Seraph: Continuous Queries on Property Graph Streams

Christopher Rost, Riccardo Tommasini, Angela Bonifati, Emanuele Della Valle, Erhard Rahm, Keith W Hare, Stefan Plantikow, Hannes Voigt, Petra Selmer

Seraph: Continuous Queries on Property Graph Streams

Christopher Rost
rost@informatik.uni-leipzig.de
Leipzig University & ScaDS.AI
Leipzig, Germany

Riccardo Tommasini
riccardo.tommasini@insa-lyon.fr
INSA Lyon
Lyon, France

Angela Bonifati
angela.bonifati@univ-lyon1.fr
Lyon 1 University & IUF
Lyon, France

Emanuele Della Valle
emanuele.dellavalle@polimi.it
Politecnico di Milano
Milano, Italy

Erhard Rahm
rahm@informatik.uni-leipzig.de
Leipzig University & ScaDS.AI
Leipzig, Germany

Keith W. Hare
keith@jcc.com
JCC Consulting
United States

Stefan Plantikow
stefan.plantikow@neo4j.com
Neo4j Inc.
United States

Petra Selmer
pselmer@bloomberg.net
Bloomberg L.P.
New York, United States

Hannes Voigt
hannes.voigt@neo4j.com
Neo4j Inc.
United States

ABSTRACT

The high expressiveness and elasticity of graphs has led to the design of a wealth of graph models and query languages, used by practitioners to model real-world processes. The property graph model and corresponding query languages, such as the widely used Cypher, have become popular in both industry and research. However, real-time data analysis and management is becoming increasingly important for today's businesses, but graph query languages lack the features to handle streaming graph data and their continuous query evaluation.

In this work, we propose Seraph, a Cypher-based language supporting native streaming features within industry-ready property graph query languages. We formally define the Seraph semantics by combining stream processing with property graphs and time-varying relations while treating time as a first-class citizen of the underlying semantics, thus laying a formal foundation necessary for future implementations. We further propose its syntax and showcase the usage of Seraph for emerging graph-based continuous queries of real-world industrial use cases.

1 INTRODUCTION

With the growing availability of information, interconnected data have become pervasive. Graphs, in particular, Property Graphs (PGs) [3] (also denoted as Labeled Property Graphs (LPGs)), are a widespread data model in many industrial domains such as healthcare, social media, cybersecurity, fraud detection and genomics. Coherently, declarative graph query languages like Cypher [22], G-CORE [4], and PGQL [49] have emerged as the formalisms of choice for expressing sophisticated information needs declaratively. Moreover, the efforts above are converging into GQL [20, 25], the future standard graph query language that will pave the road to workload portability and shared consensus to manipulate PGs.

Graphs not only exhibit a notable increase in volume but also demonstrate a significant level of dynamism [10]. When slow in frequency, the changes on graphs can be addressed by temporal graph data models and corresponding query languages [38] that include operators for exploring graph versions across time. On the other hand, a paradigm shift in the query model is needed when the frequency of changes grows and directly impacts the high throughput and low latency of query results, as it often occurs in streaming graph settings [34].

Continuous queries (CQs) [6, 42] are a class of queries that repeatedly reports the results until explicitly terminated. CQs are typically evaluated over data streams, i.e., unbounded sequences of timestamped data items [7, 42]. To deal with the unboundedness of the input streams, CQs include operators that leverage data timestamps and operate the query evaluation by recency.

In practice, CQs enable reactive analytics, and thus, they are popular in stream processing domains like network monitoring, real-time surveillance, micro mobility. In such domains, it is of paramount importance to output the results of the query before the data becomes stale. However, the high cost of designing and maintaining custom stream processing pipelines has paved the road to declarative continuous query languages [23]. The database literature shows that the declarative paradigm poses significant advantages in such domains, e.g., interoperability across systems, optimisation opportunity, and simplicity of use [33].

Now that CQs are becoming relevant for various property-graph-centric task [34, 39], it is important to bridge the gap for a declarative PG continuous query language. Table 1 shows examples of CQs for the stream processing domain mentioned above that would benefit from a graph stream data model: the first query, which devotes to **network monitoring**, asks for paths that denote anomalies in the routes to the egress switch (i.e. a router responsible for outgoing traffic in a time-based interval); the second query, which devotes to **real-time surveillance**, asks for computing the list of persons who passed by a crime scene within 30 minutes; the last query, which relates to **micro mobility**, looks for the violations of a business rule for limited time free bike rides in a given temporal lapse. It is worth noting here that the above CQs not only need to be repeatedly evaluated for incoming data, but they must also identify a temporal pattern to bind the results.

On the one hand, the CQs presented above show the need for enriching current declarative graph query languages with the necessary abstractions and expressive power to formulate continuous graph queries. However, current popular graph query languages, such as Cypher, lack these abstractions. Moreover, algebraic frameworks for streaming graph queries started to be

Domain	Examples of Graph Continuous Queries
Network Monitoring	What are the anomalous routes that connect to the egress switch **during the last 15 minutes**?
Real-Time Surveillance	Who has passed by a given crime scene **in the last 30 minutes**?
Micro Mobility	Did anyone violate the 20-minute free renting limit **in the last hour**?

Table 1: Summary of continuous information needs for use-cases in three different domains.

defined, along with data and query models and preliminary optimizations [34] but as of today these cannot be encoded as bulk queries in a declarative language. On the other hand, existing continuous declarative languages are either limited to the relational model [47] or limited to RDF. The former are incapable of expressive complex analytics such as path queries. At the same time, the latter have focused on the relational subset of SPARQL [46], neglecting advanced features such as regular path queries [34].

In this paper, we fill this gap and focus on the language aspects of graph continuous queries. Precisely, we present the design of syntactic and semantic components of a Cypher-based continuous query language for streaming property graphs, namely **Seraph**. Motivated by other ongoing GQL standardization efforts around graph query and schema languages [5, 20, 25], in which formal semantics need to be defined prior to any implementation, we define the syntax and semantics of **Seraph** and properly formalize the latter to avoid underlying ambiguities and incorrect behavior of the queries.

In designing such a language, we elicit a set of design requirements acquired from our industrial-strength use cases:

R1 **Declarative Semantics.** The language must be declarative to guarantee interoperable execution across systems, optimisations, and simplicity of adoption.

R2 **Continuous evaluation.** The language must have operators that allow the repeated evaluation over time, i.e., choosing a time interval and a sequence to evaluate the query.

R3 **Result emitting.** The language must include operators that allow controlling the report of results, i.e., what is part of the result and when it will be ready to be emitted.

R4 **Preserving expressiveness.** The language should preserve the expressiveness of the base language for querying a PG, i.e., everything that can be expressed in the base language can be expressed in the extended language.

Seraph results from a long-term collaboration between several academic institutions and Neo4j. It conjugates Cypher, a widely used graph query language with a key role in the ongoing GQL standardization, with windowing mechanisms at the core of continuous queries over streams [6, 12]. While the very first version of GQL (without temporal extensions) is expected in 2024, at the moment GQL is only available to the members of the standardization committee (ISO/IEC JTC1 SC32 WG3 Database Languages). It will take some time for GQL to be implemented into products even after the publication of the standard. This justifies and motivates our choice of focusing on Cypher as a basis for Seraph, given the wide availability of the former in several industrial products/use cases and its closeness to GQL. We believe that our work will help reach a consensus for the future temporal expansions of GQL, which are certainly deemed important but not included in the first version.

As a result, Seraph stands as a productive and industry-ready language that allows querying graph streams. The proposed language is simple, intuitive and highly expressive at the same time. Anyone who is familiar with Cypher and can address a problem in a static property graph setting will be able to use Seraph in a streaming context. The language and its detailed formalization, as presented in our work, lay the foundations for future implementations and are the necessary steps to be carried out in order to guarantee their underlying correctness. Summarizing, in this paper we make the following main contributions:

- we present the data model underlying graph continuous queries and formally describe the duality between a stream of property graphs and time-varying relations;
- we lay the foundations of a Cypher-based query model for continuous queries over property graphs streams by using the concept of snapshot reducibility from temporal relational databases;
- we formally define the syntax and semantics of Seraph, an easy-to-use Cypher-based query language that incorporate primitives for continuous evaluation. The latter is, to the best of our knowledge, nonexistent in current graph query languages. In contrast, they are urgently needed in industry-wide applications and desirable for the development of ongoing GQL standards.

The remainder of the paper is organised as follows: We will go into details of the micro-mobility example in Section 2. We provide an overview of the core of Cypher in Section 3, giving the preliminary knowledge needed to formalize Seraph, and explaining how a Cypher-only solution won't satisfy the requirements. Section 4 describes further two industrial use-cases and the use of Seraph to answer the continuous questions in Table 1. Section 5 contains the formal specification of the semantics and syntax of Seraph together with a solution of the running example. In Section 6, we give an outlook to future implementations. Finally, Section 7 discusses related work, while Section 8 concludes the paper.

2 RUNNING EXAMPLE

In this section, we describe the use case of fraud detection in the micro mobility domain mentioned in the introduction. The scenario of a fictional vehicle sharing provider described below is inspired by a real business scenario within a company namely *nextbike BY TIER*[1] from Leipzig, Germany. This company applies graph technologies for demand prediction, usage increase and optimization of rental stations and zones and their locations.

The company *RideAnywhere* is known as a leading company that operates public bike, scooter and car-sharing systems. Various rental stations and zones are available within a city, which offer electrically operated cars, bicycles (e-bikes) and e-scooters, and classic bicycles. A user can use a mobile app to rent an available vehicle at a rental station. RideAnywhere offers various price models: from half-hourly to monthly subscriptions. If a vehicle is returned, the app calculates the total price of the rental based on the duration and the existing subscription.

[1] https://www.nextbike.de

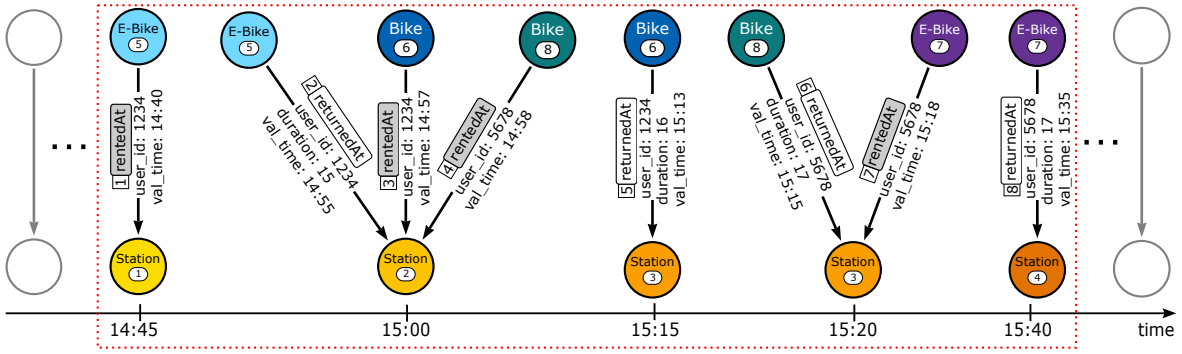

Figure 1: Stream of property graphs representing the events captured into the RideAnywhere Kafka queue.

Each rental station is connected to the RideAnywhere headquarters via a 4G connection and transmits rental and return events to a central Kafka event queue. Transmission takes place every 5 minutes for traffic and energy-saving reasons. A collected event thus represents all rentals and returns of the past 5 minutes, including the id of the station, the vehicles used and user information. We model the relationship within the event appropriately as a property graph consisting of station and vehicle nodes and relationships representing rentals and returns. Further, properties about rental time, return time, duration and unique identifiers for vehicles, users and stations, are provided.

The timeline reported in Figure 1 illustrates the events arriving in the Kafka queue of the RideAnywhere headquarter from 14:45h to 15:40h on a day in August 2022. Each event represents a property graph that contains rentals of a 5 minute period. Let us discuss the events of this example in detail.

14:45h An E-bike with id 5 was rented at station with id 1. The rental was done at 14:40h from a user with id 1234, which is stored in the edge properties. No further rentals or returns happened in the period [14:40,14:45).

15:00h The E-bike with id 5 was returned at station 2 at 14:55h. At the same station, two other bikes were rented. One of them from the same user, the other of user with id 5678.

15:15h The bike with id 6 was returned at station 3 at 15:13h.

15:20h Again at station 3, bike 8 was returned by user with id 5678 at 15:15h and an e-bike with id 7 was rented again by the same user 3 minutes later.

15:40h The E-bike with id 7 was returned at station 4 at 15:35h. No further rentals or returns happened in the period [15:35, 15:40).

Using the Neo4j Kafka Connector [32], all incoming events are merged and persisted in a Neo4j graph database. Vertices sharing the same identifier (e.g., for stations and bikes) will be merged to a single vertex. For the example graph stream of Figure 1, the resulting merged property graph of the interval from 14:45h to 15:40h is visualized in Figure 2. The graph consists of four station and four bike nodes as well as four rentals of two users represented by eight timestamped relationships.

To make the service attractive and affordable for students, an additional pricing model was developed with the local student union a few months ago. It provides that the first 20 minutes of e-bike or bicycle rental are free for valid students. If a vehicle is returned by a student and the 20-minute rental period has not been exceeded, no fee will be charged. If the time period is exceeded, the regular rate will be charged. RideAnywhere's goal is to increase the number of younger customers, achieve broader use of the service and generate more revenue by exceeding free times.

Shortly after its introduction, the student offer was widely used. The number of rentals increased by 35% compared to the average monthly usage before the student offer. However, after the first 3 months with this model, an analysis on the rental-graph found that students rarely made rentals longer than 20 minutes: only 5% of all student rentals per month. The RideAnywhere analytics team suspects that longer distances are covered by renting a vehicle again shortly thereafter (at a 5 minute interval), which is prohibited by company policy. This trick allows students to cover any length of distance using the free period multiple times.

The data analytics team is now expected to find a way to continuously detect users who use this trick so that they can be immediately alerted that this will lead to expulsion from the student offer by repeated violation. We present two solutions to this problem: one with Cypher including its drawbacks in Section 3.3, and one using Seraph that shows its strength in Section 5.4. It should be noted that the selected example with only minute-by-minute data does not do justice to the real-time character of the possibilities of continuous querying on a graph stream, but is suitable for demonstration.

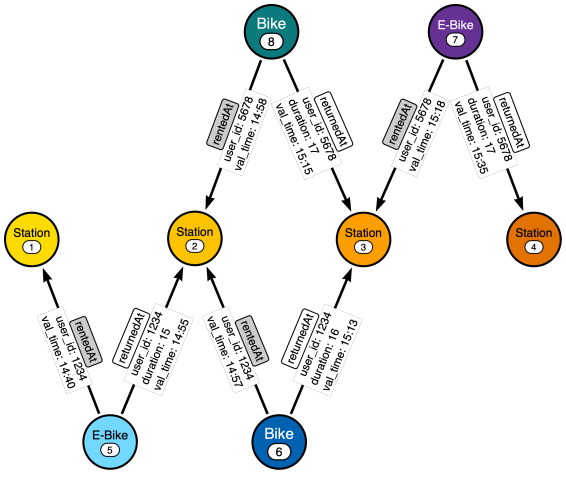

Figure 2: Graph resulting from loading the events from 14:45h to 15:45h into a Neo4j graph database.

3 BACKGROUND: THE CYPHER LANGUAGE

In the following, we provide as the necessary background the formal specification of a core subset of Cypher [22] for static property graphs. It consists of a data model including values, graphs, and tables (Section 3.1) and the query language along with its evaluation semantics including expressions, patterns, clauses, and queries (Section 3.2). Finally, we discuss the drawbacks of Cypher-only solution to the running example from Section 2 (Section 3.3).

3.1 Data model

Cypher's data model is based on three first class objects, namely values, property graphs, and relations, the latter being referred to as tables in Cypher's terminology. From [22], we consider three disjoint sets $\mathcal{K}$ of property keys, $\mathcal{N}$ of node identifiers, and $\mathcal{R}$ of relationship identifiers. These sets are all assumed to be countably infinite. The set $\mathcal{V}$ of values contains multiple inductively defined elements. We assume two base types: the integers $\mathbb{Z}$, and the type of finite strings over a finite alphabet Σ.

*Definition 3.1 (**Property graph [22]**).* Let $\mathcal{L}$ and $\mathcal{Y}$ be countable sets of node labels and relationship types, respectively. A property graph is a tuple $G = (N, R, src, trg, \iota, \lambda, \kappa)$ where:

- N is a finite subset of $\mathcal{N}$, whose elements are referred to as the nodes (also denoted as vertices) of G.
- R is a finite subset of $\mathcal{R}$, whose elements are referred to as the relationships (or edges) of G.
- src and trg are functions $R \to N$ that map a relationship to its source and target node, respectively.
- $\iota : (N \cup R) \times \mathcal{K} \to \mathcal{V}$ is a finite partial function that maps a pair (node|relationship,property key) to a value.
- $\lambda : N \to 2^{\mathcal{L}}$ is a function that maps each node id to a finite (possibly empty) set of labels.
- κ: $R \to \mathcal{Y}$ is a function that maps each relationship identifier to a relationship type.

For example, the graph of Figure 2 is formally represented in this model as a graph $G = (N, R, src, trg, \iota, \lambda, \kappa)$:

- $N = \{n_1, \ldots, n_8\}$; $R = \{r_1, \ldots, r_8\}$;
- $src = \{r_1 \mapsto n_5, r_2 \mapsto n_5, r_3 \mapsto n_6, r_4 \mapsto n_8, \ldots\}$;
- $trg = \{r_1 \mapsto n_1, r_2 \mapsto n_2, r_3 \mapsto n_2, r_4 \mapsto n_2, \ldots\}$;
- $\iota(r_1, \text{user\_id}) = 1234, \iota(r_1, \text{val\_time}) = 14{:}40, \ldots$;
- $\lambda(n_1) = \lambda(n_2) = \lambda(n_3) = \lambda(n_4) = \{\text{Station}\}$, $\lambda(n_5) = \lambda(n_7) = \{\text{E-Bike}\}, \lambda(n_6) = \lambda(n_8) = \{\text{Bike}\}$;
- $\kappa(r) = \begin{cases} \text{rentedAt} & \text{for } r \in \{r_1, r_3, r_4, r_7\}, \\ \text{returnedAt} & \text{for } r \in \{r_2, r_5, r_6, r_8\}. \end{cases}$

The flexibility of the property graph model also allows the modelling of hierarchies, e.g. by using multiple type labels per node (see λ), e.g., `:superclass:subclass` or dedicated relationship types, e.g., `(a)-[:isSubclassOf]->(b)`.

*Definition 3.2 (**Tables [22]**).* Let $\mathcal{A}$ be a countable set of names. A record is a partial function from names to values, conventionally denoted as a tuple with named fields $u = (a_1 : v_1, \ldots, a_n : v_n)$ where $a_1, \ldots, a_n$ are distinct names, and $v_1, \ldots, v_n$ are values. The order in which the fields appear is only for notation purposes. We refer to $dom(u)$, i.e., the domain of u , as the set $\{a_1, \ldots, a_n\}$ of names used in u . We use () to denote the empty record, i.e., the partial function from names to values whose domain is empty.

If $\mathcal{A}$ is a set of names, then a *table* with fields $\mathcal{A}$ is a bag, or multiset, of records u such that $dom(u) = \mathcal{A}$. A table with no fields is just a bag of copies of the empty record. Lastly, we define the bag difference of two tables T and T' as their bag difference, i.e., $T \setminus T'$.

```
query  ::= query° | query UNION query | query UNION ALL
query° ::= RETURN ret | clause query°
ret    ::= * | expr [AS a] | | ret , expr [AS a]
clause ::= [OPTIONAL] MATCH pattern_tuple [WHERE expr]
           | WITH ret [WHERE expr] | UNWIND expr AS a
pattern_tuple ::= pattern | pattern , pattern_tuple
```

Figure 3: Syntax of queries and clauses of Cypher [22].

3.2 Query language

The Cypher query language, whose syntax is presented in Figure 3, includes expressions, patterns, clauses, and queries. Due to the limited space, like in [22] we only focus on the latter two. A query is either a sequence of clauses ending with the `RETURN` statement, or a union of two queries. The semantics of queries associates a query Q and a graph G with a function $[[Q]]_G$ that takes a table and returns a table. Notably, the semantics of a query Q is a function and should not be confused with the output of Q. The evaluation of a query starts with the table containing one empty tuple, which is then progressively changed by applying functions that provide the semantics of Q's clauses. The composition of such functions, i.e., the semantics of Q, is a function again, which defines the output as:

$$output(Q, G) = [[Q]]_G(T())$$

where $T()$ is the table containing the single empty tuple ().

Let us have a look at the semantics of a pattern. The `MATCH` clause extends the set of field names of T by adding field names that correspond to names occurring in the pattern but not in u (the value to field assignments). It also adds tuples to T, based on found matches of the pattern in graphs. We show how to compute $[\![\ \texttt{MATCH}\ \pi]\!]_G(T)$, where π is the path pattern [22] to search for.

$$[\![\texttt{MATCH}\ \pi]\!]_G(T) = \biguplus_{u \in T} \{u \cdot u' | u' \in match(\pi, G, u)\}$$

The pattern π is evaluated on the graph G and extending T by adding field names and tuples based on the matches found in G. Each existing assignment $u \in T$ is extended by the assignments u' that are part of the finite set $match(\pi, G, u)$, which gives the semantics of the pattern matching of Cypher. Note that a pattern with variable length can be subsumed by a (possibly infinite) set of fixed length patterns, so-called rigid patterns [22]. Let π be a path pattern, $free(\pi)$ the union of all free variables of each node and relationship pattern occurring in π, $rigid(\pi)$ the set of all rigid patterns subsumed by π, G the graph, u an assignment, $dom(u)$ the domain of u (set of names) and p a path with node ids from N and relationship ids from R, the set is defined as:

$$match(\pi, G, u) = \biguplus_{\substack{p \in G \\ \pi' \in rigid(\pi)}} \left\{ u' \;\middle|\; \begin{array}{c} dom(u') = free(\pi) - dom(u) \\ \wedge (p, G, u \cdot u') \models \pi' \end{array} \right\}$$

Note that $\biguplus$ is a bag-union, i.e., a new occurrence u' is added to $match(\pi, G, u)$ if a new combination of π' and p is found that the pattern matching relation holds: $(p, G, u \cdot u') \models \pi'$, i.e., a path p in a graph G satisfies a pattern π under the assignments $u \cdot u'$ of values to the free variables of the pattern. Additional details of the Cypher semantics can be found in Francis et al. [22].

```
1  WITH datetime() - duration('PT60M') AS win_start,
2       datetime() AS win_end,
3  MATCH (:Bike)-[r:rentedAt]->(s:Station),
4        q = (b)-[:returnedAt|rentedAt*3..]-(o:Station)
5  WITH r, s, q, relationships(q) AS rels,
6    [n IN nodes(q) WHERE 'Station' IN labels(n) | n.id] AS
       hops
7  WHERE ALL(e IN rels WHERE
8    win_start <= e.val_time <= win_end AND
9    e.user_id = r.user_id AND e.val_time > r.val_time AND
10   (e.duration IS NULL OR e.duration < 20) )
11 RETURN r.user_id, s.id, r.val_time, hops
```

Listing 1: Cypher query to retrieve users that use the free period for two subsequent rentals in the last hour.

3.3 Running Example vs Cypher

Following up on the Section 2 example to detect subsequent rentals of the same user in the last hour, we designed the Cypher query shown in Listing 1 that implements one possible but limited solution representing this pattern. The drawbacks of this solution are evaluated at the end of this section.

The first part from section 3.3 to section 3.3 defines two timestamps as bounds of a 1h window from the moment of the query execution.

Lines 3-4 define the patterns: A bike was rented at a station s from which a path with at least a length of 3 relationships ends at a station o. The dynamic recursive pattern is assigned to a path variable q for later use. Lines 7-8 define a condition that the timestamp of all relationships of the path q have to be in the 1h window. The selection at lines 9-10 ensure the same user for all rentals and returns, guarantee that the first rental ended chronologically before the second starts (section 3.3) and both rentals do not exceed the free period of 20 minutes (section 3.3). The user id, time of the first rental and ids of all involved stations (derived in section 3.3) are returned, as stated in section 3.3.

Table 2 reports the result of the query evaluation at 15:40, showing that in the last hour, the users with ids 1234 and 5678 illegally extended their free rental time each by a second subsequent rental.

However, this one-time Cypher query computes the information need for the graph changes of one specific hourly interval, but has several drawbacks. First, the PG data model on which Cypher is based is static, i.e., there is no support of a continuous stream of graph elements. Further, the query has to be continuously evaluated on the most recent events and the results need to be computed every 5 minutes from a defined time instant, which results in the continuous evaluation requirement **R2**. Moreover, they want to get user 1234 returned at 15:15 and user 5678 at 15:40, thus, only new results as soon as the last event arrived in the last hourly period, which results in the result emitting requirement **R3**.

With Cypher, this could be realized only by external code that executes this query every 5 minutes. However, such a workaround would break the declarative paradigm (violating **R1**). Moreover,

r.user_id	s.id	r.val_time	hops
1234	1	14:40	[2,3]
5678	2	14:58	[3,4]

Table 2: Results of the Cypher query in Listing 1 at 15:40h.

the underlying system would be unaware of the continuous semantics, which would almost certainly lead to suboptimal query evaluation and possibly incorrect execution. In fact, each query will run isolated from the other, possibly considering caching mechanism design for the static case. Thus, a language like Cypher lacks a query mechanism that natively controls the continuous evaluation of the query and the result emission while at the same time preserving the expressive power of the non-streaming language, which leads to the expressiveness requirement **R4**.

4 SERAPH BY EXAMPLES

Before we get into the technical definitions, we pick up the two industrial use cases from Section 1 to justify the design (w.r.t. the requirements) and formalization of Seraph. The goal is to give high-level intuition of how a Seraph query looks and clarify what queries we target in this work. To simplify understanding of the syntax extensions, the original Cypher keywords are shown in blue and those introduced by Seraph are shown in green.

4.1 Network Monitoring

Computer networks span all levels of the stack, from physical connections up to mobile and microservices constituting a company's cloud. Graphs offer a natural way of modelling such scenarios and performing network optimization, asset management and inventory mapping. Network management is thus intrinsically a graph problem. While graph query languages like Cypher play a key role in investigating dependencies and in running diagnostic analyses (e.g., the root cause of a past network fault), Seraph offers the possibility to execute network impact analysis continuously.

Continuous Information Need: We want to monitor the network connectivity for anomalous routes continuously.

Let's assume that we model the network endpoints (e.g., servers, routers, switches and racks) of the data center as nodes and the "cables" between them as relationships. For instance, a rack HOLDS a switch that ROUTES an interface that CONNECTS a router in a network. We consider connections redundant if one of the cables gets loose or cut, i.e., the ROUTES relationship between a switch's interface and the network breaks, the number of hops can increase, but no rack can become unreachable. We know from the configuration of the network that the shortest routes from all racks to the egress router require on average 5 hops, but network events may cause this path to be longer, and we observed a standard deviation of 0.3 hops. We can identify anomalous routes using the z-score, i.e., the number of standard deviations σ by which an individual x is above or below the mean value δ of the population with $(x - \delta)/\sigma$. Our patterns are routes whose

```
1  REGISTER QUERY anomalous_routes STARTING AT datetime() {
2    MATCH path = allShortestPaths(
3    (rack:Rack)-[:HOLDS|ROUTES|CONNECTS*]-(r:Router:Egress))
4    WITHIN PT10M
5    WITH rack, avg(length(path)) as 10minAvg, path
6    WHERE (10minAvg - 5 / 0.5) >= 3
7    EMIT path
8    SNAPSHOT
9    EVERY PT1M
10 }
```

Listing 2: Monitoring computer networks using Seraph.

length has a z-score larger than 3, i.e., it is longer than 99,9% of the paths.

Listing 2 illustrates how to encode this need in a Seraph query. At each time instant, an arriving property graph represents the configuration of the entire network. The query uses the `WITHIN` (section 4.1) and the reporting `EVERY` (section 4.1) clauses to define a 10 minutes wide sliding window that reports every minute (i.e., `PT1M`) starting from the current system time (section 4.1), which meets requirement **R2**. The query finds the shortest paths from each rack to the egress router (section 4.1 and 3) and computes the average length of those paths in the last 10 minutes (section 4.1).

If the z-score of those paths related to each rack is greater than 3 (section 4.1), all paths are emitted for every evaluation, which meets requirement **R3**. Two extensions achieve this: First, using the `EMIT` clause (section 4.1) to specify the projections for the result stream (here all shortest paths by the path variable) and second by the `SNAPSHOT` streaming operator (section 4.1), which specifies that for each evaluation all result tuples will be emitted regardless of whether they have already been emitted in the previous evaluation. The result of this continuous query is a stream of so-called time-varying tables containing possibly anomalous routes.

4.2 Crime Investigations

From fraud detection to security, encompassing surveillance and contact tracing, investigations often require *connecting the dots*. Data models like POLE (Person-Object-Location-Events) underpin a number of analyses that require the identification of patterns [44]. POLE was originally intended for historical analyses that one can perform using graph query languages like Cypher. However, POLE includes temporal metadata that Seraph can exploit. Thus, it already unlocking a number of additional analyses, including, but not limited to, real-time surveillance and contact tracing.

Continuous Information Need: We shall monitor in real-time *who* is passing by crime scenes and detecting potential suspects.

As we adopt the POLE model for surveillance, we model *crimes* and *calls* as `Events`, which `OCCURRED_AT` a `Location`. Moreover, we assume that a number of smart cameras, which can identify each `Person` passing by (`NEAR_TO`), are deployed in different `Locations` within the city of London. We also consider suspects whoever has been convicted (`PARTY_TO`) for a crime of the same type as the one reported. Moreover, assuming that on average a person walks about 5km in an hour, we restrict the scope of the monitoring to an area of 3km from the crime scenes and a time range of 15 minutes.

Listing 3 illustrates how to encode the information-need above in a Seraph query. The query focuses on the last 15 minutes, reporting every 5 minutes starting at the current system time. To this extent, it uses the `WITHIN` clause once per `MATCH` (section 4.2 and section 4.2), and controls the results reporting using the `EVERY` clause (section 4.2) and `STARTING AT` (section 4.2), which satisfies requirement **R2**. The query monitors the streams of crime reports (Lines 2-4) and crosschecks if anyone, who is identified by a smart-camera, was a convicted criminal (Lines 5-8). To restrict the search space, the query looks only for cameras within 3km from the crime scenes and to those suspects that had taken part in a crime of the same type before. The functions `point()` and `distance()` are user-defined functions to perform geo-spatial comparisons. As an output, the query emits the last

```
1  REGISTER QUERY watch_for_suspects STARTING AT datetime() {
2    MATCH (call:Event)-[:OCCURRED_AT]->(l:Location)
3    WITHIN PT15M
4    WITH call, point(l) AS crime_scene
5    MATCH (crime:Event)<-[:PARTY_TO]-(person:Suspect)-[:
       NEAR_TO]->(last_seen:Location)
6    WITHIN PT15M
7    WITH call, crime, person, last_seen,
8      distance(point(last_seen), crime_scene) AS distance
9    WHERE distance < 3000 AND call.type=crime.type
10   EMIT person, last_seen, call.description
11   SNAPSHOT
12   EVERY PT5M
13 }
```

Listing 3: Looking for suspects in crime scenes using Seraph.

seen location, the suspect description, and the crime references by `EMIT` (section 4.2) and `SNAPSHOT` (section 4.2), satisfying requirement **R3**. With the queries for both use cases, one can see that Seraph expands Cypher and thus does not reduce the expressiveness, which addresses requirement **R4**.

5 FORMALIZATION OF SERAPH

This section presents how Seraph supports streaming computations while preserving the expressiveness of the Cypher language. The key elements of Seraph are aligned with the requirements of Section 1 and are summarized as follows:

- a data model that extends the PG model used by Cypher to model streams of property graphs;
- a query model for continuous query evaluation with full control of reporting (**R2,R3**);
- syntax and semantics of novel time-aware operators over the aforementioned data model (**R1,R2,R3,R4**).

Throughout all following descriptions, we use the notation summarised in Table 3.

5.1 Data Model

We first explain how the data model of Cypher can be extended to deal with property graph streams, where each graph maintains its evolution. The first component of Seraph's data model is a linearly ordered discrete *time domain* Ω like in [6, 31].

Concept	Notation	Set notation
Time instant / time domain	ω	Ω
Time interval	τ	-
Table	T	-
Time-annotated Table	$\widetilde{T}$	$\widetilde{\mathcal{T}}$
Time-varying Table	Ψ	-
Mapping	u	-
Time-annotated Mapping	μ	-
Property Graph	G	-
Snapshot Graph	$\widetilde{G}$	-
Property Graph Stream	S	-
Property Graph Substream	$\widetilde{S}$	$\widetilde{\mathcal{S}}$
Window	w	W

Table 3: Summary of notation conventions.

*Definition 5.1 (**Time**).* Ω is a infinite sequence of *time instants* $(\omega_1, \omega_2, \dots) \in \Omega$. A *time unit* is the difference between two consecutive time instants $(\omega_{i+1} - \omega_i)$ and it is constant. A *time interval* $\tau = [\omega_o, \omega_c)$ is a left-close right-open interval which starts at ω_o and ends at ω_c. Formally, it holds $\tau = \{\omega_i | \omega_i \in \Omega \wedge \omega_o \leq \omega_i < \omega_c\}$.

According to the definition above, we can define property graphs arriving in a sequence of time instants as a property graph stream (see Figure 1 for an example).

*Definition 5.2 (**Property Graph Stream**).* A Property Graph Stream S is an unbounded ordered sequence of pairs (G,ω), where:

- G is a property graph as per Definition 3.1, and
- ω is a non-decreasing timestamp.

$$S = ((G_1, \omega_1), (G_2, \omega_2), (G_3, \omega_3), (G_4, \omega_4), \dots)$$

Handling stream unboundedness is essential to Seraph's semantics. Thus, we introduce the notion of a *snapshot graph* that, in turns, builds on the concepts of *property graph substream* and *union of property graphs*.

*Definition 5.3 (**Property Graph Substream**).* Given a property graph stream S and a time interval $\tau = [\omega_o, \omega_c)$, we denote a finite subset of S in τ as a *property graph substream*:

$$\widetilde{S}_\tau = \widetilde{S}_{\omega_c}^{\omega_o} = \{(G, \omega) | (G, \omega) \in S \wedge \omega \in \tau, \Omega \wedge \omega_o \leq \omega < \omega_c\}$$

*Definition 5.4 (**Union of two Property Graphs**).* Assume that $G_1 = (N_1, R_1, src_1, trg_1, \iota_1, \lambda_1, \kappa_1)$ and $G_2 = (N_2, R_2, src_2, trg_2, \iota_2, \lambda_2, \kappa_2)$ are Property Graphs. Under UNA [41], we define the union of two Property Graphs as:

$$G_1 \cup G_2 = \begin{pmatrix} N_1 \cup N_2, R_1 \cup R_2, src_1 \cup src_2, trg_1 \cup trg_2, \\ \iota_1 \cup \iota_2, \lambda_1 \cup \lambda_2, \kappa_1 \cup \kappa_2 \end{pmatrix}$$

, if both graphs are consistent otherwise $G_1 \cup G_2 = \emptyset$.

G_1 and G_2 are consistent iff $\forall r \in R_1 \cap R_2$, it holds that $src_1(r) = src_2(r), trg_1(r) = trg_2(r), \kappa_1(r) = \kappa_2(r)$ and $\iota_1(r,k) = \iota_1(r,k) \forall k \in \mathcal{K}$ and $\forall n \in N_1 \cap N_2$, it holds that $\lambda_1(n) = \lambda_2(n)$ and $\iota_1(n,k) = \iota_1(n,k) \forall k \in \mathcal{K}$.

*Definition 5.5 (**Snapshot Graph**).* Given a time interval $\tau = [\omega_o, \omega_c)$, a *snapshot graph* $\widetilde{G}_\tau$ (also $\widetilde{G}_{\omega_o}^{\omega_c}$) is the result of the union of all property graphs $G \in \widetilde{S}_\tau$ to a single property graph using the union operation of Definition 5.4. It holds:

$$\widetilde{G}_\tau = \widetilde{G}_{\omega_o}^{\omega_c} = \bigcup_{G_i \in \widetilde{S}_{\omega_o}^{\omega_c}} G_i$$

Figure 2 shows the snapshot graph $\widetilde{G}_{14:45}^{15:45}$ resulting from coalescing the substream $\widetilde{S}_{14:45}^{15:45}$ highlighted with a red border in Figure 1.

In Section 3.2, we recall that clauses in a Cypher-query are functions that take a tableand output a table, potentially expanding the number of fields and adding new tuples. Similarly, in Seraph, we consider the time-based extensions of the notion above. In particular, a *time-varying table*, which is inspired by the time-varying relations from [9], generalizes the notion of the table into a function that maps the time Ω to a finite table. Moreover, we introduce *time-annotated tables* to extend Cypher's table with temporal boundaries.

*Definition 5.6 (**Time-annotated Table**).* Given a time interval $\tau = [\omega_o, \omega_c)$, we define a *time-annotated table* $\widetilde{T}_\tau$ (also $\widetilde{T}_{\omega_c}^{\omega_o}$) as a bag or multiset of records $\tilde{\mu}$, where each is a partial function from names to values extended with names for the temporal annotations of the interval bounds ω_o and ω_c. Extending the convention used for Cypher's tables, we denote them as a tuple:

$$\tilde{\mu} = (a_1 : v_1, \dots, a_n : v_n, win\_start : \omega_o, win\_end : \omega_c)$$

where $a_1, \dots, a_n$ are distinct names, and $v_1, \dots, v_n$ are values. The names `win_start` and `win_end` are reserved Keywords in Seraph justified by their use as identifiers for the window bounds, as per Definition 5.9. The order in which the fields appear is only for notation purposes. We refer to $dom(\tilde{\mu}) = \mathcal{A}$ as in Definition 3.2.

r.user_id	s.id	r.val_time	hops	win_start	win_end
1234	1	14:40	[2,3]	14:40	15:40
5678	2	14:58	[3,4]	14:40	15:40

Table 4: Time-annotated table as extension of Table 2.

For instance, Table 4 extends Table 2 with the aforementioned temporal annotations `win_start` and `win_end` with the values ω_o and ω_c, respectively.

*Definition 5.7 (**Time-varying Table**).* Let $\widetilde{\mathcal{T}}$ be the set of all possible $\widetilde{T}$ in Ω. A *time-varying table* Ψ is a function that maps every time instant $\omega \in \Omega$ to a time-annotated Table $\widetilde{T} \in \widetilde{\mathcal{T}}$:

$$\Psi : \Omega \rightarrow \widetilde{\mathcal{T}}$$

Given a time-varying table Ψ, we use the term $\Psi(\omega)$ to refer to the time-annotated table identified by the time-varying table at the given time instant ω. Moreover, we pose the following constraints on the definition of Ψ:

- Consistency, i.e., Ψ always identifies a time-annotated table.

$$\Psi(\omega_i) = \widetilde{T}_{\omega_c}^{\omega_o} \text{ s.t. } \forall \mu \in \widetilde{T} : \mu.\omega_o \leq \omega_i < \mu.\omega_c$$

- Chronologicality, i.e., Ψ always identifies the time-annotated table with the earliest (minimal) opening timestamp.

$$\nexists \widetilde{T}_j \text{ s.t. } \mu_j \in \widetilde{T}_j, \mu_j.\omega_o \leq \omega_i < \mu_j.\omega_c$$

$$\forall \mu_i \in \Psi(\omega_i), \mu_j.\omega_o \leq \mu_i.\omega_o$$

- Monotonicity, i.e., Ψ always identifies subsequent time-annotated tables for subsequent time instants.

$$\forall \omega_i, \omega_j \text{ s.t. } \omega_i < \omega_j, \forall \mu_i \in \Psi(\omega_i)$$

$$\forall \mu_j \in \Psi(\omega_j), \mu_i.\omega_o < \mu_j.\omega_o \leq \mu_i.\omega_c < \mu_j.\omega_c$$

For instance, the time-annotated table shown in Table 4 would be identified by a given $\Psi(\omega_i)$ for any ω_i such that 14:40 $\leq \omega_i <$ 15:40. Based on the presented Seraph's data model, we are ready to define the query model in the next section.

5.2 Query Model

This section presents the query model of Seraph that extends Cypher to enable continuous queries over a property graph stream and thus satisfies the requirements **R2** and **R3**. Indeed, Cypher supports only one-time queries, which are evaluated once by the Cypher engine and whose result is a finite table. Seraph queries, on the other hand, are intended to be continuously evaluated until explicitly halted on a potential infinite input stream.

This paradigm-shift in the query execution model is named *Continuous Semantics*, i.e., processing an infinite input produces an infinite output [42]. Continuous semantics poses the challenge of formalizing a non-terminating evaluation. In practice, it implies that the result of a *continuous query* is the set of results that

would be returned if the query would be executed at every time instant. Intuitively, if the objective computation is assumed to be stateless, continuous semantics can be achieved simply operating on each individual element in the input stream. In Seraph, this is the case for what concerns *data ingestion*. In fact, Cypher supports graph-based data ingestion by mapping elements of an input source, e.g., CSV, into property graphs. Similarly, Seraph ingestion operates on one event at time as shown in Listing 4[2].

On the other hand, the most common way to accomplish continuous semantics for stateful computations is via *snapshot reducibility*. In particular, we adapt the definition from [34], which in turn was adapted from [29], as follow:

*Definition 5.8 (**Snapshot Reducibility**).* Let S be a Property Graph Stream, CQ a continuous query, and Q its non-streaming counterpart. Snapshot reducibility states that each snapshot of the result of evaluating CQ over S is equivalent to applying Q over a snapshot of S, i.e.,

$$\forall \omega_i \in \Omega, w = [\omega_o, \omega_c)\ s.t.\ \omega_o \leq \omega_i < \omega_c, CQ(S)_w == Q(S_w)$$

Snapshot reducibility induces the definition of operators, named *Windows*, that chunk the stream into finite snapshots for defining the evaluation scope. Several alternative window semantics exist [48]. In Seraph, we focus on time-based windows which operate according to the temporal annotation of the stream elements to define intervals that help select finite portions of the input stream. A time-based window is *deterministic*, iff the set of intervals it subsumes is independent of the timestamps of stream elements.

*Definition 5.9 (**Time-based window**).* A time-based window $w = [\omega_o, \omega_c)$ is a time interval between a start time instant ω_o and an end time instant ω_c (exclusive). Let the triple $(\omega_0, \alpha, \beta)$ be a window configuration, where

- ω_0 is the earliest timestamp defining the start of the first window instance,
- α is the window size (in time units), and
- β is the slide size (in time units), such that two consecutive windows overlap of at most α - β.

A window operator $\mathcal{W}(\omega_0, \alpha, \beta)$ identifies a infinite set of windows:

$$\mathcal{W}(\omega_0, \alpha, \beta) = \left\{ w_i = [\omega_{o_i}, \omega_{c_i}) \;\middle|\; \begin{array}{l} i \in \mathbb{N}_0 \wedge \omega_{o_i} = \omega_0 + i\beta \wedge \\ \omega_{c_i} = \omega_0 + i\beta + \alpha \end{array} \right\}$$

If further holds that $|\omega_{o_i} - \omega_{c_i}| = \alpha$ and $\exists w_{i+1} s.t. |\omega_{o_i} - \omega_{o_{i+1}}| = \beta$, i.e., the distance of the lower bounds of two succeeding windows w_i and w_{i+1} is the sliding size β.

Applying a time-based window operator $\mathcal{W}$ to a Property Graph Stream S deterministically identifies an infinite set of substreams $\widetilde{\mathcal{S}}$, which lays the ground of continuous query execution.

$$\tilde{\mathcal{S}} = \mathcal{W}(\omega_0, \alpha, \beta)(S) = \left\{ \widetilde{S}_w | \forall w \in \mathcal{W}(\omega_0, \alpha, \beta) \right\}$$

[2]Example borrowed from Neo4j Kafka Connector: https://neo4j.com/docs/kafka/

```
LOAD STREAM FROM 'kafka:///bikes.stream' AS event
MERGE (b:Bike {id: event.bike_id})
MERGE (s:Station {id: event.location_id})
CREATE (b)-[:rentedAt {val_time: event.time,
                       user_id: event.uid}]->(s)
```

Listing 4: Example of graph-based ingestion in Seraph.

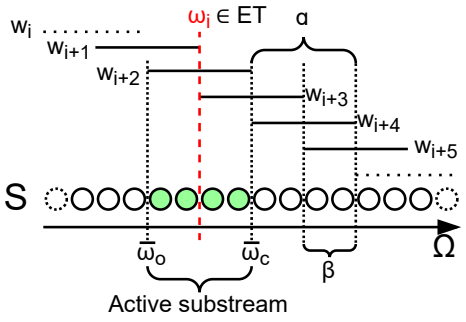

Figure 4: Selecting the active substream.

As per their characterisation [8], continuous queries yield their results as if the queries were evaluated for every time instant. Since such an approach is impractical, stream processing engines typically control the execution by customising the reporting of results [21]. However, delegating the definition of the reporting to the internals of the engines has caused idiosyncrasies in the operational semantics [1] in the past that may lead equivalent queries to produce different results on different engines [18]. Moreover, declarative control of the query results reporting is a well-known stream processing desideratum [40]. To this extent, we define the sequence of *evaluation time instants* as follows.

*Definition 5.10 (**Evaluation time instants**).* We define the sequence of time instants at which an evaluation of the query occurs as *evaluation time instants*. Such a sequence, namely ET, is potentially infinite. Notably, the sequence depends on the initial time instant ω_0 and the slide size β, as defined in Definition 5.9. In particular, we define the ET sequence as follows:

$$ET = \{\omega \mid (\omega - \omega_0)/\beta = 0\}$$

For every $\omega_i \in ET$, a query evaluation is triggered. It is now necessary to identify the property graph substream $\widetilde{S}_\tau$ with $\omega_i \in \tau$, from which a snapshot graph $\widetilde{G}_\tau$ is constructed that is the input of the query evaluation. We refer to $\widetilde{S}_\tau$ as the *active substream*.

*Definition 5.11 (**Active Substream**).* Given a time instant ω_i and the infinite set of all substreams $\tilde{\mathcal{S}}$, the *active substream* $\widetilde{S}_{\overline{w}}$ is the *earliest* property graph substream of all substreams that are valid at ω_i. I.e., it exists one window $\overline{w} = [\overline{\omega_o}, \overline{\omega_c}) \in \mathcal{W}(\omega_0, \alpha, \beta)$ such that $\omega_i \in \overline{w}$ and $\forall w = [\omega_o, \omega_c) \in \mathcal{W}(\omega_0, \alpha, \beta) : \overline{\omega_o} = min(\omega_o)$.

For hopping time-based window operators (also denoted as *tumbling*), the identification is intuitive, since there is just one substream per time instant: $\widetilde{S}_{\overline{w}} = \widetilde{S}_\tau$ with $\overline{w} = \tau = [\omega_o, \omega_c) = [\overline{\omega_o}, \overline{\omega_c})$. For overlapping time-based window operators (also denoted as *sliding*), i.e., $\mathcal{W}(\omega_0, \alpha, \beta)$ s.t. $\beta < \alpha$, multiple substreams could be identified at each ω_i. In such scenario, we consider the one with earliest opening timestamp as defined above.

Figure 4 illustrates the identification of the active substream. One can see the set of windows $w_i, w_{i+1}, \ldots$ where each has the size α and a distance of β. The infinite property graph stream S is represented as multiple circles ◯. For a given evaluation time instant $\omega_i \in ET$, marked with a red dashed line, two windows exist that include this time instant: w_{i+2} and w_{i+3}. Note that $\omega_i \notin w_{i+1}$, since a window is a close-open interval excluding the upper interval bound. From both windows w_{i+2} and w_{i+3} we select the one with the earliest (smallest) lower interval bound ω_o

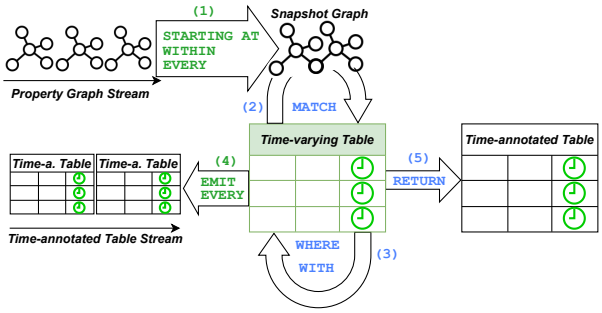

Figure 5: Seraph's data and query model Interaction.

as $\overline{w} = [\overline{\omega_o}, \overline{\omega_c})$. The substream $\widetilde{S}_{\overline{w}}$ is thus the active substream, whose property graphs are marked with green circles in the figure.

The union of all property graphs of $\widetilde{S}_{\overline{w}}$ results in a snapshot graph $\widetilde{G}_{\overline{w}}$, which we call *active snapshot graph*. On each evaluation time instant, the inner Cypher query is evaluated on the respective active snapshot graph. Each query evaluation results in a time-varying table Ψ that holds the tuples μ representing the time-annotated mappings of found matches. Finally, the continuous semantics implies an infinite output stream as result of a query evaluation. *Streaming operators*, as defined by Arasu et al. [6], reintroduce the temporal dimension in the data to construct a stream. We adapt this approach for creating a output stream of timestamped tuples (μ, ω) from the time-varying table $\Psi(\omega)$.

*Definition 5.12 (**Streaming Operators**).* A streaming operator is defined by the pair (Ψ,ω), i.e., a time-varying table and a time instant, typically the current evaluation time instant $\omega_{now} \in ET$. We differentiate three streaming operators:

- The RStream outputs each tuple derived from Ψ timestamped with the evaluation time.

$$RStream(\Psi, \omega) = \{(\mu, \omega) | \mu \in \Psi(\omega)\}$$

- The IStream outputs the results that are part of the current evaluation result but are not in the previous one.

$$IStream(\Psi, \omega_j, \omega_{j-1}) = \{(\mu, \omega_j) \mid \mu \in \Psi(\omega_j) \setminus \Psi(\omega_{j-1})\}$$

- The DStream outputs the results that are part of the previous evaluation result but are not in the current one.

$$DStream(\Psi, \omega_j, \omega_{j-1}) = \{(\mu, \omega_j) \mid \mu \in \Psi(\omega_{j-1}) \setminus \Psi(\omega_j)\}$$

After a detailed formalization of the query model in this section, the syntax and semantics of Seraph will be discussed in the following by bringing all the introduced concepts together.

5.3 Formal Syntax and Semantics

Seraph's key components are declarative (**R1**) *clauses* and *queries* that operate timely on the presented data model (Section 5.1) and query model (Section 5.2). From now on we will color Seraph syntax in green, leaving Cypher syntax in blue. We preserve the expressiveness of Cypher by defining only extensions, which satisfies requirement **R4**. In Figure 5, we illustrate the interactions and transitions from one component to another. In the upper left corner, we can see the input property graph stream, formally defined as S in Definition 5.2. The combination of three clauses (marked (1)), namely `STARTING AT`, `WITHIN` and `EVERY`, form the configuration of the window operator $\mathcal{W}$ from Definition 5.9. The window operator generates substreams $\widetilde{S}_\tau$ from the property graph stream S, each of which is combined into a snapshot graph $\widetilde{G}_\tau$.

The semantics of a `MATCH` clause is the pattern matching which takes as input a (initially empty) time-varying table Ψ, evaluates a pattern π matching on the snapshot graph $\widetilde{G}_\tau$, and in turn generates a time-varying table Ψ with extended set of field names and rows as output. In the figure, this is shown as a semicircular arrow marked with (2). The set of assignments that are the result of a `MATCH` clause as a time-varying table can be filtered via a selection using the `WHERE` clause (marked (3)) and thus again has a time-varying table as the result. Likewise, a projection of a time-varying table can be made via the `WITH` clause (also marked with (3)), which serves as input for another `MATCH` clause. This concept is taken from Cypher and allows the combination of several `MATCH` clauses.

In Seraph, the output of the evaluation result of one (or more) `MATCH` clauses can be emitted in two ways: a) as a stream of time-annotated tables $\widetilde{T}$ via the `EMIT` clause (marked with (4)) or b) as a single time-annotated table $\widetilde{T}$ via `RETURN` clause (marked with (5)). The former a) converts each time-varying table into a time-annotated table at each evaluation time ET using the projections specified by `EMIT` and the evaluation time instants specified by `EVERY`. It thus creates a stream of time-annotated tables. Second b) emits only one result. At the first evaluation time instant after the start time (defined by `STARTING AT`), the query is evaluated and the resulting time-varying table is converted into a time-annotated table using the projections specified by `RETURN`.

After this high-level overview, we can now present the formal syntax of a Seraph query, which is given in Figure 6. The semantics of expressions of Cypher, like values, variables, maps, lists etc., remain unchanged and can be derived from [22]. Furthermore, we provide a Seraph query parser open-source on GitHub [15].

Queries. The `REGISTER QUERY` clause allows for registering a new query with name $a \in \mathcal{A}$ into the system that implements Seraph. The name is used to identify the registered query and allows editing and deleting a previously registered query. The `STARTING AT` clause defines the first evaluation time instant, which is important for all window semantics, since from this points all windows are defined. This time instant, given as ISO8601 datetime, is used as the configuration ω_0 of the window operator from Definition 5.9, and is constant for the registered query.

The following body of the Seraph query is encapsulated by curly braces {...} and consists of three parts: the *query*, the *stream operator* and the *evaluation interval*. A *query* is a sequence

```
query^Srph ::= REGISTER QUERY a STARTING AT time {  a ∈ 𝒜
               query^Δ
               stream_op
               EVERY range }
query^Δ    ::= RETURN ret | EMIT ret | clause^Δ query^Δ
clause^Δ   ::= MATCH pattern_tuple WITHIN range
               [WHERE expr]
               | WITH ret [WHERE expr] | UNWIND expr AS a
stream_op  ::= ON ENTERING | ON EXIT | SNAPSHOT
range      ::= <ISO_8601_duration>
time       ::= <ISO_8601_datetime>
ret        ::= * | expr [AS a] | | ret , expr [AS a]
```

Figure 6: Seraph's syntax based on Cypher's one in Figure 3.

$$
\begin{aligned}
[\![\texttt{RETURN} *]\!]_{\widetilde{G}}(\Psi,\omega) &= \Psi(\omega) \text{ where } \omega \in [\omega_o, \omega_c), \text{ and } \omega_o, \omega_c \text{ are the time annotations of } \Psi(\omega)\\
[\![\texttt{EMIT} *]\!]_{\widetilde{G}}(\Psi,\omega) &= \forall \omega_e \in ET\ [\![\texttt{RETURN} *]\!]_{\widetilde{G}}(\Psi,\omega_e) \text{ a proposal}\\
[\![\texttt{EMIT} * \texttt{ON ENTERING}]\!]_{\widetilde{G}}(\Psi,\omega) &= [\![\texttt{EMIT} *]\!]_{\widetilde{G}}(\Psi,\omega), \text{where } \Psi = \{\mu \mid \mu \in \Psi(\omega) \setminus \Psi(\omega-1)\}\\
[\![\texttt{EMIT} * \texttt{ON EXIT}]\!]_{\widetilde{G}}(\Psi,\omega) &= [\![\texttt{EMIT} *]\!]_{\widetilde{G}}(\Psi,\omega), \text{where } \Psi = \{\mu \mid \mu \in \Psi(\omega-1) \setminus \Psi(\omega)\}\\
[\![\texttt{EMIT} * \texttt{SNAPSHOT}]\!]_{\widetilde{G}}(\Psi,\omega) &= [\![\texttt{EMIT} *]\!]_{\widetilde{G}}(\Psi,\omega), \text{where } \Psi = \{\mu \mid \mu \in \Psi(\omega)\}\\
[\![\texttt{WITH} *]\!]_{\widetilde{G}}(\Psi,\omega) &= \Psi(\omega) \text{ if } \Psi(\omega) \text{ has at least one field}\\
[\![\texttt{WITH ret WHERE expr}]\!]_{\widetilde{G}}(\Psi,\omega) &= \Psi(\omega) \text{ if } \Psi(\omega) \text{ has at least one field}\\
[\![\texttt{STARTING AT } \omega_0 \texttt{ MATCH } \pi \texttt{ WITHIN } \alpha \texttt{ EVERY } \beta]\!]_S &= [\![\texttt{MATCH } \pi]\!]_S^{W(\omega_0,\alpha,\beta)}(\Psi,\omega)\\
&\widehat{=}\ [\![\texttt{MATCH } \pi]\!]_{W(\omega_0,\alpha,\beta)(S)}(\Psi,\omega)\\
&\widehat{=}\ [\![\texttt{MATCH } \pi]\!]_{\widetilde{S}^{\omega_o}_{\omega_c}(\omega)}(\Psi,\omega)\\
&\widehat{=}\ [\![\texttt{MATCH } \pi]\!]_{\widetilde{G}_{\overline{w}}}(\Psi,\omega)\\
&= \biguplus_{\mu\in\Psi(\omega)} \{\mu\cdot\mu' \mid \mu' \in \overline{match}(\pi,\widetilde{G},\mu)\}
\end{aligned}
$$

Figure 7: Formal semantics of Seraph query and clauses.

of clauses ending with the `RETURN` or `EMIT` statement. Both contain the return list, which is either $*$, or a sequence of expressions, optionally followed by `AS` a, to provide their names. They define what to include in the query result set. The *stream operator* determines which streaming operator is used, which are defined in Definition 5.12. In particular, the `SNAPSHOT` clause specifies that the *RStream* operator has to be used, while the `ON ENTERING` and `ON EXIT` clauses allow for selecting *IStream* and *DStream*, respectively. The *evaluation interval*, i.e., sequence of evaluation time instances, can be specified using the `EVERY` clause, together with the `STARTING AT` clause. In particular, the `EVERY` clause defines the frequency of the evaluation, which can be specified with an ISO 8601 duration. The `STARTING AT` clause, instead, defines the first evaluation time instant as an ISO 8601 datetime.

Clauses. Seraph clauses are functions that take time-varying tables and produce time-varying tables. Analogous to Cypher, matching clauses are pattern matching statements of the form `MATCH pattern WITHIN range WHERE expr`, where `WHERE` is optional. The width parameter of windows is defined using the `WITHIN` clause, which is attached to every `MATCH` and its pattern definition. Thus, every pattern can be matched in its own window width. The `MATCH` clause extends the set of field names of $\Psi(\omega)$ by adding field names that correspond to names occurring in the pattern but not in μ. It also adds tuples to $\Psi(\omega)$, based on matches of the pattern that are found in the snapshot graph $\widetilde{G}_\tau$. Analogous to the Cypher definitions is `UNWIND` another clause that expands the set fields, and `WITH` clauses that can change the set of fields. In addition, `WITH` allows query parts to be chained together, piping the results from one to be used as starting points or criteria in the next.

Finally, we model the continuous evaluation process by including the evaluation time in the Cypher evaluation semantics. The continuous query answering is done by executing the query at each time instant of the sequence *ET* Given a fixed time instant, the operators can work in a time-agnostic way composing the semantics of Cypher in the one of Seraph. The semantics of queries associates a query *SQ* and a snapshot Property Graph $\widetilde{G}$ with a function $[\![SQ]\!]_{\widetilde{G}}$ that takes a time-varying table and a time instant and returns a time-varying table. The evaluation of a query starts with the time-varying table containing one empty tuple, which is then progressively changed by applying functions that provide the semantics of *SQ*'s clauses. The composition of such functions, i.e., the semantics of *SQ*, is also a function, which defines the output as:

$$output(SQ,\widetilde{G},\omega) = [[SQ]]_{\widetilde{G}}(\Psi,\omega)$$

This new concept requires a revision of the definitions of the existing Cypher evaluation of queries, clauses and expressions. We show all continuous evaluation semantics of redefined queries and clauses in Figure 7. In particular, the semantics of the `MATCH` clause is described through the set $\overline{match}(\pi,\widetilde{G},\mu)$, which is a redesign of the $match(\bar{\pi},G,u)$ of Cypher (Section 3.2) with respect to the continuous evaluation semantic and the concept of snapshot graphs. Let π a path pattern, $\widetilde{G}$ a snapshot graph, μ a time-annotated assignment, p a path in $\widetilde{G}$, π' a path pattern in the set of all rigid paths rigid(π) and $(p,\widetilde{G},\mu * \mu') \models \pi'$ as satisfaction of π' in a path p in $\widetilde{G}$, the set of matches is defined as follows:

$$\overline{match}(\pi,\widetilde{G},\mu) = \biguplus_{\substack{p\in\widetilde{G}\\ \pi'\in\text{rigid}(\pi)}} \left\{ \mu' \;\middle|\; \begin{array}{c}\text{dom}(\mu')=\text{free}(\pi)-\text{dom}(\mu)\\ \wedge (p,\widetilde{G},\mu*\mu')\models\pi'\end{array} \right\}$$

For space reasons, we did not include the semantics of non-essential language components like `UNWIND`, `OPTIONAL`, renaming, and expressions. However, under the snapshot reducibility assumption [34], the continuous extension of such an operation is trivial. For a complete overview, we invite the interested reader to read the Cypher technical report [22].

5.4 Running Example vs Seraph

In this section we define a continuous query for the micromobility example of Section 2. With Seraph, the analytics team of RideAnywhere can register a continuous query that checks the rentals for student users applying the described trick of subsequent rentals. Lets discuss the Seraph query depicted in Listing 5, which continuously monitors the rentals for the pattern.

The `REGISTER QUERY` clause (section 5.4) allows for naming and registering the query into the system that manages Seraph queries. To define the time when the first evaluation will start, the `STARTING AT` clause is used, with a time instant. The query itself (section 5.4 to 13) is the query body that defines the pattern, its conditions, the projections and result emitting. Let us go through the query and compare it with the Cypher solution given in

```
REGISTER QUERY student_trick STARTING AT 2022-10-14T14:45
{
  MATCH (:Bike)-[r:rentedAt]->(s:Station),
        q = (b)-[:returnedAt|rentedAt*3..]-(o:Station)
  WITHIN PT1H
  WITH r, s, q, relationships(q) AS rels,
    [n IN nodes(q) WHERE 'Station' IN labels(n) | n.id] AS
      hops
  WHERE ALL(e IN rels WHERE
    e.user_id = r.user_id AND e.val_time > r.val_time AND
    (e.duration IS NULL OR e.duration < 20) )
  EMIT r.user_id, s.id, r.val_time, hops
  ON ENTERING
  EVERY PT5M
}
```

Listing 5: Continuously retrieve users that use the free period for two subsequent rentals in the last hour using Seraph.

r.user_id	s.id	r.val_time	hops	win_start	win_end
1234	1	14:40	[2,3]	14:15	15:15

Table 5: Outputs of Seraph continuous query at 15:15h.

r.user_id	s.id	r.val_time	hops	win_start	win_end
5678	2	14:58	[3,4]	14:40	15:40

Table 6: Outputs of Seraph continuous query at 15:40h.

Listing 1. Since the desired window behavior is now natively supported by Seraph, we directly start by defining the desired pattern.

The `MATCH` clause defines the pattern π we are looking for (section 5.4 to 4). Note that compared to the previous Cypher query, the predicate applying the edge filtering for the window is obsolete. By `WITHIN` we define the width of the window for this pattern, which is 1 hour (`PT1H`). The predicates of the `WHERE` clause are equal to the ones of the Cypher query. Instead of the `RETURN` clause we use the `EMIT` clause to get a continuous stream of time-annotated tables and define the projected attributes for the resulting tuples enhanced with bounds of the current window that is built by Seraph (`win_start` and `win_end`). At section 5.4 the `ON ENTERING` operator allows for emitting only new matches entering the window, which satisfies requirement **R3**. The `EVERY` operator specifies the frequency of the evaluation process. Here, we define it as 5 minutes specified by `PT5M`. The operators `STARTING AT`, `WITHIN` and `EVERY` build the continuous evaluation and thus satisfy requirement **R2**. As Seraph only expands Cypher, we preserve the expressiveness and hence meet requirement **R4**.

To summarize, every 5 minutes starting from 14:45h, the system evaluates a pattern on the active snapshot graph defined by a 1h window and emits a stream of time-annotated tables, including the users that extend their rental time by using subsequent free rentals.

Let us analyze the output of the query at different time instants.

14:45h The 1h window covers only the outer left graph depicted in Figure 1. Just a bike was rented, which needs no notification.

15:00h The two left graphs in Figure 1 are merged. User `1234` returned a bike and rented one again. User `5678` rented a bike, too. However, the resulting snapshot graph is queried without any match.

15:15h The three left graphs in Figure 1 are in the active substream and thus merged to a snapshot graph, which leads to a match: user `1234` applying the trick. Since 15:15h is an evaluation time, the time-annotated table (Table 5) is emitted.

15:20h The fourth graph arrived with the information that user `5678` returned and rented again a bike.

15:40h All graphs of Figure 1 are in the active substream and thus unified to a snapshot graph (cf. Figure 2). Another match is found: user `5678` is applying the trick, too. The query's output at 15:40h is depicted in Table 6. Since we used `ON ENTERING`, just the new match, i.e., user `5678`, is part of the resulting time-annotated table.

6 IMPLEMENTATION

Since this paper provides the formal description of Seraph for paving the road to future implementations, we briefly discuss our plans in such a direction.

Graph Stream Processing (GSP) Engine. We built an proof of concept implementation [35] of a GSP engine with Seraph language support. It is open-source available under Apache-2.0 license and based on Neo4j and RSP4J [45], a library for fast-prototyping stream processing engines. A query parser [15] based on ANTLR is also available to validate the syntax design. Notably, this first POC has the goal of empirically proving Seraph's feasibility and enabling various tests and evaluations. In the short term, we also plan to test other Cypher-compatible embedded graph engines like Kuzu [26] or Memgraph [30].

Optimizations. We plan a first round of optimization focusing on query planning at different levels, including native operators and efficient window maintenance. We also plan to explore the adoption of advanced windowing as described in the recent survey [50], as well as optimizations regarding concurrent queries and avoidable re-executions on equal window contents.

Distributed GSP. In the medium term, we plan to explore a distributed implementation of Seraph based on a stream processing framework such as Apache Flink [14] or Apache Spark [43]. Here we can benefit from previous work on Gradoop [27, 38], a distributed temporal property graph analysis framework with integrated Cypher-based pattern matching based on Apache Flink.

Optimizations. A second round of optimization will focus on system-level investigation as in [24]. In particular, we will explore operator placement and fusions, graph stream partitioning and distributed join algorithms.

7 RELATED WORK

This section positions Seraph in the literature and discuss its relation with other work on dynamic graphs and stream processing.

Temporal, Dynamic and Streaming Graphs are attempts to extend existing graph data models with temporal dimension(s) [10].

Temporal graphs are graphs that maintain the history of its evolution [37, 38]. They enable temporal analysis, algorithms and querying the graph at the current or a past state in a batch-processing way. For this adapted declarative languages exist [17, 36, 38] but despite the possibility of historical path queries, they do not provide continuous querying of graph streams like Seraph.

Dynamic Graphs are graphs whose elements are unpredictably

updated by insertions and deletions [10]. The data system that manages the dynamic graph either stores the most recent version of the graph or the graph's entire change history. Differently from Seraph, languages for querying dynamic graphs do not necessarily require a continuous semantics, but scalable and time-sensitive query answering in presence of changes is one of the key research focuses.

Streaming Graphs (also called *Graph Streams*) are dynamic graphs that grow indefinitely [2] and, as for Seraph, query answering must take unboundedness into account. Moreover, the data management system is assumed to be unable to store the whole graph state, therefore it focuses on the finite sub-graph that is relevant for the query answering. Pacaci et al. introduced in [34] a data model and a query evaluation algebra on streaming graphs including semantics of persistent regular path queries (RPQ). Their considered stream consists of single relationships not graphs as in Seraph. Our work can be seen as a complement of their work on streaming complex graph queries in a landscape where no declarative and industry-ready language exist.

Sakr et al. show in [39] that there is a current need for systems that can model and process both dynamic graphs and graph streams. An essential research challenge is the investigation of graph query operators for path-oriented semantics on graph streams. These are necessary for standardized graph languages like GQL. With Seraph and its query model and semantics, we address exactly this need. Kankanamge et al. present with Graphflow [28] a prototype in-memory graph database supporting continuous subgraph queries. Unlike Seraph, the queries cannot be evaluated on property graph streams and windowed queries are not supported.

Declarative Stream Processing Languages have been around for two decades. Most of the existing solutions, including those associated with the Big Data initiative [23], present an SQL-like syntax [47], e.g. Streaming SQL [9], and build upon the Continuous Query Language model (CQL) [13]. CQL prescribes making the management of (relational) streams orthogonal to the management of relations. They defined three operator families, namely Stream-to-Relation, Relation-to-Relation and Relation-to-Stream, which formalize the interoperability between relations and streams.

Seraph follows CQL orthogonalisation principle because it makes the language compositional and maintainable [16]. However, differently from CQL, Seraph works on a stream of property graphs and provides the primitives to fully control the reporting. Learning from Dindar et al. [21], who showed how the operational semantics of stream processing engines is often uncontrollable by the user, Seraph's `EMIT` ... `EVERY` ... clauses give an end-to-end view of what impacts execution semantics from inputs to output. Hence, Seraph users have full control on the query execution semantics.

Finally, RSP-QL was proposed by the Semantic Web community in the late 2000s' to accommodate the need for processing heterogeneous data streams. Seraph is similar to RSP-QL [19], which in turn extends CQL work on RDF streams. The main differences between them are the data model, i.e., Seraph adopts streams of property graphs; and the query model: Seraph temporal approach is maintained after windowing.

8 CONCLUSIONS AND FUTURE WORK

This paper presented Seraph, a declarative graph query language that compositionally enriches Cypher for dealing with streams of property graphs and continuous query answering. In particular, it shows that Seraph is designed to overcome Cypher limitations for continuous processing: i) Seraph's data model can represent streams of property graphs; ii) Seraph's query model enables continuous evaluation on top of Cypher semantics by creating snapshot graphs from the graph stream using windowing and evaluating the query under snapshot reducibility. Moreover, iii) we showed Seraph's features in three industrial use cases: network monitoring, real-time tracing and bike sharing (our running example). The formal foundations we lay in this paper will pave the road for future continuous graph query languages, such as a continuous extension to GQL, the ISO standard graph query language, whose first version is expected to appear in 2024.

In addition to the implementation plans, we will explore i) how to query multiple streams simultaneously, ii) partition a property graph stream in logical substreams, and iii) incorporate static graph data within the continuous computation. Finally, we plan to vi) focus on graph-to-graph transformations as in GQL.

Finally, the current limitations of Seraph are in the support of features of the language used. For example, Cypher does not currently fully support conjunctive regular path queries (CRPQs), which means that CRPQs are not supported in Seraph either. A more expressive language, such as a future Cypher version or GQL with temporal extensions, would overcome such limitations.

ACKNOWLEDGMENTS

C. Rost and E. Rahm acknowledge the Federal Ministry of Education and Research of Germany's financial support and the Sächsische Staatsministerium für Wissenschaft, Kultur und Tourismus for ScaDS.AI.

R. Tommasini is supported by the French Research Agency under grant agreement nr. ANR-22-CE23-0001 Polyflow.

A. Bonifati is supported by the French Research Agency under grant agreement nr. ANR-22-CE92-0025 HyGraph. The ideas presented in this paper originated and developed in the context of the Dagstuhl seminar 19491 on Big Graph Processing systems [11].

REFERENCES

[1] Lorenzo Affetti, Riccardo Tommasini, Alessandro Margara, Gianpaolo Cugola, and Emanuele Della Valle. 2017. Defining the execution semantics of stream processing engines. *J. Big Data* (2017). https://doi.org/10.1186/s40537-017-0072-9

[2] Charu C. Aggarwal. 2018. Extracting Real-Time Insights from Graphs and Social Streams. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval, SIGIR 2018, Ann Arbor, MI, USA, July 08-12, 2018*, Kevyn Collins-Thompson, Qiaozhu Mei, Brian D. Davison, Yiqun Liu, and Emine Yilmaz (Eds.). ACM, 1339. https://doi.org/10.1145/3209978.3210212

[3] Renzo Angles. 2018. The Property Graph Database Model. In *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018 (CEUR Workshop Proceedings)*, Dan Olteanu and Barbara Poblete (Eds.), Vol. 2100. CEUR-WS.org. https://ceur-ws.org/Vol-2100/paper26.pdf

[4] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. 2018. G-CORE: A Core for Future Graph Query Languages. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1421–1432. https://doi.org/10.1145/3183713.3190654

[5] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen Savkovic, Michael Schmidt, Juan Sequeda, Slawek Staworko, Dominik Tomaszuk, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Dusan Zivkovic. 2023. PG-Schema: Schemas for Property Graphs. *Proc. ACM Manag. Data* 1, 2 (2023), 198:1–198:25. https://doi.org/10.1145/3589778

[6] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *VLDB J.* 15, 2 (2006), 121–142. https://doi.org/10.1007/s00778-004-0147-z

[7] Shivnath Babu and Jennifer Widom. 2001. Continuous Queries over Data Streams. *SIGMOD Rec.* 30, 3 (2001), 109–120. https://doi.org/10.1145/603867.

603884

[8] Daniel Barbará. 1999. The Characterization of Continuous Queries. *Int. J. Cooperative Inf. Syst.* 8, 4 (1999), 295. https://doi.org/10.1142/S0218843099000150

[9] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth L. Knowles. 2019. One SQL to Rule Them All - an Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1757–1772. https://doi.org/10.1145/3299869.3314040

[10] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. 2023. Practice of Streaming Processing of Dynamic Graphs: Concepts, Models, and Systems. *IEEE Trans. Parallel Distributed Syst.* 34, 6 (2023), 1860–1876. https://doi.org/10.1109/TPDS.2021.3131677

[11] Angela Bonifati, Alexandru Iosup, Sherif Sakr, and Hannes Voigt. 2020. Big Graph Processing Systems (Dagstuhl Seminar 19491). *Dagstuhl Reports* 9, 12 (2020), 1–27. https://doi.org/10.4230/DagRep.9.12.1

[12] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura M. Haas, Renée J. Miller, and Nesime Tatbul. 2010. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. *Proc. VLDB Endow.* 3, 1 (2010), 232–243. https://doi.org/10.14778/1920841.1920874

[13] Antal Buza. 2006. Extension of CQL over Dynamic Databases. *J. UCS* 12, 9 (2006), 1165–1176. https://doi.org/10.3217/jucs-012-09-1165

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

[15] Christopher Rost and Riccardo Tommassini and Maximilian Zimmer and Julian Pielmaier. 2023. *Seraph Query Parser.* Retrieved August 15, 2023 from https://github.com/dbs-leipzig/seraph-parser

[16] C. J. Date. 1984. Some Principles of Good Language Design (with especial reference to the design of database languages). *SIGMOD Rec.* 14, 3 (1984), 1–7. https://doi.org/10.1145/984549.984550

[17] Ariel Debrouvier, Eliseo Parodi, Matías Perazzo, Valeria Soliani, and Alejandro A. Vaisman. 2021. A model and query language for temporal graph databases. *VLDB J.* 30, 5 (2021), 825–858. https://doi.org/10.1007/s00778-021-00675-4

[18] Daniele Dell'Aglio, Jean-Paul Calbimonte, Marco Balduini, Óscar Corcho, and Emanuele Della Valle. 2013. On Correctness in RDF Stream Processor Benchmarking. In *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II (Lecture Notes in Computer Science)*, Harith Alani, Lalana Kagal, Achille Fokoue, Paul Groth, Chris Biemann, Josiane Xavier Parreira, Lora Aroyo, Natasha F. Noy, Chris Welty, and Krzysztof Janowicz (Eds.), Vol. 8219. Springer, 326–342. https://doi.org/10.1007/978-3-642-41338-4_21

[19] Daniele Dell'Aglio, Emanuele Della Valle, Jean-Paul Calbimonte, and Óscar Corcho. 2014. RSP-QL Semantics: A Unifying Query Model to Explain Heterogeneity of RDF Stream Processing Systems. *Int. J. Semantic Web Inf. Syst.* 10, 4 (2014), 17–44. https://doi.org/10.4018/ijswis.2014100102

[20] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 2246–2258. https://doi.org/10.1145/3514221.3526057

[21] Nihal Dindar, Nesime Tatbul, Renée J. Miller, Laura M. Haas, and Irina Botan. 2013. Modeling the execution semantics of stream processing engines with SECRET. *VLDB J.* 22, 4 (2013), 421–446. https://doi.org/10.1007/s00778-012-0297-3

[22] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1433–1445. https://doi.org/10.1145/3183713.3190657

[23] Martin Hirzel, Guillaume Baudart, Angela Bonifati, Emanuele Della Valle, Sherif Sakr, and Akrivi Vlachou. 2018. Stream Processing Languages in the Big Data Era. *SIGMOD Rec.* 47, 2 (2018), 29–40. https://doi.org/10.1145/3299887.3299892

[24] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv.* 46, 4, Article 46 (mar 2014), 34 pages. https://doi.org/10.1145/2528412

[25] ISO Central Secretary. [n.d.]. *Information Technology - Database Languages - GQL.* Standard ISO/IEC WD 39075. International Organization for Standardization, Geneva, CH. https://www.iso.org/standard/76120.html

[26] Guodong Jin, Xiyang Feng, Ziyi Chen, Chang Liu, and Semih Salihoglu. 2023. KÙZU Graph Database Management System. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p48-jin.pdf

[27] Martin Junghanns, Max Kießling, Alex Averbuch, André Petermann, and Erhard Rahm. 2017. Cypher-based Graph Pattern Matching in Gradoop. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017*, Peter A. Boncz and Josep Lluís Larriba-Pey (Eds.). ACM, 3:1–3:8. https://doi.org/10.1145/3078447.3078450

[28] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedhbi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 1695–1698. https://doi.org/10.1145/3035918.3056445

[29] Jürgen Krämer and Bernhard Seeger. 2009. Semantics and implementation of continuous sliding window queries over data streams. *ACM Trans. Database Syst.* 34, 1 (2009), 4:1–4:49. https://doi.org/10.1145/1508857.1508861

[30] Memgraph Ltd. 2023. *MemGraph - Open Source Graph Database.* Retrieved August 15, 2023 from https://memgraph.com

[31] Vera Zaychik Moffitt and Julia Stoyanovich. 2017. Temporal graph algebra. In *Proceedings of The 16th International Symposium on Database Programming Languages, DBPL 2017, Munich, Germany, September 1, 2017*, Tiark Rompf and Alexander Alexandrov (Eds.). ACM, 10:1–10:12. https://doi.org/10.1145/3122831.3122838

[32] Neo4j. 2023. *Kafka Connect Neo4j Connector User Guide.* Retrieved August 15, 2023 from https://neo4j.com/docs/kafka/

[33] M. Tamer Özsu and Patrick Valduriez. 2020. *Principles of Distributed Database Systems, 4th Edition.* Springer. https://doi.org/10.1007/978-3-030-26253-2

[34] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. 2022. Evaluating Complex Queries on Streaming Graphs. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022.* IEEE, 272–285. https://doi.org/10.1109/ICDE53745.2022.00025

[35] Riccardo Tommasini and Christopher Rost and Julian Pielmaier and Christopher Lausch. 2023. *Seraph RSP4J implementation GitHub.* Retrieved August 15, 2023 from https://github.com/dbs-leipzig/seraph-rsp4j-impl

[36] Christopher Rost, Philip Fritzsche, Lucas Schons, Maximilian Zimmer, Dieter Gawlick, and Erhard Rahm. 2021. Bitemporal Property Graphs to Organize Evolving Systems. *arXiv preprint arXiv:2111.13499* (2021).

[37] Christopher Rost, Kevin Gomez, Peter Christen, and Erhard Rahm. 2023. Evolution of Degree Metrics in Large Temporal Graphs. In *Datenbanksysteme für Business, Technologie und Web (BTW 2023) (LNI)*, Vol. P-331. Gesellschaft für Informatik e.V., 485–507. https://doi.org/10.18420/BTW2023-23

[38] Christopher Rost, Kevin Gómez, Matthias Täschner, Philip Fritzsche, Lucas Schons, Lukas Christ, Timo Adameit, Martin Junghanns, and Erhard Rahm. 2022. Distributed temporal graph analytics with GRADOOP. *VLDB J.* 31, 2 (2022), 375–401. https://doi.org/10.1007/s00778-021-00667-4

[39] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid G. 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 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. 2021. The future is big graphs: a community view on graph processing systems. *Commun. ACM* 64, 9 (2021), 62–71. https://doi.org/10.1145/3434642

[40] Michael Stonebraker, Ugur Çetintemel, and Stanley B. Zdonik. 2005. The 8 requirements of real-time stream processing. *SIGMOD Rec.* 34, 4 (2005), 42–47. https://doi.org/10.1145/1107499.1107504

[41] Jiao Tao, Evren Sirin, Jie Bao, and Deborah L. McGuinness. 2010. Integrity Constraints in OWL. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, Maria Fox and David Poole (Eds.). AAAI Press, 1443–1448. https://doi.org/10.1609/aaai.v24i1.7525

[42] Douglas B. Terry, David Goldberg, David A. Nichols, and Brian M. Oki. 1992. Continuous Queries over Append-Only Databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 2-5, 1992*, Michael Stonebraker (Ed.). ACM Press, 321–330. https://doi.org/10.1145/130283.130333

[43] The Apache Software Foundation. 2023. *Apache Spark: Unified engine for large-scale data analytics.* Retrieved August 15, 2023 from https://spark.apache.org

[44] Neo4j Inc. Tom Geudens. 2022. *POLE Investigations.* Retrieved August 15, 2023 from https://www.slideshare.net/neo4j/pole-investigations

[45] Riccardo Tommasini, Pieter Bonte, Femke Ongenae, and Emanuele Della Valle. 2021. RSP4J: An API for RDF Stream Processing. In *The Semantic Web - 18th International Conference, ESWC 2021, Virtual Event, June 6-10, 2021, Proceedings (Lecture Notes in Computer Science)*, Ruben Verborgh, Katja Hose, Heiko Paulheim, Pierre-Antoine Champin, Maria Maleshkova, Óscar Corcho, Petar Ristoski, and Mehwish Alam (Eds.), Vol. 12731. Springer, 565–581. https://doi.org/10.1007/978-3-030-77385-4_34

[46] Riccardo Tommasini, Pieter Bonte, Fabiano Spiga, and Emanuele Della Valle. 2023. *Streaming linked data : from vision to practice.* Springer. XVI, 158 pages. https://doi.org/10.1007/978-3-031-15371-6

[47] Riccardo Tommasini, Sherif Sakr, Emanuele Della Valle, and Hojjat Jafarpour. 2020. Declarative Languages for Big Streaming Data. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, Angela Bonifati, Yongluan

Zhou, Marcos Antonio Vaz Salles, Alexander Böhm, Dan Olteanu, George H. L. Fletcher, Arijit Khan, and Bin Yang (Eds.). OpenProceedings.org, 643–646. https://doi.org/10.5441/002/edbt.2020.84

[48] Jonas Traub, Philipp Marian Grulich, Alejandro Rodriguez Cuellar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2021. Scotty: General and Efficient Open-source Window Aggregation for Stream Processing Systems. *ACM Trans. Database Syst.* 46, 1 (2021), 1:1–1:46. https://doi.org/10.1145/3433675

[49] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, Redwood Shores, CA, USA, June 24 - 24, 2016*, Peter A. Boncz and Josep-Lluís Larriba-Pey (Eds.). ACM, 7. https://doi.org/10.1145/2960414.2960421

[50] Juliane Verwiebe, Philipp M. Grulich, Jonas Traub, and Volker Markl. 2023. Survey of window types for aggregation in stream processing systems. *VLDB J.* 32, 5 (2023), 985–1011. https://doi.org/10.1007/s00778-022-00778-6