Querying Property Graphs with XPath

Marko Junkkari(✉), Sami-Santeri Svensk, and Jyrki Nummenmaa

Tampere University, Tampere, Finland
{marko.junkkari,jyrki.nummenmaa}@tuni.fi,
samisanterisvensk@gmail.com

Abstract. XPath has been established as the de facto standard for searching data items from hierarchical XML structures. Due to its popularity and compact path expressions, XPath has also been recognized as a query language candidate for graph databases where the structure does not follow a hierarchical order. Graph databases are based on graph theory and the data are organized accordingly. Among different types of graphs, property graphs have gained special interest because they allow data associated with edges as well as vertices, reflecting that edges represent relationships and relationships are generally allowed to have properties, just like entities. Earlier proposals to apply XPath to graph databases do not allow manipulation of the properties of edges in a property graph. The present study focuses on this issue. We show how XPath can be applied to full-scale property graphs. This requires a novel mapping of XPath primitives to the primitives of property graphs. Based on this mapping, we define graph-based semantics for XPath by regular path queries, an established logical approach for querying vertices and edges.

Keywords: Property graph · Graph database · XPath · Regular path query

1 Introduction and Related Work

Graph databases [2, 6, 17, 26] are in growing demand for analyzing linked data in various domains [3, 26]. In graph databases, the data is organized using graph structures, emerging from heavily studied graph theory [5], giving graph databases a strong theoretical foundation [20]. There is, however, no common data model for all graph databases [34]. Common features exist, though, like for instance index-free adjacency [34]. Retrieving data from graph databases can be performed using the graph operations defined in the graph theory [2, 12]. Like other NoSQL databases, graph databases store semi-structured data containing the schema within the data [13].

There is a consensus on modeling entities as vertices and their relationships as edges. A special type of graph data model, called property graphs [33], has emerged both in the theoretical context, e.g. [18, 26], and in practical implementations [29]. A property graph relies on vertices and edges, both of which can be labeled. The most popular graph database Neo4J [29] is based on the property graph model. The graph database community has recently seen the rise of commercial query languages such as Cypher [28], PGQL [32], SQL/PGQ [16] and Gremlin [7]. In 2024, Graph Query Language (GQL) became the standard query language for property graphs [21].

The history of graph query languages starts from the 1980s, when the language G was proposed in [15] to query edge-labeled graph with regular expressions. Mendelzon and others [27] created the semantics based on language G to find a simple path between two vertices. This approach describing paths is nowadays known in literature as regular path queries (RPQ). RPQ:s have influenced the research from then on and have had a huge impact for query language design, and the navigation can be found from many of those languages. Since then, RPQs have been extended with various extensions to enhance expressivity. Such extensions include RPQ with inverse [14], conjunctive RPQ [14], extended conjunctive RPQ [9] and RPQ with data tests [25]. Path queries bring an expressive way to query databases [4]. In our opinion, RPQ forms a similar basis to graph query languages as relational algebra or calculus formed to the relational query languages.

Pattern matching, finding nodes connected by paths, and aggregating the results are important features for graph query languages [4, 38]. These features have also been identified to form the core of XPath [35], a thoroughly researched e.g. [10, 11] query language to address parts in XML [36] documents. Even though XPath is designed to operate on tree-like-structured XML documents, Buneman [13] sees these structures essentially as rooted graphs. Angles and others [2] describe XML as restricted type of graph and see essential theoretical basis, graph theory influencing both graph databases and XML documents. Despite the fact that XPath has been developed originally for addressing parts of XML documents, Libkin and others [24] see XPath as probably overlooked as a candidate language for graph databases, as its goal seems very similar to querying graph databases.

Libkin and others [24, 25] have noted the proximity of XPath to first order logic or modal logic and van Rest and others [32] compare XPath to Tarski's algebra, which has similarities to the basics of many graph query languages. XPath and RPQ have been compared in the context of graph databases [8, 19, 23, 24]. Oltenau and others [30] describe the relationship between XPath and RPQ as an abstraction of the navigational features of XPath, where support for XPath axes child, descendant, parent and ancestor is provided. Despite the similarities, XPath cannot fully be subsumed by RPQ or its various extensions [24]. In some scenarios, XPath goes beyond the path queries by defining patterns that cannot be captured by paths [25]. Libkin and others [25] study the capabilities of potential languages, including XPath, to query graph databases combining topology and data. According to them, XPath succeeds by describing the properties of paths and patterns, considering both their purely navigational aspects and the data contained in the database. Their analysis is on a theoretical level, and they do not provide an implementation or an approach for implementation. Barceló [8] has mentioned XPath's branching operator providing good expressive power in graphs. In general, there is a wide consensus that XPath is a promising querying approach to graph data model.

Libkin and others [24] have created semantics for XPath in graphs and studied the expressiveness and complexity of various XPath formalisms called GXPath. Their semantics were expressed with respect to a graph structure called data graph. Research of XPath in graphs has focused on simple graph structures like data graphs [24, 25], where the data is contained in nodes as single values. There has not yet been a direct link between the formalism of XPath and the increasingly common property graph data model, even though both are data models designed for semi-structured data and have similar capabilities to store data into properties of the elements building the structure of the model. We define how the primitives of XPath are mapped to the primitives of the property graph. The path steps of XPath are not labeled, and they do not have properties while the edges of a property graph are labeled and may possess properties. We manipulate both vertices and edges as XPath nodes. We match selected fragments of XPath to the features of property graphs and define compilation from XPath to RPQ by attribute grammars [22].

The rest of the paper is organized as follows. Section 2 introduces the graph structure and notations for regular path queries. In Sect. 3, we define the mapping between XPath primitives and graph primitives. In this context, we give an informal introduction for using XPath in property graphs. In Sect. 4, we define PRQ based semantics and use them to evaluate an example query. We give our conclusions in Sect. 5.

2 Property Graphs and Regular Path Queries

Property graphs are directed multigraphs, where vertices represent entities and edges relationships between them [5, 12]. Figure 1 illustrates this conceptualization without properties. Customer, Order, Product, Supplier and Category are entities, modeled as vertices. Arrows represent relationships, modeled as directed edges.

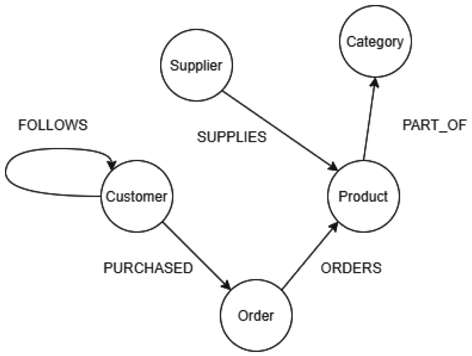

Fig. 1. Example graph

We follow the definitions of Angles and others [5], however, we do not allow multi-labeling of vertices and edges nor multivalued properties. Let L be a set of labels, P a set of properties and PV a set of property values. Then, the property graph can formally be defined as the tuple $G = (V, E, \rho, \lambda, \sigma)$, where.

1. V is a finite set of the vertices.
2. E is a finite set of edges.
3. ρ: $E \rightarrow (V \times V)$ assigns each edge of E to a pair of nodes in V.
4. λ: $(V \cup E) \rightarrow L$ labels edges and vertices with the set L.
5. σ: $(V \cup E)(V \cup E) \times P \rightarrow PV$ assigns a value to a property of a vertex or edge.

Following this notation, the example graph used through the paper can be expressed as follows:

- V = {v1, v2, v3, v4, v5, v6}
- E = {e1, e2, e3, e4, e5}
- P = {type, name, quantity, country}
- L = {Customer, Supplier, Category, Product, Order, SUPPLIES, PURCHASED, ORDERS, PART_OF, FOLLOWS}
- $\lambda = \{\langle v1, Customer\rangle, \langle v2, Customer\rangle, \langle v3, Order\rangle, \langle v4, Supplier\rangle, \langle v5, \Pr oduct\rangle, \langle v6, Category\rangle, \langle e1, FOLLOWS\rangle, \langle e2, PURCHASED\rangle, \langle e3, ORDERS\rangle, \langle e4, SUPPLIES\rangle, \langle e5, PART\_OF\rangle\}$
- ρ = {⟨e1, ⟨v1, v2⟩⟩, ⟨e2, ⟨v1, v3⟩⟩, ⟨e3, ⟨v3, v5⟩⟩, ⟨e5, ⟨v5, v6⟩⟩, ⟨e5, ⟨v4, v6⟩⟩}
- σ = {⟨⟨v6, type⟩, junk⟩, ⟨⟨v3, name⟩, Smith⟩, ⟨⟨e3, quantity⟩, 10⟩, ⟨⟨v4, Country⟩, UF⟩, ⟨⟨v5, Name⟩, toy⟩}

We can allow L to assign similar labels to vertices and edges. Our example does not do that, so we can use labels when referring to vertices and edges.

Regular path queries are based on regular expressions determining one or several paths in a graph. Formally, a regular path query can be expressed by a triple (X, RE, Y), where X and Y are variables that refer to the end and start vertices of the underlying path expression, and RE is a regular expression over the vocabulary of edge labels. Operations, such as repetitions (+, *), can be used and a regular expression may contain a complex clause with patterns including various edges. However, for the purpose of the present study, complex regular expressions are not needed. RE can be represented in the form -label- > where the label may involve the + postfix for denoting one or several occurrences of the edge. In RE, it is also possible to express that no label is specified. This is denoted by -·- >. Thus, any path can be denoted by -· + – >. Property value restrictions for an edge can be represented within parentheses. For example, -ORDERS(Quantity = 10)- > refers to the order edges that have the quantity property with value 10.

The labels of vertices can be restricted with additional facts of the form (X, is, label), where X is a variable. For example, (X –· + – > Y) Λ (X, is, Customer) Λ (Y, is, Order) refers to any path from a customer vertex to an order vertex. Property value restrictions of vertices can be represented by additional facts. For example, (Z, is, Category) Λ (Z, has, type = 'junk') means that Category has an attribute with name 'type' and value 'junk'.

3 XPath for Graph Database

XPath consists of path steps Axis::node_test[predicate] separated by slashes (/). An axis determines the relationships between a context node and the connected processing nodes. The most common axes are self, child, parent and descendant-or-self. Element label and attribute are the most common node types. A predicate can determine conditions for both nodes and paths. Using predicates, a search tree structure can be represented in a serialized form, where each branch can have individual predicates. For the most common axes, abbreviators are established: the child axis is the default value, and it is typically not expressed, double dash (//) denotes descendants, dot (.) corresponds to the self axis, two dots (..) determines the parent axis and @ is refers to an attribute.

The simplest way to apply XPath to graphs would be by mapping nodes to vertices and child relationships to directed edges. For example, the XPath expression Customer/Product would be mapped to the edge Customer $\rightarrow$ Product. The expression Customer//Category would correspond to any directed path from Customer to Category. Following this approach, attributes can be used in referring to the properties of vertices. For example, Category[@type ='junk'] could denote the type property having value 'junk' in a Category node.

The problem of this trivial approach is that it is not suitable for property graphs where edges are labelled and may have properties. The child relationships in XML are not labelled, and they have no properties. Therefore, we propose that also edges are manipulated as XPath nodes. In other words, we map a path in a graph to an XPath path. A path in a graph is started from a vertex, and every other member is an edge, and every other a vertex [15]. For example, in the expression a/b/c/d/e, the nodes a, c and e refer to vertices and b and d refer to edges. In other words, a/b returns the edge b and a/b/c returns the vertex c. It is possible to refer the properties of vertices and edges as our example in Sect. 4 will demonstrate.

Like in XPath, the asterisk refers to an unlabeled node, for both vertices and edges. For example, Order/*/Product means a step from an order to a product via any edge whereas Order/ORDERS/* means any successor of an order through an ORDERS edge. The descendant notation can also be used in the context of graphs. The path Customer//Category determines all paths from a customer to a category. It is worth noting that the descendant relationship may refer to either vertices or edges if the asterisk is used. For example, in the expression Order//* the asterisks may refer to ORDERS, Product, PART_OF or Category. As such this kind of expression is hardly useful but this allows powerful expressions in queries containing uncertain aspects. For example, it is possible to determine a path from a node to another and give a value restriction to an attribute in an edge or a vertex in the path. In the path Customer//*[unitPrice > 10 000]//Category, the unit price can be in any node between Customer and Category.

The branches can be expressed between square brackets in serialized expressions. For example, the path Order[ORDERS/Product[@name = 'toy']] refers to an order that contains a toy. The double dot denotes a parent relationship. In the context of graphs, we interpret this as traversal to the inverse direction in a directed edge. For example, the path Product/../Supplier denotes the inverse direction from a product to a supplier. The parent notation can also be used in serialized expressions. For example in the path Product[../Supplier[@Country ='UK']/*/Category, the fragment../Supplier[@Country ='UK'] means that a product vertex must have an inverse path to a supplier whose country is UK.

Above, the abbreviations of XPath are used for navigation. Basically, navigation is based on axes that determine the displacement from a context node to its relatives. The relationship between XPath axes and their interpretation in graph databases is as follows, with the arrow from axis to interpretation: *child* $\rightarrow$ *immediate successor; parent* $\rightarrow$ *immediate predecessor; descendant* $\rightarrow$ *successor; ancestor* $\rightarrow$ *predecessor; self* $\rightarrow$ *self; descendant-or-self* $\rightarrow$ *successor or self; ancestor-or-self* $\rightarrow$ *predecessor or self;*

attribute → *property*. The axes *following, preceding, following-sibling,* and *preceding-sibling* do not have an interpretation in a graph.

4 Regular Path Query Based Semantics

Attribute grammars (AGs), widely used in compilers [1, 37], are used to define both the syntax and semantics of a formal language [22, 31]. We introduce only such notational conventions for attribute grammars that are applied in this study.

Let AG = (G, A, R), be a triple where G is a context free grammar, A is a finite set of attributes and R is a finite set of semantic rules associated with the attributes. A context free grammar G = (NT, T, P, D) defines a syntax for a formal language. In G, NT is a finite set of non-terminals and T is a finite set of terminals such that $NT \cap T = \emptyset$. Elements in NT and T are called grammar symbols. P is a finite set of productions. Each production is of the form $X \rightarrow \alpha$, where $X \in N$ and $\alpha \in (NT \cup N)^*$. P may contain alternative productions e.g. $X \rightarrow \alpha 1$ and $X \rightarrow \alpha 2$ so that $\alpha 1 \neq \alpha 2$. D ($\in$ N) is a start symbol. Each attribute ($\in$ A) is associated with one or several non-terminals and if $X \in N$, then the attribute set of X is denoted by A(X). A(X) is partitioned into two exclusive sets: inherited attributes I(X) and synthesized attributes S(X) so that $I(X) \cap S(X) = \emptyset$ and $I(X) \cup S(X) = A(X)$. Usually there are two ways to denote the selection of an attribute a of symbol X. One follows a record style, X.a; the other one follows a functional style a(X). In this paper we adopt the second one. Each semantic rule ($\in$ R) is associated with a production p ($\in$ P) to define the evaluation of a synthesized attribute of the symbol in the left hand side of p, or the evaluation of an inherited attribute of a symbol on the right-hand side of p.

We define the context free grammar G_{XRPQ} as the tuple (NT, T, P, Q) where NT = {Q, P, E, V, VN, EN}, T = {/, //,.., *, [,], @, =, ≠, <, >} ∪ E_names ∪ V_names ∪ A_names ∪ A_values. Q is the starting symbol, and P is the set of productions represented in the second column of Table 1. E_names and V_names are the names of edges and vertices, respectively. A_names and A_values are the sets of property names and values. We define the attribute grammar AG_{XRPQ} to be the triple (G_{XRPQ}, A, R) where A = {*v*, *result*, *ret*, *name*, *var*, *first_var*} and R is the semantic rules associated with the P. The attributes in A have the following intention:

- *v* is an inherited attribute for the variable associated with a vertex or an edge.
- *result* is a synthesized attribute that describes the final conjunctive query of the regular path queries.
- *res* is a synthesized attribute that contains a set of regular path queries associated within parsing the XPath query.
- *var* is a synthesized attribute that contains the variable associated with a vertex.
- *first_var* is a synthesized attribute that contains the variable associated with the first vertex of a sub-path.
- The attributes are associated with the grammar symbols as follows:
- I(V) = I(V) = {*v*}
- S(Q) = {*result*}
- S(P) = {ret, first_var}
- S(E) = S(EN) = {*name*}

- S(VN) = {*name*, *var*}
- S(V) = {*var*, *ret*}

The rules of R are represented in the third and fourth columns of Table 1. In the third column, the function *new*() generates a new variable.

Table 1. An attribute grammar for compiling XPath to regular query queries

Id	Production	Inherited	Synthesized
1	Q → P		$\boldsymbol{result(Q) = \bigwedge_{x \epsilon ret(P)} x}$
2	P1 → V / E / P2	*v*(V) = new()	*ret*(P1) = {(*var*(V) -*name*(E)-> *first_var*(P2))} ∪ *ret*(V) ∪ *ret*(P2) *first_var*(P1) = v(V)
3	P1 → V /../ P2	*v*(V) = new()	*ret*(P1) = {*first_var*(P2) -•-> (*var*(V))} ∪ *ret*(V) ∪ *ret*(P2) *first_var*(P1) = *v*(V)
4	P → V	*v*(V) = new()	*ret*(P)=*ret*(V) *first_var*(V)=*v*(V)
5	P1 → V//P2	*v*(V) = new()	*ret*(P1) = {(*v*(V) -.+-> *first_var*(P1))} ∪ *ret*(V) ∪ *ret*(P2) *first_var*(P1) = *v*(V)
6	V → VN[E/P]	*v*(NV) = *v*(V)	*ret*(V) = {(*v*(VN), **is**, *name*(VN)), (*v*(VN) -*name*(E)-> *first_var*(P))} ∪ *ret*(P) *var*(V) = *v*(VN)
7	V → VN	*v*(NV) = *v*(V)	*ret*(V) = {(*v*(VN), **is**, *name*(VN))}, if *name*(VN) ≠ 'null' ∅ , otherwise *var*(V) = *v*(VN)
8	E → EN		*name*(E) = *name*(EN)
9	V1 → V2[@name o d]	*v*(V2) = *v*(V1)	*ret*(V1) = *ret*(V2) ∪ {(*v*(V2), has, name), (name o d)}, where name ∈ names and d ∈ values and o ∈ {=, ≠, <, >} *var*(V1) = *v*(V2)
10	E1 → E2[@name o d]		*name*(E1) = *name*(E2) ⊕ (name o d) , where name ∈ names and d ∈ values and o ∈ {=, ≠, <, >}
11	VN → name		*name*(VN) = name, where name ∈ V-names
12	EN → name		*name*(EN) = name, where name ∈ E-names
13	VN → *		*name*(VN) = 'null'
14	EN → *		*name*(EN) = •

Related to the graph of Fig. 1, an example evaluation for an XPath query

```
Customer/*/Order/ORDERS[@quantity=10]/Product[PART_OF/
Category[@type = 'junk']]/../Supplier
```

is given in Fig. 2. The query associates customers and suppliers in a case where a customer has bought 10 pieces of products whose category are 'junk', and the products have the same supplier. Variables are labeled X1, X2, X3, etc. following their creating order.

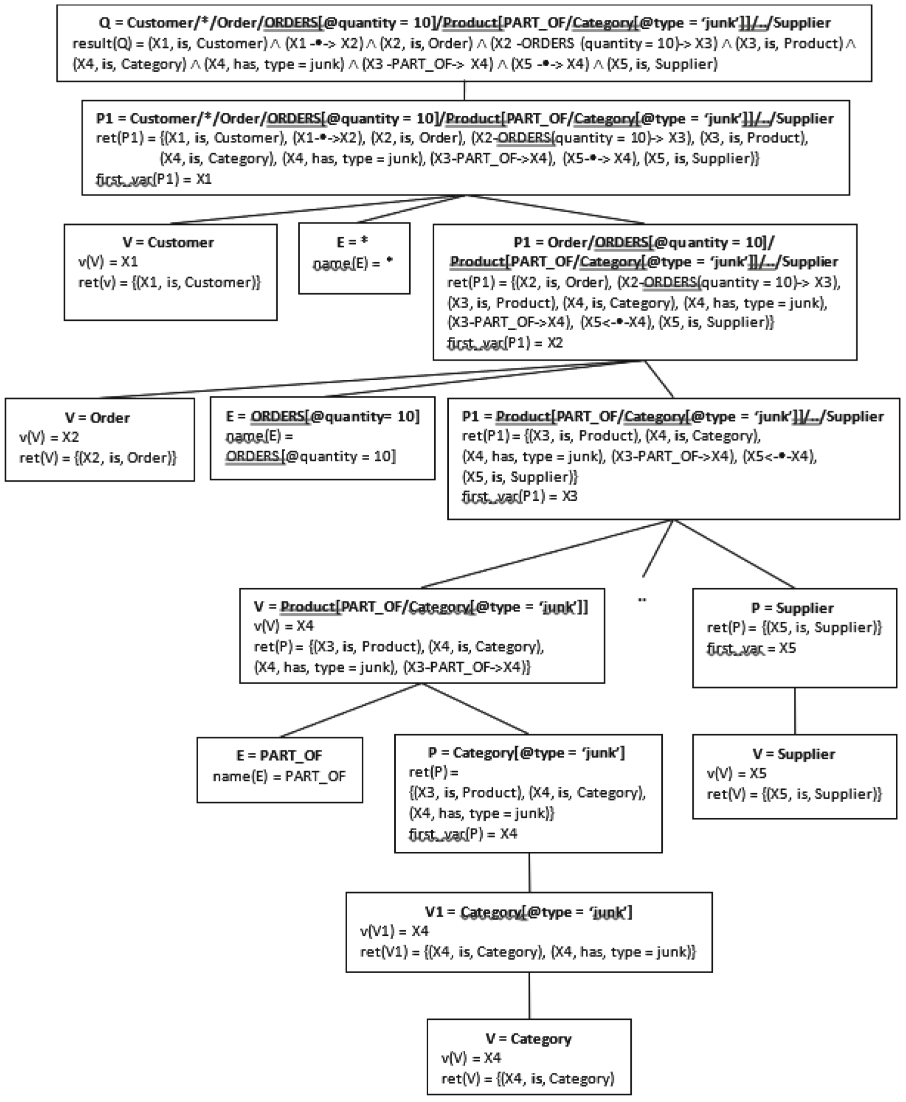
Q = Customer/*/Order/ORDERS[@quantity = 10]/Product[PART_OF/Category[@type = 'junk']]/../Supplier
result(Q) = (X1, is, Customer) ∧ (X1 -•-> X2) ∧ (X2, is, Order) ∧ (X2 -ORDERS (quantity = 10)-> X3) ∧ (X3, is, Product) ∧ (X4, is, Category) ∧ (X4, has, type = junk) ∧ (X3 -PART_OF-> X4) ∧ (X5 -•-> X4) ∧ (X5, is, Supplier)
P1 = Customer/*/Order/ORDERS[@quantity = 10]/Product[PART_OF/Category[@type = 'junk']]/../Supplier
ret(P1) = {(X1, is, Customer), (X1-•->X2), (X2, is, Order), (X2-ORDERS(quantity = 10)-> X3), (X3, is, Product), (X4, is, Category), (X4, has, type = junk), (X3-PART_OF->X4), (X5-•-> X4), (X5, is, Supplier)}
first_var(P1) = X1
V = Customer
v(V) = X1
ret(v) = {(X1, is, Customer)}
E = *
name(E) = *
P1 = Order/ORDERS[@quantity = 10]/Product[PART_OF/Category[@type = 'junk']]/../Supplier
ret(P1) = {(X2, is, Order), (X2-ORDERS(quantity = 10)-> X3), (X3, is, Product), (X4, is, Category), (X4, has, type = junk), (X3-PART_OF->X4), (X5<-•-X4), (X5, is, Supplier)}
first_var(P1) = X2
V = Order
v(V) = X2
ret(V) = {(X2, is, Order)}
E = ORDERS[@quantity= 10]
name(E) = ORDERS[@quantity = 10]
P1 = Product[PART_OF/Category[@type = 'junk']]/../Supplier
ret(P1) = {(X3, is, Product), (X4, is, Category), (X4, has, type = junk), (X3-PART_OF->X4), (X5<-•-X4), (X5, is, Supplier)}
first_var(P1) = X3
V = Product[PART_OF/Category[@type = 'junk']]
v(V) = X4
ret(P) = {(X3, is, Product), (X4, is, Category), (X4, has, type = junk), (X3-PART_OF->X4)}
..
P = Supplier
ret(P) = {(X5, is, Supplier)}
first_var = X5
E = PART_OF
name(E) = PART_OF
P = Category[@type = 'junk']
ret(P) = {(X3, is, Product), (X4, is, Category), (X4, has, type = junk)}
first_var(P) = X4
V = Supplier
v(V) = X5
ret(V) = {(X5, is, Supplier)}
V1 = Category[@type = 'junk']
v(V1) = X4
ret(V1) = {(X4, is, Category), (X4, has, type = junk)}
V = Category
v(V) = X4
ret(V) = {(X4, is, Category)}

Fig. 2. Parsing an XPath Query.

5 Conclusions

The likely reason why XPath has not been applied to property graphs is that XML data model does not contain the properties of relationships. We solve the problem by mapping XPath nodes to both vertices and edges in the graph data model, thus enabling consistent handling of vertices and edges. Using attribute grammars, we compile XPath expressions to regular path queries in a natural way, giving a framework to compile XPath to other graph query languages. Expressing XPath's nested paths in the predicates treats structural and data filtering as equal, which can be seen as a non-conventional approach.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers Principles, Techniques and Tools. Addison-Wesley, Reading (1986)
2. Angles, R., Gutierrez, C.: Survey of graph database models. ACM Comput. Surv. **40**(1), 1–39 (2008)
3. Angles, R., Prat-Perez, A., Dominguez-Sal, D., Larruba-Pey, J.L.: Benchmarking database systems for social network. In: First International Workshop on Graph Data Management Experiences and Systems, pp. 1–7 (2013)
4. Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., Vrgoč, D.: Foundations of modern query languages for graph databases. ACM Comput. Surv. **50**(5), 1–40 (2017)
5. Angles, R.: The Property Graph Database Model. Universidad de Talca, Department of Computer Science (2018)
6. Angles, R., et al.: G-CORE: A core for future graph query language. In: Proceedings of the 2018 International Conference on Management of Data, pp. 1421–1432 (2018)
7. Apache TinkerPop, Gremlin Query Language. https://tinkerpop.apache.org/gremlin.html. Accessed 16 Oct 2024
8. Barceló, P.: Querying graph databases. In: Proceedings of the 32nd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 175–187 (2013)
9. Barceló, P., Libkin, L., Lin, A.W., Wood, P.T.: Expressive languages for path queries over graph-structured data. ACM Trans. Database Syst. **37**(4), 1–46 (2012)
10. Benedikt, M., Koch, C.: XPath leashed. ACM Comput. Surv. **41**(1), 1–54 (2009)
11. Benedikt, M., Wenfei, F., Kuper, G.: Structural properties of XPath fragments. Theoret. Comput. Sci. **336**, 3–31 (2005)
12. van Bruggen, R.: Learning Neo4j: Run Blazingly Fast Queries on Complex Graph Datasets with the Power of the Neo4j Graph Database. Packt Publishing (2014)
13. Buneman, P.: Semistructured data. In: Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 117–121. (1997)
14. Calvanese, D., De Giacomo, G., Lenzerini, M., Vardi, M.Y.: Containment of conjunctive regular path queries with inverse. In KR **2000**, 176–185 (2000)
15. Cruz, I., Mendelzon, A., Wood, P.: A graphical query language supporting recursion. SIGMOD Record **16**(3), 323–330 (1987)
16. Deutsch, A., et al.: Graph pattern matching in GQL and SQL/PGQ. In: Proceedings of the 2022 International Conference on Management of Data, pp. 2246–2258 (2022)
17. Foulds, L.R.: Graph Theory Applications. Springer, New York (1992)
18. Francis, N., et al.: Cypher: An evolving query language for property graphs. In: Proceedings of the 2018 International Conference on Management of Data, pp. 1433–1445. (2018)
19. Furche, T., Linse, B., Bry, F., Plexouakis, D., Gottlob, G.: RDF querying: Language constructs and evaluation methods compared. In: Barahona, P., Bry, F., Franconi, E., Henze, N., Sattler, U. (eds), Reasoning Web. Reasoning Web 2006. LNCS, vol. 4126, pp. 1–52, Springer, Heidelberg (2006)
20. Harrison, G.: Next Generation Databases NoSQL and Big Data. Apress (2015)
21. ISO, ISO/IEC 39075:2024. https://www.iso.org/standard/76120.html. Accessed 16 Mar 2025
22. Knuth, D.: Semantics of context-free languages. Math. Syst. Theory **2**(2), 127–145 (1968)
23. Kostylev, E., Reutter, J., Vrgoč, D.: Static analysis of navigational XPath over graph databases. Inf. Process. Lett. **116**(7), 467–474 (2016)

24. Libkin, L., Martens, W., Vrgoč, D.: Querying graph databases with XPath. In: Proceedings of the 16th International Conference on Database Theory, pp. 129–140 (2013)
25. Libkin, L., Martens, W., Vrgoč, D.: Querying graphs with data. J. ACM **63**(2), 1–53 (2016)
26. Maiolo, S., Etcheverry, L., Marotta, A.: Data profiling in property graph databases. ACM J. Data Inf. Qual. **12**(4), 1–27 (2020)
27. Mendelzon, A., Wood, P.T.: Finding regular simple paths in graph databases. SIAM J. Comput. **24**(6), 1235–1258 (1995)
28. Neo4J, Cypher Query Language. https://neo4j.com/developer/cypher/. Accessed 28 Oct 2024
29. Neo4J. https://neo4j.com/. Accessed 16 Oct 2024
30. Oltenau, D., Furche, T., Bry, F.: Evaluating complex queries against XML streams with polynomial combined complexity. Key Technol. Data Manage. **3112**, 31–44 (2004)
31. Paakki, J.: Attribute grammar paradigms - a high-level methodology in language implementation. ACM Comput. Surv. **27**(2), 196–255 (1995)
32. van Rest, O., Hong, S., Jinha, K., Meng, X., Chafi, H.: PGQL: a property graph query language. In: Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, pp. 1–6 (2016)
33. Rodriquez, M., Neubauer, P.: Constructions from dots and lines. Bull. Am. Soc. Inf. Sci. Technol. **36**(6), 35–41 (2010)
34. de Virgilio, R., Maccioni, A., Torlone, R.: Converting relational to graph databases. In: First International Workshop on Graph Data Management Experiences and Systems, pp. 1–6 (2013)
35. W3C, XML Path Language (XPath). Accessed 16 Oct 2024
36. W3C, Resource Description Framework (RDF) Concepts and Abstract Syntax. Accessed 16 Oct 2024
37. Waite, W., Goos, G.: Compiler Construction. Springer, New York (1983)
38. Wood, P.: Query languages for graph databases. ACM SIGMOD Rec. **41**(1), 50–60 (2012)