure 1, and we can find $p' = \langle v_6, s_0\rangle \rightarrow \langle v_3, s_1\rangle \rightarrow \langle v_5, s_0\rangle$. All the new paths we can build with source $\langle v_6, s_0\rangle$ contain fragment p. By replacing p with p' we can get existing paths with no smaller timestamps. Therefore, all these new paths can not generate effective updates to the result set, and we can prune this branch.

Third, for each $de_j = \overrightarrow{T_{v_i,s_i}, T_{v_j,s_j}}$ in DP, we compare timestamp of $\langle v_j, s_j\rangle$ in the TI-map of of T_{v_x,s_x} with $Min\{dp.ts, de_j.ts\}$. If the timestamp in TI-map is no smaller, we do not need to check the TI-map of T_{v_j,s_j}. In Figure 4, when we arrive at T_{v_1,s_0} following the dependency path $dp = T_{v_1,s_0} \rightarrow T_{v_2,s_1}$, the new dependency path we build from T_{v_1,s_0} to T_{v_4,s_1} is $dp' = dp \circ \overrightarrow{T_{v_2,s_1}, T_{v_4,s_1}}$ with timestamp 3. We find the timestamp of $\langle v_4, s_1\rangle$ in the TI-map of T_{v_1,s_0} is 5, indicating that there is an existing dependency path with a larger timestamp 5. We do not need to concatenate dp' with paths recorded in the TI-map of T_{v_4,s_1} anymore, as they cannot have larger timestamps than existing paths, and cannot influence the result set.

The last kind of pruning is conducted in local path building in Step 1. When we try to add a node to an LM tree in Algorithm 1, we first check the TI-map. If the node is already in the TI-map and its timestamp is no smaller than the new local path, there is already an existing path to this node, and we can not find a new path with a larger timestamp. Therefore, we prune the search branch.

The above pruning can also be used in normal trees, We can compute timestamp of the latest path to a specific node by checking TI-maps of all landmarks in this normal tree. As such computation is a bit costly, we only use it in the backward search pruning.

The expiration of LM-SRPQ is similar to S-PATH, except that we need to delete expired TI-map entries and dependency graph edges besides expired Δ tree nodes.

The second and the fourth kinds of pruning utilize timestamps of non-landmark nodes, which can be only achieved with TI-maps. With these pruning tricks, we can guarantee that we visit each edge and each node in the dependency graph at most once when inserting a product graph edge, no matter how many dependency edges are induced by it. We prove this in detail in the technical report [1]. While LM-DF does not have this guarantee and suffers from high forward search cost in multiple dependency trees. The experiments in Section 4.6 show that TI-map-based method is faster than LM-DF, and we use it as the default version of LM-SRPQ.

Table 2: Queries used in experiments

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	$(a_1 + a_2 + ... + a_k)^*$

Compared with S-PATH, though we need additional dependency graph traversal and path concatenation, we greatly decrease the cost of updating Δ trees. For example, if the update in Figure 4 happens in S-PATH, we need to update the subtree of $\langle v_2, s_1\rangle$ with a graph traversal three times in T_{v_1,s_0}, T_{v_6,s_0} and T_{v_8,s_0}. While in LM-SRPQ, we only need to update T_{v_2,s_1} once. In the three precursor trees, we produce new results by path concatenations, which are in fact computations with integer-type timestamps. This procedure is much faster than updating Δ trees with a graph traversal. As a result, the overall processing speed improves in most cases. Our experimental results confirm this. We have the following theorem about the time and space cost of LM-SRPQ and S-PATH:

Theorem 3.2. *LM-SRPQ and S-PATH both have space cost of $O(k|V|^2)$ The cost of each update operation of both algorithms is $O(k^2|V||E|log(k|V|))$, and the cost of each expiration operation is $O(k|V|^2)$. k is the number of states in the DFA. $|V|$ and $|E|$ are the number of vertices and edges in the snapshot graph, respectively.*

Specifically, in each update, the cost of updating Δ trees in LM-SRPQ is $O(k^2|V||E|log(k|V|))$, the cost of dependency graph traversal is $O(k|E_d|)$, and the cost of update TI-maps is $O(k^2|V|^2)$. Without TI-maps, the dependency graph traversal cost will be $O(k|V|^2 \times |E_d|log(|V_d|))$. $|E_d|$ and $|V_d|$ are the number of edges and nodes in the dependency graph, respectively. $|E_d| = O(|V|^2)$ and $|V_d| = O(|V|)$. Detailed proof can be found in the technical report [1]. We omit it here due to space limitaion.

4 EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate LM-SRPQ over two real-world datasets and one synthetic dataset. Details about the datasets, workloads and experimental settings are discussed in Section 4.1 and Section 4.2, respectively. In Section 4.3 and Section 4.4, we compare LM-SRPQ with prior art S-PATH algorithm [26] on both memory usage and processing speed. In Section 4.5, we evaluate the scalability of LM-SRPQ and S-PATH. In Section 4.6, we carry out an ablation study to evaluate the effect of different techniques.

4.1 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 [6], plus a popular non-recursive query Q4 abc [3]. We set $k = 3$ for queries with a variable number of labels, as there are

[3] there is another query $(a_1 + a_2 + ... + a_k)^+$ used in [25]. As we omit self-join results, its answer has no difference with Q10 $(a_1 + a_2 + ... + a_k)^*$. Thus we omit this query.

only 3 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 datasets 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. We set the window size to 20 days and sliding interval to 1 day.

LDBC: This is a synthetic dataset that simulates real-world interactions in social networks [12]. The 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. 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 the sliding interval to 6 hours.

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 shuffle 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 the sliding interval to $256k$ edges.

4.2 Experiment Implementation

Both S-PATH and LM-SRPQ are implemented with C++ and compiled with GCC 5.4.0 and O3 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 a traversal 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.

4.3 Memory Comparison

We measure the average memory usage of LM-SRPQ and S-PATH, as shown in Figure 6. 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. As LM-SRPQ depends on merging common subtrees to save memory, the number and size of common subtrees influence the improvement significantly. 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-maps and dependency edges 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.

4.4 Processing Speed Comparison

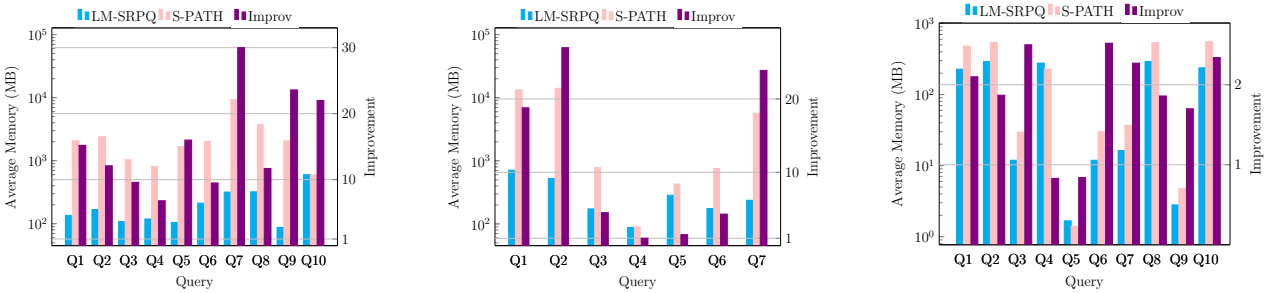
We measure the throughput of edge insertions, and the results are shown in Figure 7. The unit is edge per second (eps). We also present the improvement brought by LM-SRPQ. The improvement is calculated as $\frac{throughput\ of\ LM-SRPQ}{throughput\ of\ S-PATH}$. The experiment settings are the same as the experiments in Section 4.3.

As shown in these figures, LM-SRPQ outperforms S-PATH in most queries. The improvement of throughput reaches at most 4.5. By merging common subtrees, LM-SRPQ decreases the Δ tree forest size a lot, and decreases the cost to update these Δ trees as well. Though it induces new cost of dependency graph traversal and path concatenation, the pruning techniques have strictly bounded the cost. As a result, LM-SRPQ has a higher overall speed. 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.

4.5 Scalability Evaluation

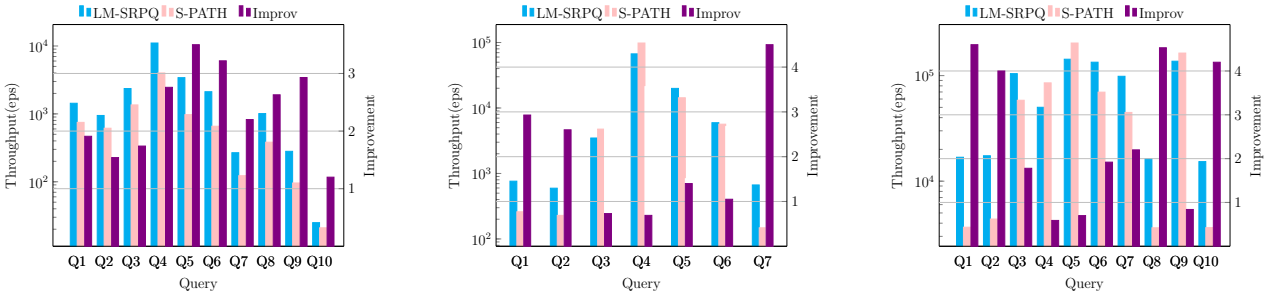
In this section, we vary the window size to test the scalability of LM-SRPQ and S-PATH. The experimental results are shown in Figure 8. The dataset we use in this experiment is StackOverflow. We choose Q1, Q4 and Q8 as representative queries in order to keep the figure legible. These queries cover three 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. As

[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/



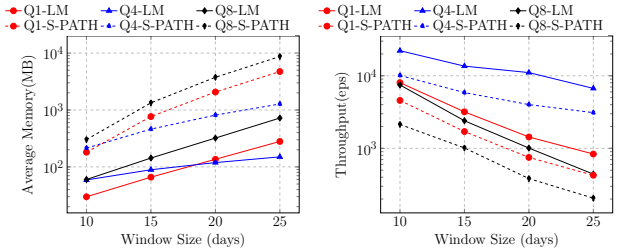
(a) StackOverflow (b) LDBC (c) Yago2s

Figure 6: Average memory usage comparison



(a) StackOverflow (b) LDBC (c) Yago2s

Figure 7: Throughput comparison



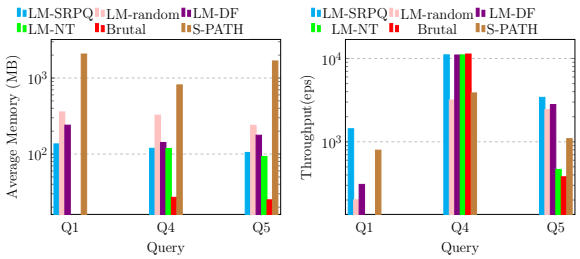
(a) Average Memory Usage (b) Throughput

Figure 8: Performance vary with window size

Theorem 3.2 shows, the memory and time cost of both algorithms have a square relationship with the snapshot graph size, and the average snapshot graph size grows linearly with the window size. 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.

4.6 Ablation Study

In this section, we evaluate the effect of different techniques, including TI-maps and the landmark selection algorithm. We implement several variants of LM-SRPQ. The first variant is LM-SRPQ with no TI-maps, denoted as LM-NT. In the second variant, we randomly select a landmark set at the end of each sliding interval. To be specific, we first find candidate nodes that show up in at least 2 Δ trees. Then we randomly select 20% nodes from the candidate set as landmarks. We call this variant LM-random. We also compare LM-SRPQ with the dependency forest-based algorithm LM-DF and a brutal search algorithm that only materializes the product graph and searches new paths from scratch upon each tuple arrival. We use dataset StackOverflow. The brutal search algorithm and LM-NT can finish processing in one week only in Q4 and Q5. Therefore we present the result of these 2 queries, plus another query Q1. The experimental results are shown in Figure 9. Note that the brutal search algorithm and LM-NT have no query result in Q1 as they cannot finish processing in time.



(a) Average Memory Usage (b) Throughput

Figure 9: Performance of different variant

The result shows that LM-NT, LM-DF and the brutal search algorithm have lower throughput than LM-SRPQ in most cases. However, Q4 is a special case, where the throughput of these variants is compatible with LM-SRPQ. The major cost in these variants is graph traversal. In Q4 there are no Kleene stars. The lengths of paths are no more than 3, and the graph traversal cost is bounded. But according to [6], most popular RPQ queries contain Kleene stars, and these variants are slow in those queries. The memory usage of the brutal search algorithm is always the smallest, as it ensures that each node is only stored once. LM-NT also has smaller memory usage than LM-SRPQ, as it stores no TI-maps. But as discussed above, these two variants suffer from low throughput. The memory usage of LM-DF is larger than LM-SRPQ, because dependency trees consume more memory than TI-maps. Besides, LM-random always has larger memory usage and lower throughput than LM-SRPQ, confirming the effect of our landmark selection algorithm.

We break down the memory usage of LM-SRPQ in the above 3 queries in Table 3. TI-maps may contribute to the major part of memory usage in complex queries like Q1, but we think it is a necessary cost. Because the throughput will decrease by orders of magnitudes without TI-maps according to Figure 9. We also show the average latency of each landmark selection and the ratio of total landmark selection time to the total processing time in Table 4. The result confirms that our landmark selection is fast.

Table 3: Break down memory usage of LM-SRPQ

Query	Normal trees (MB)	LM trees (MB)	TI-maps (MB)
Q1	28.5	6.9	84.8
Q4	75.3	23.6	5.4
Q5	49.5	20.9	17.5

Table 4: Landmark selection time in LM-SRPQ

Query	Average latency(s)	Ratio of landmark selection time to total processing time
Q1	0.198	1.2%
Q4	0.217	6.1%
Q5	0.164	1.7%

5 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 [3, 15, 19] and graph partition [7, 28, 33], 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 [9, 18, 22], triangle counting [14, 31, 35], cycle detection [29], as well as building dynamic indexes to support adhoc queries like connectivity [8, 30, 37].

Regular Path Query (RPQ): RPQ has been widely used in graph query languages and systems [4, 5, 13]. 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 [23]. Due to such high complexity of simple-path RPQ, most works use arbitrary path semantics.

Former RPQ algorithms can be divided into two kinds: automaton-based approaches and algebra-based approaches. For the first kind, there are G [10], 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. [20] which builds two automatons and performs bi-directional search with them. Koschmieder et al. [21] propose an algorithm that splits the RPQ into fragments with rare labels and performs bi-directional search for each fragment. Inju Na et.al. [24] decrease the cost of graph traversal in RPQ evaluation by merging strongly connected components in the graph. A recent work [34] also approximately answers RPQ with random walks. The representative work of the second kind is α-RA based method [2], which extends traditional relational algebra with α operator for transitive closure computation. This method has been widely used in various SPARQL engines [13]. Yakovets et.al.[36] 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 [25], 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 [26], they further extend the definition of persistent RPQ and merge it into an algebra for complex queries in streaming graphs. The algorithm for streaming RPQ in [26], named S-PATH, maintains time information for results to support further analysis. In this paper, we focus on arbitrary path semantics, and maintain time information for query results like [26]. But we use sliding window model to define the validity of edges, paths and query results, rather than the validity interval model in [26].

6 CONCLUSION

In this paper, we propose a novel algorithm for persistent RPQ in streaming graphs, named LM-SRPQ. It is built on the foundation of prior art, which transforms RPQ problem in the snapshot graph to a reachability problem in a product graph, and builds Δ trees to materialize paths in the product graph. 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 30 times smaller than prior art and the throughput is at most 4.5 times higher, and its superiority is more significant in complex queries and dense graphs.

ACKNOWLEDGMENTS

This work was supported by the Research Grants Council of Hong Kong, China, No.14205520, and NSFC grants under U20A20174, 61932001.

REFERENCES

[1] 2023. *Source code and Technical Report of LM-SRPQ.* https://github.com/StreamingTriangle/LM-SRPQ

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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).

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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).

[16] 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)*.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.