

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
apa374@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
```

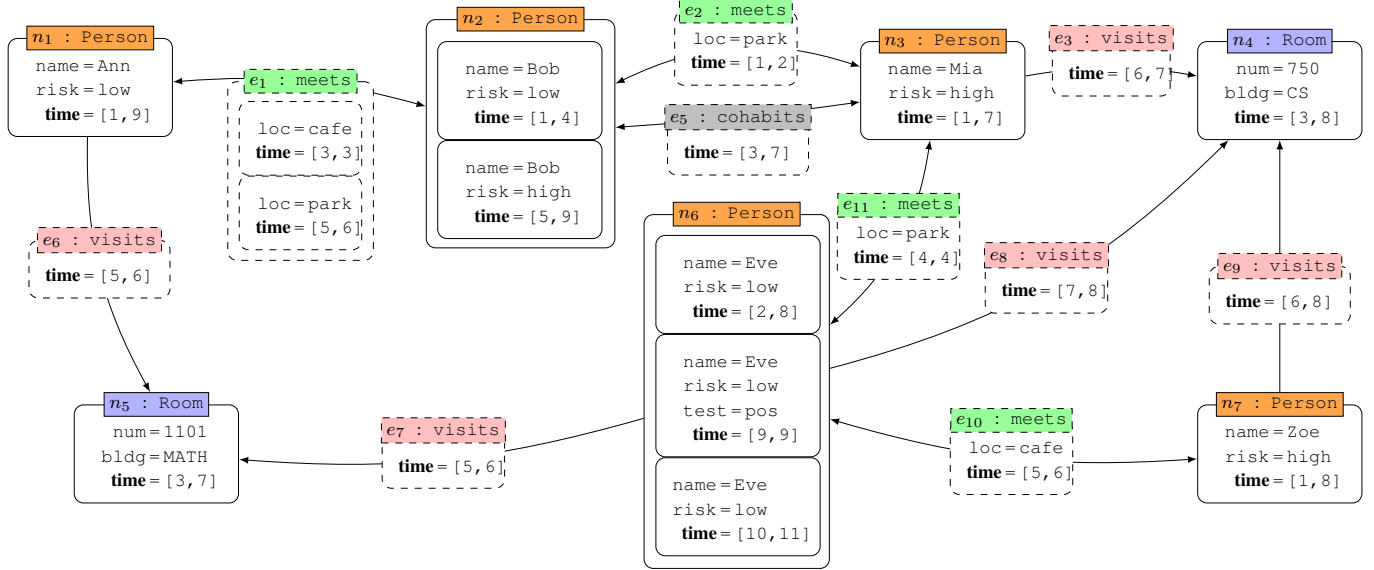


Fig. 1. A TPG used for contact tracing. The graph contains two types of nodes, **Person** and **Room**, and three types of edges: bi-directional edges **meets** and **cohabits**, and directed edge **visits**. **Person** nodes have properties **name**, **risk** ('high' or 'low') of complications, and **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 the Appendix. 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 [19], which includes representation of time [20]–[22], semantics of temporal models [23], temporal

algebras [24], and access methods [25]. Results of some of this work are part of the SQL:2011 standard [26].

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 [27]–[37]. 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. [38]–[40] 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 [41]. 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], [41] 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], [42]) 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 [43].

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 $V \cap E = \emptyset$;
- $\rho : E \rightarrow (N \times N)$ is a function that maps an edge to its source and destination nodes;
- $\lambda : (N \cup E) \rightarrow Lab$ is a function that maps a node or an edge to its label;
- $\xi : (N \cup E) \times \Omega \rightarrow \{true, false\}$ 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 $\xi(v_2, t) = true$.
- $\sigma : (N \cup E) \times Prop \times \Omega \rightarrow Val$ 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, \dots, 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 $\Omega = \{1, \dots, 11\}$, although any set of consecutive natural numbers containing Ω can serve as the temporal domain of this TPG, for example the set $\{0, \dots, 15\}$. The TPG is a multi-graph: 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 the Appendix.

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 [41], PGQL [44], 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 [41] and PGQL [44], these languages can accommodate such temporal graph patterns as well. These languages play a fundamental role in the ongoing graph query language standardization effort [45], 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 [41] and PGQL [44] 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	x	x_time
n_1	1	n_2	1
...
n_1	9	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 $\mathbf{x} \mapsto n_1, \mathbf{x_time} \mapsto 1$ 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	x	x_time	x	x_time
n_1	1	n_2	1	n_6	2
...
n_1	9	n_2	4	n_6	11

Observe that the binding $\mathbf{x} \mapsto n_2, \mathbf{x_time} \mapsto 4$ 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], [41], [44], 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 [46]. In fact, there is even a fixed query Q for which this negative result holds [46]. This means that the problem of computing, given a graph G as input, the answer to Q over G is intractable in data complexity [47].

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], [41], [44], 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

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

which, in the formal TPGs notation (see Definition III.1), represents the fact that $\rho(e) = (n, n')$, $\xi(n, t) = true$, $\xi(e, t) = true$, and $\xi(n', t) = true$. 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 infectious 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	x	x_time
n_3	4	n_7	6
n_3	7	n_7	7
n_7	5	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], [41], [44] 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], [48]–[50] 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], [48]–[54]. 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 the Appendix.

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:

$$\text{path} ::= \text{test} \mid \text{axis} \mid (\text{path}/\text{path}) \mid (\text{path} + \text{path}) \mid \text{path}[n, m] \mid \text{path}[n, _]$$
 (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, `(path/path)` is used for the concatenation of two TRPQs, `(path + path)` allows for the disjunction of two TRPQs, `path[n, m]` allows `path` to be repeated a number of times that is between n and m , whereas `path[n, _]` only imposes a lower bound of at least n repetitions of expression `path`. The Kleene star `path*` can be expressed as `path[0, _]`, and the expression `path[_, n]` is equivalent to `path[0, n]`.

Conditions on temporal objects are defined by the grammar:

$$\text{test} ::= \text{Node} \mid \text{Edge} \mid \ell \mid p \mapsto v \mid < k \mid \exists \mid$$

$$(?path) \mid (\text{test} \vee \text{test}) \mid (\text{test} \wedge \text{test}) \mid (\neg \text{test})$$
 (3)

where $\ell \in \text{Lab}$, $p \in \text{Prop}$, $v \in \text{Val}$, 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 $(?path)$, 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*:

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

Operators **F**, **B** move structurally in a TPG: **F** moves forward in the direction of an edge, and **B** moves backward in the reverse direction of an edge. Operators **N**, **P** move temporally in a TPG: **N** moves to the next time point, and **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 **P** and **N** in `NavL[PC,NOI]`, respectively, while structural navigation operators `BWD` and `FWD` in the practical query language correspond to the operators **B** and **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:

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

Observe that $(\text{Node} \wedge \text{Person} \wedge \text{test} \mapsto \text{pos})$ 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, $(\text{Node} \wedge \text{Person} \wedge \text{test} \mapsto \text{pos})$ is used to represent the expression `(x:Person {test = 'pos'})`. Moreover, temporal navigation operator **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 $(\text{Node} \wedge \exists)$ 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 exist 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 **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 $\llbracket \text{path} \rrbracket_G$, 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 $\text{src}(e) = v_1$ and $\text{tgt}(e) = v_2$ whenever $\rho(e) = (v_1, v_2)$, and assume that $\text{PTO}(G) = (N \cup E) \times \Omega \times$

$(N \cup E) \times \Omega$. Then the evaluation of the axes in grammar (2) is defined as:

$$\begin{aligned} \llbracket \mathbf{F} \rrbracket_G &= \{(v, t, e, t) \in \text{PTO}(G) \mid \text{src}(e) = v\} \cup \{(e, t, v, t) \in \text{PTO}(G) \mid \text{tgt}(e) = v\} \\ \llbracket \mathbf{B} \rrbracket_G &= \{(v, t, e, t) \in \text{PTO}(G) \mid \text{tgt}(e) = v\} \cup \{(e, t, v, t) \in \text{PTO}(G) \mid \text{src}(e) = v\} \\ \llbracket \mathbf{N} \rrbracket_G &= \{(o, t_1, o, t_2) \in \text{PTO}(G) \mid t_2 = t_1 + 1\} \\ \llbracket \mathbf{P} \rrbracket_G &= \{(o, t_1, o, t_2) \in \text{PTO}(G) \mid t_2 = t_1 - 1\} \end{aligned}$$

Moreover, assuming that path, path₁ and path₂ are expressions in NavL[PC,NOI], we have that:

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

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: $\llbracket \text{test} \rrbracket_G = \{(o, t, o, t) \in \text{PTO}(G) \mid (o, t) \models \text{test}\}$. 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 $(o, t) \models \text{test}$. Formally, this is recursively defined as follows (omitting the usual semantics for Boolean connectives):

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

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 $\text{can}(\cdot)$ is a canonical translation from an ITPG into an equivalent TPG, we have that: $\llbracket \text{path} \rrbracket_I = \llbracket \text{path} \rrbracket_{\text{can}(I)}$.

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 $\text{Eval}(\mathcal{G}, \mathcal{L})$ as the problem of verifying whether $(o, t, o', t') \in \llbracket \text{path} \rrbracket_G$, 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 $\text{Eval}(\mathcal{G}, \mathcal{L})$ for different fragments \mathcal{L} of $\text{NavL}[\text{PC}, \text{NOI}]$, we can understand how the use of the operators in $\text{NavL}[\text{PC}, \text{NOI}]$ affects the complexity of the evaluation problem, and which operators are more difficult to implement.

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

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

- 1) $\text{Eval}(\text{TPG}, \text{NavL}[\text{PC}, \text{NOI}])$ and $\text{Eval}(\text{ITPG}, \text{NavL}[\text{PC}])$ can be solved in polynomial time.
- 2) $\text{Eval}(\text{ITPG}, \text{NavL}[\text{NOI}])$ is Σ_2^P -hard, and $\text{Eval}(\text{ITPG}, \text{NavL}[\text{PC}, \text{NOI}])$ is PSPACE-complete.

The results of this section can guide future implementations of $\text{NavL}[\text{PC}, \text{NOI}]$ over interval-timestamped TPGs. The main insight is that, while $\text{Eval}(\text{ITPG}, \text{NavL}[\text{NOI}])$ and $\text{Eval}(\text{ITPG}, \text{NavL}[\text{PC}, \text{NOI}])$ 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 [41], 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 [41]. 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 $\text{NavL}[\text{PC}, \text{NOI}]$ that includes all queries of Section IV over interval-timestamped TPGs. We use Rust and the *Itertools* library [55], which efficiently implements dataflow operators, supports lazy evaluation of expressions, and collects data only when necessary. For multi-threaded implementation, we use *Rayon-Rs* [56], an interface over dataflow operators. Our algorithms can be implemented using any system that supports the dataflow model, such as Apache Spark [57], Apache Flink [58], Timely [59] and Differential dataflow [60].

We represent a TPG as a pair of interval-timestamped temporal relations $\text{Nodes}(\text{id}, \text{label}, \text{properties}, \text{time})$ and $\text{Edges}(\text{id}, \text{src}, \text{tgt}, \text{label}, \text{properties}, \text{time})$, 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], [50]. 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 [24] 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 $\mathbf{x} = n_1$, $\mathbf{z} = e_1$, $\mathbf{y} = n_2$, with $\mathbf{x_time} = \mathbf{y_time} = \mathbf{z_time} = 5$, 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 \mathbf{z} to the time immediately before \mathbf{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 \mathbf{z} that person \mathbf{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 \mathbf{x} and \mathbf{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**, $\mathbf{x_time}$ may or may not be the same as $\mathbf{z_time}$, 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 checked 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 [61]. 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. [62] 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.

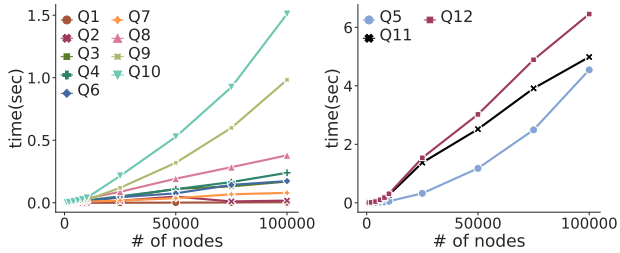


Fig. 2. Effect of graph size on query execution time, on G1-G10.

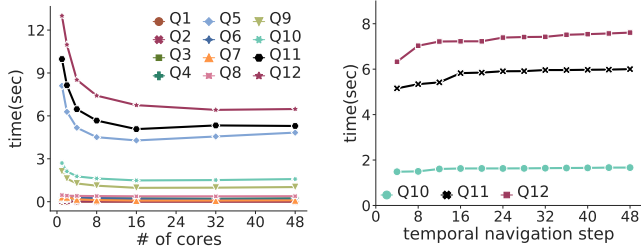


Fig. 3. Effect of parallelism on G10. Fig. 4. Effect of temp. nav. on G10.

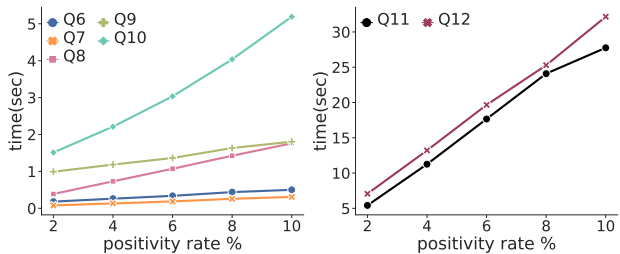


Fig. 5. Effect of positivity rate on query execution time, on G10.

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 $\times 10$ nodes and $\times 100$ 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. We observe a linear relationship between positivity rate and execution time for all queries.

In our final experiment, 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 the number of temporal navigation steps increases the execution time. This increase is initially linear, but plateaus when m reaches 16, because of the cumulative effect of increasing m .

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. Another natural direction is to incorporate our methods into existing graph processing systems like GraphX [63], Portal [64] or Neo4j [65], and to investigate a range of systems questions, including the impact of different object timestamping strategies, temporal coalescing strategies, and indexing methods on performance.

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. Barceló, 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. McGlohan, 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 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] L. Liu and M. T. Zsu, *Encyclopedia of Database Systems*, 1st ed. Boston, MA: Springer Publishing Company, Incorporated, 2009.
- [20] 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>
- [21] 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>
- [22] 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>
- [23] 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>
- [24] 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>
- [25] 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>
- [26] 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>
- [27] 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>
- [28] 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>
- [29] A. Ferreira, "Building a reference combinatorial model for MANETs," *IEEE Network*, vol. 18, no. 5, pp. 24–29, 2004.
- [30] 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.
- [31] 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>
- [32] —, "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>
- [33] 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.
- [34] 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.

- [35] 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>
- [36] 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?doi=2588555.2612184>
- [37] 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.
- [38] 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>
- [39] 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>
- [40] 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>
- [41] 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>
- [42] 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>
- [43] 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.
- [44] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi, "Pqql: 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>
- [45] Association of ISO Graph Query Language Proponents, "GQL standard," 2020, <https://www.gqlstandards.org>.
- [46] L. Libkin, W. Martens, and D. Vrgoc, "Querying graphs with data," *J. ACM*, vol. 63, no. 2, pp. 14:1–14:53, 2016.
- [47] 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>
- [48] 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>
- [49] 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.
- [50] 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>
- [51] J. Clark and S. DeRose, "XML path language (XPath) version 1.0," W3C Recommendation 16 November 1999.
- [52] M. Marx, "Conditional XPath," *ACM Trans. Database Syst.*, vol. 30, no. 4, pp. 929–959, 2005.
- [53] 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.
- [54] J. Robie, M. Dyck, and J. Spiegel, "XML path language (XPath) 3.1," W3C Recommendation 21 March 2017.
- [55] Rust-Itertools, "rust-iterools/iterools." [Online]. Available: <https://github.com/rust-iterools/iterools>
- [56] Rayon-Rs, "Rayon-rs/rayon: Rayon: A data parallelism library for rust." [Online]. Available: <https://github.com/rayon-rs/rayon/>
- [57] 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>
- [58] 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.
- [59] 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.
- [60] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard, "Differential dataflow," in *CIDR*, 2013.
- [61] 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.
- [62] 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.
- [63] 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>
- [64] 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>
- [65] "Neo4j: What is a graph database?" <https://neo4j.com/developer/graph-database/#property-graph>, [Online]; accessed 18-July-2017].
- [66] J. F. Allen, "Maintaining Knowledge about Temporal Intervals," *Communications of the ACM*, vol. 26, no. 11, pp. 832–843, 1983.
- [67] M. Böhlen, *Temporal Coalescing*. Boston, MA: Springer US, 2009, pp. 2932–2936. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-39940-9_388
- [68] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.
- [69] P. Berman, M. Karpinski, L. L. Larmore, W. Plandowski, and W. Rytter, "On the complexity of pattern matching for highly compressed two-dimensional texts," in *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching*, ser. CPM '97. Berlin, Heidelberg: Springer-Verlag, 1997, p. 40–51.
- [70] L. J. Stockmeyer and A. R. Meyer, "Word problems requiring exponential time (preliminary report)," in *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, ser. STOC '73. New York, NY, USA: Association for Computing Machinery, 1973, p. 1–9. [Online]. Available: <https://doi.org/10.1145/800125.804029>
- [71] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1990.

APPENDIX A
FORMAL DEFINITION OF 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 of natural numbers $\{i \in \mathbb{N} \mid a \leq i \leq b\}$ (that is, to specify this interval, we just need to mention its starting point a and its ending point b). Using Allen's interval algebra [66], given two intervals $[a_1, b_1]$ and $[a_2, b_2]$, we say that $[a_1, b_1]$ occurs during $[a_2, b_2]$ if $a_2 \leq a_1$ and $b_1 \leq b_2$, $[a_1, b_1]$ meets $[a_2, b_2]$ if $b_1 + 1 = b_2$, and $[a_1, b_1]$ is before $[a_2, b_2]$ if $b_1 + 1 < a_2$.

A finite family \mathcal{F} of intervals is said to be *coalesced* [67] if $\mathcal{F} = \{[a_1, b_1], \dots, [a_n, b_n]\}$ and $[a_j, b_j]$ is before $[a_{j+1}, b_{j+1}]$ for every $j \in \{1, \dots, n-1\}$. For example, $\mathcal{F}_1 = \{[1, 4], [6, 8]\}$ is coalesced, while $\mathcal{F}_2 = \{[1, 2], [3, 4], [6, 8]\}$ is not, because $[1, 2]$ meets $[3, 4]$. The set of all finite coalesced families of intervals is denoted by FC. Observe that $\emptyset \in \text{FC}$. Moreover, given $\mathcal{F}_1, \mathcal{F}_2 \in \text{FC}$, family \mathcal{F}_1 is said to be contained in family \mathcal{F}_2 , denoted by $\mathcal{F}_1 \sqsubseteq \mathcal{F}_2$, if for every $[a_1, b_1] \in \mathcal{F}_1$, there exists $[a_2, b_2] \in \mathcal{F}_2$ such that $[a_1, b_1]$ occurs during $[a_2, b_2]$. Finally, given an interval Ω , we use $\text{FC}(\Omega)$ to denote the set of all families $\mathcal{F} \in \text{FC}$ such that for every $[a, b] \in \mathcal{F}$, it holds that $[a, b]$ occurs during Ω .

Given an interval $[a, b]$ and $v \in \text{Val}$, the pair $(v, [a, b])$ is a *valued* interval. A finite family \mathcal{F} of valued intervals is said to be coalesced if $\mathcal{F} = \{(v_1, [a_1, b_1]), \dots, (v_n, [a_n, b_n])\}$ and for every $j \in \{1, \dots, n-1\}$, either $[a_j, b_j]$ is before $[a_{j+1}, b_{j+1}]$, or $[a_j, b_j]$ meets $[a_{j+1}, b_{j+1}]$ and $v_j \neq v_{j+1}$. For example, $\mathcal{F}_1 = \{(v, [1, 2]), (v, [5, 8])\}$ and $\mathcal{F}_2 = \{(v, [1, 2]), (w, [3, 4])\}$ are both coalesced (assuming that $v \neq w$). On the other hand, $\mathcal{F}_3 = \{(v, [1, 2]), (v, [3, 4])\}$ is not coalesced because $[1, 2]$ meets $[3, 4]$ and these intervals have the same value in \mathcal{F}_3 . Moreover, the set of all finite coalesced families of valued intervals is denoted by vFC. Finally, given an interval Ω , we use $\text{vFC}(\Omega)$ to denote the set of all families $\mathcal{F} \in \text{vFC}$ such that for every $(v, [a, b]) \in \mathcal{F}$, it holds that $[a, b]$ occurs during Ω .

With these ingredients, we can introduce the notion of interval-timestamped temporal property graph.

Definition A.1. An interval-timestamped temporal property graph (ITPG) is a tuple $I = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$, where N , E , ρ and λ are defined exactly as for the case of TPGs (see Definition III.1). Moreover,

- Ω is an interval of \mathbb{N} ;
- $\xi : (N \cup E) \rightarrow \text{FC}(\Omega)$ is a function that maps a node or an edge to a finite coalesced family of intervals occurring during Ω ;
- $\sigma : (N \cup E) \times \text{Prop} \rightarrow \text{vFC}(\Omega)$ is a function that maps a node or an edge, and a property name to a finite coalesced family of valued intervals occurring during Ω .

In addition, I satisfies the following conditions:

- If $\rho(e) = (n_1, n_2)$, then $\xi(e) \sqsubseteq \xi(n_1)$ and $\xi(e) \sqsubseteq \xi(n_2)$.
- There exists a finite set of pairs $(o, p) \in (N \cup E) \times \text{Prop}$ such that $\sigma(o, p) \neq \emptyset$. Moreover, if $\sigma(o, p) = \{(v_1, [a_1, b_1]), \dots, (v_n, [a_n, b_n])\}$, then $\{[a_1, b_1], \dots, [a_n, b_n]\} \sqsubseteq \xi(o)$.

In the definition of an ITPG, given a node or edge o , function ξ indicates the time intervals where o exists, and function σ indicates the values of a property p for o . More precisely, if $\sigma(o, p) = \{(v_1, [a_1, b_1]), \dots, (v_n, [a_n, b_n])\}$, then the value of property p for o is v_j in every time point in the interval $[a_j, b_j]$ ($1 \leq j \leq n$). Moreover, observe that two additional conditions are imposed on I , which enforce that an ITPG conceptually corresponds to a finite sequence of valid conventional property graphs. In particular, as was the case for TPGs, an edge can only exist at a time when both of the nodes it connects exist, and a property can only take on a value at a time when the corresponding node or edge exists. For instance, assume that $I = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$ is an ITPG corresponding to our running example in Figure 1. Then, we have that $\Omega = [1, 11]$, $\xi(n_2) = \{[1, 9]\}$, $\xi(n_3) = \{[1, 7]\}$ and $\xi(e_2) = \{[1, 2]\}$, so that $\xi(e_2) \sqsubseteq \xi(n_2)$ and $\xi(e_2) \sqsubseteq \xi(n_3)$. Moreover, for the property risk, we have that $\sigma(n_2, \text{risk}) = \{(\text{low}, [1, 4]), (\text{high}, [5, 9])\}$.

We conclude this section by observing that there is a one-to-one correspondence between TPGs and ITPGs. On the one hand, each TPG can be transformed in polynomial-time into a ITPG, by putting in the same interval consecutive time points with the same values. On the other hand, each ITPG can be transformed in exponential-time into a TPG, by replacing each interval by the set of time points represented by it.

APPENDIX B
FORMAL DEFINITION OF SOME FRAGMENTS OF NavL[PC,NOI]

Removing numerical occurrence indicators. We start by considering a restriction of our query language in which numerical occurrence indicators are not allowed. Formally, this means that grammar (2) is replaced by:

$$\text{path} ::= \text{test} \mid \text{axis} \mid (\text{path}/\text{path}) \mid (\text{path} + \text{path}) \quad (5)$$

The resulting language is called NavL[PC].

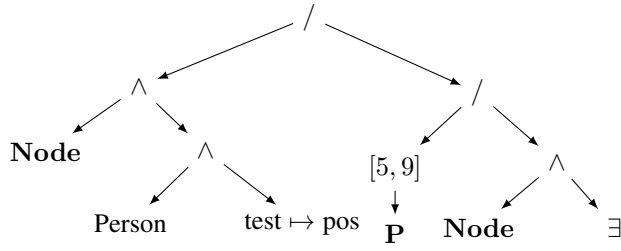


Fig. 6. A parsing tree of NavL[PC,NOI]-expression $\text{path} = (\mathbf{Node} \wedge \text{Person} \wedge \text{test} \mapsto \text{pos})/\mathbf{P}[5,9]/(\mathbf{Node} \wedge \exists)$.

Removing path conditions. Consider a second restriction of our language in which there are no path conditions. Formally, this means that instead of grammar (3), we use:

$$\text{test} ::= \mathbf{Node} \mid \mathbf{Edge} \mid \ell \mid p \mapsto v \mid < k \mid \exists \mid (\text{test} \vee \text{test}) \mid (\text{test} \wedge \text{test}) \mid (\neg \text{test}) \quad (6)$$

The resulting language is called NavL[NOI].

Allowing numerical occurrence indicators only in the axes. Consider a grammar for tests as in (6), where path conditions are not allowed, and a grammar for path expressions where numerical occurrence indicators are only used in the axes:

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

The resulting language is called NavL[ANOI], where ANOI refers to numerical occurrence indicators used only in the axes.

APPENDIX C PROOF OF THEOREM V.1

A. Eval(TPG, NavL[PC,NOI]) can be solved in polynomial time

In this section, we show that NavL[PC, NOI] can be evaluated in polynomial time, considering a computational model where accessing the distinct elements of a TPG takes time $O(1)$. More precisely, for a TPG $G = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$, it is assumed that the following operations can be performed in time $O(1)$: given $e \in E$ and $n_1, n_2 \in N$, check whether $\rho(e) = (n_1, n_2)$; given $e \in E$, compute $\text{src}(e)$ and $\text{tgt}(G)$; given $o \in (N \cup E)$ and $\ell \in \text{Lab}$, check whether $\lambda(o) = \ell$; given $o \in (N \cup E)$ and $t \in \Omega$, check whether $\xi(o, t) = \text{true}$; and given $o \in (N \cup E)$, $p \in \text{Prop}$ and $v \in \text{Val}$, check whether $\sigma(o, p, t) = v$. Moreover, we use notation $\|\text{path}\|$ for the length of NavL[PC, NOI]-expression path as an input string over an appropriate alphabet. Then, it is possible to prove the following.

Theorem C.1. *There exists an algorithm that, given a temporal property graph G and a NavL[PC, NOI]-expression path , computes $\llbracket \text{path} \rrbracket_G$ in time $\tilde{O}(\|\text{path}\|^2 \cdot |\Omega|^2 \cdot (|N| + |E|)^2)$.*

In the rest of this section, we describe the polynomial-time algorithm in the statement of Theorem C.1. Let $G = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$ be a TPG and path be an expression in NavL[PC,NOI]. Moreover, assume that $M = |\Omega| \cdot (|N| + |E|)$ is the number of distinct (existing or non-existing) temporal objects in G . The algorithm constructs a parsing tree of path , where each node is associated with an operator in NavL[PC,NOI], and then, by using a bottom-up approach, computes, for each node u , the set of tuples (o, t, o', t') that satisfy the operator labeling u . For example, a parsing tree of NavL[PC,NOI]-expression $(\mathbf{Node} \wedge \text{Person} \wedge \text{test} \mapsto \text{pos})/\mathbf{P}[5,9]/(\mathbf{Node} \wedge \exists)$ is shown in Figure 6. The algorithm starts by evaluating the leaves: $\llbracket \mathbf{Node} \rrbracket_G$, $\llbracket \text{Person} \rrbracket_G$, $\llbracket \text{test} \mapsto \text{pos} \rrbracket_G$, $\llbracket \mathbf{P} \rrbracket_G$ and $\llbracket \exists \rrbracket_G$, according to the semantics defined in Section V-A, and then it combines the resulting tables by using the operators \wedge , $[5, 9]$ and $/$ in the order specified by the parsing tree. For instance, in the right-hand side of the tree, once $\llbracket \mathbf{P} \rrbracket_G$, $\llbracket \mathbf{Node} \rrbracket_G$ and $\llbracket \exists \rrbracket_G$ have been computed, the algorithm continues by constructing $\llbracket \mathbf{Node} \wedge \exists \rrbracket_G$, followed by $\llbracket \mathbf{P}[5, 9] \rrbracket_G$ and then by $\llbracket \mathbf{P}[5, 9]/(\mathbf{Node} \wedge \exists) \rrbracket_G$. Notice that at any given moment, the result of at most $\|\text{path}\|$ nodes is stored in the form of a table, each one with as many pairs of temporal objects as there are available, i.e., with at most M^2 tuples.

We now explain in detail the different components of the algorithm, paying particular attention to the expressions of the form $\text{path}_1[n, m]$ and $\text{path}_2[n, _]$, as they are the most expensive to evaluate in NavL[PC,NOI]. Initially, for each leaf of the parsing tree of path , we have to process either a test \mathbf{Node} , \mathbf{Edge} , ℓ , $p \mapsto v$, \exists or $< k$, or a navigation operator \mathbf{N} , \mathbf{P} , \mathbf{F} , or \mathbf{B} . Basic tests can be evaluated in time $O(M)$ just by considering each tuple of the form $(o, t, o, t) \in \text{PTO}(G)$ and checking in $O(1)$ whether (o, t) satisfies the test. As for navigation operators, each one of them can also be evaluated in time $O(M)$ (recall that the existence of nodes or edges at a given time point is not required in the language). For example, $\llbracket \mathbf{P} \rrbracket_G$ can be

Algorithm 1: COMPUTE_{REPETITION}(G, T_1, n)

Input : A TPG G , a table T_1 such that $T_1 = \llbracket \text{path}_1 \rrbracket_G$ for some NavL[PC,NOI]-expression path_1 , and $n \geq 0$
Output: $\llbracket \text{path}_1[n, n] \rrbracket_G$
if $n = 0$ **then**
 | **return** $\{(o, t, o, t) \in \text{PTO}(G)\}$
else if $n = 1$ **then**
 | **return** T_1
 $n' \leftarrow \lfloor n/2 \rfloor$
 $T_2 \leftarrow \text{COMPUTE}_{\text{REPETITION}}(G, T_1, n')$
 Compute $T_3 = \{(o_1, t_1, o_2, t_2) \mid \exists(o, t) : (o_1, t_1, o, t) \in T_2 \text{ and } (o, t, o_2, t_2) \in T_2\}$ by doing a sort-merge join
 if n is even **then**
 | **return** T_3
 Compute $T_4 = \{(o_1, t_1, o_2, t_2) \mid \exists(o, t) : (o_1, t_1, o, t) \in T_3 \text{ and } (o, t, o_2, t_2) \in T_1\}$ by doing a sort-merge join
 return T_4

constructed just by considering all objects $o \in V \cup E$ and then generating tuples $(o, t, o, t-1)$ such that $t \in \Omega$ and $t-1 \in \Omega$, while $\llbracket \mathbf{F} \rrbracket_G$ can be constructed by considering all edges $e \in E$, and then generating tuples $(\text{src}(e), t, e, t)$ and $(e, t, \text{tgt}(e), t)$ for each $t \in \Omega$.

For each internal node u of the parsing tree of path , we must consider one of the following two cases. Assume first that the label of u is either \wedge, \vee, \neg or $?$, so that u represents a more complex test expression $(\text{test}_1 \wedge \text{test}_2)$, $(\text{test}_1 \vee \text{test}_2)$, $(\neg \text{test}_1)$ or $(? \text{path}_1)$. If u represents test $(\text{test}_1 \wedge \text{test}_2)$, then the algorithm has already computed $T_1 = \llbracket \text{test}_1 \rrbracket_G$ and $T_2 = \llbracket \text{test}_2 \rrbracket_G$. Hence, to construct $\llbracket \text{test}_1 \wedge \text{test}_2 \rrbracket_G$, the algorithm needs to compute the intersection of T_1 and T_2 , which can be done in time $\tilde{O}(M^2)$ by sorting both tables (each of size at most M^2) and iterating with two pointers, one on each table, to see which elements occur in both. Recall that the notation $\tilde{O}(M^2)$ ignores the logarithmic factors, which in this case appear when sorting tables T_1 and T_2 . The case where u represents either $(\text{test}_1 \vee \text{test}_2)$ or $(\neg \text{test}_1)$ can be treated in a similar way. Finally, if u represents test $(? \text{path}_1)$, for each tuple $(o, t, o', t') \in \llbracket \text{path}_1 \rrbracket_G$, we need to include the tuple (o, t, o, t) in the table for u , as $(o, t) \models (? \text{path}_1)$ if and only if $(o, t, o', t') \in \llbracket \text{path}_1 \rrbracket_G$ for some temporal object (o', t') in G . This can be done in time $O(M^2)$ as $\llbracket \text{path}_1 \rrbracket_G$ contains at most M^2 tuples.

Assume now that the label of u is either $/, +, [n, m]$ or $[n, _]$, so that u represents a more complex path expression $(\text{path}_1/\text{path}_2)$, $(\text{path}_1 + \text{path}_2)$, $\text{path}_1[m, n]$ or $\text{path}_1[m, _]$. If u represents expression $(\text{path}_1/\text{path}_2)$, then the algorithm has already computed $T_1 = \llbracket \text{path}_1 \rrbracket_G$ and $T_2 = \llbracket \text{path}_2 \rrbracket_G$. Hence, to construct $\llbracket \text{path}_1/\text{path}_2 \rrbracket_G$, the algorithm just need to sort T_1 by the third and fourth columns (the second pair of temporal objects), sort T_2 by the first and second column (the first pair of temporal objects), and then join T_1 with T_2 by looking at matching temporal objects on those columns. The overall time for this construction is $\tilde{O}(M^2)$, as it corresponds to a sort-merge join on two tables with at most M^2 tuples. If u represents the expression $(\text{path}_1 + \text{path}_2)$, the algorithm computes the union of T_1 and T_2 . If u represents the expression $\text{path}_1[n, m]$, then the algorithm proceeds as follows, assuming that $T_1 = \llbracket \text{path}_1 \rrbracket_G$ has already been computed. Given that $\text{path}_1[n, m]$ is equivalent to the expression $\text{path}_1[n, n]/\text{path}_1[0, m-n]$, the procedure first runs Algorithm 1 COMPUTE_{REPETITION}(G, T_1, n) to compute $\llbracket \text{path}_1[n, n] \rrbracket_G$ in a similar way to the exponentiation by squaring algorithm [68]. If $n = m$, then we are ready in time $\tilde{O}(\|\text{path}_1\| \cdot M^2)$, since the most expensive operation is the sort-merge join, which is carried out in time $\tilde{O}(M^2)$ and at most $O(\log(n))$ times, that is, $O(\|\text{path}_1\|)$ times. Otherwise, Algorithm 2 COMPUTE_{INTERVALREPETITION}($G, T_1, T_2, m-n$) is called to compute $\llbracket \text{path}_1[n, n]/\text{path}_1[0, m-n] \rrbracket_G$, where $T_2 = \llbracket \text{path}_1[n, n] \rrbracket_G$ is the result of invoking COMPUTE_{REPETITION}(G, T_1, n). Here, $O(\log(m-n))$ sort-merge joins have to be carried out, which is again $O(\|\text{path}_1\|)$, so this takes a total time of $\tilde{O}(\|\text{path}_1\| \cdot M^2)$.

Finally, if u represents the expression $\text{path}_1[n, _]$, then the computation process is similar to the previous one, assuming that $T_1 = \llbracket \text{path}_1 \rrbracket_G$ has already been computed. As before, we act as if we have to compute the table for another, but equivalent, expression, $\text{path}_1[n, n]/\text{path}_1[0, M^2]$, which is done by first computing $T_2 = \text{COMPUTE}_{\text{REPETITION}}(G, T_1, n)$, and then invoking COMPUTE_{INTERVALREPETITION}(G, T_1, T_2, M^2). This takes time $\tilde{O}(\|\text{path}_1\| \cdot M^2)$ as the sort-merge join is carried out in time $\tilde{O}(M^2)$, and $O(\log(n) + \log(M^2))$ such joins need to be computed, that is $\tilde{O}(\|\text{path}_1\|)$ such joins. Notice that $\llbracket \text{path}_1[0, _] \rrbracket_G = \llbracket \text{path}_1[0, M^2] \rrbracket_G$ because: (a) $\llbracket \text{path}_1[0, k] \rrbracket_G \subseteq \llbracket \text{path}_1[0, k+1] \rrbracket_G$ for every $k \geq 0$; (b) $|\llbracket \text{path}_1[0, k] \rrbracket_G| \leq M^2$ for every $k \geq 0$; and (c) if $\llbracket \text{path}_1[0, k] \rrbracket_G = \llbracket \text{path}_1[0, k+1] \rrbracket_G$, then $\llbracket \text{path}_1[0, k] \rrbracket_G = \llbracket \text{path}_1[0, k'] \rrbracket_G$ for every $k' > k$.

In summary, the table associated to each node of the parsing tree of path can be computed in time $\tilde{O}(\|\text{path}\| \cdot M^2)$. Given that there are at most $O(\|\text{path}\|)$ such nodes, the total computation time is $\tilde{O}(\|\text{path}\|^2 \cdot M^2)$, that is, $\tilde{O}(\|\text{path}\|^2 \cdot |\Omega|^2 \cdot (|N| + |E|)^2)$. This concludes the proof of Theorem C.1.

Algorithm 2: COMPUTEINTERVALREPETITION(G, T_1, T_2, n)

Input : A TPG G , a table T_i such that $T_i = \llbracket \text{path}_i \rrbracket_G$ for some NavL[PC,NOI]-expression path_i for $i = 1, 2$, and $n > 0$

Output: $\llbracket \text{path}_1 / \text{path}_2 \rrbracket_{[0, n]}_G$

if $n = 1$ **then**

 | **return** $T_1 \cup T_2$

$n' \leftarrow \lfloor n/2 \rfloor$

$T_3 \leftarrow \text{COMPUTEINTERVALREPETITION}(G, T_1, T_2, n')$

Compute $T_4 = \{(o_1, t_1, o_2, t_2) \mid \exists(o, t) : (o_1, t_1, o, t) \in T_3 \text{ and } (o, t, o_2, t_2) \in T_3\}$ by doing a sort-merge join

$T_5 \leftarrow T_3 \cup T_4$

if n is even **then**

 | **return** T_5

Compute $T_6 = \{(o_1, t_1, o_2, t_2) \mid \exists(o, t) : (o_1, t_1, o, t) \in T_5 \text{ and } (o, t, o_2, t_2) \in T_2\}$ by doing a sort-merge join

return $T_5 \cup T_6$

B. Eval(ITPG, NavL[PC]) can be solved in polynomial time

First of all notice that basic tests such as **Node**, **Edge**, $\ell, p \mapsto v$, $< k$ and \exists can be checked in $O(1)$, so in absence of numerical occurrence indicators, checking a test is equivalent to checking the satisfaction of a Boolean formula on a given valuation, which can be done efficiently in time $O(n)$, where n is the size of the formula. We will assume in the following then that we have a linear-time function $\text{CHECKTESTNOPC}(C, (o, t), \text{test})$ that takes as input an ITPG C , a temporal object (o, t) in C and a test expression test in NavL[PC], and returns *true* if $(o, t) \models \text{test}$ in C , and *false* otherwise.

To show that $\text{Eval}(\text{ITPG}, \text{NavL}[\text{PC}])$ is in PTIME, we present a polynomial-time procedure in Algorithm 3, called $\text{TUPLEEVALSOLVEONLYPC}$, that, given an ITPG C , a tuple representing a pair of temporal objects (o_1, t_1, o_2, t_2) and an expression r in NavL[PC], checks whether $(o_1, t_1, o_2, t_2) \in \llbracket r \rrbracket_C$. In what follows, we show that Algorithm 3 works in polynomial time.

Notice first that we do not directly return the result. Instead, we first look at a hashing table that stores previously stored results, and only if this was not previously computed, we compute the result for the input, and store it in the table before returning the value. We employ this to avoid an exponential number of calls when recursively calling the algorithm. This is possible since, in absence of numerical occurrence indicators, navigation is done at most one step at a time. Thus, if (o_2, t_2) is a temporal object reached from (o_1, t_1) using an expression r , then $|t_1 - t_2|$ is at most the number of symbols **N** and **P** occurring in r . Hence, if $\|r\|$ is the length of expression r , then there are at most $O(\|r\| \cdot |N \cup E|)$ temporal objects that we will need to consider for this call, which means, at most $O(\|r\|^2 \cdot |N \cup E|^2)$ tuples representing pairs of temporal objects. Hence, given that there are at most $\|r\|$ sub-expressions of r that can be reached in the tree decomposition of r , we need to store at most $O(\|r\|^3 \cdot |N \cup E|^2)$ different results for $\text{TUPLEEVALSOLVEONLYPC}$.

The rest of the algorithm is quite straightforward, and it considers the case when the result has not been precomputed. If r is a temporal navigation operator, then by the definitions of **N** and **P**, a single temporal object can be reached, $(o_1, t_1 + 1)$ or $(o_1, t_1 - 1)$, respectively, so we check that (o_2, t_2) is equal to that respective temporal object. This can be easily done in $O(1)$. If r is a spatial navigation operator, then we have to look at the mapping from edges to source and destination nodes, ρ , to determine the set of objects that can be reached. If o is an edge, and we move forward, we look for its destination node, if we move backward, for its source node. If o is a node, and we move forward, we are looking for the edges that have o as their source, and if we move backward, then we look for those who have o as their destination. The whole process can be done in time $O(\|\rho\|)$, which is $O(\|C\|)$. When r is testing a condition, we just call the previously mentioned algorithm CHECKTESTNOPC to check whether $(o, t) \models r$ in C . This last base case can be done in time $O(\|r\|)$.

When r is of the form $(r_1 + r_2)$, where r_1 and r_2 are expressions in NavL[PC], we have that $(o_1, t_1, o_2, t_2) \in \llbracket (r_1 + r_2) \rrbracket_C$ if and only if $(o_1, t_1, o_2, t_2) \in \llbracket r_1 \rrbracket_C$ or $(o_1, t_1, o_2, t_2) \in \llbracket r_2 \rrbracket_C$, so it suffices that the call to $\text{TUPLEEVALSOLVEONLYPC}$ with any of inputs r_1 or r_2 returns *true*, and this can be easily checked by the algorithm. The last case is when r is of the form (r_1 / r_2) where r_1 and r_2 are expressions in NavL[PC], where we have that $(o_1, t_1, o_2, t_2) \in \llbracket r \rrbracket_C$ if there exists a temporal object (o, t) such that $(o_1, t_1, o, t) \in \llbracket r_1 \rrbracket_C$ and $(o, t, o_2, t_2) \in \llbracket r_2 \rrbracket_C$. We know that in absence of numerical occurrence indicators, t will be at distance at most $\|r_1\|$ from t_1 and at most $\|r_2\|$ from t_2 , since one can move only as many times as there are **N** and **P** symbols in the respective formulas, so this gives us a polynomial-size set from which we can extract candidates to satisfy this condition.

Finally, notice that at every call to $\text{TUPLEEVALSOLVEONLYPC}(C, (o_1, t_1, o_2, t_2), r)$, we either already have computed the value for the key (o_1, t_1, o_2, t_2, r) , in which case we can give an answer immediately, or we are computing a new value to store in the hash table H , which has size bounded by $O(\|r\|^3 \cdot |N \cup E|^2)$. In any case, since the most expensive step performs

Algorithm 3: TUPLEEVALSOLVEONLYPC($C, (o_1, t_1, o_2, t_2), r$)

Input : An ITPG $C = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$, an expression r in NavL[PC] and a pair of temporal objects $(o_1, t_1, o_2, t_2) \in \text{PTO}(C)$

Output : *true* if $(o_1, t_1, o_2, t_2) \in \llbracket r \rrbracket_C$, and *false* otherwise

Initialization: A global variable H initially empty, storing a hash table with the currently computed values for TUPLEEVALSOLVEONLYPC.

```
1 if  $(o_1, t_1, o_2, t_2, r)$  is a key in hash table  $H$  then
2 |   return  $H[(o_1, t_1, o_2, t_2, r)]$ 
3 if  $r = \mathbf{N}$  then
4 |    $A \leftarrow (o_1 = o_2 \text{ and } t_2 = t_1 + 1)$ 
5 else if  $r = \mathbf{P}$  then
6 |    $A \leftarrow (o_1 = o_2 \text{ and } t_2 = t_1 - 1)$ 
7 else if  $r = \mathbf{F}$  then
8 |    $A \leftarrow (t_1 = t_2 \text{ and } ((o_1 \in E \text{ and } o_2 = \text{tgt}(o_1)) \text{ or } (o_2 \in E \text{ and } o_1 = \text{src}(o_2))))$ 
9 else if  $r = \mathbf{B}$  then
10 |   $A \leftarrow (t_1 = t_2 \text{ and } ((o_1 \in E \text{ and } o_2 = \text{src}(o_1)) \text{ or } (o_2 \in E \text{ and } o_1 = \text{tgt}(o_2))))$ 
11 else if  $r$  is a test then
12 |   $A \leftarrow ((o_1, t_1) = (o_2, t_2) \text{ and } \text{CHECKTESTNOPC}(C, (o_1, t_1), r))$ 
13 else if  $r = (r_1 + r_2)$  then
14 |   $A \leftarrow \text{TUPLEEVALSOLVEONLYPC}(C, (o_1, t_1, o_2, t_2), r_1)$ 
15 |  | if not  $A$  then
16 |  |    $A \leftarrow \text{TUPLEEVALSOLVEONLYPC}(C, (o_1, t_1, o_2, t_2), r_2)$ 
17 else if  $r = (r_1 / r_2)$  then
18 |   $A \leftarrow \text{false}$ 
19 |  |  $l_1 \leftarrow \llbracket r_1 \rrbracket$ 
20 |  |  $l_2 \leftarrow \llbracket r_2 \rrbracket$ 
21 |  | foreach  $(o, t) \in (N \cup E) \times \{t \in \Omega \mid (|t - t_1| \leq l_1 \wedge |t - t_2| \leq l_2)\}$  do
22 |  | | if  $\text{TUPLEEVALSOLVEONLYPC}(C, (o_1, t_1, o, t), r_1)$  then
23 |  | | | if  $\text{TUPLEEVALSOLVEONLYPC}(C, (o, t, o_2, t_2), r_2)$  then
24 |  | | |    $A \leftarrow \text{true}$ 
25 |  | | |   break
26 Store the value  $A$  for  $\text{TUPLEEVALSOLVEONLYPC}(C, (o_1, t_1, o_2, t_2), r)$  in the hash table  $H$  with key  $(o_1, t_1, o_2, t_2, r)$ 
27 return  $A$ 
```

at most $O(|N \cup E| \cdot \llbracket r \rrbracket)$ recursive calls, we will be getting an answer in time $O(\llbracket r \rrbracket^4 \cdot |N \cup E|^3)$, which is polynomial in the size of the input.

C. Eval(ITPG, NavL[NOI]) is Σ_2^P -hard

Consider the following decision problem called Generalized Subset Sum (G-SUBSET-SUM) which is known to be Σ_2^P -complete [69]:

Problem: G-SUBSET-SUM

Input: Natural numbers vectors u and w of dimensions $\dim(u)$ and $\dim(w)$, respectively, and a positive integer $S \in \mathbb{N}$

Output: *true* if there exists $x \in \{0, 1\}^{\dim(u)}$ such that, for all $y \in \{0, 1\}^{\dim(w)}$, it holds that $x \cdot u + y \cdot w \neq S$, and *false* otherwise.

In this problem, $a \cdot b$ represents the inner product between vectors a and b . Given vectors $u = (u_1, \dots, u_n) \in \mathbb{N}^n$ and $w = (w_1, \dots, w_m) \in \mathbb{N}^m$, and the integer $S \in \mathbb{N}$, the goal is to provide a polynomial-time algorithm that returns a ITPG C , a tuple (o_1, t_1, o_2, t_2) , and an expression r in NavL[NOI] such that $(o_1, t_1, o_2, t_2) \in \llbracket r \rrbracket_C$ if and only if $\exists x \in \{0, 1\}^n \forall y \in \{0, 1\}^m x \cdot u + y \cdot w \neq S$.

Let $M = 2 \cdot \left(\sum_{i=1}^n u_i + \sum_{j=1}^m w_j \right)$, which can be easily computed in polynomial time from u and w . Then C will be the ITPG $C = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$ where $\Omega = [0, 2 \cdot M]$, $N = \{v\}$, $E = \emptyset$, ρ is an empty function, $\lambda(v) = l$, $\xi(v) = \{[0, 2 \cdot M]\}$ and σ is an empty function. In other words, C is a ITPG consisting of only one node existing from time 0 to time $2 \cdot M$, with no edges or properties. The tuple (o_1, t_1, o_2, t_2) in our reduction will be given by $(v, M, v, 2 \cdot M)$. As for the expression

r , it will be defined recursively as follows. First define an expression for each component u_i of u that will represent whether $u_i \cdot x_i$ will be chosen to be u_i or 0:

$$r_{u_i} = \mathbf{N}[u_i, u_i][0, 1].$$

The idea is that the time t of the temporal object that is being reached will store the sum given by $x \cdot u + y \cdot w$, plus M to avoid having negative numbers on the time dimension when testing that the result is different from S (this will be explained in more detail later). Define then an expression for u , representing the sum accumulated by the $\exists x \in \{0, 1\}^n$ part of the problem:

$$r_u = r_{u_1} / \dots / r_{u_n}.$$

We will now use a recursive construction to represent the sum accumulated by the $\forall y \in \{0, 1\}^m$ part of the problem. First define condition $r_{t \neq S+M}$, that represents that the accumulated sum is not S :

$$r_{t \neq S+M} = (\lt S + M \vee \neg \lt S + M + 1)$$

By taking $r_0 := r_{t \neq S+M}$, now recursively define r_{j+1} from r_j , for $j \in \{0, \dots, m-1\}$, as follows:

$$r_{j+1} = (\mathbf{N}[w_{j+1}, w_{j+1}] / r_j / \mathbf{P}[2 \cdot w_{j+1}, 2 \cdot w_{j+1}][2, 2] / \mathbf{N}[2 \cdot w_{j+1}, 2 \cdot w_{j+1}])$$

The formula $r_w = r_m$ will allow to iterate over all the accumulated sums implied by the $\forall y \in \{0, 1\}^m$ part of the problem. Finally, r is defined as follows:

$$r = r_u / r_w / \mathbf{N}[0, _] / (\neg \lt 2 \cdot M)$$

We now prove that $(v, M, v, 2 \cdot M) \in \llbracket r \rrbracket_C$ if and only if $\exists x \in \{0, 1\}^n \forall y \in \{0, 1\}^m x \cdot u + y \cdot w \neq S$.

Assume that both t and t' are in Ω . By induction on the definition of numerical occurrence indicators, it is easy to see that $(v, t, v, t') \in \llbracket \mathbf{N}[k, k] \rrbracket_C$ if and only if $t' = t + k$. Hence, by definition of r_{u_i} , we have that $(v, t, v, t') \in \llbracket r_{u_i} \rrbracket_C$ if and only if $t' = t$ (0 occurrences) or $t' = t + u_i$ (1 occurrence), or what is equivalent, if there exists $x_i \in \{0, 1\}$ such that $t' = t + x_i \cdot u_i$. In fact, it can be proved by induction that $(v, t, v, t') \in \llbracket r_u \rrbracket_C$ if and only if $\exists x \in \{0, 1\}^n$ such that $t' = t + x \cdot u$. We will demonstrate something stronger, which is that $(v, t, v, t') \in \llbracket r_{u_1} / \dots / r_{u_n} \rrbracket_C$ if and only if there exists $(x_1, \dots, x_i) \in \{0, 1\}^i$ such that $t' = t + \sum_{k=1}^i x_k \cdot u_k$.

Our base case will be checking the property for r_{u_1} , which, by the same exact reasoning as above, satisfies that $(v, t, v, t') \in \llbracket r_{u_1} \rrbracket_C$ if and only if there exists $x_1 \in \{0, 1\}$ such that $t' = t + x_1 \cdot u_1$. Suppose now that $(v, t, v, t') \in \llbracket r_{u_1} / \dots / r_{u_i} \rrbracket_C$ if and only if there exists $(x_1, \dots, x_i) \in \{0, 1\}^i$ such that $t' = t + \sum_{k=1}^i x_k \cdot u_k$. We also know by the previous reasoning that $(v, t', v, t'') \in \llbracket r_{u_{i+1}} \rrbracket_C$ if and only if there exists $x_{i+1} \in \{0, 1\}$ such that $t'' = t' + x_{i+1} \cdot u_{i+1}$. Hence, by definition of (r_1/r_2) , we get that $(v, t, v, t'') \in \llbracket r_{u_1} / \dots / r_{u_{i+1}} \rrbracket_C$ if and only if there exists (v, t') such that there exists $(x_1, \dots, x_i) \in \{0, 1\}^i$ such that $t' = t + \sum_{k=1}^i x_k \cdot u_k$ and there exists $x_{i+1} \in \{0, 1\}$ such that $t'' = t' + x_{i+1} \cdot u_{i+1}$, i.e., if and only if there exists $(x_1, \dots, x_{i+1}) \in \{0, 1\}^{i+1}$ such that $t'' = t + \sum_{k=1}^{i+1} x_k \cdot u_k$. This yields the result.

Moreover, by definition of $\llbracket (r_1/r_2) \rrbracket_C$, we get that $(v, M, v, 2 \cdot M) \in \llbracket r \rrbracket_C$ if and only if there exists $x \in \{0, 1\}^n$ such that:

$$(v, M + x \cdot u, v, 2 \cdot M) \in \llbracket r_v / \mathbf{N}[0, _] / (\neg \lt 2 \cdot M) \rrbracket_C$$

Notice also that the right part of this formula is built in the following way: $(\neg \lt 2 \cdot M)$ is a test that is only satisfied by $(v, 2 \cdot M, v, 2 \cdot M)$, whereas $\mathbf{N}[0, _]$ is satisfied by any tuple (v, t, v, t') such that $t \leq t'$. Hence, $(v, t, v, t') \in \llbracket \mathbf{N}[0, _] / (\neg \lt 2 \cdot M) \rrbracket_C$ if and only if t is any time point in Ω and $t' = 2 \cdot M$. Thus, all we need to prove now is that there exists some time point t such that $(v, M + x \cdot u, v, t) \in \llbracket r_v \rrbracket_C$ if and only if $\forall y \in \{0, 1\}^m x \cdot u + y \cdot w \neq S$.

First, we show by induction that if $(v, t, v, t') \in \llbracket r_j \rrbracket_C$, then $t = t'$. The base case is trivial, since r_0 is a test. For the inductive case, assume that if $(v, t, v, t') \in \llbracket r_j \rrbracket_C$, then $t' = t$. For conciseness, define $a := w_{j+1}$. In the case of $j+1$, we know that if $(v, t_1, v, t_2) \in \llbracket \mathbf{N}[a, a] / r_j / \mathbf{P}[2a, 2a] \rrbracket_C$, then there exist time points t_3 and t_4 such that $(v, t_1, v, t_3) \in \llbracket \mathbf{N}[a, a] \rrbracket_C$, $(v, t_3, v, t_4) \in \llbracket r_j \rrbracket_C$ and $(v, t_4, v, t_2) \in \llbracket \mathbf{P}[2a, 2a] \rrbracket_C$. Notice then that these conditions only hold respectively if $t_3 = t_1 + a$, by definition of operator \mathbf{N} and numerical occurrence indicators, $t_3 = t_4$ by induction hypothesis and $t_2 = t_4 - 2a$. Thus, $t_2 = t_4 - 2a = t_3 - 2a = t_1 - a$. It is clear that if $(v, t_1, v, t_5) \in \llbracket (\mathbf{N}[a, a] / r_k / \mathbf{P}[2a, 2a])[2, 2] \rrbracket_C$, then $t_5 = t_1 - 2a$. Finally, if $(v, t_5, v, t') \in \llbracket \mathbf{N}[2a, 2a] \rrbracket_C$, then $t' = t_5 + 2a$, so by definition of $\llbracket (r_1/r_2) \rrbracket_C$, we conclude that if $(v, t, v, t') \in \llbracket r_{j+1} \rrbracket_C$, then $t' = t$.

Given the conclusion in the previous paragraph, all we need to prove now is that $(v, M + x \cdot u, v, M + x \cdot u) \in \llbracket r_v \rrbracket_C$ if and only if $\forall y \in \{0, 1\}^m x \cdot u + y \cdot w \neq S$. In fact, we will prove by induction a stronger condition:

$$\text{for every } k \in \{0, \dots, m\}, \text{ it holds that } \forall y \in \{0, 1\}^k t + \sum_{j=1}^k y_j \cdot w_j \neq S \text{ if and only if } (v, M + t, v, M + t) \in \llbracket r_k \rrbracket_C$$

The case when $t = x \cdot u$ and $k = m$ yields Σ_2^P -hardness of $\text{Eval}(\text{ITPG}, \text{NavL}[\text{NOI}])$ as a result. In the base case $k = 0$, we need to prove that $t \neq S$ if and only if $(v, M+t, v, M+t) \in \llbracket r_0 \rrbracket_C$. Recall that $r_0 = r_{t \neq S+M}$. It can be easily checked that $(v, t) \models r_0$ if and only if $t \neq S+M$. Also, r_0 is a test, so $(v, t, v, t') \in \llbracket r_0 \rrbracket_C$ if and only if $t = t'$ and $t \neq S+M$. Hence, we have that $(v, M+t, v, M+t) \in \llbracket r_0 \rrbracket_C$ if and only if $M+t = M+t$ (which is trivially satisfied) and $M+t \neq M+S$, i.e., if and only if $t \neq S$. For the inductive case, assume that for $k \in \{0, \dots, m\}$, it holds that:

$$\forall y \in \{0, 1\}^k \ t + \sum_{j=1}^k y_j \cdot w_j \neq S \text{ if and only if } (v, M+t, v, M+t) \in \llbracket r_k \rrbracket_C.$$

Then we have to prove that:

$$\forall y \in \{0, 1\}^{k+1} \ t + \sum_{j=1}^{k+1} y_j \cdot w_j \neq S \text{ if and only if } (v, M+t, v, M+t) \in \llbracket r_{k+1} \rrbracket_C$$

To prove this, define again $a := w_{k+1}$ for conciseness. Recall that $r_{k+1} = ((\mathbf{N}[a, a] / r_k / \mathbf{P}[2a, 2a])[2, 2] / \mathbf{N}[2a, 2a])$. Notice then that $(v, M+t, v, t') \in \llbracket \mathbf{N}[a, a] / r_k / \mathbf{P}[2a, 2a] \rrbracket_C$ if and only if there exist time points t_1 and t_2 such that $(v, M+t, v, t_1) \in \llbracket \mathbf{N}[a, a] \rrbracket_C$, $(v, t_1, v, t_2) \in \llbracket r_k \rrbracket_C$ and $(v, t_2, v, t') \in \llbracket \mathbf{P}[2a, 2a] \rrbracket_C$. The first condition is equivalent to having that $t_1 = M+t+a$. The second condition implies that $t_1 = t_2$, given what we proved in the previous paragraphs. Hence, given that $(v, M+t+a, v, M+t+a) \in \llbracket r_k \rrbracket_C$, we conclude by induction hypothesis that $\forall y \in \{0, 1\}^k \ t+a + \sum_{j=1}^k y_j \cdot w_j \neq S$. The third condition is equivalent to having that $t' = t_2 - 2a$. Altogether, this means that $(v, M+t, v, t') \in \llbracket \mathbf{N}[a, a] / r_k / \mathbf{P}[2a, 2a] \rrbracket_C$ if and only if $t' = M+t-a$ and $\forall y \in \{0, 1\}^k$, it holds that $t+a + \sum_{j=1}^k y_j \cdot w_j \neq S$. Now, this means that $(v, M+t, v, t') \in \llbracket (\mathbf{N}[a, a] / r_k / \mathbf{P}[2a, 2a])[2, 2] \rrbracket_C$ if there exists a time point t'' such that $t'' = M+t-a$, $t' = M+(t-a)-a = M+t-2a$, $\forall y \in \{0, 1\}^k$, it holds that $t+a + \sum_{j=1}^k y_j \cdot w_j \neq S$, and $\forall y \in \{0, 1\}^k$, it holds that $(t-a) + a + \sum_{j=1}^k y_j \cdot w_j \neq S$. Therefore, $(v, M+t, v, t') \in \llbracket (\mathbf{N}[a, a] / r_k / \mathbf{P}[2a, 2a])[2, 2] \rrbracket_C$ if and only if $t' = M+t-2a$ and for every $y \in \{0, 1\}^{k+1}$, it holds that $t + \sum_{j=1}^{k+1} y_j \cdot w_j \neq S$. Given that for $t_1, t_2 \in \Omega$, it holds that $(v, t_1, v, t_2) \in \llbracket \mathbf{N}[2a, 2a] \rrbracket_C$ if and only if $t_2 = t_1 + 2a$, we conclude that $(v, M+t, v, t') \in \llbracket (\mathbf{N}[a, a] / r_k / \mathbf{P}[2a, 2a])[2, 2] / \mathbf{N}[2a, 2a] \rrbracket_C$ if and only if $t' = M+t$ and $\forall y \in \{0, 1\}^{k+1} \ t + \sum_{j=1}^{k+1} y_j \cdot w_j \neq S$. Finally, this gives us the result we are trying to prove, that is, $(v, M+t, v, M+t) \in \llbracket (\mathbf{N}[a, a] / r_k / \mathbf{P}[2a, 2a])[2, 2] / \mathbf{N}[2a, 2a] \rrbracket_C$ if and only if $\forall y \in \{0, 1\}^{k+1} \ t + \sum_{j=1}^{k+1} y_j \cdot w_j \neq S$.

To conclude the proof of the theorem, notice that r can be constructed in polynomial time with respect to the sizes of u , v and S , so the entire reduction can be computed in polynomial time.

D. $\text{Eval}(\text{ITPG}, \text{NavL}[\text{PC}, \text{NOI}])$ is PSPACE-complete

Consider the following well-known decision problem called True Quantified Boolean Formula (TQBF), which is well known to be PSPACE-complete [70]:

Problem:	TQBF
Input:	A quantified Boolean formula $\psi = Q_1 x_1 \dots Q_n x_n \varphi(x_1, \dots, x_n)$ in prenex normal form where $\varphi(x_1, \dots, x_n)$ is a Boolean formula on variables x_1, \dots, x_n , and Q_1, \dots, Q_n are quantifiers (\forall or \exists).
Output:	<i>true</i> if ψ is valid, and <i>false</i> otherwise.

Without loss of generality, φ can be assumed to be in conjunctive normal form. We will show that TQBF can be reduced to our problem $\text{Eval}(\text{ITPG}, \text{NavL}[\text{PC}, \text{NOI}])$ by proceeding in three steps. Let $\psi = Q_1 x_1 \dots Q_n x_n \varphi(x_1, \dots, x_n)$. First, we will show that a predicate $\text{bit}(i, t)$ (defined below) can be written in our language. Then, by using that predicate, we will show that φ can be encoded in our language. Finally, we will show that an expression representing the quantifiers of ψ can be added to the expression encoding φ , which yields the result.

We start with a QBF formula ψ as described above to build an input for the problem $\text{Eval}(\text{ITPG}, \text{NavL}[\text{PC}, \text{NOI}])$. More precisely, the input ITPG will be $C = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$ where $\Omega = [0, 2^n - 1]$, $N = \{v\}$, $E = \emptyset$, ρ is an empty function, $\lambda(v) = l$, $\xi(v) = \{[0, 2^n - 1]\}$ and σ is an empty function. In other words, C is an ITPG consisting of only one node existing from time 0 to time $2^n - 1$, with no edges or properties. Moreover, we will build an expression r such that $(v, 0, v, 0) \in \llbracket r \rrbracket_C$ if and only if ψ is valid, which concludes the reduction. The steps to construct r are shown next.

Step 1: Expressing the predicate bit with an expression in $\text{NavL}[\text{PC}, \text{NOI}]$. Consider predicate $\text{bit}(i, t)$ that tests whether the i -th bit of time t (from right to left when written in its binary representation) is 1. For instance, $\text{bit}(1, 0)$ is false, and $\text{bit}(5, 30)$ is true, since the first bit of 0 is 0, whereas 30 is 11110 in binary, and its fifth bit is 1. Now, consider the following expression:

$$r_i = ? (\mathbf{P} [2^i, 2^i] [0, _] / (< 2^i \wedge \neg < 2^{i-1}))$$

Notice that r_i is a test. Thus, for a pair of temporal objects (o_1, t_1, o_2, t_2) to satisfy r_i , (o_1, t_1) must be equal to (o_2, t_2) . The expression to satisfy is a path test, so there must be some temporal object (o_3, t_3) , such that $(o_1, t_1, o_3, t_3) \in \llbracket \mathbf{P} [2^i, 2^i] [0, _] / (< 2^i \wedge \neg < 2^{i-1}) \rrbracket_C$. Since the right part is a test as well, we can split the expression into two parts. Firstly, we must have that $(o_1, t_1, o_3, t_3) \in \llbracket \mathbf{P} [2^i, 2^i] [0, _] \rrbracket_C$, which implies that $t_3 = t_1 - k * 2^i$ for some integer $k \geq 0$. Secondly, we must have that $(o_3, t_3) \models (< 2^i \wedge \neg < 2^{i-1})$, which implies that $2^{i-1} \leq t_3 < 2^i$. This means that by writing t_3 in its binary form, we get 1 as its i -th bit. Together, these two conditions imply that t_1 also has 1 as its i -th bit, when written in its binary form. In consequence, we get that

$$(o, t, o, t) \in \llbracket r_i \rrbracket_C \text{ if and only if } \text{bit}(i, t) \text{ is } \textit{true}$$

Besides, we trivially get that:

$$(o, t, o, t) \in \llbracket \neg r_i \rrbracket_C \text{ if and only if } \text{bit}(i, t) \text{ is } \textit{false}$$

Finally, notice that both r_i and $\neg r_i$ have linear length with respect to i , which will be important later.

Step 2: Expressing any CNF formula in NavL[PC, NOI]. Assume that

$$\varphi(x_1, \dots, x_n) = \bigwedge_{j=1}^m \bigvee_{k=1}^{m_j} l_{j,k}$$

where for every j and k , $l_{j,k}$ is a literal, *i.e.*, either a variable in $\{x_1, \dots, x_n\}$ or its negation. Then, for every $j \in \{1, \dots, m\}$ and $k \in \{1, \dots, m_j\}$, we define:

$$L_{j,k} = \begin{cases} r_i & \text{if } l_{j,k} = x_i \\ \neg r_i & \text{if } l_{j,k} = \neg x_i \end{cases}$$

We can use these expressions to build a regular expression that tests the satisfiability of our formula $\varphi(x_1, \dots, x_n)$ by any valuation $\sigma : \{x_1, \dots, x_n\} \rightarrow \{\textit{false}, \textit{true}\}$. To do this, we will use a time value $t \in [0, 2^n - 1]$ to represent a valuation σ_t , where $\sigma_t(x_i) = \textit{true}$ if and only if the i -th bit of t is 1. We can do so by employing the expressions $L_{i,k}$ on tests along with conjunctions (\wedge) and disjunctions (\vee), which are also present in NavL[PC, NOI]:

$$r_{\varphi(x_1, \dots, x_n)} = \bigwedge_{j=1}^m \bigvee_{k=1}^{m_j} L_{j,k}$$

Here again, $r_{\varphi(x_1, \dots, x_n)}$ is a test, so if it is satisfied by (o_1, t_1, o_2, t_2) , then $(o_1, t_1) = (o_2, t_2)$. Furthermore, we show that, because of how the expressions $L_{j,k}$ are defined, we have:

$$(o, t, o, t) \in \llbracket r_{\varphi(x_1, \dots, x_n)} \rrbracket_C \text{ if and only if } \sigma_t(\varphi(x_1, \dots, x_n)) = \textit{true}$$

To show the previous assertion, first assume that $(o, t, o, t) \in \llbracket r_{\varphi(x_1, \dots, x_n)} \rrbracket_C$. Because of how the expression is defined, for each $j \in \{1, \dots, m\}$ we must have that $(o, t, o, t) \in \llbracket \bigvee_{k=1}^{m_j} L_{j,k} \rrbracket_C$. Hence, for any arbitrary $j \in \{1, \dots, m\}$, we immediately get that there must exist $k \in \{1, \dots, m_j\}$ such that $(o, t, o, t) \in \llbracket L_{j,k} \rrbracket_C$. If $l_{j,k} = x_i$, then we must also have that $(o, t, o, t) \in \llbracket r_i \rrbracket_C$, which, as we already showed, is equivalent to having that $\text{bit}(i, t)$ is *true*, which in turn is equivalent to $\sigma_t(x_i) = \textit{true}$. Otherwise, if $l_{j,k} = \neg x_i$, then we must have that $(o, t, o, t) \in \llbracket \neg r_i \rrbracket_C$, which, as we also showed, is equivalent to having that $\text{bit}(i, t)$ is *false*, which in turn is equivalent to $\sigma_t(x_i) = \textit{false}$. In both cases, we get that $\sigma_t(l_{j,k}) = \textit{true}$, which means that $\sigma_t(\bigvee_{k=1}^{m_j} l_{j,k}) = \textit{true}$. Since this holds for an arbitrary value j , it holds for the entire conjunction. Hence, $\sigma_t(\varphi(x_1, \dots, x_n)) = \textit{true}$.

Now, to show that the inverse is also true, assume that there is some $t \in \Omega$ such that $\sigma_t(\varphi(x_1, \dots, x_n)) = \textit{true}$. By definition, this means that for every $j \in \{1, \dots, m\}$, $\sigma_t(\bigvee_{k=1}^{m_j} l_{j,k}) = \textit{true}$. In turn, this means that for every j there is some $k \in \{1, \dots, m_j\}$ such that $\sigma_t(l_{j,k}) = \textit{true}$. As we saw earlier, this condition is equivalent to having that $(o, t, o, t) \in \llbracket L_{j,k} \rrbracket_C$, hence, for every $j \in \{1, \dots, m\}$, we also have that $(o, t, o, t) \in \llbracket \bigvee_{k=1}^{m_j} L_{j,k} \rrbracket_C$. Given that this is true for every j , by definition, it is also true for the conjunction of them. Hence, we get that $(o, t, o, t) \in \llbracket r_{\varphi(x_1, \dots, x_n)} \rrbracket_C$. This concludes the proof in Step 2.

Observation: Notice that the previous results already implies NP-hardness and CONP-hardness for the problem. Since every valuation $\sigma : \{x_1, \dots, x_n\} \rightarrow \{\textit{false}, \textit{true}\}$ has a corresponding time point t such that $\sigma = \sigma_t$, and a Boolean CNF formula $\varphi(x_1, \dots, x_n)$ on n variables x_1, \dots, x_n is satisfiable if and only if there exists a valuation σ such that $\sigma(\varphi(x_1, \dots, x_n)) = \textit{true}$, it is also true that $\varphi(x_1, \dots, x_n)$ is satisfiable if and only if $(v, 0, v, 0) \in \llbracket ?(\mathbf{N}[0, _] / r_{\varphi(x_1, \dots, x_n)}) \rrbracket_C$ (that is, advance in time to an arbitrary time point t and check the condition that implies that $\sigma_t(\varphi(x_1, \dots, x_n)) = \textit{true}$ for the temporal object (v, t)). Similarly, $\varphi(x_1, \dots, x_n)$ is a tautology if and only if $(v, 0, v, 0) \in \llbracket \neg ?(\mathbf{N}[0, _] / \neg r_{\varphi(x_1, \dots, x_n)}) \rrbracket_C$ (there is no path to a time

point t such that $\sigma_t(\varphi(x_1, \dots, x_n)) = \text{false}$, hence there is no valuation that makes φ to be *false*). In what follows, we show PSPACE-hardness of the problem.

Step 3: Expressing satisfiability of quantified Boolean formulae with an expression in NavL[PC,NOI]. Consider a quantified Boolean formula in prenex normal form

$$Q_1 x_1 \dots Q_n x_n \varphi(x_1, \dots, x_n)$$

Since we already have a way to express $\varphi(x_1, \dots, x_n)$, we only need to express the possible valuations (*i.e.*, time points) generated by the sequence of quantifiers Q_1, \dots, Q_n .

First, assume Q_i is the existential quantifier (\exists). This means that we can either make x_i take valuation *true* or *false* to satisfy our formula. Considering time points, this is equivalent to have either 1 or 0 at the i -th bit of the time t at which we are standing. The intuition is that this can easily be expressed by starting at time 0, and then deciding whether to move into a future time point with the expression $(\mathbf{N}[2^{i-1}, 2^{i-1}] + \mathbf{N}[0, 0])$ to set the i -th bit of the time point. Notice that if there are only expressions of the form $\mathbf{N}[2^{i-1}, 2^{i-1}]$, and they are only mentioned once (at least, as prefixes of our test expressions) for each i , they will not affect other bits of t . Hence, if s_{i+1} represents the part of the subformula $Q_{i+1} x_{i+1} \dots Q_n x_n \varphi(x_1, \dots, x_n)$, then the subformula $\exists x_i Q_{i+1} x_{i+1} \dots Q_n x_n \varphi(x_1, \dots, x_n)$ can be represented by first navigating through time, only affecting the first $i - 1$ bits, and then testing that the reached temporal object satisfies the following test:

$$s_i = ? ((\mathbf{N}[2^{i-1}, 2^{i-1}] + \mathbf{N}[0, 0]) / s_{i+1}).$$

In turn, if Q_i is the universal quantifier (\forall), we will employ the fact that $\forall x \psi(x)$ is equivalent to $\neg \exists x \neg \psi(x)$ for every formula $\psi(x)$ with free variable x . Hence, if s_{i+1} represents the part of the subformula $Q_{i+1} x_{i+1} \dots Q_n x_n \varphi(x_1, \dots, x_n)$, then the subformula $\forall x_i Q_{i+1} x_{i+1} \dots Q_n x_n \varphi(x_1, \dots, x_n)$ can be represented by first only navigating through time, only affecting the first $i - 1$ bits, and then testing whether the reached temporal object satisfies the following test:

$$s_i = \neg (? ((\mathbf{N}[2^{i-1}, 2^{i-1}] + \mathbf{N}[0, 0]) / (\neg s_{i+1}))).$$

Finally, define $s_{n+1} = r_{\varphi(x_1, \dots, x_n)}$. We claim that, for $i \in \{n+1, n, \dots, 1\}$ (*i.e.*, 0 to n quantifiers), if $t < 2^{i-1}$, then:

$$(v, t, v, t) \in \llbracket s_i \rrbracket_C \text{ if and only if } Q_i x_i \dots Q_n x_n \varphi(\sigma_t(x_1), \dots, \sigma_t(x_{i-1}), x_i, \dots, x_n) \text{ is valid.}$$

In particular, for n quantifiers, *i.e.*, when $i = 1$, this result gives us PSPACE-hardness for Eval(ITPG, NavL[PC,NOI]), since we will have that ψ is *true* if and only if $(v, 0, v, 0) \in \llbracket s_1 \rrbracket_C$, where C and s_1 can be constructed in polynomial time in the size of ψ . We will prove this claim by induction over the number of quantifiers preceding $\varphi(x_1, \dots, x_n)$.

The base case consists of the formula with no quantified variables, *i.e.*, when $i = n + 1$. We must show that if $t < 2^n$, then $(v, t, v, t) \in \llbracket r_{\varphi(x_1, \dots, x_n)} \rrbracket_C$ if and only if $\varphi(\sigma(x_1), \dots, \sigma(x_n))$ is *true*. Notice that $\varphi(\sigma(x_1), \dots, \sigma(x_n))$ is *true* is equivalent to having that $\sigma_t(\varphi(x_1, \dots, x_n))$ is *true*. Hence, by step 2, this is equivalent to having that $(v, t, v, t) \in \llbracket r_{\varphi(x_1, \dots, x_n)} \rrbracket_C$.

For the inductive case, assume that the claim holds for k quantifiers, for some k such that $0 \leq k \leq n$. This means that for $i = n - k + 1$, if $t < 2^{i-1}$, then the following condition holds:

$$(v, t, v, t) \in \llbracket s_i \rrbracket_C \text{ if and only if } Q_i x_i \dots Q_n x_n \varphi(\sigma_t(x_1), \dots, \sigma_t(x_{i-1}), x_i, \dots, x_n) \text{ is valid}$$

We then have to prove that the condition holds for $k + 1$ quantifiers, *i.e.*, for $i - 1 = n - k$. That is, if $t' < 2^{i-2}$, then we have to show that:

$$(v, t', v, t') \in \llbracket s_{i-1} \rrbracket_C \text{ if and only if } Q_{i-1} x_{i-1} \dots Q_n x_n \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-2}), x_{i-1}, \dots, x_n) \text{ is valid} \quad (7)$$

Let $t' < 2^{i-2}$, and consider the following cases.

- If $Q_{i-1} = \exists$, recall that

$$s_{i-1} = ? ((\mathbf{N}[2^{i-2}, 2^{i-2}] + \mathbf{N}[0, 0]) / s_i).$$

Notice then that by definition of s_{i-1} , we have that $(v, t', v, t') \in \llbracket s_{i-1} \rrbracket_C$ if and only if there exists $t_1 \in \Omega$ such that $(v, t', v, t_1) \in \llbracket (\mathbf{N}[2^{i-2}, 2^{i-2}] + \mathbf{N}[0, 0]) / s_i \rrbracket_C$. By definition of s_i , the previous condition holds if and only if $(v, t', v, t_1) \in \llbracket (\mathbf{N}[2^{i-2}, 2^{i-2}] + \mathbf{N}[0, 0]) \rrbracket_C$ and $(v, t_1, v, t_1) \in \llbracket s_i \rrbracket_C$. Now, this means that $(v, t', v, t') \in \llbracket s_{i-1} \rrbracket_C$ if and only if there exists $t_1 \in \{t', t' + 2^{i-2}\}$ satisfying that $(v, t_1, v, t_1) \in \llbracket s_i \rrbracket_C$.

To prove the direction (\Rightarrow) of (7) assume that $(v, t', v, t') \in \llbracket s_{i-1} \rrbracket_C$, which implies that there exists $t_1 \in \{t', t' + 2^{i-2}\}$ satisfying that $(v, t_1, v, t_1) \in \llbracket s_i \rrbracket_C$. Given that t' is an integer with $i - 2$ bits, t_1 comes from either putting 1 or 0 as the $(i - 1)$ -th bit of t' , which means that $t_1 < 2^{i-1}$ must hold. By induction hypothesis, this implies that $Q_i x_i \dots Q_n x_n \varphi(\sigma_{t_1}(x_1), \dots, \sigma_{t_1}(x_{i-1}), x_i, \dots, x_n)$ is valid. Since t_1 and t' share the

same first $i - 2$ bits, we get that $\sigma_{t'}(x_j) = \sigma_{t_1}(x_j)$ for $j \in \{1, \dots, i - 2\}$. Therefore, we conclude that $\exists x_{i-1} Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-2}), x_{i-1}, \dots, x_n)$ is valid.

To prove the direction (\Leftarrow) of (7) suppose that $\exists x_{i-1} Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-2}), x_{i-1}, \dots, x_n)$ is valid. Then we know that $Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-2}), b, x_i, \dots, x_n)$ is valid for some value $b \in \{true, false\}$. Define $\mathbb{1}_b$ as 1 if $b = true$, and as 0 otherwise. Notice then that by taking $t_1 = t' + 2^{i-2} \cdot \mathbb{1}_b$, we get that $\sigma_{t_1}(x_{i-1}) = b$. Moreover, $\sigma_{t_1}(x_j) = \sigma_{t'}(x_j)$ for every $j \in \{1, \dots, i - 2\}$ since $t' < 2^{i-2}$ and t_1 shares all its first $i - 2$ bits with t' . In consequence, this gives us that $Q_i x_i \dots Q_n x_n \varphi(\sigma_{t_1}(x_1), \dots, \sigma_{t_1}(x_{i-1}), x_i, \dots, x_n)$ is valid. By induction, this means that $(v, t_1, v, t_1) \in \llbracket s_i \rrbracket_C$. Since $t_1 = t' + 2^{i-2} \cdot \mathbb{1}_b$, we have that either $t_1 = t'$ or $t_1 = t' + 2^{i-2}$. In any case, we get that $(v, t', v, t_1) \in \llbracket \mathbf{N}[2^{i-2}, 2^{i-2}] + \mathbf{N}[0, 0] \rrbracket_C$, so we have that $(v, t, v, t_1) \in \llbracket (\mathbf{N}[2^{i-2}, 2^{i-2}] + \mathbf{N}[0, 0]) / s_i \rrbracket_C$. By definition of s_{i-1} , we conclude that $(v, t', v, t') \in \llbracket s_{i-1} \rrbracket_C$, which was to be shown.

- If $Q_{i-1} = \forall$, recall that

$$s_{i-1} = \neg? ((\mathbf{N}[2^{i-2}, 2^{i-2}] + \mathbf{N}[0, 0]) / (\neg s_i)).$$

Now, by definition of s_{i-1} , we have that $(v, t', v, t') \in \llbracket s_{i-1} \rrbracket_C$ if and only if

$$(v, t') \not\models ? ((\mathbf{N}[2^{i-2}, 2^{i-2}] + \mathbf{N}[0, 0]) / (\neg s_i)).$$

This, in turn, is equivalent to the fact that there is no time point $t_1 \in \Omega$ such that $(v, t', v, t_1) \in \llbracket (\mathbf{N}[2^{i-2}, 2^{i-2}] + \mathbf{N}[0, 0]) / (\neg s_i) \rrbracket_C$. Hence, we know that $(v, t', v, t') \in \llbracket s_{i-1} \rrbracket_C$ if and only if, for every time point $t_1 \in \Omega$:

$$(v, t', v, t_1) \notin \llbracket (\mathbf{N}[2^{i-2}, 2^{i-2}] + \mathbf{N}[0, 0]) / (\neg s_i) \rrbracket_C.$$

This condition means that each t_1 satisfies $(v, t', v, t_1) \notin \llbracket (\mathbf{N}[2^{i-2}, 2^{i-2}] + \mathbf{N}[0, 0]) \rrbracket_C$ or $(v, t_1) \not\models (\neg s_i)$. This, in turn, is equivalent to saying that if $t_1 \in \{t', t' + 2^{i-2}\}$ then $(v, t_1) \not\models (\neg s_i)$, i.e., $(v, t_1) \models s_i$. As a consequence, $(v, t', v, t') \in \llbracket s_{i-1} \rrbracket_C$ if and only if $(v, t') \models s_i$ and $(v, t' + 2^{i-2}) \models s_i$, which is equivalent to having that $(v, t', v, t') \in \llbracket s_i \rrbracket_C$ and $(v, t' + 2^{i-2}, v, t' + 2^{i-2}) \in \llbracket s_i \rrbracket_C$.

To prove the direction (\Rightarrow) of (7) suppose that $(v, t', v, t') \in \llbracket s_{i-1} \rrbracket_C$, so we also have that $(v, t', v, t') \in \llbracket s_i \rrbracket_C$ and $(v, t' + 2^{i-2}, v, t' + 2^{i-2}) \in \llbracket s_i \rrbracket_C$. Notice then that $t' < 2^{i-2}$, so t' and $t' + 2^{i-2}$ are both smaller than 2^{i-1} . By induction hypothesis, we get then that both quantified Boolean formulae $Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-1}), x_i, \dots, x_n)$ and $Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'+2^{i-2}}(x_1), \dots, \sigma_{t'+2^{i-2}}(x_{i-1}), x_i, \dots, x_n)$ are valid. Furthermore, since t' is smaller than 2^{i-2} , $\sigma_{t'}(x_{i-1}) = false$, and since $t' + 2^{i-2}$ only differs from t' in its $(i - 1)$ -th bit, which is 1, we get that $\sigma_{t'+2^{i-2}}(x_{i-1}) = true$ and, for every $j \in \{1, \dots, i - 2\}$, it holds that $\sigma_{t'}(x_j) = \sigma_{t'+2^{i-2}}(x_j)$. Hence, both quantified Boolean formulae $Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-2}), false, x_i, \dots, x_n)$ and $Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-2}), true, x_i, \dots, x_n)$ are valid. Therefore, we conclude that the quantified Boolean formula $\forall x_{i-1} Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-2}), x_{i-1}, x_i, \dots, x_n)$ is valid.

To show the direction (\Leftarrow) of (7) suppose that $\forall x_{i-1} Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-2}), x_{i-1}, x_i, \dots, x_n)$ is valid. Then we get that both quantified Boolean formulae $Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-2}), false, x_i, \dots, x_n)$ and $Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-2}), true, x_i, \dots, x_n)$ are valid. As before, since t' is smaller than 2^{i-2} , $\sigma_{t'}(x_{i-1}) = false$, and since $t' + 2^{i-2}$ only differs from t' in its $(i - 1)$ -th bit, which is 1, we get that $\sigma_{t'+2^{i-2}}(x_{i-1}) = true$ and for every $j \in \{1, \dots, i - 2\}$, it holds that $\sigma_{t'}(x_j) = \sigma_{t'+2^{i-2}}(x_j)$. This allows us to conclude that both $Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-1}), x_i, \dots, x_n)$ and $Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'+2^{i-2}}(x_1), \dots, \sigma_{t'+2^{i-2}}(x_{i-1}), x_i, \dots, x_n)$ are valid. Finally, by induction hypothesis, this implies that $(v, t', v, t') \in \llbracket s_i \rrbracket_C$ and $(v, t' + 2^{i-2}, v, t' + 2^{i-2}) \in \llbracket s_i \rrbracket_C$, which, as shown before, holds if and only if $(v, t', v, t') \in \llbracket s_{i-1} \rrbracket_C$, which concludes the proof for this case.

As we mentioned, all this together implies that $\text{Eval}(\text{ITPG}, \text{NavL}[\text{PC}, \text{NOI}])$ is PSPACE-hard, since $\text{ITPG } C$, expression s_1 and tuple $(v, 0, v, 0)$ can be constructed in polynomial time in the size of ψ , and the problem of determining whether ψ is valid can be reduced to the problem of verifying whether $(v, 0, v, 0) \in \llbracket s_1 \rrbracket_C$.

Thus, it only remains to show that $\text{Eval}(\text{ITPG}, \text{NavL}[\text{PC}, \text{NOI}])$ is in PSPACE. To do this, we will provide an algorithm in PSPACE that, given an $\text{ITPG } C$, an expression r in $\text{NavL}[\text{PC}, \text{NOI}]$, and a tuple $(o, t, o', t') \in \text{PTO}(C)$, computes whether $(o, t, o', t') \in \llbracket r \rrbracket_C$. More precisely, Algorithm $\text{TUPLEEVALSOLVE}(C, r, (o_1, t_1, o_2, t_2))$ is defined as follows.

Next we show that for every $\text{ITPG } C = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$, expression r in $\text{NavL}[\text{PC}, \text{NOI}]$ and tuple $(o_1, t_1, o_2, t_2) \in \text{PTO}(C)$, it holds that $(o_1, t_1, o_2, t_2) \in \llbracket r \rrbracket_C$ if and only if $\text{TUPLEEVALSOLVE}(C, r, (o_1, t_1, o_2, t_2))$ returns *true*. Besides, we will prove that TUPLEEVALSOLVE works in polynomial space in the size of the input.

Notice firstly that the algorithm is recursive, and that the depth of the recursion is polynomial, since at every step on which the algorithm is called, the size of the path expression strictly decreases. There is one exception, that happens when r is of the form $\text{path}[n, _]$. Notice that here this expression is treated as if it was $\text{path}[n, m]$, where $m = n + |\Omega| \cdot |N \cup E|$ (a term with polynomial size with respect to the input). Thus, the whole expression r can be thought as an equivalent expression r' where all terms of the form $\text{path}[n, _]$ are replaced with similar ones, in a manner that makes the whole input remain polynomial

Algorithm 4: TUPLEEVALSOLVE($C, r, (o_1, t_1, o_2, t_2)$) (part I)

Input : An ITPG $C = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$, an expression r in NavL[PC, NOI] and a pair of temporal objects $(o_1, t_1, o_2, t_2) \in \text{PTO}(C)$

Output: *true* if and only if $(o_1, t_1, o_2, t_2) \in \llbracket r \rrbracket_C$

```
1 if  $r$  is a test then
2   if  $(o_1, t_1) \neq (o_2, t_2)$  then
3     return false
4   if  $r = \text{Node}$  then
5     return  $(o_1 \in N)$ 
6   else if  $r = \text{Edge}$  then
7     return  $(o_1 \in E)$ 
8   else if  $r = \ell$  for some  $\ell \in \text{Lab}$  then
9     return  $\lambda(o_1) = \ell$ 
10  else if  $r = p \mapsto v$  for some  $p \in \text{Prop}$  and  $v \in \text{Val}$  then
11    foreach valued interval  $(v', I) \in \sigma(o_1, p)$  do
12      if  $t_1 \in I$  then
13        return  $(v' = v)$ 
14  else if  $r = < k$  with  $k \in \Omega$  then
15    return  $(t_1 < k)$ 
16  else if  $r = \exists$  then
17    foreach interval  $I \in \xi(o_1)$  do
18      if  $t_1 \in I$  then
19        return true
20  else if  $r = (?r')$  then
21    foreach  $(o', t') \in (N \cup E) \times \Omega$  do
22      if TUPLEEVALSOLVE( $C, r', (o_1, t_1, o', t')$ ) then
23        return true
24  else if  $r = (\text{test}_1 \vee \text{test}_2)$  then
25    return TUPLEEVALSOLVE( $C, \text{test}_1, (o_1, t_1, o_1, t_1)$ ) or TUPLEEVALSOLVE( $C, \text{test}_2, (o_1, t_1, o_1, t_1)$ )
26  else if  $r = (\text{test}_1 \wedge \text{test}_2)$  then
27    return TUPLEEVALSOLVE( $C, \text{test}_1, (o_1, t_1, o_1, t_1)$ ) and TUPLEEVALSOLVE( $C, \text{test}_2, (o_1, t_1, o_1, t_1)$ )
28  else if  $r = (\neg r')$  then
29    return not TUPLEEVALSOLVE( $C, r', (o_1, t_1, o_1, t_1)$ )
30 else if  $r = \mathbf{N}$  then
31   return  $o_1 = o_2$  and  $t_2 = t_1 + 1$ 
32 else if  $r = \mathbf{P}$  then
33   return  $o_1 = o_2$  and  $t_2 = t_1 - 1$ 
34 else if  $r = \mathbf{F}$  then
35   return  $t_1 = t_2$  and  $((o_1 \in E$  and  $o_2 = \text{tgt}(o_1))$  or  $(o_2 \in E$  and  $o_1 = \text{src}(o_2)))$ 
36 else if  $r = \mathbf{B}$  then
37   return  $t_1 = t_2$  and  $((o_1 \in E$  and  $o_2 = \text{src}(o_1))$  or  $(o_2 \in E$  and  $o_1 = \text{tgt}(o_2)))$ 
38 else if  $r = (r_1 + r_2)$  then
39   return TUPLEEVALSOLVE( $C, r_1, (o_1, t_1, o_2, t_2)$ ) or TUPLEEVALSOLVE( $C, r_2, (o_1, t_1, o_2, t_2)$ )
40 else if  $r = (r_1 / r_2)$  then
41   foreach  $(o', t') \in (N \cup E) \times \Omega$  do
42     if TUPLEEVALSOLVE( $C, r_1, (o_1, t_1, o', t')$ ) and TUPLEEVALSOLVE( $C, r_2, (o', t', o_2, t_2)$ ) then
43     return true
```

Algorithm 5: TUPLEEVALSOLVE $(C, r, (o_1, t_1, o_2, t_2))$ (part II)

```
44 else if  $r = r'[n, m]$  then
45   if  $m = n$  then
46     if  $n = 0$  then
47       return  $(o_1, t_1) = (o_2, t_2)$ 
48     else if  $n = 1$  then
49       return TUPLEEVALSOLVE  $(C, r', (o_1, t_1, o_2, t_2))$ 
50      $l \leftarrow \lfloor n/2 \rfloor$ 
51     if  $m$  is even then
52       foreach  $(o', t') \in (N \cup E) \times \Omega$  do
53         if TUPLEEVALSOLVE  $(C, r'[l, l], (o_1, t_1, o', t'))$  and TUPLEEVALSOLVE  $(C, r'[l, l], (o', t', o_2, t_2))$  then
54           return true
55     else if  $m$  is odd then
56       foreach  $(o', t') \in (N \cup E) \times \Omega$  do
57         foreach  $(o'', t'') \in (N \cup E) \times \Omega$  do
58           if (
59             TUPLEEVALSOLVE  $(C, r'[l, l], (o_1, t_1, o', t'))$  and
60             TUPLEEVALSOLVE  $(C, r', (o', t', o'', t''))$  and
61             TUPLEEVALSOLVE  $(C, r'[l, l], (o', t', o_2, t_2))$ 
62           ) then
63             return true
64   else if  $n = 0$  then
65     if  $m = 1$  then
66       return  $(o_1, t_1) = (o_2, t_2)$  or TUPLEEVALSOLVE  $(C, r', (o_1, t_1, o_2, t_2))$ 
67      $l \leftarrow \lfloor m/2 \rfloor$ 
68     if  $m$  is even then
69       foreach  $(o', t') \in (N \cup E) \times \Omega$  do
70         if TUPLEEVALSOLVE  $(C, r'[0, l], (o_1, t_1, o', t'))$  and TUPLEEVALSOLVE  $(C, r'[0, l], (o', t', o_2, t_2))$  then
71           return true
72     else if  $m$  is odd then
73       foreach  $(o', t') \in (N \cup E) \times \Omega$  do
74         foreach  $(o'', t'') \in (N \cup E) \times \Omega$  do
75           if (
76             TUPLEEVALSOLVE  $(C, r'[0, l], (o_1, t_1, o', t'))$  and
77             TUPLEEVALSOLVE  $(C, r'[0, 1], (o', t', o'', t''))$  and
78             TUPLEEVALSOLVE  $(C, r'[0, l], (o', t', o_2, t_2))$ 
79           ) then
80             return true
81   else
82     foreach  $(o', t') \in (N \cup E) \times \Omega$  do
83       if TUPLEEVALSOLVE  $(C, r'[n, n], (o_1, t_1, o', t'))$  and TUPLEEVALSOLVE  $(C, r'[0, m - n], (o', t', o_2, t_2))$ 
84         then
85           return true
85   else if  $r = r'[n, \_]$  then
86      $m \leftarrow n + (|\Omega| \cdot |N \cup E|)^2$ 
87     return TUPLEEVALSOLVE  $(C, r'[n, m], (o_1, t_1, o_2, t_2))$ 
88   return false
```

to the original. Although it might seem that $\text{path}[n, m]$ could reach an exponential number of recursive calls, notice that this expression is always parsed as two expressions, $\text{path}[n, n]$ and $\text{path}[0, m - n]$, and then each of those is solved in a way similar to that of exponentiation by squaring, which allows to always get rid of the numerical occurrence indicator $[n, m]$ after at most $O(\log m)$ recursive calls, a number that is polynomial in the size of the input. Hence the recursion tree has polynomial height.

Secondly, assume that conjunctions (\wedge) and disjunctions (\vee) are computed from left to right, *i.e.*, $A(x) \wedge A(y)$ first computes $A(x)$ and then computes $A(y)$. Hence, at the most, we will need to have in memory as many calls to the algorithm as the recursion tree height. This number is polynomial, and since every non-recursive step is either a non-deterministic guess or clearly in polynomial time in the size of the input, we get that the whole algorithm gives an answer in PSPACE.

Now, let $C = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$ be an ITPG, let r be an expression in $\text{NavL}[\text{PC}, \text{NOI}]$ and let $(o_1, t_1, o_2, t_2) \in \text{PTO}(C)$ be a tuple concatenating two temporal objects. First suppose that $(o_1, t_1, o_2, t_2) \in \llbracket r \rrbracket_C$. We will show, by induction on the recursion level of the recursion tree, that there exists an execution of algorithm $\text{TUPLEEVALSOLVE}(C, r, (o_1, t_1, o_2, t_2))$ that returns *true*. Notice that the base case is given for those cases where there is no recursion. The base test cases, *i.e.*, when r is equal to either **Node**, **Edge**, ℓ , $p \mapsto v$, $< k$, or \exists , are easily checked, since all the algorithm does is checking their definitions over the temporal object (o_1, t_1) after checking that it is equal to the temporal object (o_2, t_2) . Notice that for property-value checking and existence checking, the default value is *false*, returned at the end of the algorithm. The base navigation operators **N**, **P**, **F** and **B** are also easily checked by their definitions. For time navigation, we check that the objects are the same and that their associated times are consecutive, whereas for spatial navigation, we check that the times are equal, and that the respective objects are consecutive, by looking at the functions tgt and src , as defined by the operators.

As for the recursive cases, assume that the property holds up to recursion level n and we want to prove that it holds at recursion level $n-1$ (one level higher in the recursion tree). Firstly, a path expression matching any of the regular expressions ($\text{test} \wedge \text{test}$), ($\text{test} \vee \text{test}$) or $(\neg \text{test})$ can also be checked quite straightforwardly by definition, and since the flow is deterministic, we will omit further formal proofs. Secondly, if the path expression r is of the form $(?r')$, then we know that $(o_1, t_1, o_2, t_2) \in \llbracket r \rrbracket_C$ if and only if $(o_1, t_1) = (o_2, t_2)$ and there exists a temporal object $(o', t') \in \text{PTO}(C)$ such that $(o_1, t_1, o', t') \in \llbracket r' \rrbracket_C$. In such case, the algorithm iterates one by one over the possible temporal objects to find one that satisfies the condition. If such temporal object exists, the call to TUPLEEVALSOLVE returns *true*, since we then know that the tuple (o_1, t_1, o', t') satisfies r' if and only if there exist an execution of the call that returns *true*. Conversely, if no such temporal object exists, all recursive calls to TUPLEEVALSOLVE will return *false* by induction hypothesis. In this case, the algorithm will finish the loop without returning and it will then reach the last line (in part II), returning *false*.

As for regular path expressions r of the form $(r_1 + r_2)$ where r_1 and r_2 are also regular path expressions, we know that by definition $\llbracket (r_1 + r_2) \rrbracket_C = \llbracket r_1 \rrbracket_C \cup \llbracket r_2 \rrbracket_C$. Hence, $(o_1, t_1, o_2, t_2) \in \llbracket r \rrbracket_C$ if and only if $(o_1, t_1, o_2, t_2) \in \llbracket r_1 \rrbracket_C$ or $(o_1, t_1, o_2, t_2) \in \llbracket r_2 \rrbracket_C$. By induction hypothesis, this means that $(o_1, t_1, o_2, t_2) \in \llbracket r \rrbracket_C$ if and only if either $\text{TUPLEEVALSOLVE}(C, r_1, (o_1, t_1, o_2, t_2))$ returns *true* or $\text{TUPLEEVALSOLVE}(C, r_2, (o_1, t_1, o_2, t_2))$ returns *true*. As the algorithm returns the disjunction of this two results, we get that $(o_1, t_1, o_2, t_2) \in \llbracket r \rrbracket_C$ if and only if $\text{TUPLEEVALSOLVE}(C, r, (o_1, t_1, o_2, t_2))$ returns *true*.

For regular path expressions r of the form (r_1 / r_2) where r_1 and r_2 are also TRPQs, we know that if $(o_1, t_1, o_2, t_2) \in \llbracket r \rrbracket_C$, then there must exist a temporal object $(o', t') \in \text{PTO}(C)$ such that $(o_1, t_1, o', t') \in \llbracket r_1 \rrbracket_C$ and $(o', t', o_2, t_2) \in \llbracket r_2 \rrbracket_C$. Thus, by iterating over all temporal object (o', t') , when we reach that exact temporal object, both $\text{TUPLEEVALSOLVE}(C, r_1, (o_1, t_1, o', t'))$ and $\text{TUPLEEVALSOLVE}(C, r_2, (o', t', o_2, t_2))$ will be *true*, so the algorithm will return *true* and $\text{true} = \text{true}$. On the other hand, if $(o_1, t_1, o_2, t_2) \notin \llbracket r \rrbracket_C$, then no matter what temporal object (o', t') is being considered, we will either have that $(o_1, t_1, o', t') \notin \llbracket r_1 \rrbracket_C$ or $(o', t', o_2, t_2) \notin \llbracket r_2 \rrbracket_C$. By induction hypothesis, this means that, for every execution, either the first call will be *false* or the second will, so the condition that makes the algorithm return *true* will not be met. Hence, the last line is reached and the algorithm returns *false*.

For expressions r matching regular expressions with numerical occurrence indicators of the form $r'[n, m]$, recall that, by definition, $\llbracket r'[n, m] \rrbracket_C = \bigcup_{k=n}^m \llbracket r'^k \rrbracket_C$. This implies that $(o_1, t_1, o_2, t_2) \in \llbracket r'[n, m] \rrbracket_C$ if and only if there exists an integer $k \in [n, m]$ such that $(o_1, t_1, o_2, t_2) \in \llbracket r'^k \rrbracket_C$. We split this case into three cases.

- 1) When $n = m$, then $(o_1, t_1, o_2, t_2) \in \llbracket r'[n, m] \rrbracket_C$ if and only if $(o_1, t_1, o_2, t_2) \in \llbracket r'^n \rrbracket_C$. Recall that the concatenation operator is associative, and that $\llbracket r'^n \rrbracket_C = \llbracket r' / r'^{n-1} \rrbracket_C = \llbracket r' / \dots / r' \rrbracket_C$ (n repetitions). Hence, if we define $l = \lfloor n/2 \rfloor$, then if n is even, $\llbracket r'^n \rrbracket_C = \llbracket (r'^l / r'^l) \rrbracket_C$, whereas if n is odd, $\llbracket r'^n \rrbracket_C = \llbracket (r'^l / r' / r'^l) \rrbracket_C$.

In the first case then, by the definition of concatenation, $(o_1, t_1, o_2, t_2) \in \llbracket (r'^l / r'^l) \rrbracket_C$ if and only if there exists a temporal object (o', t') such that $(o_1, t_1, o', t') \in \llbracket r'^l \rrbracket_C$ and $(o', t', o_2, t_2) \in \llbracket r'^l \rrbracket_C$. By induction hypothesis this is equivalent to having that there exists a temporal object (o', t') such that both $\text{TUPLEEVALSOLVE}(C, r'^l[l, l], (o_1, t_1, o', t'))$ and $\text{TUPLEEVALSOLVE}(C, r'^l[l, l], (o', t', o_2, t_2))$ return *true*, since $\llbracket r'^l \rrbracket_C = \llbracket r'^l[l, l] \rrbracket_C$. Since the algorithm iterates over all possible temporal objects to check this condition, if such temporal object exists, it will return *true*, if it does not, then it will reach the last line and return *false*.

The second case is similar, except that now we need two temporal objects as there are two concatenations, which means that $(o_1, t_1, o_2, t_2) \in \llbracket (r^l / r' / r^l) \rrbracket_C$ if and only if there exist two temporal objects (o', t') and (o'', t'') such that $(o_1, t_1, o', t') \in \llbracket r^l \rrbracket_C$, $(o', t', o'', t'') \in \llbracket r' \rrbracket_C$ and $(o'', t'', o_2, t_2) \in \llbracket r^l \rrbracket_C$. By induction hypothesis, and recalling again that $\llbracket r^l \rrbracket_C = \llbracket r^l[l, l] \rrbracket_C$, that means that $(o_1, t_1, o_2, t_2) \in \llbracket r \rrbracket_C$ if and only if there exist two temporal objects (o', t') and (o'', t'') such that the calls $\text{TUPLEEVALSOLVE}(C, r^l[l, l], (o_1, t_1, o', t'))$, $\text{TUPLEEVALSOLVE}(C, r', (o', t', o'', t''))$ and $\text{TUPLEEVALSOLVE}(C, r^l[l, l], (o'', t'', o_2, t_2))$ return *true*. Again, since the algorithm iterates over all possible pairs of temporal objects (o', t') and (o'', t'') to check this condition, if such temporal objects exist, it will return *true*, if it does not, then it will reach the last line and return *false*.

Since this recursion must stop at some point, the base case $n = 1$ is included, in which case r is $r'[1, 1]$, which is equivalent to r' , since in such case we know that $(o_1, o_2, t_1, t_2) \in \llbracket r \rrbracket_C$ if and only if $(o_1, t_1, o_2, t_2) \in \llbracket r' \rrbracket_C$. By hypothesis induction, this happens if and only if $\text{TUPLEEVALSOLVE}(C, r', (o_1, t_1, o_2, t_2))$ returns *true*, which is why the algorithm returns that result.

Finally, the recursion works for $n \geq 2$. The case when $n = 1$ is covered as a base case, so it only remains to look for the case when $n = 0$. For such case, recall that $r^0 = (\exists \vee \neg \exists)$, i.e., a test that is a tautology. Hence, $(o_1, t_1, o_2, t_2) \in \llbracket r \rrbracket_C$ if and only if $(o_1, t_1) = (o_2, t_2)$, which is what the algorithm tests.

- 2) When $n = 0$, the base case will be slightly different. We can assume that $n \neq m$ since the case where $n = m$ was already covered. Hence, the base case only needs to consider the value $m = 1$. In this case, we have that $\llbracket r \rrbracket_C = \llbracket r'[0, 1] \rrbracket_C = \llbracket r^0 \rrbracket_C \cup \llbracket r^1 \rrbracket_C$. As we already discussed, checking whether $(o_1, t_1, o_2, t_2) \in \llbracket r^0 \rrbracket_C$ comes down to checking whether $(o_1, t_1) = (o_2, t_2)$, and also $\llbracket r^1 \rrbracket_C = \llbracket r' \rrbracket_C$. By induction, we know that $(o_1, t_1, o_2, t_2) \in \llbracket r' \rrbracket_C$ if and only if $\text{TUPLEEVALSOLVE}(C, r', (o_1, t_1, o_2, t_2))$ returns *true*. Since the algorithm returns *true* if either of these conditions hold, this case is correctly covered.

As for the recursive case, it is very similar to the previous one. If we define again $l = \lfloor m/2 \rfloor$, we can notice that, if m is even, then $\llbracket r'[0, m] \rrbracket_C = \llbracket (r'[0, l] / r'[0, l]) \rrbracket_C$, whereas if m is odd, then $\llbracket r'[0, m] \rrbracket_C = \llbracket (r'[0, l] / r' / r'[0, l]) \rrbracket_C$.

To show the part when m is even, notice that $(o_1, t_1, o_2, t_2) \in \llbracket r \rrbracket_C$ if and only if $(o_1, t_1, o_2, t_2) \in \llbracket r^{j_i} \rrbracket_C$ for some $i \in [0, m]$. For every $i \in [0, m]$ there exist two integers, $j_i := \lfloor i/2 \rfloor$ and $k_i := \lceil i/2 \rceil$, both in the interval $[0, l]$, that satisfy that $j_i + k_i = i$. Since the concatenation operator is associative, this means that $\llbracket r^{j_i} \rrbracket_C = \llbracket (r^{j_i} / r^{k_i}) \rrbracket_C$, which in turn implies that $\llbracket r \rrbracket_C = \bigcup_{i=0}^m \llbracket r^{j_i} \rrbracket_C = \bigcup_{i=0}^m \llbracket (r^{j_i} / r^{k_i}) \rrbracket_C$.

Then, by definition of the concatenation operator, $(o_1, t_1, o_2, t_2) \in \bigcup_{i=0}^m \llbracket (r^{j_i} / r^{k_i}) \rrbracket_C$ if and only if there exists a temporal object (o', t') such that $(o_1, t_1, o', t') \in \llbracket r^{j_i} \rrbracket_C$ and $(o', t', o_2, t_2) \in \llbracket r^{k_i} \rrbracket_C$.

Since both j_i and k_i are bounded by l , $\llbracket r^{j_i} \rrbracket_C \subseteq \llbracket \bigcup_{i=0}^l r^{j_i} \rrbracket_C$, and $\llbracket r^{k_i} \rrbracket_C \subseteq \llbracket \bigcup_{i=0}^l r^{k_i} \rrbracket_C$, so any tuple (o, t, o', t') in $\llbracket r^{k_i} \rrbracket_C$ or $\llbracket r^{j_i} \rrbracket_C$ will also be in $\llbracket \bigcup_{i=0}^l r^{j_i} \rrbracket_C = \llbracket r'[0, l] \rrbracket_C$. Hence, $(o_1, t_1, o', t') \in \llbracket r^{j_i} \rrbracket_C$ implies that $(o_1, t_1, o', t') \in \llbracket r'[0, l] \rrbracket_C$ and $(o', t', o_2, t_2) \in \llbracket r^{k_i} \rrbracket_C$ implies that $(o', t', o_2, t_2) \in \llbracket r'[0, l] \rrbracket_C$. It can then be inferred that having that $(o_1, t_1, o_2, t_2) \in \bigcup_{i=0}^m \llbracket (r^{j_i} / r^{k_i}) \rrbracket_C$ can only hold if there exists a temporal object (o', t') satisfying that $(o_1, t_1, o', t') \in \llbracket r'[0, l] \rrbracket_C$ and $(o', t', o_2, t_2) \in \llbracket r'[0, l] \rrbracket_C$. As a result, $(o_1, t_1, o_2, t_2) \in \bigcup_{i=0}^m \llbracket (r^{j_i} / r^{k_i}) \rrbracket_C$ implicates that $(o_1, t_1, o_2, t_2) \in \llbracket (r'[0, l] / r'[0, l]) \rrbracket_C$. In consequence, we get that $\bigcup_{i=0}^m \llbracket (r^{j_i} / r^{k_i}) \rrbracket_C \subseteq \llbracket (r'[0, l] / r'[0, l]) \rrbracket_C$.

For the inverse inclusion, notice that if $(o_1, t_1, o_2, t_2) \in \llbracket (r'[0, l] / r'[0, l]) \rrbracket_C$, then there must exist a temporal object (o', t') and two integers j and k in $[0, l]$ such that $(o_1, t_1, o', t') \in \llbracket r^j \rrbracket_C$ and $(o', t', o_2, t_2) \in \llbracket r^k \rrbracket_C$, which implies that $(o_1, t_1, o_2, t_2) \in \llbracket (r^j / r^k) \rrbracket_C$. Again, since concatenation is associative, $\llbracket (r^j / r^k) \rrbracket_C = \llbracket r^{j+k} \rrbracket_C$, and because $j + k$ is at most n , $(o_1, t_1, o_2, t_2) \in \bigcup_{i=0}^n \llbracket r^i \rrbracket_C$, i.e., $(o_1, t_1, o_2, t_2) \in \llbracket r \rrbracket_C$. As a result, we get that $\llbracket (r'[0, l] / r'[0, l]) \rrbracket_C \subseteq \bigcup_{i=0}^n \llbracket (r^{j_i} / r^{k_i}) \rrbracket_C$.

Combining these two inclusions with the equality $\llbracket r \rrbracket_C = \bigcup_{i=0}^n \llbracket (r^{j_i} / r^{k_i}) \rrbracket_C$, we get that $\llbracket r \rrbracket_C = \llbracket (r'[0, l] / r'[0, l]) \rrbracket_C$.

To show the part where n is odd, notice that $(o_1, t_1, o_2, t_2) \in \llbracket r \rrbracket_C$ if and only if there exists an integer $i \in [0, m]$ such that $(o_1, t_1, o_2, t_2) \in \llbracket r^i \rrbracket_C$. Notice then that i can be written as the sum of three integers, $k_i := \lfloor i/2 \rfloor$ ($\in [0, l]$), $j_i := \lfloor i/2 \rfloor$ ($\in [0, l]$) and $s_i := (i \bmod 2)$ ($\in [0, 1]$). Since the concatenation operator is associative, this means that $\llbracket r^i \rrbracket_C = \llbracket (r^{j_i} / r^{s_i} / r^{k_i}) \rrbracket_C$, which in turn implies that $\llbracket r \rrbracket_C = \bigcup_{i=0}^m \llbracket r^i \rrbracket_C = \bigcup_{i=0}^m \llbracket (r^{j_i} / r^{s_i} / r^{k_i}) \rrbracket_C$.

Then, by definition of the concatenation operator, $(o_1, t_1, o_2, t_2) \in \bigcup_{i=0}^m \llbracket (r^{j_i} / r^{s_i} / r^{k_i}) \rrbracket_C$ if and only if there exist two temporal objects (o', t') and (o'', t'') such that $(o_1, t_1, o', t') \in \llbracket r^{j_i} \rrbracket_C$, $(o', t', o'', t'') \in \llbracket r^{s_i} \rrbracket_C$ and $(o'', t'', o_2, t_2) \in \llbracket r^{k_i} \rrbracket_C$.

Since both j_i and k_i are bounded by l , $\llbracket r^{j_i} \rrbracket_C \subseteq \llbracket \bigcup_{i=0}^l r^{j_i} \rrbracket_C$, and $\llbracket r^{k_i} \rrbracket_C \subseteq \llbracket \bigcup_{i=0}^l r^{k_i} \rrbracket_C$, so any tuple (o, t, o', t') in $\llbracket r^{k_i} \rrbracket_C$ or $\llbracket r^{j_i} \rrbracket_C$ will also be in $\llbracket \bigcup_{i=0}^l r^{j_i} \rrbracket_C = \llbracket r'[0, l] \rrbracket_C$. Similarly, any tuple in $\llbracket r^{s_i} \rrbracket_C$ will also be in $\llbracket r'[0, 1] \rrbracket_C$. Hence, $(o_1, t_1, o', t') \in \llbracket r^{j_i} \rrbracket_C$ implies that $(o_1, t_1, o', t') \in \llbracket r'[0, l] \rrbracket_C$, $(o', t', o'', t'') \in \llbracket r^{s_i} \rrbracket_C$ implies that $(o', t', o'', t'') \in \llbracket r'[0, 1] \rrbracket_C$ and $(o'', t'', o_2, t_2) \in \llbracket r^{k_i} \rrbracket_C$ implies that $(o'', t'', o_2, t_2) \in \llbracket r[0, l] \rrbracket_C$. It can then be inferred that having that $(o_1, t_1, o_2, t_2) \in \bigcup_{i=0}^m \llbracket (r^{j_i} / r^{s_i} / r^{k_i}) \rrbracket_C$ can only hold if there exist two temporal objects (o', t') and (o'', t'') satisfying that $(o_1, t_1, o', t') \in \llbracket r'[0, l] \rrbracket_C$, $(o', t', o'', t'') \in \llbracket r'[0, 1] \rrbracket_C$ and $(o'', t'', o_2, t_2) \in \llbracket r[0, l] \rrbracket_C$. As a result, $(o_1, t_1, o_2, t_2) \in \bigcup_{i=0}^m \llbracket (r^{j_i} / r^{s_i} / r^{k_i}) \rrbracket_C$ implicates that $(o_1, t_1, o_2, t_2) \in \llbracket (r'[0, l] / r'[0, 1] / r[0, l]) \rrbracket_C$. In consequence, we

get that $\bigcup_{i=0}^n \llbracket (r'^{j_i} / r'^{k_i}) \rrbracket_C \subseteq \llbracket (r'[0, l] / r'[0, 1] / r'[0, l]) \rrbracket_C$.

For the inverse inclusion, notice that if $(o_1, t_1, o_2, t_2) \in \llbracket (r'[0, l] / r'[0, 1] / r'[0, l]) \rrbracket_C$, then there must exist two temporal objects (o', t') and (o'', t'') , and three integers $j \in [0, l]$, $s \in [0, 1]$ and $k \in [0, l]$ such that $(o_1, t_1, o', t') \in \llbracket r'^j \rrbracket_C$, $(o'', t'') \in \llbracket r'^s \rrbracket_C$ and $(o'', t'', o_2, t_2) \in \llbracket r'^k \rrbracket_C$, which implies that $(o_1, t_1, o_2, t_2) \in \llbracket (r'^j / r'^s / r'^k) \rrbracket_C$. Again, since concatenation is associative, $\llbracket (r'^i / r'^s / r'^j) \rrbracket_C = \llbracket r'^{i+s+j} \rrbracket_C$, and because $i + s + j$ is at most n , $(o_1, t_1, o_2, t_2) \in \bigcup_{i=0}^n \llbracket r'^i \rrbracket_C$, i.e., $(o_1, t_1, o_2, t_2) \in \llbracket r \rrbracket_C$. As a result, we get that $\llbracket (r'[0, l] / r'[0, 1] / r'[0, l]) \rrbracket_C \subseteq \bigcup_{i=0}^n \llbracket (r'^{j_i} / r'^{k_i}) \rrbracket_C$.

As before, combining these two inclusions with the equality $\llbracket r \rrbracket_C = \bigcup_{i=0}^n \llbracket (r'^{j_i} / r'^{l_i} / r'^{k_i}) \rrbracket_C$, we get that $\llbracket r \rrbracket_C = \llbracket (r'[0, l] / r'[0, 1] / r'[0, l]) \rrbracket_C$. With these two results in mind then, i.e., that $\llbracket r'[0, m] \rrbracket_C = \llbracket (r'[0, l] / r'[0, l]) \rrbracket_C$ when m is even, whereas if m is odd, then $\llbracket r'[0, m] \rrbracket_C = \llbracket (r'[0, l] / r'[0, l]) \rrbracket_C$, we know that $(o_1, t_1, o_2, t_2) \in \llbracket r'[0, m] \rrbracket_C$ if and only if (i) m is even and there exists a temporal object (o', t') such that $(o_1, t_1, o', t') \in \llbracket r'[0, l] \rrbracket_C$ and $(o', t', o_2, t_2) \in \llbracket r'[0, l] \rrbracket_C$, or (ii) m is odd and there exist two temporal objects (o', t') and (o'', t'') such that $(o_1, t_1, o', t') \in \llbracket r'[0, l] \rrbracket_C$, $(o', t', o'', t'') \in \llbracket r'[0, 1] \rrbracket_C$ and $(o'', t'', o_2, t_2) \in \llbracket r'[0, l] \rrbracket_C$.

By induction, (i) holds if and only if there exists a temporal object (o', t') such that both $\text{TUPLEEVALSOLVE}(C, r'[0, l], (o_1, t_1, o', t'))$ and $\text{TUPLEEVALSOLVE}(C, r'[0, l], (o', t', o_2, t_2))$ return *true*. Since the algorithm iterates over all temporal objects (o', t') for this case, and then checks that both those conditions are met to return *true*, it will return *true* if (i) holds, and it will reach the last line and return *false* if no such pair existed.

Also by induction, (ii) holds if and only if there exist two temporal objects (o', t') and (o'', t'') such that the three calls $\text{TUPLEEVALSOLVE}(C, r'[0, l], (o_1, t_1, o', t'))$, $\text{TUPLEEVALSOLVE}(C, r'[0, l], (o', t', o'', t''))$ and $\text{TUPLEEVALSOLVE}(C, r'[0, l], (o'', t'', o_2, t_2))$ return *true*. Here again, since the algorithm iterates over all pairs of temporal objects (o', t') and (o'', t'') and sees if these three conditions are met to return *true*, it will return *true* if (ii) holds, and it will reach the last line and return *false* otherwise.

Hence, when $n = 0$, the algorithm also returns *true* if and only if $(o_1, t_1, o_2, t_2) \in \llbracket r \rrbracket_C$

- 3) When $m \neq n$ and $n \neq 0$, then $(o_1, t_1, o_2, t_2) \llbracket r'[n, m] \rrbracket_C$ if and only if there exists $i \in [n, m]$ such that $(o_1, t_1, o_2, t_2) \llbracket r'^i \rrbracket_C$. In this case, $i = n + d$ for some $d \in [0, m - n]$, and since the concatenation operator is associative, $(o_1, t_1, o_2, t_2) \llbracket r'^i \rrbracket_C$ if and only if there exists (o', t') such that $(o_1, t_1, o', t') \in \llbracket r'^n \rrbracket_C$ and $(o', t', o_2, t_2) \in \llbracket r'^d \rrbracket_C$. By induction, and since $\llbracket r'^n \rrbracket_C = \llbracket r'[n, n] \rrbracket_C$, $(o_1, t_1, o', t') \in \llbracket r'^n \rrbracket_C$ if and only if $\text{TUPLEEVALSOLVE}(C, r'[n, n], (o_1, t_1, o', t'))$. Similarly, $(o', t', o_2, t_2) \in \llbracket r'^d \rrbracket_C$ for some $d \in [0, m - n]$ if and only if $(o', t', o_2, t_2) \in \llbracket r'[0, m - n] \rrbracket_C$, which by induction holds if and only if $\text{TUPLEEVALSOLVE}(C, r'[0, m - n], (o', t', o_2, t_2))$.

Together, this means that $(o_1, t_1, o_2, t_2) \in \llbracket r \rrbracket_C$ if and only if there exists a temporal object (o', t') such that $\text{TUPLEEVALSOLVE}(C, r'[n, n], (o_1, t_1, o', t'))$ and $\text{TUPLEEVALSOLVE}(C, r'[0, m - n], (o', t', o_2, t_2))$. Since the algorithm iterates over all temporal objects (o', t') and checks if these conditions are met to return *true*, it will return *true* if $(o_1, t_1, o_2, t_2) \in \llbracket r \rrbracket_C$ and it will reach the last line and return *false* otherwise.

Finally, for an expression r matching a regular expressions with numerical occurrence indicators of the form $r'[n, _]$, recall that we showed in Section C-A that $\llbracket r'[n, _] \rrbracket_C = \llbracket r'[n, m] \rrbracket_C$, where the expression $m := n + (|\Omega| + |V \cup E|)^2$ is polynomial in the size of the original input. By induction, $(o_1, t_1, o_2, t_2) \in \llbracket r'[n, m] \rrbracket_C$ if and only if $\text{TUPLEEVALSOLVE}(C, r'[n, m], (o_1, t_1, o_2, t_2))$ returns *true*, which is what the algorithm returns for this case. Altogether, we proved that TUPLEEVALSOLVE works in polynomial space, and that $\text{TUPLEEVALSOLVE}(C, r, (o_1, t_1, o_2, t_2))$ returns *true* if and only if $(o_1, t_1, o_2, t_2) \in \llbracket r \rrbracket_C$. This concludes the proof of the theorem.

APPENDIX D

ALLOWING NUMERICAL OCCURRENCE INDICATORS ONLY IN THE AXES: ADDITIONAL COMPLEXITY RESULTS

A natural question is whether there is a restriction on $\text{NavL}[\text{NOI}]$ that can reduce the complexity of the evaluation problem but is still expressive enough to represent some useful queries. At this point, a restriction used in the study of XPath comes to the rescue [52]. In what follows, we show that the complexity of the evaluation problem is lower if numerical occurrence indicators are only allowed in the axes.

Theorem D.1. $\text{Eval}(\text{ITPG}, \text{NavL}[\text{ANOI}])$ is NP-complete.

Proof. To show NP-hardness, consider the following decision problem called Subset Sum (SUBSET-SUM), which is known to be NP-complete [71]:

Problem:	SUBSET-SUM
Input:	A finite set of integers $A \subseteq \mathbb{N}$, and a positive integer $S \in \mathbb{N}$
Output:	<i>true</i> if there exists a subset $A' \subseteq A$ of A such that $\sum_{a \in A'} a = S$.

Given a set $A \subseteq \mathbb{N}$, and an integer $S \in \mathbb{N}$, the goal is to provide a polynomial-time algorithm that returns an ITPG C , a tuple (o_1, t_1, o_2, t_2) , and an expression r in $\text{NavL}[\text{ANOI}]$ such that $(o_1, t_1, o_2, t_2) \in \llbracket r \rrbracket_C$ if and only if there exists $A' \subseteq A$ such

that $\sum_{a \in A'} a = S$. More specifically, C will be the ITPG $(\Omega, N, E, \rho, \lambda, \xi, \sigma)$ where $\Omega = [0, S]$, $N = \{v\}$, $E = \emptyset$, ρ is an empty function, $\lambda(v) = l$, $\xi(v) = \{\{0, S\}\}$ and σ is an empty function. In other words, C is an ITPG consisting of only one node existing from time 0 to time S , with no edges or properties. The tuple (o_1, t_1, o_2, t_2) in our reduction will be given by $(v, 0, v, S)$. Moreover, assuming that $A = \{a_1, \dots, a_n\}$, expression r is defined as follows:

$$r = (\mathbf{N}[a_1, a_1] + \mathbf{N}[0, 0]) / \dots / (\mathbf{N}[a_n, a_n] + \mathbf{N}[0, 0])$$

Notice that ITPG C , expression r in NavL[ANOI] and tuple $(v, 0, v, S)$ can be computed in polynomial time in the sizes of A and S . Besides, it is straightforward to prove that $(v, 0, v, S) \in \llbracket r \rrbracket_C$ if and only if there exists $A' \subseteq A$ such that $\sum_{a \in A'} a = S$. This concludes the of NP-hardness of $\text{Eval(ITPG, NavL[ANOI])}$.

To show that this problem is NP-complete, it only remains to show that the problem is also in NP. We present a nondeterministic algorithm that works in polynomial time, $\text{TUPLEEVALSOLVE_ANOI}$, that, given an ITPG C , an expression r in NavL[ANOI] and a pair of temporal objects (o_1, t_1, o_2, t_2) , has a run that returns *true* if and only if $(o, t, o', t') \in \llbracket r \rrbracket_C$. This procedure is presented in Algorithm 6.

Algorithm 6: $\text{TUPLEEVALSOLVE_ANOI}(C, (o_1, t_1, o_2, t_2), r)$ (part I)

Input : An ITPG $C = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$, an expression r in NavL[ANOI] and a pair of temporal objects (o_1, t_1, o_2, t_2)

Output : *true* if $(o, t, o', t') \in \llbracket r \rrbracket_C$

```

1 if  $r$  is a test then
2   if  $(o_1, t_1) \neq (o_2, t_2)$  then
3     return false
4   else if  $r = \text{Node}$  then
5     return  $(o_1 \in N)$ 
6   else if  $r = \text{Edge}$  then
7     return  $(o_1 \in E)$ 
8   else if  $r = \ell$  for some  $\ell \in \text{Lab}$  then
9     return  $(\lambda(o_1) = \ell)$ 
10  else if  $r = p \mapsto v$  for some  $p \in \text{Prop}$  and  $v \in \text{Val}$  then
11    foreach valued interval  $(v', I) \in \sigma(o_1, p)$  do
12      if  $t_1 \in I$  then
13        return  $v' = v$ 
14      return false
15  else if  $r = < k$  with  $k \in \Omega$  then
16    return  $(t_1 < k)$ 
17  else if  $r = \exists$  then
18    foreach interval  $I \in \xi(o_1)$  do
19      if  $t_1 \in I$  then
20        return true
21    return false
22  else if  $r = (\text{test}_1 \vee \text{test}_2)$  then
23    return  $\text{TUPLEEVALSOLVE}(C, (o_1, t_1, o_1, t_1), \text{test}_1)$  or  $\text{TUPLEEVALSOLVE}(C, (o_1, t_1, o_1, t_1), \text{test}_2)$ 
24  else if  $r = (\text{test}_1 \wedge \text{test}_2)$  then
25    return  $\text{TUPLEEVALSOLVE}(C, (o_1, t_1, o_1, t_1), \text{test}_1)$  and  $\text{TUPLEEVALSOLVE}(C, (o_1, t_1, o_1, t_1), \text{test}_2)$ 
26  else if  $r = (\neg r')$  then
27    return not  $\text{TUPLEEVALSOLVE}(C, (o_1, t_1, o_1, t_1), r')$ 
28 else if  $r = \mathbf{N}$  then
29   return  $(o_1 = o_2 \text{ and } t_2 = t_1 + 1)$ 
30 else if  $r = \mathbf{P}$  then
31   return  $(o_1 = o_2 \text{ and } t_2 = t_1 - 1)$ 
32 else if  $r = \mathbf{F}$  then
33   return  $(t_1 = t_2 \text{ and } ((o_1 \in E \text{ and } o_2 = \text{tgt}(o_1)) \text{ or } (o_2 \in E \text{ and } o_1 = \text{src}(o_2))))$ 
34 else if  $r = \mathbf{B}$  then
35   return  $(t_1 = t_2 \text{ and } ((o_1 \in E \text{ and } o_2 = \text{src}(o_1)) \text{ or } (o_2 \in E \text{ and } o_1 = \text{tgt}(o_2))))$ 

```

TUPLEEVALSOLVE_ANOI is very similar to TUPLEEVALSOLVE, so we will not discuss in detail what it does. Instead, we give an intuition of what the differences are that allow to return the right answer in non-deterministic polynomial time, instead of polynomial space. First, notice that if r is a test, then the algorithm works by solving basic tests efficiently, and then conjunctions, disjunctions and negations of tests are solved just by using directly the definition of these Boolean connectives. Hence, unlike what happens in the presence of path conditions, where we can have nested expressions with existential conditions and negations of existential conditions, sub-expressions for tests are efficiently solved by TUPLEEVALSOLVE_ANOI. Second, notice that for spatial navigation, we write the problem in terms of the reachability problem for graphs in a number of steps in a set $\{n, \dots, m\}$. This problem can be efficiently solved by using exponentiation by squaring on the adjacency matrix. Besides, notice that for spatial navigation expressions in $\text{NavL}[\text{ANOI}]$, we need to consider as many new objects as there are in $V \cup E$ since the time is fixed, which is why expressions $\mathbf{F}[n, _]$ and $\mathbf{B}[n, _]$ are equivalent to $\mathbf{F}[n, m]$ and $\mathbf{B}[n, m]$, respectively, with $m = n + |N \cup E| = n + |N| + |E|$. Finally, polynomial time executions are ensured by the non-deterministic guess for (o', t') in Line 64, and the fact that the depth of the recursion tree is linear with respect to the size of the input expression r . In particular, we do a single non-deterministic guess in Line 64, instead of an exponential number of attempts (with respect to the size of the representation of Ω) that would be necessary to find the right pair (o', t') in a deterministic algorithm.

Algorithm 7: TUPLEEVALSOLVE_ANOI($C, (o_1, t_1, o_2, t_2), r$) (part II)

```

36 else if  $r = \mathbf{N}[n, m]$  then
37   | return  $(o_1 = o_2 \text{ and } n \leq (t_2 - t_1) \leq m)$ 
38 else if  $r = \mathbf{P}[n, m]$  then
39   | return  $(o_1 = o_2 \text{ and } n \leq (t_1 - t_2) \leq m)$ 
40 else if  $r = \mathbf{F}[n, m]$  then
41   | if  $t_1 \neq t_2$  then
42     | return false
43   | else
44     | Let  $G = (N', E')$  be the graph where:
45     |   •  $N' = (N \cup E)$ 
46     |   •  $E' = \{(v, e) \in N \times E \mid \text{src}(e) = v\} \cup \{(e, v) \in E \times N \mid \text{tgt}(e) = v\}$ 
47     |   return  $o_2$  is reachable from  $o_1$  in  $k$  steps in  $G$ , where  $k \in \{n, \dots, m\}$ 
48 else if  $r = \mathbf{B}[n, m]$  then
49   | if  $t_1 \neq t_2$  then
50     | return false
51   | else
52     | Let  $G = (N', E')$  be the graph where:
53     |   •  $N' = (N \cup E)$ 
54     |   •  $E' = \{(v, e) \in N \times E \mid \text{tgt}(e) = v\} \cup \{(e, v) \in E \times N \mid \text{src}(e) = v\}$ 
55     |   return  $o_2$  is reachable from  $o_1$  in  $k$  steps in  $G$ , where  $k \in \{n, \dots, m\}$ 
56 else if  $r = \mathbf{N}[n, \_]$  then
57   | return  $(o_1 = o_2 \text{ and } n \leq (t_2 - t_1))$ 
58 else if  $r = \mathbf{P}[n, \_]$  then
59   | return  $(o_1 = o_2 \text{ and } n \leq (t_1 - t_2))$ 
60 else if  $r = \mathbf{F}[n, \_]$  then
61   |  $m \leftarrow n + |N| + |E|$ 
62   | return TUPLEEVALSOLVE_ANOI( $C, (o_1, t_1, o_2, t_2), \mathbf{F}[n, m]$ )
63 else if  $r = \mathbf{B}[n, \_]$  then
64   |  $m \leftarrow n + |N| + |E|$ 
65   | return TUPLEEVALSOLVE_ANOI( $C, (o_1, t_1, o_2, t_2), \mathbf{B}[n, m]$ )
66 else if  $r = (r_1 + r_2)$  then
67   | Guess  $i \in \{1, 2\}$ 
68   | return TUPLEEVALSOLVE_ANOI( $C, (o_1, t_1, o_2, t_2), r_i$ )
69 else if  $r = (r_1 / r_2)$  then
70   | Guess  $(o', t') \in (N \cup E) \times \Omega$ 
71   | return TUPLEEVALSOLVE_ANOI( $C, (o_1, t_1, o', t'), r_1$ ) and TUPLEEVALSOLVE_ANOI( $C, (o', t', o_2, t_2), r_2$ )

```

□

We have that $\text{Eval}(\text{ITPG}, \text{NavL}[\text{PC}])$ can be solved in polynomial time by Theorem V.1, and we know that $\text{Eval}(\text{ITPG}, \text{NavL}[\text{PC}])$ is NP-complete by Theorem D.1. A natural question then is whether the complexity remains the same if these functionalities are combined. Notice that $\text{Eval}(\text{ITPG}, \text{NavL}[\text{PC}, \text{NOI}])$ is PSPACE-complete, so a positive answer to this question means a significant decrease in the complexity of the query evaluation problem. Unfortunately, we show that the complexity of the entire language does not decrease by restricting numerical occurrence indicators to occur only in the axes.

Theorem D.2. $\text{Eval}(\text{ITPG}, \text{NavL}[\text{PC}, \text{ANOI}])$ is PSPACE-complete.

Proof. Notice that every expression in $\text{NavL}[\text{PC}, \text{ANOI}]$ is also an expression in $\text{NavL}[\text{PC}, \text{NOI}]$, so PSPACE-membership follows immediately from Theorem V.1. Hence, we only need to prove PSPACE-hardness for $\text{NavL}[\text{PC}, \text{ANOI}]$.

To show this, we replace test expressions r_i in the proof in Section C-D by an expression in $\text{NavL}[\text{PC}, \text{ANOI}]$ that will be denoted by q_i . Expression q_i is defined in such a way that, for every time t , it holds that $(v, t, v, t) \in \llbracket q_i \rrbracket_C$ if and only if $(v, t, v, t) \in \llbracket r_i \rrbracket_C$, i.e., if and only if $\text{bit}(i, t)$ is *true*, where $\text{bit}(i, t)$ holds if the i -th bit of time t (from right to left when written in its binary representation) is 1. More precisely, expression q_i is defined as follow:

$$q_i = ? \left(((\mathbf{P}[0, 0] + \mathbf{P}[2^n, 2^n]) / \dots / (\mathbf{P}[0, 0] + \mathbf{P}[2^i, 2^i])) / (< 2^i \wedge \neg < 2^{i-1}) \right)$$

Notice that the length of the representation 2^k is k , so the whole expression q_i has length $O(n^2)$, which is polynomial with respect to the size of ψ . Also, notice that as before, we only need a polynomial number of these expressions for the reduction, and no further nesting of numerical occurrence indicators is required for the proof. Hence, we only need to prove that $(v, t, v, t) \in \llbracket q_i \rrbracket_C$ if and only if $\text{bit}(i, t)$ is *true*. Recall that for this reduction, C is an ITPG consisting of only one node v , existing from time 0 to time $2^n - 1$, with no edges or properties, so any temporal object considered will be of the form (v, t) .

First, notice that q_i is a path test, so $(v, t, v, t) \in \llbracket q_i \rrbracket_C$ if and only if there exists a time point t' such that

$$(v, t, v, t') \in \llbracket ((\mathbf{P}[0, 0] + \mathbf{P}[2^n, 2^n]) / \dots / (\mathbf{P}[0, 0] + \mathbf{P}[2^i, 2^i])) / (< 2^i \wedge \neg < 2^{i-1}) \rrbracket_C$$

As in **Step 1** of Section C-D, since the last part of the expression is a test, this is equivalent to the existence of a time point t' such that $(v, t') \models (< 2^i \wedge \neg < 2^{i-1})$, i.e., the i -th bit of t' is 1 and

$$(v, t, v, t') \in \llbracket (\mathbf{P}[0, 0] + \mathbf{P}[2^n, 2^n]) / \dots / (\mathbf{P}[0, 0] + \mathbf{P}[2^i, 2^i]) \rrbracket_C \quad (8)$$

We now prove that $(v, t, v, t) \in \llbracket q_i \rrbracket_C$ if and only if $\text{bit}(i, t)$ is *true*. To show direction (\Leftarrow) , suppose that $\text{bit}(i, t)$ is *true*. Notice then that $(v, t_1, v, t_2) \in \llbracket (\mathbf{P}[0, 0] + \mathbf{P}[2^k, 2^k]) \rrbracket_C$, if and only if $t_2 = t_1$ or $t_2 = t_1 - 2^k$. In particular, if the $(k+1)$ -th bit of t_1 is 1, then $t_2 = t_1 - 2^k$ has a binary representation that is equal to that of t_1 except on the $(k+1)$ -th bit, and $(v, t_1, v, t_2) \in \llbracket (\mathbf{P}[0, 0] + \mathbf{P}[2^k, 2^k]) \rrbracket_C$. Similarly, if the $(k+1)$ -th bit of t_1 is 0, then $t_2 = t_1$ has a binary representation that is equal to that of t_1 , and also $(v, t_1, v, t_2) \in \llbracket (\mathbf{P}[0, 0] + \mathbf{P}[2^k, 2^k]) \rrbracket_C$. As in Section C-D, given $b \in \{\text{true}, \text{false}\}$, let $\mathbb{1}_b$ be 1 if $b = \text{true}$, and be 0 otherwise. Moreover, define the sequence of time points t_{n+1}, \dots, t_i such that $t_{n+1} = t$ and $t_k = t_{k+1} - \mathbb{1}_{\text{bit}(k+1, t)} \cdot 2^k$ for $k \in \{i, \dots, n\}$. Then for every $k \in \{i, \dots, n\}$, it holds that $(v, t_k, v, t_{k+1}) \in \llbracket (\mathbf{P}[0, 0] + \mathbf{P}[2^k, 2^k]) \rrbracket_C$, and in particular, $t' = t - \sum_{k=i}^n \mathbb{1}_{\text{bit}(k+1, t)} \cdot 2^k$ satisfies (8). Therefore, if the i -th bit of t is 1, then the i -th bit of t' will be 1 as well. Hence, given $\text{bit}(i, t)$ is *true*, we conclude that $(v, t, v, t) \in \llbracket q_i \rrbracket_C$, since for $t' = t - \sum_{k=i}^n \mathbb{1}_{\text{bit}(k+1, t)} \cdot 2^k$, equation (8) holds and $(v, t') \models (< 2^i \wedge \neg < 2^{i-1})$.

To show direction (\Rightarrow) , suppose that $(v, t, v, t) \in \llbracket q_i \rrbracket_C$. Then there exists a time point t' such that (8) holds, which only holds if there exists a sequence of time points t_{n+1}, \dots, t_i where $t_{n+1} = t$ and either $t_k = t_{k+1}$ or $t_k = t_{k+1} - 2^k$ for $k \in \{i, \dots, n\}$, and $t_i = t'$. Notice that for such values for k , 2^k is a multiple of 2^i , so $t' = t + d \cdot 2^i$ for some integer d . We conclude that the i -th bit of t is equal to 1 if and only if the i -th bit of t' is equal to 1. Moreover, $(v, t') \models (< 2^i \wedge \neg < 2^{i-1})$, so the i -th bit of t' is indeed equal to 1, so $\text{bit}(i, t)$ must be equal to *true*.

From the previous paragraphs, we conclude that $(v, t, v, t) \in \llbracket q_i \rrbracket_C$ if and only if $\text{bit}(i, t)$ is *true*. Hence, by replacing r_i with q_i in the proof of Section C-D, we deduce that $\text{NavL}[\text{PC}, \text{ANOI}]$ is also PSPACE-hard, which was to be shown. \square

APPENDIX E SUPPLEMENTARY EXPERIMENTAL RESULT

In section VII-A we discussed the impact of TGraphs size on query execution, and pointed out that the trends presented in Figure 2 can be explained by the size of output. To study this, we computed the increase in output size for graphs G2-G6 (with between 2,000 and 10,000 nodes, as summarized in Table I) relative to the size of the output for G1 (with 1,000 nodes), for each query. Figures 7 (a) and (b) show this result. In these figures, the x -axis shows the number of nodes in each graph, and the y -axis shows the relative size of output bindings table, in comparison to the output of the same query over G1. Similarly to Figure 2, it can be observed that the output size for all queries except Q5, Q9, Q10, Q11, and Q12 follows a linear trend. For Q5, Q9, Q10, Q11 and Q12, the output size increases quadratically.

Figure 7 (c) gives another presentation of these results. Here, in addition to computing the output size relative to G1 for each query (shown on the y -axis), we also computed the execution time relative to G1 (shown on the x -axis). This plot shows that relative query execution time and relative increase in output size are highly correlated for all queries, and for the majority of our queries we have perfect correlation.

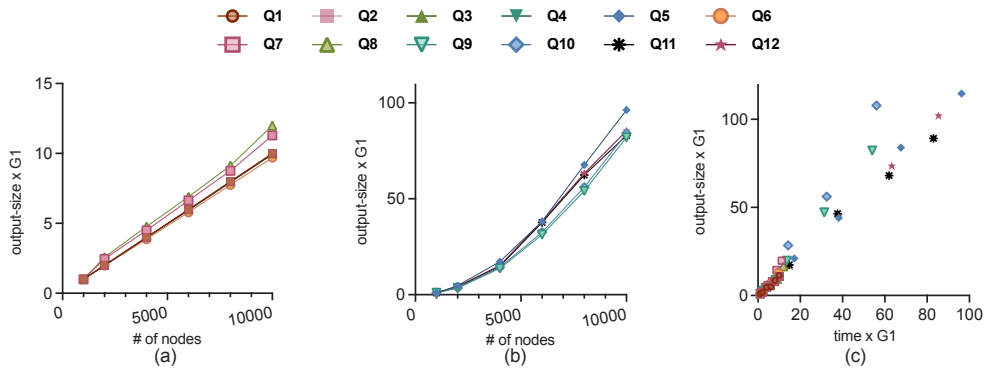


Fig. 7. Relationship between input size, output size, and query execution time for all queries. Execution time and output size are computed for graphs G2-G6 (with between 2,000 and 10,000 nodes) in Table I, in proportion to these quantities for graph G1 (with 1,000 nodes).

APPENDIX