EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science

Databases Research Group

Formal Specification and Practical Validation of Property Graph Schemas

Master Thesis

Nimo Beeren

Supervisors:
George Fletcher
Nikolay Yakovets
Felix Mannhardt

Eindhoven, October 2022

Abstract

Graph databases are increasingly receiving attention from industry and academia, due in part to their flexibility; a schema is often not required. In particular, the *property graph* model enables natural expression of data from a wide variety of domains. However, schemas can significantly benefit query optimization, data integrity, and documentation. We present a formal property graph schema model based on conceptual data modeling methods, integrating constraints on mandatory and allowed properties, property data types, edge endpoints, and edge cardinality. Moreover, we specify schema validation semantics using first-order logic rules. These rules are implemented using graph queries for *Neo4j*, *JanusGraph*, and *TigerGraph*, which we evaluate through a controlled experiment. Our results demonstrate feasibility of our approach, with execution times scaling linearly with the size of the data.

Keywords: graph databases, data modeling, property graphs, graph schemas, schema specification, schema validation, integrity constraints

Acknowledgments

I would like to warmly thank everyone who has given me energy, inspired me, and cheered me on throughout this project, which has been my biggest challenge yet. In particular, I would like to thank George, my supervisor, for his consistent guidance and our inspiring conversations. I would like to thank Michel, my dad, for always being there for me. I would like to thank Libby for helping me find my own way to flourish. Finally, I would like to thank my friends for all their support and kindness.

This thesis concludes my two-year master's degree, which has been a transformative time for me. I am grateful for the opportunities I have been given, and the path they have opened up for me. I am excited to discover the new places it leads to.

Nimo Beeren
October 2022

Table of Contents

List of Figures

List of Tables

List of Definitions

List of Listings

Chapter 1

Introduction

Graph databases have been steadily growing in popularity in recent years, receiving attention from both industry and academia. While the traditionally dominant relational database organizes data into tables and requires a fixed schema, graphs offer a simple yet powerful model consisting of nodes and edges which can be structured freely. The *property graph* model, being the predominant data model among graph databases today, associates nodes and edges with *labels* and key–value pairs known as *properties*. This enables very natural expression of data originating from a wide variety of domains.

However, the freedom that graphs permit comes at a cost. Without a schema, we miss out on opportunities for query optimization, we risk degradation of data integrity, and we lack a formally verifiable source of documentation. We aim to bring schemas back to the world of graphs, blending the flexibility of the graph model with the structure of relational databases.

Our overarching goal is to develop an end-to-end framework for property graph schema specification and validation. To this end, we propose a schema model based on common conceptual data modeling methods, integrating constraints on mandatory and allowed properties, property data types, edge endpoints, and edge cardinality. Then, the notion of schema conformance is defined using first-order logic rules. Moreover, we provide a prototypical implementation of schema validation in the form of concrete graph queries, and we investigate the practical feasibility of our approach using contemporary graph databases by means of a controlled experiment.

Evaluating the property graph database systems *Neo4j*, *JanusGraph*, and *TigerGraph*, we find that our implementation enables schema validation with acceptable execution times, which scale linearly with the size of the data. We observe a significant difference in performance between databases, where TigerGraph is fastest, followed by Neo4j, then JanusGraph. However, JanusGraph shows drastic improvement when the data does not conform to the schema.

The rest of this thesis is organized as follows. Chapter 2 describes related work on graphs and schemas in academia and industry. In Chapter 3, we formally specify our property graph data model. Chapter 4 details the design and formalization of our property graph schema model and specifies the rules for schema conformance. Chapter 5 addresses the translation of our formal rules to graph queries. In Chapter 6, we empirically evaluate our proposed implementation. Chapter 7 discusses the limitations of our work and identifies opportunities for future work. Chapter 8 concludes our work with the main takeaways and recommendations.

Chapter 2

Related Work

2.1 Graph Integrity Constraints

An area of research closely related to schema is that of integrity constraints. The problem of expressing and validating integrity constraints for graphs has received some attention in the literature. A large part of this research has been on functional dependencies, which have been studied extensively in the context of relational databases [Fagin and Vardi, 1984; Abiteboul et al., 1999]. The idea of functional dependencies has been adapted to graph data, with the first work focussing on *key constraints*. These aim to uniquely identify entities represented by nodes [Fan et al., 2015], and have been extended to include more general dependencies [Fan et al., 2016]. The current state of the art of key constraints is presented in PG-Keys [Angles et al., 2021]. For further discussion of graph integrity constraints, see Bonifati et al. [2018].

2.2 Graph Schemas

While integrity constraints are typically scoped to a subset of the database, a schema serves as a model of the entire database. We will now discuss some approaches to graph schemas which have been discussed in the literature. While these methods are based on different graph models, they may be adapted to the property graph model.

Buneman et al. [1997] propose a method that represents both data and schema as edge-labeled graphs. This enables the specification of the types of edges and paths that are allowed to occur in a database instance, but does not cover mandatory edges or more general cardinality constraints.

Colazzo and Sartiani [2015] introduce another schema formalism for edge-labeled data graphs. They define a schema as a set of pairs (r_{in}, r_{out}), where r_{in} and r_{out} are regular expressions. The semantics is that every node must match a schema element (r_{in}, r_{out}), meaning that the node's incoming edges match r_{in} and its outgoing edges match r_{out}. Using Regular Path Queries (RPQs), edge cardinality can be exactly specified. When extending the language to Nested Regular Expressions (NREs), it is possible to specify structural constraints over arbitrarily large graph patterns.

SHACL[1] is a language for specifying constraints on RDF graphs [Pan, 2009]. These constraints themselves are also represented as RDF graphs and are called *shapes*. Constraints on values, data types and cardinality are supported. This enables specification of allowed, optional, and mandatory edges. A review of the formal frameworks used to study the SHACL language and the validation of RDF graphs against SHACL schemas can be found in Pareti and Konstantinidis [2022].

2.3 Property Graph Schemas

The property graph data model has been adopted by numerous contemporary graph database systems (dating back to at least 2007 with the first release of Neo4j[2]), but a formal specification was missing until one was proposed by Angles [2018]. The author also brought integrity constraints to the property graph model and defined a basic notion of schema using first-order logic rules. In this work, no distinction was made between mandatory and optional properties, nor were cardinality constraints addressed.

Pokorný et al. [2017] describe and implement several integrity constraints for property graphs, including mandatory properties, endpoint constraints, data type constraints, cardinality constraints, and property uniqueness. In addition, they introduce *label coexistence* constraints, which express that two particular labels may not occur on the same node, or that one label may only occur together with another label. However, these constraints are not formally specified, and they are not integrated into a schema model. To specify these kinds of constraints, a syntax extension for the Cypher query language [Francis et al., 2018] is proposed. A prototypical implementation for Neo4j is given, which is briefly evaluated.

Bonifati et al. [2019] present a property graph schema validation approach based on graph homomorphisms. They differentiate between mandatory and optional properties, though edges are always interpreted as optional. Cardinality constraints are not discussed. The authors look at schemas from two different angles: they can be descriptive in the sense that they only reflect the data, or they can be prescriptive by means of enforcing constraints on data. This distinction is relevant in particular to the topic of schema evolution, where it may be desirable to switch between these two modes as an application matures. It is noted that most contemporary graph database systems only provide descriptive schema tools.

Lbath et al. [2021] look at property graph schemas from the perspective of schema inference. Their goal is to extract an accurate and complete schema for an arbitrary property graph instance. To this end, a property graph schema formalism is defined, supporting cardinality constraints, mandatory and optional edges, as well as subtyping. In addition, a syntax for schema definition is presented, and the schema inference pipeline is evaluated using real-world datasets. The problem of validating a graph against a given schema is not addressed. Statistical methods are applied in [Bonifati et al., 2022] to improve performance.

Lei [2021] proposes a semi-automated method for schema extraction which incorporates expert knowledge. The user can adjust parameters and similarity measures which affect how the schema is generated. An implementation and experimental results are given. Cardinality constraints are not covered.

[1] https://www.w3.org/TR/shacl/

[2] https://neo4j.com/open-source-project/

	Angles	Pokorný†	Bonifati	Lbath‡	Lei	Ours
Mandatory properties	×	✓	✓	✓	✓	✓
Allowed properties	×	✓	✓	✓	×	✓
Endpoint constraints	✓	✓	✓	✓	✓	✓
Data type constraints	✓	✓	✓	✓	✓	✓
Maximum cardinality	×	✓	×	✓*	×	✓
Minimum cardinality	×	✓	×	✓*	×	✓
Property uniqueness	×	✓	×	×	×	×
Label coexistence	×	✓	×	×	×	×
Subtype relations	×	×	×	✓	×	×

Table 2.1: Comparison of property graph schema formalisms found in the literature and ours as detailed in Chapter 4. ∗: only one-to-one, one-to-many, etc. †: only informal specification. ‡: no specification of conformance.

To summarize, Table 2.1 compares the main papers discussed in this section in terms of schema features, and places them alongside our schema formalism which is detailed in Chapter 4. We elaborate here on the meaning of the listed features. Mandatory property constraints express that some property must exist on a node or edge. Allowed property constraints express that no properties other than the ones explicitly specified may exist. Endpoint constraints express that edges may not connect nodes which do not conform to some specified types. Data type constraints express that the value of a property must be of a particular data type. Maximum and minimum cardinality constraints express that a node must have a particular number of incoming or outgoing edges of a particular type. Property uniqueness constraints express that no two nodes may have the same value for a particular property. Subtype relations express that an object inherits properties or incident edges from another object. Label coexistence constraints express which labels may or may not occur together on the same object.

2.4 Property Graph Databases

Current property graph database systems vary in their support and philosophy around schema. Some require the user to specify a schema, while others infer a schema from data. In this section, we discuss the differences in terms of data models and schema capabilities of three of the most popular[3] property graph databases: *Neo4j*, *JanusGraph*, and *TigerGraph*.

[3] https://db-engines.com/en/ranking/graph+dbms (accessed July 2022)

Neo4j. Being one of the first to adopt the property graph data model, Neo4j has grown to become the most popular graph database engine today. Neo4j supports the Cypher query language [Francis et al., 2018]. In their data model, edges have a single label, while nodes have one or more.

Neo4j's approach to schema is primarily implicit: after inserting data, a schema that describes the data can be automatically constructed using built-in functions. In addition, some constraints can be explicitly specified, namely existence of mandatory properties and key constraints (enterprise edition only), and uniqueness of property values. Note that if a property is a key, then it is mandatory and unique.

JanusGraph. Originating from the open-source Titan project, JanusGraph is a continuation of the effort to create a distributed and highly scalable graph database. The Gremlin query language [Rodriguez, 2015] enables pattern matching by means of graph traversals. In JanusGraph's data model, nodes and edges always have one label. Properties can have sets and lists as values, and there may exist multiple properties with the same key on the same node. Furthermore, properties themselves can have properties. In this sense, the data model is like a blend of property graphs and RDF [Pan, 2009].

JanusGraph has the largest set of schema features among all systems we have investigated. Similarly to Neo4j, JanusGraph can automatically generate a schema during operation. However, this functionality can be disabled, in which case the schema must be explicitly defined. There is built-in support for specifying which node labels, edge labels, and property names may exist. Furthermore, it can be specified which properties may exist depending on the label of a node or edge. Property values are restricted to a data type and may be single-valued or multi-valued (lists or sets). In addition, the types of the source and target nodes that may be connected by an edge with a particular label can be constrained. Edge cardinality can be constrained to one-to-one, one-to-many, many-to-one, many-to-many, or "simple" (at most one edge of a particular label between any pair of nodes). Finally, there are features to support static (immutable) nodes, time-to-live (TTL), and unidirectional edges which can only be traversed from source to target.

All of JanusGraph's schema features are centered around specifying what is allowed in the graph, but mandatory properties and edges are not supported. Cardinality constraints are supported, but not to the full extent of our definition. To be precise, JanusGraph can impose constraints on the maximum edge cardinality, but not the minimum. For example, a many-to-one schema edge ensures that every target node has at most one incoming edge of a particular type, but does not prevent a target node from having no incoming edges.

TigerGraph. A unique aspect of TigerGraph [Deutsch et al., 2019] is that it is schema-first: the entire schema must be specified before a database is instantiated. This allows for more powerful optimizations, building on decades of research on relational databases. The query language is GSQL: an extension of SQL with graph pattern matching capabilities. In TigerGraph's data model, nodes and edges have exactly one label, but nodes can be associated with any number of *tags*, which are conceptually similar to labels. Edges can be directed or undirected.

TigerGraph's schema is strict, in the sense that every node label, edge label, and property must be explicitly defined. All properties have a fixed data type, which may be singular or multi-valued. There are no null values; if a property value is missing during insertion, a default value is used. All nodes have a primary key, which may be a single property

	Neo4j Community	**Neo4j Enterprise**	**JanusGraph**	**TigerGraph**	**Ours**
Mandatory properties	×	✓	×	×	✓
Allowed properties	×	×	✓	✓	✓
Endpoint constraints	×	×	✓	✓	✓
Data type constraints	×	×	✓	✓	✓
Maximum cardinality	×	×	✓*	×	✓
Minimum cardinality	×	×	×	×	✓
Property uniqueness	✓	✓	×	✓	×
Label coexistence	×	×	×	×	×
Subtype relations	×	×	×	×	×

Table 2.2: Comparison of current property graph databases and our implementation as detailed in Chapter 5. $*$: only one-to-one, one-to-many, etc.

or a composite key. Schema edges must specify a source and target node type, though it is possible to specify multiple types on either side. By default, directed edges can only be traversed from source to target, though a reverse edge can be automatically constructed if desired.

The schema features of these three databases are summarized in Table 2.2, where they are compared to our schema formalism which is detailed in Chapter 4.

Chapter 3

Property Graphs

We start by introducing our data model, which is based on the definition of *property graph* established by the Working Group for Database Languages (WG3) as part of ISO/IEC JTC1/SC32 [Deutsch et al., 2021]. The data model we define next is used throughout this thesis.

Our notion of property graph represents data as a directed attributed multigraph. Nodes and edges carry data in the form of a set of labels and a set of key–value pairs, called *properties*. We use the umbrella term *objects* to refer to nodes and edges. Being a *multigraph*, a property graph allows the existence of multiple edges between two nodes u and v. Furthermore, we allow $u = v$, in which case the edge is called a *self-loop*. For ease of notation, we do not consider undirected edges, although they could be simulated by attaching a special label or property to an edge.

For a formal definition, we assume the existence of the following countably infinite sets: the set of labels $\mathcal{L}$, the set of property names $\mathcal{N}$ and the set of property values $\mathcal{V}$.

Definition 1 (Basic record)**.** A *record* is a finite partial function $r : \mathcal{N} \nrightarrow \mathcal{V}$ that maps some property names to property values. We denote such records as $\langle n_1 : v_1, \ldots, n_k : v_k \rangle$ The set of all records is denoted as $\mathcal{R}$.

Definition 2 (Property graph)**.** A *property graph* is a tuple

$$G = (N, E, \rho, \lambda, \pi)$$

where

- N is a finite set of nodes;
- E is a finite set of edges such that $N \cap E = \emptyset$;
- $\rho : E \to (N \times N)$ is a total function mapping edges to ordered pairs of nodes;
- $\lambda : (N \cup E) \to 2^{\mathcal{L}}$ is a total function mapping nodes and edges to a (possibly empty) set of labels;
- $\pi : (N \cup E) \to \mathcal{R}$ is a total function mapping nodes and edges to a record.

Given a node u, the set of *outgoing* edges is given by $\{e \in E \mid \exists v \in N : \rho(e) = (u, v)\}$, and the set of *incoming* edges is given by $\{e \in E \mid \exists v \in N : \rho(e) = (v, u)\}$.

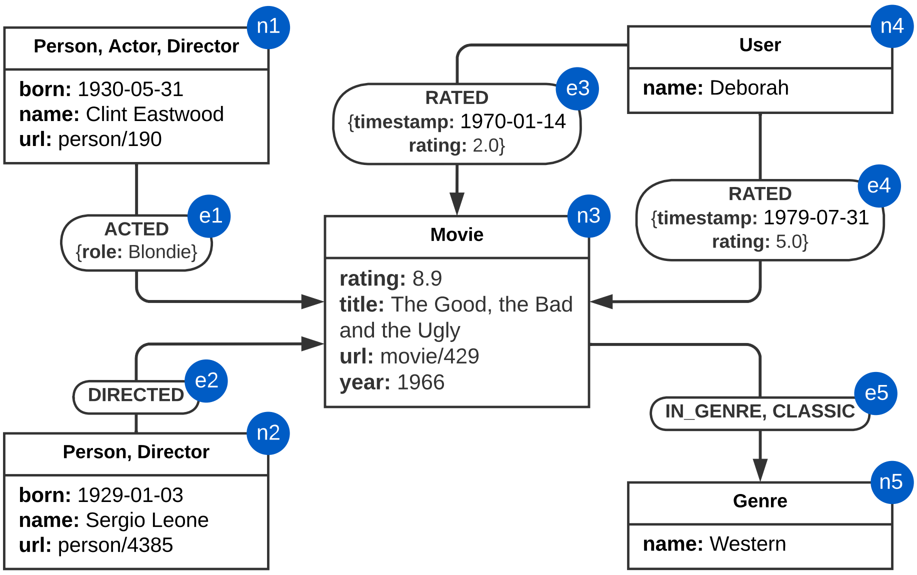


Figure 3.1: A property graph consisting of 5 nodes and 5 edges, representing a small movie database. Nodes and edges are given identifiers (in blue circles) in order to refer to them in text.

The functions src and trg map ordered pairs to their first and second element, i.e. $\mathtt{src}((u, v)) = u$ and $\mathtt{trg}((u, v)) = v$. To refer to the source and target *endpoints* of an edge e, we may write $\mathtt{src}(\rho(e))$ and $\mathtt{trg}(\rho(e))$ respectively.

An example of a property graph is given in Figure 3.1. Nodes are drawn as boxes, and edges are drawn as arrows. Node labels are written in the top compartment, and node properties are written in the bottom compartment. Edge labels are written inside a pill and edge properties are surrounded by '{' and '}', which may be omitted when an edge has no properties.

The example of Figure 3.1 can be mapped to our formal property graph model in the following way:

$$N = \{n_1, n_2, n_3, n_4, n_5\}$$
$$E = \{e_1, e_2, e_3, e_4, e_5\}$$
$$\rho(e_1) = (n_1, n_3) \quad \rho(e_2) = (n_2, n_3) \quad \rho(e_3) = (n_4, n_3) \quad \rho(e_4) = (n_4, n_3) \quad \rho(e_5) = (n_3, n_5)$$
$$\lambda(n_1) = \{\mathtt{Person}, \mathtt{Actor}, \mathtt{Director}\} \quad \lambda(n_2) = \{\mathtt{Person}, \mathtt{Director}\} \quad \lambda(n_3) = \{\mathtt{Movie}\}$$
$$\lambda(n_4) = \{\mathtt{User}\} \quad \lambda(n_5) = \{\mathtt{Genre}\} \quad \lambda(e_1) = \{\mathtt{ACTED}\} \quad \lambda(e_2) = \{\mathtt{DIRECTED}\}$$
$$\lambda(e_3) = \{\mathtt{RATED}\} \quad \lambda(e_4) = \{\mathtt{RATED}\} \quad \lambda(e_5) = \{\mathtt{IN\_GENRE}, \mathtt{CLASSIC}\}$$
$$\pi(n_1) = \langle \mathtt{born} : \mathtt{1930\text{-}05\text{-}31},\ \mathtt{name} : \mathtt{Clint\ Eastwood},\ \mathtt{url} : \mathtt{person/190} \rangle$$
$$\pi(n_2) = \langle \mathtt{born} : \mathtt{1929\text{-}01\text{-}03},\ \mathtt{name} : \mathtt{Sergio\ Leone},\ \mathtt{url} : \mathtt{person/4385} \rangle$$
$$\pi(n_3) = \langle \mathtt{rating} : \mathtt{8.9},\ \mathtt{title} : \mathtt{The\ Good,\ the\ Bad\ and\ the\ Ugly},\ \mathtt{url} : \mathtt{movie/429},$$
$$\quad \mathtt{year} : \mathtt{1966} \rangle \quad \pi(n_4) = \langle \mathtt{name} : \mathtt{Deborah} \rangle \quad \pi(n_5) = \langle \mathtt{name} : \mathtt{Western} \rangle$$
$$\pi(e_1) = \langle \mathtt{role} : \mathtt{Blondie} \rangle \quad \pi(e_2) = \langle \rangle \quad \pi(e_3) = \langle \mathtt{timestamp} : \mathtt{1970\text{-}01\text{-}14},\ \mathtt{rating} : \mathtt{2.0} \rangle$$
$$\pi(e_4) = \langle \mathtt{timestamp} : \mathtt{1979\text{-}07\text{-}31},\ \mathtt{rating} : \mathtt{5.0} \rangle \quad \pi(e_5) = \langle \rangle$$

Chapter 4

Property Graph Schemas

In this chapter, we design a schema model for property graphs, and provide a formal specification. We provide a set of first-order logic rules which determine whether a property graph conforms to a schema.

A schema should allow modeling of all kinds of entities and their relationships. A more expressive schema language enables the specification of more complex constraints, but this may come at the cost of greater research and engineering effort, as well as worse runtime performance. This needs to be balanced in the design of our schema model.

To explore the schema features that are commonly used in practice, let us look at some existing data modeling techniques. As a baseline, we consider the Entity–Relationship (ER) model as proposed by Chen [1976]. This model incorporates entities, relationships, attributes, and values. An *entity* is a "thing" that can be uniquely identified, a *relationship* is an association between entities, and an *attribute–value* pair represents information about an entity or relationship. Furthermore, an entity may have a *role* in a relationship, such as "white" or "black" in a chess match relationship. These are the concepts underlying many conceptual data modeling methods in use today.

After the original specification, the ER model has been extended in various ways. For example, a notation which introduced cardinality constraints, optionality, and subtype relations was developed by Barker [1990]. With these additions, we can create a more nuanced data model which fits the real world more closely.

In the next section, we first establish a basic definition of property graph schema, which we then extend with additional features such as cardinality constraints (Section 4.2) and optional properties (Section 4.3).

4.1 Basic Definition

In this section, we define a notion of property graph schema which incorporates the basic features of the ER model: entities, relationships, attributes, and values. Table 4.1 shows how the basic elements of the ER model can be mapped to the property graph model. Note that there are no named roles in the property graph model, but the direction of an edge does allow the distinction between the source and target of an edge. Conversely, the edge direction can be represented using roles in the ER model.

ER	PG
Entity	Node
Relationship	Edge
Attribute	Property name
Value	Value
Role*	Edge direction*

Table 4.1: Mapping between entity–relationship (ER) and property graph (PG) concepts. $*$: roles and edge direction can be used for similar purposes, but are not equivalent.

To formally define property graph schemas and schema conformance, we first assume the existence of a set of property types $\mathcal{T}$. Next, we introduce several supporting concepts.

Definition 3 (Basic property conformance)**.** For each property type $\tau \in \mathcal{T}$ there is a set $[\![\tau]\!] \subseteq \mathcal{V}$ that contains all property values that *conform* to the type τ.

The concept of property type is similar to a *value set* in the ER model. Value sets and property types are used to specify which values may be associated with an attribute, or which values a property is allowed to have[1]. A minor difference is that Chen [1976] postulates that values from different value sets can be equivalent, while we do not have a notion of value equivalence.

Definition 4 (Basic record type)**.** A *record type* is a finite partial function $\tau^{\mathsf{r}} : \mathcal{N} \nrightarrow \mathcal{T}$ that maps some property names to a property type. We denote such record types as $\langle a_1 : \tau_1, \ldots, a_n : \tau_n \rangle$.

Definition 5 (Basic record conformance)**.** We say that a record r *conforms* to a record type τ^{r}, denoted $r \in [\![\tau^{\mathsf{r}}]\!]$, if for each property name $k \in \mathcal{N}$ it holds that (1) $r(k)$ is defined iff $\tau^{\mathsf{r}}(k)$ is defined and (2) $r(k) \in [\![\tau^{\mathsf{r}}(k)]\!]$ if $r(k)$ and $\tau^{\mathsf{r}}(k)$ are defined.

Definition 6 (Object type)**.** An *object type* is a pair $\tau^{\mathsf{o}} = (L, \tau^{\mathsf{r}})$ where $L \subseteq \mathcal{L}$ is a finite set of labels and and τ^{r} a record type. The set of all object types is denoted as $\mathcal{T}^{\mathsf{o}}$.

Definition 7 (Object conformance)**.** Let $G = (N, E, \rho, \lambda, \pi)$ be a property graph and $\tau^{\mathsf{o}} = (L, \tau^{\mathsf{r}})$ an object type. The set of objects that *conform* to τ^{o} is defined as $[\![\tau^{\mathsf{o}}]\!] = \{o \in N \cup E \mid \lambda(o) = L \wedge \pi(o) \in [\![\tau^{\mathsf{r}}]\!]\}$.

Definition 8 (Basic property graph schema)**.** A *property graph schema* is a tuple

$$S = (N, E, \rho, \omega)$$

where

- N is a finite set of schema nodes;
- E is a finite set of schema edges such that $N \cap E = \emptyset$;
- $\rho : E \to (N \times N)$ is a total function mapping schema edges to ordered pairs of schema nodes;
- $\omega : (N \cup E) \to \mathcal{T}^{\mathsf{o}}$ is a total function mapping schema objects to object types.

[1] Note the subtle difference in terminology: an ER attribute has a value, whereas a property has a name and a value.

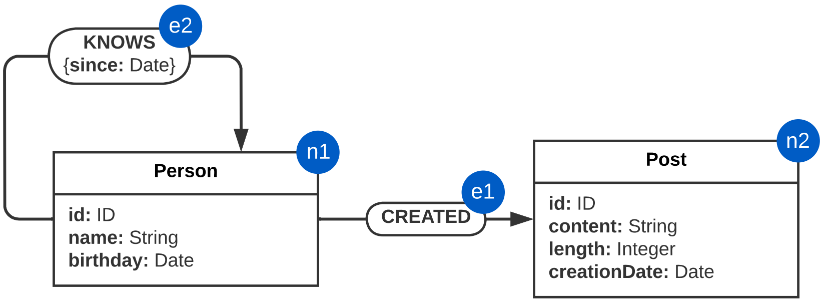


Figure 4.1: A basic property graph schema.

An example of a basic property graph schema is given in Figure 4.1. The similarity between property graphs and schemas allows us to visualize and think about them using the same mental model. Schema nodes and edges are drawn in the same way as data graphs, but properties are associated with types rather than concrete values.

Note that a property graph schema can be simulated by a property graph if we allow properties to take property types as values, i.e. $\mathcal{T} \subseteq \mathcal{V}$. Then we let λ and π take the role of ω: for all schema objects o in the property graph schema, if $\omega(o) = (L, \tau^{\mathsf{r}})$ then $\lambda(o) = L$ and $\pi(o) = \tau^{\mathsf{r}}$.

The example of Figure 4.1 can be mapped to our basic property graph schema model in the following way:

$$
\begin{aligned}
&N = \{n_1, n_2\}\\
&E = \{e_1, e_2\}\\
&\rho(e1) = (n_1, n_2) \quad \rho(e_2) = (n_1, n_1)\\
&\omega(n_1) = (\{\texttt{Person}\}, \langle \texttt{id} : \texttt{ID},\ \texttt{name} : \texttt{String},\ \texttt{birthday} : \texttt{Date}\rangle)\\
&\omega(n_2) = (\{\texttt{Post}\}, \langle \texttt{id} : \texttt{ID},\ \texttt{content} : \texttt{String},\ \texttt{length} : \texttt{Integer},\ \texttt{creationDate} : \texttt{Date}\rangle)\\
&\omega(e_1) = (\{\texttt{CREATED}\}, \langle\rangle) \quad \omega(e_2) = (\{\texttt{KNOWS}\}, \langle \texttt{since} : \texttt{Date}\rangle)
\end{aligned}
$$

Here, we use `ID`, `String`, `Date`, and `Integer` to denote property types. Their semantics may depend on the specific implementation. For example, $[\![\texttt{ID}]\!]$ could be the set of all UUIDs[2], and $[\![\texttt{Integer}]\!]$ could be the set of all 32-bit integers.

We can relate our schema formalism to the ER model once again. Our definitions of schema nodes and schema edges are analogous to the concepts of *entity set* and *relationship set*, respectively. Entity sets and schema nodes represent classes of entities that have something in common, whereas relationship sets and schema edges represent classes of relationships. Table 4.2 summarizes the relationship between the ER model and our schema formalism.

To make it easier to reason about objects conforming to schema objects, we introduce the *conformance relation*.

Definition 9 (Conformance relation)**.** Given a property graph

$$G = (N, E, \rho, \lambda, \pi)$$

[2] https://www.rfc-editor.org/rfc/rfc4122

ER	PG Schema
Entity set	Schema node
Relationship set	Schema edge
Value set	Property type*

Table 4.2: Mapping between entity–relationship (ER) and property graph (PG) schema concepts. $*$: strictly speaking, a property type τ is not a set, but $[\![\tau]\!]$ is.

and a property graph schema

$$S = (N', E', \rho', \omega)$$

we define the binary *conformance relation*

$$\sqsubseteq\ = \{(o, o') \in (N \cup E) \times (N' \cup E') \mid o \in [\![\omega(o')]\!]\}$$

We say that an object o *conforms* to a schema object o' if and only if $o \sqsubseteq o'$.

Finally, we define what it means for a property graph to *conform* to a schema.

Definition 10 (Basic schema conformance)**.** Given a property graph

$$G = (N, E, \rho, \lambda, \pi)$$

and a property graph schema

$$S = (N', E', \rho', \omega)$$

we say that G *conforms* to S if and only if all of the following rules hold.

1. Every node n conforms to some schema node n':

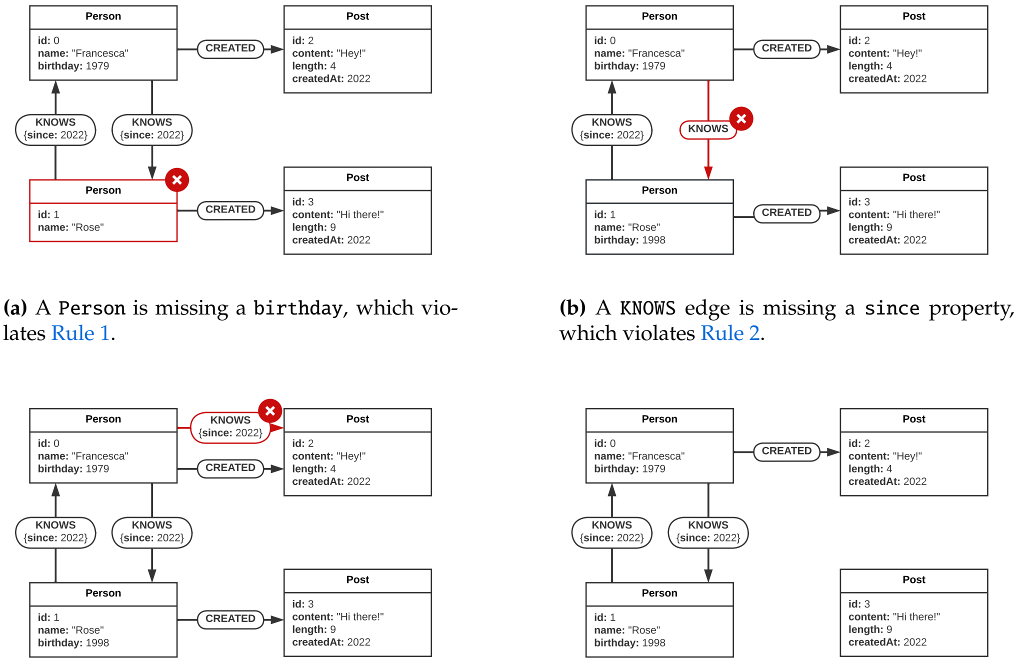
$$\forall n \in N\ \exists n' \in N' : n \sqsubseteq n'$$

2. Every edge e conforms to some schema edge e', and the source and target nodes of e conform to the respective endpoints of e':

$$\begin{aligned}&\forall e \in E\ \exists e' \in E' : \\ &\quad e \sqsubseteq e' \wedge \mathtt{src}(\rho(e)) \sqsubseteq \mathtt{src}(\rho'(e')) \wedge \mathtt{trg}(\rho(e)) \sqsubseteq \mathtt{trg}(\rho'(e'))\end{aligned}$$

Intuitively, Rule 1 specifies the types of nodes that are allowed to exist in the graph. If there exists a node in the graph that is not specified in the schema, the graph does not conform. Rule 2 similarly specifies the allowed types of edges. In contrast to Rule 1, it looks not only at the properties of the edge itself, but also at the source and target nodes. This prevents a node from having an edge that is not explicitly allowed, even if that edge itself does conform to some schema edge.

Figure 4.2 contains some examples of property graphs which are validated against the schema of Figure 4.1. In particular, Figure 4.2d shows a case that we might want to prevent (a `Post` with no creator), but the current schema formalism cannot express this. In general, it is not possible to specify that an edge is mandatory under these definitions. In the next subsection, we introduce a notion of cardinality constraints which makes this possible.



(a) A `Person` is missing a `birthday`, which violates Rule 1.

(b) A `KNOWS` edge is missing a `since` property, which violates Rule 2.

(c) The target of a `KNOWS` edge is not a `Person`, which violates Rule 2.

(d) Conforms to the schema. A `Post` is missing an incoming `CREATED` edge, but this does not violate any rules.

Figure 4.2: Examples of property graphs validated against the schema of Figure 4.1. Violating nodes and edges are marked and have a red outline.

4.2 Cardinality Constraints

We first introduce a generalization of the existential quantifier which enables counting the number of distinct variables that satisfy a predicate.

Definition 11 (Counting quantifier)**.** The *counting quantifier* is defined as follows. Given two numbers $n, m \in \mathbb{N}$, a predicate P, and a set X, define

- $\exists^{\geq n} x \in X : P(x) \equiv \exists x_1, \ldots, x_n \in X : P(x_1) \wedge \ldots \wedge P(x_n) \wedge \forall 1 \leq i < j \leq n : x_i \neq x_j$;
- $\exists^{\leq n} x \in X : P(x) \equiv \exists x_1, \ldots, x_n, x_{n+1}, \ldots, x_k \in X : P(x_1) \wedge \ldots \wedge P(x_k) \implies \forall n < i < j \leq k : x_i = x_j$;
- $\exists^{[n,m]} x \in X : P(x) \equiv \exists^{\geq n} x \in X : P(x) \wedge \exists^{\leq m} x' \in X : P(x')$;
- $\exists^{[n,*]} x \in X : P(x) \equiv \exists^{\geq n} x \in X : P(x)$.

Here, the $\equiv$ operator denotes logical equivalence.

Next, we introduce the notion of a *cardinality constraint*.

Definition 12 (Cardinality constraint)**.** A *cardinality constraint* is an ordered pair of intervals $([n_1, m_1], [n_2, m_2])$ where $n_1, n_2 \in \mathbb{N}$ and $m_1, m_2 \in \mathbb{N}^*$ with $\mathbb{N} = \{0, 1, 2, \ldots\}$ and $\mathbb{N}^* = \mathbb{N} \cup \{*\}$. The set of all cardinality constraints is denoted as C.

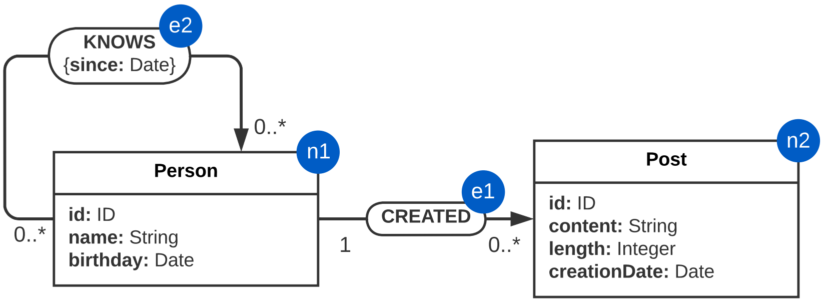


Figure 4.3: A property graph schema with cardinality constraints.

The two intervals of a cardinality constraint apply to the source and target of an edge, respectively. We also use the functions `src` and `trg` to refer to the first and second interval of a cardinality constraint, i.e. $\texttt{src}([n_1, m_1], [n_2, m_2]) = [n_1, m_1]$ and $\texttt{trg}([n_1, m_1], [n_2, m_2]) = [n_2, m_2]$.

Next, we revise the definitions of property graph schema and schema conformance, making use of our newly defined cardinality constraints. The following definitions subsume Definition 8 and 10.

Definition 13 (Property graph schema)**.** A *property graph schema* is a tuple

$$S = (N, E, \rho, \omega, \eta)$$

where

- N is a finite set of schema nodes;
- E is a finite set of schema edges such that $N \cap E = \emptyset$;
- $\rho : E \to (N \times N)$ is a total function mapping schema edges to ordered pairs of schema nodes;
- $\omega : (N \cup E) \to \mathcal{T}^{\text{o}}$ is a total function mapping schema objects to object types;
- $\eta : E \to C$ is a total function mapping schema edges to cardinality constraints.

An example of a property graph schema with cardinality constraints is given in Figure 4.3. Intervals such as $[n, m]$ are written as $n..m$, and $[n, n]$ is written simply as n, following notation established in UML [ISO/IEC 19501:2005, 2005]. Furthermore, we use the "look-across" notation (as opposed to "look-here"), meaning that the interval indicates the minimum and maximum number of edges that the node on the other side of the edge must participate in. Contrary to some data modeling diagram conventions, the arrow head is unrelated to cardinality, instead it specifies the direction of an edge. Note that every property graph that conforms to Figure 4.1 also conforms to Figure 4.3, but not the other way around.

The example of Figure 4.3 can be mapped to our property graph schema model in the following way:

$N = \{n_1, n_2\}$
$E = \{e_1, e_2\}$
$\rho(e1) = (n_1, n_2) \quad \rho(e_2) = (n_1, n_1)$
$\omega(n_1) = (\{\texttt{Person}\}, \langle \texttt{id} : \texttt{ID},\ \texttt{name} : \texttt{String},\ \texttt{birthday} : \texttt{Date} \rangle)$
$\omega(n_2) = (\{\texttt{Post}\}, \langle \texttt{id} : \texttt{ID},\ \texttt{content} : \texttt{String},\ \texttt{length} : \texttt{Integer},\ \texttt{creationDate} : \texttt{Date} \rangle)$
$\omega(e_1) = (\{\texttt{CREATED}\}, \langle\rangle) \quad \omega(e_2) = (\{\texttt{KNOWS}\}, \langle \texttt{since} : \texttt{Date} \rangle)$
$\eta(e_1) = ([1,1],[0,*]) \quad \eta(e_2) = ([0,*],[0,*])$

Next, we redefine schema conformance, taking cardinality constraints into account. This definition subsumes Definition 10.

Definition 14 (Schema conformance)**.** Given a property graph

$$G = (N, E, \rho, \lambda, \pi)$$

and a property graph schema

$$S = (N', E', \rho', \omega, \eta)$$

we say that G *conforms* to S if and only if all of the following rules hold.

1. Every node n conforms to some schema node n':

$$\forall n \in N\ \exists n' \in N' : n \sqsubseteq n'$$

2. Every edge e conforms to some schema edge e', and the source and target nodes of e conform to the respective endpoints of e':

$$\begin{aligned} &\forall e \in E\ \exists e' \in E' : \\ &\quad e \sqsubseteq e' \land \texttt{src}(\rho(e)) \sqsubseteq \texttt{src}(\rho'(e')) \land \texttt{trg}(\rho(e)) \sqsubseteq \texttt{trg}(\rho'(e')) \end{aligned}$$

3. If a node n conforms to the source of a schema edge e', it must have the right number of outgoing edges of the right type:

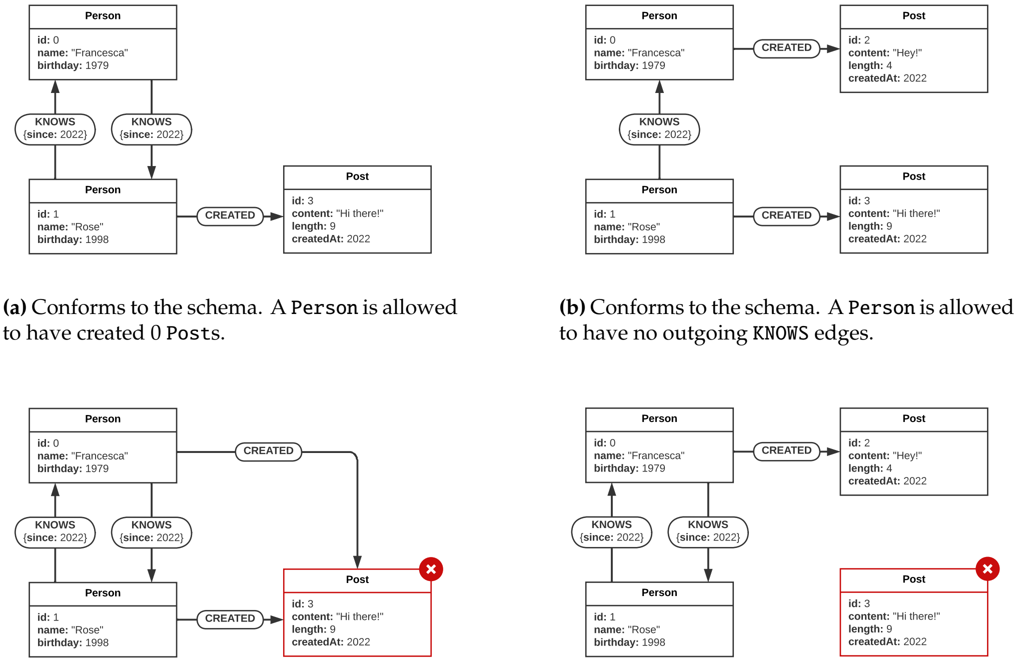
$$\begin{aligned} &\forall n \in N\ \forall e' \in E' : \\ &\quad \Big[n \sqsubseteq \texttt{src}(\rho(e')) \implies \exists^{\texttt{trg}(\eta(e'))} e \in E : \\ &\qquad e \sqsubseteq e' \land \texttt{src}(\rho(e)) = n \land \texttt{trg}(\rho(e)) \sqsubseteq \texttt{trg}(\rho(e')) \Big] \end{aligned}$$

4. If a node n conforms to the target of a schema edge e', it must have the right number of incoming edges of the right type:

$$\begin{aligned} &\forall n \in N\ \forall e' \in E' : \\ &\quad \Big[n \sqsubseteq \texttt{trg}(\rho(e')) \implies \exists^{\texttt{src}(\eta(e'))} e \in E : \\ &\qquad e \sqsubseteq e' \land \texttt{src}(\rho(e)) \sqsubseteq \texttt{src}(\rho(e')) \land \texttt{trg}(\rho(e)) = n \Big] \end{aligned}$$

Compared to Definition 10, Rule 1 and 2 are unchanged, while Rule 3 and 4 enforce cardinality constraints on edges. With these new rules, we can enforce mandatory edges, i.e. a cardinality constraint of "at least one".

Figure 4.4 contains some examples of property graphs which are validated against the schema of Figure 4.3, taking into account the new rules.



(a) Conforms to the schema. A `Person` is allowed to have created 0 `Posts`.

(b) Conforms to the schema. A `Person` is allowed to have no outgoing `KNOWS` edges.

(c) This `Post` has too many incoming `CREATED` edges, which violates Rule 4.

(d) A `Post` is missing an incoming `CREATED` edge, which violates Rule 4.

Figure 4.4: Examples of property graphs validated against the schema of Figure 4.3. Violating nodes and edges are marked and have a red outline.

4.3 Optional Properties

We revise the definition of record by adding a special property value **undef**, which indicates that the value of a property is not defined. The following definitions subsume Definition 1, 3, 4, and 5.

Definition 15 (Record)**.** A *record* is a total function $r : \mathcal{N} \to \mathcal{V} \cup \{\textbf{undef}\}$ that maps property names to property values or the special value **undef**. We denote such records as $\langle n_1 : v_1, \ldots, n_k : v_k \rangle$. The set of all records is denoted as $\mathcal{R}$.

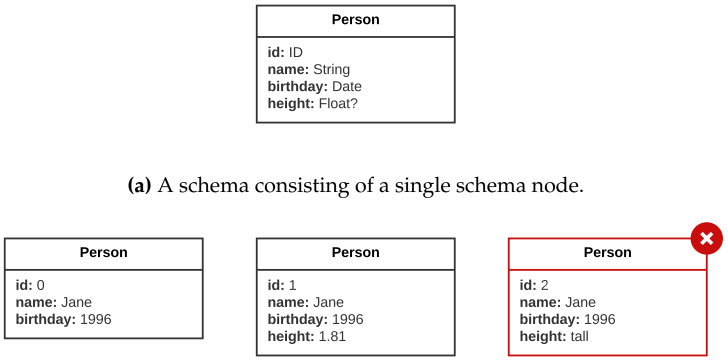
For a record r and a property name $k \in \mathcal{N}$ such that $r(k)$ was previously undefined, we now say $r(k) = \textbf{undef}$. This can be seen as the "default" value of a property.

We adjust the definitions of property conformance, record type, and record conformance accordingly.

Definition 16 (Property conformance)**.** For each property type $\tau \in \mathcal{T}$ there is a set $[\![\tau]\!] \subseteq \mathcal{V} \cup \{\textbf{undef}\}$ that contains all property values that *conform* to the type τ. We say τ is *optional* iff $\textbf{undef} \in [\![\tau]\!]$. We use the notation $\tau?$ to mark a property as optional, i.e. $[\![\tau?]\!] = [\![\tau]\!] \cup \{\textbf{undef}\}$.

Definition 17 (Record type)**.** A *record type* is a total function $\tau^{\mathsf{r}} : \mathcal{N} \to \mathcal{T}$ that maps property names to property types. We denote such record types as $\langle n_1 : \tau_1, \ldots, n_k : \tau_k \rangle$.

Definition 18 (Record conformance)**.** We say that a record r *conforms* to a record type τ^{r}, denoted $r \in [\![\tau^{\mathsf{r}}]\!]$, if and only if for each property name $k \in \mathcal{N}$ it holds that $r(k) \in [\![\tau^{\mathsf{r}}(k)]\!]$.



(a) A schema consisting of a single schema node.

(b) A property graph consisting of three disconnected nodes. Properties with a value of **undef** are not shown. The node with `id : 2` violates Rule 1 because the `height` property value is of the wrong type.

Figure 4.5: A property graph validated against a schema with an optional property.

An example of a schema containing an optional property is shown in Figure 4.5. This example illustrates that optional properties may be omitted, but if they are present, they must have a value of the right type.

4.4 Open Record Types

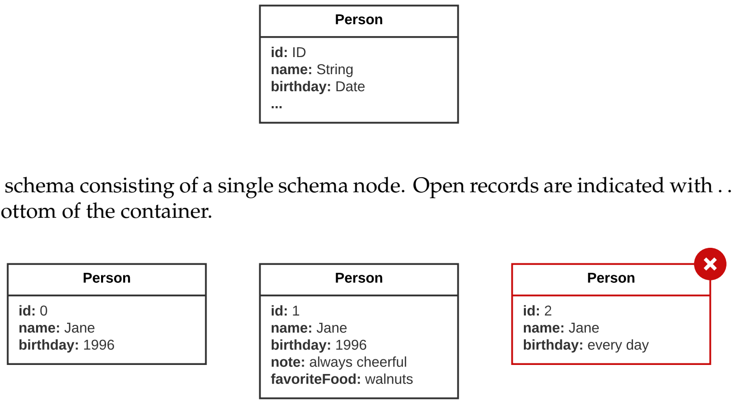
It may be desirable to allow a record to have properties with any name or any value. Such *open record types* can already be expressed under the current definitions. For a record type τ^{r}, we can allow properties with any name by setting $\tau^{\mathsf{r}}(k) = \tau?$ for all (previously unspecified) property names $k \in \mathcal{N}$, where τ is an arbitrary property type. Note that there may be infinitely many such k, so in practice we may want to have a special syntax to express this. We can allow these properties to have any value by choosing a τ such that $[\![\tau]\!] = \mathcal{V}$, or we can restrict them to a subset of $\mathcal{V}$.

An example of a schema containing an open record type is shown in Figure 4.6. An open record type allows properties with any name, but still requires explicitly defined properties to be of the right type.

4.5 Mapping Functionality to Rules

In Chapter 2, we compared the capabilities of existing property graph schema implementation. We can now see how most of the features in Table 2.2 can be mapped to one of our conformance rules.

Mandatory and allowed properties are captured by our notion of record conformance (Definition 18), which corresponds to Rule 1. Endpoint constraints correspond to Rule 2. Data type constraints are captured by our notion of object conformance (Definition 7) and correspond to Rule 1. Cardinality constraints correspond to Rule 3 and 4. Property uniqueness, label coexistence, and subtype relations are not covered by our formalism.



(a) A schema consisting of a single schema node. Open records are indicated with . . . at the bottom of the container.

(b) A property graph consisting of three disconnected nodes. The node with `id : 2` violates Rule 1 because the `birthday` property value is of the wrong type.

Figure 4.6: A property graph validated against a schema with an open record type.

Chapter 5

Schema Validation in Practice

Our property graph schema formalism is capable of expressing constraints in common conceptual data modeling methods. However, we have found that existing property graph database systems are limited in the kinds of constraints they can enforce, in particular cardinality constraints.

In this chapter, we describe how existing database systems can be extended to support all kinds of constraints that can be expressed with our schema formalism. Where possible, we make use of the schema functionality exposed by the database engine. For constraints that cannot be validated in this way, we provide graph queries.

We focus here on the same three databases we covered in Chapter 2: *Neo4j*, *JanusGraph*, and *TigerGraph*. This choice is motivated by the diversity of schema approaches employed by these systems. Neo4j's schemas are mostly inferred from data, whereas TigerGraph requires an explicit schema, and JanusGraph is schema-optional.

5.1 Assumptions

To reduce the complexity of the implementation, we assume that the set of labels of a schema object functionally determines the record type of that schema object. Recall from Chapter 4 that every object must conform to a schema object (Rule 1 and 2). This implies that if an object has the same set of labels as a schema object, it must conform to that schema object (since there exists no schema object with the same label set and a different record type). We find that this assumption holds in many domains, including the datasets we study in Chapter 6.

Under this assumption, we can simplify the conformance rules. Informally:

1. (a) Every node has the same set of labels as some schema node.
   (b) All mandatory properties exist on the node.
   (c) The node has no properties that are not allowed.
   (d) All node properties have a value of the right type.
2. (a) Every edge has the same set of labels as some schema edge.
   (b) All mandatory properties exist on the edge.
   (c) The edge has no properties that are not allowed.

(d) All edge properties have a value of the right type.

(e) Both endpoints of the edge have the right labels.

3. Every node has the right number of outgoing edges with the right labels.
4. Every node has the right number of incoming edges with the right labels.

5.2 Validation Variants

We consider two variants of the schema validation problem:

- **Binary validation:** given a graph and a schema, determine whether the graph conforms to the schema or not.
- **Full validation:** given a graph and a schema, find all graph objects which cause a violation of at least one of the rules of schema conformance.

While a binary validation method may be sufficient for some use cases, full validation is required when we want to explain why a graph does not conform. Binary validation is a strictly easier problem because we can stop as soon as we find a single violation, while full validation always needs to find all violations in the graph.

5.3 Implementation

We describe how each conformance rule can be validated using the three databases in our scope. Both the binary and full validation variants are discussed. The full source code is available on GitHub[1].

Neo4j. We start by using the built-in schema functionality that Neo4j offers. To ensure the existence of mandatory properties (Rule 1b and 2b), we use Neo4j's `CONSTRAINT` functionality, which lets us specify which properties are mandatory for nodes and edges with a particular label set (Enterprise edition only). An example is given in Listing 5.1.

```
CREATE CONSTRAINT FOR (m:Movie) REQUIRE m.imdbId IS NOT NULL;
CREATE CONSTRAINT FOR (m:Movie) REQUIRE m.movieId IS NOT NULL;
CREATE CONSTRAINT FOR (m:Movie) REQUIRE m.title IS NOT NULL;
```

Listing 5.1: A set of Cypher statements to create constraints which require the existence of some mandatory properties on `Movie` nodes.

Next, we tackle the validation of the node and edge labels (Rule 1a and 2a) using Cypher queries. Neo4j provides functions `db.labels()` and `db.relationshipTypes()`, which respectively return a list of all node and edge labels that exist in the database. Unfortunately, these functions do not tell us which *combinations* of labels exist. Since nodes can have multiple labels, we must fall back to a `MATCH` query which checks that every node has one of the allowed combinations of labels (see Listing 5.2). Since edges in Neo4j can only have one label (called a *relationship type*), we can use `db.relationshipTypes()` to determine if there exist any edges with a label that is not allowed (see Listing 5.3). This is sufficient for binary validation, but if we want to find out precisely which edges have an illegal label, we again need to use a `MATCH` query.

[1] https://github.com/nimobeeren/thesis

```
WITH [["Actor", "Person"], ["Movie"], ["User"], ...] AS
    allowedNodeLabelSets
MATCH (n)
WHERE NOT labels(n) IN allowedNodeLabelSets
RETURN n
```

Listing 5.2: A Cypher query to find all nodes that have a label set that is not allowed.

```
WITH ["ACTED_IN", "DIRECTED", ...] AS allowedEdgeLabels
CALL db.relationshipTypes() YIELD relationshipType AS allTypes
RETURN all(type IN collect(allTypes) WHERE type IN
    allowedEdgeLabels)
```

Listing 5.3: A Cypher query to check if there are any edge labels that are not allowed. Note that this query returns a Boolean and does not explain which edges have an illegal label.

Next, we look at the properties. The functions `db.schema.nodeTypeProperties()` and `db.schema.relTypeProperties()` provide useful information about the properties that exist on nodes and edges, as well as their data types. This allows us to determine if there are any nodes or edges that have properties that are not allowed (Rule 1c and 2c) or have the wrong data type (Rule 1d and 2d). See Listing 5.4 for an example of such a query. Again, if we want to find the violating objects, we need to use a `MATCH` query that scans all properties of all objects in the database. In that case, we use the `apoc.meta.type()` function (provided by the APOC library[2]) to get the data type of a property at query-time. Listing 5.5 shows an example.

```
CALL db.schema.nodeTypeProperties() YIELD nodeLabels,
    propertyName, propertyTypes
WHERE "User" IN nodeLabels AND (
    NOT propertyName IN ["userId", "name"]
    OR propertyName = "userId" AND propertyTypes <> ["String"]
    OR propertyName = "name" AND propertyTypes <> ["String"]
)
RETURN count(nodeLabels) = 0
```

Listing 5.4: A Cypher query to check if there are any `User` nodes with properties that are not allowed or have the wrong data type. If the result is `true`, there are no violations.

```
WITH {
    userId: "STRING",
    name: "STRING"
} AS propertyTypes
MATCH (n:User)
WHERE NOT all(pKey IN keys(n) WHERE pKey IN keys(propertyTypes)
    AND apoc.meta.type(n[pKey]) = propertyTypes[pKey])
RETURN n
```

Listing 5.5: A Cypher query to find all `User` nodes with properties that are not allowed or have the wrong data type.

[2] https://neo4j.com/labs/apoc/

To check Rule 2e, we do a MATCH query for every edge label and look at the source and target nodes. For every edge, there must exist a schema edge such that the labels of the source and target in the data match the labels of the source and target in the schema. An example of such a query is shown in Listing 5.6.

```
MATCH (n)-[e:ACTED_IN]->(m)
WHERE NOT "Actor" IN labels(n) OR NOT labels(m) = ["Movie"]
RETURN e;
```

Listing 5.6: A Cypher query that finds all ACTED_IN edges where the endpoints have the wrong label. Note that the source node n may have additional labels.

Finally, we check the cardinality constraints (Rule 3 and 4). Since the schemas of our datasets only contain "one or more" constraints, a simple MATCH query for each relevant node label is sufficient. See Listing 5.7 for an example.

```
MATCH (a:Actor)
WHERE NOT (a)-[:ACTED_IN]->(:Movie)
RETURN a;
```

Listing 5.7: A Cypher query to find ACTOR nodes that are missing a mandatory ACTED_IN edge.

To give the database engine the best opportunity to perform query optimization, the queries described here are merged as much as possible. These merged queries are then executed sequentially. This is particularly relevant for the binary variant, since it can terminate as soon as a single violation is found. For MATCH queries, we always return a value using `count(variable) = 0`. This theoretically allows the query to terminate early, but it is up to the query engine to implement this optimization.

JanusGraph. The extensive set of schema features that JanusGraph provides allow us to specify most conformance rules at the time of schema definition, preventing any non-conforming data from being inserted. The complete set of allowed node and edge labels (Rule 1a and 2a) are defined using `makeVertexLabel()` and `makeEdgeLabel()`. Because JanusGraph only allows a single label on an object, we concatenate the labels if more than one is present in the data. This does not affect the semantics (as long as the order of concatenation is consistent), since we only ever check for equality of label sets (see Definition 7).

For each edge label, the method `addConnection()` lets us specify the allowed labels of the edge endpoints (Rule 2e). The maximum cardinality for an edge label is set with `multiplicity()`, but this does not provide any guarantees on minimum cardinality.

The complete set of allowed property names is defined using `makePropertyKey()`. Next, each label is mapped to a set of property names using `addProperties()`. This prevents objects from having properties that are not allowed (Rule 1c and 2c). The data type of each property is defined using `dataType()` (Rule 1d and 2d). In JanusGraph, there cannot exist two properties with the same name and different data type. If this is desired, the property name must be disambiguated, which could be achieved by prefixing it with the object label.

For an example of all of JanusGraph's built-in schema functions working together, have a look at Listing 5.8.

```
// Vertex labels
VertexLabel Actor = mgmt.makeVertexLabel("Actor").make();
VertexLabel Movie = mgmt.makeVertexLabel("Movie").make();
// Edge labels
EdgeLabel ACTED_IN = mgmt.makeEdgeLabel("ACTED_IN").
    multiplicity(Multiplicity.SIMPLE).make();
// Edge connections
mgmt.addConnection(ACTED_IN, Actor, Movie);
// Property keys
PropertyKey nameKey = mgmt.makePropertyKey("name").dataType(
    String.class).make();
PropertyKey titleKey = mgmt.makePropertyKey("title").dataType(
    String.class).make();
PropertyKey revenueKey = mgmt.makePropertyKey("revenue").
    dataType(Long.class).make();
PropertyKey roleKey = mgmt.makePropertyKey("role").dataType(
    String.class).make();
// Property mapping
mgmt.addProperties(Actor, nameKey);
mgmt.addProperties(Movie, titleKey, revenueKey);
mgmt.addProperties(ACTED_IN, roleKey);
```

Listing 5.8: A fragment of the Recommendations schema, expressed using JanusGraph's schema methods. Mandatory properties and cardinality constraints cannot be validated in this way. The `Multiplicity.SIMPLE` configuration ensures there is no more than one `ACTED_IN` edge between any given actor and movie.

The remaining rules cannot be fully enforced by JanusGraph's schema, so we write queries to achieve this. Missing mandatory properties (Rule 1b and 2b) are found using a combination of `hasLabel` and `hasNot` steps of the Gremlin query language (see Listing 5.9). For cardinality constraints, we leverage the fact that our schemas only have constraints of the form "one or more". We can validate these constraints using a combination of `hasLabel`, `outE`, and `inE` steps (see Listing 5.10).

```
g.V().hasLabel("User").or(hasNot("name"), hasNot("userId"))
```

Listing 5.9: A Gremlin query to find `User` nodes which are missing a mandatory `name` or `userId` property.

```
g.V().hasLabel(P.within("Actor", "ActorDirector"))
    .not(outE("ACTED_IN"))
```

Listing 5.10: A Gremlin query to find `Actor` nodes which are missing an outgoing `ACTED_IN` edge. Note that `ActorDirector` is the concatenation of the `Actor` and `Director` labels.

If we are only interested in the binary validation problem, we only need to determine whether the query result is empty. This is achieved using Java's hasNext() iterator method, which tells the query engine to stop traversing the graph as soon as one result has been found. To give the query engine the best chance of terminating early, we write the entire validation as a single disjunctive query. JanusGraph unfortunately does not allow a single query to range over all nodes and all edges, so instead we execute one query for nodes and one for edges, in sequence.

TigerGraph. Before any data is loaded, TigerGraph requires specification of a schema describing all nodes, edges and their properties. This is done with a series of CREATE VERTEX and CREATE EDGE statements. Using these built-in schema capabilities, we can already guarantee conformance to some of our rules. Every node and edge has a label that appears in the schema (Rule 1a and 2a) and has a fixed set of properties associated with that label (Rule 1c and 2c). Every property has a fixed data type (Rule 1d and 2d). Furthermore, every node type has a primary key. Finally, every edge type must specify a source and target node types (Rule 2e).

```
CREATE VERTEX Actor (
    PRIMARY_ID id STRING,
    name STRING
)
CREATE VERTEX Movie (
    PRIMARY_ID id STRING,
    title STRING,
    revenue INT
)
CREATE DIRECTED EDGE ACTED_IN (
    FROM Actor,
    TO Movie,
    role STRING
)
```

Listing 5.11: A fragment of the Recommendations schema, expressed in TigerGraph's schema definition language.

In TigerGraph, all properties must have a value as soon as the object is instantiated (there are no null values). When a value is missing, it is set to a default value, such as the empty string, the number 0, or the first of January 1970. While it is possible to prevent objects with missing values from being loaded, in some cases we may want to check for missing values in an existing database. To this end, we use a query to find all objects which have a mandatory property with the default value (validating Rule 1b and 2b). If the default value naturally appears in valid data, then we use another indicator value. The query consists of a set of simple SELECT-FROM-WHERE patterns which range over all nodes and edges, as in Listing 5.12.

```
SetAccum<VERTEX> @@violatingNodes;

violatingMovies =
    SELECT movie
    FROM Movie:movie
    WHERE movie.imdbId == "" OR movie.movieId == ""
        OR movie.title == "";
    ACCUM @@violatingNodes += movie;

violatingUsers =
    SELECT user
    FROM User:user
    WHERE user.name == "" OR user.userId == "";
    ACCUM @@violatingNodes += user;

PRINT @@violatingNodes;
```

Listing 5.12: A GSQL query to find `Movie` and `User` nodes which have a mandatory property with a default value. All violating nodes are collected in the accumulator variable `@@violatingNodes`, which is returned at the end.

To cover the final remaining rules, cardinality constraints are validated using another query. For outgoing edges (Rule 3), we use the built-in `outdegree()` function, which returns the number of outgoing edges with a particular label for a given node (see Listing 5.13). By default, TigerGraph maintains an index that stores these statistics for every node. For incoming edges (Rule 4), we use another TigerGraph feature known as an accumulator. This lets us compute the number of incoming edges for all relevant nodes in a single pass over the graph. Afterwards, the violating nodes are picked out (see Listing 5.14).

```
SELECT actor
FROM Actor:actor
WHERE actor.outdegree("ACTED_IN") == 0
```

Listing 5.13: A GSQL query to find `Actor` nodes which are missing an outgoing `ACTED_IN` edge.

```
SumAccum<int> @numContainerOf;
tmp =
    SELECT post
    FROM :n -(CONTAINER_OF>)- Post:post
    ACCUM post.@numContainerOf += 1;
violatingActors =
    SELECT post
    FROM Post:post
    WHERE post.@numContainerOf == 0
```

Listing 5.14: A GSQL query to find `Post` nodes which are missing an incoming `CONTAINER_OF` edge.

All TigerGraph queries are *installed* before execution, which allows the query engine to prepare various optimizations for efficient execution. In the binary variant queries, we always return something of the form `@@accumulator.size() == 0`, which indicates that the query could terminate before iterating over the entire graph. Again, it is up to the query engine to recognize this and optimize accordingly.

Chapter 6

Empirical Evaluation

In this chapter, we evaluate the performance of our prototypical implementation through a controlled experiment. We evaluate the three databases we have focused on throughout this thesis: Neo4j, JanusGraph, and TigerGraph. The presented results may help the reader to determine whether our approach is feasible for their use case.

6.1 Research Questions

To explore the cost of validating property graph schema conformance using currently available graph database systems, we aim to answer the following research questions:

RQ1 How does validation time differ between the binary and full validation variants?

RQ2 How is validation time related to the scale of the data?

RQ3 How is validation time related to the amount of schema violations?

RQ4 How does validation time differ between database systems?

6.2 Methodology

To answer our research questions, we perform a set of experiments which test the performance of our implementation described in Chapter 5 in various realistic scenarios. For each database, we apply the binary and full validation methods to datasets with different schemas and scales, and we investigate the effect of violations in the data. For each workload, we perform three subsequent runs, from which we report the average. The next subsections describe the performance-related factors we considered.

6.2.1 Dependent Variable

Validation time. We are interested in the time it takes to validate conformance of a graph to a schema. For Neo4j and TigerGraph, we measure this using the execution time as listed in the internal query logs. Even though we depend on the logs to provide a fair and accurate measurement, we believe this method is best because it ignores irrelevant

Dataset	|N|	|E|	|N′|	|E′|	Raw (MB)
Recommendations	28,863	166,261	6	6	22
SNB (SF0.1)	327,588	1,477,965	14	20	100
SNB (SF0.3)	908,224	4,583,118	14	20	295
SNB (SF1)	3,181,724	17,256,038	14	20	1109

Table 6.1: Size statistics of the datasets used in our experiments. $|N|$ and $|E|$ represent the number of nodes and edges. $|N'|$ and $|E'|$ represent the number of schema nodes and schema edges. The raw size is measured as the combined size of all CSV files from which the data is loaded.

things, such as the time taken to print the results. JanusGraph does not provide such logs, so we use Java's `System.currentTimeMillis()` method, which reports the wall-clock time. If multiple queries are needed to produce an answer, we take the sum of their execution times.

6.2.2 Independent Variables

Database. The different decisions regarding schemas made by our three databases of interest may affect their performance on a schema validation workload. We use Neo4j Enterprise Edition 4.4.8 (supporting mandatory property constraints), JanusGraph 0.6.2, and TigerGraph 3.6.0.

Dataset. To determine the impact of different schemas and scales on the performance of our implementation, we apply our methods to two datasets:

- The **Recommendations Graph**[1] is a relatively small dataset with a simple schema, provided by Neo4j. It consists of real data sourced from *Open Movie Database*[2] and *MovieLens*[3]. It contains movies, actors, directors, and users who rate movies. The schema is given in Figure 6.1. This dataset is intended as a working example for people who want to explore the functionality of a graph database. The small size makes it suitable as a proof-of-concept for our schema validation methods.
- The **LDBC Social Network Benchmark** (SNB) [Angles et al., 2020] is a synthetic dataset, designed to evaluate graph-like data management technologies in a realistic setting. The dataset consists of users, messages, likes, and other concepts that can be found in the domain of social networks. The schema is of moderate size and is visualized in Figure 6.2. The data is made available at various scales, ranging from less than a gigabyte up to a terabyte of raw data. We run multiple workloads with scale factor 0.1, 0.3, and 1. This lets us analyze how our solution scales with the size of the data.

These datasets are diverse in scale, ranging from roughly 200K to 20M objects. However, they are all small enough to fit entirely in memory. This choice is motivated primarily by convenience, keeping validation times relatively short and allowing us to perform tests in many different scenarios. Some statistics regarding the size of the datasets are given in Table 6.1.

[1] https://github.com/neo4j-graph-examples/recommendations
[2] https://www.omdbapi.com/
[3] https://grouplens.org/datasets/movielens/

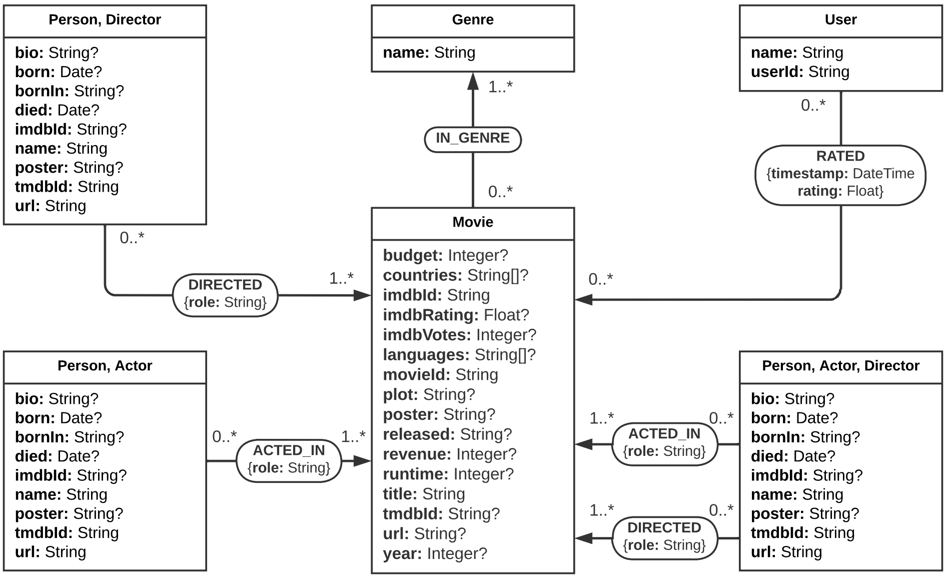


Figure 6.1: The schema of the Recommendations Graph dataset. This schema was constructed manually by analyzing the data.

Validation variant. As described in Section 5.2, we consider two variants of the validation problem: binary and full validation. For binary validation, we measure the time until the first schema violation is found (or until the last query has finished, if there are no violations). This is because a single violation is enough to determine that the data does not conform to the schema. For the full variant, we measure until the last query has finished, because all violations must be found.

Violation rate. We expect the validation time of the binary variant to be affected by the number of violations in the data. If the data graph conforms to the schema, the entire graph is scanned. However, as soon as a single violation is found, the validation process can terminate. The higher the violation rate, the sooner we expect to terminate.

We consider three levels of violation rate:

- *None:* the graph conforms to the schema.
- *Single:* a single node violates a constraint.
- *Many:* roughly 50% of all nodes violate a constraint.

The way violations are introduced depends on the database system under test, due to their differences in schema capabilities and data model. For JanusGraph, we remove a mandatory property from nodes. Because TigerGraph requires every property to have a value, we set it to the default value instead. Because Neo4j supports constraints that prevent missing mandatory properties, we remove a mandatory edge instead. Edges are not modified, hence they do not violate any constraints.

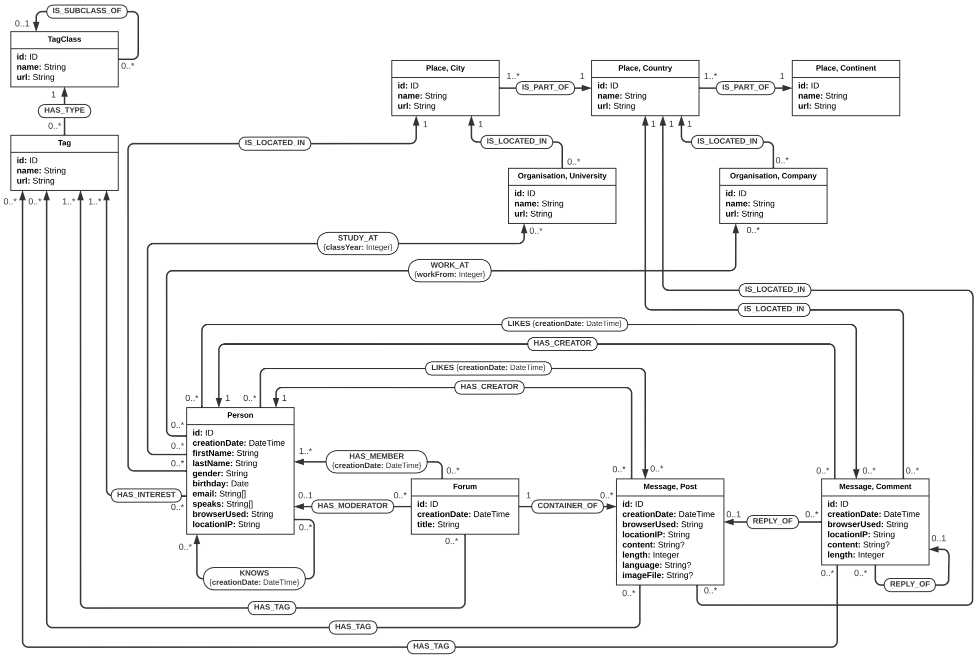


Figure 6.2: The schema of the SNB dataset. This schema was created by modifying the schema from Angles et al. [2020] to not rely on subtype relations. Because our schema formalism does not support subtyping, the resulting schema is more permissive than the original. This limitation is discussed in detail in Chapter 7.

The choice to only introduce violations on nodes is convenient, but we expect it to be sufficient to have a measurable effect. If we assume the validation process to be a random search through all objects, we can model it as a sequence of random trials with probability $1 - p$, where p is the proportion of objects that violate a constraint. Thus, the time until the first violation is detected shrinks exponentially when the number of violations in the data grows. Even though all of our datasets have more edges than nodes, introducing a violation in 50% of nodes will significantly increase p, as compared to a single violation or none at all.

6.2.3 Parameters

We control the following parameters so that they are the same for every workload.

Indexing and caching. To enable a fair comparison between databases, we attempt to put an equal amount of effort into optimization for each of them. We only use indexes that the database system creates by default. Most of these are not useful to us, because their purpose is to find sets of objects with specific labels or properties. The exception is TigerGraph's outdegree index (see Section 5.3), which may speed up the validation of cardinality constraints. To facilitate independence between subsequent runs, caching is disabled for Neo4j and JanusGraph. TigerGraph does not provide any caching functionality that we know of.

Hardware. All workloads are run on a single machine with 4 CPU cores, 6 gigabytes of dedicated memory, and a solid-state drive with enough capacity to store all data.

6.3 Results

In this section, we show the results of our experiments. We attempt to answer each research question using plots and statistical tests. The complete set of results is available on GitHub[4].

6.3.1 RQ1: Validation Variant

We run both the binary and full validation queries on all datasets. For these workloads, no violations are introduced. While we are also interested in the effect of violations on the binary variant, this is addressed in Subsection 6.3.3. We perform an independent two-sided T-test ["Student" Gosset, 1908] to test the null hypothesis that the mean validation time is the same for the full and binary variant, with all other variables fixed. The results for each database are shown in Figure 6.3 and Table 6.2. The queries for JanusGraph on the SNB dataset with scale factor 0.3 and larger did not complete within an hour, and are not shown here.

For Neo4j, we find a statistically significant difference ($p < 0.05$) between the full and binary variants, for all datasets except the Recommendations Graph. This difference could be explained by Neo4j's schema statistics, which are retrieved in constant time and eliminate the need for some of the queries in the full variant. However, on the smaller Recommendations dataset this advantage is no longer apparent, possibly because the constant lookup time outweighs the time saved by skipping some queries. We do not find evidence to suggest a difference between the two variants for JanusGraph and TigerGraph. This is expected, since the queries for both variants are very similar, and the binary variant cannot terminate early because the data conforms to the schema.

6.3.2 RQ2: Scale

To further investigate the relationship between validation time and the scale of the data, Figure 6.4 plots the validation time against the number of objects in the data graph. To keep the schema and structure constant, we only consider the SNB dataset, but at different scale factors. We do not plot the results for JanusGraph, because only the smallest scale factor could be successfully tested.

These results suggest a linear relationship between the number of objects and validation time. This conclusion is supported by fitting a simple linear regression model using the least squares method. We use the number of objects as the explanatory variable and the validation time as the response variable. The high R^2 values suggest a good fit. This linear relationship is expected, since our queries perform a linear scan of the nodes and edges in the data graph.

6.3.3 RQ3: Violation Rate

The workloads so far did not contain schema violations, meaning that the data conformed to the schema. To evaluate how schema violations affect validation time, we choose a fixed dataset and introduce some errors into the data. We choose the SNB dataset at SF0.1, because all databases could successfully complete full validation, and the results of RQ1 suggested there was more potential for improvement as compared to the Recommendations dataset. We only test the binary validation variant, because we

[4] https://github.com/nimobeeren/thesis/blob/main/analysis/results.csv

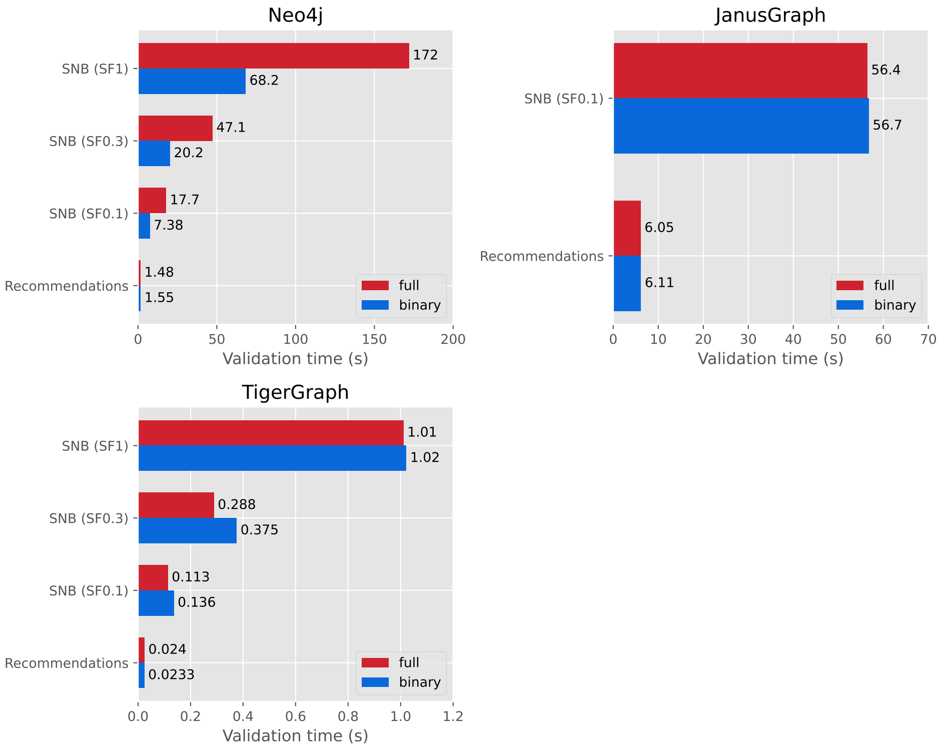


Figure 6.3: Mean validation time for binary and full variants on all datasets with no schema violations.

Database	**Dataset**	**Mean validation time (s)**		**t**	**p**
		Full	**Binary**		
Neo4j	SNB (SF1)	171.89	68.17	49.9	**<0.0001***
	SNB (SF0.3)	47.13	20.18	28.9	**<0.0001***
	SNB (SF0.1)	17.71	7.38	31.3	**<0.0001***
	Recommendations	1.48	1.55	-0.37	0.731
JanusGraph	SNB (SF0.1)	56.41	56.74	-0.56	0.608
	Recommendations	6.05	6.11	-0.42	0.697
TigerGraph	SNB (SF1)	1.01	1.02	-0.66	0.546
	SNB (SF0.3)	0.29	0.38	-1.34	0.252
	SNB (SF0.1)	0.11	0.14	-0.78	0.478
	Recommendations	0.02	0.02	0.32	0.768

Table 6.2: Mean validation time for binary and full variants on all datasets with no schema violations. A two-sided T-test is performed to test for a difference between the mean validation time for the full and binary variant. We give the *t-statistic* as defined by ”Student” Gosset [1908], as well as the p-value. Significant results ($p < 0.05$) are marked with a *.

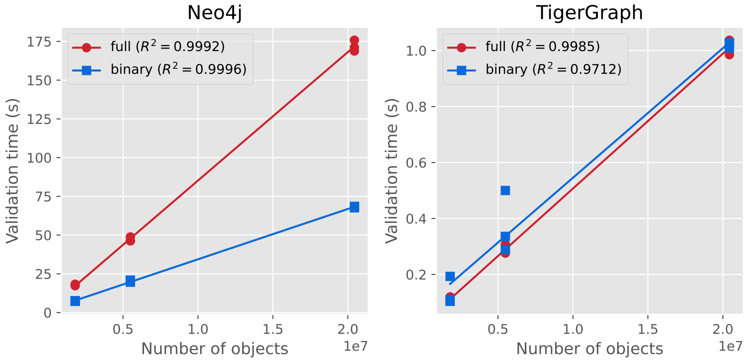


Figure 6.4: Validation time against number of objects in the data graph for the SNB dataset with no schema violations. The markers represent individual measurements and the lines represent fitted linear regression models.

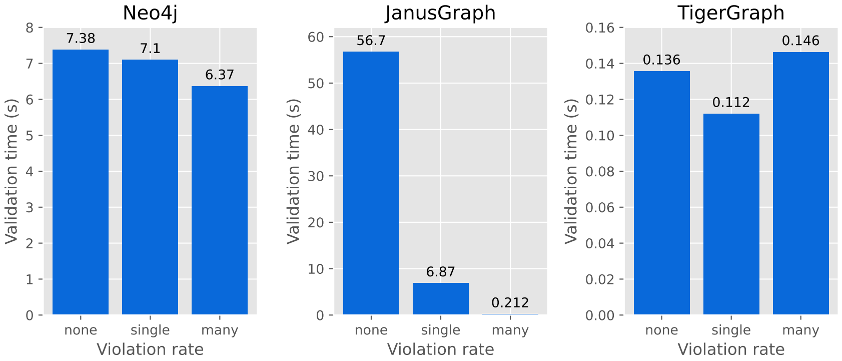


Figure 6.5: Mean validation time at different violation rates for binary validation of the SNB (SF0.1) dataset.

do not expect the amount of violations to have a significant impact on the full variant. A one-way ANOVA is performed to test the null hypothesis that the mean validation time is the same for all violation rates for a particular database. The results are shown in Figure 6.5 and Table 6.3.

The results show different patterns for each database. For Neo4j, increasing the violation rate decreases validation time, but only by a small amount. For JanusGraph, even a single violation cuts the validation time by an order of magnitude, and when half of all nodes violate a constraint, validation time is reduced significantly once more. For TigerGraph, we found no evidence that increasing violation rate reduces validation time.

The different patterns could be explained by the order in which different kinds of violations are checked in our queries. Recall that in Neo4j, we introduce violations by removing mandatory edges. This happens to be the last thing that is checked in our validation queries, hence all other checks still need to be completed. The fact that there is any improvement at all does suggest that Neo4j recognizes it can terminate the query as soon as it finds a violation. For JanusGraph, we only need two queries: the first checks all nodes and the second checks all edges. Since the violation always appears

Database	Mean validation time (s)			F	p
	None	Single	Many		
Neo4j	7.38	7.10	6.37	14.21	**0.005***
JanusGraph	56.74	6.87	0.21	27152	< **0.0001***
TigerGraph	0.14	0.11	0.15	0.55	0.601

Table 6.3: Mean validation time at different violation rates for binary validation of the SNB (SF0.1) dataset. A one-way ANOVA is performed to test for a difference between the mean validation time among all violation rates. We give the F-statistic as defined by Snedecor and Cochran [1989] as well as the p-value. Significant results ($p < 0.05$) are marked with a *.

in one of the nodes, the edge query can be skipped. This yields significant time gains, since there are roughly 5 times more edges than nodes. By contrast, TigerGraph shows no indication of improvement when introducing violations, which suggests that it does not terminate early.

6.3.4 RQ4: Database

From the previous results, we can see there is a clear difference in performance between database systems. In nearly all scenarios, TigerGraph beats Neo4j by an order of magnitude, and JanusGraph is another order of magnitude slower. The only exception occurs when we introduce violations to the data. When there is a single violation, JanusGraph's performance is roughly equal to Neo4j's, and when there are many violations, JanusGraph is on par with TigerGraph.

We are hesitant to draw conclusions about the optimal level of performance that could be achieved with better optimizations specific to each database, since indexes and other optimizations can have a large impact on execution time. However, these results give an indication of the level of performance that can be expected from an initial implementation.

6.3.5 Summary

We summarize the main results of our experiment:

- Schema validation can feasibly be performed with Neo4j and TigerGraph for datasets of 20M objects. JanusGraph did not complete on a dataset of 5M objects.
- In nearly all scenarios, TigerGraph is much faster than Neo4j, and Neo4j is much faster than JanusGraph. However, when schema violations are introduced to the data, JanusGraph's performance drastically improves.
- Neo4j and TigerGraph scale approximately linearly with the size of the data.
- Schema violations have a moderate impact on validation time for Neo4j, a very significant impact for JanusGraph, and no discernible impact for TigerGraph. This suggests Neo4j and JanusGraph terminate the query as soon as a violation is detected.
- Only Neo4j shows a significant difference in validation time between the full and binary variants when the data conforms to the schema, but this difference is no longer apparent when the data is small (~200K objects).

Further discussion regarding the validity of our results can be found in Section 7.2.

Chapter 7

Limitations and Future Work

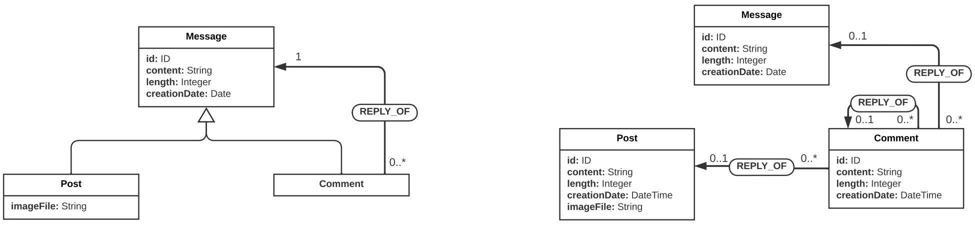
7.1 Schema Formalism

We discuss limitations and opportunities for improvement of our schema formalism as defined in Chapter 4.

Key constraints. Our schema formalism does not support property uniqueness or key constraints. While key constraints were already present in ER models [Chen, 1976], their application to property graphs is non-trivial. As argued in the PG-Keys proposal [Angles et al., 2021], there is a need for granular and flexible key constraints. Keys should be applicable to nodes, edges, and properties, which could all represent concrete entities. Moreover, key scopes and descriptors should be defined in a way that allows objects to be keyed by their relations to other objects. Finally, keys may be *exclusive*, *mandatory*, *singleton*, or any combination thereof. Future work should consolidate PG-Keys with a comprehensive schema formalism such as ours.

Subtyping. Barker [1990] introduced the concept of subtyping to the ER model, Lbath et al. [2021] included subtyping in their schema inference method, and we have seen subtype relations appearing in the SNB property graph schema (Figure 6.2). Clearly, there is a need for subtyping in property graph schemas. This would help model domains containing type hierarchies more accurately, and would reduce duplication of schema edges and properties.

In general, subtype relations cannot be expressed using our formalism. The problem is illustrated by the schema depicted in Figure 7.1a. How could we express this using our schema formalism? An attempt is given in Figure 7.1b, where properties of the supertype `Message` are copied and distributed over all its subtypes, and the edge from `Comment` to `Message` is accompanied by an edge from `Comment` to Post and from `Comment` to itself. While the properties are modeled correctly in this case, it breaks down when we look at the cardinalities.



(a) A schema with a subtype relation. The arrow with a solid white head indicates a "subtype of" relation.

(b) An attempt to express the same schema without subtype relations. Note that this schema is strictly more permissive than (a).

Figure 7.1: A schema consisting of a supertype with two subtypes which cannot be modeled using our schema formalism. Any property graph that conforms to (a) also conforms to (b), but not the other way around.

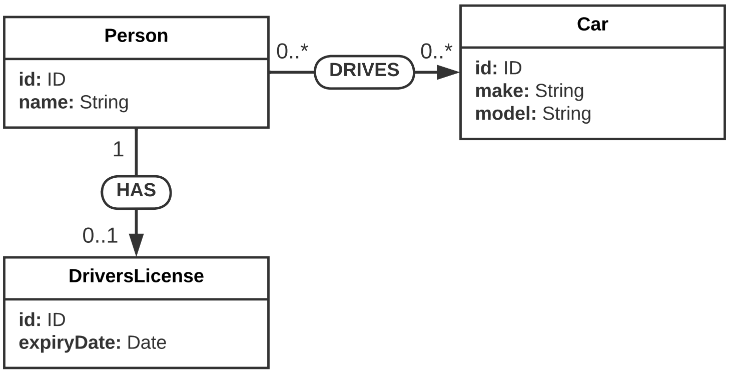


Figure 7.2: A schema illustrating a larger graph pattern constraint which cannot be captured by our schema formalism. While we can specify constraints on edge cardinalities, we cannot ensure that a person only drives a car if they have a driver's license.

In Figure 7.1a, a constraint is specified which we can express in words as "every `Comment` is a reply of exactly one `Message` (or a subtype of `Message`)". However, it is not possible to specify such a constraint without subtype relations, as illustrated by Figure 7.1b. This solution allows `Comments` which are not a reply to anything, or which are a reply to more than one thing. Then again, if we disregard the cardinality constraints, the two schemas are equivalent. This example shows that our schema formalism can be used to model some but not all subtype relations.

Larger graph patterns. It is not possible to specify constraints over a graph pattern larger than two neighboring nodes and the edge between them. For example, say we want to allow a person to drive a car, but only when they have a driver's license. Figure 7.2 depicts a schema that models these entities and their relations using three nodes and two edges. Unfortunately, it is not possible to prevent a person from driving a car without having a driver's license. This illustrates the limited scope of the constraints that we can specify.

Mutual exclusivity. Another kind of constraint introduced by Barker [1990] is mutual exclusivity of edges. For example, a `Person` may have a `WORKS_AT` edge to a `Bank` or a `TradingCompany`, but not both. Mutual exclusivity could be incorporated into our schema formalism, though we should carefully consider whether the benefits for users weigh up against the added complexity. Some constraints may be easier to understand and implement efficiently through separate queries or another constraint language that sits on top of our schema validation queries.

7.2 Empirical Evaluation

We discuss limitations and threats to the validity of our experiments of Chapter 6.

Internal validity. This concerns the degree of confidence in the causal relationships which we inferred from our results. The way we measured validation time poses a threat to internal validity, because the instrumentation differed between databases. For Neo4j and TigerGraph, we rely on the query logs to yield an accurate measurement, while for JanusGraph we use the wall-clock time. The method of time measuring might create some variance between databases, although this can only explain a small part of the difference we observed.

Furthermore, some variance in validation time can be explained by background processes and low-level CPU optimizations. We suspect this factor to have a small impact, though possibly significant when measuring on a sub-second timescale.

Construct validity. This concerns the way our variables of interest were operationalized. One threat to construct validity comes from the way we introduced schema violations into the data. This was done by either removing a mandatory property or a mandatory edge for a node (depending on the database). In reality, different kinds of violations could occur: extraneous properties, incorrect data types, or incorrect edge cardinality and others. These could have differing effects on validation time.

External validity. This concerns the generalization of our results. One such threat comes from the scale of the datasets used in our experiment, which ranged from roughly 200K to 20M objects. We observed a linear relationship between the number of objects and the validation time. However, this relationship may not generalize to very large datasets, especially when they are too large to fit on a single disk.

Another threat to external validity arises from the two schemas we have analyzed (Recommendations and SNB). The number of schemas is limited, and we did not perform any experiments to isolate the effect of different schemas. Moreover, the schemas are similar in terms of the kinds of constraints they express. Neither of the schemas contain cardinality constraints of the form “exactly n”, even though this can be expressed with our schema formalism. Future work could investigate the relationship between validation time and schema size or the kinds of constraints in the schema. This could be done by validating the same datasets against several schemas.

Finally, it is unclear whether the differences between databases would persist when further optimizations are made. We tried to put equal effort into optimization for each database, but results may differ when more labor is done.

7.3 Miscellaneous

Generic validation implementation. In our approach, the validation queries are hand-crafted for each schema. This requires knowledge of the query language and the schema functionality of a particular database. To make things easier for the user, we could let them provide a schema in a generic schema language or diagram, and generate the validation queries instead.

Alternatively, when the schema and data are both given as graphs, validation could be achieved using generic graph pattern matching queries. In fact, we have provided such a generic implementation for Neo4j[1]. Unfortunately, validation times were prohibitive for all but the smallest graphs. However, there are certainly opportunities for performance gains.

Validation algorithms. There are advantages to our approach using graph queries for validation: they are easy to adjust, portable between database versions, and convenient to prototype. However, when performance is the primary goal, a lower-level implementation may be more suitable. Specialized validation algorithms and indexes could be developed to achieve performance benefits. Moreover, since all of our validation rules have a limited scope, validation queries may be parallelized and distributed across multiple machines.

If a database instance is frequently updated, the cost of scanning the entire graph on every update may be prohibitive. In such a scenario, an incremental validation algorithm could provide significant benefits. Due to the limited scope of our validation rules, only the fragment of the graph that is affected by the update would need to be revalidated. Such an algorithm could be used to reject any updates that cause a schema violation.

Schema-aware query optimization. Schema knowledge may be used by the query engine to improve query planning. Such optimizations are widely employed in relational databases [Chakravarthy et al., 1990; Meier et al., 2013; Silberschatz et al., 2011], and have been studied in the context of graph databases as well [Buneman et al., 1997; Popa and Tannen, 1999]. Similar methods could be applied to property graph databases when a schema is present.

Query type checking and inference. The presence of schemas may benefit the usability of graph databases. Through query type checking, incorrectly typed parameters can be detected. Moreover, the result type of a query could be inferred to ensure the result is of the expected type before execution. Query type inference could be integrated with type systems of programming languages to achieve type safety across the database and the application. A type inference approach like the one presented by Colazzo and Sartiani [2015] may be adapted to the property graph model.

Relation to other notions of schema conformance. It may be of theoretical interest to investigate how our notion of conformance is related to other methods found in the literature, such as simulation [Buneman et al., 1997] and homomorphisms [Bonifati et al., 2019]. If they are equivalent, results from other work may be applied to our model as well.

[1] https://github.com/nimobeeren/thesis/tree/main/neo4j/generic

Chapter 8

Conclusion

We have presented an end-to-end framework for property graph schema specification and validation, showing how constraints based on common conceptual data modeling methods can be captured in a unified schema model. Moreover, we have shown how a property graph can be validated against a schema using first-order logic rules. Through an empirical study, we have evaluated our prototypical implementation in various scenarios, demonstrating feasibility for many use-cases. All database systems that we tested were able to complete some schema validation workloads, although there were significant differences in execution time.

Our schema model is able to capture many kinds of constraints, but not all. We consider key constraints as one of the most important features to be added in the future. Furthermore, the inclusion of subtyping would benefit data modeling capabilities. Other kinds of constraints could be integrated into our model as needed, or they could be realized as a separate layer on top of our framework.

In our experiment, TigerGraph outperformed both Neo4j and JanusGraph by a clear margin in almost all scenarios. This advantage can be explained at least partially by TigerGraph's focus on schema. While a fixed schema limits flexibility, other database vendors may be able to make significant performance gains if support for explicit schemas is improved. Interestingly, JanusGraph's performance vastly improved when schema violations were introduced into the data, whereas TigerGraph was unaffected. This highlights an opportunity for optimization in TigerGraph, which does not seem to terminate queries even though the final result is known.

We see ample opportunity for performance improvements, either through specialized validation algorithms, indexes, or optimization of validation queries. We have already observed sub-second validation times on datasets of millions of objects, which encourages us to believe that schema validation will be possible on an interactive time scale, especially when incremental algorithms are employed.

There are many open research challenges tangential to our work. We are excited to see how the presence of types in graph databases will benefit people who work with data by enhancing query performance, aiding query specification, and bridging the gap between the programming language and the database.

Bibliography

Abiteboul, S., Buneman, P., and Suciu, D. (1999). *Data on the web: from relations to semistructured data and XML*. Morgan Kaufmann.

Angles, R. (2018). The property graph database model. In *AMW*.

Angles, R., Antal, J. B., Averbuch, A., Birler, A., Boncz, P., Búr, M., Erling, O., Gubichev, A., Haprian, V., Kaufmann, M., Pey, J. L. L., Martínez, N., Marton, J., Paradies, M., Pham, M.-D., Prat-Pérez, A., Spasić, M., Steer, B. A., Szakállas, D., Szárnyas, G., Waudby, J., Wu, M., and Zhang, Y. (2020). The LDBC social network benchmark.

Angles, R., Bonifati, A., Dumbrava, S., Fletcher, G., Hare, K. W., Hidders, J., Lee, V. E., Li, B., Libkin, L., Martens, W., Murlak, F., Perryman, J., Savković, O., Schmidt, M., Sequeda, J., Staworko, S., and Tomaszuk, D. (2021). PG-Keys: Keys for property graphs. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pages 2423–2436, New York, NY, USA. Association for Computing Machinery.

Barker, R. (1990). *CASE Method: Entity Relationship Modelling*. Addison-Wesley.

Bonifati, A., Dumbrava, S., and Mir, N. (2022). Hierarchical clustering for property graph schema discovery. In *EDBT*, pages 449–453.

Bonifati, A., Fletcher, G., Voigt, H., and Yakovets, N. (2018). *Querying Graphs*. Number 51 in Synthesis Lectures on Data Management. Morgan & Claypool Publishers.

Bonifati, A., Furniss, P., Green, A., Harmer, R., Oshurko, E., and Voigt, H. (2019). Schema validation and evolution for graph databases. In Laender, A. H. F., Pernici, B., Lim, E.-P., and de Oliveira, J. P. M., editors, *Conceptual Modeling*, pages 448–456, Cham. Springer.

Buneman, P., Davidson, S., Fernandez, M., and Suciu, D. (1997). Adding structure to unstructured data. In Afrati, F. and Kolaitis, P., editors, *Database Theory — ICDT '97*, pages 336–350, Berlin, Heidelberg. Springer.

Chakravarthy, U. S., Grant, J., and Minker, J. (1990). Logic-based approach to semantic query optimization. *ACM Trans. Database Syst.*, 15(2):162–207.

Chen, P. P.-S. (1976). The entity-relationship model—toward a unified view of data. *ACM transactions on database systems (TODS)*, 1(1):9–36.

Colazzo, D. and Sartiani, C. (2015). Typing regular path query languages for data graphs. In *Proceedings of the 15th Symposium on Database Programming Languages*, DBPL 2015, pages 69–78, New York, NY, USA. Association for Computing Machinery.

Deutsch, A., Francis, N., Green, A., Hare, K., Li, B., Libkin, L., Lindaaker, T., Marsault, V., Martens, W., Michels, J., Murlak, F., Plantikow, S., Selmer, P., Voigt, H., van Rest, O., Vrgoč, D., Wu, M., and Zemke, F. (2021). Graph pattern matching in GQL and SQL/PGQ.

Deutsch, A., Xu, Y., Wu, M., and Lee, V. (2019). TigerGraph: A native MPP graph database.

Fagin, R. and Vardi, M. Y. (1984). The theory of data dependencies — an overview. In Paredaens, J., editor, *Automata, Languages and Programming*, pages 1–22, Berlin, Heidelberg. Springer.

Fan, W., Fan, Z., Tian, C., and Dong, X. L. (2015). Keys for graphs. *Proc. VLDB Endow.*, 8(12):1590–1601.

Fan, W., Wu, Y., and Xu, J. (2016). Functional dependencies for graphs. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1843–1857, New York, NY, USA. Association for Computing Machinery.

Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., and Taylor, A. (2018). Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1433–1445, New York, NY, USA. Association for Computing Machinery.

ISO/IEC 19501:2005 (2005). Information technology – Open Distributed Processing – Unified Modeling Language (UML) Version 1.4.2. Standard, International Organization for Standardization, Geneva, CH.

Lbath, H., Bonifati, A., and Harmer, R. (2021). Schema inference for property graphs. In *EDBT 2021 - 24th International Conference on Extending Database Technology*, EDBT, pages 499–504, Nicosia, Cyprus. Short Paper.

Lei, X. (2021). Property graph schema extraction. Master's thesis, Eindhoven University of Technology.

Meier, M., Schmidt, M., Wei, F., and Lausen, G. (2013). Semantic query optimization in the presence of types. *Journal of Computer and System Sciences*, 79(6):937–957. JCSS Foundations of Data Management.

Pan, J. Z. (2009). Resource description framework. In *Handbook on ontologies*, pages 71–90. Springer.

Pareti, P. and Konstantinidis, G. (2022). A review of SHACL: From data validation to schema reasoning for RDF graphs. In Šimkus, M. and Varzinczak, I., editors, *Reasoning Web. Declarative Artificial Intelligence*, pages 115–144, Cham. Springer.

Pokorný, J., Valenta, M., and Kovačič, J. (2017). Integrity constraints in graph databases. *Procedia Computer Science*, 109:975–981. 8th International Conference on Ambient Systems, Networks and Technologies, ANT-2017 and the 7th International Conference on Sustainable Energy Information Technology, SEIT 2017, 16-19 May 2017, Madeira, Portugal.

Popa, L. and Tannen, V. (1999). An equational chase for path-conjunctive queries, constraints, views. In *International Conference on Database Theory*, pages 39–57. Springer.

Rodriguez, M. A. (2015). The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*, DBPL 2015, pages 1–10, New York, NY, USA. Association for Computing Machinery.

Silberschatz, A., Korth, H. F., and Sudarshan, S. (2011). *Database System Concepts*, volume 6. McGraw-Hill, New York, NY, USA.

Snedecor, G. W. and Cochran, W. G. (1989). *Statistical Methods*, chapter 10. Iowa State University Press, eigth edition.

”Student” Gosset, W. S. (1908). The probable error of a mean. *Biometrika*, 6(1):1–25.