MWP: Multi-Window Parallel Evaluation of Regular Path Queries on Streaming Graphs

SIYUAN ZHANG, School of Computer Science, Fudan University, China
ZHENYING HE, School of Computer Science, Fudan University, China
YINAN JING, School of Computer Science, Fudan University, China
KAI ZHANG, School of Computer Science, Fudan University, China
X. SEAN WANG, School of Computer Science, Fudan University, China

A persistent Regular Path Query (RPQ) on a streaming graph is to continuously find every pair of vertices that are connected by a path in the graph within a sliding window, such that the edge label sequence of this path matches a given regular expression. The existing RPQ evaluation algorithm in the literature incrementally maintains a set of spanning-tree-like data structures to quickly form query results and to avoid reprocessing edges that are shared by multiple sliding windows. This approach allows parallel processing of the graph edges within a sliding window but requires a blocking expiration phase between sliding windows to remove the old edges. This blocking phase can significantly degrade the query performance, especially when the edges arrive quickly and the sliding windows overlap significantly.

This paper presents a new RPQ evaluation strategy called Multi-Window Parallel (MWP) method leveraging a new data structure called Timestamped Rooted Digraph (TRD). The novel idea is to incrementally maintain TRDs for the quick formulation of query results, like the aforementioned spanning trees, but simultaneously contain needed information for multiple sliding windows. MWP eliminates the forced blocking expiration phase. Only when memory runs low, a quick "dirty garbage collection" (DGC) process is done to remove some unneeded edges and nodes on TRDs, without incurring large costs. Extensive experiments on real graph datasets show that MWP significantly outperforms the existing algorithm in terms of throughput, tail latency, and scalability, and that DGC provides an effective solution for releasing memory with minimum impact.

CCS Concepts: • **Information systems** → *Data management systems.*

Additional Key Words and Phrases: Regular Path Queries; Streaming Graphs; Parallel Processing; Persistent Query Evaluation

ACM Reference Format:
Siyuan Zhang, Zhenying He, Yinan Jing, Kai Zhang, and X. Sean Wang. 2024. MWP: Multi-Window Parallel Evaluation of Regular Path Queries on Streaming Graphs. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 5 (February 2024), 26 pages. https://doi.org/10.1145/3639260

1 INTRODUCTION

A *streaming graph* takes the form of a sequence of edges that arrive over time, representing relationships between entities in various fields. It is widely applied in real-time analysis applications

Authors' addresses: Siyuan Zhang, siyuanzhang17@fudan.edu.cn, School of Computer Science, Fudan University, Shanghai, China; Zhenying He, zhenying@fudan.edu.cn, School of Computer Science, Fudan University, Shanghai, China; Yinan Jing, jingyn@fudan.edu.cn, School of Computer Science, Fudan University, Shanghai, China; Kai Zhang, zhangk@fudan.edu.cn, School of Computer Science, Fudan University, Shanghai, China; X. Sean Wang, xywangCS@fudan.edu.cn, School of Computer Science, Fudan University, Shanghai, China.

ACM 2836-6573/2024/2-ART5
https://doi.org/10.1145/3639260

such as recommendation systems [16, 21, 40], social network analysis [11, 22, 34], and financial risk monitoring [39]. For example, merchant fraud detection in e-commerce may have buyers and sellers represented as vertices and payment activities represented as edges. In the context of streaming graphs, the persistent Regular Path Query (RPQ) evaluation is to continuously find pairs of different vertices connected by paths whose label sequences match a user-given regular expression [38]. The RPQ evaluation on the streaming graph usually adopts a *time-based sliding window model* [12] to only consider the edges belonging to the same sliding window.

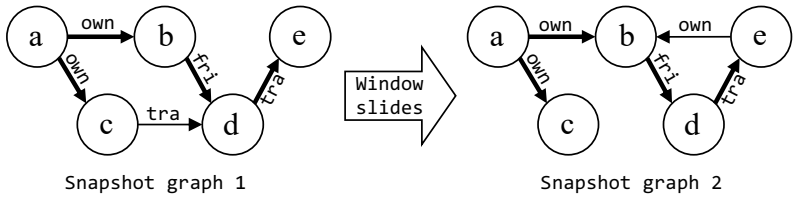


Fig. 1. The snapshots of a streaming graph in two neighboring windows. The thicker edges are shared edges between the two snapshots.

Consider the RPQ with query regular expression $R = (\texttt{own} \circ \texttt{fri} \circ \texttt{tra})$ on a streaming graph. Figure 1 shows two snapshots of the streaming graph with edges in two neighboring sliding windows. Note that the two snapshots share some edges shown as thicker arrows. For the "snapshot graph 1" (the left graph in Figure 1), the label sequence of the path $a \rightarrow b \rightarrow d \rightarrow e$ is "`own, fri, tra`" and matches R. Thus, the vertex pair (a, e) is a query result in this window. Similarly, we get the result (a, e) from the graph in the next window shown as "snapshot graph 2."

To enable efficient RPQ evaluation on streaming graphs, Pacaci et al. [38] proposed an algorithm (we call it the PBO algorithm in this paper and described in more detail in Section 2). PBO incrementally maintains a set of query-spanning trees to construct query results as the edges arrive. The key idea is to allow the reuse of the data elements (such as those for the edges marked as thicker arrows in Figure 1) from the previous window to avoid repeated processing of edges across multiple windows. At the end of a sliding window, the query results can be collected from the query-spanning tree, and the edges that are not in the next sliding window are removed from the query-spanning trees (we call this removal process "expiration").

The PBO algorithm is generally quite efficient but has a significant bottleneck with its expiration process. Indeed, while edges that arrive before the end of a sliding window may be processed in parallel, all edges belonging to the next sliding window (but not in the current window) must wait until the expiration process ends. This is a blocking phase of the algorithm, which greatly degrades the performance, especially when the edges arrive quickly and the sliding windows overlap significantly. The experiments in Section 5 confirm this observation.

In this paper, we propose a new streaming query strategy named *Multi-Window Parallel* (MWP) method inspired by the concept of *Multi-Version Concurrency Control* [30] in databases. The idea of MWP is to provide a data structure that contains multiple versions corresponding to multiple windows, eliminating the blocking phase between sliding windows. The new query data structure is called Timestamped Rooted Digraph (TRD). Each TRD records all the paths in the streaming graph whose label sequence (partially) matches the regular expression. This is similar to that of the query spanning tree in [38], but each node in TRDs follows the MWP strategy to record the paths from the root to itself in multiple sliding windows. Although TRDs record multiple versions, the data structure is quite efficient in avoiding repeated processing of edges. Indeed, the amortized time complexity of insertion for each edge in MWP is the same as that of PBO. By eliminating the

blocking phase, MWP provides more opportunities for parallel processing and outperforms PBO significantly, as will be shown in Section 5.

Though the MWP strategy eliminates the forced blocking expiration phases, memory consumption still needs to be managed. We do it with a form of "garbage collection" (GC) that prunes out the edges in TRDs that are surely no longer involved in all future query results. This GC may be executed less frequently than the expiration process in PBO. Furthermore, we propose a "dirty GC" (DGC) method. The novel idea is to only prune edges based on a simple sufficient condition and leave the TRDs in a bit of "dirty" but correct state. The theoretical worst-time complexity of DGC is shown to be lower than that of the expiration process in PBO. In practice, along with the fact that DGC only needs to be sporadically executed rather than must be done between every two neighboring sliding windows as in PBO, DGC can significantly free up memory without overly impacting query efficiency, as shown in Section 5.

To the best of our knowledge, MWP is the first window-based streaming query strategy that leverages multi-versioning to address the blocking phase between sliding windows. It enables a non-blocking, highly parallel implementation for maintaining query structures for persistent streaming queries. This strategy may provide a solution for similar streaming query algorithms based on the sliding window model, such as cycle detection [39], aggregate query [33], triangle counting [20] and path query [38]. We discuss further extensions of MWP in Section 6.

Our contributions are highlighted as follows:

- We propose a novel strategy called *Multi-Window Parallel* method that efficiently records multi-versions of the query structures for multiple sliding windows.
- We propose a "dirty garbage collection" strategy for fast execution of garbage collection to limit the memory usage.
- We implement our proposed MWP method and conduct extensive experiments to confirm its efficiency.

The remainder of this paper is structured as follows. Section 2 presents the problem definition and briefly describes PBO. Section 3 provides a detailed description of the TRDs and the MWP strategy. Section 4 gives the process of DGC on TRDs. Experimental results and detailed analysis are given in Section 5. Section 6 describes the related work. Finally, Section 7 summarizes the paper.

2 PRELIMINARIES

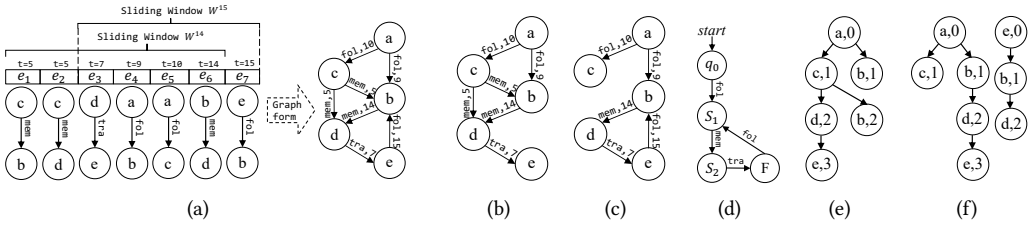


Fig. 2. (a) a streaming graph S (the left part) consisting of two sliding windows W^{14} and W^{15} (marked in the graph) with window size $|W| = 10$. The right part in (a) is its graph representation by connecting the same vertices of tuples in S, (b) the snapshot graph of S in sliding window W^{14} marked in (a), note that W^{14} with $|W| = 10$ only contains the tuples whose timestamp is in $(4, 14]$, (c) the snapshot graph of S in sliding window W^{15}, (d) a DFA for query expression $R = (fol \circ mem \circ tra)^+$, (e) & (f) the spanning trees generated by the snapshot graphs in (b) & (c) respectively.

In this section, we provide a formal definition of the streaming graph and the RPQ evaluation task. We also introduce the PBO algorithm with its query-spanning trees [38]. For simplicity, we focus here on an append-only input stream where tuples are added to the graph without explicit deletion. Explicit deletion on streaming graphs will be discussed in Section 4.4.

Definition 1. ***Steaming Graph**: A* streaming graph *S is a sequence of tuples $S = \tau_1, \tau_2, \cdots, \tau_n$, where each tuple $\tau_i = (e_i, ts(\tau_i))$ indicates a directed edge $e_i = (v_x, v_y, \phi(e_i))$ from the* from-vertex *v_x to the* to-vertex *v_y with label $\phi(e_i)$ and timestamp $ts(\tau_i)$. The domain of the timestamp is the positive integers. The sequence continuously expands as the new tuples arrive. For each pair of tuples τ_i and τ_j, where $i \leq j$, $ts(\tau_i) \leq ts(\tau_j)$. Edges between the same pair of vertices v_x and v_y with different labels can co-exist, possibly with different timestamps.*

An example of a streaming graph is shown in Figure 2(a). It is worth noting that the edges in the streaming graph are directed, labeled, and may form multiple paths between the same two vertices. To provide a precise definition, we introduce the concept of a graph path and path label.

Definition 2. ***Graph path and Path label**: A* graph path *that connects vertices v_x and v_y is a sequence of edges $p = e_1, e_2, \cdots, e_n$ where v_x is the from-vertex of e_1 and v_y is the to-vertex of e_n, and for each two adjacent edges e_{k-1} and e_k in p, the to-vertex of e_{k-1} is the same as the from-vertex of e_k. The corresponding label sequence $\phi(e_1), \phi(e_2), \cdots, \phi(e_n)$ denoted by $\phi(p)$ (by abusing notation) is the* path label *of p.*

The graph representation of streaming graph S is shown on the right of Figure 2(a). The labels and timestamps of the edges are marked next to them. The label sequence of path $p : a \rightarrow c \rightarrow b \rightarrow d \rightarrow e$ is $\phi(p) =$ `fol, mem, mem, tra`.

To restrict the valid tuples in a streaming graph when querying, we use a *time-based sliding window* (sliding window for brevity in this paper) as in many real-world applications.

Definition 3. ***Sliding window over Streaming graph**: A sliding window W with window size $|W|$ and sliding step length $Step(W)$ at time t, denoted W^t (with $|W|$ and $Step(W)$ understood), on a streaming graph S consists of the set of tuples $\tau_i \in S$ such that $ts(\tau_i) \in (t - |W|, t]$.*

All the edges in a sliding window form a "snapshot graph" for the window. The streaming graph S in Figure 2(a) contains two neighboring sliding windows: W^{14} and W^{15} of window size $|W| = 10$ with sliding step length $Step(W) = 1$. Note that edges e_1 and e_2 that both have the timestamp of 5 are not included in W^{15} by definition. The corresponding snapshot graphs are in Figures 2(b) and 2(c). For simplicity, we assume that the start time of the first sliding window is 1, i.e., the first sliding window is $W^{|W|}$

Based on this definition of the sliding windows on a streaming graph, we introduce the Streaming RPQ evaluation task.

Definition 4. ***(Persistent) RPQ**: Let Σ be a finite alphabet set consisting of all the labels in a streaming graph S. A* regular expression *R over Σ and the set $L(R)$, which consists of all the label sequences that match R, are defined in the usual way. A (Persistent) RPQ evaluation on a streaming graph S with regular expression R and sliding window W is to find all the vertex pairs (u, v), where $u \neq v$, such that there exists a path p from vertex u to v such that $\phi(p)$ matches R on the snapshot graph for the sliding window W^t for some positive integer $t \geq |W|$.*

Formally, denote the set of all the above pairs in the sliding window W^t as $Q^t_{R,W}$. Then,

$$Q^t_{R,W} = \{(u, v) | \text{there exists a path } p \text{ on } S \text{ from vertex } u \text{ to } v \text{ such that } \phi(p) \text{ matches } R \text{ and } t - |W| < ts(\tau_i) \leq t \text{ for each tuple } \tau_i \text{ in } p\}$$

$Q^t_{R,W}$ may be simplified as Q^t since both the regular expression R and sliding window W are determined in advance.

In the Streaming RPQ evaluation task, the goal is to continuously find Q^t for $t = |W|, |W| + Step(W), |W| + 2 * Step(W), \cdots$. We now illustrate the Streaming RPQ evaluation with an example.

Example 1. *Consider the RPQ evaluation with regular expression $R = (\texttt{fol} \circ \texttt{mem} \circ \texttt{tra})^+$ and sliding window size $|W| = 10$ with sliding step length 1 on the streaming graph S in Figure 2(a). For the "snapshot graph 1" of the streaming graph in sliding window W^{14} shown in Figure 2(b), the label sequence of the graph paths $a \rightarrow c \rightarrow d \rightarrow e$ and $a \rightarrow b \rightarrow d \rightarrow e$ are both "*`fol,mem,tra`*", which matches R. Thus the streaming RPQ evaluation result in $Q^{t=14}_{R,W} = \{(a, e)\}$.*

The snapshot graph of the next sliding window W^{15} is shown in Figure 2(c). The $Q^{t=15}_{R,W} = \{(a, e)\}$ holds by the only path $a \rightarrow b \rightarrow d \rightarrow e$. Note that path $e \rightarrow b \rightarrow d \rightarrow e$ is excluded in the definition since the start-vertex and the end-vertex are identical.

Definition 5. ***Deterministic Finite Automaton**: A* Deterministic Finite Automaton *(DFA) for R refers to a 5-tuple $DFA(R) = (Q, L, \delta, q_0, F)$ where (i) Q is a finite set of states, (ii) L is a finite set of symbols, (iii) mapping function $\delta : Q \times L \rightarrow Q$, (iv) q_0 is the initial state of DFA, and F is a set of final states that $q_0 \in Q$ and $F \subseteq Q$.*

For a given regular expression R, we may construct a *Non-Deterministic Finite Automaton* that matches the R using Thompson's construction algorithm [42]. Then, we use Hopcroft's algorithm [23] to build its minimal Deterministic Finite Automaton, denoted as $DFA(R)$. An example DFA of $R = (\texttt{fol} \circ \texttt{mem} \circ \texttt{tra})^+$ is shown in Figure 2(d). Note that the DFA is constructed before RPQ evaluation starts.

Definition 6. ***Query Spanning Tree [38]**: Given a $DFA(R) = (Q, L, \delta, q_0, F)$ and a snapshot graph $G = (V, E, \psi)$, where V is a set of vertices, E is a set of edges and ψ maps each edge to a label, a* query spanning tree *(spanning tree for short) generated from G with root (x, q_0), denoted $Tree_x$, is a tree with each tree node being a pair (u, s_u), where $u \in V$ and $s_u \in Q$, and each edge $((u, s_u), (v, s_v))$ in $Tree_x$ satisfies $e = (u, v) \in E$, $\psi(e) = l$ and $\delta(s_u, l) = s_v$. Each node in a spanning tree appears only once (i.e., no two nodes have the same vertex and state pairs simultaneously), and each node, except the root, has exactly one parent node.*

A spanning tree is constructed from a given snapshot graph of W^t guided by the $DFA(R)$ such that each node in the spanning tree is labeled with a graph vertex and a state in $DFA(R)$. The root has the initial state of $DFA(R)$. A path from the root to each node corresponds to a sequence of edges in the graph and forms a *prefix* of a label sequence accepted by the $DFA(R)$. For each node (v, s_v), where $x_v \in F$, in $Tree_x$, (x, v) is a query result in the given sliding window. Here, we briefly illustrate the process of maintaining spanning trees in PBO and provide an example. For more details about PBO, please refer to [38].

The maintenance of the query spanning tree is as follows: For an input tuple $((u, v, l), t)$, it is labeled with all state pairs (s_u, s_v) satisfying $\delta(s_u, l) = s_v$. Then, for each such a pair, the edge $((u, s_u), (v, s_v))$ is inserted into each query spanning tree $Tree_x$ if the node (u, s_u) is in $Tree_x$ but (v, s_v) is not. An exception is that when (u, s_u) is not in any spanning tree and s_u is $q_0 \in DFA(R)$, then a new spanning tree $Tree_u$ is formed with the only edge $((u, s_u), (v, s_v))$. Subsequently, PBO extends the newly inserted edge by recursively connecting other tuples available within the current window that can be linked to it until there are no more tuples that can be further connected. When the window slides, an expiration phase starts. For each tuple $((u, v, l), t)$ that is moved out of the current window, each edge of the form $((u, s_u), (v, s_v))$ is removed from a spanning tree with $s_v = \delta(s_u, l)$. If a node is "orphaned" from the deletion, PBO searches for the tuples in the stream in

the current window to try to connect it back to the spanning tree unless the root node is deleted, in which case the spanning tree is removed. Explicit tuple deletion within a sliding window is discussed in Section 4.4.

We now illustrate the above process with an example.

Example 2. *Consider the streaming RPQ evaluation in Example 1. The $DFA(R)$ for regular expression R is shown in Figure 2(d). Here we mark the states in Q of $DFA(R)$ as numbers 0 to 3, respectively. For the snapshot graph in window W^{14} shown in Figure 2(b), the corresponding spanning tree $Tree_a$ is shown in Figure 2(e). Each path in $Tree_a$ from the root to a node has its label sequence partially matching R. The query result (a, e) is obtained in $Tree_a$ since the vertex e is labeled with a final state of the $DFA(R)$, which means the label sequence of the path from the root a to e matches R.*

When the time window slides to W^{15}, tuple $(e_7, 15)$ arrives, tuples $(e_1, 5)$ and $(e_2, 5)$ are moved out of the current window.

For e_1 and e_2 in the query spanning tree $Tree_a$ in W^{14}, the tree edges $((c, 1), (d, 2))$ and $((c, 1), (b, 2))$ need to be removed since both edges (c, b, mem) and (c, d, mem) have moved out of the current sliding window W^{15}. Node $(d, 2)$ in $Tree_a$ becomes "orphaned". Then, PBO traverses the tuples in W^{15} and finds another edge (b, d, mem) that can connect this node to $Tree_a$. The final spanning trees generated by the snapshot graph in window W^{15} shown in Figure 2(c) is presented in Figure 2(f).

For e_7, edge (e, b, fol) can be labeled with two state pairs $(0, 1)$ and $(3, 1)$ by $DFA(R)$. Edge $((e, 3), (b, 1))$ is ignored since node $(e, 3)$ is not included in any existing spanning trees. State 0 is the initial state in $DFA(R)$, and there is no spanning tree $tree_e$ here. Thus, a new spanning tree $Tree_e$ is created, and the tree edge $((e, 0), (b, 1))$ is inserted into it. The other tuples in this window that can extend this edge are also traversed and recursively inserted into $Tree_e$. The final spanning tree $Tree_e$ is shown in the right of Figure 2(f).

The amortized time complexity of PBO in inserting a tuple to the spanning trees is $O(n \cdot k^2)$ as shown in [38], where n is the number of distinct vertices in the snapshot graph G, and k is the number of states in $DFA(R)$. We now describe the time complexity of the expiration process in PBO [38]. In order to accelerate the expiration process, PBO [38] marks each node (except the root) in the spanning trees with the minimum timestamp among the timestamps of the edges leading to it from the root. Thus, when PBO executes expiration, it incurs a cost of $O(log_2 m)$ to find and remove expired tuples whose timestamps are not in the next sliding window, assuming the tuples in the streaming graph are sorted with their timestamps. For each removed tuple, there is a cost of $O(n \cdot k^2)$ to locate and remove the tree edges in all the spanning trees, where m is the number of edges in G, n and k are the same as above. Afterwards, there is a need to recursively traverse the entire snapshot graph to adjust the tree edges' timestamp. Thus, the overall time complexity of expiration in PBO is $O(n \cdot m \cdot k^2)$ since the above removal process needs to be done for each of the m tuples. Note that the expiration in PBO is a blocking phase invoked each time the sliding window moves, and the time consumption of expiration is high.

Table 1 summarizes the notations used in this paper.

3 MWP WITH TRD

In this section, we introduce the streaming RPQ evaluation method Multiple-Window Parallel (MWP). We begin by defining the concept of *Timestamped Rooted Digraph* (TRD). Next, we provide a detailed description of the MWP method using TRD and a theoretical analysis of our proposed algorithms. We will introduce the "dirty garbage collection" in the next section.

3.1 TRD

We first provide a formal definition for the TRD.

Table 1. Notation Table

Notation	Meaning
$e = (v_i, v_j, \phi(e))$	A directed edge e from vertex v_i to v_j with label $\phi(e)$
$S, S^{\tau_n} = \tau_1, \tau_2, \cdots, \tau_n$	Streaming graph S containing a sequence of tuples τ. S^{τ_n} is the sub-sequence of S up to a specific τ_n
$\tau = (e, ts(\tau))$	Tuple τ of S containing an edge e and a timestamp $ts(\tau)$
$p = e_1, e_2, \cdots, e_n$	A graph path that is a sequence of edges
$\phi(p)$	The label sequence of path p
W, W^t	Sliding window W on S with length $\|W\|$ and sliding step length $Step(W)$. W^t is the set of tuples τ in S where $ts(\tau) \in (t - \|W\|, t]$
$R, DFA(R)$	Regular expression R. $DFA(R)$ is the Deterministic Finite Automaton of R
$Q^t_{R,W}$	Set of results in streaming RPQ evaluation with query R and sliding window W at time t
$Tree_x$	Spanning Tree rooted at x
T_x	TRD rooted at x
Δ	Set of TRDs

Definition 7. ***Timestamped Rooted Digraph****: For a given regular expression R and a streaming graph S, a* timestamped rooted digraph *for S guided by $DFA(R) = (Q, L, \delta, q_0, F)$ is a connected directed graph with nodes of the form (u, q), where u is a vertex of S and q is a state of $DFA(R)$. Each TRD has a root node (x, q_0), and thus the TRD is denoted T_x. Each edge $((u, s_u), (v, s_v))$ in T_x satisfies $\delta(s_u, l) = s_v$, where l is a label of a edge in S from u to v. Each node in T_x is associated with a non-empty finite set of positive integers called timestamps. The timestamps of the root contain all the positive integers by default if not otherwise specified. No two nodes in a TRD have the same vertex and state.*

Note that the root node in a TRD has a path to reach each of the other nodes in the TRD by definition 7.

We assume that the $DFA(R)$ is a minimum deterministic finite automaton for R. By the above definition, the label sequence of each path in a TRD from the root to any other node forms a prefix of a sequence that matches the query expression R. If the state of the last node on the path is a final state of $DFA(R)$, the label sequence then matches R.

As indicated in the definition, each node in a TRD is associated with a set of timestamps. If t is a timestamp of a node, then it means that the node "exists in the window W^t." In the sequel, we say a node "exists at t" to mean that the node can be used to form a path in the sliding window W^t. A node may exist in multiple sliding windows if it is associated with multiple timestamps. We denote the timestamp set of a node (u, s_u) as $(u, s_u).ts$.

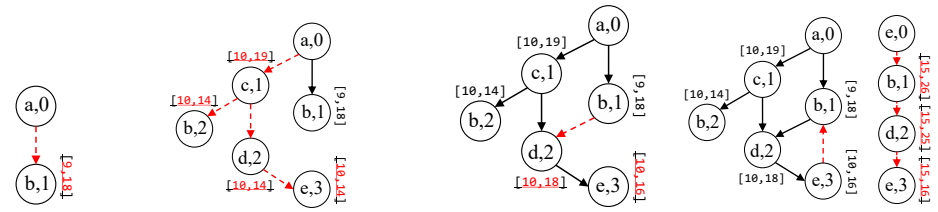
Definition 8. ***Valid TRD****: Given a TRD T_x, and a node (u, s_u) in T_x, we say that a timestamp t on node (u, s_u) is* valid *if t is in the timestamp set of each node in one of its paths from root to (u, s_u). If all the timestamps on all the nodes in T_x are valid, then T_x is valid.*

We only need to use valid TRDs in our MWP method (except after DGC, see Section 4). Intuitively, on a valid TRD, if t is a timestamp associated with a node (u, s_u), then there is a path in W^t from the root to (u, s_u).

An example of valid TRDs is shown in Figure 3. We use Δ to denote the set of valid TRDs.

3.2 MWP method

This section describes the MWP method, which consists of two algorithms, *RPQ* (Algorithm 1) and *Update* (Algorithm 2), for RPQ evaluation. These algorithms incrementally maintain valid TRDs over the append-only streaming graph.



(a) Δ at $t = 9$ (b) Δ at $t = 10$ (c) Δ at $t = 14$ (d) Δ at $t = 15$

Fig. 3. The Δ generated from streaming graph S^τ in Figure 2(a) with window size 10, sliding step length 1 and $DFA(R)$ in Figure 2(d). The newly added edges are shown with red dashes, and the updated timestamp sets are highlighted in red and underlined. We use intervals to represent sets of timestamps.

Algorithm *RPQ* (Algorithm 1) processes all tuples $\tau = ((u, v, l), t)$ from the streaming graph S^τ in the order they arrive. We use S^τ to include the tuples that have not been removed yet (with garbage collection in Section 4) and arrived no later than τ. We also maintain all the TRDs denoted Δ.

For every tuple $((u, v, l), t)$, Algorithm *RPQ* first identifies vertex pairs (u, v) with state pairs (s_u, s_v) that satisfy $\delta(s_u, l) = s_v$. Then, for each of these state pairs (s_u, s_v):

- If $s_u = q_0$ and T_u does not exist in Δ, which means there needs to be a new TRD rooted in u, then add T_u to Δ (Line 5).

- Algorithm *Update* is invoked for each edge $((u, s_u), (v, s_v))$ and each TRD in Δ containing node (u, s_u) (Line 6 to 8).

Algorithm 1: *RPQ*

Globally maintained: TRD set Δ, the streaming graph S^τ
Input: Tuple $\tau = (e, t)$ where $e = (u, v, l)$

```
1 begin
2     Add τ to S^τ;
3     foreach (s_u, s_v) s.t. δ(s_u, l) = s_v do
4         if s_u = q_0 ∧ T_u ∉ Δ then
5             Create a new TRD T_u and add it to Δ;
6         foreach T_x ∈ Δ do
7             if (u, s_u) ∈ T_x then
8                 Call Update(T_x, ((u, s_u), (v, s_v)), t);
```

Algorithm *Update* (Algorithm 2) inserts an edge $e = ((u, s_u), (v, s_v))$ with timestamp t into a TRD T_x and updates the timestamp set of the affected nodes. Recall that t is the timestamp of the tuple $((u, v, l), t)$ processed by Algorithm *RPQ*. First, it initializes the timestamp set of node (v, s_v) to empty if $(v, s_v) \notin T_x$ (Line 3). Then it identifies the set $T = [t, t + |W|)$ such that for each $t' \in T$, the sliding window $W^{t'}$ contains t, and updates $(v, s_v).ts$ by adding these new "existing" timestamps, which is the intersection between the parent node (u, s_u)'s timestamp set and T (Line 4). Intuitively, if there is any path from the root to (u, s_u) that exists within a sliding window ($W^{t'}$), this path can also be extended to include node (v, s_v) within that same window ($W^{t'}$). It is important to note here that we assume all TRDs are kept valid. If the new node (v, s_v) has a non-empty timestamp, it should be inserted into the TRD (Line 6). The insertion of edge e triggers an update of TRD T_x

if there is a change in $(v, s_v).ts$ (Line 7), which requires recursively invoking Algorithm *Update* within S^τ (Line 11) to update T_x due to the changes caused by the new tuple.

In the algorithm, if $(v, s_v).ts$ is modified and $s_v \in F$, then it sends the result pair (v, s_v) and timestamp set $(v, s_v).ts$ to a thread specifically for collecting results (Line 9). We call the thread *Result Recorder Thread*.

Algorithm 2: *Update*

Globally maintained: TRD set Δ, the streaming graph S^τ
Input: TRD T_x, Edge $e = ((u, s_u), (v, s_v))$, timestamp t

```
1  begin
2      if (v, s_v) ∉ T_x then
3          Let (v, s_v).ts = ∅ // Initialize
4      Let (v, s_v).ts = (v, s_v).ts ∪ ((u, s_u).ts ∩ [t, t + |W|));
5      if (v, s_v).ts ≠ ∅ ∧ (v, s_v) ∉ T_x then
6          Insert ((u, s_u), (v, s_v)) into T_x;
7      if (v, s_v).ts is updated in Line 4 then
8          if s_v ∈ F ∧ v ≠ x then
9              Send (x, v) and (v, s_v).ts to the thread responsible for collecting and reporting
                 results;
               // collect results
10         foreach ((v, w, l), t'') ∈ S^τ ∧ δ(s_v, l) = s_w do
11             Call Update(T_x, ((v, s_v), (w, s_w)), t'');
               // recursively update
```

We now illustrate our proposed algorithms with an example.

Example 3. *Consider the RPQ evaluation on the streaming graph S^τ shown in Figure 2(a). The query conditions are the same as in Example 1. The regular expression R is* $(\texttt{fol} \circ \texttt{mem} \circ \texttt{tra})^+$ *and its corresponding $DFA(R)$ is shown in Figure 2(d). The window size $|W|$ is 10, and the sliding step length is 1. In addition, we label the states q_0, q_1, q_2, and q_3 of $DFA(R)$ as numbers 0 to 3, respectively.*

Before $t = 9$, all tuples won't cause the creation of TRDs since we need a tuple with the label "`fol`*" to create a root node with state 0.*

At time $t = 9$, edge e_4 arrives and creates the TRD T_a shown in Figure 3(a) since the vertex a in e_4 is labeled with an initial state 0. The time interval $[9, 18]$ indicates that node $(b, 1)$ exists in sliding window W^t for all $t \in [9, 18]$ (In reality, W^t does not exist if $t < 10$, but this does not effect the correctness.). The edges marked with red dashes are newly added to this TRD, and the timestamp sets highlighted in red and underlined indicate that they have been modified. Note that the call to Algorithm Update with all the tuples that arrived before $t = 9$ does not cause any changes to T_a since each tuple will make a new edge on T_a but does not satisfy the insertion condition (Line 5).

At time $t = 10$, upon the arrival of edge e_5, T_a has a new edge $((a, 0), (c, 1))$. Then edges e_1, e_2, and e_3 are recursively traversed, and additional edges are added to the existing TRD T_a. As a result, nodes $(b, 2)$, $(d, 2)$ and $(e, 3)$ are created as the descendants of node $(c, 1)$. The updated TRD after inserting edge e_5 is shown in Figure 3(b). It should be noted that vertex b can have multiple states in this TRD T_x.

At time $t = 14$, edge e_6 is labeled with state pairs $(1, 2)$ and added to T_a, which creates a new path $(a, 0) \rightarrow (b, 1) \rightarrow (d, 2) \rightarrow (e, 3)$ in this TRD. The timestamp set of $(d, 2)$ is updated by taking the

union of the new existing timestamps from the new path, causing the updating of $(e, 3).ts$. *Any more calls to Algorithm Update do not cause any changes to the TRD. The corresponding TRD* T_a *is shown in Figure 3(c).*

When edge e_7 *arrives with timestamp* $t = 15$, *it creates a new TRD* T_e *with state pair* $(0, 1)$ *and is inserted into the TRD* T_a *with state pair* $(3, 1)$. *The insertion process on* T_a *terminates early as none of the node's timestamp sets were updated. The final snapshot of* Δ *is shown in Figure 3(d).*

Theorem 1. *Algorithm RPQ is complete and correct.*

Proof Sketch: To prove this, we need to show that Algorithm *RPQ* can find all the query results for the given streaming tuples S^τ. Firstly, all the paths whose label sequence partially matches R and exist in a certain timestamp in the streaming graph can be founded in Δ. Besides, Algorithm *Update* ensures that when there is a path from node x to node v in a TRD T_x whose label sequence matches R, and all edges on this path exist at a same sliding window W^t, then the node (v, s_v) where $s_v \in F$ is added to T_x and vertex pair (x, v) is added to Q^t. Additionally, the timestamp set $(v, s_v).ts$ calculated by Algorithm *Update* contains all such t. Thus, the Algorithm *RPQ* is complete and correct.

Lemma 1. *The timestamp set* $(v, s_v).ts$ *in a TRD* T_x *can be updated at most* $|W|$ *times by Algorithm Update with the same tuple* τ.

Proof: Consider the invocation for Algorithm *Update* with an edge $e = ((u, s_u), (v, s_v))$ and a timestamp t. It is clear that this edge can exist in a sliding window $W^{t'}$ for all $t' \in [t, t + |W|)$, which is illustrated earlier. Since there are at most $|W|$ sliding windows in $[t, t + |W|)$, the $(v, s_v).ts$ on the edge e and timestamp t can be updated by line 4 at most $|W|$ times. Therefore, for a certain tuple $\tau = ((u, v, l), t)$ that satisfies $\delta(s_u) = s_v$, $(v, s_v).ts$ in a TRD can be updated by it at most $|W|$ times.

Theorem 2. *The amortized time consumption of Algorithm RPQ is* $O(n \cdot k^2)$ *over* m *tuples in* S^τ, *where* n *is the number of distinct vertices in* S^τ, k *is the number of states in* $DFA(R)$.

Proof: Inserting a new edge in Line 3 and sending messages in in Line 4 take a constant time, which is the same as that in PBO [38]. Thus, the time complexity of Algorithm *RPQ* is determined by the number of times Algorithm *Update* is invoked.

Let's consider the processing of a tuple τ with edge $e = (u, v, l)$ and timestamp t. In automaton $DFA(R)$, there can be at most k^2 state pairs (s_u, s_v) that satisfy $\delta(s_u, l) = s_v$. Therefore, each tuple can produce at most k^2 edges in TRDs. Besides, the recursive invocation of Algorithm *Update* occurs when $(v, s_v).ts$ is updated. According to Lemma 1, Algorithm *Update* with parameters $((u, s_u), (v, s_v))$ and t takes at most $|W|$ updates on $(v, s_v).ts$ in a TRD. Note that there are at most n TRDs. The total number of invocations for Algorithm *Update* is bounded by:

$$n \cdot m \cdot k^2 \cdot |W|$$

Thus, the amortized time complexity of Algorithm *RPQ* is $O(n \cdot k^2)$. Note that the amortized time complexity of Algorithm *RPQ* is the same as the amortized insertion time complexity of PBO.

The timestamp set on each node can be compressed by only recording the timestamps t when W^t with length $|W|$ and sliding step length $Step(W)$ exists, that is, t satisfies: $|W| \leq t$ and $(t - |W|)$ mod $Step(W) == 0$. These are the same timestamps used for *Result Recording Thread* to update results Q in line 9 of Algorithm *Update*. It means that the number of timestamp updates, as stated in Lemma 1, can be limited to at most $|W|$.

Parallel Strategy: The description of our algorithms is in terms of sequential execution (single thread). Parallel execution with multithread is feasible for MWP. The *parallel strategy* for

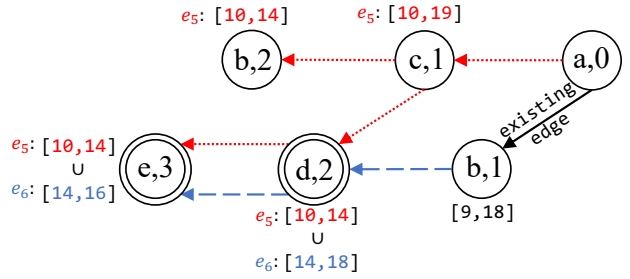


Fig. 4. The process of parallel invoking Algorithm *RPQ* for edges e_5 and e_6 on TRD T_a is shown in Figure 3(a). The red dotted arrows are added by processing edge e_5, while the blue dashed arrows are added by processing edge e_6. The updated timestamp sets of nodes are marked with e_5 and e_6, respectively. Conflicts in updating the timestamp set for nodes $(d, 2)$ and $(e, 3)$ are highlighted with double circles.

MWP involves parallelly executing Algorithm *RPQ* (Algorithm 1) for each tuple in the streaming graph S^τ. No synchronization is needed for Algorithm *RPQ* with different τ. The Algorithm *Update* (Algorithm 2) can also be invoked in parallel but need careful management on data access by implementing a *Readers-Writer Lock* for the timestamp set on each node in all TRDs. We call the Algorithm *RPQ* (Algorithm 1) execution (as well as all the subsequent calls to Algorithm *Update*) with a single tuple τ as a *RPQ task*. The updates from different *RPQ tasks* to the timestamp set of a node in line 4 of Algorithm *Update* must be atomic for correctness, which is guaranteed by the lock mentioned above.

For example, Figure 4 shows the parallel executing of two *RPQ tasks* with tuples $(e_5, 10)$ and $(e_6, 14)$ on maintaining the same TRD T_a. The processes of these two invocations in T_a, which include inserting edges and updating timestamps, are marked in red and blue, respectively. When both invocations update the timestamp set of node $(d, 2)$ in Line 4 of Algorithm *Update*, one is blocked until another completes. Note that regardless of the order in which the timestamp set is updated by these two invocations, nodes $(d, 2)$ and $(e, 3)$ always produce consistent timestamps in T_a. The above "Parallel Strategy" has no effect on the correctness of the results.

4 GARBAGE COLLECTION (GC)

In this section, we turn our attention to the garbage collection (GC) on the TRDs. We first present the calculation for the cut-off time of garbage collection in Section 4.1. Then we provide a naive GC approach in Section 4.2, and introduce our proposed DGC algorithm in Section 4.3. Finally, we provide a brief discussion of MWP on supporting processing explicit deletions in Section 4.4.

4.1 GC Cut-off Time

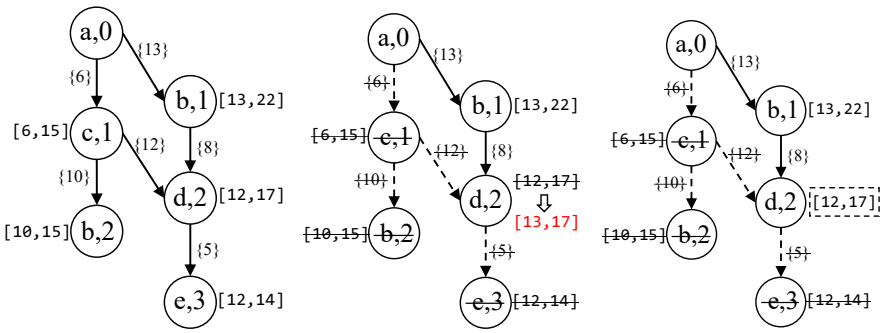
Given a time t, we want to identify the *garbage collection cut-off time* t_{gc} such that if all the tuples with a timestamp less than t have been processed via Algorithm *RPQ*, then all tuples in S^τ with timestamp less than t_{gc} are no longer relevant to produce any new query result after time t. The value t_{gc}, given t, is calculated as follows. Among all the tuples with timestamps less than t, assume the tuple with the largest timestamp is τ'. Then t_{gc} is formally defined as: $t_{gc} = ts(\tau') - |W|$.

The calculated GC cut-off time t_{gc} is correct since the timestamp of a tuple that is less than t_{gc} is not going to be in the same sliding window with any tuple with a future timestamp (i.e., after $ts(\tau')$). Hence, removing tuples before t_{gc} will not impact the new results in the windows $W^{\tau'}$ for all $t' > ts(\tau')$ by Definition 4.

4.2 Naive GC on TRDs

The naive GC strategy identifies the old tuples in the maintained streaming graph whose timestamp is less than a given GC cut-off time t_{gc}. These identified tuples are then removed from the graph, and the TRDs in Δ are adjusted accordingly. The pruning process for the streaming graph S^τ involves simply removing these old tuples. The adjustment on TRDs is based on the fact that each edge $((u, s_u), (v, s_v))$ in a TRD T_x is due to a set of tuples (called the "associated tuples" of the edge) with the form $((u, v, l), t)$ in S^τ when building T_x with Algorithm *RPQ*. When a particular tuple $((u, v, l), t)$ is removed from S^τ, it may be necessary to update the timestamp set of node (v, s_v) by recalculating its existing times using a process similar to one in Algorithm *Update*. The edge $((u, s_u), (v, s_v))$ will be removed if all its associated tuples are removed, and the node (v, s_v) will be removed if there are no incoming edges or when its timestamp set $(v, s_v).ts$, after recalculation, becomes empty. Additionally, the timestamp set of all the descendants of (v, s_v) in T_x also needs to be recursively updated to maintain the validity of the TRD.

Note that the associated set of tuples in S^τ for each edge in TRD can be simply implemented by recording the tuples when executing Algorithm $Update$. We illustrate the naive GC with a simple example.



(a) TRD before GC (b) Naive GC (c) DGC

Fig. 5. (a) a TRD T_a where each edge in T_a is marked with the timestamps of its associated tuples. (b) and (c) the naive GC and DGC processes with GC threshold $t_{gc} = 7$. The removed edges, nodes, and the corresponding timestamps are marked in dashes or strikethroughs, and the newly updated timestamps are marked in red.

Example 4. *Consider the GC process with GC cut-off time $t_{gc} = 7$ on TRD T_a shown in Figure 5(a). Each edge in T_a is labeled with the timestamps of its associated tuples in the streaming graph (in this example, an edge associates with only one tuple). Hence, edges $((a, 0), (c, 1))$ and $((d, 2), (e, 3))$ are removed from T_a (marked with dashed lines in the figure) since all the associated tuples are removed. Additionally, nodes $(c, 1)$ and $(e, 3)$ will be removed since all their incoming edges are removed. Edges $((c, 1), (d, 2))$ and $((c, 1), (b, 2))$, as well as node $(b, 2)$, will also be removed due to the removal of node $(c, 1)$. These removals cause a recalculation of the timestamp set of node $(d, 2)$. The final snapshot of T_a using naive GC is shown in Figure 5(b), where the new timestamp set of $(d, 2)$ is marked in red.*

In the above example, if there were additional edges from the nodes $(c, 1)$, $(b, 2)$, $(d, 2)$ and $(e, 3)$ in T_a, the same update operations would be applied to all of them. This process can be highly time-consuming since it may involve updating the timestamp set of every node in the entire TRD, and each node may be visited many times. Note that the expiration in the PBO method essentially

employs this naive algorithm with $t_{gc} = t - |W|$ when t reaches the end of a sliding window, albeit on their spanning trees.

4.3 DGC on TRDs

We propose a strategy called Dirty Garbage Collection (DGC) to achieve faster garbage collection on TRDs than the naive one. The novel idea is to avoid the recalculation of timestamp sets for the affected nodes, as done in the naive GC scheme, but we may leave the TRDs in a bit of "dirty" but correct state.

We summarize DGC in Algorithm *DGC* (Algorithm 3). For a given GC cut-off time t_{gc}, a DGC first removes all the tuples in S^τ (see Algorithm *RPQ*) that have a timestamp less than t_{gc} (Line 2 and 3). For each edge in a TRD, if all the associated tuples have a timestamp less than t_{gc}, then the edge is removed from the TRD (Line 7). Next, we remove all nodes (and all the outgoing edges from them) if there are no incoming edges to them (Line 10). We do this last step repeatedly until no nodes can be removed.

Here we illustrate the Algorithm **DGC** with an example.

Example 5. *Consider the same GC conditions in Example 4. Figure 5(c) illustrates the GC process using DGC methods. In TRD T_a, the removal of edges and nodes is the same as in Example 4. Unlike the naive GC method, however, the DGC method does not update $(d, 2).ts$, which is boxed in a dashed line in this figure, even though it may result in some invalid timestamps like $t = 12$ because $12 \notin (b, 1).ts$.*

Algorithm 3: *DGC*

Globally maintained: TRD set Δ, the streaming graph S^τ
Input: GC cut-off time $t_{gc} \wedge t_{gc} < ts(\tau)$

```
1  begin
2      Let P = {((u, v, l), t) ∈ S^τ | t < t_gc};
       // candidate removal tuples
3      Remove tuples in P from S^τ ;
4      foreach T_x ∈ Δ do
5          foreach ((u, v, l), t) ∈ P ∧ δ(s_u, l) = s_v ∧ e' = ((u, s_u), (v, s_v)) ∈ T_x do
6              if all tuple associated with e' have a timestamp less than t_gc then
7                  Remove edge e' from T_x;
8      repeat
9          if there is a node in a TRD in Δ has no incoming edge then
10             Remove the node along with all its outgoing edges;
11     until no change is made to Δ;
```

Theorem 3. *The Algorithm **DGC** is correct.*

Proof: We only need to show that the invalid timestamps in the resulting TRDs have no impact on the further queries in the RPQ evaluation. Note that though the DGC may break the validity of TRDs by removing some edges without recalculating timestamps for the affected nodes in TRDs (as shown in Example 5), each timestamp t' that is greater or equal to $t_{gc} + |W|$ remains valid on the TRD. Indeed, the fact that t' was valid before DGC was because there is a sequence of tuples in S^τ such that the timestamps of them are all in $(t' - |W|, t']$. Since $t' - |W| \geq t_{gc}$, we know the

timestamps of these tuples are no less than t_{gc} and hence not removed by DGC. Therefore, the path still exists on TRD after pruning. Additionally, since for the sliding windows to be considered by RPQ after DGC, say $W^{t''}$, we have $t'' \geq t \geq t_{gc} + |W|$, then the pruned TRDs will yield the same query results in all subsequent sliding windows $W^{t'}$ for all $t' \leq t$. Thus, Algorithm *RPQ* performs well in the TRDs after DGC for the future tuples since those invalid timestamps are ignored in Line 4 of Algorithm *Update*.

Theorem 4. *The worst time complexity of Algorithm* ***DGC*** *is $O(n \cdot m \cdot k^2)$, where n is the number of distinct vertices in a streaming graph of S^τ and k is the number of states in $DFA(R)$.*

Proof: Firstly, since tuples in the streaming graph are sorted with their timestamps, the maintenance of the streaming graph takes $O(log_2 m)$, where m is the number of tuples in S^τ. For each removed tuple τ', we look for TRD edges that are associated with τ' and test if all its associated tuples are removed. If so, remove this edge. This part of the algorithm takes $n \cdot m \cdot k^2$ time, where n is the number of distinct vertices, and k is the number of states in $DFA(R)$. The other part of time complexity in Algorithm DGC is the number of loops executed in Line 8 to 11. The worst-case scenario occurs when all nodes in TRDs are removed, as mentioned earlier. Note that each node in a TRD can be visited only once. Therefore, this loop can be executed $n^2 \cdot k$ times in the worst-case, since there are most n TRDs, each with $n \cdot k$ nodes. The final time complexity of Algorithm DGC is $O(log_2 m + n \cdot m \cdot k^2 + n^2 \cdot k)$, which is less than that of PBO ($O(log_2 m + n \cdot m \cdot k^2 + n \cdot m \cdot k^2)$). We note that the Naive GC only differs from DGC in recalculating the timestamp sets of nodes associated with the removed edges. Thus, the complexity of Naive GC is $O(log_2 m + n \cdot m \cdot k^2 \cdot n \cdot k + n^2 \cdot k)$, i.e., $n \cdot k$ multiplied to the middle term in the complexity of DGC. This complexity is even higher than that of PBO due to the graph structure, instead of the tree structure, used in MWP. Therefore, we do not consider Naive GC further in this paper.

The DGC is responsible for deleting "garbage" data structures and freeing up memory, and can be executed when memory runs low. The explicit execution for DGC can improve system performance during query evaluation, especially when traversing these "garbage" data structures in Algorithm *RPQ* is more costly than executing DGC. In such cases, it is recommended to execute DGC. We have conducted extensive experiments and will provide a detailed discussion in Section 5.4.3.

Parallel Strategy: The DGC can be executed in a highly parallel fashion. Each removal of nodes and edges on a TRD (Line 10 in Algorithm *DGC*) can be done in parallel. After removing the old tuples from the streaming graph and edges from TRDs, the recursive traversal (Line 8 to 10) can be done in parallel with the premise that each node can only be removed once based on the atomic operation in Line 9 of Algorithm *DGC*. If multiple DGC processes (for different tuples) on the same TRD are to remove the same node and all its outgoing edges in Line 10, only the first thread will succeed, and all others will do nothing. Here, we refer to each such recursively traversal in TRDs as a *GC task*, which involves executing Lines 8 to 11 in Algorithm *DGC*. Note that the parallel processing of the *GC tasks* is correct since even when two tasks meet at the same node in TRDs, only one can continue the traversal from this node.

4.4 Explicit Deletion via GC

A deletion tuple $\tau = (e, ts(\tau))$ is to explicitly remove all tuples with edge e and timestamp less than $ts(\tau)$ in the streaming graph, which needs to adjust the query structures by removing the affected edges. We now briefly discuss the support for deletion tuples in MWP and compare it with that of in PBO.

In MWP, the deletion can be treated with Naive GC. When an edge is removed from a TRD, this TRD is locked from collecting results (Line 9 in Algorithm *Update*), and all the descendant nodes of this removed edge will recursively update their timestamp set.

PBO produces more blocking phases if deletion happens. Indeed, for each deletion tuple, PBO first sets the timestamp of all candidate removal edges in the query spanning trees to "infinitesimal" and then explicitly calls an expiration phase to correctly process these edges, which incurs an extra blocking phase within a sliding window. In contrast, although an explicit deletion in MWP also needs to lock the affected TRD, block the processing of other tuples on the same TRD, only the TRDs with deletion are locked, and the other tuples can still be parallelly processed. Note that PBO may also adopt this locking approach to reduce blocking during deletion, but this may not provide much relief since the blocking in PBO at each sliding window is necessary and severely degrades the performance. We hence do not further compare MWP and PBO in terms of explicit tuple deletion in this paper.

5 EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate the performance of the proposed methods over two real-world datasets *Yago3* and *Stackoverflow*. We first introduce the experiment settings in Sections 5.1 and 5.2. The evaluation metrics are presented in Section 5.3. Experimental results and their analysis are in Section 5.4.

Here we give a brief highlight of our experimental results:

- MWP achieves a throughput up to 15.2 times higher than PBO, with a significantly lower tail latency (only 0.2% of that of PBO for certain queries).
- The performance of MWP improves as the sliding step length increases, while PBO shows the opposite trend. When *Step(W) = 25* on dataset *Stackoverflow* with query Q_2 and $|W| = 30$, MWP experiences tail latency that is only 2% of that of PBO.
- DGC strategy performs well in the RPQ evaluation system by sufficiently freeing up memory.
- As the streaming speed increases, MWP demonstrates higher and more stable performance (in terms of throughput and tail latency) compared to PBO.

5.1 System Setup

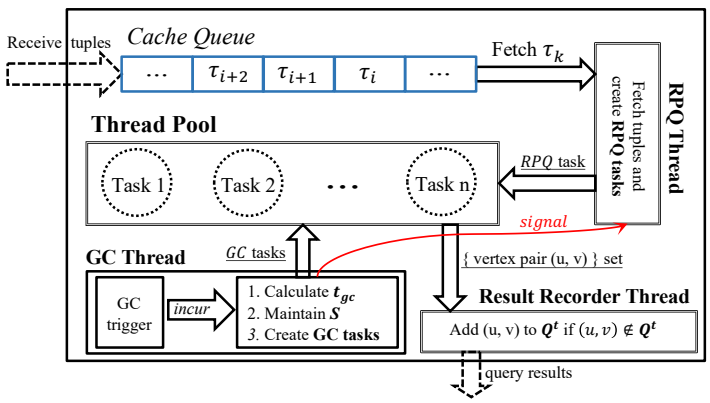


Fig. 6. Overview of RPQ evaluation system consisting of four types of threads. The arrows between different threads are the transmitted messages

Figure 6 provides an overview of the RPQ evaluation system. It consists of a data structure called the "Cache Queue" (highlighted in blue) and four types of threads (highlighted in bold):

(1) The "Cache Queue" continuously receives tuples from an external source and appends them to the end of the queue.

(2) The "RPQ thread" fetches tuples from the front of the "Cache Queue." For each tuple, it creates a "*RPQ task*" and sends it to the "Thread Pool" for processing. For PBO, the *RPQ task* is to maintain query spanning trees (Definition 6), while for MWP, it is the task explained in Section 3.

(3) The "Thread Pool" maintains the streaming graph S^τ (i.e., the tuples that have been fetched by the RPQ thread and not deleted by GC). It consists of multiple threads, with each thread handling one task at a time. Tasks can be either *RPQ tasks* or *GC tasks* and can be executed concurrently with different threads. When handling *RPQ tasks*, any newly generated query result (u, v) along with its timestamp t is sent to another thread known as the "Result Recorder Thread."

(4) The "GC thread" includes a trigger that determines if GC conditions are met. If so, a synchronization signal (indicated by the red arrow in the figure) is sent to suspend fetching new tuples from the "Cache Queue." It then calculates GC cut-off time t_{gc}, updates streaming graph S^τ by removing old tuples, creates a set of "*GC tasks*," and sends them to the Thread Pool for processing. For PBO, the GC is triggered every time the window slides, and the *GC tasks* are to remove all expired edges and adjust the structure of the query spanning trees. For MWP, the GC conditions are customized, and the *GC tasks* are those described in Section 4. Once GC is completed, a signal is sent to resume fetching new tuples by the RPQ thread.

(5) The "Result Recorder Thread" receives vertex pairs and timestamps from threads in the "Thread Pool," checks the result set Q^t for each timestamp t in $(v, s_v).ts$ and adds (u, v) to Q^t if not already in it. It continuously reports new query results and the windows in which they exist.

It is important to note that all four types of threads run simultaneously during RPQ evaluation. However, *GC tasks* and *RPQ tasks* cannot run simultaneously.

5.2 Implementation and Experiment Setup

Implementation: We implemented the whole system in Java 17. For the implementation of the components of this system: **(1)** The "Cache Queue" is implemented as a concurrent queue that supports parallel adding and polling operations. **(2)** The "Thread Pool" is a fixed-size thread pool provided by the Java API. **(3)** The "DGC trigger" in the "DGC Thread" monitors GC conditions every 10 seconds during RPQ evaluation. Message passing between different threads is achieved through shared variables. **(4)** The "Result Recorder Thread" is implemented as described in Section 5.1.

For the implementation of the involved data structures, which are all kept in memory: **(i)** The streaming graph S^τ (i.e., the streaming tuples that have been fetched by the RPQ thread and possibly "chopped" by the *GC tasks*) is stored both as a digraph and an ordered list of tuples. Vertices with their associated tuples in S^τ are stored in a hash table to improve traversal and maintenance efficiency, such as line 10 in Algorithm *Update* and lines 2 & 3 in Algorithm *DGC*. Note that the streaming graph S^τ is shared among all the threads in the system. **(ii)** All TRDs in Δ are indexed by their root nodes and recorded in a concurrent hash table. Each node (u, s_u) also has a correlated hash-based index for efficient lookups. Each TRD maintains two timestamps to mark the maximum and minimum timestamp of currently included tuples for fast pruning during DGC. **(iii)** Timestamp set on each node is recorded as multiple intervals to efficiently calculate the intersections between different timestamp sets within TRDs and limit memory usage since a timestamp set may contain many timestamp values.

Datasets: We introduce two widely-used datasets as follows:

- **Stackoverflow** (SO) is a network of Question & Answer. We form a temporal network of interactions on the stack exchange website containing 2.6M users (vertices) and 74M interactions (edges) about 6 years up to now [37]. There are three kinds of interactions (the labels of the edge) for an edge $((u, v), l, t)$: $(l = a)$ if user u answered user v's question, $(l = b)$

u commented v's answer, or ($l = c$) if u commented v's question, where t is the time when the interaction happened. The window size $|W|$ is set to 1 month, and the sliding step length $Step(1)$ is set to 1 day unless otherwise specified.

- **YAGO3** is a real-world RDF dataset about people, cities, countries, movies, and organizations from the multilingual Wikipedia [31], which contains more than 17M entities (vertices), 150M facts (edges between entities) about these entities, and more than 100 labels on these facts. We select the core facts, with 16M entities, 55M edges, and 36 labels, as the test dataset and manually mark each edge with a non-decreasing integer as its timestamp after randomly shuffling these data. Here we set the sliding window size in YAGO3 to 1M (containing 1M - 5M tuples) and the sliding step length to 50k, which is similar to the condition setting in [38].

Queries: The regular expressions used in the experiments are shown in Table 2. These queries, which are some of the most common recursive queries in real workloads [8], have different complexities [15] and are also used in the PBO evaluation [38]. Note that query Q_8 on dataset SO is represented as $R = (a + b) \circ c^*$ since the SO dataset only contains three kinds of labels.

Table 2. Some of the most common RPQs in real workloads

Tag	Query	Tag	Query
Q_1	a^*	Q_2	$a \circ b^* \circ c^*$
Q_3	$a \circ (b \circ c)^*$	Q_4	$a^* \circ b^*$
Q_5	$a \circ b^*$	Q_6	$a? \circ b$
Q_7	$(a_1 + a_2 + a_3 + \cdots + a_k)^*$		
Q_8	$(a_1 + a_2 + a_3 + \cdots + a_k) \circ b^*$		

Experiment Settings: All experiments were run on a Linux server with 64 physical cores Intel(R) Xeon(R) Gold 5318Y CPU @ 2.10GHz, and 512GB (16 x 32GB) DDR4 2933MHz RDIMM ECC Memory. The experiment results are recorded when the caches are warm and the sliding window is full, making the metrics more representative of the system when it runs normally.

In our experiments, we use the method of reading files to simulate the arrival of tuples of the stream and simulate different arrival speeds of tuples by controlling the speed at which files are read. Besides, we set the trigger for executing DGC in all experiments when an insufficient memory situation occurs except in Section 5.4.3, in which we evaluate the strategy of triggering DGC.

5.3 Metrics

The metrics we use to evaluate the query performance consist of: *throughput*, *tail latency*, *waiting time*, and *processing time*.

Throughput: The throughput of the system can be evaluated by calculating the average number of processed tuples per second on processing a streaming graph within a fixed time interval. Here we evaluate the throughput under the premise that the whole streaming graphs are already added to the "Cache Queue" if not otherwise stated, and record the average throughput during a 10-minute period in a warm cache, such that the throughput can represent the actual processing ability of the system.

Tail Latency (99th percentile): The tail latency is a statistic measured using a sliding window, which captures query conditions in real workloads [39]. It represents the time difference between when all tuples in a window have arrived and when the results within that window are produced. The arrival time of sliding window W^t corresponds to the clock time at which tuples within window W^t all have arrived, while the production time of query results refers to the clock time they are added to Q. Since tuples in window W^t may not arrive at exactly the same clock time,

we calculate their average arrival clock time as the arrival time of the tuples in W^t. It's important to note that we only record the earliest production time for identical results within a sliding window.

To calculate statistical tail latency in RPQ evaluation, we follow this formulaic approach: First, each tuple τ is assigned a clock time $st(ts(\tau))$, which is added to the cache queue. This allows us to compute the average of these clock times as an estimate for the arrival time of tuples in window W^t, denoted as $\overline{st(t)}$. Second, when a vertex pair (u, v) is added to Q^t at a specific clock time (Ct), we add this value to set $rt(t)$ containing all clock times when results are added to Q^t. For each query result $(x, y) \in Q^t$, if it was added at clock time (Ct), its tail latency can be calculated as $max\{Ct - \overline{st(t)}, 0\}$, considering that $\overline{st(t)}$ might be less than or equal to Ct. Based on this information, we determine the tail latency $tl(t)$ for each sliding window W^t using the set:

$$tl(t) = \{max\{0, Ct - \overline{st(t)}\} | \forall Ct \in rt(t)\}$$

We collect the tail latency $tl(t)$ for all the query results in the sliding windows W^t during a 10-minute period in a warm cache and calculate its *99th* percentile value.

Waiting Time & Processing Time (99th percentile): We monitor the waiting and processing times on a per-tuple basis. For a tuple τ, its waiting time is when it has been in the cache queue (received in the Cache Queue but not fetched by the RPQ Thread yet). The processing time is the time consumption of executing its *RPQ tasks* in the Thread Pool. Both waiting time and processing time are represented by its *99th percentile* value during our evaluation.

5.4 Experimental Results

5.4.1 Throughput and Tail Latency: We evaluated the throughput and tail latency of both MWP and PBO methods for all queries on both datasets. The tuple-arrival speed was set to 200 tuples per second for dataset SO and 100,000 for Yago3 when monitoring tail latency.

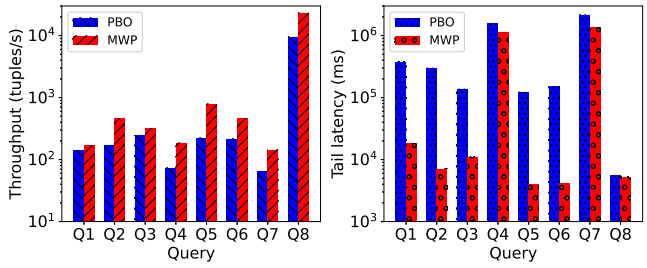
As shown in Figures 7(a) and 7(b), MWP outperforms PBO in terms of both throughput and tail latency across all queries. MWP achieves a throughput that is 1.55 to 3.53 times higher than that of PBO on SO, and 2.79 to 15.2 times higher on Yago3. Additionally, its tail latency is only between 94.32% to 2.3% of that achieved by PBO on SO, and between 2.8% to 0.2% on Yago3.

The effectiveness of MWP is more pronounced in Yago3 due to its sparser nature and more labels compared to SO, which quickens the processing of each tuple and allows more parallel processes.

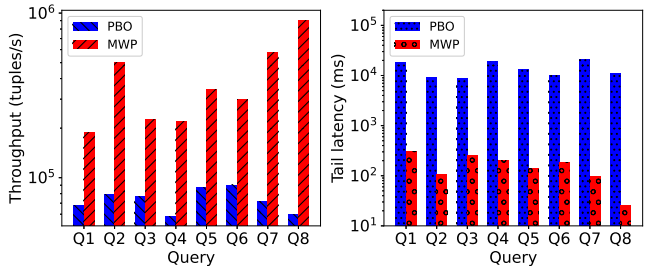
We plot the waiting and processing time for each tuple on both methods over the SO dataset to better understand the diverse performance on different queries in Figure 7(c). MWP has a higher processing time but a significantly lower waiting time compared to PBO due to the non-blocking nature of MWP. As a result, the cost for MWP to process each tuple (processing time + waiting time) is much cheaper than PBO. This finding aligns with our previous analysis in Section 3, which identified the blocking phase during window movement as a major bottleneck that greatly increases the waiting time in PBO. In Figure 7(c), the waiting time and processing time for queries Q_3, Q_4, and Q_7 are different from other queries. This is because different queries incur a different number of TRDs to be inserted and different maintenance costs for each TRD.

5.4.2 Scalability: We evaluated the scalability of two methods for streaming graph queries, specifically by analyzing the impact of window size, sliding step length, and parallelization on throughput and tail latency using query Q_3 on SO. The other conditions were kept consistent with those in Section 5.4.1.

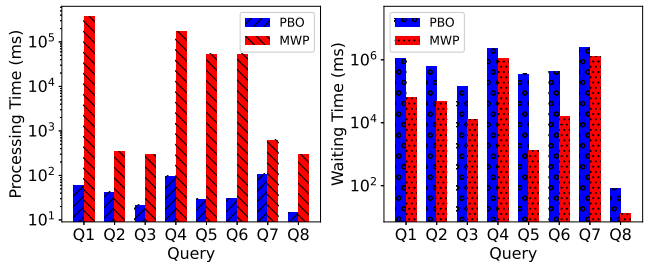
Figures 8(a) & 8(b) present the throughput and tail latency results for PBO and MWP as the window size varies from 7 to 90 (days) (with sliding step length 7), and as the sliding step length varies from 1 to 25 (days) (with window size 30).



(a) SO



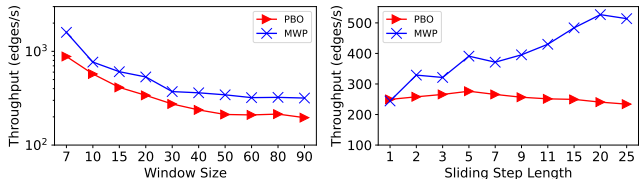
(b) Yago3



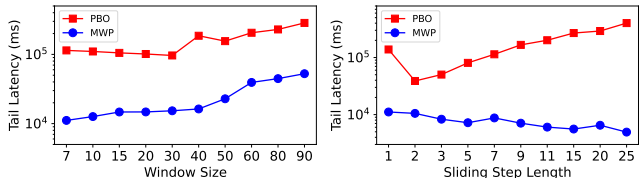
(c) Time consumption on SO

Fig. 7. (a) and (b) the throughput and tail latency of both MWP and PBO on Stackoverflow and Yago3, respectively. (c) the average waiting time in the cache queue and the processing time of the tuples in Stackoverflow. Y axis is given in the log-scale.

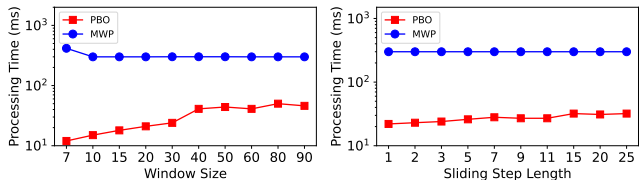
We have the following observations: (i) For different window sizes, both methods exhibit lower latency and higher throughput with smaller windows. However, MWP outperforms PBO as the window size increases. MWP achieves a throughput of 1.3 to 2 times higher than PBO while maintaining a tail latency of only about 19.1% to 8.7% of those of PBO across all window sizes. Additionally, although both methods experience an increase in tail latency with larger window size, MWP's increase is much slower compared to PBO (note that Y-axis uses log-scale). (ii) For different sliding step lengths, MWP consistently achieves a throughput that is approximately 1.02 to 2.19 times higher than that of PBO. Notably, even when $Step(W)$ reaches its maximum value of 25 (days), MWP still maintains a significantly lower tail latency compared to PBO (approximately 27% to 2%). As sliding step length increases further, MWP continues to demonstrate superior performance in terms of both throughput and tail latency over PBO.



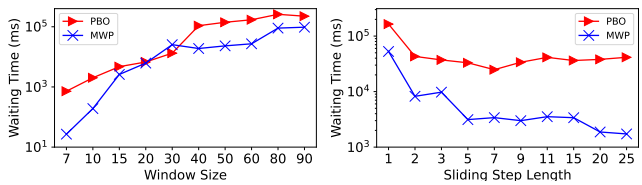
(a) Throughput



(b) Tail Latency



(c) Processing Time



(d) Waiting Time

Fig. 8. (a) & (b) the throughput and tail latency of RPQ evaluation on SO with Query Q_3 on different window size ($Step(W) = 7$) and sliding step length ($|W| = 30$), (c) & (d) the corresponding processing and waiting time

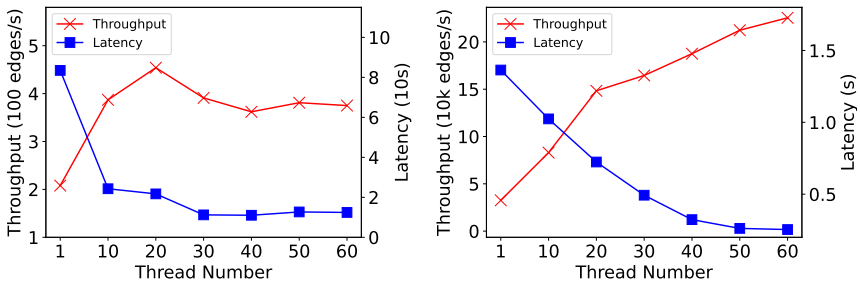
To gain deeper insights into how throughput and tail latency vary based on window size and sliding step length for both methods, we monitored the processing and waiting time of each tuple and conducted a detailed analysis.

Figures 8(c) & 8(d) illustrates the processing time and waiting time of each tuple under different window sizes. Here is an explanation of the data in this figure: (i) As window size increases, PBO's processing time steadily rises, resulting in longer waiting periods for subsequent tuples. In contrast, our proposed MWP shows minimal fluctuations in processing times as window size changes. However, its waiting time does increase with larger window size; nevertheless, it remains significantly lower than that of PBO under similar conditions. This can be attributed to the fact that enlarging the window size increases the number of query structures (spanning trees and TRDs) to which new tuples can be added, thereby increasing the cost of processing a tuple. However, MWP effectively mitigates this rise by managing timestamp sets in TRDs across multiple intervals. (ii) With the increase of the sliding step length, PBO experiences gradual increases in both processing time and waiting time. Conversely, MWP exhibits no significant changes in processing time but

demonstrates rapid reductions in waiting time as the sliding step length increases due to the fact that tuples need not to wait for the expiration process before being processed.

Based on these experimental results from Figures 8(c) & 8(d), we observe that increasing the window size leads to an increased tail latency (as shown in Figure 8(b)) due to the rapid increases in waiting time. The decrease in tail latency for MWP and increase for PBO when sliding step length is increased can also be explained by their respective trends observed in waiting time.

Overall, these experiments demonstrate that MWP exhibits better scalability compared to PBO.



(a) SO (b) Yago3

Fig. 9. Throughput and tail latency of Q_3 on SO and Yago3 with different number of available threads

We now show how MWP scales with parallelization. Figure 9 presents the throughput and tail latency of MWP under different numbers of available threads in Thread Pool when executing Q_3 on SO with $|W| = 30$ and $Step(W) = 7$ and Yago3 with $|W| = 1M$ and $Step(W) = 50k$, respectively. For SO in Figure 9(a), the throughput exhibits an upward trend as the number of threads increases from 1 to 20, and it maintains stability with further increments in thread count after a brief dip. Notably, as the thread count increases from 1 to 30, there is an initial reduction in tail latency, followed by sustained stability as the thread count continues to rise, similar to the throughput. For Yago3 in Figure 9(b), the throughput continues to rise as the number of threads increases, but the upward trend gradually slows down with the increasing thread count. Simultaneously, latency decreases as the thread count increases, and the changes in the decreasing trend align with the throughput.

The trends of these two metrics can be explained as follows. The RPQ tasks are randomly assigned to threads in the Thread Pool for execution. When the available number of threads is limited, the processing ability cannot catch up with the task-arrival speed, leading to lower throughput and higher tail latency. As the available thread count increases, throughput gradually rises, and tail latency progressively decreases on both SO and Yago3. However, as the thread count continues to increase, the probability of different threads simultaneously maintaining the same TRD also keeps growing, causing more locking of threads that limits further improvements in both throughput and latency. Compared to Yago3, the SO dataset is much more dense, and increasing the thread count further has even a small negative impact on the performance due to management overhead. Note that the parallelization on such dense graphs may be improved by assigning the RPQ task of maintaining the same TRD to the same thread as much as possible to reduce the waiting time due to locking. But how in general parallelization the tasks on dense graphs remain a future research problem.

5.4.3 Performance of DGC:. The DGC on MWP only needs to be executed when the memory runs low. However, having too many old edges in TRDs can significantly reduce query performance by

increasing traversal overhead. We now explore the timing to explicitly execute DGC by monitoring the ratio of "old" nodes in the TRDs. The "old" nodes are defined as follows: for a given GC cut-off time t_{gc}, all nodes in TRDs whose timestamps are all less than t_{gc} are called the "old" nodes. Here the t_{gc} is continuously calculated every time 2k tuples have been processed. The condition to execute DGC is defined on the ratio: (1 - *the number of "old" nodes / all nodes in TRDs*) x 100%. When the ratio exceeds a given ratio threshold, referred to as the "GC threshold," the DGC is executed. When the GC threshold is set to 0%, we only execute DGC when the memory runs low. On the other hand, if this GC threshold is set to 100%, the DGC is executed for each window movement (adopting *naive GC strategy* if the ratio still exceeds the GC threshold after executing DGC). In this experiment, we explore the effect of different GC thresholds on system performance and evaluate how DGC affects memory freeing (represented by the number of nodes, edges, and TRDs).

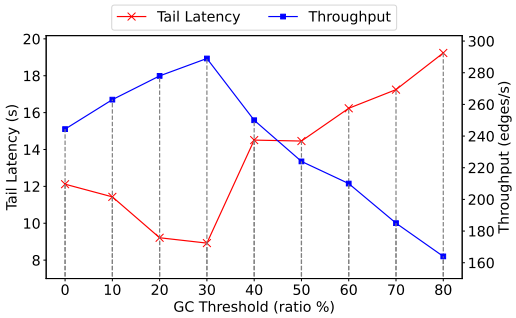


Fig. 10. Throughput and tail latency on SO with different GC threshold.

Figure 10 presents the trend of tail latency and throughput with query Q_3, where $|W| = 30$ and $Step(W) = 7$ on SO. The graph shows how these metrics change as the GC threshold increases from 0% to 80%, with an increment of 10%. The throughput initially rises as the GC threshold increases from 0% to 30% but decreases continuously beyond that point. Conversely, tail latency follows an opposite pattern. Both throughput and tail latency reach their best values at a GC threshold of 30%, which validates our analysis in Section 5.4.3 that pruning old tuples can improve query performance, but these advantages quickly diminish as we further increase the GC threshold since frequent GC due to large thresholds leads to performance degradation, particularly for tail latency.

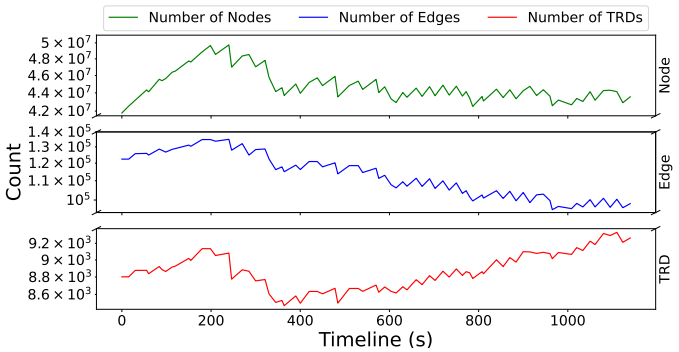
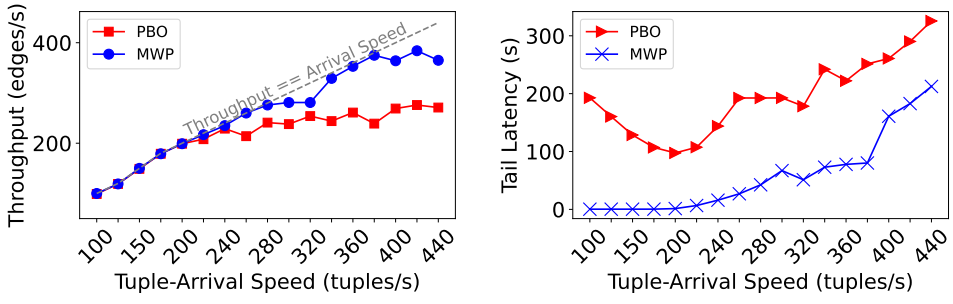


Fig. 11. The number of nodes, edges, and TRDs in memory.

We evaluated the performance of DGC in reducing memory consumption. Figure 11 illustrates the changes in the number of nodes, edges, and TRDs over time under the same experimental

conditions as before, except for fixing the GC threshold to 40%. These values were recorded every 10 seconds after the initial GC operation. The number of stored edges refers to the number of edges in TRDs, while the number of stored nodes represents the existing nodes in the TRDs. It is important to note that these metrics can only decrease right after any execution of DGC. In this experiment, DGC was executed approximately every 33 seconds on average and successfully removed old nodes, edges, and TRDs by 5% to 20% each time. This indicates that our DGC method effectively works for RPQ evaluation (see Figure 11).



(a) Throughput (b) Tail Latency

Fig. 12. Throughput and tail latency on SO with different tuple-arrival speeds.

5.4.4 Stability on Various Tuple-arrival Speed. For most real-world applications, it is essential to maintain a stable throughput and tail latency when querying under different tuple-arrival speeds [20, 39]. In this experiment, we controlled the arrival speed of tuples and evaluated the throughput (which should not exceed the arrival speed) and tail latency using query Q_3, with a window size of $|W| = 30$ and step length of $Step(W) = 7$.

Figure 12(a) illustrates that both methods show an increasing throughput as the tuple-arrival speed reaches 240 per second. After reaching this point, the throughput of PBO stabilizes while those of MWP continue to rise until the tuple-arrival speed reaches approximately 370 per second. It aligns with our previous experimental results and indicates their maximum "processing abilities".

Consider the tail latency when the tuple-arrival speed increases as shown in Figure 12(b). Both methods experience a rapid increase in tail latency once the speed surpasses their respective processing abilities. This is due to longer waiting times in the cache queue. When the tuple-arrival speed remains below their processing abilities, MWP demonstrates more stable tail latency compared to PBO. Therefore, we can conclude that MWP outperforms PBO in terms of stability and performance under varying tuple-arrival speeds.

Note that the tail latency of PBO decreases first as the tuple-arrival speed rises from 100 to 200, and then increases as the tuple-arrival speed continuously rises. This can be explained by the fact that PBO produces results at the end of each sliding window, and the low arrival speed prolongs the time in sliding to the next window, thus delaying the time to produce results.

6 RELATED WORK

Regular Path Queries: RPQs and corresponding evaluation methods are widely used in graph querying [1, 2, 17]. Earlier works on evaluating RPQs can be loosely divided into navigational and relational approaches. The navigational approaches including splitting RPQ by its rare labels [27, 35], evaluating RPQs using the concept of *Brzozowski derivatives* [36], bidirectional random

walks [43], or based on partial answers [44]. [5] provides a time- and space-efficient strategy that combines the Glushkov automaton [19] with ring index [4]. Other relational approaches use recursive queries, most focusing on the recursive SQL queries [14, 45] and relational algebra [18, 24]. Recent approaches leverage software or hardware acceleration techniques, including evaluating RPQs on top of field programmable gate arrays to enable parallelism [32] and using just-in-time compilation to directly evaluate the RPQ on the graph [41]. Note that all of these works focus on static graphs. [38] is the first work to consider persistent RPQ evaluation on streaming graphs. The detailed illustration for [38] is provided in Section 2.

Streaming Graph Processing: Existing streaming graph processing target developing efficient incremental algorithms to maintain matches of a given query as the graph changes. Examples include connectivity [25], shortest path [7], transitive closure [28]. Most related to our work is based on the sliding window model that maintains dynamic query structures for queries as the window moves, such as cycle detection [39], aggregate query [33], triangle counting [20], and path query [38]. The blocking phase between two neighboring windows affects query performance to varying degrees; ours is, to the best of our knowledge, the first work to consider parallel processing multiple windows on streaming graphs.

Streaming Processing System: There has been a significant amount of work on RDF stream processing, involves designing a communication interface for streaming RDF systems based on the Linked Data Notification protocol [9], providing a framework for publishing RDF streams on the web [3], extending SPARQLv1.0 [3] for reasoning and a complex event pattern matching on RDF streams, and computing entailments for streaming reasoning with schema-enhanced graph pattern matching based on the existence of RDF schemas [26]. The persistent query evaluation over RDF streams such as C-SPARQL [6], CQELS [29], SPARQL$_{stream}$ [10] and RSP-QL [13] are designed for SPARQLv1.0. Thus one cannot formulate path expressions such as RPQs, which is previously reported in [46]. Our contributions are orthogonal to existing work on streaming RDF systems, which is similar to [38].

7 CONCLUSION AND FUTURE WORK

In this paper, we present a multi-window parallel processing strategy called MWP and introduce a new data structure named TRD for the non-blocking parallel evaluation of RPQs on streaming graphs. Additionally, we propose an efficient approach for swift garbage collection on the TRDs. Experimental analysis conducted on real-world streaming graphs demonstrates the outstanding performance and scalability benefits of our approach. While our primary focus is on RPQs in this paper, the MWP strategy has the capability to accommodate various window-based queries, such as cycle detection, by addressing the blocking phase between adjacent windows and adopting a parallel processing strategy.

Furthermore, this work assumes a parallel framework with shared memory. To extend its applicability to distributed systems like *Flink* or *MPI*, it is necessary to design the map-reduce phases or communication messages for processing TRDs across workers. Additionally, a well-thought-out streaming partition strategy for TRDs should be devised to minimize communication costs between workers and balance the workload. Future research directions that we contemplate encompass: (i) supporting RPQs with complex constraints and (ii) applying the MWP strategy to other streaming graph queries.

ACKNOWLEDGMENTS

We would like to thank all the anonymous reviewers for their insightful comments and suggestions. This work was supported by NSFC (62272106, 62072113).

REFERENCES

[1] 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.

[2] 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.

[3] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. 2011. EP-SPARQL: a unified language for event processing and stream reasoning. In *Proceedings of the 20th international conference on World wide web*. 635–644.

[4] Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, Juan L Reutter, Javiel Rojas-Ledesma, and Adrián Soto. 2021. Worst-case optimal graph joins in almost no space. In *Proceedings of the 2021 International Conference on Management of Data*. 102–114.

[5] Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, and Javiel Rojas-Ledesma. 2022. Time-and space-efficient regular path queries. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 3091–3105.

[6] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. 2009. C-SPARQL: SPARQL for continuous querying. In *Proceedings of the 18th international conference on World wide web*. 1061–1062.

[7] Aaron Bernstein. 2013. Maintaining shortest paths under deletions in weighted directed graphs. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*. 725–734.

[8] Angela Bonifati, Wim Martens, and Thomas Timm. 2019. Navigating the Maze of Wikidata Query Logs. *the web conference* (2019).

[9] Jean-Paul Calbimonte. 2017. Linked data notifications for rdf streams. In *Proceedings of the Web Stream Processing workshop (WSP 2017) and the 2nd International Workshop on Ontology Modularity, Contextuality, and Evolution (WOMoCoE 2017) co-located with 16th International Semantic Web Conference (ISWC 2017)*. 22 October 2017.

[10] Jean-Paul Calbimonte, Oscar Corcho, and Alasdair JG Gray. 2010. Enabling ontology-based access to streaming data sources. In *The Semantic Web–ISWC 2010: 9th International Semantic Web Conference, ISWC 2010, Shanghai, China, November 7-11, 2010, Revised Selected Papers, Part I 9*. Springer, 96–111.

[11] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems*. 85–98.

[12] 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.

[13] Daniele Dell'Aglio, Jean-Paul Calbimonte, Emanuele Della Valle, and Oscar Corcho. 2015. Towards a unified language for RDF stream query processing. In *European Semantic Web Conference*. Springer, 353–363.

[14] Saumen Dey, Víctor Cuevas-Vicenttín, Sven Köhler, Eric Gribkoff, Michael Wang, and Bertram Ludäscher. 2013. On implementing provenance-aware regular path queries with relational query engines. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*. 214–223.

[15] Andrzej Ehrenfeucht and Paul Zeiger. 1974. Complexity measures for regular expressions. In *Proceedings of the sixth annual ACM symposium on Theory of computing*. 75–79.

[16] Chantat Eksombatchai, Pranav Jindal, Jerry Zitao Liu, Yuchen Liu, Rahul Sharma, Charles Sugnet, Mark Ulrich, and Jure Leskovec. 2018. Pixie: A system for recommending 3+ billion items to 200+ million users in real-time. In *Proceedings of the 2018 world wide web conference*. 1775–1784.

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

[18] Valeria Fionda, Giuseppe Pirrò, and Mariano P Consens. 2019. Querying knowledge graphs with extended property paths. *Semantic Web* 10, 6 (2019), 1127–1168.

[19] Victor Mikhaylovich Glushkov. 1961. The abstract theory of automata. *Russian Mathematical Surveys* 16, 5 (1961), 1.

[20] 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.

[21] 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)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hotcloud18/presentation/grewal

[22] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–14.

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

[24] Louis Jachiet, Pierre Genevès, Nils Gesbert, and Nabil Layaïda. 2020. On the optimization of recursive relational queries: Application to graph queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 681–697.

[25] Bruce M Kapron, Valerie King, and Ben Mountjoy. 2013. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*. SIAM, 1131–1142.

[26] Srdjan Komazec, Davide Cerri, and Dieter Fensel. 2012. Sparkwave: continuous schema-enhanced pattern matching over RDF data streams. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. 58–68.

[27] 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.

[28] Jakub Łącki. 2011. Improved deterministic algorithms for decremental transitive closure and strongly connected components. In *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 1438–1445.

[29] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. 2011. A native and adaptive approach for unified processing of linked streams and linked data. In *International Semantic Web Conference*. Springer, 370–388.

[30] David Lomet, Alan Fekete, Rui Wang, and Peter Ward. 2012. Multi-version concurrency via timestamp range conflict management. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 714–725.

[31] Farzaneh Mahdisoltani, Joanna Biega, and Fabian Suchanek. 2014. Yago3: A knowledge base from multilingual wikipedias. In *7th biennial conference on innovative data systems research*. CIDR Conference.

[32] Kento Miura, Toshiyuki Amagasa, Hiroyuki Kitagawa, R Bordawekar, and T Lahiri. 2019. Accelerating Regular Path Queries using FPGA.. In *ADMS@ VLDB*. 47–54.

[33] Jayanta Mondal and Amol Deshpande. 2014. Eagr: Supporting continuous ego-centric aggregate queries over large dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of data*. 1335–1346.

[34] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 439–455.

[35] Van-Quyet Nguyen and Kyungbaek Kim. 2017. Efficient regular path query evaluation by splitting with unit-subquery cost matrix. *IEICE TRANSACTIONS on Information and Systems* 100, 10 (2017), 2648–2652.

[36] Maurizio Nolé and Carlo Sartiani. 2016. Regular path queries on massive graphs. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*. 1–12.

[37] Nigini Oliveira, Michael Muller, Nazareno Andrade, and Katharina Reinecke. 2018. The exchange in StackExchange: Divergences between Stack Overflow and its culturally diverse participants. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (2018), 1–22.

[38] 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.

[39] 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.

[40] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. 2016. GraphJet: Real-time content recommendations at Twitter. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1281–1292.

[41] Frank Tetzel, Wolfgang Lehner, and Romans Kasperovics. 2020. Efficient Compilation of Regular Path Queries. *Datenbank-Spektrum* 20 (2020), 243–259.

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

[43] 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.

[44] Xin Wang, Junhu Wang, and Xiaowang Zhang. 2016. Efficient distributed regular path queries on rdf graphs using partial evaluation. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 1933–1936.

[45] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. 2013. Evaluation of SPARQL Property Paths via Recursive SQL. *AMW* 1087 (2013).

[46] Ying Zhang, Pham Minh Duc, Oscar Corcho, and Jean-Paul Calbimonte. 2012. SRBench: a streaming RDF/SPARQL benchmark. In *The Semantic Web–ISWC 2012: 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I 11*. Springer, 641–657.

Received July 2023; revised October 2023; accepted November 2023