# Schema Validation and Evolution for Graph Databases

Angela Bonifati[1][0000−0002−9582−869X], Peter Furniss[2], Alastair Green[2], Russ Harmer[3][0000−0002−0817−1029], Eugenia Oshurko[3], and Hannes Voigt[2]

[1] Lyon 1 University & CNRS Liris, France
angela.bonifati@univ-lyon1.fr
[2] Neo4J, Germany & UK
{peter.furniss,alastair.green,hannes.voigt}@neo4j.com
[3] UdL, CNRS, ENS Lyon, UCBL1, France
{russell.harmer,ievgeniia.oshurko}@ens-lyon.fr

**Abstract.** Despite the maturity of commercial graph databases, little consensus has been reached so far on the standardization of data definition languages (DDLs) for property graphs (PG). Discussion on the characteristics of PG schemas is ongoing in many standardization and community groups. Although some basic aspects of a schema are already present in most commercial graph databases, full support is missing allowing to constraint property graphs with more or less flexibility.
In this paper, we show how schema validation can be enforced through homomorphisms between PG schemas and PG instances by leveraging a concise schema DDL inspired by Cypher syntax. We also briefly discuss PG schema evolution that relies on graph rewriting operations allowing to consider both prescriptive and descriptive schemas.

**Keywords:** Graph Databases · Graph Schema Modelling · Graph Schema Validation.

## 1 Introduction

Property graph databases are modern data management systems that use graph structures, such as nodes, edges and properties, to encode semantically complex data [3]. Graph database technology has made tremendous progress with many commercial products—such as Neo4j, Oracle PGX, SAP HANA Graph, Redis Graph, Cypher for Apache Spark and TigerGraph—and yet little consensus has been reached so far on the standardization of graph data querying and manipulation or of data definition languages (DDLs).

The aim of ISO SC32/WG3 is to develop a new international standardized query language—called GQL[4]—for property graphs, with support from the activities of the wider community such as OpenCypher[5] and G-Core [1].

[4] https://www.gqlstandards.org/
[5] http://www.opencypher.org/

Standardization of graph data querying and manipulation is therefore well under way.

At present, there are only a few examples of property graph systems offering schema and DDL, e.g. Neo4j's Cypher for Apache Spark and TigerGraph. Neo4j 3.5 already provides the means to express certain basic aspects of schemas, e.g. the use of *unique property* and *property existence* constraints enables us to *enforce* nodes (or edges) to have certain properties. However, this does not allow users to express more advanced aspects of schemas such as specifying, for a given node or edge label, the collection of all possible associated properties; or constraining whether or not an edge may exist between nodes with certain labels.

In this paper, we make the following specific contributions: (i) a schema model (and corresponding DDL) specifying labels and (mandatory) properties for nodes and edges with mixed composition and facilitating strict typing of every graph element (Section 2); (ii) a mathematical framework for schema validation allowing us to construct both instances and schemas as property graphs and to enforce schema validation through the existence of a homomorphism from instance to schema (Section 3); and (iii) graph rewriting rules [5] and their application to propagate changes from schema to instance (or vice versa) while keeping the instance and schema consistent at all times (Section 4).

## 2 PG Schema Language

We introduce in this section an OpenCypher-based[6] schema DDL for Property Graphs (PG). Although informing and feeding the ongoing standardization process, our DLL must not be intended as a standard proposal since its main purpose is to substantiate the algorithmic contributions presented in the paper. The basic components of a schema definition assume a finite set of *labels* $\mathcal{L}$, a set of *property keys* $\mathcal{K}$ and a finite set of data types $\mathcal{T}$.

**Property graph type.** A property graph type is a triple $(\mathcal{BT}, \mathcal{NT}, \mathcal{ET})$ where $\mathcal{BT}$ is a set of element types, $\mathcal{NT}$ is a set of node types and $\mathcal{ET}$ is a set of edge types. A property graph type provides the schema for a PG. Multiple PGs can share a property graph type to the effect that they will have the same schema.

**Property type.** A property type is a pair $(k, t)$, where $k \in \mathcal{K}$ is the property key and $t \in \mathcal{T}$ is its data type.

**Element type.** An element type $b \in \mathcal{BT}$ is a 4-tuple $(l, P, M, E)$, where $l \in \mathcal{L}$ is a label, $P$ is a set of property types, $M \subseteq P$ is a subset of mandatory property types and $E \subseteq \mathcal{BT}$ is the set of element types that $b$ extends.

Hence, "`Message {content: STRING?, length: INTEGER}`" is a declaration of the element type $m = (\texttt{Message}, \{pt_1, pt_2\}, \{pt_2\}, \emptyset)$, where $pt_1 = (\texttt{content}, \texttt{STRING})$ and $p_2 = (\texttt{length}, \texttt{INTEGER})$; while "`Post :: Message {language: STRING?}`" declares the element type $p = (\texttt{Post}, \{pt_3 = (\texttt{language}, \texttt{STRING})\}, \emptyset, \{m\})$.

An element type is allowed to extend multiple other element types, but must not extend itself either directly or indirectly. All element types of a property

[6] `https://s3.amazonaws.com/artifacts.opencypher.org/openCypher9.pdf`

graph type must be disambiguated by their label. Where clear from context, we use the label to denote the corresponding element type.

**Exposed (mandatory) property types and labels.** The set of exposed property types of an element type $b = (l, P, M, E)$ is defined as $\texttt{prop}(b) := P \cup \bigcup_{c \in E} \texttt{prop}(c)$, i.e. all the property types that $b$ possesses, either directly or through inheritance. Similarly, we define $\texttt{mand}(b)$ to be the set of exposed mandatory property types of $b$ and $\texttt{labels}(b)$ to be the set of exposed labels of $b$. For instance, for element type $p$ from above we have $\texttt{prop}(p) = \{pt_1, pt_2, pt_3\}$, $\texttt{mand}(p) = \{pt_2\}$, and $\texttt{labels}(p) = \{\texttt{Post}, \texttt{Message}\}$.

For an element type $b$ to be valid, $\texttt{prop}(b)$ must not have two or more property types with the same property key, i.e. all properties types of a element type are disambiguated by their property key. Where clear from context, we will use the property key to denote the corresponding property type. For instance, for the element type $p$ above, we have $\texttt{prop}(p) = \{\texttt{content}, \texttt{length}, \texttt{language}\}$, $\texttt{mand}(p) = \{\texttt{length}\}$ and $\texttt{labels}(p) = \{\texttt{Post}, \texttt{Message}\}$. Note that $\texttt{labels}(b)$ is unambiguous for all element types $b$ of a property graph type.

**Node type.** A node type $nt \in \mathcal{ET}$ is a 1-tuple $(b)$, where $b \in \mathcal{BT}$ is an element type. For a node type $nt = (b)$, we define $\texttt{prop}(nt) = \texttt{prop}(b)$, $\texttt{mand}(nt) = \texttt{mand}(b)$, and $\texttt{labels}(nt) = \texttt{labels}(b)$.

**Edge type.** An edge type $et \in \mathcal{ET}$ is a triple $(s, b, t)$, where $s$, $b$, and $t$ are element types. Exposed (mandatory) property and label sets are defined analogously to node types based on $b$. Note that $s$ and $t$ need not be node types. This allows a single edge type to be inherited by multiple node types.

**Example.** The following snippet of the OpenCypher PG schema DDL creates a property graph type that captures an excerpt of the LDBC SNB [8] schema [7].

```
CREATE GRAPH TYPE snb (
  // element types
  Person {
    firstName : STRING, lastName : STRING
  },
  Message {
    creationDate : TIMESTAMP, browserUsed : STRING
  },
  Comment <: Message {},
  Post <: Message {
    imageFile : STRING?
  },
  // node types
  (Person), (Post), (Comment),
  // edge types
  (Person)-[KNOWS]->(Person),
  (Person)-[LIKES]->(Message),
  (Message)-[HAS_CREATOR]->(Person),
  (Comment)-[REPLY_OF]->(Message)
```

[7] The complete PG schema encoding of LDBC SNB is reported in [4].

)

## 3 Schema Validation

In this section, we provide a mathematical formalization of our schema notion that, in particular, allows us to interpret a DDL specification as a PG. We present the mathematical definitions of schemas and instances as property graphs in Section 3.1 and then discuss the application of homomorphisms to the schema validation problem in Section 3.2.

### 3.1 Schemas and instances as property graphs

We fix countable sets $\mathcal{O}$, $\mathcal{K}$ and $\mathcal{V}$ of *objects*, *keys* and *values* respectively. For the purposes of this paper, we assume that $\mathcal{V}$ contains (at least) basic types of integers, booleans, strings and dates.

A *property graph* is defined to be a tuple $(N, E, \eta, P, \nu, M)$ where $N$ and $E$ are disjoint, finite subsets of $\mathcal{O}$ called *nodes* and *edges*; $\eta : E \to N \times N$ is a function assigning a source and target node to each edge; $P \subseteq (N \cup E) \times \mathcal{K}$ is a finite set of *properties*; $\nu \subseteq P \times \mathcal{V}$ is a finite relation, assigning *sets of values* to properties; and $M \subseteq P$ is a set of *mandatory* properties. The requirement that $\nu$ be finite means that each node and each edge has finitely many properties, each of which has a finite set of associated values.

A schema $(\mathcal{BT}, \mathcal{NT}, \mathcal{ET})$ specified in our DDL from Section 2 can be interpreted as a property graph $S$ in the following way. The nodes $N_S$ are the node types $\mathcal{NT}$and we have an edge $e \in E_S$ from $n_1$ to $n_2$ in $E_S$ if, for some $l_1 \in \texttt{labels}(n_1)$ and $l_2 \in \texttt{labels}(n_2)$, there is an edge type $(n_1, e, n_2) \in \mathcal{ET}$. Note that a node type always gives rise to a single node of $S$ whereas an edge type may give rise to many edges in the schema graph; this is how inheritance in the DDL syntax is 'expanded out' in the schema graph $S$ interpreting the property graph type. Each node and edge has the (mandatory) properties specified by its corresponding node or edge type. As an example, the schema defined in Section 2 and interpreted as a property graph is illustrated in Figure 1.

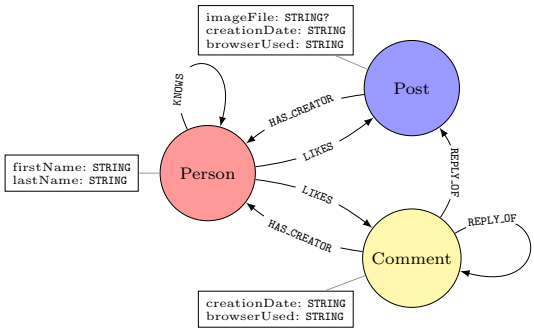



Fig. 1: An extract from the SNB schema

### 3.2 Schema validation via graph homomorphisms

Let $G$ and $S$ be property graphs where $N_G \cup E_G$ and $N_S \cup E_S$ are disjoint. A *homomorphism* $h : G \to S$ is a function $h_{\mathcal{N}} : N_G \to N_S$ and a function $h_{\mathcal{E}} : E_G \to E_S$, mapping nodes and edges of $G$ to nodes and edges of $S$, such that $\eta_S \circ h_{\mathcal{E}} = (h_{\mathcal{N}} \times h_{\mathcal{N}}) \circ \eta_G$. We write $h := h_{\mathcal{N}} \cup h_{\mathcal{E}}$. We further require that (i) if $(x, k) \in P_G$ then $(h(x), k) \in P_S$; (ii) if $((x, k), v) \in \nu_G$ then $((h(x), k), v) \in \nu_S$; and (iii) if $(h(x), k) \in M_S$ then $(x, k) \in M_G$.

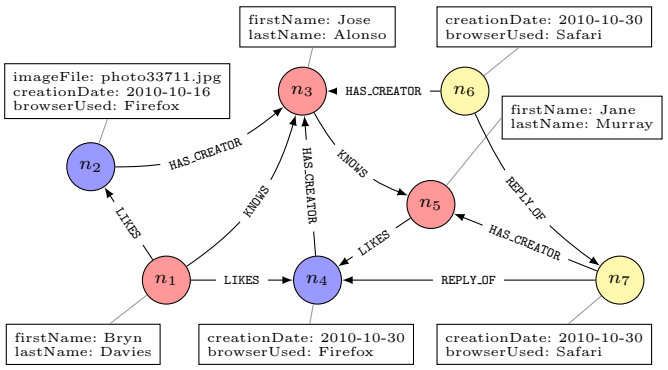



Fig. 2: A valid instance of the SNB schema extract

We can view a homomorphism $h : G \to S$ as a formalization of the notion *schema validation*, i.e. that $G$ respects the 'schema' $S$: each node/edge $x$ of $G$ is an instance of the schema node/edge $h(x)$; edges in $S$ constrain which edges can exist in $G$; and properties that are mandatory in the schema $S$ are mandatory (so must occur) in $G$. In the example instance $G$ of Figure 2, we have used colours to encode the homomorphism $h$, i.e. all yellow nodes are `Comment`s, etc. In the DDL of Section 2, the fact that all element types are disambiguated by their label would also allow us to determine $h$ provided we include these labels in the instance $G$.

**The `ReGraph` library.** The Python library `ReGraph`[8] provides an implementation of the presented system. It enables us to construct property graphs and structure them into hierarchies (DAGs) of graphs via homomorphisms. In this paper we limit our use of the library to the special case of two graphs connected by a single homomorphism, i.e. $h : G \to S$ as this is sufficient to express that $G$ respects the schema $S$. Our system thus provides an *abstraction barrier* that gives the illusion that the underlying Neo4j graph is, in fact, two separate graphs—a data graph and a schema—related by a homomorphism that guarantees schema validation. In the next section, we explain briefly how *updates* to either of these graphs are performed in such a way as to maintain the invariant of schema validation.

[8] `https://github.com/Kappa-Dev/ReGraph`

## 4 Property graph rewriting

In our approach, the data graph *and* its schema are represented as PGs; as such, we can use graph rewriting rules [5] to perform updates of either. Informally, a rewriting rule consists of a *pattern*—of which there can be zero, one or many *instances* in the graph $G$ we wish to modify—together with a collection of modifications to be effected. In the case of PGs, these operations are: addition and deletion of elements; cloning and merging of nodes; and modification of the set of values associated with a property. The rule is applied by selecting an instance in $G$ and performing the associated operations. The effect of a rule application remains localized to the subgraph of $G$ picked out by the choice of instance which, in practice, is very small compared with $G$ itself.

In general, an update invalidates the homomorphism that previously existed and which guaranteed compliance of the data to the schema. In our mathematical formulation, and its associated implementation discussed briefly below, we *automatically* recompute a canonically updated homomorphism that restores compliance [9]. The way in which compliance can be broken—and the process by which we restore it—depends on whether the update was made to the data graph or to the schema.

In the first case, compliance can be broken by the addition of nodes, edges or properties or by the merging of nodes in the data graph. By default, the addition of a new element $e$ is *propagated* to the schema, i.e. we add a new element to the schema to type $e$ in the data graph. We can further specify that $e$ is actually typed by an existing element of the schema; this can be done explicitly by the user or, more commonly, computed automatically through the use of labels. However, in the case of the merge of two nodes, their associated typing nodes in the schema *must* be merged—unless they already had the same type (in which case no change to the schema is necessary).

In the second case, compliance can be broken by the deletion of an element or by the cloning of nodes in the schema. By default, the cloning of a node $n$ is *propagated* to the data graph, i.e. we clone *all* instances of $n$ in the data graph. For some or all instances of $n$, we may not wish to propagate but rather specify the particular clone of $n$ that should be used to type it, i.e. a *concept refinement*; again, this can be specified directly by the user or computed automatically through the use of labels. However, in the case of the deletion of an element, we *must* delete *all* its instances in the data graph.

An update of the data graph that propagates to the schema can be *blocked* in our implementation. This would be appropriate in situations where the schema is already well-developed and we expect all incoming data to comply, i.e. we consider our schema to be *prescriptive*. However, in an earlier phase of development, the ability to propagate *automatically* new elements to the schema enables the user to focus simply on gathering their data of interest and allows the schema to adapt appropriately, i.e. the schema is considered to be *descriptive*. As such, our approach—in addition to providing the guarantee that updates never break schema compliance—also provides support for the natural development cycle of an application.

In our implementation, a rewriting rule is translated into a Cypher query that manipulates the underlying Neo4j graph in such a way as to preserve the correspondence with the data and schema graphs. As outlined above, an update of one graph may—but need not necessarily—induce a further update of the other to maintain schema validation. A detailed account will be included in the long version of this paper and can be found in the arXiv preprint [4].

## 5 Related Work

Schema evolution [17] is a well established topic in data management. A set of principles ruling out schema and instance evolution under schema constraints was discussed in [10]. Various approaches exist to increase usability and efficiency, e.g. schema evolution-aware query languages [18] or providing a general framework to describe database evolution in the context of evolving applications [7]. Meta Model Management 2.0 [2] introduced tools to match, merge and diff given relational schema versions. The resulting mappings couple the evolution of the schema and the data; however, they are complex relationships between heterogeneous schemas, as in data integration and ETL scenarios, i.e. they only deal with schema evolution after the fact. Recently, PRISM [6] and InVerDa [11] have provided advanced database schema evolution tools. PRISM focuses on plain database evolution but allows the answering of queries using former schema versions with respect to the current data. InVerDa provides co-existing schema versions via bidirectional transformations with symmetric relational lenses [12]. However, none of the above approaches goes beyond a prescriptive schema.

SHACL [14] is a language for validating RDF graphs. Shapes are used to validate RDF instances against a set of conditions. SHACL supports RDF term restrictions, cardinality constraints, and predicate constraints. Research on ontologies also considered the problem of update propagation to instances using Description Logic mappings [13]. However, such mappings are quite complex when contrasted with the implicit homomorphisms considered in our work. The distinction between descriptive and prescriptive schemas as carried out in our paper is reminiscent of open and close tuple types as used for instance in JSON [16]. In particular, the schema flexibility pointed out in our work affects not only types but entire portions of the schemas and as such is more general.

Graph rewriting has been used in a variety of areas related to knowledge representation and meta-modelling. For example, triple graph grammars [19,15]—which correspond very closely to our rewriting rules—provide a means to specify bidirectional model transformations and have been used in various applications such as conformance testing and model synchronization.

## 6 Concluding Remarks

We have presented a schema DDL for PGs following the ASCII-art syntax of Cypher and shown how schema validation and evolution can be simulated via a mathematical framework that enforces and maintains schema validation.

Our next step is to enrich the DDL for the expression of finer constraints and to define a DML for our graph update operations.

*Acknowledgements.* We would like to thank Petra Selmer (Neo4j) for her careful proof reading and useful feedback. This work was partially funded by a grant from the Fédération Informatique de Lyon.

## References

1. Angles, R., et al.: G-CORE: A core for future graph query languages. In: SIGMOD. pp. 1421–1432 (2018)
2. Bernstein, P.A., Melnik, S.: Model Management 2.0: Manipulating Richer Mappings. In: SIGMOD. pp. 1–12 (2007)
3. Bonifati, A., Fletcher, G., Voigt, H., Yakovets, N.: Querying Graphs. Synthesis Lectures on Data Management, Morgan & Claypool Publishers (2018)
4. Bonifati, A., Furniss, P., Green, A., Harmer, R., Oshurko, E., Voigt, H.: Schema Validation and Evolution for Graph Databases. CoRR **abs/1902.06427** (2019)
5. Corradini, A., Heindel, T., Hermann, F., König, B.: Sesqui-pushout rewriting. In: International Conference on Graph Transformation. pp. 30–45. Springer (2006)
6. Curino, C., Moon, H.J., Zaniolo, C.: Graceful Database Schema Evolution: the PRISM Workbench. PVLDB **1**(1), 761–772 (Aug 2008)
7. Domínguez, E., Lloret, J., Rubio, A.L., Zapata, M.A.: MeDEA: A database evolution architecture with traceability. DKE **65**(3), 419–441 (Jun 2008)
8. Erling, O., Averbuch, A., Larriba-Pey, J., Chafi, H., Gubichev, A., Prat-Pérez, A., Pham, M., Boncz, P.A.: The LDBC Social Network Benchmark: Interactive Workload. In: SIGMOD. pp. 619–630 (2015)
9. Harmer, R., Oshurko, E.: Knowledge representation and update in hierarchies of graphs. In: International Conference on Graph Transformation. Springer (2019)
10. Hartung, M., Terwilliger, J.F., Rahm, E.: Recent advances in schema and ontology evolution. In: Schema Matching and Mapping, pp. 149–190 (2011)
11. Herrmann, K., Voigt, H., Pedersen, T.B., Lehner, W.: Multi-schema-version data management: data independence in the twenty-first century. The VLDB Journal **27**(4), 547–571 (2018)
12. Hofmann, M., Pierce, B.C., Wagner, D.: Symmetric Lenses. In: POPL (2011)
13. Kharlamov, E., Zheleznyakov, D., Calvanese, D.: Capturing model-based ontology evolution at the instance level: The case of dl-lite. J. Comput. Syst. Sci. **79**(6), 835–872 (2013)
14. Knublauch, H., Kontokostas, D.: Shapes Constraint Language (SHACL). W3C Recommendation 20 July 2017
15. Königs, A., Schürr, A.: Tool Integration with Triple Graph Grammars - A Survey. Electronic Notes in Theoretical Computer Science **148**(1), 113–150 (2006)
16. Ong, K.W., Papakonstantinou, Y., Vernoux, R.: The SQL++ semi-structured data model and query language: A capabilities survey of sql-on-hadoop, nosql and newsql databases. CoRR **abs/1405.3631** (2014)
17. Rahm, E., Bernstein, P.A.: An Online Bibliography on Schema Evolution. SIGMOD Record **35**(4), 30–31 (Dec 2006)
18. Roddick, J.F.: SQL/SE - A Query Language Extension for Databases Supporting Schema Evolution. SIGMOD Record **21**(3), 10–16 (Sep 1992)
19. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Workshop on Graph-Theoretic Concepts in Computer Science. pp. 151–163 (1994)