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

In this paper, 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 model and a streaming graph 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 based on the dataflow computational model shows the feasibility of our approach and allows us to gauge the high performance gains obtained for query processing over streaming graphs.

1 INTRODUCTION

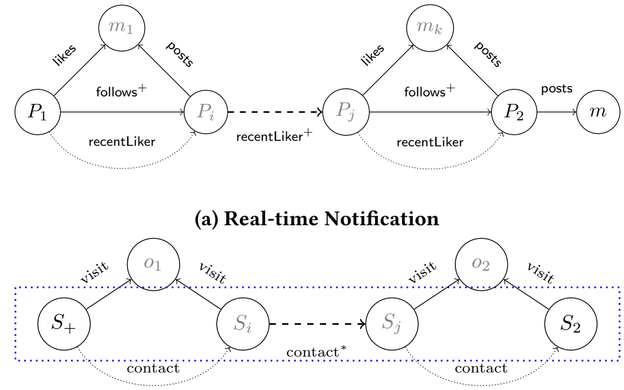
Modern applications in many domains now operate on very high speed streaming graphs. For example, Twitter's recommendation system ingests 12K events/sec on average [34], Alibaba transaction graph processes 30K edges/sec at its peak [65]. A recent survey [67] reports that these workloads are prevalent in real applications, and efficient querying of these streaming graphs is a crucial task for applications that monitor complex patterns and relationships.

Existing graph DBMSs 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 to 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 as demonstrated in the following two real examples.

Example 1 (Real-time Notification). In many online social networking applications users post original content, sometimes link this to other people's contents and react each other's posts – these interactions are modeled as a graph. We say that a user u_2 is a recent liker for another user u_1 if u_2 recently likes posts that are created by u_1 and u_2 and u_1 are connected through friends. The goal of the recommendation service is to notify users, in real-time, of new contents that are posted by others that are connected by a path of recent liker pattern. This real-time notification task is a persistent query over the streaming graph of user interactions that returns the recommended content in real-time, as depicted in Figure 1a.

Example 2 (Physical Contact Tracking). There are a number of Covid-19 contact tracing applications that model interactions as graphs.[1] In these applications, people are represented as vertices

[1]https://www.datanami.com/2020/03/12/tracking-the-spread-of-coronavirus-with-graph-databases/



(a) Real-time Notification

(b) Contact Tracing

Figure 1: Graph patterns representing queries in (a) Example 1, and (b) Example 2.

and two people are said to be in contact if they visit the same space in the last 14 days (this is a simplification). The goal is to notify people with a potential chain of contacts when they are linked to someone who tested positive. As shown in Figure 1b, the task of contact tracing is also a persistent graph query that returns the chain of contacts on a sliding window over this streaming graph of people's contacts.

As demonstrated by these examples, real-world applications that feature complex graph patterns require:

- **R1:** subgraph queries that find matches of a given graph pattern;
- **R2:** path navigation queries that traverse paths based on user specified constraints; and
- **R3:** the ability to manipulate and return paths (i.e., treat paths as first-class citizens of the data model).

These requirements are only partially addressed by existing graph DBMSs in the context of one-time queries over static graphs. Query languages of these graph DBMSs (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) [16, 78]. However, UCRPQ lacks algebraic closure and composability, limiting optimization opportunities.[2] 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. Consequently, UCRPQ-based query languages limit path navigation queries to boolean

[2]Relational algebra is closed over a set of relational operators and common relational techniques such as query rewriting and view-based query optimization utilize the algebraic closure of operators of the relational algebra.

reachability, lacking the ability to return and manipulate paths. G-CORE [7] addresses these limitations at the language level and introduces a query language proposal that heavily influences the standardization efforts for a query language for graph DBMSs.[3] To the best of our knowledge, there is no work that uniformly addresses all three requirements in the context of persistent query processing over streaming graphs. A number of specialized algorithms focus on evaluating subgraph pattern queries on streaming graphs [6, 21, 40, 49, 65], and our previous work focuses on path navigation queries and introduces first streaming algorithms for RPQ evaluation over streaming graphs [62]. Our objective in this paper is to introduce a general-purpose framework that addresses the above discussed requirements of real-world applications (which feature complex graph patterns) in a uniform and principled manner.

Querying streaming data in real-time imposes novel requirements in addition to challenges of graph processing:

- **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.

Most existing work on graph querying employ the *snapshot* model, which assumes that graphs are static and fully available, and that ad hoc queries reflect the current state of the database (e.g., [23, 43, 70, 72, 76, 79, 81]). Our experiments show that graph DBMSs that are based on the snapshot model are not able to keep up with the high arrival rates [63]. The unsuitability of existing graph DBMSs for querying streaming data have motivated the design of specialized applications that process streaming graphs (e.g., [21, 34, 40, 49, 65]). However, due to the lack of a principled approach that has benefitted relational DBMSs, these applications rely on ad hoc solutions that are tailored for a particular task. 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 introduce a general-purpose query processing framework for streaming graphs that consists of: (i) streaming graph model and algebra with well-founded semantics, and (ii) dataflow-based query processor with efficient physical operators for persistent query processing over streaming graphs. 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. Built on the Regular Query (RQ) model[4], our proposed *Streaming Graph Algebra* (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 proposed framework treats paths as first-class citizens of its data model, enabling the proposed algebra to express queries that return and manipulate paths (**R3**). Yet, such representation with well-founded semantics is only half of the answer. We also provide a prototype system on top of the *Timely Dataflow* [59] for efficient evaluation of these queries based on our proposed SGA. In particular, we first introduce efficient incremental algorithms that utilize temporal patterns of sliding window movements over the streaming graph as physical operators for SGA primitives (**R4** & **R5**), then describe how to cast persistent graph queries as dataflow computations that consist of these physical operators. Ours, to the best of our knowledge, is the first work to precisely describe semantics of persistent query evaluation over streaming graphs that combines subgraph pattern and path navigation queries in a principled manner with the notion of paths as first-class citizens, and to provide efficient algorithms to evaluate such queries consistent with the semantics. Our contributions are:

- a streaming graph model (Section 3), a query model (Section 4), and a streaming graph algebra (Section 5.1) providing precise semantics of persistent graph queries with subgraph patterns, path navigations and windowing constructs;
- incorporation of paths as first-class citizens, enabling the algebra to express queries that return and manipulate paths;
- development of algebraic transformation rules for the SGA and a query formulation and plan generation methodology;
- proposing physical operators for the proposed SGA (Section 6) and a streaming graph query processor based on the dataflow computational model (Section 7.1.1); and
- extensive experimental analysis on real and synthetic datasets to demonstrate the feasibility and the performance of our approach for query processing over streaming graphs (Section 7).

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 [10], Aurora [3], Borealis [2]). In contrast, modern Data Stream Processing Systems (DSPS) such as Storm [73], Heron [45], Flink [20] do not necessarily offer a full set of DBMS functionality. Existing literature (as surveyed by Hirzel et al. [38]) heavily focus on general-purpose systems and do not consider core graph querying functionality such as *subgraph pattern matching* and *path navigation*. Similarly, there has been a significant interest in RDF streams (e.g., stream reasoning [9, 42], data exchange [18, 54]).[5] Some of the streaming RDF systems include SPARQL extensions for persistent query evaluation over RDF streams such as C-SPARQL [14], CQELS [48], SPARQL$_{stream}$ [19] and W3C proposal RSP-QL [24]. These are the most similar to our problem setting. However, these RDF systems are designed for SPARQLv1.0 and cannot formulate path expressions such as RPQs that cover more than 99% of all recursive queries abundantly found in massive Wikidata query logs [17]. Furthermore, query processing engines of these systems do not employ incremental operators, except Sparkwave [42] 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 that are optimized for streaming workloads.

Existing work on streaming graph systems, by and large, focus on either (i) maintenance of graph snapshots under a stream

[3] See https://www.gqlstandards.org/.

[4] A computationally well-behaved subset of Datalog that has tractable query evaluation and decidable query containment, similar to relational algebra [66].

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

of updates for iterative graph analytics workloads, or (ii) specialized systems for persistent query workloads that are tailored for the task at hand. One of the earlier systems in the first category, STINGER [25], proposes an adjacency list-based data structure optimized for fast ingestion of streaming graphs. GraphOne [46, 47] uses a novel versioning scheme to support concurrent reads and writes on the most recent snapshot of the graph. Analytical engines such as GraphIn [69] and GraphTau [39] extend the popular vertex-centric model with incremental computation primitives to minimize redundant computation across consecutive snapshots. More recently, systems such as GraPu [71] and GraphBolt [53] introduce novel dependency tracking schemes to transparently maintain results of graph analytics workloads by utilizing structural properties such as monotonicity. This line of research primarily focuses on building and maintaining graph snapshots from streaming graphs for iterative graph analytics workloads with little or no focus on graph querying functionalities.

2.2 Incremental View Maintenance

A persistent query over sliding windows can be formulated as an Incremental View Maintenance (IVM) problem, where the view definition is the query itself and window movements correspond to updates to the underlying database. In the IVM model, the goal is to incrementally maintain the view – results of a persistent query – upon changes to the underlying database – insertions (expirations) into (from) a sliding window. The classical *Counting* [35] algorithm maintains the number of alternative derivations for each derived tuple in a Select-Project-Join view to determine when a tuple no longer belongs to the view. DBToaster [41] introduces the concept of *higher-order* views for group-by aggregates and represents each view definition using a hierarchy of views that reduce the overall maintenance cost. F-IVM [61] further extends higher-order views with a factorized representation of these views to reduce the amount of state and the computation cost. ViewDF [80] extends existing IVM techniques with windowing constructs to speed up query processing over sliding windows. Although conceptually similar, these techniques are not suitable for recursive graph queries that are addressed in this paper, primarily because of the potentially infinite results for recursive graph queries.

The classical *DRed* algorithm [35] adapts the *semi-naive* strategy to support recursive views: it first deletes all derived tuples that depend on the deleted tuple, then re-derives the tuples that still have an alternative derivation after the deletion. DRed might over-estimate the set of deleted tuples and might re-derive the entire view. Storing the *how*-provenance – the set of all tuples that might be used to derive a tuple – might prevent over-estimation; however, it significantly increases the amount of state that the algorithm needs to maintain. The provenance information can be encoded in the form of boolean polynomials and the boolean absorption law can be used to reduce the amount of additional information that needs to be maintained [52]. Thus, it is possible to adapt recursive IVM techniques to evaluate streaming graph queries, but they ignore the structure of graph queries and inherent temporal patterns of streaming graphs. Our methods, in contrast, exploit the structure of the problem to minimize the cost of persistent graph query evaluation over streaming graphs.

2.3 Graph Algorithms

Temporal Graph Algebra (TGA) [58] adapts temporal relational operators in PGM to provide systematic support for analytics over evolving graphs. Its implementation on Spark introduces physical operators for graph analytics at different levels of resolutions [5]. Similar to ours, TGA introduces a set of *time-aware* algebraic primitives, but, it focuses on exploratory graph analytics over the entire history of changes. In contrast, the primary objective of our framework is to continuously evaluate graph queries as the underlying (potentially unbounded) graph changes, i.e., persistent query processing over streaming graphs.

The closest to ours are specialized incremental algorithms on dynamic and streaming graphs [6, 21, 40, 49, 62, 65]. Some of these [21, 40] study the problem of subgraph pattern matching under a stream of edge updates. Their focus is developing efficient incremental algorithms to maintain matches of a given subgraph pattern as the underlying graph changes. Similarly, Ammar et al. [6] present worst-case-optimal join-based algorithms for distributed evaluation of subgraph pattern queries. Li et al. [49] present an efficient algorithm for subgraph isomorphism search over streaming graphs with timing-order constraints. GraphS [65] introduces efficient index structures that are optimized for cycle detection queries. These are all specialized algorithms and systems for particular query workloads. In previous work [62], we study the design space of algorithms for path navigation over streaming graphs and provide algorithms for persistent evaluation of Regular Path Queries, a tiny subset of the class of queries that we address in this paper. To the best of our knowledge, this is the first work to describe the precise semantics of generalized streaming graph queries. We provide a unified framework to represent and optimize persistent graph queries over streaming graphs featuring both path navigation and subgraph pattern matching as well as windowing constructs that are commonly used in practice.

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, Σ is a set of labels, $\psi : E \longrightarrow V \times V$ is an incidence function and $\phi : E \longrightarrow \Sigma$ 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 $u \xrightarrow{p} v : \langle e_1, \cdots, e_n \rangle$. 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 $\mathbb{T} = (\mathcal{T}, \leq)$ to define a discrete, total ordered time domain and use timestamps $t \in \mathcal{T}$ 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 (src, trg, l, t) where $src, trg \in V$ are end-points of an edge $e \in E$, $l \in \Sigma$ is the label of the edge e, and*

$t \in \mathcal{T}$ *is the event (application) timestamp assigned by the source, i.e.,* $\psi(e) = (src, trg)$ *and* $\phi(e) = l$. [6]

Definition 4 (Input Graph Stream). *An input graph stream is a continuously growing sequence of streaming graph edges* $S^I = [sge_1, sge_2, \cdots]$ *where each sge represents an edge in graph G and sges are non-decreasingly ordered by their timestamps.*[7]

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* $t \in \mathcal{T}$ *for which* $ts \leq t < exp$.

Timestamps are commonly used to represent the time instant in which the interaction represented by the sge occured [49, 62, 65]. 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.[8] 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 $u \xrightarrow{p} v : \langle e_1, \cdots, e_n \rangle$ 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* $G = (V, E, P, \Sigma, \psi, \rho, \phi)$ *where V is a set of vertices, E is a set of edges, P is a set of paths,* Σ *is a set of labels,* $\psi : E \rightarrow V \times V$ *is an incidence function,* $\rho : P \rightarrow E \times \cdots \times E$ *is a total function that 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* ϕ *are disjoint, i.e.,* $\phi(E) \cap \phi(P) = \emptyset$.

[6]We assume that sges are generated by a single source and arrive in order, and leave out-of-order arrival as future work.
[7]We use [] to denote ordered streams throughout the paper
[8]Commonly referred as `NOW` windows as described in Section 5.1.

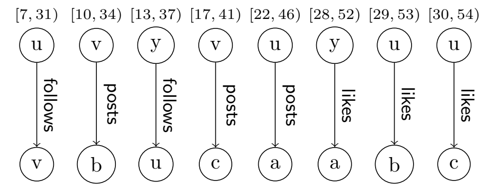


Figure 2: An excerpt of the streaming graph of the real-time notification query in Example 1.

The function ρ assigns to each $p : u \xrightarrow{p} v \in P$ an actual path $\langle e_1, \cdots, e_n \rangle$ in graph G satisfying: for every $i \in [1, \cdots, n)$, $\psi(e_i) = (src_i, trg_i)$, $trg_i = src_{i+1}$, and $src_1 = u, trg_n = v$. 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* $sgt = (src, trg, l, [ts, exp), \mathcal{D})$ *where distinguished (explicit) attributes* $src, trg \in V$ *are endpoints of an edge* $e \in E$ *or a path* $p \in P$ *in graph G and* $l \in \Sigma$ *is its label, and non-distinguished (implicit) attributes* $[ts, exp) \in \mathcal{T} \times \mathcal{T}$ *is a half-open time-interval representing t's validity and* $\mathcal{D}$ *is a payload consists of edges in E that participated in the generation of the tuple t.*

Streaming graph tuples generalize sges (Definition 3) to represent, 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 $E^I \subset E$ to denote the set of input graph edges, and $\phi(E^I)$ to denote the fixed set of labels that are reserved for input graph edges. Additionally, non-distinguished (implicit) attribute $\mathcal{D}$ of an sgt t captures the path p, i.e., sequence of edges, in case the sgt t represents a path. Otherwise, $\mathcal{D}$ is the edge e that the sgt t represents.

Definition 8 (Streaming Graph). *A streaming graph S is a continuously 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 *(*$t_i < t_j$ *for* $i < j$*).*

Figure 2 depicts an excerpt of the streaming graph of the social networking application given in Example 1, where sgts represent input graph edges (validity intervals are assigned by the windowing operator as explained in Section 5.1). The input graph edge from u to v with label *likes* is modelled as the sgt $t = (u, a, likes, [13, 37)), \mathcal{D} = \{(u, likes, a)\})$.

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 sgt [44]. This corresponds to the *negative tuple* approach [29, 33]. Section 6 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 $\{S_{l_1}, \cdots, S_{l_n}\}$ where each S_{l_i} consists of sgts of S with the label l_i, i.e., $S = \bigcup_{l \in \Sigma}(S_l)$*

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). *Sgts $t_1 = (u_1, v_1, l_1, [ts_1, exp_1), \mathcal{D}_1)$ and $t_2 = (u_2, v_2, l_2, [ts_2, exp_2), \mathcal{D}_2)$ are value-equivalent iff their distinguished attributes are equal, i.e., they both 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 [51]. We extend the *coalesce* primitive from temporal database literature [22] to sgts with an aggregation function over the non-distinguished payload attribute, $\mathcal{D}$, as shown below:

Definition 11 (Coalesce Primitive). *The coalesce primitive merges a set of value-equivalent sgts $\{t_1, \cdots, t_n\}$, $t_i = (src, trg, l, [ts_i, exp_i), \mathcal{D}_i)$ for $1 \le i \le n$ with overlapping or adjacent validity intervals using an operator-specific aggregation function f_{agg} over the payload attribute $\mathcal{D}$:*

$$coalesce_{f_{agg}}(\{t_1 \cdots, t_n\}) =$$
$$(src, trg, l, [\min_{1 \le i \le n}(ts_i), \max_{1 \le i \le n}(exp_i)), f_{agg}(\mathcal{D}_1, \cdots, \mathcal{D}_n))$$

Distinguished attributes src, trg and the label l of sgts in a streaming graph S represent the topology of a materialized path graph. Hence, a finite subset of a streaming graph S corresponds to a materialized path graph over the set of edges and paths that are in the streaming graph and the set of vertices that are adjacent to these. We now use this to define snapshot graphs and the property of *snapshot reducibility.*

Definition 12 (Snapshot Graph). *A snapshot graph G_t of a streaming graph S is defined by a mapping from each time instant in $\mathcal{T}$ to a finite set of sgts in S. At any given time t, the content of a mapping $\tau_t(S)$ defines a snapshot graph $G_t = (V_t, E_t, P_t, \Sigma_t, \psi, \rho, \phi)$ where $E_t = \{e_i \mid e_i.ts \le t < e_i.exp\}$ is the set of all edges that are valid at time t, $P_t = \{p_i \mid p_i.ts \le t < p_i.exp\}$, and V_t is the set of all vertices that are endpoints of edges and paths in E_t and P_t, respectively.*

Value-equivalence (Definition 10) and the coalesce primitive (Definition 11) ensure that snapshot graphs have the *set semantics*, i.e., at any point in time t, the snapshot graph G_t of a streaming graph S, a vertex, edge and path exists at most once.

Next, we define the notion of snapshot reducibility that enables us to precisely define the semantics of streaming queries and operators using their non-streaming counterparts. Snapshot reducibility is a commonly used concept in temporal databases to generalize non-temporal queries and operators operators to temporal ones [22].

Definition 13 (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 corresponding snapshots of S, i.e., $\forall t \in \mathcal{T}, \tau_t(Q(S)) = Q^O(\tau_t(S))$.*

4 STREAMING GRAPH QUERY MODEL

This section presents our streaming graph query (SGQ) model. We formulate streaming graph queries based on a streaming generalization of the Regular Query (RQ) model. RQ provides a good basis for building a general-purpose framework for persistent query evaluation over streaming graphs, because (i) unlike UCRPQ, it is closed under transitive closure and composable, (ii) it has the expressive power as the existing graph query languages such as SPARQL v1.1, Cypher, PGQL (RQ strictly subsumes UCRPQ on which these are based), and (iii) its query evaluation and containment complexity is reasonable [66]. Due to its well-defined semantics and computational behaviour, RQ is gaining popularity as a logical foundation for graph queries, both in theory [15, 16] and in practice [7].

Definition 14 (Regular Queries (RQ) – Following [66]). *The class of Regular Queries is the subset of non-recursive Datalog with a finite set of rules where each rule is of the form:*[9]

$$head \leftarrow body_1, \cdots, body_n$$

Each $body_i$ is either (i) a binary predicate $l(src, trg)$ where $l \in \Sigma$ is a label, or (ii) $(l^*(src, trg)$ as $d)$, which is a transitive closure over $l(src, trg)$ for a labels $l \in \Sigma, d \in \Sigma \setminus \phi(E)$, and each head predicate (head) is a binary predicate with $d(src, trg)$ for a label $d \in \Sigma \setminus \phi(E)$ except the reserved predicate Answer $\notin \Sigma$ of an arbitrary arity (not necessarily binary).

In other words, an RQ is a binary, non-recursive Datalog program extended with the transitive closure of binary predicates where input graph edges with a label $l \in \phi(E^I)$ corresponds to instances of the extensional schema (EDB) and derived edges and paths with a label $l \in \Sigma \setminus \phi(E^I)$ correspond to instances of the intensional schema (IDB). EDBs are predicates that appear only on the right-hand-side of the rules, which correspond to stored relations in Datalog [4]. Similarly, we define IDBs as predicates that appear in the rule heads, which correspond to output relations in Datalog.

Example 3 (Regular Query). Consider the real-time notification query in Example 1 and its graph pattern given in Figure 1a. The one-time query for the same graph pattern is represented as the following RQ:

$$RL(u_1, u_2) \leftarrow l(u_1, m_1), f(u_1, u2), p(u_2, m_1)$$
$$Notify(u, m) \leftarrow RL^+(u, u_2) as RLP, p(u_2, m)$$
$$Answer(u, m) \leftarrow Notify(u, m)$$

where predicates l, f, p, RL, RLP represent labels likes, follows, post, recent liker and recent liker path, respectively.

[9]We say a Datalog program is *recursive* iff its dependency graph contains cycles. The *dependency graph* of a Datalog program is a directed graph whose vertices are predicates of its rules and edges represent dependencies between predicates, i.e., there is an edge from p to q if q appears in the body of rule with head predicate p.

In the following, we define the semantics of SGQ using snapshot reducibility. It is known that for many operations such as joins and aggregation, exact results cannot be computed with a finite memory over unbounded streams [11]. 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 highly desirable. 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. Additionally, as opposed to streaming approximation techniques that trade off exact answer in favour of bounding the space requirements, window-based query evaluation enables us to provide exact query answers w.r.t. window specifications. Hence, we adopt the *time-based sliding window* for streaming graph queries in the rest of the paper.

Definition 15 (Streaming Graph Query (SGQ)). *A streaming graph query Q is a Regular Query defined over a streaming graph S and a time-based sliding window $\mathcal{W}_{\mathcal{T}}$. We define the semantics of an SGQ Q using the corresponding, one-time Regular query Q^O and the notion of snapshot reducibility (Definition 13):*

$$\forall t \in \mathcal{T}, \quad \tau_t\big(Q(S, \mathcal{W}_{\mathcal{T}})\big) = Q^O\Big(\tau_t\big(\mathcal{W}_{\mathcal{T}}(S)\big)\Big)$$

In other words, the snapshot of the resulting streaming graph of an SGQ Q at time t, i.e., $\tau_t\big(Q(S, \mathcal{W}_{\mathcal{T}})\big)$, is equivalent to the result of the corresponding one-time query Q^O over the snapshot of the input streaming graph at time t, i.e., $Q^O\Big(\tau_t\big(\mathcal{W}_{\mathcal{T}}(S)\big)\Big)$. Consequently, the output streaming graph of an SGQ can be obtained from the sequence of snapshots that is the result of a 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 of SGQ $\tau_t\ big(Q(S, \mathcal{W}_{\mathcal{T}})\big)$ if there is an edge $e = (u, v, l)$ or a path $p : u \xrightarrow{p} v$ with $l = \phi^p(p)$ in the resulting snapshot graph of the corresponding one-time query $G_{t_i} = Q^O\Big(\tau_t\big(\mathcal{W}_{\mathcal{T}}(S)\big)\Big)$ for $ts \leq t < exp$.

The specification of an SGQ is built upon that of one-time RQ, with an additional window specification that slides over the input graph stream. For instance, the SGQ for the real-time notification application in Example 1 is defined as the RQ given in Example 3 with a window specification of 24 hours over the logical partitioning of its input graph stream $S = S^I_{likes} \cup S^I_{posts} \cup S^I_{follows}$.

5 STREAMING GRAPH ALGEBRA

In this section, we present 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). We subsequently describe how to transform a given SGQ (Definition 15) into its canonical SGA expression and illustrate logical query plans (Section 5.2) and argue about its closedness and composibility (Section 5.3). Finally, we present transformation rules holding in our algebra (Section 5.4).

5.1 SGA Operators

Inputs to each SGA operator are one or more logically partitioned streaming graphs S_a (Definition 9) where each S_a contains sgts that

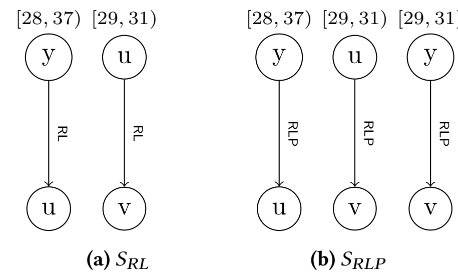


Figure 3: (a) Resulting streaming graph of a subgraph pattern operator for the diamond graph pattern of Example 1 on the streaming graph given in Figure 2, and (b) resulting streaming graph of a path navigation operator for the same query over the streaming graph given in Figure 3a.

represent edges or paths with the same label $a \in \Sigma$. The output of each operator is also a streaming graph S_o where each sgt has the label $o \in \Sigma \setminus \phi(E^I)$.[10] In the remainder, we formally and precisely define the semantics of each operator. The possible physical implementations of these operators are discussed in Section 6. 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** $\mathcal{W}$ 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 $\mathcal{T}$ and the optional slide interval β, i.e., $\mathcal{W}_{\mathcal{T},\beta}(S^I) := S : \big[(u, v, l, [t, exp), \mathcal{D} : e(u, v, l)) \mid (u, v, l, t) \in S^I \wedge exp = \lfloor t/\beta \rfloor \cdot \beta + \mathcal{T}\big]$.*

The window size $\mathcal{T}$ determines the length of the validity interval of sgts and the slide interval β controls the granularity at which the time-based sliding window progress, as defined in [10, 62]. In case β is not provided, it can be treated as $\beta = 1$, i.e., single time instant with the smallest granularity, and it defines a sliding window that progress every time instant.

WSCAN operator precisely defines the semantics of time-based sliding windows. Informally, WSCAN act as an interface between the source and the query plans and it is responsible for providing data from input graph streams to the system, 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, thus they generalize their non-streaming counterparts with implicit handling of time.

Example 4. Consider the real-time notification query of Example 1 that specifies a 24-hour window of interest. WSCAN $\mathcal{W}_{24}$ adjusts

[10] $\phi(E^I) \subset \Sigma$ is reserved for input graph edges and cannot be used by operators as labels for resulting sgts. In other words, $\phi(E^I) \subset \Sigma$ corresponds to EDBs in Datalog as described in Section 4.

validity intervals of sges of the input graph stream and produces a streaming graph where each sgt is valid for 24 hours, as shown in Figure 2.

Two special cases of windows are commonly adopted in literature: NOW windows that capture only the current time by assigning the interval $[t, t+1)$ to a tuple with timestamp t, and UNBOUNDED windows that capture the entire history of the streaming graph by assigning the interval $[t, \infty)$ to a tuple with timestamp t [10, 44]. Applied over a streaming graph, NOW windows capture all tuples that emerge at the current time instant and produce the identity, whereas UNBOUNDED windows accumulate all tuples that appear in the streaming graph so far. We use NOW windows whenever a window specification is omitted from a given SGQ (Definition 15).

Definition 17 (FILTER). ***Filter operator*** *$\sigma_\Phi(S)$ is defined over a streaming graph S and a boolean predicate Φ involving the distinguished attributes of sgts, and its output stream consists of sgts of S on which Φ evaluates to true. Formally:*

$$\sigma_\Phi(S) = \big[(u, v, l, [ts, exp), \mathcal{D}) \mid (src, trg, l, [ts, exp), \mathcal{D}) \in S \land \Phi((src, trg, l, \mathcal{D}))\big].$$

Definition 18 (UNION). *Union $\cup^{[d]}$ with an optional output label parameter $d \in \Sigma \setminus \phi(E^I)$ merges sgts of two streaming graphs S_1 and S_2, and assigns the new label d if provided. Formally:*

$$S_1 \cup^{[d]} S_2 = \big[t \mid t \in S_1 \lor t \in S_2\big]$$

Definition 19 (PATTERN). *The streaming **subgraph pattern operator** is defined as $\bowtie_\Phi^{src,trg,d}(S_{l_1}, \cdots, S_{l_n})$ where each S_{l_i} is a streaming graph, Φ is a conjunction of a finite number of terms in the form $pos_i = pos_j$ for $pos_i, pos_j \in \{src_1, trg_1, \cdots, src_n, trg_n\}$ where src_i, trg_i correspond to source and target of sgts in S_{l_i}, and $src, trg \in \{src_1, trg_1, \cdots, src_n, trg_n\}$ represent the source and target of resulting sgts, and $d \in \Sigma \setminus \phi(E^I)$ represent the label of the resulting sgts. Formally:*

$$\begin{aligned}\bowtie_\Phi^{src,trg,d}&(S_{l_1}, \cdots, S_{l_n}) = \big[(u, v, d, [ts, exp), \mathcal{D} : e(u, v, l)) \mid \\ &\exists t_i = (src_i, trg_i, l_i, [ts_i, exp_i), \mathcal{D}_i) \in S_{l_i}, 1 \le i \le n \land \\ &\Phi((src_1, trg_1, \cdots, src_n, trg_n)) \land \\ &u = src \land v = trg \land \bigcap_{1 \le i \le n} [ts_i, exp_i) \ne \emptyset \land \\ &ts = \max_{1 \le i \le n}(ts_i) \land exp = \min_{1 \le i \le n}(exp_i)\big].\end{aligned}$$

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 5. Consider the real-time notification query given in Example 1; the recent liker relationship defined in the form of a triangle pattern can be represented with PATTERN $\bowtie_\phi^{src1,src4,RL}$ where $\phi = (trg_1 = trg_2 \land src_1 = src_3 \land src_2 = trg_3)$. Figure 3a shows its output over the streaming graph given in Figure 2. It 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 recent liker.

SGA operators may produce multiple value-equivalent sgts with adjacent or overlapping validity intervals. Unless otherwise specified, the *coalesce primitive* (Definition 11) is applied to their outputs to maintain the set semantics of streaming graphs and their snapshots. To illustrate, consider PATTERN in the above example: over the streaming graph given in Figure 2, 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 by as shown in Figure 3a.

Definition 20 (PATH). *The streaming **path navigation operator** is defined as $\mathcal{P}_R^d(S_{l_1}, \cdots, S_{l_n})$ where R is a regular expression over the alphabet $\{l_1, \cdots, l_n\} \subseteq \Sigma$, and $d \in \Sigma \setminus \phi(E^I)$ designates the label of the resulting sgts. The streaming graph tuple $t = (u, v, l, [ts, exp), \mathcal{D} : p)$ is an answer for a path navigation query $\mathcal{P}_R^l$ if there exists a path p between u and v in the snapshot of S at time t, i.e., $p : u \xrightarrow{p} v \in \tau_t(S) = G_t$, and the label sequence of the path p, $\phi^p(p)$ is a word in the regular language $L(R)$. Formally:*

$$\begin{aligned}\mathcal{P}_R^d(S_{l_1}, \cdots, S_{l_n}) = \big[&(u, v, d, [ts, exp), \mathcal{D}) \mid \exists p : u \xrightarrow{p} v \land \\ &\forall e_i \in p, \exists t_i = (src_i, trg_i, l_i, [ts_i, exp_i), \mathcal{D}_i) \in S_{l_i} \land \\ &\phi^p(p) \in L(R) \land \bigcap_{t \in p} [t.ts, t.exp) \ne \emptyset \land \\ &ts = \max_{t \in p}(t.ts) \land exp = \min_{t \in p}(t.exp) \land \mathcal{D} = p\big].\end{aligned}$$

PATH finds pairs of vertices that are connected by a path where (i) each edge in the path is valid at the same time instant, and (ii) path label is a word in the regular language defined by the query. This closely follows the Regular Path Query (RPQ) model where path constraints are expressed using a regular expression over the set of labels [78]. Path navigation queries in the RPQ model are evaluated under *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 [8, 12, 78]. In this paper, we adopt the arbitrary path semantics due to its widespread adoption in modern graph query languages [7, 8, 75], and the tractability of the corresponding evaluation problem [12].

Example 6. Consider the same running example given in Figure 1; the path navigation over the derived recent liker edges is represented by PATH $\mathcal{P}_{RL^+}^{RLP}$. Figure 3b shows its resulting streaming graph over the output streaming graph (Figure 3a) of the PATTERN of Example 5. It consists of sgts $(y, RLP, u, [28, 37), (y, RL, u))$, $(u, RLP, v, [29, 31), (u, RL, v))$, and $(y, RLP, v, [29, 31), \langle(y, RL, u), (u, RL, v)\rangle$ that correspond to materialized paths with label recent liker path of length one and two.

Most existing work on the RPQ model focuses on the problem of determining reachability between pairs of vertices in a graph connected by a path conforming to given regular expression [43, 50, 57, 62]. By adapting the materialized path graph model (Definition 6), we pinpoint that PATH is equipped with the ability to return paths, i.e., each resulting sgt contains the actual sequence of edges that form the path with a label sequence conforming to given regular expression.

SGA builds on the Regular Property Graph Algebra (RPGA) [16], which is itself based on Regular Queries (RQ). Of course, both RPGA and RQ formulate graph queries over static property graphs, while SGA allows persistent graph queries over streaming property graphs. SGA operators are defined over streaming graphs (Definition 8), and they access and manipulate validity intervals implicitly, thus they generalize their non-streaming counterparts with implicit handling of time. This follows from the fact that the semantics of SGA operators are defined through *snapshot reducibility* (Definition 13), that is, the snapshot of the result of a streaming operator on a streaming graph S at time t is equal to the result of the corresponding non-streaming operator on the snapshot of the streaming graph S at time t.

5.2 Formulating Queries in SGA

SGA can express all queries that can be specified by SGQ (Section 4). In this section, we describe a set of rules that can be applied to convert a SGQ query to a SGA expression.

Given a SGQ $Q(S, \mathcal{W}^{\mathcal{T}})$ and the label-based logical partitioning (Definition 9) of its input streaming graph $S = \bigcup_{l \in \Sigma}(S_l)$, Algorithm **SGQParser** produces the canonical SGA expression. In brief, Algorithm **SGQParser** processes the predicates of a given SGQ and generates the corresponding SGA expression in a bottom-up manner: each EDB l corresponds to a WSCAN over an input streaming graph S_l^I, each application of transitive closure corresponds to a PATH, each IDB d corresponds to a UNION or PATTERN based on the body of the corresponding rule.

Algorithm SGQParser:

input : Streaming Graph Query $Q(S, \mathcal{W}^{\mathcal{T}})$
output: SGA Expression e

1 $G_Q \leftarrow$ `Graph`(Q) `// dependency graph`
2 $[r_1, \cdots, r_n] \leftarrow$ `TopSort`(G_Q) `// topological sort`
3 $exp \leftarrow []$ `// empty mapping`
4 **for** $1 \leq i \leq n$ **do**
5   **switch** r_i **do**
6   **case** $l(src, trg), l \in \phi(E^I)$ **do**
7       $exp[l] \leftarrow \mathcal{W}^{\mathcal{T}}(S_l)$
8   **case** $l^*(x, y) asd$ **do**
9       $exp[d] \leftarrow \mathcal{P}^d_{l*}(exp[l])$
10   **otherwise do**
11       $d(src, trg) \leftarrow r_i.head$, $[b_1, \cdots, b_n] \leftarrow r_i.body$
12       $\Phi \leftarrow$ `GenPred`$(r_i.body)$
13       $e \leftarrow \bowtie_{\Phi}^{src,trg,d} (exp[b_1], \cdots, exp[b_n])$
14   **if** $exp[d] \neq \emptyset$ **then**
15         $exp[d] \leftarrow exp[d] \cup e$
16   **else**
17         $exp[d] \leftarrow e$
18 **return** $exp[Answer]$

Theorem 7. *Streaming graph queries (SGQ) can be expressed in the proposed SGA, i.e., there exists a SGA expression $e \in SGA$ for any $Q \in SGQ$.*

Proof. We prove that the Algorithm **SGQParser** is guaranteed to terminate and to create a canonical SGA expression for a given SGQ Q. The dependency graph of an RQ is acyclic as RQ are non-recursive (Definition 14), hence, Line 2 is guarenteed to define a partial order over the Q's predicates. Algorithm **SGQParser** 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 $d(src, trg) := l_1(src_1, trg_1), \cdots, l_n(src_n, trg_n)$, 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 guarenteed to have SGA expressions for each predicate $r_j, 1 \leq j \leq i$ when processing the predicate r_i. Once all predicates are processed, Line 18 returns the SGA expression of the reserved *Answer* predicate. Hence, we conclude that Algorithm **SGQParser** correctly constructs an SGA expression for a given SGQ. □

The complexity of evaluating SGA expressions is the same as that of RQ given their relationship as noted above – it is NP-complete in combined complexity and NLogspace-complete in data complexity [16, 66].

Example 8 (Canonical Translation). Consider the real-time notification task in Example 1 and its corresponding RQ given in Example 3. Algorithm **SGQParser** generates the following canonical SGA expression for its corresponding SGQ with a sliding window of 24 hours:

$$\bowtie_{\phi_2}^{(src1, trg2, notify)} \Big(\mathcal{P}^{RLP}_{RL^+}\Big(\bowtie_{\phi_1}^{src1, src2, RL} \big(\mathcal{W}_{24}(S_l), \mathcal{W}_{24}(S_p), \mathcal{P}^{FP}_{f^+}(\mathcal{W}_{24}(S_f))\big)\Big), \mathcal{W}_{24}(S_p) \Big)$$

$\phi_1 = (trg_1 = trg_2 \wedge src_1 = src_3 \wedge src_2 = trg_3)$,

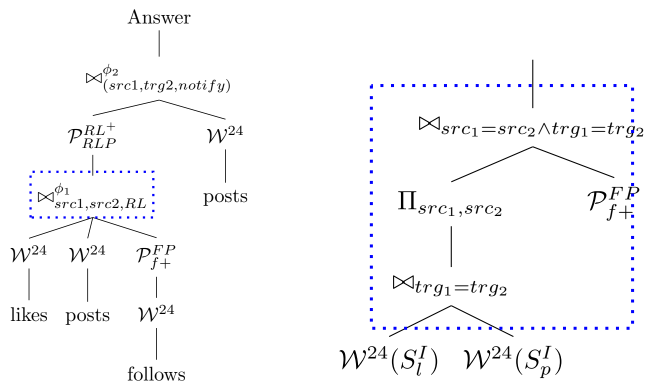
$\phi_2 = (trg_1 = src_2)$

Figure 4a illustrates the logical plan for the same SGQ that consists of logical operators of SGA.

5.3 Closedness and Composability

SGA operators proposed in this paper are closed over streaming graphs, that is, the output of an SGA operator is a valid streaming graph if its inputs are valid streaming graphs. The fact that SGA is a closed query language over the streaming graph model (Section 3) means that SGA queries (expressions) are composable, i.e., the output of one query (expression) can be used as input of another query (expression).

Given Theorem 7, SGQ language is also closed – each query takes one or more streaming graphs as input and produces a streaming graph as output. Furthermore, it is composable as the output of a query can be the input of the subsequent query. This is in contrast to the existing graph query languages such as SPARQL and Cypher that are not composable and may not be closed. Cypher 9 requires graphs as input, but produce tables as output so the language is neither closed nor composable – the results of a Cypher query cannot



(a) Logical plan for Example 8. **(b) Join tree for PATTERN.**

Figure 4: (a) Logical query plan for the real-time notification task (Example 1) based on its canonical SGA expression (Example 8), and (b) the join tree that consists of binary hash joins for PATTERN in (a).

be used as input to a subsequent one without additional processing. SPARQL can generate graphs as output using the `CONSTRUCT` clause, and is therefore closed; however, it requires query results to be made persistent and therefore not easily composable [16]. Closedness should be a required property of any algebra as it enables query rewriting (see next section) and query optimization. Composability is a desired feature for a declarative query language as it facilitates query decomposition, view-based query evaluation, query rewriting etc.

5.4 Transformation Rules

As noted above, closedness of an algebra is important for query rewriting to explore the space of equivalent relational algebra expressions; this is a key component of query optimization. Although we do not address the optimization problems in this paper (that is future work), we briefly highlight some of the possible transformation rules for generating equivalent query plans to demonstrate the possibilities provided by SGA.

Some of the traditional relational transformation strategies such as join ordering, predicate push down are applicable to UNION, FILTER and PATTERN due to snapshot-reducibility. UNION and FILTER operators are streaming generalizations of corresponding relational union and selection operators, and the PATTERN operator can be represented using a series of equijoins.

Below, we describe transformation rules involving the other novel SGA operators:

Transformation Rules Involving WSCAN: The WSCAN operator $\mathcal{W}^{\mathcal{T}}$ 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) $\mathcal{W}^{\mathcal{T}}(\sigma^{\phi}(S)) = \sigma^{\phi}(\mathcal{W}^{\mathcal{T}}(S))$
(2) $\mathcal{W}^{\mathcal{T}}(S_1 \cup S_2) = \mathcal{W}^{\mathcal{T}}(S_1) \cup \mathcal{W}^{\mathcal{T}}(S_2)$

Transformation Rules Involving PATH: We identify two transformation rules of the PATH operator for regular expressions with alternation and concatenation:

(1) Alternation: $\mathcal{P}^{d}_{a|b}(S_a, S_b) = \bigcup^{d}(S_a, S_b)$
(2) Concatenation: $\mathcal{P}^{d}_{a \cdot b}(S_a, S_b) = \bowtie^{src1,trg2,d}_{trg_1=src_2}(S_a, S_b)$

These transformation rules enable the exploration of a rich plan space for SGQ that are represented in the proposed SGA. In particular, the novel PATH operator 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 main approaches: graph traversals guided by finite automata, or relational algebra extended with transitive closure, i.e., *alpha*-RA [12, 27, 43, 62, 68, 79]. Yakovets et al. introduce a hybrid approach (Waveguide) and model the cost factors that impact the efficiency of RPQ evaluation on static graphs [79]. SGA enables the representation of these approaches in a uniform manner, and the above transformation rules enable us to explore plan space that subsumes these existing plans. Section 7.4 shows how an example application of these transformation rules and presents a micro-benchmark demonstrating the potential benefits of query optimizations through the exploration of the rich plan space due to SGA's transformation rules.

6 OPERATOR IMPLEMENTATIONS

This section introduces a physical operator algebra for SGA that consists of non-blocking algorithms as physical operator implementations. The intuition behind our physical algebra is the following: in the absence of explicit deletions, expirations from windows and result set exhibit a temporal pattern in the way that streaming graph tuples are inserted. We show that utilizing this pattern enables novel algorithms for query processing and window maintenance. We also describe how to handle explicit deletions using *negative tuples* for applications that require explicit deletions of previously inserted edges.

As noted earlier, our focus is on *incremental* evaluation of persistent queries where the goal is to avoid re-computation of the entire result by only computing the changes to the output as new input arrives. It is desired that physical operators of a streaming system have non-blocking behaviour, i.e., they do not need the entire input to be available before producing the first result. Operators with blocking behaviour, such as set difference and nested loop joins, can be unblocked by restricting their range to a finite subset of an unbounded stream [11, 31]. Hence, sliding windows that are used to restrict the scope of queries to recent data, a desired feature in many streaming applications, provide a tool for incremental computation of such operators over unbounded streams. Algorithms we describe here constitute non-blocking, incremental implementations of operators in our streaming graph algebra consistent with the semantics (Section 5.1). 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.

The standard implementations of stateless operators FILTER and UNION can be directly used in SGA. Similarly, WSCAN is a stateless operator that can be implemented using the standard *map* operator that adjusts the validity interval of an sgt according to the given window specification.

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.

6.1 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 [77] 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 [32, 74]. This produces an append-only stream of results for internal windows that do not invalidate previously reported results upon expiration of their participating tuples [30]. 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 leafs represent streaming graphs as input streams and internal nodes represent pipelined hash join operators. For instance, Figure 4b shows the join tree for PATTERN of SGQ Q_1 whose logical plan is given in Figure 4a. We leave the problem of finding efficient join plans (e.g. using *worst-case optimal joins* [60]) for subgraph pattern queries over streaming graphs as a future work and use the ordering of subgraph pattern predicates given in the query.

6.2 Path Navigation

The semantics of PATH follows the RPQ model, which specifies path constraints as a regular expression over the alphabet of edge labels and checks whether a path exists with a label that satisfies the given regular expression [12, 57]. We propose the *Streaming Path Navigation* **S-PATH** algorithm as a physical operator implementation of PATH. **S-PATH** follows the automata-based RPQ evaluation strategy [57, 62] using the arbitrary path semantics as discussed in Section 5.1.

Algorithm **S-PATH** incrementally performs a traversal of the underlying snapshot graph under the constraints of a given RPQ as sgts arrive. It first constructs a DFA from the regular expression of a PATH operator, and initializes a spanning forest-based data structure, called $\Delta - \mathsf{PATH}$, that is used as the internal operator state during query processing. $\Delta - \mathsf{PATH}$ is used to maintain a path segment, i.e., a partial result, between each pair of vertices in

Algorithm S-PATH:

input : Input streaming graph S, Regular expression R, output label o
output: Output streaming graph S_O

$A(S, \Sigma, \delta, s_0, F) \leftarrow \mathtt{ConstructDFA}(R)$
Initialize $\Delta - \mathsf{PATH}$
$S_O \leftarrow \emptyset$
$\mathrm{R} \leftarrow \emptyset$
foreach $(u, v, l, [ts, exp), \mathcal{D}) \in S$ **do**
  foreach $s, t \in S$ *where* $t = \delta(s, l)$ **do**
    if $s = s_0 \wedge T_u \notin \Delta - \mathsf{PATH}$ **then**
      add T_u with root node (u, s_0)
    $\mathrm{T} \leftarrow \mathtt{ExpandableTrees}(\Delta - \mathsf{PATH}, (u, s), ts)$
    foreach $T_x \in \mathrm{T}$ **do**
      if $(v, t) \notin T_x$ **then**
        $\mathrm{R} \leftarrow \mathrm{R}+$ **Expand**(T_x, (u, s), (v, t), $e(u, v)$)
      else if $(v, t).exp < min((u, s).exp, exp)$ **then**
        $\mathrm{R} \leftarrow \mathrm{R}+$ **Propagate**(T_x, (u, s), (v, t), $e = (u, v)$)
foreach *sgt* $t \in R$ **do**
  push t to S_O

Algorithm Expand:

input : Spanning Tree T_x rooted at (x, s_0),parent (u, s), child (v, t), edge $e(u, v)$
output: Set of results R

$\mathrm{R} \leftarrow \emptyset$
Insert (v, t) as (u, s)'s child
$(v, t).ts = max(e.ts, (u, s).ts)$
$(v, t).exp = min(e.exp, (u, s).exp)$
if $t \in F$ **then**
  $p \leftarrow \mathsf{PATH}(T_x, (v, t))$
  $R \leftarrow R + (x, v, O, [(v, t).ts, (v, t).exp), p)$
foreach *edge* $e(v, w) \in G_{ts}$ *s.t.* $\delta(t, \phi(e)) = q$ **do**
  if $(w, q) \notin T_x$ **then**
    $\mathrm{R} \leftarrow \mathrm{R}+$ **Expand**(T_x, (v, t), (w, q), $e(v, w)$)
  else if $(w, q).exp < min((v, t).exp, e.exp)$ **then**
    $\mathrm{R} \leftarrow \mathrm{R}+$ **Propagate**(T_x, (v, t), (w, q), $e(v, w)$)
return R

Algorithm Propagate:

input : Spanning Tree T_x rooted at (x, s_0), parent (u, s), child (v, t), edge $e(u, v)$
output: Set of results R

$\mathrm{R} \leftarrow \emptyset$
$(v, t).pt = (u, s)$
$(v, t).ts = min((v, t).ts, max(e.ts, (u, s).ts))$
$(v, t).exp = max((v, t).exp, min(e.exp, (u, s).exp))$
if $t \in F$ **then**
  $p \leftarrow \mathsf{PATH}(T_x, (v, t))$
  $\mathrm{R} \leftarrow \mathrm{R} + (x, v, O, [(v, t).ts, (v, t).exp), p)$
foreach *edge* $e = (v, w) \in G_{ts}$ *s.t.* $\delta(t, \phi(e)) = q$ **do**
  if $(w, q).exp < min((v, t).exp, e.exp)$ **then**
    $\mathrm{R} \leftarrow \mathrm{R}+$ **Propagate**(T_x, (v, t), (w, q), $e(v, w)$)
return R

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 $\Delta - \mathsf{PATH}$ 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 $\mathcal{P}_d^R$ 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 $\phi^p(p)$ such that $s = \delta^*(s_0, \phi^p(p))$.*

A node $(u, s) \in T_x$ indicates that there is a path p in the snapshot graph with label $\phi^p(p)$ such that $s = \delta^*(s_0, \phi^p(p))$, 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 largest 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 `max` over the expiry timestamp of path segments. [11] 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 expirations have a temporal order unlike explicit deletions, and S-PATH utilizes these temporal patterns to simplify window maintenance.

Definition 22 ($\Delta - \mathsf{PATH}$ Index). *Given an automaton A for the regular expression R of a PATH operator $\mathcal{P}_d^R$ and a streaming graph S at time t, $\Delta - \mathsf{PATH}$ 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 $\delta(s_0, l) \neq \emptyset$ and $src = x$.*

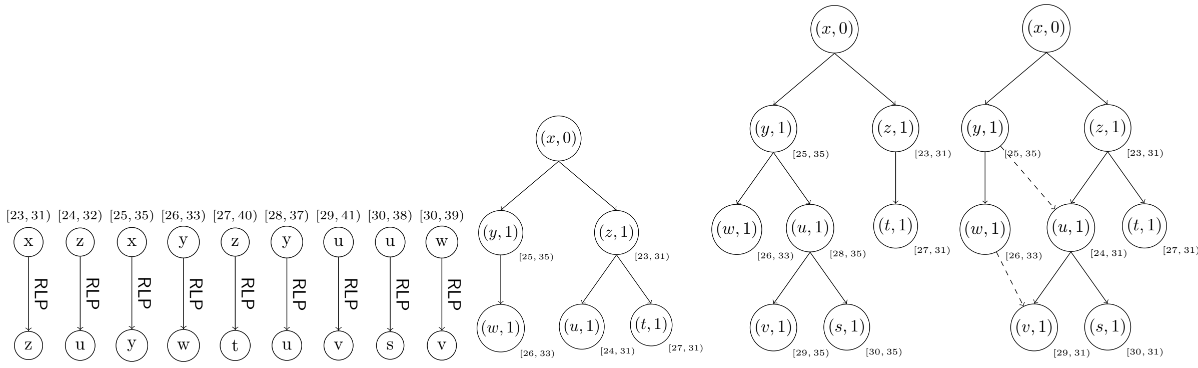
$\Delta - \mathsf{PATH}$ encodes a single entry for each pair of vertices under the constraints of a given query, consistent with the set semantics of snapshot graphs (Section 3). Due to spanning-tree construction (Definition 21), actual paths can easily be recovered by following the parent pointers; hence, $\Delta - \mathsf{PATH}$ constitutes a compact representation of intermediate results for path navigation queries over materialized path graphs. Our implementation models $\Delta - \mathsf{PATH}$ 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 $t = (u, v, l, [ts, exp), \mathcal{D})$, Algorithm **S-PATH** probes this inverted index of $\Delta - \mathsf{PATH}$ 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 $s \in \{s \in S \mid \delta(s, l) \neq \emptyset\}$ (Line 9). If the target node (v, t) for $t = \delta(s, l)$ 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 $\Delta - \mathsf{PATH}$, i.e., the target node (v, t) is already in T_x, Algorithm **S-PATH** compares its expiry timestamp with the new candidate (Line 13). 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 (v, t) 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 in Example 1 whose SGA expression is given in Example 8. Figure 5a shows an excerpt of the streaming graph input to the PATH operator $\mathcal{P}_{RLP}^{RL^+}$. Figure 5b depicts a spanning tree $T_x \in \Delta - \mathsf{PATH}$ at $t = 27$. Upon arrival of the sgt $(y, u, RL, [28, 37), \mathcal{D} = \{(y, RL, u)\})$ 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 and the parent pointer of $(u, 1) \in T_x$ is updated (Line 13 in Algorithm **S-PATH**). Then, incoming sgts at times $t = 28$ and $t = 29$ are processed by Algorithm **Expand** as corresponding target nodes $(v, 1)$ and $(s, 1)$ are not in T_x, adding $(v, 1)$ and $(s, 1)$ as children of $(u, 1)$. At $t = 30$, the incoming sgt $(w, v, RL, [30, 39), \mathcal{D} = \{(w, RL, v)\})$ might extend the path segment from $(x, 0)$ to $(v, 1)$ with expiry timestamp 33. However, Algorithm **S-PATH** does not make any modification to $\Delta - \mathsf{PATH}$, as $(v, 1)$ is already in T_x with a larger timestamp (Line 13). Figure 5c depicts the resulting spanning tree T_x at $t = 30$.

As described earlier, $\Delta - \mathsf{PATH}$ stores a *parent* pointer for each node pointing to its parent node in the corresponding spanning tree, and Algorithm **Propagate** updates these pointers during processing. By traversing these parent pointers for each resulting sgt, Algorithm **Expand** can construct the actual path (Line 7) and return it as a part of the resulting sgt, i.e., it populates the implicit payload attribute $\mathcal{D}$ of the resulting sgt with the sequence of edges that forms the resulting path. The cost of this operation is $O(l)$ where l is the length of the resulting path.

$\Delta - \mathsf{PATH}$ guarantees that the expiry timestamp of a node (u, s) in T_x is equal to largest expiry timestamp of all paths between x and u in the snapshot graph with a label l such that $s = \delta^*(s_0, l)$. Consequently, for a node $(u, s) \in T_x$ with expiry timestamp smaller than t, there cannot be another path from x to u with an equivalent label that is valid at time t. Consider the spanning tree given in Figure 5c. Algorithm **S-PATH** can directly determine, without additional processing, that nodes $(z, 1)$ and $(t, 1)$ are expired as their expiry timestamp is 31. Thus, at any given time t, Algorithm **S-PATH** can simply ignore a node $(u, s) \in T_x$ with expiry timestamps smaller than t (Line 13) and such nodes can be removed from $\Delta - \mathsf{PATH}$. To prevent $\Delta - \mathsf{PATH}$ from growing unboundedly due to expired tuples, a background process periodically purges expired tuples from $\Delta - \mathsf{PATH}$.

[11] Arbitrary path semantics provides the flexibility for the aggregation function f_{agg} of the coalesce primitive.



(a) Streaming Graph S **(b) $t = 27$** **(c) $t = 30$** **(d) $t = 30$ for [62]**

Figure 5: (a) A streaming graph S_{RLP} as the input for PATH operator, (b) spanning tree T_x at $t = 28$, (c) spanning tree T_x at $t = 30$ of the proposed algorithm following the *direct* approach, and (d) spanning tree T_x at $t = 30$ of [62] following the *negative tuple* approach.

Our strategy of utilizing the validity intervals of path segments is in contrast to our previous work [62], which studies the design space of streaming RPQ evaluation under arbitrary and simple path semantics. Algorithms in [62] are based on the *negative tuple* approach; expirations 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* [35], optimized for RPQ evaluation on streaming graphs. Although its cost can be amortized over a slide interval, using the negative tuple approach for window management incurs a non-negligible overhead, as analytically and empirically presented in [62]. Instead, our proposed algorithm for the PATH operator 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. We argue that such an approach can greatly simplify window management in the absence of explicit deletions, i.e., sliding windows over append-only streaming graphs. The following example illustrates how the *negative tuple* and *direct* approaches differ, respectively for [62] and our proposed algorithm.

Example 10. Consider the same real-time notification query as in Example 9. Both approaches behave similarly until $t = 28$ as all vertex-state pairs in T_x have a single derivation at $t = 27$ (Figure 5b). Upon arrival of the sgt $(y, u, HI, [28, 37), \mathcal{D} = \{(y, HI, u)\})$ at $t = 28$, the negative tuple approach as in [62] does not update T_x as $(u, 1)$ is already in T_x, whereas the direct approach as used in this paper updates the validity interval and the parent pointer of $(u, 1) \in T_x$ (Line 13 in Algorithm **S-PATH**). Then, incoming sgts at times $t = 28$ and $t = 29$ are processed similarly, adding $(v, 1)$ and $(s, 1)$ as children of $(u, 1)$. Figures 5c and 5d depict the corresponding spanning trees at $t = 30$ for the *direct* and the *negative tuple* approaches, respectively. Note that in Figure 5c, the validity intervals of nodes in the subtree rooted at node $(u, 1)$ reflects the newly discovered path from x to u through y in G_{30}. The *negative tuple* and the *direct* approach differs at $t = 31$ as multiple nodes expire. The *negative tuple* used as in [62] marks the entire subtree of $(z, 1)$ as potentially expired (Figure 5d), and performs a traversal of the snapshot graph G_{31} to find alternative, valid paths for expired nodes. These traversals undo the effect of expired sgts via explicit deletions. Upon discovering alternative paths for nodes $(u, 1)$, $(v, 1)$ and $(s, 1)$ that are valid at time $t = 31$, they are re-inserted into T_x. Instead, our proposed algorithm can *directly* determine the expired nodes based on the validity intervals (nodes $(z, 1)$ and $(t, 1)$ as shown in Figure 5c) without additional processing.

6.3 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.[12] 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.

Similarly, we adopt the *negative tuple* approach for explicit deletions of sgts in PATH operator. In brief, upon explicit deletion of an sgt, we first identify tree-edges that disconnect spanning trees in $\Delta - \mathsf{PATH}$. For each such tree-edge, we first mark the nodes in the subtree that are disconnected due to the explicit deletion. Then, for

[12] The *negative tuple* approach can also be used to signal expirations through an explicit deletion of the corresponding sgt to undo its effect [29, 33, 62]. 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 [33].

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. The use of negative tuples for explicit deletions in the context of RPQ evaluation is first proposed by Pacaci et al [62]. However, that technique uses the *negative tuple* approach for both explicit deletions and window management, i.e., processing of expired sgts due to sliding window movements. This is in contrast to our strategy of utilizing the validity intervals of path segments in $\Delta - \mathsf{PATH}$ index to *directly* locate expired tuples, which simplifies the window management by eliminating the need for explicit processing of expired sgts.

7 EXPERIMENTAL ANALYSIS

Our objective in this section is to demonstrate the feasibility and the performance of the algebraic framework we propose in this paper. We first describe a prototype streaming graph query processor that includes the algorithms presented in Section 6 as physical operators of the proposed SGA (Section 7.1.1). Using this query processor, we provide and end-to-end performance analysis of our algebraic approach for persistent evaluation of streaming graph queries (Section 7.2). Then, we assess the scalability by varying the window size $\mathcal{T}$ and the slide interval β (Section 7.3). Finally, we highlight the practical benefits of the proposed SGA by exploring the rich plan space through SGA's transformation rules and demonstrate the potential performance improvements of exploring this plan space (Section 7.4).

We emphasize that the design of the "best" physical operators for the defined algebra or the full-fledged query optimization is not the purpose of this paper. We provide the physical implementations in Section 6 and their performance in this section to demonstrate that it is possible to implement SGA efficiently. Development of alternative physical implementations and query optimization is part of our future work.

The highlights of our experimental analysis are as follows:

- Algebraic framework we propose on this paper provides a principled approach to formulate and evaluate streaming graph queries with complex graph patterns, and it can offer significant performance gains for persistent evaluation of streaming graph queries compared to a general-purpose solution.
- Utilizing structural properties of streaming graph queries and temporal patterns of sliding windows yield efficient physical operators for the proposed SGA. In particular, the novel **S-PATH** Algorithm for PATH can process can improve the performance of recursive queries by simplifying the processing of expired tuples.
- Physical implementations of our stateful operators based on the *direct* approach provide robust performance w.r.t. varying slide intervals, i.e., they can provide fresh results at a fine granularity regardless of β.
- The algebraic-approach for persistent evaluation of SGQ that is advocated in this paper enables the exploration of a rich plan space for SGQ, offering significant computational gains for persistent evaluation of SGQ.

7.1 Experimental Setup

7.1.1 Prototype Implementation. We implement a streaming graph query processor based on the algebraic framework we propose in this paper as a part of the S-graffito Streaming Graph Management System[13]. Our streaming graph query processor is implemented as a layer on top of Timely Dataflow (TD) [59]. TD is a distributed, data-parallel dataflow engine for high-throughput, low-latency computations that are modeled as possibly cyclic dataflows. We implement algorithms presented in Section 6 as TD operators, and, given an SGQ, we construct a dataflow computation that consists of these operators.

Our prototype adopts the standard implementations of stateless UNION and FILTER from TD, and implements WSCAN using the standard *map* operator. As described in Section 6.1, PATTERN is implemented as a series of symmetric hash joins based on the *direct* approach [33, 36, 37]. Finally, PATH is implemented using the novel **S-PATH** algorithm that we propose in this paper. TD's novel progress tracking mechanism enables us to coordinate the logical time between the operators of the dataflow graph, eliminating inconsistency and delayed result issue that might arise with the *direct* approach [29, 33].

Our current implementation transforms the operator tree of a given SGQ into a dataflow graph to be executed on TD. We first obtain the canonical SGA expression and the corresponding logical query plan for a given SGQ (Section 5.2). Physical query plans are directly derived from their logical counterparts by substituting each logical operator with its physical counterpart (Section 6). Consequently, resulting dataflow graphs are tree-shaped similar to logical plans that are based on the canonical SGA expressions. Each dataflow, i.e., physical plan, consists of one or more windowing operators that are placed at the sources and a sink operator that pushes results back to the user incrementally, as they are generated.

7.1.2 Setup. Experiments are run on a Linux server with 32 physical cores and 256GB memory. We use the slide interval β to control the granularity at which the time-based sliding window slides. Consequently, β controls the size of input batches and the temporal granularity of the output. 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.

7.1.3 Datasets. **Stackoverflow** (SO) is a temporal graph of user interactions on this website containing 63M interactions (edges) of 2.2M users (vertices), spanning 8 years [64]. 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. Its highly dense and cyclic nature causes a

[13] https://dsg-uwaterloo.github.io/s-graffito/

high number of intermediate results and resulting paths; so it is the most challenging one for the proposed algorithms. We set the window size $\mathcal{T}$ to 1 month and the slide interval β to 1 day unless specified otherwise.

LDBC SNB is a synthetic social network graph that is designed to simulate real-world interactions in an online social network [26]. We extract the update stream of the LDBC workload, which exhibits 8 different types of interactions. We use a scale factor of 10 with approximately 7.2M users and posts (vertices) and 40M user interactions (edges). In our experiments, we use replyOf, hasCreator and likes edges between users and posts, and knows edges between users. LDBC update stream spans 3.5 months of user activity and we set $|W| = 1$ month and $\beta = 1$ day unless specified otherwise.

7.1.4 Workloads. To the best of our knowledge, no current benchmark exists featuring RQ for graph databases. The existing benchmarks are limited to UCRPQ, thus not reaching the full expressivity of RQ even for static graphs. Although there exist specialized streaming RDF benchmarks such as LSBench [1] and Stream WatDiv [28], they only focus on SPARQL v1.0 (thus not even including simple RPQs), and their workloads do not contain any recursive queries. Hence, we formulate a set streaming graph queries from existing UCRPQ-based workloads as follows: we first collect a set of query graphs in the form of UCRPQ from previously published benchmarks and studies [13, 17, 26, 62, 79], and we compose a set of complex graph patterns from this collection by applying a Kleene star over each graph pattern. Table 1 lists the set of graph patterns of increasing expressivity (from RPQ to complex RQ with complex graph patterns) that we use to define streaming graph queries. $Q_1 - Q_4$ correspond to commonly used RPQ in existing studies [17, 62, 79], and we use those to test our PATH operator. Q_5 & $Q6$ are CRPQ-based complex graph patterns based on LDBC SNB queries IS7 and IC7 [26]. For instance, Q_6 – LDBC SNB query IC7 – with edge labels knows, likes and hasCreator asks for recent likers of person's messages that are connected by a path of friends. Q_7 & Q_8 – Examples 1 & 2, respectively – are the most expressive RQ-based complex graph patterns that we construct to demonstrate the abilities of our proposed SGA to unify subgraph pattern and path navigation queries in a structured manner and to treat paths as first-class citizens. For instance, Q_7 defines a path query over the complex graph pattern of Q_6; it finds arbitrary length paths where users are connected by the recent liker pattern. Note that this query cannot be expressed in existing graph query languages such as Cypher and SPARQL (thus it cannot be evaluated on static graphs by corresponding offline engines without additional processing). Finally, for each dataset, we instantiate the query workload from these graph patterns by choosing appropriate bindings, i.e., edge labels, for each query edge and by setting the duration of time-based sliding windows $\mathcal{W}_\mathcal{T}$ based on characteristics of the particular streaming graph used.

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 99^{th} 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 & cyclic. The throughput ranges from hundreds of edges-per-second for the SO to hundreds of thousands of edges-per-second for the LDBC.

Table 1: $Q_1 - Q_4$ correspond to common RPQ observed in real-world query logs [17], and $Q_5 - Q_8$ 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 $IC7$ [26], respectively, 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, and Q_8 corresponds to the complex graph pattern given in Example and 2. a, b and c correspond to edge predicates that are instantiated based on the dataset characteristics.

Name	Query
Q_1	$?x, ?y \leftarrow ?x\ a^*\ ?y$
Q_2	$?x, ?y \leftarrow ?x\ a \circ b^*\ ?y$
Q_3	$?x, ?y \leftarrow ?x\ a \circ b^* \circ c^*\ ?y$
Q_4	$?x, ?y \leftarrow ?x\ (a \circ b \circ c)^+\ ?y$
Q_5	$RR(m1, m2) \leftarrow a(x, y),\ b(m1, x),\ b(m2, y),\ c(m2, m1)$
Q_6	$RL(x, y) \leftarrow a^+(x, y),\ b(x, m),\ c(m, y)$
Q_7	$RL(x, y) \leftarrow a^+(x, y),\ b(x, m),\ c(m, y)$ $Ans(x, m) \leftarrow RL^+(x, y), c(m, y)$
Q_8	$P^+(x, y) \leftarrow a(x, z),\ a(y, z)$ $Ans(x, m) \leftarrow P^+(x, y)$

7.2.2 Comparative Analysis. Existing work on query processing over streaming data such as DSMSs and stream RDF systems cannot process queries in Table 1 as they focus on relational queries and SPARQL v1.0, respectively (as discussed in §1). Differential Dataflow (DD) [56] is a state-of-the-art system built atop TD for incremental maintenance of arbitrary dataflows. In addition to standard relational operators such as *map*, *join*, *distinct* etc., DD provides *iterate* operator for fixed-point computations, which enables DD to maintain cyclic dataflows over evolving datasets under arbitrary changes. Consequently, DD can be used to evaluate SGQ on time-based sliding windows over a streaming graph by maintaining the window content as an evolving collection where each window slide triggers: (i) insertion of new sgts, and (ii) the deletion of old sgts that expire from the window. In contrast to physical operators of SGA that are based on *direct* approach that utilizes the temporal patterns of time-based sliding windows, this corresponds to the *negative tuple* approach that relies on *explicit deletions* for expirations (§6). DD constitutes a strong baseline for evaluation and enables us to assess the potential benefits of our algebraic framework.

Given a canonical SGA expression of an SGQ, we construct a dataflow graph of DD operators: PATTERN is modeled as a series of joins, PATH is modeled as a fixed-point computation over the underlying path pattern. Each time-based sliding window is represented as an evolving collection; incoming (expiring) sgts are inserted (deleted) into (from) this collection as the window slides. In particular, we extend the WSCAN operator to emit a negative tuple when an sgt expires, similar to SEQ-WINDOW of CQL [10]. The *distinct* operator is used to provide the set semantics over resulting streaming graphs (Def. 12). Table 2 (DD) reports the throughput and tail latency of DD dataflows for all queries in Table 1.

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 LDBC-SF10 graphs with $|W|$ = 30 days and β = 1 day.

Graph	System	Q_1		Q_2		Q_3		Q_4		Q_5		Q_6		Q_7		Q_8	
		Tput	TL	Tput	TL	Tput	TL	Tput	TL	Tput	TL	Tput	TL	Tput	TL	Tput	TL
SO	SGA	**2884**	**4**	**9074**	**4.9**	**391**	177	348	94.9	**234058**	**0.4**	**625**	**51.4**	**353**	**52.6**	**262**	87.8
	DD	1209	6.3	4512	5.8	368	**121.7**	**374**	**82.8**	63330	1	283	72.6	275	74	173	**82.5**
LDBC	SGA	95903	1.4	244653	1.8	224342	1.9	278647	0.4	**14000**	**79.5**	**450957**	**0.8**	**130651**	**10.8**	30622	7.7
	DD	**121133**	**0.8**	**299245**	**1.2**	**316267**	**1.1**	**303068**	**0.2**	12053	109.5	402048	0.9	21284	141	**39853**	**3.8**

Overall, our SGA-based query processor outperforms the DD baseline for the majority of the queries on SO and provides a competitive performance on the LDBC dataset[14]. Due to highly cyclic structure of SO, there are many alternative paths between each pair of vertices, and the Algorithm **S-PATH** for PATH manages to utilize the temporal patterns of sliding window movements to simplify expirations by maintaining a compact representation of valid path segments (§6.2). DD-based query processor provides better performance on linear path queries Q_1–Q_4 on LDBC, but not others. This is due to the tree-shaped structure of replyOf edges in LDBC, where there is only one path between a pair of vertices, so Algorithm **S-PATH**'s optimizations do not apply. Performance variations on LDBC suggest optimization opportunities for recursive graph queries when selecting physical operators implementations, as in the case for streaming relational joins [33]. In summary, these results demonstrate the feasibility of our algebraic approach for evaluating SGQ and our physical operator implementations. In particular, employing the *direct* approach by utilizing the temporal patterns of sliding window movements have significant performance advantages for evaluating recursive queries on cyclic graphs.

7.3 Sensitivity Analysis

In this section, we analyze the impact of the window size $\mathcal{T}$ and the slide interval β on end-to-end query performance of the proposed streaming graph query processor. We use SO graph for this experiment as it is dense, cyclic structure stresses our operator implementations. Figure 6a 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 $\mathcal{T}$, 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 β on performance. As previously mentioned, the slide interval β controls the time-granularity at which the sliding window progresses, and our prototype implementation uses β to control the input batch size. Figure 6b 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 utilize batching to improve throughput with larger batch sizes. Consequently, the tail latency of window movements increases with increasing slide interval. This is in contrast to DD whose throughput increases with increasing β as shown in Figure 7. DD and its underlying indexing mechanism, i.e., shared arrangements [55], are designed to utilize batching and improve throughput with increasing batching size: all sgts that arrive within one interval are batched together with a single logical timestamp (epoch) and DD operators can explore the latency vs throughput trade-off by changing the granularity of each epoch. The investigation of batching within SGA operators and the identification of other optimization opportunities is a topic of future work.

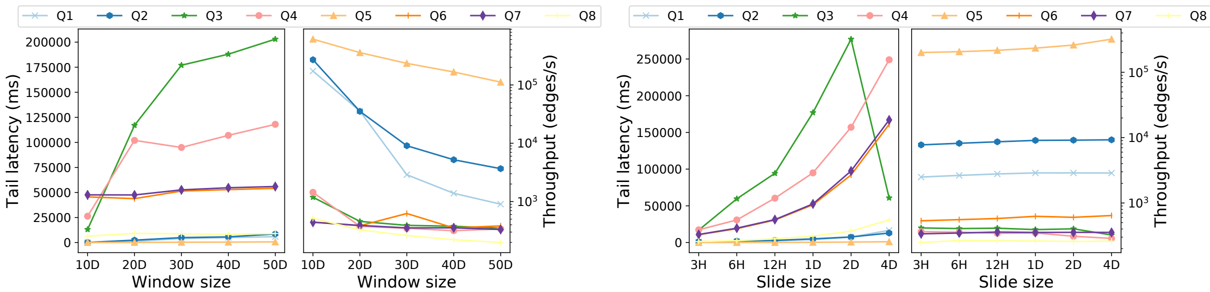
[14] On LDBC, Q_6 & Q_7 do not have the Kleene plus over a as it causes DD to timeout.

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 in §5.4. Although we compile physical query plans directly from the canonical SGA expression of a given SGQ without addressing the optimization issues in this paper, 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. 8 demonstrates the throughput and the tail latency of different plans obtained from following equivalent SGA expressions for Q_4:

- SGA: $\mathcal{P}^{l}_{d^{+}}(\bowtie^{src_1,trg_3,d}_{trg_1=src_2 \wedge trg_2=src_3} (S_a, S_b, S_c))$
- P1: $\mathcal{P}^{l}_{(a\cdot b\cdot c)^{+}}(S_a, S_b, S_c)$
- P2: $\mathcal{P}^{l}_{(a\cdot d)^{+}}(S_a, \bowtie^{src_1,trg_2,d}_{trg_1=src_2} (S_b, S_c))$
- P3: $\mathcal{P}^{l}_{(d\cdot c)^{+}}(S_c, \bowtie^{src_1,trg_2,d}_{trg_1=src_2} (S_a, S_b))$

The first expression, **SGA**, is the canonical SGA expression for Q_4 and is generated by the Algorithm **SGQParser**. It corresponds to a fixed-point computation over the linear pattern $(a \cdot b \cdot c)$ (as employed by DD) and such plans are called *loop-caching* in literature as they enable re-use of the intermediate results for the base pattern $(a \cdot b \cdot c)$ [79]. **P1**, **P2** and **P3** 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 PATH of the proposed SGA. Fig. 8 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) as shown in Figure 9. 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.



(a) Window Size **(b) Slide**

Figure 6: The tail latency of each window slide and the aggregate throughput of the proposed streaming graph query processor with increasing (a) window size $\mathcal{T}$ and (b) slide interval β on SO graph.

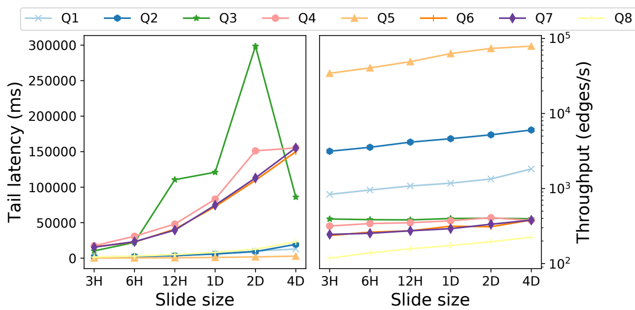
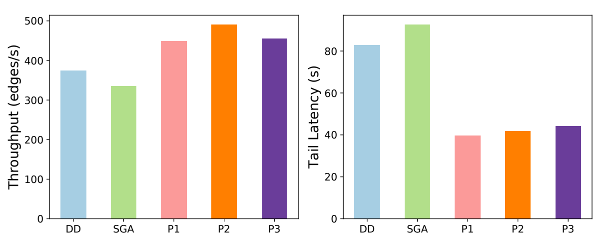
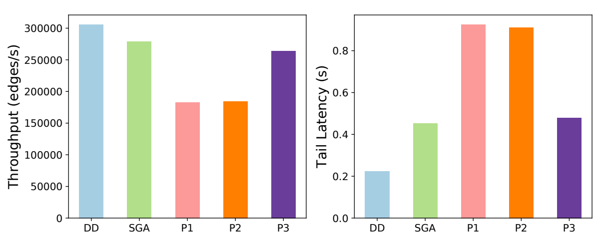


Figure 7: The tail latency of each window slide and the aggregate throughput of SGQ evaluation on DD with increasing slide interval β on SO graph.

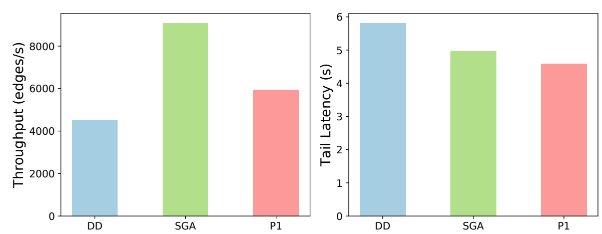


(a) SO

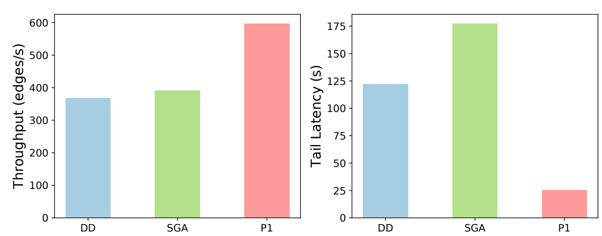


(b) LDBC-SF10

Figure 8: The throughput and tail latency of Q_4 on (a) SO and (b) LDBC graphs for DD, SGA and three other equivalent physical plans generated via SGA transformation rules (Section 5.4).



(a) Q_2



(b) Q_3

Figure 9: The throughput and tail latency of (a) Q_2, and (b) Q_3 on SO for DD, SGA for the default plan and an alternative equivalent physical plan generated via SGA transformation rules (Section 5.4).

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 system that consists of physical operator implementations as an embodiment of the proposed framework. The proposed SGA treats paths as first-class citizens and it provides the foundational framework to precisely describe the semantics of complex streaming graph queries that combine path navigation and subgraph pattern queries in a uniform manner. Experimental analyses on real-world and synthetic streaming graphs demonstrate the feasibility and the potential performance gains of our framework. Future research directions we consider are: (i) to design an SGA-based query optimizer that adaptively improves the query execution performance w.r.t. changing system conditions, and (ii) to extend our framework with attribute-based predicates to fully support the property graph model. There is, of course, much work to be done in developing alternative physical operator implementations.

REFERENCES

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

[2] 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.

[3] 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.

[4] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases.* Vol. 8. Addison Wesley.

[5] Amir Aghasadeghi, Vera Zaychik Moffitt, Sebastian Schelter, and Julia Stoyanovich. 2020. Zooming Out on an Evolving Graph.. In *Proc. 23rd Int. Conf. on Extending Database Technology.* 25–36.

[6] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed Evaluation of Subgraph Queries Using Worst-case Optimal and Low-Memory Dataflows. *Proc. VLDB Endowment* 11, 6 (2018), 691–704. https://doi.org/10.14778/3184470.3184473

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[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] Angela Bonifati and Stefania Dumbrava. 2019. Graph queries: From theory to practice. *ACM SIGMOD Rec.* 47, 4 (2019), 5–16.

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

[17] 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.

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

[19] 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.

[20] 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

[21] Sutanay Choudhury, Lawrence B. Holder, George Chin Jr, Khushbu Agarwal, and John Feo. 2015. A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs. In *Proc. 18th Int. Conf. on Extending Database Technology.* 157–168. https://doi.org/10.5441/002/edbt.2015.15

[22] Christian S Jensen James Clifford, Ramez Elmasri, Curtis Dyreson, Fabio Grandi Wolfgang K&fer Nick Kline, Nikos Lorentzos, Yamzis Mitsopoulos, Angelo Montanari, Daniel Nonen Elisa Peressi Barbara Pernici, John F Roddick Nandlal L Sarda, and Maria Rita Scalas Arie Segev. 1994. A consensus glossary of temporal database concepts. *ACM SIGMOD Rec.* 23, 1 (1994).

[23] 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.

[24] 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.

[25] David Ediger, Rob McColl, Jason Riedy, and David A Bader. 2012. Stinger: High performance data structure for streaming graphs. In *Proc. 2012 IEEE Conf. on High Performance Extreme Comp.* IEEE, 1–5.

[26] 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

[27] 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.

[28] 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.

[29] Thanaa M Ghanem, Moustafa A Hammad, Mohamed F Mokbel, Walid G Aref, and Ahmed K Elmagarmid. 2006. Incremental evaluation of sliding-window queries over data streams. 19, 1 (2006), 57–72.

[30] L. Golab. 2006. *Sliding Window Query Processing over Data Streams.* Ph.D. Dissertation. University of Waterloo.

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

[32] Lukasz Golab and M. Tamer Özsu. 2003. Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. In *Proc. 29th Int. Conf. on Very Large Data Bases.* 500–511.

[33] 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.

[34] 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.*

[35] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. In *Proc. ACM SIGMOD Int. Conf. on Management of Data.* 157–166.

[36] M. Hammad, W. Aref, M. Franklin, M. Mokbel, and A. Elmagarmid. 2003. *Efficient Execution of Sliding Window Queries over Data Streams.* Technical Report CSD TR 03-035. Purdue University.

[37] M. Hammad, M. Mokbel, M. Ali, W. Aref, A. Catlin, A. Elmagarmid, M. Eltabakh, M. Elfeky, T. Ghanem, R. Gwadera, I. Ilyas, M. Marzouk, and X. Xiong. 2004. Nile: a query processing engine for data streams. In *Proc. 20th Int. Conf. on Data Engineering.* 851.

[38] 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.

[39] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. 2016. Time-evolving graph processing at scale. In *Proc. 4th Int. Workshop on Graph Data Management Experiences and Systems.* 1–6.

[40] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data. In *Proc. ACM SIGMOD Int. Conf. on Management of Data.* 411–426.

[41] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. 2014. DBToaster: Higher-Order Delta Processing for Dynamic, Frequently Fresh Views. *VLDB J.* 23, 2 (2014), 253–278.

[42] 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.

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

[44] Jürgen Krämer and Bernhard Seeger. 2009. Semantics and implementation of continuous sliding window queries over data streams. *ACM Trans. Database Syst.* 34, 1 (2009), 1–49.

[45] 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

[46] Pradeep Kumar and H Howie Huang. 2019. GraphOne: A data store for real-time analytics on evolving graphs. In *Proc. 17th USENIX Conf. on File and Storage Technologies.* 249–263.

[47] Pradeep Kumar and H Howie Huang. 2020. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. *ACM Trans. Storage* 15, 4 (2020), 1–40.

[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] Youhuan Li, Lei Zou, M Tamer Özsu, and Dongyan Zhao. 2019. Time constrained continuous subgraph search over streaming graphs. In *Proc. 35th Int. Conf. on Data Engineering.* IEEE Press, 1082–1093.

[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] Ling Liu and M. Tamer Özsu (Eds.). 2009. *Encyclopedia of Database Systems*. Springer.

[52] Mengmeng Liu, N.E. Taylor, Wenchao Zhou, Z.G. Ives, and Boon Thau Loo. 2009. Recursive Computation of Regions and Connectivity in Networks. In *Proc. 25th Int. Conf. on Data Engineering*. 1108–1119. https://doi.org/10.1109/ICDE.2009.36

[53] Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-driven synchronous processing of streaming graphs. In *Proc. 14th ACM SIGOPS/EuroSys European Conf. on Comp. Syst.* 1–16.

[54] 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.

[55] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. 2020. Shared Arrangements: Practical Inter-Query Sharing for Streaming Dataflows. *Proc. VLDB Endowment* 13, 10 (2020), 1793–1806.

[56] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *Proc. 6th Biennial Conf. on Innovative Data Systems Research*.

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

[58] Vera Zaychik Moffitt and Julia Stoyanovich. 2017. Temporal graph algebra. In *Proc. 16th Int. Symposium on Database Programming Languages*. 1–12.

[59] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proc. 24th ACM Symp. on Operating System Principles*. 439–455.

[60] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew strikes back. *ACM SIGMOD Rec.* 42, 4 (Feb 2014), 5–16. https://doi.org/10.1145/2590989.2590991

[61] Milos Nikolic and Dan Olteanu. 2018. Incremental view maintenance with triple lock factorization benefits. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 365–380.

[62] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. 2020. Regular Path Query Evaluation on Streaming Graphs. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*. ACM, 1415–1430. https://doi.org/10.1145/3318464.3389733

[63] 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

[64] 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.

[65] 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.

[66] Juan L Reutter, Miguel Romero, and Moshe Y Vardi. 2017. Regular queries on graph databases. *Theory of Comput. Syst.* 61, 1 (2017), 31–83.

[67] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2020. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *VLDB J.* 29 (2020), 595—618. https://doi.org/10.1007/s00778-019-00548-x

[68] Semih Salihoglu and Nikolay Yakovets. 2019. Graph query processing. *Encyclopedia of Big Data Technologies* (2019), 890–898.

[69] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. 2016. Graphin: An online high performance incremental graph processing framework. In *European Conference on Parallel Processing*. Springer, 319–333.

[70] 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

[71] Feng Sheng, Qiang Cao, Haoran Cai, Jie Yao, and Changsheng Xie. 2018. GraPU: Accelerate streaming graph analysis through preprocessing buffered updates. In *Proc. 9th ACM Symp. on Cloud Computing*. 301–312.

[72] 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.

[73] 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.

[74] Tolga Urhan and M.J. Franklin. 2000. XJoin: A Reactively-Scheduled Pipelined Join Operator. *Q. Bull. IEEE TC on Data Eng.* 23 (2000), 27.

[75] 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.

[76] 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

[77] A.N. Wilschut and P.M.G. Apers. 1991. Dataflow query execution in a parallel main-memory environment. In *Proc. 1st Int. Conf. on Parallel and Distributed Information Systems*. 68–77.

[78] Peter T Wood. 2012. Query languages for graph databases. *ACM SIGMOD Rec.* 41, 1 (2012), 50–60.

[79] 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.

[80] Yuke Yang, Lukasz Golab, and M. Tamer Özsu. 2017. ViewDF: Declarative Incremental View Maintenance for Streaming Data. *Inf. Syst.* 71 (2017), 55–67. https://doi.org/doi.org/10.1016/j.is.2017.07.002

[81] 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