PG-Schema: Schemas for Property Graphs

RENZO ANGLES, Faculty of Engineering, Universidad de Talca, Chile
ANGELA BONIFATI, Lyon 1 University & Liris CNRS, France
STEFANIA DUMBRAVA, ENSIIE & SAMOVAR - Institut Polytechnique de Paris, France
GEORGE FLETCHER, Eindhoven University of Technology, Netherlands
ALASTAIR GREEN, LDBC, UK
JAN HIDDERS, Birkbeck, University of London, UK
BEI LI, Google, USA
LEONID LIBKIN, University of Edinburgh, UK and RelationalAI & ENS, PSL University, France
VICTOR MARSAULT, LIGM, Université Gustave Eiffel, CNRS, France
WIM MARTENS, University of Bayreuth, Germany
FILIP MURLAK, University of Warsaw, Poland
STEFAN PLANTIKOW, Neo4j, Germany
OGNJEN SAVKOVIĆ, Free University of Bozen-Bolzano, Italy
MICHAEL SCHMIDT, Amazon Web Services, USA
JUAN SEQUEDA, data.world, USA
SŁAWEK STAWORKO, RelationalAI, USA and Univ. Lille, CNRS, UMR 9189 CRIStAL, France
DOMINIK TOMASZUK, University of Bialystok, Poland
HANNES VOIGT, Neo4j, Germany
DOMAGOJ VRGOČ, University of Zagreb, Croatia and PUC Chile, Chile
MINGXI WU, TigerGraph, USA
DUŠAN ŽIVKOVIĆ, Integral Data Solutions, UK

Property graphs have reached a high level of maturity, witnessed by multiple robust graph database systems as well as the ongoing ISO standardization effort aiming at creating a new standard Graph Query Language (GQL). Yet, despite documented demand, schema support is limited both in existing systems and in the first

Authors' addresses: Renzo Angles, Faculty of Engineering, Universidad de Talca, Curicó, Chile, rangles@utalca.cl; Angela Bonifati, Lyon 1 University & Liris CNRS, Villeurbanne, France, angela.bonifati@univ-lyon1.fr; Stefania Dumbrava, ENSIIE & SAMOVAR - Institut Polytechnique de Paris, Paris, France, stefania.dumbrava@ensiie.fr; George Fletcher, Eindhoven University of Technology, Eindhoven, Netherlands, g.h.l.fletcher@tue.nl; Alastair Green, LDBC, London, UK, alastair@acm.org; Jan Hidders, Birkbeck, University of London, London, UK, j.hidders@bbk.ac.uk; Bei Li, Google, Mountain View, USA, bei@google.com; Leonid Libkin, University of Edinburgh, Edinburgh, UK and RelationalAI & ENS, PSL University, Paris, France, l@libk.in; Victor Marsault, LIGM, Université Gustave Eiffel, CNRS, Champs-sur-Marne, France, victor.marsault@univ-eiffel.fr; Wim Martens, University of Bayreuth, Bayreuth, Germany, wim.martens@uni-bayreuth.de; Filip Murlak, University of Warsaw, Warsaw, Poland, f.murlak@uw.edu.pl; Stefan Plantikow, Neo4j, Berlin, Germany, stefan.plantikow@neo4j.com; Ognjen Savković, Free University of Bozen-Bolzano, Bolzano, Italy, ognjen.savkovic@unibz.it; Michael Schmidt, Amazon Web Services, Seattle, USA, schmdtm@amazon.com; Juan Sequeda, data.world, Austin, USA, juan@data.world; Sławek Staworko, RelationalAI, Berkeley, USA and Univ. Lille, CNRS, UMR 9189 CRIStAL, F-59000 Lille, France, slawek.staworko@relational.ai; Dominik Tomaszuk, University of Bialystok, Bialystok, Poland, d.tomaszuk@uwb.edu.pl; Hannes Voigt, Neo4j, Leipzig, Germany, hannes.voigt@neo4j.com; Domagoj Vrgoč, University of Zagreb, Zagreb, Croatia and PUC Chile, Santiago de Chile, Chile, vrdomagoj@uc.cl; Mingxi Wu, TigerGraph, Redwood City, USA, mingxi.wu@tigergraph.com; Dušan Živković, Integral Data Solutions, London, UK, dusan.zivkovic@me.com.

2836-6573/2023/6-ART198
https://doi.org/10.1145/3589778

version of the GQL Standard. It is anticipated that the second version of the GQL Standard will include a rich DDL. Aiming to inspire the development of GQL and enhance the capabilities of graph database systems, we propose PG-Schema, a simple yet powerful formalism for specifying property graph schemas. It features PG-Types with flexible type definitions supporting multi-inheritance, as well as expressive constraints based on the recently proposed PG-Keys formalism. We provide the formal syntax and semantics of PG-Schema, which meet principled design requirements grounded in contemporary property graph management scenarios, and offer a detailed comparison of its features with those of existing schema languages and graph database systems.

CCS Concepts: • **Information systems → Integrity checking**; • **Theory of computation → Data modeling**; **Database constraints theory**.

Additional Key Words and Phrases: property graphs; schemas; graph databases

ACM Reference Format:
Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen Savković, Michael Schmidt, Juan Sequeda, Sławek Staworko, Dominik Tomaszuk, Hannes Voigt, Domagoj Vrgoč, Mingxi Wu, and Dušan Živković. 2023. PG-Schema: Schemas for Property Graphs. *Proc. ACM Manag. Data* 1, 2, Article 198 (June 2023), 25 pages. https://doi.org/10.1145/3589778

1 INTRODUCTION

Property graphs have come of age. The property graph data model is widely used in social and transportation networks, biological networks, finance, cyber security, logistics, and planning domains to represent interconnected multi-labeled data enhanced with properties given by key/value pairs [47]. Its maturity is reflected in the ongoing efforts by ISO (International Organization for Standardization)[1] to create a standard Graph Query Language GQL, which is expected to appear in 2024 [1, 17].

Despite the maturity of commercial and open-source property graph databases, their schema support is limited. Schemas are a fundamental building block for many data systems. They provide structure to data in a formal language, and are used in different scenarios. In the *schema-first* scenario, dominating in production settings of stable systems, the schema is provided during the setup and plays a *prescriptive role*, limiting data modifications. In the *flexible schema* scenario, suitable for rapid application development and data integration, schema information comes together with data and plays a *descriptive role*, telling users and systems what to expect in the data. In the *partial schema* scenario, applicable at advanced development stages, the user wants to enforce a prescriptive schema over stable parts of the data and maintain a descriptive schema depicting the whole data including its evolving parts.

A recent survey of graph processing users [45] revealed that schema compliance is a highly desirable feature that is lacking in property graph database systems. Our inspection of eleven property graph engines reveals a fragmented landscape, where no system offers comprehensive support for schemas, which should allow the user to impose structure on nodes, edges, and properties of the underlying graph instances, as well as enforce constraints. This calls for a unified property graph schema language.

Our goal is to support the endeavours surrounding the GQL standard and accelerate the development of a future standardized property graph schema language by presenting a concrete proposal. Our proposal, called PG-Schema, consolidates and extends discussions arising out of the Property Graph Schema Working Group of the Linked Data Benchmark Council [25]. This model

[1]ISO's Working Group for Database Languages is known as ISO/IEC JTC1 SC32 WG3.

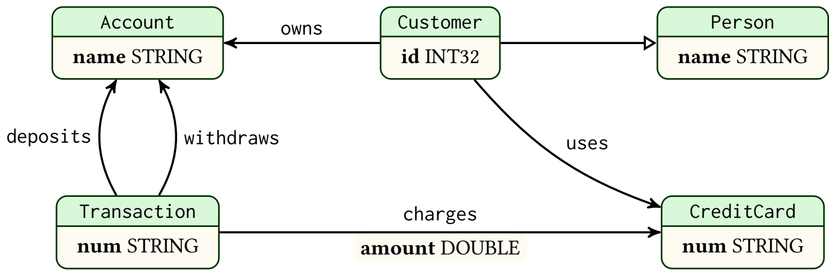

Fig. 1. A diagram of a fraud graph schema.

```
CREATE GRAPH TYPE  fraudGraphType STRICT {
  (personType: Person {name STRING}),
  (customerType: personType & Customer {id INT32}),
  (creditCardType: CreditCard {num STRING}),
  (transactionType: Transaction {num STRING}),
  (accountType: Account {id INT32}),
  (:customerType)
    -[ownsType: owns]->
  (:accountType),
  (:customerType)
    -[usesType: uses]->
  (:creditCardType),
  (:transactionType)
    -[chargesType: charges {amount DOUBLE}]->
  (:creditCardType),
  (:transactionType)
    -[activityType: deposits|withdraws]->
  (:accountType)
}
```

Fig. 2. PG-Schema of a fraud graph schema.

of providing recommendations to standards committees has proven successful, as evidenced by G-CORE [4] and PG-Keys [5] influencing GQL[2].

To illustrate the key features required of schemas for property graphs, we now provide a concrete example of a schema in fraud detection, a common application of graph databases [22], and show how the schema can be used in interactive graph exploration, which in itself is a common functionality provided by graph databases [23, 24, 38, 39, 48]. The diagram in Figure 1 represents a schema describing a fraud graph. This schema is used by Andrea, who works as a financial compliance officer and utilizes an interactive graph explorer to investigate fraud. The following user session highlights how schema information can enhance Andrea's user experience. (The scenario is highly simplified and does not reflect the actual complexity of such tasks and applicable techniques.)

[2]Seven authors of this paper are also members of ISO/IEC JTC1 SC32 WG3.

(1) Andrea opens the graph explorer and connects to the fraud graph, one of the resources in the data catalog. While establishing the connection, the application bootstraps by loading schema information.
(2) Andrea first seeks to identify pairs of suspicious customers. Being aware of the schema, the graph explorer leverages the node type definitions to construct a start page that proposes search for any of the entities available in the domain: `Customer`, `Account`, etc. Andrea proceeds with `Customer`.
(3) Based on the schema information, the graph explorer dynamically constructs a `Customer` search form. It contains separate search fields for the known properties of `Customer` nodes: in our example `id` and `name` (inherited from `Person`). Based on the data type constraints in the schema, the `id` field accepts only integers as input. Andrea uses the `name` field to search for customers of interest and obtains a visual representation of the customer nodes in response.
(4) To inspect potential fraudulent behavior, Andrea needs to understand the connections between customers. Exploiting the schema, the graph explorer leverages the edge type constraints to enumerate and propose specific types of connections. For instance, knowing that `Customers` use `CreditCards` and `Transactions` charge `CreditCards`, it identifies structural connection patterns, written here in Cypher/GQL style,

```
(x:Customer)
 -[:uses]->(:CreditCard)<-[:uses]-
(y:Customer),
```

```
(x:Customer)-[:uses]->(:CreditCard)
  <-[:charges]-(t:Transaction)-[:charges]->
(:CreditCard)<-[:uses]-(y:Customer)
```

and lets Andrea choose the patterns of interest.
(5) Andrea selects the first pattern, which aims to identify shared credit cards that were used by two customers. Based on the schema knowledge, the application constructs and executes an efficient query that quickly identifies all shared credit cards usages between customers.
(6) The graph explorer visualizes the results of the connection search. Upon further investigation, Andrea confirms suspicious cross-customer usage of the same credit card and classifies the cases as fraudulent behavior.

As illustrated by this sample session, the graph explorer would not have been able to effectively guide Andrea through the exploration without concrete schema information. The suggestion of property-specific search restrictions is made possible by *content types*. The schema-assisted query formulation leverages *node* and *edge types*. Going beyond our example, it is easy to see how other type and constraint information may help: for instance, key constraints could indicate preferred search fields; more general participation constraints may improve the schema-assisted query formulation process; orthogonal tooling for schema generation might give graph explorers a standardized path to approximate schema information in case it has not been provided by the graph database authors. These considerations lead to a number of *design requirements* (see Section 2), upon which we base our proposal. The design requirements reflect the consensus of all authors, bringing to bear the theory and contemporary practice of graph schemas.

Our proposed PG-Schema (*Property Graph Schema Language*) comprises PG-Types and PG-Keys. PG-Types specify possible combinations of labels and properties in nodes and edges of different types, as well as constraining the types of edges allowed between nodes of certain types, with the help of a rich inheritance mechanism and abstract types. PG-Keys [5] support diverse integrity constraints, including keys and participation constraints. Via the mechanism of *strict* and *loose* schemas, and partial validation, PG-Schema supports both the descriptive and prescriptive function of schemas. No existing graph database system nor currently envisioned standard covers this arsenal of features; nor do they provide full schema validation support. Our PG-Schema proposal provides both existing systems (reviewed in Section 5) and forthcoming standards with these features, responding to users' demands.

We give a detailed description of PG-Schema in Section 4. As a sample, consider the graph type definition in Figure 2, representing part of the schema diagram of Figure 1. The graph type `fraudGraphType` specifies node types (e.g., `personType`, `customerType`), and edge types (e.g., `activityType`) using the ASCII-art notation of Cypher [21], also adopted by GQL. Properties are declared in curly braces `{...}`. For example, the first statement defines a node type `personType` with label `Person` and a property with key `name` of type `STRING`; and the last statement defines an edge type `activityType`, whose label is either `deposits` or `withdraws`, which connects nodes of types `transactionType` and `accountType`.

Contributions. We make the following contributions:

(1) an analysis of the requirements for property graph schemas;
(2) a proposal for a flexible, agile, usable, and expressive formalism called PG-Schema that fulfills these requirements;
(3) full syntax and semantics of PG-Schema, making the proposal easy to incorporate into both standards and systems;
(4) a parser for PG-Schema [9];
(5) a detailed analysis of schemas in other structured and semi-structured data models and practical graph database systems, as well as their comparison with PG-Schema.

Our contributions impact the following audiences:

(a) graph database standards committee members, who can build upon our recommendations for upcoming features,
(b) graph database vendors, who can use our framework as a guideline to incorporate schemas in their systems, and
(c) researchers, who can use a concrete model of schemas for property graphs as a basis for further investigation.

2 DESIGN REQUIREMENTS

In this section, we elaborate on the design requirements for a suitable notion of a schema for property graphs. These requirements reflect a consensus reached in the course of a systematic multi-phase process informed by the scientific literature, the key use cases from industry, and the forthcoming GQL and SQL/PGQ standards [1, 2] (drafts of both standards are available to LDBC members).

2.1 Property Graphs and Database Schemas

Beyond the ubiquity of applications that focus on graph-structured data and graph analytics, the popularity of graph databases is also generally attributed to the following factors [34, 35, 43, 46].

Agility. Thanks to its proximity to conceptual data models, the property graph data model allows for an efficient translation from domain concepts to database items. This facilitates *high responsiveness* i.e., the ability to quickly and reliably adapt to emerging organizational and domain needs, which is often achieved with iterative and incremental software development processes.

Flexibility. Graph databases are aligned with an iterative and incremental development methodology because they do not require a rigid schema-based data governance mechanism, but rather favor test-driven development, which embraces the additive nature of graphs. The data does not need to be modeled in exhaustive detail in advance but rather *new kinds of objects and relationships can emerge naturally* as new domain needs are addressed by evolving applications.

Database schemas have a number of important functions that can be split into two general categories [12].

Descriptive function. Schemas provide a key to understanding the semantics of the data stored in a database. More precisely, a schema allows to construct a (mental) map between real-world information and structured data used to represent it. This knowledge is essential for any user and application that wishes to access and potentially modify the information stored in a database.

Prescriptive function. A schema is a contract between the database and its users that provides guarantees for reading from the database and limits the possible data manipulations that write to the database. To ensure that the contract is respected, a mandatory schema can be enforced by the database management system.

Our primary objective is to develop a schema formalism for property graph databases that can effectively serve both descriptive and prescriptive roles, while also facilitating and possibly enhancing the strengths of property graph databases. Additionally, we aim at a formalism which enforces correct data modeling practices through its syntax. First, however, we discuss the meaning of two fundamental terms that are essential for defining schemas but are often confused.

2.2 Types and Constraints

The notions of *type* and *constraint* are two main building blocks in virtually any database schema formalism. When used sensibly, they enable the division of schematic information into self-contained fragments that correspond to real-world classes of objects (types) and pieces of knowledge about them (constraints). In general, there is no clear distinction between types and constraints, e.g., data value types are often considered as implicit (domain) constraints. We employ the following nomenclature throughout the paper.

Type is a property that is assigned to elements (data values, nodes, edges) of a property graph database. Types group together similar elements that represent the same kind of real-world object and/or that share common properties, e.g., the set of applicable operations and the types of their results.

Constraint is a closed formula over a vocabulary that permits quantification over elements of the same type. The purpose of constraints is to impose limitations and to express semantic information about real-world objects.

Both types and constraints impose limitations (and provide guarantees) but types are of local nature while constraints are more global. More precisely, checking that an element has a given type should require only inspecting the element itself and possibly the immediately incident elements. On the other hand, validating constraints may require inspecting numerous database elements that need not even be directly connected. Consequently, types are typically verified statically whereas constraints are dynamically verified.

2.3 Requirements

Property Graph Types. The descriptive function of schemas can be particularly beneficial to the agility of property graph databases. Indeed, agility requires a good grasp of the correspondence between database objects and real-world entities, which is precisely the descriptive function of schemas. To communicate this correspondence efficiently, schemas should allow and even encourage a use of types that is consistent with how information about real-world objects is normally/typically divided into nodes, edges, and properties of a property graph database. Nodes are used to represent individual objects with all their attributes stored as node properties.

R1 Node types. Schemas must allow defining types for nodes that specify their labels and properties.

Edges are used to represent relationships between objects of given kinds, and therefore should additionally specify node types of their endpoints.

R2 Edge types. Schemas must allow defining types for edges that specify their labels and properties as well as the types of incident nodes.

Naturally, schemas must also allow a great degree of expressiveness when describing the *content* of nodes and edges, i.e., sets of properties and their values.

R3 Content types. Schemas must support a practical repertoire of data types in content types.

Property Graph Constraints. To further ensure that schemas can properly fulfill the descriptive role and strengthen the agility of property graph databases, we additionally consider the data modeling power that a suitable schema formalism should have. We deliberately target minimal data modeling capabilities and as a reference point we take the most basic variant of Entity-Relationship (ER) diagrams (see Section 5), as the ultimate lower bound in the expressiveness of conceptual modelling languages. To that end, schemas, in addition to the previous requirements, must allow defining keys, which also provide the ability to define weak entities and functional (one-to-many) relationships.

R4 Key constraints. Schemas must allow specifying key constraints on sets of nodes or edges of a given type.

Schemas must allow for participation constraints, which mandate that nodes of a given type participate in a relationship of a given type.

R5 Participation constraints. Schemas must allow specifying participation constraints.

Finally, ER diagrams allow defining hierarchies of node types, a data modeling feature that is even more crucial for property graphs, where a single node may be an instance of multiple node types.

R6 Type hierarchies. Schemas must allow specifying type hierarchies.

Flexibility. As we saw in Sections 1 and 2.1, Property Graphs in practice are often popular in dynamic applications with volatile and evolving graph structures, where new kinds of objects are introduced following the evolving application demands. These typical scenarios require support for flexible schema design across the full range between schema-first and schema-later, with evolvable and extensible schemas.

R7 Evolving data. Schemas must allow defining node, edge, and content types with a finely-grained degree of flexibility in the face of evolving data.

R8 Compositionality. Schemas must provide a fine-grained mechanism for compositions of compatible types of nodes and edges.

Usability. Finally, schemas must be usable in practice. The basic requirements here are that the formalism must be implementable, and have well-defined semantics and a human-friendly declarative syntax. Furthermore, schemas must be easy to derive from graph instances and validation of graph instances with respect to schemas must be efficient. These basic requirements are fundamental for the practical success of any schema solution, as we saw in Section 1.

R9 Schema generation. There should be an intuitive easy-to-derive constraint-free schema for each property graph that can serve as a descriptive schema in case one is not specified.

R10 Syntax and semantics. The schema language must have an intuitive declarative syntax and a well-defined semantics.

R11 Validation. Schemas must allow efficient validation and validation error reporting.

3 DATA MODEL

We assume countable sets $\mathcal{L}$, $\mathcal{K}$, and $\mathcal{V}$ of *labels*, *property names (keys)*, and *property values*. A *record* with keys from $\mathcal{K}$ and values from $\mathcal{V}$ is a finite-domain partial function $o\colon \mathcal{K} \nrightarrow \mathcal{V}$ mapping keys to values. We write $\mathcal{R}$ for the set of all records.

Definition 3.1 (Property Graph). A *property graph* is defined as 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 $N \cap E = \emptyset$;
- $\rho : E \rightarrow (N \times N)$ is a total function mapping edges to ordered pairs of nodes (the endpoints of the edge);
- $\lambda : (N \cup E) \rightarrow 2^{\mathcal{L}}$ is a total function mapping nodes and edges to finite sets of labels (including the empty set);
- $\pi : (N \cup E) \rightarrow \mathcal{R}$ is a function mapping nodes and edges to records.

For an edge $e \in E$ with $\rho_G(e) = (u, v)$, the nodes u and v are the *endpoints* of e, where u is the *source* and v is the *target* of e. For an element $x \in N \cup E$, the record $\pi(x)$ collects all *properties* of x (key-value pairs) and is called the *content* of x.

4 PG-SCHEMA

Most existing data definition languages for relational and semistructured data consist of two parts: *types*, which define the basic topological structure of the data, and *constraints*, which define data integrity. Likewise, PG-Schema consists of two parts. The first part, PG-Types, describes the shape of data and the types of its components such as nodes and edges, reflecting and extending work on SQL/PGQ schemas [2, 26], Graph DDL in the openCypher Morpheus project for Apache Spark [36], and GQL graph types [1, 51].[3] It specifies

- node types, describing the allowed combinations of labels and contents;
- edge types, describing the allowed combinations of labels, contents, and endpoint types; and
- graph types, describing the types of nodes and edges present in the graph.

The second part describes constraints imposed on the typed data. Here, we propose a slight extension of the existing proposal called PG-Keys [5], which specifies integrity constraints such as keys and participation constraints, much like openCypher constraints [44].

This section starts with a guided tour of PG-Types; the full syntax can be found in Figure 3 and a parser is available on Zenodo [9] (with a third-party web alternative [56]). We then define their semantics formally, provide a validation algorithm, and explain how PG-Types interact with PG-Keys.

4.1 PG-Types by Example

We first discuss the basic ingredients of PG-Types (node types, edge types, and graph types) and then move on to more sophisticated aspects such as inheritance and abstract types. We use GQL's predefined data types like `DATE`, `STRING`, and `INT`. These are orthogonal to our proposal and could, in principle, be replaced by any other set of data types. Nevertheless, they take care of requirement **R3**.

Generally, there are two main options for creating types in schemas. One can create *open* types and *closed* types. Both kinds of types are able to specify content that they *require to be present.* The difference between the two is what they allow in addition to the explicitly mentioned content:

[3]Note that in GQL, the term *GQL-schema* refers not to a schema in our sense, but to a dictionary of primary catalog objects such as graphs, graph types, or procedures.

```
pgschema            ::= (createType ";"?)+
createType          ::= createNodeType | createEdgeType | createGraphType
createNodeType      ::= CREATE NODE TYPE (ABSTRACT)? nodeType
createEdgeType      ::= CREATE EDGE TYPE (ABSTRACT)? edgeType
createGraphType     ::= CREATE GRAPH TYPE graphType
graphType           ::= typeName graphTypeMode graphTypeImports? graphTypeElements
graphTypeMode       ::= STRICT | LOOSE
graphTypeImports    ::= IMPORTS typeName ("," typeName)*
graphTypeElements   ::= "{" elementTypes? "}"
elementTypes        ::= elementType ("," elementType )*
elementType         ::= typeName | nodeType | edgeType
nodeType            ::= "(" typeName labelPropertySpec ")"
edgeType            ::= endpointType "-" middleType "->" endpointType
middleType          ::= "[" typeName labelPropertySpec "]"
endpointType        ::= "(" labelPropertySpec ")"
labelPropertySpec   ::= (":" labelSpec)? OPEN? propertySpec?
labelSpec           ::= "(" labelSpec ")"
                      | "[" labelSpec "]"
                      | labelSpec ( ( "|" | "&" ) labelSpec | "?" )
                      | label | typeName
propertySpec        ::= "{" (properties ("," OPEN)? | OPEN)? "}"
properties          ::= property ("," property)*
property            ::= (OPTIONAL)? key propertyType
typeName            ::= StringLiteral
label               ::= l   for  l ∈ 𝓛
key                 ::= k   for  k ∈ 𝒦
propertyType        ::= b   for  b ∈ 𝓑
```

Fig. 3. Core productions of the PG-Schema grammar with labels $\mathcal{L}$, keys $\mathcal{K}$, and base property types $\mathcal{B}$

closed types forbid any content that is not explicitly mentioned, whereas open types allow any such content. Closed types are what we have in SQL, but also in programming languages such as C++ and Java. Open types are the default in JSON Schema. We provide both options here, and use the keyword `OPEN` to indicate the places where we use open types. We use declarative syntax closely aligned with the syntax of types in GQL. It adopts the evocative ASCII-art formatting `( )` for node types and `( )-[ ]->( )` for edge types, originating from Cypher [21].

Base Node Types. The most basic type is a node type. The following example specifies a node type for representing a person:

```
(personType: Person {name STRING, OPTIONAL birthday DATE})
```

It specifies a node of type `personType` with a label `Person`. To distinguish type names from labels, we end type names with the suffix `Type`. By default, types are closed. That is, `Person` is the only allowed label. To permit nodes of type `personType` to have arbitrary additional labels, one should use the keyword `OPEN` and write

```
(personType: Person OPEN {name STRING, OPTIONAL birthday DATE})
```

In terms of properties, the type requires the node to have a property `name` of type `STRING`. Optionally, the node can have a property `birthday`. If it is present, it should have type `DATE`. No additional properties are allowed for this node type. Again, if we would like to allow them, we should write

```
{name STRING, OPTIONAL birthday DATE, OPEN}
```

inside the definition. More precisely, this content description specifies that nodes should have `name` of type `STRING` and arbitrary additional properties. If the property `birthday` is present, its type should be `DATE`. Notice that the `OPEN` modifier applies independently to labels and properties: `OPEN` inside `{...}` applies to properties only and the occurrence outside applies to labels only.

Nodes in property graphs carry *sets* of labels. In PG-Types, we can associate multiple labels to a node type using the `&`-operator:

```
(customerType: Person & Customer {name STRING, OPTIONAL since DATE})
```

The node type `customerType` requires nodes to carry both labels `Person` and `Customer`, and no other labels. In general, we specify the allowed combinations of labels with a variant of *label expressions* built from ℓ (labels) and ℓ? (optional labels) using operators & (and), | (choice). Syntactically, these constitute a subset of label expressions used by GQL and SQL/PGQ for pattern matching in queries. We define their semantics in Section 4.2. Intuitively, `A & B?` would require `A` and additionally allow `B`; and `A | B` gives the choice between the label `A` or `B` (not allowing both). It is easy to define an *inclusive or* `A \/ B` as syntactic sugar for `A | B | (A & B)`. A label expression can be accompanied with `OPEN` which, if specified, allows arbitrary additional labels.

This part of PG-Schema fulfils requirement **R1**. Since we will introduce more advanced node types later using inheritance, we refer to the node types that we explained here as *base node types*.

Base Edge Types. Let us define an edge type called `friendType`. Edges of type `friendType` carry the labels `Knows` and `Likes`, and connect two nodes of type `personType`. They are required to have a property `since` of type `DATE`. The ASCII art `()-[]->()` indicates that we are talking about edges.

```
(:personType)
  -[friendType: Knows & Likes {since DATE}]->
(:personType)
```

If one would like to be more liberal and allow `customerType` nodes on the ends of `friendType` edges, one could use the `|`-operator:

```
(:personType|customerType)
  -[friendType: Knows & Likes {since DATE}]->
(:personType|customerType)
```

One could be even more liberal and use `personType OPEN` to allow arbitrary labels and properties in addition to the material required by `personType`. This part of PG-Schema fulfils requirement **R2**.

Graph Types. A graph type combines node and edge types in one syntactic construct. It includes the types of the schema, as we will see here, but also the constraints, which we will see in Section 4.4. Here is an example:

```
CREATE GRAPH TYPE fraudGraphType STRICT {
  (personType: Person {name STRING, OPTIONAL birthday DATE}),
  (customerType: Person & Customer {name STRING, OPTIONAL since DATE}),
  (suspiciousType: Suspicious OPEN {reason STRING, OPEN}),
  (:personType|customerType)
    -[friendType: Knows & Likes]->
  (:personType|customerType)
}
```

The graph type `fraudGraphType` contains three node types and one edge type. The keyword `STRICT` specifies how a property graph should be typed against the schema. It means that, for a graph G to be valid w.r.t. `fraudGraphType`, it should be possible to assign at least one type within `fraudGraphType` to every node and every edge of G. The alternative, `LOOSE`, allows for *partial validation*, addressing **R7**. Informally, it means that the validation process simply assigns types to as many nodes and edges in the graph as possible, but without the restriction that every node or edge should receive at least one type. We discuss this further in Section 4.2.

We would like to point out the difference between open/closed element types and loose/strict graph types. Why do we use different terminology (and keywords) here? Element types work fundamentally differently from graph types. A node type of the form

```
(nodeType: Label {prop STRING, ...})
```

requires each node of type `nodeType` to have a property `prop`. A graph type such as `fraudGraphType` does not require nodes of type `customerType`. It merely requires that every node gets assigned *some* node type declared in the graph type. Therefore, an open node type can require a given label to be present in a node, but a loose graph type cannot require a given element type to be present in the graph.

The example also shows the keyword `CREATE`. If a node or edge type is created as a catalog object, the declaration should likewise be preceded by `CREATE NODE TYPE` or `CREATE EDGE TYPE`, respectively. Node (and edge) types outside `CREATE GRAPH TYPE` statements should therefore always start with `CREATE`. If `personType` and `customerType` had been already created outside, one could define `fraudGraphType` more succinctly as follows.

```
CREATE GRAPH TYPE fraudGraphType STRICT {
  personType,   // import the type personType
  customerType, // import the type customerType
  (suspiciousType: Suspicious OPEN {reason STRING, OPEN}),
  (:personType|customerType)
    -[friendType: Knows & Likes]->
  (:personType|customerType)
}
```

This leads us to a subtle difference between simply *referring* to a type that has been declared outside of the definition of a graph type, versus *importing* such a type. By default, we are always allowed to refer to any type t that is a catalog object. So, by omitting the import of `personType` and `customerType`, the edge type `friendType` would still be well-defined. However, by *importing* t we also allow objects in the graph type to be assigned the type t, which is important for the notion of *validity* of a graph. When checking if a property graph G is valid against `FraudGraphType`, one needs to be able to assign at least one type t to each element of G such that t is either *declared within* `FraudGraphType` or *imported to* `FraudGraphType`.

Inheritance. Specifying contents for all relevant combinations of labels explicitly can be cumbersome and error-prone. We therefore allow reusing previously defined types in definitions of other types. Such reuse not only makes schemas more compact and modular, but also allows schema designers to follow a natural approach of classifying things as more general or more specialised, as is done in object-oriented modeling. With this mechanism, we fulfil requirement **R6** (type hierarchies).

In the following example, the node type `employeeType` inherits labels and properties from `personType` and `salariedType`.

```
(salariedType: Salaried {salary INT})
(employeeType: personType & salariedType)
```

That is, a node of type `employeeType` has the labels `Person` (inherited from `personType`) and `Salaried` (inherited from `salariedType`). Its properties are `name` and optionally `birthday` (inherited from `personType`), as well as `salary` (inherited from `salariedType`). Note that inheritance automatically conflates properties that are compatible. If `salariedType` had a property `name`, then `employeeType` would only be well-defined if its `name` were a `STRING`.

Similar to nodes, PG-Types allow using edge types when specifying another edge type, which allows inheritance for edge types:

```
(:employeeType)
  -[buddyType: friendType {since DATE, casual BOOL}]->
(:employeeType)
```

The edge type `buddyType` is an edge of type `friendType` but restricts the end nodes to be of type `employeeType`, i.e., end nodes also need to have a `salary` property and `Salaried` label. The type additionally requires the properties `since` of type `DATE` and `casual` of type `BOOL`. Notice that `since` is already required by `friendType`, so type `buddyType` would not change if we omitted it from the definition of `buddyType`. Intuitively, we can think that inherited types collect all the property specifications of the parent types, and add the newly specified ones. If we declared `since` to be of a different type than `DATE`, the resulting edge type would be impossible to instantiate, and as such it would be redundant. The precise rules for how edge types and node types of endpoints are combined are in Section 4.2.

We also support *graph inheritance*, which amounts to importing all node and edge types from one graph type to another graph type. For example, by writing

```
CREATE GRAPH TYPE fraudGraphType STRICT IMPORTS socialGraphType {...}
```

we import to `fraudGraphType` all types in `socialGraphType`.

Including Types in Label Expressions. We can combine inheritance with adding new properties or labels. For instance, if we wrote

```
(employeeType: personType & salariedType {birthday DATE})
```

then `birthday` would be a mandatory property in nodes of type `employeeType`, in addition to inherited properties `salary` and `name`.

Formally, we combine properties using the ⊕-operator, inspired by mixins [13] and explained in Section 4.2. Abstractly, if we define types

```
(xType: A & B {propertyA INT, propertyB INT})
(yType: B & C {propertyB INT, propertyC INT})
(zType: xType & yType)
```

then nodes of type `zType` have all labels `A`, `B`, and `C`, and all properties `propertyA`, `propertyB`, and `propertyC`. Since both `xType` and `yType` are closed, this means that a node can be of type `zType`, but not of type `xType` and not of type `yType`.

If both `xType` and `yType` were open types, i.e., declared their label sets and contents to be `OPEN`, then nodes of type `zType` would automatically fulfill `xType` and `yType`. That is, for open types, we support *intersection types* via the operator `&`.

More generally, in the definitions of types we allow arbitrary expressions built from labels and previously defined types using operators `?`, `&`, and `|`. Of course, declarations of node types should only refer to node types and similarly for edge types. Also, references should not be cyclic, as is standard in inheritance hierarchies. Notice that using `|` we can define a *union type*, which allows going beyond base types. For instance,

```
(aType: A {propertyA INT})
(bType: B {propertyB INT})
(cType: aType | bType)
```

creates a node type `cType` which either has the label `A` or `B`. However, `A` is only allowed to occur together with `propertyA` and `B` only with `propertyB` (but not `A` with `propertyB`). One cannot define `cType` as a base type, since base types always admit all combinations of matching label sets and matching property sets. Furthermore, if one of the base types were open, the derived type would automatically be open. This holds for both label openness and property openness. However, if both base types are closed, the derived type can be declared open (independently for labels and properties).

The inclusion of types in label expressions makes the formalism highly compositional, fulfilling requirement **R8**.

Abstract Types. In some cases, one may want to declare a type as *abstract*, which means that it cannot be directly instantiated.

```
ABSTRACT (salariedType {salary INT})
(employeeType: personType & salariedType)
```

Notice that salariedType specifies nodes with no labels and a single property salary. On its own, this type may not be very useful, since we expect nodes with property salary to have labels and possibly other properties as well. Through inheritance from salariedType, the type employeeType matches nodes with all the labels and properties allowed in personType, plus an additional property salary.

4.2 Formal Definition and Semantics

So far we have been talking about graph types by means of syntax. We now present a syntax-independent definition. Later we shall see how the two connect, thus providing the semantics for our declarative syntax, and fulfilling design requirement **R10**.

Types and conformance. Recall from Section 3 that $\mathcal{L}$ is the set of labels, and $\mathcal{R}$ the set of all possible records. We define a *formal base type* as a pair (L, R), where $L \subseteq \mathcal{L}$ and $R \subseteq \mathcal{R}$. We write $\mathcal{T}$ for the set of all formal base types. An element (a node or an edge) with label set K and content o *conforms* to a formal base type (L, R), if $K = L$ and $o \in R$. For the formal definition, we allow arbitrary subsets of $\mathcal{R}$ to form base types. In the concrete syntax of the previous section, these will be given by record types; e.g., for the type {a INT, b STRING}, the set R consists of all partial functions that map a to an integer and b to a string.

Definition 4.1. A *formal graph type* is 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;
- $\nu_S : N_S \longrightarrow 2^{\mathcal{T}}$ maps node type names to sets of formal base types;
- $\eta_S : E_S \longrightarrow 2^{\mathcal{T}\times\mathcal{T}\times\mathcal{T}}$ 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.

For brevity, we shall often refer to the elements of N_S and E_S as node and edge types, rather than node and edge type names. For dealing with *strict* and *loose* typing, we will use slightly different but connected notions, namely *conformance* and *typings*.

Definition 4.2. 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.

The *typing of* G wrt. S is the mapping $\text{Types} : N_G \cup E_G \longrightarrow 2^{N_S} \cup 2^{N_S}$ defined as follows for all $u \in N_G$ and $e \in E_G$:

$$\text{Types}(u) = \{\tau \in N_S \mid u \text{ conforms to } \tau\}, \qquad \text{Types}(e) = \{\tau \in E_S \mid e \text{ conforms to } \tau\}.$$

Hence, G conforms to S if Types maps all nodes and edges to non-empty sets of types.

Schema compilation. We now explain how to interpret the syntax described in Section 4.1 in terms of formal graph types introduced above, thus providing the semantics for the syntax. This process, which we call *schema compilation*, will effectively amount to unravelling and normalising all type definitions.

Let T be a syntactically represented graph type. We shall define a corresponding formal graph type $S = (N_S, E_S, \nu_S, \eta_S)$. For N_S and E_S we take the sets of node and edge type names used in T. Because type definitions in T are acyclic, we can use a bottom-up approach to unravel them.

Consider a node type definition $(\tau{:}F)$ in T. Recall that F is an 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. 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, much like a regular expression describes how to generate words.

Let $t_\emptyset = (\emptyset, \{\bot\})$ and $t_\ell = (\{\ell\}, \{\bot\})$, where $\bot$ stands for the empty record. These are the empty formal base type and the formal base type of a single label, with no content. We add content using content descriptions, which are record types written as

$$r = \{\, [\texttt{OPTIONAL}]\; k_1\; \mathbb{b}_1,\; \ldots,\; [\texttt{OPTIONAL}]\; k_n\; \mathbb{b}_n,\; [\texttt{OPEN}] \,\}$$

where the square brackets mean that the keywords OPTIONAL and OPEN are optional, the k_is are keys from $\mathcal{K}$, and the $\mathbb{b}_i$s are base property types such as INT or DATE. Let $\mathbb{B}_i$ be the extent of $\mathbb{b}_i$ (e.g., $\mathbb{Z}$ for INT). When the keyword OPEN is present, the semantics $(\!|r|\!)$ of r is the set of all records $o \in \mathcal{R}$ such that for all $i \le n$, if $k_i \in \mathrm{dom}(o)$ then $o(k_i) \in \mathbb{B}_i$, and k_i must belong to $\mathrm{dom}(o)$ unless it is preceded by the keyword OPTIONAL. When the keyword OPEN is absent, we additionally require that $\mathrm{dom}(o) \subseteq \{k_1, \ldots, k_n\}$.

With these tools, we could define the semantics of node types with a single label and a content description. In order to handle more complex types we need a way to combine formal base types. We begin from records. We call records $o_1, o_2 \in \mathcal{R}$ *compatible* if $o_1(k) = o_2(k)$ for each $k \in \mathrm{dom}(o_1) \cap \mathrm{dom}(o_2)$. For compatible o_1 and o_2 we define their *combination* $o_1 \oplus o_2$ as $(o_1 \oplus o_2)(k) = o_1(k)$ for $k \in \mathrm{dom}(o_1)$ and $(o_1 \oplus o_2)(k) = o_2(k)$ for $k \in \mathrm{dom}(o_2) \setminus \mathrm{dom}(o_1)$. 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 $o_1 \oplus o_2$ for compatible $o_1 \in O_1$ and $o_2 \in O_2$. This operation is akin to the natural join known from relational algebra. The only difference is that in relational algebra columns are fixed for each relation, whereas a set of records may contain records with different sets of keys. We lift the $\oplus$ operator to formal base types by letting $(L_1, R_1) \oplus (L_2, R_2) = (L_1 \cup L_2, R_1 \oplus R_2)$. Note that in the absence of content, this amounts to taking the union of two sets of labels.

Now we can define the semantics recursively for all subexpressions of F as follows:

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

With that, the semantics of τ is defined as

$$\nu_S(\tau) = (\!|F|\!).$$

For edge types, we proceed in the same vein as for node types. Consider an edge type in T defined as $(:F_{\mathrm{src}})$-$[\tau{:}F]$->$(:F_{\mathrm{tgt}})$. Expressions F_{src} and F_{tgt} specifying the source and target endpoints are interpreted as explained above. Expression F defines the set $[\![F]\!] \subseteq \mathcal{T} \times \mathcal{T} \times \mathcal{T}$ of triples of formal base types, according to the following rules:

$$[\![\ell]\!] = \{(t_\emptyset, t_\ell, t_\emptyset)\}, \qquad [\![\sigma]\!] = \eta_S(\sigma), \qquad [\![F_1 \mid F_2]\!] = [\![F_1]\!] \cup [\![F_2]\!],$$
$$[\![F_1?]\!] = [\![F_1]\!] \cup \{(t_\emptyset, t_\emptyset, t_\emptyset)\}, \qquad [\![F_1 \,\&\, F_2]\!] = [\![F_1]\!] \oplus [\![F_2]\!],$$
$$[\![F_1 \texttt{ OPEN}]\!] = \{(t_1, (L,R), t_2) \mid \exists L' \subseteq L \text{ s.t. } (t_1, (L',R), t_2) \in [\![F_1]\!]\},$$
$$[\![F_1\, r]\!] = \{(t_1, (L, R \oplus (\!|r|\!)), t_2) \mid (t_1, (L,R), t_2) \in [\![F_1]\!]\},$$

where the $\oplus$ operator is extended to sets $Y_1, Y_2 \subseteq \mathcal{T} \times \mathcal{T} \times \mathcal{T}$ of triples of formal base types by

$$Y_1 \oplus Y_2 = \{(s_1 \oplus t_1, s \oplus t, s_2 \oplus t_2) \mid (s_1, s, s_2) \in Y_1, (t_1, t, t_2) \in Y_2\}.$$

With that, we define the semantics of the edge type τ, specified by `(:`F_{src}`)-[`τ`:`F`]->(:`F_{tgt}`)`, as

$$\eta_S(\tau) = ((\!|F_{\text{src}}|\!) \times \{t_\emptyset\} \times (\!|F_{\text{tgt}}|\!)) \oplus [\![F]\!].$$

After all type definitions in T have been unravelled, as the last step, we restrict the sets N_S and E_S to type names whose definitions in T are not preceded by the keyword `ABSTRACT`.

Having defined how a syntactically represented graph type T corresponds to a formal graph type $S_T = (N_S, E_S, \nu_S, \eta_S)$, we can now explain validation of a property graph G against `LOOSE` and `STRICT` graph types. In both cases, one begins by computing the typing of G w.r.t. S_T. If T is `LOOSE`, this is where the *validation of G against T* ends; the mapping Types fully specifies which element of G can be assigned which types from T. If T is `STRICT`, we do one more step, namely we test if Types assigns at least one type in S_T to each element in G. If it does, we say that G *conforms to* T.

4.3 Validation and Graph Type Generation

We now discuss how to validate a graph against a graph type. For a formal graph type, the process is straightforward: for each node type we identify the nodes that conform to it, and we use this information to identify for every edge type the set of edges that conform to it. The validation for general graph types, defined with the syntax in Section 4.1, can be accomplished efficiently with an analogous procedure thanks to the mathematical simplicity of the schema compilation rules in Section 4.2. More importantly, such a validation procedure can be implemented in a reasonably expressive graph query language. In essence, such a language would need to support standard set operations and would need to allow identifying nodes and edges based on their labels, property names, and property value types. Consequently, the proposed graph schema formalism satisfies requirement **R11**.

We also propose a simple method of generating a graph type for a given property graph. For every node, we introduce an anonymous node type that fits precisely its set of labels and properties, we gather those types into a set, and finally generate their names. Using anonymous types allows to eliminate repetitions of syntactically identical types easily. We next apply an analogous procedure for edges remembering to use previously identified node types of the endpoints. Finally, we group types of all elements into the resulting graph type. This shows the satisfaction of requirement **R9**. One might wish for a more refined method of schema generation, but those fall into the domain of schema inference [11, 27], which is out of the scope of this paper and left as future work.

4.4 Adding Constraints

Our focus until now was on *types* in PG-Schema. The other crucial aspect of PG-Schema is its *constraints.* To this end, we leverage existing work on *keys for property graphs* [5], called PG-Keys. Despite their name, PG-Keys go beyond the capability of expressing key constraints. Statements in PG-Keys are of the form

`FOR` $p(x)$ `<qualifier>` $q(x, \bar{y})$, where <qualifier> specifies the kind of constraint that is being expressed and consists of combinations of EXCLUSIVE, MANDATORY, and SINGLETON. Both $p(x)$ and $q(x,\bar{y})$ are queries. For instance, if we want to express that, for every output x of $p(x)$ there should be at least one tuple $\bar{y} = (y_1, y_2, \ldots, y_n)$ that satisfies $q(x,\bar{y})$, we write FOR $p(x)$ MANDATORY $q(x, y_1, \ldots, y_n)$. SINGLETON would mean that there should be at most one such $\bar{y}$ for each x, and EXCLUSIVE that no $\bar{y}$ should be shared by two different values of $\bar{x}$. Inside the queries, we can use the keyword WITHIN to make clear what the output of the queries is, i.e., what we want to be EXCLUSIVE, etc.

In PG-Schema, we slightly extend the syntax of PG-Keys by allowing the constraints to refer to a *type name* at each point where PG-Keys allows a label. The semantics of the resulting expression is that *a type name t matches every node that conforms to t*.

Consider the following code snippet, describing a graph with two kinds of nodes, persons (personType) and customers (customerType), and friend-edges between persons (the edge type friendType, requiring labels Knows and Likes and allowing label Bestie on the edge).

```
CREATE GRAPH TYPE socialGraphType STRICT {
  (personType: Person {name STRING, id INT}),
  (customerType: Customer {id INT}),
  (:personType)
    -[friendType: Knows & Likes & Bestie?]->
  (:personType),
  // Constraints
  FOR (x:personType)
    EXCLUSIVE MANDATORY SINGLETON x.id,
  FOR (x:customerType)
    MANDATORY y.id WITHIN (y:personType) WHERE y.id = x.id,
  FOR (x:personType)
    SINGLETON y WITHIN (x)-[y: friendType & Bestie]->()
}
```

Apart from type declarations, the graph type also has three PG-Key constraints. The first expresses that the value of the property id should be a key for nodes of type personType. The second PG-Key expresses that every id value of a customer should be an id of a person, which is a foreign key. PG-Keys (and therefore PG-Schema) can therefore handle *key and foreign key constraints* (requirement **R4**). The third PG-Key expresses that each person is allowed to have at most one best friend. Notice that y: friendType & Bestie means that y should have label Bestie and it should conform to the type friendType. If we wrote MANDATORY y WITHIN (x)-[y:friendType]->() in the second constraint, it would express that each person participates in the friendType relation, i.e., it expresses a participation constraint (requirement **R5**). PG-Keys are also powerful enough to express SQL-style CHECK constraints, such as

```
FOR (x:salariedType)
  MANDATORY x.salary >= 0,
```

or denial constraints, such as

```
FOR (x:customerType)
  MANDATORY (x:!employeeType),
```

where ! denotes negation.

The semantics of PG-Keys is the same in loose and strict graph types. In particular, constraints in loose graph types are not trivially satisfied and can be useful. For instance, by changing STRICT to LOOSE in the graph type above, we allow nodes that are neither of type personType, nor of type customerType, but id must still be a key for all personType nodes.

Concerning validation, notice that the typing of a property graph G w.r.t. a graph type S can be computed efficiently. Once we have this typing, PG-Keys can be evaluated as in the original paper [5], using types as if they were labels in the graph.

We conclude this section with the observation that PG-Schema satisfies all requirements identified in Section 2.

5 RELATIONSHIP TO OTHER PARADIGMS

We now compare PG-Schema with existing schema formalisms and with existing graph schema technologies. We consider a wide range of formalisms. First, we consider *conceptual data models* such as the Entity-Relationship Model and its variants. Second, we discuss graph schemas for RDF graphs stemming from the *Semantic Web* setting. Third, we overview schema languages for *tree-structured data* formalisms such as XML and JSON. A detailed description of these existing schema formalisms is omitted for space reasons and can be found in an external technical report [10].

In order to perform this comparison, we define several features in Section 5.1, and discuss their support in Section 5.2. Finally, we propose potential PG-Schema extensions in Section 5.3.

5.1 Existing Graph Schema Features

We briefly describe the main features used to compare state-of-the-art graph schema languages in Table 1. We group these into: type features, constraint features, and schema features.

Type features. We considered: **(PDT)** the number of built-in primitive data types, **(UIT)** type constructors for union and intersection types, **(TH)** type hierarchies, **(AT)** abstract types, **(OCT)** open and closed types, **(EP)** edge properties, **(MOP)** mandatory and optional properties, **(CPT)** complex nested property types consisting of nested collection types, and **(RC)** range constraints.

Constraint features. We examined: **(KC)** key constraints, **(MP)** mandatory participation of certain types of nodes in certain types of edges, **(CC)** cardinality constraints for such participation, and **(BRC)** properties of binary relations defined by certain edges, such as (ir)reflexivity, (in)transitivity, (a)cyclicity, (a/anti)symmetry, etc.

Schema features. We assessed: **(TV)** if validation is tractable, **(ISP)** if introspection is possible, i.e., the schema can be queried like a graph instance, and finallly **(SFPX)** if the schema can be specified to be (1) *first*, and subsequently enforcing it for all instances, (2) *partial*, allowing some of its components to be descriptive (e.g., the element types) and some to be prescriptive (e.g., the constraints), or (3) *flexible*, creating and updating instances in an unconstrained manner while possibly maintaining a descriptive schema.

5.2 Support of the Features

We comment on some of the differences between PG-Schema and the existing state-of-the-art graph schema formalisms and systems from Table 1, with a focus on existing graph technologies.

Conceptual data models. ER-based data models tend to be agnostic with respect to attribute types, since these may depend on the back-end for which the data model is designed. Most support inheritance hierarchies and, in that way, can model union and intersection types. Entity types can be modelled as abstract types by indicating that their entities must belong to at least one of their subtypes. Since the final goal is to design a relational schema, which is closed, none of them support open types. Most ER-based models allow attributes to be composed and/or multi-valued, and so can model complex nested values. A surprising restriction is that most conceptual data models only allow a single key and require it to be a single attribute. A notable exception is ORM2, which can

Table 1. Overview of the features supported by state-of-the-art graph schema formalisms

PDT = *Primitive Data Types*, **UIT** = *Union and Intersection Types*, **TH** = *Type Hierarchy*, **AT** = *Abstract Types*, **OCT** = *Open/Closed Types*, **EP** = *Edge Properties*, **MOP** = *Mandatory/Optional Properties*, **CPT** = *Complex Property Types*, **RC** = *Range Constraints*, **KC** = *Key Constraints*, **MP** = *Mandatory Participation*, **CC** = *Cardinality Constraints*, **BRC** = *Binary-Relation Constraints*, **TV** = *Tractable Validation*, **IS** = *Introspection*, **SFPX** = *Schema First/Partial/fleXible*

	PDT	**UIT**	**TH**	**AT**	**OCT**	**EP**	**MOP**	**CPT**	**RC**	**KC**	**MP**	**CC**	**BRC**	**TV**	**IS**	**SFPX**
Chen ER [15]	[-]	-	-	-	c	n/e	m	-	[-]	[✓]	✓	-	-	[-]	-	f
Extended ER [53]	[-]	n/e	n/e/p	n/e	c	n/e	m/o	✓	[-]	✓	✓	✓	-	[-]	-	f
Enhanced ER [18]	[-]	n/e	n/e	n	c	n/e	m/o	✓	[-]	[✓]	✓	✓	-	[-]	-	f
ORM2 [28]	[-]	n/e/p	n/e/p	n/e	c	n/e	m/o	[-]	✓	✓	✓	✓	✓	[-]	-	f
UML Class Diagrams [20]	[5]	n	n	n	c	n/e	m/o	[-]	✓	✓	✓	✓	[✓]	[-]	-	f
RDFS [14]	34	-	n/e/p	-	o	[-]	[o]	-	-	-	-	-	-	✓	✓	f/[p]/x
OWL [30]	[33]	n/e/p	n/e/p	-	o	n	[m]/[o]	-	✓	✓	[-]	✓	✓	[✓]	✓	f/[p]/x
SHACL [33]	34	n/e/p	n/e/p	-	o/c	n	m/o	-	✓	[✓]	✓	✓	-	[✓]	✓	f/p/x
ShEx [7, 50]	34	n/e/p	n/e/p	-	o/c	n	m/o	-	✓	[✓]	✓	✓	-	[✓]	✓	f/p/x
DTD [59]	6	[n]	-	-	o/c	n	m/o	[-]	-	[-]	✓	-	-	✓	[✓]	f/x
JSON Schema [57]	6	n/e	n/e/p	n	o/c	n	m/o	✓	✓	[✓]	✓	✓	-	✓	✓	f/x
RELAX NG [31]	[2]	n	n/e/p	n	o/c	n	m/o	✓	✓	[-]	✓	-	-	✓	✓	f/x
XML Schema [49]	[47]	n	n/e/p	n	o/c	n	m/o	✓	✓	✓	✓	✓	-	✓	✓	f/x
GraphQL SDL [19, 29]	5	n/[e]	n/e	n/e	c	n/e	m/o	✓	[✓]	✓	✓	-	-	[-]	✓	f/[p]
openCypher Schema [12]	[oC]	[n]	n/e	n/e	c	n/e	m	✓	-	✓	✓	-	-	✓	✓	p/x
SQL/PGQ [2]	[SQL]	-	[n]/[e]	n/e	c	n/e	m/o	✓	[✓]	✓	✓	[-]	-	✓	-	f
GQL [1]	[-]	-	-	-	[o]/c	n/e	m/o	-	-	-	-	-	-	✓	-	f/x
AgensGraph [3]	[-]	-	n/e/p	-	o	[n]/[e]	m/o	[✓]	[✓]	[✓]	✓	-	-	[✓]	[✓]	f/[p]/x
ArangoDB [6]	6	[n]/[e]	n/e/p	n	o/c	n/e	m/o	✓	[✓]	[✓]	✓	[✓]	-	[✓]	✓	f/x
DataStax [16]	[25]	-	-	-	[o]	n/e	m/o	✓	[✓]	✓	✓	-	-	?	✓	f/[p]/x
JanusGraph [32]	12	-	-	-	[o]	n/e	m/o	✓	-	✓	✓	✓	-	✓	[✓]	f/[p]/x
Nebula Graph/nGQL [58]	5	-	-	-	c	n/e	m/o	-	-	-	-	-	-	[-]	✓	f
Neo4j [37]	11	[n]	[-]	[-]	o	n/e	m/o	[✓]	-	✓	-	[-]	-	-	✓	p/x
Oracle/PGQL [40]	11	[n]/[e]	-	-	c	n/e	[m]/[o]	-	[✓]	[✓]	[✓]	[-]	-	[✓]	✓	f/[p]/x
OrientDB/SQL [41]	23	[n]/[e]	[n]/[e]	[n]/[e]	o/c	n/e	[m]/[o]	✓	[✓]	[✓]	[✓]	[-]	-	[✓]	[✓]	f/[p]
Sparksee [52]	8	-	-	-	c	n/e	m/o	[✓]	-	✓	✓	-	-	[-]	[✓]	f
TigerGraph/GSQL [54]	8	[n]/[e]	-	[-]	c	n/e	[m]/[o]	✓	[✓]	✓	-	[-]	[✓]	[✓]	✓	f
TypeDB/TypeQL [55]	5	-	n/e/p	n/e/p	c	n/e	m	✓	[✓]	✓	✓	[-]	✓	[-]	✓	f
PG-Schema	[-]	n/e	n/e/[-]	n/e/[-]	o/c	n/e	m/o	[-]	[-]	✓	✓	[-]	[-]	✓	-	f/p/x

Legend: '✓' = *supported*, '-' = *not supported*, '?' = *unknown*, $[x]$ = *qualified* x, n/e/p = *supported for (n)odes, (e)dges, and (p)roperties*, o/c = *(o)pen and (c)losed*, m/o = *(m)andatory and (o)ptional*, f/p/x = *schema (f)irst, (p)artial, and fle(x)ible*, oC = *openCypher*,

support any number of composed keys. Schema validation is not applicable in this context, as there is no notion of a schema-independent instance.

RDF formalisms. RDF-based formalisms inherit XML datatypes with some limitations (PDT). Both SHACL and ShEx are based on a kind of *open* semantics in which the closeness of a constraint needs to be specified with a keyword *close* (OCT). The element properties are expressible only over the nodes, except the recent proposal of RDF-star, that extends RDF exactly with properties over edges (EP). SHACL and ShEx are missing explicit support for key-like constraints (KC), but allow for cardinality constraints (CC), to which SHACL applies set and ShEx bag semantics. The complexity of validation (TV) of RDF-based formalisms is a well-researched topic. While it is not tractable in general for the most expressive cases, practically useful fragments do have this property.

Tree-structured data. DTDs support union types (UIT) in content type expressions by allowing disjunction, despite it not being applicable to the names of XML element types. This is, however, possible in both RELAX NG and XML Schema, where we have full union types, although both require the regular expressions that describe the content to be deterministic. DTDs do not support inheritance (TH) or abstract types (AT), although element types can be embedded in the content of other element types. However, RELAX NG and XML Schema have explicit support for these features. In DTDs, the content of an element can be made open by using `PCDATA`, which allows any XML fragment as content, so (OCT) is more or less supported. In RELAX NG and XML Schema, it is possible to have an even more sophisticated mix of open and closed parts within a content type. Attributes of XML elements can be declared as required and are optional by default. For some basic attribute types there are multi-valued variants that can be assigned, but, otherwise, there are no complex nested attribute types. Here again, RELAX NG and XML Schema offer a set of type constructors that enable users to build new and more complex attribute types. Concerning (KC), both in DTDs and RELAX NG, an attribute can be defined as key, by giving it the type `ID`, but there is no way to declare an arbitrary set of attributes as key. XML Schema, on the other hand, has a very elaborate notion of key constraint that can define a complex key for elements where this key consists of several values, is not restricted to attributes of the element, and is applicable only to a specified set of elements. DTDs support mandatory participation (MP), since they can require that the content of a certain element be nonempty or that a certain attribute reference some document element. This is the case in XML Schema, which also supports foreign keys, and RELAX NG. DTDs offer some support for (IS), in the sense that they are regarded as included in the XML document, and so are accessible by software processing the document. For both XML Schema and RELAX NG there is an XML syntax, and so they allow full introspection. For DTDs, XML Schema, and RELAX NG, it holds that XML documents can be created without a schema, but it is also possible to restrict updates to only allow those which maintain the document's conformance to the schema; thus, for (SFPX), both schema-first and schema-flexible are supported.

JSON Schema supports unions and intersections of open types (UIT), through the `anyOf` and, respectively, `allOf` keywords. The latter also supports building type hierarchies (TH), but only for open types. Whether types are instantiated depends on the choice of the root object, and so there is arguably support for abstract types (AT). One can control whether types are open or closed by setting `additionalProperties` to `True` or `False`. Fields are optional by default, but can be marked as mandatory through the `required` construct. Range constraints (RC) are limited to numerical values and strings. Field values support (CPT), as they can encode nested objects and arrays. Key constraints (KC) are not supported and cardinality constraints (CC) specify bounds for arrays. Finally, as the schema is itself a JSON object, it allows introspection (IS).

Existing graph technologies. We now discuss the extent to which type, constraints, and schema features are supported in several state-of-the-art graph schema languages and systems.

Type features, such as union and intersection types (UIT), type hierarchies (TH), and abstract types (AT), are supported by GraphQL, and JSON Schema. These capabilities are also found in SQL-based systems, such as OrientDB/SQL, and in systems able to leverage JSON Schema, such as ArangoDB and AgensGraph. The strongly-typed TypeDB database has a rich type system that also offers subtyping for entities, relations, and attributes/properties. While other considered systems cannot directly handle TH, some can emulate it through multi-labels by adding all the intended parent types of a node as its additional labels. Nevertheless, this is problematic for validation, as one cannot ensure that all subtypes have been assigned the correct label. Note that most examined technologies only implement closed types (OCT), except ArangoDB and OrientDB, in which also open types are possible, and Neo4j, which considers types open and, hence, extensible, by default.

Nodes and edges can be enriched with element properties (EP) in all surveyed graph technologies, and in AgensGraph these can be defined using JSON objects. Such properties are optional by default in AgensGraph, ArangoDB, DataStax, JanusGraph, Neo4j, and Sparksee, though users can define mandatory constraints to enforce them being non-nullable. In Nebula Graph, users can specify, when designing their schema, whether null-valued attributes are allowed, while in TypeDB these are not supported. Finally, the (MOP) feature is present in SQL-based technologies. Most reviewed graph languages and system also allow for (CPT), although specific restrictions sometimes apply. For example, in Neo4j, complex property values can only be homogeneous lists of simple types and byte arrays, despite the latter not being first-class Cypher data types. AgensGraph draws its support for (CPT) from openCypher and JSON, while in Sparksee, multi-valued properties can only be defined using array attributes, using all but the `String` and `Text` data types. In addition, range constraints (RC) can be specified for any data type, in SQL-based technologies, and for numerical values and strings, in systems that build on JSON Schema or that provide regular expressions, such as TypeDB.

Regarding *constraint features*, we remark that key constraints are available in all reviewed graph schema languages except GQL. At a system level, AgensGraph, Neo4j, and Sparksee support node uniqueness constraints, disallowing the same property values from appearing in more than one node of a given label or type, while ArangoDB enables specifying `uniqueItems` for arrays, thanks to JSON Schema. Some technologies, such as DataStax, Oracle/PGQL, and TigerGraph, offer primary keys for nodes, which enforce property values to be unique, mandatory, and single-valued. TigerGraph GSQL also supports the notion of a discriminator, which is an attribute or set of attributes that can be used to uniquely identify an edge, when multiple instances of a given type exist between a pair of vertices. Finally, the considered SQL-based systems can rely on SQL's mechanism for defining unique key constraints for tables. These systems also feature mandatory participation (MP) and uniqueness constraints.

More general forms of cardinality constraints (CC) are only provided by a few systems. Among these, JanusGraph allows declaring edge label multiplicity: the `MULTI` and `SIMPLE` keywords can specify whether multiple edges or at most one can be defined between any node pair; `MANY2ONE` and `ONE2MANY` respectively allow at most one outgoing/incoming edge, without constraining the number of incoming/outgoing ones. The system also provides property key cardinalities, i.e., declaring whether one (`SINGLE`), an arbitrary number (`LIST`), or multiple, non-duplicate values (`SET`) can be associated with a node key. Other examples include ArangoDB, leveraging JSON Schema's `minProperties`/`maxProperties` keywords to restrict the number of object properties, and TypeDB, providing high-level CCs at the type level that require relationships to have at least one role that specifies their nature. TypeDB is also the only system that handles binary-relation constraints (BRC), such as symmetry and transitivity, by expressing them via inference rules. In TigerGraph, the support for (BRC) is limited to declaring reverse edge types.

Tractable validation (TV) is a *schema feature* supported by systems that leverage JSON Schema and SQL. In JanusGraph and Sparksee, the schema is defined through their specific APIs and there is no formal account of their schema validation mechanisms. Concerning introspection (IS), all reviewed systems support it either directly, through the query language itself, or indirectly, via a management API (like in JanusGraph and Sparksee). Finally, the only reviewed system that natively supports (SPFX) is OrientDB/SQL, which has schema-first, schema-less, and explicit schema-hybrid modes. Partial conformance is possible in all systems that are not exclusively schema-first or that do not have native schema mechanisms, like ArangoDB, which relies on JSON Schema.

PG-Schema. Like SQL/PGQ and GQL, PG-Schema views a set of node and edge types as the core of a graph database schema. The support for type features is essentially complete, as discussed in Section 4, except that (CPT) and (RC), as well as (UIT) and (TH) for properties, are delegated to the property type system. While concrete property types have been used in examples, PG-Schema deliberately leaves the choice of the property type system open, which allows it to function as an embedded language in both GQL and SQL/PGQ, offering suitable property type features. In Section 5.3 we discuss how some of these features could be supported directly in PG-Schema.

In terms of constraint features, PG-Schema is also quite comprehensive. As discussed in Section 4.4, it supports not only (KC) and (MP), but also denial constraints and uniqueness constraints. In Section 5.3 we discuss an extension to support general cardinality constraints (CC) that fits well with the support for (KC) and (MP).

Important design principles for PG-Schema were to preserve the spirit of schemas in SQL/PGQ and graph types in GQL, keeping node and edge types locally verifiable, while at the same time offering a powerful mechanism to express constraints. This is why PG-Keys are clearly separated from PG-Types, and constraints can refer to types, but types cannot refer to constraints. This is in contrast with other approaches, such as SHACL, where there is no such separation and a small change in one element may affect the types of many distant elements.

Because types are locally verifiable, PG-Schema has tractable validation (TV), as long as each constraint alone is tractable, which is the case for key, participation, and cardinality constraints. PG-Schema fully supports (SFPX), as it allows defining strict schemas (schema first), loose schemas that only enforce constraints on typed elements (partial schema), and schemas that allow every graph (flexible schema); note that it is possible to generate a descriptive schema, as required (see Section 4.3). Finally, owing to locally verifiable types and the design of PG-Keys, if a graph only partially conforms to a given schema (strict or loose), this can be easily explained to users by indicating which elements are typed and which satisfy the constraints, thus supporting meaningful partial validation.

Basic graph types can be naturally represented as property graphs, as shown in Figure 1. However, there is currently no commonly agreed-upon way of reflecting the powerful mechanism of type combinations in PG-Types; hence, introspection (IS) is not supported. How such a representation can capture all features and yet remain intuitive, is an issue for future research.

Conclusion. The table shows that there is quite some variety in which features are supported and also that no formalism or system covers all of them. This likely reflects that they tend to target different sets of use cases. Likewise, PG-Schema does not attempt to cover all features, but it does aim to provide a foundation that could be extended to do so.

5.3 Possible Extensions of PG-Schema

Let us now discuss briefly how some of the currently unsupported features from Table 1 could be integrated into PG-Schema.

Range constraints. Some schema languages allow for range constraints (RC). The syntax of PG-Schema can be thus extended, specifying restrictions on acceptable values for properties. For instance, the following example defines a node type Book, with properties title (a string with maximum 100 characters), genre (an enumeration), and isbn (a string conforming to a regular expression):

```
(bookType: Book {
  title STRING(100),
  genre ENUM("Prose", "Poetry", "Dramatic"),
  isbn STRING ^(?=(?:\D*\d){10}(?:(?:\D*\d){3})?$)[\d-]+$})
```

The restrictions allowed in the example above can be based on XSD facets [42], with additional features, such as enumerations, implemented similarly.

Complex datatypes. We have only included primitive datatypes in PG-Schema. Looking at the CPT column in Table 1 we see that complex property values are widely supported in other formalisms. Our syntax can easily be extended to support collections. For instance, if we wish to specify that a `name` is an array of strings, we could write `name STRING ARRAY {1,2}`. The (optional) annotation in curly braces specifies the minimum and the maximum number of elements in the array.

Intersections and unions for content types. In Section 4, we assumed that union and intersection can be used in element types. Such combinations can also be introduced into properties (see UIT in Table 1). We can for example allow the *union* (|) and *intersection* (&) operators from label expressions also between property types and between content types. The following example allows the `name` to be broken down into a `givenName` and a `familyName`:

```
(personType: Person
  ( {name STRING} | {givenName STRING, familyName STRING} )
  & {height (INT | FLOAT)})
```

Note that we use round brackets to group content definitions.

Advanced cardinalities. The notation of PG-Keys can be readily extended to specify cardinality constraints by taking the general form `FOR` $p(x)$ `<qualifier>` $q(x, \bar{y})$, and allowing for `<qualifier>` an expression of the form `COUNT <lower bound>?..<upper bound>? OF`, expressing that the number of distinct results returned by $q(x, \bar{y})$ must be within that range. If the upper bound and lower bound are identical, we allow the short-hand `COUNT <bound> OF`.

The constraint stating that each department has at least two employees working for it, could be written as

```
  FOR (d: Department)
    COUNT 2.. OF e WITHIN (e: Employee)-[:worksIn]->(d).
```

And if employees can work on at most 3 projects, this could be written as

```
  FOR (e: Employee)
    COUNT 0..3 OF p WITHIN (e)-[: worksOn]->(p: Project).
```

This notation also allows us to express disjointness and denial constraints without using negation in patterns. For example, if reptiles cannot be amphibians, we can write this as

```
  FOR (a: Amphibian)
    COUNT 0 OF (a: Reptile).
```

As discussed in [8], such constraints are relevant for many practical use cases and can be efficiently evaluated.

6 SUMMARY AND LOOKING AHEAD

PG-Schema is the first unifying schema language for property graphs, which serves as a recommendation for future versions of GQL. This work is the result of academia and industry collaborating to bridge gaps and accelerate standardization efforts that benefit both communities at large.

Summary. PG-Schema is a schema language that caters to basic needs such as defining node and edge types, as well as advanced scenarios such as expressing complex type hierarchies and integrity constraints. It has been designed to support both descriptive and prescriptive roles, with a focus on enabling agile evolution, flexible validation, and usability. The language comes with an ASCII-art, yet formal, syntax and well-defined semantics. The core of PG-Schema centers around the rich PG-Types type system, with desirable features such as compositionality, abstract types, type hierarchies, and multi-inheritance, as well as around PG-Keys, which allows the expression of complex key and participation constraints. The language thus supports a wide-range of capabilities, largely absent from the state-of-the-art schema languages and systems we have reviewed. Finally, PG-Schema is easily extensible with further features, such as range constraints, complex data types, content type combinators, and advanced cardinalities.

Looking Ahead. In addition to the impact on standardization efforts, this is an opportunity for graph database vendors to increase functionality and support current and future customer demands. Our work also provides a basis for future research by the academic community. Finally, this successful high-impact academia-industry collaboration model is one we hope will be replicated by other communities at large, in data management and beyond.

ACKNOWLEDGMENTS

We thank all LDBC PGSWG members for the discussions around property graph schemas. Renzo Angles was supported by ANID Millennium Science Initiative Program, Code ICN17_002 and ANID FONDECYT Chile through grant 1221727. Angela Bonifati and Leonid Libkin were supported by ANR-21-CE48-0015 VeriGraph. Leonid Libkin was supported by a Leverhulme Trust Research fellowship and EPSRC grant S003800. Wim Martens was supported by ANR project EQUUS ANR-19-CE48-0019; funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 431183758. Filip Murlak was supported by NCN grant 2018/30/E/ST6/00042. Domagoj Vrgoč was supported by ANID Millennium Science Initiative Program, Code ICN17_002 and ANID Fondecyt Regular project 1221799. For the purposes of open access, the authors have applied a CC BY public copyright licence to any Author Accepted Manuscript version arising from this submission.

REFERENCES

[1] ISO/IEC 39075. 2023. *Information technology — Database languages — GQL.* Standard. International Organization for Standardization, Geneva, CH.

[2] ISO/IEC 9075-16. 2022. *Information technology — Database languages SQL — Part 16: Property Graph Queries (SQL/PGQ).* Standard. International Organization for Standardization, Geneva, CH.

[3] AgensGraph. 2022. AgensGraph. https://bitnine.net/agensgraph (visited: 2022-11).

[4] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. 2018. G-CORE: A Core for Future Graph Query Languages. In *SIGMOD Conference.* ACM, 1421–1432.

[5] 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 Savkovic, Michael Schmidt, Juan F. Sequeda, Slawek Staworko, and Dominik Tomaszuk. 2021. PG-Keys: Keys for Property Graphs. In *International Conference on Management of Data (SIGMOD).* ACM, 2423–2436.

[6] ArangoDB. 2022. ArangoDB. https://www.arangodb.com/ (visited: 2022-11).

[7] Thomas Baker and Eric Prud'hommeaux. 2019. *Shape Expressions (ShEx) 2.1 Primer.* W3C Community Group Final Report. W3C. https://shex.io/shex-primer/index.html.

[8] Angela Bonifati, Stefania Dumbrava, George Fletcher, Jan Hidders, Matthias Hofer, Wim Martens, Filip Murlak, Joshua Shinavier, Sławek Staworko, and Dominik Tomaszuk. 2022. Threshold Queries in Theory and in the Wild. *Proc. VLDB Endow.* 15, 5 (may 2022), 1105–1118. https://doi.org/10.14778/3510397.3510407

[9] Angela Bonifati, Stefania Dumbrava, George Fletcher, Jan Hidders, Bei Li, Leonid Libkin, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen Savković, Juan Sequeda, Sławek Staworko, Dominik Tomaszuk, Hannes Voigt, Domagoj Vrgoč, and Mingxi Wu. 2022. domel/pgschema: PG-Schema Grammar 0.3. (Nov 2022). https://doi.org/10.5281/zenodo.7362078 https://zenodo.org/record/7362078.

[10] Angela Bonifati, Stefania Dumbrava, George Fletcher, Jan Hidders, Bei Li, Leonid Libkin, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen Savković, Juan Sequeda, Sławek Staworko, Dominik Tomaszuk, Hannes Voigt, Domagoj Vrgoč, and Mingxi Wu. 2022. PG-Schema: Schemas for Property Graphs. https://doi.org/10.48550/arXiv.2211.10962

[11] Angela Bonifati, Stefania-Gabriela Dumbrava, Emile Martinez, Fatemeh Ghasemi, Malo Jaffré, Pacome Luton, and Thomas Pickles. 2022. DiscoPG: Property Graph Schema Discovery and Exploration. *Proc. VLDB Endow.* 15, 12 (2022), 3654–3657. https://www.vldb.org/pvldb/vol15/p3654-bonifati.pdf

[12] Angela Bonifati, Peter Furniss, Alastair Green, Russ Harmer, Eugenia Oshurko, and Hannes Voigt. 2019. Schema Validation and Evolution for Graph Databases. In *ER (Lecture Notes in Computer Science, Vol. 11788).* Springer, 448–456.

[13] Gilad Bracha and William R. Cook. 1990. Mixin-based Inheritance. In *Conference on Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming, OOPSLA/ECOOP 1990, Ottawa, Canada, October 21-25, 1990, Proceedings*, Akinori Yonezawa (Ed.). ACM, 303–311. https://doi.org/10.1145/

97945.97982

[14] Dan Brickley and Ramanathan Guha. 2014. *RDF Schema 1.1.* W3C Recommendation. W3C. https://www.w3.org/TR/2014/REC-rdf-schema-20140225/.

[15] Peter P. Chen. 1976. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Trans. Database Syst.* 1, 1 (1976), 9–36.

[16] DataStax. 2022. DataStax. https://www.datastax.com/ (visited: 2022-11).

[17] 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, Oskar van Rest, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 2246–2258.

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

[19] Facebook. 2018. GraphQL. https://spec.graphql.org/June2018/.

[20] Martin Fowler. 2003. *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (3 ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[21] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *SIGMOD Conference*. ACM, 1433–1445.

[22] D.K. Gosnell and M. Broecheler. 2022. The Practitioner's Guide to Graph Data. https://gra.fo/faq/ (visited: 2022-11).

[23] Gra.fo. 2022. Gra.fo. https://gra.fo/faq/ (visited: 2022-11).

[24] TigerGraph GraphStudioTM. 2022. TigerGraph GraphStudioTM. https://docs.tigergraph.com/gui/current/graphstudio/overview (visited: 2022-11).

[25] LDBC Property Graph Schema Working Group. 2020. *LDBC Property Graph Schema contributions to WG3.* Open Access to External Paper OAEP-2023-04. Linked Data Benchmark Council (LDBC). https://doi.org/10.54285/ldbc.OFJF3566 Edited and presented by Jan Hidders, George Fletcher and Bei Li.

[26] Neo4j SQL Working Group, Peter Furniss, and Alastair Green. 2018. *SQL/PGQ data model and graph schema.* Open Access to External Paper OAEP-2023-01. Linked Data Benchmark Council (LDBC). https://doi.org/10.54285/ldbc.QZSK3559

[27] Benoît Groz, Aurélien Lemay, Slawek Staworko, and Piotr Wieczorek. 2022. Inference of Shape Graphs for Graph Databases. In *25th International Conference on Database Theory, ICDT 2022, March 29 to April 1, 2022, Edinburgh, UK (Virtual Conference) (LIPIcs, Vol. 220)*, Dan Olteanu and Nils Vortmeier (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 14:1–14:20. https://doi.org/10.4230/LIPIcs.ICDT.2022.14

[28] Terry Halpin. 2015. *Object-Role Modeling Fundamentals: A Practical Guide to Data Modeling with ORM.* Technics Publications.

[29] Olaf Hartig and Jorge Pérez. 2018. Semantics and complexity of GraphQL. In *Proceedings of the 2018 World Wide Web Conference.* 1155–1164.

[30] Pascal Hitzler, Sebastian Rudolph, Markus Krötzsch, Peter Patel-Schneider, and Bijan Parsia. 2012. *OWL 2 Web Ontology Language Primer (Second Edition).* W3C Recommendation. W3C. https://www.w3.org/TR/2012/REC-owl2-primer-20121211/.

[31] ISO/IEC 19757-2:2008 2008. *Information technology — Document Schema Definition Language (DSDL) — Part 2: Regular-grammar-based validation — RELAX NG.* Standard. International Organization for Standardization, Geneva, CH.

[32] JanusGraph. 2022. JanusGraph. https://janusgraph.org/ (visited: 2022-11).

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

[34] Mark Needham and Amy E. Hodler. 2019. *Graph Algorithms.* O'Relly Media.

[35] Neo4j 2016. *The Definitive Guide to Graph Databases for the RDBMS Developer.* Neo4j.

[36] Neo4j. 2019. Graph DDL (Data Definition Language). https://github.com/opencypher/morpheus/blob/master/documentation/asciidoc/backend-sql-graphddl.adoc (visited: 2023-04).

[37] Neo4j. 2022. Neo4j. https://neo4j.com/ (visited: 2022-11).

[38] Neo4j. 2022. Neo4j Browser. https://neo4j.com/product/developer-tools/#browser (visited: 2022-11).

[39] Graph Notebooks. 2022. Graph Notebooks. https://github.com/aws/graph-notebook (visited: 2022-11).

[40] Oracle. 2022. Oracle Spatial and Graph. https://www.oracle.com/database/technologies/spatialandgraph.html (visited: 2022-11).

[41] OrientDB. 2022. OrientDB. https://orientdb.org/ (visited: 2022-11).

[42] David Peterson, Sandy Gao, Paul V. Biron, Michael Sperberg-McQueen, Ashok Malhotra, and Henry Thompson. 2012. *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes.* W3C Recommendation. W3C. https://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/.

[43] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. *Graph Databases.* O'Reilly Media.

[44] Mats Rydberg. 2016. *Cypher schema constraints proposal.* Open Access to External Paper OAEP-2023-03. Linked Data Benchmark Council (LDBC). https://doi.org/10.54285/ldbc.KKHM1756

[45] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *VLDB J.* 29, 2-3 (2020), 595–618.

[46] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *The VLDB Journal* 29, 2 (2020), 595–618. https://doi.org/10.1007/s00778-019-00548-x

[47] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid G. Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. 2021. The future is big graphs: a community view on graph processing systems. *Commun. ACM* 64, 9 (2021), 62–71.

[48] Tom Sawyer. 2022. Graph Database Browser. https://www.tomsawyer.com/graph-database-browser (visited: 2022-11).

[49] Michael Sperberg-McQueen, Henry Thompson, David Beech, Murray Maloney, Noah Mendelsohn, and Sandy Gao. 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/.

[50] Slawomir Staworko, Iovka Boneva, José Emilio Labra Gayo, Samuel Hym, Eric Gordon Prud'Hommeaux, and Harold Solbrig. 2015. Complexity and Expressiveness of ShEx for RDF. In *18th International Conference on Database Theory (ICDT 2015).*

[51] Neo4j Query Languages Standards & Research Team. 2019. *Introduction to GQL Schema design.* Open Access to External Paper OAEP-2023-02. Linked Data Benchmark Council (LDBC). https://doi.org/10.54285/ldbc.EPWQ6741 Edited by Alastair Green and Hannes Voigt.

[52] Sparsity Technologies. 2022. Sparksee. https://sparsity-technologies.com/#sparksee (visited: 2022-11).

[53] Bernhard Thalheim. 2018. Extended Entity-Relationship Model. In *Encyclopedia of Database Systems, Second Edition*, Ling Liu and M. Tamer Özsu (Eds.). Springer. https://doi.org/10.1007/978-1-4614-8265-9_157

[54] TigerGraph. 2022. TigerGraph. https://www.tigergraph.com/ (visited: 2022-11).

[55] Vaticle. 2022. TypeDB. https://vaticle.com/ (visited: 2022-11).

[56] Damian Wileński and Dominik Tomaszuk. 2022. damianw27/pgs-grammar-check: Version 1.0.0. (Nov 2022). https://doi.org/10.5281/zenodo.7344227 Available at https://damianw27.github.io/pgs-grammar-check/.

[57] Austin Wright, Henry Andrews, Ben Hutton, and Greg Dennis. 2020. *JSON Schema: A Media Type for Describing JSON Documents.* Draft. Internet Engineering Task Force.

[58] Min Wu, Xinglu Yi, Hui Yu, Yu Liu, and Yujue Wang. 2022. Nebula Graph: An open source distributed graph database. *CoRR* abs/2206.07278 (2022).

[59] François Yergeau, Michael Sperberg-McQueen, Tim Bray, Jean Paoli, and Eve Maler. 2008. *Extensible Markup Language (XML) 1.0 (Fifth Edition).* W3C Recommendation. W3C. https://www.w3.org/TR/2008/REC-xml-20081126/.

Received November 2022; revised February 2023; accepted March 2023