Querying Graph Databases with XPath

Leonid Libkin
University of Edinburgh
libkin@inf.ed.ac.uk

Wim Martens
Universität Bayreuth
wim.martens@uni-bayreuth.de

Domagoj Vrgoč
University of Edinburgh
domagoj.vrgoc@ed.ac.uk

ABSTRACT

XPath plays a prominent role as an XML navigational language due to several factors, including its ability to express queries of interest, its close connection to yardstick database query languages (e.g., first-order logic), and the low complexity of query evaluation for many fragments. Another common database model — graph databases — also requires a heavy use of navigation in queries; yet it largely adopts a different approach to querying, relying on reachability patterns expressed with regular constraints.

Our goal here is to investigate the behavior and applicability of XPath-like languages for querying graph databases, concentrating on their expressiveness and complexity of query evaluation. We are particularly interested in a model of graph data that combines navigation through graphs with querying data held in the nodes, such as, for example, in a social network scenario. As navigational languages, we use analogs of core and regular XPath and augment them with various tests on data values. We relate these languages to first-order logic, its transitive closure extensions, and finite-variable fragments thereof, proving several capture results. In addition, we describe their relative expressive power. We then show that they behave very well computationally: they have a low-degree polynomial combined complexity, which becomes linear for several fragments. Furthermore, we introduce new types of tests for XPath languages that let them capture first-order logic with data comparisons and prove that the low complexity bounds continue to apply to such extended languages. Therefore, XPath-like languages seem to be very well-suited to query graphs.

Categories and Subject Descriptors

F.4.1 [**Mathematical logic and formal languages**]: Mathematical logic; H.2.3 [**Database management**]: Languages—*Query Languages*

General Terms

Theory, Languages, Algorithms

Keywords

XPath, Graph Databases, data values

1. INTRODUCTION

Managing graph-structured data has been an active topic over the past few years; there are multiple existing systems, both proprietary and open-source, and there is a growing body of research literature on graph databases (see, e.g., a survey [5]). There are numerous uses in modern applications whose data structure is naturally represented as graphs: for instance, RDF triples are typically viewed as edges in labeled graphs [27, 33, 37] and so are connections between people in social networks [21, 39, 40]. The Semantic Web and social networks are often cited as the key application areas of graph databases, but there are many others such as biology, network traffic, crime detection, and modeling object-oriented data.

When it comes to querying graph databases, one can, of course, ask standard relational queries, for instance, about information on people in a social network. What makes graph databases different, however, is the ability to ask queries about their *topology*, essentially looking for reachability patterns and, more generally, subgraph patterns [11, 21, 20]. A basic building block for such queries is typically a *regular path query*, or an *RPQ*, that selects nodes connected by a path described by a regular language over the labeling alphabet [19]. Extensions of RPQs with more complex patterns, backward navigation, relations over paths, and mixing labels and data in nodes have been studied extensively too [1, 6, 7, 11, 12, 18, 32].

Over the past decade, navigational queries have been studied in depth in a different framework extending the relational model, namely in XML. Most formalisms for describing and querying XML crucially depend on its path language XPath [44]. The goal of XPath is seemingly very similar to the goal of many queries in graph databases: it describes properties of paths, taking into account both their purely navigational properties and data that is found in XML documents. The popularity of XPath is largely due to several factors:

- it defines many properties of paths that are relevant for navigational queries;

- it achieves expressiveness that relates naturally to yardstick languages for databases (such as first-order logic, its fragments, or extensions with some form of recursion); and

- it has good computational properties over XML, notably tractable combined complexity for many fragments and even linear-time complexity for some of them.

In view of these desirable properties, it is natural to ask whether XPath-like languages can achieve the right balance of expressiveness and complexity of query evaluation in the context of graph databases. This is the question we address in this paper.

There appear to be two ways to use XPath as a graph database language. The first possibility is to essentially stick to the idea of RPQs and use XPath to describe paths between nodes. While XPath on words with data is well understood by now [9, 22], this idea has a significant drawback: in the presence of data, evaluating RPQs quickly becomes intractable [32], ruling out XPath as an add-on to RPQs.

A different approach is to apply XPath queries to the entire graph database, rather than paths selected by RPQs and similar queries. This is the approach we pursue. To a limited extent it was tried before. On the practical side, XPath-like languages have been used to query graph data (e.g., [13, 26]), without any analysis of their expressiveness and complexity, however. On the theoretical side, several papers investigated XPath-like languages from the modal perspective, dropping the assumption that they are evaluated on trees, for instance, [2, 35]. In fact, this was mainly done in the context of semi-structured data and preceded much of XPath investigation in the XML literature. The focus of papers such as [1, 2, 35] is primarily on static analysis (containment) and, in general, the settings disregard data values in graph databases.

Thus, our goal is to investigate how XPath-languages can be used to query graph databases. In particular, we want to understand both the navigational querying power of such languages, and their ability to handle navigation and data together in graph databases. In this investigation, we can take advantage of the vast existing XML literature on algorithmic and language-theoretic aspects of XPath.

Of course there is no such thing as *the* XPath: the language comes in many shapes and flavors. To start with, languages can talk about purely structural properties of documents, or they can add tests based on data values carried by documents. We have the same dichotomy for graph databases: in fact most earlier formalisms dealt with navigational queries [7, 11, 18, 19], but more recently extensions to data values have been looked at [20, 32].

The second key parameter is the expressiveness of navigational querying. The basic language one usually starts with is *core* XPath [16, 25]. It can also come in several versions, and what we use as the basic language here is essentially an adaption of Core XPath 2.0 [16] to graph databases. The reason behind it is the equivalence of the language to first-order logic, the yardstick language for relational databases (and we shall extend the equivalence to graph databases). For restricting expressiveness, we shall look at positive fragments (again, as is common in the XPath literature). For giving the language more power, we look at adding the transitive closure operator, obtaining an analog of *regular* XPath [12, 14, 16] which itself has close connections to PDL [28].

Flavor of the languages. We use several versions of XPath-like languages for graph databases. Like XPath (or closely related logics such as PDL and CTL*), they have node tests and path formulae, and as the basic axes they use letters from the alphabet labeling graph edges. For instance, $a^* \cdot (b^-)^*$ finds pairs of nodes connected by a path that starts with a-edges in the forward direction, followed by b-edges in the backward direction. The reader familiar with conjunctive RPQs will immediately recognize one in this expression, but the expressiveness of languages we consider is not limited to such queries. Formulae may include node tests: for instance, $a^*[c] \cdot (b^-)^*$ modifies the above expression by requiring that the node where the a-labels switch to b-labels also has an outgoing c-edge. And crucially, node tests can refer to data values and have XPath-like conditions over them. For instance, the expression $a^*[=5] \cdot (b^-)^*$ checks if the data value in that intermediate node is 5, and $a^*[\langle a = b \rangle] \cdot (b^-)^*$ checks if that node has two outgoing edges, labeled a and b, to nodes that store the same data value.

We define several versions of XPath for graph databases. The core language is denoted by GXPath_{core} and the analog of regular XPath by GXPath_{reg}. We augment them by different types of tests on data values, such as testing for constant values (like $[=5]$), or for comparisons of data values at the end of paths (like $[\langle a = b \rangle]$).

Summary of the results. We start by studying the expressive power. The first set of results concerns with pure navigational power (no data-value comparisons). It turns out that GXPath_{core} captures precisely FO^3, first-order logic with 3 variables, like its analog (core XPath 2.0) on trees. The difference, though, is that on graphs $\mathrm{FO} \neq \mathrm{FO}^3$, but on trees the two are the same. The proof establishes connection with relation algebra [43] which was recently studied in connection with pure navigational querying of graph databases, but from a rather different angle (see [23, 24] which considered relative expressiveness of fragments of relation algebra based on sets of operators).

Note that on trees there is another way of capturing FO, by means of *conditional* XPath [36], which adds the until-operator. We show that on graphs the analog of conditional XPath goes beyond FO.

When we move to GXPath_{reg}, we show that the positive fragment of it captures precisely the *nested regular expressions* [38], proposed as the navigational mechanism for SPARQL. This further confirms the usefulness of XPath for graph querying. Full GXPath_{reg} is more expressive and corresponds to a fragment of the transitive closure logic. We also show that it is incompatible with other graph languages

such as RPQs and several of their extensions.

With data value comparisons, we show that adding the two types of node tests described above increases expressiveness and we provide the exact comparisons of the power of language fragments with different types of tests. Even the strongest tests do not give $\mathsf{GXPath}_{\mathrm{core}}$ the power to capture FO^3 with data value comparisons, but we produce a different type of tests that elevate the language to the full power of FO^3. These were introduced in [32]: an example of such a test is $a_=$, selecting a-labeled edges between nodes with the same value.

We then move to the study of the complexity of XPath languages. We further extend them with numerical path formulae: for instance, $a^{n,m}$, for $n < m$, says that two nodes are reachable by a path of as whose length is between n and m. These comparisons, proposed in the SPARQL recommendations [29], do not affect expressiveness, but they can make expressions much more succinct [33].

We show that the complexity of *all* XPath languages on graphs inherits nice properties from XPath on trees, due to the 'modal' nature of the language: the combined complexity is always polynomial. Even more, it is always a low-degree polynomial. In fact, the query complexity is linear for all the fragments we consider. The data complexity is not worse than quadratic for navigational $\mathsf{GXPath}_{\mathrm{reg}}$ and linear for its positive fragments. With data comparisons added, data complexity becomes quadratic (or better) again. When numerical path formulae are added, the data complexity is not worse than cubic.

The main conclusion is that XPath-like languages over graph databases should not be overlooked due to the combination of their expressiveness and low complexity of query evaluation.

Remark. Many ideas behind XPath came from logics initially designed for arbitrary labeled transition systems, for instance PDL and CTL*. So, in a way, adapting XPath to graph databases, whose underlying model is, in essence, labeled transition systems, may look like going back to the origins. Nonetheless, this is not quite so, and there are indeed results to show that are specifically tailored to the graph database context.

To start with, we concentrate on graphs that carry *data*: this is crucial in the database scenario, but is generally disregarded in verification and model checking. Handling data has been studied extensively in the XPath context, so we can combine both model-checking techniques for navigational features with XPath techniques for handling data. Some of the features, such as counting, are specifically added in response to SPARQL recommendations.

The second distinction is that there is a mismatch between features naturally required by logics of programs and by logics for querying graph data. Even though an occasional fragment of an XPath-like language may coincide with an existing logic (e.g., PDL is what becomes navigational path-positive graph XPath in our classification), most of the time

we concentrate on languages that do not have precise counterparts on the program logic side.

Organization. We define graph databases in Section 2. In Section 3 we introduce our XPath-like languages. In Section 4 we study their expressive power, and in Section 5 we investigate their complexity. In Section 6 we introduce new data tests that go beyond those present in XPath and study their expressiveness and complexity. Concluding remarks are in Section 7. Due to space limitations, most proofs are only sketched here.

2. PRELIMINARIES

We first describe graph databases. We assume a model in which edges are labeled by letters from a finite alphabet Σ and nodes can contain data values from a countably infinite set $\mathcal{D}$ (for instance, attributes of people in a social network). For simplicity of notation only, we assume a single data value per node, as is often done in modeling XML with data trees [42]. This is not a restriction at all, as different attributes can be added by adding extra outgoing edges with specified labels (again, in the same way as data trees model XML documents).

DEFINITION 2.1 (DATA GRAPHS). *A* data graph *(over Σ and $\mathcal{D}$) is a triple $G = \langle V, E, \rho \rangle$, where:*

- *V is a finite set of nodes;*
- *$E \subseteq V \times \Sigma \times V$ is a set of labeled edges; and*
- *$\rho : V \to \mathcal{D}$ is a function that assigns a data value to each node in V.*

When we deal with purely navigational queries, i.e., those not taking into account data values, we refer to graph $\langle V, E \rangle$, omitting the function ρ. We write E_a for the set of a-labeled edges, i.e., $E_a = \{(v, v') \mid (v, a, v') \in E\}$.

A path from node v_1 to v_n in a graph is a sequence

$$\pi = v_1 a_1 v_2 a_2 v_3 \ldots v_{n-1} a_{n-1} v_n \qquad (1)$$

such that each (v_i, a_i, v_{i+1}), for $i < n$, is an edge in E. We use the notation $\lambda(\pi)$ to denote the label of path π, i.e., the word $a_1 \ldots a_{n-1} \in \Sigma^*$.

Navigation languages for graph databases.

Most navigational formalisms for querying graph databases are based on *regular path queries*, or *RPQs* [19], and their extensions. An RPQ is an expression of the form $x \xrightarrow{L} y$, where L is a regular language over Σ (typically represented by a regular expression or an NFA). Given a Σ-labeled graph $G = \langle V, E \rangle$, the answer to an RPQ as above is the set of pairs of nodes (v, v') such that there is a path π from v to v' with $\lambda(\pi) \in L$.

Conjunctive RPQs, or *CRPQs* [18] are the closure of RPQs under conjunction and existential quantification. Formally,

they are expressions of the form

$$\varphi(\bar{x}) \;=\; \exists \bar{y} \bigwedge_{i=1}^{n} (z_i \xrightarrow{L_i} u_i), \qquad (2)$$

where all variables z_i, u_i come from $\bar{x}, \bar{y}$. The semantics naturally extends the semantics of RPQs: $\varphi(\bar{a})$ is true in G iff there is a tuple $\bar{b}$ of nodes such that, for every $i \leq n$, every pair v_i, v_i' interpreting z_i and u_i is in the answer to the RPQ $z_i \xrightarrow{L_i} u_i$.

These have been further extended, for instance, to 2CRPQs that allow navigation in both directions (i.e., the edges can be traversed both forwards and backwards [11]), U2CRPQs that allow unions, or to *extended* CRPQs, in which paths witnessing the RPQs $z_i \xrightarrow{L_i} u_i$ can be named and compared for relationships between them, defined as regular or even rational relations [7, 6].

3. XPATH-LIKE LANGUAGES FOR GRAPHS

We follow the standard way of defining XPath fragments [10, 12, 22, 25, 36, 16] and introduce some variants of *graph XPath*, or GXPath, to be interpreted over graph databases. As usual, XPath formulae are divided into *path formulae*, producing sets of pairs of nodes, and *node tests*, producing sets of nodes. Path formulae will be denoted by letters from the beginning of the Greek alphabet ($\alpha, \beta, \ldots$) and node formulae by letters from the end of the Greek alphabet ($\varphi, \psi, \ldots$).

Since we deal with data values, we need to define *data tests* permitted in node formulae. There will be two kinds of them.

1. <u>Constant tests</u>: For each data value $c \in \mathcal{D}$, we have two tests $=c$ and $\neq c$. The intended meaning is to test if the data value in the current node equals to, or differs from, constant c.

   The fragment of GXPath that uses constant tests will be denoted by GXPath(c).

2. <u>Equality/inequality tests</u>: These are typical XPath (in)equality tests of the form $\langle \alpha = \beta \rangle$ and $\langle \alpha \neq \beta \rangle$, where α and β are path expressions. The intended meaning is to check for the existence of two paths, one satisfying α and the other satisfying β, which end with equal (resp., different) data values.

   The appropriate fragment will be denoted by GXPath(eq). If we have both constant tests and equality tests, we denote resulting fragments by GXPath(c, eq).

Next we define expressions of GXPath. As already mentioned in the introduction, we look at *core* and *regular* versions of XPath. They both have node and path expressions. Node expressions in all fragments are given by the grammar:

$$\varphi, \psi := \top \mid \mathsf{test} \mid \neg \varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \langle \alpha \rangle$$

where test is one of the permitted data tests defined earlier, and α is a path expression.

The path formulae of the two flavors of GXPath are given below. In both cases a ranges over Σ.

Path expressions of *Regular graph XPath*, denoted by GXPath$_{\text{reg}}$, are given by:

$$\alpha, \beta := \varepsilon \mid \_ \mid a \mid a^- \mid [\varphi] \mid \alpha \cdot \beta \mid \alpha \cup \beta \mid \overline{\alpha} \mid \alpha^*$$

Path expressions of *Core graph XPath* denoted by GXPath$_{\text{core}}$ are given by:

$$\alpha, \beta := \varepsilon \mid \_ \mid a \mid a^- \mid a^* \mid a^{-*} \mid [\varphi] \mid \alpha \cdot \beta \mid \alpha \cup \beta \mid \overline{\alpha}$$

We call this fragment "Core graph XPath", since it is natural to view edge labels (and their reverse) in data graphs as the single-step axes of the usual XPath on trees. For instance, a and a^- could be similar to "child" and "parent". Thus, in our core fragment, we only allow transitive closure over navigational single-step axes, as is done in Core XPath on trees. Note that we did not explicitly define the counterpart of node label tests in GXPath node expressions to avoid notational clutter, but all the results remain true if we add them.

Finally, we consider another feature that was recently proposed in the context of navigational languages on graphs (such as in SPARQL 1.1 [29]), namely counters. The idea is to extend all grammars defining path formulae with new path expressions

$$\alpha^{n,m}$$

for $n, m \in \mathbb{N}$ and $n < m$. Informally, this means that we have a path that consists of some k chunks, each satisfying α, with $n \leq k \leq m$.

When counting is present in the language, we denote it by #GXPath, e.g., #GXPath$_{\text{core}}$.

Given these path and node formulae, we can combine GXPath$_{\text{core}}$ and GXPath$_{\text{reg}}$ with different flavors of node tests or counting, starting with purely navigational fragments (neither c nor eq tests are allowed) and up to having both c and eq tests. For example, #GXPath$_{\text{reg}}$(c, eq) is defined by mutual recursion as follows:

$$\alpha, \beta := \varepsilon \mid \_ \mid a \mid a^- \mid [\varphi] \mid \alpha \cdot \beta \mid \alpha \cup \beta \mid \overline{\alpha} \mid \alpha^* \mid \alpha^{n,m}$$
$$\varphi, \psi := \neg \varphi \mid \varphi \wedge \psi \mid \langle \alpha \rangle \mid =c \mid \neq c \mid \langle \alpha = \beta \rangle \mid \langle \alpha \neq \beta \rangle$$

with c ranging over constants.

We define the semantics with respect to a data graph $G = \langle V, E, \rho \rangle$. The semantics $[\![\alpha]\!]^G$ of a path expression α is a set of pairs of vertices and the semantics of a node test, $[\![\varphi]\!]^G$, is a set of vertices. The definitions are given in Figure 1. In that definition, by R^k we mean the k-fold composition of a binary relation R, i.e., $R \circ R \circ \ldots \circ R$, with R occurring k times.

Remark. Note that each path expression α can be transformed into a node test by the means of $\langle \alpha \rangle$ operator. In particular, we can test if a node has a b-successor by writing, for instance, $\langle b \rangle$. To reduce the clutter when using such tests in path expressions, we shall often omit the $\langle \rangle$ braces and write e.g. $a[b]$ instead of $a[\langle b \rangle]$.

<table>
<tr><td colspan="3" align="center">Path expressions</td></tr>
</table>

$$\llbracket \varepsilon \rrbracket^G = \{(v,v) \mid v \in V\}$$
$$\llbracket \_ \rrbracket^G = \{(v,v') \mid (v,a,v') \in E \text{ for some } a\}$$
$$\llbracket a \rrbracket^G = \{(v,v') \mid (v,a,v') \in E\}$$
$$\llbracket a^- \rrbracket^G = \{(v,v') \mid (v',a,v) \in E\}$$
$$\llbracket \alpha^* \rrbracket^G = \text{the reflexive transitive closure of } \llbracket \alpha \rrbracket^G$$
$$\llbracket \alpha \cdot \beta \rrbracket^G = \llbracket \alpha \rrbracket^G \circ \llbracket \beta \rrbracket^G$$
$$\llbracket \alpha \cup \beta \rrbracket^G = \llbracket \alpha \rrbracket^G \cup \llbracket \beta \rrbracket^G$$
$$\llbracket \overline{\alpha} \rrbracket^G = V \times V - \llbracket \alpha \rrbracket^G$$
$$\llbracket [\varphi] \rrbracket^G = \{(v,v) \in G \mid v \in \llbracket \varphi \rrbracket^G\}$$
$$\llbracket \alpha^{n,m} \rrbracket^G = \bigcup_{k=n}^{m} (\llbracket \alpha \rrbracket^G)^k$$

<div align="center">Node tests</div>

$$\llbracket \langle \alpha \rangle \rrbracket^G = \pi_1(\llbracket \alpha \rrbracket^G) = \{v \mid \exists v' \ (v,v') \in \llbracket \alpha \rrbracket^G\}$$
$$\llbracket \neg \varphi \rrbracket^G = V - \llbracket \varphi \rrbracket^G$$
$$\llbracket \varphi \wedge \psi \rrbracket^G = \llbracket \varphi \rrbracket^G \cap \llbracket \psi \rrbracket^G$$
$$\llbracket \varphi \vee \psi \rrbracket^G = \llbracket \varphi \rrbracket^G \cup \llbracket \psi \rrbracket^G$$
$$\llbracket =c \rrbracket^G = \{v \in V \mid \rho(v) = c\}$$
$$\llbracket \neq c \rrbracket^G = \{v \in V \mid \rho(v) \neq c\}$$
$$\llbracket \langle \alpha = \beta \rangle \rrbracket^G = \{v \in V \mid \exists v', v'' \ (v,v') \in \llbracket \alpha \rrbracket^G, \ (v,v'') \in \llbracket \beta \rrbracket^G, \rho(v) = \rho(v')\}$$
$$\llbracket \langle \alpha \neq \beta \rangle \rrbracket^G = \{v \in V \mid \exists v', v'' \ (v,v') \in \llbracket \alpha \rrbracket^G, \ (v,v'') \in \llbracket \beta \rrbracket^G, \rho(v) \neq \rho(v')\}$$

Figure 1: Semantics of Graph XPath expressions with respect to $G = \langle V, E, \rho \rangle$

Basic expressiveness results.

Some expressions are readily definable with those we have. For instance, Boolean operations $\alpha \cap \beta$ and $\alpha - \beta$ with the natural semantics are definable. Indeed, $\alpha - \beta$ is definable as $\overline{\overline{\alpha} \cup \beta}$, and intersection is definable with union and complement. So when necessary, we shall use intersection and set difference in path expressions.

Counting expressions $\alpha^{n,m}$ are definable too: they abbreviate $\alpha \cdots \alpha \cdot (\alpha \cup \varepsilon) \cdots (\alpha \cup \varepsilon)$, where we have a concatenation of n times α and $m - n$ times $(\alpha \cup \varepsilon)$. Thus, adding counters does not influence expressivity of any of the fragments, since we always allow concatenation and union. However, counting expressions can be exponentially more succinct than their smallest equivalent regular expressions (independent of whether n and m are represented in binary or in unary) [33]. We will exhibit a query evaluation algorithm with polynomial-time complexity even for such expressions with counters represented in binary.

We next give three examples of GXPath expressions to illustrate what sort of queries one can ask using these languages.

1. The expression $(a[b])^*$ will simply give us all pairs (x,y) of nodes that are connected by a path of the following form:
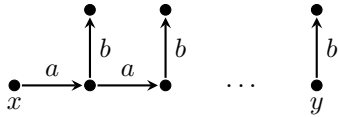
That is, x and y are connected by an a^* labelled path such that each node on the path also has an outgoing b-labelled edge. (Nodes that are different in the picture do not have to be different in the graph.)

2. The expression $\langle aa^* \neq bc^- \rangle$ will give us all nodes x such that there are nodes y and z, reachable by aa^* and bc^- respectively, with different data values. For example in the graph given in the following image the nodes x_1 and x_2 will be selected by our query, while x_3 will not.
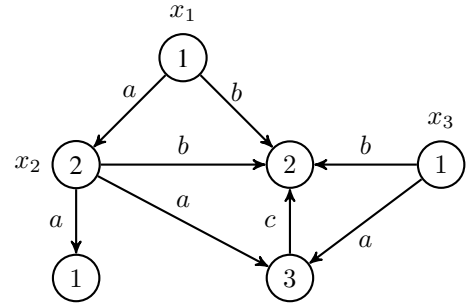
3. The expression $\langle (a[=5] \cdot (a[=5])^*) \cap \varepsilon \rangle$ will extract all the nodes x such that there is a cycle starting at x in which each edge is labelled by a and each node has the data value 5. In particular the node x will have data value 5. Note that this example illustrates how we can define loops using GXPath.

As another observation on the expressiveness of the language, note that we can define a test $\langle \alpha = c \rangle$, with the semantics $\{v \mid \exists v' \ (v,v') \in \llbracket \alpha \rrbracket^G \text{ and } \rho(v') = c\}$, by using the expression $\langle \alpha[=c] \rangle$.

Another thing worth noting is that node expressions can be defined in terms of path operators. For example $\varphi \wedge \psi$ is defined by the expression $\langle \varepsilon[\varphi] \cdot \varepsilon[\psi] \rangle$, while $\neg \varphi$ is defined by $\langle \varepsilon[\overline{\varphi}] \rangle$.

Complement and positive fragments.

In standard XPath dialects on trees, complementation operators are not included and one usually shows that languages are closed under negation. This is no longer true for arbitrary graphs, due to the following.

PROPOSITION 3.1. *Neither $\overline{\alpha}$ nor $\alpha - \beta$ is definable in GXPath$_{reg}$ without complement on path expressions.*

The proof is an immediate consequence of the following observation. Given a data graph G, let $V_1, \ldots, V_m$ be sets of nodes of its (maximal) connected components (with respect to the edge relation $\bigcup_{a \in \Sigma} E_a$). Then a simple induction on the structure of the expressions of GXPath$_{reg}$ without complement on path expressions shows that for each expression α, we have $[\![\alpha]\!]^G \subseteq \bigcup_{i \leq m} V_i \times V_i$. However, both path complementation $\overline{\alpha}$ and path difference $\alpha - \beta$ violate this property.

In what follows, we consider fragments of our languages that restrict complementation and negation. There are two kinds of them, the first corresponding to the well-studied notion of positive XPath.

- The *positive fragments* are obtained by removing $\neg\varphi$ and $\overline{\alpha}$ from the definitions of node and path formulae. We use the superscript pos to denote them, i.e., we write GXPath$_{core}^{pos}$ and GXPath$_{reg}^{pos}$.
- The *path-positive fragments* are obtained by removing $\overline{\alpha}$ from the definitions of path formulae, but keeping $\neg\varphi$ in the definitions of node formulae. We use the superscript path-pos to denote them, i.e., we write GXPath$_{core}^{path-pos}$ and GXPath$_{reg}^{path-pos}$.

4. EXPRESSIVE POWER OF LANGUAGES

The goal of this section is to analyze the expressiveness of various XPath-like formalisms for graph databases that we introduced. We start with navigational features and then analyze languages that handle data comparisons. Additional analysis of expressiveness is given in Section 6 where we extend languages with more powerful comparisons of data.

4.1 Expressiveness of navigational languages

We provide three types of analysis of expressiveness of navigational features of dialects of graph XPath:

- We compare them with FO, fragments and extensions. The core language will capture FO^3. This is similar to a capture result for trees [36]; the main difference is that on graphs, unlike on trees, this falls short of full FO. We also provide a counterpart of this result for GXPath$_{reg}$, adding the transitive closure operator.
- We compare them with commonly used graph languages, such as nested regular expressions [38] and CRPQs (and relatives). We identify a fragment of GXPath$_{reg}$ that captures nested regular expressions, but show that generally, XPath flavors are incompatible with CRPQs and their extensions.

- We look at the analog of conditional XPath [36] which captures FO over trees and show that, in contrast, over graph databases, it can express queries that are not FO-definable.

4.1.1 Comparisons with FO and relatives

To compare expressiveness of GXPath fragments with first-order logic, we need to explain how to represent graph databases as FO structures. Since all the formalisms can express reachability queries (at least with respect to a single label), we view graphs as FO structures

$$G = \langle V, (E_a, E_{a^*})_{a \in \Sigma} \rangle$$

where $E_a = \{(v, v') \mid (v, a, v') \in E\}$ and E_{a^*} is its reflexive-transitive closure.

Recall that FO^k stands for the k-variable fragment of FO, i.e., the set of all FO formulae that use variables from a fixed set $x_1, \ldots, x_k$. As we mentioned, on trees, the core fragment of XPath 2.0 was shown to capture FO^3. We now prove that the same remains true without restriction to trees.

THEOREM 4.1. *GXPath$_{core}$ = FO^3 with respect to both path queries and node tests.*

Proof sketch. The idea behind this proof is to use the characterization of FO^3 in terms of relation algebras [43]. These are algebras of binary relations over some domain V and are closed under composition of relations, complementation (over V^2), union and reverse relation. We start with a base set $A_1, \ldots, A_n$ of binary relations over V and interpret the operations in a standard way. We shall be using relation algebras over V whose base relations are those in the FO vocabulary, i.e., the E_as and the E_{a^*}s.

We can then show that, for each FO^3 formula $F(x, y)$ with free variables x and y (in the vocabulary of the E_as and E_{a^*}s), there is a path expression α_F of GXPath$_{core}$ such that $(a, b) \in [\![\alpha_F]\!]^G$ iff $G \models F(a, b)$. This is done by going through relation algebra (with base binary relations E_a, E_{a^*}, for $a \in \Sigma$) and showing that such an algebra is equivalent with path expressions of GXPath$_{core}$ over the class of graphs where E_{a^*} is the reflexive transitive closure of E_a. The other direction is actually much simpler as we only have to give a translation from GXPath$_{core}$ formulas to FO^3 formulas.

Note that going through relation algebra works only for formulas with two free variables. To show that every formula $F(x)$ with a single free variable is equivalent to a GXPath$_{core}$ node test we can define $F'(x, y)$ as $(x = y) \wedge F(x)$, and find an equivalent path expression $\alpha_{F'}$. Then we simply set $\varphi_F := \langle \alpha_{F'} \rangle$ to get the node expression equivalent to F. □

Not all results about the expressiveness of XPath on trees extend to graphs. For instance, on trees, the regular fragment with no negation on paths (i.e., the path-positive fragment) can express all of FO [36]. This fails over graphs: GXPath$_{reg}$ fails to express even all of FO^2 when restricted to its path-positive fragment (i.e, the fragment that still permits unary negation).

PROPOSITION 4.2. *There exists a binary FO^2 query that is not definable in $\mathsf{GXPath}^{path\text{-}pos}_{reg}$.*

Proof sketch. The idea is to observe that path-positive fragments of GXPath cannot define the universal binary relation on an input graph. The query not definable in $\mathsf{GXPath}^{path\text{-}pos}_{reg}$ is then the one saying that there are at least two nodes in a given graph. $\square$

We now move to GXPath_{reg} and relate it to a fragment of FO^*, the parameter-free fragment of the transitive-closure logic. The language of FO^* extends the one of FO with a transitive closure operator that can be applied to formulas with precisely two free variables. That is, for any FO formula $F(x, y)$, the formula $F^*(x, y)$ is also an FO^* formula. The semantics is the reflexive-transitive closure of the semantics of F. That is, $G \models F^*(a, b)$ iff $a = b$ or there is a sequence of nodes $a = v_0, v_1, \ldots, v_n = b$ for $n > 0$ such that $G \models F(v_i, v_{i+1})$ whenever $0 \le i < n$.

By $(\mathrm{FO}^*)^k$ we mean the k-variable fragment of FO^*. Note that when we deal with FO^* and $(\mathrm{FO}^*)^k$, we can view graphs as structures of the vocabulary $(E_a)_{a \in \Sigma}$, since all the E_{a^*}s are definable, and there is no reason to include them in the language explicitly.

Over trees, regular XPath is known to be equal to $(\mathrm{FO}^*)^3$ [14]. The next theorem shows that over graphs, these logics coincide as well.

THEOREM 4.3. *$\mathsf{GXPath}_{reg} = (\mathrm{FO}^*)^3$.*

Proof sketch. The containment of GXPath_{reg} in $(\mathrm{FO}^*)^3$ is done by a routine translation.

To show the converse, we use techniques similar to those in the proof of Theorem 4.1: we extend $(\mathrm{FO}^*)^3$ and relation algebra equivalence to state that relation algebra with the transitive closure operator has equal expressive power to $(\mathrm{FO}^*)^3$ over the class of all labeled graphs. For this one can simply check that the inductive proof from [4] can be extended by adding two extra inductive clauses. Namely, when going from relation algebra to FO^3 we simply state that expressions of the form R^* are equivalent to $F_R^*(x, y)$, where F_R is the formula equivalent to R. In the other direction we simply state that $F^*(x, y)$ is equivalent to $(R_F(x, y))^*$. Here by $R_F(x, y)$ we denote the expression equivalent to $F(x, y)$, when the variables are used in that particular order. After that one verifies that the correctness proof of [4] applies. $\square$

What about the relative expressive power of GXPath_{core} and GXPath_{reg}? For positive fragments, known results on trees (see [16]) imply the following.

COROLLARY 4.4. *$\mathsf{GXPath}^{pos}_{core} \subsetneq \mathsf{GXPath}^{pos}_{reg}$.*

We shall later see that the strict separation applies to full languages. This is not completely straightforward even though GXPath_{core} is equivalent to a fragment of FO, since the latter uses the vocabulary with transitive closures. This makes

it harder to apply standard techniques, such as locality, directly. We shall see how to establish separation when we deal with conditional XPath in Section 4.1.3.

4.1.2 Comparisons with path queries

Our next goal is to compare the expressiveness of XPath formalisms for graphs with that of other established formalisms. We start with *nested regular expressions*, which have been proposed as a navigational mechanism of SPARQL for querying RDF data [38]. After that we look at traditional languages such as RPQs, CRPQs, and relatives.

Nested regular expressions. These expressions, abbreviated as NRE, over a finite alphabet Σ extend ordinary regular expressions with the nesting operator and inverses [38]. Formally they are defined as follows:

$$n := \varepsilon \ | \ a \ | \ a^- \ | \ n \cdot n \ | \ n^* \ | \ n + n \ | \ [n]$$

where a ranges over Σ.

Intuitively NREs define binary relations consisting of pairs of nodes connected by a path specified by the NRE. When interpreted on a data graph G the relations are defined inductively as follows:

$$
\begin{aligned}
\llbracket \varepsilon \rrbracket^G &= \{(v, v) \mid v \in V\} \\
\llbracket a \rrbracket^G &= \{(v, v') \mid (v, a, v') \in E\} \\
\llbracket a^- \rrbracket^G &= \{(v, v') \mid (v', a, v) \in E\} \\
\llbracket n \cdot n' \rrbracket^G &= \llbracket n \rrbracket^G \circ \llbracket n' \rrbracket^G \\
\llbracket n + n' \rrbracket^G &= \llbracket n \rrbracket^G \cup \llbracket n' \rrbracket^G \\
\llbracket n^* \rrbracket^G &= \text{the reflexive transitive closure of } \llbracket n \rrbracket^G \\
\llbracket [n] \rrbracket^G &= \{(v, v) \mid \exists v' \text{ such that } (v, v') \in \llbracket n \rrbracket^G\}.
\end{aligned}
$$

As expected, GXPath_{reg} is strictly more expressive than NREs. However, we show that NREs do capture the positive fragment of GXPath_{reg}.

THEOREM 4.5. *$\mathsf{GXPath}^{pos}_{reg} = \mathsf{NRE} \subsetneq \mathsf{GXPath}^{path\text{-}pos}_{reg}$.*

Proof sketch. To show that NREs are strictly weaker, consider a path formula $\alpha = a[\neg\langle b \rangle]$. Then one can prove by induction that over the graph below, every NRE has a nonempty answer.
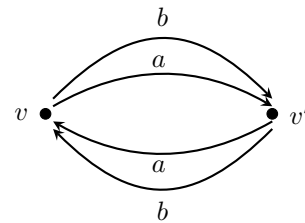
This of course gives us the desired result, since $\llbracket \alpha \rrbracket^G = \emptyset$. $\square$

Comparison with CRPQs. We will show that XPath-like formalisms are incomparable with CRPQs and similar queries in terms of their navigational expressiveness. The simple restriction, $\mathsf{GXPath}^{pos}_{reg}$, is not subsumed by

CRPQs. In fact it is not even subsumed by unions of two-way CRPQs (which allow navigation in both ways). On the other hand, CRPQs are not subsumed by the strongest of our navigational languages, GXPath$_{reg}$.

THEOREM 4.6. *CRPQs and GXPath fragments are incomparable:*

- *GXPath$_{reg}^{pos}$ $\not\subseteq$ CRPQ (even stronger, there are GXPath$_{reg}^{pos}$ queries not definable by U2CRPQs);*

- *CRPQ $\not\subseteq$ GXPath$_{reg}$.*

Proof sketch. The first item follows from Theorem 4.5 and the fact that U2CRPQs cannot simulate certain NRE queries [8]. To see the second item, we first show that for every GXPath$_{reg}$ expression e there exists an $\mathcal{L}_{\infty\omega}^3$ formula F_e equivalent to it. Recall that by $\mathcal{L}_{\infty\omega}^3$ we mean the infinitary first-order logic that uses only three variables (i.e. extension of FO^3 with infinite conjunctions and disjunctions). This is done by a standard induction on GXPath$_{reg}$ expressions with variable reuse, see, e.g., [31].

Consider now two graphs, K_3 and K_4, with all edges labeled a. It is well know that they cannot be distinguished by $\mathcal{L}_{\infty\omega}^3$ since the duplicator has a winning strategy in the 3-pebble game on them. However, they can be distinguished by a CRPQ $\varphi(x, y)$ that states the existence of nodes z and u and all possible a-edges between x, y, z, u except self-loops. □

On the other hand, the positive fragment of GXPath$_{core}$ can be captured by unions of two-way CRPQs.

PROPOSITION 4.7. *GXPath$_{core}^{pos}$ $\subsetneq$ U2CRPQ.*

4.1.3 *Conditional* GXPath

It was shown in [36] that to capture FO over XML trees, one can use *conditional* XPath, which essentially adds the temporal *until operator*. That is, it expands the core-XPath's a^* with $(a[\varphi])^*$, which checks that the test $[\varphi]$ is true on an a-labeled path. Formally, its path formulae are given by:

$$\alpha, \beta \quad := \quad \varepsilon \mid \_ \mid a \mid a^- \mid a^* \mid a^{-^*} \mid (a[\varphi])^* \mid (a^-[\varphi])^* \mid$$
$$[\varphi] \mid \alpha \cdot \beta \mid \alpha \cup \beta \mid \overline{\alpha}$$

We refer to this language as GXPath$_{cond}$. We now show that the FO capture result fails rather dramatically over graphs: there are even positive GXPath$_{cond}$ queries not expressible in FO.

THEOREM 4.8. *There is a GXPath$_{cond}^{pos}$ query not expressible in* FO.

Note that the standard inexpressibility tools for FO, such as locality, cannot be applied straightforwardly since the vocabulary of graphs already contains all the transitive closures E_{a^*}; in fact this means that in GXPath$_{cond}^{pos}$ the query asking for transitive closures of base relations is trivially definable, even though it is not definable in FO over the E_as. So the way around this is to combine locality with the composition method: we use locality to establish a winning strategy

for the duplicator in a game that does not involve transitive closures, and then use composition to extend the winning strategy to handle transitive closures.

Proof sketch. We consider graphs over alphabet $\{a, b, s, t\}$, with s, t being labels used to mark nodes between which we shall test reachability. We are interested in the following property P: from a node with an incoming s-edge to the node with an outgoing t-edge, there is an a-path such that each node on the path has a b-successor.

We then exhibit two families of graphs, G_1^m and G_2^m, over the usual graph vocabulary, so that for each m we have $G_m^1 \equiv_m G_m^2$ (i.e., the duplicator has a winning strategy in the usual m-round game), and all the G_m^1s have property P, and G_m^2s do not. To do so, we use Hanf-locality and the property that Hanf-locality, with a sufficiently long radius ($\geq 3^m$), implies the $\equiv_m$ relation (cf. [31]).

We then extend graphs G_1^m and G_2^m with a single new node v that has a-edges to and from every other node. Then a composition argument shows that the m-round winning strategy of the duplicator extends to the m-round winning strategy in the game on extended graphs viewed as structures in the vocabulary that includes all the transitive closures. Basically, the duplicator follows the original game on G_1^m and G_2^m, and if the spoiler plays the new added node in one graph, then the duplicator responds with the added node in the other graph. What makes it work as a strategy in the extended vocabulary is that in the expanded graphs the interpretations of the transitive closures are easy: E_{a^*} is the total relation and no other relation E_x has a path of length 2. This implies that the property P is not FO-definable even in the expanded vocabulary.

But note that P is definable in GXPath$_{cond}$ by $s(a[b])^*[t]$. So assuming GXPath$_{cond} \subseteq$ FO we get a contradiction. □

We can now fulfill our promise and establish separation between GXPath$_{core}$ and GXPath$_{reg}$. Since GXPath$_{core} \subseteq$ FO and we just saw a conditional (and thus regular) GXPath query not expressible in FO, we have:

COROLLARY 4.9. *GXPath$_{core}$ $\subsetneq$ GXPath$_{reg}$.*

4.2 Expressiveness of data languages

By coupling the basic navigational languages – GXPath$_{core}$ and GXPath$_{reg}$ – with various possibilities of data tests, such as no data tests, constant tests, equality tests, or both, we obtain eight languages, ranging from GXPath$_{core}$ to GXPath$_{reg}$(c, eq). Recall that adding counting does not affect expressiveness, only the complexity of query evaluation.
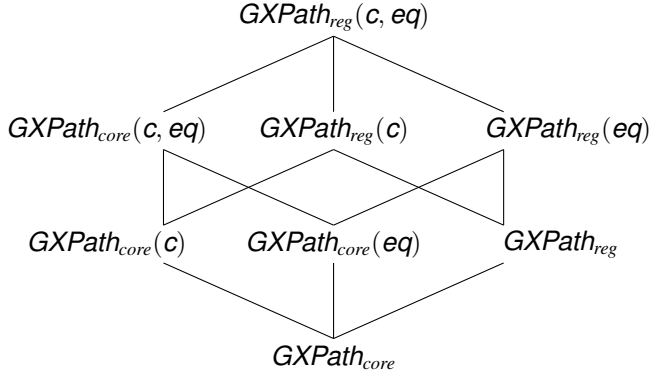
The question is then, how do these fragments compare to each other? Basically, each fragment is of the form $L(t)$, where L is the navigational language and t is the set of allowed data tests. We next show that there are no unexpected interdependencies: that is, $L(t)$ is strictly less expressive than $L'(t')$ iff:

1. $L \subseteq L'$ (in other words, $L = L'$ or $L =$ GXPath$_{core}$ and $L' =$ GXPath$_{reg}$),

2. $t \subseteq t'$; and

3. at least one of the above inclusions is strict (i.e., either $L = \mathsf{GXPath}_{\text{core}}$ and $L = \mathsf{GXPath}_{\text{reg}}$, or $t \subsetneq t'$).

Formally, we state the following.

THEOREM 4.10. *The relative expressive power of graph XPath languages with data comparisons is as shown below:*

$$GXPath_{reg}(c, eq)$$

$$GXPath_{core}(c, eq) \qquad GXPath_{reg}(c) \qquad GXPath_{reg}(eq)$$

$$GXPath_{core}(c) \qquad GXPath_{core}(eq) \qquad GXPath_{reg}$$
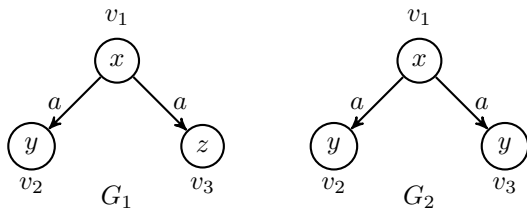
$$GXPath_{core}$$

Here a line upwards means that the fragment is strictly contained in the other, while the lack of the line means that the fragments are incomparable.

Proof. The result follows from Corollary 4.9 (for navigational fragments) and the following two observations which show that c tests and eq tests are not mutually definable. Namely, take an alphabet Σ containing letter a. Let c be a fixed data value. Then:

- There is no $\mathsf{GXPath}_{\text{reg}}(\mathsf{eq})$ expression equivalent to the $\mathsf{GXPath}_{\text{core}}(\mathsf{c})$ query $q_{\mathsf{c}} := (= c)$.
- There is no $\mathsf{GXPath}_{\text{reg}}(\mathsf{c})$ expression equivalent to the $\mathsf{GXPath}_{\text{core}}(\mathsf{eq})$ query $q_{\mathsf{eq}} := \langle a = a \rangle$.

For the first item, simply take two single-node data graphs G_1 and G_2, with G_1's single node holding value c, and G_2 holding a different value c'. Hence, $[\![q_{\mathsf{c}}]\!]^{G_1}$ selects the only node of G_1, while $[\![q_{\mathsf{c}}]\!]^{G_2} = \emptyset$. However, a straightforward induction on the structure of expressions shows that for every $\mathsf{GXPath}_{\text{reg}}(\mathsf{eq})$ query e we have $[\![e]\!]^{G_1} = [\![e]\!]^{G_2}$.

For the second item assume that there is an $\mathsf{GXPath}_{\text{reg}}(\mathsf{c})$ expression ex equivalent to q_{eq}. Take any three pairwise distinct data values x, y, z that are different from all the constants appearing in ex and let G_1 and G_2 be as below:

$$
\begin{array}{ccc}
& v_1 & \\
& x & \\
a \swarrow & & \searrow a \\
y & & z \\
v_2 & & v_3 \\
& G_1 &
\end{array}
\qquad
\begin{array}{ccc}
& v_1 & \\
& x & \\
a \swarrow & & \searrow a \\
y & & y \\
v_2 & & v_3 \\
& G_2 &
\end{array}
$$

One can show by straightforward induction on $\mathsf{GXPath}_{\text{reg}}(\mathsf{c})$ expressions e that use only constants

appearing in ex that $[\![e]\!]^{G_1} = [\![e]\!]^{G_2}$. Thus, q_{eq} cannot be a $\mathsf{GXPath}_{\text{reg}}(\mathsf{c})$ expression, since $[\![q_{\mathsf{eq}}]\!]^{G_1} \neq [\![q_{\mathsf{eq}}]\!]^{G_2}$.

Note that this also shows that $\mathsf{GXPath}_{\text{core}} \subsetneq \mathsf{GXPath}_{\text{core}}(\mathsf{c})$ and $\mathsf{GXPath}_{\text{core}} \subsetneq \mathsf{GXPath}_{\text{core}}(\mathsf{eq})$. □

We saw that for navigational features, core graph XPath captures FO^3. The question is whether this continues to be so in the presence of data tests. First, we need to explain how to describe data graphs as FO-structures to talk about FO with data tests.

Following the standard approach for data words and data trees [42], we do so by adding a binary predicate for testing if two nodes hold the same data value. That is, a data graph is then viewed as a structure $G = \langle V, (E_a, E_{a^*})_{a \in \Sigma}, \sim \rangle$ where $v \sim v'$ iff $\rho(v) = \rho(v')$. To be clear that we deal with FO over that vocabulary, we shall write $\mathrm{FO}(\sim)$. If we want to talk about constant data tests (i.e., $= c$), we assume that the FO vocabulary contains constants. In that case we shall refer to $\mathrm{FO}(\mathsf{c}, \sim)$.

It turns out that the equivalence with FO^3 breaks when we add tests on data that have been seen so far.

THEOREM 4.11.
- $\mathit{GXPath}_{\mathit{core}}(\mathit{eq}) \subsetneq \mathrm{FO}^3(\sim)$;
- $\mathit{GXPath}_{\mathit{core}}(\mathit{c}, \mathit{eq}) \subsetneq \mathrm{FO}^3(\mathit{c}, \sim)$.

Proof sketch. The first containment uses the translation into FO^3 shown in the proof of Theorem 4.1. For the new data operators, we use the following. If $e = \langle \alpha = \beta \rangle$ then

$$F_e(x) \equiv \exists y, z (y \sim z \wedge F_\alpha(x, y) \wedge \exists y (z = y \wedge F_\beta(x, y)))$$

and likewise for the inequality comparison.

Translation of constants is self-evident.

To prove strictness we show that the FO^3 query $F(x, y) \equiv x \sim y$ is not definable in $\mathsf{GXPath}_{\text{reg}}(\mathsf{c}, \mathsf{eq})$. Note that F defines the set of all pairs of nodes carrying the same data value. The proof of this is implicit in the proof of Proposition 6.1. □

Thus, the standard XPath data tests are insufficient for capturing FO^3 over data graphs. Nonetheless, there is a simple extension of data tests that lets core graph XPath capture $\mathrm{FO}^3(\sim)$; we shall present it in Section 6.

5. COMPLEXITY OF QUERY EVALUATION

In this section we investigate the complexity of querying graph databases using variants of GXPath. We consider two problems. One is QUERY EVALUATION, which is essentially model checking: we have a graph database, a query (i.e., a path expression), and a pair of nodes, and we want to check if the pair of nodes is in the query result. That is, we deal with the following decision problem.

PROBLEM:	QUERY EVALUATION
INPUT:	A graph $G = (V, E)$,
	a path expression α, nodes $v, v' \in V$.
QUESTION:	Is $(v, v') \in [\![\alpha]\!]^G$?

The second version we consider is QUERY COMPUTATION, which actually computes the result of a query and outputs it. Normally, when one deals with path expressions, one fixes a context node v and looks for all nodes v' such that (v, v') satisfies the expression. We deal with a slightly more general version here, where the context node need not be single.

PROBLEM:	QUERY COMPUTATION
INPUT:	A graph $G = (V, E)$, a path expression α,
	and a set of nodes $S \subseteq V$.
OUTPUT:	All $v' \in V$ such that there exists a $v \in S$
	with $(v, v') \in [\![\alpha]\!]^G$.

Note that in both problems we deal with *combined complexity*, as the query is a part of the input.

For measuring complexity, we let $|G|$ denote the size of the graph and $|\alpha|$ (resp., $|\varphi|$) denote the size of the path expression α (resp., node expression φ).

The main result of this section is that the combined complexity remains in polynomial time for all fragments we defined in Section 3. Not only that, but the exponents are low, ranging from linear to cubic. Notice that for navigational fragments, the low (and even linear) complexity should not come as a surprise. We noticed that $\mathsf{GXPath}_{reg}^{path\text{-}pos}$ is essentially PDL, for which global model checking is known to have linear-time complexity [3, 17]. Also, polynomial-time combined complexity results are known for pure navigational GXPath_{reg} from the PDL perspective as well [30].

Our main contribution is thus to establish the low combined complexity bounds for fragments that handle two new features we added on top of navigational languages: *data value comparisons* and *counters*. The former does increase expressiveness; the latter, as already remarked, does not, but it can make expressions exponentially more succinct. Thus, work is needed to keep combined complexity polynomial when counters are added.

For obtaining the linear-time complexities in this section, we assume a total order on the labels of edges. We assume that graphs are represented as adjacency lists such that we can obtain, for a given node v, the outgoing edges or the incoming edges, sorted in increasing order of labels, in constant time. (We note that the linear-time algorithm from [3] for PDL model checking also assumes that adjacency lists are sorted.) As we said, the following result is immediate from PDL model checking techniques:

FACT 5.1. *Both* QUERY EVALUATION *and* QUERY COMPUTATION *problems for* $\mathsf{GXPath}_{reg}^{path\text{-}pos}$ *can be solved in linear time, i.e.,* $O(|\alpha| \cdot |G|)$.

PROOF. Since global model checking for PDL is in linear time [3, 17], it is immediate that QUERY EVALUATION is in time $O(|\alpha| \cdot |G|)$. From this, the same bound for QUERY COMPUTATION can also be derived. Given a query α and a set S, we can mark the nodes in S with a special predicate that occurs nowhere in α. We can then modify query α and use the algorithm for global model checking for PDL to obtain the required output of QUERY COMPUTATION. $\square$

The main upper bound in this section shows that combined complexity of both problems is polynomial for the most expressive language we have: regular graph XPath with counting, constant tests, and equality tests.

THEOREM 5.2. *Both* QUERY EVALUATION *and* QUERY COMPUTATION *problems for* $\#\mathsf{GXPath}_{reg}(c, eq)$ *can be solved in polynomial time, specifically, i.e.,* $O(|\alpha| \cdot |G|^3)$.

Proof sketch. We can do a dynamic programming algorithm that considers the parse tree of α in a bottom-up fashion and computes, for every subexpression β of α, the table $[\![\beta]\!]^G$. For each subexpression φ, we store in a bit-vector the nodes of G that match φ.

When we see a subexpression of the form $\beta^{n,m}$, we compute the adjacency matrix representation of $[\![\beta]\!]^G$ and compute $[\![\beta^{n,m}]\!]^G$ by using fast squaring methods. This approach is similar to the one used for regular expressions with counters (cf. [33]). $\square$

The algorithm for Theorem 5.2 is based on connectivity matrix multiplications for dealing with the counters. If the queries do not have counters, we can evaluate them more efficiently because we can avoid matrix multiplications. This allows us to drop data complexity from cubic to quadratic.

THEOREM 5.3. *Both* QUERY EVALUATION *and* QUERY COMPUTATION *problems for* $\mathsf{GXPath}_{reg}(c, eq)$ *can be solved in polynomial time, specifically, i.e.,* $O(|\alpha| \cdot |G|^2)$.

The algorithm in the proof for Theorem 5.3 actually computes the entire relation $[\![\alpha]\!]^G$. Since the size of this relation can be quadratic in the worst-case, a significantly faster algorithm for computing $[\![\alpha]\!]^G$ cannot be expected.

A result related to Theorem 5.3 is shown in [30]. Here the combined complexity is investigated for an extension of PDL which includes the complement operator and context-free path expressions, and a model-checking algorithm based on adjacency matrix operation is presented. The algorithm from [30] uses time $O(|\varphi|^2 \cdot |V|^5)$, where $|V|$ is the number of nodes in G.

Finally, we give a fragment that takes data value comparisons into consideration and still permits linear-time query evaluation and computation. We do so by noticing that using only constant tests (as opposed to equality tests) does not introduce extra complexity for evaluation of path-positive GXPath. Since the latter is essentially PDL, we get an algorithm that is linear in both the query and the data. That is, we have the following.

THEOREM 5.4. *Both* QUERY EVALUATION *and* QUERY COMPUTATION *problems for* $\mathsf{GXPath}^{path\text{-}pos}_{reg}(\mathbf{c})$ *can be solved in linear time, i.e.,* $O(|\alpha| \cdot |G|)$.

6. BEYOND XPATH TESTS

We have seen previously that the equivalence of core graph XPath and FO^3 established for pure navigational fragments does not extend to data tests, as $\mathsf{GXPath}_{core}(\mathsf{eq}) \subsetneq \mathrm{FO}^3(\sim)$. This naturally leads to a question: what can be added to data tests to capture the full power of FO^3?

The answer to this is to add a different type of *equality comparison*, not present in XPath but used previously to enhance the power of RPQs and CRPQs [32]. These are defined by adding two expressions to the grammar for α: one is $\alpha_=$, the other is $\alpha_{\neq}$. Semantics, over data graphs, is

$$\begin{aligned}
[\![\alpha_=]\!]^G &= \{(v,v') \in [\![\alpha]\!]^G \mid \rho(v) = \rho(v')\} \\
[\![\alpha_{\neq}]\!]^G &= \{(v,v') \in [\![\alpha]\!]^G \mid \rho(v) \neq \rho(v')\}
\end{aligned}$$

In other words, we test whether data values at the beginning and at the end of a path are the same, or different. Such an extension is denoted by $\sim$, i.e. we talk about languages $\mathsf{GXPath}(\sim)$ (with the usual sub- and superscripts).

The first observation is that these tests indeed add to the expressiveness of the languages.

PROPOSITION 6.1. *The path query* $a_=$*, for* $a \in \Sigma$*, is not definable in* $\mathsf{GXPath}_{reg}(\mathbf{c}, \mathsf{eq})$.

Note that this query, $a_=$, is definable on trees by the $\mathsf{GXPath}_{core}(\mathsf{eq})$ query $[\langle \varepsilon = a \rangle] \cdot a \cdot [\langle \varepsilon = a^- \rangle]$. This is because the parent of a given node is unique. However, on graphs this is not always the case, and thus new equality tests add power.

The proof (omitted here) shows that, even though $\mathsf{GXPath}_{reg}(\mathbf{c}, \mathsf{eq})$ can test if a node has an a-successor with the same data value by the means of expression $\langle \varepsilon = a \rangle$, which will return the set $\{v \in V \mid \exists v' \in V \mid (v,v') \in [\![a_=]\!]^G\}$, it has no means of retrieving that specific successor.

With the extra power given to us by the equality tests, we can capture FO^3 over data graphs.

THEOREM 6.2. $\mathsf{GXPath}_{core}(\sim) = \mathrm{FO}^3(\sim)$.

Proof sketch. We follow the technique of the proof of Theorem 4.1. All of the translations used there still apply. The proof that relation algebra is contained in the language $\mathsf{GXPath}_{core}(\sim)$ is the same as without data values. We only have to add conversion of the new symbol $\sim$: if $R = \sim$, then $e = \varepsilon \cup (\bar{\varepsilon})_=$.

For the other direction we have to show how to translate new path expressions $\alpha_=$ and $\alpha_{\neq}$ into $\mathrm{FO}^3(\sim)$. This is done as follows: if $e = \alpha_=$ then $F_e(x,y) \equiv F_\alpha(x,y) \wedge x \sim y$ and likewise for inequality. The equivalences easily follow. Now

the theorem follows from the equivalence of relation algebra and FO^3 [43]. $\square$

From this we know that every $\mathsf{GXPath}_{core}(\mathsf{eq})$ query can be expressed in $\mathsf{GXPath}_{core}(\sim)$. We can also perform such a transformation explicitly. It is not difficult to see that every node expression of the form $\langle \alpha = \beta \rangle$ is equivalent to $\mathsf{GXPath}_{core}(\sim)$ expression $\langle \alpha \cdot (\alpha^- \cdot \beta)_= \cdot \beta^- \cap \varepsilon \rangle$, and similarly for $\neq$.

Query evaluation in the extended fragment.

We have added new features to the language. They increased its expressiveness, so the question arises whether the complexity of query evaluation (and query computation) suffers from this addition. The good news is that even with this addition, we retain polynomial combined complexity of both query evaluation and query computation, still with a low-degree polynomial.

THEOREM 6.3. *Both* QUERY EVALUATION *and* QUERY COMPUTATION *problems for* $\mathsf{GXPath}_{reg}(\mathbf{c}, \mathsf{eq}, \sim)$ *can be solved in polynomial time, specifically in time* $O(|\alpha| \cdot |G|^2)$.

Proof sketch. The algorithm is similar to the one of Theorem 5.3, which computes a table $[\![\beta]\!]^G$ for every path subexpression β. We now alter it so that, every time when we meet a subexpression of the form $\beta_=$, we walk through the table for $[\![\beta]\!]^G$ and remove all tuples with different data values, thus obtaining a table for $[\![\beta_=]\!]^G$. We proceed similarly for $[\![\beta_{\neq}]\!]^G$. $\square$

7. CONCLUSIONS

After conducting the study of XPath-like languages over graph databases, our main conclusion is that they were perhaps unfairly overlooked as potential query languages. And indeed, some practitioners do attempt to use them [13, 26] probably following the intuition that such languages should behave well in the graph database context.

Our goal was to provide a theoretical foundation for this intuition. We did it by studying the expressiveness and complexity of various XPath formalisms over graph databases. Our languages included purely navigational features, to which one could add any of several features for handling data stored in graph databases. The navigational features corresponded to core and regular flavors of XPath, while data tests included different comparisons of data values: either XPath-like based on standard node tests, or more advanced, doing tests involving endpoints of paths.

We showed that the languages correspond to typical yardsticks for relational and XML languages. The navigational power of the core language captures FO^3, and this continues to be so with the most advanced data test (although typical XPath data tests fall short of the full FO power). For fragments based on regular XPath, we had correspondence with a three-variable fragment of the transitive closure logic.

We showed that the languages fit in well with some of the features proposed for SPARQL, the language for query-

ing RDF data. Specifically, we showed that one of the navigational fragments we deal with corresponds precisely to a popular navigational formalism for SPARQL, namely nested regular expressions, and we handled numerical tests on the numbers of repetitions of path, also proposed in the SPARQL standardization effort.

We then demonstrated that all the languages behave very well computationally: the combined complexity of all of them is polynomial. In fact, it is always a low-degree polynomial. The worst case complexity is cubic, and it applies only in the case of numerical tests, which are known to make expressions exponentially more succinct. In other cases, it is quadratic, and can even drop to linear for some fragments.

Given these desirable properties of XPath-like languages for graph databases, we believe this theoretical study justifies an attempt to experiment with those in practical scenarios, perhaps in less ad hoc way than was done in [13, 26]. As for a concrete language to adapt, we believe languages based on $\text{GXPath}_{\text{reg}}^{\text{path-pos}}$ hold a lot of promise. The navigational part is essentially PDL and therefore firmly rooted in logic. Complexity-wise it behaves like XPath: it can be evaluated in linear time and its satisfiability problem is EXPTIME-complete [28]. Perhaps more important, its proximity to XPath makes it very accessible to practitioners who are familiar with XML technology. In addition, as our study shows, it can be augmented with all other features (equality and constant tests, counting, even equality tests going beyond XPath) without incurring significant complexity costs.

8. REFERENCES

[1] S. Abiteboul, V. Vianu. Regular path queries with constraints. *J. Comput. Syst. Sci.* 58(3):428–452 (1999).
[2] N. Alechina, S. Demri, M. de Rijke. A modal perspective on path constraints. *J. Log. Comput.* 13(6): 939–956 (2003).
[3] N. Alechina and N. Immerman. Reachability logic: An efficient fragment of transitive closure logic. *L. J. of the IGPL* 8(3):325–337 (2000).
[4] H. Andréka, I. Németi, and I. Sain. Algebraic logic. In *Handbook of Philosophical Logic*, vol. 2, Springer 2001.
[5] R. Angles, C. Gutiérrez. Survey of graph database models. *ACM Comput. Surv.* 40(1): (2008).
[6] P. Barceló, D. Figueira, L. Libkin. Graph logics with rational relations and the generalized intersection problem. In *LICS 2012*.
[7] P. Barceló, C. Hurtado, L. Libkin, P. Wood. Expressive languages for path queries over graph-structured data. In *PODS*, pages 3–14, 2010.
[8] P. Barceló, J. Pérez and J. L. Reutter. Relative expressiveness of nested regular expressions, In *AMW 2012*.
[9] M. Bojanczyk and S. Lasota. An extension of data automata that captures XPath. In *LICS 2010*, pages 243–252.
[10] M. Bojanczyk, P. Parys. XPath evaluation in linear time. *J. ACM* 58(4): 17 (2011).
[11] D. Calvanese, G. de Giacomo, M. Lenzerini, M. Y. Vardi. Containment of conjunctive regular path queries with inverse. In *KR'00*, pages 176–185.
[12] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. An automata-theoretic approach to regular XPath. In *DBPL 2009*, 18-35.

[13] S. Cassidy. Generalizing XPath for directed graphs. In *Extreme Markup Languages*, 2003.
[14] B. ten Cate. The expressivity of XPath with transitive closure. In *PODS 2006*, pages 328–337.
[15] B. ten Cate, C. Lutz. The complexity of query containment in expressive fragments of XPath 2.0. *J. ACM* 56(6): (2009).
[16] B. ten Cate, M. Marx. Navigational XPath: calculus and algebra. *SIGMOD Record* 36(2): 19–26 (2007).
[17] R. Cleaveland and B. Steffen. A Linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2(2):121–147, 1993.
[18] M. P. Consens, A. O. Mendelzon. GraphLog: a visual formalism for real life recursion. In *PODS'90*, pages 404–416.
[19] I. Cruz, A. Mendelzon, P. Wood. A graphical query language supporting recursion. In *SIGMOD'87*, pages 323–330.
[20] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu. Graph pattern matching: from intractable to polynomial time. *PVLDB* 3(1): 264–275 (2010).
[21] W. Fan. Graph pattern matching revised for social network analysis. In *ICDT 2012*, pages 8–21.
[22] D. Figueira. Reasoning on words and trees with data. PhD thesis, 2010.
[23] G. H. L. Fletcher, M. Gyssens, D. Leinders, J. Van den Bussche, D. Van Gucht, S. Vansummeren, Y. Wu. Relative expressive power of navigational querying on graphs. *ICDT 2011*, 197-207
[24] G. H. L. Fletcher, M. Gyssens, D. Leinders, J. Van den Bussche, D. Van Gucht, S. Vansummeren, Y. Wu. The impact of transitive closure on the boolean expressiveness of navigational query languages on graphs. *FoIKS 2012*, 124-143
[25] G. Gottlob, C. Koch, R. Pichler. Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst.* 30(2):444–491 (2005).
[26] Gremlin Language. https://github.com/tinkerpop/gremlin/wiki
[27] C. Gutierrez, C. Hurtado, A. Mendelzon. Foundations of semantic Web databases. *J. Comput. Syst. Sci.* 77(3): 520–541 (2011).
[28] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
[29] S. Harris and A. Seaborne. SPARQL 1.1 Query Language. W3C Working Draft 5 January 2012. http://www.w3.org/TR/2012/WD-sparql11-query-20120105/
[30] M. Lange. Model checking propositional dynamic logic with all extras. *J. Applied Logic* 4(1), 39–49, 2006.
[31] L. Libkin. *Elements of Finite Model Theory*. Springer, 2004.
[32] L. Libkin and D. Vrgoč. Regular path queries on graphs with data. *ICDT 2012*, pages 74–85.
[33] K. Losemann and W. Martens. The complexity of evaluating path expressions in SPARQL. *PODS 2012*, pages 101–112.
[34] C. Lutz, D. Walther. PDL with negation of atomic programs. *In IJCAR 2004*, pages 259–273.
[35] M. Marx. XPath and modal logics of finite DAGs. In *TABLEAUX 2003*, pages 150–164.
[36] M. Marx. Conditional XPath. *ACM Trans. Database Syst.* 30(4): 929–959 (2005).
[37] J. Pérez, M. Arenas, C. Gutierrez. Semantics and complexity of SPARQL. *ACM TODS* 34(3): 2009.
[38] J. Pérez, M. Arenas, C. Gutierrez. nSPARQL: A navigational language for RDF. In J. Web Sem. 8(4): 255–270 (2010).
[39] R. Ronen and O. Shmueli. SoQL: a language for querying and creating data in social networks. In *ICDE 2009*.
[40] M. San Martín, C. Gutierrez. Representing, querying and transforming social networks with RDF/SPARQL. In *ESWC 2009*, pages 293–307.
[41] Th. Schwentick. XPath query containment. *SIGMOD Record* 33(1): 101–109 (2004).
[42] L. Segoufin. Static analysis of XML processing with data values. *SIGMOD Record* 36(1): 31–38 (2007).
[43] A. Tarski and S. Givant. *A Formalization of Set Theory Without Variables*. AMS, 1987.
[44] XML Path Language (XPath). www.w3.org/TR/xpath.