On Property Constraints in PG-Schema

Dominik Tomaszuk
d.tomaszuk@uwb.edu.pl
University of Bialystok
Bialystok, Poland

Jose Emilio Labra Gayo
labra@uniovi.es
University of Oviedo
Oviedo, Spain

Abstract

Property graphs have become a fundamental data model for representing complex, interconnected data across diverse domains. Their flexibility, achieved by annotating nodes and edges with properties, enables rich and expressive data modeling. However, this flexibility often comes at the cost of data consistency and quality, as many property graph systems lack robust schema mechanisms. To address this gap, we present PG-Schema-PC (PG-Schema for Property Constraints), a principled extension of the PG-Schema language tailored to express constraints over property sets. PG-Schema-PC introduces constructs for defining structural, cardinality, and range constraints, allowing schema designers to specify alternative property groupings, control property multiplicity, and validate value-level conditions. We formalize the abstract grammar and semantics of PG-Schema-PC and demonstrate its capabilities through practical examples inspired by real-world use cases.

CCS Concepts

• **Information systems → Graph-based database models**; *Data model extensions*; Database management system engines.

Keywords

Property Graphs, Graph Schema Languages, PG-Schema, Property Constraints, Graph Databases

ACM Reference Format:
Dominik Tomaszuk and Jose Emilio Labra Gayo. 2025. On Property Constraints in PG-Schema. In *K-CAP 2025: The Thirteenth International Conference on Knowledge Capture, December 10–12, Dayton, Ohio, USA*. ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Property graphs have emerged as a fundamental data model for representing complex, interconnected data across a wide range of domains, including social networks [19], biomedical [19], chemical [29], biological [20], construction engineering [17], financial [5], and mathematical [33]. Their intuitive structure, representing entities as nodes, relationships as edges, and additional metadata as properties on both, enables rich and flexible data modeling. However, this same flexibility presents challenges for ensuring data quality, consistency, and semantic correctness. Without rigorous schema mechanisms, property graph data can become incomplete, inconsistent, or invalid with respect to application-specific business rules.

To address these issues, schema languages for property graphs aim to specify constraints over properties, thereby enabling validation, improving data interoperability, and promoting system maintainability. Existing schema formalisms, such as PG-Schema [3], XML Schema [30], JSON Schema [34], and shape languages for RDF (e.g., SHACL [16] and ShEx [28]), provide foundational capabilities for defining types and simple constraints. However, they often lack sufficient expressive power to capture the nuanced requirements of real-world applications, especially at the property level. These requirements include support for alternative property groupings, fine-grained cardinality constraints, and value-level validation using facets such as patterns, numeric ranges, or enumerations.

Consider, for example, the task of modeling personal identity information, where an entity may provide either a single `name` or a pair of `givenName` and `familyName` properties—both representations should be accepted, but one complete representation must be present. Similarly, applications may need to enforce constraints that govern how sets of properties co-occur, or impose conditions such as mandatory fields, bounded repetitions, or allowable formats for literal values. While some schema languages (e.g., JSON Schema) offer expressive constructs like `anyOf` and `allOf`, their adaptation to the property graph paradigm remains limited or ad hoc.

In this paper, we introduce PG-Schema-PC (PG-Schema with Property Constraints), a principled extension of the PG-Schema language designed to meet these challenges. PG-Schema-PC incorporates expressive structural, cardinality, and range constraints over property sets, enabling precise schema specifications for diverse application contexts. Our approach draws inspiration from existing standards while adapting them to the specific semantics and data structures of property graphs. The key contributions of this work are as follows:

- We identify and formalize design requirements motivated by real-world use cases that necessitate expressive constraints over property graphs.
- We introduce structural constraints for grouping properties via disjunctive and conjunctive patterns, allowing alternative and composite representations.
- We extend cardinality controls to support bounded occurrences for individual properties and groups, including optionality and repetition.
- We define facet-based constraints for validating literal values, supporting patterns, numeric comparisons, length constraints, and enumerations.
- We provide a formal abstract grammar and semantics for PG-Schema-PC, and discuss practical validation strategies.

K-CAP 2025, Dayton, Ohio, USA

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

We also present the PG-Schema-PC concrete grammar [32] as well as a prototype implementation written in Rust [18].
- We position PG-Schema-PC within the broader schema language ecosystem, demonstrating how it fills critical gaps left by existing approaches.

Through PG-Schema-PC, we aim to bridge the gap between the expressive needs of real-world property graph applications and the formal constraint mechanisms available in current schema languages, offering a robust and extensible foundation for graph data validation and modeling.

The remainder of this paper is structured as follows. Section 2 introduces motivating use cases and derives the corresponding design requirements. Section 3 provides formal preliminaries, including the underlying property graph model and the PG-Schema baseline. In Section 4, we present the abstract grammar and illustrative examples of the proposed constraint mechanisms. Section 5 defines the formal interpretation of property constraint declarations over property graphs using a cardinality-aware record type system, while Section 6 discusses validation strategies. Section 7 offers a comparative analysis of PG-Schema-PC against related formalisms. Finally, Section 8 summarizes our contributions and outlines future research directions.

2 Design Requirements

This section outlines core design requirements for schema modeling, emphasizing flexibility, precision, and constraint expressiveness. Specific use cases (Subsection 2.1) and detailed requirements (Subsection 2.2) illustrate these needs clearly.

2.1 Use Cases

The following use cases highlight typical scenarios encountered when designing schemas for graph-based data structures. They reflect varying demands in terms of structural flexibility, value precision, and constraint complexity in the context of properties.

UC1 In many real-world scenarios, schema designers need to allow different ways to describe the same kind of entity. For example, one user might provide a single `name` field (e.g., "Madonna"), while another might prefer to use a combination of `givenName` and `familyName` (e.g., "John" + "Smith"). To support such variation, the schema must allow alternative property groupings, but also enforce that at least one complete and valid grouping is present. This promotes flexibility without compromising data completeness or consistency. **Related Requirements:** R1, R2, R3

UC2 Many graph-based applications require that a node includes a fixed set of core properties. For example, a product entity may need to include a product identifier, at least one descriptive label, and a category code. Each of these properties is essential, and validation must ensure they all appear together and follow type and cardinality rules. This use case emphasizes the importance of representing conjunctive constraints, ensuring that multiple required attributes co-occur. **Related Requirements:** R1, R2, R3, R5

UC3 A schema must support specifying how many times a property can occur. For example, the `name` of a person might be required exactly once, `birthDate` might be optional but limited to a single value, and `aliases` might allow multiple optional entries. These constraints are critical for precise modeling and ensuring data integrity. Validation systems must be able to enforce these rules to prevent under- or over-specification. **Related Requirements:** R2, R3

UC4 Some applications need to support highly flexible property values. For instance, in a social media graph, the `follows` property might point to a user, an organization, a topic, or even an external URL. The schema must allow this by permitting values of any type, including different node categories or literal values. This kind of generality is essential in semi-structured data models and systems that integrate heterogeneous sources. **Related Requirements:** R3

UC5 Real-world domains often require enforcing additional conditions on literal values. For example, students enrolled in adult education must be over 18, or email addresses must match a standard pattern. These are value-level constraints, also known as facets, which go beyond mere type declarations. Validation mechanisms must support numeric comparison, pattern matching (e.g., regex), and length constraints to enforce these rules directly in the schema. **Related Requirements:** R3, R4

UC6 Some cases involve not just alternatives, but repeating groups of alternatives with bounds. For instance, a multilingual user profile might contain up to two naming variants, where each variant is either a single `name` or a pair of `givenName` and `familyName`. The schema must express both the alternative forms and the cardinality over the group as a whole. This supports flexible but well-controlled modeling in multilingual, multi-regional, or versioned data. **Related Requirements:** R1, R2, R3

2.2 Requirements

This subsection explicitly defines the requirements derived from the presented use cases. Each requirement addresses specific capabilities essential for robust schema definition and validation.

R1 The schema language must allow the definition of complex structural patterns for nodes. This includes support for specifying multiple alternative configurations as well as configurations that require all properties to co-occur. Such capabilities are necessary to accurately represent flexible and compound data models.

R2 The schema language must support precise control over the number of occurrences of a property on an element. It must be possible to declare whether a property is required, optional, repeatable, or constrained to occur within specified bounds.

R3 The schema language must support the specification of the expected type of each property value. This includes basic datatypes such as strings and integers, union types, references to other nodes, and untyped or generic values when flexibility is needed.

R4 The schema language must allow for the specification of constraints on literal values. These include numeric comparisons, pattern-based validation using regular expressions,

and constraints on the length of string values, in order to enforce business rules and improve data quality.

R5 The schema language must support constraints that restrict the set of valid values for a property. This includes explicit enumerations of acceptable literals.

3 Preliminaries

This section introduces foundational concepts required to understand later formalizations. It begins by defining records, the structures used for node and edge contents and their semantics, formally defines property graphs (Subsection 3.1), and briefly presents PG-Schema (Subsection 3.2), the schema formalism later extended with property constraints.

Given a set of values $\mathcal{V}$, $\text{SET}(\mathcal{V})$ represents the set of all subsets of $\mathcal{V}$. Given a set of property names or keys $\mathcal{K}$, a record o is a finite-domain partial function $o\colon \mathcal{K} \twoheadrightarrow \text{SET}(\mathcal{V})$ that maps keys $k \in \mathcal{K}$ to sets of values $v \in \text{SET}(\mathcal{V})$. We will denote $\langle k_1 : vs_1, \ldots, k_n : vs_n\rangle$ the record with domain $\{k_1, \ldots k_n\}$ that assigns the sets of values vs_i to k_i. For simplicity, when the value is a singleton set $\{v\}$, we denote it as v. We will write $\langle\rangle$ for the empty record and $\mathcal{R}$ for the set of all records. Given two records $o_1, o_2 \in \mathcal{R}$ we define their *combination* $o_1 \oplus o_2$ as a record whose domain $dom(o_1 \oplus o_2) = dom(o_1) \cup dom(o_2)$ such that $(o_1 \oplus o_2)(k) = o_1(k) \cap o_2(k)$ for $k \in \text{dom}(o_1) \cap \text{dom}(o_2)$, $(o_1 \oplus o_2)(k) = o_1(k)$ for $k \in \text{dom}(o_1) \setminus \text{dom}(o_2)$, and $(o_1 \oplus o_2)(k) = o_2(k)$ for $k \in \text{dom}(o_2) \setminus \text{dom}(o_1)$. For sets $O_1, O_2 \subseteq \mathcal{R}$ we let $O_1 \oplus O_2$ be the set of all records of the form $o_1 \oplus o_2$.

3.1 Property Graphs

We formally define a property graph by specifying its structural components (nodes, edges, properties) with labeling and value assignment functions, assuming countable sets of labels $\mathcal{L}$, keys $\mathcal{K}$, and primitive values $\mathcal{V}$.

Definition 3.1. *A property graph is a tuple $G = (N, E, \rho, \lambda, \pi)$ where:*

(1) *N is a finite set of identifiers for nodes;*
(2) *E is a finite set of identifiers for edges such that $N \cap E = \emptyset$;*
(3) *$\rho : E \to (N \times N)$ is a total function that associates each edge with a pair of nodes (s, t) where s is the source and t is the target;*
(4) *$\lambda : (N \cup E) \twoheadrightarrow \text{SET}(\boldsymbol{L})$ is a partial function that associates each node/edge with a set of labels;*
(5) *$\pi : (N \cup E) \to \mathcal{R}$ is a function mapping nodes and edges to records.*

Following that definition, the property graph shown in Figure 1 could be the tuple $(N, E, P, \rho, \lambda, \pi)$ where:

$$N = \{n_1, n_2, n_3\}, \qquad E = \{e_1, e_2\},$$
$$\rho(e_1) = (n_1, n_2), \qquad \rho(e_2) = (n_2, n_3)$$
$$\lambda(n_1) = \{Person\}, \qquad \lambda(n_2) = \{Person, Student\},$$
$$\lambda(n_3) = \{Course\}, \qquad \lambda(e_1) = \{knows\},$$
$$\lambda(e_2) = \{enrolled\},$$
$$\pi(n_1) = \langle name : alice, age : 23\rangle, \qquad \pi(n_2) = \langle name : bob\rangle,$$
$$\pi(n_3) = \langle name : Algebra\rangle, \qquad \pi(e_1) = \langle since : 2020\rangle,$$
$$\pi(e_2) = \langle start : 2024, end : 2025\rangle,$$

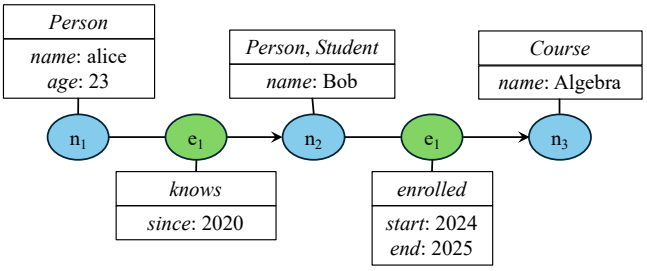

Figure 1: Property Graph example

3.2 PG-Schema

PG-Schema [3] provides mechanisms for specifying node and edge types, which are combined into graph types, allowing precise and flexible schema definitions.

PG-Schemas consist of two parts: PG-Types and PG-Keys. PG-Types specifies node and edge types describing the allowed combinations of labels and contents; as well as the types of sources and targets for edges. PG-Keys [2] describes constraints on the typed data, specifying integrity constraints such as keys and more advanced constraints.

Example 3.1. *The following example defines a simple PG-Schema for an educational graph.*

```
CREATE GRAPH TYPE Course {
(personType: Person {
  name: STRING, OPTIONAL age: INT })
(studentType: Person & Student {
  name STRING, age INT })
(courseType: Course) { name STRING })
(:personType OPEN)
  -[:knows { since DATE} ]->
    (:personType OPEN)
(:studentType)
  -[:enrolled { start DATE, OPTIONAL end DATE } ]->
    (:courseType)
}
```

A PG-Schema is defined as a set of nodes and edge type declarations, where each type declaration contains a specification for the possible labels and another one for the content. For label specifications, we adopt an `OPEN` modifier that controls openness of label matching: `(:personType OPEN)` on both sides of an edge `:knows` means that the incident nodes must conform to `personType` yet may carry additional labels (e.g., `Person & Student`), because `OPEN` relaxes matching over label sets; without `OPEN`, only exactly the label sets generated by `personType` are admissible (no supersets). In [3], the content declaration is defined by basic types, while in this paper, we extend it to more complex types with constraints and value sets.

More formally, a PG-Schema is a tuple $S = (N_S, E_S, n_S, e_S)$ where

- N_S and E_S are disjoint finite sets of node and edge type names;
- $n_S : N_S \to F$ maps node type names to a label-property specification F;
- $e_S : E_S \to F \times F \times F$ maps edge type names to three label-property specifications le_1, le_2, le_3 where le_1 specifies the source of the edge, le_2 the edge, and le_3 the target.

Label-property specifications F are expression built from labels ℓ and node type names σ using operators ?, &, and |, followed by an optional keyword OPEN and an optional content description r.

The semantics of PG-Schema is defined in terms of formal base types unraveling the label-property specifications which are assumed to be non-recursive.

Definition 3.2 (Formal base type). *A pair (L, R), where $L \subseteq \mathcal{L}$ and $R \subseteq \mathcal{R}$ is a* formal base type.

A formal base type represents the type associated to nodes and edges. It is a set of labels and a set of possible records. We write $\mathcal{T}$ for the set of all formal base types.

Example 3.2. *For the expression:*

```
userType: Person {
 name STRING, OPTIONAL age Int
}
```

The formal base type userType will be the pair (L, R) where $L = \{Person\}$ and $R = \{\{name : String\}, \{name : String, age : Int\}\}$

Definition 3.3 (Formal Graph Type). *A tuple $S = (N_S, E_S, \nu_S, \eta_S)$ where*

- *N_S and E_S are disjoint finite sets of node and edge type names;*
- *$\nu_S : N_S \to 2^{\mathcal{T}}$ maps node type names to sets of formal base types;*
- *$\eta_S : E_S \to 2^{\mathcal{T}\times\mathcal{T}\times\mathcal{T}}$ maps edge type names to sets of triples of formal base types: one for the source node, one for the edge itself, and one for the target node is a* formal graph type.

For brevity, the elements of N_S and E_S are called node and edge types, rather than node and edge type names.

Definition 3.4 (Conformance). *Let $G = (N_G, E_G, \lambda_G, \rho_G, \pi_G)$ be a property graph and $S = (N_S, E_S, \nu_S, \eta_S)$ be a formal graph type. A node $v \in N_G$* conforms *to a node type $\tau \in N_S$ if it conforms to a formal base type in $\nu_S(\tau)$.*

An edge $e \in E_G$ conforms *to an edge type $\sigma \in E_S$ if for the pair $(v_1, v_2) = \rho_G(e)$ there is a triple $(t_1, t, t_2) \in \eta_S(\sigma)$ such that v_1 conforms to t_1, e conforms to t, and v_2 conforms t_2.*

A property graph G conforms *to a formal graph type S if every element in G conforms to at least one type in S.*

Assume that ν_S is already defined over all node type names σ used in F (the base case is when τ is defined as a base node type). The expression F defines the family $[\![F]\!] \subseteq \mathcal{T}$ of formal base types allowed for type τ. Intuitively speaking, F describes how the allowed formal base types can be generated, starting from the simplest ones.

The semantics is defined recursively for all subexpressions of F:

$$[\![\ell]\!] = \{\tau_\ell\} \qquad [\![\sigma]\!] = \nu_S(\sigma)\,,$$
$$[\![F_1?]\!] = [\![F_1]\!] \cup \{\tau_\emptyset\} \quad [\![F_1 \mid F_2]\!] = [\![F_1]\!] \cup [\![F_2]\!]\,.$$
$$[\![F_1 \,\&\, F_2]\!] = \{(L_1 \cup L_2, R_1 \oplus R_2) \mid (L_i, R_i) \in [\![F_i]\!] \text{ for } i = 1, 2\}\,,$$
$$[\![F_1 \text{ OPEN}]\!] = \{(L, R) \mid \exists L' \subseteq L \text{ such that } (L', R) \in [\![F_1]\!]\}\,,$$
$$[\![F_1\, r]\!] = \{(L, R \oplus [\![r]\!]) \mid (L, R) \in [\![F_1]\!]\}\,.$$

In [3], the record content is defined mainly as a basic subset of values while in this paper we introduce a richer content model which will be presented in the following sections. For brevity, we omit the semantic definition of edge types which follows the same approach and is presented in that paper.

4 Property Constraints

This section presents property constraints as an extension to PG-Schema, introducing an abstract grammar for PG-Schema-PC and covering structural, cardinality, and range constraints.

4.1 Structural Constraints

Structural constraints at the level of properties are features relevant to schema design. While many schema languages focus on applying such constructs to nodes or types, their application to properties increases flexibility by allowing multiple alternative representations of data within a single definition. For example, a schema may permit a property to take either an integer or a float as its value, or require a property to satisfy multiple typing conditions simultaneously.

OneOf: At least one of the listed alternatives must be satisfied. In practice, the `OneOf` pattern enables mutually exclusive alternatives within a single property block, permitting the instance author to choose the most appropriate representation from a controlled set. Conceptually, it corresponds to a logical disjunction over the declared alternatives, which must be satisfied by at least one (and potentially exactly one, depending on additional cardinality settings). Alternatives are separated by `|`, and round brackets group property blocks.

Example 4.1 (User-type identification alternatives). *This example demonstrates a schema in which a node of type* `userType` *must satisfy at least one of two alternative property groupings (by using round braces): either the single property* `name`*, or the combination of* `givenName` *and* `familyName`*.*

```
CREATE NODE TYPE userType : User {
  ( name STRING ) |
  ( givenName STRING, familyName STRING )
}
```

EachOf: All listed constraints must be satisfied simultaneously. The `EachOf` pattern requires all specified properties (or constraints) to be fulfilled together. The following example shows a definition of a node type `productType`, where multiple constraints apply at once. Conjuncts are separated by `,`, and round brackets group property blocks.

Example 4.2 (Product-type mandatory properties). *This example defines a node type* `productType` *that must include all three listed properties simultaneously:* `productId` *(string),* `label` *(string), and* `category` *(string or integer).*

```
CREATE NODE TYPE productType : Product {
    ( productId STRING ) ,
    ( label STRING ) ,
    ( category STRING | INT )
}
```

This schema specifies that the property `productId` is required, and must be a string; the property `label` is required and must be a string; the property `category` is required and may be either a string or an integer, enabling flexible textual or numeric categorization. The `EachOf` construct mandates simultaneous fulfillment of all properties, while `OneOf` allows selection among alternatives,

optionally constrained by cardinality. These constraints enhance schema expressiveness, particularly suited for heterogeneous data environments.

4.2 Cardinality Constraints

Cardinality constraints, in contrast, define how many times a property type may participate in a given element. These constraints capture lower and upper bounds for relationship participation, enabling specification of concepts such as optional properties, and fixed-size collections. Cardinality is crucial for expressing participation constraints and enforcing structural consistency within graph data.

Defining cardinalities for specific properties. The following example demonstrates how to declare different cardinalities for properties associated with a node type `personType`. This allows for precise specification of which properties are mandatory, optional, or repeatable.

Example 4.3 (Person-type property cardinalities). *This example shows a node type* `personType` *with three properties whose occurrences are constrained respectively to exactly one, zero-or-more, and optional instances.*

```
CREATE NODE TYPE personType : Person {
  name STRING,
  aliases LIST STRING {0,*},
  OPTIONAL birthDate DATE
}
```

The semantics of each declaration are as follows. The property `name` is mandatory and must occur exactly once; the absence of an explicit multiplicity on the property implies a default of `1..1`, indicating a required singleton. The property `aliases` is a single property whose value is a list of strings; the annotation `{0,*}` specifies the list cardinality (element multiplicity), meaning the list may contain zero or more elements. The property `birthDate` is optional, i.e., the property may be omitted. Note that for properties marked `OPTIONAL`, the default multiplicity is `0..1` (optional singleton), not `1..1`.

This expressiveness allows schema designers to balance between flexibility and structure, ensuring that data adheres to intended usage patterns while supporting real-world variability.

4.3 Range Constraints

Range constraints refer to restrictions placed on the admissible values of properties in a schema. These restrictions are typically expressed using data type facets such as minimum or maximum length, numeric bounds, regular expressions, or enumerated values. In XML Schema [30], these are specified through constructs like `minInclusive`, `maxLength`, `pattern`, or `enumeration`, allowing for precise control over property values. In RDF-based languages such as SHACL [16] and ShEx [28], similar constraint mechanisms are provided through dedicated vocabulary constructs. JSON Schema [34], likewise, supports value constraints on numeric types, strings, and structured data.

ANY datatype: allowing arbitrary values. The following example defines a node type `userType` in which the `follows` property is permitted to refer to a value of any data type:

Example 4.4 (Untyped `follows` property). *This example demonstrates a node type* `userType` *whose* `follows` *property can accept a value of any datatype, providing maximum flexibility at the cost of static validation.*

```
CREATE NODE TYPE userType : User {
  name STRING,
  follows ANY
}
```

Here, the use of `ANY` signals that the `follows` property is untyped or polymorphic. This is analogous to the dot `.` wildcard in ShEx, which matches any value irrespective of its datatype. While this provides maximal flexibility, it disables most forms of static validation and should be used with caution in controlled data environments. Importantly, `ANY` is an explicit, intentional declaration of permissiveness that is semantically distinct from omitting a datatype under a closed-world assumption (CWA). In PG-Schema-PC, the absence of a datatype (or of a property specification) *does not* license arbitrary values; rather, it indicates that such values are not part of the declared content and may be rejected unless admitted by other mechanisms (e.g., separate declarations or openness policies). By contrast, declaring `ANY` commits the schema to accept values of *any* datatype for the property, including heterogeneous mixtures, while other orthogonal constraints (e.g., cardinality or structural constraints) may still apply.

Facets: applying constraints to restrict value range or format. The next example imposes a numeric constraint on an integer property using a `CHECK` condition, which refers to the restriction mechanism known from SQL [13]:

Example 4.5 (Age constraint with `CHECK` facet). *This example defines a node type* `adultStudentType` *whose* `age` *property must be an integer strictly greater than 18, enforced via a* `CHECK` *facet.*

```
CREATE NODE TYPE adultStudentType {
    age INT CHECK age > 18
}
```

This schema fragment defines a node type `adultStudentType` where the property `age` must be an integer strictly greater than 18. The `CHECK` constraint enforces a lower bound and guarantees that only adults (according to a given jurisdiction) are accepted.

A different form of facet constraint is shown below, using a regular expression to validate the format of email addresses:

Example 4.6 (Email format validation with `REGEX` facet). *This example defines a node type* `emailUserType` *whose* `email` *property must match a basic e-mail address pattern expressed as a regular expression.*

```
CREATE NODE TYPE emailUserType {
    email STRING CHECK REGEX '^[\w.-]+@[\w.-]+\.\w+$'
}
```

This definition of `emailUserType` introduces a constraint on the `email` property such that its value must match a specific regular expression pattern. The expression approximates a basic email address format, requiring a sequence of alphanumeric characters (possibly including dots or hyphens), followed by an `@` symbol, a domain name, and a top-level domain. While this pattern is simplified and does not fully capture the complexity of email addresses,

it demonstrates the application of pattern-based validation to constrain string values.

Example 4.7 (Username length constraint with CHECK facet). *This example defines a node type* `loginUser` *whose* `username` *property must have a length between 3 and 32 characters, enforced via a* CHECK *facet.*

```
CREATE NODE TYPE loginUser {
  username STRING CHECK length(username) BETWEEN 3 AND 32
}
```

This schema fragment ensures that `username` values are neither too short nor excessively long. The use of `BETWEEN` makes the bounds inclusive (i.e., lengths of 3 and 32 are both valid). The constraint is complementary to pattern-based checks (e.g., `REGEX`) and can be combined with them when more precise control over permitted strings is required.

In the above examples, facets are applied to restrict the domain of admissible values for a property, thereby enabling strong guarantees about the integrity, correctness, and semantics of graph data. These mechanisms are especially important in systems that aim to interoperate across heterogeneous datasets or enforce regulatory compliance.

5 PG-Schema-PC formal definition

In this section, we formally define the syntax and semantics of PG-Schema-PC, a PG-Schema extension supporting complex property constraints, with an abstract grammar for optional, conjunctive, and disjunctive declarations, and semantics based on cardinality-aware record types.

5.1 Abstract grammar

In this section we present an abstract grammar for PG-Schema-PC which formally captures the syntax required to define complex property constraints. The concrete grammar of the PG-Schema language extension is presented in [32][1]. The abstract grammar is specified in terms of property-value specifiers *pvs* which are defined as a property-value definition *pv* followed by an optional *OPEN* declaration.

pvs	::=	$\{\ pv\ \}\ [OPEN]$	property-value specifier
pv	::=	$[OPTIONAL]\ p\ ts$	optional property p with ts
	|	$pv_1\ ,\ pv_2$	Each of pv_1 and pv_2
	|	$pv_1 \mid pv_2$	One of pv_1 and pv_2
	|	ϵ	Empty content
ts	::=	$t\ [card]$	type t with cardinality $card$
t	::=	$t_1\ \&\ t_2$	conjunction of t_1 and t_2
	|	$t_1 \mid t_2$	disjunction of t_1 and t_2
	|	$\Delta\ [\text{cond}]$	Datatype Δ satisfying cond
$card$	::=	$\{\ n, max\ \}$	$n \in \mathbb{N}$
max	::=	$n \mid *$	

Type specifiers are defined as single types followed by an optional cardinality specification. Single types can be primitive data types STRING, INT, DATE, etc. which are represented by Δ followed by an optional condition cond or union/intersection of those types. The cardinality *card* specifies the number of allowed elements of the value set. If it is absent, it is assumed that the value set is a singleton. cond is a boolean expression like > 18. In t, & binds more strongly than |; round brackets allowed to enforce order.

[1]There are also available the railroad diagrams on https://domel.github.io/pg-schema-pc/rr-diagrams.html

5.2 PG-Schema-PC Semantics

The semantics of PG-Schema-PC follows the semantics of PG-Schema defined in [3] and extends the definition of the content using record types over sets of values.

Definition 5.1 (Record type). *A* record type *is a record* $\tau : \mathcal{K} \nrightarrow SET(\mathcal{V})$ *where the values are basic types defined as subsets* $\Delta \in SET(\mathcal{V})$ *of the set of primitive values* $\mathcal{V}$.

We assume a set of builtin primitive subsets of $\mathcal{V}$ which can be defined by name like *Int*, *String*, *Date*, etc. and some built-in constraints which are expressed as boolean combination of numerical comparisons, regular expressions, etc. We will use the notation Δ^n_m to describe subsets $\Delta' \in SET(\mathcal{V})$ such that the number of elements $|\Delta'|$ is between m and n and $\forall x \in \Delta',\ x \in \Delta$. In case $n = *$, the upper bound is unlimited. By default, Δ will be assumed as Δ^1_1 in the record type definitions.

Example 5.1. *The record type* $\tau = \langle name : String, age : Integer \wedge > 18, aliases : String^*_1 \rangle$ *denotes records with a property name that can have a singleton value of type* String*, a property age with an integer value greater that 18 and one or more aliases of type* String.

Definition 5.2 (Record type conformance). *A record* o *conforms with record type* τ*, denoted as* $o \vDash \tau$ *if* $dom(o) \subseteq dom(\tau)$ *and for all* $k_i \in \text{dom}(o)$*, all the values* $v \in o(k_i)$ *conform to* $\tau(k_i)$*, i.e.* $v \in \tau(k_i)$

The property-value specifications defined in section 5.1 will be compiled to formal base types according to the following rules:

$$\llbracket pv_1 \mid pv_2 \rrbracket = \llbracket pv_1 \rrbracket \cup \llbracket pv_2 \rrbracket \qquad \llbracket pv_1, pv_2 \rrbracket = \llbracket pv_1 \rrbracket \oplus \llbracket pv_2 \rrbracket$$
$$\llbracket p\ ts \rrbracket = \langle p : \llbracket ts \rrbracket \rangle \qquad \llbracket \text{OPTIONAL } p\ ts \rrbracket = \langle p : \llbracket ts \rrbracket \rangle \cup \langle\rangle$$
$$\llbracket \epsilon \rrbracket = \langle\rangle$$

The semantics of type-specifications ts is defined as:

$$\llbracket t\ card \rrbracket = \llbracket t \rrbracket^n_m \text{ where } card = \{m, n\} \quad \llbracket t \rrbracket = \llbracket t \rrbracket^1_1$$
$$\llbracket t_1 \& t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket \qquad \llbracket t_1 | t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$$
$$\llbracket \Delta \rrbracket = \Delta \qquad \llbracket \Delta\ \text{cond} \rrbracket = \{v \in \Delta | cond(v) = \text{true}\}$$

The content r of a node or edge conforms to a property-value specification pvs ($r \vDash pvs$) if there is a type record $\tau \in \llbracket pvs \rrbracket$ such that $r \vDash \tau$. Additionally, if *OPEN* is absent in pvs, we also require that $dom(r) = dom(\tau)$.

6 Validation of PG-Schema-PC

The crucial task related to PG-Schema-PC is validation; that is, determining whether a given property graph satisfies a given constraint. Additionally, to characterize the properties of a relation within a dataset, some scenarios can be conveniently simulated with PG-Keys [2]. Below, we illustrate how selected constraints can be rendered in a PG-Keys–like notation augmented with lightweight extensions. Importantly, while these examples are faithful to the intent and design principles of PG-Keys, they go beyond the

core language: the added constructs have no direct counterpart in the official grammar or semantics. In PG-Schema-PC, we formalize the meaning of these constructs (see Section 5.2) and, moreover, provide a substantially simpler native syntax that subsumes the wider use cases (see Section 5.1). By design, the core PG-Keys formalism does not support structural constraints at the property level. Its cardinality constraints cover only the presence/absence case (i.e., max 0/1), and range constraints over property values are not part of the core language.

Structural Constraints. The fragment concerning the `category` property in Example 4.2 can be represented in PG-Keys:

```
FOR (p:productType) MANDATORY SINGLETON c WITHIN
p.category WHERE type(p.category) IN ['STRING','INT']
```

Example 4.1 can be expressed like:

```
FOR (u:userType)
MANDATORY k WITHIN
 ( (u) WHERE EXISTS(u.name)
  RETURN u.name AS k )
  UNION  ALL
 ( (u) WHERE EXISTS(u.givenName) AND EXISTS(u.familyName)
  RETURN [u.givenName , u.familyName] AS k )
```

Cardinality Constraints. The fragment concerning the `aliases` property in Example 4.3 can be represented in PG-Keys:

```
FOR (p:personType) MANDATORY l COUNT 0..* OF
WITHIN p.aliases
```

Range Constraints. The fragment concerning the `age` property in Example 4.5 can be represented in PG-Keys:

```
FOR (s:adultStudentType) MANDATORY SINGLETON a
    WITHIN s.age WHERE a > 18
```

Example 4.6 looks similar:

```
FOR (x:emailUserType) MANDATORY SINGLETON e
    WITHIN x.email WHERE e =~ '^[\\w.-]+@[\\w.-]+\\.\\w+$
     '
```

7 Relationship to other paradigms

PG-Schema integrates traditional conceptual modeling, tree-structured validation, and graph-native paradigms, unifying declarative expressiveness and structural rigor to facilitate interoperability in hybrid data ecosystems.

7.1 Overview of existing schema formalisms

Conceptual data models. UML [24] remains the standard visual notation for object-oriented conceptual modeling, with class diagrams and OCL constraints expressing rich semantics. ORM2 [11] employs logic-based predicates and roles instead of attributes, facilitating verbalisation of diagrams into controlled natural language, ideal for precise yet accessible business rules. EER modeling [9] enhances the ER model with inheritance, generalisation, and category constructs, supporting relational and object-relational implementations.

Tree-structured data. XML DTD [7], limited in datatypes, led to XML Schema 1.1 [30], which introduced complex types, rich datatypes, keys, and type derivation under deterministic validation constraints. RELAX NG [12] defines regular tree languages with grammar-oriented XML and compact syntaxes, supporting interleaving and composability. JSON Schema [34] validates JSON structures using keywords for object properties, array constraints, recursion via `$ref`, and combinators such as `allOf` and `oneOf`.

RDF formalisms. OWL2 [21], based on the description-logic SROIQ, supports disjointness, equivalence, complex operators, and qualified cardinalities, with profiles (EL, QL, RL) balancing expressiveness and computational tractability. For prescriptive integrity constraints, SHACL [16] groups constraints into shapes targeting specific nodes, while ShEx [28] offers regular-expression-based node constraints with XML-inspired closed/open-content semantics and Kleene closures. It has also added an inheritance mechanism [6] which enables more reusability between data models. Our approach for PG-Schema-PC is inspired by ShEx, although collisions between repeated properties are handled by record combinations. ProGS [26] is a SHACL-inspired shapes language for property graphs that adds edge-shapes, qualified edge counts, and constraints over node- and edge-attached properties. In RDF deployments, property-graph features can be emulated by materializing node properties as ordinary triples and edge properties via different methods of reification.

Existing graph technologies. SQL standard ISO/IEC9075-16:2023 (SQL/PGQ) [13] integrates property-graph querying within relational algebra, ensuring ACID and optimized analytics. ISO/IEC 39075:2024 Graph Query Language (GQL) [14], inspired by Cypher, supports schema-aware graph operations independently of SQL. GraphQL's Schema Definition Language (SDL) [10] defines executable schemas with object types, fields, arguments, and introspection, enabling evolvable graph APIs.

Graph database systems. JanusGraph [15] offers distributed graph storage with Gremlin and ACID compliance. Neo4j [23] provides native storage and Cypher querying with analytics integration. ArangoDB [4] supports hybrid document and graph models queried via AQL. TigerGraph [31] focuses on distributed analytics using GSQL for multi-hop traversal. AgensGraph [1] extends PostgreSQL with Cypher for integrated relational-graph data. DataStax Graph [8] embeds Gremlin in Cassandra for global scale. Nebula Graph [22] employs a shared-nothing architecture supporting GQL. Oracle Database23ai[25] introduces SQL/PGQ for relational-backed property graphs. Sparksee [27] compresses attributed graphs efficiently for constrained environments.

7.2 Support of the features

We summarize important differences between PG-Schema-PC and state-of-the-art schema formalisms and systems (Table 1).

Range constraints are widely supported. SHACL, ShEx, OWL, XML Schema, RELAX NG, SQL, and JSON Schema provide native support. SQL-based systems like Oracle PGQL, OrientDB, and PostgreSQL with PGQ support them via typed columns and `CHECK` constraints. Systems such as AgensGraph, ArangoDB, and DataStax offer indirect support using JSON Schema or embedded logic. In contrast, graph databases including Neo4j, JanusGraph, TigerGraph, and Nebula Graph lack native support, emphasizing flexibility. UML and ORM2 support basic constraints but require external

non-standardized extensions for advanced features like regex patterns[2]. Enhanced ER offers basic value constraints but lacks detailed numeric or regex restrictions. GraphQL SDL supports basic scalars but lacks built-in advanced constraints.

Cardinality constraints are common among conceptual models and schema languages. Enhanced ER, ORM2, UML, XML Schema, JSON Schema, RELAX NG, SQL, and JanusGraph explicitly define multiplicities. XML Schema uses `minOccurs` and `maxOccurs`, JSON Schema employs fields like `minItems`, `maxItems`, and JanusGraph includes specific multiplicity constructs. SQL-based systems enforce cardinalities through uniqueness and foreign keys. ArangoDB partially relies on JSON Schema fields. OWL 2's qualified cardinality constraints are enforced via logical entailment and inconsistency detection, so they mainly serve ontology consistency checking under OWA, unlike SHACL/ShEx, which provide explicit, closed-world validation reports. Oracle PGQ uses basic keys but lacks detailed cardinality bounds. GQL does not formally specify property occurrence constraints. ProGS supports qualified counting over *sets* of property values, aligning with SHACL/ShEx semantics. Specifically, in SHACL, ShEx, and ProGS, counting is performed over a *set* of value nodes: RDF is a set of triples (no duplicates, no order), and in SHACL and ProGS, distinct values rather than list elements with order.

Structural constraints on properties are explicitly supported by JSON Schema, XML Schema, RELAX NG, SHACL, and ShEx through combinators and compound constraints. UML models structural variability via inheritance and polymorphism but lacks direct alternative property typing. ORM2 and Enhanced ER support subtype hierarchies but do not directly support alternative types per property. DTDs offer alternative element structures but not alternative attribute data types. OWL 2 lacks disjunctive data property types. GraphQL supports structural alternatives via union types for objects, not scalar properties. Most graph databases, including Neo4j, JanusGraph, TigerGraph, and Nebula Graph, do not natively support structural constraints. ArangoDB, using JSON Schema, is one of few offering expressive structural property constraints, highlighting the rarity and potential value of this feature.

Even when other formalisms cover range, cardinality, and structural constraints in full, their validation models and data abstractions do not transfer directly to property graphs, where constraints must range over node- and edge-attached properties, multi-edges, and label- or path-sensitive structure. By design, PG-Schema-PC specializes these constraint families to the property-graph setting, preserving PG-Schema typing while enabling closed-world, instance-level checks over edge properties and node properties

Approach	Range	Cardinality	Structural
UML	½	✓	½
ORM2	½	✓	½
Enhanced ER	½	✓	½
JSON Schema	✓	✓	✓
XML Schema	✓	✓	✓
DTD	×	×	½
RELAX NG	✓	✓	✓
SHACL	✓	½	✓
ShEx	✓	½	✓
OWL 2	✓	½	×
ProGS	✓	½	✓
SQL	✓	✓	×
GQL	×	×	½
GraphQL (SDL)	½	×	½
JanusGraph	×	✓	×
Neo4j	×	×	×
ArangoDB	½	½	✓
TigerGraph	×	×	×
AgensGraph	½	×	×
DataStax	½	×	×
Nebula Graph	×	×	×
Oracle/PGQ	½	½	×
Sparksee	×	×	½
PG-Schema-PC	✓	✓	✓

Table 1: Support for three classes of constraints in selected schema languages and technologies. ✓ – full native support; ½ – partial or indirect support; × – no meaningful support.

8 Conclusion

We introduced PG-Schema-PC, a principled extension of PG-Schema designed to enhance the expressiveness and robustness of property graph schemas. PG-Schema-PC addresses existing gaps at the property level by integrating structural, cardinality, and range constraints essential for modeling complex scenarios.

Our framework enables schema designers to specify alternative and conjunctive property groupings, enforce cardinalities, and validate values using patterns, numeric constraints, and enumerations. Through illustrative examples and formal definitions, we demonstrated the flexibility and utility of our approach in realistic use cases.

We positioned PG-Schema-PC within the landscape of existing formalisms, highlighting its capabilities compared to XML Schema, JSON Schema, SHACL, ShEx, and contemporary graph databases. Our analysis shows how PG-Schema-PC bridges expressiveness gaps, offering powerful tools for data quality, integrity, and semantic correctness.

Future work includes developing practical validation tools for automated instance checking, empirical evaluation of validation performance, and exploring advanced schema composition and inheritance to manage schema evolution, property collisions, and formalizing the expression of negation within property constraints.

[2]External extensions (e.g., OCL/regular expressions) exist, but there is no standard and no portable tools.

References

[1] AgensGraph 2017. AgensGraph. URL: https://bitnine.net/agensgraph.
[2] Renzo Angles and et al. 2021. PG-Keys: Keys for Property Graphs. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD 2021)* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2423–2436. doi:10.1145/3448016.3457561

[3] Renzo Angles and et al. 2023. PG-Schema: Schemas for Property Graphs. *Proc. ACM Manag. Data* 1, 2, Article 198 (June 2023), 25 pages. doi:10.1145/3589778

[4] ArangoDB 2012. ArangoDB. URL: https://www.arangodb.com/.

[5] Luigi Bellomarini, Lorenzo Bencivelli, Claudia Biancotti, Livia Blasi, Francesco Paolo Conteduca, Andrea Gentili, Rosario Laurendi, Davide Magnanimi, Michele Savini Zangrandi, Flavia Tonelli, et al. 2022. Reasoning on Company Takeovers: From Tactic to Strategy. *Data & Knowledge Engineering* 141 (2022), 102073. doi:10.1016/j.datak.2022.102073

[6] Iovka Boneva, Jose Emilio Labra Gayo, Eric Prud'hommeaux, Katherine Thornton, and Andra Waagmeester. 2025. Shape Expressions with Inheritance. In *The Semantic Web: 22nd European Semantic Web Conference, ESWC 2025, Portoroz, Slovenia, June 1–5, 2025, Proceedings, Part I* (Portoroz, Slovenia). Springer-Verlag, Berlin, Heidelberg, 481–499. doi:10.1007/978-3-031-94575-5_26

[7] Tim Bray, John Cowan, François Yergeau, Michael Sperberg-McQueen, Jean Paoli, and Eve Maler. 2006. *Extensible Markup Language (XML) 1.1 (Second Edition).* W3C Recommendation. W3C. https://www.w3.org/TR/2006/REC-xml11-20060816/

[8] DataStaxGraph 2016. DataStax Graph. URL: https://www.datastax.com/products/datastax-graph.

[9] Ramez Elmasri and Shamkant B. Navathe. 2015. *Fundamentals of Database Systems (7th Edition).* Pearson.

[10] GraphQL Foundation. 2021. GraphQL Specification (October 2021 Edition). Online Specification, GraphQL Working Group. URL: https://spec.graphql.org/October2021/.

[11] Terry Halpin. 2018. Object-role modeling. In *Encyclopedia of Database Systems*. Springer, 2540–2546.

[12] International Organization for Standardization. 2008. *Information technology – Document Schema Definition Language (DSDL) – Part 2: Regular-grammar-based validation – RELAX NG.* Technical Report ISO/IEC 19757-2:2008. International Organization for Standardization.

[13] International Organization for Standardization. 2023. *Information technology – Database languages – SQL – Part 16: Property Graph Queries (SQL/PGQ).* Technical Report ISO/IEC 9075-16:2023. International Organization for Standardization.

[14] International Organization for Standardization. 2024. *Information technology – Database languages – GQL.* Technical Report ISO/IEC 39075:2024. International Organization for Standardization.

[15] JanusGraph 2017. JanusGraph. URL: https://janusgraph.org/.

[16] Holger Knublauch and Dimitris Kontokostas. 2017. *Shapes Constraint Language (SHACL).* W3C Recommendation. W3C. https://www.w3.org/TR/2017/REC-shacl-20170720/.

[17] Vishal Kumar and Evelyn Ai Lin Evelyn Teo. 2021. Exploring the application of property graph model in visualizing COBie data. *Journal of facilities management* 19, 4 (2021), 500–526.

[18] Jose Emilio Labra-Gayo and Dominik Tomaszuk. 2025. *PGSchema_PC.* doi:10.5281/zenodo.16738599

[19] Patricia E. Nalwoga Lutu. 2019. Using Twitter Mentions and a Graph Database to Analyse Social Network Centrality. In *Proceedings of the 2019 6th International Conference on Soft Computing & Machine Intelligence (ISCMI).* IEEE, 155–159. doi:10.1109/ISCMI47871.2019.9004313

[20] Ilya Mazein, Adrien Rougny, Alexander Mazein, Ron Henkel, Lea Gütebier, Lea Michaelis, Marek Ostaszewski, Reinhard Schneider, Venkata Satagopam, Lars Juhl Jensen, Dagmar Waltemath, Judith A H Wodke, and Irina Balaur. 2024. Graph databases in systems biology: a systematic review. *Briefings in Bioinformatics* 25, 6 (11 2024), bbae561. doi:10.1093/bib/bbae561

[21] Boris Motik, Peter F. Patel-Schneider, and Bernardo Cuenca Grau. 2012. *OWL 2 Web Ontology Language Direct Semantics (Second Edition).* W3C Recommendation. W3C. https://www.w3.org/TR/2012/REC-owl2-direct-semantics-20121211/

[22] NebulaGraph 2019. NebulaGraph. URL: https://nebula-graph.io/.

[23] Neo4j 2007. Neo4j. URL: https://neo4j.com/.

[24] Object Management Group (OMG). 2017. *Unified Modeling Language (UML) Version 2.5.1.* Technical Report formal/17-12-05. OMG. OMG Standard, December 2017.

[25] Oracle23ai 2023. Oracle Database 23ai. URL: https://www.oracle.com/database/23ai/.

[26] Philipp Seifer, Ralf Lämmel, and Steffen Staab. 2021. Progs: Property graph shapes language. In *International Semantic Web Conference*. Springer, 392–409.

[27] Sparksee 2008. Sparksee. URL: https://sparsity-technologies.com/sparksee-graph-database-management-system/.

[28] Sławek Staworko, Iovka Boneva, Jose E. Labra Gayo, Samuel Hym, Eric G. Prud'hommeaux, and Harold Solbrig. 2015. Complexity and Expressiveness of ShEx for RDF. In *18th International Conference on Database Theory (ICDT 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 31).* Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 195–211. doi:10.4230/LIPIcs.ICDT.2015.195

[29] Łukasz Szeremeta. 2018. SDFEater: A Parser for Chemoinformatics Formats. *ChemRxiv* (September 2018). doi:10.26434/chemrxiv.7123193.v2 This content is a preprint and has not been peer-reviewed.

[30] Henry Thompson, Murray Maloney, Michael Sperberg-McQueen, David Beech, Sandy Gao, and Noah Mendelsohn. 2012. *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures.* W3C Recommendation. W3C. https://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/.

[31] TigerGraph 2017. TigerGraph. URL: https://www.tigergraph.com/.

[32] Dominik Tomaszuk and Jose Emilio Labra Gayo. 2025. *PG-Schema for Property Constraints.* doi:10.5281/zenodo.16737630

[33] Dominik Tomaszuk, Łukasz Szeremeta, and Artur Korniłowicz. 2023. MMLKG: Knowledge Graph for Mathematical Definitions, Statements and Proofs. *Scientific Data* 10 (2023), 791. doi:10.1038/s41597-023-02681-3

[34] Austin Wright, Henry Andrews, Ben Hutton, and Greg Dennis. 2022. *JSON Schema: A Media Type for Describing JSON Documents.* Internet-Draft draft-bhutton-json-schema-01. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-bhutton-json-schema/01/ Work in Progress.