Models and Algorithms for Persistent Queries over Streaming Graphs

by

Anıl Paçacı

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2022

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: George Fletcher
Professor, Dept. of Mathematics and Computer Science,
Eindhoven University of Technology

Supervisor(s): M. Tamer Özsu
Professor, Cheriton School of Computer Science,
University of Waterloo

Internal Member: Ken Salem
Professor, Cheriton School of Computer Science,
University of Waterloo

Internal Member: Semih Salihoglu
Associate Professor, Cheriton School of Computer Science,
University of Waterloo

Internal-External Member: Lukasz Golab
Professor, Dept. of Management Sciences,
University of Waterloo

Other Member(s): Angela Bonifati
Professor, Dept. of Computer Science,
Lyon 1 University

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Some portions of the this theses are based on the peer-reviewed joint work with Prof. M. Tamer Özsu and Prof. Angela Bonifati, in which I am the first author and the primary contributor [131, 132].

Abstract

It is natural to model and represent interaction data as graphs in a broad range of domains such as online social networks, protein interaction data, and e-commerce applications. A number of emerging applications require continuous processing and querying of interaction data that evolves at a high rate, in near real-time, which can be modelled as a *streaming graph*. Persistent queries, where queries are registered into the system and new results are generated incrementally as the graph edges arrive, facilitate online analysis and real-time monitoring over streaming data. Processing persistent queries over streaming graphs combines two seemingly different but challenging problems: graph querying and streaming processing. Existing systems fail to support these workloads due to (i) the complexity of graph queries that feature recursive path navigations, subgraph patterns, and path manipulations, and (ii) the unboundedness and growth rate of streaming graphs that make it infeasible to employ batch algorithms. Consequently, a growing number of applications rely on specialized solutions tailored to specific application needs. This thesis introduces foundational techniques for efficient processing of persistent queries over streaming graphs to support this emerging class of applications in a principled manner.

The main contribution of this thesis is the design and development of a general-purpose streaming graph query processing framework. The novel challenges of persistent queries over streaming graphs dictate rethinking the components of the well-established query processor architecture, and this thesis introduces the models and algorithms to address these challenges uniformly. The central notion of *Streaming Graph Query* precisely characterizes the semantics of persistent queries over streaming graphs, making it possible to reason about the expressiveness and the complexity of queries targeted by the aforementioned applications. *Streaming Graph Algebra*, defined as a closure of a set of operators over streaming graphs, provides the primitive building blocks for evaluating and optimizing streaming graph queries. Efficient, incremental algorithms as the physical implementations of streaming graph algebra operators are provided, enabling streaming graph queries to be evaluated in a data-driven fashion. It is shown that the proposed algebra constitutes the foundational tool for the cost-based optimization of streaming graph queries by providing an algebraic basis for query evaluation. Overall, this thesis provides principled solutions to fundamental challenges for efficient querying of streaming graphs and describes the design and implementation of a general-purpose streaming graph query processing framework.

Acknowledgements

I would like to express that I am extremely grateful to my supervisor, M. Tamer Özsu, for his continuous guidance throughout my Ph.D. studies. I thank him for his patience in allowing me the freedom to become an independent researcher while making himself available to discuss research whenever I felt lost. His decades of experience in data management fostered my interest in every aspect of database systems, and his appreciation for preciseness encouraged me to focus on simplicity in both design and writing. I learned a lot from him, both professionally and personally, and this thesis would not have been possible without his support. It is a privilege to work with him, and I am honoured to call him my mentor and colleague.

I am also grateful to Angela Bonifati, who made significant contributions to this research. Her expertise in graph querying was invaluable in identifying interesting research problems and developing an appreciation of formal models. She has been a dear mentor and collaborator over the years that I worked on the research in this thesis.

I would also like to thank my committee members, Ken Salem, and Semih Salihoglu, for their advice, help, and support throughout my graduate studies. I am also thankful to my internal-external examiner Lukasz Golab and my external examiner George Fletcher for their invaluable feedback and comments.

I met many wonderful researchers and friends during my graduate studies, and their support helped me greatly both professionally and personally. I am thankful to Khuzaima Daudjee for countless discussions throughout these years and for the many hats he is willing to put on, a researcher, a teacher, and a friend. I was fortunate to overlap with a number of fellow graduate students during my time in the Data Systems Group, including Aida Sheshbolouki, Amine Mhedhbi, Brad Glasbergen, Kerem Akillioglu, Khaled Ammar, Libo Gao, Michael Abebe, Mustafa Korkmaz, Siddhartha Sahu, and Zeynep Korkmaz. Our discussions on research, graduate school, and everything else was unexpected but invaluable parts of my graduate school experience. The cheerful presence of my dear friends Amit Levy, Atulan Deep, Becca Meyers, Cagil Torgal, Berkan Alanbay, Camila Pavan, Cenk Koknar, Doruk Aksoy, Erik Hintz, Nate Braniff, Nathan Harms, Nupur Maheshwari, Raisa Sharmin, Ryan Kinnear, Sabria Farheen, Sajin Sasy, and Sengul Yildiz provided me with the encouragement to keep going.

I would like to thank my family for their support. My sister Yasemin and my parents, Mukadder and Fahrettin, provided unconditional love and endless support throughout my life. I am deeply grateful for your irreplaceable presence in my life.

Finally, I would like to express special thanks to my wife, Ali, for her kindness, support, understanding, and love. You had to go through the entire process with me, providing your unconditional support at every turn. I am forever thankful to you.

And most of all, I am forever grateful for the love and companionship of my sweet furry angel girl. You are the softest most precious thing in the whole universe.

Dedication

This thesis is dedicated to my little labrahuahua Kiwi and my love Ali.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Streaming Graph Processing

Graphs are used to model complex interactions in various domains ranging from social network analysis to communication network monitoring, from retailer customer analysis to bioinformatics. Graph processing systems empower such applications by enabling querying and processing of both the data stored in the graph and its topology, and they have gained significant attention both in the industry (e.g., JanusGraph[1], Neo4j[2], TigerGraph [49]) and academia [30, 172, 144]. Many real-world applications generate graphs over time as new edges are produced, resulting in *streaming graph*s. Consider an e-commerce application: each entity (such as users, messages, and items) can be modeled as a vertex, and each interaction (such as clicks, reviews, and purchases) can be modeled as an edge. The application receives and processes a sequence of graph vertices (users, items, etc.) and/or edges (as users purchase items, like content, etc.) – the model in this thesis is one of streaming edges with new vertices added implicitly. The graphs induced by these edges are unbounded, i.e., they continuously evolve over time, and their arrival rates can be very high. For example, Twitter's recommendation system ingests 12K events/sec on average [75], Alibaba transaction graph processes 30K edges/sec at its peak [137]. A recent survey [141] 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.

[1] https://janusgraph.org

[2] https://neo4j.com/

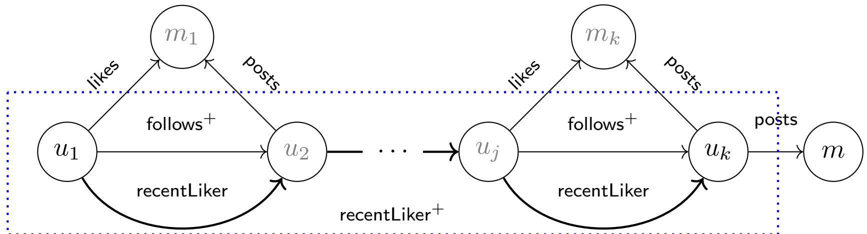


Figure 1.1: Complex graph pattern representing the query in Example 1

Persistent queries on streaming graphs enable users to continuously obtain new results on rapidly changing data, and existing graph DBMSs are not designed to keep up with the arrival rates of many real-world applications [134]. Existing graph DBMSs mostly follow the traditional database paradigm where data is persistent, and queries are transient. Consequently, they do not support persistent query semantics where queries are registered into the system and results are generated incrementally as the graph edges arrive. As demonstrated in the following examples, persistent queries on streaming graphs facilitate online analysis and real-time query processing, the latter being an important functionality of future graph processing engines [143].

Example 1. *In many online social networking applications, users post original content, sometimes link this to other users' content, and react to each other's posts – these interactions are modeled as a complex graph pattern as the one shown in Figure 1.1. A user u_2 is a* recentLiker *for another user u_1 if u_2 has recently liked posts that are created by u_1 and u_2, and u_1 are following each other. The goal of the recommendation service is to notify users, in real-time, of new content that is posted by others that are connected by a path of* recentLiker *relationship – these constraints are modeled as a complex graph pattern like the one shown in Figure 1.1. The service might provide the context for its recommendations by returning the full paths of people who are recent likers, such as the path between users u_1 and u_k. This real-time notification task is an example of a persistent query over the streaming graph of user interactions that returns the recommended content in real-time.*

Example 2 (Physical Contact Tracking)**.** *A number of Covid-19 contact tracing applications model interactions as graphs*[3] *where people are represented as vertices and an edge represents* contact *between two people if they visit the same space in the last 14 days (this is a simplification). The goal is to notify people of a potential chain of contact with someone*

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

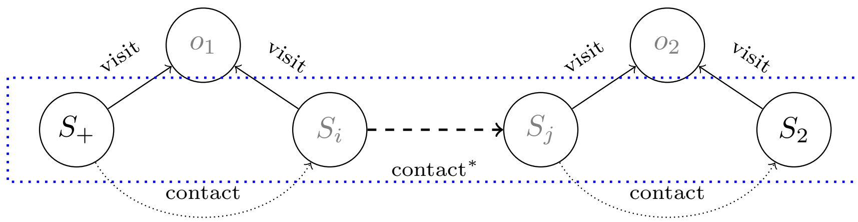


Figure 1.2: Complex graph pattern representing the query in Example 2

who tested positive. As shown in Figure 1.2 (bottom), the task of contact tracing is also a persistent graph query that returns the chain of contacts on a time window over this streaming graph of people's contacts.

The primary objective of this thesis is to study models, algorithms, and system architectures for the efficient processing of persistent graph queries over large streaming graphs with very high edge arrival rates. This thesis takes steps toward the development of a Streaming Graph Management System (SGMS) architecture by (i) identifying the research challenges for efficient querying of streaming graphs, and (ii) describing a principled design for a general-purpose streaming graph query processor that supports efficient execution of persistent queries over streaming graphs.

1.2 Research Challenges

Efficient querying of streaming graphs as in Examples 1 and 2 requires tackling together two already challenging problems: *graph querying* and *stream processing*. In particular, evaluating graph queries with complex patterns requires:

- **(R1)** subgraph queries that find matches of a given graph pattern (e.g. in Figure 1.1 the triangle pattern involving posts, likes and transitive closure of the follows relationship);
- **(R2)** path navigation queries that traverse paths based on user specified constraints (e.g. in Figure 1.1 arbitrary-length paths of the recentLiker relationship); and
- **(R3)** the ability to treat paths as first-class citizens of the data model, hence to manipulate and return paths (e.g. in Figure 1.1 the query returns the full paths of recentLiker).

Even in the context of one-time queries over static graphs, which has been the focus of existing research on graph querying, these are poorly addressed by existing graph database management systems (DBMS) and their query languages. Subgraph queries are akin to *conjunctive queries* (CQ), where data graph is represented as a binary relation, and existing relational techniques – multiway join algorithms in particular – can be employed to evaluate these queries. Navigational queries, on the other hand, cannot be easily expressed in the relational model [18], and alternative models for path navigation queries have long been studied in the context of semi-structured data and object DBMSs, and recently graph DBMSs. Graph DBMSs adopt *regular path queries* (RPQ) as the de-facto standard for navigational queries where path constraints are expressed as regular expressions over edge labels. The first two requirements are commonly addressed by closing the class of RPQ under conjunction and disjunction – this is known as *unions of conjunctive RPQs* (UCRPQ) [171, 30]. Although widely used in practice, UCRPQ is not considered to be a natural language to formulate many real-world graph queries due to its lack of algebraic closure and inability to express relations among paths [166, 138]. No existing work uniformly addresses all three requirements of graph querying.

Addressing the above requirements of graph querying become more complex in the context of persistent queries over streaming graphs, which is the focus of this thesis. Querying streaming data in real-time imposes additional and novel requirements:

- **(R4)** unbounded graph streams make 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.

Unboundedness and high-velocity arrivals have been studied within the context of the relational model but not within the context of streaming graphs. A common thread in these relational streaming systems is to restrict the scope of queries by evaluating them over a window of data from the stream using non-blocking implementations of existing relational operators, i.e., physical operator implementations that do not need the entire input to be available before producing the first result. The use of windowing constructs has been adapted for persistent query evaluation over RDF streams. There exists streaming RDF systems with various SPARQL extensions such as C-SPARQL [21], CQELS [100], SPARQL$_{stream}$ [36] and W3C proposal RSP-QL [48]. These systems are designed for SPARQLv1.0; consequently, they are limited to subgraph patterns in the form of *basic graph patterns* (BGP), and they do not support path navigation queries. Furthermore, query processing engines of these systems do not employ incremental operators. Most importantly, taken together, **R1-R5** form the challenges that any streaming graph query system should tackle, and there are no current systems that handle these requirements.

1.2.1 Limitations of Existing Systems

The proliferation of graph data has resulted in a number of graph processing systems in the past decade. Distributed graph processing engines (Pregel [110], GraphX [70], PowerGraph [69]) focus on running offline analytical workloads on static graphs, such as PageRank, connected component analysis etc. Graph DBMSs such as Neo4j, JanusGraph, and TigerGraph specialize in online querying and manipulation of graph-structured data. Their UCRPQ-based query languages lack algebraic closure and do not provide full composability, limiting reuse and decomposition of queries for query optimization, view-based query evaluation etc. Furthermore, the output of a path navigation query is typically a set of pairs of vertices that are connected by a path under the constraints of a given regular expression. Hence, these languages limit path navigation queries to boolean reachability without the ability to return and manipulate paths. G-CORE [11] addresses these limitations at the language specification level and has influenced the standardization efforts for a query language for graph DBMSs[4]. No existing work uniformly addresses all three requirements of graph querying. Furthermore, these systems predominantly employ the snapshot model, which assumes that graphs are static and fully available, and ad hoc queries reflect the current state of the database. Consequently, they neglect the continuous nature of streaming graph workloads described above.

Existing streaming systems, on the other hand, either (i) focus on one-dimensional streams in the relational model, which lacks path navigation features or (ii) provide generic computation models that are not optimized for the streaming graph workloads targeted in this thesis. Some streaming graph workloads are handled by non-streaming and specialized systems by performing repeated batch computations over windows of edges (e.g., [141]), because proper streaming solutions do not exist, not because this is the appropriate computation model. These specialized streaming solutions can provide satisfactory performance for the task at hand; however, they are not flexible to process any other workloads as underlying data structures and algorithms are designed for a particular task. A general query framework that addresses the above discussed requirements in a uniform and principled manner is currently missing, hindering the development of a general-purpose query processor for streaming graphs.

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

1.2.2 Long-term Vision

To address the aforementioned challenges of streaming graph querying, this thesis argues for a principled design of a general-purpose streaming graph query processing framework that consists of: (i) a formal query model and general-purpose algebra with well-founded semantics, and (ii) a data-driven query processor with efficient, non-blocking operator implementations. In analogy to traditional DBMSs, a general-purpose streaming graph query processing framework should provide the machinery to realize the well-known steps of query processing for streaming graph queries as follows:

1. a streaming graph query expressed in a declarative, high-level user language is translated into a query plan that consists of logical operators with precise semantics;

2. algebraic transformation rules are used to generate a set of equivalent plans for the given query and to explore the plan space through query rewrites;

3. a cost model that is based on the statistics of the underlying data and the system conditions is used by the optimizer to find a "good" plan among the set of equivalent plans for the given query;

4. the execution plan is built by selecting appropriate physical implementations of logical operators that are incremental and non-blocking;

5. the execution engine continuously executes the persistent query upon arrival of new edges to obtain new results.

The life cycle of a query in this reference query processor architecture is depicted in Figure 1.3. This mimics the query processor architecture of relational DBMSs with all its attendant advantages: a declarative query language lets users to specify *what* data to retrieve and leaves the issue of *how* to retrieve it to the query processor itself. This has profound impact on a query processor's design and performance. First, queries can be formulated using a high-level "declarative" interface in which the query processor can reason about their semantics and correctness. This provides the query processor the necessary degrees of freedom to optimize the execution of the query. Once the user query is mapped to an internal representation (e.g., an algebraic expression), the query optimization problem can be translated into a search problem: the query optimizer searches the space of equivalent plans guided by heuristics rules and cost estimations based on data statistics. Second, the decoupling of semantics from the implementation makes it possible to develop sound and efficient implementations for the query processing primitives and to compose query

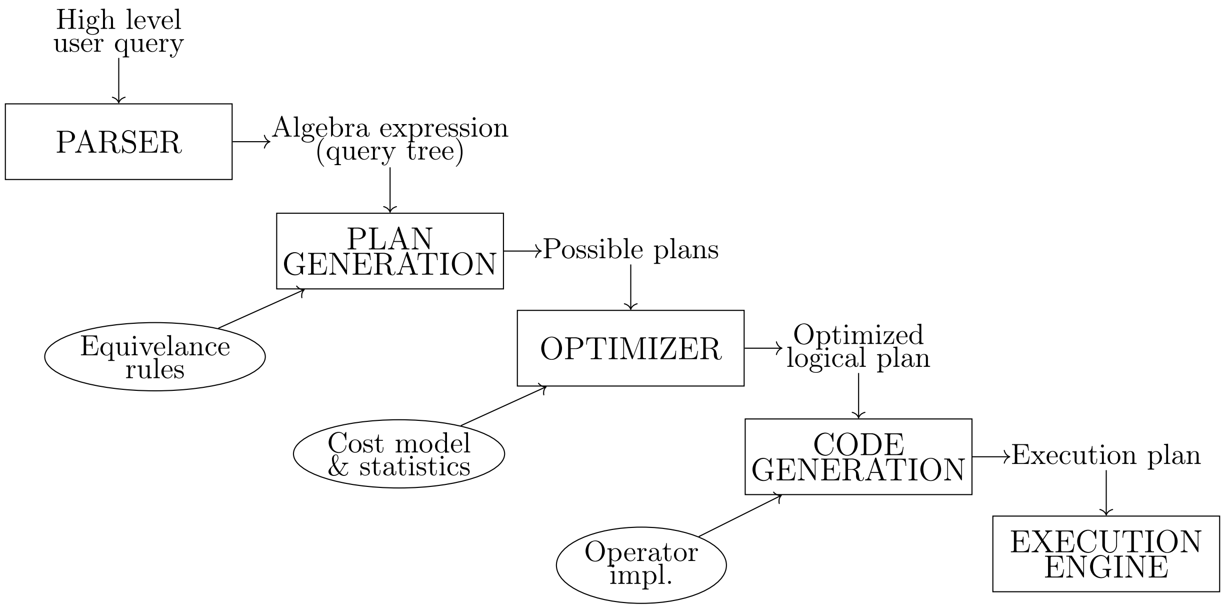


Figure 1.3: Reference architecture of a Streaming Graph Query Processor

processing pipelines using these primitives while ensuring correctness. The goal of this thesis is to identify and to tackle the research challenges to realize such a general-purpose query processing framework for streaming graph queries.

1.3 Contributions and Organization

This thesis presents the design and implementation of a general-purpose query processing framework for streaming graphs that addresses all of the above-discussed requirements (Section 1.2) in a uniform and principled manner. Models and algorithms introduced in this thesis are implemented as a part of the S-Graffito Streaming Graph Management System. [5] The remainder of this thesis is organized as follows:

- Chapter 2 presents background information and summarizes the related work on graph query processing and stream management.

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

- Chapter 3 establishes formal foundations for representing the class of queries targeted in this thesis. The streaming graph query (SGQ) model and its underlying streaming graph data model provide precise semantics of persistent graph queries with complex patterns. This chapter also provides concrete examples on how to formulate SGQ using a slight extension of G-CORE, a high-level, declarative graph query language.

- The ability to query, manipulate and return paths (**R2** & **R3**) is essential in graph querying, and Chapter 4 takes a closer look at the evaluation of path navigation queries over streaming graphs. The Regular Path Query (RPQ) model is used to formulate path constraints, and the design space of persistent RPQ evaluation algorithms is studied in two main dimensions: the path semantics they support and the result semantics based on application requirements. This chapter introduces the first streaming algorithms in the literature that cover the entire design space in a uniform manner.

- Chapter 5 focuses on the design and implementation of a query processor for evaluating SGQ, and it contains two contributions. First, it introduces the Streaming Graph Algebra (SGA), which consists of a set of primitive operators to formulate query evaluation plans for SGQ. An algorithm for translating SGQs into SGA expressions is also provided. Second, this chapter describes a prototype implementation of a streaming query processor based on SGA and provides a non-blocking, incremental algorithm as a physical implementation of each SGA operator.

- Chapter 6 studies the optimization of streaming graph queries based on the algebraic framework proposed in Chapter 5. This chapter begins with a set of transformation rules held in SGA that enables the systematic exploration of the plan space through query rewrites. It then introduces a cost model that quantifies the processing cost of SGA operators and expressions per unit time, capturing the data-driven nature of query evaluation over unbounded streaming graphs. Chapter 6 finally describes the cost-based optimization of SGQs and presents an exemplar optimizer implementation based on Apache Calcite.

The models and techniques presented in this thesis provide the foundational tools to achieve the long-term vision laid out in Section 1.2.2. Nonetheless, some aspects of this long-term vision are beyond the scope of this thesis.

First of all, the streaming graph query processing framework described in this thesis only focuses on the topology of the underlying graph and does not yet include property values. Incorporating attribute-based predicates to fully support to property graph model requires additional research. Second, the prototype streaming graph query processor described in Chapter 5 presents a single physical implementation for each SGA operator. Additional work that would enrich the implementation includes additional transformation rules for plan space enumeration and the development of alternative physical operators (Appendix A takes a step towards this direction and describes an alternative implementation for algorithms presented in Chapter 4). Third, the query processing framework presented in this thesis only deals with the optimization of SGQs under given system conditions. The performance of a query evaluation plan might change in the lifespan of a persistent query due to changes in the system conditions such as available memory and network bandwidth, changes in the arrival rate, or characteristics of the input streaming graph. *Adaptive* query processing techniques and their integration into the proposed framework are left as future work. Finally, techniques presented in this thesis are for the execution of streaming graph queries in centralized settings, and it is possible to develop techniques for scaling out for distributed environments (a comprehensive experimental study on streaming algorithms for graph partitioning and their impact on the performance of graph processing systems are presented in [133]).

Chapter 2

Background & Related Work

Querying streaming graphs combines two seemingly disparate but relevant problems. As described in Section 1.2, both stream processing and graph processing pose unique challenges, and they have been the focus of extensive research in the past two decades. This chapter first provides an overview of the relevant work on stream processing and graph processing systems. Then, it surveys existing models and algorithms for query processing over graph-structured data.

2.1 Stream Processing Systems

A data stream is defined as an ordered sequence of tuples where each tuple consists of a timestamp and a payload. Data streams are used in applications such as sensor networks, financial applications, and network monitoring, where the data arrivals are rapid, continuous, and possibly unbounded. The increasing need for real-time monitoring and analytics fostered the emergence of stream processing systems that are designed for rapid and continuous ingestion of data items. Unlike traditional DBMSs, these systems are generally push-based (data-driven), where queries are continuously evaluated as new data arrive. Stream processing systems are the focus of extensive research in the data management community, and the relevant literature can be broadly categorized into (i) stream management systems focusing on the relational model, and (ii) general-purpose stream processing engines.

Early research on stream processing mainly focuses on relational streams where the individual stream elements are in the form of relational tuples with a pre-defined schema.

This model represents streams as time-varying relations, and query semantics are described based on standard relational operators. STREAM [15] provides a SQL-like declarative query language called CQL, and provides three types of operators to extend relational semantics to the streaming model. Relation-to-relation operators correspond to operators of the standard relational algebra, relation-to-stream and stream-to-relation operators transform relation to streams and vice versa. Then, the semantics of CQL queries are described by:

1. converting streams to relational using windowing operators;
2. evaluating the query over relations;
3. and, translating the resulting relation back to streams.

Aurora [2] and its distributed version Borealis [1] employ a procedural approach to the formulation and execution of persistent queries over data streams. In Aurora, users directly specify queries by forming query plans through a graphical user interface. Aurora provides a set of operators that are streaming adaptations of their relational counterparts. For instance, the join operator takes a windowing specification (i.e., window size w) and produces a join result over two input tuples if the query predicate holds and tuples are at most w time units apart.

A common thread across all these is their relational core and non-blocking implementations of standard relational operators. Consequently, they do not support graph queries with recursive path navigations and patterns, which is the focus of this thesis. Nonetheless, non-blocking implementations of standard relational operators such as filter and join can be adapted for the continuous processing of graph queries (as will be described in Chapter 5).

Recent advances in cloud computing and the success of shared-nothing systems such as MapReduce have resulted in many Data Stream Processing Engines (DSPEs). They differ from their earlier counterparts as modern DSPEs are mostly scale-out solutions that do not necessarily offer the full set of DBMS functionality. These engines provide low-level system constructs such as data partitioning, scheduling, and operator queues upon which application developers can implement the business logic. Applications are expressed as *dataflow* graphs where the vertices represent computations and edges between them represent the flow of data, i.e., streams, between vertices. The majority of DSPEs (Flink [38], Storm [161], and its successor Heron [96]) do not support iterative (or recursive) computations in the streaming settings; consequently, they require the dataflow graph to be a directed acyclic graph (DAG). Naiad and its underlying Timely Dataflow computation model [125] relax this DAG assumption by allowing loops in the dataflow graph. As such, the Timely Dataflow model is able to represent and execute arbitrary (possibly cyclic) dataflow graphs. Chapter 5 shows that the class of queries targeted in this thesis can be represented as cyclic dataflow graphs and can be processed by such a system.

There are some recent efforts to bridge the gap between these two classes of systems by providing support for the relational model and declarative SQL-like queries (e.g., Spark Structured Streaming, Flink SQL, and Materialized). Nonetheless, existing DSMSs and DSPSs either (i) focus on the relational model, which lacks path navigation features, or (ii) provide general-purpose computation models that are not optimized for the streaming graph workloads targeted in this thesis.

2.2 Graph Processing Systems

A number of graph processing systems have been introduced in the last decade. These are typically divided into graph analytics engines and graph DBMSs based on the workloads they target. Systems in the former category focus on offline graph analytics, and the systems in the latter category specialize in online graph queries, similar to the OLAP vs. OLTP distinction in relational DBMSs. The rest of this section first provides an overview of modern graph processing systems following this classification, then discusses existing systems that are specifically designed for processing streaming graphs.

2.2.1 Graph Processing Engines

Many analytical graph workloads are iterative, where the entire graph is processed in each iteration until a fixpoint is reached. Examples include graph algorithms like PageRank, weakly connected components; analytical tasks such as triangle counting; and machine learning and data mining algorithms such as belief propagation and collaborative filtering. Existing OLAP systems based on the relational model are ill-suited for such computations due to the irregular, highly interconnected structure of real-world graphs that leads to many-to-many joins and an explosion of intermediate results. Furthermore, iterative algorithms cannot be easily represented in relational query languages. Graph processing systems (e.g., Pregel [110], Giraph [14], PowerGraph [69] and PowerLyra [41]) specialize on such workloads by providing (i) programming APIs that make is easy to express iterative computations, and (ii) computational models that are optimized for these tasks.

These systems are predominantly scale-out processing engines that do not necessarily provide full database management functionality. They follow the Bulk Synchronous Parallel (BSP) [164] model, where the computation is performed iteratively through user-defined vertex functions. Two popular programming models are *vertex-centric block synchronous* where vertices push their state along the edges of the graph at the end of each iteration, and *vertex-centric Gather, Apply, Scatter* (GAS) where the state is pulled (rather than pushed) by vertices at the beginning of each iteration. Due to their support for iterations, graph queries with subgraph patterns and path navigation queries targeted in this thesis can be expressed as BSP computations [56, 169]. Nonetheless, these systems are not suitable for continuously processing such queries over streaming graphs due to their offline nature. A comprehensive analysis of graph processing systems can be found in surveys [117] and performance studies [81, 10].

2.2.2 Graph Database Management Systems

The other important class of graph workloads is *online queries* that focuses on interactive querying and manipulation of the underlying graph-structured data. Unlike analytic workloads, online graph queries are usually not iterative and require access to only a portion of the graph (e.g., reachability queries, pattern matching, neighbourhood traversals). Although the relational model can represent graph-structured data, traditional RDBMSs fail to provide intuitive interfaces and efficient operations for queries that involve path navigations. In addition, representing highly connected data in the relational model results in a large amount of many-to-many relations, which can produce complex, join-heavy SQL statements for graph queries. Graph DBMSs model and store the graph-structured data by indexing the adjacency information for each entity in adjacency lists, allowing intuitive expression and efficient processing of online graph workloads.

Two alternative data models are commonly used in graph DBMSs. With the rise of semantic web, one line of work use the RDF model where the data is modeled as a directed, edge-labeled multi-graph (e.g., Virtuoso [54], RDF-3X [127], gStore [153, 176]). RDF DBMSs use a standardized query language, SPARQL, which provides capabilities for expressing subgraph patterns (basic graph patterns – BGPs) and path reachability queries (property paths in SPARQL v1.1). The second class of graph DBMSs such as Neo4j, JanusGraph, and Oracle's Graph Database employ the property graph model (PGM). Property graphs are directed, edge-labeled multi-graphs where each edge and vertex might be associated with an arbitrary number of key-value pairs, i.e., properties [30]. Unlike RDF systems with a structured, standardized query language, these systems lack a standardized interface. Most vendors have proprietary APIs and languages with variances in features and capabilities. JanusGraph uses the Gremlin query language from Apache Tinkerpop[1], which is a procedural query language that allows users to describe how a query is evaluated by chaining operators. Neo4j's Cypher is a declarative language that uses ASCII-art style pattern matching as its building blocks. Similarly, Oracle's PGQL is a declarative language with SQL-like constructs. There exist open-source efforts to unify the graph query language space: G-CORE [11] is a graph query language proposal that aims to capture and to extend core functionalities found in existing languages, and GQL[2] is a recent standardization effort for a standalone query language for PGM, similar to SQL for the relational model.

Although most graph DBMSs support updates, these systems predominantly employ the snapshot model, which assumes that the underlying graph is fully available, and ad hoc queries reflect the current state of the database. Consequently, these systems and their query languages neglect the continuous nature of streaming graph workloads targeted in this thesis.

2.2.3 Streaming Graph Systems

Streaming graph systems have emerged to enable the processing of evolving graphs, addressing the limitations of graph processing engines and graph DBMSs. Existing work on streaming graph systems, by and large, focuses on either (i) maintenance of graph snapshots under a stream of updates for iterative graph analytic workloads or (ii) specialized systems for persistent query workloads that are tailored for the task in hand. One of the earlier systems in the first category, STINGER [52], proposes an adjacency list-based data structure optimized for fast ingestion of streaming graphs. GraphOne [97, 98] uses a novel versioning scheme to support concurrent reads and writes on the most recent snapshot of the graph. Analytic engines such as GraphIn [149] and GraphTau [86] extend the popular vertex-centric model with incremental computation primitives to minimize redundant computation across consecutive snapshots. More recently, systems such as GraPu [154] and GraphBolt [113] introduce novel dependency tracking schemes to transparently maintain results of graph analytic 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. The unbounded nature of streaming graphs and the need for real-time answers on recent data make it infeasible to employ snapshot-based techniques for the class of queries targeted in this thesis.

Driven by the performance requirements of real-world applications, existing work on

[1] https://tinkerpop.apache.org

[2] https://www.gqlstandards.org/

persistent query processing over streaming graphs are highly specialized systems. Twitter's streaming graph systems, GraphJet [151] and RecService [75], focus on real-time pattern detection for a fixed set of pre-defined graph patterns. Similarly, Alibaba's fraud detection system relies on detecting cycles over the streaming graph of user interactions on its e-commerce platform [137]. Although these solutions provide satisfactory performance for the task at hand, they lack the flexibility to support a wide range of real-world scenarios.

Finally, there has been a significant amount of work on various aspects of RDF stream processing[3]. Calbimonte [35] designs a communication interface for streaming RDF systems based on the Linked Data Notification protocol. TripleWave [116] focuses on the problem of RDF stream deployment and introduces a framework for publishing RDF streams on the web. EP-SPARQL [13] extends SPARQLv1.0 for reasoning and a complex event pattern matching on RDF streams. Similarly, SparkWave [93] is designed for streaming reasoning with schema-enhanced graph pattern matching and relies on the existence of RDF schemas to compute entailments. None of these are processing engines, so they do not provide query processing capabilities. Contributions of this research are orthogonal to existing work on streaming RDF systems. However, techniques proposed in this thesis can be integrated into these systems as they incorporate query processing capabilities.

2.3 Graph Querying

This section first surveys the landscape of graph query languages, then provides an overview of the relevant work on algorithmic techniques for graph querying.

2.3.1 Graph Query Languages

Graph query workloads targeted by graph DBMSs feature subgraph patterns and navigations, commonly modelled using conjunctive queries (CQ) and regular path queries (RPQ), respectively. The UCRPQ model – *unions of conjunctive RPQs* – provides the ability to express path navigation and subgraph pattern queries uniformly by closing the class of RPQ under disjunction and conjunction [18]. Conceptually, a UCRPQ query is defined by replacing edge labels of a conjunctive query (subgraph pattern query) with regular expressions (path navigation query). Graph query languages employed by existing systems (Section 2.2.2) are predominantly based on the UCRPQ model, with slight differences

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

in their implementation details. For instance, Neo4j's Cypher is based on isomorphism-based matching semantics and limits path navigations to reachability queries over single edge labels. Oracle's PGQL provides support for both homomorphism and isomorphism-based matching semantics. The original SPARQL standard only features subgraph pattern queries via homomorphism, and SPARQL v1.1 incorporates property paths consistent with the RPQ model. Despite its widespread adoption, the UCRPQ model is not considered to be a natural language to formulate many real-world graph queries due to its lack of algebraic closure and inability to express relations among paths [166, 138]. G-CORE [11] addresses these limitations at the language specification level and has influenced the standardization efforts for a graph query language.[4] It is based on the subset of Datalog called Regular Queries (RQ) – non-recursive Datalog extended with the transitive closure over binary relations. It has been recently shown that RQ is computationally well-behaved, i.e., its evaluation is tractable under data complexity, and the containment is decidable [138]. As RQ properly generalizes UCRPQ and has the property of algebraic closure, it is considered a natural candidate to formulate graph queries. Nonetheless, all these focus on ad-hoc queries over static graphs and do not provide support for formulating persistent queries over streaming graphs.

There exists streaming RDF systems with various SPARQL extensions for persistent query evaluation over RDF streams such as SPARQL_{stream} [36], C-SPARQL [21], CQELS [100] and W3C proposal RSP-QL [48]. However, these systems are designed for SPARQLv1.0, and they do not have the notion of *property paths* from SPARQLv1.1. Thus one cannot formulate recursive path queries such as RPQs that cover more than 99% of all recursive queries found in massive Wikidata query logs [32]. The lack of property path support of these systems is previously reported by an independent RDF streaming benchmark, SR-Bench [175] (see Table 3 in [175]). Furthermore, query processing engines of these systems do not employ incremental operators, except Sparkwave [93] that focuses on stream reasoning.

Similar to the streaming graph query processing framework proposed in this thesis, some systems employ an algebraic approach to graph query processing. TriAL [105] is a triple-based query algebra designed for one-time navigational queries over static triple-stores. Nevertheless, it only focuses on path navigation queries and cannot be used as a standalone graph query language. Temporal Graph Algebra (TGA) [123] adapts temporal relational operators in the context of PGM to support analytics over evolving graphs. Its implementation on Spark introduces physical operators for graph analytics [6]. However, it is designed for exploratory graph analytics over the entire history of changes. In contrast, SGQ and the corresponding SGA proposed in this thesis focus on persistent graph queries

[4] see https://www.gqlstandards.org/

over (potentially unbounded) streaming graphs, and they can express complex graph patterns expressed in high-level user languages (Section 3.3.2).

2.3.2 Algorithms for Graph Querying

Graph queries in general feature subgraph pattern queries and path navigation queries. Subgraph pattern queries are akin to conjunctive queries in the relational model. They can be represented as multi-way join queries, which have been extensively studied in the context of relational databases and graph querying. Traditionally, such multi-way joins are evaluated by a series of binary joins, which is recently shown to be sub-optimal for cyclic subgraph queries as the size of intermediate results can be asymptotically larger than the final output [129]. Recent *worst-case optimal* (WCO) join algorithms attain worst-case optimality by joining all relations at once for each join attribute instead of a series of pairwise joins [128]. EmptyHeaded [4, 3] combines WCO joins with binary joins using the generalized hyper-tree decomposition of query graphs. GraphFlowDB [88, 122] further extends the space of such hybrid plans and introduces an adaptive, cost-based optimizer for subgraph pattern queries that combines WCO and binary joins in a principled manner.

RPQ is the de-facto formalism for path navigation queries in practical graph query languages, striking a balance between expressiveness and computational complexity [12, 30, 156, 11]. The research on RPQs focuses on various problems such as containment [37], enumeration [114], learnability [28]. Most related to the query processing framework studied in this thesis is the RPQ evaluation problem. The seminal work of Mendelzon and Wood [121] shows that RPQ evaluation under simple path semantics is NP-hard for arbitrary graphs and queries. They identify the conditions for graphs and regular languages where the RPQ evaluation problem is computable in polynomial time. Bagan et al. [20] prove a trichotomy, and establish a comprehensive classification of the complexity of RPQ evaluation under simple path semantics. They introduce a maximal class of regular languages, C_{tract}, for which the problem of RPQ evaluation under simple path semantics is tractable and NP-complete for any language that does not belong to C_{tract}.

RPQ evaluation strategies follow two main approaches: automata-based and relational algebra-based. **G** [44], one of the earliest graph query languages, builds a finite automaton from a given RPQ to guide the traversal of the graph. Kochut et al. [92] study RPQ evaluation in the context of SPARQL and propose an algorithm that uses two automatons, one for the original expression and one for the reversed expression, to guide a bidirectional BFS on the graph. Addressing the memory overhead of BFS traversals, Koschmieder et al. [94] decompose a query into smaller fragments based on rare labels and perform a series of bidirectional searches to answer individual subqueries. A recent work by Wadhwa et al. [168] uses random walk-based sampling for approximate RPQ evaluation. The other alternative for RPQ evaluation is α-RA which extends the standard relational algebra with the α operator for transitive closure computation [7]. α-RA-based RPQ evaluation strategies are used in various SPARQL engines [54]. Histogram-based path indexes on top of a relational engine can speed-up processing RPQs with bounded length [58]. α-RA-based RPQ evaluation is not suitable for persistent RPQ evaluation on streaming graphs as it relies on blocking join and α operators. Yakovets et al. [172] show that these two approaches are incomparable, and they can be combined to explore a larger plan space for SPARQL evaluation. Various formalisms such as pebble automata, register automata, and monadic second-order logic with data comparisons extend RPQs with data values for the property graph model [104, 106]. Although RPQs and corresponding evaluation methods are widely used in graph querying [12, 11, 54], all of these works focus on static graphs.

2.3.3 Incremental View Maintenance

A persistent query over sliding windows can be formulated as an Incremental View Maintenance (IVM) problem. The view definition is the query itself, and window movements correspond to updates to the underlying database. In the IVM approach, 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* [78] 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 [91] introduces the concept of *higher-order* views for group-by aggregates and represents each view definition using a hierarchy of views that reduces the overall maintenance cost. F-IVM [130] further extends higher-order views with a factorized representation of these views to reduce the amount of state and the computation cost. ViewDF [173] 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 addressed in this thesis, primarily because of the potentially infinite results for recursive graph queries.

The classical *DRed* algorithm [78] 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 [108]. Thus, it is possible to adapt recursive IVM techniques to evaluate streaming graph queries, but these ignore the structure of graph queries and inherent temporal patterns of streaming graphs. Techniques studied in the thesis, in contrast, exploit the query structure to minimize the cost of persistent graph query evaluation over streaming graphs.

2.3.4 Dynamic & Streaming Graph Algorithms

The theoretical research on streaming graphs primarily focuses on maintaining approximations of structural graph properties such as triangle count and spanners (see [118] for an extensive survey). Earlier work on streaming algorithms for graphs is motivated by the limitations of main memory, focusing on modelling graph algorithms in the streaming settings. Many graph problems are hard in sublinear space, i.e., exact solutions of these algorithms cannot be computed without storing all the vertices in the graph (which might not be feasible for unbounded streams). Consequently, researchers have focused on the *semi-streaming* model for the study of streaming graph algorithms where the set of vertices can be stored in memory but not the set of edges [126]. There exist a large body of work on approximating graph algorithms in the semi-streaming model including PageRank estimation [145], graph matching [57], finding common neighbours [34], [23, 33] (see [118] for a survey). This thesis adopts the windowed evaluation model to process unbounded streams with bounded memory, a standard solution in streaming systems for bounding the space requirement and restricting the scope of queries to recent data, a desired feature in many applications[65, 17]. Compared to approximation-based methods, window-based query evaluation enables exact query answers w.r.t. window specifications. Nonetheless, these streaming approximation techniques are orthogonal to the query processing algorithms studied in this thesis, and they can be incorporated to provide support for approximate query processing.

Many graph problems are also studied in the dynamic graph model, where algorithms may use enough memory to store the entire graph and compute how the output changes as the graph is updated. Examples include connectivity [89], shortest paths [25], reachability [139], transitive closure [99]. TurboFlux [90] is a specialized subgraph pattern matching system that incrementally maintains matching results over a dynamic graph that is updated with edge arrivals. GraphFlow [88] is an active graph database that employs the delta decomposition technique [26] for incrementally maintaining subgraph pattern queries using the worst-case-optimal Generic Join [129] algorithm. Ammar et al. [9] adapt worst-case optimal join and delta decomposition to the dataflow computation model for continuous subgraph pattern matching in distributed settings.

Fan et al. [55] propose a characterization of various graph problems in the dynamic model based on the complexity of incrementally maintaining query results over dynamic graphs, including subgraph isomorphism that can be used for evaluating subgraph pattern queries and RPQ that can be used for evaluated recursive path queries. They show that most graph problems are unbounded under edge updates, i.e., the cost of computing changes to query answers cannot be expressed as a polynomial of the size of the changes in the input and output. They propose alternative characterizations for the effectiveness of dynamic graph problems and show that efficient dynamic algorithms are possible. Specifically, they prove that RPQ is bounded relative to its batch counterpart; the batch algorithm can be efficiently incrementalized by minimizing unnecessary computation.

Chapter 3

Streaming Graph Queries

3.1 Introduction

The unsuitability of existing graph DBMSs for querying streaming data has motivated the design of specialized systems addressing singular features and application needs. For instance, a number of specialized algorithms focus on evaluating subgraph queries on streaming graphs [9, 103, 137, 90, 42]. However, a general-purpose model and framework that unifies existing graph querying and streaming querying functionality in a principled manner is missing. To develop a general-purpose query processing framework for streaming graphs, it is crucial to describe the precise semantics of the target query class. This chapter describes the *Streaming Graph Query* (SGQ) model that constitutes the formal basis of the query processing framework introduced in this thesis. SGQ describes the precise semantics of persistent query evaluation over streaming graphs, the class of queries targeted in this thesis. Section 3.1.1 begins by analyzing the existing work on graph query models w.r.t. the requirements for querying streaming graphs outlined in Section 1.2. Section 3.2 presents the *Streaming Graph* data model and Section 3.3 introduces the formal SGQ model. Section 3.4 concludes this chapter by summarizing its contributions and by providing an overview of the role of SGQ model in the streaming graph query processing framework introduced in this thesis.

3.1.1 Analysis of Existing Graph Query Languages

Querying graph structure requires combining path navigation and subgraph pattern matching features (**R1** & **R2**), as discussed in Section 1.2. Augmenting the class of RPQ with

Operation	SPARQL v1.1	Cypher	G-CORE	Static	Streaming
Reachability	✓	✓	✓	✓	✓
Endpoints	✓	✓	✓	✓	✓
Named Paths	×	✓	✓	✓	✓
Returning Paths	×	✓	✓	✓	✓[1]
Storing Paths	×	×	✓	✓	✓

Table 3.1: Summary of path operations in practical graph query languages.

disjunction and conjunction results in UCRPQ – unions of conjunctions of RPQ [18]. Conceptually, a UCRPQ query is defined by replacing edge labels of a conjunctive query with regular expressions. It is easy to see that every subgraph pattern query is a UCRPQ query where query edges are mapped to edges in the data graph. Although UCRPQ forms the basis of earlier graph query languages such as Cypher and SPARQL v1.1 [12], it is not considered to be a natural language to formulate many real-world graph queries due to its lack of algebraic closure and inability to express relations among paths [166, 138]. Paths provide higher-level abstractions to model complex real-world relationships, and returning and manipulating paths is a fundamental operation in graph querying (**R3**). Consequently, this section focuses on the semantics of path querying features in existing graph languages SPARQL v1.1, Cypher and G-CORE and describes their implications on the complexity of query evaluation in the static and the streaming contexts. Table 3.1 provides an overview of path querying features in existing graph query languages.

Path navigation queries of SPARQL v1.1, i.e., property paths, originally adopted the arbitrary path semantics with a counting based approach, where the duplicity of a result pair is preserved. Subsequent intractability results prove that query evaluation under such semantics is not feasible in practice [16]. In general, path query evaluation based on counting semantics is intractable; therefore, W3C has adapted existential, set-based semantics for SPARQL property paths [156]. Yet, path navigation queries in SPARQL v1.1 only feature reachability semantics, e.g., test if there exists a path between two vertices satisfying user-specified conditions, and do not provide a mechanism to represent paths (i.e., named path with variable assignments or returning paths).

Similarly, Cypher is based on UCRPQs [12], yet, it uses no-repeated-edge (simple trail) semantics for path navigation queries [60]. RPQ evaluation under simple trail semantics is

[1] As described in detail in Chapter 4, parent pointers in Δ tree index can be utilized to construct the resulting path with $\mathcal{O}(|p|)$ cost where $|p|$ is the length of the corresponding path.

an NP-complete problem, even in data complexity [115]. Cypher addresses the intractability of RPQ evaluation problem by limiting the use of Kleene star over only a single edge label [60]. Such simple path expressions are known to belong to the tractable class of *restricted regular expressions*, whose evaluation under such semantics is tractable in data complexity [121]. Like SPARQL, Cypher queries are not composable, i.e., the output of a Cypher query over a graph is a table. This is due to the lack of algebraic closure of the class of UCRPQs. Unlike SPARQL, Cypher queries might return and manipulate paths through the use of named paths. Although this facilitates the support for a wider class of path navigation features, it is a potential source of intractability. A path query in Cypher returns all matches for the given variable-length path pattern, not just its existence [59]. This evaluation semantics corresponds to the exhaustive enumeration of all paths, which might be infinitely many in the presence of cycles. Although Cypher's simple trail semantics combined with its restrictions on the use of Kleene star ensure finiteness, exhaustive enumeration of all paths might take exponential time in the size of the graph.

Unlike the previous two, G-CORE extends the property graph model with objectified paths [30], i.e., treating paths as first-class citizens. A G-CORE query can explicitly materialize paths in the results, so-called *stored paths*. Stored paths are first-class citizens of the data model, i.e., they have labels and properties similar to vertices and edges, and subsequent queries can manipulate stored paths. G-CORE queries allow arbitrary operations on stored paths as these are materialized in the graph, and they do not need to be computed. *Virtual paths*, on the other hand, refer to paths that are computed during the lifetime of a query and correspond to paths in other languages that do not support stored paths, i.e., Cypher and SPARQL. G-CORE, by default, uses shortest-path based arbitrary path semantics and therefore has tractable data complexity. Unlike Cypher, G-CORE explicitly avoids exhaustive enumeration of all paths due to an infinite amount of results [11]. Hence, it does not support path operations that require exhaustive enumeration. In addition, G-CORE is based on the class of RQs and inherits its algebraic closure. Hence, G-CORE is a composable query language, and it supports view definitions, and path queries over derived edges. Addressing all three requirements on graph querying (**R1** - **R3**) makes G-CORE suitable to express the class of queries target in this thesis, as discussed later in Section 3.3.2.

To date, there has been a little work on extending these languages to the streaming model except SPARQL extensions for persistent query evaluation over RDF streams. Streaming RDF query languages C-SPARQL [21], CQELS [100], SPARQL$_{stream}$ [36] and RSP-QL [48] are the most similar to the class of queries targeted in this thesis, but they are designed for SPARQLv1.0. Consequently, they cannot formulate path expressions such as RPQs that cover a significant portion of all recursive queries found in Wikidata query logs

Table 3.2: Notations used throughout the thesis.

Σ	Set of labels
ψ	Mapping from edges to pairs of vertices
ρ	Mapping from paths to sequence of edges
ϕ	Mapping from edges and paths to labels
$[ts, exp)$	Half-open validity interval
$u \xrightarrow{p} v$	Path p between vertices u and v
$\tau_t(S)$	Snapshot of a streaming graph s at time t
$\mathcal{Q}$	Streaming graph query (SGQ)
$\mathcal{Q}^O$	One-time graph query
ω	Window size
β	Optional window slide interval
Φ	Boolean predicate
R	Regular expression over Σ
$\mathcal{W}_{\omega,\beta}$	Windowing operator WSCAN
σ_Φ	Selection operator FILTER
$\bowtie_\Phi^{src,trg,d}$	Subgraph pattern operator PATTERN
$\mathcal{P}_R^d$	Path navigation operator PATH

by a recent analysis [32]. Furthermore, query processing engines of these systems do not employ incremental operators, except Sparkwave [93] that focuses on stream reasoning.

3.2 Data Model: Streaming Graphs

3.2.1 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, $\psi : E \rightarrow V \times V$ is an incidence function and $\phi : E \rightarrow \Sigma$ is an edge labelling function.*

Definition 2 (Path and Path Label)**.** *Given $u, v \in V$, a path p from u to v in graph G is a sequence of edges $u \xrightarrow{p} v : \langle e_1, \cdots, e_n \rangle$ such that $x_i, y_i \in V$ are endpoints of an edge $e_i \in E$ and $y_i = x_{i+1}$ for $i \in [1, n)$. The label sequence of a path p is defined as the concatenation of edge labels, i.e., $\phi^p(p) = \phi(e_1) \cdots \phi(e_n) \in \Sigma^*$.*

$\mathbb{T} = (\mathcal{T}, \leq)$ denotes a discrete, total ordered time domain and $t \in \mathcal{T}$ is a timestamp that denotes a time instant. Without loss of generality, the remainder of the thesis 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 *and* trg *are vertices,* l *represents the label of the sge, and* $t \in \mathcal{T}$ *is the event (application) timestamp assigned by the external data source.*[2]

Definition 4 (Input Graph Stream). *An input graph stream is a continuously growing sequence of streaming graph edges* $S^I = \left[sge_1, sge_2, \cdots\right]$ *where each* sge_i (src_i, trg_i, l_i, t_i) *represents an edge* $e \in E$ *labeled* $l_i \in \Sigma$ *between vertices* $src_i, trg_i \in V$ *and sges are non-decreasingly ordered by their timestamps.*[3]

Figure 3.1 depicts an excerpt of the input graph stream of the application in Example 1, where each tuple represent an interaction between two vertices and each timestamp represent the time instant that the interaction occurs.

Input graph streams represent external data sources that generate and provide the system with the graph-structured data. The 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 at which the interaction represented by the sge occured [137, 131, 103]. Alternatively, intervals are used to represent the period of validity of sges, because using *validity intervals* leads to a succinct representation and simplifies operator semantics by separating the specification of window constructs from operator implementation. As an example, each sge with timestamp t can be assigned a validity interval $[t, t+1)$ that corresponds to a single time unit with smallest granularity that cannot be decomposed into smaller time units.[4] Similarly, an sge $e = (u, v, l, [ts, exp))$ with a validity interval is equivalent to a set of sges $\{(u, v, l, t_1), \cdots, (u, v, l, t_n)\}$ where $t_1 = ts$ and $t_n = exp - 1$. Time-based sliding windows (to be precisely defined momentarily in Section 3.3.1) are used to assign validity intervals based on the windowing specifications of a given query.

[2]It is assumed that sges are generated by a single external data source and arrive in order; out-of-order arrival is left as future work.

[3][] denotes ordered streams throughout the thesis

[4]Commonly referred as `NOW` windows as described in Chapter 5.

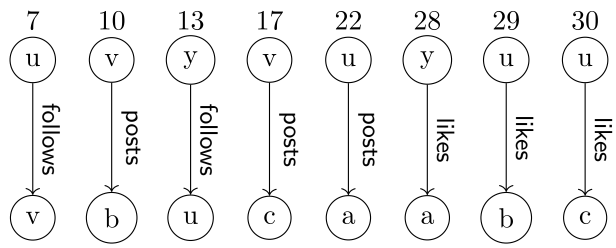


Figure 3.1: The input graph stream from an online social network of Ex. 1.

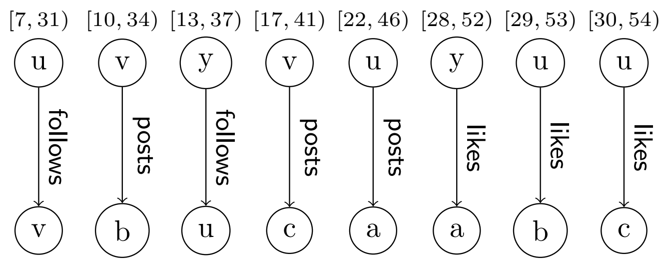


Figure 3.2: The streaming graph obtained from the input graph stream in Figure 3.1 where the validity interval of each element is set based on a $24h$ window.

3.2.2 Streaming Graphs

The discussion in this section focuses on the logical representation of streaming graphs that is used throughout the thesis. Because the class of queries targeted in this thesis feature both subgraph patterns and path navigations, queries can return paths (**R3**). Consequently, the directed labeled graph model is extended 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, queries

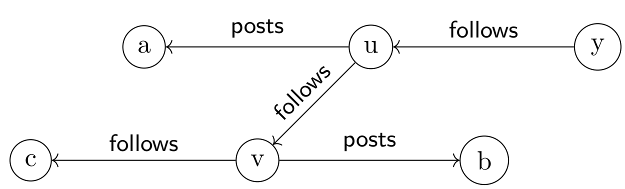


Figure 3.3: The snapshot graph of the streaming graph in Figure 3.2 at $t = 25$

that manipulate and return paths produce outputs over the same data model, enabling composability. In addition, it enables queries with complex graph patterns that stitch edges and paths as will be shown in Chapter 5.

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,* $\psi : E \to V \times V$ *is an incidence function,* $\rho : P \to E \times \cdots \times E$ *is a total function that assigns each path to a finite, ordered sequence of edges in* E*, and* $\phi : (E \cup P) \to \Sigma$ *is a labeling function, where images of* E *and* P *under* ϕ *are disjoint, i.e.,* $\phi(E) \cap \phi(P) = \emptyset$*.*

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 $i \in [1, n)$, $\psi(e_i) = (src_i, trg_i)$, $trg_i = src_{i+1}$, 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 $P = \emptyset$. The notion of streaming graph edges (Definition 3) is generalized as follows:

Definition 7 (Streaming Graph Tuple)**.** *A streaming graph tuple (sgt) is a quintuple* $sgt = (src, trg, l, [ts, exp), \mathcal{D})$ *where* src *and* trg *are vertices,* l *is the label of the sgt, and* $[ts, exp) \in \mathcal{T} \times \mathcal{T}$ *is a half-open time-interval representing* t*'s validity and* $\mathcal{D}$ *is the payload associated with the sgt* 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). The notation $E^I \subset E$ is used 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, the payload $\mathcal{D}$ of an sgt t represents 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* $S = [t_1, t_2, \cdots]$ *where each sgt* t_i *represents an edge* $e \in E$ *or a path* $p \in P$ *between vertices* $src, trg \in V$ *with label* l*,* $\mathcal{D}$ *is a payload consists of edges in* E *in* e *or* p *and each sgt* t_i *arrives at a particular time* ts_i *(*$ts_i < ts_j$ *for* $i < j$*).*

Figure 3.2 depicts an excerpt of the streaming graph derived from the input graph streaming in Figure 3.1 by assigning a time interval to each tuple (validity intervals are assigned by a time-based sliding window – see Definition 16). src, trg and the label l

are called the *distinguished* attributes and represent the topology of a materialized path graph.

Unless otherwise specified, streaming graphs considered in this thesis are *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 [95]. This corresponds to the *negative tuple* approach [63, 67]. Processing of insertions, deletions and expirations under alternative window semantics for physical operator implementations are described in detail in Chapters 4 and 5.

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., $S = \bigcup_{l \in \Sigma}(S_l)$*

The label-based partitioning of streaming graphs provides a coherent representation for inputs and outputs of operators in logical operator algebra (Chapter 5). At the logical level, it can be performed by the filter operator of the logical algebra (precisely defined in Definition 31), and operators of the logical 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 an edge or a path with the same label l between the same vertices with possibly different validity intervals and payloads. 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 [107]. [5] The *coalesce* primitive defined in temporal database literature [51] is extended to sgts with an aggregation function over the non-distinguished payload attribute, $\mathcal{D}$, as shown below:

Definition 11 (Coalesce Primitive). *The coalesce primitive that maps a set of value-equivalent sgts $\{t_1, \cdots, t_n\}$ ($t_i = (src, trg, l, [ts_i, exp_i), \mathcal{D}_i)$ for $1 \leq i \leq n$) with overlapping or adjacent validity intervals into a single value-equivalent sgt (i.e., with the same distinguished attributes src, trg and l) by merging their validity intervals and applying an*

[5] This is in contrast to *identity-equivalence* in object databases, where two tuples are equal if and only if they have the same identifier, regardless of the values of their attributes.

operator-specific aggregation function f_{agg} *over the payload attribute* $\mathcal{D}$*:*

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

Distinguished attributes src, trg and the label l of sgts in a streaming graph S can be used to define 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. This is used to define snapshot graphs and the property of *snapshot reducibility*.

Definition 12 (Snapshot Graph)**.** *A snapshot of a streaming graph* S *is defined by a mapping* τ *from each time instant* $t \in \mathcal{T}$ *to a finite set of sgts in* S *that are valid at time* t*. Applying the coalesce primitive (Definition 11) to all valid tuples at time* t*, a snapshot* $\tau_t(S)$ *induces a materialized path graph* $G_t = (V_t, E_t, P_t, \Sigma_t, \psi, \rho, \phi)$ *where* $E_t = \{e_i \mid e_i.ts \leq t < e_i.exp\}$ *is the set of all edges that are valid at time* t*,* $P_t = \{p_i \mid p_i.ts \leq t < p_i.exp\}$ *is the set of all paths that are valid at time* t*, and* V_t *is the set of all vertices that are endpoints of edges and paths in* E_t *and* P_t*, respectively.*

Definition 12 implies 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. In the presence of multiple value-equivalent sgts that are valid at time t, the coalesce primitive produces a single sgt by merging their validity intervals.

Definition 13 (Streaming Graph Equivalence)**.** *Two streaming graphs* S_1 *and* S_2 *are said to be equivalent if and only if their snapshot graphs are equal:*

$$S_1 \equiv S_2 \iff \forall t \in \mathcal{T}, \tau_t(S_1) \equiv \tau_t(S_2) \tag{3.1}$$

Remark 1 (Insert-delete streams)**.** *The use of validity intervals in the streaming graph model is previously explored in the context of relational streams, referred as the time-interval approach [63, 95]. A semantically equivalent alternative, the negative-tuple approach, is used in several relational-stream systems such as STREAM [15] and Nile [80]. In the negative-tuple approach, each stream elements is associated with either + or −, denoting addition and deletions, respectively. Validity of each tuple is defined by a pair of elements where the positive element signals the start timestamp and the negative element signals the expiration. Consequently, the negative-tuple and the direct approach can be used interchangeably to model streaming graphs. Each streaming graph tuple* $(src, trg, l, [ts, exp), \mathcal{D})$ *is expressed by a pair of tuples* $< (src, trg, l, ts, \mathcal{D}, +), (src, trg, l, exp, \mathcal{D}, -) >$*. Albeit semantically equivalent, the negative-tuple approach potentially duplicates the number of tuples flowing through the system, impacting the overall system performance [95].*

3.3 Streaming Graph Queries

This section presents the proposed *streaming graph query* (SGQ) model. First, a formal definition of SGQ using Datalog is provided, enabling the specification of precise SGQ semantics and to reason about its expressiveness. This is followed by a discussion of how SGQ captures a significant subset of existing graph query languages and provide concrete examples on how to formulate SGQ using a slight extension of G-CORE.

3.3.1 Formal Query Model

SGQ is based on a streaming generalization of the Regular Query (RQ) model [138]. Informally, RQ corresponds to *binary, non-recursive* subset of Datalog with transitive closure and provides a principled way to combine subgraph patterns and path navigations. 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 therefore composable, (ii) it has more expressive power than 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 [138]. Due to its well-defined semantics and computational behaviour, RQ has been gaining popularity as a logical foundation for graph queries, both in theory [30, 29] and in practice [11]. Indeed, RQ captures the core of the contemporary graph query language G-CORE that is used throughout this chapter.

Definition 14 (Regular Queries (RQ) – Following [138])**.** *The class of Regular Queries is the subset of non-recursive Datalog with a finite set of rules where each rule has the form* [6]*:*

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

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 label* $l \in \Sigma, d \in \Sigma \setminus \phi(E)$*, 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* $\notin \Sigma$*. The result of a Regular Query is a set of variable bindings for the predicate Answer.*

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

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 $l \in \phi(E^I)$ correspond 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 [5]. Similarly, IDBs are defined 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 in Figure 1.1. The one-time query*[7] *based on the same graph pattern corresponds to the following RQ:*

$$\begin{aligned} RL(u_1, u_2) &\leftarrow l(u_1, m_1), f^+(u_1, u_2) \textit{ as } FP, p(u_2, m_1) \\ Notify(u, m) &\leftarrow RL^+(u, v) \textit{ as } RLP, p(v, m) \\ Answer(u, m) &\leftarrow Notify(u, m) \end{aligned}$$

where predicates l, f, FP, p, RL, RLP *represent labels* likes*,* follows*,* followsPath*,* post*,* recent-Liker *and* recentLikerPath*, respectively.*

The notion of snapshot reducibility enables the precise definition of the semantics of streaming queries and operators using their non-streaming counterparts. Snapshot reducibility is used in temporal databases to generalize non-temporal queries and operators to temporal ones [51].

Definition 15 (Snapshot-Reducibility)**.** *Let* $\mathcal{Q}_\mathcal{S}$ *be streaming graph query over a streaming graph* S_I*, and* $\mathcal{Q}_O$ *its non-streaming, static (one-time) counterpart. Snapshot reducibility states that at any given time* t*, the snapshot graph of the output streaming graph* $S_O = Q_S(S_I)$ *is equivalent to the result of applying the one-time query* Q_O *over the corresponding snapshot of the input* S_I*, i.e.,* $\forall t \in \omega, \tau_t\big(\mathcal{Q}_\mathcal{S}(S_I)\big) = \mathcal{Q}_O\big(\tau_t(S_I)\big)$.

Following existing research [65], the semantics of persistent evaluation of SGQ is defined using the notion of snapshot reducibility (Definition 15). It is known that for many operations such as joins and aggregation, exact results cannot be computed with a finite memory over unbounded streams [17]. In streaming systems, a common solution for bounding the space requirement is to evaluate queries on a window of data from the

[7] *One-time* queries are evaluated over the current state of the database at the query time whereas persistent (*streaming*) queries are evaluated continuously and produce results as new tuples arrive and old tuples expire.

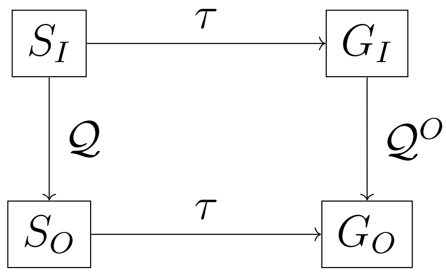


Figure 3.4: Snapshot reducibility (adapted from [95]).

stream. The windowed evaluation model provides a tool to process unbounded streams with bounded memory, and restricts the scope of queries to recent data, a desired feature in many applications[65, 17]. Additionally, as opposed to streaming approximation techniques that trade off exact answers in favour of bounding the space requirements, window-based query evaluation enables exact query answers w.r.t. window specifications. Consequently, the *time-based sliding window* model is adopted where a fixed size (in terms of time units) window is defined that slides at well-defined intervals [65]. In the context of streaming graphs, new graph edges enter the window during the window interval, and when the window slides, some of the "old" edges leave the window (i.e., expire).

Definition 16 (Time-Based Sliding Window)**.** *A time-based sliding window $\mathcal{W}_\omega$ over an input streaming graph S^I is defined by an interval length ω and an optional slide interval β. The window contents $\mathcal{W}_\omega(S^I)$ at any given time t is a multiset of streaming graph edges where the timestamp of ts_i of each sge sge_i is in the window interval, i.e., $\{sge_i \mid t - \omega \leq ts_i < t\ \}$.*

Remark 2 (Algebraic operators and stream transformations)**.** *In Chapter 5 the WSCAN operator is defined that transforms a given input graph stream S^I into a streaming graph S where the validity interval of each sgt on the output streaming graph is assigned in conformance with Definition 16.*

Remark 3 (Snapshot graphs over input graph streams)**.** *At any given time t, the graph induced by the contents of a time-based sliding window over an input graph stream $\mathcal{W}_\omega(S^I)$ is isomorphic to the snapshot graph $G_t = \tau_t\big(\mathcal{W}_\omega(S^I)\big)$, where the validity interval of every element in the graph is equal to the window size ω. This equivalence directly follows from the definitions of time-based sliding windows (Definition 16) and snapshots graphs (Definition 12), and it is used in the following to define semantics of SGQ evaluation.*

Definition 17 (Streaming Graph Query – SGQ)**.** *An SGQ query Q_S is an RQ defined over an input streaming graph S_I and a time-based sliding window $\mathcal{W}_\omega$ whose semantics is* *defined using the corresponding, one-time RQ Q_O and the notion of snapshot reducibility (Definition 15):*

$$\forall t \in \omega, \quad \tau_t\big(Q_S(S_I, \mathcal{W}_\omega)\big) = Q_O\Big(\tau_t\big(\mathcal{W}_\omega(S_I)\big)\Big)$$

Figure 3.4 illustrates the correspondence between streaming and one-time graph queries: at any given time, the set of valid tuples in the result of a streaming query induces a graph that is equivalent to the result of applying the corresponding one-time query over the snapshots of the input streaming graphs. A direct consequence of such a relationship is that SGQ can be evaluated by repeatedly executing the corresponding one-time query, known as *query re-evaluation* [63]. Specifically, the resulting streaming graph of an SGQ can be obtained from the sequence of snapshots that is the result of repeated evaluation of the corresponding one-time query at every time instant: an sgt $(u, v, l, [ts, expiry), D)$ is in the resulting streaming graph for an SGQ Q_S $\tau_t\big(Q_S(S_I, \mathcal{W}_\omega)\big)$ 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 $G_{t_i} = Q_O\Big(\tau_t\big(\mathcal{W}_\omega(S_I)\big)\Big)$ for $ts \leq t < exp$. However, such a strategy is wasteful as the input differences between two consecutive instants are likely to be small.[8] Alternatively, *incremental evaluation* computes the changes in the output as new sgts arrive and old sgts expire due to window movements. The focus in this thesis is on the *incremental evaluation* method and the concept of *snapshot reducibility* is used to ensure correct evaluation semantics.

3.3.2 SGQ in Practice

The SGQ model formalizes the important class of streaming graph queries using a logic-based formalism. It captures the core features of current graph query languages such as subgraph pattern and reachability-based path queries. In this section, SGQ's expressive power is illustrated by mapping core G-CORE constructs to SGQ. G-CORE is chosen for demonstration due to the following reasons. G-CORE fulfills all three requirements of graph querying (**R1**, **R2** & **R3** in Section 1.2). Other existing languages (e.g., SPARQL v1.1, Cypher, PGQL) can only partially satisfy these requirements due to (i) the lack of algebraic closure and composability, and (ii) limited path navigation capability [30]. Moreover, G-CORE supports SGQ capabilities such as the treatment of paths as first-class citizens and returning graphs. Finally, G-CORE is one of the more prominent language

[8] The performance overhead of *query re-evaluation* is empirically analyzed for path navigation query fragment of SGQ in Section 4.5.6.

```
PATH RL = (u1)-/ <:follows^*> /->(u2),
          (u1)-[:likes]->(m1)<-[:posts]-(u2)
CONSTRUCT (u) -[:notify]-> (m)
MATCH (u)-/ p<~RL*> /->(v),
      (v)-[:posts]->(m),
ON social_stream WINDOW(24h) SLIDE(1h)
```

Figure 3.5: G-CORE representation of the SGQ in Example 1.

specifications influencing the ongoing standardization process of a graph query language GQL [9].

Since the graph data model that is used in this work does not yet include properties, the focus is on a subset of G-CORE where queries do not contain predicates or aggregation over property values. This particular subset already covers many important key features:

- **(a)** returning graphs,
- **(b)** ASCII-art syntax for pattern matching,
- **(c)** joins over multiple graphs,
- **(d)** views and optionals,
- **(e)** RPQ-based reachability queries,
- **(f)** and powerful path patterns as demonstrated in the two examples discussed below.

G-CORE is originally targeted for one-time queries over static property graphs and it does not provide native windowing constructs. Examples used in this section slightly extends the `ON` clause with a `WINDOW` clause to incorporate window specifications. In particular, a time-based sliding window is defined by the newly introduced `WINDOW` clause that specifies the window length, and an optional `SLIDE` clause that specifies the slide interval, following a streaming graph reference in the `ON` clause.

Example 4. *The G-CORE query in Figure 3.5 represents the real-time notification example in Example 1 (its corresponding RQ is already given in Example 3). Its* `PATH` *and* `MATCH` *clauses use ASCII-art syntax (**b**) to define complex graph patterns (**f**) with RPQ-based reachability (**e**), and its* `CONSTRUCT` *clause returns a streaming graph of* notify *edges (**a**).*

[9] https://www.gqlstandards.org

```
GRAPH VIEW rec_stream AS (
  CONSTRUCT (u1) -[:recommendation]-> (p)
  MATCH (u1)
    OPTIONAL (u1)-[:follows]->(u2)
    OPTIONAL (u1)-[:likes]->(m)<-[:posts]-(u2)
  ON social_stream WINDOW(24 hours)
  MATCH (c) -[:purchase]->(p)
  ON tx_stream WINDOW(30d) SLIDE(1d)
  WHERE (u2) = (c) )
```

Figure 3.6: G-CORE representation of the query in Example 5

Example 5. *Consider the G-CORE query in Figure 3.6 that combines streaming information from a social network of user interactions and a transaction network of customer purchases to drive product recommendations. Its defines a view over the resulting streaming graph of* recommendation *edges (**d**) by joining patterns from two streaming graphs (**c**), and its* `MATCH` *clause features optional predicates to incorporate two alternative social interactions (**d**). Its graph pattern corresponds to the following RQ:*

$$\begin{aligned}
ACQ(u_1, u_2) &\leftarrow l(u_1, m_1), p(u_2, m_1) \\
ACQ(u_1, u_2) &\leftarrow f(u_1, u2) \\
REC(u, p) &\leftarrow ACQ(u_1, u_2), pur(u_2, p) \\
Answer(u, p) &\leftarrow REC(u, p)
\end{aligned}$$

where predicates l, f, p, pur, ACQ, REC represent labels likes, follows, post, purchase, acquaintance, and recommendation, respectively.

3.4 Discussion

This chapter introduces the *Streaming Graph Queries* and its underlying *Streaming Graph* data model that constitutes the formal foundations of the streaming graph query processing framework proposed in this thesis. SGQ, as a logic-based formalism, enables declarative specification of queries that feature subgraph patterns and path navigations, independent of particular algorithms and implementations. Its semantics is presented by providing a mapping from SGQ to a subset of Datalog by using the notion of snapshot reducibility.

This enables precise characterization of the expressive power and the complexity of the query model – this is similar to showing the mapping from a subset of relational calculus to first-order logic.

By augmenting unions of conjunctive queries with transitive closure, the RQ and the SGQ models strictly subsume the class of conjunctive queries (CQ) and regular path queries (RPQ), which are commonly used to model subgraph pattern and path navigation queries in practice, respectively (**R1** & **R2**). Furthermore, the SGQ model treats paths as first-class citizens of the underlying data model, enabling users to formulate queries that return and manipulate paths in a declarative manner (**R3**). Having streaming graphs as both inputs and outputs of queries, SGQ is closed over the streaming graph data model and provides full composability. Time-based sliding windows provide a deterministic solution to evaluate streaming graph queries on unbounded streams by restricting the scope of queries on recent data (**R4**). The precise semantics of SGQ evaluation is described using the notion of snapshot-reducibility, allowing development of non-blocking physical operator for incremental evaluation as will be shown in Chapter 5 (**R5**).

As described in Section 3.3, SGQ evaluation over streaming graphs can be reduced to its static counterpart by repeatedly executing the corresponding one-time query over a sequence of snapshot graphs. In particular, a given one-time RQ can be decomposed into a dataflow graph using its dependency graph (see Definition 14). Conjunctive queries can be evaluated using existing relational techniques – multiway join algorithms in particular. Similarly, regular path queries can be evaluated using automata- or relational-based algorithms for recursive queries [12, 11, 54].

Albeit semantically correct, such *query re-evaluation* is inefficient. *Incremental* evaluation, on the other hand, avoids re-computation of the entire result by only computing the changes to the output as new input arrives. It is desired that query evaluation algorithms in a streaming system have non-blocking behaviour, i.e., they do not need the entire input to be available before producing the first result. There is a plethora of techniques for evaluating conjunctive queries in the streaming settings such as the specialized algorithms for subgraph matching on streaming graphs [9, 103, 137, 90, 42]. Evaluation of path navigation queries in the streaming settings, on the other hand, have received little attention except dynamic reachability algorithms [99, 139]. To date, there has been no work that considers the problem of RPQ evaluation over streaming graphs. Therefore, in the next chapter, the focus is on this particular subset of streaming graph queries and their incremental evaluation.

Chapter 4

Regular Path Query Evaluation on Streaming Graphs

4.1 Introduction

Incremental evaluation of SGQ requires non-blocking algorithm implementations that compute the changes to the output as new inputs arrive. Path navigation queries are an important feature of graph querying, yet, their incremental evaluation has received little attention so far except dynamic reachability algorithms. In this chapter, non-blocking algorithm implementations are described for this important subclass of the SGQ model, focusing on the problem of continuously evaluating path navigation queries over streaming graphs. The Regular Path Query (RPQ) model is adopted, because it is the de-facto

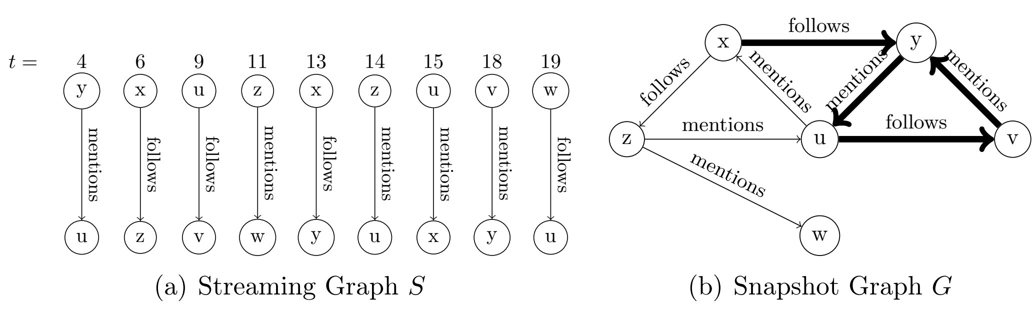
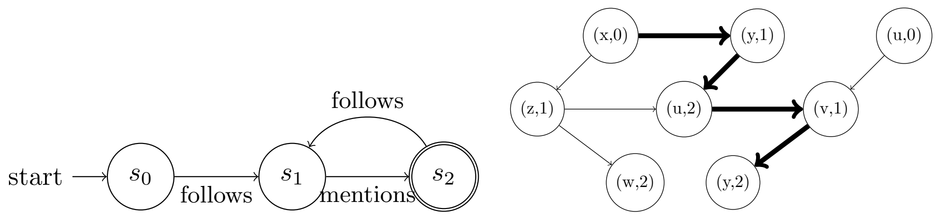


Figure 4.1: (a) A streaming graph S of a social networking application, and (b) its snapshot at $t = 18$.



(a) Query Graph Q_1 (b) Product Graph $P_{G,A}$

Figure 4.2: (a) Automaton for the query $Q_1 : (follows \circ mentions)^+$, and (b) the product graph $P_{G,A}$.

formalism for *path navigation* queries in practical graph query languages. RPQ specifies path constraints that are expressed using a regular expression over the alphabet of edge labels and checks whether a path exists with a label that satisfies the given regular expression [121, 18]. The RPQ model provides the basic navigational mechanism to encode graph queries, striking a balance between expressiveness and computational complexity [12, 30, 156, 11, 165]. Consider the streaming graph of a social network application presented in Figure 4.1(a). The query $Q_1 : (follows \circ mentions)^+$ in Figure 4.2(a) represents a pattern for a real-time notification query where user `x` is notified of other users who are connected by a path whose edge labels are even lengths of alternating *follows* and *mentions*. At time $t = 18$, the pair of users (`x`,`y`) is connected by such a path, shown by bold edges in Figure 4.1(b).

It is known that for many streaming algorithms the space requirement is lower bounded by the stream size [17]. Since the stream is unbounded, deterministic RPQ evaluation is infeasible without storing all the edges of the graph (by reduction to the length-2 path problem that is infeasible in sublinear space [57]). Following the SGQ model introduced in Chapter 4, streaming RPQ evaluation uses the *time-based sliding window* model where a fixed size (in terms of time units) window is defined that slides at well-defined intervals [65]. Managing this window processing as part of RPQ evaluation is challenging and the proposed solutions address the issue in a uniform manner.

The design space of persistent RPQ evaluation algorithms can be identified in two main dimensions: the path semantics they support and the result semantics based on application requirements. Along the first dimension, this thesis proposes efficient incremental algorithms for both *arbitrary* and *simple* path semantics. The former allows a path to traverse the same vertex multiple times, whereas under the latter semantics a path cannot traverse the same vertex more than once [12]. Consider the example graph given in Figure 4.1(b); the sequence of vertices $\langle x, y, u, v, y \rangle$ is a valid path for query Q_1 with arbitrary

Table 4.1: Amortized time complexities of the proposed algorithms for a streaming graph S with m edges and n vertices and RPQ Q_R whose automata has k states.

Path Semantics \ Result Semantics	Append-Only	Explicit Deletions
Arbitrary (Section 4.3)	$\mathcal{O}(n \cdot k^2)$	$\mathcal{O}(n^2 \cdot k)$
Simple[1] (Section 4.4)	$\mathcal{O}(n \cdot k^2)$	$\mathcal{O}(n^2 \cdot k)$.

path semantics whereas the simple path semantics does not traverse this path as it visits vertex y twice. Along the second dimension, the algorithms first consider *append-only* streams where tuples in the window expire only due to window movements. They are then extended to support *explicit deletions* to deal with cases where users/applications might explicitly delete a previously arrived edge. The *negative tuples* approach [68] is used to process explicit deletions. Table 4.1 presents the combined complexities of the proposed algorithms in each quadrant in terms of their amortized cost over a sequence of operations.

These are the first streaming algorithms to address RPQ evaluation on sliding windows over streaming graphs under both arbitrary (Section 4.3) and simple path semantics (Section 4.4). The proposed algorithm for streaming RPQ evaluation under arbitrary path semantics incrementally maintains results for a query Q_R on a sliding window $\mathcal{W}$ over a streaming graph S as new edges enter and old edges expire due to window slide. The algorithm follows the implicit window semantics, where newly arriving edges are processed as they arrive (and new results appended to the output stream) while the removal of expired edges occur at user-specified slide intervals. The algorithm utilizes the temporal pattern of window movements to simplify the state maintenance and the removal of expired edges. As shown in Table 4.1, the amortized cost of an edge insertion is $\mathcal{O}(n \cdot k^2)$, so the worst-case complexity of the proposed algorithm over m edges matches the worst-case complexity of the corresponding batch algorithm over a graph that consists of m edges (i.e., $\mathcal{O}(m \cdot n \cdot k^2)$ as shown in Section 4.3). In other words, over a sequence of edges, the proposed algorithm for streaming RPQ evaluation under arbitrary path semantics runs in time asymptotically no worse than the corresponding batch algorithm over the graph induced by the same edges.

The static version of the RPQ evaluation problem is NP-hard in its most general form [121], which has caused existing work to focus only on arbitrary path semantics. Yet, it is

[1]These results hold in the absence of conflicts, a condition on cyclic structure of the query and graph that is precisely defined in Section 4.4.1.

proven to be tractable when restricted to certain classes of regular expressions or by imposing restrictions on the graph instances [121, 20]. A recent analysis [31, 32] of real-world SPARQL logs shows that a large portion of RPQs posed by users does indeed fall into those tractable classes, motivating the design of efficient algorithms for streaming RPQ evaluation under simple path semantics. An algorithm is proposed that admits efficient solutions for streaming RPQs under simple path semantics in the absence of conflicts, a condition on the cyclic structure of graphs that enables efficient batch algorithms (precisely defined in Section 4.4.1) [121]. Indeed, this algorithm has the same amortized time complexity as the proposed algorithm for arbitrary path semantics under the same condition as shown in Table 4.1. This has two implications: (i) the proposed algorithm for streaming RPQs under simple path semantics carries over the existing tractability results to the streaming settings, and (ii) its amortized complexity is equivalent to its arbitrary path semantics counterpart under these conditions.

The proposed algorithms incrementally maintain query answers as the window slides thus eliminating the computational overhead of the naive strategy of batch computation after each window movement. These algorithms handle expirations by utilizing the temporal pattern of window movements where edges expire in the same order they are inserted into a window. Furthermore, they support negative tuples to accommodate applications where users might explicitly delete a previously inserted edge. Albeit relatively rare, explicit deletions are a desired feature of real-world applications that process and query streaming graphs, and it is known to require special attention [67]. Unlike expirations due to sliding window movements that follow a temporal pattern, the arbitrary nature of explicit deletions incurs additional complexity for the proposed algorithms, as shown in Table 4.1. Section 4.3.2 describes how explicit deletions can be handled in a uniform manner by utilizing the machinery developed for window management. The performance of the proposed algorithms are empirically evaluated using a variety of real-world and synthetic streaming graphs on real-world RPQs that cover more than 99% of all recursive queries found in Wikidata query logs by a recent analysis [32] (Section 4.5).

4.2 Preliminaries

The *streaming graph* data model introduced in Section 3.2 is used to represent input and output streaming graphs of RPQs. For ease of representation, the *negative-tuple* approach is used throughout this chapter as the underlying stream representation. Each sgt is associated with an operation type, i.e., insert (+) or delete (−), to denote the validity of elements. This enables the uniform handling of the entire design space of streaming

RPQ algorithms – both the alternative window semantics and handling of expirations and explicit deletions. Recall that these two representations are semantically equivalent and can be used interchangeably, as previously described in Remark 1.

Figure 4.1(a) shows an excerpt of a streaming graph S, and Figure 4.1(b) shows its snapshot graph $G_{18} = \tau_{18}\big(\mathcal{W}_{15}(S)\big)$ defined by window $\mathcal{W}$ with $\omega = 15$.

A time-based sliding window $\mathcal{W}$ (Definition 16) might progress either at every time unit, i.e. $\beta = 1$ (*eager evaluation*; resp. *expiration*) or at $\beta > 1$ intervals (*lazy evaluation*; resp. *expiration*) [136]. Eager evaluation produces fresh results but windows can be expired lazily if queries do not produce premature expirations [67]. The proposed algorithms use eager evaluation ($\beta = 1$) but lazy expiration ($\beta > 1$) as it enables the separation of window maintenance from processing of incoming sgts (Section 4.3.1).

Definition 18 (Regular Expression & Regular Language)**.** *A regular expression R over an alphabet Σ is defined as $R ::= \epsilon \mid a \mid \ R \circ R \mid R + R \mid R^*$ where (i) ϵ denotes the empty string, (ii) $a \in \Sigma$ denotes a character in the alphabet, (iii) $\circ$ denotes the concatenation operator, (iv) $+$ denotes the alternation operator, and (v) $*$ represents the Kleene star. $\neg$ is used to denote the negation of an expression, and R^+ to denote 1 or more repetitions of R.A regular language $L(R)$ is the set of all strings that can be described by the regular expression R.*

Definition 19 (Regular Path Query – RPQ)**.** *A one-time Regular Path Query Q_R over a static graph G asks for pairs of vertices (u, v) that are connected by a path p from u to v in graph G, where the path label $\phi^p(p)$ is a word in the regular language defined by the regular expression R over the graph's edge labels Σ, i.e., $\phi^p(p) \in L(R)$. Answer to query Q_R^O over G, $Q_R^O(G)$, is the set of all pairs of vertices that are connected by such paths.*

Sliding windows adhere to two alternative semantics: *implicit* and *explicit* [68]. Implicit windows add new results to query output as new sgts arrive and do not invalidate the previously reported results upon their expiry as the window moves. In the absence of explicit edge deletions, the query results are monotonic. Under this model, the result set of a streaming RPQ over a streaming graph S and a sliding window $\mathcal{W}$ at time t contains all paths in all previous snapshot graphs G_π where $0 < \pi \leq t$, i.e., $\tau_t\big(Q_R(S, \mathcal{W})\big) = \bigcup_{0<\pi\leq t} Q_R^O\Big(\tau_\pi\big(\mathcal{W}(S)\big)\Big)$. Alternatively, explicit windows remove previously reported results involving tuples (i.e., sgts) that have expired from the window; hence, persistent queries with explicit windows are akin to incremental view maintenance. Under this model, the result set of a streaming RPQ over a streaming graph S and a sliding window $\mathcal{W}$ at time t contains only the paths in the snapshot G_t of the streaming graph, i.e., $\tau_t\big(Q_R(S,\mathcal{W})\big) = Q_R^O\Big(\tau_t\big(\mathcal{W}(S)\big)\Big)$. Explicit windows, by definition, produce non-monotonic results as previous results are negated when the window moves [68]. The rest of this section assumes the implicit window model as it enables the preservation of the monotonicity of query results and produces an append-only stream of query results (in the absence of explicit deletions).

Definition 20 (Streaming RPQ)**.** *Following the SGQ model (Definition 17), a streaming RPQ is defined over a streaming graph S and a sliding window $\mathcal{W}_\omega$ of size ω. A pair of vertices (u, v) is an answer for a streaming RPQ, Q_R, at time t if there exists a path p between u and v in $G_t = \tau_t\big(\mathcal{W}_\omega\big)$, i.e., timestamps of every edges in p are in the window interval. The timestamp $p.ts$ of a path p is the minimum timestamp among all edges of p. Under the implicit window model, the resulting streaming graph of a streaming RPQ Q_R over a streaming graph S and a sliding window $\mathcal{W}$ is an append-only stream of sgts (u, v, l_O, ts, p) where there exists a path p between u and v with label $\phi^p(p) \in L(R)$ and all the edges in p are at most one window length, i.e., ω time units, apart. Formally:*

$$\forall t \in \mathcal{T}, \tau_t\big(Q_R(S,\mathcal{W}_\omega)\big) = \{(u, v, l_O, ts, p) \mid \exists p : u \xrightarrow{p} v \wedge l_O = \phi(p) \in L(R) \wedge \max_{e \in p}(e.ts) < p.ts + \omega \leq t\}$$

Definition 21 (Deterministic Finite Automaton)**.** *Given a regular expression R, $A = (S, \Sigma, \delta, s_0, F)$ is a Deterministic Finite Automaton (DFA) for $L(R)$ where S is the set of states, Σ is the input alphabet, $\delta\colon S \times \Sigma \to S$ is the state transition function, $s_0 \in S$ is the start state and $F \subseteq S$ is the set of final states. δ^* is the extended transition function defined as:*

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

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

Definition 22 (Product Graph)**.** *Given a graph $G = (V, E, \Sigma, \psi, \phi)$ and a DFA $A = (S, \Sigma, \delta, s_0, F)$, the product graph $P_{G,A}$ is defined as a quintiple $(V_P, E_P, \Sigma, \psi_P, \phi_P)$ where $V_P = V \times S$, $E_P \subseteq V_P \times V_P$ is a set of edges, $\psi_P : E_P \to V_P \times V_P$ is an incidence function such that $((u, s), (v, t))$ is in E_P iff $(u, v) \in E$ and $\delta(s, \phi(u, v)) = t$.*

Figure 4.2(b) shows the product graph of G_18 (Figure 4.1(b)) and the DFA A of the query Q_1 (Figure 4.2(a)).

For a given RPQ with a regular expression R, Thompson's construction algorithm [160] is first used to create a NDFA that recognizes the language $L(R)$, then the equivalent minimal DFA, A, is created using Hopcroft's algorithm [84]. In the remainder of this chapter, A and the product graph $P_{G,A}$ are used to describe the proposed algorithms for RPQ evaluation in the streaming graph model.

4.3 RPQ with Arbitrary Semantics

The focus of this section is the problem of RPQ evaluation over sliding windows of streaming graphs under arbitrary path semantics, that is, finding pairs of vertices $u, v \in V$ where (i) there exists a (not necessarily simple) path p between u and v with a label $\phi^p(p)$ in the language $L(R)$, and (ii) timestamps of all edges in path p are in the range of window $\mathcal{W}$. Append-only streams are considered where the query results are monotonic (under *implicit window* model) such that existing results do not expire from the result set when input tuples expire from the window [68]. Then, the algorithms are extended to support negative tuples to handle explicit edge deletions.

Batch Algorithm: RPQs can be evaluated in polynomial time under arbitrary path semantics [121]. Given a product graph $P_{G,A}$, there is a path p in G from x to y with label w that is in $L(R)$ if and only if there is a path in $P_{G,A}$ from (x, s_0) to (y, s_f), where $s_f \in F$. The batch RPQ evaluation algorithm under arbitrary path semantics traverses the product graph $P_{G,A}$ by simultaneously traversing graph G and the automaton A. The time complexity of the batch algorithm is $\mathcal{O}(n \cdot m \cdot k^2)$ under the assumption that there are more edges than isolated vertices in G.

4.3.1 RPQ over Append-Only Streams

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

Definition 23 (Δ Tree Index). *Given an automaton A for a query Q_R and a snapshot G_{ts} of a streaming graph S over a window $\mathcal{W}$ at time ts, Δ is a collection of spanning trees where*

Algorithm RAPQ:

input : Input streaming graph S,
Window size ω,
Regular expression R,
output label O

output: Output streaming graph S_O

$A(S, \Sigma, \delta, s_0, F) \leftarrow$ `ConstructDFA`(R)
`Initialize` Δ
$S_O \leftarrow \emptyset$
R $\leftarrow \emptyset$
foreach $sgt = (u, v, l, ts, \mathcal{D}) \in S$ **do**
    $G_{ts} \leftarrow G_{ts-1}$ (op) $e(u, v)$ `// update snapshot graph`
    ExpiryRAPQ$(G_{ts}, \omega, T_x, ts\ )\ \forall T_x \in \Delta$ `// on user-defined slide intervals`
    foreach $s, t \in S$ *where* $t = \delta(s, l)$ **do**
        if $s = s_0 \wedge T_u \notin \Delta$ **then**
            add T_u with root node (u, s_0)
        end
        foreach $T_x \in \Delta$ **do**
            if $(u, s) \in T_x \wedge (u, s).ts > ts - \omega$ **then**
                if $(v, t) \notin T_x \vee (v, t).ts < min((u, s).ts, ts)$ **then**
                    $R \leftarrow R+$Insert$(G_{ts},\ T_x,\ (u, s),\ (v, t),\ e(u, v))$
                end
            end
        end
    end
end
foreach $sgt\ t \in R$ **do**
    push t to S_O
end

Algorithm Insert:

input : Snapshot graph G_{ts},
Spanning Tree T_x rooted at (x, s_0),
parent node (u, s),
child node (v, t),
edge (u, v)
output: The set of results R

$R \leftarrow \emptyset$
$(v, t).pt = (u, s)$
$(v, t).ts = min(ts, (u, s).ts)$ **if** $(v, t) \notin T_x$ **then**
    if $t \in F$ **then**
        $p \leftarrow \texttt{PATH}(T_x, (v, t))$
        $R \leftarrow R + (x, v, O, (v, t).ts, p)$
    end
    foreach *edge* $(v, w) \in G_{ts}$ *s.t.* $\delta(t, \phi(v, w)) = q$ **do**
        if $(w, q) \notin T_x \vee (w, q).ts < min((v, t).ts, (v, w).ts)$ **then**
            $R \leftarrow R+$Insert$(G_{ts},\ T_x,\ (v, t),\ (w, q),\ e(v, w))$
        end
    end
end
return R

each tree T_x is rooted at a vertex $x \in G_{ts}$ for which there is a corresponding node in the product graph of A and G_{ts} with the start state s_0, i.e., $\Delta = \{T_x \mid x \in G_{ts} \wedge (x, s_0) \in V_{P_{G,A}}\}$.

In the remainder, the term "vertex" denotes endpoints of sgts, and the term "node" denotes vertex-state pairs in spanning trees.

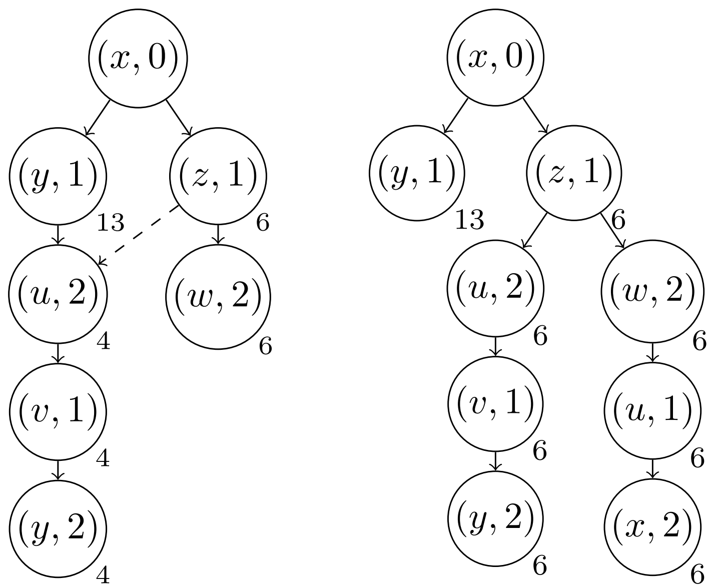
A node $(u, s) \in T_x$ at time τ indicates that there is a path p in G_{ts} from x to u with label $\phi^p(p)$ and timestamp $p.ts$ such that $\delta^*(s_0, \phi^p(p)) = s$ and $(\tau - \omega) < p.ts \leq \tau$, i.e., word $\phi^p(p) \in \Sigma^*$ takes the automaton A from the initial state s_0 to a state s and the timestamp of the path is in the window range. Each node (u, s) in a tree T_x maintains a pointer $(u, s).pt$ to its parent in T_x. Additionally, the timestamp $(u, s).ts$ is the minimum timestamp among all edges in the path from (x, s_0) to (u, s) in the spanning tree T_x, following Definition 20.

Note that Algorithm **RAPQ** can directly use the window size ω to determine the validity of streaming graph edges and spanning tree nodes as the snapshot graph G_{ts} is defined over an input graph stream through a time-based sliding window $\mathcal{W}$ whose window size is ω (Remark 3). Algorithm **RAPQ** continuously updates G_{ts} upon arrival of new edges and expiry of old edges (Line 6). In addition to G_{ts}, it maintains a tree index (Δ) to support efficient incremental RPQ evaluation that enables efficient RPQ evaluation on sliding windows over streaming graphs.

Example 6. *Figure 4.3(a) illustrates a spanning tree $T_x \in \Delta$ for the streaming graph S and the RPQ Q_1 given in Figure 4.2 at time $t = 18$. The tree in Figure 4.3(a) is constructed through a traversal of the product graph starting from node $(x, 0)$, visiting nodes $(y, 1)$, $(u, 2)$, $(v, 1)$ and $(y, 2)$, forming the path from the root to the node $(y, 2)$ in Figure 4.3(a). Similar to the batch algorithm, this corresponds to the traversal of the path $\langle x, y, u, v, y\rangle$ in the snapshot of the streaming graph (Figure 4.1(b)) with label ⟨follows, mentions,follows, mentions⟩ taking the automaton from state 0 to 2 through the path $\langle 0, 1, 2, 1, 2\rangle$ in the corresponding automaton (Figure 4.2(a)). The timestamp of the node $(y, 2) \in T_x$ at $t = 18$ is 4 as the edge with the minimum timestamp on the path from the root is $(y, mentions, u)$ with $ts = 4$.*

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

1. *A node (u, s) with timestamp ts is in T_x if there exists a path p in G_{ts} from x to u with label $\phi^p(p)$ and timestamp $(u, s).ts$ such that $s = \delta^*(s_0, \phi^p(p))$ and $(u, s).ts = p.ts \in (ts - \omega, ts]$, i.e., there exists a path p in G_{ts} from x to u with label $\phi^p(p)$ such that $\phi^p(p)$ is a prefix of a word in $L(R)$ and all edges are in the window $\mathcal{W}$.*



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

Figure 4.3: A spanning tree $T_x \in \Delta$ for the example given in Figure 4.2 rooted at $(x, 0)$ (a) before and (b) after the edge $e(w, u)$ with label $follows$ at $t = 19$ is consumed. The timestamp of each node given at the corner.

2. *At any given time ts, a node (u, s) appears in a spanning tree T_x at most once with a timestamp in the range $(ts - \omega, ts]$.*

Proof. First step is to show that Algorithm **ExpiryRAPQ** maintains the two invariants of the Δ tree index. The second invariant is preserved as Algorithm **ExpiryRAPQ** does not add any node to a spanning tree $T_x \in \Delta$. For each spanning tree $T_x \in \Delta$, Line 2 of the algorithm identifies the set of nodes that are potentially expired at time ts, $P = \{(v, t) \in T_x \mid (v, t).ts \leq ts - \omega\}$. Initially, all expired nodes are removed from the spanning tree T_x (Line 3). Algorithm **Insert** is invoked for each expired node $(v, t) \in P$ if there exists a valid edge in the window G_{ts} from another valid node in T_x (Line 7). Finally, nodes that are reconnected to the spanning tree T_x by Algorithm **Insert** are removed from P as there exists an alternative path from the root through (u, s). As a result, Algorithm **ExpiryRAPQ** removes a node (v, t) from the spanning tree T_x if there does not exist any path p in G_{ts} from x to u with a label l such that $s = \delta^*(s_0, l)$ and $p.ts > ts - \omega$, preserving the first invariant.

It is easy to see that the second invariant is preserved after each call to Algorithm **RAPQ** given that Algorithm **ExpiryRAPQ** preserves both invariants. The second invariant is preserved as Line 3 of Algorithm **Insert** adds the node (v, t) to a spanning tree T_x only if it has not been previously inserted.

It can be shown that Algorithm **RAPQ** preserves the first invariant by induction on the length of the path. For the base case $n = 1$, consider that $sgt = (u, v, l, ts, \mathcal{D})$ arrives in the window S at time ts. Line 8 in Algorithm **RAPQ** identifies each state t where there is a transition from the initial state s_0 with label l, i.e., $\delta(s_0, l) = t$. The path from (u, s_0) to (v, t) is added to T_x with $(v, t).ts = ts$. For the non-base case, consider a node $v \in G_{ts}$ where there exists a path p of length n from x where $t = \delta^*(s_0, \phi^p(p))$ and $p.ts > ts - \omega$. Let (u, s) be the predecessor of (v, t) in the path, that is edge (u, v) is in G_{ts} with label l and $\delta(s, l) = t$. By the inductive hypothesis, the node (u, s) is in T_x as there exists a path q of length $n-1$ from x to u in G_{ts} where $s = \delta^*(s_0, \phi(q))$ and $q.ts > ts-\omega$. If the timestamp of the edge $e(u, v) \in G_{ts}$ is within the window interval (i.e., $ts - \omega < e.ts < ts$) when the node (u, s) is inserted into T_x , then the proposed algorithm invokes Algorithm Insert with node (u, s) as parent and node (v, t) as child (Line 15) and its adds (v, t) into T_x with timestamp $(v, t).ts = min(e.ts, (u, s).ts)$ (Line 3). If the edge $e(u, v)$ is processed by the proposed algorithm after the node (u, s) is inserted in T_x ($e.ts > (u, s).ts$), then Line 10 in Algorithm **Insert** guarantees that Algorithm **Insert** is invoked with the node (v, t). Lines 2 and 3 in Algorithm **Insert** adds the node (v, t) to T_x, and properly updates its parent pointer to (u, s) and its timestamp $(v, t).ts = min(e.ts, (u, s).ts)$. The first invariant is preserved in either case as $ts - \omega < p.ts = (v, t).ts \leq ts$. Therefore Algorithm **RAPQ** also preserves the first invariant. □

The first invariant allows tracing all reachable nodes from a root node (x, s_0) whereas the second invariant prevents Algorithm **RAPQ** from visiting the same vertex in the same state more than once in the same tree. Consider the example in Figure 4.3(a): node $(u, 2)$ is not added as a child of the node $(x, 1)$ after traversing edge $(x, u) \in S$ with label *mentions* since $(u, 2)$ is already reachable from $(x, 0)$.

Algorithm **ExpiryRAPQ** is invoked at pre-defined slide intervals to remove expired nodes from Δ. For each $T_x \in \Delta$, it identifies the set of candidate nodes whose timestamps are not within the window interval (Line 2) and temporarily removes those from T_x (Line 3). For each candidate (v, t), Algorithm **Insert** finds an incoming edge from another valid node in T_x (Line 7) and it reconnects the subtree rooted at (v, t) to T_x. Nodes with no valid incoming edges are permanently removed from T_x. Algorithm **ExpiryRAPQ** might traverse the entire snapshot graph G_{ts} in the worst case. This can be used to undo previously reported results if explicit window semantics is required (Line 14), yet, it is only used to process explicit deletions as described in Section 4.3.2.

Example 7. *Consider the example provided in Figure 4.3(b) and assume that window size is* $\omega = 15$ *time units. Upon arrival of edge* (w, u) *with label* $follows$ *at* $t = 19$*, nodes* $(u, 1)$ *and* $(x, 2)$ *are added to* T_x *as descendants of* $(w, 2)$*. Also, paths leading to nodes* $(u, 2)$*,* $(v, 1)$ *and* $(y, 2)$ *are expired as their timestamp is* 4 *(due to the edge* (y, u) *with a timestamp* 4*). Algorithm **ExpiryRAPQ** searches incoming edges of vertex* u *in* G_{ts} *and identifies that there exists a valid edge* (z, u) *with label* $mentions$ *and timestamp* 14*. As a result, node* $(u, 2)$ *and its subtree is reconnected to node* $(z, 1)$*.*

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

Proof. Algorithm **RAPQ** terminates as Line 3 ensures that no node is visited more than once in any spanning tree in Δ.

If: If direction follows trivially from the first invariant of spanning trees. Lemma 1 guarantees that node (u, s) is inserted into the spanning tree T_x if there exists a path p in the snapshot graph G_{ts} of the window $\mathcal{W}$ at time ts from x to u satisfying R. Line 6 in Algorithm **Insert** adds the pair $(x, u, O, (u, s).ts, p)$ to the output streaming graph S_O if the target state is an accepting state, $s \in F$.

Only If: If the algorithm adds $(x, u, O, (u, s_f).ts, p)$ to R, then it must traverse a path p from x to u in G_{ts} where $s_f = \delta^*(s_0, \phi^p(p)), s_f \in F$ and $p.ts \in (ts - \omega, ts]$. Let n be the length of such path p. For any (x, u, O, ts, p) that is added to R, Algorithm **Insert**

must have been invoked with the node (u, s_f) as the child node for some $s_f \in F$ (Line 15 in **RAPQ** or Line 10 in **Insert**). Therefore, the proof proceeds by showing that node (u, s_f) with timestamp $(u, s_f).ts \in (ts - \omega, ts]$ for some $s_f \in F$ is added to the spanning tree T_x only if there exists a path p of length n with the same timestamp in G_{ts} from x to u satisfying R. For the base case of $n = 1$, assume there exists a tuple $(x, u, l, ts, \mathcal{D})$ where $\delta(s_0, l) = s_f$ for some $s_f \in F$. **Algorithm RAPQ** (Line 15) invokes Algorithm **Insert** with parameters (x, s_0) as the parent node and (u, s_f) as the child node, then streaming graph tuple $(x, u, O, (u, s_f).ts, p)$ is added to the result set (Line 6). Let's assume that there exists a path q of length $n - 1$ in G_{ts} from x to v where $t = \delta^*(s_0, \phi^p(p))$ and there exists a node (v, t) in T_x where $(v, t).ts = q.ts \in (ts - \omega, ts]$. For the node (u, s) to be added to the spanning tree T_x with timestamp $(u, s).ts \in (ts - \omega, ts]$, Algorithm **Insert** must have been invoked with (u, s) by Line 15 of Algorithm **RAPQ** or Line 10 of Algorithm **Insert**. In either case, there must be an edge $e(v, u) \in G_{ts}$ where $s = \delta(t, \phi(u, v))$, and $e.ts \in (ts - \omega, ts]$. Therefore, this implies that there exists a path of length n in G_{ts} from x to u, thus concluding the proof. □

Theorem 2. *The amortized cost of Algorithm **RAPQ** is $\mathcal{O}(n \cdot k^2)$, where n is the number of distinct vertices in the G_{ts} defined by the window $\mathcal{W}$ over the streaming graph S and k is the number of states in the corresponding automaton A of the the query Q_R.*

Proof. Consider a streaming graph edge $(u, v, l, ts, \mathcal{D})$ arriving for processing at time ts. Updating window G_{ts} with the incoming sge (Line 6) takes constant time. Thus, the time complexity of Algorithm **RAPQ** is the total number of times Algorithm **Insert** is invoked.

First, it is shown that the amortized cost of updating a single spanning tree T_x rooted at (x, s_0) is constant in window size. For an edge (u, v) with label l, there could be k many parent nodes $(u, s) \in T_x$ for each state s and k many child nodes (v, t) for each state t Consequently, there could be at most k^2 invocations of Algorithm **Insert** for a given spanning tree T_x. Upon arrival of the edge $e(u, v)$, Algorithm **Insert** is invoked with nodes (u, s) as parent and (v, t) as child either when (u, s) is already in T_x at time ts, $ts - \omega < (u, s).ts \leq ts$ (Line 15 in Algorithm **RAPQ**), or when (u, s) is added to T_x at a later point in time $(u, s).ts > ts$ (Line 10 in Algorithm **Insert**). Note that Algorithm **Insert** is invoked with these parameters at most once as Line 3 of Algorithm **Insert** extends a node (v, t) only if it is not in T_x. The second invariant (Lemma 1) guarantees that (u, s) appears in a spanning tree T_x at most once. Therefore, Algorithm **Insert** is invoked at most $m \cdot k^2$ over a sequence of m tuples. As there are at most n spanning trees in Δ, one for each $x \in G_{ts}$, the total amortized cost is $\mathcal{O}(n \cdot k^2)$. □

Algorithm ExpiryRAPQ:

```
input  : Snapshot graph G_ts,
         Window size ω
         timestamp ts,
         Spanning tree T_x
output:  The set of invalidated results R_I
R_I ← ∅
set P = {(v,t) ∈ T_x | (v,t).ts ≤ ts − ω} // potentially expired nodes
T_x ← T_x \ P // prune T_x
foreach (v,t) ∈ P do
    foreach edge e(u,v) ∈ G_ts do
        if (u,s) ∈ T_x ∧ t = δ(s, φ(u,v)) then
            P ← P \ Insert(T_x, (u,s), (v,t), e(u,v))
        end
    end
end
foreach (v,t) ∈ P do
    if t ∈ F then
        p ← PATH(T_x, (v,t))
        R ← R + (x, v, O, (v,t).ts, p)
    end
end
return R_I
```

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

4.3.2 Explicit Deletions

The majority of real-world applications process append-only streaming graphs where existing tuples in the window expire only due to window movements. However, there are applications that require users to explicitly delete a previously inserted edge. Algorithm **ExpiryRAPQ** proposed in Section 4.3.1 can be utilized to support such explicit edge deletions. Remember that in the append-only case, a node (v, t) in a spanning tree $T_x \in \Delta$ is only removed when its timestamp falls outside the window range. An explicit deletion might require $(v, t) \in T_x$ to be removed if the deleted edge is on the path from (x, s_0) to (v, t) in the spanning tree T_x. Algorithm **ExpiryRAPQ** is used to remove such nodes so that explicit deletions and window management are handled in a uniform manner.

Definition 24 (Tree Edge). *Given a spanning tree T_x at time ts, an edge $e(u, v)$ with label l is a tree-edge w.r.t T_x if (u, s) is the parent of (v, t) in T_x and there is a transition from state s to t with label l, i.e., $(u, s) \in T_x$, $(v, t) \in T_x$, $t = \delta(s, l)$, and $(v, t).pt = (u, s)$.*

Algorithm **Delete** finds spanning trees where a deleted edge (u, v) is a tree-edge (Line 3) as per Definition 24. Deletion of the tree-edge from (u, s) to (v, t) in T_x disconnects (v, t) and its descendants from T_x. Algorithm **Delete** traverses the subtree rooted at (v, t) and sets the timestamp of each node to $-\infty$, essentially marking them as expired (Line 5). Algorithm **ExpiryRAPQ** processes each expired node in Δ and checks if there exists an alternative path comprised of valid edges in the window. Algorithm **Delete** invokes Algorithm **ExpiryRAPQ** (Line 9) to manage explicit deletions using the same machinery of window management. Deletion of a non-tree edge, on the other hand, leaves spanning trees unchanged so no modification is necessary other than updating the window content G_{ts}.

Theorem 3. *The amortized cost of Algorithm **Delete** is $\mathcal{O}(n^2 \cdot k)$ over a sequence of explicit edge deletions.*

Proof. Consider the cost of an explicit deletion over a single spanning tree $T_x \in \Delta$, rooted at (x, s_0). Given a negative tuple $(u, v, l, ts, \mathcal{D})$, Line 3 identifies the corresponding set of

Algorithm Delete:

input : Negative tuple $sgt = (u, v, l, ts, \mathcal{D})$,
Snapshot graph G_{ts},
Window size ω

output: The set of invalidated results R_I

1 $R_I \leftarrow \emptyset$
2 **foreach** $T_x \in \Delta$ **do**
3   **foreach** $s, t \in S \mid t = \delta(s, l) \wedge (v, t) \in T_x \wedge (v, t).pt = (u, s)$ **do**
4         $T_{(x,v,t)} \leftarrow$ the subtree of (v, t) in T_x
5   **foreach** $(w, q) \in T_{(x,v,t)}$ **do**
6             $(w, q).ts = -\infty$
7   **end**
8   **end**
9     $R_I \leftarrow R_I \cup$ *ExpiryRAPQ*$(G_{ts}, \omega, T_x, ts)$
10 **end**
11 **return** R_I

tree edges in T_x in $\mathcal{O}(n \cdot k)$ time. For each such tree edge from (u, s) to (v, t) in T_x, Line 4 traverses the spanning tree T_x starting from (v, t) to identify the set of nodes that are possibly affected by the deleted edge, thus its cost is $\mathcal{O}(n \cdot k)$. Once timestamps of nodes in the subtree of (v, t) is set to $-\infty$, Line 9 invokes Algorithm **ExpiryRAPQ** to process all expired nodes in T_x, whose time complexity is $\mathcal{O}(m \cdot k^2)$. There can be at most $m \cdot k^2$ edges in the product graph of snapshot G_{ts} with m edges and automaton A with k edges. The amortized time complexity of maintaining a single spanning tree $T_x \in \Delta$ over a sequence of m explicit deletion is $\mathcal{O}(n \cdot k)$ since at most $n \cdot k$ of those edges are tree edges. Algorithm **Delete** does not need to process non-tree edges as a removal of a non-tree edge only need to update the window G_{ts}, which is a constant time operation. Therefore, the amortized cost of Algorithm **Delete** over a sequence of m explicit edge deletions is $\mathcal{O}(n^2 \cdot k)$. □

4.4 RPQ with Simple Path Semantics

RPQ evaluation on streaming graphs under the simple path semantics involves finding pairs of vertices $u, v \in V$ where there exists a simple path (no repeating vertices) p between u and v with a path label w in the language $L(R)$.

The decision problem for the static version of Regular Simple Path Query (RSPQ), i.e., deciding whether a pair of vertices $u, v \in V$ is in the result set of a RSPQ Q_R^O, is

NP-complete for certain fixed regular expressions, making the general problem NP-hard [121]. Mendelzon and Wood [121] show that there exists a batch algorithm to evaluate RSPQs on static graphs in the absence of conflicts, a condition on the cyclic structure of the graph G and the regular language $L(R)$ of the query Q_R^O.

Definition 25 (Suffix Language)**.** *Given an automaton* $A = (S, \Sigma, \delta, s_0, F)$*, the suffix language of a state* s *is defined as* $[s] = \{w \in \Sigma^* \mid \delta^*(s, w) \in F\}$*; that is, the set of all strings that take* A *from state* s *to a final state* $s_f \in F$*.*

Definition 26 (Containment Property)**.** *Automaton* $A = (S, \Sigma, \delta, s_0, F)$ *has the suffix language containment property if for each pair* $(s, t) \in S \times S$ *such that* s *and* t *are on a path from* s_0 *to some final state and* t *is a successor of* s*,* $[s] \supseteq [t]$*.*

The suffix language containment relation ia computed and stored for all pairs of states during query registration, i.e., the time when the query Q_R is first posed, and use these in the proposed streaming algorithm to detect conflicts. The conflicts can now be precisely defined.

Definition 27 (Conflict)**.** *There is a conflict at a vertex* u *if and only if a traversal of the product graph* $P_{G,A}$ *starting from an initial node* $(x, s_0) \in P_{G,A}$ *visits node* u *in states* s *and* t*, and* $[s] \not\supseteq [t]$*. In other words, a tree* T_X *is said to have a conflict between states* s *and* t *at vertex* u *if* (u, s) *is an ancestor of* (u, t) *in the spanning tree* T_x *and* $[s] \not\supseteq [t]$*.*

Example 8. *Consider the streaming graph and the query in Figures 4.1 and 4.2, respectively, and the their spanning tree given in Figure 4.3(a). The node* $(y, 2)$ *is added as a child of the node* $(v, 1)$ *when edge* (v, y) *arrives at* $t = 18$*. Based on Definition 27, there is a conflict at vertex* v *as the path* p *from the root node* $(x, 0)$ *visits the vertex* v *at states* 1 *and* 2*, and* $[1] \not\supseteq [2]$*.*

Batch Algorithm: Similar to the batch algorithm in Section 4.3, the batch RSPQ algorithm [121] starts a DFS traversal of the product graph from every vertex $x \in V$ with the start state s_0, and constructs a DFS tree, T_x. Each DFS tree maintains a set of markings that is used to prevent a vertex being visited more than once in the same state in a T_x. A node (u, s) is added to the set of markings only if the depth-first traversal starting from the node (u, s) is completed and no conflict is detected. Mendelzon and Wood [121] show that a RSPQ Q_R can be evaluated in $\mathcal{O}(n \cdot m)$ in terms of the size of the graph G by the batch algorithm in the absence of conflicts – the same as the batch algorithm for RAPQ evaluation presented in Section 4.3. A query Q_R^O on a graph G is conflict-free if: (i) the automaton A of R has the suffix language containment property, (ii) G is an acyclic graph, or (iii) G complies with a cycle constraint compatible with R. In following, the persistent RSPQ evaluation problem is considered where the notion of *conflict-freedom* [121] is shown to be applicable to sliding windows over streaming graphs, admitting an efficient evaluation algorithm in the absence of conflicts.

4.4.1 Append-only Streams

First, consider an incremental algorithm for RSPQ evaluation based on its RAPQ counterpart (Algorithm **RSPQ**) with implicit window semantics with complexity matching that of the batch algorithm for RSPQ evaluation on static graphs [121], i.e., it admits efficient solutions under the same conditions as the batch algorithm.

Definition 28 (Prefix Paths). *Given a node $(u, s) \in T_x$, the prefix path p for node (u, s) is defined as the path from the root to (u, s). The notation $p[v], v \in V$ is adopted to denote the set of states that are visited in vertex v in path p, i.e., $p[v] = \{s \in S \mid (v, s) \in p\}$.*

Definition 29 (Conflict Predecessor). *A node $(u, s) \in T_x$ is a conflict predecessor if for some successor (w, t) of (u, s) in T_x, (w, q) is the first occurrence of vertex w in the prefix path of (u, s) and there is a conflict between q and t at w, i.e., $[q] \not\supseteq [t]$.*

In addition to tree index Δ of Algorithm **RAPQ** in Section 4.3, Algorithm **RSPQ** maintains a set of markings M_x for each spanning tree T_x. The set of markings M_x for a spanning tree T_x is the set of nodes in T_x with no descendants that are conflict predecessors (Definition 29). In the absence of conflicts, there is no conflict predecessor and M_x contains all nodes in T_x. Algorithm **RSPQ** does not visit a node in M_x (Lines 16 in Algorithm **RSPQ** and 15 in Algorithm **Extend**) and therefore a node (u, s) appears in the spanning tree T_x at most once in the absence of conflicts. Consequently, Algorithm **RSPQ** maintains the second invariant of Δ and behaves similar to the Algorithm **RAPQ** presented in Section 4.3.1. On static graphs, the batch algorithm adds a node (u, s) to the set of markings only after the entire depth-first traversal of the product graph from (u, s) is completed, ensuring that the set M_x is monotonically growing. On the other hand, tuples that arrive later in the streaming graph S might lead to a conflict with a node (u, s) that is already in M_x, and Algorithm **RSPQ** removes (u, s)'s ancestors from the set of markings M_x. As described later, Algorithm **RSPQ** correctly identifies these conflicts and updates the spanning tree T_x and its set of markings M_x to ensure correctness. The conflict detection mechanism signals to the algorithm that the corresponding traversal cannot be pruned even if it visits a previously visited vertex. In other words, a node $(u, s) \notin M_x$ may be visited more than once in a spanning tree T_x to ensure correctness. Consequently,

Algorithm RSPQ:

input : Input streaming graph S,
Window size ω,
Regular expression R,
output label O
output: Output streaming graph S_O

$A(S, \Sigma, \delta, s_0, F) \leftarrow$ `ConstructDFA`(R)
`Initialize` Δ
$S_O \leftarrow \emptyset$
R $\leftarrow \emptyset$
foreach $sgt = (u, v, l, ts, \mathcal{D}) \in S$ **do**
  $G_{ts} \leftarrow G_{ts-1}$ (op) $e(u, v)$ `// update snapshot graph`

  ExpiryRSPQ$(G_{ts}, \omega, T_x, ts\ )\ \forall T_x \in \Delta$ `// on user-defined slide intervals`
  foreach $s, t \in S$ *where* $t = \delta(s, l)$ **do**
    if $s = s_0 \wedge T_u \notin \Delta$ **then**
      add T_u with root node (u, s_0)
    end
    foreach $T_x \in \Delta$ **do**
      if $(u, s) \in T_x \wedge (u, s).ts > ts - \omega$ **then**
        $p \leftarrow PATH(T_x, (u, s))$ `// the prefix path`
        if $t \notin p[v] \wedge (v, t) \notin M_x$ **then**
          $R \leftarrow R+$ Extend$(G_{ts},\ T_x, p, (v, t), e(u, v))$
        end
      end
    end
  end
end
foreach *sgt* $t \in R$ **do**
  push t to S_O
end

Algorithm Extend:

input : Snapshot graph G_{ts},
Spanning tree T_x rooted at (x, s_0),
Prefix path p,
child node (v, t),
edge (u, v)
output: Set of results R

$R \leftarrow \emptyset$
if $q = FIRST(p[v])$ *and* $[q] \not\supseteq [t]$ **then**
  Unmark(G_{ts}, T_x, p) `//` q `and` t `have a conflict at vertex` v
else
  if $t \in F$ **then**
    $R \leftarrow R + (x, v, O, (v, t).ts, p)$
  end
  if $(v, t) \notin T_x$ **then**
    $M_x \leftarrow M_x \bigcup (v, t)$
  end
  add (v, t) as (u, s)'s child in T_x
  $p_{new} \leftarrow p + [v, t]$
  $p_{new}.ts = min(e.ts, p.ts)$
  foreach *edge* $e(v, w) \in G_{ts}$ *s.t.* $\delta(t, \phi(e)) = r$ **do**
    if $r \notin p_{new}[w] \wedge (w, r) \notin M_x$ **then**
      $R \leftarrow R +$ Extend$(G_{ts}, T_x, p_{new}, (w, r), e(v, w))$
    end
  end
end
return R

Algorithm Unmark:

input : Snapshot graph G_{ts},
Spanning Tree T_x,
Prefix Path p

$Q \leftarrow \emptyset$
while $p \neq \emptyset \wedge (v,t) = LAST(p) \wedge (v,t) \in M_x$ **do**
  $M_x \leftarrow M_X \setminus (v,t)$
  $Q \leftarrow Q + (v,t)$
  $p \leftarrow PATH(T_x, (v,t).parent$
end
foreach $(v,t) \in Q$ **do**
  foreach *edge* $e(w,v) \in G_{W,\tau}$ *s.t.* $t = \delta(q, \phi(e))$ **do**
    if $(w,q) \in T_x \wedge t \notin p[v]$ **then**
      $p_{candidate} \leftarrow PATH(T_x, (w,q))$
      Extend(G_{ts}, $T_x, p_{candidate}, (v,t), e(v,w)$)
    end
  end
end

Algorithm ExpiryRSPQ:

input : Snapshot graph G_{ts},
Window size ω
timestamp ts,
Spanning tree T_x
output: The set of invalidated results R_I

$R_I \leftarrow \emptyset$
$E = \{(v,t) \in T_x \mid (v,t).ts \leq ts - \omega\}$ `// expired nodes`
$P \leftarrow M_x \cap E$
$T_x \leftarrow T_x \setminus E$ `// prune` T_x
$M_x \leftarrow M_x \setminus E$ `// prune` M_x
foreach $(v,t) \in P$ **do**
  foreach $(u,v) \in G_{ts}$ *s.t.* $(u,s) \in T_x \wedge t = \delta(s, \phi(u,v))$ **do**
    $p \leftarrow PATH(T_x, (u,s))$
    $P \leftarrow P\setminus$ Extend$(G_{ts}, T_x, p, (v,t), e(u,v))$
  end
end
foreach $(w,q) \in P$ **do**
  if *all siblings of* (w,q) *are in* M_x **then**
    $M_x \leftarrow M_x + (w,q).parent$
  end
  if $q \in F$ **then**
    $R_I \leftarrow R_I + (x, w, O, (w,q).ts, p)$
  end
end
return R_I

Algorithm **RSPQ** traverses every simple path that satisfies the given query Q_R if every node in T_x is a conflict predecessor ($M_x = \emptyset$), leading to exponential time execution in the worst case. In summary, Algorithm **RSPQ** differs from its arbitrary path semantics counterpart in two major points: (i) it may traverse a vertex in the same state more than once if a conflict is discovered at the vertex, and (ii) it keeps track of conflicts and maintains a set of markings to prevent multiple visits of the same vertex in the same state whenever possible.

For each incoming tuple $(u, v, l, ts, \mathcal{D})$, Algorithm **RSPQ** finds prefix paths of all $(u, s) \in T_x$ (Line 15); that is, the set of paths in T_x from the root node to (u, s) (note that there exists a single such node (u, s) and its corresponding prefix path if $(u, s) \in M_x$). Then it performs one of the following four steps for each node $(u, s) \in T_x$ and its corresponding prefix path p:

1. $t \in p[v]$: The vertex v is visited in the same state t as before, thus path p is pruned as extending it with (v, t) leads to a cycle in the product graph $P_{G,A}$ (Line 16 in **RSPQ** and Line 15 **Extend**).

2. $(v, t) \in M_x$: The target node (v, t) has already been visited in T_x and it has no conflict predecessor descendant. Therefore path p is pruned (Line 16 in **RSPQ**, 15 in **Extend**).

3. $q = FIRST(p[v])$ and $[q] \not\supseteq [t]$: States q and t have a conflict at vertex v (Line 2 in **Extend**), making (u, s) a conflict predecessor. Therefore, all ancestors of (u, s) in T_x are removed from M_x (Algorithm **Unmark**). During unmarking of a node $(v_i, s_i) \in M_x$, all $(w, q) \in T_x$ where $(w, v_i) \in G_{ts}$ and $s_i = \delta(q, \phi(w, v_i))$ are considered as candidate for traversal as they were previously pruned due to (v_i, s_i) being marked.

4. Otherwise path p is extended with (v, t), i.e., (v, t) is added as a child to (u, s) in T_x. (Line 4 in **Extend**)

As described previously, an important difference between the proposed streaming algorithm and the batch algorithm [121] is that the streaming version may remove nodes from the set of markings M_x whereas a node in M_x cannot be removed in the batch model. Hence, the batch algorithm can safely prune a path p if it reaches a node $(u, s) \in M_x$ as the suffix language containment property ensures correctness. The streaming model, on the other hand, requires a special treatment as M_x is not monotonically growing. Case 2 above prunes a path p if it reaches a node $(u, s) \in M_x$ as in the batch algorithm. Unlike the batch algorithm, a node (u, s) may be removed from M_x due to a conflict that is caused by

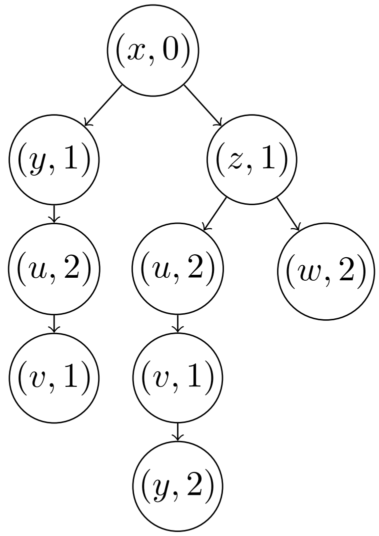


Figure 4.4: A spanning tree T_x constructed by Algorithm RSPQ for the example in Figure 4.2.

an edge that later arrives. This conflict implies that path p should not have been pruned. Case 3 above and Algorithm **Unmark** address exactly this scenario: ancestors of a conflict predecessor is removed from M_x.

Whenever a node (u, s) is removed from M_x due to a conflict at one of its descendants, Algorithm **Unmark** finds all paths that are previously pruned due to (u, s) by traversing incoming edges of $(u, s) \in G_{ts}$ and invokes Algorithm **Extend** for each such path. It enables Algorithm **Extend** to backtrack and evaluate all paths that would not be pruned by Case 2 if (u, s) were not in M_x, ensuring the correctness of the algorithm.

The following example illustrates this behaviour of Algorithm **RSPQ**.

Example 9. *Consider the streaming graph and the query in Figures 4.1 and 4.2, respectively, and the their spanning tree given in Figure 4.3(a). Assume for now that Algorithm* ***RSPQ*** *does not detect conflicts and only traverses simple paths in* G_{ts}*. After processing edge* (x, y) *at time* $t = 13$*, it adds node* $(u, 2)$ *as a successor of* $(y, 1)$*. Edge* (z, u) *arrives at* $t = 14$*, however* $(u, 2)$ *is not added as* $(z, 1)$*'s child as* $(u, 2)$ *already exists in* T_x*. Later at* $t = 18$*, edge* (v, y) *arrives, but* $(y, 2)$ *is not added to the spanning tree* T_x *as the path* $\langle x, y, u, v, y \rangle$ *forms a cycle in* G_{ts}*. As a result,* $(y, 2)$ *is never visited and* (x, y) *is never reported even though there exists a simple path in* G_{ts} *from* x *to* y*, that is* $\langle x, z, u, v, y \rangle$*.*

Instead, Algorithm ***RSPQ*** *detects the conflict at the vertex* v *between states* 1 *and* 2 *after edge* (v, y) *arrives at time* $t = 18$ *as* $FIRST(p[y]) = 1$ *and* $[1] \not\supseteq [2]$*. Algorithm*

Unmark *removes all ancestors of* $(y, 2)$ *from* M_x *and, during unmarking of* $(u, 2)$*, the prefix path* p *from* $(x, 0)$ *to* $(z, 1)$ *is extended with* $(u, 2)$*. Finally, Algorithm* ***Extend*** *traverses the simple path* $\langle x, z, u, v, y\rangle$ *and updates the result set. Figure 4.4 depicts the spanning tree* $T_x \in \Delta$ *at time* $t = 18$*.*

Similar to its arbitrary counterpart, Algorithm **RSPQ** invokes Algorithm **ExpiryRSPQ** at each user-defined slide interval β. It first identifies the set of candidate nodes whose timestamp is not in $(ts - \omega, ts]$ (Line 2). Unmarked candidate nodes ($M_x \setminus E$) can safely be removed from T_x as the unmarking procedure already considers all valid edges to an unmarked node. Hence, Algorithm **ExpiryRSPQ** reconnects a candidate node with a valid edge only if it is marked (Line 6). Finally, it extends the set of marking with nodes that are not conflict predecessors any longer (Line 12).

Theorem 4. *The algorithm* ***RSPQ*** *is correct and complete.*

Proof. **If:** If the proposed algorithm traverses the path p, it correctly adds it to the result set R and consecutively to the output streaming graph S_O (Line 6 and 24 in Algorithm **Extend**). The reason p is not traversed is due to a marked node (Case 2 of the proposed algorithm) as no vertex appears more than once in p (as it is a simple path). Let the last node visited in p be (v, t) and its successor on p be (w, r). The initial part of path p from (x, s_0) to (v, t) is not extended by (w, r) as $(w, r) \in M_x$ If (w, r) is removed from M_x due to a conflict predecessor descendant of (w, r), Algorithm **Unmark** guarantees that the initial part of path p from (x, s_0) to (v, t) is extended with (w, r) as $(v, t) \in T_x$ and $(v, w) \in E$ and $r = \delta(t, \phi(v, w))$ (Line 2 of Algorithm **Unmark**). As a result, the path p from (v, t) to (u, s_f) is discovered and $(x, u, O, (u, s_f).ts, p)$ is added to S_O. If (w, r) remains in M_x, (w, r) does not have any descendants that is a conflict predecessor. Therefore, (u, s) must have been traversed as a descendant of (w, r), adding $(x, u, O, (u, s).ts, p)$ to S_O.

Only if: Assume that p is not simple, meaning that there exists a node v that appears in p more than once. The first such occurrence is $(v, s_1) \in p$ and the last such occurrence is $(v, s_2) \in p$. For (v, s_2) to be visited, $[s_1] \not\supseteq [s_2]$ must have been false (Line 2 in Algorithm **Extend**). The containment property (Definition 26) implies that there exists a path p' from (v, s_1) to (u, s_f^2), $s_f^2 \in F$ such that the sequence of vertices on p' is identical to those in p from (v, s_2) to (u, s_f). Note that (v, s_1) and (v, s_2) are the first and last occurrences of v in p, therefore there exists a simple path in $P_{G,A}$ from (x, s_0) to $(u, s_f^2), s_f^2 \in F$ where the vertex v appears only once. A simple induction on the number of repeated vertices concludes that there is a simple path in G from x to u where the path label is in $L(R)$, and thus $(x, u, O, (v, s_f^2).ts, p)$ is added to S_O. □

Theorem 5. *The amortized cost of Algorithm **RSPQ** is $\mathcal{O}(n \cdot k^2)$, where n is the number of distinct vertices in the window $\mathcal{W}$ and k is the number of states in the corresponding automaton A of the query Q_R.*

Proof. The proposed algorithm might take exponential time in the size of the stream in the presence of conflicts as RSPQ evaluation is NP-hard in its general form [121]. Therefore, first focus on streaming RSPQ evaluation in the absence of conflicts and show that the cost of updating a single spanning tree T_x and its markings M_x is constant in the size of the stream.

The cost of Algorithm **RSPQ** for updating a single spanning tree T_x is determined by the total cost of invocations of Algorithm **Extend**. In the absence of conflicts, Algorithm **Extend** never invokes Algorithm **Unmark**, and the cost of updating R (Line 6), M_x (Line 9) and T_x (Line 11) are all constant. Therefore the cost of Algorithm **Extend** and thus the cost of Algorithm **RSPQ** are determined by the number of invocations of Algorithm **Extend**.

Algorithm **Extend** checks if a prefix path p whose last node in (u, s) for some $t = \delta(s, l)$ can be extended with (v, t). Each node (v, t) appears in T_x at most once. The first time Algorithm **Extend** is invoked with some prefix path p and node (v, t), path p is extended and node (v, t) is added to T_x and M_x (Line 4). Consecutive invocations of Algorithm **Extend** with node (v, t) do not perform any modifications on T_x or M_x as (v, t) is guaranteed to remain marked in absence of conflicts. Therefore, each node (v, t) appears only once in each spanning tree T_x in the absence of conflicts (a node is removed from M_x only if a conflict is discovered at Line 2). For an incoming tuple with edge (u, v) with label l, there can be at most k^2 pairs of prefix path p of (u, s) and node (v, t), for each $s, t \in S$. Algorithm **Extend** is invoked for each such pair at most once; either (i) when the edge $e(u, v)$ first appears in the stream and $(u, s) \in T_x$ but not (v, t) (Line 17), or (ii) $e(u, v)$ with label l previously appeared in the stream when (u, s) is first added to T_x and $(v, t) \notin T_x$ (Line 16). Over a stream of m tuples, Algorithm **Extend** is invoked $\mathcal{O}(m \cdot k^2)$ times for the maintenance of a spanning tree T_x. Therefore, amortized cost of maintaining a spanning tree T_x over a stream of m edges is $\mathcal{O}(k^2)$. Given that there are $\mathcal{O}(n)$ spanning trees, one for each $x \in V$, the amortized complexity of Algorithm **RSPQ** is $\mathcal{O}(n \cdot k^2)$ per tuple. □

Consequently, the amortized cost of Algorithm **RSPQ** is linear in the number n of vertices in the snapshot graph G_{ts}, similarly to its RAPQ counterpart (described in Section 4.3.2).

4.4.2 Explicit Deletions

Algorithm **RSPQ** processes negative tuples similar to its RAPQ counterpart. First, it identifies whether the edge of a negative tuple $(x, u, l, ts, \mathcal{D})$ corresponds to a tree edge in a spanning tree T_x. If so, it traverses the subtree of the deleted edge in T_x and sets the timestamp of each node $-\infty$. Similar to its RAPQ counterpart, it invokes Algorithm **ExpiryRSPQ** on the spanning tree T_x. Finally, Algorithm **ExpiryRSPQ** processes each expired node in T_x (whose timestamp is set to $-\infty$) and reconnects it to the spanning tree T_x if there exists another path from a valid node in T_x.

Similar to explicit edge deletion under arbitrary path semantics, the amortized time complexity of processing sequence of m explicit edge deletions is $\mathcal{O}(n^2 \cdot k)$ where n is the number of distinct vertices and k is the number of states in the corresponding automaton of a RSPQ Q_R.

4.5 Experimental Analysis

The feasibility of the proposed persistent RPQ evaluation algorithms are evaluated over both real and synthetic streaming graphs. First the throughput and the edge processing latency of Algorithm **RAPQ** is systematically evaluated over append-only streaming graphs, analyzing the factors affecting its performance (Section 4.5.2). Then, its scalability is assessed by varying the window size ω, the slide interval β and the query size $|Q_R|$ (Section 4.5.3). The overhead of Algorithm **Delete** over Algorithm **RAPQ** for explicit deletions is analyzed in Section 4.5.4 whereas Section 4.5.5 analyzes the feasibility of **RSPQ** for persistent RPQ evaluation under simple path semantics. Finally the proposed algorithms are compared with other systems (Section 4.5.6). Since this the first work to address RPQ evaluation over streaming graphs, the only possible comparison is against an emulation of persistent RPQ evaluation using RDF systems with SPARQL property path support.

The highlights of the results are as follows:

1. The proposed persistent RPQ evaluation algorithms maintain sub-millisecond edge processing latency on real-world workloads, and can process up-to tens of thousands of edges-per-second on a single machine.

2. The tail (99th percentile) latency of the algorithms increases linearly with the window size ω, confirming the amortized costs in Table 4.1.

3. The cost of expiring old tuples grows linearly with the slide interval β, which enables constant overhead regardless of β when amortized over the slide interval.

4. Explicit deletions can incur up to 50% performance degradation on tail latency, however the impact stays relatively steady with the increasing ratio of deletions.

5. Although RPQ evaluation under simple path semantics is NP-hard in its most-general form, the results indicate that the majority of the queries formulated on real-world and synthetic streaming graphs can be evaluated with $2\times$ to $5\times$ overhead on the tail latency.

6. The proposed algorithms achieve up to three orders of magnitude better performance when compared to existing RDF systems that emulate stream processing functionalities, substantiating the need for streaming algorithms for persistent RPQ evaluation on streaming graphs.

4.5.1 Methodology

Implementation

For this study, simple, in-memory prototypes of the proposed algorithms are developed — out-of-core processing is left as future work. The tree index Δ is implemented as a concurrent hash-based index where each vertex $v \in G_{ts}$ is mapped to its corresponding spanning tree T_x. Similarly, each spanning tree T_x is assisted with an additional hash-based index for efficient node look-ups. RAPQ (**RAPQ** and **ExpiryRAPQ**), RSPQ algorithms (**RSPQ**, and **ExpiryRAPQ**) employ *intra-query parallelism* by deploying a thread pool to process multiple spanning trees in parallel that are accessed for each incoming edge. In particular, the processing of each spanning tree T_x is handled by a single thread of execution, enabling consistency within a context of single spanning tree and parallelism across the tree index Δ. Window management is parallelized similarly.

Experiments are run on a Linux server with 32 physical cores and 256GB memory with the total number of execution threads set to the number of available physical cores. The metric is the time it takes to process each tuple; report the average throughput and the tail latency (99^{th} percentile) after ten minutes of processing on warm caches are reported. The prototype implementation is a closed system where each arriving sgt is processed sequentially. Thus, the throughput is inversely correlated with the mean latency.

Table 4.2: The most common RPQs used in real-world workloads (retrieved from Table 4 in [32]).

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

Workloads and Datasets

Although there exists streaming RDF benchmarks such as LSBench[2] and Stream WatDiv [62], their workloads do not contain any recursive queries, and they generate streaming graphs with very limited form of recursion. Therefore, persistent RPQs used in these experiments are formulated using the most common recursive queries found in real-world applications, leveraging recent studies [31, 32] that analyze real-world SPARQL query logs. The most common 10 recursive queries from [32] are selected; these cover more than 99% of all recursive queries found in Wikidata query logs. In addition, the most common non-recursive query (with no Kleene stars) is added for completeness, even though these are easier to evaluate as resulting paths have fixed size. Table 4.2 reports the set of real-world RPQs used in the experiments. For queries with variable number of edge labels, k is set to 3 as the SO graph only has three distinct labels. Table 4.3 lists the values of edge labels for graphs are used in these experiments. These queries are run over the following real and synthetic edge-labeled graphs.

Stackoverflow (SO) is a temporal graph of user interactions on this website containing 63M interactions (edges) of 2.2M users (vertices), spanning 8 years [135]. Each directed edge (u, v) with timestamp t denotes an interaction between two users: (i) user u answered user v's questions at time t, (ii) user u commented on user v's question, or (iii) comment at time t. SO graph is more homogeneous and much more cyclic than other datasets used in this study as it contains only a single type of vertex and three different edge labels. 7 out of 11 queries in Table 4.2 have at least 3 labels and cover all edges in the graph. Its highly dense and cyclic nature causes a high number of intermediate results and resulting paths; therefore, this graph constitutes the most challenging one for the proposed algorithms. The window size ω is set to 1 month and the slide interval β is set to 1 day unless specified

[2] https://code.google.com/archive/p/lsbench/

Table 4.3: Values of label variables in real-world RPQs (Table 4.2) for graphs used in this chapter.

Graph	Predicates
SO	knows, replyOf, hasCreator, likes
LDBC SNB	a2q, c2a, c2q
Yago2s	happenedIn, hasCapital, participatedIn

otherwise.

LDBC SNB is synthetic social network graph that is designed to simulate real-world interactions in social networking applications [53]. The update stream of the LDBC workload is extracted, which exhibits 8 different types of interactions users can perform. The streaming graphs generated by LDBC consists of two recursive relations: knows and replyOf. Therefore, Q_4, Q_5, Q_9 and Q_{10} in Table 4.2 cannot be meaningfully formulated over the LDBC streaming graphs; other remaining queries are used from Table 4.2. A scale factor of 10 is used that generates approximately 7.2M users and posts (vertices) and 40M user interactions (edges). LDBC update stream spans 3.5 months of user activity and the window size ω is set to 10 days and the slide interval β is set to 1 day unless specified otherwise.

Yago2s is a real-world RDF dataset containing 220M triples (edges) with approximately 72M different subjects (vertices).[3] Unlike existing streaming RDF benchmarks, Yago2s includes a rich schema (∼100 different labels) and the full set of queries listed in Table 4.2 can be represented over Yago2s. To emulate sliding windows on Yago2s RDF graph, a monotonically non-decreasing timestamp is assigned to each RDF triple at a fixed rate. Thus, each window defined over Yago2s has equal number of edges. The window size ω is set such that each window contains approximately 10M edges and the slide interval β to 1M edges, unless specified otherwise.

Additionally, gMark [19] graph and query workload generator is employed to systematically analyze the effect of query size $|Q_R|$. A pre-configured schema is deployed that mimics the characteristics of LDBC SNB graph and generates a synthetic graph with 100M vertices and 220M edges. It also creates synthetic query workloads where the query size ranges from 2 to 20 (the size of a query, $|Q_R|$, is the number of labels in the regular expression R and the number of occurrences of $*$ and $+$). Each RPQ is formulated by grouping labels into concatenations and alternations of size up to 3 where each group has a 50%

[3] https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/

probability of having $*$ and $+$. As gMark generates the entire LDBC SNB network as a single static graph, a monotonically non-decreasing timestamp is assigned to each edge at a fixed rate.

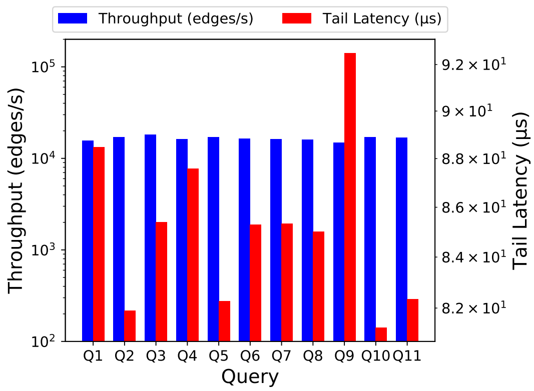
4.5.2 Throughput & Tail Latency

Figure 4.5 shows the throughput and tail latency of Algorithm **RAPQ** for all queries on all datasets. The algorithm discards a tuple whose label is not in the alphabet Σ_Q of Q_R as it cannot be part of any resulting path. Hence, what is reported is only the latency of tuples whose labels match a label in the given query. First, observe that the performance is generally lower for the SO graph due to its label density and its highly cyclic nature. The tail latency of Algorithm **RAPQ** is below 100ms for even the slowest query Q_3 on the SO graph and it is in sub-milliseconds for most queries on Yago2s and LDBC graphs. Similarly, the throughput of the algorithm varies from hundreds of edges-per-second for the SO graph (Figure 4.5(c)) to tens of thousands of edges-per-second for LDBC graph (Figure 4.5(b)).

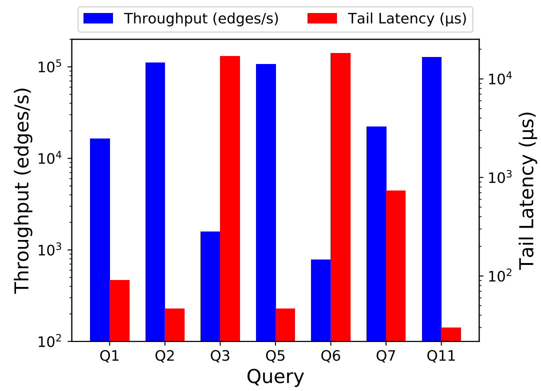
The total number of trees and nodes in the tree index Δ of Algorithm **RAPQ** on the SO graph provides a better understanding of the diverse performance characteristics of different queries (Figure 4.6). Recall that nodes and their corresponding paths in a spanning tree $T_x \in \Delta$ represent partial results of a persistent RPQ. Therefore, the amount of work performed by the algorithm grows with the size of tree index Δ. As expected, a negative correlation between the throughput of a query (Figure 4.5(c)) and its tree index size (Figure 4.6). It is known that cycles have significant impact on the run time of queries [31], and the analysis confirms this. In particular, Q_3 and Q_6 have the largest index sizes and therefore the lowest throughput, which can be explained by the fact that they contain multiple Kleene stars. Similarly, Q_4 and Q_9 have a Kleene star over alternation of symbols, which covers all the edges in the graph as the SO graph has only three types of user interactions. Therefore, Q_4 and Q_9 both have large index sizes, which negatively impacts the performance. In parallel, Q_{11} has the highest throughput on all datasets as it is the only fixed size, non-recursive query employed in the experiments.

4.5.3 Scalability & Sensitivity Analysis

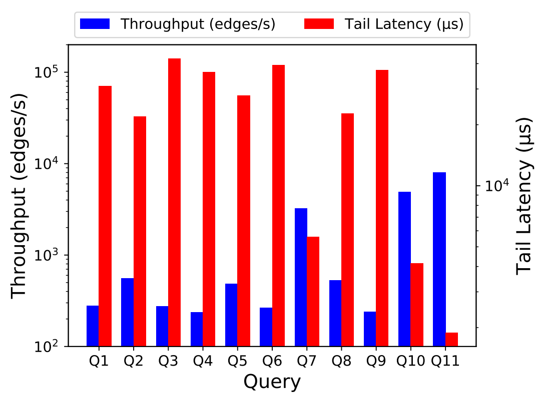
In this section, the impact of the window size ω and the slide interval β on algorithm performance is studied followed by an analysis of the performance implications of the use of DFAs and the query size $|Q_R|$.



(a) Yago2s



(b) LDBC SF10



(c) Stackoverflow

Figure 4.5: Throughput and tail latency of the Algorithm **RAPQ**. Y axis is given in log-scale.

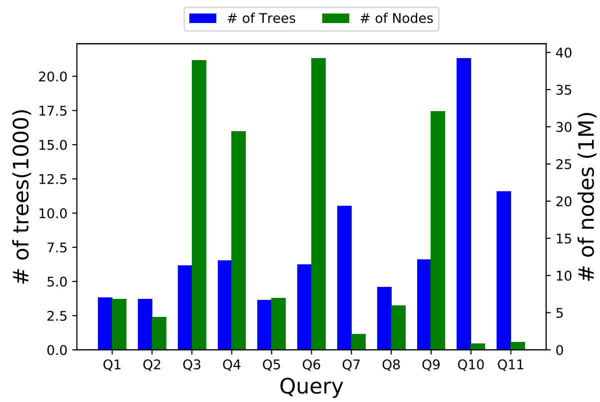


Figure 4.6: Size of the tree index Δ on the SO graph.

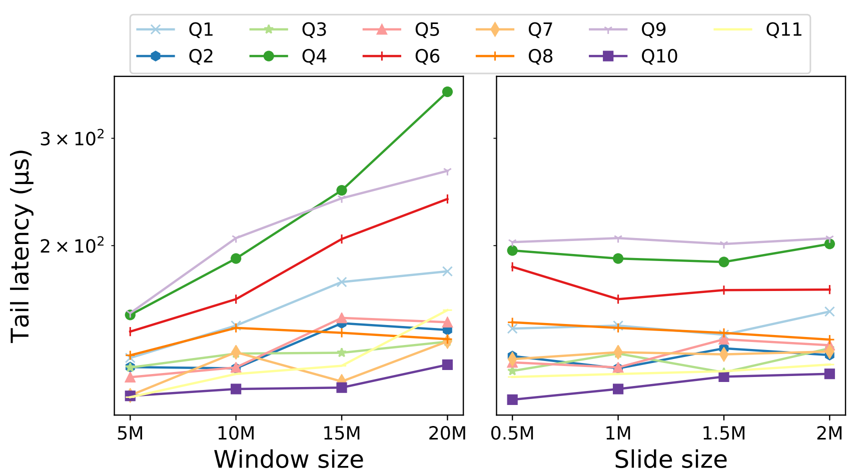


Figure 4.7: The tail latency on Yago2s graph with various ω and β.

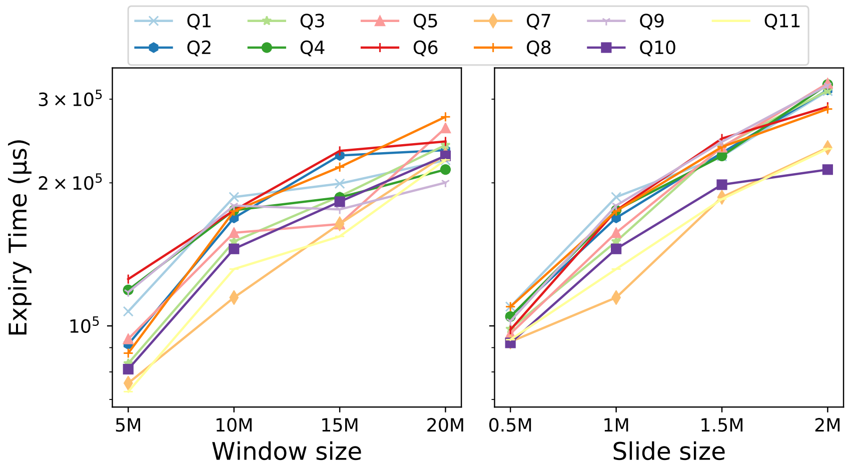
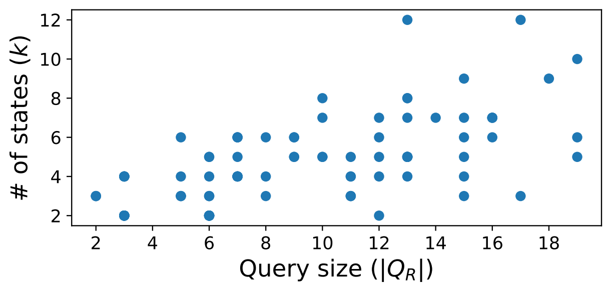


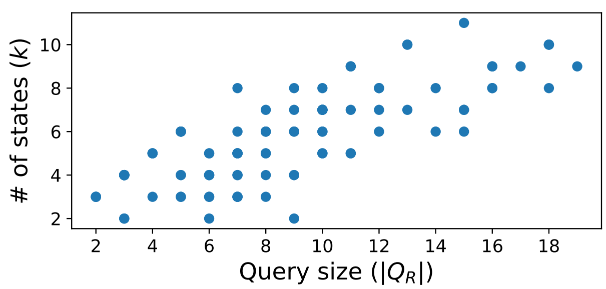
Figure 4.8: The window maintenance cost on Yago2s graph with various ω and β.

In this experiment, Yago2s dataset is used, since windows with a fixed number of edges that are created over Yago2s enable precise assessment of the impact of window size. Figure 4.7 presents the tail latency of the algorithm where the window size changes from 5M edges to 20M edges with 5M intervals. As expected, the tail latency for all tested queries increases with increasing ω, which conforms with the amortized cost analysis of Algorithm **RAPQ** in Section 4.3.1. Similarly, the time spent on Algorithm **ExpiryRAPQ** increases with increasing window size ω (Figure 4.8), in line with the complexity analysis given in Section 4.3.1. The same experiment is replicated using LDBC and Stream WatDiv datasets by varying the scale factor which in turn increases the number of edges in each window. The results show a degradation on the performance with increasing scale factor on Stream WatDiv, confirming the Yago2s findings. However, no similar trend is observed on LDBC graphs, which is due to the linear scaling of the total number of edges and vertices with the scale factor. Increasing the scale factor reduces the density of the graph, which may cause the proposed algorithms to perform even better in some instances due to a smaller tree index size. Furthermore, only a subset of queries can be formulated on these datasets as described previously. Therefore, only the findings on Yago2s graph are explicitly reported.

Next, the impact of the slide interval β on the performance of the proposed algorithms is assessed. Figure 4.7 plots the tail latency of Algorithm **RAPQ** against β and shows that the slide interval does not impact the performance. Recall that Algorithm **ExpiryRAPQ** is invoked periodically to remove expired tuples from the tree index Δ. It first identifies the set of expired nodes in a given spanning tree $T_x \in \Delta$, and searches their incoming edges to find a valid edge from a valid node in T_x. Therefore, Algorithm **ExpiryRAPQ** might



(a) gMark - LDBC Schema



(b) gMark - WatDiv Schema

Figure 4.9: The number of states k in corresponding DFAs of queries in the synthetic workload for (a) the social network schema that mimics the characteristics of LDBC SNB benchmark, and (b) the online shop schema that mimics the characteristics of WatDiv benchmark.

traverse the entire snapshot graph G_{ts} in the worst-case, regardless of the slide interval β. However, Figure 4.8 shows that the time spent on expiry of old tuples grows with increasing β, which causes its overhead to stay constant over time regardless of the slide interval β. Therefore, this algorithm is robust to the slide interval β. It also suggests that the complexity analysis of Algorithm **ExpiryRAPQ** given in Section 4.3.1 is not tight.

Finally, the effect of the query size $|Q_R|$ and the automata size k on the performance of the algorithms is assessed using a set of 100 synthetic RPQs that are generated using gMark. Combined complexities of the algorithms presented in Section 4.3 and Section 4.4 are polynomial in the number of states k, which might be exponential in the query size $|Q_R|$. Figures 4.9(a) and 4.9(b) show the total number of states in minimized DFAs for 100 RPQs that created using gMark's social network and online shop schemas, respectively. In practice, the size of the DFA does not grow exponentially with increasing query size for the considered RPQs despite the theoretical upper bound. Green et al. [74] has also

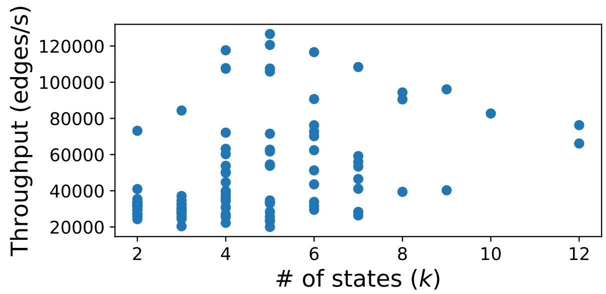


Figure 4.10: Throughput of the Algorithm RAPQ for the synthetic RPQ workload.

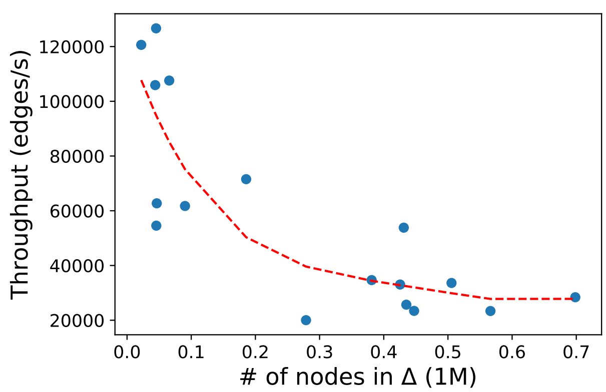


Figure 4.11: Throughput and tree index Δ size for synthetic RPQs with $k = 5$

indicated that exponential DFA growth is of little concern for most practical applications in the context of XML stream processing.

The next study considers the impact of the automata size k on performance. Figure 4.10 plots the throughput against the number of states k in the minimal automata for synthetic RPQs generated by gMark. No significant impact of k on performance is observed; yet, performance differences for queries with the same number of states in their corresponding DFA can be up to 6×. Such performance difference for RPQ evaluation has already been observed on static graphs and has been attributed to query label selectivities and the size of intermediate results [172]. To further verify this hypothesis in the streaming model, Figure 4.11 depicts the throughput versus the tree index Δ size for queries with $k = 5$. Confirming the results in Section 4.5.2, a negative correlation exists between the throughput of a query and its tree index size.

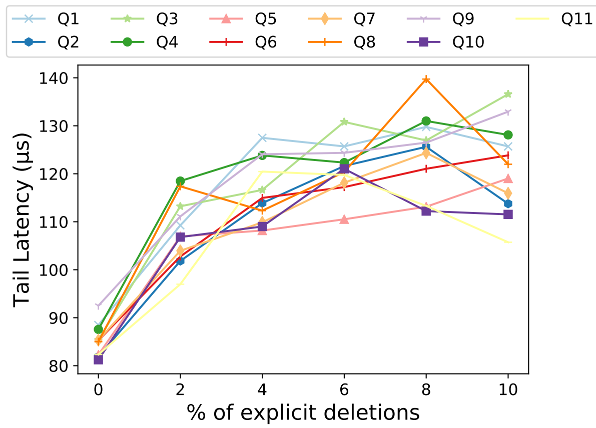


Figure 4.12: Impact of the ratio of explicit deletions on tail latency for all queries on Yago2s RDF graph.

4.5.4 Explicit Edge Deletions

Although most real-life streaming graphs are append-only, some applications require explicit edge deletions, which can be processed in the framework (Section 4.3.2). Explicit deletions are generated in this study by reinserting a previously consumed edge as a negative tuple and varying the ratio of negative tuples in the stream. Figure 4.12 plots tail latency of all queries on Yago2s varying deletion ratio from 2% to 10%. In line with the findings in the previous section, explicit deletions incur performance degradation due to the overhead of the expiry procedure (Figure 4.8). However, this overhead quickly flattens and does not increase with the deletion ratio. This is explained by the fact that the sizes of the snapshot graph G_{ts}, and the tree index Δ decrease with increasing deletion ratio.

4.5.5 RPQ under Simple Path Semantics

Results in Section 4.4 confirm that the amortized time complexity of Algorithm **RSPQ** under simple path semantics is the same as its RAPQ counterpart in the absence of conflicts.

In this section, the feasibility and the performance of this algorithm are empirically analyzed. Table 4.4 lists the queries that can be successfully evaluated under simple path semantics on each graph. Q_1, Q_4 and Q_{11} are restricted regular expressions, a condition that implies conflict-freedom in any arbitrary graph. Therefore, these queries are successfully

Table 4.4: Queries that can be evaluated under simple path semantics & the relative slowdown.

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

evaluated on all tested graphs (except Q_4 that cannot be defined over LDBC graph as discussed in Section 4.5.1). In particular, all queries are free of conflicts on Yago2s, and they can successfully be evaluated.

Table 4.4 also reports the overhead of enforcing simple path semantics on the tail latency. This overhead is simply due to conflict detection and the maintenance of markings for each spanning tree in the tree index Δ. Overall, these results suggest the feasibility of enforcing simple path semantics for majority of real-world queries, considering that most queries are conflict-free on heterogeneous, sparse graphs such as RDF graphs and social networks. Conversely, the arbitrary path semantics may be the only practical alternative for applications with homogeneous, highly cyclic graphs such as communication networks like Stackoverflow.

4.5.6 Comparison with Other Systems

This is the first work that investigates the execution of persistent RPQs over streaming graphs; therefore, there are no systems with which a direct comparison can be performed. However, there are a number of streaming RDF systems that can potentially be considered. These were reviewed in Chapter 2; unfortunately, as noted in that chapter, these systems only support SPARQL v1.0 and therefore cannot handle path expressions or recursive queries. With the introduction of property paths in SPARQL v1.1, the support for path queries have been added to a few RDF systems such as Virtuoso [54] and RDF-3X [77, 76]. However, these are designed for static RDF datasets, and they do not support persistent query evaluation. Therefore, in this study persistent query execution over Virtuoso is emulated to highlight the benefit of using incremental algorithms for persistent query evaluation on streaming graphs.

A middle layer is built on top of Virtuoso that emulates persistent query evaluation over sliding windows, similar to Algorithm **RAPQ**. This layer inserts each incoming tuple into Virtuoso and evaluates the query on the RDF graph that is constructed from the content of the window $\mathcal{W}$ at any given time t. For fairness, Virtuoso is configured to

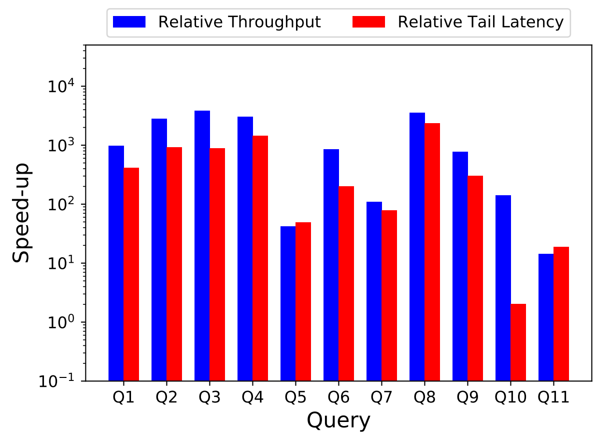


Figure 4.13: Relative speed-up of Algorithm RAPQ over Virtuoso for all queries on Yago2s RDF graph. Y axis is given in log-scale.

be memory-resident and its transaction logging is disabled to eliminate the overhead of transaction processing. In this study, Yago2s RDF graph is used with default ω and β for this experiment. Queries Q_1, Q_4, Q_6, Q_8, Q_9 and Q_{10} are modified by prepending a single predicate a to each query due to Virtuoso's limitation forbidding vertex variables on both ends of property paths at the same time. Figure 4.13 plots the average speed-up of **RAPQ** with respect to this simulation for both throughput and tail-latency. **RAPQ** consistently outperforms Virtuoso across all queries and provide up to 3 orders of magnitude better throughput and tail latency. This is because Virtuoso re-evaluates the RPQ on the entire window and cannot utilize the results of previous computations. Conversely, **RAPQ** indexes traversals in Δ and only explores the part of the snapshot graph G_{ts} that were not previously explored. In summary, these results suggest that incremental evaluation as in the proposed algorithms have significant performance advantages in executing RPQs over streaming graphs.

Remark 4 (Window maintenance cost)**.** *The combined time complexity of the window management (expiry) routine is $\mathcal{O}(n \cdot m \cdot k^2)$, which implies that the expiry procedure, in the worst case, might traverse the entire snapshot graph G_{ts} that is constructed from the contents of the window $\mathcal{W}$ at time ts. However, this cost is amortized over the slide interval β, as described in Section 4.3.2. Figure 4.8 shows that the time spent on the expiry of old tuples grows linearly with increasing β, which causes its overhead to stay constant over time regardless of the slide interval β. In addition, the time spent for the expiry of old tuples on Virtuoso is measured, similar to those of Algorithm **ExpiryRAPQ**. All the queries listed*

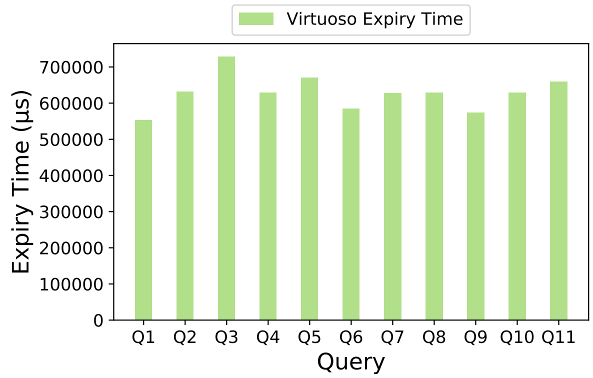


Figure 4.14: The average window maintenance cost for Virtuoso on Yago2s RDF graph with $\omega = 10M$ and $\beta = 1M$.

in Table 4.2 are used on Yago2s RDF graph with $\omega = 10M$ and $\beta = 1M$ so the results are comparable with ones reported in Figure 4.8. Figure 4.14 shows the average window maintenance cost on Virtuoso for $Q_1 - Q_{11}$; algorithms proposed in this chapter spent less time on window management across all the tested queries.

4.6 Discussion

This chapter studies the evaluation of Regular Path Queries over streaming graphs. As identified in Chapter 3, path navigation queries are an essential feature of graph querying, and the RPQ model provides the basic navigational mechanism to express path navigation queries adapted by many practical graph query languages. However, the existing literature on RPQ evaluation solely focus on static graphs. The characterization in this chapter of the design space of persistent RPQ evaluation algorithms allows the study of alternative path semantics and window semantics in a uniform manner: algorithms are proposed for both arbitrary and simple path semantics that can handle explicit deletions. In particular, the algorithm for simple path semantics has the same complexity as the algorithm for arbitrary path semantics in the absence of conflicts, and it admits efficient solutions under the same condition as the batch algorithm. Experimental analyses using a variety of real-world RPQs and streaming graphs show that proposed algorithms can support up to tens of thousands of edges-per-second while maintaining sub-second tail latency.

The unbounded nature of the streaming graphs makes it infeasible to employ blocking, batch algorithms for query evaluation; therefore, it is crucial to employ query evaluation algorithms with non-blocking behaviour in the streaming graph query processing framework proposed in this thesis. Algorithms presented in this chapter facilitate incremental evaluation of the path navigation fragment of the queries targeted in this thesis. In the next chapter, a Streaming Graph Algebra (SGA) is proposed, which precisely defines the semantics of query operators and query execution plans in the context of streaming graphs. The algorithms proposed in this chapter are utilized as physical operator implementations in the SGA query processor prototype.

Chapter 5

An Algebraic Framework for Evaluation Streaming Graph Queries

5.1 Introduction

There is a plethora of algorithms for evaluating persistent queries over streaming graphs. These specialized algorithms are largely tailored for the needs of singular applications and often rely on different semantics and computational models such as subgraph pattern queries [9, 90, 42] and its specialized forms [103], cycle detection queries [137] and path navigation queries [131]. A general-purpose streaming graph query processor needs to unify all these functionality in a principled manner. The lack of a foundational framework inhibits the development of a general-purpose query processor for streaming graphs.

Chapter 3 introduced SGQ as a formal query model that addressed the requirements of streaming graph querying outlined in this thesis. SGQ constitutes a tool to describe the semantics of streaming graph queries targeted in this thesis and to study its expressivity and complexity. Nonetheless, as a logic-based, declarative language, SGQ does not describe how streaming graph queries are evaluated. Chapter 4 studied the evaluation of RPQ over streaming graphs and provided non-blocking, incremental algorithms for this important subclass of SGQ. The objective of this chapter is to study the design of a streaming graph query processor for the SGQ model.

A crucial task in designing general-purpose streaming graph query processor is to identify a set of primitive operators that can be used as the building blocks of query execution plans with the following requirements:

- their semantics should be consistent with the SGQ model;
- they should be expressive enough to support the evaluation of queries that can be represented in the SGQ model;
- they should be closed over the *streaming graph* data model (Section 3.2), making it possible to construct complex pipelines; and
- they should yield to incremental, non-blocking implementations.

This chapter introduces the Streaming Graph Algebra (SGA) as the foundational basis for representing query evaluation plans and describes a prototype implementation of a streaming graph query processor based on SGA. SGA defines a stream-native operator algebra for SGQ as a temporal generalization of the Regular Property Graph Algebra (RPGA) [30]. This chapter provides a concrete algorithm to generate an SGA expression for any given SGQ, showing that SGA has the expressive power to formulate query evaluation plans for all queries that can be represented in the SGQ model. All SGA operators take and return streaming graphs as inputs and outputs, making it possible to form arbitrarily complex query evaluation pipelines while ensuring correctness. An important characteristics of the SGA is the explicit use of windowing primitives. As previously described in Chapter 3, time-based sliding windows are used to restrict the scope of computations over potentially unbounded streams. Rather than integrating window semantics into every single operator, SGA simplifies operator semantics through an explicit time-based sliding windowing operator that adjusts validity intervals of sgts. This chapter also provides at least a single, non-blocking implementation of each SGA operator, and describes a concrete implementation of a streaming query processor based on the Timely Dataflow (TD) system [125]. TD provides abstractions to model low-level system details such as operator scheduling, inter-operator queues etc., and it lend itself to realizing the framework proposed in this chapter by implementing the SGA operators and query execution plans using TD abstractions.

In the remainder of this chapter, Section 5.2 first introduces the Streaming Graph Algebra (SGA) and the translation of SGQs into SGA expressions. Section 5.3 provides an overview of a streaming graph query processor based on SGA, and Section 5.4 describes physical operator implementations in details. Finally, Section 5.5 presents an experimental evaluation of a prototype implementation of SGA-based streaming graph query processor, and Section 5.6 concludes this chapter by summarizing its contributions in the context of the streaming graph query processing framework proposed in this thesis.

5.2 Streaming Graph Algebra

This section presents the logical foundation of the streaming graph query processing framework. The streaming graph algebra (SGA) and the semantics of its operators are first introduced (Section 5.2.1). SGA's role in the proposed framework is similar to that of relational algebra (RA) in relational systems: it enables formulation of query plans independent of specific physical implementations. It differs from RA in the following ways to tackle the aforementioned challenges of streaming graph querying (**R1-R5** in Chapter 1):

- SGA is a closure of a set of operators over *graph streams*, not static relations – this distinction is important;
- SGA operators generalize their non-temporal counterparts through implicit treatment of sgts' validity intervals;
- time-based sliding windows and path navigations are specified via novel WSCAN and PATH, respectively;
- SGA supports processing of paths as first-class citizens.

Section 5.2.2 describes transformation of SGQs (Definition 17) into canonical SGA expressions and illustrates logical query plans, and Section 5.2.3 discusses its closedness and composability.

5.2.1 SGA Operators

For ease of exposition, the rest of this chapter assumes that inputs to each SGA operator are partitioned into one more streaming graphs S_a based on tuple labels where each S_a contains sgts with the same label $a \in \Sigma$ (see Section 3.2 for a detailed discussion). The output of each operator is also a streaming graph S_o where each sgt has the label $o \in \Sigma \setminus \phi(E^I)$.[1] SGA contains the following operators: windowing (Definition 30), filter (Definition 31), union (Definition 32), subgraph pattern (Definition 33), and path navigation (Definition 34).

[1] $\phi(E^I) \subset \Sigma$ 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 3.3.

Definition 30 (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 ω and the optional slide interval β, i.e.,* $\mathcal{W}_{\omega,\beta}(S^I) := S : \left[(u, v, l, [t, exp), \mathcal{D} : e(u, v, l)) \mid (u, v, l, t) \in S^I \wedge exp = \lfloor t/\beta \rfloor \cdot \beta + \omega\right]$.

The window size ω determines the length of the validity interval of sgts and the slide interval β controls the granularity at which the time-based sliding window progresses [15, 131]. If β is not provided, default is $\beta = 1$, i.e., single time instant with the smallest granularity, and it defines a sliding window that progresses at every time instant.

The WSCAN operator defines the semantics of time-based sliding windows. It acts as an interface between the external streaming graph sources and the query plans and it is responsible for providing data from input graph streams to a query plan, similar to the *scan* operator in relational systems. WSCAN manipulates the implicit temporal attribute of sgts and associates a time interval to each sgt representing its validity. The use of *time-interval* representing streaming graphs provide a concise representation for validity of sgts by treating time differently than the data stored in the graph. (see Remark 1 for a detailed discussion). The use of an explicit windowing operator makes it possible to distinguish operator semantics from window semantics and eliminates the redundancy caused by integrating sliding window constructs into each operator of the algebra. SGA operators access and manipulate validity intervals implicitly, generalizing their non-streaming counterparts with implicit handling of time.

Example 10. *Consider the real-time notification task of Example 1 with a 24-hour window of interest.* WSCAN $\mathcal{W}_{24}$ *sets 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 3.2.*

Definition 31 (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:*

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

Definition 32 (UNION)**.** *Union $\cup^{[d]}$ with an optional output label $d \in \Sigma \setminus \phi(E^I)$ merges sgts of two streaming graphs S_1 and S_2, and assigns the new label d if provided. Formally:*

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

Definition 33 (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 *are endpoints of sgts in* S_{l_i}*, and* $src, trg \in \{src_1, trg_1, \cdots, src_n, trg_n\}$ *are the endpoints of resulting sgts, and* $d \in \Sigma \setminus \phi(E^I)$ *represent the label of the resulting sgts. Formally:*

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

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 11. *Consider the real-time notification query given in Example 1; the* recentLiker *relationship defined in the form of a triangle pattern can be represented with* PATTERN $\bowtie_{\phi}^{src1,src4,RL}$ *where* $\phi = (trg_1 = trg_2 \wedge src_1 = src_3 \wedge src_2 = trg_3)$*. Its output over the streaming graph, given in Figure 3.2, consists of sgts* $(y, RL, u, [28, 37), (y, RL, u))$ *and* $(u, RL, v, [29, 31), (u, RL, v))$ *that correspond to derived edges with label* recentLiker*.*

SGA operators may produce multiple value-equivalent sgts with adjacent or overlapping validity intervals. Unless otherwise specified, such sgts in resulting streaming graphs of SGA operators are coalesced to maintain the set semantics of streaming graphs and their snapshots (Definition 10). To illustrate, consider PATTERN in the above example: over the streaming graph given in Figure 3.2, 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.

Definition 34 (PATH)**.** *The streaming* ***path navigation operator*** *with RPQ semantics 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 sgt* $t = (u, v, l, [ts, exp), \mathcal{D} : p)$ *is an answer for* $\mathcal{P}_R^l$ *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:*

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

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 RPQ model where path constraints are expressed using a regular expression over the set of labels [171]. Path navigation queries in the RPQ model are evaluated under *arbitrary* and *simple* path semantics (as discussed in detail in Chapter 4). 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 [12, 171, 18]. The remainder of this chapter adopts the arbitrary path semantics due to its widespread adoption in modern graph query languages [11, 12, 165], and the tractability of the corresponding evaluation problem [18].

Example 12. *In the example of Figure 1 the path navigation over the derived* recentLiker *edges is represented by* PATH $\mathcal{P}^{RLP}_{RL^+}$. *Its output over the resulting streaming graph of* PATTERN *of Example 11 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* recentLikerPath *of length one and two.*

Most existing work on RPQ focuses on finding pairs of vertices that are reachable by a path conforming to given regular expression [121, 106, 94, 131]. By adapting the materialized path graph model (Definition 6), Streaming Graph Algebra with its PATH operator 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) [30], which is itself based on Regular Queries (RQ). Of course, both RPGA and RQ formulate graph queries over static graphs, while 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 window semantics are explicitly defined via the WSCAN operator, and the semantics of the remaining SGA operators are defined such that they satisfy the *snapshot reducibility* (Definition 15). That is, the snapshot of the result of an SGA operator over 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.

5.2.2 Formulating Query Plans in SGA

SGA can express all queries that can be specified by SGQ (Section 3.3). This section provides an algorithm for the conversion.

Given a SGQ $Q(S, \mathcal{W}_\omega)$ over a streaming graph and a time-based sliding window definition, Algorithm **SGQParser** produces the canonical SGA expression. The algorithm 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.

Theorem 6. *There exists a SGA expression $e \in SGA$ for any $Q \in SGQ$.*

Proof. The dependency graph of an RQ is acyclic as RQ is non-recursive (Definition 14); hence, Line 2 is guaranteed to define a partial order over Q's predicates. Algorithm **SGQParser** generates an SGA expression for each predicate in this order (Line 4) and caches it in *exp* array. In particular, Line 8 generates an SGA expression for each EDB predicate and Line 11 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 15 generates a PATTERN expression. Finally, Line 16 generates a UNION expression if there are multiple rules with the same head predicate d. Due to the partial order defined by the dependency graph G_Q, *exp* is guaranteed to have SGA expressions for each predicate $r_j (1 \leq j \leq i)$ when processing predicate r_i. Once all predicates are processed, Line 25 returns the SGA expression of the *Answer* predicate. Hence, Algorithm **SGQParser** correctly constructs an SGA expression for a given SGQ. □

The complexity of evaluating SGA expressions is the same as RQ given their relationship noted above: NP-complete in combined complexity and NLogspace-complete in data complexity [138, 30].

Figure 5.1 (left) illustrates the logical plan for the same SGQ that consists of SGA operators.

Algorithm SGQParser:

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

$G_Q \leftarrow$ `Graph`(Q) `// dependency graph`
$[r_1, \cdots, r_n] \leftarrow$ `TopSort`(G_Q) `// topological sort`
$exp \leftarrow []$ `// empty mapping`
for $1 \leq i \leq n$ **do**
  switch r_i **do**
    case $l(src, trg), l \in \phi(E^I)$ **do**
      $exp[l] \leftarrow \mathcal{W}_\omega(S_l)$
    end
    case $l^*(x, y) asd$ **do**
      $exp[d] \leftarrow \mathcal{P}^d_{l*}(exp[l])$
    end
    otherwise do
      $d(src, trg) \leftarrow r_i.head$, $[b_1, \cdots, b_n] \leftarrow r_i.body$
      $\Phi \leftarrow$ `GenPred`$(r_i.body)$
      $e \leftarrow \bowtie^{src,trg,d}_{\Phi} (exp[b_1], \cdots, exp[b_n])$
      if $exp[d] \neq \emptyset$ **then**
        $exp[d] \leftarrow exp[d] \cup e$
      end
      else
        $exp[d] \leftarrow e$
      end
    end
  end
end
return $exp[Answer]$

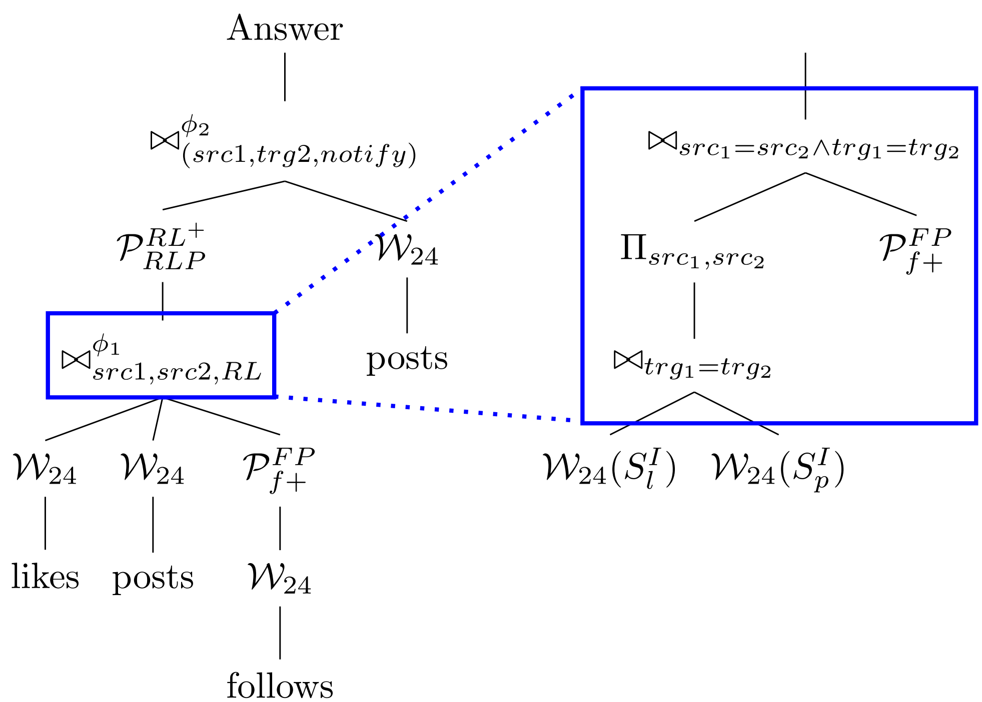


Figure 5.1: (left) Logical plan for the SGA expression in Example 13, and (right) binary join tree for its PATTERN.

Example 13 (Canonical Translation)**.** *For the real-time notification task in Example 1 and its corresponding RQ in Example 3, Algorithm **SGQParser** generates the following canonical SGA expression for its corresponding SGQ with a sliding window of 24 hours:*

$$\begin{aligned}
&\bowtie_{\phi_2}^{(src1,trg2,notify)} \Bigg(\\
&\quad \mathcal{P}_{RL^+}^{RLP}\Big(\bowtie_{\phi_1}^{src1,src2,RL} \big(\mathcal{W}_{24}(S_l), \mathcal{W}_{24}(S_p), \mathcal{P}_{f^+}^{FP}\big(\mathcal{W}_{24}(S_f)\big)\big)\Big), \\
&\quad \mathcal{W}_{24}(S_p) \quad \Bigg) \\
&\phi_1 = (trg_1 = trg_2 \wedge src_1 = src_3 \wedge src_2 = trg_3), \\
&\phi_2 = (trg_1 = src_2)
\end{aligned}$$

5.2.3 Closedness and Composability

Algebraic closure is a required property of any query algebra as it enables query rewriting and query optimization. Composability is a desired feature for a declarative query language as it facilitates query decomposition, view-based evaluation, query rewriting etc. SGA operators are closed over streaming graphs as defined in Section 3.2; that is, the output of an SGA operator is a valid streaming graph if its inputs are valid streaming graphs. Thus SGA queries are composable, i.e., the output of one query can be used as input of another query.

SGQ language is also closed (Theorem 6) – each query takes one or more streaming graphs as input and produces a streaming graph as output. It is also composable as the output of a query can be the input of the subsequent query. As such, G-CORE variation that is used as the user-level query language example in this thesis (Section 3.3.2) attains composability exactly as its original version is composable over property graphs [11]. This is in contrast to the other graph query languages that lack an algebraic basis, e.g., SPARQL and Cypher are not composable and may not be closed. Cypher 9 requires graphs as input, but produces tables as output so the language is neither closed nor composable – the results of a Cypher query cannot be used as input to a subsequent one without additional processing. SPARQL can produce 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 [30].

5.3 Query Processor Overview

This section describes implementation details of a prototype streaming graph query processor[2] based on SGA. The goal is to build a plausible conceptual framework for expressing and evaluating SGQ – not to have a full-fledged streaming graph management system. Consequently, the focus in this chapter is the construction of query execution plans and physical implementations of logical SGA operators. Optimization of query evaluation plans is discussed in the following chapter.

Conceptually, SGQ can be evaluated by repeatedly evaluating from scratch the corresponding one-time query at each point in time (Section 3.3.1). Albeit semantically correct, such a re-execution strategy is, of course, infeasible. Streaming systems instead focus on the *incremental* evaluation of persistent queries where re-running the query from scratch is avoided by computing the changes to the output in real-time as the stream is ingested. Such data-driven (push-based), incremental execution is key to supporting high ingestion rates as opposed to demand-driven (pull-based) query processing employed in traditional systems [71]. Streaming systems like Apache Flink, Spark Streaming and Timely Dataflow (TD) provide abstractions for communication, scheduling, distribution etc., enabling users to build efficient streaming applications without focusing on low-level system issues. With proper care to implementing windowing constructs, operator semantics, etc., any streaming system that supports stateful, iterative (recursive) computations can be used to evaluate persistent graph queries. Apache Flink and Spark Streaming do not have support for iterative computations in the streaming settings – Flink's iteration API and Spark's GraphX library can handle recursion, but they are both limited to batch computations. Extensions of Flink's Iteration API and GraphX to support incremental computations are not straightforward and require adjustments to these systems that go beyond the research goals of this thesis. TD, on the other hand, provides abstractions to model such computations, and it lends itself to realizing a prototype streaming graph query processor by implementing the physical algebra operators using TD abstractions. Consequently, the prototype implementation presented here uses TD as the underlying execution engine and focus on providing building blocks for expressing and evaluating streaming graph queries while leaving low-level stream handling to the underlying engine.

Applications in TD are expressed as a directed graph of operations where vertices correspond to user-defined computations and edges correspond to the flow of data between them. In executing an SGQ, the query processor first creates a logical plan from the canonical SGA expression of the given SGQ using the Algorithm SGQParser (Section

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

5.2.2). The syntax tree of the canonical SGA expression, where leaf nodes represent input streaming graphs, intermediate nodes represent logical SGA operators, and edges represent the stream of tuples between the operators, corresponds to the logical query plan. Figure 5.1 (left) depicts the logical query plan generated from the canonical SGA expression in Example 13 for the real-time notification task in Example 1. The physical execution plan in the form of a TD dataflow graph for a given logical plan is constructed by:

1. creating source vertices for the leaves of the logical query plan that consume input graph streams;
2. replacing logical SGA operators with physical operator implementations (to be described momentarily in Section 5.4); and
3. creating a sink vertex for the root of the logical query plan that pushes results back to the application.

Consequently, resulting physical execution plans (dataflow graphs) are tree-shaped, similar to the logical plans that are based on the canonical SGA expressions. Figure 5.2(a) illustrates the physical execution plan in form of a TD dataflow constructed using the logical plan in Figure 5.1. Physical operator implementations, i.e., vertices of these physical execution plans, are described in Section 5.4 in detail.

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

TD's Differential Dataflow (DD) layer [120] provides a set of built-in, high-level programming primitives (operators) that can be used to compose arbitrary dataflows for general-purpose computations, and it automatically incrementalizes these. Consequently, DD can be asked to evaluate SGQ by (i) creating a dataflow of DD operators for a given SGQ and (ii) maintaining the window content as an evolving collection. Indeed, such a strategy is used as a competitive baseline for evaluating the performance of the streaming graph query processor implementation described in this chapter (Section 5.5.2). In particular, the physical execution plan in the form of a dataflow of DD operators for a given SGQ is constructed by:

1. generating the logical plan using the canonical SGA expression as described above;

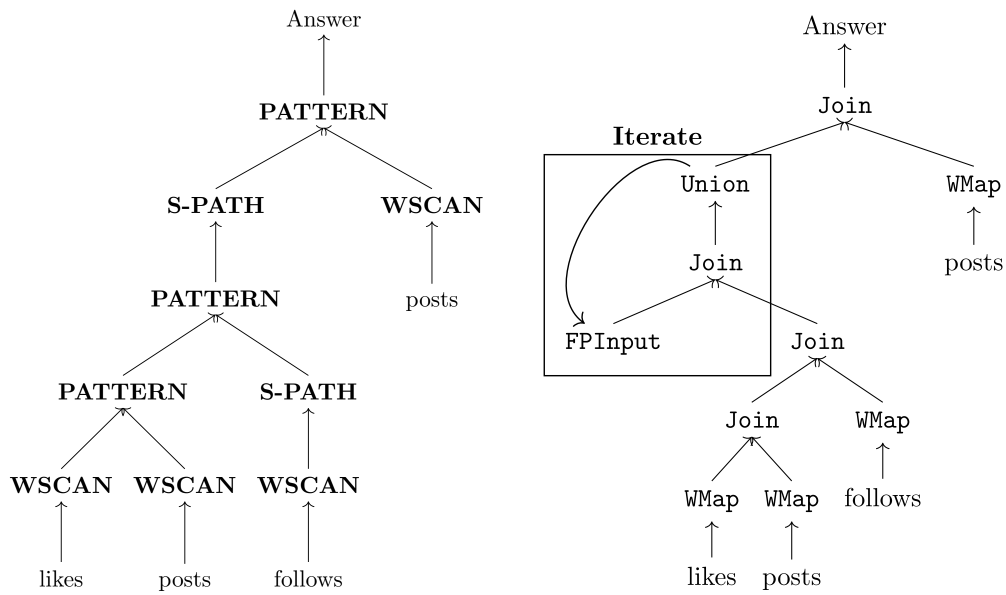


Figure 5.2: SGA and DD based physical execution plans based on the logical query plan in Figure 5.1 for the the real-time notification task in Example 1.

2. representing sliding windows over input graph streams as dynamic collections where window movements corresponds to insertions and deletions to the underlying collection;

3. mapping stateless operators FILTER and UNION to DD's *filter* and *concat* operators;

4. mapping PATTERN operator to DD's *join*; and

5. mapping PATH operator to DD's *iterate*.

Figure 5.2(b) illustrates the physical execution plan in form of a DD dataflow constructed using the logical plan in Figure 5.1. However, DD's generality comes at a performance cost for evaluating SGQ, as shown in Section 5.5.2. Next section describes how to devise physical operator implementations specific to SGQ by utilizing the properties of the SGQ model.

5.4 Physical Operator Algebra

This section describes the physical operator implementations incorporated in the streaming graph query processor described in the previous section. Note that these are exemplars to demonstrate the implementability of the SGA operators and to show their effectiveness as illustrated in the experimental study (Section 5.5); other physical implementations are certainly possible and it is expected that further research on streaming graph querying and graph algebras will uncover alternatives (as it has occurred in relational query processing).

5.4.1 Stateless Operators

Physical operator implementations for streaming systems have two requirements: they should be push-based and non-blocking, so they do not need the entire input to be available before producing the first result. Stateless operators produce a resulting sgt by processing a single incoming sgt; therefore, physical implementation of stateless operators do not need to maintain internal state. The standard dataflow implementations of stateless FILTER and UNION operators can be used in SGA, and WSCAN can be implemented via *map* operator that adjusts the validity intervals of sgts based on window specifications.

WSCAN

The time-based sliding window operator WSCAN takes an input graph stream and produce a streaming graph where the validity interval of each sgts is set based on the window specification (Algorithm **WSCAN**).

Algorithm WSCAN:

input : Input graph stream S^I, window size ω, output label d
output: Streaming graph S_O

$S_O \leftarrow \emptyset$
foreach $(u, v, l, t, \mathcal{D}) \in S^I$ **do**
  push $(src, trg, d, [t, t+\omega), \mathcal{D} : e(u, v, l))$ to S_O
end

FILTER

FILTER evaluates a predicate Φ on each incoming sgt; it appends the sgts to the output streaming graph if the predicate evaluates to true and discards it otherwise (Algorithm **FILTER**).

Algorithm FILTER:

```
input  : Streaming graph S, predicate φ, output label d
output: Streaming graph S_O
S_O ← ∅
foreach t = (u, v, l, [ts, exp), D) ∈ S do
    if Φ((src, trg, l, D)) then
        // evaluate the predicate
        push (src, trg, d, [ts, exp), D) to S_O
    end
end
```

UNION

FILTER produces a single output streaming graph by appending all sgts from its input streaming graphs (Algorithm **UNION**).

Algorithm UNION:

```
input  : Streaming graphs S_1, S_2, output label d
output: Streaming graph S_O
S_O ← ∅
foreach t = (u, v, l, [ts, exp), D) ∈ S_1, S_2 do
    push (src, trg, d, [ts, exp), D) to S_O
end
```

5.4.2 Stateful Operators

Stateful operators need to maintain an internal operator state that is accessed during query processing. This section focuses on the stateful operators of SGA, i.e., PATTERN and PATH. These operators maintain an excerpt of their input streams that 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.

PATTERN

Subgraph pattern queries can be modeled as conjunctive queries, which is commonly evaluated using a series of non-blocking binary joins such as pipelined hash join [170, 64]. A binary join tree is constructed for a given PATTERN operator where leafs represent streaming graphs as input streams and internal nodes represent pipelined hash join operators. For instance, Figure 5.1 (right) shows the logical plan for the query in Example 1 and the join tree for its PATTERN. The ordering of predicates in PATTERN is used to construct the join tree.

Remember that the standard implementation of pipelined hash join is based on the *negative tuple* approach [170]: 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 insertion (expiration) matches [163, 66]. A straightforward adaptation of this standard implementation for time-based sliding windows represents each sgt by a pair of elements that are processed by the operator: a positive (+) element signaling sgt's insertion, and a negative (−) element signaling sgt's expiration (Remark 1). Based on the observation that expirations from a time-based sliding window follow a temporal pattern [67], it is possible to determine exactly when a resulting sgts expires and to eliminate the need for negative tuples. A resulting sgt is expired when one of its participating input sgts expire; consequently, the validity interval of a resulting sgt is the intersection of validity intervals of its participating sgts (see Definition 33 for the semantics of PATTERN). Algorithm **PATTERN** utilizes this temporal pattern of window movements to eliminate the use of negative tuples for signaling expirations. It maintains a priority queue based on the expiry timestamps of sgts as a secondary index to the internal operator state, i.e., elements of the priority queue are references to sgts in internal hash tables and priorities are expiry timestamps. As windows slide, expired sgts can be directly located and removed from the operator state without negative (−) elements to signal their expirations.

PATH

DD's *iterate* allows constructing cyclic dataflows that can model arbitrary nested iterations, and it can be used to evaluate PATH and its recursive path expressions (Section 5.3).

Algorithm PATTERN:

input : Streaming graphs S_1 and S_2,
predicate Φ,
output fields $src, trg \in \{src_1, trg_1, src_2, trg_2\}$,
output label d

output: Streaming graph S_O

$S_O \leftarrow \emptyset$
`Initialize` OS_1 `// operator state for the left input`
`Initialize` OS_2 `// operator state for the right input`
foreach $t = (u, v, l, [ts, exp), \mathcal{D}) \in \{S_1, S_2\}$ **do**
    $\Gamma \leftarrow \emptyset$ `// Set of matching tuples`
    if $t \in S_1$ **then**
        `// sgt` t `is from` S_1
        `Insert(`OS_1, t`) // insert the new sgt to` OS_1
        `Expiry(`OS_2, ts`) // remove expired sgts from` OS_2
        $\Gamma \leftarrow$ `Probe(`OS_2, t, Φ`) // retrieve matching tuples`
    end
    else
        `// sgt` t `is from` S_2
        `Insert(`OS_2, t`) // insert the new sgt to` OS_2
        `Expiry(`OS_1, ts`) // remove expired sgts from` OS_1
        $\Gamma \leftarrow$ `Probe(`OS_1, t, Φ`) // retrieve matching tuples`
    end
    foreach $t' = (u', v', l', [ts', exp'), \mathcal{D})' \in \Gamma$ **do**
        push $(src, trg, d, [ts, exp) \cup [ts', exp'), \mathcal{D} \circ \mathcal{D}')$ to S_O
    end
end

However, the use of recursion in SGQ – the class of queries targeted in this thesis – is limited to transitive closure (Section 3.3), and the RPQ-based semantics of PATH is sufficient to evaluate this limited form of recursion (Theorem 6). Consequently, the streaming RPQ evaluation algorithms presented in Chapter 4 can be used as a physical, non-blocking implementation for PATH. Remember that the Algorithm **RAPQ** incrementally performs a traversal of the underlying snapshot graph under the constraints of a given RPQ and maintains a compact representation of partial path segments in a spanning forest. Such a compact representation facilitates the recovery of actual paths and allows queries to return and manipulate paths as first-class citizens. Additionally, adopting a specialized streaming RPQ algorithm as the non-blocking, physical implementation of PATH operator eliminates the need for cycles in physical execution plans.

The notion of *update pattern awareness* [67] can be adapted for the physical implementation of PATH, similar to PATTERN, and the temporal pattern of expirations from time-based sliding windows can be used to simplify state maintenance. In a nutshell, Algorithm **S-PATH** utilizes the validity intervals of sgts to maintain a single entry for each intermediate path segment by finding the path with largest expiry timestamp, that is, the path segment that will expire furthest in the future. The tree index Δ and its spanning trees are augmented with a priority queue as a secondary index based on the expiry timestamps of paths segments. This enables finding expired path segments through look-ups on the secondary index without the need for Algorithm **ExpiryRAPQ**, simplifying the state maintenance for PATH. This is possible due to the separation of the implementation of sliding windows from operator semantics via an explicit WSCAN operator. The modified algorithm (**S-PATH**) is used as the physical implementation of the PATH operator (Appendix A describes it in detail). The following example illustrates how **S-PATH** utilizes the temporal pattern of window movements to simplify state maintenance for processing expirations.

Example 14. *Consider the SGQ of Example 1 whose SGA expression is given in Example 13, and excerpt of the input to $\mathcal{P}_{RLP}^{RL^+}$ (Figure 5.3(a)). Both approaches behave similarly until $t = 28$ as all vertex-state pairs in T_x have a single derivation at $t = 27$ (Figure 5.3(b)). Upon arrival of the sgt $(y, u, HI, [28, 37), \mathcal{D} = \{(y, HI, u)\})$ at $t = 28$, the negative tuple approach does not update T_x as $(u, 1)$ is already in T_x, whereas the direct approach updates the validity interval and the parent pointer of $(u, 1) \in T_x$ (Line 23 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 5.4(a) and 5.4(b) depict the corresponding spanning trees at $t = 30$ for the direct and the negative tuple approaches, respectively. Note that in Figure 5.4(a), 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*

Algorithm S-PATH:

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

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

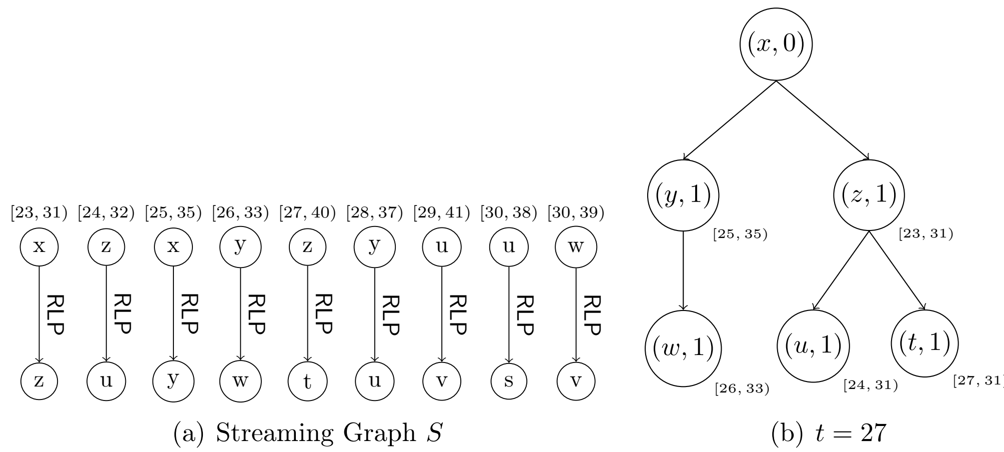
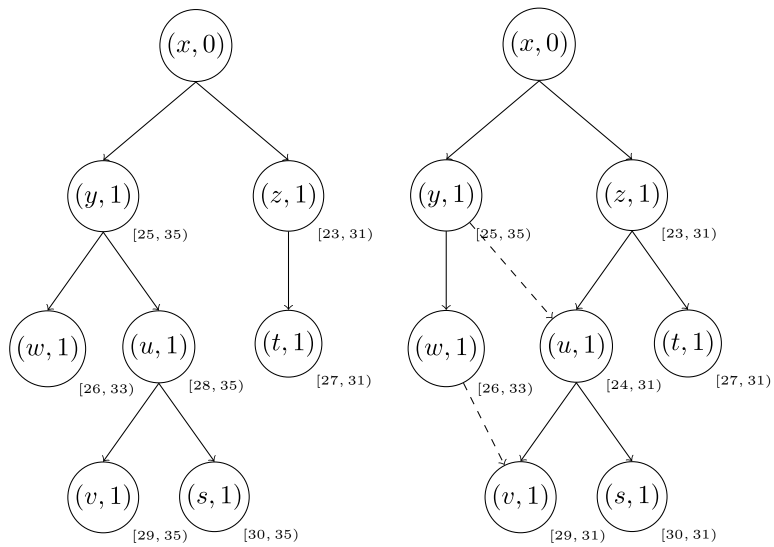


Figure 5.3: (a) A streaming graph S_{RLP} as the input for PATH operator, (b) spanning tree T_x at $t = 28$.

differs at $t = 31$ as multiple nodes expire. The original Algorithm ***ExpiryRAPQ*** *marks the entire subtree of $(z, 1)$ as potentially expired (Figure 5.4(b)), 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, Algorithm S-PATH can directly determine the expired nodes based on the validity intervals (nodes $(z, 1)$ and $(t, 1)$ as shown in Figure 5.4(a)) without additional processing.*

5.5 Experimental Analysis

The main objective of this section is to demonstrate the feasibility of implementing a performant system based on the algebraic framework proposed in this chapter. In the remainder, Section 5.5.1 describes the workloads and streaming graphs used for the experimental analysis, Section 5.5.2 provides an end-to-end performance analysis of the proposed algebraic approach for persistent evaluation of streaming graph queries using the prototype implementation described in Section 5.4. Finally, Section 5.5.3 assess the scalability by varying the window size ω and the slide interval β.



(a) Direct (b) NT

Figure 5.4: Spanning tree T_x at $t = 30$ following the (a) *direct* approach, (b) spanning (b) the *negative tuple* approach for PATH.

5.5.1 Methodology

Setup

Experiments are run on a Linux server with 32 physical cores and 256GB memory. For each query and configuration, 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 are reported.

Datasets

Stackoverflow (SO) and **LDBC SNB** (SNB) graphs are used for the experimental analysis; these are publicly available, large-scale graphs with labelled and timestamped edges on which persistent queries with complex graph patterns can be formulated. SO is a temporal graph of user interactions on the stackoverflow website containing 63M interactions (edges) of 2.2M users (vertices), spanning 8 years [135], and SNB is a synthetic social network graph that simulates the interactions of an online social network [53]. The update stream of the LDBC workload that contains 8 different types of interactions are extracted from its data generator, and replyOf, hasCreator and likes edges between users and posts, and knows edges between users are used. Unless specified otherwise, experiments on LDBC workload use the scale factor of 10 with 7.2M users and posts (vertices) and 40M user interactions (edges). SO contains only a single type of vertex and 3 different edge labels, and its cyclic nature causes a high number of intermediate results and resulting paths; so it is the most challenging one for the proposed algorithms. Finally, the window size ω is set to 1 month and the slide interval β is set to 1 day unless specified otherwise.

Workloads

A through literature search revealed that no current benchmark exists featuring RQ for graph DBMSs. The existing benchmarks are limited to UCRPQ thus not capturing the full expressivity of RQ even for static graphs. Streaming RDF benchmarks such as LSBench (https://code.google.com/archive/p/lsbench/) and Stream WatDiv [62] only focus on SPARQL v1.0 (thus not even including simple RPQs), and their workloads do not contain any recursive queries. Hence, the set stremaing graph queries used for the experiments are formulated from existing UCRPQ-based workloads: First, a set of graph patterns in the form of UCRPQ from existing benchmarks and studies [172, 32, 131, 53, 19] are collected, and a set of complex graph patterns are constructed from these UCRPQs by applying a Kleene star over each graph patterns. Table 5.1 lists the set of graph patterns of increasing expressivity (from RPQ to complex RQ with complex graph patterns) that are used to define streaming graph queries. $Q_1 - Q_4$ are commonly used RPQs in existing studies [172, 32, 131], and they are used to test SGA's PATH operator. Q_5 & Q_6 are CRPQ-based complex graph patterns based on SNB queries IS7 and IC7 [53]. For instance, Q_6 – IC7 of SNB – with edge labels knows, likes and hasCreator asks for recent likers of a person's messages that are also connected by a path of friends. Q_7 – Ex. 1 – is the most expressive RQ-based complex graph patterns used to demonstrate the abilities of the proposed SGA to unify subgraph pattern and path navigation queries in a structured manner and to treat paths as first-class citizens. It defines a path query over the complex graph pattern of Q_6; it finds arbitrary length paths where users are connected by the recentLiker pattern. Note that this query cannot be expressed in existing graph query languages such as Cypher and SPARQL due to the presence of recursion over a graph pattern (these UCRPQ-based languages limit recursion over edges). The final query workload from this set of complex graph patterns is instantiated by choosing appropriate predicates, i.e., edge labels, for each query edge from each dataset and by setting the duration of time-based sliding windows $\mathcal{W}_\omega$ as described above. Finally, the physical query execution plan for each query is constructed using its canonical SGA expression as previously described in Section 5.3.

Table 5.1: $Q_1 - Q_4$ correspond to common RPQ observed in real-world query logs [32], and $Q_5 - Q_7$ are Datalog encodings of RQ-based complex graph patterns that are used to define streaming graph queries. Q_5 and Q_6 correspond to complex graph patterns of LDBC SNB queries $IS7$ and $IC7$ [53], respectively, and Q_7 corresponds to the complex graph pattern given in Example 1 that is defined as a recursive path query over the graph pattern of Q_6. a, b and c correspond to edge predicates that are instantiated based on the dataset characteristics.

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	$RL(x, y) \leftarrow a^+(x, y),\ b(x, m),\ c(m, y)$ $Ans(x, m) \leftarrow RL^+(x, y), c(m, y)$

5.5.2 Query Processing Performance

Throughput & Tail Latency

Table 5.2 (SGA) shows the aggregated throughput and tail latency of the streaming graph query processor introduced in this chapter for all queries in Table 5.1. Streaming graph edges whose label is not in a given SGQ is discarded, and 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 and cyclic. The throughput ranges from hundreds of edges-per-second for the SO to hundreds of thousands of edges-per-second for SNB.

Comparative Analysis

Existing work on query processing over streaming data such as data stream management systems and streaming RDF systems cannot process queries in Table 5.1 as they focus on relational queries and SPARQL v1.0, respectively (Chapter 2). TD with its DD layer is the only general-purpose system that can be used to incrementally evaluate recursive computations that are modelled as cyclic dataflows. Consequently, the comparative analysis presented here considers two systems built on top of TD: DD and the SGA-based streaming

Table 5.2: (Tput) The throughput (edges/s) and (TL) the tail latency (s) of SGA and DD systems for queries in Table 5.1 on SO and SNB graphs with $\omega = 30$ days and $\beta = 1$ day.

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

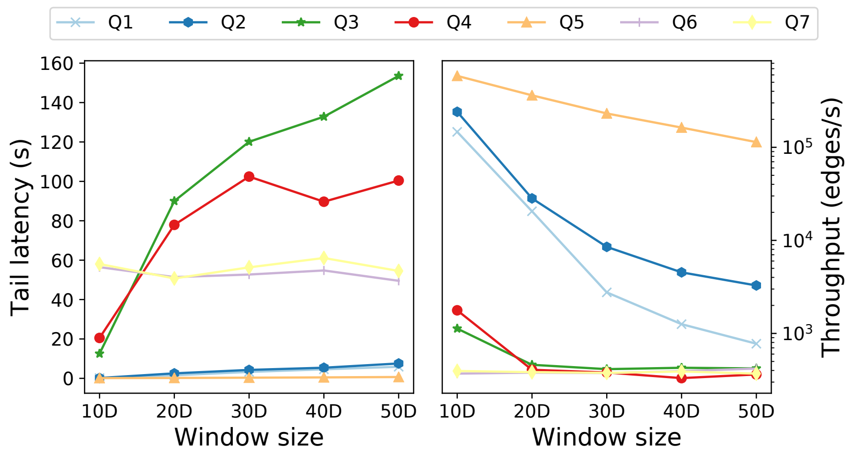


Figure 5.5: The performance of the SGQ processor prototype wrt window size ω on SO graph.

graph query processor described in Section 5.3; other comparisons would not be appropriate or fair without these supporting primitives. Table 5.2 (DD) reports the throughput and tail latency of DD dataflows for all queries in Table 5.1. Overall, the SGA-based query processor outperforms the DD baseline on SO and provides a competitive performance on the SNB dataset. On SNB, Q_6 & Q_7 do not have the Kleene-plus over a as it causes DD to timeout. Due to highly cyclic structure of SO, there are many alternative paths between each pair of vertices, and the streaming RPQ algorithm for PATH implementation (Chapter 4) maintains a compact representation of valid path segments and utilizes the temporal patterns of sliding window movements to simplify expirations (Section 5.4). DD-based query processor provides better performance on linear path queries Q_1–Q_4 on SNB, but not others. This is due to the tree-shaped structure of replyOf edges in SNB, where there is only one path between a pair of vertices, so PATH specific optimizations do not apply. Performance variations on SNB suggest optimization opportunities for recursive graph queries when selecting physical operator implementations, as in the case for streaming relational joins [67]. These results demonstrate the feasibility of the algebraic approach for evaluating SGQ that is introduced in this chapter.

5.5.3 Sensitivity Analysis

This section analyzes the impact of the window size ω and the slide interval β on end-to-end query performance of the proposed streaming graph query processor. SO graph is

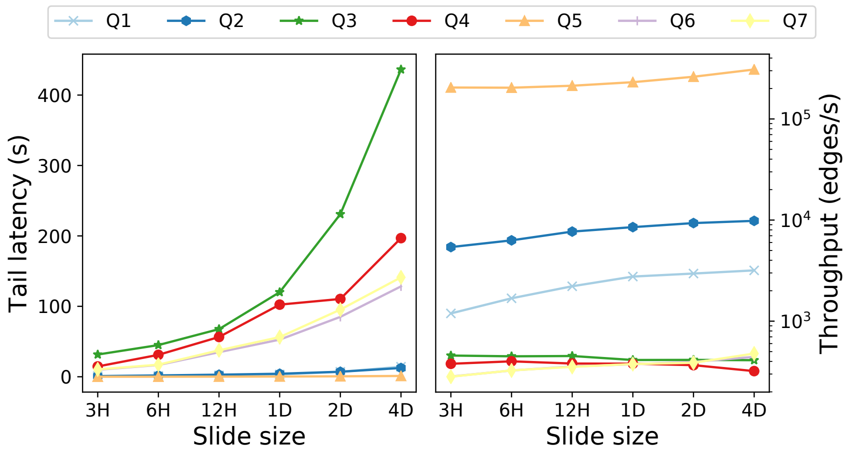


Figure 5.6: The performance of the SGQ processor prototype wrt slide interval β on SO graph.

used for this experiment as its dense and cyclic structure pose a challenge for physical implementations of the stateful operators PATTERN and PATH. Fig. 5.5 reports the aggregate throughput and the tail latency for each query across various window sizes ω. As expected, the throughput of all tested queries decreases with increasing ω, 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.

Analysis of the impact of the slide interval β on performance reveals a dissimilar behaviour. As previously mentioned, the slide interval β controls the time-granularity at which the sliding window progresses, and the prototype implementation introduced in this chapter uses β to control the input batch size. Figure 5.6 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 5.7. DD and its underlying indexing mechanism, i.e., shared arrangements [119], 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

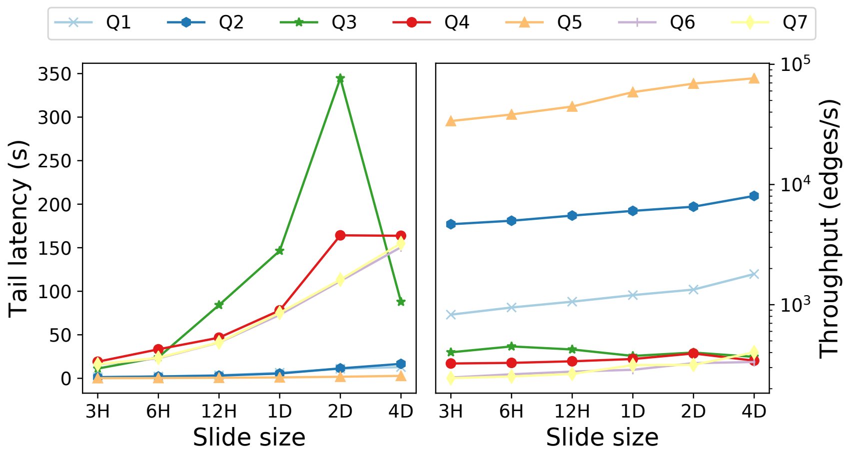


Figure 5.7: The tail latency of each window slide and the aggregate throughput of SGQ evaluation on DD with increasing slide interval β on SO graph.

identification of other optimization opportunities is a topic of future work.

5.6 Discussion

This chapter studies the evaluation of streaming graph queries and describes a prototype implementation of a streaming graph query processor. Its main contribution is the Streaming Graph Algebra that consists of a set of operators defined over streaming graphs, complementing the SGQ model (Chapter 3) as the foundational basis of the streaming graph query processing framework introduced in this thesis. SGA provides primitives for expressing query evaluation plans for SGQ as:

- its operators form temporal generalizations of their non-temporal counterparts through implicit treatment of sgts' validity intervals;
- time-based sliding windows and path navigations are specified via novel WSCAN and PATH operators, respectively; and
- SGA expressions return and manipulate paths as paths as treated as first-class citizens of the data model.

By defining the semantics of a set of operators that can be used as building blocks, SGA enables representation of query evaluation plans independent of system-specific details.

Section 5.2.1 proves that SGA has at least the same expressive power as SGA and provides a concrete algorithm for translating SGQs into their canonical SGA expressions. Clear separation of operator and query semantics from implementation details simplifies (i) the implementation of physical operators consistent with the semantics, and (ii) the design of a query processor that use these physical operators as building blocks. Sections 5.3 and 5.4 describe alternative physical implementations for SGA operators using the TD system as the underlying execution engine: one based on the general-purpose DD primitives and one based on SGQ-specific implementations, respectively.

An important result presented in this chapter is the closedness of SGA and the composability of SGA expressions over streaming graphs. Optimization of SGQs in a principled way relies on the systematic exploration of the space of equivalent plans, which is only possible by first establishing an algebraic representation amenable to rewrites through equivalence rules. Defined as the closure of a set of operators over the streaming graph data model, SGA provides such a representation for SGQ in which a query optimizer can reason about transformations and equivalences of query execution plans. The next chapter introduces a set of transformation rules that hold in SGA and describes the design of an SGA-based query optimizer for the systematic exploration of the rich plan space using these rules.

Chapter 6

Optimization of Streaming Graph Queries

6.1 Introduction

It is well-known that the performance of different evaluation plans for a query may be widely different: Figure 6.1 illustrates the performance variations of different but equivalent plans for a streaming graph query with a complex path pattern over two different streaming graphs [132] (generation of equivalent plans will be discussed momentarily in Section 6.2). Finding the right evaluation plan for a given query, known as *query optimization*, is a notoriously challenging problem. The separation of query semantics from the implementation details provides the necessary degree of freedom to explore the space of possible plans systematically, and query optimizers find an "efficient" execution plan for a given query from among a subset of possible execution plans. At a high level, query optimization can be defined as a search problem with three components [40]:

- a search space with a set of operators and a set of transformation rules that represents the set of equivalent query evaluation plans for a given query;
- a cost model that estimates a relative measure of the resource usages of query evaluation plans;
- and an enumeration algorithm for systematic exploration of the plan space.

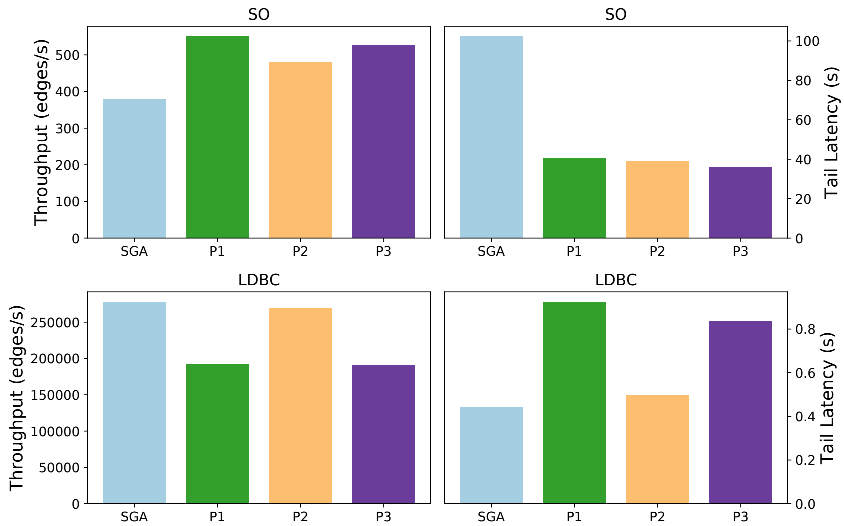


Figure 6.1: The throughput and tail latency of Q_4 (Table 5.1 in Chapter 5) on (top) SO and (bottom) SNB for equivalent SGA plans.

Query optimization is perhaps one of the most studied topics in database systems, and existing work predominantly focuses on relational queries in the snapshot model. Consequently, realizing the long-term vision envisioned in this thesis (Section 1.2.2) requires re-thinking this optimizer architecture in the context of streaming graph queries. First of all, the search space of an SGQ optimizer needs to incorporate plans with path navigation queries. SGA and its PATH operator provide the necessary tool to represent query evaluation plans for such queries. Next, rules with well-defined semantics representing equivalences between SGA expressions are needed. This enables the optimizer to explore the search space and to find equivalent plans through algebraic rewrites. Also, traditional cost models estimate the resource usage (e.g., execution time, network cost) required to complete the execution of a given query by using a set of statistics available about the underlying dataset. However, SGQs are continuously evaluated over potentially unbounded streaming graphs, requiring a fundamental change in cost metrics and statistics used by the cost model.

This chapter addresses the aforementioned challenges of SGQ optimization. It describes the design of a cost-based SGQ optimizer framework in the context of the streaming graph query processing framework proposed in this thesis. Building a full-fledged optimizer is an enormous undertaking, as demonstrated by the five decades of work on relational query optimizers. The objective here is to provide the foundational tools upon which SGQ optimizers can be developed. In line with this objective, this chapter first formally defines the search space for SGQ evaluation plans by introducing a set of transformation rules for SGA operators (Section 6.2). Some of the rules are streaming generalization of their non-streaming counterparts. A set of new rules for SGA's novel operators WSCAN and PATH is introduced, enabling the search space to incorporate plans for queries that return and manipulate paths. These rules define equivalences between SGA expressions (and corresponding query evaluation plans) and enable systematic exploration of the plan space described by SGA expressions. Then, an SGA-specific cost model is described for estimating the resource usage of SGA operators (Section 6.3). This cost model is based on the *unit-time* model, which was originally developed for relational joins over tuple streams [87], and characterizes the resource usage of continuous query operators and query plans per unit application time. Using SGA-specific operator formulas, it is shown that the output streaming graph characteristics and the resource usage of an operator can be estimated based on its input streaming graph characteristics. Given the estimations for its individual operators, the resource usage of an entire query plan is calculated by summing up the operators' resource usage. A prototype implementation of a Cascades-style cost-based SGQ optimizer is described (Section 6.4). This implementation is based on the Apache Calcite optimizer framework [24] and incorporates the search space and the cost model introduced in this chapter. Finally, an experimental study that demonstrates the feasibility of the cost-based optimization of SGQ using this prototype implementation is presented (Section 6.5).

6.2 Search Space

The search space for query optimization has two main components: (i) a set of operators defined over the underlying data model for representing query plans, and (ii) a set of transformation rules that are used to rewrite an algebraic expression into equivalent ones. As shown in Section 5.2.1, SGA and its operators provide the foundational basis to represent query evaluation plans for SGQ. This section describes a set of transformation rules holding in SGA in the form of algebraic equivalences. These rules formally describe the equivalences between query evaluation plans for SGQ (Definition 35) and enable query optimizers to explore the search space through query rewrites systematically.

Definition 35 (Plan Equivalence)**.** *Let P_1 and P_2 be query evaluation plans that consist of SGA operators for two SGA expressions over the same set of input streaming graphs, and let S_1 and S_2 be their output streaming graphs, respectively. P_1 and P_2 (and their corresponding SGA expressions) are said to be equivalent if and only if their output streaming graphs are* *equivalent (Definition 13). That is, P_1 and P_2 are equivalent iff snapshots of their output streaming graphs, $\tau_t(S_1)$ and $\tau_t(S_2)$ are equivalent at any given time $t \in \mathbf{T}$.*

$$P_1 \equiv P_2 \Longleftrightarrow S_1 \equiv S_2$$

6.2.1 Conventional Transformation Rules

Recall that UNION, FILTER and PATTERN operators are streaming generalizations of their relational counterparts and their semantics are formaly defined using the notion of *snapshot-reducibility* (Definition 15). Consequently, some of the traditional relational transformation techniques such as join ordering, and predicate pushdown are applicable in SGA. The set of algebraic transformation rules that are derived from their relational counterparts are as follows:

1. Commutativity of UNION: $S_a \cup^{[d]} S_b \equiv S_b \cup^{[d]} S_a$
2. Idempotentancy of FILTER: $\sigma_\Phi\big(\sigma_\Phi(S)\big) \equiv \sigma_\Phi(S)$
3. Conjunctive FILTER predicates: $\sigma_{\Phi_2}\big(\sigma_{\Phi_1}(S)\big) \equiv \sigma_{\Phi_1 \wedge \Phi_2}(S)$
4. Disjunctive FILTER predicates: $\sigma_{\Phi_2}(S) \cup \sigma_{\Phi_1}(S) \equiv \sigma_{\Phi_1 \vee \Phi_2}(S)$
5. Associativity of FILTER: $\sigma_{\Phi_2}\big(\sigma_{\Phi_1}(S)\big) \equiv \sigma_{\Phi_1}\big(\sigma_{\Phi_2}(S)\big)$
6. Distributivity of FILTER over UNION: $\sigma_\Phi\big(S_a \cup^{[d]} S_b\big) \equiv \sigma_\Phi(S_a) \cup^{[d]} \sigma_\Phi(S_b)$
7. Commutativity of PATTERN: $\bowtie_{trg_1=src_2}^{src1,trg2,d} (S_a, S_b) \equiv \bowtie_{trg_2=src_1}^{src2,trg1,d} (S_b, S_a)$
8. Associativity of PATTERN:
   $\bowtie_{trg_1 \equiv src_2}^{src1,trg2,d} \big(S_a, \bowtie_{trg_1=src_2}^{src1,trg2,d_1} (S_b, S_c)\big) \equiv \bowtie_{trg_1=src_2}^{src1,trg2,d} \big(\bowtie_{trg_1=src_2}^{src1,trg2,d_1} (S_a, S_b), S_c\big)$
9. FILTER pushdown through PATTERN (left):
   $\sigma_\Phi\big(\bowtie_{trg_1=src_2}^{src1,trg2,d} (S_a, S_b)\big) \equiv \bowtie_{trg_1=src_2}^{src1,trg2,d} \big(\sigma_\Phi(S_a), S_b\big), if attr(\Phi) \in \{src1, trg1, l_a\}$
10. FILTER pushdown through PATTERN (right):
    $\sigma_\Phi\big(\bowtie_{trg_1=src_2}^{src1,trg2,d} (S_a, S_b)\big) \equiv \bowtie_{trg_1=src_2}^{src1,trg2,d} \big(S_a, \sigma_\Phi(S_b)\big), if attr(\Phi) \in \{src2, trg2, l_b\}$

Lemma 2. *Conventional transformation rules are applicable in SGA over expressions involving UNION, FILTER and PATTERN operators.*

Proof. The correctness of the above transformation rules over SGA directly follows from the *snapshot reducibility* of UNION, FILTER and PATTERN to their relational counterparts and the equivalence of streaming graphs (Definition 13). Let P_1 be an SGA expression for a streaming graph query Q, and P_2 be the optimized SGA expressions after applying a conventional transformation rule. To prove the equivalence of P_1 and P_2 (hence the correctness of conventional transformation rules), it is sufficient to show that output streaming graphs S_1 and S_2 are equivalent. Due to snapshot reducibility, at any point in time t, snapshot graphs $\tau_t(S_1)$ and $\tau_t(S_1)$ are equivalent to the outputs of one-time counterparts of the original and optimized plans P_1 and P_2, respectively. The correctness of the conventional rules over one-time relational operators implies that snapshot graphs $\tau_t(S_1)$ and $\tau_t(S_2)$ are equivalent at any given time t, showing that output streaming graphs S_1 and S_2 are equivalent. Consequently, the original and optimized SGA expressions are equivalent, concluding the proof. □

6.2.2 Transformation Rules for WSCAN

WSCAN ($\mathcal{W}_\omega$) commutes with operators that do not alter the validity intervals of sgts, i.e., UNION and FILTER. Formally:

12. FILTER pushdown through WSCAN: $\mathcal{W}_\omega(\sigma_\phi(S)) \equiv \sigma_\phi(\mathcal{W}_\omega(S))$
13. Distributivity of WSCAN over UNION: $\mathcal{W}_\omega(S_1 \cup^{[d]} S_2) \equiv \mathcal{W}_\omega(S_1) \cup^{[d]} \mathcal{W}_\omega(S_2)$

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. The correctness of transformation rules for WSCAN directly follows from the definitions of stateless SGA operators UNION and FILTER (Section 5.2.1). WSCAN commutes with these stateless operators as UNION and FILTER operate only on the explicit attributes of sgts, i.e., they do not manipulate the validity intervals of tuples, whereas WSCAN only operate on the implicit attributes of sgts.

6.2.3 Transformation Rules for PATH

Remember that the semantics of PATH is based on the RPQ model where path expressions are specified as regular expressions over the alphabet of edge labels. A complex regular expression R can be decomposed into its fragments where each fragment is a sub-expression of R. This decomposition of R is used to transform an SGA expressions involving PATH into its equivalent expressions. Formally:

14. Decomposition of concatenation: $\mathcal{P}^d_{r_1 \cdot r_2}(S) \equiv \bowtie^{src1,trg2,d}_{trg_1=src_2} \left(\mathcal{P}^{d_1}_{r_1}(S), \mathcal{P}^{d_2}_{r_2}(S)\right)$
15. Decomposition of alternation: $\mathcal{P}^d_{r_1|r_2}(S) \equiv \mathcal{P}^d_{r_1}(S) \cup \mathcal{P}^d_{r_2}(S)$
16. Substitution of transition: $\mathcal{P}^d_a(S) \equiv S_a$, if $a \in \Sigma$
17. Decomposition of Kleene star: $\mathcal{P}^d_{r^*}(S) \equiv \mathcal{P}^d_{d_1^*}\left(\mathcal{P}^{d_1}_r(S)\right)$

The correctness of these rules follows from the semantics of PATH (Definition 34) and the structure of regular expressions (Definition 18). In the following, the proof for decomposition of concatenation rule is provided; the proofs for other rules are quite similar and straightforward.

Lemma 3 (Decomposition of concatenation)**.** *Transformation rule 14 correctly decomposes a PATH operator with concatenation, that is, the output streaming graphs of the original and optimized expressions over the same set of input streaming graphs are equivalent.*

Proof. Definition 35 states that two SGA expressions are equivalent iff their output streaming graphs are equivalent. Consequently, the proof proceeds by showing the equivalence of output streaming graphs of the original expression $P_1 = \mathcal{P}^d_{r_1 \cdot r_2}(S)$ and the optimized expression $P_2 = \bowtie^{src1,trg2,d}_{trg_1=src_2} \left(\mathcal{P}^{d_1}_{r_1}(S), \mathcal{P}^{d_2}_{r_2}(S)\right)$. An sgt $t = (u, v, d, [ts, exp), \mathcal{D})$ is in the output streaming graph of the original SGA expression P_1 if and only if there exists a length-2 path $\xrightarrow{p}: \langle t_1, t_2 \rangle$ between vertices u and v s.t.:

1. $t_1 = (u, x, r_1, [ts_1, exp_1), \mathcal{D}_\infty) \in S$,
2. $t_2 = (x, v, r_2, [ts_2, exp_2), \mathcal{D}_\in) \in S$,
3. and $[ts, exp) = [ts_1, exp_1) \cap [ts_2, exp_2)$.

For any such pair of sgts $t_1, t_2 \in S$, the optimized SGA expression produces the same sgt $t = (u, v, d, [ts, exp), \mathcal{D})$ as:

1. there exists $t'_1 = (u, x, d_1, [ts_1, exp_1), \mathcal{D}) \in \mathcal{P}^{d_1}_{r_1}(S)$,
2. there exists $t'_2 = (x, v, d_2, [ts_2, exp_2), \mathcal{D}) \in \mathcal{P}^{d_2}_{r_2}(S)$,

3. $\Phi(t'_1, t'_2) =$ `true` and $[ts, exp) = [ts_1, exp_1) \cap [ts_2, exp_2)$

Following the definition of PATTERN, it is easy to see that no other pair of sgts in S could participate in a resulting sgt for P_2. Consequently, the output streaming graphs of the original and optimized SGA expressions are the same, which concludes the proof. □

These transformation rules enable the exploration of a rich plan space for SGQ that is represented by SGA. Remember that except for the windowing operator WSCAN, SGA operators are *snapshot-reducible* to their one-time, non-temporal counterparts. Consequently, these transformation rules (except the transformation rules for WSCAN presented in Section 6.2.2) are applicable one-time queries over static graphs. In particular, PATH and its transformation rules enable the integration of existing approaches for RPQ evaluation with standard optimization techniques such as join ordering and pushing down selection in a principled manner. Traditionally, path query evaluation follows two main approaches: graph traversals guided by finite automata or relational algebra extended with transitive closure, i.e., *alpha*-RA [131, 94, 144, 55, 18, 172]. Yakovets et al. introduce a hybrid approach (Waveguide) and model the cost factors that impact the efficiency of RPQ evaluation on static graphs [172]. SGA enables the representation of these approaches in a uniform manner, and the above transformation rules enable the exploration of the plan space that subsumes these existing plans. Section 6.5.3 demonstrates the use of these transformation rules and illustrates the potential benefits of exploring the rich plan space offered by SGA.

6.3 Cost Model

Cost models enable query optimizers to make resource usage predictions based on system characteristics and to choose the "right" evaluation plan from the set of possible plans. Traditional cost models rely on intermediate result cardinalities for estimating the execution time of a given query plan. It is well-documented that traditional, cardinality-based cost models are not applicable in the streaming model [167, 68] as (i) the notion of "execution time" is ill-suited for long-running, persistent queries, and (ii) unbounded streams make it infeasible or even impossible to estimate cardinalities. Consequently, a novel cost model to estimate the resource usage SGQ evaluation plans is needed. This section first describes the streaming graph characteristics used to model the inputs and outputs of SGA operators. Then, operator formulas that are used to estimate these characteristics of the output streaming graph of an SGA operator based on its input streaming graphs are provided. Based on these characteristics, the resource usage of physical implementations of SGA operators and their corresponding cost formulas are described.

6.3.1 Streaming Graph Characteristics

A streaming graph S is modelled by the following parameters:

- r – the arrival rate of a streaming graph, which is determined by the average time between the start timestamps of consecutive sgts
- v – the average length of validity intervals in a streaming graph

The arrival rate r corresponds to the number of sgts processed per unit application time and impacts the resource usage of every SGA operator. The average interval length v, on the other hand, represents the duration for which an sgt will be valid. Consequently, v controls the duration that an incoming sgt could participate in future results for stateful SGA operators PATTERN and PATH. These two together determine the size of the data structures that are used to maintain the internal operator state. As in the rest of this thesis (Section 3.2), the cost model uses the event (application) time, and both parameters are computed per application time. Consequently, the output streaming graph of an operator (query plan) solely depends on its input streaming graphs, and it is not impacted by the physical properties of the underlying execution engine, such as the physical algorithm implementations, scheduling decisions, or the parallelization model. In that sense, these parameters represent the logical properties of streaming graphs that are relevant to the cost model, and the estimation of these parameters is analogous to *cardinality estimation* in traditional cost models of RDBMSs.

In the following, formulas for estimating the output streaming graph characteristics of SGA operators are provided. Table 6.1 lists the terms used in operator formulas.

Operator Formula 1 (WSCAN)**.** *Given a WSCAN operator with window size ω, the parameters of S_O can be estimated as:*

$$r_O = r_i \tag{6.1}$$

$$v_O = \omega \tag{6.2}$$

WSCAN adjusts the interval length of each sgt by setting expiration timestamp to $ts+\omega$ (Definition 30), directly controlling the average interval length of S_O.

Table 6.1: Summary of terms and definitions used in estimation this chapter.

S_i	the i^{th} input streaming graph
r_i	rate of the i^{th} input streaming graph
v_i	average interval length of the i^{th} input streaming graph
Φ	Boolean operator predicate
f	operator selectivity
S_O	the output streaming graph
r_O	estimated rate of the operator output
v_O	estimated average interval length of the operator output
C_t	the average cost of evaluating a single sgt
C_O	the average cost of pushing a single sgt through a streaming graph
U_d	the update cost of a single entry in a data structure `d`
P_d	the access cost of a single entry in a data structure `d`

Operator Formula 2 (FILTER)**.** *Given a FILTER operator with a predicate Φ, let f be the selectivity of Φ, i.e., the ratio of sgts that satisfy the predicate Φ in the input streaming graph S_i. The parameters of S_O can be estimated as:*

$$r_O = f \cdot r_i \tag{6.3}$$

$$v_O = v_i \tag{6.4}$$

Output rate of FILTER is reduced by the selectivity f as the probability of an sgt satisfying the filter predicate Φ is f. As the filter operator does not change the validity interval of sgts, the validity interval (and the window size) of the output stream remains the same.

Operator Formula 3 (UNION)**.** *Given a UNION operator with n input $S_1, \ldots, S_n$, the parameters of S_O can be estimated as:*

$$r_O = \sum_{i=1}^{n} r_i \tag{6.5}$$

$$v_O = \frac{\sum_{i=1}^{n} v_i \cdot r_i}{r_O} \tag{6.6}$$

Output rate of UNION is simply the sum of the rates of all of its input streaming graphs. Average length of validity intervals, on the other hand, is computed as the weighted average length of its inputs. As r_i determines the number of sgts with validity interval v_i, the average validity interval length is computed by the sum of weighted validity intervals divided by the the output rate (the sum of weights).

Operator Formula 4 (PATTERN)**.** *Given PATTERN operator over two input streaming graphs S_1, S_2 where the selectivity of the predicate Φ is f, the parameters of S_O can be estimated as:*

$$r_O = f \cdot (v_2 \cdot r_2) \cdot r_1 + f \cdot (v_1 \cdot r_1) \cdot r_2 \tag{6.7}$$

$$v_O = \frac{v_1 \cdot v_2}{v_1 + v_2} \tag{6.8}$$

The first part of the Equation 6.7 corresponds to the output rate due to arrivals for the input streaming graph S_1 while the second part does so for S_2. $(v_2 \cdot r_2)$ is the average number of sgts from S_2 that might have temporal overlap with arrivals from S_1. Consequently, $f \cdot (v_2 \cdot r_2) \cdot r_1$ is the average number of resulting sgts due to arrivals from S_1 during a given time instant, i.e., the output rate due to arrivals from S_1. Similarly, the second part of the equation computes the output rate due to arrivals from S_2.

The validity intervals of sgts in the output streaming graph of PATTERN is the intersection of the validity intervals of participating sgts. Consequently, v_O corresponds to the average overlap of validity intervals from S_1 and S_2. Without loss of generality, let $v_1 \geq v2$. By shifting an interval of length v_2 over an interval of length v_1, the expected length of an overlap can be calculated. In brief, there are $v_1 + v_2 - 1$ combinations for two intervals of length v_1 and v_2: $v_1 - v_2 + 1$ complete overlaps of size v_2, two overlaps of size $v_2 - 1$, two overlaps of size $v_2 - 2$ and it continues in this pattern. Under the uniformity assumption (i.e., the distribution of the length of validity intervals of a streaming graph is uniform), the expected overlap length is $(v_2) \cdot (v_1 - v_2 + 1) + 2 \cdot \sum_{i=1}^{v_2 - 1} i = \frac{v_1 \cdot v_2}{v_1 + v_2 - 1}$.

Note that these calculations depend solely on the semantics of the PATTERN and do not make assumptions about particular physical implementations. Section 6.3.2 investigates the impact of particular physical implementations on the resource usage.

Remark 5 (Complex PATH expressions)**.** *PATH is an operator with an arbitrary arity that might feature a complex path expression that consists of multiple operations (e.g., alternation, concatenation, and Kleene star). To estimate the characteristics of the output streaming graph of PATH with a complex path expression R, R is first decomposed into its fragments using its parse tree. The parse tree of a regular expression R is a tree where leafs are symbols (edge labels) from the alphabet and the nodes are primitive regular expressions, i.e., regular expressions with a single concatenation, alternation or Kleene star operation (Figure 6.2 illustrates the parse tree of the regular expression $R = \left((a|(b \cdot c))^* \cdot d\right)^*$). Then,* *the characteristics of the final expression can be estimated using the following formulas for individual nodes in a bottom-up manner.*

Operator Formula 5 (PATH with concatenation)**.** *Given a path expression $R = a \cdot b$ over streaming graphs S_a and S_b, the parameters of S_O can be estimated as:*

$$r_O = f \cdot (v_b \cdot r_b) \cdot r_a + f \cdot (v_a \cdot r_a) \cdot r_b \tag{6.9}$$

$$v_O = \frac{v_a \cdot v_b}{v_a + v_b} \tag{6.10}$$

Remember that PATH with a path expression $R = a \cdot b$ over streaming graphs S_a and S_b corresponds to a PATTERN with a subgraph pattern of sgts over labels a and b in the form of a linear path (Section 6.2). Consequently, the operator formulas of PATTERN can be used for a PATH operator over streaming graphs $\{S_a, S_b\}$ with a path expression $R = a \cdot b$. f corresponds to the selectivity of the linear path pattern $a \cdot b$ (similar to the selectivity of $\text{PATTERN}^{src1,trg2,d}_{trg1=src2}(S_a, S_b)$).

Operator Formula 6 (PATH with alternation)**.** *Given a $R = a \mid b$ over streaming graphs $\{S_a, S_b\}$, the parameters of S_O can be estimated as:*

$$r_O = r_a + r_b \tag{6.11}$$

$$v_O = \frac{r_a \cdot v_a + r_b \cdot v_b}{r_O} \tag{6.12}$$

Similarly, PATH with path expression $R = a \mid b$ over streaming graphs $\{S_a, S_b\}$ is equivalent to union of S_a and S_b and therefore the formulas for UNION can be for PATH with alternation.

PATH operator can be used to query arbitrary-length paths by using a path expression with a Kleene star, which corresponds to the traversal of the given expression *zero or more* times. Consequently, cost formulas for PATH with a Kleene star depends on the maximum path length, i.e., the number of iterations. Here, it is assumed that the maximum path length for PATH with a path expression R^* is given.[1]

Operator Formula 7 (PATH with Kleene star)**.** *Given the maximum path length m, the path expression R^* can be written using alternation and concatenation as follows:*

$$R^* = R^0 \mid R^1 \mid \cdots \mid R^m \text{ where } R^0 =, R^i = R^{i-1} \cdot R \tag{6.13}$$

[1]Existing graph query languages such as Cypher and SPARQL v1.1 explicitly support the bounded Kleene operator to specify lower and upper bounds for lengths of path expressions. Also, the maximum path length for path expression R^* can be bounded by the *diameter* of the graph induced by R relation on any graph.

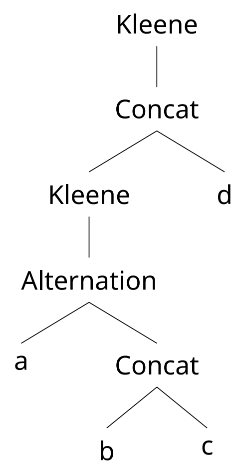


Figure 6.2: The parse tree obtained by decomposition of the complex path expression $\big((a|(b \cdot c))^* \cdot d\big)^*$.

Then, the rate and the average interval length for each path segment of length i, i.e., R^i, can be estimated using the concatenation *formula as follows:*

$$r_{R^i} = i \cdot r \cdot (frv)^{i-1} \tag{6.14}$$

$$v_{R^i} = \frac{v}{i} \tag{6.15}$$

Finally, given a R^* over a streaming graph S_R and the maximum path length m, the parameters of S_O can be estimated as:

$$r_O = \sum_{i=1}^{m} r_{R^i} \tag{6.16}$$

$$v_O = \frac{\sum_{i=1}^{m} v_{R^i} \cdot r_{R^i}}{r_O} \tag{6.17}$$

The following example demonstrates how a complex path expression with multiple operations can be decomposed into its simple fragments.

Example 15 (Complex path expression)**.** *Consider the path expression $R = \big((a|(b{\cdot}c))^*{\cdot}d\big)^*$. Its parse tree is shown in Figure 6.2. Given a PATH over streaming graphs $\{S_a, S_b, S_c, S_d\}$, characteristics of S_O can be computed by applying above formulas in a bottom-up manner:*

1. *Formula 5 over* b *and* c *to obtain* $(b \cdot c)$
2. *Formula 6 over* a *and* $(b \cdot c)$ *to obtain* $(a|(b \cdot c))$
3. *Formula 7 over* $(a|(b \cdot c))$ *to obtain* $(a|(b \cdot c))^*$
4. *Formula 5 over* $(a|(b \cdot c))^*$ *and* d *to obtain* $\big((a|(b \cdot c))^* \cdot d\big)$
5. *Formula 7 over* $\big((a|(b \cdot c))^* \cdot d\big)$ *to obtain* $\big((a|(b \cdot c))^* \cdot d\big)^*$

6.3.2 Operator Cost Formulas

The previous section describes the estimation of streaming graph characteristics for SGA operators. This is analogous to the cardinality estimation of intermediate results in traditional cost models of RDBMSs, as the output streaming graph characteristics of an operator solely depend on its input streaming graphs and the operator semantics, not particular physical implementations. Based on these streaming graph characteristics, this section describes how to estimate the resource consumption of SGQ query evaluation plans. In addition to the streaming graph characteristics described in the previous section (the rate r and the average length of validity intervals v), the following constants are used the cost model:

- C_t – the average cost of evaluating a single sgt by the particular physical operator implementation
- C_O – the average cost of pushing a single sgt through output streaming graph of an operator

In the following, cost functions that estimate the processing costs of physical implementations of SGA operators are provided. These cost functions are based on the physical implementations presented in Section 5.4, and they model the dominant factors of corresponding algorithms in terms of operations performed over the attributes of sgts. They estimate the processing cost of an operator per unit application time at a steady state, i.e., after the initialization of operators according to sliding window definitions.

Operator Formula 8 (WSCAN)**.** *WSCAN's processing cost per unit-time is calculated as:*

$$C_t = r_O \cdot C_O \tag{6.18}$$

Operator Formula 9 (FILTER)**.** *FILTER's processing cost per unit-time is calculated as:*

$$C_t = r_i \cdot C_\sigma + r_O \cdot C_O \tag{6.19}$$

where C_σ is the cost of evaluating the filter predicate on a single sgt; and, r_i and r_O represents the rate of input and output streaming graphs, respectively. FILTER processes every incoming sgts and append the ones that satisfy the selection predicate.

Operator Formula 10 (UNION)**.** *UNION's processing cost per unit-time is calculated as:*

$$C_t = r_O \cdot C_O \tag{6.20}$$

where r_O represents the output rate of UNION.

The cost formulas for stateless operators solely depend on per-tuple processing costs as each incoming sgt is processed on the fly. PATTERN and PATH, on the other hand, are stateful operators that maintain internal operator states, and the cost formulas for these operators include the update and look-up operations over their corresponding data structures.

Remark 6 (PATTERN Operator State)**.** *The operator state of PATTERN consists of all sgts from one input that might have overlapping validity intervals with any future sgts from the other input. Hence, each sgt from one input needs to be kept as long as its validity interval can overlap with sgts from the other input. The physical implementation of PATTERN (Section 5.4.2) is based on the well-known symmetric hash join algorithm: each incoming sgts is processed by (i) inserting the sgt into the operator state, (ii) performing lookups to find matching tuples, and (iii) removing expired sgts from the operator state. The internal operator state is maintained as a hash table. Upon the arrival of an sgt, it is inserted into its corresponding hash table, and the other table is probed for matches. However, determining expired sgts might require a complete scan of the hash table for the worst case. The physical implementation of PATTERN described in Section 5.4.2 maintains a secondary index to efficiently identify expired sgts. References to the sgts in the primary index are maintained in a priority queue organized in accordance with their expiration timestamps, allowing the algorithm to locate the expired sgts directly.*

Operator Formula 11. *PATTERN's processing cost per unit-time is calculated as:*

$$C_t = r_1 \cdot (I_1 + P_2 + E_2) + r_2 \cdot (I_2 + P_1 + E_1) + r_O \cdot C_O \tag{6.21}$$

where I_1, P_1, and E_1 corresponds to the insertion, processing and expiration cost of a single sgt from the first input of PATTERN and their detailed formulas are given below. Due to the symmetric nature of the physical implementation of PATTERN (Section 5.4.2), these costs regarding the second input can be calculated in a similar manner.

$$I_1 = U_h + U_{pq} \cdot \log(r_1 \cdot v_1) \tag{6.22}$$

$$P_1 = P_h \tag{6.23}$$

$$E_1 = U_h + U_{pq} \cdot \log(r_1 \cdot v_1) \tag{6.24}$$

In brief, the insertion cost I_1 is a combination of inserting an entry into the hash table and the priority-queue-based secondary index for S_1, which is logarithmic in the number of sgts maintained for S_1. Similarly, processing expirations involves probing the secondary index to retrieve expired sgts and removing those from the hash index. Consequently, E_1 is calculated as a combination of updating the hash index and the priority queue-based secondary index. Finally, P_1 is the cost of performing a lookup over the corresponding hash table, which is expected to be constant on average in a typical in-memory hash table implementation.

Remark 7 (PATH Operator State)**.** *The physical implementation of PATH is based on the the streaming RPQ algorithm S-PATH (Section 5.4). The operator state of PATH is maintained as the specialized data structure: Δ-tree index, which encodes partial path segments for the path expression R in the form of spanning trees where each spanning tree T_x consists of all nodes that are reachable by the vertex x through a path whose label conforms to R. Each incoming sgt is processed by (i) inserting new path segments into the Δ-tree index due to the incoming sgt, and (ii) removing invalid path segments from the Δ-tree index due to expired sgts. Similar to PATTERN, the Δ-tree index and its spanning trees are organized using a hash table that is backed by a priority queue-based secondary index to locate the expired sgts efficiently.*

As described Remark 7, the physical implementation of PATH is based on the streaming RPQ algorithm S-PATH. Chapter 4 provides a detailed amortized complexity analysis for both insertion and expiration operations over the Δ-tree index, which the cost formula for PATH is based on.

Operator Formula 12 (PATH)**.** *Given a PATH with a regular expression R as its path conditions over streaming graphs $\{S_1, \cdots, S_k\}$, its processing cost per unit-time is calculated as:*

$$C_t = r_i \cdot (I + E) + r_O \cdot C_O \tag{6.25}$$

$$\tag{6.26}$$

where r_i is the rate of the union of the input streaming graphs $\{S_1, \cdots, S_k\}$, and I and E correspond to the insertion and expiration cost, respectively, of a single sgt. Their detailed formulas based on the complexity analysis presented in Chapter 4 are given below. The cost formulas rely on the number of vertices in the snapshot graph of the input streaming graphs and the number of states in the minimal DFA for the path expression R, which are denoted by n and k, respectively.

$$I = P_h \cdot n \cdot k^2 + U_{pq} \cdot n \cdot k \cdot \log(n \cdot k) + U_h \cdot k \tag{6.27}$$
$$E = P_h \cdot n^2 \cdot k + P_{pq} \cdot n \cdot \log n + U_{pq} \cdot n \cdot k \cdot \log(n \cdot k) + U_h \cdot k \tag{6.28}$$

The insertion cost I can be calculated as a combination of (i) the cost of performing look-ups on the Δ-tree index to check existence of nodes, and (ii) the cost updating the Δ-tree index for new path segments. Similarly, the expiration cost E is a combination (i) the cost of priority queue look-ups to determine potentially expired spanning trees nodes, and (ii) the cost of updating the the spanning trees due to expirations.

Finally, based on these operator formulas for calculating the resource usage of individual SGA operators, the resource usage of a given execution plan is computed by summing up the individual costs of all operators in the plan. This simplifies the cost model and enables the cost model to provide resource usage estimations independent of low-level system issues such as scheduling and parallelism. Consequently, the planning decisions guided by the resource usage estimations described here are not affected by the internals of the underlying execution engine. They solely depend on: (i) input streaming graph characteristics, (ii) the query semantics, and (iii) particular physical implementations of SGA operators.

6.4 Prototype Implementation

As described previously, the main objective of this chapter is to lay out the fundamental primitives for SGQ optimizers, and the previous sections formally define the *search space* for SGA-based query evaluation plans and an SGA-specific *cost model* for SGQ. The final component of query optimization is the *enumeration algorithm* that finds an "efficient" plan for a given query from the space of equivalent plans. Plan space enumeration has been extensively studied, which has led to several *extensible* optimizer frameworks. Extensible optimizer frameworks employ rule engines that allow the addition of new transformation rules and operator definitions to extend the search space and use generalized cost functions that allow changes in the cost estimation. Existing optimizer frameworks can be broadly categorized based on their enumeration algorithms: (i) dynamic programming-based bottom-up techniques such as Starburst [147, 82] and (ii) memoization based top-down techniques such as Volcano and Cascades [72, 73]. This section describes a prototype implementation of a cost-based SGQ optimizer with a top-down, Cascades-style enumeration algorithm. In the remainder, Section 6.4.1 describes how to integrate these SGA-specific primitives into Apache Calcite, a state-of-the-art Cascades-style optimizer framework and Section 6.4.2 discusses the limitations of this prototype implementation by analyzing the underlying assumptions.

6.4.1 Apache Calcite Integration

Apache Calcite is a modular, extensible query processing framework for heterogeneous data sources [24]. At its core, Calcite's cost-based optimizer employs a top-down enumeration algorithm with extensible operator algebra, equivalence rules, and cost formulas. Similar to Cascades, plan enumeration in Calcite is done in a single step using two types of rules: *transformation rules* that define logical equivalences and *implementation rules* that map logical operators to their physical implementations. Calcite uses *trait*s to enforce physical properties associated with operators such as sort order and partitioning key. An important feature of Calcite is the *calling convention trait* that represents the execution backend where the query plan is executed. It enables Calcite to choose appropriate physical operator implementations for a given logical expression. Given a logical algebra expression, *trait*s, and the optimizer uses the cost model to find the final, optimized query evaluation plan. Extending Calcite for SGQ optimization requires:

- integrating SGA operators and their algebraic equivalences to define the search space for SGQ evaluation plans, and
- incorporating SGA-specific statistics and cost formulas to estimate resource usage of query plans that consist of SGA operators.

Calcite data adapters are used to define data access for different sources. The prototype SGQ optimizer incorporates the streaming graph data model (Section 3.2) by defining the structure (format) of streaming graph tuples using Calcite primitive data types. Then, SGA operators defined in Section 5.2.1 are implemented as logical operators with streaming graphs as input and outputs. These operators are not associated with a *calling convention* as they are used to form logical query evaluation plans that are independent of a particular execution backend. The canonical SGA expressions generated from SGQs (Section 5.2.2) are formed using these logical operators. Algebraic equivalences given in Section 6.2 are provided as *transformation* rules over these logical operators. Finally, following the Calcite terminology, `TD` calling convention is created to represent TD as the target execution engine (Section 5.3), and the physical operator implementations described in Section 5.4 are defined using the `TD` calling convention. An *implementation* rule for each SGA operator is used to map SGA operators to their corresponding physical implementations.

Calcite provides interfaces to plug custom metadata information into the optimizer. In SGA, the inputs and outputs of operators and query evaluation plans are *streaming graphs*. Consequently, the metadata catalogue is extended with *rate* and *average interval length* to account for streaming graph characteristics. The prototype SGQ optimizer implements operator formulas given in Section 6.3.1 to estimate the characteristics of intermediate and output streaming graphs. These estimations, in turn, are used in operator cost formulas (Section 6.3.2) to estimate the resource usage of physical implementations of SGA operators. Finally, the prototype SGQ optimizer calculates the cost of an evaluation plan as the cumulative sum of that of all of its operators.

6.4.2 Underlying Assumptions

The prototype SGQ optimizer models streaming graphs using r and v to determine the time between consecutive sgts and the length of sgts' validity intervals. These parameters are assumed to be steady on average during the execution of a query – a common assumption in streaming systems [167, 66]. Representing these parameters using sophisticated models that can capture complex distributions is possible, but continuously updating such models would be fairly expensive in a streaming environment. Consequently, the operator formulas in Section 6.3.1 use averages for these parameters.

Another critical parameter in operator cost formulas is the *selectivity factor* (f) – the number of tuples that satisfy a given predicate. Selectivity estimation is a challenging problem with a significant impact on the quality of cost model and cost estimations [102]. This chapter makes the standard assumptions of uniformity and inclusion to simplify the selectivity estimation. In the context of streaming graph queries, uniformity refers to the uniform distribution of source and target values of sgts in a given streaming graph, i.e., the uniform degree distribution of vertices in the snapshot graph of a streaming graph. Inclusion refers to the containment of vertex sets when two streaming graphs are joined together. Under these assumptions, the System-R approach [147] can be used to estimate the selectivity factors of SGA operators by maintaining the number of distinct source and target vertex counts for each input and intermediate streaming graph.

Remember that the main objective of this chapter is to demonstrate the feasibility of cost-based optimization of SGQs using the streaming graph query processing framework introduced in this thesis. Consequently, the prototype implementation and its experimental analysis presented in this chapter are designed under these assumptions. In real-world applications, the characteristics of streaming graphs are expected to fluctuate during the lifetime of a streaming graph query; consequently, the cost estimations and query planning might need to be adjusted periodically as the underlying streaming graphs evolve. The issue of adaptive query optimization is orthogonal to the design of such an SGQ optimizer and is not pursued further.

6.5 Experimental Analysis

6.5.1 Methodology

This section presents an experimental evaluation to validate the optimizer framework presented in this chapter. The objective is to understand (i) whether the cost model and its operator formulas can correctly predict the behaviour of operators and (ii) whether the optimizer can pick an "efficient" plan from the space of equivalent plans for a given SGQ using the cost model and the algebraic transformation rules. Prototype SGQ optimizer implementation described in Section 6.4 is integrated into the SGQ processor described in Chapter 5, and the same test environment is used for the evaluation.

The experimental evaluation presented here uses synthetic streaming graphs to control the degree distribution of vertices and to ensure that the underlying assumptions about the characteristics of the input streaming graph hold (Section 6.4.2). First, a synthetic graph dataset with multiple labels is generated using the gMark [19] graph generator. The generated graph consists of 1M vertices, 45M edges, and nine different edge labels. A continuous for loop consumes its edges and generates an input streaming graph by assigning monotonically increasing timestamps, which in turn controls the rate of the input streaming graph. As commonly done for the evaluation of streaming systems [65, 95], the input streaming graph is pushed through the query plan as fast as possible, which corresponds to the maximum input load the query processor can handle. Under these settings, the system time and the application time are not necessarily equivalent as the input rate depends on the system's processing capacity, and it might be greater than the original stream rate. Nonetheless, this does not alter either the operator state or the query answer, as both depend on the temporal semantics based on the application timestamp.

Furthermore, this reveals the maximum load that the underlying system can sustain under given parameters.

For each experiment, two measurements are reported: (i) measured system latency as the average time it takes to process sgts of one unit of application time (i.e., the first and last sgts are separated by one unit of application time), (ii) and the estimated model cost as the cumulative cost that is calculated by the query optimizer using the cost model presented in Section 6.3. Remember that the cost formulas for operator resource usage estimation use cost constants to mask implementation-specific details and system-dependent costs. A common practice in literature is to use constant weight factors that are either hard-coded by system developers or chosen empirically [101, 87, 109]. The prototype optimizer used here takes a similar approach and uses weight factors calculated empirically.

6.5.2 Validation of the Cost Model

WSCAN, UNION and FILTER are stateless operators where each sgt is processed on the fly, independent of others. The processing cost of stateless operators depends only on the rate of the input streaming graph. Figure 6.3 shows the measured system latency (left) and the estimated model costs (right) of WSCAN over input graph streams with various rates and interval lengths. It is seen that the estimated model cost of WSCAN only depends on the rate of the input graph stream, not the average length of validity intervals. Other stateless operators UNION and FILTER exhibit similar trends. Overall, it is seen that the relative performance of WSCAN with different parameters (i.e., rate and average interval length) are similar to estimated model costs.

PATTERN and PATH are stateful operators where the processing of each sgt depends on the internal operator state. Therefore, their processing costs are affected by both the rate and the average length of their input streaming graphs. In the following, the cost formulas for stateful SGA operators PATTERN and PATH are validated by comparing the measured and the estimated costs.

Figure 6.4 shows measured system (left) and estimated model costs (right) of PATTERN ($\bowtie_{\Phi}^{src1,trg2,d}(S_1, S_2)$ for $\Phi = \{src_2 = trg_1\}$) over streaming graphs with various characteristics. In this experiment, the total rate of the two input streaming graphs remains constant (1000 sgts/sec) while their relative rates vary. This ensures that, for a given validity interval length (i.e., window size), the total number of sgts maintained in the internal state remains the same while the output rate of PATTERN differs. In other words, the cost of maintaining the internal data structures are the same (e.g., insertion and expiration costs), whereas the cost of producing the output sgts changes. In line with the cost formulas for

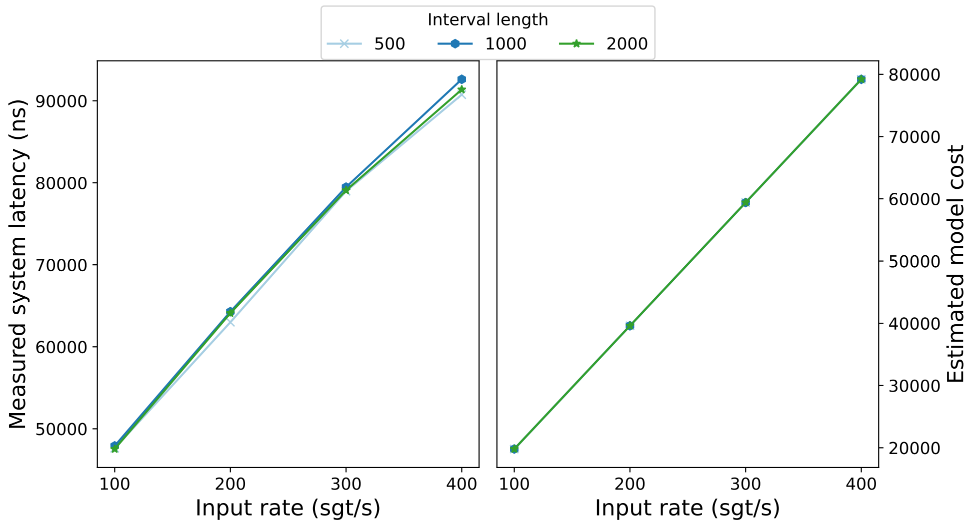


Figure 6.3: (left) Measured system and (right) estimated model costs WSCAN operator with varying input graph stream rates.

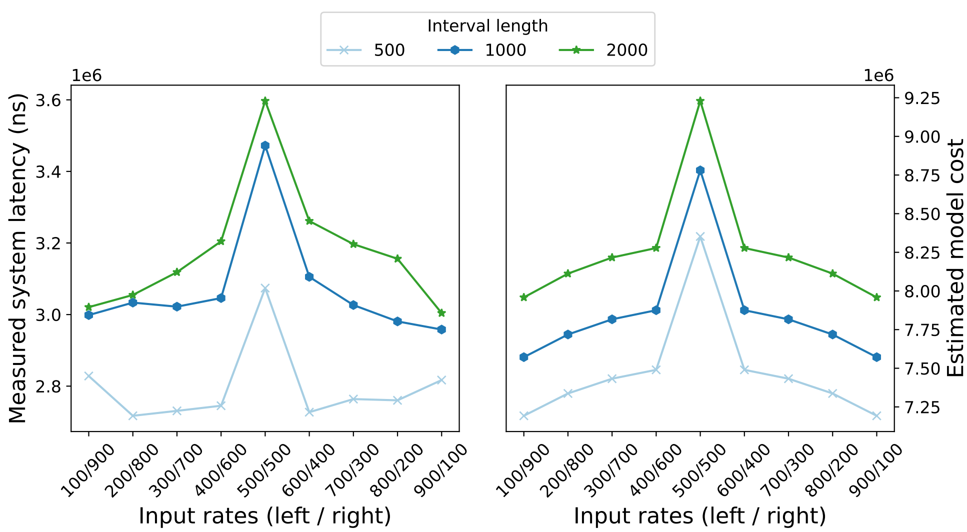


Figure 6.4: (left) Measured system and (right) estimated model costs PATTERN operator with varying input graph stream rates.

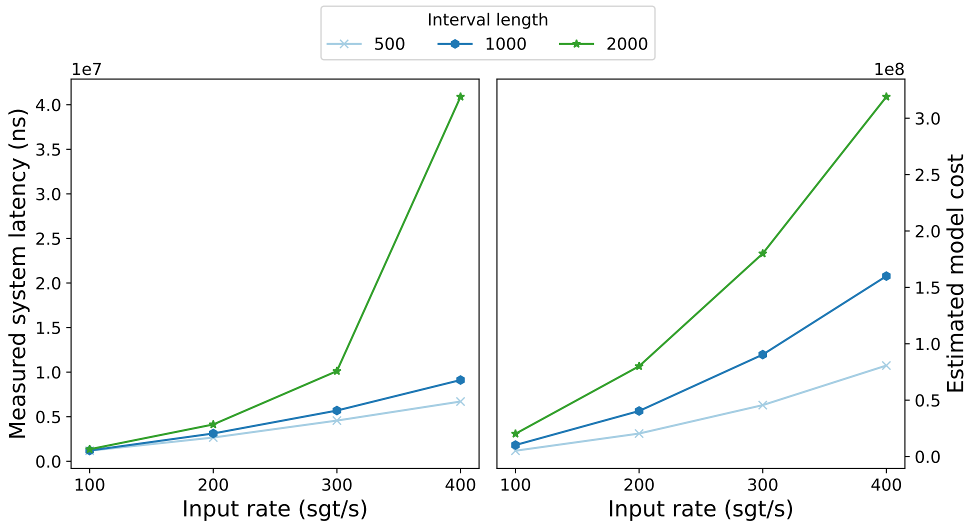


Figure 6.5: (left) Measured processing and (right) estimated model costs PATH operator with varying input graph stream rates.

PATTERN, the processing cost is highest when the rate of two input streaming graphs is equal as the rate of the output streaming graph is the highest under this combination.[2] Additionally, the processing cost of PATTERN is not affected by the ordering of its inputs. This is expected: the physical implementation of PATTERN is based on the symmetric hash join algorithm where the processing of the two input graph streams are identical (Section 5.4.2). Most importantly, it is seen that the cost model's estimation accurately captures the shape of PATTERN's resource usage graph.

A similar trend is observed for PATH. Figure 6.5 shows the measured system cost (left) and the estimated model cost (right) for $\mathcal{P}_R^d(S_a)$ where $R = a^+$ over input streaming graphs with different characteristics. Overall, the estimated model cost exhibits the same trend as the measured processing latency, though the actual differences between plans are not the same as predicted. This is a common trend observed in all experiments because the cost model only considers the dominant factors of physical operator implementations, not the system-specific implementation details such as queueing, scheduling, etc.

Table 6.2: Characteristics of the input streaming graphs used in the experiments for ordering complex query plans.

Input streaming graph	Rate r	Interval length ω
S_a	2000 sgt/s	100s
S_b	4000 sgt/s	100s
S_c	6000 sgt/s	100s
S_d	8000 sgt/s	100s

```
GRAPH VIEW output_stream AS (
  CONSTRUCT (s) -[:r]-> (t)
  MATCH (u1)
    (v)-[:a]->(s1)
    (s)-[:b]->(v2)
    (s)-[:c]->(v3)
    (s)-[:d]->(t)
  ON input_stream WINDOW(100) )
```

Figure 6.6: G-CORE representation of a star subgraph pattern query Q_s.

6.5.3 Ordering Complex Query Plans

The previous section provides an empirical validation of the cost functions for each SGA operator presented in Section 6.3. This section further validates the prototype SGQ optimizer framework by analyzing its optimization decisions, i.e., whether the optimizer picks an efficient plan among a set of equivalent plans for SGQs with complex patterns. For these experiments, the optimizer is modified to output all non-trivial plans and their corresponding costs for two SGQs with complex graph patterns: Q_s features a star-shaped subgraph pattern that tests the join ordering decisions of the proposed optimizer, and Q_p is recursive path pattern (similar to Q_4 from Table 5.1 in Chapter 5) that tests the PATH operator and its novel transformation rules. Figure 6.6 and 6.7 depict the G-CORE representations of Q_s and Q_p, respectively.

Consider the star-shaped subgraph pattern query Q_S: the following SGA expressions Q_s^1-Q_s^4 represent the four equivalent plans that are generated by the query optimizer using PATTERN's transformation rules. Figure 6.8 (right) illustrates estimated model costs per unit-time for all four plans over the same input streaming graphs (whose characteristics are

[2]See Equation 6.7 for the calculation of output streaming graph rate of PATTERN.

```
GRAPH VIEW output_stream AS (
  CONSTRUCT (u1) -[:l]-> (u2)
  MATCH (u1) -/ <:(a/b/c)+> /-> (u2)
  ON input_stream WINDOW(100) )
```

Figure 6.7: G-CORE representation of a recursive path navigation query Q_p.

given in Table 6.2): Q_s^3 has the lowest estimated cost among all plans, and picked by the proposed SGQ optimizer for Q_s. Figure 6.8 (left) shows the measured processing costs for these plans; the lowest cost plan for the start query is Q_s^3, as predicted by the optimizer. In addition, it is seen that the relative processing costs of the four plans are similar to the estimated costs, and the optimizer's estimations correctly order the query evaluation plans for Q_s.

- Q_s^1: $\bowtie_{s1=s2}^{t1,t2,r} \left(S_a, \bowtie_{s1=s2}^{s1,t2,bcd} \left(S_c, \bowtie_{s1=s2}^{s1,t2,bd} (S_b, S_d)\right)\right)$
- Q_s^2: $\bowtie_{s1=s2}^{t1,t2,r} \left(S_a, \bowtie_{s1=s2}^{s1,t2,bcd} \left(S_b, \bowtie_{s1=s2}^{s1,t2,cd} (S_c, S_d)\right)\right)$
- Q_s^3: $\bowtie_{s1=s2}^{t1,t2,r} \left(\bowtie_{s1=s2}^{s1,t2,abc} \left(S_c, \bowtie_{s1=s2}^{s1,t2,ab} (S_b, S_a)\right), S_d\right)$
- Q_s^4: $\bowtie_{s1=s2}^{t1,t2,r} \left(\bowtie_{s1=s2}^{s1,t2,abc} \left(S_b, \bowtie_{s1=s2}^{s1,t2,ac} (S_c, S_a)\right), S_d\right)$

Similarly, consider the recursive path navigation query Q_p: the following SGA expressions represent the seven plans that are generated by the prototype optimizer using PATH's transformation rules. The first expression, Q_p^1 represents the FA-based evaluation plan that is commonly used in literature for the evaluation of RPQs. Q_p^2 and Q_p^3 represent α-RA-based plans that are based on the materialization of intermediate results. Q_p^4-Q_p^7 represent novel hybrid plans that are possible due to PATH operator and its transformation rules. Figure 6.9 (left) shows the measured average processing time (i.e., latency) per unit application time over the input streaming graphs in Table 6.2: Q_p^2 is has the lowest processing cost. Estimated model costs for all seven plans are depicted in Figure 6.9 (right): the optimizer correctly estimates that Q_p^2 has the lowest processing cost per unit application time. Furthermore, the relative ordering of plans by the optimizer matches the ordering of plans by their measured processing cost, further validating the feasibility of the optimizer presented in this chapter.

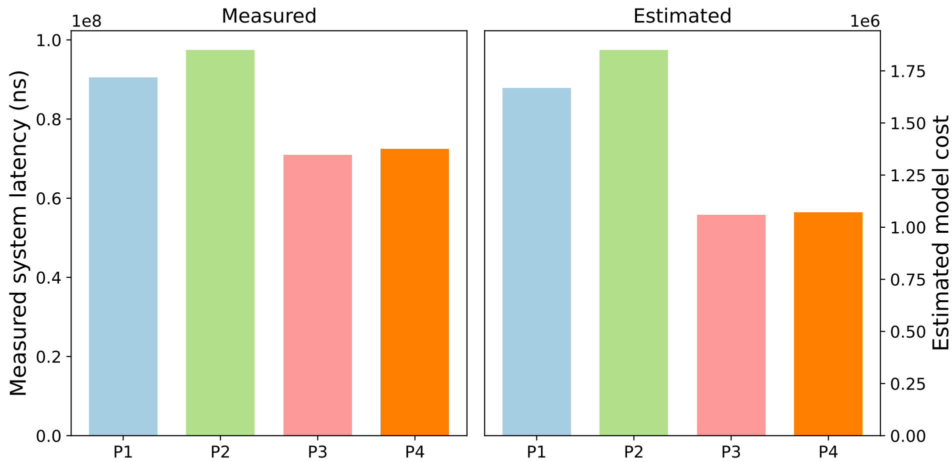


Figure 6.8: (left) Measured processing and (right) estimated model costs of different plans ($Q_s^1 - Q_s^4$) for the star pattern query Q_s.

- Q_p^1: $\mathcal{P}^l_{(a\cdot b\cdot c)^+}(S_a, S_b, S_c)$
- Q_p^2: $\mathcal{P}^l_{q^+}\Big(\bowtie^{src_1,trg_2,q}_{trg1=src_2} \big(\bowtie^{src_1,trg_2,p}_{trg_1=src_2} (S_a, S_b), S_c\big)\Big)$
- Q_p^3: $\mathcal{P}^l_{q^+}\Big(\bowtie^{q}_{trg1=src_2} \big(S_a, \bowtie^{src_1,trg_2,p}_{trg_1=src_2} (S_b, S_c)\big)\Big)$
- Q_p^4: $\mathcal{P}^l_{(d\cdot c)^+}\big(S_c, \bowtie^{src_1,trg_2,d}_{trg_1=src_2} (S_a, S_b)\big)$
- Q_p^5: $\mathcal{P}^l_{(a\cdot d)^+}\big(S_a, \bowtie^{src_1,trg_2,d}_{trg_1=src_2} (S_b, S_c)\big)$
- Q_p^6: $\mathcal{P}^l_{q^+}\Big(\bowtie^{src_1,trg_2,q}_{trg1=src_2} \big(\mathcal{P}^p_{a\cdot b}(S_a, S_b), S_c\big)\Big)$
- Q_p^7: $\mathcal{P}^l_{q^+}\Big(\bowtie^{src_1,trg_2,q}_{trg1=src_2} \big(S_a, \mathcal{P}^p_{b\cdot c}(S_b, S_c)\big)\Big)$

A common trend observed in all experiments is that the estimated model costs exhibit similar trends as the measured processing costs. However, actual differences between plans are not as high as predicted. This is primarily due to: (i) the cost model only considers the operations performed over the attributes of sgts and operators' internal data structures, not system-specific implementation details such as inter-operator queues and scheduling, (ii) and operator cost formulas do not consider system issues such as caching, parallelism,

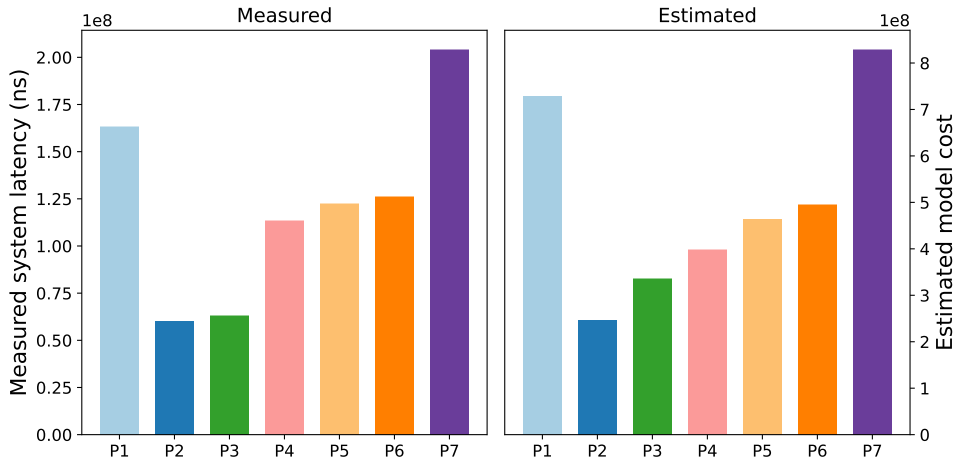


Figure 6.9: (left) Measured processing and (right) estimated model costs of different plans ($Q_p^1 - Q_p^7$) for the recursive path pattern query Q_p.

etc. Nonetheless, the ranking of plans by the estimated model cost matches the actual ranking.

6.6 Discussion

This chapter studies optimization of SGQs in the context of the streaming graph query processing framework proposed in this thesis. It formally defines the search space by introducing a set of transformation rules held in SGA. It is shown that some of the traditional relational transformation strategies such as join ordering and predicate pushdown apply to SGA's UNION, FILTER and PATTERN operators due to snapshot-reducibility. Additionally, new rules involving novel SGA operators WSCAN and PATH are described. These rules are expressed in the form of equivalences between SGA expressions, and they facilitate the generation of equivalent SGA expressions for a given SGQ through algebraic rewrites. Second, a cost model for estimating the resource usage of SGA-based query evaluation plans is introduced. This cost model identifies the necessary streaming graph characteristics to model resource usage of SGA operators and defines closed-form formulas for each SGA operator. The cost model presented in this chapter is based on the *per unit-time* cost model, originally developed for relational joins over relational streams [87]. Finally, a concrete implementation of a cost-based SGQ optimizer as an extension of Apache Calcite

– a Cascades-style extensible optimizer framework – is described. This prototype implementation incorporates the search space and the cost model presented in this chapter into Apache Calcite. It adapts Calcite's top-down (Cascades-style) search algorithm for plan space enumeration. The feasibility of cost-based optimization of SGQ is shown through an experimental analysis using this prototype implementation.

It is essential to highlight that the main objective of this chapter is to provide the foundational tools upon which SGQ optimizers can be built. Consequently, the prototype implementation described here has several limitations due to the underlying assumptions. First of all, the prototype SGQ optimizer lacks a sophisticated selectivity estimation mechanism, and it adopts the System-R approach for selectivity estimation. Consequently, the selectivity formulas make the standard assumptions on the distribution of values and inclusion of domains. Also, the characteristics of streaming graphs are expected to be steady on average, enabling the optimizer to use averages instead of complex distributions that are expensive to calculate and maintain. Experimental analysis shows that the optimizer can accurately estimate the relative resource usage of different plans and choose the "right" plan for a given query when the characteristics of input streaming graphs are inline with these assumptions. Nevertheless, it might produce estimates with significant errors in cases where these assumptions do not hold, or the system conditions and streaming graph characteristics might drastically change over time. Finally, the SGA search space consists of a single physical implementation for each SGA operator, limiting the space of possible execution plans considered by the prototype implementation. On the other hand, the SGQ optimization framework described in this chapter makes adding new physical implementations as simple as defining (i) a new *implementation rule* (Section 6.4) and (ii) an operator-specific cost formula. Section 7.2 discusses these limitations in detail and lays out possible directions for future research in the context of the query processing framework presented in this thesis.

Chapter 7

Conclusions and Future Work

7.1 Summary of Contributions

This thesis studies the problem of query processing over streaming graphs and introduces models and algorithms for representing, evaluating, and optimizing complex queries over streaming graphs. The primary motivation behind this study is to support an emerging class of applications that continuously monitor and process interaction data that can be modeled as a streaming graph. This is a challenging problem due to (i) the complexity of processing graph queries with subgraph patterns and path navigations, and (ii) the need for non-blocking, incremental techniques to tackle the unboundedness and arrival rate of real-world streaming graphs. This thesis develops a principled approach to supporting this class of workloads, and presents principled solutions to a number of technical challenges that need to be addressed. The main contribution is the design and implementation of a general-purpose streaming graph persistent query processing framework. This framework realizes the well-known steps of a query processing pipeline by rethinking its components in the context of streaming graph queries, from query representation and plan generation to cost-based query optimization and physical operator implementations.

The central query model, Streaming Graph Queries (SGQ), is introduced in Chapter 3. SGQ is based on a subset of Datalog that consists of binary, non-recursive predicates augmented with transitive closure. This formalism has multiple advantages. First, the SGQ model can provably express the class of workloads targeted in this thesis. It unifies subgraph patterns and path navigations by properly closing conjunctive queries under recursion. Second, its underlying data model treats paths as first-class objects, enabling queries to return and manipulate paths. Also, the SGQ model constitutes a streaming generalization of RPGQ [30] by incorporating time-based sliding windows into its query semantics, enabling it to formalize queries in existing graph query languages in the streaming context. This is demonstrated by mapping G-CORE constructs to SGQ.

The SGQ model precisely describes what the query results should be at any point time; however, it does not prescribe how SGQs can be evaluated efficiently. As in any streaming system, it is desirable for SGQs to be evaluated *incrementally*, avoiding recomputation of the entire results by only computing the changes to the output as new sgts arrive. Chapter 4 focuses on the most pressing challenge for incremental evaluation of streaming graph queries and proposes the first streaming algorithms for RPQs – the de-facto standard for expressing path navigations. The design space of streaming RPQ algorithms is categorized along two dimensions, and concrete algorithms that uniformly treat this design space are introduced along with their formal properties. These algorithms enable efficient, incremental evaluation of path navigations over streaming graphs and form the basis for a physical implementation of a general-purpose path navigation operator.

Chapter 5 formally introduces the foundational basis of the streaming graph query processor proposed in this thesis. Streaming Graph Algebra (SGA) is defined as a closure of a set of logical operators over streaming graphs. SGA provides the precise definition of operator semantics and query evaluation plans independent of low-level system details. SGA's expressivity is proven by showing a mapping from SGQs to SGA expressions, demonstrating the feasibility of the proposed algebraic approach for modelling SGQ evaluation plans. Chapter 5 also describes a prototype implementation of a streaming graph query processor based on SGA. This prototype consists of non-blocking, incremental algorithms as physical implementations of SGA operators. The feasibility and the performance benefits of the proposed SGA-based query processor are shown empirically.

Finally, Chapter 6 discusses the optimization of SGQs. The query optimization problem is defined as a search problem over the space of possible plans for a given query. This chapter first formally defines the search space over SGA expressions and introduces a set of rewrite rules in the form of algebraic equivalences for the systematic exploration of the search space. Then, a cost model for SGA-based query evaluation plans is developed. The cost model provides resource usage estimations for physical query evaluation plans and enables the optimizer to choose an "efficient" one among equivalent plans. Finally, a prototype implementation of a cost-based SGQ optimizer based on the Apache Calcite optimizer framework is described. This prototype employs a Volcano-style top-down search algorithm and systematically explores the search space through the proposed rewrite rules and cost model.

To conclude, this thesis addresses one of the most fundamental research challenges in streaming graph processing and provides the foundational tools for the design and development of a general-purpose streaming graph query processor. This is an important step towards the vision laid out in Section 1.2.2. The proposed techniques have been implemented as a part of the prototype system called S-Graffito[1]. The main objective of this prototype is to demonstrate the feasibility of the models and algorithms proposed in this thesis and to show the potential benefits that can be gained through empirical analysis over real-world and synthetic streaming graphs.

7.2 Directions for Future Research

7.2.1 Querying Graphs with Data

The class of queries considered in this thesis consists of structure-based predicates that query the graph topology. However, the Property Graph Model (PGM) contains data values with vertices and edges, and many practical applications query the data stored on the graph in addition to its topology [30]. It is possible to treat streaming graphs as relational streams by representing each attribute in a separate column and using relational stream languages to query the stored in the graph, such as CQL [15]. A critical issue involves supporting attribute-based predicates in recursive path navigations. Consider the recursive path expression of the running example given in Figure 1.1 and assume that each vertex contains attributes such as name, age, and city. A simple extension of this query that restricts paths only to contain users from the same city is already outside existing query models. One possible area for exploration is the use of the *register automata* model that allows comparisons of data values along paths [106, 104]. Preliminary results suggest that evaluating path queries with data is a non-trivial problem [124], and no systems support these in the streaming model.

7.2.2 Extending the Query Processor

The cost-based optimization framework used in the proposed prototype has several limitations that need to be improved in the future. First, it provides a single physical implementation for each SGA operator. Also, the prototype optimizer employs a System-R style selectivity estimation technique and makes the standard assumptions of uniformity, inclusion, and independence, which might produce significant estimation errors on real-world datasets [101].

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

Physical operator implementations described in Chapter 5 are exemplars to demonstrate the implementability of the SGA operators and to show their effectiveness; alternative physical implementations are certainly possible. For instance, Ammar et al. [9] introduce a worst-case optimal join algorithm for incremental evaluation of subgraph pattern queries that can be adapted as an alternative physical implementation of the PATTERN operator. As discussed in Chapter 6, new physical implementations can be incorporated by augmenting the cost model with new operator formulas and introducing corresponding *implementation* rules, which the proposed optimizer can use to navigate the "extended" space of plans. Furthermore, new algebraic equivalences in the form of transformation rules can be defined to extend the space of plans considered by the query optimizer.

Tackling the standard assumptions of uniformity, inclusion, and independence for selectivity estimation is one of the grand challenges in database research, and the streaming model poses additional challenges. Histogram-based methods are commonly adopted in traditional cost-based query optimizers, but their maintenance cost makes them unfeasible for streaming graphs. A potential solution is to use streaming graph summarization techniques such as TCM [159] that constructs a *graph sketch* incrementally. Graph-based sketches preserve the underlying graph's structure and can be used to approximate predicate selectivities. A recent trend in query optimization is to use learned models for cardinality estimation [158, 111, 112]. Existing research mostly focuses on static settings due to the high up-front cost of model learning. A potential avenue for further research is to investigate *online* learning techniques for incrementally maintaining the underlying model as the input streaming graphs evolve.

7.2.3 Adaptive query processing

Applications require predictable, stable performance to be robust to workload changes. Evaluation of persistent SGQs is particularly challenging as the cost of a query plan might change in the lifespan of a query due to changes in the system conditions such as available memory and network bandwidth, changes in the arrival rate or distribution of the input streaming graph. A possible direction for future research is to employ *adaptive* query processing and optimization techniques based on the framework proposed in this thesis. Two important questions that need to be answered are: (i) how to detect significant drifts in underlying streaming graph characteristics and (ii) how to react to these changes and adapt the physical execution plan efficiently.

The query optimizer and its corresponding cost model presented in Chapter 6 assume that the streaming graph characteristics are stable over time and use averages to quantify these characteristics. One potential solution for detecting drifts in input characteristics is maintaining online averages and periodically updating the cost estimations. When the difference between the original cost estimate and the updated cost estimate exceeds a pre-defined threshold, the query processor can re-evaluate whether the chosen plan is still "optimal" under the updated input streaming graph characteristics.

A significant drift in the input streaming graph characteristics might render the existing query plan "sub-optimal", and modifying or replacing the current execution plan is complicated in the presence of stateful operators like PATH and PATTERN. Dynamic plan migration requires deriving the internal state of the operators in the new plan from the old plan to produce correct results. Furthermore, the migration itself has an associated cost that should be considered during the optimization phase. The cost model proposed in this thesis can be augmented with the cost of plan migrations, enabling the query optimizer to assess the potential benefits of updating the execution plan against the cost of doing so.

7.2.4 Scaling-out SGQ Processing

This thesis focuses on centralized settings, but scale-out systems are arguably the most reasonable approach to tackle real-world streaming graphs' size and growth rate. A fundamental issue in designing scale-out systems concerns data partitioning, which is the process of physically or logically distributing a dataset to a set of machines. It enables many queries to be executed at different sites in parallel in the form of inter-query and intra-query parallelism. A recent development in graph partitioning is the streaming model that performs a single pass over the stream and makes partitioning decisions on the fly, a natural fit for applications considered in this thesis. Compared to traditional partitioning algorithms, the streaming model significantly reduces partitioning time and enables graph to be partitioned as it becomes available. A potential avenue for future research is distributed implementations of SGA's operators that can utilize streaming graph partitioning techniques to reduce the communication cost.

References

[1] 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. The design of the Borealis stream processing engine. In *Proc. 2nd Biennial Conf. on Innovative Data Systems Research*, pages 277–289, 2005.

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

[3] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. EmptyHeaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, 42(4):20:1–20:44, October 2017.

[4] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 431–446, 2016.

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

[6] Amir Aghasadeghi, Vera Zaychik Moffitt, Sebastian Schelter, and Julia Stoyanovich. Zooming out on an evolving graph. In *Proc. 23rd Int. Conf. on Extending Database Technology*, pages 25–36, 2020.

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

[8] Mehmet Altınel and Michael J Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proc. 26th Int. Conf. on Very Large Data Bases*, pages 53–64, 2000.

[9] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed evaluation of subgraph queries using worst-case optimal and low-memory dataflows. *Proc. VLDB Endowment*, 11(6):691–704, 2018.

[10] Khaled Ammar and M. Tamer Özsu. Experimental analysis of distributed graph systems. *Proc. VLDB Endowment*, 11(10):1151–1164, 2018.

[11] Renzo Angles, Marcelo Arenas, Pablo Barcelo, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, Oskar van Rest, and Hannes Voigt. G-CORE: A core for future graph query languages. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1421–1432, 2018.

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

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

[14] Apache Giraph. http://giraph.apache.org, 2016. Last accessed June 2019.

[15] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.

[16] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. Counting beyond a yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *Proc. 21st Int. World Wide Web Conf.*, pages 629–638, 2012.

[17] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 1–16, 2002.

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

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

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

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

[22] Pablo Barceló, Leonid Libkin, Anthony W Lin, and Peter T Wood. Expressive languages for path queries over graph-structured data. *ACM Trans. Database Syst.*, 37(4):31, 2012.

[23] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient algorithms for large-scale local triangle counting. *ACM Trans. Knowl. Discov. Data*, 4(3):13, 2010.

[24] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. Apache Calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 221–230, 2018.

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

[26] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 61–71, 1986.

[27] Carlyna Bondiombouy and Patrick Valduriez. Query processing in multistore systems: an overview. *Int. J. Cloud Computing*, 5(4):309–346, 2016.

[28] Angela Bonifati, Radu Ciucanu, and Aurélien Lemay. Learning path queries on graph databases. In *Proc. 18th Int. Conf. on Extending Database Technology*, pages 109–120, 2015.

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

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

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

[32] Angela Bonifati, Wim Martens, and Thomas Timm. Navigating the maze of Wikidata query logs. In *Proc. 28th Int. World Wide Web Conf.*, pages 127–138, 2019.

[33] Vladimir Braverman, Rafail Ostrovsky, and Dan Vilenchik. How hard is counting triangles in the streaming model? In *Proc. 40th Int. Colloquium on Automata, Languages, and Programming*, pages 244–254, 2013.

[34] Adam L Buchsbaum, Raffaele Giancarlo, and Jeffery R Westbrook. On finding common neighborhoods in massive graphs. *Theor. Comp. Sci.*, 299(1-3):707–718, 2003.

[35] Jean-Paul Calbimonte. Linked data notifications for RDF streams. In *WSP/WOM-oCoE@ ISWC*, pages 66–73, 2017.

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

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

[38] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink™: Stream and batch processing in a single engine. *Q. Bull. IEEE TC on Data Eng.*, 38(4):28–38, 2015.

[39] C-Y Chan, Pascal Felber, Minos Garofalakis, and Rajeev Rastogi. Efficient filtering of XML documents with XPath expressions. *VLDB J.*, 11(4):354–379, 2002.

[40] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proc. 17th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 34–43, 1998.

[41] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proc. 10th ACM SIGOPS/EuroSys European Conf. on Comp. Syst.*, pages 1:1–1:15, 2015.

[42] Sutanay Choudhury, Lawrence B. Holder, George Chin Jr, Khushbu Agarwal, and John Feo. A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs. In *Proc. 18th Int. Conf. on Extending Database Technology*, pages 157–168, 2015.

[43] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. on Comput.*, 32(5):1338, 2003.

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

[45] Isabel F Cruz and Theodore S Norvell. Aggregative closure: An extension of transitive closure. In *Proc. 5th Int. Conf. on Data Engineering*, pages 384–391, 1989.

[46] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *Proc. 13th Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 635–644, 2002.

[47] Daniele Dell'Aglio, Jean-Paul Calbimonte, Marco Balduini, Oscar Corcho, and Emanuele Della Valle. On correctness in RDF stream processor benchmarking. In *Proc. 12th Int. Semantic Web Conf.*, pages 326–342, 2013.

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

[49] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. Tigergraph: A native MPP graph database. *arXiv preprint arXiv:1901.08248*, 2019.

[50] Yanlei Diao, Mehmet Altinel, Michael J Franklin, Hao Zhang, and Peter Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Database Syst.*, 28(4):467–516, 2003.

[51] Curtis Dyreson, Fabio Grandi, Wolfgang Käfer, Nick Kline, Nikos Lorentzos, Yannis Mitsopoulos, Angelo Montanari, Daniel Nonen, Elisa Peressi, Barbara Pernici, John F. Roddick, Nandlal L. Sarda, Maria Rita Scalas, Arie Segev, Richard Thomas Snodgrass, Mike D. Soo, Abdullah Tansel, Paolo Tiberio, and Gio Wiederhold. A consensus glossary of temporal database concepts. *ACM SIGMOD Rec.*, 23(1):52–64, 1994.

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

[53] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. The LDBC social network benchmark: Interactive workload. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 619–630, 2015.

[54] Orri Erling and Ivan Mikhailov. RDF support in the Virtuoso DBMS. In Tassilo Pellegrini, Sóren Auer, Klaus Tochtermann, and Sebastian Schaffert, editors, *Networked Knowledge-Networked Media*, pages 7–24. Springer, 2009.

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

[56] Arash Fard, M Usman Nisar, Lakshmish Ramaswamy, John A Miller, and Matthew Saltz. A distributed vertex-centric approach for pattern matching in massive graphs. In *Proc. 2013 IEEE Int. Conf. on Big Data*, pages 403–411, 2013.

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

[58] George H. L. Fletcher, Jeroen Peters, and Alexandra Poulovassilis. Efficient regular path query evaluation using path indexes. In *Proc. 19th Int. Conf. on Extending Database Technology*, pages 636–639, 2016.

[59] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Martin Schuster, Petra Selmer, et al. Formal semantics of the language cypher. *arXiv preprint arXiv:1802.09984*, 2018.

[60] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1433–1445, 2018.

[61] Jun Gao, Chang Zhou, and Jeffrey Xu Yu. Toward continuous pattern detection over evolving large graph with snapshot isolation. *VLDB J.*, 25(2):269–290, 2016.

[62] Libo Gao, Lukasz Golab, M. Tamer Özsu, and Gunes Aluc. Stream WatDiv – a streaming RDF benchmark. In *Proc. Int. Workshop on Semantic Big Data*, pages 3:1–3:6, 2018.

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

[64] L. Golab. *Sliding Window Query Processing over Data Streams*. PhD thesis, University of Waterloo, 2006.

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

[66] Lukasz Golab and M. Tamer Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proc. 29th Int. Conf. on Very Large Data Bases*, pages 500–511, 2003.

[67] Lukasz Golab and M. Tamer Özsu. Update-pattern-aware modeling and processing of continuous queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 658–669, 2005.

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

[69] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. 10th USENIX Symp. on Operating System Design and Implementation*, pages 17–30, 2012.

[70] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: graph processing in a distributed dataflow framework graph processing in a distributed dataflow framework. In *Proc. 11th USENIX Symp. on Operating System Design and Implementation*, pages 599–613, 2014.

[71] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[72] G. Graefe and W. McKenna. The Volcano optimizer generator. In *Proc. 9th Int. Conf. on Data Engineering*, pages 209–218, 1993.

[73] Goetz Graefe. The Cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

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

[75] Ajeet Grewal, Jerry Jiang, Gary Lam, Tristan Jung, Lohith Vuddemarri, Quannan Li, Aaditya Landge, and Jimmy Lin. Recservice: Multi-tenant distributed real-time graph processing at Twitter. In *Proc. 10th USENIX Workshop on Hot Topics in Cloud Computing*, 2018.

[76] Andrey Gubichev. *Query Processing and Optimization in Graph Databases*. PhD thesis, Technische Universität München, 2015.

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

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

[79] M. Hammad, W. Aref, M. Franklin, M. Mokbel, and A. Elmagarmid. Efficient execution of sliding window queries over data streams. Technical Report CSD TR 03-035, Purdue University, 2003.

[80] 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. Nile: a query processing engine for data streams. In *Proc. 20th Int. Conf. on Data Engineering*, page 851, 2004.

[81] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Özsu, Xingfang Wang, and Tianqi Jin. An experimental comparison of Pregel-like graph processing systems. *Proc. VLDB Endowment*, 7(12):1047–1058, 2014.

[82] W. Hasan and H. Pirahesh. Query rewrite optimization in Starburst. Technical Report TR RJ 6367, IBM Alamden Research Center, August 1988.

[83] Martin Hirzel, Guillaume Baudart, Angela Bonifati, Emanuele Della Valle, Sherif Sakr, and Akrivi Vlachou. Stream processing languages in the big data era. *ACM SIGMOD Rec.*, 47(2):29–40, 2018.

[84] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In Zvi Kohavi and Azaria Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[85] Juraj Hromkovič, Sebastian Seibert, and Thomas Wilke. Translating regular expressions into small ε-free nondeterministic finite automata. *J. Comp. and System Sci.*, 62(4):565–588, 2001.

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

[87] J. Kang, J.F. Naughton, and S.D. Viglas. Evaluating window joins over unbounded streams. In *Proc. 19th Int. Conf. on Data Engineering*, pages 341–352, 2003.

[88] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An active graph database. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1695–1698, 2017.

[89] Bruce M Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proc. 24th Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 1131–1142, 2013.

[90] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 411–426, 2018.

[91] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.*, 23(2):253–278, 2014.

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

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

[94] André Koschmieder and Ulf Leser. Regular path queries on large graphs. In *Proc. 24th Int. Conf. on Scientific and Statistical Database Management*, pages 177–194, 2012.

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

[96] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter Heron: Stream processing at scale. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 239–250, 2015.

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

[98] Pradeep Kumar and H Howie Huang. Graphone: A data store for real-time analytics on evolving graphs. *ACM Trans. Storage*, 15(4):1–40, 2020.

[99] Jakub Łącki. Improved deterministic algorithms for decremental transitive closure and strongly connected components. In *Proc. 22nd Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 1438–1445, 2011.

[100] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *Proc. 10th Int. Semantic Web Conf.*, pages 370–388, 2011.

[101] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endowment*, 9(3):204–215, 2015.

[102] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. Query optimization through the looking glass, and what we found running the join order benchmark. *VLDB J.*, 27(5):643–668, 2018.

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

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

[105] Leonid Libkin, Juan L Reutter, Adrian Soto, and Domagoj Vrgoč. TriAL: A navigational algebra for RDF triplestores. *ACM Trans. Database Syst.*, 43(1):1–46, 2018.

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

[107] Ling Liu and M. Tamer Özsu, editors. *Encyclopedia of Database Systems*. Springer, 2009.

[108] Mengmeng Liu, N.E. Taylor, Wenchao Zhou, Z.G. Ives, and Boon Thau Loo. Recursive computation of regions and connectivity in networks. In *Proc. 25th Int. Conf. on Data Engineering*, pages 1108–1119, 2009.

[109] L.F. Mackert and G.M. Lohman. R* optimizer validation and perfromance evaluation for distributed queries. In *Proc. 12th Int. Conf. on Very Large Data Bases*, pages 149–159, 1986.

[110] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 135–146, 2010.

[111] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1275–1288, 2021.

[112] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *Proc. VLDB Endowment*, 12(11):1705–1718, 2019.

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

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

[115] Wim Martens and Tina Trautner. Dichotomies for evaluating simple regular path queries. *ACM Trans. Database Syst.*, 44(4):1–46, 2019.

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

[117] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2):25:1–25:39, 2015.

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

[119] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. Shared arrangements: Practical inter-query sharing for streaming dataflows. *Proc. VLDB Endowment*, 13(10):1793–1806, 2020.

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

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

[122] Amine Mhedhbi, Chathura Kankanamge, and Semih Salihoglu. Optimizing one-time and continuous subgraph queries using worst-case optimal joins. *ACM Trans. Database Syst.*, 46(2):1–45, 2021.

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

[124] Thomas Mulder, Nikolay Yakovets, and George H. L. Fletcher. Towards planning of regular queries with memory. In *Proc. 23rd Int. Conf. on Extending Database Technology*, pages 451–454, 2020.

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

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

[127] Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, September 2009.

[128] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *J. ACM*, 65(3):16, 2018.

[129] Hung Q Ngo, Christopher Ré, and Atri Rudra. Skew strikes back. *ACM SIGMOD Rec.*, 42(4):5–16, Feb 2014.

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

[131] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. Regular path query evaluation on streaming graphs. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1415–1430, May 2020.

[132] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. Evaluating complex queries on streaming graphs. In *Proc. 38th Int. Conf. on Data Engineering*, pages 272–285, 2022.

[133] Anil Pacaci and M. Tamer Özsu. Experimental analysis of streaming algorithms for graph partitioning. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1375–1392, 2019.

[134] Anil Pacaci, Alice Zhou, Jimmy Lin, and M. Tamer Özsu. Do we need specialized graph databases?: Benchmarking real-time social networking applications. In *Proc. 5th Int. Workshop on Graph Data Management Experiences & Systems*, pages 12:1–12:7, 2017.

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

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

[137] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. Real-time constrained cycle detection in large dynamic graphs. *Proc. VLDB Endowment*, 11(12):1876–1888, 2018.

[138] Juan L Reutter, Miguel Romero, and Moshe Y Vardi. Regular queries on graph databases. *Theor. Comp. Syst.*, 61(1):31–83, 2017.

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

[140] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endowment*, 11(4):420–431, 2018.

[141] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *VLDB J.*, 29:595—618, 2020.

[142] Siddhartha Sahu and Semih Salihoglu. Graphsurge: Graph analytics on view collections using differential computation. *arXiv preprint arXiv:2004.05297*, 2020.

[143] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei R. Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. The future is big graphs! A community view on graph processing systems. *Commun. ACM*, 64(9):62–71, 2021.

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

[145] Atish Das Sarma, Sreenivas Gollapudi, and Rina Panigrahy. Estimating PageRank on graph streams. In *Proc. 27th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 69–78, 2008.

[146] Luc Segoufin. Static analysis of XML processing with data values. *ACM SIGMOD Rec.*, 36(1):31–38, 2007.

[147] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 23–34, 1979.

[148] Konstantinos Semertzidis, Evaggelia Pitoura, and Kostas Lillis. TimeReach: Historical Reachability Queries on Evolving Graphs. In *Proc. 18th Int. Conf. on Extending Database Technology*, pages 121–132, 2015.

[149] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L. Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. Graphin: An online high performance incremental graph processing framework. In *Proc. Euro-Par 2016: Parallel Processing*, pages 319–333, 2016.

[150] S. Seufert, A. Anand, S. Bedathur, and G. Weikum. Ferrari: Flexible and efficient reachability range assignment for graph indexing. In *Proc. 29th Int. Conf. on Data Engineering*, pages 1009–1020, 2013.

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

[152] Arun Sharma. Dragon: A distributed graph query engine. https://engineering.fb.com/data-infrastructure/dragon-a-distributed-graph-query-engine/, March 2016.

[153] Xuchuan Shen, Lei Zou, M. Tamer Özsu, Lei Chen, Youhuan Li, Shuo Han, and Dongyan Zhao. A graph-based RDF triple store. In *Proc. 31st Int. Conf. on Data Engineering*, pages 1508–1511, 2015.

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

[155] Chunyao Song, Tingjian Ge, Cindy X. Chen, and Jie Wang. Event Pattern Matching over Graph Streams. *Proc. VLDB Endowment*, 8(4):413–424, 2014.

[156] Andy Seaborne Steve Harris. SPARQL 1.1 query language. https://www.w3.org/TR/sparql11-query/, 2013.

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

[158] Ji Sun, Guoliang Li, and Nan Tang. Learned cardinality estimation for similarity queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1745–1757, 2021.

[159] Nan Tang, Qing Chen, and Prasenjit Mitra. Graph stream summarization: From big bang to big crunch. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1481–1496, 2016.

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

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

[162] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[163] Tolga Urhan and M.J. Franklin. XJoin: A reactively-scheduled pipelined join operator. *Q. Bull. IEEE TC on Data Eng.*, 23:27, 2000.

[164] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

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

[166] Moshe Y Vardi. A theory of regular queries. In *Proc. 35th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 1–9, 2016.

[167] S. Viglas and J. Naughton. Rate-based query optimization for streaming information sources. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 37–48, 2002.

[168] Sarisht Wadhwa, Anagh Prasad, Sayan Ranu, Amitabha Bagchi, and Srikanta Bedathur. Efficiently answering regular simple path queries on large labeled networks. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, SIGMOD '19, pages 1463–1480, 2019.

[169] Xin Wang, Simiao Wang, Yueqi Xin, Yajun Yang, Jianxin Li, and Xiaofei Wang. Distributed Pregel-based provenance-aware regular path query processing on RDF knowledge graphs. *Proc. Web Conf. 2020*, 23(3):1465–1496, 2020.

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

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

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

[173] Yuke Yang, Lukasz Golab, and M. Tamer Özsu. ViewDF: Declarative incremental view maintenance for streaming data. *Inf. Syst.*, 71:55–67, 2017.

[174] Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. GRAIL: scalable reachability index for large graphs. *Proc. VLDB Endowment*, 3(1):276–284, 2010.

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

[176] Lei Zou, Jinghui Mo, Lei Chen, M. Tamer Özsu, and Dongyan Zhao. gStore: answering SPARQL queries via subgraph matching. *Proc. VLDB Endowment*, 4(8):482–493, 2011.

APPENDICES

Appendix A

Algorithm S-PATH

Section 5.4 describes the use of validity intervals in conjunction with an explicit WSCAN operator for devising alternative physical implementations for SGA's stateful PATTERN and PATH operators based on the *direct* approach. This section describes the novel *Streaming Path Navigation* (**S-PATH**) that can be used as an alternative physical operator for the PATH operator (Definition 34) in detail. In contrast to the streaming RPQ algorithm described in Chapter 4, Algorithm **S-PATH** utilizes the validity intervals of path segments to simplify the state maintenance in the absence of explicit deletions. Algorithm **RAPQ** in Chapter 4 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, it 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* [78], optimized for RPQ evaluation on streaming graphs. Instead, **S-PATH** utilizes the temporal pattern of sliding window movements and adopt the *direct* approach, i.e., it can *directly* determine expired tuples based on their validity intervals. This is possible due to the separation of the implementation of sliding windows from operator semantics via an explicit WSCAN operator.

Algorithm **S-PATH** 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 $\Delta - \texttt{PATH}$, that is used as the internal operator state during query processing. $\Delta - \texttt{PATH}$ is used to maintain a path segment, i.e., a partial result, between each pair of vertices in the form a spanning forest under the constraints of a given RPQ, consistent with Definition 34. Upon the arrival of an sgt, Algorithm **S-PATH** probes

$\Delta - \texttt{PATH}$ 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 36 (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. [1] 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, expired sgts can be *directly* found 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 37 ($\Delta - \texttt{PATH}$ 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, $\Delta - \texttt{PATH}$ is a collection of spanning trees (Definition 36) 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 $src = x$.*

$\Delta - \texttt{PATH}$ 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 3.2). Due to spanning-tree construction (Definition 36), actual paths can easily be recovered by following

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

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

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**

  $p \leftarrow$ `PATH`$(T_x, (v, t))$

  $R \leftarrow R + (x, v, O, [(v, t).ts, (v, t).exp), p)$

end

foreach *edge* $e(v, w) \in G_{ts}$ *s.t.* $\delta(t, \phi(e)) = q$ **do**

  if $(w, q) \notin T_x$ **then**

    R $\leftarrow$ R+ **Expand**(T_x, (v, t), (w, q), $e(v, w)$)

  end

  else if $(w, q).exp < min((v, t).exp, e.exp)$ **then**

    R $\leftarrow$ R+ **Propagate**(T_x, (v, t), (w, q), $e(v, w)$)

  end

end

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

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**

  $p \leftarrow$ `PATH`$(T_x, (v, t))$

  R $\leftarrow$ R $+ (x, v, O, [(v, t).ts, (v, t).exp), p)$

end

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**

    R $\leftarrow$ R+ **Propagate**$(T_x,\ (v, t),\ (w, q),\ e(v, w))$

  end

end

return R

the parent pointers; hence, $\Delta - \texttt{PATH}$ constitutes a compact representation of intermediate results for path navigation queries over materialized path graphs. $\Delta - \texttt{PATH}$ is designed 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 $\Delta - \texttt{PATH}$ 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 18). 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 $\Delta - \texttt{PATH}$, i.e., the target node (v, t) is already in T_x, Algorithm **S-PATH** compares its expiry timestamp with the new candidate (Line 23). 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.