The Property Graph Database Model

Renzo Angles

[1] Dept. of Computer Science, Universidad de Talca, Chile
[2] Center for Semantic Web Research

Abstract. Most of the current graph database systems have been designed to support property graphs. Surprisingly, there is no standard specification of the database model behind such systems. This paper presents a formal definition of the property graph database model. Specifically, we define the property graph data structure, basic notions of integrity constraints (e.g. graph schema), and a graph query language.

1   Introduction

A *graph database system*, or just *graph database*, is a system specifically designed for managing graph-like data following the basic principles of database systems [5]. The graph databases are gaining relevance in the industry due to their use in several domains where graphs and network analytics are required [17]. Popular graph databases are Neptune [1], Cosmos [10], Neo4j [18] and Titan [19].

The fundamental abstraction behind a database system is its database model. A *Database Model* should define three main components: a set of data structure types (i.e. the data model), a set of query operators or inference rules (i.e. the query language), and a set of integrity rules [7]. In the context of graph databases, a *Graph Database Model* is a model where data structures for the schema and/or instances are modeled as graphs (or generalizations of them), the data manipulation is expressed by graph-oriented operations (i.e. a graph query language), and appropriate integrity constraints are defined over the graph structure [4].

Despite all of the work already developed around graph databases, the current market of graph databases is fragmented, starting with the lack of consensus about a unique graph data model over which all the systems could be developed. Although most of the systems are designed to store property graphs (i.e. labeled multigraphs where both nodes and edges can contain pairs of the form property-value), there could be considerable differences among the provided components and their features, specially the definition of query languages and integrity constraints. The absence of such standardization causes several problems. In particular, the research on methods and techniques applicable to all the systems is very restricted due to the absence of a common theoretical foundation.

The main contribution of this paper is a formal definition of the property graph database model. Specifically, we define the property graph data structure (Section 2), propose basic notions of integrity constraints (Section 3), and describe a graph query language (Section 4).

1.1 Related work

The notion of property graph was introduced by Rodriguez and Neubauer in [15]. It is possible to find several works presenting informal definitions of the property graph model (e.g. [18]), but the number of works presenting a formal definition is reduced [6,9,16]. Several notions of integrity constraints for graph databases are presented in [4]. However, the literature about integrity constraints for property graph databases is very restricted [11,13,12]. There is no standard query language for property graphs, although some proposals are available [2]. G-CORE [3] is a recent proposal which integrates the main and relevant features provided by current graph query languages like Cypher [8] and PGQL [14].

## 2 The property graph data model

Informally, a property graph is a directed labelled multigraph with the special characteristic that each node or edge could maintain a set (possibly empty) of property-value pairs. From a data modeling point of view, a node represents an entity, an edge represents a relationship between entities, and a property represents an specific feature of an entity or relationship. Next we present a formal definition of the abstract notion described above.

Assume that $\mathbf{L}$ is an infinite set of labels (for nodes and edges), $\mathbf{P}$ is an infinite set of property names, $\mathbf{V}$ is an infinite set of atomic values, and $\mathbf{T}$ is a finite set of datatypes (e.g., *integer*). Given a set X, we assume that $\mathrm{SET}^+(X)$ is the set of all finite subsets of X, excluding the empty set. Given a value $v \in \mathbf{V}$, the function $\mathrm{type}(v)$ returns the data type of v. The values in $\mathbf{V}$ will be distinguished as quoted strings.

Definition 1. *A property graph is a tuple $G = (N, E, \rho, \lambda, \sigma)$ where:*

1. *N is a finite set of nodes (also called vertices);*
2. *E is a finite set of edges such that E has no elements in common with N;*
3. *$\rho : E \to (N \times N)$ is a total function that associates each edge in E with a pair of nodes in N (i.e., ρ is the usual incidence function in graph theory);*
4. *$\lambda : (N \cup E) \to \mathrm{SET}^+(\mathbf{L})$ is a partial function that associates a node/edge with a set of labels from L (i.e., λ is a labeling function for nodes and edges);*
5. *$\sigma : (N \cup E) \times \mathbf{P} \to \mathrm{SET}^+(\mathbf{V})$ is a partial function that associates nodes/edges with properties, and for each property it assigns a set of values from $\mathbf{V}$.*

Given two nodes $n_1, n_2 \in N$ and an edge $e \in E$, such that $\rho(e) = (n_1, n_2)$, we will say that n_1 and n_2 are the "source node" and the "target node" of e respectively. Additionally, given a property $(o, p) \in (N \cup E) \times \mathbf{P}$ and the assignment $\sigma(o, p) = \{v_1, \ldots, v_n\}$, we will use $(o, p) = v_i$ with $1 \leq i \leq n$ as a shorthand representation for a single property where o is the "property owner", p is the "property name" and v is the "property value". Note that our definition supports multiple labels for nodes and edges, and multiple values for the same property.
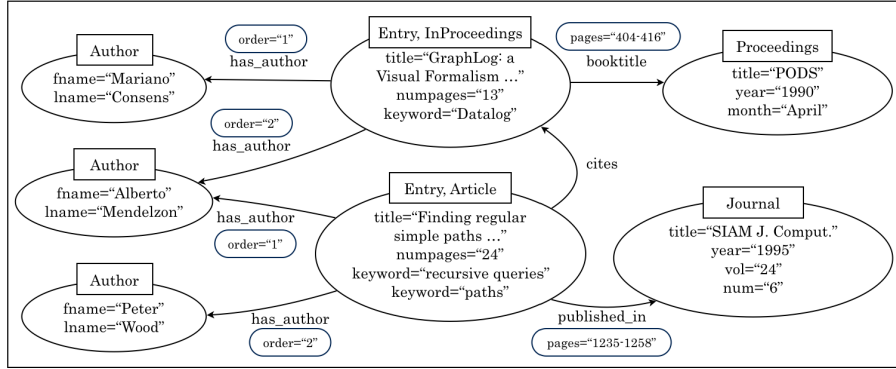
**Fig. 1.** Example of property graph representing bibliographic information.

Figure 1 presents a graphical representation of a property graph that contains bibliographic information. Following our formal definition, we will have that:

$N = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7\}$
$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$
$\lambda(n_1) = \{\text{Author}\}, (n_1, \text{fname}) = \text{"Mariano"}, (n_1, \text{lname}) = \text{"Consens"}$
$\lambda(n_2) = \{\text{Author}\}, (n_2, \text{fname}) = \text{"Alberto"}, (n_2, \text{lname}) = \text{"Mendelzon"}$
$\lambda(n_3) = \{\text{Author}\}, (n_3, \text{fname}) = \text{"Peter"}, (n_3, \text{lname}) = \text{"Wood"}$
$\lambda(n_4) = \{\text{Article}\}, (n_4, \text{title}) = \text{"GraphLog ..."}, (n_4, \text{numpages}) = \text{"13"}, (n_4, \text{keyword}) = \text{"Datalog"}$
$\lambda(n_5) = \{\text{Article}\}, (n_5, \text{title}) = \text{"Finding ..."}, (n_5, \text{numpages}) = \text{"24"}$
$(n_5, \text{keyword}) = \text{"recursive queries"}, (n_5, \text{keyword}) = \text{"paths"}$
$\lambda(n_6) = \{\text{Conference}\}, (n_6, \text{title}) = \text{"PODS"}, (n_6, \text{year}) = \text{"1990"}, (n_6, \text{month}) = \text{"April"}$
$\lambda(n_7) = \{\text{Journal}\}, (n_7, \text{title}) = \text{"SIAM ..."}, (n_7, \text{year}) = \text{"1995"}, (n_7, \text{vol}) = \text{"24"}, (n_7, \text{num}) = \text{"6"}$
$\rho(e_1) = (n_4, n_1), \lambda(e_1) = \{\text{has\_author}\}, (e_1, \text{order}) = \text{"1"}$
$\rho(e_2) = (n_4, n_2), \lambda(e_2) = \{\text{has\_author}\}, (e_2, \text{order}) = \text{"2"}$
$\rho(e_3) = (n_5, n_2), \lambda(e_3) = \{\text{has\_author}\}, (e_3, \text{order}) = \text{"1"}$
$\rho(e_4) = (n_5, n_3), \lambda(e_4) = \{\text{has\_author}\}, (e_4, \text{order}) = \text{"2"}$
$\rho(e_5) = (n_5, n_4), \lambda(e_5) = \{\text{cites}\}$
$\rho(e_6) = (n_4, n_6), \lambda(e_6) = \{\text{published\_at}\}, (e_6, \text{pages}) = \text{"404–416"}$
$\rho(e_7) = (n_5, n_7), \lambda(e_7) = \{\text{published\_at}\}, (e_7, \text{pages}) = \text{"1234–1258"}$

3   Integrity Constraints

Integrity constraints are general statements and rules that define the set of consistent database states or changes of state or both [7]. In the literature about graph database models we can find the following notions of integrity constrains: schema instance-consistency, identity and referential integrity constraints, functional and inclusion dependencies [4]. In this paper we concentrate our interest on constraints related to schema-instance consistency.

A data schema is a powerful data modeling feature that allows to describe the structure of the data and enforce its consistency. In this sense, a graph schema allows to define the graph structure by specifying the types of nodes, the types of edges, and the properties for such types. Next, we introduce a basic notion of property graph schema which is enough to be supported by current graph databases. Additionally, we present a set of special constraints that can be added to define more strict schemas.
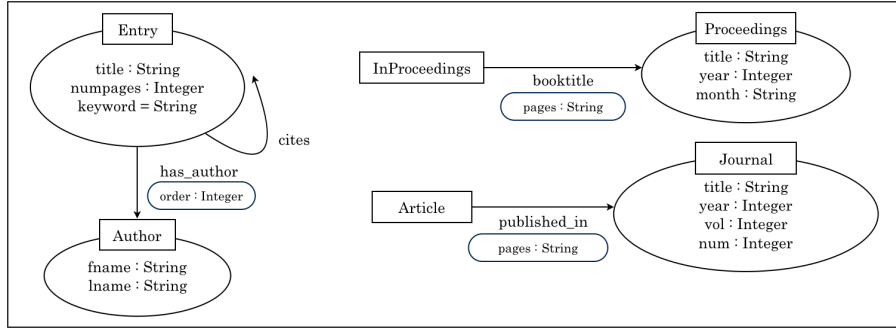
Fig. 2. Example of property graph schema for modeling bibliographic information.

3.1 Property graph schema

Recall that **L** is an infinite set of labels, **P** is an infinite set of property names, and **T** is a finite set of datatypes (e.g., *String, Integer, etc.*). Informally, a property graph schema defines the node types, the edge types, and the properties for such types (including the corresponding datatypes).

Definition 2. *A property graph schema is a tuple $S = (T_N, T_E, \beta, \delta)$ where:*

1. *$T_N \subset \boldsymbol{L}$ is a finite set of labels representing node types;*
2. *$T_E \subset \boldsymbol{L}$ is a finite set of labels representing edge types, satisfying that T_E and T_N are disjoint;*
3. *$\beta : (T_N \cup T_E) \times \boldsymbol{P} \to \boldsymbol{T}$ is a partial function that defines the properties for node and edge types, and the datatypes of the corresponding values;*
4. *$\delta : (T_N, T_N) \to \mathrm{SET}^+(T_E)$ is a partial function that defines the edge types allowed between a given pair of node types.*

Figure 2 presents a graphical representation of a property graph schema for the property graph presented in Figure 1. The formal description of such schema is given as follows:

$T_N = \{$ Entry, Author, InProceedings, Article, Proceedings, Journal $\}$
$T_E = \{$ has_author, cites, booktitle, published_in $\}$
$\beta(\text{Entry,title}) = \text{String}, \ldots, \beta(\text{Journal,num}) = \text{Integer},$
$\beta(\text{has\_author,order}) = \text{Integer}, \ldots, \beta(\text{published\_in,pages}) = \text{String}$
$\delta(\text{Entry,Author}) = \{\text{has\_author}\}, \delta(\text{Entry,Entry}) = \{\text{cite}\},$
$\delta(\text{InProceeedings,Proceedings}) = \{\text{booktitle}\}, \delta(\text{Article,Journal}) = \{\text{published\_in}\}$

Schema-instance consistency. The notion of schema-instance consistency implies that an instance property graph satisfies the structural restrictions stablished by a given property graph schema. Such notion is formally defined as follows.

Definition 3. *Given a property graph $G = (N, E, \rho, \lambda, \sigma)$ and a property graph schema $S = (T_N, T_E, \beta, \delta)$, we say that G is* valid *with respect to S when:*

1. *For each node $n \in N$, it applies that $\lambda(n) \subseteq T_N$;*

2. *For each edge $e \in E$, it applies that $\lambda(e) \subseteq T_E$;*
3. *For each node or edge property $(o, p) = v$ in G, it satisfies that there exists $\beta(t_o, p) = dt$ in S such that $t_o \in \lambda(o)$ and $\mathrm{type}(v) = dt$;*
4. *For each edge $e \in E$ such that $\rho(e) = (n, n')$, it satisfies that there exist $l_e \in \delta(t, t')$ such that $l_e \in \lambda(e)$, $t \in \lambda(n)$ and $t' \in \lambda(n')$.*

Assume that a *graph database* is a pair $D(S, G)$ where S is a property graph schema and G is a property graph. We will say that D satisfies the schema-instance consistency constraint if G is valid with respect to S.

3.2 Additional constraints

Note that the definition of property graph schema is basic in the sense of allowing the description of the minimal characteristics for the data in a graph database. However, such basic definition can be extended to support additional constraints.

Mandatory and optional properties / edges. Note that the function β in Definition 2 does not differentiate between required and optional properties. In order to support such feature, we can define that the domain of β can be divided in two sets P_m and P_o, i.e. $P_m \cup P_o = \mathrm{dom}(\beta)$, where P_m is the set of mandatory properties and P_o the set of optional properties. Additionally, we need to include the following restriction in Definition 3: "For each pair $(t, p) \in P_m$, and every node / edge o in G such that $t \in \lambda(o)$, it applies that $(o, p) \in \mathrm{dom}(\sigma)$, i.e. the node / edge o contains the property p". A similar approach could be applied to introduce mandatory and optional edges.

Unique attributes. Given a node (or edge) type t, it could be useful to define an attribute whose value is unique for each instance of t. Given a property graph schema $S = (T_N, T_E, \beta, \delta)$, assume that $P_u \subset \mathrm{dom}(\beta)$ is the set of unique properties. Given a property graph $G = (N, E, \rho, \lambda, \sigma)$, we say that G satisfies a unique property $(t, p) \in P_u$ iff, for each pair of distinct objects o_1, o_2 such that $t \in \lambda(o_1)$ and $t \in \lambda(o_2)$ it applies that $(o_1, p) \neq (o_2, p)$.

Cardinality constraints. A cardinality constraint could define the number of occurrences of a given property (for a node or edge type), or the precise number of outgoing edges for a given node type. Mandatory and optional constraints could be viewed as cardinality constraints (e.g. an optional property is defined by a minimal cardinality of 0 and a maximum cardinality of 1).

4 Query Language

In this section we present a basic graph query language for the property graph data model defined above. In general terms, a query takes as input a graph (called the *target graph*), applies graph pattern matching, and returns a table of values as output. The syntax of the language is a combination of three representative query languages (i.e. SQL, SPARQL and Cypher). The semantics of the language are defined in terms of a transformation to non-recursive safe Datalog with negation.

4.1 Syntax

The syntax of the query language is based on four main clauses: `SELECT`, `FROM`, `MATCH` and `WHERE`. These clauses allow to express basic pattern matching queries. Additionally, the query language introduces the `UNMATCH` and `UNION` clauses in order to support the negation and union of graph patterns respectively.

Pattern matching. The core feature of the language is the support to express a graph pattern which is matched against the target graph. Example 1 shows an example of pattern matching query. In general terms, the `SELECT` clause defines the output of the query (specifically, a table of values), the `FROM` clause defines the input graph, the `MATCH` clause defines a graph pattern, and the `WHERE` clause defines conditions over the graph pattern. Next we describe each clause in detail.

Example 1. Example of pattern matching query. The query returns the names of authors which have papers in common, i.e. the co-authorship relationship.

```
1    SELECT n2.fname AS Author1, n3.fname AS Author2, n1.title AS EntryTitle
2    FROM "biblio"
3    MATCH (n1:Entry)-[e1:has_author]->(n2:Author), (n1)-[e2:has_author]->(n3)
4    WHERE n2 != n3 AND e1.order < e2.order
```

Assume that a *property graph database* is a collection of property graphs, each one having a name. Hence, the `FROM` clause defines a comma separated list of graph names that conforms the graph to be queried. In the Example 1, the `FROM` clause defines a single target graph named "biblio" which corresponds to the one presented in Figure 1. In the case of having multiple graphs, the target graph will be the union of nodes and edges occurring in all the graphs.

The `MATCH` clause allows to define a *graph pattern* which is a collection of basic graph patterns. A *basic graph pattern* (BGP) is an expression of the form $(v_1 : t_1) - [v_2 : t_2]- > (v_3 : t_3)$ where v_1 and v_3 are variables referencing nodes, v_2 is a variable referencing an edge, t_1 and t_3 are node type labels, and t_2 is an edge type label. The expression $(v_1 : t_1)$ is also a valid BGP which allows to obtain the nodes in the graph. To be valid, a BGP must contain at least a node variable. A set of BGPs is called a *join graph pattern*. In our example, the join GP matches the nodes `n1`, `n2` and `n3` such that `n1` is of type `Entry`, `n2` is of type `Author`, `n3` does not specify a type. Both `n2` and `n3` are connected with `n1` by an edge `has_author`, and the corresponding edges are bound by variables `e1` and `e2`.

The `WHERE` clause allows to define a filter condition that restrict the subgraphs obtained by the pattern matching step. A *filter condition* is defined as a set of equality conditions and build-in functions, connected by the operators `AND`, `OR` and `NOT`. The expressions $v_1 \ \theta \ v_2$, $v_1.p_1 \ \theta \ v_2.p_2$ and $v_1.p_1 \ \theta \ a$ are *equality conditions* where v_1 and v_2 are variables, p_1 and p_2 are property names, a is an atomic value, and θ is an equality operator in the set $\{=,!=,>,>=,<,<=\}$. In our example, the filter condition indicates that the Authors `n2` and `n3` must be distinct, and the order of Author `n2` is lower than the Author `n3`. Note that the former condition applies to a node variable, and the latter applies to an edge variable.

Finally, the `SELECT` clause defines the output of the query called the result table. A *result table* is a tabular structure (equivalent to a relational table) where the head contains property names and the body contains atomic values. The `SELECT` clause defines a comma separated list of expressions of the form $v.p$ `AS` c where, v is a variable ocurring in the `MATCH` clause, p is a property name, and c is the name of the corresponding column in the result table. Hence, the `SELECT` clause in Example 1 allows to return the following result table.

Author1	Author2	EntryTitle
"Mariano"	"Alberto"	"GraphLog: a Visual Formalism ..."
"Alberto"	"Peter"	"Finding regular simple paths ..."

Negation. The negation of graph patterns is supported by including the `UNMATCH` clause, which also includes a graph pattern expression. In general terms, the `MATCH` clause is evaluated as a positive graph pattern matching whereas `UNMATCH` is evaluated as a safe negation of graph patterns. In order to satisfy such safe condition, it is mandatory that at least one node variable of the `UNMATCH` clause also occurs in the `MATCH` clause. The `UNMATCH` clause can be accompanied by its own `WHERE` clause.

Example 2 shows a negation of graph patterns. First, the `MATCH` and the `WHERE` clauses allow to obtain all the pairs of nodes (`n2`, `n3`) of type Author (i.e. the cartesian product of authors). Then, the `UNMATCH` clause allows to remove the pairs of authors having a paper in common. Hence, the result of the pattern matching step is the set of pairs of authors having no papers in common.

Example 2. Negation of graph patterns. The query returns the names of authors having no papers in common.

```
1    SELECT n2.fname AS Author1, n3.fname AS Author2
2    FROM "biblio"
3    MATCH (n2:Author), (n3:Author)
4    WHERE n2 != n3
5    UNMATCH  (n1)-[:has_author]->(n2), (n1)-[:has_author]->(n3)
```

Union. Given two join graph patterns, P_1 and P_2, a union graph pattern is an expression of the form P_1 `UNION` P_2. In general terms, the `UNION` operator combines the solution of P_1 with the solutions of P_2, the variables of P_1 plus the variables of P_2. Example 3 shows a query expressing the union of graph patterns.

Example 3. Union of graph patterns. The query returns the list of entries in the database, and for each entry it includes the conference's title or the journal's title when necessary.

```
1    SELECT n1.title AS EntryTitle, n2.title AS Booktitle, n3.title AS Journal
2    FROM "biblio"
3    MATCH (n1:Entry)-[:booktitle]->(n2:Proceedings)
4        UNION (n1:Entry)-[:published_in]->(n3:Journal)
```

The result of the query in Example 3 is the following result table:

EntryTitle	Booktitle	Journal
"GraphLog: a Visual Formalism ..."	"PODS"	null
"Finding regular simple paths ..."	null	"SIAM J. Comput."

Note that above result table contains null values. It occurs when the set of variables of the join graph patterns, connected by the `UNION`, are different.

4.2 Semantics

In order to provide a semantics for our query language, we present a method to translate a property graph and a query into a Datalog program. The query transformation allows to show the expressive power of our query language. For the sake of space, the method will be informally presented by using examples.

Data transformation. Assume that G is the property graph presented in Figure 1. Then, G could be transformed into the set of Datalog facts presented in Example 4.

Example 4. Datalog rules obtained from a property graph.

```
1    Node(10), Label(10,"Author"), Prop(101,10,"fname","Mariano"), Prop(102,10,"lname","Consens"),
2    Node(11), Label(11,"Author"), Prop(111,11,"fname","Alberto"),
3     Prop(112,11,"lname","Mendelzon"),
4    Node(12), Label(12,"Author"), Prop(121,12,"fname","Peter"), Prop(122,12,"lname","Wood"),
5    Node(13), Label(13,"Entry"), Label(13,"InProceedings"), Prop(131,13,"title","Graphlog ..."),
6    Prop(132,13,"numpages","13"), Prop(133,13,"keyword","Datalog"),
7    Node(14), Label(14,"Entry"), Label(14,"Article"), Prop(141,14,"title","Finding ..."),
8     Prop(142,14,"numpages","24"), Prop(143,14,"keyword","recursive queries"),
9     Prop(144,14,"keyword","paths"),
10   Node(15), Label(15,"Proceedings"), Prop(151,15,"title","PODS"),
11    Prop(152,15,"year","1990"), Prop(153,15,"month","April"),
12   Node(16), Label(16,"Journal"), Prop(161,16,"title","SIAM ..."), Prop(162,16,"year","1995"),
13    Prop(163,16,"vol","24"), Prop(164,16,"num","16"),
14   Edge(20,13,10), Label(20,"has_author"), Prop(201,20,"order","1"),
15   Edge(21,13,11), Label(21,"has_author"), Prop(211,21,"order","2"),
16   Edge(22,14,11), Label(22,"has_author"), Prop(221,22,"order","1"),
17   Edge(23,14,12), Label(23,"has_author"), Prop(231,23,"order","2"),
18   Edge(24,14,13), Label(24,"cites"),
19   Edge(25,13,15), Label(25,"booktitle"), Prop(251,25,"pages","404-416"),
20   Edge(26,14,16), Label(26,"published_in"), Prop(261,26,"order","1235-1258")
```

Let us explain the meaning of all the facts shown in Example 4. The fact `Node(10)` encodes a node where 10 is the node identifier. `Label(10,"Author")` encodes the type of the node having identifier 10 (the same applies to edges). The fact `Prop(101,10,"fname","Mariano")` encodes a property where 101 is the property identifier and 10 is the identifier of the node having the property (the same applies to edges). `edge(20,13,10)` encodes an edge where 20 is the edge identifier, 13 is the source node identifier, and 10 is the target node identifier. Note that there are identifiers for nodes, edges and properties. In the case of the properties, the identifiers are fundamental to support multivalued properties.

Query transformation. Given the extensional database (of facts) presented above, we can add intentional predicates to query the data, i.e. a Datalog query. For each example graph query Q presented above, we will present a Datalog query Q' satisfying that Q and Q' are equivalent, i.e. their results are equivalent.

Example 5 shows a datalog query which is equivalent to the graph query presented in Example 1. The datalog query defines specific predicates to represent the main clauses in the graph query, i.e. `select(...)`, `match(...)` and `where(...)`. Additionally, the predicate `gp(...)` indicates a graph pattern, `jpgX(...)` indicates a join graph pattern, and `condX(...)` indicates a filter condition. In general terms, a pattern matching query is transformed into a set of positive rules.

Example 5. Datalog query equivalent to the query presented in Example 1.

```
1  select(v1,v2,v3):- match(n1,e1,n2,e2,n3), Prop(p1,n2,"fname",v1),
2   Prop(p2,n3,"fname",v2), Prop(p3,n1,"title",v3)
3  match(n1,e1,n2,e2,n3):- gp(n1,e1,n2,e2,n3), where(n1,e1,n2,e2,n3)
4  gp(n1,e1,n2,e2,n3):- jgp1(n1,e1,n2,e2,n3)
5  jgp1(n1,e1,n2,e2,n3):- bgp1(n1,e1,n2), bgp2(n1,e2,n3)
6  bgp1(n1,e1,n2):- Node(n1), Label(n1,"Entry"), Node(n2), Label(n2,"Author"),
7                   Edge(e1,n1,n2), Label(e1,"has_author")
8  bgp2(n1,e2,n3):- Node(n1), Node(n3),
9                   Edge(e2,n1,n3), Label(e2,"has_author")
10 where(n1,e1,n2,e2,n3):- cond1(n1,e1,n2,e2,n3), cond2(n1,e1,n2,e2,n3)
11 cond1(n1,e1,n2,e2,n3):- jgp1(n1,e1,n2,e2,n3), n2 != n3
12 cond2(n1,e1,n2,e2,n3):- jgp1(n1,e1,n2,e2,n3), Prop(id1,e1,"order",v1),
13                         Prop(id2,e2,"order",v2), v1 < v2
```

Example 6. Datalog query equivalent to the query presented in Example 2.

```
1  select(v1,v2):- match(n2,n3), Prop(p1,n2,"fname",v1), Prop(p2,n3,"fname",v2)
2  match(n2,n3):- gp(n2,n3), where(n2,n3), not unmatch(n2,n3)
3  gp(n2,n3):- jgp1(n2,n3)
4  jgp1(n2,n3):- bgp1(n2), bgp2(n3)
5  bgp1(n2):- Node(n2), Label(n2,"Author")
6  bgp2(n3):- Node(n3), Label(n3,"Author")
7  where(n2,n3):- cond1(n2,n3)
8  cond1(n2,n3):- jgp1(n2,n3), n2 != n3
9  unmatch(n2,n3):- jgp2(n1,n2,n3)
10 jgp2(n1,n2,n3):- bgp3(n1,n2), bgp4(n1,n3)
11 bgp3(n1,n2):- Node(n1), Node(n2), Edge(e,n1,n2) Label(e,"has_author")
12 bgp4(n1,n3):- Node(n1), Node(n3), Edge(e,n1,n3) Label(e,"has_author")
```

Example 7. Datalog query equivalent to the query presented in Example 3.

```
1  select(v1,v2,v3):- match(n1,n2,n3), Prop(p1,n1,"title",v1),
2                     Prop(p2,n2,"title",v2), Prop(p3,n3,"title",v3)
3  match(n1,n2,n3):- gp(n1,n2,n3)
4  gp(n1,n2,n3):- ugp1(n1,n2,n3)
5  ugp1(n1,n2,n3):- bgp1(n1,n2), Null(n3)
6  ugp1(n1,n2,n3):- bgp2(n1,n3), Null(n2)
7  bgp1(n1,n2):- Node(n1), Label(n1,"Entry"), Node(n2), Label(n2,"Proceedings"),
8               Edge(e,n1,n2), Label(e,"booktitle")
9  bgp2(n1,n3):- Node(n1), Label(n1,"Entry"), Node(n2), Label(n2,"Journal"),
10              Edge(e,n1,n2), Label(e,"published_in")
11 Null("null")
```

Example 6 shows a datalog query which is equivalent to the graph query presented in Example 2. The main difference is given by the use of the predicate `unmatch(...)` to implement the negation of graph patterns. In the line 10, we can see that the bindings for `unmatch(n2,n3)` are given by the join graph pattern `jgp2(n1,n2,n3)`. Additionally, the rule presented in line 3 shows the use of the `not` operator to implement the "negation by failure" of the predicate `unmatch`. In order to generate a safe datalog program, we need to ensure that all the variables occurring in `unmatch(...)` also occur in `gp(...)`.

Example 7 shows a datalog query which is equivalent to the graph query presented in Example 3. In this case, the predicate `ugp1(n1,n2,n3)` encodes a

union graph pattern that combines the predicates `bgp1(n1,n2)` and `bgp2(n1,n3)`. Considering that `bgp1` and `bgp2` have disjoint variables, we need to use the `Null` predicate in order to assign the `"null"` value when necessary. In this sense, the datalog program must contain the fact `Null(null)` (see line 13).

The examples presented above show that any graph query can be transformed into a Datalog query. Moreover, it is possible to see that our language supports the same types of queries provided by Datalog, i.e. join, union and equalities.

References

1. Amazon Neptune - Fast, reliable graph database build for cloud: https://aws.amazon.com/neptune/
2. Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., Vrgoĉ, D.: Foundations of modern query languages for graph databases. CSUR 50(5) (2017)
3. Angles, R., Arenas, M., Barceló, P., Boncz, P., Fletcher, G., Gutierrez, C., Lindaaker, T., Paradies, M., Plantikow, S., Sequeda, J., van Rest, O., Voigt, H.: G-core: A core for future graph query languages. In: SIGMOD (2018)
4. Angles, R., Gutierrez, C.: Survey of graph database models. CSUR 40(1) (2008)
5. Angles, R., Gutierrez, C.: Introduction to graph data management. Tech. rep., https://arxiv.org/abs/1801.00036 (December 2017)
6. Ciglan, M., Averbuch, A., Hluchy, L.: Benchmarking Traversal Operations over Graph Databases. In: ICDE Workshops. pp. 186–189 (2012)
7. Codd, E.F.: Data Models in Database Management. In: Workshop on Data abstraction, Databases and Conceptual Modeling. pp. 112–114. ACM Press (1980)
8. Cypher - Graph Query Language: http://neo4j.com/developer/cypher-query-language/
9. Hartig, O.: Reconciliation of RDF* and Property Graphs. Tech. rep., http://arxiv.org/abs/1409.3288 (2014)
10. Microsoft Azure Cosmos DB: `https://docs.microsoft.com/en-us/azure/cosmos-db/introduction`
11. Pokorný, J.: Conceptual and database modelling of graph databases. In: IDEAS. pp. 370–377. ACM (2016)
12. Pokorný, J., Valenta, M., Kovačič, J.: Integrity constraints in graph databases. Procedia Computer Science 109, 975 – 981 (2017)
13. Rabuzin, K., Sestak, M., Konecki, M.: Implementing unique integrity constraint in graph databases. In: ICCGI. pp. 48–53 (2016)
14. van Rest, O., Hong, S., Kim, J., Meng, X., Chafi, H.: PGQL: a property graph query language. In: GRADES. p. 7. ACM (2016)
15. Rodriguez, M.A., Neubauer, P.: Constructions from dots and lines. Bul. Am. Soc. Info. Sci. Tech. 36(6), 35–41 (2010)
16. Rodriguez, M.A., Neubauer, P.: The graph traversal pattern. Tech. rep., AT&Ti and NeoTechnology (April 2010)
17. Sakr, S., Pardede, E.: Graph Data Management: Techniques and Applications. IGI Global, 1st edn. (2011)
18. The Neo4j Graph Platform: https://neo4j.com/
19. Titan - Distributed Graph Database: http://titan.thinkaurelius.com/