Evaluating Complex Queries on Streaming Graphs
Anil Pacaci, Angela Bonifati, M. Tamer Ozsu

To cite this version:
Anil Pacaci, Angela Bonifati, M. Tamer Ozsu. Evaluating Complex Queries on Streaming Graphs. 2022 IEEE 38th International Conference on Data Engineering (ICDE), May 2022, Kuala Lumpur, Malaysia. pp.272-285, 10.1109/ICDE53745.2022.00025 . hal-03907496

HAL Id: hal-03907496
https://hal.science/hal-03907496
Submitted on 20 Dec 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Evaluating Complex Queries 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 the problem of evaluating persistent queries over streaming graphs in a principled fashion. These queries need to be evaluated over unbounded and very high speed graph streams. We define a streaming graph data model and query model incorporating navigational queries, subgraph queries and paths as first-class citizens. To support this full-fledged query model we develop a streaming graph algebra that describes the precise semantics of persistent graph queries with their complex constructs. We present transformation rules and describe query formulation and plan generation for persistent graph queries over streaming graphs. Our implementation of a streaming graph query processor shows the feasibility of our approach and allows us to gauge the high performance gains obtained for query processing over streaming graphs.

1 INTRODUCTION

Many modern applications in various domains now operate on very high speed streaming graphs [62]. For example, Twitter’s recommendation system ingests 12K events/sec on average [31], Alibaba transaction graph processes 30K edges/sec at its peak [60]. Efficient querying of these streaming graphs is a crucial task for these applications that monitor complex graph patterns and relationships.

Existing graph DBMSs mostly follow the traditional database paradigm where data is persistent and queries are transient; consequently, they do not support persistent query semantics where queries are registered into the system and results are generated incrementally as the graph edges arrive. Persistent queries on streaming graphs enable users to continuously obtain new results on rapidly changing data, supporting online analysis and real-time query processing, the latter being an important functionality of future graph processing engines [64]. This is demonstrated in the following example.

Example 1. In many online social networking applications users post original content, sometimes link this to other users’ content and react to each other’s posts. We say that a user $u_2$ is a recentLiker for another user $u_1$ if $u_2$ has recently liked posts that are created by $u_1$ and $u_2$, and $u_1$ are following each other. The goal of the recommendation service is to notify users, in real-time, of new content that are posted by others that are connected by a path of recentLiker relationship – these constraints are modeled as a complex graph pattern as the one shown in Figure 1. The service might provide the context for its recommendations by returning the full paths of people who are recent likers such as the path between users $u_1$ and $u_2$. This real-time notification task is an example of a persistent query over the streaming graph of user interactions that returns the recommended content in real-time.

Real-world applications that feature complex graph patterns as the one shown in the above example require:

- (R1) subgraph queries that find matches of a given graph pattern (e.g. in Figure 1 the triangle pattern involving posts, likes and transitive closure of the follows relationship)
- (R2) path navigation queries that traverse paths based on user specified constraints (e.g. in Figure 1 arbitrary-length paths of the recentLiker relationship); and
- (R3) the ability to treat paths as first-class citizens of the data model, hence to manipulate and return paths (e.g. in Figure 1 the query returns the full paths of recentLiker).

Even in the context of one-time queries over static graphs, these requirements are poorly addressed by existing graph DBMSs and their languages. Existing query languages (e.g., PGQL, SPARQL v1.1, Cypher) address the first two issues by replacing edge labels of a subgraph pattern with regular expressions – this is known as unions of conjunctive RPQs (UCRPQ) [14, 72]. However, UCRPQ lacks algebraic closure and composability, limiting optimization opportunities. Furthermore, the output of a path navigation query is typically a set of pairs of vertices that are connected by a path under the constraints of a given regular expression. Hence, UCRPQ-based query languages limit path navigation queries to boolean reachability, without the ability to return and manipulate paths. G-CORE [6] addresses these limitations at the language specification level and has influenced the standardization efforts for a query language for graph DBMSs (see https://www.gqlstandards.org/). To the best of our knowledge, there is no work that uniformly addresses all three requirements.

The issues become more complex in the context of persistent query processing over streaming graphs, which is the focus of this paper. A number of specialized algorithms focus on evaluating subgraph pattern queries on streaming graphs [5, 18, 35, 44, 60], and our previous work focuses on path navigation queries and introduces the first streaming algorithms for RPQ evaluation [57], a limited subset of UCRPQ. However, a general-purpose model and framework that address the above discussed requirements of real-world applications (which feature complex graph patterns) in a uniform and principled manner is currently missing.

Querying streaming data in real-time imposes additional and novel requirements:
• (R4) graph streams are unbounded, making it infeasible to employ batch algorithms on the entire stream; and
• (R5) graph edges arrive at a very high rate and real-time answers are required as the graph emerges.

Existing graph DBMSs based on the snapshot model are not able to keep up with the high arrival rates [58]. The unsuitability of existing graph DBMSs for querying streaming data has motivated the design of specialized systems addressing singular application needs (e.g., [18, 31, 35, 44, 60]). The lack of systematic support for query processing over streaming graphs hinders the development of a general-purpose query processor for streaming graphs.

In this paper, we study the design of a general-purpose query processor for streaming graphs that addresses all of the above-discussed requirements in a uniform and principled manner. In analogy to traditional DBMSs, our framework provides the foundational tools to realize the well-known steps of query processing for streaming graph queries as follows:

(1) a streaming graph query expressed in a declarative, high-level user language is translated into a query plan that consists of logical operators with precise semantics;
(2) algebraic transformation rules are used to explore the plan space through query rewrites to find a “good” one, paving the way for query optimization;
(3) the execution plan is built by selecting appropriate physical implementations of logical operators that are incremental and non-blocking;
(4) the execution engine continuously executes the persistent query upon arrival of new edges to obtain new results.

Our focus in this paper is on the following key aspects of this framework:

(1) We introduce the formal Streaming Graph Query model (SGQ – Section 4) based on a streaming graph model (Section 3). SGQ is based on the Regular Query (RQ) model1, and provide precise semantics for persistent graph queries with subgraph patterns, path navigations and windowing constructs. Most of all, SGQ treats paths as first-class citizens, enabling queries to return and manipulate paths. Moreover, SGQ is closed under transitive closure and composable.

(2) We present the Streaming Graph Algebra (SGA – Section 5.1) as a foundational basis for evaluating SGQ and provide an algorithm for translating SGQ into SGA expressions (Section 5.2). Utilizing SGA’s closedness (Section 5.3), we introduce transformation rules for novel SGA operators for systematic exploration of the plan space (Section 5.4).

(3) We describe query execution plans (Section 6.1) and non-blocking physical implementations (Section 6.2).

(4) We describe a prototype implementation of a streaming graph query processor based on Timely Dataflow [54], i.e., the dataflow computational model (Section 6).

Unlike existing work on streaming graphs that relies on ad hoc algorithms, our algebraic approach provides the foundational framework to precisely describe the semantics of complex persistent graph queries over streaming graphs and to optimize such queries by query optimizers. SGA unifies path navigation and subgraph pattern queries in a structured manner (R1 & R2), i.e., it attains composability by properly closing UCRPQ under recursion. In addition, our framework treats paths as first-class citizens of its data model, enabling the proposed algebra to express queries that return and manipulate paths (R3). Extensive experimental analysis over real and synthetic datasets of our prototype implementation that incorporates incremental, non-blocking algorithms as physical operators (R4 & R5) demonstrates the feasibility and the performance gains of our approach. To the best of our knowledge, this is the first work to study the design of a streaming graph query processor, and to introduce an end-to-end solution to evaluating streaming graph queries with complex constructs.

2 RELATED WORK

2.1 Stream Processing Systems

Early research on stream processing primarily adapt the relational model and its query operators to the streaming setting (e.g., STREAM [8], Aurora [2], Borealis [1]). In contrast, modern Data Stream Processing Systems (DSPS) such as Storm [68], Heron [40], Flink [17] do not necessarily offer a full set of DBMS functionality. Existing literature (as surveyed by Hirzel et al. [33]) heavily focus on general-purpose systems and do not consider core graph querying functionality such as subgraph pattern matching and path navigation.

Existing work on streaming graph systems, by and large, target graph analytics workloads with little or no focus on graph querying functionalities on which we focus in this paper (e.g., STINGER [21], GraphOne [41, 42], GraphIn [66], GraphTau [34], GraPu [67], GraphBolt [49]). The primary focus of these systems is to build and maintain a sequence of snapshots for iterative graph analytics workloads under a stream of updates. SPARQL extensions have also been proposed for persistent query processing over RDF streams (e.g., C-SPARQL [12], CQELS [43], SPARQLstream [16] and RSP-QL [28]). These are the most similar to our setting, but they are designed for SPARQLv1.0 and cannot formulate path expressions such as RPQs that cover more than 99% of all recursive queries found in massive Wikidata query logs [15]. Also, query processors of these systems do not employ incremental operators, except Sparkwave [37] that focuses on stream reasoning. Our proposed framework supports complex graph patterns arising in existing graph query languages, including SPARQL v1.1 property paths, and introduces non-blocking operators optimized for streaming workloads.

2.2 Incremental View Maintenance

A persistent query over sliding windows can be formulated as an IVM problem, where the view definition is the query itself and window movements are updates to the underlying database. Although conceptually similar, the algorithms (e.g., the classical Counting [32] algorithm) and systems (e.g., DBToaster [36], F-IVM [56], ViewDF [74]) in this category primarily target non-recursive queries. General-purpose IVM techniques for recursive queries (such as DRed [32] and Absorption Provenance [48]) ignore the structure of graph queries and the inherent temporal patterns of streaming graphs, resulting in significant computational overhead to compute and store a large number of derivations. Differential

1 A computationally well-behaved subset of Datalog with tractable query evaluation and decidable query containment, similar to relational algebra [61].
Dataflow (DD) [51] is a state-of-the-art distributed system for incremental maintenance of possibly cyclic dataflows. Graphsurge [63] uses DD as the underlying execution layer to share computation across multiple, possibly recursive views over static graphs. As we show in this paper (Section 6.2), DD can also be used for evaluating SQQ. Our framework, in contrast to these, exploits the structure of graph queries and the temporal patterns of sliding windows to minimize the cost of evaluating complex queries on streaming graphs (Section 7.2.2).

2.3 Graph Algorithms
TriAL [45] and Temporal Graph Algebra (TGA) [53] adopt an algebraic approach to graph query processing, similar to ours. TriAL is designed to be used for one-time navigational queries over static triplestores, and it cannot be used as a standalone graph query language. In contrast, our proposed SGA is designed as a standalone language for persistent graph queries over streaming graphs, and it can express complex graph patterns expressed in high-level user languages (Section 4.2). TGA extends temporal relational operators to PGM, and its Spark implementation introduces physical operator implementations for its algebraic primitives [4]. However, it is designed for exploratory graph analytics over the entire history of changes. In contrast, our framework is designed to continuously evaluate graph queries as the underlying (potentially unbounded) streaming graph changes.

The closest to ours are specialized algorithms on dynamic and streaming graphs [5, 18, 35, 44, 57, 60]. Some of these [18, 35] study the incremental evaluation of subgraph pattern queries. Their focus is developing efficient incremental algorithms to maintain matches of a given subgraph pattern as the underlying graph changes. Ammar et al. [5] present distributed worst-case-optimal join algorithms for subgraph pattern queries. Li et al. [44] study subgraph isomorphism search over streaming graphs with timing-order constraints. Graphs [60] introduces efficient index structures that are optimized for cycle detection queries. In previous work [57], we study the design space of algorithms for path navigation over streaming graphs and provide algorithms for persistent evaluation of RPQ, a subset of the class of queries that we address in this paper. In contrast we target a general query model that captures the precise semantics of streaming graph queries.

3 DATA MODEL: STREAMING GRAPHS
3.1 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, \( \Sigma \) 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 (Path and Path Label). Given \( u, v \in V \), a path \( p \) from \( u \) to \( v \) in graph \( G \) is a sequence of edges \( u \xrightarrow{p} v : (e_1, \ldots, e_n) \). The label sequence of a path \( p \) is defined as the concatenation of edge labels, i.e., \( \phi^p(p) = \phi(e_1) \cdots \phi(e_n) \in \Sigma^* \).

We use \( T = (T, \leq) \) to define a discrete, total ordered time domain and use timestamps \( t \in T \) to denote time instants. Without loss of generality, the rest of the paper uses non-negative integers to represent timestamps.

Definition 3 (Streaming Graph Edge). A streaming graph edge (sge) is a quadruple \((\text{src}, \text{trg}, l, t)\) where \( \text{src}, \text{trg} \in V \) are endpoints of an edge \( e \in E \), \( l \in \Sigma \) is the label of the edge \( e \), and \( t \in T \) is the event (application) timestamp assigned by the source, i.e., \( \psi(e) = (\text{src}, \text{trg}) \) and \( \phi(e) = l \).

Definition 4 (Input Graph Stream). An input graph stream is a continuously growing sequence of streaming graph edges \( S^t = \{\text{sge}_1, \text{sge}_2, \cdots\} \) where each sge represents an edge in graph \( G \) and sges are non-decreasingly ordered by their timestamps.\(^3\)

Input graph streams represent external sources that provide the system with the graph-structured data. Our proposed framework uses a different format that generalizes Definition 4 to also represent intermediate results and outputs of persistent queries (Definition 7).

Definition 5 (Validity Interval). A validity interval is a half-open time interval \([ts, exp)\) consisting of all distinct time instants \( t \in T \) for which \( ts \leq t < exp \).

Timestamps are commonly used to represent the time instant in which the interaction represented by the sge occurred [44, 57, 60]. Alternatively, we use intervals to represent the period of validity of sges. In this paper, we argue that using validity intervals leads to a succinct representation and simplifies operator semantics by separating the specification of window constructs from operator implementation. As an example, each sge with timestamp \( t \) can be assigned a validity interval \( [t, t+1) \) that corresponds to a single time unit with smallest granularity that cannot be decomposed into smaller time units.\(^4\) Similarly, an sge \( e = (u, v, l, [ts, exp]) \) with a validity interval is equivalent to a set of sges \( \{(u, v, l, t_1), \cdots, (u, v, l, t_n)\} \) where \( t_1 = ts \) and \( t_n = exp - 1 \). Windowing operator (to be precisely defined momentarily in Section 5.1) are used to assign validity intervals based on the windowing specifications of a given query.

3.2 Streaming Graphs
We now describe the logical representation of streaming graphs that is used throughout the paper. First, we extend the directed labeled graph model with materialized paths to represent paths as first-class citizens of the data model. As per Definition 2, a path between vertices \( u \) and \( v \) is a sequence of edges \( u \xrightarrow{p} v : (e_1, \ldots, e_n) \) that connects vertices \( u \) and \( v \), i.e., the path \( p \) defines a higher-order relationship between vertices \( u \) and \( v \) through a sequence of edges. By treating paths as first-class citizens like vertices and edges, the materialized path graph model enables queries to have paths as inputs and outputs. In addition, it enables edges and paths to be stitched together to form complex graph patterns as will be shown in Section 5.1.

Definition 6 (Materialized Path Graph). A materialized path graph is a 7-tuple \( G = (V, E, P, \Sigma, \psi, \phi, \rho) \) where \( V \) is a set of vertices, \( E \) is a set of edges, \( P \) is a set of paths, \( \Sigma \) is a set of labels, \( \psi : E \rightarrow V \times V \) is an incidence function, \( \phi : E \rightarrow \Sigma \) is an edge labelling function that

\(^3\)We assume that sges are generated by a single source and arrive in order, and leave out-of-order arrival as future work.

\(^4\)Commonly referred as NO8 windows as described in Section 5.1.
assigns each path to a finite, ordered sequence of edges in $E$, and
\[ \phi : (E \cup P) \rightarrow \Sigma \] is a labeling function, where images of $E$ and $P$
under $\phi$ are disjoint, i.e., $\phi(E) \cap \phi(P) = \emptyset$.

The function $\rho$ assigns to each $p : u \rightarrow v \in P$ an actual path
$(e_1, \ldots, e_n)$ in graph $G$ satisfying: for every $i \in [1, \ldots, n]$, $\psi(e_i) =$
$(\text{src}_i, \text{trg}_i)$. $\text{trg}_i = \text{src}_{i+1}$, and $\text{src}_1 = u, \text{trg}_n = v$. Materialized
path graph is a strict generalization of the directed labeled graph
model (Definition 1), i.e., each directed labeled graph $G$ is also a
materialized path graph where $P = \emptyset$. We now generalize the notion
of streaming graph edges (Definition 3) as follows:

**Definition 7 (Streaming Graph Tuple).** A streaming graph
tuple (sgt) is a quintuple $\text{sgt} = (\text{src}, \text{trg}, l, [ts, \text{exp}], D)$ where distin-
guished (explicit) attributes $\text{src}, \text{trg} \in V$ are endpoints of an edge
e $E$ or a path $p \in P$ in graph $G$ and $l \in \Sigma$ is its label, and non-
distinguished (implicit) attributes $[ts, \text{exp}] \in T \times T$ is a half-open
time-interval representing $t$’s validity and $D$ is a payload consists of
dges in $E$ that participated in the generation of the tuple $t$.

Streaming graph tuples generalize sges (Definition 3) to repre-
sent, in addition to input graph edges, derived edges (new edges
as operator and query results that are not necessarily part of the
input graph) and paths (sequence of edges as operator and query
results). We use the notation $E' \subseteq E$ to denote the set of input graph
dges, and $\phi(E')$ to denote the fixed set of labels that are reserved
for input graph edges. Additionally, non-distinguished (implicit) attribute $D$ of an sgt $t$ captures the path $p$, i.e., sequence of edges,
in case the sgt $t$ represents a path. Otherwise, $D$ is the edge $e$ that
the sgt $t$ represents.

**Definition 8 (Streaming Graph).** A streaming graph $S$ is a con-
tinuously growing sequence of streaming graph tuples $S = [t_1, t_2, \cdots]$ in which each tuple $t_i$ arrives at a particular time $t_i$ ($t_1 < t_j$ for $i < j$).

Figure 2 depicts an excerpt of the input graph stream of the
application in Example 1, where each tuple represent an interaction
and each timestamp represent the time instant that the interaction
occurs. Its corresponding streaming graph where each tuple is
assigned a time interval is shown in Figure 3 (validity intervals are
assigned by WSCAN operator – see Section 5.1).

Unless otherwise specified, we consider streaming graphs to be
append-only, i.e., each sgt represents an insertion, and use the
direct approach to process expirations due to window movements.
Explicit deletions of previously arrived sgts can be supported by
explicitly manipulating the validity interval of a previously arrived

![Figure 2: The input graph stream from an online social network of Example 1](image1)

![Figure 3: Streaming graph obtained from the input graph stream in Figure 2 where the validity interval of each element is set based on a 24h window](image2)

![Figure 4: Snapshot graph of of the streaming graph in Figure 3 at $t = 25$.](image3)

sgt [39]. This corresponds to the negative tuple approach [25, 29].
Section 6.2 describes the processing of insertions, deletions and expirations under alternative window semantics for physical operator implementations.

**Definition 9 (Logical Partitioning).** A logical partitioning of
a streaming graph $S$ is a label-based partitioning of its tuples and it
produces a set of disjoint streaming graphs $\{S_1, \cdots, S_n\}$ where each
$S_i$ consists of sgts of $S$ with the label $l_i$, i.e., $S = \bigcup_{i \in L} S_i$

This label-based partitioning of streaming graphs provides a coherent representation for inputs and outputs of operators in
logical algebra (Section 5.1). At the logical level, it can be performed by the filter operator of the logical algebra (precisely defined in
Definition 17), and logical operators of our algebra process logically partitioned streaming graphs as their inputs and outputs unless
otherwise specified.

**Definition 10 (Value-Equivalence).** Sgets $t_1 = (u_1, v_1, l_1, [ts_1, \text{exp}_1], D_1)$ and $t_2 = (u_2, v_2, l_2, [ts_2, \text{exp}_2], D_2)$ are value-
equivalent iff their distinguished attributes are equal, i.e., $t_1$ either
represent the same edge or the same path possibly with different
validity intervals. Formally, $t_1 = t_2 \Leftrightarrow u_1 = u_2, v_1 = v_2, l_1 = l_2$. Value-equivalence is used for temporal coalescing of tuples with
adjacent or overlapping validity intervals [47]. We extend the coa-
lescent primitive from temporal database literature [19] to sgts with an
aggregation function over the non-distinguished payload attribute,
$D$, as shown below:

**Definition 11 (Coalesce Primitive).** The coalesce primitive
merges a set of value-equivalent sgts $\{t_1, \cdots, t_n\}$, $t_i =
(src, trg, l, [ts, \text{exp}], D_i)$ for $1 \leq i \leq n$ with overlapping or adjacent
validity intervals using an operator-specific aggregation function $f_{agg}$
We formally describe SGQ based on a streaming generalization of the dependency graph of a Datalog program is a directed graph whose vertices are its predicates and edges represent dependencies between predicates, i.e., there is an edge from p to q if q appears in the body of a rule with head predicate p. A Datalog program is non-recursive iff its dependency graph is acyclic, i.e., no predicate depends recursively on itself.

Definition 14 (Snapshot-Reducibility). Let \( S \) be a streaming graph, \( Q \) a streaming graph query and \( Q^O \) its non-streaming, one-time counterpart. Snapshot reducibility states that each snapshot of the result of applying \( Q \) to \( S \) is equal to the result of applying its non-streaming version \( Q^O \) over the corresponding snapshots of \( S \), i.e., \( \forall t \in T, \tau_t(Q(S)) = Q^O(\tau_t(S)) \).

Following existing research [27], we define the semantics of persistent evaluation of SGQ using the notion of snapshot reducibility (Definition 14). It is known that for many operations such as joins

We formally describe SGQ based on a streaming generalization of the dependency graph of a Datalog program is a directed graph whose vertices are its predicates and edges represent dependencies between predicates, i.e., there is an edge from p to q if q appears in the body of a rule with head predicate p. A Datalog program is non-recursive iff its dependency graph is acyclic, i.e., no predicate depends recursively on itself.

Definition 14 (Snapshot-Reducibility). Let \( S \) be a streaming graph, \( Q \) a streaming graph query and \( Q^O \) its non-streaming, one-time counterpart. Snapshot reducibility states that each snapshot of the result of applying \( Q \) to \( S \) is equal to the result of applying its non-streaming version \( Q^O \) over the corresponding snapshots of \( S \), i.e., \( \forall t \in T, \tau_t(Q(S)) = Q^O(\tau_t(S)) \).

Following existing research [27], we define the semantics of persistent evaluation of SGQ using the notion of snapshot reducibility (Definition 14). It is known that for many operations such as joins
and aggregation, exact results cannot be computed with a finite memory over unbounded streams [9]. In streaming systems, a common solution for bounding the space requirement is to evaluate queries on a window of data from the stream. The windowed evaluation model provides a tool to process unbounded streams with bounded memory, and restricts the scope of queries to recent data, a desired feature in many applications [9, 27]. Additionally, as opposed to streaming approximation techniques that trade off exact answers in favour of bounding the space requirements, window-based query evaluation enables exact query answers w.r.t. window specifications. Hence, we adopt the time-based sliding window for SGQ in the remainder.

**Definition 15 (Streaming Graph Query – SGQ).** An SGQ query \( Q \) is an RQ defined over a streaming graph \( S \) and a time-based sliding window \( W_T \) whose semantics is defined using the corresponding, one-time RQ \( Q^O \) and the notion of snapshot reducibility (Definition 14):

\[
\forall t \in T, \quad \tau_t(Q(S, W_T)) = Q^O(\tau_t(W_T(S)))
\]

Figure 5 illustrates the correspondence between streaming and one-time graph queries. A direct consequence of such a relationship is that SGQ can be evaluated by repeatedly executing the corresponding one-time query, known as query re-evaluation [25]. Specifically, the resulting streaming graph of an SGQ can be obtained from the sequence of snapshots that is the result of repeated evaluation of the corresponding one-time query at every time instant: an sgt \( (u, v, l, [ts, expiry], D) \) is in the resulting streaming graph for an SGQ \( Q \) if there is an edge \( e = (u, v, l) \) or a path \( p : u \to v \) with \( l = \phi^R(p) \) in the resulting snapshot graph of the corresponding one-time query \( G_{ts} = Q^O(\tau_t(W_T(S))) \) for \( ts \leq t < expiry \). However, such a strategy is wasteful as the input differences between two consecutive instants are likely to be small. Alternatively, incremental evaluation computes the changes in the output as new sgts arrive and old sgts expire due to window movements. In this paper, we focus on the incremental evaluation method and use the concept of snapshot reducibility to ensure correct evaluation semantics.

### 4.2 SGQ in Practice

The SGQ model formalizes an important class of graph queries in the streaming model. It captures the core features of current graph query languages such as subgraph pattern and reachability-based path queries. In this section, we demonstrate the SGQ’s expressive power by mapping core G-CORE constructs to SGQ. We choose G-CORE for demonstration due to the following reasons. G-CORE fulfills all three requirements of graph querying (R1, R2 & R3 in Section 1). Other existing languages (e.g., SPARQL v1.1, Cypher, PGQL) can only partially satisfy these requirements due to (i) the lack of algebraic closure and composability, and (ii) limited path navigation capability [14]. Moreover, G-CORE supports SGQ capabilities such as the treatment of paths as first-class citizens and returning graphs. Finally, G-CORE is one of the more prominent language specifications influencing the ongoing standardization process of a graph query language GQL (https://www.gqlstandards.org).

**Example 3.** The G-CORE query in Figure 6 represents the real-time notification example in Example 1 (its corresponding RQ is already given in Example 2). Its **PATH** and **MATCH** clauses use ASCII-art syntax (b) to define complex graph patterns (f) with RPQ-based reachability queries, (f) and powerful path patterns as demonstrated in the two examples we discuss below.

**Example 4.** Consider the G-CORE query in Figure 7 that combines streaming information from a social network of user interactions and a transaction network of customer purchases to drive product recommendations. Its defines a view over the resulting streaming graph of recommendation edges (d) by joining patterns from two streaming graphs (c), and its **MATCH** clause features optional predicates to incorporate two alternative social interactions (d). Its graph pattern...
corresponds to the following RQ:
\[ \text{ACQ}(u_1, u_2) \leftarrow l(u_1, m_1), p(u_2, m_1) \]
\[ \text{ACQ}(u_1, u_2) \leftarrow f(u_1, u_2) \]
\[ \text{REC}(u, p) \leftarrow \text{ACQ}(u_1, u_2), p(u_2, p) \]
\[ \text{Answer}(u, p) \leftarrow \text{REC}(u, p) \]
where predicates \( l, f, p, p, \text{ACQ}, \text{REC} \) represent labels likes, follows, post, purchase, acquaintance, and recommendation, respectively.

5 STREAMING GRAPH ALGEBRA

This section presents the logical foundation of our streaming graph query processing framework. We first introduce streaming graph algebra (SGA) and the semantics of its operators (Section 5.1). The main motivation behind the proposed SGA is similar to that of relational DBMSs: it enables us to formulate and represent query plans independent of specific physical implementations. We subsequently describe how to transform a given SQ (Definition 15) into its canonical SGA expression and illustrate logical query plans (Section 5.2) and prove its closedness and composability (Section 5.3). Finally, we present some transformation rules for SGA to demonstrate alternative logical plan generation opportunities (Section 5.4).

5.1 SGA Operators

For ease of exposition, the rest of the paper assumes that inputs to each SGA operator are partitioned into one more streaming graphs \( S_g \) based on tuple labels where each \( S_g \) contains sgts with the same label \( a \in \Sigma \). The output of each operator is also a streaming graph \( S_g \) where each sgt has the label \( a \in \Sigma \setminus \phi(E') \). SGA contains the following operators: windowing (Definition 16), filter (Definition 17), union (Definition 18), subgraph pattern (Definition 19), and path navigation (Definition 20).

Definition 16 (WSCAN). The windowing operator \( W \) transforms a given input graph stream \( S^I \) to a streaming graph \( S \) by adjusting the validity interval of each sgt based on the window size \( T \) and the optional slide interval \( \beta \), i.e., \( W_{T, \beta}(S^I) := S = \{ (u, v, l, [t, \exp], D) : e(u, v, l) \in S^I \wedge \exp = [t / \beta] \cdot \beta + T \} \).

The window size \( T \) determines the length of the validity interval of sgts and the slide interval \( \beta \) controls the granularity at which the time-based sliding window progresses [8, 57]. If \( \beta \) is not provided, default is \( \beta = 1 \), i.e., single time instant with the smallest granularity, and it defines a sliding window that progresses at every time instant.

The WSCAN operator defines the semantics of time-based sliding windows. It acts as an interface between the external streaming graph sources and the query plans and it is responsible for providing data from input graph streams to a query plan, similar to the scan operator in relational systems. WSCAN manipulates the implicit temporal attribute and associates a time interval to each sgt representing its validity. Our model of representing streaming graphs (Definition 8) provides a concise representation of validity intervals and enables operators to treat time differently than the data stored in the graph. This enables us to distinguish operator semantics from window semantics and eliminates the redundancy caused by integrating sliding window constructs into each operator of the algebra. SGA operators access and manipulate validity intervals implicitly, generalizing their non-streaming counterparts with implicit handling of time.

Example 5. Consider the real-time notification task of Example 1 with a 24-hour window of interest. WSCAN \( W_2 \) sets validity intervals of sgts of the input graph stream and produces a streaming graph where each sgt is valid for 24 hours, as shown in Figure 3.

Definition 17 (FILTER). The filter operator \( \text{Filter}_{\Phi}(S) \) is defined over a streaming graph \( S \) and a boolean predicate \( \Phi \) involving the distinguished attributes of sgts, and its output stream consists of sgts of \( S \) on which \( \Phi \) evaluates to true. Formally:
\[ \text{Filter}_{\Phi}(S) = \{ (u, v, l, [t, \exp], D) \mid (\text{src}, \text{trg}, l, [t, \exp], D) \in S \wedge \Phi((\text{src}, \text{trg}, l, D)) \} \]

Definition 18 (UNION). Union \( \cup^{[d]} \) with an optional output label \( d \in \Sigma \setminus \phi(E') \) merges sgts of two streaming graphs \( S_1 \) and \( S_2 \), and assigns the new label \( d \) if provided. Formally:
\[ S_1 \cup^{[d]} S_2 = \{ t \mid t \in S_1 \lor t \in S_2 \} \]

Definition 19 (PATTERN). The streaming subgraph pattern operator is defined as \( \text{Pattern}_{\Phi}^{\text{src, trg}, d}(S_1, \ldots, S_n) \) where each \( S_i \) is a streaming graph, \( \Phi \) is a conjunction of a finite number of terms in the form \( \phi\text{pos}_i = \phi\text{pos}_{i'} \) for \( \phi\text{pos}_i, \phi\text{pos}_{i'} \in \{ \text{src}_1, \text{trg}_1, \ldots, \text{src}_n, \text{trg}_n \} \) where \( \text{src}_i, \text{trg}_i \) are endpoints of sgts in \( S_i \), and \( \text{src}, \text{trg} \in \{ \text{src}_1, \text{trg}_1, \ldots, \text{src}_n, \text{trg}_n \} \) are the endpoints of resulting sgts, and \( d \in \Sigma \setminus \phi(E') \) represents the label of the resulting sgt. Formally:
\[ \text{Pattern}_{\Phi}^{\text{src, trg}, d}(S_1, \ldots, S_n) = \{ (u, v, d, [t, \exp], D) : e(u, v, l) \mid \exists (u, v, [t, \exp], D) \in S_i, 1 \leq i \leq n \wedge \Phi((\text{src}_i, \text{trg}_i, l, [t, \exp])) \wedge u = \text{src} \wedge v = \text{trg} \wedge \bigcap_{t \leq t \leq n} \{ t, \exp \} \neq \emptyset \wedge t = \max_{t \leq t \leq n} \{ t, \exp \} \} \]

Given a subgraph pattern expressed as a conjunctive query, PATTERN finds a mapping from vertices in the stream to free variables where (i) all query predicates hold over the mapping, and (ii) there exists a time instant at which each edge in the mapping is valid.

Example 6. Consider the real-time notification query given in Example 1; the recentLiker relationship defined in the form of a triangle pattern can be represented with PATTERN \( \text{Pattern}_{\Phi}^{\text{src, trg}, RL} \) where \( \phi = (\text{trg}_1 = \text{trg}_2 \land \text{src}_1 = \text{src}_2 \land \text{src}_1 \neq \text{src}_2) \). Its output over the streaming graph, given in Figure 3, consists of sgts \((y, RL, u, [28, 37], (y, RL, u)) \) and \((u, RL, v, [29, 31], (u, RL, v)) \) that correspond to derived edges with label recentLiker.

SGA operators may produce multiple value-equivalent sgts with adjacent or overlapping validity intervals. Unless otherwise specified, such sgts in resulting streaming graphs of SGA operators are coalesced to maintain the set semantics of streaming graphs and their snapshots (Definition 10). To illustrate, consider PATTERN in the above example: the pattern in Figure 3, the
PATTERN operator finds two distinct subgraphs with vertices \((u, b, v)\) and \((u, c, v)\). Consequently, it produces two value-equivalent tuples \((u, RL, v, [29, 31], (u, RL, v))\) and \((u, RL, v, [30, 31], (u, RL, v))\), which are coalesced into a single sgt by merging their validity intervals.

**Definition 20 (PATH).** The streaming path navigation operator is defined as \(\text{PATH}^\phi_R(S_{t_1}, \ldots, S_{t_n})\) where \(R\) is a regular expression over the alphabet \(\{l_1, \ldots, l_m\} \subseteq \Sigma\), and \(d \in \Sigma \setminus \phi(E^1)\) designates the label of the resulting sgt. The sgt \(t = (u, v, l, [ts, exp], D : p)\) is an answer for \(\text{PATH}^\phi_R\) if there exists a path \(p\) between \(u\) and \(v\) in the snapshot of \(S\) at time \(t\), i.e., \(p : u \xrightarrow{\phi \{l\}} v \in \tau_r(S) = G_t\), and the label sequence of the path \(p\), \(\phi^p(p)\), is a word in the regular language \(L(R)\). Formally:

\[
\text{PATH}^\phi_R(S_{t_1}, \ldots, S_{t_n}) = \{(u, v, d, [ts, exp], D) \mid \exists p : u \xrightarrow{\phi \{l\}} v \land \\
\forall t_i \in p, \exists t_i = (src_i, trg_i, l_i, [ts_i, exp_i], D_i) \in S_{t_i} \land \\
\phi^p(p) \in L(R) \land \bigcap_{t \in p} (t, ts, t.exp) \neq \emptyset \land \\
ts = \max_{t \in p} (t, ts) \land exp = \min_{t \in p} (t.exp) \land D = p\}.
\]

**Theorem 1.** There exists a SGA expression \(e \in \text{SGA}\) for any \(Q \in \text{SGQ}\).

**Proof.** The dependency graph of an RQ is acyclic as RQ is non-recursive (Definition 13); hence, Line 2 is guaranteed to define a partial order over \(Q\)’s predicates. Algorithm \(\text{SGQPParser}\) generates an SGA expression for each predicate in this order and caches it in exp array. In particular, Line 7 generates an SGA expression for each EDB predicate and Line 9 generates a PATH expression for each body predicate with a Kleene star. For each rule \(d(s_r, s trg) := l_1(s_{src1}, s_{trg1}), \ldots, l_n(s_{srcn}, s_{trgn})\), Line 13 generates a PATTERN expression. Finally, Line 14 generates a UNION expression if there are multiple rules with the same head predicate \(d\). As each predicate is processed based on the partial order defined by the dependency graph \(G_Q\) (Line 4), exp is guaranteed to have SGA expressions for each predicate \(r_t(1 \leq j \leq l)\) when processing predicate \(r_t\). Once all predicates are processed, Line 18 returns the SGA expression of the Answer predicate. Hence, Algorithm \(\text{SGQPParser}\) correctly constructs an SGA expression for a given SGQ.

The complexity of evaluating SGA expressions is the same as RQ given their relationship noted above: NP-complete in combined complexity and NLogspace-complete in data complexity [14, 61].
Composability is a desired feature for a declarative query language as it facilitates query decomposition, view-based query evaluation, query rewriting etc. SGA operators are closed over streaming graphs as defined in Section 3; that is, the output of an SGA operator that subsumes these existing plans. An example application of these transformation rules involves the other novel SGA operators:

**Transformation Rules for WSCAN:** WSCAN (\(W^T\)) commutes with operators that do not alter the validity intervals of sgts, i.e., UNION and FILTER. Pushing FILTER down the WSCAN operator can potentially reduce the rate of sgts and consequently the amount of state the windowing operator needs to maintain. Formally:

1. \(W^T(\sigma^T(S)) = \sigma^T(W^T(S))\)
2. \(W^T(S_1 \cup S_2) = W^T(S_1) \cup W^T(S_2)\)

**Transformation Rules for PATH:** We identify two transformation rules for PATH:

1. Alternation: \(P_{a,b}^d(S_a, S_b) = \bigcup^d(S_a, S_b)\)
2. Concatenation: \(P_{a,b}^d(S_a, S_b) := \exists_{\sigma^d} \triangleright_{a,b} \bigcup^d(S_a, S_b)\)

These transformation rules enable the exploration of a rich plan space for SQG that are represented by SGA. In particular, PATH and its transformation rules enable the integration of existing approaches for RPQ evaluation with standard optimization techniques such as join ordering and pushing down selection in a principled manner. Traditionally, path query evaluation follows two approaches [10, 23, 38, 57, 65, 73]: graph traversals guided by finite automata, or relational algebra extended with transitive closure, i.e., alpha-RA. Yakovets et al. introduce a hybrid approach (Waveguide) and model the cost factors that impact the efficiency of RPQ evaluation on static graphs [73]. SGA enables the representation of these approaches in a uniform manner, and the above transformation rules enable the exploration of a rich plan space that subsumes these existing plans. An example application of these transformation rules and a micro-benchmark demonstrating the potential benefits of query optimizations through plan space exploration is given in Section 7.4.

### 6 Implementation

We implemented a prototype streaming graph query processor (https://dsg.uwaterloo.github.io/s-graffito/) based on the algebraic framework we propose in this paper. Our goal is to build a plausible
conceptual framework for expressing and evaluating SGQ – not to have a complete system. Hence, we focus on the construction of query execution plans and physical implementations of logical SGA operators.

6.1 Query Processor

Conceptually, SGQ can be evaluated by repeatedly evaluating from scratch the corresponding one-time query at each point in time (Section 4.1). Albeit semantically correct, such a re-execution strategy is, of course, infeasible. In streaming systems, the focus is on incremental evaluation of persistent queries where the goal is to avoid re-computing the entire result by only computing the changes to the output in real-time as new input arrives. Consequently, queries are executed in a data-driven (push-based) manner as opposed to demand-driven (pull-based) query processing employed in traditional relational DBMSs [30]. As such, one can implement the framework proposed in this paper over existing streaming systems such as Apache Flink, Spark Streaming and Timely Dataflow. For our prototype, we use Timely Dataflow (TD) as the underlying execution engine.

Applications in TD are expressed as a directed graph of operations where vertices correspond to user-defined computations and edges correspond to the flow of data between them. In executing an SGQ, the query processor first creates a logical plan from the canonical SGA expression of the given query. The physical execution plan in the form of a dataflow graph is constructed by: (i) creating source vertices for leaves of the logical query plan that receives input graph streams, (ii) replacing logical SGA operators with physical operator implementations, and (iii) creating a sink vertex for the root of the logical query plan that pushes results back to the user. Section 6.2 describes physical operator implementations in detail.

TD associates each input data with a logical timestamp that enables fine-grained synchronization and progress tracking. We represent each input graph stream as an evolving collection where each item represents an sge (Definition 4) and event timestamps assigned by the source are used as logical timestamps. Upon the arrival of a new edge, TD propagates the corresponding sge through the physical execution plan and computes the new output at the given logical timestamp.

6.2 Physical Operator Algebra

6.2.1 Overview. Physical operator implementations for streaming systems have two requirements: they should be push-based and non-blocking, so they do not need the entire input to be available before producing the first result. The standard dataflow implementations of stateless FILTER and UNION operators can be directly used in SGA, and WSCAN can be implemented via the standard map operator that adjusts the validity intervals of sgts based on window specifications. We focus on the stateful operators PATTERN and PATH that need to maintain an internal operator state that is accessed during query processing. This state is updated as new sgts enter the window and old sgts expire. As discussed earlier, time-based sliding windows ensure that the portion of the input that may contribute to any future result is finite, making incremental, non-blocking computation possible.

TD’s Differential Dataflow (DD) layer [51] provides a set of built-in, high-level programming primitives (operators) that can be used to compose arbitrary dataflows for general-purpose computations, and it automatically incrementalizes these. Consequently, DD can be asked to evaluate SGQ by (i) creating a dataflow of DD operators for a given SGQ and (ii) maintaining the window content as an evolving collection. Indeed, we use such a strategy as a competitive baseline in our experimental evaluation (Section 7.2.2). However, DD’s generality comes at a performance cost for evaluating SGQ, as we show in Section 7.2.2. Below, we describe how to devise physical operator implementations specific to SGQ by utilizing the properties of the SGQ model. Of course, these are not the only physical operator implementations that are possible for SGA; other implementations can possibly be developed, and these are exemplars to demonstrate the implementability of the SGA operators. They are also what we use in the experiments.

6.2.2 Pattern Matching. Implementation of PATTERN models subgraph patterns as conjunctive queries that can be evaluated using a series of joins. There is a rich literature of streaming join implementations that can be used. Symmetric hash join [71] is commonly used to implement non-blocking joins in the streaming model: a hash table is built for each input stream and upon arrival (expiration) of a tuple, it is inserted into (removed from) its corresponding hash table and other tables are probed for matches [28, 69]. This produces an append-only stream of results for internal windows that do not invalidate previously reported results upon expiration of their participating tuples [26]. For external windows that require eviction of old results as the windows slide forward, expired results can be determined by maintaining expiration timestamps: a join result expires when one of its participating tuples expire. Our use of validity intervals enables the user or the application to adopt both window semantics without the need for explicit processing of expired input tuples.

Given a subgraph pattern, we take the standard approach of creating a binary join tree where leaves represent streaming graphs as input streams and internal nodes represent pipelined hash join operators. For instance, Figure 8 (right) shows the logical plan for the query in Example 1 and the join tree for its PATTERN. In our prototype, we use the ordering of predicates in PATTERN to construct the join tree and leave the problem of finding efficient join plans (e.g. using worst-case optimal joins [55]) for future investigation.

6.2.3 Path Navigation. DD’s iterate allows constructing cyclic dataflows that can model arbitrary nested iterations, and it can be used to evaluate PATH and its recursive path expressions. However, the use of recursion in SGQ is limited to transitive closure (Section 4), and the RPQ-based semantics of PATH is sufficient to evaluate this limited form of recursion (Theorem 1). In previous work [57], we study the design space of streaming algorithms for persistent RPQ under arbitrary path semantics. In brief, the streaming RPQ algorithm follows the automata-based RPQ evaluation method [14] and maintains a spanning forest-based data structure, called Λ-tree index, that enables the compact representation of partial path segments. In our prototype, we employ the streaming RPQ algorithm in [57] as the non-blocking, physical implementation of PATH operator, which eliminates the need for cycles in physical execution plans. Additionally, a spanning-tree-based representation of intermediate
results enables us to recover actual paths and allows queries to return and manipulate paths as first-class citizens.

6.2.4 Customized PATH Implementation. We now describe a novel algorithm Streaming Path Navigation (S-PATH) that can be used as an alternative physical operator for the PATH operator (Definition 20). In contrast to our previous work [57], S-PATH utilizes the validity intervals of path segments to simplify the state maintenance in the absence of explicit deletions. Algorithms in [57] are based on the negative tuple approach; expiration due to window movements are processed using the same machinery as explicit deletions. Upon expiration (deletion) of an edge, their algorithm first finds all results that are affected by the expiration (deletion), then it traverses the snapshot graph to ensure that there is no alternative path leading to the same result. This corresponds to re-derivation step of DRed [32], optimized for RPQ evaluation on streaming graphs. Instead, S-PATH utilizes the temporal pattern of sliding window movements and adopt the direct approach, i.e., it can directly determine expired tuples based on their validity intervals. This is possible due to the separation of the implementation of sliding windows from operator semantics via an explicit WSCAN operator.

Algorithm S-PATH:

input : Input streaming graph \( S \), Regular expression \( R \), output label \( o \)
output : Output streaming graph \( S_0 \)
1 \( A(\Sigma, \Delta, \delta, s_0, F) \leftarrow \text{ConstructDFA}(R) \)
2 Initialize \( \Delta - \text{PATH} \)
3 \( S_0 \leftarrow \emptyset \)
4 \( R \leftarrow \emptyset \)
5 foreach \((u,v,l,[t_s,exp],\Sigma) \in S\) do
6   foreach \( s,t \in S \) where \( t = \delta(s,t) \) do
7     if \( s = s_0 \land T_s \notin \Delta - \text{PATH} \) then
8       add \( T_s \) with root node \((u,s_0)\)
9     if \( s = s_0 \) then
10        if \((u,t) \notin T_s \) then
11           \( R \leftarrow R+ \text{Expand}(T_{u,s_0}, (u,t), e(u,v)) \)
12        else if \((u,t).\text{exp} < \text{exp} \) then
13           \( R \leftarrow R+ \text{Propagate}(T_{u,s_0}, (u,t), e(u,v)) \)
14     \( T \leftarrow \text{ExpandableTrees}(\Delta - \text{PATH}, (u,s), t_s) \)
15     foreach \( T \in T \) do
16        if \((u,t) \notin T \) then
17           \( R \leftarrow R+ \text{Expand}(T_{u,s_0}, (u,t), e(u,v)) \)
18        else if \((u,t).\text{exp} < \text{min}(e(u,v).\text{exp}, e(u,v)) \) then
19           \( R \leftarrow R+ \text{Propagate}(T_{u,s_0}, (u,t), e(u,v)) \)
20     foreach sgt \( t \in R \) do
21        push \( t \) to \( S_2 \)

Algorithm S-PATH incrementally performs a traversal of the underlying snapshot graph under the constraints of a given RPQ as sgt arrives. It first constructs a DFA from the regular expression of a PATH operator, and initializes a spanning forest-based data structure, called \( \Delta - \text{PATH} \), that is used as the internal operator state during query processing. \( \Delta - \text{PATH} \) is used to maintain a path segment, i.e., a partial result, between each pair of vertices in the form a spanning forest under the constraints of a given RPQ, consistent with Definition 20. Upon the arrival of an sgt, Algorithm S-PATH probes \( \Delta - \text{PATH} \) to retrieve partial path segments that can be extended with the edge (or a path segment) of the incoming sgt. Each partial path segment is extended with the incoming sgt, and Algorithm S-PATH traverses the snapshot graph \( G_t \) until no further expansion is possible.

**Definition 21 (Spanning Tree \( T_x \)).** Given an automaton \( A \) for the regular expression \( R \) of a PATH operator \( P_d^R \) and a streaming graph \( S \) at time \( t \), a spanning tree \( T_x \) forms a compact representation of valid path segments that are reachable from the vertex \( x \) in \( G_t \) under the constraints of a given RPQ, i.e., a vertex-state pair \((u,s)\) is in \( T_x \) at time \( t \) if there exists a path \( p \) in \( G_t \) from \( x \) to \( u \) with label \( \phi(p) \) such that \( s = s^*(s_0, \phi(p)) \).

A node \((u,s) \in T_x \) indicates that there is a path \( p \) in the snapshot graph with label \( \phi(p) \) such that \( s = s^*(s_0, \phi(p)) \), and this path can simply be constructed by following parent pointers \((u,s).pt \) in \( T_x \). Under the arbitrary path semantics, there are potentially infinitely many path segments between a pair of vertices that conform to a given RPQ due to the presence of cycles in the snapshot graph and a Kleene star in the given RPQ. Among those, S-PATH materializes the path segment with the longest expiry timestamp, that is, the path segment that will expire furthest in the future.
Consequently, for each node \((u, s) \in T_x\), the sequence of vertices in the path from the root node to \((u, s)\) corresponds to the path from \(x\) to \(u\) in the snapshot graph with the largest expiry timestamp. This is achieved by the coalesce primitive (Definition 11) with an aggregation function \(\text{max}\) over the expiry timestamp of path segments.\(^7\) Upon expiration of a node \((u, s) \in T_x\) and its corresponding path segment in the snapshot graph, this guarantees that there cannot be an alternative path segment between \(x\) and \(u\) that have not yet expired. Hence, we can directly find expired tuples based on their expiry timestamps. This is based on the observation that expiration has a temporal order unlike explicit deletions, and S-PATH utilizes these temporal patterns to simplify window maintenance.

Definition 22 (\(\Delta - \text{PATH INDEX}\)). Given an automaton \(A\) for the regular expression \(R\) of a \(\text{PATH}\) operator \(\mathcal{P}_R^D\) and a streaming graph \(S\) at time \(t\), \(\Delta - \text{PATH}\) is a collection of spanning trees (Definition 21) where each tree \(T_x\) is rooted at a vertex \(x \in G_t\) for which there is an sgt \(t \in S(t)\) with a label \(l\) such that \(\delta(s_l, l) \neq \emptyset\) and \(\text{src} = x\).

\(\Delta - \text{PATH}\) encodes a single entry for each pair of vertices under the constraints of a given query, consistent with the set semantics of snapshot graphs (Section 3). Due to spanning-tree construction (Definition 21), actual paths can easily be recovered by following the parent pointers; hence, \(\Delta - \text{PATH}\) constitutes a compact representation of intermediate results for path navigation queries over materialized path graphs. Our implementation models \(\Delta - \text{PATH}\) as a hash-based inverted index from vertex-state pairs to spanning trees, enabling quick look-up to locate all spanning trees that contain a particular vertex-state pair. Upon arrival of an sgt \(t = (u, v, l, \{ts, exp\}, D)\), Algorithm S-PATH probes this inverted index of \(\Delta - \text{PATH}\) to retrieve all path segments that can be extended with the incoming sgt, that is, spanning trees that have the node \((u, s)\) with an expiry timestamp smaller than \(ts\) for any state \(s \in \{\delta(s_l, l) \neq \emptyset\}\) (Line 14). If the target node \((u, s)\) for \(t = (s, l)\) is not in the spanning tree \(T_x\), Algorithm Expand is invoked to expand the existing path segment from \((x, 0)\) to \((u, s)\) with the node \((v, t)\) and to create a new leaf node as a child of \((u, s)\). In case there already exists a path segment between vertices \((x, 0)\) and \((v, t)\) in \(\Delta - \text{PATH}\), i.e., the target node \((v, t)\) is already in \(T_x\), Algorithm S-PATH compares its expiry timestamp with the new candidate (Line 18). If the extension of the existing path segment from \((x, 0)\) to \((u, s)\) with \((v, t)\) results in a larger expiry timestamp than \((v, t).\exp\), Algorithm Propagate is invoked to update the expiry timestamp of \((u, v, l, \{\text{ts}, \text{exp}\})\) and its children in \(T_x\). Algorithms Expand and Propagate traverse the snapshot graph until no further update is possible. The following example illustrates the behaviour of Algorithm S-PATH on our running example.

Example 9. Consider the real-time notification query as in Example 1 whose SGA expression is given in Example 8. Figure 9a shows an excerpt of the streaming graph input to the \(\text{PATH}\) operator \(\mathcal{P}_R^D\). Figure 9b depicts a spanning tree \(T_x\) in \(\Delta - \text{PATH}\) at \(t = 27\). Upon arrival of the sgt \((y, u, RL, \{28, 37\})\), \(D = \{(y, RL, u)\}\) at \(t = 28\), Algorithm S-PATH extends the path segment from \((x, 0)\) to \((y, 1)\) with \((u, 1)\), and compares its expiry timestamp with that of \((u, 1)\) that is already in \(T_x\). As the new extension has larger expiry timestamp, the validity interval

\[^7\]Arbitrary path semantics provides the flexibility for the aggregation function \(f_{agg}\) of the coalesce primitive.
determine the expired nodes based on the validity intervals (nodes (z, 1) and (t, 1) as shown in Figure 9c) without additional processing.

6.2.5 Explicit Deletions. Physical implementations of PATTERN and PATH rely on the direct approach that utilizes the temporal pattern of sliding windows for state maintenance; the expiration timestamps are used to directly locate expired tuples. On append-only streaming graphs, existing sgts only expire due to window movements. Albeit rare, certain applications might require explicit deletions of previously inserted sgts, which necessitates the use of the negative tuple approach to handle such explicit deletions.\(^8\) Deleted sgt, with an additional flag to denote deletion – i.e., a negative tuple, is used to undo the effect of the original sgt on the operator state and to invalidate previously reported results, if necessary. For pipelined hash join, which is used in the implementation of PATTERN, processing of negative tuples is the same as original input tuples: a negative tuple is removed from its corresponding hash table and other tables are probed to find corresponding deleted results.

The use of negative tuples for explicit deletions in the context of RPQ evaluation is first proposed by Pacaci et al \[57\]. Here, we describe how to adopt the negative tuple approach for explicit deletions of sgts in Algorithm S-PATH. In brief, upon explicit deletion of an sgt, we first identify tree-edges that disconnect spanning trees in \(\Delta - \text{PATH}\). For each such tree-edge, we first mark the nodes in the subtree that are disconnected due to the explicit deletion. Then, for each node in this subtree, we use a Dijkstra-based traversal over the snapshot graph to find an alternative path with the largest expiry timestamp. Dijkstra’s algorithm guarantees that we can efficiently find the path with the largest expiry timestamp for each marked node, consistent with Definition 22. A marked node is removed from the spanning tree only if there is no alternative valid path. Deletion of a non-tree edge does not require any modification as it leaves spanning trees unchanged.

7 EXPERIMENTAL ANALYSIS

Our objective is to demonstrate the feasibility of implementing a performant system that incorporates the algebraic framework we propose in this paper. Using the prototype implementation described in §6, we first provide an end-to-end performance analysis of our algebraic approach for persistent evaluation of streaming graph queries (§7.2). Then, we assess the scalability by varying the window size \(T\) and the slide interval \(\beta\) (§7.3). Finally, we highlight the benefits of the proposed SGA in exploring the rich plan space through transformation rules and demonstrate the potential performance improvements of exploring this plan space (§7.4).

7.1 Experimental Setup

7.1.1 Setup. Experiments are run on a Linux server with 32 physical cores and 256GB memory. For each query and configuration, we report the tail latency of each window slide, i.e., the total time to process all arriving and expired sgts upon window movement and to produce new results, and the average throughput after ten minutes of processing on warm caches.

7.1.2 Datasets. We use Stackoverflow (SO) and LDBC SNB (SNB) graphs for our experimental analysis; these are publicly available, large-scale graphs with labelled and timestamped edges on which persistent queries with complex graph patterns can be formulated. SO is a temporal graph of user interactions on the stackoverflow website containing 63M interactions (edges) of 2.2M users (vertices), spanning 8 years [59], and SNB is a synthetic social network graph that simulates the interactions of an online social network [22]. We extract the update stream of the LDBC workload that contains 8 different types of interactions, and we use replyOf, hasCreator and likes edges between users and posts, and knows edges between users. We use a scale factor of 10 with 7.2M users and posts (vertices) and 40M user interactions (edges). SO contains only a single type of vertex and 3 different edge labels, and its 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

---

\(^8\)The negative tuple approach can also be used to signal expirations through an explicit deletion of the corresponding sgt to undo its effect [25, 29, 57]. Indeed, the negative-tuple approach can be used for incremental evaluation of an arbitrary computation, whereas direct approach is applicable to negation-free queries over append-only streaming graphs [29].
Table 1: $Q_1 \sim Q_4$ correspond to common RPQ observed in real-world query logs [15], and $Q_5 \sim Q_7$ are Datalog encodings of RQ-based complex graph patterns that we use to define streaming graph queries. $Q_5$ and $Q_6$ correspond to complex graph patterns of LDBC SNB queries $IS7$ and $IC7$ [22], respectively, and $Q_7$ corresponds to the complex graph pattern given in Example 1 that is defined as a recursive path query over the graph pattern of $Q_6$. $a, b$ and $c$ correspond to edge predicates that are instantiated based on the dataset characteristics.

<table>
<thead>
<tr>
<th>Name</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q_1$</td>
<td>$?x,?y \leftarrow ?x\ a\ ?y$</td>
</tr>
<tr>
<td>$Q_2$</td>
<td>$?x,?y \leftarrow ?x\ a\ b^*\ ?y$</td>
</tr>
<tr>
<td>$Q_3$</td>
<td>$?x,?y \leftarrow ?x\ a\ b^<em>\ oc^</em>\ ?y$</td>
</tr>
<tr>
<td>$Q_4$</td>
<td>$RR(m1,m2) \leftarrow (a(x,y), b(m1,x), b(m2,y), c(m2,m1))$</td>
</tr>
<tr>
<td>$Q_6$</td>
<td>$RL(x,y) \leftarrow a^+(x,y), b(m,x), c(m,y)$</td>
</tr>
<tr>
<td>$Q_7$</td>
<td>$Ans(x,m) \leftarrow RL^+(x,y), (c,m,y)$</td>
</tr>
</tbody>
</table>

7.2 Query Processing Performance

7.2.1 Throughput & Tail Latency. Table 2 (SGA) shows the aggregated throughput and tail latency of our streaming graph query processor for all queries in Table 1. We discard each streaming graph edge whose label is not in a given SGQ. Tail latencies reflect the 99th percentile latency of processing a window slide and produce the corresponding resulting sgts. Across queries, the performance is lower for SO graph because it is dense and cyclic. The throughput ranges from hundreds of edges-per-second for the SO to hundreds of thousands of edges-per-second for SNB.

7.2.2 Comparative Analysis. Existing work on query processing over streaming data such as data stream management systems and streaming RDF systems cannot process queries in Table 1 as they focus on relational queries and SPARQL v1.0, respectively (§1). To the best of our knowledge, TD with its DD layer is the only general-purpose system that can be used to incrementally evaluate recursive computations that are modelled as cyclic dataflows. Table 2 (DD) reports the throughput and tail latency of DD dataflows for all queries in Table 1. Overall, our SGA-based query processor outperforms the DD baseline on SO and provides a competitive performance on the SNB dataset. On SNB, $Q_6$ & $Q_7$ do not have the Kleene-plus over $a$ as it causes DD to timeout. Due to highly cyclic structure of SO, there are many alternative paths between each pair of vertices, and our streaming RPQ algorithm for PATH implementation maintains a compact representation of valid path segments and utilizes the temporal patterns of sliding window movements to simplify expirations (§6.2). DD-based query processor provides better performance on linear path queries $Q_1 \sim Q_4$ on SNB, but not others. This is due to the tree-shaped structure of replyOf edges in SNB, where there is only one path between a pair of vertices, so PATH specific optimizations do not apply. Performance variations on SNB suggest optimization opportunities for recursive graph queries when selecting physical operator implementations, as in the case for streaming relational joins [29]. These results demonstrate the feasibility of our algebraic approach for evaluating SGQ and our physical operator implementations.

7.3 Sensitivity Analysis

In this section, we analyze the impact of the window size $T$ and the slide interval $\beta$ on end-to-end query performance of the proposed streaming graph query processor. We use SO graph for this experiment as its dense, cyclic structure stresses our operator implementations. Fig. 10a reports the aggregate throughput and the tail latency for each query across various window sizes. As expected, the throughput of all tested queries decreases with increasing $T$, as a larger window size increases the # of sgts in each window. Similarly, the tail latency of each window slide increases with the increasing window size.

We also assess the impact of the slide interval $\beta$ on performance. As previously mentioned, the slide interval $\beta$ controls the time granularity at which the sliding window progresses, and our prototype implementation uses $\beta$ to control the input batch size. Figure 10b shows that the aggregate throughput and the tail latency for each query remain stable across varying slide intervals. This is due to tuple-oriented implementation of physical operators of SGA; SGA operators are designed to process each incoming tuple eagerly in favour of minimizing tuple-processing latency, and they do not
Table 2: (Tput) The throughput (edges/s) and (TL) the tail latency (s) of SGA and DD for queries in Table 1 on SO and SNB graphs with $|W|=30$ days and $\beta=1$ day.

<table>
<thead>
<tr>
<th>Graph</th>
<th>System</th>
<th>$Q_1$ Tput TL</th>
<th>$Q_2$ Tput TL</th>
<th>$Q_3$ Tput TL</th>
<th>$Q_4$ Tput TL</th>
<th>$Q_5$ Tput TL</th>
<th>$Q_6$ Tput TL</th>
<th>$Q_7$ Tput TL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SO</td>
<td>SGA</td>
<td>2762 3.3</td>
<td>8513 4.3</td>
<td>413 120</td>
<td>379 102.4</td>
<td>231064 0.3</td>
<td>374 52.7</td>
<td>376 56.3</td>
</tr>
<tr>
<td></td>
<td>DD</td>
<td>1209 6.3</td>
<td>4512 5.8</td>
<td>368 121.7</td>
<td>374 82.8</td>
<td>63330 1</td>
<td>283 72.6</td>
<td>275 74</td>
</tr>
<tr>
<td>SNB</td>
<td>SGA</td>
<td>97187 1.5</td>
<td>237313 1.9</td>
<td>245766 1.9</td>
<td>277475 0.4</td>
<td>13345 79.1</td>
<td>428592 0.8</td>
<td>131250 10.2</td>
</tr>
<tr>
<td></td>
<td>DD</td>
<td>121133 0.8</td>
<td>299245 1.2</td>
<td>316267 1.1</td>
<td>303068 0.2</td>
<td>12053 109.5</td>
<td>402048 0.9</td>
<td>21284 141</td>
</tr>
</tbody>
</table>

Figure 10: The tail latency of each window slide and the aggregate throughput of the proposed streaming graph query processor with increasing (a) window size $\tau$ and (b) slide interval $\beta$ on SO graph.

Figure 11: The tail latency of each window slide and the aggregate throughput of SGA evaluation on DD with increasing slide interval $\beta$ on SO graph.

7.4 Exploring the Plan Space

SGA proposed in this paper enables a rich foundation for logical SGQ optimization through query rewriting as previously discussed (§5.4). In this paper, we do not address the full scope of optimization issues (full optimizer development is the topic of ongoing research), but we design a micro-benchmark to highlight the possibilities provided by SGA. In particular, we choose $Q_4$ (Table 1) as its linear pattern combined with a Kleene plus demonstrates the potential benefits of SGA transformation rules involving SGA’s novel PATH operator. Fig. 12 demonstrates the throughput and the tail latency of different plans obtained from following equivalent SGA expressions for $Q_4$:
These results suggest further optimization opportunities for logical iteration over the base pattern as query rewrites that are generated by SGA \( Q \) and 60% reduction in the latency. We observe a similar behaviour on due to novel \( \text{PATH} \) rules given in §5.4, and represent novel plans that are possible obtained from the canonical SGA expression using the transformation rules (Section 5.4).

Figure 13: The throughput and tail latency of (a) \( Q_2 \) on (top) SO and (bottom) SNB for the default SGA plan and an alternative equivalent physical plan generated via SGA transformation rules (Section 5.4).

Figure 14: The throughput and tail latency of (a) \( Q_3 \) on (top) SO and (bottom) SNB for the default SGA plan and an alternative equivalent physical plan generated via SGA transformation rules (Section 5.4).

- SGA: \( P_1^{d\text{a}} (\star_{t_{\text{src}}=\text{src}_1,t_{\text{trg}}=\text{trg}_1=\text{src}_2} (S_{\text{a}},S_{\text{b}},S_{\text{c}})) \)
- P1: \( P_1^{l\text{(a\text{-}b\text{-}c)}} (S_{\text{a}},S_{\text{b}},S_{\text{c}}) \)
- P2: \( P_1^{l\text{(a\text{-}d)}} (S_{\text{a}},\star_{t_{\text{src}}=\text{src}_1,t_{\text{trg}}=\text{trg}_2=\text{src}_2} (S_{\text{b}},S_{\text{c}})) \)
- P3: \( P_1^{l\text{(d\text{-}c)}} (S_{\text{c}},\star_{t_{\text{src}}=\text{src}_1,t_{\text{trg}}=\text{trg}_2} (S_{\text{a}},S_{\text{b}})) \)

The first expression, SGA, is the canonical SGA expression for \( Q_4 \) that is generated by the Algorithm SGQParser. Such plans are called \( \text{loop-caching} \) in literature as they enable re-use of the intermediate results for the base pattern \((a \cdot b \cdot c)\) \( [73] \). \( P_1, P_2 \) and \( P_3 \) are obtained from the canonical SGA expression using the transformation rules given in §5.4, and represent novel plans that are possible due to novel \( \text{PATH} \) operator. Fig. 12 clearly illustrates the potential benefits of exploring the rich plan space offered by SGA: some of the newly computed plans provide up to 60% increase in throughput and 60% reduction in the latency. We observe a similar behaviour on other path queries \( Q_2 \) and \( Q_3 \) (up to 50% difference in throughput). These results suggest further optimization opportunities for logical query optimization as query rewrites that are generated by SGA transformation rules can provide significant performance benefits for evaluating SGQ over streaming graphs.

7.5 Alternative Physical Operators

Our prototype implementation employs the automata-based streaming RPQ algorithm of \([57]\) as the physical implementation of the \( \text{PATH} \) operator. Section 6.2.4 describes an alternative physical implementation (\( \text{Streaming Path Navigation – S-PATH} \)) for the novel \( \text{PATH} \) operator of our proposed SGA based on the direct approach. Table 3 compares its impact on the performance of our proposed streaming graph query processor. The performance impact of adopting the Algorithm \( \text{S-PATH} \) for \( \text{PATH} \) varies depending on the workload characteristics: it provides improvements in throughput for most queries on the SO graph, whereas the performance differences on the SNB graph is small. This can be attributed to the cyclic nature of the SO graph. A higher number of intermediate path segments increases the size of the internal operator state that the physical implementations of the \( \text{PATH} \) operator need to maintain. Overall, these results illustrate the potential benefits of exploring alternative operator implementations. In particular, cost-based optimization techniques for determining suitable operator implementations based on data and query characteristics might provide significant performance benefits for evaluating SGQ over streaming graphs.

8 CONCLUSION AND FUTURE WORK

This paper introduces a general-purpose query processing framework for streaming graphs that consists of (i) streaming graph query model and algebra with well-founded semantics, and (ii) a prototype streaming graph query processor as an embodiment of the proposed framework. The SGQ model addresses the requirements we set forth for querying streaming graphs by capturing both common features of existing graph query languages and novel features that are beyond the power of existing languages (such as the treatment of paths as first-class citizens, powerful path patterns and composability). SGA and its transformation rules provide the foundational framework to express and evaluate streaming graph queries. Our prototype streaming graph query processor exemplifies physical operator implementations specific to streaming graph queries due to this framework. Experimental analyses on real-world and synthetic streaming graphs demonstrate the feasibility and the potential performance gains of our approach.

SGQ and SGA establish the basis of systematic study of query processing issues over streaming graphs. We are now engaged in research along two dimensions: (i) designing an SGA-based query optimizer for the systematic exploration of the rich plan space using SGA’s transformation rules, and (ii) incorporating attribute-based predicates to fully support the property graph model.

Additional work that would enrich the implementation include the development of alternative physical operators and additional transformation rules for plan space enumeration. Given that the SGQ-SGA framework is complete with a clear semantics (as captured in RQ model), and given that SGQ incorporates the capabilities of existing and emerging query languages, future work can focus on efficient implementation.
<table>
<thead>
<tr>
<th>Graph</th>
<th>System</th>
<th>$Q_1$ Tput TL</th>
<th>$Q_2$ Tput TL</th>
<th>$Q_3$ Tput TL</th>
<th>$Q_1$ Tput TL</th>
<th>$Q_2$ Tput TL</th>
<th>$Q_3$ Tput TL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SO</td>
<td>S-PATH Improvement</td>
<td>2884.4</td>
<td>9074.4</td>
<td>391.4</td>
<td>348.9</td>
<td>234058.4</td>
<td>625.514</td>
</tr>
<tr>
<td>SNB</td>
<td>S-PATH Improvement</td>
<td>9590.3</td>
<td>14465.3</td>
<td>1.8</td>
<td>224342.9</td>
<td>0.4</td>
<td>14000.7</td>
</tr>
</tbody>
</table>

Table 3: The impact of S-PATH on the performance of the throughput (edges/s) and (TL) the tail latency (s) of the queries in Table 1 on SO and SNB graphs with $|W| = 30$ days and $\beta = 1$ day.


