A Formal Design Framework for Practical Property Graph Schema Languages

Nimo Beeren
Eindhoven University of Technology
Eindhoven, The Netherlands
nimobeeren@gmail.com

George Fletcher
Eindhoven University of Technology
Eindhoven, The Netherlands
g.h.l.fletcher@tue.nl

ABSTRACT

Graph databases are increasingly receiving attention from industry and academia, due in part to their flexibility; a schema is often not required. However, schemas can significantly benefit query optimization, data integrity, and documentation. There currently does not exist a formal framework which captures the design space of state-of-the-art schema solutions. We present a formal design framework for property graph schema languages based on first-order logic rules, which balances expressivity and practicality. We show how this framework can be adapted to integrate a core set of constraints common in conceptual data modeling methods. To demonstrate practical feasibility, this model is implemented using graph queries for modern graph database systems, which we evaluate through a controlled experiment. We find that validation time scales linearly with the size of the data, while only using unoptimized straightforward implementations.

1 INTRODUCTION

Graph databases have been steadily growing in popularity in recent years, receiving attention from both industry and academia. Graphs offer a simple yet powerful model consisting of nodes and edges which can be structured freely. The *property graph* model, being a predominant data model among graph databases today, associates nodes and edges with *labels* and key–value pairs known as *properties*. This enables a 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. Current work on property graph schema solutions aims to bring schemas back to the world of graphs, blending the flexibility of the graph model with the structure of relational databases. However, there currently is no systematic formal framework which captures the state of the art. Such a framework is necessary to progress towards standardization of property graph schema languages, to formally understand the relative capabilities of current solutions, and to identify missing capabilities needed for practical impact.

To remedy this lack of systematic understanding of property graph schema design, in this paper we make the following concrete **contributions**. (1) We propose a formal framework for property graph schema languages which is general enough to study the state of the art. (2) Next, we provide an instantiation of this framework in the form of a concrete schema language which integrates constraints from common conceptual data modeling methods. (3) Moreover, we provide a prototypical implementation of schema validation in the form of graph queries, and (4) we investigate the practical feasibility of our approach using contemporary graph databases by means of a controlled experiment.

2 RELATED WORK

Graph Schema Formalisms. Several proposals address schemas for graph-based data models other than property graphs. Early work enables the specification of allowed edges and paths for edge-labeled graphs using simulation [1, 9]. Others have proposed methods based on regular expressions, supporting constraints over graph patterns [12]. However, these approaches are not directly applicable to property graphs.

A basic notion of property graph schema using first-order logic rules was first defined in [2]. In this work, no distinction is made between mandatory and optional properties. Another approach uses restrictive homomorphism semantics for schema validation [8]. This method does support mandatory properties, but neither of these solutions support edge cardinality constraints.

Property Graph Schema Implementations. Current property graph database systems vary in their support and philosophy regarding 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[1] property graph databases: *Neo4j*, *JanusGraph*, and *TigerGraph*.

Neo4j[2] is the most popular graph database engine as of today. Neo4j's approach to schema is primarily implicit: after inserting data, a schema that describes the data can be retrieved using built-in functions. In addition, some constraints can be explicitly specified, including mandatory properties.

JanusGraph[3] has the most comprehensive set of schema features among popular systems. In addition to automatically generating a schema, explicit schema definition is also supported. It can be specified which properties may exist depending on the label of a node or edge. Furthermore, property values are restricted to a data type and may be single-valued or multi-valued. 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 or many-to-many. These features are centered around specifying what is allowed in the graph, but mandatory properties and edges are not supported. JanusGraph can impose constraints on the maximum edge cardinality, but not the minimum.

TigerGraph [14] is schema-first; the entire schema must be specified before a database is instantiated. This enables powerful optimizations, building on decades of research on relational databases. TigerGraph's schema is strict, in the sense that every node label, edge label, and property must be explicitly defined.

[1]https://db-engines.com/en/ranking/graph+dbms (accessed September 2022)
[2]https://neo4j.com/
[3]https://janusgraph.org/

Table 1: Comparison of schema features supported by property graph schema formalisms and implementations: Angles (AG) [2], Bonifati et al. (BN) [8], Neo4j (NJ)[2], JanusGraph (JG)[3], TigerGraph (TG) [14], and our solution.

	AG	BN	NJ	JG	TG	Ours
Mandatory properties	×	✓	✓	×	✓	✓
Allowed properties	×	✓	×	✓	✓	✓
Endpoint constraints	✓	✓	×	✓	✓	✓
Data type constraints	✓	✓	×	✓	✓	✓
Optional properties	×	✓	×	✓	×	✓
Maximum cardinality	×	×	×	✓	×	✓
Minimum cardinality	×	×	×	×	×	✓

Moreover, all allowed properties are also mandatory. All properties have a fixed data type, which may be singular or multi-valued. Schema edges must specify the source and target node type(s).

Summary. Table 1 compares the state-of-the-art with our solution. Features: Mandatory property constraints express that some property must be defined on a node or edge. Allowed property constraints express that no properties other than those specified in the schema may be defined. Endpoint constraints express that edges must not connect nodes which do not conform to some particular types. Data type constraints express that the value of a property must be of a particular data type. Optional property constraints express that the value of a property must be either undefined or of a particular type. Maximum and minimum cardinality constraints express that a node must have a particular number of incoming or outgoing edges of a particular type.

3 PRELIMINARIES

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 [13]. 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. For simplicity, we do not consider undirected edges, although they could be easily simulated. 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 3.1 (Basic record). A *record* is a finite partial function $r : \mathcal{N} \rightharpoonup \mathcal{V}$ 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 3.2 (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 $N \cap E = \emptyset$;
- $\rho : E \longrightarrow (N \times N)$ is a total function mapping edges to ordered pairs of nodes;
- $\lambda : (N \cup E) \longrightarrow 2^{\mathcal{L}}$ is a total function mapping nodes and edges to a (possibly empty) set of labels;
- $\pi : (N \cup E) \longrightarrow \mathcal{R}$ is a total function mapping nodes and edges to a record.

Given a node u, the set of *outgoing* edges is $\{e \in E \mid \exists v \in N : \rho(e) = (u, v)\}$, and the set of *incoming* edges is $\{e \in E \mid \exists v \in N : \rho(e) = (v, u)\}$. The functions $\texttt{src}$ and $\texttt{trg}$ map ordered pairs to their first and second element, i.e. $\texttt{src}((u, v)) = u$ and $\texttt{trg}((u, v)) = v$. To refer to the source and target *endpoints* of an edge e, we may write $\texttt{src}(\rho(e))$ and $\texttt{trg}(\rho(e))$ respectively.

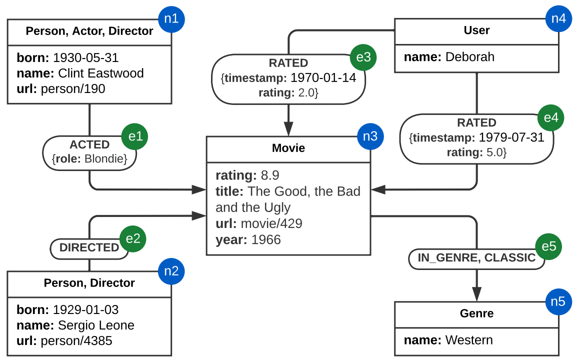

Figure 1: A property graph consisting of 5 nodes and 5 edges, representing a small movie database. Nodes and edges are given identifiers (in blue and green circles) to refer to them in text.

An example property graph is given in Figure 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. This example can be mapped to our formal property graph model as follows:

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

4 BASIC PROPERTY GRAPH SCHEMA

We assume the existence of a set of property types $\mathcal{T}$. We next introduce several supporting concepts.

Definition 4.1 (Basic property conformance). For each property type $t \in \mathcal{T}$ there is a set $[\![t]\!] \subseteq \mathcal{V}$ that consists of all property values that *conform* to the type t.

Definition 4.2 (Basic record type). A *record type* is a finite partial function $t^{\mathsf{r}} : \mathcal{N} \rightharpoonup \mathcal{T}$ that maps some property names to a property type. We denote record types as $\langle a_1 : t_1, \ldots, a_n : t_n \rangle$. The set of all record types is denoted as $\mathcal{T}^{\mathsf{r}}$.

Definition 4.3 (Basic record conformance). We say that a record r *conforms* to a record type t^{r}, denoted $r \in [\![t^{\mathsf{r}}]\!]$, if for each property name $k \in \mathcal{N}$ it holds that (1) $r(k)$ is defined iff $t^{\mathsf{r}}(k)$ is defined and (2) $r(k) \in [\![t^{\mathsf{r}}(k)]\!]$ if $r(k)$ and $t^{\mathsf{r}}(k)$ are defined.

Next, we define a basic notion of schema.

Definition 4.4 (Basic property graph schema). A *property graph schema* is a tuple $S = (N, E, \rho, \lambda, \tau)$ 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;
- $\lambda : (N \cup E) \to 2^{\mathcal{L}}$ is a total function mapping nodes and edges to a (possibly empty) set of labels;
- $\tau : (N \cup E) \to \mathcal{T}^{\mathsf{r}}$ is a total function mapping nodes and edges to record types.

The similarity between property graphs and schemas allows us to think about them in similar ways. The key difference is that schemas associate properties 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. $\mathcal{T} \subseteq \mathcal{V}$.

Finally, we define *conformance* of a property graph to a schema.

Definition 4.5 (Conformance relation). Given a property graph $G = (N, E, \rho, \lambda, \pi)$ and a property graph schema $S = (N', E', \rho', \lambda', \tau)$ we define the binary *conformance relation* $\sqsubseteq = \{(o, o') \in (N \cup E) \times (N' \cup E') \mid \lambda(o) = \lambda'(o') \wedge \pi(o) \in [\![\tau(o')]\!]\}$. We say that an object o *conforms* to a schema object o' if and only if $o \sqsubseteq o'$.

Definition 4.6 (Basic schema conformance). Given a property graph $G = (N, E, \rho, \lambda, \pi)$ and a property graph schema $S = (N', E', \rho', \tau)$, 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':

$$\forall e \in E\ \exists e' \in E' : e \sqsubseteq e' \wedge \mathtt{src}(\rho(e)) \sqsubseteq \mathtt{src}(\rho'(e')) \\ \wedge\ \mathtt{trg}(\rho(e)) \sqsubseteq \mathtt{trg}(\rho'(e'))$$

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 types of edges that are allowed. Under these definitions, every property is effectively mandatory, and no properties may exist other than those specified in the schema. Hence mandatory property constraints, allowed property constraints, and data type constraints are maintained. Note that rule 2 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 incident edge that is not explicitly allowed, even if that edge itself does conform to some schema edge. Hence, endpoint constraints are also maintained. An example of a property graph schema is given in Figure 2. The notation is similar to the one used for property graphs, with some additions for cardinality constraints and optional properties (explained in Section 5).

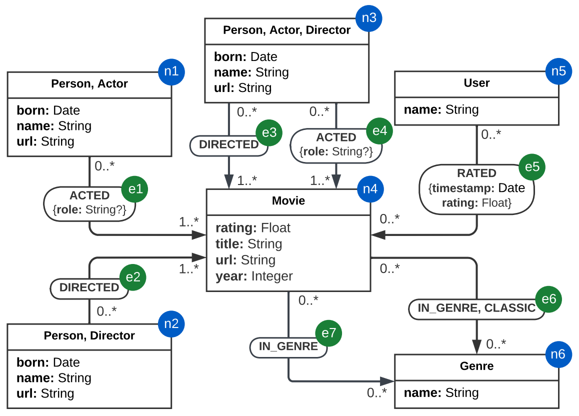

Figure 2: A property graph schema representing for a movie database. Nodes and edges are given identifiers (in blue and green circles) to refer to them in text. The graph of Figure 1 conforms to this schema.

5 ADAPTING THE SCHEMA LANGUAGE

Schema languages must balance expressivity with practicality. To determine which features to include, we follow common conceptual modeling methods such as the Entity–Relationship (ER) model [11]. This model bears resemblance to the property graph model, as entities can be mapped to nodes, relationships to edges, and attribute–value pairs to properties. The ER model can be used to express constraints such as mandatory properties, endpoint constraints, and data type constraints. After the original specification, the ER model has been extended in various ways. For example, [6] proposed a notation which introduced cardinality constraints and optional properties. To achieve feature parity with this extended ER model, we adapt our basic schema model to support two additional kinds of constraints: cardinality constraints and optional properties.

5.1 Cardinality Constraints

To improve readability of our conformance rules, we first introduce a generalization of the existential quantifier which enables counting the number of distinct variables that satisfy a predicate.

Definition 5.1 (Counting quantifier). The *counting quantifier* is defined as follows. Given two natural 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 5.2 (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.

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. $\mathtt{src}([n_1, m_1],\ [n_2, m_2]) = [n_1, m_1]$ and $\mathtt{trg}([n_1, m_1],\ [n_2, m_2]) = [n_2, m_2]$.

Next, we revise the definitions of property graph schema and schema conformance, making use of cardinality constraints. The following definitions subsume Definition 4.4 and 4.6.

Definition 5.3 (Property graph schema). A *property graph schema* is a tuple $S = (N, E, \rho, \lambda, \tau, \eta)$ 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;
- $\lambda : (N \cup E) \to 2^{\mathcal{L}}$ is a total function mapping nodes and edges to a (possibly empty) set of labels;
- $\tau : (N \cup E) \to \mathcal{T}^{\mathrm{r}}$ is a total function mapping nodes and edges to record types;
- $\eta : E \to C$ is a total function mapping schema edges to cardinality constraints.

Definition 5.4 (Schema conformance). Given a property graph $G = (N, E, \rho, \lambda, \pi)$ and a property graph schema $S = (N', E', \rho', \lambda', \tau, \eta)$ we say that G *conforms* to S if and only if all of the following rules hold.

(1) Same as rule 1 of Definition 4.6.
(2) Same as rule 2 of Definition 4.6.
(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:

$$\forall n \in N\ \forall e' \in E' :$$
$$\big[n \sqsubseteq \mathtt{src}(\rho(e')) \implies \exists^{\mathtt{trg}(\eta(e'))} e \in E :$$
$$e \sqsubseteq e' \wedge \mathtt{src}(\rho(e)) = n \wedge \mathtt{trg}(\rho(e)) \sqsubseteq \mathtt{trg}(\rho(e'))\big]$$

(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:

$$\forall n \in N\ \forall e' \in E' : \quad \big[n \sqsubseteq \mathtt{trg}(\rho(e')) \implies \exists^{\mathtt{src}(\eta(e'))} e \in E :$$
$$e \sqsubseteq e' \wedge \mathtt{src}(\rho(e)) \sqsubseteq \mathtt{src}(\rho(e')) \wedge \mathtt{trg}(\rho(e)) = n\big]$$

Compared to Definition 4.6, rule 1 and 2 are unchanged, while rule 3 and 4 maintain minimum and maximum cardinality constraints on edges. With these new rules, we can also enforce mandatory edges, i.e. a cardinality constraint of "at least one".

Cardinality constraints are denoted in Figure 2 by annotating each edge with two intervals (one at the source and one at the target). Intervals such as $[n, m]$ are written as $n..m$, following UML notation [15]. Moreover, we use the "look-across" notation, 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. In this example, $\eta(e_1) = ([0, *], [1, *])$.

5.2 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 definition subsumes Definition 3.1.

Definition 5.5 (Record). A *record* is a total function $r : \mathcal{N} \to \mathcal{V} \cup \{\mathbf{undef}\}$ that maps property names to property values or the special value **undef**. We denote such records as $\langle n_1 : v_1, \dots, n_k : v_k \rangle$. 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 $r(k) = \mathbf{undef}$. This can be seen as the "default" value of a property.

We adjust the definitions of property conformance, record type, and record conformance accordingly. These subsume Definition 4.1, 4.2, and 4.3.

Definition 5.6 (Property conformance). For each property type $t \in \mathcal{T}$ there is a set $[[t]] \subseteq \mathcal{V} \cup \{\mathbf{undef}\}$ that contains all property values that *conform* to the type t. We say t is *optional* iff $\mathbf{undef} \in [[t]]$. We use the notation $t?$ to mark a property as optional, i.e. $[[t?]] = [[t]] \cup \{\mathbf{undef}\}$.

Definition 5.7 (Record type). A *record type* is a total function $t^{\mathrm{r}} : \mathcal{N} \to \mathcal{T}$ that maps property names to property types. We denote such record types as $\langle n_1 : t_1, \dots, n_k : t_k \rangle$.

Definition 5.8 (Record conformance). We say that a record r *conforms* to a record type t^{r}, denoted $r \in [[t^{\mathrm{r}}]]$, if and only if for each property name $k \in \mathcal{N}$ it holds that $r(k) \in [[t^{\mathrm{r}}(k)]]$.

With these final definitions we can maintain optional property constraints. An example is given in Figure 2, where the role properties on e_1 and e_4 are optional.

6 PRACTICAL VALIDATION

Assumptions. To reduce the complexity of implementation, we assume that the set of labels of a schema object functionally determines the record type of that schema object. Recall from Section 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 Section 7.

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. With binary validation we can stop as soon as we find a single violation, while full validation always requires finding all violations.

Implementation. We implement our method using three modern graph database systems which each take a unique schema approach. Where possible, we make use of the schema functionality exposed by the database engine. For constraints that cannot be validated in this way, we use graph queries. In all cases, we leave the database system internals unchanged. Further details and example queries are provided in [7], and the complete source code is available on GitHub[4].

7 EMPIRICAL EVALUATION

We next evaluate the performance of our prototypical implementation through a controlled experiment using three graph database systems which we will call *GDB1*, *GDB2*, and *GDB3*. The presented results may help the reader to determine whether our approach is feasible for their use case. To explore the cost of validating property graph schema conformance, we aim to answer the following research questions:

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

Table 2: Size statistics of datasets. $|N|$ and $|E|$ represent the number of nodes and edges. $|N'|$ and $|E'|$ represent the number of schema nodes and schema edges.

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

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?

7.1 Methodology

We perform a set of performance experiments in various realistic scenarios. For each workload, we perform three subsequent runs, from which we report the average.

Validation time. We are interested in the time it takes to validate conformance of a graph to a schema. For GDB1 and GDB3, 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 things, such as the time taken to print the results. GDB2 does not provide such logs, so we use the wall-clock time. If multiple queries are needed to produce an answer, we take the sum of their execution times.

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**[5] is a relatively small dataset with a simple schema, provided by Neo4j. It consists of real data sourced from *Open Movie Database*[6] and *MovieLens*[7]. It contains movies, actors, directors, and users who rate movies. The **LDBC Social Network Benchmark** (SNB) [3] 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 social network concepts. We run multiple workloads with different scale factors (SF), which lets us analyze how our solution scales with the size of the data. These datasets are diverse in scale, but 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 2. Their schemas are provided in [7].

Validation variant. 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). 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 schema violations in the

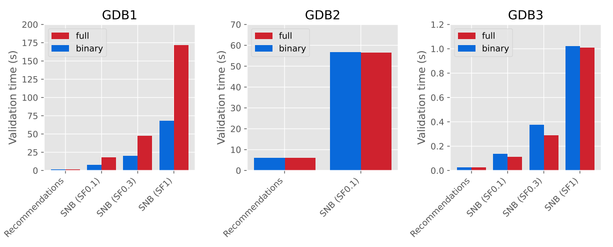

Figure 3: Mean validation time for binary and full variants on all datasets with no schema violations.

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), and *many* (roughly 50% of all nodes violate a constraint).

The way violations are introduced depends on the database system under test. For GDB2, we remove a mandatory property from nodes. Because GDB3 requires every property to have a value, we set it to a default value instead. Because GDB1 supports constraints that prevent missing mandatory properties, we remove a mandatory edge instead.

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.

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 GDB3's outdegree index, which returns the number of outgoing edges with a particular label for a given node. To facilitate independence between subsequent runs, caching is disabled.

Hardware. All workloads are run on a single machine with 4 CPU cores and 6 gigabytes of dedicated memory.

7.2 Results

RQ1: Validation Variant. We run both the binary and full validation queries on all datasets. For these workloads, no violations are introduced. A two-sided T-test is performed to test for a difference between the mean validation time for the full and binary variant. The results for each database are shown in Figure 3. The queries for GDB2 on the SNB dataset with scale factor 0.3 and larger did not complete within an hour, and are excluded.

For GDB1, 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 GDB1's schema statistics, which are retrieved in constant time and eliminate the need for some of the queries in the full

[5]https://github.com/neo4j-graph-examples/recommendations
[6]https://www.omdbapi.com/
[7]https://grouplens.org/datasets/movielens/

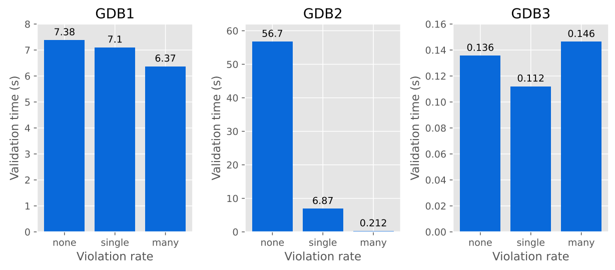

Figure 4: Mean validation time at different violation rates for binary validation of the SNB (SF0.1) dataset.

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 of a difference between the two variants for GDB2 and GDB3. 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.

RQ2: Scale. Our results suggest a linear relationship between the number of objects and validation time. This is expected, since our queries perform a linear scan of the nodes and edges in the data graph. Further details can be found in [7].

RQ3: Violation Rate. To evaluate how schema violations affect validation time, we choose a fixed dataset and introduce some errors to the data. We choose the SNB dataset at SF0.1, because all databases could successfully complete the validation process. We only test the binary validation variant, because we do not expect the amount of violations to have a significant impact on the full variant.

The results in Figure 4 show different patterns for each database. A one-way ANOVA is performed to test for a difference between the mean validation time among all violation rates. For GDB1, increasing the violation rate decreases validation time ($p = 0.005$), but only by a small amount. For GDB2, violations cut validation time by two orders of magnitude ($p < 0.0001$). For GDB3, we found no evidence ($p = 0.601$) that increasing violation rate affects validation time. For further details see [7].

RQ4: Database. From the previous results, we can see there is a clear difference in performance between database systems. In nearly all scenarios, GDB3 beats GDB1 by an order of magnitude, and GDB2 is another order of magnitude slower. The only exception occurs when we introduce violations to the data. When there is a single violation, GDB2's performance is roughly equal to GDB1's, and when there are many violations, GDB2 is on par with GDB3.

Summary. (1) Schema validation can feasibly be performed with GDB1 and GDB3 for datasets of 20M objects. GDB2 did not complete on a dataset of 5M objects. (2) In nearly all scenarios, GDB3 is much faster than GDB1, and GDB1 is much faster than GDB2. However, when schema violations are introduced to the data, GDB2's performance drastically improves. (3) GDB1 and GDB3 scale approximately linearly with the size of the data. (4) Schema violations have a moderate impact on validation time for GDB1, a very significant impact for GDB2, and no discernible impact for GDB3. This suggests GDB1 and GDB2 terminate the query as soon as a violation is detected. (5) Only GDB1 shows a significant difference in validation time between the full and binary variants when the data conforms to the schema, but this effect is no longer apparent when the data is small (~200K objects).

8 LIMITATIONS, FUTURE WORK, AND CONCLUSIONS

Limitations. More features could be added as needed to our language, e.g., key constraints [4, 11], subtyping [6, 16], mutual exclusivity [6], and larger graph patterns [12].

Some threats to the validity of our experimental results arise. Considering internal validity, the way we measured time could explain a small part of the variance between databases. Moreover, some variance can be explained by background processes and low-level CPU optimizations. This could have a noticeable effect when measuring on a sub-second timescale, though this is unlikely to change our conclusions. Considering construct validity, we operationalized schema violations by removing a mandatory property or edge, but other kinds of violations were not analyzed. Considering external validity, we used datasets up to 20M objects, but the observed relationships may not generalize to larger datasets, especially when they are too large to fit in memory. Furthermore, the two schemas consisted of similar constraints, and did not include all the kinds of constraints that can be expressed in our schema language. Finally, we are hesitant to draw conclusions about the optimal level of performance that could be achieved with better optimizations specific to each database. However, our results give an indication of the level of performance that can be expected from an initial implementation. Further opportunities for performance improvement are identified in [7].

Future Work. First, a formal analysis of our framework could improve understanding of its relation to other methods based on simulation [9] and homomorphism semantics [8] as well as the recently appearing work on PG-Schema [5]. Furthermore, the study of specialized validation algorithms and indexes could improve performance. For example, an incremental algorithm could provide significant benefit when a database is frequently updated. Moreover, schema information may be used to optimize query planning, as is often done in relational databases [9, 10, 17]. In addition, query type inference and type checking [12] could benefit usability of graph databases by detecting incompatible parameter types and inferring a query result type prior to execution. Additional important broader areas for investigation of our framework include: schema extraction and inference solutions for our model; the empirical impact of more complex schemas and different application domains; the interaction of schema enforcement and transaction management; and, a finer study of the search strategies employed by industrial systems for query-driven schema validation.

Concluding remarks. We have demonstrated practical feasibility of our approach for datasets up to 20M objects, even with straightforward non-optimized implementations. We observed that our implementation scales linearly with the size of the data. In general, we observed a large difference in performance between popular property graph databases. Our framework enables further theoretical and practical study of property graph schema languages, as highlighted in Section 8.

Acknowledgements. We thank Jan Hidders, Alex Poulovassilis, and Peter Wood for earlier discussions on property graph schema design.

REFERENCES

[1] Serge Abiteboul, Peter Buneman, and Dan Suciu. 1999. *Data on the web: from relations to semistructured data and XML.* Morgan Kaufmann.

[2] Renzo Angles. 2018. The Property Graph Database Model. In *AMW*.

[3] Renzo Angles, János Benjamin Antal, Alex Averbuch, Altan Birler, Peter Boncz, Márton Búr, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba Pey, Norbert Martínez, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasić, Benjamin A. Steer, Dávid Szakállas, Gábor Szárnyas, Jack Waudby, Mingxi Wu, and Yuchen Zhang. 2020. The LDBC Social Network Benchmark. https://doi.org/10.48550/ARXIV.2001.02299

[4] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith W. Hare, Jan Hidders, Victor E. Lee, Bei Li, Leonid Libkin, Wim Martens, Filip Murlak, Josh Perryman, Ognjen Savković, Michael Schmidt, Juan Sequeda, Slawek Staworko, and Dominik Tomaszuk. 2021. PG-Keys: Keys for Property Graphs. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2423–2436. 9781450383431 https://doi.org/10.1145/3448016.3457561

[5] Renzo Angles et al. 2023. PG-Schema: Schemas for Property Graphs. In *SIGMOD 2023*.

[6] Richard Barker. 1990. *CASE Method: Entity Relationship Modelling.* Addison-Wesley.

[7] Nimo Beeren. 2022. *Formal Specification and Practical Validation of Property Graph Schemas.* Master's thesis. Eindhoven University of Technology.

[8] Angela Bonifati, Peter Furniss, Alastair Green, Russ Harmer, Eugenia Oshurko, and Hannes Voigt. 2019. Schema Validation and Evolution for Graph Databases. In *Conceptual Modeling*, Alberto H. F. Laender, Barbara Pernici, Ee-Peng Lim, and José Palazzo M. de Oliveira (Eds.). Springer, Cham, 448–456. 978-3-030-33223-5

[9] Peter Buneman, Susan Davidson, Mary Fernandez, and Dan Suciu. 1997. Adding structure to unstructured data. In *Database Theory — ICDT '97*, Foto Afrati and Phokion Kolaitis (Eds.). Springer, Berlin, Heidelberg, 336–350. 978-3-540-49682-3

[10] Upen S. Chakravarthy, John Grant, and Jack Minker. 1990. Logic-Based Approach to Semantic Query Optimization. *ACM Trans. Database Syst.* 15, 2 (jun 1990), 162–207. 0362-5915 https://doi.org/10.1145/78922.78924

[11] Peter Pin-Shan Chen. 1976. The entity-relationship model—toward a unified view of data. *ACM transactions on database systems (TODS)* 1, 1 (1976), 9–36.

[12] Dario Colazzo and Carlo Sartiani. 2015. Typing Regular Path Query Languages for Data Graphs. In *Proceedings of the 15th Symposium on Database Programming Languages (DBPL 2015)*. Association for Computing Machinery, New York, NY, USA, 69–78. 9781450339025 https://doi.org/10.1145/2815072.2815082

[13] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Hannes Voigt, Oskar van Rest, Domagoj Vrgoč, Mingxi Wu, and Fred Zemke. 2021. Graph Pattern Matching in GQL and SQL/PGQ. https://doi.org/10.48550/ARXIV.2112.06217

[14] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. 2019. TigerGraph: A Native MPP Graph Database. https://doi.org/10.48550/ARXIV.1901.08248

[15] 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. https://www.iso.org/standard/32620.html

[16] Hanâ Lbath, Angela Bonifati, and Russ Harmer. 2021. Schema Inference for Property Graphs. In *EDBT 2021 - 24th International Conference on Extending Database Technology (EDBT)*. Nicosia, Cyprus, 499–504. https://doi.org/10.5441/002/edbt.2021.58 Short Paper.

[17] Michael Meier, Michael Schmidt, Fang Wei, and Georg Lausen. 2013. Semantic query optimization in the presence of types. *J. Comput. System Sci.* 79, 6 (2013), 937–957. 0022-0000 https://doi.org/10.1016/j.jcss.2013.01.010 JCSS Foundations of Data Management.