Temporal Regular Path Queries

Marcelo Arenas, Pedro Bahamondes
Universidad Católica & IMFD, Chile
marenas@ing.puc.cl, pibahamondes@uc.cl

Amir Aghasadeghi, Julia Stoyanovich
New York University, USA
amirpouya@nyu.edu, stoyanovich@nyu.edu

***Abstract*—In the last decade, substantial progress has been made towards standardizing the syntax of graph query languages, and towards understanding their semantics and complexity of evaluation. In this paper, we consider temporal property graphs (TPGs) and propose temporal regular path queries (TRPQs) that incorporate time into TPG navigation. Starting with design principles, we propose a natural syntactic extension of the MATCH clause of popular graph query languages. We then formally present the semantics of TRPQs, and study the complexity of their evaluation. We show that TRPQs can be evaluated in polynomial time if TPGs are time-stamped with time points, and identify fragments of the TRPQ language that admit efficient evaluation over a more succinct interval-annotated representation. Finally, we implement a fragment of the language in a state-of-the-art dataflow framework, and experimentally demonstrate that TRPQ can be evaluated efficiently.**

***Index Terms*—graph query languages, temporal query languages**

I. Introduction

The importance of networks in scientific and commercial domains is undeniable. Networks are represented by graphs, and we will use the terms *network* and *graph* interchangeably. Considerable research and engineering effort is devoted to the development of effective and efficient graph representations and query languages. Property graphs have emerged as the de facto standard, and have been studied extensively, with efforts underway to unify the semantics of query languages for these graphs [1], [2]. Many interesting questions about graphs are related to their evolution rather than to their static state [3]–[11]. Consequently, several recent proposals seek to extend query languages for property graphs with time [12]–[16].

Our focus in this paper is on incorporating time into path queries. More precisely, we (a) outline the design principles for a temporal extension of Regular Path Queries (RPQs) with time; (b) propose a natural syntactic extension of state of the art query languages for conventional (non-temporal) property graphs, which supports temporal RPQs (TRPQs); (c) formally present the semantics of this language; (d) study the complexity of evaluation of several variants of this language; (e) implement a practical fragment of this language in a dataflow framework; and (f) empirically demonstrate that TRPQs can be evaluated efficiently. We show that, by adhering to the design principles that draw on decades of work on graph databases and on temporal relational databases, we are able to achieve polynomial-time complexity of evaluation, paving the way to implementations that are both usable and practical, as supported by our implementation and experiments.

A. Running example

As a preview of our proposed methods, consider Figure 1 that depicts a contact tracing network for a communicable disease with airborne transmission between people in enclosed locations on a university campus. In this network, different actors and their interactions are presented as a *temporal property graph* or TPG for short. (We will define temporal property graphs formally in Section III).

As in conventional property graphs [1], nodes and edges in a TPG are labeled. The graph in Figure 1 contains two types of nodes, `Person` and `Room` (representing a classroom), and three types of edges: bi-directional edges `meets` and `cohabits` (lives together), and directed edge `visits`. Nodes and edges have optional properties that are associated with values. For example, node n_1 of type `Person` has properties `name` with value `'Ann'` and `risk` with value `'low'`. As another example, edge e_2 of type `meets` has property `loc` with value `'park'`.

The purpose of the graph in Figure 1 is to allow identification of individuals who may have been exposed to the disease. In particular, we are interested in identifying potentially infected individuals who are considered high risk, due to age or pre-existing conditions. These types of questions can be naturally phrased as *temporal regular path queries* (TRPQs) that interrogate reachability over time. We will give an example of a TRPQ momentarily.

To support TRPQs, all nodes and edges in a TPG are associated with *time intervals of validity* (or *intervals* for short) that represent consecutive time points during which no change occurred for a node or an edge, in terms of its existence or property values. For example, node n_1 (Ann) is associated with the interval `[1, 9]`, signifying that n_1 was present in the graph and took on the specified property values during 9 consecutive time points. As another example, node n_2 (Bob) exists during the same interval as n_1, but undergoes a change in the value of the property `risk` at time 4, when it changes from `'low'` to `'high'`. We represent a change in the state of an entity (a node or an edge) with nested boxes inside an outer box that denotes the entity in Figure 1.

Now, consider an example of a TRPQ that extends the syntax of Cypher to retrieve the list of high-risk people (`x`) who met someone (`y`), who subsequently tested positive for an infectious disease:

```
MATCH (x:Person {risk = 'high'})-
  /FWD/:meets/FWD/NEXT*/-(y:Person {test = 'pos'})
ON contact_tracing
```

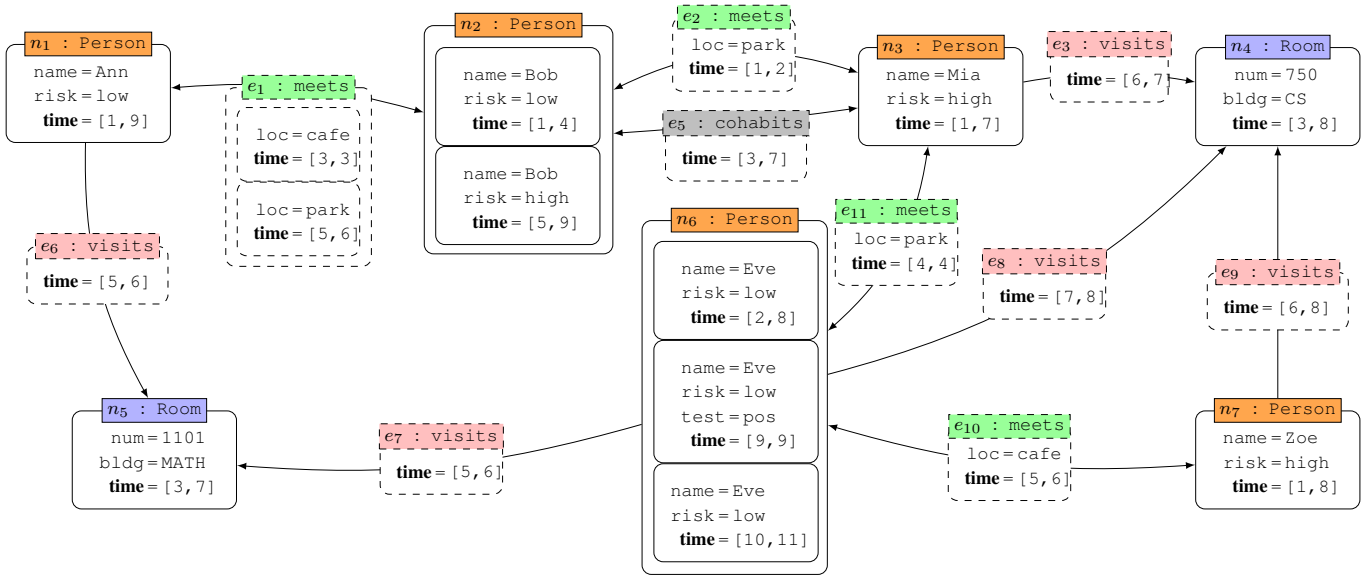

Fig. 1. A TPG used for contact tracing. The graph contains two types of nodes, `Person` and `Room`, and three types of edges: bi-directional edges `meets` and `cohabits`, and directed edge `visits`. `Person` nodes have properties `name`, `risk` (`'high'` or `'low'`) of complications, and `test` (`'pos'` or `'neg'`) of disease status. Eve (node n_6) tested positive for a communicable disease at time 9.

This contact tracing query produces the following *temporal binding table* when evaluated over the TPG in Figure 1:

x	x_time	y	y_time
n_7	5	n_6	9
n_7	6	n_6	9
n_3	4	n_6	9

B. Summary of our approach

In the remainder of this paper, we formally develop the concepts that are necessary to evaluate this and other useful TRPQs over TPGs. We adopt a conceptual TPG model that naturally extends property graphs with time, and is both simple and sufficiently flexible to support the evolution of graph topology and of the properties of its nodes and edges. We evaluate TRPQs on TPGs under *point-based semantics* [17], in which operators adhere to two principles: snapshot reducibility and extended snapshot reducibility, discussed in Section II. Our conceptual TPG model admits two logical representations that differ in the kind of time-stamping they use [18]. One associates objects with time points, while the other associates them with time intervals, for a more compact representation.

Design principles. We carefully designed our TRPQ language based on the following principles:

Navigability: Include operators that refer to the dynamics of navigating through the TPG: temporal navigation refers to movements on the graph over time, and structural navigation refers to movements across locations in its topology.

Navigation orthogonality: Temporal and structural navigation operators must be orthogonal, allowing non-simultaneous single-step temporal and structural movement.

Node-edge symmetry: The language should treat nodes and edges as first-class citizens, supporting equivalent operations.

Static testability: Testing is independent of navigation.

Snapshot reducibility: When time is removed from a query, pairs of temporal objects satisfying the query should correspond to pairs of objects in a single snapshot of the graph, and every pair satisfying the query in the snapshot of the TPG should correspond to a path satisfying it in the TPG.

By adhering to these principles, we achieved polynomial-time complexity of evaluation for TPGs that are time-stamped with time points, and also identified a significant fragment of the language that can be efficiently evaluated for interval time-stamped TPGs. In addition to theoretical results, these principles also allowed us to efficiently implement TRPQs by decoupling non-temporal and temporal processing.

Paper organization: We first give some background on temporal graph models and path query languages in Section II. We then formally define temporal graphs in Section III. We go on to propose a syntax for adding time to a practical graph query language in Section IV. Next, in Section V, we give the precise syntax and semantics of the language, and study the complexity of evaluating it. We describe an implementation of our language over an interval-based TPG in Section VI, and present results of an experimental evaluation in Section VII. We conclude in Section VIII. Additional complexity results and proofs, and supplementary experiments are available in technical report [19]. System implementation and experimental evaluation are available at https://github.com/amirpouya/tpath.

II. Background and Related Work

Substantial research has been undertaken in the area of *temporal relational databases* since the 1980s, producing a significant body of work [20], which includes representation of time [21]–[23], semantics of temporal models [24], temporal

algebras [25], and access methods [26]. Results of some of this work are part of the SQL:2011 standard [27].

a) Temporal graph models: Temporal graph models differ in what temporal semantics they encode, what time representation they use (time point, interval, or implicitly with a sequence), what entities they time-stamp (graphs, nodes, edges, or attribute-value assignments), and whether they represent evolution of topology only or also of the attributes. With a few exceptions, discussed next, the current de facto standard representation of temporal graphs is the *snapshot sequence*, where a state of a graph is associated with either a time point or an interval during which the graph was in that state [28]–[38]. This representation supports operations within each snapshot under the principle of *snapshot reducibility*, namely, that applying a temporal operator to a database is equivalent to applying the non-temporal variant of the operator to each database state [17]. For example, the G* system [15] stores a temporal graph as a snapshot sequence and provides two query languages, the procedural PGQL and the declarative DGQL. PGQL includes operators such as retrieving graph vertices and their edges at a given time point, along with non-graph operators like aggregation, union, projection, and join. Neither PGQL nor DGQL support temporal path queries.

The fundamental disadvantage of using the snapshot sequence as the conceptual representation of a temporal graph is that it does not support operations that explicitly reference temporal information. Semantics of operations that make explicit references to time are formalized as the principle of *extended snapshot reducibility*, where timestamps are made available to operators by propagating time as data [17]. Considering that our goal in this work is to support temporal regular path queries, having access to temporal information during navigation is crucial.

In response to this important limitation of the snapshot sequence representation, proposals have been made to annotate graph nodes, edges, or attributes with time. Moffitt and Stoyanovich [16] proposed to model property graph evolution by associating intervals of validity with nodes, edges, and property values. They also developed a compositional temporal graph algebra that provides a temporal generalization of common graph operations including subgraph, node creation, union, and join, but does not include reachability or path constructs. In our work, we adopt a similar representation of temporal graphs, but focus on temporal regular path queries.

b) Paths in temporal graphs: Specific kinds of path queries over temporal graphs have been considered in the literature. Wu et al. [39]–[41] studied path query variants over temporal graphs, in which nodes are time-invariant and edges are associated with a starting time and an ending time. (Nodes and edges do not have type labels or attributes.) The authors introduced four types of "minimum temporal path" queries, including the earliest-arriving path and the fastest path, which can be seen as generalizations of the shortest path query for temporal graphs. They proposed algorithms and indexing methods to process minimum temporal path and temporal reachability queries efficiently.

Byun et al. [12] introduced ChronoGraph, a temporal graph traversal system in which edges are traversed time-forward. The authors show three use cases: temporal breadth-first search, temporal depth-first search, and temporal single-source shortest-path, instantiated over Apache Tinkerpop. Johnson et al. [14] introduced Nepal, a query language that has SQL-like syntax and supports regular path queries over temporal multi-layer communication networks, represented by temporal graphs that associate a sequence of intervals of validity with each node and edge. The key novelty of this work are time-travel path queries to retrieve past network states. Finally, Debrouvier et al. [13] introduced T-GQL, a query language for TPGs with Cypher-like syntax [42]. T-GQL operates over graphs in which (a) nodes persists but their attributes (with values) can change over time, and so are associated with periods of validity; and (b) edges are associated with periods of validity but their attributes are time-invariant. This asymmetry in the handling of nodes and edges is due to the authors' commitment to a specific (lower-level) representation of such TPGs in a conventional property graph system. Specifically, they assume that Objects (representing nodes), Attributes, and Values are stored as conventional property graph nodes, whereas time intervals are stored as properties of these nodes. Temporal edges are, in turn, stored as conventional edges, with time interval as one of their properties. T-GQL supports three types of path queries over such graphs, syntactically specified with the help of named functions: (1) "Continuous path" queries retrieve paths valid during each time point—snapshot semantics. (2) "Pairwise continuous paths" require that the incoming and the outgoing edge for a node being traversed must exist during some overlapping time period. (3) "Consecutive paths" encode temporal journeys; for example, to indicate a way to fly from Tokyo to Buenos Aires with a couple of stopovers in a temporal graph for flight scheduling. Consecutive paths are used in T-GQL for encoding earliest arrival, latest departure, fastest, and shortest path queries.

A more detailed comparison of our proposal with other temporal query languages is given in Section V-C. In summary, our proposal differs from prior work in that we develop a general-purpose query language for temporal paths, which works over a simple conceptual definition of temporal property graphs and is nonetheless general enough to represent different kinds of temporal and structural evolution of such graphs. Our language is syntactically simple: it directly, and minimally, extends the MATCH clause of popular graph query languages, and does not rely on custom functions. In fact, as we show in Section V-B, there is a simple way to define its formal semantics, which allows us to develop efficient algorithms for query evaluation.

III. A Temporal Graph Model

In this section, we formalize the notion of temporal property graph, which extends the widely used notion of property graph [1], [2], [42] to include explicit access to time. In this way, we can model the evolution of the topology of such a graph, as well as the changes in node and edge properties.

A temporal property graph defines a point-based representation of the evolution of a property graph, which is a simple and suitable framework to represent and reason about this evolution. However, time-stamping objects with time points may be impractical in terms of space overhead. This motivates the development of interval-based representations, which are common for temporal models for both relations (e.g., [18], [43]) and graphs (e.g., [12], [13], [16]). In this section, we also define a succinct representation of temporal property graphs that uses interval time-stamping. Notice that point-based temporal semantics requires this succinct representation to be temporally coalesced: a pair of value-equivalent temporally adjacent intervals should be stored as a single interval, and this property should be maintained through operations [44].

A. Temporal property graphs

Assume *Lab*, *Prop* and *Val* to be sets of label names, property names and actual values, respectively. We define temporal property graphs over finite sets of time points. Time points can take on values that correspond to the units of time as appropriate for the application domain, and may represent seconds, weeks, or years. For the sake of presentation, we represent the universe of time points by $\mathbb{N}$: a temporal domain Ω is a finite set of consecutive natural numbers, that is, $\Omega = \{i \in \mathbb{N} \mid a \leq i \leq b\}$ for some $a, b \in \mathbb{N}$ such that $a \leq b$.

Definition III.1. *A temporal property graph (TPG) is a tuple* $G = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$*, where*

- Ω *is a temporal domain;* N *is a finite set of* nodes*,* E *is a finite set of* edges*, and* $V \cap E = \emptyset$*;*
- $\rho : E \to (N \times N)$ *is a function that maps an edge to its source and destination nodes;*
- $\lambda : (N \cup E) \to$ *Lab is a function that maps a node or an edge to its label;*
- $\xi : (N \cup E) \times \Omega \to \{true, false\}$ *is a function that maps a node or an edge, and a time point to a Boolean. Moreover, if* $\xi(e, t) = true$ *and* $\rho(e) = (v_1, v_2)$*, then* $\xi(v_1, t) = true$ *and* $\xi(v_2, t) = true$*.*
- $\sigma : (N \cup E) \times Prop \times \Omega \to Val$ *is a partial function that maps a node or an edge, a property name, and a time point to a value. Moreover, there exists a finite number of triples* $(o, p, t) \in (N \cup E) \times Prop \times \Omega$ *such that* $\sigma(o, p, t)$ *is defined, and if* $\sigma(o, p, t)$ *is defined, then* $\xi(o, t) = true$*.*

Observe that Ω in Definition III.1 denotes the *temporal domain* of G, a finite set of linearly ordered time points starting from the time associated with the earliest *snapshot* of G, and ending with the time associated with its latest snapshot, where a snapshot of G refers to a conventional (non-temporal) property graph that represents the state of G at a given time point. Function ρ in Definition III.1 is used to provide the starting and ending nodes of an edge, function λ provides the label of a node or an edge, and function ξ indicates whether a node or an edge exists at a given time point in Ω (which corresponds to *true*). Finally, function σ indicates the value of a property for a node or an edge at a given time point in Ω.

Two conditions are imposed on TPGs to enforce that they conceptually correspond to sequences of valid conventional property graphs. In particular, an edge can only exist at a time when both of the nodes it connects exist, and that a property can only take on a value at a time when the corresponding object exists. Moreover, observe that by imposing that $\sigma(o, p, t)$ be defined for a *finite* number of triples (o, p, t), we are ensuring that each node or edge can have values for a finite number of properties, so that each TPG has a finite representation. Finally, Definition III.1 assumes, for simplicity, that property values are drawn from the infinite set *Val*. That is, we do not distinguish between different data types. If a distinction is necessary, then *Val* can be replaced by a domain of values of some k different data types, $Val_1, \ldots, Val_k$.

Recall our running example discussed in Section I-A and shown in Figure 1. This example illustrates Definition III.1; it shows a TPG used for contact tracing for a communicable disease, with airborne transmission between people (represented by nodes with label `Person`) in enclosed locations (*e.g.,* nodes with label `Room`). This TPG has a temporal domain $\Omega = \{1, \ldots, 11\}$, although any set of consecutive natural numbers containing Ω can serve as the temporal domain of this TPG, for example the set $\{0, \ldots, 15\}$. The TPG is a multigraph: n_2 and n_3 are connected by two edges, e_2 and e_5.

In the TPG in Figure 1, `Person` nodes have properties name, risk (`'low'` or `'high'`), and test (`'pos'` or `'neg'`). For example, Eve, represented by node n_6, is known to have tested positive for the disease at time 9. Note that each node and edge refers to a specific time-invariant real-life object or event. A TPG records observed states of these objects. In fact, real-life objects correspond to a sequence of temporal objects, each with a set of properties. For instance, node n_2 corresponds to a sequence of 9 temporal objects, one for each time point 1 through 9. These are represented in the figure by two boxes inside the outer box for n_2, one for each interval during which no change occurred: $[1, 4]$ with name Bob, and low risk, and $[5, 9]$ with name Bob, and high risk. To simplify the figure, we do not show internal boxes for nodes or edges associated with a single time interval, such as n_1 and e_6.

B. Interval-timestamped temporal property graphs

An interval of $\mathbb{N}$ is a term of the form $[a, b]$ with $a, b \in \mathbb{N}$ and $a \leq b$, which is used as a concise representation of the set $\{i \in \mathbb{N} \mid a \leq i \leq b\}$ between its starting point a and its ending point b. Each TPG $G = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$ can be transformed into an Interval-timestamped Temporal Property Graph (ITPG), by putting the consecutive time points with the same values into the interval. More precisely, an ITPG $I = (\Omega', N, E, \rho, \lambda, \xi', \sigma')$ encoding G is defined in the following way. The temporal domain $\Omega = \{i \in \mathbb{N} \mid a \leq i \leq b\}$ of G is replaced by the interval $\Omega' = [a, b]$, and N, E, ρ, λ are the same as in G. Moreover, ξ' is a function that maps each object $o \in (N \cup E)$ to a set of maximal intervals where o exists according to function ξ. For example, for $\Omega = \{1, 2, 3, 4, 5\}$ and node n such that $\xi(n, 1) = \xi(n, 2) = \xi(n, 3) = \xi(n, 5) =$ *true* and $\xi(n, 4) =$ *false*, it holds that $\Omega' = [1, 5]$ and

$\xi'(n) = \{[1,3],[5,5]\}$. Notice that $\xi'(n)$ could not be defined as $\{[1,2],[3,3],[5,5]\}$ since $[1,2]$ is not a maximal interval where n exists. In other words, the set of intervals in $\xi'(n)$ has to be *coalesced*. Finally, function σ' is generated from σ in a similar way as ξ'. The formal definition of ITPG can be found in technical report [19].

IV. Adding Time to a Practical Graph Query Language

The main goal of this paper is to introduce a simple yet general query language for temporal property graphs. In this section, we give a guided tour of the query language, using the TPG shown in Figure 1 as the running example. All queries, except those presented alongside their equivalent rewritings, are numbered **Q1** through **Q12**, and will be used in the experimental evaluation in Section VII.

The **MATCH** clause is a fundamental construct in popular graph query languages such as Cypher [42], PGQL [45], and G-Core [2]. By using graph patterns, the **MATCH** clause allows to bind variables with objects in a property graph, giving rise to *binding tables* that are subsequently processed by the other components of the query language. As an important step towards the construction of a temporal graph query language, we show how the **MATCH** clause can be extended to bind variables with temporal objects in a TPG. In particular, we show how the syntax and semantics of the query language G-Core [2] can be extended to accommodate temporal graph patterns. As the syntax and semantics of G-Core are compatible with those of Cypher [42] and PGQL [45], these languages can accommodate such temporal graph patterns as well. These languages play a fundamental role in the ongoing graph query language standardization effort [46], and our proposal can provide a natural temporal extension for this standard.

Our proposed syntax for temporal regular path queries can be summarized as the following extension of the **MATCH** clause:

```
MATCH (x)-/path/-(y) ON graph
```

Here, **graph** is either a TPG or an ITPG, and **path** is an expression that can contain temporal and structural navigation operators, together with some other functionalities like testing the label of a node or an edge, and verifying the value of a property of a node or an edge. We will present the formal semantics of the language in Section V.

As a first example, assume that **contact_tracing** is the TPG shown in Figure 1. Then, the following G-Core expression extracts the list of people from **contact_tracing**:

```
Q1 MATCH (x:Person) ON contact_tracing
```

The operator **ON** specifies that **contact_tracing** is the input graph, and **(x:Person)** indicates that **x** is a variable to be assigned nodes with label **Person** from the input graph. The evaluation of a **MATCH** clause in G-Core results in a table consisting of bindings that assign to each variable an object from the input graph: a node, an edge, a label, or a property value. The result of evaluating **Q1** is the binding table:

x
n_1
n_2
n_3
n_6
n_7

At this point, two observations should be made: (i) G-Core does not consider **contact_tracing** as a temporal property graph, so no explicit time is associated with the objects in a binding table; (ii) Cypher [42] and PGQL [45] produce the same bindings as G-Core when evaluating the previous **MATCH** clause. How should this clause be evaluated if **contact_tracing** is considered as a temporal property graph? The first issue is that variables in the **MATCH** clause are to be assigned temporal objects; for example, **(x:Person)** indicates that **x** is a variable to be assigned a temporal object (v,t), where v is a node with label **Person** that exists at time point t. This issue is addressed by adding an extra column for each variable to indicate the time point when that variable exists (table entries appear side-by-side to save vertical space):

x	x_time
n_1	1
...	...
n_1	9

x	x_time
n_2	1
...	...
n_7	8

Observe that the time point t for each value v of **x** is stored in the column **x_time**. Hence, the binding **x** $\mapsto n_1$, **x_time** $\mapsto 1$ is in the resulting table, since n_1 is a node with label **Person** that exists at time point 1 in **contact_tracing**, and similarly for the other bindings. This illustrates that TRPQs without temporal navigation operate under snapshot reducibility, a design principle discussed in Section I-B.

Having explained how bindings to temporal objects are represented, we can now illustrate the main features of our query language. As in other popular graph query languages, we use curly brackets to indicate restrictions on property values. As our first example, consider the following **MATCH** clause:

```
Q2 MATCH (x:Person {risk = 'low'})
   ON contact_tracing
```

The expression **{risk = 'low'}** is used to indicate that the value of property **risk** must be **'low'**. The following binding table is the result of evaluating the previous **MATCH** clause:

x	x_time
n_1	1
...	...
n_1	9

x	x_time
n_2	1
...	...
n_2	4

x	x_time
n_6	2
...	...
n_6	11

Observe that the binding **x** $\mapsto n_2$, **x_time** $\mapsto 4$ is in this table, since n_2 is a node such that the label of n_2 is **Person**, n_2 exists at time point 4, and the value of property **risk** is **'low'** for n_2 at time point 4, and likewise for the other bindings in this table. As a second example, consider the following query:

```
Q3 MATCH (x:Person {risk = 'low' AND time = '1'})
   ON contact_tracing
```

In this case, we use the reserved word **time** to indicate that we are considering temporal objects at time point 1. The following is the result of evaluating this **MATCH** clause:

x	x_time
n_1	1
n_2	1

Other operators can limit the time under consideration, for example, to consider temporal objects at time less than 10:

```
Q4 MATCH (x:Person {risk = 'low' AND time < '10'})
   ON contact_tracing
```

Now, suppose that we want to retrieve the pairs of low- and high-risk people who have met, along with information about their meeting. For this, we can use the following query:

```
Q5 MATCH (x:Person {risk = 'low'})-
         [z:meets]->(y:Person {risk = 'high'})
   ON contact_tracing
```

The result of evaluating this **MATCH** clause is:

x	x_time	z	z_time	y	y_time
n_1	5	e_1	5	n_2	5
n_1	6	e_1	6	n_2	6
n_2	1	e_2	1	n_3	1
n_2	2	e_2	2	n_3	2

As in other popular graph query languages [2], [42], [45], an expression of the form **-[:meets]->** indicates the existence of an edge with label **meets**. We assign the variable **z** to the temporal object that represents that edge.

Importantly, an expression of the form **-[...]->** represents the structural navigation operator that is conceptually evaluated over the snapshots (temporal states) of the graph. This is the reason why each binding in the resulting table has the same value in columns **x_time**, **z_time**, and **y_time** . For example, the binding **x** $\mapsto n_1$, **x_time** $\mapsto 5$, **z** $\mapsto e_1$, **z_time** $\mapsto 5$, **y** $\mapsto n_2$, **y_time** $\mapsto 5$ is in this table, since n_1 is a low-risk person at time point 5, n_2 is a high-risk person at time point 5, and there exists an edge e_1 with label **meets** between n_1 and n_2 at time point 5.

To ensure that our proposal is practically useful, a minimum requirement is that queries can be evaluated in polynomial time over TPGs. Hence, we have to choose very carefully how structural navigation is combined with temporal navigation, and how we refer to time in the query language, as the complexity can quickly become intractable when navigation patterns are combined with functionalities for comparing property values [47]. In fact, there is even a fixed query Q for which this negative result holds [47]. This means that the problem of computing, given a graph G as input, the answer to Q over G is intractable in data complexity [48].

The basic temporal navigation operators in our language are **PREV** and **NEXT** that move by one unit of time into the past and into the future, respectively. Consider the following query:

```
Q6 MATCH (x:Person {test = 'pos'})-
         /PREV/-(y:Person)
   ON contact_tracing
```

Here, **x** and **y** are temporal objects that correspond to *the same* real-world object —a node of type **Person**. In this case, **x** has the value **'pos'** in the property **test**, meaning that **x** tested positive at some time point, and **y** denotes the same node at the time immediately before testing positive. Temporal navigation allows single-step temporal movement, and is orthogonal to structural navigation, following navigation orthogonality, discussed in Section I-B. Note that **PREV** and **NEXT** reference timestamps, operating under extended snapshot reducibility [17], discussed in Section II.

This example illustrates the use of notation **-/.../-** to specify a pattern that a path connecting objects **x** and **y** must satisfy. In general, such a pattern is a regular expression that can include temporal and structural operators (see formal definition in Section V). In this example, assuming that the temporal object (o_1, t_1) corresponds to **(x:Person {test = 'pos'})**, and the temporal object (o_2, t_2) corresponds to **(y:Person)**, then the expression **-/PREV/-** indicates that (o_1, t_1) must be connected with (o_2, t_2) through a path conforming to **PREV**, that is, $t_2 = t_1 - 1$. Importantly, **-/PREV/-** is evaluated under the restriction that no structural navigation must have occurred, given the separation between temporal and structural navigation that we are arguing for in this work. Hence, we conclude that $o_2 = o_1$. The following binding table is the result of evaluating **Q6**:

x	x_time	y	y_time
n_6	9	n_6	8

Temporal and structural navigation can be combined to retrieve information about which room person **x** was visiting immediately before she received a positive test result:

```
MATCH (x:Person {test = 'pos'})-
      /PREV/-(y:Person)-[:visits]->(z:Room)
ON contact_tracing
```

The result of evaluating this **MATCH** clause is:

x	x_time	y	y_time	z	z_time
n_6	9	n_6	8	n_4	8

Observe that the temporal operator **PREV** moves from (**x**, **x_time**) to (**y**, **y_time**), while the structural operator **-[:visits]->** moves from (**y**, **y_time**) to (**z**, **z_time**). Hence, temporal and structural navigation are carried out separately. Besides, observe that the intermediate variable **y** is not needed when retrieving the list of rooms that person **x** was visiting, we just included it to show the paths that are constructed when using different operators. The following simplified **MATCH** clause

```
MATCH (x:Person {test = 'pos'})-
      /PREV/-()-[:visits]->(z:Room)
ON contact_tracing
```

can be used to obtain the desired answer:

x	x_time	z	z_time
n_6	9	n_4	8

At this point the reader may be wondering why the language is asymmetric, and it includes different notation for temporal and structural navigation. We have kept the notation **-[...]->** to be compatible with graph query languages used today [2], [42], [45], but an important feature of our proposal is the use of notation **-/.../-** to include regular expressions combining temporal and structural operators. Hence, we include two basic

structural navigation operators, **BWD** ("backward") and **FWD** ("forward"), that are analogous to the temporal operators **PREV** and **NEXT**. Assume that an edge is given

$$(n, t) \xrightarrow{(e,t)} (n', t), \tag{1}$$

which, in the formal TPGs notation (see Definition III.1), represents the fact that $\rho(e) = (n, n')$, $\xi(n, t) = \mathit{true}$, $\xi(e, t) = \mathit{true}$, and $\xi(n', t) = \mathit{true}$. Then, operator **FWD** moves forward from node n to edge e, or from edge e to node n', while keeping time t unchanged. That is, **FWD** operates in a TPG snapshot corresponding to time t. Similarly, operator **BWD** moves backwards from node n' to edge e, and from edge e to node n in a TPG snapshot corresponding to time t. Thus, we can rewrite the previous **MATCH** clause as follows:

```
Q7 MATCH (x:Person {test = 'pos'})-
         /PREV/FWD/:visits/FWD/-(z:Room)
   ON contact_tracing
```

The regular expression **PREV/FWD/:meets/FWD** uses the concatenation operator **/** to indicate that operator **PREV** has to be executed first followed by the expression **FWD/:visits/FWD**, which is executed in the same way. (The precise syntax and semantics of such expressions are presented in Section V.) Observe that in our query language, the expression **-[:visits]->** is equivalent to **-/FWD/:visits/FWD/-**. This is because, given an edge of the form of Expression (1), the first operator **FWD** moves from n to e, then **:visits** checks that the label of e is **visits**, and finally the last operator **FWD** moves from e to n', thus obtaining the same result as using the operator **-[:visits]->** in an edge of the form of Expression (1).

So far we only looked at expressions that navigate one step at a time, temporally or structurally. Our language also supports the Kleene star, indicating zero or more occurrences of an operator. For example, **Q8** retrieves the list of rooms person **x** visited at any time prior to receiving a positive test (including also at the time when **x** received the test):

```
Q8 MATCH (x:Person {test = 'pos'})-
         /PREV*/FWD/:visits/FWD/-(z:Room)
   ON contact_tracing
```

producing the following temporal bindings:

x	x_time	z	z_time
n_6	9	n_4	8
n_6	9	n_4	7
n_6	9	n_5	6
n_6	9	n_5	5

As another example, we can retrieve the high-risk people who met someone who subsequently tested positive for an infectious disease:

```
Q9 MATCH (x:Person {risk = 'high'})-
         /FWD/:meets/FWD/NEXT*/-({test = 'pos'})
   ON contact_tracing
```

Recall that the temporal operator **NEXT** moves in time by one unit into the future. This query returns the following temporal bindings when evaluated over the graph in Figure 1:

x	x_time
n_3	4
n_7	5
n_7	6

Observe that the term **({test = 'pos'})** does not include a variable, as we are not storing the contacts who tested positive to avoid stigmatizing them, and only record those who are potentially at risk for complications.

Moreover, our query language allows to specify the number of times an operator is used. Thus, assuming that the time unit in **contact_tracing** is 5 minutes, we can retrieve the list of high-risk people who met someone who tested positive for an infectious disease 1 hour prior to the meeting:

```
Q10 MATCH (x:Person {risk = 'high'})-
          /FWD/:meets/FWD/PREV[0,12]-
          ({test = 'pos'})
    ON contact_tracing
```

Next, consider the following notion of close contact for an infectious disease: If person a visits the same room as person b, and b tests positive for this disease at most two weeks after they visited the same room as a, then a is considered to have been in close contact with an infected person. The **MATCH** clause below retrieves high-risk people who have been in close contact with an infected person:

```
Q11 MATCH (x:Person {risk = 'high'})-
          /FWD/:visits/FWD/:Room/BWD/:visits/
              BWD/NEXT[0,12]/-({test = 'pos'})
    ON contact_tracing
```

Observe that, as was the case for edge labels, node labels can be used inside an expression **-/.../-**, and so **-/:Room/-** in the expression above is equivalent to **-(:Room)-**. The query **Q11** produces the following binding table:

x	x_time
n_3	7
n_7	7
n_7	8

As the final example, assume that if person a meets with person b, and b tests positive for an infections disease at most two weeks after their meeting, then a should also be considered to have been in close contact with an infected person. **Q11** can be extended to consider this additional case:

```
MATCH (x:Person {risk = 'high'})-
      /(FWD/:meets/FWD/NEXT[0,12]) +
       (FWD/:visits/FWD/:Room/BWD/:visits/
        BWD/NEXT[0,12])/-({test = 'pos'})
ON contact_tracing
```

This query produces the following bindings:

x	x_time
n_3	4
n_3	7
n_7	5

x	x_time
n_7	6
n_7	7
n_7	8

As usual in regular expressions, operator **+** represents union. Thus, the regular expression in the previous **MATCH** clause indicates that the results of **FWD/:meets/FWD/NEXT[0,12]** should be put together with the results of **FWD/:visits/FWD/:Room/BWD/:visits/BWD/NEXT[0,12]**. Observe that

parentheses are used to have unambiguous expressions that can be parsed in a unique way. For example, the previous expression can be rewritten as follows to avoid using the temporal operator `NEXT[0,12]` twice. (Observe the required use of parentheses to get the desired effect.)

```
Q12 MATCH (x:Person {risk = 'high'})-
          /(FWD/:meets/FWD +
            FWD/:visits/FWD/:Room/BWD/:visits/
            BWD)/NEXT[0,12]/-({test = 'pos'})
    ON contact_tracing
```

In this section, we illustrated the main features of our proposed language and showed how popular graph query languages [2], [42], [45] can be extended to include these features. We will define the syntax and the semantics of our language next.

V. Temporal Regular Path Queries

In this section, we provide a formal syntax and semantics for the expression `path` described in the previous section, and study the complexity of evaluating it. In Section V-A, we extend the widely used notion of regular path query [1], [49]–[51] to deal with temporal objects in TPGs, which gives rise to the language NavL[PC,NOI]. Moreover, we show in Section V-A how NavL[PC,NOI] provides a formalization of the practical query language proposed in the previous section. Then we define the semantics of NavL[PC,NOI] in Section V-B, by following the definition of widely used query languages such as XPath and regular path queries [1], [49]–[55]. Moreover, we study in Section V-B the complexity of the evaluation problem for NavL[PC,NOI] for TPGs and ITPGs. Finally, we provide in Section V-C a comparison of our proposal with other temporal query languages. Proofs and additional results can be found in technical report [19].

A. Syntax of NavL[PC, NOI], and its relationship with the practical query language

Recall that labels, property names, and property values are drawn from the sets *Lab*, *Prop*, and *Val*, respectively. Then the expressions in NavL[PC,NOI], which are called temporal regular path queries (TRPQs), are defined by the grammar:

$$\mathsf{path} ::= \mathsf{test} \mid \mathsf{axis} \mid (\mathsf{path}/\mathsf{path}) \mid (\mathsf{path} + \mathsf{path}) \mid \mathsf{path}[n,m] \mid \mathsf{path}[n,\_] \quad (2)$$

where n and m are natural numbers such that $n \leq m$. Intuitively, test checks a condition on a given node or edge at a given time point, axis allows structural or temporal navigation, $(\mathsf{path}/\mathsf{path})$ is used for the concatenation of two TRPQs, $(\mathsf{path} + \mathsf{path})$ allows for the disjunction of two TRPQs, $\mathsf{path}[n,m]$ allows path to be repeated a number of times that is between n and m, whereas $\mathsf{path}[n,\_]$ only imposes a lower bound of at least n repetitions of expression path. The Kleene star path^* can be expressed as $\mathsf{path}[0,\_]$, and the expression $\mathsf{path}[\_,n]$ is equivalent to $\mathsf{path}[0,n]$.

Conditions on temporal objects are defined by the grammar:

$$\mathsf{test} ::= \mathbf{Node} \mid \mathbf{Edge} \mid \ell \mid p \mapsto v \mid < k \mid \exists \mid (?\mathsf{path}) \mid (\mathsf{test} \vee \mathsf{test}) \mid (\mathsf{test} \wedge \mathsf{test}) \mid (\neg \mathsf{test}) \quad (3)$$

where $\ell \in \mathit{Lab}$, $p \in \mathit{Prop}$, $v \in \mathit{Val}$, and $k \in \mathbb{N}$. Intuitively, test is meant to be applied to a temporal object, that is, to a pair (o,t) with object o and time point t. **Node** and **Edge** test whether the object is a node or an edge, respectively; the term ℓ checks whether the label of the object is ℓ; the term $p \mapsto v$ checks whether the value of property p is v for the object at the given time point; $\exists$ checks whether the object exists at the given time point; and $< k$ checks whether the current time point is less than k. Further, test can be $(?\mathsf{path})$, where path is an expression satisfying grammar (2), meaning that there is a path starting on the tested temporal object that satisfies path. Finally, test can be a disjunction or a conjunction of a pair of test expressions, or a negation of a test expression.

Furthermore, the following grammar defines *navigation*:

$$\mathsf{axis} ::= \mathbf{F} \mid \mathbf{B} \mid \mathbf{N} \mid \mathbf{P} \quad (4)$$

Operators $\mathbf{F}$, $\mathbf{B}$ move structurally in a TPG: $\mathbf{F}$ moves forward in the direction of an edge, and $\mathbf{B}$ moves backward in the reverse direction of an edge. Operators $\mathbf{N}$, $\mathbf{P}$ move temporally in a TPG: $\mathbf{N}$ moves to the next time point, and $\mathbf{P}$ moves to the previous time point.

Having a formal definition of the syntax of NavL[PC,NOI], we show that this language provides a formalization of the practical query language of Section IV. More precisely, temporal navigation operators `PREV` and `NEXT` in the practical query language correspond to the analogous operators $\mathbf{P}$ and $\mathbf{N}$ in NavL[PC,NOI], respectively, while structural navigation operators `BWD` and `FWD` in the practical query language correspond to the operators $\mathbf{B}$ and $\mathbf{F}$ in NavL[PC,NOI], respectively. Then consider the following `MATCH` clause over an arbitrary TPG:

```
MATCH (x:Person {test = 'pos'})-/PREV/-(y)
ON graph
```

Our task is to construct a query path in NavL[PC,NOI] such that the evaluation of this `MATCH` clause over `graph` is equivalent to the evaluation of path over this TPG. The following expression satisfies this condition:

$$(\mathbf{Node} \wedge \mathrm{Person} \wedge \mathrm{test} \mapsto \mathrm{pos})/\mathbf{P}/(\mathbf{Node} \wedge \exists)$$

Observe that $(\mathbf{Node} \wedge \mathrm{Person} \wedge \mathrm{test} \mapsto \mathrm{pos})$ is used to check whether the following conditions are satisfied for a temporal object (o,t): o is a node with label Person and with value pos in the property test at time point t. Notice that, by definition of TPGs, the fact that test $\mapsto$ pos holds at time t implies that node o exists at this time point. Hence, $(\mathbf{Node} \wedge \mathrm{Person} \wedge \mathrm{test} \mapsto \mathrm{pos})$ is used to represent the expression `(x:Person {test = 'pos'})`. Moreover, temporal navigation operator $\mathbf{P}$ is used to move from the temporal object (o,t) to a temporal object (o,t') such that $t' = t-1$, so that it is used to represent the expression `-/PREV/-`. Finally, the condition $(\mathbf{Node} \wedge \exists)$ is used to test that o is a node that exists at time t'. Observe that we explicitly need to mention the condition $\exists$, as expressions in NavL[PC,NOI] do not enforce the existence of temporal objects by default. The main reason

to choose such a semantics is that there are many scenarios where moving through temporal objects that do not exists is useful, in particular when these temporal objects only exist at certain time points. For example, if a room is unavailable for some time, then the temporal path expression

$$(\text{Room} \wedge \neg\exists)/(\mathbf{N}/\neg\exists)[0,\_]/(\text{Room} \wedge \exists)$$

can be used to look for the next time the room is available. Here, $(\mathbf{N}/\neg\exists)[0,\_]$ moves through an arbitrary number of time points during which the room is unavailable, until the condition $\exists$ holds, and the room becomes available.

As a second example, consider query **Q8** from Section IV. Based on the previous discussion, such a query can be represented as the following TRPQ:

$$(\mathbf{Node} \wedge \text{Person} \wedge \text{test} \mapsto \text{pos})/ (\mathbf{P}/\exists)[0,\_]/\mathbf{F}/(\text{visits} \wedge \exists)/\mathbf{F}/(\mathbf{Node} \wedge \text{Room}),$$

where all temporal objects must exist, as required in Section IV. Note that we have not explicitly included the existence condition on the last room node, as the existence of an edge at time point t implies, according to the definition of TPGs, the existence of its starting and ending nodes.

As an additional example, consider query **Q12** from Section IV, which uses many of the features of NavL[PC,NOI]. This query corresponds to the temporal path expression:

$$(\mathbf{Node} \wedge \text{Person} \wedge \text{risk} \mapsto \text{high})/(\mathbf{F}/(\text{meets} \wedge \exists)/\mathbf{F} + \mathbf{F}/(\text{visits} \wedge \exists)/\mathbf{F}/\text{Room}/\mathbf{B}/(\text{visits} \wedge \exists)/\mathbf{B})/ (\mathbf{N}/\exists)[0,12]/(\mathbf{Node} \wedge \text{test} \mapsto \text{pos})$$

As our final example, consider query **Q4** from Section IV. The use of a condition over the reserved word **time** is represented in NavL[PC,NOI] by the condition $< k$. For example, **time < '10'** is represented by the condition < 10, as a temporal object (o,t) satisfies < 10 if, and only if, $t < 10$. Hence, **Q4** is equivalent to the following query in NavL[PC,NOI]:

$$(\mathbf{Node} \wedge \text{Person} \wedge \text{risk} \mapsto \text{low} \wedge < 10)$$

Notice that abbreviations can be introduced for some of the operators described in this section, and some other common operators, to make notation of the formal language easier to use. For example, we could use condition $= k$, which is written in NavL[PC,NOI] as $(< k+1 \wedge \neg(< k))$, and operator $\mathbf{NE}$ that moves by one unit into the future if the object that is reached exists. However, as such operators are expressible in NavL[PC,NOI], we prefer to use a minimal notation in this formal language to simplify its definition and analysis.

B. Semantics and complexity of NavL[PC,NOI]

Let $G = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$ be a TPG. Given an expression path in NavL[PC,NOI], the evaluation of path over G, denoted by $[\![\text{path}]\!]_G$, is defined by the set of tuples (o,t,o',t') such that there exists a sequence of temporal objects starting in (o,t), ending in (o',t'), and conforming to path. More precisely, assume that $\mathsf{src}(e) = v_1$ and $\mathsf{tgt}(e) = v_2$ whenever $\rho(e) = (v_1, v_2)$, and assume that $\mathsf{PTO}(G) = (N \cup E) \times \Omega \times (N \cup E) \times \Omega$. Then the evaluation of the axes in grammar (2) is defined as:

$$\begin{aligned}
[\![\mathbf{F}]\!]_G &= \{(v,t,e,t) \in \mathsf{PTO}(G) \mid \mathsf{src}(e) = v\} \cup \\
&\quad \{(e,t,v,t) \in \mathsf{PTO}(G) \mid \mathsf{tgt}(e) = v\} \\
[\![\mathbf{B}]\!]_G &= \{(v,t,e,t) \in \mathsf{PTO}(G) \mid \mathsf{tgt}(e) = v\} \cup \\
&\quad \{(e,t,v,t) \in \mathsf{PTO}(G) \mid \mathsf{src}(e) = v\} \\
[\![\mathbf{N}]\!]_G &= \{(o,t_1,o,t_2) \in \mathsf{PTO}(G) \mid t_2 = t_1 + 1\} \\
[\![\mathbf{P}]\!]_G &= \{(o,t_1,o,t_2) \in \mathsf{PTO}(G) \mid t_2 = t_1 - 1\}
\end{aligned}$$

Moreover, assuming that, path, path_1 and path_2 are expressions in NavL[PC,NOI], we have that:

$$\begin{aligned}
[\![(\mathsf{path}_1/\mathsf{path}_2)]\!]_G &= \{(o_1,t_1,o_2,t_2) \in \mathsf{PTO}(G) \mid \\
&\qquad \exists(o,t) : (o_1,t_1,o,t) \in [\![\mathsf{path}_1]\!]_G \\
&\qquad\quad \text{and } (o,t,o_2,t_2) \in [\![\mathsf{path}_2]\!]_G\}, \\
[\![(\mathsf{path}_1 + \mathsf{path}_2)]\!]_G &= [\![\mathsf{path}_1]\!]_G \cup [\![\mathsf{path}_2]\!]_G, \\
[\![\mathsf{path}[n,m]]\!]_G &= \bigcup_{k=n}^{m} [\![\mathsf{path}^k]\!]_G, \\
[\![\mathsf{path}[n,_]]\!]_G &= \bigcup_{k\geq n} [\![\mathsf{path}^k]\!]_G,
\end{aligned}$$

where path^k is defined as the concatenation of path with itself k times. Finally, the evaluation of an expression test, defined according to grammar (3), is a navigation expression that stays in the same temporal object if test is satisfied: $[\![\mathsf{test}]\!]_G = \{(o,t,o,t) \in \mathsf{PTO}(G) \mid (o,t) \models \mathsf{test}\}$. Hence, to conclude the definition of the semantic of NavL[PC,NOI], we need to indicate when a temporal object (o,t) satisfies a condition test, which is denoted by $(o,t) \models \mathsf{test}$. Formally, this is recursively defined as follows (omitting the usual semantics for Boolean connectives):

- If test $= \mathbf{Node}$, then $(o,t) \models \mathsf{test}$ if $o \in N$;
- If test $= \mathbf{Edge}$, then $(o,t) \models \mathsf{test}$ if $o \in E$;
- If test $= \ell$, with $\ell \in \mathit{Lab}$, then $(o,t) \models \mathsf{test}$ if $\lambda(o) = \ell$;
- If test $= p \mapsto v$, with $p \in \mathit{Prop}$ and $v \in \mathit{Val}$, then $(o,t) \models \mathsf{test}$ if $\sigma(o,p,t)$ is defined and $\sigma(o,p,t) = v$;
- If test $= \exists$, then $(o,t) \models \mathsf{test}$ if $\xi(o,t) = \mathit{true}$;
- If test $= < k$, then $(o,t) \models \mathsf{test}$ if $t < k$;
- If test $= (?\mathsf{path})$ for an expression path conforming to grammar (2), then $(o,t) \models \mathsf{test}$ if there exists a temporal object (o',t') in G such that $(o,t,o',t') \in [\![\mathsf{path}]\!]_G$.

To define the evaluation of an expression path over a interval-timestamped temporal property graph I, we just need to translate I into an equivalent TPG and consider the previous definition. Formally, assuming that $\mathsf{can}(\cdot)$ is a canonical translation from an ITPG into an equivalent TPG, we have that: $[\![\mathsf{path}]\!]_I = [\![\mathsf{path}]\!]_{\mathsf{can}(I)}$.

Having a formal definition of TRPQs allows not only to provide an unambiguous definition of the practical query language of Section IV, but also to formally study the complexity of evaluating this language. Assuming that $\mathcal{G}$ is a class of graphs and $\mathcal{L}$ is a query language, define $\mathsf{Eval}(\mathcal{G}, \mathcal{L})$ as the problem of verifying whether $(o,t,o',t') \in [\![\mathsf{path}]\!]_G$, for an

input consisting of a graph $G \in \mathcal{G}$, an expression path in $\mathcal{L}$ and a pair (o, t), (o', t') of temporal objects in G. By studying the complexity of Eval$(\mathcal{G}, \mathcal{L})$ for different fragments $\mathcal{L}$ of NavL[PC,NOI], we can understand how the use of the operators in NavL[PC,NOI] affects the complexity of the evaluation problem, and which operators are mode difficult to implement.

Assume that NavL[PC] is the fragment of NavL[PC,NOI] obtained by disallowing numerical occurrence indicators, while NavL[NOI] is the fragment of NavL[PC,NOI] obtained by disallowing path conditions.

Theorem V.1. *The following results hold.*

1) Eval(TPG, NavL[PC,NOI]) *and* Eval(ITPG, NavL[PC]) *can be solved in polynomial time.*
2) Eval(ITPG, NavL[NOI]) *is* Σ_2^p*-hard, and* Eval(ITPG, NavL[PC,NOI]) *is* PSPACE*-complete.*

The results of this section can guide future implementations of NavL[PC,NOI] over interval-timestamped TPGs. The main insight is that, while Eval(ITPG, NavL[NOI]) and Eval(ITPG, NavL[PC,NOI]) are intractable, the language including only path conditions can be efficiently evaluated over such graphs.

C. A comparison with T-GQL and Cypher

T-GQL is a recently proposed temporal query language [13] developed on top of Cypher [42], a popular graph query language. We now compare our TRPQs with T-GQL, and with the alternative of implementing a temporal graph query language that encodes time intervals as lists directly in Cypher.

First, consider the five design principles of our language, described in Section I-B. Since Cypher's data model does not explicitly consider time, it is not surprising that it does not satisfy navigability, navigation orthogonality, static testability, or snapshot reducibility, and only node-edge symmetry is satisfied. T-GQL satisfies navigability, navigation orthogonality and snapshot reducibility, but it treats nodes and edges differently, violating node-edge symmetry. Moreover, T-GQL test conditions do not satisfy static testability.

Second, consider the complexity of the query evaluation problem. As shown in Theorem V.1, our query language can be evaluated in polynomial time over temporal property graphs. In contrast, the evaluation problem for Cypher is intractable, even if we focus on non-temporal property graphs (i.e., a temporal property graph consisting of a single timestamp). In fact, a fixed query that checks for the existence of two disjoint paths from the same source node to the same destination node can be expressed in Cypher and is known to be NP-hard [42]. Whether these intractability results carry over T-GQL is not clear, as an exact characterization of T-GQL as a fragment of Cypher has not yet been provided.

Finally, we compare the expressive power of our proposal with Cypher and T-GQL. As Cypher is a general purpose graph query language, it is not surprising that every query in our proposal can be expressed in it, but at the cost of using unnatural and expensive time interval encodings. However, we can show that some natural TRPQs cannot be expressed in T-GQL. First, consider a graph for travel scheduling that includes different transportation services, such as flights, trains, and buses. By the definition of consecutive path in [13], it is not possible to express a query in T-GQL that indicates how to go from one city to another combining different transportation services, which can be easily expressed in our proposal. As a more fundamental example, consider a query that retrieves paths that combine an arbitrary number of temporal journeys, some of them moving to the future and some to the past. Such a combination of temporal journeys cannot be specified in T-GQL, while it can be handled by our proposal.

VI. Implementation

We implement a fragment of NavL[PC,NOI] that includes all queries of Section IV over interval-timestamped TPGs. We use Rust and the Itertools library [56], which efficiently implements dataflow operators, supports lazy evaluation of expressions, and collects data only when necessary. For multi-threaded implementation, we use Rayon-Rs [57], an interface over dataflow operators. Our algorithms can be implemented using any system that supports the dataflow model, such as Apache Spark [58], Apache Flink [59], Timely [60] and Differential dataflow [61].

We represent a TPG as a pair of interval-timestamped temporal relations **Nodes**(**id**, **label**, **properties**, **time**) and **Edges**(**id**, **src**, **tgt**, **label**, **properties**, **time**), where **properties** are a set of key-value pairs. For example, for node n_2 and edge e_1 from Figure 1, we have:

Nodes

id	label	properties	time
n_2	Person	{name = 'Bob', risk = 'low'}	[1, 4]
n_2	Person	{name = 'Bob', risk = 'high'}	[5, 9]

Edges

id	src	tgt	label	properties	time
e_1	n_1	n_2	meets	{loc = 'cafe'}	[3, 3]
e_1	n_1	n_2	meets	{loc = 'park'}	[5, 6]

By the formal definition of TRPQs in Section V, we know that temporal and structural navigation operators are orthogonal, in the sense that the language allows non-simultaneous single-step time and structural movements. Hence, we break down the evaluation of a TRPQ into **Step 1:** evaluating the *structural navigation* portion of the path expression over the interval-based TPG; **Step 2:** evaluating the *temporal navigation* portion of the path expression over the interval-based intermediate result; and **Step 3:** if needed, transforming the intermediate result into a point-wise representation for the final portion of evaluation and materialization.

Evaluation of conventional path queries in **Step 1** is a well-studied problem [1], [51]. In this work, we select an optimized select-project-join execution plan for each query in Section IV, and then implement these plans using Itertools operators in Rust.We implement in-memory hash-join that uses interval-based reasoning to identify temporally-aligned [25] matches. For example, for `Q5`, we compute the intersection of the validity intervals for `x`, `y` and `z`. For TRPQs without temporal navigation (`Q1`-`Q5`), the final bindings table can be returned

TABLE I
TEMPORAL PROPERTY GRAPHS USED IN EXPERIMENTS.

	# nodes	# edges	# temp. nodes	# temp. edges
G1	1,000	12,000	3,500	14,000
G2	2,000	30,000	7,000	35,000
G3	4,000	84,000	14,000	94,000
G4	6,000	158,000	20,000	180,000
G5	8,000	253,000	28,000	282,000
G6	10,000	371,000	34,000	413,000
G7	25,000	2,046,000	85,000	2,215,000
G8	50,000	7,370,000	170,000	8,048,000
G9	75,000	15,717,000	256,000	17,554,000
G10	100,000	28,996,000	340,000	32,255,000

after this step, and it can remain temporally coalesced. For example, the coalesced binding table for `Q5` will contain:

x	x_time	z	z_time	y	y_time
n_1	[5,6]	e_1	[5,6]	n_2	[5,6]
n_2	[1,2]	e_2	[1,2]	n_3	[1,2]

The interpretation of this temporally coalesced result is snapshot-based: we bind `x` $= n_1$, `z` $= e_1$, `y`$= n_2$, with `x_time` = `y_time` = `z_time` = 5, and similarly for time 6.

Step 2: To evaluate the *temporal navigation* portion of the path expression, we use interval-based reasoning to join and prune out potential matches that do not satisfy the temporal constraint. For example, for `Q7`, we can limit the validity interval of `z` to the time immediately before `x` was tested positive. Note that interval intersection and union can be computed in constant time based on interval boundaries.

Step 3: For the final portion of query evaluation, we may need to use point-wise reasoning for *temporal navigation*. For example, `Q8` retrieves the list of rooms `z` that person `x` visited at or prior to the time of testing positive. The `PREV` operator is defined over time points, and we need to compare pairs of time points of `x` and `z` to correctly identify person-room pairs. Furthermore, *result generation* for TRPQs that use temporal navigation must compute point-based bindings. Returning to our example, in the result of `Q8`, `x_time` may or may not be the same as `z_time`, and so we cannot use an interval representation for the output bindings such as (n_6, [5,6], n_5, [3,5]), because such a representation is inherently snapshot-based and it does not uniquely map to a set of point-wise temporal bindings over n_6 and n_5.

An exception are TRPQs that return a single variable, such as `Q9`-`Q12`. Results of such queries can be returned temporally coalesced for compactness, although this rarely translates to savings in the running time of query execution, because temporal constraints must be check over a point-based representation for these queries in Step 3, as discussed above.

VII. EXPERIMENTAL EVALUATION

All experiments were run as a multi-threaded Rust application on a single cluster node with 64 GB of RAM and an Intel Xeon Platinum 8268 CPU, using the Slurm scheduler [62]. According to our results (Figure 3), performance for demanding queries was best at 16 CPU cores, and we use this setting in all experiments, unless noted otherwise. Reported execution times are averages of 5 runs. In most cases, the coefficient of variation of the running time was less than 6% (max 10%).

TABLE II
EXECUTION TIME OF QUERIES Q1 THROUGH Q12 FOR GRAPH G10.

	interval-based time (s)	total time (s)	output size
Q1	0.004	0.004	341,278
Q2	0.017	0.017	278,931
Q3	0.016	0.016	26,494
Q4	0.038	0.038	116,021
Q5	4.546	4.546	743,714
Q6	0.096	0.173	86,553
Q7	0.036	0.079	47,287
Q8	0.025	0.379	1,277,729
Q9	0.828	0.983	1,234,922
Q10	0.899	1.509	3,927,763
Q11	1.375	4.986	22,961,108
Q12	2.434	6.455	26,888,871

A. Experimental datasets

We built interval-timestamped TPGs (per Sec. III-B) similar to Figure 1 using a trajectory dataset generated by Ojagh et al. [63] to study COVID-19 contact tracing. The authors tracked 20 individuals on the University of Calgary campus, and used that data to simulate trajectories of individuals visiting campus locations, recording the times when individuals entered and exited those locations. The synthetic dataset of Ojagh et al. records time up to a second. To make this data more realistic, we (i) made temporal resolution coarser, mapping timestamps to 5-min windows, and (ii) associated individuals with locations where they spent at least 2.5 min.

Our goal was to have an interval-timestamped graph with two types of nodes, `Person` and `Room` (representing classrooms), and two types of edges, `visits` and `meets`. To achieve this, we represented 100,000 individuals as `Person` nodes, with their periods of validity corresponding to visits of classrooms. Next, from among 410 unique locations in the dataset, we selected 100 most frequently visited as nodes of type `Room`, with periods of validity defined by the times of first entrance and last exit. Then, we added a `visits` edge between each person and each room they visit, with an appropriate time interval. We used information about the remaining 310 locations to add bi-directional `meets` edges between a pair of individuals who were at the same location at the same time. Finally, we randomly selected 18% of the `Person` nodes (proportion of the Canadian population aged 65+) as high risk for disease complications, and fixed this property over the lifespan of those nodes.

To study the impact of graph size on performance, we created graphs at different scale factors by randomly selecting a subset of the `Person` nodes of a given size, and keeping only the valid edges. To study the impact of query selectivity on performance, we selected between 2% and 10% of the `Person` nodes as positive for COVID-19, assigning the time of a positive test uniformly at random from the temporal domain of the graph, and keeping the selected nodes as positive for the remainder of their lifespan.

Table I summarized the temporal graphs used in our experiments. The largest graph has 100,000 unique `Person` nodes, 100 unique `Room` nodes, and a temporal domain of 48 time points, each representing a 5-minute window. This corresponds to 340,000 temporal nodes and over 32 million temporal edges.

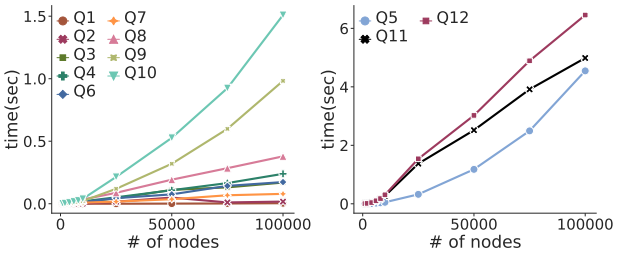

Fig. 2. Effect of graph size on query execution time, on G1-G10.

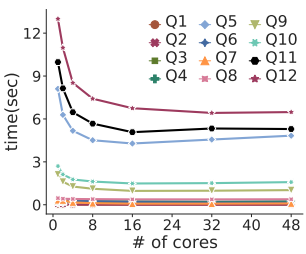

Fig. 3. Effect of parallelism on G10.

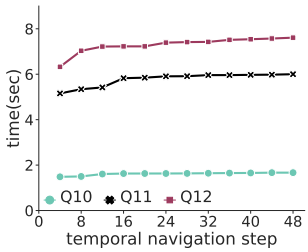

Fig. 4. Effect of temp. nav. on G10.

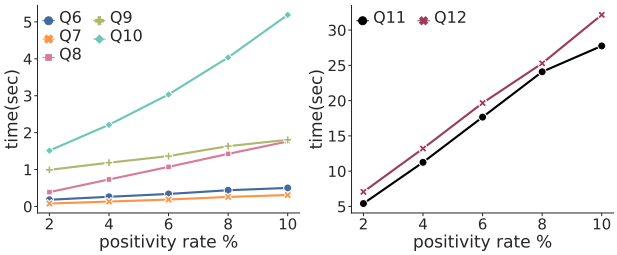

Fig. 5. Effect of positivity rate on query execution time, on G10.

B. Results

For the first experiment, we executed queries **Q1**-**Q12**, discussed in Section IV, over graph G10 (Table I). Table II shows the execution time of each query in seconds, and its output size in the number of tuples in the bindings table. Recall from Section VI that Steps 1 and 2 of query evaluation act on the interval representation or TRPG, while Step 3 expands the output of Step 2 into a point-based representation to check any remaining temporal constraints. Our implementation uses lazy evaluation. Decoupling the execution times of Steps 1 and 2 for the purpose of measurement would degrade performance, and we report these times jointly as "interval-based time" in Table II. Queries **Q1**-**Q5** do not use temporal navigation, and so interval-based time and total time coincide and the output can remain temporally coalesced. In contrast, **Q6**-**Q12** use temporal navigation; they require both interval-based and point-based processing, and the output for these queries is point-based.

We observe that most queries execute in less than 1 sec. The most challenging queries, **Q11** and **Q12**, both produce over 22 million tuples in the output and take at most 6.5 sec.

In the second experiment, we execute all queries over graphs G1-G10 to study the impact of graph size on query performance. Figure 2 shows this result, with the number of unique **Person** nodes on the x-axis, and execution time in seconds on the y-axis. Observe that the running time increases linearly for all queries except **Q5**, **Q9**, and **Q10** where the time increases approximately quadratically with increasing graph size. Increase in the running time is nearly perfectly explained by the increase in the size of the output. For example, increasing input size by a factor of x10 nodes and x100 edges (G6 to G10) increases output size of **Q11** (resp. **Q12**) by a factor of 18.39 (resp. 19.29), and it increases the execution time by a factor of 18.89 (resp. 19.29).

In our third experiment, we studied the impact of parallelism on performance. Figure 3 shows the result of this experiment over the largest graph, G10, with the number of CPU cores on the x-axis and execution time in seconds on the y-axis. (The number of threads is the number of CPUs + 1.) Observe that the most demanding queries **Q5**, **Q10**, **Q11**, and **Q12** substantially benefit from increased parallelism, with best performance at 16 cores. For example, **Q12** executes in 6.45 sec on 16 cores, down from 13 sec on 1 core.

Queries **Q6**-**Q11** all select **Person** nodes that at some point had a positive COVID-19 test. In our next experiment, we vary the positivity rate from 2% to 10%, thus impacting query selectivity, and study its effect on execution time. Figure 5 shows the result of this experiment over the largest graph, G10, with positivity rate on the x-axis and execution time on the y-axis, showing a linear relationship.

Finally, we consider the effect of temporal navigation on query performance. We select queries **Q10**, **Q11** and **Q12** because they all contain a temporal navigation operator with a numerical occurrence indicator (**PREV[n,m]** in **Q10** and **NEXT [n,m]** in **Q11** and **Q12**). We set $n = 0$, and vary the maximum number of temporal navigation steps m between 4 and 48 in increments of 4. Figure 4 shows the result over G10 with m on the x-axis and query execution time on the y-axis. We observe that increasing m increases the execution time linearly, and plateaus when m reaches 16.

VIII. Conclusions and Future Work

We considered temporal property graphs (TPGs) and proposed temporal regular path queries (TRPQs) that incorporate time into TPG navigation. Starting with design principles, we proposed a natural syntactic extension of the MATCH clause of popular query languages, formally presented the semantics of TRPQs, and studied the complexity of their evaluation. We also demonstrated that a fragment of the TRPQ language can be implemented efficiently. We hope that our work on the syntax and semantics, the positive complexity results, and our implementation and evaluation will pave the way to usable and practical production-level implementations of TRPQs.

An interesting future direction is to add support for aggregation and grouping, to incorporate our methods into existing graph processing systems like GraphX [64], Portal [65] or Neo4j [66], and to investigate a range of systems questions.

IX. Acknowledgements

M. Arenas and P. Bahamondes were funded by ANID - Millennium Science Initiative Program - Code ICN17_002 and Fondecyt grant 1191337. This research was supported in part by NSF Award No. 1916505, and by NYU IT High Performance Computing resources, services, and staff expertise.

REFERENCES

[1] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc, "Foundations of modern query languages for graph databases," *ACM Comput. Surv.*, vol. 50, no. 5, pp. 68:1–68:40, 2017. [Online]. Available: http://doi.acm.org/10.1145/3104031

[2] R. Angles, M. Arenas, P. Barcelo, P. Boncz, G. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda, O. van Rest, and H. Voigt, "G-core: A core for future graph query languages," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1421–1432. [Online]. Available: https://doi.org/10.1145/3183713.3190654

[3] M. Goetz, J. Leskovec, M. McGlohon, and C. Faloutsos, "Modeling blog dynamics," in *Proceedings of the Third International Conference on Weblogs and Social Media, ICWSM 2009, San Jose, California, USA, May 17-20, 2009*, E. Adar, M. Hurst, T. Finin, N. S. Glance, N. Nicolov, and B. L. Tseng, Eds. San Jose, CA: The AAAI Press, 2009, pp. 26–33. [Online]. Available: http://aaai.org/ocs/index.php/ICWSM/09/paper/view/152

[4] J. Leskovec, L. A. Adamic, and B. A. Huberman, "The dynamics of viral marketing," *ACM Trans. Web*, vol. 1, no. 1, p. 5–es, May 2007. [Online]. Available: https://doi.org/10.1145/1232722.1232727

[5] J. Leskovec, L. Backstrom, R. Kumar, and A. Tomkins, "Microscopic evolution of social networks," in *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 462–470. [Online]. Available: https://doi.org/10.1145/1401890.1401948

[6] P. Sarkar, D. Chakrabarti, and M. I. Jordan, "Nonparametric link prediction in dynamic networks," in *Proceedings of the 29th International Coference on International Conference on Machine Learning*, ser. ICML'12. Madison, WI, USA: Omnipress, 2012, p. 1897–1904.

[7] S. Asur, S. Parthasarathy, and D. Ucar, "An event-based framework for characterizing the evolutionary behavior of interaction graphs," *ACM Trans. Knowl. Discov. Data*, vol. 3, no. 4, Dec. 2009. [Online]. Available: https://doi.org/10.1145/1631162.1631164

[8] A. Beyer, P. Thomason, X. Li, J. Scott, and J. Fisher, "Mechanistic insights into metabolic disturbance during type-2 diabetes and obesity using qualitative networks," *Transactions on Computational Systems Biology XII, Special Issue on Modeling Methodologies*, vol. 12, pp. 146–162, 2010. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11712-1_4

[9] J. M. Stuart, E. Segal, D. Koller, and S. K. Kim, "A gene-coexpression network for global discovery of conserved genetic modules," *Science*, vol. 5643, no. 302, pp. 249—255, 2003.

[10] J. Chan, J. Bailey, and C. Leckie, "Discovering correlated spatio-temporal changes in evolving graphs," *Knowledge and Information Systems*, vol. 16, no. 1, pp. 53–96, 2008.

[11] P. Papadimitriou, A. Dasdan, and H. Garcia-Molina, "Web graph similarity for anomaly detection," *J. Internet Services and Applications*, vol. 1, no. 1, pp. 19–30, 2010. [Online]. Available: http://dx.doi.org/10.1007/s13174-010-0003-x

[12] J. Byun, S. Woo, and D. Kim, "Chronograph: Enabling temporal graph traversals for efficient information diffusion analysis over time," *IEEE Trans. Knowl. Data Eng.*, vol. 32, no. 3, pp. 424–437, 2020. [Online]. Available: https://doi.org/10.1109/TKDE.2019.2891565

[13] A. Debrouvier, E. Parodi, M. Perazzo, V. Soliani, and A. Vaisman, "A model and query language for temporal graph databases," *VLDB Journal*, 2021.

[14] T. Johnson, Y. Kanza, L. V. S. Lakshmanan, and V. Shkapenyuk, "Nepal: a path query language for communication networks," in *Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics, NDA@SIGMOD 2016, San Francisco, California, USA, July 1, 2016*, A. Arora, S. Roy, and S. Mehta, Eds. ACM, 2016, pp. 6:1–6:8. [Online]. Available: https://doi.org/10.1145/2980523.2980530

[15] A. G. Labouseur, J. Birnbaum, P. W. Olsen, S. R. Spillane, J. Vijayan, J. H. Hwang, and W. S. Han, "The G* graph database: efficiently managing large distributed dynamic graphs," *Distributed and Parallel Databases*, vol. 33, no. 4, pp. 479–514, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10619-014-7140-3

[16] V. Z. Moffitt and J. Stoyanovich, "Temporal graph algebra," in *Proceedings of The 16th International Symposium on Database Programming Languages*, ser. DBPL '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3122831.3122838

[17] M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass, "Temporal Statement Modifiers," *ACM Transactions on Database Systems*, vol. 25, no. 4, pp. 407–456, 2000.

[18] A. Montanari and J. Chomicki, *Time Domain*. Boston, MA: Springer US, 2009, pp. 3103–3107. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-39940-9_427

[19] M. Arenas, P. Bahamondes, A. Aghasadeghi, and J. Stoyanovich, "Temporal regular path queries," *CoRR*, vol. abs/2107.01241, 2021. [Online]. Available: https://arxiv.org/abs/2107.01241

[20] L. Liu and M. T. Zsu, *Encyclopedia of Database Systems*, 1st ed. Boston, MA: Springer Publishing Company, Incorporated, 2009.

[21] J. Clifford and A. U. Tansel, "On an algebra for historical relational databases: Two views," in *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '85. New York, NY, USA: Association for Computing Machinery, 1985, p. 247–265. [Online]. Available: https://doi.org/10.1145/318898.318922

[22] C. S. Jensen, M. D. Soo, and R. T. Snodgrass, "Unifying temporal data models via a conceptual model," *Information Systems*, vol. 19, no. 7, pp. 513 – 547, 1994. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0306437994900132

[23] R. Snodgrass and I. Ahn, "A taxonomy of time databases," in *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '85. New York, NY, USA: ACM, 1985, pp. 236–246. [Online]. Available: http://doi.acm.org/10.1145/318898.318921

[24] M. H. Böhlen, R. Busatto, and C. S. Jensen, "Point Versus Interval-based Temporal Data Models," in *Proceedings of the 14th IEEE ICDE*. Orlando, FL: IEEE, 1998, pp. 192–200. [Online]. Available: http://people.cs.aau.dk/{~}csj/Thesis/pdf/chapter7.pdf

[25] A. Dignös, M. H. Böhlen, and J. Gamper, "Temporal alignment," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 433–444. [Online]. Available: https://doi.org/10.1145/2213836.2213886

[26] B. Salzberg and V. J. Tsotras, "Comparison of access methods for time-evolving data," *ACM Computing Surveys*, vol. 31, no. 2, pp. 158–221, jun 1999. [Online]. Available: http://portal.acm.org/citation.cfm?doid=319806.319816

[27] K. G. Kulkarni and J. Michels, "Temporal features in SQL: 2011," *SIGMOD Record*, vol. 41, no. 3, pp. 34–43, 2012. [Online]. Available: http://doi.acm.org/10.1145/2380776.2380786

[28] K. M. Borgwardt, H.-P. Kriegel, and P. Wackersreuther, "Pattern mining in frequent dynamic subgraphs," in *Proceedings of the Sixth International Conference on Data Mining*, ser. ICDM '06. USA: IEEE Computer Society, 2006, p. 818–822. [Online]. Available: https://doi.org/10.1109/ICDM.2006.124

[29] A. Fard, A. Abdolrashidi, L. Ramaswamy, and J. Miller, "Towards Efficient Query Processing on Massive Time-Evolving Graphs," in *Proceedings of the 8th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2012, pp. 567–574. [Online]. Available: http://eudl.eu/doi/10.4108/icst.collaboratecom.2012.250532

[30] A. Ferreira, "Building a reference combinatorial model for MANETs," *IEEE Network*, vol. 18, no. 5, pp. 24–29, 2004.

[31] A. Kan, J. Chan, J. Bailey, and C. Leckie, "A query based approach for mining evolving graphs," in *Proceedings of the Eighth Australasian Data Mining Conference - Volume 101*, ser. AusDM '09. AUS: Australian Computer Society, Inc., 2009, p. 139–150.

[32] U. Khurana and A. Deshpande, "Efficient snapshot retrieval over historical graph data," in *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ser. ICDE '13. USA: IEEE Computer Society, 2013, p. 997–1008. [Online]. Available: https://doi.org/10.1109/ICDE.2013.6544892

[33] ——, "Storing and Analyzing Historical Graph Data at Scale," in *Proceedings of the 19th International Conference on Extending Database Technology, EDBT'16*, Bordeaux, France, 2016, pp. 65–76. [Online]. Available: http://arxiv.org/abs/1509.08960

[34] M. Lahiri and T. Berger-Wolf, "Mining Periodic Behavior in Dynamic Social Networks," in *2008 Eighth IEEE International Conference on Data Mining*, 2008, pp. 373–382.

[35] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng, "On Querying Historical Evolving Graph Sequences," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 726–737, 2011.

[36] K. Semertzidis, E. Pitoura, and K. Lillis, "Timereach: Historical reachability queries on evolving graphs," in *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015*, G. Alonso, F. Geerts, L. Popa, P. Barceló, J. Teubner, M. Ugarte, J. V. den Bussche, and J. Paredaens, Eds. Brussels, Belgium: OpenProceedings.org, 2015, pp. 121–132. [Online]. Available: https://doi.org/10.5441/002/edbt.2015.12

[37] K. Sricharan and K. Das, "Localizing anomalous changes in time-evolving graphs," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, Snowbird, Utah USA, 2014, pp. 1347–1358. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2588555.2612184

[38] L. Yang, L. Qi, Y. Zhao, B. Gao, and T. Liu, "Link analysis using time series of web graphs," in *Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, CIKM 2007, Lisbon, Portugal, November 6-10, 2007*, M. J. Silva, A. H. F. Laender, R. A. Baeza-Yates, D. L. McGuinness, B. Olstad, Ø. H. Olsen, and A. O. Falcão, Eds. ACM, 2007, pp. 1011–1014.

[39] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu, "Path problems in temporal graphs," *Proc. VLDB Endow.*, vol. 7, no. 9, pp. 721–732, 2014. [Online]. Available: http://www.vldb.org/pvldb/vol7/p721-wu.pdf

[40] H. Wu, J. Cheng, Y. Ke, S. Huang, Y. Huang, and H. Wu, "Efficient algorithms for temporal path computation," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 11, pp. 2927–2942, 2016. [Online]. Available: https://doi.org/10.1109/TKDE.2016.2594065

[41] H. Wu, Y. Huang, J. Cheng, J. Li, and Y. Ke, "Reachability and time-based path queries in temporal graphs," in *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. IEEE Computer Society, 2016, pp. 145–156. [Online]. Available: https://doi.org/10.1109/ICDE.2016.7498236

[42] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, "Cypher: An evolving query language for property graphs," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1433–1445. [Online]. Available: https://doi.org/10.1145/3183713.3190657

[43] A. Dignös, M. H. Böhlen, and J. Gamper, "Temporal alignment," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, Eds. ACM, 2012, pp. 433–444. [Online]. Available: https://doi.org/10.1145/2213836.2213886

[44] M. H. Böhlen, R. T. Snodgrass, and M. D. Soo, "Coalescing in temporal databases," in *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, 1996, pp. 180–191.

[45] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi, "Pgql: A property graph query language," in *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, ser. GRADES '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2960414.2960421

[46] Association of ISO Graph Query Language Proponents, "GQL standard," 2020, https://www.gqlstandards.org.

[47] L. Libkin, W. Martens, and D. Vrgoc, "Querying graphs with data," *J. ACM*, vol. 63, no. 2, pp. 14:1–14:53, 2016.

[48] M. Y. Vardi, "The complexity of relational query languages (extended abstract)," in *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, ser. STOC '82. New York, NY, USA: Association for Computing Machinery, 1982, p. 137–146. [Online]. Available: https://doi.org/10.1145/800070.802186

[49] S. Abiteboul and V. Vianu, "Regular path queries with constraints," in *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ser. PODS '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 122–133. [Online]. Available: https://doi.org/10.1145/263661.263676

[50] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi, "Rewriting of regular expressions and regular path queries," *Journal of Computer and System Sciences*, vol. 64, no. 3, pp. 443–465, 2002.

[51] P. Barceló Baeza, "Querying graph databases," in *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, ser. PODS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 175–188. [Online]. Available: https://doi.org/10.1145/2463664.2465216

[52] J. Clark and S. DeRose, "XML path language (XPath) version 1.0," W3C Recommendation 16 November 1999.

[53] M. Marx, "Conditional XPath," *ACM Trans. Database Syst.*, vol. 30, no. 4, pp. 929–959, 2005.

[54] G. Gottlob, C. Koch, and R. Pichler, "Efficient algorithms for processing XPath queries," *ACM Trans. Database Syst.*, vol. 30, no. 2, pp. 444–491, 2005.

[55] J. Robie, M. Dyck, and J. Spiegel, "XML path language (XPath) 3.1," W3C Recommendation 21 March 2017.

[56] Rust-Itertools, "rust-itertools/itertools." [Online]. Available: https://github.com/rust-itertools/itertools

[57] Rayon-Rs, "Rayon-rs/rayon: Rayon: A data parallelism library for rust." [Online]. Available: https://github.com/rayon-rs/rayon/

[58] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: a unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016. [Online]. Available: http://doi.acm.org/10.1145/2934664

[59] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[60] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 439–455.

[61] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard, "Differential dataflow," in *CIDR*, 2013.

[62] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on job scheduling strategies for parallel processing*. Springer, 2003, pp. 44–60.

[63] S. Ojagh, S. Saeedi, and S. H. Liang, "A person-to-person and person-to-place covid-19 contact tracing system based on ogc indoorgml," *ISPRS International Journal of Geo-Information*, vol. 10, no. 1, p. 2, 2021.

[64] J. Gonzalez, Y. Low, and H. Gu, "Powergraph: Distributed graph-parallel computation on natural graphs," in *OSDI'12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, 2012, pp. 17–30. [Online]. Available: https://www.usenix.org/system/files/conference/osdi12/osdi12-final-167.pdf

[65] A. Aghasadeghi, V. Z. Moffitt, S. Schelter, and J. Stoyanovich, "Zooming out on an evolving graph," in *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, A. Bonifati, Y. Zhou, M. A. V. Salles, A. Böhm, D. Olteanu, G. H. L. Fletcher, A. Khan, and B. Yang, Eds. OpenProceedings.org, 2020, pp. 25–36. [Online]. Available: https://doi.org/10.5441/002/edbt.2020.04

[66] "Neo4j: What is a graph database?" https://neo4j.com/developer/graph-database/#property-graph, [Online; accessed 18-July-2017].