Regular Path Query Evaluation on Streaming Graphs

Anil Pacaci
University of Waterloo
apacaci@uwaterloo.ca

Angela Bonifati
Lyon 1 University
angela.bonifati@univ-lyon1.fr

M. Tamer Özsu
University of Waterloo
tamer.ozsu@uwaterloo.ca

ABSTRACT

We study persistent query evaluation over streaming graphs, which is becoming increasingly important. We focus on navigational queries that determine if there exists a path between two entities that satisfies a user-specified constraint. We adopt the Regular Path Query (RPQ) model that specifies navigational patterns with labeled constraints. We propose deterministic algorithms to efficiently evaluate persistent RPQs under both *arbitrary* and *simple* path semantics in a uniform manner. Experimental analysis on real and synthetic streaming graphs shows that the proposed algorithms can process up to tens of thousands of edges per second and efficiently answer RPQs that are commonly used in real-world workloads.

CCS CONCEPTS

• **Information systems** → *Graph-based database models*; *Stream management*; Query languages for non-relational engines.

KEYWORDS

Regular Path Queries; Streaming Graphs; Persistent Query Evaluation

ACM Reference Format:
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 (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3389733

1 INTRODUCTION

Graphs are used to model complex interactions in various domains ranging from social network analysis to communication network monitoring, from retailer customer analysis to bioinformatics. Many real-world applications generate graphs over time as new edges are produced resulting in *streaming graph*s [62]. Consider an e-commerce application: each user and item can be modelled as a vertex and each user interaction such as clicks, reviews, purchases can be modelled as an edge. The system receives and processes a sequence of graph edges (as users purchase items, like them, etc). These graphs are unbounded, and the edge arrival rates can be very high: Twitter's recommendation system ingests 12K events/sec on average [37], Alibaba's user-product graph processes 30K edges/sec at its peak [60]. Recent experiments show that existing graph DBMSs are not able to keep up with the arrival rates of many real streaming graphs [57].

Efficient querying of streaming graphs is a crucial task for applications that monitor complex patterns and, in particular, *persistent queries* that are registered to the system and whose results are generated incrementally as the graph edges arrive. Querying streaming data in real-time imposes novel requirements in addition to challenges of graph processing: (i) graph edges arrive at a very high rate and real-time answers are required as the graph emerges, and (ii) graph streams are unbounded, making it infeasible to employ batch algorithms on the entire stream. Most existing work focus on the *snapshot* model, which assumes that graphs are static and fully available, and adhoc queries reflect the current state of the database (e.g., [24, 45, 63, 65, 69–71]). The *dynamic graph* model addresses the evolving nature of these graphs; however, algorithms in this model assume that the entire graph is fully available and they compute how the output changes as the graph is updated [15, 42, 47, 61].

In this paper, we study the problem of persistent query processing over streaming graphs, addressing the limitations of existing approaches. We adopt the Regular Path Query (RPQ) model that focuses on *path navigation*, e.g., finding pairs of users in a network connected by a path whose label (i.e., the labels of edges in the path) matches path constraints. RPQ specifies path constraints that are expressed using a regular expression over the alphabet of edge labels and checks whether a path exists with a label that satisfies the given regular expression [11, 54]. The RPQ model provides the basic navigational mechanism to encode graph queries, striking a balance between expressiveness and computational complexity [6, 7, 17, 64, 68]. Consider the streaming graph of a social network application presented in Figure 1(a). The query $Q_1 : (follows \circ mentions)^+$ in Figure 1(c) represents a

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389733

pattern for a real-time notification query where user x is notified of other users who are connected by a path whose edge labels are even lengths of alternating *follows* and *mentions*. At time $t = 18$, the pair of users (x, y) is connected by such a path, shown by bold edges in Figure 1(b).

It is known that for many streaming algorithms the space requirement is lower bounded by the stream size [10]. Since the stream is unbounded, deterministic RPQ evaluation is infeasible without storing all the edges of the graph (by reduction to the length-2 path problem that is infeasible in sublinear space [30]). In streaming systems, a general solution for bounding the space requirement is to evaluate queries on a *window* of data from the stream. In a large number of applications, focusing on the most recent data is desirable. Thus, the windowed evaluation model not only provides a tool to process unbounded streams with bounded memory but also restricts the scope of queries on recent data, a desired feature in many streaming applications. In this paper we consider the *time-based sliding window* model where a fixed size (in terms of time units) window is defined that slides at well-defined intervals [33]. In our context, new graph edges enter the window during the window interval, and when the window slides, some of the "old" edges leave the window (i.e., expire). Managing this window processing as part of RPQ evaluation is challenging and our solutions address the issue in a uniform manner.

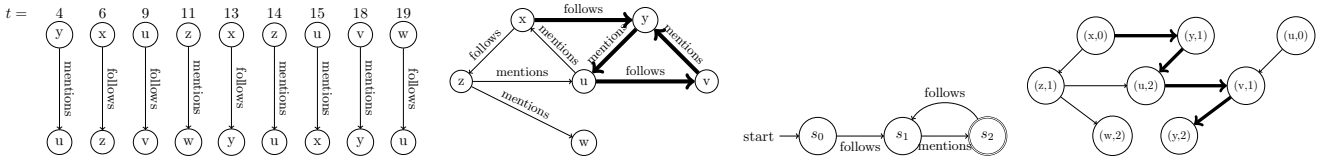
In this paper, for the first time, we study the design space of persistent RPQ evaluation algorithms in two main dimensions: the path semantics they support and the result semantics based on application requirements. Along the first dimension, we propose efficient incremental algorithms for both *arbitrary* and *simple* path semantics. The former allows a path to traverse the same vertex multiple times, whereas under the latter semantics a path cannot traverse the same vertex more than once [7]. Consider the example graph given in Figure 1(b); the sequence of vertices $\langle x, y, u, v, y \rangle$ is a valid path for query Q_1 with arbitrary path semantics whereas the simple path semantics does not traverse this path as it visits vertex y twice. Along the second dimension, we consider *append-only* streams where tuples in the window expire only due to window movements, then extend our algorithms to support *explicit deletions* to deal with cases where users/applications might explicitly delete a previously arrived edge. We use the *negative tuples* approach [35] to process explicit deletions. Table 1 presents the combined complexities of the proposed algorithms in each quadrant in terms of amortized cost.

To the best of our knowledge, these are the first streaming algorithms to address RPQ evaluation on sliding windows over streaming graphs under both arbitrary (§3) and simple path semantics (§4). Our proposed algorithm for streaming RPQ evaluation under arbitrary path semantics incrementally maintains results for a query Q_R on a sliding window W over a streaming graph S as new edges enter and old edges expire due to window slide. We follow the implicit window semantics, where newly arriving edges are processed as they arrive (and new results appended to the output stream) while the removal of expired edges occur at user-specified slide intervals. We then turn our attention to simple path semantics (§4). The static version of the RPQ evaluation problem is NP-hard in its most general form [54], which has caused existing work to focus only on arbitrary path semantics. Yet, it is proven to be tractable when restricted to certain classes of regular expressions or by imposing restrictions on the graph instances [13, 54]. A recent analysis [18, 19] of real-world SPARQL logs shows that a large portion of RPQs posed by users does indeed fall into those tractable classes, motivating the design of efficient algorithms for streaming RPQ evaluation under simple path semantics. Our proposed algorithm admits efficient solutions for streaming RPQs under simple path semantics in the absence of conflicts, a condition on the cyclic structure of graphs that enables efficient batch algorithms (precisely defined in §4.1) [54]. Indeed, this algorithm has the same amortized time complexity as the proposed algorithm for arbitrary path semantics under the same condition. The proposed algorithms incrementally maintain query answers as the window slides thus eliminating the computational overhead of the naive strategy of batch computation after each window movement. Furthermore, they support negative tuples to accommodate applications where users might explicitly delete a previously inserted edge. Albeit relatively rare, explicit deletions are a desired feature of real-world applications that process and query streaming graphs, and it is known to require special attention [34]. We show that window management and explicit deletions can be handled in a uniform manner using the same machinery (§3.2). Finally, we empirically evaluate the performance of our proposed algorithms using a variety of real-world and synthetic streaming graphs on real-world RPQs that cover more than 99% of all recursive queries abundantly found in massive Wikidata query logs [19] (§5).

Table 1: Amortized time complexities of the proposed algorithms for a streaming graph S with m edges and n vertices and RPQ Q_R whose automata has k states.

Path Semantics \ Result Semantics	Append-Only	Explicit Deletions
Arbitrary (§3)	$O(n \cdot k^2)$	$O(n^2 \cdot k)$
Simple[1] (§4)	$O(n \cdot k^2)$	$O(n^2 \cdot k)$.

[1]These results hold in the absence of conflicts, a condition on cyclic structure of the query and graph that is precisely defined in §4.1.



(a) Streaming Graph S (b) Snapshot Graph G (c) Query Graph Q_1 (d) Product Graph $P_{G,A}$

Figure 1: (a) A streaming graph S of a social networking application, (b) the snapshot of S at $t = 18$, (c) automaton for the query $Q_1 : (\textit{follows} \circ \textit{mentions})^+$, and (d) the product graph $P_{G,A}$.

2 PRELIMINARIES

Definition 1 (Graph). *A directed labeled graph is a quintuple $G = (V, E, \Sigma, \psi, \phi)$ where V is a set of vertices, E is a set of edges, Σ is a set of labels, $\psi : E \rightarrow V \times V$ is an incidence function and $\phi : E \rightarrow \Sigma$ is an edge labelling function.*

Definition 2 (Streaming Graph Tuple). *A streaming graph tuple (sgt) t is a quadruple (τ, e, l, op) where τ is the event (application) timestamp of the tuple assigned by the data source, $e = (u, v)$ is the directed edge with source vertex u and target vertex v, $l \in \Sigma$ is the label of the edge e and op is the type of the edge, i.e., insert (+) or delete (−) .*

Definition 3 (Streaming Graph). *A streaming graph S is a constantly growing sequence of streaming graph tuples (sgts) $S = \langle t_1, t_2, \cdots, t_m \rangle$ in which each tuple t_i arrives at a particular time τ_i ($\tau_i < \tau_j$ for $i < j$).*

In this paper, we assume that sgts[2] are generated by a single source and arrive in source timestamp order τ_i, which defines their ordering in the stream. We leave the problem of out-of-order delivery as future work.

Definition 4 (Time-based Window). *A time-based window W over a streaming graph S is defined by a time interval $(W^b, W^e]$ where W^b and W^e are the beginning and end times of window W and $W^e - W^b = |W|$. The window contents $W(c)$ is the multiset of sgts where the timestamp τ_i of each sgt t_i is in the window interval, i.e., $W(c) = \{t_i \mid W_b < \tau_i \leq W_e\}$.*[3]

Definition 5 (Time-based Sliding Window). *A time-based sliding window W with a slide interval β is a time-based window that progresses every β time units. At any time point τ, a time-based sliding window W with a slide interval β defines a time interval $(W^b, W^e]$ where $W^e = \lfloor \tau/\beta \rfloor \cdot \beta$ and $W^b = W^e - |W|$. The contents of W at time τ defines a snapshot graph $G_{W,\tau} = (V_{W,\tau}, E_{W,\tau}, \Sigma_{W,\tau}, \psi, \phi)$ where $E_{W,\tau}$ is the set of all edges that appear in sgts in W and $V_{W,\tau}$ is the set of vertices that are endpoints of edges in $E_{W,\tau}$.*

Figure 1(a) shows an excerpt of a streaming graph S at $t = 19$. Figure 1(b) shows the snapshot graph $G_{W,18}$ defined by window W with $|W| = 15$ over this graph S.

A time-based sliding window W might progress either at every time unit, i.e. $\beta = 1$ (*eager evaluation*; resp. *expiration*) or at $\beta > 1$ intervals (*lazy evaluation*; resp. *expiration*) [59]. Eager evaluation produces fresh results but windows can be expired lazily if queries do not produce premature expirations [34]. We use eager evaluation ($\beta = 1$) but lazy expiration ($\beta > 1$) as it enables us to separate window maintenance from processing of incoming sgts (§3.1).

Definition 6 (Path and Path Label). *Given $u, v \in V$, a path p from u to v in graph G is a sequence of edges $u \xrightarrow{p} v : \langle (v_0, l_0, v_1), \cdots, (v_{n-1}, l_{n-1}, v_n) \rangle$ where $v_0 = u$ and $v_n = v$. The label of a path p is denoted by $\phi(p) = l_0 l_1 \cdots l_{n-1} \in \Sigma^*$.*

Definition 7 (Regular Expression & Regular Language). *A regular expression R over an alphabet Σ is defined as $R ::= \epsilon \mid a \mid R \circ R \mid R + R \mid R^*$ where (i) ϵ denotes the empty string, (ii) $a \in \Sigma$ denotes a character in the alphabet, (iii) $\circ$ denotes the concatenation operator, (iv) $+$ denotes the alternation operator, and (v) $*$ represents the Kleene star. We use $\neg$ to denote the negation of an expression, and R^+ to denote 1 or more repetitions of R.A regular language $L(R)$ is the set of all strings that can be described by the regular expression R.*

Definition 8 (Regular Path Query – RPQ). *A Regular Path Query Q_R asks for pairs of vertices (u, v) that are connected by a path p from u to v in graph G, where the path label $\phi(p)$ is a word in the regular language defined by the regular expression R over the graph's edge labels Σ, i.e., $\phi(p) \in L(R)$. Answer to query Q_R over G, $Q_R(G)$, is the set of all pairs of vertices that are connected by such paths.*

Sliding windows adhere to two alternative semantics: *implicit* and *explicit* [35]. Implicit windows add new results to query output as new sgts arrive and do not invalidate the previously reported results upon their expiry as the window moves. In the absence of explicit edge deletions, the query results are monotonic. Under this model, the result set of a streaming RPQ over a streaming graph S and a sliding window W at time τ contains all paths in all previous snapshot graphs $G_{W,\pi}$ where $0 < \pi \leq \tau$, i.e., $Q_R(S, W, \tau) = \bigcup_{0<\pi\leq\tau} Q_R(G_{W,\pi})$. Alternatively, explicit windows remove

[2]We use "sgt" and "tuple" interchangeably.

[3]We use W interchangeably to refer to a window interval or its contents.

previously reported results involving tuples (i.e., sgts) that have expired from the window; hence, persistent queries with explicit windows are akin to incremental view maintenance. Under this model, the result set of a streaming RPQ over a streaming graph S and a sliding window W at time τ contains only the paths in the snapshot $G_{W,\tau}$ of the streaming graph, i.e., $Q_R(S, W, \tau) = Q_R(G_{W,\tau})$. Explicit windows, by definition, produce non-monotonic results as previous results are negated when the window moves [35]. We employ the implicit window model in this paper as it enables us to preserve the monotonicity of query results and produce an append-only stream of query results (in the absence of explicit deletions).

Definition 9 (Streaming RPQ). *A streaming RPQ is defined over a streaming graph S and a sliding window W. A pair of vertices (u, v) is an answer for a streaming RPQ, Q_R, at time τ if there exists a path p between u and v in $G_{W,\tau}$, i.e., all edges in p are in window W. We define the timestamp $p.ts$ of a path p as the minimum timestamp among all edges of p. Under the implicit window model, the result set of a streaming RPQ Q_R over a streaming graph S and a sliding window W is an append-only stream of pairs of vertices (u, v) where there exists a path p between u and v with label $\phi(p) \in L(R)$ and all the edges in p are at most one window length, i.e., $|W|$ time units, apart. Formally:*

$$Q_R(S, W, \tau) = \{(u, v) \mid \exists p : u \xrightarrow{p} v \wedge \phi(p) \in L(R) \wedge \max_{e \in p}(e.ts) < p.ts + |W| \leq \tau\}$$

Definition 10 (Deterministic Finite Automaton). *Given a regular expression R, $A = (S, \Sigma, \delta, s_0, F)$ is a Deterministic Finite Automaton (DFA) for $L(R)$ where S is the set of states, Σ is the input alphabet, $\delta : S \times \Sigma \to S$ is the state transition function, $s_0 \in S$ is the start state and $F \subseteq S$ is the set of final states. δ^* is the extended transition function defined as:*

$$\delta^*(s, w \circ a) = \delta(\delta^*(s, w), a)$$

where $s \in S$, $a \in \Sigma$, $w \in \Sigma^*$, and $\delta^*(s, \epsilon) = s$ for the empty string ϵ. We say that a word w is in the language accepted by A if $\delta^*(w, s_0) = s_f$ for some $s_f \in F$.

Definition 11 (Product Graph). *Given a graph $G = (V, E, \Sigma, \phi)$ and a DFA $A = (S, \Sigma, \delta, s_0, F)$, we define the product graph $P_{G,A} = (V_P, E_P, \Sigma, \phi_P)$ where $V_P = V \times S$, $E_P \subseteq V_P \times V_P$, and $((u, s), (v, t))$ is in E_P iff $(u, v) \in E$ and $\delta(s, \phi(u, v)) = t$.*

Figure 1(d) shows the product graph of $G_{W,18}$ (Figure 1(b)) and the DFA A of the query Q_1 (Figure 1(c)).

For a given RPQ, Q_R, we first use Thompson's construction algorithm [66] to create a NDFA that recognizes the language $L(R)$, then create the equivalent minimal DFA, A, using Hopcroft's algorithm [41]. In the rest of the paper, we use A and the product graph $P_{G,A}$ to describe the proposed algorithms for RPQ evaluation in the streaming graph model.

3 RPQ WITH ARBITRARY SEMANTICS

We now study the problem of RPQ evaluation over sliding windows of streaming graphs under arbitrary path semantics, i.e., finding pairs of vertices $(u, v) \in V$ where (i) there exists a (not necessarily simple) path p between u and v with a label $\phi(p) \in L(R)$, and (ii) timestamps of all edges in path p are in the range of window W. We first consider append-only streams where the query results are monotonic (under *implicit window* model). Then, we discuss extensions to support negative tuples to handle explicit edge deletions.

Batch Algorithm. RPQs can be evaluated in $O(n \cdot m \cdot k^2)$ time under arbitrary path semantics (under the assumption that there are more edges than isolated vertices in G). Given a product graph $P_{G,A}$, there is a path p in G from x to y with label w that is in $L(R)$ if and only if there is a path in $P_{G,A}$ from (x, s_0) to (y, s_f), where $s_f \in F$. The batch RPQ evaluation algorithm under arbitrary path semantics traverses the product graph $P_{G,A}$ by simultaneously traversing graph G and the automaton A.

3.1 RPQ over Append-Only Streams

We first present an incremental algorithm for Regular Arbitrary Path Query (RAPQ) evaluation over append-only streams. As noted above, using implicit window semantics, RAPQs are monotonic, i.e., $Q_R(S, W, \tau) \subseteq Q_R(S, W, \tau + \epsilon)$ for all $\tau, \epsilon \geq 0$. Algorithm **RAPQ** consumes a sequence of append-only tuples (i.e., op is +), and simultaneously traverses the the snapshot graph $G_{W,\tau}$ of the window W over a graph stream S and the automaton A_τ of Q_R for each tuple t_τ, and it produces an append-only stream of results for $Q_R(S, W, \tau)$. As in the case of the batch algorithm, such traversal of the snapshot graph $G_{W,\tau}$ guided with the automaton A emulates a traversal of the product graph $P_{G,A}$.

Algorithm RAPQ:

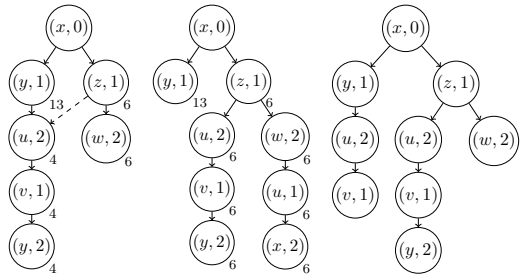
```
   input : Incoming tuple t_τ = (τ, e_τ, l, op), e_τ = (u, v),
1  G_{W,τ} ← G_{W,τ−1} (op) e_τ
2  ExpiryRAPQ(G_{W,τ}, T_x, τ) ∀T_x ∈ Δ // with β intervals
3  set of results R ← ∅
4  foreach T_x ∈ Δ do
5      foreach s, t ∈ S where t = δ(s, l) do
6          if (u, s) ∈ T_x ∧ (u, s).ts > τ − |W| then
7              if (v, t) ∉ T_x ∨ (v, t).ts < min((u, s).ts, τ) then
8                  R ← R+Insert(T_x, (u, s), (v, t), e = (u, v))
9  Q_R(S, W, τ) ← Q_R(S, W, τ − 1) + R
```

Definition 12 (Δ Tree Index). *Given an automaton A for a query Q_R and a snapshot $G_{W,\tau}$ of a streaming graph S at time τ, Δ is a collection of spanning trees where each tree T_x is rooted at a vertex $x \in G_{W,\tau}$ for which there is a corresponding node in the product graph of A and $G_{W,\tau}$ with the start state*

Algorithm Insert:

input : Spanning Tree T_x rooted at (x, s_0), parent node (u, s), child node (v, t), Edge $e = (u, v)$,
output: The set of results R

1 $R \leftarrow \emptyset$
2 $(v, t).pt = (u, s)$
3 $(v, t).ts = min(e.ts, (u, s).ts)$
4 **if** $(v, t) \notin T_x$ **then**
5 **if** $t \in F$ **then** $R \leftarrow R + (x, v)$;
6 **foreach** *edge* $(v, w) \in W_{G,\tau}$ *s.t.* $\delta(t, \phi(v, w)) = q$ **do**
7 **if** $(w, q) \notin T_x \vee (w, q).ts < min((v, t).ts, (v, w).ts)$ **then**
8 $R \leftarrow R$+Insert(T_x, (v, t), (w, q), $e = (v, w)$)
9 **return** R



(a) t=18 (b) t=19 (c) t=18

Figure 2: A spanning tree $T_x \in \Delta$ constructed by Algorithm RAPQ for the example given in Figure 1 (a) before and (b) after the edge $e = (w, u)$ with label *follows* at $t = 19$ is consumed. (c) The same spanning tree constructed by constructed by Algorithm RSPQ at $t = 18$ (§4.1). The timestamp of a node is given at the corner.

s_0, *i.e.*, $\Delta = \{T_x \mid x \in G_{W,\tau} \wedge (x, s_0) \in V_{P_{G,A}}\}$.[4]

A node $(u, s) \in T_x$ at time τ indicates that there is a path p in $G_{W,\tau}$ from x to u with label $\phi(p)$ and timestamp $p.ts$ such that $\delta^*(s_0, \phi(p)) = s$ and $(\tau - |W|) < p.ts \leq \tau$, i.e., word $\phi(p) \in \Sigma^*$ takes the automaton A from the initial state s_0 to a state s and the timestamp of the path is in the window range. Each node (u, s) in a tree T_x maintains a pointer $(u, s).pt$ to its parent in T_x. The timestamp $(u, s).ts$ is the minimum timestamp among all edges in the path p, following Definition 9.

The proposed algorithm continuously updates $G_{W,\tau}$ upon arrival of new edges and expiry of old edges. In addition to $G_{W,\tau}$, it maintains a tree index (Δ) to support efficient incremental RPQ evaluation that enables efficient RPQ evaluation on sliding windows over streaming graphs.

Example 1. Figure 2(a) depicts a spanning tree $T_x \in \Delta$ for the streaming graph S and the RPQ Q_1 (Figure 1) at time $t = 18$. The tree in Figure 2(a) is constructed by traversing $P_{G,A}$ starting from node $(x, 0)$, visiting nodes $(y, 1)$, $(u, 2)$, $(v, 1)$ and $(y, 2)$, forming the path from the root to node $(y, 2)$. Similar to the batch algorithm, this corresponds to the traversal of the path $\langle x, y, u, v, y\rangle$ in the snapshot of the streaming graph (Figure 1(b)) with label $\langle$*follows, mentions, follows, mentions*$\rangle$ taking the corresponding automaton (Figure 1(c)) from state 0 to 2 via path $\langle 0, 1, 2, 1, 2\rangle$. The timestamp of node $(y, 2) \in T_x$ at $t = 18$ is 4 as the edge with the minimum timestamp on the path from the root is $(y,$ *mentions*$, u)$ with $\tau = 4$.

Lemma 1. *The proposed Algorithm **RAPQ** maintains the following two invariants of the Δ tree index:*

(1) *A node (u, s) with timestamp ts is in T_x if there exists a path p in $G_{W,\tau}$ from x to u with label $\phi(p)$ and timestamp $(u, s).ts$ such that $s = \delta^*(s_0, \phi(p))$ and $(u, s).ts = p.ts \in (\tau - |W|, \tau]$, i.e., there exists a path p in G_τ from x to u with label $\phi(p)$ such that $\phi(p)$ is a prefix of a word in $L(R)$ and all edges are in the window W.*
(2) *At any given time τ, a node (u, s) appears in a spanning tree T_x at most once with a timestamp in the range $(\tau - W, \tau]$.*

Proof Sketch: The proof[5] constructs an inductive argument for the first invariant: a node (v, t) is added to T_x if there exists an edge (u, v) with label l such that there is a transition from state s to t in the corresponding automaton with label l and (u, s) is reachable from the root node (x, s_0), i.e., $t = \delta(s, l)$ and $(u, s) \in T_x$. The second invariant follows from Line 4 of Algorithm **Insert**, which adds node (v, t) to T_x only if it is not already in T_x. □

The first invariant allows tracing all reachable nodes from a root node (x, s_0) whereas the second invariant prevents Algorithm **RAPQ** from visiting the same vertex in the same state more than once in the same tree. Consider the example in Figure 2(a): node $(u, 2)$ is not added as a child of the node $(x, 1)$ after consuming edge $(x, u) \in S$ with label *mentions* since $(u, 2)$ is already reachable from $(x, 0)$.

Algorithm **ExpiryRAPQ** is invoked at pre-defined slide intervals to remove expired nodes from Δ. For each $T_x \in \Delta$, it identifies the set of candidate nodes whose timestamps are not in $(\tau - |W|, \tau]$ (Line 2) and temporarily removes those from T_x (Line 3). For each candidate (v, t), Algorithm **Insert** finds an incoming edge from another valid node in T_x (Line 7) and it reconnects the subtree rooted at (v, t) to T_x. Nodes with no valid incoming edges are permanently removed from T_x. Algorithm **ExpiryRAPQ** might traverse the entire snapshot graph $G_{W,\tau}$ in the worst case. This can be used to undo previously reported results if explicit window semantics is required (Line 9), yet, we only do so to process explicit deletions as described in §3.2.

[4] In the remainder, we use the term "vertex" to denote endpoints of sgts, and the term "node" to denote vertex-state pairs in spanning trees.

[5] The full proofs that are not in the paper are given in the long version [56]

Example 2. In the example of Figure 2(b) assume that window size is 15 time units. Upon arrival of edge (w, u) with label *follows* at $t = 19$, nodes $(u, 1)$ and $(x, 2)$ are added to T_x as descendants of $(w, 2)$. Also, paths leading to nodes $(u, 2)$, $(v, 1)$ and $(y, 2)$ are expired as their timestamp is 4 (due to the edge (y, u) with a timestamp 4). Algorithm **ExpiryRAPQ** searches incoming edges of vertex u in $G_{W,\tau}$ and identifies that there exists a valid edge (z, u) with label *mentions* and timestamp 14. As a result, node $(u, 2)$ and its subtree is reconnected to node $(z, 1)$.

Theorem 3. *Algorithm **RAPQ** is correct and complete.*

Proof Sketch: It is necessary to show that (x, u) is added to $Q_R(G_{W,\tau})$ at time τ if and only if there exists a path p the snapshot graph $G_{W,\tau}$ from x to u satisfying Q_R where all edges are in window W. The essential is the only if direction, that is: a node (v, t) may only be inserted into the spanning tree T_x as a result of traversing a path in the product graph from (x, s_0) to (v, t) where all edges are in window W. □

Algorithm ExpiryRAPQ:

input : Window $G_{W,\tau}$, timestamp τ, Spanning tree T_x
output: The set of invalidated results R_I

1 $R_I \leftarrow \emptyset$
2 $P = \{(v, t) \in T_x \mid (v, t).ts \leq \tau - |W|\}$ // candidate nodes
3 $T_x \leftarrow T_x \setminus P$ // prune T_x
4 **foreach** $(v, t) \in P$ **do**
5 **foreach** $(u, v) \in W_{G,\tau}$ **do**
6 **if** $(u, s) \in T_x \wedge t = \delta(s, \phi(u, v))$ **then**
7 $P \leftarrow P \setminus$ Insert$(T_x, (u, s), (v, t), (u, v))$
8 **foreach** $(v, t) \in P$ **do**
9 **if** $t \in F$ **then** $R_I \leftarrow R_I + (x, v)$;
10 **return** R_I

Theorem 4. *The amortized cost of Algorithm **RAPQ** is $O(n \cdot k^2)$, where n is the number of distinct vertices in the window W and k is the number of states in the corresponding automaton A of the the query Q_R.*

Proof. Consider a tuple t_τ with an edge $e = (u, v)$ and label l arriving for processing. Updating window $G_{W,\tau}$ with edge e (Line 1) takes constant time. Thus, the time complexity of Algorithm **RAPQ** is the total number of times Algorithm **Insert** is invoked.

First, we show that the amortized cost of updating a single spanning tree T_x rooted at (x, s_0) is constant in window size. For an edge (u, v) with label l, there could be k many parent nodes $(u, s) \in T_x$ for each state s, and thus there could be at most k^2 invocations of Algorithm **Insert** with child node (v, t), for each state t. Upon arrival of the edge $e = (u, v)$, Algorithm **Insert** is invoked with nodes (u, s) as parent and (v, t) as child either when (u, s) is already in T_x at time τ, $\tau - |W| < (u, s).ts \leq \tau$ (Line 8 in Algorithm **RAPQ**), or when (u, s) is added to T_x at a later point in time $(u, s).ts > \tau$ (Line 8 in Algorithm **Insert**). Note that Algorithm **Insert** is invoked with these parameters at most once as Line 4 of Algorithm **Insert** extends a node (v, t) only if it is not in T_x. The second invariant (Lemma 1) guarantees that (u, s) appears in a spanning tree T_x at most once. Therefore, Algorithm **Insert** is invoked at most $m \cdot k^2$ over a sequence of m tuples. As there are at most n spanning trees in Δ, one for each $x \in G_{W,\tau}$, the total amortized cost is $O(n \cdot k^2)$. □

Consequently, the amortized cost of Algorithm **RAPQ** is $O(n)$ in terms of the number of vertices in the snapshot graph $G_{W,\tau}$. As described previously, Algorithm **ExpiryRAPQ** might traverse the entire product graph and its worst case complexity is $O(m \cdot k^2)$. Therefore, the total cost of window maintenance over n spanning trees is $O(n \cdot m \cdot k^2)$. This cost is amortized over the slide interval β.

3.2 Explicit Deletions

The majority of real-world applications process append-only streaming graphs where existing tuples in the window expire only due to window movements. However, there are applications that require users to explicitly delete a previously inserted edge. Algorithm **ExpiryRAPQ** proposed in §3.1 can be utilized to support such explicit edge deletions. In the append-only case, a node (v, t) in a spanning tree $T_x \in \Delta$ is only removed when its timestamp falls outside the window range. An explicit deletion might require $(v, t) \in T_x$ to be removed if the deleted edge is on the path from (x, s_0) to (v, t) in the spanning tree T_x. We utilize Algorithm **ExpiryRAPQ** to remove such nodes so that explicit deletions and window management are handled in a uniform manner.

Definition 13 (Tree-Edge). *Given a spanning tree T_x, an edge $e = (u, v)$ with label l is a tree-edge w.r.t T_x if (u, s) is the parent of (v, t) in T_x and there is a transition from state s to t with label l, i.e., $(u, s) \in T_x$, $(v, t) \in T_x$, $t = \delta(s, l)$, and $(v, t).pt = (u, s)$.*

Algorithm **Delete** finds spanning trees where a deleted edge (u, v) is a tree-edge (Line 3) as per Definition 13. Deletion of the tree-edge from (u, s) to (v, t) in T_x disconnects (v, t) and its descendants from T_x. Algorithm **Delete** traverses the subtree rooted at (v, t) and sets the timestamp of each node to $-\infty$, essentially marking them as expired (Line 5). Algorithm **ExpiryRAPQ** processes each expired node in Δ and checks if there exists an alternative path comprised of valid edges in the window. Algorithm **Delete** invokes Algorithm **ExpiryRAPQ** (Line 7) to manage explicit deletions using the same machinery of window management. Deletion of a non-tree edge, on the other hand, leaves spanning trees unchanged so no modification is necessary other than updating the window content $G_{W,\tau}$.

Algorithm Delete:

```
input : Incoming tuple t_τ = (τ, e_τ, l, −), e_τ = (u, v),
        Window G_{W,τ−1}
output: The set of invalidated results R_I
1 R_I ← ∅
2 foreach T_x ∈ Δ do
3     foreach
        s, t ∈ S | t = δ(s, l) ∧ (v, t) ∈ T_x ∧ (v, t).pt = (u, s) do
4         T_(x,v,t) ← the subtree of (v, t) in T_x
5         foreach (w, q) ∈ T_(x,v,t) do
6             (w, q).ts = −∞
7     R_I ← R_I ∪ ExpiryRAPQ(W_{G,τ}, T_x, τ)
8 return R_I
```

Theorem 5. *The amortized cost of Algorithm* ***Delete*** *is* $O(n^2 \cdot k)$ *over a sequence of explicit edge deletions.*

Proof. First, we evaluate the cost of an explicit deletion over a single spanning tree $T_x \in \Delta$, rooted at (x, s_0). Given a negative tuple with edge (u, v) and label l, Line 3 identifies the corresponding set of tree-edges in T_x in $O(n \cdot k)$ time. For each such tree-edge from (u, s) to (v, t) in T_x, Line 4 traverses the subtree of T_x starting from (v, t) to identify the set of potentially expired nodes, and sets their timestamps to $-\infty$. Line 7 invokes Algorithm **ExpiryRAPQ** to process all expired nodes in T_x in $O(m \cdot k^2)$ time. There are at most $m \cdot k^2$ edges in the product graph $PG_{G,A}$. The amortized time complexity of maintaining a single spanning tree $T_x \in \Delta$ over a sequence of m explicit deletions is $O(n \cdot k)$ since at most $n \cdot k$ of those edges are tree-edges. Algorithm **Delete** does not need to process non-tree edges as the removal of a non-tree edge only need to update the window $G_{W,\tau}$ with a constant cost. Thus, the amortized cost of Algorithm **Delete** over a sequence of m explicit edge deletions is $O(n^2 \cdot k)$. □

4 RPQ WITH SIMPLE PATH SEMANTICS

In this section, we turn our attention to the problem of persistent RPQ evaluation on streaming graphs under the simple path semantics, that is finding pairs of vertices $(u, v) \in V$ where there exists a simple path (no repeating vertices) p between u and v with a path label $\phi(p)$ in the language $L(R)$.

The decision problem for Regular Simple Path Query (RSPQ), i.e., deciding whether a pair of vertices $(u, v) \in V$ is in the result set of a RSPQ Q_R, is NP-complete for certain fixed regular expressions, making the general problem NP-hard [54]. Mendelzon and Wood [54] show that there exists a batch algorithm to evaluate RSPQs on static graphs in the absence of conflicts, a condition on the cyclic structure of the graph G and the regular language $L(R)$ of the query Q_R.

Definition 14 (Suffix Language). *Given an automaton* $A = (S, \Sigma, \delta, s_0, F)$*, the suffix language of a state* s *is defined as* $[s] = \{w \in \Sigma^* \mid \delta^*(s, w) \in F\}$*; that is, the set of all strings that take* A *from state* s *to a final state* $s_f \in F$*.*

Definition 15 (Containment Property). *Automaton* $A = (S, \Sigma, \delta, s_0, F)$ *has the suffix language containment property if for each pair* $(s, t) \in S \times S$ *such that* s *and* t *are on a path from* s_0 *to some final state and* t *is a successor of* s*,* $[s] \supseteq [t]$*.*

We compute and store the suffix language containment relation for all pairs of states during query registration to the system, and use it to detect conflicts (Definition 16).

Definition 16 (Conflict). *A conflict exists at a vertex* u *if and only if a traversal of the product graph* $P_{G,A}$ *starting from an initial node* $(x, s_0) \in P_{G,A}$ *visit node* u *in states* s *and* t*, and* $[s] \not\supseteq [t]$*. In other words, a tree* T_X *is said to have a conflict between states* s *and* t *at vertex* u *if* (u, s) *is an ancestor of* (u, t) *in the spanning tree* T_x *and* $[s] \not\supseteq [t]$*.*

Example 6. Consider the streaming graph and the query in Figure 1 and the its spanning tree given in Figure 2(a). The node $(y, 2)$ is added as a child of the node $(v, 1)$ when edge (v, y) arrives at $t = 18$. Based on Definition 16, there is a conflict at vertex v as the path p from the root node $(x, 0)$ visits the vertex v at states 1 and 2, and $[1] \not\supseteq [2]$.

Batch Algorithm. Similar to the batch algorithm in §3, the batch RSPQ algorithm [54] simultaneously traverses the graph G and the DFA A. For every vertex $x \in V$, it maintains a set of markings that is used to prevent visiting a vertex more than once in the same state in T_x. A node (u, s) is marked only if the depth-first traversal starting from the node (u, s) is completed and no conflict is detected. The batch RSPQ algorithm has $O(n \cdot m)$ time complexity in terms of the size of the graph G in the absence of conflicts – the same as the batch RAPQ algorithm (§3). A query Q_R on a graph G is conflict-free if: (i) the DFA A of Q_R has the containment property, (ii) G is an acyclic graph, or (iii) G complies with a cycle constraint compatible with R. In following, we study the persistent RSPQ evaluation problem and show that the notion of *conflict-freedom* [54] is applicable to sliding windows over streaming graphs, admitting an efficient evaluation algorithm in the absence of conflicts.

4.1 Append-only Streams

First, we present an incremental algorithm for RSPQ evaluation based on its RAPQ counterpart (Algorithm **RSPQ**) with implicit window semantics and show that this algorithm admits efficient solutions under the same conditions as the batch algorithm for RSPQ evaluation on static graphs [54].

Definition 17 (Prefix Paths). *Given a node* $(u, s) \in T_x$*, we say that the path from the root to* (u, s) *is the prefix path* p *for node* (u, s)*. We use the notation* $p[v], v \in V$ *to denote the set of states that are visited in vertex* v *in path* p*, i.e.,* $p[v] = \{s \in S \mid (v, s) \in p\}$*.*

Algorithm RSPQ:

input : Incoming tuple $t_\tau = (\tau, e_\tau, l, op), e_\tau = (u, v)$
1 $G_{W,\tau} \leftarrow G_{W,\tau-1} + e_\tau$
2 ExpiryRSPQ($G_{W,\tau}, T_x, \tau$) $\forall T_x \in \Delta$ // with β intervals
3 set of results $R \leftarrow \emptyset$
4 **foreach** $T_x \in \Delta$ **do**
5 **foreach** $s, t \in S$ *where* $t = \delta(s, l)$ **do**
6 **if** $(u, s) \in T_x \wedge (u, s).ts > \tau - |W|$ **then**
7 $p \leftarrow PATH(T_x, (u, s))$ // the prefix path
8 **if** $t \notin p[v] \wedge (v, t) \notin M_x$ **then**
9 $R \leftarrow R +$ Extend($T_x, p, (v, t), e_\tau$)
10 $Q_R(S, W, \tau) \leftarrow Q_R(S, W, \tau - 1) + R$

Algorithm Extend:

input : Spanning Tree T_x, Prefix Path p, Node (v, t), Edge $e = (u, v)$
output: Set of results R
1 $R \leftarrow \emptyset$
2 **if** $q = FIRST(p[v])$ *and* $[q] \not\supseteq [t]$ **then**
3 Unmark(T_x, p) // q and t have a conflict at v
4 **else**
5 **if** $t \in F$ **then** $R \leftarrow R + (x, v)$;
6 **if** $(v, t) \notin T_x$ **then** $M_x \leftarrow M_x \bigcup (v, t)$;
7 add (v, t) as (u, s)'s child in T_x
8 $p_{new} \leftarrow p + [v, t]$
9 $p_{new}.ts = min(e.ts, p.ts)$
10 **foreach** *edge* $e = (v, w) \in W_{G,\tau}$ *s.t.* $\delta(t, \phi(e)) = r$ **do**
11 **if** $r \notin p_{new}[w] \wedge (w, r) \notin M_x$ **then**
12 $R \leftarrow R +$ Extend($T_x, p_{new}, (w, r), e$)
13 **return** R

Algorithm Unmark:

input : Spanning Tree T_x, Prefix Path p
1 $Q \leftarrow \emptyset$
2 **while** $p \neq \emptyset \wedge (v, t) = LAST(p) \wedge (v, t) \in M_x$ **do**
3 $M_x \leftarrow M_X \setminus (v, t)$
4 $Q \leftarrow Q + (v, t)$
5 $p \leftarrow PATH(T_x, (v, t).parent$
6 **foreach** $(v, t) \in Q$ **do**
7 **foreach** *edge* $e = (w, v) \in G_{W,\tau}$ *s.t.* $t = \delta(q, \phi(e))$ **do**
8 **if** $(w, q) \in T_x \wedge t \notin p[v]$ **then**
9 $p_{candidate} \leftarrow PATH(T_x, (w, q))$
10 Extend($T_x, p_{candidate}, (v, t), e$)

Definition 18 (Conflict Predecessor). *A node $(u, s) \in T_x$ is a conflict predecessor if for some successor (w, t) of (u, s) in T_x, (w, q) is the first occurrence of vertex w in the prefix path of (u, s) and there is a conflict between q and t at w, i.e., $[q] \not\supseteq [t]$.*

In addition to tree index Δ of Algorithm **RAPQ** in §3, Algorithm **RSPQ** maintains a set of markings M_x for each spanning tree T_x. The set of markings M_x for a spanning tree

Algorithm ExpiryRSPQ:

input : Window $G_{W,\tau}$, timestamp τ, Spanning Tree T_x
output: The set of invalidated results R_I
1 $R_I \leftarrow \emptyset$
2 $E = \{(v, t) \in T_x \mid (v, t).ts \leq \tau - |W|\}$ // expired nodes
3 $P \leftarrow M_x \cap E$
4 $T_x \leftarrow T_x \setminus E$ // prune T_x
5 $M_x \leftarrow M_x \setminus E$ // prune M_x
6 **foreach** $(v, t) \in P$ **do**
7 **foreach** $(u, v) \in W_{G,\tau}$ *s.t.* $(u, s) \in T_x \wedge t = \delta(s, \phi(u, v))$ **do**
8 $p \leftarrow PATH(T_x, (u, s))$
9 $P \leftarrow P \setminus$ Extend($T_x, p, (v, t), (u, v)$)
10 **foreach** $(w, q) \in P$ **do**
11 **if** *all siblings of* (w, q) *are in* M_x **then**
12 $M_x \leftarrow M_x + (w, q).parent$
13 **if** $q \in F$ **then** $R_I \leftarrow R_I + (x, w)$;
14 **return** R_I

T_x is the set of nodes in T_x with no descendants that are conflict predecessors (Definition 18). In the absence of conflicts, there is no conflict predecessor and M_x contains all nodes in T_x. Algorithm **RSPQ** does not visit a node in M_x (Lines 8 in Algorithm **RSPQ** and 11 in Algorithm **Extend**). Hence, a node (u, s) appears in the spanning tree T_x at most once in the absence of conflicts. Consequently, Algorithm **RSPQ** maintains the second invariant of Δ and behaves similar to the Algorithm **RAPQ** (§3.1). On static graphs, the batch algorithm adds a node (u, s) to the set of markings only after the entire depth-first traversal of the product graph from (u, s) is completed, ensuring that M_x is monotonically growing. But, tuples that arrive later in the streaming graph S might lead to a conflict with a node (u, s) that is already in M_x As described later, Algorithm **RSPQ** correctly identifies these conflicts and updates the spanning tree T_x and removes (u, s)'s ancestors from M_x to ensure correctness. The conflict detection mechanism signals that the corresponding traversal cannot be pruned even if it visits a previously visited vertex. In other words, a node $(u, s) \notin M_x$ may be visited more than once in a spanning tree T_x to ensure correctness. Consequently, Algorithm **RSPQ** traverses every simple path that satisfies the given query Q_R if every node in T_x is a conflict predecessor ($M_x = \emptyset$), leading to exponential time execution in the worst case. In summary, Algorithm **RSPQ** differs from Algorithm **RAPQ** in two major points: (i) it may traverse a vertex in the same state more than once if a conflict is discovered at the vertex, and (ii) it keeps track of conflicts and maintains a set of markings to prevent multiple visits of the same vertex in the same state whenever possible.

For each incoming tuple $t_\tau(ts, e, l, +), e = (u, v)$, Algorithm **RSPQ** finds prefix paths of all $(u, s) \in T_x$ (Line 7)

(there exists a single such node (u, s) and its corresponding prefix path if $(u, s) \in M_x$). Then it performs one of the following four steps for each node $(u, s) \in T_x$ and its corresponding prefix path p:

(1) $t \in p[v]$: Vertex v is visited in the same state t as before, thus path p is pruned as extending it with (v, t) leads to a cycle in the product graph $P_{G,A}$ (Line 8 in **RSPQ** and Line 11 **Extend**).
(2) $(v, t) \in M_x$: The target node (v, t) has already been visited in T_x and it has no conflict predecessor descendant. Thus, path p is pruned (Line 8 in **RSPQ**, 11 in **Extend**).
(3) $q = FIRST(p[v])$ and $[q] \not\supseteq [t]$: States q and t have a conflict at vertex v (Line 2 in **Extend**), making (u, s) a conflict predecessor. Therefore, all ancestors of (u, s) in T_x are removed from M_x (Algorithm **Unmark**). During unmarking of a node $(v_i, s_i) \in M_x$, all $(w, q) \in T_x$ where $(w, v_i) \in G_{W,\tau}$ and $s_i = \delta(q, \phi(w, v_i))$ are considered as candidate for traversal as they were previously pruned due to (v_i, s_i) being marked.
(4) Otherwise path p is extended with (v, t), i.e., (v, t) is added as a child to (u, s) in T_x. (Line 4 in **Extend**)

As described previously, an important difference between Algorithm **RSPQ** and the batch algorithm is that the streaming version may remove nodes from M_x whereas a node stays in M_x once added in the batch model. Hence, the batch algorithm can safely prune a path p if it vists a node $(u, s) \in M_x$ as the containment property ensures correctness. Whereas, the streaming model requires a special treatment as M_x is not monotonically growing. Case 2 above prunes a path p if it reaches a node $(u, s) \in M_x$. Unlike the batch algorithm, a node (u, s) may be removed from M_x due to a conflict that is caused by an edge that later arrives. This conflict implies that path p should not have been pruned. Case 3 above (Algorithm **Unmark**) addresses exactly this scenario: ancestors of a conflict predecessor are removed from M_x.

Whenever a node (u, s) is removed from M_x due to a conflict at one of its descendants, Algorithm **Unmark** finds all paths that are previously pruned due to (u, s) by traversing incoming edges of $(u, s) \in G_{W,\tau}$ and invokes Algorithm **Extend** for each such path. Algorithm **Extend** backtracks and evaluates all paths that would not be pruned by Case 2 if (u, s) were not in M_x, ensuring correctness. The following example illustrates this behavior of Algorithm **RSPQ**.

Example 7. Consider the streaming graph and the query in Figure 1 and its spanning tree given in Figure 2(a), and assume for now that Algorithm **RSPQ** does not detect conflicts and only traverses simple paths in $G_{W,\tau}$. After processing edge (x, y) at time $t = 13$, it adds $(u, 2)$ as a child of $(y, 1)$. Edge (z, u) arrives at $t = 14$, but $(u, 2)$ is not added as $(z, 1)$'s child as $(u, 2)$ already exists in T_x. Later at $t = 18$, edge (v, y) arrives, but $(y, 2)$ is not added to the spanning tree T_x as the path $\langle x, y, u, v, y\rangle$ forms a cycle in $G_{W,\tau}$. As a result, $(y, 2)$ is never visited and (x, y) is never reported although there exists a simple path in $G_{W,\tau}$ from x to y, that is $\langle x, z, u, v, y\rangle$.

Instead, Algorithm **RSPQ** detects the conflict at the vertex v between states 1 and 2 after edge (v, y) arrives at time $t = 18$ as $FIRST(p[y]) = 1$ and $[1] \not\supseteq [2]$. Algorithm **Unmark** removes all ancestors of $(y, 2)$ from M_x and, during unmarking of $(u, 2)$, the prefix path p from $(x, 0)$ to $(z, 1)$ is extended with $(u, 2)$. Finally, Algorithm **Extend** traverses the simple path $\langle x, z, u, v, y\rangle$ and adds (x, y) to the result set. Figure 2(c) depicts the spanning tree $T_x \in \Delta$ at time $t = 18$.

Similar to its arbitrary counterpart, Algorithm **ExpiryRSPQ** is invoked at pre-defined slide intervals β. It first identifies the set of candidate nodes whose timestamps are not in $(\tau - |W|, \tau]$ (Line 2). Unmarked candidate nodes $(M_x \setminus E)$ can safely be removed from T_x as the unmarking procedure considers all valid edges to an unmarked node. Hence, Algorithm **ExpiryRSPQ** reconnects a candidate node with a valid edge only if it is in M_x (Line 6). Finally, it updates M_x with nodes that are not conflict predecessors any longer (Line 10).

Theorem 8. *Algorithm **RSPQ** is correct and complete.*

Proof Sketch: The proof is similar to the proof of Theorem 3. The key argument is that, if the pair (x, u) is added to the result set after traversing a non-simple path p (in which a vertex v appears more than once), there must be an alternative simple path p' in $G_{W,\tau}$ from x to u. The proof constructs an inductive argument on the number of appearances of vertex v in p, and use the suffix language containment property (Definition 15) to show the existence of such p' from (x, s_0) to (u, s_f) for some $s_f \in F$. □

Theorem 9. *The amortized cost of Algorithm **RSPQ** is $O(n \cdot k^2)$, where n is the number of distinct vertices in the window W and k is the number of states in the corresponding automaton A of the query Q_R.*

Proof Sketch: The essential point of the proof is that, in the absence of conflicts, all nodes in a given spanning tree T_x are also in M_x as there is no conflict predecessor in T_x. Case 2 above guarantees that a vertex v is visited at state t if $(v, t) \notin M_x$ (Line 8 in **RSPQ** and Line 11 **Extend**), hence, the amortized time complexity of the proposed algorithm is the same as Algorithm **RAPQ** (Theorem 4 in §3.1). □

Consequently, the amortized cost of Algorithm **RSPQ** is linear in the number of vertices in the snapshot graph $G_{W,\tau}$ as its RAPQ counterpart (§ 3.1). Explicit deletions are handled in the same manner as its RAPQ counterpart (§3.2). Its amortized cost over a sequence of m explicit deletions is is $O(n^2 \cdot k)$, in the absence of conflicts, where n is the number of distinct vertices and k is the number of states in the corresponding DFA of a RSPQ Q_R.

5 EXPERIMENTAL ANALYSIS

We study the feasibility of the proposed persistent RPQ evaluation algorithms on both real-world and synthetic streaming graphs. We first systematically evaluate the throughput and latency of Algorithm **RAPQ** on append-only streaming graphs (§5.2.1). Then, we assess its scalability by varying the window size $|W|$, the slide interval β and the query size $|Q_R|$ (§5.2.2). The overhead of Algorithm **Delete** over Algorithm **RAPQ** for explicit deletions is analyzed in §5.2.3 whereas §5.2.4 analyzes the feasibility of **RSPQ** for persistent RPQ evaluation under simple path semantics.

The highlights of our results are as follows:

(1) The proposed persistent RPQ evaluation algorithms maintain sub-millisecond edge processing latency on real-world workloads, and can process up-to tens of thousands of edges-per-second on a single machine.
(2) The tail (99th percentile) latency of the algorithms increases linearly with the window size $|W|$, confirming the amortized costs in Table 1.
(3) The cost of expiring old tuples grows linearly with the slide interval β, which enables constant overhead regardless of β when amortized over the slide interval.
(4) Explicit deletions can incur up to 50% performance degradation on tail latency, however the impact stays relatively steady with the increasing ratio of deletions.
(5) Although RPQ evaluation under simple path semantics is NP-hard in the worst-case, the results indicate that the majority of the queries formulated on real-world and synthetic streaming graphs can be evaluated with 2× to 5× overhead on the tail latency.

5.1 Experimental Setup

Implementation: The prototype system is an in-memory implementation in Java 13 and includes algorithms in §3 and §4 — we leave out-of-core processing as future work. The tree index Δ is implemented as a concurrent hash-based index where each vertex $v \in G_{W,\tau}$ is mapped to its corresponding spanning tree T_x. Each spanning tree T_x is assisted with an additional hash-based index for efficient node lookups. RAPQ (**RAPQ** and **ExpiryRAPQ**), RSPQ algorithms (**RSPQ**, and **ExpiryRAPQ**) employ *intra-query parallelism* by deploying a thread pool to process multiple spanning trees in parallel that are accessed for each incoming edge. Window management is parallelized similarly.

Experiments are run on a Linux server with 32 physical cores and 256GB memory with the number of execution threads set to the number of physical cores. We report the average tuple processing throughput and the tail latency (99^{th} percentile) after ten minutes of processing on warm caches. Our prototype implementation is a closed system, i.e., each arriving tuple t_τ is processed sequentially. Thus, the throughput is inversely correlated with the mean latency.

Table 2: Most common RPQs in real workloads.

Name	Query	Name	Query
Q_1	a^*	Q_7	$a \circ b \circ c^*$
Q_2	$a \circ b^*$	Q_8	$a? \circ b^*$
Q_3	$a \circ b^* \circ c^*$	Q_9	$(a_1 + a_2 + \cdots + a_k)^+$
Q_4	$(a_1 + a_2 + \cdots + a_k)^*$	Q_{10}	$(a_1 + a_2 + \cdots + a_k) \circ b^*$
Q_5	$a \circ b^* \circ c$	Q_{11}	$a_1 \circ a_2 \circ \cdots \circ a_k$
Q_6	$a^* \circ b^*$		

Workloads and Datasets: Although there exists streaming RDF benchmarks such as LSBench [1] and Stream WatDiv [32], their workloads do not contain any recursive queries, and they generate streaming graphs with very limited form of recursion. Therefore, we formulate persistent RPQs using the most common recursive queries found in real-world applications, leveraging recent studies [18, 19] that analyze real-world SPARQL query logs. We choose the most common 10 recursive queries from [19], which cover more than 99% of all recursive queries found in Wikidata query logs (Table 2). In addition, we use the most common non-recursive query (with no Kleene stars) for completeness. We set $k = 3$ for queries with variable number of edge labels as the SO graph only has three distinct labels. We run these over the following real and synthetic edge-labeled graphs.

Stackoverflow (SO) is a temporal graph of user interactions on this website containing 63M interactions (edges) of 2.2M users (vertices), spanning 8 years [58]. Each directed edge (u, v) with timestamp t denotes an interaction: (i) user u answered user v's questions at time t, (ii) u commented on v's question, or (iii) comment at time t. SO is more homogeneous and more cyclic than other graphs we use as it contains only a single type of vertex and 3 different edge labels. 7 out of 11 queries in Table 2 have at least 3 labels and cover all edges in the graph. Its highly dense and cyclic nature causes a high number of intermediate results and resulting paths; so it is the most challenging one for the proposed algorithms. We set the window size $|W|$ to 1 month and the slide interval β to 1 day unless specified otherwise.

LDBC SNB is synthetic social network graph that is designed to simulate real-world interactions in an online social network [27]. We extract the update stream of the LDBC workload, which exhibits 8 different types of interactions. The streaming graph generated by LDBC consists of two recursive relations: *knows* and *replyOf*. Thus, Q_4, Q_5, Q_9 and Q_{10} in Table 2 cannot be meaningfully formulated over them. We use a scale factor of 10 with approximately 7.2M users and posts (vertices) and 40M user interactions (edges). LDBC update stream spans 3.5 months of user activity and we set $|W| = 10$ days and $\beta = 1$ day unless specified otherwise.

Yago2s is a real-world RDF dataset containing 220M triples (edges) with approximately 72M different subjects (vertices) [2]. Yago2s includes a rich schema (~100 different labels) and

allows the representation of all the queries in Table 2. To emulate sliding windows on Yago2s RDF graph, we assign a monotonically non-decreasing timestamp to each RDF triple at a fixed rate. Thus, each window defined over Yago2s has equal number of edges (≈10M) and slide every 1M edges.

We use **gMark** [12] graph and query workload generator to assess the impact of query size $|Q_R|$ defined as the sum of the number of labels in the regular expression R and the number of occurrences of $*$ and $+$. We use a preconfigured schema that mimics the characteristics of LDBC SNB graph to generate a synthetic graph with 100M vertices and 220M edges, and create synthetic RPQ workloads where the query size ranges from 2 to 20. Each RPQ is formulated by grouping labels into concatenations and alternations of size up to 3, and each group has a 50% probability of having $*$ and $+$. As gMark generates a static graph, we assign a monotonically non-decreasing timestamp to each edge at a fixed rate.

5.2 Experimental Results

5.2.1 ***Throughput & Tail Latency:*** Figure 3 shows the throughput and tail latency of Algorithm **RAPQ** for all queries on all datasets. The algorithm discards a tuple whose label is not in the alphabet Σ_Q of Q_R as it cannot be part of any resulting path. Hence, we only measure and report latency of tuples whose labels match a label in the given query. The performance is generally lower for the SO graph due to its label density and highly cyclic nature. The tail latency of Algorithm **RAPQ** is below 100ms even for the slowest query Q_3 on SO and it is in sub-milliseconds for most queries on Yago2s and LDBC. Similarly, the throughput of the algorithm varies from hundreds of edges-per-second for the SO (Figure 3(c)) to tens of thousands of edges-per-second for LDBC (Figure 3(b)).

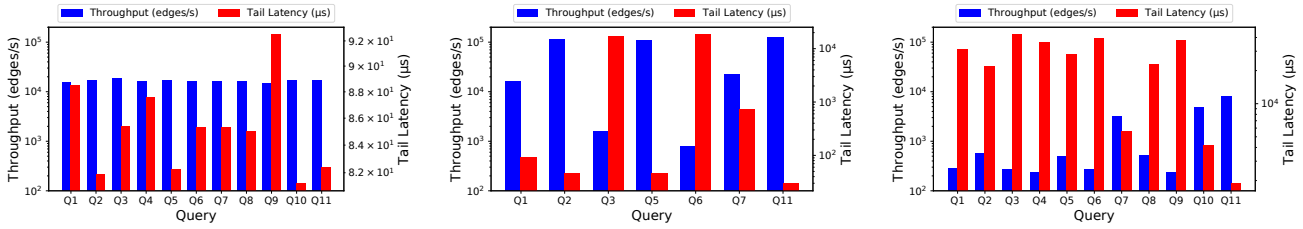
We plot the total number of trees and nodes in the tree index Δ (Definition 12) on SO to better understand diverse performance characteristics of different queries. Recall that nodes in a spanning tree $T_x \in \Delta$ represent partial traversals during query execution. Hence, the amount of work performed by the algorithm grows with the size of tree index Δ. As expected, we observe a negative correlation between the throughput of a query (Figure 3(c)) and its tree index size (Figure 4). It is known that cycles have significant impact on the run time of queries [18], and our results confirm this. In particular, Q_3 and Q_6 have the largest index sizes and therefore the lowest throughput, which can be explained by the presence of multiple Kleene stars. Similarly, Q_4 and Q_9 have a Kleene star over alternation of symbols, which covers all the edges in the graph as SO has only 3 types of user interactions. Thus, Q_4 and Q_9 both have large index sizes, which negatively impacts the performance. Q_{11} has the highest throughput on all datasets as it is the only fixed size, non-recursive query tested in our experiments.

Ours is the first work that investigates the execution of persistent RPQs over streaming graphs. As such, there are no existing systems we can directly compare with. In order to build a baseline, we emulate persistent query execution over sliding windows into one of the existing RDF systems with SPARQL property path support (e.g., Virtuoso [28], RDF-3X [38, 39]). We emulate persistent queries over Virtuoso in order to highlight the benefits of using incremental algorithms for persistent query evaluation over streaming graphs. The obtained experimental results (omitted for space constraints and reported in [56]) show that our algorithms provide far superior throughput and tail latency. Furthermore, they suggest that incremental evaluation as in the proposed algorithms have non-negligible performance advantages in executing RPQs over streaming graphs.

5.2.2 ***Scalability & Sensitivity Analysis:*** We first assess the impact of the window size $|W|$ and the slide interval β, then consider performance implications of the use of DFAs and the query size $|Q_R|$.

We use the Yago2s dataset for this experiment as windows with a fixed number of edges we created over Yago2s enable us to precisely assess the impact of window size. Figure 5(a) presents the tail latency of our algorithm, where the window size changes from 5M edges to 20M edges with 5M intervals. As expected, the tail latency for all tested queries increases with $|W|$, conforming the amortized cost analysis in §3.1. Similarly, the time spent on Algorithm **ExpiryRAPQ** increases with window size $|W|$ (Figure 5(b)), in line with the complexity analysis in §3.1. We replicate the same experiment using LDBC and Stream WatDiv datasets by varying the scale factor, which in turn increases the number of edges in each window. We observe a degradation on the performance with increasing scale factor on Stream WatDiv, confirming our findings on Yago2s. But, this is not observed on LDBC, which is due to the linear scaling of the total number of edges and vertices with the scale factor. Increasing the scale factor reduces the density of the graph, causing the algorithms to perform even better in some instances due to a smaller tree index size. Detailed results on these graphs are omitted due to space constraints.

Next, we assess the impact of the slide interval β on algorithm performance. Figure 5(a) plots the tail latency of Algorithm **RAPQ** against β and shows that the slide interval does not impact performance. Recall that Algorithm **ExpiryRAPQ** is invoked periodically to remove expired tuples from the tree index Δ. As described in §3.1, Algorithm **ExpiryRAPQ** might traverse the entire snapshot graph $G_{W,\tau}$ in the worst-case, regardless of the slide interval β. Yet, Figure 5(b) shows that the time spent on tuple expiry grows with increasing β, which causes its overhead to stay constant over time regardless of β. Therefore, this algorithm is



(a) Yago2s (b) LDBC SF10 (c) Stackoverflow

Figure 3: Throughput and tail latency of the Algorithm RAPQ. Y axis is given in log-scale.

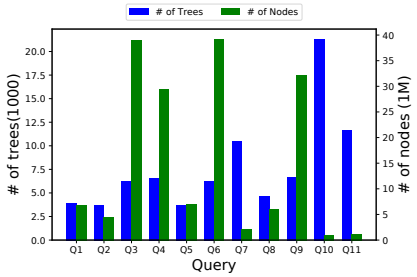


Figure 4: Size of the tree index Δ on the SO graph.

robust w.r.t. β. It also suggests that the complexity analysis of Algorithm **ExpiryRAPQ** given in §3.1 is not tight.

Finally, we analyze the effect of the query size $|Q_R|$ and the automata size k on the performance of our algorithms using a set of 100 synthetic RPQs that are generated using gMark. Combined complexities of the algorithms presented in §3 and §4 are polynomial in the number of states k, which might be exponential in the query size $|Q_R|$. Figure 6 (left) shows the total number of states in minimized DFAs for 100 RPQs we created using gMark; in practice, we do not observe this exponential growth for the considered RPQs despite the theoretical upper bound. Green et al. [36] have also indicated that exponential DFA growth is of little concern for most practical applications in the context of XML streams.

Next, we focus on the impact of the automata size k on performance. Figure 6 (right) plots the throughput against the number of states k in the minimal automata for synthetic RPQs generated by gMark. No significant impact of k on performance is observed; yet, performance differences for queries with the same number of states in their corresponding DFA can be up to 6×. Such trends for RPQ evaluation has already been observed on static graphs and has been attributed to query label selectivities and the size of intermediate results [70]. To further verify this hypothesis in the streaming model, we focus on throughput and the tree index Δ size for queries with $k = 5$. We observe a negative correlation between query throughput and tree index size, confirming our results in § 5.2.1. These results are in the long version of our paper due to space constraints (Figure 10 in [56]).

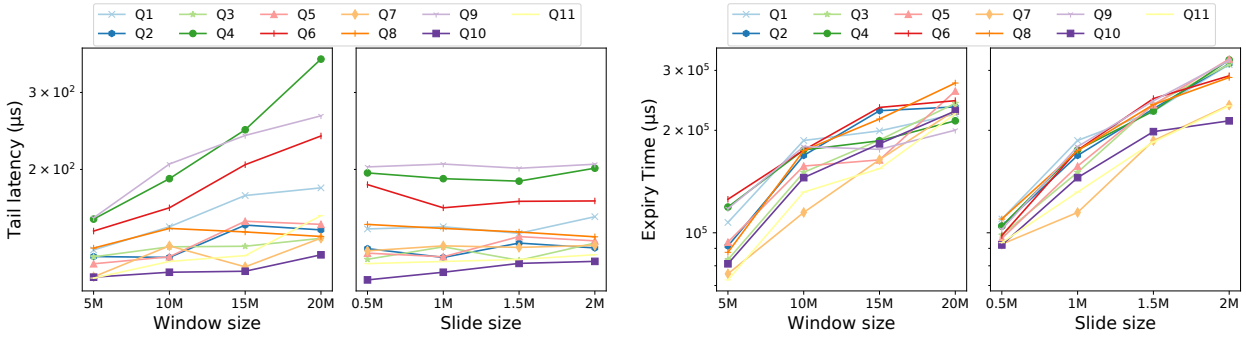
Table 3: Relative slowdown of queries with RSPQ.

Graph	Succesfull Queries	Latency Overhead
Yago2s	All	1.8 × −2.1×
Stackoverflow	$Q_1, Q_4, Q_7, Q_{10}, Q_{11}$	1.4 × −5.4×
LDBC SF10	$Q_1, Q_2, Q_5, Q_7, Q_{11}$	1.8 × −3×

5.2.3 ***Explicit Edge Deletions:*** Although most real-life streaming graphs are append-only, some applications require explicit edge deletions, which can be processed in our framework (§3.2). We generate explicit deletions by reinserting a previously consumed edge as a negative tuple and varying the ratio of negative tuples in the stream. Figure 7 plots tail latency of all queries on Yago2s varying deletion ratio from 2% to 10%. In line with our findings in the previous section, explicit deletions incur performance degradation due to the overhead of the expiry procedure (Figure 5(b)). However, this overhead quickly flattens and does not increase with the deletion ratio. This is explained by the fact that the sizes of the snapshot graph $G_{W,\tau}$, and the tree index Δ decrease with increasing deletion ratio.

5.2.4 ***RPQ under Simple Path Semantics:*** We showed (§4) that the amortized time complexity of Algorithm **RSPQ** under simple path semantics is the same as its RAPQ counterpart in the absence of conflicts. We now empirically analyze the feasibility and the performance of this algorithm. Table 3 lists the queries that can be successfully evaluated under simple path semantics on each graph. Q_1, Q_4 and Q_{11} are restricted regular expressions, a condition that implies conflict-freedom in any arbitrary graph. Consequently, these queries are successfully evaluated on all graphs we tested (except LDBC that cannot execute Q_4 as discussed in §5.1). In particular, all queries are free of conflicts on Yago2s, and they can be successfully evaluated.

Table 3 also reports the overhead of enforcing simple path semantics on the tail latency. This overhead is simply due to conflict detection and the maintenance of markings for each spanning tree in the tree index Δ. Overall, these results suggest the feasibility of enforcing simple path semantics for majority of real-world queries, considering that most



(a) Tail Latency (b) Window Management Time

Figure 5: The tail latency (a) and the average window maintenance cost (b) with various $|W|$ and β.

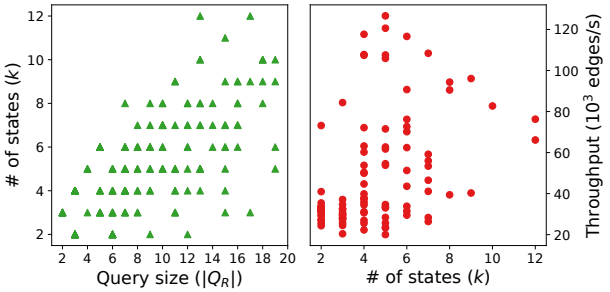


Figure 6: The impact of the query length $|Q_R|$ on the automata size, k, and the throughput.

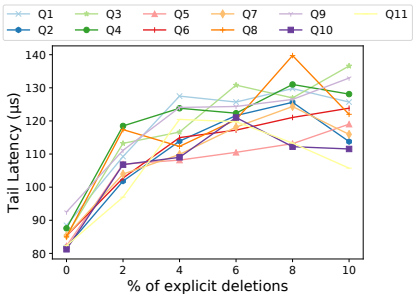


Figure 7: Impact of the ratio of explicit deletions on tail latency for all queries on Yago2s RDF graph.

queries are conflict-free on heterogeneous, sparse graphs such as RDF graphs and social networks. Conversely, we argue that arbitrary path semantics may be the only practical alternative for applications with homogeneous, highly cyclic graphs such as communication networks like Stackoverflow.

6 RELATED WORK

Stream Processing Systems: Early research on stream processing primarily adopt the relational model and its query operators in the streaming settings (STREAM [9], Aurora [4], Borealis [3]). Whereas, modern Data Stream Processing Systems (DSPS) such as Storm [67], Heron [46], Flink [23] are mostly scale-out solutions that do not necessarily offer a full set of DBMS functionality. Existing literature (as surveyed by Hirzel et al. [40]) heavily focus on general-purpose systems and do not consider core graph querying functionality such as *subgraph pattern matching* and *path navigation*.

There has been a significant amount of work on various aspects of RDF stream processing[6]. Calbimonte [20] designs a communication interface for streaming RDF systems based on the Linked Data Notification protocol. TripleWave [52] focuses on the problem of RDF stream deployment and introduces a framework for publishing RDF streams on the web. EP-SPARQL [8] extends SPARQLv1.0 for reasoning and a complex event pattern matching on RDF streams. Similarly, SparkWave [44] is designed for streaming reasoning with schema-enhanced graph pattern matching and relies on the existence of RDF schemas to compute entailments. None of these are processing engines, so they do not provide query processing capabilities. Most similar to ours are streaming RDF systems with various SPARQL extensions for persistent query evaluation over RDF streams such as C-SPARQL [14], CQELS [48], SPARQL$_{stream}$ [21] and W3C proposal RSP-QL [26]. However, these systems are designed for SPARQLv1.0, and they do not have the notion of *property paths* from SPARQLv1.1. Thus one cannot formulate path expressions such as RPQs that cover more than 99% of all recursive queries abundantly found in massive Wikidata query logs [19]. The lack of property path support of these systems is previously reported by an independent RDF streaming benchmark, SR-Bench [72] (see Table 3 in [72]). Furthermore, query processing engines of these systems do not employ incremental operators, except Sparkwave [44] that focuses on stream reasoning. On the contrary, our proposed algorithms incrementally maintain results for a persistent query Q_R as the graph edges arrive. Our contributions are orthogonal to existing work on streaming RDF systems, although the

[6]https://www.w3.org/community/rsp/wiki/Main_Page

algorithms proposed in this paper can be integrated into these systems as they incorporate SPARQLv1.1 (i.e., property paths) to provide native RPQ support.

Streaming & Dynamic Graph Theory: Earlier work on streaming graph algorithms is motivated by the limitations of main memory, and existing literature has widely adopted the *semi-streaming* model for graphs where the set of vertices can be stored in memory but not the set of edges [55], due to infeasibility of graph problems in sublinear space. There exist a plethora of approximation algorithms in this model, and we refer interested readers to [53] for a survey.

Graph problems are widely studied in the dynamic graph model where algorithms may use the necessary memory to store the entire graph and compute how the output changes as the graph is updated. Examples include connectivity [42], shortest path [15], transitive closure [47]. Most related to ours is dynamic reachability, which can be used to solve RPQ under arbitrary path semantics given the entire product graph (Definition 11). The state-of-the-art dynamic reachability algorithm has $O(m + n)$ amortized update time [61]. Our proposed algorithms have a lower amortized cost, $O(n)$, for insertions at the expense of $O(n^2)$ amortized time for deletions – a trade-off justified by the insert-heavy nature of real-world streaming graphs. Fan et al. [29] characterize the complexity of various graph problems, including RPQ evaluation, in the dynamic model and show that most graph problems are unbounded under edge updates, i.e., the cost of computing changes to query answers cannot be expressed as a polynomial of the size of the changes in the input and output. They prove that RPQ is bounded relative to its batch counterpart; the batch algorithm can be efficiently incrementalized by minimizing unnecessary computation.

Regular Path Queries: The research on RPQs focuses on various problems such as containment [22], enumeration [51], learnability [16]. Most related to ours is the RPQ evaluation problem. The seminal work of Mendelzon and Wood [54] shows that RPQ evaluation under simple path semantics is NP-hard for arbitrary graphs and queries. They identify the conditions for graphs and regular languages where the introduce a maximal class of regular languages, C_{tract}, for which the problem of RPQ evaluation under simple path semantics is tractable.

RPQ evaluation strategies follow two main approaches: automata-based and relational algebra-based. G [25], one of the earliest graph query languages, builds a finite automaton from a given RPQ to guide the traversal on the graph. Kochut et al. [43] study RPQ evaluation in the context of SPARQL and propose an algorithm that uses two automatons, one for the original expression and one for the reversed expression, to guide a bidirectional BFS on the graph. Addressing the memory overhead of BFS traversals, Koschmieder et al. [45] decompose a query into smaller fragments based on rare labels and perform a series of bidirectional searches to answer individual subqueries. A recent work by Wadhwa et al. [69] uses random walk-based sampling for approximate RPQ evaluation. The other alternative for RPQ evaluation is α-RA that extends the standard relational algebra with the α operator for transitive closure computation [5]. α-RA-based RPQ evaluation strategies are used in various SPARQL engines [28]. Histogram-based path indexes on top of a relational engine can speed-up processing RPQs with bounded length [31]. α-RA-based RPQ evaluation is not suitable for persistent RPQ evaluation on streaming graphs as it relies on blocking join and α operators. Hence, we adapt the automata-based RPQ evaluation in this paper and introduce non-blocking, incremental algorithms for persistent RPQ evaluation. Besides, Yakovets et al. [70] show that these two approaches are incomparable and they can be combined to explore a larger plan space for SPARQL evaluation. Various formalisms such as pebble automata, register automata, monadic second-order logic with data comparisons extend RPQs with data values for the property graph model [49, 50]. Although RPQs and corresponding evaluation methods are widely used in graph querying [6, 7, 28], all of these works focus on static graphs; ours is, to the best of our knowledge, the first work to consider persistent RPQ evaluation on streaming graphs.

7 CONCLUSION AND FUTURE WORK

In this paper, for the first time, we study the problem of efficient persistent RPQ evaluation on sliding windows over streaming graphs.The proposed algorithms process explicit edge deletions under both arbitrary and simple path semantics in a uniform manner. In particular, the algorithm for simple path semantics has the same complexity as the algorithm for arbitrary path semantics in the absence of conflicts, and it admits efficient solutions under the same condition as the batch algorithm. Experimental analyses using a variety of real-world RPQs and streaming graphs show that proposed algorithms can support up to tens of thousands of edges-per-second while maintaining sub-second tail latency. Future research directions we consider in this project are: (i) to extend our algorithms with attribute-based predicates to fully support the popular property graph data model, and (ii) to investigate multi-query optimization techniques to share computation across multiple persistent RPQs.

ACKNOWLEDGMENTS

This research was partially supported by grants from Natural Sciences and Engineering Research Council (NSERC) of Canada and Waterloo-Huawei Joint Innovation Lab. This research started during Angela Bonifati's sabbatical leave (supported by INRIA) at the University of Waterloo in 2019.

REFERENCES

[1] 2012. LSBench Code. https://code.google.com/archive/p/lsbench/

[2] 2018. Yago: A High-Quality Knowledge Base. https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/

[3] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. 2005. The Design of the Borealis Stream Processing Engine. In *Proc. 2nd Biennial Conf. on Innovative Data Systems Research*. 277–289.

[4] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: a new model and architecture for data stream management. *VLDB J.* 12, 2 (2003), 120–139.

[5] Rakesh Agrawal. 1988. Alpha: An extension of relational algebra to express a class of recursive queries. *IEEE Trans. Softw. Eng.* 14, 7 (1988), 879–885.

[6] Renzo Angles, Marcelo Arenas, Pablo Barcelo, 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 *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 1421–1432.

[7] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of modern query languages for graph databases. *ACM Comput. Surv.* 50, 5 (2017), 68.

[8] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. 2011. EP-SPARQL: a unified language for event processing and stream reasoning. In *Proc. 20th Int. World Wide Web Conf.* 635–644.

[9] A. Arasu, S. Babu, and J. Widom. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB J.* 15, 2 (2006), 121–142.

[10] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. 2002. Models and Issues in Data Stream Systems. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*. 1–16.

[11] Pablo Barceló Baeza. 2013. Querying graph databases. In *Proc. 32nd ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*. 175–188.

[12] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George HL Fletcher, Aurélien Lemay, and Nicky Advokaat. 2016. gMark: schema-driven generation of graphs and queries. *IEEE Trans. Knowl. and Data Eng.* 29, 4 (2016), 856–869.

[13] Guillaume Bagan, Angela Bonifati, and Benoît Groz. 2013. A trichotomy for regular simple path queries on graphs. In *Proc. 32nd ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*. 261–272.

[14] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. 2009. C-SPARQL: SPARQL for continuous querying. In *Proc. 18th Int. World Wide Web Conf.* 1061–1062.

[15] Aaron Bernstein. 2016. Maintaining shortest paths under deletions in weighted directed graphs. *SIAM J. on Comput.* 45, 2 (2016), 548–574.

[16] Angela Bonifati, Radu Ciucanu, and Aurélien Lemay. 2015. Learning Path Queries on Graph Databases. In *Proc. 18th Int. Conf. on Extending Database Technology*. Bruxelles, Belgium, 109–120. https://doi.org/10.5441/002/edbt.2015.11

[17] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. Querying Graphs. *Synthesis Lectures on Data Management* 10, 3 (2018), 1–184.

[18] Angela Bonifati, Wim Martens, and Thomas Timm. 2017. An analytical study of large SPARQL query logs. *Proc. VLDB Endowment* 11, 2 (2017), 149–161.

[19] Angela Bonifati, Wim Martens, and Thomas Timm. 2019. Navigating the Maze of Wikidata Query Logs. In *Proc. 28th Int. World Wide Web Conf.* 127–138.

[20] Jean-Paul Calbimonte. 2017. Linked Data Notifications for RDF Streams.. In *WSP/WOMoCoE@ ISWC*. 66–73.

[21] Jean-Paul Calbimonte, Oscar Corcho, and Alasdair JG Gray. 2010. Enabling ontology-based access to streaming data sources. In *Proc. 9th Int. Semantic Web Conf.* 96–111.

[22] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi. 2000. Query processing using views for regular path queries with inverse. In *Proc. 19th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*. 58–66.

[23] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *Q. Bull. IEEE TC on Data Eng.* 38, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf

[24] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and Distance Queries via 2-Hop Labels. *SIAM J. on Comput.* 32, 5 (2003), 1338.

[25] Isabel F Cruz, Alberto O Mendelzon, and Peter T Wood. 1987. A graphical query language supporting recursion. In *ACM SIGMOD Rec.*, Vol. 16. 323–330.

[26] Daniele Dell'Aglio, Jean-Paul Calbimonte, Emanuele Della Valle, and Oscar Corcho. 2015. Towards a unified language for RDF stream query processing. In *Proc. 12th Extended Semantic Web Conf.* 353–363.

[27] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 619–630. https://doi.org/10.1145/2723372.2742786

[28] Orri Erling and Ivan Mikhailov. 2009. RDF Support in the Virtuoso DBMS. In *Networked Knowledge-Networked Media*, Tassilo Pellegrini, Sóren Auer, Klaus Tochtermann, and Sebastian Schaffert (Eds.). 7–24.

[29] Wenfei Fan, Chunming Hu, and Chao Tian. 2017. Incremental graph computations: Doable and undoable. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 155–169.

[30] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. 2005. On graph problems in a semi-streaming model. *Theor. Comp. Sci.* 348, 2-3 (2005), 207–216.

[31] George H. L. Fletcher, Jeroen Peters, and Alexandra Poulovassilis. 2016. Efficient regular path query evaluation using path indexes. In *Proc. 19th Int. Conf. on Extending Database Technology*, Evaggelia Pitoura, Sofian Maabout, Georgia Koutrika, Amélie Marian, Letizia Tanca, Ioana Manolescu, and Kostas Stefanidis (Eds.). 636–639. https://doi.org/10.5441/002/edbt.2016.67

[32] Libo Gao, Lukasz Golab, M. Tamer Özsu, and Gunes Aluc. 2018. Stream WatDiv – A Streaming RDF Benchmark. In *Proc. ACM SIGMOD Workshop on Semantic Big Data*. 3:1–3:6.

[33] Lukasz Golab and M. Tamer Özsu. 2003. Issues in data stream management. *ACM SIGMOD Rec.* 32, 2 (2003), 5–14.

[34] Lukasz Golab and M. Tamer Özsu. 2005. Update-Pattern-Aware Modeling and Processing of Continuous Queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 658–669.

[35] Lukasz Golab and M. Tamer Özsu. 2010. *Data Stream Systems*. Morgan & Claypool.

[36] Todd J Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. 2003. Processing XML streams with deterministic automata. In *Proc. 9th Int. Conf. on Database Theory*. 173–189.

[37] Ajeet Grewal, Jerry Jiang, Gary Lam, Tristan Jung, Lohith Vuddemarri, Quannan Li, Aaditya Landge, and Jimmy Lin. [n. d.]. RecService: Multi-Tenant Distributed Real-Time Graph Processing at Twitter. In *Proc. 10th USENIX Workshop on Hot Topics in Cloud Computing*.

[38] Andrey Gubichev. 2015. *Query Processing and Optimization in Graph Databases*. Ph.D. Dissertation. Technische Universität München.

[39] Andrey Gubichev, Srikanta J Bedathur, and Stephan Seufert. 2013. Sparqling kleene: fast property paths in RDF-3X. In *Proc. 1st Int. Workshop on Graph Data Management Experiences and Systems*. 14.

[40] Martin Hirzel, Guillaume Baudart, Angela Bonifati, Emanuele Della Valle, Sherif Sakr, and Akrivi Vlachou. 2018. Stream Processing Languages in the Big Data Era. *ACM SIGMOD Rec.* 47, 2 (2018), 29–40.

[41] John Hopcroft. 1971. An n log n algorithm for minimizing states in a finite automaton. Elsevier Science Publishers, 189–196.

[42] Bruce M Kapron, Valerie King, and Ben Mountjoy. 2013. Dynamic graph connectivity in polylogarithmic worst case time. 1131–1142.

[43] Krys J Kochut and Maciej Janik. 2007. SPARQLeR: Extended SPARQL for semantic association discovery. In *Proc. 4th European Semantic Web Conf.* 145–159.

[44] Srdjan Komazec, Davide Cerri, and Dieter Fensel. 2012. Sparkwave: continuous schema-enhanced pattern matching over RDF data streams. In *Proc. 6th Int. Conf. Distributed Event-Based Systems*. 58–68.

[45] André Koschmieder and Ulf Leser. 2012. Regular path queries on large graphs. In *SSDBM12*. 177–194.

[46] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 239–250. https://doi.org/10.1145/2723372.2742788

[47] Jakub Łącki. 2011. Improved deterministic algorithms for decremental transitive closure and strongly connected components. 1438–1445.

[48] 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 *Proc. 10th Int. Semantic Web Conf.* 370–388.

[49] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. 2016. Querying graphs with data. *J. ACM* 63, 2 (2016), 14.

[50] Leonid Libkin and Domagoj Vrgoč. 2012. Regular path queries on graphs with data. In *Proc. 15th Int. Conf. on Database Theory*. 74–85.

[51] Wim Martens and Tina Trautner. 2017. Enumeration problems for regular path queries. *arXiv preprint arXiv:1710.02317* (2017).

[52] Andrea Mauri, Jean-Paul Calbimonte, Daniele Dell'Aglio, Marco Balduini, Marco Brambilla, Emanuele Della Valle, and Karl Aberer. 2016. Triplewave: Spreading RDF streams on the web. In *Proc. 15th Int. Semantic Web Conf.* 140–149.

[53] Andrew McGregor. 2014. Graph stream algorithms: a survey. *ACM SIGMOD Rec.* 43, 1 (2014), 9–20.

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

[55] Shanmugavelayutham Muthukrishnan et al. 2005. Data streams: Algorithms and applications. *Trends in Theoretical Computed Science* 1, 2 (2005), 117–236.

[56] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. 2020. Regular Path Query Evaluation on Streaming Graphs. *arXiv preprint arXiv:2004.02012* (2020).

[57] Anil Pacaci, Alice Zhou, Jimmy Lin, and M. Tamer Özsu. 2017. Do We Need Specialized Graph Databases?: Benchmarking Real-Time Social Networking Applications. In *Proc. 5th Int. Workshop on Graph Data Management Experiences and Systems*. Article 12, 7 pages. https://doi.org/10.1145/3078447.3078459

[58] Ashwin Paranjape, Austin R Benson, and Jure Leskovec. 2017. Motifs in temporal networks. In *Proc. 10th ACM Int. Conf. Web Search and Data Mining*. 601–610.

[59] Kostas Patroumpas and Timos Sellis. 2006. Window specification over data streams. In *Advances in Database Technology, Proc. 10th Int. Conf. on Extending Database Technology*. 445–464.

[60] 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. *Proc. VLDB Endowment* 11, 12 (2018), 1876–1888.

[61] Liam Roditty and Uri Zwick. 2016. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. *SIAM J. on Comput.* 45, 3 (2016), 712–733.

[62] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2018. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proc. VLDB Endowment* 11, 4 (2018), 420–431.

[63] S. Seufert, A. Anand, S. Bedathur, and G. Weikum. 2013. FERRARI: Flexible and efficient reachability range assignment for graph indexing. In *Proc. 29th Int. Conf. on Data Engineering*. 1009–1020. https://doi.org/10.1109/ICDE.2013.6544893

[64] Andy Seaborne Steve Harris. [n. d.]. SPARQL 1.1 Query Language. https://www.w3.org/TR/sparql11-query/

[65] Jiao Su, Qing Zhu, Hao Wei, and Jeffrey Xu Yu. 2016. Reachability querying: can it be even faster? *IEEE Trans. Knowl. and Data Eng.* 29, 3 (2016), 683–697.

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

[67] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 147–156.

[68] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *Proc. 4th Int. Workshop on Graph Data Management Experiences and Systems*. 7.

[69] Sarisht Wadhwa, Anagh Prasad, Sayan Ranu, Amitabha Bagchi, and Srikanta Bedathur. 2019. Efficiently Answering Regular Simple Path Queries on Large Labeled Networks. In *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '19)*. New York, NY, USA, 1463–1480. https://doi.org/10.1145/3299869.3319882

[70] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. 2016. Query planning for evaluating SPARQL property paths. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 1875–1889.

[71] Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. 2010. GRAIL: scalable reachability index for large graphs. *Proc. VLDB Endowment* 3, 1 (2010), 276–284. Issue 1-2. http://dl.acm.org/citation.cfm?id=1920841.1920879

[72] Ying Zhang, Pham Minh Duc, Oscar Corcho, and Jean-Paul Calbimonte. 2012. SRBench: A Streaming RDF/SPARQL Benchmark. In *Proc. 11th Int. Semantic Web Conf.* 641–657.